

APPMOB - Asynchronous Javascript

Olivier Liechti & Simon Oulevay
COMEM Applications Mobiles

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Javascript code execution is single-threaded.

Therefore, all time-consuming operations are always asynchronous, or it would block the UI:

- HTTP Requests
- Geolocation Requests
- Web Workers
- Etc.

Asynchronous Callbacks

With **synchronous** code,
the result is immediately
available.

```
var items = getItem();  
  
// do something with items
```

With **asynchronous** code,
you are given the result
when it is ready, typically in
a **callback function**.

```
downloadItems(function(items) {  
    // do something with items  
});
```

Using Callback Functions

It doesn't matter how you declare a callback function. The three following examples have exactly the same effect.

```
downloadItems(function(items) {  
    // do something with items  
});
```

```
function callback(items) {  
    // do something with items  
}
```

```
downloadItems(callback);
```

```
var callback = function(items) {  
    // do something with items  
};
```

```
downloadItems(callback);
```

Do not forget that **asynchronous code does not execute in order!**

```
var a = "a";  
console.log(a);
```

```
setTimeout(function() {  
  a = a + "b";  
  console.log(a);  
, 1000);
```

```
a = a + "c";  
console.log(a);
```



"a"
"ac"
"acb"

```
downloadItems(function(items) {  
  console.log("Items downloaded");  
});
```

```
console.log("Yeehaw!");
```



"Yeehaw!"
"Items downloaded"

There are two main patterns when it comes to handling errors with asynchronous operations:

- Error Callbacks
- Node.js-style Callbacks

With this pattern, you give two functions to the asynchronous operation: a **success callback**, and an **error callback**. **Only one of them will be called**, depending on whether the operation is successful or not.

Here are two equivalent ways to write it:

```
downloadItems(function(items) {  
    // do something with items  
}, function(error) {  
    // or handle the error  
});
```

```
function onSuccess(items) {  
    // do something with items  
}
```

```
function onError(error) {  
    // or handle the error  
}
```

```
downloadItems(onSuccess, onError);
```

Error Callback Implementation

This is how you could write the **downloadItems** asynchronous function in an AngularJS application to use the error callback pattern.

```
function downloadItems(successCallback, errorCallback) {  
  $http({  
    method: "GET",  
    url: "http://example.com"  
  }).success(function(data) {  
    successCallback(data);  
  }).error(function(error) {  
    errorCallback(error);  
  });  
}
```

If you don't transform the data in this function, you can also give the callbacks directly to **\$http**.

```
function downloadItems(successCallback, errorCallback) {  
  $http({  
    method: "GET",  
    url: "http://example.com"  
  }).success(successCallback).error(errorCallback);  
}
```


With this pattern, you only give one callback function to the asynchronous operation. It has **two arguments: the error**, and **the result**. When the operation is successful, the error will not be set.

Here are two equivalent ways to write it:

```
downloadItems(function(error, items) {  
  if (error) {  
    // handle the error  
  } else {  
    // do something with items  
  }  
});
```

```
function callback(error, items) {  
  if (error) {  
    // or handle the error  
  } else {  
    // do something with items  
  }  
}  
  
downloadItems(callback);
```

Node.js-style Callback Implementation

This is how you could write the **downloadItems** asynchronous function in an AngularJS application to use the Node.js-style callback pattern.

```
function downloadItems(callback) {  
  $http({  
    method: "GET",  
    url: "http://example.com"  
  }).success(function(data) {  
    callback(null, data);  
  }).error(function(error) {  
    callback(error);  
  });  
}
```

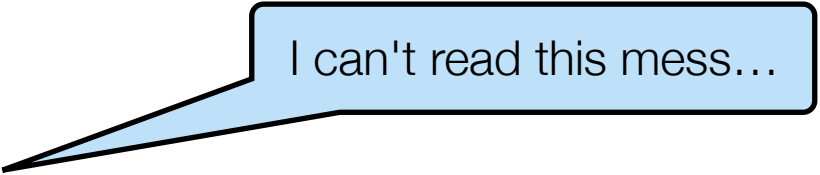
Remember, with a Node.js-style callback, the callback has **two arguments: the error** and **the result**.
If there's no error, the first argument should be **null**.

If there's an error, we only need to give the first argument to the callback: **the error**.

Callback Hell

This is what often happens when you don't have the right tools to manage callbacks:

```
var item = {};  
  
downloadItemTypes(function(types) {  
    item.type = types[0];  
  
    takePicture(function(imageData) {  
  
        uploadPicture(imageData, function(imageUrl) {  
            item.imageUrl = imageUrl;  
  
            postItem(item, function(createdItem) {  
                // phew, we made it...  
            }, function(error) {  
                console.log("Could not create item: " + error);  
            });  
        }, function(error) {  
            console.log("Could not upload picture: " + error);  
        });  
    }, function(error) {  
        console.log("Could not take picture: " + error);  
    });  
}, function(error) {  
    console.log("Could not download items: " + error);  
});
```



I can't read this mess...

A control flow library is one solution to the **callback hell** problem.

The **async** library is an example. It was originally developed for Node.js, so it uses **Node.js-style callbacks**.

<https://github.com/caolan/async/>

With **async**, you can easily run successive asynchronous operations.

```
var item = {};  
  
function takePicture(callback) {  
  camera.getPhoto(function(imageData) {  
    callback(null, imageData);  
  }, function(error) {  
    callback(error);  
  });  
}  
  
function uploadPicture(imageData, callback) {  
  // ...  
  callback(null, imageUrl);  
}  
  
function postItem(imageUrl, callback) {  
  // ...  
  callback(null, createdItem);  
}
```

Each asynchronous operation will pass its result to the next.

You give async a **completion callback**. If any of the operations fail, the **error** will be sent to your completion callback. If all operations succeed, you will receive the result of the last one.

```
function onComplete(err, result) {  
  if (error) {  
    // handle the error  
  } else {  
    var createdItem = result;  
    // do something with the item  
  }  
}  
  
async.waterfall([  
  takePicture,  
  uploadPicture,  
  postItem  
, onComplete];
```

Async will run each operation **in turn** and call the completion callback at the end or as soon as there's an error.

You can also easily run functions in parallel.

You give async a **completion callback**. If any of the parallel operations fail, the **error of the first to fail** will be sent to your completion callback. If all operations succeed, you will receive an array of the results of each operation.

```
function downloadFile(callback) {
  $http({ ... }).success(function(data) {
    callback(null, file);
  }).error(function(error) {
    callback(error);
  });
}

function downloadImage(callback) {
  // ...
}

function downloadThumbnail(callback) {
  // ...
}
```

```
function onComplete(err, results) {
  if (error) {
    // handle the error
  } else {
    var file = results[0];
    var image = results[1];
    var thumbnail = results[2];
    // do something with the results
  }
}
```

```
async.parallel([
  downloadFile,
  downloadImage,
  downloadThumbnail
], onComplete);
```

Async will run your
three operations **in
parallel**.

Async supports much more.
Read the documentation.

› Control Flow

- `series`
- `parallel`
- `parallelLimit`
- `whilst`
- `dowhilst`
- `until`
- `doUntil`
- `forever`
- `waterfall`
- `compose`
- `seq`
- `applyEach`
- `applyEachSeries`
- `queue`
- `priorityQueue`
- `cargo`
- `auto`
- `retry`
- `iterator`
- `apply`
- `nextTick`
- `times`
- `timesSeries`

Collections

- `each`
- `eachSeries`
- `eachLimit`
- `map`
- `mapSeries`
- `mapLimit`
- `filter`
- `filterSeries`
- `reject`
- `rejectSeries`
- `reduce`
- `reduceRight`
- `detect`
- `detectSeries`
- `sortBy`
- `some`
- `every`
- `concat`
- `concatSeries`

Promises are another solution to the **callback hell** problem.

<https://promisesaplus.com/>

What is a Promise?

A promise represents the **eventual result** of an **asynchronous operation**.

The primary way of interacting with a promise is through its **then** method, which registers **callbacks** to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

Promise Example

Let's say that **downloadItems** returns a promise. The promise is **not the actual result** of the operation. It is the **promise of a result at some point in the future.**

```
var promise = downloadItems();
```

```
function onFulfilled(items) {  
  // do something with items  
}
```

A promise can either be **fulfilled...**

```
function onRejected(error) {  
  // or handle the error  
}
```

... or **rejected.**

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

To know when the promise is fulfilled or rejected, call the **then** method with the appropriate callbacks. The first callback function will be called if the promise is fulfilled, the second if the promise is rejected.

How Do I Create a Promise?

Some libraries are already based on promises.

For example, Angular's **\$http** service already returns a promise. Instead of using **success** and **error**, use **then**:

```
var promise = $http({
  url: "http://example.com"
});

function onFulfilled(items) {
  // do something with items
}

function onRejected(error) {
  // or handle the error
}

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

How Do I Create a Promise?

If you have an asynchronous call which doesn't return a promise and you want to use one, you need a promise library like **q**:

<https://github.com/krisowal/q/>

Note that AngularJS already includes a **lightweight version of q**:

[https://docs.angularjs.org/api/ng/service/\\$q/](https://docs.angularjs.org/api/ng/service/$q/)

How Do I Create a Promise?

```
.factory("CameraService", function($q) {  
  return {  
    getPicture: function(options) {  
      var deferred = $q.defer();  
  
      navigator.camera.getPicture(function(result) {  
        deferred.resolve(result);  
      }, function(err) {  
        deferred.reject(err);  
      }, options);  
  
      return deferred.promise;  
    }  
  }  
});
```

Create a **deferred object**. This object contains a promise and can be used to resolve or reject that promise.

Inject the **\$q** service.

Call the **asynchronous operation**.

If the operation is successful, **fulfil the promise** with the result.

Otherwise, **reject the promise** with the error.

Now you can call it, get the promise and use its **then** method.

```
var options = { ... };  
CameraService.getPicture(options).then(function(result) {  
  // do something with the result  
}, function(error) {  
  // handle the error  
});
```

Promise Chains

Promises can be chained.
The functions will be executed one after the other.

```
takePicture().then(uploadPicture).then(postItem);
```

Note that here we are **calling a function which returns a promise**.

But here, we are **NOT calling the next function**. We are giving it to the **then** function of the promise. **uploadPicture** will be called after the first promise has been resolved.

```
takePicture().then(uploadPicture).then(postItem).then(function(result) {  
    // do something with the result  
}, function(error) {  
    // or handle the error  
});
```

You can also add an **error callback** at the end. Any error in the chain will interrupt the chain and end up here.

You can add a **fulfilment callback** at the end of the promise chain.

Error Handling

Fulfilled!

Fulfilled!

Fulfilled!

```
takePicture().then(uploadPicture).then(postItem).then(function(result) {  
    // do something with the result  
}, function(error) {  
    // or handle the error  
});
```

The **fulfilment**
callback is called.

Fulfilled!

Rejected!

This is **not**
called.

```
takePicture().then(uploadPicture).then(postItem).then(function(result) {  
    // do something with the result  
}, function(error) {  
    // or handle the error  
});
```

The **error callback**
is called.

Exceptions are **automatically caught** by the promise chain.

Fulfilled!

An **error** is thrown!

This is **not**
called.

```
takePicture().then(uploadPicture).then(postItem).then(function(result) {  
    // do something with the result  
}, function(error) {  
    // or handle the error  
});
```

The **error callback** is called with
the error that was thrown.

Parallel Execution with Promises

You can also easily run asynchronous operations **in parallel** with promises:

```
function downloadFile() {  
    return $http({ ... });  
}  
  
function downloadImage() {  
    // ...  
    return deferred.promise;  
}  
  
function downloadThumbnail() {  
    // ...  
    return deferred.promise;  
}
```

```
$q.all([  
    downloadFile(),  
    downloadImage(),  
    downloadThumbnail()  
]).then(function(results) {  
    // you receive an array of the results  
}, function(error) {  
    // handle the error  
});
```