nodeJs , npm , express, ...

Table des matières

I - NodeJs : utilisations, principes.....4
1. Principaux cas d'utilisation de nodeJs.....4
2. Vue d'ensemble sur le fonctionnement de nodeJs.....7
3. programmation asynchrone (nodeJs).....14

II - Vue d'ensemble (node , npm , express, ...).....15
1. Ecosystème node+npm.....15
2. Express.....15
3. Exemple élémentaire "node+express".....16

III - Node et npm : installation et utilisation.....18
1. Installation de node et npm.....18
2. Configuration et utilisation de npm.....19
3. Utilisation basique de node.....21

IV - Modules - nodeJs.....22
1. Modules dans environnement "nodeJs".....22
2. Modules "cjs/commonjs" (exports , require).....23
3. Modules "es2015" / "typescript" / env. "nodeJs".....24

V - Express (essentiel).....27
1. Essentiel de Express.....27

VI - Web services REST avec express.....35
1. WS REST élémentaire avec node+express.....35
2. Avec mode post et authentification minimaliste.....37
3. Autorisations "CORS".....39
4. Divers éléments structurants.....40

VII - Accès aux bases de données (node).....45
1. Accès à MySQL via mysqljs/mysql (node).....45
2. Accès à MongoDB (No-SQL , JSON) via node.....50

VIII - Annexe – Tests (mocha , chai , ...).....55
1. Tests avec Mocha + Chai (env. nodeJs).....55

IX - Annexe – ORM Sequelize.....65
1. ORM Sequelize (node).....65
2. Utilisation de Sequelize v5.x avec Typescript.....76

X - Annexe – apidoc (pour api rest).....83
1. Api doc.....83

XI - Annexe – Utilitaires (grunt , gulp , ...).....87
1. GRUNT.....87
2. GULP.....89

XII - Annexe – WEB Services "REST".....93
1. Généralités sur Web-Services REST.....93
2. Limitations Ajax sans CORS.....100
3. CORS (Cross Origin Resource Sharing).....101

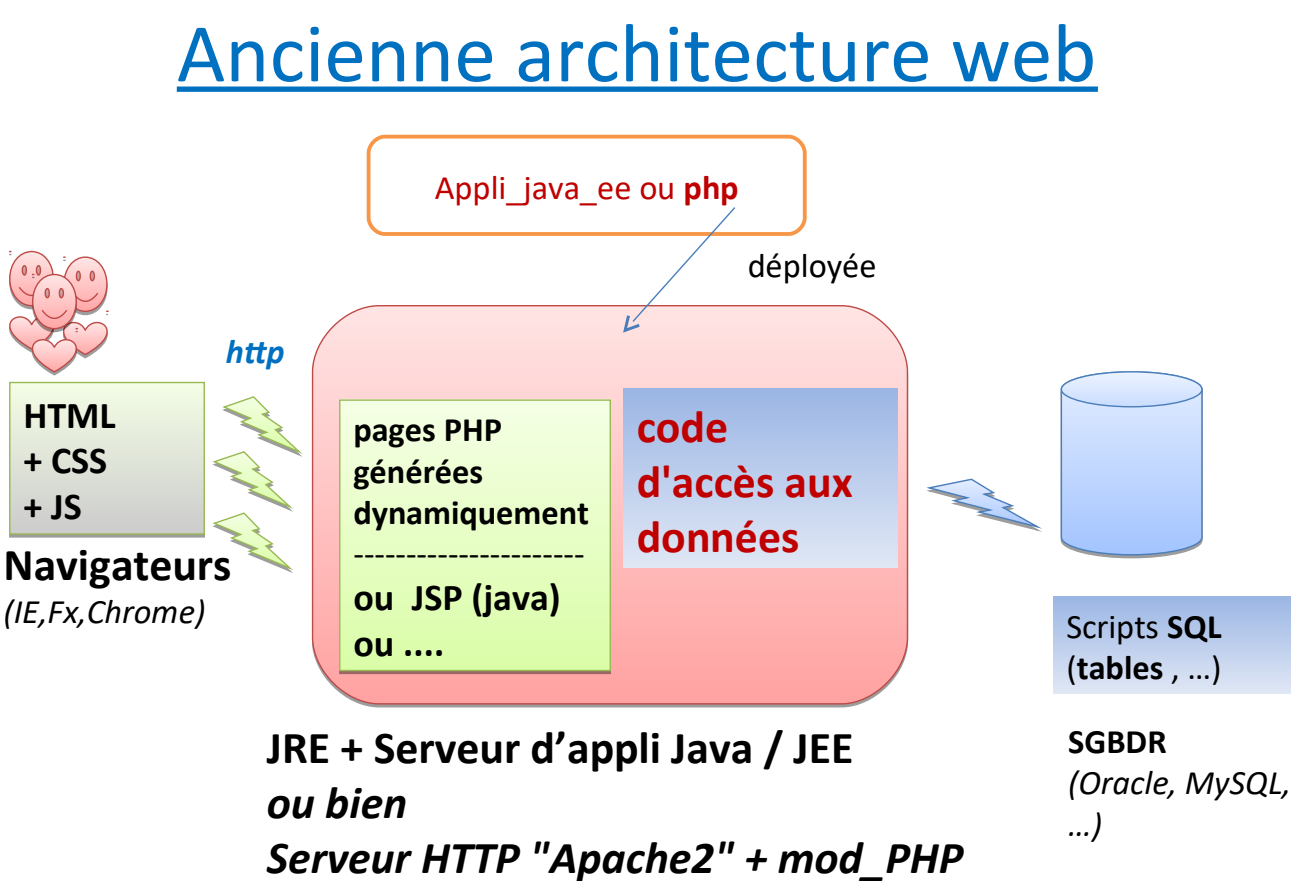
XIII - Annexe – Sécurité - WEB Services "REST".....104
1. Api Key.....104
2. Token d'authentification.....105

XIV - Annexe – Bibliographie, Liens WEB + TP.....109
1. Bibliographie et liens vers sites "internet".....109
2. TP.....109

I - NodeJs : utilisations, principes

1. Principaux cas d'utilisation de nodeJs

1.1. Architecture logicielle alternative ("MEAN" ou ...) à "LAMP"



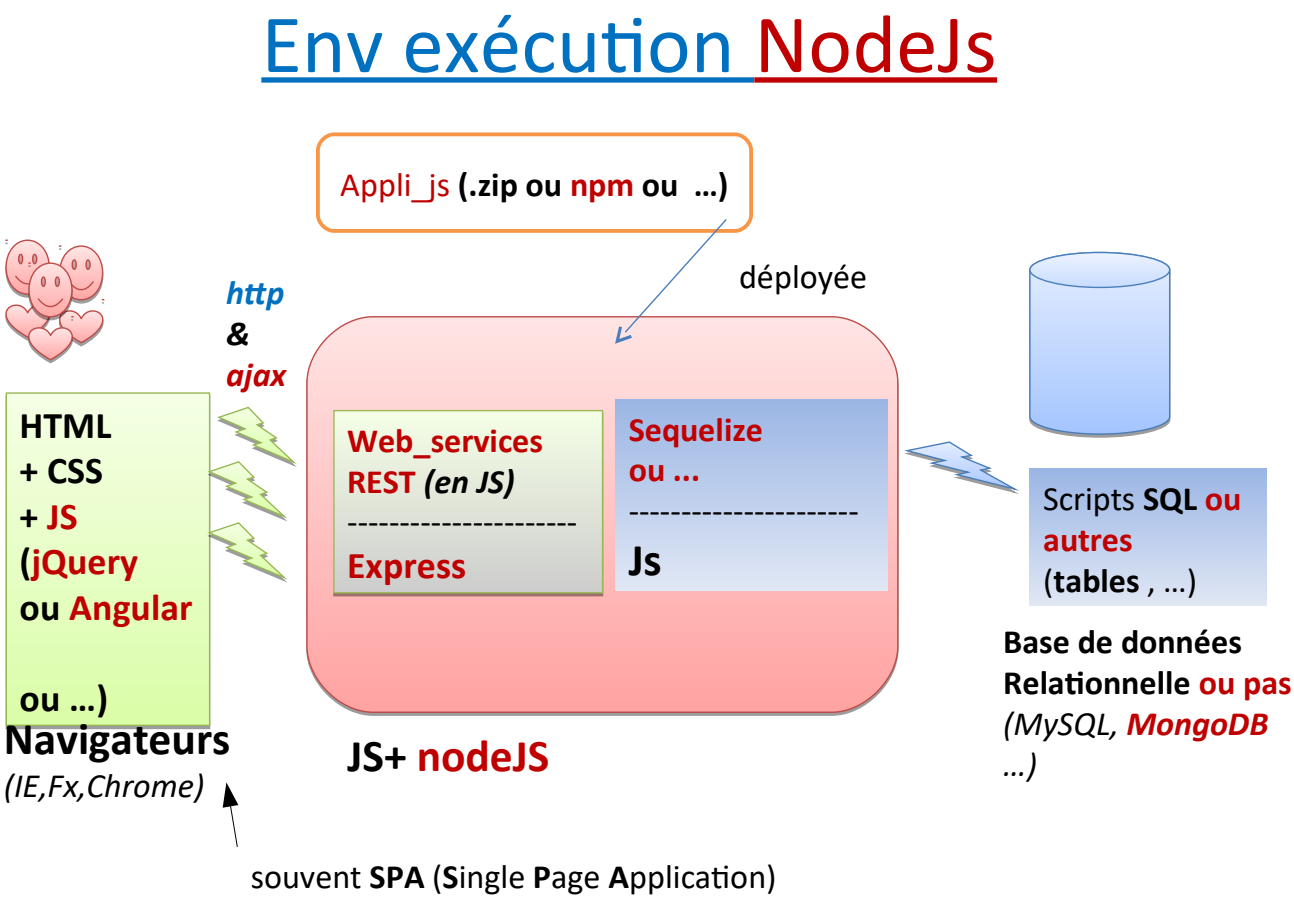
Dans ancienne architecture :

- presque tout coté serveur , le navigateur interprète et affiche des pages entièrement générées (de manière dynamique) coté serveur en fonction des valeurs en base de données
- le navigateur utilise "javascript" essentiellement pour dynamiser les pages (effets , ...)
- coté serveur : beaucoup de multi-threading (ex : Apache 2 ou JEE/Java)
- exemple classique : architecture **LAMP** = **L**inux / **A**pache / **M**ysql / **PHP**

Principaux défauts :

- problème de performance lorsque très nombreux utilisateurs simultanés
- IHM moyennement réactive
- dispersion des technologies "IHM/WEB/serveur" :PHP , JSP , ...(au cas par cas)

1.2. Plate-forme d'exécution des api REST nécessaires en arrière plan des applications "Single Page"



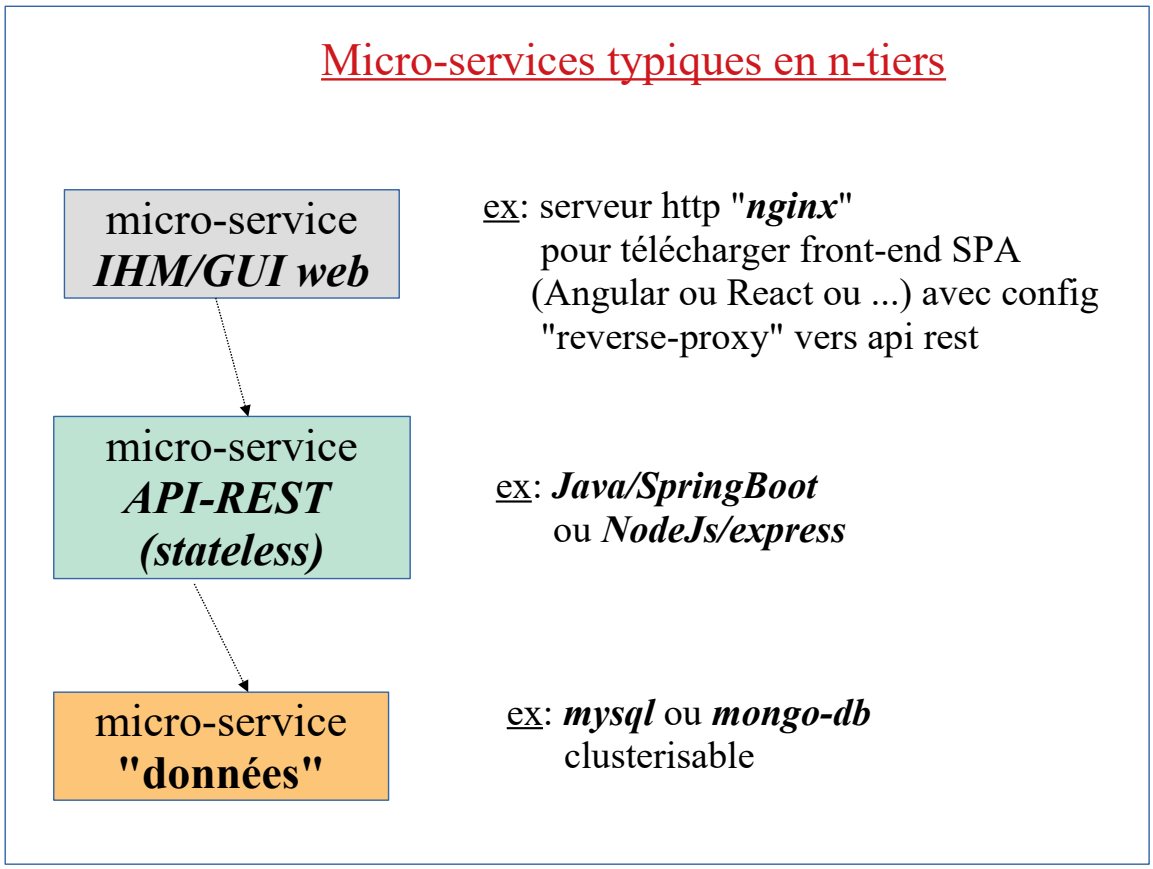
Dans nouvelle architecture :

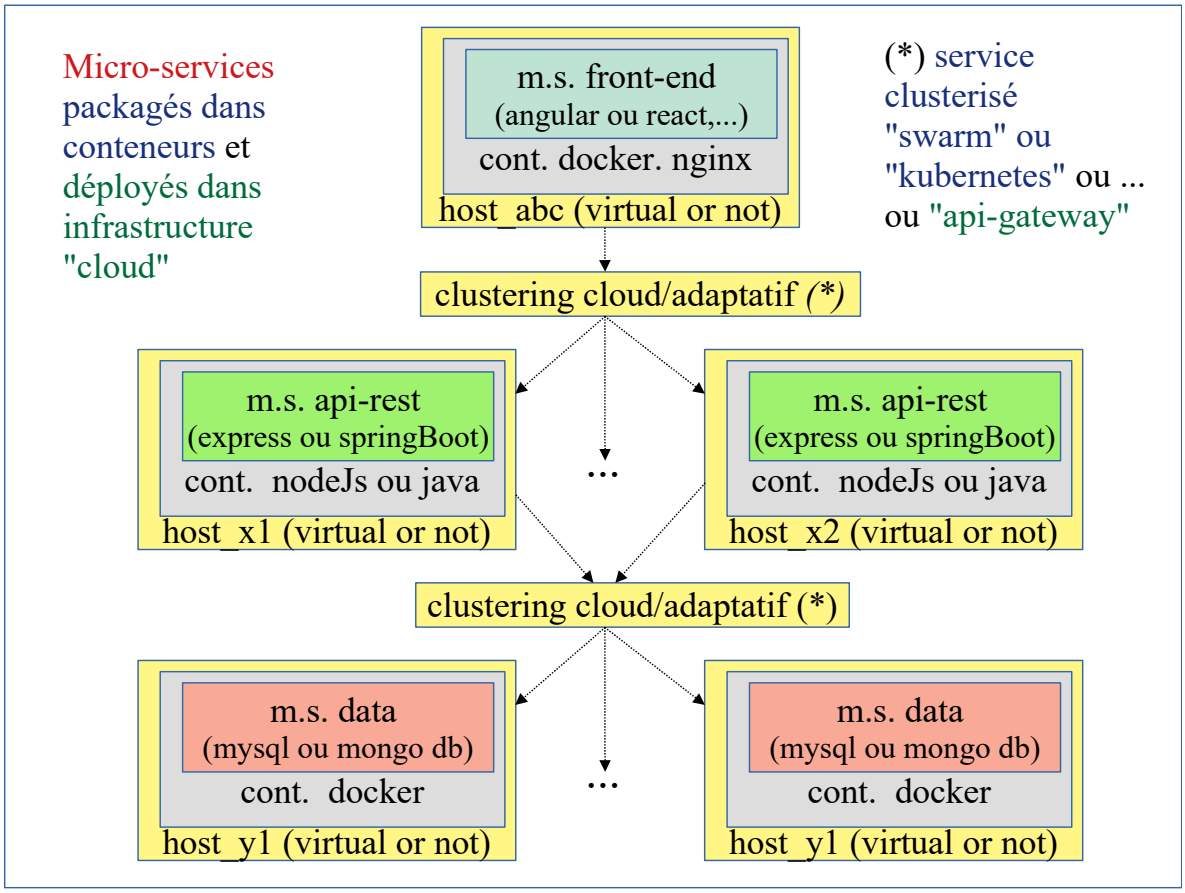
- le coté serveur est moins sollicité : juste besoin gérer (retourner ou accepter) des données JSON à travers un paquet de WebServices "REST" , plus besoin de générer des pages HTML complètes, plus besoin de gérer des "sessions utilisateurs coté serveur" , plus besoin de gérer la navigation entre les pages
- le navigateur utilise beaucoup plus "javascript" (appels HTTP/ajax + Api DOM , ...)
- coté serveur : beaucoup moins de thread mais api asynchrones (ex : Node / express, ...)
- exemple classique : architecture **MEAN** = **M**ongo / **E**xpress / **A**ngular / **N**ode

Principaux avantages:

- meilleurs performances lorsque très nombreux utilisateurs simultanés
- IHM plus réactive (meilleurs temps de réaction)
- meilleur séparation "frontEnd (HTML/CSS/JS)" et "backend (Api_REST + accès DB)

1.3. Bonne solution pour mettre en oeuvre un micro service





2. Vue d'ensemble sur le fonctionnement de nodeJs

2.1. Présentation de nodeJs

node Js

node (nodeJs) est un environnement d'exécution javascript basé sur un moteur V8 (*"machine virtuelle" ou "moteur d'interprétation" javascript*).

Très souvent utilisé coté serveur , cet environnement d'exécution est basé sur une logique d'exécution asynchrone très légère et très efficace.

Pouvant charger du code modulaire compartimenté en modules téléchargeables (ex : "*express*") , nodeJs est essentiellement utilisé pour implémenter des *web services REST* (invoqués via HTTP).

2.2. Historique de nodeJs

Historique de nodeJs

NodeJs a été créé par **Ryan Dahl** en 2009. Son développement et sa maintenance sont effectués par l'entreprise Joyent .

Dès **2009** : NodeJs combine le moteur javascript V8 de Google, une boucle événementielle et une api I/O asynchrone .

En **2010** , un gestionnaire de modules téléchargeable "**npm**" est intégré à nodeJs (publish, install, ...) ce qui permet de partager le code efficacement.

De 2011 à 2015 : plein de fork, de join , ...
et parallèlement un très grand nombre de modules mis au point ou améliorés (express, ...)

Aujourd'hui : site officiel de nodeJs : **https://nodejs.org**

2.3. Principes de fonctionnement de nodeJs

REPL (Read–Eval–Print Loop)

Un langage interprété (ex : "lisp" , "basic" ou "javascript") peut (à un haut niveau , quelque fois proche d'une interaction utilisateur) effectuer une boucle dite "REPL" de ce type :

loop({
- read expression *(ex : **read** "2*4+5" in variable **expr**)*
- eval result *(ex : res=**eval**(expr) = 13)*
- print result *(ex : console.log("res="+res) ;)*
})

vision fonctionnelle :

```
(loop (print (eval (read()))))
```

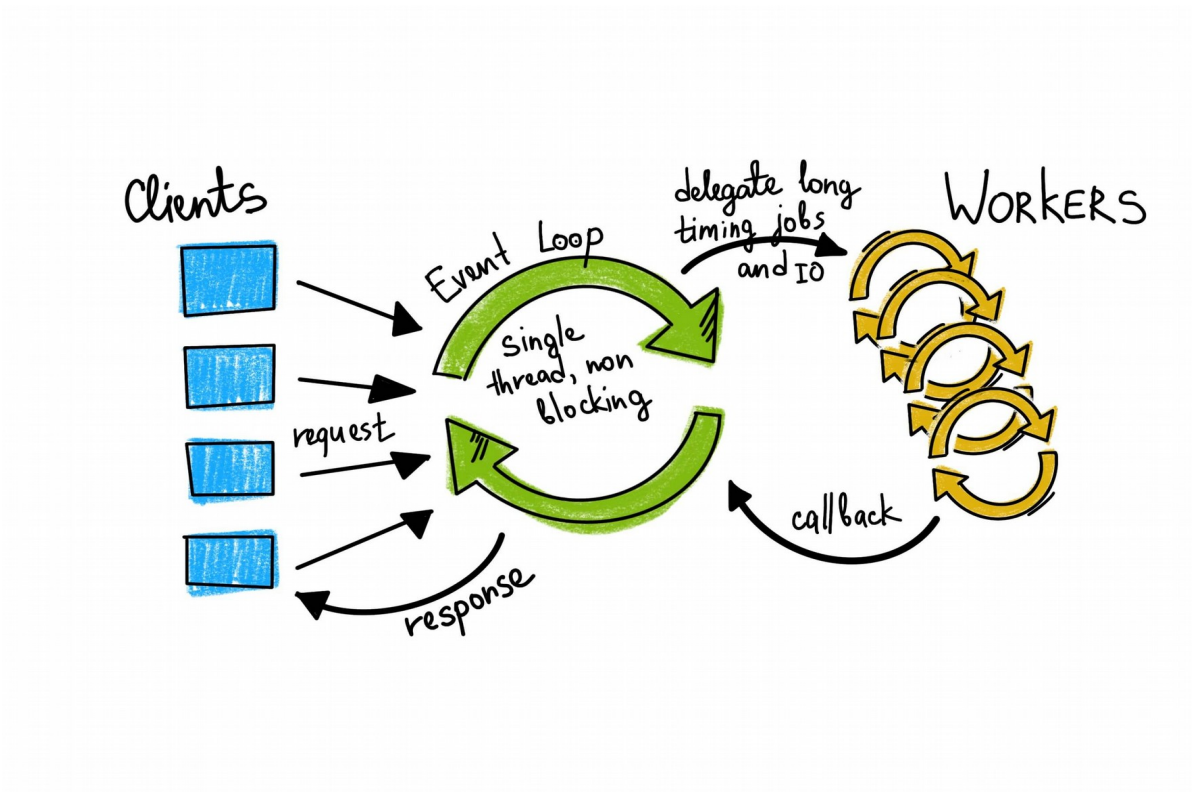
```
expression: 2+3
result=5
expression: 4+4
result=8
expression: 2*6+1
result=13
expression: fin
end
```

NB : l'instruction **eval**() du langage javascript permet de déclencher l'interprétation (et exécution) immédiate d'expression javascript récupérée sous forme de simple chaîne de caractères .
(exemple : eval("2*Math.PI*r"))

NodeJs s'est en partie inspiré de REPL mais a dû mettre en place une adaptation sophistiquée tenant compte des éléments suivants :

- pas de simple interactions "utilisateur" en mode texte
  mais interactions "web" (html/js/http/...) vues comme des événements utilisateurs à gérer.
- certaines tâches à déclencher son potentiellement longues et doivent être effectuées en mode asynchrone non bloquant

Single Threaded Event Loop Model

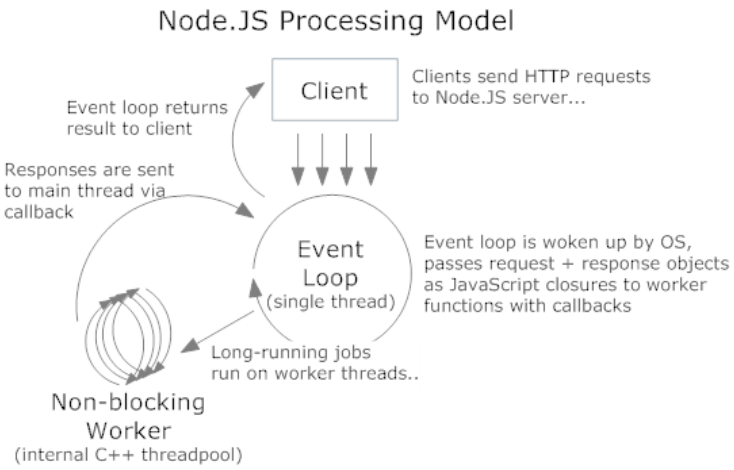


Les interactions "utilisateurs/clients" sont concrètement des appels http/ajax allant d'un navigateur web vers le serveur nodeJs.

NodeJs traite toutes les requêtes entrantes avec un seul grand thread principal qui :

- boucle sans arrêt sur :
  - réceptionner requêtes (vues comme des événements)
  - évaluer des réponses/résultats
  - renvoyer les réponses via http (vers navigateur ou application cliente)
- n'effectue que des actions non bloquantes (pour ne pas bloquer les autres utilisateurs/clients)

NB :
Certaines opérations longues sont déléguées à des "workers / thread secondaires" en tâche de fond qui vont à leur tour essayer d'optimiser les I/O en mode asynchrone non bloquant lorsque c'est possible (ex : accès non bloquant au système de fichiers , communications réseaux via des mécanismes "non blocking sockets" , ...)



Speudo Code of Main EventLoop of nodeJs :

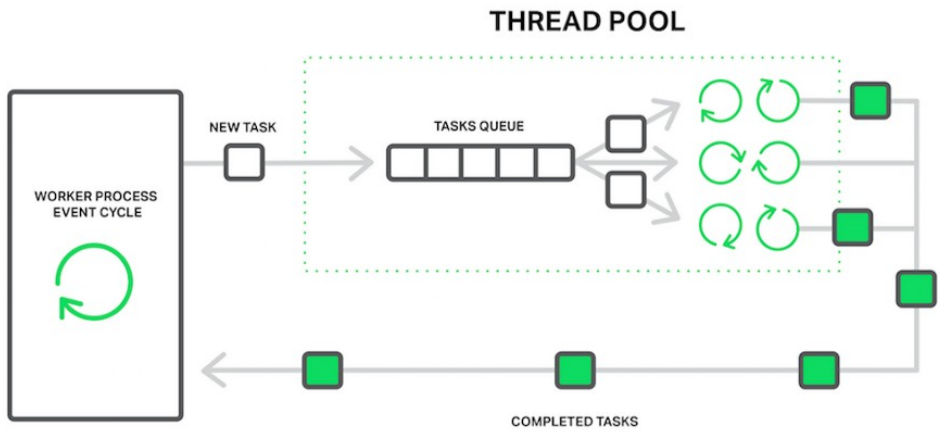
```
while(true){
    if(Event Queue receives a JavaScript Function Call){
        request = EventQueue.getClientRequest();
        if(request requires BlokingIO or
            takes more computation time)
            Assign request to main back Thread "T1"
        else
            Process and Prepare response immediatly
        }
    }
```

NB :

- Lorsque les traitements long effectués par les "workers / threads secondaires" en arrière plan sont terminés , les résultats sont récupérés du coté "single thread main javascript loop" via des fonctions "callbacks" dont les arguments sont les résultats (positifs ou négatifs)
- la majorité des résultats/réponses sont positifs (exemple : données recherchées , ...) et certaines réponses sont négatives (ex : ressources (fichiers/urls) inaccessibles , timeout , ...)

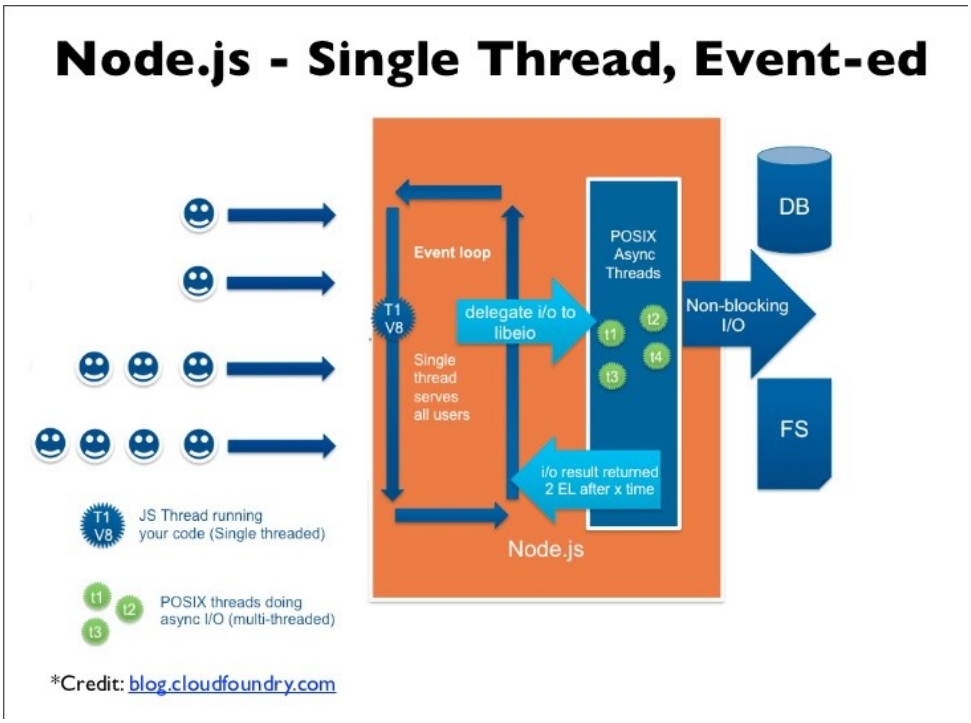
==> autrement dit , beaucoup de code est souvent nécessaire pour tenir compte de l'issue d'un traitement long déclenché en différé . Heureusement qu'il existe try/catch !

Exemple d'exécution asynchrone en c/c++ : nginx

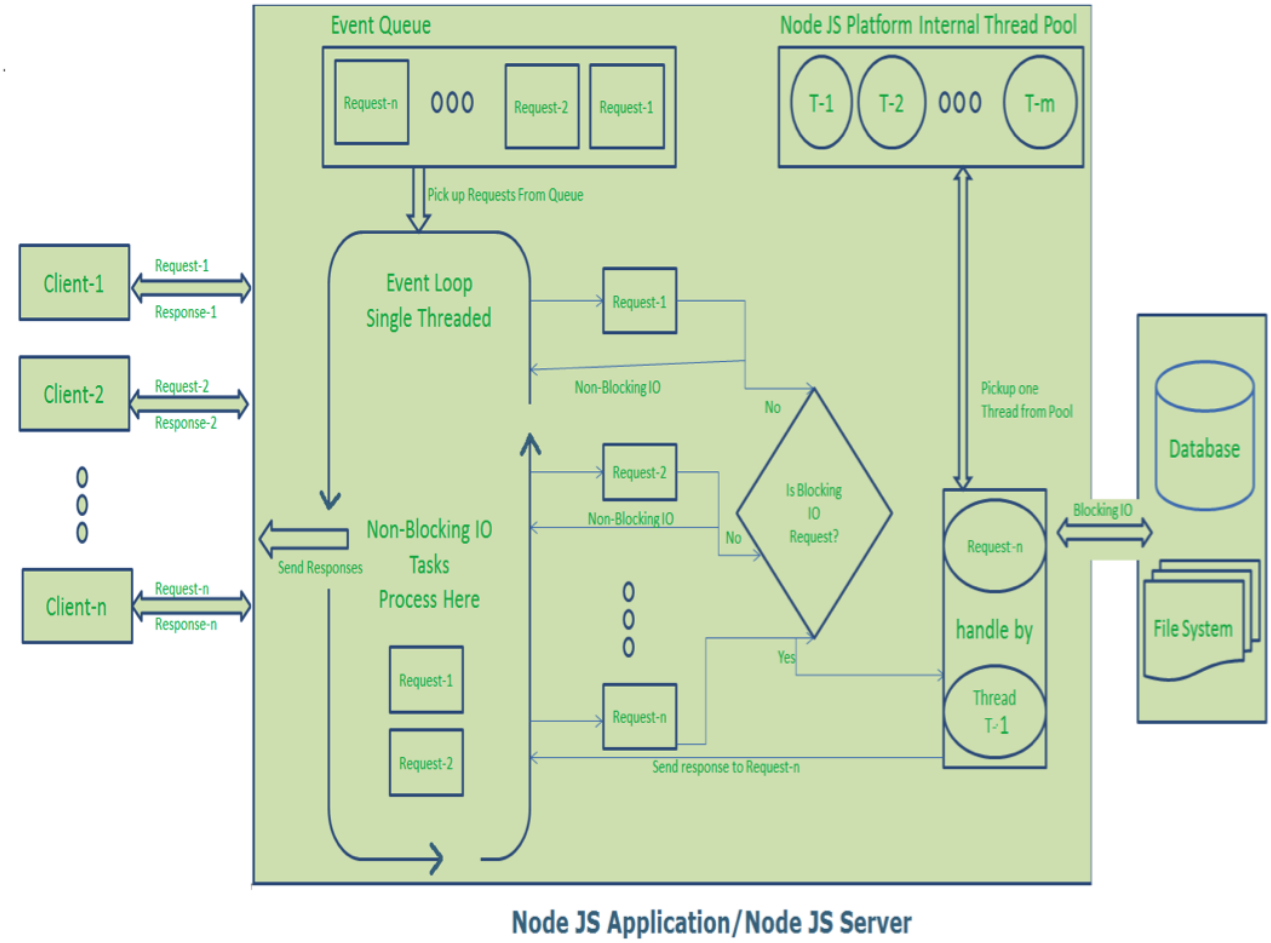


Dans nodeJs (en arrière plan) :
Threads asynchrones "POSIX" (avec un maximum de "non blocking I/O").

Architecture interne de nodeJs



Vue d'ensemble sur les mécanismes internes de nodeJs :



3. programmation asynchrone (nodeJs)

3.1. pas séquentiel , pas appels bloquants

Une application "nodeJs" se programme d'une manière très différente d'une application PHP ou Java/JEE :

- Pas d'appel bloquant mais toutes les opérations potentiellement longues sont déclenchées de manières asynchrones avec des fonctions "callbacks" à coder pour récupérer les résultats en différé.
  Principaux appels non bloquant et asynchrones : "requêtes SQL".

- Code très rarement séquentiel/linéaire mais plutôt événementiel (avec plein de points d'entrées) : dès qu'un événement survient , on déclenche un bloc de code pour le gérer.
  Les actions asynchrones alors déclenchées sont alors à l'origine d'autres événements et ainsi de suite ...

3.2. callbacks (avec ou sans Promise , async-await)

Les traitements liés au fonctions "callbacks" doivent quelquefois être synchronisés/ordonnés pour gérer une certaine logique métier (règle de gestion, algorithme, ...).
Sans astuce , ceci peut devenir une prise de tête souvent appelé "l'enfer des callbacks" .

Heureusement, les "Promise" (es2015) et les "async-await" permettent de :
- rendre le code plus fiable et plus lisibles
- simplifier la syntaxe
- retarder l'entrée à l'asile des développeurs

NB : la compréhension de "async/await" passe par une étude préalable des "Promise" .

II - 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) .

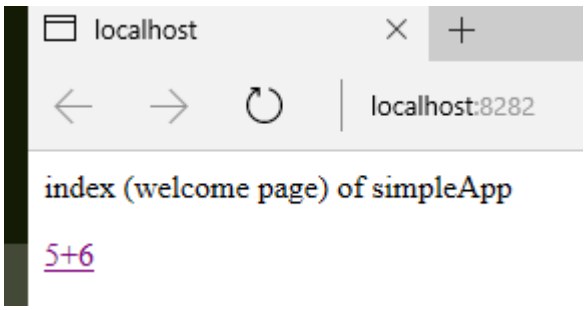
3. Exemple élémentaire "node+express"

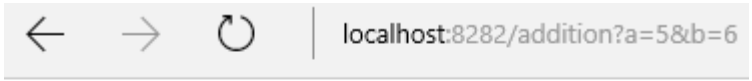
first_express_server.js

```
//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 :





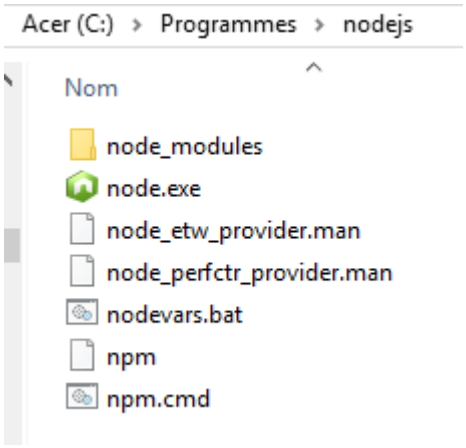
a=5
b=6
a+b=11

III - Node et npm : installation et utilisation

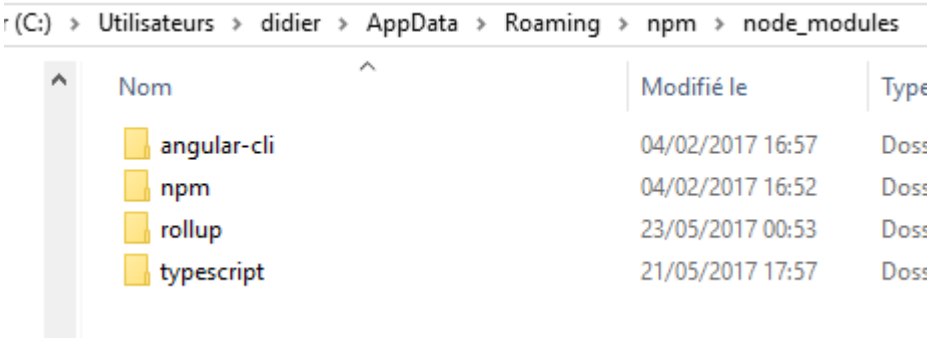
1. Installation de node et npm

Téléchargement de l'installeur **node-v10.15.3-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
v10.15.3 (ou autre)

npm --version
6.4.1 (ou autre)

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": "^4.17.0",
   "mongoose": "^4.11.0"
 },
….
```

Sans l'option **--save** (ou son alias **-s**) , 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
ou bien (autre exemple) :

npm install --save mongodb@2.0.55

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***

IV - Modules - nodeJs

1. Modules dans environnement "nodeJs"

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

Grands principes :
- chaque module correspond à un fichier séparé
- **seuls les éléments exportés par un module pourront être vus par les autres**
- un module doit commencer par importer certains éléments d'un autre module avant de pouvoir les utiliser

Intérêts de la programmation à base de modules :
- moins d'effets de bords (moins de conflits de noms de variables , types , ...)
- écosystème à la fois simple/efficace et très extensible (sans forcer une complexité inutile : on ne charge que les modules nécessaires)
- chargement dynamique (souple et performant)
- écosystème ouvert (à des évolutions , de nouveaux modules concurrents, ...)
- ...

Il existe cependant plusieurs technologies de modules dans le monde javascript :
- L'écosystème nodeJs utilise depuis longtemps en interne des modules au format "**cjs/commonjs**" avec une syntaxe "***exports... = ...***" et "***var mxy = require('xy')***"
- Depuis la version normalisée "es6/es2015" de javascript/ecmaScript , certains modules peuvent être codés en s'appuyant sur la syntaxe des **modules es2015** (***export*** ... , ***import { ...} from '..'***)

Modules supportés par la version 10 de nodeJs :
- Enormément de modules sont aujourd'hui codés en javascript et basés sur la technologie historique "cjs/commonjs" .
- Seules les dernières versions de nodeJs commencent à pouvoir directement interpréter des modules "es2015" (via l'option ***-r esm*** de **node**) et avec l'installation du module esm (npm install esm).

Pour ne pas apporter de confusion entre les deux technologies de modules de l'écosystème nodeJs :
- les modules au format traditionnel "commonjs" sont des fichiers avec l'extension **".js"**
- les modules au format récent "es2015" sont des fichiers avec l'extension **".mjs"** (très rares)

Cadre classique (selon le langage choisi) :

En 2018,2019 , la plupart des projets sont basés sur l'une des 2 stratégies suivantes :
- programmation directe en javascript es6/es2015 (avec la syntaxe require de commonjs)
- programmation d'un **code source en typescript (sur ensemble de es6/es2015) avec des instructions "import { ...} from '...' "** transformé/transpilé en code javascript et modules commonjs via l'option ***"module": "commonjs"*** de **tsconfig.json** .

2. Modules "cjs/commonjs" (exports , require)

2.1. 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 !!!***

2.2. 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
<p><strong>5 + 6 = 11</strong></p>
```

3. Modules "es2015" / "typescript" / env. "nodeJs"

code_source.ts ----> transformation/transpilation **tsc** -------> code javascript à exécuter via node
import {...} from '...' **var ... = require('...')**

tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2017",          /* si nodeJs récent , sinon target "es2015" ou "es5" */
   "module": "commonjs",
   "lib": ["es2015"],   /* ou autre  */
...
}
```

3.1. Exportation des éléments en typescript

math-util.ts

```
  export function additionner(x : number  , y : number) {
    return x + y;
  }
  export function multiplier(x : number  , y : number) {
    return x * y;
  }
```

ou bien

```
function additionner(x : number  , y : number) {
    return x + y;
  }
function mult(x : number  , y : number) {
    return x * y;
  }
export { additionner, mult as multiplier };
```

On peut également exporter de la même façon des classes , des variables , ...

3.2. Importation des éléments en typescript

main.ts

```
import { additionner as add, multiplier } from "./math-util";


function carre(x){
  return multiplier(x,x) ;
}
/*
var msg1 = "Le carre de 5 est " + carre(5);   console.log(msg1);
var msg2 = "4 * 3 vaut " + multiplier(4, 3);  console.log(msg2);
var msg3 = "5 + 6 vaut " + add(5, 6); console.log(msg3);
*/
```

Variantes d'importation avec éventuels préfixes :

main.ts

```
import { f1 , f2 , f3 , f4  } from "./modxy";
let f_msg=f1('abc')+'-'+f2('abc')+'-'+f3('abc')+'-'+f4('abc');
```

ou bien

```
import * as f from "./modxy";
let f_msg=f.f1('abc')+'-'+f.f2('abc')+'-'+f.f3('abc')+'-'+f.f4('abc');
```

3.3. default export (one per module)

xy.ts

```
export function mult(x:number, y:number) {
    return x * y;
}

//export default function_or_object_or_class (ONE PER MODULE)
export default {
        name : "xy",
        features : { x : 1 , y: 3 }
}
```

main.ts

```
import xy , { mult }  from "./xy";
...
let msg = xy.name + "--" + JSON.stringify(xy.features) + "--" + mult(3,4);
```

3.4. importation de modules chargés dans node_modules via npm

```
import * as http from 'http';
import * as bodyParser from 'body-parser';
import express , { Request, Response } from 'express';
...
```

NB : **from 'http'** (et pas ~~from './http'~~)

V - Express (essentiel)

1. Essentiel de Express

1.1. Http sans express

Une application web basique basée que sur le coeur de nodeJs peut s'appuyer sur le module fondamental "**http**" *(toujours disponible , sans besoin de npm install).*

basic_http_server.js

```
var http = require('http');
//import * as http from 'http'; //es2015 , typescript

var myHttpFunction = function(req , res ) {
  res.writeHead(200 , {"Content-Type": "text/html"}); //OK=200
  res.write("<html> <body> <b> hello world </b> </body></html>")
  res.end();
};

var server = http.createServer(myHttpFunction);
console.log("http://localhost:8282")
server.listen(8282);
```

1.2. sur la route vers express

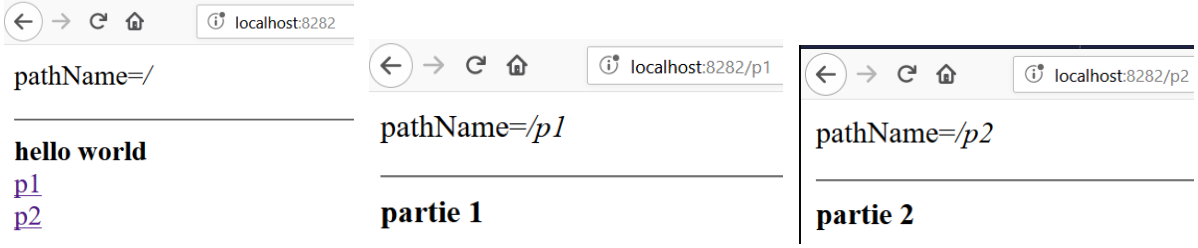
```
var http = require('http');
var url = require('url');
//import * as http from 'http'; import * as url from 'url';

var myHttpFunction = function(req : any, res : any) {
  res.writeHead(200 , {"Content-Type": "text/html"}); //OK=200
  res.write("<html> <body>");
  var pathName= url.parse(req.url).pathname; // "/" ou "/p1" ou "/p2"
  res.write("<p>pathName=<i>"+pathName+"</i></p><hr/>");
  switch(pathName){
    case '/p1' :
       res.write("<b> partie 1 </b>");    break;
    case '/p2':
       res.write("<b> partie 2 </b>");    break;
    case "/":
    default:
      res.write("<b> hello world </b><br/>");
      res.write("<a href='p1'>p1</a><br/>");
      res.write("<a href='p2'>p2</a><br/>");
  }
  res.write("</body></html>");
  res.end();
};
var server = http.createServer(myHttpFunction);  console.log("http://localhost:8282")
```

```
server.listen(8282);
```

Dans l'exemple de code précédent , l'instruction

```
var pathName= url.parse(req.url).pathname;
```

permet de récupérer dans la variable "pathName" la fin de l'url de la requête.
On peut ainsi différencier la réponse associée via un switch/case :



Pour avoir la même fonctionnalité (en beaucoup plus sophistiqué) sur un vrai projet , on utilise systématiquement le module/framework "**express**" permettant de différencier les réponses en fonctions de la fin de l'url de la requête entrante ("route vers un morceaux de code ou un autre").

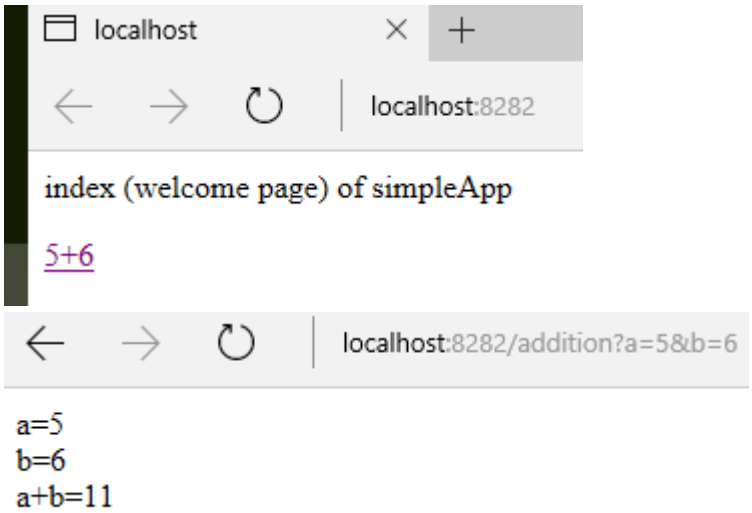
1.3. Utilisation élémentaire de express (avec routes simples)

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 :



Adaptations pour le langage **typescript** :

first_express_server**.ts**

```
import express , { Request, Response } from 'express';

var app :express.Application = express();

app.get('/', function(req :Request, res :Response ) {
...
}
...
```

1.4. Gestion par express d'une partie "statique"

server.js/ts

```
...
//les routes en /html/... seront gérées par express
//par de simples renvois des fichiers statiques du répertoire "./html"
app.use('/html', express.static(__dirname+"/html"));
...
```

Ceci permet à express de retourner directement fichiers nécessitants aucune interprétation coté serveur (ex: index.html , images , css , ...)

éventuelle route de redirection vers une page statique :

```
app.get('/', function(req , res  ) {
  res.redirect('/html/index.html');
});
```

Via cette route , en se connectant à *http://localhost:8282/* on se retrouve automatiquement redirigé vers la page statique *http://localhost:8282/html/index.html* (ex: SPA angular ou ...) .

NB : une page "statique coté serveur" pourra (lorsqu'elle fonctionnera (une fois téléchargée) au niveau du navigateur) appeler des web services REST de l'application courante "Node+express".

1.5. Expression des routes "express" :

app.**get**() , app.**post**() , app.**put**() , app.**delete**() , ... selon la méthode HTTP (GET , POST , PUT , DELETE) de la requête entrante .

1.6. Récupération des paramètres HTTP :

Pour récupérer les valeurs des paramètres véhiculés en fin d'URL en mode GET
(ex : .../*addition*?**a=5&b=6**) , la syntaxe à employer est

```
var a = req.query.a ;
var b = req.query.b ;
```

La récupération des valeurs des paramètres HTTP véhiculés en mode **POST** dans le corps (***body***) de la requête entrante s'effectue avec la syntaxe **req.body.*paramName*** et nécessite la préparation et l'enregistrement d'un "***bodyParser***" :

```
...
var bodyParser = require('body-parser');
//import * as bodyParser from 'body-parser';
...
//support parsing of application/x-www-form-urlencoded post data
app.use(bodyParser.urlencoded({ extended: true }));
...
//POST addition with body containing a=5&b=6 (application/x-www-form-urlencoded)
app.post('/addition', function(req : Request, res : Response) {
  let va = Number(req.body.a);
  let vb = Number(req.body.b);
  ...
});
```

1.7. Utilisation d'un moteur de template (ex : EJS, handlebars)

Générer directement coté serveur des pages html à coup de ***res.write(" ... ")*** c'est fastidieux , peu lisible, peu maintenable et pas productif sur un projet sérieux.

Pour générer plus simplement des pages HTML coté serveur on peut :

- utiliser le design pattern "**MVC** (Model Vue Controller)"
- utiliser un "***moteur de templates***" (ex : **Jade**, **EJS** , **Handlebars**, ...)

Principes de fonctionnement :

1. la requête HTTP est d'abord gérée par une route express ("contrôleur") permettant d'analyser les paramètres et d'effectuer (par délégation) des traitements appropriés
2. Le code final de la route express prépare un objet javascript ("modèle") permettant de passer des valeurs de la réponse à retourner vers le "template" de page HTML.
3. Durant la phase de rendu, le moteur de template effectue un remplacement des zones {{a}} ... {{b}} ... par les valeurs correspondantes de l'objet "modèle/js" passé en argument de la fonction **res.render**('view_name' , { a : ... , b : ... }) .

1.8. Exemple "Express + handlebars"

package.json

```
...
"dependencies": {
   "@types/express": "^4.16.1",
   "@types/express-handlebars": "0.0.32",
   "express": "^4.17.0",
   "express-handlebars": "^3.1.0"
 }
...
```

(à installer via "***npm install -s ...***") .

Structure appropriée des répertoires :

```
├── server.js
└── views
    ├── server-home.handlebars
    │   calcul.handlebars
    │   addition.handlebars
    └── layouts
        └── main.handlebars
```

server.js/ts

```
var express = require('express');
var exphbs  = require('express-handlebars')

//import exphbs  from 'express-handlebars';


var app /* :express.Application */ = express();
app.engine('handlebars', exphbs());
app.set('view engine', 'handlebars');

app.get('/', function(req , res  ) {
  res.redirect('/server-home');
});

app.get('/server-home', function(req , res  ) {
    res.render('server-home');//rendering views/server-home.handlebars in the context of
                              //views/layouts/main.handlebars
});

app.get('/calcul', function(req , res  ) {
   res.render('calcul'); //views/calcul.handlebars
});

//GET addition?a=5&b=6
app.get('/addition', function(req , res ) {
  let va = Number(req.query.a);
  let vb = Number(req.query.b);
  let vaPlusVb = va+vb;
  res.render('addResult', { a : va ,
                            b: vb ,
                            resAdd : vaPlusVb });
  //rendering views/addResult.handlebars with js values for  {{a}} , {{b}} , {{resAdd}}
});

app.listen(8282 , function () {
  console.log("http://localhost:8282");
});
```

main template "views/layouts/***main.handlebars***"

```
<html>
<head>
   <meta charset="utf-8">
   <title>{{title}}</title>
</head>
<body>
   {{{body}}}
   <hr/>
   <a href="/server-home" >server-home page</a> <br/>
   <a href="/html/index.html" >(main) index.html</a>
</body>
</html>
```

template "views/***server-home.handlebars***"

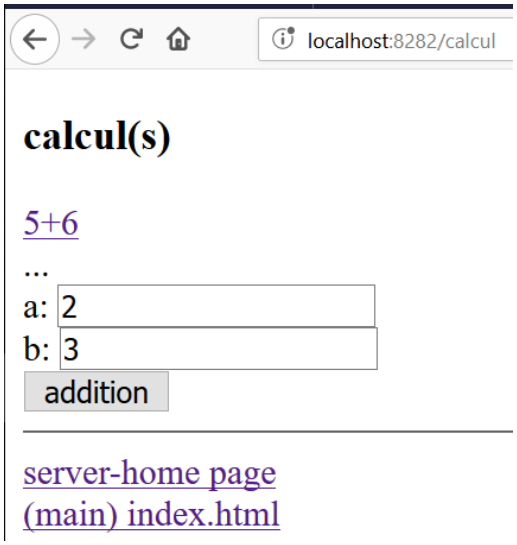
```
<p>...</p>  <a href="calcul"> nouveau calcul</a><br/>
```

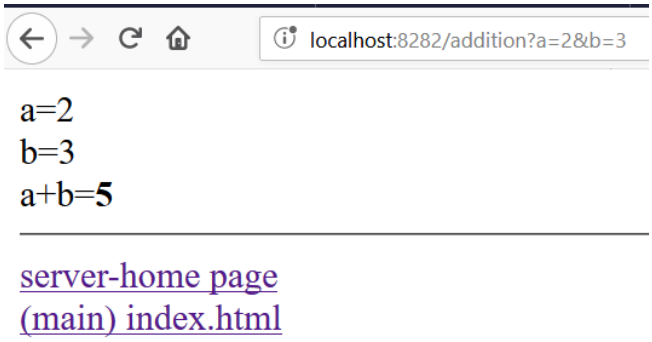
template "views/***calcul.handlebars***"

```
<h3>calcul(s)</h3>
<a href="addition?a=5&b=6">5+6</a><br/>
...
<form method="GET" action="addition">
   a: <input name="a" value="2" /> <br/>
   b: <input name="b" value="3" /> <br/>
   <input type='submit' value="addition">
</form>
```

template "views/***addResult.handlebars***"

```
a={{a}} <br/>
b={{b}} <br/>
a+b=<b>{{resAdd}}</b><br/>
```


localhost:8282/calcul
calcul(s)
5+6
...
a: 2
b: 3
addition
server-home page
(main) index.html


localhost:8282/addition?a=2&b=3
a=2
b=3
a+b=5
server-home page
(main) index.html

VI - Web services REST avec express

1. WS REST élémentaire avec node+express

1.1. Récupérer des données entrantes au format JSON

La récupération des valeurs JSON véhiculés en mode **POST** ou **PUT** dans le corps (***body***) de la requête entrante s'effectue avec la syntaxe **req.body** et nécessite la préparation et l'enregistrement d'un "***bodyParser***" :

```
...
var bodyParser = require('body-parser');
//import * as bodyParser from 'body-parser';
...
//support parsing of JSON post data
var jsonParser = bodyParser.json() ;
app.use(jsonParser);
...
//POST ... with body { "firstname" : "Jean"  , "lastname" : "Bon" }
app.post('/xyz', function(req : Request, res : Response) {
  var user = req.body ; //as javascript object
});
```

NB : il existe une variante de la méthode app.post() où l'on peut passer un "bodyParser" spécifique en second paramètre (pour certains cas pointus/spécifiques):

```
// POST /login gets urlencoded bodies :
app.post('/login', urlencodedParser, function (req, res) {
  res.send('welcome, ' + req.body.username)
})

// POST /api/users gets JSON bodies :
app.post('/api/users', jsonParser, function (req, res) {
  // use user in req.body
})
```

Dans le cas d'une api homogène (quasiment tout en JSON) , il est tout de même plus simple de paramétrer l'utilisation par défaut d'un bodyParser JSON via app.use() :

```
var jsonParser = bodyParser.json() ;
app.use(jsonParser)
```

1.2. Renvoyer des données/réponses au format JSON

La fonction prédéfinie ***res.send(jsObject)*** effectue en interne a peu près les opérations suivantes :

```
res.setHeader('Content-Type', 'application/json');
res.write(JSON.stringify(jsObject));
res.end();
```

Cette méthode ".send()" est donc tout à fait appropriée pour retourner la réponse "JSON" à un appel de Web Service REST .

1.3. Renvoyer si besoin des statuts d'erreur (http)

Via éventuelle fonction utilitaire :

```
function sendDataOrError(err,data,res){
        if(err==null) {
                        if(data!=null)
                                res.send(data);
                        else res.status(404).send(null);//not found
                }
                else res.status(500).send({error: err});//internal error (ex: mongo access)
}
```

Via "errorHandler" :

...

1.4. Exemple

```
var express = require('express');
var myGenericMongoClient = require('./my_generic_mongo_client');
var app = express();
…


function sendDataOrError(err,data,res){
        if(err==null) {
                        if(data!=null)
                                res.send(data);
                        else res.status(404).send(null);//not found
                }
                else res.status(500).send({error: err});//internal error (ex: mongo access)
}
```

```
// GET (array) /minibank/operations?numCpt=1
app.get('/minibank/operations', function(req, res,next) {
  myGenericMongoClient.genericFindList('operations', { 'compte' :
Number(req.query.numCpt) },
                function(err,tabOperations){
                        sendDataOrError(err,tabOperations,res);
                });
});

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

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 , myGenericMongoClient.***genericFind....()*** correspond à un élément d'un module utilitaire qui récupère des données dans une base mongoDB .

2. Avec mode post et authentification minimaliste

```
var express = require('express');
var bodyParser = require('body-parser');   //dépendance indirecte de express via npm
var app = express();

var myGenericMongoClient = require('./my_generic_mongo_client');

var uuid = require('uuid'); //to generate a simple token

app.use(bodyParser.json()); // to parse JSON input data and generate js object : (req.body)
app.use(bodyParser.urlencoded({ extended: true}));

...

// POST /minibank/verifyAuth  {  "numClient" : 1 , "password" : "pwd1" }
app.post('/minibank/verifyAuth', function(req, res,next) {
        var verifAuth = req.body; // JSON input data as jsObject with ok = null
        console.log("verifAuth :" +JSON.stringify(verifAuth));
         if(verifAuth.password == ("pwd" + verifAuth.numClient) ){
                verifAuth.ok= true;
                verifAuth.token=uuid.v4();
```

```
                //éventuelle transmission parallèle via champ "x-auth-token" :
                res.header("x-auth-token", verifAuth.token);
                //+stockage dans une map pour verif ulterieure : ....
                }
        else {
                verifAuth.ok= false;
                verifAuth.token = null;
        }
   res.send(verifAuth);    // send  back with ok = true or false and token
});

// GET /minibank/comptes/1
app.get('/minibank/comptes/:numero',
   displayHeaders,  verifTokenInHeaders  /*un peu securisé*/ ,
   function(req, res,next) {
        myGenericMongoClient.genericFindOne('comptes', { '_id' : Number(req.params.numero) },
                function(err,compte){
                                        sendDataOrError(err,compte,res);
                });
});

function sendDataOrError(err,data,res){
        if(err==null) {
                        if(data!=null)
                                res.send(data);
                        else res.status(404).send(null);//not found
                }
                else res.status(500).send({error: err});//internal error (ex: mongo access)
}


//var secureMode = false;
var secureMode = true;

function extractAndVerifToken(authorizationHeader){
        if(secureMode==false) return true;
        /*else*/
        if(authorizationHeader!=null ){
                if(authorizationHeader.startsWith("Bearer")){
                        var token = authorizationHeader.substring(7);
                        console.log("extracted token:" + token);
                        //code extremement simplifié ici:
                        //idealement à comparer avec token stocké en cache (si uuid token)
                        //ou bien tester validité avec token "jwt"
                        if(token != null && token.length>0)
                                return true ;
                         else
                                return false;
                }
                else
                        return false;
        }
        else
```

```
                    return false;
}

// verif bearer token in Authorization headers of request :
function verifTokenInHeaders(req, res, next) {
  if( extractAndVerifToken(req.headers.authorization))
     next();
  else
          res.status(401).send(null);//401=Unautorized or 403=Forbidden
}

// display Authorization in request (with bearer token):
function displayHeaders(req, res, next) {
  //console.log(JSON.stringify(req.headers));
  var authorization = req.headers.authorization;
  console.log("Authorization: " + authorization);
  next();
}

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

3. 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");
});
```

4. Divers éléments structurants

4.1. ORDRE IMPORTANT et Routers en tant que modules annexes

server.ts

```
import express  from 'express';
import * as bodyParser from 'body-parser';
export const  app :express.Application = express();
import { apiErrorHandler} from './api/apiErrorHandler'
import { apiRouter } from './api/apiRoutes';
import { initSequelize } from './model/global-db-model'


//PRE TRAITEMENTS (à placer en haut de server.ts) !!!!

//support parsing of JSON post data
var jsonParser = bodyParser.json() ;
app.use(jsonParser);


//ROUTES ORDINAIRES (après PRE traitements , avant POST traitements)

app.use(apiRouter); //delegate REST API routes to apiRouter
//app.use(apiRouter2); //delegate REST API routes to apiRouter2


//POST TRAITEMENTS (à placer en bas de server.ts) !!!

app.use(apiErrorHandler); //pour gérer les erreurs/exceptions
                          //pas rattrapées par try/catch et qui se propagent
                          //alors automatiquement au niveau "express appelant mon code"
                          //ou bien pour gérer les erreurs explicitement déléguées ici via next(err) ;

export const server = app.listen(8282 , function () {
   console.log("http://localhost:8282");
   initSequelize(); // ou autre initialisation
});
```

api/apiRoutes.ts

```
import { Request, Response ,NextFunction, Router} from 'express';
import { Devise } from '../model/devise';
//import { ErrorWithStatus , NotFoundError, ConflictError} from '../error/errorWithStatus'
import { MemoryMapDeviseService } from '../dao/memoryMapDeviseService';
```

```
import { SqDeviseService } from '../dao/sqDeviseService';
import { DeviseDataService } from '../dao/deviseDataService';

export const apiRouter = Router();

var  deviseService : DeviseDataService = new SqDeviseService();


apiRouter.route('/devise/:code')
.get( function(req :Request, res :Response , next: NextFunction ) {
    let codeDevise = req.params.numero;
    deviseService.findById(codeDevise)
     .then((devise)=> { res.send(devise) })
    .catch((err)=>{next(err);} );
});

//POST ... with body { "code": "M1" , "nom" : "monnaie1" , "change" : 1.123 }
apiRouter.route('/devise').post( function(req :Request, res :Response , next: NextFunction ) {
   let  devise :Devise =  req.body ; //as javascript object
   //deviseService.insert(devise)
   deviseService.saveOrUpdate(devise)
   .then((savedDevise)=> { res.send(savedDevise)})
   .catch((err)=>next(err));
});


// DELETE http://localhost:8282/devise/EUR
apiRouter.route('/devise/:code')
.delete( function(req :Request, res :Response , next: NextFunction ) {
   let codeDevise = req.params.code;
   deviseService.deleteById(codeDevise)
   .then(()=> { res.status(200).send({ "action" : "devise with code="+codeDevise + " was
deleted"});})
   .catch((err)=>next(err));
});


// http://localhost:8282/devise renvoyant tout [ {} , {}]
// http://localhost:8282/devise?changeMini=1.1 renvoyant [{}] selon critere
apiRouter.route('/devise').get(function(req :Request, res :Response , next: NextFunction ) {
   let  changeMini = req.query.changeMini;
   deviseService.findAll()
   .then((deviseArray)=> {
      if(changeMini){
         //filtrage selon critère changeMini:
         deviseArray = deviseArray.filter((dev)=>dev.change >= changeMini);
      }
      res.send(deviseArray)
   })
   .catch((err)=>next(err));
});
```

4.2. Gestionnaire d'erreurs

Classe(s) pratique(s) pour remonter des erreurs/exceptions avec plus de précision que Error("message") :

error/errorWithStatus.ts

```
// ErrorWithStatus est une version améliorée de Error (err.message)
// avec un attribut status (404,500,...) permettant une automatisation
// du retour du status http dans le "apiErrorHandler"

//NB: Error is a very special class (native)
//subclass cannot be test with instanceof , ...

export class ErrorWithStatus extends Error {
   public msg : string;
   public status : number
   constructor(message:string,status:number=500){
      super(message);       this.msg= message;       this.status=status;
   }
   public static extractStatusInNativeError(e: Error):number{
    let status=500; //500 (Internal Server Error)
    let jsonStr = JSON.stringify(e);
    let errWithStatus = JSON.parse(jsonStr);
    if(errWithStatus.status)
       status = errWithStatus.status;
    return status;
   }
}

export class NotFoundError extends ErrorWithStatus{
    constructor(message:string="not found",status:number=404){
       super(message,status);
    }
}

export class ConflictError extends ErrorWithStatus{
   constructor(message:string="conflict (with already existing)",status:number=409){
      super(message,status);
   }
}
```

gestionnaire d'erreur général (à enregistrer **à la fin** de server.ts) :

apiErrorHandler.ts

```
import { Request, Response ,NextFunction, ErrorRequestHandler} from 'express';
...
export const apiErrorHandler : ErrorRequestHandler =
function (err: any, req:Request, res: Response, next: NextFunction) {
//console.log("in apiErrorHandler err=", err + " " + JSON.stringify(err));
//console.log("in apiErrorHandler typeof err=",typeof err);
if(typeof err == 'string'){
  res.status(500).json({errorCode:'500', message: 'Internal Server Error :' + err});
}
else if(err instanceof Error){
  //console.log("in apiErrorHandler err is instanceof Error");
  let status = ErrorWithStatus.extractStatusInNativeError(err);
  res.status(status).json({errorCode:`${status}`, message: err.message});
}
else
  res.status(500).json({errorCode:'500', message: 'Internal Server Error'});
}
```

Le gestionnaire d'erreur ci dessus renvoie un status et un message d'erreur plus ou moins précis selon le fait que l'erreur est :

- une simple chaîne de caractère
- une instance de Error (peut être de type ErrorWithStatus et dont on essaie d'y extraire le status)
- une autre chose inconnue

Remontée des erreurs , exceptions dans apiRoutes.ts

Attention, attention : un simple throw "message erreur"

ou throw new Error("...")
ou throw new ErrorWithStatus("..." , 404) ;

ne fonctionne bien qu'en mode simpliste/synchrone .

En mode classique asynchrone, cela fait planter le serveur !!!

.../...

Solution 1 : via next(err)

```
//POST ... with body { "code": "M1" , "nom" : "monnaie1" , "change" : 1.123 }
apiRouter.route('/devise').post( function(req :Request, res :Response , next: NextFunction ) {
let devise :Devise = req.body ; //as javascript object
//deviseService.insert(devise)
deviseService.saveOrUpdate(devise)
.then((savedDevise)=> { res.send(savedDevise)})
.catch( (err)=> { next(err) });
});
```

Solution 2 : via wapper pour async et throw

fonction utilitaire permettant de bien remonter les erreurs/exceptions ou de renvoyer un bon résultat au format json:

```
function asyncToResp(fn : Function) {
    return function(req :Request, res :Response , next: NextFunction) {
        // Make sure to `.catch()` any errors and pass them along to the `next()`
        // middleware in the chain, in this case the error handler.
        fn(req, res, next)
        .then((data:object)=> { res.send(data) })
        .catch(next);
        };
}
```

Utilisation :

```
apiRouter.route('/devise/:code')
.get( asyncToResp( async function(req :Request, res :Response , next: NextFunction){
        let codeDevise = req.params.code;
        let devise = await deviseService.findById(codeDevise)
        return devise;
}));
```

et magiquement :

- en cas d'exception --> try/catch automatique et next(err) --> apiErrorHandler(err,...) déclenché automatiquement
- quand tout va bien --> valeur retournée dans fonction async --> Promise.resolve() automatique et res.send() déclenché automatiquement via asyncToResp()

VII - Accès aux bases de données (node)

1. Accès à MySQL via mysqljs/mysql (node)

1.1. Installation de mysqljs/mysql ou mysql2

Installation d'une version stable via

```
npm install --save mysql
```

ou bien

```
npm install --save mysql2
```

Exemple de version dans package.son :

```
...
"dependencies": {
  // "mysql": "^2.17.1",
    "mysql2": "^1.6.5",
   "sequelize": "^5.8.6"
 }
```

sites de référence : **https://github.com/mysqljs/mysql**
https://www.npmjs.com/package/mysql2

NB:

- mysql2 est une version améliorée de mysql d'un point de vue "performance" .
- l'api "mysql2" est en très grande partie compatible avec l'api "mysql".
- l'ORM "sequelize" nécessite (depuis la v4) l'utilisation de "mysql2"

1.2. Etablissement d'une connexion :

```
var mysql = require('mysql2');

var cnx = mysql.createConnection({
  host: "localhost",
  port: "3306",
  database : "minibank_db_node",
  user: "root",
  password: "root"
});

cnx.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  onConnectedToMysqlDB(cnx);
});
```

D'autres possibilités sont envisageables : déclencher une query() sur une connexion non établie active automatiquement un .connect() interne .

1.3. Déconnexion

Déconnexion douce (le temps de traiter les requêtes en cours):

```
cnx.end( function(err) {
  // The connection is terminated now
});
```

ou bien plus brutalement : ***cnx.destroy() ;*** sans callback pour une déconnexion immédiate

1.4. Déclenchement de requetes

Exemple simple de déclenchement de requête :

```
function onConnectedToMysqlDB(cn){
        var sql="select * from Client";
        cn.query(sql, function (err, results ) {
                if (err) throw err;
                console.log("Results: " + JSON.stringify(results));
        });
}
```

==>

Results: [{"nom":"Therieur","prenom":"Alex","numClient":2,...}]

Plus précisément **cn.query(sqlString, callback)** permet de déclencher une requête SQL simple (sans remplacement de valeur de paramètres variables) tandis que

.query(sqlStringWithParam, valuesOfParams, callback)

permet de passer une requête comportant des ? (liés à des paramètres variables) et un tableau de valeurs pour ces paramètres .

Exemples :

```
connection.query('SELECT * FROM `books` WHERE `author` = "David"',
                 function (error, results, fields) {
...
}) ;
```

```
connection.query('SELECT * FROM `books` WHERE `author` = ?', ['David'],
                 function (error, results, fields) {
}) ;
```

NB :

- La requête sql peut soit être passée à .query() sous forme de "string" , soit passée comme un objet littéral javascript avec des options { sql : "..." , timeout : 40000 }
- Si une requête SQL simple ne possède qu'un seul paramètre (?) , on peut éventuellement

passer directement cette valeur sans les [ ] d'un tableau englobant .

Exemple :

```
connection.query({
   sql: 'SELECT * FROM `books` WHERE `author` = ?' ,
   timeout: 40000, // 40s
  },
  ['David'] /* ou 'David' */,
  function (error, results, fields) {
    ...
  }
);
```

Finalement la fonction callback comporte jusqu'à 3 paramètres

```
function (error, results, fields) {
  // error will be an Error if one occurred during the query
  // results will contain the results of the query
  // fields will contain information about the returned results fields (if any)
}
```

NB : `connection.escape()` est appelée en interne pour remplacer de façons sophistiquées les valeurs des ? variables de la requête sql (prise en compte des formats , de la sécurisation, ...)

Exemples :

```
connection.query('UPDATE users SET foo = ?, bar = ?, baz = ? WHERE id = ?',
                        ['a', 'b', 'c', userId], function (error, results, fields) {
                                if (error) throw error;
                                // ...
});
```

```
var post  = {id: 1, title: 'Hello MySQL'};
var query = connection.query('INSERT INTO posts SET ?', post, function (error, results, fields) {
        if (error) throw error;
        // ....
});
console.log(query.sql); // INSERT INTO posts SET `id` = 1, `title` = 'Hello MySQL'
```

Récupération de la valeur d'une clef primaire auto-incrémentée par le serveur MySQL:

```
cn.query('INSERT INTO Client SET ?', {prenom: 'Jean' , nom: 'Bon'}, function (error, results) {
            if (error) throw error;
            var autoIncrId = results.insertId;
            console.log("autoIncrId="+autoIncrId + " " + typeof autoIncrId); //autoIncrId=6 number
   });
```

NB : au sein de la callback, **results.affectedRows** permet de récupérer le nombre de lignes affectées par un ordre SQL de type insert, update ou delete .

Et dans le cas particulier d'un ordre UPDATE , **results.changedRows** permet de connaître le nombre de lignes dont les valeurs ont changé .

1.5. Quelques exemples "CRUD" :

```
const nouvelleAdresse = { idAdr : null , codePostal: '76000' , ville: 'Rouen' , rue : '123 rue xyz' };
cn.query('INSERT INTO Adresse SET ?', nouvelleAdresse, function (error, results, fields) {
      if (error) throw error;
            console.log('In table Adresse, last insert ID:', results.insertId);
   });
```

```
cn.query('SELECT * FROM Adresse', function (error, results, fields) {
      if (error) throw error;
            console.log('list of Adresse:', JSON.stringify(results));
   });
```

```
cn.query('UPDATE Adresse SET rue = ? Where idAdr = ?',
                              ['rue qui va bien', 3], (err, result) => {
                                    if (err) throw err;
                                    console.log(`Changed ${result.changedRows} row(s)`);
});
```

```
cn.query('DELETE FROM Adresse WHERE idAdr = ?', [4], (err, result) => {
   if (err) throw err;
    console.log(`Deleted ${result.affectedRows} row(s)`);
});
```

Attention : ne pas enchaîner ceci en mode synchrone mais en mode asynchrone (avec éventuelles "Promise" ou async/await) !!!

1.6. Transactions simples :

```
connection.beginTransaction(function(err) {
 if (err) { throw err; }
 connection.query('INSERT INTO posts SET title=?', title, function (error, results, fields) {
   if (error) {
     return connection.rollback(function() {
       throw error;
     });
   }

   var log = 'Post ' + results.insertId + ' added';

   connection.query('INSERT INTO log SET data=?', log, function (error, results, fields) {
```

```
    if (error) {
      return connection.rollback(function() {
        throw error;
      });
    }
    connection.commit(function(err) {
      if (err) {
        return connection.rollback(function() {
          throw err;
        });
      }
      console.log('success!');
    });
  });
});
```

.beginTransaction(), .commit() et .rollback() sont de simples fonctions qui déclenchent en interne les ordres sql "START TRANSACTION", "COMMIT" et "ROLLBACK" .

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

2.1. Via mongodb/MongoClient (sans mongoose)

my_generic_mongo_client.js

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

//NB: sans connexion internet le localhost est moins bien géré que
// 127.0.0.1 sur certains ordinateurs (ex: windows 8 , 10 ).
var mongoDbUrl = 'mongodb://127.0.0.1:27017/test'; //by default
var currentDb=null; //current MongoDB connection

var setMongoDbUrl = function(dbUrl){
        mongoDbUrl = dbUrl;
}

var closeCurrentMongoDBConnection = function(){
        currentDb.close();
        currentDb=null;
}

var executeInMongoDbConnection = function(callback_with_db) {
  if(currentDb==null){
   MongoClient.connect(mongoDbUrl, function(err, db) {
        if(err!=null) {
```

```
                console.log("mongoDb connection error = " + err + " for dbUrl=" + mongoDbUrl );
        }
        assert.equal(null, err);//arret de l'execution ici si err != null
        console.log("Connected correctly to mongodb database" );
        currentDb = db;
        callback_with_db(db);
        });
  }else{
        callback_with_db(currentDb);  //réutilisation de la connection.
  }
}



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);
                        });
         });
};

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);
                 });
   });
};

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);
                 });
   });
};


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);
                }
                callback_with_err_and_item(err,item);
                });
  });
};

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

Utilisation :

```
var express = require('express');
var app = express();
var myGenericMongoClient = require('./my_generic_mongo_client');

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

// GET /minibank/clients/1
app.get('/minibank/clients/:numero',function(req, res,next) {
        myGenericMongoClient.genericFindOne('clients',
                { '_id' : Number(req.params.numero) },
                 /* anonymous callback function as lambda expression :*/
                        (err,client)=>{ sendDataOrError(err,client,res); }
                );
});

function sendDataOrError(err,data,res){
        if(err==null) {
                        if(data!=null)
                                res.send(data);
                        else res.status(404).send(null);//not found
                }
                else res.status(500).send({error: err});//internal error (ex: mongo access)
}
….
```

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"

ANNEXES

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

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

Mocha (+Chai) est la technologie de test préconisée pour nodeJs .
La technologie "mocha" ressemble beaucoup à jasmine . Elle doit être complétée par "chai" 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-dev

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

//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. variantes syntaxiques pour chai

- should() n'est qu'une variante de expect .
- assert se rapproche du style des assertions java/JUnit .

via assert.

```
const { assert } = require('chai')

assert(val)
assert.fail(actual, expected)
assert.ok(val)                          // is truthy
assert.equal(actual, expected)          // compare with ==
assert.strictEqual(actual, expected)    // compare with ===
assert.deepEqual(actual, expected)      // deep equal check

assert.isTrue(val)
assert.isFalse(val)

assert.isNull(val)
assert.isNotNull(val)
assert.isUndefined(val)
assert.isDefined(val)
assert.isFunction(val)
assert.isObject(val)
assert.isArray(val)
assert.isString(val)
assert.isNumber(val)
assert.isBoolean(val)

assert.typeOf(/tea/, 'regexp') // Object.prototype.toString()
assert.instanceOf(chai, Tea)
assert.include([ a,b,c ], a)
assert.match(val, /regexp/)
assert.property(obj, 'tea') // 'tea' in object
assert.deepProperty(obj, 'tea.green')
assert.propertyVal(person, 'name', 'John')
assert.deepPropertyVal(post, 'author.name', 'John')

assert.lengthOf(object, 3)
assert.throws(function() { ... })
assert.throws(function() { ... }, /reference error/)
```

```
assert.doesNotThrow

assert.operator(1, '<', 2)
assert.closeTo(actual, expected)
```

via expect(...).to... (BDD syntax : Behavior Driven Development) :

```
const { expect } = require('chai')

expect(object)
  .to.equal(expected)
  .to.eql(expected)        // deep equality
  .to.deep.equal(expected) // same as .eql
  .to.be.a('string')
  .to.include(val)

  .be.ok(val)
  .be.true
  .be.false
  .to.exist

  .to.be.null
  .to.be.undefined
  .to.be.empty
  .to.be.arguments
  .to.be.function
  .to.be.instanceOf

  .to.be.gt(5)  // aka: .above .greaterThan
  .to.be.gte(5) // aka: .at.least
  .to.be.lt(5)  // aka: .below

  .to.respondTo('bar')
  .to.satisfy((n) => n > 0)

  .to.have.members([2, 3, 4])
  .to.have.keys(['foo'])
  .to.have.key('foo')
  .to.have.lengthOf(3)

expect(() => { ··· })
  .to.throw(/not a function/)
```

1.4. Test de service interne en typescript (avec Promise es2015)

```
import chai from 'chai';
import { DeviseDataService } from "../dao/devisedataService";
import { MemoryMapDeviseService } from "../dao/memoryMapDeviseService";
import { Devise } from '../model/devise';
let expect = chai.expect;

var deviseDataService : DeviseDataService =new MemoryMapDeviseService();

describe("internal deviseService", function() {

   it("euro for code EUR", function(done) {
     deviseDataService.findById("EUR")
     .then((deviseEur)=> {
            expect(deviseEur.nom).equals("euro");
            console.log("**" +JSON.stringify(deviseEur));
            done();//pour indiquer a mocha que le test unitaire est fini
            })
     .catch((err)=>console.log("erreur:"+err));
   });

   it("saveOrUpdate_et_getByIdEnchaine", function(done) {
     let nouvelleDev :Devise = { code : "Da1" , nom : "devise a1" , change : 123};
     //let nouvelleDev :Devise = { code : "Da1Wrong" , nom : "devise a1" , change : 123};
     deviseDataService.saveOrUpdate(nouvelleDev)
     .then((dEnregistree)=>{
       //...
       return deviseDataService.findById("Da1");
     })
     .then((deviseRelue)=>{
       expect(deviseRelue.nom).equals("devise a1");
       done();
     })
     .catch((err)=>{ console.log("err:" + err);
                done(err);//mieux que expect.fail("...")
                 })
   });
  });
});
```

1.5. Test de service interne en typescript (avec async/await)

```
import chai from 'chai';
import { DeviseDataService } from "../dao/devisedataService";
import { MemoryMapDeviseService } from "../dao/memoryMapDeviseService";
import { Devise } from '../model/devise';
let expect = chai.expect;

var deviseDataService : DeviseDataService =new MemoryMapDeviseService();

describe("internal deviseService", function() {

  describe("getAllDevises", function() {
    it("returning at least 4 devises", async function() {
      let devises: Devise[] = await deviseDataService.findAll();
      expect(devises.length).to.gte(4); //greater or equals
    });
  });

  describe("getDeviseByCode", function() {
   it("euro for code EUR", async function() {
      let deviseEur: Devise = await deviseDataService.findById("EUR");
      //console.log(JSON.stringify(deviseEur));
      expect(deviseEur.nom).equals("euro");
    });

   it("saveOrUpdate_et_getByIdEnchaine", async function() {
    try{
      let nouvelleDev :Devise = { code : "Da1" , nom : "devise a1" , change : 123};
      //let nouvelleDev :Devise = { code : "Da1Wrong" , nom : "devise a1" , change : 123};
      let dEnregistree = await deviseDataService.saveOrUpdate(nouvelleDev);
      let deviseRelue = await deviseDataService.findById("Da1");
      expect(deviseRelue.nom).equals("devise a1");
    }
    catch(err){
      console.log("err:" + err);
       throw err;
      }
  });

  });

});
```

--> plus d'appel à done() mais préfixe **async** pour la fonction codant le test unitaire

--> plus d'appel à done(err) mais **throw** err dans bloc try/catch au sein d'une fonction async

1.6. Eventuels pré et post traitements (before, after)

```
describe("internal deviseService", function() {

 before(function(done) {
  // runs before all tests :
  //insertion d'un jeu de données:
  sequelize.sync({logging: console.log})
                              .then(
                                    ()=>{
          console.log("sequelize is initialized");
          deviseDataService.saveOrUpdate(new DeviseObject("EUR" , "euro" , 1))
          .then(()=>deviseDataService.saveOrUpdate(new DeviseObject("USD" , "dollar" , 1.1)))
          .then(()=>deviseDataService.saveOrUpdate(new DeviseObject("GBP" , "livre" , 0.9)))
          .then(()=>deviseDataService.saveOrUpdate(new DeviseObject("JPY" , "yen" , 132)))
          .then(()=>{done()});
        }
                              ).catch( (err:any) => { console.log('An error occurred :', err);  });

 });

 describe("getAllDevises", function() {
  it("returning at least 4 devises", async function() {
    let devises: Devise[] = await deviseDataService.findAll();
    expect(devises.length).to.gte(4); //greater or equals
  });
 });

 describe("getDeviseByCode", function() {
  it("euro for code EUR", async function() {
    let deviseEur: Devise = await deviseDataService.findById("EUR");
    //console.log(JSON.stringify(deviseEur));
    expect(deviseEur.nom).equals("euro");
  });
 });
});
```

before()	exécuté une seule fois avant tous les tests de même niveau
after()	exécuté une seule fois après tous les tests de même niveau
beforeEach()	exécuté (plusieurs fois) avant chaque test de même niveau
afterEach()	exécuté (plusieurs fois) après chaque test de même niveau

1.7. Test de web service REST via "request"

NB :

- **request** est une api (module) de nodeJs qui permet d'**effectuer des appels HTTP** .
- **request** peut être utiliser au sein de test mais également au sein d'une application appelant en interne certains services vers une autre application (délégation) .
- **un test (d'intégration ou "end-to-end") basé sur request ne peut fonctionner que si le serveur à tester a été préalablement démarré** .

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 traitée apres appel asynchrone)
      });
    });

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

1.8. Test de web service rest via chai-http (ici en typescript)

chai-http est une extension de chai qui permet de :

- effectuer des assertions/vérifications de niveau "communication http"
- lancer si besoin le serveur nodeJs à tester

dans package.json :

```
...
"devDependencies": {
"@types/chai": "^4.1.7",
"@types/chai-http": "^4.2.0",
"@types/express": "^4.16.1",
"@types/mocha": "^5.2.7",
"chai": "^4.2.0",
"chai-http": "^4.3.0",
"mocha": "^6.1.4"
}
```

server.ts

```
import express from 'express';
export const app :express.Application = express();
import { apiRouter } from './api/apiRoutes';
import { initSequelize } from './model/global-db-model'
...
//ROUTES ORDINAIRES (apres PRE traitements , avant POST traitements)
app.use(apiRouter); //delegate REST API routes to apiRouter
...
export const server = app.listen(8282 , function () {
        console.log("http://localhost:8282");
        initSequelize();
});
```

test/xyz.spec.ts

```
import chai from 'chai';
import chaiHttp from 'chai-http';
import { app , server } from '../server';
import { Devise } from '../model/devise';
let expect = chai.expect;

// Configure chai :
chai.use(chaiHttp);
//chai.should();

describe("devise api", function() {

  before(function(done) {
    // runs before all tests :
    //insertion d'un jeu de données via http call:
    chai.request(app)
              .post('/devise')
              .send({code:"EUR" , nom : "euro" , change : 1 })
              .end((err, res) => { done(); });
  });

  after(function() {
    // runs after all tests : close server
    server.close();
  });

  describe("getDeviseByCode", function() {

    it("returns status 200 and a devise object with good name", function(done) {
      chai.request(app)
              .get('/devise/EUR')
              .end((err, res) => {
                 //res.should.have.status(200);
                 chai.expect(res).status(200);
                 let obj = res.body;
                 //obj.should.be.a('object');
                 chai.expect(obj).a('object');
                 let devise = <Devise> obj;
                 //console.log(JSON.stringify(devise));
                 chai.expect(devise.nom).equals("euro");
                 done();
              });
    });
  });
});
```

IX - Annexe – ORM Sequelize

1. ORM Sequelize (node)

"**Sequelize**" est une des technologies "ORM" disponibles dans l'écosystème nodeJs .
"Sequelize" est en 2018,2019 la **technologie "ORM"** la plus utilisée dans le monde "**nodeJs**"

"ORM" signifie "**O**bject **R**elational **M**apping" .
Dans le contexte nodeJs/Sequelize , des objets "javascript" pourront être mis en correspondance avec des enregistrements d'une des bases de données relationnelles utilisables :
- **mysql** ou **mariadb** *(via mysql2 ou mariadb)*
- **postgres** *(via pg pg-hstore)*
- **sqlLite** *(via sqlite3)*
- **sqlserver** *(de Miscrosoft) via tedious*

L'**api Sequelize** est ***asynchrone*** et s'appuie sur les "***Promises***" de es2015 et peut être invoquée via async/await de es2017 . L'utilisation de "Typescript" est possible et facultative .

Principe de fonctionnement de "Sequelize" :

1. **Paramétrage d'une connexion** à une base de données relationnelle (avec ***dialect=mysql*** par exemple)
2. **Paramétrage d'un modèle objet de persistance** (pendant "Sequelize" d'un "schéma relationnel") : ***Liste de classes d'objets persistants*** avec ***structures précises*** et éventuelles associations (1-n , n-n, ...)
3. **Utilisation des classes d'objets persistants définies au niveau du modèle** sequelize pour déclencher les opérations de persistance (CRUD) avec un code "orienté objet" et des ***requêtes SQL générées et déclenchées automatiquement*** .

Ces 3 phases peuvent être réparties au sein de fichiers/modules complémentaires :
- ***db-config.js*** et ***database.cfg.json*** *(config connexion à une base de données en mode "sequelize")*
- ***db-model.js*** *(définitions des classes persistantes (proche "tables") du modèle "sequelize")*
- ***my-sequelize-app.js*** *(utilisation des classes définies dans le modèle)*

1.1. Configuration d'une connexion à une base de données

db-config.js *(version simple avec URL de connexion)*

```
var Sequelize = require('sequelize');
const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');
//on exporte pour utiliser notre connexion depuis les autre fichiers :
var exports = module.exports = {};  exports.sequelize = sequelize;
```

version plus sophistiquée avec fichier de paramétrage :

database.cfg.json

```
{ "dev": {    "dialect": "mysql",  "host": "localhost", "port": 3306,
              "database": "minibank_db_node2",
              "user": "root",        "password": "root" },
  "prod": {    "dialect": "",   "user": "",     "database": " ",         "password": ""          }
}
```

db-config.js *(version classique)*

```
var Sequelize = require('sequelize');

var env="dev" ; //or "prod"
var confDb = require('./database.cfg.json')[env];
var password = confDb.password ? confDb.password : null;

// initialize database connection :
//sequelize = new Sequelize('database', 'username', 'password',
//                          {host: 'localhost', dialect: 'mysql',logging: false,});
var sequelize = new Sequelize(confDb.database, confDb.user, password,
                          { dialect: confDb.dialect,
                            port : confDb.port,
                          logging: false, // false or console.log,// pour voir les logs de sequelize
                          define: {
                                  timestamps: false
                                  }
                          }
                  );

sequelize.authenticate()
  .then(function() {
                          console.log('Connection has been established successfully.');
        }, function(err) {
                  console.log('Unable to connect to the database:', err);
        });

//on exporte pour utiliser notre connexion depuis les autre fichiers :
var exports = module.exports = {};  exports.sequelize = sequelize;
```

1.2. Configuration d'un modèle de persistance (sequelize)

Cette première approche se focalisera sur les définitions fondamentales , certains détails seront approfondis ultérieurement :

db-model.js

```
var sequelize = require('./db-config.js');
var Sequelize = require("sequelize");//Sequelize.STRING,Sequelize.INTEGER, ....

class Customer extends Sequelize.Model {};
Customer.init({
      /*id: {type: Sequelize.INTEGER, autoIncrement: true, primaryKey: true},*/
         lastName: { type: Sequelize.STRING(64),allowNull: false          },
         firstName: { type: Sequelize.STRING(64),allowNull: false          },
         phoneNumber: { type: Sequelize.STRING(16),allowNull: false    },
         email: { type: Sequelize.STRING(64),allowNull: true        }
      },
      { sequelize, modelName: 'customer' ,  freezeTableName: true });

var AddressOfCustomer = sequelize.define('addressOfCustomer', {
         idAddr: {type: Sequelize.INTEGER, autoIncrement: true, primaryKey: true},
         numberAndStreet: { type: Sequelize.STRING(64),allowNull: false        },
         zip: { type: Sequelize.STRING(64),allowNull: false          },
         town: { type: Sequelize.STRING(64),allowNull: false        }
         },
         { tableName: 'address_of_customer', timestamps: false , underscored : true});

var exports = module.exports = {};
exports.sequelize = sequelize;
exports.AddressOfCustomer = AddressOfCustomer;
exports.Customer = Customer;
```

NB : 2 façons de définir une classe de persistance :

```
class Xxx extends Sequelize.Model {};
   et Xxx.init({attrDefs } , { sequelize, modelName: 'xxx' });
```

ou bien

```
var Xxx = sequelize.define( { attrDefs} , { tableName= 'Xxx' , ... });
```

sachant que sequelize.define() appel .init() en interne

NB: si ***timestamps : true*** (par défaut) alors

```
CREATE TABLE IF NOT EXISTS "Xxx" (
 "id" SERIAL,
 "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL,
 "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL,
 ...)
```

Nom de table par défaut selon paramètre freezeTableName :

Si modelName="customer" alors

default tableName = "customer**s**" *(with **s** suffix)* without freezeTableName: true

ou bien

default tableName = "customer" *(without s suffix)* with ***freezeTableName: true***

Clef primaire par défaut avec Sequelize pour chaque table :

id: {type: Sequelize.INTEGER or ... or Sequelize.SERIAL ,
autoIncrement: true, primaryKey: true},

Création des tables par défaut si elles n'existent pas déjà :

Type d'ordre SQL déclenché à l'initialisation de "Sequelize" :

```
CREATE TABLE IF NOT EXISTS `customer`
```

Conséquences :

- Si la table 'cutomer' existe déjà avec une structure différente ==> BUG , la TABLE ou bien la DATABASE doit être re-créée (drop , create) .
- En mode développement un script de création/réinitialisation de base peut être pratique et a souvent besoin d'être relancé :

```
DROP DATABASE IF EXISTS minibank_db_node2;
CREATE DATABASE minibank_db_node2 charset=utf8;
```

Nom de colonnes "snake_case" ou "camelCase" :

- Par défaut les noms des colonnes sont les mêmes que ceux mentionnés dans les définitions (souvent en "camelCase" (ex : *idAdr* , *numberAndStreet*).
- Si le paramètre **underscored** est fixé à la valeur **true** alors tous les noms de colonne d'une table seront en "snake_case" (ex: **id_addr** , **number_and_street**)

NB : Plein d'autres détails sont précisés dans la documentation officielle de "sequelize" .

1.3. Utilisation du modèle de persistance

my-sequelize-app.js

```
var MyModel = require('./db-model.js');

//MyModel.sequelize.sync()
MyModel.sequelize.sync({logging: console.log})
                        .then( ()=>{ doJobWithSequelize();})
                        .catch( (err) => { console.log('An error occurred :', err);  });
```

1.4. Insertions/créations :

```
MyModel.Customer.create({id:null, firstName: 'Jean', lastName: 'Bon',
                         phoneNumber : '0605040302',
                         email: 'jean.bon@charcuterie.fr'  })
.then((c) => { console.log("saved Customer :" + JSON.stringify(c));
        console.log("Customer auto-generated ID:", c.id);          }
 );
```

==>
saved Customer : { "id":1,"firstName":"Jean","lastName":"Bon",
"phoneNumber":"0605040302","email":"jean.bon@charcuterie.fr"}
Customer auto-generated ID: 1

1.5. Select/recherches :

Recherches multiples :

```
const findCriterias = {
      where: {lastName : 'Therieur'}, //on veux uniquement ceux qui ont lastName = 'Therieur'
      order: [['lastName', 'ASC']] //classer par ordre alphabétique sur le lastName
   };

//MyModel.Customer.findAll().then(adrs => {
MyModel.Customer.findAll(findCriterias).then(customers => {
      //on récupère ici un tableau "customers" contenant une liste de customers
      console.log("*** Customers de nom=Therieur :" + JSON.stringify(customers));
   })
.catch(function (e) {  console.log(e);    });
```

==>

**** Customers de nom=Therieur :*
[{"id":2,"lastName":"Therieur","firstName":"Alex","phoneNumber":"0605040301",
"email":"alex.therieur@charcuterie.fr"},
{"id":3,"lastName":"Therieur","firstName":"Alain","phoneNumber":"0608040301",
"email":"alex.therieur@charcuterie.fr"}]

Recherche unique/précise :

```
MyModel.Customer.findOne( { where : { id : 1} } ).then(c => {
      //on récupère ici l'unique Customer recherchée par sont id (primary key)
      console.log("** Customer avec id=1 : " + JSON.stringify(c));
   }) ;
```

==>

*** Customer avec id=1 : {"id":1,"lastName":"Bon","firstName":"Jean",*
"phoneNumber":"0605040302","email":"jean.bon@charcuterie.fr"}

Recherche selon clef primaire :

```
MyModel.Customer.findByPk(1).then(c => {
      console.log("** Customer avec id=1 : " + JSON.stringify(c));
   }).catch(function (e) {       console.log(e);    });
```

Recherche avec requête SQL brute spécifique (comme si sans ORM) :

```
const mySpecificRawSqlRequest = "SELECT firstName as prenom , lastName as nom ,
phoneNumber as telephone FROM customer WHERE id = $1";
const param_id_customer=1;
MyModel.sequelize.query(mySpecificRawSqlRequest,
                                {bind: [param_id_customer],
                                 type: MyModel.sequelize.QueryTypes.SELECT})
         .then(results => {
      console.log(">>>> Specific Request results="+JSON.stringify(results));
   }) ;
```

==>

>>>> Specific Request results=[{"prenom":"Jean","nom":"Bon","telephone":"0605040302"}]

1.6. Update/Mise à jour :

//exemple de requête d'update d'un Customer :

```
MyModel.Customer.update(
        {phoneNumber: '0606060606'},
        {where: {id: 1}}
   ).then(nbCustomersModifies => {
      console.log("nbCustomersModifies:" + nbCustomersModifies);
   }).catch(function (e) {
      console.log(e);
   });
```

==> nbCustomersModifies:1

1.7. Delete/suppression :

```
MyModel.Customer.destroy({  where: { id: 3 }}).then(() => {
  console.log("Customer avec id=3 supprimé");
}).catch(function (e) {
    console.log(e);
 });
```

```
//fonction supprimant tous les éléments d'une table selon le modelClassName "Customer" or ... :
function deleteAllPromise(modelClassName){
        return MyModel[modelClassName].destroy({ where: {} });
}
```

1.8. Enchaînement "CRUD" avec Promise ".then ... .then"

Rappel important :
Les fonctions de l'api "Sequelize" sont **asynchrones** et les valeurs de retour sont des "**Promise**" que l'on peut gérer par .then().then().catch().

Exemple d'enchaînement simple :

L'exemple suivant va enchaîner :
- la suppression de tous les anciens "customers"
- insertions d'un "customer"
- récupération du client enregistré dans variable c1 (avec c1.id = résultat auto_increment)
- idem pour c2 et c3
- la récupération de tous les "Customer" (une fois toutes les insertions effectuées)
- la mise à jour en base du numéro de téléphone du client c1
- la récupération des valeurs modifiées et enregistrées du client c1 (avec nouveau numéro de téléphone)

```
let c1,c2,c3;
deleteAllPromise("Customer").
then(()=>MyModel.Customer.create({id:null, firstName: 'Jean', lastName: 'Bon',
            phoneNumber : '0605040302',    email: 'jean.bon@charcuterie.fr'  }))
.then((c) => { c1 = c;
     return MyModel.Customer.create({id:null, firstName: 'Alex', lastName: 'Therieur',
           phoneNumber : '0605040301',    email: 'alex.therieur@charcuterie.fr'  });
})
.then((c) => { c2 = c;
          return MyModel.Customer.create({id:null, firstName: 'Alain', lastName: 'Therieur',
       phoneNumber : '0608040301',    email: 'alex.therieur@charcuterie.fr'  });
})
.then((c) => { c3 = c;  })
.then(() => /* implicit return without { } */ MyModel.Customer.findAll())
.then(customers => { console.log("*** Customers :" + JSON.stringify(customers));    })
.then(()=>MyModel.Customer.update({phoneNumber: '0606060606'},
                          {where: {id: c1.id}})
 )
.then(nbCustomersModifies => {
      console.log("nbCustomersModifies:" + nbCustomersModifies);
                  return MyModel.Customer.findByPk(c1.id);
   })
.then(c => console.log("** updated Customer avec id=1 : " + JSON.stringify(c))    )
.catch((e) =>{  console.log(e); })
```

==>

**** Customers :*
*[{"id":4,"lastName":"Bon","firstName":"Jean",**"phoneNumber":"0605040302"**,"email":"jean.bon@charcuterie.fr"},"id":5,"lastName":"Therieur","firstName":"Alex","phoneNumber":"0605040301","email":"alex.therieur@charcuterie.fr"},"id":6,"lastName":"Therieur","firstName":"Alain","phoneNumber":"0608040301","email":"alex.therieur@charcuterie.fr"}]*
nbCustomersModifies:1
*** updated Customer avec id=1 : {"id":4,"lastName":"Bon","firstName":"Jean",*
***"phoneNumber":"0606060606"**,"email":"jean.bon@charcuterie.fr"}*

Même exemple plus lisible avec async/await de es2017 :

```
async function doJob1WithSequelizeAndAsyncAwait(){

try{
await deleteAllPromise("Customer");
let c1= await MyModel.Customer.create({id:null, firstName: 'Jean', lastName: 'Bon',
     phoneNumber : '0605040302',  email: 'jean.bon@charcuterie.fr'  });
let c2 = await MyModel.Customer.create({id:null, firstName: 'Alex', lastName: 'Therieur',
           phoneNumber : '0605040301',    email: 'alex.therieur@charcuterie.fr'  });
let c3 = await MyModel.Customer.create({id:null, firstName: 'Alain', lastName: 'Therieur',
        phoneNumber : '0608040301',    email: 'alex.therieur@charcuterie.fr'  });
const customers = await MyModel.Customer.findAll();
console.log("### Customers :" + JSON.stringify(customers));
let nbCustomersModifies = await MyModel.Customer.update({phoneNumber: '0606060606'},
                          {where: {id: c1.id}});
console.log("nbCustomersModifies:" + nbCustomersModifies);
c1= await MyModel.Customer.findByPk(c1.id);
console.log("## updated Customer avec id=1 : " + JSON.stringify(c1))
}
catch(e){
          console.log(e);
}

} //Rappel : await ne peut être appelé qu'au sein d'une fonction déclarée "async" et permet
d'attendre la valeur issue de la résolution d'une promesse (valeur de retour d'une fonction appelée)
```

1.9. Associations (jointures) : vue d'ensemble

- BelongsTo
- HasOne
- HasMany
- BelongsToMany

1.10. Associations "1-n" / "n-1" classique (foreignKey)

Exemple "plusieurs Operations bancaires pour un Compte" :

```
var sequelize = require('./db-config.js');
var Sequelize = require("sequelize"); //Sequelize.STRING,Sequelize.INTEGER, ....

class Account extends Sequelize.Model {};
Account.init({
    number: {type: Sequelize.INTEGER, autoIncrement: true, primaryKey: true},
   label: { type: Sequelize.STRING(64),allowNull: false   },
   /*solde*/balance: { type: Sequelize.DOUBLE ,allowNull: false  , validate : { min: -1500 }   }
}, { sequelize, modelName: 'account' ,  freezeTableName: true });

class Operation extends Sequelize.Model {};
Operation.init({
   number: {type: Sequelize.INTEGER, autoIncrement: true, primaryKey: true},
   label: { type: Sequelize.STRING(64),allowNull: false   },
   amount: { type: Sequelize.DOUBLE ,allowNull: false            },
  dateOp: { type: Sequelize.DATEONLY ,allowNull: false          }
}, { sequelize, modelName: 'operation' ,  freezeTableName: true });

Operation.belongsTo(Account); //une opération est un détail d'un seul compte
//--> on pourra accéder au compte lié à l'opération courante via op.account .

Account.hasMany(Operation);  //un compte peut comporter plusieurs opérations
//--> on pourra parcourir les opérations liées à un compte via compteXy.operations
//--> on pourra ajouter une operation à un compte via compteXy.addOperation(opZz) ;

var exports = module.exports = {};  exports.sequelize = sequelize;
exports.Account = Account;
exports.Operation = Operation;
```

Remarques :

Effet sur le modèle relationnel :

Une seule des 2 instructions suivantes (***Operation.belongsTo(Account);*** ou bien ***Account.hasMany(Operation);***) suffit à générer une clef étrangère dans la table "operation(s)" .

Le nom par défaut de la clef étrangère (ici dans la table operation) est '***accountNumber***'
soit = ***modelName associé*** + ***pkName du modèle (table) référencé(e)*** .

Effet de "as :" sur le modèle objet (javascript) :

```
//Account.hasMany(Operation , { as : "lastOperations" } );
```

as ="roleName" correspond à une vision orientée objet de l'association (role UML)
et correspondra à un attribut de l'objet javascript permettant de naviguer d'un niveau à l'autre
NB: **la valeur par défaut est le modelName référencé (avec un suffixe "s" si "many").**
la valeur de as est considérée par sequelize comme un "alias" qu'il faut régulièrement préciser
(exemple : MyModel.Account.findAll({ include: [{model: MyModel.Operation , as: "lastOperations" }] }))
==>pour faire simple modelName commençant par une minuscule et pas trop de "as:"

Exemple de code exploitant l'association "n-1" / "1-n" :

```
let ac1 ;
...
.then( () => MyModel.Account.create({number:null, label: 'bank_account_xx', balance : 100.0  }) )
.then( (ac) => { ac1=ac;
        return MyModel.Operation.create({number:null, label: 'achat x1',
                                                amount : -50.0 , dateOp:"2019-05-23"  });
        })
.then((newOp) => ac1.addOperation(newOp))
...
```

NB : ***.addOperation()*** existe du fait de l'instruction Account.***hasMany***(Operation);

```
...
.then(()=> MyModel.Account.findAll( { include: [{model: MyModel.Operation} ] }))
.then(accounts => {
      //console.log("*** accounts =" + JSON.stringify(accounts));
      accounts.forEach((account) => {
        //console.log("** account =" + JSON.stringify(account));
        console.log(`* account ${account.number} ${account.label} ${account.balance} `);
              account.operations.forEach((op) => {
                  console.log(`****** operation ${op.number} ${op.label}
                                ${op.amount} ${op.dateOp}`);
                });
      });
  })
....
```

NB :
{ include: [{model: MyModel.Operation}] } permet une sorte de "***fetch JOINED operations***" pour chaque opération remontée par findAll() .
et ***account.operations*** existe du fait de la déclaration Account.***hasMany***(Operation); au sein du modèle.

2. Utilisation de Sequelize v5.x avec Typescript

config/database.cfg.ts

```
export default{
"dev": {
        "dialect": "mysql",
        "host": "localhost",
        "port": 3306,
        "database": "deviseApiDb",
        "user": "root",
        "password": "root"
        },
"prod": {
        "dialect": "mysql",
        "host": "devise.db.service",
        "port": 3306,
        "database": "deviseApiDb",
        "user": "root",
        "password": "root"
         }
}
```

config/db-config.ts

```
import dbCfg from './database.cfg';
export interface IDbConfig {
        dialect: "mssql" | "mysql" | "postgres" | "sqlite" | "mariadb",
        host: string,
        port: number,
        database : string
        user: string
        password : string ;
}

let mode = process.env.MODE;
//env variable MODE=dev or MODE=prod when launching node
if(mode == undefined) mode = "dev";
//console.log("in db-config, mode="+mode);
export const confDb : IDbConfig =
            (mode == "dev") ? dbCfg.dev as any : dbCfg.prod as any;
console.log("in db-config, confDb.host="+confDb.host);
if(confDb.port === undefined) {
  if(confDb.dialect=="mysql") {
          confDb.port=3306;
  }
}
```

model/devise.ts

```
//GENERIC PART (SEQUELIZE OR NOT)
export interface Devise  {
  code :string ;
  nom :string ;
  change :number ;
}
//exemples: ("USD" , "dollar" , 1) ,  ("EUR" , "euro" , 0.9)

//real class for instanciation ,  with constructor .
export class DeviseObject implements Devise {
  constructor(public code:string = "?" ,   public nom:string = "?",  public change:number= 0){
  }
}
```

Le fichier ci-dessus est volontairement indépendant de Sequelize !!!

La partie spécifique "Sequelize" est placée dans le fichier annexe suivant :

model/sq-devise.ts

```
import { Devise } from './devise';
import { Sequelize, Model, DataTypes, BuildOptions } from 'sequelize';
/* import { HasManyGetAssociationsMixin, HasManyAddAssociationMixin,
HasManyHasAssociationMixin, Association, HasManyCountAssociationsMixin,
HasManyCreateAssociationMixin } from 'sequelize'; */


//SqDevise est une interface mixant la structure de données "Devise" au "Model" Sequelize :
export interface /*DeviseModel*/ SqDevise extends Model, Devise {
}
 // Need to declare the static model so `findOne` etc. use correct types.
 export type DeviseModelStatic = typeof Model & {
   new (values?: object, options?: BuildOptions): SqDevise /*DeviseModel*/;
 }
 // fonction exportée (à appeler) définissant la structure de la table (dans base de données) :
 export function initDeviseModel(sequelize: Sequelize):DeviseModelStatic{
 const DeviseDefineModel = <DeviseModelStatic> sequelize.define('devise', {
     code: {type: DataTypes.STRING(32), autoIncrement: false, primaryKey: true},
     nom: { type: DataTypes.STRING(64),allowNull: false          },
     change: { type: DataTypes.DOUBLE,allowNull: false          }
     }, {    freezeTableName: true , });
 return DeviseDefineModel;
}
```

NB :

- avec Sequelize v4 , il fallait utiliser @types/sequelize (avec XxxAttributes , XxxInstance , ...)
- depuis Sequelize v5, la partie "définitions typescript" est déjà intégrée dans "sequelize" et il faut faire avec la documentation officielle "http://docs.sequelizejs.com/manual/typescript"
- besoin de `@types/node` ,`@types/validator` , `@types/bluebird` dans package.json :

```
"devDependencies": {
"@types/bluebird": "^3.5.27",
"@types/express": "^4.16.1",
"@types/node": "^12.0.8",
"@types/validator": "^10.11.1",
}
```

model/global-db-model.ts

```
import { Sequelize , Model }from "sequelize";
import { DeviseModelStatic , initDeviseModel } from './sq-devise';
import { confDb  } from "../config/db-config"
//import { PaysModelStatic , initPaysModel } from "./sq-pays";

export class MyApiModels {
   public devises! : DeviseModelStatic;
   //public pays! : PaysModelStatic;
 }

 export class MySqDatabase {
   private models: MyApiModels;
   private sequelize: Sequelize;
   public dbname: string = "unknown";

   constructor() {
    this.models = new MyApiModels();
    let model: any;
    let sqOptions = {
        dialect: confDb.dialect,
        port : confDb.port,
        host : confDb.host,
```

```
        logging: /*console.log*/false, // false or console.log,// permet de voir les logs de sequelize
        define: {
           timestamps: false
        }
    };
    var password = confDb.password ? confDb.password : "";
    this.sequelize = new Sequelize(confDb.database, confDb.user, password, sqOptions);
    this.dbname = confDb.database;

    this.models.devises= initDeviseModel(this.sequelize);
    //this.models.pays= initPaysModel(this.sequelize) ;
  }

  getModels() {      return this.models;    }
  getSequelize() {      return this.sequelize;    }
}

export const database: MySqDatabase = new MySqDatabase();
export const models = database.getModels();
export const sequelize: Sequelize = database.getSequelize();

export function initSequelize(){
  sequelize.sync({logging: console.log})
        .then( ()=>{    console.log("sequelize is initialized");   }
        ).catch( (err:any) => { console.log('An error occurred :', err);  });
}
```

Utilisation au sein de **dao/sqDeviseService**

```
import { DeviseDataService } from './deviseDataService'
import { Devise } from '../model/devise';
import { SqDevise, DeviseModelStatic } from '../model/sq-devise';
import { models } from '../model/global-db-model';
import { NotFoundError, ConflictError } from '../api/apiErrorHandler';
//"strictNullChecks": false in tsconfig.json
```

```
export class SqDeviseService implements DeviseDataService{

  deviseModelStatic : DeviseModelStatic = models.devises;

  constructor(){}

  findById(code: string): Promise<Devise> {
    return new Promise<Devise>((resolve: Function, reject: Function) => {
      this.deviseModelStatic.findByPk(code)
       .then((obj: SqDevise) => {
         //returning null by default if not Found
         if(obj!=null)
            resolve(obj);
         else
           reject(new NotFoundError("devise not found with code="+code));
        })
        .catch((error: Error) => { reject(error);});
     });
  }

  findAll(): Promise<Devise[]> {
    return new Promise((resolve,reject) => {
       this.deviseModelStatic.findAll()
      .then((objects: SqDevise[]) => { resolve(objects);  })
       .catch((error: Error) => {  reject(error);  });
    });
  }


  insert(d: Devise): Promise<Devise> {
    return new Promise((resolve,reject) => {
       this.deviseModelStatic.create(d)
       .then((obj: SqDevise) => { resolve(obj); })
       .catch((error: any) => {   reject(error);  });
   });
  }
```

```
  saveOrUpdate(d: Devise): Promise<Devise> {
    return new Promise((resolve,reject) => {
      //.upsert() is appropriate for saveOrUpdate if no auto_incr
      this.deviseModelStatic.upsert(d)
      .then((ok: Boolean) => {  resolve(d); })
      .catch((error: any) => {  reject(error); });
   });
  }

  deleteById(codeDev: string): Promise<void> {
     return new Promise((resolve,reject) => {
        this.deviseModelStatic.destroy( { where: { code : codeDev} } )
        .then(() => {  resolve(); })
        .catch((error: any) => {   reject(error);  });
    });
  }
}
```

2.1. compléments si associations

model/sq-devise.ts

```
...
import { HasManyGetAssociationsMixin, HasManyAddAssociationMixin,
HasManyHasAssociationMixin, Association, HasManyCountAssociationsMixin,
HasManyCreateAssociationMixin } from 'sequelize';
...
export interface /*DeviseModel*/ SqDevise extends Model, Devise {
  getPays: HasManyGetAssociationsMixin<Pays>;
  addPays: HasManyAddAssociationMixin<Pays, number>;
  hasPays: HasManyHasAssociationMixin<Pays, number>;
  countPays: HasManyCountAssociationsMixin;
 }
...
```

NB : sqDeviseEuro.**addPays**(objPaysFr) ou bien sqDeviseEuro.**addPays**(idPaysFr)

dans **model/global-db-model.ts**

```
...
  export class MySqDatabase {
    ...
```

```
  constructor() {
    ....
   this.models.devises= initDeviseModel(this.sequelize);
   this.models.pays= initPaysModel(this.sequelize);
   this.models.devises.hasMany(this.models.pays);
  }
 ....
 }
 ...
```

Exemple d'utilisation :

```
async function doAssociationJob(){
 try{
   let devC1 : SqDevise =
    await models.devises.findByPk("CC1", { include :[models.devises.associations.pays]});
  if(devC1==null){
              devC1= await models.devises.create({code: 'CC1',name: 'MonaieC1',change : 1234 });
              let pays1forDevC1 :Pays = await models.pays.create({name:"Pays1",capitale:"Cap1"});
              await devC1.addPays(pays1forDevC1);
              console.log(JSON.stringify(pays1forDevC1));
              let pays2forDevC1 :Pays = await models.pays.create({name:"Pays2",capitale:"Cap2"});
              await devC1.addPays(pays2forDevC1);
              console.log(JSON.stringify(pays2forDevC1));
             }
              //console.log(devC1.code, devC1.name, devC1.change);
              console.log(JSON.stringify(devC2)); //devise avec sous partie .pays

             }
      catch(e){
            console.log(JSON.stringify(e));
        }
}
```

X - Annexe – apidoc (pour api rest)

1. Api doc

"Api Doc" est une extension *(existante dans plein de langage de programmation : Javascript, java , php, python, ...)* **permettant de générer une bonne documentation sur une API REST** *(structures des services "web" exposés)* **en se basant sur une analyse d'un code source comportant des commentaires spéciaux**

Documentation de référence : **http://apidocjs.com/**

1.1. Exemple de commentaire "api-doc" :

```
/**
 * @api {get} /user/:id Request User information
 * @apiName GetUser
 * @apiGroup User
 *
 * @apiParam {Number} id Users unique ID.
 *
 * @apiSuccess {String} firstname Firstname of the User.
 * @apiSuccess {String} lastname  Lastname of the User.
 */
```

(à placer au dessus d'une route express)

1.2. Installation de apidoc

```
npm install -g apidoc
```

1.3. Utilisation directe en ligne de commande

apidoc **-h** (help)

```
apidoc -i myapp/ -o apidoc/ [-t mytemplate/ ]
```

(**i**nput dir) (**o**utput dir) (**t**emplate dir)

NB : **un template par défaut est fourni par apidoc** *(--> option -t pas indispensable)*

1.4. configuration de apidoc

apidoc.json (ou bien partie apidoc :{ ...} de package.json)

```
{
  "name": "example",
  "version": "0.1.0",
  "description": "api for ...",
  "title": "My Xyz Api ",
  "url" : "https://api.github.com/v1"
}
```

1.5. Utilisation indirecte via grunt ou gulp

via grunt :

```
npm install grunt-apidoc –save-dev
```

doc de référence : github.com/apidoc/grunt-apidoc

via gulp :

```
npm install --save-dev gulp-apidoc
```

doc de référence :https://www.npmjs.com/package/gulp-apidoc

exemple :

```
var gulp = require('gulp'),
    apidoc = require('gulp-apidoc');

gulp.task('apidoc', function(done){
          apidoc({
            src: "example/",
            dest: "build/"
          },done);
});
```

1.6. Quelques syntaxes et exemples

```
/**
 * @apidefine DeviseStructure
 * @apiSuccess {String} devise.code code of Devise (ex: EUR , USD, GBP , JPY, ...)
 * @apiSuccess {String} devise.nom  name of Devise (ex: euro , dollar , livre , yen)
 * @apiSuccess {Number} devise.change  change for 1 euro.
 */
```

```
/**
* @api {get} /devise/:code Request Devise values by code
* @apiName GetDeviseByCode
* @apiGroup Devise
*
* @apiParam {String} code unique code of Devise (ex: EUR , USD, GBP , JPY)
*
* @apiSuccess {Object} devise devise values as json string
* @apiUse DeviseStructure
* @apiSuccessExample {json} Success
* HTTP/1.1 200 OK
* {"code":"EUR","nom":"euro","change":1}
* @apiErrorExample {json} List error
* HTTP/1.1 500 Internal Server Error
* HTTP/1.1 404 Not Found Error
*/
apiRouter.route('/devise/:code')
.get(....) ;
```

Devise - Request Devise values by code

GET

```
http://localhost:8282/devise/:code
```

Paramètre

Champ	Type	Description
code	String	unique code of Devise (ex: EUR , USD, GBP , JPY)

Success 200

Champ	Type	Description
devise	Object	devise values as json string
code	String	code of Devise (ex: EUR , USD, GBP , JPY, ...)
nom	String	name of Devise (ex: euro , dollar , livre , yen)
change	Number	change for 1 euro.

Success

```
HTTP/1.1 200 OK
{"code":"EUR","nom":"euro","change":1}
```

List error

```
HTTP/1.1 500 Internal Server Error
HTTP/1.1 404 Not Found Error
```

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

1. GRUNT

1.1. Grunt (Javascript Task Runner)

"**Grunt**" s'appuyant lui même sur npm pour le téléchargement/installation , permet d'**automatiser des tâches de développement répétitives** telles que :
- **nettoyage** de certains répertoires et fichiers temporaires (via *grunt-contrib-clean*)
- **analyse/vérification** du code javascript (via *grunt-contrib-jshint*)
- éventuelle **pré-compilations** (ex : scss → css ,
  ts (typeScript/anagular2) → js ,
  …) via *grunt-typescript* , ...
- éventuelle concaténation de fichiers javascript (via *grunt-contrib-concat*)
- éventuelle **compression** (*minification*) du code (via *grunt-contrib-uglify ou ...)*
- éventuels déclenchements automatiques dès qu'un fichier a changé *(via grunt-contrib-watch)*
- **lancement de tests** *(via grunt-karma* ou *....)*

la configuration de "**Grunt**" s'effectue via un fichier javascript de configuration (fonction de configuration javascript pilotant des modules "**npm/nodeJs**" et avec données/paramètres de syntaxe proche JSON) .

1.2. Configuration et utilisation de Grunt

Installation (en mode global) du lanceur de Grunt en ligne de commande :

npm install -g grunt-cli

Installation (locale au projet) de quelques plugins souvent utiles :

npm install grunt grunt-contrib-uglify grunt-karma --save-dev
ou bien édition de package.json + npm update

Installation (locale au projet) de quelques autres plugins souvent utiles :

fichier ***package.json***

```
{
 "name": "karma-through-grunt",
 "version": "0.0.1",
 "description": "...",
 "main": "...",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "...",
 "devDependencies": {
   "karma-jasmine": "~0.3.6",
```

```
    "karma-chrome-launcher": "~0.2.2",
    "karma-jasmine-html-reporter": "~0.2.0",
    "grunt-contrib-clean" : "~0.7.0" ,
    "grunt-contrib-jshint" : "~0.12.0"
  }
}
```

npm update

Configuration de GRUNT via le fichier **gruntfile.js** :

```
module.exports = function(grunt) {
  // Configuration de Grunt
  grunt.initConfig({

          //NB: npm update (after change dependencies in package.json)
          pkg: grunt.file.readJSON('package.json') ,

          clean: {
                // Deletes all old .js files, but skips old min.js files
                js: ["path/to/dir/old/*.js", "!path/to/dir/old/*.min.js"] ,

                // delete all files and directories here
                build: ["build/xx/yy", "dist" , "dest"],
          } ,

          jshint: {
                        all: ['gruntfile.js', 'js/**/*.js', 'test/**/*.js']
                        // options in .jshintrc file
                        },

           uglify: {
                        my_target: {
                          files: [
                                { src: 'js/*js', dest: 'dest/common.js'},
                                { src: 'test/unit/*js', dest: 'dest/test.js'}
                        ]
          }
      } ,

      karma: {
         unit: {
            configFile: 'karma.conf.js'
         }
      }
  });

  // A very basic logging task :
  grunt.registerTask('basic_log', '', function() {
    grunt.log.write('Logging some stuff...').ok();
  });

  // Définition des tâches/plugins Grunt :
```

```
  grunt.loadNpmTasks('grunt-contrib-clean');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-karma');

  grunt.registerTask('default', [ 'basic_log' , 'clean' , 'jshint' ,'uglify', 'karma']);
};
```

Exemple de contenu du fichier .jshintrc

```
{  "curly": true,
  "eqnull": true,
  "eqeqeq": true,
  "undef": true,
  "globals": {
    "jQuery": true
  }
}
```

Lancement de GRUNT (depuis le répertoire contenant gruntfile.js) :

grunt

2. GULP

Ancien exemple (pour angular 2) :

gulpfile.js

```
const gulp = require("gulp");
const del = require("del");
const tsc = require('gulp-typescript');
const sourcemaps = require('gulp-sourcemaps');
const ignore = require('gulp-ignore');
const tscProject = tsc.createProject("tsconfig.json");

gulp.task('default', ['build']);

gulp.task('build', ['copy_resources','compile','copy-libs' ], function(){
    console.log("Building the project ...");
});

gulp.task('clean', function () {
```

```
  return del('dist/**/*');
});

// TypeScript compile (mode "dev" avec sourcemaps )
gulp.task('compile', ['tslint'], function () {
    var tsResult =
    gulp.src('src/**/*.ts')
    //tscProject.src() //prend en compte l'option "rootDir" de tsconfig.json
    .pipe(sourcemaps.init())
    .pipe(tscProject(/*tsc.reporter.defaultReporter()*/)) ;
    //.pipe(gulp.dest('dist')); //si pas suite avec sourcemaps.write

    return tsResult.js //.js en plus depuis la v3
        .pipe(sourcemaps.write(".", {sourceRoot: '/src'}))
        .pipe(gulp.dest('dist'));
});



//Copy all resources that are not TypeScript files into build/dist directory (.html , .css , .json, ...)
gulp.task('copy_resources', function (){
    return gulp.src(["src/**/*", "!**/*.ts"])
          .pipe(gulp.dest("dist"));
});

gulp.task("copy-libs",[ ] ,  function (){
    return gulp.src([
            'reflect-metadata/Reflect.js',
            'zone.js/dist/zone.js',
            'core-js/client/shim.min.js',
            'systemjs/dist/system-polyfills.js',
            'systemjs/dist/system.src.js',
            'rxjs/**',
            '@angular/core/bundles/core.umd.js',
            '@angular/common/bundles/common.umd.js',
            '@angular/compiler/bundles/compiler.umd.js',
            '@angular/platform-browser/bundles/platform-browser.umd.js',
```

```
        '@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',
        '@angular/http/bundles/http.umd.js',
        '@angular/router/bundles/router.umd.js',
        '@angular/forms/bundles/forms.umd.js',
        '@angular/upgrade/bundles/upgrade.umd.js'
      ], {cwd: "node_modules/**"}) /* Glob required here. */
      .pipe(ignore.exclude([ "**/*.map" , "**/*.ts" , "**/*.txt" , "**/*.md" , "**/*.json"]))
      .pipe(gulp.dest("dist/lib"));
});
```

Voici finalement la configuration "npm" nécessaire pour "gulp":

package.json

```
…
"scripts": {
…
 "build": "gulp clean && gulp build"
},
…
 "devDependencies": {
    ...
    "del": "^2.2.0",
    "gulp": "^3.9.1",
    "gulp-sourcemaps": "^2.2.0",
    "gulp-typescript": "^3.1.2",
    "gulp-ignore": "^2.0.2",
   ...
  },
...
```

Commande à lancer (dans un terminal depuis la racine du projet) :

une seule fois :

npm i -g gulp

npm i -g gulp-cli

npm install

régulièrement :

gulp clean

gulp build

…

ou bien indirectement

npm run build

XII - Annexe – WEB Services "REST"

1. Généralités sur Web-Services REST

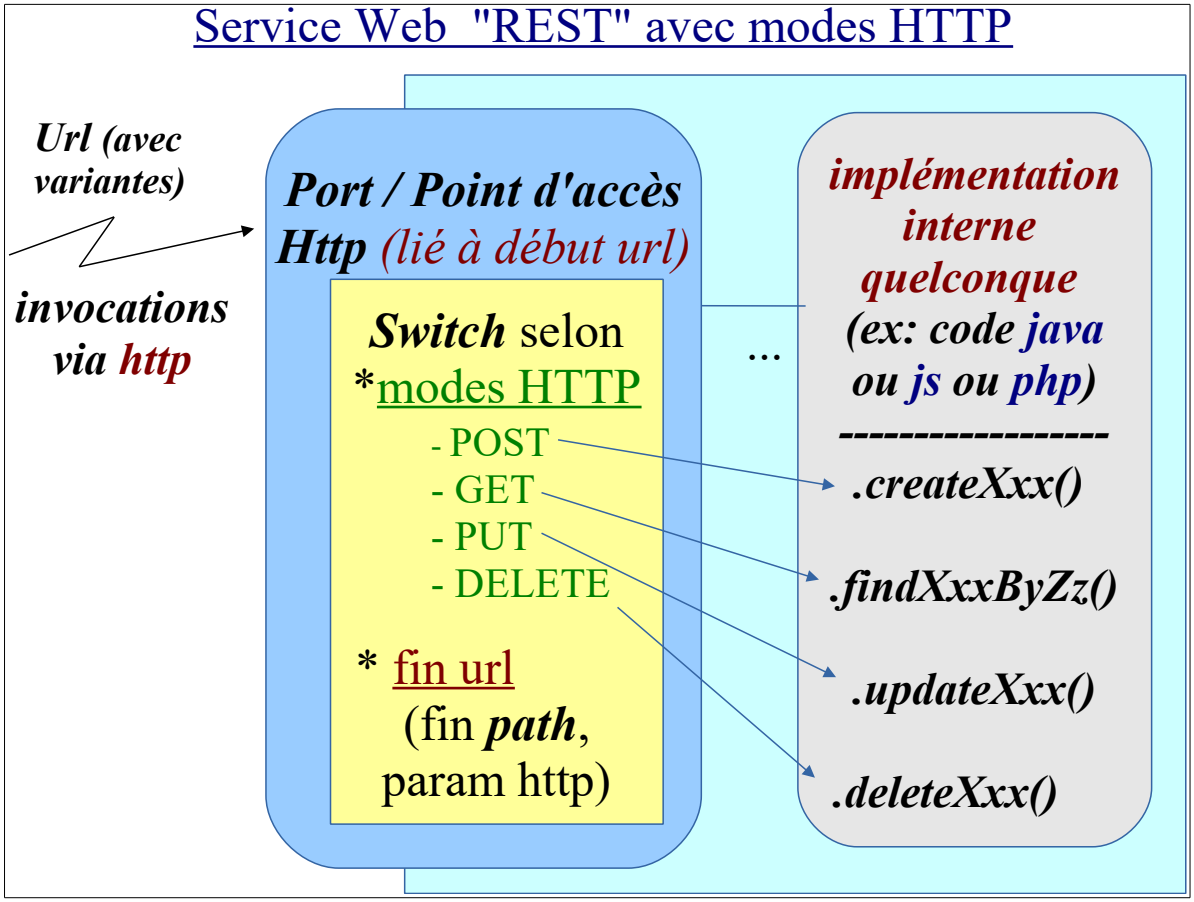
2 grands types de services WEB: **SOAP/XML** et **REST/HTTP**

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 **Re**presentational **S**tate **T**ransfert.
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».*

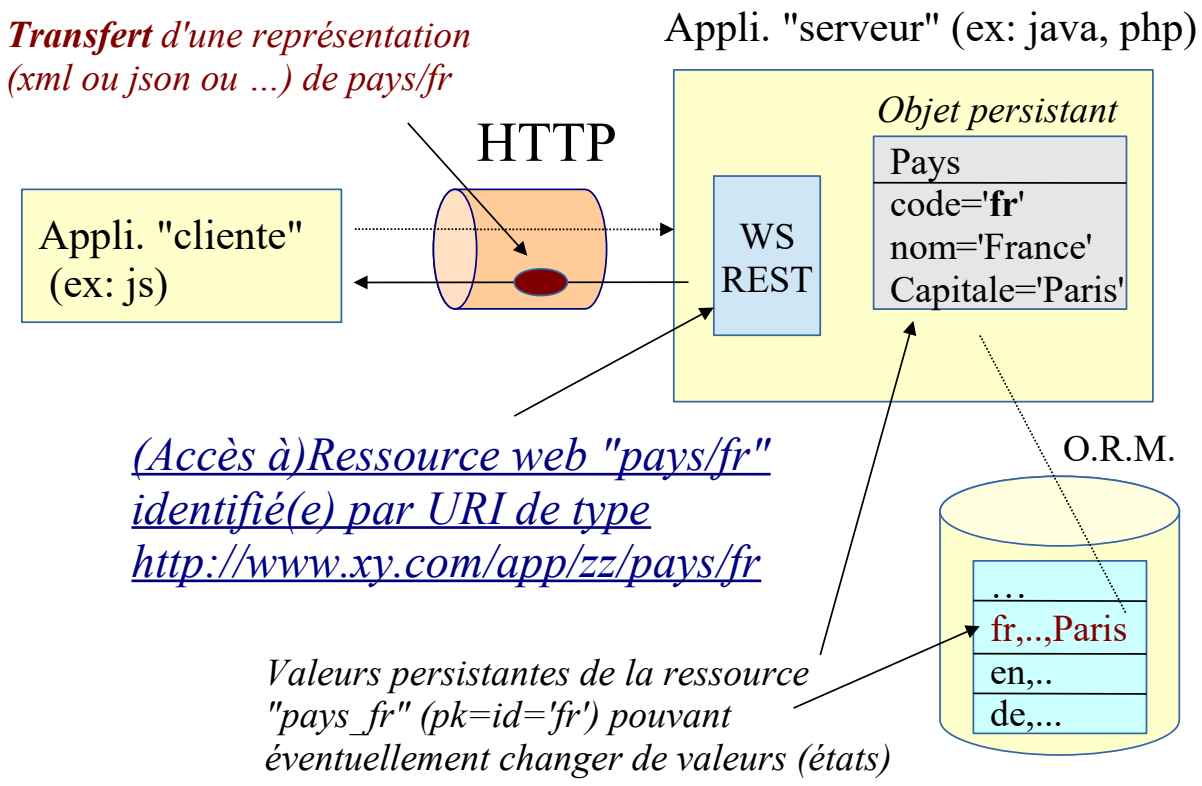
L'information de base, dans une architecture REST, est appelée **ressource**.
Toute information (à sémantique stable) qui peut être nommée est une ressource: un article , une photo, une personne , un service ou n'importe quel concept.

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")

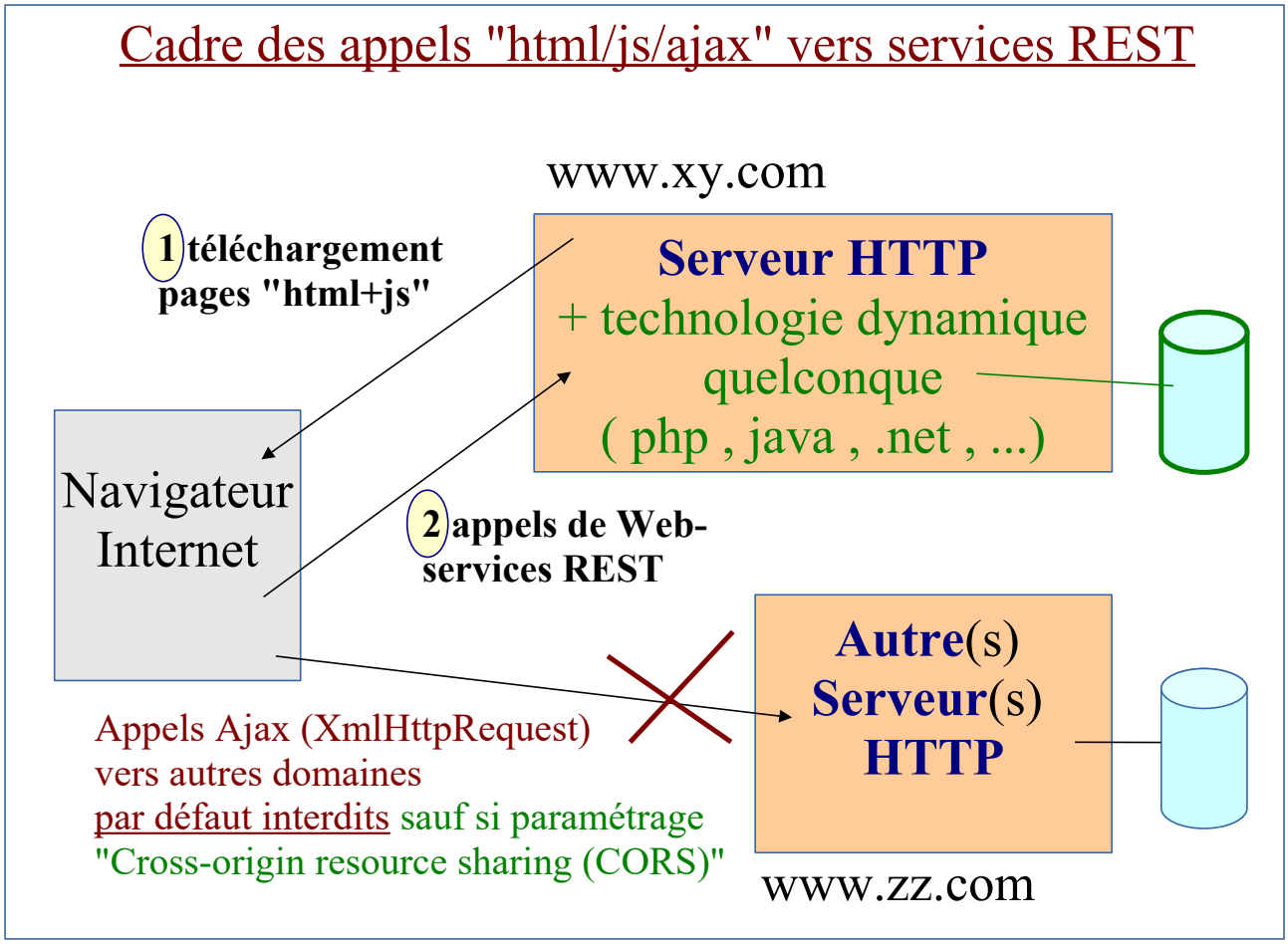
http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417
http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417

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

Paramétrage "CORS" avec Spring-mvc

```
import org.springframework.web.bind.annotation.CrossOrigin;
...
@RestController
@CrossOrigin(origins = "*")
//@CrossOrigin(origins = { "http://localhost:4200" ,
//                         "http://www.partenaire-particulier.com" })
@RequestMapping(value="/rest/products" , headers="Accept=application/json")
public class ProductCtrl {...
}
```

*et/ou implémentation de **WebMvcConfigurerAdapter***
comportant une redéfinition de public void **addCorsMappings**(CorsRegistry registry) .

XIII - Annexe – Sécurité - WEB Services "REST"

1. Api Key

Un web service hébergé par une entreprise et rendu accessible sur internet a un certain coût de fonctionnement (courant électrique , serveurs , ...) .

Pour limiter des abus (ex : appel en boucle) ou bien pour obtenir un paiement en contre partie d'une bonne qualité de service , un web service public est souvent invocable que si l'on renseigne une "api_key" (au niveau de l'URL ou bien au niveau de l'entête la requête HTTP).

Une "**api_key**" est très souvent de type "**uuid/guid**" .

Critères d'une api_key :

- lié à un abonnement (gratuit ou payant) , ex : compte utilisateur / compte d'entreprise
- ne doit idéalement pas être diffusé (à garder secret)
- souvent lié à un compteur d'invocations (limite selon prix d'abonnement)
- doit pouvoir être administré (régénéré si perdu/volé , ...)
  et les modifications doivent pouvoir être immédiatement ou rapidement prises en compte.

Exemple :

Le site **https://fixer.io** héberge un web service REST permettant de récupérer les taux de change (valeurs de *"USD"* , *"GBP"* , *"JPY"* , ... vis à vis de "*EUR*" par défaut).

Début 2018, ce web service était directement invocable sans "api_key" .

Courant 2018, ce web service est maintenant invocable qu'avec une "**api_key**" **liée à un compte utilisateur** *"gratuit" ou bien "payant"* selon le mode d'abonnement (options, fréquence d'invocation, ...).

URL d'appel sans "api_key" : **http://data.fixer.io/api/latest**

Réponse :

```
{
"success":false,
"error":{"code":101,"type":"missing_access_key",
        "info":"You have not supplied an API Access Key. [Required format:
                access_key=YOUR_ACCESS_KEY]"
    }
```

}

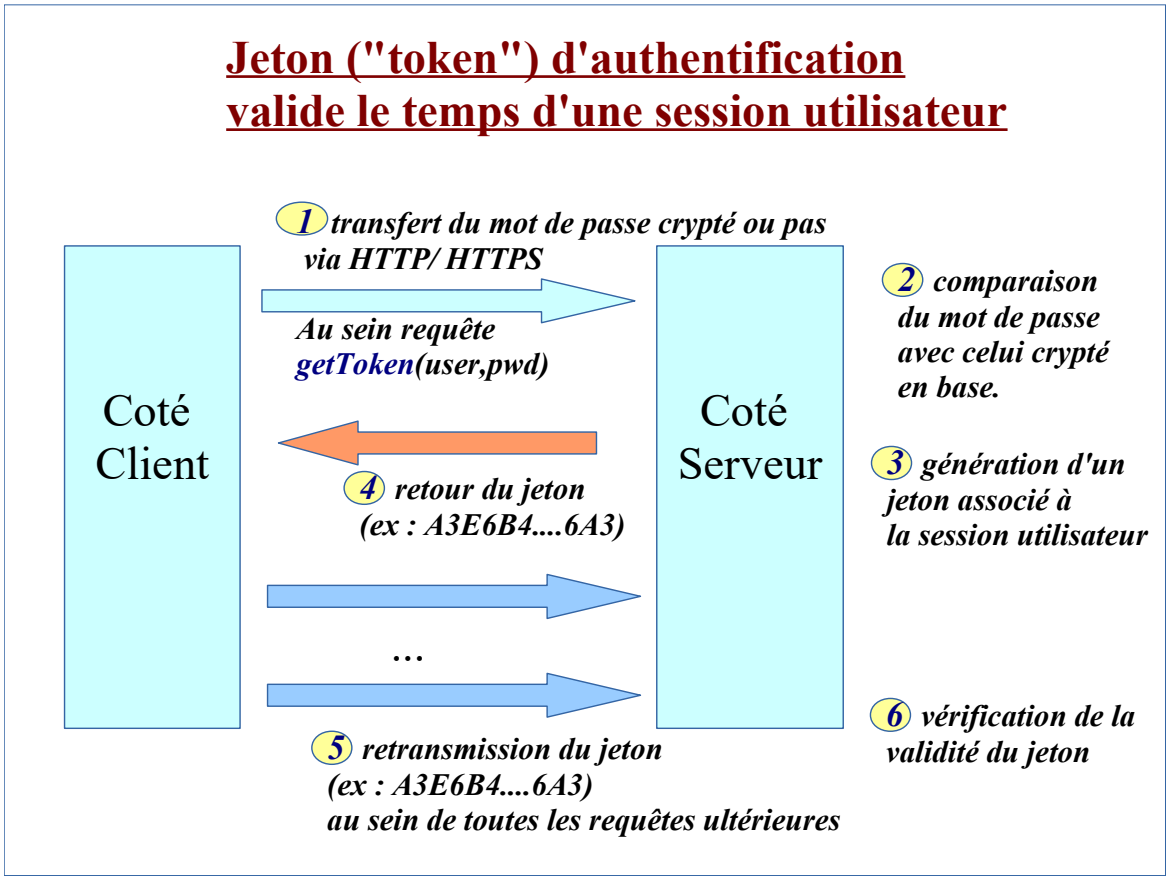
URL d'invocation avec api_key valide :

http://data.fixer.io/api/latest?access_key=*26ca93ee7.....aaa27cab235*

{
"success":true, "timestamp":1538984646, "base":"EUR", "date":"2018-10-08",
"rates":
{"AED":4.224369,...,"DKK":7.460075,"DOP":57.311592,"DZD":136.091172,"EGP":20.596249,
"ERN":17.250477,"ETB":31.695652,"EUR":1,"FJD":2.46956,"FKP":0.88584,"GBP":0.879667,.
...,"JPY":130.858498,....,"USD":1.15005,...,"ZWL":370.724343}
}

2. Token d'authentification

2.1. Tokens : notions et principes



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.

2.2. Bearer Token (au porteur) / normalisé HTTP

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 …***

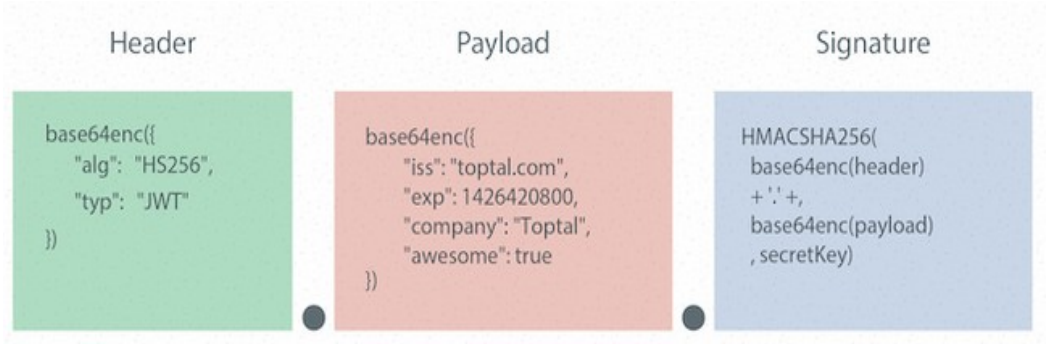
Le terme anglais "**Bearer**" signifiant "au **porteur**" en français indique que la simple possession d'un jeton valide par une application cliente devrait normalement , après transmission HTTP, permettre au serveur d'autoriser le traitement d'une requête (après vérification de l'existence du jeton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

2.3. JWT (Json Web Token)

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" , ...)

XIV - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP