Redux

formation.tech

Introduction

Introduction - State of the art

- Redux est une bibliothèque de gestion de state créé pour React par Dan Abramov à l'occasion d'une conférence sur le Time Travel Debugging à React Europe : https://www.youtube.com/watch?v=xsSnOQynTHs
- Concurrents :
  - React : flux (déprécié), MobX, Recoil, Zustand, or Jotai
  - Angular : NgRx, NGXS
  - Vue : Pinia, Vuex
- Redux peut s'utiliser dans un autre contexte que React. On utilise la bibliothèque React Redux pour l'intégration
- Redux Toolkit (RTK) apparu en 2020 simplifie énormément l'utilisation de Redux

Introduction - Quand faut-il utiliser Redux ?

- Selon la documentation :
  https://redux.js.org/introduction/getting-started#should-you-use-redux
  https://redux.js.org/faq/general#when-should-i-use-redux

  - Vous avez une grande quantité de données dans le state qui sont utilisées à plein d'endroit de l'application
  - Le state est mise à jour fréquemment dans le temps
  - La logique pour mettre à jour le state peut-être complexe
  - L'application a une moyenne à grande base de code, et a été créé par plusieurs développeurs
  - Vous avez besoin de savoir comment le state est mis à jour dans le temps
  - Vous avez besoin d'une source unique pour votre state
  - Vous trouvez que garder tout votre state dans un votre composant racine n'est plus suffisant

Introduction - Quand faut-il utiliser Redux ?

- La même donnée est-elle utilisée pour piloter différents composants ?
- Avez vous besoin de créer de nouvelles données dérivées de ces données ?
- Y-a-t'il une valeur que vous souhaiteriez restaurer dans un état précédent (time travel debugging, undo/redo) ?
- Voulez-vous mettre cette donnée en cache ?
- Pour profiter de son écosystème plutôt que de réinventer les cas courants : https://redux.js.org/introduction/ecosystem

Introduction - State React vs State Redux ?

- State React vs State Redux ?
  https://github.com/reduxjs/redux/issues/1287#issuecomment-175351978
- Utilisez React pour un state éphémère qui n'impacte pas l'application globalement comme l'ouverture d'un menu ou une valeur de formulaire
- Utilisez Redux pour un state qui impacte l'app globalement ou qui est modifié de manière complexe. Par exemple, des utilisateurs mise en cache ou un brouillon d'article de blog

Introduction - Flux

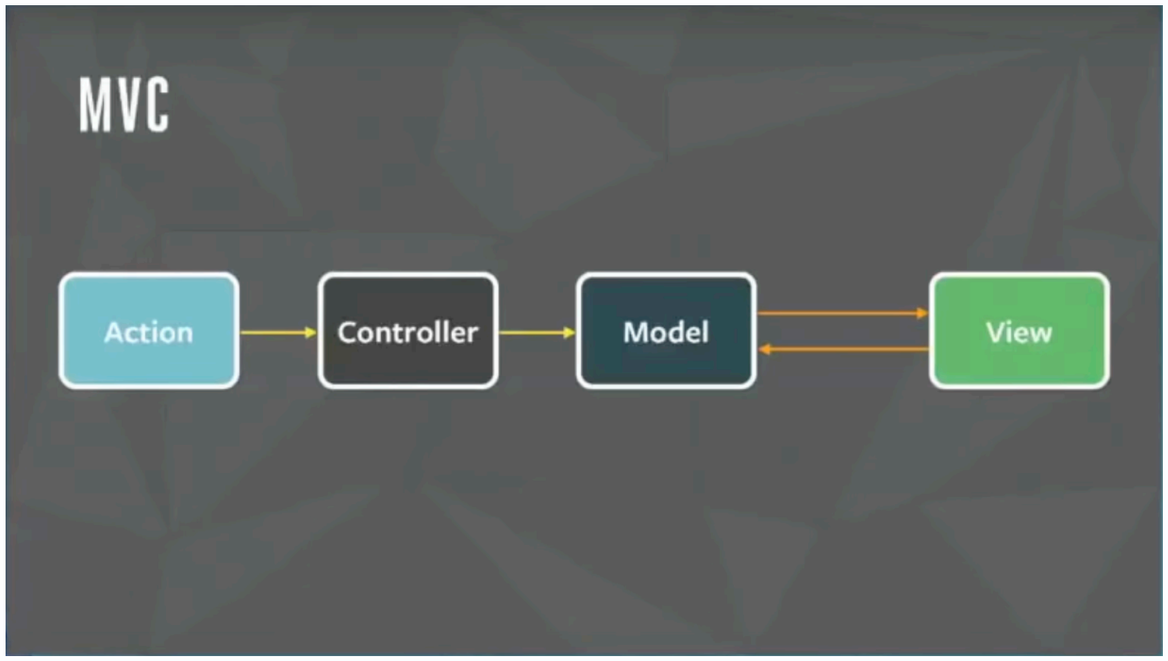
- Inspiré par Flux (Facebook)
  Redux est inspiré de Flux, une architecture proposée par Facebook pour les applications front-end présentée pour la première fois dans cette conférence : https://www.youtube.com/watch?v=nYkdrAPrdcw

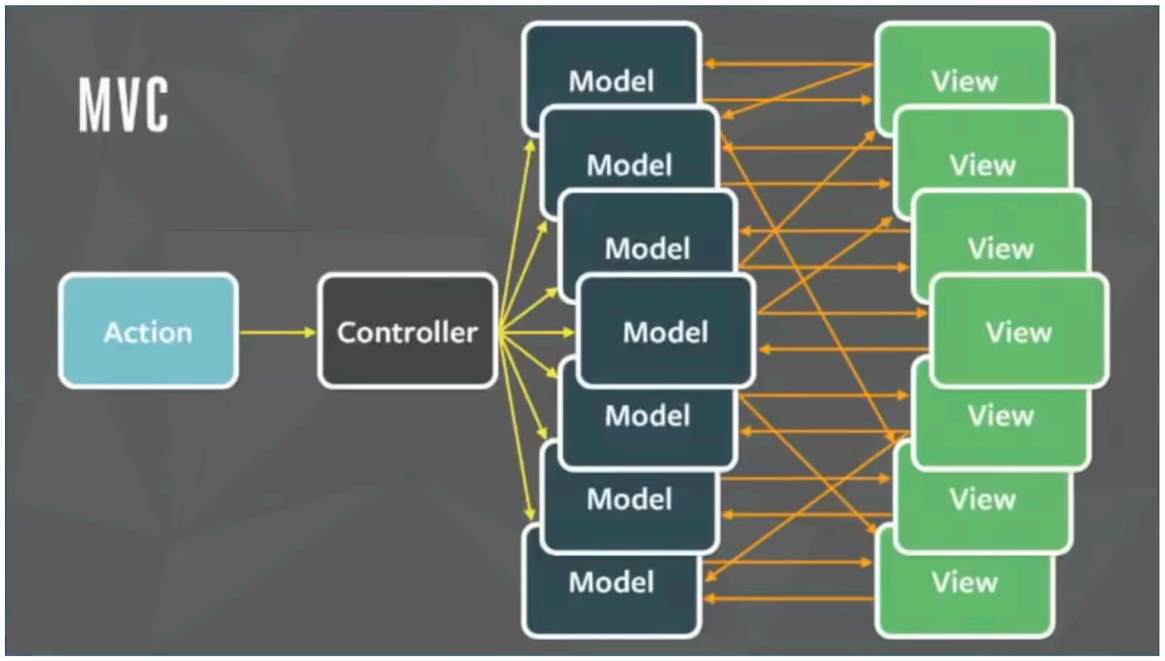
- En plus de Flux, d'autres patterns sont suivis :

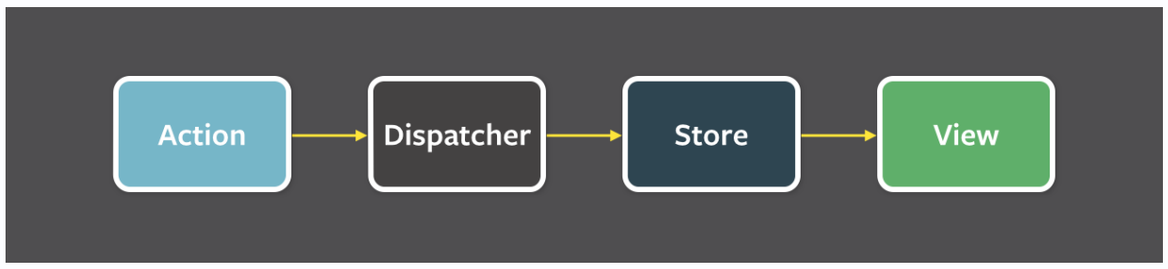
  - Observer : https://fr.wikipedia.org/wiki/Observateur_(patron_de_conception)
  - CQRS : https://martinfowler.com/bliki/CQRS.html
  - Event Sourcing : https://martinfowler.com/eaaDev/EventSourcing.html

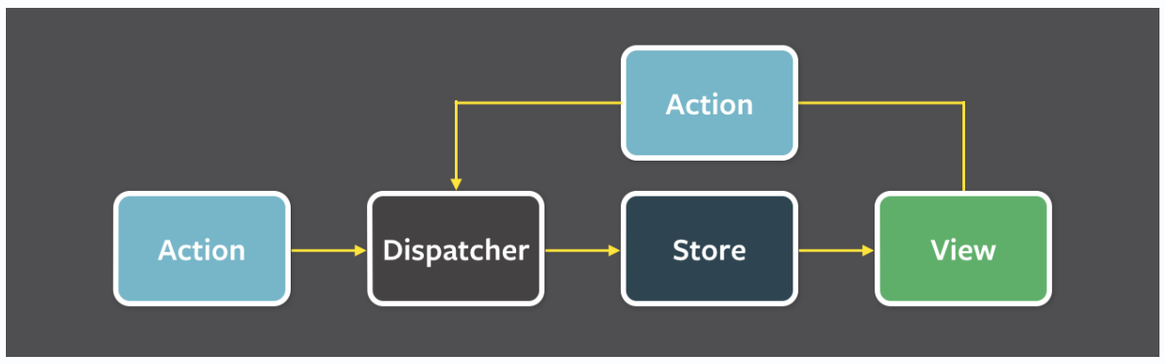
Introduction - MVC vs Flux

- En résumé, l'idée est de passer de MVC qui implique un échange bidirectionnel entre le model et la vue à Flux qui propose un échange unidirectionnel (Unidirectional Data Flow)



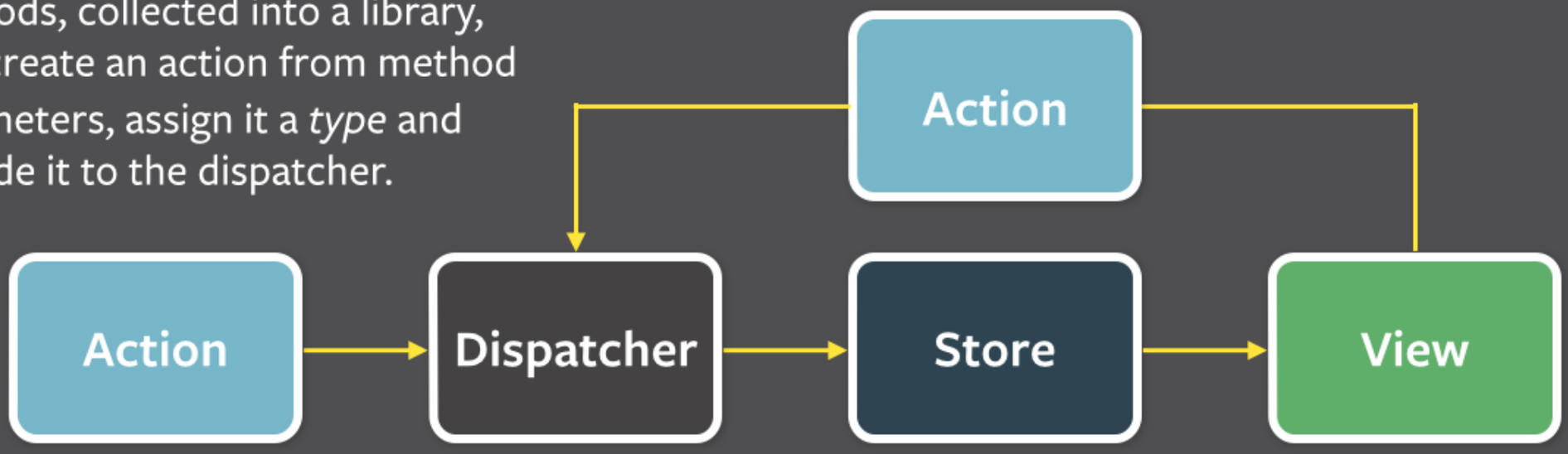






Introduction - Concepts de Flux

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.

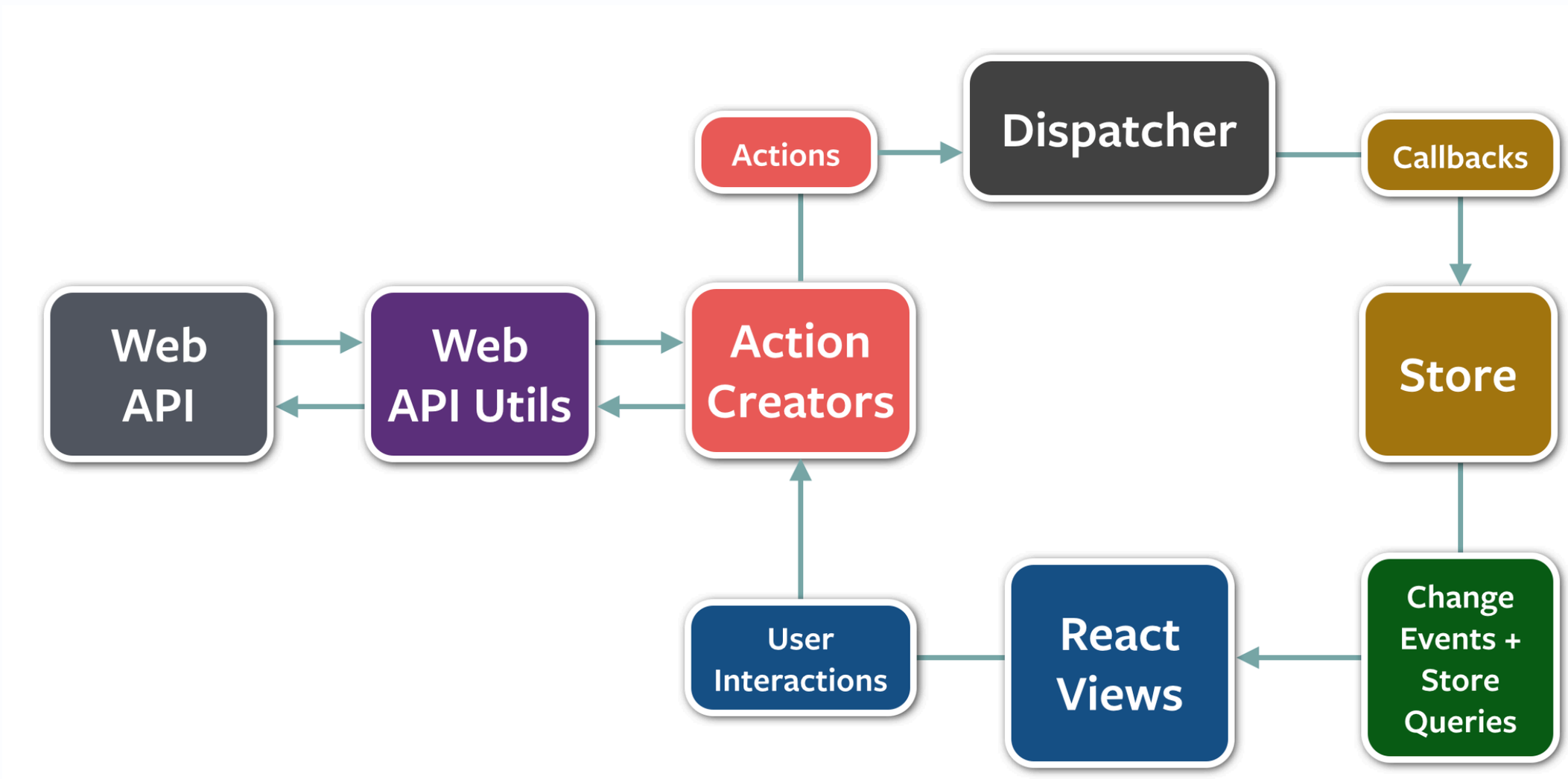


Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

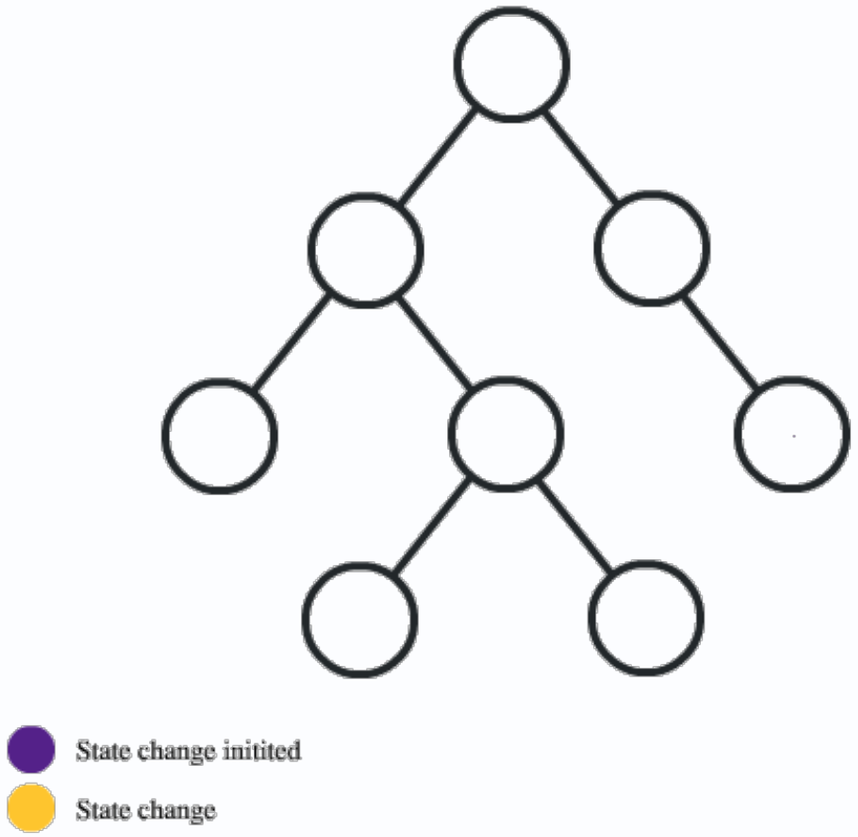
After stores update themselves in response to an action, they emit a *change* event.

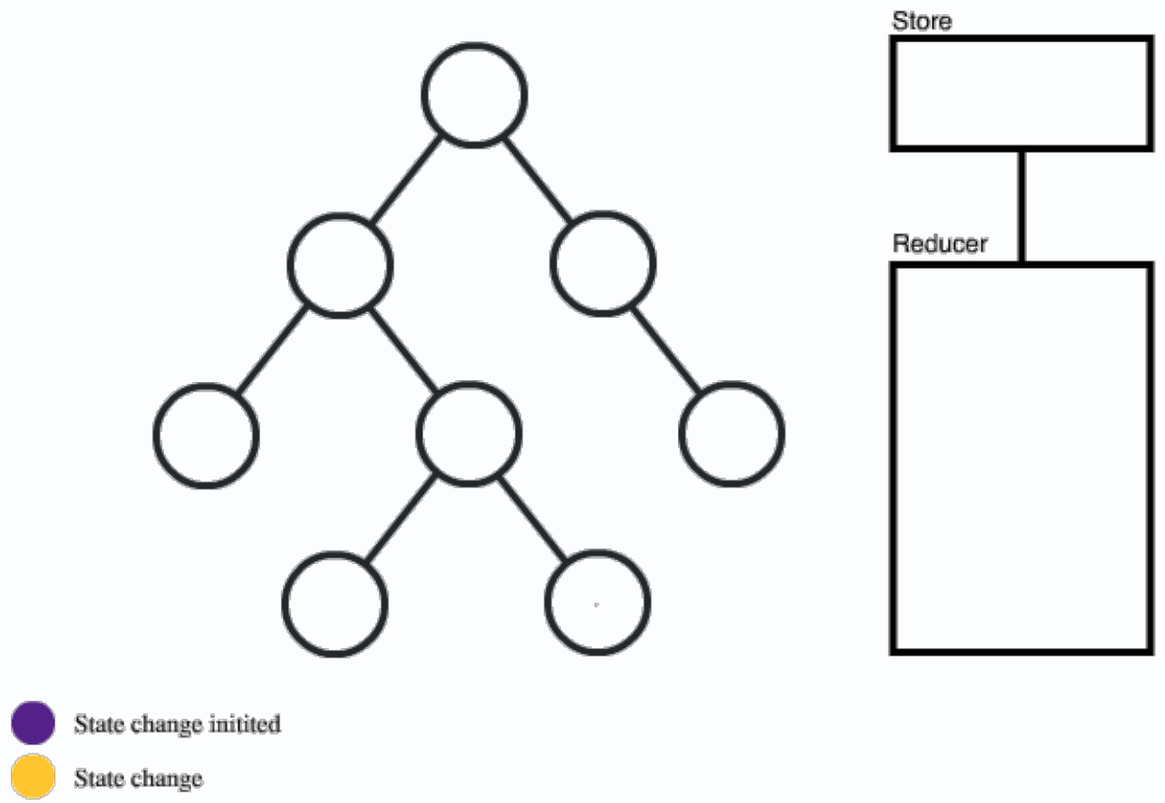
Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

Introduction - Architecture Redux



Introduction - Echanges inter-composants





formation.tech

Mise en place

Mise en place - Installation

- Redux
  - `npm install redux`
  - `yarn add redux`

Mise en place - State

- https://redux.js.org/understanding/thinking-in-redux/three-principles
- Single source of truth : le state global est défini sous forme d'un objet présent dans un store unique
- Le state est en lecture seule, on y apporte des modifications au travers de fonctions pures
- Le state doit être sérialisable (ne pas y mettre de fonctions...)
- Il faut penser son state comme une base de données, ne pas dupliquer la données...
-
```
const initialState = {
  name: 'Romain',
  likes: 10,
};
```

Mise en place - Reducer

- Le reducer est la fonction (callback) qui permet de mettre à jour le state
- Le reducer est une fonction pure :
  - elle est prédictive, appelée avec un state et un action donnée elle a toujours le même retour
  - elle ne modifie pas ses paramètres, elle retourne un nouvel objet state (immuabilité)
  - elle n'a de side-effect, elle n'appelle pas de fonctions externes qui ne sont pas pures (fetch, localStorage...)

```
// fonction pure
function addition(a, b) {
  return Number(a) + Number(b);
}
```

```
// fonctions impures
function getRandomIntInclusive(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function validateUser(user) {
  localStorage.setItem('user', user);
  return user === 'Romain';
}

function userToUpperCase(user) {
  user.prenom = user.prenom.toUpperCase();
  return user;
}
```

Mise en place - Reducer

- Le reducer est une fonction qui reçoit state, action et retourne le nouveau state :

```
const nextState = reducer(state, action);
```

- Exemple :

```
function reducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT_LIKES':
      return {
        ...state,
        likes: state.likes + 1,
      };
    case 'UPDATE_NAME':
      return {
        ...state,
        name: action.newName,
      };
    default:
      return state;
  }
}
```

- Redux va appeler le reducer une première fois pour initialiser le state avec initialState
- Si l'action n'a pas a être traitée on se contente de retourner le state précédent

Mise en place - Store

- Le store est le point d'entrée dans Redux, il contient 4 méthodes :
  - getState
  - dispatch
  - subscribe
  - replaceReducer

- On le créé à partir de la fonction createStore (déprécié) ou legacy_createStore car Redux encourage désormais l'utilisation de la bibliothèque Redux Toolkit

```
const store = legacy_createStore(reducer);
```

- 2 paramètres optionnel, preloadedState (utilisé si le state a été sérialité lors d'une visite précédente et enhancer qui permet d'appliquer des middlewares (plugins)

```
const store = legacy_createStore(
  reducer,
  JSON.parse(localStorage.getItem('store')),
  applyMiddleware(logger),
);
```

Mise en place - Actions et dispatcher

- Pour mettre à jour on doit appeler la fonction dispatch du store avec un objet action en paramètre
- L'action doit obligatoirement contenir une clé *type*, de préférence de type string (on évite les Symbol qui compliquent l'utilisation des outils de développement)

```
store.dispatch({ type: 'INCREMENT_LIKES' });
store.dispatch({ type: 'UPDATE_NAME', newName: 'Toto' });
```

- Les versions récentes de la doc recommande l'utilisation de type de la forme : domain/eventName par exemple :
  - likes/incrementLikes
  - name/updateName

Mise en place - Lire le state

- Pour lire le state on utile store.getState()
- A chaque fois que le store est mis à jour par le reducer suite à un dispatch, Redux appellera le callback de store.subscribe() :

```
store.subscribe(() => {
  console.log('current state', store.getState());
  console.log('name', store.getState().name);
  console.log('likes', store.getState().likes);
});
```

formation.tech

Immuabilité

Immuabilité - Introduction

- Lors de la modification d'un objet, le changement peut-être muable en modifiant l'objet d'origine ou immuable en créant un nouvel objet
- Les algorithmes de détections de changements préfèreront les changements immuables, ayant ainsi juste à comparer les références plutôt que l'ensemble du contenu de l'objet
- Exemple, en JS les tableaux sont muables, les chaines de caractères immuables

```
const firstName = 'Romain';
firstName.concat(' Edouard');
console.log(firstName); // Romain

const firstNames = ['Romain'];
firstNames.push('Edouard');
console.log(firstNames); // Romain,Edouard
```

Immuabilité - Tableaux

- Ajouter à la fin

```
const firstNames = ['Romain', 'Edouard'];

function append(array, value) {
  return [...array, value];
}

const newfirstNames = append(firstNames, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
const firstNames = ['Romain', 'Edouard'];

function prepend(array, value) {
  return [value, ...array];
}

const newfirstNames = prepend(firstNames, 'Jean');
console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Ajouter à un indice donné

```
const firstNames = ['Romain', 'Edouard'];

function insertAt(array, value, i) {
  return [
    ...array.slice(0, i),
    value,
    ...array.slice(i),
  ];
}
```

```
const newfirstNames = insertAt(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Modifier un élément

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  return [
    ...array.slice(0, i),
    value,
    ...array.slice(i + 1),
  ];
}
```

```
const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Modifier un élément (alternative)

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  const newArray = [...array];
  newArray[i] = value
  return newArray;
}

const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Modifier un élément (alternative avec .map)

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  return array.map((el, currentI) => (i === currentI ? value : el));
}

const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Supprimer un élément

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  return [
    ...array.slice(0, i),
    ...array.slice(i + 1),
  ];
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Supprimer un élément (alternative)

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  const newArray = [...array];
  newArray.slice(i, 1);
  return newArray;
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Supprimer un élément (alternative avec .filter)

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  return array.filter((el, currentI) => i !== currentI);
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Objet

- Ajouter un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function add(object, key, value) {
  return {
    ...object,
    [key]: value,
  };
}

const newContact = add(contact, 'city', 'Paris');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

Immuabilité - Objet

- Modifier un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function modify(object, key, value) {
  return {
    ...object,
    [key]: value,
  };
}

const newContact = modify(contact, 'firstName', 'Thomas');
console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

Immuabilité - Objet

- Supprimer un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function remove(object, key) {
  const { [key]: val, ...rest } = object;
  return rest;
}

const newContact = remove(contact, 'lastName');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

Immuabilité - Immutable.js

- Pour simplifier la manipulation d'objets ou de tableaux immuables, Facebook a créé Immutable.js
- Installation
  `npm install immutable`

Immuabilité - Immutable.js List

- Ajouter à la fin

```
const immutable = require('immutable');

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.push('Jean');
console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
const immutable = require('immutable');

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.unshift('Jean');
console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js List

- Ajouter à un indice donné

```
const immutable = require('immutable');

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.insert(1, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js List

- Modifier un élément

```
const immutable = require('immutable');

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.set(1, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js List

- Supprimer un élément

```
const immutable = require('immutable');

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.delete(1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js Map

- Ajouter un élément

```
const immutable = require('immutable');

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.set('city', 'Paris');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

Immuabilité - Immutable.js Map

- Modifier un élément

```
const immutable = require('immutable');

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.set('firstName', 'Thomas');
console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

Immuabilité - Immutable.js Map

- Supprimer un élément

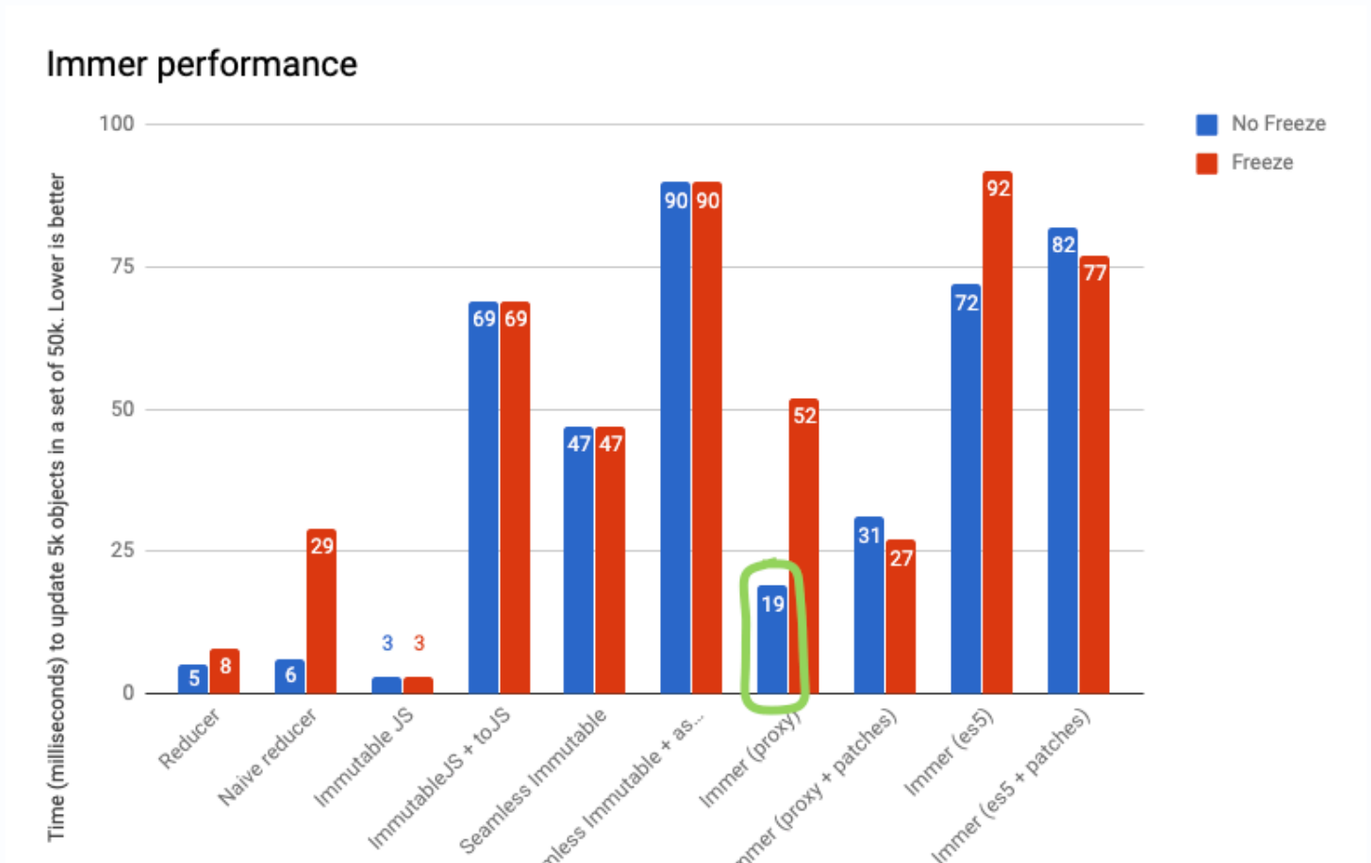
```
const immutable = require('immutable');

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.remove('lastName');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

Immuabilité - Immer

- Problème avec Immutable.js, il est nécessaire d'exécuter du code pour désérialiser du JSON
- Une autre approche avec Immer.js qui va "traduire" du code muable en code immuable
- Cela va avoir un impact sur les performances mais le code sera beaucoup plus lisible et simple à maintenir
- Bench pour mettre à jour 5000 objets dans un tableau de 50000 :



Immuabilité - Immer Tableaux

- Ajouter à la fin

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  draft.push('Jean');
});

console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  draft.unshift('Jean');
});

console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Tableaux

- Ajouter à un indice donné

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  const index = 1;
  draft.splice(index, 0, 'Jean')
});

console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Tableaux

- Modifier un élément

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  const index = 1;
  draft[index] = 'Jean';
});

console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Tableaux

- Supprimer un élément

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  const index = 1;
  draft.splice(index, 1);
});

console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Objets

- Ajouter un élément

```
import { produce } from 'immer';

const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

const newContact = produce(contact, (draft) => {
  draft.city = 'Paris';
});

console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

Immuabilité - Immer Objets

- Modifier un élément

```
import { produce } from 'immer';

const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

const newContact = produce(contact, (draft) => {
  draft.firstName = 'Thomas';
});

console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

Immuabilité - Immer Objets

- Supprimer un élément

```
import { produce } from 'immer';

const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

const newContact = produce(contact, (draft) => {
  delete draft.lastName
});

console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

Immuabilité - Immer Objets

- Immer est particulièrement utile lorsque qu'il y a des objets ou tableaux imbriqués
- Sans Immer :

```
const contacts = [
  { firstName: 'Romain', address: { city: 'Paris' } },
  { firstName: 'Edouard', address: { city: 'Lille' } },
  { firstName: 'Brice', address: { city: 'Nice' } },
];

const newContacts = [
  ...contacts.slice(0, 1),
  {
    ...contacts[1],
    address: {
      ...contacts[1].address,
      city: 'Bordeaux'
    }
  },
  ...contacts.slice(1 + 1),
]

console.log(JSON.stringify(newContacts));
// [
//   {"firstName":"Romain","address":{"city":"Paris"}},
//   {"firstName":"Edouard","address":{"city":"Bordeaux"}},
//   {"firstName":"Brice","address":{"city":"Nice"}}
// ]
console.log(contacts === newContacts); // false
```

Immuabilité - Immer Objets

- Immer est particulièrement utile lorsque qu'il y a des objets ou tableaux imbriqués
- Avec Immer :

```
import { produce } from 'immer';

const contacts = [
  { firstName: 'Romain', address: { city: 'Paris' } },
  { firstName: 'Edouard', address: { city: 'Lille' } },
  { firstName: 'Brice', address: { city: 'Nice' } },
];
```

```
const newContacts = produce(contacts, (draft) => {
  draft[1].address.city = 'Bordeaux';
});

console.log(JSON.stringify(newContacts));
// [
//   {"firstName":"Romain","address":{"city":"Paris"}},
//   {"firstName":"Edouard","address":{"city":"Bordeaux"}},
//   {"firstName":"Brice","address":{"city":"Nice"}}
// ]
console.log(contacts === newContacts); // false
```

formation.tech

Bonnes pratiques

Bonnes pratiques - Actions creators

- Pour faciliter la création d'actions et les pouvoir les réutiliser plus facilement à différents endroit de l'application, on utilise des fonctions appelées actions creators

```
export function incrementLikes() {
  return {
    type: 'INCREMENT_LIKES',
  };
}

export function updateName(newName) {
  return {
    type: 'UPDATE_NAME',
    payload: newName,
  };
}

store.dispatch(incrementLikes());
store.dispatch(updateName('Toto'));
```

Bonnes pratiques - Factoriser les types

- Pour ne pas avoir de typo entre le nom du type de l'actionCreator et du reducer on factorise les types, par exemple avec une constante :

```
export const INCREMENT_LIKES = 'INCREMENT_LIKES';
export const UPDATE_NAME = 'UPDATE_NAME';



export function reducer(state = initialState, action) {
  switch (action.type) {
    case INCREMENT_LIKES:
      return {
        ...state,
        likes: state.likes + 1,
      };
    case UPDATE_NAME:
      return {
        ...state,
        name: action.payload,
      };
      default:
      return state;
  }
}
```

```
export function incrementLikes() {
  return {
    type: INCREMENT_LIKES,
  };
}

export function updateName(newName) {
  return {
    type: UPDATE_NAME,
    payload: newName,
  };
}
```

Bonnes pratiques - Selectors

- Pour lire le state on passe par des fonctions appelées selectors
- Le code pourra ainsi être testé unitairement
- C'est d'autant plus important lorsque la lecture deviendra complexe par exemple lorsque les données à afficher seront dérivées du state

```
export function likesSelector(state) {
  return state.likes;
}

export function nameSelector(state) {
  return state.name;
}
```

Bonnes pratiques - Flux Standard Action

- Il existe une convention pour les actions : Flux Standard Action (FSA) : https://github.com/redux-utilities/flux-standard-action
- Le type se défini avec la clé *type* (obligatoire avec Redux)
- S'il y a une valeur, on utilise la clé *payload*

```
{
  type: 'ADD_TODO',
  payload: {
    text: 'Do something.'
  }
}
```

- Si l'action représente une erreur, la clé *payload* est de type Error ou dérivé et on ajoute la clé *error: true*

```
{
  type: 'FETCH_TODOS_FAILED',
  error: true,
  payload: new Error('500 Internal Server Error')
}
```

Bonnes pratiques - Flux Standard Action

- On peut ajouter une clé *meta* pour ajouter des données (par exemple l'id du composant dans lequel la donnée doit s'afficher, si plusieurs composants identiques sont présent à l'écran)

```
{
  type: 'DELETE_TODO',
  payload: {
    id: 4
  },
  meta: {
    todoListId: 2
  }
}
```

Bonnes pratiques - Reducers

- Les reducers sont des fonctions pures, ils peuvent cependant appeler d'autres fonctions pures pour décomposer la manipulation du state :

```
export function likesReducer(state = initialState.likes, action) {
  switch (action.type) {
    case INCREMENT_LIKES:
      return state + 1;
    default:
      return state;
  }
}
```

```
export function nameReducer(state = initialState.name, action) {
  switch (action.type) {
    case UPDATE_NAME:
      return action.payload;
    default:
      return state;
  }
}

export function reducer(state = initialState, action) {
  switch (action.type) {
    case INCREMENT_LIKES:
      return {
        ...state,
        likes: likesReducer(state.likes, action),
      };
    case UPDATE_NAME:
      return {
        ...state,
        name: nameReducer(state.name, action),
```

Bonnes pratiques - combineReducer

- Pour simplifier la création de reducers combinés, Redux exporte une fonction combineReducer :

```
export function likesReducer(state = initialState.likes, action) {
  switch (action.type) {
    case INCREMENT_LIKES:
      return state + 1;
    default:
      return state;
  }
}

export function nameReducer(state = initialState.name, action) {
  switch (action.type) {
    case UPDATE_NAME:
      return action.payload;
    default:
      return state;
  }
}

export const reducer = combineReducers({
  likes: likesReducer,
  name: nameReducer,
});
```

Bonnes pratiques - combineReducer

- Il est possible d'appeler combineReducer avec plusieurs niveaux :

```
export const reducer = combineReducers({
  likes: likesReducer,
  user: combineReducers({
    name: nameReducer,
    address: addressReducer,
  })
});
```

- Attention cependant, lorsqu'on appelle combineReducer, chaque reducer n'a accès qu'à un morceau du state, si des données venant d'une autre partie sont nécessaires il faudra les ajouter dans l'action

formation.tech

Redux Toolkit (RTK)

Redux Toolkit - Introduction

- Lorsque qu'on commence à mettre en place les bonnes pratiques, il y a beaucoup d'étapes pour connecter notre application à Redux :
  - définir la forme du state
  - définir les actions et créer les actions creators
  - créer les constantes
  - implémenter le ou les reducers
  - écrire les selectors
  - configurer le store et les middlewares
- Pour nous faire gagner du temps Redux propose une bibliothèque officielle appelée Redux Toolkit ou RTK
- Des bibliothèques plus anciennes non-officielles ont existé comme redux-actions

Redux Toolkit - Installation et mise en place

- Avec Redux Toolkit il n'est plus nécessaire d'installer Redux qui est une dépendance de RTK :
  `npm i @reduxjs/toolkit`

- Création du store avec configureStore :

```
const store = configureStore({
  reducer: reducer,
});
```

- configureStore appelle déjà combineReducers lorsqu'on passe un objet

```
const store = configureStore({
  reducer: {
    likes: likesReducer,
    name: nameReducer
  },
});
```

Redux Toolkit - Installation et mise en place

- Par défaut les devTools sont activés, devTools === true, on peut les désactiver selon une condition

```
const store = configureStore({
  reducer: reducer,
  devTools: process.env.NODE_ENV === 'development'
});
```

- ou les paramétrer de façon plus fine :

```
const store = configureStore({
  reducer: reducer,
  devTools: {
    maxAge: 10,
    features: {
      import: true,
      export: true,
    }
  }
});
```

Redux Toolkit - Installation et mise en place

- Pour précharger un state issu d'une précédente visite on utilise preloadedState

```
const store = configureStore({
  reducer: reducer,
  preloadedState: JSON.parse(localStorage.getItem('state'))
});
```

- On peut également configurer des stores enhancers et middlewares

Redux Toolkit - createAction

- createAction est un utilitaire pour créer des actions creators :

```
import { createAction } from '@reduxjs/toolkit';

export const incrementLikes = createAction('INCREMENT_LIKES');
export const updateName = createAction('UPDATE_NAME');

console.log(incrementLikes()); // { type: 'INCREMENT_LIKES' }
console.log(updateName('Romain')); // { type: 'UPDATE_NAME', payload: 'Romain' }
```

- Il permet également de se passer des constantes car l'action creator va factoriser le type :

```
import { createAction } from '@reduxjs/toolkit';

export const incrementLikes = createAction('INCREMENT_LIKES');
export const updateName = createAction('UPDATE_NAME');

console.log(incrementLikes.type); // INCREMENT_LIKES
console.log(updateName.type); // UPDATE_NAME
```

Redux Toolkit - createAction

- Lorsque l'action contient des valeurs générées ou par défaut, on peut simplifier la création côté composant en utilisant une fonction de préparation :

```
import { createAction, nanoid } from '@reduxjs/toolkit';

export const createUser = createAction('CREATE_USER', (username) => {
  return {
    payload: {
      id: nanoid(),
      username: username,
    },
  };
});

console.log(createUser('romain'));
// {
//   type: 'CREATE_USER',
//   payload: { id: 'xkXSsY2DSDSkuX4RvvRvu', username: 'romain' }
// }
```

Redux Toolkit - createReducer

- Pour simplifier la creation des reducers on utilise createReducer
- createReducer repose sur Immer, le code muable sera automatiquement traduit en code immuable

```
import { createReducer } from '@reduxjs/toolkit';
import { incrementLikes, updateName } from './actions.js';

const initialState = {
  name: 'Romain',
  likes: 10,
};

export const nameReducer = createReducer(initialState, (builder) => {
  builder
    .addCase(incrementLikes, (state, action) => {
      state.likes++;
    })
    .addCase(updateName, (state, action) => {
      state.name = action.payload;
    });
});
```

- addCase reçoit un type string ou un action creator en premier paramètre

Redux Toolkit - createReducer

- Si on ne souhaite pas utiliser Immer, il faut retourner le prochain state :

```
export const reducer = createReducer(initialState, (builder) => {
  builder
    .addCase(incrementLikes, (state, action) => {
      return { ...state, likes: state.likes + 1 }
    })
    .addCase(updateName, (state, action) => {
      return { ...state, name: action.payload };
    });
});
```

Redux Toolkit - createReducer

- On peut également utiliser un matcher pour traiter un ensemble d'action

```
import { createReducer, isAnyOf, isFulfilled, isPending, isRejected }
from '@reduxjs/toolkit';

const initialState = {
  requestsCount: 0,
};

export const reducer = createReducer(initialState, (builder) => {
  builder
    .addMatcher(isPending, (state) => {
      state.requestsCount++;
    })
    .addMatcher(isAnyOf(isFulfilled, isRejected), (state) => {
      state.requestsCount--;
    });
});
```

- Un matcher est une fonction qui reçoit une action en paramètre et retourne un booléen

```
function isPendingAction(action) {
  return typeof action.type === 'string' && action.type.endsWith('/pending')
}
```

Redux Toolkit - createSlice

- Pour aller encore plus vite, on peut combiner createAction et createReducer en un seul appel avec createSlice :

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
  name: 'Romain',
  likes: 10,
};

const likesSlice = createSlice({
  name: 'likes',
  initialState: initialState.likes,
  reducers: {
    incrementLikes(state, action) {
      return state + 1;
    },
  },
});

export const { incrementLikes } = likesSlice.actions;
export const reducer = {
  likes: likesSlice.reducer,
};
```

- name va préfixer les action types (ici likes/incrementLikes)
- reducers est à la fois la définition du reducer et le nom de l'action creator

Redux Toolkit - createSlice

- Si besoin d'un code plus pointu on peut écrire les reducers dans un style plus proche de createReducer avec extraReducers

```
export const createUser = createAction('createUser', (username) => {
  return {
    payload: {
      id: nanoid(),
      username: username,
    },
  };
});
```

```
const likesSlice = createSlice({
  name: 'users',
  initialState: initialState.users,
  reducers: {
    deleteUserById(state, action) {
      state.splice(action.payload, 1);
    },
  },
  extraReducers(builder) {
    builder.addCase(createUser, (state, action) => {
      return [...state, action.payload];
    });
  }
});
```

Redux Toolkit - combineSlices

- Pour combiner les reducers on peut également utiliser combineSlice :

```
export const reducer = combineSlices(nameSlice, likesSlice);
```

- Equivalent à :

```
export const reducer = {
  name: nameSlice.reducer,
  likes: likesSlice.reducer,
};
```

formation.tech

React Redux

React Redux - Mise en place

- Redux ayant été conçu comme une bibliothèque indépendante / framework agnostic, on utilise l'intégration officielle React Redux quand on souhaite l'utiliser avec React
- Installation : `npm i react-redux`
- Redux exporte un composant *Provider* qui permet de passer le store à l'application en utilisant le context :

```
import { configureStore } from '@reduxjs/toolkit';
import { Provider } from 'react-redux';
import { reducer } from './store/reducers';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(
  <Provider store={configureStore({ reducer: reducer })}>
    <App />
  </Provider>,
);
```

React Redux - Hooks API

- Une fois Provider utilisé à la racine de l'application, le plus simple pour interagir avec Redux est d'utiliser les Hooks de React Redux :
  - useSelector
  - useDispatch
  - useStore (rarement utilisé)
- Rappelons que les Hooks sont disponible depuis React 16.3 et uniquement dans des function components

React Redux - useSelector

- useSelector permet de récupérer une valeur dans le state en utilisant un selector :

```
import { useSelector } from 'react-redux';
import Hello from './Hello';
import { nameSelector } from './store/selectors';

function App() {
  const name = useSelector(nameSelector);

  return (
    <div className="App">
      <Hello name={name} />
    </div>
  );
}
```

- Le sélecteur est appelé avec l'ensemble du state
- Le sélecteur s'exécutera à chaque dispatch (voir reselect pour optimiser)
- Si sa valeur de retour est la même qu'au précédent appel, le composant n'est pas re-rendu (mémoisation)

React Redux - useSelector

- Par défaut useSelector utilise === pour tester l'égalité, si besoin on peut lui spécifier une autre fonction comme shallowEqual (2 objets ou tableaux avec le même contenu au premier niveau)

```
import { shallowEqual, useSelector } from 'react-redux';
import { usersSelector } from './store/selectors';

function App() {
  const users = useSelector(usersSelector, { equalityFn: shallowEqual });
  return (
    <div className="App">
      {users.map((user) => <div key={user.id}>{user.name}</div>)}
    </div>
  );
}
```

- En développement useSelector réalise des vérifications supplémentaires :
  - noopCheck : que le sélecteur ne retourne pas le state dans son ensemble
  - stabilityCheck: que 2 appels successifs (sans dispatch) au sélecteur retourne la même référence pour éviter de re-rendre le composant

React Redux - useDispatch

- Pour dispatcher une action on récupère la fonction dispatch avec useDispatch

```
import { useDispatch, useSelector } from 'react-redux';
import LikesButton from './LikesButton';
import { likesSelector } from './store/selectors';
import { incrementLikes } from './store/actions';

function App() {
  const likes = useSelector(likesSelector);
  const dispatch = useDispatch();

  function handleIncrement() {
    dispatch(incrementLikes());
  }

  return (
    <div className="App">
      <LikesButton likes={likes} onIncrement={handleIncrement} />
    </div>
  );
}
```

React Redux - connect

- Si on utilise des classes ou une version de React inférieure à 16.3 il faudra utiliser la fonction connect
- connect est une fonction qui retourne un Higher Order Component (une fonction qui reçoit un composant et retourne un composant qui l'encapsule)
- Le premier paramètre de connect est une fonction mapStateToProps qui retourne un objet dont les clés sont le noms des props du composant à encapsulé, créées à partir du state reçu en paramètres (via les selectors idéalement)

```
import { connect} from 'react-redux';
import Hello from './Hello';
import { nameSelector } from './store/selectors';

function mapStateToProps(state) {
  return { name: nameSelector(state) };
}

const HelloContainer = connect(mapStateToProps)(Hello);

function App() {
  return (
    <div className="App">
      <HelloContainer />
    </div>
  );
}
```

React Redux - connect

- Si on ne passe qu'un seul paramètre, connect va également transmettre au composant encapsulé une prop dispatch

```
function LikeButton({ likes, dispatch }) {
  function handleClick() {
    dispatch(incrementLikes());
  }

  return (
    <button className="LikesButton" onClick={handleClick}>
      {likes}
    </button>
  );
}

function mapStateToProps(state) {
  return {
    likes: likesSelector(state),
  };
}

const LikeButtonContainer = connect(mapStateToProps)(LikeButton);

function App() {
  return (
    <div className="App">
      <LikeButtonContainer />
    </div>
  );
}
```

React Redux - connect

- Pour rendre le composant indépendant de Redux et plus facile à réutiliser et tester, on préfèrera lui injecter une fonction qui appelera dispatch
- Cela est possible grace à une second callback passé à connect : *mapDispatchToProps*

```
function LikeButton({ likes, onIncrement }) {
  function handleClick() {
    onIncrement();
  }

  return <button className="LikesButton" onClick={handleClick}>{likes}</button>;
}

function mapStateToProps(state) {
  return {
    likes: likesSelector(state),
  };
}

function mapDispatchToProps(dispatch) {
  return {
    onIncrement: () => dispatch(incrementLikes()),
  };
}

const LikeButtonContainer = connect(mapStateToProps, mapDispatchToProps)(LikeButton);
```

formation.tech

Ecosystème

Ecosystème - Introduction

- Une des forces de Redux est son écosystème, de nombreuses bibliothèques gravitent autour de Redux pour faciliter le développement
- On retrouve :
  - des intégrations avec des frameworks tiers (Angular, Ember...)
  - des outils pour faciliter le développement, le debug ou les tests
  - des middlewares qui modifient le comportement de Redux
  - des Higher Order Reducers pour mettre en place la persistence et le undo/redo
  - des intégrations avec des concepts tiers : formulaires, router...
- On retrouve des liens vers des libs/projets de l'écosystème :
  - https://redux.js.org/introduction/ecosystem
  - https://redux.js.org/usage/deriving-data-selectors#alternative-selector-libraries
  - https://redux.js.org/usage/structuring-reducers/normalizing-state-shape#normalizing-nested-data

Ecosystème - Redux DevTools

- Redux DevTools est l'évolution de l'extension créée par Dan Abramov qui a donné naissance à Redux
- Installation :
  - Chrome : https://chromewebstore.google.com/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
  - Firefox : https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/
  - Edge : https://microsoftedge.microsoft.com/addons/detail/redux-devtools/nnkgneoiohoecpdiaponcejilbhhikei
  - Standalone app : https://github.com/reduxjs/redux-devtools/tree/main/packages/redux-devtools-app
  - React component : https://github.com/reduxjs/redux-devtools/tree/main/packages/redux-devtools

Ecosystème - Redux DevTools

- Configuration :
  - Avec Redux Toolkit
    Les devtools sont configurés par défaut, on peut les désactiver dans certains environnement :

```
const store = configureStore({
  reducer: reducer,
  devTools: process.env.NODE_ENV === 'development'
});
```

  - Ou bien les paramétrer plus finement

```
const store = configureStore({
  reducer: reducer,
  devTools: {
    maxAge: 10,
    features: {
      import: true,
      export: true,
    }
  }
});
```

Ecosystème - Redux DevTools

- Configuration :
  - Via les variables globales exposée par l'extension :

```
import { reducer } from './store/reducers';
import { legacy_createStore } from 'redux';

const store = legacy_createStore(
  reducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
);
```

  - De façon plus fine si besoin de passer d'autres middlewares :

```
const composeEnhancers =
  typeof window === 'object' && window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__
    ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__({
        // Specify extension’s options like name, actionsDenylist,
actionsCreators, serialize...
      })
    : compose;

const enhancer = composeEnhancers(
  applyMiddleware(...middlewares),
  // other store enhancers if any
);

const store = legacy_createStore(reducer, enhancer);
```

Ecosystème - Redux DevTools

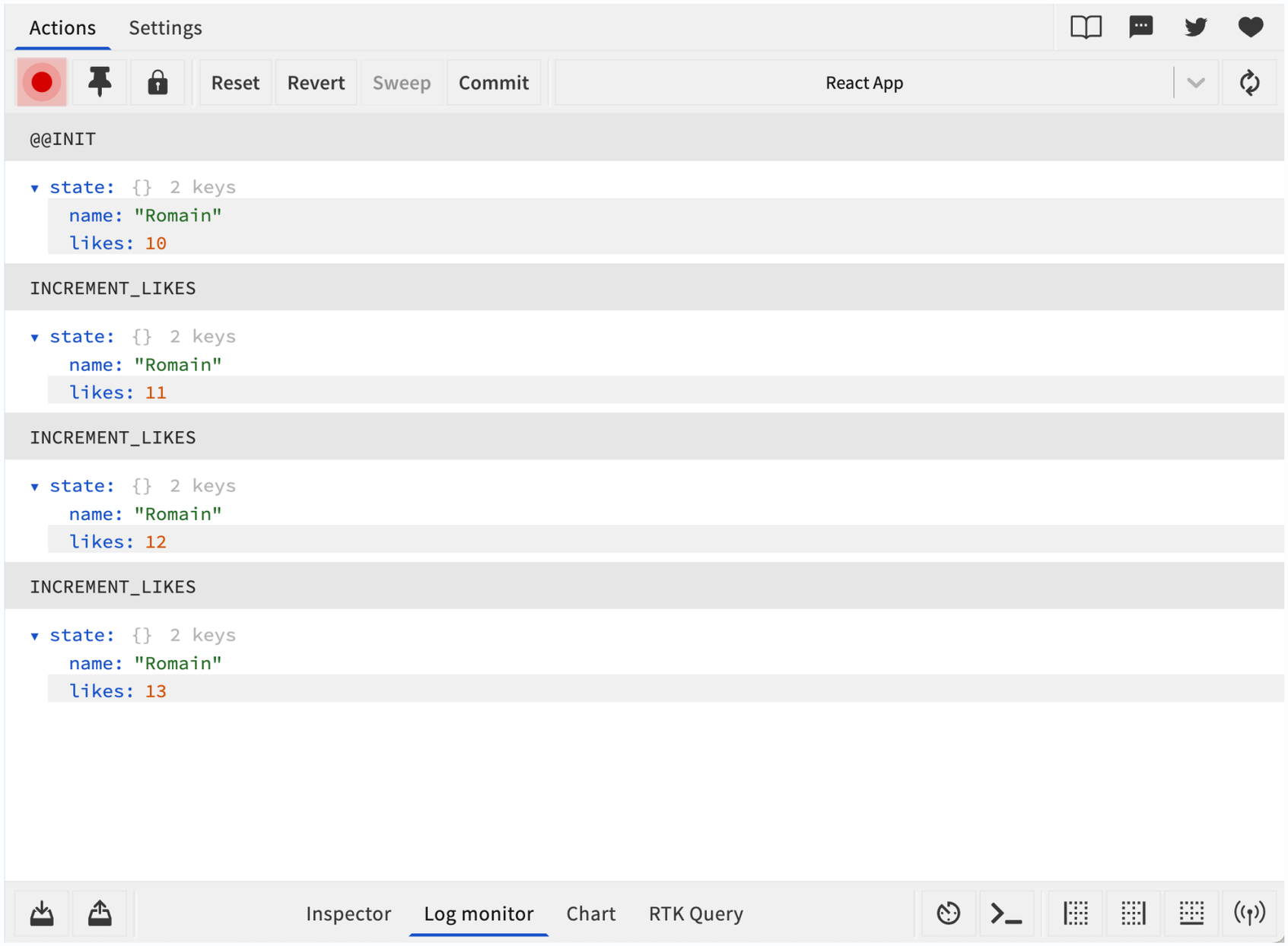
- Configuration :
  - Via un paquet npm : `npm i @redux-devtools/extension`

```
import { applyMiddleware, legacy_createStore } from 'redux';
import { composeWithDevTools } from '@redux-devtools/extension';

const composeEnhancers = composeWithDevTools({
  // Specify name here, actionsDenylist, actionsCreators and other options if
needed
});
const store = legacy_createStore(
  reducer,
  /* preloadedState, */ composeEnhancers(
    applyMiddleware(...middlewares),
    // other store enhancers if any
  ),
);
```

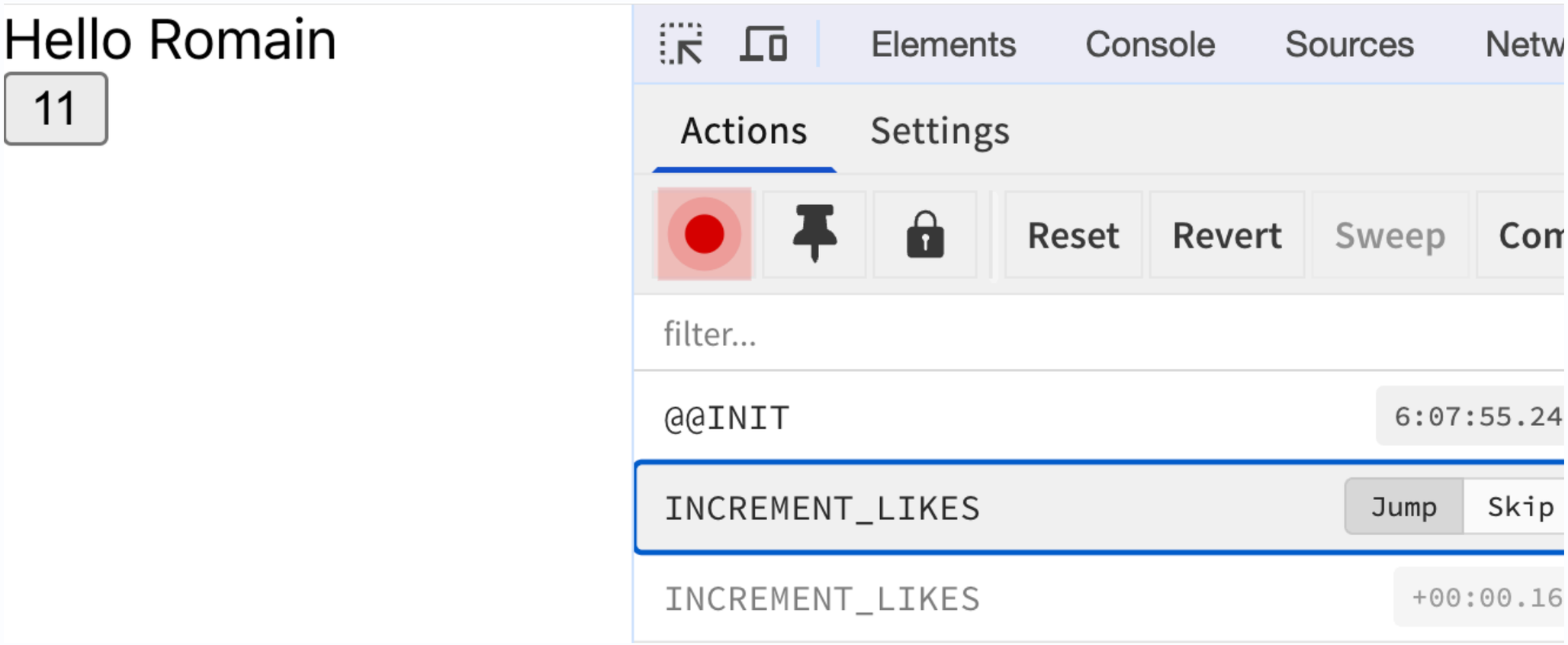
Ecosystème - Redux DevTools

- Fonctionnalités
  - Log Monitor



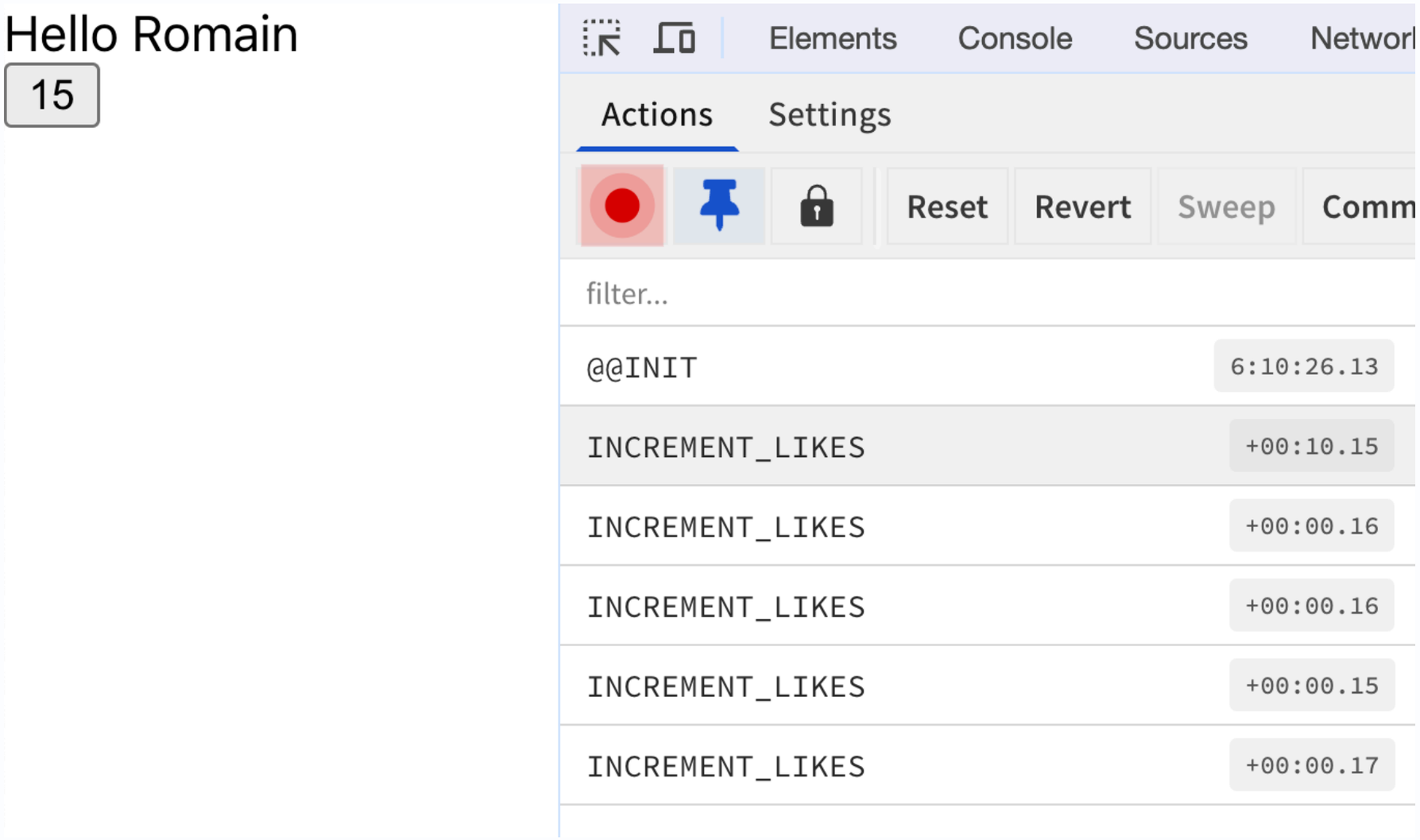
Ecosystème - Redux DevTools

- Fonctionnalités
  - Jump : revenir à une action passée
  - Skip : ignorer une action



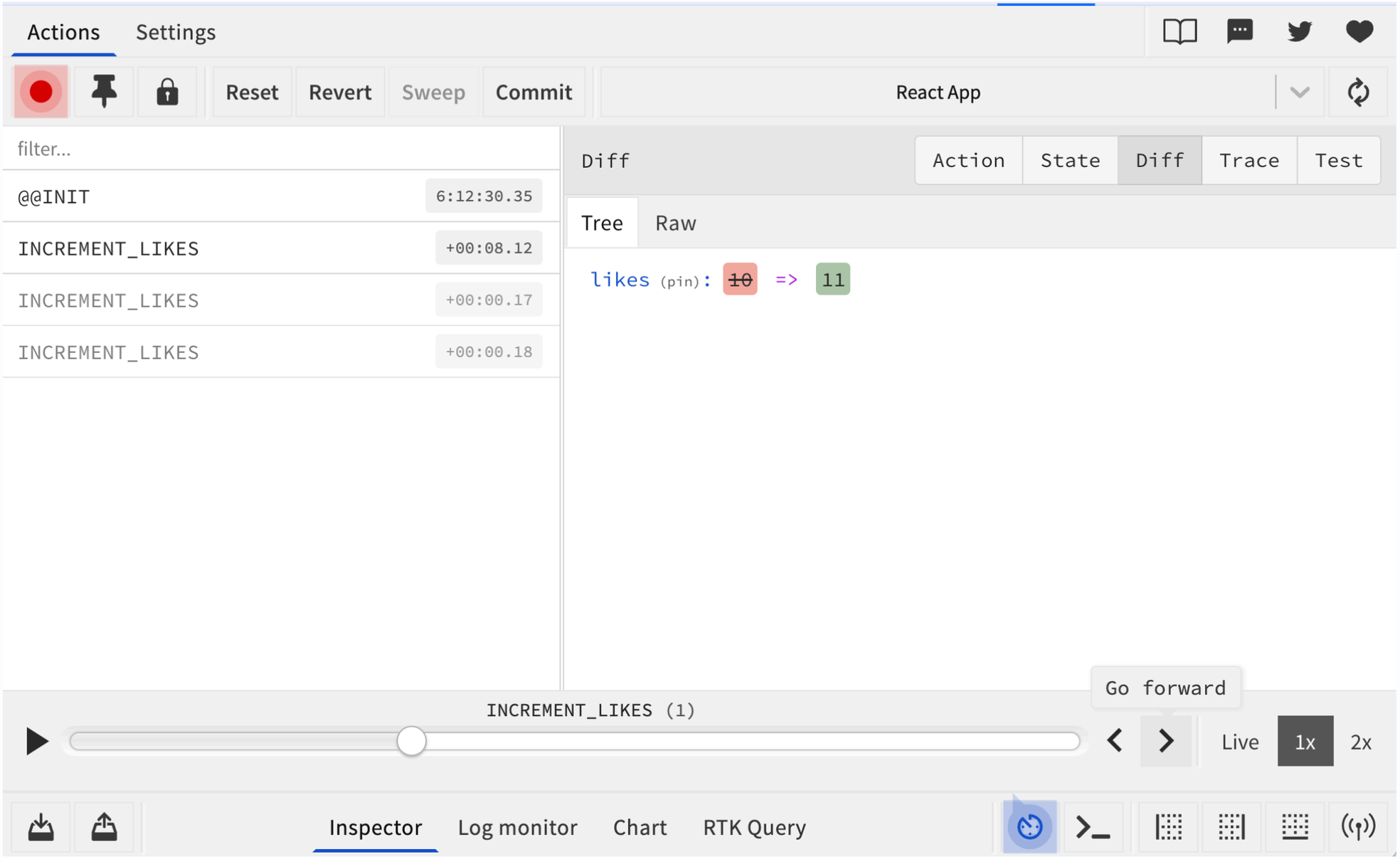
Ecosystème - Redux DevTools

- Fonctionnalités
  - Persist : rejouer les actions après rechargement de la page



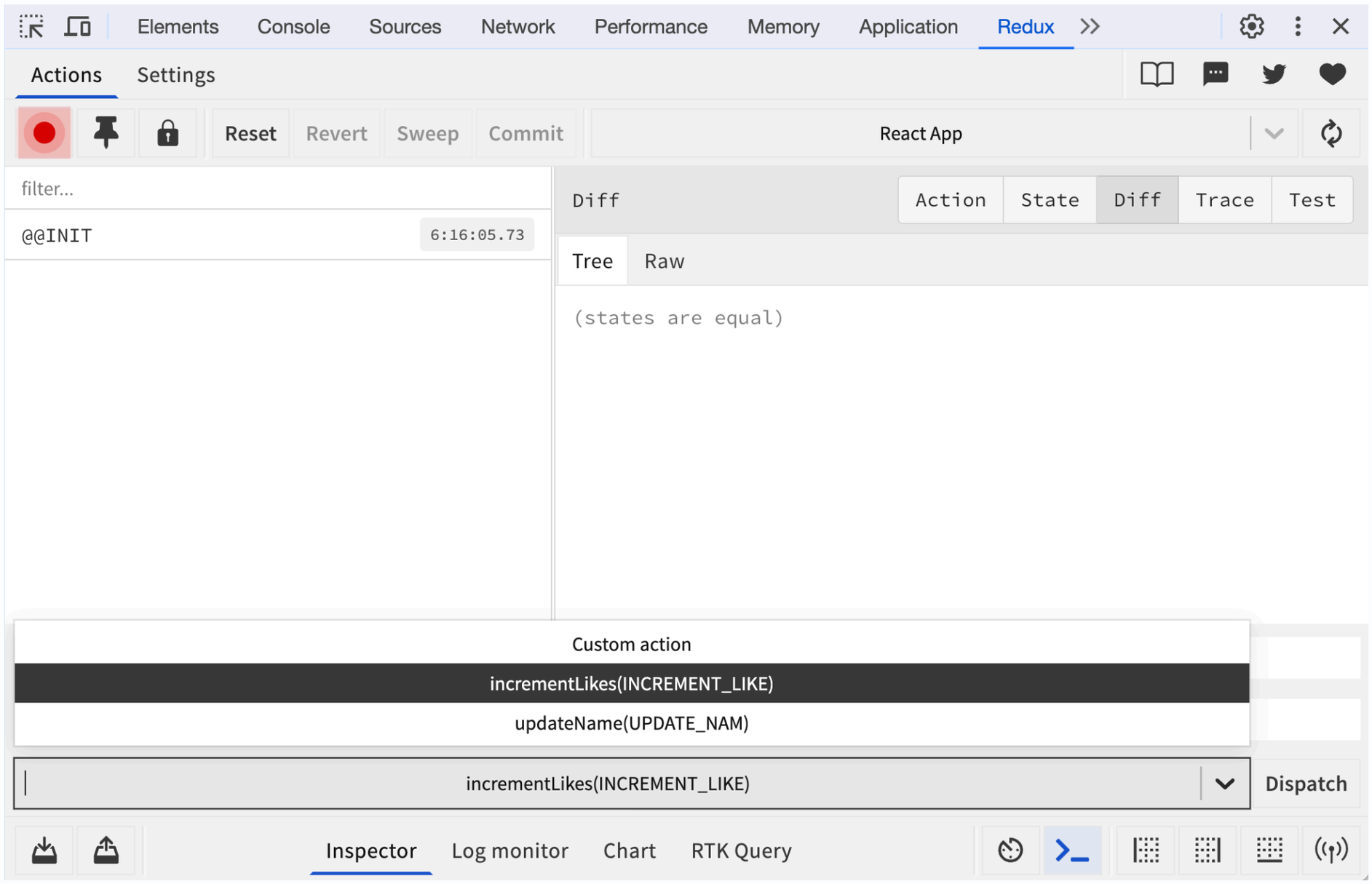
Ecosystème - Redux DevTools

- Fonctionnalités
  - Slider : avancer / reculer / rejouer les enchainements d'actions



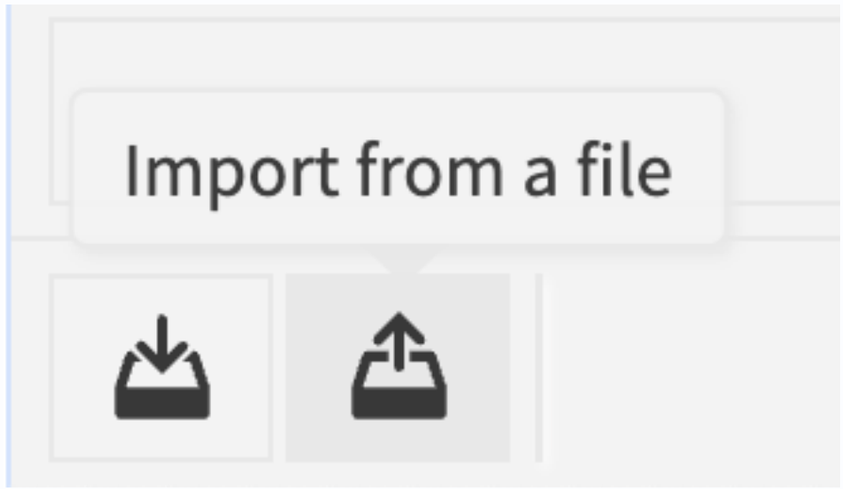
Ecosystème - Redux DevTools

- Fonctionnalités
  - Dispatcher : saisir voir sélectionner ses actions même si l'interface n'existe pas encore



Ecosystème - Redux DevTools

- Fonctionnalités
  - Exporter / Importer : pour rejouer des actions sur un autre machine (en les partageant via un ticket Github par exemple)



Ecosystème - Redux Thunk

- Redux Thunk permet de passer une fonction à dispatch plutôt qu'une action
- Cette fonction aura accès à dispatch et au state et pourra donc dispatcher de nouvelle action
- C'est particulièrement utile pour les enchainement asynchrone :
  - afficher une notification qui doit disparaitre au bout de 5 secondes
  - envoyer une requête HTTP qui doit afficher un loader pendant la requête puis les données ou une erreur lors de la réponse

```
export function showNotification(payload) {
  return function (dispatch, getState) {
    dispatch({ type: 'SHOW_NOTIFICATION', payload });
    setTimeout(() => {
      dispatch({ type: 'HIDE_NOTIFICATION', payload });
    }, 5000);
  };
}
```

```
dispatch(showNotification('My message'));
```

Ecosystème - Redux Thunk

- Installation : npm i redux-thunk puis activation avec applyMiddleware à la création du store
- Avec Redux Toolkit :
  - ce middleware est déjà installé par défaut
  - une fonction createAsyncThunk simplifie la création d'actions basées sur redux-thunk, 3 actions seront créées dont les types seront suffixés par /pending, /fulfilled et /rejected
  - cette fonction contient 3 propriété .pending, .fulfilled et .rejected pour être traités par createReducer ou extraReducer de createSlice

Ecosystème - Redux Thunk

```
const state = {
  todos: {
    items: [],
    loading: false,
    errorMessage: '',
  },
};

export const fetchTodos = createAsyncThunk('fetchTodos', async () => {
  const res = await fetch('https://jsonplaceholder.typicode.com/todos');
  const data = await res.json();
  return data;
});

export const reducer = createReducer(state.todos, (builder) => {
  builder
    .addCase(fetchTodos.pending, (state, action) => {
      state.loading = true;
    })
    .addCase(fetchTodos.fulfilled, (state, action) => {
      state.loading = false;
      state.errorMessage = '';
      state.items = action.payload;
    })
    .addCase(fetchTodos.rejected, (state, action) => {
      state.loading = false;
      state.errorMessage = action.payload;
      state.items = [];
    });
});
```

Ecosystème - Redux Thunk vs Redux Saga

- Redux Thunk
  https://stackoverflow.com/questions/35411423/how-to-dispatch-a-redux-action-with-a-timeout/35415559#35415559
- Redux Saga
  https://stackoverflow.com/questions/35411423/how-to-dispatch-a-redux-action-with-a-timeout/35415559#38574266

Ecosystème - RTK Default Middleware

- En plus de Redux Thunk, Redux Toolkit intègre également 3 autres middlewares par défaut, en développement uniquement :

  - Immutability check middleware
    Vérifie que le reducer modifie le state de façon immuable (pas d'inquiétude à avoir avec RTK qui intègre Immer.js)

  - Serializability check middleware
    Vérifie que le state soit bien sérialisable (ne contienne pas de fonction...)

  - Action creator check middleware
    Vérifie que l'action creator est bien appelée au moment du dispatch (il ne manque pas les parenthèses de l'appel)

Ecosystème - Custom Middleware

- Un middleware custom

```
import { Middleware } from '@reduxjs/toolkit';

const logger: Middleware =
  ({ getState, dispatch }) =>
  (next) =>
  (action) => {
    console.log('will dispatch', action);

    // Call the next dispatch method in the middleware chain.
    const returnValue = next(action);

    console.log('state after dispatch', getState());

    // This will likely be the action itself, unless
    // a middleware further in chain changed it.
    return returnValue;
  };

export default logger;
```

Ecosystème - Custom Middleware

- Le type middleware de Redux a été modifié entre Redux 4 et 5 ce qui nécessite parfois forcer l'utilisation de Redux 5

- npm (uniquement pour redux-saga)

```
"overrides": {
  "redux-saga": {
    "redux": "^5.0.0"
  }
}
```

- Yarn (uniquement pou redux-saga)

```
{
  "resolutions": {
    "redux-saga/redux": "^5.0.0"
  }
}
```

- npm (global)

```
"overrides": {
  "redux": "^5.0.0"
}
```

- Yarn (global)

```
{
  "resolutions": {
    "redux": "^5.0.0"
  }
}
```

Ecosystème - createListeningMiddleware

- createListeningMiddleware fait partie de Redux Toolkit, il permet de gérer des effets comme Redux Saga, tout en proposant un API plus simple et plus performant
- Pour les cas les plus complexes, le code sera plus simple à maintenir avec Redux Saga ou Redux Observable
- A la manière de useEffect, createListeningMiddleware permet de jouer des effets lors de modifications apportées au store

```
const listenerMiddleware = createListenerMiddleware();

listenerMiddleware.startListening({
  matcher: isAction,
  effect: async (action) => {
    await sendToAnalytics(action);
  },
});

const store = configureStore({
  reducer: reducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().prepend(listenerMiddleware.middleware),
});
```

Ecosystème - createListeningMiddleware

- En plus de l'effet, startListening permet d'écouter :
  - un type (string)
  - un action creator créé avec RTK
  - un matcher (une fonction appelée avec l'action)
  - un predicate (une fonction appelée avec l'action, le state actuel, le state précédent)

```
// 1) Action type string
listenerMiddleware.startListening({ type: 'todos/todoAdded', effect })
// 2) RTK action creator
listenerMiddleware.startListening({ actionCreator: todoAdded, effect })
// 3) RTK matcher function
listenerMiddleware.startListening({
  matcher: isAnyOf(todoAdded, todoToggled),
  effect,
})
// 4) Listener predicate
listenerMiddleware.startListening({
  predicate: (action, currentState, previousState) => {
    // return true when the listener should run
  },
  effect,
})
```

Ecosystème - Redux Saga

- Redux Saga propose un API plus pointu que createListeningMiddleware en se basant sur les générateurs

```ts
import createSagaMiddleware from 'redux-saga';
import { call, takeEvery, throttle } from 'redux-saga/effects';

async function sendToAnalytics(actions: UnknownAction[]) {
  console.log(actions);
}

function* analyticsSaga() {
  const actions: UnknownAction[] = [];
  yield takeEvery('*', function (action: UnknownAction) {
    actions.push(action);
  });
  yield throttle(1000, '*', function* () {
    yield call(sendToAnalytics, actions);
    actions.splice(0);
  });
}

const sagaMiddleware = createSagaMiddleware();

const store = configureStore({
  reducer: reducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(sagaMiddleware),
});

sagaMiddleware.run(analyticsSaga);
```

```json
"overrides": {
  "redux-saga": {
    "redux": "^5.0.0"
  }
}
```

Ecosystème - Redux Observable

- Redux Observable est également une alternative à condition de maitrise la bibliothèque RxJS

```
import { Epic, createEpicMiddleware } from 'redux-observable';
import { EMPTY, bufferTime, filter, mergeMap, tap } from 'rxjs';

async function sendToAnalytics(actions: UnknownAction[]) {
  console.log(actions);
}

const analyticsEpic: Epic<UnknownAction> = (action$) =>
  action$.pipe(
    bufferTime(1000),
    filter((actions) => actions.length > 0),
    tap(async (actions) => {
      await sendToAnalytics(actions);
    }),
    mergeMap(() => EMPTY),
  );

const epicMiddleware = createEpicMiddleware<UnknownAction>();

const store = configureStore({
  reducer: reducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(epicMiddleware),
});

epicMiddleware.run(analyticsEpic);
```

Ecosystème - Reselect

- Reselect, intégré par défaut dans RTK permet de mémoiser les sélecteurs
- La mémoisation est un technique d'optimisation qui permet de ne pas rappeler une fonction si les paramètres n'ont pas changé depuis le dernier appel
- Ne fonctionne qu'avec des fonctions pures

```
const { memoize } = require('lodash');

function findLowerCount(arrayNbs, val) {
  return arrayNbs.filter((el) => el < val).length;
}

// Avec memoisation
const findLowerCountMemo = memoize(findLowerCount);
console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5));
console.timeEnd('findLowerCountMemo'); // 71.366ms

console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5)); // pas de rappel
console.timeEnd('findLowerCountMemo'); // 0.102ms

console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5)); // pas de rappel
console.timeEnd('findLowerCountMemo'); // 0.045ms
```

Ecosystème - Reselect

- Exemple

```
const memoizedSelectCompletedTodos = createSelector(
  [(state: RootState) => state.todos],
  (todos) => {
    console.log('memoized selector ran')
    return todos.filter(todo => todo.completed === true)
  }
)

memoizedSelectCompletedTodos(state) // memoized selector ran
memoizedSelectCompletedTodos(state)
```

- Le premier paramètre est un tableau de sélecteurs dont chaque retour sera passé en paramètre d'entrée du sélecteur mémoisée

Ecosystème - Redux Persist

- Permet de faire persister tout ou partie du store

```
import { configureStore } from '@reduxjs/toolkit';
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage'; // defaults to localStorage for web
import { PersistGate } from 'redux-persist/integration/react';

import reducer from './store/reducers';

const persistedReducer = persistReducer(
  {
    key: 'root',
    storage,
    throttle: 1000,
  },
  reducer,
);
const store = configureStore({
  reducer: persistedReducer,
});
const persistor = persistStore(store);

function App() {
  return (
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <RootComponent />
      </PersistGate>
    </Provider>
  );
}
```

Ecosystème - Redux Undo

- Documentation sur comment implémenter le undo/redo : https://redux.js.org/usage/implementing-undo-history

```
const undoableReducer = undoable(reducer)
```

- Avec ce type de reducer le state va ressembler à :

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: {
    past: [
      [],
      [{ text: 'Use Redux' }],
      [{ text: 'Use Redux', complete: true }]
    ],
    present: [
      { text: 'Use Redux', complete: true },
      { text: 'Implement Undo' }
    ],
    future: [
      [
        { text: 'Use Redux', complete: true },
        { text: 'Implement Undo', complete: true }
      ]
    ]
  }
}
```

Ecosystème - Redux Undo

- Il faut mettre à jour les sélecteurs pour qu'ils récupèrent la clé *present*
- *Pour exécuter les undo/redo on utilise des actions creators*

```
import { ActionCreators as UndoActionCreators } from 'redux-undo';

UndoActionCreators.undo();
UndoActionCreators.redo();
```

Ecosystème - Normaliser

- Pour accélérer la recherche dans le state et s'en servir comme d'une base de données on peut normaliser les listes de la façon suivante :

```
{
  users: {
    ids: ["user1", "user2", "user3"],
    entities: {
      "user1": {id: "user1", firstName: "user1", lastName: "user1"},
      "user2": {id: "user2", firstName: "user2", lastName: "user2"},
      "user3": {id: "user3", firstName: "user3", lastName: "user3"},
    }
  }
}
```

- Historiquement on utilisait la lib normalizr (plus maintenue) : https://github.com/paularmstrong/normalizr
- Avec RTK on peut utiliser createEntityAdapter

Ecosystème - createEntityAdapter

- createEntityAdapter s'inspire de NgRx/entities

```ts
export const todosAdapter = createEntityAdapter<Todo>({
  sortComparer: (a, b) => a.title.localeCompare(b.title),
});

const initialState: RootState = {
  todos: {
    newTodo: 'XYZ',
    items: todosAdapter.getInitialState(),
  },
};
```

```ts
export const todosSlice = createSlice({
  initialState: initialState.todos,
  name: 'todos',
  reducers: {
    addTodo(state, action: PayloadAction<Todo>) {
      todosAdapter.addOne(state.items, action.payload);
    },
    updateNewTodo(state, action: PayloadAction<string>) {
      state.newTodo = action.payload;
    },
  }
});
```

Ecosystème - createEntityAdapter

- L'adaptateur expose ses propres sélecteurs

```
const { selectAll } = todosAdapter.getSelectors();

export function itemsSelector(state: RootState): Todo[] {
  return selectAll(state.todos.items);
}
```

Ecosystème - RTK Query

- RTK Query est un framework qui repose sur Redux
- Inspiré de React Query, SWR, Apollo...

```
export const usersApi = createApi({
  reducerPath: 'usersApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://jsonplaceholder.typicode.com',
  }),
  endpoints: (builder) => ({
    getUsers: builder.query<User[], void>({
      query: () => `/users`,
    }),
    getUserById: builder.query<User, number>({
      query: (id) => `/users/${id}`,
    }),
    postUser: builder.mutation<User, User>({
      query(user) {
        return {
          url: `/users`,
          method: 'POST',
          body: JSON.stringify(user),
        };
      },
    }),
  }),
});
```

Ecosystème - RTK Query

- createApi "hérite" de createSlice on peut l'utiliser avec combineSlices

```
export const reducer = combineSlices(nameSlice, likesSlice, usersApi);
```

- Au moment de configurer le store il faudra enregistrer un middleware :

```
configureStore({ reducer: reducer, middleware: (getDefaultMiddleware) =>
getDefaultMiddleware().concat(usersApi.middleware) })
```

- Si on importe l'intégration avec React on peut directement générer des hook

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';

// ...

export const { useGetUsersQuery } = usersApi;
```

Ecosystème - RTK Query

- Coté composant on pourra appeler les méthodes au travers de ces hooks

```
import { useGetUsersQuery } from "./store/slices";

function Users() {
  const { data, isLoading } = useGetUsersQuery();

  if (isLoading) {
    return <p>Loading...</p>
  }

  return (
    <ul>
      {data?.map((user) => <li key={user.id}>{user.name}</li>)}
    </ul>
  )
}

export default Users;
```