Promises, Promises

BY @DOMENIC

Domenic Denicola

- http://domenic.me
- https://github.com/domenic
- https://npmjs.org/~domenic
- http://slideshare.net/domenicdenicola

Things I'm doing:

- @esdiscuss on Twitter
- The Promises/A+ spec
- HTML5DevConf, NodePDX, NodeConf

The Promise Abstraction

Callbacks are a hack

- They are literally the simplest thing that could work.
- But as a replacement for synchronous control flow, they suck.

- There’s no consistency in callback APIs.
- There’s no guarantees.
- We lose the flow of our code writing callbacks that tie together other callbacks.
- We lose the stack-unwinding semantics of exceptions, forcing us to handle errors explicitly at every step.

Promises are the right abstraction

Instead of calling a passed callback, return a promise:

```
readFile("file.txt", function (err, result) {
    // continue here…
});

// becomes

var promiseForResult = readFile("file.txt");
```

Promise guarantees

```
promiseForResult.then(onFulfilled, onRejected);
```

- Only one of **onFulfilled** or **onRejected** will be called.
- **onFulfilled** will be called with a single fulfillment value (⇔ return value).
- **onRejected** will be called with a single rejection reason (⇔ thrown exception).
- If the promise is already settled, the handlers will still be called once you attach them.
- The handlers will always be called asynchronously.

Promises can be chained

```
var transformedPromise = originalPromise.then(onFulfilled, onRejected);
```

- If the called handler returns a value, **transformedPromise** will be *resolved* with that value:
  - If the returned value is a promise, we adopt its state.
  - Otherwise, **transformedPromise** is fulfilled with that value.
- If the called handler throws an exception, **transformedPromise** will be rejected with that exception.

The Sync ⇔ Async Parallel

```
var result, threw = false;

try {
    result = doSomethingSync();
} catch (ex) {
    threw = true;
    handle(ex);
}

if (!threw) process(result);
```

```
doSomethingAsync().then(
    process,
    handle
);
```

Case 1: Simple Functional Transform

```
var user = getUser();
var userName = user.name;

// becomes

var userNamePromise = getUser().then(function (user) {
    return user.name;
});
```

Case 2: Reacting with an Exception

```
var user = getUser();
if (user === null)
    throw new Error("null user!");
```

becomes

```
var userPromise = getUser().then(function (user) {
    if (user === null)
        throw new Error("null user!");
    return user;
});
```

Case 3: Handling an Exception

```
try {
    updateUser(data);
} catch (ex) {
    console.log("There was an error:", ex);
}

// becomes

var updatePromise = updateUser(data).then(undefined, function (ex) {
    console.log("There was an error:", ex);
});
```

Case 4: Rethrowing an Exception

```
try {
    updateUser(data);
} catch (ex) {
    throw new Error("Updating user failed. Details: " + ex.message);
}

// becomes

var updatePromise = updateUser(data).then(undefined, function (ex) {
    throw new Error("Updating user failed. Details: " + ex.message);
});
```

Bonus Async Case: Waiting

```
var name = promptForNewUserName();
updateUser({ name: name });
refreshUI();

// becomes

promptForNewUserName()
    .then(function (name) {
        return updateUser({ name: name });
    })
    .then(refreshUI);
```

Promises Give You Back Exception Propagation

```
getUser("Domenic", function (user) {
    getBestFriend(user, function (friend) {
        ui.showBestFriend(friend);
    });
});
```

Promises Give You Back Exception Propagation

```
getUser("Domenic", function (err, user) {
    if (err) {
        ui.error(err);
    } else {
        getBestFriend(user, function (err, friend) {
            if (err) {
                ui.error(err);
            } else {
                ui.showBestFriend(friend);
            }
        });
    }
});
```

Promises Give You Back Exception Propagation

```
getUser("Domenic")
    .then(getBestFriend)
    .then(ui.showBestFriend, ui.error);
```

Promises as First-Class Objects

- Because promises are first-class objects, you can build simple operations on them instead of tying callbacks together:

```
// Fulfills with an array of results, or rejects if any reject
all([getUserData(), getCompanyData()]);

// Fulfills as soon as either completes, or rejects if both reject
any([storeDataOnServer1(), storeDataOnServer2()]);

// If writeFile accepts promises as arguments, and readFile returns one:
writeFile("dest.txt", readFile("source.txt"));
```

The Promises/A+ Story

@DOMENIC

Prehistory

- “Discovered” circa 1989.
- Much of modern promises are inspired by the E programming language.
- They’ve made their way into many languages:
  - .NET’s Task<T>
  - java.util.concurrent.Future
  - Python’s PEP 3148
  - C++ 11’s std::future

CommonJS Promises/A

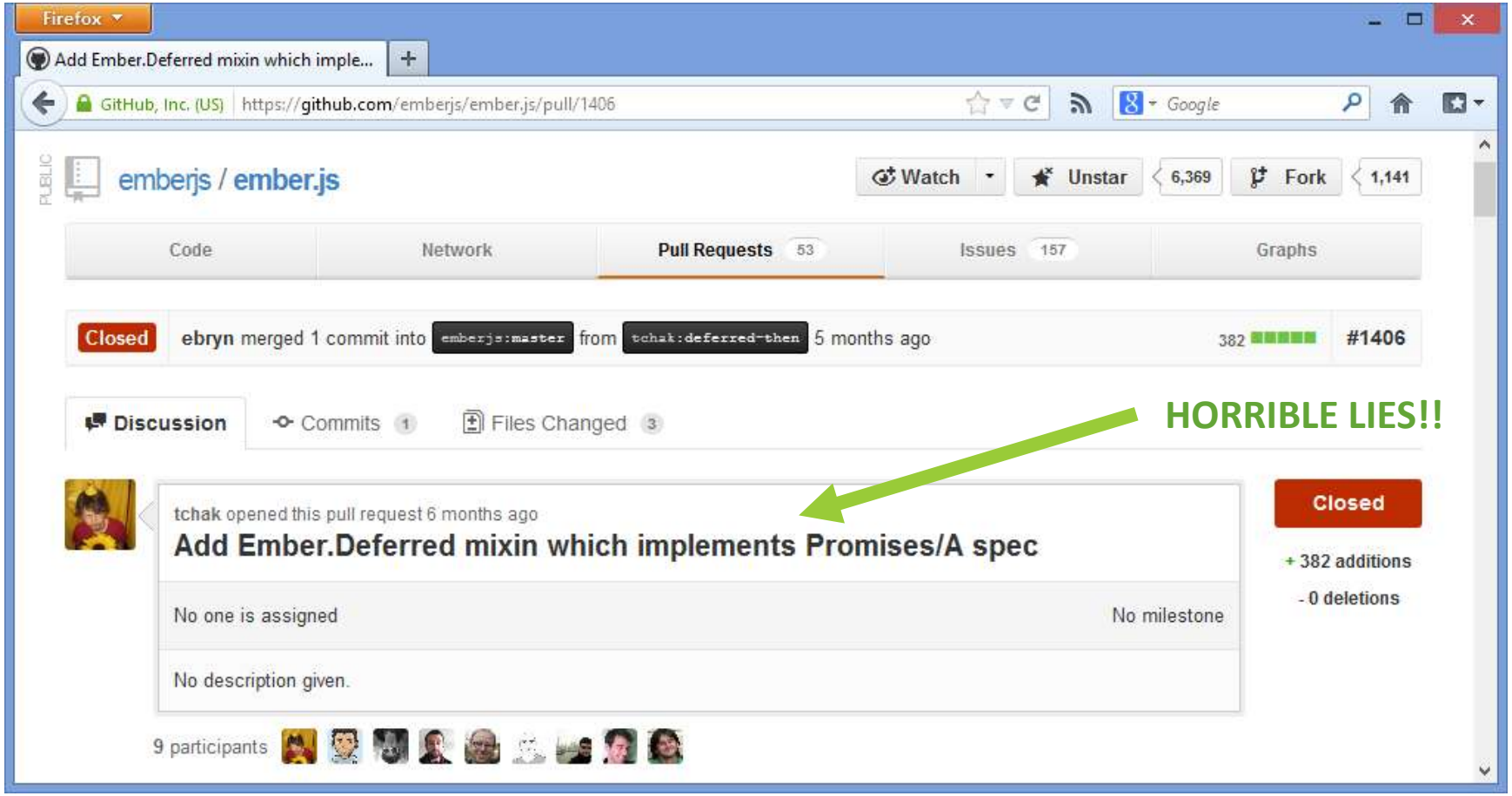
- Inspired by early implementations: ref_send, Dojo, …
- But…
  - Underspecified
  - Missing key features
  - Often misinterpreted

$.Deferred

jQuery's $.Deferred is a very buggy attempted implementation, that entirely misses the sync ⇔ async parallel:

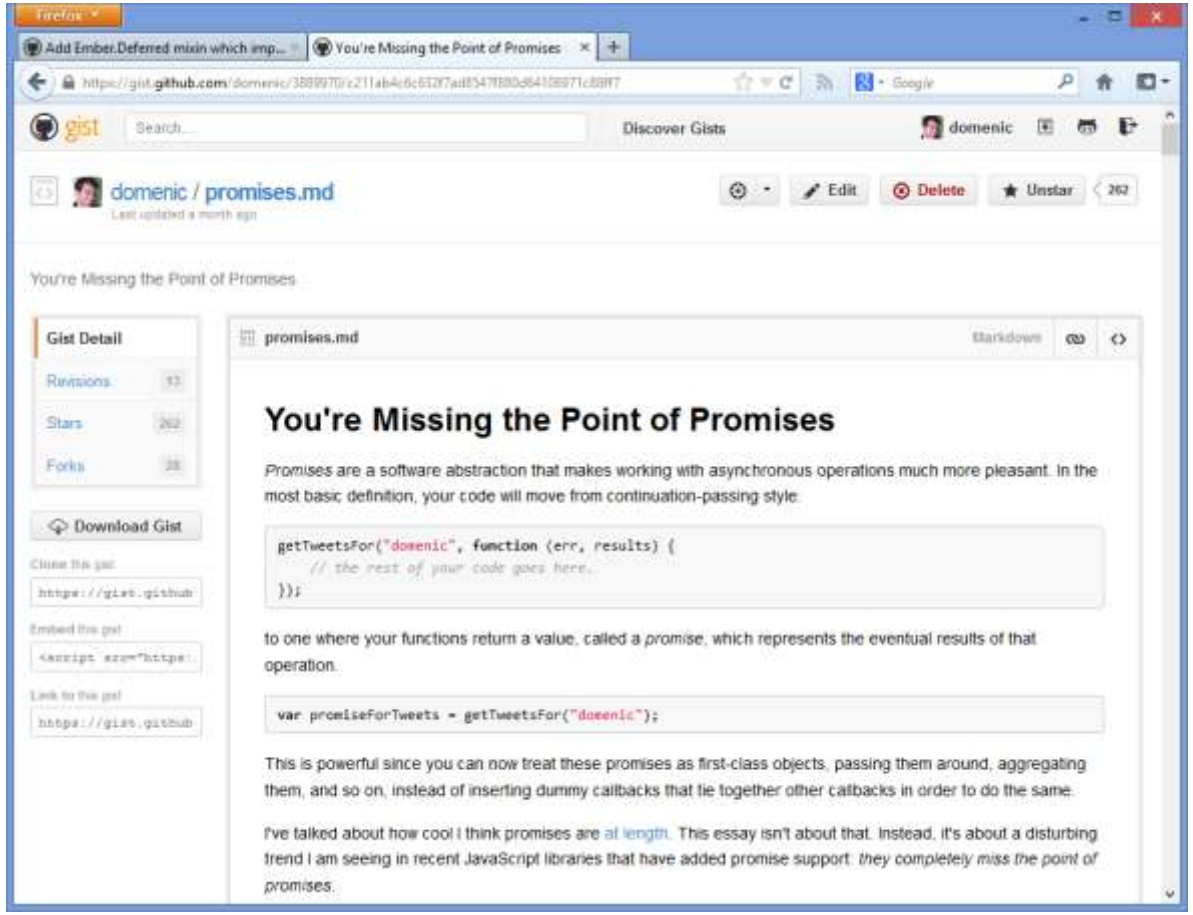
- Multiple fulfillment values and rejection reasons
- Only supports scenario 1 (functional transformation); doesn't handle errors
- Not interoperable with other "thenables."
- Before 1.8, did not support returning a promise

All Was Quiet, Until…



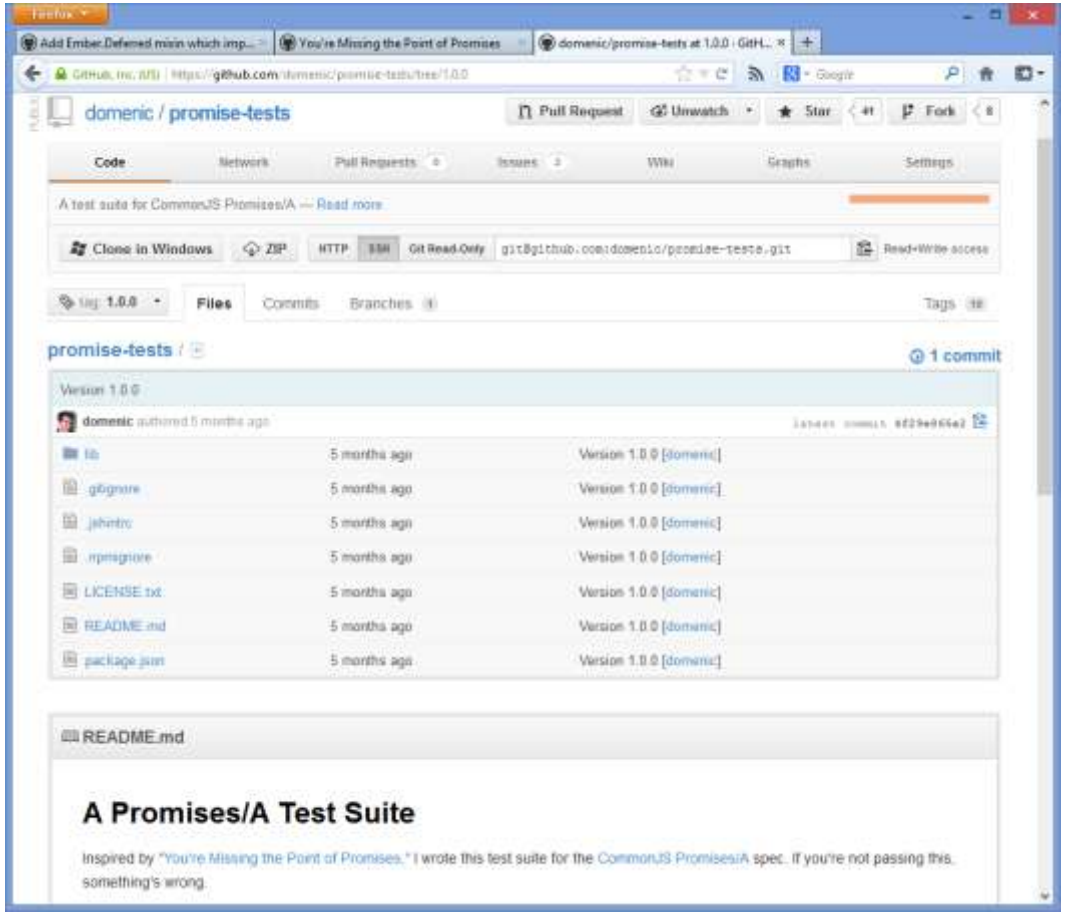
@DOMENIC

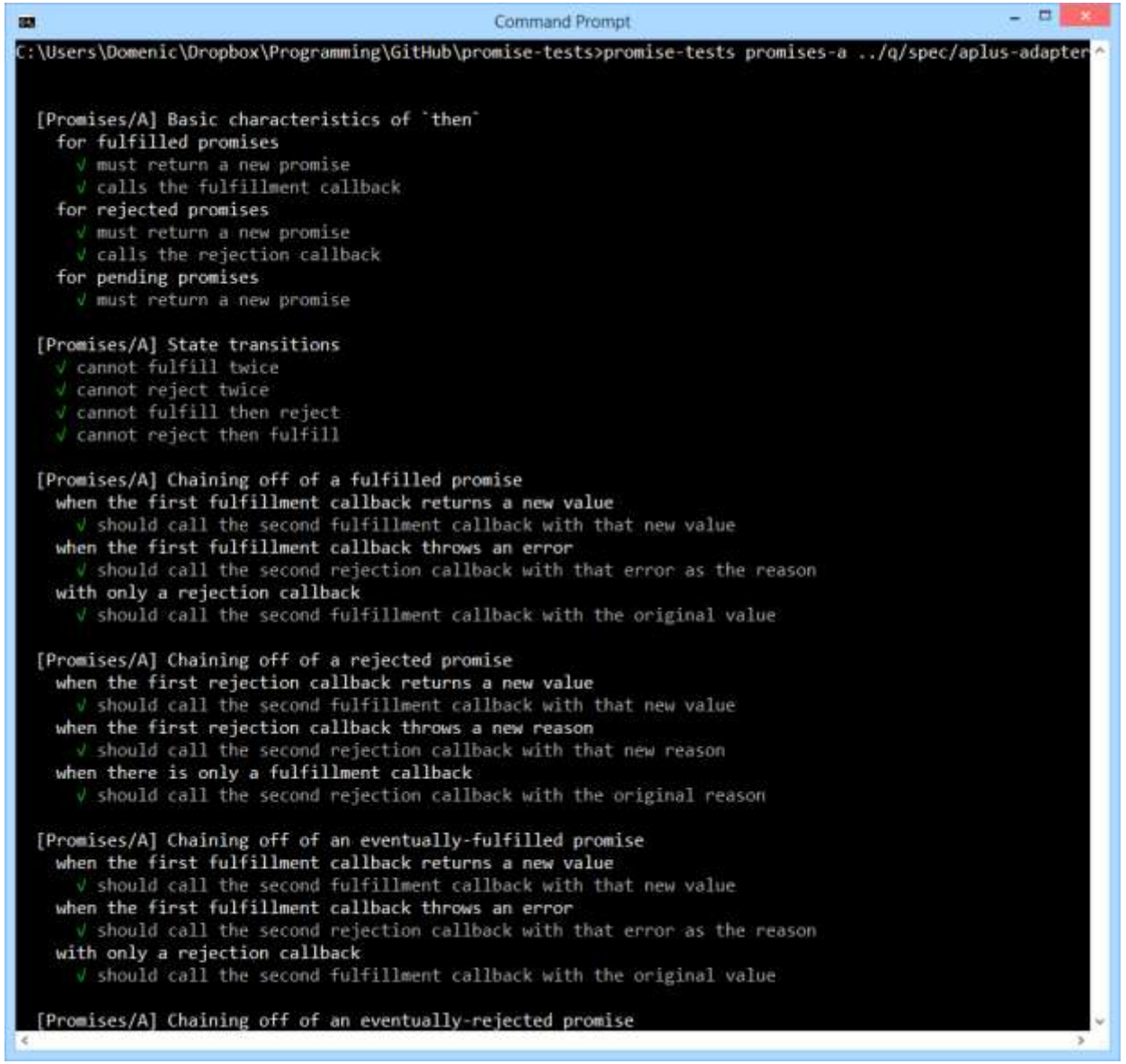
I Got Angry



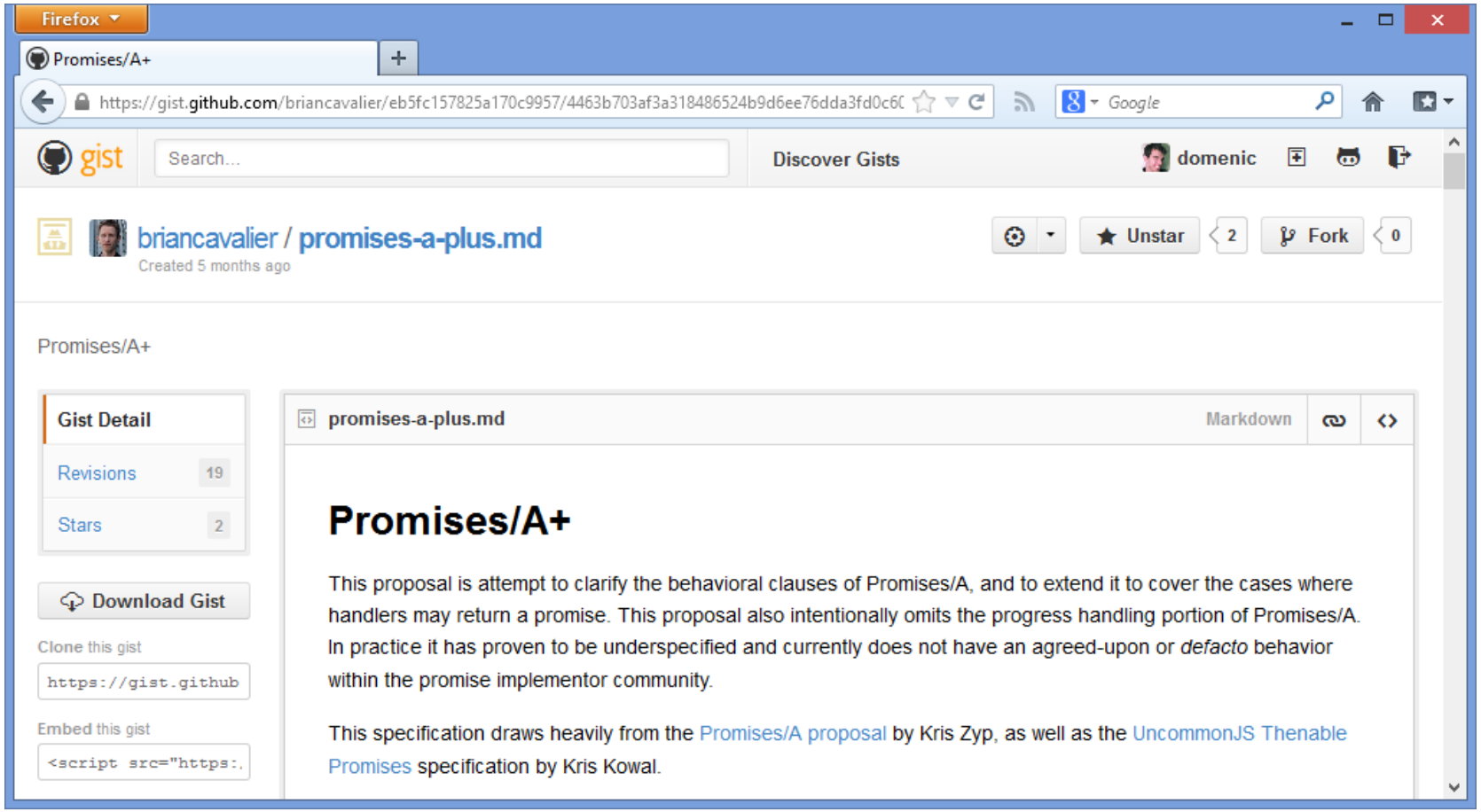
@DOMENIC

Then I Did Something About It



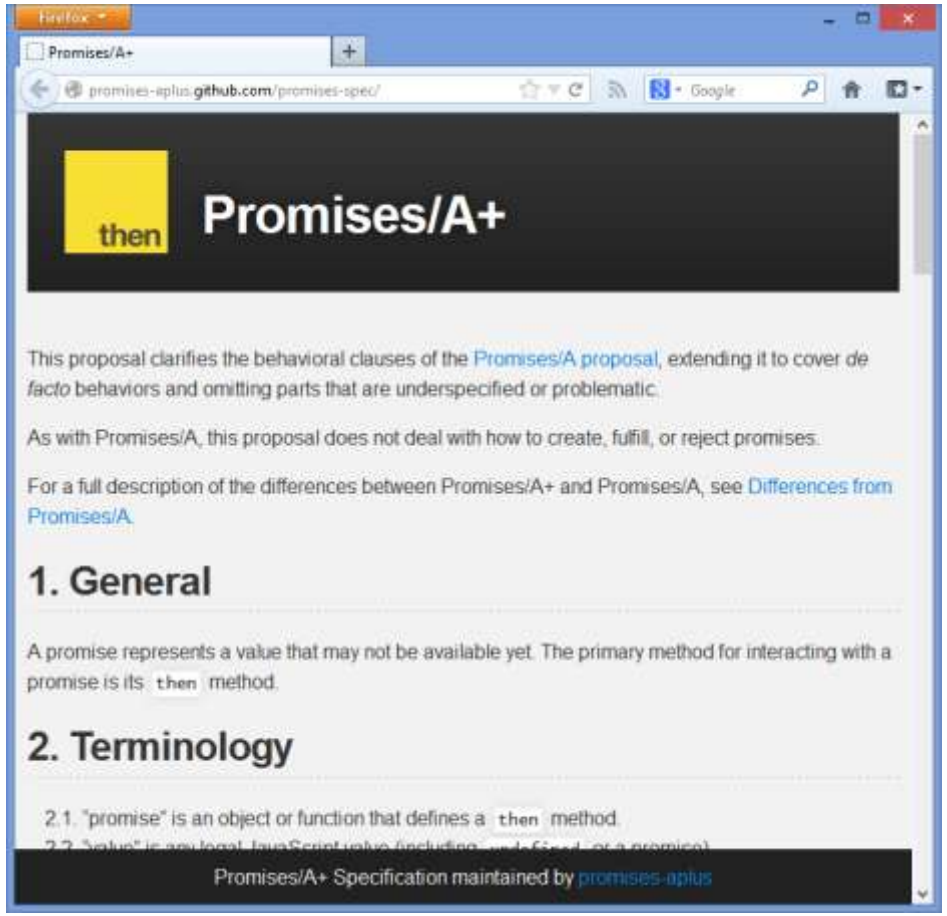

Command Prompt
C:\Users\Domenic\Dropbox\Programming\GitHub\promise-tests>promise-tests promises-a ../q/spec/aplus-adapter
[Promises/A] Basic characteristics of `then`
for fulfilled promises
√ must return a new promise
√ calls the fulfillment callback
for rejected promises
√ must return a new promise
√ calls the rejection callback
for pending promises
√ must return a new promise
[Promises/A] State transitions
√ cannot fulfill twice
√ cannot reject twice
√ cannot fulfill then reject
√ cannot reject then fulfill
[Promises/A] Chaining off of a fulfilled promise
when the first fulfillment callback returns a new value
√ should call the second fulfillment callback with that new value
when the first fulfillment callback throws an error
√ should call the second rejection callback with that error as the reason
with only a rejection callback
√ should call the second fulfillment callback with the original value
[Promises/A] Chaining off of a rejected promise
when the first rejection callback returns a new value
√ should call the second fulfillment callback with that new value
when the first rejection callback throws a new reason
√ should call the second rejection callback with that new reason
when there is only a fulfillment callback
√ should call the second rejection callback with the original reason
[Promises/A] Chaining off of an eventually-fulfilled promise
when the first fulfillment callback returns a new value
√ should call the second fulfillment callback with that new value
when the first fulfillment callback throws an error
√ should call the second rejection callback with that error as the reason
with only a rejection callback
√ should call the second fulfillment callback with the original value
[Promises/A] Chaining off of an eventually-rejected promise

Then Things Got Awesome



@DOMENIC

Fast-Forward a Few Months…



@DOMENIC

I Think It's Been a Success

- >20 conformant implementations, with more showing up constantly
  - Even one in ActionScript 3!
- The creation of RSVP.js specifically so that Ember could have Promises/A+ compatible promises
- Version 1.1 of the spec almost ready, nailing down some unspecified points
- Several other sibling specs under active development: promise creation, cancellation, progress, …

Even the DOM and TC39 are getting in on this

- Alex Russell's DOMFuture promise library, for possibly using promises in future or existing DOM APIs
- Convergence with Mark Miller's concurrency strawman, for integrating promises into the language

Promises in Your Code

Some practical guidance

@DOMENIC

First, Choose a Library

- My top picks:
  - Q, by Kris Kowal and myself: https://github.com/kriskowal/q
  - When.js, by Brian Cavalier: https://github.com/cujojs/when
  - RSVP.js, by Yehuda Katz: https://github.com/tildeio/rsvp.js

- If you ever see a jQuery promise, kill it with fire:

```
var realPromise = Q(jQueryPromise);
var realPromise = when(jQueryPromise);
```

Keep The Sync ⇔ Async Parallel In Mind

- Use promises for single operations that can result in fulfillment (⇔ returning a value) or rejection (⇔ throwing an exception).

- If you're ever stuck, ask "how would I structure this code if it were synchronous?"
  - The only exception is multiple parallel operations, which has no sync counterpart.

Promises Are *Not*

- A replacement for events
- A replacement for streams
- A way of doing functional reactive programming

They work together:

- An event can trigger from one part of your UI, causing the event handler to *trigger a promise-returning function*
- A HTTP request function can return *a promise for a stream*

The Unhandled Rejection Pitfall

This hits the top of the stack:

```
throw new Error("boo!");
```

This stays inert:

```
var promise = doSomething().then(function () {
    throw new Error("boo!");
});
```

Avoiding the Unhandled Rejection Pitfall

- *Always* either:
  - **return** the promise to your caller;
  - or call **.done()** on it to signal that any unhandled rejections should explode

```
function getUserName() {
    return getUser().then(function (user) {
      return user.name;
    });
}

getUserName().then(function (userName) {
    console.log("User name: ", userName);
}).done();
```

Promise Patterns: try/catch/finally

```
ui.startSpinner();
getUser("Domenic")
    .then(getBestFriend)
    .then(ui.showBestFriend)
    .catch(ui.error)
    .finally(ui.stopSpinner)
    .done();
```

Promise Patterns: all + spread

```
Q.all([getUser(), getCompany()]).then(function (results) {
    console.log("user = ", results[0]);
    console.log("company = ", results[1]);
}).done();

Q.all([getUser(), getCompany()]).spread(function (user, company) {
    console.log("user = ", user);
    console.log("company = ", company);
}).done();
```

Promise Patterns: map + all

```
var userIds = ["123", "456", "789"];

Q.all(userIds.map(getUserById))
    .then(function (users) {
        console.log("all the users: ", users);
    })
    .done();
```

Promise Patterns: message sending

```
var userData = getUserData();

userData
    .then(createUserViewModel)
    .invoke("setStatus", "loaded")
    .done();

userData
    .get("friends")
    .get("0")
    .get("name")
    .then(setBestFriendsNameInUI)
    .done();
```

Promise Patterns: Denodeify

```
var readFile = Q.denodeify(fs.readFile);
var readDir = Q.denodeify(fs.readdir);

readDir("/tmp")
    .get("0")
    .then(readFile)
    .then(function (data) {
        console.log("The first temporary file contains: ", data);
    })
    .catch(function (error) {
        console.log("One of the steps failed: ", error);
    })
    .done();
```

@DOMENIC

Advanced Promise Magic

(Bonus round!)

Coroutines

"Coroutines are computer program components that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations."

Generators = Shallow Coroutines

```
function* fibonacci() {
    var [prev, curr] = [0, 1];
    while (true) {
        [prev, curr] = [curr, prev + curr];
        yield curr;
    }
}

for (n of fibonnaci()) {
    console.log(n);
}
```

@DOMENIC

http://taskjs.org/

task.js: Generators + Promises = Tasks

```
spawn(function* () {
    var data = yield $.ajax(url);
    $("#result").html(data);
    var status = $("#status").html("Download complete.");
    yield status.fadeIn().promise();
    yield sleep(2000);
    status.fadeOut();
});
```

task.js Even Works on Exceptions

```
spawn(function* () {
    var user;
    try {
        user = yield getUser();
    } catch (err) {
        ui.showError(err);
        return;
    }

    ui.updateUser(user);
});
```

Remote Promises

```
userPromise
    .get("friends")
    .get("0")
    .invoke("calculateFriendshipCoefficient")
    .then(displayInUI)
    .done();
```

What if … **userPromise** referred to a *remote object?!*

https://github.com/kriskowal/q-connection/

Q Connection

- Can connect to web workers, <iframe>s, or web sockets

```
var Q = require("q");
var Connection = require("q-comm");
var remote = Connection(port, local);

// a promise for a remote object!
var userPromise = remote.getUser();
```

Promise Pipelining

- Usual “remote object” systems fall down in a few ways:
  - They would see the first request, and return the entire friends array.
  - They can’t invoke methods that involved closed-over state, only methods that you can send over the wire.
  - Workarounds involve complex serialization and rehydration approaches, i.e. require coupling the client and the server.
- With promises as the abstraction, we can “pipeline” messages from one side to the other, returning the ultimately-desired result.

What's next

- Start using promises in your code: client, server, everywhere.
- Be aware that you want a Promises/A+ compatible library—beware jQuery.
- Generators are almost ready in Firefox, Chrome, and Node.js.
- Investigate promises for real-time communication with Q-Connection.
- Look forward to promises in the DOM, and maybe some syntactic support in ECMAScript 7!