Programmation Réactive

Principes fondamentaux et application au Web

Plan

- Introduction
- Quelles limites de MVC
- Quelques principes généraux
- En pratique avec React
- Redux
- Traitements de flux génériques

Plan

- **Introduction**
- Quelles limites de MVC
- Quelques principes généraux
- En pratique avec React
- Redux
- Traitements de flux génériques

Qu'est ce que la programmation réactive ?

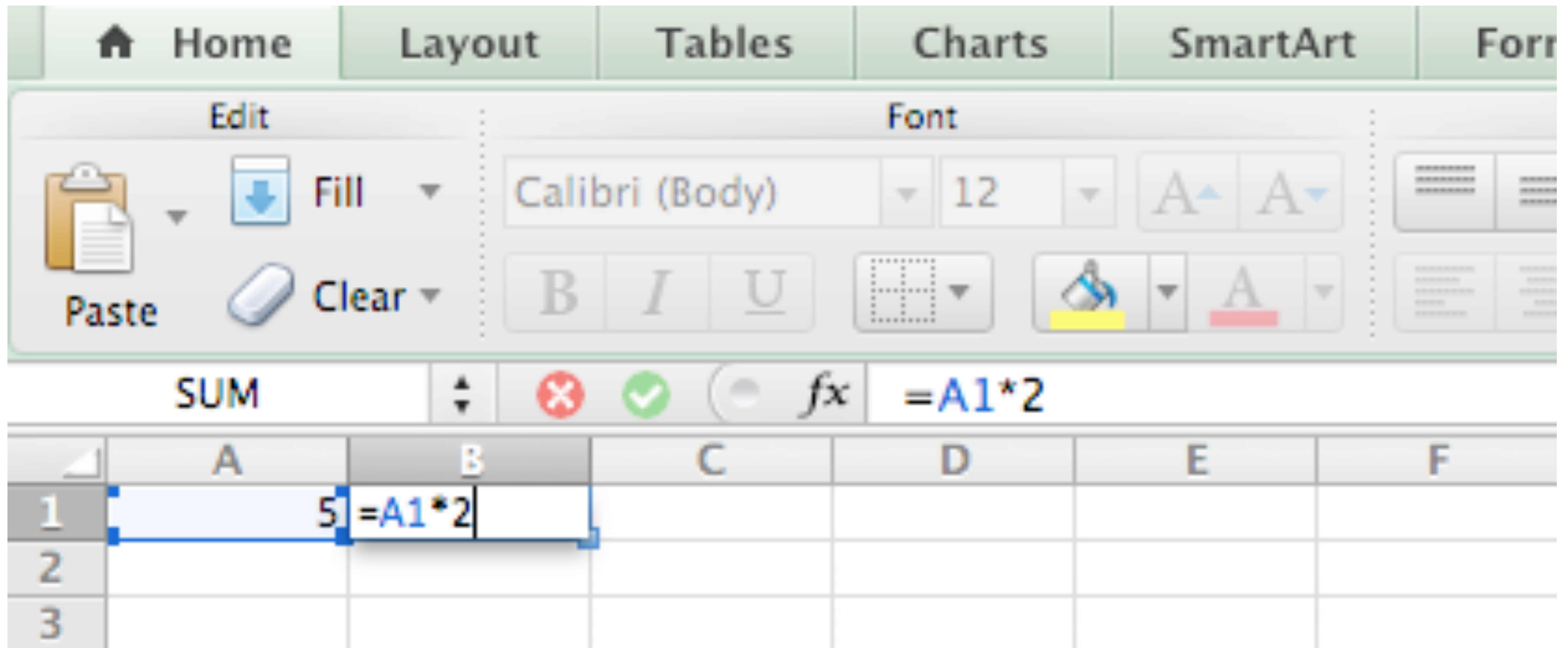
Une approche visant à mieux gérer les flux

Deux types de flux

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

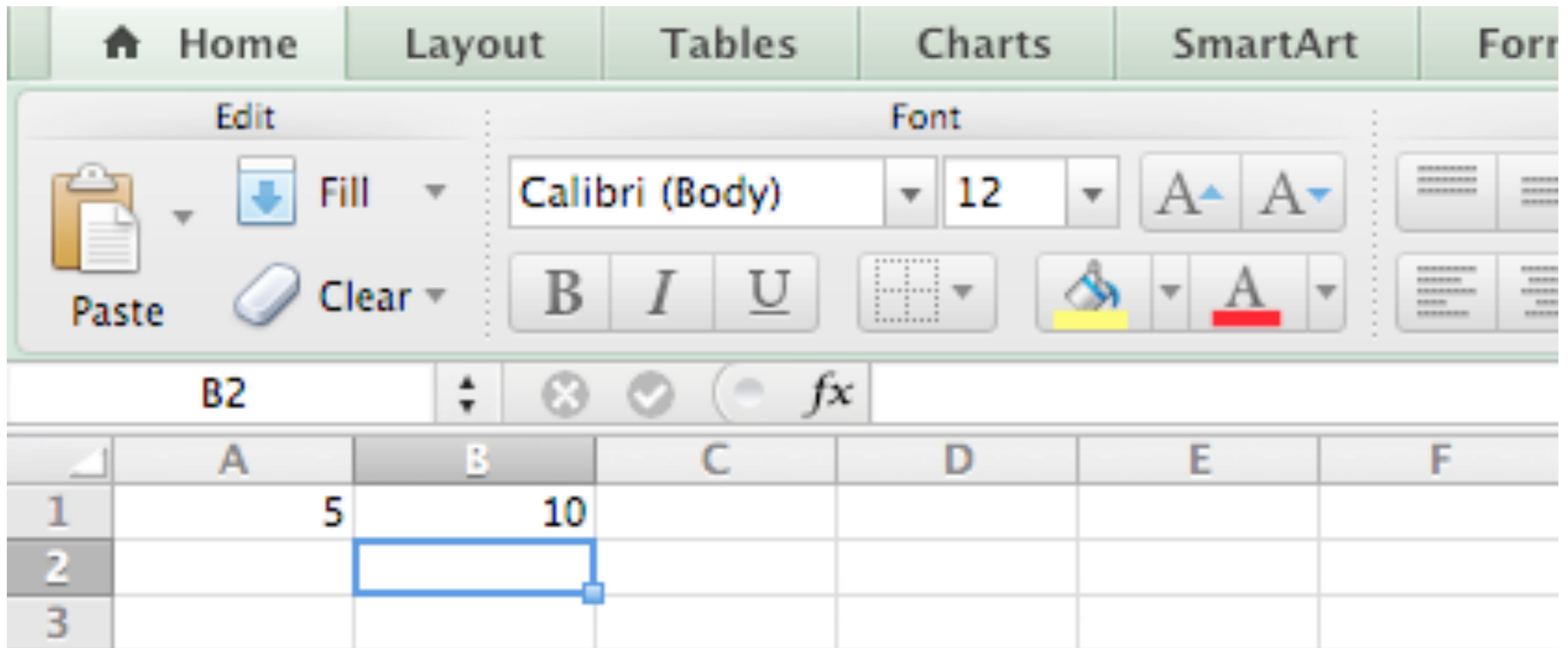
Idée : dépasser les callbacks ou le patron Observer.

Ou avez vous vu ça ?



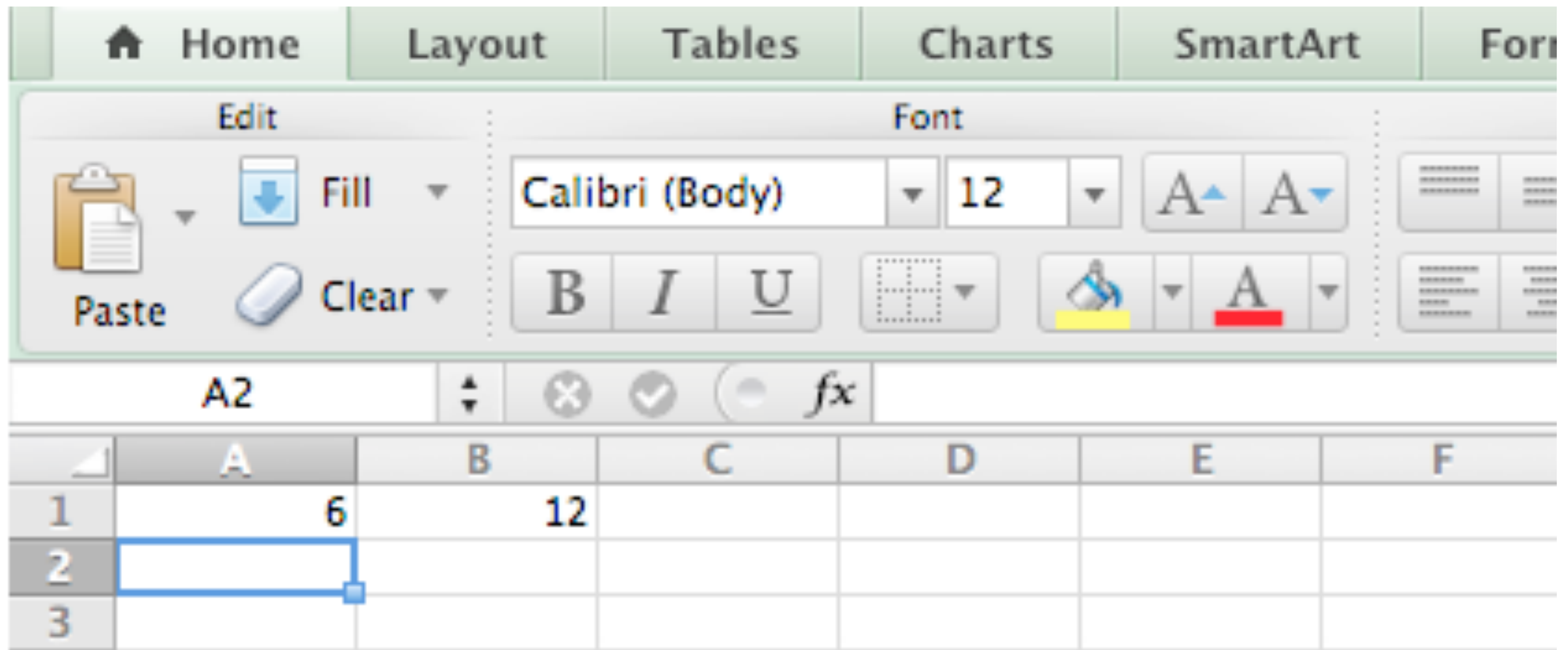
http://www.hanselsolutions.com/blog/surf-talk/shiny-surf.html#/9

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

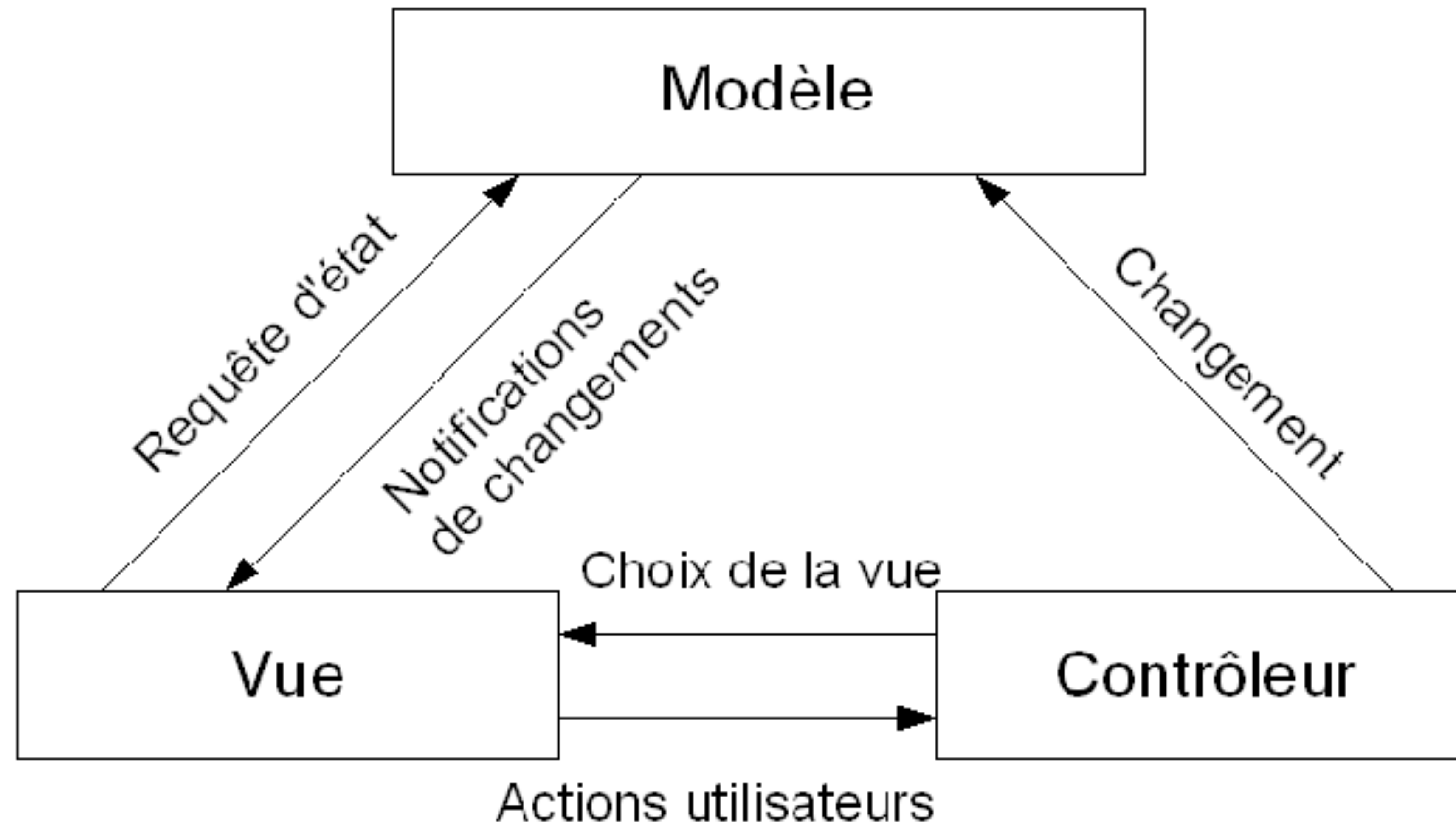
Exemples

À vous

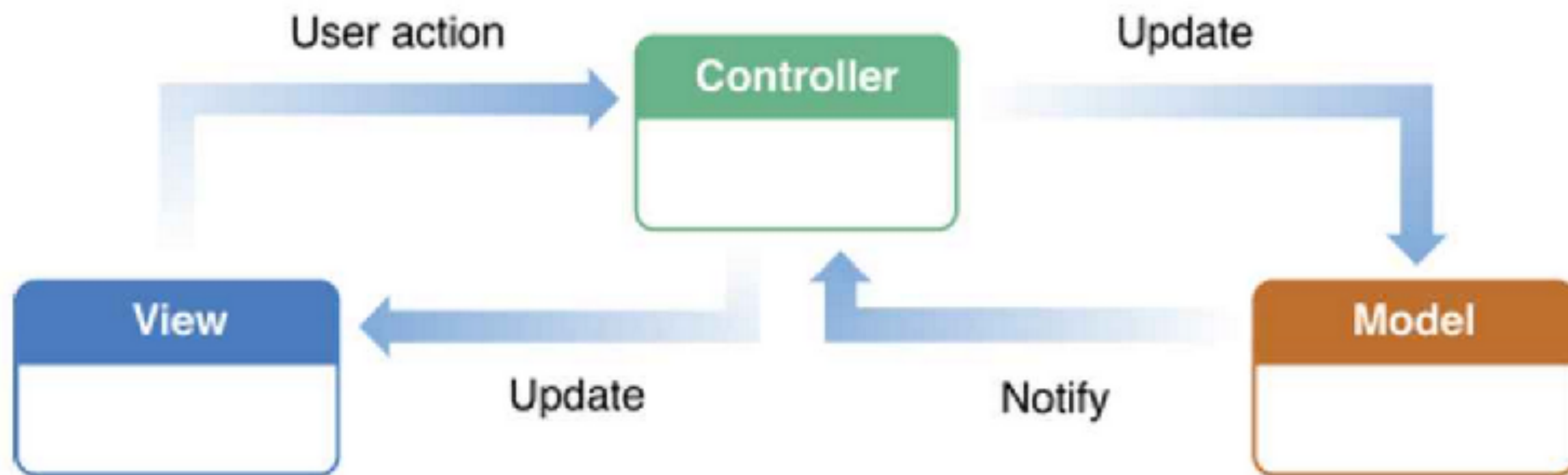
Plan

- Introduction
- **Quelles limites de MVC**
- Quelques principes généraux
- En pratique avec React
- Redux
- Traitements de flux génériques

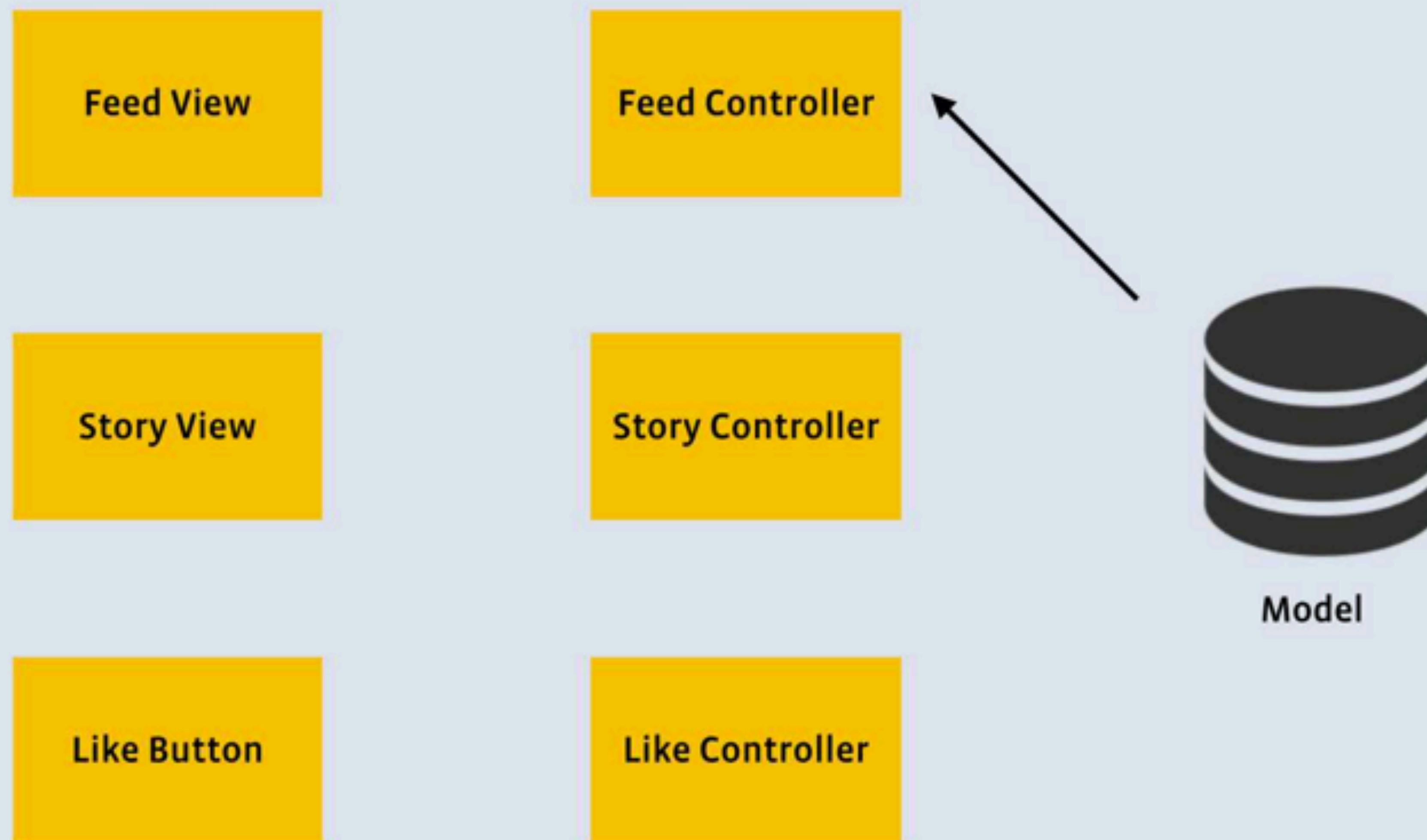
MVC - plutôt natif



MVC - plutôt Web

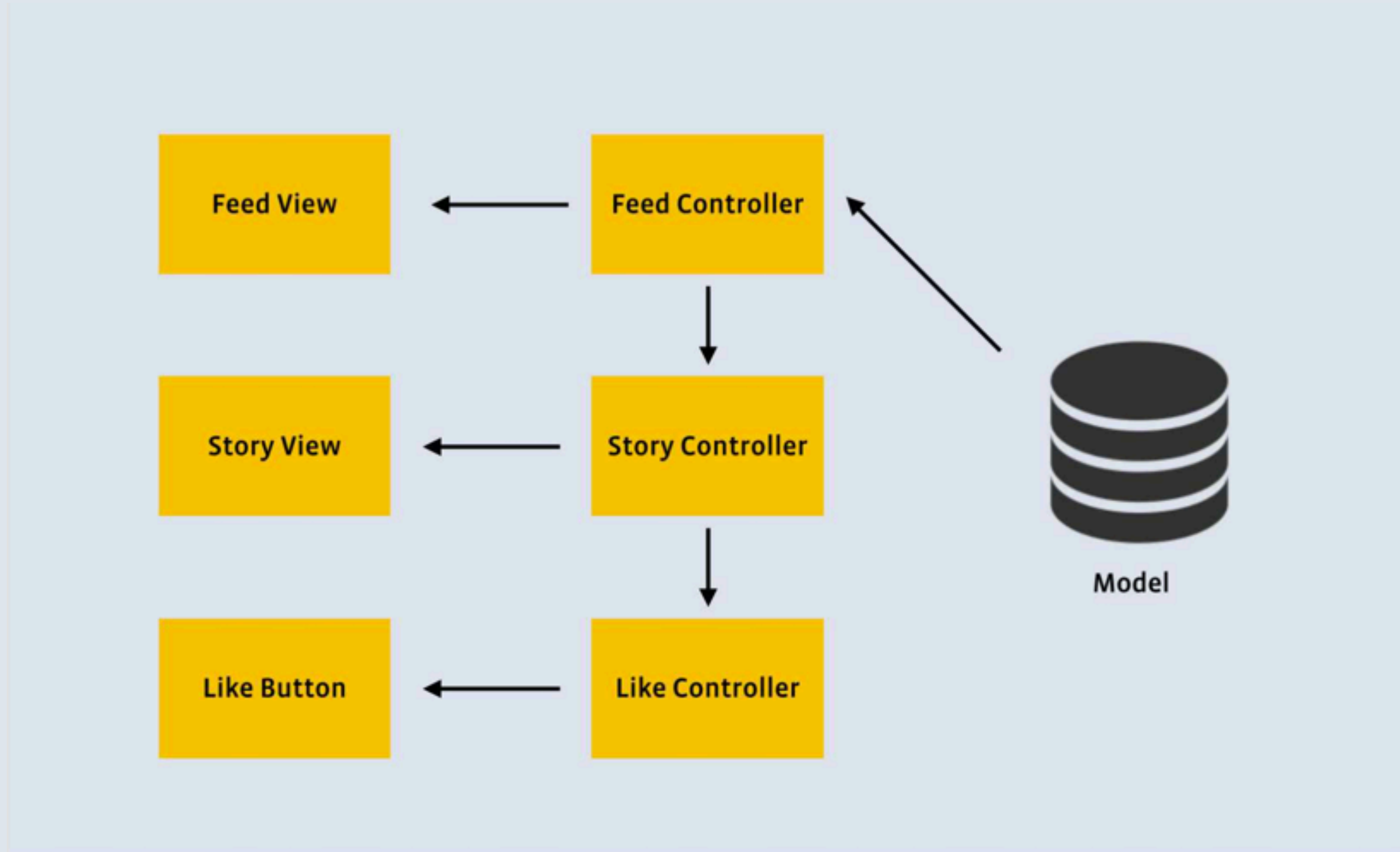


MVC en pratique



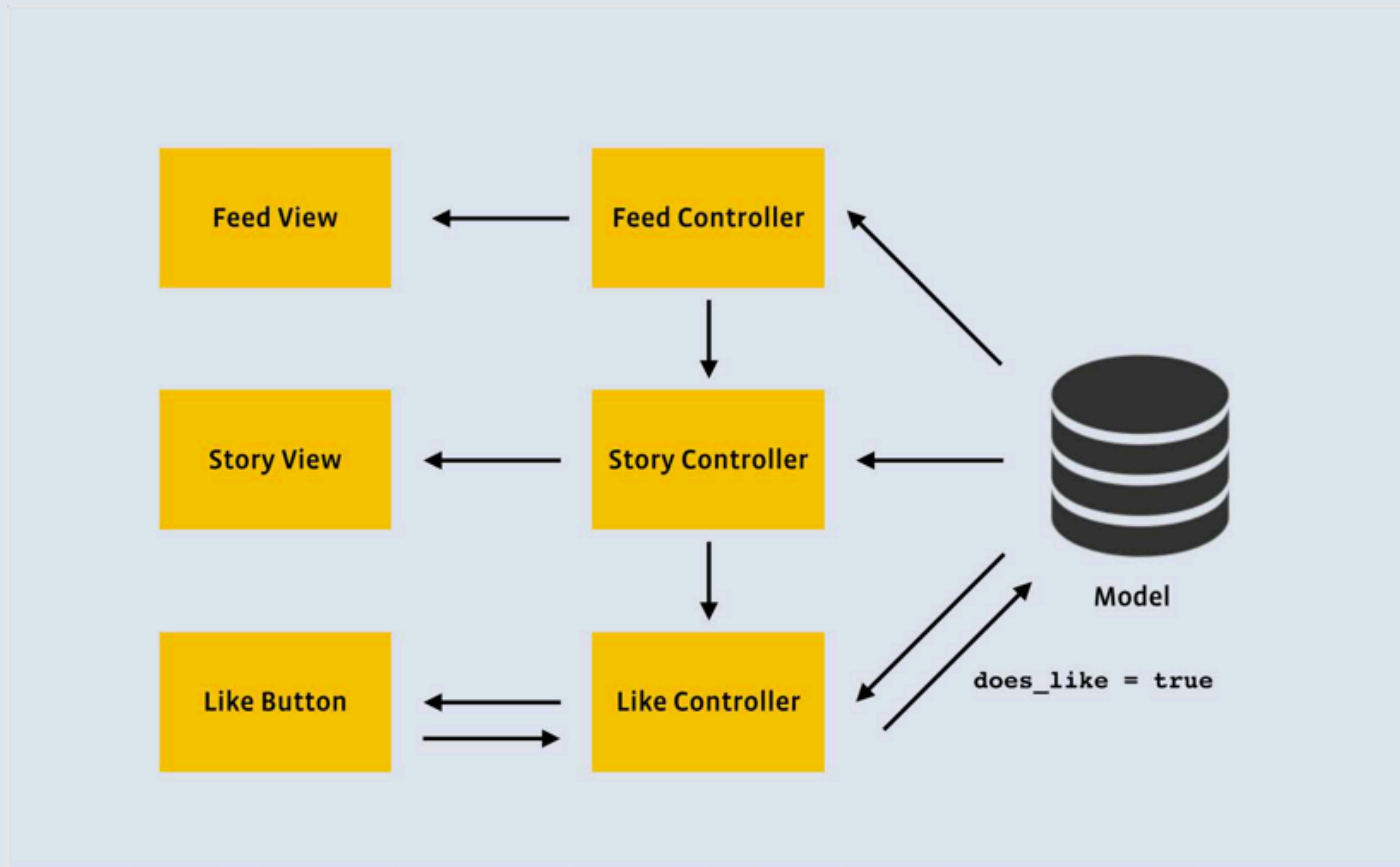
https://www.infoq.com/presentations/fp-facebook-mobile-apps/

MVC en pratique

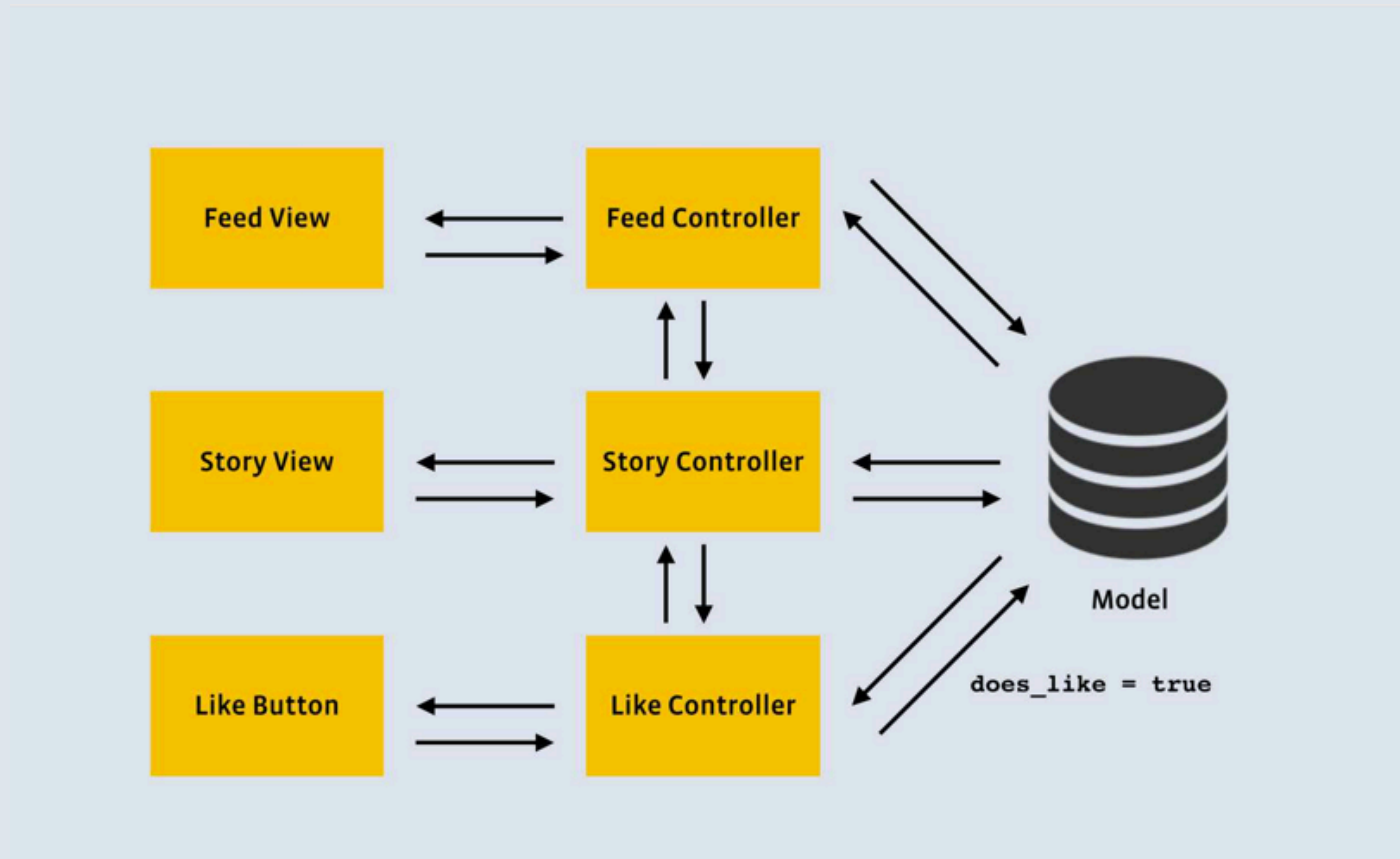


https://www.infoq.com/presentations/fp-facebook-mobile-apps/

MVC en pratique



MVC en pratique



https://www.infoq.com/presentations/fp-facebook-mobile-apps/

En pratique

Problèmes

1. La vue gère son état “en interne” -> elle est mutable mais influe sur le modèle quand même.

2. Un changement implique une cascade d’inter-dépendances

- Lenteurs (re-dessins multiples) sur le thread principal
- Race conditions, à cause d’opérations atomique
- Complexité et risque d’inter-blocage

Plan

- Introduction
- Quelles limites de MVC
- **Quelques principes généraux**
- En pratique avec React
- Redux
- Traitements de flux génériques

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

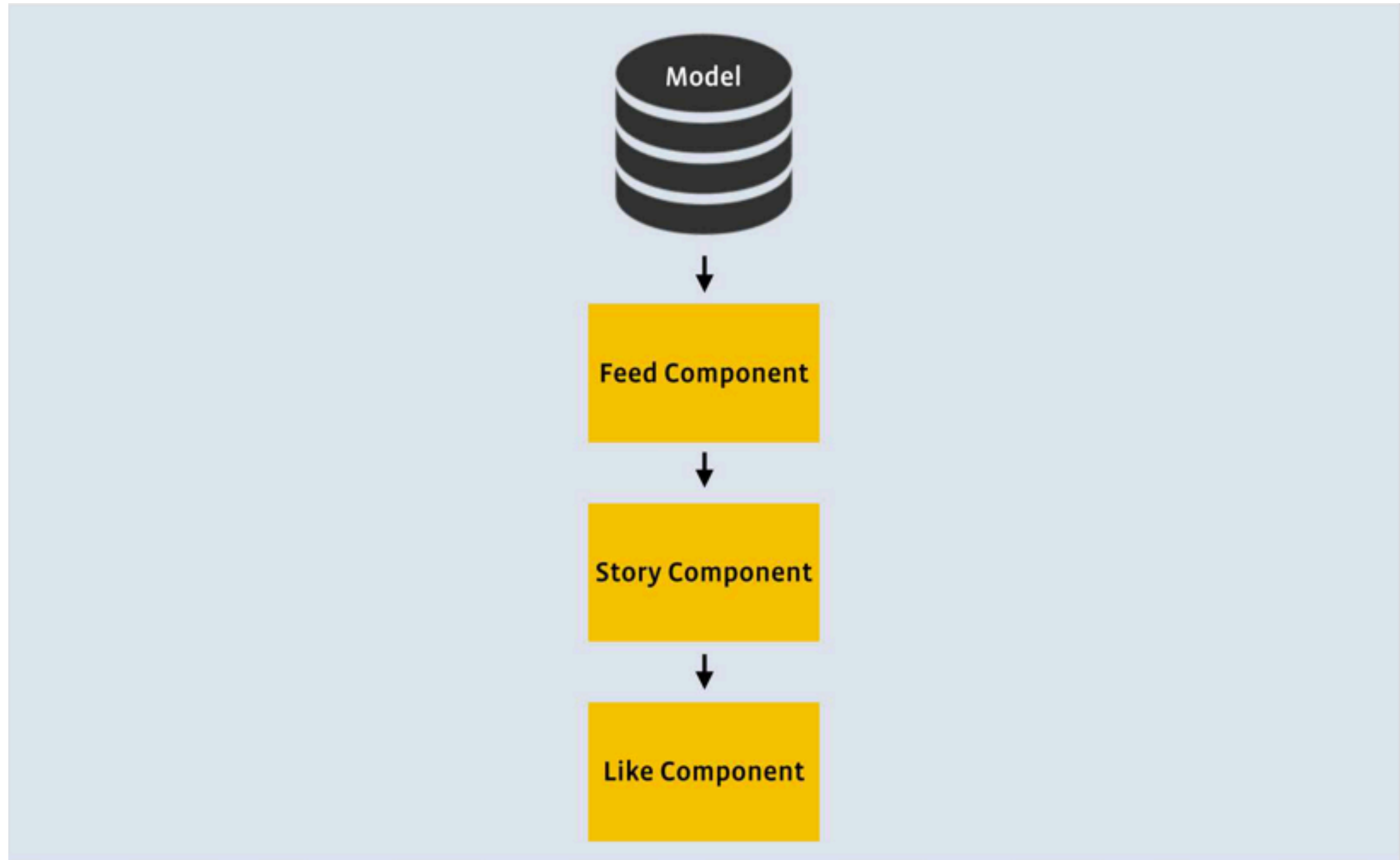
Facilite la prog. purement fonctionnelle (pratique pour plein de choses, évite les effets de bords, facilite le undo)

Une seule source de “vérité”

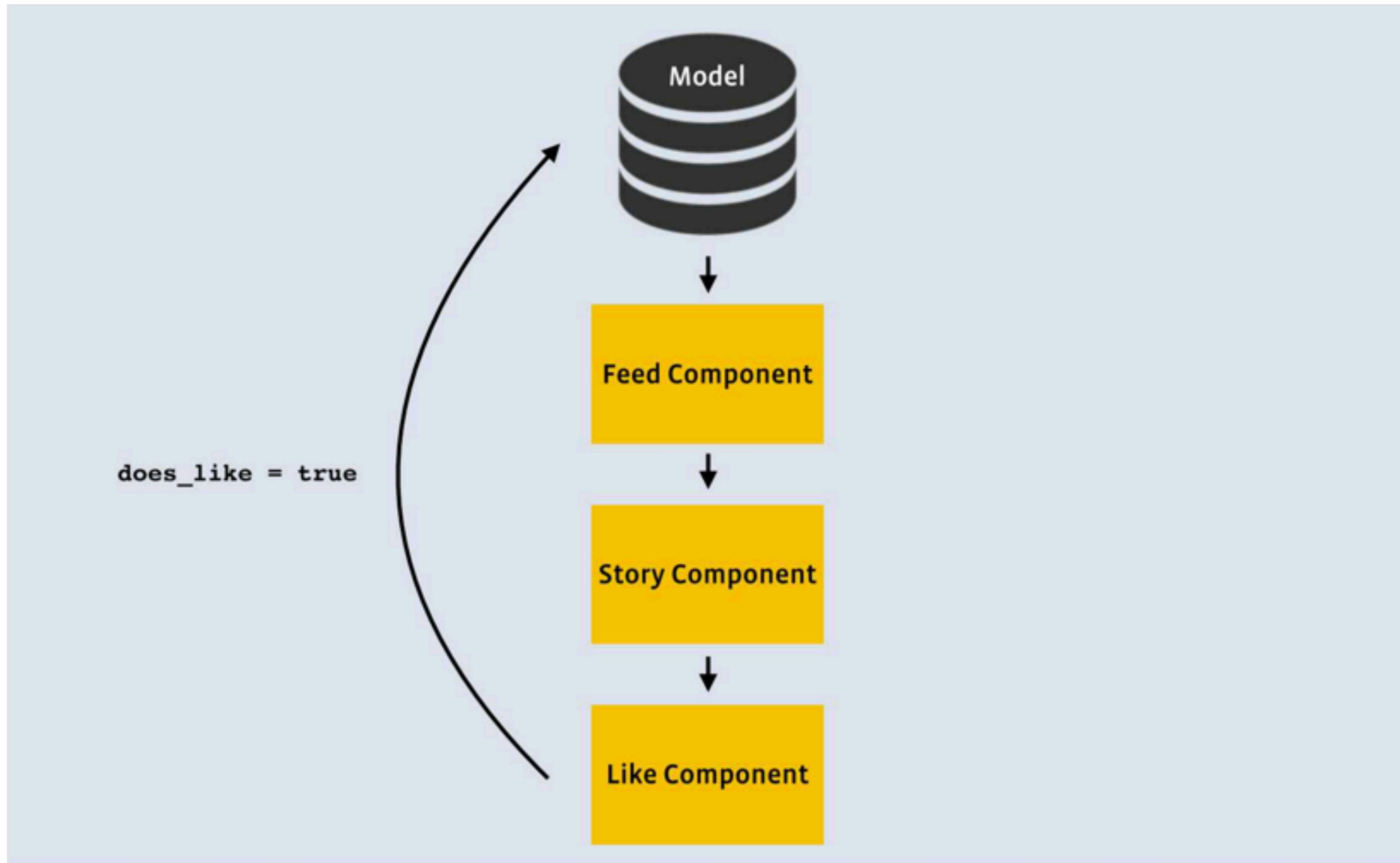
Facilite le caching

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

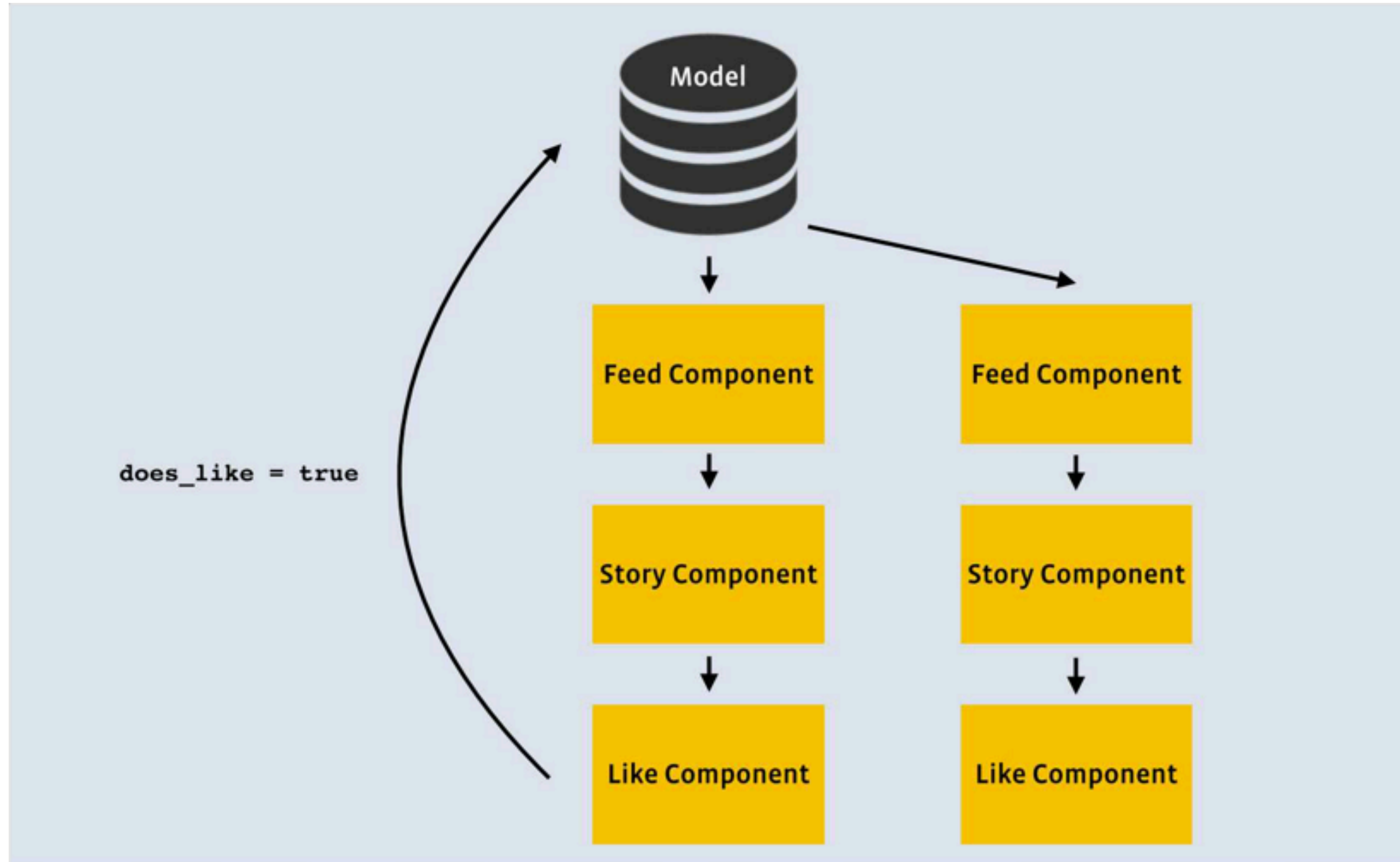
Principe généraux de React + Redux (Flux)



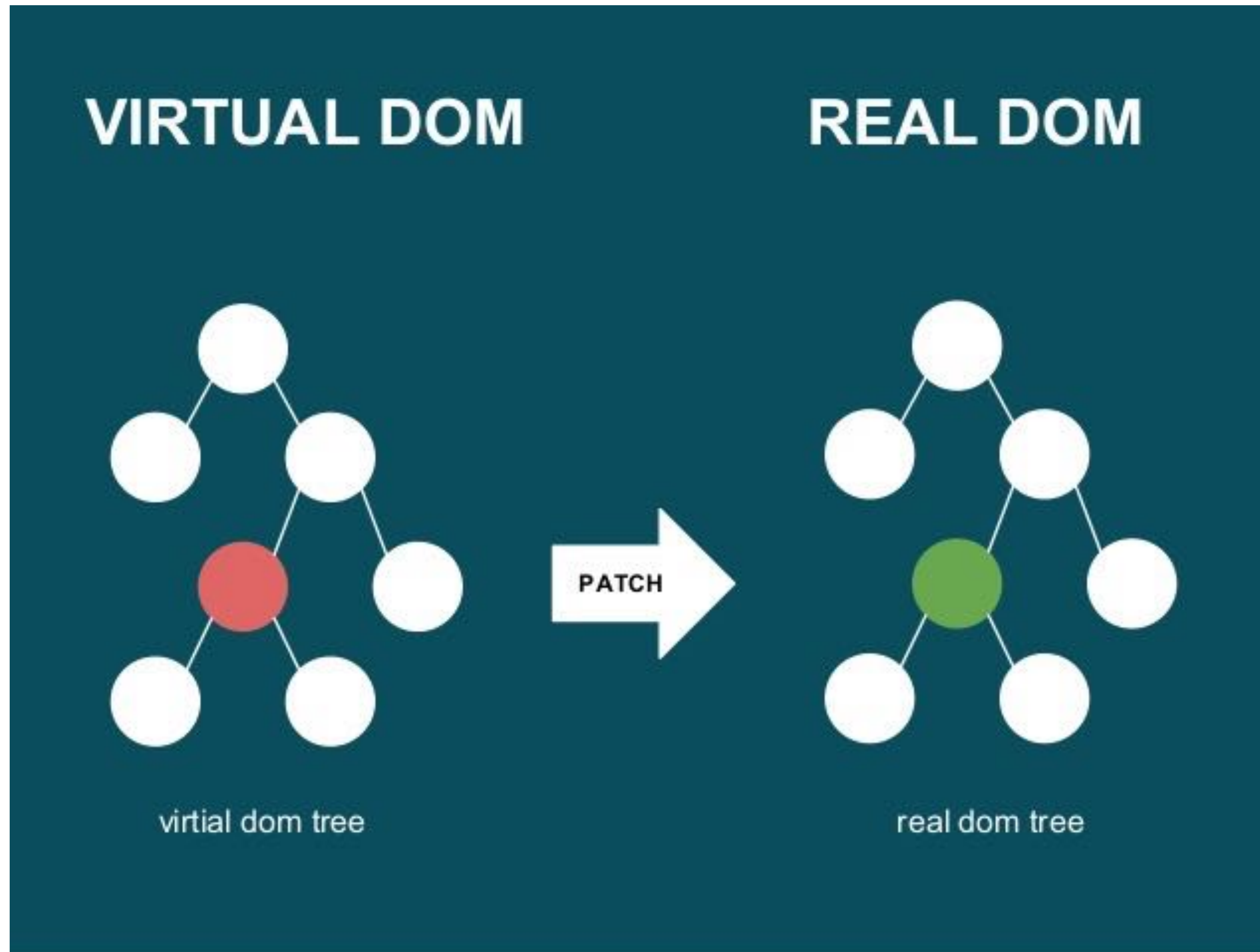
On modifie le modèle directement

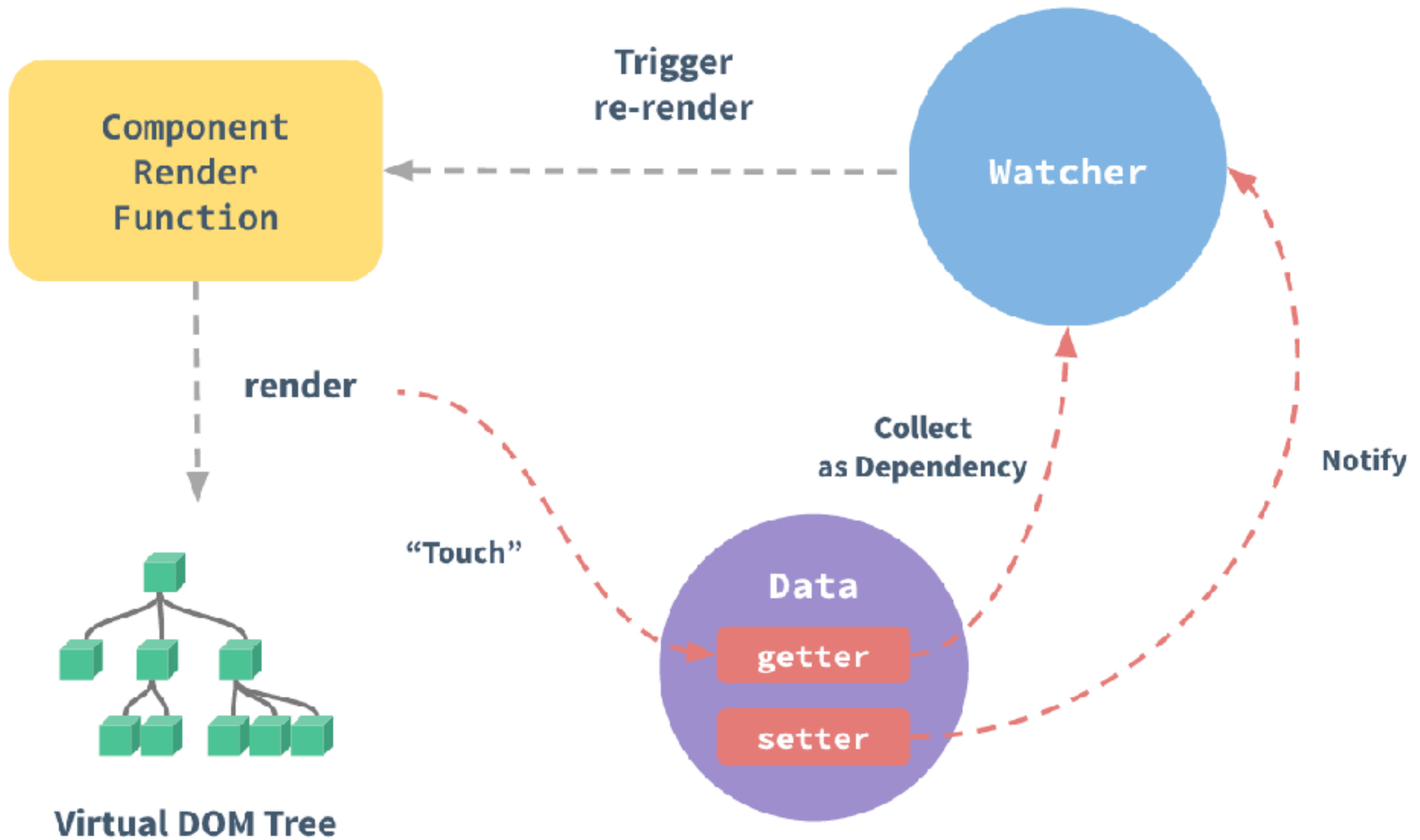


Nouvel arbre de rendu



Un DOM Virtuel




Trigger
re-render
Component
Render
Function
Watcher
render
Collect
as Dependency
Notify
"Touch"
Data
getter
setter
Virtual DOM Tree

Modèle immuable

- Les objets restent immuables
- On ne modifie pas le modèle mais on effectue des opérations dessus
- Quand un changement arrive, un nouvel arbre est créé depuis le haut
- Les “stores” de haut niveau reçoivent des mise à jour (updates) de façon asynchrone

Les bibliothèques Javascript

Guide API Examples Blog Community

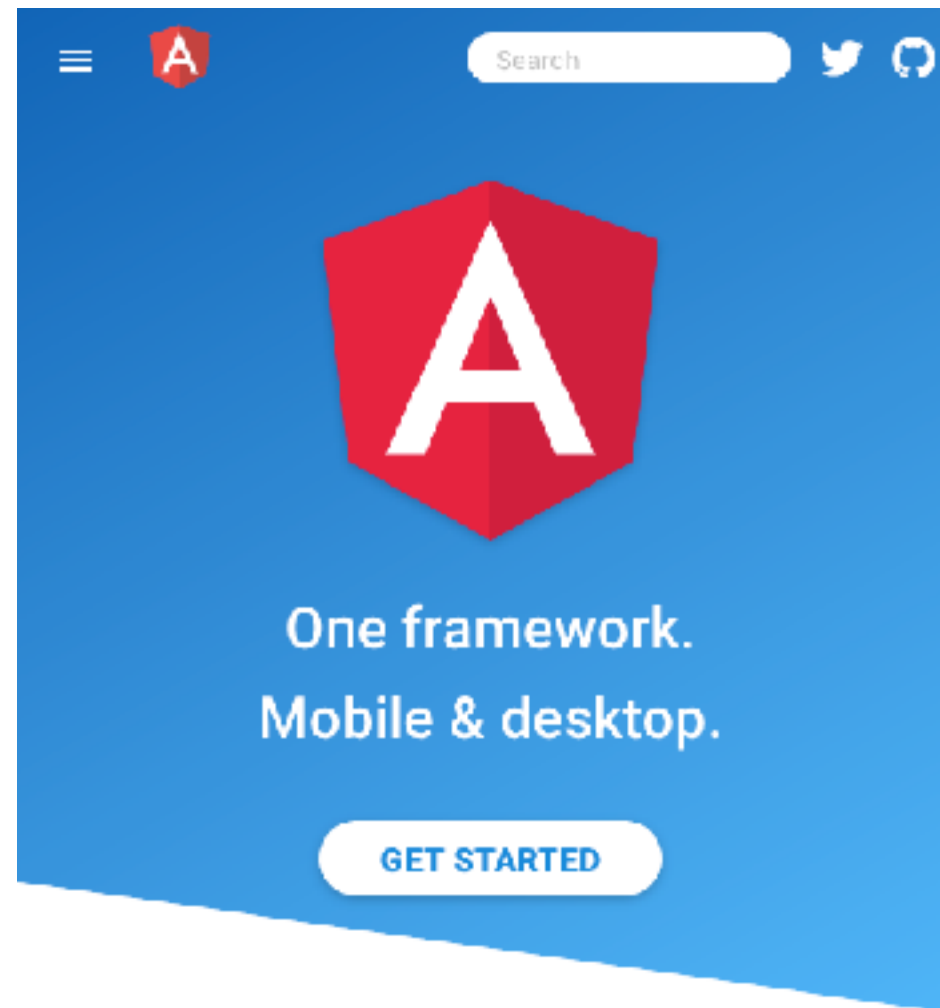
Vue.js

Reactive Components for Modern Web Interfaces

Install v1.0.24

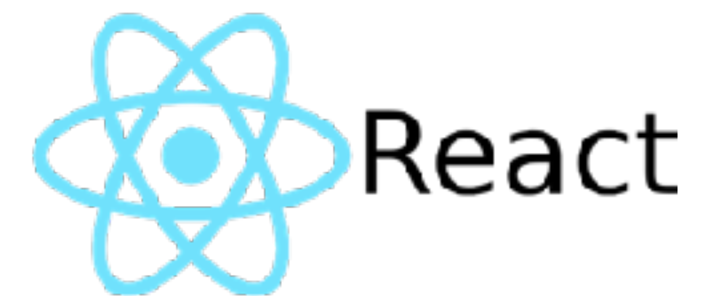
Follow @vuejs Star 18,631 Support Vue.js

中文 | 日本語 | Italiano



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
- Quelles limites de MVC
- Quelques principes généraux
- **En pratique avec React**
- Redux
- Traitements de flux génériques

React

Gère la vue

Quelques principes

- **Centré composant**
- Déclaratif
- Composition plutôt qu'héritage
- Encapsulation et réactivité

Des composants

Search...

Only show products in stock

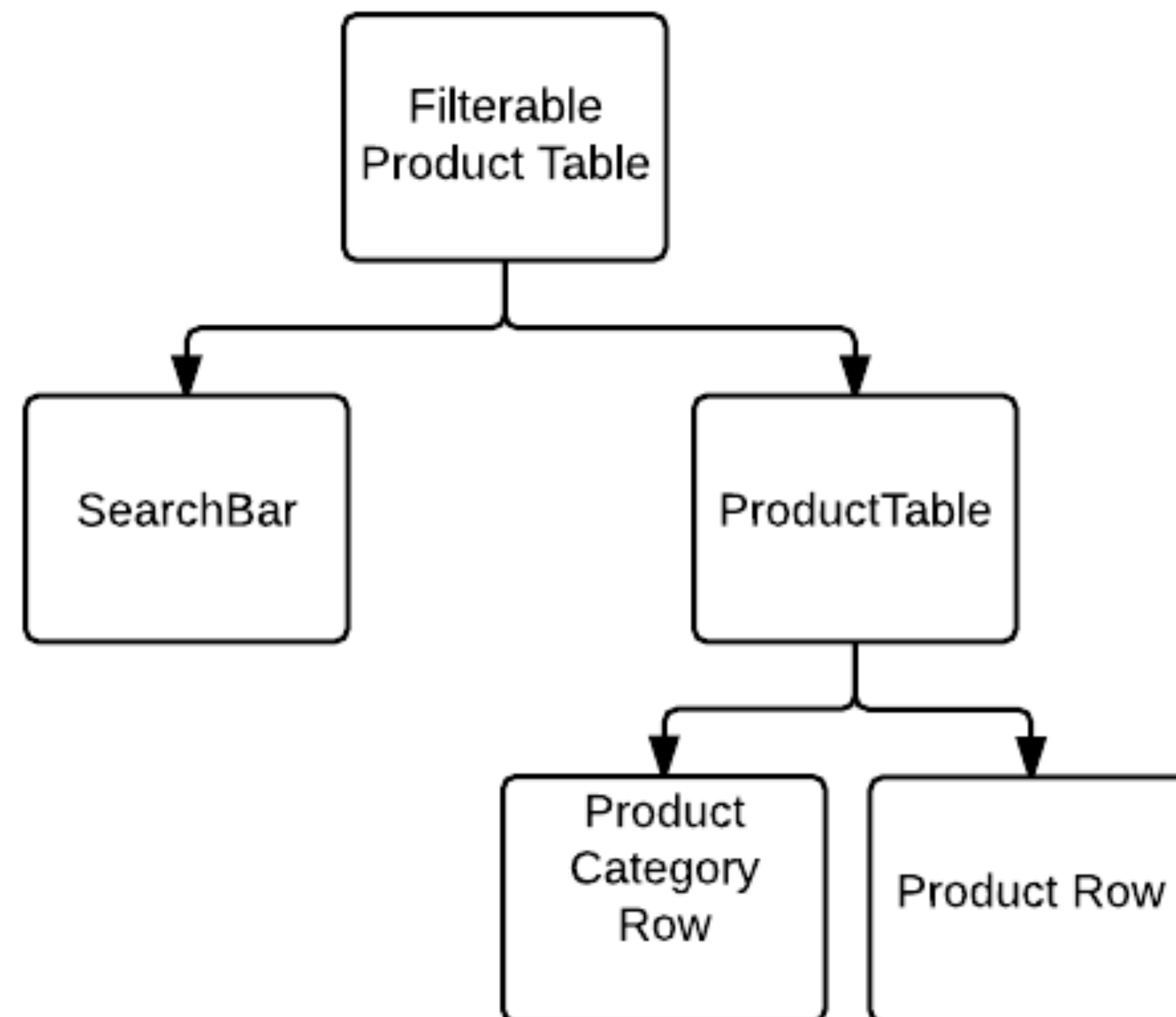
Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99

Des composants

Search...

Only show products in stock

Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99



Composant classe 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 pour un composant fonctionnel

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

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

JSX

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

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

Sans JSX

Composition

Skin: Material

20 Apr 2020 – 26 Apr 2020

il 21 | Wed, April 22 | Thu, April 23 | Fri, April 24

12:00 - 13:35 NEW EVENT

Description: New event

Event type: Select event type

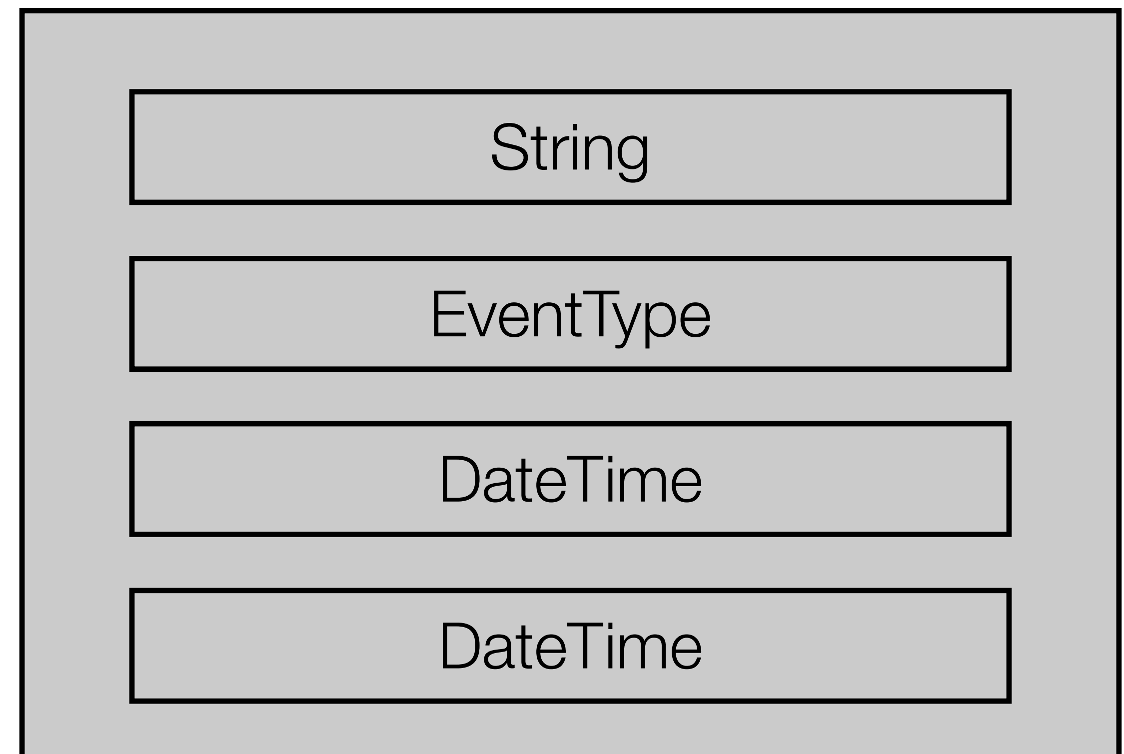
Time period: 12:00 26 April 2020

13:35 26 April 2020

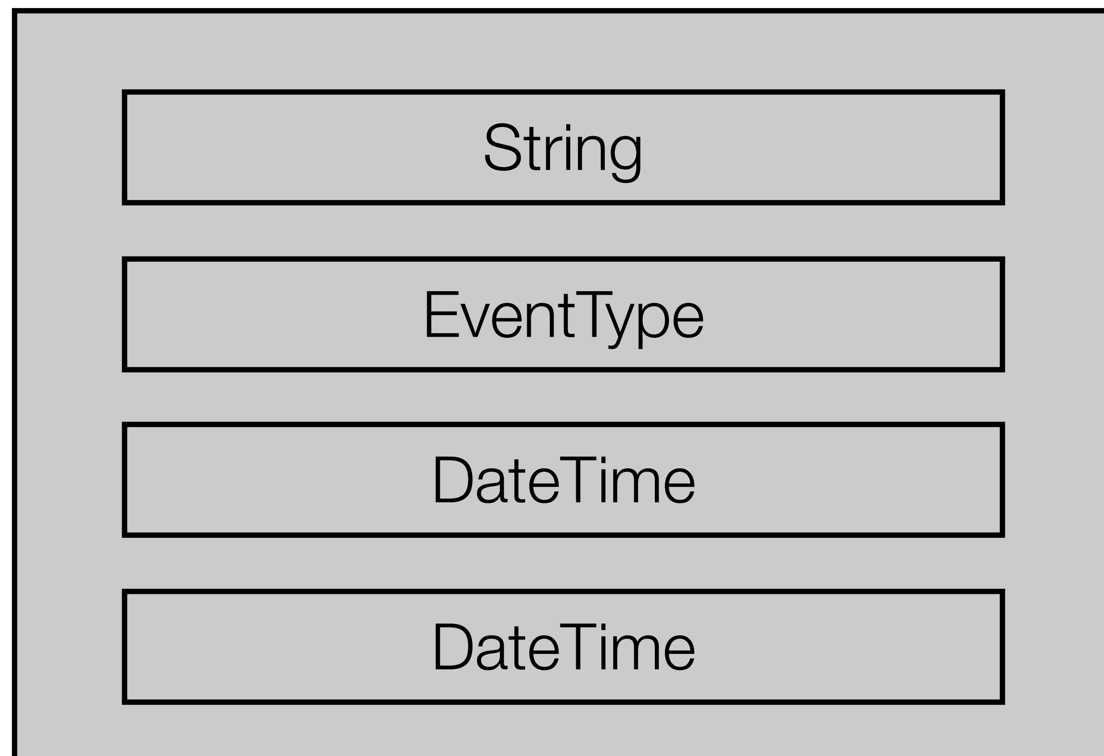
DELETE CANCEL SAVE

16:00 - 17:30 Game of Thrones

nd city



Composition

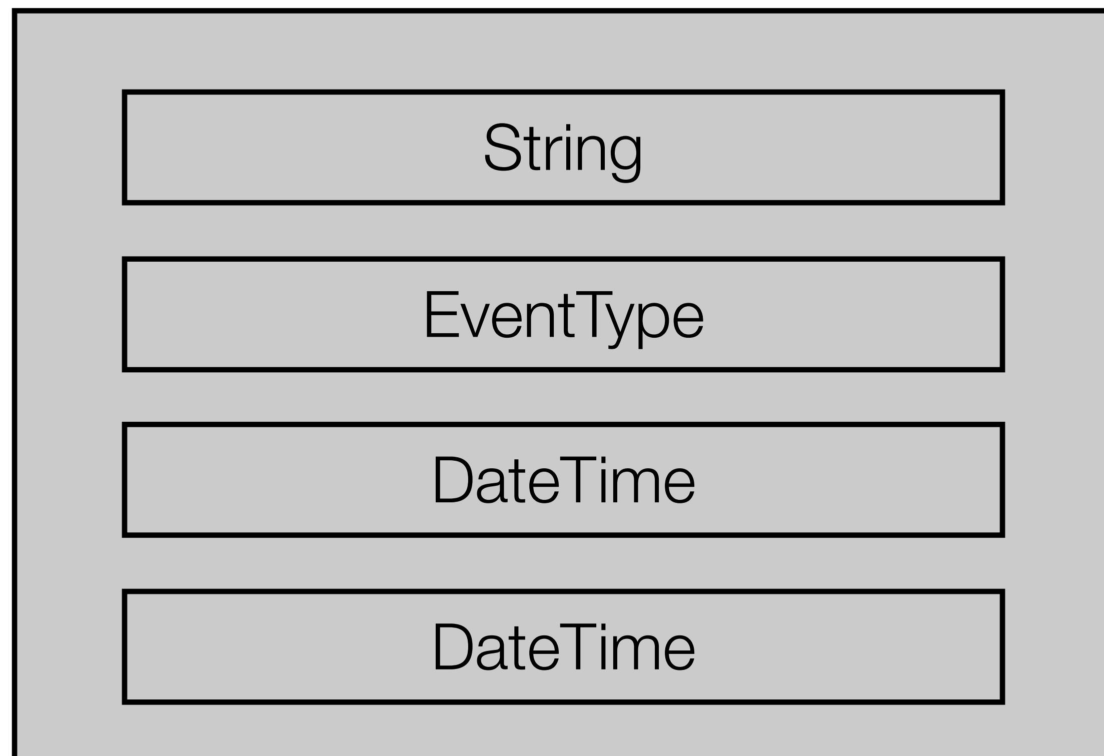


Avec TypeScript :

```
interface Props {
    description: String
    event: EventType
    start: DateTime
    end: DateTime
}
```

(Différent avec du jsx classique)

Composition



Avec TypeScript :

```
interface Props {
    description: String
    event: EventType
    start: DateTime
    end: DateTime
}
```

(Différent avec du jsx classique)

On type tout pour être sur que tout se passe bien

→ pas de undefined

React

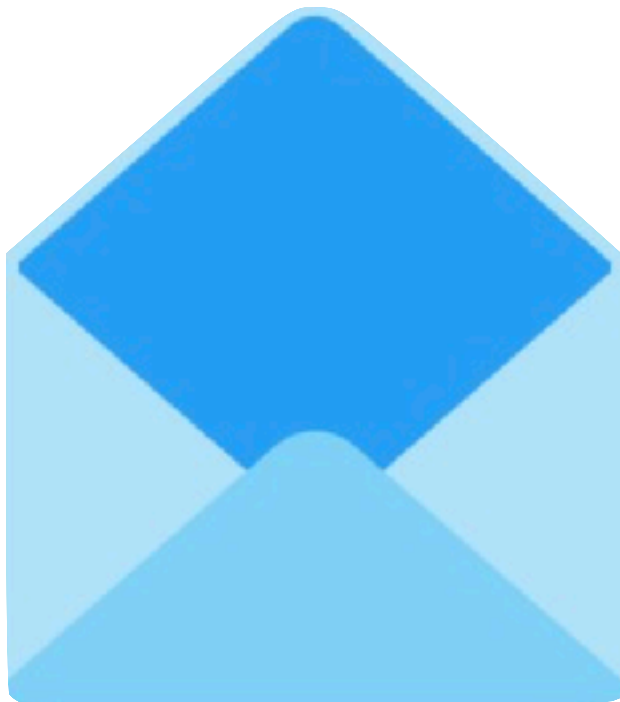
Gère la vue

Quelques principes

- Centré composant
- **Déclaratif**
- Composition plutôt qu'héritage
- Encapsulation et réactivité

Déclaratif

Example :



```
function render(count) {
  if (count > 0 && !hasBadge()) {
    addBadge();
  } else if (count === 0 && hasBadge()) {
    removeBadge();
  }
  if (count > 99 && !hasFire()) {
    addFire();
    setBadgeText('99+')
  } else if (count <= 99 && hasFire()) {
    removeFire();
  }
  if (count > 0 && !hasPaper()) {
    removePaper();
  } else if (count === 0 && hasPaper()) {
    addPaper();
  }
  if (count <= 99) {
    setBadgeText(count.toString())
  }
}
```

To define this UI imperatively, lets say we have an update function. We receive our new state, the updated count, and we have to update the UI. We might implement it something like this.

In this case we see that getting to the *next* state very much depends on what our *current* state is.

https://speakerdeck.com/lelandrichardson/cross-language-react?slide=17

```
function render(count) {
  return (
    <Envelope fire={count > 99} paper={count > 0}>
      <Badge visible={count > 0}>
        {count}
      </Badge>
    </Envelope>
  );
}
```

An identical UI, expressed declaratively, is somewhat simpler. Provided a count, we simple return the state of the UI that is desired for that count.

While in the imperative version we had to consider what the previous state of the UI was, in the declarative approach we no longer need to do so, as the underlying framework or runtime is figuring this out for you. In most cases this is all that is desired.

https://speakerdeck.com/lelandrichardson/cross-language-react?slide=17

Déclaratif

Un composant n'est pas une vue

La création de la vue est déléguée à React

On décrit le comportement,

- on évite de modifier l'état,
- on passe des props aux composants enfants

React

Gère la vue

Quelques principes

- Centré composant
- Déclaratif
- Composition plutôt qu'héritage
- Encapsulation et réactivité

Composition vs héritage

https://reactjs.org/docs/composition-vs-inheritance.html

1. On assemble des composants plutôt
2. On fait circuler l'information entre eux

Héritage classique de vues

```
class FancyBox extends View { /* ... */ }

class BlogPost extends View { /* ... */ }

class EditForm extends FormView { /* ... */ }

class FancyBlogPost extends ??? { /* ... */ }

class FancyEditForm extends ??? { /* ... */ }
```

Approche par composition

```
function FancyBox({ children }) {
  return <View style={fancy}>{children}</View>
}

function BlogPost(props) { /* ... */ }

function EditForm(props) { /* ... */ }

function FancyBlogPost(props) {
  return <FancyBox>
    <BlogPost {...props} />
  </FancyBox>
}

function FancyEditForm(props) {
  return <FancyBox>
    <EditForm {...props} />
  </FancyBox>
}
```

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>);
    }
}
```

React

Gère la vue

Quelques principes

- Centré composant
- Déclaratif
- Composition plutôt qu'héritage
- Encapsulation et réactivité

Encapsulation et réactivité

- Un composant reçoit des propriétés (props)
  = données qui ne changent pas (immutable)

- Un composant gère sont état (state)
  = données qui changent (mutable) *localement*

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

Encapsulation et réactivité

Au niveau d'une application

- L'état descend via des props
- L'état remonte par des callbacks

- Avec Redux on évite de faire remonter l'état, on diffuse une action de changement.

Isolement / encapsulation

React

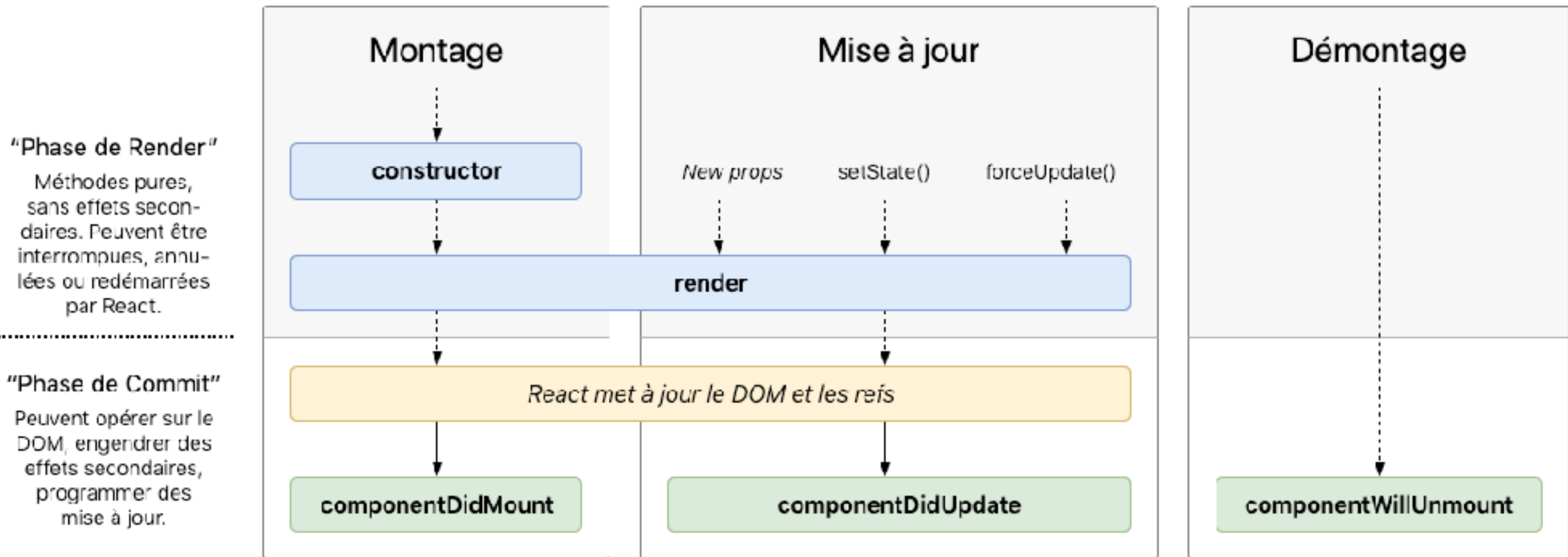
Gère la vue

Quelques principes

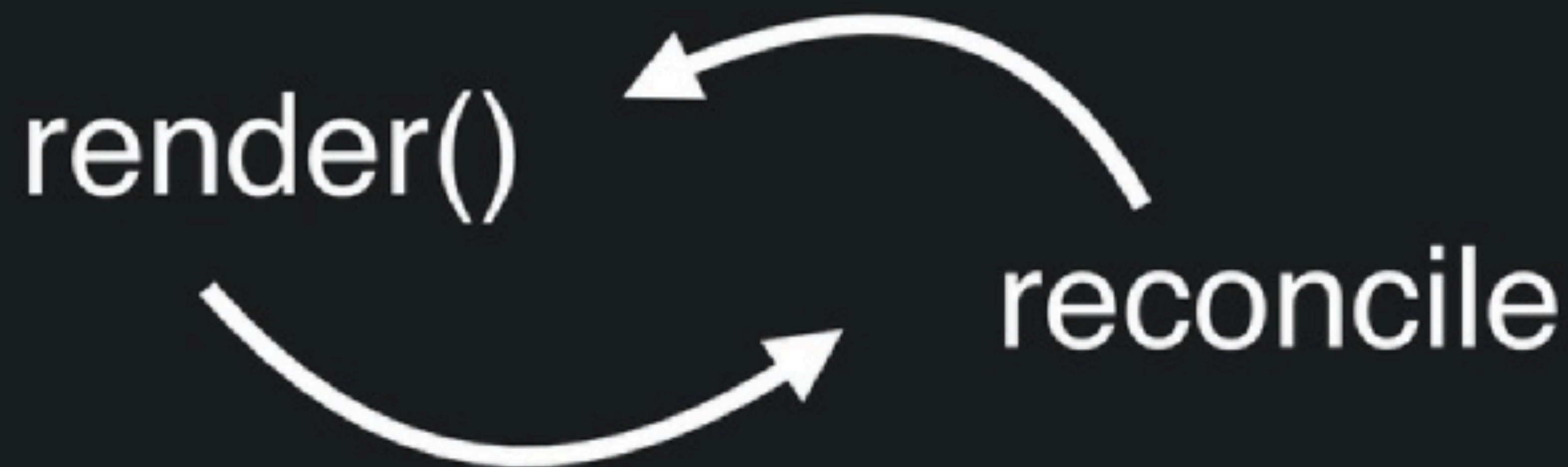
- Centré composant
- Déclaratif
- Composition plutôt qu'héritage
- Encapsulation et réactivité

Comment décider quoi re-dessiner ?

Cycle de vie des composants



http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/



So when we update our UI, first the render function of our component is called. This returns a lightweight data structure that essentially encodes which components and views and their attributes should be.

We take that result and compare it to the result that was returned the previous time. This is called reconciliation. If we encounter more components here, we call render on that component and start the process all over again until there are no more components to reconcile.

render()

reconcile

Reconciliation also adds operations to a queue which are needed to update the views into their new state.

operations

- create #1
- create #2
- create #3
- insert #2->#1
- insert #3->#1
- move #3, 0
- setAttr #1, ‘enabled’, true

commit

Whenever the full tree or subtree is reconciled, we are able to “commit” the queue of operations which actually updates the pixels on the screen. There are various ways we can think of working through these three phases.

Hooks

Question :

- *comment conserver l’état d’un composant entre deux rendus ?*

State Hooks

https://reactjs.org/docs/hooks-overview.html

On signifie à React ce qui constitue l'état de notre composant pour lui déléguer sa gestion entre deux renders.

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Effect Hooks

Gérer les effets de bords proprement via une API unifiée.
Ici le composant modifie “le monde extérieur” (le titre de la page au rafraichissement du DOM)

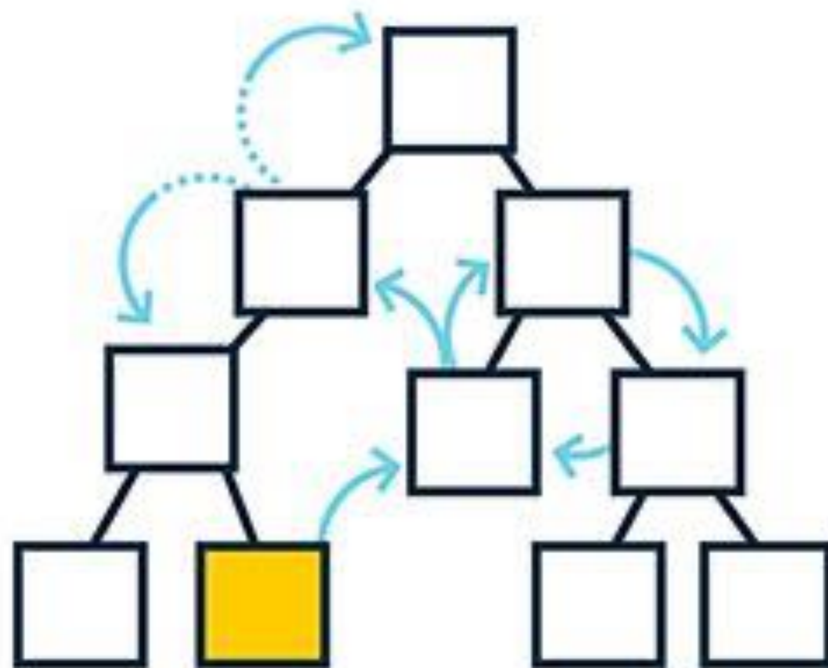
```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

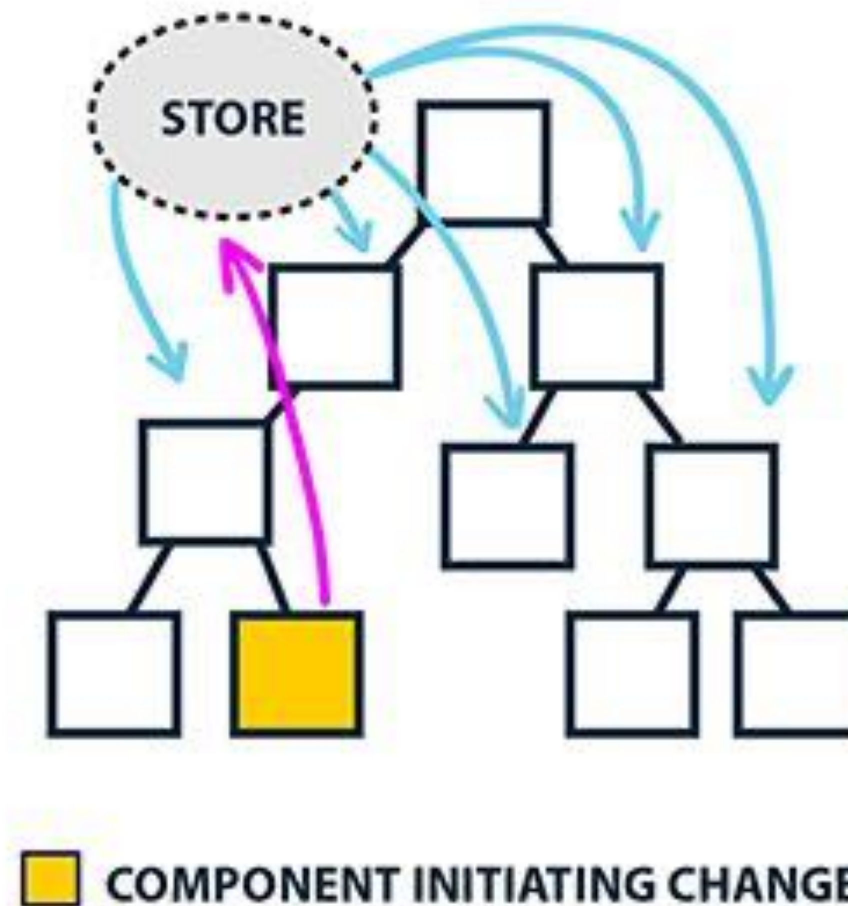
Plan

- Introduction
- Quelles limites de MVC
- Quelques principes généraux
- En pratique avec React
- **Redux**
- Traitement réactif de flux

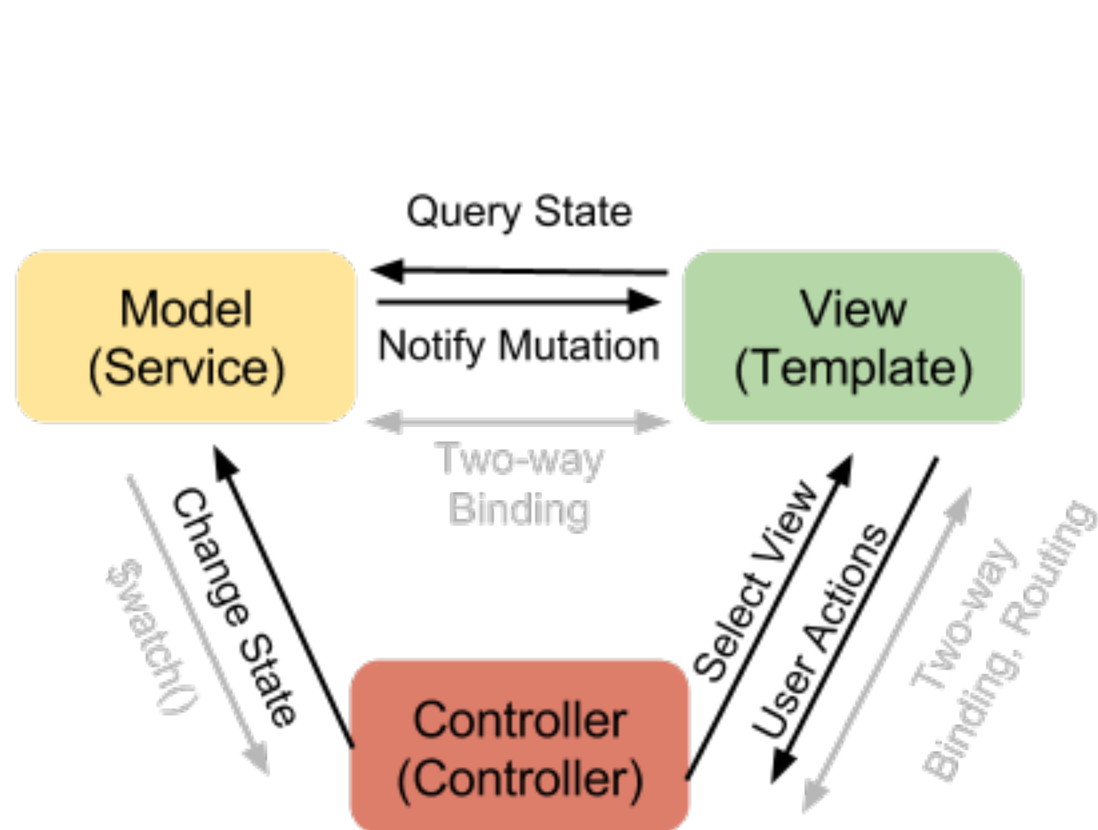
Pourquoi Redux

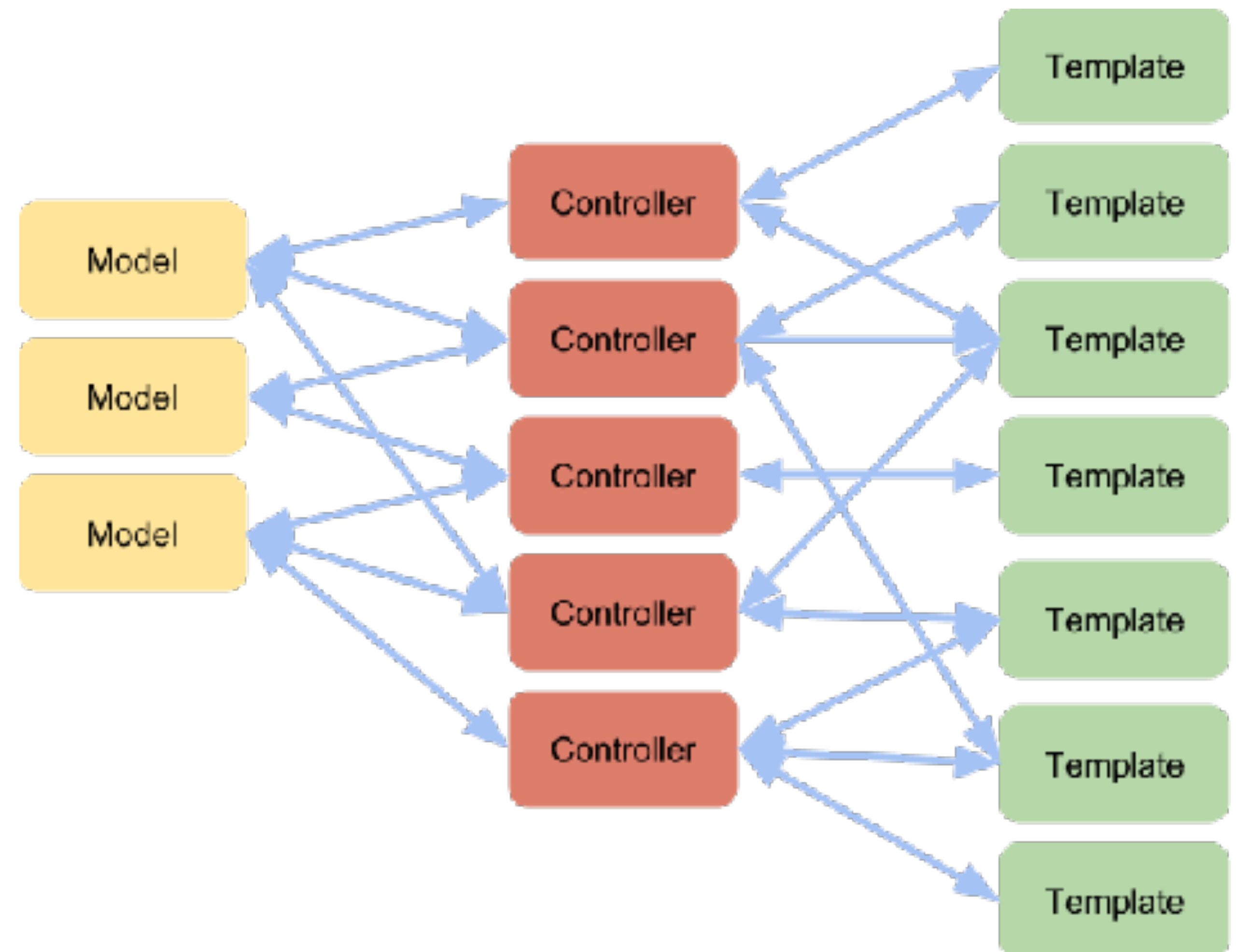
https://www.foreach.be/blog/why-the-react-redux-combo-works-like-magic





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



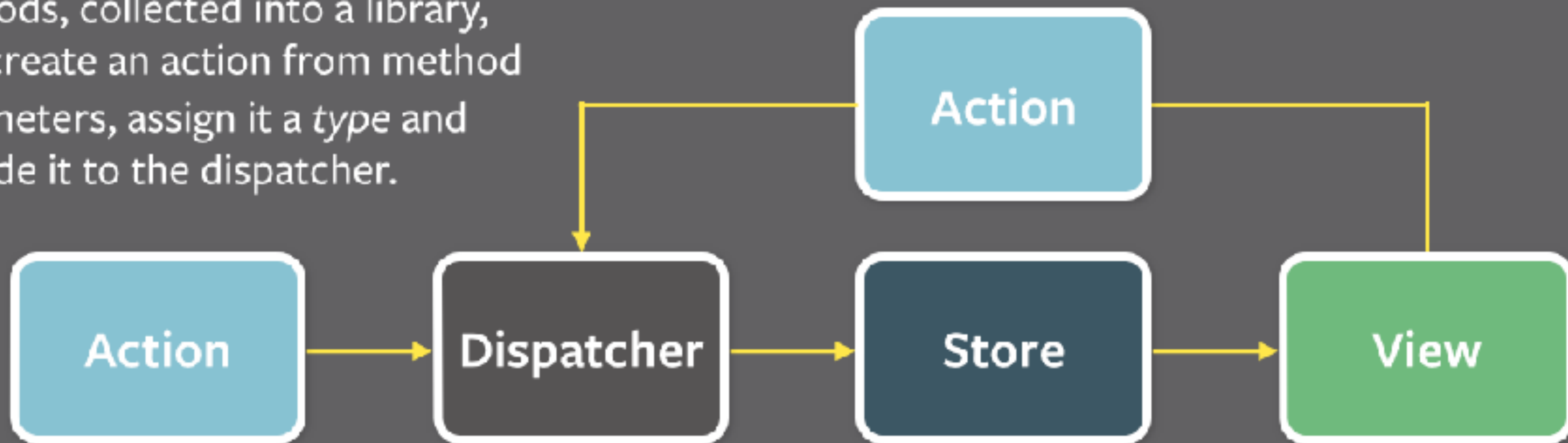


MVC

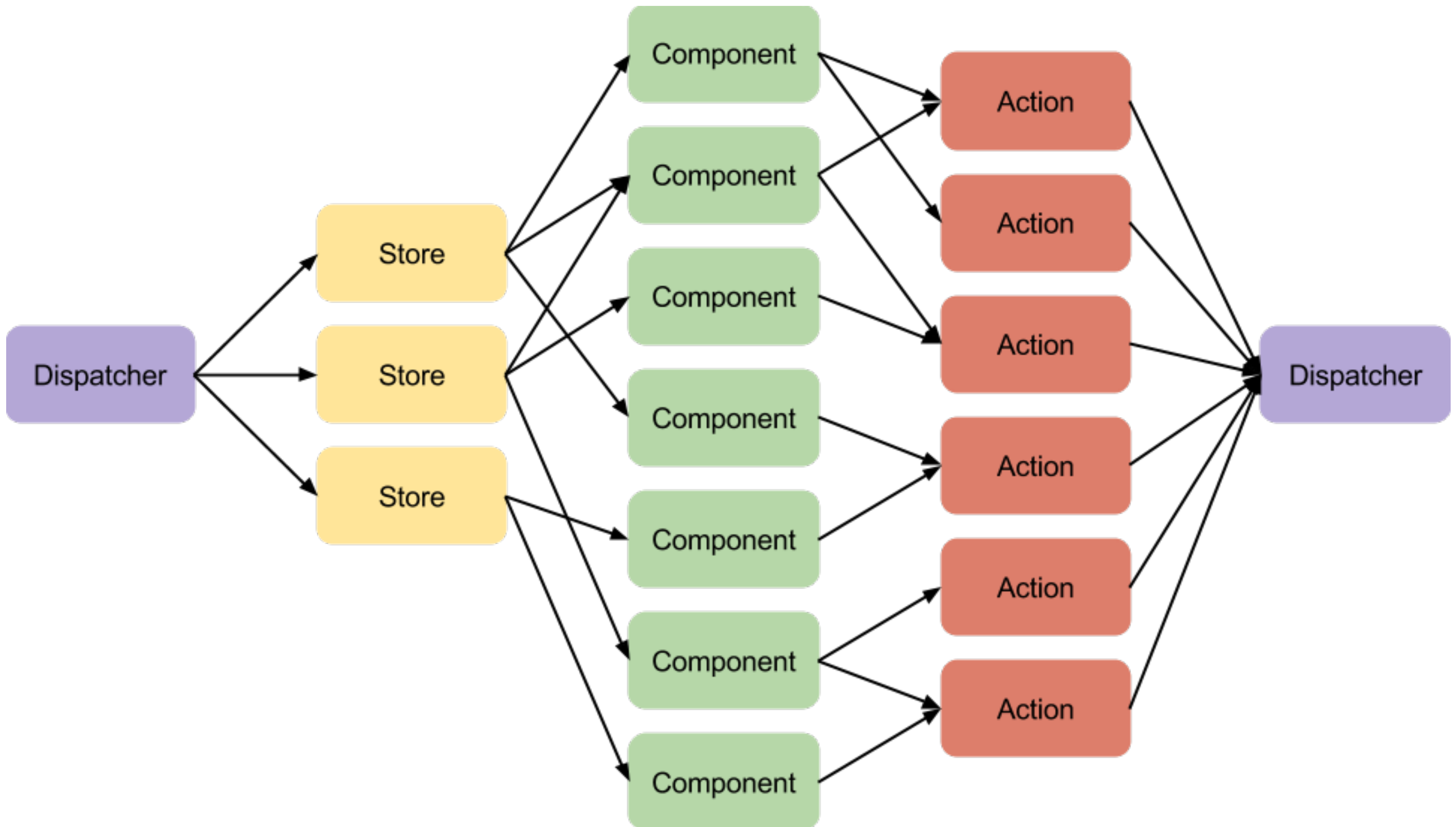
MVVM

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



En pratique sur une application



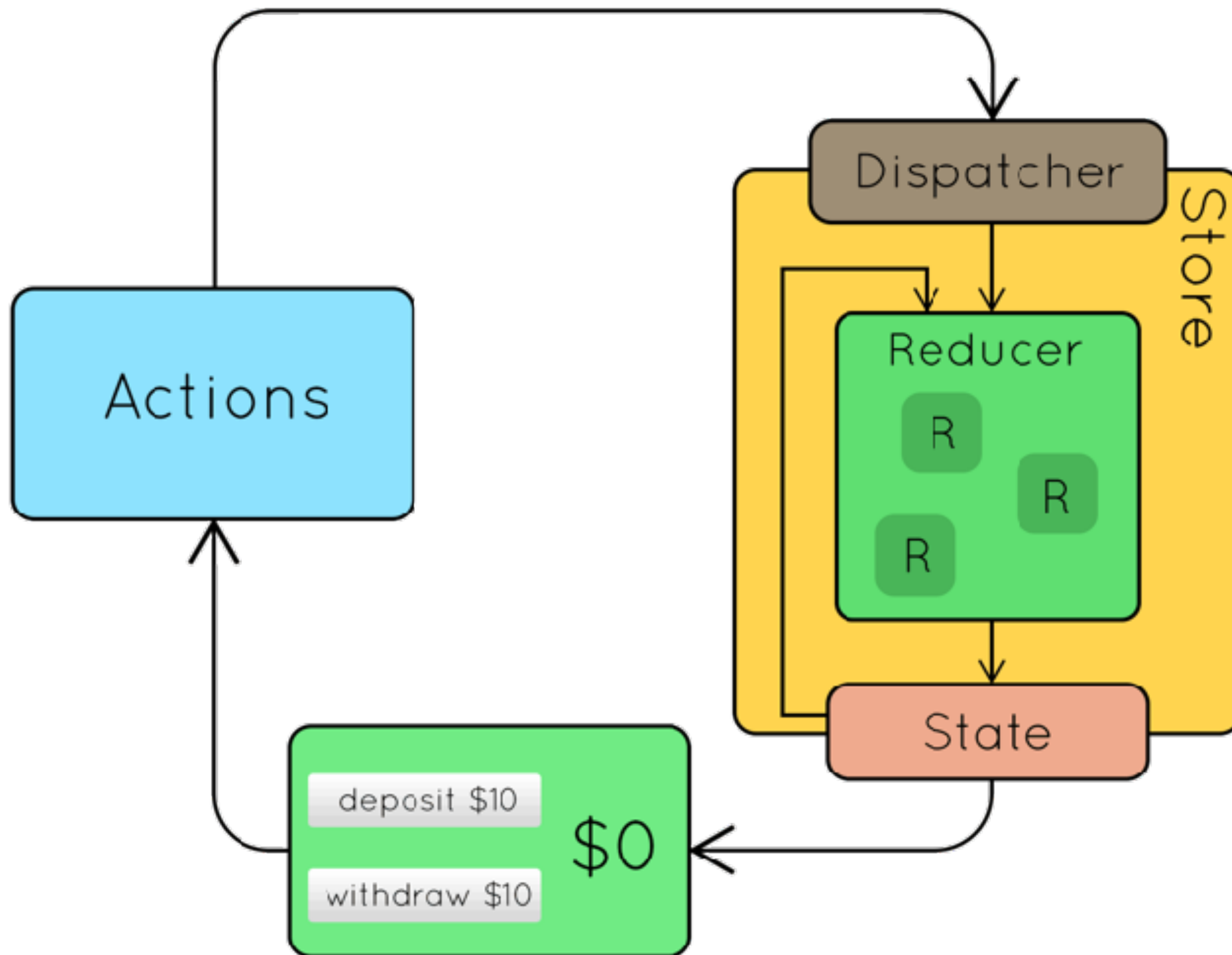
Redux : une implémentation de l'archi Flux

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

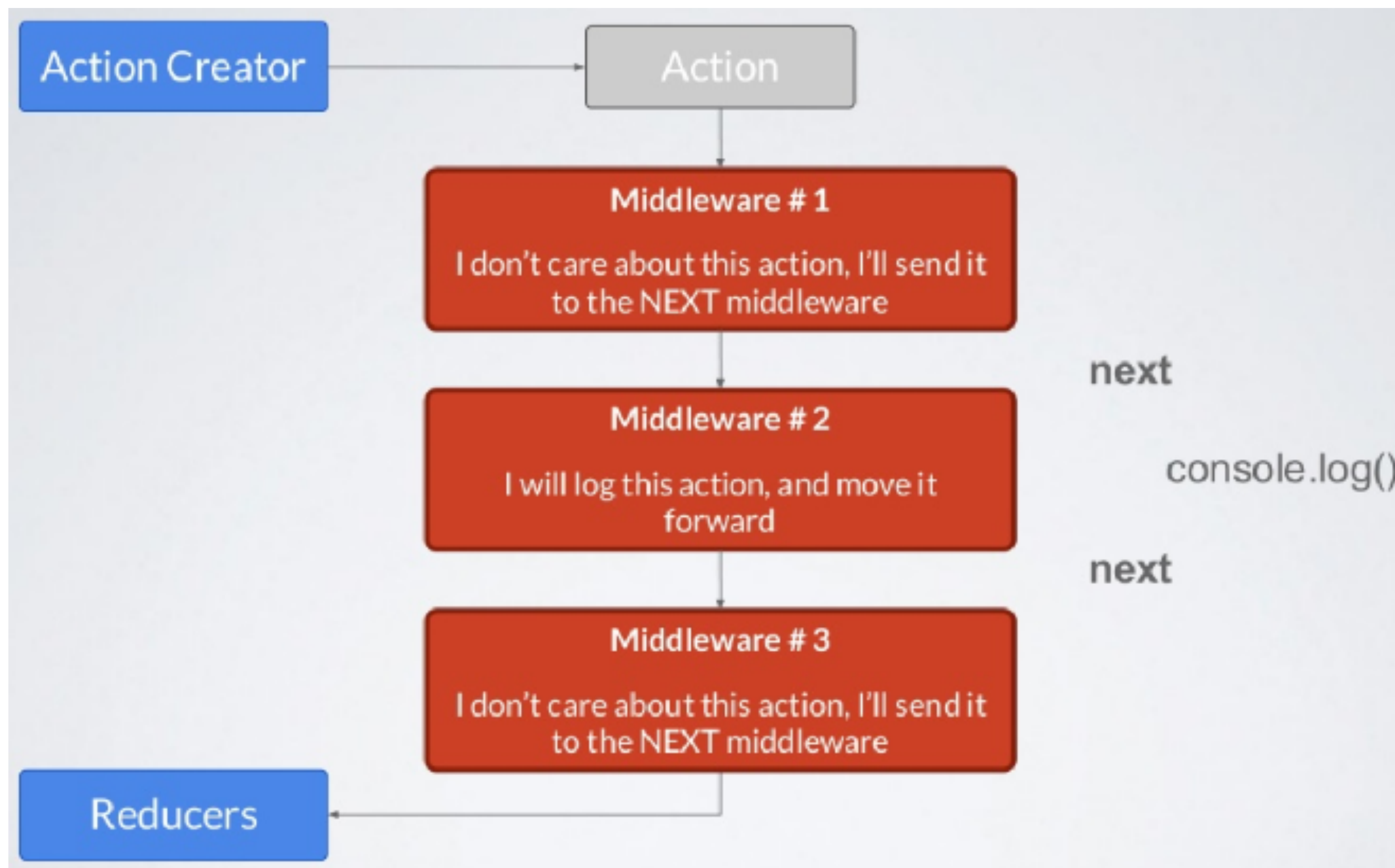
```
import {createStore} from "redux";

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

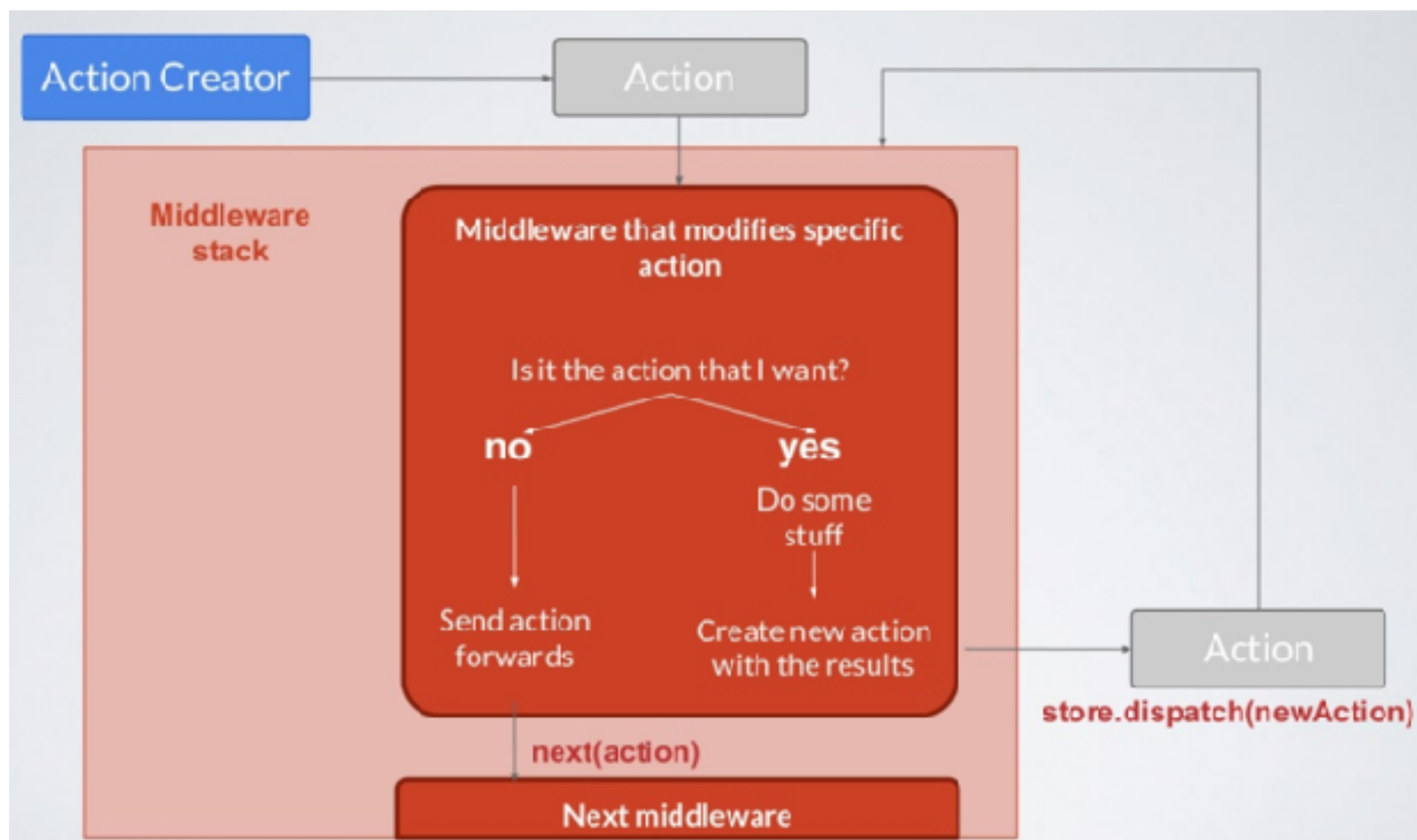
Redux Middleware



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 utiliser Redux ?

Les composants React gèrent déjà leur état interne.

Redux :

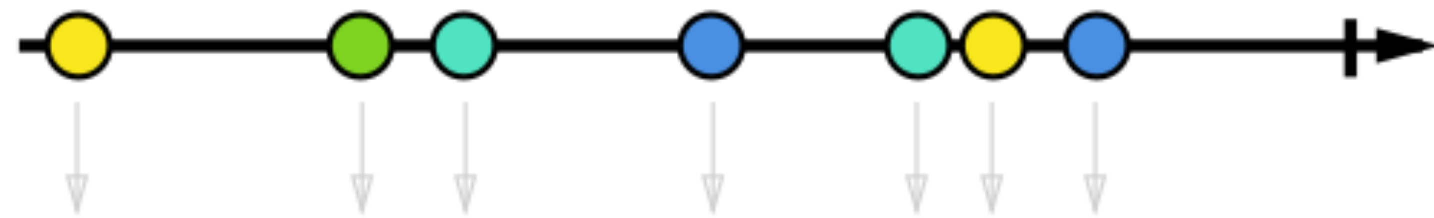
1. Gestion centralisée de l'état global de l'application
2. Si plusieurs composants partagent les mêmes données, les stocker dans redux permet une gestion coordonnée
3. Time-travel debugging (on peut revenir à des états passés)
4. Hot reloading pour le dev
   sans Redux: modif de composant -> état perdu

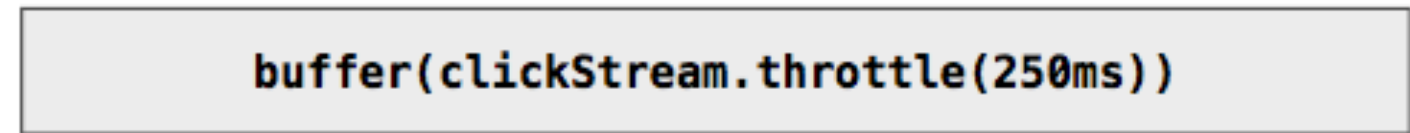
Plan

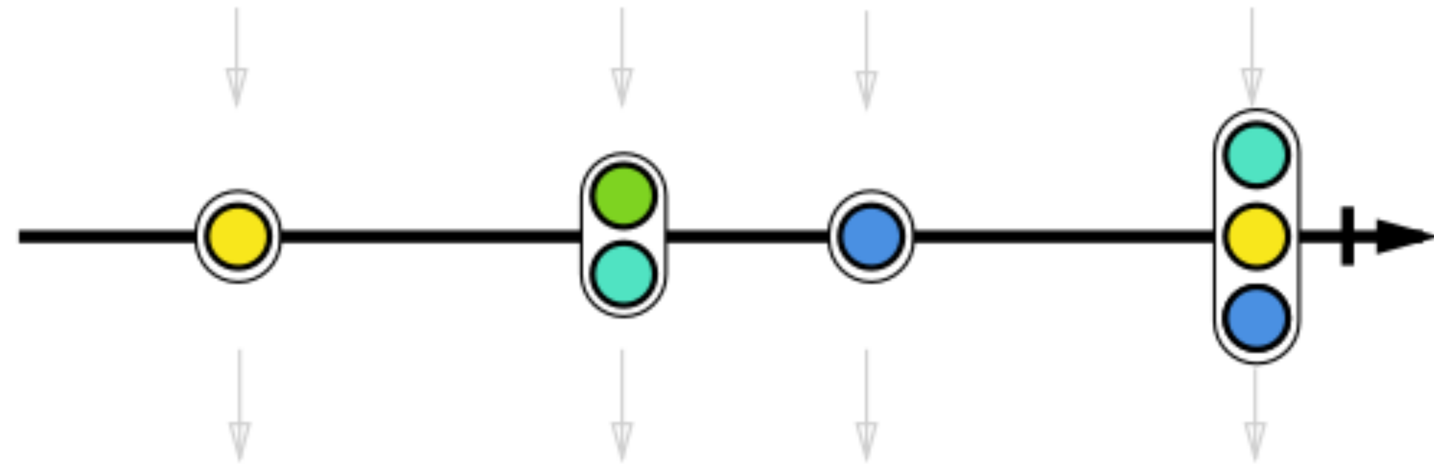
- Introduction
- Quelles limites de MVC
- Quelques principes généraux
- En pratique avec React
- Redux
- **Traitement réactif de flux**

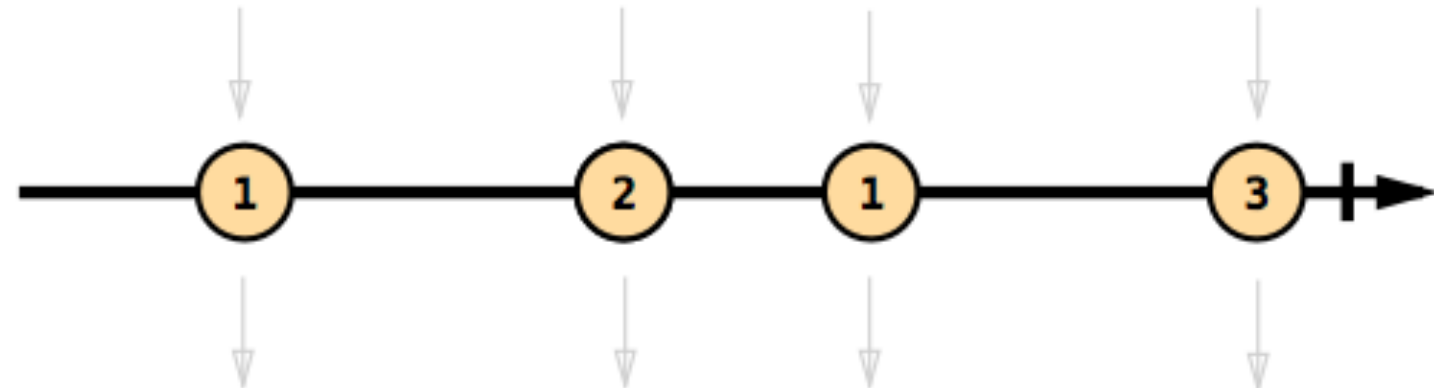
Un exemple de transformation

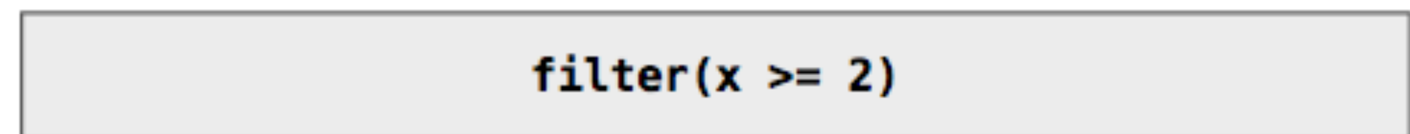
https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

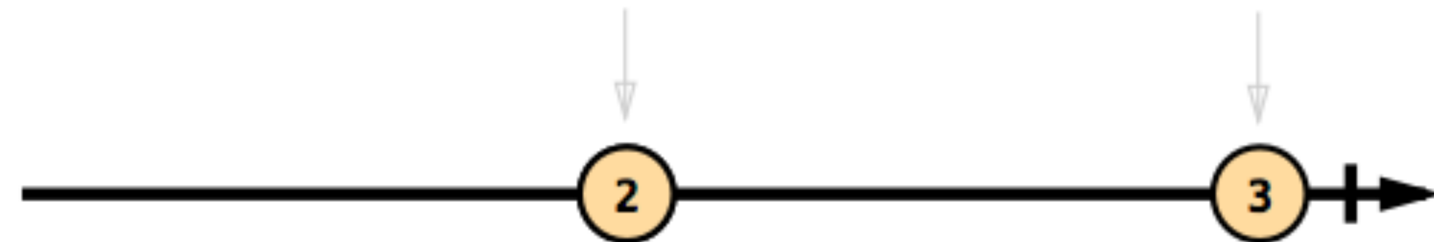




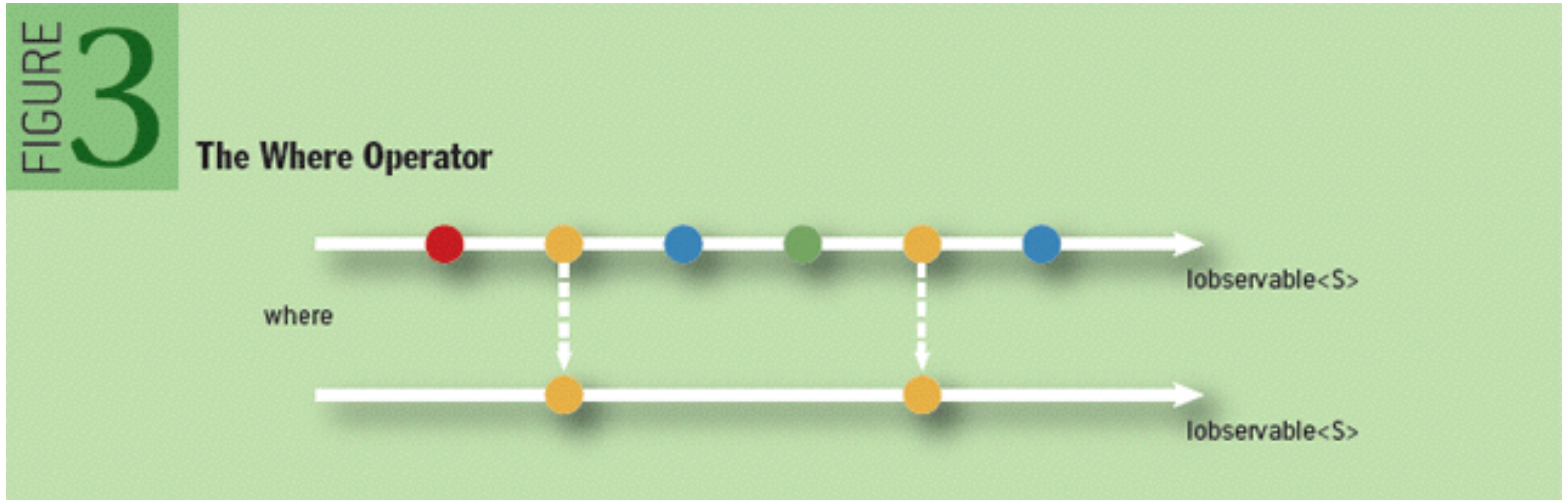




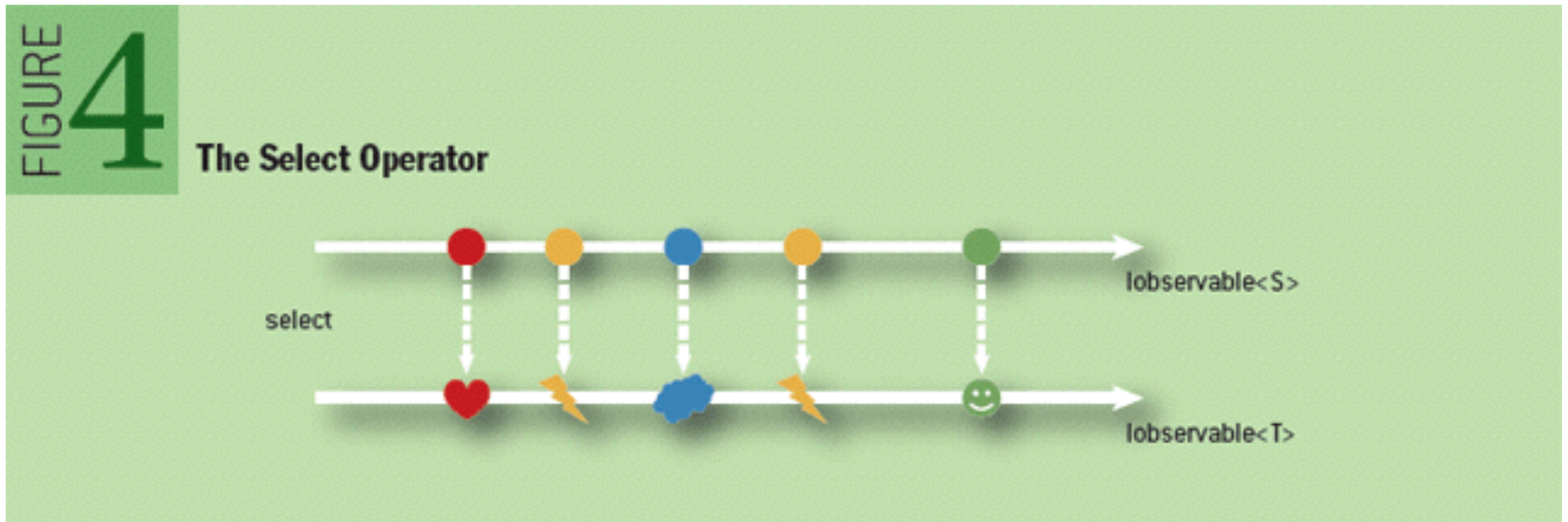




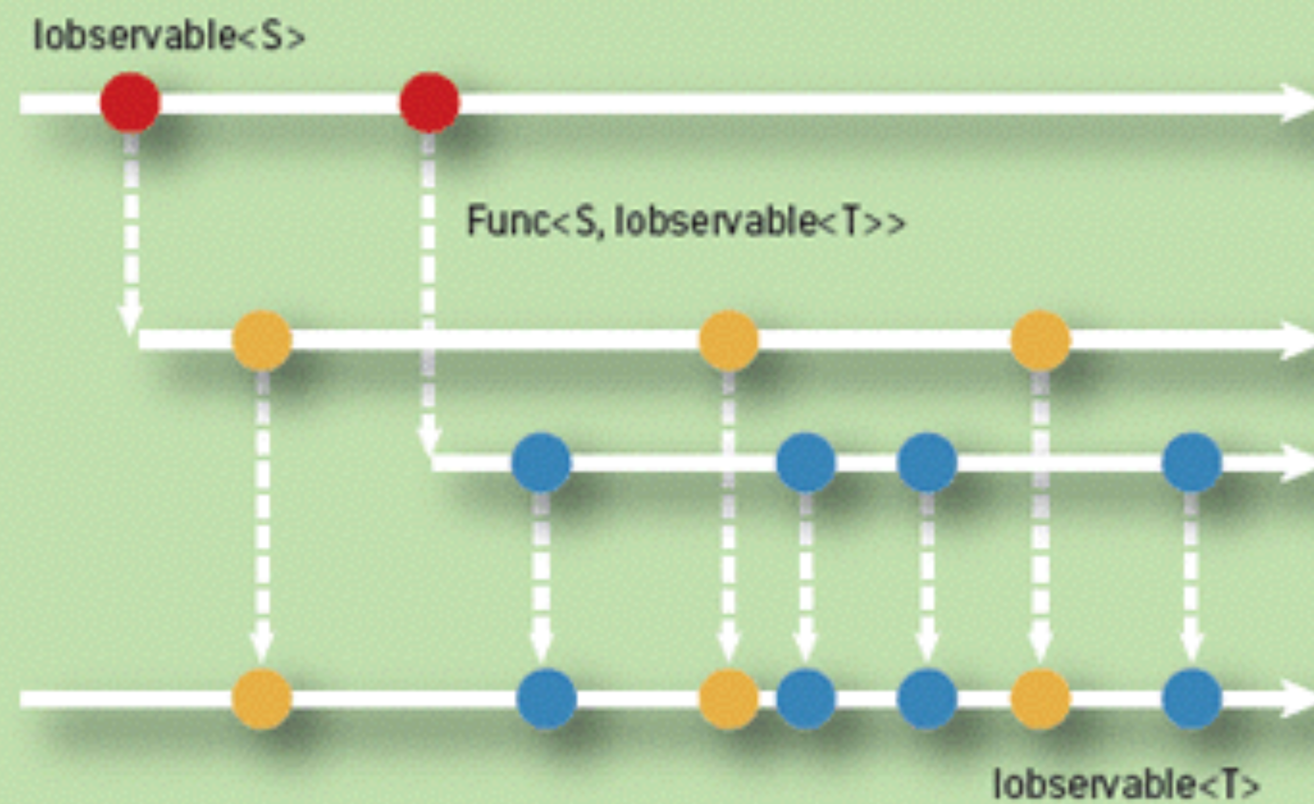
Where



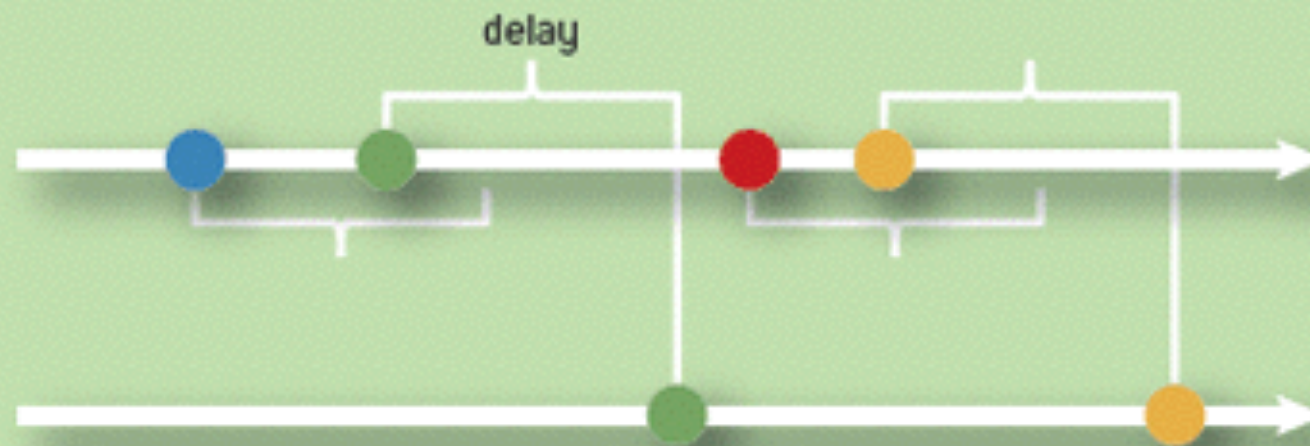
Select



SelectMany : plusieurs flux



Throttle



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/