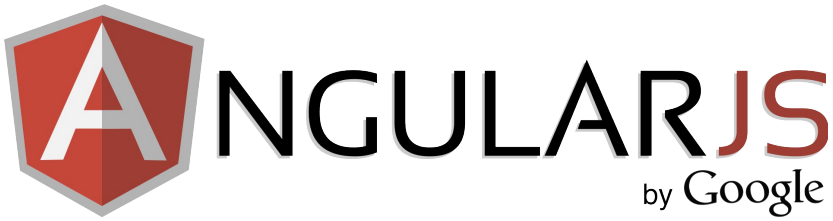
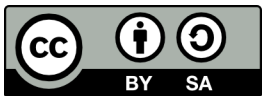
Créer une «application web »
avec
AngularJS, nodeJS et mongoDB

v110813

Brouillon





Notes de l'auteur

Ce document est à l'origine destiné aux élèves de seconde, du lycée G Fichet de Bonneville (Haute-Savoie), qui ont choisi de suivre l'enseignement d'exploration PSN (Pratique Scientifique et Numérique). Il est aussi utilisable en spécialité ISN. Dans tous les cas, des connaissances en JavaScript, HTML et CSS sont indispensables avant d'aborder AngularJS et nodeJS (voir, par exemple, les 19 activités « JavaScript, HTML et CSS » que j'ai rédigées pour les élèves). Ce document n'a pas la prétention d'être la « bible » d'AngularJS, de nodeJS ou de mongoDB (n'oubliez pas qu'il s'adresse à des élèves ayant une pratique limitée de la programmation), il a été conçu pour travailler « en collaboration » avec l'enseignant (certaines parties ne sont pas détaillées, les enseignants devront adapter leurs explications en fonction des contraintes locales (OS installé, possibilité d'installer des logiciels sur les machines.....)).

Les premières pages de la deuxième partie abordent des sujets relativement théoriques (client-serveur, protocole HTTP...) ce qui va à l'encontre de l'esprit de ce document. Cependant cette digression me semble indispensable vu les sujets abordés dans cette seconde partie.

David Roche

1re partie : les bases d'AngularJS

AngularJS ? Qu'est ce que c'est ?

AngularJS (site officiel) est un framework JavaScript open-source (rechercher la définition d'open-source et de framework) développé par Google. Il a été créé par Miško Hevery (en 2010). La version 1.0 (première version dite stable) date de juin 2012.
Ce framework facilite grandement le développement d'application web selon le modèle MVC (Modèle Vue Controleur). Nous aurons l'occasion de revenir plus tard sur ces notions, mais vous avez peut-être déjà eu l'occasion d'utiliser des applications web (souvent appelées «web app»). Par exemple Gmail et GoogleDoc sont des « web app » (site internet ressemblant à des applications «classiques»).
Notez bien qu'AngularJS n'est pas un "logiciel", vous allez donc produire du code (HTML, JavaScript et CSS).

Première approche

AngularJS va nous permettre de rajouter de nouveaux attributs aux balises HTML, tous ces nouveaux attributs commenceront par : ng-
Par exemple nous pourrons avoir .....
Dans notre 1er exemple nous utiliserons 2 attributs apportés par AngularJS : ng-app et ng-controller, mais il en existe beaucoup d'autres que nous étudierons plus tard dans ce document.

Nous allons pouvoir introduire ce que nous appellerons pour l'instant des « variables » directement dans le code HTML (ces « variables » pourront être contrôlées grâce au JavaScript). Ceci va rendre notre page HTML dynamique (les données affichées pourront évoluer au cours de temps). Ces « variables » seront facilement reconnaissables, car elles seront encadrées par des accolades : {{maVariable}}
exemple : <h1>{{monTitre}}</h1>.
Le navigateur n'affichera pas {{monTitre}}, mais la valeur contenue dans la « variable » monTitre.

Un premier exemple

À faire vous même

code HTML ex1.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex1.js"></script>
</head>
<body ng-app>
	<h1 ng-controller="monControl">{{maVariable}}</h1>
</body>
</html>
```

code JavaScript ex1.js

```
function monControl($scope){
	$scope.maVariable="Hello World !";
}
```

Quelques remarques sur le contenu de la balise head du fichier HTML :

- Avant de commencer vous devez télécharger le framework sur le site officiel (http://angularjs.org/), choisir « stable », « Minified » et cliquez sur Download. Récupérer le fichier « `angular.min.js` ».
- La ligne «`<script src="lib/angular.min.js"></script>`» va vous permettre d'utiliser AngularJS. À vous de l'adapter en fonction du dossier qui accueille le fichier « `angular.min.js` ».
- La ligne suivante («`<script src="javascript/ex1.js"></script>`») va nous permettre d'accéder à notre fichier JavaScript «`ex1.js`», ici aussi, attention au chemin choisi.

Vous avez sans doute remarqué l'attribut ng-app au niveau de la balise body. Cela signifie simplement qu'AngularJS sera actif de la balise <body> jusqu'à la balise </body>.

L'attribut ng-controller dans la balise <h1> va nous permettre de définir un contrôleur. Ce contrôleur sera actif de la balise <h1> jusqu'à la balise </h1>. Tout ce qui se trouve entre les balises <h1> et </h1> sera soumis à la

« surveillance » du contrôleur «monControl», dans cette « zone », c'est lui le « patron », c'est lui le chef d'orchestre.
Qu'est-ce qu'un contrôleur dans AngularJS ?
Ce contrôleur correspond à une fonction JavaScript présente dans le fichier « ex1.js » (d'où l'intérêt de la ligne « `<script src="javascript/ex1.js"></script>` »). Analysons cette fonction :
La fonction «`monControl`» possède un argument : «`$scope`». Cet argument est fondamental.
Un des principes de base (et une des forces) d'AngularJS se nomme le data-binding, qu'est-ce que le data-binding ? C'est la capacité à échanger des informations (des données) entre la partie HTML et la partie JavaScript. Cet échange de données peut se faire dans les 2 sens : du HTML vers le JavaScript et du JavaScript vers le HTML.
Pour « transporter » ces données, AngularJS utilise un objet (au sens informatique du terme, si nécessaire revoir l'activité 14). Cet objet est l'argument de la fonction «`monControl`», c'est-à-dire «`$scope`».

Analysons maintenant l'unique ligne qui compose notre fonction «`monControl`» (notre contrôleur) :
```
$scope.maVariable="Hello World !";
```

Nous avons {{maVariable}} du côté HTML et $scope.maVariable du côté JavaScript, nous pourrons considérer ces 2 entités comme identiques. Comme dit plus haut, l'objet $scope n'arrête pas de faire des « aller-retour » entre le HTML et le JavaScript. Comme tous les objets, il possède des propriétés, maVariable est une de ses propriétés d'où la notation pointée : « `$scope.maVariable` » .
Si vous avez un peu de mal avec tout cela, pour simplifier les choses, dites-vous que si l'on a {{maVariable}} du côté HTML on devra avoir $scope.maVariable du côté JavaScript et que tout changement au niveau $scope.maVariable entraînera un changement au niveau de {{maVariable}} (et vis versa dans d'autres situations).

En vous aidant des explications qui viennent d'être fournies, décrire le résultat attendu si vous ouvrez le fichier ex1.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....
Vérifiez votre hypothèse

Manipulons des objets

Il est possible de manipuler des objets avec AngularJS

À faire vous même

code HTML ex2.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex2.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<h1>Fiche d'identité n°{{infoPerso.id}}</h1>
	<p>Nom : {{infoPerso.nom}}</p>
	<p>Prénom : {{infoPerso.prenom}}</p>
	<p>Date de Naissance : {{infoPerso.dateNaissance}}</p>
</body>
</html>
```

code JavaScript ex2.js
```
maFiche={
	id : "7845",
	nom :"Durand",
	prenom :"Jean-Pierre",
	dateNaissance :"17/08/1967"
}

function monControl($scope){
	$scope.infoPerso=maFiche;
}
```

Cet exemple ne devrait vous poser de difficulté. Au lieu d'une « simple » variable, nous manipulons ici un objet (infoPerso côté HTML et $scope.infoPerso côté serveur). Ici aussi, le contrôleur (la fonction monControl) ne comporte qu'une ligne : «`$scope.infoPerso=maFiche;`»
Pour comprendre l'utilité de cette ligne, vous avez juste à savoir que « objet1=objet2 » permet d'attribuer à objet1 toutes les propriétés d'objet2 (méthodes et attributs). Après l'exécution de la ligne «`$scope.infoPerso=maFiche;`», infoPerso se retrouve avec les attributs suivants : id, nom, prenom et date de naissance (les attributs de l'objet maFiche). Ceci explique donc les {{infoPerso.id}}, {{infoPerso.nom}}, {{infoPerso.prenom}} ou encore {{infoPerso.dateNaissance}} du code HTML.
En cas de difficulté, n'hésitez pas à vous replonger dans l'activité 14.

Décrire le résultat attendu en cas « d'ouverture » du fichier ex2.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....
Vérifiez votre hypothèse

utilisation ng-repeat

ng-repeat est une directive AngularJS qui va nous permettre de dupliquer du code HTML automatiquement et surtout intelligemment.

Imaginer que vous voulez afficher une liste de fruit à l'aide des balises HTML <ul> et <li> (si vous ne connaissez pas ces balises recherchez sur internet des informations à leur sujet).
Nous aurions ce genre de code :

```
<ul>
	<li>banane</li>
	<li>pomme</li>
	<li>ananas</li>
	<li>pêche</li>
	<li>fraise</li>
</ul>
```

Cela peut vite devenir rébarbatif....

Grâce à la directive ng-repeat et à un tableau JavaScript contenant cette liste de fruits (mesFruitsTab=["banane", "pomme", "ananas"," pêche","fraise"]), il va être possible d'écrire beaucoup moins de code HTML.

À ***faire vous même***

code HTML ex3.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex3.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<h1>Voici ma liste de fruits</h1>
	<ul>
		<li ng-repeat="fruit in mesFruits">{{fruit}}</li>
	</ul>
</body>
</html>
```

code JavaScript ex3.js

```
var mesFruitsTab=["banane","pomme", "ananas","pêche","fraise"] ;
function monControl($scope){
	$scope.mesFruits=mesFruitsTab;
}
```

NB : il aurait été possible d'écrire directement :
«`$scope.mesFruits=["banane","pomme", "ananas","pêche","fraise"] ;`»
sans passer par l'intermédiaire de «`mesFruitsTab`»

ng-repeat va permettre de répéter la balise <li> autant de fois nécessaire pour que tous les éléments du tableau «mesFruits» soit traités. {{fruit}} sera successivement remplacé par banane, pomme, ananas, pêche et fraise.
Si l'on a :

```
<maBalise ng-repeat="maVariable in monTableau">{{maVariable}}</maBalise>
```

avec `monTableau=[element1,element2,element3]`

Le code HTML généré par AngularJS sera :

```
<maBalise>element1</maBalise>
<maBalise>element2</maBalise>
<maBalise>element3</maBalise>
```

Décrire le résultat attendu en cas « d'ouverture » du fichier ex3.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....

À ***faire vous même : mini-projet 1***

Il est possible d'avoir un tableau contenant des objets :

```
monTableau=[
        {id : "7845",
        nom :"Durand",
        prenom :"Jean-Pierre",
        dateNaissance :"17/08/1967"},
        {id : "6578",
        nom :"Dupond",
        prenom :"Gérard",
        dateNaissance :"23/04/1984"},
        {id : "9876",
        nom :"Robert",
        prenom :"Gabriel",
        dateNaissance :"21/02/1991"}
]
```

Reprenez l'exemple traité dans l'exemple 2 (affichage d'une fiche de renseignements personnels). Sauf que cette fois ci, vous devrez produire du code permettant d'afficher les 3 fiches de renseignements les unes sous les autres :

Fiche d'identité n°7845

Nom : Durand

Prénom : Jean-Pierre

Date de Naissance : 17/08/1967

Fiche d'identité n°6578

Nom : Dupond

Prénom : Gérard

Date de Naissance : 23/04/1984

Fiche d'identité n°9876

Nom : Robert

Prénom : Gabriel

Date de Naissance : 21/02/1991

Des images avec ng-src

Les objets peuvent aussi contenir des URL, notamment des URL d'images.

Voici un autre exemple illustrant la directive ng-src :

À ***faire vous même***

Testez et étudiez attentivement cet exemple

code HTML ex4.html

```
<!doctype html>
<html lang="fr" ng-app>
<head>
	<meta Charest="UTF-8">
	<title>ex4 AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex4.js"></script>
	<link rel="stylesheet" href="css/ex4.css">
</head>
<body ng-controller="monControleur">
	<h1 id="monTitre">Séries en série</h1>
	<div class="row" id="mesSeries" ng-repeat="serie in tabSerie">
		<div class="offset5 span2"><img ng-src="{{serie.urlImage}}"/></div>
		<div class="span5">
			<h2>{{serie.titre}}</h2>
			<p>Titre original : {{serie.titreOr}}</p>
			<p>Créateur(s) : {{serie.createur}}</p>
			<p>Etat : {{serie.etat}}
		</div>
	</div>
</body>
</html>
```

code JavaScript ex4.js

```
monControleur=function($scope){
	$scope.tabSerie=[
		{
			titre:"Le Trône de fer",
			titreOr:"Game of Thrones",
			createur:" David Benioff et D. B. Weiss",
			urlImage:"image/got.jpg",
			etat:"en cours saison 04 diffusée au printemps 2014"},
		{
			titre:"Lost : Les Disparus",
			titreOr:"Lost",
			createur:"J. J. Abrams, Damon Lindelof et Jeffrey Lieber",
			urlImage:"image/lost.jpg",
			etat:"terminée"},
		{
			titre:"Homeland",
			titreOr:"Homeland",
			createur:"Howard Gordon et Alex Gansa",
			urlImage:"image/homeland.jpg",
			etat:"en cours, saison 03 diffusée en septembre 2013"}
	]

}
```

code CSS ex4.css

bootstrap twitter

```
+
/*Ajout pour ex4*/
body{
  background-color:grey;
}
#monTitre{
```

```
  text-align:center;
  margin: 30px;
  font-size: 60px;
}
#mesSeries{
  margin-bottom:20px;
}
```

Attention : le fichier ex4.css est composé du bootstrap twitter et du code ci-dessus.

J'attire votre attention sur la balise image «`<img ng-src="{{serie.urlImage}}"/>` », l'attribut classique src ne fonctionnera pas ici (faites l'essai), il faudra donc systématiquement utiliser ng-src en lieu et place de src.

Autre élément important, l'utilisation du css est quasi indispensable, ne négligez pas cet aspect des choses.

Un peu d'interaction avec ng-click

Grâce à ng-click il est possible de « réagir » au clic de l'utilisateur en appelant la fonction de votre choix. La balise qui aura pour attribut ng-click deviendra donc « cliquable » :
soit <balise ng-click= "mafonction()">.....</balise>, un clic sur le contenu de cette balise entraînera l'exécution de la fonction mafonction().

À ***faire vous même***

code HTML ex5.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex5.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<button ng-click="affichage()">Afficher la liste</button>
	<ul>
		<li ng-repeat="fruit in mesFruits">{{fruit}}</li>
	</ul>
</body>
</html>
```

code JavaScript ex5.js

```
var mesFruitsTab=["banane","pomme", "ananas","pêche","fraise"] ;
function monControl($scope){
	$scope.affichage=function(){
		$scope.mesFruits=mesFruitsTab;
	}
}
```

Décrire le résultat attendu en cas « d'ouverture » du fichier ex5.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....
Vérifiez votre hypothèse

Plusieurs remarques sur cet exemple :

- les fonctions qui seront exécutées en cas de clic devront être des méthodes de l'objet $scope d'où le «`$scope.affichage`»
- nous verrons juste après une autre méthode pour faire apparaître et disparaître des éléments d'une page.
- Nous avons choisi un bouton comme élément « cliquable », toute autre balise peut convenir (div, a, img.....)

ng-show et ng-hide

Il est possible de masquer le contenu d'une balise en utilisant la directive ng-show dans ladite balise :

```
<balise ng-show="isVisible">.....</balise>
```
Si dans le contrôleur (fichier JavaScript) la variable « $scope.isVisible» est égale à true, le contenu de la balise sera visible, si « $scope.isVisible» est égale à false, le contenu de la balise sera invisible.
La directive ng-hide fonctionne exactement de la même façon, mais « à l'envers » (visible si false et invisible si true).
Je précise que le nom choisi «isVisible » est arbitraire, vous pouvez choisir le nom qui vous convient.

À faire vous même

code HTML ex6.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex6.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<button ng-click="affichage()">Cliquez ici</button>
	<ul ng-show="visible">
		<li ng-repeat="fruit in mesFruits">{{fruit}}</li>
	</ul>
</body>
</html>
```

code JavaScript ex6.js

```
var mesFruitsTab=["banane","pomme", "ananas","pêche","fraise"] ;
function monControl($scope){
	$scope.visible=false;
	$scope.mesFruits=mesFruitsTab;
	$scope.affichage=function(){
		if ($scope.visible==false){
			$scope.visible=true;
		}
		else{
			$scope.visible=false;
		}
	}
}
```

Décrire le résultat attendu en cas « d'ouverture » du fichier ex6.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....

la balise input et la directive ng-model

Les sites n'utilisant pas, à un moment ou à un autre, la balise input sont très rares. AngularJS va énormément simplifier la gestion des données entrées par le visiteur du site.
La directive ng-model pourra être associé à une balise input :

```
<input type="text" ng-model="maValeur">
```

Les données entrées par l'utilisateur du site pourront être récupérées dans un contrôleur JavaScript ($scope.maValeur).

À ***faire vous même***

code HTML ex7.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
```

```
	<script src="javascript/ex7.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<h1>Ma liste de fruits</h1>
	<form>
		<p>Ajouter un fruit: <input type="text" ng-model="monFruit"/></p>
		<button ng-click="ajout()">Valider</button>
	</form>
	<ul>
		<li ng-repeat="fruit in mesFruits">{{fruit}}</li>
	</ul>
</body>
</html>
```

code JavaScript ex7.js

```
var mesFruitsTab=[] ;
function monControl($scope){
	$scope.mesFruits=mesFruitsTab;
	$scope.ajout=function(){
		if ($scope.monFruit!=""){
			$scope.mesFruits.push($scope.monFruit) ;
			$scope.monFruit="";
		}
	}
}
```

Décrire le résultat attendu en cas « d'ouverture » du fichier ex6.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....

À *faire vous même : miniprojet 2*

Vous êtes chargé de mettre en place une application de saisie des nouveaux élèves dans un lycée.
Votre application devra proposée un formulaire de saisi avec :

- le nom
- le prénom
- la date de naissance
- la classe (choisir une classe parmi une liste : utilisation de la balise select)

La validation du formulaire devra se faire par l'intermédiaire d'un bouton.

Les élèves nouvellement saisis devront apparaître sur la page au fur et à mesure de la saisie.

Checkbox

Il est possible d'utiliser une checkbox (balise input avec un « type="checkbox" »)

```
<input type="checkbox" ng-model="checked"/>
```

Si la checkbox est cochée, on aura alors checked=true si elle est décochée on aura checked=false (vous pouvez utiliser un autre nom que «checked» pour la variable.

À faire vous même

Inventez un exemple utilisant le type «checkbox» et la directive ng-model qui va avec. Vous êtes bien sûr autorisé à utiliser d'autres éléments de votre choix.

Les filtres

AngularJS prévoit l'application de filtres qui permet de sélectionner ou de mettre en forme l'information. Nous allons ici voir un exemple de l'utilisation des filtres.
En ajoutant un filtre à une directive ng-repeat, seuls les éléments correspondant au filtre seront affichés .

Voici la façon de procéder : `ng-repeat="x in tab | filter:maVariable"`
sans le filtre toutes les valeurs contenues dans la tableau tab seront affichées. Avec le filtre seule la valeur correspondant à maVariable sera affichée (à condition bien sûr que cette valeur se trouve dans le tableau)

À *faire vous même*

code HTML ex8.html

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex8.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<h1>Ma liste de fruits</h1>
	<form>
		<p>rechercher un fruit <input type="text" ng-model="recherche"/></p>
	</form>
	<ul>
		<li ng-repeat="fruit in mesFruits | filter:recherche">{{fruit}}</li>
	</ul>
</body>
</html>
```

code JavaScript ex8.js

```
var mesFruitsTab=["banane","pomme","fraise","kiwi","pêche","cerise","nectarine",
"noix","framboise","noisette","raisin"];
function monControl($scope){
	$scope.mesFruits=mesFruitsTab;
}
```

Décrire le résultat attendu en cas « d'ouverture » du fichier ex6.html avec un navigateur internet (Firefox, Chrome....)
….....
.....
.....

Il existe d'autres filtres, je vous invite à consulter la documentation officielle :
http://docs.angularjs.org/guide/dev_guide.templates.filters

2e partie : nodeJS et AngularJS

notion de client-serveur

Notion de client-serveur

Que se passe-t-il lorsque vous tapez dans la barre d'adresse de votre navigateur «http://www.google.fr» ?

Votre ordinateur (que l'on appellera le client) va chercher à entrer en communication avec un autre ordinateur (que l'on appellera le serveur) se trouvant probablement à des milliers de kilomètres de chez vous. Pour pouvoir établir cette communication, il faut bien sûr que les 2 ordinateurs : le client et le serveur soient « reliés », on dira que nos 2 ordinateurs sont en réseau.

Il existe énormément de réseaux, certains réseaux sont reliés à d'autres réseaux qui sont eux-mêmes reliés à d'autres réseaux.....ce qui forme « des réseaux de réseaux de réseaux..... ». Savez-vous comment on appelle cet assemblage multiple de réseaux ? Internet !

Mon but ici n'est pas de vous expliquez comment font les ordinateurs pour se trouver dans cet « amas de réseaux », si ce sujet vous intéresse, vous rencontrerez certains termes, comme : « serveur DNS », « routeur », « adresse IP », « routage ».....

En tapant «http://www.google.fr», votre machine va chercher à entrer en communication avec le serveur portant le nom «www.google.fr» (en faite c'est plus compliqué, pour les puristes nous dirons donc que la communication va être établie avec le serveur www du domaine google.fr, mais bon, pour la suite nous pourrons nous contenter de l'explication « simplifiée »).

Une fois la liaison établie, le client et le serveur vont échanger des informations en dialoguant :

client : bonjour www.google.fr (ou bonjour www se trouvant dans le domaine google.fr), pourrait tu m'envoyer le code html contenu dans le fichier index.html

serveur : OK, voici le code html demandé

client : je constate que des images, du code css et du code JavaScript sont utilisés, peux-tu me les envoyer

serveur : OK, les voici

Évidemment ce dialogue est très imagé, mais il porte tout de même une part de « vérité ».

J'espère que vous commencez à comprendre les termes client (celui qui demande quelque chose) et serveur (celui qui fournit ce qui a été demandé).

et le HTTP ?

Nous venons de voir que les ordinateurs communiquent entre eux. Pour ce faire, ils utilisent ce que l'on appelle des protocoles.

Selon Wikipedia, dans le cas général, protocole :

On nomme protocole les conventions qui facilitent une communication sans faire directement partie du sujet de la communication elle-même.

En électronique et en informatique (toujours selon Wikipedia) :

un protocole de communication est un ensemble de contraintes permettant d'établir une communication entre deux entités (dans le cas qui nous intéresse 2 ordinateurs)

Pour que la communication soit possible, le client et le serveur doivent avoir des règles communes, ces règles sont définies dans un protocole. Comme vous l'avez sans doute deviné, le protocole de communication employé ici se nomme HTTP.

Le protocole HTTP (HyperText Transfer Protocol) a été inventé par Tim Berners-Lee (1955-....) au début des années 1990. Tim Berners-Lee est aussi à l'origine du langage HTML et des « adresses web ». C'est la combinaison de ces 3 éléments (HTTP, HTML, « adresse web ») que l'on nomme aujourd'hui le « web » (« web » qu'il ne faut pas confondre avec l'internet, même si le web utilise l'internet).

Le HTTP va permettre au client d'effectuer des requêtes à destination d'un serveur. En retour, le serveur va envoyer une réponse.

Voici une version simplifiée de la composition d'une requête :

- la méthode employée pour effectuer la requête
- l'URL de la ressource
- la version du protocole utilisé par le client (souvent HTTP 1.1)
- le navigateur employé (Firefox, Chrome) et sa version
- le type du document demandé (par exemple HTML)

- …..

Certaines de ces lignes sont optionnelles.

Voici un exemple de requête HTTP (la méthode, l'URL et la version du protocole se trouvent sur la première ligne) :

```
GET /mondossier/monFichier.html HTTP/1.1
User-Agent : Mozilla/5.0
Accept : text/html
```

Revenons uniquement sur 2 aspects (si nécessaire nous reviendrons sur les autres plus tard) : la méthode employée et l'URL.

Les méthodes des requêtes HTTP

Une requête HTTP utilise une méthode (c'est une commande qui demande au serveur d'effectuer une certaine action). Voici la liste des méthodes disponibles :

GET, HEAD, POST, OPTIONS, CONNECT, TRACE, PUT, PATCH, DELETE

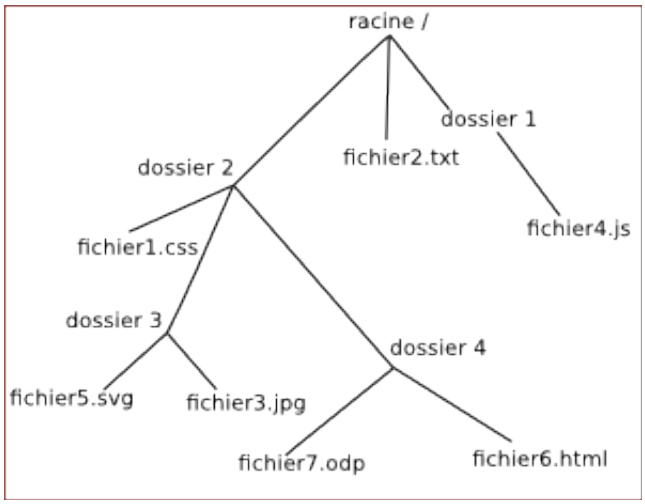
Détaillons 4 de ces méthodes :

- GET : C'est la méthode la plus courante pour demander une ressource. Elle est sans effet sur la ressource.
- POST : Cette méthode est utilisée pour soumettre des données en vue d'un traitement (côté serveur). Typiquement c'est la méthode employée lorsque l'on envoie au serveur les données issues d'un formulaire (balise <form>, nous aurons l'occasion de voir des exemples plus tard).
- DELETE : Cette méthode permet de supprimer une ressource sur le serveur.
- PUT : Cette méthode permet de modifier une ressource sur le serveur

L'URL (et l'URI)

Une URI (Uniform Ressource Identifier) permet d'identifier une ressource sur un réseau, une URL est un cas particulier d'URI. Nous ne nous attarderons pas sur les subtiles différences entre une URI et une URL et à partir de maintenant je parlerai exclusivement d'URL (par souci de simplification).

L'URL indique « l'endroit » où se trouve une ressource sur le serveur. Un fichier peut se trouver dans un dossier qui peut lui-même se trouver dans un autre dossier.....on parle d'une structure en arborescence, car elle ressemble à un arbre à l'envers :



Comme vous pouvez le constater, la base de l'arbre s'appelle la racine de l'arborescence et se représente par un /

Chemin absolu ou chemin relatif ?

Pour indiquer la position d'un fichier (ou d'un dossier) dans l'arborescence, il existe 2 méthodes : indiquer un chemin absolu ou indiquer un chemin relatif. Le chemin absolu doit indiquer « le chemin » depuis la racine. Par exemple l'URL du fichier fichier3.txt sera : /dossier2/dossier3/fichier3.jpg

Remarquez que nous démarrons bien de la racine / (attention les symboles de séparation sont aussi des /)

Imaginons maintenant que le fichier fichier1.css fasse appel au fichier fichier3.jpg (comme un fichier HTML peut faire appel à un fichier CSS ou JavaScript). Il est possible d'indiquer le chemin non pas depuis la racine, mais depuis le dossier (dossier2) qui accueille le fichier1.css, nous parlerons alors de chemin relatif :

dossier3/fichier3.jpg

Remarquez l'absence du / au début du chemin (c'est cela qui nous permettra de distinguer un chemin relatif et un chemin absolu).

Imaginons maintenant que nous désirions indiquer le chemin relatif du fichier fichier1.txt depuis le dossier dossier4.

Comment faire ?

Il faut « reculer » d'1 « cran » dans l'arborescence (pour se retrouver dans le dossier dossier2 et ainsi pouvoir repartir vers la bonne « branche ». Pour ce faire il faut utiliser 2 points : ..

../dossier2/fichier3.jpg

Il est tout à fait possible de remonter de plusieurs « crans » : ../../ depuis le dossier dossier4 permet de « retourner » à la racine.

Remarque : la façon d'écrire les chemins (avec des slash (/) comme séparateurs) est propre aux systèmes dits « UNIX », par exemple GNU/Linux ou encore Mac OS. Sous Windows, ce n'est pas le slash qui est utilisé, mais l'antislash (\). Pour ce qui nous concerne ici : les chemins réseau (et donc le web), pas de problème, c'est le slash qui est utilisé.

Réponse du serveur à une requête HTTP

Une fois la requête reçue, le serveur va renvoyer une réponse, voici un exemple de réponse du serveur :

```
HTTP/1.1 200 OK
Date: Thu, 15 feb 2013 12:02:32 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) DAV/2 SVN/1.1.4
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=ISO-8859-1
```

```
<!doctype html>
<html lang="fr">
<head>
<meta Charest="utf-8">
<title>Voici mon site</title>
</head>
<body>
<h1>Hello World! Ceci est un titre</h1>
<p>Ceci est un <strong>paragraphe</strong>. Avez-vous bien compris ?</p>
</body>
</html>
```

Nous n'allons pas détailler cette réponse, voici quelques explications sur les éléments qui nous seront indispensables par la suite :

Commençons par la fin : le serveur renvoie du code HTML, une fois ce code reçu par le client, il est interprété par le navigateur qui affiche le résultat à l'écran. Cette partie correspond au corps de la réponse.

La 1re ligne se nomme la ligne de statut :

- `HTTP/1.1 :` version de HTTP utilisé par le serveur
- `200 :` code indiquant que le document recherché par le client a bien été trouvé par le serveur. Il existe d'autres codes dont un que vous connaissez peut-être déjà : le code 404 qui signifie : «Le document recherché n'a pu être trouvé».

Les 5 lignes suivantes constituent l'en-tête de la réponse, une ligne nous intéresse plus particulièrement : «`Server: Apache/2.0.54 (Debian GNU/Linux)`».

Les serveurs HTTP

Il existe différents types de serveur capable de répondre à des requêtes HTTP (on les appelle serveurs HTTP ou encore serveur web). Que faut-il pour constituer un serveur web ?

- un ordinateur (souvent ce sont des machines spécialisées : elles sont conçues pour fonctionner 24h/24h....., mais il est possible d'utiliser un ordinateur « classique » (surtout si votre but est uniquement de faire des tests).

- un système d'exploitation : Les distributions GNU/Linux sont, pour différentes raisons, à privilégier. Dans la réponse HTTP que nous avons étudiée ci-dessus, le serveur fonctionne sous une distribution GNU/Linux dénommée Debian (mais ici aussi, si vous êtes sous Windows, il est tout de même possible de mettre en place un serveur web).
- Un logiciel destiné à recevoir les requêtes HTTP et à fournir des réponses. Un des logiciels les plus « populaires » se nomme Apache (il équipe plus de la moitié de serveur web en activité dans le monde !), mais il en existe d'autres : nginx, lighttpd.....(j'ai volontairement choisi d'évoquer uniquement les solutions « libres », mais vous devez savoir qu'il existe aussi des solutions « propriétaires », si c'est 2 termes vous sont inconnus, il serait bon de rechercher leur signification).

Les langages côté serveur

Il y a quelques années, le web était dit « statique » : le concepteur de site web écrivait son code HTML et ce code était simplement envoyé par le serveur au client. Les personnes qui consultaient le site avaient toutes le droit à la même page, le web était purement « consultatif » (nous étions encore très loin des « single page application » que nous verrons plus tard).

Les choses ont ensuite évolué : les serveurs sont aujourd'hui capables de générer eux même du code HTML. Les résultats qui s'afficheront à l'écran dépendront donc des demandes effectuées par l'utilisateur du site : le web est devenu dynamique.

Différents langages de programmation peuvent être utilisés « côté serveur » afin de permettre au serveur de générer lui même le code HTML à envoyer. Le plus utilisé encore aujourd'hui se nomme PHP (PHP Hypertext Preprocessor : c'est un acronyme récursif comme GNU qui signifie GNU's Not Unix). Nous n'utiliserons pas PHP, je n'insisterai donc pas sur ce sujet. D'autres langages sont utilisables côté serveur (pour permettre la génération dynamique de code HTML), voici un liste non exhaustive : Java, Python, ASP....

J'ai volontairement omis un langage : le JavaScript. L'utilisation du JavaScript côté serveur est relativement récente, mais se développe à vitesse grand V.

Pour ne pas partir avec de fausses idées en tête, vous devez bien comprendre que nous allons parler de JavaScript qui s'exécute sur le serveur. Quand vous associez une page HTML avec un script JavaScript (à l'aide de la balise <script>), ce script est bien stocké sur le serveur, mais il est envoyé au client (par l'intermédiaire d'une requête HTTP) afin d'être exécuté. Attention à ne pas confondre le JavaScript exécuté côté client (par le navigateur) et le JavaScript exécuté côté serveur.

Et nodeJS fût....

Comme je viens de vous le rappeler, jusqu'à une époque récente, le JavaScript était uniquement destiné à être exécuté sur le client. Pour être plus précis, tous les navigateurs web (Firefox, Chrome, Opéra...) sont « équipés » d'un moteur JavaScript. Ce moteur JavaScript a pour but d'interpréter le code et d'afficher le résultat dans le navigateur.

Le moteur du navigateur made in Google, Chrome, se nomme V8.

Ryan Dahl a décidé, en 2009, d'utiliser le moteur V8 afin de permettre l'exécution de JavaScript côté serveur (c'est à dire sans utiliser de navigateur web) : nodeJS était né !

Je précise que nodeJS n'est ni le premier, ni le seul à proposer du JavaScript côté serveur, mais force et de constater qu'aujourd'hui c'est lui qui à le vent en poupe.

Précisons une chose : nodeJS n'est pas un serveur web (comme Apache par exemple), c'est juste une plateforme permettant d'exécuter du JavaScript. Avec nodeJS nous allons devoir programmer notre serveur web !

Installation de nodeJS et programmation d'un serveur web

Il existe beaucoup de tutoriaux qui décrivent l'installation de nodeJS sur les plateformes « classiques » (GNU/linux, windows ou encore Mac OS). Je vous propose le tutoriel de Mathieu Nebra sur le site du zéro : http://www.siteduzero.com/informatique/tutoriels/des-applications-ultra-rapides-avec-node-js/installer-node-js

À *faire vous même*

Une fois l'installation terminée, ouvrez une console (aussi appelé terminal) et tapez « node -v ». Si votre installation s'est bien passée, vous devriez voir la version de nodeJS s'afficher dans la console.

Quelle est la version de nodeJS que vous allez utiliser ? ….....

Nous allons écrire notre premier programme avec nodeJS. Vous allez créer un dossier dans l'endroit de votre choix et créer un fichier JavaScript vierge.

Il existe de nombreux modules « livrés » d'office avec nodeJS, parmi ceci nous allons utiliser le module http.

À *faire vous même*

Saisissez le code suivant dans le fichier que vous venez de créer et enregistrez-le (vous nommerez ce fichier ex9.js)

code JavaScript (nodeJS) ex9.js

```
var http = require('http');
var serveur = http.createServer(function(req, res) {
    res.writeHead(200, {"Content-Type": "text/html"});
    res.write('<!doctype html>'
                    +'<html lang="fr">'
                    +'<head>'
                    +'<meta Charest="utf-8">'
                    +'<title>nodeJS</title>'
                    +'</head>'
                    +'<body>'
                    +'<h1>Hello World! Ceci est un titre</h1>'
                    +'<p>Ceci est un <strong>paragraphe</strong>.</p>'
                    +'</body>'
                    +'</html>');
    res.end();
});
serveur.listen(8080);
```

En utilisant la console, placez-vous dans le dossier qui accueille le fichier ex9.js (n'oubliez pas les antislashs si vous êtes sous Windows). Entrez dans la console : «node ex9.js »
Sans fermer la console, ouvrez un navigateur web, dans la barre d'adresse, tapez : « localhost:8080 »
Normalement notre page web devrait s'afficher.

Quelques explications s'imposent :

Depuis le début de la deuxième partie, j'insiste sur le fait que la consultation d'un site internet est un échange de données entre 2 ordinateurs distants (un client et un serveur), or, ici, nous n'utilisons qu'un ordinateur ?
Dans toute la phase de développement, il est tout à fait possible de n'utiliser qu'un seul ordinateur qui jouera à la fois (et en même temps) le rôle du client et le rôle serveur.
L'exécution, dans la console, de la commande node ex9.js « démarre » le serveur, une fois le serveur démarré il « attend » les requêtes HTTP en provenance d'un client. Le navigateur web va envoyer ces requêtes au serveur. Mais comment entrer en communication avec le serveur ? Il suffit d'utiliser une adresse un peu spéciale : localhost (localhost indique au navigateur web que le serveur se trouve sur la même machine que lui).
Le : 8080 définie le port utilisé par le serveur. Plusieurs applications peuvent utiliser la même connexion réseau à condition de ne pas utiliser le même port (on parle aussi de socket). Ici notre serveur « écoute » et « attend » une requête HTTP sur le port 8080(nous aurions pu en choisir un autre).
Passons maintenant à l'étude du code :
«`var http = require('http');`» : nodeJS possède de nombreux modules afin d'étendre ces possibilités de bases. Nous utilisons ici le module http qui permet de traiter des requêtes HTTP (et donc de développer un serveur web). Nous créons un objet (au sens programmation orientée objet, avec donc des attributs et des méthodes) que l'on nomme tout simplement http (nous aurions pu prendre un autre nom) à l'aide de l'instruction « require ».
La deuxième ligne est très longue, nous allons la décortiquer :
«`var serveur = http.createServer(...`» : nous créons un objet dénommé « serveur ». Pour créer cet objet, nous utilisons une méthode de l'objet http : createServer. La méthode createServer accepte un paramètre : une fonction anonyme (en JavaScript, une fonction anonyme est une fonction qui ne porte pas de nom). Quel est le rôle de cette fonction anonyme ?
Nous rentrons ici dans le cœur de ce qui fait nodeJS : les callbacks
La programmation sous nodeJS est un peu particulière, tout est une question d'événement. La fonction anonyme passée en paramètre de la méthode createServer sera exécutée à chaque fois que le serveur recevra une requête HTTP venant de l'extérieur, le reste du temps, notre programme passera son temps à attendre. Cette fonction anonyme est appelée « callback » de l'événement requête HTTP (je vous donne le terme, car vous le rencontrerez sans doute si vous étudiez nodeJS à l'aide de ressources trouvées sur internet).

Étudions maintenant cette fonction de callback :

```
«function(req, res) {
    res.writeHead(200, {"Content-Type": "text/html"});
    res.write('<!doctype html>'
                    +'<html lang="fr">'
```

```
                    +'<head>'
                    +'<meta Charest="utf-8">'
                    +'<title>nodeJS</title>'
                    +'</head>'
                    +'<body>'
                    +'<h1>Hello World! Ceci est un titre</h1>'
                    +'<p>Ceci est un <strong>paragraphe</strong>.</p>'
                    +'</body>'
                    +'</html>');
      res.end() ;
}»
```

La fonction de callback prend deux paramètres : req (pour request) et res (pour response (réponse en anglais)). req et res sont tous les deux des objets, req va nous permettre de manipuler tout ce qui touche à la requête HTTP reçue par le serveur alors que res permettra de manipuler tout ce qui touche à la réponse du serveur (il est tout à fait possible de choisir d'autres noms : le premier paramètre concernera la requête et le second la réponse).
«`res.writeHead(200, {"Content-Type": "text/html"});`» : permet de « remplir » l'en tête de la réponse HTTP avec le code de retour 200 (« Tout c'est bien passé ») et avec le type de document renvoyé (du HTML).

```
«res.write('<!doctype html>'
                    +'<html lang="fr">'
                    +'<head>'
                    +'<meta Charest="utf-8">'
                    +'<title>nodeJS</title>'
                    +'</head>'
                    +'<body>'
                    +'<h1>Hello World! Ceci est un titre</h1>'
                    +'<p>Ceci est un <strong>paragraphe</strong>.</p>'
                    +'</body>'
                    +'</html>');»
```

res.write permet de remplir le corps de la réponse HTTP : nous renvoyons une chaine de caractère (concaténation de plusieurs chaines de caractères) correspondant à du code HTML (j'ai placé les différentes balises les unes sous les autres afin de faciliter la lecture, sachez qu'il aurait été possible d'avoir : '`<!doctype html><html lang="fr"><head>....</html>'`. C'est moins lisible, mais cela évite les concaténations).
«`res.end() ;`» : permet de terminer notre réponse HTTP avant son envoi vers le client.

Cette notion de fonction de callback est fondamentale pour la suite, si vous avez du mal, n'hésitez pas à poser des questions avant d'aller plus loin.

Il nous reste une dernière ligne à étudier : «`serveur.listen(8080);`». On applique la méthode listen à l'objet «serveur» (objet qui a été créé à l'aide de la méthode createServer), cette méthode « met en route le serveur » et lui indique le port sur lequel il doit « écouter » afin de recevoir les requêtes HTTP.

Cet exemple fonctionne, mais il peut satisfaisant ! Allons-nous nous amuser à mettre du code HTML dans notre fichier JavaScript ? Bien sûr que non, pour faciliter les choses nous allons utiliser un framework de nodeJS : express.

expressJS

ExpressJS (http://expressjs.com/) est un module pour nodeJS développé, entre autres, par Tj Holowaychuk. Nous n'allons utiliser qu'une petite partie des possibilités de ce framework.
ExpressJS n'est pas intégré par défaut à nodeJS, il faut l'installer à l'aide de l'outil npm « node packaged modules ».

À *faire vous même*

À l'aide de la console, placez-vous dans le dossier qui va accueillir votre fichier JavaScript (côté serveur), puis tapez : npm install express
Cela devrait créer un nouveau dossier (node_modules). Si vous changez de dossier de travail, il faudra réinstaller express (chaque dossier contient sa propre installation d'express).

Passons au premier exemple :

Coder les 2 fichiers suivants (ex10.html devra se trouver dans un sous-dossier app)

code JavaScript (nodeJS) ex10.js

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

```
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex10.html');
});
app.listen(8080);
```

code HTLM ex10.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
<meta Charest="utf-8">
<title>Voici mon site</title>
</head>
<body>
	<h1>Hello World! Ceci est un titre</h1>
	<p>Ceci est un <strong>paragraphe</strong></p>
</body>
</html>
```

Quel est le résultat (après avoir démarré le serveur, entrez http://localhost:8080 dans la barre d'adresse d'un navigateur web) ?
….....
.....

Analyse du fichier ex10.js :
« `var express = require('express');` » : nous commençons par « importer » le module express (indispensable à partir du moment où vous voulez utiliser express)
«`var app = express();` » : nous créons un objet « générique » qui nous permettra de manipuler tous les outils fournis par express.
«`app.get('/', function (req, res) {`
`	res.sendfile(__dirname + '/app/ex10.html');`
`});` »
Cette ligne est fondamentale.
Nous utilisons la méthode get de l'objet app : Cette méthode est très importante puisque c'est elle qui va permettre à notre serveur de répondre aux requêtes HTTP de type GET (nous verrons plus loin qu'il existe aussi une méthode post).
Cette méthode accepte deux paramètres : '/' et une fonction anonyme.

Quand vous écrivez l'adresse d'un site, on peut distinguer 3 parties :
http://www.monsite.fr/mondossier/

en vert : le protocole
en bleu : l'adresse du serveur
en rouge : un chemin (URL)

Le premier paramètre de la méthode get correspond à la partie en rouge.

Sur notre machine de « test» (client et serveur sur un même ordinateur), la fonction anonyme (2e paramètre de la méthode get) sera exécutée quand l'adresse suivante : http://localhost:8080/ (le / final n'étant pas obligatoire, cela revient à écrire : http://localhost:8080)

Attention : Avec express, cette URL (1er paramètre de la méthode get) n'a pas besoin de correspondre à l'emplacement réel de la ressource sur le serveur. Si je reprends l'exemple ci-dessus (http://www.monsite.fr/mondossier/), il se peut tout à fait que le dossier mondossier n'existe pas réellement sur le serveur. Nous verrons un peu plus loin l'intérêt de ces URL « fantômes» (avec la technique REST).

Étudions maintenant la fonction anonyme « `function (req, res) {`
`	res.sendfile(__dirname + '/app/ex10.html');`
`}`»
Comme avec le module http vu plus haut, nous avons ici affaire en une fonction callback ayant 2 paramètres : req et res (exactement la même signification que dans l'exemple ex9.js).
Nous utilisons la méthode sendfile de l'objet res, qui, comme son nom l'indique permet de renvoyer le contenu d'un fichier dans le corps de la réponse HTTP. Cette méthode a pour paramètre le chemin du fichier à renvoyer.
Comme vous pouvez le constater, nous utilisons la variable __dirname qui contient le chemin absolu du dossier courant (dossier contenant le fichier ex10.js). En fonction de la machine, ce chemin absolu vers le dossier courant sera différent,

d'où l'intérêt de __dirname.
Avec «`__dirname + '/app/ex10.html'`», nous avons donc une concaténation entre le chemin absolu et le chemin du fichier à envoyer. C'est un peu compliqué, donc voici un exemple :
Sur ma machine, le chemin absolu menant au fichier ex10.html est : /home/lycee/nodejs/app/ex10.html
Le dossier nodejs contient (en plus du dossier app) le fichier ex10.js, le chemin absolu menant au fichier ex10.js est donc : /home/lycee/nodejs. Nous avons donc __dirname='/home/lycee/nodejs'.
En faisant la concaténation «`__dirname +'/app/ex10.html'`», j'obtiens bien le chemin absolu du fichier ex10.html.

Imaginons maintenant que je copie le dossier nodejs (et son contenu) sur une autre machine. Sur cette autre machine le chemin absolu qui mène au fichier ex10.js est : /siteinternet/apprendrenodejs/nodejs, nous avons donc __dirname='/siteinternet/apprendrenodejs/nodejs'. La concaténation «`__dirname +'/app/ex10.html'`» nous permet donc bien de retrouver le chemin absolu du fichier ex10.html sur cette nouvelle machine. Le changement de machine ne nécessite pas de modification du code.

La dernière ligne «`app.listen(8080);`» ne devrait pas vous poser de problème si vous avez bien compris l'exemple précédent (ex9.js).

AngularJS et nodeJS

Nous allons maintenant pouvoir réutiliser ce que vous avez appris dans la 1re partie de ce document et utiliser AngularJS.

À *faire vous même*

Reprenons l'exemple n°1 de la première partie :

code HTLM ex11.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex11.js"></script>
</head>
<body ng-app>
	<h1 ng-controller="monControl">{{maVariable}}</h1>
</body>
</html>
```

code JavaScript ex11.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope){
	$scope.maVariable="Hello World !";
}
```

code JavaScript ex11n.js (nodejs)

```
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex11.html');
});
app.listen(8080);
```

N'oubliez pas de créer un dossier app/lib/ qui contiendra le fichier AngularJS (angular.min.js)
Analyse du fichier ex11n.js :

Une seule nouveauté à signaler dans cet exemple :

```
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
```

Nous utilisons la méthode « use » de l'objet app.
Le fichier html nous indique que le fichier ex11.js se trouve dans le dossier javascript et que le fichier angular.min.js se trouve dans le dossier lib. Il faut indiquer à express où se trouvent « réellement » ces fichiers sur le serveur. Pour cela nous utilisons une méthode (static) de l'objet express, nous passons le chemin absolu à utiliser («`__dirname + '/app/javascript'`») pour trouver le fichier.
En résumé, si je prends comme exemple la 2e ligne «`app.use('/lib', express.static(__dirname + '/app/lib'));`», si dans le fichier html (ou ailleurs...) on recherche un fichier se trouvant dans le répertoire « lib », il faudra allez le chercher dans le répertoire « lib » ayant pour chemin absolu «`__dirname + '/app/lib'`».
Sans ces 2 lignes, cet exemple ne fonctionnerait pas.

Envoyer du JSON

Dans le cas des sites web « classiques», la plupart du temps, le serveur renvoie du code HTML au client (que ce soit du code HTML « statique » ou « fabriqué à la volée » par le serveur). À chaque échange client-serveur, une nouvelle page s'affiche ce qui rend la consultation du site quelque peu « hachée » (surtout si la connexion internet n'est pas très performante).
Il est possible d'éviter ces envois incessants de code HTML en « demandant » au serveur de nous envoyer non plus tout le code HTML, mais seulement ce qui est nécessaire pour actualiser la page côté client. Pour ce faire, une série de technologies est mise en œuvre. Toutes ces techniques sont regroupées sous le nom d'AJAX (Asynchronous JavaScript And XML).
Mon but n'est pas de m'étendre sur AJAX, vous devez juste savoir qu'à l'origine les données qui vont permettre l'actualisation de la page web transitent entre le serveur et le client sous la forme de fichier XML (eXtensible Markup Language). Pour différentes raisons, le XML est de moins en moins utilisé au profit du JSON (JavaScript Object Notation), nous devrions donc utiliser le terme AJAJ à la place d'AJAX !
Sans le savoir, le JSON, vous connaissez déjà puisqu'à peu de choses près cela ressemble aux objets JavaScript que nous avons déjà rencontrés.
Dans les exemples qui vont suivre, le serveur enverra principalement au client des données au format JSON (et très peu de code HTML).

À *faire vous même*

Étudiez cet exemple

code JavaScript ex12n.js (nodejs)

```
var ficheInfo=[
	{
		id:1,
		nom:"Depuis",
		prenom:"Jean"
	},
	{
		id:2,
		nom:"Durand",
		prenom:"Christian"
	},
	{
		id:3,
		nom:"Martin",
		prenom:"Michel"
	}
]
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));


app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex12.html');
});
app.get('/api/affiche', function(req, res) {
  res.json(ficheInfo);
});
app.listen(8080);
```

code JavaScript ex12.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.affichage=function(){
		$http.get('/api/affiche')
		.success(function(data){
			$scope.listePerso=data;
		})
	}
}
```

code HTLM ex12.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex12.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<button ng-click="affichage()">Afficher la liste</button>
	<h1>Liste des adhérents</h1>
	<div ng-repeat="perso in listePerso">
		<h3>Identifiant : {{perso.id}}</h3>
		<p>Nom : {{perso.nom}}</p>
		<p>Prénom : {{perso.prenom}}</p>
	</div>
</body>
</html>
```

Je vous laisse étudier cet exemple (qui commence vraiment à se complexifier), voici quelques éléments pour vous aider :

ex12.html : ras (AngularJS « classique »)

ex12.js :
Nous utilisons une structure un peu spéciale : « `$http.get(...) .success(function(data){ ...})` »
$http est un objet propre à AngularJS, il permet d'envoyer une requête HTTP au serveur.
Nous appliquons la méthode get (`$http.get('/api/affiche')`) à l'objet $http. Le paramètre de la méthode get correspond à l'URL de destination : nous aurons une requête de type GET vers l'URL /api/affiche (en utilisant la notation déjà vue plus haut : «`GET /api/affiche HTTP/1.1`»).
Intéressons-nous ensuite au «`.success(function(data){ ...})` » : en cas de réponse positive du serveur (code 200), la fonction anonyme se trouvant en paramètre de la méthode success sera appelée.
«`function(data){ $scope.listePerso=data; })` » : le paramètre data correspond à la ressource renvoyée par le serveur (dans notre exemple c'est un tableau contenant des objets JavaScript).
«`$scope.listePerso=data;`» : Le tableau d'objet envoyé par le serveur (qui porte le nom de ficheInfo côté serveur) est maintenant exploitable côté client (sous le nom de listePerso).
Noter que pour pouvoir utiliser $http il faut ajouter le paramètre $http à la fonction « contrôleur » (`function monControl($scope,$http){ ...`)

ex12n.js :
ficheInfo est un tableau contenant des objets JavaScript.
« `app.get('/api/affiche', function(req, res) {res.json(ficheInfo);});` » : nous avons déjà eu l'occasion de parler de la méthode get de l'objet app, en revanche, le contenu de la fonction de callback est intéressant.
« `function(req, res) {res.json(ficheInfo);}`»: json est une méthode (de l'objet res) permettant de renvoyer dans le corps de la réponse HTTP un objet JavaScript ou un tableau contenant des objets JavaScript (ce qui est notre cas ici).

À la lumière de ces précisions, essayez de décrire le résultat attendu (n'hésitez pas à détailler la requête et la réponse HTTP en précisant le contenu de la réponse).
….....
.....
.....

« Autocorrigez »-vous en lançant le serveur (ex12n.js) et en rentrant «localhost:8080» dans la barre d'adresse d'un navigateur web.

Utilisation de la méthode POST

À *faire vous même*

Étudiez cet exemple

code JavaScript ex13n.js (nodejs)

```
var ficheInfo=[]
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());

app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex13.html');
});
app.get('/api/affiche', function(req, res) {
  res.json(ficheInfo);
});
app.post('/api/formulaire', function(req,res) {
	monPerso=req.body;
	monPerso.id=1+ficheInfo.length
	ficheInfo.push(monPerso);
	res.send();
});
app.listen(8080);
```

code JavaScript ex13.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.affiche=function(){
		$http.get('/api/affiche')
		.success(function(data){
			$scope.listePerso=data;
		})
	}
	$scope.traitForm=function(){
		$http.post('/api/formulaire',$scope.perso)
		.success(function(data){
			$scope.perso={};
			$scope.affiche();
		})
	}
	$scope.affiche();
}
```

code HTLM ex13.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex13.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<form>
		<p>Nom <input type:'text' ng-model="perso.nom"></p>
		<p>Prénom <input type:'text' ng-model="perso.prenom"></p>
		<button ng-click="traitForm()">Valider</button>
	</form>
	<h1>Liste des adhérents</h1>
```

```
	<div ng-repeat="perso in listePerso">
		<h3>N° fiche : {{perso.id}}</h3>
		<p>Nom : {{perso.nom}}</p>
		<p>Prénom : {{perso.prenom}}</p>
	</div>
</body>
</html>
```

Quelques informations sur cet exemple :

ex13.html : ras

ex13.js :
Nous utilisons $http pour cette fois-ci envoyer une requête HTTP de type POST (c'est logique puisque nous envoyons le contenu d'un formulaire vers le serveur) «`$http.post('/api/formulaire',$scope.perso)` », vous remarquerez que la méthode post de l'objet $http prend 2 paramètres, l'URL (comme pour la méthode get) et les données du formulaire à envoyer au serveur (ici aussi nous utilisons un objet JavaScript, vous pouvez constater que comme dit plus haut, tous nos transferts client-serveur et serveur-client se font en JSON).
Je vous laisse analyser la méthode success associer à $http.

ex13n.js
«`app.use(express.bodyParser());`» : cette ligne est indispensable à partir du moment où votre serveur reçoit du JSON (formulaire en provenance du client). Express utilise ce que l'on appelle un parseur (analyseur syntaxique en bon français) afin de pouvoir exploiter les données reçues.
«
```
app.post('/api/formulaire', function(req,res) {
	monPerso=req.body;
	monPerso.id=1+ficheInfo.length;
	ficheInfo.push(monPerso);
	res.send();
```
`});`» : nous traitons une requête de type POST.
La fonction anonyme est exécutée dès que le serveur reçoit une requête HTTP de type POST à l'URL «`'/api/formulaire'`».
Contenu de la fonction anonyme :
«`monPerso=req.body;`» : req.body (attribut body de l'objet req) contient les données contenues dans le formulaire envoyé par le client. Plus généralement il contient les données transportées par la requête HTTP de type POST.
Je vous laisse analyser les 2 lignes suivantes : «`monPerso.id=1+ficheInfo.length;`» et «`ficheInfo.push(monPerso);`».
La dernière ligne, «`res.send();`», est très importante, car le client a fait une requête, il faut donc que le serveur réponde à cette requête. La réponse du serveur à un corps vide, mais cela n'a pas d'importance (il faut que le code 200 soit retourné au client : «`HTTP/1.1 200 OK`»).

À la lumière de ces précisions, essayez de décrire le résultat attendu (toujours en détaillant les requêtes et les réponses HTTP)
….....
.....
.....
.....

« Autocorrigez »-vous en lançant le serveur (ex13n.js) et en rentrant «localhost:8080» dans la barre d'adresse d'un navigateur web.

Vous avez sans doute compris la technique employée dans ces exemples (12 et 13) : le client envoie des « ordres » au serveur par l'intermédiaire des URL. Cette technique est très développée sur le web, on la dénomme REST.
De « grands acteurs » du web (Google, Facebook, twitter,....) fournissent ce que l'on appelle des API (Application Programming Interface), souvent basées sur REST, permettant aux développeurs de créer des applications capables d'interagir avec leurs serveurs. C'est comme cela que l'on voit fleurir, par exemple, des clients Twitter développés par des indépendants (Tweetbot, Seesmic,....).

Paramètres dans une requête HTTP de type GET

Il peut être utile, dans certains cas, de faire passer des paramètres par l'intermédiaire d'une URL lors d'une requête HTTP de type GET.

À *faire vous même*

Étudiez cet exemple

code JavaScript ex14n.js (nodejs)

```
var ficheInfo=[
	{
		id:1,
		nom:"Durand",
		prenom:"Pierre"
	},
	{
		id:2,
		nom:"Dupont",
		prenom:"Christophe"
	},
	{
		id:3,
		nom:"Martin",
		prenom:"Michel"
	}
]
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());

app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex14.html');
});
app.get('/api/affiche/:id', function(req, res) {
	for (var i=0;i<ficheInfo.length;i++) {
		if (ficheInfo[i].id==req.params.id){
			res.json(ficheInfo[i]);
		}
	}
	res.send();
});
app.listen(8080);
```

code JavaScript ex14.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.afficheFiche=function(){
		$http.get('/api/affiche/'+$scope.ident)
		.success(function(data){
			$scope.fiche=data;
			$scope.ident="";
		})
	}
}
```

code HTLM ex14.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex14.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<form>
		<p>Recherche par n° de fiche <input type:'text' ng-model="ident"></p>
		<button ng-click="afficheFiche()">Valider</button>
	</form>
```

```
        <h3>résultat recherche</h3>
        <p>Nom : {{fiche.nom}}</p>
        <p>Prénom : {{fiche.prenom}}</p>
</body>
</html>
```

Dans ex14.js, j'attire votre attention sur «`$http.get('api/affiche/'+$scope.ident)` » (surtout sur la dernière partie de l'URL, d'où vient le «`$scope.ident`» ?).
La seule véritable nouveauté se trouve dans le fichier ex14n.js avec «`app.get('/api/affiche/:id',...` ». Ici, id est un paramètre (ce sont les 2 points placés juste avant qui lui donne ce statut de paramètre). Si l'URL est «/api/affiche/1» alors le «`req.params.id`» sera égal à 1. Si l'URL est «/api/affiche/5» alors le «`req.params.id`» sera égal à 5.....
En règle générale, quand on aura une URL de type « /.../.../:parametre » on pourra utiliser ce paramètre dans la fonction de callback sous la forme req.params.parametre.
Il est tout à fait possible d'avoir plusieurs paramètres dans l'URL : « /.../.../:parametre1/:parametre2 » que l'on retrouvera plus loin dans « req.params.parametre1 » et « req.params.parametre2 ».

Essayez de décrire (et surtout d'expliquer) le résultat attendu.
….....
.....
.....
.....

À *faire vous même*

L'exemple 14 a un gros défaut : il n'avertit pas l'utilisateur que « la fiche recherchée n'existe pas ». Modifiez l'exemple 14 afin d'éliminer ce défaut.

À *faire vous même*

Écrire le code nécessaire pour que la recherche ne porte plus sur le n° de fiche, mais sur le nom de la personne.
Attention, il faudra qu'en cas d'homonyme, toutes les fiches correspondantes soient affichées.
Exemple :
si je pars du tableau suivant :

```
var ficheInfo=[ { id:1, nom:"Durand", prenom:"Pierre" }, { id:2, nom:"Dupont", prenom:"Christophe" }, { id:3, nom:"Martin", prenom:"Michel" }, { id:4, nom:"Martin", prenom:"Jean-Louis" }, { id:5, nom:"Durand", prenom:"Mathieu" } ]
```

La recherche « Durand » devra m'afficher les 2 fiches suivantes :

résultat recherche pour Durand

Fiche n°1

Nom : Durand

Prénom : Pierre

Fiche n°5

Nom : Durand

Prénom : Mathieu

À *faire vous même*

Sachant qu'il est possible d'émettre une requête HTTP de type DELETE ($http.delete voir la documentation http://docs.angularjs.org/api/ng.$http), permettant de supprimer une fiche à partir de son numéro. Vous partirez du tableau suivant :

```
var ficheInfo=[{id:1, nom:"Durand", prenom:"Pierre"},
{id:2,nom:"Dupont",prenom:"Christophe"},{id:3,nom:"Martin",prenom:"Michel"},
{id:4,nom:"Martin",prenom:"Jean-Louis"},{id:5,nom:"Durand",prenom:"Mathieu"}]
```

Vous pourrez utiliser la méthode splice qui permet de supprimer un ou plusieurs éléments d'un tableau :
tab.splice(i,nb) => i indice de départ, nb nombre d'éléments à supprimer à partir de i

Retour sur AngularJS : les routes

Jusqu'à présent nous avons uniquement conçu des «applications » (ou site web) d'une seule page. AngularJS gère très bien les applications multipages.

Le serveur ne comporte pas beaucoup de nouveautés, nous déclarons seulement quelques URL supplémentaires.

code JavaScript ex15n.js (nodejs)

```
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use('/image', express.static(__dirname + '/app/image'));
app.use('/css', express.static(__dirname + '/app/css'));
app.use('/template', express.static(__dirname + '/app/template'));
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex15.html');
});
app.listen(8080);
```

Afin de rendre les choses plus lisibles (et mieux ordonnées) nous allons créer un nouveau fichier (config_ex15.js).

code JavaScript config_ex15.js (dans un sous-dossier app/JavaScript/)

```
var app_ex15=angular.module('app_ex15',[]);
app_ex15.config(function($routeProvider){
	$routeProvider
		.when('/home',{templateUrl:'template/ex15_1.html',controller:''})
		.when('/formulaire',{templateUrl:'template/ex15_2.html',controller:''})
		.when('/article',{templateUrl:'template/ex15_3.html'})
		.otherwise({redirectTo:'/home'});
})
```

À la première ligne nous créons un objet app_ex15, la méthode module va permettre de créer cet objet. Cette méthode module prend 2 paramètres : le nom de l'application (c'est une nouveauté, voir les remarques sur le fichier ex15.html) et un tableau que nous n'utiliserons ici (nous utilisons donc un tableau vide).
Nous avons ensuite une structure de type : app_ex15.config(function($routeProvider){$routeProvider.when(..).when(..)
Nous allons définir les différentes pages accessibles dans les différents when (1 when par page).
Détaillons le 1er exemple :
«`when('/home'{templateUrl:'template/ex15_1.html',controller:'controex15_1'})`»
Quand l'utilisateur clique, par exemple sur un lien, AngularJS utilise le fichier défini dans l'objet («`templateUrl:'template/ex15_1.html'`»), afin d'afficher du contenu.
Le «`controller:'controex15_1'`» permet de définir le contrôleur qui sera associé au contenu de la page ex15_1.html (il sera donc inutile de définir ce contrôleur dans le fichier HTML) .
Si l'URL n'est pas répertoriée dans la série de when, elle sera, par défaut, traitée comme si c'était l'URL /home («`.otherwise({redirectTo:'/home'})`».

code JavaScript ex15.js (dans un sous-dossier app/JavaScript/)

```
function controex15_1 ($scope){}
function controex15_2 ($scope){}
function controex15_3 ($scope){}
```

Le but de cet exemple étant d'illustrer uniquement la notion routeProvider, les contrôleurs sont « vides ».

code HTLM ex15.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/config_ex15.js"></script>
	<script src="javascript/ex15.js"></script>
	<link rel="stylesheet" href="css/ex15.css">
</head>
```

```
<body ng-app="app_ex15">
	<header>
		<h1>Lorem Ipsum</h1>
		<h4>ne plus reddat quam</h4>
	</header>
	<div class="row-fluid">
		<nav class="span2">
			<ul class="nav nav-list">
				<li class="nav-header">Menu</li>
				<li><a href="#/home">Home</a></li>
				<li><a href="#/formulaire">Formulaire</a></li>
				<li><a href="#/article">Article</a></li>
			</ul>
		</nav>
		<div class="span10" ng-view>
		</div>
	</div>
	<footer>
		<p>footer : cum artibus honestissimis erudiretur.Altera sententia est</p>
	</footer>
</body>
</html>
```

Ce fichier constitue « l'ossature » de notre application. En fin de fichier, vous allez trouver une balise div qui s'ouvre et qui se referme immédiatement «`<div class="span10" ng-view></div>`». La directive AngularJS ng-view permet d'indiquer au système que du code HTML va être inclus entre les balises ouvrante et fermante div.

code HTLM ex15_1.html (dans un sous-dossier app)

```
<div class="row-fluid">
	<p class="span7">......</p>
<img class="span2" src="image/ex15_7.jpg"/>
</div>
<div class="row-fluid">
	<img class="span2" src="image/ex15_2.jpg"/>
	<p class="span5">.....</p>
	<p class="span5">...</p>
</div>
```

code HTLM ex15_2.html (dans un sous-dossier app)

```
<form class='well'>
  <fieldset>
    <legend>ipsum iam dicta quidem</legend>
    <label class="radio">
      <input type="radio" name="origine"/>
      mecum aut in M </label>
    <label for="web" class="radio">
      <input type="radio" name="origine"/>
      ima domo</label>
    <label for="textarea">amicitiam paribus </label>
    <p><input  type='email'/></p>
    <button type='submit' class='btn btn-inverse'>mertum</button>
  </fieldset>
</form>
```

code HTLM ex15_3.html (dans un sous-dossier app)

```
<h2>Qui ut huic virilem</h2>
<div class='row-fluid'>
	<img class='span3' src='image/ex15_6.jpg'/>
	<h2 class='span9'>calculos vocare amicitiam
</div>
<div class='row-fluid'>
	<p class="span4">....</p>
	<p class="span4">....</p>
	<p class="span4">....</p>
</div>
```

N.B : Par souci de place je n'ai pas fait figurer le contenu « Lorem ipsum » dans les paragraphes des fichiers 15_x.html

Le code contenu dans les fichiers ex15_1.html, ex15_2.html et ex15_3.html sera intégré au code contenu dans le fichier ex15.html (dans les conditions exposées ci-dessus).
Si l'utilisateur clique dans le menu sur Formulaire (balise <a>, URL: #/formulaire), le $routeProvider nous indique que

le code HTML contenu dans le fichier ex15_2.html sera intégré entre les balises «`<div class="span10" ng-view></div>`» du fichier ex15.html. J'attire votre intention sur le fait que l'URL contenue dans la balise <a> devra être de la forme #/xxxx (ne surtout pas oublier le #) : dans ex15.html <a href="`#/xxxx`"> entraînera un «`$routeProvider.when('/xxxx',...`» dans le fichier config_ex15.js.
AngularJS gère donc la navigation entre les pages d'une façon efficace et «intelligente» car les éléments communs à toutes les pages (header, footer et menu du fichier ex15.html) n'auront pas à être rechargé (utilisation de technologie de type Ajax).

code CSS ex15.css (dans un sous-dossier app/css/)+bootstrap twitter

```
/*Ajout pour ex15*/

h1{
  text-align:center;
  margin: 30px;
  font-size: 60px;
}
h4{
  text-align:center;
}
body{
  width:70%;
  margin-left: auto;
  margin-right: auto;

}
p{
  text-align:justify;
}
footer{
  margin-top:60px;
}
header{
  margin-bottom:60px;
}
.nav-header{
  font-size: 20px;
}
```

Attention : le fichier ex15.css est composé du bootstrap twitter et du code ci-dessus.

3e partie : Base de données
MongoDB

Stocker des données

Jusqu'à présent, les données étaient stockées dans la mémoire vive du serveur (le tableau d'objet JavaScript se trouve en mémoire vive).
Si pour une raison ou autre le serveur s'éteint, toutes les données seront définitivement perdues (la mémoire vive d'un ordinateur n'est pas permanente, on parle de mémoire volatile).
La solution est d'utiliser une mémoire dite de masse qui n'a pas besoin d'être alimenté en électricité pour conserver les données : les disques durs, les clés usb, les cartes SD sont des mémoires de masse (la capacité de stockage est aussi beaucoup plus importante que la RAM, en contrepartie, la mémoire de masse est beaucoup moins rapide (lecture/écriture) que la mémoire vive).
Ok, nous allons donc stocker sur le disque dur au lieu de simplement le garder en mémoire, mais sous quelle forme ?
Il est tout à fait possible d'écrire les données à conserver dans un fichier qui sera stocké sur le disque dur. Cette solution qui a le mérite de la simplicité peut montrer rapidement ses limites, notamment quand la structure des données devient complexe.
Nous allons donc utiliser un système extrêmement répandu : la base de données
En fin de compte, les données seront tout de même stockées dans des fichiers qui se trouveront sur un disque dur, mais la base de données va organiser toutes ses données d'une manière extrêmement complexe, ce qui permettra à l'utilisateur de lire ou d'écrire des données de façon simple et rapide. Il n'est pas nécessaire de comprendre le fonctionnement interne de la base de données pour l'utiliser.
Il existe beaucoup de systèmes différents : MySQL, Oracle, Microsoft SQL server.....toutes ces bases de données ont un point commun : elles sont de type SQL (le SQL est un langage utilisé pour effectuer des opérations sur une base de données (création, lecture, écriture...)). On parle aussi de bases de données relationnelles.
Depuis quelque temps, l'utilisation de bases de données « noSQL » (**n**ot **o**nly SQL) se développe (ces bases de données n'utilisent pas le SQL comme langage de requête), elles ont de nombreux avantages (mais aussi quelques inconvénients !) par rapport aux bases de données SQL, mais, je n'en parlerai pas ici.
Dans ce qui suit, nous allons utiliser une base de données noSQL dénommée MongoDB (DB pour Data Base). Pourquoi elle et pas une autre ? MongoDB gère les données au format JSON, ce qui nous fera toujours ça de moins à apprendre.
Pour utiliser MongoDB avec un serveur tournant sous nodeJS, nous allons devoir installer 3 éléments :
- MongoDB (http://www.mongodb.org/downloads
voir aussi pour l'installation sous Windows : http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/)
- MongoDB pour nodeJS en faisant un « npm install mongodb »
- Mongoose «npm install mongoose » (mongoose facilite l'utilisation de MongoDB avec nodeJS)

En cas de difficultés pendant cette phase d'installation, n'hésitez pas à consulter les nombreux tutoriaux présents sur le net.

Premier contact avec mongoDB (et mongoose)

MongoDB fonctionne comme un serveur. Sous linux, en théorie, il se lance automatiquement, sous Windows ce n'est pas le cas : il faut suivre les instructions données ici : http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

Nous allons étudier un premier exemple simple puisqu'il sera constitué d'uniquement un fichier (code JavaScript nodeJS). Dans cet exemple, point d'AngularJS (nous n'aurons pas de code HTML), point d'ExpressJS (nous n'aurons pas de serveur HTTP).

code JavaScript ex16n.js (nodejs)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB');
var persoSchema=mongoose.Schema({
	nom:{type:String},
	prenom:{type:String}
});
var Perso=mongoose.model('Perso',persoSchema);
var nouveauPerso=new Perso({
	nom:'Durand',
	prenom:'Pierre'
});
nouveauPerso.save(function(err){
	if (err){
		console.log("erreur d'écriture")
	}
	else{
		console.log("enregistrement effectué");
```

```
	}
});
```

La première ligne devrait vous êtes familière, je ne reviens pas dessus.
«`mongoose.connect('mongodb://localhost/maDB');`» permet de se connecter à une base de données de type mongoDB, sur la machine qui exécute le programme en cours (ici ex16n.js). Cette base de données se nomme maDB. Si maDB n'existe pas, elle est automatiquement créée.

MongoDB repose sur le concept de « schema ». Le « schema » est en quelque sorte le squelette des données que nous allons enregistrer.
«
```
var persoSchema=mongoose.Schema({
	nom:{type:String},
	prenom:{type:String}
})
```
»
Le « schema » persoSchema permettra d'enregistrer 2 attributs (« nom » et « prenom ») tous les deux de type String. Remarquez qu'ici aussi les objets JavaScript sont omniprésents.

Autre aspect important le « model ». Nous créons un « model » à partir d'un « schema » :
«`var Perso=mongoose.model('Perso',persoSchema)`»
Nous créons le « model » Perso à partir du « schema » persoSchema. Cette ligne est un peu toujours la même :
var monModel=mongoose.model('monModel', monSchema), remarquez que «monModel» intervient 2 fois.

Nous créons ensuite un nouveau document (mongoDB est une base de données noSQL dite «orientée document») :
«
```
var nouveauPerso=new Perso({
	nom:'Durand',
	prenom:'Pierre'
})
```
»
Ici aussi nous retrouvons un objet JavaScript !
Nous devons avoir une structure du type «var monDocument=new monModele({...})»

Il nous reste à enregistrer notre nouveau document dans la base de données à l'aide de la méthode save :
«
```
nouveauPerso.save(function(err){
	if (err){
		console.log("erreur d'écriture") ;
	}
	else{
		console.log("enregistrement effectué");
	}
})
```
»
La méthode save prend en paramètre une fonction de callback (fonction anonyme). Cette fonction de callback est appelée à la fin de l'opération d'enregistrement dans la base de données. Le paramètre «err» est égal à false si aucune erreur a été rencontrée lors de l'enregistrement et est égal à true dans le cas contraire.
console.log(...) permet d'afficher un message dans la console.

À *faire vous même*

Testez l'exemple ex16n.js en ouvrant une console et en tapant «node ex16n.js»(attention **sous Windows**, le serveur mongoDB devra sans doute être démarré au préalable avec un « mongod.exe » dans une autre console, vous aurez donc 2 consoles ouvertes une pour node et une pour mongod.exe).
Après moins d'une seconde « d'attente », si tout se passe bien vous devriez voir apparaître dans la console :
«enregistrement effectué» (résultat du «`console.log("enregistrement effectué");`»)

Il semble que tout s'est bien passé, mais nous allons tout de même le vérifier à l'aide du « shell mongoDB » :
Le shell va nous permettre d'effectuer des opérations sur la base de données en passant directement par la console. Pour lancer le « shell mongoDB », vous devez taper dans une console « mongo » (sous Windows «mongo.exe»).
NB : nodejs peut être arrêté, il est donc possible d'utiliser la console qui vous a servi à exécuter « ex16n.js ».

Une fois le shell lancé (vous devriez avoir un > au niveau du prompt (définition de prompt ici)), vous allez pouvoir vérifier que notre enregistrement c'est bien déroulé :
tapez : «use maDB» (afin d'indiquer à mongoDB la base de données que nous allons utiliser)
affichage attendu : «switched to db maDB»
tapez : «db.persos.find()» (attention : pas de p majuscule et ajout d'un s final à Perso (notre «model»))
affichage attendu : { "nom" : "Durand", "prenom" : "Pierre", "_id" : ObjectId("51f53700686443a50b000001"), "__v" : 0 }

_id et __v sont des attributs internes utilisés par mongoDB, je n'en parlerai pas. Vous pouvez constater que nous retrouvons bien les données enregistrées lors de l'exécution de ex16n.js. Ces données sont stockées au format JSON (enfin, pour être tout à fait exact au format BSON), elles seront donc très facilement exploitables (surtout par un programme codé en JavaScript).

Modifiez le programme ex16n.js en changeant le « nom » et le « prenom » à enregistrer. Après avoir relancé ex16n.js, consultez de nouveau la base de données comme nous venons de le faire.
Que constatez-vous ?
….....
.....
.....

Dans le shell mongoDB, quelle est l'action d'un «db.persos.remove() » ?
….....
.....
.....

Première utilisation du trio AngularJS, expressJS et mongoDB

Nous allons nous baser sur l'exemple 13, la majorité du code ci-dessous vous sera donc familier.

code JavaScript ex17n.js (nodejs)

```
//gestion de la DB
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB17');
var persoSchema=mongoose.Schema({
	nom:{type:String},
	prenom:{type:String},
	age:{type:Number},
	poids:{type:Number}
});
var Perso=mongoose.model('Perso',persoSchema);
//gestion du serveur HTTP
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex17.html');
});
app.post('/api/formulaire', function(req,res) {
	//création d'un nouveau document destiné à la DB
	var nouveauPerso=new Perso({
		nom:req.body.nom,
		prenom:req.body.prenom,
		age:req.body.age,
		poids:req.body.poids
	});
	//enregistrement dans la DB
	nouveauPerso.save(function(err){
		if (err){
			res.send('err');
		}
		else{
			res.send();
		}
	})
});
app.listen(8080);
```

code JavaScript ex17.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.traitForm=function(){
		$http.post('/api/formulaire',$scope.perso)
		.success(function(data){
			if (data=='err'){
				alert("Désolé un problème est survenu lors de l'enregistrement");
			}
			else{
				alert("La fiche a bien été enregistrée");
			}
			$scope.perso={};
		})
	}
}
```

code HTLM ex17.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex17.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<form>
		<p>Nom <input type:'text' ng-model="perso.nom"></p>
		<p>Prénom <input type:'text' ng-model="perso.prenom"></p>
		<p>Age <input type:'text' ng-model="perso.age"></p>
		<p>Poids <input type:'text' ng-model="perso.poids"></p>
		<button ng-click="traitForm()">Valider</button>
	</form>
</body>
</html>
```

Les fichiers ex17.js et ex17.html ne devraient vous poser aucun problème, je ne reviens pas dessus.
Abordons les éléments liés à la gestion de la base de données dans le fichier ex17n.js

«
```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB17');
var persoSchema=mongoose.Schema({
	nom:{type:String},
	prenom:{type:String},
	age:{type:Number},
	poids:{type:Number}
});
var Perso=mongoose.model('Perso',persoSchema);
```
»

Rien de très difficile dans cette partie : nous nous connectons à la base de données maDB17, nous créons un « schema » et à partir de ce « schema » nous créons un « model ».

«
```
app.post('/api/formulaire', function(req,res) {
	//création d'un nouveau document destiné à la DB
	var nouveauPerso=new Perso({
		nom:req.body.nom,
		prenom:req.body.prenom,
		age:req.body.age,
		poids:req.body.poids
	});
	//enregistrement dans la DB
	nouveauPerso.save(function(err){
		if (err==true){
			res.send('err');
		}
		else{
			res.send();
		}
	})
});
```
»

Nous créons un document (basé sur le « model » Perso) nommé «nouveauPerso», en utilisant les données issues du corps de la requête HTTP de type POST (les données du formulaire).
Nous sauvegardons ensuite ce document dans la base de données à l'aide de la méthode save (en gérant une réponse HTTP différente au cas où une erreur issue de la base de données devait apparaître).

À *faire vous même*
Testez et étudiez l'exemple 17. Utilisez le shell mongoDB afin de vérifier que l'enregistrement s'est bien effectué.
Que se passe-t-il si l'utilisateur entre des lettres dans la partie du formulaire prévue pour accueillir l'âge ?
….....
.....
.....
Pourquoi ?
….....
.....
.....
Faites les modifications nécessaires pour que ce problème n'arrive plus en cas d'erreur de saisie de l'âge ou du poids (vous pourrez par exemple utiliser la méthode isNaN).

Lecture dans la base de données : les requêtes

Nous allons maintenant lire dans la base de données et gérer l'affichage des données. Nous allons, une nouvelle fois, reprendre l'exemple 13.

code JavaScript ex18n.js (nodejs)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB17');
var persoSchema=mongoose.Schema({
	nom:{type:String},
	prenom:{type:String},
	age:{type:Number},
	poids:{type:Number}
});
var Perso=mongoose.model('Perso',persoSchema);
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex18.html');
});
app.get('/api/affiche', function(req, res) {
	Perso.find(null)
	.exec(function(err,fiches){
		if (err==true){
			res.send('err');
		}
		else{
			res.json(fiches);
		}
	})
});
app.post('/api/formulaire', function(req,res) {
	var nouveauPerso=new Perso({
		nom:req.body.nom,
		prenom:req.body.prenom,
		age:req.body.age,
		poids:req.body.poids
	});
	nouveauPerso.save(function(err){
		if (err){
			res.send('err');
		}
		else{
			res.send();
		}
	})
});
```

```
app.listen(8080);
```

code JavaScript ex18.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.affiche=function(){
		$http.get('/api/affiche')
		.success(function(data){
			if (data!='err'){
				$scope.listePerso=data;
			}
		})
	}
	$scope.traitForm=function(){
		$http.post('/api/formulaire',$scope.perso)
		.success(function(data){
			if (data=='err'){
				alert("Désolé un problème est survenu lors de l'enregistrement");
			}
			else{
				$scope.perso={};
				$scope.affiche();
			}
		})
	}
	$scope.affiche();
}
```

code HTLM ex18.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
	<meta Charest="UTF-8">
	<title>Test AngularJS</title>
	<script src="lib/angular.min.js"></script>
	<script src="javascript/ex18.js"></script>
</head>
<body ng-app ng-controller="monControl">
	<form>
		<p>Nom <input type:'text' ng-model="perso.nom"></p>
		<p>Prénom <input type:'text' ng-model="perso.prenom"></p>
		<p>Age <input type:'text' ng-model="perso.age"></p>
		<p>Poids <input type:'text' ng-model="perso.poids"></p>
		<button ng-click="traitForm()">Valider</button>
	</form>
	<h1>Liste des fiches</h1>
	<div ng-repeat="perso in listePerso">
		<p>Nom : {{perso.nom}}</p>
		<p>Prénom : {{perso.prenom}}</p>
		<p>Age : {{perso.age}}</p>
		<p>Poids : {{perso.poids}}</p>
	</div>
</body>
</html>
```

Encore une fois, très peu de nouveauté dans cet exemple. La seule vraie nouveauté se situe dans le fichier ex18n.js :
«`app.get('/api/affiche', function(req, res) {`

```
	Perso.find(null)
	.exec(function(err,fiches){
		if (err==true){
			res.send('err');
		}
		else{
			res.json(fiches);
		}
	})
});
```

»

Comme précédemment, cette partie va nous permettre d'envoyer une réponse HTTP au client (avec le «`res.json(fiches);`». À la différence de l'exemple 13, les données renvoyées vers le client sont issues d'une base de données.

La récupération de ces données s'effectue grâce à la structure suivante :

«
```
Perso.find(null)
.exec(function(err,fiches){
	if (err==true){
		res.send('err');
	}
	else{
		res.json(fiches);
	}
})
```
»

Le « find » devrait vous rappeler le shell mongoDB. Une différence à noter par rapport au shell mongoDB, nous avons ici «model.find(...)» (avec dans notre exemple « model » = « Perso ») : nous laissons le P majuscule et nous n'ajoutons pas de s à la fin du nom du « model » (à la différence du shell).
La méthode find prend un paramètre : la requête . Ici, nous récupérons toutes les entrées, sans faire aucune sélection, d'où la présence du « null ».
Notez la structure : «`Perso.find(...) .exec(function(...){ ...} )`».
La méthode « exec » prend en paramètre une fonction de callback (exécutée une fois que la base de données a répondu à la requête).
Cette fonction de callback prend 2 paramètres : err (même signification que précédemment) et fiches (mais vous pouvez choisir le nom qui vous plaira) qui est un tableau contenant les données (sous forme d'objet JavaScript) retournées par la base de données. Il est très important de remarquer que le « fiches » de l'exemple 18 a exactement la même nature que le «ficheInfo » de l'exemple 13 (tableau d'objet JavaScript), ce qui, bien évidemment, facilite le traitement ultérieur des données et le retour de ces données vers le client avec le «`res.json(fiches);`».

À *faire vous même*
Testez et étudiez l'exemple 18.

Faire des requêtes plus complexes

Il est possible de sélectionner certaines entrées (données) et de laisser tomber les autres (nous renverrons alors au client un tableau contenant uniquement les données pertinentes).
Voici quelques exemples de requêtes se basant sur l'exemple 18 (par souci de clarté je n'ai pas mis le contenu de la fonction callback de la méthode « exec ») :

`Perso.find({nom:'toto'}) .exec(...)` => cette requête renverra uniquement les personnes ayant pour nom toto

À *faire vous même*

Que fera cette requête ?

```
Perso.find({prenom:'titi'}) .exec(...)
```

….....

`Perso.find({age:{$gt:18}}) .exec(...)=>` cette requête renverra uniquement les personnes qui ont plus de 18 ans (age>18, le signe $gt signifie >)

`Perso.find({age:{$lt:18}}) .exec(...)=>` cette requête renverra uniquement les personnes qui ont moins de 18 ans (age<18, le signe $lt signifie <)

`Perso.find({poids:{$gt:50,$lt:90}}) .exec(...)=>` cette requête renverra uniquement les personnes qui ont un poids compris entre 50 et 90 kg.

Il est tout à fait possible de combiner plusieurs contraintes de recherche à l'aide de la méthode « where » :

```
Perso.find({prenom :'Pierre'})
.where('age').gt(18).lt(45)
.exec(...)
```

Cette requête renverra uniquement les fiches des personnes se prénommant Pierre et ayant entre 18 et 45 ans (notez l'écriture des signes < et > : where('age').gt(18) => age>18 et where('age').lt(45)=>age<45)

Il est possible de combiner le nombre de « where » que vous désirez

```
Perso.find({prenom :'Pierre'})
.where('age').gt(18).lt(45)
.where('nom').equals('Toto')
.exec(...)
```

Désigne les personnes se nommant Pierre Toto et ayant entre 18 et 45 ans

Il est possible de limiter le nombre de résultats renvoyé avec « limit»

```
Perso.find({nom :'Toto'})
.limit(2)
.exec(...)
```

Ne renverra que les 2 premiers résultats (les autres seront ignorés)

Il y a beaucoup d'autres possibilités de requêtes, je vous conseille de consulter la documentation officielle de Mongoose (voir l'annexe).

code JavaScript ex19n.js (nodejs)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB17');
var persoSchema=mongoose.Schema({
	nom:{type:String},
	prenom:{type:String},
	age:{type:Number},
	poids:{type:Number}
});
var Perso=mongoose.model('Perso',persoSchema);
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex19.html');
});
…
…
…
…
…
…
…
…
…
…
…
app.listen(8080);
```

code JavaScript ex19.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.pasResu=false;
	$scope.resu=false;
	$scope.erreur=false;
	$scope.affiche=function(){
		$http.get('/api/affiche/'+$scope.recherche)
		.success(function(data){
			$scope.erreur=false;
			$scope.recherche="";
			if (data!='err'){
				if (data!=''){
					$scope.listePerso=data;
					$scope.resu=true;
					$scope.pasResu=false;
```

```
                        }
                        else{
                                $scope.resu=false;
                                $scope.pasResu=true;
                        }
                }
                else{
                        $scope.resu=false;
                        $scope.pasResu=false;
                        $scope.erreur=true;
                }
        })
    }
}
```

code HTLM ex19.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
    <meta Charest="UTF-8">
    <title>Test AngularJS</title>
    <script src="lib/angular.min.js"></script>
    <script src="javascript/ex19.js"></script>
</head>
<body ng-app ng-controller="monControl">
    <h1>Rechercher des fiches</h1>
    <form>
        <p>Nom <input type:'text' ng-model="recherche"></p>
        <button ng-click="affiche()">Valider</button>
    </form>
    <p ng-show='pasResu'>désolé, aucun résultat</p>
    <p ng-show='erreur'>désolé, une erreur est survenue</p>
    <p ng-show='resu'>Résulat(s) :</p>
    <div ng-repeat="pers in listePerso" ng-show='resu'>
        <p>Nom : {{pers.nom}}</p>
        <p>Prénom : {{pers.prenom}}</p>
        <p>Age : {{pers.age}}</p>
        <p>Poids : {{pers.poids}}</p>
    </div>
</body>
</html>
```

À *faire vous même*

Après avoir étudié l'exemple 19, complétez le fichier ex19n.js afin de mettre en place un système de recherche par nom (l'utilisateur entre un nom, le système affiche toutes les fiches correspondant à ce nom). Une fois terminée, testez votre réponse.

Supprimer des entrées (requête HTTP de type DELETE)

Si à la place de la méthode « find », si vous utilisez la méthode « remove », toutes les entrées (le fiches) concernées par la requête seront effacées de la base de données (il est possible de faire de requêtes complexes avec remove comme avec find). Avec remove, la fonction de callback (exec) ne prend qu'un paramètre (err).

code JavaScript ex20n.js (nodejs)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB17');
var persoSchema=mongoose.Schema({
    nom:{type:String},
    prenom:{type:String},
    age:{type:Number},
    poids:{type:Number}
});
var Perso=mongoose.model('Perso',persoSchema);
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
```

```
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex20.html');
});
app.get('/api/affiche/:nom', function(req, res) {
	Perso.find({nom:req.params.nom})
	.exec(function(err,fiches){
		if (err==true){
			res.send('err');
		}
		else{
			res.json(fiches);
		}
	})
});
app.delete('/api/suppr/:id', function(req, res) {
	Perso.remove({_id:req.params.id})
	.exec(function(err){
		if (err==true){
			res.send('err');
		}
		else{
			res.send('');
		}
	})
});
app.listen(8080);
```

code JavaScript ex20.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.pasResu=false;
	$scope.resu=false;
	$scope.erreur=false;
	$scope.resuSup=false;
	$scope.erreurSup=false;
	$scope.affiche=function(){
		$http.get('/api/affiche/'+$scope.recherche)
		.success(function(data){
			$scope.recherche="";
			$scope.resuSup=false;
			$scope.erreurSup=false;
			if (data!='err'){
				$scope.erreur=false;
				if (data!=''){
					$scope.listePerso=data;
					$scope.resu=true;
					$scope.pasResu=false;
				}
				else{
					$scope.resu=false;
					$scope.pasResu=true;
				}
			}
			else{
				$scope.resu=false;
				$scope.pasResu=false;
				$scope.erreur=true;
			}
		})
	}
	$scope.supr=function(fiche){
		$http.delete('/api/suppr/'+fiche._id)
		.success(function(data){
			$scope.recherche="";
			$scope.resu=false;
			$scope.pasResu=false;
			$scope.erreur=false;
			if (data!='err'){
				$scope.resuSup=true;
```

```
                        $scope.erreurSup=false;
                }
                else{
                        $scope.resuSup=false;
                        $scope.erreurSup=true;
                }
        })
    }
}
```

code HTLM ex20.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
    <meta Charest="UTF-8">
    <title>Test AngularJS</title>
    <script src="lib/angular.min.js"></script>
    <script src="javascript/ex20.js"></script>
</head>
<body ng-app ng-controller="monControl">
    <h1>Rechercher des fiches</h1>
    <form>
        <p>Nom <input type:'text' ng-model="recherche"></p>
        <button ng-click="affiche()">Valider</button>
    </form>
    <p ng-show='pasResu'>désolé, aucun résultat</p>
    <p ng-show='erreur'>désolé, une erreur est survenue</p>
    <p ng-show='resu'>Résulat(s) :</p>
    <div ng-repeat="pers in listePerso" ng-show='resu'>
        <p>Nom : {{pers.nom}}</p>
        <p>Prénom : {{pers.prenom}}</p>
        <p>Age : {{pers.age}}</p>
        <p>Poids : {{pers.poids}}</p>
        <a href='#' ng-click='supr(pers)'>supprimer</a>
    </div>
    <p ng-show='erreurSup'>désolé, une erreur est survenue</p>
    <p ng-show='resuSup'>La fiche a bien été supprimée</p>
</body>
</html>
```

À *faire vous même*

Étudiez attentivement l'exemple 20. Comme déjà dit, _id est généré automatiquement par mongoDB, chaque entrée possède un _id qui lui est propre. Il peut donc être utilisé pour cibler une entrée (sans crainte de se tromper).

Modifier des entrées (requête HTTP de type PUT)

Il est possible de modifier des entrées grâce à la méthode update. Cette méthode update fonctionne sur le principe suivant : update({condition},{modification}).
un exemple : update({nom : 'toto'}, {age:18}) => l'entrée ayant nom=toto sera mis à jour, maintenant age sera égale à 18.
Notez que nous utilisons la méthode PUT pour faire notre requête HTTP de mis à jour.

code JavaScript ex21n.js (nodejs)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/maDB17');
var persoSchema=mongoose.Schema({
    nom:{type:String},
    prenom:{type:String},
    age:{type:Number},
    poids:{type:Number}
});
var Perso=mongoose.model('Perso',persoSchema);
var express = require('express');
var app = express();
app.use('/JavaScript', express.static(__dirname + '/app/javascript'));
```

```
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use(express.bodyParser());
app.get('/', function (req, res) {
	res.sendfile(__dirname + '/app/ex21.html');
});
app.get('/api/affiche/:nom', function(req, res) {
	Perso.find({nom:req.params.nom})
	.exec(function(err,fiches){
		if (err==true){
			res.send('err');
		}
		else{
			res.json(fiches);
		}
	})
});
app.put('/api/edit/:id',function(req,res){
	Perso.update({_id:req.params.id},
	{nom:req.body.nom,prenom:req.body.prenom,age:req.body.age, poids:req.body.poids})
	.exec(function(err){
		if (err==true){
			res.send('err');
		}
		else{
			res.send();
		}
	})
})
app.listen(8080);
```

code JavaScript ex21.js (dans un sous-dossier app/JavaScript/)

```
function monControl($scope,$http){
	$scope.pasResu=false;
	$scope.resu=false;
	$scope.erreur=false;
	$scope.edit=false;
	$scope.resuEd=false;
	$scope.erreurEd=false;
	$scope.affiche=function(){
		$http.get('/api/affiche/'+$scope.recherche)
		.success(function(data){
			$scope.resuEd=false;
			$scope.erreurEd=false;
			$scope.recherche="";
			if (data!='err'){
				$scope.erreur=false;
				if (data!=''){
					$scope.listePerso=data;
					$scope.resu=true;
					$scope.pasResu=false;
				}
				else{
					$scope.resu=false;
					$scope.pasResu=true;
				}
			}
			else{
				$scope.resu=false;
				$scope.pasResu=false;
				$scope.erreur=true;
			}
		})
	}
	$scope.edition=function(p){
		$scope.edit=true;
		$scope.resu=false;
		$scope.fiche=p;
	}
	$scope.traitEd=function(){
		if ($scope.perso.nom==null){
```

```
                $scope.perso.nom=$scope.fiche.nom;
            }
            if ($scope.perso.prenom==null){
                $scope.perso.prenom=$scope.fiche.prenom;
            }
            if ($scope.perso.age==null){
                $scope.perso.age=$scope.fiche.age;
            }
            if ($scope.perso.poids==null){
                $scope.perso.poids=$scope.fiche.poids;
            }
            $http.put('/api/edit/'+$scope.fiche._id,$scope.perso)
            .success(function(data){
                $scope.resu=false;
                $scope.pasResu=false;
                $scope.erreur=false;
                $scope.edit=false;
                $scope.perso="";
                if (data!='err'){
                    $scope.resuEd=true;
                    $scope.erreurEd=false;
                }
                else{
                    $scope.resuEd=false;
                    $scope.erreurEd=true;
                }
            })
        }
}
```

code HTLM ex21.html (dans un sous-dossier app)

```
<!doctype html>
<html lang="fr">
<head>
        <meta charset="UTF-8">
        <title>Test AngularJS</title>
        <script src="lib/angular.min.js"></script>
        <script src="javascript/ex21.js"></script>
</head>
<body ng-app ng-controller="monControl">
        <h1>Rechercher des fiches</h1>
        <form>
            <p>Nom <input type:'text' ng-model="recherche"></p>
            <button ng-click="affiche()">Valider</button>
        </form>
        <p ng-show='pasResu'>désolé, aucun résultat</p>
        <p ng-show='erreur'>désolé, une erreur est survenue</p>
        <p ng-show='resu'>Résulat(s) :</p>
        <div ng-repeat="pers in listePerso" ng-show='resu'>
            <p>Nom : {{pers.nom}}</p>
            <p>Prénom : {{pers.prenom}}</p>
            <p>Age : {{pers.age}}</p>
            <p>Poids : {{pers.poids}}</p>
            <a ng-click='edition(pers)' href="#">Editer cette fiche</a>
        </div>
        <h1 ng-show='edit'>Edition de la fiche</h1>
        <form ng-show='edit'>
            <p>Nom <input type:'text' placeholder='{{fiche.nom}}' ng-model="perso.nom"></p>
            <p>Prénom <input type:'text' placeholder='{{fiche.prenom}}'ng-model="perso.prenom">
            </p>
            <p>Age <input type:'text' placeholder='{{fiche.age}}' ng-model="perso.age"></p>
            <p>Poids <input type:'text' placeholder='{{fiche.poids}}' ng-model="perso.poids"></p>
            <button ng-click="traitEd()">Valider</button>
        </form>
        <p ng-show='erreurEd'>désolé, une erreur est survenue</p>
        <p ng-show='resuEd'>La fiche a bien été modifiée</p>
</body>
</html>
```

À *faire vous même*

Étudiez attentivement l'exemple 21 et n'hésitez pas à poser des questions

ANNEXE 1

Même si nous avons abordé beaucoup de notions, nous n'avons fait que survoler la question. Pour aller plus loin, je vous propose ici quelques liens vers des ressources de qualités (pour la plupart anglophone) :

AngularJS

grakikart (fr) : http://www.grafikart.fr/tutoriels/angularjs/angularjs-todo-369
présentation de Thierry Lau (fr) : https://www.youtube.com/watch?v=8Xmh54KO5K8&list=WLB1993796CDF62730
série de vidéos proposée sur le site officiel (en)**:** http://www.youtube.com/user/angularjs
cours vidéo site egghead.io (en): http://www.egghead.io/

blog francophone AngularJS (fr) : http://www.frangular.com/
tutoriaux sur le site officiel (en) : http://docs.angularjs.org/tutorial/index

nodeJS (expressJS)

cours vidéos expressJS (en) : http://www.youtube.com/watch?v=JhtdC86wWIk&list=TLd5L5qKmw5vI
cours vidéos nodeJS (en) : http://nodetuts.com/

site officiel nodeJS (en) : http://nodejs.org/
site officiel expressJS (en) : http://expressjs.com/
site du zéro (fr) : http://www.siteduzero.com/informatique/tutoriels/des-applications-ultra-rapides-avec-node-js

mongoDB (mongoose)

site officiel mongoDB (en) : http://www.mongodb.org/
site officiel mongoose (en) : http://mongoosejs.com/

tutorial atinux (fr) : http://www.atinux.fr/2011/10/15/tutoriel-sur-mongoose-mongodb-avec-node-js/