Tutoriel Angular ®

Hervé Le Cornec, herve.le.cornec@free.fr

Avant propos

Je développe des applications web depuis plus de 25 ans, et depuis tout ce temps la technologie javascript n'a fait que progresser. La première révolution fut Ajax, qui permettait enfin au javascript d'envoyer et decevoir des données depuis un serveur au travers du réseau. Puis vint la seconde révolution, JQuery, qui facilitait grandement les interactions entre le javascript d'un composant et son template HTML. Vint ensuite la troisième révolution, celle du moteur javascript de du navigateur Chrome, qui était d'une puissance jamais égalée auparavant. Aujourd'hui appelé V8 dans sa dernière version, il équipe désormais tous les navigateurs du marché, tant il est puissant. Vint enfin la quatrième révolution : Angular.

Angular est un framework javascript dont la technologie est en totale rupture avec les frameworks classiques (Jquery, React, Vue, ...). Il résout d'une manière simple et élégante les deux problèmes principaux de tout développement d'applications web : la communication entre composants et la gestion des asynchronismes. Il est conçu pour alléger le plus possible le travail d'infrastructure du code javascript, pour permettre au codeur de passer le maximum de son temps sur ce qui est directement utile à l'utilisateur, c'est à dire l'interface. Par exemple l'utilisateur n'aura cure d'un développement Redux faisant communiquer deux composants entre eux, tout ce dont il a besoin c'est d'afficher les variables sur l'interface, et la mécanique sous-jascente ne lui importe pas. Un peu comme un automobiliste ne souhaite pas savoir comment fonctionne le moteur de sa voiture, tout ce qu'il veut c'est la conduire.

Pour parvenir à cela Angular a inventé la technologie de la boucle infinie qui scanne toute l'application par intervalles de 10 millisecondes, et met à jour ce qui a changé depuis la dernière boucle. C'est une technologie puissante, bien que de mise en œuvre très aisée car standard par défaut, et qui simplifie considérablement le développement, améliorant ainsi la performance, les délais, les coûts, la maintenabilité et l'évolutivité. Aucun autre framework n'utilise cette technologie.

Ce cours est destiné à être une initiation et un mementum des choses les plus essentielles dont vous aurez besoin si vous souhaitez développer en Angular. Donner une documentation exhaustive, en français, aurait été difficile à rédiger tant cette documentation est volumineuse. Nous avons donc choisi de présenter les éléments les plus indispensables qui vous permettront de développer à peu près toutes les applications que vous souhaiterez. Lorsque vous maîtriserez les éléments de ce cours vous pourrez vous reporter à la documentation officielle pour entrer dans plus de détails.

Ce cours fut historiquement écrit pour la version 5, nous l'avons revu pour la version 14, et nous prions le lecteur d'être indulgent s'il trouve encore des traces de version 5 ça et là.

Table of Contents

1) Pourquoi Angular

1.1) Framework de développement

Angular est un framework de développement d'applications web, c'est à dire d'applications destinées à être jouée dans un onglet de navigateur internet. Au final, lorsque vous aurez développé votre application, que vous l'aurez « buildée », le résultat sera un ensemble de fichiers javascript, HTML et CSS, qu'il vous suffira de déposer dans un répertoire d'un serveur web autorisé à la consultation pour vos utilisateurs.

Vous pourriez directement coder manuellement en javascript/HTML/CSS cet ensemble de fichiers, sans utiliser Angular, et votre application serait identique. En revanche Angular met à votre disposition tout un environnement de développement qui vous facilite grandement le travail de codage, et pour des applications complexes et sécurisées, c'est une aide devenue aujourd'hui indispensable.

Angular met par exemple à votre disposition un serveur web, de technologie Node.js, dédié uniquement à votre développement, et qui affichera votre application en cours de développement de façon intelligente. Par exemple si vous apportez une modification à un fichier, Angular fera automatiquement un nouveau build puis rechargera la page avec la nouvelle version de votre application. Angular vous garantit aussi par défaut la sécurité maximale de votre application, en ne permettant pas les injections de code malveillant. Son organisation par modules et composants, avec des services très élaborés, vous maintient dans une obligation de produire un code très structuré, donc facilement maintenable et très évolutif.

Et puis Angular utilise préférentiellement le Typescript comme langage de développement, même si après un « build » l'application est écrite en javascript. Le Typescript permet aux développeurs de culture backend, utilisant des langages typés (C#, Java, Python, ...), de ne pas être directement confronté au javascript qui est un langage non typé, et donc libertaire en ce sens que la plus grande liberté est laissée au programmeur, rendant difficile la coopération entre bibliothèques.

Mais en cela rien de très différenciant avec les autres frameworks tels que React ou Vue, alors pourquoi choisir Angular ? La réponse à cette question mérite un paragraphe entier, le suivant.

1.2) Une technologie unique

1.2.1) Connexions entre les composants

Ce qui différencie fondamentalement Angular des autres framework, c'est sa technologie. Sans que vous n'ayez besoin de le coder, les applications Angular sont basées par défaut sur une boucle infinie qui se répète toutes les 10ms (si le navigateur répond correctement). À chaque boucle Angular scanne toute l'application et remet à jour ce qui doit l'être, car modifié depuis la dernière boucle. Et cela fait toute la différence.

En effet en javascript il n'y a pas de notion de variable globale. Chaque composant d'une page web (un menu, un input, ...) est ignorant des autres composants. Un composant se résume à un code HTML associé à un code javascript dédié. Rien n'est prévu pour faire communiquer les composants entre eux.

Prenons un exemple. Notre application possède un bandeau de tête (composant) affichant le nom de l'utilisateur, et une page profil (un autre composant) permettant à l'utilisateur de modifier son nom. À l'évidence l'utilisateur s'attend à ce qu'en changeant son nom dans la page profil, il voit aussi son nom changer sur le bandeau. L'utilisateur voudrait bien aussi que cette opération réclame un minimum de clics, voir aucun si possible.

Sans utiliser Angular, je devrais coder dans le second composant (la page profil) un callback pour chaque modification de l'input « nom », ce callback appelant le script du premier composant (le bandeau) pour lui demander de modifier le nom affiché. Et si le nom doit apparaître à N positions dans l'application, vous devrez coder N callbacks. Imaginez alors une application plus complexe, avec de nombreux composants interdépendants, et le codage monstrueux des callbacks entrecroisés. Plus l'application grossira et plus elle sera complexe, difficile à maintenir et à faire évoluer. C'est ce que nous avons vécu avec JQuery, avant l'arrivée des frameworks.

Les frameworks les plus en vogue sont React et Vue. Ils apportent un plus par rapport à JQuery en ce sens qu'ils simplifient le code, souvent de façon très élégante. Cependant ils fonctionnent toujours sur le même principe d'indépendance des composants, à relier entre eux par un store de type Redux. C'est ce Redux qui gérera vos callbacks, toujours aussi nombreux et enchevêtrés, mais mieux organisés et centralisés. Il vous faudra cependant toujours coder ces callbacks, en Redux certes, mais vous devrez les coder.

Le framework Angular quant à lui fonctionne différemment, il est bâti sur une structure qui par défaut met à disposition des variables globales partagées. Pour Angular une application est un objet unique dont les méthodes et propriétés sont les différents composants. Ainsi tous les composants sont nativement interconnectés. Si deux composants importent le même service, la modification d'une variable de ce service se répercutera immédiatement (en 10ms) sur l'autre composant, sans avoir besoin d'écrire une seule ligne de code. Si le composant bandeau importe le même service que le composant profil, appelons le store, la modification de store.nom se répercutera immédiatement (10ms) dans le bandeau, sans aucun callback à programmer. En fait on pourrait dire que Angular est doté d'une sorte de store Redux, mais automatique et couvrant toute l'application par défaut. Imaginez alors le gain de temps et de complexité entre la programmation du Redux avec ses callbacks entrecroisés, et ... rien à programmer. Le code final est terriblement allégé, donc l'application est plus performante, les délais de développement s'écroulent drastiquement, la maintenance est beaucoup plus aisée, l'évolutivité est garantie, donc tout cela coûte moins cher. Il est là l'intérêt d'Angular, dans son two-way data binding, non plus réservé à l'intérieur d'un composant unique, mais fonctionnant aussi entre composants.

Le lecteur qui m'a suivi jusqu'ici comprendra alors qu'injecter un Redux like, NgRx pour ne pas le nommer, dans Angular est la preuve qu'on ignore le fonctionnement technique d'Angular. C'est dégrader lourdement ses capacités technologiques, jusqu'à même les faire totalement disparaître. C'est la chose à ne surtout pas faire.

1.2.2) Gestion de l'asynchronisme

Tout ce qui se passe sur une page web est asynchrone. Les requêtes au serveur, les affichages d'image et de composants, les clics de l'utilisateur, etc. Disons tout de suite que cet asynchronisme est un mal nécessaire qu'il faut dompter plutôt que de l'obliger à devenir séquentiel. En effet il suffirait alors que le réseau soit lent pour bloquer l'affichage de la page en attendant qu'une image soit téléchargée. Un conseil absolu est donc de ne jamais utiliser les « await » qui transforment l'asynchrone javascript en piteux langage séquentiel.

Le javascript est donc équipé en standard de gestion des asynchronismes. Il définit les « promise » et les « observables » qui sont des promises plus évoluées. Ces « promise » sont des callbacks chargés d'attendre que la donnée asynchrone lui parvienne, puis d'exécuter une action au retour de ces dernières, peu importe le temps qu'il aura fallu pour que les données parviennent enfin.

.

Avec React les « observables » sont stockés dans le store Redux interne, puis propagés jusque dans les composants, où ils sont finalement résolus. On passe donc un temps non négligeable à jongler avec les observables qui s'entre croisent.

Avec Angular ce problème n'existe pas. Lorsqu'une service api est appelé, son callback ne fait qu'une seule

chose : résoudre immédiatement l'observable en données disponibles dans un service « **`store`** ». Ensuite tous les composants qui importent ce service **`store`** disposeront immédiatement des données. On peut donc directement demander dans les templates HTML d'Angular d'afficher **`store.data`**, mais comme **`data`** est asynchrone il faudra écrire **`store?.data`**, et Angular attendra alors sagement que data soit instanciée avant de l'afficher.

C'est la façon qu'à Angular de gérer les asynchronismes, en les résolvant le plus vite possible pour ne pas avoir à les traîner jusque dans les composants et leurs templates HTML. Cette simplicité permet une fois encore un codage minimal, donc un délai court, une maintenabilité facilité et une évolutivité garantie.

2) Mise en place

2.1) Installation d'Angular

2.1.1) Framework de développement

Angular est un framework de développement d'applications web, c'est à dire d'applications destinées à être jouée dans un onglet de navigateur internet. Au final, lorsque vous aurez développé votre application, que vous l'aurez « buildée », le résultat sera un ensemble de fichiers javascript, HTML et CSS, qu'il vous suffira de déposer dans un répertoire d'un serveur web autorisé à la consultation pour vos utilisateurs. Vous pourriez directement coder manuellement en javascript/HTML/CSS cet ensemble de fichiers, sans utiliser Angular, et votre application serait identique. En revanche Angular met à votre disposition tout un environnement de développement qui vous facilite grandement le travail de codage, et pour des applications complexes et sécurisées, c'est une aide devenue aujourd'hui indispensable.

Angular met par exemple à votre disposition un serveur web, de technologie Node.js, dédié uniquement à votre développement, et qui affichera votre application en cours de développement de façon intelligente. Par exemple si vous apportez une modification à un fichier, Angular fera automatiquement un nouveau build puis rechargera la page avec la nouvelle version de votre application. Angular vous garantit aussi par défaut la sécurité maximale de votre application, en ne permettant pas les injections de code malveillant. Et bien d'autres choses bien sûr qui seront évoquées tout au long de ce cours.

Mais avant cela voyons les prérequis à l'utilisation d'Angular.

2.1.2) Prérequis

Avant d'installer Angular il faut installer Node.js sur la machine de développement. Pour cela il faudra que vous soyez root (unix) ou administrateur (Windows) de votre machine. Pour le reste il vous suffit de suivre les instructions d'installation sur le site web de Node.js.

Attention cependant, Node et Angular changent de versions de façon indépendante et certaines versions de Node peuvent ne pas convenir à une version particulière d'Angular. Vous devrez vérifier que les versions de Node et Angular que vous aurez choisies cohabitent correctement. Le plus souvent vous vous apercevrez du problème lorsque le build d'Angular ne se fera pas, avec une litanie de messages d'erreur incompréhensibles.

Une fois installé Node vous procurera l'accès à deux commandes très utiles : **`npm`** et **`npx`**.

2.1.3) Installation globale

Si vous êtes root ou administrateur de votre machine, vous pouvez demander à Node d'installer Angular de façon globale :

```
npm install -g @angular/cli
```

Ainsi lorsque vous voudrez créer une application Angular, où que vous soyez sur votre file system, il vous suffira de taper :

```
ng new my-app
```

ng est la contraction de A**ng**ulard. Ensuite ouvrez le répertoire **my-app** avec votre éditeur de code préféré, puis lancez :

```
cd my-app
ng serve
```

Et c'est parti, votre appli **my-app** est disponible sur le port **http://localhost:4200**.

*Le problème avec l'installation globale est de maintenir des applications qui dépendent de versions différentes d'Angular. L'utilisation de la commande globale **ng**, de version 14 par exemple, sera incompatible pour une application historique développée à l'époque en Angular 7, mais qui donne entière satisfaction aux utilisateurs en production. Le mieux est alors d'upgrader votre application de 7 à 14, c'est la bonne pratique, mais il arrive cependant parfois que cela ne soit pas possible, car non prioritaire dans une entreprise par exemple. Il faut dans ce cas savoir gérer la situation avec des versions d'Angular locales et non plus globales.*

2.1.4) Installation non globale

Si vous n'avez pas les droits administrateur sur votre machine, et donc que vous ne pouvez pas installer Angular globalement, vous devrez pratiquer autrement. C'est Node qui se chargera de vous donner les ressources nécessaires à faire fonctionner le framework.
Pour installer une application Angular localement il faut taper :

```
npx -p @angular/cli ng new my-app
```

On comprend ici que c'est **npx**, une commande fournie par Node, qui lance angular, **ng**, et plus vous directement. En vous plaçant dans le répertoire myapp, vous ne pourrez plus non plus lancer la commande ng serve, car le système ne trouvera aucune installation globale de ng. Il faudra alors lancer :

```
cd my-app
npm run ng serve
```

Ici **ng** est lancé par **npm**.

*Cette façon de faire est très utile car si vous avez un Angular 14 installé en global, et que my-app est une application en Angular 7, la commande ci-dessus utilisant **npm** indiquera à Angular qu'il doit se référencer à l'Angular 7 contenu dans le répertoire **node_modules** de **my-app**, et non pas chercher*

un Angular global.

De façon générale les commandes sont toujours lancées par l'intermédiaire de **`npm`** si on travaille de façon locale dans une application. Ainsi pour builder on fera :

```
npm run ng build
```

Dans ces deux dernière commandes, **`serve`** et **`build`** sont des scripts décrits dans le fichier **`package.json`** contenu dans le répertoire de l'application, et généré automatiquement à la création de celle-ci.

2.1.5) Structure d'une initiale d'une application

L'arborescence des répertoires et fichiers créés par Angular est la suivante :

```
// Répertoire des builds de production ou de development de votre applicaiton .
├── dist

// Répertoire de stockage des bibliothèques nécessaires au fonctionnement
// de l'environnement Angular et des bibliothèques applicatives, telles
// que défini dans le fichier package.json
├── node_modules

// Répertoire des tests end to end.
├── e2e

// Répertoire du code de l'application
├── src
│   ├── app
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.component.ts
│   │   └── app.module.ts

// Répertoire de configuration des environnements (dev, prod, ...)
│   ├── environments
│   │   ├── environment.prod.ts
│   │   └── environment.ts

// Fichiers fondamentaux html et typescript
│   ├── index.html
│   ├── main.ts
│   ├── favicon.ico
│   ├── polyfills.ts
│   ├── styles.css

// Prépare et lance les tests unitaires
│   ├── test.ts

// Fichiers de configuration typescript
│   ├── tsconfig.app.json
│   ├── tsconfig.spec.json

// Fichiers de configuration des types typescript
│   └── typings.d.ts

// Répertoire des tests fonctionnels
├── e2e
```

```
// Fichiers de configuration des éléments utilisés par Angular
├── angular.json
├── package.json
├── README.md
└── tslint.json
```

2.1.6) Fichiers remarquables

.angular.json

C'est le fichier de configuration d'Angular. Il définit tous les éléments nécessaires au fonctionnement d'Angular. On y indique quelles bibliothèques seront utilisées, on y décrit les limites des builds, on spécifie les répertoires de travail, les types de « compilateurs », etc.

package.json

C'est le fichier emblématique de npm (Node Package Manager) . Il définit toutes les bibliothèques, et leurs versions, nécessaires à construire l'application. Lorsque vous avez cloné une application à partir d'un repository, il faut ensuite installer les dépendances avant de pouvoir la faire fonctionner. Pour cela on tape :

```
npm i
```

Cette commande indique à **npm** d'installer les bibliothèques décrites dans le fichier **package.json**, ainsi que leurs dépendances.

*Si vous souhaitez intégrer une nouvelle bibliothèque ou une nouvelle version de bibliothèque, modifiez votre fichier **package.json**, puis lancez la commande **npm i**, seules les modifications demandées seront prises en compte, le reste étant déja up to date.*

Ce fichier définit aussi les raccourcis des exécutables qu'on voudra utiliser pour gérer cette application, comme **build**, **start**, ou toute autre commande qu'on voudra y inscrire, par exemple :

```
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "start443": "ng serve --port 443",
    "build": "ng build",
    "buildParis": "ng build –configuration angularParis.json",
    "buildLondres": "ng build –configuration angularLondre.json",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
```

Ainsi lancer

```
npm run start
```

ou

```
npm run ng serve
```

revient au même.

tsconfig.json

Typescript, produit par Microsoft mais cédé au consortium Apache, est une surcouche de javascript permettant d'écrire du javascript en respectant les règles de la programmation orientée objet, familières aux développeurs de Java, ou C++, mais étrangères au monde du javascript. Sa « compilation » (à vrai dire un terme impropre) est destinée à produire du javascript pur. ts.config.json est le fichier de configuration du compilateur typescript. Voir https://www.typescriptlang.org/

tslint.json

Lint est un analyseur de qualité de code, il détecte les fautes d'écriture qui auront un impact sur la minification d'un code javascript (oubli d'un point-virgule, oubli d'une accolade, …). Le fichier tslint.json sert à paramétrer Lint qui sera lancé à la compilation (build), en le rendant plus ou moins permissif.
Voir https://www.npmjs.com/package/lint

src/index.html

Angular est un framework pour Single Page Application, c'est à dire que toute l'application ne tient que dans une seule page, celle-ci. Elle intégrera dynamiquement toute l'application, à l'intérieur de son tag <app-root></app-root>. A priori ce fichier n'a aucun besoin d'être modifié.

src/styles.css

Ce fichier contient les styles qui seront accessibles dans toute l'application. Chaque composant d'Angular possède néanmoins son propre fichier de style, ou sa propre définition de style, qui s'ajoutera à ce style global, ou le modifiera.

src/main.ts

Permet de compiler l'Application Angular avec le compilateur JIT.

src/tsconsfig.app.ts

Configure la façon dont Typescript est compilé en Javascript. Documentation ici :
http://www.typescriptlang.org/docs/handbook/tsconfig-json.html

src/tsconfig.spec.ts

Permet la compilation de Typescript pour les tests unitaires.

environments

Dans ce répertoire se trouvent les fichiers d'environnement utiles à Angular pour produire les builds de développement ou de production. Vous pouvez produire autant de fichiers d'environnement que vous souhaitez, chacun avec la syntaxe

```
environnement.XXXX.ts
```

et il faudra alors lancer les build en conséquence :

```
ng build –environnement XXXX
```

Par défaut la commande ***`ng build`*** *utilise le fichier* ***`environnement.prod.ts`***.

e2e

Répertoire pour les fichiers de configuration des tests fonctionnels.

2.1.7) Lancer l'application

L'application est directement utilisable, sans même avoir mis en lace un serveur web, car Angular vient avec un serveur Node intégré. Pour le lancer il faut ce placer dans le répertoire créé par Angular, puis lancer le serveur :

```
# cd my-app
# ng serve
```

Il suffit alors d'afficher l'application dans votre navigateur préféré, en se connectant à l'URL par défaut :

```
localhost:4200
```

Le port du server Node Express peut être modifié de deux façons. Soit au lancement en ajoutant l'option `--port` :

```
# ng serve --port 8080
```

soit en configurant le port dans le fichier `.angular-cli.json`, dans la section « `defaults` » :

```
"defaults": {
  "serve" : {
     "port" : 8080
  }
}
```

Dans ces exemple on a choisi le port 8080 mais on peut préférer tout autre port.

- *Il est conseillé d 'utiliser le navigateur Chrome pour le développement car il est doté d'un débugger très efficace et très complet (plus d'outils -> outils de développement).*
- *Node et Angular sont des productions de Google, comme Chrome, ce qui rend l'environnement de développement homogène.*

2.1.8) Génération de code

Angular génère directement le code des modules, composants, services et autres éléments propres à Angular. Pour cela sa syntaxe est simple :

```
# ng generate <element name>
```

ou de façon plus concise

```
# ng g <element name>
```

où **element** peut être : **class, component, directive, enum, guard, interface, module, pipe, service**, et **name** est le nom que vous voulez donner à l'élément.

A noter que les éléments sont générés dans le répertoire `src/app`, et que le code du module général de l'application `app.module.ts` est automatiquement modifié pour en tenir compte. Ce la dit, comme nous le verrons, il nous faudra organiser nos fichiers et nos modules d'une autre façon pour plus de lisibilité et de maintenabilité. Cela dit, nous utiliserons ce générateur d'Angular tout au long de ce cours.

2.1.9) Inclusion de bibliothèques externes

Angular et Angular sont capable d'intégrer toute autre bibliothèque javascript, ce qui est un grand avantage. Chaque bibliothèque a ses règles d'installation, comme nous le verrons plus loin par exemple avec Bootstrap, mais en général on utilise npm pour ce faire :

```
# npm install ma_bibliothèque --save
```

l'option --`save` sert à inscrire la bibliothèque importée dans le fichier package.json qui fait notamment état de tous les éléments logiciels utiles à l'application et au compilateur. La bibliothèque est alors importée depuis le réseau et rangée dans le répertoire `node_modules`.

2.1.10) Compilation (build)

Une fois la phase de développement terminée, il faut compiler le code afin de le minifier et de l'optimiser pour obtenir de meilleures performances. La commande à lancer est :

```
# ng build
```

Le résultat de la compilation est stockée dans le répertoire `dist` créé par Angular. Par défault cette commande utilisera le fichier `environment.prod.ts`, mais il est possible de spécifier pour quelle environnement on souhaite compiler, par exemple :

```
ng build –environnement XXXX
```

se basera sur le fichier `environment.XXXX.ts`.

2.1.11) Tests

L'objet de ce cours ne concerne ni les tests unitaires ni les tests fonctionnels, nous renvoyons donc le lecteur à la documentation s'il veut en savoir plus sur ces domaines, qui sont des univers à eux seuls. Notez simplement ici que Angular crée des fichiers `.specs.ts`, destinés aux tests unitaires, à la génération de chaque composant nécessaire à Angular, et qu'il crée un répertoire `e2e` destiné à recevoir les codes de tests fonctionnels.

2.1.12) Typescript

En matière de front-end web on pourrait s'attendre à coder en javascript. Angular le permet, mais il préconise plutôt l'utilisation de Typescript. Il s'agit d'un langage objet surcouche du javascript qui se compile en javascript. Il est destiné à plonger le développeur rompu aux langages objets classiques (java, C++, …) dans un monde qui lui est plus familier que le monde javascript. Ce dernier est en effet un langage « libertaire » qui utilise la notion atypique de prototypage et qui déroute largement le développeur académique. Typescript

est open source, développé par Microsoft et publié sous licence Apache. C'est un univers à lui tout seul qui va au-delà de l'objectif de ce cours, nous laissons donc le lecteur se renseigner à son propos sur la documentation officielle : https://www.typescriptlang.org/docs/home.html.

2.2) Intégrer Bootstrap

Pour les besoins de ce cours nous allons intégrer Bootstrap (https://getbootstrap.com), qui est la bibliothèque CSS de base la plus répandue, dans sa dernière version. Bootstrap ne sera pleinement fonctionnel qu'en intégrant également font-awesome, Jquery.js et popper.js, voici donc comment installer ces bibliothèques pour les utiliser dans une application Angular :

```
# cd my-app
# npm install bootstrap --save
```

• `npm install` *utilisé avec l'option* **`--save`** *permet d'inclure l'élément installé au fichier package.json.*

Pour intégrer complètement les css Bootstrap, il faut ensuite ouvrir le fichier `.angular-cli.json` et modifier les sections `styles` et `scripts` :

```
"styles": [
  "../node_modules/bootstrap/dist/css/bootstrap.min.css",
  "styles.css"
],
"scripts": [
  "../node_modules/bootstrap/dist/js/bootstrap.min.js"
],
```

Une fois cette installation opérée, Bootstrap est disponible dans toute l'application.

Cette façon de faire sera utilisée à l'indentique pour toute autre bibliothèque extérieure à l'application mais indispensable à cette dernière (Ag-grid, PrimeNg, ...).

3) Angular

3.1) Organisation des éléments

Dans Angular on distingue deux types d'éléments : les modules et les autres. Au nombre des autres on compte les components, les directives, les pipes et les services. Nous décrirons ces éléments plus loin dans ce cours.

Un module est a priori « standalone », c'est à dire qu'il se suffit à lui même, tandis que les autres éléments dépendront forcément d'un module. Les modules peuvent embarquer d'autres modules.

Une application Angular nécessite donc la présence d'au moins un module, appelé **`app.module.ts`** de façon standard. C'est la racine de l'application qui sera structurée en cascade, c'est à dire en arborescence de

modules contenant d'autres modules et/ou certains des autres éléments. Tous les composants, directives, services, etc., ayant une portée globale à l'application doivent être déclarés dans ce module.

Un des éléments autre le plus indispensable à Angular est le **`component`**. C'est lui qui embarque le HTML qui sera affiché. Un composant peut être global à l'application, comme un header de site qui devra apparaître sur toutes les pages par exemple, mais il peut aussi être réservé à un seul module, ou bien utilisé par plusieurs modules sans être global. Il en va de même pour les autres éléments, pipes, directives et services.

A cause du partage possible des éléments, il est nécessaire d'organiser l'arborescence des fichiers les contenant afin qu'ils reflètent au mieux leur arborescence fonctionnelle dans l'application. Une bonne pratique est donc de s'organiser comme suit :

- créer un répertoire lib contenant les modules et éléments globaux ou partageables
- créer un répertoire pour chaque page qui sera affiché par l'application, contenant son module et ses éléments dédiés
- inclure les éléments dédiés à un seul module dans le répertoire de ce module

Une telle organisation donnera une arborescence comme celle qui suit, où m représente un module, p une page, d une directive, etc. :

```
app
├── lib
│   ├── modules
│   │   ├── m1
│   │   │   ├── c1
│   │   │   ├── c2
│   │   │   ├── ...
│   │   │   ├── d1
│   │   │   ├── d2
│   │   │   ├── ...
│   │   ├── m2
│   │   ├── ...
│   ├── components
│   │   ├── c1
│   │   ├── c2
│   │   ├── ...
│   ├── directives
│   │   ├── d1
│   │   ├── d2
│   │   ├── ...
│   ├── pipes
│   │   ├── p1
│   │   ├── p2
│   │   ├── ...
│   ├── services
│   │   ├── s1
│   │   ├── s2
│   │   ├── ...
├── pages
│   ├── page1
│   │   ├── m
│   │   ├── c1
│   │   ├── c2
│   │   ├── ...
│   │   ├── d1
```

```
│   │   ├── d2
│   │   ├── ...
│   │
│   ├── page2
│   │   ├── m
│   │   ├── c1
│   │   ├── c2
│   │   ├── ...
│   │   ├── d1
│   │   ├── d2
│   │   ├── ...
```

3.2) Module

3.2.1) Introduction

En Angular ce ne sont pas les composants qui sont partageables, mais les modules. Si vous essayez d'embarquer un même composant dans deux modules différents, vous obtiendrez une erreur à la compilation, vous disant que le composant est embarqué dans plus de un seul module.

Un module embarque toutes sortes d'éléments, par exemple des composants, des pipes et des services :

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MyOtherModule } from 'lib/modules/my-module.module.ts';
import { Composant1 } from './composant1.component.ts';
import { Composant2 } from './composant2.component.ts';
import { Pipe1 } from './pipes/pipe.component.ts';
import { MyService } from './services/my-service.service.ts';

@NgModule({
  declarations: [
    Composant1,
    Composant2
  ]
  imports: [
    CommonModule,
    MyOtherModule
  ],
  providers: [
    Pipe1,
    MyService
  ],
  exports : [
    Composant1,
    Composant2
  ]
})
export class MyModule { }
```

Les modules embarqués n'apparaissent que dans les **`imports`**, tandis que les composants peuvent apparaître dans les **`declarations`** mais aussi les **`exports`**, si on veut que le composant soit disponible lorsqu'un module différent importe le module de base (**`MyModule`** ci-dessus). Les services et les pipes sont quant à eux déclarés dans les **`providers`**.

Les éléments embaqués dans un module ne seront reconnus et actifs que dans l'arborescence dont le module

est l'initiateur.

La figure ci-dessous décrit l'architecture modulaire d'Angular :

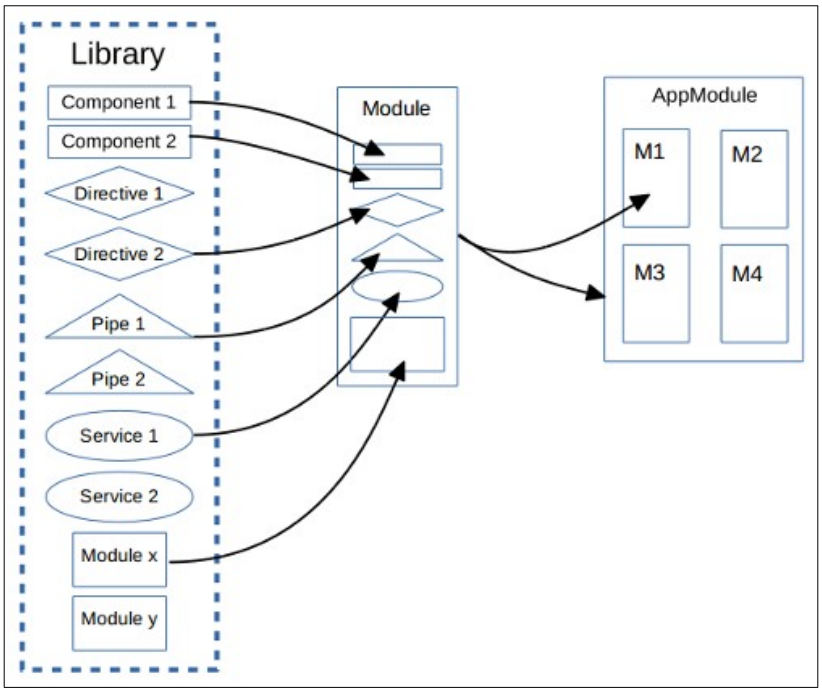


Figure 1 : Angular est constitué de modules qui peuvent contenir d'autres modules, des composants, des directives, des filtres (pipes) et des services.

3.2.2) Création

On génère un module avec la commande :

```
ng g module my-module
```

Le module généré le sera en camel case, c'est à dire que la commande ci-dessus créera le module logique **`MyModule`** dans le fichier **`my-module.module.ts`**.

Puisque les modules ont vocation à être réutilisables, une bonne pratique est de constituer une bibliothèque dans laquelle on stockera les différents éléments nécessaires à Angular. Cette bibliothèque peut se trouver dans n'importe quel répertoire à l'intérieur du répertoire **`app`**.

À titre d'exemple, qui nous servira tout au long de ce cours, créons donc cette bibliothèque et plaçons nous dans le répertoire des modules :

```
# cd my-app/src/app
# mkdir lib
# mkdir lib/modules
```

```
# cd lib/modules
```

Nous allons créer un module (avec Angular) qui sera chargé d'afficher un header à notre application. Nous l'appelons site-header (mais il pourrait s'appeler toto, titi, ou n'importe quoi d'autre) :

```
# ng generate module site-header
```

Angular génère alors un répertoire `site-header` contenant le fichier `site-header.module.ts` :

```
lib
├── modules
│   ├── site-header
│   │   └── site-header.module.ts
```

Voici le contenu du fichier `site-header.module.ts` :

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class SiteHeaderModule { }
```

A noter que le module que nous avons créé exporte la classe **`SiteHeaderModule`**. *C'est cette classe qui nous servira de référence si nous voulons intégrer ce module à un autre module, comme nous le verrons ci-dessous.*

3.2.3) Structure

Nous voyons qu'un module élémentaire fait appel à des modules internes de Angular : `ngModule` et `CommonModule`. Sans ces modules internes le module que nous venons de créer ne fonctionnera pas.

NgModule est le plus fondamental des modules d'Angular, c'est lui qui autorisera l'utilisation de la syntaxe

```
@NgModule({...})
export class ...
```

`CommonModule` est le module qui permettra l'utilisation des directives natives d'Angular (**`ngFor`**, **`ngIf`**, **`...`**), ainsi que ses filtres (pipes) natifs (**`DatePipe`**, **`UpperCasePipe`**, ...). Nous verrons plus loin leur signification et leur usage.

On constate que le module spécifie des **`imports`** et des **`declarations`**. En fait il est possible de spécifier d'autres éléments de paramétrages, décrits dans le tableau suivant.

Paramètre	Description

`imports`	Un tableau listant tous les modules nécessaires au fonctionnement du module considéré. On peut y trouver des modules personnalisés, comme des modules internes d'Angular, par exemple **`BrowserModule`** qui permet les interactions avec le navigateur, ou `FormModule` qui permet la gestion des formulaires. *A noter que **`NgModule`** n'a pas besoin d'être importé, car il est structurel.* *Le module natif **`BrowserModule`** n'a besoin d'être importé qu'une seule fois, dans le module racine **`AppModule`**.*
`declarations`	Un tableau listant les composants, directives et pipes qui seront utilisés par le module.
`bootstrap`	Un tableau contenant les composants racines sur lesquels se base Angular pour construire l'application. En pratique une application ne possède qu'un seul bootstrap applelé **`AppComponent`** dont le sélecteur apparaît dans le fichier **`index.html`**. Ainsi le paramètre bootstrap n'existe que dans le module racine **`AppModule`** du fichier **`app.module.ts`**.
`exports`	Un tableau listant les composants, directives et pipes que le module exportera vers d'autres modules, où ils seront utilisables.
`providers`	Un tableau qui liste les services utilisés par le module, par exemple un service permettant de récupérer des données sur un serveur.

3.2.4) Intégration à l'application

Nous allons intégrer le module site-header que nous venons de créer dans notre application, c'est à dire dans le module `AppModule` décrit dans le fichier `app.module.ts`. Voici comment :

app.module.ts

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

//modules personnalisés
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';

//composants
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    SiteHeaderModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ayant modifié ainsi le fichier `app.module.ts`, le live-loading d'Angular provoque le rechargement automatique de l'application, mais pour l'instant nous ne voyons aucun changement sur la page affichée par

le navigateur, et pour cause, nous n'avons encore spécifié aucun nouveau composant, or ce sont les composants qui spécifient le code HTML que l'application doit afficher. Il nous faut donc créer un composant au module `SiteHeaderModule`, c'est le sujet du chapitre suivant.

3.3) Component

3.3.1) Introduction

Les composants sont sans doute les éléments les plus essentiels d'une application Angular. Ce sont eux qui vont produire le code HTML, les feuilles de styles, les interactions utilisateurs et en général toute la logique associée au DOM. C'est à leur écriture que vous passerez le plus de temps.

3.3.2) Création

Angular permet de générer un composant de base de façon simple avec la commande :

```
# ng generate component site-header
```

Une fois encore, une bonne pratique est de le créer dans la bibliothèque des éléments. A ce stade on peut se demander si le composant qui va spécifier le code du header de notre site doit se trouver dans le répertoire `lib/modules/site-header`, ou plutôt dans un répertoire `lib/components/site-header`. Les deux sont possibles, comme d'ailleurs tout autre emplacement à l'intérieur du répertoire `app`, c'est une question de choix d'organisation.

Un composant générique, utilisable par différents modules, sera mieux placé dans lib/components, tandis qu'un composant dédié à un module sera mieux placé s'il l'est à l'intérieur du répertoire du module concerné, ici lib/modules/site-header. A titre d'exemple nous choisissons de le placer dans lib/components, après avoir créé ce répertoire :

```
# mkdir lib/components
# cd lib/components
# ng generate component site-header
```

Angular génère alors le répertoire lib/components/site-header, contenant les fichiers suivants :

```
lib
├── components
│   ├── site-header
│   │   ├── site-header.component.css
│   │   ├── site-header.component.html
│   │   ├── site-header.component.spec.ts
│   │   ├── site-header.component.ts
```

Voici à quoi servent ces fichiers :

Fichier	Utilité
`site-header.component.css`	Contient la feuille de style à associer au composants. *Par défaut les styles appliqués sont ceux du fichier*

	`src/styles.css`, *mais ces derniers sont surchargés par le fichier CSS du composant.*
`site-header.component.html`	Contient le code HTML qui s'affichera
`site-header.component.spec.ts`	Contient le code de test du composant
`site-header.component.ts`	Contient le code Angular du composant

A noter aussi qu'il modifie automatiquement le fichier `app.module.ts`, pour le faire importer ce composant :

app.module.ts

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

//modules personnalisés
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';

//composants
import { AppComponent } from './app.component';
import { SiteHeaderComponent } from
'./lib/components/site-header/site-header.component';

@NgModule({
  declarations: [
    AppComponent,
    SiteHeaderComponent
  ],
  imports: [
    BrowserModule,
    SiteHeaderModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Cela est très sympathique de la part d'Angular de nous faciliter la tâche, mais cette façon de faire peut vite devenir problématique. En effet si notre application est un tant soit peu complexe, le fichier `app.module.ts` va devenir une jungle où des dizaines de composants vont figurer, le rendant illisible. Nous allons donc supprimer les lignes soulignées en bleu dans le fichier `app.module.ts`, et plutôt faire intégrer ce composant par le module site-header, c'est à dire modifier le fichier `site-header.module.ts` de la façon suivante :

site-header.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { SiteHeaderComponent } from
'../../components/site-header/site-header.component';

@NgModule({
```

```
  imports: [
    CommonModule
  ],
  declarations: [
    SiteHeaderComponent
  ]
})
export class SiteHeaderModule { }
```

Pour que le composant site-header soit néanmoins accessible pour toutes les pages de l'application, il est nécessaire d'importer le module `SiteHeaderModule` dans le module général de l'application :

app.module.ts :

```
...
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';
...

@NgModule({
...
  imports: [
    ...
    SiteHeaderModule
  ],
...
})
...
```

3.3.3) Description

Le live-loading d'Angular ayant fonctionné, nous ne voyons toujours rien apparaître sur la page de notre application. Nous allons y venir, mais pour ce faire, il nous faut d'abord décrire la structure d'un composant.

Voici le fichier `site-header.components.ts` créé par Angular :

site-header.component.ts

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

@Component({
  selector: 'app-site-header',
  templateUrl: './site-header.component.html',
  styleUrls: ['./site-header.component.css']
})
export class SiteHeaderComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

La première chose à constater est l'import de `Component` et `OnInit` depuis le cœur d'Angular. Ces éléments sont nécessaires pour décrire un composant personnalisé à partir de la structure native des composants (Component), et pour exécuter son code au chargement de l'application (OnInit) par l'usage de la fonction `ngOnInit()`.

La seconde chose remarquable est l'export de la classe `SiteHeaderComponent`. C'est dans cet objet que sera écrit toute la logique que nous voulons associer à notre composant : variables, fonctions, … Nous verrons plus loin que les éléments de cette logique pourront alors être appelés depuis les tags HTML du template associé.

Enfin, on constate que la fonction à décorateur `@Component` comporte des paramètres. Voici leur signification :

templateUrl

Ce paramètre permet de spécifier le fichier qui contient le code HTML du composant, dans notre cas il s'agit de
`./site-header.component.html`.

Il peut être remplacé par le paramètre simple `template`, et dans ce cas le code HTML n'est pas stocké dans un fichier, mais écrit directement dans le composant. Par exemple :

```
  template: `<div>Ceci est le code HTML du composant</div>`
```

Noter qu'ici le code est inclus entre deux quotes inverses : `` ` ``

Cette seconde façon de faire est cependant à déconseiller car elle rend vite illisible le fichier du composant.

selector

Cet paramètre permet de définir le tag HTML qui sera rempli par le code HTML spécifié par le paramètre précédent, `templateUrl`, ou `template`. En pratique avec notre exemple , cela advient dès que le tag

```
<app-site-header></app-site-header>
```

apparaît dans dans le code HTML d'un composant qui reconnaît notre composant `SiteHeaderComponent`.

styleUrls

Ce paramètre permet de définir le (ou les) fichier qui contiendra CSS à appliquer au code HTML du composant, et **qui se superposeront aux styles déjà décrits dans le fichier** `src/styles.css`.

*Attention, ce paramètre est un **tableau** d'URLs :* `['url1', 'url2', ...]`

Ici encore il est possible de remplacer ce paramètre par le paramètre styles, qui est aussi un tableau, contenant cette fois-ci directement le code CSS désiré. Par exemple :

```
styles : [
  `
     p { color : blue}
     h1 {background-color : grey}
  `,
  `
```

```
    h3 { font-style : italic}
  `
]
```

Noter qu'ici encore le code est inclus entre deux quotes inverses : `

La même remarque que pour le template HTML est cependant à faire : cette façon de procéder rend vite le fichier du composant illisible, elle est donc à déconseiller.

Encapsulation

Ce paramètre n'apparaît pas dans le composant créé par Angular car son utilité n'est pas évidente a priori. Il permet de spécifier de quelle façon on souhaite appliquer les CSS. Pour l'utiliser il faut importer ce module natif `ViewEncapsulation` d'Angular dans notre fichier `site-header.component.ts` :

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

Puis on ajoute ce paramètre à la définition de notre composant :

```
...
@Component({
  selector: 'app-site-header',
  templateUrl: './site-header.component.html',
  styleUrls: ['./site-header.component.css'],
  encapsulation : ViewEncapsulation.Native
})
...
```

Les valeurs de l'encapsulation peuvent être :

- **`ViewEncapsulation.Emulated`** : c'est la valeur par défaut, le style du composant ne s'applique qu'à lui
- **`ViewEncapsulation.None`** : les styles du composant explicitement stipulés deviennent les styles de tous les tag HTML des autres composants ayant la même classe
- **`ViewEncapsulation.Native`** : tous les styles du composant qui ne sont pas explicitement stipulés dans son CSS sont oubliés

3.3.4) Exemple de notre composant SiteHeaderComponent

Nous allons enfin voir apparaître quelque chose de nouveau sur la page de notre application. Pour ce faire modifions le fichier `site-header.component.html` comme suit :

site-header.component.html

```
<nav class="navbar navbar-expand-md navbar-dark bg-primary fixed-top">
  <a class="navbar-brand" href="">My App</a>
</nav>
```

Ce qui est défini dans ce code HTML est une navbar de Bootstrap, dont la couleur de fond est « primary », et qui reste fixe en haut de la fenêtre du navigateur, même si la page est scrollée.

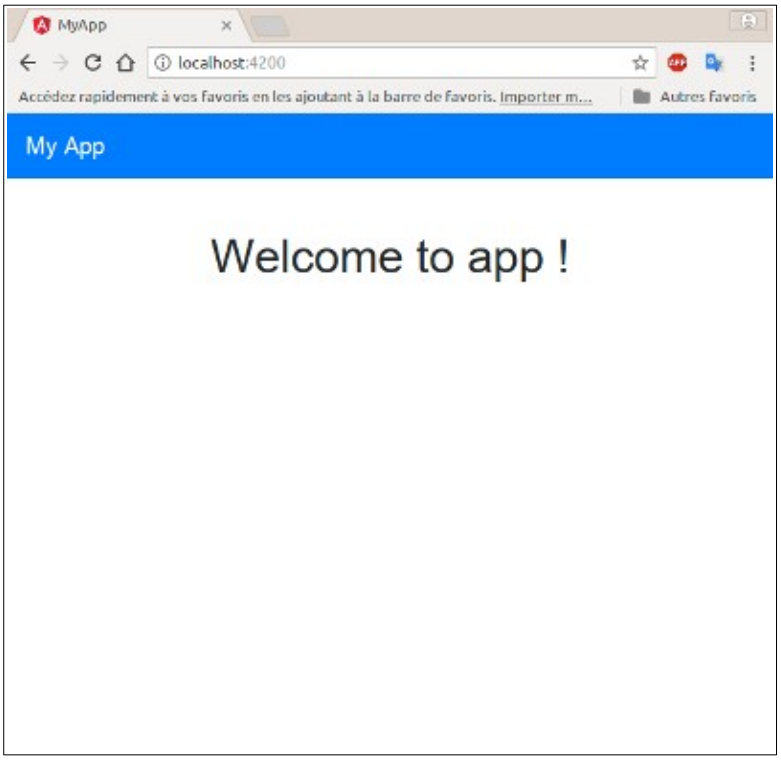
Ceci fait, il faut faire appel au decorator de notre composant `SiteHeaderComponent`, dans le template du composant principal `AppComponent`, décrit dans le fichier `app.component.html`. Ce dernier, une fois débarrassé de son image et des liens qui ne nous importe pas, devient donc :

app.component.html

```
<app-site-header></app-site-header>

<div style="text-align:center;margin-top:100px">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>
```

*Notez qu'on a ajouté le style **`margin-top:100px`**, afin que le site header ne recouvre pas le titre de la page.*

On sauvegarde les fichiers modifiés, et le live-reload de Angular nous affiche maintenant :



Nous en avons terminé avec la présentation des composants. Dans le chapitre « Nested components » nous reviendrons sur ces éléments fondamentaux et verrons comment les imbriquer les uns dans les autres, et aussi comment faire communiquer des données et des événements entre eux.

3.4) Data et event bindings

Nous allons maintenant décrire ce que Angular a sans doute de plus remarquable, à savoir les data et event bindings qui permettent de lier le DOM et le javascript de façon automatique.

A titre d'introduction notons que le template de notre application exemple, `app.component.html`, fait appel à l'élément `{{title}}`, tandis que la classe associée `AppComponent`, décrite dans le fichier `app.component.ts` définit la propriété `title` :

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

Vous aurez compris que la syntaxe `{{title}}` stipulée dans le HTML fait appel automatiquement à la propriété `title` de la classe AppComponent, sans qu'il y ait besoin d'écrire de code supplémentaire, par exemple en utilisant un `getElementById()` qui aurait été nécessaire pour effectuer cette opération en javascript pur. Cette capacité d'Angular est nommé le « data binding », qui s'accompagne également du « event binding ». Cette syntaxe à double accolades est nommée « property binding », mais il existe 3 autres sortes de bindings que nous allons passer en revue.

En général le binding consiste à écrire une syntaxe particulière dans le HTML qui se réfère à des propriétés ou des fonctions décrites dans le javascript, ou typescript, associé au HTML.

3.4.1) Property binding {{ }}

Comme nous venons de le voir sa syntaxe est la suivante :

```
{{myProperty}}
```

Elle permet d'afficher la valeur d'une variable, ou d'une propriété, **à des emplacements du HTML prévu pour l'affichage de texte**, au sens large. A titre d'exemple nous allons définir un l'objet `myObject` dans la classe `AppComponent` de notre exemple de la façon suivante :

app.component.ts

```
export class AppComponent {
  title = 'app';

  myObject = {
    id : 123,
    title : 'Ceci est un exemple',
    tableau : [ "alpha", "beta", "gamma"],
    preferences : {
      couleur : "bleu",
      fruit : "pomme"
    }
  }
```

```
}
```

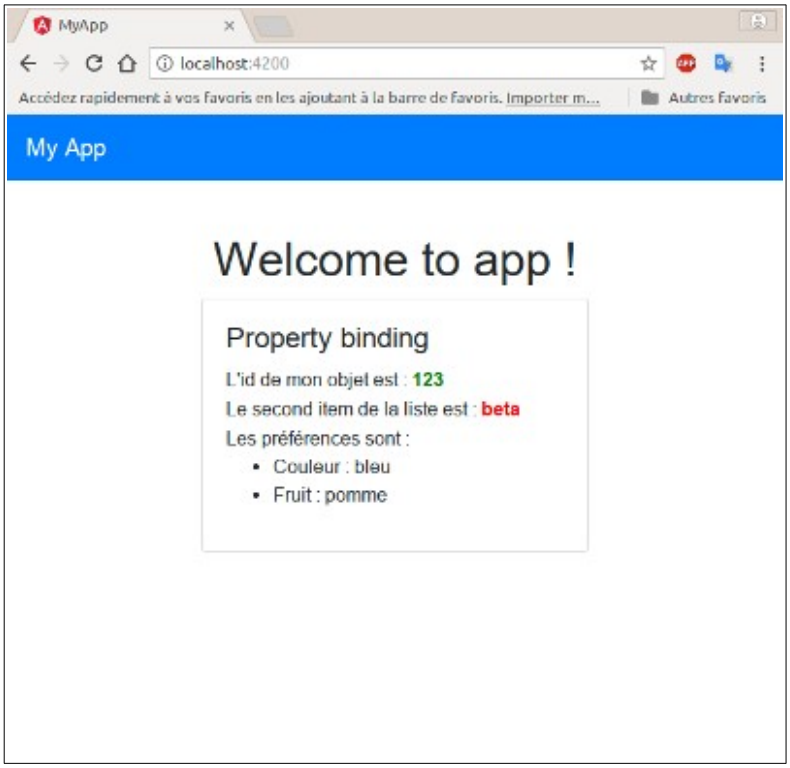
Notez que nous n'avons pas déclaré la variable `myObject` *en tant que* `public` *ou* `private`*, c'est parce que par défaut les variables sans cette précision sont* `public`*.*

Ensuite nous allons modifier le template HTML de ce composant, c'est à dire le fichier `app.component.html`, de la façon suivante :

app.component.html

```
<app-site-header></app-site-header>

<div style="text-align:center;margin-top:100px">
  <h1>
    Welcome to {{ title }} !
  </h1>
</div>

<div class="card" style="margin : auto;width:50%">
  <div class="card-body">
    <h4>Property binding</h4>
    <div>L'id de mon objet est :
      <span style="color:green;font-weight:bold">
        {{myObject.id}}
      </span>
    </div>
    <div>Le second item de la liste est :
      <span style="color:red;font-weight:bold">
          {{myObject.tableau[1]}}
      </span>
    </div>
    <div>
      Les préférences sont :
      <ul>
        <li>Couleur : {{myObject.preferences.couleur}}</li>
        <li>Fruit : {{myObject.preferences.fruit}}</li>
      </ul>
    </div>
  </div>
</div>
```

Le résultat apparaît dans le navigateur :



Le property binding est donc relativement trivial. On remarque cependant qu'il ne se réfère pas uniquement à des variables simples, mais aussi à des propriétés d'objets où à des éléments de tableau.

Ce que nous venons de réaliser ici est du « **one way data binding** », du javascript vers le HTML. Pour s'en persuader nous allons modifier la couleur préférée dans le script, de « bleu » vers « rouge », après avoir attendu 3 secondes, pour laisser le temps à la page de se charger et afficher la valeur initiale « bleu ». Pour cela nous modifions notre script de la façon suivante :

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';

  myObject = {
    id : 123,
    title : 'Ceci est un exemple',
    tableau : [ "alpha", "beta", "gamma"],
    preferences : {
      couleur : "bleu",
      fruit : "pomme"
    }
  }

  constructor() { }

  ngOnInit() {
```

```
    setTimeout(()=>{
      this.myObject.preferences.couleur = "rouge";
    },3000);
  }

}
```

Notez l'import du module `OnInit` *nécessaire à effectuer des opérations au chargement de la page grâce à la fonction* `ngOnInit()`

Attention, le temps est compté en millièmes de secondes en javascript, ainsi 3000 millisecondes représente 3 secondes.

Le résultat est qu'au bout de trois secondes la page affiche « rouge » pour la couleur préférée :

3.4.2) Attribute binding [ ]

Il est possible d'associer une valeur définie par une variable du javascript à un attribut du DOM.
La syntaxe générale est la suivante :

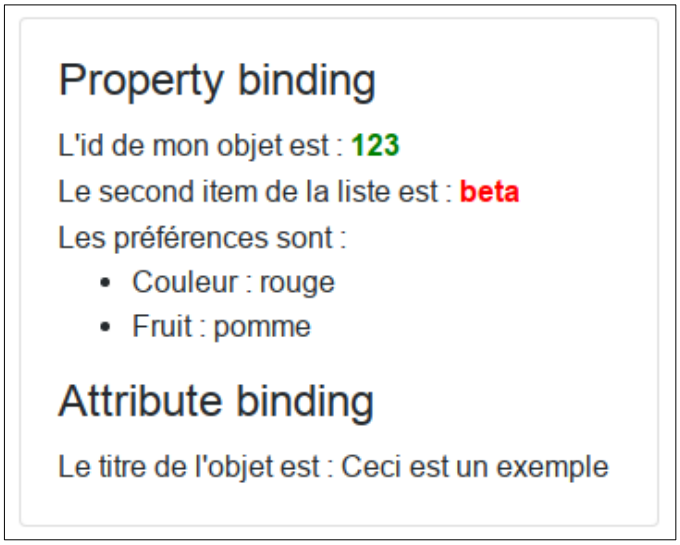
```
<myTag [myAttribute]="myVariable"></myTag>
```

Pour illustrer ce fonctionnement, ajoutons les lignes suivantes à notre template HTML :

app.module.html

```
<h3>Attribute binding</h3>
<div>Le titre de l'objet est : <span [innerText]="myObject.title"></span></div>
```

Puisque nous n'avons pas modifié `myObject` dans notre script, l'affichage obtenu est alors :



Attention :
- *certains attributs sont reconnus par le DOM, mais pas par le HTML, c'est le cas de `innerText` par exemple,*
- *certains attributs sont reconnus par le DOM et le HTML, c'est le cas de `id` par exemple.*
- *certains attributs sont reconnus par le HTML mais pas par le DOM, c'est le cas de `style` par exemple*

Seuls les attributs du DOM peuvent être utilisés par l'attribute binding.

L'attribute binding est aussi un one way binding, du javascript vers le HTML. Dans la section « Nested Components » nous verrons qu'il est aussi utilisé pour échanger des données entre composants.

3.4.3) Event binding ( )

Pour l'instant nous avons vu que le one way binding ne fonctionnait que du javascript vers le HTML, mais cela n'importe pas les actions de l'utilisateur depuis l'interface HTML vers le javascript. C'est dans ce but qu'Angular propose l'event binding qui consiste à déclencher une fonction dans le script lorsque l'utilisateur agit un élément de l'interface.

Sa syntaxe est la suivante :

```
<myTag (myAction)="myFunction()"></myTag>
```

Ici `myAction` est un « DOM event » tel que click, keyup, scroll, … Les DOM events sont nombreux, on en trouve une liste exhaustive dans la documentation Mozilla : https://developer.mozilla.org/fr/docs/Web/Events

Par exemple, un bouton cliquable déclenchant l'exécution de `myFunction()` sera :

```
<button (click)= "myFunction()"></button>
```

Bien sûr, `myFunction()` doit être présente dans la classe du composant. Ajoutons donc cette fonction à notre classe `AppComponent` :

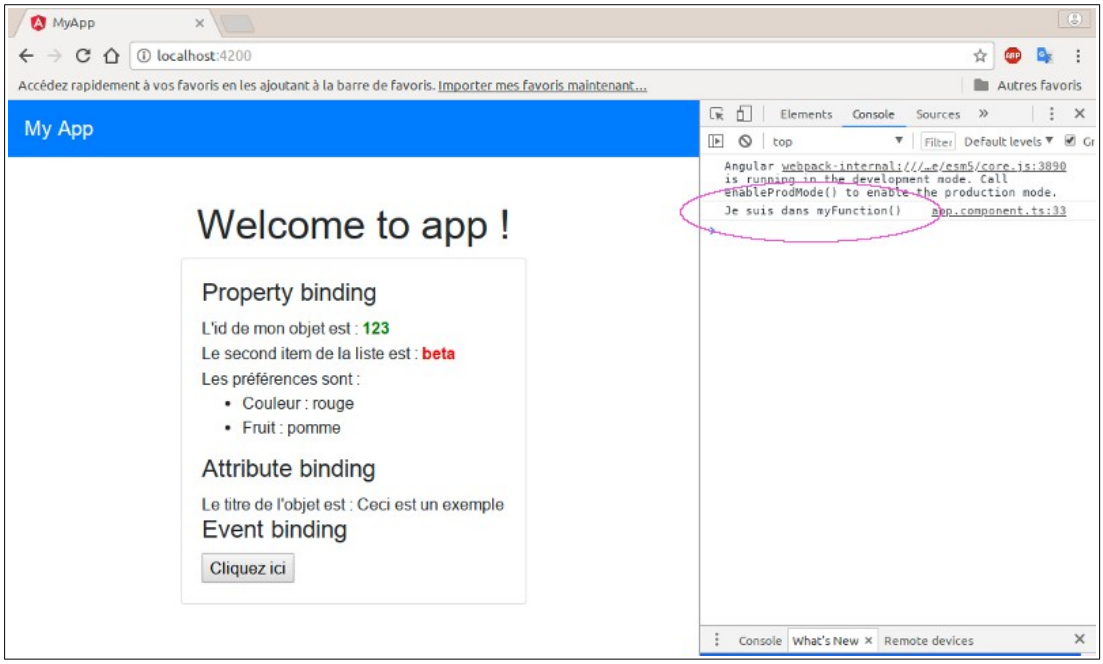
app.component.ts

```
export class AppComponent {
  ...

  myFunction() {
    console.log('Je suis dans myFunction()');
  }

}
```

et en regard ajoutons ce lignes dans le template `app.component.html` :

```
<h3>Event binding</h3>
<button (click)="myFunction()">Cliquez ici</button>
```

Le bouton apparaît sur la page et en cliquant dessus le texte apparaît dans la console du débugger :



L'event binding peut être considéré à nouveau comme un « one way binding », mais cette fois-ci depuis le HTML vers le javascript.

Il nous reste maintenant à voir le two way data binding.

3.4.4) Two way data binding [(ngModel)]

Si on y réfléchit bien, un utilisateur a relativement peu de possibilités de modifier explicitement une valeur du DOM. Il peut utiliser les input, select, textarea ou checkbox. Ce sont donc ces seuls tags HTML qui pourront contenir la syntaxe du two way binding, à savoir le mot `ngModel` inclus dans une « banana box » [( )] :

```
<input [(ngModel)]="myVariable">
```

L'input sera rempli par la valeur de `myVariable` donnée par le javascript, et en retour si l'utilisateur change

le contenu de l'input dans le HTML, `myVariable` prendra cette modification en compte dans le javascript.

Pour utiliser cette fonctionnalité il faut au préalable importer le module `FormsModule` d'angular, qui gère tous les composants des forms, c'est à dire modifier le module dont dépend le composant, dans notre cas `app.module.ts`, de la façon suivante :

app.module.ts

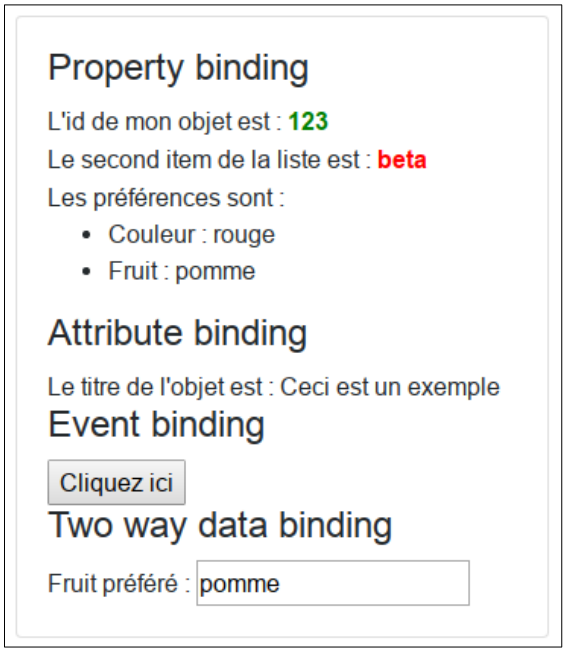
```
//Modules natifs
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';

//modules personnalisés
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';

//composants
import { AppComponent } from './app.component';
//import { LinksListComponent } from './lib/components/links-list/links-
list.component';

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

Ceci fait, modifions le template HTML `app.component.html` en y ajoutant le code suivant :

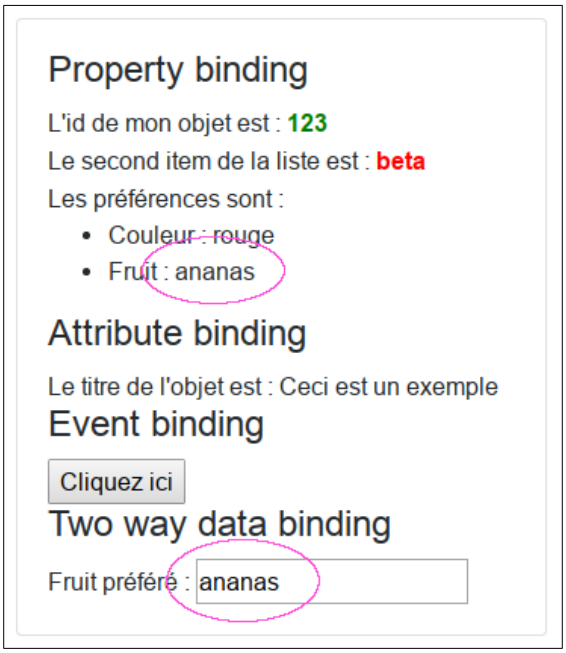
app.component.html

```
<h3>Two way data binding</h3>
Fruit préféré :
<input type="text" [(ngModel)]="myObject.preferences.fruit" />
```

Une fois la page rechargée, elle affiche :



Si l'utilisateur change la valeur de l'input pour « ananas », l'affichage devient :



On remarque que la modification de `myObject.preferences.fruit` dans l'input modifie simultanément la valeur correspondante dans le property binding `{{myObject.preferences.fruit}}`. Ceci montre que le javascript a bien été modifié lorsque l'utilisateur a modifié l'input.

Nous verrons que le two way data binding peut aussi se faire entre composants imbriqués, ou nested components, sous une forme qui n'utilise pas le mot clé `ngModel`.

3.5) Structural directives

Une autre spécificité remarquable d'Angular est de permettre d'inclure de la logique directement dans le HTML, alors qu'en javascript classique toute la logique est codée dans les scripts. Cette opération est réalisée par les structural directives, qui sont au nombre de 3 :

- **`*ngIf`** : le tag ne doit être affiché que si une variable possède une certaine valeur
- **`*ngFor`** : répète un tag autant de fois qu'il y a d'éléments dans une liste
- **`[ngSwitch]`** : dans une liste de tags seul celui correspondant à un critère particulier doit être affiché

Passons les en revue.

3.5.1) *ngIf

La syntaxe typique de `*ngIf` dans le HTML est la suivante :

```
<mytag *ngIf="expression">...</mytag>
```

Le tag ne s'affichera que si *`expression`*, qui est une expression logique, est vraie. A titre d'exemple, on peut demander que la lige « id de mon objet » contenu dans notre application, ne s'affiche que si l'id est supérieur à 200, pour ce faire nous modifions le HTML de app.component.ts de la façon suivante :

app.component.html

```
    <div *ngIf="myObject.id>200">
       L'id de mon objet est :
       <span style="color:green;font-weight:bold">
          {{myObject.id}}
       </span>
    </div>
```

En l'occurrence nous avons donné à `myObject.id` la valeur `123`, qui est inférieure à `200`, par suite le tag écrit ci-dessus ne s'affichera pas.

3.5.2) *ngFor

La syntaxe typique de `*ngFor` dans le HTML est la suivante :

```
<mytag *ngFor="let item of myArray">...</mytag>
```

Le tag sera alors répété autant de fois qu'il y a d'élément dans `myArray`. A chaque itération `item` prendra la valeur des éléments successifs de `myArray`. Les mots `item` et `myArray` ne sont en rien obligatoires, vous pouvez choisir ce que vous voulez à la place.

Appliquons cela au tableau myObject.table de notre exemple, en modifiant le template du composant AppComponent :

app.component.html

```
    <h4>Property binding</h4>
    <div>
      Les éléments de la liste sont :
      <ul>
```

```
        <li *ngFor="let item of myObject.tableau">
          valeur =
          <span style="color:red;font-weight:bold">
            {{item}}
          </span>
        </li>
      </ul>
    </div>
```

L'affichage donnera alors :

A noter qu'il est parfois utile d'utiliser l'index de l'élément affiché. Celui-ci se récupère de la façon suivante :

```
  <li *ngFor="let item of myObject.tableau; let idx = index">
```

Attention, l'utilisation directe de `index` dans le code ne fonctionnera pas, il est nécessaire de définir une autre variable, par exemple `idx` dans notre cas, mais vous pouvez choisir ce que vous voulez à la place.

Ainsi le code suivant :

app.component.html

```
    <div>
      Les éléments de la liste sont :
      <ul>
        <li *ngFor="let item of myObject.tableau; let idx = index">
          valeur {{idx}} =
          <span style="color:red;font-weight:bold">
            {{item}}
          </span>
        </li>
      </ul>
    </div>
```

donnera l'affichage

3.5.3) [ngSwitch]

Cette directive est l'équivalent du switch de javascript. On comprendra donc qu'elle soit associée à deux autre directives : ***ngSwitchCase** et ***ngSwitchDefault**.

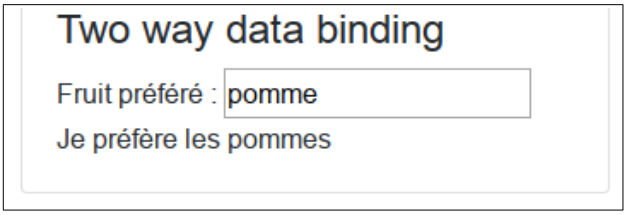
Avec l'exemple de notre application, la syntaxe est la suivante :

app.component.html

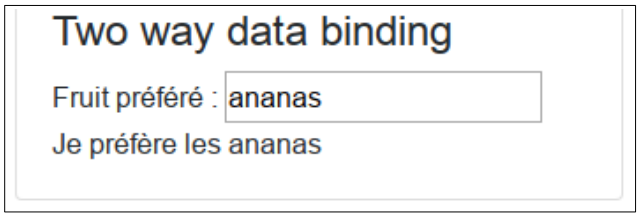
```
<h4>Two way data binding</h4>
Fruit préféré :
<input type="text" [(ngModel)]="myObject.preferences.fruit" />
<div [ngSwitch]="myObject.preferences.fruit">
  <span *ngSwitchCase="'pomme'">Je préfère les pommes</span>
  <span *ngSwitchCase="'ananas'">Je préfère les ananas</span>
  <span *ngSwitchCase="'banane'">Je préfère les bananes</span>
  <span *ngSwitchDefault>Je n'ai pas de préférence</span>
</div>
```

Attention :
- *on n'utilise pas `*ngSwitch` mais **`[ngSwitch]`**, cependant on utilise **`*ngSwitchCase`** et **`*ngSwitchDefault`**, car en réalité `ngSwitch` est un attribute directive, les deux autres étant des structural directives*
- *notez bien la syntaxe quotes entre double quotes : **`"'pomme'"`***

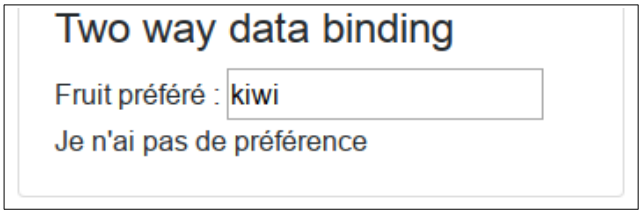
Après le live-loading l'affichage de ce code donnera :



Si on modifie l'input avec « ananas » on obtient :



Si enfin on modifie l'input avec «kiwi» on obtient :



3.6) Filtres (pipes)

Angular permet de transformer l'affichage d'une variable grâce à une syntaxe particulière dans le code du property binding. Si par exemple on veut que la variable `myName` soit affichée en lettres capitales on peut utiliser la syntaxe suivante :

```
{{myVariable | uppercase}}
```

On constate l'usage de l'opérateur pipe « | », c'est pour cela qu'on parle de pipe Angular, plutôt que de filtre angular. Il existe deux sortes de pipes, les pipes natifs, ou built-in pipes, c'est le cas de `uppercase`, et les pipes personnalisés, ou custom pipes.

Il est possible d'enchaîner les pipes avec la syntaxe :

```
{{myVariable | pipe1 | pipe2 | pipe 3 ...}}
```

Enfin, les pipes peuvent être paramétrés, avec la syntaxe suivante :

```
{{myVariable :'param1' :'param2' ...}}
```

Notez l'utilisation des quotes ' dans l'usage des paramètres

3.6.1) Built-in pipes

La liste des pipes natifs est donnée dans la documentation : https://angular.io/api?type=pipe. Nous allons en passer quelques uns en revue.

Attention à l'heure où nous écrivons ces lignes, le paramètre `locale` n'est pas encore disponible en dehors de 'en-US'

date

```
date_expression | date[:format[:timezone[:locale]]]
```

Exemple :
En posant `myDate = new Date(2018,0,15,6)`, 15 janvier 2018 à 6h,
le code `{{myDate | date : 'short' :'+0200'}}` affichera « **1/15/18, 1:00 AM** »,
le code `{{myDate | date : 'dd/MM/y'}}` affichera « **15/01/2018** »,
le code `{{myDate | date : 'dd-MM-y hh:ii:ss'}}` affichera « **2018-01-15 06:00:00** »,

currency

```
number_expression | currency[:currencyCode[:display[:digitInfo[:locale]]]]
```

Exemple :
En posant `prix = 125.75`,
le code `{{prix | currency : 'USD' :'symbol'}}` affichera « $**125.75** »,
le code `{{prix | currency : 'EUR' :'symbol'}}` affichera « **€125.75** »,
le code `{{prix | currency : 'EUR' :'symbol' :'3.1-1'}}` affichera « **€125.8** »,

number

```
number_expression | number[:digitInfo[:locale]]
```

Exemple :
En posant `nombre = 58.0389`,
le code `{{nombre | number}}` affichera « **58.0389** »,
le code `{{nombre | number :'3.1-1'}}` affichera « **058.0** »,
le code `{{nombre | number :'3.1-3'}}` affichera « **058.039** »,

percent

```
number_expression | percent[:digitInfo[:locale]]
```

Exemple :
En posant `nombre = 0.58`,
le code `{{nombre | percent}}` affichera « **58 %** »,

slice

```
array_or_string_expression | slice:start[:end]
```

Exemple
En posant `myString = 'Bonjour les amis'`,
le code `{{myString | slice :'0' :'7'}}` affichera « **Bonjour** »,
le code `{{myString | slice :'8' :'11'}}` affichera « **les** »,

3.6.2) Custom pipes

Il est possible de produire des pipes personnalisés. A titre d'exemple nous allons créer un pipe qui transforme une longitude/latitude depuis un nombre décimal, vers une expression en degrés, minutes et secondes d'angles.

La première chose à faire est de créer un répertoire pipes dans le répertoire lib de notre application (mais vous pourriez faire un tout autre choix), de s'y placer et de demander à Angular de générer un pipe :

```
# mkdir lib/pipes
# cd lib/pipes
# ng generate pipe lonlat
```

Angular crée alors deux fichiers : `lonlat.pipe.ts`, qui est le pipe en en lui même, et `lonlat.pipe.spec.ts`, qui est le fichier de tests unitaires du pipe. Voici le code du pipe :

lonlat.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'lonlat'
})
export class LonlatPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    return null;
  }

}
```

Simultanément Angular ajoute un import dans le module de l'application AppModule, et inscrit le pipe dans ses déclarations :

app.module.ts

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

//modules personnalisés
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';

//composants
import { AppComponent } from './app.component';

import { LonlatPipe } from './lib/pipes/lonlat.pipe';

@NgModule({
  declarations: [
    AppComponent,
    LonlatPipe
  ],
  imports: [
    BrowserModule,
    FormsModule,
    SiteHeaderModule
  ],
  providers: [
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Notez que nous laisserons ce pipe dans le module principal de l'application car a priori il sera utile de façon générale dans toute l'application.

Modifions maintenant notre pipe de la façon suivante :

lonlat.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'lonlat'
})
```

```
export class LonlatPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    let deg = Math.floor(value);
    let min = Math.floor((value-deg)*60);
    let sec = Math.floor(((value-deg)*60-min)*60);
    return deg+'° '+min+'\' '+sec+'"';
  }
}
```

En posant `nombre=48.34895`, le code `{{nombre | lonlat}}` affichera « **48° 20' 56"** »

3.7) Composants imbriqués (Nested components)

Un composant peut inclure un autre composant, pour cela il suffit de déclarer le composant enfant dans les imports du module qui contient ce composant, puis d'inclure son selector dans le HTML du composant parent.

3.7.1) Création et intégration

Nous avons précédemment créé le composant SiteHeaderComponent. Nous allons donc y inclure une série de liens, qui nous servirons plus tard pour évoquer le routing. Pour ce faire nous nous plaçons à la console dans le répertoire des composants et nous générons le composant links-list :

```
# cd lib/components
# ng generate component links-list
```

Nous modifions ensuite le composant LinksListComponent comme suit :

links-list.component.ts

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

@Component({
  selector: 'app-links-list',
  templateUrl: './links-list.component.html',
  styleUrls: ['./links-list.component.css']
})
export class LinksListComponent implements OnInit {

  links = [
      { title : 'Page 1', url : 'page1'},
      { title : 'Page 2', url : 'page2'},
      { title : 'Admin', url : 'admin'},
    ];

  constructor() { }

  ngOnInit() {
  }

}
```

et le template de ce composant comme suit :

links-list.component.html

```
<div class="collapse navbar-collapse">
  <ul class="navbar-nav mr-auto ml-auto">
    <li class="nav-item" *ngFor="let ln of links">
      <a class="nav-link" [href]="ln.url">{{ln.title}}</a>
    </li>
  </ul>
</div>
```

A noter que nous utilisons :
- *la structural directive* `*ngFor`
- *l'attribute binding avec* `[href]`
- *le property binding avec* `{{ln.title}}`

Bien sûr il nous faut importer ce nouveau composant dans le module parent qui le contiendra, à savoir SiteHeaderModule :

site-header.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { SiteHeaderComponent } from
    '../../components/site-header/site-header.component';
import { LinksListComponent } from
    '../../components/links-list/links-list.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    SiteHeaderComponent,
    LinksListComponent
  ],
  exports : [
    SiteHeaderComponent,
    LinksListComponent
  ]
})
export class SiteHeaderModule { }
```

A noter que nous en profitons pour exporter le composant à partir de ce module, au cas où il serait utile pour un autre module.

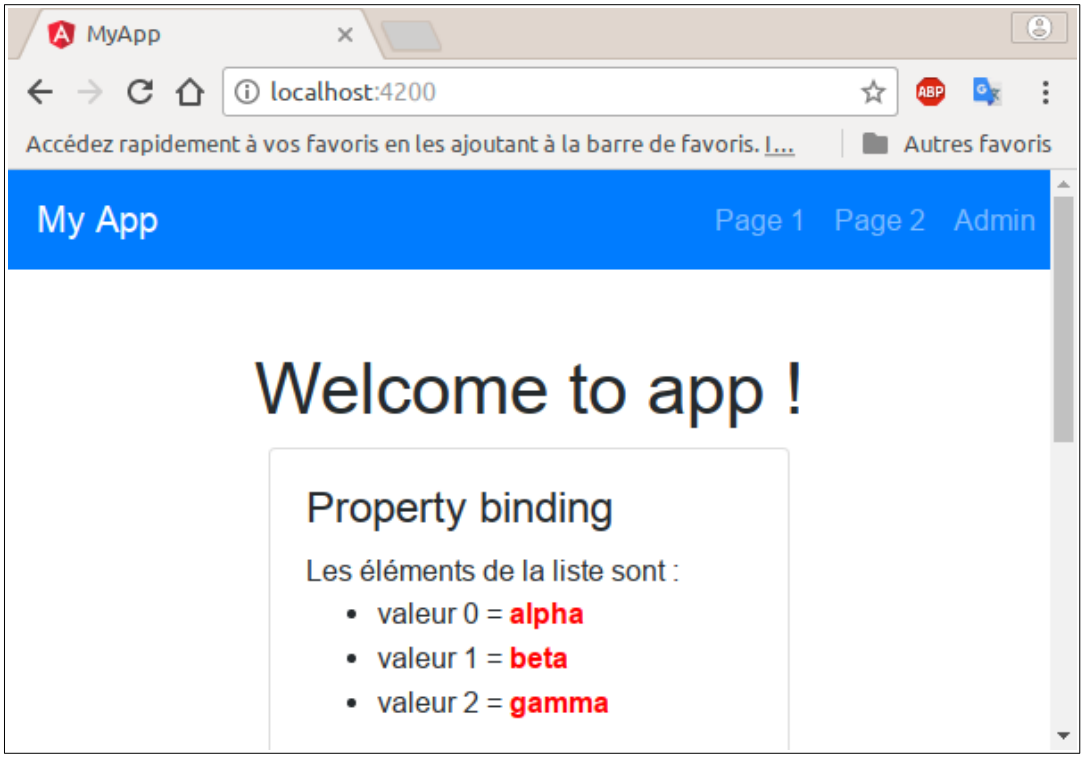
Il ne nous reste plus alors qu'à intégrer le decorator du composant enfant au template HTML du composant parent `SiteHeaserComponent` :

site-header.component.html

```
<nav class="navbar navbar-expand-sm navbar-dark bg-primary fixed-top">
  <a class="navbar-brand" href="#">My App</a>
  <app-links-list></app-links-list>
</nav>
```

A noter que le composant enfant ayant déjà été importé dans le module contenant les deux composants, parent et enfant, le composant parent n'a pas besoin d'importer le composant enfant. L'import ne se fait donc que dans le module.

L'affichage montre alors que notre liste de liens a bien été incluse dans le composant `SiteHeaderComponent` :



On peut cliquer sur les liens et vérifier dans la barre d'adresse qu'on est bien redirigé où il faut (`http://localhost:4200/page1`, `http://localhost:4200/page2`, et `http://localhost:4200/admin`*), cependant nous n'avons encore défini aucune règle de routing, car nous verrons cela plus loin, et par suite nous sommes toujours redirigé vers la même page d'accueil.*

A noter qu'une anomalie du fonctionnement de Bootstrap avec Angular empêche la liste de liens d'être alignée à droite. Il nous faut donc apporter une modification au CSS du composant links-list en ajoutant au fichier **`site-header.component.css`** *le code suivant :*

```
app-links-list {
  position:absolute;
  right:0;
}
```

3.7.2) Échange de données du parent vers l'enfant : @Input

Au lieu de définir la liste de liens dans le composant enfant LinksListComponent, nous allons le faire dans le composant parent `SiteHeaderComponent`, puis la communiquer à l'enfant `LinksListComponent`. Pour cela nous modifions d'abord le composant parent :

site-header.components.ts

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

@Component({
  selector: 'app-site-header',
  templateUrl: './site-header.component.html',
  styleUrls: ['./site-header.component.css']
})
export class SiteHeaderComponent implements OnInit {

  links = [
      { title : 'Page 1', url : 'page1'},
      { title : 'Page 2', url : 'page2'},
      { title : 'Admin', url : 'admin'},
    ];

  constructor() { }

  ngOnInit() { }

}
```

Puis nous modifions le composant enfant pour le préparer à recevoir des données, en lui indiquant le nom de la variable qui lui sera communiqué par le composant parent. Cela se fait grâce au décorateur @Input :

links-list.components.ts

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

@Component({
  selector: 'app-links-list',
  templateUrl: './links-list.component.html',
  styleUrls: ['./links-list.component.css']
})
export class LinksListComponent implements OnInit {

  @Input() links : any;
  constructor() {}
  ngOnInit() {}
}
```

A noter que pour utiliser le décorateur `@Input`, il faut importer le module Input depuis le cœur d'Angular.

Enfin il faut modifier le template HTML du parent en lui spécifiant la variable qu'il doit exporter vers le composant enfant :

site-header.component.html

```
<nav class="navbar navbar-expand-sm navbar-dark bg-primary fixed-top">
  <a class="navbar-brand" href="#">My App</a>
  <app-links-list [links]="links"></app-links-list>
</nav>
```

A noter qu'on utilise ici l'attribute binding avec `[links]`, ainsi toute modification de links dans le parent sera prise en compte dans l'enfant.

Important :
- *il est ici possible d'utiliser le* ***two way data binding*** *en remplaçant* `[links]` *par* `[(links)]`. *Dans ce cas toute modification de* `links` *dans l'enfant remontera vers le parent, et vice-versa.*
- à cause du délai de chargement une variable peut ne pas être disponible dans le template lors de son premier affichage. Il est alors utile de se servir du « Elvis operator » :

```
<span>{{maVariable?.myProperty}}</span>
```

Bien évidemment, l'affichage de ce code donnera la même chose que la figure précédente.

3.7.3) Echange de données de l'enfant vers le parent : #myChild

Il est possible de récupérer des variables du composant enfant pour s'en servir dans le template HTML du composant parent. Replaçons d'abord la liste de liens dans le composant enfant, et supprimons la du composant parent :

links-list.component.ts

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

@Component({
  selector: 'app-links-list',
  templateUrl: './links-list.component.html',
  styleUrls: ['./links-list.component.css']
})
export class LinksListComponent implements OnInit {

  links = [
      { title : 'Page 1', url : 'page1'},
      { title : 'Page 2', url : 'page2'},
      { title : 'Admin', url : 'admin'},
    ];

  constructor() {

  }

  ngOnInit() {
  }

}
```

site-header.component.ts

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

@Component({
  selector: 'app-site-header',
  templateUrl: './site-header.component.html',
  styleUrls: ['./site-header.component.css']
})
export class SiteHeaderComponent implements OnInit {
```

```
  constructor() { }

  ngOnInit() { }

}
```

A noter que le composant parent, bien qu'attendant à recevoir des données, n'utilise pas le module `Input` *d'Angular, et qu'il n'a donc pas besoin de l'importer.*

Ensuite tout est fait dans le template HTML du composant parent, qu'on modifie de la façon suivante :

site-header.component.html

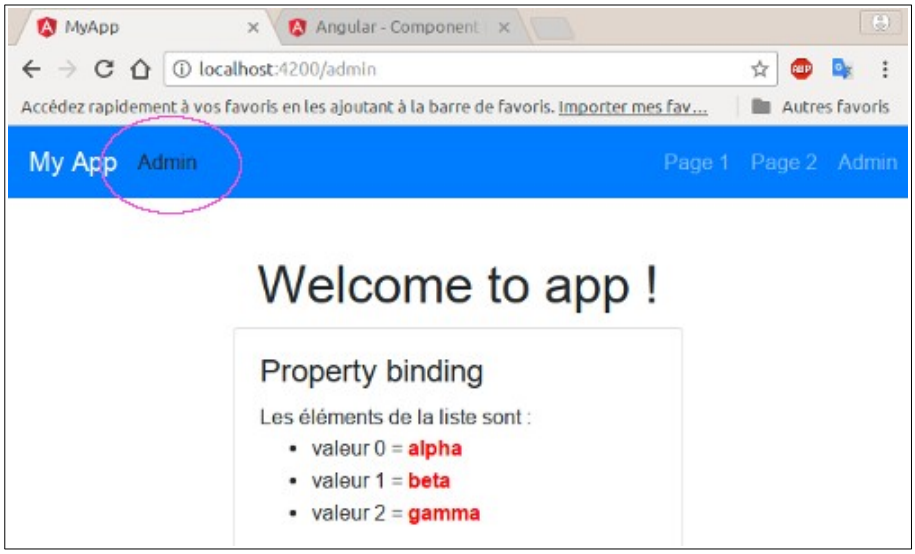
```
<nav class="navbar navbar-expand-sm navbar-dark bg-primary fixed-top">
  <a class="navbar-brand" href="#">My App</a>
  <app-links-list #fromchild></app-links-list>
  <span>{{fromchild.links[2].title}}</span>
</nav>
```

A noter la syntaxe #

On indique ainsi au parent (`SiteHeaderComponent`) que l'enfant (`LinksListComponent`) lui expédiera des données qu'il pourra utiliser grâce à la variable `fromchild` (le nom importe peu, vous pouvez faire un tout autre choix), **qui symbolisera la classe du composant enfant**. Ensuite il peut utiliser cette variable dans son template HTML **exactement comme si cette variable avait été définie dans sa propre classe**. A titre d'exemple nous avons demandé dans ce code d'afficher le titre du 3ème lien avec `{{fromchild.links[2].title}}`.

Important : toutes les propriétés, ***mais aussi les méthodes*** *du composant enfant peuvent ainsi être remontées vers le parent*

Ce code fera apparaître « Admin » à côté de « My App » dans le header du site :



3.7.4) Echange d'événement de l'enfant vers le parent : @Output

Lorsqu'un événement (click, focus, …) intervient dans le composant enfant il peut être intéressant de le faire savoir au composant parent. C'est toute l'utilité du module **`Output`** d'Angular. Voyons comment faire cela en ajoutant un bouton de login à notre composant enfant, et en récupérant ce qu'il émet dans le composant parent.

Commençons par placer le bouton Bootstrap de login dans le template HTML du composant enfant `LinksListComponent` :

links-list.component.html

```
<div class="collapse navbar-collapse">
  <ul class="navbar-nav mr-auto ml-auto">
    <li class="nav-item" *ngFor="let l of links; let idx = index">
      <a class="nav-link" [href]="l.url">{{l.title}}</a>
    </li>
  </ul>
  <button
    class="btn btn-secondary"
    (click)="emitLoginClick()"
    style="margin-right:10px">
      Login
  </button>
</div>
```

A noter que nous avons ajouté le style `margin-right:10px` *au bouton pour qu'il ne soit pas collé au bord droit de la fenêtre d'affichage.*

Ce code indique que lorsque le bouton est cliqué, il exécute la fonction `emitLoginClick()`, que nous allons devoir déclarer dans la classe du composant `LinksListComponent` .

Outre la déclaration de la fonction `emitLoginClick()`, le composant enfant (`LinksListComponent`) doit aussi déclarer qu'il va émettre un événement, grâce au module `Output` d'Angular. Ce dernier, que nous nommerons `loginClick`, ne peut être que de type `EventEmitter`, qui est un autre module d'Angular. Par suite nous modifions la classe `LinksListComponent` ainsi :

links-list.component.ts

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

@Component({
  selector: 'app-links-list',
  templateUrl: './links-list.component.html',
  styleUrls: ['./links-list.component.css']
})
export class LinksListComponent implements OnInit {

  @Input() links : any;
  @Output() loginClick = new EventEmitter<string>();


  constructor() {

  }

  ngOnInit() {
  }
```

```
  emitLoginClick() {
    this.loginClick.emit('---> Le bouton login a été cliqué !')
  }

}
```

A noter que pour utiliser les modules `Output` *et* `EventEmitter`*, nous devons d'abord les importer depuis le cœur d'Angular.*

On remarque dans ce code que l'`EventEmitter`, qui s'appelle `loginClick`, émet une chaîne de caractère grâce à la méthode `emit()`. Il serait néanmoins possible d'émettre plutôt un objet, et dans ce cas le code serait par exemple le suivant :

links-list.component.ts

```
...

export class LinksListComponent implements OnInit {

  @Input() links : any;
  @Output() loginClick = new EventEmitter<any>();

  ...

  emitLoginClick() {
    this.loginClick.emit({
       id : 456,
       message : '---> Le bouton login a été cliqué !'
    })
  }

}
```

A noter que l'`EventEmitter` *est alors déclaré comme renvoyant le type* `any`*, et plus* `string`*.*

Pour faire simple, nous resterons cependant ici sur l'émission d'une chaîne de caractères plutôt qu'un objet.

Nous avons fait tout ce qu'il faut du côté composant enfant, passons donc au composant parent `SiteHeaderComponent`. Commençons par modifier son template HTML pour lui indiquer qu'il va recevoir l'`EventEmitter` nommé `loginClick` depuis le composant enfant, et qu'il devra exécuter une fonction, nommons la `onLoginClick()`, qui prendra en paramètre l'événement renvoyé, à savoir notre chaîne de caractère :

site-header.component.html

```
<nav class="navbar navbar-expand-sm navbar-dark bg-primary fixed-top">
  <a class="navbar-brand" href="#">My App</a>
  <app-links-list [links]="links" (loginClick)="onLoginClick($event)"></app-links-list>
</nav>
```

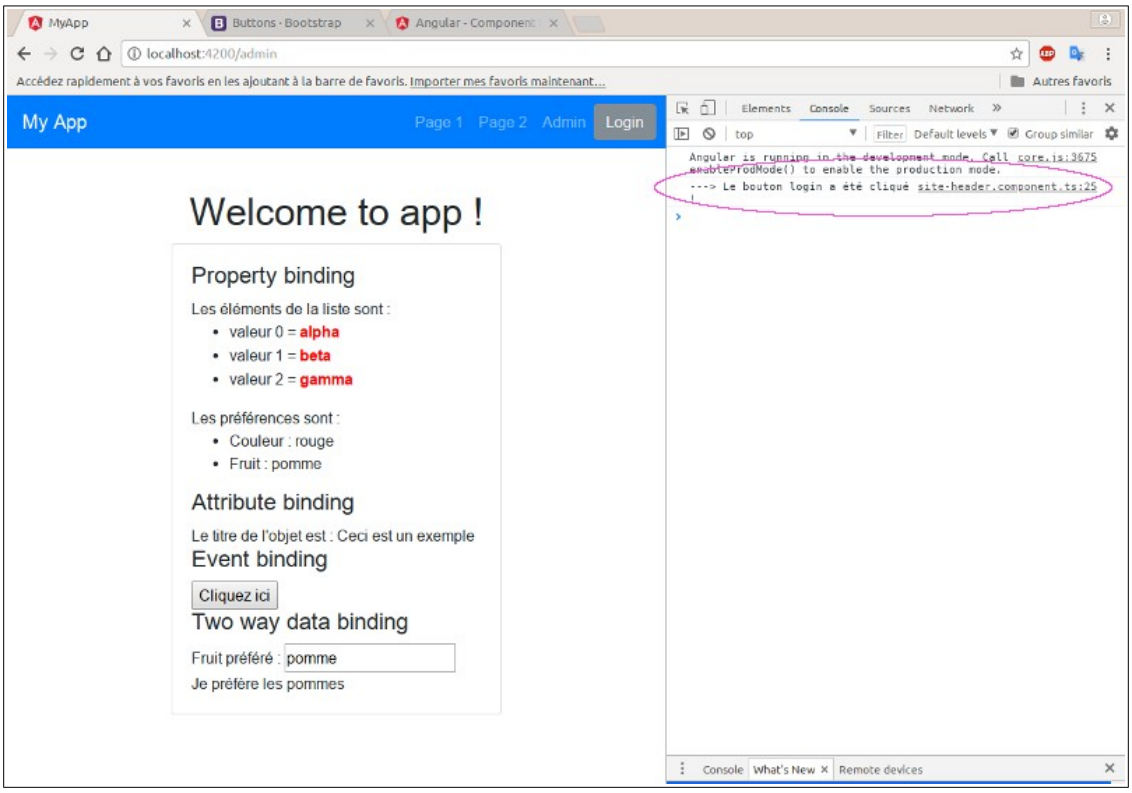
Il ne nous reste plus alors qu'à déclarer la fonction `onLoginClick()` dans la classe du composant parent :

site-header.component.ts

```
...

export class SiteHeaderComponent implements OnInit {

  ...

  onLoginClick(message) {
    console.log(message);
  }

}
```

A noter que le composant parent n'a rien de nouveau à importer

La figure suivante donne l'affichage obtenu après avoir cliqué sur le bouton de login :



On voit que le message expédié par le composant enfant a bien été reçu et affiché par le composant parent.

3.8) Services

Une des notions importantes d'Angular est le service, qui a vocation à servir des propriétés et des méthodes dans toute l'application, mais aussi dans différentes applications si on a pris soin de le situer dans une bibliothèque. Une des fonction très utiles d'un service est de récupérer des données sur un serveur pour les mettre à disposition de l'application. Nous allons donc voir deux types de service, l'un étant classique, et l'autre dédié à la récupération de données par http.

3.8.1) Service simple

Les services ayant vocation à être horizontaux, il est une bonne pratique de les situer dans la bibliothèque générale, par exemple dans un répertoires « service ». Pour générer un service nous utiliseront une nouvelle fois les capacités d'Angular. Voici comment faire pour créer un service `UserService` :

```
# cd app/lib
# mkdir services
# cd services
# ng generate service user
```

Angular gébère alors deux fichiers, l'un de test `user.service.spec.ts`, et l'autre contenant le service à proprement parler : `user.service.ts` :

user.service.ts

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

@Injectable()
export class UserService {

  constructor() { }

}
```

Notez que le service est de type **`@Injectable`**, *et qu'il faut donc importer le module* `Injectable` *depuis le cœur d'Angular.*

Définissons une propriété « `data` » et une méthode « `transform()` » pour ce service :

user.service.ts

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

@Injectable()
export class UserService {

  constructor() { }

  data = {
    table : ['a','b','c'],
    text : 'Ceci est du texte provenant de UserService',
    obj : {
      prop1 : 123,
      prop2 : 'abc'
    }
  }

  transform(input) {
    return 'Input was : '+input;
  }

}
```

Le service peut alors être injecté dans un module ou un composant d'une façon un peu particulière. Nous allons l'injecter dans notre composant principal `AppComponent` de la façon suivante :

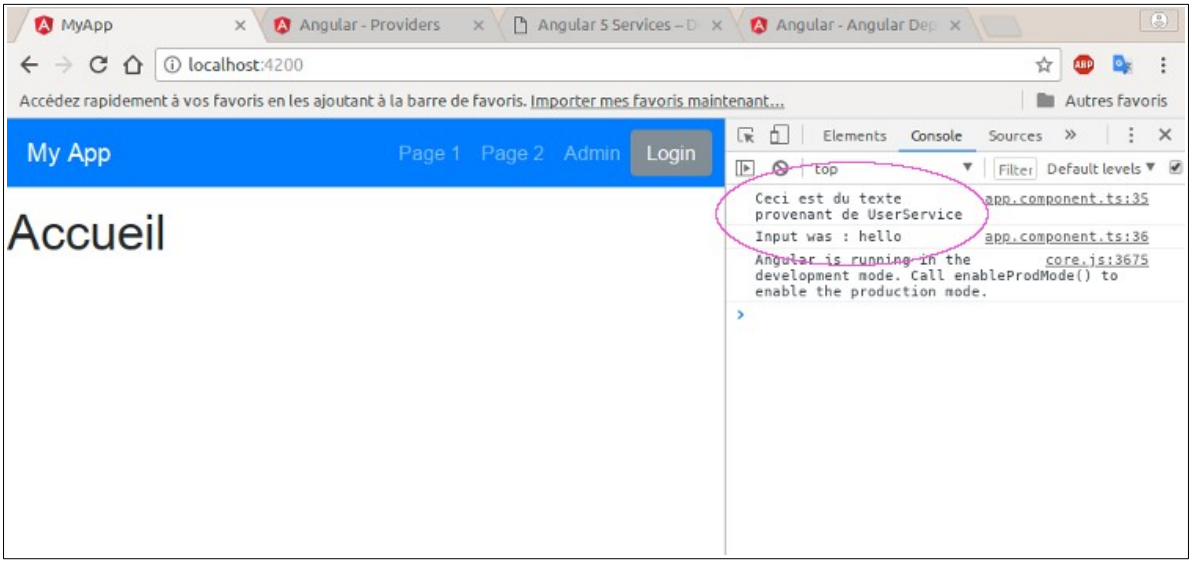
app.component.ts

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

import { UserService } from './lib/services/user.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers : [UserService]
})
export class AppComponent {

  constructor(
    private myService: UserService
  ) { }

  ngOnInit() {

    console.log(this.myService.data.text)
    console.log(this.myService.transform('hello'))

  }

}
```

A noter que le service importé doit être ajouté à la liste des **`providers`** *du composants.*

Attention, étant un injectable, le service doit être entré parmi les paramètres du `constructor`*. C'est une spécificité des injectables.*

Au rechargement de l'application l'affichage donnera :



3.8.2) Partage des données d'un service

Il est souvent pratique de partager les données d'un service entre plusieurs composants. Par exemple un profil

utilisateur, modifié par un composant, devrait se retrouver dans son état modifié dans un autre composant. Voyons comment implémenter cela.

Nous allons créer un service « counter », qui propose un compteur qui sera incrémenté à chaque fois qu'une page est appelée.

Créons d'abord le service :

```
# cd lib/services
# ng generate service counter
```

Puis ajoutons à ce service une méthode qui incrémente un compteur :

counter.service.ts

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

@Injectable()
export class CounterService {

  counter:number = 0;

  constructor() { }

  increment() {
    this.counter ++;
  }

}
```

On souhaite que ce service soit disponible dans tous les composants de l'application, il faut donc d'abord l'injecter dans le module principal de l'application AppModule :

app.module.ts

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

//modules personnalisés
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';
import { AppRoutingModule } from './app-routing.module';

//composants
import { AppComponent } from './app.component';

//pipes
import { LonlatPipe } from './lib/pipes/lonlat.pipe';

//services
import { CounterService } from './lib/services/counter.service';


@NgModule({
  declarations: [
    AppComponent,
    LonlatPipe
  ],
  imports: [
```

```
    BrowserModule,
    FormsModule,
    AppRoutingModule,
    SiteHeaderModule
  ],
  providers: [
    CounterService
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

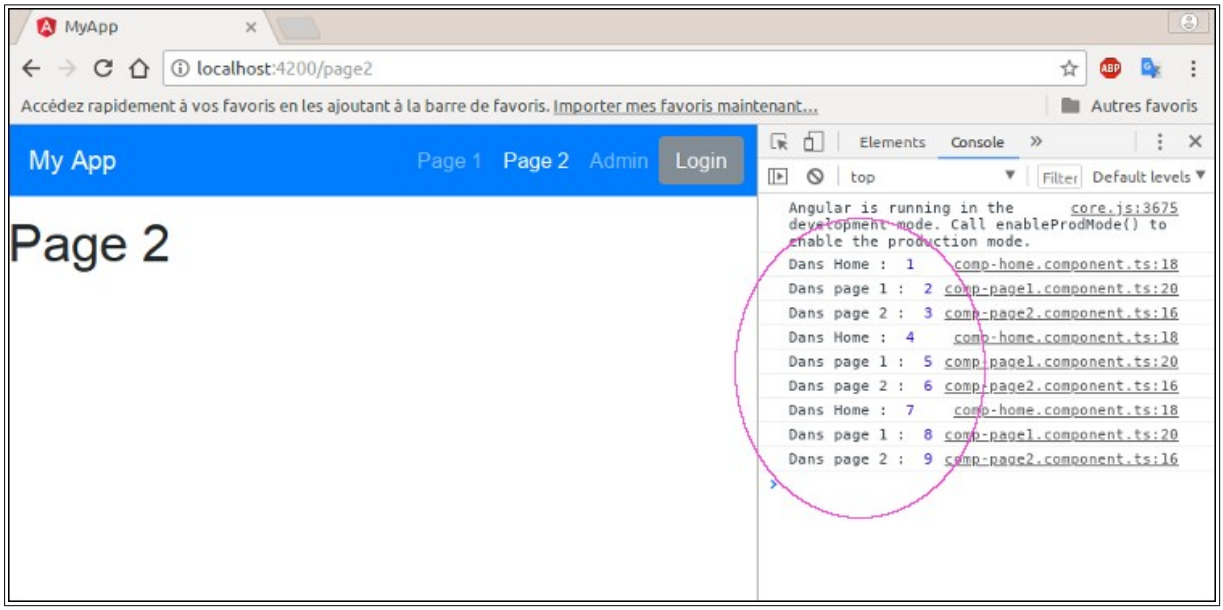
Nous allons maintenant appeler ce service dans les composants des Page 1, Page 2 et Home. Voici comment écrire le composant de la page 1 :

comp-page1.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute} from '@angular/router';

import { CounterService } from '../../../lib/services/counter.service';

@Component({
  selector: 'app-comp-page1',
  templateUrl: './comp-page1.component.html',
  styleUrls: ['./comp-page1.component.css'],
})
export class CompPage1Component implements OnInit {

  constructor(
    private counterService:CounterService
  ) { }

  ngOnInit() {

    this.counterService.increment();
    console.log('Dans page 1 : ',this.counterService.counter);

  }

}
```

Il faut ensuite pratiquer de la même façon dans Page 2 et Home.

Au rechargement de la page, on clique sur Page 1, Page 2 et Home successivement, à plusieurs reprises et on observe que le compteur s'incrémente, sans jamais se remettre à 0 :



Si le compteur ne se remet pas à 0 à chaque changement de page c'est parce que il n'a été injecté que dans un seul provider, celui de `AppModule` *, qui est le général à toute l'application :*

app.module.ts

```
...
  providers: [
    CounterService
  ],
...
```

Il n'existe alors qu'une seule instance du service pour toute l'application. Si au contraire nous avions injecté le service dans chaque composant des pages, chacune aurait eu un compteur distinct.

3.8.3) La bonne pratique du service store global

Il est de bonne pratique de définir un service **`store.service.ts`** commun à toute l'application, et donc monté dans les providers du **`AppModule`**. Il servira à résoudre les asynchronismes (voir le chapitre sur HTTP) mais aussi à partager des données et objets entre tous les éléments, composants ou services, qui importeront ce service global **`store`**. Ce service devient la colonne vertébrale de votre application, le référentiel commun auquels tous les autres éléments se réfèreront pour se synchroniser.

Le service store peut être utilisé directement dans les templates HTML, sans avoir besoin de déclarer aucune variable locale au composant. Par exemple pour modifier instantanément le nom de l'utilisateur dans toute l'application, on pourra se servir du tag input suivant :

```
<input type="text" [(ngModel)]="store.user && store.user.name" />
```

Le composant qui inclut ce tag remettra à jour instantanément toute l'application sans avoir à déclarer aucun **`@Input`**, **`@Output`**, sans avoir à déclarer la moindre variable locale.

Notez la syntaxe ***`[(ngModel)]="store.user && store.user.name"`*** *qui permet à* ***`ngModel`*** *de gérer l'asynchronisme lié à la propriété* ***`user`*** *de* ***`store`***.

En outre, si l'état complet de votre application est décrit pas votre service store, à la propriété configuration par exemple, il suffit de sauvegarde store.configuration à un instant t, dans le localStorage, l'IndexedDb ou sur un serveur. Si vous avez besoin de revenir en arrière en retrouvant l'état exact de l'application telle qu'il était à l'instant t, rien de plus simple, avec un code ultra court du type :

```
this.store = this.storeBackup(t);
```

La boucle infinie d'Angular se charge alors de remettre à jour l'application entière. Ceci est particulièrement adapté aux applications qui ont besoin d'une fonctionnalité back/forward de leur état, comme par exemple un log formulaire de plusieurs pages.

3.9) Routing

Angular est un framework de « single page application », c'est à dire que toute application Angular est contenue dans une seule page, `index.html`, et plus particulièrement à l'intérieur du tag `<app-root></app-root>`. Pourtant il faut donner l'impression à l'utilisateur qu'il se trouve sur une application composée de pages différentes, et a priori indépendantes. C'est l'objet du routing.

Chaque module peut posséder son propre routing mais Angular en distingue cependant deux sortes : le routing « for root », ou routing principal de l'application, qui concerne le module principal `AppModule`, et les routings secondaires « for child » concernant les autres modules internes.

La multiplication des pages peut vite rendre l'arborescence des répertoires et fichiers d'une application complexe et illisible, c'est pourquoi il est nécessaire de s'organiser. En la matière Angular est très permissif, et vous pouvez choisir l'organisation qui vous plaira, nous allons cependant en proposer une qui nous semble être une bonne pratique.

3.9.1) Organisation des pages

Chaque page « virtuelle », vers laquelle le routing redirige, doit être soutenue par un module, et son code HTML n'est autre que le template d'un composant contenu dans ce module. Dès lors, puisque nous avons constitué une bibliothèque contenant nos modules et composants, il peut paraître logique d'y situer aussi les pages du site. Il faut cependant remarquer que chaque site possède des pages qui lui sont propres, et rares seront les cas où des pages seront réutilisables dans des applications différentes. Ainsi nous voyons qu'il est préférable de séparer nos éléments réutilisables (répertoire lib), de nos éléments uniques concernant les pages. Nous allons donc créer un nouveau répertoire pages dans lequel nous placerons composants et modules de nos pages :

```
# mkdir src/app/pages
```

Puisque nous avons déjà créé un module header pour notre site, contenant une série de liens « Page 1, Page 2 et Admin, nous allons implémenter cette structure pour nos pages. Voici comment nous allons procéder : on se place dans le répertoire des pages, et pour chacune d'entre elles on crée un module, c'est à dire un répertoire contenant un fichier `xxxxxx.module.ts`. On se place ensuite dans ce répertoire où on crée le composant de la page :

```
# cd src/app/pages
# ng generate module page1
# cd page1
# ng generate component comp-page1
```

Au total l'arborescence que nous venons de créer est la suivante :

```
...
    ├── pages
    │   ├── page1
    │   │   ├── page1.module.ts
    │   │   ├── comp-page1
    │   │   │   ├── comp-page1.component.css
    │   │   │   ├── comp-page1.component.html
    │   │   │   ├── comp-page1.component.spec.ts
    │   │   │   ├── comp-page1.component.ts
```

Cette structure est intéressante car elle permettra de regrouper dans le répertoire de la page concernée tous les éléments uniquement dédiés à cette page (modules, composants, pipes, directives, …).

Pour créer la page « Page 2 » et la page « Admin », il suffit de procéder de la même façon.

Pour ce qui concerne la page d'accueil, pour l'instant elle est contenue dans le template de notre module principal AppComponent. Il va nous falloir changer cela et créer une page supplémentaire, que nous nommerons Home, de la même façon que nous avons créé les autres pages, et qui contiendra dans le template de son composant le HTML qui devra apparaître lorsque l'utilisateur se trouve sur la page d'accueil.

Ainsi nous allons définir les templates de nos pages de la façon suivante :

comp-xxxxxx.component.ts

```
<div class="topSiteMargin"></div>
<h1>xxxxxx</h1>
```

Dans ce code `xxxxxx` sera remplacé par page1, page2, admin ou home.

Notez que nous avons créé une classe générale `topSiteMargin` qui sert à décaler le contenu des templates vers le bas, pour qu'il ne soit pas recouvert par le header de l'application. Comme cet élément nous sera nécessaire pour toutes les pages du site, nous le plaçons dans le fichier de style général `src/styles.css` qui devient :

src/styles.css

```
@import '../node_modules/bootstrap/dist/css/bootstrap.min.css';

.topSiteMargin {
  margin-top : 70px;
}
```

3.9.2) Routage principal

3.9.2.1) AppRoutingModule

La première chose à faire est de créer un module dédié au routing. Nous pourrions procéder autrement en écrivant le code qu'il contiendra directement dans le module principal de l'application AppModule, mais une fois encore cela nuirait à la lisibilité. La bonne pratique est donc de créer un module AppRoutingModule, puis de l'importer dans AppModule.

Voici le code de notre module de routing :

app/app-routing.module.ts

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

import { CompHomeComponent } from './pages/home/comp-home/comp-home.component';
import { CompPage1Component } from './pages/page1/comp-page1/comp-page1.component';
import { CompPage2Component } from './pages/page2/comp-page2/comp-page2.component';
import { CompAdminComponent } from './pages/admin/comp-admin/comp-admin.component';

const routes: Routes = [
    { path : ''     , component : CompHomeComponent },
    { path : 'page1', component : CompPage1Component },
    { path : 'page2', component : CompPage2Component },
    { path : 'admin', component : CompAdminComponent },
    { path : '**'   , redirectTo : '', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  declarations : [
    CompHomeComponent,
    CompPage1Component,
    CompPage2Component,
    CompAdminComponent
  ],
  providers: []
})

export class AppRoutingModule { }
```

Plusieurs choses sont à noter dans ce code :

- Les urls sont définies relativement à l'URL racine du site (`localhost:4200/` dans notre cas).
- Il est nécessaire d'importer les modules `RouterModule` et `Routes` depuis `@angular/router`
- Les routes sont déclarées dans un objet de type `Routes`
- Le Router module est importé de façon particulière : `RouterModule.forRoot(routes)`, et nous verrons plus loin que cette syntaxe est réservée au routing principal, les routeurs secondaires, dédiés à des modules inclus au modules principal, utiliseront en revanche la syntaxe : `RouterModule.forChild(routes)`
- Nous exportons le module `RouterModule` pour qu'il soit disponible dans le module principal
- Le path `'**'` représente n'importe quelle URL qui ne soit pas décrite dans le module de routage
- la propriété `redirectTo` permet de rediriger vers une URL particulière, déjà existante parmi les routes définies. Dans ce cas la propriété `pathMatch` doit être présente, et nous renvoyons à la documentation pour plus de précisions sur ce paramètre.

La définition des routes est relativement logique, elle utilise une propriété `path` qui définit l'URL de la page concernée, et lui fait correspondre le composant qu'il faudra afficher lorsque cette URL sera réclamée.

3.9.2.2) AppModule

Bien sûr il nous faut maintenant importer ce module de routing dans le module principal. Nous le faisons d'une façon très classique :

app.module.ts

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

//modules personnalisés
import { SiteHeaderModule } from './lib/modules/site-header/site-header.module';
import { AppRoutingModule } from './app-routing.module';

//composants
import { AppComponent } from './app.component';

//pipes
import { LonlatPipe } from './lib/pipes/lonlat.pipe';



@NgModule({
  declarations: [
    AppComponent,
    LonlatPipe
  ],
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule,
    SiteHeaderModule
  ],
  providers: [
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3.9.2.3) router-outlet

Il nous reste maintenant à définir où les différentes pages doivent s'afficher dans le template HTML du module principal. Cela se fait grâce à un tag **réservé** : **`<router-outlet></router-outlet>`**, de la façon suivante :

app.component.html

```
<app-site-header></app-site-header>
<router-outlet></router-outlet>
```

Tout ce que le routeur affiche le sera à l'intérieur de ce tag.

A noter que nous laissons le tag du header du site en dehors de `router-outlet`*, car nous voulons que*

le header reste le même quelle que soit la page choisie.

Tout est désormais en place et fonctionnel. En cliquant sur les liens du header du site, décrits par le composant `LinksListComponent`, on est redirigé vers la page correspondante.

3.9.3) Eléments de Routage

3.9.3.1) routerLink et routerLinkActive

Le routage se fait généralement par des liens cliquables, comme nous l'avons fait dans notre `SiteHeaderComponent` faisant appel à `LinksListComponent`. Il est alors intéressant de modifier la classe du lien qui est en cours d'affichage. Angular permet cela grâce aux attributs `routerLink` et `routerLinkActive`.

Pour les mettre en œuvre il nous faut d'abord importer `RouterModule` dans le module contenant les liens de routage, c'est à dire dans notre cas `SiteHeaderModule` :

site-header.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { RouterModule } from '@angular/router';

import { SiteHeaderComponent } from
    '../../components/site-header/site-header.component';
import { LinksListComponent } from '../../components/links-list/links-list.component';

@NgModule({
  imports: [
    CommonModule,
    RouterModule,
    FormsModule
  ],
  declarations: [
    SiteHeaderComponent,
    LinksListComponent
  ],
  exports : [
    SiteHeaderComponent,
    LinksListComponent
  ]
})
export class SiteHeaderModule { }
```

Ensuite on modifie la syntaxe du template HTML du composant `LinksListComponent` contenu dans notre module `SiteHeaderModule`, de la façon suivante :

links-list-component.html

```
<div class="collapse navbar-collapse">
  <ul class="navbar-nav mr-auto ml-auto">
    <li class="nav-item" *ngFor="let l of links; let idx = index">
      <a class="nav-link"
         [routerLink]="l.url"
         routerLinkActive="active">
```

```
            {{l.title}}
        </a>
      </li>
  </ul>
  <button   class="btn btn-secondary"
            (click)="emitLoginClick()"
            style="margin-right:10px">Login</button>
</div>
```

Ce code indique que lorsque la route sélectionnée est active, il faut appliquer la classe CSS « `active` » au lien de routage correspondant. Cette classe doit exister par ailleurs, ce qui est le cas de Bootstrap qui nous la fournit.

Attention, remarquez :
- *que nous avons modifié* `[href]` *en* **`[routerLink]`**, *sans quoi cette fonctionnalité ne serait pas active.*
- *l'utilisation du camel case des attributs* `routerLink` *et* `routerLinkActive`, *alors que les attributs HTML ne sont pas case sensitive.*
- *lorsque le lien est dynamique on utilise* **`[routerLink]`**, *mqis on utilise simplement* **`routerLink`** *lorsqu'il est statique*

3.9.3.2) URL paramétrées

Angular permet d'utiliser des routes paramétrées. On peut par exemple paramétrer la page 1 de notre application en fonction d'un `id`. Dans ce cas il faut ajouter une route à notre module de routage, qui possédera la syntaxe « **`:id`** » :

app-routing.module.ts

```
...

const routes: Routes = [
    { path : '', component : CompHomeComponent },
    { path : 'page1', component : CompPage1Component },
    { path : 'page1/:id', component : CompPage1Component },
    { path : 'page2', component : CompPage2Component },
    { path : 'admin', component : CompAdminComponent },
    { path : '**', redirectTo : '', pathMatch: 'full' }
];

...
```

Avec ce code une url de type **`/page1/15`** sera accessible.

On peut ajouter autant de paramètre qu'on veut, par exemple :

```
    { path : 'page1/:pays/:region/:ville/:id', component : CompPage1Component },
```

Ce code fera accéder par exemple à une url de type **`/page1/france/bretagne/rennes/123`**

3.9.3.3) ActivatedRoute

Il s'agit d'un service Angular qui donne des informations sur la route activée. Ce service est très riche et nous renvoyons à sa documentation pour en connaître les détails (https://angular.io/api/router/ActivatedRoute),

nous contentant ici d'en décrire l'essentiel.

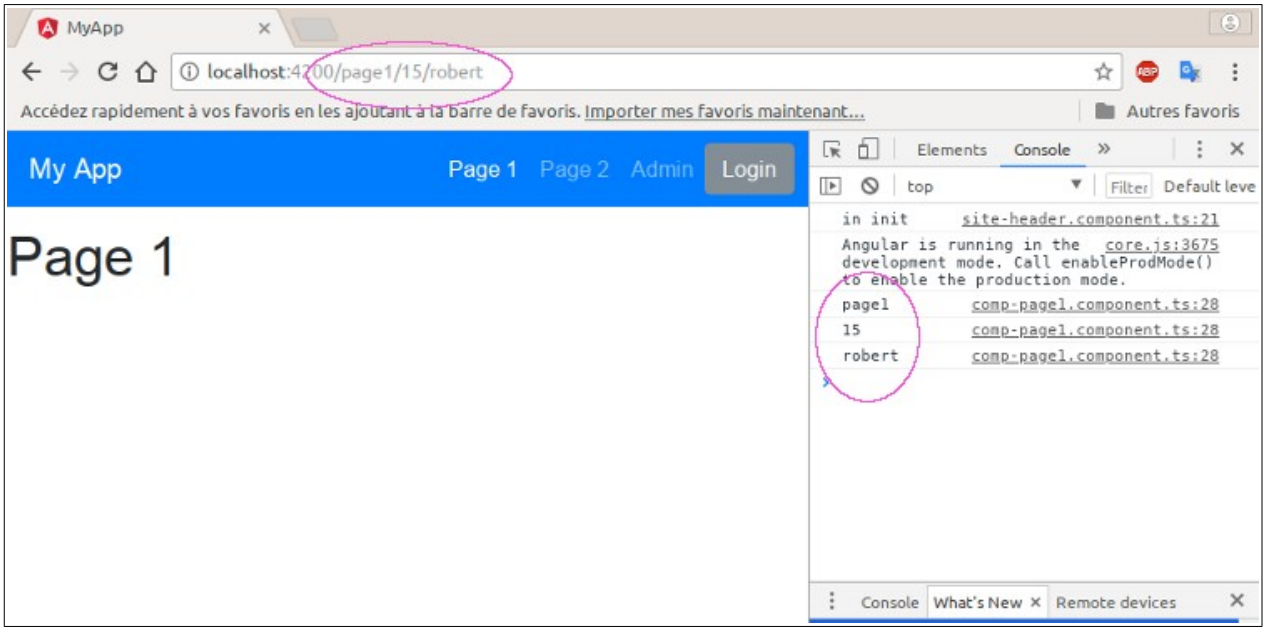
Pour l'utiliser dans un composant il faut l'y importer de la façon suivante :

comp-page1.component.ts

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

@Component({
  selector: 'app-comp-page1',
  templateUrl: './comp-page1.component.html',
  styleUrls: ['./comp-page1.component.css']
})
export class CompPage1Component implements OnInit {

  constructor(
    private activeRoute: ActivatedRoute
  ) { }

  ngOnInit() {

    for(let i=0; i<this.activeRoute.snapshot.url.length; i++) {
      console.log(this.activeRoute.snapshot.url[i].path);
    }
  }
```

A noter que nous intégrons le service `ActivatedRoute` *en l'introduisant dans les paramètres du* `constructor`. *Nous verrons plus loin que c'est la méthode d'Angular pour intégrer tous les services*

L'affichage de cet exemple donnera :



Voici deux autres façons de récupérer les paramètres de l'url active :

```
    console.log(this.activeRoute.snapshot.paramMap.get('id')); //affiche 15
    console.log(this.activeRoute.snapshot.params.name); //affiche robert
```

A noter que `id` et `name` sont récupérables seulement s'ils sont des paramètres stipulés dans la définition du routage :

app-routing.module.ts

```
  ...
    { path : 'page1/:id/:name', component : CompPage1Component },
  ...
```

3.9.3.4) Rediriger par le script

Le service Router permet de rediriger vers une page déclarée dans le routage, à partir du script javascript, c'est à dire sans besoin de clic de l'utilisateur. Pour cela il faut bien sûr avoir importé ce service. Voici un exemple d'utilisation :

comp-page1.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-comp-page1',
  templateUrl: './comp-page1.component.html',
  styleUrls: ['./comp-page1.component.css']
})
export class CompPage1Component implements OnInit {

  constructor(
    private router: Router
  ) { }

  ngOnInit() {
    this.router.navigate(['/page1', '18', 'hector']);
  }

}
```

A noter une nouvelle fois qu'un service doit être déclaré comme paramètre du constructor pour être utilisable, comme nous le verrons plus loin.

Ce code redirigera automatiquement vers l'url « **`localhost:4200/page1/18/hector`** », à condition bien sûr que le fichier de routing comporte :

app-routing.module.ts

```
  ...
    { path : 'page1/:id/:name', component : CompPage1Component },
  ...
```

3.9.3.5) URL avec GET data

Une question à ce stade est de savoir comment récupérer les données passées en argument GET dans l'URL. Par exemple pour l'URL **`localhost:4200/page1/18/hector?myvar=hello`**, nous souhaitons récupérer l'objet `{myvar : "hello"}`. Voici comment faire :

comp-page1.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute} from '@angular/router';
import 'rxjs/add/operator/filter';

@Component({
  selector: 'app-comp-page1',
  templateUrl: './comp-page1.component.html',
  styleUrls: ['./comp-page1.component.css']
})
export class CompPage1Component implements OnInit {

  data:any;

  constructor(
    private activeRoute: ActivatedRoute,
  ) { }

  ngOnInit() {

    this.activeRoute.queryParams
          .filter(params => params.myvar)
          .subscribe(params => {
              this.data = params;
          });

    console.log(this.data);
  }

}
```

A noter l'utilisation de **`rxjs`** *dans les imports. Il s'agit d'une bibliothèque qui gère les « observables » qui sont des éléments asynchrones. Nous reviendrons plus loin sur les observable dans un chapitre qui leur est dédié.*

3.9.3.6) Stop flickering

Nous pouvons constater un défaut majeur de la navigation : en cliquant sur un lien la totalité de l'application est rechargée, provoquant un flash visuel désagréable. Le moyen d'éviter cela est d'ajouter le paramètre `initialNavigation` aux imports du module de routing :

app-routing.module.ts

```
...

imports: [RouterModule.forRoot(routes,{ initialNavigation: 'enabled' })],

...
```

3.9.3.7) Protection des routes : CanActivate

Il est possible de protéger les routes, c'est à dire d'y donner accès uniquement si certaines conditions sont remplies. Supposons par exemple que notre application autorise toutes les routes à tout le monde, excepté la Page 1 qui ne sera accessible que si l'utilisateur possède un rôle d'ingénieur. Nous allons mettre cela en œuvre.

Pour commencer nous allons créer une classe User qui définira un rôle. En pratique une telle classe sera instanciée après reconnaissance de l'utilisateur à partir du serveur, par un login, par exemple, mais nous allons ici simplifier en écrivant une classe explicite. Pour ce faire nous créons la classe User de la façon suivante :

```
# ng generate class user
# mv user.ts lib/class
```

Ensuite nous l'éditons pour y inscrire le code suivant :

user.ts

```
export class User {
  role = 'engineer' ;
 cando = {} ;
}
```

En pratique les propriétés d'un utilisateur seront bien sûr plus riches, mais notre classe suffira pour l'exemple.

Notez que la propriété `cando` *est ici introduite car nous nous en servirons pour expliquer la méthode* `resolve` *d'une route.*

Créons maintenant le service `RouteGuardService` qui sera appelé pour vérifier que l'utilisateur possède le bon rôle pour consulter la bonne page, et plaçons le dans le répertoire `lib/services` :

```
# ng generate service authGuard
# mv route-guard.service.ts route-guard.service.spec.ts lib/services
```

Editons alors son code de la façon suivante :

route-guard.service.ts

```
import { Injectable } from '@angular/core';
import { CanActivate} from '@angular/router';
import { User } from '../class/user';
import {  ActivatedRoute,
          ActivatedRouteSnapshot,
          RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable()
export class RouteGuardService implements CanActivate {

  constructor( private user: User ) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean>|Promise<boolean>|boolean {

    let authaurized:boolean = true;
    let mainPage:string  = state.url.replace(/^\/([^\/]+)\/.*/, '$1');

    switch(mainPage) {
      case 'page1': {
        if(this.user.role!=='engineer') {
```

```
          authaurized = false;
        }
        break;
      }
      default : {
        authaurized = true;
        break;
      }
    }

    return authaurized;
  }

}
```

Dans ce service c'est la méthode `canActivate` qui effectuera les vérifications d'accès aux routes. Elle peut retourner soit un observable, soit une promise, soit un booléen. Les deux premiers cas concernent le retour d'éléments asynchrone, dont nous parlerons plus loin dans ce document. Cela arrive par exemple si on doit appeler un serveur pour déterminer les autorisations. Mais faisons simple : notre service retourne ici un simple booléen true/false. Notons aussi que cette méthode prend deux paramètres en entrée de types `ActivatedRouteSnapshot` et `RouterStateSnapshot`.

Le code de ce service nous indique que toutes les routes sont ouvertes à tous, excepté la Page 1 qui est refusée aux utilisateurs n'ayant pas un rôle d'ingénieur.

Il nous reste désormais à importer le service `RouteGuardService` dans le module de routage, et d'indiquer à la route concernée qu'elle ne sera accessible qu'après vérification des autorisations donnée par ce service. Choisissons par exemple de protéger ainsi la « Page 1 » de notre application, le module de routage sera alors modifier comme suit :

app-routing.module.ts

```
...
import { RouteGuardService } from './lib/services/allow-group.service';
import { UserToken } from './lib/class/user-token';
...

const routes: Routes = [
    ...
    {
      path : 'page1',
      component : CompPage1Component,
      canActivate: [RouteGuardService]
    },
    ...
];

@NgModule({
  ...
  providers: [
    ...
    RouteGuardService,
    User,
    ...
  ]
})
export class AppRoutingModule { }
```

Notez que nous avons conçu le service **`RouteGuardService`** *pour qu'il puisse concerner toutes les routes possibles, même si nous ne traitons ici que l'accès à la Page 1.*

Avec cette protection, un utilisateur n'ayant pas le rôle d'ingénieur ne verra aucun effet se produire lorsqu'il cliquera sur le lien « Page 1 ».

Attention : *Il est important de noter que la protection des routes n'est en général pas suffisante pour sécuriser une application. Il se peut en effet qu'un utilisateur soit autorisé à afficher une page, mais que ses autorisations ne lui permettent pas d'afficher, ou d'éditer, tel ou tel élément de cette page. Dans ce cas il faut une gestion des permissions plus élaborée, toujours par l'intermédiaire d'un service, directement importé dans le composant de la page concernée. Dans un tel cas l'utilisation de* **`canActivate`** *dans le routage se doublera d'une autre méthode de vérification, c'est à dire qu'il y aura au moins deux services d'autorisation à maintenir, ce qui n'est pas idéal. Il sera donc souvent plus intéressant de ne conserver qu'un seul service de vérification des autorisations, celui importé dans la page.*

3.9.3.8) Associer des données statiques à une route : data

Il peut être utile de passer des données associées à une route. Angular permet cela grâce à la propriété data de la définition d'une route. Prenons l'exemple d'un tel passage de données à la Page 1, à partir du module de routage. Modifions ce dernier de la façon suivante :

app-routing.module.ts

```
...
const routes: Routes = [
    ...
    {
      path : 'page1',
      component : CompPage1Component,
      data : {text:'this is my Text'}
    },
    ...
];
...
```

Il est alors possible de récupérer les données décrites par data dans la page 1, grâce à l'import du module `ActivatedRoute` dans le composant de la page concernée :

comp-page1.component.ts

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

@Component({
  selector: 'app-comp-page1',
  templateUrl: './comp-page1.component.html',
  styleUrls: ['./comp-page1.component.css'],
})
export class CompPage1Component implements OnInit {

  constructor(
    private route: ActivatedRoute
  ) { }

  ngOnInit() {
```

```
    console.log(this.route.snapshot.data) ;

  }

}
```

Notez que **`data`** *peut décrire un objet (Json) ou un tableau (array)*

Attention : **`data`** *n'est disponible que lorsque la route est acceptée, c'est à dire une fois le routage résolu, pas avant.*

3.9.3.9) Effectuer une opération avant le routage : resolve

Dans certaines situations il est utile d'aller plus loin que le passage de données statiques, tel que nous venons de le voir au paragraphe précédent, avec la propriété `data`. Ainsi on peut vouloir associer le résultat d'un service au routage, ce service étant exécuté **avant l'affichage de la route concernée**. Pour ce faire il faut d'abord créer un service un peu spécial, qui importera certains modules de routage, puis le déclarer dans notre module de routage, pour la page concernée.

En reprenant l'exemple du présent chapitre, qui affiche uniquement la Page 1 si l'utilisateur possède le rôle `engineer`, nous allons créer un service `RoleGuardService` qui stipulera que le rôle `engineer` a le droit d'éditer la page consultée.

Commençons par créer le service `RoleGuardService` :

role-guard.service.ts

```
import { Injectable } from '@angular/core';
import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot} from '@angular/router';
import { Observable } from 'rxjs';
import { User } from '../class/user';


@Injectable()
export class RoleGuardService implements Resolve<User> {

  constructor(public user: User) { }

  resolve(
      route: ActivatedRouteSnapshot,
      state: RouterStateSnapshot
    ): Observable<any>|Promise<any>|any {

    return this.getCando(this.user)
  }

  getCando(user:User):any {

    if(user.role==='engineer') {
      user.cando = {
        edit : true
      }
    }
    else {
      user.cando = {
        edit : false
```

```
      }
    }
    return user.cando;

  }
}
```

Notez l'utilisation de **`resolve`** *comme méthode du service, ainsi que les imports depuis* **`@angular/router`**

Dans cet exemple nous autorisons seulement les ingénieurs à éditer des données dans l'application.

Maintenant modifions le routage de notre application pour qu'il exécute ce service avant d'afficher la Page 1 :

app-routing.module.ts

```
...
import { RoleGuardService } from './lib/services/role-guard.service';
...

const routes: Routes = [
    ...
    {
      path : 'page1',
      component : CompPage1Component,
      canActivate: [AllowGroupService],
      data : {text:'this is my Text'},
      resolve : { userCando : RoleGuardService }
    }
    ...
];

@NgModule({
  ...
  providers: [
    ...
    RoleGuardService,
    ...
  ]
})
export class AppRoutingModule { }
```

Enfin, observons ce que la Page 1 reçoit en affichant le résultat dans la console :

comp-page1.component.ts

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

@Component({
  selector: 'app-comp-page1',
  templateUrl: './comp-page1.component.html',
  styleUrls: ['./comp-page1.component.css'],
})
export class CompPage1Component implements OnInit {

  constructor(
    private route: ActivatedRoute,
  ) { }
```

```
  ngOnInit() {

    console.log(this.route.snapshot.data.userCando)

  }

}
```

Notez que le résultat du **`resolve`** *est intégré à la propriété* **`data`** *du snapshot de la route active*

3.9.4) Routage secondaire

Nous avons vu le routage simple implémenté dans le module principal de l'application AppModule. Nous allons maintenant voir qu'on peut aussi définir des routages enfants et même déclarer un routage dans un module autre que le module principal. Auparavant une mise en place est nécessaire.

3.9.4.1) Mise en place

Nous souhaitons qu'une fois sur la page « Admin » nous ayons accès à un sous menu de navigation affichant deux pages , « Profile » et « Statistics ». Pour cela nous commençons par créer deux nouvelles pages, dans le répertoire pages/admin, de la façon dont nous avons généré les autres pages :

```
# cd pages/admin
# ng generate module profile
# cd profile
# ng generate component comp-profile
# cd ..
# ng generate module statistics
# cd statistics
# ng generate component comp-statistics
```

Ensuite nous modifions le template HTML du composant CompAdminComponent pour qu'il puisse recevoir le menu secondaire et afficher les pages « Profile » et « Statistics » :

comp-admin.component.html

```
<div class="topSiteMargin"></div>
<div class="container-fluid">
  <div class="row">
    <div class="card" style="width: 150px; border:0;background-color:lightgrey">
      <div class="card-body">
        <h5 class="card-title">Admin</h5>
        <a routerLink="profile" class="nav-link">Profile</a>
        <a routerLink="stat" class="nav-link">Statistics</a>
      </div>
    </div>
    <div class="col-xs-11" style="padding:20px">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

Dans ce template, structuré avec Bootstrap, nous remarquons deux choses importantes :

- l'utilisation de `routerLink` au lieu de `href` dans les liens de redirection, cela évite le flickering et permet de désigner le lien relativement à `/admin`

- la présence du tag `router-outlet` qui sera le réceptacle des pages désignées par les sous-routages.

Cette mise en place étant terminée, voyons comment coder le routage secondaire.

3.9.4.2) Children routing

Nous allons nous servir du fichier de routage principal de l'application pour y inscrire le routage secondaire lié à la page « Admin », c'est à dire modifier le fichier app-routing.module.ts de la façon suivante :

app-routing.module.ts

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

import { CompHomeComponent } from './pages/home/comp-home/comp-home.component';
import { CompPage1Component } from './pages/page1/comp-page1/comp-page1.component';
import { CompPage2Component } from './pages/page2/comp-page2/comp-page2.component';
import { CompAdminComponent } from './pages/admin/comp-admin/comp-admin.component';
import { CompProfileComponent } from
    './pages/admin/pages/profile/comp-profile/comp-profile.component';
import { CompStatisticsComponent } from
    './pages/admin/pages/statistics/comp-statistics/comp-statistics.component';

const routes: Routes = [
    { path : '', component : CompHomeComponent },
    { path : 'page1', component : CompPage1Component },
    { path : 'page1/:id/:name', component : CompPage1Component },
    { path : 'page2', component : CompPage2Component },
    { path : 'admin',
      component : CompAdminComponent,
      children : [
        { path : '', redirectTo : 'profile', pathMatch: 'full' },
        { path : 'profile', component : CompProfileComponent},
        { path : 'stat', component : CompStatisticsComponent},
        { path : '**', redirectTo : '', pathMatch: 'full' }
      ]
    },
    { path : '**', redirectTo : '', pathMatch: 'full' }
];

@NgModule({
  imports: [
    RouterModule.forRoot(routes,{ initialNavigation: 'enabled' })
  ],
  exports: [RouterModule],
  declarations : [
    CompHomeComponent,
    CompPage1Component,
    CompPage2Component,
    CompAdminComponent,
    CompProfileComponent,
    CompStatisticsComponent
  ],
  providers: []
})
export class AppRoutingModule { }
```

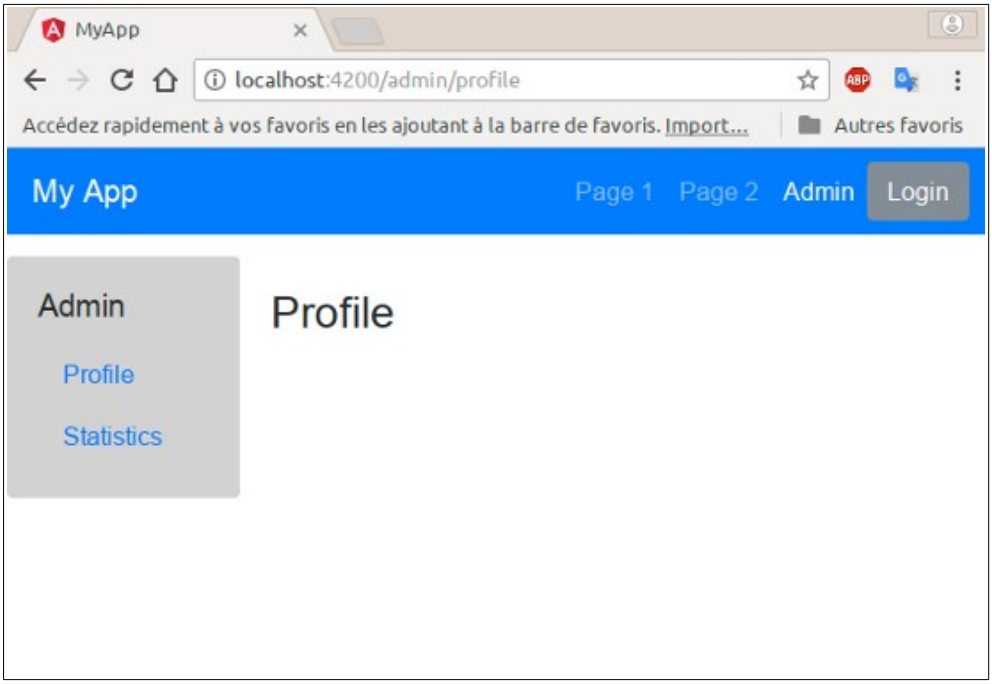
Classiquement, comme vu précédemment, nous importons les composants `CompProfileComponent` et `CompStatisticsComponent`, et nous les ajoutons aux déclarations du module. La chose nouvelle est que nous ajoutons au routage de la page « Admin » la propriété `children` qui est un tableau de routage

tout à fait similaire au tableau de routage principal.

Attention : les `path` *routage secondaire sont relatif au* `path` *principal dont ils dépendent (ici* `admin`*)*

Dans ce tableau de routage nous demandons à afficher par défaut la page « Profile » lorsque la route `/admin` *est réclamée.*

Affichons le résultat de ce code, et cliquons sur le lien « Admin » dans le header :



3.9.4.3) RouterModule.forChild

Il existe une autre façon de faire du routage secondaire, en ajoutant un module de routage au module dont dépendra ce routage, `AdminModule` dans notre cas. Voici comment procéder.

Il faut d'abord créer un module de routage, à l'instar de ce que nous avons fait pour le routage principal, mais cette fois-ci dédié au module Admin. Pour ce faire nous créons le fichier `admin-routing.module.ts` :

admin-routing.module.ts

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

import { CompAdminComponent } from './comp-admin/comp-admin.component';
import { CompProfileComponent } from
    './pages/profile/comp-profile/comp-profile.component';
import { CompStatisticsComponent } from
    './pages/statistics/comp-statistics/comp-statistics.component';

const adminRoutes: Routes = [
  { path: '',
    component: CompAdminComponent,
    children : [
```

```
      { path : '', redirectTo : 'profile', pathMatch: 'full' },
      { path : 'profile', component : CompProfileComponent},
      { path : 'stat', component : CompStatisticsComponent},
      { path : '**', redirectTo : '', pathMatch: 'full' }

    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes),
  ],
  exports: [RouterModule],
  declarations : [],
  providers: []
})
export class AdminRoutingModule { }
```

A noter l'utilisation de **`RouterModule.forChild()`** *à la place de* `RouterModule.forRoot()`

Attention, il ne peut exister qu'une seule utilisation de `RouterModule.forRoot()`*, seulement dans le routage principal, mais il peut exister plusieurs routages secondaires, et donc plusieurs utilisations de* `RouterModule.forChild()`

Ensuite il faut importer ce module de routage dans le module Admin :

admin.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule, Routes } from '@angular/router';

import { AdminRoutingModule } from './admin-routing.module';

import { CompAdminComponent } from './comp-admin/comp-admin.component';
import { CompProfileComponent } from
    './pages/profile/comp-profile/comp-profile.component';
import { CompStatisticsComponent } from
    './pages/statistics/comp-statistics/comp-statistics.component';

@NgModule({
  imports: [
    CommonModule,
    RouterModule,
    AdminRoutingModule
  ],
  declarations: [
    CompAdminComponent,
    CompProfileComponent,
    CompStatisticsComponent
  ]
})
export class AdminModule { }
```

Enfin, il nous faut modifier le fichier de routage principal de l'application en utilisant la syntaxe :

app-routing.module.ts

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

import { CompHomeComponent } from './pages/home/comp-home/comp-home.component';
import { CompPage1Component } from './pages/page1/comp-page1/comp-page1.component';
import { CompPage2Component } from './pages/page2/comp-page2/comp-page2.component';

const routes: Routes = [
    { path : '', component : CompHomeComponent },
    { path : 'page1', component : CompPage1Component },
    { path : 'page1/:id/:name', component : CompPage1Component },
    { path : 'page2', component : CompPage2Component },
    { path : 'admin',
      loadChildren:'./pages/admin/admin.module#AdminModule',
    },
    { path : '**', redirectTo : '', pathMatch: 'full' }
];

@NgModule({
  imports: [
    RouterModule.forRoot(routes,{ initialNavigation: 'enabled' })
  ],
  exports: [RouterModule],
  declarations : [
    CompHomeComponent,
    CompPage1Component,
    CompPage2Component,
  ],
  providers: []
})
export class AppRoutingModule { }
```

Au total l'affichage sera identique à celui obtenu dans le paragraphe précédent. Il peut donc paraître inutile de structurer le routage comme nous venons de le faire ici, cependant il peut être pratique si on veut modulariser une application en vue de la réutilisation des modules dans d'autres applications.

Notez que ce qui était valable pour la propriété `data` *et la méthode* `resolve` *du routing principal, reste identique pour les routes secondaires.*

3.9.4.4) Protection des routes : CanActivateChild

Bien sûr les child routes peuvent être sécurisées de la même façon que nous avons sécurisé les routes principales, et notamment la Page 1 dans le chapitre précédent. On peut aussi associer les propriétés `data` et `resolve` à une child route.

Pour autoriser les childs routes déclarées dans la section Admin, nous modifions AppRoutingModule de la façon suivante :

app-routing.module.ts

```
...
import { RouteGuardService } from '../../lib/services/allow-group.service';
import { RoleGuardService } from '../../lib/services/role-guard.service';
...

const routes: Routes = [
    { path : '', component : CompHomeComponent },
    {
```

```
      path : 'page1',
      component : CompPage1Component,
      canActivate: [RouteGuardService],
      data : {text:'this is my Text'},
      resolve : { userCando : RoleGuardService }
    },
    { path : 'page1/:id/:name', component : CompPage1Component },
    { path : 'page2', component : CompPage2Component },
    { path : 'admin',
      canActivateChild: [RouteGuardService],
      loadChildren:'./pages/admin/admin.module#AdminModule',
    },
    { path : '**', redirectTo : '', pathMatch: 'full' }
];
@NgModule({
  ...
})
export class AdminRoutingModule { }
```

Notez l'utilisation de **`canActivateChild`** *pour la protection du routage secondaire, au lieu de* **`canActivate`** *pour la protection du routage principal.*

Ensuite il faut modifier le service RouteGuardService pour lui faire refuser toutes les routes enfants de Admin :

route-guard.service.ts

```
import { Injectable } from '@angular/core';
import { CanActivate, CanActivateChild } from '@angular/router';
import { User } from '../class/user';
import { ActivatedRoute, ActivatedRouteSnapshot,RouterStateSnapshot } from
'@angular/router';
import { Observable } from 'rxjs';

@Injectable()
export class RouteGuardService  implements CanActivate, CanActivateChild {

  constructor( private user: User ) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean>|Promise<boolean>|boolean {

    let authaurized:boolean = true;
    let mainPage:string  = state.url.replace(/^\/([^\/]+)\/.*/, '$1');

    switch(mainPage) {
      case 'page1': {
        if(this.user.role!=='engineer') {
          authaurized = false;
        }
        break;
      }
      case 'admin': {
        if(this.user.role!=='engineer') {
          authaurized = false;
        }
        break;
      }
      default : {
```

```
        authaurized = true;
        break;
      }
    }

    return authaurized;

  }

  public canActivateChild(
    childRoute: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | Observable<boolean> | Promise<boolean> {
    return this.canActivate(childRoute, state);
  }

}
```

Notez que la méthode **`canActivateChild`** *renvoie simplement vers la méthode* **`canActivate`**. *On peut procéder autrement mais cette façon de faire permet de gérer la totalité du routage dans une seule fonction.*

Dans cet exemple toutes les routes du module `AdminRoutingModule` (vers la page `profile`, `stat`, etc.) seront sécurisées par le service `RouteGuardService`.

3.9.5) Lazy loading

Les performances d'affichage d'une page web dans un onglet de navigateur sont tellement réduite qu'il est intéressant de ne charger que ce qui est essentiel. Sur une grosse application il arrive souvent que certains utilisateurs utilisent certaines fonctionnalités, tandis que d'autre en utiliseront des différentes. Par exemple si seul l'administrateur peut afficher le bureau d'administration de l'application, l'utilisateur ne se servira jamais du code et des templates HTML qui lui sont associés. Il est donc inutile que l'utilisateur charge le module « Admin », au risque de dégrader ses performances. C'est dans ce but qu'a été conçu le « lazy loading ». Cette technique permet de ne charger du javascript uniquement lorsque c'est absolument nécessaire.

La bonne pratique est de définir un app-routing.module.ts, par exemple comme suit :

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { RouteGuardService } from './lib/services/route-guard.service';
...
const routes: Routes = [
---
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin.module').then((mod : any) =>
      mod.AdminModule
     ),
    canActivate: [RouteGuardService]
  },
  {
    path: 'skipper',
    loadChildren: () => import('./skipper/skipper.module').then((mod : any) =>
```

```
      mod.SkipperModule
    ),
    canActivate: [RouteGuardService]
  },
...
]

@NgModule({
  imports: [
    RouterModule.forChild(routes, { useHash: true }),
       ...
  ]

...
}
export class AppRoutingModule { }
```

Dans cet exemple, la section de l'application réservée à l'administrateur est une lazy route, ainsi que celle de la partie de l'application réservée aux skippers.
Bien sûr le **`AppModule`** doit importer le routing module :

app.module.ts

```
import { AppRoutingModule } from ./app-routing.module';
...
imports : [
  AppRoutingModule
  ...
]
...
```

On comprend que chaque lazy route est contenue dans un module, lui même embarquant des composants, et services.

Le module décrivant une lazy route sera équivalent en structure au module AppRoutingModule, avec une description de ses sous-routes réservée à cette lazy route.

3.10) Custom directives

Une directive est une classe particulière qui va chercher dans le HTML le tag qui possède un attribut ou une classe particulière, et qui agit sur lui.

3.10.1) Création

Prenons pour exemple une directive « `ColorizeDirective` » qui, au passage de la souris, va changer la couleur de fond d'un élément possédant l'attribut « `colorize` ».

Créons la directive :

```
# cd lib
# mkdir directives
# cd directives
# ng generate directive colorize
```

Editons ensuite la directive et modifions le code de base créé par Angular de la façon suivante :

colorize.directive.ts

```
import { Directive, Renderer, ElementRef } from '@angular/core';

@Directive({
  selector: '[colorize]'
})
export class ColorizeDirective {

  constructor(
    private el: ElementRef,
    private renderer: Renderer
  ) {
      renderer.setElementStyle(el.nativeElement, 'backgroundColor', '#ddd');
  }

}
```

Quelques explications sont ici nécessaires :

- Le Module **`Renderer`** permet à Angular de gérer toutes sortes de navigateurs, notamment des terminaux mobiles,
- Le module **`ElementRef`** permet de manipuler des éléments du DOM
- Le sélector **`[colorize]`** signifie que Angular va chercher tous les éléments du DOM qui ont un **attribut** colorize. Il est aussi possible d'utiliser un sélecteur de **classe CSS**, par exemple si nous voulions sélectionner tous les éléments qui possèdent la classe « `colorize` », nous écririons :

```
  selector: '.colorize'
```

- Il faut instancier `Renderer` et `ElementRef` **dans les paramètres du `constructor`** de la directive
- On modifie la couleur de fond de l'élément concerné dans le corps du `constructor`

Notre directive est désormais prête, reste à la rendre disponible dans l'application, mais il y a ici une subtilité liée à l'organisation de notre application et à l'utilisation du module de routage. C'est ce que nous allons décrire.

3.10.2) Bonne pratique d'intégration

A priori notre directive est destinée à être utilisée dans les pages désignées par le routage, mais pas dans le module supérieur AppModule, comme c'est le cas pour le header du site. C'est donc le module de routage qui doit l'importer, comme nous l'avons déjà fait remarqué dans la section « **Organisation des éléments/ Généralisation et cas particulier** ». Il en ira d'ailleurs de même pour tous les éléments qui se trouveront dans la même situation (pipes, composants, services, modules), bien que pour l'instant nous ayons importé un pipe et un service dans le module principal, car ices derniers pourraient nous être utiles dans la globalité de l'application, header compris.

Si nous devons intégrer de nombreux éléments, le fichier de routage va vite devenir illisible, c'est pourquoi une bonne pratique est d'importer les éléments partagés dans un module dédié à cela, puis d'importer ce seul module dans AppRoutinModule. C'est ce que nous allons faire.

Créons donc un module partagé « shared » dans notre bibliothèque des modules :

```
# ng generate module shared
# mv shared lib/modules
```

Puis demandons lui d'intégrer notre directive, et de l'exporter :

shared.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { ColorizeDirective } from '../../directives/colorize.directive';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    ColorizeDirective
  ],
  exports : [
    ColorizeDirective
  ]
})
export class SharedModule { }
```

Ensuite intégrons ce module au module de routage :

app-routing.module.ts

```
...

import { SharedModule } from './lib/modules/shared/shared.module';

...

@NgModule({
  imports: [
    ...
    SharedModule
  ],
  exports: [
    ...
    SharedModule
  ],
  ...
})
export class AppRoutingModule { }
```

Nous pouvons désormais utiliser cette directive dans du HTML de notre application. Modifions par exemple le template du composant Home en ajoutant l'attribut « `colorize` » au titre de la page :

comp-home.component.ts

```
<div class="topSiteMargin"></div>
<h1 colorize>Accueil</h1>
```

Au rechargement la page affichera :

Attention : répétons le, l'import de `SharedModule` *(qui exporte* `ColorizeDirective`*) directement dans* `AppModule` *ne permettra pas d'utiliser la directive dans le template du composant* `CompHomeComponent` *car ce dernier est décrit dans le module de routage.*

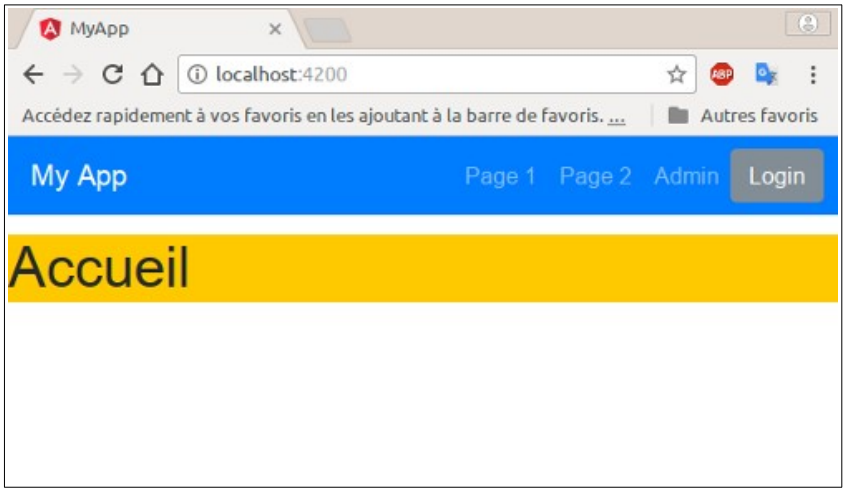
3.10.3) Gestion des événements

Pour l'instant notre directive est très statique. Voyons donc comment la faire réagir à des événements, par exemple en changeant sa couleur lorsque la souris survole le composant concerné. Pour cela il nous faut intégrer le module **`HostListener`** à la directive. Comme son nom l'indique ce module écoute les événements utilisateurs sur l'élément qu'on lui désigne. Modifions donc notre directive :

colorize.directive.ts

```
import { Directive, Renderer, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[colorize]'
})
export class ColorizeDirective {

  constructor(
    private el: ElementRef,
    private renderer: Renderer
  ) {
      renderer.setElementStyle(el.nativeElement, 'backgroundColor', '#ddd');
  }

    @HostListener('mouseenter', ['$event']) onMouseEnter(event: Event) {
      this.renderer.setElementStyle(this.el.nativeElement, 'backgroundColor', '#fc0');
    }

    @HostListener('mouseleave', ['$event']) onMouseLeave(event: Event) {
      this.renderer.setElementStyle(this.el.nativeElement, 'backgroundColor', '#ddd');
    }

}
```

Désormais lorsque la souris passe sur l'élément titre associé à la directive colorize nous obtenons :



3.10.4) Directive @Input

Il serait pratique de spécifier la couleur de notre bandeau, non plus en dur dans le code de la directive, mais directement dans le template HTML, de cette façon :

comp-home.component.html

```
<div class="topSiteMargin"></div>
<h1 colorize color="#aaf">Accueil</h1>
<app-comment></app-comment>
```

On s'attend alors à ce que la couleur de fond du bandeau devienne bleu lorsqu'on passe la souris. Pour faire cela il faut prévenir la directive que la couleur lui sera passée en paramètre, et qu'elle pourra alors l'utiliser dans son code. Cela se fait grâce au décorateur @Input :

colorize.directive.ts

```
import { Directive, Renderer, ElementRef, HostListener, Input} from '@angular/core';

@Directive({
  selector: '[colorize]',
})
export class ColorizeDirective {

  @Input() color:string;

  constructor(
    private el: ElementRef,
    private renderer: Renderer
  ) {
      renderer.setElementStyle(el.nativeElement, 'backgroundColor', '#ddd');
  }

  @HostListener('mouseenter', ['$event']) onMouseEnter(event: Event) {
    this.renderer.setElementStyle(
       this.el.nativeElement, 'backgroundColor', this.color
    );
  }

  @HostListener('mouseleave', ['$event']) onMouseLeave(event: Event) {
    this.renderer.setElementStyle(this.el.nativeElement, 'backgroundColor', '#ddd');
  }
```

```
}
```

Au rechargement l'affichage est le même que celui de la figure précédente, à cela près que le bandeau devient bleu, et plus jaune, au passage de la souris.

On peut ajouter autant d'inputs qu'on souhaite dans une directive.

Attention : Angular lit le HTML après avoir lu le `constructor`*, ainsi* ***les input d'une directive ne seront pas accessibles dans le*** **`constructor`** *de cette directive. Si on souhaite que la valeur d'un input de directive soit effectif au chargement de la page, il faut s'en servir dans la fonction* **`ngOnInit()`** *plutôt que dans le* `constructor`*.*

3.11) Forms

Angular met permet des procédures de contrôle et de validation des formulaires, que nous allons décrire ici. A titre d'exemple nous créerons un composant « comment » destiné à recueillir les commentaires des utilisateurs.

3.11.1) Création

Générons donc le composant et plaçons le dans la bibliothèque des composants :

```
# ng generate component comment
# mv comment lib/components
```

Editons ensuite la classe de ce composant pour y définir le data model correspondant au formulaire :

comment.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-comment',
  templateUrl: './comment.component.html',
  styleUrls: ['./comment.component.css']
})
export class CommentComponent implements OnInit {

  myform: FormGroup;

  constructor() { }

  ngOnInit() {

   this.myform = new FormGroup({
       comment : new FormControl(),
       email: new FormControl()
   });

  }
```

```
 onSubmit() {
   console.log(this.myform.value);
 }

}
```

Notez :
- *l'import des modules* **`FormGroup`** *et* **`FormControl`** *depuis Angular/forms*
- *un* `FormGroup` *contient des* `FormControl`*, mais peut aussi contenir d'autres* `FormGroup`
- la présence de la fonction **`onSubmit()`** (vous pouvez choisir un autre nom) destinée à être exécutée à la validation du formulaire

Ecrivons maintenant le code HTML du template correspondant :

comment.component.html

```
<div class="container">
  <div class="card" style="width: 18rem;">
    <div class="card-body">
      <h5 class="card-title">Laissez un commentaire</h5>

      <form [formGroup]="myform">
        <div class="form-group" (ngSubmit)="onSubmit()">
          <label>Commentaire</label>
          <textarea class="form-control"
                    rows="3"
                    formControlName="comment"
                    required
          ></textarea>
        </div>
        <div class="form-group">
          <label>E-mail</label>
          <input  type="text"
                  class="form-control"
                  placeholder="Adresse e-mail"
                  formControlName="email"
                  required
          />
        </div>
        <button type="submit" class="btn btn-primary">Valider</button>
      </form>

    </div>
  </div>
</div>
```

Notez
- *l'utilisation de* **`formControlName`** *qui permettra à Angular de lier les éléments textarea et input à a classe du composant.*
- l*a présence de* **`(ngSubmit)="onSubmit()"`** *qui délègue à la fonction* `onSubmit()` *le traitement des données reçues par le composant lorsqu'il est validé.*

3.11.2) Intégration et exécution

Pour que ce composant puisse être partagé sur différentes pages de l'application, nous l'importerons dans

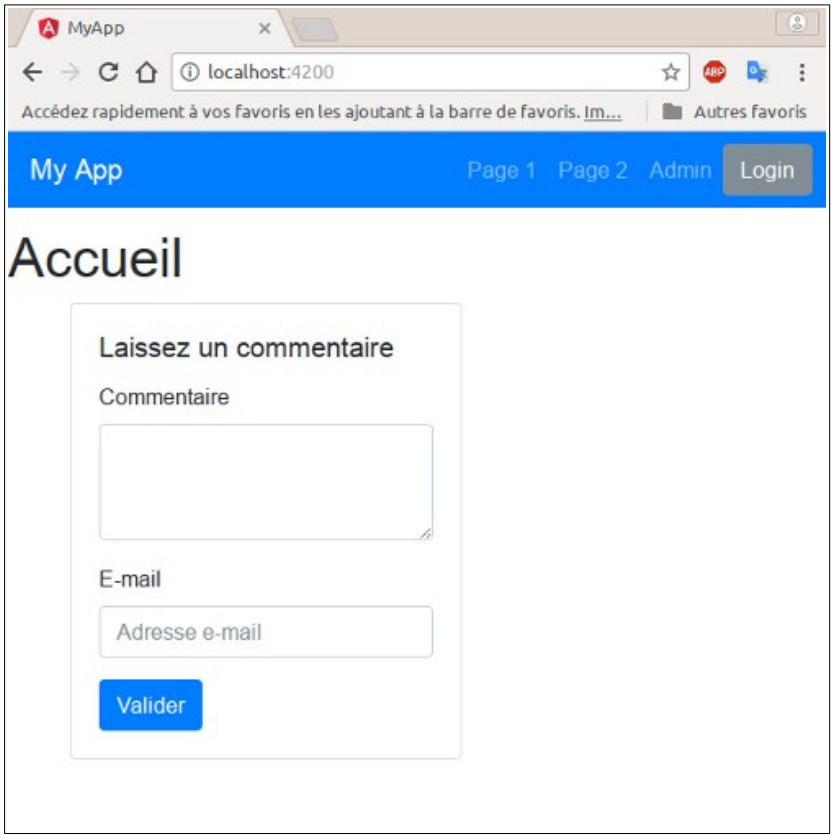
notre module `SharedModule` :

shared.module.ts

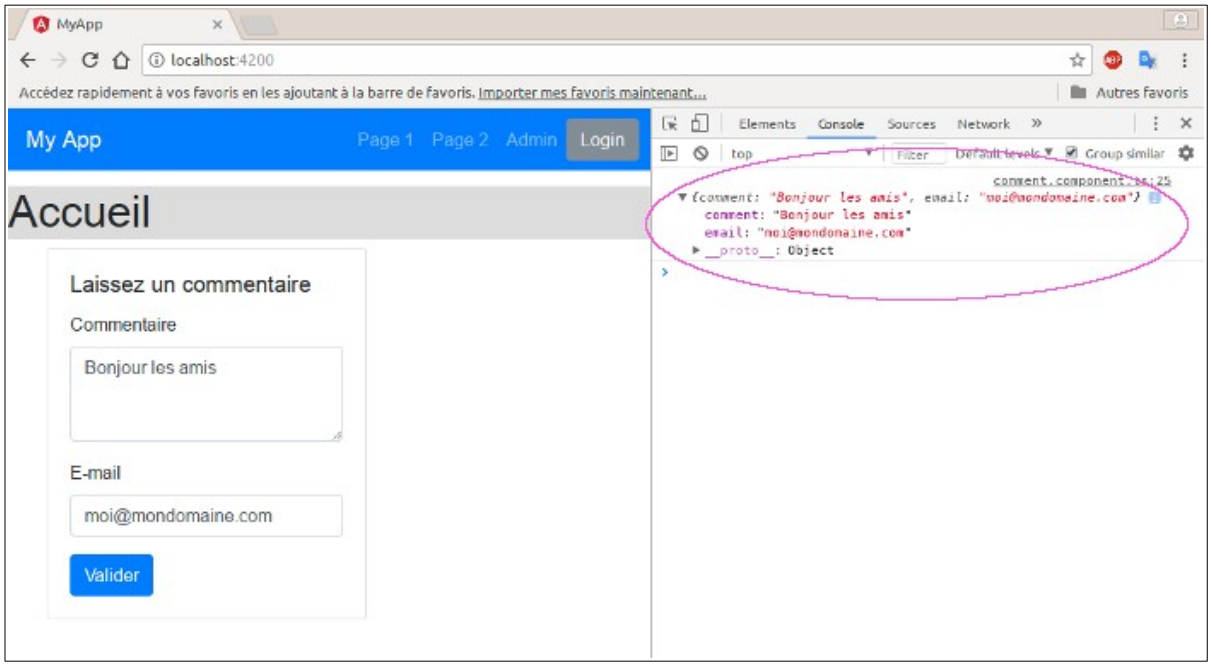
```
...

import { ReactiveFormsModule } from '@angular/forms';

...

import { CommentComponent } from '../../components/comment/comment.component';

...

@NgModule({
  imports: [
    CommonModule,
    ReactiveFormsModule
  ],
  declarations: [
    CommentComponent,

    ...
  ],
  exports : [
    CommentComponent,

    ...
  ]
})

export class SharedModule { }
```

Attention : nous utilisons le module **`ReactiveFormsModule`** *plutôt que le module* `FormsModule` *car ce dernier comporte un bug à l'heure où nous écrivons ces lignes, et provoque un message d'erreur.*

Une fois ceci réalisé le rechargement de la page affiche ce qui suit :



A l'exécution nous obtenons l'affichage suivant :



3.11.3) Validation

Angular intègre des validateurs de formulaires. Pour les utiliser il faut importer le module Validators et

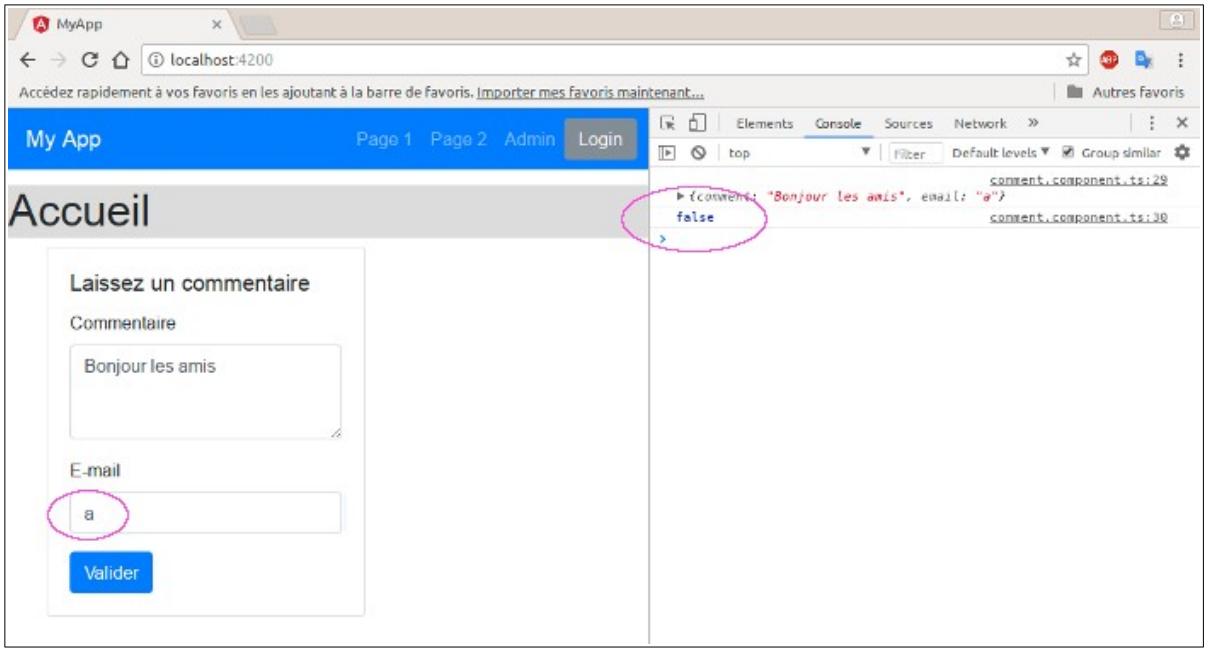
modifier le data model de notre formulaire. Voici par exemple comment intégrer la validation de l'e-mail :

comment.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-comment',
  templateUrl: './comment.component.html',
  styleUrls: ['./comment.component.css']
})

export class CommentComponent implements OnInit {

  myform: FormGroup;

  constructor() { }

  ngOnInit() {

    this.myform = new FormGroup({
       comment : new FormControl(),
       email: new FormControl('', [
            Validators.required,
            Validators.pattern("[^ @]*@[^ @]*")
       ]),
   });

  }

   onSubmit() {
      console.log(this.myform.value);
      console.log(this.myform.controls.email.valid);
  }

}
```

*A noter l'expression régulière « **[^ @]*@[^ @]*** » que doit vérifier l'email pour être validé.*

Avec ce code la propriété **this.myform.controls.email.valid** sera vraie ou fausse si l'email respecte ou non le critère du « pattern » de son validateur. Voici ce que donne l'affichage avec un email non valide :



Charge à vous de déclencher toute action en fonction du retour du validateur.

Il existe de nombreux validateurs prédéfinis (pattern, min et max, min et maxlength, …) mais vous pouvez aussi créer vos validateurs personnalisés. Nous vous renvoyons vers la documentation pour en savoir plus (https://angular.io/guide/form-validation).

3.11.4) Reset

Pour revenir à l'état initial d'un formulaire il suffit d'utiliser la methode `reset()` qui lui est associée par défaut. Ajoutons d'abord au template un bouton «Annuler» dont le clic renvoie vers l'exécution d'une fonction `resetForm()` :

comment.component.html

```
...
      <form [formGroup]="myform" (ngSubmit)="onSubmit()">
        ...
        <button type="submit" class="btn btn-primary">Valider</button>
        <button (click)="resetForm()" class="btn btn-primary">Annuler</button>
      </form>
...
```

Ajoutons ensuite la fonction resetForm() à notre composant :

comment.component.ts

```
…
export class CommentComponent implements OnInit {

  ...

  constructor() {}

  ngOnInit() {

    ...
```

```
  }

  ...

  resetForm() {
    this.myform.reset();
  }

  ...
}
```

3.11.5) Reactive forms

Vous avez certainement déjà rencontré des formulaires, de recherche par exemple, qui vous font des propositions interactivement, à chaque fois que vous entrez un nouveau caractère dans un input. C'est tout l'objet des « Reactive forms » d'Angular. Nous verrons dans le chapitre suivant, « Asynchronisme », que tous les événements d'interface sont asynchrones en javascript. Par conséquent il faut traiter ce problème d'une façon particulière, en utilisant les Observables qui sont des objets capables de gérer l'asynchronisme. La description exhaustive des observables n'est pas dans l'objectif de ce cours, nous vous renvoyons vers la documentation si vous souhaitez en savoir plus : http://reactivex.io/documentation/observable.html.

La première chose à faire est d'importer le module ReactiveFormsModule dans le module qui gère le composant du formulaire. Dans notre cas il s'agit de `SharedModule` :

shared.module.ts

```
...
import { ReactiveFormsModule } from '@angular/forms';
...
```

En fait nous l'avions déjà importé, à la place du `FormsModule` simple, à cause d'un problème de bug sur la version d'Angular que nous utilisons (voir plus haut).

Ensuite il faut modifier le composant du formulaire, et en premier lieu importer les observables, car il s'agit d'un fonctionnement asynchrone, puis il faut associer une action à l'élément concerné par ce comportement avec une syntaxe particulière, réservée aux observables. Nous allons traiter ainsi le champ « E-mail » :

Comment.component.ts

```
import { Component, OnInit, Output, EventEmitter } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-comment',
  templateUrl: './comment.component.html',
  styleUrls: ['./comment.component.css']
})

export class CommentComponent implements OnInit {

  myform: FormGroup;

  constructor() {}

  ngOnInit() {
```

```
    this.myform = new FormGroup({
       comment : new FormControl(),
       email: new FormControl('', [
            Validators.required,
            Validators.pattern("[^ @]*@[^ @]*")
        ]),
    });

    this.myform.controls.email.valueChanges
        .subscribe(term => {
          console.log('email is: ',this.myform.controls.email.value)
        });

  }

  onSubmit() {
    console.log(this.myform.value);
    console.log(this.myform.controls.email.valid);
  }

  resetForm() {
    this.myform.reset();
  }

}
```

L'utilisation de `subscribe` est typique du fonctionnement des observables, et en l'occurrence notre observable est ici `valueChanges`.

Notez qu'avec la structure du data model que nous utilisons pour notre `FormGroup`*, l'objet* `email` *n'est pas accessible par* `this.myform.email`*, mais par* `this.myform.controls.email`.

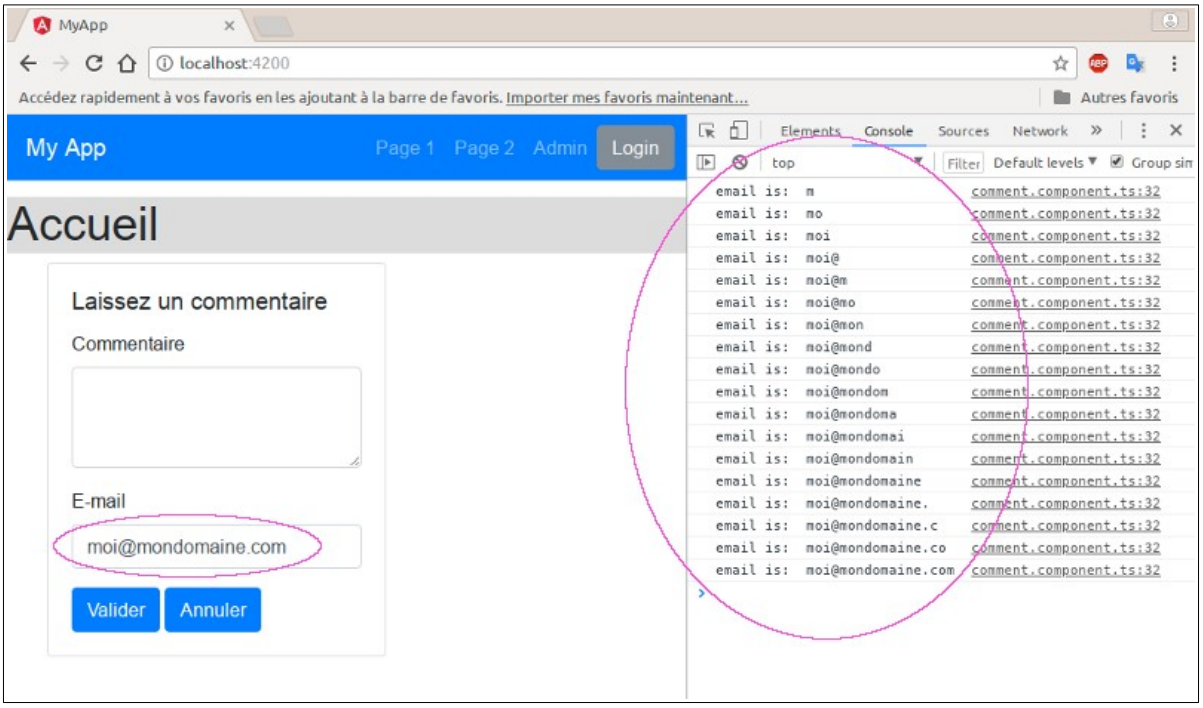
Il ne nous reste plus qu'à modifier le template de notre formulaire de la façon suivante :

comment.component.html

```
...
        <div class="form-group">
          <label>E-mail</label>
          <input  type="text"
                  class="form-control"
                  placeholder="Adresse e-mail"
                  [formControl]="myform.controls.email"
                  required
          />
        </div>
...
```

Notez qu'on a remplacé `formControlName="email"` *par* **`[formControl]="myform.controls.email"`**

Au rechargement, et après avoir complété le champ email, voici ce que nous obtenons :



3.12) HTTP et asynchronismes

3.12.1) Le service HttpClient

Tout ce qu'un codeur attend d'une api c'est de pouvoir expédier une requête à un serveur et d'en obtenir une répose. Par conséquent il a besoin d'une fonction à seulement deux paramètres, une url lui indiquant l'adresse du serveur à interroger, avec un payload constitué d'une requête sous un format objet (JSON). Et comme tout ça passe par le réseau, la réponse sera forcément asynchrone, c'est à dire que la fonction retournera un observable. Par conséquent la fonction idéale serait :

```
myApiFunction(url : string, payload : any).subscribe((data : any) => {
  this.myCallback(data);
}
```

Alors Angular vous fournit le service **`HttpClient`** qui fait exactement cela. Dans votre composant il haut importer le service **`HttpClient`** :

my-component.component.ts

```
...
import { HttpClient }  from '@angular/common/http';
...

@Component({
  selector: 'app-comp-page2',
  templateUrl: './comp-page2.component.html',
  styleUrls: ['./comp-page2.component.css'],
  providers: [DataService]
})
export class CompPage2Component implements OnInit {

  constructor(
```

```
    private http:HttpClient
  ) {}

  ngOnInit() {

    const url = 'myserver/page2/api5';
    const request = {
      page : 36,
      authors : [ {name : 'abc', editor : 'def'}, ... ],
      ageOfCptain : 25
    }

    this.http.post(url, request).subscribe((data : any) => {
      this.callback(data) ;
    }) ;
```

```
  }

  callback(data : any) {
    ...
  }

}
```

Le service **`HttpClient`** est accompagné de plusieurs autres services d'api spécialisés : **`HttpResponse`**, **`HttpParams`**, **`HttpRequest`**, ... Nous laisserons le lecteur visiter la documentation officielle pour obtenir plus d'informations.

3.12.2) Résolution des observables

Jongler avec les observables est est complexe et périlleux, dès lors il faut les résoudre le plus vite possible dans un réceptacle qui pourra être atteint par n'importe quel composant de l'application. La bonne pratique est de réaliser cela avec un service standard Angular, qu'on nomme généralement **`store`** et qui est chargé dans les providers du **`AppModule`**. Ainsi ce service sera commun à toute l'application. Tout composant, ou autre service qui importera le service **`store`** sera alors directement connecté même référentiel **`store`**. Toute modifications d'une propriété de **`store`** se répercutera immédiatement dans les composants et services qui l'importent, sans avoir à écrire une seule ligne de code.

Cette propriété technologique des services Angular doit être mise à profit pour éviter d'avoir à manipuler des observables. En intégrant un tel service store, le composant précédent donnerait ceci :

my-component.component.ts

```
...
import { HttpClient }  from '@angular/common/http';
import { StoreService }  from './services/store.service';
...

@Component({
  selector: 'app-comp-page2',
  templateUrl: './comp-page2.component.html',
  styleUrls: ['./comp-page2.component.css'],
  providers: [DataService]
})
export class CompPage2Component implements OnInit {
```

```
  constructor(
     private http: HttpClient
     private store: Store.service
  ) {}

  ngOnInit() {

    const url = 'myserver/page2/api5';
    const request = {
      page: 36,
      authors: [ {name: 'abc', editor: 'def'}, ... ],
      ageOfCptain: 25
    }

    this.http.post(url, request).subscribe((data: any) => {
      this.callback(data);
    });
```

```
  }

  callback(data: any) {
    this.store.data = data;
    ...
  }

}
```

3.12.3) Utilisation des données asynchrones dans les templates

Les templates des composants reconnaissent l'elvis operator « **?.** » dans les templates HTML, qui indique au javascript que la variable qui suit est asynchrone. Ainsi lorqu'on veut afficher **{{store.data}}** le javascript provoquera une erreur d'asynchronisme, en revanche en demandant d'afficher **{{store?.data}}**, le javascript n'émettra aucune erreur, et affichera **store.data** dès que cet objet sera instancié. Par exemple :

```
<div>Name: {{store?.user.name}}</div>
```

Si l'elvis operator indique à Angular qu'il doit scanner à chaque boucle pour vérifier si data a été instanciée, il arrive que cela ne convienne pas dans certaines situations.

Il faut alors utiliser les tags **ng-container**:

```
<ng-container *ngIf="condition"> ... </ng-container>
```

La condition peut-être par exemple l'instanciation de la propriété **store.data**. Ainsi tout élément HTML inclus à l'intérieur de cette balise ne sera affiché que lorsque **store.data** aura été instancié.

Avec cette façon de gérer les asynchronismes du javascript vous simplifierez considérablement le codage, c'est une bonne pratique à appliquer absolument.

3.12.4) HTTPInterceptor

Il peut être utile de modifier les propriétés d'une requête HTTP avant de l'expédier au serveur. Cela est utile par exemple si vous avez stocké un token d'authentification dans le localStorage, et que vous souhaitez l'en

extraire pour l'inclure à toutes vos requêtes vers le serveur. Angular permet de le faire grâce au module `HttpInterceptor`, qui permet de modifier les headers de requêtes HTTP. Voyons comment faire cela sur un exemple simple.

Pour commencer il faut créer un service d'interception que nous nommerons `InterceptorOutService`, et que nous placerons comme d'habitude dans `lib/services` :

```
# ng generate service interceptor-out
# mv interceptor-out.service.ts interceptor-out.service.spec.ts /lib/services
```

Editons ensuite notre service et demandons lui d'intégrer les propriétés de l'utilisateur, incluant ce qu'il est autorisé à faire par le service `RoleGuardService`, de la façon suivante :

interceptor-out.service.ts

```
import { Injectable } from '@angular/core';
import {  HttpInterceptor,
          HttpHandler,
          HttpRequest,
          HttpEvent } from '@angular/common/http';
import { Observable } from 'rxjs';
import { RoleGuardService} from './role-guard.service';
import { User } from '../class/user';

@Injectable()
export class InterceptorOutService implements HttpInterceptor {

  constructor(
    private user:User,
    private roleService:RoleGuardService
  ) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    this.user.cando = this.roleService.getCando(this.user);

    const modified = req.clone(
       {setHeaders: {'Custom-Header': JSON.stringify(this.user)}}
    );

    return next.handle(modified);
  }
}
```

Notez l'utilisation de **`next-handle()`** *qui indique à Angular de poursuivre son processus HTTP après avoir tenu compte de l'intercepteur.*

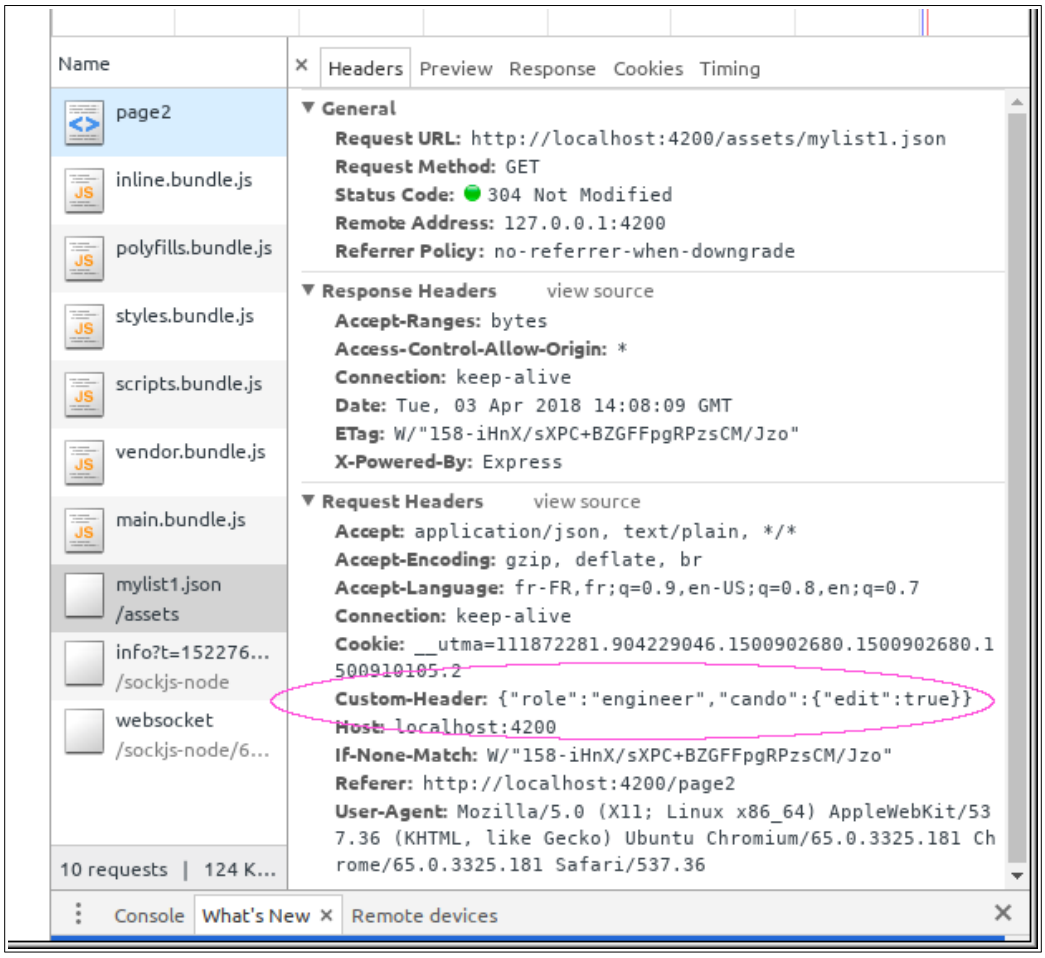
Pour finir il est nécessaire d'indiquer au module responsable, dans notre cas AppRoutingModule, qu'il devra utiliser l'intercepteur pour toute requête HTTP. Cela se fait en modifiant les providers du module :

app-routing.module.ts

```
...
import { HttpClientModule, HttpClient, HTTP_INTERCEPTORS} from '@angular/common/http';
...

@NgModule({
  ...
```

```
  providers: [
    ...
    {
      provide: HTTP_INTERCEPTORS,
      useClass: InterceptorOutService,
      multi: true
    },
    ...
  ]
})
export class AppRoutingModule { }
```

Désormais, un clic vers la Page 2, appelant l'url serveur http://localhost:4200/assets/mylist1.json, par l'intermédiaire de la fonction `getPromiseHttpData()`, qui est une méthode du service `DataService`, expédie une requête dont le header a été enrichi comme présenté sur l'image suivante servie par le debugger du navigateur :



Notez qu'il est possible de se faire succéder autant d'intercepteurs que désiré, ces derniers étant joués dans leur ordre d'apparition dans les **`providers`** *du module concerné, ici* **`AppRoutingModule`**.

3.12.5) Plus sur les API HTTP

Dans les paragraphes précédents nous avons utilisé uniquement la méthode GET, sans paramètres, mais Angular permet aussi les méthodes GET avec paramètre, POST PUT et DELETE. De façon classique ces méthodes doivent être associées à des options, ces dernières étant ajoutées en paramètre, en plus de l'url. Ce cours n'a pas vocation à décrire ces méthodes en détail et nous renvoyons donc vers la documentation pour en savoir plus : https://angular.io/guide/http

3.13) Documentation

Tous les éléments logiciels que nous avons utilisés dans ce document possèdent de nombreuses subtilités et fonctionnalités qu'il eut été trop long de présenter en détail ici. Nous laissons à la charge du lecteur d'approfondir ses connaissances en se référant aux différentes documentations :

- **Angular** : https://angular.io/docs
- **Node** : https://nodejs.org
- **Bootstrap** : https://getbootstrap.com/docs/4.0/getting-started/introduction/
- **Typescript** : https://www.typescriptlang.org/docs/home.html