A Short History

of

XMLHttpRequest & Promises

A brownbag presentation at

MX®

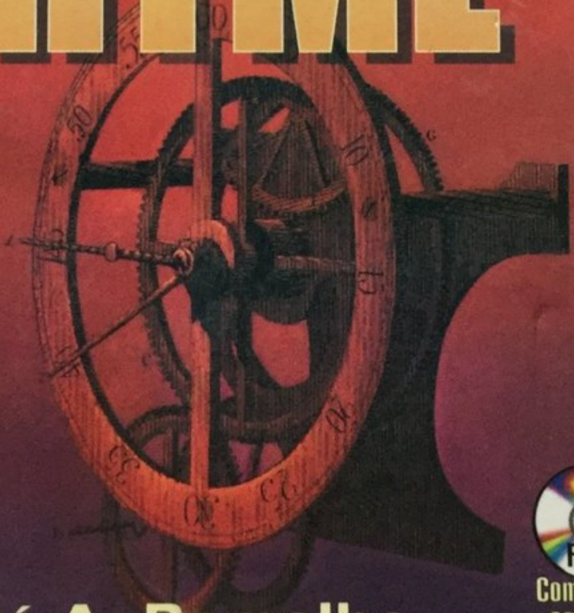
by Seth House @whiteinge

XMLHttpRequest

History of XHR

Learn

Advanced HTML 4.0 with DHTML



José A. Ramalho

Free Companion CD-ROM Included

Fetch data after page load

Fetch data after page load

- iframe trick.

Fetch data after page load

- iframe trick.
- Keep HTTP response connection open & stream the response body.

Fetch data after page load

- iframe trick.
- Keep HTTP response connection open & stream the response body.
- Dynamically add script tags.

The first XMLHttpRequest

- 1998 — Outlook Web Access (OWA) project.

The first XMLHttpRequest

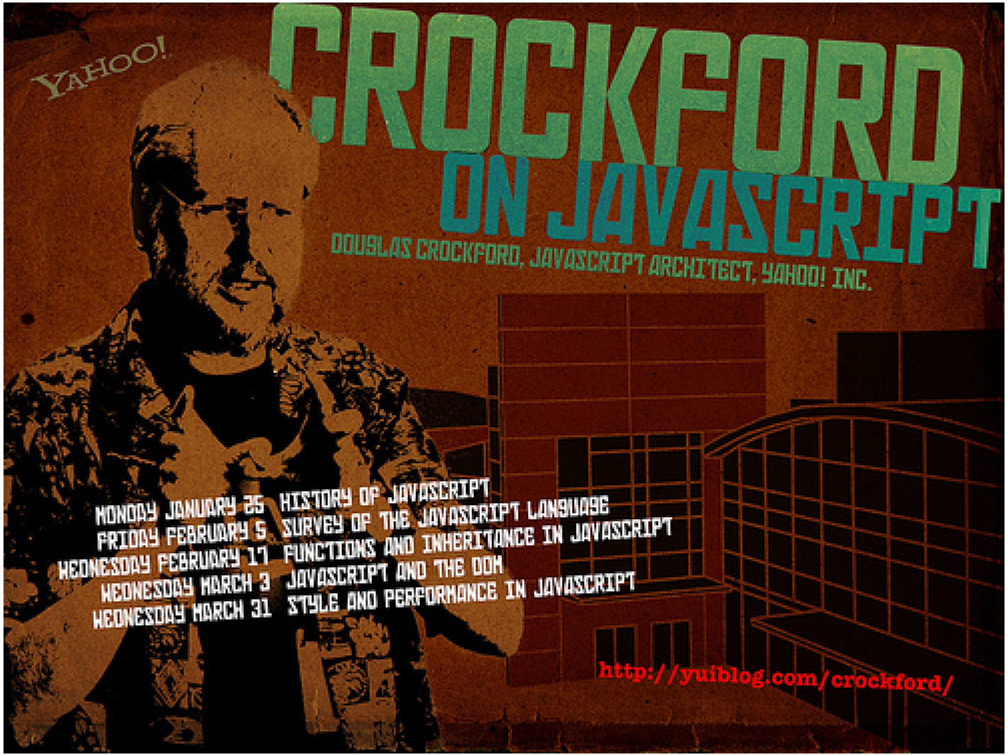
- 1998 — Outlook Web Access (OWA) project.

  > The IE project was just weeks away from beta 2 which was their last beta before the release. [...] I realized that the MSXML library shipped with IE and I had some good contacts over in the XML team who would probably help out- [we] struck a deal to ship the thing as part of the MSXML library.
  >
  > — https://web.archive.org/web/20160630074121/http://www.alexhopmann.com/xmlhttp.htm

Adoption

Browser	XHR added
Mozilla (rel. 2000)	2000 (compl. 2002)
Safari (rel. 2003)	2004
Opera	2005

Five years pass...



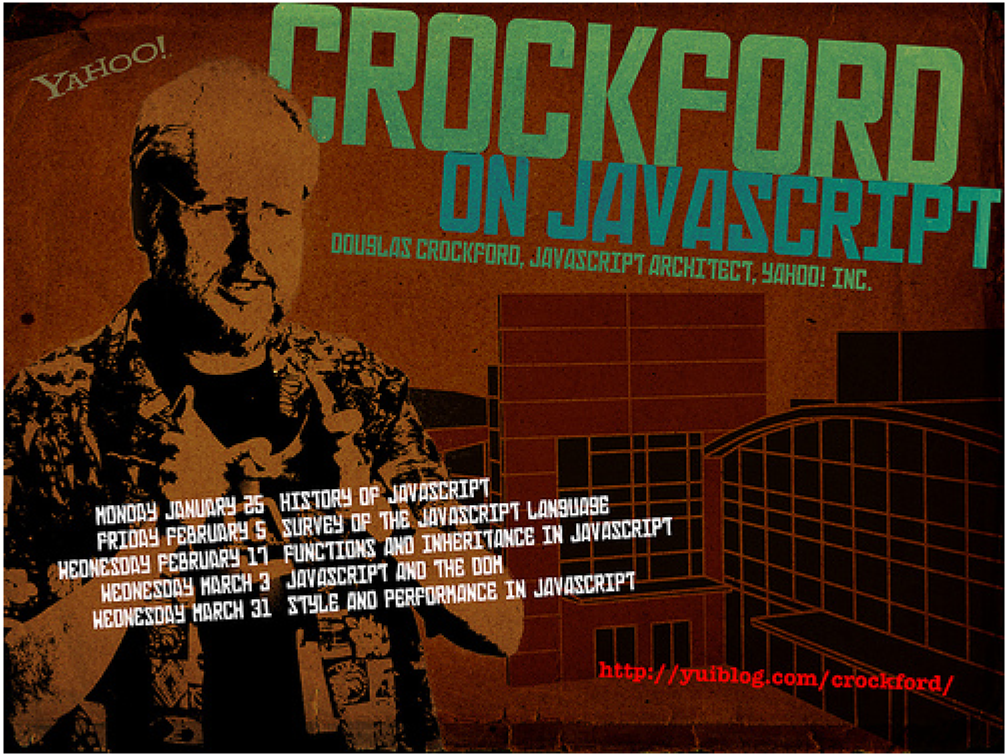
CROCKFORD ON JAVASCRIPT

DOUGLAS CROCKFORD, JAVASCRIPT ARCHITECT, YAHOO! INC.

MONDAY JANUARY 25 HISTORY OF JAVASCRIPT
FRIDAY FEBRUARY 5 SURVEY OF THE JAVASCRIPT LANGUAGE
WEDNESDAY FEBRUARY 17 FUNCTIONS AND INHERITANCE IN JAVASCRIPT
WEDNESDAY MARCH 3 JAVASCRIPT AND THE DOM
WEDNESDAY MARCH 31 STYLE AND PERFORMANCE IN JAVASCRIPT

http://yuiblog.com/crockford/

Five years pass...


YAHOO!
CROCKFORD
ON JAVASCRIPT
DOUGLAS CROCKFORD, JAVASCRIPT ARCHITECT, YAHOO! INC.
MONDAY JANUARY 25 HISTORY OF JAVASCRIPT
FRIDAY FEBRUARY 5 SURVEY OF THE JAVASCRIPT LANGUAGE
WEDNESDAY FEBRUARY 17 FUNCTIONS AND INHERITANCE IN JAVASCRIPT
WEDNESDAY MARCH 3 JAVASCRIPT AND THE DOM
WEDNESDAY MARCH 31 STYLE AND PERFORMANCE IN JAVASCRIPT
http://yuiblog.com/crockford/

Bugs, browser wars, Netscape dies, Microsoft wins.

JavaScript should have died with Netscape, but...

JavaScript should have died with Netscape, but...

- 2005 — "Ajax" (JavaScript, CSS, DOM, & XHR)

  > [A]pproach the "richness and responsiveness" of desktop applications.

Inevitable

> [...] these things take 3-5 years, so its not much of a surprise that the stuff that was developed incrementally between 1996 and 1998 actually started to hit it big in 2000-2002 and really exploded in 2005-2006.
>
> —
>
> https://web.archive.org/web/20160630074121/http://www.alexhopmann.com/xmlhttp.htm

Standardization

Year	Event
2006	W3C XMLHttpRequest
2006	jQuery released
2007	IE 7 (no ActiveX)
2008-2011	XHR2 (absorbed into original spec)

XHR capabilities

Basic usage

```
var oReq = new XMLHttpRequest();
oReq.addEventListener('load', function() {
    console.log(this.responseText);
});
oReq.open('GET', 'https://api.github.com/users');
oReq.send();
```

Common usage (callback)

```javascript
function xhr(method, path, data, headers, callback) {
    var req = new XMLHttpRequest(),
        default_headers = { /* ... */ };


    req.open(method.toUpperCase(), path, true);
    // For each: req.setRequestHeader(key, value);

    req.onreadystatechange = function(ev) {
        if (req.readyState !== 4) { return }

        if (req.status === 200) {
            callback(ev.target.response)
        } else {
            /* log error or whatevs */
        }
    };

    req.send(data);
}
```

Common usage (promise)

```
function xhr(method, path, data, headers) {
    var req = new XMLHttpRequest(),
        default_headers = { /* ... */ },
        deferred = Q.defer();

    req.open(method.toUpperCase(), path, true);
    // For each: req.setRequestHeader(key, value);

    req.onreadystatechange = function(ev) {
        if (req.readyState !== 4) { return }

        if (req.status === 200) {
            deferred.resolve(e.target.response);
        } else {
            deferred.reject(e.target.response);
        }
    };

    req.send(data);
    return deferred.promise;
}
```

Cancellable (abort)

```
var oReq = new XMLHttpRequest();
oReq.open('GET', 'https://httpbin.org/delay/1000');
oReq.onreadystatechange = console.log;
oReq.send();
setTimeout(() => oReq.abort(), 100);
```

Timeout

(FF implementation courtesy of our own Alex Vincent!)

```
var oReq = new XMLHttpRequest();
oReq.open('GET', 'https://httpbin.org/delay/5000');
oReq.onreadystatechange = console.log;
oReq.timeout = 1000;
oReq.send();
```

Progress

```
var oReq = new XMLHttpRequest();
oReq.open('GET', 'https://httpbin.org/drip');
oReq.onprogress = ev =>
    console.log('XXX', (ev.loaded / ev.total) * 100, '%')
oReq.send();
```

Stream

```javascript
var oReq = new XMLHttpRequest();
oReq.open('GET', 'https://httpbin.org/drip');
oReq.seenBytes = 0;
oReq.onreadystatechange = () => {
  if (oReq.readyState === 3) {
    oReq.seenBytes = oReq.responseText.length;
    console.log('seenBytes', oReq.seenBytes);
  }
};
oReq.send();
```

Tangent: Server-sent Events

Simple, one-directional stream.

Tangent: Server-sent Events

Simple, one-directional stream.

```python
# Python server
response.headers["Content-Type"] = "text/event-stream"
response.headers["Connection"] = "keep-alive"

def listen():
    events = get_events()
    yield str("retry: 400\n")

    while True:
        data = next(events)
        yield str("tag: {0}\n").format(data.get("tag", ""))
        yield str("data: {0}\n\n").format(json.dumps(data))
```

Tangent: Server-sent Events

Simple, one-directional stream.

```python
# Python server
response.headers["Content-Type"] = "text/event-stream"
response.headers["Connection"] = "keep-alive"

def listen():
    events = get_events()
    yield str("retry: 400\n")

    while True:
        data = next(events)
        yield str("tag: {0}\n").format(data.get("tag", ""))
        yield str("data: {0}\n\n").format(json.dumps(data))
```

```javascript
// JavaScript client
var stream = new EventSource('/stream');
stream.onmessage = function(ev) {
    console.log('XXX', ev.data)
}
```

XHR control structures

- Agnostic.
- Success callback, error callback, promise, stream, task, etc.

Fetch

Basic usage

Clean & simple API

```
fetch('https://api.github.com/users', { /* options */ })
    .then(response => response.json())
    .then(console.log);
```

Robust usage

```
fetch('https://api.github.com/users', { /* options */ })
    .then(response => {
        if (!response.ok) {
            throw new Error('Network response was not ok')
        }
        return response;
    })
    .then(response => {
        if (response.headers
                .get('content-type')
                .includes('application/json')) {
            return response.json();
        } else {
            return response.body();
        }
    })
    .then(console.log)
    .catch(console.error);  // beware: optional!
```

Fetch only implements a subset of XHR

Missing:

- Abort.
- Timeout.
- Progress.
- Stream (new in evergreens; no IE 11).
- Control flow agnosticism.

Promise-based

Inherits all the drawbacks of promises.
(See next section.)

Ajax libs are *always* wrapped

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
    - Add common request headers.

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
    - Add common request headers.
    - Send request as JSON.

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:
    (add auth header or opt-in to sending cookies).

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:
    (add auth header or opt-in to sending cookies).
  - Enable CORS.

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:
    (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
    - Add common request headers.
    - Send request as JSON.
    - Authenticate requests:
      (add auth header or opt-in to sending cookies).
    - Enable CORS.
    - Send XSRF token.
- Consistent response parsing:

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
    - Add common request headers.
    - Send request as JSON.
    - Authenticate requests:
      (add auth header or opt-in to sending cookies).
    - Enable CORS.
    - Send XSRF token.
- Consistent response parsing:
    - Parse as JSON.

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:

    (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.
- Consistent response parsing:
  - Parse as JSON.
  - Handle redirects:

    301 (Permanent), 302 (Temporary).

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:

    (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.
- Consistent response parsing:
  - Parse as JSON.
  - Handle redirects:

    301 (Permanent), 302 (Temporary).
  - Handle no-content responses:

    201 (Created), 202 (Accepted), 204 (No Content).

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:

    (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.
- Consistent response parsing:
  - Parse as JSON.
  - Handle redirects:

    301 (Permanent), 302 (Temporary).
  - Handle no-content responses:

    201 (Created), 202 (Accepted), 204 (No Content).
  - Handle authorization responses:

    401 (Unauthorized), 403 (Forbidden).

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:
    (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.
- Consistent response parsing:
  - Parse as JSON.
  - Handle redirects:
    301 (Permanent), 302 (Temporary).
  - Handle no-content responses:
    201 (Created), 202 (Accepted), 204 (No Content).
  - Handle authorization responses:
    401 (Unauthorized), 403 (Forbidden).
  - Success vs error responses:
    400 (Client Error), 404 (Not Found), 409 (Conflict),
    500 (Server Error), 502 (Gateway Unavailable), 503 (Service Unavailable).

Ajax libs are *always* wrapped

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests:

    (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.
- Consistent response parsing:
  - Parse as JSON.
  - Handle redirects:

    301 (Permanent), 302 (Temporary).
  - Handle no-content responses:

    201 (Created), 202 (Accepted), 204 (No Content).
  - Handle authorization responses:

    401 (Unauthorized), 403 (Forbidden).
  - Success vs error responses:

    400 (Client Error), 404 (Not Found), 409 (Conflict),
    500 (Server Error), 502 (Gateway Unavailable), 503 (Service Unavailable).
- Response caching & conditional-GET requests:

  If-None-Match/If-Modified-Since, 304 (Not Modified).

Wrap XHR or wrap fetch

```
import {myAjax} from 'utils/ajax';

myAjax(/* params */)
    .then(() => console.log(`Am I XHR or fetch?`));
```

Wrap XHR or wrap fetch

```
import {myAjax} from 'utils/ajax';

myAjax(/* params */)
    .then(() => console.log(`Am I XHR or fetch?`));
```

The underlying implementation *doesn't matter* if it suits your needs.
XHR, fetch, jQuery, Axios, Rx, etc.

Wrap XHR or wrap fetch

```
import {myAjax} from 'utils/ajax';

myAjax(/* params */)
    .then(() => console.log(`Am I XHR or fetch?`));
```

The underlying implementation *doesn't matter* if it suits your needs.
XHR, fetch, jQuery, Axios, Rx, etc.

Choose an API, choose a primitive, and wrap it.

Wrap XHR or wrap fetch

```
import {myAjax} from 'utils/ajax';

myAjax(/* params */)
    .then(() => console.log(`Am I XHR or fetch?`));
```

The underlying implementation *doesn't matter* if it suits your needs.
XHR, fetch, jQuery, Axios, Rx, etc.

Choose an API, choose a primitive, and wrap it.

...and the implementation should be changeable. This is the *Facade* pattern.

Promises

(Not an introduction or how-to.)

Before promises

Callbacks

Callbacks

Blocking

```
['foo', 'bar', 'baz'].map(x => x.toUpperCase());
```

Implemented via pointers (C) or first-class functions.

Callbacks

Blocking

```
['foo', 'bar', 'baz'].map(x => x.toUpperCase());
```

Implemented via pointers (C) or first-class functions.

Deferred

```
setTimeout(() => console.log('Call me'), 1000);
```

Run function on a different thread or processor (or queue).

Callbacks

Blocking

```
['foo', 'bar', 'baz'].map(x => x.toUpperCase());
```

Implemented via pointers (C) or first-class functions.

Deferred

```
setTimeout(() => console.log('Call me'), 1000);
```

Run function on a different thread or processor (or queue).

...we're only talking about deferred callbacks here.

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.
- Program control flow is explicit — a continuation.

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.
- Program control flow is explicit — a continuation.
- Callbacks are a *small subset* of true CPS.

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.
- Program control flow is explicit — a continuation.
- Callbacks are a *small subset* of true CPS.
- call/cc

  `race()`/`amb()`, backtracking, coroutines, generators, engines, threads, try/catch.

  Breaking a long computation into chunks.

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.
- Program control flow is explicit — a continuation.
- Callbacks are a *small subset* of true CPS.
- call/cc

  `race()`/`amb()`, backtracking, coroutines, generators, engines, threads, try/catch.

  Breaking a long computation into chunks.
- Hard to write/maintain.

  (Often a compile target, esp for langs with TCO.)

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.
- Program control flow is explicit — a continuation.
- Callbacks are a *small subset* of true CPS.
- call/cc

  `race()`/`amb()`, backtracking, coroutines, generators, engines, threads, try/catch.

  Breaking a long computation into chunks.
- Hard to write/maintain.

  (Often a compile target, esp for langs with TCO.)
- 1976/1977 — Promises/futures first described.
- 1988 — Promise pipelines first described.

Continuation passing style (CPS)

- 1975 — first implemented in Scheme.
- Program control flow is explicit — a continuation.
- Callbacks are a *small subset* of true CPS.
- call/cc

  `race()`/`amb()`, backtracking, coroutines, generators, engines, threads, try/catch.

  Breaking a long computation into chunks.
- Hard to write/maintain.

  (Often a compile target, esp for langs with TCO.)
- 1976/1977 — Promises/futures first described.
- 1988 — Promise pipelines first described.
- 1980-1990s — Monads from mathematics first linked to programming.

Tangent: Lisp/Scheme and ML languages

Tangent: Lisp/Scheme and ML languages

- Entreaties: these concepts have merit!

Tangent: Lisp/Scheme and ML languages

- Entreaties: these concepts have merit!
- First class functions
- Higher order functions
- Principled function composition.

Tangent: Lisp/Scheme and ML languages

- Entreaties: these concepts have merit!
- First class functions
- Higher order functions
- Principled function composition.
- All things we take for granted in JS today.

  (Hopefully TCO soon as well.)

Then Came Promises

Pyramid of doom (callback hell)

Pyramid of doom (callback hell)

Problem

```
doSomething(function(result) {
    doSomethingElse(result, function(newResult) {
        doThirdThing(newResult, function(finalResult) {
            console.log('Got the final result: ' + finalResult);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Pyramid of doom (callback hell)

Problem

```javascript
doSomething(function(result) {
    doSomethingElse(result, function(newResult) {
        doThirdThing(newResult, function(finalResult) {
            console.log('Got the final result: ' + finalResult);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Solution

```javascript
doSomething()
    .then(result => doSomethingElse(result))
    .then(newResult => doThirdThing(newResult))
    .then(finalResult => {
        console.log(`Got the final result: ${finalResult}`);
    })
    .catch(failureCallback);
```

Three states

Initialized

```
const myPromise = new Promise((resolve, reject) => {
    /* snip */
});
```

Three states

Initialized

```
const myPromise = new Promise((resolve, reject) => {
    /* snip */
});
```

Resolved

```
myPromise.then(/* snip */);
```

Three states

Initialized

```
const myPromise = new Promise((resolve, reject) => {
    /* snip */
});
```

Resolved

```
myPromise.then(/* snip */);
```

Rejected

```
myPromise.catch(/* snip */);
```

Three states

Initialized

```
const myPromise = new Promise((resolve, reject) => {
    /* snip */
});
```

Resolved

```
myPromise.then(/* snip */);
```

Rejected

```
myPromise.catch(/* snip */);
```

(*No* visibility and *little* control over the current state.)

Caching *and* control flow

```
const myUsers = fetch('https://api.github.com/users');
myUsers.then(console.log);
myUsers.then(console.log);
myUsers.then(console.log);
```

Caching *and* control flow

```
const myUsers = fetch('https://api.github.com/users');
myUsers.then(console.log);
myUsers.then(console.log);
myUsers.then(console.log);
```

(*No* visibility and *no* control over the cache.)

Always async

...Even if already resolved. For consistency.

Tangent: Various async queues in JS

Tangent: Various async queues in JS

Task
A queue of things to run in each turn of the event loop.

Tangent: Various async queues in JS

Task
A queue of things to run in each turn of the event loop.

Microtask
A queue of things to run during a single task.

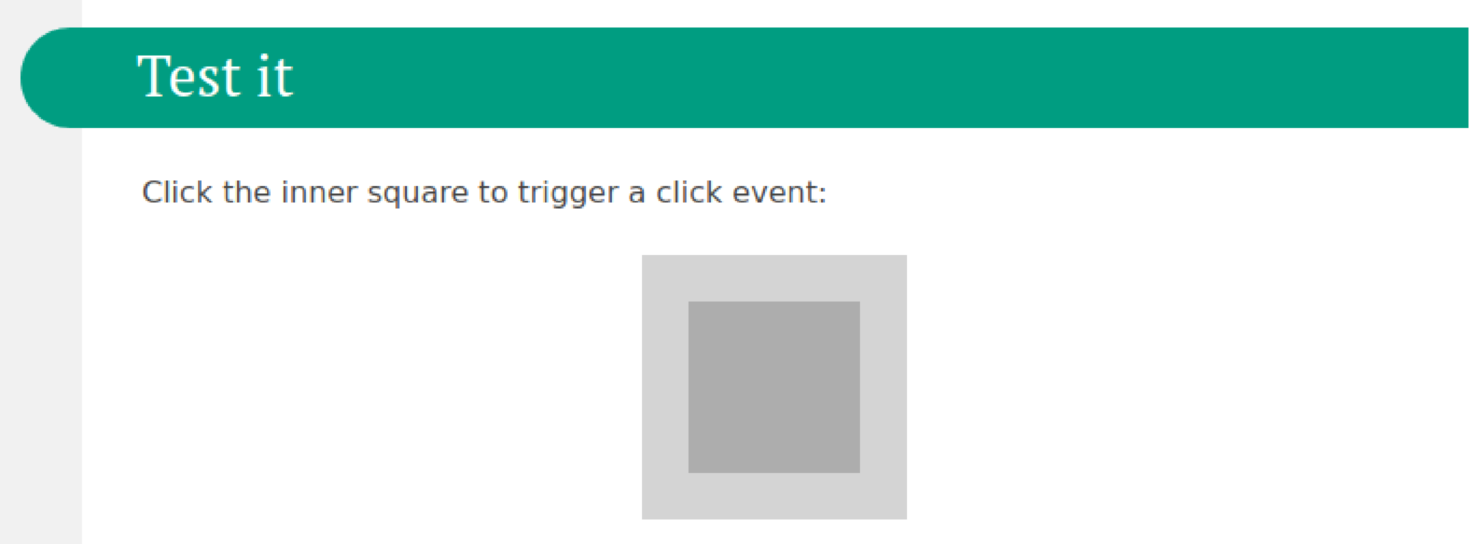
Tangent: Various async queues in JS

Task
A queue of things to run in each turn of the event loop.

Microtask
A queue of things to run during a single task.

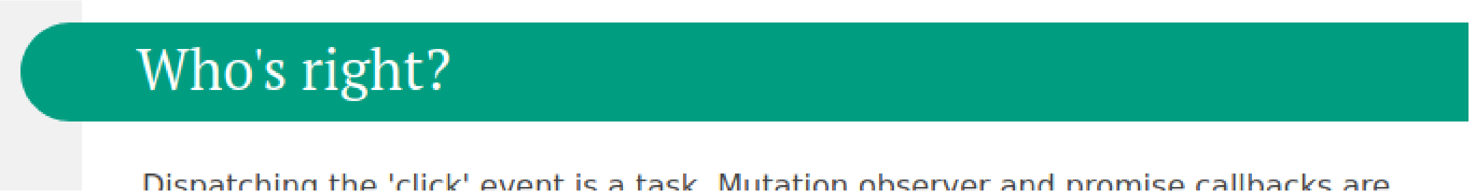
requestAnimationFrame
A queue of things to run at an optimal time for rendering performance.

Tangent: Browser async implementations



Test it

Click the inner square to trigger a click event:



Who's right?

Dispatching the 'click' event is a task. Mutation observer and promise callbacks are

Promises, a missed opportunity

Many, many variants of async control structures

Promises, deferreds, tasks, futures, reactive streams, FRP streams, callbags.

Existing library ecosystem

- q
  (progress, handle unhandled errors, introspection of current state)

Existing library ecosystem

- q

  (progress, handle unhandled errors, introspection of current state)
- bluebird

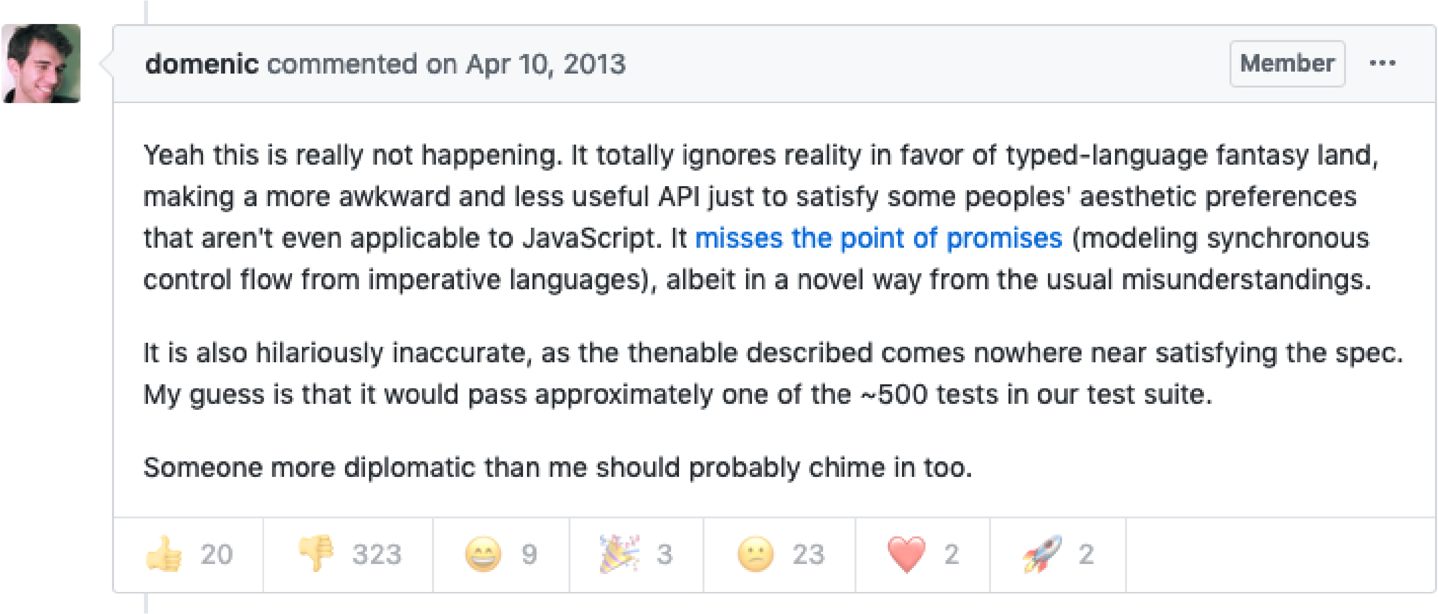
  (progress, cancelation)

Existing library ecosystem

- q
  (progress, handle unhandled errors, introspection of current state)
- bluebird
  (progress, cancelation)
- when
  (by the cujojs folk; lift, join, spread, fold, finally, else, tap, delay, timeout, inspect, progress, map, filter, reduce, and more)

The infamous GitHub discussion

The infamous GitHub discussion



domenic commented on Apr 10, 2013

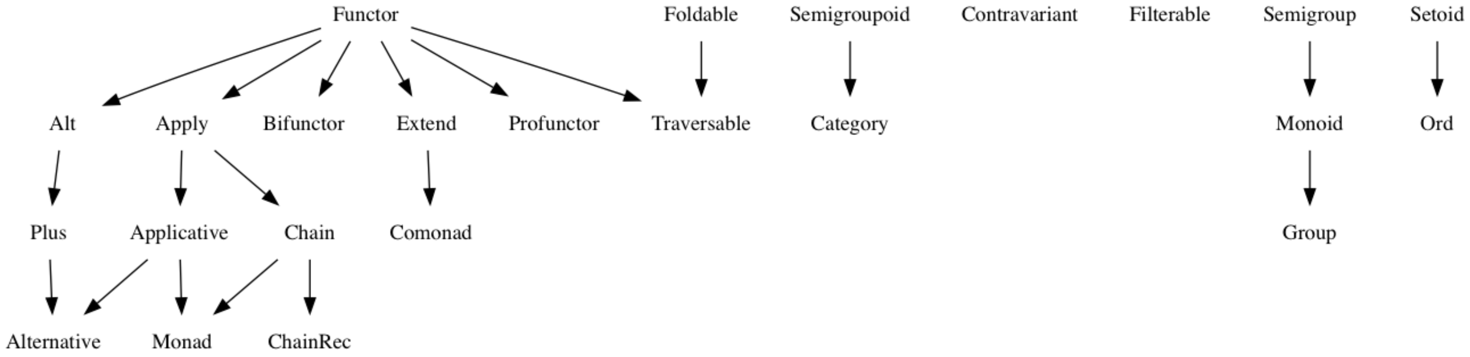
Yeah this is really not happening. It totally ignores reality in favor of typed-language fantasy land, making a more awkward and less useful API just to satisfy some peoples' aesthetic preferences that aren't even applicable to JavaScript. It misses the point of promises (modeling synchronous control flow from imperative languages), albeit in a novel way from the usual misunderstandings.

It is also hilariously inaccurate, as the thenable described comes nowhere near satisfying the spec. My guess is that it would pass approximately one of the ~500 tests in our test suite.

Someone more diplomatic than me should probably chime in too.

Fantasy Land Specification

(aka "Algebraic JavaScript Specification")



Function composition (got it right!)

```
const capitalize = x => x.toUpperCase();
const exclaim = x => `${x}!`;

Promise.resolve('foo')
    .then(capitalize)
    .then(exclaim)
    .then(console.log)
// => FOO!
```

Function composition (got it right!)

```javascript
const capitalize = x => x.toUpperCase();
const exclaim = x => `${x}!`;

Promise.resolve('foo')
    .then(capitalize)
    .then(exclaim)
    .then(console.log)
// => FOO!
```

Equivalent to!

```javascript
const compose = (f, g) => (...args) => f(g(...args));

Promise.resolve('bar')
    .then(compose(exclaim, capitalize))
    .then(console.log);
// => BAR!
```

Combinators (got it wrong)

Only two composition operators:

- `Promise.all()`
- `Promise.race()`

(Possible to add more but limited by the next two problems.)

Auto-flattening (got it wrong)

No inner-promises — prevents generating promises for composition.

```
const makeTimer = time => new Promise(res => setTimeout(res, time))

console.time('Want one second')
Promise.resolve(null)
    .then(() => makeTimer(1000))
    .then(generatedTimer =>
        // No way to both generate and parallelize.
        Promise.all([
            generatedTimer,
            makeTimer(1000),
        ]))
    .then(() => console.timeEnd('Want one second'))
```

Eager execution (got it wrong)

Composition requires absolute knowledge of when the promise was initialized.

```
const delay1sec = new Promise(res => setTimeout(res, 1000));

// Does not necessarily wait for 1 second.
// Depends entirely on when/where delay1sec was initialized.
Promise.all([
    delay1sec,
    Promise.resolve('foo'),
])
.then(console.log)
```

Tasks (implementation)

```
const compose = (f, g) => (...args) => f(g(...args));

class Task {
    constructor(fork) { this._fork = fork }

    fork(rej, res) {
        try { return this._fork(rej, res) }
        catch (e) { return rej(e) }
    }
    map(f) { return new Task((rej, res) =>
        this.fork(rej, compose(res, f))) }
    chain(f) { return new Task((rej,res) =>
        this.fork(rej, x => f(x).fork(rej, res))) }

    static of(x) { return new Task((rej, res) => res(x)) }
}
```

Tasks (basic use; sequential flattening)

```
const makeTimer = time =>
    new Task((rej, res) => setTimeout(res, time))

console.time('XXX')
makeTimer(1000)
    .chain(() => makeTimer(1000))
    .fork(console.error, () => console.timeEnd('XXX'))
```

Tasks (generated; no flattening)

```
const makeTimer = time =>
    new Task((rej, res) => setTimeout(res, time))

console.time('XXX')
Task.of(null)
    .map(() => [1000, 1000, 1000].map(x => makeTimer(x)))
    .chain(timerArray => timerArray.reduce(
        (ts, t) => ts.chain(() => t),
        Task.of(null)))
    .fork(console.error, () => console.timeEnd('XXX'))
```

Control structure for request/response

Possible states of a request/response cycle:

1. Not requested.
2. In-flight.
3. One or more successes, or error.

Tangent: Streams

- Rx

  Highly composeable, caching & control flow & granular control over both, sync or async & granular control over sync/task/microtask/animationframe/virtual queues, initialize, resolve (zero or more times), reject, cancel, retry, progress, timeout.

Tangent: Streams

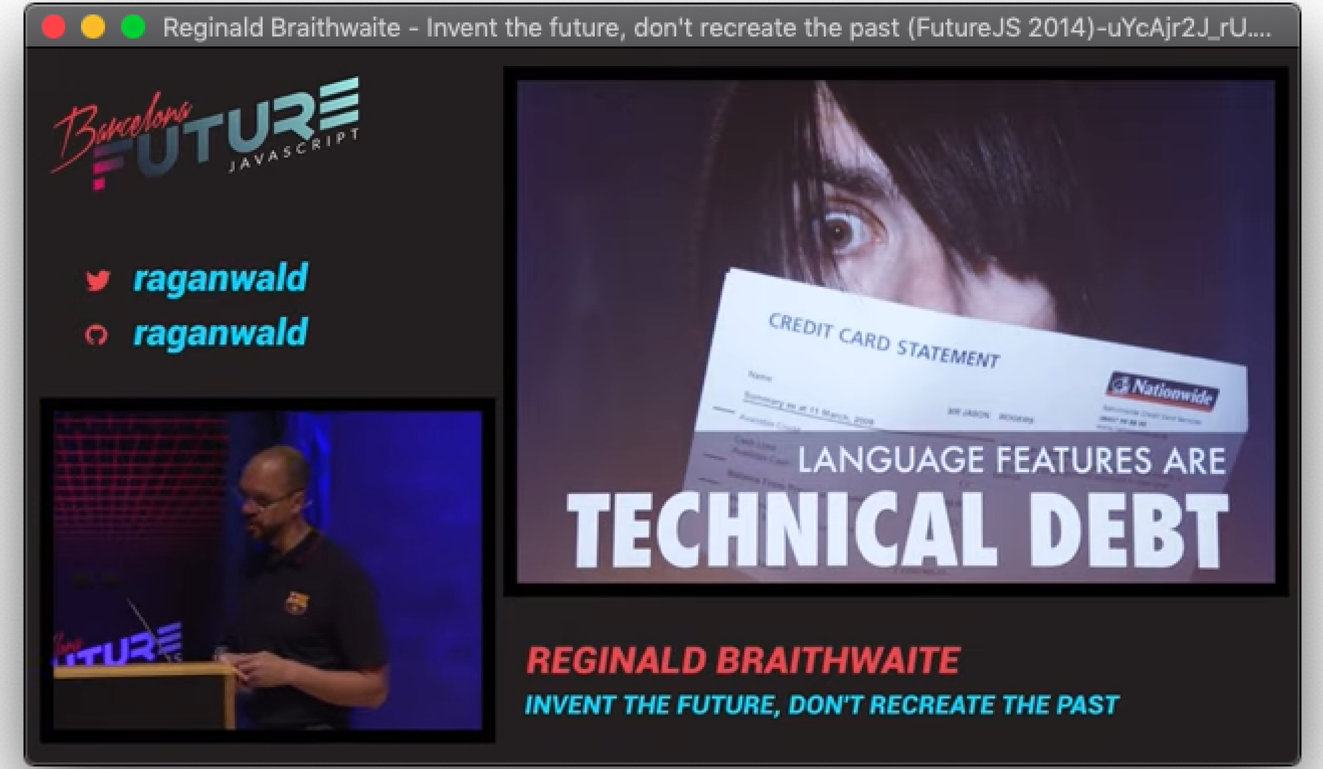
- Rx

  Highly composeable, caching & control flow & granular control over both, sync or async & granular control over sync/task/microtask/animationframe/virtual queues, initialize, resolve (zero or more times), reject, cancel, retry, progress, timeout.
- The core *idea* defines what is possible, not the implementation.

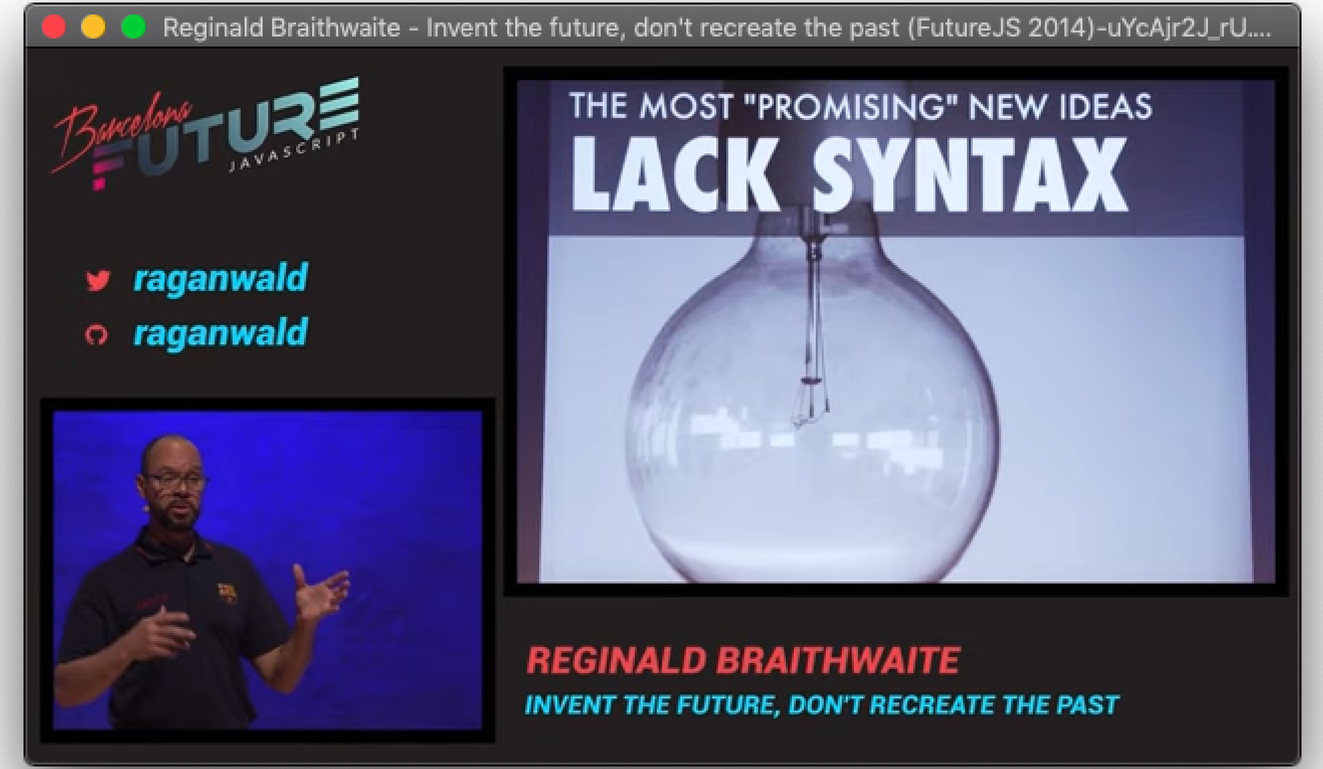
  Most, XStream, Callbags, Flyd

The Future

Language features vs. user space


Reginald Braithwaite - Invent the future, don't recreate the past (FutureJS 2014)-uYcAjr2J_rU....

Syntax vs. language maintenance



Reginald Braithwaite - Invent the future, don't recreate the past (FutureJS 2014)-uYcAjr2J_rU....

Barcelona FUTURE JAVASCRIPT
raganwald
raganwald
THE MOST "PROMISING" NEW IDEAS
LACK SYNTAX
REGINALD BRAITHWAITE
INVENT THE FUTURE, DON'T RECREATE THE PAST

Async/await

Inherits all the downsides of promises...because it *is* promises

Debate: is "hiding" async a valuable end-goal?

Debate: Crockford, Raganwald — keep JS small