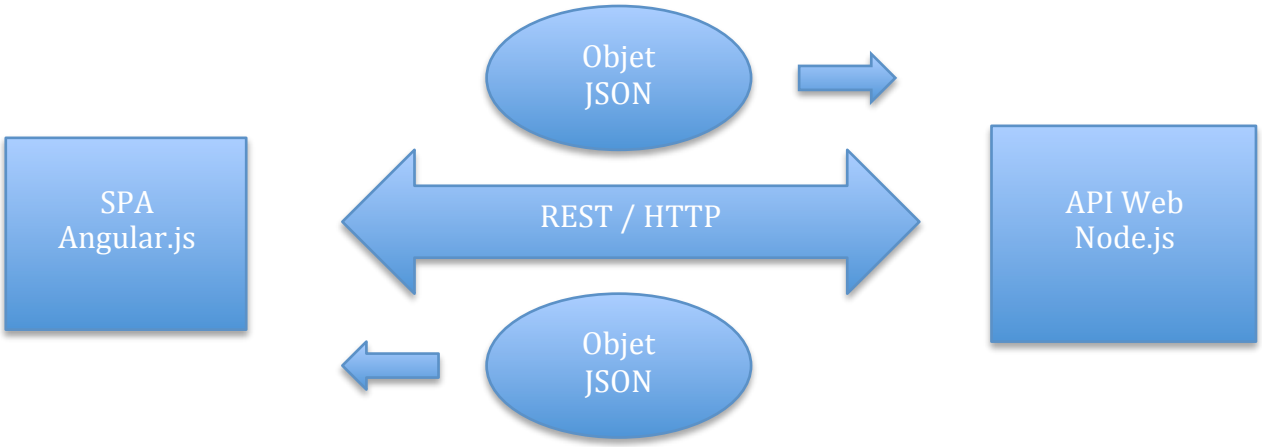
Gestion à distance de comptes bancaires

But du TP : écrire une application Client/Serveur avec Angular.js et Node.js permettant de gérer des comptes bancaires. Un serveur basé sur Node gérera les comptes bancaires et permettra à des clients de se connecter et d'effectuer les opérations suivantes :

- Créer un compte en banque.
- Consulter la position d'un compte.
- Ajouter une somme sur un compte.
- Retirer une somme d'un compte.

La communication entre le Client Angular et le Serveur Node utilisera HTTP en mode REST et consistera en l'échange d'objets au format JSON.



I. Préparation du TP :

Récupérez le module `banque.js` gérant l'ensemble de comptes. Banque.js contient la partie métier du serveur.

En salle de TP à l'UPS, node devrait être préinstallé. Vous pouvez passer directement à la partie I.4.
Si vous êtes sous linux, il est probable, que vous deviez suivre les étapes suivantes.

1) Installez node.js (depuis nodejs.org et en choisissant la version LTS et Linux Binaries -32 bits pour les salles de TP-).
   a) Décompresser le fichier **OÙ `xxxx` est la version de node** :

```
xz –d node-vXXXX-linux-x86.tar.xz
```

Attention : dans les salles de TP il faut impérativement décompresser, désarchiver node et travailler en dehors du répertoire Documents qui est un répertoire partagé avec Windows et qui ne gère donc pas les spécificités de Linux (liens symboliques notamment).
Par contre, on vous conseille de bien sauvegarder les fichiers sur lesquels vous travaillez dans ce répertoire (ou sur une clé USB) car les répertoires utilisateurs sont spécifiques à une machine donnée.
Attention bis (quelque soit la machine sur laquelle vous travaillez) : On vous conseille aussi de ne créer que des répertoires sans espace ou caractères spéciaux qui peuvent poser problème.

   b) Désarchiver :

```
tar xvf node-vXXXX-linux-x86.tar
```

c) Placer node dans votre répertoire principal :

```
mv node-vXXXX-linux-x86 ~/node
```

2) Pour utiliser plus facilement node, modifier le fichier `~/.bashrc` et ajouter à la fin :

```
// contenu .bashrc
…
export PATH=~/node/bin:$PATH
```

3) Pour ré-interpréter votre .bashrc :

```
. .bashrc
```

4) Préparation des répertoires du projet

Notre application se compose d'un serveur node qui va jouer le rôle :

- de serveur d'API REST (pour permettre à l'applications cliente d'interagir avec la partie métier) ;
- de serveur web « classique » pour permettre à un utilisateur d'accéder à l'application cliente (accéder aux pages web, angular, bootstrap… sans passer la file://...)

a) Créer un répertoire pour le TP (par exemple ici Banque) et se placer dans ce répertoire. Ce répertoire contiendra TOUTE l'applications (partie serveur ET partie cliente)

```
mkdir Banque
cd Banque
```

b) Créer un sous-répertoire dans Banque pour contenir l'applications cliente (par exemple ici public) :

NE PAS SE PLACER DANS CE SOUS REPERTOIRE

```
mkdir public
```

5) Récupérer les cadriciels utilisés dans le projet

ATTENTION : Faire attention d'être bien positionné dans le répertoire du projet (« Banque » par exemple et non pas dans « public » ou ailleurs…)

a) Installer express (cadriciel permettant de créer des applications web côté serveur avec Node.js)

```
npm install express
```

b) Installer le plugin `body-parser` pour express

Dans son utilisation, le client (l'application Angular) enverra/recevra des objets au format JSON à Node.

Notre métier gère des objets (format JSON aussi).

Pour les échanges (via http), nous devons utiliser des chaines de caractères. Il conviendra donc de transtyper ces objets pour pouvoir les échanger. Nous pouvons le faire « à la main » ou utiliser un module de Node (body-parser) pour le faire à notre place.

Pour la seconde solution, utilisera le module `body-parser` :

```
npm install body-parser
```

6) Premier test. Nous allons tester notre installation avec un code Express basique. Créer un fichier nommé **testxp.js** contenant le code suivant :

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

var app = express() ;
app.get('/', function (req, res) {
    res.send('Hello World!') ;
}) ;

app.listen(3000, function () {
    console.log('Example app listening on port 3000!') ;
}) ;
```

a) Lancement :

```
node main
```

Pour visualiser la page il faudra taper dans un navigateur : http://localhost:3000/

En cas de réussite :

- le navigateur affiche « Hello World ! »
- la console node dès le lancement du programme « `Example app listening on port 3000!` ».

II. Développement de la partie serveur :

La partie serveur sera entièrement gérée par une application Node.js. Cette application se compose de deux parties : métier et configuation/api. Le fichier Banque.js contient la partie métier. La seconde partie est à développer.

La seconde partie (configuration/api) joue deux rôles :

- Main : analogue aux autres langages de développement, cette partie initialise l'application et configure les différents modules utilisés. Par exemple express.
- Api : Cette partie va contenir l'ensembles des routes que le client pourra utiliser pour interagir avec le serveur Node.js.

Toute cette seconde partie peut se faire dans un seul fichier que nous nommerons (c'est un choix) Main.js. **Par la suite nous pourrons exécuter notre application par un simple « node main » (à la condition d'être positionné dans le répertoire « Banque »).**

a) Créer un fichier nommé « Main.js » avec votre éditeur js. Ce fichier sera situé dans le répertoire « Banque »

2) Configurer l'application serveur RESTful en utilisant Express.js (Il peut être utile de revoir le cours REST ;))

a) Récupération des modules nécessaires à l'application :

Pour cette première application Express, nous avons besoin d'Express, de Body-Parser et du module Banque :

```
var express = require('express') ; // import module express
var bodyParser = require('body-parser'); // import module body-parser
var banque = require('./banque'); // import module banque
```

b) Instanciation d'express.
Pour pouvoir fonctionner, express doit être instancié :

```
var app = express() ;
```

c) Déclaration d'utilisation de body-parser à express.
Express a besoin de savoir que body-parser peut être employé par la suite. Il faut donc lui indiquer :
A faire juste après la création de l'application app :

```
app.use(bodyParser.json()); // inclusion du plugin pour parser du JSON
```

d) Ajout du service des pages FrontEnd
Le serveur web intégré à Express peut héberger un site web (dont l'application Angular utilisée par le client). Nous avons décidé plus tôt que notre application cliente se situait dans le sous-répertoire public. Le principe est de demander à express de faire la chose suivante :
Chemin virtuel (exposé via les URLs) = chemin physique à exposer

Ainsi, pour demander à express de la servir aux navigateurs des clients, il faut ajouter **avant toutes les routes** :

```
// …/banque/public (physique) = / (url)
app.use(express.static(__dirname + '/public'));

// on choisit maintenant d'exposer un 2eme repertoire
// …/banque/bower_components (physique) = /bower_components (url)

app.use('/bower_components',
          express.static(__dirname + '/bower_components'));
```

Vous êtes libres de rajouter des routes statiques selon vos besoins.

e) Demande de lancement du serveur Web intégré et ouverture du service
A la fin du code, il reste à demander à express de démarrer son service (app.listen), sur le port de notre choix (ici 3000) et éventuellement d'afficher un message sur la console quand le service est prêt à fonctionner :

```
app.listen(3000, function () {
    console.log('Example app listening on port 3000!') ;
}) ;
```

3) Définition des routes & Utilisation du BackOffice
Modifiez votre code pour faire le serveur de comptes bancaires en mode RESTful (POST pour la création, GET pour la consultation, PUT pour les modifications des comptes).

Pour tester les différentes requêtes, nous vous conseillons d'utiliser un plugin pour votre navigateur :
- Restlet client pour Chrome : https://restlet.com/modules/client/
- Restclient pour Firefox : https://addons.mozilla.org/fr/firefox/addon/restclient/

a) Définition d'une route et récupération de paramètres d'URL
Dans notre application, pour les méthodes `GET` et `PUT` il convient d'identifier le compte que l'on veut consulter ou modifier (c'est le `:id` ci-dessous).
A l'intérieur d'une fonction associée à une route, on peut y accéder via `req.params.id`

```
// Lorsqu'un client fait appel à GET pour l'URL /compte/xxx
// où xxx peut être n'importe quoi
app.get('/compte/:id', function(req, res)
 {
   // affiche la valeur du paramètre id sur la console de node
   console.log(req.params.id) ;

   // envoie la valeur du paramètre id au navigateur du client ayant fait
   // l'appel
   res.send(req.params.id) ;
 }) ;
```

b) Sans body-parser :

Une fois les opérations réalisées sur les comptes, il convient de renvoyer à l'application Angular un objet au format JSON. Sans body-parser, pour bien envoyer du JSON, il faudra envoyer une chaine de caractères représentant l'objet JSON dans le champs Body (exemple : « { "id" : 1, "somme" : 100 } ») et préciser « Content-Type: application/json » comme entête (Header). Par exemple :

```
// exemple :
app.get('/compte/:id', function (req, res) {
   res.set('Content-Type', 'application/json');
   res.send('{ "id" : 1, "somme" : 100 } ') ;
}) ;
```

c) Avec body-parser :

Renvoi du JSON

Une fois les opérations réalisées sur les comptes, il convient de renvoyer à l'application Angular un objet au format JSON comme ceci :

```
// Envoi d'un objet
res.json(objet);

// Envoi avec code http personnalisé :
res.status(404).json(
   { error: "Le compte d'id "+req.body.id+" n'existe pas." }
);
```

III. Configuration de l'outillage pour la partie cliente :

Dans les TPs précédents (Calculatrice), nous avons dû gérer manuellement les dépendances de nos fichiers HTML que cela soit pour les CSS (bootstrap.min.css) ou Angular (angular.min.js par exemple). Nous avons utiliser les CDN pour nous simplifier la vie. Toutefois, cette solution n'était pas pérenne.
Pour autant, dans une SPA d'envergure disposant de dizaines de partielles par exemple, il est exclu de gérer à la main ces aspects.
De nombreux outils existent. Nous vous proposons une introduction à quelques outils pour vous simplifier cette gestion.

Bower

Bower est un gestionnaire de paquets orienté web. Il permet ainsi d'une ligne de récupérer, mettre à jour ou supprimer les paquets utilisés dans notre applications cliente. Par paquet, on peut citer Bootstrap, Angular…

Lors de son utilisation, on utilise Bower dans le répertoire du projet. Dans notre cas, ici, il s'agit du répertoire public. Bower va créer un répertoire nommé bower_components dans public et y placer les paquets téléchargés. Il n'est nul besoin de modifier ce répertoire.

L'installation de bower s'effectue via npm :

```
npm install -g bower
```

Bower s'utilise de la sorte : bower install nom_paquetage

```
bower install nom_paquetage
```

IV. Développement de la partie cliente :

La partie cliente sera réalisée par une SPA en utilisant Angular :

La consommation des services web REST utilisera le service `$resource` fourni par le module `ngResource` comme présenté dans le cours Angular.