My first approach was to simply call one API after I had returned the result of the other:

$http({method: 'GET', url: '/api-one-url', cache: 'true'}).then(function(apiOneData){

$http({method: 'GET', url: '/api-two-url', cache: 'true'}).then(function(apiTwoData){

console.log(apiOneData, apiTwoData);

});

});

There are many problems with this approach, firstly the second API call wouldn’t be called until the first had been completed which will delay the results and secondly the code simply doesn’t look very nice and imagine how this would look if we had 7 or 8 different API calls all nested within one another like this. With this in mind I thought there must be a nicer way to do this. This is where I decided to look at how I could combine the promises

 Upon looking at the Q documentation I learnt about the $q.all() method. To use the$q.all() method we simply pass it an array of promises we want to resolve, what this does is it takes the array of promises and returns us a single promise which will resolve once all the original promises have been resolved. The data passed to our .then() method when using $q.all() is an array of the returned values in the same order that we passed our original promises.

If we then refactor the original two API calls from earlier it now using the Q.all() method it would look like this:

var promise1 = $http({method: 'GET', url: 'a/pi-one-url', cache: 'true'});

var promise2 = $http({method: 'GET', url: '/api-two-url', cache: 'true'});

$q.all([promise1, promise2]).then(function(data){

console.log(data[0], data[1]);

});

This is much more legible than the original nested method we showed before. Hopefully this short post on the$q.all() method will help you to understand how to combine two promises when using AngularJS. If you are using promises a lot in AngularJS I would encourage you to spend some time looking over the Q documentation as in addition to $q.all() there are many more useful features you can take advantage of.

**Promise (Deferred) Motivation**

In JavaScript, asynchronous methods usually use callbacks in order to inform a success or a failure state. The Geolocation api, for example, requires success and failure callbacks in order to [get the current position](https://developer.mozilla.org/en-US/docs/Web/API/Geolocation.getCurrentPosition):

Use callbacks in Geolocation api

function success(position) {

var coords = position.coords;

console.log('Your current position is ' + coords.latitude + ' X ' + coords.longitude);

}

function error(err) {

console.warn('ERROR(' + err.code + '): ' + err.message);

}

navigator.geolocation.getCurrentPosition(success, error);

Another example is XMLHttpRequest ([used to perform ajax calls](https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest)). It has onreadystatechange callback property that is called whenever the readyState attribute changes:

Callback use in XHR

var xhr = new window.XMLHttpRequest();

xhr.open('GET', 'http://www.webdev.com', true);

xhr.onreadystatechange = function() {

if (xhr.readyState === 4) {

if (xhr.status === 200) {

console.log('Success');

}

}

};

xhr.send();

There are many other examples of asynchronicity in JavaScript. Working with callbacks gets complicated when there is a need to synchronize several asynchronous operations.

**Sequentially Executing (Pyramid Of Doom)**

Assume we have N asynchronous methods: async1(success, failure), async2(success, failure), …, asyncN(success, failure) and we want to execute them sequentially, one after another, upon success. Each method gets success and failure callbacks so the code will be:

Execute asynchronous methods sequentially

async1(function() {

async2(function() {

async3(function() {

async4(function() {

....

....

....

asyncN(null, null);

....

....

....

}, null);

}, null);

}, null);

}, null);

And here we get the famous callback pyramid of doom. Although there are nicer ways to write this code (separate that waterfall into functions for example), this is really hard to read and maintain.

**Parallel Executing**

Assume we have N asynchronous methods: async1(success, failure), async2(success, failure), …, asyncN(success, failure) and we want to execute them parallely and alert a message **at the end of all**. Each method gets success and failure callbacks so the code will be:

Execute asynchronous methods parallel

var counter = N;

function success() {

counter --;

if (counter === 0) {

alert('done!');

}

}

async1(success);

async2(success);

....

....

asyncN(success);

We declared a counter with initial value equals to the total asynchronous methods to execute. When each method is done, we decrease the counter by one and check whether this was the last execution. This solution is not simple for implementation or maintain, especially when each asynchronous method passes a result value to success(). In such case we will have to keep the results of each execution.

In both examples, at the time of execution of an asynchronic operation, we had to specify how it will be handled using a success callback. In other words, when we use callbacks, the asynchronic operation needs a reference to its continuation, but this continuation might not be its business. This can lead to tightly coupled modules and services which makes it difficult when reusing or testing code.

**What are Promise and Deferred?**

A deferred represents the result of an asynchronic operation. It exposes an interface that can be used for signaling the state and the result of the operation it represents. It also provides a way to get the associated promise instance.   
A promise provides an interface for interacting with it’s related deferred, and so, allows for interested parties to get access to the state and the result of the deferred operation.   
When creating a deferred, it’s state is pending and it doesn’t have any result. When we resolve() or reject() the deferred, it changes it’s state to resolved or rejected. Still, we can get the associated promise immediately after creating a deferred and even assign interactions with it’s future result. Those interactions will occur only after the deferred rejected or resolved.

When it comes to coupling, by using promises we can easily create an asynchronic operation before even decide what’s going to happen after the resolve. This is why coupling is looser. An asynchronic operation doesn’t have to know how it continues, it only has to signal when it is ready.

While deferred has methods for changing the state of an operation, a promise exposes only methods needed to handle and figure out the state, but not methods that can change the state. This is why in a function, returning a promise and not a deferred is a good practice. This prevents from external code to interfere the progress or the state of an operation.

There are several implementations of promises in different languages (JavaScript, JAVA, C++, Python and more) and frameworks (NodeJS, jQuery for JavaScript). AngularJS has a promise implementation under the$q service.

**How to use Deferrers and Promises**

After understanding promises and their motivation, now is the time to see how to use Deferrers and Promises. As said before, there are several implementations of promises, and so, different implementations may have different ways of usage. This section will use [the AngularJS implementation of promise](https://docs.angularjs.org/api/ng/service/$q) – the $q service. Still, if you use a different implementation of promises, don’t worry, most of the methods I’ll describe here are equal for all implementations and if not, there is always an equivalent method.

**Basic usage**

First things first, let’s create a deferred!

Creating a deferred

var myFirstDeferred = $q.defer();

As simple as can be, myFirstDeferred holds a deferred that can be resolved or rejected whenever an asynchronous operation is done. Assume we have an asynchronous method async(success, failure) that gets success and failure callbacks as parameters. When async is done, we want to resolve or reject myFirstDeferred with the result (value or error reason):

Resolve and reject a deferred

async(function(value) {

myFirstDeferred.resolve(value);

}, function(errorReason) {

myFirstDeferred.reject(errorReason);

});

Since $q‘s resolve and reject methods don’t depend on a context in order to work, we can simply write:

Resolve and reject a deferred

async(myFirstDeferred.resolve, myFirstDeferred.reject);

Taking the promise out of myFirstDeferred and assigning operations upon success or failure is pretty easy:

Using the promise

var myFirstPromise = myFirstDeferred.promise;

myFirstPromise

.then(function(data) {

console.log('My first promise succeeded', data);

}, function(error) {

console.log('My first promise failed', error);

});

Keep on mind that we can assign the success and failure operations right after creating the deferred (even before calling to async()) and that we can assign as many operations as we like:

Using the promise several times

var anotherDeferred = $q.defer();

anotherDeferred.promise

.then(function(data) {

console.log('This success method was assigned BEFORE calling to async()', data);

}, function(error) {

console.log('This failure method was assigned BEFORE calling to async()', error);

});

async(anotherDeferred.resolve, anotherDeferred.reject);

anotherDeferred.promise

.then(function(data) {

console.log('This ANOTHER success method was assigned AFTER calling to async()', data);

}, function(error) {

console.log('This ANOTHER failure method was assigned AFTER calling to async()', error);

});

If async() successes, both success methods will occur. The same is for failure.   
A good approach is to wrap asynchronous operations with a function that returns a promise. This way the caller will be able to assign success and failure callbacks the way he likes, but will not be able to interfere the deferred state:

Wrap asynchronous operation

function getData() {

var deferred = $q.defer();

async(deferred.resolve, deferred.reject);

return deferred.promise;

}

...

... // Later, in a different file

var dataPromise = getData()

...

...

... // Much later, at the bottom of that file :)

dataPromise

.then(function(data) {

console.log('Success!', data);

}, function(error) {

console.log('Failure...', error);

});

Up to here, when we used promises, we assigned both success and failure callbacks. But, there is also a way to assign only success or only failure functions:

Assign only success or failure callback to promise

promise.then(function() {

console.log('Assign only success callback to promise');

});

promise.catch(function() {

console.log('Assign only failure callback to promise');

// This is a shorthand for `promise.then(null, errorCallback)`

});

Passing only success callback to promise.then() will assign only success callback and using promise.catch()will assign only failure callback. catch() is actually a shorthand for promise.then(null, errorCallback).   
In case we want to perform the same operation both on fulfillment or rejection of a promise, we can use promise.finally():

Using finally

promise.finally(function() {

console.log('Assign a function that will be invoked both upon success and failure');

});

This is equivalent to:

Finally explained

var callback = function() {

console.log('Assign a function that will be invoked both upon success and failure');

};

promise.then(callback, callback);

**Chaining values and promises**

Assume we have an asynchronous function async() that returns a promise. I have this interesting block of code:

Values chaining

var promise1 = async();

var promise2 = promise1.then(function(x) {

return x+1;

});

As you can understand from that code, promise1.then() returns another promise, and I named it promise2. When promise1 is resolved with a value x, the success callback executes and returns x+1. At this point promise2 is resolved with x+1.   
Another similar example:

Values chaining

var promise2 = async().then(function(data) {

console.log(data);

... // Do something with data

// Returns nothing!

});

Here, when the promise that returned from async() is resolved, the success callback does it’s job and then promise2 is resolved with no data (undefined).   
As you can see, ***promises can chain values and are always resolved after the callback occurs with the returned value***.   
In order to demonstrate it, here is a silly example that uses promises (there is no really a need to use promises here):

Values chaining example

// Let's imagine this is really an asynchronous function

function async(value) {

var deferred = $q.defer();

var asyncCalculation = value / 2;

deferred.resolve(asyncCalculation);

return deferred.promise;

}

var promise = async(8)

.then(function(x) {

return x+1;

})

.then(function(x) {

return x\*2;

})

.then(function(x) {

return x-1;

});

promise.then(function(x) {

console.log(x);

});

This promises chain starts with calling to async(8) which fulfills the promise with the value 4. This value passes through all the success callbacks and so the value 9 is logged ((8 / 2 + 1) \* 2 - 1).

What happens if we chain another promise (and not a value)? Assume we have two asynchronous functions, async1() and async2(), each returns a promise. Let’s see the following:

Promises chaining

var promise = async1()

.then(function(data) {

// Assume async2() needs the response of async1() in order to work

var async2Promise = async2(data);

return async2Promise;

});

Here, unlike the previous example, the success callback performs another asynchronous operation and returns a promise. The value returned from async1().then() is a promise as expected, but now it can be resolved or rejected according to async2Promise and with it’s resolve value or reject reason.   
Since async2() gets data as a parameter which is the value that async1() is resolved with, and since async2() returns a promise, we can simply write:

Promises chaining

var promise = async1()

.then(async2);

Here is another demonstration (again, the usage of promises in async1() and async2() is not mandatory and for demonstration purposes only):

Promises chaining example

// Let's imagine those are really asynchronous functions

function async1(value) {

var deferred = $q.defer();

var asyncCalculation = value \* 2;

deferred.resolve(asyncCalculation);

return deferred.promise;

}

function async2(value) {

var deferred = $q.defer();

var asyncCalculation = value + 1;

deferred.resolve(asyncCalculation);

return deferred.promise;

}

var promise = async1(10)

.then(function(x) {

return async2(x);

});

promise.then(function(x) {

console.log(x);

});

First, we call async1(10) which fulfills the promise and resolves it with the value 20. Then the success callback is executed and async2(20) returns a promise that is fulfilled with the value 21. Therefore promiseis resolved with the value 21 and this is what logged.   
A nice thing is that I can write the same example but with more readable code:

Promises chaining – readable

function logValue(value) {

console.log(value);

}

async1(10)

.then(async2)

.then(logValue);

It is easy to see that first we call to async1(), then we call to async2() and at the end we call to logValue(). Each method gets the previous resolved value as a parameter. Naming functions with proper names will also make it easy to understand.   
All the previous examples with promises chaining were pretty optimistic since they all succeeded. But in case a promise is rejected for any reason, the chained promise will also be rejected:

Promises chaining example

// Let's imagine those are really asynchronous functions

function async1(value) {

var deferred = $q.defer();

var asyncCalculation = value \* 2;

deferred.resolve(asyncCalculation);

return deferred.promise;

}

function async2(value) {

var deferred = $q.defer();

deferred.reject('rejected for demonstration!');

return deferred.promise;

}

var promise = async1(10)

.then(function(x) {

return async2(x);

});

promise.then(

function(x) { console.log(x); },

function(reason) { console.log('Error: ' + reason); });

As you can understand from this example, Error: rejected for demonstration! will be logged eventually.   
***Promises can chain promises and are resolved or rejected according to the chained promise (with the chained promise resolve value or reject reason)***.   
Here is a more advanced example of promises chaining:

Promises chaining advanced example

async1()

.then(async2)

.then(async3)

.catch(handleReject)

.finally(freeResources);

In this example we invoked async1(), async2() and async3() one after another, synchronously. In case of any rejection in any one of those methods, the success invocation will stop and handleReject() will occur. At the end, freeResources() will occur no matter of success and failure. For instance, if async2() will return a rejected promise, async3() will not occur and handleReject() will be invoked with the rejection reason of async2(). And at the end freeResources() will occur.

**Useful methods**

$q has several helper methods that can be a great help when using promises. As I said before, other promises implementations have the same methods, probably with a different name.

Sometimes we need to return a rejected promise. Instead of creating a new promise and rejecting it, we can use [$q.reject(reason)](https://docs.angularjs.org/api/ng/service/$q#reject). $q.reject(reason) returns a rejected promise with a reason. Example:

$q.reject(reason) example

var promise = async().then(function(value) {

if (isSatisfied(value)) {

return value;

} else {

return $q.reject('value is not satisfied');

}

}, function(reason) {

if (canRecovered(reason)) {

return newPromiseOrValue;

} else {

return $q.reject(reason);

}

});

If async() is resolved with a satisfied value, the value is chained and thus promise will be resolved with it. If the value is not satisfied, a rejected promise is chained and promise will be rejected.   
If async() is rejected with a reason that can be recovered, a new value or promise will be chained. If the reason cannot be recovered, a rejected promise is chained and eventually promise will be rejected.

Similar to $q.reject(reason), sometimes we need to return a resolved promise with a value. Instead of creating a new promise and resolving it, we can use [$q.when(value)](https://docs.angularjs.org/api/ng/service/$q#when).

Using $q.when(value)

function getDataFromBackend(query) {

var data = searchInCache(query);

if (data) {

return $q.when(data);

} else {

return makeAsyncBackendCall(query);

}

}

In this example I wrote a function that should retrieve a data from my backend. But, before performing the backend call, the function searches the data in the cache. Since I want this function to always return a promise, in case the data is found in the cache, the function returns $q.when(data).   
A cool thing with $q.when(value) is that if value is a 3rd party thenable promise (like jQuery’s Deferred), this method can wrap it and convert it into a $q promise. This way we can easily use other promises implementations with AngularJS.   
$.ajax() of jQuery, for example, returns such thenable promise. The following converts it into angular $q promise:

Using $q.when(jQueryPromise)

var jQueryPromise = $.ajax({

...

...

...

});

var angularPromise = $q.when(jQueryPromise);

Sometimes we need to perform several asynchronous operations, no matter the order, and to be notified when they all done. [$q.all(promisesArr)](https://docs.angularjs.org/api/ng/service/$q#all) can help us with that. Assume we have N methods that return promises: async1(), ..., asyncN(). The following code will log done only when all operations are resolved successfully:

$q.all(promisesArr) example

var allPromise = $q.all([

async1(),

async2(),

....

....

asyncN()

]);

allPromise.then(function(values) {

var value1 = values[0],

value2 = values[1],

....

....

valueN = values[N];

console.log('done');

});

$q.all(promisesArr) returns a promise that is resolved only when all the promises in promisesArr are resolved.  
Keep in mind that if any of the promises is rejected, the resulting promise will be rejected as well.

Up to here we have learned how to create a deferred, how to reject and resolve it and how to get an access to it’s promise. We also seen some useful helping methods that can make our code cleaner and more readable. I think that now is the time for a practical.

**Promises using $q service**

Let’s say we have an amazing application with an amazing registration form. In order to register, a user has to supply his current geolocation coordinates, his photo and a username. To perform the registration action, our backend architecture requires the following from the frontend:

1. Provide geolocation longitude and latitude if possible
2. Upload the user photo to our photos storage server and provide a url of it
3. Reserve the username upon username selection and provide username reservation id

For supporting that, let’s create the following simple functions (I decided to make this separation of functions in order to explain better). Look carefully and see that those methods are asynchronous and this is where promises come in:

**Function that retrieves the current geolocation coordinates**

getGeolocationCoordinates()

function getGeolocationCoordinates() {

var deferred = $q.defer();

navigator.geolocation.getCurrentPosition(

function(position) { deferred.resolve(position.coords); },

function(error) { deferred.resolve(null); }

);

return deferred.promise;

}

getGeolocationCoordinates() declares a deferred and then asks the browser for the current position. Since the geolocation coordinates are not mandatory, both the success and failure callbacks that are provided to navigator.geolocation.getCurrentPosition() resolve the deferred with some result. In case of failure the result will be null. At the end, the deferred’s promise is returned.

**Function that reads a local file and returns it’s content**

readFile()

function readFile(fileBlob) {

var deferred = $q.defer();

var reader = new FileReader();

reader.onload = function () { deferred.resolve(reader.result); };

reader.onerror = function () { deferred.reject(); };

try {

reader.readAsDataURL(fileBlob);

} catch (e) {

deferred.reject(e);

}

return deferred.promise;

}

readFile() gets a file blob (the output of <input type="file"> field) and uses [FileReader](https://developer.mozilla.org/en-US/docs/Web/API/FileReader" \t "_blank) to read it’s content. Before reading the data and returning a promise, readFile() assigned onload and onerror callbacks that resolve and reject the deferred accordingly with the result. Notice that I decided to wrap reader.readAsDataURL(fileBlob); with try {} catch() {} block in order to handle run time exceptions. In case of an exception, the deferred is rejected.

**Function that gets file content and uploads it to files storage**

uploadFile()

function uploadFile(fileData) {

var jQueryPromise = $.ajax({

method: 'POST',

url: '<endpoint for our files storage upload action>',

data: fileData

});

return $q.when(jQueryPromise);

}

Since everyone knows jQuery, I decided to use [$.ajax()](http://www.webdeveasy.com/javascript-promises-and-angularjs-q-service/) in uploadFile(). $.ajax() returns a promise, which is actually what we need. But, this promise is a jQuery’s promise implementation and not $q. Fortunately, here we can use $q.when(value) method, so uploadFile() uses it and returns a promise.

**Function that reserves a username and returns the reservation id**

reserveUsername()

function reserveUsername(username) {

return $http.post('<endpoint for username reservation action>', {

username: username

});

}

Here I used [$http](http://www.webdeveasy.com/javascript-promises-and-angularjs-q-service/) service of AngularJS. $http.post() returns a promise which indicates the post status. This promise is created by $q service inside $http.post() and this will be the return value.

Now that we have all the methods needed for registration, let’s wrap them in a service called appService(you can see the complete app-service.js at the end of this tutorial).

**Application Controller**

Our application controller is pretty simple. It uses $scope, $q and appCervice which are injected in the controller definition. The controller also contains several methods for handling the data (at the end of this tutorial you can find the full source code of this controller).

**Longitude and Latitude**

I don’t want the user to be able to enter values for longitude and latitude, the only way to set values on those fields is by getting the geolocation from the device. Here is a markup of two input elements, they are both bound to model and have a readonly attribute.

Longitude the latitude inputs

<div>

Longitude

<input type="text" readonly="readonly" ng-model="coords.longitude" />

</div>

<div>

Latitude

<input type="text" readonly="readonly" ng-model="coords.latitude" />

</div>

In the controller, all we have to do is to call getGeolocationCoordinates() and set the coordinates data when we get the result:

Handling geolocation

appService.getGeolocationCoordinates()

.then(function setCoords(coordsData) {

$scope.coords = coordsData;

});

**User name**

Here is the markup for user name input. I also added error support by showing the error message and adding an error class in case of an error. Whenever the username input changes, $scope.reserveUsername()is called.

User name input

<div ng-class="{ error: usernameError }">

User Name

<div>

<input type="text" ng-model="username" ng-change="reserveUsername()" />

<div ng-bind="usernameError"></div>

</div>

</div>

$scope.reserveUsername() should use appService to reserve the new username, set username reservation data upon success and set an error upon failure.

Handling user name

var reservationPromise = $q.reject('No username reservation had made');

$scope.reserveUsername = function() {

var newUsername = $scope.username;

reservationPromise = appService.reserveUsername(newUsername)

.then(function setUsernameReservation(reservation) {

$scope.reservation = reservation;

})

.catch(function setUsernameError() {

$scope.usernameError = error;

return $q.reject($scope.usernameError);

});

}

First, reservationPromise is initialized with a rejected promise to handle a case where no reservation will be made.   
When $scope.reserveUsername() happened, a backend reservation occurs. On success, setUsernameReservation() doesn’t return a promise and reservationPromise is resolved (values chaining causes the resulted promise to de resolved). On failure, setUsernameError() returns a rejected promise and so reservationPromise is rejected with an error message (promises chaining causes the resulted promise to be resolved or rejected according to the chained promise).

**User Photo**

The user photo field contains several components: the file input, a placeholder for the selected photo url, a placeholder for the selected image and a placeholder for an error message. I also used here a directive I wrote, named filePathChanged, that triggers a function whenever the user selects a file. You can see the code of the directive down this page.

User photo input

<div ng-class="{ error: photoError }">

Select Photo

<input type="file" file-path-changed="fileSelected(files)">

<span ng-bind="photoError"></span>

<span ng-if="photoUrl" ng-bind="photoUrl"></span>

<img ng-if="photoData" ng-src="{{ photoData }}" />

</div>

Let’s see the implementation of $scope.fileSelected(files).

Handling user image

var photoPromise = $q.reject('No user photo selected');

$scope.fileSelected = function(files) {

if (files && files.length > 0) {

var filePath = files[0];

photoPromise = appService.readFile(filePath)

.then(function setPhotoData(photoData) {

$scope.photoData = photoData;

return photoData;

})

.then(appService.uploadFile)

.then(function setPhotoUrl(photoUrl) {

$scope.photoUrl = photoUrl;

})

.catch(function setPhotoError(error) {

$scope.photoError = 'An error has occurred: ' + error;

return $q.reject($scope.photoError);

});

}

};

This code is simple. First we verify that a file is supplied. Next we read the file using appService.readFile()and set the photo data in a model. Then we upload the file data, get a url for the photo and set the photo url in a model. In case of any error we set the error message in a model and chain rejected promise.

**Registration**

Our last step is the registration button. We have to create a function that collects the longitude and latitude, the selected photo url and the reservation id of the chosen username. In any case of an error with the username reservation or the photo handling, this function has to reflect an error message. Remember, the longitude and latitude are not mandatory so if they are not available at the time of the registration – it will not stop the process.

Register() method

$scope.register = function() {

$q.all([

reservationPromise,

photoPromise

]).then(function doRegistrationCall() {

var longitude = $scope.data.coords && $scope.data.coords.longitude;

var latitude = $scope.data.coords && $scope.data.coords.latitude;

var reservationId = $scope.data.reservation.token;

var photoUrl = $scope.data.photoUrl;

doRegistration(longitude, latitude, reservationId, photoUrl);

}, function setSubmitError(error) {

$scope.submitError = error;

});

};

Here we used $q.all() because we want to perform an operation after username reservation and photo handling are both done. In case of any rejection we mark that the registration failed by assigning submitError model. In this example, doRegistration() is a method that does the registration call to the backend.

That’s all! our registration process is now complete.   
Here is the full source of our small application.

**app-service.js**

app-service.js

window.module.factory('appService', ['jquery', '$http', '$q', function($, $http, $q) {

function getGeolocationCoordinates() {

var deferred = $q.defer();

navigator.geolocation.getCurrentPosition(

function(position) { deferred.resolve(position.coords); },

function(error) { deferred.resolve(null); }

);

return deferred.promise;

}

function readFile(fileBlob) {

var deferred = $q.defer();

var reader = new FileReader();

reader.onload = function () { deferred.resolve(reader.result); };

reader.onerror = function () { deferred.reject(); };

try {

reader.readAsDataURL(fileBlob);

} catch (e) {

deferred.reject(e);

}

return deferred.promise;

}

function uploadFile(fileData) {

// var jQueryPromise = $.ajax({

// method: 'POST',

// url: '<endpoint for our files storage upload action>',

// data: fileData

// });

var deferred = $.Deferred();

setTimeout(function() {

deferred.resolve('www.myimage.com/123');

}, 200);

var jQueryPromise = deferred.promise();

return $q.when(jQueryPromise);

}

var reserveCount = 0;

function reserveUsername(username) {

// return $http.post('<endpoint for username reservation action>', {

// username: username

// });

var deferred = $q.defer();

setTimeout(function() {

if (reserveCount > 0 && reserveCount % 3 === 0) {

deferred.reject('error reserving "' + username + '"');

} else {

var token = 'token' + reserveCount;

deferred.resolve({

token: token,

username: username

});

}

reserveCount ++;

}, 300);

return deferred.promise;

}

return {

getGeolocationCoordinates: getGeolocationCoordinates,

readFile: readFile,

uploadFile: uploadFile,

reserveUsername: reserveUsername

};

}]);

Note: in order to mimic uploadFile() and reserveUsername() without implementing a backend, I’ve created a custom code that sometimes is resolved and sometimes is rejected.   
Now we can proceed with the controller implementation.

**app-controller.js**

app-controller.js

window.module.controller('appController', ['$scope', '$q', 'appService', function($scope, $q, appService) {

$scope.data = { errors: { } };

function setCoords(coordsData) {

$scope.data.coords = coordsData;

}

function setPhotoData(photoData) {

return $scope.data.photoData = photoData;

}

function setPhotoUrl(photoUrl) {

return $scope.data.photoUrl = photoUrl;

}

function clearPhotoError() {

delete $scope.data.errors.photo;

}

function setPhotoError(error) {

$scope.data.errors.photo = 'An error has occurred: ' + error;

return $q.reject($scope.data.errors.photo);

}

function clearUsernameError() {

delete $scope.data.errors.username;

}

function setUsernameError(error) {

$scope.data.errors.username = error;

return $q.reject($scope.data.errors.username);

}

function setUsernameReservation(reservation) {

$scope.data.reservation = reservation;

}

function setSubmitError(error) {

$scope.data.errors.submit = error;

}

function clearSubmitError() {

delete $scope.data.errors.submit;

}

function doRegistration(longitude, latitude, reservationId, photoUrl) {

$scope.data.success = true;

$scope.storedJSON = JSON.stringify({

longitude: longitude,

latitude: latitude,

reservationId: reservationId,

photoUrl: photoUrl

});

}

appService.getGeolocationCoordinates()

.then(setCoords);

var photoPromise = $q.reject('No user photo selected');

$scope.fileSelected = function(files) {

if (files && files.length > 0) {

var filePath = files[0];

clearPhotoError();

photoPromise = appService.readFile(filePath)

.then(setPhotoData)

.then(appService.uploadFile)

.then(setPhotoUrl)

.catch(setPhotoError);

}

};

var reservationPromise = $q.reject('No username reservation had made');

$scope.reserveUsername = function() {

var newUsername = $scope.data.username;

clearUsernameError();

reservationPromise = appService.reserveUsername(newUsername)

.then(setUsernameReservation)

.catch(setUsernameError);

}

$scope.register = function() {

$q.all([

reservationPromise,

photoPromise

]).then(function() {

var longitude = $scope.data.coords && $scope.data.coords.longitude;

var latitude = $scope.data.coords && $scope.data.coords.latitude;

var reservationId = $scope.data.reservation.token;

var photoUrl = $scope.data.photoUrl;

clearSubmitError();

doRegistration(longitude, latitude, reservationId, photoUrl);

}, function(error) {

setSubmitError(error);

});

};

}]);

**index.html**

index.html

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<title></title>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js"></script>

<script type="text/javascript">

window.jQuery || document.write('<script src="/scripts/libs/jquery.js"><\/script>');

</script>

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.14/angular.min.js"></script>

<script type="text/javascript">

window.angular || document.write('<script src="/scripts/libs/angular.js"><\/script>');

</script>

<link rel="stylesheet" href="/style/semantic.css" />

<link rel="stylesheet" href="/style/app.css" />

</head>

<body ng-app="demo-app">

<form class="ui form segment" ng-controller="appController">

<div class="two fields">

<div class="field">

<label for="longitude">Longitude</label>

<input id="longitude" type="text" readonly="readonly" ng-model="data.coords.longitude" placeholder="No Longitude" />

</div>

<div class="field">

<label for="latitude">Latitude</label>

<input id="latitude" type="text" readonly="readonly" ng-model="data.coords.latitude" placeholder="No Latitude" />

</div>

</div>

<div class="field username" ng-class="{ error: data.errors.username }">

<label for="username">User Name</label>

<div class="ui labeled icon input">

<input id="username" type="text" ng-model="data.username" ng-change="reserveUsername()" placeholder="User Name" />

<div class="ui red label pointing above" ng-bind="data.errors.username"></div>

<i class="circular ban circle icon"></i>

<i class="circular checkmark icon" ng-if="data.reservation"></i>

<div class="ui corner label">

<i class="icon asterisk"></i>

</div>

</div>

</div>

<div class="inline field user-photo" ng-class="{ error: data.errors.photo }">

<label for="file" class="ui icon button">

<i class="file icon"></i>

Select Photo

</label>

<input type="file" id="file" file-path-changed="fileSelected(files)">

<span class="ui red label" ng-bind="data.errors.photo"></span>

<span class="ui green label" ng-if="data.photoUrl" ng-bind="data.photoUrl"></span>

<div class="ui segment" ng-if="data.photoData">

<img class="rounded ui image" ng-src="{{ data.photoData }}" />

</div>

</div>

<div class="field">

<div class="ui blue submit button" ng-click="register()">Register</div>

</div>

<div class="field">

<span class="ui red label" ng-if="data.errors.submit" ng-bind="data.errors.submit"></span>

<span class="ui green label" ng-if="data.success">

Registration Seccess with {{data.coords.longitude ? 'longitude =' + data.coords.longitude : 'no longitude' }},

{{data.coords.latitude ? 'latitude =' + data.coords.latitude : 'no latitude' }},

username = {{data.username}}, photo url = {{data.photoUrl}}

</span>

</div>

</form>

<script type="text/javascript" src="scripts/module.js"></script>

<script type="text/javascript" src="scripts/directives.js"></script>

<script type="text/javascript" src="scripts/app-service.js"></script>

<script type="text/javascript" src="scripts/app-controller.js"></script>

</body>

</html>

Here I used [a nice CSS framework](http://semantic-ui.com/) named *Semantic UI* in order to produce a better looking UI. Therefore this markup contains many classes and other elements.

**directives.js**

We only have one directive named filePathChanged.

directives.js

window.module.directive('filePathChanged', function() {

return {

restrict: 'A',

scope: {

filePathChanged: '&'

},

link: function (scope, element, attrs) {

element.bind('change', function() {

scope.filePathChanged({ files: element.prop('files') });

});

}

};

});

**Summary**

After reading this article you should know by now that working with callbacks might make a hard life especially when synchronization is needed (parallel and sequentially executing). You were introduced with the deferrers and promises solution and saw how to use it. Through this article you saw explanations and examples of important promises methods and learned about promises chaining. At the end you got a practical tutorial. Here is a short list to summarize the methods mentioned in this article:

* var deferred = $q.defer(); – creates a new deferred
* deferred.resolve(value); – resolves a deferred with a value
* deferred.reject(reason); – rejects a deferred with a reason
* var promise = deferred.promise; – gets a promise from deferred
* promise.then(success, failure); – assigns callbacks for success (resolve) and failure (reject)
* promise.catch(failure); – assigns failure callback (equals to promise.then(null, failure))
* promise.finally(always); – assign a callback to be called both on success or failure
* var promise = $q.reject(reason); – returns rejected promise with a reason
* var promise = $q.when(valueOrPromise); – wraps value or other implementation of thenable promise with AngularJS promise
* var promise = $q.all(promisesArr); – returns a promise that will be resolved only when all promises in `promisesArr` are resolved

<http://plnkr.co/edit/3Eoq0zmv5NnOpHvyccaS?p=preview>

Here are three additional links:

* [The tutorial application in action](http://www.webdeveasy.com/code/javascript-promises-and-angularjs-q-service/index.html)
* [The tutorial source on GitHub](https://github.com/naorye/angulajs-q-service-tutorial)
* [The tutorial source zip file](https://github.com/naorye/angulajs-q-service-tutorial/archive/master.zip)