Gram

- JS avancé
- Initialisation Angular
- Composants
- Databinding
- Directives
- Pipes
- Services
- Routing
- Formulaires
- Décorateurs
- Publication d'un projet

Côté Front, on est souvent amenés à manipuler des données, qui sont généralement reçues en format JSON :

nom.json

```
[
  {city: 'Paris', quantity: 84},
  {city: 'Marseille', quantity: 29},
  {city: 'Lyon', quantity: 14},
]
```

💡 Les valeurs peuvent être des chaînes de caractères, nombres, tableaux, objets ou booléens.

.length : retourne le nombre d'entrées
.push() : permet de rajouter une valeur à un tableau

📎 documentation JSON

nom.js

```
const piscines = [
  {city: 'Paris', quantity: 84},
  {city: 'Marseille', quantity: 29},
  {city: 'Lyon', quantity: 14},
];
```

.map() : permet de construire un nouveau tableau en utilisant les valeurs d'un premier :

nom.js

```
const villes = piscines.map((piscine) => {
      return piscine.city
});

const villes = piscines.map((piscine) => piscine.city);
// ['Paris', 'Marseille', 'Lyon']
```

🚨 Attention, sans les accolades il faut préciser un return !

nom.js

```
const piscines = [
  {city: 'Paris', quantity: 84},
  {city: 'Marseille', quantity: 29},
  {city: 'Lyon', quantity: 14},
];
```

.filter() : permet de filtrer les valeurs d'un tableau pour en créer un second :

nom.js

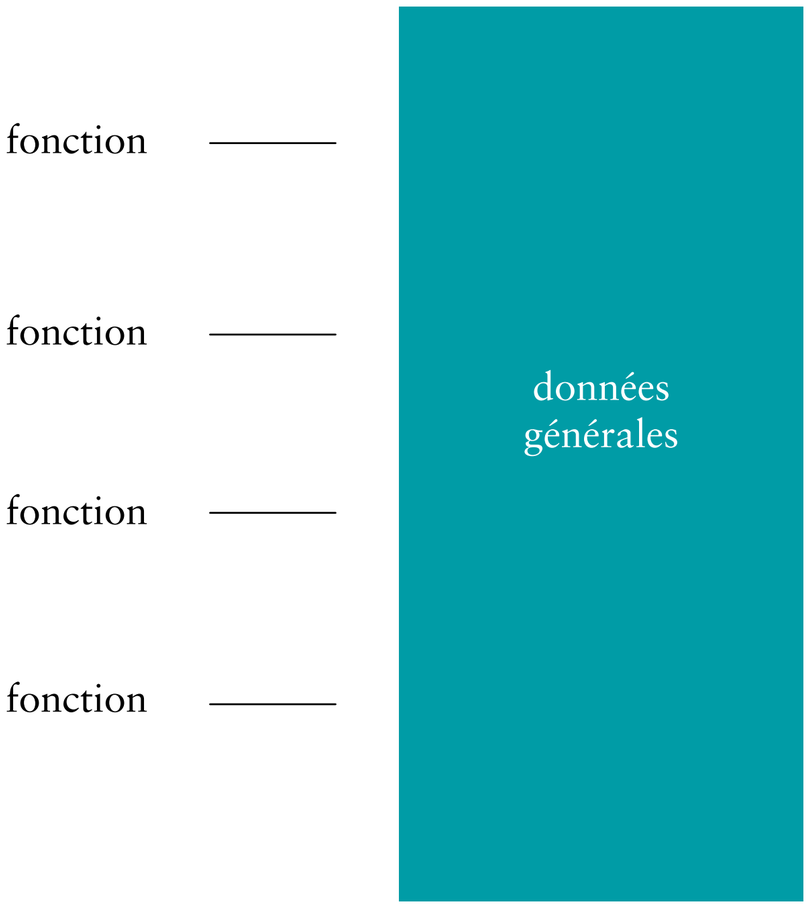
```
const sup20 = piscines.filter((piscine) => piscine.quantity > 20);
// [ {ville: 'Paris', quantity: 84}, {ville: 'Marseille', quantity: 29}]
```

.reduce() : permet de retourner une valeur, le second paramètre est la valeur initiale :

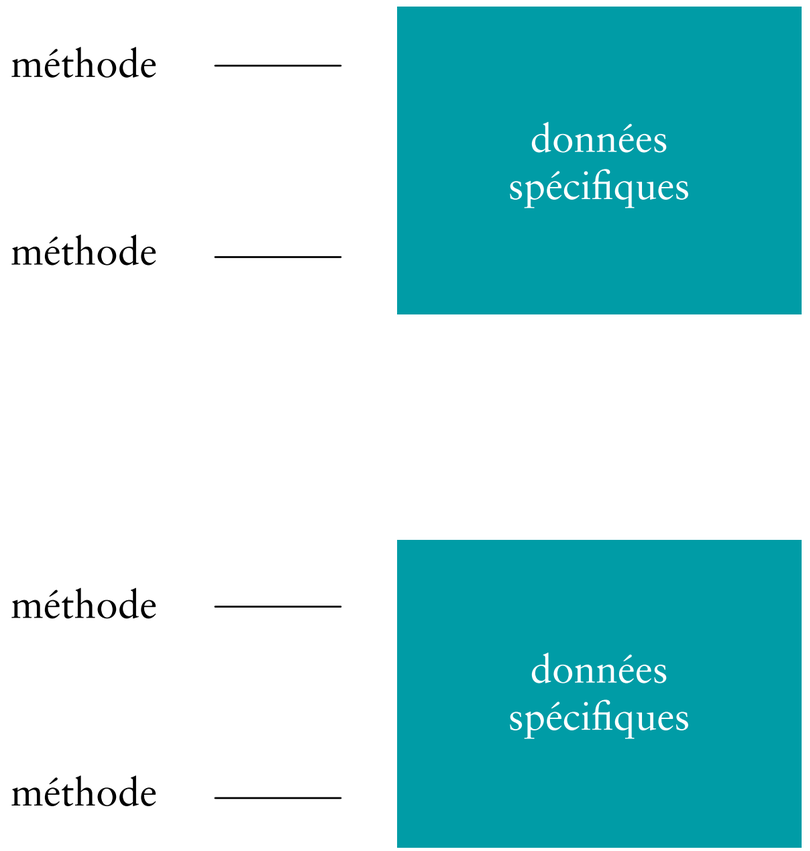
nom.js

```
const total = piscines.reduce((cumul, piscine) => cumul + piscine.quantity , 0);
// 127
```

Programmation procédurale



Programmation orientée objet



JS avancé : les Classes

📖 Les classes :

Permettent de créer un modèle qui sera utilisé pour appeler différentes fonctions (appelées méthodes). C'est la base de la programmation orientée objet (POO), utilisé par d'autres languages.

app.js

```
1   class Etudiant {
2       constructor(prenom, nom) {   // prenom est un paramètre
3           this.prenom  = prenom;  // this.prenom est une propriété
4           this.nom = nom;
5       }
6       fullName() { // fullName est une méthode
7           return `${this.prenom} ${this.nom}`;
8       }
9   }
10
11  const tartampion = new Etudiant('Tar', 'Tampion');  // 'Tar' est un argument
12  console.log( tartampion.fullName() );
13  const machinchose = new Etudiant('Machin', 'Chose');
14  console.log( machinchose.fullName() );
```

🚨 Par convention, les classes commencent toujours par une majuscule

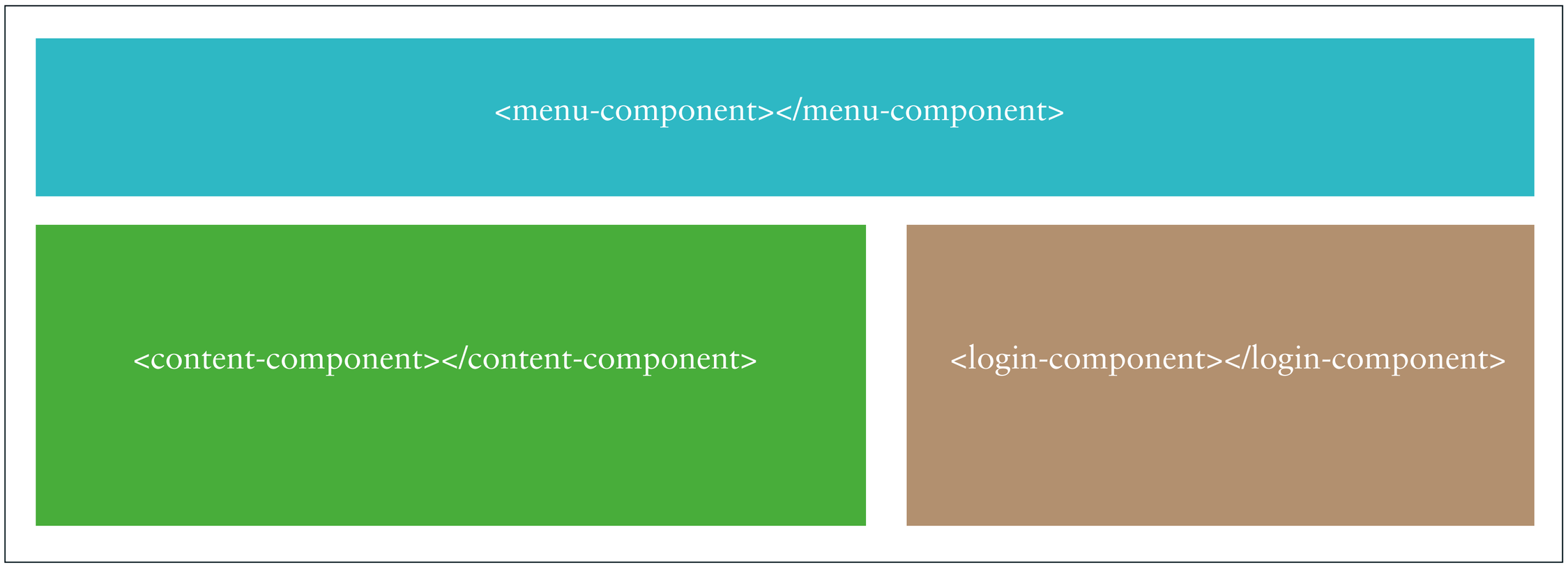
L'héritage d'une classe vers une autre vas permettre de bénéficier des fonctionnalités de la première classe au sein de la deuxième (qu'il s'agisse de ses propriétés ou ses méthodes) :

app.js

```
class Resultat extends Etudiant {
    constructor(prenom, nom, note) {
        super(prenom, nom);
        this.note = note;
    }
    resultat() {
        if(this.note > 10){
            console.log(`${this.prenom} ${this.nom} a réussi son examen`);
        } else {
            console.log(`${super.fullName()} doit repasser son examen`);
        }
    }
}

const jeandupont = new Resultat('Jean', 'Dupont', 15);
jeandupont.resultat();
```

📖 **Les composants :** Appelés 'component' en anglais, ce sont des éléments HTML que l'on peut créer (matérialisés sous forme de balises) afin de pouvoir les utiliser à différents endroit de notre projet et les réutiliser sur d'autres projets.

<app-root></app-root>



app.js

```
1   class TestComponent extends HTMLElement {
2       constructor() {
3           super();
4           this.innerText = "test";
5       }
6       connectedCallback() {
7           console.log("Ajouté à la page");
8       }
9       disconnectedCallback() {
10          console.log("Retiré de la page");
11      }
12  }
13
14  customElements.define("test-component", TestComponent);
```

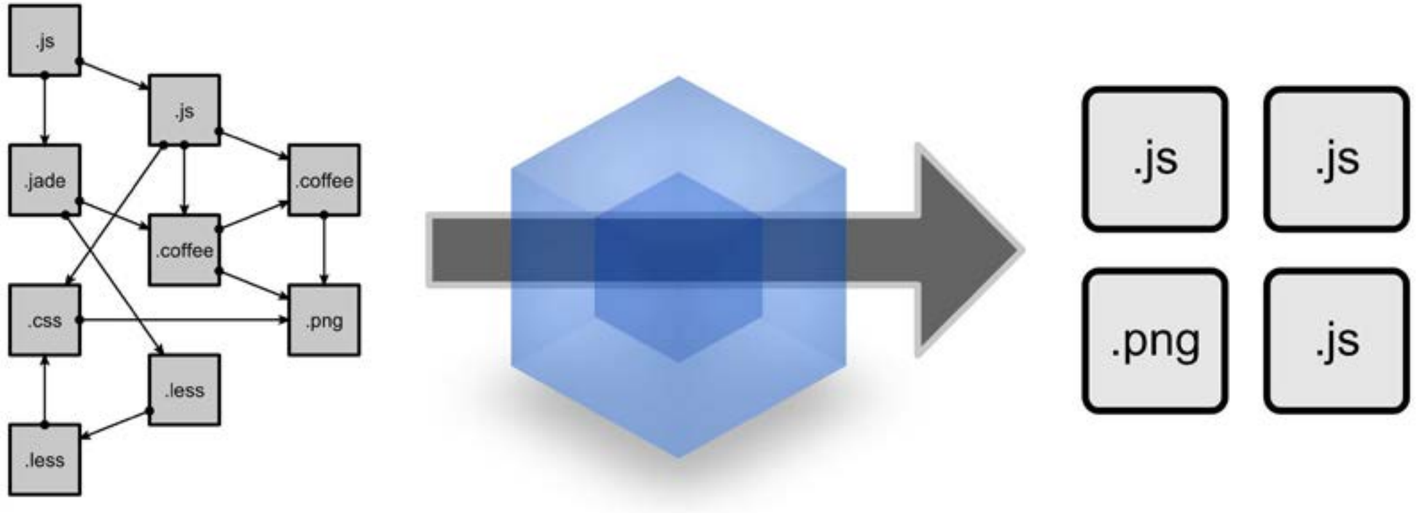
index.html

```
1   <test-component></test-component>
```

WebPack :

Il s'agit d'un module bundler, HMR (Hot Module Replacement), permet de gérer différents fichiers, de les regrouper dans un ou plusieurs modules.

Il permet notament la réecriture de standards (ES6 -> ES5, SASS -> CSS ...), la minification, la concatenation, la gestion du live reload, la mise en place d'un serveur web pour le développement ...



lien GitHub

```
git clone https://github.com/mwieth/Webpack-4-boilerplate.git
```

Changer nom du dossier comme souhaité, puis se rendre dans le dossier (mv ancien nouveau) :

```
cd monDossier
```

Installation des dépendances du projet :

```
npm install
```

Ouverture du serveur :

```
npm run build:dev
```

webpack

webpack.dev.js

webpack.prod.js

- Fichiers définitions WebPack par extension :
  - un fichier pour la phase de développement
  - un fichier pour la phase de production

package.json

- Dépendances du projet
- Commandes du projet

src

- Espace de travail

Installation de dépendances :

```
npm install jquery
```

> 💡 On retrouvera la dépendance jquery dans la liste des dépendances du fichier package.json et dans le dossier node_modules qui regroupe nos dépendances de développement

Si l'on souhaite supprimer une dépendance :

```
npm  uninstall jquery
```

src/index.js

```
1   import $ from 'jquery';
2   $('h1').html('test');
```

Réutilisons la classe créée précédemment et exportons là :

src/scripts/etudiant.js

```
export class Etudiant {
    ...
}
```

Maintenant, importons là dans le fichier index :

src/index.js

```
import { Etudiant } from './scripts/etudiant.js';
const tartampion = new Etudiant('Tar', 'Tampion');
console.log( tartampion.fullName() );
```

Une fois que le site est terminé, on lance la commande de production :

```
npm run build:prod
```


On retrouvera l'ensemble des fichiers directement dans le dossier dist, ce dossier que l'on mettra en ligne

📖 **TypeScript :**

- Langage de programmation **libre et open source** développé par Microsoft.
- Permet d'**améliorer et de sécuriser** la production de code JavaScript.
- Surcouche de JavaScript (tout code JavaScript ES6 fonctionnel)
- code TypeScript transcompilé en JavaScript
- permet un typage des variables et fonctions.

📎 documentation

Installation de TypeScript :

```
npm install -g typescript  // ajouter sudo sur mac
```

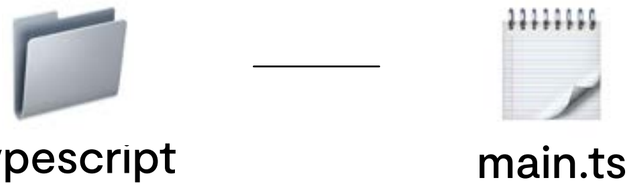
On peut ensuite vérifier qu'il a bien été installé :

```
tsc -v
```

L'extension des fichiers TypeScript est .ts :

... .ts

Créons dans un nouveau dossier le fichier main.ts :



main.ts

```ts
const test = 'test';
console.log(test);
```

```
tsc main.ts   // l'extension .ts est optionnelle
```

Automatiquement génère les fichiers et dossiers :

Si l'on souhaite que notre code TypeScript soit compilé automatiquement :

```
tsc main.ts --watch
```

Pour quitter l'auto-compilation :

CTRL + C

On vas pouvoir définir les types de nos variables :

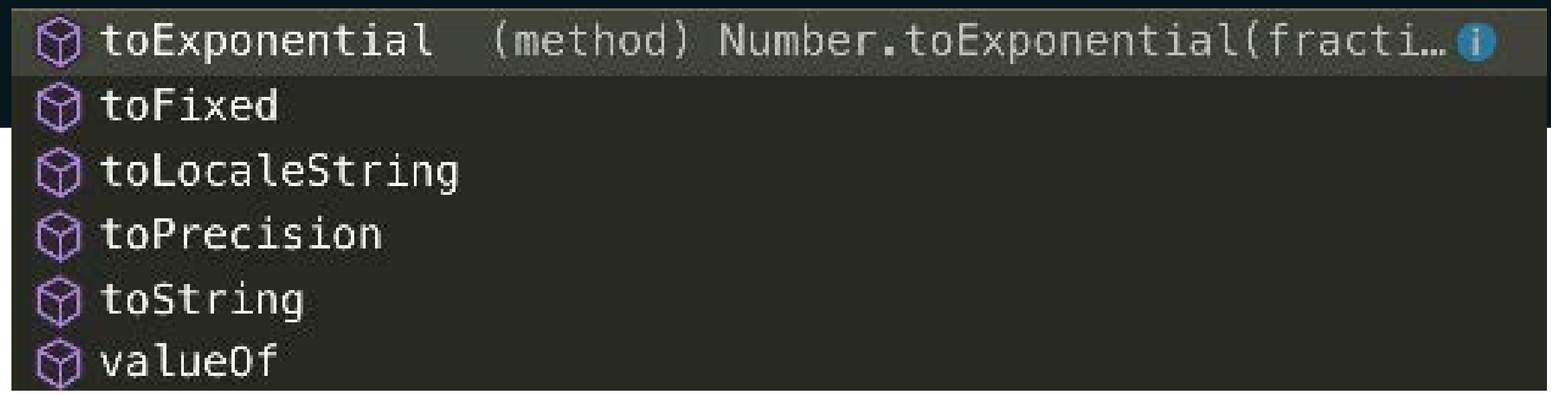
main.ts

```ts
let validation: boolean = true;
let total: number = 0;
let ecole: string = 'webforce3';
```

Une fois défini, le code retournera une erreur si la valeur ne respecte pas le type,
par contre les méthodes javaScript seront automatiquement proposées en fonction du type grace à l'intelliSense:

main.ts

```ts
total = 'test'; // retournera une erreur
total.
```



Il existe deux manières possibles de définir les tableaux (array) :

main.ts

```ts
let listeNombres: number[] = [1, 2, 3];
// ou :
let listeNombres: Array<number> = [1, 2, 3];
```

Il est possible qu'un tableau soit composé de différents types (tuple), il faut donc lui préciser en respectant l'ordre :

main.ts

```ts
let notesTableau: [string, number] = ['prenom', 0];
```

Pour un objet, il faut préciser les types des clés :

main.ts

```ts
let notes: {prenom: string, note: number} = {prenom: 'prenom', note: 0};
```

Si l'on ne connaît pas en amont :

main.ts

```ts
let maVariable: any;
...
maVariable = 'test';
maVariable.toUppercase();
```

unknown est moins permissif que any, on le définit plus tard :

main.ts

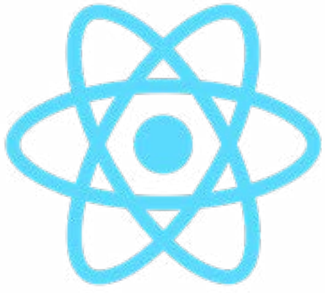
```ts
let maVariable: unknown;
...
maVariable = 'test';
maVariable.toUppercase();
```

enfin, lorsque l'on ne retourne pas de valeur, dans une fonction par exemple :

main.ts

```ts
function warnUser(): void {
    console.log('test');
}
```

2010 / Google



2013 / Facebook

2014 / Evan You
(ancien ingénieur Google)

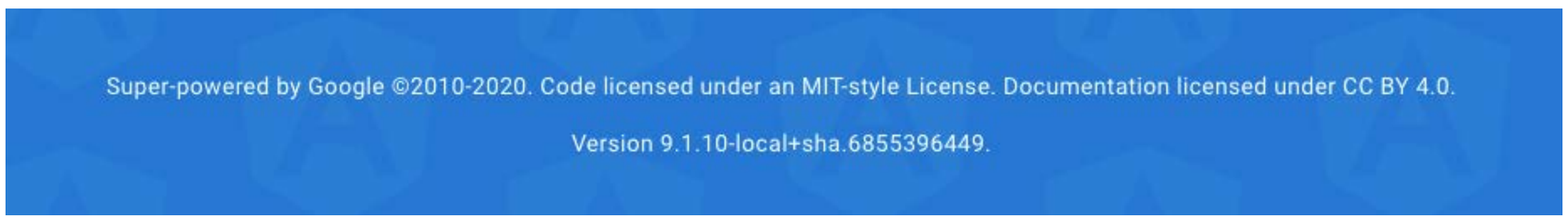
💡 Créer des applications clients, on parle de SPA

📖 SPA : "Single Page Application"
Une application web monopage est une application web accessible via une page web unique.
Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée, et de fluidifier ainsi l'expérience utilisateur.
- *Wikipedia*

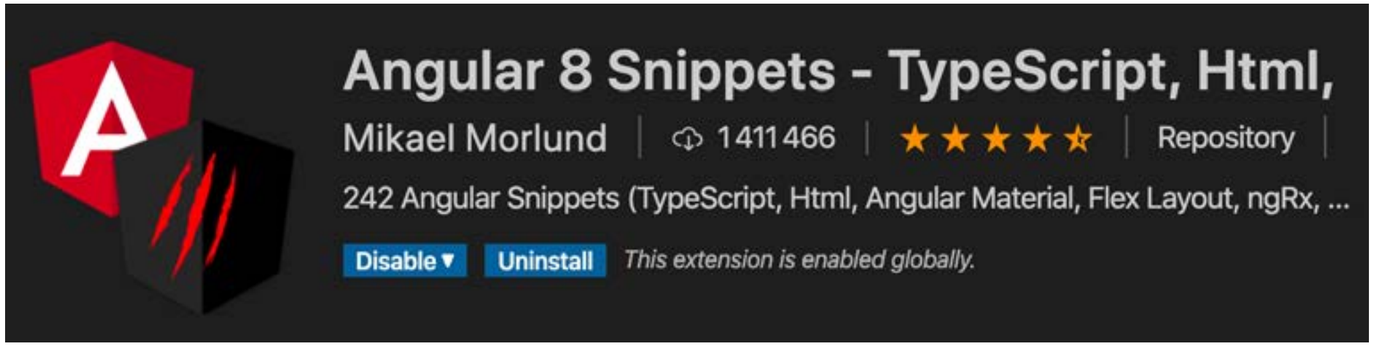
2010 / Angular JS

2016 / Angular 2 -> Angular

Depuis, deux nouvelles versions par an ... mais compatible :)

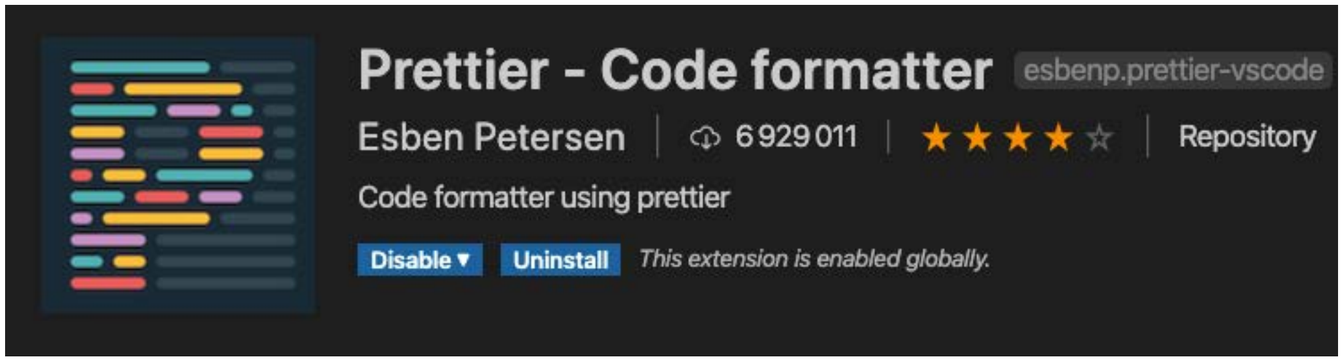


Installer le package de snippets :



On peut commencer à écrire ng- ...

Documentation



Initialisation

Vérifier que Node JS et NPM sont installés :

```
node -v
```

```
npm -v
```

📎 installation Node JS

Il faut ensuite installer Angular CLI (rajouter **sudo** sur Mac pour autoriser l'installation):

```
npm install -g @angular/cli    // ng version
ng new nom-application  // routing + Sass
cd nom-application
ng serve -o  // o pour open
```

📎 installation Angular CLI

📖 **ng** : aNGular

package.json

Il contient les différentes dépendances du projet (**dependencies / devDependencies**), ainsi que les commandes exécutables (**scripts**)

node_modules

Les dépendances sont installées dans le dossier node_modules

src

Contient le code que nous allons éditer

app

Les différents composants et modules

composants

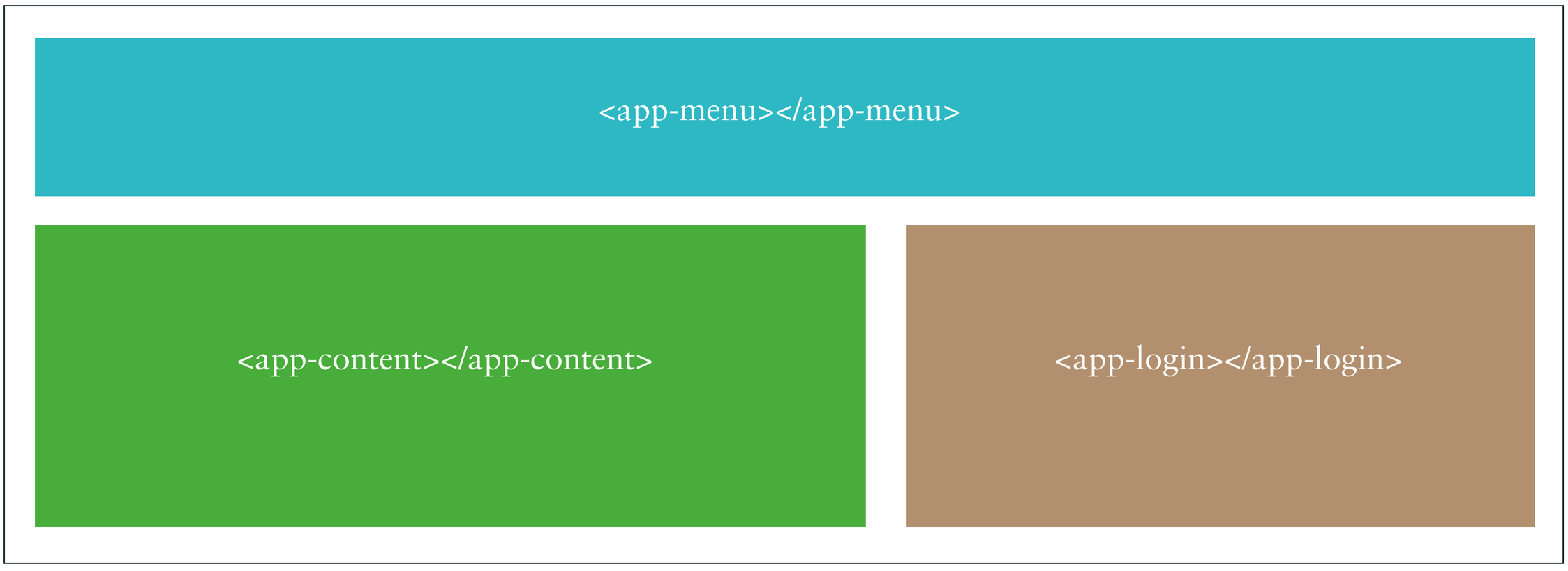
services

interfaces

Structure Component

Une page est un ensemble de composants :

<app-root></app-root>



index.html

```
<app-root></app-root>
```

Notre appel **app-root** provient de la déclaration de notre composant :

src/app/app.component.ts

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

Créer un composant

Pour ajouter un composant, il faut utiliser le CLI d'Angular :

```
ng generate component nom
```

Il est possible d'utiliser des raccourcis :

```
ng g c nom
```

Pour voir ce qu'effectue une action sans la réaliser, il faut ajouter --dry-run à la fin de la déclaration :

```
ng g c nom  --dry-run
```

Lors de la création de nos composants, on peut indiquer que l'on souhaite un template et des styles en ligne :

```
ng g c nom -it -is  // inline template, inline styles
```

Automatiquement, Angular est venu ajouter ce nouveau composant à la liste des imports, ainsi qu'à ses déclarations :

src/app/app.module.ts

```
...
import { NouveauComponent } from './nouveau/nouveau.component';

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

Il va également créer un dossier à l'intérieur du dossier app, avec le nom du composant, et qui contient un fichier html, un scss, et un ts :

src/app/test.component.ts

```ts
@Component({
  selector: 'app-nouveau',
  templateUrl: './nouveau.component.html',
  styleUrls: ['./nouveau.component.scss']
})
```

On peut désormais faire référence à ce nouveau composant dans notre composant App :

src/app/app.component.html

```html
<div>
  Je suis dans le composant App
  <app-nouveau></app-nouveau>
</div>
```

Supprimer un composant

🚨 Il n'y a pas de commande CLI pour supprimer un composant, il faut le faire manuellement :

1. 

Il faut commencer par supprimer le dossier du composant.

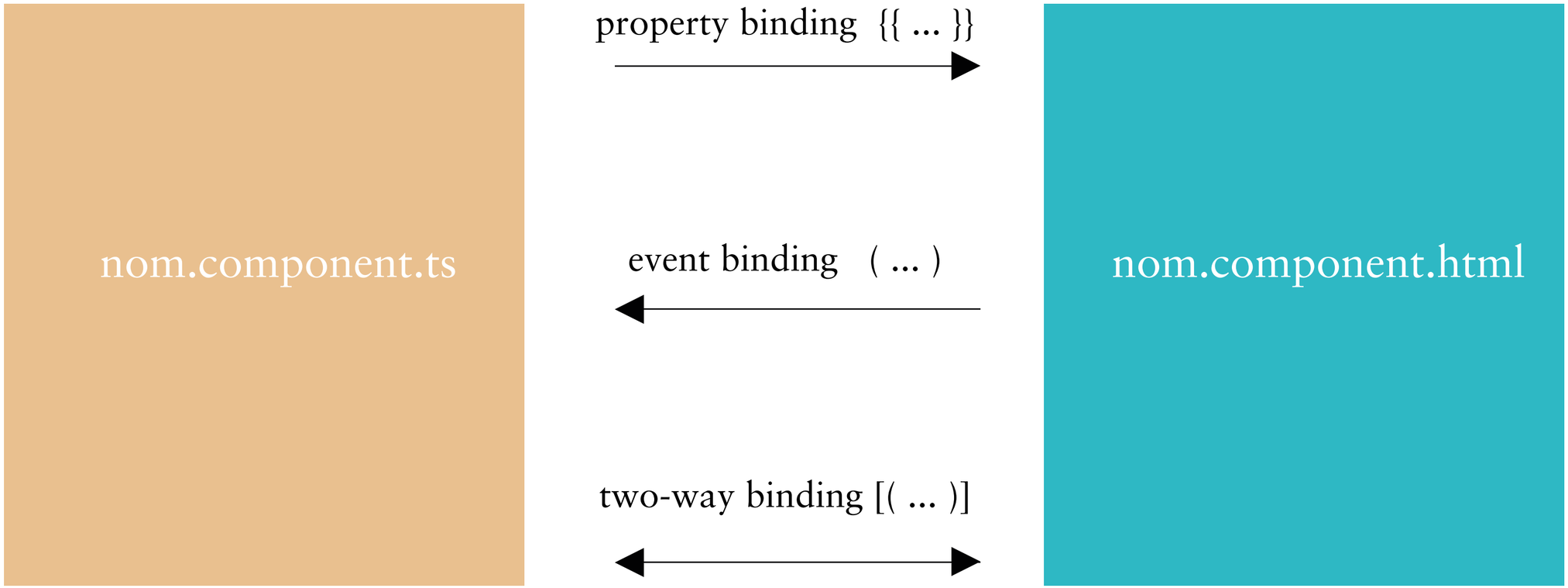
2. Puis retirer les lignes correspondantes dans le fichier module.

src/app/app.module.html

```
import { NomComponent } from './nom/nom.component';
...
@NgModule({
  declarations: [
    NomComponent,
```

3. Enfin, il faut vérifier qu'il n'y a plus d'utilisation de ce composant dans les autres templates.

Au sein d'un composant, nous allons souhaiter une communication de données entre notre HTML et notre TS:



On peut déclarer des propriétés dans notre fichier typescript :

src/app/nouveau/nouveau.component.ts

```ts
export class NouveauComponent implements OnInit {

  title: string = 'Mon composant';
  quantity: number = 4;

  constructor() { }
```

Pour par la suite pouvoir les utiliser dans notre fichier HTML :

src/app/nouveau/nouveau.component.html

```html
{{ title }} - {{ quantity }}
```

On peut également faire appel à des méthodes :

src/app/nouveau/nouveau.component.ts

```ts
export class NouveauComponent implements OnInit {
  total: number = 4;
  getTaxes() {
    return this.total * 0.2;
  }
}
```

src/app/nouveau/nouveau.component.html

```
TVA : {{ getTaxes() }}
```

Il est possible de modifier dynamiquement les attributs d'un élément du DOM :

src/app/nouveau/nouveau.component.ts

```ts
export class NouveauComponent {
    total: number = 4;
    more: boolean = false;

    constructor() {
        if(this.total > 3) this.more = true;
    }
}
```

Pour l'associer à la variable, on utilise les crochets :

src/app/nouveau/nouveau.component.html

```html
<button [disabled]="!more">Envoyer</button>
```

Databinding : Event binding

Les Event Bindings permettent de communiquer du HTML vers TS :

src/app/nouveau/nouveau.component.html

```html
<button (click)="buy()">Acheter</button>
```

src/app/nouveau/nouveau.component.ts

```ts
export class NomComponent {
  buy() {
    console.log('acheter');
  }
}
```

Qui correspond à l'écriture procédurale :

📎 événements JS

nom.js

```js
document.querySelector('button').addEventListener('click', () => this.buy() );
```

Utilisé notament dans les formulaire, il permet par exemple de lier une propriété ET un événement.

Il faut d'abord importer le module FormsModule depuis @angular/forms

src/app/app.module.ts

```
import { FormsModule } from '@angular/forms';


@NgModule({
  ...
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  ...
})
export class AppModule { }
```

L'écriture associe les parenthèse de l'event binding, et les crochets de property-binding :

src/app/nouveau/nouveau.component.ts

```ts
name: string = "Nom"
```

src/app/nouveau/nouveau.component.html

```html
<h1> {{ name }} </h1>
<input type="text" class="form-control" [(ngModel)]="name">
```

📎 Informations sur ngModel

Databinding : Récapitulatif

`{{ name }}`	Lorsque je souhaite afficher une propriété de mon fichier TS.
`{{ methode() }}`	Lorsque je souhaite exécuter une méthode de mon fichier TS.
`[attribut]="name"`	Lorsque je souhaite donner une valeur d'attribut à partir de mon fichier TS.
`(event)="methode()"`	Lorsque je souhaite communiquer à partir d'un événement.
`[(ngModel)]="name"`	Lorsque je souhaite une communication bi-latérale.

Modification de Sass vers scss :

1. 

angular.json

Remplacer dans le fichier les occurences de sass par scss

2. 

... .sass

... .scss

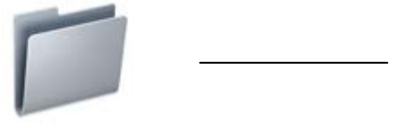
Renommer les extensions de fichier

3. 

... .ts

Remplacer dans les fichiers typescript les réferrences aux fichiers sass

Les styles et assets :



src/app/nouveau/nouveau.component.html

```
<img src="/assets/image.jpg"/>
```

src/app/nouveau/nouveau.component.scss

```
element {
    background: url("/assets/image.jpg");
}
```

Il est possible de créer un fichier pour nos variables et de l'importer

src/app/styles.scss

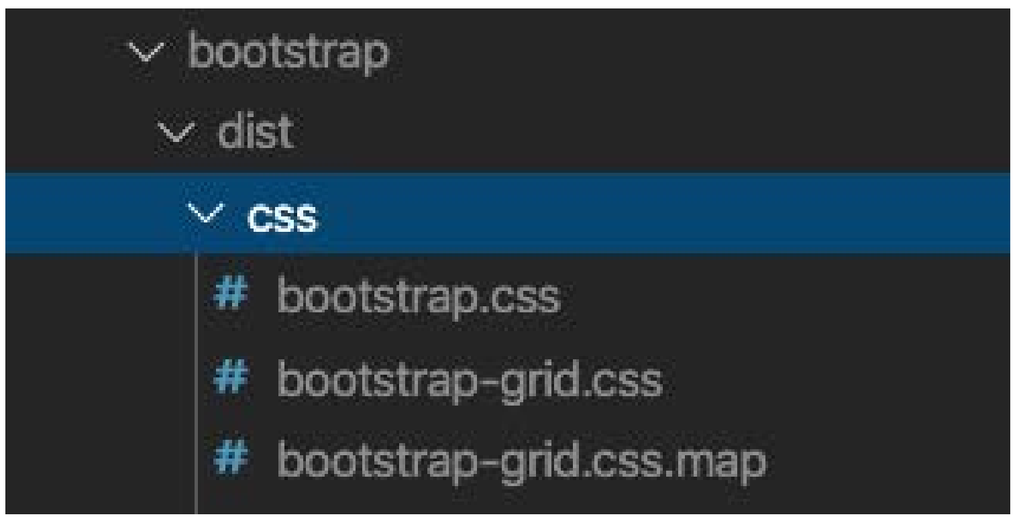
```
@import “~src/variables.scss”;

h1 {
  color: $header-color;
}
```

Ajout de Bootstrap :

```
npm install bootstrap@next --save
```


node_modules


package.json



Il faut ensuite l'ajouter manuellement au fichier angular.json :

angular.json

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

🚨 Quand vous effectuez des modifications au fichier package.json ou angular.json, il faut quitter la console (CTR + C) et relancer : ng serve -o

(également possible dans le head de index.html, ou dans le fichier styles.scss)

styles.scss

```
@import "~bootstrap/dist/css/bootstrap.css"
```

Imaginons que l'on souhaite ajouter jQuery à certains composant de notre web app :

```
npm i jquery
```

📎 npm jQuery

En regardant la documentation de NPM, vous remarquerez la possibilité de l'utiliser en tant que module :

src/app/app.component.ts

```ts
import $ from 'jquery';
...
ngOnInit(): void {
    $('h1').html('test');
}
```

Les directives

📖 **Directives :** Les directives sont des instructions intégrées dans le DOM, que l'on écrit dans notre fichier template HTML.

Directives d'attributs : ajout de classes, styles, de maniètre dynamique.
Directives structurelles : gestion des boucles, conditions.

Directives d'attributs:

src/app/app.component.html

```
1   <h1 [ngStyle]="{ color: color }"> ... </h1>
```

💡 Si besoin de conditions, on utilise une condition ternaire.

src/app/app.component.html

```
1   <h1 [ngClass]="{'text-success': valid == true}"> ... </h1>
```

Les directives structurelles commencent toujours par une asterisque *

***ngIf :** Permet la gestion des conditions.

src/app/nouveau/nouveau.component.html

```
<p *ngIf="valid == true">Validé</p>
```

src/app/nouveau/nouveau.component.html

```
<p *ngIf="valid == true; else nonvalid">Validé</p>

<ng-template #nonvalid>
    <h1>Non validé</h1>
</ng-template>
```

*ngFor : Permet la gestion des boucles.

src/app/nouveau/nouveau.component.ts

```ts
export class AppComponent {
  animaux: string[] = [ 'Chien', 'Chat'];
  constructor() {}
}
```

src/app/nouveau/nouveau.component.html

```html
<p *ngFor="let animal of animaux"> {{ animal }} </p>
```

À partir d'une boucle, il est également possible d'appeler un autre composant :

src/app/nouveau/nouveau.component.html

```
<app-animal *ngFor="let animal of animaux"></app-animal>
```

L'index peut nous être utile dans certaines situations, il faut créer une variable dans notre boucle :

src/app/nouveau/nouveau.component.html

```
<app-animal *ngFor="let animal of animaux ;  let i = index"></app-animal>
```

📖 **Interface :** Une interface TypeScript permet de définir la signature (ou le contrat) d'une classe où même une fonction. On peut donc l'utiliser comme un "type" de propriété.

```
ng generate interface dossier/nom
```

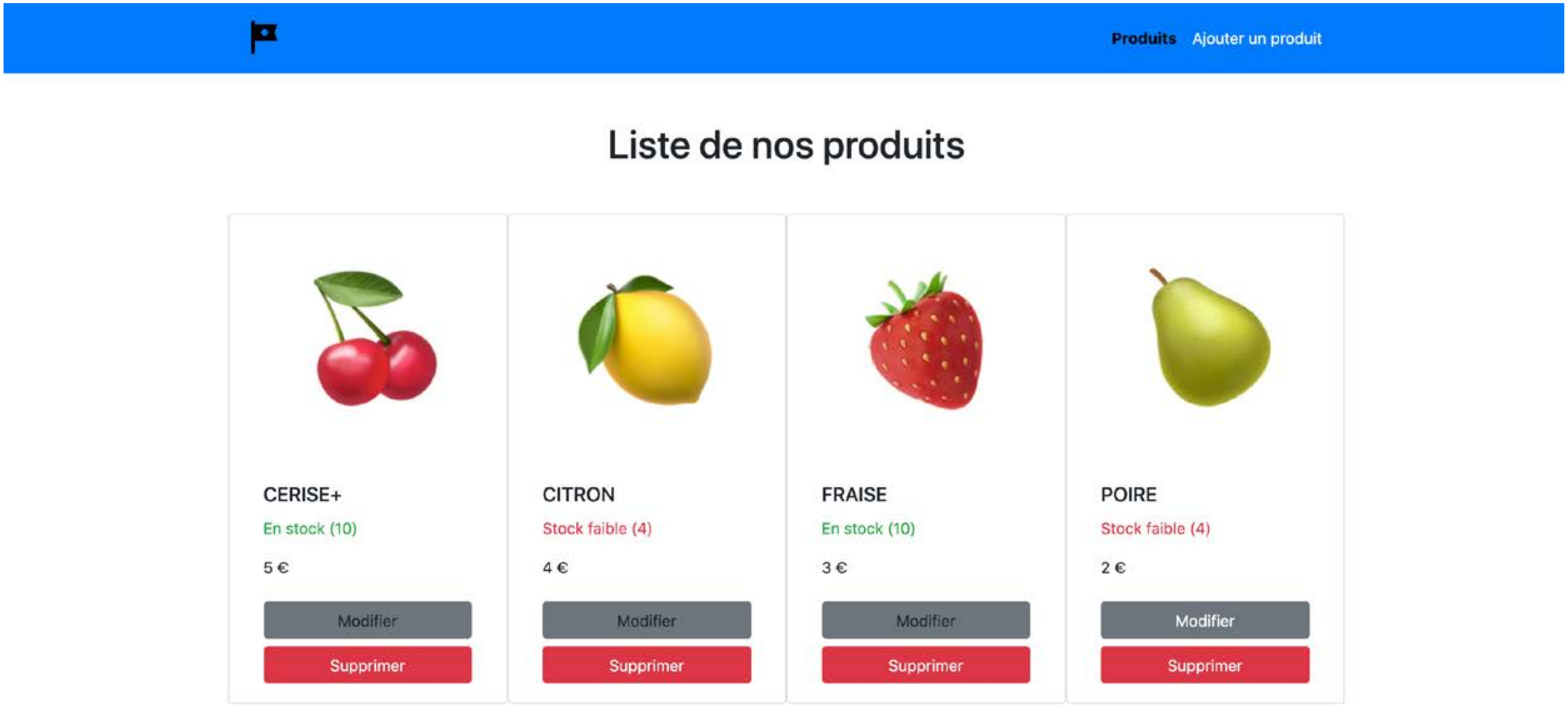
src/app/interfaces/nom.ts

```ts
export interface Animal {
  name: string;
  quantity?: number;  // ? : propriété facultative
}
```

Lorsque l'on souhaite l'utiliser, il faudra bien penser à l'importer :

src/app/components/nom/nom.component.ts

```ts
import { Animal } from 'src/app/interfaces/animal';
...
animal: Animal    //    animaux: Animal[]
```


Produits
Ajouter un produit
Liste de nos produits
CERISE+
En stock (10)
5 €
Modifier
Supprimer
CITRON
Stock faible (4)
4 €
Modifier
Supprimer
FRAISE
En stock (10)
3 €
Modifier
Supprimer
POIRE
Stock faible (4)
2 €
Modifier
Supprimer

Les Pipes

📖 **Pipe :** Les pipes permettent de manipuler les données lors de leur affichage dans le DOM.

💡 **Les Pipes s'écrivant avec une barre verticale : |**

Imaginons que l'on récupére une données de date :

src/app/app.component.js

```
export class AppComponent {
  lastUpdate: Date = new Date();
  ...
}
```

Si on l'affiche telle quelle :

src/app/app.component.html

```
<p>Mis à jour : {{ lastUpdate }}</p>
```

On peut donc utiliser un pipe pour transformer le résultat :

src/app/app.component.html

```
1    <p>Mis à jour : {{ lastUpdate | date }}</p>
```

Les pipes ont des paramètres que l'on peut configurer, par exemple :

src/app/app.component.html

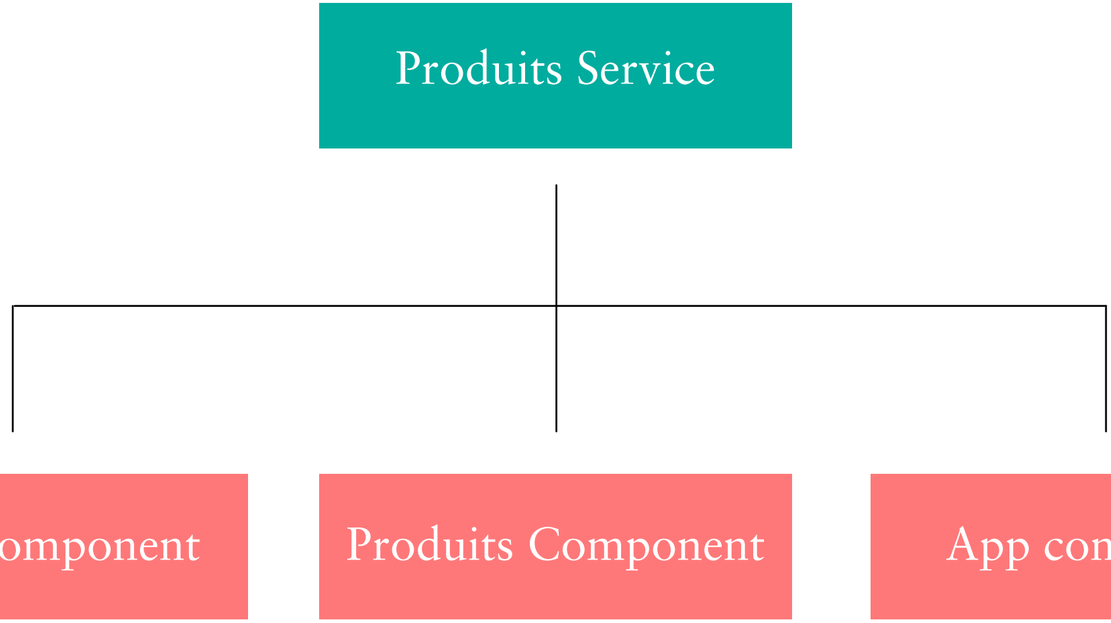
```
1    <p>Mis à jour : {{ lastUpdate | date : 'short' }}</p>
```

src/app/app.component.html

```
1    <p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' }}</p>
```

📎 Liste des pipes

Les Services



- DRY (Don't Repeat Yourself) : Communiquer des informations à différents composants
- Single Responsability Principle (chaque élément doit avoir une seule fonctionalité)
- Convention : ... .service.ts
- Permet aussi de communiquer avec une base de données / API

Puis ajoutons un service 'animaux' :

```
ng generate service services/animaux
```

ou avec le racourci :

```
ng g s services/animaux
```

Une fois crée, il faut l'importer et l'ajouter aux providers de App :

src/app/app.module.ts

```
import { AnimauxService } from './services/animaux.service;
@NgModule({
  ...
  providers: [
    AnimauxService
  ],
})';
```

Puis l'injecter dans un composant, le composant principal par exemple :

src/app/nom.component.ts

```
1   import { AnimauxService } from './services/animaux.service;
2   constructor(private _animaux: AnimauxService) {}
```

On peut ensuite accéder aux propriétés et méthodes du service :

src/app/nom.component.ts

```
1   this._animaux.maMethode();      this._animaux.maPropriete;
```

💡 Par convention, lorsque l'on injecte un service, on le fait commencer par un underscore _

Route : Il s'agit des instructions d'affichage à suivre pour chaque URL, c'est-à-dire quel(s) component(s) il faut afficher à quel(s) endroit(s) pour un URL donné.


produits


panier

src/app/app-routing.module.ts

```ts
import { PanierComponent } from './components/panier/panier.component';
import { ErrorComponent } from './components/error/error.component';

const routes: Routes = [
  { path: 'panier', component: PanierComponent },
  { path: '', redirectTo: '/panier', pathMatch: 'full' },
  { path: '**', component: ErrorComponent },
];
```

💡 **path** correspond à ce qui vient après **localhost:4200/** , donc pas besoin de mettre le slash initial!
'' correspond à la page d'accueil de l'application web.
** correspond à une route inexistante.

Dans notre template, on utilise la balise **router-outlet** qui affiche les différents contenus :

src/app/app.component.html

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

On peut utiliser les liens dans notre template :

src/app/app.component.html

```
<ul class="nav navbar-nav">
    <li><a routerLinkActive="active" routerLink="panier">panier</a></li>
    ...
</ul>
```

💡 **routerLinkActive** permet d'ajouter des classes uniquement si il s'agit de la route actuelle.

Ou dans notre fichier typeScript, en important Route au préalable :

src/app/auth/auth.component.ts

```
import { Router } from '@angular/router';
...
constructor(private router: Router) {}
onSignIn() {
    this.router.navigate(['panier']);
}
```

src/app/app-routing.module.ts

```
const routes: Routes = [
  { path: 'animaux/:id', component: EnfantComponent },
];
```

💡 les : permettent d'inclure des paramètres à nos routes (pensez à des articles de blog par exemple)

Puis au template de notre vue :

src/app/components/parent/parent.component.html

```
<div *ngFor="let animal of animaux; let i = index">
    <a [routerLink]="['/animaux', i]">Détail</a>
</div>
```

On peut aussi récupérer ce paramètre dans notre composant enfant :

src/app/components/enfant/enfant.component.ts

```
1   import { ActivatedRoute } from '@angular/router';
2   ...
3   id: number;
4   constructor(private router: ActivatedRoute) { }
5   ngOnInit() {
6       this.id = this.router.snapshot.params['id'];
7   }
```

Dans notre service, nous pouvons créer une méthode pour récupérer les information de l'élément grace à son ID :

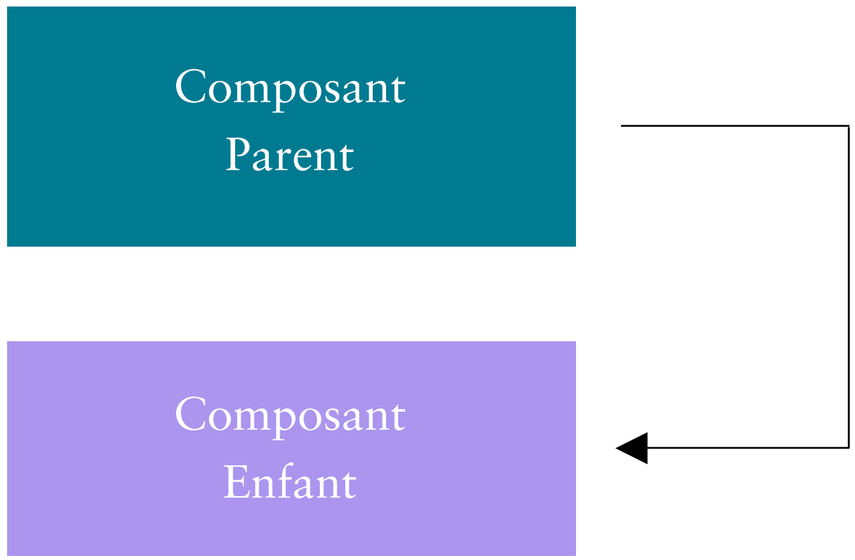
src/app/services/nom.service.ts

```ts
getAnimal(id: number) {
    return this.animaux[id];
}
```

Si nous avons injecté le service dans le composant enfant, il sera alors possible de récupérer les informations le concernant :

src/app/components/enfant/enfant.component.ts

```ts
ngOnInit() {
  ...
  this.animal = this._animal.getAnimal(id);
}
```



Les décorateurs **@Input()** permettent de transférer des informations du composant parent vers le composant enfant.

Il faut les exporter depuis le template de notre composant parent :

src/app/components/parent/parent.components.html

```
<app-enfant *ngFor="let animal of animaux; let i = index"  [id]="i"></app-enfant>
```

Puis les importer dans notre composant enfant :

src/app/components/enfant/enfant.components.ts

```
import { Component, OnInit, Input } from '@angular/core';
...
export class EnfantComponent implements OnInit {
  @Input() id: number;
  ...
}
```

Il existe 2 types de formulaires dans Angular :

- **Template Driven Form :** La logique est essentiellement dans le template (avec ngModel)
- **Reactive Forms :** La logique est essentiellement dans le fichier typeScript.

src/app/app.module.ts

```
1   import { FormsModule } from '@angular/forms';
2
3
4   @NgModule({
5     ...
6     imports: [
7       BrowserModule,
8       AppRoutingModule,
9       FormsModule
10    ],
11    ...
12  })
13  export class AppModule { }
```

💡 **FormsModule** permet d'accéder aux fonctionnalités de formulaire d'Angular

Template Driven Form : settings

1. Il faut relier le formulaire à ngForm en lui donnant un nom ainsi qu'une méthode d'envoi:

src/app/nom.component.html

```
<form #animalForm="ngForm" (ngSubmit)="onSubmit(animalForm)">
    ...
    <button type="submit">Envoyer</button>
</form>
```

2. Chaque information du formulaire que l'on souhaite récupérer doit implémenter la directive ngModel.

src/app/nom.component.html

```
<input type="text" name="animal" ngModel>
```

3. On peut enfin récupérer les informations depuis notre méthode

src/app/nom.component.ts

```
onSubmit(animalForm) {
    console.log(animalForm.value);
 }
```