An Intro to Redux-saga

•••

and using redux-saga in your application

Cameron Fraser & Matthew Holman

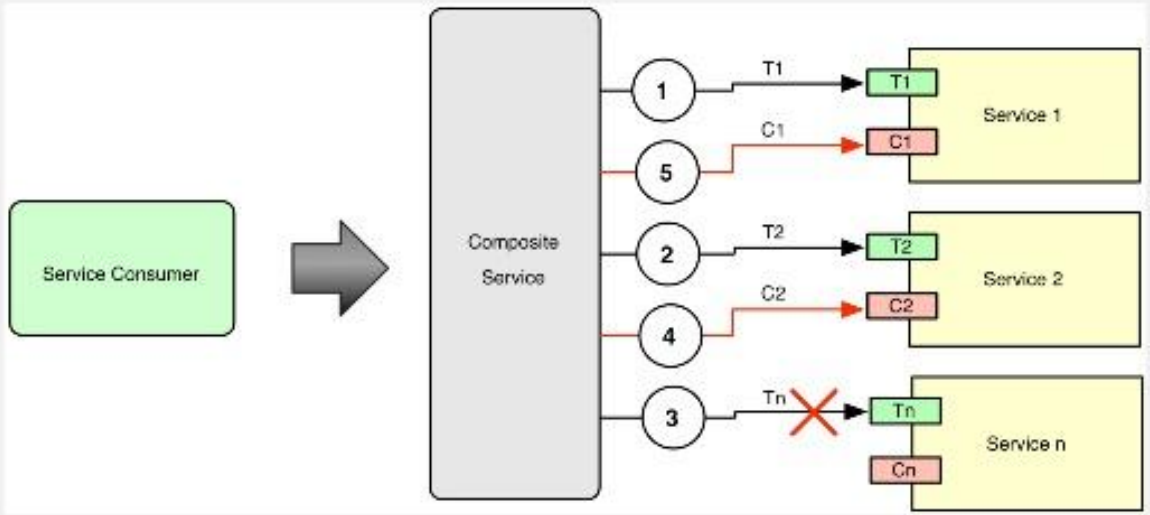
Matt Holman & Cameron Fraser

- React enthusiasts!
- Frontend developers at Cylance, currently developing a react-based, enterprise SPA.
- Cylance is hiring
  - Frontend Developer
  - Also have backend dev & QA roles!

What is redux-saga?

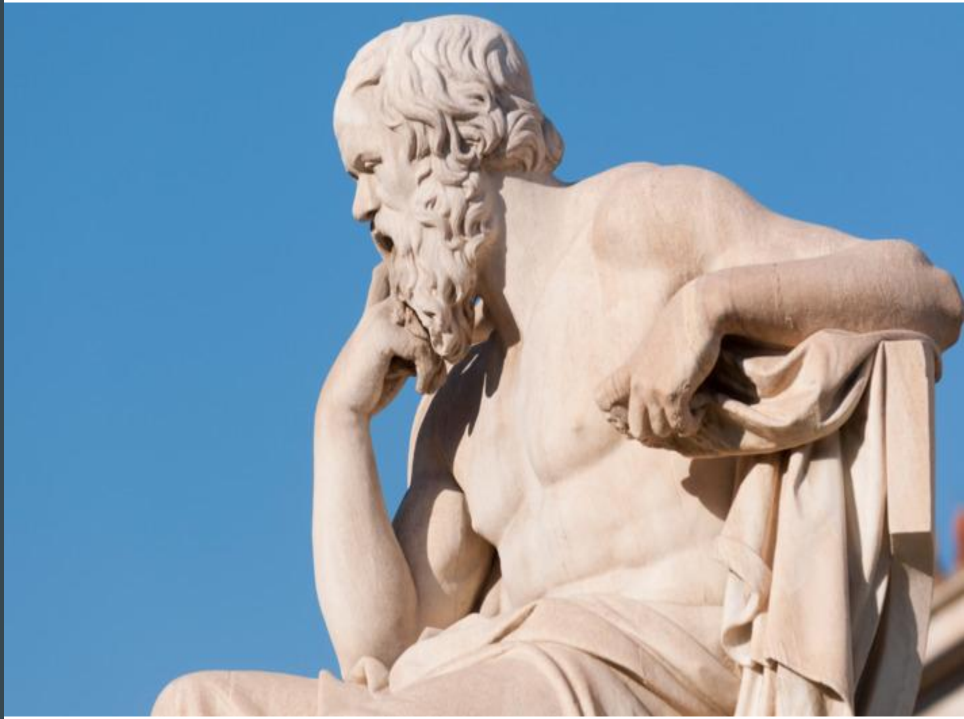
Saga Pattern

Ensures that each step of the business process has a compensating action to undo the work completed in the case of partial failures.



1. Hector Garcia-Molina and Kenneth Salem first defined sagas in relation to distributed systems in their 1987 Princeton University research paper titled ‘Sagas’.

2. Caitie McCaffrey’s “Applying the Saga Pattern”. An excellent YouTube video from 2015 where Caitie explains her experience applying the saga pattern while she was working on Halo 4.



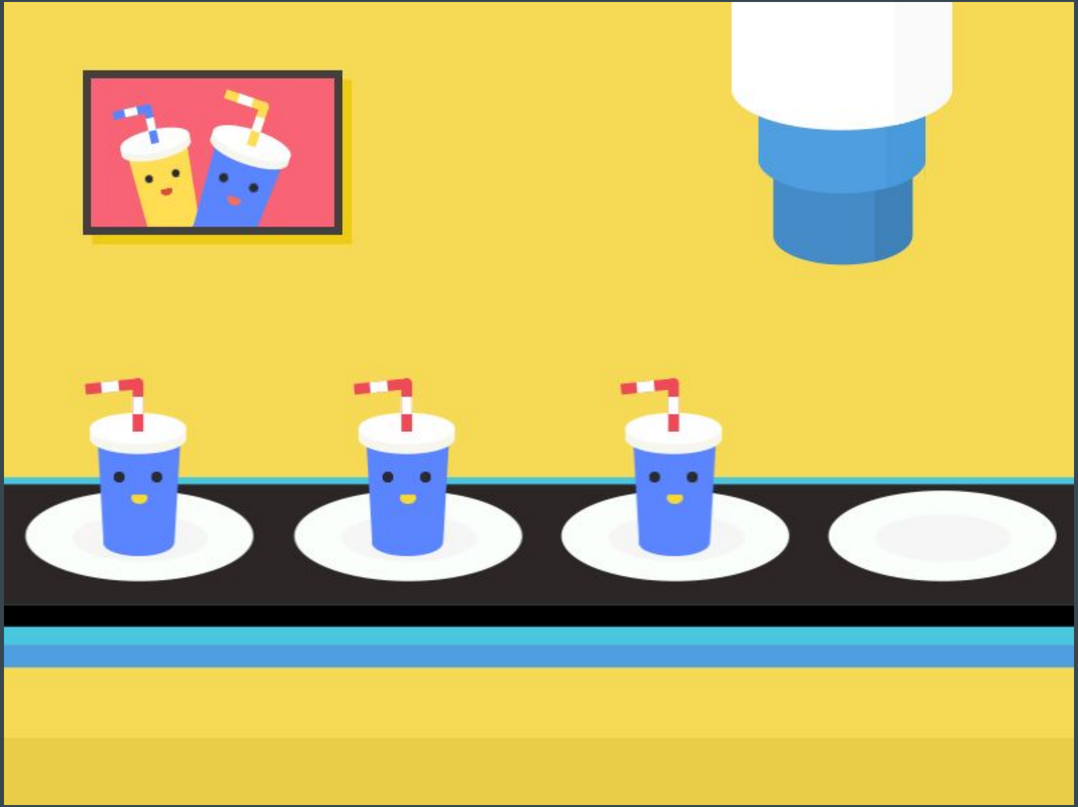
What is ***redux***-saga?

Redux-saga is...

- Redux middleware that helps manage side-effects (API calls, accessing browser cache, etc.) in your application.

- “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.”

Generators



According to the MDN docs: “Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.”

- Generators are a special type of function that do not necessarily run to completion. They can be paused and resumed by outside code.
- This allows us to write code in a synchronous looking and readable format.
- When a generator is first invoked it will return an iterator and by calling .next() on the iterator function, we tell the generator to start execution and stop at the first ‘yield’ keyword it encounters and then yield the value.
- The value returned when calling the .next() function is an object. It has a ‘value’ property and a ‘done’ property. The value contains the yielded value and the ‘done’ property is a boolean that indicates if the function is complete.

Iterator / Generator Comparison

```
for (let i = 1; i < 5; i += 1) {
   console.log(i);
}

// this will immediately return:
// 1
// 2
// 3
// 4
```

```
function* generatorForLoop(num) {
   for (let i = 1; i < num; i += 1) {
     yield console.log(i);
   }
}

const genForLoop = generatorForLoop(5);

// nothing is output until we call .next()
genForLoop.next(); // 1

// even though the console log displays 1, the return
// value is actually the object described earlier,
// which looks like:
// { value: 1, done: false }

// the next value isn't output until the .next() call:
genForLoop.next(); // 2

// and so on...
genForLoop.next(); // 3
genForLoop.next(); // 4
```

Generators

- Generators are currently supported by most browsers.
- Read What Are JavaScript Generators and How to Use Them for a more in-depth explanation.
- Watch Max Stoiber’s ‘Write Simple Async Code with JavaScript Generators’ egghead course.

Why use redux-saga?

Isolate & Manage Side Effects

“In computer science, a function or expression is said to have a side effect if it modifies some state outside its local environment or has an observable interaction with the outside world besides returning a value. Example side effects of a particular function might consist in performing I/O, modifying a non-local variable, modifying a static local variable, modifying an argument passed by reference, or calling other side-effect functions.” - wikipedia

One of the most common side effects is probably a network request (API call), which has your code communicating with a third party (and thus making the request, causing logs to be recorded, caches to be saved or updated, all sorts of effects that are outside the function.

```
componentDidMount() {

   this.fetchAUserList();

}

async fetchAUserList(params) {

   try {
     // Make the API call…
     const res = await axios.get('https://swapi.co/api/people', params);
     this.setState({ userListData: res });

   } catch (error) {
     // If there is an error, we log it with our logger
     this.logError('GET User List Failed', error);
   }
}
```

```
componentDidMount() {

   this.props.fetchAUserList();

}



// ...
// ...

const mapDispatchToProps = (dispatch) => {
   return bindActionCreators({
        fetchAUserList: fetchAUserList
   }, dispatch);
};


// May use this life cycle method if you need to
// re-fetch based on updated-props
// for example, changing the number of users
// to display on a page:
componentDidUpdate(params) {

   this.props.fetchAUserList(params);

}
```

OXI CLEAN

Code Your Async Calls in a Synchronous Fashion

```
export default function* getSomethingSaga() {

   const requestName = 'get-some-cool-data';
   yield put(requestStarted(requestName));

   try {
       const response = yield call(fetch, 'https://swapi.co/api/species');
       yield put(requestSuccess(response, requestName));
       yield put(setSomeData(response));
   } catch (error) {
       yield put(createErrorNotification('Unable to retrieve anything cool.', error));
       yield put(requestFailure(error, requestName));
   } finally {
       yield put(requestFinished(requestName));
   }
}
```

Redux-saga side effect functions are pure!

- A saga does not actually execute the API request.
- Instead, it returns a pure object: { type: 'CALL', func, args }
- The actual *fetch()* execution is taken care of by the redux-saga middleware, and will return the value back into our generator (hence the yield keyword) or throw an error if there was one.
- This is a powerful concept and makes testing sagas much easier than testing thunks!

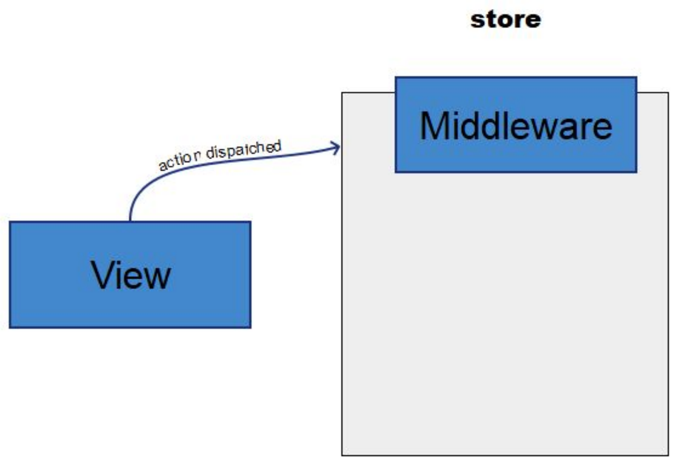
When to start using redux-saga?

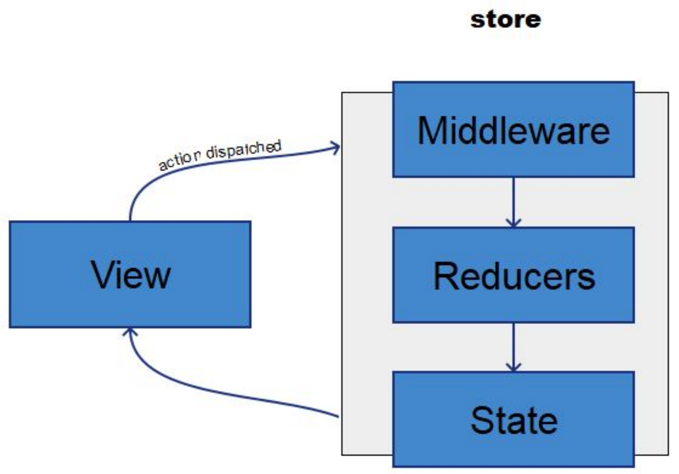
When to start using redux-saga

- API calls and other side effects are becoming difficult to manage
- You plan on rapidly scaling your application in the near future.
- You want an efficient way to organize and manage side effects within your application.
- You want an easy way to test async calls and other side effects.
- Your components are currently coupled with side effects and could benefit from abstracting them away.

MakeAGIF.com

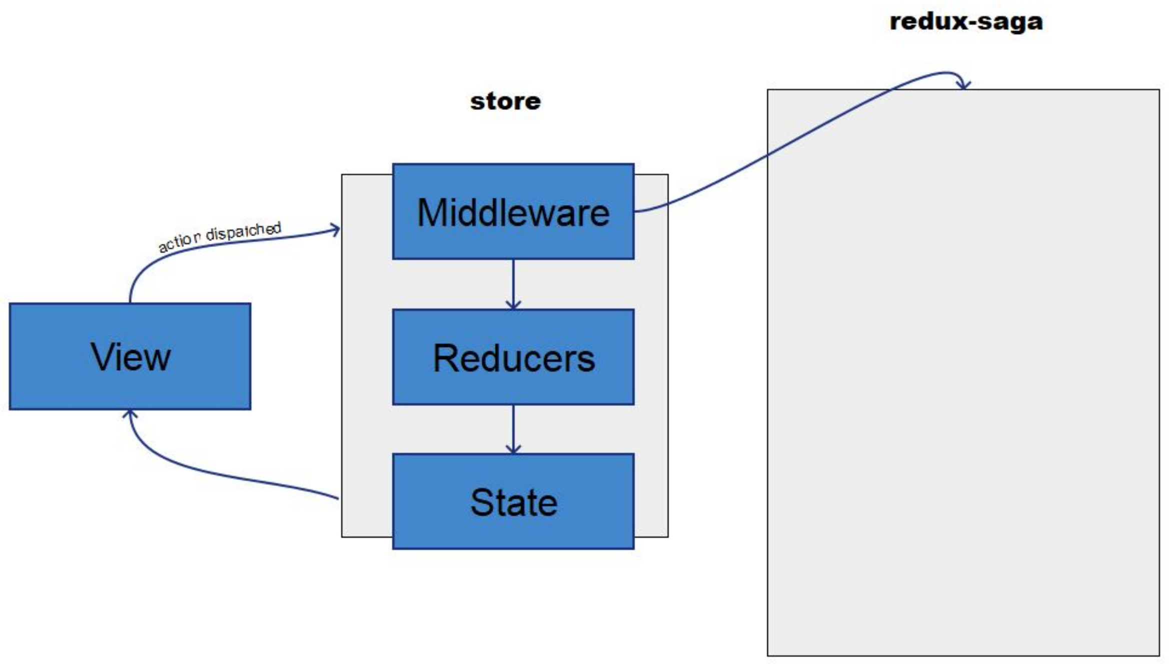
View

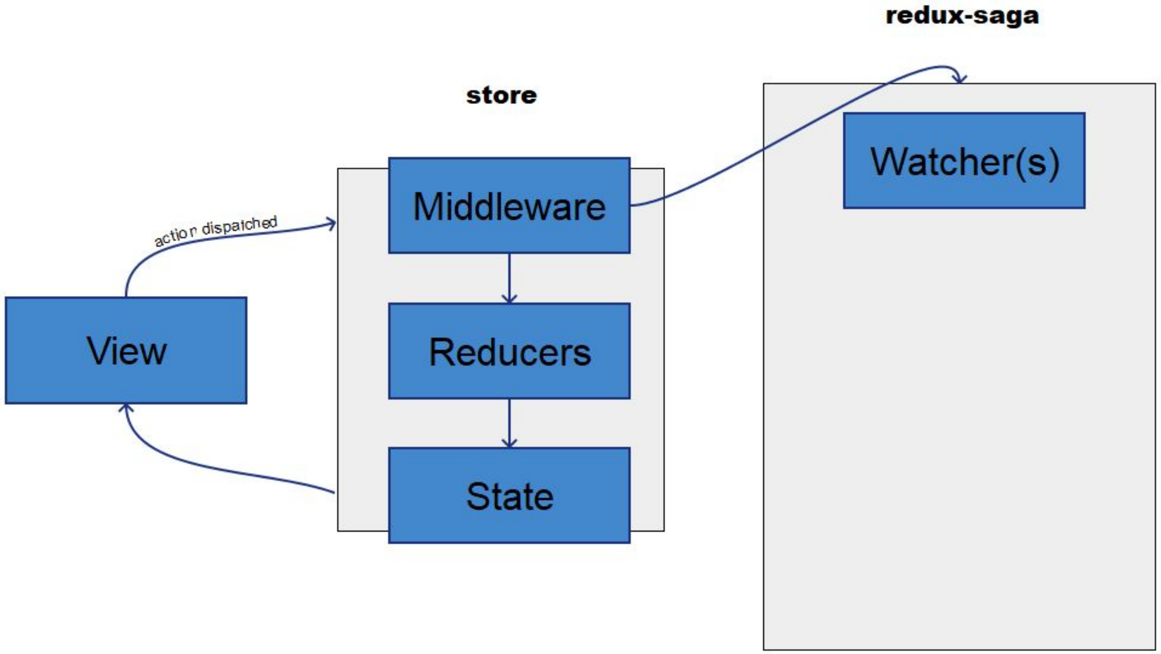

store
Middleware
action dispatched
View

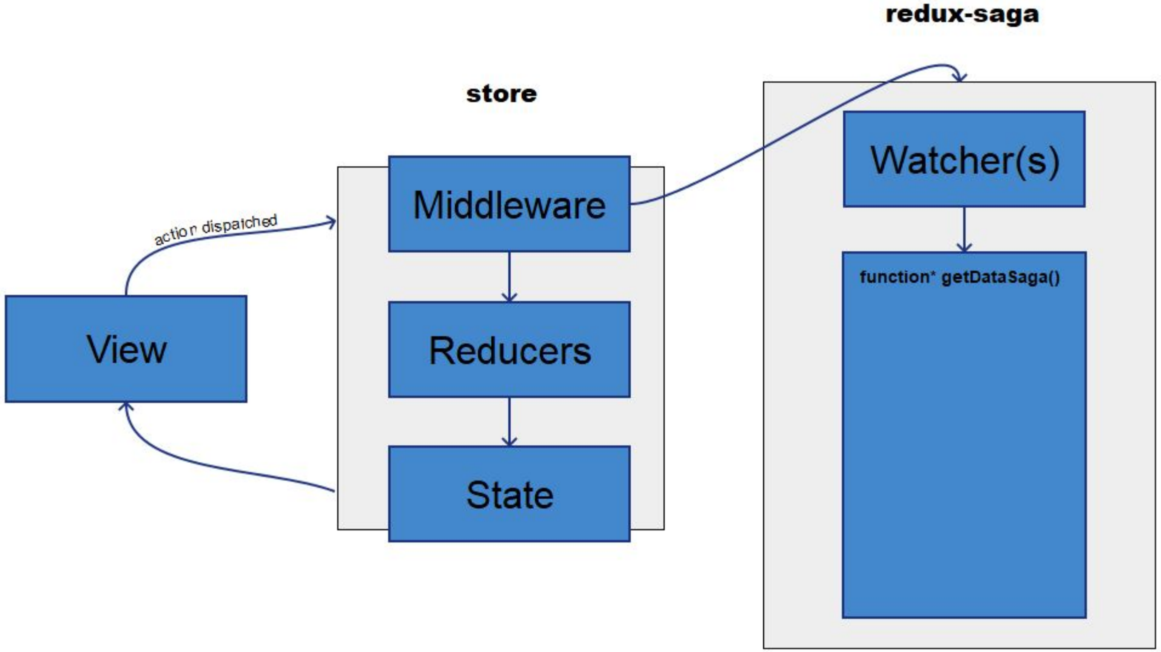

store
Middleware
Reducers
State
View
action dispatched

From the Redux-saga docs:

“It's important to note that when an action is dispatched to the store, the middleware first forwards the action to the reducers and then notifies the Sagas. This means that when you query the Store's State, you get the State after the action has been applied. However, this behavior is only guaranteed if all subsequent middlewares call next(action) synchronously. If any subsequent middleware calls next(action) asynchronously (which is unusual but possible), then the sagas will get the state from before the action is applied. Therefore it is recommended to review the source of each subsequent middleware to ensure it calls next(action) synchronously, or else ensure that redux-saga is the last middleware in the call chain.”

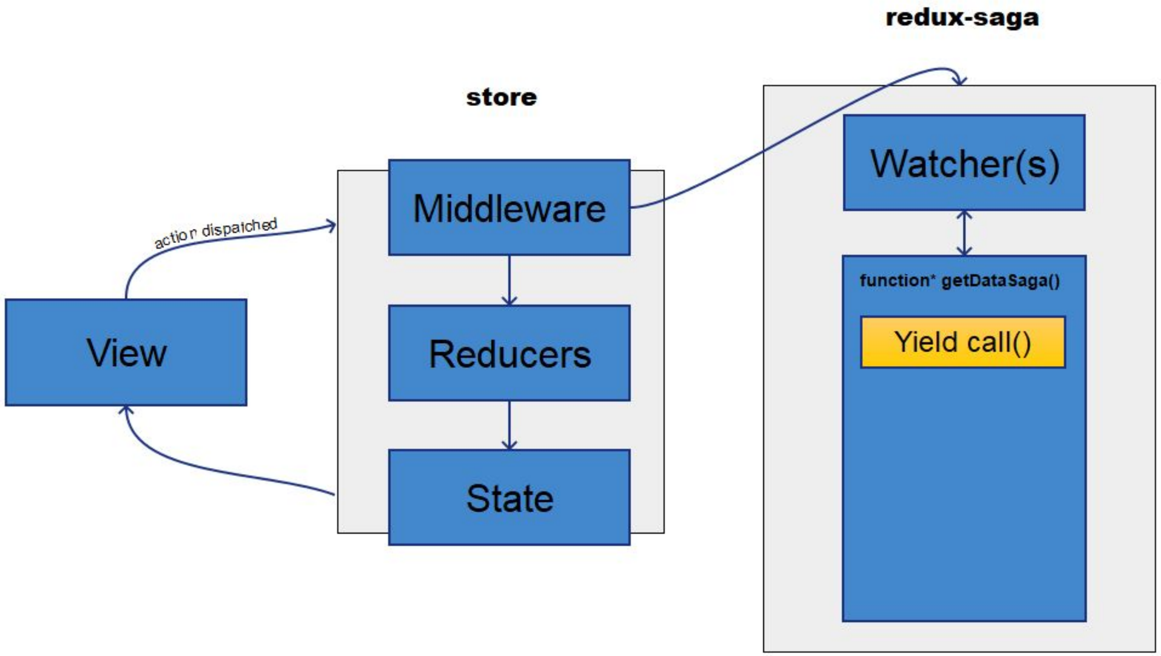

redux-saga
store
Middleware
Reducers
State
View
action dispatched

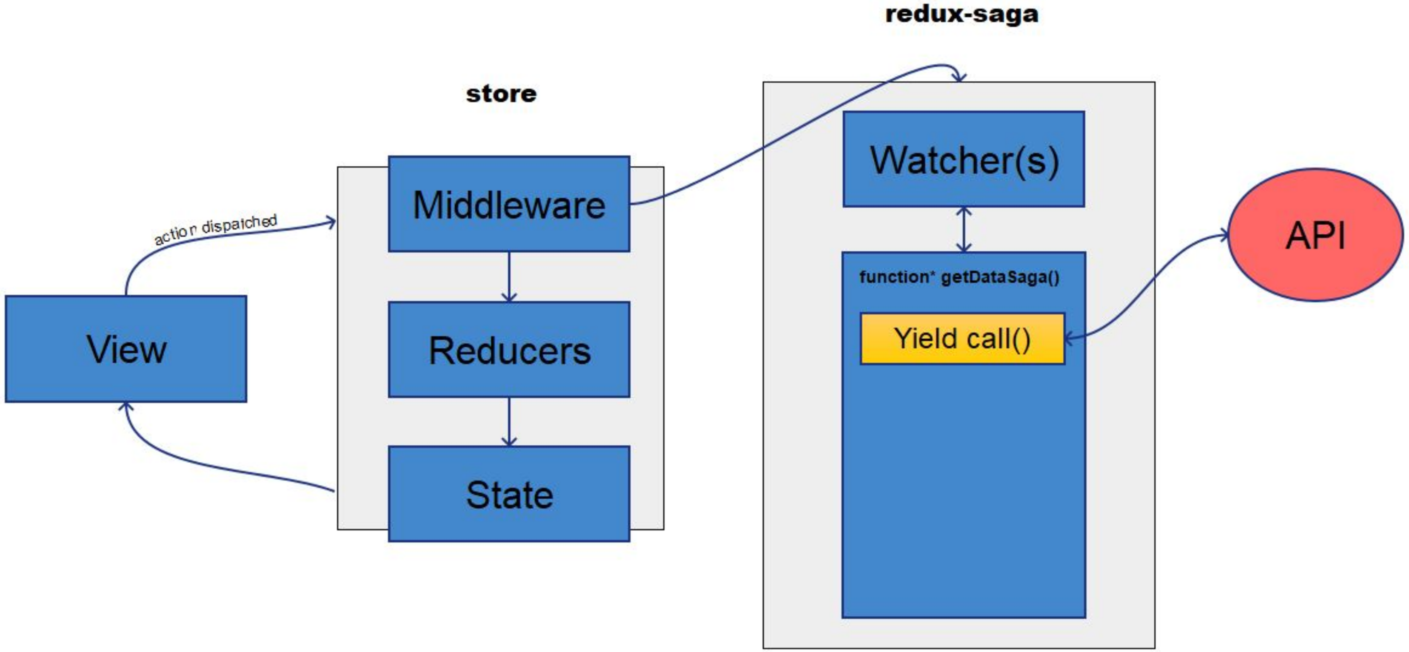

redux-saga
store
Watcher(s)
Middleware
action dispatched
View
Reducers
State

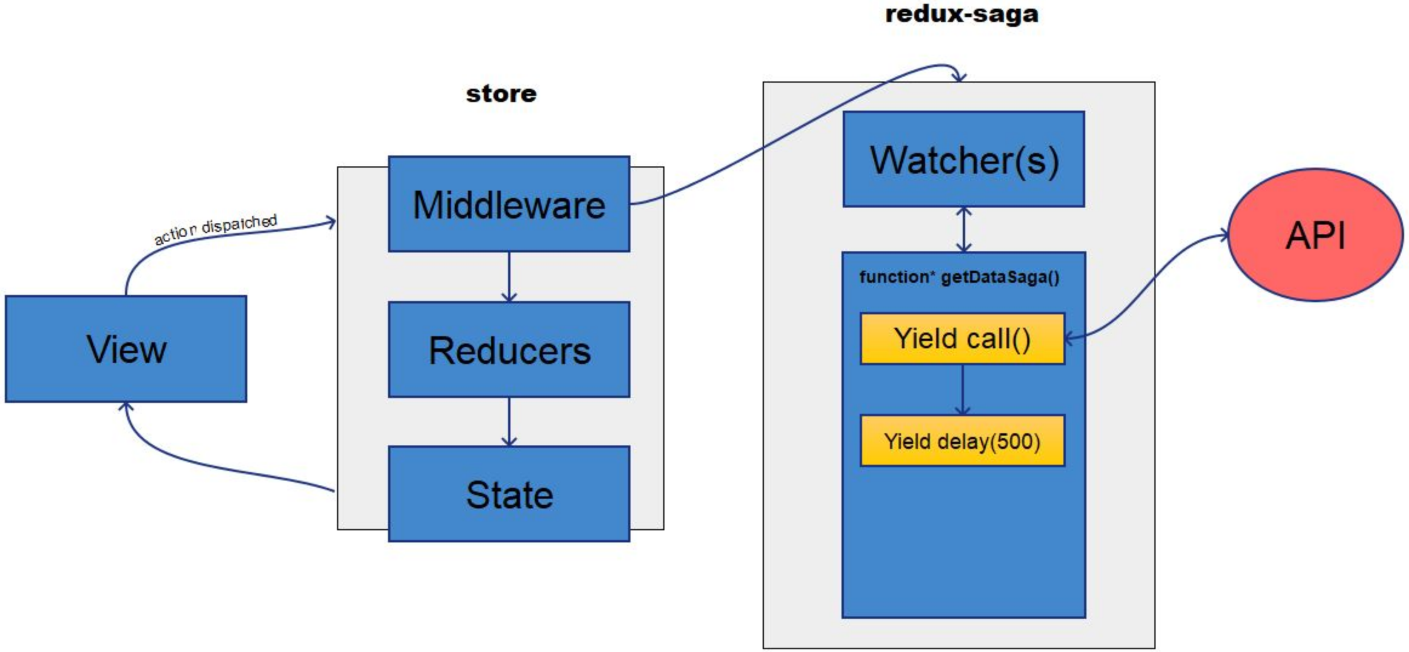

redux-saga
store
Watcher(s)
Middleware
action dispatched
function* getDataSaga()
View
Reducers
State

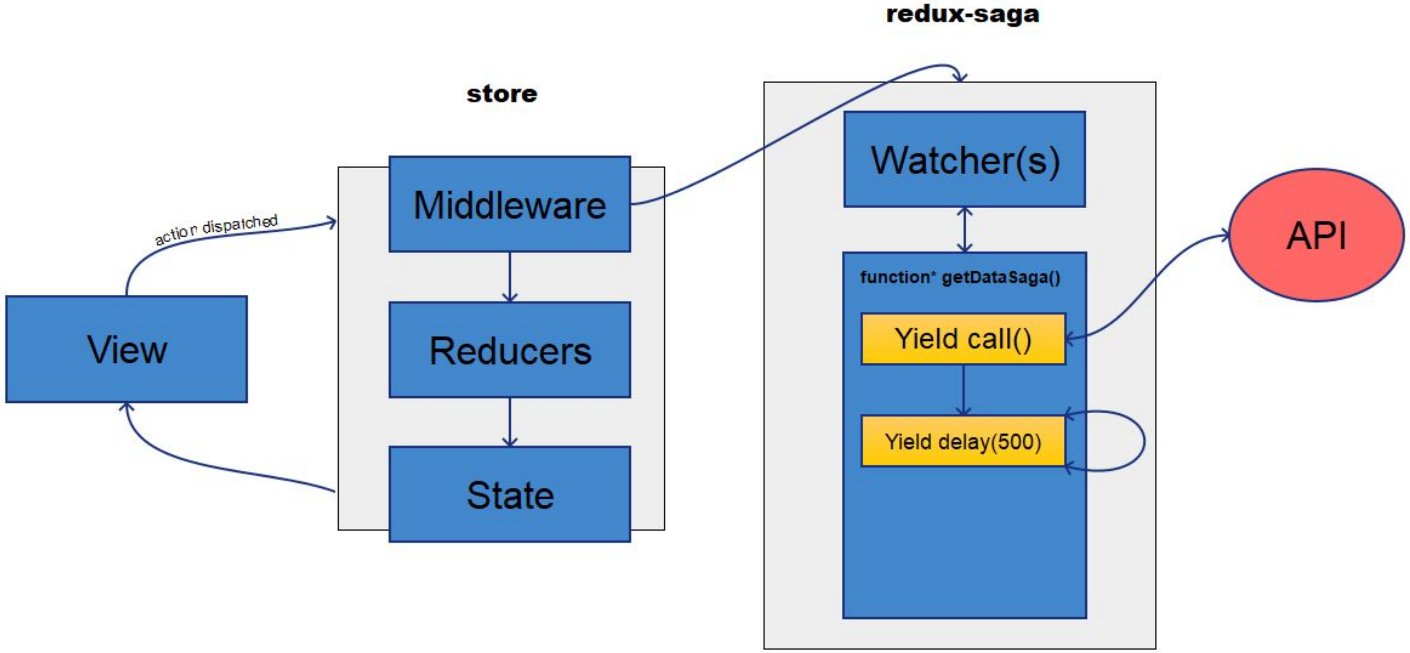
Redux-saga effects

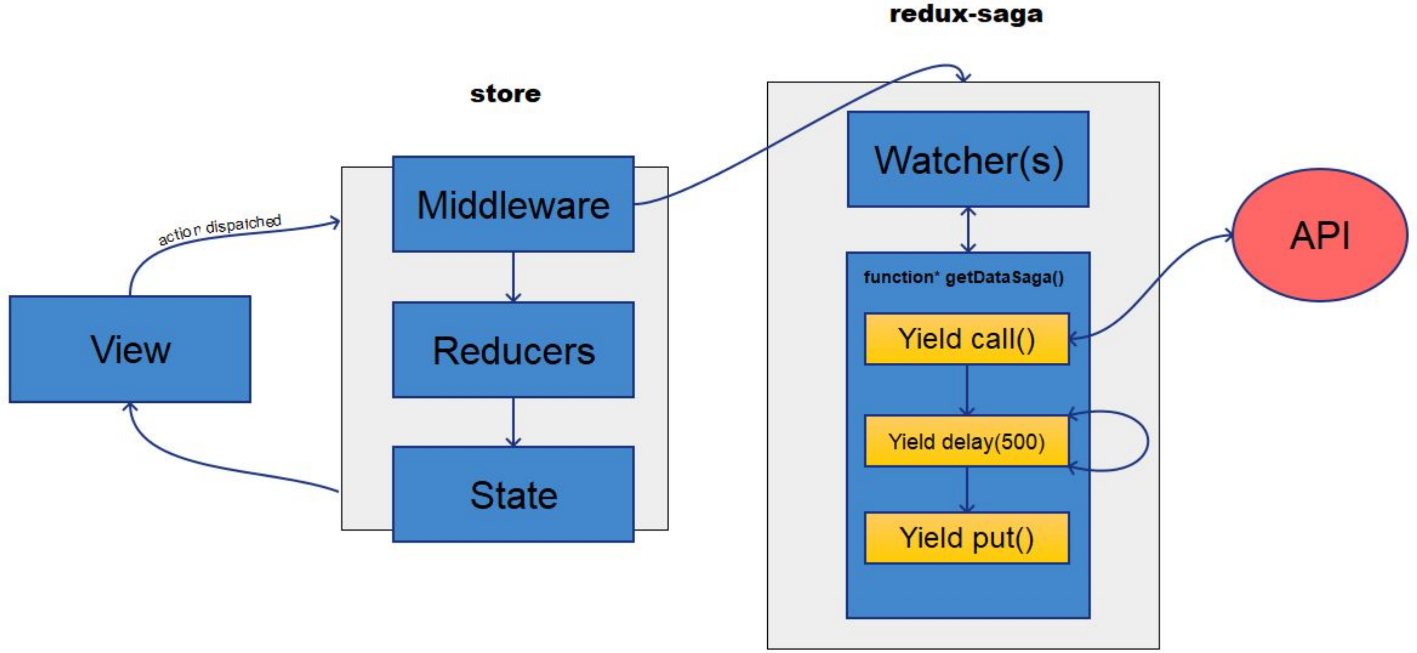
- Helpers:
  - takeEvery()
  - takeLatest()
- Creators:
  - take()
  - put()
  - call()
  - apply()
  - fork()
  - cancel()
- Combinators:
  - race()
  - all()
- Utils:
  - channel()
  - delay()

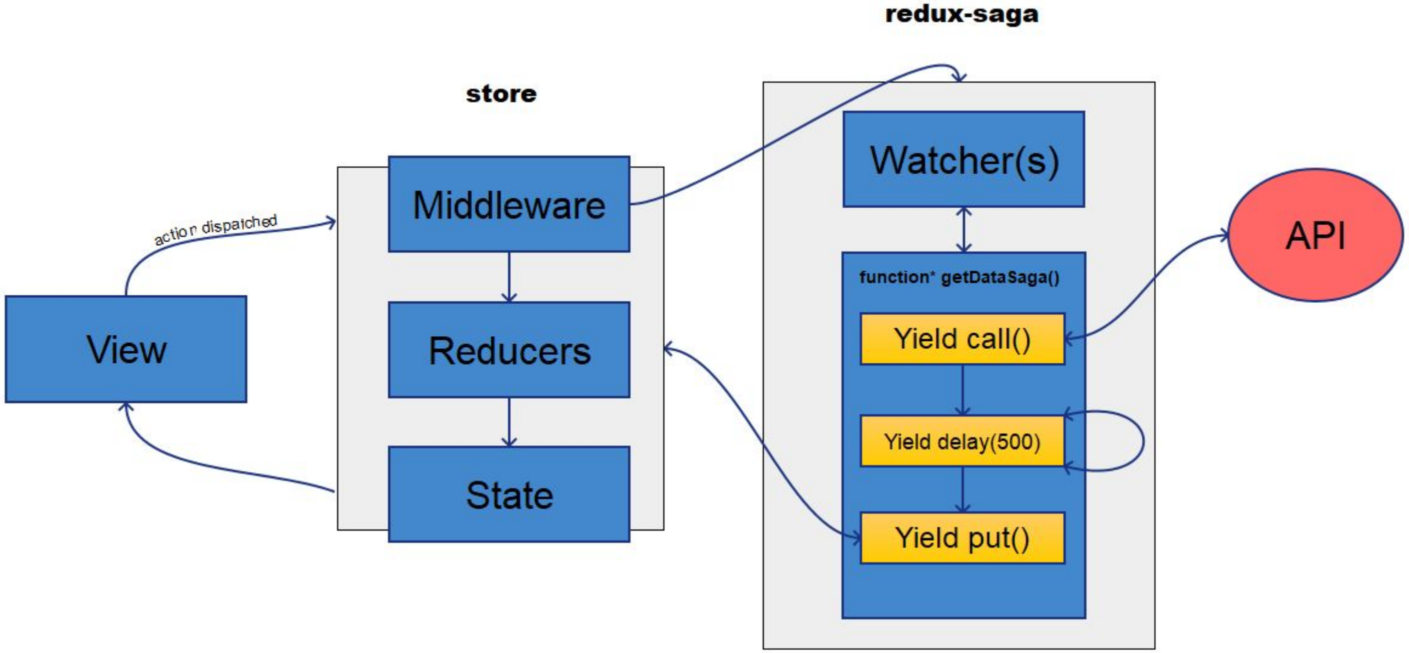

redux-saga
store
Middleware
Reducers
State
View
action dispatched
Watcher(s)
function* getDataSaga()
Yield call()

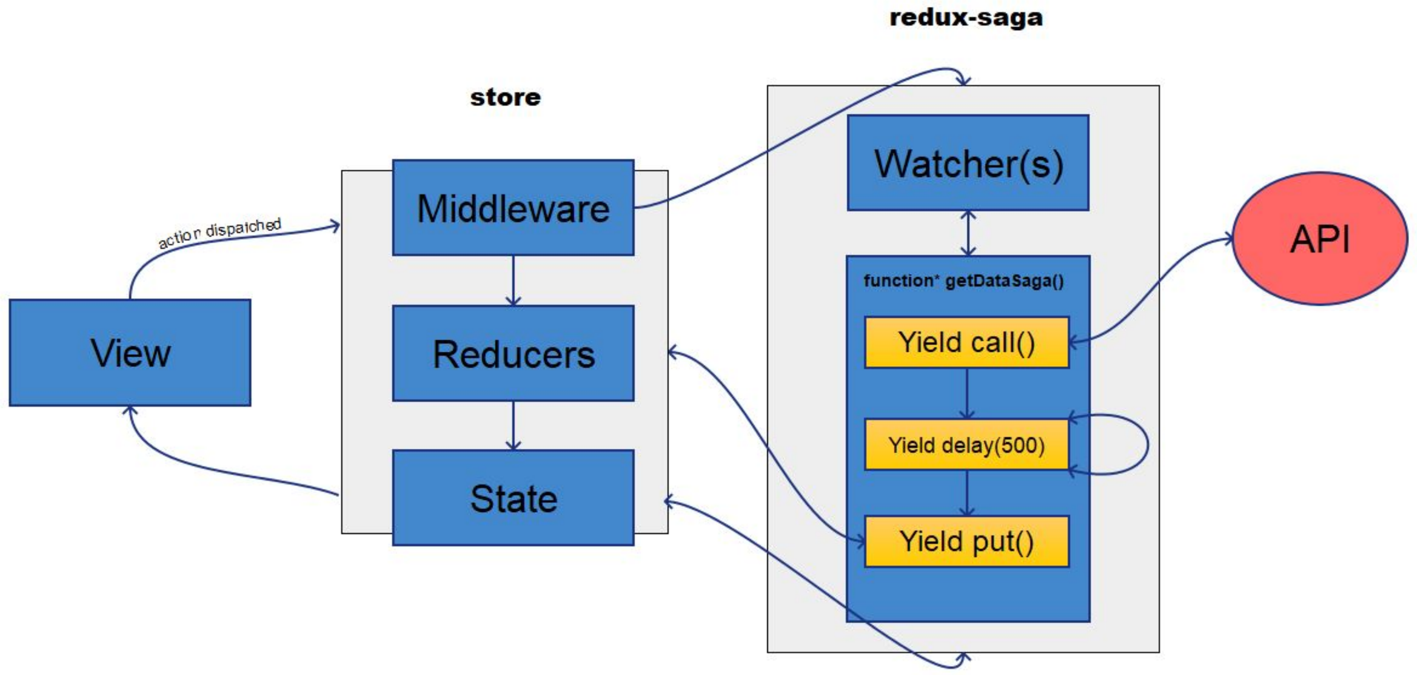

redux-saga
store
Watcher(s)
Middleware
API
action dispatched
function* getDataSaga()
View
Reducers
Yield call()
State

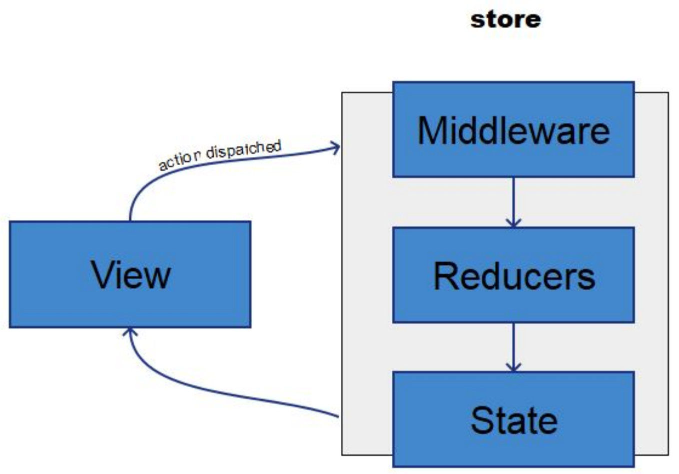

redux-saga
store
Watcher(s)
Middleware
API
action dispatched
function* getDataSaga()
View
Reducers
Yield call()
Yield delay(500)
State


redux-saga
store
Middleware
Reducers
State
View
action dispatched
Watcher(s)
function* getDataSaga()
Yield call()
Yield delay(500)
API


redux-saga
store
Middleware
Reducers
State
View
action dispatched
Watcher(s)
function* getDataSaga()
Yield call()
Yield delay(500)
Yield put()
API


redux-saga
store
Middleware
Reducers
State
View
action dispatched
Watcher(s)
function* getDataSaga()
Yield call()
Yield delay(500)
Yield put()
API


redux-saga
store
Watcher(s)
Middleware
API
action dispatched
function* getDataSaga()
View
Yield call()
Reducers
Yield delay(500)
State
Yield put()


store
Middleware
Reducers
State
View
action dispatched

Create a watcher for the API request action

```
import { takeLatest } from 'redux-saga/effects';
import { GET_SOMETHING_COOL, } from 'app/actions';
import { getSomethingSaga } from 'app/sagas';

export default function* watchers() {
   yield takeLatest(GET_SOMETHING_COOL, getSomethingSaga);
}
```

```
import { call, put } from 'redux-saga/effects';
import { requestSuccess, requestFailure, requestStarted } from 'app/actions';


export default function* getSomethingSaga() {

   const requestName = 'get-some-cool-data';
   yield put(requestStarted(requestName));

   try {
       const response = yield call(fetch, 'https://swapi.co/api/species');
       yield put(requestSuccess(requestName));
       yield put(setSomeData(response));
   } catch (error) {
       yield put(createErrorNotification('Unable to retrieve anything cool.', error));
       yield put(requestFailure(error, requestName));

} finally {
       yield put(requestFinished(requestName));
   }
}
```

163
ARVIND
DAILY NEWS
NEW YORK, NEW YORK

Q & A