Architecture MEAN avec **Angular 2**

(MEAN = MongoDB, Express, Angular et Node.js)

-

MongoDB : insertion de données, interrogations, interfaçage avec Node

-

Node : routes REST, renvoi de données formatées en JSON, connexion à MongoDB

-

Angular 2 : architecture, composants, modules, templates, injections et routage

Pierre Pompidor

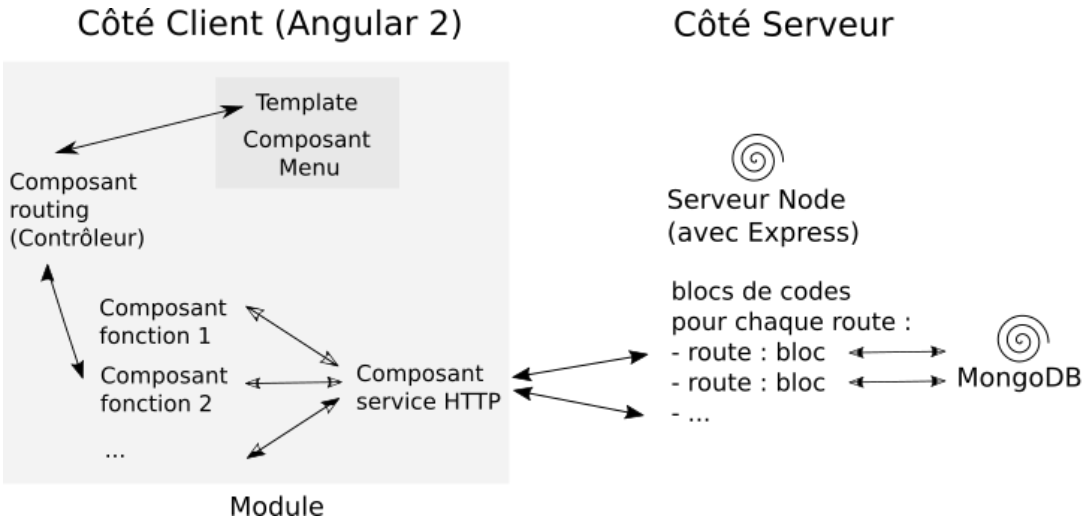
Table des matières

1 Introduction générale :

1.1 Architecture MEAN d'une application web :

Le graphique ci-dessous présente l'architecture d'une application web construite :
- sur le côté serveur avec **Node.js** qui va :
  - répondre à des **routes REST**
  - faire appel à une base de données gérée par **MongoDB** pour accéder à des **documents** gérées dans des **collections**
  - opérer des traitements de filtres avant de renvoyer les données sous formes d'objets javascript sérialisés (du **JSON**)
- sur le côté client avec **Angular 2** qui :
  - gère un seul module (il y aura plusieurs modules dans le projet à développer (voir ci-après))
  - gère un composant service injecté dans différents composants
  - gère un composant qui fait office de contrôleur.

Les notions **de modules, composants et d'injection de services** (cas spécifique de l'injection de dépendances) sont les points les plus intéressants d'Angular 2.



1.2 Le projet :

Le projet consistera à créer une maquette d'une application d'e-commerce respectant les fonctionnalités suivantes :
- cette maquette sera construite via l'architecture MEAN ;
- la base de données MongoDB comprendra au moins deux collections :
  - une collection mémorisant les **produits en ventes** ;
  - une collection mémorisant les **paniers des clients** en cours.
- Angular 2 gérera deux modules :
  - un module permettant de rechercher des produits par :
    - la liste des types
    - la liste des marques
    - un type, une marque spécifique...

    des raffinements supplémentaires suite à une première recherche (par exemple sur les prix) devront être aussi possibles.
  - un module permettant d'ajouter ou d'enlever des produits dans le panier.
    Ces modules pourront donc être ainsi réutilisés dans d'autres applications.

2 Node.js :

Node.js est un environnement permettant d'exécuter du code Javascript hors d'un navigateur. Son architecture est événementielle et modulaire. Il est orienté réseaux car il possède pour les principaux systèmes d'exploitation (Mac OS, Unix et Windows) des modules réseaux (DNS, HTTP, TCP, TLS/SSL, UDP). Il est réputé pour sa légèreté (vu sa modularité) et son efficacité.
(Dans ce qui suit, nous dzsignerons Node.js par Node.)

Les composants Angular 2 de notre application client se fourniront donc en données auprès de services web gérés par Node.

Nous utiliserons deux modules de Node :
— **express** pour accéder à Node via des URL au format REST ;
— **fs** pour accéder à des fichiers (par exemple au format JSON) stockés sur le serveur.

Nous utiliserons aussi une connexion à une base de données NoSQL avec **MongoDB**.

2.1 Installation de Node.js :

Procédure complète sur `http://doc.ubuntu-fr.org/nodejs`.
(Node est déjà installé sur les machines de TP (enfin, je l'espère).

```
sudo apt-get update
sudo apt-get install nodejs
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

npm est le gestionnaire de modules de Node (il est installé avec Node).
Les modules seront installés :
— dans `/usr/local/lib/node_modules` si **l'option -g** est utilisée : `npm install -g ...`
— ou sinon dans le répertoite courant.

2.2 "Hello word" avec Node :

Vous remarquerez dans le code suivant l'utilisation d'**un module** avec l'instruction **require**.

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

var server = http.createServer(function(request, response){
    response.end('Hello World de Node\n');
});

server.listen(8888);
```

2.3 Module Express :

Ce module permet d'accéder au serveur Node en utilisant **des URL au format REST**.
Ce format permet de masquer l'implémentation de l'action en adoptant une syntaxe orientée données.

Comparez la syntaxe de cette URL "classique" mettant en oeuvre un script PHP :

```
localhost/afficheProduitsParMarque.php?marque=Apple
```

avec celle-ci :

```
localhost/produits/marque/Apple
```

Les "puristes" utilisent aussi la méthode HTTP pour désigner la méta-action à effectuer :

```
  méthode PUT    : création des données (create)
  méthode GET    : accès aux données (read)
  méthode POST   : modification des données (update)
  méthode DELETE : suppression des données (delete)
```

Cette pratique s'appelle **CRUD**.

2.3.1 Installation du module Express :

```
sudo npm install express
export NODE_PATH=/usr/local/lib/node_modules (dans le .bashrc)
```

2.3.2 "Hello word" avec Node et Express :

Un "hello word" avec Node+express (fichier bonjourDeNode.js) :

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

app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Bonjour');
});

app.listen(8888);
```

2.3.3 Test du serveur Node avec Express :

— dans le terminal, lancez le serveur : `node bonjourDeNode.js` ou `nodejs bonjourDeNode.js`

— dans votre navigateur, appelez le serveur : `localhost:8888` :
→ affichage de "Bonjour"

2.4 Module fs :

Ce module permet au serveur node d'accéder au système de fichiers et de permettre par exemple à Node de renvoyer un fichier JSON. `sudo npm install -g express`

2.4.1 Création d'un fichier de données JSON (Produits.json) :

Créons par exemple une collection (liste d'objets) stockant des informations sur des produits en ventes sur un site d'e-commerce :

```
[
  {'type': 'téléphone', 'marque': 'Apple',   'nom': 'iphone'},
  {'type': 'téléphone', 'marque': 'Samsung', 'nom': 'Galaxy S7'},
  {'type': 'ipod',      'marque': 'Apple',   'nom': 'ipod touch'}
]
```

A priori, il n'est pas nécessaire d'encadrer les noms des propriétés d'un objet javasscript par des guillemets, mais je vous recommande quand même de le faire, certains parsers l'exigeant ;) ...

2.4.2 Renvoi au client d'un fichier contenant des données formatées en JSON :

Voici un exemple de serveur Node renvoyant au client un fichier contenant des données formatées en JSON. Ce fichier s'appelle *Produits.json* et le serveur le renvoie sur la route */produits*.

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

app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Bonjour');
});

xapp.get('/produits', function(req, res) {
    res.setHeader('Content-Type', 'application/json');
    res.setHeader('Access-Control-Allow-Origin', '*');
    var readable = fs.createReadStream("Produits.json");
    readable.on('open', function()  { readable.pipe(res);
                                      console.log("Liste des produits renvoyée"); });
    readable.on('error', function() { res.end("[]"); });
});

app.listen(8888);
```

2.4.3 Tests du serveur Node avec Express et fs :

— dans le terminal, lancez le serveur : `node bonjourDeNode.js`

— dans votre navigateur, appelez le serveur : `localhost:8888/produits` :
→ affichage de la collection JSON

2.4.4 Renvoi au client de données issues d'un fichier contenant des données formatées en JSON :

Cet exemple reprend l'exemple précédent mais la route est ici accompagnée d'un paramètre *marque* et le serveur parse la collection *Produits* désérialisée à partir du fichier *Produits.json* pour ne renvoyer que les données correspondant au paramètre.

```
app.get('/produits/:marque', function(req, res) {
    var marqueAChercher = req.params.marque;
    console.log("serveur node : /produits/"+marqueAChercher);
    var listeProduits = JSON.parse(fs.readFileSync("Produits.json", "UTF-8"));
    var produitsDeLaMarque = [];
    for (var i=0; i < listeProduits.length; i++) {
         if (listeProduits[i].marque == marqueAChercher) {
             produitsDeLaMarque.push( listeProduits[i] ); }
    }
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Content-Type', 'application/json');
    var json = JSON.stringify(produitsDeLaMarque);
    console.log(" -> json : "+json);
    res.end(json);
});
app.listen(8888);
```

3 Aperçu de MongoDB

MongoDB est une **base de données NoSQL** qui gère des données JSON (dans un format optimisé (BSON) : c'est donc une option idéale pour créer une application web javascriptienne !
Voici le lien officiel : `https://docs.mongodb.org/manual/tutorial`

3.1 Installation de MongoDB :

Attention : (a priori) MongoDB est déjà installée sur les machines des salles de TP !

```
echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse"
     | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
sudo apt-get install -y mongodb-org
```

3.2 Lancement du service et vérfication de celui-ci :

```
sudo service mongod start
(MongoDB tourne par défaut sur le port 27017)
sudo netstat -antup
```

3.3 Utilisation de MongoDB en ligne de commandes :

Pour gérer MongoDB en ligne de commandes : `mongo`

3.3.1 Liste des bases de données :

`show dbs`

3.3.2 Création d'une base :

`use NomDeLaBase`

3.3.3 Création d'une collection :

La **collection** sera créée lors de la création des premiers **documents**.
La collection peut être (de loin) considérée comme étant une table, et un document un tuple de celle-ci.

`db.NomDeLaCollection.insert(collection)`

Exemple :

```
db.Produits.insert([{'type': 'téléphone', 'marque':'Apple',   'nom': 'iphone'},
                    {'type': 'téléphone', 'marque':'Samsung', 'nom': 'Galaxy S7'},
                    {'type': 'téléphone', 'marque':'Samsung', 'nom': 'ipod touch'}
                   ])
```

Les documents peuvent aussi être insérés à partir d'un fichier JSON :

```
mongoimport --db nomDeLaBase --collection nomDeLaCollection --file fichier.json
```

3.3.4 Interrogation d'une collection :

L'instruction **find** peut comporter en paramètre un objet qui va opérer un filtre.

`db.Produits.find(ObjetDeSelection)`

Exemples :

```
db.Produits.find()
db.Produits.find({'marque' : 'Apple'})
```

3.3.5 Suppression d'un document :

```
db.NomDeLaCollection.remove(ObjetDeSelection)
```

3.3.6 Suppression d'une collection :

```
db.NomDeLaCollection.drop()
```

3.4 Installation du module MongoDB pour Node :

```
npm install mongodb --save
```

3.5 Script Node de vérification de la connexion

```
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');

var url = 'mongodb://localhost:27017/test';
// La base test sera créée
MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  console.log("Connected correctly to server.");
  db.close();
});
```

3.6 Exemple d'un script Node/Mongo d'insertion de données :

Dans :
— une base dont le nom est spécifiée dans l'URL,
— une **collection** gérée par la base
— insertion d'objets appelés **documents**

```
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');
var ObjectId = require('mongodb').ObjectID;
var url = 'mongodb://localhost:27017/VentesEnLigne';

var insertDocument = function(db, callback) {
   db.collection('Produits').insert(
       [ {'type': 'téléphone', 'marque': 'Apple', 'nom':'iphone'},
         {'type': 'téléphone', 'marque': 'Samsung', 'nom':'Galaxy S7'},
         {'type': 'ipod', 'marque': 'Apple', 'nom': 'ipod touch'}
       ]
     , function(err, result) {
         assert.equal(err, null);
         console.log("Insertions de documents dans la collection Produits.");
         callback(result);
   });
};

MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  insertDocument(db, function() {
      db.close();
  });
});
```

3.7 Exemple d'un script Node/Mongo d'interrogation de données :

L'interrogation se fait via les correspondances **propriété → valeur**
La première requête liste tous les documents de **la collection Produits**,
La seconde requête filtre les documents sur une valeur de propriété.
Les résultats sont affichés dans le terminal.

```
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');
var ObjectId = require('mongodb').ObjectID;
var url = 'mongodb://localhost:27017/VentesEnligne';

var findProduits = function(db, search, callback) {
   var cursor = db.collection('Produits').find( search );
   cursor.each(function(err, doc) {
      assert.equal(err, null);
      if (doc != null) {
          //console.dir(doc);
          for (var p in doc) {
             console.log(p+" : "+doc[p]);
          }
      } else { callback(); }
      console.log("\n");
   });
};

MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  findProduits(db, {}, function() {
      db.close();
  });
});

MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  findProduits(db, {"marque": "Apple"}, function() {
      db.close();
  });
});
```

4 ES6 (Javascript 2015) :

ECMASCRIPT est un ensemble de normes de programmation.
La version 6 (dite abusivement "Javascript 2015") a apporté son lot de nouvelles syntaxes : trois mises en oeuvre par Node vont nous intéresser particulièrement :
Le spécificateur let, les fat arrows et les observables.

4.1 Le spécificateur let :

Le **spécificateur let** permet de déclarer une variable dont la portée est le bloc d'instructions.

```
for (let i=0; i < 10; i++) { ... }
```

4.2 La fat arrow :

La fat arrow (=>) est un raccourci syntaxique pour les fonctions de callback.

Par exemple :
```
var liste = [1, 2, 3];
liste.forEach(function callback(valeur) {
   console.log(valeur);
});
```

devient :
```
liste.forEach((valeur) => console.log(valeur));
```

mais elle modifie aussi le contexte de `this`.
Reprenons le même exemple (à tester via un interpréteur JS respectant complètement la norme ES6).

```
var liste = [1, 2, 3];
liste.forEach(function callback(valeur) {
  console.log(valeur+"/"+this.length);  // this.length ne fonctionne pas
                                        // "this" ne référence plus la liste
                                        // (le contexte a changé)
});
```

```
liste.forEach((valeur) => console.log(valeur+"/"+this.length));
// "this" référence bien la liste
```

4.3 Les observables :

Les observables sont des objets qui vont délivrer un flux asynchrone de données.
Une méthode **subscribe** sera déclenchée à chaque fois que le service va délivrer un flux complet de données.

Par exemple Angular 2 peut mettre en oeuvre le **service http** dont la **méthode getJson** renvoie **un observable** :
```
http.getJSON(URL).subscribe(res => this.resultat = res,
                            err => console.error(err),
                            () => console.log('done'));
```

La méthode subscribe gère trois fonctions de callback :
— la première est appelée avec les données en paramètre ;
— la seconde est appelée quand une erreur survient ;
— la troisième est appelée quand le flux de données est terminée.

5 Angular 2

5.1 Introduction :

Le **framework d'application** web Javascript **Angular 2** créé par Google permet :

— à l'instar d'**AngularJS** (la première version d'Angular), de découpler les interfaces (la **vue** du patron de conception MVC) des traitements métiers et des données (les données et les traitements constituant le **modèle** du patron de conception MVC) ;

— d'utiliser des surcouches au langage Javascript pour faire de la programmation objets en utilisant des **classes** et le **typage statique** (dans ce support sera introduite la surcouche **TypeScript** mais Angular 2 accepte aussi la surcouche **Dart** créée par Google) ;

— de créer de vrais **composants** métiers ainsi que des **modules** regroupant des composants ;

— en mode développement, de regénérer les pages web dès qu'une modification est effectuée sur les codes sources (grâce à des scrutateurs = les **watchers**).

Attention : Angular 2 est vraiment différent de la première version d'Angular.
Angular 2 vise à produire du logiciel réutilisable via la création de composants regroupés en modules.

5.2 Installation :

Pouvant être naturellement couplé avec le **serveur Javascript Node.js** (voir la seconde section de ce support de cours), Angular 2 s'installe via le **gestionnaire de paquets de NodeJS** appelé **npm**.

Voici la démarche à suivre sous Linux/Ubuntu :

Préliminaire :
sudo apt-get install nodejs npm
nodejs et npm sont déjà installés sur les machines de TP (enfin, je l'espère)

Ensuite vu la multitude de paquets à installer et la complexité des dépendances, je vous conseille de cloner un projet de base déjà réalisé, par exemple le quickstart TypeScript :
https ://angular.io/docs/ts/latest/quickstart.html.
→ attention : ce polycopié datant du 1er septembre 2016, il se peut donc que les codes que vous allez télécharger et leurs dépendances soient un peu différents, mais vous devriez pouvoir les adapter sans souci.

Pour le cloner, créez un dossier (par exemple *angular2-ts-quickstart*) et utilisez **git** :
mkdir <nomProjet>
git clone https ://github.com/angular/quickstart.git <nomProjet>
Votre projet peut simplement s'appeler quickstart.

Déplacez-vous dans le dossier correspondant au projet (angular2-ts-quickstart d'origine mais vous l'avez sans doute renommé) : **cd <nomProjet>**

Puis lancez l'installation des paquets décrits dans le fichier de configuration **package.json** (ce fichier va décrire toutes les dépendances nécessaires à l'instar des fichiers *manifest* sous Java) :

```
npm install
npm postinstall
```

Attention :
— cela peut être long... ;
— de vilaines erreurs peuvent apparaître en rouge sang ;
— mais ce qui est important est la création d'un dossier **node_modules** contenant les modules nécessaires à l'exécution de votre application

Les paquets npm seront donc installés localement dans le répertoire courant. Si vous êtes administrateur sur votre machine, vous pouvez aussi les installer globalement (option -g de npm).

Avant de continuer, nous allons découvrir la surcouche au langage Javascript qu'est **TypeScript**.

5.3 TypeScript :

TypeScript est une surcouche au langage Javascript créée par Microsoft en 2012.
Il permet de :

- typer les variables lors de leurs déclarations (typage statique) ;
- créer des **classes** et les instancier ;
- gérer des interfaces (au sens de la programmation objets) ;
- gérer des spécificateurs d'accès (**public**, **protected**, **private**) ;
- gérer des paramètres optionnels (en ajoutant un point d'interrogation accolé à leurs noms).

Le code Typescript va être transformé en code Javascript via un *transpiler* (pour ma part j'aime bien utiliser le terme transcodeur) nommé **tsc** :
par exemple, en ligne de commande :

```
tsc code.ts
```

où *code.ts* est un code TypeScript → va générer *code.js* qui sera un code javascript "pur".

Remarquez ainsi que vous pouvez donc mélanger du code Typescript avec du code Javascript et que cela peut conduire à des sommets fascinants d'illisibilité.

5.3.1 Typage statique des variables :

Syntaxe du typage d'une variable : var variable **:type** = valeur
Une espace peut être mise optionnellement avant ou après les " :" (pour ma part j'aime bien en mettre une avant), **mais surtout n'utilisez pas le spécificateur var.**

Exemples :

```
i :number = 1;
bool :boolean = true;
nom :string = "Zorro";
```

Bien entendu, la valeur de retour d'une fonction peut être aussi typée :

```
function Bonjour(prenom: string) :string {
   return "Bonjour "+prenom;
}
```

Le contenu de listes ou d'autres variables non scalaires peut être aussi typé :

```
liste :number[] = [1, 2, 3];
ou
liste  :Array<number> = [1, 2, 3];
```

Un type générique existe, c'est **any** : `maVariable :any;`

5.3.2 Gestion des classes :

Les classes sont gérées ainsi bien sûr que leur héritage :

```
class Vehicule {
   private avis: string;

   constructor(avis: string) {
       this.avis = avis;
   }

   donnerAvis(): void {
       console.log("Avis : "+this.avis);
   }
}

class Voiture extends Vehicule {
   constructor(avis: string) {
      super(avis);
   }
}

var fluence: Voiture = new Voiture("Belle voiture");
fluence.donnerAvis();  // Affichage de "Avis : Belle voiture"

console.log("Ai-je le droit d'avoir directement cet avis : "+fluence.avis);
// tsc affiche une erreur, mais le message est quand même affiché...
```

Voici le délicieux code Javascript généré :

```
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
var Vehicule = (function () {
    function Vehicule(avis) {
        this.avis = avis;
    }
    Vehicule.prototype.jeSuis = function () {
        console.log("Avis : " + this.avis);
    };
    return Vehicule;
}());
var Voiture = (function (_super) {
    __extends(Voiture, _super);
    function Voiture(avis) {
        _super.call(this, avis);
    }
    return Voiture;
}(Vehicule));
var fluence = new Voiture("Belle voiture");
fluence.jeSuis();
console.log("Ai-je le droit d'avoir directement cet avis : " + fluence.avis);
```

5.3.3 Gestion d'interfaces :

Les interfaces vont définir le "comportement" des objets instanciés par la classe.

```
interface monInterface {
  // liste des attributs à définir et des méthodes à implémenter
}

class maClasse implements Interface {
  // ...
}
```

5.4 Angular 2 est un framework MVC (voire MVVM) :

Angular 2 comme la première version Angular associé à un serveur Node décline le **modèle MVC** (Modèle Vue Contrôleur) mais le rend plus complexe par son architecture également bâti sur des composants et des modules.

- la **vue** va correspondre aux **templates** (codes HTML dans lequels des données seront visualisées/saisies par l'utilisateur) ;
- le **contrôleur** va être implémenté via un composant qui va organiser les routes (voir les dernières sections de ce polycopié) ;
- le **modèle** qui
  - correspond aux **données** :
    - fournies par le serveur Node aux composants qui en ont besoin (le serveur Node pouvant lui-même se connecter à une base de données NoSQL) ;
    - ou directement gérées par les composants Angular 2 (ce qui semble criticable).
  - les **traitements métiers** :
    - effectués par les composants Angular 2 ;
    - voire en amont au niveau du serveur Node.

L'utilisation du "*two way data binding*", liaison bidirectionnelle de données entre les templates et le modèle (optionnel en Angular 2) décline le modèle MVC en un **modèle Vue-Vue-Modèle** (modèle MVVM),

5.5 Architecture d'une application Angular 2 :

Voici le contenu type du dossier hébergeant une application Angular 2 utilisant TypeScript (d'autres fichiers annexes ne figurent pas dans cet exemple) :

```
dossier de l'application
  |
  |___ node_modules : les paquets npm installés localement
  |
  |___ package.json : le fichier de configuration du projet
  |
  |___ index.html : la page web frontale
  |
  |___ app :le dossier qui va contenir les composants métiers
       |
       |___ app.<nomComposant>.ts : les composants Angular 2 écrits en TypeScript
       |___ app.<nomComposant>.ts
       |___ ...
       |
       |___ app.<nomComposant>.js : les composants Angular 2 transcodés en Javascript
       |___ app.<nomComposant>.js   (création automatique)
       |___ ...
       |
       |___ main.ts : le code TypScript qui va "activer" les composants
  |
  |___ templates :
       |
       |___ <nomComposant>.html : les templates HTML associés aux composants
       |___ <nomComposant>.html
       |___ ...
  |
  |___ styles : dossier contant les fichiers CSS (par défaut un fichier styles.css est créé dans l
```

Maintenant nous allons reprendre notre route en déclinant le quickstart en plusieurs versions.

5.6 La notion de composant :

En effet, l'originalité d'Angular 2 par rapport à l'ancienne version d'Angular est de se baser sur l'utilisation de **composants** a priori indépendants.

Un **composant** est une classe Typescript qui est *décorée* par le **Décorateur @Component** :

```
@Component({
  selector: 'my-component'
})
class MyComponent {
}
```

Le décorateur est une annotation qui permet à TypeScript de générer le code Javascript nécessaire à la mise en oeuvre de ce qui est recherché.

Le **sélecteur** déclare la balise interfacera le composant dans une page web :

```
<body>
   <my-component></my-component>
</body>
```

Le template (qui peut être externalisé) définit le code HTML qui va être injecté entre les balises précédemment définies :

```
@Component({ selector: 'my-component'})
@View({template: "Hello {{qui}} !"})
class MyComponent {
  qui:string = 'toi';
}
```

Ce template est défini dans cet exemple dans un objet géré par le **décorateur @View** Il est aussi possible de référencer ce template sans utiliser ce décorateur (voir exemple ci-après).

5.6.1 Le cycle de vie d'un composant :

Un comosant est créé au moment de sa première invocation.
Il traverse plusieurs états, deux vont nous intéresser.

Lors de son intialisation, des traitements peuvent être spécifiés dans la méthode **ngOnInit** :

```
ngOnInit() { ... }
```

A chaque changement de valeur d'une de ses propriétés, des traitements peuvent être spécifiés dans la méthode **ngDoCheck** :

```
ngDoCheck() { ... }
```

Revenons maintenant à notre "Hello world".

5.7 Le "Hello world" d'Angular 2 :

Notre "Hello Word" est dérivé de celui proposé par le quickstart (car il utilise un template et une feuille de style externalisés).
Le dossier principal va se nommer **Hello** (nom non conventionnel).
Un seul composant sera utilisé : **Hello/app/Hello.ts**

5.7.1 Démarche à suivre :

Cloner le quickstart dans le dossier *Hello* : **cp -R quickstart Hello**
Déplacez-vous dans le dossier **Hello**.

Etapes de la construction du "Hello world" :

- Création du **composant Hello** et mise en place de son activation :
  → déplacez-vous dans le dossier **app** et supprimez tout son contenu :
  rm * (on repart de zéro pour bien tout comprendre)
  → créez les fichiers **Hello.ts** et **main.ts** (voir ci-après).
- création des blocs HTML correspondant à l'interfaçage du composant *Hello* :
  → remontez dans le dossier principal
  → créez les répertoires **templates** et **styles**
  → créez les fichiers **hello.html** dans **templates** et **hello.css** dans **styles** (voir ci-après)
- Interfaçage du composant dans la page web :
  → modifiez le fichier **index.html** (voir ci-après)

Et enfin, remontez dans **Hello** et lancez la commande **npm start**

La commande **npm start** va lancer :
- un serveur **lite-server** dont le rôle est de scruter tout changement dans les fichiers sources càd :
  - les fichiers TypeScript concernant les composants métiers (dans le répertoire **app**) ;
  - les fichiers html et css concernant les templates et les styles associés à ces composants ;

— le fichier d'activation des composants (main.ts) et le frontal (index.html)
— et dans le cas où un fichier TypeScript a été modifié, d'appeler le transcodeur **tsc** pour générer les fichiers Javascript correspondants.

Dans ce premier exemple, le nom du composant (*Hello*) et du fichier qui contient la classe TypeScript qui le définit (*Hello.ts*) ne suivent pas les conventions usuelles :
— le composant devrait se nommer *HelloComponent* ;
— le fichier devrait se nommer *Hello.component.ts*.
C'est juste pour être temporairement rebelle.

5.7.2 Création du composant (Hello/app/Hello.ts) :

Vous remarquerez dans l'exemple ci-dessous que le corps de la classe est vide.
Dans l'exemple suivant (un "hello world" plus avancé), cela ne sera pas le cas.

```
import {Component} from '@angular/core';

@Component({
    selector:    'hello',
    templateUrl: 'templates/hello.html',
    styleUrls:   ['styles/hello.css']
})
export class Hello { }
```

ou avec un décorateur @View (mais qui est déprécié)

```
import {Component} from '@angular/core';

@Component({selector: 'hello'})
@View({
    templateUrl: 'templates/hello.html',
    styleUrls:   ['styles/hello.css']
})
export class Hello { }
```

5.7.3 L'activation du composant (Hello/app/main.ts) :

Tout composant doit être bootstrappé pour être activé.
Si les composants sont regroupés en modules (voir exemple ci-après), ils seront bootstrappés par l'intermédiaire du module.

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {Hello}  from './Hello';

bootstrap(Hello);
```

5.7.4 Le template HTML (Hello/templates/hello.html) :

```
<span> Bonjour toi </span>
```

5.7.5 Le style (Hello/styles/hello.css) :

```
span { font-size: 18; font-style: bold; }
```

5.7.6 Le frontal (Hello/index.html) :

Modifiez le corps (entre les balises *body*) du fichier **index.html** comme suit :

```
<html>
  <head>
     <title> Hello Word </title>
     <meta charset="UTF-8">
     ... les liens nécessaires les fichiers Javascript
  </head>

  <body>
     <hello> Loading... </hello>
  </body>
</html>
```

Et donc, remontez dans **Hello** et lancez la commande **npm start** qui va :
— exécuter la commande associée à l'entrée **start** dans **package.json** :
soit lancer en parallèle le **watcher** et le **transpiler**
— le **watcher** va scruter tous les fichiers qui ont été dernièrement modifiés ;
— et si le fichier modifié est un code typescript (.ts), va appeler le **transpiler** pour générer le code javascript associé (.js)
— et votre navigateur devrait vous dire "bonjour"...

5.8 Le "Hello world" un peu plus avancé :

Nous allons maintenant sophistiquer notre exemple, en lui apportant quelques améliorations :
— nous respectons les conventions de nommage (ah la rebellion n'a qu'un temps) ;
— la salutation sera faite à une personne dont le nom sera saisi dans une zone de saisie.
ce nom sera initialisé à "chef" et il y aura donc une double liaison (un *two way data bindings*) entre la vue (le template) et le modèle (la donnée gérée par le comoosant) ;
— le composant sera herbergé dans un **module** :
si pour l'instant, cela n'apporte rien de plus à notre exemple, cela nous permettra à terme de structurer les composants par rapport à une fonctionnalité globale (par exemple, développement du module qui gère le panier de mon site de e-commerce) ;

5.8.1 Création du composant (Hello/app/Hello.component.ts) :

Contrairement à l'exemple précédent le corps de la classe n'est plus vide : la propriété *toi* est initialisée !

```
import {Component} from '@angular/core';

@Component({
    selector:    'hello',
    templateUrl: 'templates/hello.html',
    styleUrls:   ['styles/hello.css']
})
export class HelloComponent {
    toi :string = 'chef';
}
```

5.8.2 Création du module (Hello/app/app.module.ts) :

Vous remarquerez l'import obligatoire du module **FormsModule** pour mettre en place le *two way data bindings*.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';

import { HelloComponent }  from './Hello.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ HelloComponent ],
  bootstrap:    [ HelloComponent ]
})
export class AppModule { }
```

5.8.3 Le template HTML (Hello/templates/hello.html) :

Le template HTML inclut :

— l'injection de la valeur de la propriété *toi* du composant ;
— une zone de saisie qui permet de modifier la propriété du composant *HelloComponent*.
Cette liaison, appelée **two way data binding** est spécifiée par l'attribut : `[(ngModel)]`.

Ces deux aspects seront développés dans la dernière section de ce polycopié sur les compléments à Angular 2.

```
<span> Bonjour {{toi}} </span> <br/>
<input [(ngModel)]="toi" />
```

5.8.4 Le style (Hello/styles/hello.css) :

```
*  { font-size: 24; font-style: bold; }
```

5.8.5 Le frontal (Hello/index.html) :

Le frontal ne subit pas de modification :

```
<html>
  <head>
     <title> Hello Word </title>
     <meta charset="UTF-8">
     ... les liens nécessaires les fichiers Javascript
  </head>

  <body>
     <hello> Loading... </hello>
  </body>
</html>
```

6 Connexion Angular 2 avec Node :

Nous allons maintenant essayer d'**opérer une requête Ajax (méthode GET) sur le serveur Node** via un composant nommé **Ajax**.
Comme pour nos premiers essais **Hello**, nous allons cloner le quickstart et supprimer le contenu du dossier **app**.
Notre serveur renverra une collection JSON (la liste des produits sur la route **/produits**.

6.1 Le composant qui utilise le module http (AJAX/app/Ajax.component.ts) :

Ce composant doit utiliser une méthode GET d'un service Ajax.
Pour ce faire nous allons utiliser **le module Http**.

Notre composant (**Ajax**) va déclarer/utiliser ce module **dans son constructeur**.
Ce mécanisme est appelé **l'injection de service**.

6.1.1 Première version :

Cette première version utilise directement l'onjet de type **Observable** que retourne la méthode **http.get**.
La méthode **map** permet d'appliquerb un traitement avant de renvoyer l'objet.

```
import {Component} from '@angular/core';
import {HTTP_PROVIDERS, Http, Response} from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
    selector:    'ajax',
    providers:   [HTTP_PROVIDERS],
    templateUrl: 'templates/ajax.html',
    styleUrls:   ['styles/ajax.css']
})
export class AjaxComponent {
    private http: Http;
    public items;

    public constructor(http: Http) {
       this.http = http;
    }

    private appel_serveur() {
       console.log('Importation de données');
       this.http.get('http://localhost:8888/produits')
                     .map((res:Response) => res.json())
                     .subscribe(res => this.items = res,
                                err => console.error(err),
                                () => console.log('done') );
    }

    public action() {
       this.appel_serveur();
    }
}
```

6.1.2 Seconde version :

Cette seconde version (fragments) utilise un objet de type **Promise** (une promesse à venir...).
Cet objet évite l'empilement de callbacks en proposant un chaînage de la méthode *then* (d'une manière générale, pas ici).

```
...
import 'rxjs/add/operator/toPromise';

...
       this.http.get('http://localhost:8888/produits')
                .toPromise()
                .then(response => response.json().data as this.items)
                .catch(this.handleError);
  }

  private handleError(error: any): Promise<any> {
    console.error(error);
  }
...
```

6.2 Activation du composant (Ajax/app/main.ts) :

```
import {bootstrap}     from '@angular/platform-browser-dynamic';
import {AjaxComponent} from './Ajax.component';

bootstrap(AjaxComponent);
```

6.3 Frontal (Ajax/templates/ajax.html) :

Nous allons voir apparaître :
— une syntaxe particulière pour stigmatiser les écouteurs d'événements et les itérations sur les éléments de listes à afficher
→ voir chapitre *Bindings* ;
— que la collection **items** définie dans le composant est "bindée" avec le template (via un mécanisme transparent de **data-binding**)
la propriété **nom** de chaque objet de la collection est **interpolée** dans le HTML.

l

```
<h1> Import de données JSON </h1>

<button (click)="action($event)"> Importation </button>
<ul>
        <li *ngFor="let item of items">{{item.nom}}</li>
</ul>
```

6.4 Ajax/styles/ajax.css :

```
button { font-size: 18pt; color: red; }
li     { font-size: 18pt; color: green; }
```

6.5 Ajax/index.html :

```
<html>
  <head>
     <title> Importation de données JSON via Ajax </title>
     <meta charset="UTF-8">
     ... les liens nécessaires les fichiers Javascript
  </head>
  <body>
     <ajax> Loading... </ajax>
  </body>
</html>
```

7 Bindings (entre la vue et le modèle) :

7.1 Event Binding :

Les écouteurs d'événements sont encadrés par des **parenthèses** ; exemple : `<button (click)="action()"></button>`

Voici l'exemple d'un code qui réfléchit la valeur saisie dans une zone de saisie :

```
import {Component} from '@angular/core';

@Component({
  selector: 'miroir',
  template: `
    <input #saisie (keyup)="onKey(saisie.value)">
    <p>Valeur : {{value}}</p>
  `
})
export class Miroir {
  value:string = '';
  onKey(value: string) {
    this.value = value;
  }
}
```

7.2 Property binding :

Comme il a été vu dans la paragraphe précédent, l'**interpolation** d'une propriété d'une classe/composant TypeScript dans un template HTML est effectuée grâce aux **doubles accolades** ; exemple : `<p> Bonjour {{nom}} </p>`

Si la propriété à interpoler est la valeur d'un attribut d'une balise HTML, l'attribut est encadré par des **crochets** pour indiquer qu'il est pris en charge par Angular.

Voici un exemple de modification dynamique d'une image.

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'gestionImage',
  template: `
    <input #saisie (keyup)="changeImage(saisie.value)">
    <img [src]="URLImage" />
  `
})
export class GestionImage {
  @Input() src:any;
  URLImage: string = "modif.png";  // placé ici au-dessus de app

  changeImage(URL: string) {
       this.URLImage = URL;
  }
}
```

Dans ce code, nous voyons apparaître :
— la **variable locale** qui permet de référencer le noeud de l'arbre DOM correspondant à la balise ;
— le **décorateur @Input** qui permet de déclarer les attributs de balises HTML pris en charge par Angular.

7.3 Two way Data Binding :

Le **two way data binding** n'est pas actif par défaut sous Angular 2 contrairement à la première version d'AngularJS (via le feu-scope).
En effet, ce mécanisme de liaison de données automatique entre la vue et le modèle (par exemple le changement de la valeur d'une zone de saisie dans un template permettant automatiquement de mettre à jour une variable du module (AngularJS) / du composant (Angular 2) et réciproquement), est très coûteux en terme de ressources.

Il peut être implémenter grâce à une syntaxe spécifique : **[(ngModel)]**.

```
import {Component} from '@angular/core';

@Component({
selector: 'dataBinding',
template: `
    <input [(ngModel)]="valeur"/>
    {{valeur}}
    `
})
export class DataBinding {}
```

8 Les décorateurs usuels :

Un décorateur est le nom d'une structure de patron de conception.

8.1 Le décorateur @Component :

Transforme une classe Typescript en un composant :

```
import {Component} from '@angular/core';
@Component({
    selector: 'hello',
    template: 'Bonjour toi !'
})
export class Hello {}
```

8.2 Le décorateur @Injectable :

Transforme un composant/classe Typescript en un composant injectable dans le constructeur d'une autre composant via une injection de dépendance. Un exemple de ce fonctionnement est décrit dans la section consacrée à l'injection de dépendance.

8.3 Le décorateur @Input :

Déclare un attribut d'une balise HTML dont la valeur est prise en charge par Angular 2 :

```
@Component({
  ...
  template: '<img [src]="URLImage" />'
})
export class GestionImage {
  @Input() src:any;
  ...
}
```

8.4 Le décorateur @NgModule :

Déclare une classe comme étant un module regroupant des composants.
Un module peut être vu comme un **package**.

9 L'injection de dépendances :

L'injection de dépendances est un point fort d'Angular 2.
Il permet de donner à un composant les dépendances/services dont il a besoin lors de son instanciation.

Dans l'exemple précédent sur le *composant Ajax (Ajax.component.ts)* qui sollicitait Node pour récupérer des données, le module HTTP est directement injecté dans son constructeur.
Ainsi, sous Angular 2, il est donc possible de donner à n'importe quel composant les services dont il a besoin en les déclarant dans son constructeur :

```
...
export class ComposantComponent {
    ...
    public constructor(private service :Service, ...) {}
    ...
}
```

9.1 Exemple d'injection de service :

Imaginons que j'ai besoin du module HTTP dans plusieurs composants de mon application qui vont solliciter le même serveur Node (cela sera le cas dans le dernier exemple de routage présenté dans ce polycopié),

Par exemple voici notre service qui encapusle le module HTTP :

```
@Injectable()
export class RechercheService {
    constructor(private http :Http) {}

    getJSON(parametres :string) :Observable<any> {
      let observable :Observable<any> =  this.http.get(url)
                                            .map((res:Response) => res.json());
      return observable;
    }
}
```

Vous remarquerez que j'ai créé une **classe injectable** (injectable class) **RechercheService** "décorée" par le décorateur **@Injectable** ce qui nous permet de l'utiliser comme **service**.

Le composant ainsi créé **être injecté dans d'autres composants** qui en auront besoin :

```
  ...
export class ComposantComponent {
    public items :any;

    public constructor(private recherche :RechercheService, private route: ActivatedRoute) {}

    ngOnInit() {
       this.route.params.subscribe(params => {
                  this.recherche.getJSON(params[...])
                       .subscribe(res => this.items = res,
                                  err => console.error(err),
                                  () => console.log('done'));

                  });
   }
}
```

10 Gestion des routes (le "routing") :

Pour gérer les routes, il faut implémenter un composant spécifique.
Même si la notion de contrôleur perd un peu de son intensité par rapport à la première version d'Angular (AngularJS), le composant qui va implémenter les routes peut être considéré comme le contrôleur de l'application.

Le **routage/routing** via des routes REST va être codé en deux étapes :
— du template vers le contrôleur ;
— du contrôleur vers le composant cible.

10.1 Du template vers le contrôleur et du contrôleur vers le composant cible :

10.1.1 Du template vers le contrôleur :

routerLink est l'argument à spécifier dans une ancre HTML pour invoquer le contrôleur via une route REST.

```
  <a [routerLink]="['/routeVersLeContrôleur']"> ... </a>
ou plus simplement
  <a routerLink="/routeVersLeContrôleur"> ... </a>
```

10.1.2 Du template vers le contrôleur avec une route avec un paramètre :

```
<a [routerLink]="['/routeVersLeContrôleur', paramètre]"> ... </a>
```

10.2 Du contrôleur vers le composant cible :

10.2.1 Du contrôleur vers le composant cible sans paramètre :

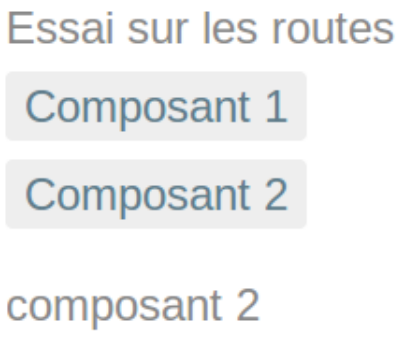
```
const appRoutes: Routes = [
  { path: 'routeVersLeContrôleur', component: Composant },
];
```

10.2.2 Du contrôleur vers le composant cible avec un paramètre :

```
const appRoutes: Routes = [,
  { path: 'routeVersLeContrôleur/:paramètre', component: Composant }
];
```

10.3 Un premier exemple de menu pour invoquer un composant parmi plusieurs :

Ce premier exemple montre comment afficher un menu permettant d'invoquer un composant.



L'internaute vient de cliquer sur l'item "Composant2", le composant2 est invoqué est affiche "composant2".

Dans l'exemple suivant, nous aurons les codes suivants :
— les composants :

- les deux composants routables :
  - Un composant qui affiche "composant1" (*Routes/app/Composant1.component.ts*)
  - Un composant qui affiche "composant2" (*Routes/app/Composant2.component.ts*)
- Le composant qui permet d'appeler l'un des deux composants précédents (*Routes/app/Menu.component.ts*)
- Le composant implémentant les routes (*Routes/app/app.routing.ts*)

— le code regroupant les composants dans un module (*Routes/app/app.module.ts*)
— le code activant le module (*Routes/app/main.ts*)
— la page web frontale (*Routes/index.html*) :)

10.3.1 Premier composant routable (ExempleRoutes/app/Composant1.component.ts) :

```
import {Component} from '@angular/core';

@Component({
    selector:    'composant1',
    templateUrl: 'templates/composant1.html',
    styleUrls:   ['styles/composant1.css']
})
export class Composant1Component {}
```

Ce code est lié au template (*Routes/templates/composant1.html*) :

```
<span> composant 1 </span>
```

et au style (*Routes/styles/composant1.css*) :

```
*  { font-size: 24; font-style: bold; }
```

10.3.2 Second composant routable (ExempleRoutes/app/Composant2.component.ts) :

Ce second composant est similaire au précédent mais affiche "composant 2".

10.3.3 Le menu (Routes/app/Menu.component.ts) :

Le menu va créer deux ancres permettant de sélectionner le composant voulu.

```
import {Component} from '@angular/core';

@Component({
    selector:    'menu',
    templateUrl: 'templates/menu.html',
    styleUrls:   ['styles/menu.css']
})
export class MenuComponent {
    titre = 'Essai sur les routes';
}
```

Ce code est lié au template (*Routes/templates/menu.html*) :

```
<span> {{titre}} </span> <br/>
<nav>
       <a routerLink="composant1"> Composant 1 </a> <br/>
       <a routerLink="composant2"> Composant 2 </a>
</nav>
<br/>
<router-outlet></router-outlet>
```

Il faut remarquer deux choses importantes :
— **routerLink** qui définit la route (coir paragraphe ci-après)
— **¡router-outlet¿** qui définit la zone où le composant routé sera interfacé.

et au style (*Routes/styles/menu.css*) :

```
*  { font-size: 24; font-style: bold; }
```

10.3.4 Le composant implémentant les routes (Routes/app/app.routing.ts) :

```
import { ModuleWithProviders }  from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { Composant1Component }  from './Composant1.component';
import { Composant2Component }  from './Composant2.component';

const appRoutes: Routes = [
  { path: 'composant1', component: Composant1Component },
  { path: 'composant2', component: Composant2Component },
];

export const appRoutingProviders: any[] = [ ];

export const routing: ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

10.3.5 Le module regroupant les composants précédents (Routes/app/app.module.ts) :

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';

import { routing, appRoutingProviders } from './app.routing';

import { MenuComponent }        from './Menu.component';
import { Composant1Component }  from './Composant1.component';
import { Composant2Component }  from './Composant2.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule, routing ],
  declarations: [ MenuComponent, Composant1Component, Composant2Component ],
  providers:    [ appRoutingProviders ],
  bootstrap:    [ MenuComponent ]
})
export class AppModule { }
```

10.3.6 Le code activant le module (Routes/app/main.ts) :

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

// Compile and launch the module
platformBrowserDynamic().bootstrapModule(AppModule);
```

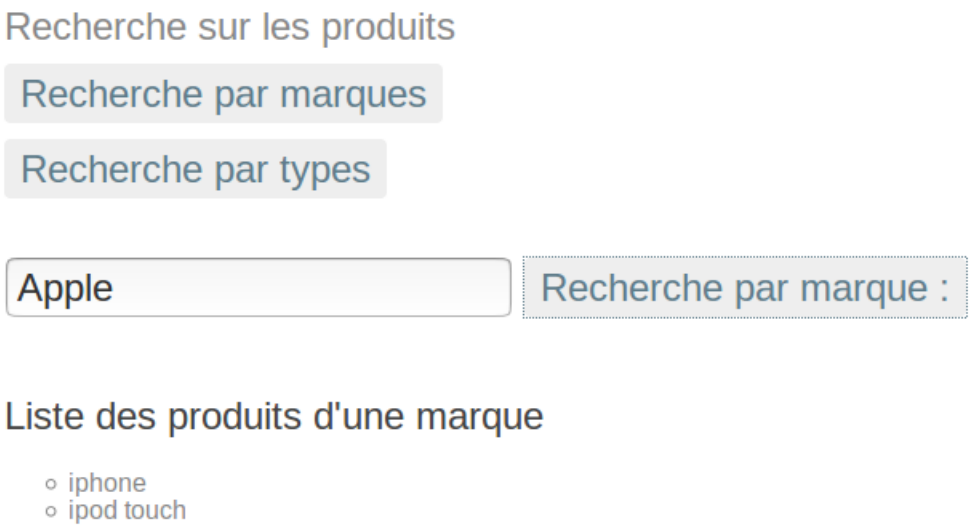
10.3.7 La page web frontale (Routes/index.html) :

```
<html>
  <head>
     <base href="/">

     <title> Importation de données JSON via Ajax </title>
     <meta charset="UTF-8">
     ... les liens nécessaires les fichiers Javascript
  </head>

  <body>
     <menu> Loading... </menu>
  </body>
</html>
```

10.4 Un exemple plus complet (avec injection et paramétrage d'une route) :

Ce second exemple montre :

— comment afficher un menu permettant d'invoquer un composant **en lui passant un paramètre**.
— comment utiliser un service via une **injection de dépendance**.



L'internaute vient de saisir "Apple" dans la zone de saisie et cliquer sur le bouton "Recherche par marque".

Dans l'exemple suivant, nous aurons les codes suivants :
— Les composants :
  — Les composants routables (seul le troisième appelé avec un paramètre sera exhibé) :
    — Un composant qui affiche les types de produits (*ECOMMERCE/app/RechercheTypes.component.ts*)
    — Un composant qui affiche les marques des produits (*ECOMMERCE/app/RechercheMarques.component.ts*)
    — **Un composant qui affiche les produits d'une marque donnée** (*ECOMMERCE/app/RechercheParMarq*
  — **Le composant qui implémente le service HTTP utilisé par les trois composants précédents** (*ECOMMERCE/app/Recherche.service.ts*)
  — Le composant qui permet d'appeler l'un des composants (*ECOMMERCE/app/Menu.component.ts*)
  — Le composant implémentant les routes (*ECOMMERCE/app/app.routing.ts*)
— Le code regroupant les composants dans un module (*ECOMMERCE/app/app.module.ts*)
— Le code activant le module (*ECOMMERCE/app/main.ts*) → même structure que dans le premier exemple
— La page web frontale (*ECOMMERCE/index.html*) :) → même structure que dans le premier exemple

10.4.1 Le composant RechercheParMarque (ECOMMERCE/app/RechercheParMarque.component.ts) :

```
import {Component, OnInit} from '@angular/core';
import {ActivatedRoute} from '@angular/router';
import { RechercheService } from './Recherche.service';

@Component({
    templateUrl: 'templates/rechercheParMarque.html',
})
export class RechercheParMarqueComponent {
    public items :any;

    public constructor(private recherche :RechercheService, private route: ActivatedRoute) {}

    ngOnInit() {
       this.route.params.subscribe(params => {
                  this.recherche.getJSON("marque/"+params['marque'])
                       .subscribe(res => this.items = res,
                                  err => console.error(err),
                                  () => console.log('done'));

                  });
   }
}
```

10.4.2 Le service Recherche (ECOMMERCE/app/Recherche.service.ts) :

```
import {Injectable} from '@angular/core';
import {Http, Response} from '@angular/http';
import {Observable} from 'rxjs/Rx';
import 'rxjs/add/operator/map';

@Injectable()
export class RechercheService {
    constructor(private http :Http) {}

    getJSON(parametres :string) :Observable<any> {
           let url :string = "http://localhost:8888/produits/"+parametres;
           let observable :Observable<any> =  this.http.get(url)
                                                   .map((res:Response) => res.json());

           return observable;
    }
}
```

10.4.3 Le menu (ECOMMERCE/app/Menu.component.ts) :

```
import {Component} from '@angular/core';

@Component({
    selector:    'menu',
    templateUrl: 'templates/menu.html',
    styleUrls:   ['styles/menu.css']
})
export class MenuComponent {
    titre = 'Recherche sur les produits';
    marque :string = 'Apple';

    setMarque( value :string) {
      this.marque = value;
   }
}
```

Ce code est lié au template (*ECOMMERCE/templates/menu.html*) :

```
<span> {{titre}} </span> <br/>
<nav>
       <a [routerLink]="['/rechercheMarques']"> Recherche par marques </a> <br/>
       <a [routerLink]="['/rechercheTypes']"> Recherche par types </a>
       <br/><br/>
       <input (keyup)="setMarque(choixMarque.value)" value="Apple" #choixMarque />
       <a [routerLink]="['/rechercheParMarque', marque]"> Recherche par marque : </a>
</nav>
<br/>
<router-outlet></router-outlet>
```

10.4.4 Le composant implémentant les routes (ECOMMERCE/app/app.routing.ts) :

```
import { ModuleWithProviders }  from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { RechercheMarquesComponent }   from './RechercheMarques.component';
import { RechercheTypesComponent }     from './RechercheTypes.component';
import { RechercheParMarqueComponent } from './RechercheParMarque.component';

const appRoutes: Routes = [
  { path: 'rechercheMarques', component: RechercheMarquesComponent },
  { path: 'rechercheTypes', component: RechercheTypesComponent },
  { path: 'rechercheParMarque/:marque', component: RechercheParMarqueComponent }
];

export const appRoutingProviders: any[] = [ ];

export const routing: ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

10.4.5 Le module regroupant les composants précédents (ECOMMERCE/app/app.module.ts) :

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule }    from '@angular/http';

import { routing, appRoutingProviders } from './app.routing';

import { RechercheService } from './Recherche.service';

import { MenuComponent }                from './Menu.component';
import { RechercheMarquesComponent }    from './RechercheMarques.component';
import { RechercheTypesComponent }      from './RechercheTypes.component';
import { RechercheParMarqueComponent } from './RechercheParMarque.component';

@NgModule({
  imports:      [ BrowserModule, HttpModule, routing ],
  declarations: [ MenuComponent, RechercheMarquesComponent, RechercheTypesComponent, RechercheParM
  providers:    [ RechercheService ],
  bootstrap:    [ MenuComponent ]
})
export class AppModule { }
```

Index