Dev mobile 103 : Les formulaires et Redux

Prérequis

Après avoir suivi la Dev mobile 102, il vous faudra aussi rattraper l'état de l'application sur le gitlab : https://gitlab.ginfo.centrale-marseille.fr/ginfo/formations/appli_foder

Objectifs

Dans cette formation, nous allons ajouter un formulaire, pour pouvoir laisser un commentaire sur une session de formation via l'application. Ensuite, nous ajouterons une mémoire à notre application : elle retiendra quels commentaires ont été laissés par l'utilisateur, pour les afficher différemment.

J'ai bien dit objectif, parce que je commence à préparer la formation 24h avant la première session. Donc on va essayer quoi.

Les formulaires

Pour ajouter notre formulaire, on va commencer par créer un component. On crée un fichier `CommentForm.js` et on y mets le code suivant :

```
import React from 'react';
import {View} from "react-native";

export default function commentForm (props) {

    return (
        <View style={{backgroundColor:'red', flex:1}}/>
    )
}
```

Quoi ? Comment ? On crée un component mais c'est pas une classe ? On fait une fonction ?
Oui, on peut créer des components fonction. Ici, on doit, parce qu'on va utiliser un concept assez compliqué, les Hook, et ce n'est possible que dans un component fonction.

On intègre ce component dans notre `CommentList`, entre le header et la flatlist.

On va installer la librairie nécessaire pour les formulaires :

```
  npm install react-hook-form
```

Et quelques librairies afin d'ajouter des icones Font Awesome à notre appli :

```
  npm install react-native-svg fortawesome
```

On ajoute les imports :

```
import {Controller, useForm} from 'react-hook-form';
import {FontAwesomeIcon} from "@fortawesome/react-native-fontawesome";
import {faCheck} from "@fortawesome/free-solid-svg-icons";
```

Dans la fonction, on appelle le hook `useForm`. Pour faire simple : c'est lui qui va gérer notre formulaire. On défini aussi une fonction `onSubmit`, qui sera appelée lors de la validation du commentaire. C'est ici que l'on enverra le commentaire à l'API.

```
const {control, handleSubmit, formState: {errors}} = useForm();

const onSubmit = data => {}
```

Ensuite on intègre dans notre component une `TextInput` gérée par le formulaire, et une `TouchableOpacity` pour soumettre le commentaire.

```
<View style={styles.container}>
    <Controller
        control={control}
        render={({ field: { onChange, onBlur, value } }) => (
            <TextInput
                style={styles.input}
                onBlur={onBlur}
                onChangeText={onChange}
                 value={value}
                placeholder={"Laisser un commentaire"}
                placeholderTextColor={'lightgrey'}
            />
        )}
        name="contenu"
        rules={{ required: true }}
        defaultValue=""
    />
    <TouchableOpacity
        style={styles.button}
        onPress={handleSubmit(onSubmit)}
    >
        <FontAwesomeIcon icon={faCheck} color={'white'} size={30}/>
    </TouchableOpacity>
</View>
```

En mettant les bon styles, on obtient ça :

l'image

Le code en entier :

```
import React from 'react';
import {View, TouchableOpacity, TextInput, StyleSheet} from "react-native";
import {Controller, useForm} from 'react-hook-form';
import {FontAwesomeIcon} from "@fortawesome/react-native-fontawesome";
import {faCheck} from "@fortawesome/free-solid-svg-icons";

export default function commentForm (props) {

    const {control, handleSubmit, formState: {errors}} = useForm();

    const onSubmit = data => {}

    return (
        <View style={styles.container}>
            <Controller
                control={control}
                render={({ field: { onChange, onBlur, value } }) => (
                    <TextInput
                        style={styles.input}
                        onBlur={onBlur}
                        onChangeText={onChange}
                        value={value}
                        placeholder={"Laisser un commentaire"}
                        placeholderTextColor={'lightgrey'}
                    />
                )}
                name="contenu"
                rules={{ required: true }}
                defaultValue=""
            />
            <TouchableOpacity
                style={styles.button}
                onPress={handleSubmit(onSubmit)}
            >
                <FontAwesomeIcon icon={faCheck} color={'white'} size={30}/>
            </TouchableOpacity>
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        flexDirection: 'row',
        marginBottom: 10,
    },
    input: {
        flex: 4,
        justifyContent: 'center',
        backgroundColor: 'white',
        borderRadius: 10,
```

```
        marginLeft: 10,
        paddingLeft: 5,
        marginRight: 5,
    },
    button: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
        backgroundColor: '#42b649',
        borderRadius: 10,
        marginRight: 10,
    }
});
```

Tout va maintenant se passer au niveau de la fonction `onSubmit`. On peut commencer par ajouter un

```
console.log(data)
```

Quand on écris un commentaire et qu'on valide, on voit apparaître dans la console un dictionnaire avec le contenu du commentaire.

Remarque : si on n'écris rien, la fonction `onsubmit` n'est pas appelée ! C'est grâce au `rules={ { required : true }}`, on ne veut pas envoyer à l'API un commentaire vide.

Maintenant, on vet envoyer ce commentaire à l'API https://feedback-forma.ginfo.centrale-marseille.fr/. La route est : `/commenter/{id}`. Il faut donc passer l'id de la session au component, via ses props.

Pour la requête, attention ! C'est plus compliqué que ce qu'on a fait jusqu'à maintenant. En effet, l'API n'accepte que des requêtes de type `POST` sur cette route. Il faut passer le commentaire en paramètre de la requête. Ici, on utilise `fetch` pour envoyer la requête, mais on n'a pas besoin de la réponse. On va simplement aficher le status de la réponse.

```
const onSubmit = data => {
    let jsonD = new FormData()
    jsonD.append("json", JSON.stringify(data))
    let request = new Request(
        'https://feedback-forma.ginfo.centrale-marseille.fr/commenter/' +
props.session.id,
        {method:'POST', body:jsonD}
    )
    fetch(request).then((response) => {
        console.log(response.status)
    })
}
```

On a réussi à envoyer un commentaire !

Mais pour le voir apparaître dans l'application, il faut recharger la page. On peut faire retour et revenir sur la session, mais évidemment on veut pas laisser ça dans notre application.

Donc crée une méthode de `CommentList`, qui mets `isLoading = true` et appelle `forceUpdate`, puis on la passe dans les props du component `CommentForm`, qu'on appelle après le `console.log`. Et voilà, quand on valide le commentaire, la page de chargement s'affiche, puis on retrouve notre nouveau commentaire tout en bas.

Redux

Rom a dit : “Redux c'est super compliqué, c'est pas possible que tu comprennes comment ça marche, même moi je comprends pas. Tu fait que copier les codes d'exemple en vrai”

Il a peut-être pas si tort, mais bon.

Du coup, on veut stocker des informations dans notre application, et qu'elles soient conservées même quand on change de page ou quoi (c'est à dire qu'elles ne soient pas conservées au niveau du component, mais dans un `state` global).

Là c'est un peu technique : on va configurer un store, en utilisant des reducers. Les reducers, on les écrit, c'est là qu'on va gérer les données enregistrées, définir les modifications qu'on peut y faire et tout.

On crée un dossier `Store`, avec un dossier `Reducer` dedans. On crée un fichier `commentsReducer.js`, avec le contenu suivant :

```
const initialState = {
    comments:[]
}

function handleComments(state=initialState, action){
    switch (action.type) {
        case 'ADD_COMMENT':
            let comments = state.comments
            comments.push(action.commentId)
            return {
                ...state,
                comments: comments,
            }
        default:
            return state
    }
}

export default handleComments;
```

Ça, c'est le `Reducer`. On va ensuite créer un fichier `configureStore.js`. Dedans :

```
import { createStore } from 'redux';
import handleComments from "./Reducers/commentsReducer";

export default createStore(handleComments);
```

Assez simple non ? On installe `react-redux` :

```
  npm install react-redux
```

Dans le `App.js`, on importe ce `Store` et on entoure toute notre appli par un `Provider`:

```
import {Provider} from "react-redux";
import Store from './Store/configureStore';


<Provider store={Store}>
```

Dans le `CommentForm`, il faut importer les `Hooks` nécessaires : `useSelector` pour d'abonner au `State` global, `useDispatch` pour pouvoir envoyer des actions au `Reducer`.

```
import {useSelector, useDispatch} from "react-redux";

const dispatch = useDispatch();
const state = useSelector(state => state);
```

Ensuite, dans les `then` qui suivent la requête, on va créer une action et la dispatcher :

```
let action = {
    type: 'ADD_COMMENT',
    commentId: data.comentId
}
dispatch(action)
```

Si on log le state, on voit qu'un identifiant s'ajoute dans la liste lorsqu'on ajoute un commentaire.

On va maintenant connecter notre `CommentList` au `Store`, pour afficher les commentaires publiés par l'utilisateur d'une façon différente.

On importe `connect` de `react-redux`, on défini une fonction `mapStateToProps`, et on utilise la fonction `connect` au moment de l'export. Maintenant, la liste des commentaires est accessible via `this.props.comments`

```
import {connect} from "react-redux";

const mapStateToProps = (state) => {
    return state
}

export default connect(mapStateToProps)(CommentList);
```

Maintenant, on va passer à notre `CommentItem` une prop `isMine` (un booléen), et faire en sorte qu'il affiche nos commentaires différement (on change la couleur de fond par exemple)

```
<CommentItem comment={item}
isMine={this.props.comments.includes(item.id.toString())}/>
```

Voilà on a une belle appli mobile qui marche !