AngularJS 2.0 Promise Design Doc

*Status: Draft*

*Authors: Jeff Cross (*[*crossj@google.com*](mailto:crossj@google.com)*)*

# Objective

AngularJS 2.0 needs promise support to build out the core libraries that make up AngularJS, as well as to provide users of AngularJS with a default option for using promises with AngularJS 2.0 apps.

ES6 Promises are now a first class citizen of the Web Platform, and are being implemented in major browsers[[1]](#footnote-0)[[2]](#footnote-1)! As such, there is less burden on frameworks to implement all the underpinnings of promises. This document will explore the opportunity to provide APIs to supplement the native implementation to improve developer experience and testing, and will propose a detailed design to be implemented as part of AngularJS 2.0.

One challenge posed by having a native implementation of Promise is that the internal mechanisms are now hidden from JavaScript, particularly the queueing and execution of async microtasks, which presents some challenges to the way tests are written. This doc will attempt to identify such issues, and propose solutions.

In addition to testing challenges, this doc will explore ways of making Promises more enjoyable to use for end users by exploring different APIs.

## Example ES6 Promise Implementation

|  |
| --- |
| function doneCounting(val) {  alert('Counted to ' + val);  return val; }   function couldNotCount(reason) {  alert('Could not count');  return reason; }   function resolver(resolve, reject) {  setTimeout(function doneCounting() {  resolve(5);  }, 5000); }   var countToFive = new Promise(resolver); countToFive.then(doneCounting, couldNotCount); |

## Requirements

### As Native As Possible

In an ideal world, when developers need the control-flow semantics that promises provide, they would use *only* the native Promise implementations found in JavaScript runtimes. This design doc should aim to come close to this ideal scenario, and provide justification for cases where the design deviates from using only natively-available constructs.

### Small

Any solution built on top of the native Promise implementation should be small in terms of:

1. Bytes
2. API surface - Easy to use, easy to maintain
3. Complexity - Easy to comprehend

### Swappable

Developers should be able to use any promise library they want with their AngularJS 2 applications, without worrying about complexities associated with AngularJS 1, such as the digest loop.

### Intuitive

Any promise-related API within the scope of this design doc should be simple to understand, and consistent with the design of other libraries in AngularJS 2.0.

### Testable

#### Expectations

The common approach for testing a Promise, or something based on promises, is checking that the expected chain of functions was called at the right time, with the right values, based on various possible resolutions. For example:

* The “expected chain” could be all resolve functions passed into “then”, presuming the promise was resolved successfully, or reject functions if not.
* The “right time” could be after calling a fetch() method on a service (and after flushing any underlying mocks, like XMLHttpRequest).
* The “right value” could be an http response or error.
* The “possible resolutions” could be reject or resolve, with many different possible values for either.

Below is an example of testing one method of a promise-based service, incorporating all of the priorities from the previous paragraph. The UserService’s “fetch” method is tested once with a successful http response, once with a 404. The tests use a partially-implemented mock of XMLHttpRequest to synchronously provide different responses to the request made within the service.

|  |
| --- |
| describe('UserService', function() {  var XMLHttpRequest = function () {  XMLHttpRequest.connections = XMLHttpRequest.connections || [];  XMLHttpRequest.connections.push(this);  };   XMLHttpRequest.prototype.open = function(method, url, async) {  this.method = method;  this.url = url;  this.async = async;  };    XMLHttpRequest.prototype.send = function(data) {  *//Oversimplified example*  };   XMLHttpRequest.flush = function() {  XMLHttpRequest.whens.forEach(function(when, whenI) {  XMLHttpRequest.connections.forEach(function(xhr, xhrI) {  var error;  if (when.url === xhr.url && when.method === xhr.method) {  XMLHttpRequest.whens.splice(whenI, 1);  XMLHttpRequest.connections.splice(whenI, 1);  error = when.responseCode >= 400;  xhr.responseText = when.responseText;  xhr.onload.call(xhr, error);  }  });  });   if(XMLHttpRequest.connections.length || XMLHttpRequest.whens.length) {  throw new Error('Unflushed connections or unresolved whens');  }  };   XMLHttpRequest.when = function(method, url) {  XMLHttpRequest.whens = XMLHttpRequest.whens || [];  var newWhen = {method: method, url: url};  XMLHttpRequest.whens.push(newWhen);   return {  respond: function(text) {  if (typeof text === 'number') {  newWhen.responseCode = text;  }  else {  newWhen.responseText = text;  }  }  }  };   var UserService = function () {};  UserService.prototype.fetch = function () {  var service = this;  return new Promise(function(resolve, reject) {  var xhr = new XMLHttpRequest();  xhr.open('get', '/users');  xhr.send();  xhr.onload = function (e) {  if (e) {  service.users = [];  reject(e);  }  else {  service.users = JSON.parse(this.responseText).users;  resolve(service.users);  }  }  });  };   it('should fetch users from a server', function(done) {  var responseSpy = jasmine.createSpy().and.callThrough();  var responseText = '{"users":["Jeff"]}';  var userService = new UserService;   userService.fetch().  then(responseSpy).  then(function() {  expect(responseSpy).toHaveBeenCalledWith(['Jeff']);  expect(userService.users).toEqual(['Jeff']);  done();  });   XMLHttpRequest.when('get', '/users').respond(responseText);  XMLHttpRequest.flush();  });    it('should set an empty users array if fetching users fails', function(done) {  var userService = new UserService;   userService.fetch().  then(null, function (response) {  expect(userService.users).toEqual([]);  done();  });   XMLHttpRequest.when('get', '/users').respond(404);  XMLHttpRequest.flush();  }); }); |

#### Synchronous-ability

Promises resolve asynchronously (as they should).

A challenge of using native Promises in tests is that native browser implementations use hidden mechanisms to make the fulfillment of promises happen asynchronously. Since the mechanisms are hidden, it’s impossible to mock them with synchronous mechanisms underneath. This means that any unit tests of features that require promises to work cannot be synchronous when using the browser-built-in Promise constructor.

While most testing frameworks provide a means to run tests asynchronously, this is not ideal for a couple of reasons:

1. Asynchronous tests require more code per test, extra closures and callback passing, all distracting from the true nature of the test.
2. Synchronous tests, where the test author is in control of task scheduling and execution allow us to repeatedly write tests where the order of events is explicit and intentional. This means we can test behavior of various race conditions.
3. Asynchronous tests are slower than synchronous tests (each micro second matters if you have a unit test suite of thousands of tests that you want to execute almost instantly many times for minute).

Example async test (taken from test sample under [Expectations](#h.se8958sjyw45)):

|  |
| --- |
| it('should fetch users from a server', function(done) {  var responseSpy = jasmine.createSpy().and.callThrough();  var responseText = '{"users":["Jeff"]}';  var userService = new UserService;   userService.fetch().  then(responseSpy).  then(function() {  expect(responseSpy).toHaveBeenCalledWith(['Jeff']);  expect(userService.users).toEqual(['Jeff']);  done();  });   XMLHttpRequest.when('get', '/users').respond(responseText);  XMLHttpRequest.flush(); }); |

This test is not ideal for these reasons:

* The test will be slightly slower due to underlying async mechanisms
* The expect() within the final “then” may not ever be called, failing the test with a generic timeout error, depriving the user of information about the value of the object being evaluated.

# Prior Art

* [January 20 ES6 Spec](https://people.mozilla.org/~jorendorff/es6-draft.html) (Most recent draft of ES6 spec, with re-addition of Promise)
* [Promises-unwrapping](https://github.com/domenic/promises-unwrapping) - Proving ground for ES6 Promise spec
* [Angular 1 $q](http://docs.angularjs.org/api/ng/service/$q)
* [Kris Kowal’s Q](https://github.com/kriskowal/q)
* [Tilde RSVP](https://github.com/tildeio/rsvp.js)
* [V8 Promise Implementation](https://code.google.com/p/v8/source/browse/branches/bleeding_edge/src/promise.js?r=18113)

# Detailed Design

## Design Overview

* Create a module called ngDeferred
* Create a lightweight Deferred class around Promise
* Create a flushable mock implementation of Promise
* Use dependency injection to inject Promise (or Mock) into Deferred
* Eventually, any promise-based DOM APIs will need to be mocked to be synchronously testable (outside the scope of this doc)

## Deferred

The detailed design provides a simple API to create and resolve promises, via a class called Deferred. Instances of deferred will have three fields:

* promise - a new promise
* resolve - alias to the resolve function provided to the resolver function during Promise instantiation
* reject - alias to the reject function provided to the resolver function during Promise instantiation

Providing this small wrapper around Promise accomplishes these objectives:

* A flatter API is accomplished by avoiding the need to pass in a resolver function at construction time, allowing developers to just called deferred.reject() or resolve() instead outside the context of the resolver function.
* Easy swapping/mocking of the underlying Promise mechanism using dependency injection (pending resolution of [this issue](https://github.com/angular/di.js/issues/22)).

Example “fetch” method from “[Expectations](#h.se8958sjyw45)” using native Promise:

|  |
| --- |
| UserService.prototype.fetch = function () {  var service = this;  return new Promise(function(resolve, reject) {  var xhr = new XMLHttpRequest();  xhr.open('get', '/users');  xhr.send();  xhr.onload = function (e) {  if (e) {  service.users = [];  reject(e);  }  else {  service.users = JSON.parse(this.responseText).users;  resolve(service.users);  }  }  }); }; |

Same example with Deferred:

|  |
| --- |
| import {Deferred} from 'ngDeferred'; UserService.prototype.fetch = function () {  var service = this;  var deferred = new Deferred();  var xhr = new XMLHttpRequest();  xhr.open('get', '/users');  xhr.send();  xhr.onload = function (e) {  if (e) {  service.users = [];  deferred.reject(e);  }  else {  service.users = JSON.parse(this.responseText).users;  deferred.resolve(service.users);  }  };  return deferred.promise;  }; |

## Promise Mock

A mock Promise class will be created with the same API as the native ES6 Promise, with some additional fields:

* static flush() - immediately calls all queued fulfillment functions for all promises with pending fulfillment (resolve/reject) operations
* ~~static state(promise) - current state of the given promise (pending, rejected, or resolved)~~

### Objection 1: Just Mock the Async Mechanism

Some research and experimentation was spent determining if the native Promise could be used in fully synchronous tests, by overriding mechanisms that may be used by the class under the covers to accomplish asynchronous execution (e.g. setTimeout). The Promise implementation, at least in V8, uses microtask mechanisms that are not exposed to JavaScript to support the async execution of tasks, which makes this approach impossible.

### Objection 2: Why Not Just Mock the Async Mechanism in Traceur’s Promise?

Since Traceur provides its own implementation of ES6 Promise (based on V8’s implementation), we could submit a patch to Traceur that would allow us to manually flush the async mechanism, saving ourselves the effort of creating a Mock Promise. The promise implementation in tests shouldn’t always require the Traceur Promise implementation to run successfully. Down the road, when ES6 is more ubiquitous in JavaScript engines, all tests against promise-backed features should still pass without assuming that tests are able to use the traceur implementation of Promise.

### Objection 3: Why Not Just Monkey-Patch Promise Constructor and “then”

Because the promise object does not expose the chain of fulfillment operations, it would require rewriting Promise.prototype.then to register fulfillment functions to a proprietary queue outside of the native promise. But this still leaves the problem that calling the resolve or reject functions within the resolver function would still use the internal fulfillment chain, ignoring whatever had been created within our proprietary queue. Therefore, it would be necessary to overwrite the Promise constructor entirely in order to overwrite the resolve and reject functions passed into the resolver function, at which point so much of the Promise has been monkey-patched that there is no net value to this approach vs creating a standalone mock. It’s also worth noting that any DOM APIs that use Promise would probably maintain an internal reference to the real Promise constructor, and would ignore the new constructor (but would probably still use the updated “then” function that would have been changed on the Promise prototype).

# Caveats

The mock design requires two changes to Angular’s Dependency Injection.

1. @Inject annotation should allow constructing objects explicitly, injecting some arguments, and providing some (although the Deferred constructor in this design does not accept any additional arguments)[[3]](#footnote-2).
2. @Inject annotation should allow injecting un-instantiated classes so that mocking classes is easier[[4]](#footnote-3).

Example implementation:

|  |
| --- |
| @Inject(Promise)  class Deferred {  constructor (Promise) {  this.promise = new Promise();  }  }  var d = new Deferred();  d.promise instanceof Promise //true |

## Test Strategy

The Promise mock will provide a means of synchronously flushing all pending promise operations, but will not provide new APIs for resolving or rejecting a promise, beyond what is available in the native Promise implementation. Instead, each test will need to determine how to synchronously resolve any underlying asynchronous code so that the promise’s resolve or reject function gets called when expected, with the expected value. In the following example, setTimeout is mocked so that the resolver function can be flushed synchronously:

|  |
| --- |
| describe('Timeout Example', function() {  function delayResolver(resolve, reject) {  setTimeout(function doneCounting() {  resolve(5);  }, 5000);  }    it('should increment the value in 5 seconds', function () {  var flushTimeout, timeCount;  window.setTimeout = function(fn, delay) {  flushTimeout = fn;  }  var countToFive = new Promise(delayResolver);  countToFive.  then(function (count) {  timeCount = count;  });  flushTimeout();  Promise.flush();  expect(timeCount).toBe(5);  }); }); |

See [Promise Mock](#h.h04xyeft3tnd) for more details on testing.

Since new DOM APIs, like [Service Workers](https://github.com/slightlyoff/ServiceWorker/tree/cf459d473ae09f6994e8539113d277cbd2bce939), are using promises, any code written against those APIs will either have to work against a mock implementation of the API, or be written as an async test. The recommended approach is to create a mock for any DOM API that is inherently asynchronous.

# Work Breakdown

* Build Deferred and publish to Github/npm
* Build Promise Mock and publish to Github/npm, alongside updates to DI

1. https://groups.google.com/forum/#!forum/v8-users [↑](#footnote-ref-0)
2. https://developer.mozilla.org/en-US/Firefox/Releases/25 [↑](#footnote-ref-1)
3. https://github.com/angular/di.js/issues/22 [↑](#footnote-ref-2)
4. https://github.com/angular/di.js/issues/30 [↑](#footnote-ref-3)