IRON
HACK

Advanced JavaScript I

@salvadelapuente
salva@unoyunodiez.com

@gtorodelvalle
gtorodelvalle@gmail.com

The goal is to translate a server-supported web application into a **single page application**.

Each chapter has a **main exercise**.
The exercise will lead you to use
several JavaScript techniques.

Techniques and other concepts are explained during the chapter. In addition, **lots of references** are provided for further reading.

Index

1. Rationale: why SPA? Why not?
2. Client navigation
3. Modularization and HTML templating
4. Asynchronous JavaScript and JSON

Rationale:
Why SPA? Why not?

Why SPA?

- ✓ Improved server performance.
- ✓ Presentation is decoupled from data.
- ✓ Better UI.

Why not?

- ✗ Break of the Web paradigm: where are URL!?
- ✗ Crawler opaqueness.
- ✗ Slow UI.

Further reading

Comparison of JavaScript Frameworks

Are single page apps bad?

Important considerations when building single page web apps

The Tech Behind the New Twitter.com

The Tree Slider

Exercise

Start the rails server and perform the following queries with curl:

```
$ curl -X GET localhost:3000/posts

$ curl -X GET -H "Accept: application/json" localhost:3000/posts

$ curl -X GET localhost:3000/posts/1

$ curl -X GET -H "Accept: application/json" localhost:3000/posts/1
```

Client navigation

Chapter Exercise: the router | tag v1-router

Create a piece of JS to handle the navigation in the client. Assume your navigation sections are marked in the HTML with the attribute:

`data-navigation-section=”<name>”`

The code should intercept click events for links and use the href attribute to show the specific section according to a given map of patterns and sections. It should be convenient to call a custom callback for each navigation section to populate the view:

```
var ROUTES = {
  '/$':                   ['post-list', postListView],
  '/posts$':              ['post-list', postListView],
  '/posts/(\\d+)$':       ['show-post', showPostView],
  '/posts/(\\d+)/edit$': ['edit-post', editPostView],
  '/posts/new$':          ['new-post', newPostView]
};
```

First step is to **prevent the default navigation** by adding a custom callback on anchor elements.

The anchor elements <a> live in the document, let's see how to **ask the document** for these nodes.

The Document Object Model

- The DOM is a structure representing the document as a tree of nodes.
- Nodes can be of several types: html, text, comments... Those of html type are called elements.
- There are methods focused on elements.
- Most return HTMLCollection instances: they are always up to date so query once and cache!

Code sample: querying the DOM

```
// Get the element with the id set to 'an-id'
var element = document.getElementById('an-id');

// Get the live list of all <a> nodes
var liveList = element.getElementsByTagName('a');

// Get the live list of all nodes with class 'a-class'
var liveList = element.getElementByClassName('a-class');

// Get the first element in the document matching the CSS selector
var element = element.querySelector('#css .valid .selector');

// Get all the elements in the document matching the CSS selector
var simpleList = element.querySelectorAll('#css .valid .selector');

// Notice only getElementById() mut be applied on document. The remaining
// methods can be applied to other elements to restrict the scope.
```

Now we have to attach a listener for clicks on anchor elements to prevent default navigation from happening and replace it with the client one.

The event model

- Events are the way JavaScript uses to say **something has happened**.
- To react upon an event, you need a listener or callback.
- A listener or callback is no more than a function wich will **receive the event** and process it.
- HTML5 defines two event models: bubbling and capturing.

Code sample: attaching event listeners

```
// To add the function as callback
element.onclick = function (event) {
  /* Process the event. */
};

// To add the function as a new listener
element.addEventListener('click', function (event) {
  event.preventDefault(); // to avoid the browser's default action
  event.stopPropagation(); // to stop bubbling to / being capture by the
next element
}, false);

// The last parameter indicates if using capture (true) or bubbling
(false). Listeners added to the capture stage are not the same as those
added to the bubbling stage.

// To remove a listener you need a reference to it
function sayHello(evt) { alert('Hello!'); }
element.addEventListener('click', sayHello, true);
element.removeEventListener('click', sayHello, false); // This won't work
element.removeEventListener('click', sayHello, true);
```

Code sample: detect the DOM is parsed

```
// This snippet allows you to only run your code once the DOM
// is completely parsed.
if (document.readyState !== 'loading') {
  doInitialization();
}
else {
  document.addEventListener('DOMContentLoaded', function onDOMParsed() {
    document.removeEventListener('DOMContentLoaded', onDOMParsed);
    doInitialization();
  });
}

function doInitialization() {
  /* Perform initialization tasks */
}
```

Now you can intercept browser's navigation. Use the href attribute and see which pattern is following.

Each pattern is a **regular expression**. Find the first matching pattern and move to that section.

Regular expressions

- Regular expressions are a **language to describe strings**.
- Writing brackets we can **capture substrings** inside a string.
- Not all languages can be expressed using regular expressions (i.e. HTML5).
- String methods: split, replace, match & search.
- RegExp methods: test, exec.

Code sample: function examples

```
// string.split(): split by no alphabetical characters
var r = 'aBc2dE fg-hI'.split(/[^a-z]/i);
console.log(r);
// ['aBc', 'dE', 'fg', 'hI']

// string.replace(): replace no alphabetical characters by a dash
// You need the 'g' or you will replace only the first occurrence
var r = 'aBc2dE fg-hI'.replace(/[^a-z]/ig, '-');
console.log(r);
// aBc-dE-fg-hI

// string.match(): obtain the color components in a #RRGGBB triplet
var r = '#ff05A2'.match(/#([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]{2})/i);
console.log(r);
// ['#ff05A2', 'ff', '05', 'A2']

// string.search(): looking for the @ character inside a string
var r = 'hola@unoyunodiez.com'.search(/@/);
console.log(r);
// 4

// regexp.test(): returns true if the string is described by the regexp
/#[0-9a-f]{2}[0-9a-f]{2}[0-9a-f]{2}/i .test('#af01D2'); // true
/#[0-9a-f]{2}[0-9a-f]{2}[0-9a-f]{2}/i .test('#XX01D2'); // false

// regexp.exec(): returns the same as string.match()
```

Now you know which route you should follow. **Hide the current section, show the new one**…

…and **call the view function** with the section, querystring parameters and captured groups as **arguments**.

Methods .apply() and .call()

- **Functions are objects**. Special objects that can be executed. To execute a function, call its special method .apply().
- When executed, functions have two special variables set by JavaScript: this & arguments.
- When called, ***this*** **is the object from which we call the function** and ***arguments*** **is a list with the parameters**.
- But you can alter these automatically determined values by using .apply(), .call() and .bind().

Code sample: setting *this*

```
var anObject = {
  testThis: function (obj) { console.log(this === obj); }
};
var anotherObject = {};

// Normally, `this` is the object from which we call the function
anObject.testThis(anObject); // will print true
anObject['testThis'](anObject); // will print true

// As a function is an object, I can assign it...
var f = anObject.testThis;
// ...now use it!
f(anObject); // will print false
f(window); // will print true (in strict mode, print false)
f(undefined); // will print false (in strict mode, print true)

// With .apply() or .call() I can force the `this` value
f.apply(anotherObject, [anotherObject]); // will print true
f.call(anotherObject, anotherObject); // will print true
// First parameter is the value for `this`

// With bind() I create another function where `this`
// will be forced to be some value.
var f = anObject.testThis.bind(anotherObject);
f(anotherObject); // will print true
```

Code sample: setting *arguments*

```
var anObject = {
  testArguments: function () {
    var args = [].slice.call(arguments);
    console.log(args.length + ' arguments: '+ args);
  }
};
var someArguments = [1, null, 3, 'hello'];

// Normally, `arguments` is bound to the list of passed arguments
anObject.testArguments(1, 2, 3); // will print 3 arguments: 1, 2, 3

// But I can force it to be any iterable
anObject.testArguments.apply(anObject, someArguments);
// will print 4 arguments: 1, , 3, hello

// Or specify each separatedly
anObject.testArguments.call(anObject); // will print 0 arguments:
anObject.testArguments.call(anObject, 'hello');
// will print 1 arguments: hello

// With bind() I create another function where some of
// the parameters are already set.
var f = anObject.testArguments.bind(anObject, 1, 2);
f(3, 4); // will print 4 arguments: 1, 2, 3, 4
```

Some additional advices you should take into account…

List manipulation and loops

- Remember *for* loops admit **more than one sentence** in each clausule.
- **Cache the length of the iterable**, avoid to recalculate in each iteration.
- Lot of functions return array-like objects. To convert into JS arrays, **.call() the Array's .slice() method**.
- Use functional style Array's methods: every, filter, forEach, map, reduce, reduceRight and some.
- Use ***for ... in ...*** in addition to ***hasOwnProperty()*** to retrieve object's keys.

Code sample: loops

```
// Reverse a list
var list = [1, 2, 3, 4, 5];
for (var aux, i = 0, j = list.length - 1; i < j; i++, j--) {
  aux = list[j];
  list[j] = list[i];
  list[i] = aux;
}

// Cache the length
var links = document.getElementsByTagName('a');
for (var i = 0, l = links.length; i < l; i++) {
  /* Do things with links[i]... */
}

// Arguments is not an array...
function f() {
  var args = [].slice.call(arguments);
  // ...but args is an array now.
}

// Pass through the keys of an object
var object = { a: 1, b: 2, c: 3 };
for (var key in object) if (object.hasOwnProperty(key)) {
  /* Do things with the object key */
}
```

Code sample: functional style

```javascript
var numbers = [1, 2, 4, 6, 9];
function isEven(number) { return number % 2 === 0; }

// .every() returns true if the function returns
// true for every item in the list.
var allEven = numbers.every(isEven);

// .filter() returns a new list with the objects passing the function
var evenNumbers = numbers.filter(isEven);

// .forEach() pass through the items performing an action
function print(value) { console.log(value); }
numbers.forEach(print);

// .map() apply the function on each element in the array
function x2(value) { return 2 * value; }
var doubleNumbers = numbers.map(x2);

// .reduce() progressively applies the function for each element in the list
function concatenate(a, b) { return a + b; }
var concatenation = numbers.reduce(concatenate, '');

// .some() returns true if the function returns true for some item
var someEven = numbers.some(isEven);
```

Further reading

Live DOM viewer

JavaScript Array method reference

Array's iteration methods

Functional programming

Regex 101 for JavaScript

Learning JavaScript with Object Graphs

...and more!

Regex Crossword

The event model

JavaScript metaprogramming

Learn Regular Expressions

Exercise

Notice if you click on your browser's back button, navigation is not working. Use the new HTML5 History API to re-enable browser navigation and to avoid breaking the web paradigm.

Exercise

Instead of providing all the navigation sections in the NAVIGATION_SECTIONS variable, try to automatically detect all of them.

Exercise

A common mistake is to add `/` to the routes. Change `ROUTES` patterns to allow an optional `/` at the end of the route to make your routes more reliable.

Modularization and HTML templating

Chapter Exercise: templates | tag v2-template

Create a JS library to handle HTML templates. A template is a piece of valid HTML marked with the custom attribute:

`data-template=”name”`

A template contains placeholders to fill with data from JS objects or arrays. A full Jasmine specification can be found on:

`/webapp/specs/templateSpec.js`

```
<li data-template="post-list.entry">
  <span class="title"><a
  href="/posts/{{ id }}">{{ title }}</a></span>
  <menu>
    <a class="icon" href="/posts/{{ id }}/edit">?</a>
    <a class="icon" data-method="delete" href="/posts/{{ id }}">?</a>
 </menu>
</li>
```

This time we count with a Jasmine specification file and a **test-runner**.
In addition, you can see at the views.js file to check the library will be used.

The goal is simple: just code, code and code until the tests are **green**!

First thing you will realize is the spec is testing members of a **module** but JS lacks from this feature…

Isolating JavaScript

- One of the main problems with JS is the lack of a module scope.
- The revealing module pattern try to solve the module problem by providing a ***module scope*** and a some way to *export* (reveal) functionality.
- To do this we use a so called autofunction passing the dependencies and the object where exporting into as parameters.
- An autofunction is a simpe function literal **called immediately after the definition**.

Code sample: revealing module patterns

```
// This is the original pattern
var myModule = (function () {
  'use strict' // Use only once! It has function scope.

  /* Your module initialization... */

  // Here you reveal your module.
  return {
    name1: aPublicMethod,
    ...
  };

}());

// But I prefer this one
(function (global, dependency1, dependency2) {
  'use stric'

  /* The module initialization... */

  global.myModule = {
    name1: aPublicMethod,
    ...
  };

}(this, dependency1, dependency2, ...));
```

Code sample: autofunctions

```
// This is a function definition.
function f() {
  alert('Hello!');
};

// This is a function literal (a definition can not appear at the right
// of an assignation).
var f = function () {
  alert('Hello!');
};

// This is a function literal too (a definition can not appear inside
// parenthesis).
(function () {
  alert('Hello!');
});

// Literals are object! So...
(function () {
  alert('Hello!');
}.call(undefined));

// Or simply
(function () {
  alert('Hello!');
}());
```

IRON HACK

Now you'll see the spec is testing Template instances, let's do some OOP programming...

(but before, you should understand the basis of **prototypical inheritance**)

The prototype chain

- In JavaScript code reuse is achieved by delegation. If a member can not be found inside an object, its prototype is looked for the same member instead.
- There are two forms of creating new objects: by augmentation or via new operator.
- When calling functions, ***this* is always the object from which retrieving the member**, no matter where the function really is.

Code sample: augmentation

```
// How a superhero is born
var human = Object.create(null);
human.force = 1;
human.speed = 1;
human.intelligence = 1;

var clarkKent = Object.create(human);
clarkKent.name = 'Clark Kent';
clarkKent.force = 10;
clarkKent.speed = 10;
clarkKent.fly = function () { console.log(this.name + ' is flying!'); };
clarkKent.describe = function () {
  console.log(this.name + ' wears like a hipster.');
};

var superman = Object.create(clarkKent);
superman.name = 'Superman';
superman.describe = function () {
  console.log(this.name + ' wearas a red cape.');
};
superman.getSecretId = function () {
  var alterEgo = Object.getPrototypeOf(this);
  return alterEgo.name;
};
```

Code sample: testing augmentation

```
// They are not the same object
console.log(superman === clarkKent);

// They have some specific / not shared methods
console.log(clarkKent.describe !== superman.describe);
console.log(clarkKent.describe());
console.log(superman.describe());

// But clarkKent is the prototype / delegate of superman
console.log(clarkKent === Object.getPrototypeOf(superman));

// And they share some behavior
console.log(clarkKent.fly === superman.fly);
superman.fly();
clarkKent.fly();
// Results are different because `this` is different in each call

// Any object can access its prototype
console.log('The man behind: ' + superman.getSecretId());

// There is no hierarchy with augmentation: we rely on duck typing
```

Trust me when I say you there is no more than objects and prototype chains in JS. But you can emulate some classical OOP concepts…

Classical OOP: classes

- In JavaScript, classes does not exists but they are simulated with functions.
- Every function has a *prototype* member to put **the functionality to be shared** by the class' instances.
- To create an instance, use new followed by a call to the function.
- The operator new performs an augmentation of the function's prototype member and pass the new object to the function as this.

Code sample: classes

```
function Human(name, force, speed, intelligence) {
  this.name = name;
  this.force = force || 1;
  this.speed = speed || 1;
  this.intelligence = intelligence || 1;
};

Human.prototype.describe = function() {
  console.log(this.name + ' looks like a normal guy.');
  console.log('Force: ' + this.force);
  console.log('Speed: ' + this.speed);
  console.log('Intelligence: ' + this.intelligence);
};

var clark = new Human('Clark Kent', 10, 10, 1);
var bruce = new Human('Bruce Bane', 2, 1, 10);

clark.describe();
bruce.describe();
console.log(clark.describe === bruce.describe);

// The best benefit: hierarchy is traceable, enable strong typing
console.log(clark instanceof Human);
console.log(bruce instanceof Human);
```

Code sample: classes **are** augmentation

```
function Human(name) { this.name = name; };

function newObject(constructor) {
  var constructorArguments = [].slice.call(arguments, 1);
  var that = Object.create(constructor.prototype);
  var those = constructor.apply(that, constructorArguments);
  return those === undefined ? that : those;
}

function instanceOf(instance, klass) {
  var proto = Object.getPrototypeOf(instance)
  var wanted = klass.prototype;
  while (proto) {
    if (proto === wanted) return true;
    proto = Object.getPrototypeOf(proto);
  }
  return false;
}

var clark = new Human('Clark Kent');
console.log(clark.name);
console.log(instanceOf(clark, Human));

var bruce = newObject(Human, 'Bruce Bane');
console.log(bruce.name);
console.log(instanceOf(bruce, Date));
```

Ok, that's ok, but what about templating, DOM manipulation and all those other stuff I need for the **main exercise**!?

DOM manipulations, reflows and repaints

- There are DOM operations to modify the current DOM structure.
- A reflow happens when a geometric property is changed or even when some attributes are read.
- A repaint happens when the change affects appearance.
- To minimize the chance of reflow we have several good practices: code reordering, detached manipulation, document fragments…

Code sample: element creation

```
// Element creation
var element = document.createElement('p');

// Some attributes are directly accesible while you need
// setAttribute() and getAttribute() for others.
element.id = 'paragraph-1';
element.setAttribute('title', 'Online documentation in the MDN');

// Classes are manipulated with their own methods: add, remove, contains
element.classList.add('info');

// To specify the textual content
element.textContent = 'You have more documentation in the MDN';

// Or the HTML content
element.innerHTML = 'You have more documentation in the
<strong>MDN</strong>';

// We said the element is offline because is not attached to the DOM
```

Code sample: DOM manipulation

```javascript
// To clone, add, remove and replace elements from the DOM
element.appendChild(anotherElement); // add an element
element.parentNode.removeChild(element); // remove the element
element.parentNode.replaceChild(newElement, element); // replaces the element
reference.parentNode.insertBefore(element, reference); // guess it!

var newElement = element.clone(true);
// Be careful, now newElement and element have the same id
// With true, all the nodes inside element have been copied as well

// Create a document fragment, fill it with some items and add it to the DOM
var ul = document.getElementById('list');
var item = document.createElement('li');
var container = document.createDocumentFragment();
for (var newItem, i = 1; i <= 31; i++) {
  newItem = item.clone(true);
  newItem.textContent = i;
  container.appendChild(newItem);
}
ul.appendChild(container);

// Differences between innerHTML and textContent
var element = document.createElement('span');
element.innerHTML = 'Hy <strong>there!</strong>';
console.log(element.textContent);
```

Pro gaze: recursion, tail recursion and asynchronous recursion

On recursivity

- Sometimes is natural to define a problem in terms of itself.
- Every recursive function has one or more base cases and a recursive case.
- Each recursive case must approach to the base case.
- Pro tip: so called TCO (tail-call optimization) could lead to virtually infinite recursion steps.

Code sample: recursion optimizations

```
// A simple definition: can handle concat(40000) but concat(50000) is too much.
function concat(n) {
  if (n === 0) return '';
  return n + concat(n - 1);
}
// Same characteristics but this form allow us to transform it into...
function tailConcat(n) {

  return concat(n, '');

  function concat(n, accum) {
    if (n === 0) return accum;
    return concat(n - 1, accum + n);
  }
}
// ...this other. The problem is each iteration takes a min of 4ms. Very slow!
function optimizedTailConcat(n, cb) {

  return concat(n, '', cb);

  function concat(n, accum, cb) {
    if (n === 0) cb(accum);
    setTimeout(function () {
      concat(n - 1, accum + n, cb);
    });
  }
}
```

Code sample: continuation

```
// Best of both worlds:
function fasterOptimizedTailConcat(n, cb) {

  return concat(n, '', cb, 1);

  function concat(n, accum, cb, it) {
    if (n === 0) cb(accum + '');
    // Space from asynchronous calls
    if (it === 1000) {
      setTimeout(function () {
        concat(n - 1, accum + n, cb, 1);
      });
    }
   // Speed of synchronous recursion
    else {
      concat(n - 1, accum + n, cb, it + 1);
    }
  }
}
```

Further reading

ECMAScript 6 modules: the future is now

Universal Module Definition

Object Playground

On Duck Typing

Organizing Programs Without Classes

Gecko reflow visualization

…and more!

Mastering recursive programming

Real-world examples of recursion

Exercise

Modify the template library to accept keys using dot notation. Start modifying the specification. Allow indices as well.

Exercise

Write a jsperf testcase to explore the distinct ways of maniputaling the DOM: via innerHTML with a string, by programatically building DOM nodes with the API or by using a DocumentFragment. Explain your observations.

Asynchronous JavaScript And JSON

IRON HACK

Chapter Exercise: AJAX model | tag v3-model

Create a JS proxy in charge of communicate with the service server and perform basic operations such as create, delete, modify and retrieve posts; and create, delete and list comments.

Running the server on localhost:3000, you can test the server with curl:

```
$ curl -X GET -H "Accept: application/json" localhost:3000/posts\?page\=2

$ curl -X GET -H "Accept: application/json" localhost:3000/posts/1

$ curl -X POST -H "Accept: application/json" -H "Content-Type: application/json" -d
'{"post":{"text":"Body","post_picture":"","title":"Title"}}' localhost:3000/posts/

$ curl -X PATCH -H "Accept: application/json" -H "Content-Type: application/json" -d
'{"post":{"text":"Body","post_picture":"","title":"Title"}}' localhost:3000/posts/1

$ curl -X DELETE -H "Accept: application/json" localhost:3000/posts/1

$ curl -X GET localhost:3000/posts/1/comments

$ curl -X DELETE -H "Accept: application/json" localhost:3000/posts/1/comments/2

$ curl -X POST -H "Accept: application/json"-H -H "Content-Type: application/json"
-d '{“comment”:{“commenter”:“Mabel”,“body”:“Wadleeeeeees!”}}'
localhost:3000/posts/1/comments
```

The ramaining steps are simple, we have a RESTful web service and we need to communicate with it.

To do this, we need a model proxy in the client to abstract and expose the REST API.

And you have another **spec file** with a **sinon mocked XHR object** for easy testing.

Our blog is supported by a **RESTful** webservice allowing **CRUD** operations. What does it mean?

REST and CRUD

- REST is not a technology, nor a standard.
  REST is an architecture: a definition of componentes, their roles and relationships.
- In the Web, REST is supported by (but not limited to) the HTTP standard.
- From a Web Service perspective, the most popular principle behing REST is the uniform interface:
  - URLs to identify resources
  - CRUD operations on resources

Code sample: resource's REST API

```
class CommentsController < ApplicationController
  def create
    @post = Post.find(params[:post_id])
    @comment = @post.comments.create(params[:comment].permit(:commenter, :body))
    respond_to do |format|
      format.html { redirect_to post_path(@post) }
      format.json {
        render :json => { :status => :created, :post_id => @post.id,
                          :id => @comment.id },
               :status => :ok, :location => @comment
      }
    end
  end

  def destroy
    @post = Post.find(params[:post_id])
    @comment = @post.comments.find(params[:id])
    @comment.destroy
    respond_to do |format|
      format.html { redirect_to post_path(@post) }
      format.json {
        render :json => { :status => :deleted },
               :status => :ok
      }
    end
  end

  def index
    @comments = Post.find(params[:post_id]).comments
    respond_to do |format|
      format.json { render json: @comments }
    end
  end
end
```

An how can we access a REST API using **HTTP**?

HTTP for RESTful API

- A priori, HTTP fulfill the constrains of REST so it is possible to implement REST with HTTP. We need:
  - The URL to access the resource.
  - The HTTP verb (POST, GET, PATCH, DELETE) for the method.
  - The HTTP “Accept” and “Content-Type” headers for the resource's representation as a MIME type.
  - A representation format such as JSON, HTML, XML, JPEG, ZIP...

Code sample: HTTP request and response

```
GET /trends?k=3f1e9b3007&pc=true&src=module HTTP/1.1
Host: twitter.com
pragma: no-cache
accept-encoding: gzip,deflate,sdch
x-requested-with: XMLHttpRequest
accept-language: en-GB,en-US;q=0.8,en;q=0.6,es;q=0.4
user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.77
Safari/537.36
accept: application/json, text/javascript, */*; q=0.01
cache-control: no-cache
cookie: guest_id=....; original_referer=...
referer: https://twitter.com/

HTTP/1.1 200 OK
cache-control: private, max-age=300
content-encoding: gzip
content-length: 548
content-type: text/javascript; charset=utf-8
date: Sat, 25 Jan 2014 15:56:37 GMT
expires: Sat, 25 Jan 2014 16:01:37 GMT
last-modified: Sat, 25 Jan 2014 15:51:37 GMT
ms: A
server: tfe
set-cookie: _twitter_sess=...; Path=/; Domain=.twitter.com; Secure; HTTPOnly
status: 200 OK
strict-transport-security: max-age=631138519
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-transaction: 30099ce584e0e1f5
x-xss-protection: 1; mode=block

{"module_html":"<div class=\"flex-module trends-container \">...</div>","personalized":true,"woeid":1}
```

Ok, that's cool in the server side with all the curl-like stuff but what about the **browser**?

The XMLHttpRequest object

- To perform specific HTTP requests, IE (yes, Microsoft) introduced the XMLHttpRequest (XHR since now) object.
- The XHR allows the browser to make an asynchronous request.
- Asynchronous (tasks) means once the operation is started, the program flow is immediately returned to the caller not waiting for the result that can come at any moment.
- Requests end with a status code indicating the result of the request.

Code sample: a complete XHR example

```
// This creates a new XHR object
var xhr = new XMLHttpRequest();

// To setup, start by openning a connection with a verb and a location
xhr.open('POST', 'http://localhost:3000/posts/');

// Which format is the client expecting as answer
xhr.setRequestHeader('Accept', 'application/json');
xhr.responseType = 'json'; // response is transformed into a JS object

// In which format is the data sent to the server
xhr.setRequestHeader('Content-Type', 'application/json');

// Callbacks, this will be called every time the request progress to a new state
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) { // the xhr has finished
    if (xhr.status === 200) { // the result is OK
      /* Inform about the success */
    }
    else {
      /* Inform about the error */
    }
  }
};

// Finally send the request
xhr.send('{"resource":{"name":"Salva"}}');
```

Further reading

New Tricks in XMLHttpRequest2

How to GET a Cup of Coffee

Best Practices for a Pragmatic RESTful API

JSON.org

JSON at MDN

Understanding HATEOAS

HATEOAS and the PayPal REST Payment API

Exercise

Add some expectations to check the callback is called with null as first parameter when success (status code in [200, 299] range).

Exercise

Add some tests to check the callback is called with the error code as first parameter when the status code is not in the [200, 299] range.

And thats all young padawan... no go and code in peace.

Seriously?

Not!

There is a lot of room for improvement. Now we're going to **refactor views.js** and go deeper in **JS OOP**.

But before, get used to the code by doing some of these **tasks**:

Exercise

Display the comments from the most recent to the oldest one.

Exercise

The API accepts data URL as the picture format. Complete creating / updating posts by adding the picture. You will need to use the File API and read the selected file as data URL.

Exercise

Note how we encapsulate a lot of functions inside a view function. The problem with this approach is every time you call the view function, all the inner functions are recreated. Could you think about a better approach for this where much less memory was spent?

Advanced JavaScript I