Promises And Design Patterns In AngularJS

By Freek Wielstra February 23, 2014 $q · AngularJS · Javascript · Promises 17 Comments

The traditional way to deal with asynchronous tasks in Javascript are callbacks; call a method, give it a function reference to execute once that method is done.

```
$.get('api/gizmo/42', function(gizmo) {
  console.log(gizmo); // or whatever
});
```

This is pretty neat, but, it has some drawbacks; for one, combining or chaining multiple asynchronous processes is tricky; it either leads to a lot of boilerplate code, or what's known as callback hell (nesting callbacks and calls in each other):

```
$.get('api/gizmo/42', function(gizmo) {
  $.get('api/foobars/' + gizmo, function(foobar) {
    $.get('api/barbaz/' + foobar, function(bazbar) {
      doSomethingWith(gizmo, foobar, bazbar);
    }, errorCallback);
  }, errorCallback);
}, errorCallback);
```

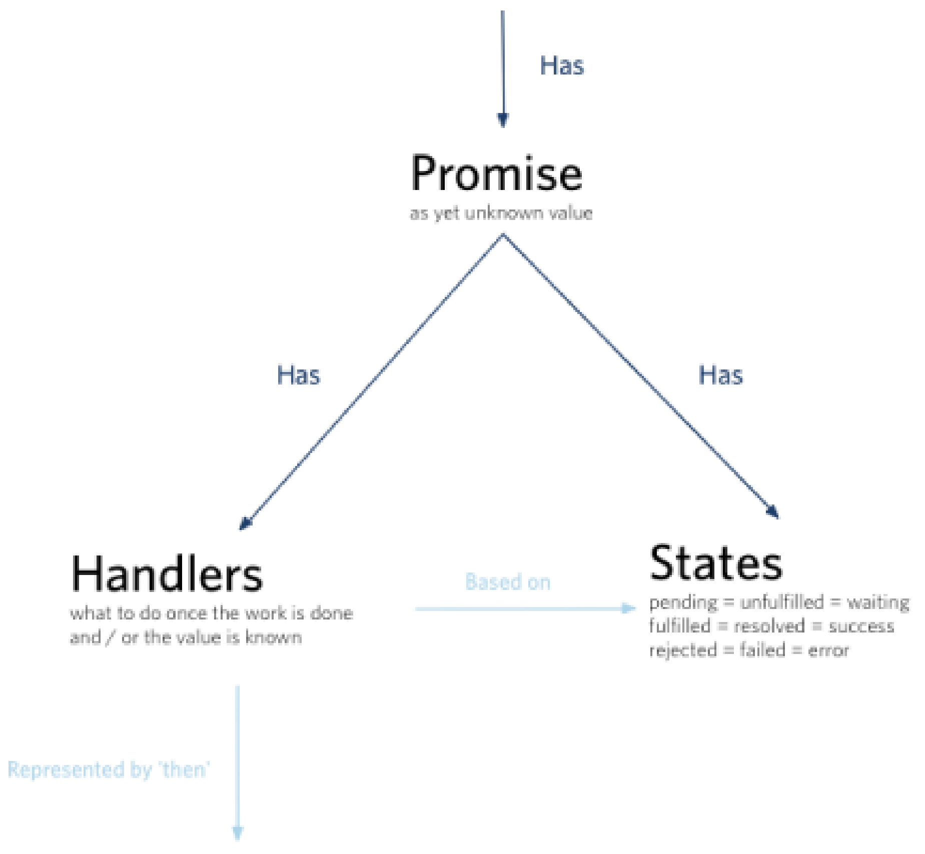
You get the idea. In Javascript however, there is an alternative to dealing with asynchronous code: Futures, although in Javascript they're often referred to as Promises. The CommonJS standards committee has released a spec that defines this API called Promises.

The concept behind promises is pretty simple, and has two components:

- Deferreds, representing **units of work**, and
- Promises, representing **data** from those Deferreds.

Basically, you use a Deferred as a communications object to signal the start, progress, and completion of work.

A Promise in turn is an object output by a Deferred that represents data; it has a certain State (pending, fulfilled or rejected), and Handlers, or callback methods th uld be called once a promise resolves, rejects, or gives a progress update.



Has
Promise
as yet unknown value
Has
Has
Handlers
what to do once the work is done and / or the value is known
Based on
States
pending = unfulfilled = waiting
fulfilled = resolved = success
rejected = failed = error
Represented by 'then'

ource: http://blog.mediumequalsmessage.com/promise-deferred-objects-in-javascript-pt1-theory-and-emantics

An important thing that differentiates promises from callbacks is that you can attach a handler *after* the promise state goes to *resolved*. This allows you to pass data that may or may not be there yet around in your application, cache it, etc, so that its consumers can perform operations on the data either immediately or as soon as it arrives.

For the remainder of this article we'll talk about promises and such in the context of AngularJS.

AngularJS relies heavily on promises throughout its codebase, both the framework and the application code you write in it. AngularJS uses its own implementation of the Promises spec, the $q service, which in turn is a lightweight implementation of the Q library.

$q implements all of the Deferred / Promise methods described above, plus a few in *$q* itself: *$q.defer()*, which creates a new Deferred object; *$q.all()*, which allows you to wait for multiple promises to resolve, and the methods *$q.when()* and *$q.reject()*, utility methods we'll go into later on.

$q.defer() returns a Deferred object, which has the methods *resolve()*, *reject()*, and *notify()*. Deferred has a property *promise*, which is the promise object that can be passed around the application.

The promise object has another three methods: *.then()*, which is the only method required by the Promises spec, taking three callbacks as arguments; one for success, one for failure, and one for notifications.

$q adds two methods on top of the Promise spec though: *catch()*, which can be used to have a centralized function to be called if *any* of the promises in a promise chain fails, and *finally()*, a method that will always be called regardless of success or failure of the promises. Note that these are not to be confused or used in combination with Javascript's exception handling: an exception thrown inside a promise will not be caught by *catch()*.

Simple Promise Example

Here's a basic example of using $q, Deferred, and Promise in one. As a disclaimer, none of the code examples in this post have been tested; they also lack the appropriate angular service and dependency definitions, etcetera. But they should provide a good enough example to start fiddling with them yourself.

First, we create a new unit of work by creating a Deferred object, using *$q.defer()*:

```
var deferred = $q.defer();
```

Next, we'll grab the *promise* from the Deferred and attach some behavior to it.

```
var promise = deferred.promise;

promise.then(function success(data) {
  console.log(data);
}, function error(msg) {
  console.error(msg);
});
```

Finally, we perform some fake work and indicate we're done by telling the deferred:

```
deferred.resolve('all done!');
```

Of course, that's not really asynchronous, so we can just fake that using Angular's *$timeout* service (or Javascript's *setTimeout*, but, prefer *$timeout* in Angular applications so you can mock / test it)

```
$timeout(function() {
  deferred.resolve('All done... eventually');
}, 1000);
```

And the fun part: we can attach multiple *then()*s to a single promise, as well as attach *then()*s *after* the promise has resolved:

```
var deferred = $q.defer();
var promise = deferred.promise;

// assign behavior before resolving
promise.then(function (data) {
  console.log('before:', data);
});

deferred.resolve('Oh look we\'re done already.')

// assign behavior after resolving
promise.then(function (data) {
  console.log('after:', data);
});
```

Now, what if some error occurred? We'll use *deferred.reject()*, which will cause the second argument of *then()* to be called. Just like callbacks.

```
var deferred = $q.defer();
var promise = deferred.promise;

promise.then(function success(data) {
  console.log('Success!', data);
}, function error(msg) {
  console.error('Failure!', msg);
});

deferred.reject('We failed :(');
```

As an alternative to passing a second argument to *then()*, you can *chain* it with a *catch()*, which will be called if anything goes wrong in the promise chain (more on chaining later):

```
promise
  .then(function success(data) {
    console.log(data);
  })
  .catch(function error(msg) {
    console.error(msg);
  });
```

As an aside, for longer-term processes (like uploads, long calculations, batch operations, etc), you can use *deferred.notify()* and the third argument of *then()* to give the promise's listeners a status update:

```
var deferred = $q.defer();
var promise = deferred.promise;

promise
  .then(function success(data) {
    console.log(data);
  },
  function error(error) {
    console.error(error);
  },
  function notification(notification) {
    console.info(notification);
  }));

 var progress = 0;
 var interval = $interval(function() {
  if (progress >= 100) {
    $interval.cancel(interval);
    deferred.resolve('All done!');
  }
  progress += 10;
  deferred.notify(progress + '%...');
 }, 100)
```

Chaining Promises

We've seen earlier that you can attach multiple handlers (*then()*) to a single promise. The nice part about the promise API is that it allows chaining of handlers:

```
promise
  .then(doSomething)
  .then(doSomethingElse)
  .then(doSomethingMore)
  .catch(logError);
```

For a simple example, this allows you to neatly separate your function calls into pure, single-purpose functions, instead of one-thing-does-all; double bonus if you can re-use those functions for multiple promise-like tasks, just like how you would chain functional methods (on lists and the like).

It becomes more powerful if you use the result of a previous asynchronous to trigger a next one. By default, a chain like the one above will pass the returned object to the next *then()*. Example:

```
var deferred = $q.defer();
var promise = deferred.promise;

promise
  .then(function(val) {
    console.log(val);
    return 'B';
  })
  .then(function(val) {
    console.log(val);
    return 'C'
  })
  .then(function(val) {
    console.log(val);
   });

deferred.resolve('A');
```

This will output the following to the console:

```
A
B
C
```

This is a simple example though. It becomes really powerful if your *then()* callback returns *another romise*. In that case, the next *then()* will only be executed once that promise resolves. This pattern an be used for serial HTTP requests, for example (where a request depends on the result of a revious one):

```
var deferred = $q.defer();
var promise = deferred.promise;

// resolve it after a second
$timeout(function() {
  deferred.resolve('foo');
}, 1000);

promise
  .then(function(one) {
    console.log('Promise one resolved with ', one);

    var anotherDeferred = $q.defer();

    // resolve after another second

    $timeout(function() {
      anotherDeferred.resolve('bar');
    }, 1000);

    return anotherDeferred.promise;
  })
  .then(function(two) {
    console.log('Promise two resolved with ', two);
  });
```

n summary:

- Promise chains will call the next 'then' in the chain with the return value of the previous 'then' callback (or undefined if none)
- If a 'then' callback returns a promise object, the next 'then' will only execute if/when that promise resolves
- A final 'catch' at the end of the chain will provide a single error handling point for the entire chain
- A 'finally' at the end of the chain will always be executed regardless of promise resolving or rejection, for cleanup purposes.

Parallel Promises And 'Promise-Ifying' Plain Values

ne method I mentioned brielfly was *$q.all()*, which allows you to wait for multiple promises to esolve in parallel, with a single callback to be executed when all promises resolve. In Angular, this ethod has two ways to be called: with an *Array* or an *Object*. The *Array* variant takes an array and alls the *.then()* callback with a single array result object, where the results of each promise orrespond with their index in the input array:

```
$q.all([promiseOne, promiseTwo, promiseThree])
  .then(function(results) {
    console.log(results[0], results[1], results[2]);
  });
```

The second variant accepts an *Object* of promises, allowing you to give names to those promises in our callback method (making them more descriptive):

```
$q.all({ first: promiseOne, second: promiseTwo, third: promiseThree })
  .then(function(results) {
    console.log(results.first, results.second, results.third);
  });
```

would only recommend using the array notation if you can batch-process the result, i.e. if you treat he results equally. The object notation is more suitable for self-documenting code.

Another utility method is *$q.when()*, which is useful if you just want to create a promise out of a plain ariable, or if you're simply not sure if you're dealing with a promise object.

```
$q.when('foo')
  .then(function(bar) {
    console.log(bar);
  });

$q.when(aPromise)
  .then(function(baz) {
    console.log(baz);
  });

$q.when(valueOrPromise)
  .then(function(boz) {
    // well you get the idea.
  })
```

q.when() is also useful for things like caching in services:

```
angular.module('myApp').service('MyService', function($q, MyResource) {

  var cachedSomething;

  this.getSomething = function() {
    if (cachedSomething) {
      return $q.when(cachedSomething);
    }

    // on first call, return the result of MyResource.get()
    // note that 'then()' is chainable / returns a promise,
    // so we can return that instead of a separate promise object
    return MyResource.get().$promise
      .then(function(something) {
        cachedSomething = something
      });
  };
});
```

And then call it like this:

```
MyService.getSomething()
    .then(function(something) {
        console.log(something);
    });
```

Practical Applications In AngularJS

Most I/O in Angular returns promises or promise-compatible ('then-able') objects, however often with a twist. $http's documentation indicates it returns a *HttpPromise* object, which is a Promise but with two extra (utility) methods, probably to not scare off jQuery users too much. It defines the methods *success()* and *error()*, which correspond to the first and second argument of a *then()* callback, respectively.

Angular's *$resource* service, a wrapper around *$http* for REST-endpoints, is also a bit odd; the generated methods (*get()*, *save()* andsoforth) accept a second and third argument as success and error callbacks, while they also return an object that gets populated with the requested data when he request is resolved. It does not return a promise object directly; instead, the object returned by a esource's *get()* method has a property *$promise*, which exposes the backing promise object.

On the one side, it's inconsistent with *$http* and how everything in Angular is or should be a promise out on the other side it allows a developer to simply assign the result of *$resource.get()* to the *$scope*. Previously, a developer could assign any promise to the *$scope*, but since Angular 1.2 that has been deprecated: see this commit where it was deprecated.

Personally, I like to have a consistent API, so I wrap pretty much all I/O in a *Service* that will always eturn a *promise* object, but also because calling a *$resource* is often a bit rough around the edges. Here's a random example:

```
angular.module('fooApp')
  .service('BarResource', function ($resource) {
    return $resource('api/bar/:id');
  })

  .service('BarService', function (BarResource) {

    this.getBar = function (id) {
      return BarResource.get({
        id: id
      }).$promise;
    }

  });
```

This example is a bit obscure because passing the id argument to *BarResource* looks a bit duplicate, out it makes sense if you've got a complex object but need to call a service with just an ID property from it. The advantage of the above is that in your controller, you know that anything you get from a *Service* will always be a *promise* object; you don't have to wonder whether it's a promise or resource esult or a *HttpPromise*, which in turn makes your code more consistent and predictable - and since avascript is weakly typed and as far as I know there's no IDE out there yet that can tell you what type a method returns without developer-added annotations, that's pretty important.

Practical Chaining Example

One part of the codebase we are currently working on has calls that rely on the results of a previous call. Promises are ideal for that, and allow you to write in an easy to read code style as long as you keep your code clean. Consider the following example:

```
angular.module('WebShopApp')
  .controller('CheckoutCtrl', function($scope, $log, CustomerService, CartService, CheckoutService) {

    function calculateTotals(cart) {
      cart.total = cart.products.reduce(function(prev, current) {
        return prev.price + current.price;
      };

      return cart;
    }

    CustomerService.getCustomer(currentCustomer)
      .then(CartService.getCart) // getCart() needs a customer object, returns a cart
      .then(calculateTotals)
      .then(CheckoutService.createCheckout) // createCheckout() needs a cart object, returns a checkout c
      .then(function(checkout) {
        $scope.checkout = checkout;
      })
      .catch($log.error)

    });
```

This combines getting data asynchronously (customers, carts, creating a checkout) with processing data synchronously (*calculateTotals*); the implementation however doesn't know or need to know whether those various services are async or not, it will just wait for the methods to complete, async or not. In this case, *getCart()* could fetch data from local storage, *createCheckout()* could perform a HTTP request to make sure the products are all in stock, etcetera. But from the consumer's point of view (the one making the calls), it doesn't matter; it Just Works. And it clearly states what it's doing, just as long as you remeber that the result of the previous *then()* is passed to the next.

And of course it's self-documenting and concise.

Testing Promise-Based Code

Testing promises is easy enough. You can either go the hard way and have your test create mock objects that expose a *then()* method, which is called directly. However, to keep things simple, I just use *$q* to create promises - it's a very fast library and you're guaranteed to not be missing any promise implementation subtleties. The following spec tries to demonstrate how to mock out the various services used above. Note that it is rather verbose and long, but, I haven't found a way around it yet outside of making utility methods for promise creation (pointers to making it shorter / more concise are welcome).

```
describe('The Checkout controller', function() {

  beforeEach(module('WebShopApp'));

  it('should do something with promises', inject(function($controller, $q, $rootScope) {

    // create mocks; in this case I use jasmine, which has been good enough for me so far as a mocking li
    var CustomerService = jasmine.createSpyObj('CustomerService', ['getCustomer']);
    var CartService = jasmine.createSpyObj('CartService', ['getCart']);
    var CheckoutService = jasmine.createSpyObj('CheckoutService', ['createCheckout']);

    var $scope = $rootScope.$new();
    var $log = jasmine.createSpyObj('$log', ['error']);

    // Create deferreds for each of the (promise-based) services
    var customerServiceDeferred = $q.defer();
    var cartServiceDeferred = $q.defer();
    var checkoutServiceDeferred = $q.defer();

    // Have the mocks return their respective deferred's promises
    CustomerService.getCustomer.andReturn(customerServiceDeferred.promise);
    CartService.getCart.andReturn(cartServiceDeferred.promise);
    CheckoutService.createCheckout.andReturn(checkoutServiceDeferred.promise);

    // Create the controller; this will trigger the first call (getCustomer) to be executed,
    // and it will hold until we start resolving promises.
    $controller("CheckoutCtrl", {
      $scope: $scope,
      CustomerService: CustomerService,
      CartService: CartService,
      CheckoutService: CheckoutService
    });

    // Resolve the first customer.
    var firstCustomer = {id: "customer 1"};
    customerServiceDeferred.resolve(firstCustomer);

    // ... However: this *will not* trigger the 'then()' callback to be called yet;
    // we need to tell Angular to go and run a cycle first:

    $rootScope.$apply();

    expect(CartService.getCart).toHaveBeenCalledWith(firstCustomer);

    // setup the next promise resolution
    var cart = {products: [ { price: 1 }, { price: 2 } ]}
    cartServiceDeferred.resolve(cart);

    // apply the next 'then'
    $rootScope.$apply();

    var expectedCart = angular.copy(cart);
    cart.total = 3;

    expect(CheckoutService.createCheckout).toHaveBeenCalledWith(expectedCart);

    // Resolve the checkout service
    var checkout = {total: 3}; // doesn't really matter
    checkoutServiceDeferred.resolve(checkout);

    // apply the next 'then'
    $rootScope.$apply();

    expect($scope.checkout).toEqual(checkout);

    expect($log.error).not.toHaveBeenCalled();
  }));
});
```

As you can see, testing promise code is about ten times as long as the code itself; I don't know if / ıow to have the same power in less code, but, maybe there's a library out there I haven't found (or nade) yet.

To get full test coverage, one will have to write tests wherein all three services fail to resolve, one ıfter the other, to make sure the error is logged. While not clearly visible in the code, the code / process actually does have a lot of branches; every promise can, after all, resolve or reject; true or alse, or branch out. But, that level of testing granularity is up to you in the end.

 hope this article gives people some insight into promises and how they can be used in any Angular ıpplication. I think I've only scratched the surface on the possibilities, both in this article and in the AngularJS projects I've done so far; for such a simple API and such simple concepts, the power and mpact promises have on most Javascript applications is baffling. Combined with high-level unctional utilities and a clean codebase, promises allow you to write your application in a clean, naintainable and easily altered fashion; add a handler, move them around, change the mplementation, all these things are easy to do and comprehend if you've got promises under control.

With that in mind, it's rather odd that promises were scrapped from NodeJS early on in its development in favor of the current callback nature; I haven't dived into it completely yet, but it seems it had performance issues that weren't compatible with Node's own goals. I do think it makes sense though, if you consider NodeJS to be a low-level library; there are plenty of libraries out there that add the higher-level promises API to Node (like the aforementioned Q).

Another note is that I wrote this post with AngularJS in mind, however, promises and promise-like programming has been possible in the grandfather of Javascript libraries for a couple of years now, Query; Deferreds were added in jQuery 1.5 (January 2011). Not all plugins may be using them

:onsistently though.

Similarly, Backbone.js' Model api also exposes promises in its methods (*save()* etc), however what I understand it doesn't really work right alongside its model events. I might be wrong though, it's been a while.

would definitely recommend aiming for a promise-based front-end application whenever developing a new webapp, it makes the code so much cleaner, especially combined with functional programming paradigms. More functional programming patterns can be found in Reginald Braithwaite's Javascript Allongé book, free to read on LeanPub; some of those should also be useful n writing promise-based code.

COMMENTS (17)

Mathews - Reply
March 2, 2014 at 8:20 pm

Excellent write up - this was very helpful for me!

Colin Huang - Reply
March 5, 2014 at 9:21 pm

A well written piece on Promises in AngularJS and up to date with the Promise API in the current version (1.2.x)

LeJuan5150 - Reply
April 27, 2014 at 12:06 am

Fantastic ... Been trying to wrap my head around promises, especially promise chaining. Your article helped a ton!

Jim Beasley - Reply
June 13, 2014 at 1:50 am

Excellent and well written! It helped me to get a better handle on promises.

Robin - Reply
June 26, 2014 at 9:16 pm

Compiled in a very nice way. Excellent!!

Ranjan - Reply
July 22, 2014 at 3:08 pm

This is great stuff..very helpful..thanks a lot.