nodeJs , npm , express

Table des matières

I - Vue d'ensemble (node , npm , express, ...)

1. Ecosystème node+npm

node (nodeJs) est un **environnement d'exécution javascript** permettant essentiellement de :
- compartimenter le code à exécuter en **modules** (import/export)
- exécuter du code en mode "**appels asynchrones non bloquants** + callback" (sans avoir recours à une multitudes de threads)
- exécuter directement du code javascript sans avoir à utiliser un navigateur web

npm ***(node package manager)*** est une sous partie fondamentale de node qui permet de :
- **télécharger et gérer des packages utiles à une application** (librairies réutilisables)
- télécharger et utiliser des utilitaires pour la phase de développement (ex : grunt , jasmine , gulp , …)
- **prendre en compte les dépendances entre packages** (téléchargements indirects)
- générer éventuellement de nouveaux packages réutilisables (à déployer)
- ….

node est à peu près l'équivalent "javascript" d'une machine virtuelle java.
npm ressemble un peu à maven de java : téléchargement des librairies , construction d'applications.

Un **projet basé sur npm** se configure avec le fichier ***package.json*** et les packages téléchargés sont placés dans le sous répertoire **node_modules** .

Principales utilisations/applications de node :
- application "serveur" en javascript (répondant à des requêtes HTTP)
- application autonome (ex : StarUML2 = éditeur de diagrammes UML , …)
- ….

2. Express

Express correspond à un des packages téléchargeables via npm et exécutables via node.
L**a technologie "express" permet de répondre à des requêtes HTTP** et ressemble un peu à un Servlet java ou à un script CGI .

A fond **basé sur des mécanismes souples et asynchrones (avec "routes" et "callbacks")** ,
"express" permet de coder assez facilement/efficacement des applications capables de :
- **générer dynamiquement des pages HTML** (ou autres)
- mettre en œuvre des **web services "REST"** (souvent au format "JSON") .
- prendre en charge les détails du protocoles **HTTP** (authentification "basic" et/ou "bearer" , autorisations "CORS" , ….)
- ….

"express" est souvent considéré comme une technologie de bas niveau lorsque l'on la compare à d'autres technologies "web / coté serveur" telles que ASP , JSP , PHP , …

"express" permet de construire et retourner très rapidement une réponse HTTP (avec tout un tas de

paramétrages fin si nécessaire) . Pour tout ce qui touche au format de la réponse à générer , il faut utiliser des technologies complémentaires (ex : templates de pages HTML avec remplacements de valeurs) .

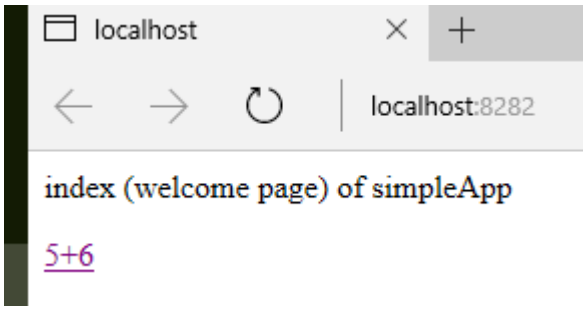
3. Exemple élémentaire "node+express"

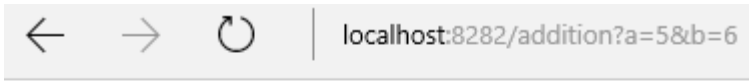
first_express_server.js

```
//modules to load:
var express = require('express');

var app = express();

app.get('/', function(req, res , next) {
     res.setHeader('Content-Type', 'text/html');
     res.write("<html> <body>");
     res.write('<p>index (welcome page) of simpleApp</p>');
     res.write('<a href="addition?a=5&b=6">5+6</a>');
     res.write("</body></html>");
     res.end();
});

//GET addition?a=5&b=6
app.get('/addition', function(req, res , next) {
     a = Number(req.query.a);   b = Number(req.query.b);
     resAdd = a+b;
     res.setHeader('Content-Type', 'text/html');
     res.write("<html> <body>");
     res.write('a=' + a + '<br/>');  res.write('b=' + b + '<br/>');
     res.write('a+b=' + resAdd + '<br/>');
     res.write("</body></html>");
     res.end();
});

app.listen(8282 , function () {
  console.log("simple express node server listening at 8282");
});
```

lancement : **node first_express_server.js**

via http://localhost:8282 au sein d'un navigateur web , on obtient le résultat suivant :





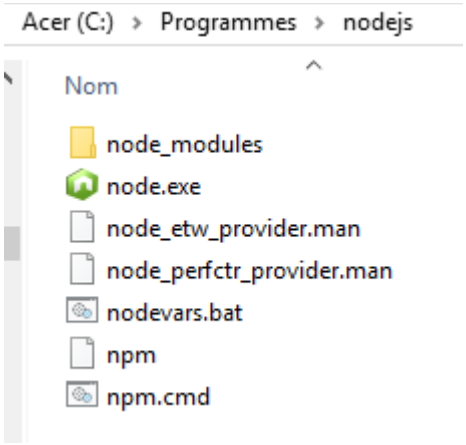
a=5
b=6
a+b=11

II - Node et npm : installation et utilisation

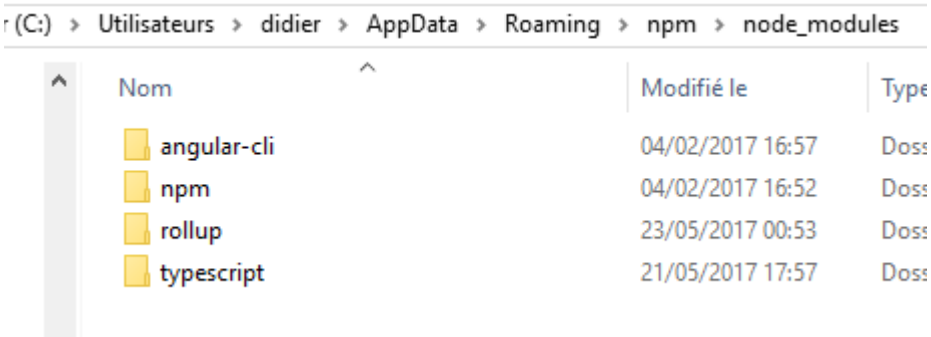
1. Installation de node et npm

Téléchargement de l'installeur **node-v6.10.3-x64.msi** (ou autre) depuis le site officiel de nodeJs
(https://nodejs.org)
Lancer l'installation et se laisser guider par les menus.
Cette opération permet sous windows d'installer node et npm en même temps .

Sur une machine windows 64bits , nodejs s'installe par défaut dans ***C:\Program Files\nodejs***



Et le répertoire pour les installations de packages en mode "global" (-g) est par défaut
C:\Users\username\AppData\Roaming\npm\node_modules



Vérification de l'installation (dans un shell "CMD") :

node --version
v6.9.4 (ou autre)

npm --version
4.1.2 (ou autre)
4.1.3

2. Configuration et utilisation de npm

2.1. Initialisation d'un nouveau projet

Un développeur utilise généralement npm dans le cadre d'un projet spécifique (ex : xyz).
Après avoir créer un répertoire pour ce projet (ex : C:\tmp\temp_nodejs\xyz) et s'être placé dessus, on peut lancer la c*ommande interactive* **npm init** de façon à **générer un début de fichier "*package.json*"**

Exemple de fichier ***package.json*** généré :

```
{
  "name": "xyz",
  "version": "1.0.0",
  "description": "projet xyz",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "didier",
  "license": "ISC"
}
```

2.2. installation de nouveau package en ligne de commande :

npm install --save express
npm install --save mongoose

permet de télécharger les packages "express" et "mongoose" (ainsi que tous les packages indirectement nécessaires par analyse de dépendances) dans le sous répertoire **nodes_modules** et de mettre à jour la liste des dépendances dans le fichier **package.json** :

```
…. ,
"dependencies": {
   "express": "^4.15.3",
   "mongoose": "^4.11.0"
  },
….
```

Sans l'option **--save** , les packages sont téléchargés mais le fichier package.json n'est pas modifié.

Par défaut , c'est la dernière version du package qui est téléchargé et utilisé.
Il est possible de choisir une **version spécifique** en la précisant après le caractère @ :

npm install --save mongoose@4.10

Autre procédure possible :

1) **éditer** le fichier ***package.json*** en y ajoutant des dépendances (au sein de la partie "dependencies") :
   exemple :
   "dependencies": {
   "express": "^4.15.3",
   "markdown": "^0.5.0",
   "mongoose": "^4.10.8"
   }
2) lancer **npm install** (ou *npm update* ultérieurement) sans argument

Ceci permet de lancer le téléchargement et installation du package "mardown" dans le sous répertoire **node_modules** .

Installation de packages utilitaires (pour le développement) :

Si l'on souhaite ensuite expliciter une dépendance de "développement" au sein d'un projet , on peut utiliser l'option **--save-dev** de **npm install** de façon ajouter celle ci dans la partie "devDependencies" de package.json :

npm install **--save-dev grunt**

```
…. ,
"devDependencies": {
   "grunt": "^1.0.1"
 }
...
```

2.3. Installation en mode global (-g)

L'option **-g** de **npm install** permet une installation en mode global : le package téléchargé sera installé dans ***C:\Users\username\AppData\Roaming\npm\node_modules*** sous windows 64bits (ou ailleurs sur d'autres systèmes) **et sera ainsi disponible (en mode partagé) par tous les projets** .

Le mode global est souvent utilisé pour installer des packages correspondant à des "utilitaires de développement" (ex : grunt) .

Exemple :

npm install -g grunt

3. Utilisation basique de node

hello_world.js

```
console.log("hello world");
```

node ***hello_world.js***

III - Modules - nodeJs

1. Modules (export , import)

Un des grands atouts de nodejs est un programmation à base de **modules bien compartimentés**.

1.1. Module avec élément(s) exporté(s)

mycomputer_module.js

```
var myAddStringFct = function(a,b) {
  result=a+b;
  resultString = "" + a + " + " + b +" = " + result;
  return resultString;
};

exports.myAddStringFct = myAddStringFct;
```

NB: ***Seuls les éléments exportés seront vus par les autres modules !!!***

1.2. Importation de module(s)

basic_exemple_with_modules.js

```
//chargement / importation des modules :
var mycomputer_module = require('./mycomputer_module'); // ./ for searching in local relative
var markdown = require('markdown').markdown; // without "./" in node_modules sub directory

//utilisation des modules importés :
var x=5;
var y=6;
var resString = mycomputer_module.myAddStringFct(x,y);

console.log(resString);
var resHtmlString = markdown.toHTML("**"+resString+"**");
//NB: "markdown" est un mini langage de balisage
// où un encadrement par ** génère un équivalent de
// <strong> HTML (proche de <bold>)
console.log(resHtmlString);
```

node basic_exemple_with_modules

résultats:

5 + 6 = 11

5 + 6 = 11

IV - Express (essentiel)

1. Essentiel de Express

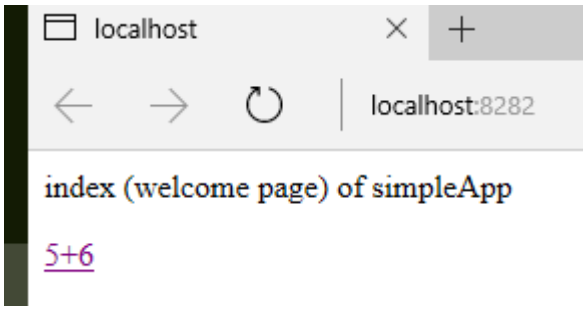
1.1. Exemple élémentaire (rappel)

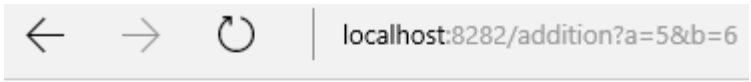
first_express_server.js

```
//modules to load:
var express = require('express');

var app = express();

app.get('/', function(req, res , next) {
     res.setHeader('Content-Type', 'text/html');
     res.write("<html> <body>");
     res.write('<p>index (welcome page) of simpleApp</p>');
     res.write('<a href="addition?a=5&b=6">5+6</a>');
     res.write("</body></html>");
     res.end();
});

//GET addition?a=5&b=6
app.get('/addition', function(req, res , next) {
     a = Number(req.query.a);   b = Number(req.query.b);
     resAdd = a+b;
     res.setHeader('Content-Type', 'text/html');
     res.write("<html> <body>");
     res.write('a=' + a + '<br/>');  res.write('b=' + b + '<br/>');
     res.write('a+b=' + resAdd + '<br/>');
     res.write("</body></html>");
     res.end();
});

app.listen(8282 , function () {
  console.log("simple express node server listening at 8282");
});
```

lancement : **node first_express_server.js**

via http://localhost:8282 au sein d'un navigateur web , on obtient le résultat suivant :





a=5
b=6
a+b=11

V - Web services REST avec express

1. WS REST élémentaire avec node+express

```
var express = require('express');
var app = express();
…
app.use(express.bodyParser()); //to parse JSON input data (req.body)

var sendGenericJsonExpressResponse = function(collectionName,query,res){
        res.setHeader('Content-Type', 'application/json');
        myDAO.genericFind(collectionName,query,onResultReady);
        function onResultReady(err,jsObject){
                res.write(JSON.stringify(jsObject));
                res.end();
        }
}

// GET (array) /minibank/operations?numCpt=1
app.get('/minibank/operations', function(req, res,next) {
        sendGenericJsonExpressResponse('operations',
                                        { 'compte' : Number(req.query.numCpt) },res);
});

// GET /minibank/comptes/1
app.get('/minibank/comptes/:numero', function(req, res,next) {
        sendGenericJsonExpressResponse('comptes', { '_id' : Number(req.params.numero) },res);
});

// POST /minibank/verifyAuth  {  "numClient" : 1 , "password" : "pwd1" }
app.post('/minibank/verifyAuth', function(req, res,next) {
        var verifAuth = req.body; // JSON input data with ok = null
        console.log("verifAuth :" +JSON.stringify(verifAuth));
         if(verifAuth.password == ("pwd" + verifAuth.numClient) )
                verifAuth.ok= true;
        else
                verifAuth.ok= false;
        res.send(verifAuth);    // send  back with ok = true or false
});

app.listen(8282 , function () {
  console.log("rest server listening at 8282");
});
```

NB : **req.query.**pxy récupère la valeur d'un paramètre http en fin d'URL (**?pxy=valXy**&pzz=valZz)
req.params.pxy récupère la valeur d'un paramètre logique (avec:) en fin d'URL

dans cet exemple , ***myDAO.genericFind(collectionName,query***,onResultReady); correspond à un

élément d'un module utilitaire qui récupère des données dans une base mongoDB .

2. Autorisations "CORS"

```
var express = require('express');
var app = express();
…

// CORS enabled with express/node-js :
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
                          //ou avec  "www.xyz.com" à la place de "*" en production
  res.header("Access-Control-Allow-Headers",
             "Origin, X-Requested-With, Content-Type, Accept");
  next();
});

…
…
app.get(...) , app.post(...) , …

app.listen(8282 , function () {
  console.log("rest server with CORS enabled listening at 8282");
});
```

VI - Accès aux bases de données (node)

1. Accès à un SGBDR (SQL) via node

```
npm install mysqljs/mysql
```

...

2. Accès à MongoDB (No-SQL , JSON) via node

2.1. Via mongodb/MongoClient (sans mongoose)

my_dao_module.js

```
//myDAO module (with MongoDB/MongoClient)
var MongoClient = require('mongodb').MongoClient;
var ObjectId = require('mongodb').ObjectID;
var assert = require('assert');

var executeInMongoDbConnection = function(callback_with_db) {
  var dbUrl = 'mongodb://localhost:27017/test' ; //by default
  MongoClient.connect(dbUrl, function(err, db) {
          if(err!=null) {
                  console.log("mongoDb connection error = " + err);
          }
          assert.equal(null, err);
          //console.log("Connected correctly to mongodb database" );
          callback_with_db(db);
        });
}


var genericUpdateOne = function(collectionName,id,changes,callback_with_err_and_results) {
   executeInMongoDbConnection( function(db) {
          db.collection(collectionName).updateOne( { '_id' : id }, { $set :  changes } ,
                        function(err, results) {
                                if(err!=null) {
                                 console.log("genericUpdateOne error = " + err);
                                }
                                callback_with_err_and_results(err,results);
                                db.close();
                        });
         });
};

var genericInsertOne = function(collectionName,newOne,callback_with_err_and_newId) {
```

```
  executeInMongoDbConnection( function(db) {
        db.collection(collectionName).insertOne( newOne , function(err, result) {
              if(err!=null) {
                console.log("genericInsertOne error = " + err);
                newId=null;
              }
              else {newId=newOne._id;
              }
              callback_with_err_and_newId(err,newId);
              db.close();
             });
  });
};

var genericFindList = function(collectionName,query,callback_with_err_and_array) {
   executeInMongoDbConnection( function(db) {
        var cursor = db.collection(collectionName).find(query);
        cursor.toArray(function(err, arr) {
              callback_with_err_and_array(err,arr);
              db.close();
             });
  });
};


var genericFindOne = function(collectionName,query, callback_with_err_and_item) {
 executeInMongoDbConnection( function(db) {
        db.collection(collectionName).findOne(query , function(err, item) {
              if(err!=null) {
              console.log("genericFindById error = " + err);
           }
            assert.equal(null, err);
            callback_with_err_and_item(err,item);
            db.close();
            });
  });
};

exports.genericUpdateOne = genericUpdateOne;
exports.genericInsertOne = genericInsertOne;
exports.genericFindList = genericFindList;
exports.genericFindOne = genericFindOne;
```

Utilisation :

```
var express = require('express');
var app = express();
var myDaoModule = require('./my_dao_module');

// GET /minibank/clients/1
app.get('/minibank/clients/:numero', function(req, res,next) {
```

```
   sendGenericJsonExpressSingleResponse('clients', { '_id' : Number(req.params.numero) },res);
});

var sendGenericJsonExpressSingleResponse = function(collectionName,query,res){
        res.setHeader('Content-Type', 'application/json');
        myDaoModule.genericFindOne(collectionName,query,onResultReady);
        function onResultReady(err,jsObject){
                res.write(JSON.stringify(jsObject));
                res.end();
        }
}

….
```

2.2. Via mongoose

Mongoose permet de mettre en œuvre une sorte de correspondance automatique entre un type d'objet javascript et un type document "mongoDB" .

ODM : Object Document Mapping.

Mode opératoire :

- on définit une structure de données (mongoose.Shema(....))
- on peut ensuite appeler des méthodes ".find() , .save() , .remove() , …" pour effectuer des opérations "CRUD" dans une base de données "mongoDB"

VII - Événements - nodeJs

1. event

...

VIII - NodeJs: aspects divers et avancés

1. Suppression/remplacement d'attribut javascript

```
var obj = { prenom:"jean", nom:"Bon" , age:25 };
console.log(JSON.stringify(obj));
obj.name=obj.nom;  //ajout de la propriété .name (par copie de la valeur de .nom)
console.log(JSON.stringify(obj));
delete obj.nom; //supression de la propriété .nom
console.log(JSON.stringify(obj));
```

2. Saisie asynchrone en mode texte (stdin)

2.1. Saisies implicitement en boucle

```
var stdin = process.openStdin();  //ou process.stdin

stdin.addListener("data", function(d) {
    // note:  d is an object, and when converted to a string it will
    // end with a linefeed.  so we (rather crudely) account for that
    // with toString() and then trim()
    console.log("you entered: [" +   d.toString().trim() + "]");
  });
```

2.2. Saisie unitaire avec question préalable et callback

```
var stdin = process.stdin;
var stdout = process.stdout;

function ask(question, callback) {
 stdin.resume();
 stdout.write(question + ": ");
 stdin.once('data', function(data) {
  data = data.toString().trim();
    callback(data);
  });
}

//utilisation chaînée avec callbacks imbriquées:

ask("x", function(valX){
                var x=Number(valX);
                        ask("y", function(valY){
                                        var y=Number(valY);
                                        var res=x+y ;
```

```
                                        console.log("res = (x+y)=" +res);
                                        process.exit();
                        });
    });
```

3. Promise , Q et enchaînements asynchrones

```
var Q = require('q');
var stdin = process.stdin;
var stdout = process.stdout;

function askPromise(question){
   var deferred = Q.defer();
   stdin.resume();
   stdout.write(question + ": ");
   stdin.once('data', function(data) {
      data =  data.toString().trim();
      deferred.resolve(data); //valeur renvoyée en différé (lorsque prête)
                              // ceci déclenchera .then(...) externe
   });
   return deferred.promise; //retour immédiat (non bloquant) d'une promesse
}
```

Utilisation (enchaînement contrôlé de "Promise") :

```
var x,y;
askPromise("x")
   .then( function (valX){
         console.log("valX="+valX); x=valX;
         return askPromise("y");
         })
   .then( function (valY){
         console.log("valY="+valY); y=valY;
         var res=Number(x)+Number(y);
         console.log("res=(x+y)=" + res);
         console.log("fin"); process.exit();
   });
```

ANNEXES

IX - Annexe – Tests (mocha , chai , ...)

1. Tests avec Mocha + Chain (env. nodeJs)

Mocha (+Chain) est la technologie de test préconisée pour nodeJs .
La technologie "mocha" ressemble beaucoup à jasmine . Elle doit être complétée par "chain" pour les assertions/vérifications (expect) .

Il est ainsi assez facile de tester unitairement un composant d'une application nodeJs.
En utilisant en plus le package "request" de façon à déclencher des requêtes HTTP, on peut également invoquer et tester un WS-REST construit avec node+express .

1.1. Structure et commandes

package.json
test/xy-spec.js
app/xy.js

Elément à ajouter/paramétrer dans package.json

```
"scripts": {
   "test": "./node_modules/.bin/mocha --reporter spec"
 },
```

//for nodemon xxx.js (watch mode) in place of node xxx.js
npm install -g nodemon

//for mocha test runner (in node):
npm install mocha --save

//for chai expect/assert:
npm install chai --save

//for rest ws test:
npm install request --save

//for rest app:
npm install express --save

Lancement des tests :

npm run test
ou bien
npm test

NB : Les exemples de code ci-après sont issus du tutorial
"*Getting Started with Node.js and Mocha – Semaphore*" accessible par recherche internet .

1.2. Test unitaire simple

Exemple de composant à tester :

app/converter.js

```
exports.rgbToHex = function(red, green, blue) {
  var redHex   = red.toString(16);
  var greenHex = green.toString(16);
  var blueHex  = blue.toString(16);
  return pad(redHex) + pad(greenHex) + pad(blueHex);
};

function pad(hex) {   return (hex.length === 1 ? "0" + hex : hex);
}

exports.hexToRgb = function(hex) {
  var red   = parseInt(hex.substring(0, 2), 16);
  var green = parseInt(hex.substring(2, 4), 16);
  var blue  = parseInt(hex.substring(4, 6), 16);
  return [red, green, blue];
};
```

exemple de test:

test/converter-spec.js

```
var expect    = require("chai").expect;
var converter = require("../app/converter");

describe("Color Code Converter", function() {
  describe("RGB to Hex conversion", function() {
    it("converts the basic colors", function() {
      var redHex   = converter.rgbToHex(255, 0, 0);
      var greenHex = converter.rgbToHex(0, 255, 0);
      var blueHex  = converter.rgbToHex(0, 0, 255);
      expect(redHex).to.equal("ff0000");
      expect(greenHex).to.equal("00ff00");
      expect(blueHex).to.equal("0000ff");
    });
  });

  describe("Hex to RGB conversion", function() {
    it("converts the basic colors", function() {
      var red   = converter.hexToRgb("ff0000");
      var green = converter.hexToRgb("00ff00");
      var blue  = converter.hexToRgb("0000ff");
      expect(red).to.deep.equal([255, 0, 0]);
      expect(green).to.deep.equal([0, 255, 0]);
      expect(blue).to.deep.equal([0, 0, 255]);
    });
  });
```

```
});
```

1.3. Test de web service REST

Exemple de service à tester :

app/server.js

```
var express = require("express");
var app = express();
var converter = require("./converter");

app.get("/rgbToHex", function(req, res) {
  var red   = parseInt(req.query.red, 10);
  var green = parseInt(req.query.green, 10);
  var blue  = parseInt(req.query.blue, 10);
  var hex = converter.rgbToHex(red, green, blue);
  res.send(hex);
});

//...
app.listen(3000);
```

Exemple de test :

test/server-spec.js

```
var expect  = require("chai").expect;
var request = require("request");

describe("Color Code Converter API", function() {
  describe("RGB to Hex conversion", function() {

    var url = "http://localhost:3000/rgbToHex?red=255&green=255&blue=255";

    it("returns status 200", function(done) {
      request(url, function(error, response, body) {
        expect(response.statusCode).to.equal(200);
        done();//pour marquer la fin du test (réponse traitee apres appel asynchrone)
      });
    });

    it("returns the color in hex", function(done) {
      request(url, function(error, response, body) {
        expect(body).to.equal("ffffff");
        done();
      });
    });

  });
});
```

X - Annexe – Utilitaires (grunt , gulp , ...)

.... insérer ici une section (liaison) vers le sous chapitre "partie1" de référence via raccourci
.... insérer ici une section (liaison) vers le sous chapitre "partie2" de référence via racc.

XI - Annexe – intégration continue / node

.... insérer ici une section (liaison) vers le sous chapitre "partie1" de référence via raccourci
.... insérer ici une section (liaison) vers le sous chapitre "partie2" de référence via racc.

XII - Annexe – WEB Services "REST"

1. Généralités sur Web-Services REST

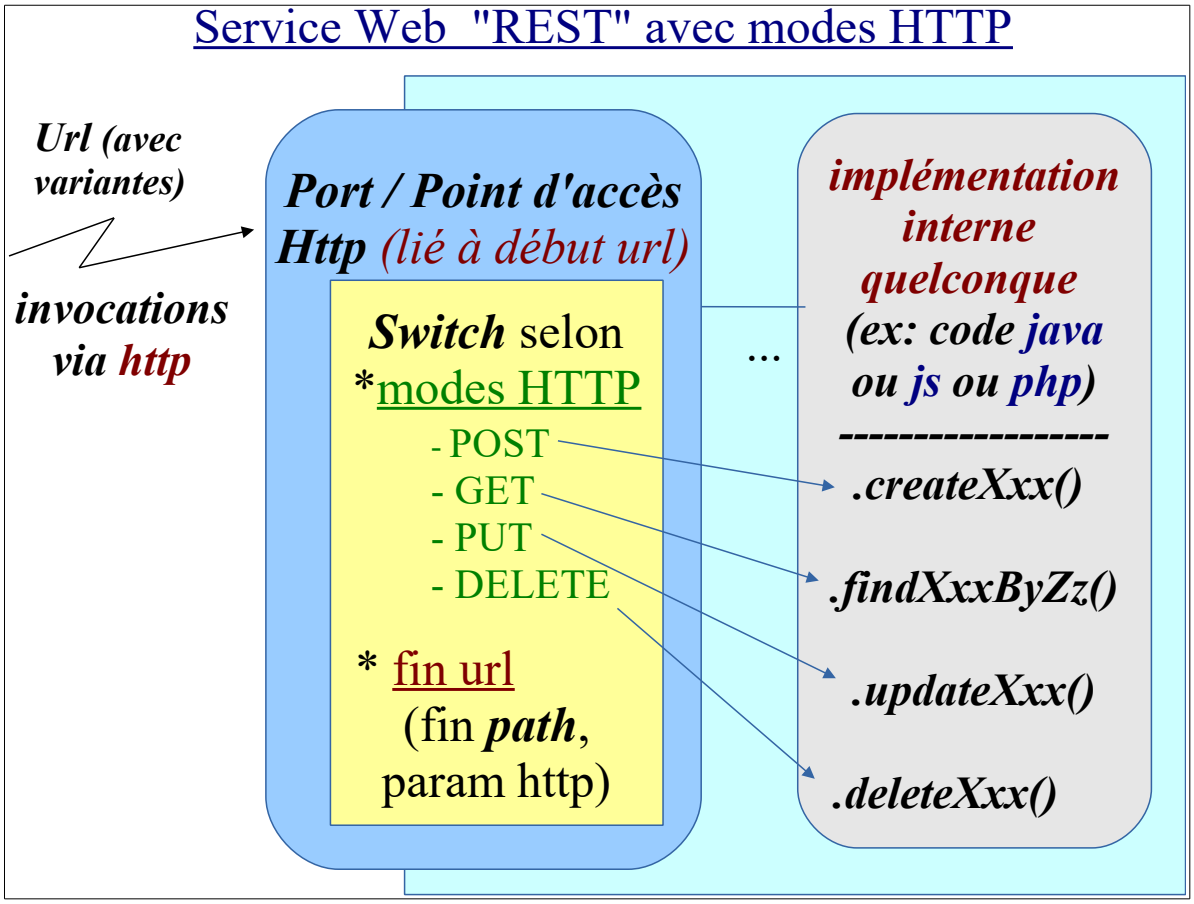
2 grands types de services WEB: **SOAP/XML** et **REST/HTTP**

WS- (SOAP / XML)*

- "**Payload**" systématiquement en **XML** *(sauf pièces attachées / HTTP)*
- **Enveloppe SOAP** en XML *(header facultatif pour extensions)*
- **Protocole de transport au choix (HTTP, JMS, ...)**
- Sémantique quelconque (*appels méthodes*) , **description WSDL**
- **Plutôt** orienté Middleware SOA **(arrière plan)**

REST (HTTP)

- **"Payload" au choix (XML , HTML , JSON, ...)**
- Pas d'enveloppe imposée
- **Protocole de transport = toujours HTTP.**
- Sémantique "**CRUD**" (***modes http* PUT,GET,POST,DELETE)**
- **Plutôt** orienté IHM Web/Web2 **(avant plan)**



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("XML" , "JSON" et éventuellement "txt" ou "html") les web-services "REST" offrent des résultats qui nécessitent généralement peu de re-traitements pour être mis en forme au sein d'une IHM web.

Le format "au cas par cas" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "JSON" les web-services "REST" conviennent parfaitement à des appels (ou implémentations) au sein du langage javascript .

La relative simplicité des URLs d'invocation des services "REST" permet des appels plus immédiats (*un simple href="..." suffit en mode GET pour les recherches de données*) .

La compacité/simplicité des messages "JSON" souvent associés à "REST" permet d'obtenir d'assez bonnes performances .

REST = style d'architecture (conventions)

REST est l'acronyme de **Representational State Transfert**.
C'est un **style d'architecture** qui a été décrit par ***Roy Thomas Fielding*** dans sa thèse *«Architectural Styles and the Design of Network-based Software Architectures».*

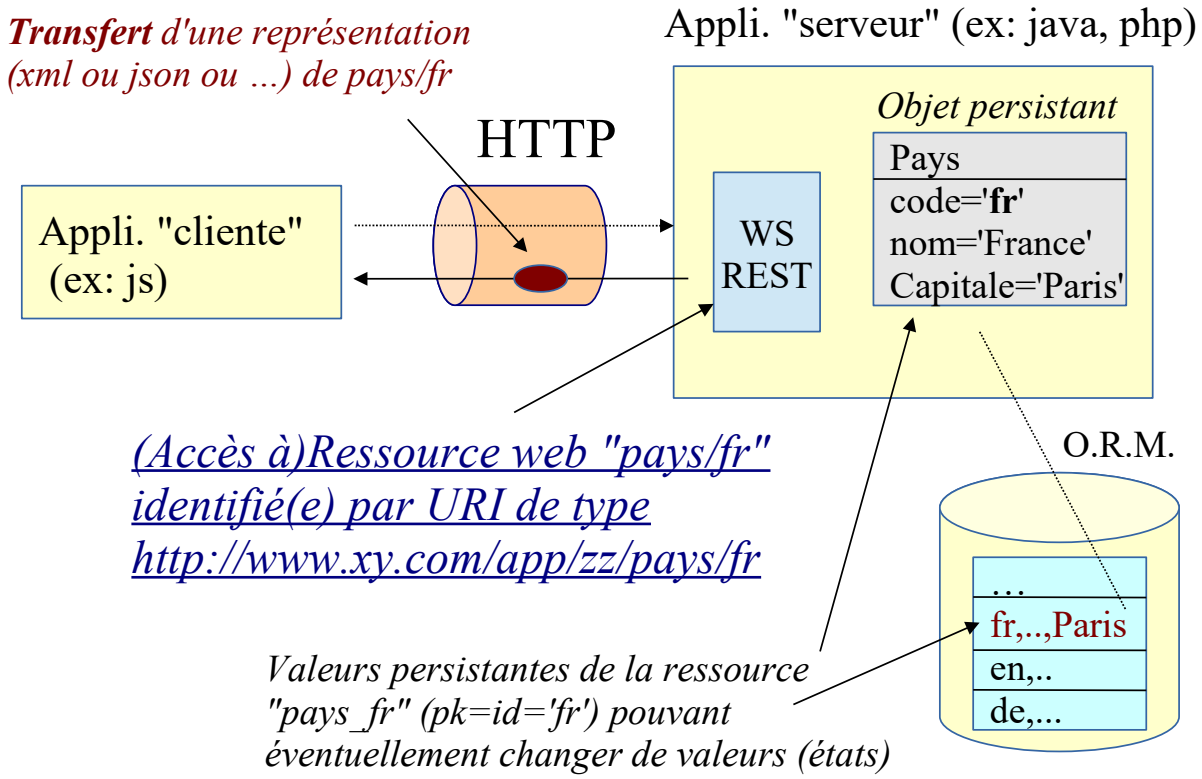
L'information de base, dans une architecture REST, est appelée **ressource**.
Toute information (à sémantique stable) qui peut être nommée est une ressource: un article , une photo, une personne , un service ou n'importe quel concept.

Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.
Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML , XML ou JSON**.

REST : transferts de représentations de ressources



REST et principaux formats (xml,json)

Une invocation d'URL de service REST peut être accompagnée de données (en entrée ou en sortie) pouvant prendre des formats quelconques :
text/plain , text/html , application/xml , application/json , …

Dans le cas d'une lecture/recherche d'informations , le format du résultat retourné pourra (selon les cas) être :

- **imposé (en dur) par le code du service REST** .

- **au choix (xml , json) et précisé par une partie de l'url**

- **au choix (xml , json) et précisé par le champ "Accept :" de l'entête HTTP de la requête**. (exemple: Accept: application/json) .

Dans tous les cas, la réponse HTTP devra avoir son format précisé via le champ habituel ***Content-Type:*** *application/json* de l'entête.

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JS0N sont :
- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

Les types de données valables sont :
- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

```
[
  {
    "nom": "article a",
    "prix": 3.05,
    "disponible": false,
    "descriptif": "article1"
  },
  {
    "nom": "article b",
    "prix": 13.05,
    "disponible": true,
    "descriptif": null
  }
]
```

une liste d'articles

une personne

```
{
  "nom": "xxxx",
  "prenom": "yyyy",
  "age": 25
}
```

REST et méthodes HTTP (verbes)

Les **méthodes HTTP** sont utilisées pour indiquer la **sémantique des actions demandées** :
- **GET** : **lecture/recherche** d'information
- **POST** : **envoi** d'information
- **PUT** : **mise à jour** d'information
- **DELETE** : **suppression** d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource **http://monsite.com/adherents**

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient http://monsite.com/adherents?ageMinimum=20

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que **http://monsite.com/adherents/4**

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

Exemple concret de service REST : "Elevation API"

L'entreprise "***Google***" fourni gratuitement certains services WEB de type REST.
"***Elevation API***" est un service REST de Google qui renvoie l'altitude d'un point de la planète selon ses coordonnées (latitude ,longitude) .
La documentation complète se trouve au bout de l'URL suivante :

https://developers.google.com/maps/documentation/elevation/?hl=fr

Sachant que les coordonnées du Mont blanc sont :
Lat/Lon : 45.8325 N / 6.86417 E *(GPS : 32T 334120 5077656)*
Les invocations suivantes (du service web rest "api/elevation")

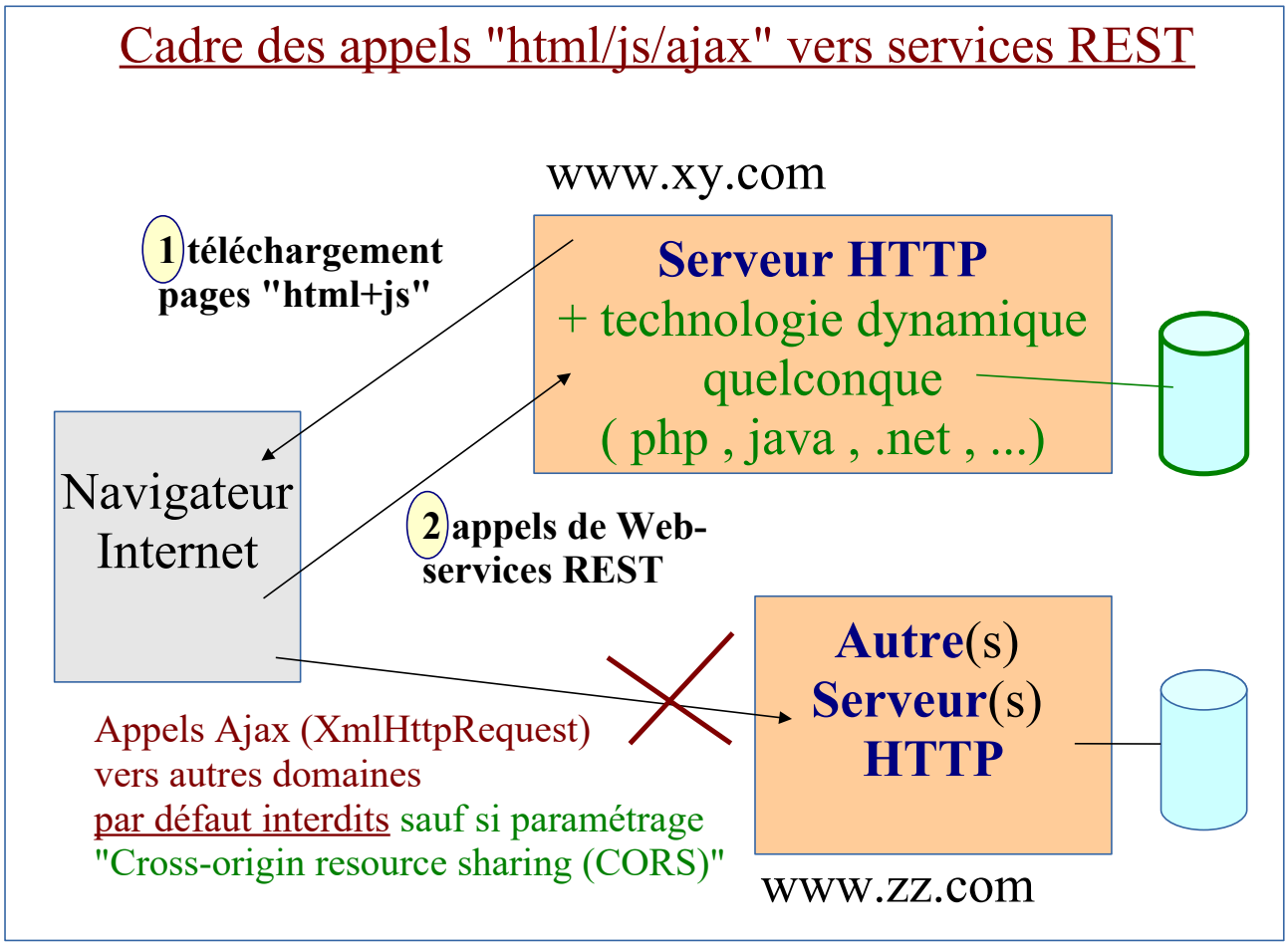
http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417
http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417

donne les résultats suivants "json" ou "xml":

```
{  "results" : [
     {
       "elevation" : 4766.466796875,
       "location" : {
         "lat" : 45.8325,
         "lng" : 6.86417
       },
       "resolution" : 152.7032318115234
     }
   ],   "status" : "OK"
}
```

```
?xml version="1.0" encoding="UTF-8"?>
<ElevationResponse>
 <status>OK</status>
 <result>
  <location>
   <lat>45.8325000</lat>
   <lng>6.8641700</lng>
  </location>
  <elevation>4766.4667969</elevation>
  <resolution>152.7032318</resolution>
 </result>
</ElevationResponse>
```

2. Limitations Ajax sans CORS



3. CORS (Cross Origin Resource Sharing)

CORS=Cross Origin Resource Sharing

CORS est une **norme du W3C** qui précise certains **champs** à placer dans une **entête HTTP** qui serviront à échanger entre le navigateur et le serveur des informations qui serviront à décider si une requête sera ou pas acceptée.

(utile si domaines différents) , dans requête simple ou bien dans pré-échange préliminaire quelquefois déclenché en plus :

Au sein d'une requête "demande autorisation" envoyée du client vers le serveur :

Origin: http://www.xy.com

Dans la "réponse à demande d'autorisation" renvoyée par le serveur :

Access-Control-Allow-Origin: http://www.xy.com
Ou bien
Access-Control-Allow-Origin: * *(si public)*
→ *requête acceptée*

Si absence de "Access-Control-Allow-Origin :" ou bien valeur différente
---> requête refusée

CORS=Cross Origin Resource Sharing (2)

NB1: toute requête "CORS" valide doit absolument comporter le champ "***Origin :***" dans l'entête http. Ce champ est toujours construit automatiquement par le navigateur et jamais renseigné par programmation javascript.
Ceci ne protège que partiellement l'accès à certains serveurs car un "méchant hacker" utilise un "navigateur trafiqué".
Les mécanismes "CORS" protège un peu le client ordinaire (utilisant un vrai navigateur) que dans la mesure ou la page d'origine n'a pas été interceptée ni trafiquée (l'utilisation conjointe de "https" est primordiale) .

NB2 : Dans le cas (très classique/fréquent) , où la requête comporte "***Content-Type: application/json***" (ou ***application/xml*** ou …) , la norme "CORS" (considérant la requête comme étant "pas si simple") impose un **pré-échange préliminaire** appelé "**Preflighted request/response**" .

Paramétrages CORS à effectuer coté serveur

L'application qui coté serveur, fourni quelques Web Services REST , peut (et généralement doit) autoriser les requêtes "Ajax / CORS" issues d'autres domaines ("*" ou "www.xy.com").
Attention: ce n'est pas une "sécurité coté serveur" mais juste **un paramétrage autorisant ou pas à rendre service à d'autres domaines et en devant gérer la charge induite** *(taille du cluster, consommation électrique, ...)* .

```
// Exemple : CORS enabled with express/node-js :
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*"); //"*" ou "xy.com , ..."
  res.header("Access-Control-Allow-Methods",
            "POST, GET, PUT, DELETE, OPTIONS"); //default: GET, ...
  res.header("Access-Control-Allow-Headers",
     "Origin, X-Requested-With, Content-Type, Accept , Authorization");
  next();
});
```

Paramétrages CORS avec CXF et JAX-RS

```
<bean id="corsFilter" class="org.apache.cxf.rs.security.cors.
                              CrossOriginResourceSharingFilter">
  <!--  <property name="allowCredentials" value="true"/> -->
</bean>
...
<jaxrs:server id="myRestServices"  address="/rest">
        <jaxrs:providers>
              <ref bean='jacksonJsonProvider' />
              <ref bean='corsFilter' />
        </jaxrs:providers>
        <jaxrs:serviceBeans> ...
              <ref bean="serviceClientsRest" />
        </jaxrs:serviceBeans>  …
```

config spring/cxf

```
@Path("/json/gestionclients")
@Produces("application/json")
@Consumes("application/json")
@CrossOriginResourceSharing(allowAllOrigins = true)
                                  // ou bien autorisations plus fines
public class ClientRestJsonService {
...}
```

code java

XIII - Annexe – Sécurité - WEB Services "REST"

1. Sécurité pour WS-REST (Http)

Sécurité pour HTTP et Web-service REST

Etant donné qu'un web-service "REST" est toujours invoqué via "http" ou "https", la sécurisation d'un de ses accès s'effectue via des mécanismes étroitement liés aux échanges http.

Il existe **plusieurs niveaux de sécurisation/authentification (plus ou moins complexes)**. Ceux ci vont être présentés de manière progressive :

• "Basic Http Auth." **rudimentaire** ou équivalent (très déconseillé)

"Basic Http Auth." ou équivalent **en véhiculant un mot de passe crypté** via **hash**() ou bien **doublement cryptée (avec "salt")**.

• Utilisation conjointe d'un **jeton ("token")** généré coté serveur , retourné au client et retransmis via le champ "**Authorization: Bearer**" de l'entête HTTP

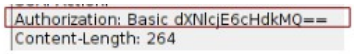
• *Pour cas sophistiqué/complexe:* **Signature des requêtes** (**HMAC**) avec ***clef secrète*** et requêtes à usage unique avec "***timestamp***" .

• Utilisation conjointe (quasi systématique) de **HTTPS/SSL** .

Positionnements possibles des informations d'authentification

En fin d'url :
http://www.xy.com/zz/resource1**?username=user1&password=pwd1**

Dans l'entête HTTP (avec **encodage base64** associé au standard "**basic http auth**") **:**



décodage immédiat (quasi "en clair")

Limitations d'une authentification rudimentaire

Placer en fin d'url (ou bien dans l'entête HTTP) **une information d'authentification en clair (ou à peine cryptée via un encodage base64)** permet seulement de limiter l'accès aux utilisateurs qui connaissent le couple ***username/password*** .

Dans le cas où un "*hacker*" intercepte la requête et en récupère une copie, il connaît alors tout de suite le mot de passe et peut alors déclencher toutes les actions qu'il désire en se faisant passer par l'utilisateur "*piraté*" .

"Basic Http Auth." ou fin d'URL avec hash(password)

Certains algorithmes standards de cryptage ("hachage") du **mot de passe** tels que "***md5***" ou "***sha1***" ou "***..256***" **rendent très difficile le décryptage de celui-ci** .

En **véhiculant** en fin d'URL (ou dans l'entête HTTP) le **mot de passe haché**
- **celui-ci ne circule plus en clair (meilleur confidentialité).**
- **la base de données des mots de passe ne comporte que des informations cryptées** et est donc moins vulnérable (si elle est piratée ou visualisée, les mots de passe "en clair" ne seront pas connus) .

Phases de la mise en oeuvre :
- Dès la saisie initiale du mot de passe, celui-ci est haché/crypté et stocké dans une base de données coté serveur.
- Lorsque le mot de passe est re-saisi coté navigateur , celui ci est de nouveau haché/crypté (via le même algorithme) avant d'être véhiculé vers le serveur
- Le serveur compare les deux "hachages/cryptages" pour vérifier une authentification correcte

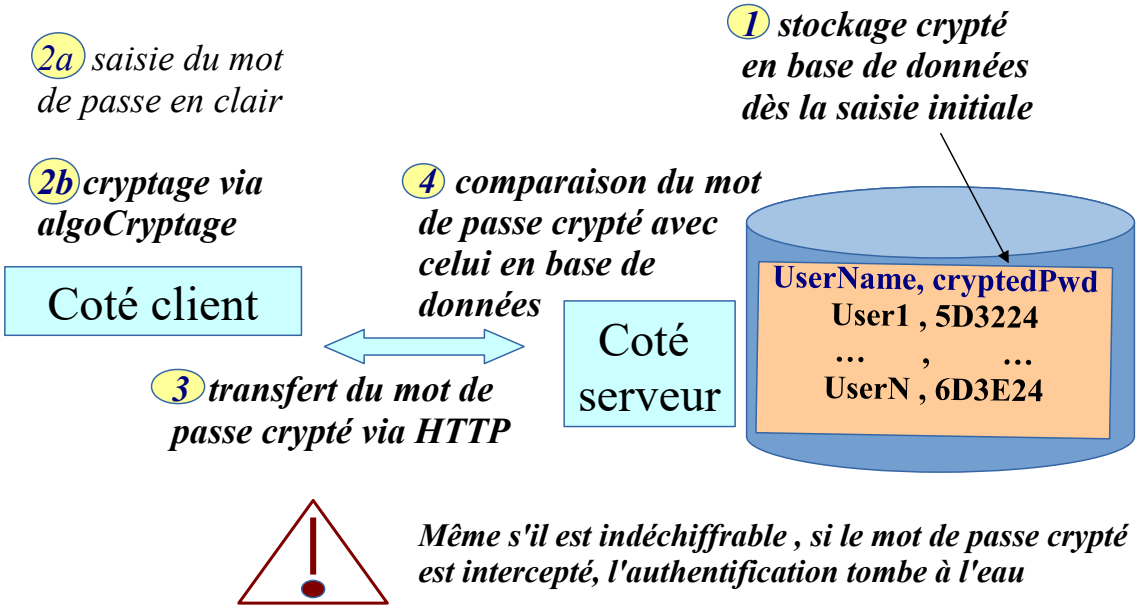
"Basic Http Auth." avec "hash" et "salt"

- Un simple cryptage/hachage "md5" , "sha1" ou autre ne suffit souvent pas car il existe des bases de données d'associations entre mots de passe courants et les hachages correspondants "md5" ou "sha1" facilement accessibles depuis le web.
- D'autre part, si le mot de passe (en clair) peut ainsi être indirectement découvert, ceci peut être extrêmement problématique dans le cas où l'utilisateur utilise un même mot de passe pour plusieurs sites ou applications .
- De façon à prévenir les risques présentés ci-dessus, on utilise souvent un **double encodage/cryptage prenant en compte une chaîne de caractère spécifique à l'application (ou à l'entreprise) appelée "salt" (comme grain de sel)** .

Exemple :
cryptedPwd = md5OuSha1("my_custom_salt" + md5OuSha1(pwd)) ;

"hash" (+ "salt") (récapitulatif)

AlgoCryptage = **md5OuSha1(pwd)** *ou bien*
md5OuSha1("my_custom_salt" + md5OuSha1(pwd)) ;



Même s'il est indéchiffrable , si le mot de passe crypté est intercepté, l'authentification tombe à l'eau

Algo. "bcrypt" pour les "password" stockés en base

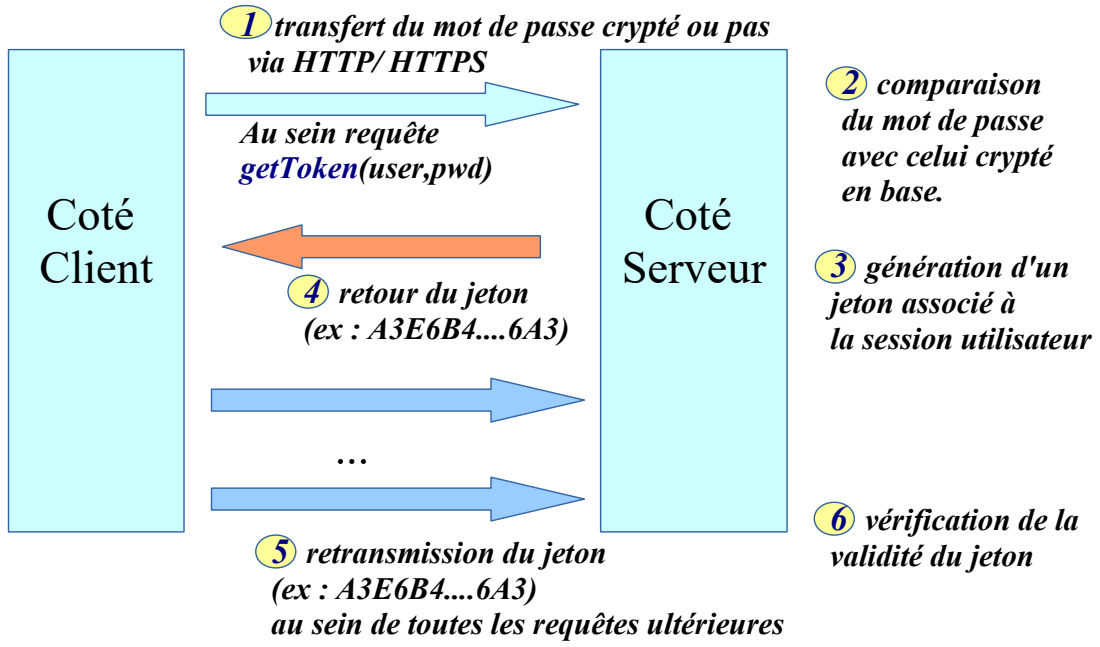
Une authentification sérieuse consiste à utiliser conjointement HTTPS , des jetons et un algorithme de cryptage pour stocker les mots de passe en base.

L'algorithme "bcrypt" est tout à fait approprié pour crypter les mots de passe à stocker en base.

"Bcrypt" génère un "salt"/"clef" aléatoirement en fonction de n=10 ou autre et le résultat du cryptage n'est pas constant.
Bien que "non constant" et "avec clef jetée" , l'algorithme "bcrypt" peut déterminer si un ancien cryptage bcrypt récupéré en base correspond ou pas à un des cryptages possibles du mot de passe reprécisé en clair .

```
// cryptage avant stockage :
var bcryptedPwd =
   bcrypt.hashSync(password, bcrypt.genSaltSync(12), null);

// comparaison lors d'une authentification :
if ( bcrypt.compareSync(password, bcryptedPwd) ) { …}   else { ...}
```

Jeton ("token") d'authentification valide le temps d'une session utilisateur



Plusieurs sortes de jetons/tokens

Il existe plusieurs sortes de jetons (normalisés ou pas).

Dans le cas le plus simple, un **jeton est généré aléatoirement** (ex : **uuid** ou …) et **sa validation consiste essentiellement à vérifier son existence** en tentant de le récupérer quelque part *(en mémoire ou en base)* et éventuellement à vérifier une date et heure d'expiration.

JWT (Json Web Token) est un **format particulier de jeton** qui **comporte 3 parties** (une entête technique , un paquet d'informations en clair (ex : username , email , expiration, …) au format JSON et une signature qui ne peut être vérifiée qu'avec la clef secrète de l'émetteur du jeton.

Bearer token (jeton au porteur) et transmission

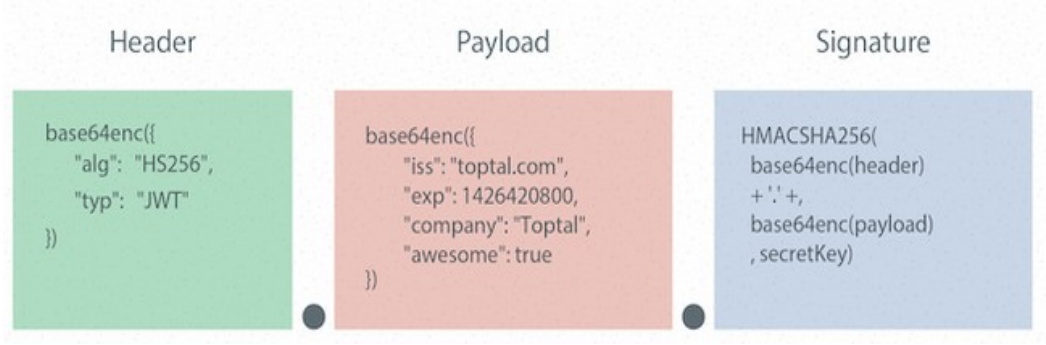
Le champ ***Authorization:*** normalisé d'une entête d'une requête HTTP peut comporter une valeur de type ***Basic …*** ou bien ***Bearer …***

Le terme anglais "**Bearer**" signifiant "au **porteur**" en français indique que la simple possession d'un jeton valide par une application cliente devrait normalement , après transmission HTTP, permettre au serveur d'autoriser le traitement d'une requête (après vérification de l'existence du jeton véhiculé parmi l'ensemble de ceux préalablement générés et pas encore expirés).

NB: Les "bearer token" sont utilisés par le protocole "O2Auth" mais peuvent également être utilisés de façon simple sans "O2Auth" dans le cadre d'une authentification "sans tierce partie" pour API REST.

NB2 : un "bearer token" peut éventuellement être au format "JWT" mais ne l'est pas toujours (voir rarement) en fonction du contexte.

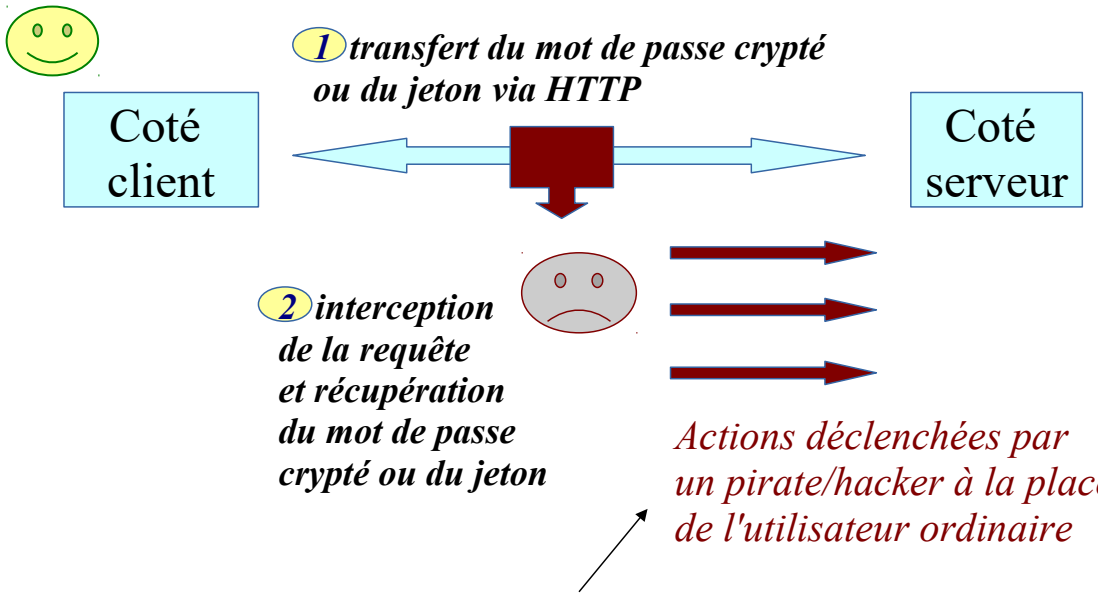
Structure jeton "JWT / Json Web Token"



Exemple:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK_ZXw

NB : "iss" signifie "issuer" (émetteur) , "iat" : issue at time
"exp" correspond à "date/heure expiration" . Le reste du "payload" est libre (au cas par cas) (ex : "company" et/ou "email" , ...)

Problématique "man in the middle"



Mauvaises actions possibles tant que reste valide l'information d'authentification interceptée (mot de passe crypté ou bien jeton). → Besoin de HTTPS et/ou HMAC

Signature des requêtes (avec clef secrète) et requête à usage unique (avec timestamp)

Pour éviter qu'une requête interceptée puisse conduire à une attaque de type "man in the middle" , on peut **ajouter une signature de requête rendant celle-ci inaltérable (non modifiable)** .

Dans le cas, où la requête serait interceptée , le "hacker" ne pourrait que la rejouer telle quelle (sans pouvoir la modifier).

Pour, tout de même **éviter, qu'une requête puisse être relancée plusieurs fois**, il suffit d'**ajouter un "timestamp" au message à envoyer**.

NB: Ces 2 techniques sont assez souvent utilisées ensembles et la technologie d'authentification associée s'appelle **HMAC** (keyed-hash message authentication code)

HMAC avec timeStamp (partie 1 / coté "client")

1) l'application cliente prépare la requête (userName ou … , timeStamp , paramètres , …) . Celle-ci peut prendre la forme d'une URL en mode GET ou bien être la base d'un calcul d'empreinte en mode POST .

2) l'application cliente élabore une signature de requête :
signature=Base64(HMAC-SHA1(UTF-8-Encoding-Of(*request*), *clef*))

Exemple (javascript) :

```
var user = "powerUser";        // Récupéré depuis la page d'authentification.
var password = "topSecret";    // Récupéré depuis la page d'authentification.
const salt = "13@!azerty"; // ou autre (selon application)
var encryptedPassword = CryptoJS.SHA1(CryptoJS.SHA1(password)+salt);
var httpVerb = "GET" ;
var currentTime = +new Date(); // valeur du timeStamp
var url = "http://www.xx.yy/product?user=" + user + "&timestamp=" + currentTime;
var httpUrl = httpVerb + ":" + url;
var signature =
      CryptoJS.HmacSHA1(httpUrl,encryptedPassword).toString(CryptoJS.enc.Base64);
url = url + "&signature=" + signature;   //à envoyer
```

HMAC avec timeStamp (partie 2 / coté "serveur")

1) l'application **serveur reçoit la requête** et en **extrait le username** (*en clair).*

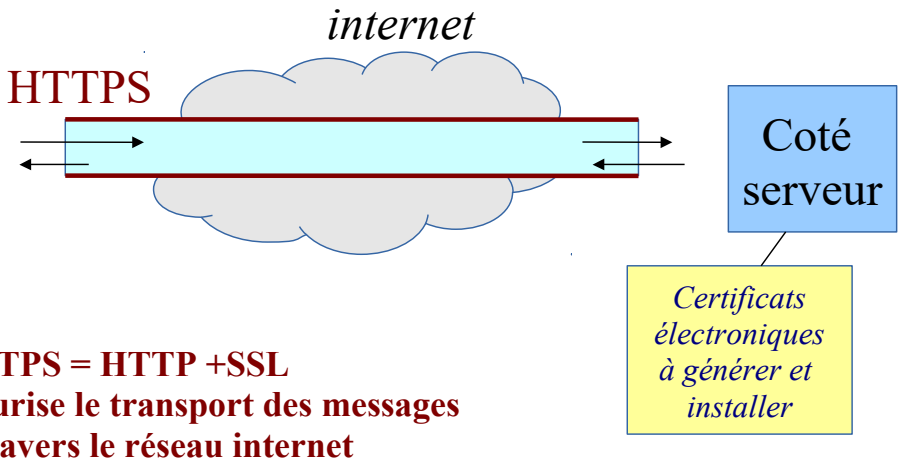
2) l'application **serveur récupère en base le mot de passe crypté** (haché , salé) de l'utilisateur et s'en sert pour **calculer une signature du message avec le même algorithme que du coté client** .

3) l'application **serveur compare les deux signatures** (reçue et re-calculée) **pour vérifier que le message n'a pas été intercepté/modifié/altéré** .

4) l'application serveur tente de récupérer en base le dernier "timeStamp" associé à l'utilisateur s'il existe (véhiculé par requête précédente).
Si le timeStamp qui accompagne la requête n'est pas inférieur ou égal au dernier "timeStamp" récupéré en base tout va bien (la requête n'a pas été lancée plusieurs fois) . Le timeStamp reçu est alors sauvegardé en base (pour le prochain test) , la requête est acceptée et traitée.

Conclusion : **HMAC avec timeStamp** garantit une **authentification robuste** mais ne gère pas la confidentialité des messages transmis, d'où l'éventuel besoin d'un complément HTTPS (HTTP + SSL) .

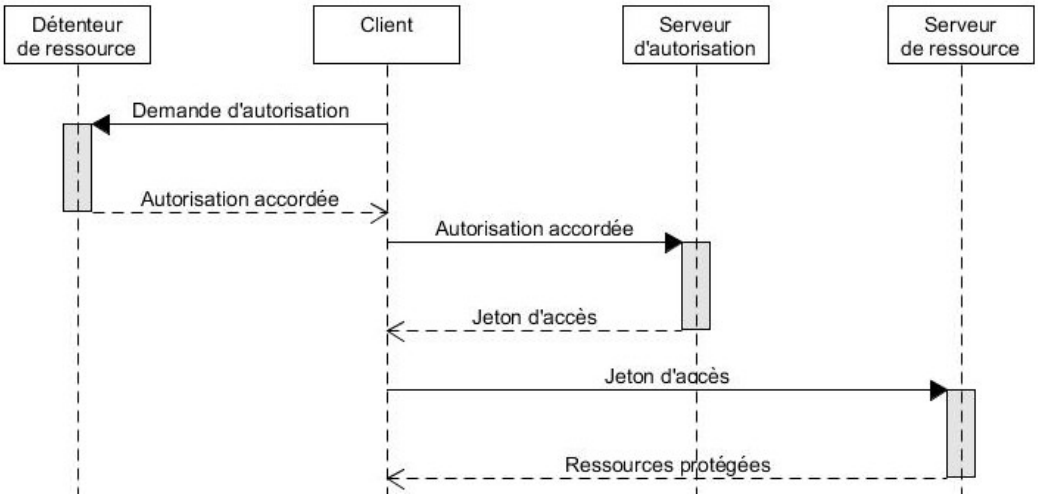
(HMAC avec timeStamp) ou ... + HTTPS



HTTPS = HTTP +SSL
sécurise le transport des messages à travers le réseau internet
(confidentialité , protection contre interception/modification, ...) mais ne gère pas (tout seul) l'authentification.

Norme/Protocole "OAuth2"

OAuth (**O**pen **Authorization**) existant en versions "1" et "2" , est une norme (RFC 6749 et 6750) qui correspond à un **protocole de "délégation d'autorisation" .** Ceci permet par exemple d'autoriser une application cliente à accéder à une API d'une autre application (ex : FaceBook , Twitter , …) de façon à accéder à des données protégées.



En résumé (pour une bonne sécurité)

Une sécurisation sérieuse d'une Api REST s'effectue quasi-systématiquement en mode **HTTPS / SSL**.
Dans ce cadre, où tous les échanges sont cryptés, on peut envisager une transmission simple du mot de passe pour obtenir en échange un **jeton** (*JWT ou pas*) à retransmettre via le champ ***"Authorization : Bearer …"*** de l'entête HTTPS d'une requête.

Certains cas pointus ou spécifiques pourront éventuellement être adressés en utilisant ***HMAC*** ou ***OAuth2*** .

Dans le cas où HTTPS/SSL n'est pas utilisé , un mot de passe doit absolument être sérieusement crypté "***hash+salt***" avant d'être transmis via le dangereux protocole HTTP .

D'autres spécificités peuvent éventuellement être ajoutées (ex: **challenge** avec une matrice de "0" , "1" , … "9" pour éviter une saisie directe d'un mot de passe numérique) .

XIV - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP