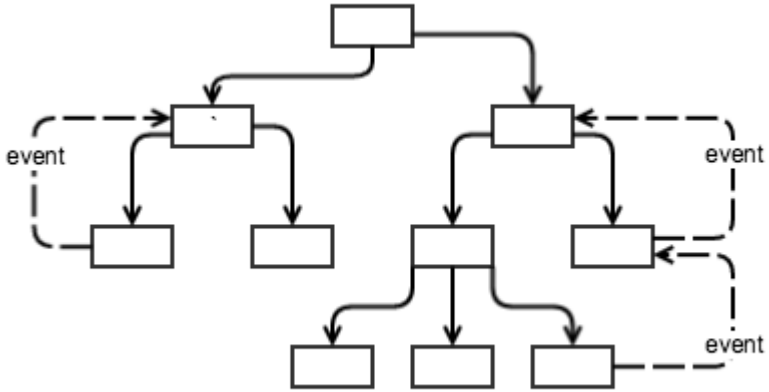


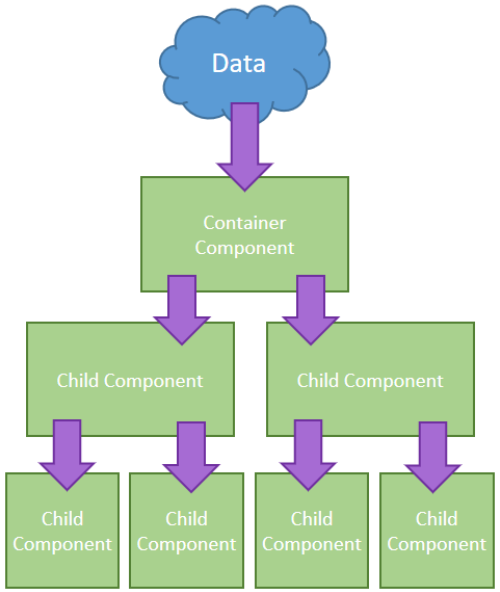
Redux

Animé par Mazen Gharbi

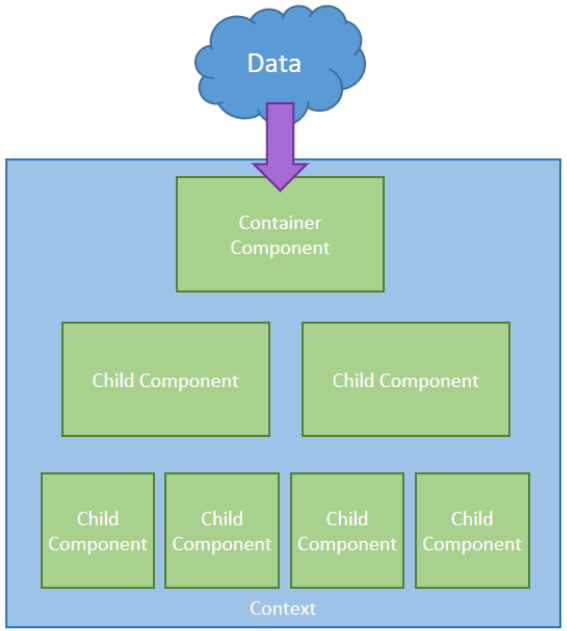
Sommaire

Context API

▷ React offre un moyen de partager de l'information à travers les composants



prop drilling



context API

Créer un contexte

customers.context.ts

```
/* Mise en place du contexte pour les clients */
export const AppContextCustomers = React.createContext({
    customers: [],
    addCustomer: () => { },
    removeCustomer: () => { },
    editCustomer: () => { },
});
```

app.ts

```
<AppContextCustomers.Provider value={this.state.customers}>
        <Root>
                <AppRoute />
        </Root>
</AppContextCustomers.Provider>
```

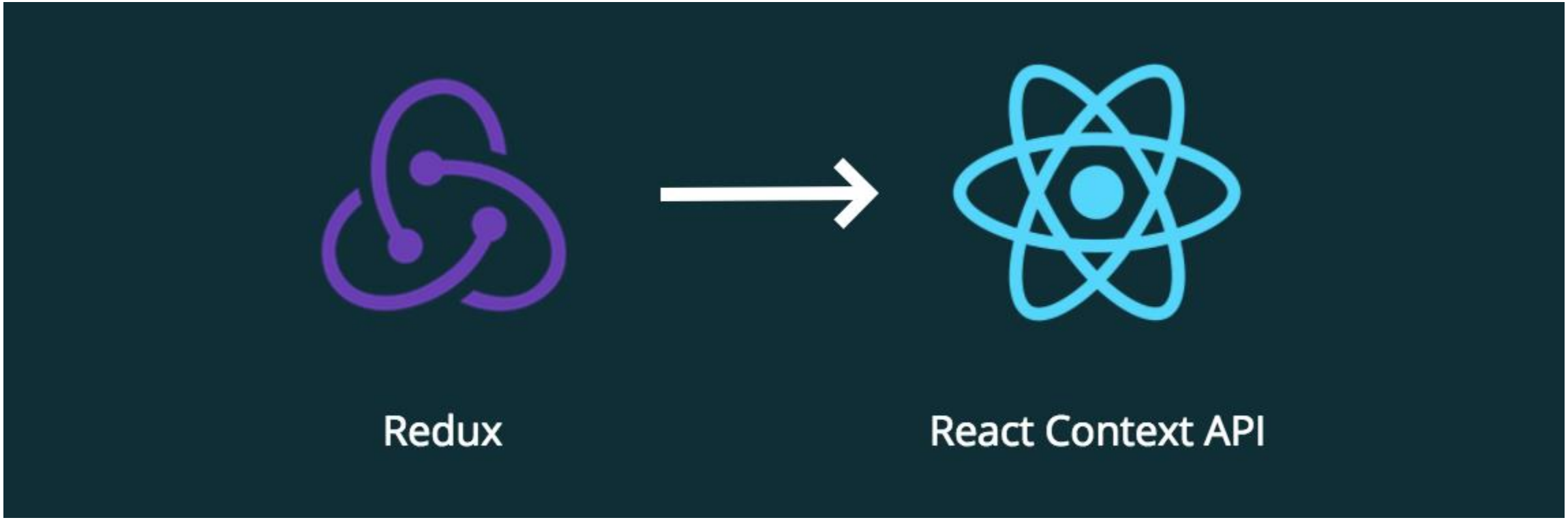
Utilisation du contexte

clients.screen.ts

```
render() {
        return (
            <AppContextCustomers.Consumer>
                {({ customers }) => (
                    (customers.length)
                        ? this.displayClients()
                        : this.displayNoClients()
                )}
            </AppContextCustomers.Consumer>
        );
}
```

```
function App() {
  // Récupération de la valeur du contexte
  const userContext = useContext(AppContextCustomers);

  return <span>{userContext.customers.length}</span>;
}
```

Pas de magie



Introduction

▷ Redux est une librairie pour la gestion d'état de manière prévisible, créée par Dan Abramov pour les applications JavaScript ;

▷ Elle est basée sur le concept de circulation unidirectionnel de données, popularisé par l'équipe Facebook avec son architecture Flux ;

▷ Elle n'a aucune dépendance avec ReactJS, et donc peut être utilisée avec d'autres frameworks JS ;

▷ Elle est la librairie préférée pour la gestion

Application State vs UI State

- Il y a généralement deux « states » dans une app :
  - Application State : état général d'une application. Peut être stocké dans une base de données ou ailleurs
  - UI State : état propre à une partie de l'application (ex: formulaire), éphémère et qui peut être effacé

- Une bonne pratique est de gérer l'Application State par Redux et le UI State par setState

- *Note: Cette règle n'est pas inscrite dans le marbre, vous pouvez utiliser Redux ou setState suivant vos propres besoins.*

Redux - Installation de l'environment

▷ On commence par installer la dépendance Redux

```
> npm install --save redux
```

▷ Malgré le fait que Redux n'ait aucune dépendance avec ReactJS, il y a une librairie officielle (créée par l'équipe Redux) qui harmonise les deux et permet ainsi d'écrire moins de code :

```
> npm install --save react-redux
```

Redux - Installation de l'environnement

▷ Pour aider au debugging d'une application utilisant Redux, il y a un utilitaire qui log tous les changements d'état dans la console :

```
> npm install --save redux-logger
```

Unidirectional Data Flow

▷ L'architecture Flux repose sur l'idée d'un flux de données unidirectionnel strict. On ne peut affecter les données qui y transitent qu'en suivant un sens précis (on ne le court-circuite pas).

▷ Redux implémente cette architecture avec un vocabulaire qui est lui est propre mais le principe est bien le même.

Unidirectional Data Flow

Unidirectional Data Flow

▷ Les composants appellent les « actions creators » pour pouvoir modifier le store (Application State)

▷ Les « actions creators » contiennent la logique principale et dispatchent des actions (events) qui sont simplement des objets qui ont un type pour les différencier et une propriété

Actions

- Les actions sont créées à partir d'un action creator. C’est une fonction qui retourne un objet avec deux propriétés:
  - type: type de l'action (pour pouvoir les différencier)
  - payload: contient le contenu de l'action

- Les actions sont dispatchées au store

Actions

- Prenons par exemple un fichier orderActions.js dans le dossier src

```
// Les actions
export const ADD_ORDER = '[RESTAURANT] Add a new order';

// Nos action-creators
export function addOrderAction(food) {
        return {
                type: ADD_ORDER,
                payload: {
                        order: {
                                food,
                                // Le + permet de caster la date en number (timestamp ici)
                                orderAt: +new Date()
                        }
                }
        }
}
```

Reducers

▷ Les reducers spécifient comment l'Application State doit changer en réponse aux actions dispatchées au store ;

▷ Leurs logiques doivent être prédictibles

- « *pour telle type action reçue avec tel payload et telle application state, je retourne systématiquement le même store* » ;

▷ En pratique, nous ne devons pas avoir une logique qui dépend du moment présent dans un reducer, sinon le nouveau store retourné ne sera pas le résultat voulu. On écrira plutôt cette logique dans un « action creator » ;

▷ Le store retourné doit toujours être un nouvel objet car Redux effectue une vérification === entre l'ancien et le nouvel state

Reducers

▷ Nous allons créer un fichier reducers.js dans le dossier src

```
import { ADD_ORDER } from './orderActions';

function ordersReducer(state = [], action) {
        switch (action.type) {
                // Ajouter à la liste des commandes
                case ADD_ORDER:
                        // La fonction "concat" renvoie un nouveau tableau (donc nouvelle ref.)
                        return state.concat([action.payload.order]);

                default:
                        // Si type qu'on ne connait pas, on retourne le state tel quel
                        // Ducoup, rien ne changera
                        return state;
        }
}
```

Etat initial de l'application state

Reducers

▷ Nous allons maintenant faire combiner un fragment de notre Application State (nous l'appellerons ordersState) avec un reducer qui retournera un nouveau state à chaque fois qu'il traitera une action

Clé du reducer, on en aura besoin par la suite

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

const reducers = combineReducers({
        ordersState: ordersReducer // Reducer créé précédemment
        // On peut intégrer d'autres reducers ici
});

export default reducers;
```

▷ On peut avoir autant de reducers que l'on veut. Il faudra les combiner avec un fragment de notre Application State comme dans notre exemple.

Store

▷ Le store est finalement le cerveau de notre application qui contient l'Application State, les reducers et les méthodes qui permettent de dispatcher une action et de souscrire aux modifications de l'Application State

▷ La dernière étape sera de créer une instance du store et de la connecter aux composants de notre application.

Store

▷ Nous devons créer une instance de notre store, qui combine tous nos reducers

```
// index.js
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducers from './reducers'; // resultat du combineReducers

const store = createStore(reducers);

ReactDom.render(
        <Provider store={store}>
                <App />
        </Provider>,
        document.getElementById('root')
);
```

Store

- Une instance d'un store Redux nous fournit trois méthodes essentielles à l'architecture que nous mettons en place:
  - dispatch envoie les actions aux reducers
  - getState retourne l'état de l'Application State
  - subscribe notifie du changement d'état

- « <Provider> » et « connect » permettent l'accès à ces méthodes

- Provider définit ce que React appelle un contexte (Context)
  - Tout comme BrowserRouter

Créer un composant

▷ Nous allons commencer par créer un composant qui affichera les commandes reçues pour notre restaurant SANS LE STORE pour l'instant

```
import React from 'react';
import PropTypes from 'prop-types';

export default function OrdersList(props) {
        return (
                <ul>
                        {props.orders.map((order) => (
                                <li>{order.food}</li>
                        ))}
                </ul>
        );
}

OrdersList.propTypes = {
        orders: PropTypes.arrayOf(
                PropTypes.shape({
                        orderAt: PropTypes.number,
                        food: PropTypes.string
                })
        )
};
```

Créer un composant

▷ En ouvrant la console, vous aurez ce message:

```
Warning: Each child in an array or iterator should have a
unique "key" prop.
```

▷ React a besoin d'une clé unique pour chaque élément venant d'un tableau (ici c'est props.orders) pour éviter de redessiner tous les éléments

```
<ul>
        {props.orders.map((order) => (
                <li key={order.orderAt}>{order.food}</li>
        ))}
</ul>
```

Créer un composant

▷ Nous allons créer une liste de commandes et les donner à App

```
// index.js
...
const initialOrders = [
        { orderAt: 1565965460, food: 'Pissaladière’},
        { orderAt: 1565965522, food: 'Salade niçoise’},
        { orderAt: 1565965839, food: 'Socca du chef'}
];

ReactDOM.render(
        <Provider store={store}>
                <App orders={initialOrders} />
        </Provider>,
        document.getElementById('root')
);
```

Créer un composant

▷ Puis dans le composant « App »

```
import React, { Component } from 'redux';
import OrdersList from './orders-list’;

function App  {
        return (
                <div>
                        <OrdersList orders={this.props.orders} />
                </div>
        );
}

export default App;
```

▷ Super, ça fonctionne ! Mais nous n’utilisons pas REDUX dutout...

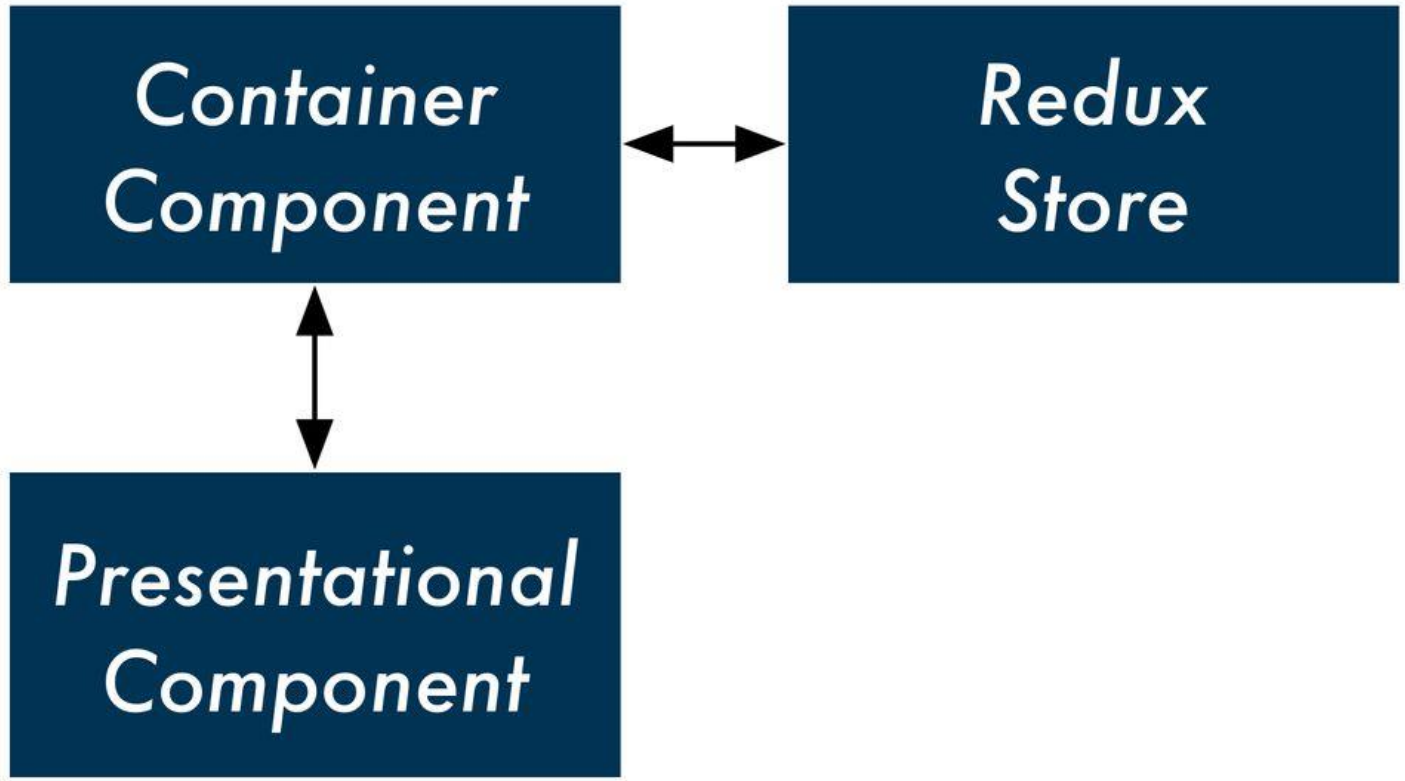
Connecter notre composant à Redux

▷ Il est temps de récupérer la liste des commandes directement au travers du store !

▷ Seuls les « élus » peuvent être attachés au store
- › On les appellent les « Containers »

▷ On va encapsuler « **<OrdersList />** » par un autre composant « **<ConnectOrderList />** » qui lui sera connecté à Redux
- › ConnectedOrdersList est ce qu'on appelle un « *Container component* »
- › OrdersList est un « *Presentational component* »

Containers


Container Component
Redux Store
Presentational Component

Création du container

```
// connect-orders-list.js
import { connect } from 'react-redux';
import OrdersList from './orders-list';

// Cette fonction permettra de faire correspondre
// des reducers ou des variables des reducers au component
// auquel le container est relié
function mapStateToProp(store) {
        return {
                orders: store.ordersState
        }
}

const containerOrdersList = connect(
        mapStateToProp
)(OrdersList);

export default containerOrdersList;
```

Connecter notre composant à Redux

▷ Nous avons besoin maintenant de remplacer « **<OrdersList />** » par « **<ConnectOrdersList />** » dans App.js ;

▷ C'est l'occasion de simplifier notre composant App en le rendant fonctionnel ;

▷ Note: la propriété orders qui est transmise à App dans index.js n'est plus utile maintenant

```
import React from 'react';
import ConnectOrdersList from './connect-orders-list';

export default function App() {
        return (
                <div>
                        <ConnectOrdersList />
                </div>
        );
}
```

Connecter notre composant à Redux

▷ L’état initial de notre Application State est vide. Il ne nous reste plus qu'à le spécifier

```
const initialOrders = [
        { orderAt: 1565965460, food: 'Pissaladière’},
        { orderAt: 1565965522, food: 'Salade niçoise’},
        { orderAt: 1565965839, food: 'Socca du chef'}
];


const initialAppState = {
        ordersState: initialOrders
};

const store = createStore(reducers, initialAppState);
```

Dispatcher une action

▷ Notre liste de commandes n'est pas encore modifiable via notre interface utilisateur ;

▷ Nous allons créer un dernier composant qui se chargera de dispatcher une action ADD_ORDER avec le nom du menu commandé

Dispatcher une action

```
import React from 'react';
import { connect } from 'react-redux';
import { addOrderAction } from './orderActions’;

function OrderComposer {
        return (
                <div>
                        <input type='text' />
                        <button>Nouvelle commande</button>
                </div>
        );
}

export default connect()(OrderComposer);
```

Dispatcher une action

▷ Nous avons besoin d'un état local qui enregistre la valeur de notre champ de texte.

```
// order-composer.js
...

function OrderComposer {
        const [order, setOrder] = useState(‘’);

        return (
                <div>
                        <input type='text' value={order}/>
                        <button>Nouvelle commande</button>
                </div>
        );
}
```

Dispatcher une action

▷ Nous ajoutons maintenant deux méthodes qui mettent à jour notre état local et dispatch l'action d'ajout de commande

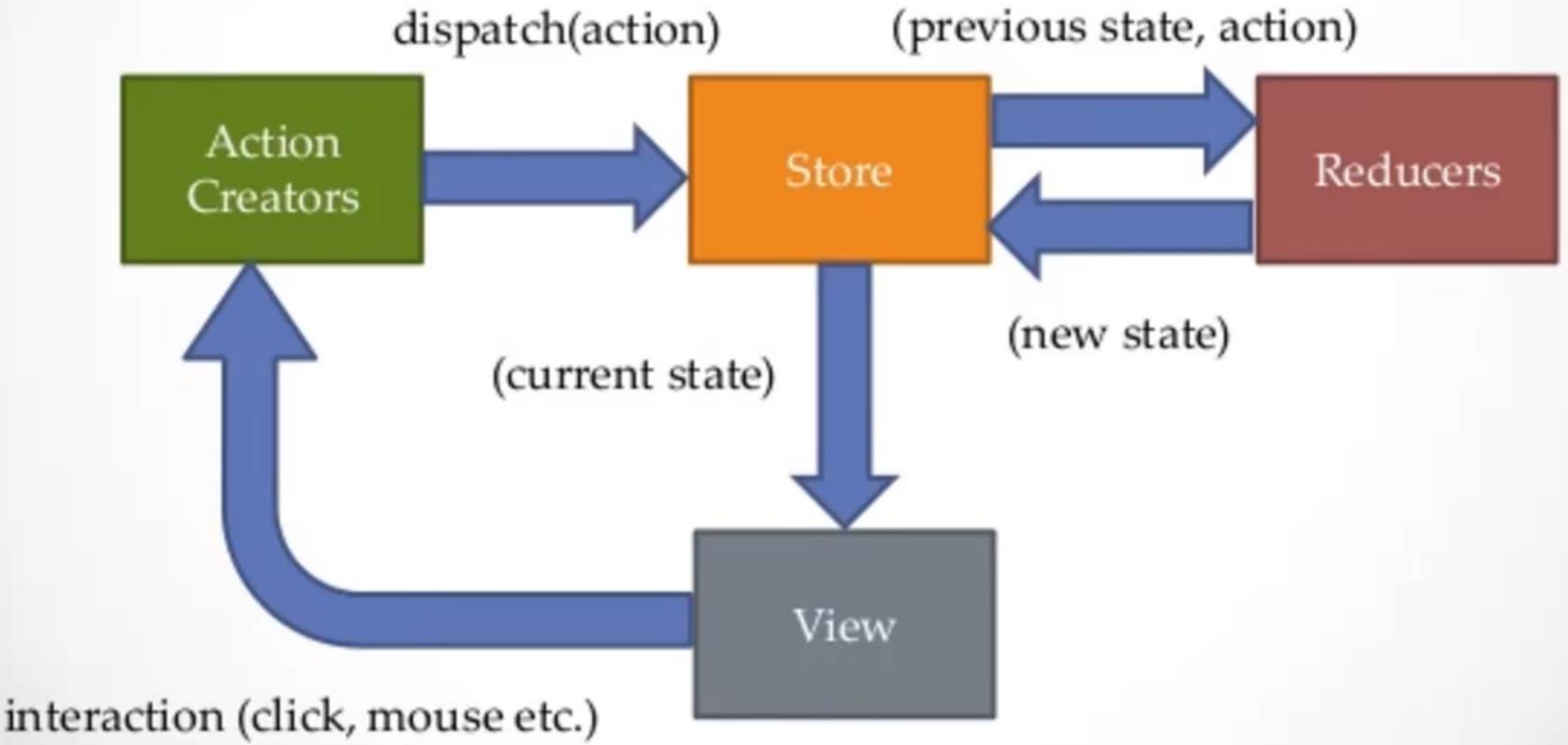
```
// order-composer.js
...
function OrderComposer {
            ...
            const updateOrder = (e) => {
                        const orderValue = e.target.value;

                        setOrder(orderValue);
            }
            // La fonction dispatch est fournie par "connect"
            const addOrder = () => {
                        if(order) {
                                    props.dispatch(
                                                addOrderAction(order)
                                    );
                        }
            }
}
```

Dispatcher une action

- Et nous finissons par affecter les listeners des events DOM.

```
...
return (
        <div>
                <input
                        type='text’
                        value={order}
                        onChange={updateOrder}
                />
                <button onClick={addOrder}>
                        Nouvelle commande
                </button>
        </div>
);
...
```

Redux Data Flow



Un exemple moins élaboré

Essayer ce code !

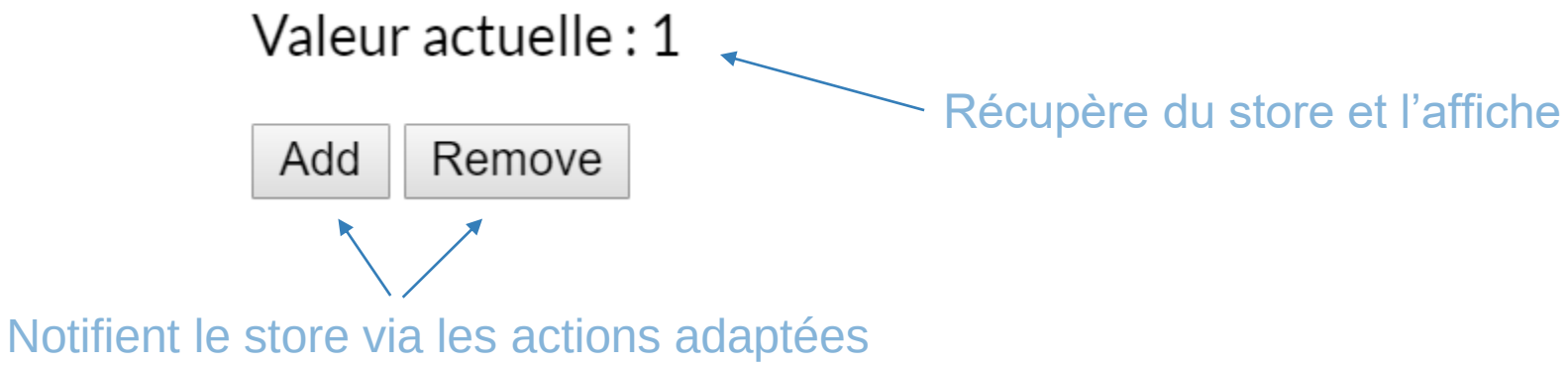
```
const reducer = function(state, action) {
    if(action.type === 'ADD') {
        // On renvoie le nouveau state généré
        return state + action.payload;
    }

    return state;
};

// Le deuxième paramètre correspond au state initial
const store = createStore(reducer, 0);

// On s'abonne au store pour être informé à chaque modification de celui-ci
store.subscribe(() => {
    console.log("Le store a été modifié :" + store.getState());
});

// On envoie un évenement au reducer
store.dispatch({type: 'ADD', payload: 1});
// => Le store a été modifié :1
store.dispatch({type: 'ADD', payload: 1});
// => Le store a été modifié :2
```

C'est ici que notre store est modifié !

Les actions sont émises

Un autre exemple

▷ Plus complexe cette fois. Nous allons tenter de mettre en place l'architecture Redux dans un contexte où un Container fournit les données à un « Presentational component ». Voici un visuel :



Un autre exemple (1)

Essayer ce code !

▷ Une première étape importante va être de créer le store à partir des reducers. Puis d'entourer notre container principal avec un provider auquel on passe notre Store :

index.js

```
let reducers = combineReducers({
    counter,
    // On peut en mettre d'autres ici
});

let store = createStore(reducers);

render(
    <Provider store={store}>
        <MainContainer />
    </Provider>,
    document.getElementById('root')
);
```

On appelle le container qui se chargera d'appeler et d'afficher le composant (en lui envoyant au préalable les informations du store !)

Un autre exemple (2)

Essayer ce code !

▷ Evidemment, on a besoin de définir les actions pour communiquer avec nos Reducers

actions/counter.actions.js

```
export function increment() {
    return {
        type: 'ADD',
        value: 1
    };
}

export function decrement() {
    return {
        type: 'REMOVE',
        value: 1
    }
}
```

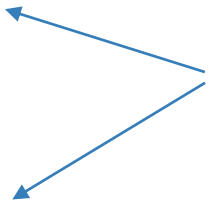
Un autre exemple (3)

Essayer ce code !

▷ Le reducer ici permet de recevoir des actions et de modifier le store en conséquence

reducers/counter.reducer.js

```
let initialState = {
    counter: 0
};

let counterReducer = function (state = initialState, action) {
    switch (action.type) {
        case 'ADD':
            return {
                ...state,
                counter: state.counter + action.value
            };
        case 'REMOVE':
            return {
                ...state,
                counter: state.counter - action.value
            };
        default:
            return state;
    }
};
```



On retourne un objet avec une nouvelle référence pour forcer le rechargement

Un autre exemple (4)

Essayer ce code !

▷ Une fois le reducer et les actions créés, il va falloir mettre en place le container qui liera les informations du store au component

containers/Main.container.js

```
import * as CounterActions from '../actions/counter.actions'

// Données du store à envoyer au composant sous forme de props
let propsMapping = (state) => ({
    counter: state.counter.counter
});

// Fonctions que l'on souhaite mettre à disposition pour notre composant
let dispatchMapping = dispatch => ({
    actions: bindActionCreators(CounterActions, dispatch)
});

let MainContainer = connect(
    propsMapping,
    dispatchMapping
)(MainApp)

export default MainContainer;
```

L'arrow function renvoie directement un objet ici

Ici on relie les données du store et les actions au composant MainApp

Un autre exemple (5)

Essayer ce code !

- Et enfin, le chapitre final : Le composant !

components/MainApp.js

```
export default function MainApp(props) {

        return (
              <div>
                  <p>Valeur actuelle : {props.counter}</p>
                  <button onClick={props.actions.increment}>Add</button> 
                  <button onClick={props.actions.decrement}>Remove</button>
              </div>
          )
    }
}
```

Les informations fournies par le container sont disponibles directement via les props

Pour les actions, il faut ajouter aller chercher dans le sous-objet « action »

Middlewares

▷ Dans le contexte Redux, il est difficile d'avoir un hook dans le flux de données que nous venons de découvrir. Heureusement que les middlewares existent!

▷ Ils règlent tout simplement ce problème en ayant accès aux actions avant qu'elles soient traitées par les reducers mais aussi après qu'un nouveau store soit généré.

Middlewares

▷ « redux-logger » est un middleware. Pour nous aider lors du debugging, on va l'ajouter à notre store.

```
// index.js
...
import { applyMiddleware, createStore } from 'redux';
import { Provider } from 'react-redux';
import reducers from './reducers';
import logger from 'redux-logger';
...
const store = createStore({
          reducers,
          initialAppState,
          applyMiddleware(logger)
})
```

Questions ?