Promises

Angular's event system (which we discuss in depth in the under the hood chapter) provides a lot of power to our Angular apps. One of the most powerful features it gives us is the automatic resolution of promises.

What's a Promise?

A promise is a method of resolving a value (or not) in an asynchronous manner. Promises are objects that represent the return value or thrown exception that a function may eventually provide. Promises are incredibly useful in dealing with remote objects, and we can think of them as a proxy for our remote objects.

Traditionally, JavaScript uses closures, or callbacks, to respond with meaningful data that is not available synchronously, such as XHR requests after a page has loaded. Rather than depending upon a callback to fire, we can interact with the data as though it has already returned.

Callbacks have worked for a long time, but the developer suffers when using them. Callbacks provide no consistency and no guaranteed call, they steal code flow when depending upon other callbacks, and they generally make debugging incredibly difficult. At every step of the way, we have to deal with *explicitly* handling errors.

Instead of firing a function and hoping to get a callback run when executing asynchronous methods, promises offer a different abstraction: They return a promise object.

For example, in traditional callback code, we might have a method that sends a message from one user to one of the user's friends.

```
// Sample callback code
User.get(fromId, {
  success: function(err, user) {
    if (err) return {error: err};
    user.friends.find(toId, function(err, friend) {
      if (err) return {error: err};
      user.sendMessage(friend, message, callback);
    });
  },
  failure: function(err) {
    return {error: err}
  }
});
```

This callback pyramid is already getting out of hand, and we haven't included any robust error-handling code, either. Additionally, we need to know the order in which the arguments are called from within our callback.

The promised-based version of the previous code might look somewhat closer to:

```
User.get(fromId)
.then(function(user) {
  return user.friends.find(toId);
}, function(err) {
  // We couldn't find the user
})
.then(function(friend) {
  return user.sendMessage(friend, message);
}, function(err) {
  // The user's friend resulted in an error
})
.then(function(success) {
  // user was sent the message
}, function(err) {
  // An error occurred
});
```

Not only is this code more readable; it is also much easier to grok. We can guarantee that the callback will resolve to a single value, rather than having to deal with the callback interface.

Notice that in the first example, we have to handle errors differently from how we handle non-errors. We need to make sure when using callbacks to handle errors, we check if an error is defined in the tradition API response signature (usually with `(err, data)` being the usual method signature). All of our API methods must implement this same structure.

In the second example, we handle the success and error in the same way. Our resultant object will receive the error in the usual manner. The promise API is specific about resolving or rejecting promises, so we also don't have to worry about our methods implementing a different method signature.

Why Promises?

Escaping from *callback hell* is just one by-product of using promises. The real point of promises is to make asynchronous functions look more like synchronous ones. With synchronous functions, we can capture both return values and exception values as expected.

We can capture errors at any point of the process and bypass future code that relies upon the error of that process. We achieve all of these things without thinking about the benefits of this synchronous code – it's simply in the nature of the code.

Thus, the point of using promises is to regain the ability to do functional composition and error bubbling while maintaining the ability of the code to run asynchronously.

Promises are *first-class* objects and carry with them a few guarantees:

- Only one resolve or reject will ever be called

  - resolve is called with a single fulfillment value
  - reject can be called with a single rejection reason
- If the promise has been resolved or rejected, any handlers depending upon them will still be called
- Handlers will always be called asynchronously

Additionally, we can chain promises and allow the code to process as it would normally run. Exceptions from one promise bubble up through the entire promise chain.

Promises are *always* executed asynchronously; we can use them without worry that they will block the rest of the app.

Promises in Angular

Angular's event loop gives Angular the unique ability to resolve promises in its `$rootScope.$evalAsync` stage (see under the hood for more detail on the run loop). The promises will sit inert until the `$digest` run loop finishes.

This fact allows for us to turn the results of a promise into the view without any extra work. It also enables us to assign the result of an XHR call directly to a property on a `$scope` object and think nothing of it.

Let's build an example that will return a list of open pull requests for AngularJS from GitHub.

Play with it[69]

```
<h1>Open Pull Requests for Angular JS</h1>

<ul ng-controller="DashboardController">
  <li ng-repeat="pr in pullRequests">
    {{ pr.title }}
  </li>
</ul>
```

If we have a service that returns a promise (covered in depth in the services chapter), we can simply interact with the promise in the `.then()` method which allows us to modify any variable on the scope and place it in the view and expect that Angular will resolve it for us:

[69]http://jsbin.com/UfotanA/4/edit

```
angular.module('myApp', [])
.controller('DashboardController', [
  '$scope', 'GithubService',
    function($scope, GithubService) {
      // GithubService's getPullRequests() method
      // returns a promise
      GithubService.getPullRequests(123)
      .then(function(data) {
        $scope.pullRequests = data.data;
      });
}]);
```

When the asynchronous call to `getPullRequests` returns, the `$scope.pullRequests` value will be available in the `.then()` method which will then update the `$scope.pullRequests` array.

How to Create a Promise

In order to create a promise in Angular, we can use the built-in `$q` service. The `$q` service provides a few methods in its deferred API.

First, we need to inject the `$q` service into the object where we want to use it.

```
angular.module('myApp', [])
.factory('GithubService', ['$q', function($q) {
  // Now we have access to the $q library
}]);
```

To create a deferred object, we call the method `defer()`:

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

The `deferred` object exposes three methods and the single `promise` property that we can use to deal with the promise.

- resolve(value)

The `resolve` function resolves the deferred promise with the value.

```
deferred.resolve({name: "Ari", username: "@auser"});
```

- reject(reason)

This method *rejects* the deferred promise with a reason. It is equivalent to resolving a promise with a rejection.

```
deferred.reject("Can't update user");
// Equivalent to
deferred.resolve($q.reject("Can't update user"));
```

- notify(value)

This method responds with the status of a promises execution.

For example, if we want to return a status from the promise, we can use the `notify()` function to deliver it.

Let’s say that we have several long-running requests that we want to make from a single promise. We can call the `notify` function to send back a notification of progress:

```
.factory('GithubService', function($q, $http) {
  // get events from repo
  var getEventsFromRepo = function() {
    // task
  }
  var service = {
    makeMultipleRequests: function(repos) {
      var d = $q.defer(),
          percentComplete = 0,
          output = [];
      for (var i = 0; i < repos.length; i++) {
        output.push(getEventsFromRepo(repos[i]));
        percentComplete = (i+1)/repos.length * 100;
        d.notify(percentComplete);
      }

      d.resolve(output);

      return d.promise;
    }
  }
  return service;
});
```

With this `makeMultipleRequests()` function on our `GithubService` object, we will receive a progress notification every time a repo has been fetched and processed.

We can use this notification in our usage of the promise by adding a third function call to the promise usage. For instance:

```
.controller('HomeController',
function($scope, GithubService) {
  GithubService.makeMultipleRequests([
    'auser/beehive', 'angular/angular.js'
  ])
  .then(function(result) {
    // Handle the result
  }, function(err) {
    // Error occurred
  }, function(percentComplete) {
    $scope.progress = percentComplete;
  });
});
```

We can access the promise as a property on the deferred object:

```
deferred.promise
```

A full example of how to create a function that responds with a promise might look similar to the following method on the `GithubService`, as mentioned above.

```
angular.module('myApp', [])
.factory('GithubService', [
  '$q', '$http',
    function($q, $http) {
      var getPullRequests = function() {
        var deferred = $q.defer();
        // Get list of open angular js pull requests from github
        $http.get('https://api.github.com/repos/angular/angular.js/pulls')
        .success(function(data) {
          deferred.resolve(data);
        })
        .error(function(reason) {
          deferred.reject(reason);
        })
        return deferred.promise;
      }

      return { // return factory object
        getPullRequests: getPullRequests
      };
}]);
```

Now we can use the promise API to interact with the `getPullRequests()` promise.

View full example[70]

In the case of the service above, we can interact with the promise in two different ways:

- then(successFn, errFn, notifyFn)

Regardless of the success or failure of the promise, `then` calls either the `successFn` or the `errFn` asynchronously as soon as the result is available. The method **always** calls callbacks with a single argument: the result or the rejection reason.

It may call the `notifyFn` callback zero or more times to provide a progress status indication *before* the promise is resolved or rejected.

The `then()` method always returns a new promise, which is either resolved or rejected through the return value of the `successFn` or the `errFn`. It also serves notifications through the `notifyFn`.

- catch(errFn)

This method is simply a helper function that allows for us to replace the `err` callback with `.catch(function(reason) {})`:

```
$http.get('/repos/angular/angular.js/pulls')
.catch(function(reason) {
  deferred.reject(reason);
});
```

- finally(callback)

The `finally` method allows us to observe the fulfillment or rejection of a promise without modifying the resulting value. The method is useful for when we need to release a resource or run some clean-up, regardless of the success or error of the promise.

We cannot call this method directly, as *finally* is a reserved word in IE JavaScript. To use `finally`, we have to call it like so:

```
promise['finally'](function() {});
```

Angular's `$q` deferred objects are chainable in that even `then` returns a promise. As soon as the promise is resolved, the promise that `then` returns is resolved or rejected.

These promise chains are how Angular can support `$http`'s interceptors.

The `$q` service is similar to the original Kris Kowal's Q library:

1. $q is integrated with the Angular $rootScope model, so resolutions and rejections happen quickly inside of Angular.
2. $q promises are integrated with Angular's templating engine, which means that any promises found in the views will be resolved or rejected in the view.
3. $q is tiny and doesn't contain the full functionality of the Q library.

[70] http://d.pr/RWBz

Chaining Requests

The `then` method returns a new derived promise after the initial promise is resolved. This return gives us the unique ability to attach yet another `then` on the result of the initial `then` method.

```
// A service that responds with a promise
GithubService.then(function(data) {
  var events = [];
  for (var i = 0; i < data.length; i++) {
    events.push(data[i].events);
  }
  return events;
}).then(function(events) {
  $scope.events = events;
});
```

In this example, we can create a *chain* of execution that allows us to interrupt the flow of the application based upon more functionality, which we can attach to different results.

This interruption allows us to pause or defer resolutions of promises at any point during the chain of execution.

This interruption is also how the `$http` service implements request and response interceptors.

The `$q` library comes with several different useful methods:

all(promises)

If we have multiple promises that we want to combine into a single promise, we can use the `$q.all(promises)` method to combine them all into a single promise. This single method takes a single argument:

- promises (array or object of promises)

Promises as an array or hash of promises

The `all()` method returns a single promise that will resolve with an array or a hash of values. Each value will correspond to the promises at the same index or key in the promises hash. If **any** of the promises are resolved with a rejection, then the resulting promise will be rejected as well.

defer()

The `defer()` method creates a deferred object. It takes no parameters, and it returns a new instance of a single deferred object.

reject(reason)

This method creates a promise that is resolved with a rejection for a specific reason. It is specifically designed to give us access to forwarding rejection in a chain of promises, which is akin to `throw` in JavaScript. In the same sense that we can *catch* an exception in JavaScript and we can forward the rejection, we need to rethrow the error. We can do so with `$q.reject(reason)`.

This method takes a single parameter:

- reason (constant, string, exception, object)

The reasons for the rejection

The `reject()` method returns a promise that has already been resolved with a rejection and the reason for the rejection.

when(value)

The `when()` function wraps an object that might be a value `then`-able promise into a `$q` promise. Doing that allows for us to deal with an object that may or may not be a promise.

The `when()` function takes a single parameter:

- value

The value or a promise

The `when()` function returns a promise that we can then use like any other promise.