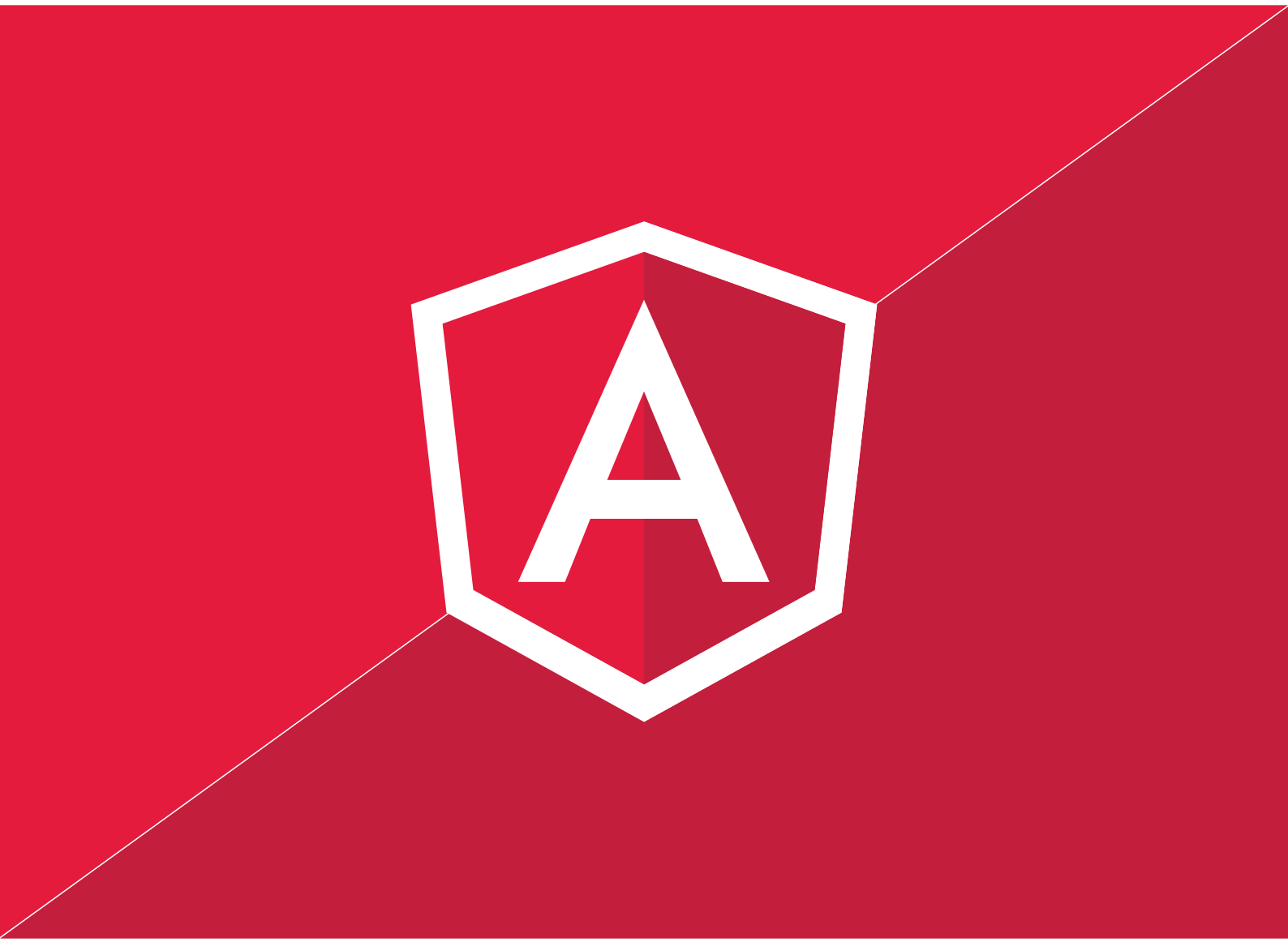


Table des matières

Qu'est ce qu'Angular ?

Angular est un framework Javascript open source, gratuit, développé par les équipes de Google. Il est une évolution de AngularJS (la version 1 du framework) avec une rupture substantielle dans les concepts initiaux à partir de la version 2.

L'objectif des équipes de développement avec la mouture 2+ est de proposer une palette de technologies intégrées prenant en compte des normes avancées du Web dans une orientation de développement logiciel. C'est à la fois un outil de facilitation dans l'intégration de concepts tels que les composants en HTML, les classes et modules Javascript et de facilitation pour l'utilisation des technologies de compilation, de minification, etc.. Angular est un écosystème complet prêt à l'emploi. Concrètement :

» Il s'appuie sur les normes Web récentes ou émergentes (classes, modules…).

» Sa mise en production s'effectue à l'aide de technologies et framework variés : NodeJS, Typescript, Webpack, Karma, Jasmin et Angular CLI.

L'écosystème d'Angular

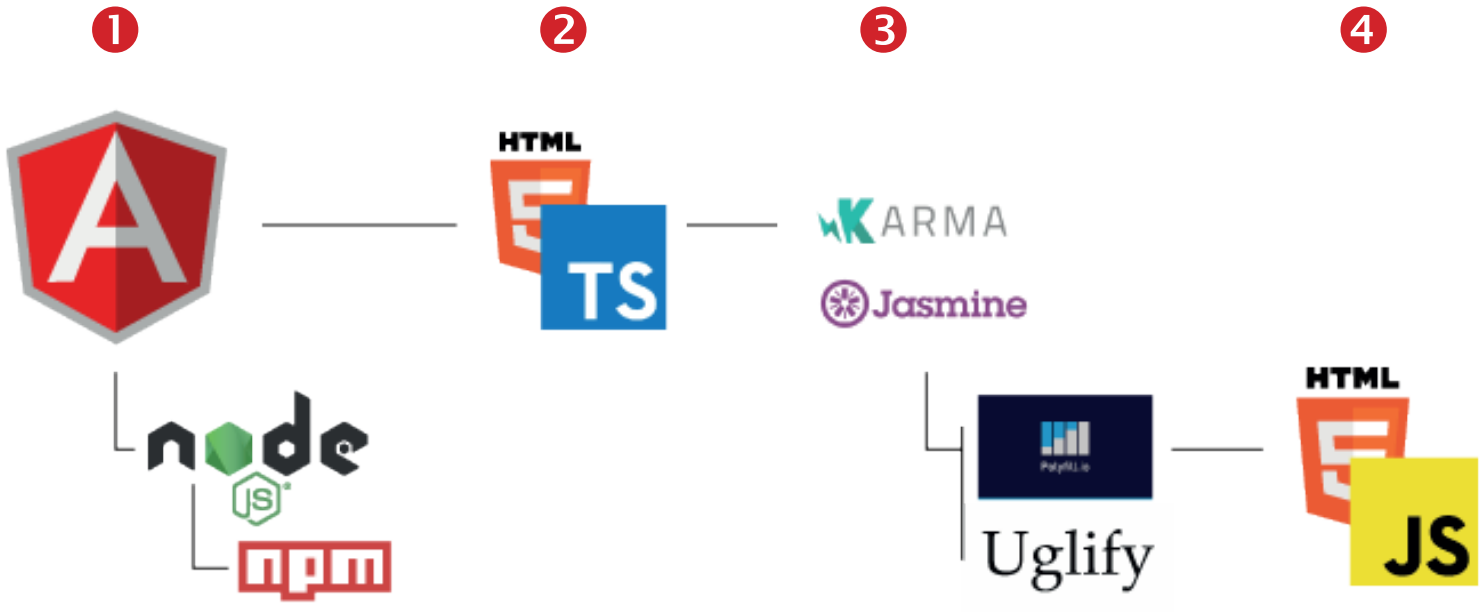


Fig. 1 - Quelques technologies de l'écosystème technologique d'Angular

1 - L'environnement technologique

Chaque colonne de ce schéma correspond à une étape du processus de production d'une ap-

plication Angular. Il se base sur NodeJS et NPM et répond à leurs critères :

» Les modules sont téléchargés dans un dossier node_modules (une convention NodeJS).

» Lors de la création d'une application Angular, un fichier package.json référence les versions utilisées et les options de la dite application.

» La création ou l'ajout de module peut se faire via des lignes de commande NPM :

```
npm install monModule
```

Dans cet exemple, le module "monModule" sera téléchargé dans le dossier node_modules de l'application.

En outre, un outil de lignes de commande a été développé en 2017 pour offrir un ensemble d'options de création dans l'environnement, Angular CLI. Cette technologie sera abordée ultérieurement.

2 - Environnement de développement

Outre Angular CLI, deux technologies centrales sont utilisées pour le développement d'applications : TypeScript et HTML.

TypeScript est un métalangage créé par les créateurs de C#. Il est transpilé en Javascript. Dans le cadre de ce cours nous l'utiliserons avec l'IDE Visual Studio Code qui l'intègre nativement.

Les HTML et CSS sont une partie du code gérant les vues. Il n'ont pas de restrictions. Le CSS sera géré directement par Angular. C'est l'une des clés de son fonctionnement. Le HTML s'enrichit de directives permettant d'interagir avec ses contenus.

PLUS D'INFORMATION DANS L'ANNEXE 1

3 - Mise en production

Plusieurs outils sont convoqués lors de la mise en production. Karma, Jasmine et Protractor sont utilisés pour les tests fonctionnels ou E2E (end to end).

Uglify va minifier les fichiers générés et Polyfills s'occupera d'une adaptation de Javascript aux différents navigateurs pour assurer une compatibilité avec les anciennes versions.

4 - HTML, CSS et Javascript

Les fichiers générés sont dans ces trois technologies. Pour en savoir plus, reportez-vous au chapitre "Mise en production".

Angular CLI

Des applications clé-en-main

Les lignes de commandes sont un usage largement développé sur l'ensemble des technologies basées sur Javascript. Angular ne déroge pas à cette règle.

Les évolutions du Web intègrent nombre de concepts professionalisant la création d'applications. Parallèlement, les équipes de Google ont intégrés toute une palette de technologies (cf. ci-dessus) complémentaires gérant différents niveaux du cycle de vie d'une application. Les lignes de commandes facilitent ainsi l'orchestration de toutes ces dimensions.

Angular CLI évolue très vite. Nous ne pourrons qu'offrir un tour d'horizon générique des principales fonctionnalités en vous invitant à vous référer à l'aide disponible sur GitHub pour des informations à jour.

ANGULAR CLI SUR GITHUB

Installer Angular CLI

C'est un module NPM. L'installation se fait en globale mais attention, des fichiers locaux seront installés avec chaque application.

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

Cette commande installe CLI en global (-g) en téléchargeant la dernière version (@latest).

Lors d'une mise à jour la même commande peut être utiisée (cf. GiHub) mais les projets locaux devront eux aussi être mis à jour. Suivez scrupuleusement les recommandations offertes sur GiHub.

(les processus de février 2018)

Désinstaller les versions locales :

```
npm uninstall -g angular-cli
npm uninstall --save-dev angular-cli
```

Désinstaller les versions globales :

```
npm uninstall -g @angular/cli
npm cache clean
# if npm version is > 5 then use `npm cache verify` to avoid errors (or to avoid using --force)
npm install -g @angular/cli@latest
```

Enfin, réinstaller la version locale :

```
rm -rf node_modules dist # use rmdir /S/Q node_modules dist in Windows Command Prompt; use rm -r -fo node_modules,dist in Windows PowerShell
npm install --save-dev @angular/cli@latest
npm install
```

Que faire avec Angular CLI ?

Une fois installé Angular CLI offre des commandes en utilisant le préfixe "ng". Par exemple :

```
ng new monApplication
```

Quelques détails sur les commandes

```
ng new monAppli
```

Va créer une application ex nihilo. Rapidement les fichiers de configuration, un premier composant ainsi que l'ensemble des fichiers de fonctionnement (styles, index.html...) seront créés. Les fichiers d'Angular seront téléchargés dans le dossier node_modules.

Attention, ils font près de 250Mo !

Vous devriez obtenir un dossier de cet ordre. Il sera détaillé plus loin :

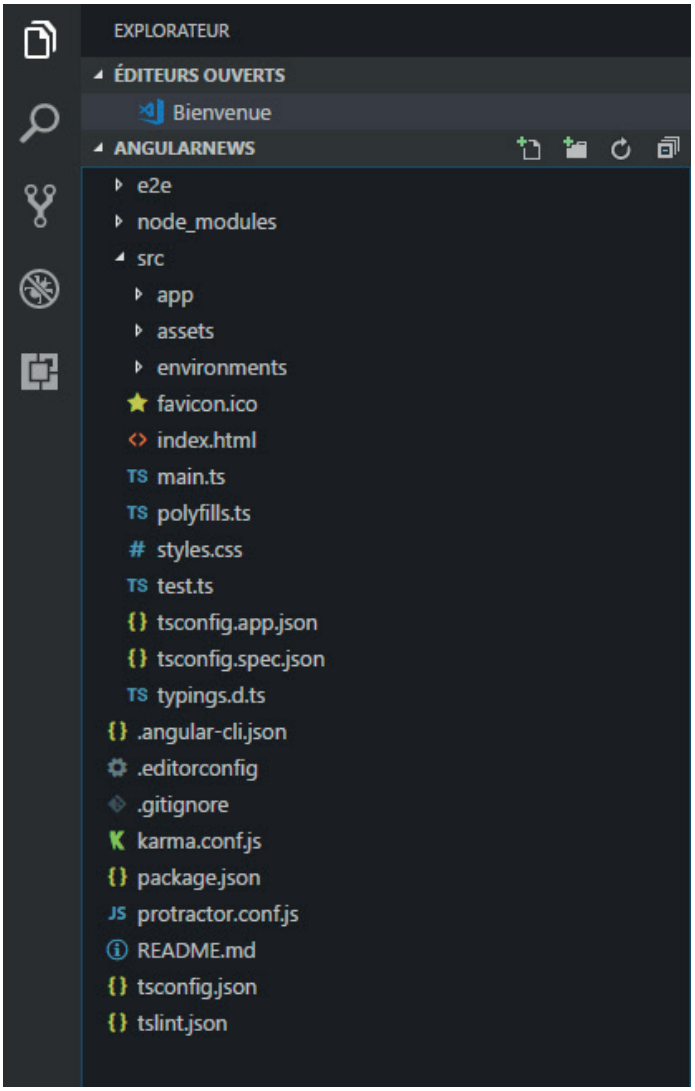


Fig. 2 - Arborescence d'un projet créé ex nihilo

ng generate

Cette commande sera la commande centrale lors de la phase de développement. Elle permet de créer la plupart des outils dont nous aurons besoin. Un exemple pour créer un composant :

ng generate component monComposant

Dans cet exemple, CLI créera un composant, il l'attachera au module principal et ajoutera un fichier HTML de template et un fichier CSS spécifique.

Ces commandes peuvent être réduites à une expression plus simple :

ng g c monComposant

Nous avons aussi accès à toutes une palette de créations telles que des services (ng g s monService), des guards... Nous aurons l'occasion de les pratique plus concrêtement dans les exercices.

Détail de l'application créée

Le dossier node_modules

Comme son nom l'indique, ce dossier est un standard de NodeJS. Il regroupe l'ensemble des fichiers opérationnels pour créer les différentes parties d'une application. La commande npm install ajoutera des modules dans ce dossier.

Le dossier src

Ce dossier regroupe les éléments du projet :

» **app** : contient l'ensemble des scripts TypeScript. Ce sera le **dossier de travail** ;

» **assets** : le dossier référencé qui sera intégré à l'application au moment de sa publication. Il est le dossier usuel pour intégrer des images et autres ressources ;

» **environment** : un dossier mécanique pour la gestion des environnement de développement ou de production.

» **index.html** : l'index de l'application ;

» **style.css** : ...

» **polyfills.ts** : le fichier gérant l'adaptation pour les navigateurs anciens ;

» **main.ts** : le fichier d'initialisation de l'application Angular. Nous n'aurons pas à y toucher.

e2e

Un dossier technique pour la gestion des tests end2end.

Le dossier src/app

Ce dossier contient un composant initial et un module app.component.ts et app.module.ts. Le composant est relié à une feuille de style et une page HTML par défaut.

Configurations d'une application

Différents fichiers sont créés avec une application pour chacune des technologies : Node, TypeScript, Angular CLI. Détails.

Fichier package.json

Ce fichier est dédié à la gestion des dépendances de l'application. C'est un fichier NodeJS servant au chargement et la mise à jour des modules installés.

Le code qui nous intéresse est compris dans les objets "dependencies" et "devDependencies". Ce sont les listes des dépendances de production et de développement utilisées. Nombre sont ajoutées à la création de l'application. D'autre peuvent ête ajoutées avec NPM :

```
npm install monModule --save (ou --save-dev)
```

La propriété --save et --save-dev ajouteront la dépendance dans ce fichier pour en gérer les mises à jours.

Toutes répondent à une mécanique de versioning SemVer (pour Semantic Versioning) avec trois numéros :

```
"@angular/animations": "^5.1.0"
```

Dans notre exemple :

- 5 : version majeure,
- 1 : version mineure,
- 0 : les corrections de bugs.

L'utilisation de codes permet ensuite de retreindre ou autoriser certains niveaux de mises à jour sur des "update" :

- ^ (accent circonflexe) : accepte les mises à jours mineures (ici version max. : 5.9.9) ;
- ~ (tilde) : n'accepte que les corrections de bugs (ici version max. : 5.0.9).

EN SAVOIR PLUS SUR SEMVER

Fichier .angular.json

Angular-cli.json est plus complet en termes d'options. Il est l'outil essentiel de paramétrage de l'application. Nous retiendrons quelques paramètres utiles :

```
"BUILD": {
        "BUILDER": "@ANGULAR-DEVKIT/BUILD-ANGULAR:BROWSER",
        "OPTIONS": {
```

```
            "OUTPUTPATH": "DEMO/",
            "INDEX": "SRC/INDEX.HTML",
            "MAIN": "SRC/MAIN.TS",
            "POLYFILLS": "SRC/POLYFILLS.TS",
            "TSCONFIG": "SRC/TSCONFIG.APP.JSON",
            "ASSETS": [
              "SRC/FAVICON.ICO",
              "SRC/ASSETS",
              "SRC/.HTACCESS"
            ],
            "STYLES": [
              "SRC/STYLES.CSS"
            ],
            "SCRIPTS": []
          },
```

La partie "apps" de la configuration règle l'organisation de l'application en fixant des aspects modifiables :

» root : le dossier de développement "src" par défaut ;

» outputPath : le dossier d'export en utilisant ng build et ng build --prod ;

» index.html et main.ts : respectivement l'adresse des fichiers de base de l'application ;

» assets : un lien vers un dossier regroupant les ressources intégrées à l'application, généralement les images. Tous les dossiers et fichiers y figurant seront copiés dans le dossier de publication "outputPath" ;

» prefix : la préfixe de l'application utilisé dans le code ;

» style : une liste de styles globaux. C'est un tableau, on peut en ajouter ;

» scripts : idem pour des scripts supplémentaires.

Fichier tsconfig.app.json

Le fichier tsconfig.json paramètre la compilation des fichiers .ts, tsx et .d.ts. Tous les fichiers importés ou listés seront convertis en suivant les règles édictées. Chaque option représente une commande de compilation.

La valeur la plus intéressante pour nous est 'target' qui permet de régler le moteur Javascript. es2015 ou es6 adressant l'ECMAScript6.

LA LISTE COMPLÈTE DES OPTIONS DISPONIBLES

Concepts généraux

Décorateurs

Interfaces

Angular s'appuie sur des concepts tirés de l'évolutions des normes ECMAScript (modules, classes...) et de TypeScript (typage, classes abstraites, interfaces) (cf. les cours correspondants). Il cumule les avantages des deux technologies pour offrir une solution se rapprochant d'un développement de logiciels. Autour de cette palette ont été développés un ensemble de principes fonctionnels.

Deux niveaux de concepts sont à prendre en compte : les concepts liés aux normes du Web et ceux mis en oeuvre par Angular avec leurs modalités (même s'ils sont eux même inspirés des normes).

» **Décorateurs**

Les décorateurs offrent un moyen d'évaluer des classes et objets pour leur adjoindre des propriétés particulières.

» **Interfaces**

Les interfaces sont essentiellement utilisées comme des schémas permettant de typer des objets.

Des décorateurs

@NgModule, @Component, @Injectable, @Pipe, @Input, @Output, @ViewChild...

Les décorateurs sont une forme de typage proposé comme norme pour les futures versions de Javascript. Angular a pris les devants en les intégrant et en faisant un usage systématique. L' exemple, ci-contre définit un service injectable.

Les décorateurs de classes (@Component, @NgModule, @Injectable…) sont les plus communs et doivent être utilisés pour définir le type de chaque classe en fonction de leur usage. Nous les détaillons plus loin.

Les décorateurs de propriétés permettent notamment des binding sur les données comme (@ Input, @Output...) afin de faire communiquer les composants.

Les décorateurs se déclarent comme des méthodes, avec des parenthèses et peuvent parfois recevoir un objet en argument (ex. @Component({}))

Les interfaces

Créer un type spécifique

Typescript a ajouté la notion d'interface qui permet de définir un type à des objets. Ces types peuvent être définis puis importés dans une classe quelconque. C'est particulièrement utile dans un contexte de développement strict en équipe. Les données reçues seront nécessairements typées correctement ou lanceront une erreur.

Nous pouvons créer un fichier nouvelles.ts et exporter une interface :

```
export interface Nouvelles {
    id:number;
    titre:string;
    description:string;
    img:string;
}
```

Nous pouvons ensuite l'importer et l'utiliser dans un composant :

```
import { Nouvelles } from '../nouvelles';
news:Nouvelles[];
newsActu:Nouvelle;
```

Nous avons un nouveau type pour nos objets.

Des propriétés optionnelles

Il est possible e rendre certaines propriétés optionnelles pour éviter les erreurs. Une propriété doit être suivie d'un ?.

```
export interface Nouvelles {
    id:number;
    titre:string;
    description:string;
    img?:string;
}
```

Les points d'interrogations peuvent être utilisés dans la plupart des déclarations d'arguments des fonctions et objets.

Concepts & outils Angular

Les interfaces sont ici utilisées pour typer des objets en définissant des propriétés stables. Elles sont héritées de TypeScript.

» **Templates & styles**

Des fichiers HTML et CSS qui possèdent des directives et processus singuliers pour interagir avec les données.

» **Modules**

Aiguilleurs de l'application, déclarent l'ensemble des scripts utilisables.

» **Composants**

Les composants pourraient s'assimiler à un vue-modèle dans un patron MVVM. Il supporte les interactions avec le gabarit HTML et gère les données qui lui sont attachées.

» **Routes**

Les routes gèrent la réécriture des URLs, les paramètres transmis et l'accès aux composants attachés à chaque route.

» **Services**

Les services fournissent des données et méthodes partagées. Ils servent généralement de plate-forme pour la distribution des données du modèle.

» **Observables**

Les observables sont intégrés depuis RxJS, déclinaison de RxJava. Ils offrent des outils avancés pour le traitement et la synchronisation de données. Une évolution des promesses.

» **Guards**

Les guards permettent de sécuriser l'accès à des URLs ou à modulariser une application.

» **Intercepteurs**

Les intercepteurs permettent de filtrer les appels HTTP pour réécrire des entêtes, valider des paramètres de sécurité (etc).

Nous allons détailler chacun de ces concepts mais avant tout abordons les templates et les styles pour aborder le cour du fonctionnement d'Angular.

» **Pipes**

Les pipes sont des filtres appliqués en directives à des données dans le HTML. Il permettent un traitement rapide de données dans le front.

Les templates

Binding / double binding

Les templates dans Angular sont un code HTML incluant un ensemble de fonctionnalités permettant de dialoguer avec les composants auxquels ils sont rattachés. Ils peuvent recevoir des données dynamiques, en renvoyer, appliquer des filtres… le tout avec un ensemble de codes. Par exemple :

{{maVariable}} affiche une variable du composant auquel la page HTML est rattachée.

```
<h1>{{titre}}</h1>
```

Dans cet exemple, le titre sera affiché à partir de la valeur de la variable "titre" du composant.

Les expressions

Les expressions sont des données injectées dans le HTML, généralement en arguments de balises. Elles sont insérées par le biais de crochets [uneValeur] et permettent d'appliquer des mises à jours après le chargement d'une application, notamment lorsqu'elles sont reliées à des variables ou méthodes de composant d'Angular. Par exemple :

```
<img [src]="objet.imgSrc" />
```

Dans cet exemple, la source de l'image sera la propriété "imgSrc" de l'objet "objet "du composant. Dans ce cas particulier, l'utilisation des crochets permettent d'éviter des erreurs de chargement des images.

Ils permettent aussi d'afficher des données liées au composant, ici en lien avec un style :

```
<p [style.backgroundImage]="['url(objet.imgSrc)']">
```

Les expressions permettent la synchronisation de données en temps réel entre le composant et le HTML. Ils sont générés par des "watchers" qui tournent en boucle en arrière plan. AngularJS a pu être critiqué pour ses pertes de performances sur les applications lourdes. Dans Angular, les options d'opérations sur les expressions ont été limitées à l'affichage pour éviter d'alourdir les performances.

Les événements & instructions

() servent à l'envoie de données dans des événements

```
<a (click)="maFonction(12)"></a>
```

En terme mnémotechnique, les crochets servent à l'entrée des données dans le HTML et les parenthèses à sortir ces données vers le composant.

L'ensemble des événements de Javascript sont exploitables avec cette syntaxe. Par exemple :

```
<button (click)="confirme()" (mouseenter)="visible=true" (mouse-
leave)="visible=false">Faire apparaître une étiquette et faire une action</
button>
```

Ici le bouton appellera la méthode confirme() dans le composant auquel il est attaché et générera des événements sur le over et le out (changement de la valeur visible du composant).

En parallèle, il est possible d'insérer plusieurs traitements sur un même événement :

```
<button (click)="confirme(); visible=false"
```

Des directives de structure

Avec Angular on peut mener des opérations directement dans le HTML en utilisant quelques directives bien pratiques : des conditions avec *ngIf et des boucles avec *ngFor, mais aussi show et hidden.

Des conditions

*ngIf permet de conditionner l'affichage d'une balise à une condition :

```
<li *ngIf=”valeur > 2”>
    <h2>{{valeur}}</h2>
</li>
```

La balise n'est incrite dans le DOM que lorsque la condition est réalisée. ngShow et ngHide ne jouent que sur la fonction display d'une balise :

```
<li [show]=”publie” [hidden]=”!valid”>
    {{obj.titre}}
</li>
```

Des boucles

*ngFor... of : une boucle sur un objet ou un tableau/ Par exemple nous pouvons dupliquer une balise en fonction d'un tableau d'objets :

```
<li *ngFor=”let obj of objets”>
    <h2>{{obj.titre}}</h2>
</li>
```

A l'intérieur de ces boucles peuvent s'appliquer des Pipes (cf. ci-dessous) ou un index :

```
<li *ngFor=”let obj of objets | async; let i = index”>
    {{i}} - {{obj.titre}}
</li>
```

Les styles

Chaque composant créé par Angular CLI aura un fichier CSS connecté qui n'adressera que ce composant. Cette question sera détaillée avec les composants (cf. ci-dessous).

Il est possible d'intégrer des fichiers globalement en paramétrant des liens dans le fichier de configuration de Angular CLI. Attention, ces fichiers seront évalués comme les autres scripts et intégrés dans le header de la page HTML. Pour ajouter des styles non compilés, mieux vaut les inscrire directement dans le fichier index.html.

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

Des styles et classes manipulables

La syntaxe la plus simple est l'utilisation de crochets. Angular identifie les propriétés CSS avec une syntaxe en camel case, inutile d'apprendre un nouveau langage :

```
<div [style.marginTop]="marge"></div>
<div [class]="'text-success'"></div>
```

NgStyle & NgClass

Deux directives permettent de créer des styles à la volée dans un template.

La directive permet de créer des styles dans une balise en lui fournissant un objet :

```
div [ngStyle]="{'background-color':'green'}"></<div>
```

On peut y ajouter des conditions par exemple :

```
<div [ngStyle]="{'background-color':perso.pays === 'fr' ? 'blue' : 'red'
}"></<div>
```

NgClass agit de la même manière pour attribuer des classes :

```
[ngClass]="{'text-success':perso.pays === 'fr'}"
```

Des variables locales

Des variables peuvent être définies dans un template pour être utilisées directement ou depuis le composant. Il suffit de les écrire comme de id css :

```
<li *ngIf="valeur > 2" #menu>
    <h2 #titre >{{valeur}}</h2>
</li>
```

Ces variables permettent le ciblage de balises dans le composant en utilisant la décoration @

ViewChild :

```
import { Component, ViewChild, ElementRef } from '@angular/core';
@ViewChild('titre') titre: ElementRef;
this.titre.nativeElement.style="color:#F00";
```

@ViewChild est un décorateur. Il s'appuie sur la classe ElementRef pour interagir avec les éléments du DOM.

Mais elles sont aussi particulièrement utiles dans les formulaires pour mesurer les états des champs ou transmettre des données d'évaluation. Nous le verrons plus loin.

Les modules

Angular est conçu sur le standard des Web Components. Le coeur de la distribution des classes dans une application est le(s) module(s). Il permet de structurer une application en rendant des briques de code disponibles dans des contextes spécifiques d'utilisation.

Lorsqu'une application est créée, un module de base l'est aussi il relie automatiquement les composants. De même, lorsque un composant est créé avec la commande ng generate, il est déclaré automatiquement dans le module principal pour l'exporter dans toute l'application.

Un module est construit sur plusieurs logiques. Il met à disposition des ressources pour les autres scripts, des composants, des services ou d'autres modules contenant eux mêmes d'autres composants et d'autres services (...).

» **declarations** – Dans cette propriété, vous devez déclarer les classes de vue de votre module. Par classes de vue, il faut comprendre "components", "directives" et "pipes";

» **exports** – Cette propriété permet de déclarer le sous-ensemble de déclaration qui sera visible par les autres modules ;

» **imports** – Cette propriété permet de déclarer les modules dont dépend notre module. Ces modules contiennent eux aussi des déclarations ;

» **providers** – Dans cette propriété, vous devez déclarer les services que vous allez créer dans le cadre de ce module. Ces services contribueront à alimenter la collection globlale des services accessibles par tous les composants de l'application.

Ci-dessous nous avons un exemple de module créé en utilisant la fonction ng new ''. Quelques commentaires :

» **BorwserModule** : une classe qui permet de créer une application. Il importe et déclare lui même la classe CommonModule qui donne accès aux directives types *ngIf.

» **NgModule** : déclare le décorateur du même nom ;

» **@NgModule** : un décorateur qui offre des instructions de compilation et de déclaration de classes :

» **imports** : importe la classe BrowserModule par défaut. Ce tableau permettra d'importer tous les modules dont nous aurons besoin dans une application ;

» **declarations** : le tableau qui déclare tous les composants et pipes de l'application.

» **providers** : un tableau dédié aux services partagés. Les services sont instanciés à chaque déclaration dans un tableau "providers". Un service peut ainsi être de fait un singleton en n'étant déclaré qu'une fois. Chaque instance dans un composant fera référence à l'instance du provider ;

» **bootstrap** : le tableau de déclaration du composant de démarrage, celui qui lancera les opérations. Seul le module de initial peut déclarer le composant de démarrage.

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

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


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

POUR ALLER PLUS LOIN SUR CETTE MÉCANIQUE

Les composants

Les composants s'apparentent à des contrôleurs dans un modèle MVC. Ils portent le code actif relié à une vue (template). En réalité, ils sont plus proches d'une Vue Modèle dans un patron de conception MVVM (Modèle, Vue, Vue-modèle). Ils sont une extension de la classe Directive.

Créer un composant

Le plus simple est avec CLI :

```
ng generate component MonComposant
```

ou

```
ng g c MonComposant
```

Lors de sa déclaration, le composant reçoit un objet en argument de son décorateur. Il peut contenir :

» un **sélecteur** indiquant la balise du composant dans l'application (selector:'app-nom'). Ce sélecteur sera le nom de la balise HTML qui pourra être intégrée dans un fichier pour faire apparaître le template HTML du composant ;

» le code HTML à afficher : **template** pour du code brut, **templateUrl** pour un lien vers un fichier ;

» la même chose pour les styles : **style** et **styleUrls**.

Par défaut, Angular CLI crée des fichiers liées pour les styles et le HTML.

Composant, HTML & sélecteur

Ci-dessous l'exemple du code de base généré par Angular lors de la création d'une application :

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-appli',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class MonComposant {

}
```

Le fichier index.html se présente ainsi :

```
<body>
  <app-appli></app-appli>
```

```
</body>
```

La balise <app-appli> a été ajoutée automatiquement, la valeur du sélecteur du composant. Un héritage de la classe Directive.

Cycle de vie des composants

Les composants peuvent implémenter un ensemble de méthodes permettant de gérer leurs cycles de vie, chacune d'elles se déclenchant à des moments particulier dans un ordre précis :

- ngOnChanges
- ngOnInit
- ngDoCheck
- ngAfterContentInit
- ngAfterContentChecked
- ngAfterViewInit
- ngAfterViewChecked
- ngOnDestroy

Les plus utilisées sont ngOnInit et ngOnDestroy. Une implémentation déclarée doit avoir une méthode liée :

```
import { Component, OnInit, OnDestroy } from '@angular/core';
export class MonComposant implements OnInit, OnDestroy {
    ngOnInit(){};
    ngOnDestroy(){};
}
```

PLUS D'INFO SUR CE CYCLE DE VIE

Constructeur

Dans une classe Javascript les constructeurs sont explicites. Cette règle est valable en TypeScript et Angular. Ils permettent d'instancier des classes importées :

```
import { Component, OnInit } from '@angular/core';
import { MonInterface } from './model/MonInterface';
import { AutreComposant } from './AutreComposant';
export class MonComposant {
    titre:MonInterface;
    constructor(private autreComposant:AutreComposant){};

    ngOnInit(){
            this.titre = this.autreComposant.titre;
```

```
    }
}
```

Dans cet exemple, nous récupérons la valeur du titre dans un second composant instancié dans le constructeur.

Binding

Les valeurs déclarées dans un composant sont directement accessibles dans le template HTML qui lui est attaché :

```
import { Component } from '@angular/core';
export class MonComposant {
    titre:string = "Bienvenue";
    description:object = {courte:'Description courte', longue:'Description longue'};
}
```

Dans le HTML :

```
<h1>{{titre}}</h1>
<p>{{description.courte}}</p>
```

Héritage & composants

Un composant peut être l'enfant d'un autre composant. Pour ce faire, il suffit de déclarer le sélecteur de l'enfant dans le HTML du composant parent. Par exemple, dans le template HTML du composant parent :

```
<app-enfant></app-enfant>
```

Le template du composant enfant sera affiché dans le composant parent.

Envoyer des données à un enfant avec @Input()

Des valeurs peuvent être échangées entre le composant parent et l'enfant en utilisant les décorateur @Input().

Déclaration d'une variable input dans le composant parent :

```
@Input()
titre:string="Bienvenue";
```

Déclaration d'une variable dans le composant enfant :

```
titreEnfant:string;
```

Ajout de la valeur dans le HTML, à l'intérieur de la balise enfant :

```
<app-enfant [titreEnfant]="titre"></app-enfant>
```

La valeur de titreEnfant sera redéfinie par la variable ttre du parent.

Les routes

Très rapidement, il sera nécessaire d'organiser les composants pour scénariser les utilisations de votre site Internet. Les routes serviront à ça en offrant un système de réécriture des URLs et en chargeant le composant correspondant. Ces fonctions sont inclues dans la classe Router de Angular.

La classe Router permet de gérer les URLs ainsi que le chargement de composants en fonction de la route.

La classe ActivatedRoute permet le transfert de paramètres.

Quelques principes

» Les routes sont contenues dans un tableau donnant toutes les listes des adresses acceptées.

» Des paramètres peuvent être transmis dans ces adresses.

» Ce tableau est déclaré dans le module principal.

» Des routes complémentaires peuvent être déclarées dans un tableau secondaire puis dans un module secondaire.

» Le fichier index.hml définit la base de calcul des routes dans la balise <base> dans le header. Toutes les routes seront calculées à partir de cette adresse, en développement comme en production :

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

Dans cet exemple nous paramétrons une adresse racine absolue pour redéfinir les réécritures à partir de la racine du site.

Des routes faciles

Nous pouvons créer une application avec des routes près paramétrées.

```
ng new monApp --routing
```

Un module "AppRoutingModule" définissant les routes sera créé et devra être injecté dans le module principal.

Les routes seront créées dans un tableau importé ensuite doit être importée et exportée dans un module :

```
const routes: Routes = [];
```

Le tableau des routes (détaillé ci-dessous).

```
import { Routes, RouterModule } from '@angular/router';
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
```

Les imports et exports sont gérés automatiquement dans le module créé. Celui-ci doit être déclaré dans les imports du module racine :

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

Détails sur le tableau des routes :

Attention, l'ordre des routes importe. Il s'agit d'un tableau qui est lu par itération aussi, le premier "path" qui match avec l'URL sera lancé, y compris pour les erreurs 404.

Un composant de démarrage sera chargé sur l'adresse racine :

```
{ path: '', component: BaseComponent },
```

Attribuer un composant pour un chemin :

```
{ path: 'actu', component: ActuComponent },
```

Passer un paramètre à l'adresse :

```
{ path: 'actu/:id', component: ActuComponent },
```

Ce paramètre pourra ensuite être récupéré dans le composant attaché sous la forme id=n.

Nous pouvons aussi gérer les erreur 404

```
{ path: '**', component: Erreur404Component }];
```

Les routes & le HTML

router-outlet

L'affichage du résultat d'une route se fait dans une balise spéciale. Cette balise peut être ajoutée à tout fichier HTML qui devra recevoir les résultats d'une route :

```
<router-outlet></router-outlet>
```

Cette balise est généralement ajoutée au minimum dans le HTML du composant racine mais

est utile aussi pour des routes enfants.

Activer une route

Dans le template peuvent être utilisés des liens pour naviguer :

```
<a href routerLink=”/news”>NEWS</a>
```

Dans cet exemple, le slash de "/news" permet de réécrire l'URL à partir de la racine du site.

Ces liens peuvent être augmentés de paramètres :

```
<a href=" [routerLink]="['/actu', 1]">Actualité</a>
```

Nous créons un tableau en injectant un paramètre qui transforme l'adresse en /actu/1. Si nous reprenons l'exemple précédent du tableau des routes, 1 sera la valeur de l'ID de la route 'actu/:id'.

Routes, paramètres & composants

Deux classes nous intéressent ici : ActivatedRoute permet de récupérer les paramètres transmis dans l'URL à l'intérieur du composant et Router permet de naviguer vers des pages.

ActivatedRoute

On importe la classe ActivatedRoute et on l'instancie dans le constructeur.

```
import { ActivatedRoute } from '@angular/router';
constructor(private routeParams:ActivatedRoute)
```

On peut ensuite souscrire à la mise à jour de l'adresse :

```
ngOnInit() {
  this.routeParams.params.subscribe(params => {
        this.newsId = +params['id']; // (+) convertit 'id' en nombre
   });
}
```

Activer une adresse dans le composant

On peut aussi ouvrir une adresse en important la classe Router et en l'instanciant de la même manière dans le constructeur. Nous pouvons ensuite activer la navigation depuis le composant et passer des paramètres :

```
this.router.navigate(['/product-details', id]);
```

Les services & l'appel de données

Les services sont des injectables. Ils sont dédiés à être partagés entre différents composants et donc particulièrement adaptés au chargement de données externes.

Pour mémoire, les services sont effectivement instanciés dans le tableau des providers du module qui le déclare

Créer un service

La commande est simple :

```
ng generate service monServ
```

ou

```
ng g s monServ
```

Notre service sera automatiquement injecté dans le module, dans le tableau "providers". Il peut être ensuite instancié dans des composants :

```
constructor(private donnees:AnewsService){}
```

Il peut ainsi être utilisé dans le code, ses propriétés et méthodes publiques peuvent ainsi devenir accessibles au composant :

```
this.actuId = this.donnees.getInfos();
```

mais aussi dans le template du composant :

```
<article>{{donnees.article}}</article>
```

C'est redoutablement efficace pour une gestion centralisée des données.

Appeler des données externes

Plusieurs mécaniques sont nécessaires au chargement et à l'exploitation des données. HttpClient fournie par Angular est la classe de base pour l'appel de données en Ajax. Elle doit être importée et instanciée. Voici un exemple :

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

```
constructor(private http:HttpClient) {
    this.chargeDonnees();
}
```

```
chargeDonnees(){
    this.http.get<Nouvelles[]>('../assets/datas/nouvelles.json')
    .subscribe(data => {
            console.log(data);
    });
```

```
}
```

La classe HttpClient est importée et instanciée dans le constructeur (http:HttpClient). Une méthode lance un appel aux données sur un fichier JSON et souscrit à la réponse à ce chargement pour récupérer les données de manière asynchrone. Lorsqu'elles sont chargées, nous les affichons dans la console.

Dans cet exemple, nous utilisons une interface pour typer les donner reçues : get<Nouvelles[]> ne pourrons recevoir QUE un tableau de Nouvelles sous peine d'erreur...

Dans cet exemple, nous avons souscrit à une méthode get de HttpClient. Cette méthode renvoie un Observable.

Les observables

BehaviorSubject

Subscription & ngOnDestroy

Les observables sont des classes tirées de RxJS. Elles sont intégrées à Angular nativement pour gérer la synchronisation des données à l'extérieur et dans une application. Ils correspondent à une évolution des outils de syncrhonisation et visent à simplifier et augmenter l'utilisation de écouteurs et des promesses.

RxJS porte une profusion de classes et méthodes qui représentent une évolution significative pour la gestion des données d'une application. Elles peuvent être triées, partagées, synchronisées, filtrées... Dans ce cours, nous n'aborderons que quelques concepts simples utiles au développement rapide d'applications. Nous vous invitons à vous plonger dans l'API et les tutoriels directement sur les site du framework (en anglais).

POUR UN TOUR D'HORIZON

LE MANUEL

L'idée des observables est de construire une forme de stream de données entre une source et une réception. Une requête appelle des données, des objets y souscrivent pour recevoir les résultats sous forme de collections.

Par l'exemple

```
const monObservable = Observable.of("On me regarde");
monObservable.subscribe((value) => console.log(value););
```

Dans cet exemple, nous créons un observable et utilisons un émetteur pour lancer des données et simuler les chargements habituels de données. Nous attachons une souscription à l'observable pour traiter les données transmises.

Rxjs propose tout un ensemble de classes et de méthodes. Il est possible de tout importer d'un coup mais nous recommandons de les importer au cas par cas en fonction de vos besoins.

Dans une utilisation plus orientée application, la souscription peut être menée depuis un ou des composants différents. Les données sont ainsi dispatchées dans toute l'application en temps réel. Exemple :

```
@Injectable()
export class DonneesService {
    const monObservable = Observable.of("On me regarde");
}
```

```
export class ParentComponent implements OnInit {
    articlesParent:Array<string>;
    constructor(private donnees:DonneesService){};
    ngOnInit() {
            this.donnees.monObservable.subscribe(
                    data => this.articlesParent = data;
            );
    }
}
```

```
export class EnfantComponent implements OnInit {
    articlesEnfant:Array<string>;
    constructor(private donnees:DonneesService){};
    ngOnInit() {
            this.donnees.monObservable.subscribe(
                    data => this.articlesEnfant = data;
            );
    }
}
```

Avec cette mécanique, nous nous connectons à l'observable du service pour obtenir ses données depuis n'importe quelle classe de notre application.

Bon, Angular étant ce qu'il est, dans ce cas de figure, il est tout aussi intéressant de partager un objet ou un tableau du service directement.

BehaviorSubject

Une application Angular est une application Javascript dans du HTML. Son chargement et le traitement de ses données se fait après le chargement du HTML. Nous avons déjà vu l'incidence de ce chargement sur les images en particulier ([src]).

Au niveau des observables le problème peut être le même. Les données qu'il donnera à synchroniser lanceront une erreur tant que les données ne seront pas chargées. Pour éviter ce problème nous pouvons utiliser les BehaviorSubject.

Le BehaviorSubject est un observable qui s'instancie avec des valeurs par défaut. Ces valeurs évitent l'émergence d'erreurs :

```
const monBehavior:BehaviorSubject<Array<number>> = new BehaviorSubject<Array<number>>([]);
```

Nous créons ici un observable typé en tableau de nombres qui lancera une valeur vide à l'instanciation évitant les erreurs de console sur des données inexistantes.

Subscription & ngOnDestroy

Les observables sont des routines... d'observation qui tournent en arrière plan. Lorsqu'un composant est supprimé dans un changement de route, l'observable continue à tourner en arrière plan. Il convient donc de le supprimer pour libérer la mémoire et éviter les problèmes des performance.

En termes fonctionnels, nous attacherons un observable à un objet 'Subscription' qui pourra être supprimé lors de la destruction du composant. Pour mémoire, le cycle de vie d'un composant se conclut par l'événement OnDestroy lancé au moment de la suppression du composant. Voici comment s'y prendre dans un composant :

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs/Subscription';
import { DatasService } from '../services/datas.service';

export class NewsImcComponent implements OnInit, OnDestroy {
 actus:string[];
 ecoute:Subscription;

 constructor(public donnees:DatasService) { }
```

```
  ngOnInit() {
   this.ecoute = this.donnees.newsSub.subscribe(
   data => {
    this.actus = data;
   });
  }
  // Nettoyer l'écoute
  ngOnDestroy(){
   this.ecoute.unsubscribe();
  }
}
```

Une exemple simple et évocateur. Nous attachons la souscription à un objet 'Subscribe'. Cette souscription peut ensuite être supprimée dans l'événement ngOnDestroy...

Comme nous l'avons suggéré plus haut, notez que nous n'importons que la classe Subscription de RxJS, ici aussi pour éviter de surcharger de classes inutiles dans le contexte.

Simple mais nécessaire.

Les pipes

Pipes Angular

Pipes personnalisés

Les pipes sont des sortent de filtres qui agissent sur les données pour les traduire dans différents formats. Nombre sont proposés par Angular. Des pipes peuvent être créés pour opérer ses propres tris.

Utilisés essentiellement dans les templates HTML, ils sont bien pratiques pour le tri de données en revanche, il faut envisager qu'ils sont traités côté utilisateur. Leur utilisation permet d'alléger des traitements côté serveur mais alourdit les traitement côté client. L'équilibre doit être trouvé en fonction des situations.

Angular propose un ensemble de Pipes par défaut pour le traitement de valeurs. Ils peuvent être personnalisés pour des tris plus spécifiques.

Pour utiliser un Pipe dans du HTML, il faut utiliser le caractère du même nom (en anglais

Pipes Angular

Majuscules, minuscules, titre

Des traitements communs peuvent être opérés sur des chaînes de caractères.

Mettre une chaîne en majuscule

```
{{ 'Superman assure' | uppercase }}
```

… en minuscules

```
{{ 'Mais il TOUT PETIT !!!' | lowercase }}
```

… et mettre des majuscules sur la première lettre des mots :

```
{{ 'même quand il assure LIKE A BOSS' | titlecase }}
```

Number

Des traitements communs peuvent être opérés sur des chiffres aussi.

Number est basé sur trois dimensions :

- les entiers : integerDigits
- les fractions minimums : minFractionDigits
- les fractions maximum : maxFractionDigits

```
{{ '12345' | number }} > renvoie 12,345
{{ '12345' | number:'6.' }} > renvoie 012,345
{{ '12345' | number:'.2' }} > renvoie 12,345.00
{{ '12345.11' | number:'.1-1 }} > renvoie 12,345.1
{{ '12345.18' | number:'.1-1 }} > renvoie 12,345.2
```

Pourcentages

Ce pipe fera un calcul de pourcentage, genre un produit en croix impossible à faire :

```
{{ 0.8 | percent }} > renvoie 80%
{{ 0.8 | percent:'.3' }} > renvoie 80.000%
```

Currency

Curency s'intéresse à … la monnaie...

```
{{ 10.6 | currency:'CAD' }} > renvoie 'CA$10.60'
```

```
{{ 10.6 | currency:'CAD':'symbol-narrow' }} > renvoie '$10.60'
{{ 10.6 | currency:'EUR':'code':'.3' }} > renvoie 'EUR10.600'
```

Date

Le traitement des dates se base sur les mêmes règles que dans les autres langages. Les variables consommées peuvent être en millisecondes ou un objet Date :

```
{{ naissance | date:'dd/MM/yyyy' }} > 05/03/1970
{{ naissance | date:'longDate' }} > July 16, 1986
```

Ca marche aussi pour les heures :

```
{{ naissance | date:'HH:mm' }} > 15:30
{{ naissance | date:'shortTime' }} > 3:30 PM
```

Slice

Le slice en pipe agit peu ou prou comme son parent en Javascript.

Dans un tableau, affichons les premiers éléments

```
{{ monTableau | slice:0:3 }}
```

Ca marche aussi sur des chaînes

```
{{ 'Ma chaîne de caractères | slice:3:9 }}
```

On peut jouer jusqu'à la fin ou à partir de la fin

```
{{ monTableau | slice:3 }}
{{ monTableau | slice:-2 }}
```

Il marche aussi avec ngFor

```
<div *ngFor="let t of tab | slice:0:2">{{ t.nom }}</div>
<div *ngFor="let t of tab | slice:0:taille">{{ t.nom }}</div>
```

Et aussi avec un alias

```
<p *ngFor="let t of tab| slice:0:2 as total">{{ total.length }} : {{ t.nom }}</p>`
```

JSON

On peut obtenir des valeurs en version json

```
<p>{{ tab | json }}</p>
```

pourrait renvoyer

```
<p>[ { "titre": "Super" }, { "titre": "man" } ]</p>
```

ASYNC

Async se base sur PromisePipe ou ObservablePipe pour afficher les donnée de… promesses et observables. Il attend que les données soient disponible pour les afficher.

```
<div>{{ salut | async }}</div>
```

```
export class SalutComponent {
    salut = new Promise(resolve => {
            setTimeout(() => resolve('Coucou'), 1000);});
}
```

Pipes personnalisés

Les pipes personnalisés sont résolus par la méthode transform() de la classe PipeTransform. Ils peuvent être générés pas Angular CLI. Un Pipe se déclare (tableau de déclarations) dans un module :

```
import { TriTableauPipe } from './tri-tableau.pipe';
```

```
declarations: [
  ...
   TriTableauPipe,
  ...
 ],
```

Le nom du Pipe est utilisé dans le HTML pour appliquer un filtre. Par exemple dans un formulaire :

```
<input name="tripays" [(ngModel)]="tri">
<label for="pays">De quel pays êtes-vous ?</label>
<select name="pays" multiple>
    <option>Vos vacances seront ?</option>
    <option *ngFor="let p of (pays | triTableau : tri)">{{p}}</option>
</select>
```

Dans cet exemple, nous trions une liste 'select' de pays en utilisant la valeur d'un champ input comme chaîne de tri.

Pour cet exemple nous avons créé un Pipe personnalisé :

```
ng generate pipe TriTableauPipe
```

...qui nous donnera un fichier de ce type, à peine modifié depuis la base générée :

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

@Pipe({
```

```
  name: 'triTableau'
})
export class TriTableauPipe implements PipeTransform {
  public transform(values: any[], filter: string): any[] {
    if (!values || !values.length) return [];
    if (!filter) return values;

    return values.filter(v => v.toLowerCase().indexOf(filter.toLowerCase()) >=
0);
  }
}
```

Dans ce exemple, nous créons une classe avec un décorateur de Pipe. Ce décorateur reçoit une chaine 'name' qui permet d'identifier le Pipe dans le HTML. C'est le nom que nous utilisons dans la boucle HTML.

La classe TriTableauPipe utilise la méthode Transforme de la classe PipeTransform pour renvoyer des valeurs :

s'il n'y a pas de valeurs insérées, le filtre renvoie un tableau vide :

```
if (!values || !values.length) return [];
```

si le Pipe ne reçoit pas de filtre (une chaîne de caractère), il renvoie toutes les valeurs :

```
if (!filter) return values;
```

Enfin, dans tous les autres cas, il renvoie un tableau en appliquant un filtre, ici, un test de présence de chaîne (indexOf) dans une boucle sur le tableau des valeurs (values.filter).

```
return values.filter(v => v.toLowerCase().indexOf(filter.toLowerCase()) >=
0);
```

Ainsi, dans cet exemple, le Pipe reçoit la tableau des pays du 'select' en argument 'values' et utilise la valeur du champ de l'input comme filtre (une chaîne de caractères). Nous verrons plus loin comment utiliser ngModel pour relier une valeur de composant à des champs de formulaire.

Un autre exemple ?

Ce Pipe nous permet d'établir un filtre dans un objet spécifique pré-déterminé :

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

@Pipe({ name: 'filtreObj' })
export class AppObjPipe implements PipeTransform {
  public transform(values: any[], filtre: string): any[] {
    if (!values || !values.length) return [];
    if (!filtre) return values;
    return values.filter(v => {
```

```
      if(v.laVariableAFiltrer){
      return v.laVariableAFiltrer.indexOf(filtre) >=0 ;
    }
   });
}}
```

Dans cet exemple nous reprenons la même mécanique, le choix opéré sur le tri se fait sur une valeur brute (laVariableAFiltrer) qui correspond à une variable d'objet. Ce Pipe nous permet donc de trier un objet spécifique sur la même mécanique que le précédent.

Sécuriser une application

Guards

Intercepteurs

Une application Javascript est téléchargée côté client. La minification aide à limiter la lisibilité du code mais ne constitue en rien une protection. Angular a intégré plusieurs mécaniques pour assurer cette sécurité, les guards et les intercepteurs.

Les premiers agissent sur les routes pour donner des accès ou télécharger des modules. Les deuxièmes lisent et réécrivent les requêtes HTTP pour en valider notamment les headers. Une mécanique particulièrement intéressante pour

Les guards

CanActivate

La méthode CanActivate des gardes est dédié au blocage et re-routage d'un lien.

Elle se relie à un service pour gérer les limites d'accès. Usuellement il renvoie un booléen ou une redirection. Si le booléen est true, la route sera activée. C'est particulièrement utile pour protéger un accès en fonction du niveau d'identification. Ce processus agit en conjonction avec les intercepteurs pour garantir la sécurité d'une application, nous verrons ça plus loin.

Créer une guard

Comme d'habitude, CLI va nous aider :

```
ng generate guard securise
```

ou

```
ng g g securise
```

Voici le fichier généré :

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from
'@angular/router';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class SecuriseGuard implements CanActivate {
 canActivate(
   next: ActivatedRouteSnapshot,
   state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |
boolean {
   return true;
 }
}
```

Une guard est un injectable, comme un service, elle doit se déclarer dans un tableau 'providers' de module. Si vous avez choisi de créer votre application avec l'option -routing vous aurez à votre disposition un module spécifique intégrant le tableau des routes. Nous déclarerons notre guard dans son provider, sinon, déclarez le dans le module de base :

```
import { SecuriseGuard } from './securise.guard';
@NgModule({
    ...
```

```
    providers:[ SecuriseGuard ]
})
```

Notez que la méthode 'canActivate' est typée en observable, ou promesse, ou booléen mais en tout état de cause, elle renvoie un booléen. A l'intérieur plusieurs méthodes sont actives :

» next : renvoie un 'ActivatedRouteSnapshot', une interface permettant de naviguer dans un arbre de routes ;

» state : utilise RouterStateSnapshot pour connaître l'état de la route.

Ces deux mécaniques servent à interagir à la volée avec la route sur laquelle la guard est attachée. La route ne sera déclenchée que si le retun est égal à true aussi, c'est dans cette méthode 'CanActivate' que doivent être menées les opérations de test sur l'autorisation de connexion.

Vérifier une identification

L'identification dépend de paramètres variables. Nous nous contenterons de considérer un service spécifique qui pourra être manipulé par des composant.

Nous pouvons créer un nouveau service et lui attribuer une valeur booléenne :

```
ng g s connexion
```

Notre nouveau service (à déclarer dans un tableau de 'providers'...) sera partagé avec les composants utiles qui récupèreront la valeur du booléen d'identification.

```
import { Injectable } from '@angular/core';
@Injectable()
export class ConnexionService {
 connecte:boolean = false;
 constructor() { }
}
```

Nous pouvons l'instancier dans un composant de connexion :

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms/src/directives/ng_form';
import { ConnexionService } from './connexion.service';

@Component({
 selector: 'app-connexion',
 templateUrl: './connexion.component.html',
 styleUrls: ['./connexion.component.css']
})
export class ConnexionComponent {
 donneesID = {login:'', mdp:''};

 constructor(public connexionService:ConnexionService) { }
}
```

Notre composant est on ne peut plus simple. Nous importons trois classe :

» component : nous faisons un composant ;

» NgForm : nous aurons besoin d'un formulaire mais ne nous attarderons pas sur cette classe qui sera largement détaillée plus loin ;

» ConnexionService : le service de connexion avec son booléen 'connecte'.

Nous ajoutons un objet pour connecter les champs 'identifiant' et de 'mot de passe' du formulaire. La suite se passe dans le HTML avec une version plus que rustique de modification de notre valeur. Voici :

```
<form (ngSubmit)="ConnexionService.connecte = true">
        <div class="form-group">
          <label for="login">Identifiant</label>
          <input id="login" class="form-control" name="login" [(ngMo-
del)]="donneesID.login" required>
        </div>
        <div class="form-group">
          <label for="mdp">Mot de passe</label>
          <input id="mdp" class="form-control" type="password"
name="mdp" [(ngModel)]="donneesID.mdp" required>
        </div>
</form>
```

Notre formulaire est assez ridicule dans l'idée d'une sécurisation, tous les codes renveront un code positif (ngSubmit... = true) néamoins, il nous permet de mettre en exergue la relation entre un composant, les instanciations de services et leur accès direct depuis le HTML. La vue (le HTML) est connectée à la Vue-Modèle (le composant), lui même en interaction avec le modèle (le service).

Cliquer sur le bouton de soumission déclenchera la fonction ngSubmit qui modifiera la valeur de 'connecte' dans le service ConnexionService.

Nous reprendrons l'exemple de ce formulaire plus bas comme exemple et en ferons quelque chose de plus solide.

D'autres guards ?

Trois autres guards existent. Nous les présentons sans développer plus en détail :

CanActivateChild

Applique le même effet que canActivate mais sur les composants enfants.

CanDeactivate

Ce guard permet d'empêcher de quitter une page. C'est notamment utile pour attendre la validation et d'un formulaire par exemple.

CanLoad

Un outil pour un chargement en arrière plan ou à la demande. Il se coordonne avec un attribut 'loadChildren' d'une route. C'est aussi un moyen de sécuriser certaines parties d'un code en ne le chargeant que lorsqu'il est rendu accessible par une identification par exemple. La publication génèrera un fichier JS supplémentaire.

```
{path: 'edition', loadChildren: 'app/edition/edition.module#EditionModule',
canLoad: [SecuriteGuard]},
```

Les intercepteurs

Angular intègre des intercepteurs de requêtes HTTP dans les classes http. Ils permettent de créer un service qui filtrera les requêtes et permettra d'en traiter les données, notamment les entêtes pour l'authentification et les status. Ils demandent un ensemble d'opérations de création et de déclaration et s'appuient sur diverses classes pour participer au processus d'envoi et de réception de requêtes : **HttpEvent**, **HttpInterceptor**, **HttpHandler**, **HttpRequest**.

Les intercepteurs sont réapparus à partir de la version 4.3.1 de Angular. Ils étaient présents dans la version 1 mais souffraient de problèmes de performance. Ils sont pour autant si essentiels qu'ils ont été rétablis.

Créer un intercepteur

A ce jour, Angular CLI ne permet pas de créer un intercepteur. Il faut donc le créer à la main. Commençons par intégrer ses classes dans le module de général (app-module) :

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

providers: [
    { provide: HTTP_INTERCEPTORS, useClass:IntercepteurService, multi: true }
],
```

Notez la définition de l'intercepteur dans le tableau des providers. Il prend une forme étendue. Cette forme est acceptable aussi pour les autres déclarations.

Cet intercepteur va gérer les erreurs 401 d'authentification renvoyée par le serveur.

Nous pouvons créer une classe pour gérer la base de notre intercepteur et y copier notre code

```
ng g class services/auth-intercepteur
```

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest, HttpErrorResponse } from '@angular/common/http';
import { Observable, pipe } from 'rxjs';
import {catchError} from "rxjs/internal/operators";

@Injectable()
export class SecuriteIntercepteur implements HttpInterceptor {

 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return next.handle(req)
  .pipe(
    catchError((error) => {
     if (error instanceof HttpErrorResponse && error.status == 401) {
      console.log("Une erreur s'est produite");
      console.log(error); // Récupérer les erreurs et les affichers
      return Observable.throw(error); // Renvoyer l'erreur
     }
    })) as any;
  }
}
```

Quelques explications ?

Tout le système est concentré dans la méthode 'intercept' de la classe HttpInterceptor. Elle renvoie un observable et plus précisément, une copie modifiée d'un observable. Cette méthode prend plusieurs arguments :

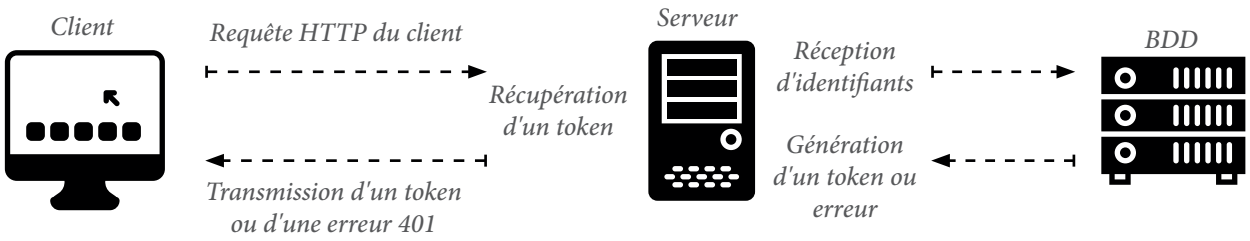
- req : une requête interceptée ;
- next : un gestionnaire de requêtes qui renverra la copie de notre observable.

Avec ces quelques éléments nous comprennons ce qui se trame. Une requête est adressée. L'intercepteur l'intercepte (...) en fait une copie, modifie la copie et renvoie la copie à la place de la requête initiale. CQFD.

Dans notre cas, nous interceptons les status d'une requête pour identifier des erreurs d'authentification renvoyées par le serveur (erreur 401) et l'affichons dans la console. C'est ici que nous pourrons penser la mécanique de gestion de erreurs en renvoyant une erreur à l'observable ou en affichant une erreur dans l'interface.

JWT, un peu plus de sécurité

Les JWT pour Javascript Web Token sont une mécanique largement utilisée pour sécuriser les application Web. Voici un schéma de fonctionnement :



Explications :

1 - Le client envoie une requête au serveur avec des identifiants ou un token d'authentification dans l'entête de la requête ;

2 - Le serveur récupère le token et renvoie les données avec le token en entête ;

3 - Si aucun token, sollicite la base de données et compare les résultats d'identification ;

4 - Si identifié, génère un token avec une clé d'identification secrête présente uniquement sur le serveur et une donnée transmise (l'identifiant par exemple). L'algorithme est un système d'encodage mélangeant les chaînes de caractères ;

5 - Renvoie alors un token d'authentifion ou une erreur 401 ;

6 - Au niveau d'Angular, nous récupérons les entêtes pour vérifier la présence d'un token ou d'une erreur 401. Si un token est présent, tous les entêtes intègreront le nouveau token pour sécuriser chacun des échanges avec le serveur.

POUR EN SAVOIR PLUS SUR LES JWT

Normalement avec cette mécanique les échanges seront sécurisés.

Le premier intercepteur que nous avons vu permet de vérifier les entêtes et de capter les erreurs 401 d'authentification. Voici un exemple d'intercepteur pour intégrer un token d'authentification à chacune de nos requêtes.

Un intercepteur d'authentification

La mécanique est la même que précédemment, nous changeons seulement l'interprétation et la gestion des échanges en intégrant le service de connexion pour récupérer et transmettre des données.

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';
import { Observable, pipe } from 'rxjs';
import {catchError} from "rxjs/internal/operators";
@Injectable()
export class SecuriteIntercepteur implements HttpInterceptor{
 /**
 * Interepteur qui ajouter un token d'identification à chaque requête HTTP sortante
  * L'intercepteur clone un requête, transforme la requête clonée et l'envoie */
 constructor() { }
 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  console.log("Interception d'une requête ... ");
  // Cloner la requête et injecter le token d'authentification
  const authReq = req.clone({ headers: req.headers.set("Authorization", "AUTH_TOKEN")});
  console.log("La requête va être envoyée avec un nouveau header intégrant une autorisation...");
  // Envoyer la nouvelle requête
  return next.handle(authReq)
  .pipe( catchError((error) => {
  console.log("Une erreur s'est produite");
  console.log(error); // Récupérer les erreurs et les affichers
  return Observable.throw(error); // Renvoyer l'erreur
  })) as any;
```

```
  }
}}
```

La chaîne AUTH_TOKEN doit ici être remplacée par le token reçu de la part du serveur pour l'intégrer à chaque requête HTTP. Nous avons instancié le service ConnexionService dans cet objectif. Avec cette technique, chaque requête est sécurisée sans autre préoccupation. Le token peut être enlevé lors d'une déconnexion.

Un petit plus allégé

Pour les petites applications il est possible de gérer les entêtes directement au niveau des requêtes sans utiliser d'intercepteur.

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

let headers = new Headers();
headers.append('Content-Type', 'application/json');
headers.append('Authorization': api_token);
let options = new RequestOptions({ headers: headers });
return this.http.get(url, options);
```

Les formulaires

La gestion des formulaires est l'une des principales légitimation d'un tel framework. Angular offre des outils d'automatisation ou de simpliciation de nombre de tâches pour la gestion des formulaires. Par exemple :

- le binding des données entre Javascript et le formulaire ;
- la gestion des événements ;
- la gestion des CSS et des interactions ;
- la transmission simplifiée des données ;
- ...

Deux méthodes permettent de gérer les formulaires : la méthode orientée template et la méthode orientée code. Les deux sont assez similaires.

Les formulaires orientés template

En premier lieu, il est nécessaire d'importer le module FormModule dans un module pour donner accès aux classes de gestion des formulaires :

```
import { FormsModule } from '@angular/forms';
imports: [
    ...
    FormsModule,
    ...
 ],
```

Nous avons maintenant accès aux classes de formulaires et pouvons les importer dans nos composants. Il s'agit des classes : FormControl et FormGroup.

FormControl est la plus petite unité du formulaire. Il étend les champs pour en contrôler le fonctionnement avec quelques méthodes et propriétés.

FormGroup se distingue légèrement par sa capacité à gérer des groupes de champs.

FormControl

FormControl étend les champs de formulaires en leur donnant accès à des propriétés et méthodes complémentaires :

- **valid** : si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.
- **invalid** : si le champ est invalide, au regard des contraintes et des validations qui lui sont appliquées.
- **errors** : un objet contenant les erreurs du champ.
- **dirty** : false jusqu'à ce que l'utilisateur modifie la valeur du champ.
- **pristine** : l'opposé de dirty.
- **touched** : false jusqu'à ce que l'utilisateur soit entré dans le champ.
- **untouched** : l'opposé de touched.
- **value** : la valeur du champ.
- **valueChanges** : un Observable qui émet à chaque changement sur le champ.
- **hasError()** : pour savoir si le contrôle a une erreur donnée.

FormGroup

FormGroup s'occupe des groupes de champs avec les mêmes propriétés et méthodes plus quelques bonus :

» **valid** : si tous les champs sont valides, alors le groupe est valide.

» **invalid** : si l'un des champs est invalide, alors le groupe est invalide.

» **errors** : un objet contenant les erreurs du groupe, ou null si le groupe est entièrement valide. Chaque erreur en constitue la clé, et la valeur associée est un tableau contenant chaque contrôle affecté par cette erreur.

» **dirty** : false jusqu'à ce qu'un des contrôles devienne "dirty".

» **pristine** : l'opposé de dirty.

» **touched** : false jusqu'à ce qu'un des contrôles devienne "touched".

» **untouched** : l'opposé de touched.

» **value** : la valeur du groupe. Pour être plus précis, c'est un objet dont les clé/valeurs sont les contrôles et leur valeur respective.

» **valueChanges** : un Observable qui émet à chaque changement sur un contrôle du groupe.

» **hasError()** : pour savoir si le groupe de contrôle a une erreur donnée.

» **get()** : pour récupérer un contrôle dans le groupe.

Au niveau du composant

La classe ngForm nous donne accès à ces propriétés au niveau du composant. Il nous faut l'importer :

```
import { NgForm } from '@angular/forms/src/directives/ng_form';
```

Nous avons ainsi accès à toutes les fonctions des formulaires. Nous pourrons :

» gérer les données avec un double binding ;

» gérer la transmission des données avec des variables locales ;

» assurer la gestion des validations et des erreurs directement dans le HTML.

Gérer les données avec un double binding

Le binding peut utiliser plusieurs méthodes en fonction des objectifs recherchés.

NgModel

NgModel nous permet d'assurer un binding avec le composant :

```
<input required name="titre" [(ngModel)]="titre">
```

Nous définissons un champ input, requis dans le formulaire, c'est du HTML et nous appliquons un double binding avec [(ngModel)].

Pour mémoire, les [] pour écrire les données, les () pour les transmettre.

Toutes les données du composant correspondant à l'identifiant du ngModel seront écrites dans

le champ (ici 'titre'). Tout changement dans le champ mettra à jour automatiquement les données du composant.

Transmission des données & variables locales

Tester des champs internes

Les variables locales vont nous permettre de gérer les échanges de données à l'intérieur du formulaire. C'est une mécanique qui permettra de gérer les états et de transmettre l'ensemble des données sur un ngSubmit.

Par exemple :

```
<input id="titre" required name="titre" [(ngModel)]="titre" #titreCtrl="ng-Model">
<div *ngIf="titreCtrl.dirty" id="titre-warning">Un titre est requis</div>
```

Dans cet exemple nous définissons une variable locale **#titreCtrl** à laquelle nous attachons la classe 'ngModel'. La DIV suivante identifie cette variable pour connaître l'état du champ et évaluer si des données ont été saisies. La condition '*ngIf' permet d'écrire ou non la DIV en fonction de l'état du champ.

Nous n'avons pas besoin d'autre code. Avec cet exemple, les données sont mises à jour automatiquement dans les deux sens et la validité du champ est testée directement dans le formulaire.

Transmettre les données du formulaire

Les données du formulaire peuvent être traitées aussi simplement avec une variable locale :

```
<form (ngSubmit)="gereFormulaire(f)" #f="ngForm" *ngIf="!authService.connecte">
<button class="btn btn-primary" type="submit" [disabled]="f.invalid">Se connecter !</button>
</form>
```

Ici nous créons une variable locale #f augmentée de la classe ngForm. Cette classe est partagée avec le composant. La variable locale est passée en argument de la fonction 'gereFormulaire' du composant. Toutes les données lui sont transmises.

En bonus, nous ajoutons une condition pour masquer le formulaire si la personne est déjà identifiée en intégrant l'instance du service authService du composant.

Le bouton de type submit déclenche l'action du formulaire. Nous en profitons pour réutiliser la variable locale #f pour vérifier sa validité. Le bouton ne devient actif que lorsque le formulaire est correctement rempli. Aucune action de soumission ne peut être entreprise en attendant.

Voyons maintenant côté composant comment ça se passe. Il nous faut d'abord importer les classes idoines :

```
import { NgForm } from '@angular/forms/src/directives/ng_form';
```

```
import { AuthService } from '../services/auth-imc.service';
export class ConnexionImcComponent implements OnInit {
 donneesID = {login:'', mdp:''};

 constructor(public authService:AuthService,) { }
    ngOnInit() {}
    gereFormulaire(f:ngForm){
            console.log(f);
    }
}
```

Dans ce composant, la fonction gereFormulaire() reçoit et affiche les valeurs du formulaire.

Digression vers les styles

Les systèmes de gestion des champs de formulaire permettent très simplement de modifier leur apparence. Lors des interactions avec un formulaire vous pourrez observer que les styles CSS sont ajoutés ou enlevés dynamiquement à la volée en fonction des interactions. Vous pourrez voir apparaître dans le code des pristine, touched, dirty au fil des clics. Ainsi, des styles CSS peuvent être ajoutés à votre application pour générer des effets visuels automatiques. Par exemple :

```
input.ng-invalid {
    border: 3px red solid;
}
```

Un champ invalide sera bordé de rouge.

```
input.ng-invalid.ng-dirty {
    border: 3px yellow solid;
}
```

Le champ devient jaune lorsque le champ est invalide mais que des données ont été saisies.

```
input.ng-valid.ng-dirty {
    border: 3px green solid;
}
```

Enfin, si les données sont valides, le champ devient automatiquement vert.

Gérer des validations et des erreurs

Les validations

Quelques validateurs sont fournis par le framework et nous aident à identifier l'état du formulaire et de ses champs. Ces validations sont directement utilisables dans les champs avec les variables locales de contrôle et les CSS :

Validators.required pour vérifier qu'un contrôle n'est pas vide ;

Validators.minLength(n) pour s'assurer que la valeur entrée a au moins n caractères ;

Validators.maxLength(n) pour s'assurer que la valeur entrée a au plus n caractères ;

Validators.email() (disponible depuis la version 4.0) pour s'assurer que la valeur entrée est une adresse email valide (bon courage pour trouver l'expression régulière par vous-même…) ;

Validators.pattern(p) pour s'assurer que la valeur entrée correspond à l'expression régulière définie.

Peut ainsi fixer des valeurs de validation pour les champs :

```
<input name="nom" required minlength="8">
```

Ici le champ doit avoir au minimum huit caractères pour être valide. Pratique pour forcer des identifiants ou des mots de passes.

Les erreurs

Nous avons deux méthodes principales pour traiter les erreurs. La première est tirée de la variable locale attachée à la classe NgModel et permet de vérifier la validité des données des champs :

```
<button type="submit" [disabled]="f.invalid">Let's Go!</button> // A mettre en parallèle avec l'exemple précédent
```

La méthode hasError() permet de vérifier un champ :

```
<input id="titre" required name="titre" minlength="8" [(ngModel)]="titre" #titreCtrl="ngModel">
<div *ngIf="titreCtrl.dirty && titreCtrl.hasError('required')" id="titre-warning">Un titre est requis</div>
```

Dans ce dernier exemple, nous enrichissons le traitement des erreurs par une validation dynamique de toutes les conditions requises.

Les formulaires orientés composants

Cette manière de gérer les formulaires permet de créer des formulaires plus dynamiques et plus évolutifs mais plus verbeux.

Les méthodes FormControl et FormGroup gèrent leur création. Elles sont membres de la classe FormBuilder :

```
import { FormBuilder } from '@angular/forms';
constructor(private formBuild:FormBuilder){}
```

A partir de cette base, nous pouvons recréer un formulaire :

```
export class FormComponent {
    connexionForm: FormGroup;
    constructor(formBuilder: FormBuilder) {
            this.connexionForm = formBuilder.group({
                    id: formBuilder.control(''),
                    mdp: formBuilder.control('')
            });
    }
    connexion() {
            console.log(this.connexionForm.value);
    }
}
```

Avec un formulaire géré au niveau du template nous utilisons la classe NgForm. Ici nous utiliserons ReactiveFormsModule. La directive formGroup nous permettra de relier notre formulaire au composant et formControlName de relier les champs :

```
<h2>S'identifier</h2>
<form (ngSubmit)="connexion()" [formGroup]="connexionForm">
    <label>Identifiant</label>
    <input formControlName="id">
    <label>Mot de passe</label>
    <input type="password" formControlName="mdp">
    <button type="submit">Se connecter</button>
</form>
```

Le HTML est assez simple :

- formControlName : donne un identifiant pour le code afin d'appliquer les traitements ;
- ngSubmit : déclenche la méthode connexion.

Nous pouvons aller un tout petit peu plus loin dans un formulaire. Voici un exemple complet pour un formulaire de connexion :

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

@Component({
selector: 'app-connexion',
templateUrl: 'conne-form.component.html'
})

export class FormComponent {
    idCtrl: FormControl;
    mdpCtrl: FormControl;
    connexionForm: FormGroup;
    constructor(fb: FormBuilder) {
            this.idCtrl = fb.control('');
            this.mdpCtrl = fb.control('');
            this.connexionForm = fb.group({
                    id: this.idCtrl,
                    mdp: this.mdpCtrl
            });
    }

    reset() {
            this.idCtrl.setValue('');
            this.mdpCtrl.setValue('');
    }

    enregistrement() {
            console.log(this.connexionForm.value);
    }
}
```

Traitement des validations côté composant

Les contrôles s'ajoutent en arument dans le contrôle du FormBuilder :

```
id: fb.control('', [Validators.required, Validators.minLength(3)]),
```

On retrouve aussi bien entendu :

- Validators.minLength(n)
- Validators.maxLength(n)
- Validators.email()
- Validators.pattern(p)

Traitement des erreurs côté composant

Les erreurs peuvent être surveillées grâce à la méthode get()

```
<div *ngIf="connexionForm.get('id').dirty && connexionForm.get('id').
hasError('required')">Un identifiant est requis</div>
```

Relativement simple.

POUR ALLER PLUS LOIN, NOUS CONSEILLONS CE TUTORIEL

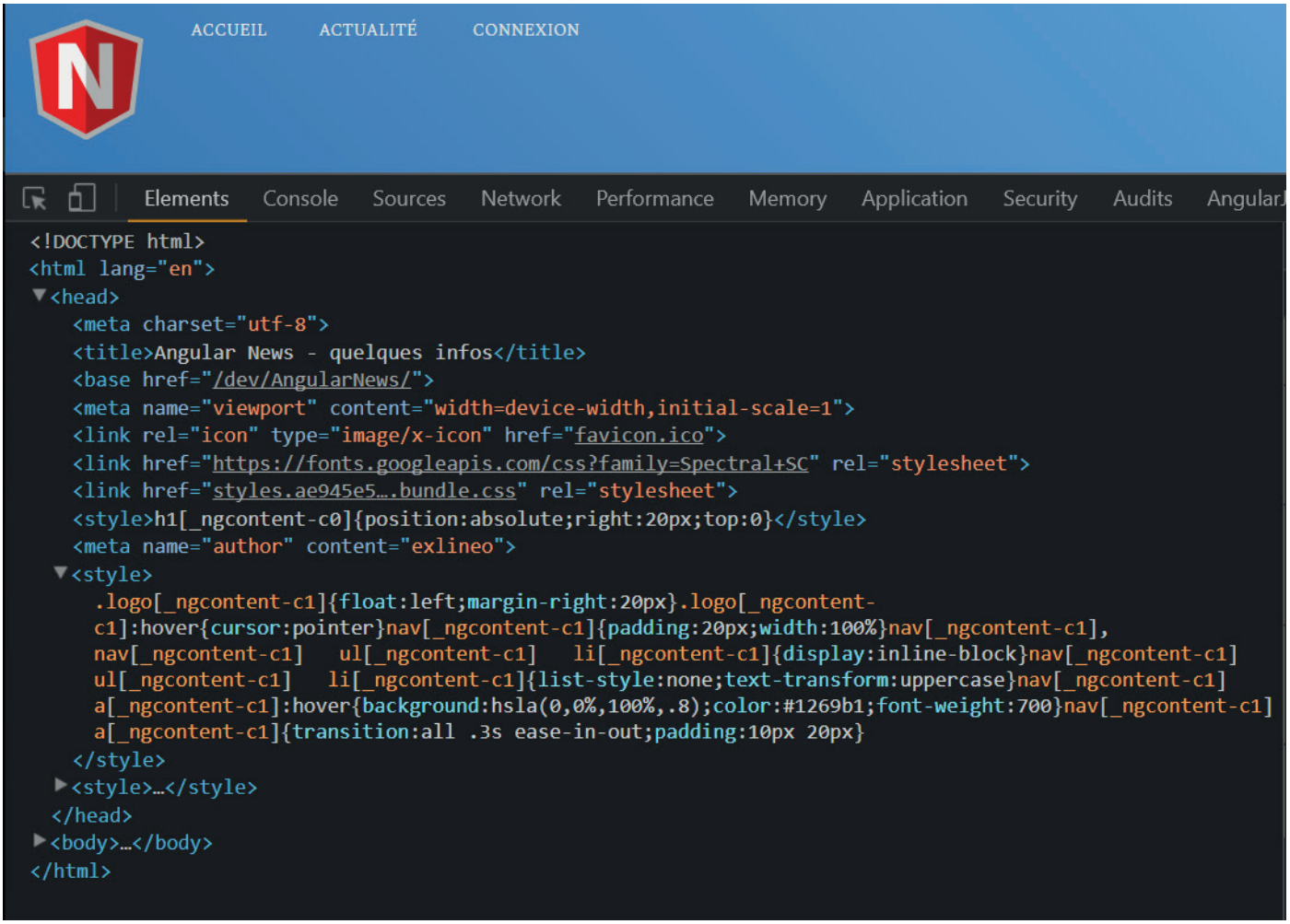
Annexes

Annexe 1

Un peu de mécanique CSS

Angular permet de créer des sites et applications sur une seule page, toutes ses mécaniques visent à gérer les accès aux contenus dans ce type de configuration. Le CSS est utilisé pour le ciblage des balises et leur identifications dans les différents contenus de l'application.

Différents composants seront créés dans une même application. A chacun d'eux une feuille de style sera attachée (Angular ClI les crée automatiquement). Les styles seront réécrit pour leur adjoindre des arguments permettant un ciblage de balises spécifiques.



Exemple de code généré par une application Angular

Dans l'exemple ci-dessus, on peut remarquer que les propriétés CSS sont augmentées d'arguments supplémentaires : [_ngcontent-c1]. Chaque propriété CSS créée en lien avec un composant sera traitée de la sorte pour assurer le ciblage entre le CSS et un composant spécifique.

C'est la mécanique centrale du fonctionnement du framework dans sa gestion d'une application sur une page.

Annexe 2

Eval et le fonctionnement d'Angular

Considérations sur Eval

Angular fonctionne sur la méthode eval() de Javascript. Cette mécanique induit un certain nombre de containtes notamment au niveau du code HTML. Chaque erreur de balise ou d'écriture dans le HTML renverra une erreur fatale.

Annexe 3

Les erreurs communes

You seem to not be depending on "@angular/core". This is an error.

Vous tentez de démarrer votre application NodeJS dans le mauvais dossier, vérifiez que vous lancez votre ng serve en étant dans le dossier où se trouve le fichier package.json.