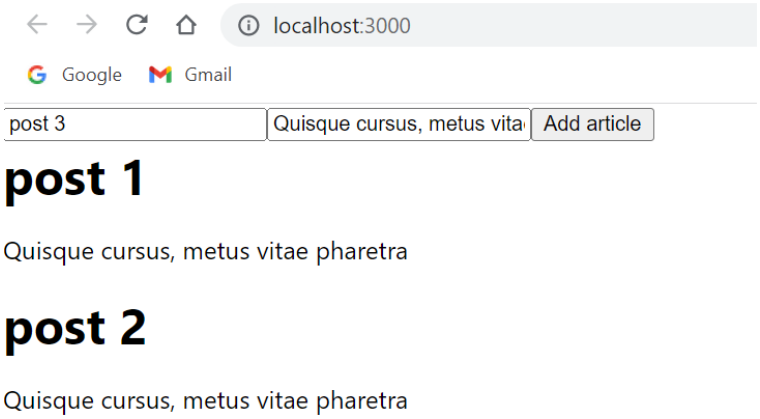
16.Concevoir une application avec Redux

On souhaite créer une application qui permet d'ajouter un article à une liste des articles.



1. **Créez une nouvelle application React avec cette commande :**

```
npx create-react-app react-redux-example
```

2. **Ajoutez les packages redux et react-redux en exécutant sur votre shell**

```
npm install redux react-redux
```

3. **Ajoutez un dossier *containers* dans le src, puis créez le fichier Articles.js.**

```
import React, { useState } from "react"
import Article from "../components/Article"
import AddArticle from "../components/AddArticle"
const Articles = () => {
 const [articles, setArticles] = useState([
  { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },
  { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },
 ])
 const saveArticle = e => {
  e.preventDefault()
 }
 return (
  <div>
   <AddArticle saveArticle={saveArticle} />
   {articles.map(article => (
    <Article key={article.id} article={article} />
   ))}
  </div>
 )
}
export default Articles
```

4. Ajoutez un dossier *components* dans le src, puis créez AddArticle.js et Article.js.

Dans *Article.js*

```
import React from "react"
const article = ({ article }) => (
 <div className="article">
  <h1>{article.title}</h1>
  <p>{article.body}</p>
 </div>
)
export default article
```

Dans *AddArticle.js*

```
import React, { useState } from "react"
const AddArticle = ({ saveArticle }) => {
 const [article, setArticle] = useState()
 const handleArticleData = e => {
  setArticle({
   article,
   [e.target.id]: e.target.value,
  })
 }
 const addNewArticle = e => {
  e.preventDefault()
  saveArticle(article)
 }
 return (
  <form onSubmit={addNewArticle} className="add-article">
   <input
    type="text"
    id="title"
    placeholder="Title"
    onChange={handleArticleData}
   />
   <input
    type="text"
    id="body"
    placeholder="Body"
    onChange={handleArticleData}
   />
   <button>Add article</button>
  </form>
 )
}
export default AddArticle
```

Dans *App.js*

```
import React from "react"
import Articles from "./containers/Articles"
function App() {
 return <Articles />
}
export default App
```

5. Qu'est-ce qu'un réducteur ?

Un réducteur est une fonction pure qui reçoit l'ancien état (précédent) et une action comme arguments, puis renvoie en sortie l'état mis à jour. Le réducteur ne gère que le code synchrone, ce qui signifie aucun effet secondaire comme une requête HTTP ou quelque chose comme ça. Nous pouvons toujours gérer du code asynchrone avec redux et nous apprendrons comment le faire plus tard. Au passage, si vous êtes confus par le terme action, pas de soucis, ce sera beaucoup plus clair plus tard. Créons d'abord notre tout premier réducteur.

La structure de vos fichiers dépend entièrement de vous, cependant je vais suivre la convention et créer un dossier store dans le projet pour contenir nos réducteurs, actions etc. Ensuite, créez un fichier reducer.js.

```
const initialState = {
   articles: [
    { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },
    { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },
   ],
 }
const reducer = (state = initialState, action) => {
   return state
}
export default reducer
```

Comme je l'ai dit plus tôt, un réducteur est juste une fonction qui reçoit l'état précédent et une action en tant que paramètres et renvoie l'état mis à jour. Ici, nous n'avons pas d'état antérieur, il ne sera donc pas défini, nous devons donc l'initialiser avec **initialState** qui contient nos articles prédéfinis.

Maintenant que nous avons installé notre réducteur, il est temps de créer notre store

6. Qu'est-ce qu'un store ?

Un magasin contient tout l'arbre d'état de notre application React. C'est là que notre état d'application vit. Vous pouvez le voir comme un gros objet JavaScript. Pour créer un store, nous avons besoin d'un réducteur à passer en argument. Nous avons déjà un réducteur, connectons-le à notre store.

Dans index.js.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { createStore } from "redux"
import reducer from "./store/reducer"
const store = createStore(reducer)
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Pour créer un store, nous devons d'abord importer **createStore** à partir du paquet redux, puis importer notre réducteur et enfin le passer en argument au store **createStore(reducer)**. Avec cela, nous avons réussi à créer notre store, mais nous n'avons pas encore fini, nous devons le connecter à notre application React.

7. **Comment connecter notre store à React?**

Pour connecter le store à React, nous devons importer une fonction d'assistance nommée **Provider** à partir du package react-redux. Ensuite, on enveloppe notre composant App avec **Provider** et on passe comme props le store qui a pour valeur notre store actuel.

Dans index.js.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { createStore } from "redux"
import { Provider } from "react-redux"
import reducer from "./store/reducer"
const store = createStore(reducer)
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
<Provider store={store}>
  <App />
</Provider>
);
```

Ensuite, nous devons connecter notre composant au store redux.

In Articles.js

```
import React from "react"
import { connect } from "react-redux"
import Article from "../components/Article"
import AddArticle from "../components/AddArticle"
```

```
const Articles = ({ articles }) => {
  const saveArticle = e => {
    e.preventDefault()
    // the logic will be updated later
  }
  return (
    <div>
      <AddArticle saveArticle={saveArticle} />
      {articles.map(article => (
        <Article key={article.id} article={article} />
      ))}
    </div>
  )
}
const mapStateToProps = state => {
  return {
    articles: state.articles,
  }
}
export default connect(mapStateToProps)(Articles)
```

Ici, nous importons d'abord **connect()**, une fonction qui retourne une fonction d'ordre supérieur et reçoit en entrée un composant. Il nous aide à connecter notre composant au store et à nous donner accès à l'état.

Ensuite, nous déclarons une nouvelle fonction nommée **mapStateToProps()** (vous pouvez la nommer comme bon vous semble). Il est utilisé pour obtenir notre état du store redux. La fonction reçoit en paramètre state stocké dans redux et retourne un objet JavaScript qui contiendra nos articles.
Et pour atteindre le magasin, nous devons passer **mapStateToProps()** à la fonction **connect()**. Il prendra notre composant Articles et retournera un composant conteneur avec les props qu'il doit injecter. Cela signifie que nous pouvons maintenant obtenir notre état contenu dans le store. L'état est reçu par le composant via des props, nous pouvons toujours afficher les articles comme avant mais maintenant via redux.

Nous avons réussi à connecter notre store à React. Passons maintenant aux actions

8. Qu'est-ce qu'une action ?

Une action est un paquet utile d'informations qui contient un type comme **REMOVE_ARTICLE** ou **ADD_ARTICLE** etc. Les actions sont distribuées à partir de votre composant. Il envoie les données de votre composant React à votre store Redux. L'action n'atteint pas le store, c'est juste le messager. Le store est changé par le réducteur.

Pour créer une action dans notre projet, nous devons créer dans notre dossier store un nouveau fichier nommé actionTypes.js.

```
export const ADD_ARTICLE = "ADD_ARTICLE"
```

Ensuite, nous avons besoin d'aller sur Articles.js et y ajouter ce code suivant.

```
import React from "react"
import { connect } from "react-redux"
import Article from "../components/Article"
import AddArticle from "../components/AddArticle"
import * as actionTypes from "../store/actionTypes"
const Articles = ({ articles, saveArticle }) => (
  <div>
   <AddArticle saveArticle={saveArticle} />
   {articles.map(article => (
    <Article key={article.id} article={article} />
   ))}
  </div>
)
const mapStateToProps = state => {
  return {
   articles: state.articles,
  }
}
const mapDispatchToProps = dispatch => {
  return {
   saveArticle: article =>
    dispatch({ type: actionTypes.ADD_ARTICLE, articleData: { article } }),
  }
}
export default connect(mapStateToProps, mapDispatchToProps)(Articles)
```

Ensuite, nous devons tout importer de **actionTypes.js**. Et créez une nouvelle fonction **mapDispatchToProps** qui reçoit une fonction **dispatch** en paramètre. Le **mapDispatchToProps** retourne un objet qui a une propriété **saveArticle**. C'est une référence à une fonction qui enverra une action dans notre store.

saveArticle contient une fonction anonyme qui reçoit notre article en argument et retourne la fonction **dispatch**. Il reçoit le type et les données à mettre à jour en tant que paramètres. Et comme vous le devinez, cela enverra l'action dans notre store. Enfin, nous devons passer **mapDispatchToProps** comme deuxième argument à la fonction **connect**. Et pour le faire fonctionner, nous devons mettre à jour notre réducteur et ajouter l'action **ADD_ARTICLE**.

Dans store/reducer.js

```
import * as actionTypes from "./actionTypes"
const initialState = {
  articles: [
    { id: 1, title: "post 1", body: "Quisque cursus, metus vitae
pharetra" },
    { id: 2, title: "post 2", body: "Quisque cursus, metus vitae
pharetra" },
```

```
  ],
}
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case actionTypes.ADD_ARTICLE:
      const newArticle = {
        id: Math.random(), // not really unique but it's just an
example
        title: action.article.title,
        body: action.article.body,
      }
      return {
        state,
        articles: state.articles.concat(newArticle),
      }
  }
  return state
}
export default reducer
```

Comme vous pouvez le voir, nous importons en premier **actionTypes**. Ensuite, nous vérifions dans notre fonction **reducer** si le type de l'action est égal à **ADD_ARTICLE**. Si c'est le cas, on crée d'abord un nouvel objet qui contient notre article, puis on l'ajoute à notre tableau d'articles. Avant de retourner l'état, nous copions l'ancien état, puis on utilise **concat** avec le nouvel article pour l'ajouter. De cette façon, nous gardons notre état sûr et immuable.

9. **Comment gérer le code asynchrone avec redux?**

Le réducteur comme je l'ai dis plus tôt ne gère que le code synchrone. Pour exécuter du code asynchrone, nous devons utiliser un créateur d'action. C'est une fonction qui renvoie une fonction ou une action devrais-je dire. Donc, pour l'utiliser dans notre projet, nous devons créer un nouveau fichier actionCreators.js.

Dans store/actionCreators.js

```
import * as actionTypes from "./actionTypes"
export const addArticle = article => {
 return {
  type: actionTypes.ADD_ARTICLE,
  article,
 }
}
```

Ici, nous déclarons un nouveau créateur d'action nommé **addArticle**. C'est une fonction qui reçoit article comme argument et renvoie le type de l'action et la valeur. Au fait, article est identique à article: article, c'est juste une syntaxe pratique ES6. Nous pouvons maintenant passer à autre chose et changer la fonction **mapDispatchToProps** dans le fichier Articles.js.

Dans Articles.js

```
import React from "react"
import { connect } from "react-redux"
import Article from "../components/Article"
import AddArticle from "../components/AddArticle"
import { addArticle } from "../store/actionCreators"
const Articles = ({ articles, saveArticle }) => (
  <div>
   <AddArticle saveArticle={saveArticle} />
   {articles.map(article => (
    <Article key={article.id} article={article} />
   ))}
  </div>
)
const mapStateToProps = state => {
  return {
   articles: state.articles,
  }
}
const mapDispatchToProps = dispatch => {
  return {
    saveArticle: article => dispatch(addArticle(article)),
  }
}
export default connect(mapStateToProps, mapDispatchToProps)(Articles)
```