Beginner Tutorial

Objectives of this tutorial

This tutorial attempts to introduce redux-saga in a (hopefully) accessible way.

For our getting started tutorial, we are going to use the trivial Counter demo from the Redux repo. The application is quite basic but is a good fit to illustrate the basic concepts of redux-saga without being lost in excessive details.

The initial setup

Before we start, clone the **tutorial repository**.

> The final code of this tutorial is located in the `sagas` branch.

Then in the command line, run:

```
$ cd redux-saga-beginner-tutorial
$ npm install
```

To start the application, run:

```
$ npm start
```

We are starting with the most basic use case: 2 buttons to `Increment` and `Decrement` a counter. Later, we will introduce asynchronous calls.

If things go well, you should see 2 buttons `Increment` and `Decrement` along with a message below showing `Counter: 0`.

> In case you encountered an issue with running the application. Feel free to create an issue on the **tutorial repo**.

Hello Sagas!

We are going to create our first Saga. Following the tradition, we will write our 'Hello, world' version for Sagas.

Create a file `sagas.js` then add the following snippet:

```
export function* helloSaga() {
  console.log('Hello Sagas!')
}
```

So nothing scary, just a normal function (except for the `*`). All it does is print a greeting message into the console.

In order to run our Saga, we need to:

- create a Saga middleware with a list of Sagas to run (so far we have only one `helloSaga`)
- connect the Saga middleware to the Redux store

We will make the changes to `main.js`:

```
// ...
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

// ...
import { helloSaga } from './sagas'

const sagaMiddleware = createSagaMiddleware()
const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(helloSaga)

const action = type => store.dispatch({type})

// rest unchanged
```

First we import our Saga from the `./sagas` module. Then we create a middleware using the factory function `createSagaMiddleware` exported by the `redux-saga` library.

Before running our `helloSaga`, we must connect our middleware to the Store using `applyMiddleware`. Then we can use the `sagaMiddleware.run(helloSaga)` to start our Saga.

So far, our Saga does nothing special. It just logs a message then exits.

Making Asynchronous calls

Now let's add something closer to the original Counter demo. To illustrate asynchronous calls, we will add another button to increment the counter 1 second after the click.

First things first, we'll provide an additional button and a callback `onIncrementAsync` to the UI component.

```
const Counter = ({ value, onIncrement, onDecrement, onIncrementAsync }) =>
  <div>
    <button onClick={onIncrementAsync}>
      Increment after 1 second
    </button>
    {' '}
    <button onClick={onIncrement}>
      Increment
    </button>
    {' '}
    <button onClick={onDecrement}>
      Decrement
    </button>
    <hr />
    <div>
      Clicked: {value} times
    </div>
  </div>
```

Next we should connect the `onIncrementAsync` of the Component to a Store action.

We will modify the `main.js` module as follows

```
function render() {
  ReactDOM.render(
    <Counter
      value={store.getState()}
      onIncrement={() => action('INCREMENT')}
      onDecrement={() => action('DECREMENT')}
      onIncrementAsync={() => action('INCREMENT_ASYNC')} />,
    document.getElementById('root')
  )
}
```

Note that unlike in redux-thunk, our component dispatches a plain object action.

Now we will introduce another Saga to perform the asynchronous call. Our use case is as follows:

> On each `INCREMENT_ASYNC` action, we want to start a task that will do the following

> - Wait 1 second then increment the counter

Add the following code to the `sagas.js` module:

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

const delay = (ms) => new Promise(res => setTimeout(res, ms))

// ...

// Our worker Saga: will perform the async increment task
export function* incrementAsync() {
  yield delay(1000)
  yield put({ type: 'INCREMENT' })
}

// Our watcher Saga: spawn a new incrementAsync task on each INCREMENT_ASYNC
export function* watchIncrementAsync() {
  yield takeEvery('INCREMENT_ASYNC', incrementAsync)
}
```

Time for some explanations.

We create a `delay` function that returns a **Promise** that will resolve after a specified number of milliseconds. We'll use this function to *block* the Generator.

Sagas are implemented as **Generator functions** that *yield* objects to the redux-saga middleware. The yielded objects are a kind of instruction to be interpreted by the middleware.

When a Promise is yielded to the middleware, the middleware will suspend the Saga until the Promise completes. In the above example, the `incrementAsync` Saga is suspended until the Promise returned by `delay` resolves, which will happen after 1 second.

Once the Promise is resolved, the middleware will resume the Saga, executing code until the next yield. In this example, the next statement is another yielded object: the result of calling `put({type: 'INCREMENT'})`, which instructs the middleware to dispatch an `INCREMENT` action.

`put` is one example of what we call an *Effect*. Effects are plain JavaScript objects which contain instructions to be fulfilled by the middleware. When a middleware retrieves an Effect yielded by a Saga, the Saga is paused until the Effect is fulfilled.

So to summarize, the `incrementAsync` Saga sleeps for 1 second via the call to `delay(1000)`, then dispatches an `INCREMENT` action.

Next, we created another Saga `watchIncrementAsync`. We use `takeEvery`, a helper function provided by `redux-saga`, to listen for dispatched `INCREMENT_ASYNC` actions and run `incrementAsync` each time.

Now we have 2 Sagas, and we need to start them both at once. To do that, we'll add a `rootSaga` that is responsible for starting our other Sagas. In the same file `sagas.js`, refactor the file as follows:

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

const delay = (ms) => new Promise(res => setTimeout(res, ms))

function* helloSaga() {
  console.log('Hello Sagas!')
}

function* incrementAsync() {
  yield delay(1000)
  yield put({ type: 'INCREMENT' })
}

function* watchIncrementAsync() {
  yield takeEvery('INCREMENT_ASYNC', incrementAsync)
}

// notice how we now only export the rootSaga
// single entry point to start all Sagas at once
```

```
export default function* rootSaga() {
  yield all([
    helloSaga(),
    watchIncrementAsync()
  ])
}
```

This Saga yields an array with the results of calling our two sagas, `helloSaga` and `watchIncrementAsync`. This means the two resulting Generators will be started in parallel. Now we only have to invoke `sagaMiddleware.run` on the root Saga in `main.js`.

```
// ...
import rootSaga from './sagas'

const sagaMiddleware = createSagaMiddleware()
const store = ...
sagaMiddleware.run(rootSaga)

// ...
```

Making our code testable

We want to test our `incrementAsync` Saga to make sure it performs the desired task.

Create another file `sagas.spec.js`:

```
import test from 'tape'

import { incrementAsync } from './sagas'

test('incrementAsync Saga test', (assert) => {
  const gen = incrementAsync()

  // now what ?
})
```

`incrementAsync` is a generator function. When run, it returns an iterator object, and the iterator's `next` method returns an object with the following shape

```
gen.next() // => { done: boolean, value: any }
```

The `value` field contains the yielded expression, i.e. the result of the expression after the `yield`. The `done` field indicates if the generator has terminated or if there are still more 'yield' expressions.

In the case of `incrementAsync`, the generator yields 2 values consecutively:

1. `yield delay(1000)`
2. `yield put({type: 'INCREMENT'})`

So if we invoke the next method of the generator 3 times consecutively we get the following results:

```
gen.next() // => { done: false, value: <result of calling delay(1000)> }
gen.next() // => { done: false, value: <result of calling put({type:
'INCREMENT'})> }
gen.next() // => { done: true, value: undefined }
```

The first 2 invocations return the results of the yield expressions. On the 3rd invocation since there is no more yield the `done` field is set to true. And since the `incrementAsync` Generator doesn't return anything (no `return` statement), the `value` field is set to `undefined`.

So now, in order to test the logic inside `incrementAsync`, we'll have to iterate over the returned Generator and check the values yielded by the generator.

```
import test from 'tape'

import { incrementAsync } from './sagas'

test('incrementAsync Saga test', (assert) => {
  const gen = incrementAsync()

  assert.deepEqual(
    gen.next(),
    { done: false, value: ??? },
    'incrementAsync should return a Promise that will resolve after 1
second'
  )
})
```

The issue is how do we test the return value of `delay`? We can't do a simple equality test on Promises. If `delay` returned a *normal* value, things would've been easier to test.

Well, `redux-saga` provides a way to make the above statement possible. Instead of calling `delay(1000)` directly inside `incrementAsync`, we'll call it *indirectly* and export it to make a subsequent deep comparison possible:

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

export const delay = (ms) => new Promise(res => setTimeout(res, ms))

// ...

export function* incrementAsync() {
  // use the call Effect
  yield call(delay, 1000)
  yield put({ type: 'INCREMENT' })
}
```

Instead of doing `yield delay(1000)`, we're now doing `yield call(delay, 1000)`. What's the difference?

In the first case, the yield expression `delay(1000)` is evaluated before it gets passed to the caller of `next` (the caller could be the middleware when running our code. It could also be our test code which runs the Generator function and iterates over the returned Generator). So what the caller gets is a Promise, like in the test code above.

In the second case, the yield expression `call(delay, 1000)` is what gets passed to the caller of `next`. `call` just like `put`, returns an Effect which instructs the middleware to call a given function with the given arguments. In fact, neither `put` nor `call` performs any dispatch or asynchronous call by themselves, they return plain JavaScript objects.

```
put({type: 'INCREMENT'}) // => { PUT: {type: 'INCREMENT'} }
call(delay, 1000)        // => { CALL: {fn: delay, args: [1000]}}
```

What happens is that the middleware examines the type of each yielded Effect then decides how to fulfill that Effect. If the Effect type is a `PUT` then it will dispatch an action to the Store. If the Effect is a `CALL` then it'll call the given function.

This separation between Effect creation and Effect execution makes it possible to test our Generator in a surprisingly easy way:

```
import test from 'tape'

import { put, call } from 'redux-saga/effects'
import { incrementAsync, delay } from './sagas'

test('incrementAsync Saga test', (assert) => {
  const gen = incrementAsync()

  assert.deepEqual(
    gen.next().value,
    call(delay, 1000),
    'incrementAsync Saga must call delay(1000)'
  )

  assert.deepEqual(
    gen.next().value,
    put({type: 'INCREMENT'}),
    'incrementAsync Saga must dispatch an INCREMENT action'
  )

  assert.deepEqual(
    gen.next(),
    { done: true, value: undefined },
    'incrementAsync Saga must be done'
  )

  assert.end()
})
```

Since `put` and `call` return plain objects, we can reuse the same functions in our test code. And to test the logic of `incrementAsync`, we iterate over the generator and do `deepEqual` tests on its values.

In order to run the above test, run:

```
$ npm test
```

which should report the results on the console.

Edit this page