Technologies Web synchrones et multi-dispositifs

CM2 - Programmation Réactive

Plan

- Introduction
- Les principes de la programmation réactive
- En pratique les transformations de flux
- React
- Redux

Qu’est ce que la programmation réactive ?

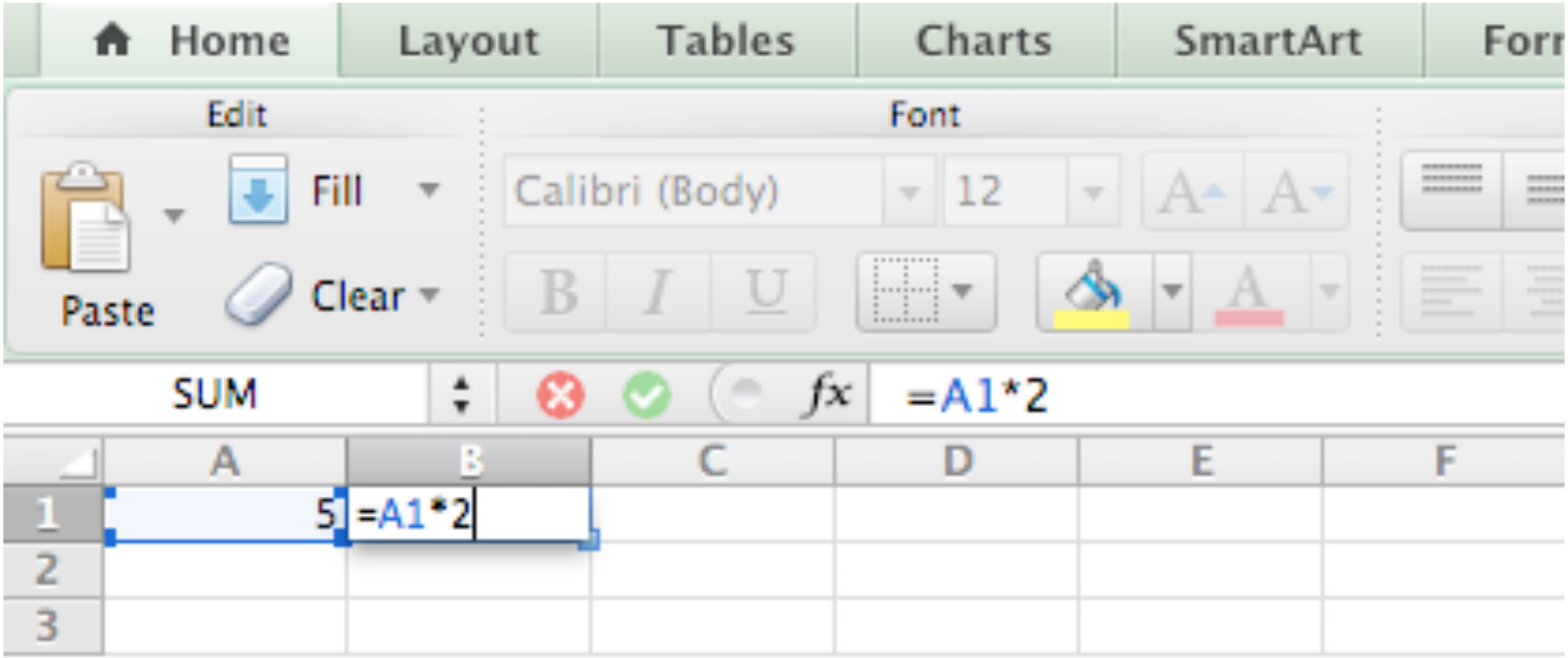
Une approche visant à mieux gérer les flux

Deux types de flux

- Des événements discrets : frappe clavier
- Des évènements continus ou *comportements* : position souris

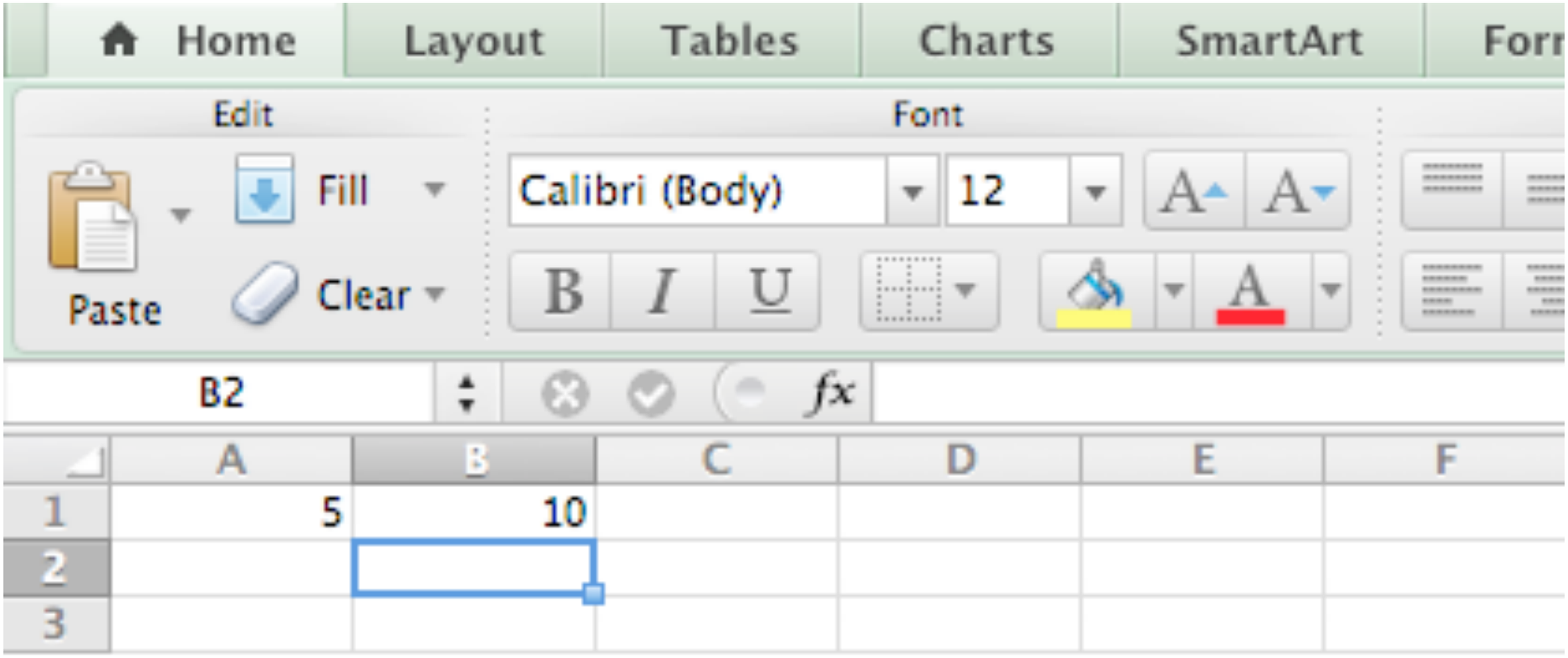
Idée : dépasser les callbacks ou le patron Observer.

Ou avez vous vu ça ?



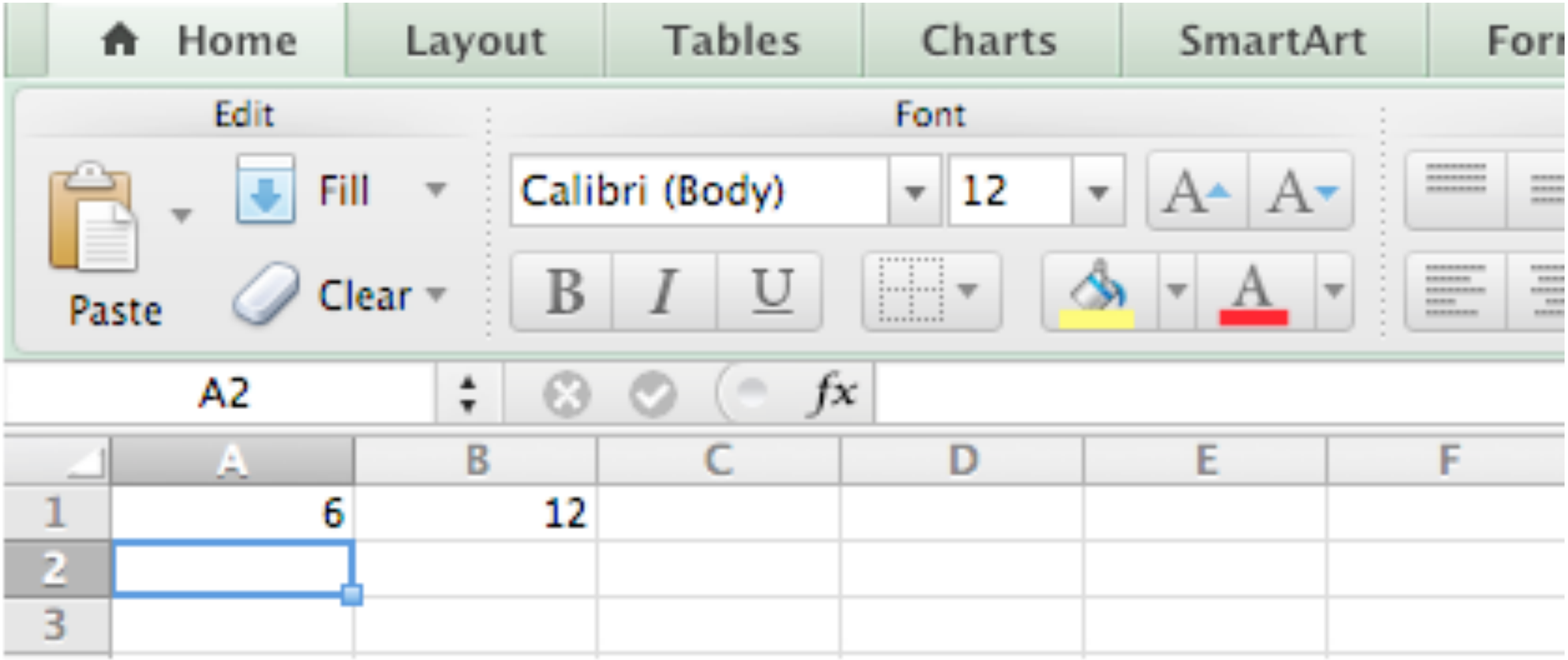
http://www.hanselsolutions.com/blog/surf-talk/shiny-surf.html#/9

Ou avez vous vu ça ?



http://www.hanselsolutions.com/blog/surf-talk/shiny-surf.html#/9

Ou avez vous vu ça ?



http://www.hanselsolutions.com/blog/surf-talk/shiny-surf.html#/9

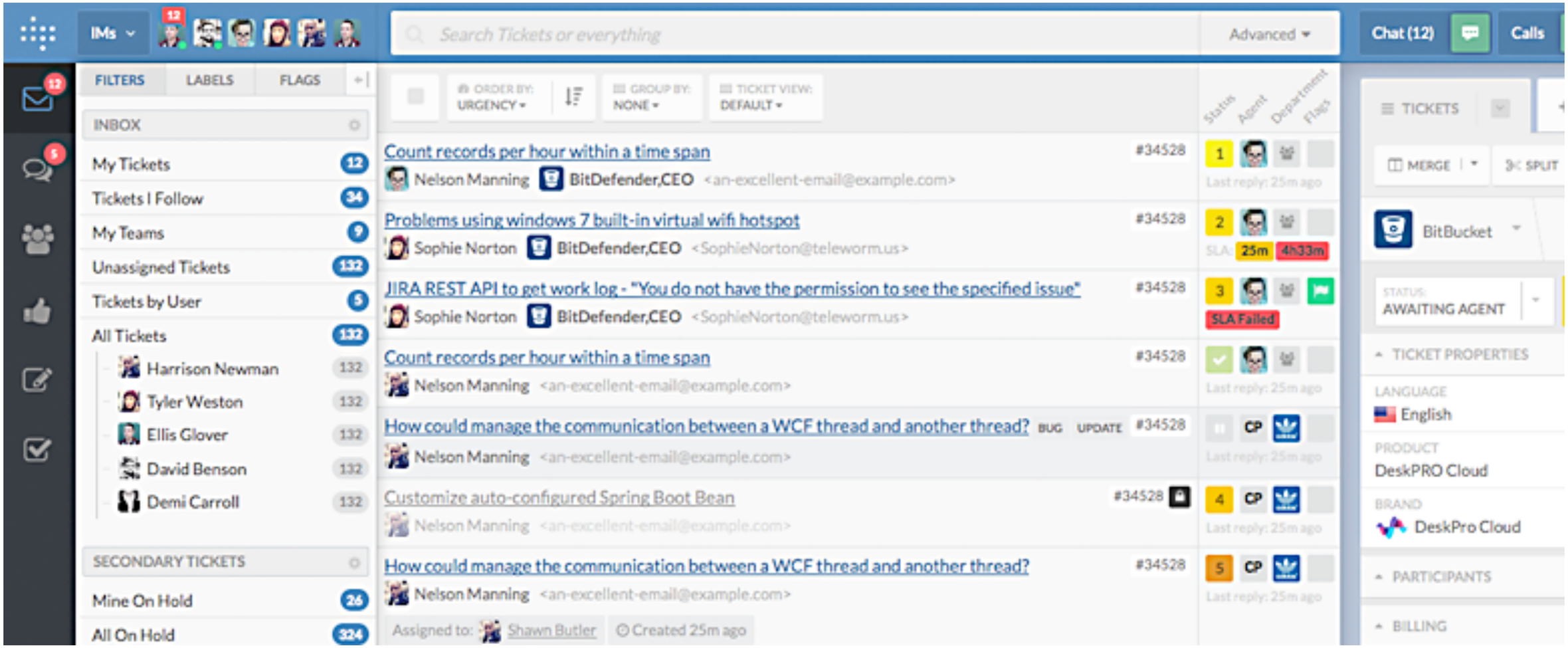
Pourquoi la programmation réactive ?

- Gestion d'évènements et de l'asynchrone
- Faible latence (contraintes sur les temps de réponse)
- Flux de données importants (et rapides).
- Tolérance aux fautes

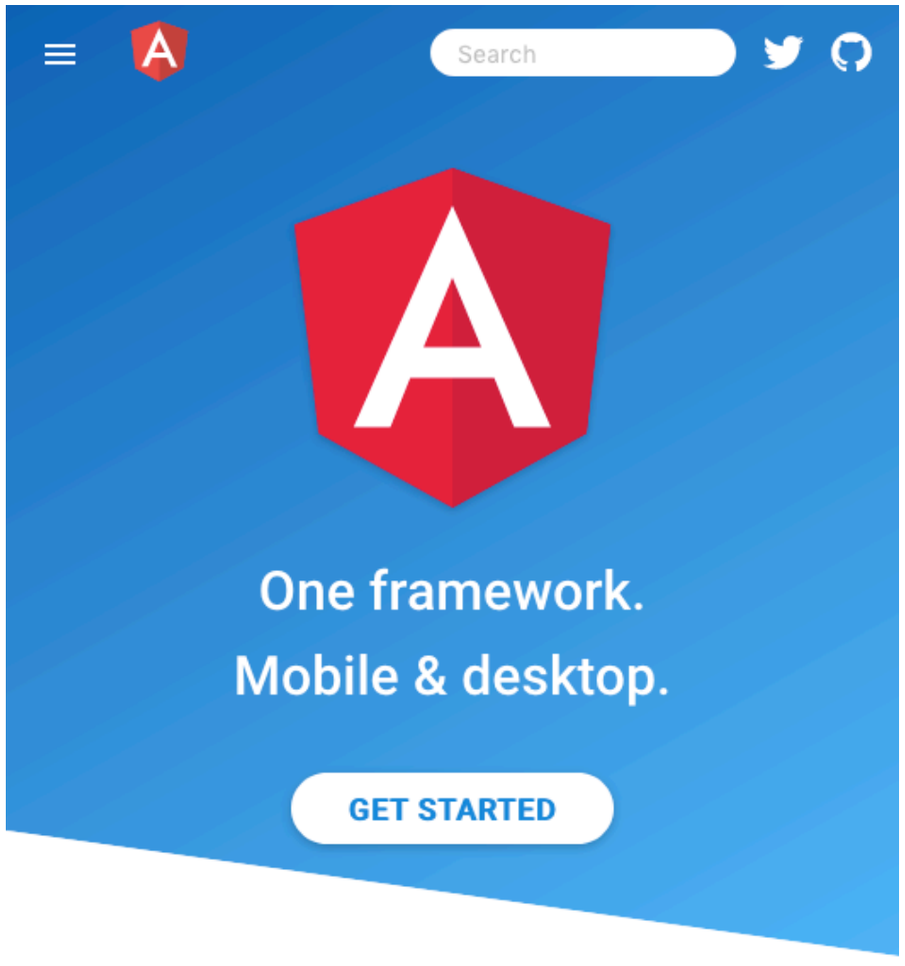
Exemples

À vous

Pourquoi la prog. réactive sur le Web ?

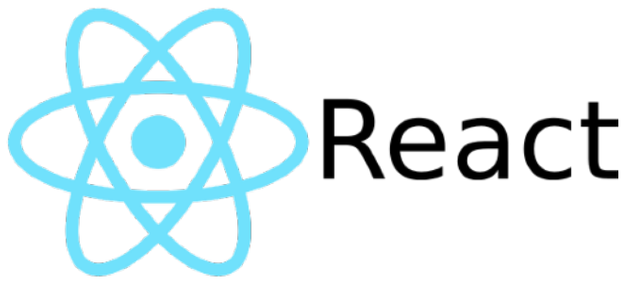


Les bibliothèques Javascript



DEVELOP ACROSS ALL PLATFORMS

Learn one way to build applications with Angular and reuse your code and abilities to build apps for any deployment target. For web, mobile web, native mobile and native desktop.



Plan

- Introduction
- **Les principes de la programmation réactive**
- En pratique les transformations de flux
- React
- Redux

Les principes de base

- Responsive,
- Résilient,
- Élastique,
- Orienté message

Responsive

- Réponse en temps voulu, si possible
- Temps de réponses rapides et fiables (limites hautes)

Résilient

- Résiste à l’échec
- Principes :
  Réplication, conteneurs, isolement, délégation
- On fait en sorte qu’un échec n’impacte qu’un seul composant

Élastique

Le système reste réactif en cas de variation de la charge de travail.

- Pas de point central
- Pas de goulot
- Distribution des entrées entre composants

Message Driven

- Passage de messages asynchrones
  -> Couplage faible, isolation
- Pas de blocage, les composants consomment les ressources quand ils peuvent

Plan

- Introduction
- Les principes de la programmation réactive
- **En pratique les transformations de flux**
- React
- Redux

Un concept important : l’immuabilité

Objet immuable (Immutable object)

- Objet dont l'état ne peut pas être modifié après sa création
- Opposé d’objet variable

Facilite la prog. purement fonctionnelle (pratique pour plein de choses, évite les effets de bords, facilite le undo)

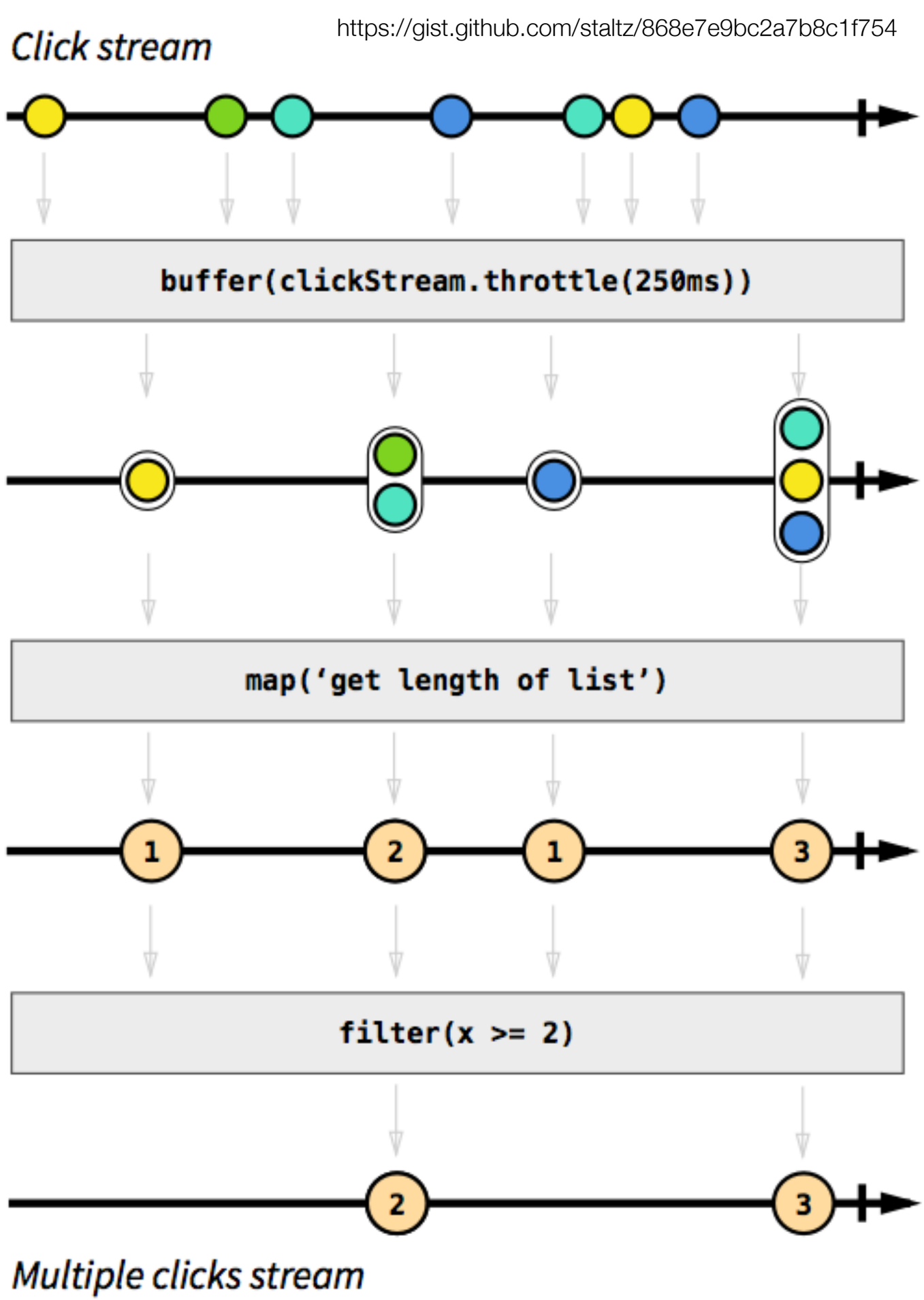
Une seule source de “vérité”

Facilite le caching

Mais ce n’est pas forcément assez : https://codewords.recurse.com/issues/six/immutability-is-not-enough

Un exemple de transformation

https://gist.github.com/staltz/868e7e9bc2a7b8c1f754



Where

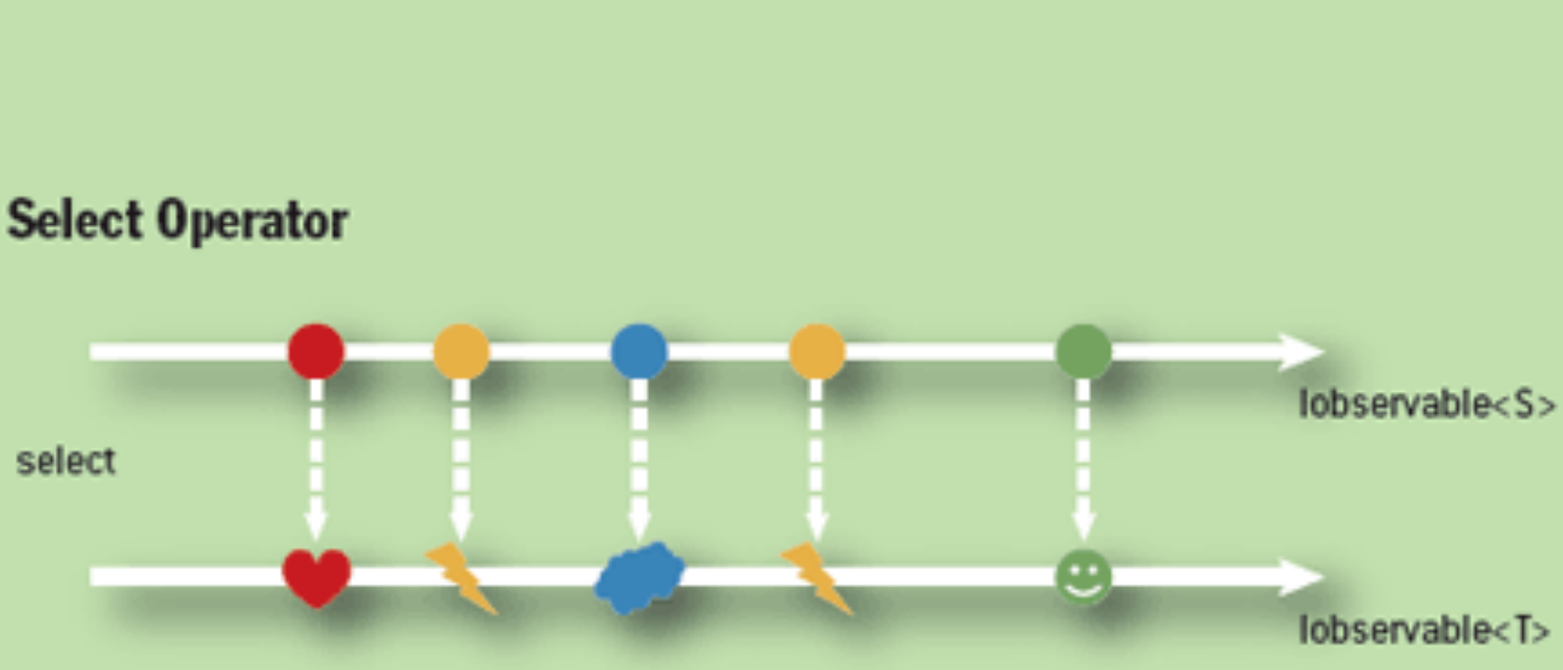
FIGURE 3

The Where Operator

Select

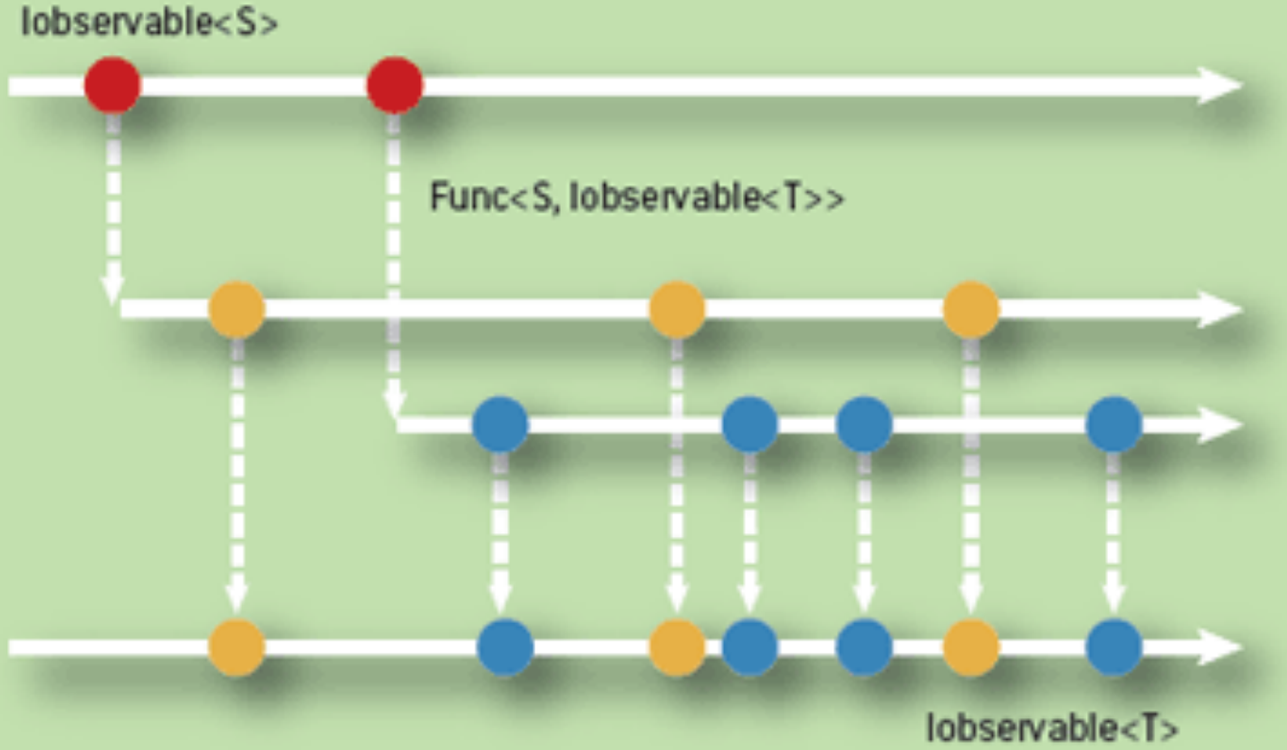
FIGURE 4

The Select Operator



SelectMany : plusieurs flux

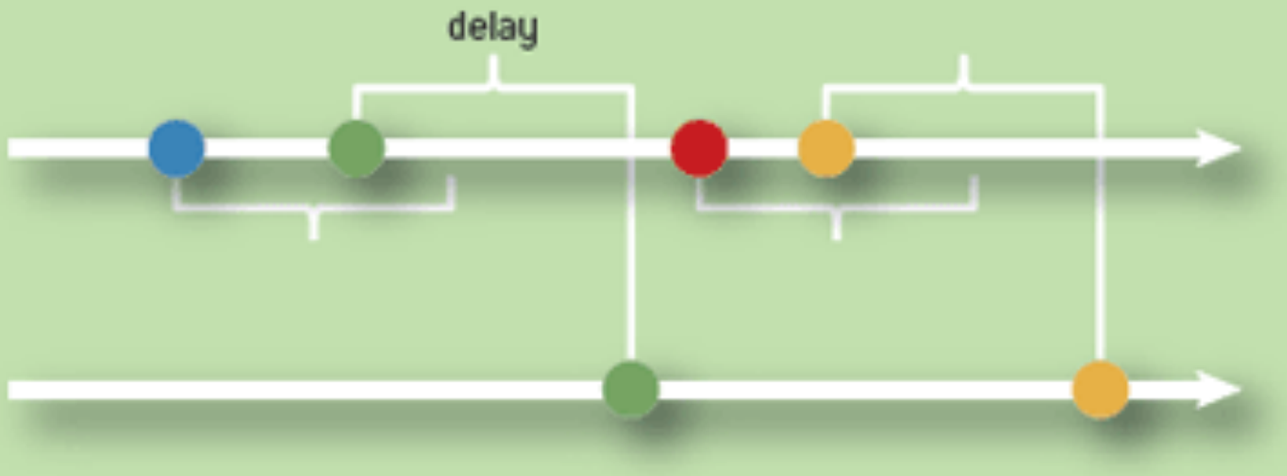
FIGURE 5 **The SelectMany Operator**



Throttle

FIGURE 9

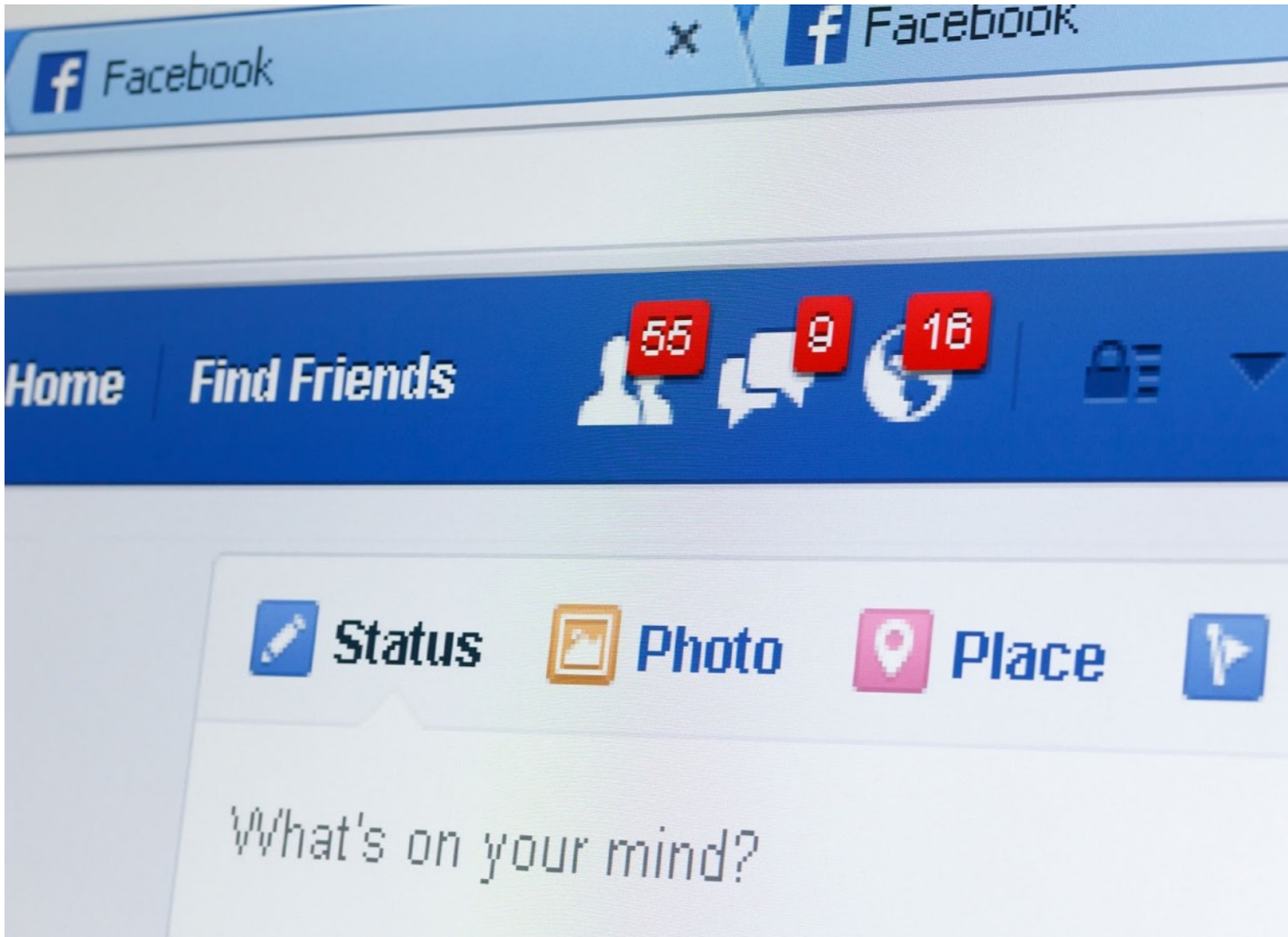
The Throttle Operator



Plan

- Introduction
- Les principes de la programmation réactive
- En pratique les transformations de flux
- **React**
- Redux

Pourquoi React ?



React

Gère la vue

Quelques principes

- Déclaratif
- Centré composant
- Réactif

Déclaratif

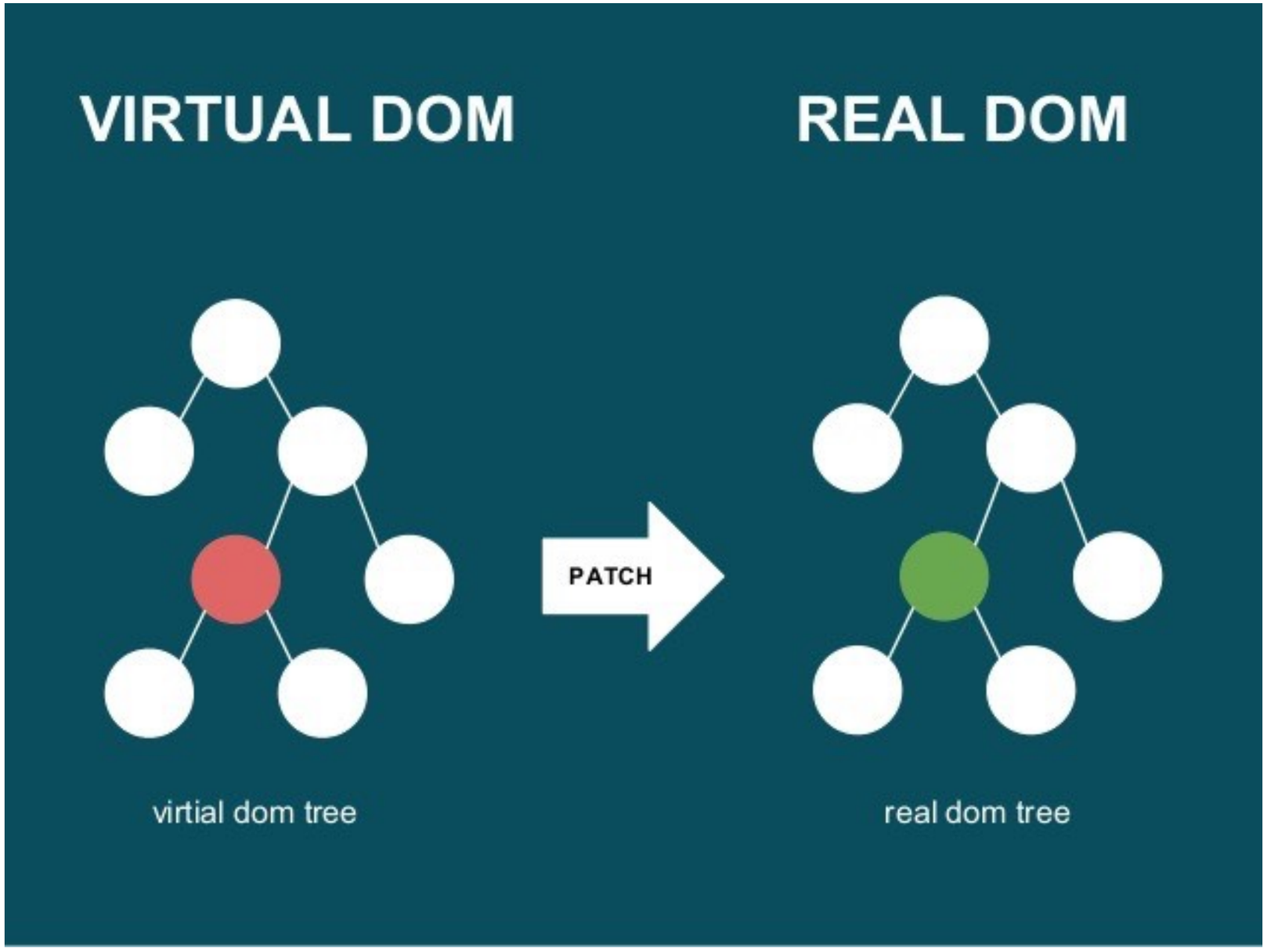
Imperative

```
$('form').on('submit', function(e) {
    e.preventDefault();
    $.ajax({
        url: '/customers',
        type: 'POST',
        data: $(this).serialize(),
        success: function(data) {
            $('.status')
                .append('<h3>' + data + '</h3>');
        }
    });
});
```

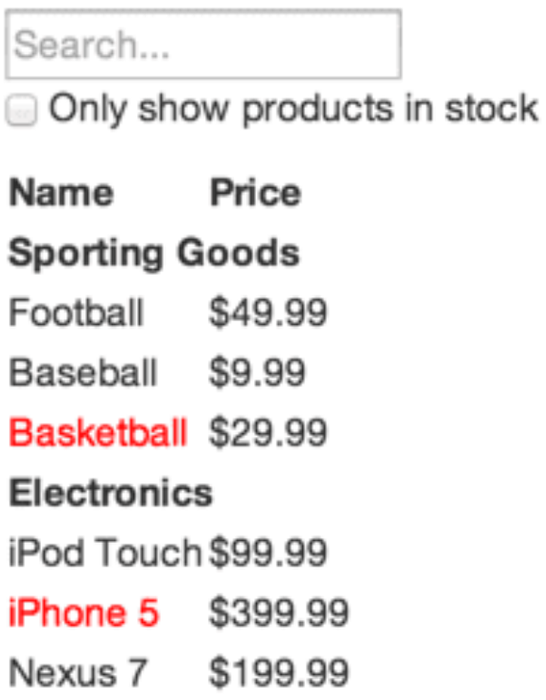
Declarative

```
class NoteBox extends React.Component {
    // ... more code ...
    render() {
        return (
            <div className="NoteBox">
                <h1>Notes</h1>
                <NoteList data={this.state.data} />
                <NoteForm onPost={this.handlePost} />
            </div>
        );
    }
};
```

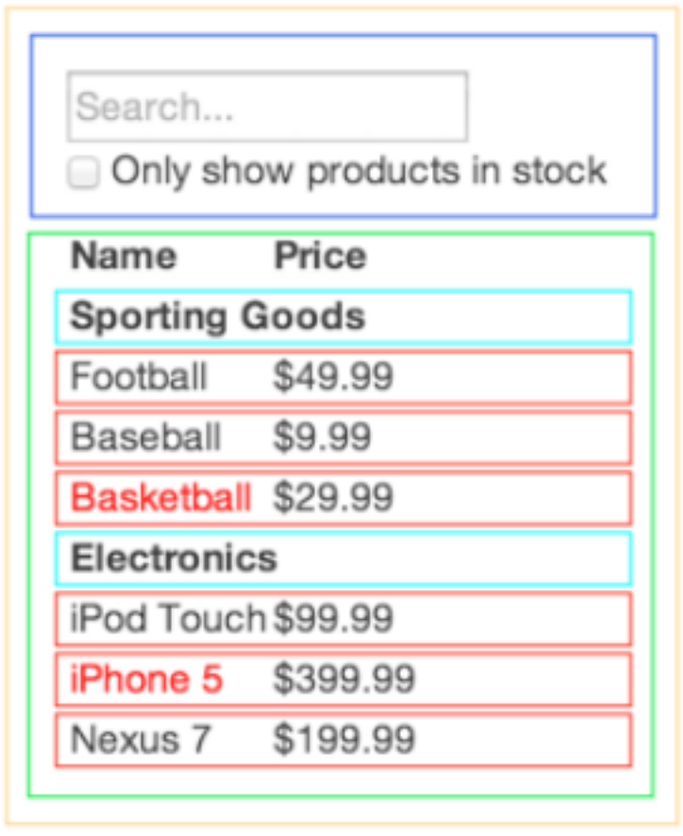
Un DOM Virtuel

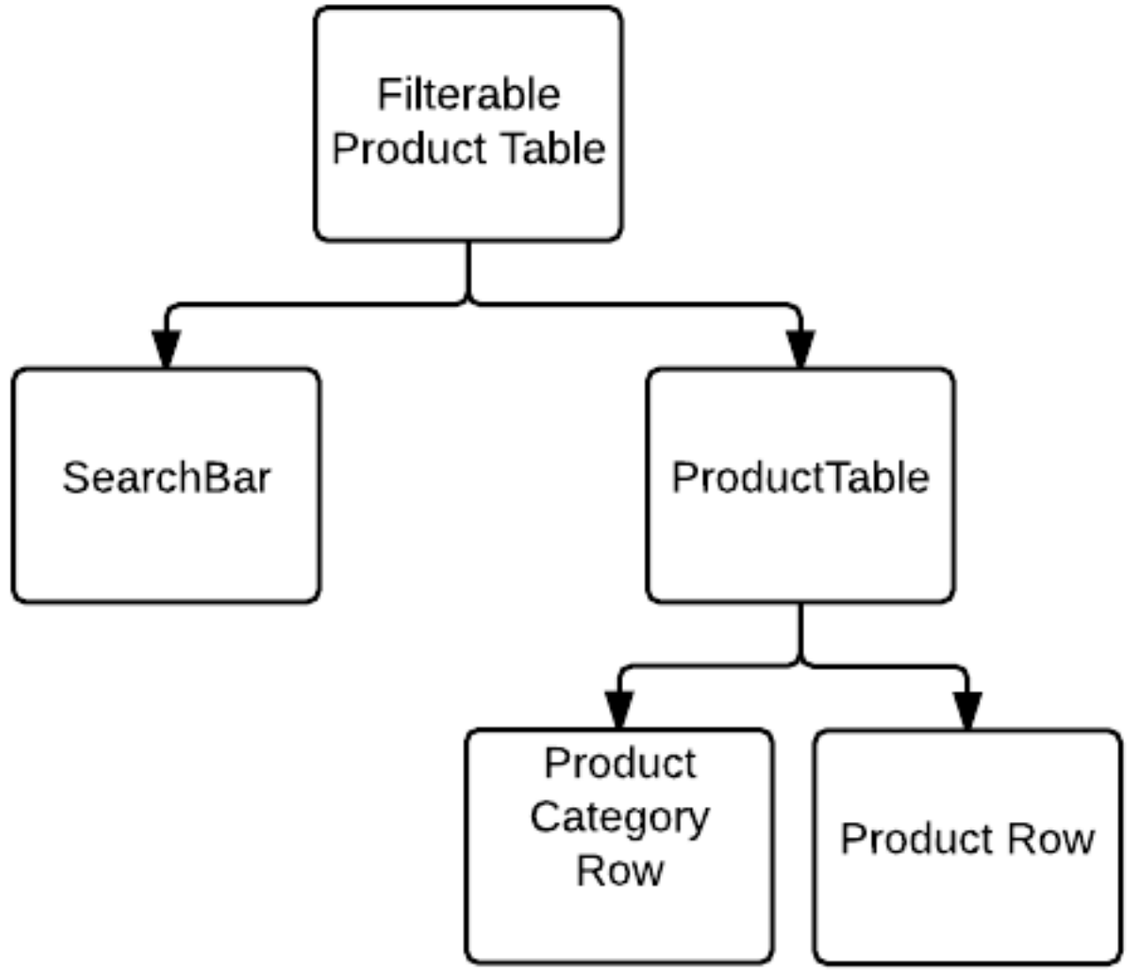


Des composants



Des composants





Composant basique

```
import React, {Component} from "react";
import ReactDOM from "react-dom";

class HelloWorld extends Component {
    render() {
        return (
            <div>
                Hello World!
            </div>
        );
    }
}

ReactDOM.render(
    <HelloWorld />,
    document.getElementById("root")
);
```

Syntaxe JSX

```
// Before
const MyComponent = (props) => (
    <div>Hello World!</div>
);

ReactDOM.render(
    <MyComponent />,
    document.getElementById("root")
);

//After
const MyComponent = (props) => (
    React.createElement("div", null, "Hello World")
);

ReactDOM.render(
    React.createElement(MyComponent),
    document.getElementById("root")
);
```

Avec JSX

Sans JSX

Deux façons de gérer les données

- Données qui changent (mutable) :
  on utilise un état (state)

- Données qui ne changent pas (immutable) :
  on utilise des propriétés (props)

- On essaie de minimiser les données qui changent quitte à refaire des calculs

Modifier un état

```
class Counter extends React.Component {
    state = {counter : 0}

    onClick = () => {
        this.setState({counter : this.state.counter + 1});
    }

    render() {
        const {counter} = this.state;

        return (
            <div>
                Button was clicked:
                <div>{counter} times</div>

                <button onClick={this.onClick} >
                    Click Me
                </button>
            </div>
        );
    }
}
render(<Counter />);
```

Button was clicked:
2 times
Click Me

Changer le state est asynchrone,
Le récupérer est synchrone

Composants conteneur/présentation

Pattern React:

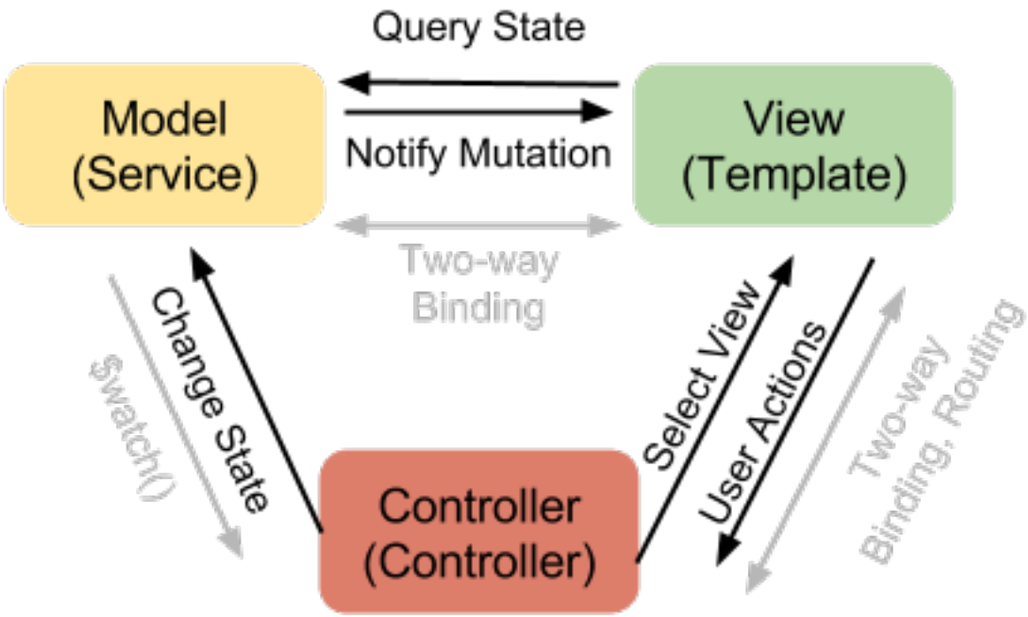
- Composant conteneur
  récupère les données et les passe en props à un composant enfant de présentation
- Compostant présentation
  s'occupe du rendu de l'interface en utilisant le prop fournit par le parent (pas de logique)

```
// Presentational component: simply displays supplied data
const SpeakerListItem = ({speaker, selected, onClick}) => {
    const itemOnClick = () => onClick(speaker);

    const content = selected ? <b>{speaker}</b> : speaker;
    return <li onClick={itemOnClick}>{content}</li>;
}

// Container component: controls data and passes it down
class ListSelectionExample extends React.Component {
    state = {speakers : allSpeakers, selectedSpeaker : null}

    render() {
        const {speakers, selectedSpeaker} = this.state;

        const speakerListItems = speakers.map(speaker => (
            <SpeakerListItem
                key={speaker}
                speaker={speaker}
                selected={speaker === selectedSpeaker}
                onClick={this.onSpeakerClicked}
            />
        ));

        return (<div><ul>{speakerListItems}</ul></div>);
    }
}
```

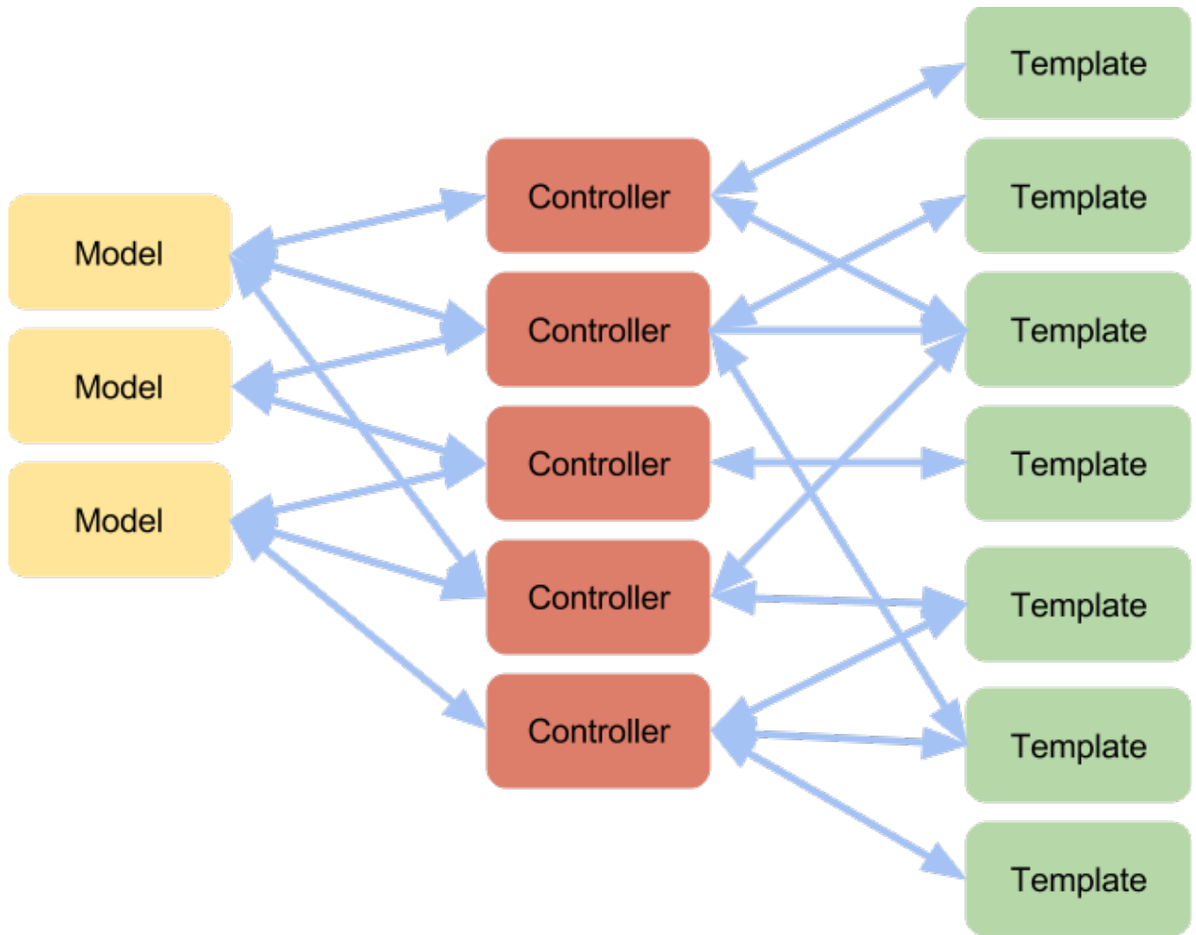
Plan

- Introduction
- Les principes de la programmation réactive
- En pratique les transformations de flux
- React
- **Redux**

MVC et MVVM https://medium.com/@davidsouther/song-flux-e1f9786579f6



MVC

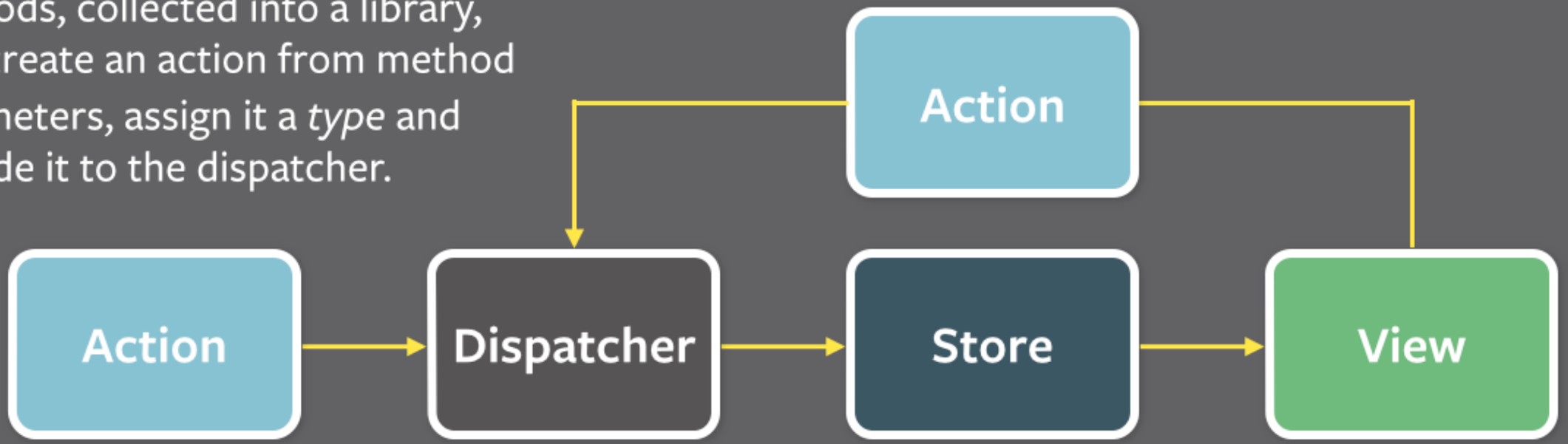


MVVM

Two-way data binding: bien jusqu'à ce que l'application devienne énorme et qu'on arrive plus à suivre les changements d'état

Le principe : un flux unidirectionnel

Action creators are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.

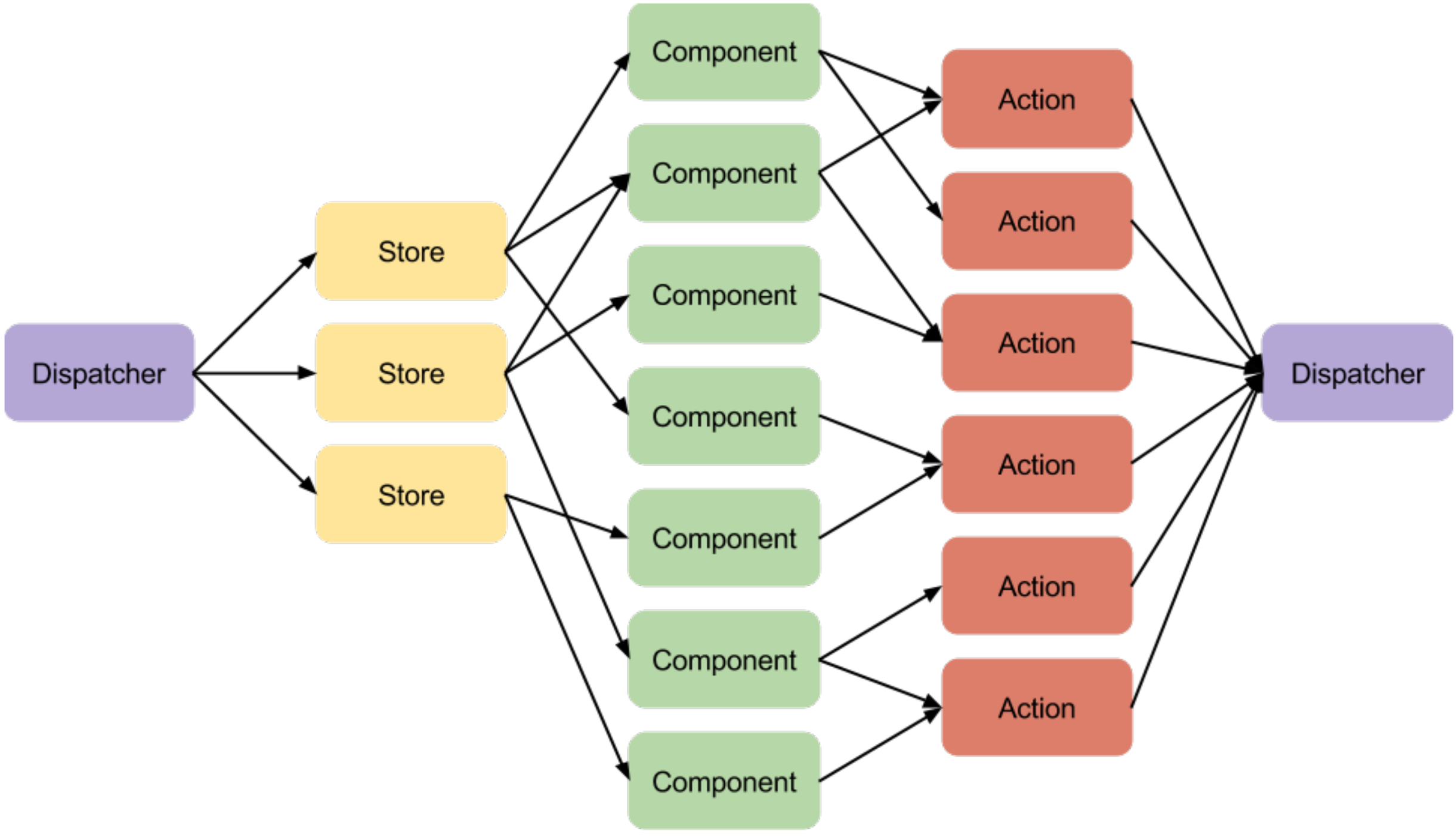


Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

En pratique sur une application



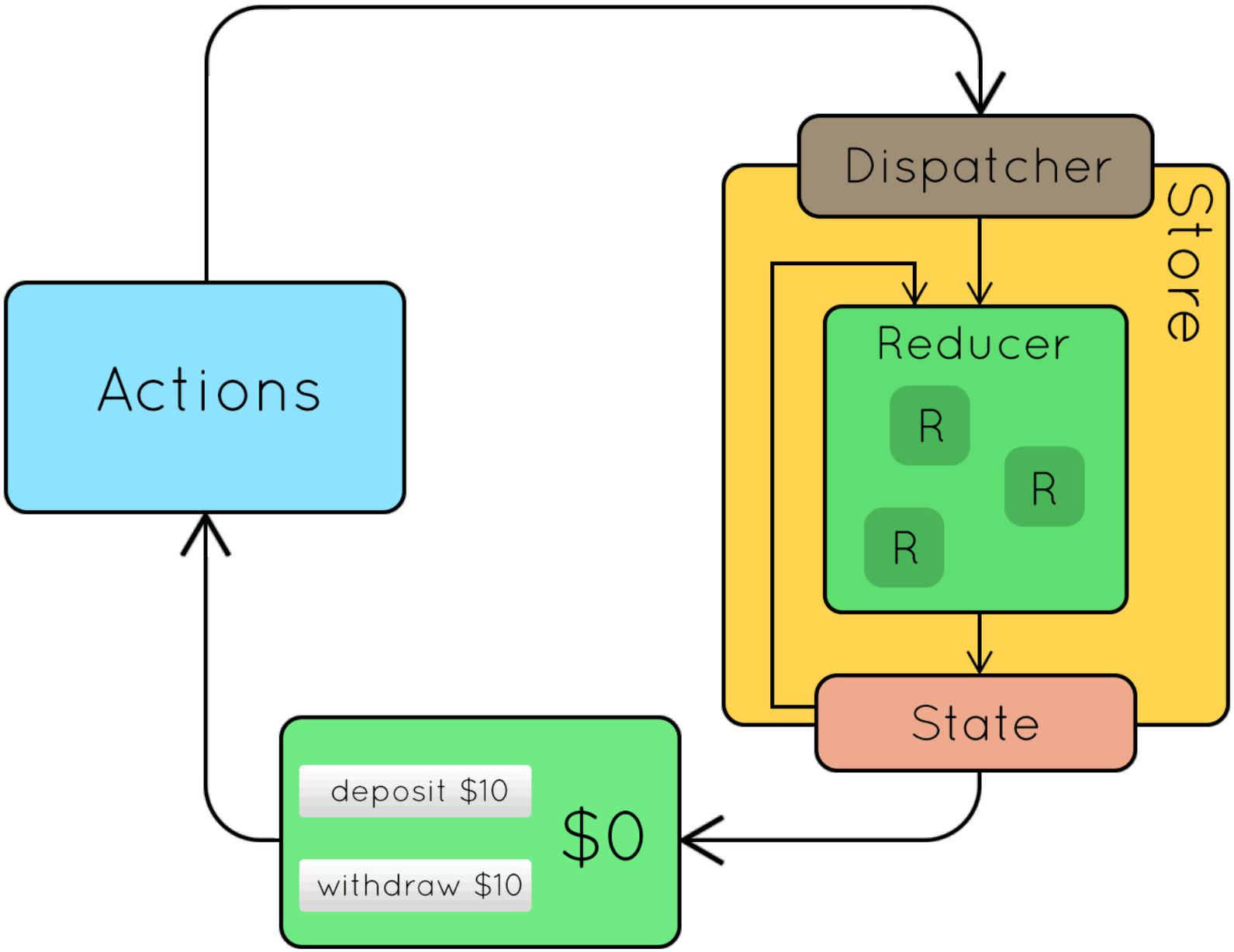
Redux : une implémentation de l'archi. Flux

Prévisible

- **Source unique de vérité**: l'état de toute l'application est stocké dans **un store.**
- **État en lecture seule**: les changements d'états sont causés par une **action**, le reste de l'application ne peut changer l'état.
- **Les changement sont des fonctions**, ces fonctions s'appellent **reducers** et sont de la forme suivante: `(state, action) => newState`

Centralisé, un seul store et arbre d'état permet: logging des changements, gestion d'API, undo/redo, …

Redux one-way data flow



Concepts de Redux

```
// App state: a plain object with many keys or "slices"
{
    todos: [{
        text: "Eat food",
        completed: true
    }, {
        text: "Exercise",
        completed: false
    }],
    visibilityFilter : "SHOW_COMPLETED"
}

// Actions: plain objects with a "type" field
{ type: "ADD_TODO", text: "Go to swimming pool" }
{ type: "TOGGLE_TODO", index: 1 }
{ type: "SET_VISIBILITY_FILTER", filter: "SHOW_ALL" }


// Action creators: functions that return an action
function addTodo(text) {
    return {
        type : "ADD_TODO",
        text
    };
}
```

State (état)

- Objets basiques

Actions

- Pour changer un état on déclenche une action. Un objet simple avec un type.

Action creators

- Encapsule la création d'actions. Pas nécessaire mais bonne pratique

Reducers

```
function visibilityReducer(state = "SHOW_ALL", action) {
    return action.type === "SET_VISIBILITY_FILTER" ?
        action.filter :
        state
}

function todosReducer(state = [], action) {
    switch (action.type) {
        case "ADD_TODO":
            return state.concat([{
                text: action.text, completed: false
            }]);
        case "TOGGLE_TODO":
            return state.map((todo, index) => {
                if(index !== action.index) return todo;
                return { text: todo.text, completed: !todo.completed }
            })
        default: return state;
    }
}

function todoApp(state = {}, action) {
    return {
        todos: todosReducer(state.todos, action),
        visibilityFilter: visibilityReducer(state.visibilityFilter, action)
    };
}
```

Les Reducers sont des fonctions pures, = sans effets de bord

(state, action) => newState

Mettent à jour les données en copiant l'état et en modifiant la copie, avant de la renvoyer (immuabilité)

Store

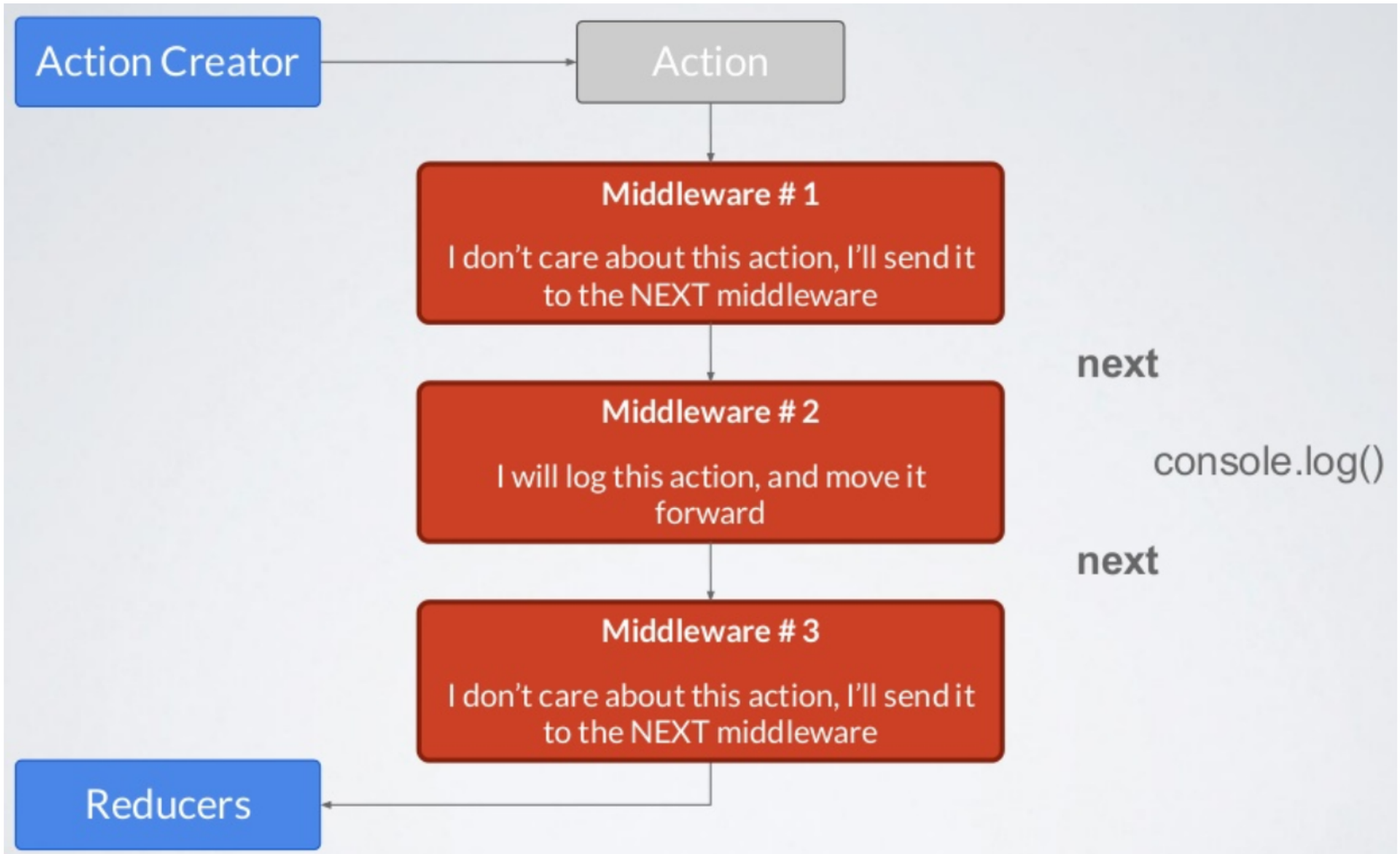
```
import {createStore} from "redux";

import rootReducerFunction from "reducers/todoApp";

const store = createStore(rootReducerFunction, preloadedState);

console.log(store.getState());
// {todos : [.....], visibilityFilter : "SHOW_COMPLETED"}

store.dispatch({ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' })
console.log(store.getState());
// {todos : [.....], visibilityFilter : "SHOW_ALL"}

const stateBefore = store.getState();
console.log(stateBefore.todos.length);
// 2

store.subscribe( () => {
    console.log("An action was dispatched");
    const stateAfter = store.getState();
    console.log(stateAfter.todos.length);
});

store.dispatch({ type: 'ADD_TODO', text: 'Go to swimming pool' });
// "An action was dispatched"
// 3
```

Un store Redux contient l'état courant.

Les stores ont 3 méthodes principales:

- dispatch
- getState
- subscribe

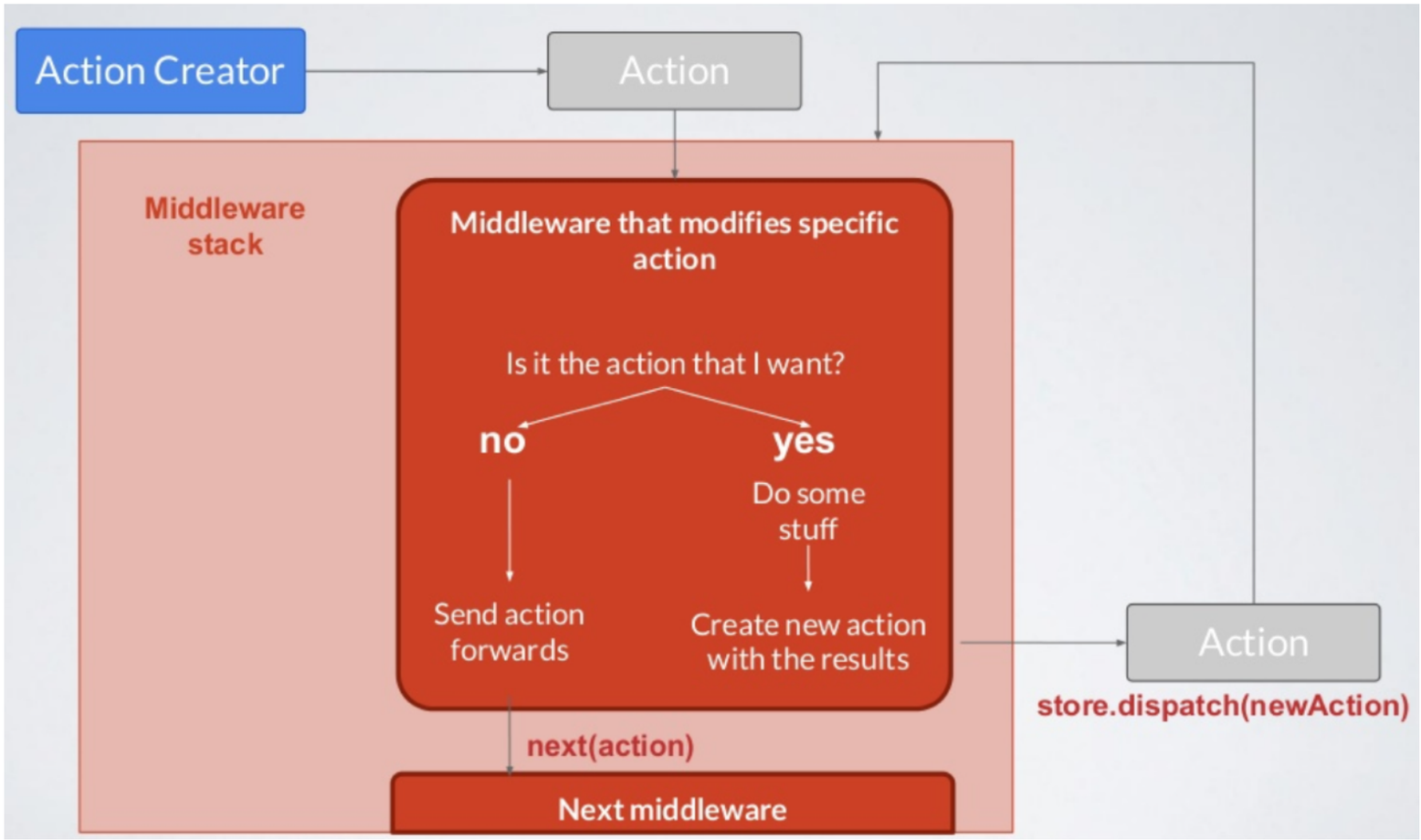
Redux Middleware



Un middleware permet de faire tourner du code après un dispatch mais avant qu'elle atteigne le reducer.

Ils peuvent être chainés

Redux Middleware



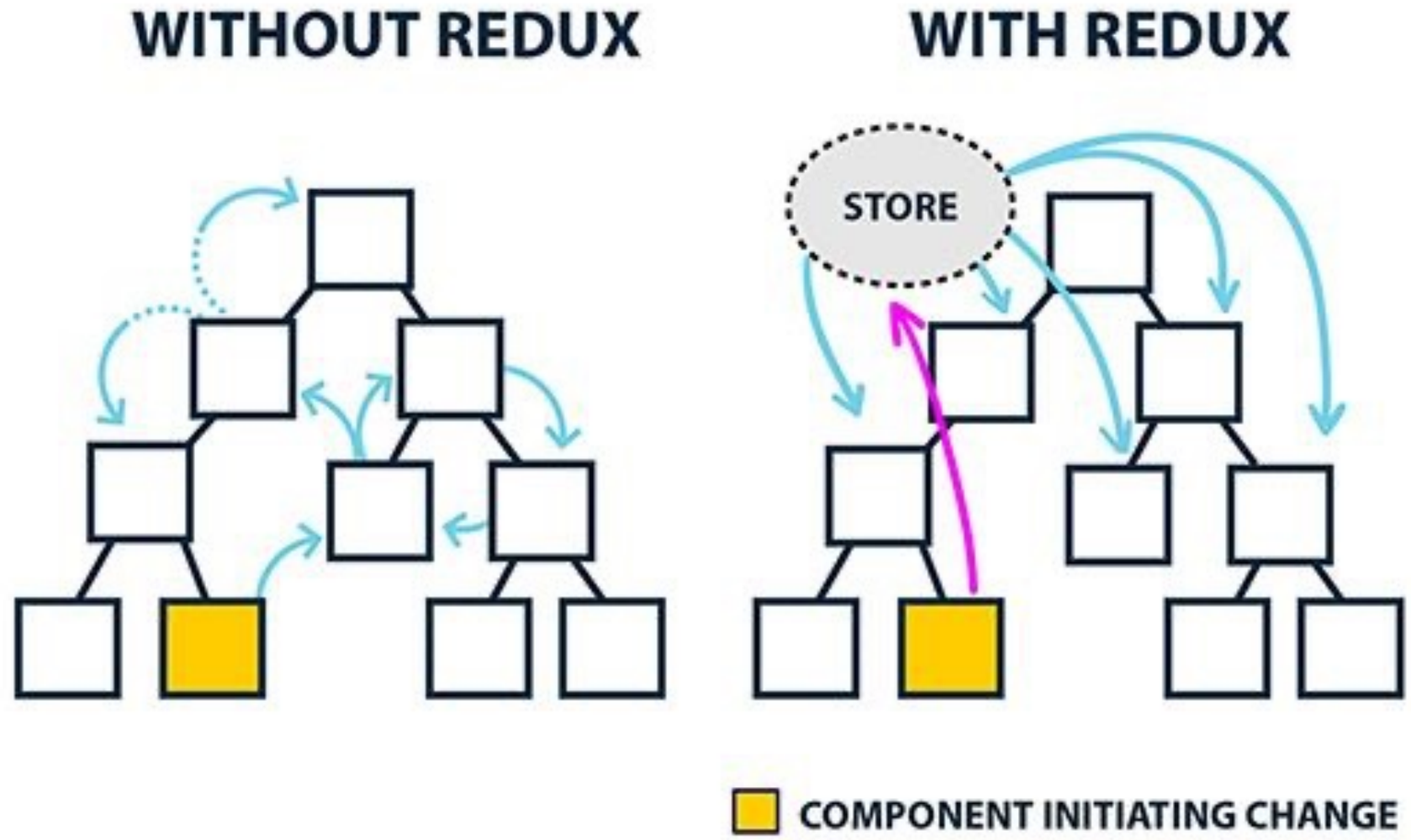
Permet d'inspecter les actions, les modifier, les stopper, en déclencher d'autres…

-> gérer la persistance avec le serveur

-> partager des actions via websockets

Pourquoi Redux

https://www.foreach.be/blog/why-the-react-redux-combo-works-like-magic



Pourquoi utiliser Redux ?

Les composants React gère déjà leur état interne.

Redux :

1. Gestion centralisée des états
2. Si plusieurs composants partagent les mêmes données, les stocker dans redux permet une meilleure gestion
3. Time-travel debugging (on peut revenir à des états passés)
4. Hot reloading pour le dev
   sans Redux: modif de composant -> état perdu

Services utilisant React+Redux

- Twitter (mobile site)
- Instagram (mobile app)
- Reddit (mobile site)
- Wordpress (Calypso admin panel)
- Jenkins (BlueOcean control panel)
- Mozilla Firefox (DevTools)
- …

Ressources

React / redux

- https://www.valentinog.com/blog/redux/
- https://blog.isquaredsoftware.com/presentations/2018-03-react-redux-intro/
- https://elijahmanor.com/talks/react-to-the-future/dist/

Mobx, une alternative à Redux

- https://blog.logrocket.com/redux-vs-mobx/

Comparaison de Angular, React et Vue (par vue)

- https://vuejs.org/v2/guide/comparison.html