Présentation d'Angular

Sorti en 2009 et développé par Google,**AngularJS** est l'un des premiers véritables frameworks Javascript (si on considère JQuery comme une simple bibliothèque).

En 2016, sortait **Angular 2** et avec lui une foule de nouveautés et de changements le rendant **incompatible avec son prédécesseur**. Depuis 2016, ce dernier connait un cycle de développement très rapide et une nouvelle version sort environ tous les 6 mois sans apporter trop de bouleversements de sorte qu'on préfère aujourd'hui parler du framework "Angular" sans préciser de numéro de version.

C'est de ce framework que nous parlerons dans ce chapitre.

Même pour un programmeur expérimenté, Angular est un outil difficile à maîtriser. Il demande une très bonne compréhension de l'orienté objet, de l'Ecmascript et des frameworks en général pour être appréhendé dans sa globalité. Si vous débutez en Javascript, il est probable que vous ne compreniez pas tout : ce n'est pas grave. L'intérêt de ce cours est de saisir l'utilité et le principe. Vous pourrez toujours revenir sur le détail plus tard

I. Angular : un framework Javascript client

Dans cette partie, nous discuterons des avantages d'un framework Javascript côté client. Si vous voulez passer directement à l'implémentation, sautez directement vers la partie II. Néanmoins, je vous conseille vivement de lire ce qui suit.

1) Un framework avant tout

a. Un parmi d'autres

Angular est donc un framework Javascript. Si vous n'êtes pas familier avec la notion de framework, je vous conseille de vous renseigner sur le sujet. Jetez par exemple un coup d'oeil au cours dédié même si les frameworks **client** tels qu'Angular diffèrent quelque peu dans leur philosophie des frameworks serveur.

Angular se situe dans la famille des frameworks client, contrairement à NodeJs qui est un framework plutôt orienté serveur. Angular n'est pas le seul dans sa catégorie. Parmi les stars du moment, citons **React** (développé par Facebook) ainsi que**VueJs**. Tous deux sont très matures et largement utilisés. Mais ils faut savoir qu'il en existe beaucoup d'autres :

- EmberJs
- Backbone
- Aurelia

Et bien d'autres !

b. Un framework client

Nous avons parlé de framework client. Cette appellation signifie que l'objectif d'Angular -et c'est aussi celui de React et de Vue- est de fournir un cadre de travail pour la manipulation du HTML et du CSS d'une page. Il ne **couvre pas** du tout tout ce qui lecture, création ou modification des **données** d'une application. Cette partie est déléguée à ce qu'on appelle des API avec lesquelles Angular va devoir dialoguer.

Angular est un framework côté client dont le champ d'action se limite à la manipulation de HTML et de CSS. L'accès aux données doit se faire en dialoguant avec des API externes.

c. Raison d'être principale

La question de l'intérêt d'un tel framework se pose à tous les programmeurs frontend. Il n'est pas forcément évident de distinguer les avantages d'Angular par rapport à d'autres outils tels que JQuery.

JQuery et ses équivalents sont des bibliothèques, c'est-à-dire une série d'outils qu'on est libre d'utiliser ou pas, et qui n'imposent pas une manière de travailler. L'avantage est une plus grande liberté structurelle, le désavantage est un investissement en temps plus important pour obtenir une organisation de code de qualité.

Angular quant à lui est un framework, c'est-à-dire qu'il impose au programmeur une organisation dans le flux de travail ce qui va avec des contraintes mais oriente très fortement le développement vers du code de qualité (bien qu'il soit toujours possible de coder n'importe comment avec Angular, rassurez-vous).

Cela dit, l'usage d'un framework client tel qu'Angular ne doit pas être systématique, comme c'est le cas de n'importe quel outil. Examinez bien les contraintes de votre projet car son usage peut parfois être davantage un fardeau qu'une aide !

2) Concept et avantages des frameworks client

Dans la paragraphe précédent, nous avons rappelé l'intérêt de travailler avec un framework tel

qu'Angular mais les remarques que nous avons faites auraient pu s'appliquer à n'importe quel framework. Voyons maintenant les avantages spécifiques aux frameworks client.

a. Problèmes posés par l'approche traditionnelle

Aux débuts du Web, surfer signifiait passer d'une page à une autre au moyen de lien hypertextes. Cela impliquait que la plupart des interactions d'un utilisateur supposait le rechargement de l'**intégralité** du code d'une page. Cette approche que l'on peut qualifier de synchrone ne posait pas problème dans le contexte d'un Web peu dynamique (où l'utilisateur était peu souvent amené à interagir avec la page).

Néanmoins, le Web s'est grandement dynamisé, à tel point qu'on parle aujourd'hui d'application Web plutôt que de site Web. Les interactions demandant le chargement de nouveaux contenus sont extrêmement fréquentes ce qui rend l'approche traditionnelle et synchrone beaucoup moins pertinente, voire caduque. Un exemple illustrant bien ce problème est celui des fenêtre de discussion en temps réel ou tchat : il n'y a pas de sens à rafraichir l'intégralité du contenu d'une page pour simplement afficher une nouvelle réponse dans une discussion.

b. Le 100% synchrone est mort, longue vie à l'asynchrone !

L'introduction d'Ajax a permis aux programmeurs de charger des nouveaux contenus dans une page indépendamment du cycle de rafraichissement des pages. Cela dit, dans la plupart des sites n'utilisant pas de frameworks client, c'est toujours le cycle classique synchrone qui prédomine dans la navigation, l'asynchrone n'étant réservé qu'à quelques cas particuliers où l'approche synchrone est difficilement envisageable.

c. Les frameworks client et le 100% asynchrone

L'idée de la plupart des frameworks asynchrones est de rendre l'usage de l'asynchrone systématique (ou quasiment). Partant du constat qu'aujourd'hui le plupart des actions utilisateurs n'exigent pas un rechargement complet de la page, leur idée est de ne charger que le contenu qui **change**, en utilisant **Ajax**. Si la navigation entre les sites/applications reste synchrone, la navigation **au sein d'un même site** doit s'effectuer de façon asynchrone. Voilà le concept.

Cette approche asynchrone prend également du sens dans le cas d'application très gourmandes en ressources ou bien très souvent sollicitées, ou encore les deux ! La passage à l'asynchrone, en plus d'accélérer le chargement des données, permet leur allègement.

d. Autres débouchés : le multiplateforme

En vérité, hormis l'asynchrone, les avantages d'Angular et des frameworks Javascript ne sont pas forcément évidents. Il est tout à fait possible d'obtenir une organisation de code très satisfaisante et d'être productif sans passer par ces frameworks, en se contentant par exemple de combiner un framework serveur (par exemple Symfony ou Laravel en PHP) avec JQuery ou même du pur Javascript. Beaucoup de projets uniquement axés Web se contentent très bien de cette approche, l'asynchrone étant mis en place uniquement quand c'est strictement nécessaire.

Mais un autre avantage de ces frameworks est qu'ils peuvent être intégrés très facilement dans le contexte d'**applications progressives** et multiplateformes, ce qui n'est pas possible avec l'approche dont nous venons de parler.

3) L'approche d'Angular

a. À fond dans la modularité

En plus de privilégier l'asynchrone, Angular met l'accent sur la modularité. Cela signifie que le code de toute application Angular doit être découpé en briques réutilisables et, dans la mesure du possible, indépendantes les unes des autres, dans le respect, ci-possible, des principes SOLID.

L'objectif est de renforcer la **ré-utilisabilité** du code, d'**éviter sa duplication**, pour garder un projet **organisé** et **maintenable**.

Pour cela, Angular introduit 2 concepts fondamentaux :

- Les **composants** (components)
- Les **modules**

D'autres concepts sont également importants tels que les**services**, les **directives** et les **pipes** mais ne sont pas essentiels à la compréhension de l'idée générale.

b. Les composants

Avant d'expliquer ce que sont les composants, voyons le problème qu'ils sont chargés de résoudre.

Dans un site classique, les pages sont stylisées et dynamisées par des fichiers CSS et Javascript **globaux**, c'est-à-dire s'exécutant sur toute la page. Cette approche pose aux programmeurs pas mal de problèmes dont voici un exemple.

Supposons un page toute simple dont voici le html :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <main>
            <button>Afficher/masquer formulaire</button>
            <form>
                <label for="name">Nom</label>
                <button type="submit">S'enregistrer</button>
            </form>
        </main>
        <footer></footer>
        <script
              src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
              integrity="sha256-3edrmyuQ0w65f8gfBsqowzjJe2iM6n0nKciPUp8y+7E="
              crossorigin="anonymous"></script>
        <script type="text/javascript" src="script.js"></script>
    </body>
</html>
```

Ce script en plus d'utiliser JQuery, inclut la fiche CSS suivante :

```
form {
    display: none;
}

.displayed {
    display: initial;
}
```

Ainsi que le script suivant :

```
$(function(){
    $('main button').click(function(e) {
        e.preventDefault();
        $('form').toggleClass('displayed');
    });
});
```

Cette page propose un bouton affichant un formulaire permettant de s'enregistrer sous un certain nom.

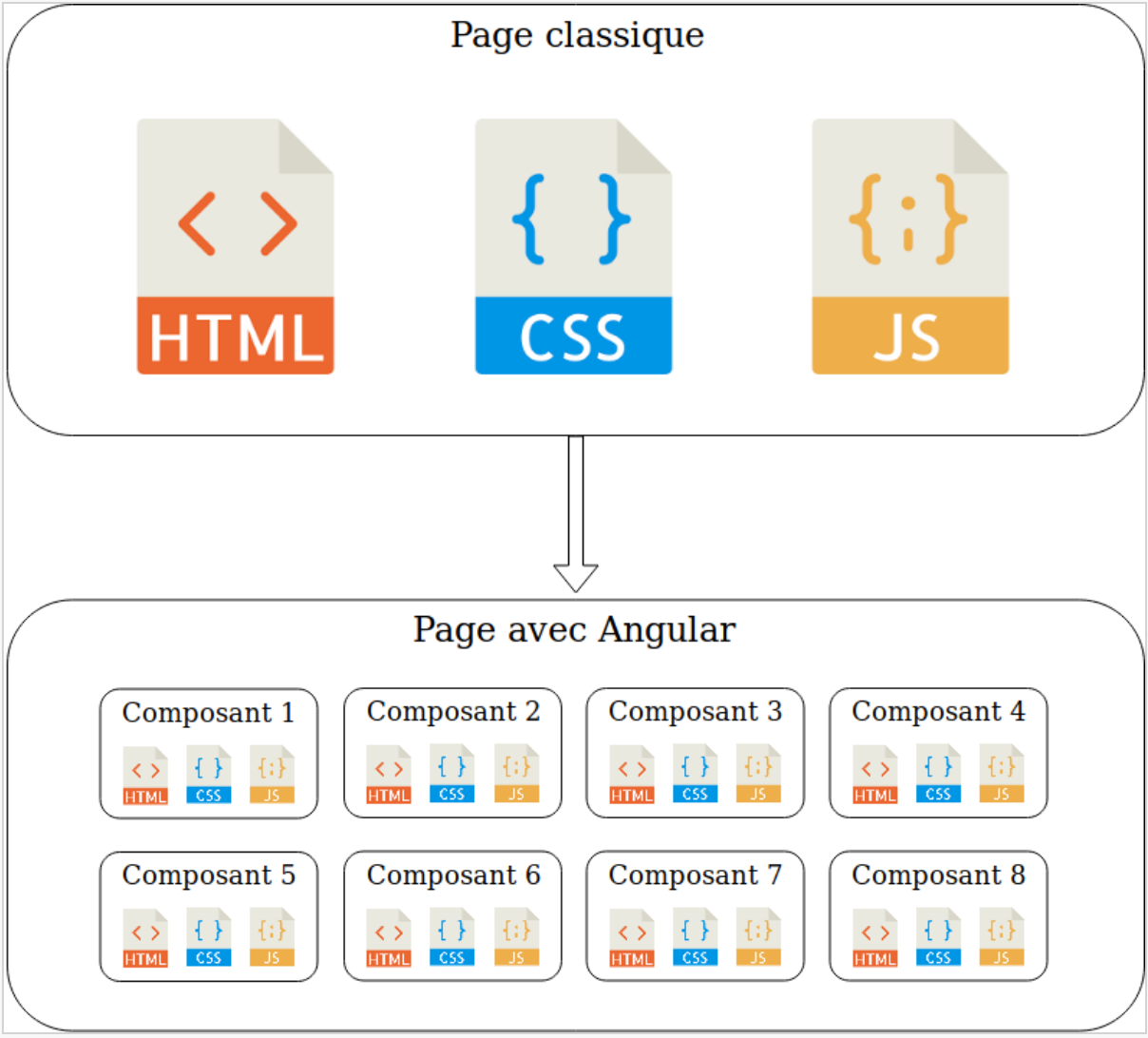
Cependant, les CSS et JS de cette page ont un problème : si je clique sur le bouton "Afficher/masquer le formulaire", le formulaire s'affiche mais si je clique ensuite sur le bouton "S'enregistrer", mon formulaire disparaît et n'est pas envoyé. Cela est dû au mauvais sélecteur `main button` de mon script, qu'il faut remplacer par `main > button`

Cependant, d'autre problèmes surviendront quand je continuerai le développement de mon site. Par exemple, si je souhaite placer un formulaire dans la balise footer, celui-ci ne s'affichera pas à cause de la règle CSS mise en place. Il faudra modifier cette règle ou en ajouter une pour que tout fonctionne, en créant un sélecteur plus précis ou en inventant un nom de classe. Ensuite ce sera le bouton "Afficher/masquer le formulaire" qui ne fonctionnera plus et qui demandera à corriger mon script etc. etc.

Comme on le voit avec cet exemple simple, le fait que mes scripts js et css soient inclus de façon globale pose des problèmes de maintenance (pas insolubles, certes, mais difficiles à résoudre). Le programmeurs doit avoir en tête l'intégralité du code de son site pour être sûr de ne pas faire de bêtise (créer une règle qui n'en casse pas une autre, utiliser un nom de classe qui n'existe pas déjà quelque part etc.)

Ce problème est encore plus critique dans le cadre d'une application ou toute la navigation est faite en Ajax et où n'importe quelle portion de code peut être incluse n'importe où.

C'est à ce genre de problème que répondent les composants (components) d'Angular. Ceux-ci regroupent de **courtes portions de HTML, de CSS et de Javascript** (de Typescript en vérité) **isolées les unes des autres**. Elles permettent simplement au programmeur de dire : "J'inclus dans ma page un code html qui est motorisé par tel script js et stylisé par tel CSS, le reste de ma page ne doit pas être impacté par ce que je viens de faire". Les scripts js et css ne sont plus globaux mais **locaux** et ne concernent plus qu'un court **segment** de ma page.



Par conséquent une page, dans Angular, doit être vue comme l'assemblage de plusieurs segments HTML autonomes car motorisés chacuns par des CSS et des JS différents.

Ces composants ont en plus la particularité d'être réutilisables à plusieurs endroits d'une même application et même -ce qui les rend très intéressants- d'applications différentes !

c. Les modules

En plus des composants, nous avons les modules. L'idée des modules est de **regrouper** les **composants** mais également les directives, les services et les pipes en **ensembles cohérents**. Cette logique de regroupement peut être une logique métier, thématique, organisationnelle ou purement technique.

Par exemple, on peut souhaiter créer un module pour tout ce qui touche à nos utilisateurs (logique métier), ou tout ce qui concerne l'affichage automatisé de données dans un tableau (logique technique).

Si le module contient des composants réutilisables dans d'autres contextes que l'application en elle-même, ils peuvent être considérés comme des bibliothèques et utilisés à l'intérieur d'autres projets ce qui les rend particulièrement intéressants.

II. Installation d'Angular

1) Prérequis : npm et angular cli

a. npm

Npm (Node package manager) est le gestionnaire de paquets de référence de Javascript. Angular étant un paquet Javascript, il est logique d'utiliser npm pour l'installer. Npm a été conçu à l'origine pour travailler avec NodeJs, lequel sera installé en même temps que npm. Installez tout cela en vous rendant ici.

b. Angular cli

Comme vous le verrez par la suite, Angular est un framework excessivement **verbeux**. Comprenez que pour obtenir ne serait-ce qu'un début de résultat, il faut écrire un grand nombre de lignes de code et de configuration. Heureusement, un outil nous permet de créer automatiquement des fichiers **génériques** dans lesquels nous pourrons écrire le code propre à notre application. Cet outil s'appelle Angular CLI (Angular Command Line Interface) et est quasiment indispensable aussi quand on développe en Angular.

Angular CLI est un outil qui s'utilise en ligne de commande et nous en ferons grand usage dans la suite de cours. Pour le suivre correctement, il vous faudra maîtriser au moins les rudiments de la ligne de commande.

Pour l'installer, le plus simple est d'ouvrir une console (terminal Windows, Linux, Mac, Cmder, git bash etc. peu importe) et de saisir la commande suivante :

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

Cette installation nous donne accès à une commande `ng` de Angular CLI que nous utiliserons très souvent.

2) Notre première application

a. Installation

Pour ce cours, nous nous baserons sur le tutoriel de référence que vous pouvez retrouver sur le site d'Angular. Dans ce projet très simple, nous entretenons une liste de noms associés à des ids (nos héros). Nous pourrons ajouter des héros, les modifier ou encore les supprimer.

Ouvrez une console, rendez-vous dans votre dossier de développement favori avec `cd` et installez Angular à l'aide de la commande suivante :

```
ng new tour-of-heroes
```

Répondez "y" à la première question et choisissez "css" pour la deuxième.

Cette commande vous permettra d'installer tous les fichiers nécessaires à l'utilisation d'Angular dans un dossier *tour-of-heroes*, ainsi qu'une **base de code déjà fonctionnelle** pour notre application. Il ne reste plus qu'à la développer !

Ouvrez le dossier tour-of-heroes avec l'éditeur de votre choix. Il en existe plusieurs qui conviennent très bien : Sublime Text, Visual Studio Code, Bracket, Atom etc.

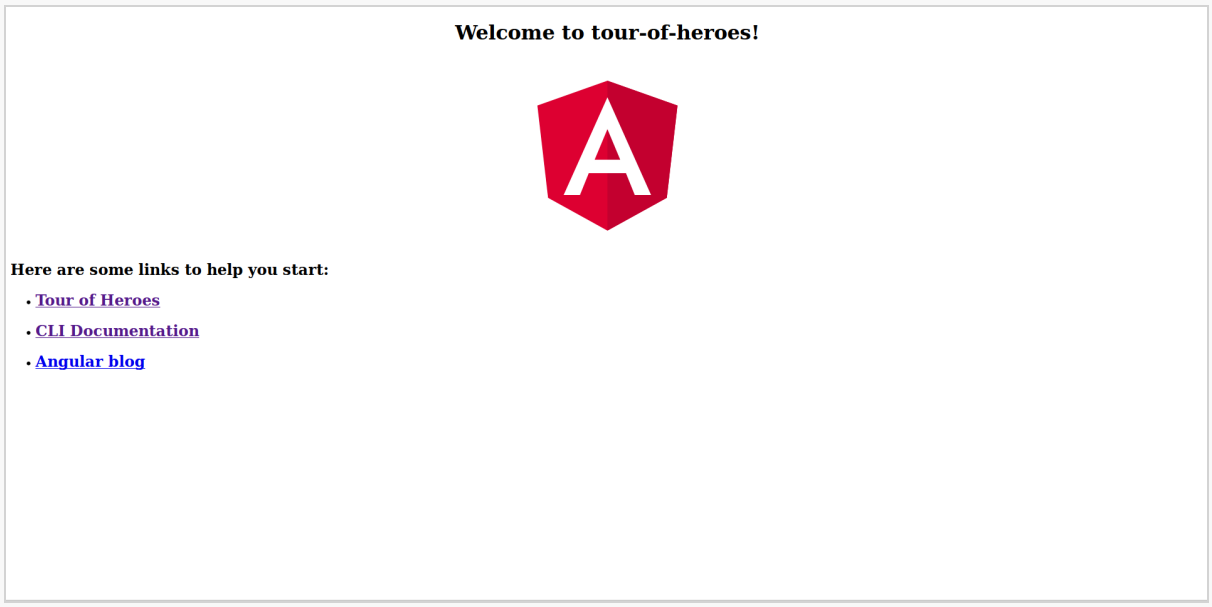
Comme vous pouvez le constater, Angular CLI a crée une structure qui compte déjà un très grand nombre de fichiers. Ne vous laissez pas impressionner, nous verrons en détail cette structure un peu plus loin.

Pour lancer votre application et voir le résultat dans votre navigateur, rendez-vous dans votre

dossier applicatif puis lancez le serveur de développement :

```
cd tour-of-heroes
ng serve --open
```

Votre navigateur devrait s'ouvrir avec la seule et unique page de votre application :



Bien que le contenu de cette page soit très intéressant, notamment les trois liens situés à gauche que je vous conseille de visiter, ce n'est pas notre application. Remplaçons le contenu de cette page en nous rendant dans le fichier *src/app/app.component.html*. Vous constaterez que ce fichier ne contient pas une structure html complète mais uniquement le *contenu* de notre page.

Remplacer le contenu de src/app/app.component.html par ce qui suit :

```
<h1>Bienvenue sur {{ title }}</h1>
```

Votre page devrait se rafraichir immédiatement et vous afficher :

Une question que vous vous posez peut-être : comment se fait-il que `{{ title }}` soit remplacé par "tour-of-heroes". Comment peut-on changer cela ?

Nous verrons en détail comment cela fonctionne mais vous pouvez déjà vous rendre dans le fichier *src/app/app.component.ts* dans lequel vous trouverez cette ligne :

```
  title = 'tour-of-heroes';
```

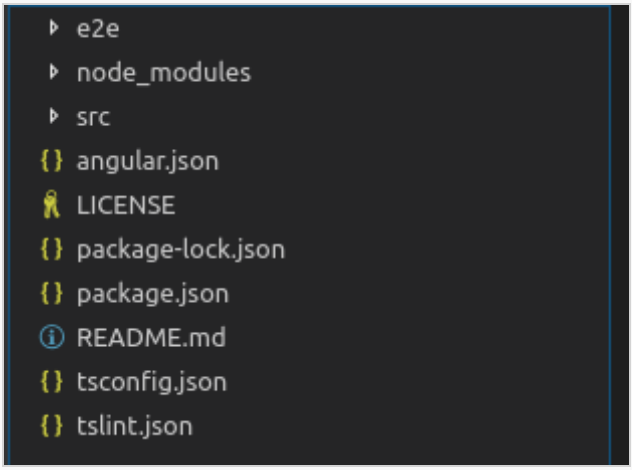
Croyez-le ou non, mais ce *title* est le même que celui que nous retrouvons dans notre html. Remplacez `'tour-of-heroes'` par `'La tour des héros'`. Le résultat dans le navigateur doit être celui escompté :

Bienvenue sur La tour des héros

Bien, maintenant que nous avons un peu joué avec notre contenu, examinons notre application plus en profondeur.

b. Structure

Comme on l'a vu, la structure de base d'une application Angular générée par Angular CLI est déjà bien fournie en fichiers. Voyons un peu comment tout cela s'organise.



- Si vous connaissez npm, vous êtes déjà familier avec**package.json**, **package-lock.json** et **node_modules** : package.json donne les directives concernant les paquets utilisés par notre projet ("je veux tel paquet avec au moins cette version") et est éditable par nous, package-lock.json contient les versions de tous les paquets effectivement installés ainsi que de leur dépendances et n'est, comme son nom l'indique, pas édité par nous mais par npm lui-même. Quant à node_modules, il contient les paquets en eux-mêmes.

- Le fichier **License** est optionnel et donne les conditions d'utilisation de votre projet.

- Le fichier **README.md** a été généré par Angulat CLI. Il doit contenir toutes les informations que vous jugez utiles pour présenter votre projet au reste du monde (utilité, but, informations d'installation et de prise en main, liens documentaires etc.)

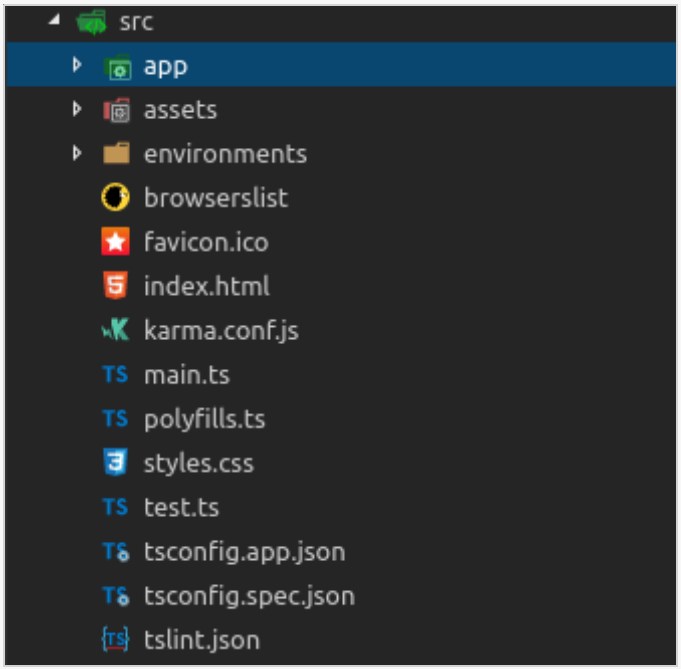
- **tsconfig.json** et **tslintjson** concernent *Typescript* qui est la variante de Javascript avec laquelle vous coderez en Angular. Typescript n'est pas encore compatible avec tous les navigateurs et doit être converti (*transpilé*) en Javascript pour être compris. Le fichier tsconfig.json a pour rôle principal de configurer cette conversion. tslint.json sert quant à lui à spécifier les règles de bon codage qui s'appliquent au Typescript écrit dans ce projet.

- **angular.json** contient toute la configuration du ou des projets Angular présents dans ce dossier. En particulier, il spécifie comment un projet Angular doit être construit, lancé ou déployé.

- **e2e/** est le dossier contenant les tests fonctionnels (ou **end to end**) du projet. Ce sont les tests concrets de votre application qui vérifient qu'elle affiche bien les bonnes choses au bon moment.

- **src/** enfin contient tout le code source de votre application ainsi que ses assets. C'est là que vous développez véritablement votre application (votre nouveau dossier préféré !).

Détaillons un peu le contenu de notre dossier**src/** :



- **browserlist** est utilisé pour spécifier la liste des navigateurs actuellement supportés par l'application.
- **favicon.ico** : l'icône de notre application
- **index.html** : le point d'entrée HTML de notre application. En Angular, il y a deux points d'entrée : le point d'entrée HTML et le point d'entrée Javascript (ou plutôt Typescript).
- **karma.conf.js** : le fichier de configuration des tests Karma
- **main.ts** : le point d'entrée Typescript. C'est en réalité le seul vrai point d'entrée de l'application, celui qui enclenche tout.
- **polyfills.ts** : permet d'importer du code pour étendre le support de votre application Angular à des navigateurs plus anciens.
- **styles.css** : contient les styles globaux de votre application. Eh oui, même si Angular a une approche composant par composant, vous pouvez tout de même (et heureusement !) inclure quelques styles globaux.
- **test.ts** : définit le contexte des tests.
- **tsconfig.app.json** : vient ajouter au tsconfig.json du dossier parent quelques éléments propres à l'application
- **tsconfig.spec.json** : pareil mais uniquement pour les tests.
- **tslint.json** : vient étendre le tslint.json du parent pour les besoin de cette appplication.
- **environments/** : un dossier contenant deux fichiers permettant de définir différentes variables d'environnement en fonction du mode (développement ou mise en ligne)

Comme vous le voyez, Angular n'a pas peur de vous faire peur :)

Cela dit, ne prêtez pas à ces fichiers trop d'attention pour l'instant. Dites-vous qu'il permettent juste à Angular de fonctionner. Deux dossiers plus intéressants sont les suivants :

- **app/** : Contient les éléments du composant**app** qui est le composant parent de toute l'application.

- **assets/** : Contient les assets (en général les images et les vidéos) propres à votre application.

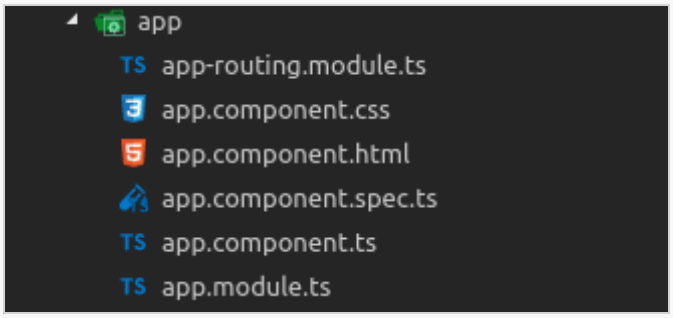
III. Fonctionnement d'Angular

Dans un prochain chapitre, nous construirons pas à pas notre application. Pour l'instant, amusons-nous avec notre seule page pour voir comment Angular fonctionne dans l'essentiel.

1) Prise en main de notre composant

a. Trois fichiers

Précédemment, nous avons vu que notre application ne comptait pour l'instant qu'un seul composant. Nous avons aussi vu qu'un composant était la combinaison d'un code Typescript, d'un template HTML et éventuellement de fiches de styles (CSS ou SASS ou encore LESS). Rendons-nous dans notre dossier *src/app/* afin d'examiner notre premier composant :



Quoi, cinq fichiers pour un seul composant ?

Non, en réalité, seulement quatre, voire**trois** si l'on omet le fichier app.component.spec.ts qui n'est là que pour les tests. Trois fichiers donc, et cela fait sens puisque un composant est un assemblage de trois aspects (Typescript, HTML et CSS). Notez que tous les fichiers qui servent à définir un composant ont un **.component** précédant l'extension.

Plus généralement, la règle de nommage de fichier en Angular est systématiquement la suivante :

[nom].[role].[extension]

Ainsi, nous avons aussi un module pour notre application qui s'appelle**app.module.ts**. Plus tard nous créerons des services dont le nom ressemblera à **user.service.ts** et pourquoi pas des directives dont le nom ressemblera à **highlight.directive.ts**.

b. Le Typescript pour les gouverner tous

Parmi ces trois aspects, le plus important -celui qui va commander les autres- est le Typescript. C'est de lui que tout part. Cette approche est peut-être nouvelle pour vous si vous avez l'habitude du système classique dans lequel tout part du HTML. Voyons un peu ce que dit notre Typescript en examinant le contenu de app.component.ts

Bien qu'ils soient de la même famille, Typescript et Javascript (Ecamscript 5) diffèrent beaucoup. Expliquer TYpescript justifierait à lui seul l'écriture de plusieurs cours. L'objectif n'est pas ici de rentrer dans le détail de Typescript mais d'en donner un aperçu pratique.

Voici le contenu :

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

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

Ça a l'air compliqué, n'est-ce-pas ? Découpons tout cela pour y voir plus clair en nous concentrant sur le code de fin :

```
export class AppComponent {
  title = 'La tour des héros';
}
```

Essentiellement, notre composant est donc une **classe**, c'est à dire une structure comprenant des **données** (les champs) et des **méthodes** (des fonctions) qui manipulent les données. C'est un résumé très sommaire et incomplet mais qui va nous suffire pour l'instant.

Ici par exemple, title est un champs qui contient notre titre.

Néanmoins, le fait d'être une classe exportée ne suffit pas à faire de AppComponent un composant. Pour faire de notre classe un composant, nous devons la **décorer** grâce à une **annotation**. Cette annotation vient se placer juste au dessus de la déclaration de notre classe :

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

Les annotations Typescript sont annoncées par un *@* suivi du nom de l'annotation. Ici, nous avons bien affaire à l'annotation **Component** qui prend en paramètre un **objet de configuration**. Dans cet objet sont précisés les 3 éléments fondamentaux d'un composant : un **sélecteur**, un **template** HTML et une (ou plusieurs) feuille(s) de **style** CSS (ou SASS ou encore LESS!).

Le fait que le template HTML et les feuilles de styles soient déclarées dans le script ne doit pas vous étonner : c'est bien le Typescript qui dirige tout et qui déclare tout, le reste (HTML, CSS) suit. Notre composant déclare donc son template et son style.

Reste à voir à quoi correspond le sélecteur. Pour le comprendre vous devez remonter d'un cran dans la hiérarchie de votre projet, direction le fichier *src/index.html* :

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>TourOfHeroes</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Dans ce fichier, nous avons une balise que vous ne connaissez pas, et moi non plus, puisque il ne s'agit pas d'une balise HTML. Cette balise est une balise **inventée** dont le but est d'être remplacée par notre composant app. Ce qui permet à cette balise d'être remplacée par notre composant est justement la propriété *selector* de notre composant que nous avons fixé à `'app-root'`. Nous aurions pu remplacer `<app-root></app-root>` par une balise existante telle que `<div id="app-root"></div>` à condition de changer la valeur de la propriété selector en `'#app-root'` mais cela est considéré comme une mauvaise pratique. On préfère, par convention, utiliser des balises personnalisées telles que *app-root*. Plus généralement, vous écrirez souvent vos balises en les **préfixant** en fonction du **module** auxquelles elles appartiennent : app-root, app-users, app-user-details etc.

Notez enfin que pour pouvoir utiliser les annotations, comme pour n'importe quoi d'autre en Typescript, vous devez les importer. Ici nous importons Component depuis @angular/core dès la première ligne.

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

2) Lien entre le template HTML et le script Typescript

L'un des aspects les plus intéressants d'Angular (et consorts) est le **lien direct** qui existe entre la **classe** du script du composant et son **template** HTML. C'est ce qui rend les frameworks Javascript particulièrement attrayants pour un développeur. Voyons cela en détail

a. Lien script vers template

Un aspect très sympathique d'Angular est la facilité avec laquelle une classe de composant transmet ses informations au HTML. En effet, n'importe quel **champs public** de notre classe est directement disponible dans notre template. Si nous mettons à jour les champs dans notre classe, ces changements seront aussitôt et automatiquement **transmis** vers notre template.

Cela explique pourquoi l'expression `{{ title }}` de app.component.html est effectivement remplacée par sa valeur dans notre classe, qui est 'La tour des héros'.

Il est également possible de contrôler les attributs de notre élément comme leur style. Ajoutons par exemple un champs `titleStyle` dans notre classe :

```
export class AppComponent {
  title = 'La tour des 1 héros';
  titleStyle = {'font-family' : 'Arial', 'font-size' : '2rem'};
```

Ce style peut être transmis dans le titre de notre template de la manière suivante :

```
<h1 [ngStyle]="titleStyle">Bienvenue sur {{ title }}</h1>
```

Notez la présence des crochets qui exprime un lien du script vers le template. Le style de notre titre est bien mis à jour :

Cela, ainsi que d'autres éléments que nous verrons plus loin, permet à notre classe de **contrôler** intégralement et facilement le contenu du template qui lui est associé.

b. Lien template vers script

Si le script contrôle le template, le template peut lui aussi solliciter le script à l'aide d'**évènements**. Les évènements HTML natifs comme "click", "keydown" ou "mousemove" sont par exemple disponibles et permettent de déclencher du code s'exécutant dans le contexte de notre classe de script.

Pour illustrer cet aspect, ajoutons une variable heroCount dans notre script app.component.ts juste après la première ligne :

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

let heroCount = 2;

// ...
```

Dans notre classe, changeons le title en `'La tour des 1 héros'` ajoutons une méthode qui incrémente cette valeur et la concatène avec notre title.

```
export class AppComponent {
  title = 'La tour des 1 héros';
  titleStyle = {'font-family' : 'Arial', 'font-size' : '2rem'};
  addHero() {
    this.title = 'La tour des ' + (heroCount++) + ' héros';
  }
}
```

Dans notre template, nous ferons en sorte d'appeler cette méthode lors du clic sur un bouton. Ajoutons ce bouton à app.component.html :

```
<h1 [ngStyle]="titleStyle">Bienvenue sur {{ title }}</h1>

<button (click)="addHero()">Ajouter un héros</button>
```

Remarquez la syntaxe bizarre `(click)="addHero()"` laquelle n'est pas comprise par le navigateur mais bien par Angular. Les parenthèses entourent le nom de l'évènement qui, une fois déclenché, exécutera le code compris entre les double quote.

Ici, c'est notre clic qui déclenchera la méthode addHero() de notre classe.

Notez que ce code s'exécute dans le contexte de la classe du composant.

Si vous rafraichissez votre page et cliquez sur le bouton, vous voyez le nombre de héros augmenter

c. Encore plus fort : lien bidirectionnel

Nous avons vu comment lier notre script à notre template grâce aux champs public de notre classe. Nous avons vu comment effectuer le lien inverse à l'aide des évènements. Ce type de liaison est appelé unidirectionnel (**one way binding**). Il est possible de faire encore plus fort en mettant en place un lien bidirectionnel (**two way binding**) : un évènement peut être déclenché, modifier les données de la classe qui seront ensuite restituée au template de façon automatique !

Pour le mettre en évidence, ajoutons un nom d'utilisateur dans notre classe :

```
export class AppComponent {
  title = 'La tour des 1 héros';
  titleStyle = {'font-family' : 'Arial', 'font-size' : '2rem'};
  username = 'Anonymous';
  // ...
```

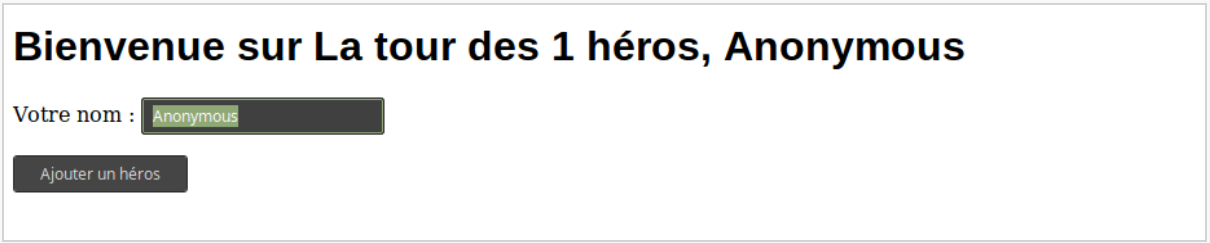
Et maintenant, dans notre template, nous ajoutons ce nom à notre titre et nous créons un champs texte permettant d'éditer ce nom :

```
<h1 [ngStyle]="titleStyle">Bienvenue sur {{ title }}, {{ username }}</h1>

<p>
    <label for="InputUsername">Votre nom <input id="InputUsername" type="text" name="na
me" [(ngModel)]="username"></label>
</p>

<!-- ... -->
```

Ce qui donne :



de même que le couple `[]` exprime un lien script -> template et `()` un lien template -> script. L'association des deux `[()]` exprime un lien bidirectionnel script <-> template.

3) Les directives structurelles : conditions et boucles

Un autre aspect essentiel d'Angular est la possibilité de mettre en place des conditions et des boucles dans notre template. Ces éléments, appelés directives structurelles permettent de renforcer encore le contrôle que le script exerce sur le template.

a. Boucles

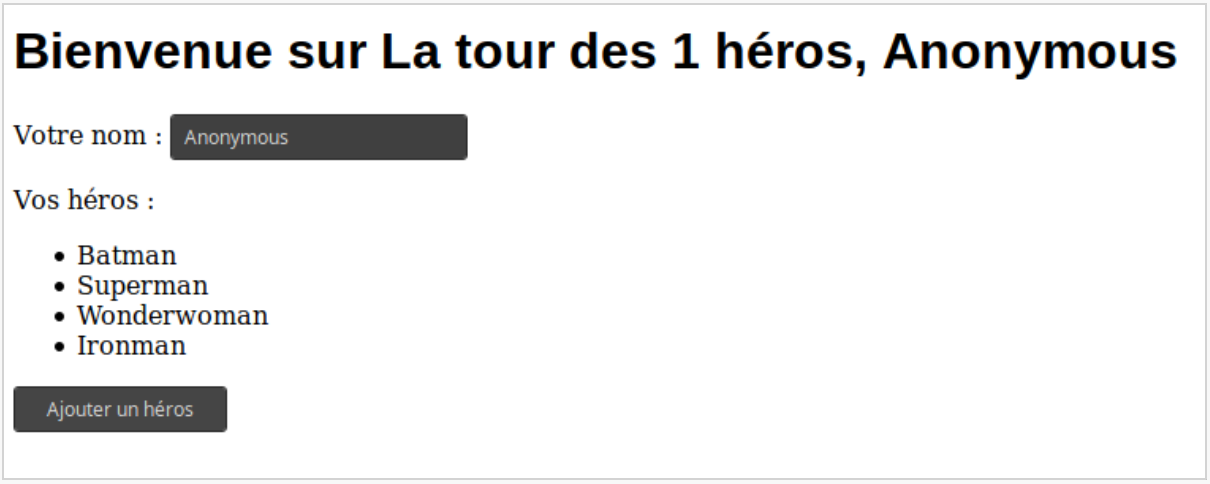
Voyons cela en ajoutant dans notre script une liste de noms de héros :

```
export class AppComponent {
  title = 'La tour des 1 héros';
  titleStyle = {'font-family' : 'Arial', 'font-size' : '2rem'};
  username = 'Anonymous';
  heroes = [
    'Batman',
    'Superman',
    'Wonderwoman',
    'Ironman'
  ];
```

L'idée va être d'afficher cette liste de héros dans notre template. Pour cela, nous allons parcourir notre tableau à l'aide d'une boucle for. En Angular, nous utilisons la directive structurelle *ngFor. Voici la forme que cela prend :

```
Vos héros :
<ul>
    <li *ngFor="let hero of heroes">{{ hero }}</li>
</ul>
```

Notre directive structurelle s'applique à chacun des éléments de notre liste et est annoncé par un *. Nous décomposons nos `heroes` en `hero` que nous affichons dans la liste. Voilà la résultat :



Ce qui est génial, c'est que nous pouvons ajouter/supprimer des éléments dans notre tableau et que ces changements seront automatiquement reflétés dans notre template. Pour s'en convaincre, plaçons un input contenant un nom de héros à ajouter à côté de notre bouton "Ajouter un héros". Dans notre script, je vais aussi ajouter un champs heroName qui sera relié à cet input en **two way binding** !

```
// ...
heroes = [
    'Batman',
    'Superman',
    'Wonderwoman',
    'Ironman'
  ];
heroToAdd = '';
//...
```

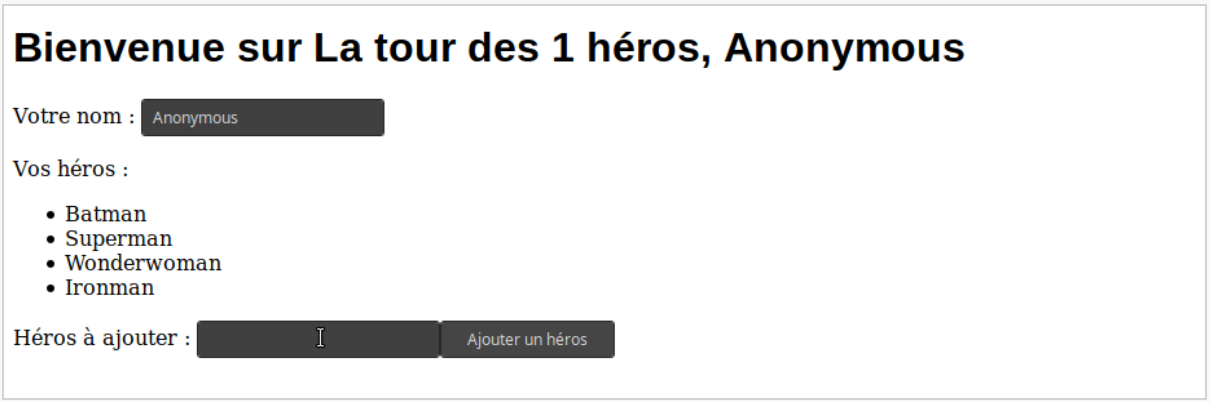
et :

```
<!-- ... -->
<label for="InputHeroName">Héros à ajouter : <input type="text" name="nameOfTheHero" id
="InputHeroName" [(ngModel)]="heroToAdd"></label>
<button (click)="addHero()">Ajouter un héros</button>
```

Le clic sur le bouton ne doit plus incrémenter la valeur de heroCount (qui d'ailleurs ne sert plus à rien) mais ajouter `heroToAdd` à notre tableau. La valeur affichée dans le titre doit quant à elle correspondre à la longueur du tableau :

```
addHero() {
  this.heroes.push(this.heroToAdd);
  this.title = 'La tour des ' + this.heroes.length + ' héros';
}
```

Ce qui donne :



Parfait, notre liste se met bien à jour. Seulement, il nous reste un petit problème, qui est que notre titre n'est pas initialisé correctement. Pour résoudre le problème, le plus simple est de ne pas passer par une variable title. et d'afficher directement la longueur du tableau de héros dans le template. Voyez plutôt :

```
/*
On désactive notre titre
title = 'La tour des 1 héros';
*/
titleStyle = {'font-family' : 'Arial', 'font-size' : '2rem'};
```

et :

```
<h1 [ngStyle]="titleStyle">Bienvenue dans la tour des {{ heroes.length }} héros, {{ user
name }}</h1>
```

Ce qui fonctionne mieux !

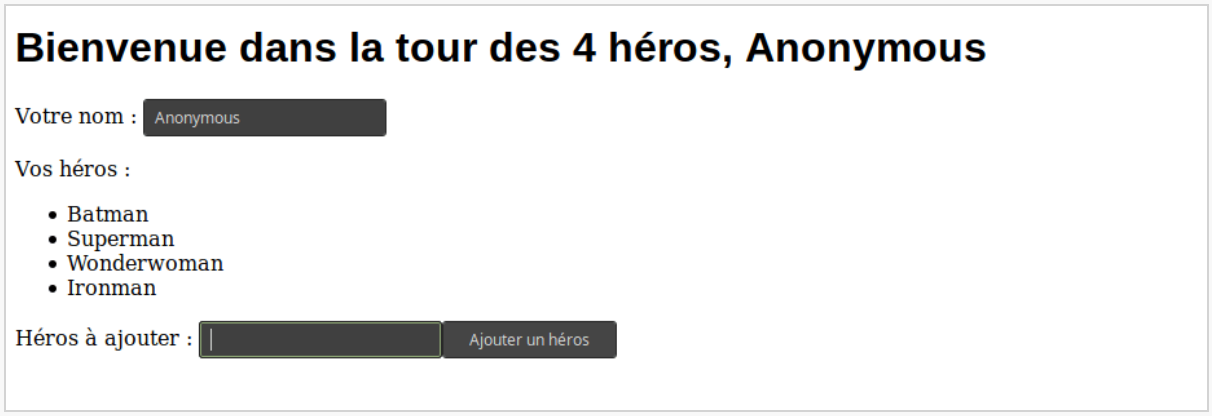
b. Conditions

Il existe évidemment des directives structurelles pour les conditions. Ainsi nous pouvons afficher un morceau de HTML si telle condition est remplie, sinon faire autre chose. Par exemple, ajoutons à notre liste un petit message si le nom du héros ajouter correspond à notre nom d'utilisateur.

```
Vos héros :
<ul>
    <li *ngFor="let hero of heroes">{{ hero }} <small *ngIf="hero == username">(c'est m
oi :))</small></li>
</ul>
```

Sans grande surprise, la directive à employer est *ngIf, elle prend comme valeur la condition correspondante !

Voilà le résultat :



Voilà pour les bases concernant les directives structurelles. Sachez qu'il existe aussi des structures **ngSwitch**.

IV. Conclusion

Dans ce cours, nous avons vu les bases vous permettant d'appréhender le fonctionnement d'Angular ainsi que son intérêt. Ce n'est évidemment pas suffisant pour pouvoir l'utiliser dans un cas réel. Pour comprendre comment construire réellement une application dans Angular, voir le chapitre consacré. Notez que les sources de l'application à ce stade d'évolution sont disponibles sur github (il faudra vous placer sur la branche "play-with-app").