Angular

La figure de proue d'une nouvelle génération de frameworks web

Angular

The leader of a new generation of web frameworks

par **Pierre POMPIDOR**

Maître de conférences en informatique

Université de Montpellier / LIRMM (Laboratoire d'Informatique de Robotique et de Micro-électronique de Montpellier)

Résumé : Dans le domaine des cadres logiciels (frameworks) permettant la création d'applications web, Angular est la figure de proue d'une nouvelle génération combinant la mise en œuvre de services web et de composants. Ainsi grâce à Angular, le navigateur charge l'essentiel de la structure de l'application lors de sa première connexion au serveur, les transactions suivantes n'opérant que d'efficaces mises à jours. Le second point essentiel est la création de composants, qui avec les modules, permettent non seulement une très grande modularité et réutilisabilité des codes, mais représentent aussi une puissante extension du langage HTML. Ces deux aspects sont illustrés via quelques aperçus de codes d'un exemple de site marchand.

Abstract : In the field of software frameworks for creating web applications, Angular is the leader of a new generation combining the implementation of web services and components. Thus, thanks to Angular, the browser loads the entire structure of the application when it first connects to the server, with subsequent transactions only performing effective updates. The second essential point is the creation of components, which together with the modules, not only allow a very high modularity and reusability of the codes, but also represent a powerful extension of the HTML language. These two aspects are illustrated via some code previews of an example of a merchant site.

Mots-clés :

Angular, services web, application monopage, composants

Keywords :

Angular, web services, SPA, components

1 Introduction

Angular est un cadre logiciel (ou *framework*) créé par Google en 2016 [2, 3]. Il permet de développer des applications web qui sont des applications informatiques s'affichant dans un navigateur, et exploitant des programmes et des données hébergés sur des machines distantes (nommées serveurs). Si un framework laisse toute liberté au développeur d'implémenter les fonctionnalités qu'il désire, il impose une structuration précise des différents programmes à réaliser. Pour sa part Angular met en œuvre deux grandes avancées du génie logiciel : l'utilisation de **services web**, et en conséquence la création d'applications dites monopages, et la mise en œuvre de **composants**.

Cet article se propose donc d'expliquer le fonctionnement et les bénéfices des services

web et des composants, illustrés par la création d'un embryon d'application qui permettrait, l'authentification d'un utilisateur faite, de visualiser la liste des produits en vente sur un site marchand.

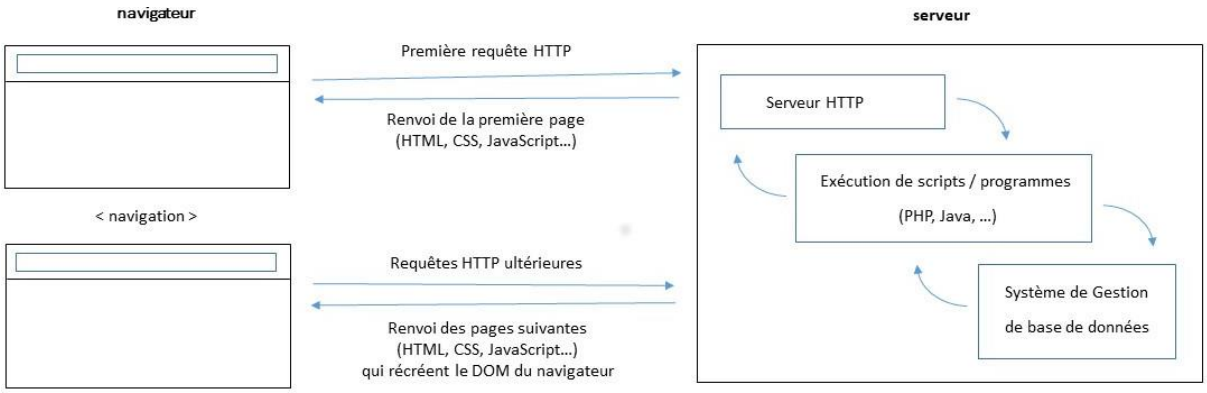
Il est important de préciser que cet article concerne Angular (à ce jour de version 8) et non AngularJS une première version, devenue obsolète, du framework de Google.

2 Application monopage versus application multipages

2.1 Architecture des applications web traditionnelles multipages

Les applications web (dites en anglais *web app*) traditionnelles sont fondées sur une architecture client-serveur dans laquelle le serveur délivre des **pages web** au navigateur en fonction de la navigation effectuée par l'internaute.
Ces pages web sont fondamentalement structurées grâce à des **balises HTML** qui mettent en page les informations qui y sont présentées, des styles CSS pour factoriser des styles graphiques entre différents éléments des pages, et enfin des codes JavaScript pour gérer du côté client - c'est-à-dire au sein du navigateur - certaines actions qui n'exigent pas du serveur le renvoi d'une nouvelle page (par exemple contrôler la validité de certains champs d'un formulaire avant sa soumission). Une petite présentation du langage HTML et de son traitement par le navigateur est présentée dans le chapitre suivant.
Il est aussi à noter qu'une évolution maintenant ancienne, désignée par l'acronyme AJAX (Asynchronous Javascript And XML), permet également de récupérer d'un serveur – de celui qui a délivré la page ou d'un serveur tiers - des données qui permettent de mettre à jour une page sans modifier sa morphologie (un exemple classique est de faire apparaître le nom d'une ville suite à la spécification par l'internaute de son adresse postale).

Figure 1 - Architecture d'une application web traditionnelle « multipages »



Dans cette architecture traditionnelle, chaque chargement d'une page implique l'allocation en mémoire du navigateur du modèle objet du document (désigné par l'acronyme anglais DOM) ce qui est très lourd. Cette notion est expliquée un peu plus loin dans cet article avec celle du langage HTML.

Il faut noter que dans notre schéma la machine distante (le serveur) regroupe à la fois le serveur HTTP (l'application qui réceptionne les requêtes HTTP et invoque les traitements à effectuer), l'évaluateur qui exécute les programmes, ainsi que le système de gestion de bases de données. Bien entendu d'autres configurations sont aussi possibles : par exemple le système de gestion de bases de données est très

fréquemment délocalisé sur un serveur dédié.

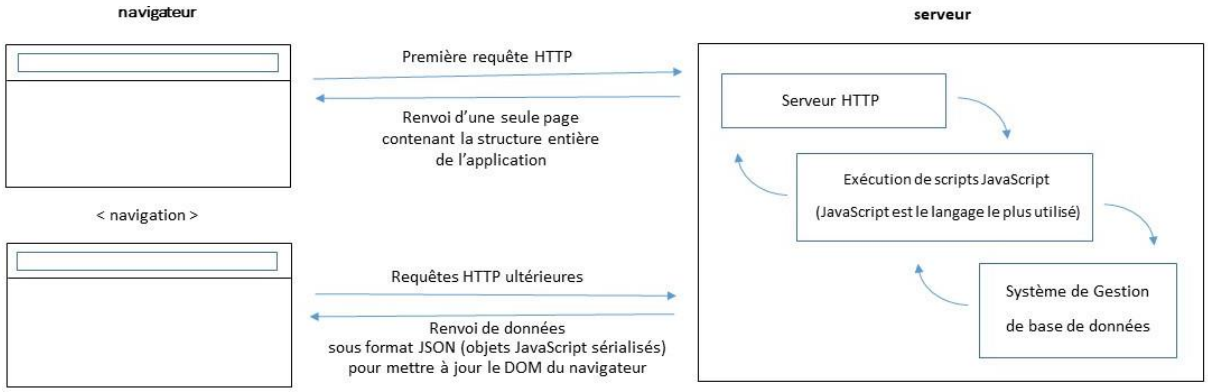
2.2 Architecture des applications web monopages

Les applications monopages, dont le framework Angular est non seulement le géniteur le plus abouti mais aussi le précurseur (avec sa première version nommée AngularJS), révolutionne l'architecture qui a été décrite précédemment.
La structure de l'application web - et donc toutes les balises HTML qui la composent - est chargée avant toute navigation, ce qui implique que seules les données à exploiter ne sont ultérieurement transmises du serveur au client.
Cette architecture présente deux avantages majeurs :

- une fois la page « unique » chargée, la navigation est beaucoup plus rapide ;
- les données (ou codes supplémentaires) peuvent être aisément acquis de différents serveurs ce qui permet une grande flexibilité dans l'architecture générale de l'application.

Il est à noter que le format le plus usité d'échange de données entre le serveur et le navigateur est la sérialisation d'objets JavaScript nommé JSON (JavaScript Object Notation). Un exemple de cette sérialisation est donné dans le chapitre suivant.

Figure 2 - Architecture d'une application monopage



Dans cette architecture la structure complète de l'application côté client - du côté du navigateur - est chargée lors de la première transaction, seules des mises à jour extrêmement plus rapides que le chargement de pages sont ultérieurement effectuées.

2.3 Positionnement d'Angular parmi d'autres frameworks web

Le transfert de données (et non de pages) entre le serveur et le client étant fondamentalement géré par le langage de programmation JavaScript via AJAX, il est naturel que les principaux frameworks permettant la création d'applications monopages - ou les bibliothèques participant à la création de celles-ci - soient basés sur JavaScript (ou sur des extensions à ce langage).
Hormis Angular dont l'explication des principes est le but de cet article, une bibliothèque et un framework très populaires doivent être cités : React et Vue.js.
En complément, nous dirons un mot sur des frameworks multipages qu'il est possible de dévoyer pour les coupler avec d'autres outils pour créer des applications monopages.

React (ou React.js) est une bibliothèque, et non un framework à proprement parler,

car elle ne permet de gérer que l'interface (la « vue » dans le paradigme de conception MVC) des applications web. Développée par Facebook depuis 2013, React permet de créer des interfaces composées de multiples briques (des composants) directement en JavaScript (ou plus exactement dans une extension à ce langage nommée JSX). Avec JSX React n'utilise donc pas le langage HTML (qui sera introduit dans le paragraphe 4.1), ou des codes dédiés à la génération d'éléments du langage HTML (les templates, dont des exemples en Angular sont donnés au paragraphe 4.2).
La flexibilité et la très grande réactivité des interfaces ainsi produites a conféré une belle notoriété à React, et il est à noter que depuis 2015 Google propose la bibliothèque Polymer dont le positionnement est similaire à celui de React.

Vue.js est un framework JavaScript développé depuis 2014 (et initié par des contributeurs chinois) qui permet à l'instar d'Angular de créer des applications web monopages côté client. Il met également en œuvre des composants logiciels qui sont manipulables comme des extensions au langage HTML (ce point sera illustré dans le paragraphe 4 dans le cadre d'Angular). Il doit donc être vu comme un concurrent direct à Angular, mais présente les avantages et inconvénients d'une architecture plus flexible, le cadre de création des composants et des modules étant beaucoup moins dirigiste ce qui induit un workflow de données entre composants bien moins balisé. Cette flexibilité est un atout pour la création rapide d'applications web, mais présente des limitations lors de la mise en place d'une importante intégration de composants s'échangeant de multiples données (ce point sera esquissé dans le paragraphe consacré à la bibliothèque Angular NgRx qui présente la solution la plus avancée à la mise à jour de données entre différents composants).

Enfin, un mot doit être dit des frameworks web multipages basés soit sur le langage de programmation serveur PHP avec comme représentants principaux Laravel et Symfony, soit sur le langage de programmation Python avec Django. Leur utilisation pour créer des applications web monopages, en les couplant avec des bibliothèques comme React ou Polymer côté client, est possible, mais dénature de fait leur essence même, et ne présente guère d'intérêt sinon de conserver du code déjà développé (ce qui est néanmoins un critère pragmatique mais important pour les sociétés de développement informatique).

En résumé de ce court comparatif, React n'a pas le même périmètre d'action qu'Angular qui permet de créer des applications web en intégrant toute leur logique métier côté client (hormis le stockage pérenne et le filtrage des données qui sont l'apanage du côté serveur).
Vue.js est un concurrent direct mais moins adapté à la création d'applications complexes intégrant un nombre conséquent de composants échangeant des données. La grande efficacité du chargement des codes de Vue.js (les codes empaquetés, ou encore « bundlelisés », c'est-à-dire mis à plats pour être transmis au navigateur), est d'ailleurs dorénavant concurrencé par le nouveau moteur de rendu d'Angular nommé Ivy disponible à partir de sa version 8. Ivy est à la fois un transpiler, (ou transcompilateur mais ce terme est peu usité), qui traduit les codes TypeScript en codes JavaScript, (TypeScript étant l'extension syntaxique de JavaScript utilisée par Angular), et aussi un empaqueteur qui transmet les codes « mis à plat » et intégrant ceux des modules au navigateur.

Enfin, utiliser un framework web multipages avec une bibliothèque de composants web telles que React ou Polymer ne peut se justifier que pour des raisons extra-fonctionnelles (savoir-faire de la société informatique, réutilisation de codes pour des raisons économiques…).

3 Du côté serveur, les services web

3.1 Les services web

Un service web est un agent logiciel qui permet l'échange de données entre deux applications informatiques connectées à Internet. Ce paragraphe décrit rapidement la mise en place de services web pour qu'ils puissent être utilisés par Angular.

Pour illustrer notre article, nous considérons que les services web sont gérés par un serveur **Node.js** connecté à un système de gestion de bases de données NoSQL **MongoDB**.
Un serveur Node.js est une plateforme JavaScript orientée réseau permettant une grande efficacité, et donc très populaire comme serveur HTTP (ou pour gérer des services de messageries instantanées, des jeux via des connexions IP…).
MongoDB est un système de gestion de bases de données NoSQL orienté documents. Des collections stockent de manière flexible des documents qui sont dans un format binaire l'équivalent d'objets JavaScript.
Le choix d'un telle architecture dénommée **MEAN** (MongoDB, Express, Angular er Node.js) est d'autant plus pertinent que tout le développement est dans ce cas sous-tendu par le langage JavaScript (ou ses extensions) et qu'en conséquence toutes les données manipulées sont des objets JavaScript.

3.2 Des données hébergées dans une base de données NoSQL

Soit le fichier *produits.json* qui regroupe dans une liste, et sous un format textuel, cinq objets JavaScript concernant la vente de compléments alimentaires :

```
[
    {"nom":"Caféine", "type":"anti-fatigue", "prix":"15"},
    {"nom":"Curcumine", "type":"anti-oxydant", "prix":"30"},
    {"nom":"Quercétine", "type":"anti-oxydant", "prix":"50"},
    {"nom":"Resvératrol", "type":"anti-oxydant", "prix":"45"},
    {"nom":"Rhodiola rosea", "type":"anti-fatigue", "prix":"25"}
]
```

La commande mongoimport installée avec le SGBD MongoDB permet de créer la collection *produits* dans la base de données *SUPERVENTES* :

```
mongoimport --db SUPERVENTES --collection produits --file produits.json
--jsonArray --drop
```

3.3 Un exemple de serveur Node.js

Voici un exemple de code JavaScript qui évalué par Node.js permet l'accès via une requête HTTP à tous les documents de la collection *produits* contenus dans la base *SUPERVENTES* :

```
const express = require('express');
const app     = express();
const MongoClient = require('mongodb').MongoClient;
const ObjectID    = require('mongodb').ObjectId;
const url         = "mongodb://localhost:27017";
```

```
MongoClient.connect(url, {useNewUrlParser: true}, (err, client) => {
    let db = client.db("SUPERVENTES");
    app.get("/produits", (req,res) => {
        res.setHeader("Content-type", "application/json");
        res.setHeader("Access-Control-Allow-Origin", "*");
        db.collection("produits").find().toArray((err, documents) => {
             res.end(JSON.stringify(documents));
        });
    });
});

app.listen(8888);
```

Ainsi si nous considérons que le serveur Node.js s'exécute sur notre machine locale, (c'est-à-dire celle où s'exécute notre navigateur), l'URL suivante :

```
http://localhost :8888/produits
```

émet une requête http au serveur Node.js qui en réponse va transmettre au navigateur les objets cités précédemment.

Dans ce code, l'interrogation de la collection MongoDB *produits* est effectuée par la méthode *find()*. Cette fonction pourrait prendre en argument un objet qui préciserait les documents à sélectionner dans la collection via une sélection de propriétés et de leurs valeurs. Sans rentrer dans des détails trop abscons, nous noterons aussi que la requête HTTP utilise la méthode GET ce qui signifie que le client demande au serveur une lecture de données (d'autres méthodes telles que POST, PUT ou DELETE demanderaient la mise à jour, l'ajout ou la suppression de données).

4 Composants et extension du langage HTML

4.1 Le langage HTML

Revenons maintenant du côté client : la présentation des données dans la fenêtre d'un navigateur est organisée grâce à l'utilisation d'un langage de description nommé **HTML** (pour HyperText Markup Language). Ce langage, très facile d'accès, mobilise des duos de balises, une balise ouvrante et une balise fermante (et quelquefois des balises autofermantes), encadrant les données ou les autres balises sur lesquelles la directive est portée. Par exemple, un texte centré et cliquable, renvoyant sur la documentation d'Angular, peut être codé ainsi :

```
<center>
   <a href="https://angular.io/"> Documentation d'Angular </a>
</center>
```

Nous remarquons que la balise ouvrante *<a>* qui crée dans notre exemple une ancre sur un texte qui peut être sélectionné (un *hypertexte)*, est accompagnée d'un attribut *href*. La majorité des balises sont ainsi enrichies d'attributs qui modifient leurs comportements.

Par ailleurs, quand du code HTML (une page) est transmis au navigateur, celui-ci va utiliser un outillage spécifique pour allouer en mémoire de l'ordinateur les différentes entités de ce code (balises, attributs informations textuelles…). Cet outillage est constitué de classes au sens de la programmation par objets, (ce point ne sera pas détaillé dans cet article), et est nommé le Document Object Model (le DOM). Par

extension le DOM désigne également toutes les données mises en mémoire et associées à une page affichée dans une fenêtre du navigateur. Dans une architecture traditionnelle, la navigation de l'internaute sur un site web implique donc l'envoi par le serveur de nouvelles pages constituées de balises HTML, chaque réception d'une page nécessitant la recréation du DOM ce qui est une opération très lourde. Dans une architecture basée sur la création d'applications monopages, la création du DOM est faite initialement (en tout cas pour sa partie correspondante aux balises et à leurs attributs), la navigation ultérieure ne provoquant que des mises à jour du DOM ce qui beaucoup plus efficace.

4.2 Les composants

La seconde et principale évolution des frameworks web que met magistralement en œuvre Angular, est l'utilisation de **composants**, cette évolution ayant un impact direct sur le langage HTML.
Un composant logiciel est une brique logicielle qui peut, telle quelle, être mise en œuvre, ses valeurs en entrée (qui peuvent moduler son fonctionnement), et ses valeurs de sortie (qui peuvent être utilisées par le composant qui l'a mis en œuvre), étant donc directement connectables aux autres éléments de l'application.
L'utilisation de composants (et aussi celle des modules qui seront évoqués dans la paragraphe suivant), permet une réutilisation optimisée des codes informatiques.

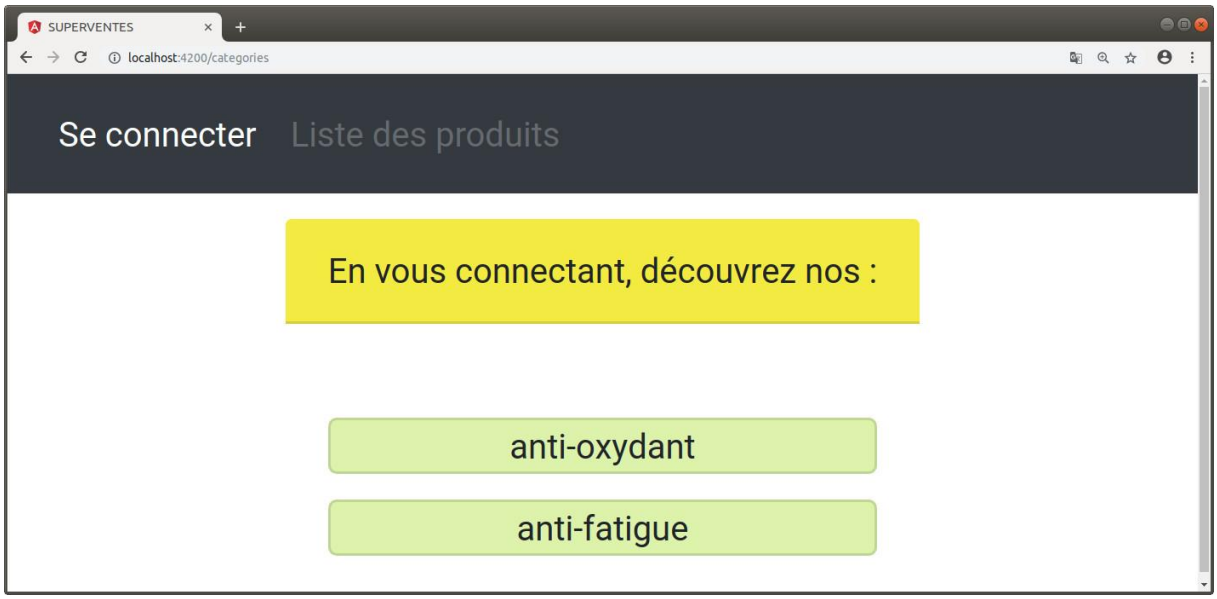
Angular met en oeuvre des **smart components** qui associent trois entités :

- les données manipulées par le composant (les variables) et les traitements associés à ceux-ci (par exemple la liste des produits à vendre et un traitement permettant de récupérer cette liste du serveur). Ces données et ces traitements sont regroupés dans un programme écrit en TypeScript qui est une extension au langage JavaScript permettant la mise en œuvre d'une programmation typée et orientée objets avec classes ;
- l'interface graphique (ou *template*) de ce composant structurée par des balises HTML ;
- et enfin l'ensemble des styles graphiques permettant de préciser l'affichage graphique des éléments de cette interface.

Dans un premier temps, nous allons nous intéresser aux interfaces graphiques avant de voir un exemple de code TypeScript.

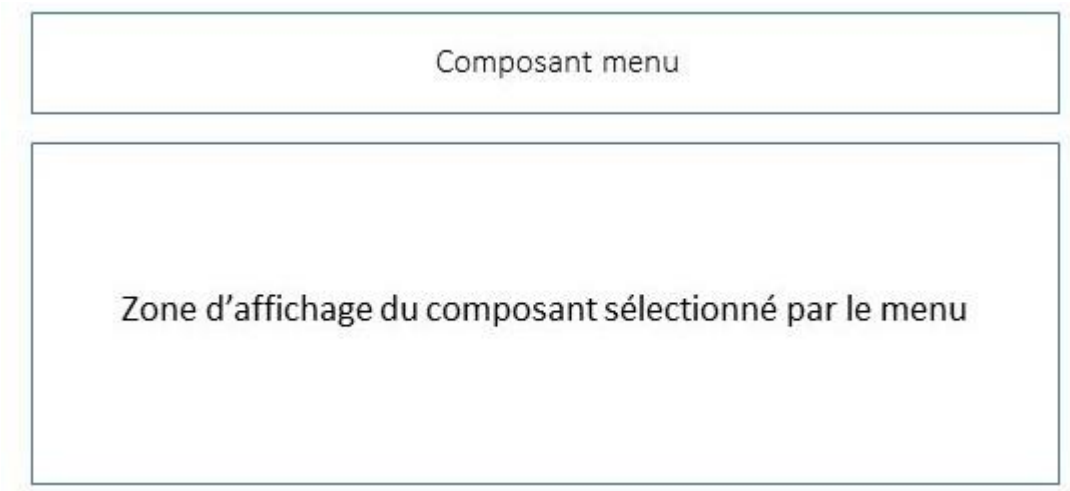
Ainsi, prenons tout d'abord comme exemple la page d'accueil de notre application Angular d'illustration (le lien sur le dépôt GiHub de celle-ci est donné en fin d'article) :

Figure 3 - Page d'accueil de notre application d'exemple



Le template de notre composant principal est structuré par l'inclusion du composant créant le menu et d'une zone d'affichage dynamique (qui exhibe ici les catégories).

Figure 4 - Structure du composant principal



Voici les balises HTML qui implémentent ce template :

```
<app-menu></app-menu>
<router-outlet></router-outlet>
```

La balise <*app-menu*> désigne le composant qui gère le menu. Ce composant est donc inséré grâce à la balise le représentant (son *sélecteur*) dans le template de notre composant principal. La balise <*app-menu*> étend donc le langage HTML.

Le template du composant *menu* pourrait être écrit ainsi (nous verrons bientôt qu'il est en fait beaucoup plus complexe, car une fois l'internaute connecté, son contenu doit évoluer) :

```
<ul>
```

```
    <li> <a routerLink="/membres/connexion"> Se connecter </a> </li>
    <li> <a class="disabled" routerLink="/produits">
              Liste des produits
         </a>
    </li>
</ul>
```

Les balises <*ul*> et <*li*> permettent de structurer une liste et ses items.

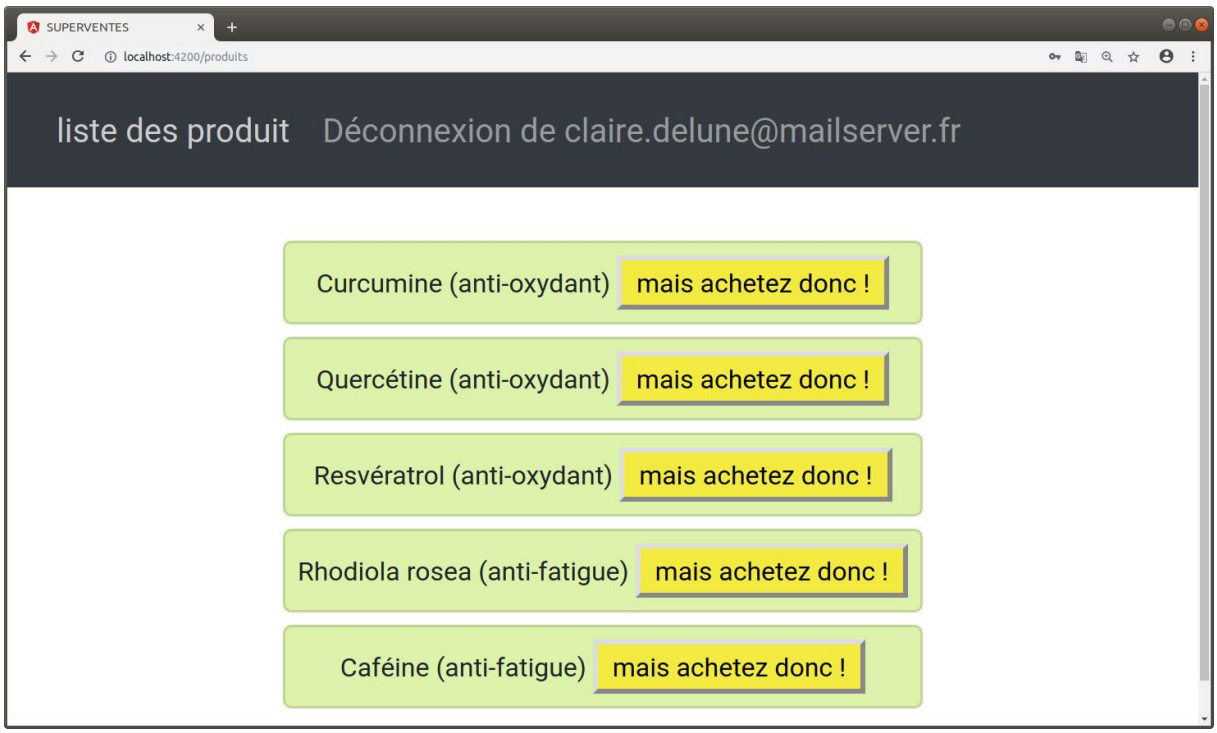
La balise <*router-outlet*> désigne la zone dans laquelle l'affichage de différents composants est dynamiquement effectué suite à la sélection d'une action. Dans notre exemple, cette zone va par la suite permettre l'affichage des compléments alimentaires. Ce point est expliqué dans le paragraphe suivant.

Enfin nous devons noter que nos exemples de code ont été simplifiés, différents styles CSS étant également mis en œuvre pour faire apparaître le menu horizontal tel que la capture d'écran le restitue.

4.3 Extensions des attributs HTML

Revenons sur l'aspect de notre application telle que la figure 3 l'a exhibé, et imaginons que l'internaute se soit identifié, (soit en créant un nouveau compte, soit en s'authentifiant), en sélectionnant le lien « Se connecter ». Le menu doit alors évoluer en rendant actif le lien sur la liste des produits en vente, mais aussi en proposant un lien de déconnexion en remplacement de celui de connexion.
Dans notre contexte d'application monopage, il faut donc que la structure du menu évolue mais cela sans recharger de nouvelle page du serveur.

Figure 5 - Page d'accueil de notre application d'exemple, l'internaute s'étant connecté, puis ayant sélectionné la liste des produits



Un moyen simple de gérer la dualité du menu est de prévoir ses deux contenus possibles grâce à deux listes HTML, et de conditionner l'activation de la liste

concernée à un état de l'application (ici l'authentification de l'internaute).
Dans l'exemple de code qui suit, cet état de l'application correspondra à la valeur de la variable *user* : si celle-ci contient l'adresse email de l'internaute, cela signifie que l'internaute s'est déjà authentifié, et la fonction *identification()* avec *user* en paramètre renverra « vrai » (et bien sûr « faux » dans le cas contraire).

```
<ul *ngIf="!identification(user)">
   <li>
       <a routerLink="/membres/connexion"> Se connecter </a>
    </li>
    <li>
       <a class="disabled" routerLink="/produits">
           Liste des produits
       </a>
    </li>
</ul>
<ul *ngIf="identification(user)">
    <li>
       <a routerLink="/produits"> liste des produit </a>
    </li>
    <li>
       <a (click)="deconnexion()"> Déconnexion de {{user}} </a>
    </li>
</ul>
```

Nous notons la présence de l'attribut **ngIf* accompagnant la balise *<ul>*. Cet attribut représente également une extension au langage HTML proposée par Angular. Il permet de conditionner l'activation d'une balise. Dans notre exemple la condition à valider pour que le menu active le lien permettant l'affichage des compléments alimentaires, et un lien de déconnexion, est que la variable *user* ait une valeur (en fait l'adresse email de l'internaute s'étant authentifié grâce à celle-ci).
Nous noterons aussi la présence d'un écouteur d'événement géré directement par Angular qui déclenche l'exécution d'une fonction JavaScript, ainsi que l'affichage de la valeur de la variable *user* gérée par le code TypeScript du composant.

Angular propose bien d'autres extensions que **ngIf*. Par exemple dans le code suivant du template du composant qui affiche la liste des compléments alimentaires, la directive structurelle **ngFor* génère autant de balise * * que de produits en encadrant le nom du produit, son type, et un bouton qui permettrait d'en acheter (mais qui en l'état n'est pas associé à une action).

Voici le code de *produits.component.html* :

```
<div *ngFor="let produit of produits”>
     {{produit['nom']}} ({{produit['type']}})
     <button> mais achetez donc ! </button>
</div>
```

Dans ce code, la variable *produit* a pour valeur chaque objet de la liste d'objets *produits* (une liste d'objets est nommée une collection). Un objet regroupe les valeurs de différentes propriétés : dans notre exemple nous affichons les valeurs des propriétés *nom* et *type*.

Sans rentrer dans les arcanes du code, jetons un coup d'oeil au code TypeScript gérant le composant *produits* contenu dans le fichier *produits.component.ts*. Il gère la liste *produits* en interrogeant le serveur pour lui donner un contenu.

```
import { Component, OnInit } from '@angular/core';
import { ProduitsService } from '../produits.service';

@Component({
  selector: 'app-produits',
  templateUrl: './produits.component.html',
  styleUrls: ['./produits.component.css']
})
export class ProduitsComponent implements OnInit {
    private produits: Object[] = new Array();
    constructor(private produitsService: ProduitsService) { }
    ngOnInit() {
        this.produitsService.getProduits().subscribe(produits => {
            this.produits = produits;
        });
    }
}
```

L'analyse de code nous montre que le composant peut être intégré dans un template via la balise *<app-produits>* (c'est son sélecteur), et que son template est contenu dans le fichier *produits.component.html*.
Par ailleurs ce code donne un exemple d'injection de dépendance qui autorise différents composants à « partager » l'instanciation d'un autre composant (appelé service). Ici c'est le cas du service *ProduitsService* programmé dans le fichier *produits.service.ts*. Ce service utilise une facette de la programmation réactive qui lui permet de s'abonner (*subscribe*) à un flux de données. Cet aspect sera un peu plus détaillé dans le paragraphe 6.

4.4 Le référencement d'applications Angular

Le référencement d'applications générant du HTML via des templates évalués du côté client pose problème car la quasi-totalité des données textuelles sont injectées dynamiquement via des codes JavaScript exécutés au niveau du navigateur. Et ainsi les informations qui ne dépendent pas de la navigation de l'internaute, (le titre d'onglet, les entêtes…), ne peuvent pas être répertoriées dans les codes sources du serveur par un moteur de recherche.
Il est à noter que cette problématique dépasse le périmètre d'Angular pour impacter non seulement les frameworks permettant la création d'applications monopages, mais plus généralement tout ceux utilisant un système de templating qui crée dynamiquement du code HTML.

Dans le cadre d'un projet Angular géré par Angular CLI, ce problème d'optimisatioon pour les moteurs de recherche (SEO), est résolu par un module nommé *Angular Universal Toolkit* qui permet de pré-générer au niveau du serveur du code HTML incluant les informations textuelles stables (par exemple le contenu de la balise *<title>* et biens d'autres). Cet article n'entrant pas dans les détails de l'utilisation de ce module, un lien sur une présentation didactique de sa mise en œuvre est donné dans la webographie en fin de celui-ci.

4.5 Modules versus composants

Après avoir découvert le concept de composant web (« web component »), nous

abordons un second point essentiel à la modularité, et donc à la réutilisabilité des codes développés avec Angular : celui-ci concerne les **modules**.

En effet, outre la structuration de l'application en composants, la modularité d'Angular est encore renforcée via la création et la ré-exploitation de modules. Lors de la création d'une application - ce point étant abordé dans le paragraphe 7 - un premier module (le *root module*) est automatiquement créé en spécifiant le composant de plus haut niveau de l'application, ainsi que l'importation des modules externes nécessaires à la création des codes JavaScript qui seront transmis au navigateur.
Des modules supplémentaires (les *feature modules*), regroupant des composants, peuvent être également créés : ils implémentent les grandes fonctionnalités de votre application qui seront réutilisées dans d'autres applications. Par exemple si vous développez, via plusieurs composants, la gestion des utilisateurs d'un site (connexion, authentification, déconnexion), il suffira de réutiliser le module ainsi créé.

Pour illustrer l'utilisation des modules, examinons la structure de celui qui, installé par défaut avec Angular, permet de mettre en œuvre des requêtes http.
Ce module se nomme *HttpClientModule* et, pour être utilisé, doit tout d'abord être référencé dans le fichier *app.module.ts* qui spécifie les éléments du root module de notre application Angular
Voici donc le squelette d'un fichier *app.module.ts* spécifiant l'utilisation du module *HttpClientModule* :

```
…
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [ <déclaration des composants> ],
  imports: [ HttpClientModule, <importation des autres modules> ],
  providers: [ <déclaration des services> ],
  bootstrap: [ <déclaration du composant de plus haut niveau> ]
})
export class AppModule { }
```

Ce paramétrage autorise les différentes ressources logicielles du module *HttpClientModule* à être utilisées dans les composants Angular (les services) qui se connecteront au serveur Node pour transférer des données.

Ainsi dans le service *ProduitsService* issu du fichier *produits.service.ts* donné en exemple ci-dessous, voici un exemple de mise en œuvre d'une des classes du module *HttpClientModule*. La classe *HttpClient* instancie (d'une manière très singulière comme nous le verrons juste après) un objet dont la méthode get() permet d'accéder aux données de notre serveur Node.js.

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

export class ProduitsService {
    constructor(private http: HttpClient) { }

    getProduits(): Observable<any> {
        return this.http.get(<url>);
    }
}
```

En utilisant le service *ProduitsService*, un composant pourra utiliser la méthode

getProduits() qui elle-même émettra une requête HTTP (grâce à la méthode *get()* de la classe *HttpClient*) au serveur Node, puis renverra de manière asynchrone (c'est-à-dire quand elles seront disponibles) les données reçues du serveur concernant la liste des produits en vente au composant l'ayant demandé. L'utilisation d'un observable en retour des méthodes *get()* et *getProduits()* est expliqué dans le paragraphe consacré à la programmation réactive.

Revenons maintenant sur l'instruction TypeScript **import** mise en œuvre dans les deux codes ci-avant. Elle permet donc :

- dans la description des dépendances d'un module d'une application Angular, d'importer toutes les ressources logicielles du module spécifié (nous allons voir que ce module correspond en fait à un dossier) ;
- dans un composant, d'importer une classe particulière d'un module.

Pour comprendre cette dualité, examinons maintenant comment le module *HttpClientModule* et la classe *HttpClient* s'offrent à notre application.

Le module est stocké avec les autres modules nécessaires à Angular dans un sous-dossier du dossier **node_modules** qui abrite tous les modules installés par **npm** le gestionnaire de modules. Le dossier *node_modules* peut être partagé par plusieurs projets, ou être spécifique à chaque application.

Traçons dans l'arborescence de *node_modules* le chemin qui aboutit au fichier client.d.ts contenant la classe *HttpClient* (les dossiers sont présentés en gras) :

```
node_modules
    @angular
          common
                http
                     package.json
                     http.d.ts
                     src
                         client.d.ts
                         … (autres fichiers du répertoire src)
                … (autres dossiers du répertoire http)
```

Cette arborescence a été un peu simplifiée, des fichiers supplémentaires (index.d.ts et public_api.d.ts) créant pour des raisons techniques de compatibilité des « rebonds » entre fichiers.

Présentons le contenu du fichier *http.d.ts* comme il pourrait être simplement écrit :

```
export { HttpBackend, HttpHandler } from './src/backend';
export { HttpClient } from './src/client';
…
```

Ce fichier autorise l'exportation de toutes les classes écrites dans les différents fichiers du répertoire *src* : il définit le contenu global du module *HttpClientModule*.

Examinons maintenant le contenu du fichier src/client.d.ts qui implémente différentes ressources dont la classe *HttpClient* :

```
…
export declare class HttpClient { … }
…
```

L'instruction **export**, duale d'*import*, permet d'exporter une classe d'un module afin qu'elle puisse être ré-utilisée.

Dans le cas ou plusieurs modules fourniraient des classes de même nom, il est possible de créer des espaces de noms (« namespaces »). Voici un exemple générique associant un espace de nom à toutes les classes d'un module :

```
import * as <espace de nom> from <lien sur le fichier>;
```

Un dernier point reste à éclaircir : comment est fait le lien entre l'importation d'un module par l'indication d'un dossier (ici le dossier *@angular/common/http*)

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

et le fichier qui liste toutes les classes de ce module (ici le fichier *http.d.ts*) ?

C'est le rôle du fichier *package.json* qui correspond à la sérialisation d'un objet JavaScript de configuration :

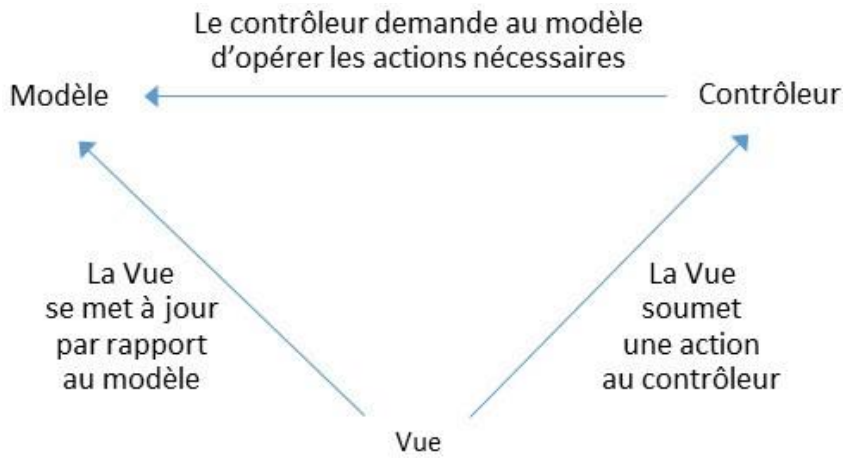
```
{
 "name": "@angular/common/http",
 "typings": "./http.d.ts",
  …
}
```

Cette architecture, un peu complexe à première vue, permet de normaliser la réutilisation des modules développés en TypeScript et destinés à être partagés par des applications Angular.

5 Mise en œuvre d'un routeur

Dans le paragraphe précédent nous avons découvert d'une part que les ancres contenues dans le template du composant *menu* portaient un attribut *routerLink*, et d'autre part, que le contenu d'une zone définie par la balise <*router-outlet*> changeait dynamiquement suivant les actions effectuées. Cet attribut et cette balise dévoilent la présence du **routeur** d'Angular qui est l'incarnation du **contrôleur** du modèle de conception logiciel Modèle-Vue-Contrôleur. Ce célèbre modèle instaure deux découplages qui permettent une meilleure évolution des applications informatiques :

- un découplage entre les données stockées par l'application et leur affichage ;
- un découplage entre l'expression des actions effectuées par l'utilisateur et les codes informatiques devant être mis en œuvre pour les satisfaire.



Dans notre exemple, quand l'utilisateur sélectionne l'ancre (le texte cliquable) « Se connecter », il envoie le message (la *route*) `/membres/connexion` au contrôleur qui doit alors sélectionner le composant à invoquer suite à cette directive.
Le même fonctionnement, mais un peu plus sophistiqué, est mis en œuvre avec la sélection de l'item « Liste des produits ». Le message `/produits` sera également envoyé au contrôleur mais celui-ci ne le prendra en compte que si l'utilisateur est déjà authentifié. Cela sera un composant spécial, un garde (ou *guard)*, qui gérera cette vérification et qui autorisera, ou pas, le contrôleur à invoquer le composant qui liste les produits en vente (la mise en œuvre des gardes n'est pas abordée dans cet article).

Les corrélations entre les messages et les composants à invoquer sont spécifiées dans une **table de routage**. Voici le contenu du fichier *app-routing.module.ts* de notre embryon d'application dans lequel apparaît cette table de routage :

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ConnexionComponent } from './connexion/connexion.component';
import { AuthGuardService } from './connexion/authguard.service';
import { ProduitsComponent } from './produits/produits.component';

const routes: Routes = [
  { path: 'membres/connexion', component: ConnexionComponent },
  { path: 'produits', component: ProduitsComponent,
           canActivate : [AuthGuardService]}
];

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

La table de routage est définie par la collection *routes*, chaque élément de cette collection étant un objet JavaScript dont les propriétés *path*, *component* et *canActivate* désignent respectivement le message que reçoit le contrôleur, le composant à invoquer et, éventuellement, le service chargé de vérifier si l'opération est permise. Dans notre exemple, les deux composants *connexion* et *produits* sont mis en œuvre par le routeur.

6 La programmation réactive et la bibliothèque NgRx

6.1 La programmation réactive et les observables

Angular intègre les principes du paradigme de la programmation réactive via l'utilisation de la bibliothèque RxJs. Initiée du patron de conception « observateur », la programmation réactive, telle qu'elle est implémentée dans cette bibliothèque, permet de scruter des flux de données (des observables), et à chaque fois qu'une donnée est reçue dans un flux, d'invoquer un traitement à effectuer.

Revenons sur la classe TypeScript gérant le composant *Produits* contenu dans le fichier *produits.component.ts*. Nous avons vu que le constructeur de cette classe créait une variable nommée *produitsService* de type *ProduitsService* :

```
…
```

```
import { ProduitsService } from '../produits.service';

export class ProduitsComponent implements OnInit {
    private produits: Object[] = new Array();

    constructor(private produitsService: ProduitsService) { }

    ngOnInit() {
        this.produitsService.getProduits().subscribe(produits => {
            this.produits = produits;
        });
    }
}
```

Dès sa création (dans la méthode *ngOnInit()*) ce composant s'abonne au flux de données renvoyé par la méthode *getProduits()* du service *produitsService* qu'il utilise. Ainsi dès que les informations sur les produits en vente sont renvoyés par la méthode *getProduits*() (qui elle-même a invoqué le service http d'Angular), ces informations sont recopiées dans la structure de données *produits* qui est utilisée dans le template associé à ce composant pour mettre à jour l'interface (voir Figure 5).

En relation avec le code précédent, voici un extrait du code de la classe *ProduitsService* qui met en œuvre la classe *HttpClient* pour établir une connexion avec le serveur Node.js et récupérer des données dans un observable :

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ProduitsService {

    private urlBase: string = <URL sur laquelle tourne Node >;
    constructor(private http: HttpClient) { }

    getProduits(): Observable<any> {
        return this.http.get(this.urlBase+'produits');
    }
    …
}
```

Les classes *HttpClient* et *ProduitsService* (qui sont également non associées à des templates) sont **injectables** : elles permettent d'être utilisées par une **injection de dépendances** via leurs déclarations dans les constructeurs des classes qui créent les objets qui les utilisent. Elles constituent toute deux des **services** (au sens Angular du terme, à ne pas confondre avec les services web gérés par le serveur Node), dont une seule instance peut être partagée par les différents objets en ayant besoin.

Pour que l'instanciation des services soit unique, ceux-ci doivent être déclarés dans la propriété **providers** de l'objet de configuration des éléments du root module de notre application :

```
…
import { HttpClientModule } from '@angular/common/http';
import { ProduitsService } from './produits.service';
```

```
@NgModule({
  declarations: [
    AppComponent,
    ConnexionComponent,
    ProduitsComponent,
    CategoriesComponent,
    MenuComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [AuthentificationService, ProduitsService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

La déclaration de la classe *ProduitsService* en valeur de la propriété *providers* en fait un **singleton** : cela signifie qu'au niveau de ce module, un seul objet est instancié et partagé par toutes les autres objets en ayant besoin et pour ce faire ont **injecté une dépendance** dans leur constructeur.

6.2 La bibliothèque NgRx et les stores

L'utilisation de la bibliothèque NgRx [1] dépasse de beaucoup les objectifs de cet article d'introduction à Angular, mais elle répond à un besoin qui peut être impératif si l'application est conséquente : centraliser toutes les données définissant l'état de celle-ci, notamment dans le but de programmer plus aisément la synchronisation de l'affichage des données dans les templates.

La mise en œuvre de la bibliothèque NgRx permet de centraliser toutes les données qui définissent un **état** de l'application dans un objet global. Par exemple, dans le cadre de notre application d'illustration, ces données correspondraient aux identifiants de l'utilisateur ainsi qu'aux données correspondant à sa dernière requête (liste des catégories de produits, liste des produits…).

Cet objet est encapsulé dans une classe appelée le **store** qui est la seule qui permette son accès, et peut le modifier via des actions typées. L'accès aux propriétés du store s'effectue par des **sélecteurs** qui prennent la forme d'observables - tels que nous les avons décrits précédemment dans cet ouvrage lors de la présentation de la programmation réactive avec RxJs - et informent ainsi tous les composants qui y sont abonnés d'un changement d'état de l'application.

Avec la mise en place d'un store, les transferts de données entre l'application Angular et le serveur Node.js sont alors directement gérés par celui-ci. Les services Angular (dont nous avons eu un exemple dans le paragraphe précédent) peuvent continuer d'être mis en œuvre, mais seulement en interface du store. Il est aussi important de noter que chaque sous-module de l'application Angular, peut être associé indépendamment à une propriété de l'objet géré par le store, et gérer ainsi une partie de l'état de l'application.

Il est à noter que NgRx implémente une simplification du pattern de conception Redux qui est lui-même similaire à une simplification du pattern Flux mis en œuvre par

Facebook pour sa bibliothèque React qui a été évoquée dans cet article.

7 Mise en œuvre d'une application Angular

7.1 Création d'une application avec Angular CLI

Intéressons-nous maintenant à la création d'une application Angular. Pour cela nous utiliserons un outil faisant office de gestionnaire de projets, nommé **Angular CLI**. Nous considérons par ailleurs que les services web (le côté serveur) sont opérationnels, et que donc *Node.js* et son gestionnaire de packages *npm* ont été installés sur notre ordinateur (Angular CLI en a besoin).

Angular CLI offre des commandes qui permettent de créer des projets, et au sein d'un projet, différentes entités logicielles dont les plus communes sont les composants et les modules (qui en regroupant des composants peuvent implémenter de grandes fonctionnalités comme la gestion d'utilisateurs, la gestion de produits…). Les squelettes de codes créés par Angular CLI sont écrits dans le langage TypeScript qui est une extension de JavaScript, cette extension permettant de typer les variables, et de créer des classes (cette notion de programmation orientée objets ne sera pas développée dans cet article).

Par ailleurs en mode de développement d'un projet, Angular CLI exécute un serveur local qui scrute toutes les modifications apportées aux codes en développement, et invoque automatiquement un *transpiler* qui génère et transmet au navigateur les codes en JavaScript que celui-ci peut interpréter.

La création d'un projet est opérée par la commande **ng** suivie de l'option *new* et du nom du projet :

```
ng new <nom du projet>
```

L'exécution de cette commande dans votre invite de commande, ou dans votre terminal Linux/Mac OS, engendre automatiquement la création de multiples dossiers et fichiers.
Listons le rôle des plus importants.

Le dossier du projet contient :

- *angular.json* : spécification de la version d'Angular utilisée, de l'endroit où sont stockés ses modules, et de ses dépendances ;
- *e2e* : dossier contenant les outils nécessaires à la spécification et l'exécution de tests ;
- *node_modules* : dossier contenant les modules nécessaires à Angular ;
- *package.json* : spécification des dépendances de votre projet ;
- *src* : dossier des codes spécifiques à l'application.

Le dossier *src* contient par défaut les dossiers et fichiers suivants :

- *main.ts* : code TypeScript spécifiant l'état global de l'application (en développement ou en production) et le module principal (ou *root module*);
- *index.html* : page HTML intégrant le composant de plus haut niveau ;
- *styles.css* : feuille des styles associés au composant à *index.html* ;
- *app* : dossier des codes du projets (modules, composants, services…) ;
- *assets* : dossier des ressources du projet (images...).

Le dossier *app* contient par défaut les fichiers suivants :

- *app.module.ts* : specification du contenu du module principal ;
- *app.component.ts* : classe TypeScript implémentant les traitements associés au composant de plus haut niveau de l'application ;
- *app.component.html* : interface graphique du composant de plus haut niveau ;
- *app.component.css* : styles CSS associés au composant de plus haut niveau.

La création de nouveaux composants est effectuée via la commande **ng** suivie des l'options *generate* et *component* et du nom du composant :

ng generate component <nom du composant>

Le nom du composant ainsi créé est automatiquement inscrit dans le fichier *app.module.ts* qui répertorie tous les composants du module. Un dossier spécifique aux codes associés à ce composant est également créé.

La création d'un module suit la même syntaxe :

ng generate module <nom du module>

7.2 Structure de l'application donnée en exemple

Dans notre exemple, nous mettons en œuvre quatre composants (des *smart components*) qui possèdent une interface graphique (un template) :

- le composant app de plus haut niveau de l'application ;
- le composant *connexion* qui gère l'identification d'un utilisateur (avec un service qui interroge le serveur et mémorise l'utilisateur authentifié) ;
- le composant *produits* (contenu dans le fichier *produits.component.ts*) qui reçoit du serveur la liste des produits et les affiche dans son template (ce composant utilise un service) ;
- le composant *categories* qui reçoit du serveur la liste des catégories de produits à vendre (ce composant utilise un service).

et deux services qui interrogent le serveur :

- le service *authguard* qui invoque le serveur pour vérifier l'authentification d'un utilisateur ;
- le service *produits* (contenu dans le fichier *produits.service.ts*) qui invoque le serveur pour récupérer la liste des compléments alimentaires et leurs catégories.

Chaque smart component est constitué de trois codes mémorisés dans le dossier créé par Angular CLI :

- la classe TypeScript (au sens de la programmation par objets) spécifiant les variables propres au composant ainsi que les traitements associés à celles-ci. Dans le code du composant *produits* nous avons par exemple entr'aperçu que la fonction *ngOnInit()* permettait de récupérer la liste des produits du serveur ;
- le template, c'est-à-dire l'interface en HTML du composant ;
- la feuille de style spécifiant les styles graphiques à lier aux éléments affichés.

7.3 Développement et mise en production

Lors du processus de développement, la commande :

```
ng serve -o
```

lance un serveur local (par défaut sur le port 4200) et ouvre un nouvel onglet dans votre navigateur qui interroge ce serveur pour recevoir les codes JavaScript traduits de TypeScript, et regroupés entre eux dans des paquetages logiciels nommés *bundles*.

Une fois la phase de développement de l'application finie, Angular CLI permet de créer les paquetages finaux pour un serveur de déploiement avec la commande suivante :

```
ng build
```

Une fois le serveur de déploiement mis en place (qui peut être un serveur Node ou autre), l'application est ainsi mise en production.

Angular CLI est de fait un gestionnaire de projets permettant la création d'une application, la mise en place de tests unitaires et fonctionnels (que nous n'avons pas eu le loisir de voir), ainsi que sa mise en production.

8 Conclusion

Cet article n'a pas eu la prétention de présenter le framework Angular dans sa globalité, ce framework représentant à ce jour la proposition la plus riche dans son domaine, et nécessitant un investissement important (il est d'ailleurs quelque fois critiqué pour cela). Nous avons surtout voulu mettre en lumière ses deux valeurs phares : l'utilisation de services web et la mise en œuvre de composants, et aussi, même si cela a juste été esquissé, de modules. Au-delà de ces deux aspects fondamentaux, Angular présente aussi d'autres qualités que nous voudrions citer. Au niveau du développement logiciel, Angular permet l'injection de dépendance qui autorise différents composants à « partager » l'instanciation d'un autre composant (appelé service). Par ailleurs l'intégration de la programmation réactive est une puissante technique de programmation permettant d'écouter des flux de données. Nous avons eu un petit aperçu de ces deux facettes. Au niveau fonctionnel, de nombreuses bibliothèques Angular mettent à disposition des développeurs une quantité impressionnante de modules et de composants déjà développés, notamment de nombreux composants graphiques (des UI components).
Nous espérons que cet article vous aura donné le goût de découvrir, ou de faire découvrir, ce puissant framework de développement d'applications web.

Glossaire

Application web monopage (Single Page Application)
Application web dont la structure est complètement transmise au navigateur lors de son chargement initial. Cette structure comprend les templates contenant les balises HTML des différents composants, et les codes JavaScript qui permettent de mettre à jour l'affichage du navigateur en fonction des données ultérieurement importées.

Bundle
Regroupement de plusieurs fichiers de programmes (il peut être désigné aussi sous le nom de package ou de paquetage).

DOM (Document Object Model)
Le modèle objet du document représente l'outillage (les classes en programmation objets) permettant d'allouer en mémoire du navigateur les différentes entités de la page web (balises, attributs informations textuelles…). Par extension, il désigne toutes les données mise en mémoire et associées à une fenêtre du navigateur.
Un lien est donné sur sa spécification officielle dans la webographie.

HTML (HyperText MarkUp Language)
Langage de programmation permettant de structurer les pages web affichées dans le navigateur, et de mettre en forme les informations contenues dans ces pages. Ce langage met en œuvre des duos de balises qui exécutent leurs directives sur les données, ou d'autres balises, qu'elles encadrent (il existe aussi des balises uniques).
Un lien est donné sur sa spécification officielle dans la webographie.

HTTP (HyperText Transfert Protocol)
Protocole de communication réseau client-serveur pour Internet.
Un lien est donné sur sa spécification officielle dans la webographie.

JavaScript
Langage de programmation permettant la programmation de scripts initialement destinés à accéder au contenu de pages web chargées en mémoire d'un navigateur (et à les modifier). Ce langage a ensuite évolué notamment avec les normes ECMAScript définies par l'ECMA (une organisation émettant, entre autres, des normes de standardisation pour les langages de programmation), et permet par ailleurs la création de serveurs avec l'utilisation de *Node.js*.

JSON (JavaScript Object Notation)
Notation sous format textuel, appelée sérialisation, d'objets JavaScript qui sont directement alloués en mémoire par un programme JavaScript auquel ils sont transmis.

MEAN (**M**ongoDB **E**xpress **A**ngular **N**ode.js)
Architecture logicielle structurant des applications web grâce à l'utilisation d'Angular comme framework client, de Node.js comme gestionnaire de services web et de MongoDB comme système de gestion de bases de données NoSQL.

MongoDB
Système de gestion de base de données NoSQL orienté Documents. Les collections d'une base de donnée gérée par MongoDB mémorise des objets sous un format binaire *BSON* qui peut être rapproché du format de sérialisation *JSON*. Le système d'interrogation des données n'utilise pas le langage SQL et les objets peuvent être modifiés dynamiquement (avec l'ajout de nouvelles propriétés).

Node.js
Interpréteur JavaScript modulaire pouvant être exécuté de manière autonome (en dehors d'un navigateur). Très orienté réseau, il est couramment utilisé pour créer des serveurs.

Route REST
Syntaxe permettant l'appel d'une ressource informatique accessible sur Internet via les méthodes proposées par le protocole réseau HTTP, et en n'exposant pas le nom d'un programme particulier à exécuter sur le serveur.
Un lien est donné sur un projet de spécification officielle dans la webographie.

SEO (Search Engine Optimization)
Ensemble des bonnes pratiques et techniques d'optimisations des sites web pour leurs référencements par les moteurs de recherche.

Service web de type REST
Agent logiciel permettant l'échange de données entre deux applications (typiquement un navigateur et un serveur) via des routes REST (et donc le protocole HTTP). Les routes REST orientées données n'exposent pas les traitements sous-jacents.

Template
Dans le contexte d'Angular, code directement écrit en HTML (ou intégrant du code TypeScript générant des balises HTML), implémentant l'interface d'un composant.

Transpiler (ou transcompilateur mais ce terme est peu usité)
Traducteur du code source d'un langage vers un autre code source. Dans le contexte de JavaScript, traducteur d'une extension à ce langage au code canonique de celui-ci (quelquefois les transpilers sont nommés abusivement compilateurs).
Le transpiler Angular (tsc ou sa nouvelle version ngtsc) traduit les codes TypeScript des composants du projet en codes JavaScript qui sont ensuite empaquetés avec ceux des modules associés, pour être transmis au navigateur.

TypeScript
Extension au langage JavaScript permettant le typage des variables et l'utilisation de classes (au sens de la programmation objets). Le développement en TypeScript nécessite l'utilisation d'un *transpiler* pour traduire le code TypeScript en code JavaScript.

Sigles, notations et symboles

Symbole	Description
DOM	Document Object Model
HTML	HyperText MarkUp Language
http	HyperText Transfert Protocol
JSON	JavaScript Object Notation
MEAN	MongoDB Express Angular Node.js
REST	Representational State Transfert
SEO	Search Engine Optimization

Sources bibliographiques

[1] NORING Christoffer - Architecture Angular Application with redux, rxjs and ngrx, Editions Packt Publishing (2018)

[2] POMPIDOR Pierre – Angular et Node.js : optimisez le développement de vos applications web avec une architecture MEAN. Editions ENI (2016).

[3] SESHADRI Shyam – Angular: Up and Running, Learning Angular, Step by Step, Editions O'Reilly Media (2018)

Lien sur le dépôt GitHub de l'application d'illustration :

https://github.com/PierrePompidor/SUPERVENTES

Sites Internet

Site officiel d'Angular : https://angular.io/

Site officiel d'Angular CLI : https://cli.angular.io/

Site officiel d'Angular Material : https://material.angular.io/

Lien didactique sur Angular Universal Toolkit :
https://techblog.fexcofts.com/2018/08/13/angular-seo-universal-toolkit/

Spécification du DOM : https://www.w3.org/DOM/

Spécification de HTML : https://www.w3.org/HTML/

Spécification de HTTP : https://www.w3.org/Protocols/

Site officiel de JSX : https://reactjs.org/docs/introducing-jsx.html

Site officiel de MongoDB : https://www.mongodb.com/fr

Site officiel de Node.js : https://nodejs.org/fr/

Site officiel de Polymer : https://www.polymer-project.org/

Site officiel de React : https://reactjs.org/

Projet de spécification de REST : https://www.w3.org/2001/sw/wiki/REST

Site officiel de Vue.js : https://vuejs.org/