Async and Redux

Juggling all the things in a contemporary web app
The Central Question

How do I handle async in redux?
Asynchronous
JS is single-threaded

Function one

Function two

Function three
JS is single-threaded

Heap

Stack
request(url, Callback)

Queue

Event Loop

API's
JS is single-threaded
JS is single-threaded

- Heap
- Stack
- API’s

- Event Loop
- Queue
  - Callback
JS is single-threaded

Heap

Stack

Callback

Event Loop

Queue

API’s
# Summary of ASYNC in JavaScript  (unscientific, biased)

<table>
<thead>
<tr>
<th>Method</th>
<th>Learning curve</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callbacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Promises</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Async/Await (feature)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generators</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Callbacks

```javascript
request('http://some.api', function() {
  // code here runs after you fetch api
})
```
Callbacks - Unexpected Order

console.log('first thing...')

request('http://some.api', function() {
    // code here runs after you fetch api
    console.log('third thing...')
})

console.log('second thing...')
fs.readdir(source, function (err, files) {
    if (err) {
        console.log('Error finding files: ' + err)
    } else {
        files.forEach(function (filename, fileIndex) {
            console.log(filename)
            gm(source + filename).size(function (err, values) {
                if (err) {
                    console.log('Error identifying file size: ' + err)
                } else {
                    console.log(filename + ' : ' + values)
                    aspect = (values.width / values.height)
                    widths.forEach(function (width, widthIndex) {
                        height = Math.round(width / aspect)
                        console.log('resizing ' + filename + ' to ' + height + 'x' + height)
                        this.resize(width, height).write(dest + 'w' + width + '_ ' + filename, function(err) {
                            if (err) console.log('Error writing file: ' + err)
                        })
                    }, this)
                }
            })
        })
    }
})
Callbacks - not on the STACK

```javascript
let p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise has resolved')
  }, 1000)
});
p.then(function(data) {
  return data
});
moreData = "Promise has resolved"
```

Debugger paused (index.js:11)
- Local
  - moreData: "Promise has resolved"
- this: Window
- Script
- Global
- Breakpoints
  - No Breakpoints
Callbacks - inversion of control

```javascript
start(trackingSomething, function() {
    startRunningMyApp()
})
```
# Summary of ASYNC in JavaScript  (unscientific, biased)

<table>
<thead>
<tr>
<th>Method</th>
<th>Learning curve</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callbacks</td>
<td>Easy</td>
<td>Pyramid of doom, hard to reason about, inversion of control</td>
</tr>
<tr>
<td>Promises</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Async/Await (feature)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generators</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Promises

- are objects
- can have 3 states: pending, fulfilled, rejected
- handles success (resolve) and failure (reject)
- has two functions - then and catch
- Immutable - can only resolve or reject once
- Promise.all - waits for ALL promises to resolve
- Promise.race - waits for the first promise to resolve
let p1 = new Promise((resolve, reject) => {
    setTimeout(function () {
        resolve('Promise has been fulfilled!')
    }, 250)
})

p1.then((success) => {
    console.log(success)
})
Promises

- many implementations (native ES6, Bluebird, Q, jQuery, YUI, RSVP)
- still has a lot of nesting which can make it hard to read
- what to do inside then actually takes a bit of time and effort to understand
- no cancellation
let p = new Promise((resolve, reject) => {})
p.then((result) => {
    // return something here
})
.then((nextResult) => {
    // or nothing will be here!
})
## Summary of ASYNC in JavaScript (unscientific, biased)

<table>
<thead>
<tr>
<th>Method</th>
<th>Learning curve</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callbacks</td>
<td>Easy</td>
<td>Pyramid of doom, hard to reason about, inversion of control</td>
</tr>
<tr>
<td>Promises</td>
<td>Hard</td>
<td>Managing scope / context, chaining hard to read, many implementations</td>
</tr>
<tr>
<td>Async/Await (feature)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generators</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Async / Await

```javascript
async function getMeStuff() {
    const a = await fetch('http://some.api');
    const b = await fetch('http://some-other.api');
    return a + b;
}
```
Async / Await

• No way to cancel async/await without helper code or third party code
# Summary of ASYNC in JavaScript (unscientific, biased)

<table>
<thead>
<tr>
<th>Method</th>
<th>Learning curve</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callbacks</td>
<td>Easy</td>
<td>Pyramid of doom, hard to reason about, inversion of control</td>
</tr>
<tr>
<td>Promises</td>
<td>Hard</td>
<td>Managing scope / context, chaining hard to read, many implementations</td>
</tr>
<tr>
<td>Async/Await (feature)</td>
<td>Easy</td>
<td>Cannot cancel</td>
</tr>
<tr>
<td>Generators</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generators

Dan Abramov @dan_abramov · 21 Dec 2015
Replying to @dan_abramov

The funny part is I initially tried to implement middleware system using generators but I didn't understand them.

gaearon

For example consider a logger. I want to log input actions and atoms, but delegate all handling to dispatcher. Should middleware be a function accepting a generator? Function accepting a generator function? Should it call `.next()` explicitly?

```javascript
function *logger(next) {
    let something = yield something else;
    // ???????????!!! I don't even
}
```
Generators

```javascript
function *generateNumbers() {
    yield 'start';
    for (var i = 0; i < 3; i++) {
        yield 'next: ' + i
    }
    return 'done'
}

let gen = generateNumbers();
console.log( gen.next() ); // { value: 'start', done: false }
console.log( gen.next() ); // { value: 'next: 0', done: false }
console.log( gen.next() ); // { value: 'next: 1', done: false }
console.log( gen.next() ); // { value: 'next: 2', done: false }
console.log( gen.next() ); // { value: 'done', done: true }
```
function *getMeStuff() {
    var a = yield fetch('http://some.api');
    var b = yield fetch('http://some-other.api');
    return a + b;
}
Generators

```javascript
function *alwaysGenerating() {
    while(true) {
        yield 'always'
    }
}

let always = alwaysGenerating()
console.log( always.next() ); // always
console.log( always.next() ); // always
```
Generators

```javascript
function *runUntil() {
  while(true) {
    try {
      yield 'trySomething'
    } catch (error) {
      yield error
    }
  }
}
```
Benefits of using Generators

• code that looks synchronous and is easy to reason about
• can pause or cancel execution of code
• can run multiple functions in parallel
Cons of using Generators

• usually need third-party libraries to reduce boilerplate
• Hard to write lots of generator code by hand
• Lots of new concepts for JavaScript developers
<table>
<thead>
<tr>
<th>Method</th>
<th>Learning curve</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callbacks</td>
<td>Easy</td>
<td>Pyramid of doom, hard to reason about, inversion of control</td>
</tr>
<tr>
<td>Promises</td>
<td>Hard</td>
<td>Managing scope / context, chaining hard to read, many implementations</td>
</tr>
<tr>
<td>Async/Await (feature)</td>
<td>Easy</td>
<td>Cannot cancel await out of the box</td>
</tr>
<tr>
<td>Generators</td>
<td>Questionable</td>
<td>Not as well known to JavaScript developers</td>
</tr>
</tbody>
</table>
How does this all relate to Redux?

- Actions
- API
- Views
- Dispatcher
- Middleware
- Reducer(s)
- State
Before Redux (and other Flux architectures)
Before Redux (and other Flux architectures)
Redux Architecture

- Actions
- API
- Views
- Dispatcher
- Middleware
- Reducer(s)
- State
Async is complicated in Redux…

- Many different ways to do async in JS
- Lots of opinions
- Pros and cons to each method
- lots of middleware options in redux to handle async
- Most async middleware uses actions for async, but some use reducers
Why middleware?

Actions  API  Dispatcher

Store  Middleware  Reducer(s)

Views  State
Sooo

<table>
<thead>
<tr>
<th>Library</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>redux-thunk</td>
<td>5,081</td>
</tr>
<tr>
<td>redux-saga</td>
<td>7,668</td>
</tr>
<tr>
<td>redux-observable</td>
<td>2,826</td>
</tr>
</tbody>
</table>
And oh hey

<table>
<thead>
<tr>
<th>Library</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>redux-promise</td>
<td>1,531</td>
</tr>
<tr>
<td>redux-loop</td>
<td>1,171</td>
</tr>
<tr>
<td>redux-promise-middleware</td>
<td>878</td>
</tr>
<tr>
<td>redux-effects</td>
<td>449</td>
</tr>
<tr>
<td>redux-side-effect</td>
<td>24</td>
</tr>
<tr>
<td>redux-funk</td>
<td>15</td>
</tr>
</tbody>
</table>
What are you using to handle async in redux?

- 59% redux-thunk
- 13% redux-promise/-middleware
- 17% redux-saga
- 11% redux-observable

142 votes • Final results
I backed Redux Saga—I think it’s an awesome project.
I think async action creators are suboptimal. How to introduce declarative effects without alienating most developers is a tricky question.
## Summary (unscientific, biased)

<table>
<thead>
<tr>
<th>Features</th>
<th>Readability</th>
<th>Testability</th>
<th>Difficulty</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>redux-thunk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redux-saga</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redux-observable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
redux-thunk

- 14 lines of code
- just a function that returns a function with `store.dispatch` and `store.getState` passed along
- can handle parallel-ish (promise.all) and race - which one comes first (promise.race)
export function fetchLyric () {
    return async (dispatch, getState) => {
        dispatch(createAction(FETCHING_LYRIC))
        try {
            const response = await axios.get('/api/passionfruit')
            const json = await response.data
            dispatch(createAction(FETCH_LYRIC_SUCCESS)(json))
        } catch (e) {
            dispatch(createAction(FETCH_LYRIC_FAILED)(e))
        }
    }
}
test('it creates FETCH_SUCCESS when fetching the lyric has been done', () => {
  moxios.wait(function () {
    let request = moxios.requests.mostRecent()
    request.respondWith({
      status: 200,
      response: {
        'lyric': "Passionate from miles away / Passive with the things you say / Passin' up on my old ways / I can't blame you no, no"
      }
    })
  })

  const expectedActions = [
    {type: FETCHING_LYRIC},
    {type: FETCH_SUCCESS, payload: {
      'lyric': "Passionate from miles away / Passive with the things you say / Passin' up on my old ways / I can't blame you no, no"
    }}
  ]

  const store = mockStore({
    isFetching: false,
    didError: false,
    lyric: ''
  })

  return store.dispatch(fetchLyric().then(() => {
    expect(store.getActions()).toEqual(expectedActions)
  }))
})
## Summary (unscientific, biased)

<table>
<thead>
<tr>
<th></th>
<th>Readability</th>
<th>Testability</th>
<th>Difficulty</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>redux-thunk</td>
<td>Questionable</td>
<td>Meh</td>
<td>Easy</td>
<td>Depends</td>
</tr>
<tr>
<td>redux-saga</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redux-observable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Redux-saga

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

export function* fetchData(action) {
    try {
        const data = yield call(Api.fetchLyric, action.payload.url)
        yield put({type: "FETCH_LYRIC_SUCCESS", data})
    } catch (error) {
        yield put({type: "FETCH_LYRIC_FAILED", error})
    }
}

function* watchFetchData() {
    yield takeEvery('FETCHING_LYRIC', fetchData)
}
```
Sagas do A LOT (part TWO)

• Redux-saga has effects helpers
• the effects helpers can run the api calls, not you
• This makes testing easy
• Once you learn the api the code is easy to reason about
Effects helpers

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

// inside of a generator function:
const products = yield call(Api.fetch, '/products')
```
Effect is a plain JavaScript object

```javascript
{
  CALL: {
    fn: Api.fetch,
    args: ['./products']
  }
}
```
Redux Saga Test

test('fetchLyric Saga test', () => {
    const gen = fetchLyric()

    expect(gen.next().value).toEqual(call(axios.get, '/api/passionfruit'))

    expect(gen.next().value).toEqual(put('{{type: FETCH_SUCCEEDED}}'))
})
test('it creates FETCH_SUCCESS when fetching the lyric has been done', () => {
  moxios.wait(function () {
    let request = moxios.requests.mostRecent()
    request.respondWith({
      status: 200,
      response: {
        'lyric': "Passionate from miles away / Passive with the things you say / Passin' up on my old ways / I can't blame you no, no"
      }
    })
  })

  const expectedActions = [
    {type: FETCHING_LYRIC},
    {type: FETCH_SUCCESS, payload: {
      'lyric': "Passionate from miles away / Passive with the things you say / Passin' up on my old ways / I can't blame you no, no"
    }}
  ]

  const store = mockStore({
    isFetching: false,
    didError: false,
    lyric: ''
  })

  return store.dispatch(fetchLyric())
    .then(() => {
      expect(store.getActions()).toEqual(expectedActions)
    })
})
Sagas are HARD

• Community is not as big (if you need help, it won’t come as quickly)
• The redux-saga api is not re-useable knowledge
• Surface area api of sagas is larger than thunk middleware
Summary (unscientific, biased)

<table>
<thead>
<tr>
<th></th>
<th>Readability</th>
<th>Testability</th>
<th>Difficulty</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>redux-thunk</td>
<td>Questionable</td>
<td>Meh</td>
<td>Easy</td>
<td>Depends</td>
</tr>
<tr>
<td>redux-saga</td>
<td>Very</td>
<td>Very</td>
<td>Hard</td>
<td>Much. So</td>
</tr>
<tr>
<td>redux-observable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
export const fetchLyric = createAction(FETCH_LYRIC)
const fetchLyricSuccess = createAction(FETCH_LYRIC_SUCCESS)
const fetchLyricFail = createAction(FETCH_LYRIC_ERROR)

export const fetchLyricEpic = action$ =>
  action$.ofType(FETCH_LYRIC)
    .mergeMap(action =>
      ajax.getJSON('/passionfruit')
        .map(response => fetchLyricSuccess(response))
        .catch(error => Observable.of(fetchLyricFail(error)))
    )
test('it creates FETCH_LYRIC_SUCCESS when OK!', () => {
    const payload = {
        'lyric': "Passionate from miles away / Passive with the things you say / Passin' up on my old ways / I can't blame you no, no"
    }

    const action$ = ActionsObservable.of({type: FETCH_LYRIC})

    return fetchLyricEpic(action$).toPromise()
        .then((actionReceived) => {
            expect(actionReceived.type).toBe('FETCH_LYRIC_SUCCESS')
            expect(actionReceived.payload).toEqual(payload)
        })
})
## Summary (unscientific, biased)

<table>
<thead>
<tr>
<th></th>
<th>Readability</th>
<th>Testability</th>
<th>Difficulty</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>redux-thunk</td>
<td>Questionable</td>
<td>Meh</td>
<td>Easy</td>
<td>Depends</td>
</tr>
<tr>
<td>redux-saga</td>
<td>Very</td>
<td>Very</td>
<td>Hard</td>
<td>Much. So</td>
</tr>
<tr>
<td>redux-observable</td>
<td>Very</td>
<td>In-progress</td>
<td>Depends</td>
<td>Much. So</td>
</tr>
</tbody>
</table>
Summary (TL; DR)

- Most of the time you can use redux-thunk
- If testing is important to you, use redux-saga
- If you need cancellation, use redux-saga
- If you have developers coming from C#, C++, PHP, or Python look at redux-saga