Programmation Réactive

Principes fondamentaux et application au Web

Plan

- **Quoi et pourquoi la réactivé**
- Quelles limites de MVC
- Architecture Flux
- Réactivité, Vue et Vue
- Traitement réactif de flux

Qu’est ce que la programmation réactive ?

Une approche visant à mieux gérer les flux

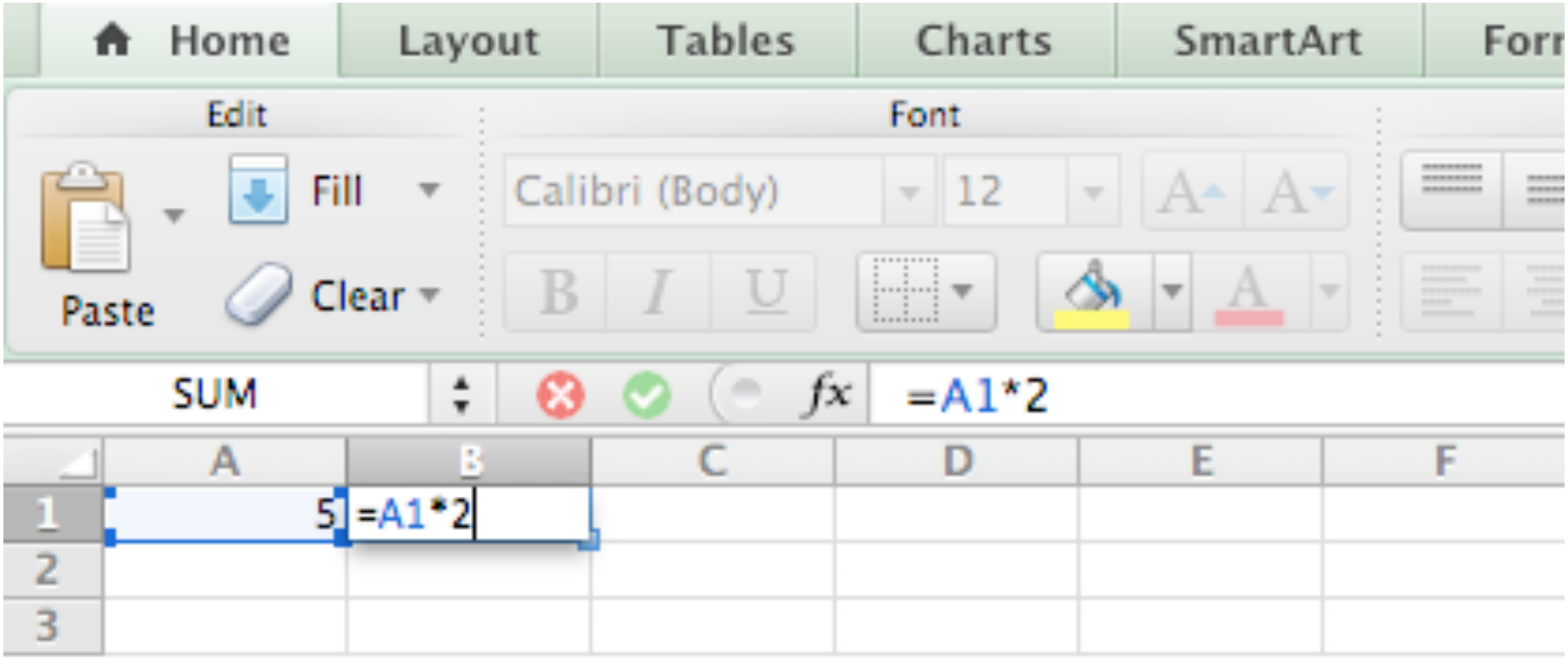
Deux types de flux

- Des événements discrets : frappe clavier, action utilisateur
- Des événements continus ou *comportements* : position souris

Dépasser les callbacks ou le patron Observer.

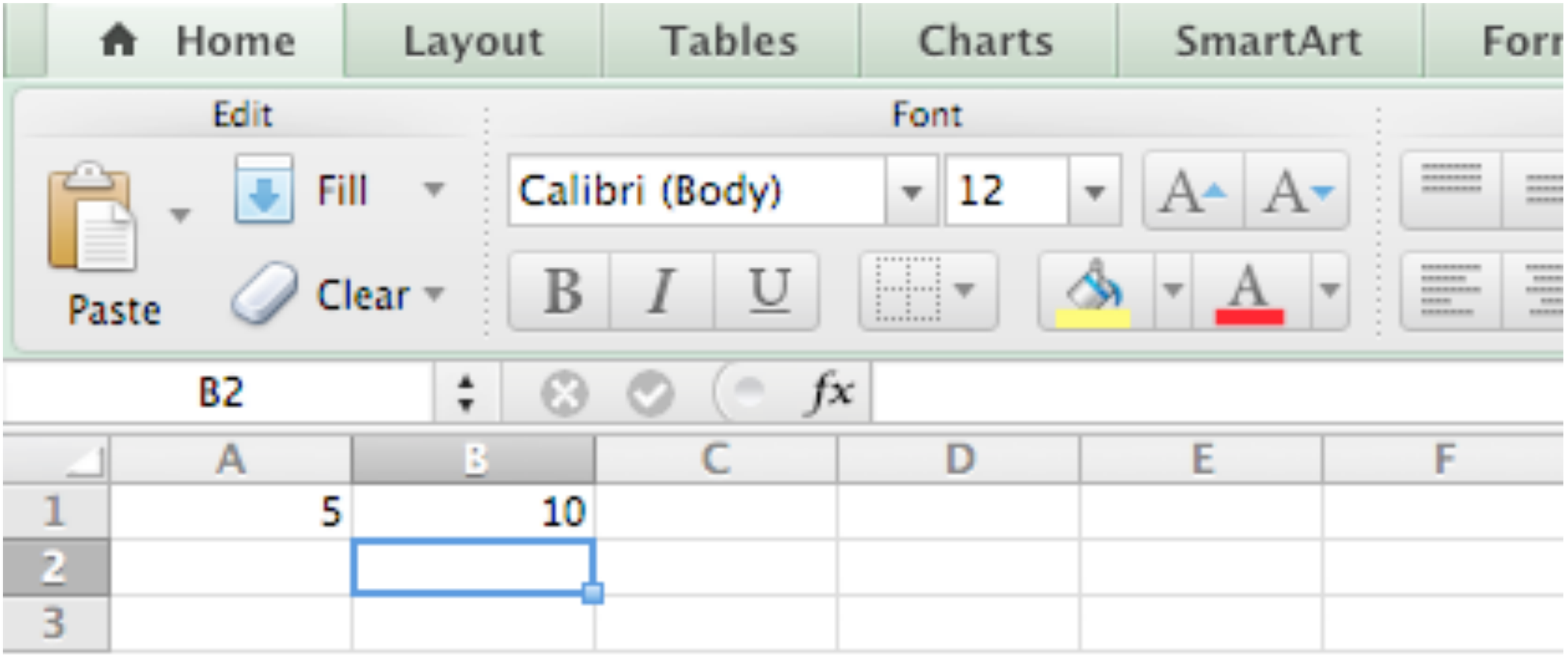
Alternative aux machines à état

Ou avez vous vu ça ?



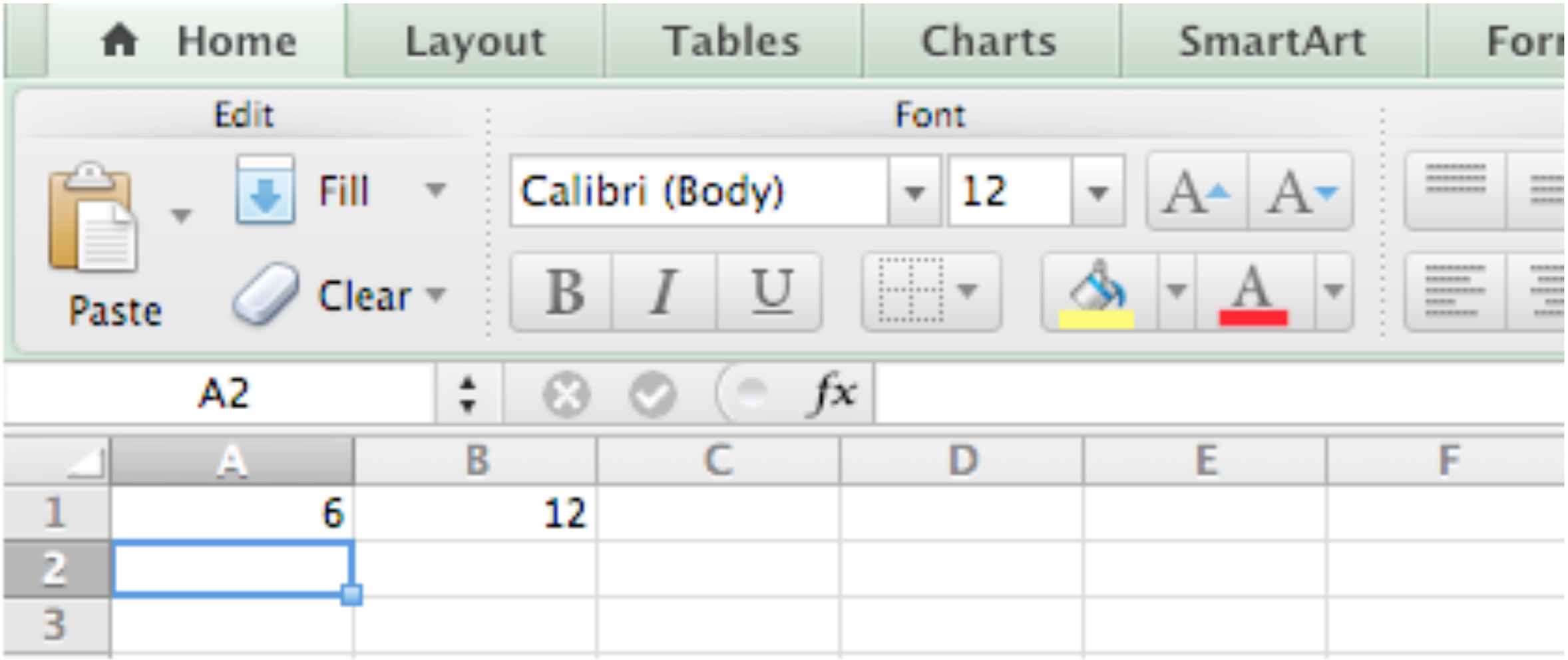
http://www.hanselsolutions.com/blog/surf-talk/shiny-surf.html#/9

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

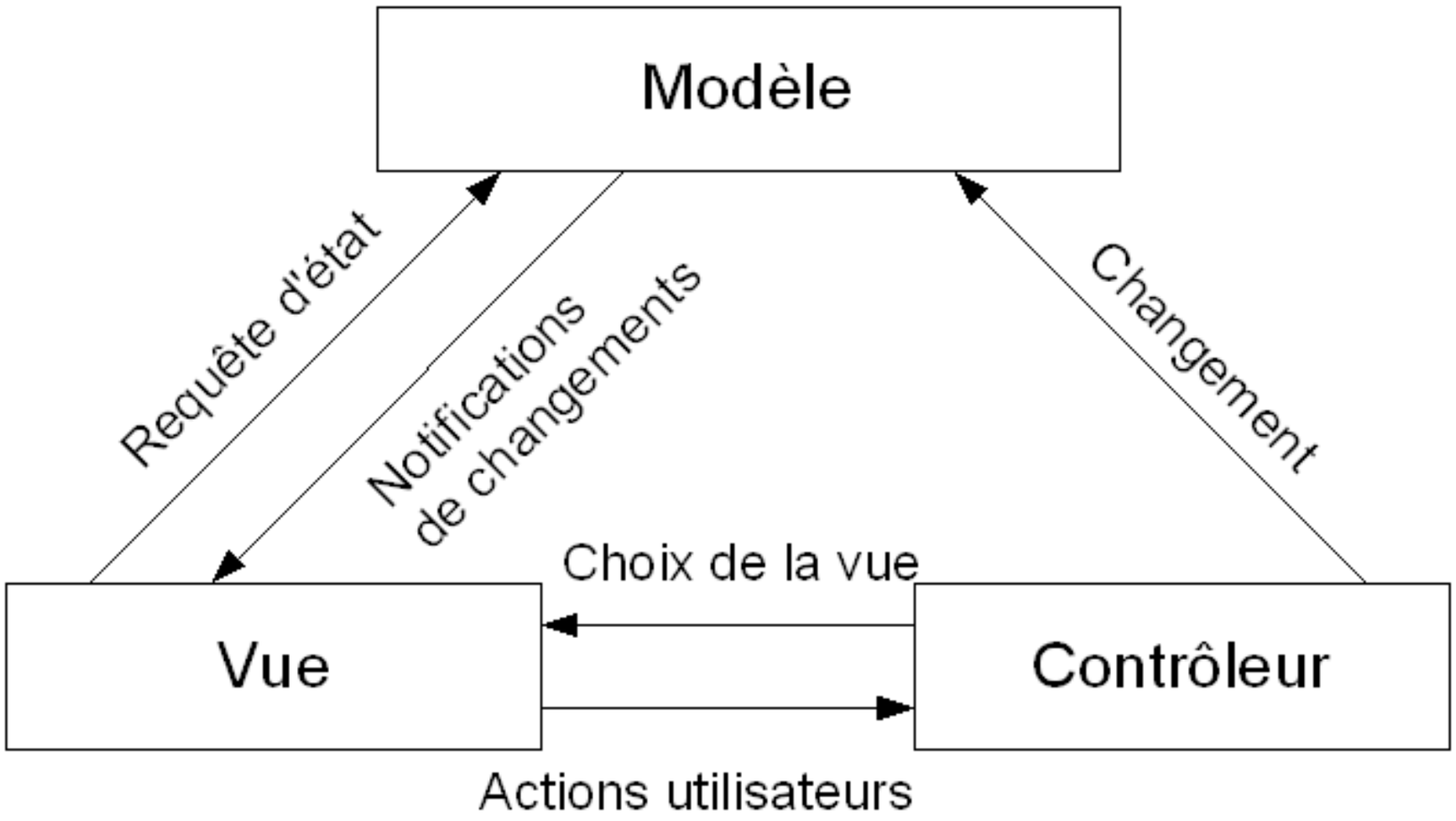
Exemples

À vous

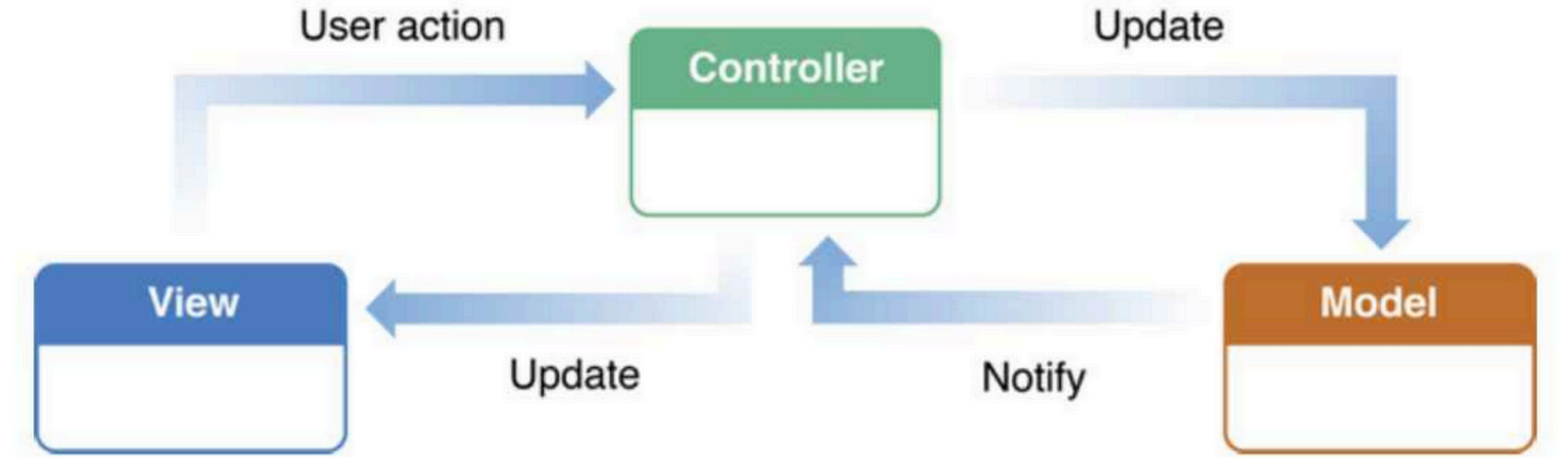
Plan

- Quoi et pourquoi la réactivé
- **Quelles limites de MVC**
- Architecture Flux
- Réactivité, Vue et Vuex
- Traitement réactif de flux

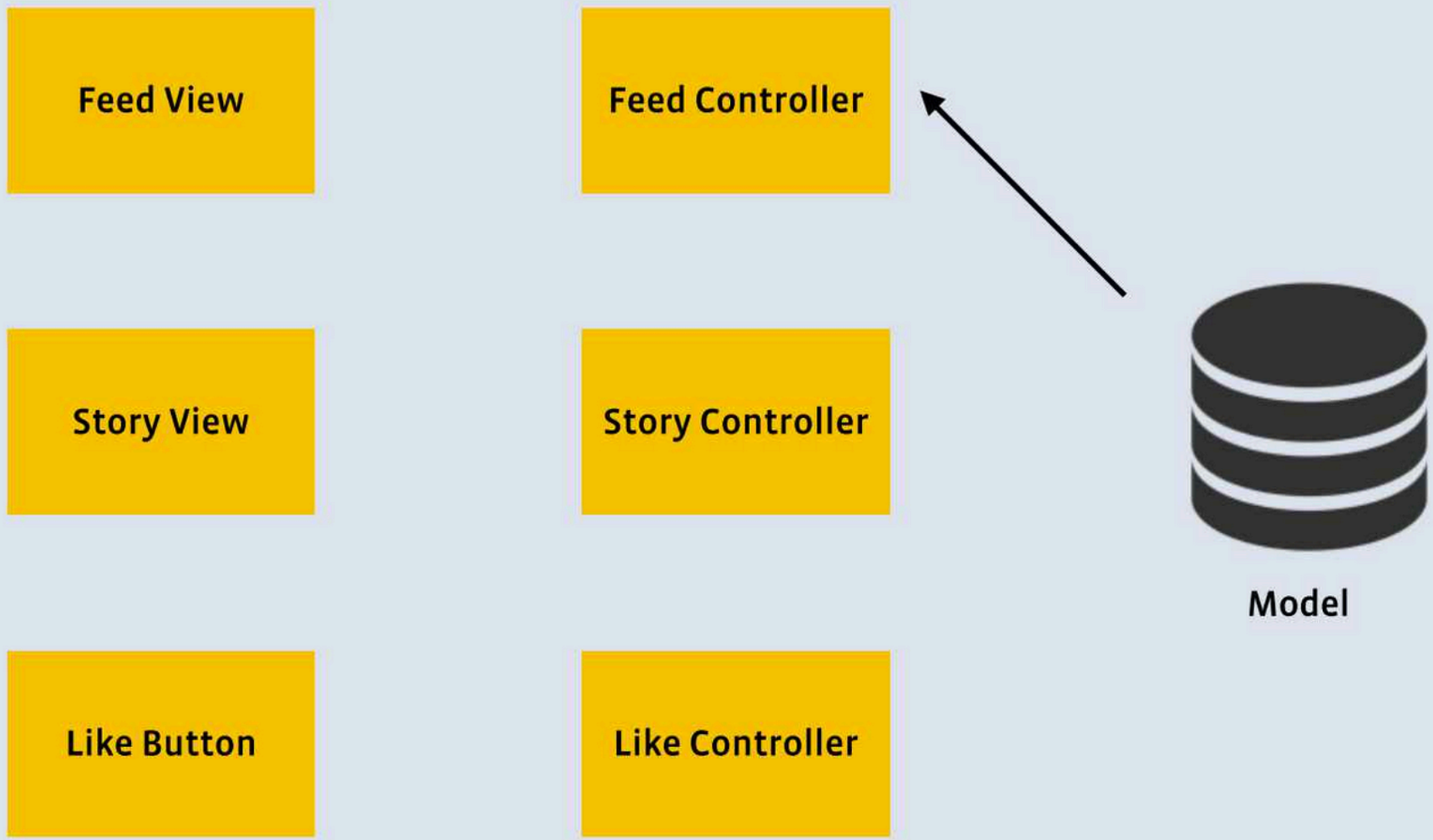
MVC - plutôt natif



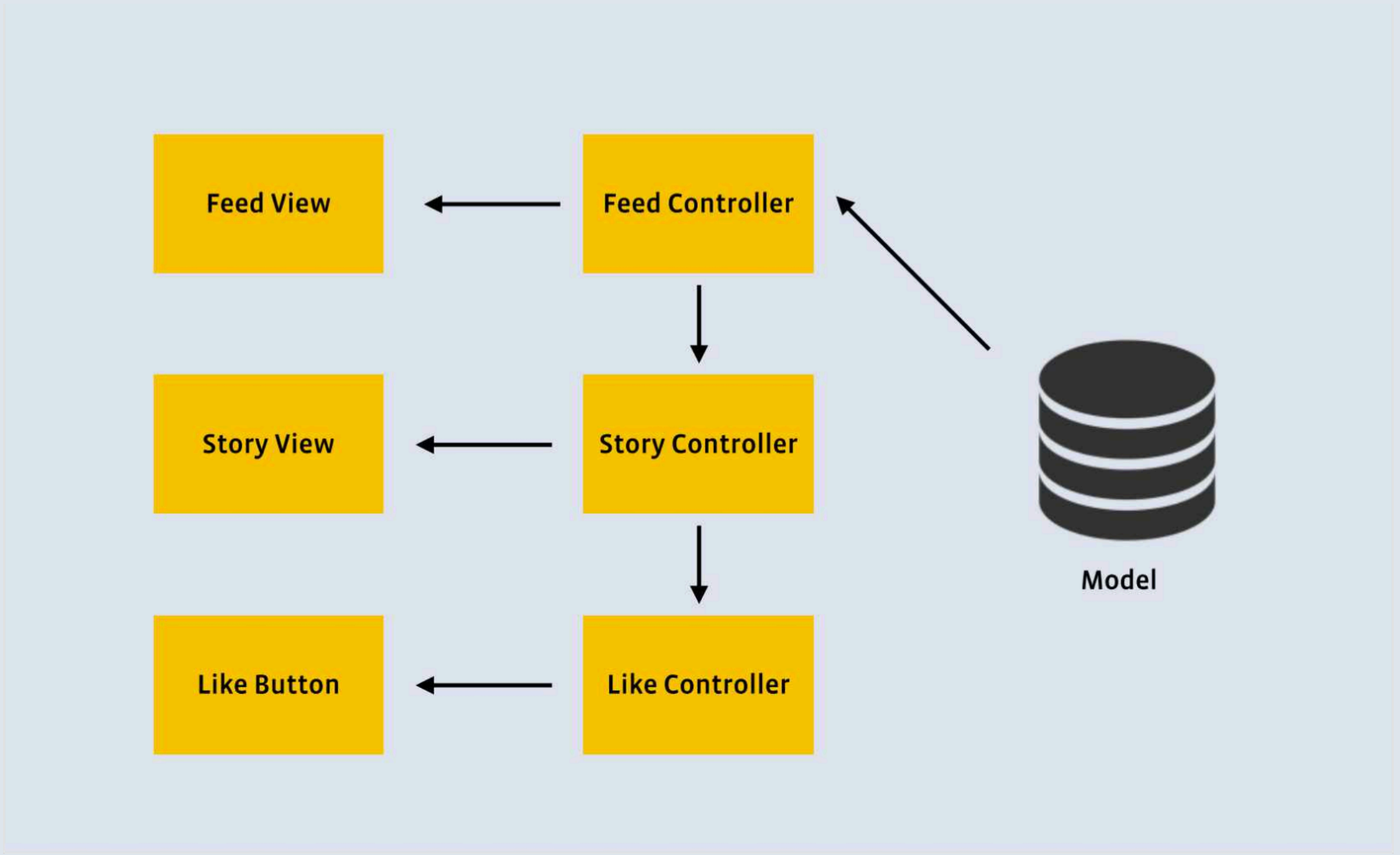
MVC - plutôt Web



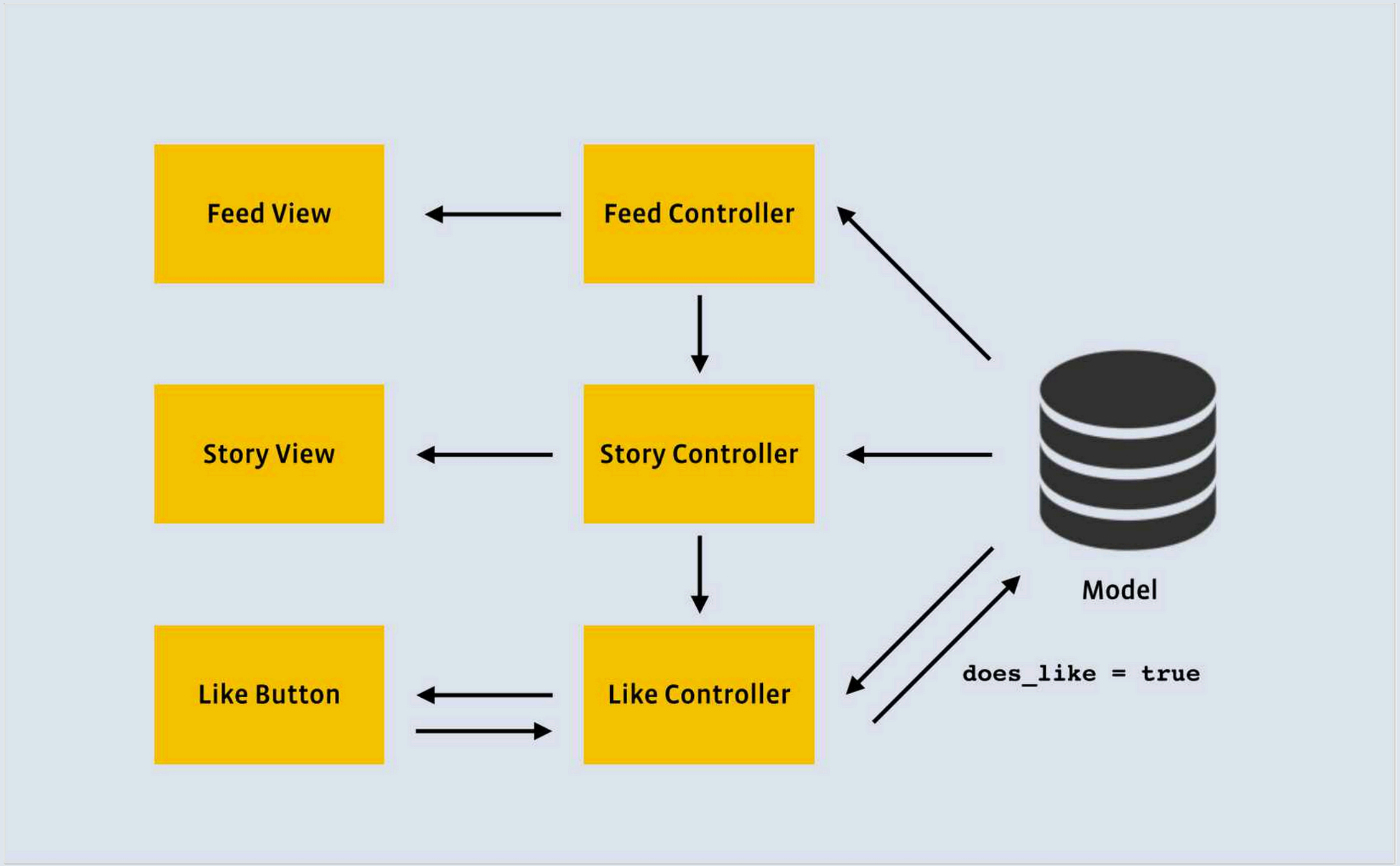
En pratique


Feed View
Feed Controller
Story View
Story Controller
Model
Like Button
Like Controller

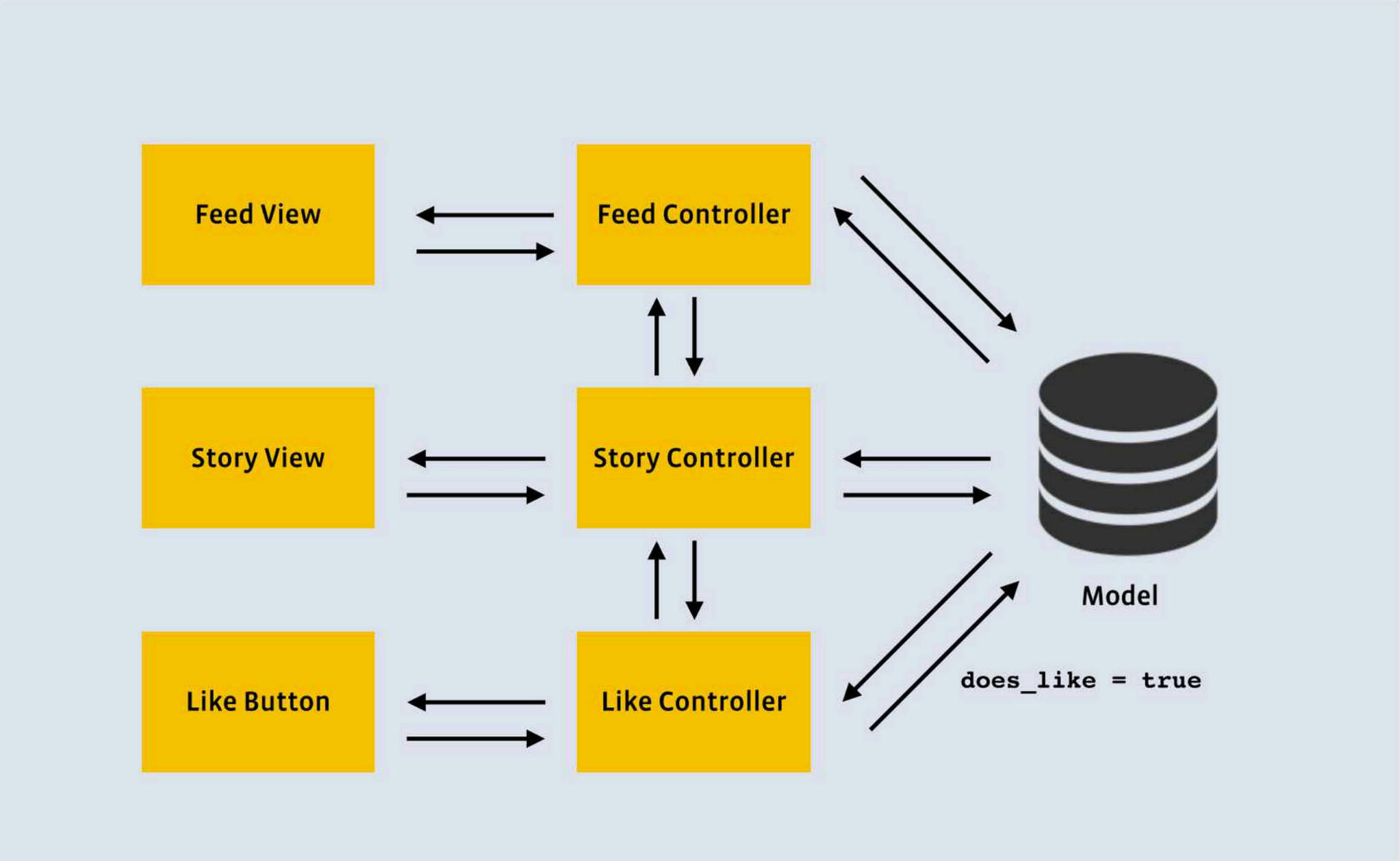
En pratique


Feed View
Feed Controller
Story View
Story Controller
Like Button
Like Controller
Model

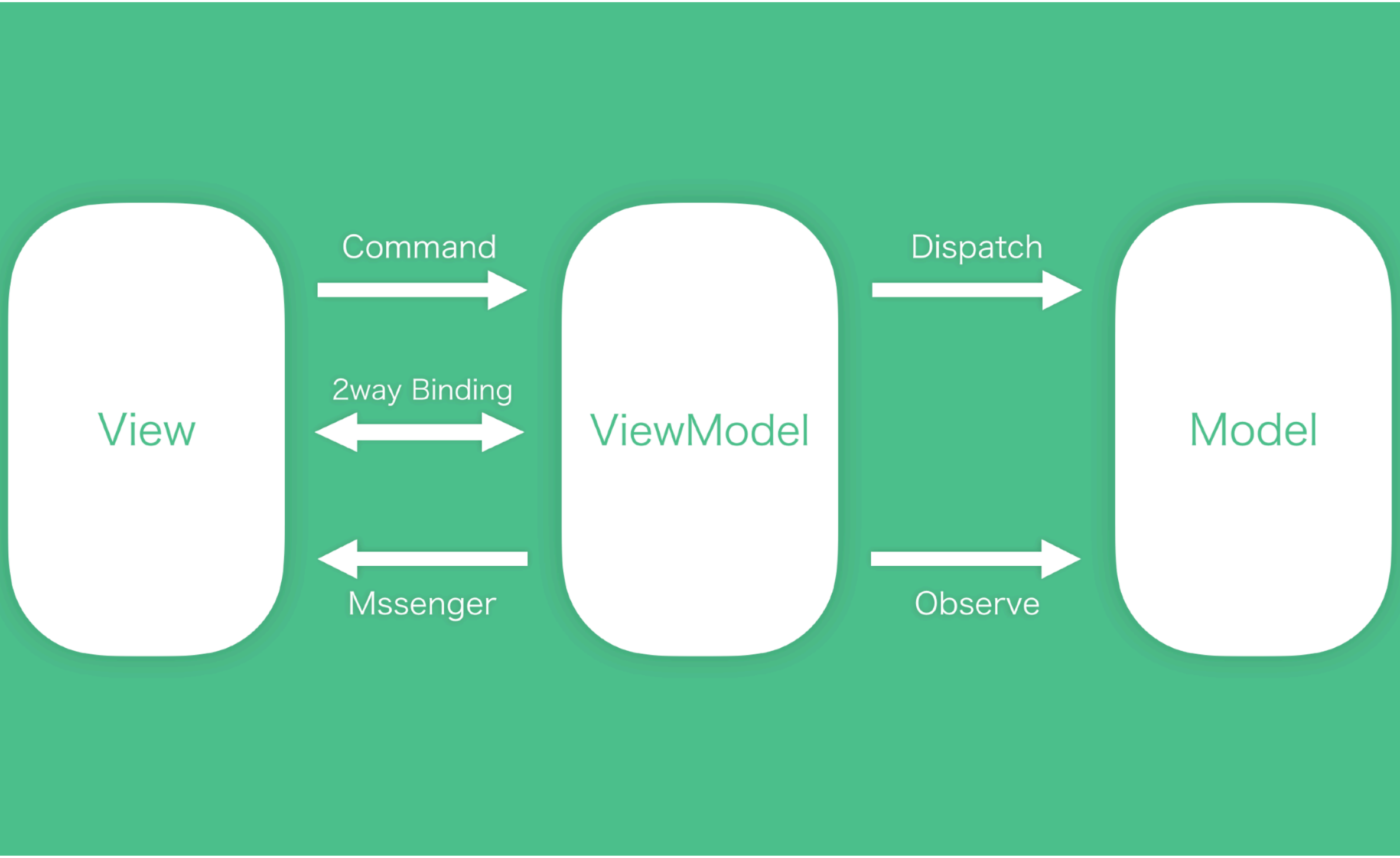
En pratique


Feed View
Feed Controller
Story View
Story Controller
Like Button
Like Controller
Model
does_like = true

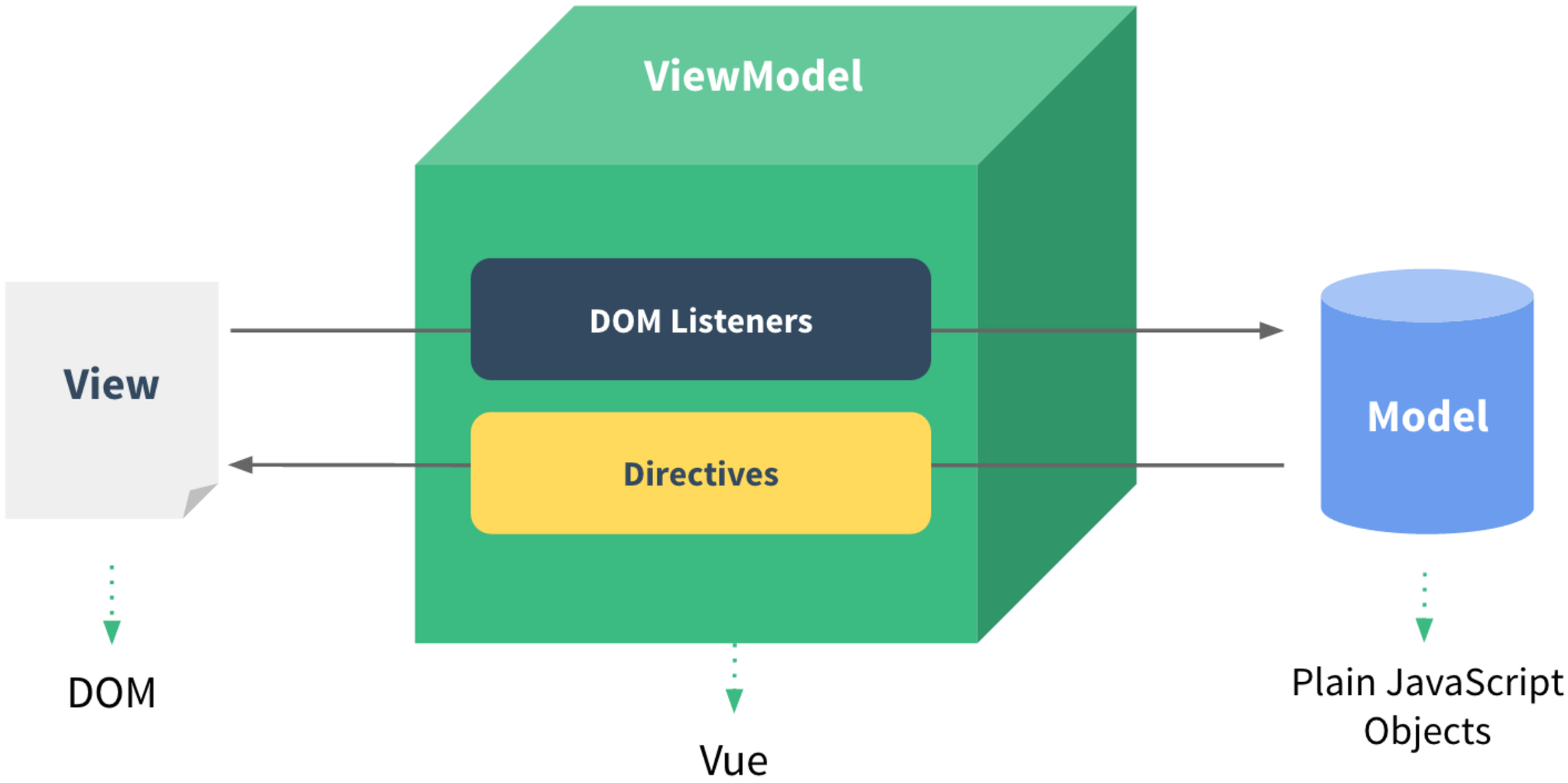
En pratique


Feed View
Feed Controller
Story View
Story Controller
Like Button
Like Controller
Model
does_like = true

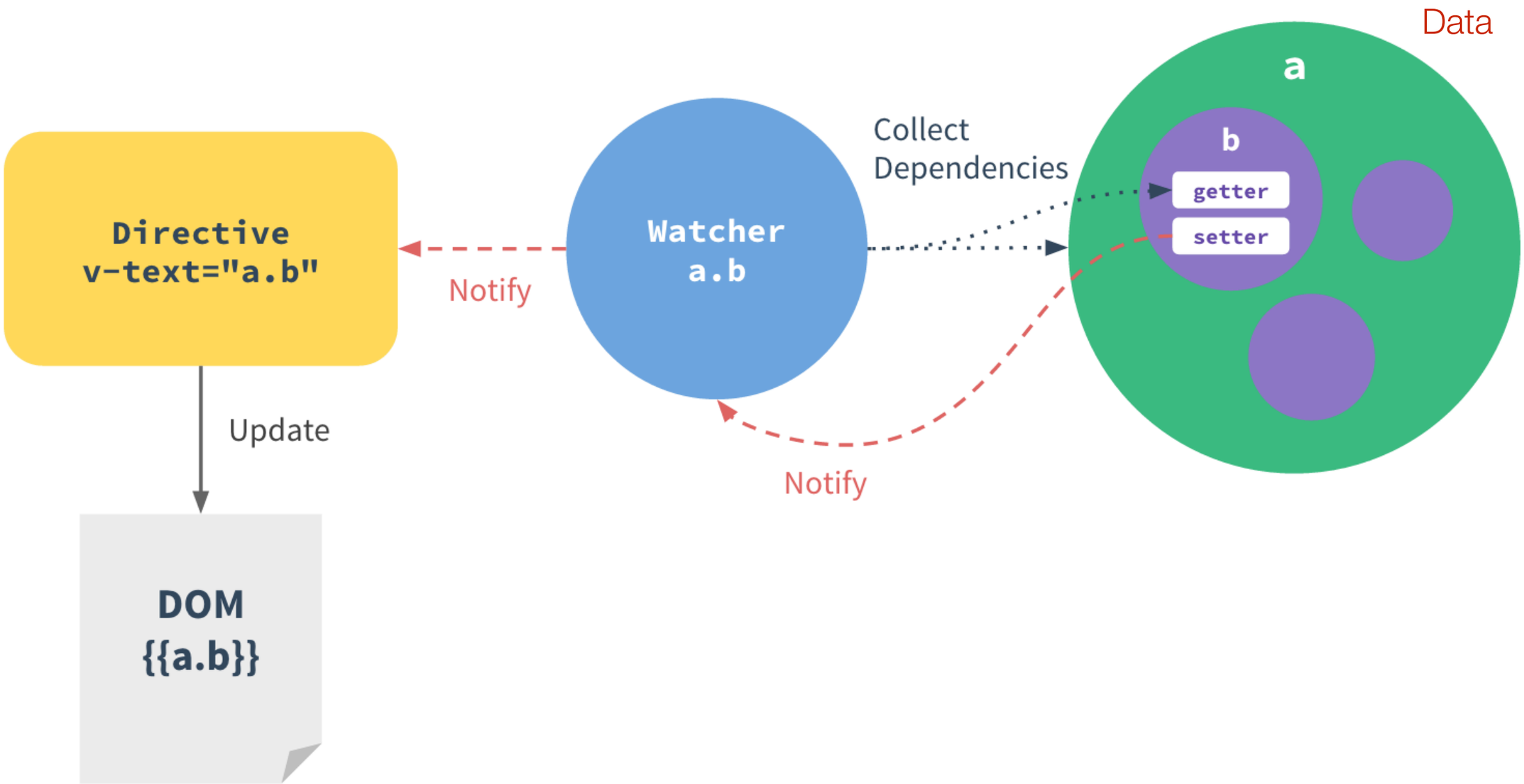
MVVM (ou MVP)

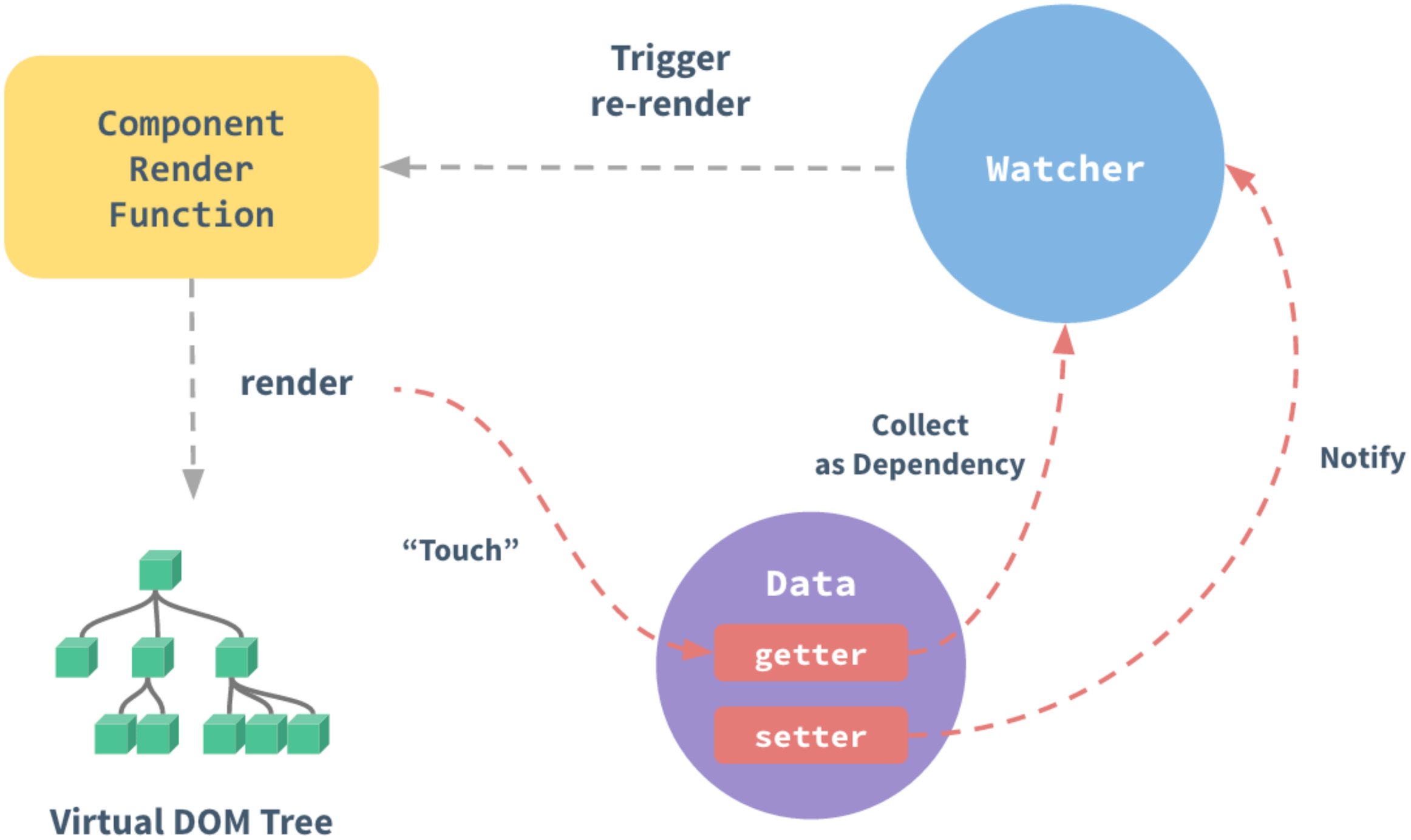

View
Command
2way Binding
Mssenger
ViewModel
Dispatch
Observe
Model

MVVM avec Vue



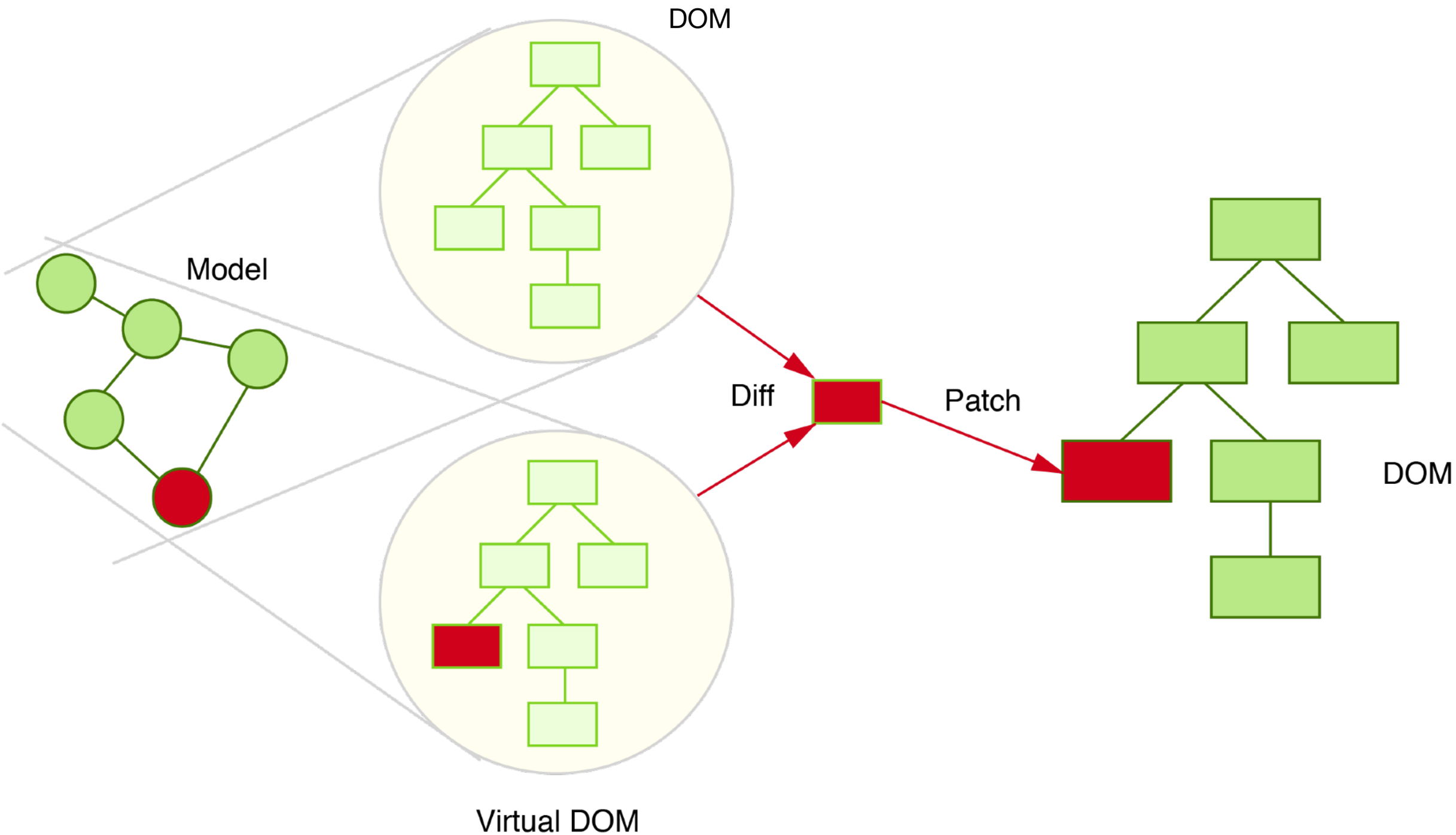
En pratique avec Vue



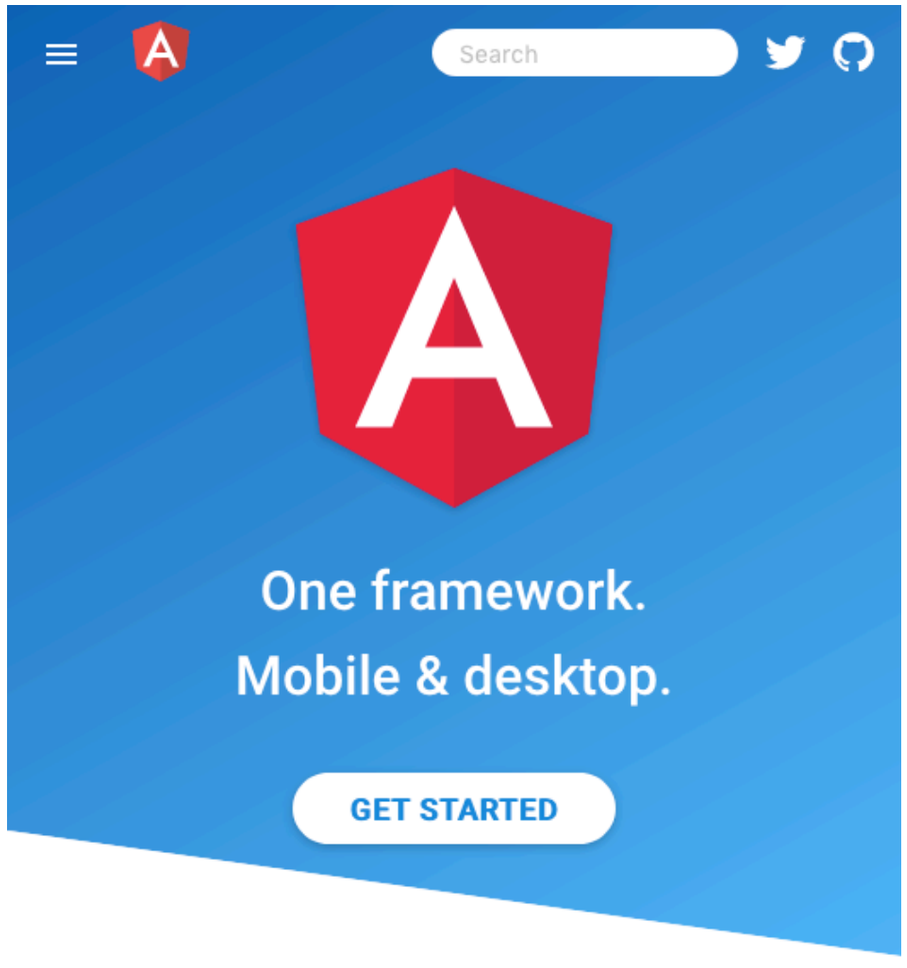

Component
Render
Function
Trigger
re-render
Watcher
render
Collect
as Dependency
Notify
“Touch”
Data
getter
setter
Virtual DOM Tree

Un DOM Virtuel

https://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

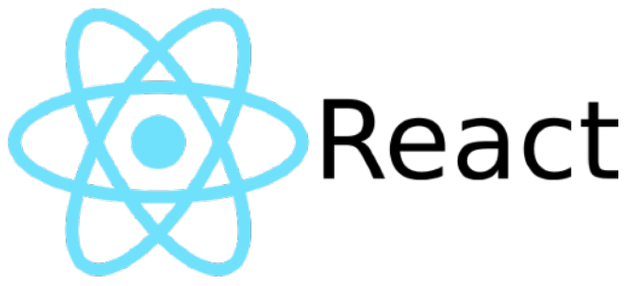


Les bibliothèques Javascript

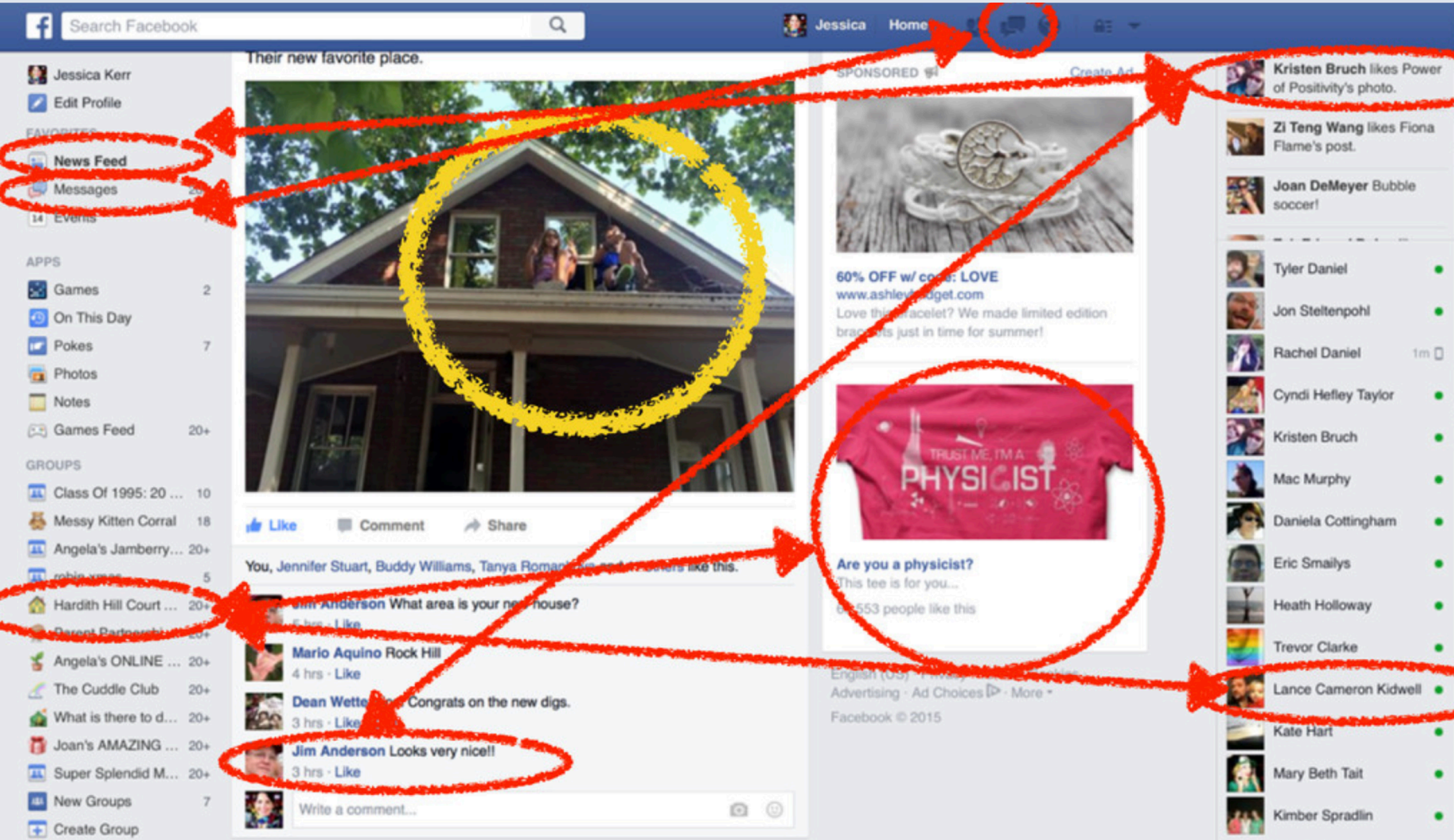


DEVELOP ACROSS ALL PLATFORMS

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.



En pratique


Search Facebook
Jessica
Home
Jessica Kerr
Edit Profile
News Feed
Messages
Events
APPS
Games
On This Day
Pokes
Photos
Notes
Games Feed
GROUPS
Class Of 1995: 20 ...
Messy Kitten Corral
Angela's Jamberry...
Hardith Hill Court ...
Angela's ONLINE ...
The Cuddle Club
What is there to d...
Joan's AMAZING ...
Super Splendid M...
New Groups
Create Group
Their new favorite place.
Like
Comment
Share
You, Jennifer Stuart, Buddy Williams, Tanya Roman
Jim Anderson What area is your new house?
Mario Aquino Rock Hill
4 hrs · Like
Dean Wette
Congrats on the new digs.
3 hrs · Like
Jim Anderson Looks very nice!!
3 hrs · Like
Write a comment...
SPONSORED
Create Ad
60% OFF w/ code: LOVE
www.ashleybridget.com
Love this bracelet? We made limited edition
bracelets just in time for summer!
TRUST ME, I'M A
PHYSICIST
Are you a physicist?
This tee is for you...
553 people like this
Advertising · Ad Choices · More
Facebook © 2015
Kristen Bruch likes Power of Positivity's photo.
Zi Teng Wang likes Fiona Flame's post.
Joan DeMeyer Bubble soccer!
Tyler Daniel
Jon Steltenpohl
Rachel Daniel
1m
Cyndi Hefley Taylor
Kristen Bruch
Mac Murphy
Daniela Cottingham
Eric Smailys
Heath Holloway
Trevor Clarke
Lance Cameron Kidwell
Kate Hart
Mary Beth Tait
Kimber Spradlin

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

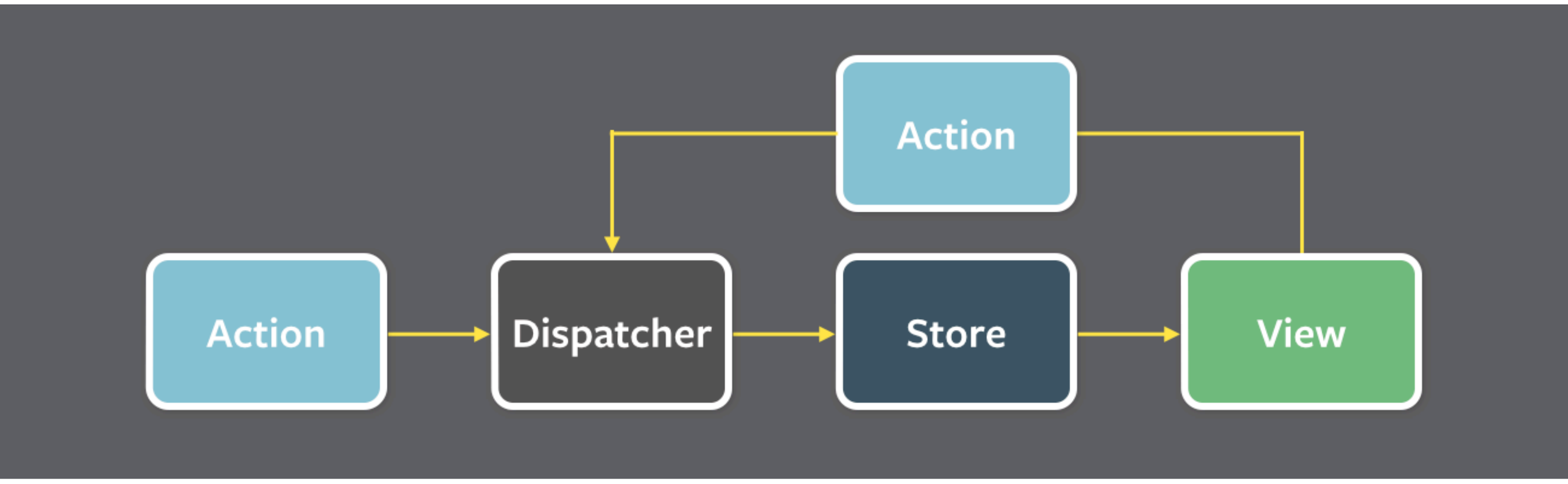
- Quoi et pourquoi la réactivé
- Quelles limites de MVC
- **Architecture Flux**
- Réactivité, Vue et Vuex
- Traitement réactif de flux

Flux de données avec Vue

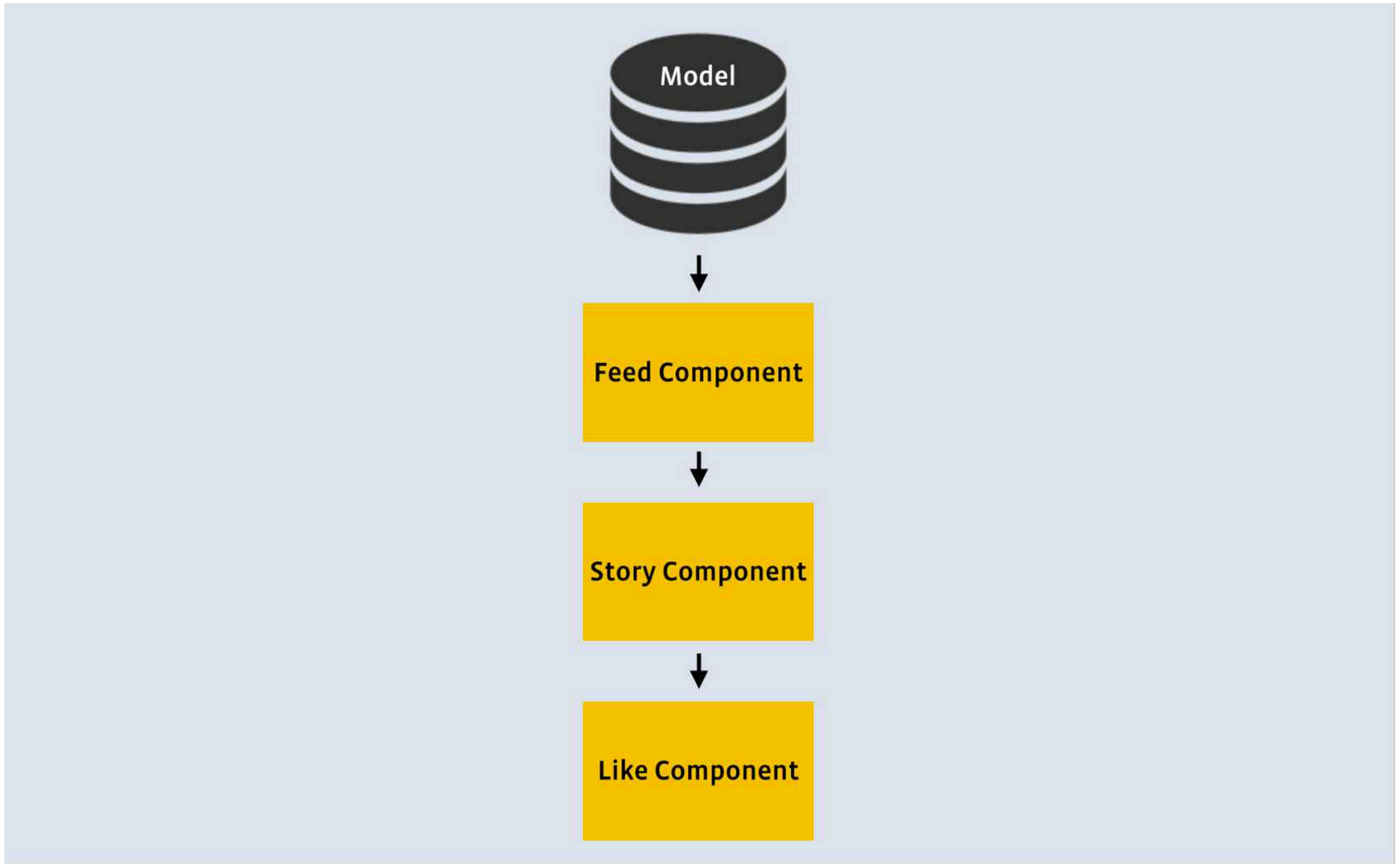
Deux approches au data-binding (lien entre données et vue):

- Two way data-binding
- One way data binding
  Le flux de données est uni-directionnel, les enfants ne modifient pas les données qui sont contrôlées par leur parents.
  Mais les enfants peuvent demander au parent de se mettre à jour (et tout rafraichir depuis le modèle).

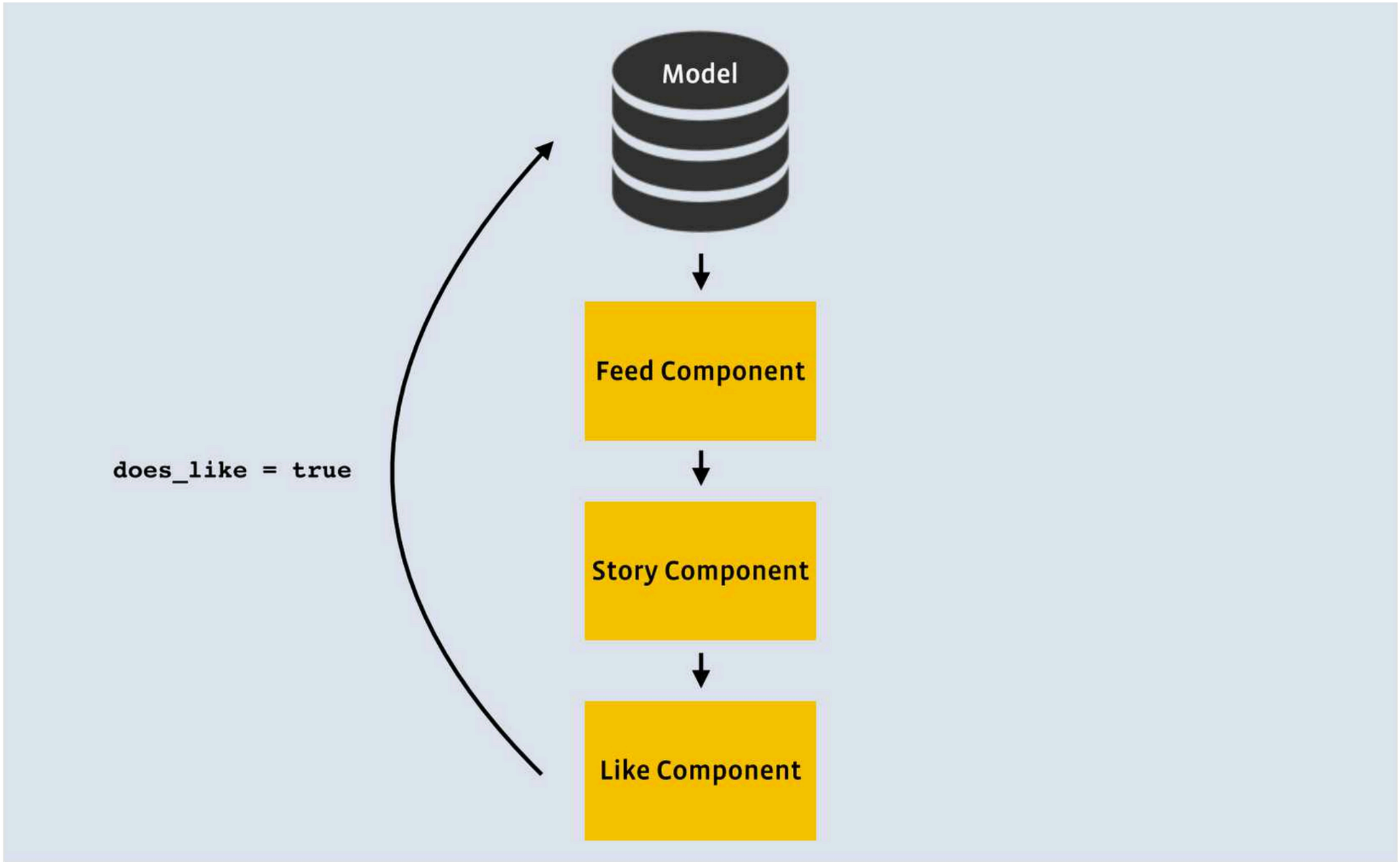
L’architecture Flux



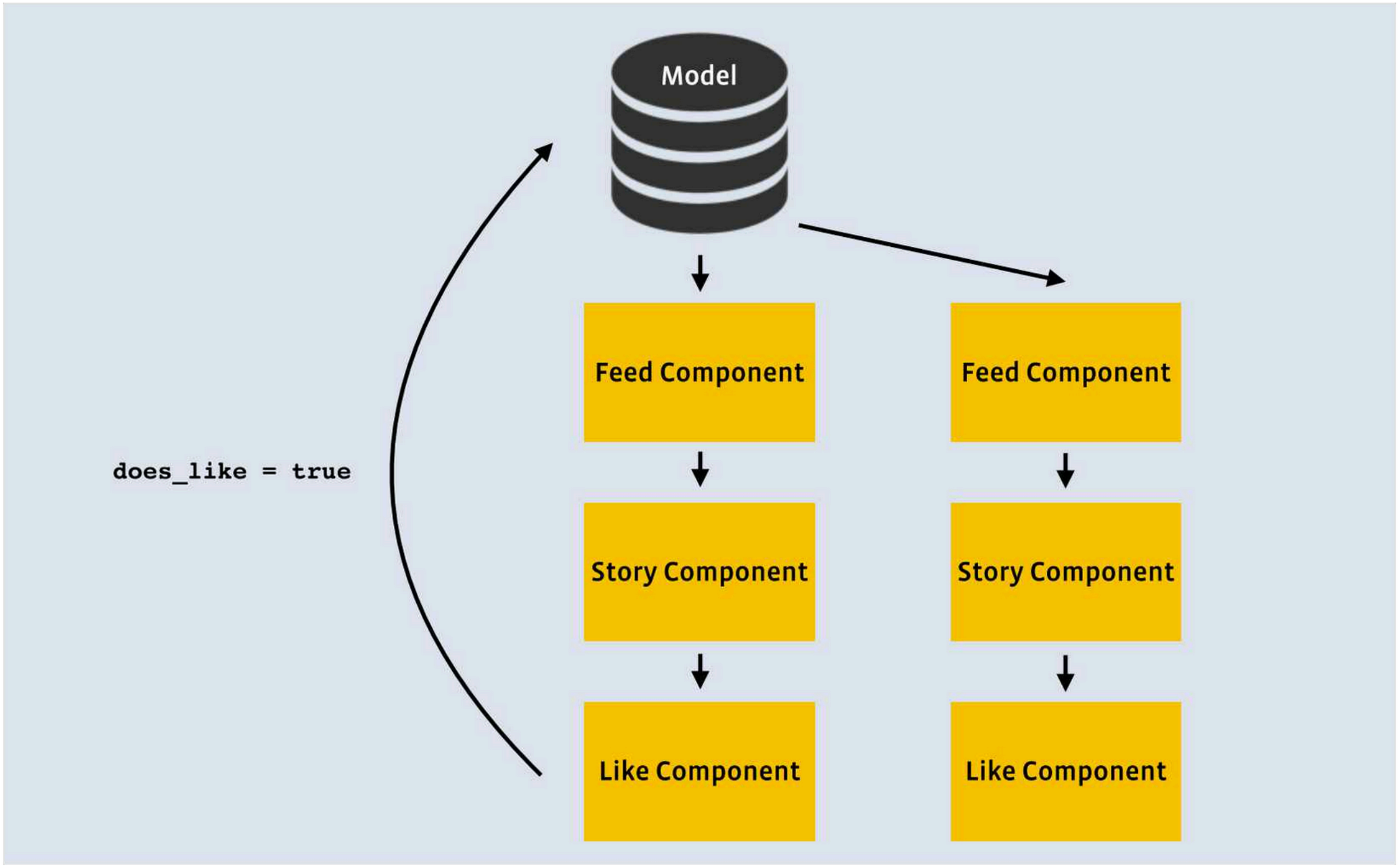
Principe généraux de Flux



On modifie le modèle directement



Nouvel arbre de rendu



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

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

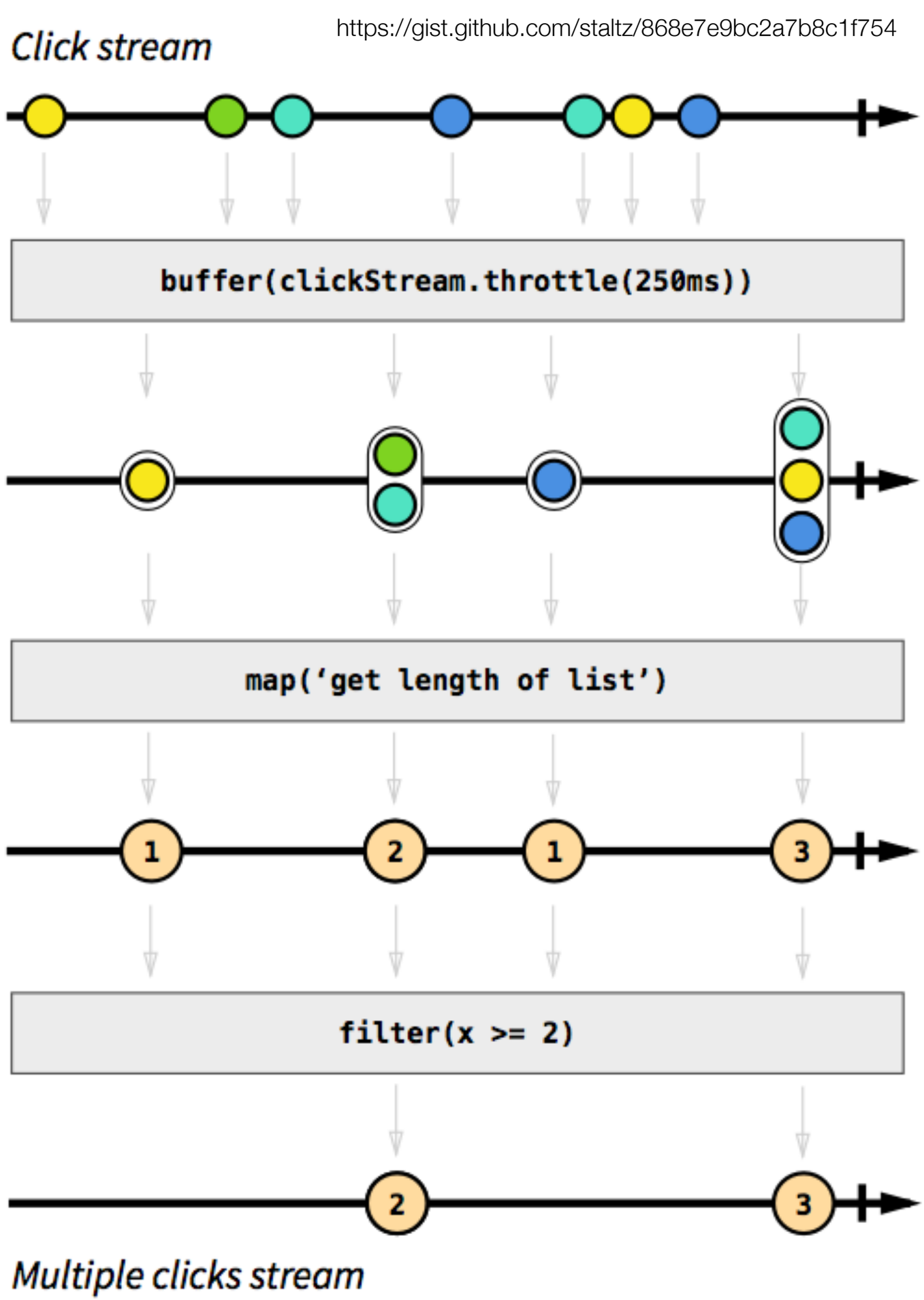
Deux façons de gérer les données

- Données qui changent (mutable) :
  on utilise un état (data)
- Données qui ne changent pas (immutable) :
  on utilise des propriétés (props)

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

Un exemple de transformation

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



Plan

- Quoi et pourquoi la réactivé
- Quelles limites de MVC
- Architecture Flux
- **Réactivité, Vue et Vuex**
- Traitement réactif de flux

Vuex : gestion d'états

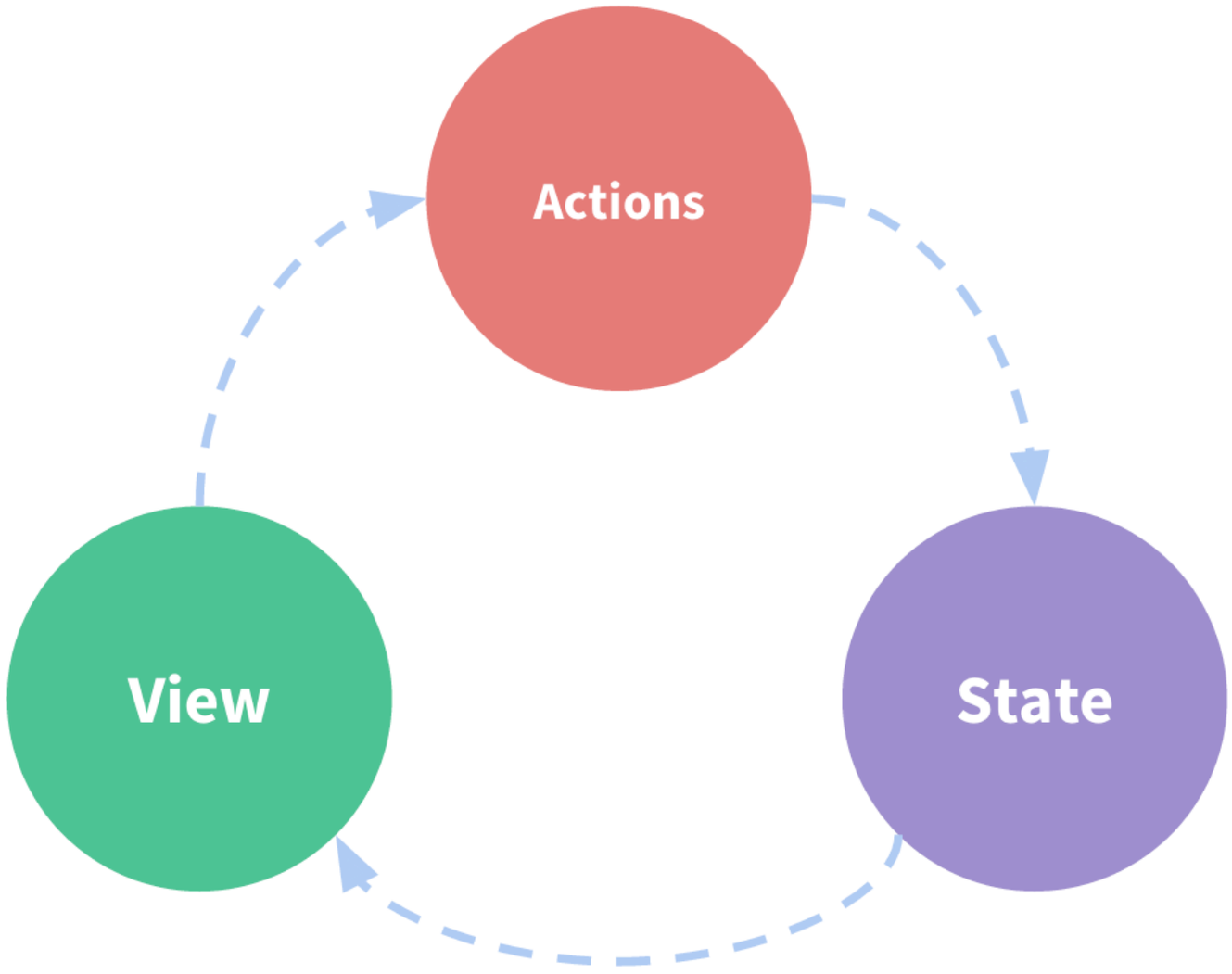
https://vuex.vuejs.org/en/intro.html

One-way data flow

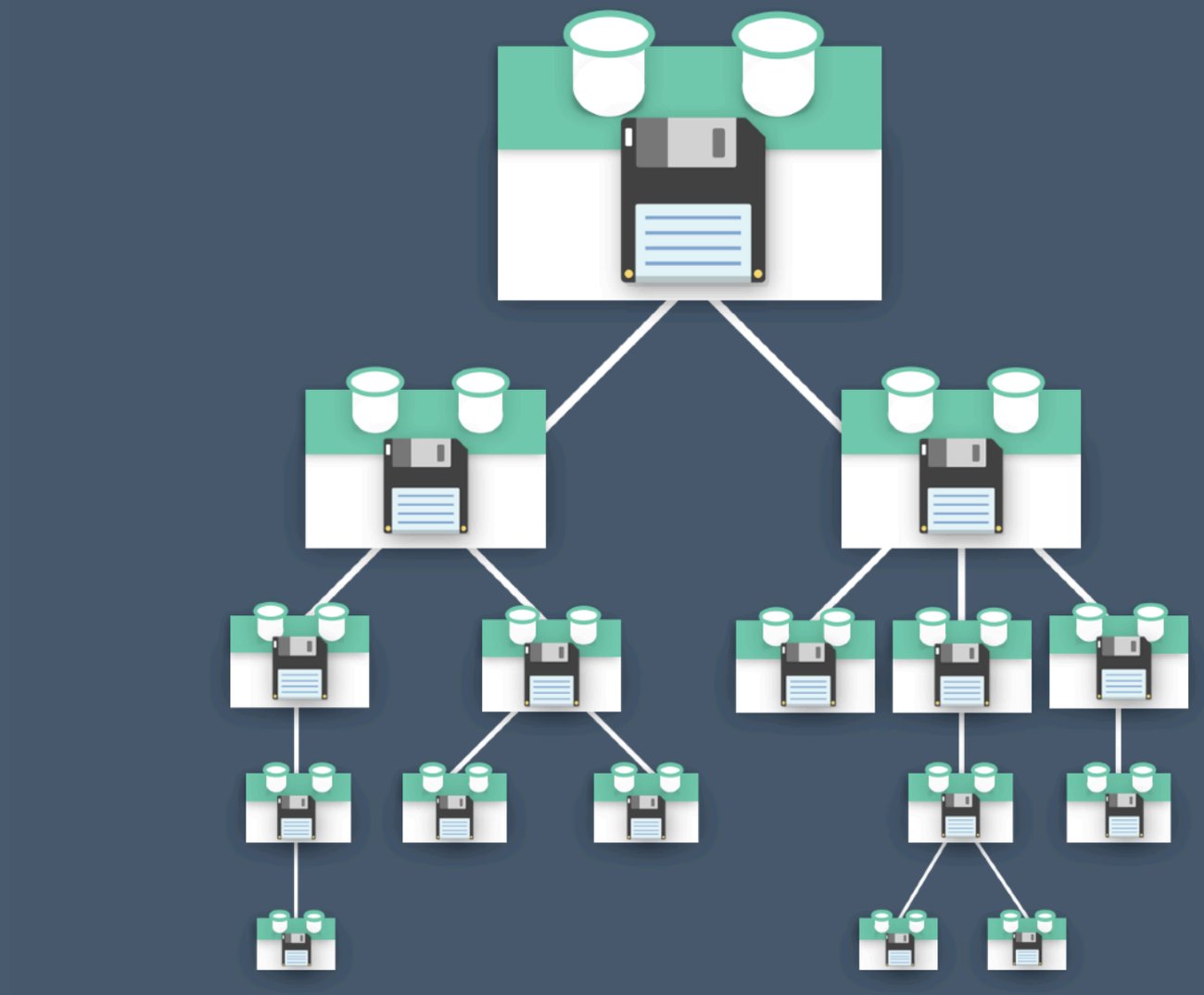
State (état), source de "vérité" pour l'application

View (vue), représentation de l'état (mapping déclaratif)

Actions, changements de l'état en réaction à des entrées de l'utilisateur au niveau de la vue (ou d'autres inputs)

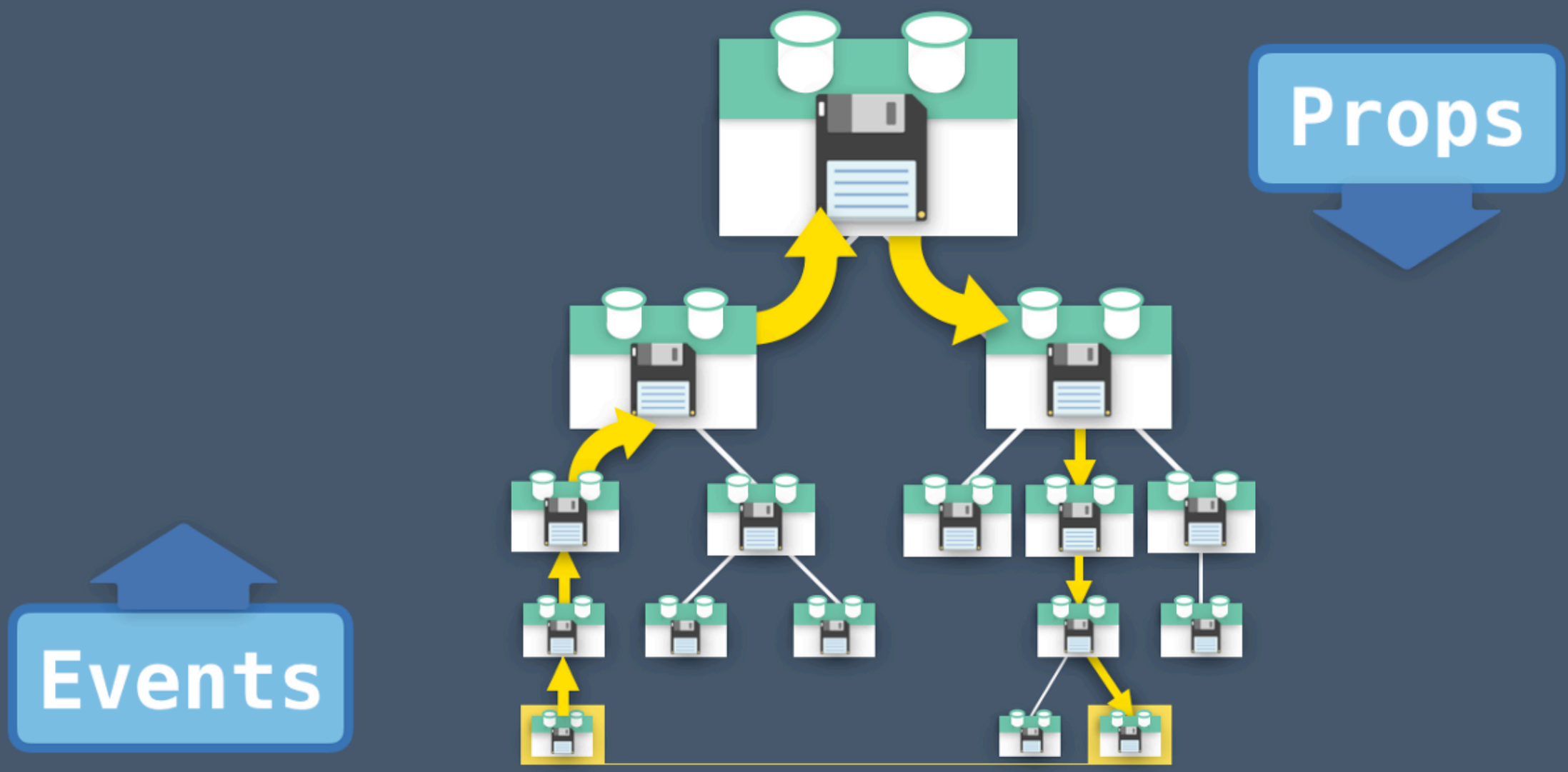


Vue et v-model



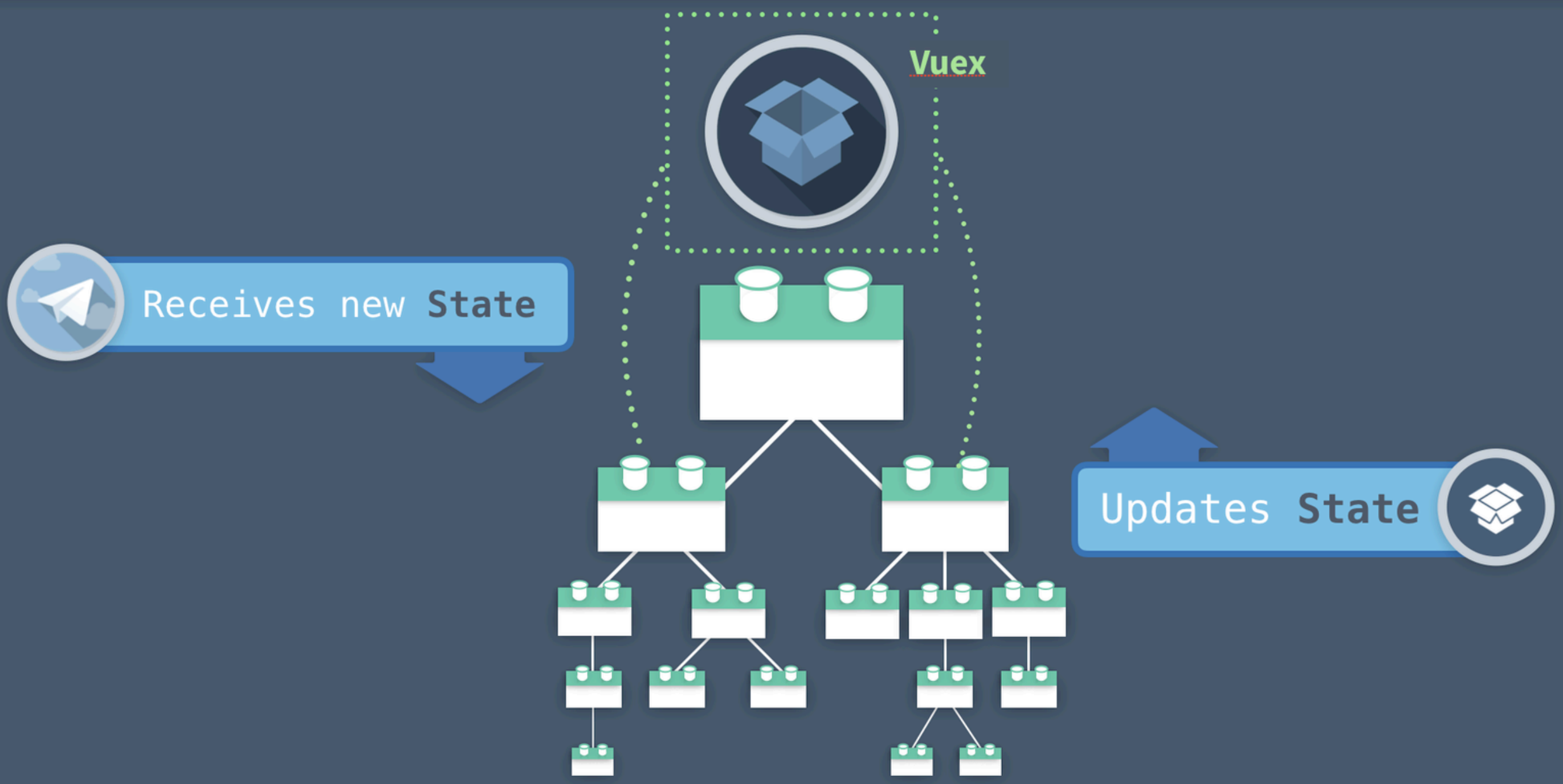
- Un état géré dans chaque composant

Problèmes



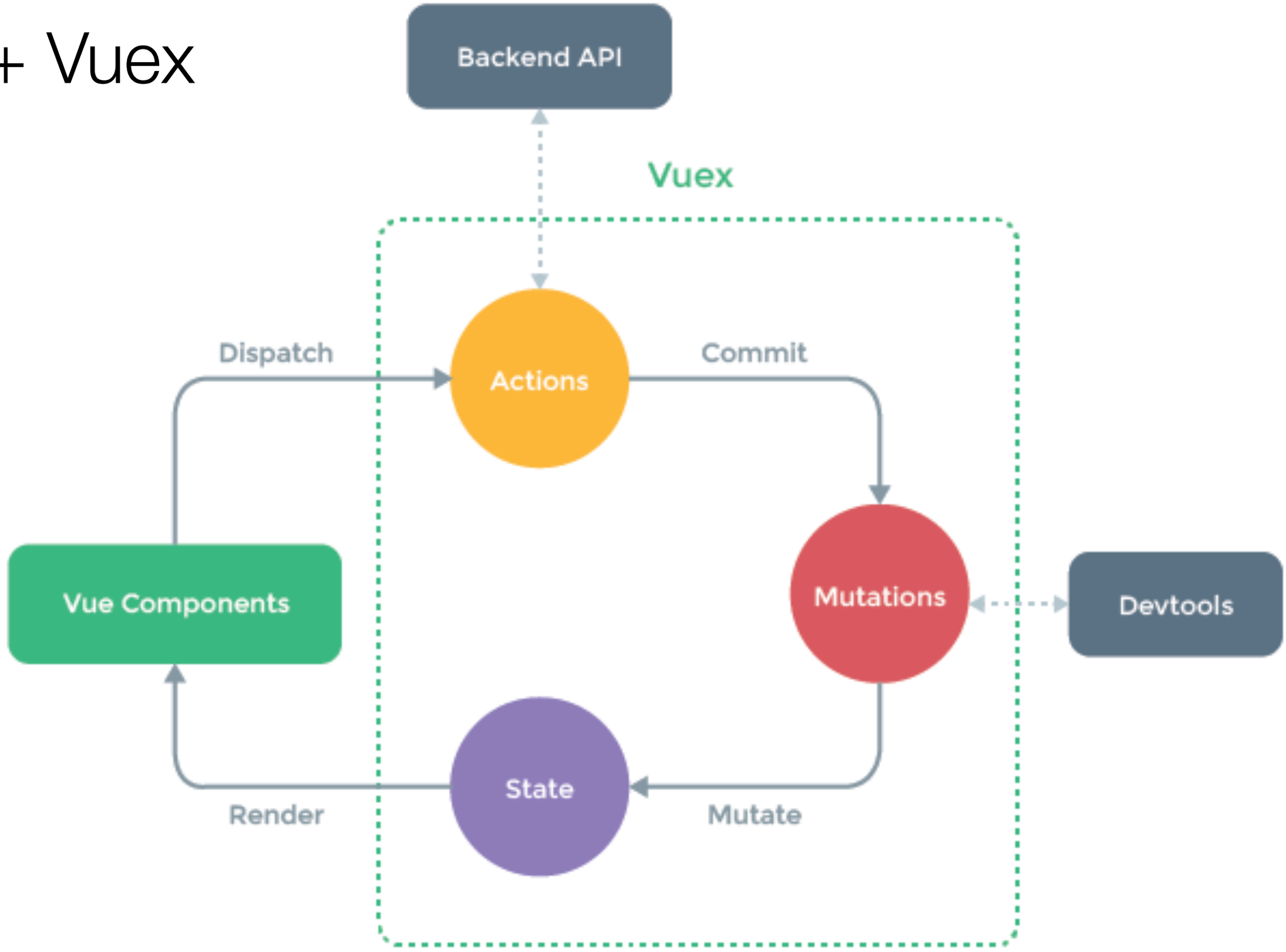
- Plusieurs vues dépendent du même bout d’état
- Plusieurs actions de différentes vues vont changer (mutate) un même bout d’état

Solution: VueX

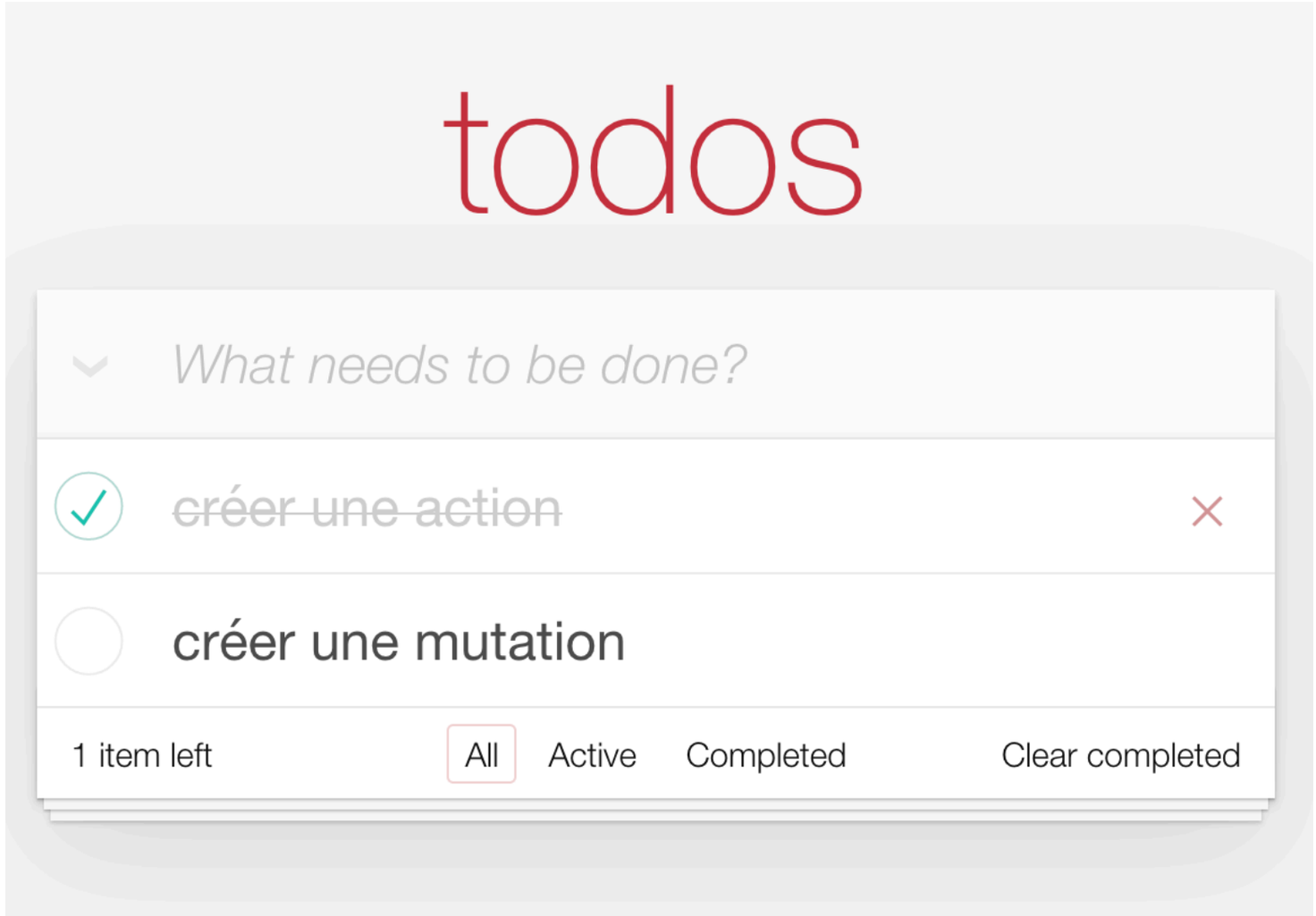


- l’état devient un gros singleton partagé par tous les composants

Vue + Vuex


Backend API
Vuex
Dispatch
Actions
Commit
Vue Components
Mutations
Devtools
State
Render
Mutate

Cas concret




```
▼ <Root>
  ▼ <App>
      <Todo key=0>
      <Todo key=1>
```

```
▼ state
  ▼ todos: Array[2]
    ▼ 0: Object
        done: true
        text: "créer une action"
    ▼ 1: Object
        done: false
        text: "créer une mutation"
```

Structure de projet

```
├── index.html
├── main.js
├── api
│   └── ... # abstractions for making API requests
├── components
│   ├── App.vue
│   └── ...
└── store
    ├── index.js      # where we assemble modules and export the store
    ├── actions.js        # root actions
    ├── mutations.js      # root mutations
    └── modules
        ├── cart.js       # cart module
        └── products.js   # products module
```

Créer son store (vue2+vuex3)

https://vuex.vuejs.org/fr/guide/

```
// app.js
import Vue from 'vue'
import store from './store'
import App from './components/
App.vue'

new Vue({
  store, // inject store to all
children
  el: '#app',
  render: h => h(App)
})
```

```
// store.js
import Vue from 'vue'
import Vuex from 'vuex'
import mutations from './mutations'
import actions from './actions'

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    todos: […] // état initial
  },
  actions,
  mutations
})
```

Créer son store (vue3+vuex4)

https://vuex.vuejs.org/guide/

```
import { createApp } from 'vue'
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state () {
    return {
      todos: […]
    }
  },
  actions,
  mutations,
})

const app = createApp({ /* your root component */ })
app.use(store)
```

Les actions

```
export default {
  addTodo ({ commit }, text) {
    commit('addTodo', {
      text,
      done: false
    })
  },

  removeTodo ({ commit }, todo) {
    commit('removeTodo', todo)
  },

  toggleTodo ({ commit }, todo) {
    commit('editTodo',
           {todo, done: !todo.done}
    )
  },

  editTodo ({ commit }, { todo, value }){
    commit('editTodo',
           {todo, text: value }
    )
  },

  toggleAll ({ state, commit }, done) {
    state.todos.forEach((todo) => {
      commit('editTodo', { todo, done })
    })
  },

  clearCompleted ({ state, commit }) {
    state.todos.filter(todo => todo.done)
      .forEach(todo => {
        commit('removeTodo', todo)
      })
  }
}
```

Les mutations

```
export const mutations = {
  addTodo(state, todo) {
    state.todos.push(todo);
  },

  removeTodo(state, todo) {
    state.todos.splice(state.todos.indexOf(todo), 1);
  },

  editTodo(state, { todo, text = todo.text, done = todo.done }) {
    const index = state.todos.indexOf(todo);

    state.todos.splice(index, 1, {
      ...todo,
      text,
      done,
    });
  },
};
```

Lien entre le composant TodoItem et le store

```
<template>
  <li class="todo"
     :class="{ completed: todo.done, editing:
editing }">
    <div class="view">
      <input class="toggle"
        type="checkbox"
        :checked="todo.done"
        @change="toggleTodo(todo)">
      <label v-text="todo.text"
             @dblclick="editing = true”>
      </label>
      <button class="destroy"
              @click=“removeTodo(todo)">
      </button>
    </div>
    <input class="edit"
      v-show="editing"
      v-focus="editing"
      :value="todo.text"
      @keyup.enter="doneEdit"
      @keyup.esc="cancelEdit"
      @blur="doneEdit">
  </li>
</template>
```

```
import { mapActions } from 'vuex'
export default {
  name: 'Todo',
  props: ['todo'],
  data () {…},
  directives: {…},
  methods: {
    ...mapActions([
      'editTodo',
      'toggleTodo',
      'removeTodo'
    ]),
    doneEdit (e) {
      const value = e.target.value.trim()
      const { todo } = this
      if (!value) {
        this.removeTodo(todo)
      } else if (this.editing) {
        this.editTodo({
          todo,
          value
        })
        this.editing = false
      }
    },
    cancelEdit (e) {…}
  }
}
```

Mutations

- Les mutations Vuex ont un “type” et un “handler”
- Pour déclencher un mutation handler, on appelle store.commit
- On utilise des constantes pour les types des mutations
- Les mutation handlers sont des fonctions **synchrones**

Critiques des mutations

posva commented on 22 Jan 2020 — Member

I think disallowing direct state modification is a rule that should be enforced at a linter level instead because runtime-wise this would only be a dev-only warning, so it would be slower during dev and require more code in the library

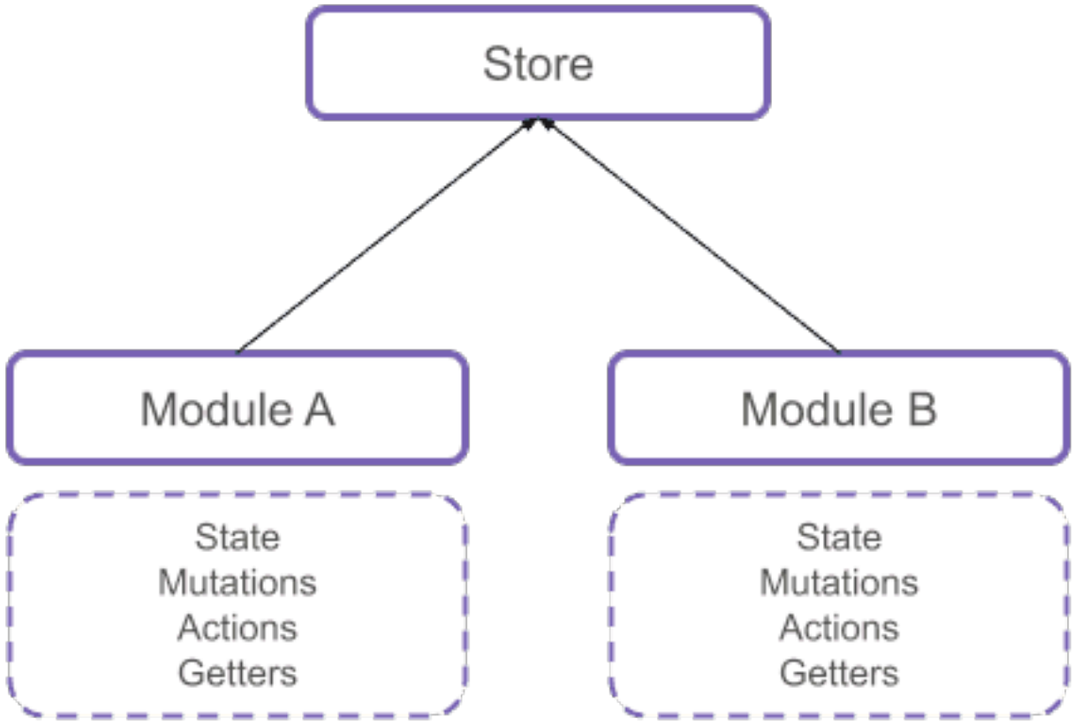
Being able to directly modify the state (or using `patch`) is intentional **to lower the entry barrier and *scale down***. After many years using Vuex, **most mutations were completely unnecessary** as they were merely doing a single operation via an assignment (`=`) or collection methods like `push`. They were **always perceived as verbose**, no matter the size of the project, **adding to the final bundle size as well**, and were useful only when grouping multiple modifications and some logic, usually in larger projects. In Pinia, **you are still able to have these groups** by using actions but you get to choose if you need them or not, being able to start simple, and scale up when necessary.

So I want to make people realise `mutations` is an unnecessary concept thanks to actions and that the cost of mutations and its verbosity is not enough to justify the possibility of organisation it brings. On top of that, one of the main aspects of mutations was devtools: being able to go back and forward. In Pinia, all modifications are still tracked so doing direct state modification still allows you to keep things organised despite it wasn't taught that way with Vuex

👍 26 ❤️ 9

- Création de Pinia (Vuex5)
- Critique similaire côté Redux (React) -> création de redux-toolkit

Vuex4 vs. Pinia



On oublie les mutations

CounterStore.js

```
import { createStore } from 'vuex'

const CounterStore = createStore({
  state() {
    return {
      count: 0
    }
  },
  actions: {
    increment(context) {
      context.commit('incrementCount')
    }
  },
  mutations: {
    incrementCount(state) {
      state.count++
    }
  }
})
```

Vuex without TS

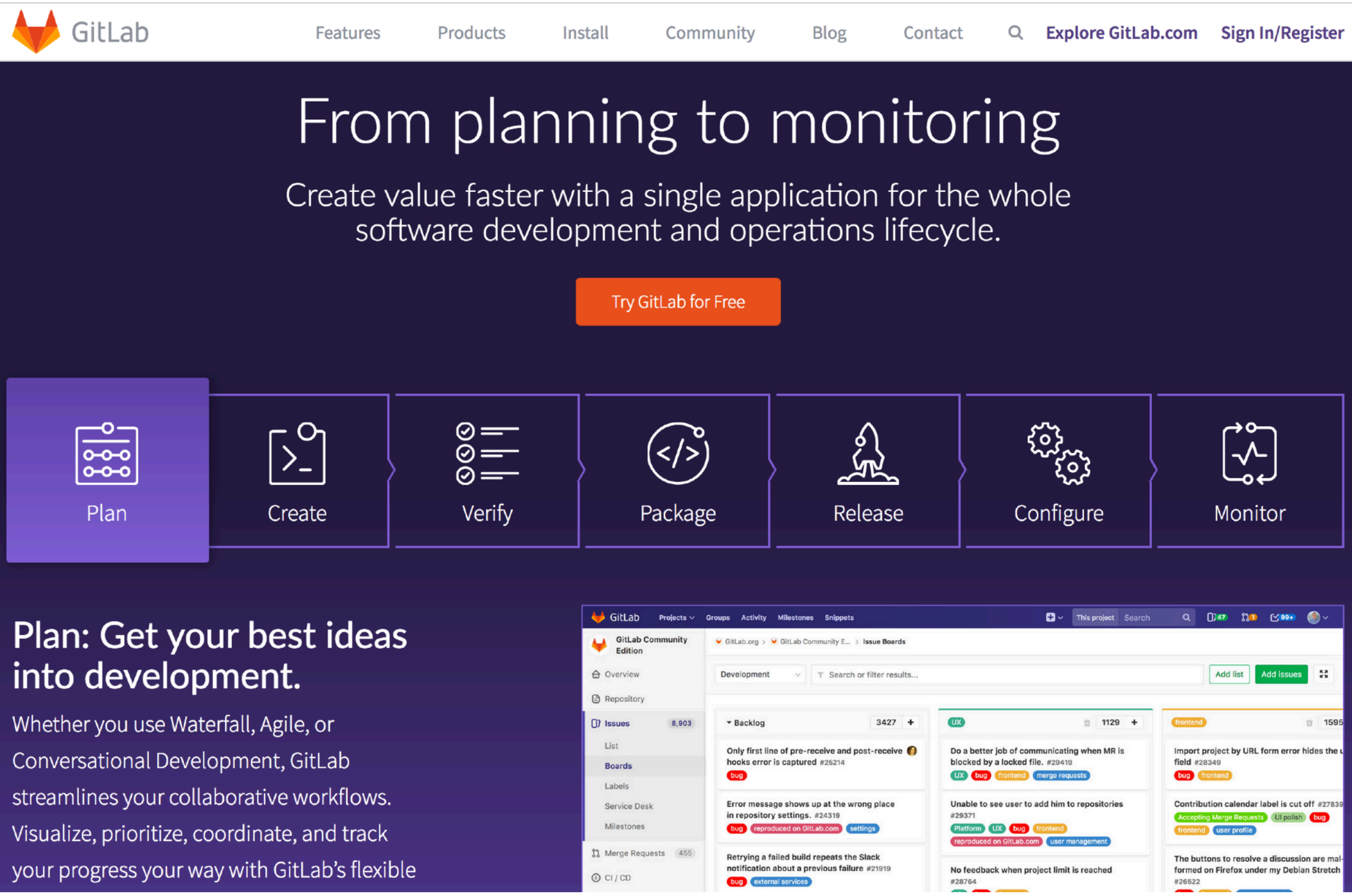
CounterStore.js

```
import { createStore } from 'pinia'

const CounterStore = createStore({
  state() {
    return {
      count: 0
    }
  },
  actions: {
    increment() {
      state.count++
    }
  }
})

// How it would be called in a component

CounterStore.increment()
```

Pinia

Qui utilise Vue ?

https://about.gitlab.com/2016/10/20/why-we-chose-vue/



Au-delà des applications : Substrats et instruments d'interaction

Beyond Applications: Interaction Substrates and Instruments

MICHEL BEAUDOUIN-LAFON, Université Paris-Saclay, CNRS, Inria

Laboratoire Interdisciplinaire des Sciences du Numérique (LISN), France

This paper introduces a new interaction model based on the concepts of *interaction substrate* for organizing digital information and *interaction instruments* for manipulating these substrates. This approach makes the concept of application unnecessary. Instead, it leads to flexible and extensible environments in which users can combine content at will and choose the tools they need to manipulate it. We present STRATIFY, a proof-of-concept implementation that combines a data-reactive approach to specify relationships among digital objects with a functional-reactive approach to handle interaction. This combination enables the creation of rich information substrates that can be freely inspected and modified, as well as interaction instruments that are decoupled from the objects they interact with, making it possible to use instruments with objects they were not designed for. We illustrate the flexibility of the approach with several examples and present directions for future work.

https://hal-pasteur.archives-ouvertes.fr/LISN-EX-SITU/hal-04014963v1

Plan

- Quoi et pourquoi la réactivé
- Quelles limites de MVC
- Architecture Flux
- Réactivité, Vue et Vuex
- **Traitement réactif de flux**

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.

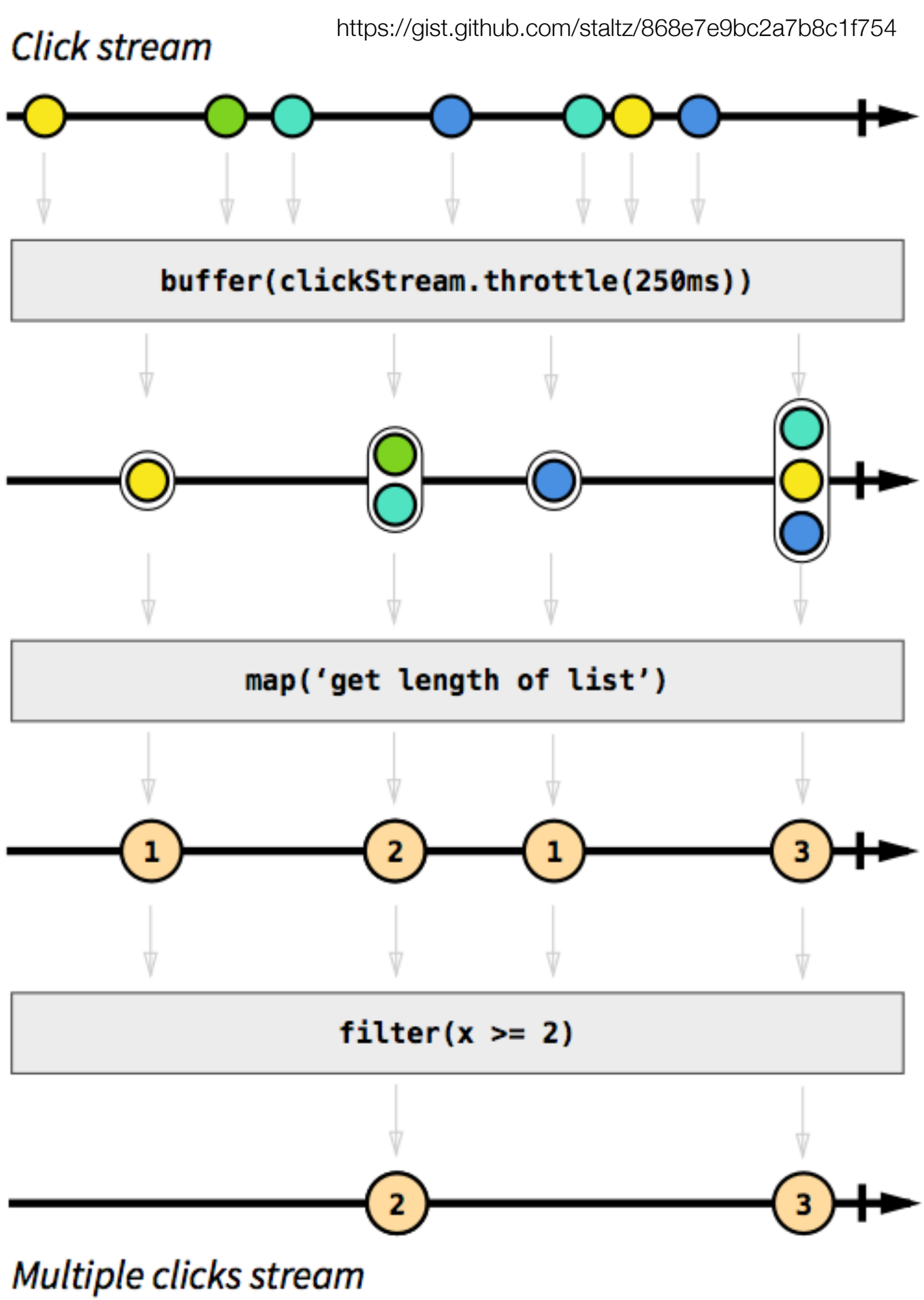
- Pas de point central
- Pas de goulot
- Distribution des entrées entre composants

Orienté message

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

Un exemple de transformation

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



Where

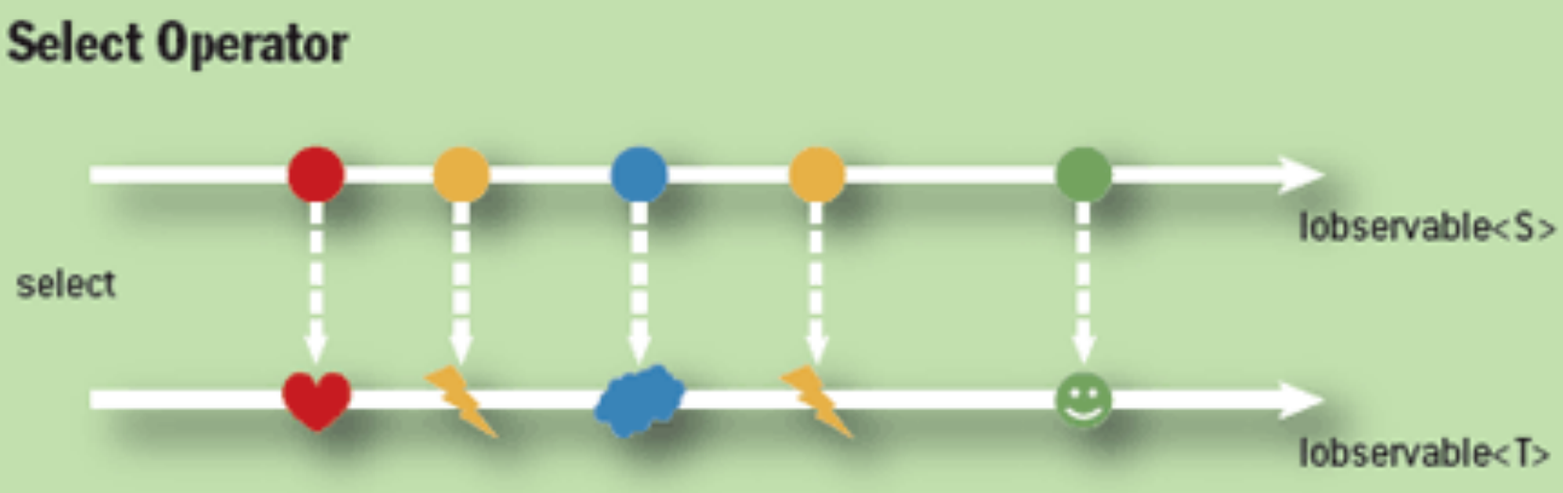
FIGURE 3

The Where Operator

Select

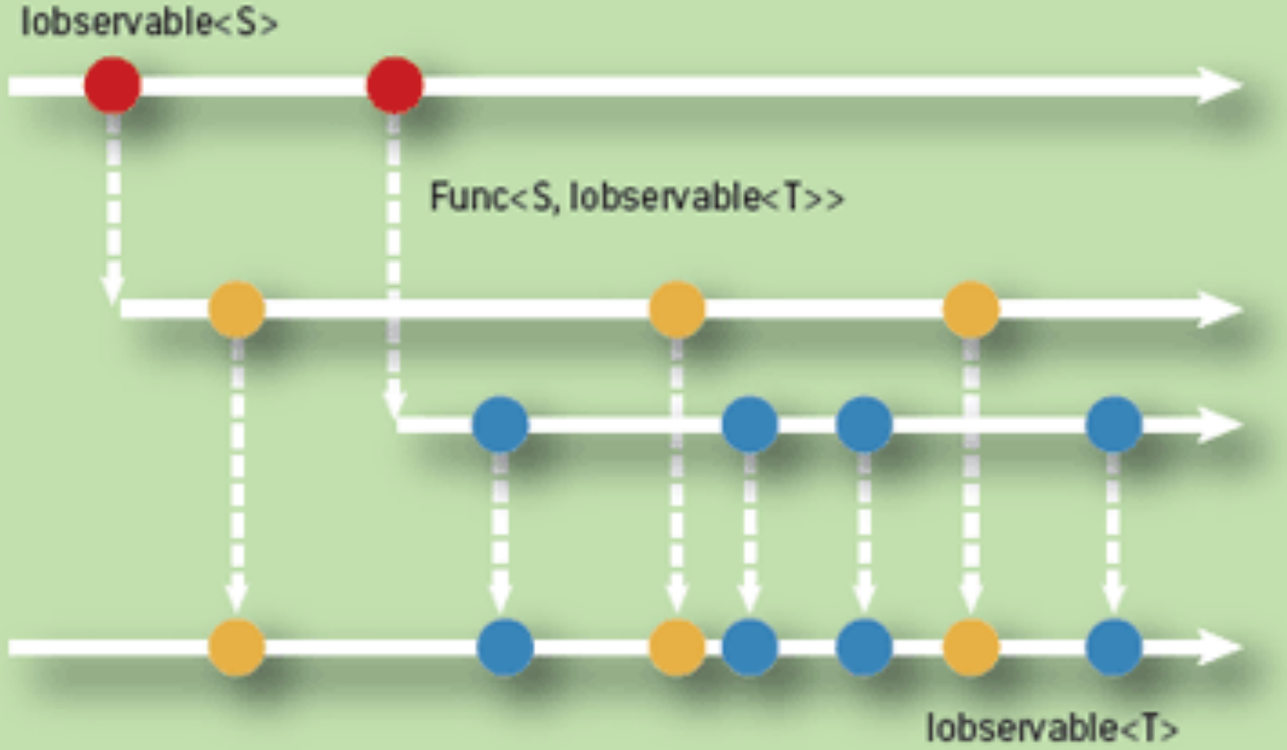
FIGURE 4

The Select Operator



SelectMany : plusieurs flux

FIGURE 5 **The SelectMany Operator**



Throttle

FIGURE 9

The Throttle Operator