Lecture 4: Persistence

Olivier Liechti
TWEB

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

Today's agenda

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

14h00 - 15h00	60'	**JavaScript 101 reminders** Prototypes and .prototype **Lecture/exercise** Consuming a REST API, coordination asynchronous operations
15h00 - 15h10	*10'*	*Break*
15h10 - 16h25	75'	**JavaScript 101: asynchronous programming** async.js, promises **Lecture: MongoDB** Data modeling, CRUD operations, drivers

On what objects can I find a "prototype" property and why is it useful?

#1 most objects do NOT have prototype property

```
var student = { "name" : "john doe"};
console.log(student.protoype); // undefined
```

#2 to find the prototype of an object, use Object.getPrototypeOf

```
var father = {};
var son = Object.create(father);
var proto = Object.getPrototypeOf(son);
console.log ( proto === father); // true
```

#3 all functions have a prototype property

```
function f() {};
console.log(f.prototype); // "{}"
```

#4 if an object **o** is created with "new **f**()", then the prototype of **o** is **f.prototype**.

```
function Student() {};
var s = new Student();
console.log( Object.getPrototypeOf(s) ===
Student.prototype); // true
```

#5 we use this feature to share code between object instances

```
function Student() {};
Student.prototype.study = function() {};

var s1 = new Student();
var s2 = new Student();
s1.study(); // s1 inherits study from its proto
s2.study(); // same thing for s2

console.log( s1.study === s2.study ); // true
```

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

Let's celebrate October!

Phase 1

Phase 1: Beers are RESOURCES

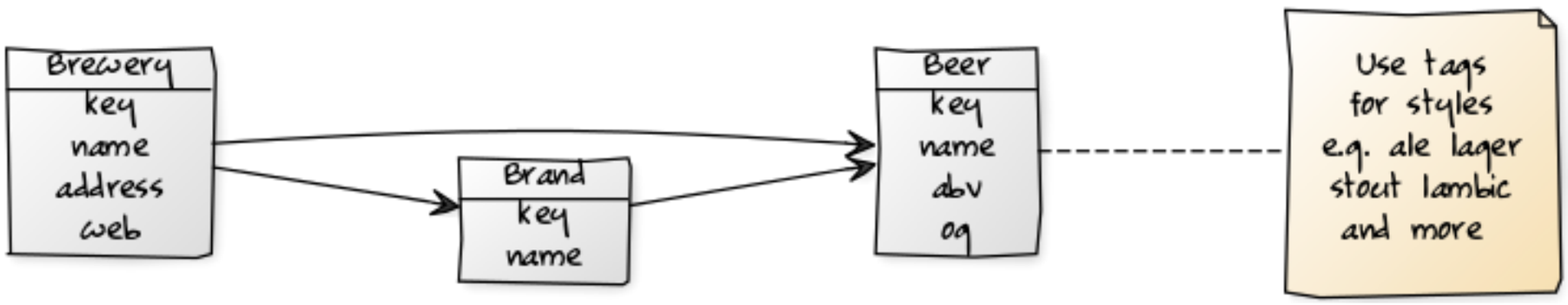
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **beer.db** is a free and open beer data project
- http://openbeer.github.io/
- Web UI: http://prost.herokuapp.com
- REST API: http://prost.herokuapp.com/api/v1

You can GET a random beer


```
{
    key: "dedollearabier",
    title: "De Dolle Arabier",
    synonyms: null,
    abv: "8.0",
    srm: null,
    og: null,
  - tags: [
        "blond",
        "belgian strong pale ale"
    ],
  - brewery: {
        key: "dedollebrouwers",
        title: "De Dolle Brouwers"
    },
  - country: {
        key: "be",
        title: "Belgium"
    }
}
```


`http://prost.herokuapp.com/api/v1/beer/rand`



Phase 1: Beers are RESOURCES

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

- **Can we write a Node.js script (command line utility) which fetches 100 random beers and stores them in local files?**

Phase 1: Beers are RESOURCES

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

- **Can we write a Node.js script (command line utility) which fetches 100 random beers and stores them in local files?**
  - How do we setup the project, knowing that we will need to use npm modules?
  - How do we invoke the REST API endpoint and fetch a JSON beer?
  - How do we store one beer in a file?
  - How do we execute these operations in sequence?
  - How do we fetch 10 beers and know that we are done?

Phase 1: Beers are RESOURCES

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

- **How do we setup the project, knowing that we will need to use npm modules?**

```
mkdir okto
cd okto
```

```
npm init
```

This will create a package.json file for your project

```
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (okto)
version: (1.0.0) 0.1.0
description: A beer utility
entry point: (index.js) okto.js
test command:
git repository:
keywords:
author: Olivier Liechti
license: (ISC)
About to write to /Users/admin/Documents/heig-vd/Teaching/TWEB/demos2015/okto/package.json:

{
  "name": "okto",
  "version": "0.1.0",
  "description": "A beer utility",
  "main": "okto.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Olivier Liechti",
  "license": "ISC"
}
```

Phase 1: Beers are RESOURCES

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

- **How do we invoke the REST API endpoint and fetch a JSON beer?**
  - If you browse through the Node.js documentation, you will find a module named **http**, with a **httpRequest()** function.
  - Third-party developers have developed more powerful http client libraries and have shared them on **npm**.
  - **request** (https://www.npmjs.com/package/request) has been downloaded 12,370,637 times in the last month.

```
npm install --save request
```

```
$ ls -l
total 8
drwxr-xr-x  3 admin  staff  102  7 oct 14:52 node_modules
-rw-r--r--  1 admin  staff  278  7 oct 14:52 package.json
```

Phase 1: Beers are RESOURCES

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

- **How do we invoke the REST API endpoint and fetch a JSON beer?**

```
touch okto.js // and then use your favorite editor...
```

```
var request = require('request');

var BEER_API_ENDPOINT = "http://prost.herokuapp.com/api/v1/beer/rand";

request(BEER_API_ENDPOINT, function (error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log("We have received a response from the BEER API.");
    console.log(body);
  } else {
    console.log("Could not get a response from the BEER API: " + error);
  }
});
```

This is a string... we would prefer a JavaScript object

```
$ node okto.js
We have received a response from the BEER API.
{"key":"unertlweissbier","title":"Unertl Wei\u00dfbier","synonyms":null,"abv":null,"srm":null,"og":null,"tags":
[],"brewery":{},"country":{"key":"de","title":"Germany"}}
```

Phase 1: Beers are RESOURCES

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

- We can define default parameters for all of our HTTP requests. We can specify that in general, we expect JSON and that the response should be automatically parsed.

```
var request = require('request');
var BEER_API_ENDPOINT = "http://prost.herokuapp.com/api/v1/beer/rand";

restClient = request.defaults( { json: true, timeout: 2000 } );

restClient.get(BEER_API_ENDPOINT, function (error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log("We have received a response from the BEER API.");
    console.log(body); console.log(body.title);
  } else {
    console.log("Could not get a response from the BEER API: " + error);
  }
});
```

```
We have received a response from the BEER API.
{ key: 'ozenoyukidokeipa',
  title: 'Ozeno Yukidoke IPA',
  synonyms: null,
  abv: '5.0',
  srm: null,
  og: null,
  tags: [],
  brewery: {},
  country: { key: 'jp', title: 'Japan' } }
Ozeno Yukidoke IPA
```

This is now a JavaScript object and it is possible to access its properties

Phase 1: Beers are RESOURCES

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

- **How do we store one beer in a file?**
- We can use the standard fs Node.js module.

```
var fs = require('fs');

function saveBeerInFile( beer ) {
    var filename = beer.key + ".json";
    fs.writeFile(filename, JSON.stringify(beer), function (err) {
      if (err) throw err;
      console.log('Beer has been saved in ' + filename);
    });
}

var beer = {
    "key" : "testBeer",
    "title" : "just a test"
}
saveBeerInFile( beer );
```

Let's validate this step independently from the HTTP request!

```
$ node okto.js
Beer has been saved in testBeer.json
$ more testBeer.json
{"key":"testBeer","title":"just a test"}
```

Phase 1: Beers are RESOURCES

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

- **How do we fetch 10 beers and know that we are done?**

```
console.log("Before the loop");
for (var i=0; i<10; i++) {
      fetchBeer(function(err, beer) {
            if (err) {
                  console.log("Could not fetch a beer, nothing we can do...");
            } else {
                  saveBeerInFile(beer);
            }
      });
}
console.log("After the loop");
```

```
$ mkdir beers
$ node okto.js
Before the loop
After the loop
Beer has been saved in beers/redoakframboisefroment.json
Beer has been saved in beers/hackerpschorrmuenchenergold.json
Beer has been saved in beers/beerlao.json
Beer has been saved in beers/kudoshefeweizen.json
Beer has been saved in beers/baunti1609lager.json
Beer has been saved in beers/castatriguera.json
Beer has been saved in beers/ambershock.json
Beer has been saved in beers/emersonsold95.json
Beer has been saved in beers/nogneoimperialstout.js
Beer has been saved in beers/vanderghinsteoudbruin.
```

This is an asynchronous process, so we have failed!

```
function saveBeerInFile(beer) {
      var filename = "beers/" + beer.key + ".json";
      fs.writeFile(filename, JSON.stringify(beer), function(err) {
            if (err) throw err;
            console.log('Beer has been saved in ' + filename);
      });
};
```

Phase 1: Beers are RESOURCES

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

- **Let's use a progress monitor to keep track of our progress.**

```
function createProgressMonitor(numberOfOperationsToPerform, callback) {
    var numberOfSuccessfulOperations = 0;
    var numberOfFailedOperations = 0;

    function checkIfDone() {
        if (numberOfSuccessfulOperations + numberOfFailedOperations >= numberOfOperationsToPerform) {
            console.log("Progress monitor has detected that all operations have been completed.");
            callback(null, {
                successes: numberOfSuccessfulOperations,
                failures: numberOfFailedOperations
            });
        }
    };

    return {
        reportSuccess: function() {
            numberOfSuccessfulOperations++;
            checkIfDone();
        },
        reportFailure: function() {
            numberOfFailedOperations++;
            checkIfDone();
        }
    };
}
```

Here, we create two functions. So 2 **closures** are formed. The two function objects will keep access to the following variables:

numberOfOperationsToPerform
numberOfSuccessfulOperations
numberOfFailedOperations

The variables are **private** and cannot be accessed directly from the code calling createProgressMonitor.

We return an object, which gives us access to two functions.

Phase 1: Beers are RESOURCES

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

- **Let's use a progress monitor to keep track of our progress.**

```
function createProgressMonitor(numberOfOperationsToPerform, callback) {
    var numberOfSuccessfulOperations = 0;
    var numberOfFailedOperations = 0;

    function checkIfDone() {
        if (numberOfSuccessfulOperations + numberOfFailedOperations >= numberOfOperationsToPerform) {
            console.log("Progress monitor has detected that all operations have been completed.");
            callback(null, {
                successes: numberOfSuccessfulOperations,
                failures: numberOfFailedOperations
            });
        }
    };

    return {
        reportSuccess: function() {
            numberOfSuccessfulOperations++;
            checkIfDone();
        },
        reportFailure: function() {
            numberOfFailedOperations++;
            checkIfDone();
        }
    };
}
```

When we detect that we are done with all operations, we will invoke this callback provided by the client

When we invoke the callback, we provide the number of successful and failed operations.

Phase 1: Beers are RESOURCES

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

- **And now, let's use the progress monitor!**

This will be called when we are completely done.

```
var numberOfBeersToFetch = 100;
var progressMonitor = createProgressMonitor(numberOfBeersToFetch, function( err, results) {
     console.log("Now, we know that all operations have completed.");
     console.log(results.successes + " beers have bee successfully fetched and saved.");
     console.log(results.failures + " beers have not been fetched or saved.");
});

for (var i = 0; i < numberOfBeersToFetch; i++) {
     fetchBeer(function(err, beer) {
          if (err) {
               console.log("Could not fetch a beer, nothing we can do...");
               progressMonitor.reportFailure();
          } else {
               saveBeerInFile(beer, function(err) {
                    if (err) {
                         progressMonitor.reportFailure();
                    } else {
                         progressMonitor.reportSuccess();
                    }
               });
          }
     });
}
```

We launch the 100 asynchronous operations

We need to inform the progress monitor about the outcome of the operations

Phase 1: Beers are RESOURCES

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

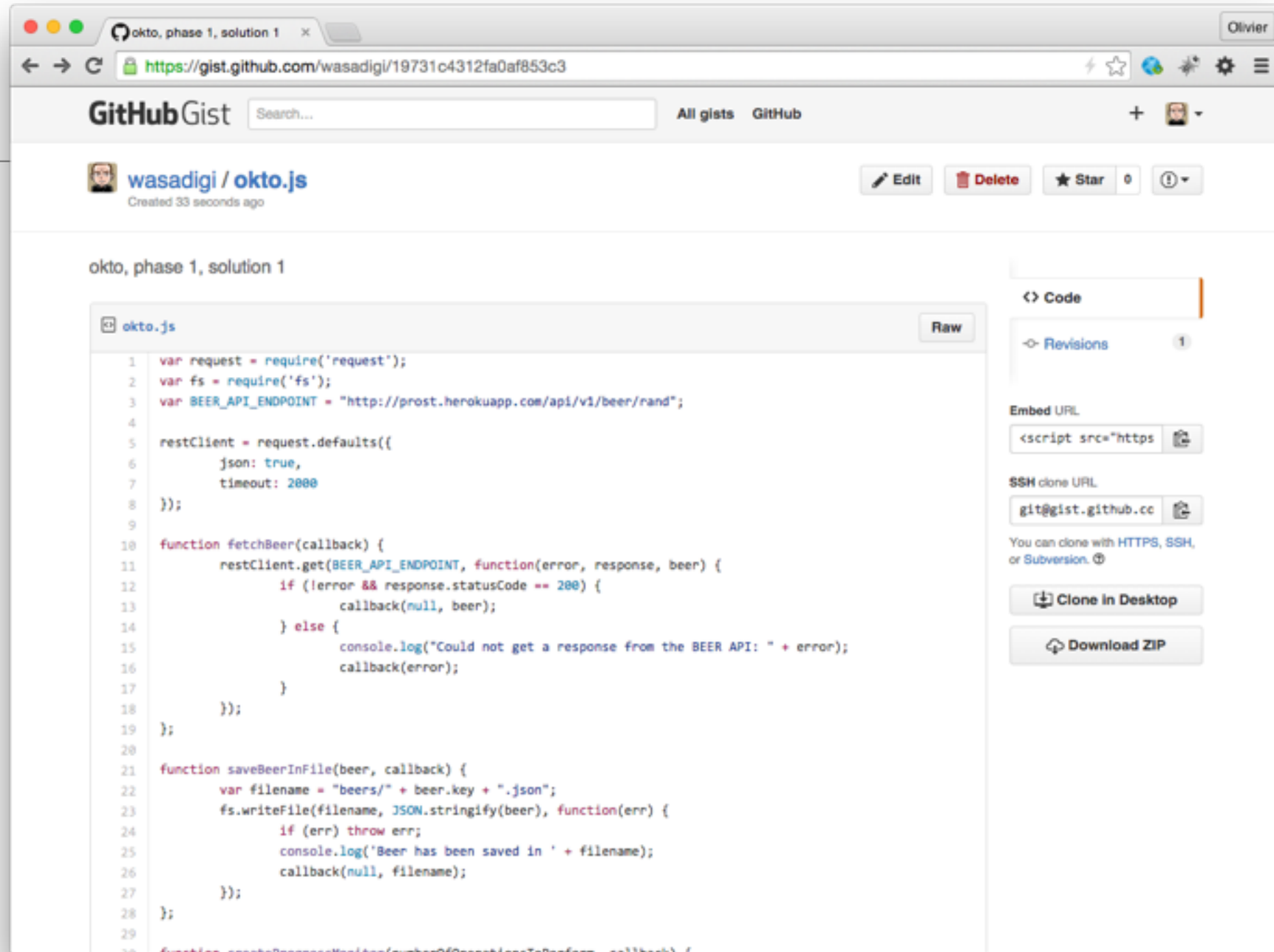
```
$ node okto.js
Beer has been saved in beers/santafe.json
Beer has been saved in beers/tipopils.json
Beer has been saved in beers/seasonalhell.json

...

Beer has been saved in beers/fohrenburgerbock.json
Beer has been saved in beers/estrelladamm.json
Beer has been saved in beers/oharasirishstout.json
Could not get a response from the BEER API: Error: ETIMEDOUT
Could not fetch a beer, nothing we can do...
Progress monitor has detected that all operations have been completed.
Now, we know that all operations have completed.
99 beers have bee successfully fetched and saved.
1 beers have not been fetched or saved.
```

The API call has timed out!

Phase



https://gist.github.com/wasadigi/19731c4312fa0af853c3

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

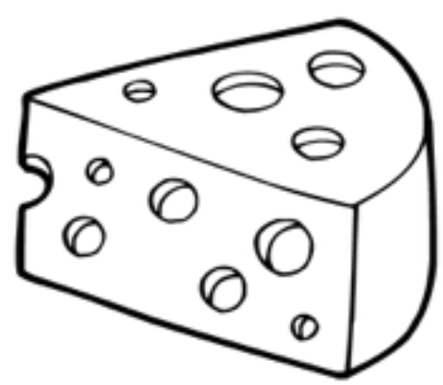
How can I execute multiple asynchronous operations in sequence?

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);
};
```

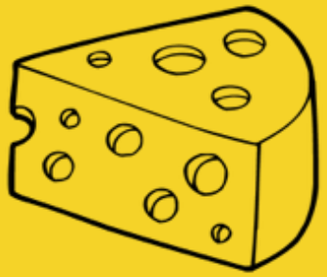
The first parameter is the **error** (if one happened) and the second one is the **result**.

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);
};
```



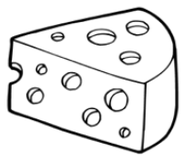
```
function sellCheese( cheese, callbackWhenCheeseHasBeenSold ) {
    console.log("Start selling " + cheese);
    setTimeout( function() {
        console.log("Done selling " + cheese);
        callbackWhenCheeseHasBeenSold(null, "MONEY");
    }, 1000);
};
```

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.");
        });
    });
});
```

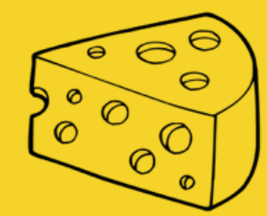
```
$ node promise.js
Start to milk cow...
Done milking cow.
I have now some MILK and can prepare cheese.
Start preparing cheese with MILK
Done preparing cheese.
I have now CHEESE and can sell it.
Start selling CHEESE
Done selling CHEESE
Youpi! I have my money.
```



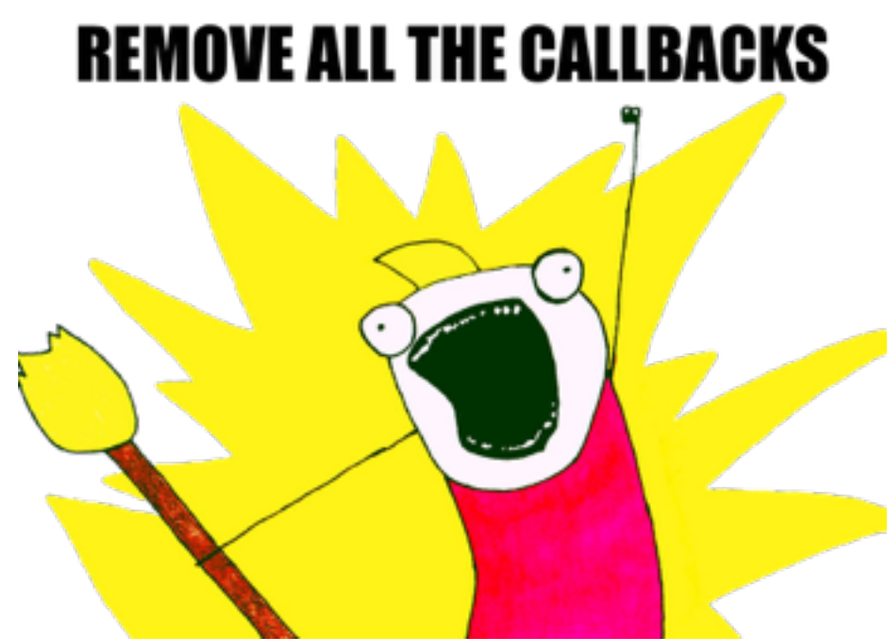
Beyond simple callbacks...

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...

- **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.");
        });
    });
});
```

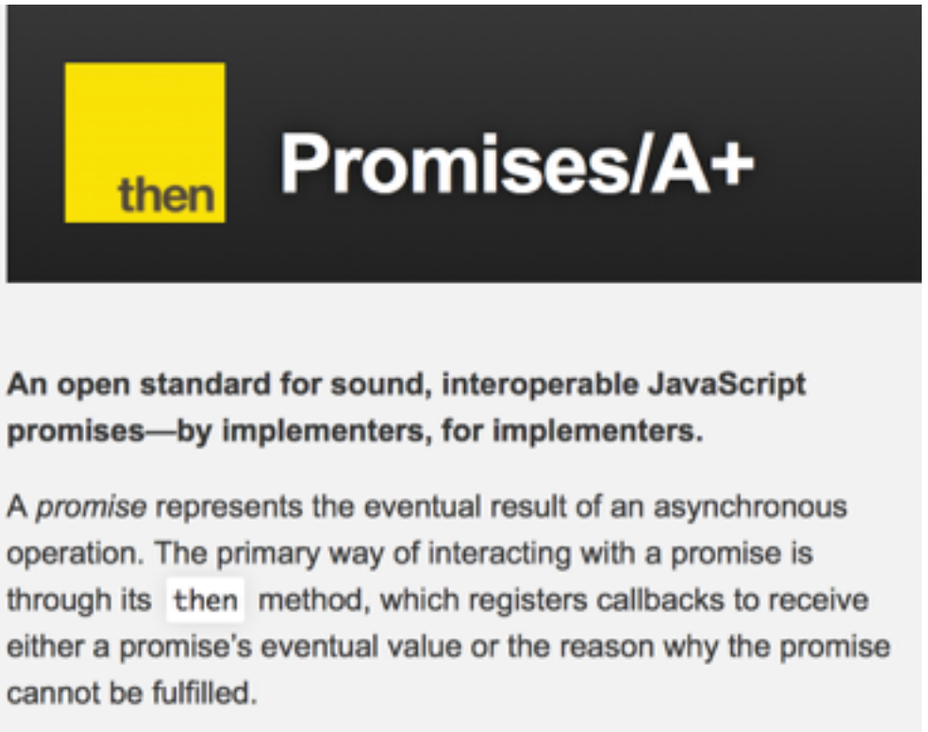
- **Into this...**

```
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**.



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

https://github.com/petkaantonov/bluebird

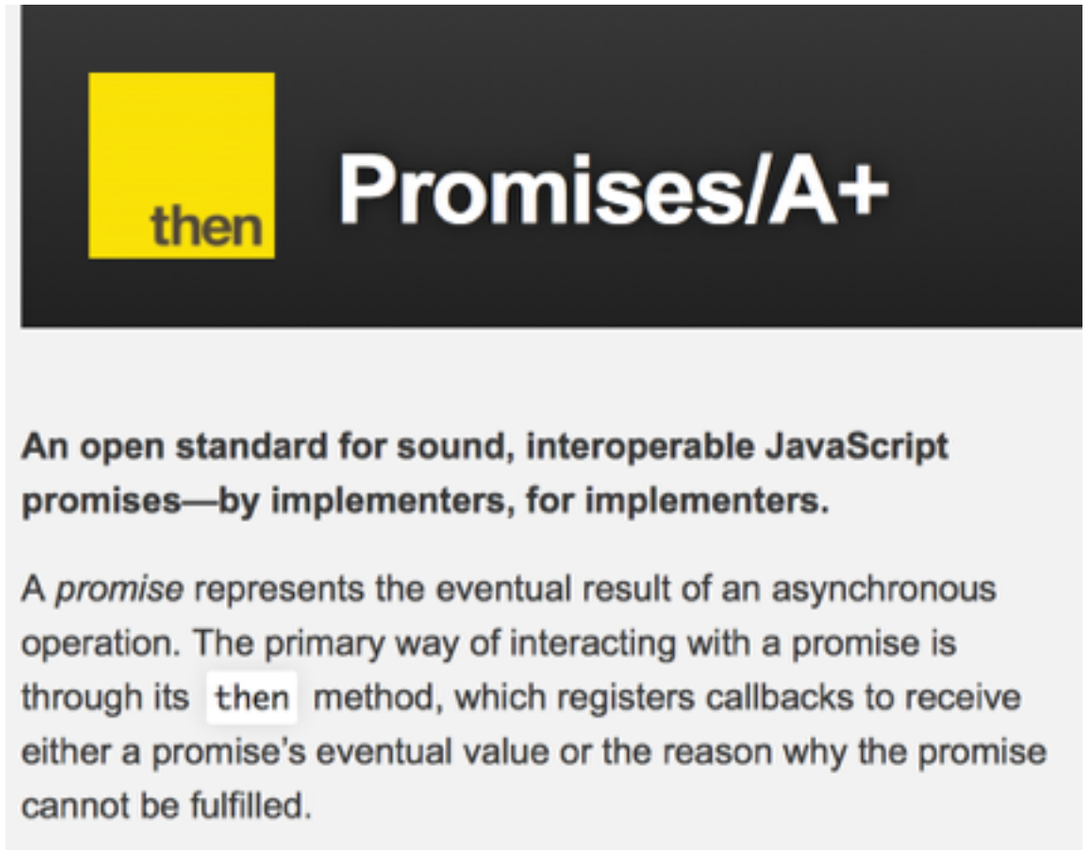
http://documentup.com/kriskowal/q/

Deferred 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."

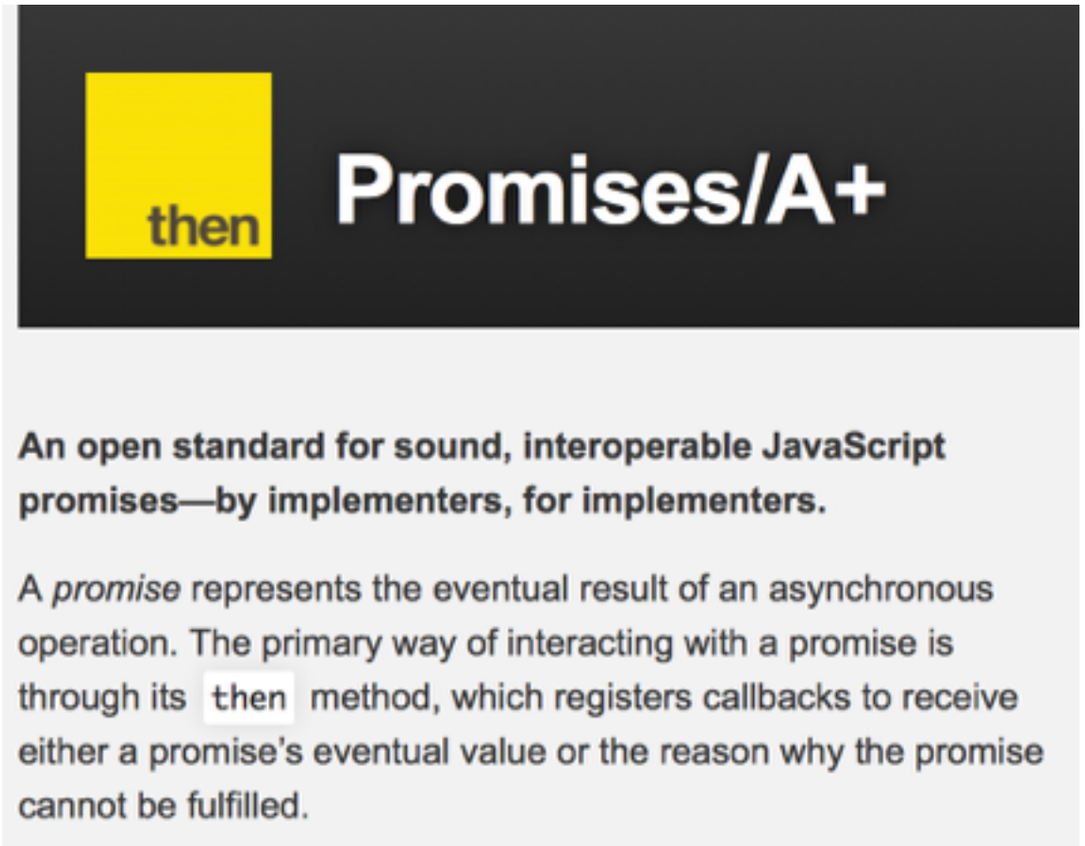


https://github.com/promises-aplus/promises-spec

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`** method 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”

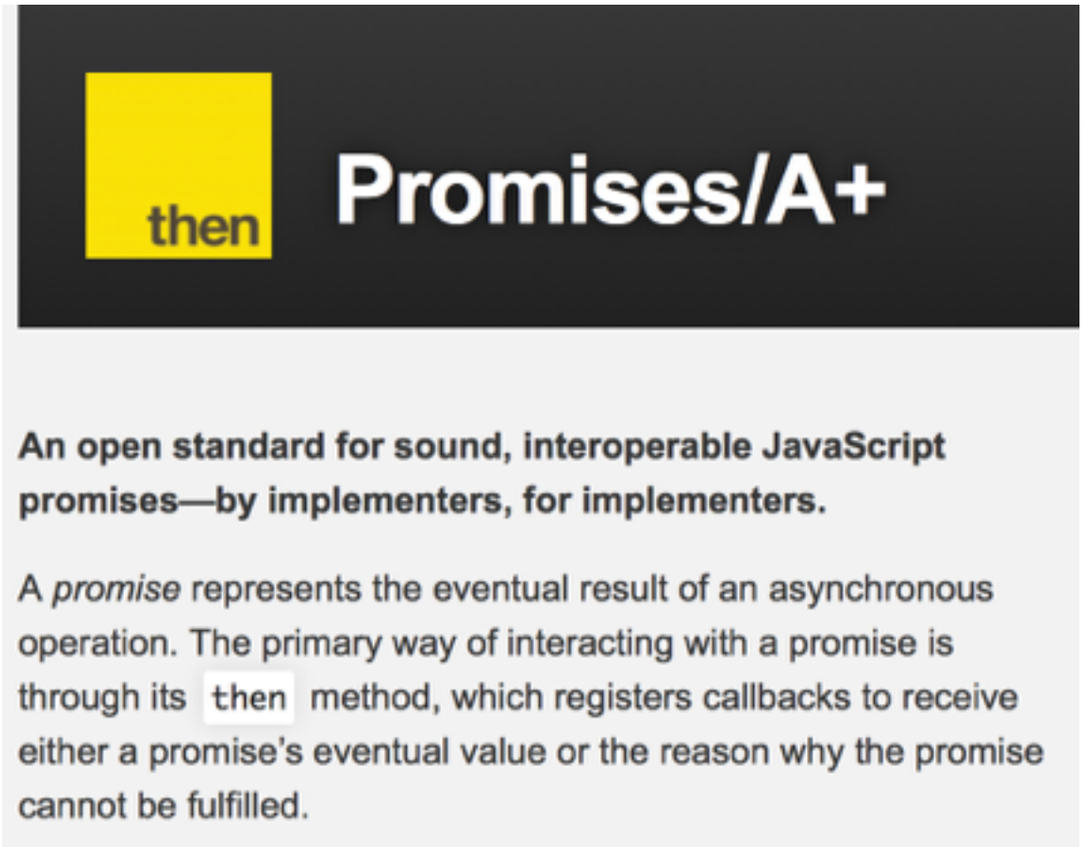


https://github.com/promises-aplus/promises-spec

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

Example 1: Deferred objects in JQuery

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

```
var d1 = new $.Deferred();
var d2 = new $.Deferred();
$.when( d1, d2 ).done(function ( v1, v2 ) {
  console.log( v1 ); // "Fish"
  console.log( v2 ); // "Pizza"
});
d1.resolve( "Fish" );
d2.resolve( "Pizza" );
```

*"a **promise** represents a value that is not yet known*
*a **deferred** represents work that is not yet finished"*

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

https://api.jquery.com/jquery.when/

```
$.when( $.ajax( "/page1.php" ), $.ajax( "/page2.php" ) )
  .then( myFunc, myFailure );
```

```
$.when( $.ajax( "/page1.php" ), $.ajax( "/page2.php" ) )
  .done( function( a1, a2 ) {
  // a1 and a2 are arguments resolved for the page1 and page2 ajax requests, respectively.
  // Each argument is an array with the following structure: [ data, statusText, jqXHR ]
    var data = a1[ 0 ] + a2[ 0 ]; // a1[ 0 ] = "Whip", a2[ 0 ] = " It"
    if ( /Whip It/.test( data ) ) {
    alert( "We got what we came for!" );
  }
});
```

Example 2: Promises with Q

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

```
function waitForGreetings1() {
      var deferred = Q.defer();
      setTimeout(function() {
            deferred.resolve("hello ");
      }, 2000);
      return deferred.promise;
};
```

These are 3 **asynchronous** operations

We **immediately** return the promise of a **future** result

```
function waitForGreetings2( previousResult ) {
      var deferred = Q.defer();
      setTimeout(function() {
            deferred.resolve(previousResult + "how are ");
      }, 200);
      return deferred.promise;
};
```

Some of them expect an **input**

```
function waitForGreetings3( previousResult) {
      var deferred = Q.defer();
      setTimeout(function() {
            deferred.resolve(previousResult + "you?");
      }, 10);
      return deferred.promise;
};
```

And produce an **output**

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

Example 2: Promises with Q

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

```
function waitForGreetings1() {
      var deferred = Q.defer();
      setTimeout(function() {
            deferred.resolve("hello ");
      }, 2000);
      return deferred.promise;
};
```

```
function waitForGreetings2( previousResult ) {
      var deferred = Q.defer();
      setTimeout(function() {
            deferred.resolve(previousResult + "how are "
      }, 200);
      return deferred.promise;
};
```

```
function waitForGreetings3( previousResult) {
      var deferred = Q.defer();
      setTimeout(function() {
            deferred.resolve(previousResult + "you?");
      }, 10);
      return deferred.promise;
};
```

We pass a function object (we don't invoke the function ourselves).

The function that we pass returns a promise.

```
waitForGreetings1()
  .then ( waitForGreetings2 )
  .then ( waitForGreetings3 )
  .then ( function(result) {
    console.log("final result : " + result);
});
```

When the promise returned by waitForGreetings1 is resolved, waitForGreetings2 is invoked (the result returned by waitForGreetings1 is passed in parameter)

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

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);
   }
  );
```

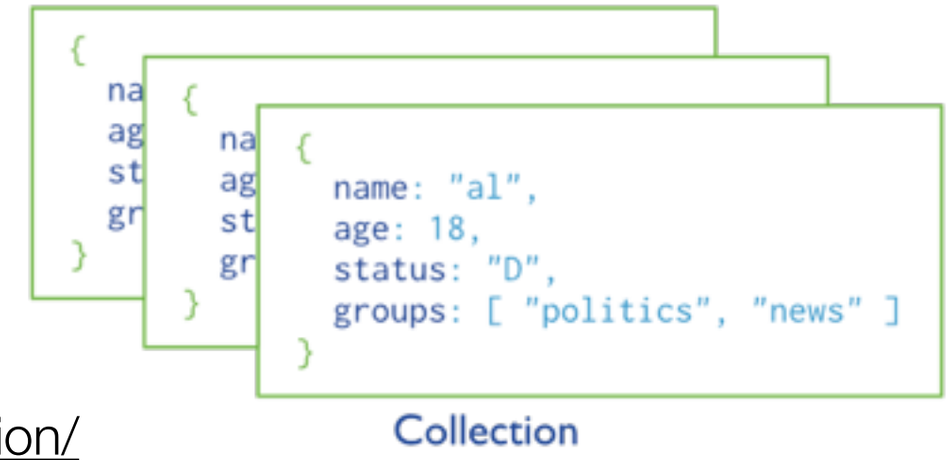
mongoDB

Document-oriented NoSQL Database

MongoDB

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

- MongoDB is one of the most popular **NoSQL** databases (and one of the first to have been categorized as such).
- It is a **schema-less document-oriented** database:
  - The data store is made of several **collections**.
  - Every collection contains a set of **documents**, which you can think of as JSON objects.
  - The **structure of documents is not defined** *a priori* and is not enforced. This means that a collection can contain documents that have different fields.

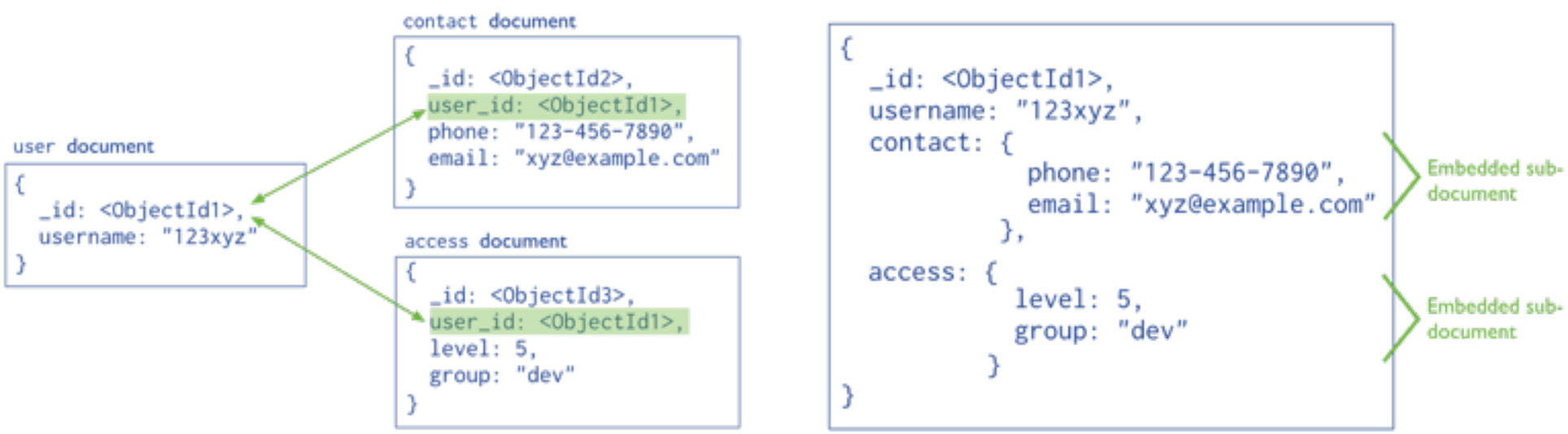


http://docs.mongodb.org/manual/core/crud-introduction/

Data modeling

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

- Creating a data model with MongoDB does not have to follow the rules that apply for relational databases. Often, they should not.
- Consider theses questions: is this a **composition** relationship (**containment**)? Is this "**aggregate**" of documents often used at the same time (i.e. can we reduce chattiness)? Would embedding lead to "a lot" of data **duplication**?



Normalized data model
(references)

Embedded data model
(sub-documents)

https://docs.mongodb.org/manual/core/data-modeling-introduction/

One-to-one relationships

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

2 documents (requires 2 queries to get all of the person data)

```
{
    _id: "joe",
    name: "Joe Bookreader"
}

{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
}
```

1 single, aggregate document (in this case, it is a better choice)

```
{
    _id: "joe",
    name: "Joe Bookreader",
    address: {
                street: "123 Fake Street",
                city: "Faketon",
                state: "MA",
                zip: "12345"
            }
}
```

Normalized data model
(references)

Embedded data model
(sub-documents)

https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/

One-to-many relationships

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

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: "12345"
}
```

MongoDB document can have an arbitrary structure, including arrays

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: "12345"
                }
              ]
}
```

https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/

One-to-many relationships

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

ok if if have few books per publisher

```
{
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher: {
                name: "O'Reilly Media",
                founded: 1980,
                location: "CA"
            }
}

{
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English",
    publisher: {
                name: "O'Reilly Media",
                founded: 1980,
                location: "CA"
            }
}
```

duplication

```
{
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA",
    books: [12346789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
    _id: 234567890,
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English"
}
```

```
{
    _id: "oreilly",
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA"
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly"
}

{
    _id: 234567890,
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English",
    publisher_id: "oreilly"
}
```

better if you have many books per publisher

https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/

Insert data in MongoDB

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

- To insert data in MongoDB, you simply have to provide a **JSON document** (with an **arbitrary structure**).
- The documents in the collection do not have to all have the same structure (this is why we talk about a **schemaless** database).

Collection Document

```
db.users.insert(
    {
        name: "sue",
        age: 26,
      status: "A",
      groups: [ "news", "sports" ]
    }
  )
```

A document is always inserted in a specific collection.

MongoDB will assign a unique ID in the **_id** field for the new document.

Document

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

insert

Collection

```
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
```

users

https://docs.mongodb.org/manual/core/crud-introduction/

Insert data in MongoDB

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

- This is one example. You can also insert multiple documents at the same time, either by passing an array of documents or by performing a bulk operation. If you are dealing with many documents, this is important for performance reasons.

```
db.inventory.insert(
   {
     item: "ABC1",
     details: {
        model: "14Q3",
        manufacturer: "XYZ Company"
     },
     stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
     category: "clothing"
   }
)
```

```
db.inventory.find()
```

```
{ "_id" : ObjectId("53d98f133bb604791249ca99"), "item" : "AB
```

https://docs.mongodb.org/manual/tutorial/insert-documents/

Update and delete data in MongoDB

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

- When you update or delete documents, you specify which documents are concerned by the operation.
- You do that by specifying update or remove criteria. Specifying "{}" means that you want to apply the operation on all documents of the collection.

```
db.users.update(                    ← collection
   { age: { $gt: 18 } },            ← update criteria
   { $set: { status: "A" } },       ← update action
   { multi: true }                  ← update option
)
```

```
db.users.remove(                    ← collection
   { status: "D" }                  ← remove criteria
)
```

https://docs.mongodb.org/manual/core/crud-introduction/

Update data in MongoDB

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

- By default, update will modify only the first document that matches the selection criteria. You can specify the "multi" options if you want to update all documents that match the criteria.

selection criteria

```
db.inventory.update(
    { item: "MNO2" },
    {
      $set: {
        category: "apparel",
        details: { model: "14Q3", manufacturer: "XYZ Company" }
      },
      $currentDate: { lastModified: true }
    }
)
```

```
db.inventory.update(
    { category: "clothing" },
    {
      $set: { category: "apparel" },
      $currentDate: { lastModified: true }
    },
    { multi: true }
)
```

update operators

lastModified will be set to the current date

https://docs.mongodb.org/manual/tutorial/modify-documents/

Update data in MongoDB

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

Update Operators

Fields

Name	Description
`$inc`	Increments the value of the field by the specified amount.
`$mul`	Multiplies the value of the field by the specified amount.
`$rename`	Renames a field.
`$setOnInsert`	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
`$set`	Sets the value of a field in a document.
`$unset`	Removes the specified field from a document.
`$min`	Only updates the field if the specified value is less than the existing field value.
`$max`	Only updates the field if the specified value is greater than the existing field value.
`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp.

Array

Operators

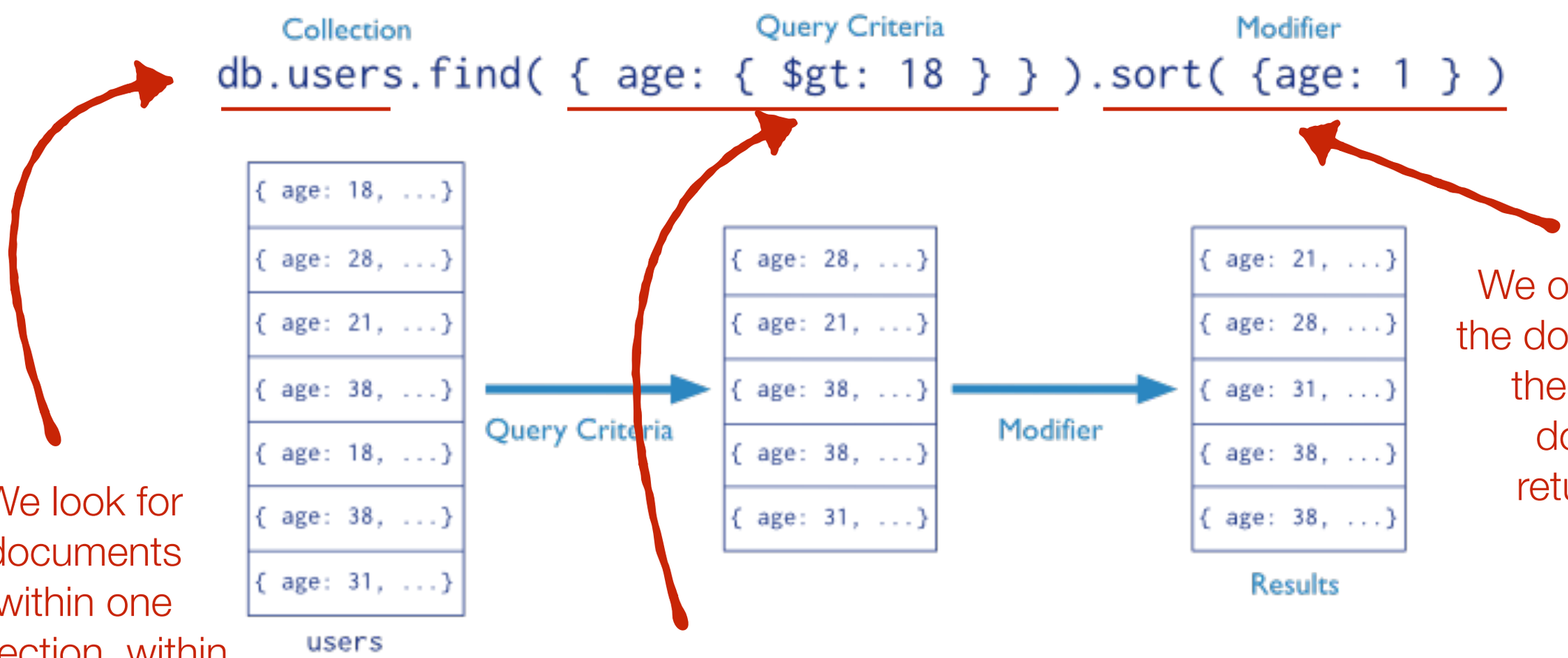
Name	Description
`$`	Acts as a placeholder to update the first element that matches the query condition in an update.
`$addToSet`	Adds elements to an array only if they do not already exist in the set.
`$pop`	Removes the first or last item of an array.
`$pullAll`	Removes all matching values from an array.
`$pull`	Removes all array elements that match a specified query.
`$pushAll`	*Deprecated.* Adds several items to an array.
`$push`	Adds an item to an array.

https://docs.mongodb.org/manual/reference/operator/update/

Query MongoDB

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

- MongoDB is one of the most popular **NoSQL** databases (and one of the first to have been categorized as such).



We look for documents within one collection, within one database

We optionally sort the documents, limit the number of documents returned, etc.

We define the criteria for which documents should be considered, and which of their fields should be considered (projection)

https://docs.mongodb.org/manual/core/crud-introduction/

Query MongoDB

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

```
db.inventory.find()
```

all documents

```
db.inventory.find( { type: "snacks" } )
```

documents, which have a field "type" with a value of "snacks"

```
db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
```

documents, which have a field "type" with have a value of "food" or "snacks"

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

documents with a type field equal to "food" AND a price field with a value less than 9.95

```
db.inventory.find(
   {
     $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
   }
)
```

documents where the quantity is more than 100 OR the price is less than 9.95

https://docs.mongodb.org/manual/tutorial/query-documents/

Query MongoDB: arrays

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

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8,
```

Exact match on the entire array

```
db.inventory.find( { ratings: 5 } )
```

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Return documents if the array contains a specific value

https://docs.mongodb.org/manual/tutorial/query-documents/

Query MongoDB: arrays

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

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: { $elemMatch: { $gt: 5, $lt: 9 } } } )
```

Documents where one element of ratings is at the same time > 5 AND < 9

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: { $gt: 5, $lt: 9 } } )
```

Documents where there is one element of ratings > 5 and one element < 9

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

https://docs.mongodb.org/manual/tutorial/query-documents/

SQL vs MongoDB

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

`SELECT * FROM users WHERE status != "A"`	`db.users.find( { status: { $ne: "A" } } )`
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`
`SELECT * FROM users WHERE status = "A" OR age = 50`	`db.users.find( { $or: [ { status: "A" } , { age: 50 } ] } )`
`SELECT * FROM users WHERE age > 25`	`db.users.find( { age: { $gt: 25 } } )`
`SELECT * FROM users WHERE age < 25`	`db.users.find( { age: { $lt: 25 } } )`
`SELECT * FROM users WHERE age > 25 AND age <= 50`	`db.users.find( { age: { $gt: 25, $lte: 50 } } )`
`SELECT * FROM users WHERE user_id like "%bc%"`	`db.users.find( { user_id: /bc/ } )`
`SELECT * FROM users WHERE status = "A" ORDER BY user_id ASC`	`db.users.find( { status: "A" } ).sort( { user_id: 1 } )`
`SELECT * FROM users WHERE status = "A" ORDER BY user_id DESC`	`db.users.find( { status: "A" } ).sort( { user_id: -1 } )`
`SELECT COUNT(*) FROM users`	`db.users.count()` *or* `db.users.find().count()`
`SELECT COUNT(user_id) FROM users`	`db.users.count( { user_id: { $exists: true } } )` *or* `db.users.find( { user_id: { $exists: true } } ).count()`
`SELECT COUNT(*) FROM users WHERE age > 30`	`db.users.count( { age: { $gt: 30 } } )` *or* `db.users.find( { age: { $gt: 30 } } ).count()`
`SELECT DISTINCT(status) FROM users`	`db.users.distinct( "status" )`

https://docs.mongodb.org/manual/reference/sql-comparison/

Query MongoDB: projections

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

- When you perform a query, you can specify which fields you are interested in (think about performance)

```
db.inventory.find( { type: 'food' } )
```

Return all fields (no second argument)

```
db.inventory.find( { type: 'food' }, { item: 1, qty: 1 } )
```

We are only interested by the item and qty fields (we will also get _id)

```
db.inventory.find( { type: 'food' }, { item: 1, qty: 1, _id:0 } )
```

We are only interested by the item and qty fields (we really don't want to get _id)

```
db.inventory.find( { type: 'food' }, { type:0 } )
```

We want all fields except type

https://docs.mongodb.org/manual/tutorial/project-fields-from-query-results/

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

mongojs

Star 1,099

A node.js module for mongodb, that emulates the official mongodb API. It wraps mongodb-native and is available through npm

```
npm install mongojs
```

build passing

Accessing MongoDB from Node.js

Accessing mongoDB from Node.js

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

- In the **Java ecosystem**, it is possible to interact with a RDBMS by using a JDBC driver:
  - The program loads the driver.
  - The program establishes a connection with the DB.
  - The program sends SQL queries to read and/or update the DB.
  - The program manipulates tabular result sets returned by the driver.
- With **Node.js and mongoDB**, the process is similar:
  - There is a **Node.js driver for mongoDB** (in fact, there are several).
  - A Node.js module can connect to a mongoDB server and issue queries to **manipulate collection and documents**.

Example 1: connect and insert

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

```
var MongoClient = require('mongodb').MongoClient;

MongoClient.connect("mongodb://localhost:27017/exampleDb", function(err, db) {
  if(err) { return console.dir(err); }

  var collection = db.collection(‘test');

  var doc1 = {'hello':'doc1'};
  var doc2 = {'hello':'doc2'};
  var lotsOfDocs = [{'hello':'doc3'}, {'hello':'doc4'}];

  collection.insert(doc1);
  collection.insert(doc2, {w:1}, function(err, result) {});
  collection.insert(lotsOfDocs, {w:1}, function(err, result) {});

});
```

http://mongodb.github.io/node-mongodb-native/api-articles/nodekoarticle1.html

Example 2: query

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

```
var MongoClient = require('mongodb').MongoClient;

MongoClient.connect("mongodb://localhost:27017/exampleDb", function(err, db) {
  if(err) { return console.dir(err); }

  var collection = db.collection('test');
  var docs = [{mykey:1}, {mykey:2}, {mykey:3}];
  collection.insert(docs, {w:1}, function(err, result) {

    // beware of memory consumption!
    collection.find().toArray(function(err, items) {});

    // better when many documents are returned
    var stream = collection.find({mykey:{$ne:2}}).stream();
    stream.on("data", function(item) {});
    stream.on("end", function() {});

    // special case when only one document is expected
    collection.findOne({mykey:1}, function(err, item) {});

  });
});
```

http://mongodb.github.io/node-mongodb-native/api-articles/nodekoarticle1.html

Accessing mongoDB from Node.js

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

- **mongojs** is a very useful npm module, which provides an alternative to the official Node.js driver. Its API is very similar to what you type on the mongo

```
// simple usage for a local db
var db = mongojs('mydb', ['mycollection']);

// the db is on a remote server (the port default to mongo)
var db = mongojs('example.com/mydb', ['mycollection']);

// we can also provide some credentials
var db = mongojs('username:password@example.com/mydb', ['mycollection']);

// connect now, and worry about collections later
var db = mongojs('mydb');
var mycollection = db.collection('mycollection');
```

mongojs

Star 1,099

A node.js module for mongodb, that emulates the official mongodb API. It wraps mongodb-native and is available through npm

```
npm install mongojs
```

build passing

```
// find everything
db.mycollection.find(function(err, docs) {
    // docs is an array of all the documents in mycollection
});

// find everything, but sort by name
db.mycollection.find().sort({name:1}, function(err, docs) {
    // docs is now a sorted array
});
```

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

Let's celebrate October!

Phase 2

Phase 2: Beers need a FRIDGE

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

- **Instead of storing beers in local files, can we store them in MongoDB?**

Phase 2: Beers need a FRIDGE

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

- **Instead of storing beers in local files, can we store them in MongoDB?**
  - How can talk to MongoDB from my Node.js script?
  - If I have a JavaScript beer object, what do I need to pass to MongoDB?
  - How do I handle duplicates (the API gives me a random beer, so it is possible to get the same beer more than once).¨
  - Once I have beers in MongoDB, how can I query the data?

Phase 2: Beers need a FRIDGE

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

- **How can talk to MongoDB from my Node.js script?**
  - There are many different ways. Some of the options are listed on the MongoDB site: http://docs.mongodb.org/ecosystem/drivers/node-js/
  - We will **not** use the officially supported driver here. Instead, we will first use the **mongojs** node module, which provides an API very similar to what you type on the mongo interactive shelll (so it's pretty cool if you are getting started with MongoDB).
  - Later on, we will use the **Mongoose** mapping framework, which provides a higher-level API (a bit like JPA offers a higher-level API than JDBC in the Java EE ecosystem).

```
npm install --save mongojs
```

Phase 2: Beers need a FRIDGE

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

- **How can talk to MongoDB from my Node.js script?**
  - Lets **create another file** (we will transform it into a Node.js module later on, but for now we only want to be able to test the DB stuff separtely).
  - Let's also **install the mongojs** module.

```
touch mokto.js
npm install --save mongojs
```

  - If we don't worry about the entire sequence of operations right now, we can try to connect to the local database and insert a document in a collection.

```
var mongojs = require('mongojs');
var db = mongojs('beersDb', ['beers']);
db.beers.insert( { "name" : "boxer", "country" : "Switzerland"}, function(err, document) {});
```

Phase 2: Beers need a FRIDGE

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

- **How can talk to MongoDB from my Node.js script?**
  - We want to close the DB connection, but we have to be careful not to do it before or while we are inserting. So at the moment, we need to do it in the callback function.

```
debugger; // that will make it easy to use "slc debug mokto.js ."

// We connect to the local server and use the beersDb database, and the beers collection
// If they don't exist, they will be created on the fly

var db = mongojs('beersDb', ['beers']);

db.beers.insert( { "name" : "boxer", "country" : "Switzerland"}, function(err, document) {
     if (err) {
          console.log("Error while inserting document " + err.errmsg);
     } else {
          console.log("Document created with _id : " + document._id)
     }
     console.log("We know that the insert has been done, so it is safe to close the DB connection.")
     db.close();
});
// if we were doing db.close() here, we would close the connection too early!
```

Phase 2: Beers need a FRIDGE

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

- **How can we insert documents from a MongoDB collection?**

```
function insertBeer( beer, callbackWhenBeerHasBeenInserted ) {
    var db = mongojs('beersDb', ['beers']);
    db.beers.insert( beer, function(err, doc) {
        db.close();
        if (err) {
            console.log("Unable to insert beer: " + beer);
            callbackWhenBeerHasBeenInserted(err);
        } else {
            console.log("Beer has been inserted: " + beer);
            callbackWhenBeerHasBeenInserted(null, doc);
        }
    });
}
```

Phase 2: Beers need a FRIDGE

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

- **How do we create a module, export functionality and import it?**
- In your module (mokto.js), use module.exports

```
function insertBeer( beer, callbackWhenBeerHasBeenInserted ) {
...
}

var beerDB = {
     insertBeer: insertBeer
}

module.exports = beerDB;
```

- In the client module (okto.js), import the module:

```
var database = require("./mokto.js");
function saveBeerInMongo(beer, callback) {
     database.insertBeer(beer, function(err, doc) {
          if (err) throw err;
          console.log("Beer has been saved in MongoDB with id " + doc._id);
          callback(null , doc);
     });
}
```

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

mongoose
elegant mongodb object modeling for node.js

mongoDB

Object Document Mapping with Mongoose

Mongoose: an ~~O~~RM for MongoDB

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

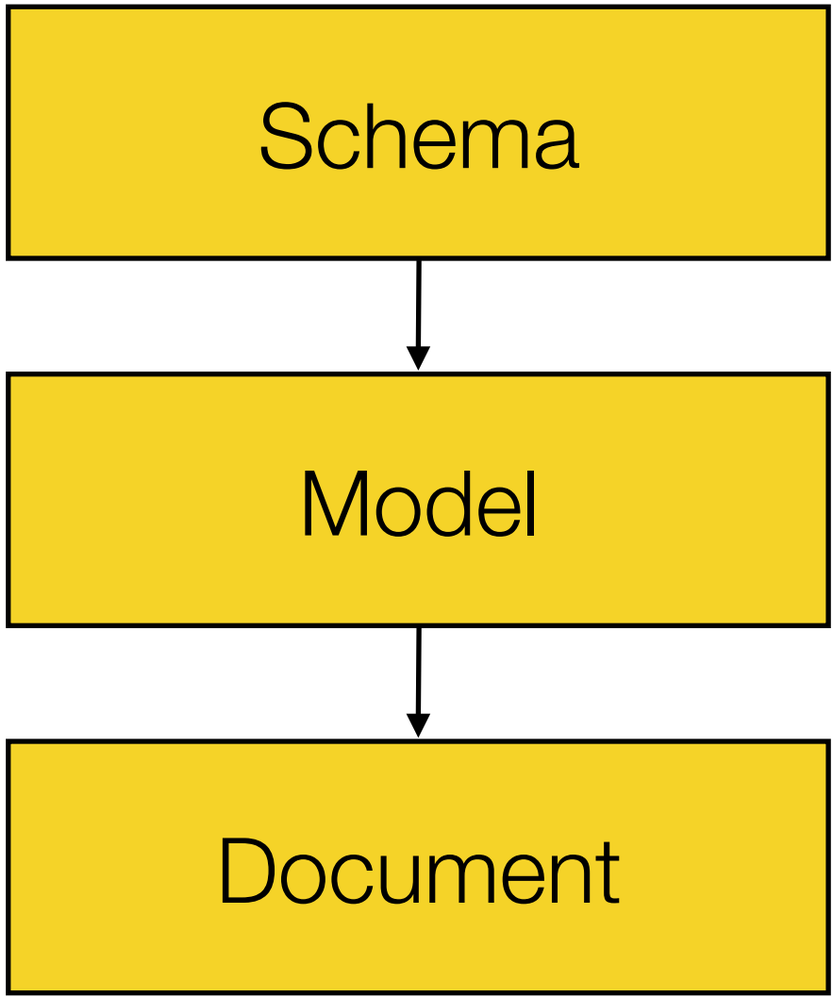
- In the **Java EE ecosystem**, we have seen how the **Java Persistence API** (JPA) specifies a standard way to interact with Object-Relational Mapping (ORM) frameworks.
  - The developer **first** creates an **object-oriented domain model**, by creating Entity classes and using various annotations (@Entity, @Id, @OneToMany, @Table, etc.)
  - He **then** uses an **Entity Manager** to **C**reate, **R**ead, **U**pdate and **D**elete objects in the DB.
  - The ORM framework takes care of the details: it **generates the schema** and the **SQL queries.**
- In the Javascript ecosystem, we have similar mechanisms. **With the particular yeoman generator that we use for the project**:
  - The authors have decided not use a relational database, but rather the **mongodb** document-oriented database.
  - They have decided to use **one of the data mapping tools** available for mongodb, namely **mongoose.** Since mongodb is a document-oriented database, it is more appropriate to talk about an Object-Document Mapping tool, rather than an ORM.

Why is not completely correct to say that mongoose is an ORM for MongoDB?

Mongoose: an ODM for MongoDB

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

*"Mongoose provides a **straight-forward**, schema-based solution to **modeling** your application data and includes built-in type **casting**, **validation**, **query** building, business logic hooks and more, out of the box."*



"Everything in Mongoose starts with a Schema. Each **schema maps to a MongoDB collection** and defines the shape of the documents within that collection."

"**Models are fancy constructors** compiled from our Schema definitions."

"Mongoose documents represent a **one-to-one mapping** to documents as stored in MongoDB. Each document is an **instance of its Model**."

http://mongoosejs.com/docs/guide.html

Example

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

schema

model

collection

```
var userSchema = new mongoose.Schema({
  name: {
    first: String,
    last: { type: String, trim: true }
  },
  age: { type: Number, min: 0 }
});
```

```
var PUser = mongoose.model('PowerUsers', userSchema);

var johndoe = new PUser ({
  name: { first: 'John', last: '  Doe   ' },
  age: 25
});

johndoe.save(function (err) {if (err) console.log ('Error on save!')});
```

document

https://devcenter.heroku.com/articles/nodejs-mongoose

Example: query

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

we can chain conditions

```
Person
.find({ occupation: /host/ })
.where('name.last').equals('Ghost')
.where('age').gt(17).lt(66)
.where('likes').in(['vaporizing', 'talking'])
.limit(10)
.sort('-occupation')
.select('name occupation')
.exec(callback);
```

we are interested in only some of the fields

we only want to get at most 10 documents

http://mongoosejs.com/docs/queries.html