Redux

Tache 1

Mettre à jour les dependances : **npm i**

Installer **redux : npm i -s redux react-redux @reduxjs/toolkit @redux-devtools/extension**

Installer **redux devtools**, c'est une extension de chrome

Installer json serveur **: npm i -g json-server**

Lancer le serveur sur le **localhost 3000 : npm run server**

Note : si vous regarder dans le fichier **package.json**, vous allez voir à quoi renvoie server

Ouvrez un autre terminal et lancer l'application **: npm start**

Tache 2

Allez dans src -→ components et la vous aurez la liste de nos composants :

App.js = tout notre site

PostForm.js =

User.js =

Utils.js = une petites fonction que l'on utilisera tout au long du projet

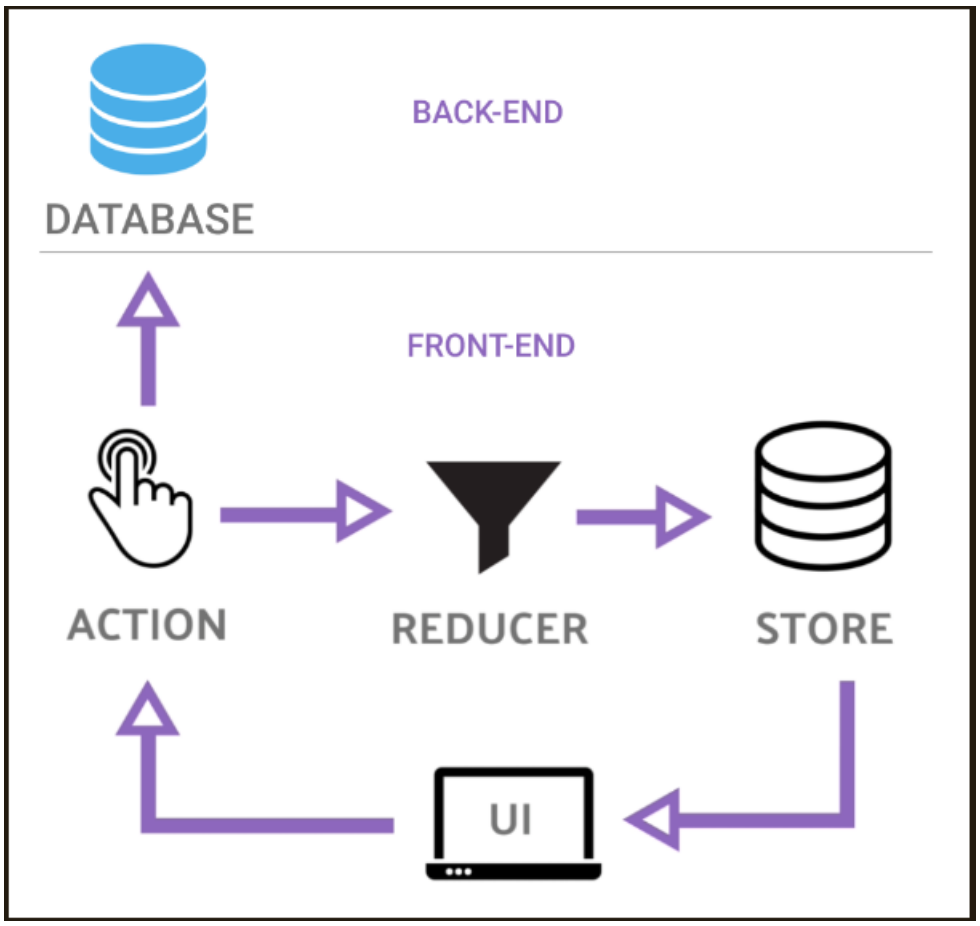
Post.js = sera nos post qui seront au niveau de la section CONTENU

Tache 3 :

Principe de redux

Le UI (user interface) qui est l'utilisateur fait une action, par exemple un like = il cree une action en redux.

Ceci nous permet de communiquer a la fois avec la base de données et le store. Donc le reducer calcule le nombre de like, ajoute dans le store et le UI est directment mis a jour sans attendre la reponse de la BD (voir image)



Tache 4

Pour configurer redux, voici la demarche a suivre :

Créer deux dossiers dans src : reducers et actions

Dans le fichier index.js, on fait cette configuration :

```
// redux
import { Provider } from "react-redux";
import { configureStore } from "@reduxjs/toolkit";
import rootReducer from "./reducers";
const store = configureStore({
  reducer: rootReducer,
  devTools: true,
});

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
```

```
  document.getElementById("root")
);
```

- On entoure notre App.js du Provider en specifiant le store
- On cree le store par la suite : en specifiant le reducer et devTools
- Ensuite on importe le reducers du dossier reducers : pas besoin de specfier le nom du fichier car lorsqu'on ne specifie pas par defaut il lit dans index.js

Dans le dossier reducers, créer un fichier index.js et y mettre ceci :

```
import { combineReducers } from "redux";

export default combineReducers({
  // reducers
});
```

Note :

Il peut avoir que vous avez cette erreur :

ERROR in ./node_modules/react-redux/dist/react-redux.mjs 873:27-53

export 'useSyncExternalStore' (imported as 'React') was not found in 'react' (possible exports: Children, Component, Fragment, Profiler, PureComponent, StrictMode, Suspense, __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED, cloneElement, createContext, createElement, createFactory, createRef, forwardRef, isValidElement, lazy, memo, useCallback, useContext, useDebugValue, useEffect, useImperativeHandle, useLayoutEffect, useMemo, useReducer, useRef, useState, version)

Ceci est un probleme de compatibilté avec la version de react qui est inferieure a la version de react-redux. Comment corriger cela :

npm list react

npm install react@^18 react-dom@^18: remplacer 18 par la version reclamé dans la premiere commande

Tache 5 : création de notre premier

Dans le dossier reducers, creons le fichier user.reducer.js et dans ce fichier, mettons ceci :

```
const initialState = { user: "EvolveD, Tene Ribot" };

export default function userReducer(state = initialState, action) {
  //switch
  return state;
}
```

Dans le fichier index.js qui se trouve dans reducers, appelez le reducers que l'on vient de créer :

```
export default combineReducers({
  // reducers
  userReducer,
});
```

Vérifier notre store dans devtools qu niveau du state

Créons un 02 reducers qui sera post.reducer.js :

```
const initialState = {};

export default function postReducer(state = initialState, action) {
  //switch
  return state;
}
```

Controllons ces reducers dans notre state de devtools

Tache 6 : creation des actions

Quand on a fait npm run serveur, si vous etiez vigilant, vous aurez vu ceci :

http://localhost:3000/posts

http://localhost:3000/user

Dans notre **bd.json** (une base de données nosql)

Donc une action c'est le composant qui dialogue avec la base de données et dit au reducer ce qu'il doit faire.

Dans le dossier actions, créer un fichier post.action.js

Apres avoir fait cela, creons l'action en question :

```
import axios from "axios";

export const GET_POSTS = "GET_POSTS";

export const getPosts = () => {
  return (dispatch) => {
    return axios.get("http://localhost:3000/posts").then((res) => {
      console.log(res.data);
    });
  };
```

```
};
```

Essayons de voir si ça marche. Dans le index.js de src, appellons cette action juste apres le store:

```
store.dispatch(getPosts());
```

dispatch permet de jouer la fonction getPosts().

Regardez dans la console. Si tout a fonctionné, vous avez les données en console

Modifions maintenant notre action :

```
import axios from "axios";

export const GET_POSTS = "GET_POSTS";

export const getPosts = () => {
  return (dispatch) => {
    return axios.get("http://localhost:3000/posts").then((res) => {
      dispatch({ type: GET_POSTS, payload: res.data });
    });
  };
};
```

En effet :

- On communique d'abord avec la base de données : recupere les données;
- Ensuite on communique avec le reducer

Allons maintenant dans le post.reducer.js et modifions le :

```
import { GET_POSTS } from "../actions/post.action";

const initialState = {};

export default function postReducer(state = initialState, action) {
  //switch
  // return state;
  switch (action.type) {
    case GET_POSTS:
      return action.payload;
    default:
      return state;
  }
}
```

Vérifions maintenant le devtools et on verra qu'on a toutes nos données là-bas

Une fois qu'on a nos données dans le store, on peut maintenant l'afficher dans notre dom :

Ouvrir **App.js** et utilisons useSelector pour recuperer les donner de postReducer :

```
const posts = useSelector((state) => state.postReducer);
  console.log(posts);
```

dans le div.post-container mettre ceci :

```
<div className="post-container">
          {posts.map((post, index) => (
            <Post post={post} key={index} />
          ))}
        </div>
```

On a un petit problème : si on actualise, on aura une page vide. Donc pour se prémunir de ce comportement, on doit utiliser notre composant Utils.js qui contient une fonction **isEmpty** qui teste si la data est vide :

```
 <div className="post-container">
          {!isEmpty(posts) &&
            posts.map((post, index) => <Post post={post} key={index} />)}
        </div>
```

Donc avec ça, on ne lance plus la map tant qu'on a pas de données.

Créons maintenant l'action du user :

- Créons le fichier user.action.js
- Copie et colle le contenu de post.action.js dans user.action.js et change partout ou on a post en user
- Jouons cette fonction dans index.js de src avec dispatch
- On modifie aussi le user.reducer.js
- Ensuite on va dans User.js pour récupérer les donner de userReducer avec useSelector et on modifie les valeurs de ce composant avec les valeurs de la bd:

```
  const user = useSelector((state) => state.userReducer);
  // console.log(user);

  return (
    <div className="user-container">
      <div className="user">
        <h3>{!isEmpty(user) && user.pseudo}</h3>
```

```
        <img src="./img/bill-gates.png" alt="bill gates" />
        <p>Age : {!isEmpty(user) && user.age}</p>
        <p>
          Like {!isEmpty(user) && user.like > 1 ? "s" : ""} :{" "}
          {!isEmpty(user) && user.likes}
        </p>
      </div>
    </div>
  );
```

Maintenant, on va faire en sorte que seulement l'utilisateur qui a créer l'article puisse l'éditer. Pour cela :

- Dans Post.js, on recupere les données du reducer :

```
const user = useSelector((state) => state.userReducer);
```

- Ensuite, on conditionne la partie du delete et edite :

```
      {!isEmpty(user) && user.pseudo === post.author && (
        <div className="edit-delete">
          <img
            src="./icons/edit.svg"
            alt="edit"
            onClick={() => setEditToggle(!editToggle)}
          />
          <img src="./icons/delete.svg" alt="delete" />
        </div>
      )}
```

Tache 7 : ajout d'un message

Dans le composant PostForm :

- Créer une variable en utilsant useRef() qui identifie le formulaire
- Créer une fonction async handleForm et le premunir du comportement par defaut lors de la soumission d'un formulaire
- Mettre l'evenement onSubmit qui declenche la fonction handleForm lors de la soumission du formulaire

```
const PostForm = () => {
  const form = useRef();
  const user = useSelector((state) => state.userReducer);
```

```
  const handleForm = async (e) => {
    e.preventDefault();
    console.log(form);
  };
  return (
    <div className="form-container">
      <form ref={form} onSubmit={(e) => handleForm(e)}>
        <input type="text" placeholder="Titre du poste" />
        <textarea placeholder="Postez vos pensées..."></textarea>
        <input type="submit" value="Envoyer" />
      </form>
    </div>
  );
};
```

- Aller dans la console,
- Écrivez quelque chose dans le formulaire et soumettez

Ensuite on modifie la fonction handleForm comme suit :

```
  const handleForm = async (e) => {
    e.preventDefault();
    // console.log(form);

    const postData = {
      author: user.pseudo,
      title: form.current[0].value,
      content: form.current[1].value,
      likess: 0,
    };

  };
```

Ensuite on peut faire l'action :

- Dans notre PostForm.js, on va créer une constante de type dispatch qui va permettre de stocker les données recuperer dans le store par useSelector :

```
const PostForm = () => {
  const form = useRef();
  const user = useSelector((state) => state.userReducer);
  const dispatch = useDispatch();

  const handleForm = async (e) => {
    e.preventDefault();
```

```
    // console.log(form);

    const postData = {
      author: user.pseudo,
      title: form.current[0].value,
      content: form.current[1].value,
      likess: 0,
    };

    dispatch(addPost(postData));
  };
```

Ensuite dans notre post.action.js, on va créer la méthode permettant de faire un post dans notre base de données :

- Copier et coller le get pour que l'on modifie comme suit :

```
import axios from "axios";

export const GET_POSTS = "GET_POSTS";
export const ADD_POST = "ADD_POST";

export const getPosts = () => {
  return (dispatch) => {
    return axios.get("http://localhost:3000/posts").then((res) => {
      dispatch({ type: GET_POSTS, payload: res.data });
    });
  };
};

export const addPost = (data) => {
  return (dispatch) => {
    return axios.post("http://localhost:3000/posts", data).then((res) => {
      dispatch({ type: ADD_POST, payload: res.data });
    });
  };
};
```

- Une fois ceci fait, repartir dans le PostForm.js effacer la ligne l'appel de **addPost** et réécrivez encore afin de faire l'importation
- Faire un test et regarder votre **bd.json**

Ajouter cette ligne dans fonction **handleForm()** qui va permettre de réinitialiser le formulaire après l'envoie :

```
form.current.reset();
```

Maintenant on va modifier le switch du reducer afin qu'il puisse changer sont state et y ajouter les nouveaux posts dans le dom :

```
  switch (action.type) {
    case GET_POSTS:
      return action.payload;
    case ADD_POST:
      return [action.payload, ...state];
    default:
      return state;
  }
```

Pourquoi l'asynchrone :

En effet il faut que l'on se rassure que dans le store lors de l'Edition d'un post, que celui-ci ait son id car s'il n'a pas d'id, il ne pourra pas mettre a jour la base de données. N'oubliez pas que tout ce qui se joue coté store n'est pas forcement en concordance avec la bd.

Donc petite modification : mettre un await sur l'appelle du dispatch et rappeler encore la bd avec la méthode getPosts()

```
  const handleForm = async (e) => {
    e.preventDefault();
    // console.log(form);

    const postData = {
      author: user.pseudo,
      title: form.current[0].value,
      content: form.current[1].value,
      likess: 0,
    };

    await dispatch(addPost(postData));
    dispatch(getPosts());
    form.current.reset();
  };
```

Tache 8 : éditer un post

On va dans le fichier Post.js :

- Dans **l'editToggle,** préciser la valeur de **onSubmit** et de **Onchange** :

```
{editToggle ? (
        <form onSubmit={(e) => handleEdit(e)}>
          <textarea
            autoFocus={true}
            defaultValue={post.content}
            onChange={(e) => setEditContent(e.target.value)}
          ></textarea>
          <input type="submit" value="Valider modification" />
        </form>
      ) : (
        <p>{post.content}</p>
      )}
```

- On crée le hooks pour la modification du contenu d'un post, on définit le **useDispatch()** et la fonction **handleEdit** :

```
  const [editContent, setEditContent] = useState(post.content);
  const dispatch = useDispatch();

  const handleEdit = (e) => {
    e.preventDefault();

    const postData = {
      title: post.title,
      author: user.pseudo,
      likes: post.likes,
      id: post.id,
      content: editContent,
    };
    dispatch(editPost(postData));
    setEditToggle(false);
  };
```

- Ensuite, on va dans post.action.js pour écrire la fonction édit

```
export const editPost = (data) => {
  return (dispatch) => {
    return axios
```

```
      .put(`http://localhost:3000/posts/${data.id}`, data)
      .then((res) => {
        dispatch({ type: EDIT_POST, payload: res.data });
      });
  };
};
```

Et dans le post.reducer.js, on écrit la logique du switch édit :

```
case EDIT_POST:
      return state.map((post) => {
        if (post.id === action.payload.id) {
          return {
            ...post,
            content: action.payload.content,
          };
        } else return post;
      });
```

Tache 9 : delete

On va rester dans la même logique :

- On écrit l'action **deletepost** :

```
export const deletePost = (postId) => {
  return (dispatch) => {
    return
axios.delete(`http://localhost:3000/posts/${postId}`).then((res) => {
      dispatch({ type: DELETE_POST, payload: postId });
    });
  };
};
```

- Dans le composant Post.js

On appelle le **disptatch** :

```
          <img
            src="./icons/delete.svg"
            alt="delete"
            onClick={() => dispatch(deletePost(post.id))}
          />
```

- Ensuite dans le reducer, on ecrit la logique :

```
case DELETE_POST:
      return state.filter((post) => post.id !== action.payload);
```

Tache 10: add likes

Nous allons aller sur la même logique :

- Dans le composant Like.js

```
const Like = ({ post }) => {
  const dispatch = useDispatch();
  const handleLike = () => {
    const postData = {
      title: post.title,
      author: post.author,
      content: post.content,
      id: post.id,
      likes: post.likes + 1,
    };
    dispatch(addPostLike(postData));
  };
```

```
<img
        onClick={() => handleLike()}
        src="./icons/clap.png"
        className="clap"
        alt="clap"
      />
```

- On définit l'action :

```
export const addPostLike = (data) => {
  return (dispatch) => {
    return axios
      .put(`http://localhost:3000/posts/${data.id}`, data)
      .then((res) => {
        dispatch({ type: ADD_POST_LIKE, payload: res.data });
      });
  };
};
```

- Définir le reducer :

```
case ADD_POST_LIKE:
      return state.map((post) => {
        if (post.id === action.payload.id) {
          return {
            ...post,
            likes: action.payload.likes,
          };
        } else return post;
      });
```

Tache 11 : add like de user

- Dans Like.js :

```
 const userData = {
      pseudo: user.pseudo,
      likes: user.likes + 1,
      age: user.age,
      id: user.id,
    };
 dispatch(addUserLike(userData));
```

On ajoute aussi au le **useSelector** afin récupérer les likes :

```
const user = useSelector((state) => state.userReducer);
```

- On définit l'action dans le user.action.js :

```
export const addUserLike = (data) => {
  return (dispatch) => {
    return axios
      .put(`http://localhost:3000/user/${data.id}`, data)
      .then((res) => {
        dispatch({ type: ADD_USER_LIKE, payload: res.data });
      });
  };
};
```

- Et on définit son reducer :

```
case ADD_USER_LIKE:
      return {
        ...state,
```

```
        likes: action.payload.likes,
      };
```

By Ribot Fleury

CEO D'EvolveD

teneceskoutse@gmail.com