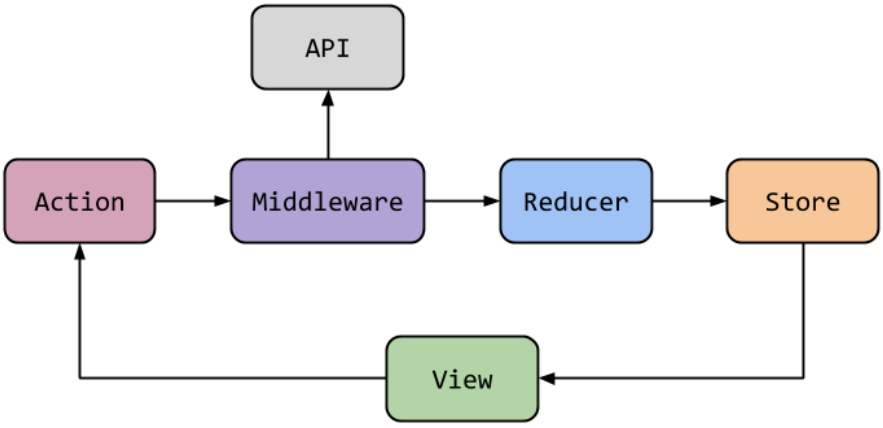
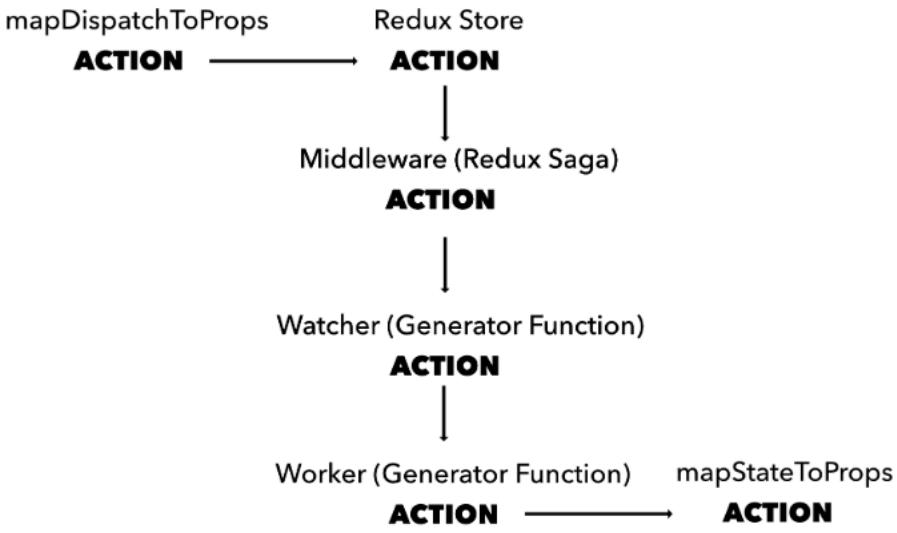
Saga

Tuesday, 14 January 2020 3:46 AM



When to use Redux Saga



In an application using Redux, when you fire an action something changes in the state of the app.

As this happens, you might need to do something that derives from this state change.

For example you might want to:

- make a HTTP call to a server

- send a WebSocket event
- fetch some data from a GraphQL server
- save something to the cache or browser local storage

…you got the idea.

Those are all things that don't really relate to the app state, or are async, and you need to move them into a place different than your actions or reducers (while you technically *could*, it's not a good way to have a clean codebase). Enter Redux Saga, a Redux middleware helping you with side effects.

How it works behind the scenes

Being a Redux Middleware, Redux Saga can intercept Redux Actions, and inject its own functionality.

A **saga** is some "story" that reacts to an **effect** that your code is causing.
We create a **middleware** with a list of **sagas** to run, which can be one or more, and we connect this middleware to the Redux store.
A **saga** is a **generator** function. When a **promise** is run and **yielded**, the middleware **suspends** the **saga** until the **promise** is **resolved**.
Once the **promise** is **resolved** the middleware **resumes** the saga, until the next **yield** statement is found, and there it is **suspended** again until its **promise resolves**.

Inside the saga code, you will generate **effects** using a few special helper functions provided by the `redux-saga` package. To start with, we can list:

- `takeEvery()`
- `takeLatest()`
- `take()`
- `call()`
- `put()`

When an **effect** is executed, the **saga** is **paused** until the **effect** is **fulfilled**.

For example:

```
import { takeEvery } from 'redux-saga/effects'

const handleNewMessage = function* handleNewMessage(params) {
  const socket = new WebSocket('ws://localhost:8989')
  yield takeEvery('ADD_MESSAGE', (action) => {
```

```
    socket.send(JSON.stringify(action))
  })
}

export default handleNewMessage
```

When the **middleware** executes the `handleNewMessage` saga, it **stops** at the `yield takeEvery` instruction and **waits** (*asynchronously*, of course) until the `ADD_MESSAGE` action is **dispatched**. Then it runs its callback, and the **saga** can **resume**.

Basic Helpers

Helpers are abstractions on top of the low-level saga APIs.
Let's introduce the most basic helpers you can use to run your effects:

- `takeEvery()`
- `takeLatest()`
- `take()`
- `put()`
- `call()`

`takeEvery()`

`takeEvery()`, used in some examples, is one of those helpers.
In the code:

```
import { takeEvery } from 'redux-saga/effects'

function* watchMessages() {
  yield takeEvery('ADD_MESSAGE', postMessageToServer)
}
```

The `watchMessages` generator pauses until an `ADD_MESSAGE` action fires, and **every time** it fires, it's going to call the `postMessageToServer` function, infinitely, and concurrently (there is no need for `postMessageToServer` to terminate its execution before a new once can run)

`takeLatest()`

Another popular helper is `takeLatest()`, which is very similar to `takeEvery()` but only allows one function handler to run at a time, avoiding concurrency. If another action is fired when the handler is still running, it will cancel it, and run again with the latest data available.
As with `takeEvery()`, the generator never stops and continues to run the effect when the specified action occurs.

`take()`

`take()` is different in that it only waits a single time. When the action it's waiting for occurs, the promise resolves and the iterator is resumed, so it can go on to the next instruction set.

`put()`

Dispatches an action to the Redux store. Instead of passing in the Redux store or the dispatch action to the saga, you can just use `put()`:

```
yield put({ type: 'INCREMENT' })
yield put({ type: "USER_FETCH_SUCCEEDED", data: data })
```

which returns a plain object that you can easily inspect in your tests (more on testing later).

`call()`

When you want to call some function in a saga, you can do so by using a yielded plain function call that returns a promise:

```
delay(1000)
```

but this does not play nice with tests. Instead, `call()` allows you to wrap that function call and returns an object that can be easily inspected:

```
call(delay, 1000)
```

returns

```
{ CALL: {fn: delay, args: [1000]}}
```

Running effects in parallel

Running effects in parallel is possible using `all()` and `race()`, which are very different in what they do.

`all()`

If you write

```
import { call } from 'redux-saga/effects'

const todos = yield call(fetch, '/api/todos')
const user = yield call(fetch, '/api/user')
```

the second `fetch()` call won't be executed until the first one succeeds.

To execute them in parallel, wrap them into `all()`:

```
import { all, call } from 'redux-saga/effects'

const [todos, user]  = yield all([
  call(fetch, '/api/todos'),
  call(fetch, '/api/user')
])
```

`all()` won't be resolved until both `call()` return.

`race()`

`race()` differs from `all()` by not waiting for all of the helpers calls to return.
It just waits for one to return, and it's done.
It's a race to see which one finishes first, and then we forget about the other participants.
It's typically used to cancel a background task that runs forever until something occurs:

```
import { race, call, take } from 'redux-saga/effects'

function* someBackgroundTask() {
  while(1) {
    //...
  }
}

yield race([
  bgTask: call(someBackgroundTask),
  cancel: take('CANCEL_TASK')
])
```

when the `CANCEL_TASK` action is emitted, we stop the other task that would otherwise run forever.