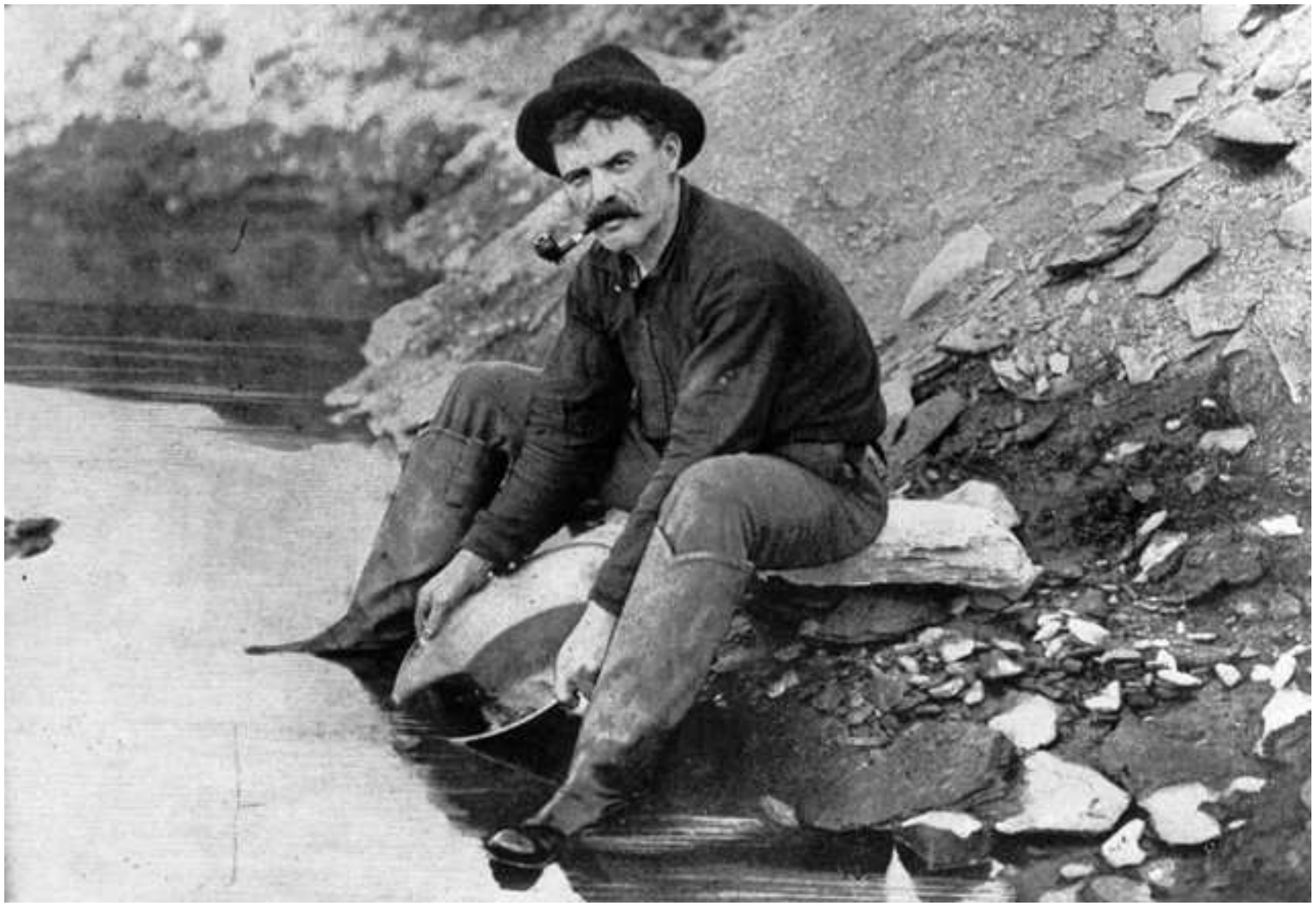
Lecture 3: async programming in JS

Olivier Liechti
TWEB

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



GitHub Explorer - iteration 3

Project - 2 features for the iteration

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

As a user, I can give provide a GitHub repo, user or organization and see "some" analysis (TBD)

Feature

What can I do with the GitHub API?

What do I want to do with the GitHub API? What is the question I want to answer? What is the data I need to get?

How do I make REST API calls from my AngularJS app?

How do I transform the data provided by the GitHub API so that I can display / graph them?

Validate, update Heroku. Validate.

API

Overview

This describes the resources that make up the official GitHub API v3. If you have any problems or requests please contact support.

Current Version

mikedeboer/node-github: node

Olivier

GitHub, Inc. [US] https://github.com/mikedeboer/node-github

This repository Search Pull requests Issues Gist

promise 1 of 1

mikedeboer / node-github

Watch 60 Star 1,543 Fork 371

Code Issues 11 Pull requests 3 Projects 0 Wiki Pulse Graphs

node library to access the GitHub API http://mikedeboer.github.com/node-github/

725 commits 5 branches 47 releases 101 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

kaizensoze edit deprecated notice ... Latest commit 25d4466 6 days ago

doc	add preview merge params to edit user repo endpoint	7 days ago
examples	add basic getNextPage example	6 days ago
lib	add preview merge params to edit user repo endpoint	7 days ago
templates	Add installations endpoint	9 days ago
test	add preview merge params to edit user repo endpoint	7 days ago
.gitignore	put apidoc back under gitignore	9 days ago
.npmignore	change filenames to camel case	5 months ago
CHANGELOG.md	bump to 3.1.1 and update changelog	9 days ago
CONTRIBUTING.md	Don't require contributors to update apidoc.	3 months ago
ISSUE_TEMPLATE.md	add issue template	6 days ago
LICENSE	move from https://github.com/c9/node-github3 to https://github.com/aj...	5 years ago
README.md	edit deprecated notice	6 days ago
package.json	bump to 3.1.1 and update changelog	9 days ago

README.md

Today's agenda

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

15:45 - 16:05	**20'**	**Intro to async programming**
16:05 - 16:20	15'	Calling REST APIs from AngularJS
16:20 - 16:35	16'	Async programming with promises
16:35 - 18:05	90'	Group work on this iteration's features

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

How can I execute multiple asynchronous operations in sequence?

Asynchronous Programming Techniques

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- We have already seen that JavaScript relies on **asynchronous programming**:
  - The JS engine is **single-threaded.** For this reason, IO operations **have to be non-blocking**.
  - An **event loop** is used both in the browser and on the server (node.js):
    - As the program executes, events are added to a queue. Every event has an associate callback function.
    - A dispatcher takes the next event in the queue and invokes the callback function (on the single thread).
    - When the callback function returns, the dispatcher takes the next event in the queue, and continues forever (it's an event **loop**).

Asynchronous Programming Techniques

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
setTimeout( function() {
  console.log("the callback has been invoked");
}, 2000);
```

An event will be added to the queue in 2000 ms. In other words, the function passed as the first argument will be invoked in 2 seconds or more (the thread might be busy when the event is posted...).

```
fs.readFile('/etc/passwd', function (err, data)
{
  if (err) throw err;
  console.log(data);
});
```

An event will be added when the file has been fully read (in a non-blocking way). When the event is taken out of the queue, the callback function has access to the file content (data).

Asynchronous Programming Techniques

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
$(document).mousemove(function(event){
  $("span").text(event.pageX + ", " +
event.pageY);
});
```

An event will be added to the queue whenever the mouse moves. In each case, the callback function has access to the event attributes (coordinates, key states, etc.).

```
$.get( "ajax/test.html", function( data ) {
$( ".result" ).html( data );
alert( "Load was performed." );
});
```

An event will be added when the AJAX request has been processed, i.e. when a response has been received. The callback function has access to the payload.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- The principle of passing a callback function when invoking an asynchronous operation is pretty straightforward.
- Things get more tricky as soon as you want to **coordinate multiple tasks**. Consider this simple example...

Do this first...

... when done, do this.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

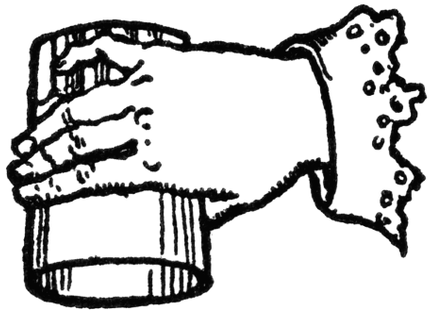
- A first attempt...

```
var milkAvailable = false;

function milkCow() {
  console.log("Starting to milk cow...");
  setTimeout(function() {
    console.log("Milk is available.");
    milkAvailable = true;
  }, 2000);
}

milkCow();
console.log("Can I drink my milk? (" + milkAvailable + ")");
```

FAIL



Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Fixing the issue with a callback...

```
var milkAvailable = false;

function milkCow(done) {
  console.log("Starting to milk cow...");
  setTimeout(function() {
    console.log("Milk is available.");
    milkAvailable = true;
    done();
  }, 2000);
}

milkCow( function() {
  console.log("Can I drink my milk? (" + milkAvailable + ")");
});
```

SUCCESS

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Ok... but what happens when I have **more than 2 tasks** that I want to execute in **sequence**?
- Let's say we want to have the **sequence** B, C, D, X, Y, Z, E, F, where X, Y and Z are asynchronous tasks.

```
function f() {
  syncB();
  syncC();
  syncD();
  asyncX();
  asyncY();
  asyncZ();
  syncE();
  syncF();
}
```

```
B result available
C result available
D result available
E result available
Z result available
Y result available
F result available
X result available
```

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Ok... but what happens when I have **more than 2 tasks** that I want to execute in sequence?
- Let's say we want to have the sequence B, C, D, X, Y, Z, E, F, where X, Y and Z are asynchronous tasks.

```
function f() {
  syncB();
  syncC();
  syncD();
  asyncX(function() {
    asyncY(function() {
      asyncZ(function() {
        syncE();
        syncF();
      });
    });
  });
}
```

```
B result available
C result available
D result available
X result available
Y result available
Z result available
E result available
F result available
```

But welcome to the "callback hell" aka "callback pyramid"

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Now, let's imagine that we have 3 asynchronous tasks. We want to invoke them in parallel and **wait until all of them complete**.
- Typical use case: you want to send several AJAX requests (to get different data models) and update your DOM once you have received all responses.

```
function f(done) {

 async1(function(r1) {
 });
  async2(function(r2) {
 });
  async3(function(r3) {
 });

  done();

}
```

Double fail: not only do I invoke done() too early, but also I don't have any result to send back...

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Now, let's imagine that we have 3 asynchronous tasks. We want to invoke them in parallel and **wait until all of them complete**.
- Typical use case: you want to send several AJAX requests (to get different data models) and update your DOM once you have received all responses.

```
function f(done) {
  var numberOfPendingTasks = 3;
  var results = [];

  function reportResult(result) {
    results.push(result);
    numberOfPendingTasks -= 1;
    if (numberOfPendingTasks === 0) {
      done(null, results);
    }
  }
  async1(function(r1) {
    reportResult(r1);
  });
  async2(function(r2) {
    reportResult(r2);
  });
  async3(function(r3) {
    reportResult(r3);
  });
}
```

When this reaches 0, I know that all the tasks have completed. I can invoke the "done" callback function that I received from the client. I can pass the array of results to the function.

When a task completes, it invokes this function and passes its result. The result is added to the array and the number of pending tasks is decremented.

The three tasks are asynchronous, so they pass their own callback functions and receive a result when the operation completes.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

A beer

The **promise** of a beer

Async libs to the rescue

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Different approaches and libraries have been proposed to make it easier to write asynchronous code.

then **Promises/A+**

An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

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.

https://github.com/promises-aplus/promises-spe

http://documentup.com/kriskowal/q/

npm

Search Packages

async ☆

Higher-order functions and common patterns for asynchronous code

```
$ npm install async
```

Want to see pretty graphs? Log in now!

248 314	downloads in the last day
1 394 801	downloads in the last week
5 831 057	downloads in the last month

jQuery
write less, do more.

Deffered objects

Promises

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- "A **promise** must be in **one of three states**: pending, fulfilled, or rejected.
- When **pending**, a promise:
  - may transition to either the fulfilled or rejected state.
- When **fulfilled**, a promise:
  - **must not transition** to any other state.
  - must have a **value**, which must not change.
- When **rejected**, a promise:
  - **must not transition** to any other state.
  - must have a **reason**, which must not change."

then **Promises/A+**

An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

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.

https://github.com/promises-aplus/promises-spec

Promises

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **“A promise must provide a `then` method to access its current or eventual value or reason.**
- A promise's **then** function accepts two arguments:
  - `promise.then(onFullfilled, onRejected)`
- If `onFulfilled` is a function:
  - it must be called after promise is fulfilled, with promise's value as its first argument.
  - it must not be called before promise is fulfilled.
  - it must not be called more than once.
- If `onRejected` is a function:
  - it must be called after promise is rejected, with promise's reason as its first argument.
  - it must not be called before promise is rejected.
  - it must not be called more than once”

then **Promises/A+**

An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

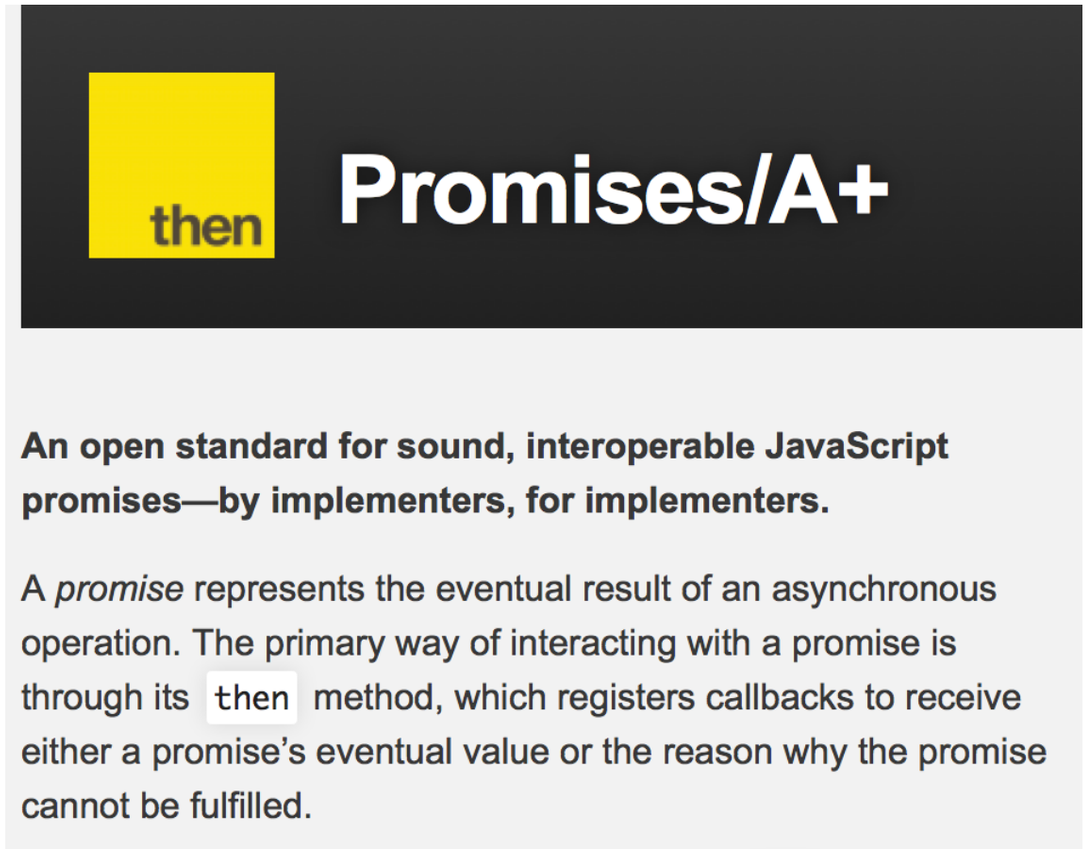
A *promise* represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its `then` method, which registers callbacks to receive either a promise’s eventual value or the reason why the promise cannot be fulfilled.

https://github.com/promises-aplus/promises-spec

Promises

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **“`then` must return a promise [3.3].**
  - `promise2 = promise1.then(onFulfilled, onRejected);`
- If either `onFulfilled` or `onRejected` returns a value x, run the Promise Resolution Procedure [[Resolve]](promise2, x).
- If either `onFulfilled` or `onRejected` throws an exception e, promise2 must be rejected with e as the reason.
- If `onFulfilled` is not a function and promise1 is fulfilled, promise2 must be fulfilled with the same value as promise1.
- If `onRejected` is not a function and promise1 is rejected, promise2 must be rejected with the same reason as promise1.”



https://github.com/promises-aplus/promises-spec

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Promises in AngularJS

How do I invoke a REST API?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- AngularJS provides two native services for making HTTP calls:
  - **$http**
  - $resource
- The second one abstracts some of the common patterns implemented by REST APIs (URL structure, verbs, etc). Personally, I prefer to use the first one.
- When you invoke a REST API from your AngularJS app, the access must be granted with respect to the same origin policy
  - Today, many REST APIs implement the CORS protocol (you won't have anything to do on the angular side).
  - Older APIs use a mechanism called JSONP. The $http service provides a function to make JSONP calls. Pay attention to the URL you provide.

Invoke a REST API

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- The following call returns a **promise**. When it resolves, you receive an object that contains the payload **data**, as well as the response metadata. This is code that you will typically write in your AngularJS **services**.

```
$http.jsonp('http://prost.herokuapp.com/api/v1/beer/rand?
callback=JSON_CALLBACK')
```

- In your **controller**, you will write something like:

```
myService.callMyRESTAPI()
  .then( function(data) {
     vm.data = data
  } );
```

- And in your templates, something like:

```
{{ vm.data.field }}
```

How do I invoke endpoints in sequence?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Very often, you must make several API calls in sequence. For instance:
  - You make a first call to retrieve an Employee. In the Empoyee payload, you have a Company ID.
  - You make a second call to retrieve the Company.
  - You want to return an object with the complete employee + company data.
- To do that, you can **chain promises**.

```
(function () {
  'use strict'

  angular
    .module('beersapi')
    .factory('beersapiService', Beersapi);

  Beersapi.$inject = ['$http'];

  function Beersapi($http) {
    return {
      fetchBeer: fetchBeer
    }

    function fetchRandomBeer() {
      return $http.jsonp('http://prost.herokuapp.com/api/v1/beer/rand?callback=JSON_CALLBACK')
        .then(function (response) {
          return response.data;
        })
    }

    function fetchBrewery(beer) {
      if (beer.brewery === undefined || beer.brewery.key == undefined) {
        return {
          beer: beer,
          brewery: {}
        }
      }
      return $http.jsonp('http://prost.herokuapp.com/api/v1/brewery/' + beer.brewery.key + "?callback=JSON_CALLBACK")
        .then(function (response) {
          return {
            beer: beer,
            brewery: response.data
          }
        })
    }

    function fetchBeer() {
      return fetchRandomBeer()
        .then(fetchBrewery);
    }
  }
})();
```

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

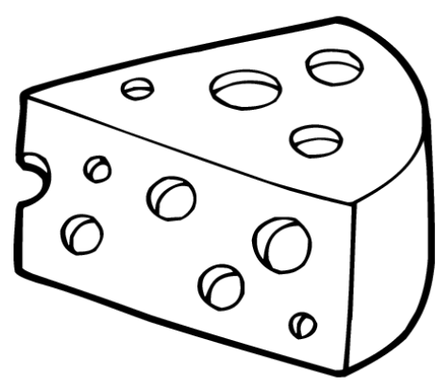
Back to the farm...

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- The principle of passing a callback function when invoking an asynchronous operation is pretty straightforward.
- Things get more tricky as soon as you want to **coordinate multiple tasks**. Consider this simple example...

First get milk...



... then make cheese...

... then sell it.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Let's prepare the individual tasks...

```
function milkCow( callbackWhenMilkIsAvailable ) {
    console.log("Start to milk cow...");
    setTimeout( function() {
        console.log("Done milking cow.");
        callbackWhenMilkIsAvailable(null, "MILK");
    }, 5000);
};
```

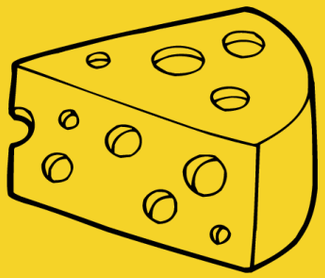
The first parameter is the **error** (if one happened) and the second one is the **result**.

Instead of calling this parameter "**callback**" or "**cb**" (like most developers), I like to use **explicit names**. It makes code easier to read, especially when you have nested functions.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
function prepareCheese( milk, callbackWhenCheeseIsAvailable ) {
    console.log("Start preparing cheese with " + milk);
    setTimeout( function() {
        console.log("Done preparing cheese.");
        callbackWhenCheeseIsAvailable(null, "CHEESE");
    }, 3000);
};
```



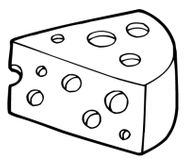
```
function sellCheese( cheese, callbackWhenCheeseHasBeenSold ) {
    console.log("Start selling " + cheese);
    setTimeout( function() {
        console.log("Done selling " + cheese);
        callbackWhenCheeseHasBeenSold(null, "MONEY");
    }, 1000);
};
```

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
milkCow( function(err, milk) {
    console.log("I have " + milk + " and can prepare cheese.");
    prepareCheese(milk, function(err, cheese) {
        console.log("I have now " + cheese + " and can sell it.");
        sellCheese(cheese, function(err, money) {
            console.log("Youpi! I have my money.");
        });
    });
});
```

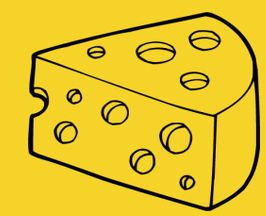
```
$ node promise.js
Start to milk cow...
Done milking cow.
I have now some MILK and can prepare cheese.
Start preparing cheese with MILK
Done preparing cheese.
I have now CHEESE and can sell it.
Start selling CHEESE
Done selling CHEESE
Youpi! I have my money.
```



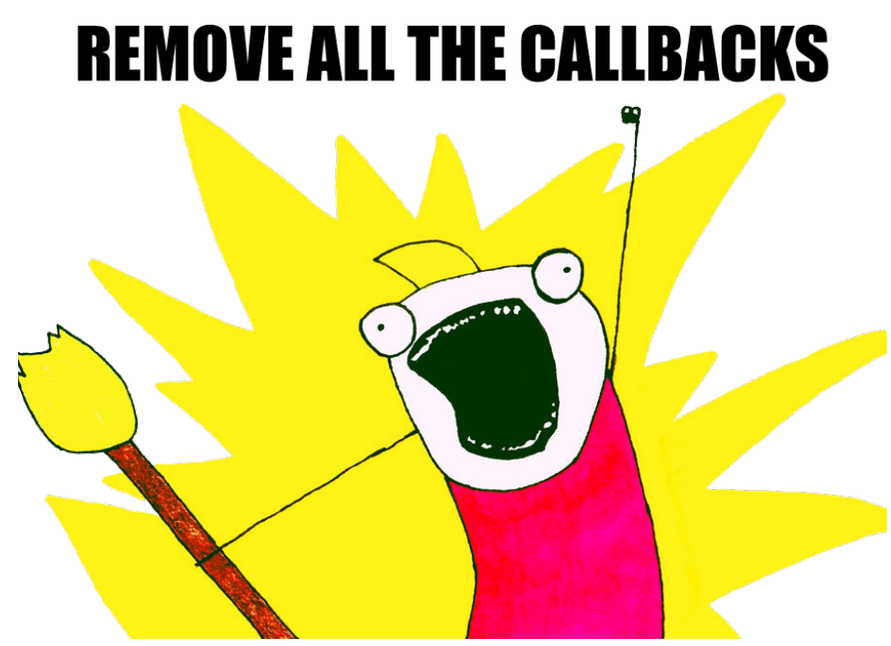
Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
milkCow( function(err, milk) {
    console.log("I have " + milk + " and can prepare cheese.");
    prepareCheese(milk, function(err, cheese) {
        console.log("I have now " + cheese + " and can sell it.");
        sellCheese(cheese, function(err, money) {
            console.log("Youpi! I have my money.");
        });
    });
});
```



At every level, you will have more than one line of code. It will quickly become difficult to know where you are... Understanding and maintaining the code will be a nightmare.



Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- First approach: use the **async.js** module

waterfall(tasks, [callback])

Runs the `tasks` array of functions in series, each passing their results to the next in the array. However, if any of the `tasks` pass an error to their own callback, the next function is not executed, and the main `callback` is immediately called with the error.

Arguments

- `tasks` - An array of functions to run, each function is passed a `callback(err, result1, result2, ...)` it must call on completion. The first argument is an error (which can be `null`) and any further arguments will be passed as arguments in order to the next task.
- `callback(err, [results])` - An optional callback to run once all the functions have completed. This will be passed the results of the last task's callback.

Example

```
async.waterfall([
    function(callback) {
        callback(null, 'one', 'two');
    },
    function(arg1, arg2, callback) {
      // arg1 now equals 'one' and arg2 now equals 'two'
        callback(null, 'three');
    },
    function(arg1, callback) {
        // arg1 now equals 'three'
        callback(null, 'done');
    }
], function (err, result) {
    // result now equals 'done'
});
```

The functions that we pass to async.waterfall must **respect a certain contract**:

the **last parameter must be callback function** (it will be provided by async.js and handle the magic)

the function must **invoke this callback when it has completed**

it must pass an error (if any) and **the results it wishes to pass** to the next function.

the function must **declare parameters for the inputs** it wishes to receive from the previous function.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **We are lucky!!!!**

```
function prepareCheese( milk, callbackWhenCheeseIsAvailable ) {
    console.log("Start preparing cheese with " + milk);
    setTimeout( function() {
        console.log("Done preparing cheese.");
        callbackWhenCheeseIsAvailable(null, "CHEESE");
    }, 3000);
};
```

The functions that we pass to async.waterfall must **respect a certain contract**:

the **last parameter must be callback function** (it will be provided by async.js and handle the magic)

the function must **invoke this callback when it has completed**

it must pass an error (if any) and **the results it wishes to pass** to the next function.

the function must **declare parameters for the inputs** it wishes to receive from the previous function.

Beyond simple callbacks...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **We can rewrite this...**

```
milkCow( function(err, milk) {
    console.log("I have " + milk + " and can prepare cheese.");
    prepareCheese(milk, function(err, cheese) {
        console.log("I have now " + cheese + " and can sell it.");
        sellCheese(cheese, function(err, money) {
            console.log("Youpi! I have my money.");
        });
    });
});
```

- **Into this...**

```
async.waterfall([milkCow, prepareCheese, sellCheese],
 function(err, results) {
    console.log("I have done all the work and now I have " + results);
});
```

Promises

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Async.js is one of the libraries that can help us with asynchronous code.
- There is a more general mechanism: **promises**.

then **Promises/A+**

An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

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.

https://github.com/promises-aplus/promises-spec

https://github.com/petkaantonov/bluebird

http://documentup.com/kriskowal/q/

then

BlueBird

jQuery
write less, do more.

Deferred objects

Going back to our cheese problem...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **We would like to rewrite this...**

```
milkCow( function(err, milk) {
    console.log("I have " + milk + " and can prepare cheese.");
    prepareCheese(milk, function(err, cheese) {
        console.log("I have now " + cheese + " and can sell it.");
        sellCheese(cheese, function(err, money) {
            console.log("Youpi! I have my money.");
        });
    });
});
```

- **Into something like...**

```
milkCow().then(prepareCheese).then(sellCheese).then( function() {...});
```

- **But our existing methods use callbacks, not promises...**

Going back to our cheese problem...

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **Bluebird is one of the most popular Promise libraries.**
- It makes it possible to "promisify" existing functions:

```
var milkCowPromisified = Promise.promisify(milkCow);
var prepareCheesePromisified = Promise.promisify(prepareCheese);
var sellCheesePromisified = Promise.promisify(sellCheese);
```

```
milkCowPromisified()
  .then(prepareCheesePromisified)
  .then(sellCheesePromisified)
  .then( function( result ) {
    console.log("Final result: " + result);
   }
  );
```