Redux a été créé pour rendre la gestion de state prévisible, en fournissant un conteneur de state unique et des règles strictes sur la façon dont le state peut être modifié.

Redux est une petite bibliothèque JavaScript et peut être utilisée avec n'importe quel framework fron-end, tel que React, Angular, jQuery.

Il utilise le modèle de « source unique de vérité ».

En bref, la source unique de vérité se réfère simplement à la relocalisation de state de l'application et de toute la logique associée en dehors de l'application, permettant à N'IMPORTE QUEL composant d'accéder aux données dont il a besoin.

Le fait d'avoir un seul conteneur de state facilite la gestion de state de votre application, car vous pouvez accéder aux données et les modifier à partir de n'importe quel composant qui en a besoin, sans avoir à transmettre les données.

**Introducing Redux**

**Redux can be used with Angular. 🡺** True

Store  
  
Examinons les concepts de base de Redux.

Dans Redux, l'état de l'application est stocké sous la forme d'un objet simple, appelé store.

Il ne devrait y avoir qu'un seul stock dans une application.

Par exemple, un magasin peut ressembler à ceci :

{   
contacts: [{   
name: 'David'   
}, {   
name: 'Amy'   
}],   
toggle: true   
}

**JS**

You cannot change the state directly. Instead, you need to **dispatch an action**.

**Store: The store is a(n):**

object

Actions & Reducer

Une action est juste un simple objet JavaScript :

{   
type: 'ADD\_CONTACT',   
name: 'James'   
}

Le code ci-dessus définit une action avec le type ADD\_CONTACT et une propriété de nom.

Une action décrit clairement pourquoi le changement de state s'est produit et peut être envoyée depuis n'importe où dans votre application.

À ce stade, nous avons juste un store, qui comprend notre data state et un objet, qui comprend certaines données qui doivent être modifiées dans state. Alors, comment procédons-nous réellement au changement ?

Pour lier le store et l'action, nous devons écrire une fonction, appelée reducer.

Il prend le state et l'action comme arguments et renvoie le state suivant de l'application.

Par example:

function contactsApp(state, action) {   
if (action.type === 'ADD\_CONTACT') {   
return [ ...state, action.name ]   
} else {   
return state   
}   
}

Le code ci-dessus définit une simple fonction reducer, qui vérifie l'action et renvoie le nouvel state.

Ces concepts sont essentiellement l'idée de Redux : vous maintenez le state global dans un store, définissez des actions pour décrire ce qu'il faut changer dans le store et écrivez des fonctions de reducer pour gérer ces actions.

Remarquez que nous n'avons touché à aucune syntaxe spécifique à React, tout ce qui précède est du JavaScript simple.

**Actions & Reducers: Fill in the blanks to create a valid action object:**

{ type: 'APPLY\_DISCOUNT'

percent25



Core Concepts

Redux peut être décrit en utilisant les principes suivants :

**Source unique de vérité**

Le State global de l'application est stocké dans un seul store.

**L'état est en lecture seule**

Vous ne pouvez modifier le state qu'en envoyant des actions. Les actions sont des objets qui contiennent des informations sur ce qui doit être changé.

**Reducers purs**

Les **reducers** sont des fonctions qui gèrent les actions et renvoient le state suivant de l'application. Les **reducers** doivent être purs, ce qui signifie qu'ils ne peuvent pas modifier le state, ils doivent renvoyer un nouvel objet de state.

Plongeons-nous dans ces concepts et créons de véritables applications React qui utilisent Redux !

**Core Concepts: Which of the following is an attribute describing a pure function?**

A function that does not change its input parameters

Actions

L'action peut être considérée comme des charges utiles d'informations qui envoient des données au store. Les actions sont représentées par un simple objet JavaScript et doivent avoir une propriété type :

{   
**type: 'ADD\_CONTACT',**  
name: 'James'   
}

**JS**

Dans l'exemple ci-dessus, nous définissons une action avec le type ADD\_CONTACT et lui fournissons une propriété name comme charge utile.

Notez que pour le type, nous utilisons toutes les lettres majuscules et les mots séparés par des traits de soulignement. Ceci est également appelé "cas du serpent". Il s'agit de la manière généralement acceptée de créer un type d'action.

Vous pouvez utiliser n'importe quel nom et structure pour les autres propriétés définissant les données dans l'action. Vous pouvez, par exemple, l'appeler charge utile et lui fournir un objet avec les données :

{   
type: 'ADD\_CONTACT',   
**payload: {   
name: "Jimmy Barnes"   
}**  
}

Vous devez transmettre le moins de données possible dans chaque action. Cela permet de garder les actions propres et faciles à lire.

**Actions: Drag and drop to create a valid action called 'CHANGE\_ITEM' with a payload object:**

{

type: 'CHANGE\_ITEM',

payload: {

text: 'J',

num: 42

} }

Action Creators

Afin d'utiliser la même action avec différentes charges utiles, ainsi que de créer du code réutilisable, nous pouvons créer des créateurs d'action.

Les créateurs d'action sont des fonctions simples qui renvoient des actions.

Par example:

function addContact(person) {   
return {   
type: 'ADD\_CONTACT',   
payload: person   
}   
}

La fonction de créateur d'action prend un paramètre de personne et l'utilise comme charge utile des actions.

Maintenant, nous pouvons utiliser le créateur d'action pour créer plusieurs nouveaux contacts en lui transmettant les données correspondantes.

Les créateurs d'action ne sont pas intégrés à la bibliothèque Redux par défaut. C'est un modèle qui a été implémenté pour créer du code qui reflète une approche plus DRY (Don't Repeat Yourself).

**Action Creators**

**Fill in the blanks to make this a valid action creator:**

addFood(food) {

{

'ADD\_FOOD',

payload: food

} }

Reducer Function

Les reducers sont des fonctions qui gèrent les actions.

La fonction prend le state actuel et l'action comme paramètres et renvoie le nouvel State.

Un Reducer peut gérer plusieurs actions, il inclut donc généralement une instruction switch pour chaque cas d'action.

**For example:**

function contactsApp(state, action) {   
switch (action.type) {   
case 'ADD\_CONTACT':   
return [ ...state, action.person ]   
default:   
return state   
}   
}

Dans le code ci-dessus, notre fonction de Reducer utilise une instruction switch pour handle les actions appropriées.

Par défaut, il renvoie simplement le State actuel.

N'oubliez pas que le reducer doit être une fonction pure, ce qui signifie qu'il ne peut pas modifier le State actuel. Il doit renvoyer un nouvel objet de State à la place.

Le cas par défaut est ajouté pour gérer les actions inconnues.

**Reducer Function**

**The default case is used to:** handle unknown actions

Multiple Reducers

Si vous avez plus d'une entité (c'est-à-dire des utilisateurs, des produits, des factures, des commandes, etc.), c'est généralement une bonne idée de les diviser en plusieurs Reducers fonctions pour séparer les préoccupations.

Redux nous donne une méthode que nous pouvons utiliser appelée combineReducers. Cela nous permet d'utiliser plus d'un réducteur afin que lorsqu'une action est envoyée, l'action soit exécutée à travers tous les réducteurs au lieu d'un seul. Cela nous permet également de séparer les préoccupations de State de notre store.

Par example:

const contactsApp = **combineReducers**({   
addContacts,   
doSomething   
})

Maintenant, notre contactsApp combine deux Reducers en un seul.

Il est recommandé de ne fournir à chaque Reducer que la partie de State qu'il doit gérer. C'est ce qu'on appelle la composition du Reducer et c'est un modèle fondamental de création d'applications Redux.

**Multiple Reducers**

**Fill in the blanks to create a valid reducer function:**

 reducer(state, action) {

 (action.type) {

'ADD\_ITEM':

return [ ...state, action.item ]

:



state

} }

Redux with React

Maintenant que nous savons ce qu'est Redux, nous pouvons commencer à créer des applications React qui utilisent Redux !

Tout d'abord, nous devons installer Redux : npm install redux

Cela installera la bibliothèque Redux.

Cependant, Redux lui-même n'est qu'une petite bibliothèque, qui peut être utilisée avec différentes technologies.

Pour l'utiliser avec React, nous devons installer une autre bibliothèque, appelée react-redux :

npm install react-redux

La bibliothèque react-redux lie React à Redux, permettant aux composants React de lire les données d'un store Redux et d'envoyer des actions au store pour mettre à jour les données.

**Redux with React**

**Fill in the blanks to install the required libraries for using Redux with React.**

 install redux

npm  

**Counter App**

Comme premier exemple, construisons l'application Counter que nous avons créée dans le module précédent en utilisant Redux !

Tout d'abord, nous devons créer notre action et le reducer correspondant.

function incrementCounter(num) {   
return {   
type: 'INCREMENT',   
num: num   
}   
}

**JS**

Le code ci-dessus déclare une fonction de création d'action nommée incrementCounter(), qui renvoie une action de type INCREMENT et la charge utile correspondante.

Notre reducer:

const initialState = {   
count: 0   
};   
  
function reducer(state = initialState, action) {   
switch(action.type) {   
case 'INCREMENT':   
return { count: state.count + action.num };   
default:   
return state;   
}   
}

**JS**

Le code ci-dessus définit une fonction Reducer, qui renvoie le nouvel State en fonction de l'action donnée. Nous incrémentons la variable State count de la valeur num fournie.

Nous fournissons également une valeur par défaut pour notre State en utilisant la variable initialState.

Rien d'extraordinaire jusqu'à présent, nous venons de créer deux fonctions simples, l'une renvoyant notre objet d'action, l'autre renvoyant un nouvel State avec le nombre incrémenté.

Creating the Store

Pour créer le store, nous appelons la fonction createStore(), qui prend le reducer comme paramètre :

const store = **createStore(reducer)**;

Mais comment passe-t-on le store à nos composants ?

Ceci est réalisé en utilisant un élément <Provider> spécial. Il rend le store disponible pour tout composant child imbriqué.

Ainsi, pour notre compteur, nous aurions ceci :

const el = **<Provider store={store}>**  
<Counter/>   
**</Provider>**;

Le Provider prend le store en tant qu'attribut et le met à la disposition de son composant child.

We need to import { createStore } and { Provider } using the following syntax:  
**import { Provider } from 'react-redux';  
import { createStore } from 'redux';**

**Creating the Store**

**Drag and drop from the options below to create a store and pass it to our Post component.**

const store = createStore(reducer);

ReactDOM.render( <Provider store={store}>

<Post />

</Provider>,

document.getElementById('root')

);

Connecting to the Store

À ce stade, nous avons créé notre action, le reducer, le store, et l'avons mis à la disposition de notre composant Counter à l'aide de l'élément Provider.

Afin de connecter notre composant au store, nous devons appeler la fonction connect().

La fonction connect() renvoie un nouveau composant, qui encapsule le composant que vous lui avez transmis et le connecte au store à l'aide de ses fonctions de paramètres spéciales.

function connect(mapStateToProps?, mapDispatchToProps?)

connect() prend deux paramètres optionnels :

mapStateToProps

Cette fonction est appelée chaque fois que le State du store change. Il reçoit le State en tant que paramètre et renvoie le State du composant.

Par exemple, pour notre compteur, nous devons renvoyer la variable state count :

function mapStateToProps(state) {   
return {   
count: state.count   
};   
}

Maintenant, notre composant peut accéder à la variable count en utilisant ses props ! Tout comme le nom de la fonction l'indique, il mappe state aux props.

**mapDispatchToProps**

Comme vous l'avez peut-être deviné d'après son nom, ce paramètre est utilisé pour mapper les fonctions de répartition aux props.

Il peut s'agir d'un simple objet définissant la fonction à mapper :

const mapDispatchToProps = {   
incrementCounter   
}

Cela peut sembler un peu déroutant, mais c'est très simple : **mapStateToProps** renvoie simplement les variables de state en tant props à notre composant, tandis que **mapDispatchToProps** permet de définir la manière dont nous répartissons les actions et rendons les fonctions de répartition disponibles en tant props.

Les deux sont facultatifs, car, par exemple, votre composant peut n'avoir besoin que de lire à partir du store.

**mapDispatchToProps** peut également être défini comme une fonction. Consultez la documentation officielle pour plus de détails.

Notez que nous devons importer **la fonction connect : import { connect } from 'react-redux'**

**Connecting to the Store**

**Fill in the blanks to create a valid mapStateToProps function, that returns the food array for our component.**

 mapStateToProps(state) 



{

food: .food

} }

Accessing The Store

À l'intérieur de notre composant, nous accédons simplement aux propriétés du store à l'aide de Props.

function Counter(props) {   
function handleClick() {   
**props.incrementCounter(1);**  
}   
return <div>   
<p>{**props.count**}</p>   
<button onClick={handleClick}>Increment</button>   
</div>;   
}

Notez que nous passons 1 comme argument à notre incrementCounter(), ce qui fait que notre compteur s'incrémente de 1. Nous pouvons changer la valeur en n'importe quel autre nombre, et notre compteur se comportera comme prévu, car nous avons géré le paramètre d'incrémentation dans notre réducteur.

Maintenant, il ne reste plus qu'à appeler la fonction connect() pour notre composant Counter et à l'afficher sur la page :

const Counter = connect(mapStateToProps, mapDispatchToProps)(Counter);   
  
const el = <Provider store={store}>   
<Counter/>   
</Provider>;

Essayez sur StackBlitz

N'oubliez pas que connect() renvoie un nouveau composant, qui encapsule le composant qu'il a reçu.

Nous avons maintenant une application React+Redux entièrement fonctionnelle !

Cela peut sembler trop de code pour une simple application de compteur, mais cette architecture est idéale pour créer des applications à grande échelle, qui utilisent de nombreux composants, plusieurs niveaux d'imbrication et gèrent beaucoup de données.

# Accessing The Store

## When a component is connected to the store, the store properties can be accessed using:Props

Project Structure

Dans notre exemple Counter, nous avons écrit tout le code dans un seul fichier source.

Généralement, les applications Web contiennent plusieurs composants, reducers et actions.

Pour rendre notre projet plus gérable, nous pouvons utiliser des fichiers source (et des dossiers) séparés pour les composants, les reducers et les actions.

Par exemple, nous pouvons déplacer notre composant Counter et la fonction de créateur d'action vers un fichier Counter.js séparé.

Pour utiliser le composant Counter dans notre index.js, nous devons d'abord l'exporter :

export default connect(mapStateToProps, mapDispatchToProps)(Counter);

Remarquez, nous exportons le composant connecté.

Maintenant, nous pouvons importer le composant dans index.js :

import Counter from './Counter';

Essayez-le sur StackBlitz

Nous utilisons **le système de modules ES6**, qui permet d'exporter et d'importer des modules.

**Project Structure**

**Fill in the blanks to correctly import the Post component, which is located in the 'components' folder.**

 Post  './  /Post';

Contact Manager

Changeons notre application Contact Manager pour utiliser Redux !

Nous avons créé un dossier séparé nommé composants pour nos composants.

Nous utilisons Redux pour conserver la liste des contacts.

L'action ADD\_PERSON est utilisée pour ajouter une nouvelle personne à la liste.

Découvrez le code complet du projet sur StackBlitz !

Notez que nous gérons l'état du nouveau formulaire de contact dans React, pas Redux, car les données ne sont utilisées que par le composant AddPersonForm pour conserver temporairement la valeur d'entrée. Vous pouvez le déplacer vers Redux, si vous le souhaitez ; nous avons juste utilisé le state des composants locaux pour plus de simplicité.

Jouez avec le code, personnalisez votre Gestionnaire de contacts en ajoutant de nouvelles actions et de nouveaux composants ! **Partagez vos créations dans la section des commentaires ci-dessous.**

**Contact Manager**

**Drag and drop from the values below to correctly export the Post component connected to the state:**

exportdefault connect(mapStateToProps**,** mapDispatchToProps) (Post);

**Module 3 Quiz**

**11 Comments**

**Which of the following are Redux principles?**

**Reducers should be pure and not modify the state.**

**The application state is stored in a store.**

**State can be changed only by dispatching actions.**

**Which of the following is the name of the required property inside the action object?**

type

**Fill in the blanks to create a valid action creator and a reducer.**

function toggle(status) {

return {

: 'TOGGLE', data: status }

}

function reducer(state, action) {

switch(action.) {

 'TOGGLE':

return { status: state.status+action.data };

default:

 state;

}

}

**Which of the following is used for selecting the part of the data from the store that the connected component needs?**

mapStateToProps

**Fill in the blanks to create a store, pass it to a valid Provider component and render it on the page:**

const store = createStore(reducer);

ReactDOM.

(

<Provider store={}>

<App />

</Provider>,

.getElementById('root') )

**Fill in the blanks to correctly export the Cart component.**

function Cart(props) {

const arr = props.cart;

const listItems = arr.map((val, index) =>

<li key={index}>{val}</li> );

return <ul>{listItems}</ul>;

}

function mapStateToProps(state) {

return {cart: state.cart}

}

Export default connect(

mapStateToProps)

(Cart)

**Rearrange the steps to correctly define the flow in Redux when a button is clicked to change data.**

The click handler is called

An action is dispatched to the store

The store calls the reducer function

The reducer returns the new state

The new state is saved in the store