




REDUX

"predictable state container for JavaScript apps"

 Page supprimée ! 

 Ajouter des labels

SUMMARY


Core Concepts


The Selector pattern

Normalizing State Shape

Middleware

Side Effects



 Page supprimée ! ...




WHY ?

From scripting to SPA


- manage more state than ever
- lost control over the when, why, and how state is updated
- mutation and asynchronicity


 Page supprimée ! 

 Ajouter des labels



Redux attempts to make state mutations predictable by imposing certain restrictions on how and when updates can happen


- *Dan Abramov - co-creator of Redux*

 Page supprimée ! ...



CORE CONCEPTS

 Page supprimée ! 

 Ajouter des labels

SINGLE SOURCE OF TRUTH

The state of your whole application is stored in an object tree within a single store.

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

STATE IS READ-ONLY

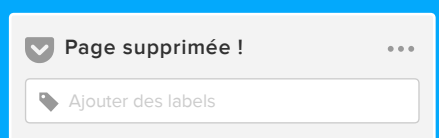
The only way to change the state is to emit an action, an object describing what happened.

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})



store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```


CHANGES ARE MADE WITH PURE FUNCTIONS

To specify how the state tree is transformed by actions,
you write pure reducers.

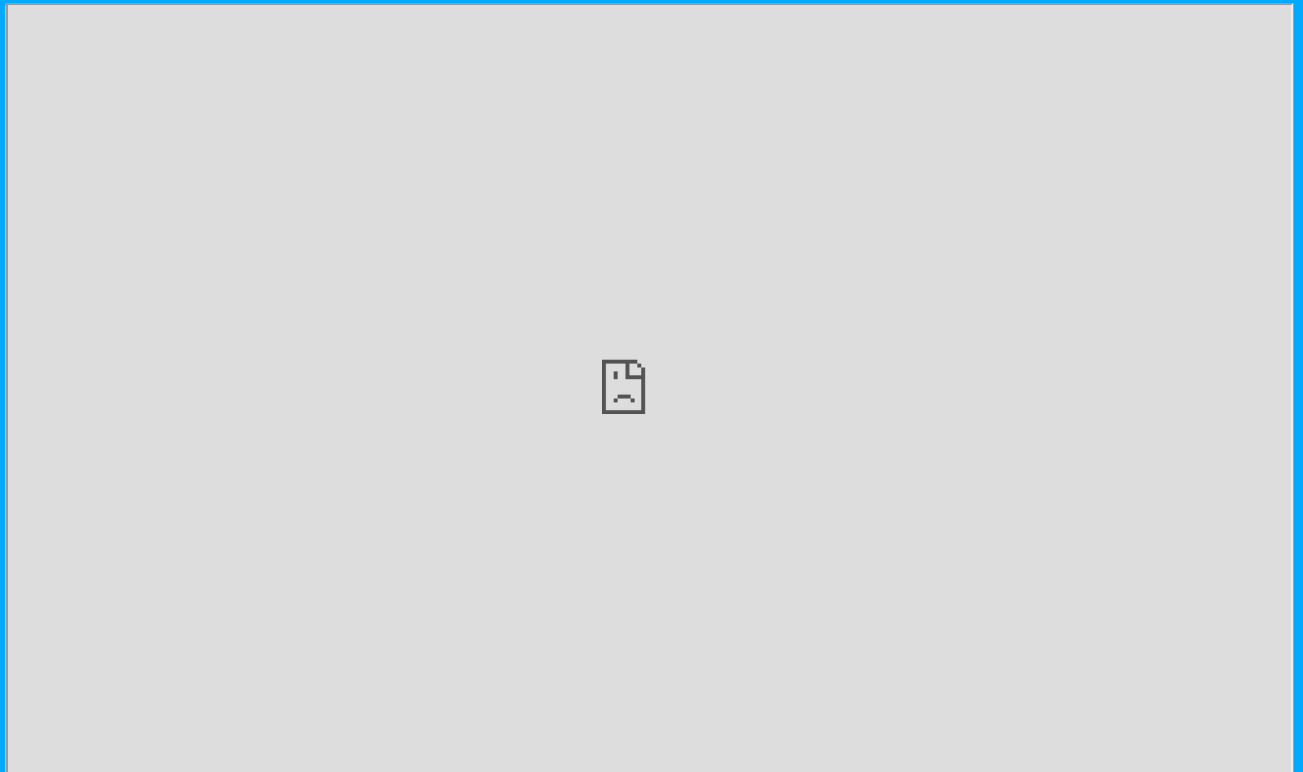




Reducers


 Page supprimée ! 

 Ajouter des labels

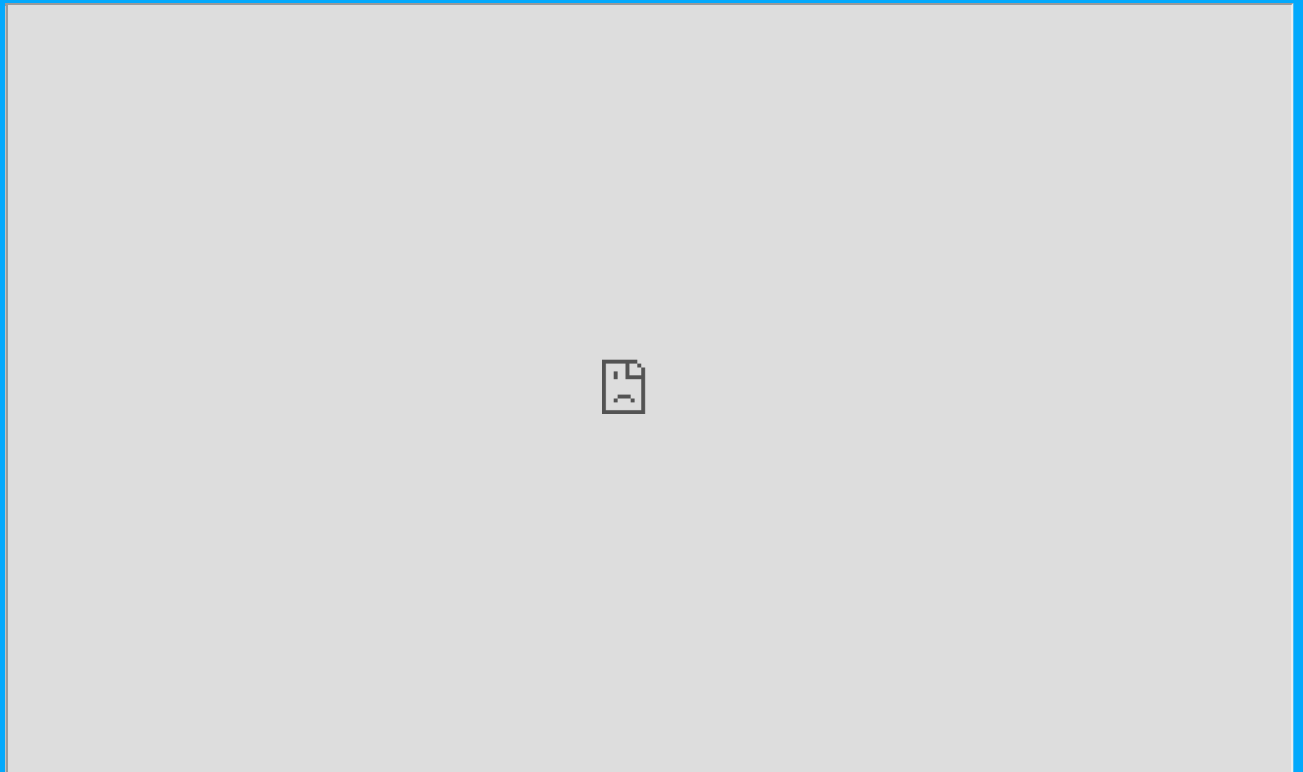
BASIC EXAMPLE






 **Page supprimée !** 

 Ajouter des labels


REAL WORD EXAMPLE




 **Page supprimée !** 



THE SELECTOR PATTERN

 Page supprimée ! ...



Without selector

```
1. function listOfItems(state = [], action = {}){
2.   switch(action.type) {
3.     case 'SHOW_ALL_ITEMS':
4.       return action.data.items
5.     default:
6.       return state;
7.   }
8. }
9.
10. const item = {
11.   id: string,
12.   text: string,
13.   completed: boolean
14. };
```

✓ Page supprimée !

...

🏷 Ajouter des labels



THE PROBLEM WITH THIS APPROACH


- The implementation of *incompleteItems* may change.
 - Computation logic occurs in *mapStateToProps*
 - Can't memoize the values of *incompleteItems*

THE SOLUTION

Selectors

- Selectors can compute derived data, allowing Redux to store the minimal possible state.
- Selectors are composable. They can be used as input to other selectors.
- Selectors are your “reading API” and should be co-located with their reducers

 Page supprimée ! 

 Ajouter des labels



With selector


```
1. function listOfItems(state = [], action = {}){
2.   switch(action.type) {
3.     case 'SHOW_ALL_ITEMS':
4.       return action.data.items
5.     default:
6.       return state;
7.   }
8. }
9.
10. function getIncompleteItems(state){
11.   return state.listOfItems.filter((item) => {
12.     return !item.completed
13.   });
14. }
```


EXERCICE

Writing a selector

[starwars-redux/tree/tp1](#)


 Page supprimée ! 


 Ajouter des labels

RESELECT

Selector library for Redux

- Selectors can compute derived data, allowing Redux to store the minimal possible state
- Selectors are efficient. A selector is not recomputed unless one of its arguments change
- Selectors are composable. They can be used as input to other selectors

 Page supprimée ! ...

 Ajouter des labels

Without Reselect

```
1. import { connect } from 'react-redux'
2. import { toggleTodo } from '../actions'
3. import TodoList from '../components/TodoList'
4.
5. const getVisibleTodos = (todos, filter) => {
6.   switch (filter) {
7.     case 'SHOW_ALL':
8.       return todos
9.     case 'SHOW_COMPLETED':
10.      return todos.filter(t => t.completed)
11.     case 'SHOW_ACTIVE':
12.      return todos.filter(t => !t.completed)
13.   }
14. }
15.
16. const mapStateToProps = (state) => {
17.   return {
18.     todos: getVisibleTodos(state.todos, state.visibilityFilter)
19.   }
20. }
21.
```

✓ Page supprimée !



🏷 Ajouter des labels



With Reselect


```
1. import { createSelector } from 'reselect'
2.
3. const getVisibilityFilter = (state) => state.visibilityFilter
4. const getTodos = (state) => state.todos
5.
6. export const getVisibleTodos = createSelector(
7.   [ getVisibilityFilter, getTodos ],
8.   (visibilityFilter, todos) => {
9.     switch (visibilityFilter) {
10.      case 'SHOW_ALL':
11.        return todos
12.      case 'SHOW_COMPLETED':
13.        return todos.filter(t => t.completed)
14.      case 'SHOW_ACTIVE':
15.        return todos.filter(t => !t.completed)
16.    }
17.  }
18. )
19.
```

Page supprimée !

Ajouter des labels

NORMALIZING STATE SHAPE


 Page supprimée ! 


 Ajouter des labels

```
1. const blogPosts = [  
2.   {  
3.     id : "post1",  
4.     author : {username : "user1", name : "User 1"},  
5.     body : ".....",  
6.     comments : [  
7.       {  
8.         id : "comment1",  
9.         author : {username : "user2", name : "User 2"},  
10.        comment : ".....",  
11.      },  
12.      {  
13.        id : "comment2",  
14.        author : {username : "user3", name : "User 3"},  
15.        comment : ".....",  
16.      }  
17.    ]  
18.  },  
19.  {  
20.    id : "post2",  
21.    author : {username : "user2", name : "User 2"},  
22.    body : ".....",  
23.    comments : [  
24.      {  
25.        id : "comment3",  
26.        author : {username : "user3", name : "User 3"},  
27.        comment : ".....",  
28.      },  
29.      {  
30.        id : "comment4",  
31.        author : {username : "user1", name : "User 1"},  
32.        comment : ".....",  
33.      },  
34.      {  
35.        id : "comment5",  
36.        author : {username : "user3", name : "User 3"},  
37.        comment : ".....",  
38.      }  
39.    ]  
40.  }  
41. ] // and repeat many times
```

THE PROBLEM WITH THIS APPROACH



- When a piece of data is duplicated in several places, it becomes harder to make sure that it is updated appropriately
- Nested data means that the corresponding reducer logic has to be more nested or more complex
- Not compliant with immutable data updates that require all ancestors in the state tree to be copied and updated as well


 Page supprimée ! ...

 Ajouter des labels

The recommended approach to managing relational or nested data in a Redux store is to treat a portion of your store as if it were a database, and keep that data in a normalized form.

- Dan Abramov - co-creator of Redux



 Page supprimée ! 


 Ajouter des labels

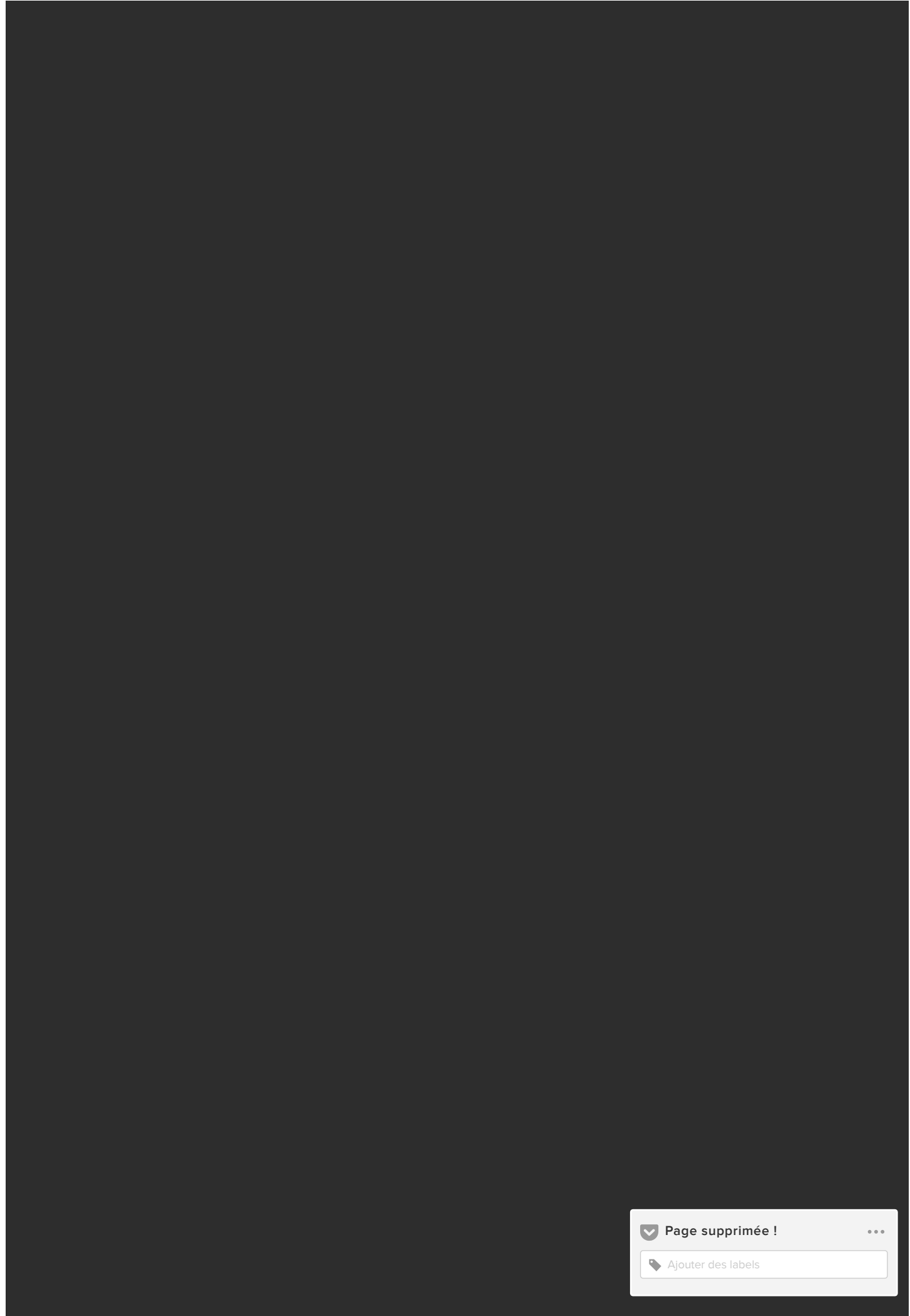
THE SOLUTION



Normalizing data


- Each type of data gets its own "table" in the state
- Each "data table" should store the individual items in an object, with the IDs of the items as keys and the items themselves as the values
- Any references to individual items should be done by storing the item's ID
- Arrays of IDs should be used to indicate ordering.

 Page supprimée ! 

 Ajouter des labels





 **Page supprimée !** 



BENEFITS

- Because each item is only defined in one place, we don't have to make changes in multiple places if that item is updated
- The reducer logic doesn't have to deal with deep levels of nesting, so it will be much simpler
- The logic for retrieving/updating a given item is now simple and consistent. Given an item's type and its ID, we can directly look it up, without having to dig through other objects to find it
- Since each data type is separated, an update like changing the text of a comment would only require new copies of the "comments > byId > comment" portion of the tree



 Page supprimée ! ...


 Ajouter des labels

EXERCICE

Normalize the state



[starwars-redux/tree/tp2](#)


 Page supprimée ! 



ORGANIZING NORMALIZED DATA IN STATE

- A typical application will likely have a mixture of relational data and non-relational data
- One common pattern is to put the relational "tables" under a common parent key, such as "entities"



 Page supprimée ! 


 Ajouter des labels

```
1. {  
2.   simpleDomainData1: {....},  
3.   simpleDomainData2: {....}  
4.   entities : {  
5.     entityType1 : {....},  
6.     entityType2 : {....}  
7.   }  
8.   ui : {  
9.     uiSection1 : {....},  
10.    uiSection2 : {....}  
11.  }  
12. }  
13.
```

RELATIONSHIPS AND TABLES

- Because we're treating a portion of our Redux store as a "database", many of the principles of database design also apply here as well
- For example, a many-to-many relationship, we can model that using an intermediate table that stores the IDs of the corresponding items (often known as a "join table" or an "associative table")



 Page supprimée ! 


 Ajouter des labels

```
1. {  
2.   entities: {  
3.     authors : { byId : {}, allIds : [] },  
4.     books : { byId : {}, allIds : [] },  
5.     authorBook : {  
6.       byId : {  
7.         1 : {  
8.           id : 1,  
9.           authorId : 5,  
10.          bookId : 22  
11.        },  
12.        2 : {  
13.          id : 2,  
14.          authorId : 5,  
15.          bookId : 15,  
16.        }  
17.        3 : {  
18.          id : 3,  
19.          authorId : 42,  
20.          bookId : 12  
21.        }  
22.      },  
23.      allIds : [1, 2, 3]  
24.    }  
25.  }  
26. }  
27.
```


NORMALIZR

- Because APIs frequently send back data in a nested form, that data needs to be transformed into a normalized shape before it can be included in the state tree
- You can define schema types and relations, feed the schema and the response data to Normalizr, and it will output a normalized transformation of the response
- That output can then be included in an action and used to update the store


 Page supprimée ! 


 Ajouter des labels

EXERCICE



Switch to Normlizr


[starwars-redux/tree/tp3](http://localhost:3001/#/0?export&starwars-redux/tree/tp3)

 Page supprimée !

 Ajouter des labels

MIDDLEWARE

 Page supprimée ! 


 Ajouter des labels


MIDDLEWARE

Redux middleware solves different problems than Express middleware, but in a conceptually similar way

It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer

Redux middlewares are used for logging, crash reporting, talking to an asynchronous API, routing, and more

 Page supprimée ! ...



 Ajouter des labels


```
1. // Basic middleware logger
2. const logger = store => next => action => {
3.   console.log('dispatching', action)
4.   let result = next(action)
5.   console.log('next state', store.getState())
6.   return result
7. }
8.
9. // Apply middleware on redux store
```

Middleware takes the next() dispatch function, and returns a dispatch function, which serves as next() to the middleware to the left, and so on. Store stays available as the top-level argument so we have access to some store methods like getState()

```
15.
16. let reducers = combineReducers(rootReducer)
17. let store = createStore(
18.   reducers,
19.   // ...
20. )
```

SIDE EFFECTS

 Page supprimée ! 



 Ajouter des labels


MANAGING SIDE EFFECTS

There's a lot of ways to write and manage asynchronous logic in Javascript.

There's no needs middlewares to use async logic in a Redux app, but it's the recommended approach.


- functions (redux-thunk)
- promises (redux-promise)
- generators (redux-saga)
- observables (redux-observable)
- ...


 Page supprimée ! 

 Ajouter des labels

REDUX-THUNK


- Action creators can return a function instead of an action
- Can be used to delay the dispatch of an action
- The inner function receives the store methods *dispatch* and *getState* as parameters


 Page supprimée ! ...

 Ajouter des labels

Thunks



```
1. export const loginThunk = (username, password) => dispatch => {
2.
3.   dispatch(loginRequest());
4.
5.   postLogin(username, password).then(({ user, msg }) =>
6.     dispatch(loginSuccess(user));
7.     setTimeout(() => {
8.       dispatch(showMessage(msg));
9.     }, 2000);
10.  }, err => {
11.    dispatch(loginFailure(err));
12.  });
13.
14. };
15.
16. dispatch(loginThunk(username, password));
17.
```


 Page supprimée ! ...

 Ajouter des labels

BENEFITS



- Simple in both concept and implementation
- Uses familiar flow control constructs
- Logic is all in one place


 Page supprimée ! 

 Ajouter des labels

THE PROBLEM WITH THIS APPROACH



- No longer dispatching plain action objects
- Thunks are harder to test
- Mixing actions and side effects in asynchronous action creators drastically increases complexity
- There is no clean/easy/etc way to cancel an in-progress thunk


 Page supprimée ! 

 Ajouter des labels

REDUX-SAGA

- Use of ES6 generator functions to control async flow
- Enables complex async workflows via background-thread-like "saga" functions.

 Page supprimée ! 

 Ajouter des labels

Saga

```
1. function* loginSaga() {
2.   while (true) {
3.     // Sleep until a login request action happens
4.     const action = yield take(LOGIN_REQUEST);
5.     const { username, password } = action.payload;
6.     try {
7.       // Login. The actual call is carried out by the middleware
8.       const result = yield call(postLogin, username, password);
9.       const { user, msg } = result;
10.      yield put(loginSuccess(user));
11.      // Wait a bit and show the message
12.      yield call(delay, 2000);
13.      yield put(showMessage(msg));
14.    }
15.    catch (err) {
16.      yield put(loginFailure(err));
17.    }
18.  }
```

Page supprimée !

Ajouter des labels

The mental model is that a saga is like a separate thread in your application that's solely responsible for side effects.

Redux-saga is a redux middleware, which means this thread can be started, paused and cancelled from the main application with normal redux actions, it has access to the full redux application state and it can dispatch redux actions as well.

- [Redux-Saga - official documentation](#)

Page supprimée !

Ajouter des labels

BUILT ON TOP OF GENERATORS

- Generators may be paused in the middle, one or many times, and resumed later, allowing other code to run during these paused periods
- Nothing can pause a generator from the outside; it pauses itself when it comes across a *yield*
- Once a generator has *yield*-paused itself, it cannot resume on its own. An external control must be used to restart the generator
- Enables 2-way message passing into and out of the generator, as it progresses. You send messages out with each *yield*, and you send messages back in with each restart.

Syntax

```
1. // Declaration
2. function* foo() {
3.     yield 1;
4.     yield 2;
5.     yield 3;
6. }
7.
8. // Do nothing
9. var it = foo();
10.
11. // Iterate over the generator
12. console.log( it.next() ); // { value:1, done:false }
13. console.log( it.next() ); // { value:2, done:false }
14. console.log( it.next() ); // { value:3, done:false }
15. console.log( it.next() ); // { value:undefined, done:true }
```


Exercice

```
1. function *foo(x) {  
2.     var y = 2 * (yield (x + 1));  
3.     var z = yield (y / 3);  
4.     return (x + y + z);  
5. }  
6.  
7. var it = foo( 5 );  
8.  
9.  
10. console.log( it.next() ); // note: not sending anything into `next()` here  
11. console.log( it.next( 12 ) );  
12. console.log( it.next( 13 ) );  
13.
```

Page supprimée !

Ajouter des labels

REDUX-SAGA/EFFECTS

To express the Saga logic we yield plain JavaScript Objects from the Generator. We call those objects *Effects*.

An *Effects* is simply an object which contains some information to be interpreted by the middleware.

You can view *Effects* like instructions to the middleware to perform some operation (invoke some asynchronous function, dispatch an action to the store).

This way, when testing the Generator, all we need to do is to check that it yields the expected instruction by doing a simple `deepEqual` on the yielded Object.

```
1. // Call
2. function* fetchProducts() {
3.   const products = yield call(Api.fetch, '/products')
4.   // handle the response...
5. }
6.
7. // Put
8. function* fetchProducts() {
9.   const products = yield call(Api.fetch, '/products')
10.  // create and yield a dispatch Effect
11.  yield put({ type: 'PRODUCTS_RECEIVED', products })
12. }
```

Creates an Effect description that instructs the middleware to call the function fn with args as arguments

```
1. // TakeLatest
2. function* watchFetchProducts() {
3.   yield takeEvery('PRODUCTS_REQUESTED', fetchProducts)
4. }
5.
6. // Take
7. function* loginFlow() {
8.   while (true) {
9.     yield take('LOGIN')
10.    // ... perform the login logic
```

Each time an action which matches pattern is dispatched to the store, takeLatest starts a new saga task in the background. If a saga task was started previously, and if this task is still running, the task will be cancelled

```
1. // Fork
2. function* loginFlow() {
3.   while (true) {
4.     const {user, password} = yield take('LOGIN_REQUEST')
5.     // fork return a Task object
6.     const task = yield fork(authorize, user, password)
7.     const action = yield take(['LOGOUT', 'LOGIN_ERROR'])
8.     if (action.type === 'LOGOUT')
9.       yield cancel(task)
10.    yield call(Api.clearItem, 'token')
11.  }
12. }
13.
14. // Select
15. export const getCart = state => state.cart
```

Creates an Effect description that instructs the middleware to perform a non-blocking call on fn



```
19. const call = yield fork(callFn, getCart,
20. // ...
21. }
22.
```


BENEFITS

- Easy to understand, easy to test
- Excellent documentation
- Logic is all in one place
- Supports very complex operations

THE PROBLEM WITH THIS APPROACH

- Unit testing requires intimate knowledge of the implementation of the saga
- Debugging is difficult


 Page supprimée ! 


 Ajouter des labels

EXERCICE

Switch from Thunk to Saga

[starwars-redux/tree/tp5](https://github.com/robertorocha/starwars-redux/tree/tp5)

 Page supprimée !

 Ajouter des labels

LINKS

- Redux <http://redux.js.org/>
- Reselect <https://github.com/reactjs/reselect>
- The Basics Of ES6 Generators <https://davidwalsh.name/es6-generators>
- Redux Saga <https://redux-saga.js.org/>