

Framework Web - Angular

David Bertolo - david.bertolo@univ-lorraine.fr



Framework web

- Objectifs - A la fin du module, vous serez capable de :
 - ▶ Comprendre les concepts utilisés par le Framework Angular
 - ▶ Réaliser une Single Page Application simple en utilisant le framework angular et en prenant en compte l'ensemble des concepts vus

Framework web

- Contenu du module

 Angular : Framework de développement web

- Évaluation
 - ▶ Projet : développement d'une Single Page Application
 - ▶ Individuel à rendre la première semaine de janvier 2023
- La conception du cours s'est appuyée sur la formation open source : SFEIR School Angular

Framework web

- Fonctionnement pratique
 - ▶ Cours qui introduit les notions
 - ▶ TP pour application, via des exercices, des notions introduites
 - ▶ Correction des TP et mise en ligne des corrections
- Support de cours ARCHE
- Projet : aide et assistance pendant les TP

Plan

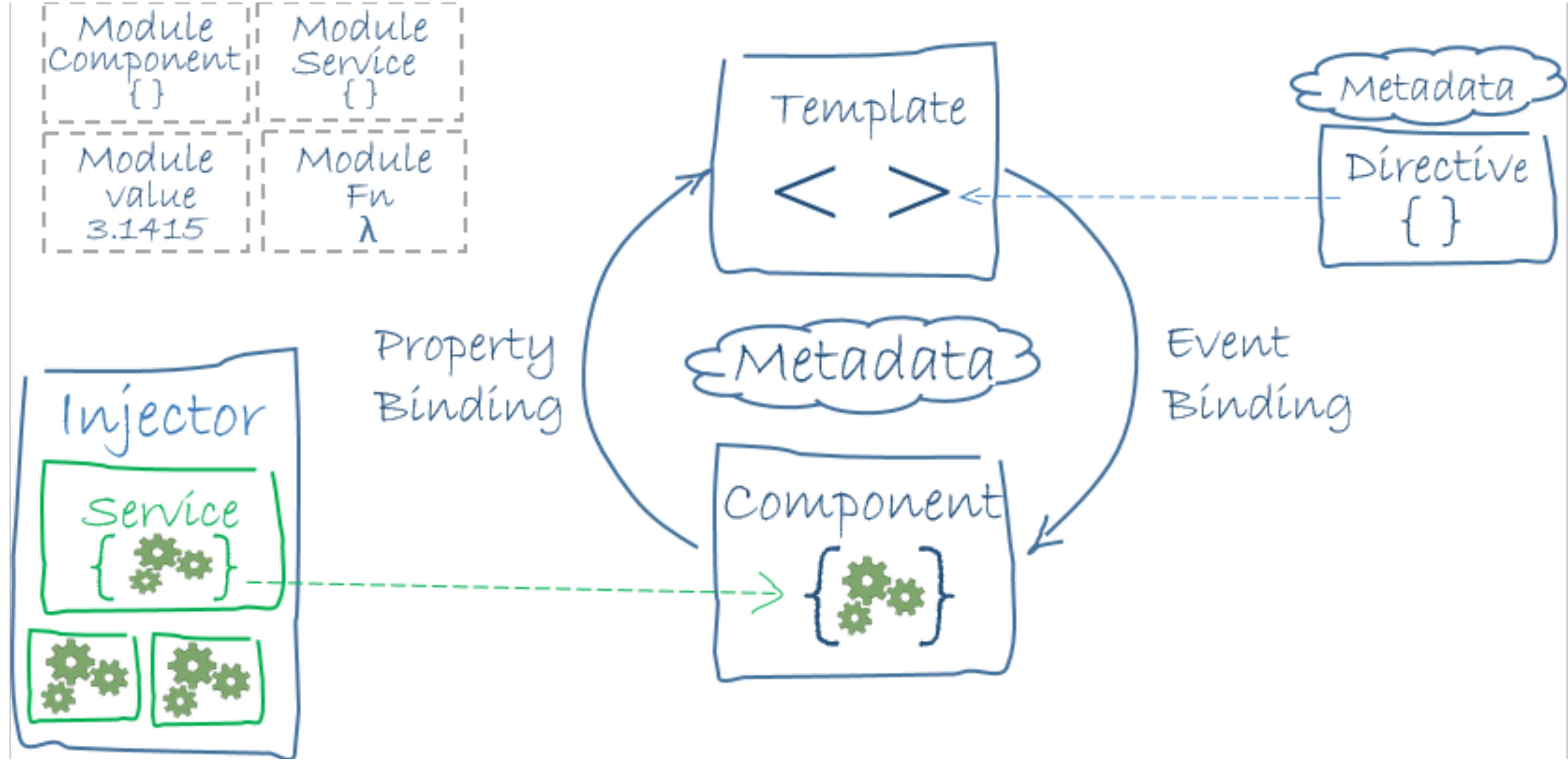
- Généralités
- Installation et Configuration
- Les composants
- Les directives
- Les pipes
- Les services
- Le routing

Généralités

- Framework pour le développement d'applications clients (Front-end)
- Open source
- Réécriture complète d'AngularJS
- Permet de créer des Single Page Applications
 - Fluidifie l'expérience utilisateur
 - Évite les chargements de pages à chaque nouvelle action
- Actuellement version 17 (08/11/2023)

Généralités

- Basé sur une architecture Modèle-Vue-Contrôleur (MVC)
- Trois langages principaux :
 - ▶ **HTML** pour la structuration
 - ▶ **SCSS** pour les styles (surcouche CSS)
 - ▶ **TypeScript** pour le côté dynamique (Javascript typé)
- Angular : <https://angular.io/>
- TypeScript : <https://www.typescriptlang.org/>

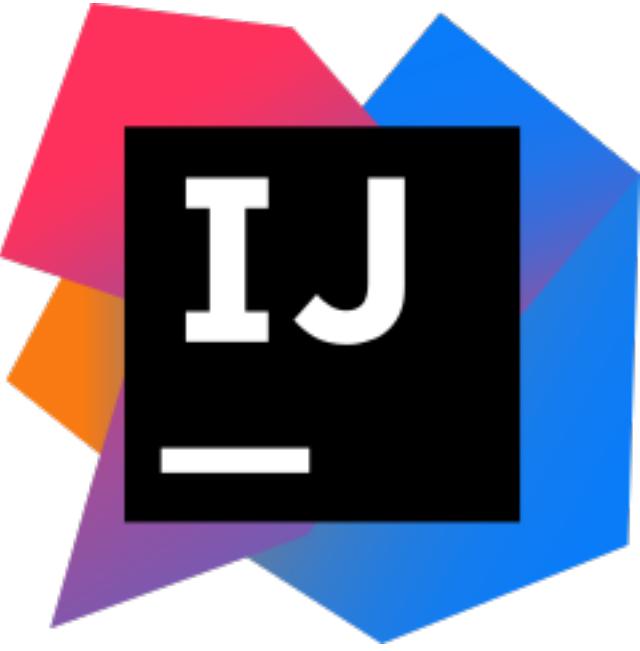


Généralités

Structure globale



IDE's



IntelliJ / **Webstorm**, solution de JetBrains avec les plugins nécessaires au développement Web :

- Auto-completion
- Language
- Pre-Debug configuration



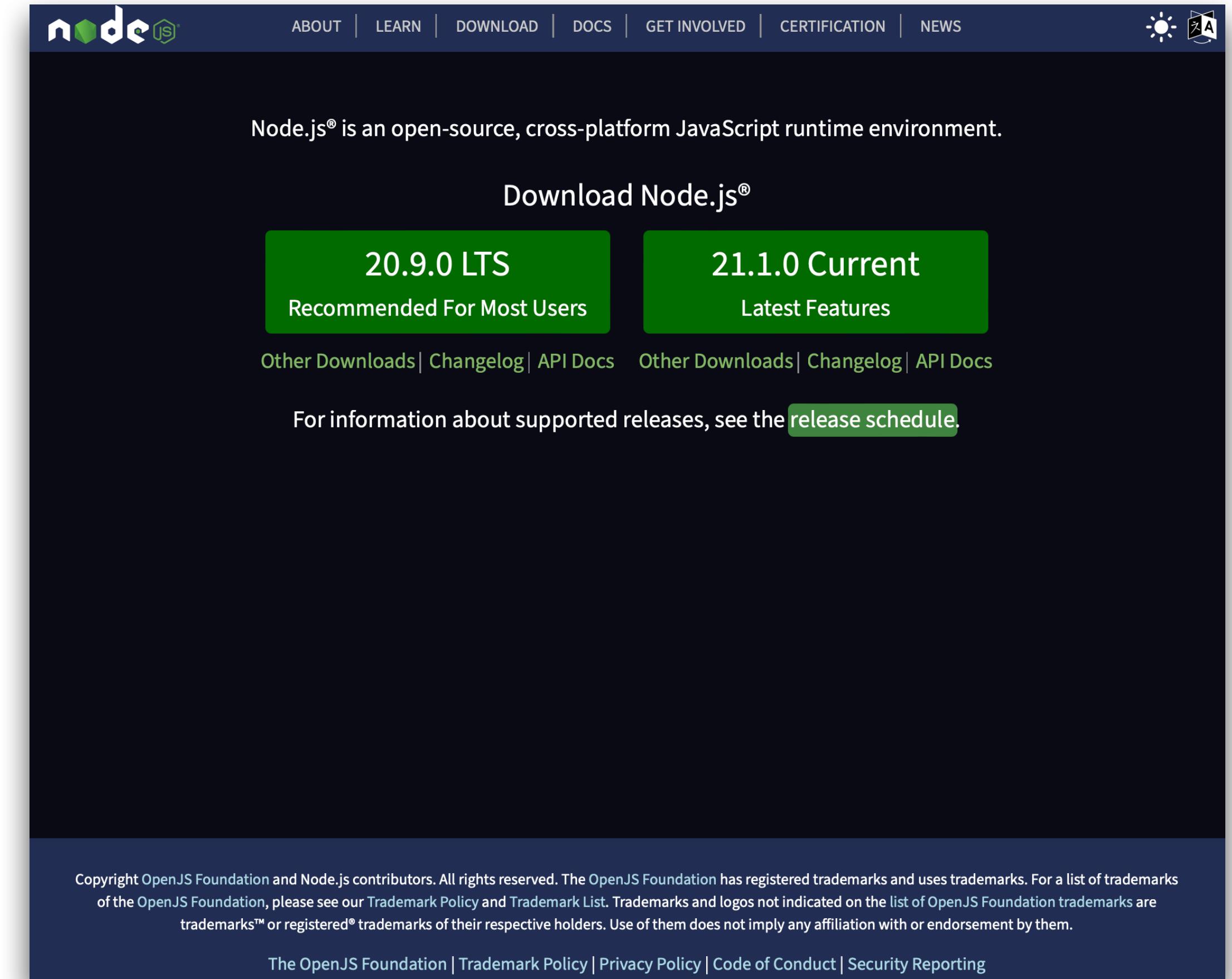
IDE's

Visual Studio Code, solution de Microsoft, nécessite d'installer des plugins pour avoir un IDE performant :

- Angular language service: autorise le Framework Syntaxe + auto-completion
- Visual Studio IntelliCode pour de l'auto-complétion intelligente
- Editor Config: pour avoir un config de l'éditeur commune avec tous les membres de votre équipe

Angular configuration

- Installer Node.js
- Prendre la version LTS (Long Term Support)
- Utilisation du Node Package Manager (**npm**)



Angular configuration

- Installer le CLI d'Angular
- CLI : Command Line Interface

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

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

- Sous Mac, possibilité d'erreur EACCES...

- ▶ Réessayer en suivant le guide :
[npm Docs](#)

```
ng v
```



```
Angular CLI: 17.0.0
Node: 20.9.0
Package Manager: npm 10.1.0
OS: darwin x64

Angular:
...
Package          Version
-----
@angular-devkit/architect    0.1700.0 (cli-only)
@angular-devkit/core         17.0.0 (cli-only)
@angular-devkit/schematics   17.0.0 (cli-only)
@schematics/angular          17.0.0 (cli-only)
```

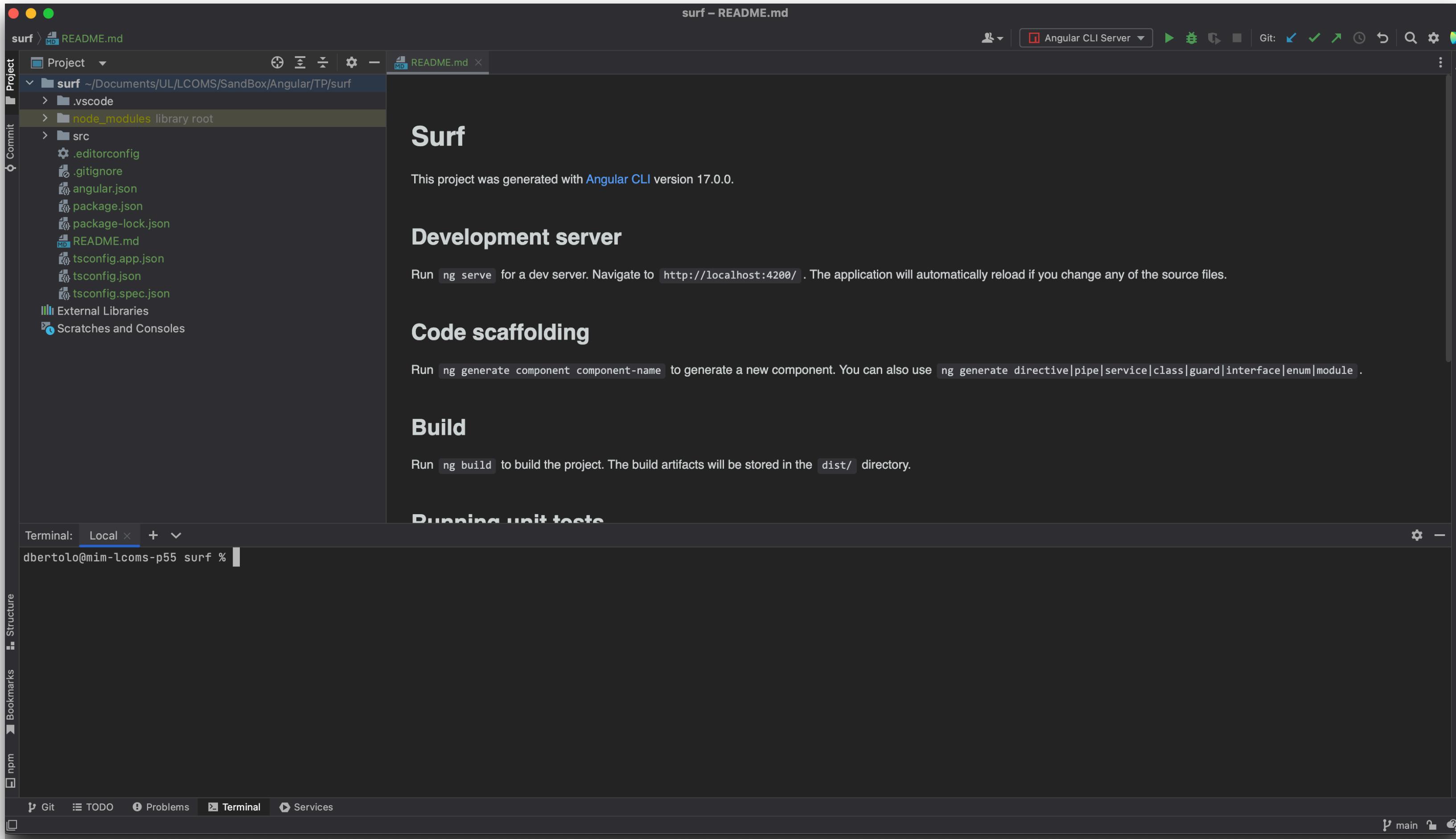
Création d'une application

```
ng new surf --standalone=no --style=scss --skip-tests=true
```

- Crédation avec le CLI
- Ne pas ajouter de **routing**
- Dans le terminal naviguer jusqu'au dossier de travail Angular
- Server-Side Rendering & Static Site Generation : **No**

```
[dbertolo@mim-lcoms-p55 TP % ng new surf --style=scss --skip-tests=true
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation
(SSG/Prerendering)? No
CREATE surf/README.md (1058 bytes)
CREATE surf/.editorconfig (274 bytes)
CREATE surf/.gitignore (548 bytes)
CREATE surf/angular.json (3345 bytes)
CREATE surf/package.json (1035 bytes)
CREATE surf/tsconfig.json (909 bytes)
CREATE surf/tsconfig.app.json (263 bytes)
CREATE surf/tsconfig.spec.json (273 bytes)
CREATE surf/.vscode/extensions.json (130 bytes)
CREATE surf/.vscode/launch.json (470 bytes)
CREATE surf/.vscode/tasks.json (938 bytes)
CREATE surf/src/main.ts (250 bytes)
CREATE surf/src/favicon.ico (15086 bytes)
CREATE surf/src/index.html (290 bytes)
CREATE surf/src/styles.scss (80 bytes)
CREATE surf/src/app/app.component.scss (0 bytes)
CREATE surf/src/app/app.component.html (20887 bytes)
CREATE surf/src/app/app.component.ts (366 bytes)
CREATE surf/src/app/app.config.ts (227 bytes)
CREATE surf/src/app/app.routes.ts (77 bytes)
CREATE surf/src/assets/.gitkeep (0 bytes)
✓ Packages installed successfully.

Author identity unknown]
```



IDE et projet

Serveur de développement

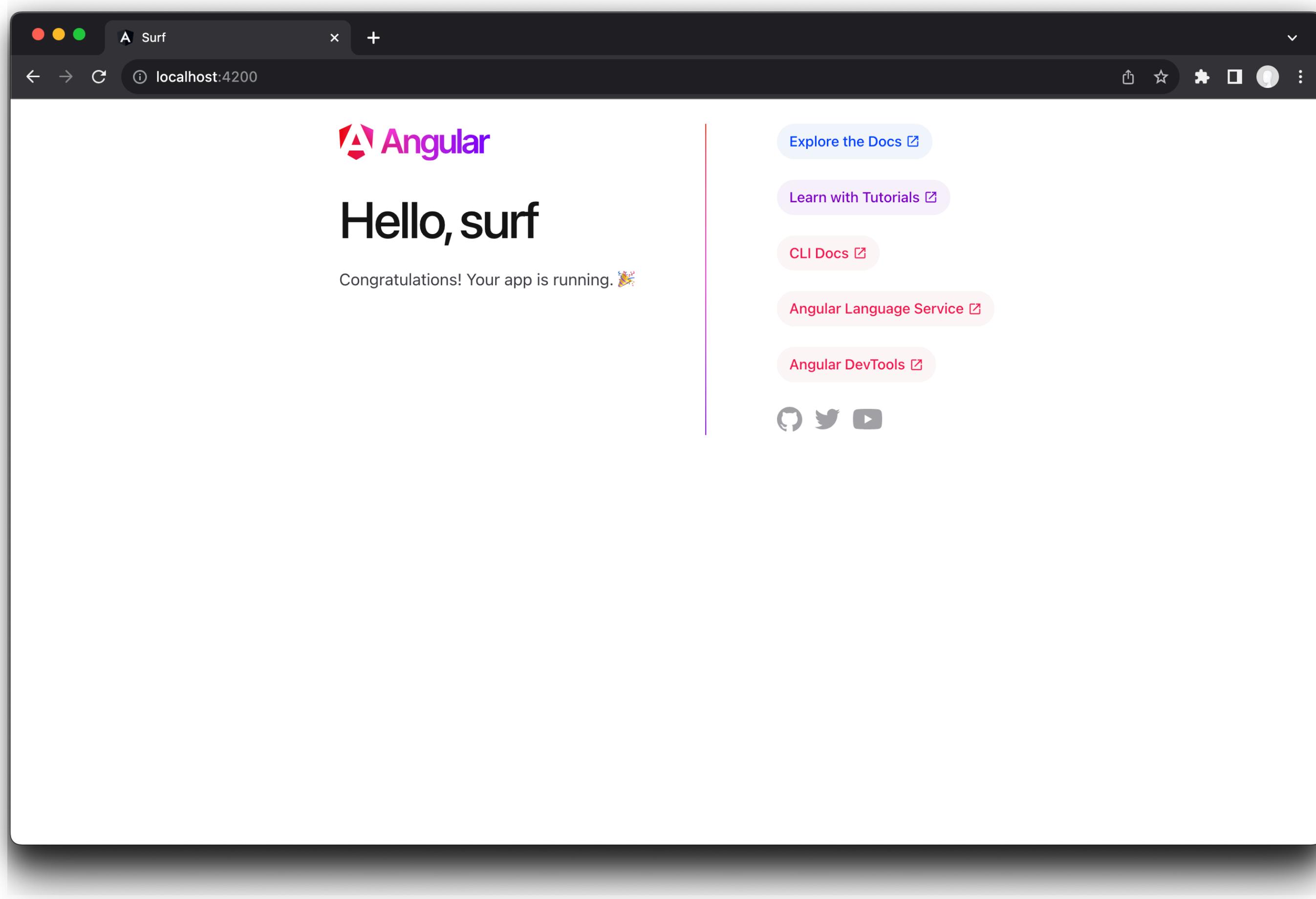
- A partir du terminal se positionner dans le dossier de l'application
- Executer : `ng serve`
- Compile et lance le serveur de développement

```
dbertolo@mim-lcoms-p55 surf % ng serve

Initial Chunk Files | Names           | Raw Size
polyfills.js        | polyfills       | 82.71 kB |
main.js             | main            | 23.37 kB |
styles.css          | styles          | 96 bytes |

| Initial Total | 106.18 kB

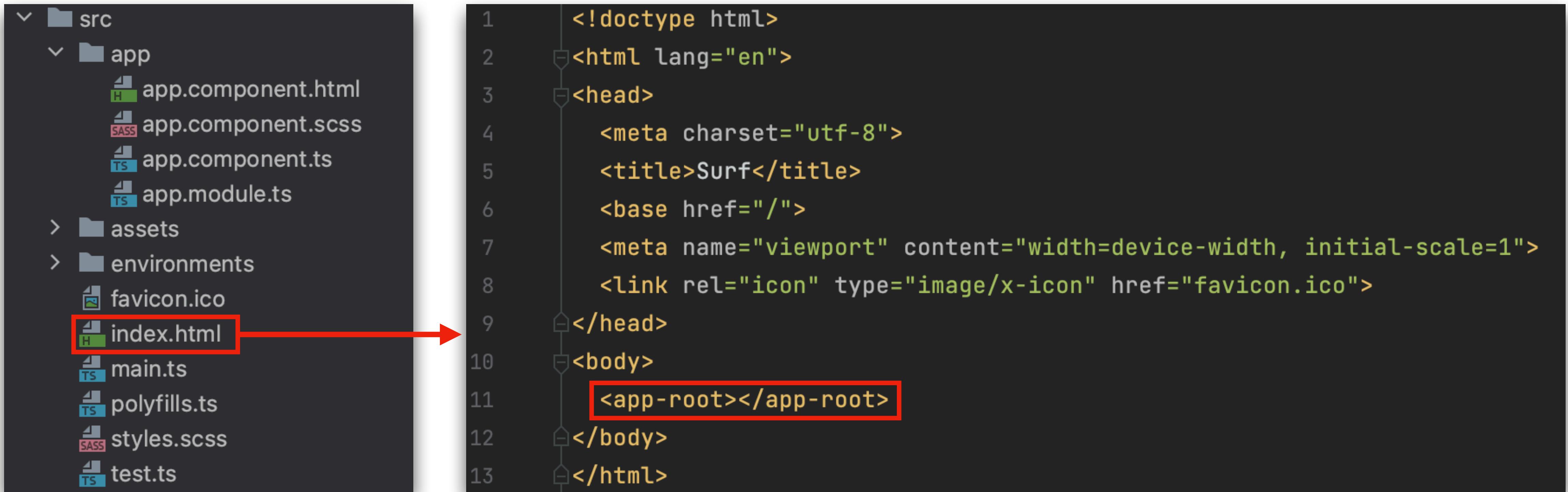
Application bundle generation complete. [4.192 seconds]
Watch mode enabled. Watching for file changes...
→ Local: http://localhost:4200/
```



Serveur de développement

<http://localhost:4200>

index.html



The image shows a file explorer on the left and a code editor on the right. In the file explorer, the 'src' directory contains 'app', 'assets', 'environments', 'favicon.ico', and 'index.html'. The 'index.html' file is highlighted with a red box and has a red arrow pointing to it in the code editor. The code editor displays the following HTML code:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Surf</title>
6     <base href="/">
7     <meta name="viewport" content="width=device-width, initial-scale=1">
8     <link rel="icon" type="image/x-icon" href="favicon.ico">
9   </head>
10  <body>
11    <app-root></app-root>
12  </body>
13 </html>
```

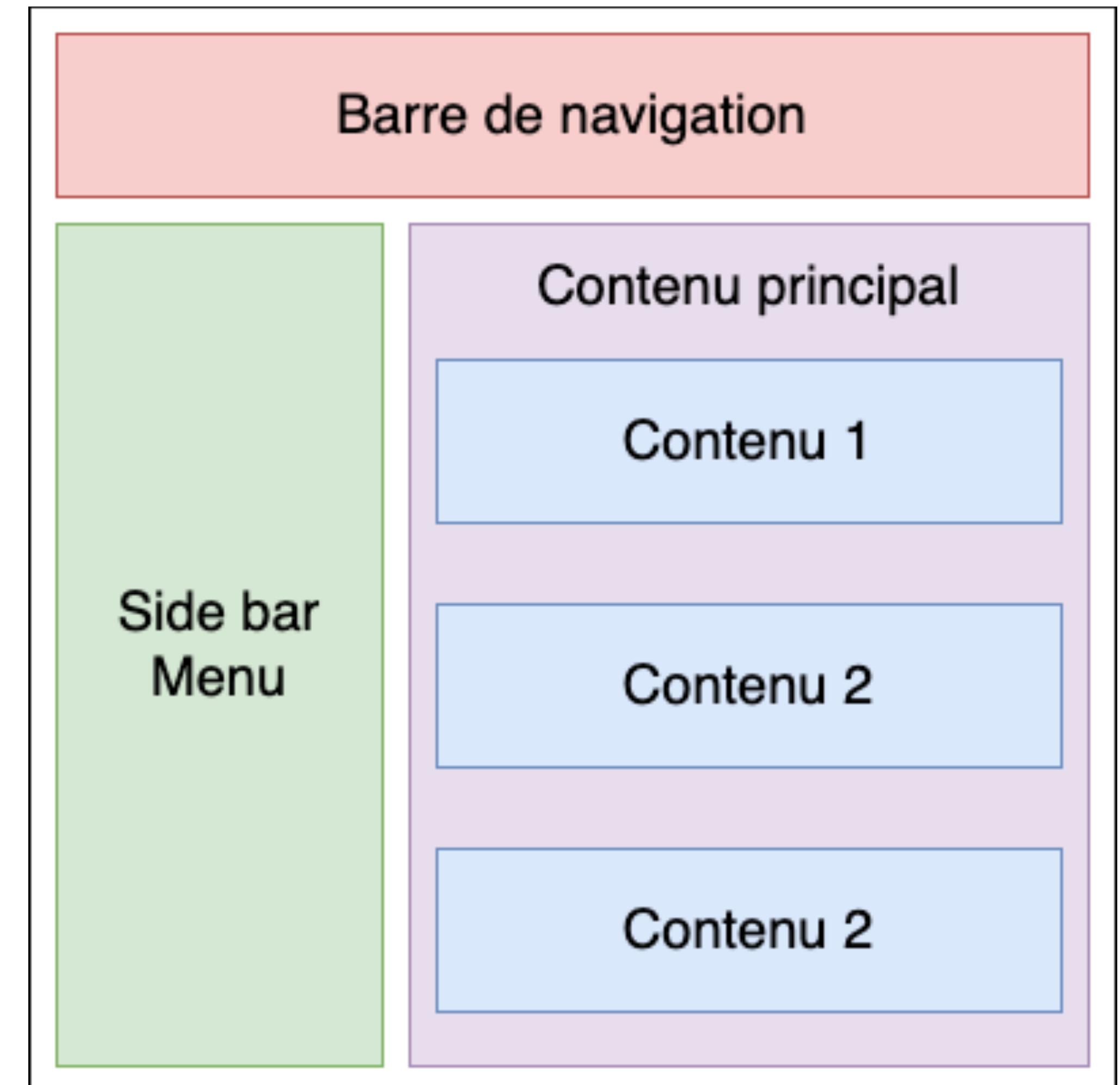
Component

Les composants

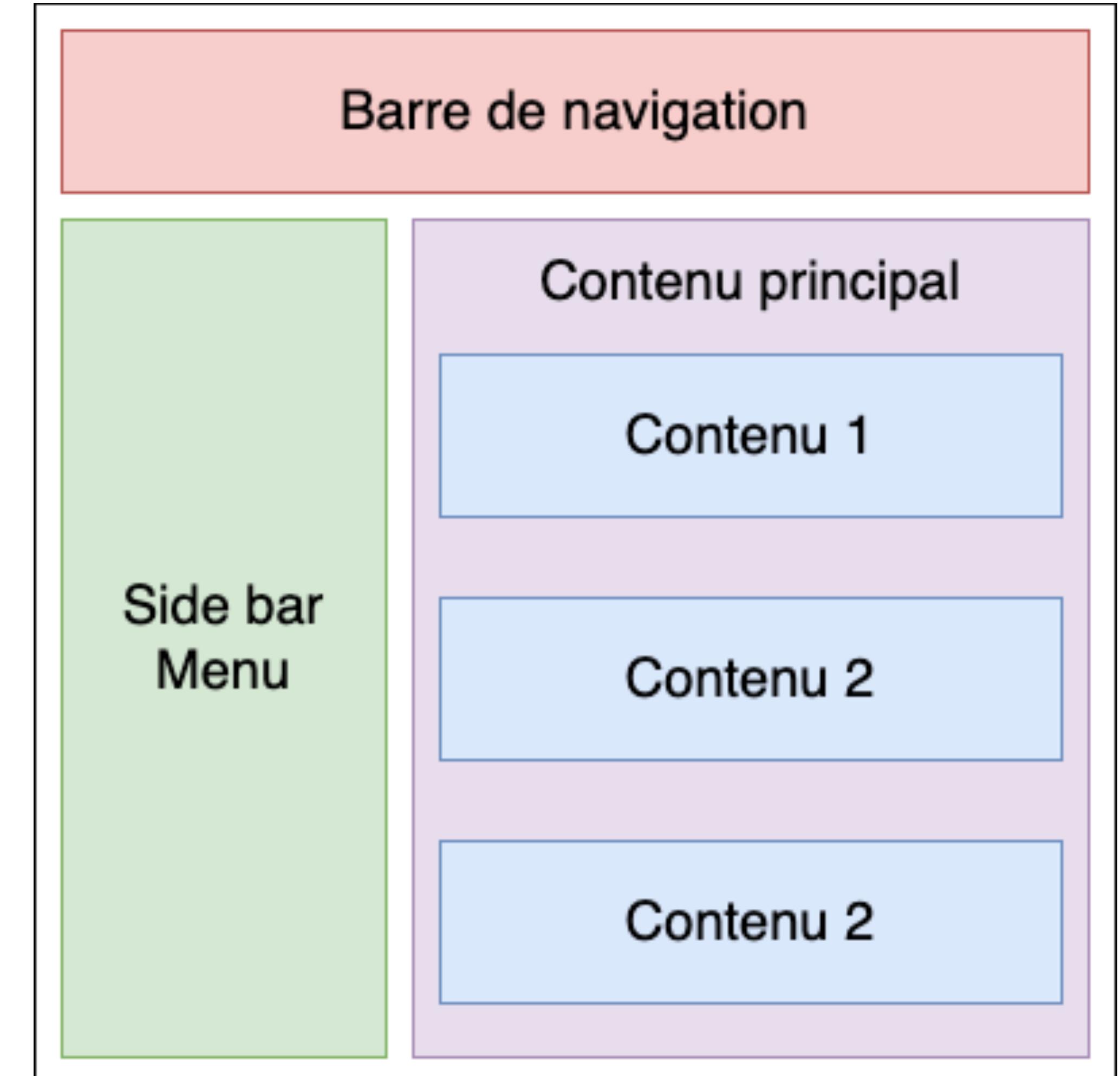
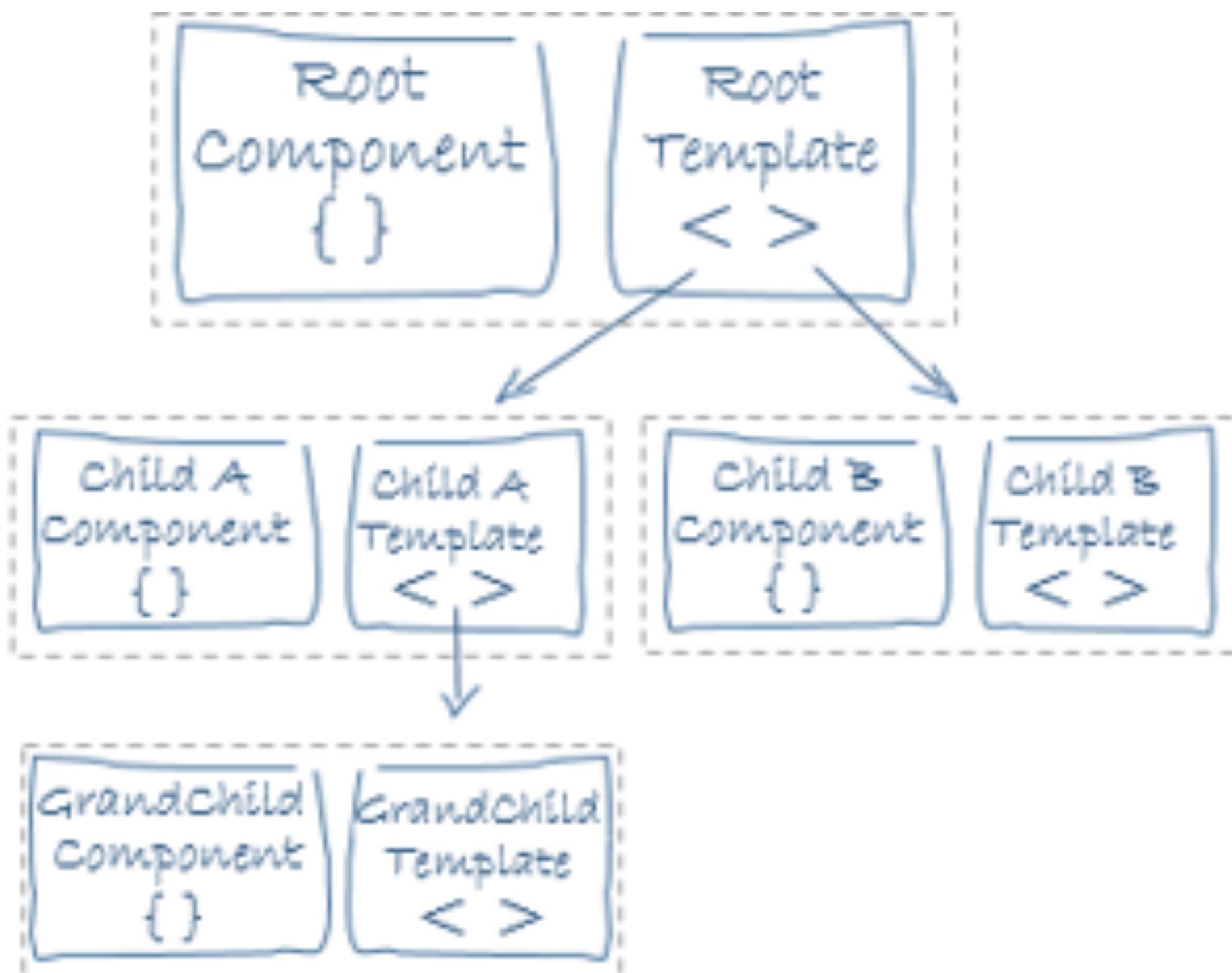


Component

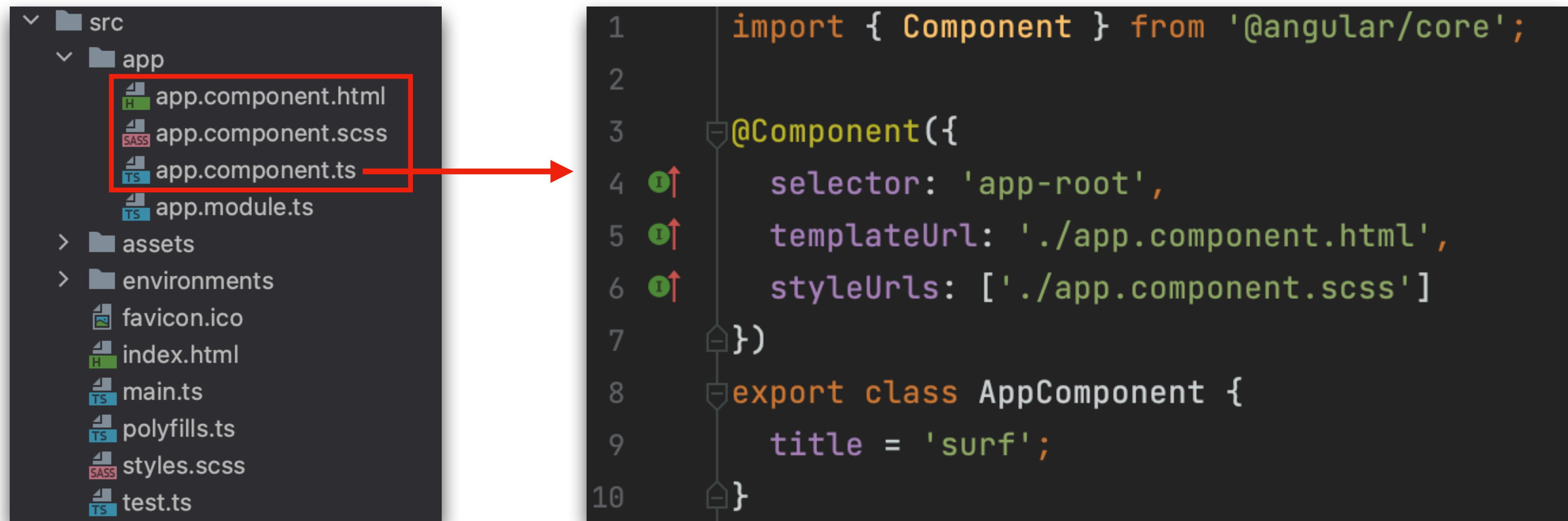
- Composant de base d'une application Angular
- Chaque rectangle pourrait être un Component
- Component réutilisable plusieurs fois
- Un Component peut en inclure un ou plusieurs autres



Component



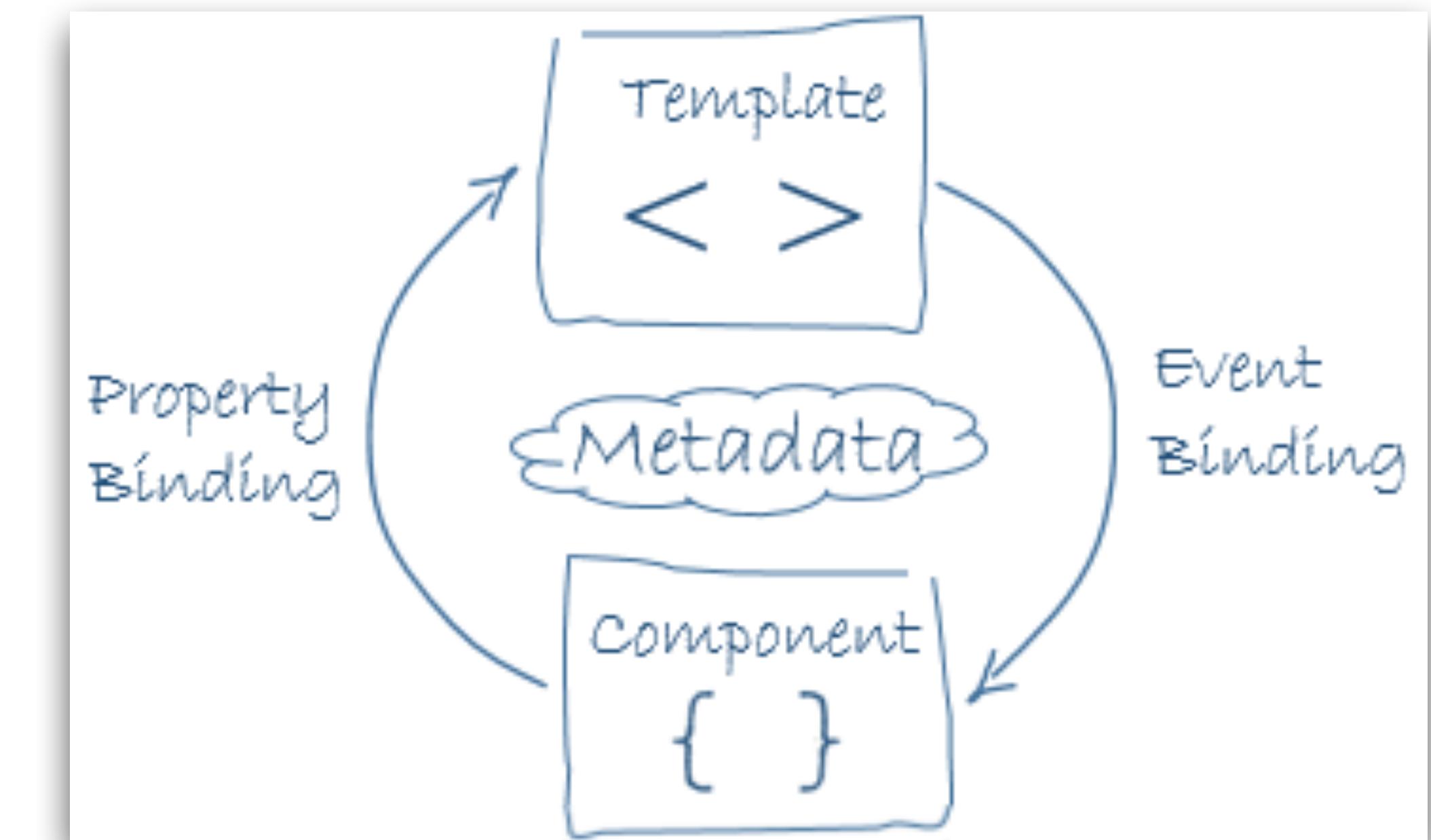
Root Component



Component

Un Component Angular est composé de trois fichiers

- **HTML** appelé **Template** contenant :
 - Le code HTML affiché dans le navigateur
 - Des instructions pour interagir avec le code TypeScript
- **TypeScript** contenant :
 - Les données du Component
 - La logique du Component
- **SCSS** contenant le style du Component



Création d'un Component

```
ng generate component nom_du_component
```

```
dbertolo@mim-lcoms-p55 surf % ng generate component surf-spot
CREATE src/app/surf-spot/surf-spot.component.scss (0 bytes)
CREATE src/app/surf-spot/surf-spot.component.html (24 bytes)
CREATE src/app/surf-spot/surf-spot.component.ts (287 bytes)
UPDATE src/app/app.module.ts (406 bytes)
```

Le CLI génère automatiquement un Component en :

- Crément les trois fichiers HTML, SCSS et TypeScript
- Ajoutant automatiquement Component au nom
- Mettant à jour le fichier app.module.ts (optionnel depuis v14)

Component

- La logique du Component utilise la syntaxe de classe de Javascript

```
export class SurfSpotComponent {  
  name: string;  
  location: string;  
  
  constructor() {  
    this.name = 'Nazaré';  
    this.location = 'Portugal'  
  }  
}
```

Component

- L'affichage du Component dans la page se fait à l'aide d'un décorateur :
 - ▶ **@Component**
- L'affichage dans le Template se fait avec le selector :

```
<app-surf-spot></app-surf-spot>
```

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

Component

The diagram illustrates the structure of a component named `surf-spot`. On the left, a file tree shows the directory `src/app/surf-spot` containing three files: `surf-spot.component.html`, `surf-spot.component.scss`, and `surf-spot.component.ts`. A red box highlights these three files, and a red arrow points from them to the corresponding code on the right.

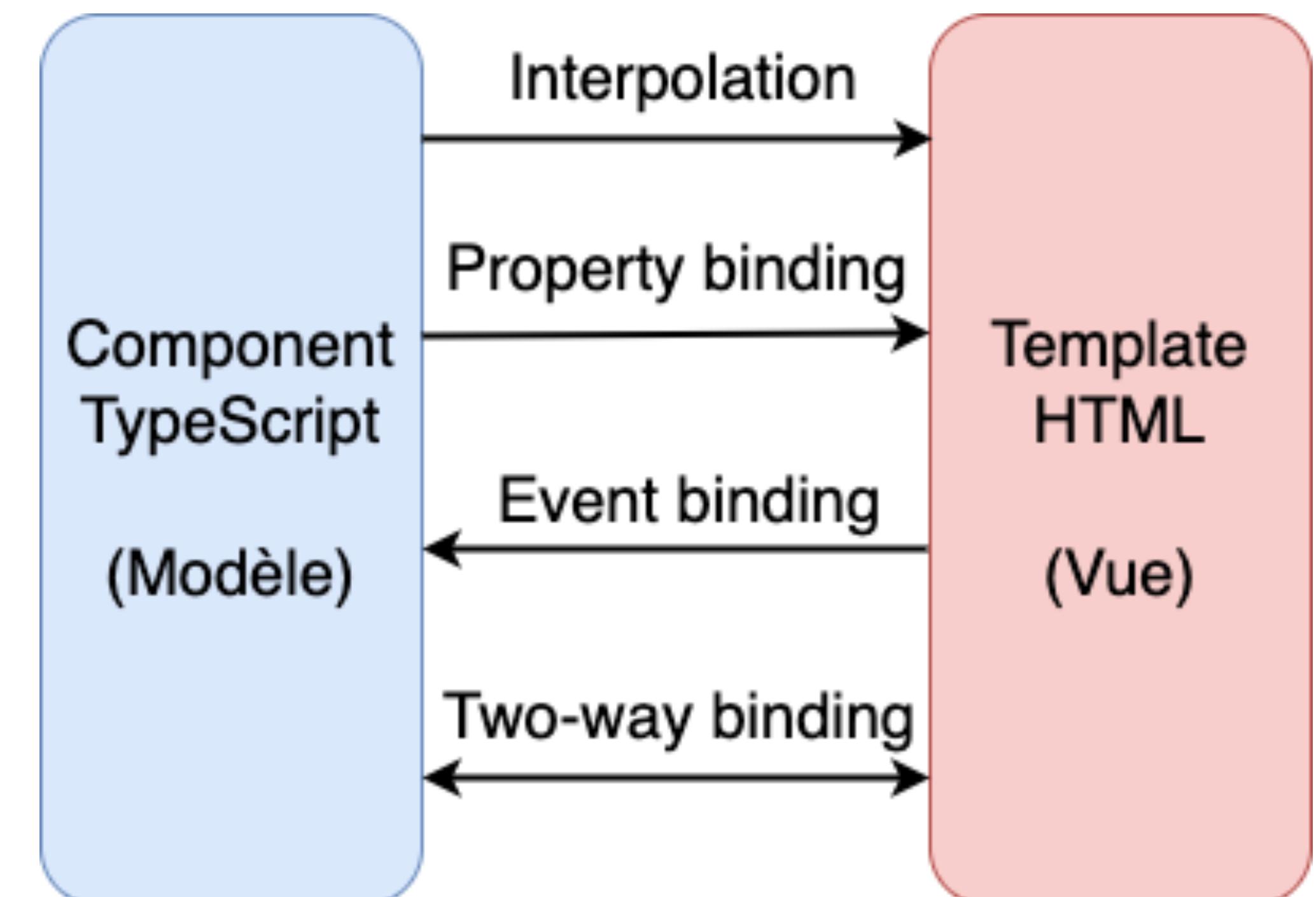
```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-surf-spot',
5   templateUrl: './surf-spot.component.html',
6   styleUrls: ['./surf-spot.component.scss']
7 })
8 export class SurfSpotComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit(): void {
13   }
14
15 }
```

Component DataBinding



Databinding & template

- Le binding permet l'échange de données entre la partie TypeScript et le template (HTML) du Component
- Quatre types de binding :
 - Interpolation
 - Property binding
 - Event binding
 - Two-way binding



Binding : modèle vers vue

- **Interpolation** : permet de transférer des données du TypeScript (modèle) dans le template HTML (vue)
 - ▶ Syntaxe : `{{ expression }}`

```
<div>Hello {{ name }}</div>


```

Binding : modèle vers vue

- **Property binding** : permet d'insérer des valeurs dans les propriétés du DOM

- ▶ Syntaxe : `[targetFooBar] = expression`

```
<img [src]="imageUrl" [alt]="description" />
```

- ▶ Cas constant

```
<show-title [title]="'My title'></show-title>  
<show-title title="My title"></show-title>
```

Binding : vue vers modèle

- **Event** : permet de transférer des données du Template HTML (vue) dans TypeScript (modèle)

- ▶ Syntaxe : `(event)="méthode()"`

```
<!-- app.component.html -->
<button type="button" (click)="onSave()">Save</button>
```

- ▶ Appelle la méthode quand un événement du DOM arrive

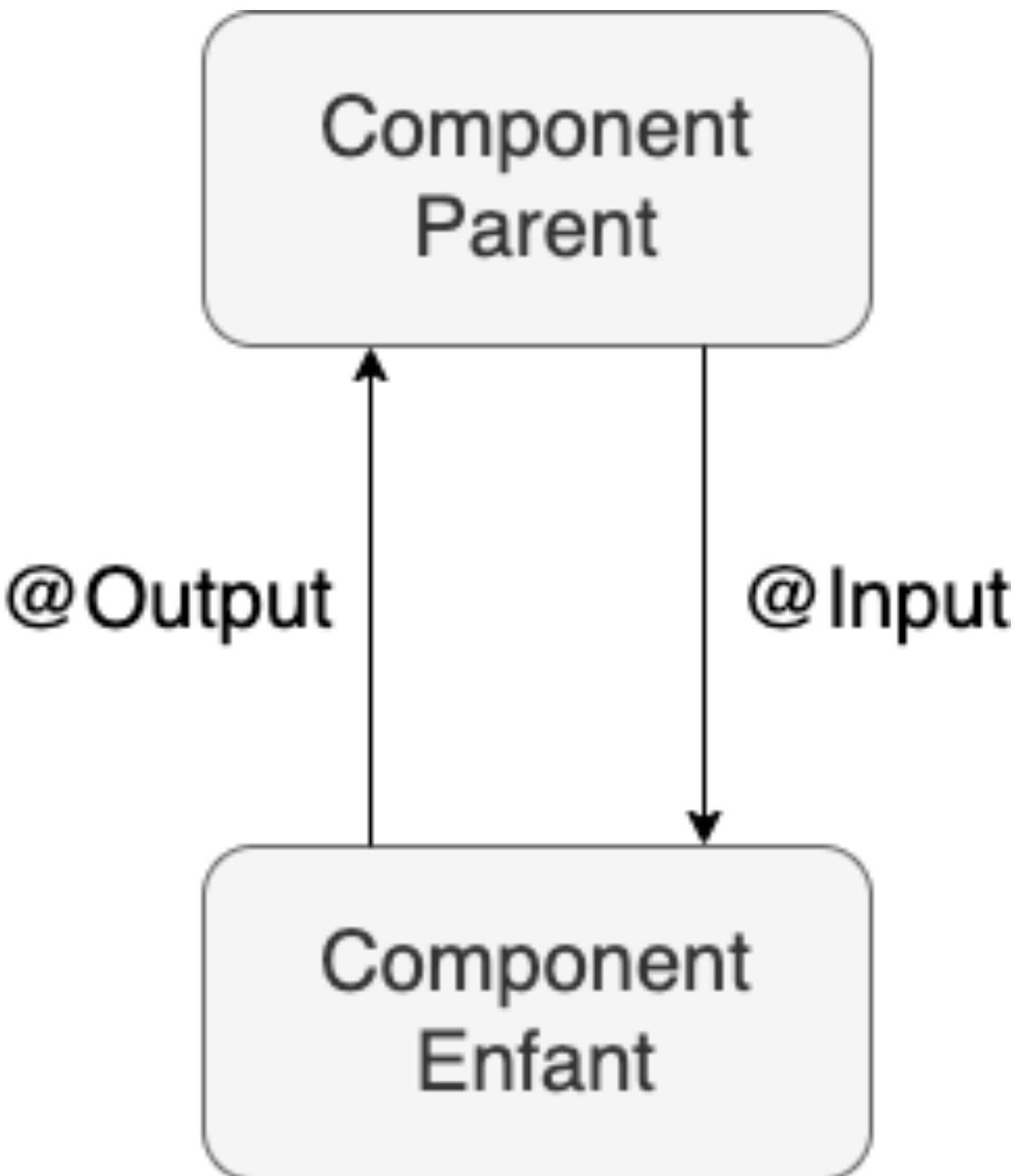
```
// app.component.ts
Export class AppComponent {
    onSave(): void { console.log('Saved'); }
}
```

Component lifecycle hooks

- **ngOnChanges()**: répond à chaque fois qu'une des propriétés en entrée changent
- **ngOnInit()**: initialise le composant/la directive après sa création (1 fois)
- **ngDoCheck()**: permet d'agir sur la détection des modifications
- **ngAfterContentInit()**: répond après l'initialisation du contenu du composant (1 fois)
- **ngAfterContentChecked()**: répond après qu'Angular ait vérifié le contenu du composant
- **ngAfterViewInit()**: répond après l'initialisation de la vue du composant (1 fois)
- **ngAfterViewChecked()**: répond après qu'Angular ait vérifié le contenu de la vue
- **ngOnDestroy()**: permet de faire du nettoyage avant la destruction du composant (1 fois)

Communication parent - enfant

- Communication du parent vers l'enfant, utilisation du décorateur :
 - ▶ **@Input**
- Communication de l'enfant vers le parent, utilisation du décorateur :
 - ▶ **@Output**



Com. parent - enfant : @Input

Component - Enfant
child.component.ts

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

@Component({
  selector: "app-child"

})

export class ChildComponent {
  @Input() name: string;
}
```

Component - Parent
parent.component.html

```
<section>
  <app-child [name]="person.name"></app-child>
</section>
```

Com. parent - enfant : @Output

Component - Enfant
child.component.ts

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

@Component({
  selector: 'app-child'
})

export class ChildComponent {
  @Output() childEvent: EventEmitter<string>;
  constructor() {
    this.childEvent = new EventEmitter<string>();
  }
  raiseEvent() {
    this.childEvent.emit('event from child');
  }
}
```

Com. parent - enfant : @Output

Component - Parent
parent.component.html

```
<section><app-child (childEvent)="onChildEvent($event)"></app-child></section>
```

parent.component.ts

```
onChildEvent(value: string): void {  
  console.info(value); // event from child  
}
```

Les directives



Les directives

- Classe qui ajoute des comportements à des éléments
- Il existe deux types de **directives** :
 - **Structurelles** (avec * devant le nom de la directive) : *ngIf, *ngFor et *ngSwitch
 - **Attribute directives** : change le comportement ou l'apparence des éléments du DOM ou des composants Angular
 - **Built-in** attribute directives : [ngStyle], [ngClass] et [ngModel]

Directives structurelles

- * matérialise les directives structurelles (comme NgIf NgFor et NgSwitch)
- Indique que l'on utilise la balise 'ng-template'
- Attention à ne pas oublier les [] si vous utilisez les templates

```
<div *ngIf="errorCount > 0">toto</div>

<!-- same as -->

<ng-template [ngIf]="errorCount > 0">
    <div>toto</div>
</ng-template>
```

- Doc Angular : <https://angular.io/guide/structural-directives>

Directives structurelles - *ngIf

- Change la structure du DOM de manière conditionnelle (apparaît ou non dans le DOM)

```
<div *ngIf="errorCount > 0" class="error">  
  {{ errorCount }} errors detected  
</div>
```

- API > @angular/common : <https://angular.io/api/common/NgIf>

Directives structurelles - *ngFor

- Itère dans une collection et génère un template par élément
- index, count, first, last, event et odd à utiliser en alias dans des variables
 - ▶ Syntaxe : *ngFor="let element of array"

```
<ul>
    <li *ngFor="let fruit of fruits; let i=index">
        {{ i }} : {{ fruit.name }}
    </li>
</ul>
```

- API > @angular/common : <https://angular.io/api/common/NgForOf>

Directives structurelles - *ngSwitch

- Change la structure du DOM de manière conditionnelle (à la manière d'un switch case)

```
<div [ngSwitch]="display">  
  <p *ngSwitchCase="list">...</p>  
  <p *ngSwitchCase="whenExpression1">...</p>  
  <!--default case when there are no matches -->  
  <p *ngSwitchDefault>...</p>  
</div>
```

- API > @angular/common : <https://angular.io/api/common/NgSwitch>

Built-in attribute directives

- Les directives d'attributs écoutent et modifient le comportement d'autres éléments, attributs, propriétés et composants HTML
 - ▶ [ngStyle] : ajoute et supprime un ensemble de styles HTML
 - ▶ [ngClass] : ajoute et supprime un ensemble de classes CSS
 - ▶ [ngModel] : ajoute une liaison de données bidirectionnelle (two-way binding) à un élément de formulaire HTML
- Doc Angular : <https://angular.io/guide/built-in-directives>

Built-in attribute directives - [ngStyle]

- Ajoute et supprime un ensemble de styles HTML
- [ngStyle] permet de définir plusieurs styles en ligne simultanément, en fonction de l'état du composant

```
<span [ngStyle]="{ color: 'rgb(0, 0, ' + surfSpot.like + ')' }">👍 {{ surfSpot.like }}</span>
```

- Doc Angular : <https://angular.io/guide/built-in-directives#ngstyle>

Built-in attribute directives - [ngClass]

- Ajoute et supprime un ensemble de classes CSS
 - Syntaxe : <div [ngClass]="{ 'class-name': condition }"></div>

```
<!-- Template HTML -->
<div class="surf-spot-card" [ngClass]="{ liked: buttonText == 'Unlike' }">

<!-- Fichier SCSS -->
.liked {
  //Définition des éléments de la classe CSS
}
```

- Doc Angular : <https://angular.io/guide/built-in-directives#ngClass>

Pipes

Affichage des données



Pipes

- Permet de transformer des chaînes de caractères, des montants en devises, des dates et d'autres données à afficher
- Fonctions simples à utiliser dans les expressions de modèles pour accepter une valeur d'entrée et renvoyer une valeur transformée
- **Built-in pipes** : DatePipe,UpperCasePipe,LowerCasePipe, TitleCasePipe,CurrencyPipe,DecimalPipe,PercentPipe...
 - ▶ Plus les autres : <https://angular.io/api/common#pipes>
- Doc Angular : <https://angular.io/guide/pipes-overview>

Pipes - Syntaxe

- A la suite d'une expression d'interpolation

```
<span> {{ expression | filter1 }}</span>
```

- On peut chaîner les pipes

```
<span> {{ expression | filter1 | filter2 }}</span>
```

- On peut passer des paramètres aux pipes

```
<span>{{ expression | filter1:param1:param2 }}</span>
```

Pipes - Formater les dates

- Formate une date selon un certain format et selon une locale

```
 {{ myDate | date:format }}
```

- Le filtre accepte un format (string) en argument

```
<span>Dernière session le {{ surfSpot.myDate | date }}</span>
<!-- Dernière session le Nov 9, 2022 -->
```

```
<span>Dernière session le {{ surfSpot.myDate | date: 'longDate' }}</span>
<!-- Dernière session le November 8, 2022 -->
```

```
<span>Dernière session le {{ surfSpot.myDate | date: 'dd/MM/yy, à HH:mm' }}</span>
<!-- Dernière session 08/11/22, à 21:01 -->
```

```
<p>Dernière session le {{ surfSpot.myDate | date: 'à HH:mm, le d MMMM yyyy' }}</p>
<!-- Dernière session à 15:07, le 9 November 2022 -->
```

Pipes - Localisation

- Changer la localisation de l'application en français
- Code à insérer dans app.module.ts
- fr-FR pour la localisation en français - France

```
1 import {LOCALE_ID, NgModule} from '@angular/core';
2 import { BrowserModule } from '@angular/platform-browser';
3 // import pour la localisation en fr
4 import { registerLocaleData} from "@angular/common";
5 import * as fr from '@angular/common/locales/fr'
6
7 import { AppComponent } from './app.component';
8 import { SurfSpotComponent } from './surf-spot/surf-spot.component';
9
10 @NgModule({
11   declarations: [
12     AppComponent,
13     SurfSpotComponent
14   ],
15   imports: [
16     BrowserModule
17   ],
18   providers: [
19     { provide: LOCALE_ID, useValue: 'fr-FR' }
20   ],
21   bootstrap: [AppComponent]
22 })
23 export class AppModule {
24   constructor() {
25     registerLocaleData(fr.default);
26   }
27 }
```

Pipes - Formater les nombres

```
this.myNumber = 1234567.89
```

```
<span>{{ myNumber }}</span>
<!-- 1234567.89 -->
```

```
<span>{{ myNumber | number }}</span>
<!-- 1 234 567,89 -->
```

```
<span>{{ myNumber | number: '1.0-0' }}</span>
<!-- 1 234 568 -->
```

```
<span>{{ myNumber | number: '1.0-1' }}</span>
<!-- 1 234 567,9 -->
```

Pipes - Formater les nombres

```
this.myPercent = 0.456  
this.amount = 1234.56
```

```
<span>{{ myPercent | percent }}</span>  
<!-- 46% -->
```

```
<span>{{ myPercent | percent: '1.0-1' }}</span>  
<!-- 45,6% -->
```

```
<span>{{ amount | currency }}</span>  
<!-- 1 234,56 $US -->
```

```
<span>{{ amount | currency: 'EUR' }}</span>  
<!-- 1 234,56 € -->
```

```
<span>{{ amount | currency: 'EUR' : code }}</span>  
<!-- 1 234,56 EUR -->
```

Pipes personnalisés

- Un Pipe se compose de :
 - ▶ une classe décorée par le décorateur **@Pipe** ayant comme propriété obligatoire **name**, propriété qui servira à appeler le pipe dans un template HTML
 - ▶ Cette classe implémente la méthode **transform** qui prend en paramètre une valeur et éventuellement un tableau d'arguments
 - ▶ Cette méthode **transform** effectue des transformations (ou pas) mais retourne toujours une nouvelle valeur
- Enregister le Pipe personnalisé dans le tableau déclaration du module (comme un composant)

Pipes personnalisés

```
// multiply-by-two.pipe.ts
import {Pipe} from '@angular/core';
@Pipe({
  name: 'multiplyByTwo'
})
export class MultiplyByTwoPipe implements PipeTransform {
  transform(value: number, args: any[]) {
    return value * 2;
  }
}
```

```
// app.module.ts
import { NgModule } from '@angular/core';
import { MultiplyByTwoPipe } from './shared/multiply-by-two.pipe'

@NgModule({
  ...
  declarations: [
    MultiplyByTwoPipe
  ],
  ...
})
export class AppModule {}
```

```
<!-- app.component.html -->
<p>{{ value | multiplyByTwo }}</p>
```

Les services



Les services

- Un service est une classe exportée
- Déclaration de la classe comme service à l'aide d'un décorateur importé depuis @angular/core :
 - ▶ `@Injectable()`
- permet de partager des données

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

@Injectable({
  providedIn: 'root'
})

export class TodoService {
  constructor() {
    this.name = 'Hello';
  }

  getName() {
    return this.name;
  }
}
```

Les services

- Pour injecter un service dans un component, on ajoute un argument au constructor du component qui a le type du service
 - ▶ `private userService: UserService`
- Permet de centraliser des interactions de façon modulaire sous forme de méthodes

Routing Navigation



Le routing

- Dans une application à page unique (SPA), toutes les fonctions de l'application existent dans une seule page HTML
- Le navigateur doit rendre uniquement les parties qui comptent pour l'utilisateur
 - pas de chargement d'une nouvelle page.
- Améliorer l'expérience utilisateur de l'application.
- Utiliser des routes pour définir la manière dont les utilisateurs naviguent dans l'application
 - pour définir comment les utilisateurs naviguent d'une partie de l'application à une autre
 - pour éviter tout comportement inattendu ou non autorisé

Le routing - Mise en place

- Importer RouterModule de @angular/router
- Définir les routes
- Mettre à jour le composant avec router-outlet
- Contrôler la navigation avec des éléments d'interface utilisateur (UI)
- Identifier la route active
- Ajouter une redirection
- Ajouter une page 404

Le routing - Configuration

- Un module de routing contient un tableau de type Routes qui contient les routes de l'application
- Une route est un objet qui spécifie le component à afficher pour chaque route
 - ▶ Syntaxe : { path: 'myPath', component: MyComponent }

```
const ROUTES: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'people', component: PeopleComponent },
  { path: 'people/:id', component: PersonComponent },
  { path: '**', component: NotFoundComponent }
];
```

Le routing - Configuration

- **path:** l'URL de route (ex: /people/:id)
- **component:** le composant associé à cette route (ex: PeopleComponent)
- **redirectTo:** le fragment d'URL vers lequel rediriger la route courante (ex: '/home')
- **pathMatch:** stratégie de redirection (full / prefix)
 - ▶ full: tente une reconnaissance depuis la racine de la route
 - ▶ prefix: tente une reconnaissance partielle de la route

Le routing - Configuration

Il existe d'autres options de configuration, permettant de réaliser du routing plus avancé :

- **outlet** : le nom de l'emplacement dans lequel le composant doit s'afficher
- **data** : données passées à la route via ActivatedRoute
- **canActivate / canDeactivate** : permet d'activer ou non la route
- **resolver** : récupère des données avant de naviguer vers la route
- **children** : un tableau de définition des sous-routes

Le routing - Mise en place

- Enregister les routes dans le routeur Angular en passant le tableau de routes avec RouterModule.forRoot()
- Deux stratégies :
 - ▶ Par 'Path', aussi nommée **PathLocationStrategy** (Mode HTML5 et pushState => Par défaut)

Exemple : localhost:4200/people/1 => { useHash: false }

- ▶ Par 'Hash', aussi nommée **HashLocationStrategy**

Exemple : localhost/#/people/1 => { useHash: true }

```
// app.module.ts
RouterModule.forRoot(routes, { useHash: true });
```

Le routing - Mise en place

```
import {RouterModule, Routes} from "@angular/router";
import {NgModule} from "@angular/core";
import {LandingPageComponent} from "./landing-page/landing-page.component";
import {SurfSpotListComponent} from "./surf-spot-list/surf-spot-list.component";

const routes: Routes = [
  {path: 'spots', component: SurfSpotListComponent},
  {path: '', component: LandingPageComponent}
];

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

export class AppRoutingModule {}
```

Le routing - Mise en place

```
// app.module.ts
import ...
// import pour la localisation en fr
import ...

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

- On enregistre le module de routing dans AppModule pour ajouter le routeur configuré à l'application
- On utilise une balise <router-outlet> pour dire à quel niveau du template le component de la route active doit être inséré.

```
<!-- app.component.html -->
<app-header></app-header>
<router-outlet></router-outlet>
```

Le routing - Dans le template

- routerLink : directive qui permet de créer des liens pour passer d'une route à l'autre

```
<a class="btn btn-info" routerLink="/people">Movies Liste</a>  
  
<a class="btn btn-info" [routerLink]=["/people"]>Movies Liste</a>  
  
<a class="btn btn-info" [routerLink]=["/people/edit/", person.id]>Edit</a>
```

- routerLinkActive : ajoutez des classes CSS aux liens correspondants à la route activée
 - ▶ Ignorer l'activation des routes enfants :
[routerLinkActiveOptions]="{ exact: true }"

```
<nav>  
  <a routerLink="home" routerLinkActive="active">Home</a>  
  <a routerLink="spots" routerLinkActive="active" [routerLinkActiveOptions]="{{exact: true}}>Spots</a>  
</nav>
```

Le routing - Dans le Typescript

- Possibilité d'injecter le Router dans les components
- `navigateByUrl()` : méthode de Router pour de la navigation programmatique.

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

@Component({
  selector: 'app-landing-page',
  templateUrl: './landing-page.component.html',
  styleUrls: ['./landing-page.component.scss']
})
export class LandingPageComponent implements OnInit {

  constructor(private router: Router) { }

  ngOnInit(): void {
  }

  onContinue() {
    this.router.navigateByUrl('spots');
  }
}
```

Le routing - Dans le Typescript

- `snapshot.params` : permet de récupérer le paramètre `id` d'une route activée en injectant `ActivatedRoute`
 - ▶ Les paramètres d'une route sont toujours de type `string`
- Pour naviguer vers une route absolue (et non relative) : ajouter un `/` au début de la route demandée.

```
import {Component, Input, OnInit} from '@angular/core';
import {SurfSpotModel} from "../models/surf-spot.model";
import {SurfSpotsService} from "../services/surf-spots.service";
import {ActivatedRoute} from "@angular/router";

@Component({
  selector: 'app-surf-spot-details',
  templateUrl: './surf-spot-details.component.html',
  styleUrls: ['./surf-spot-details.component.scss']
})
export class SurfSpotDetailsComponent implements OnInit {

  surfSpot!: SurfSpotModel;
  buttonLabel!: string;

  constructor(private surfSpotsService: SurfSpotsService,
    private route: ActivatedRoute) {
  }

  ngOnInit(): void {
    this.buttonLabel = 'Like';
    const spotId = +this.route.snapshot.params['id'];
    this.surfSpot = this.surfSpotsService.getSpotsById(spotId);
  }

  onLike() {...}
}
```

Le routing - Références

- Utiliser des routes Angular dans une application à page unique
 - ▶ <https://angular.io/guide/router-tutorial#import-routermodule-from-angularrouter>
- Tâches courantes de routing
 - ▶ <https://angular.io/guide/router>
- API @angular/router
 - ▶ <https://angular.io/api/router>

Le routing - Références

- Router références
 - ▶ <https://angular.io/guide/router-reference>
- Ajouter une navigation par routing
 - ▶ <https://angular.io/tutorial/toh-pt5>

Les formulaires



Les formulaires

- Il existe deux types de formulaire en Angular :
 - ▶ **Template driven form** : formulaire piloté par le template
 - ▶ **Reactive forms** : formulaire piloté par la logique composant

Les formulaires

Template driven form



Template driven form

Importer dans AppModule le module **FormsModule** provenant du package **@angular/forms**

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

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

Template driven form - Two-way binding

- Deux composantes obligatoires :
 - ▶ **ngModel** : le binding d'un contrôle
 - ▶ **name** : associe un nom au contrôle du champ
- Exemple 1 : binding View → Model

```
<input type="text" name="title" ngModel />
```
- Exemple 2 : binding Model → View

```
<input type="text" name="title" [ngModel]="surfSpot.name" />
```
- Exemple 3 : Two-way binding

```
<input type="text" name="title" [(ngModel)]="surfSpot.name" />
```

Template driven form

- On déclare une référence sur un formulaire prenant comme valeur la directive ngForm
 - ▶ #myForm="ngForm"
- myForm.value permet de récupérer en format JSON objet, toutes les valeurs des champs du formulaire

```
// Dans le Typescript
onSubmitForm(form: NgForm) {
    console.log(form.value);
}

<!-- dans le template -->
<form #myForm="ngForm" (ngSubmit)="onSubmitForm(myForm)"></form>
```

Template driven form

- Regrouper les champs dans un sous objet grâce à **ngModelGroup**

```
<div ngModelGroup="address">  
    <p><input ngModel name="city" /></p>  
    <p><input ngModel name="postalCode" /></p>  
</div>
```

Template driven form - Validation

- La validation se fait à l'aide des attributs html classiques de validation :
 - ▶ required
 - ▶ minLength
 - ▶ maxLength
 - ▶ pattern
 - ▶ ...
- Ok mais si un champ est requis en fonction d'un autre ?
 - ▶ on utilise la propriété liée à l'attribut

```
<input type="text" name="lastname" [(ngModel)]="person.lastname"
[required]="person.firstname" />
```

Différents états d'un contrôle

- control.pristine : l'utilisateur n'a pas interagi avec le contrôle
- control.dirty : l'utilisateur a déjà interagi avec le contrôle
- control.valid : le contrôle est valide
- control.invalid : le contrôle n'est pas valide
- control.touched : le contrôle a perdu le focus
- control.untouched : le contrôle n'a pas encore perdu le focus

Différentes classes pour le style

- .ng-valid / .ng-invalid
- .ng-pristine / .ng-dirty
- .ng-pristine / .ng-dirty
- .ng-touched / .ng-untouched

Gérer les erreurs

```
<input type="text" name="user" ngModel #userRef="ngModel" required>  
<div *ngIf="userRef.errors?.['required']">  
  <span class="help-block">Ce champ est obligatoire</span>  
</div>
```

Les formulaires

Reactive forms



Reactive forms

Importer dans AppModule le module **ReactiveFormsModule** provenant du package **@angular/forms**

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

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

Reactive forms - Syntaxe

```
// form.component.ts

import { Validators, FormControl, FormGroup } from '@angular/forms';
import { EditForm } from './edit-form.model';

@Component({...})

export class FormComponent {

  editForm: FormGroup<EditForm>;

  constructor() {
    this.editForm = new FormGroup({
      firstname: new FormControl('',
        [Validators.required, Validators.minLength(2)])
    })
  }

}
```

```
<!-- form.component.html -->

<form [formGroup]="editForm">

  <input type="text" formControlName="firstname">

  <div *ngIf="!editForm.controls.firstname.valid">
    Firstname est d'un format invalid
  </div>

  <button type="submit" [disabled]="!editForm.valid">
    Modifier
  </button>

</form>
```

Reactive forms - Syntaxe Typescript

- Possibilité 1 : en utilisant **FormGroup** et **FormControl**

```
import { FormControl, FormGroup } from '@angular/forms';
import { EditForm } from './edit-form.model';
@Component({...})
export class FormComponent {
  editForm: FormGroup<EditForm>;
  constructor() {
    this.editForm = new FormGroup({
      address: new FormGroup({
        zipCode: new FormControl(''),
        country: new FormControl('')
      })
    })
  }
}
```

Reactive forms - Syntaxe Typescript

- Possibilité 2 : en utilisant **FormGroup** et **FormBuilder**

```
...
import {FormBuilder, FormGroup} from "@angular/forms";
...

@Component({ ... })
export class NewSurfSpotComponent implements OnInit {

  surfSpotForm!: FormGroup;

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit(): void {
    this.surfSpotForm = this.formBuilder.group({
      name: [null],
      description: [null],
      imageUrl: [null],
      imageCaption: [null],
      location: [null]
    });
  }
}
```

Reactive forms - Syntaxe template

- Référence au modèle du formulaire via **formGroup**
- Mapping de controls via **formControlName**

```
<form [formGroup]="editForm">

  <label for="name">Nom du Spot</label>

  <input id="name" type="text" formControlName="name">

  <button type="submit" (click)="onSubmitForm()">Enregistrer</button>

</form>
```

Reactive forms - Syntaxe template

- Regrouper les champs dans un sous objet grâce à **formGroupName**

```
<form [formGroup]="editForm" (ngSubmit)="submitEditForm(editForm.value)">  
  <div formGroupName="address">  
    <input formControlName="zipCode" type="text" />  
    <input formControlName="country" type="text" />  
  </div>  
</form>
```

Reactive forms - Validation

- La validation des Reactive forms se fait à l'aide de **Validators**
- Les validators s'ajoutent à la configuration des champs dans le Typescript

Reactive forms - Validation

- Possibilité 1 : en utilisant **FormGroup** et **FormControl**

```
import { Validators, FormControl, FormGroup } from '@angular/forms';
import { EditForm } from './edit-form.model';
@Component({...})
export class FormComponent {
  editForm: FormGroup<EditForm>;
  constructor() {
    this.editForm = new FormGroup({
      address: new FormGroup({
        zipCode: new FormControl('', Validators.required),
        country: new FormControl('', Validators.required),
      })
    })
  }
}
```

Reactive forms - Validation

- Possibilité 2 : en utilisant **FormGroup** et **FormBuilder**

```
...
import {Validators, FormBuilder, FormGroup} from "@angular/forms";
...

@Component({ ... })
export class NewSurfSpotComponent implements OnInit {

    surfSpotForm!: FormGroup;

    constructor(private formBuilder: FormBuilder) { }

    ngOnInit(): void {
        this.urlRegex = myRegex;
        this.surfSpotForm = this.formBuilder.group({
            name: [null, Validators.required],
            description: [null, Validators.required],
            imageUrl: [null,[Validators.required, Validators.pattern(this.urlRegex)]],
            imageCaption: [null],
            location: [null, Validators.required]
        });
    }
}
```

Reactive forms - Validation

```
<form [formGroup]="surfSpotForm">  
  <div>  
    <input formControlName="name">  
    <div *ngIf="surfSpotForm.controls.name.errors?.required">  
      <span class="help-block">Ce champ est obligatoire</span>  
    </div>  
    <button type="submit" (click)="onSubmitForm()" [disabled]="surfSpotForm.invalid">  
      Enregistrer</button>  
  </div>  
</form>
```

Les observables



Les observables

- Un Observable est un objet qui **émet des valeurs au cours du temps**
- **Typé** => valeurs toujours du même type
- Peut émettre une **erreur** => l'Observable est détruit et n'émettra plus de valeurs
- Peut également être **complété** => détruit et n'émettra plus rien
- RxJS library : <https://angular.io/guide/rx-library>
 - ▶ <https://rxjs.dev/guide/observable>

Les observables

- Déclaration : on ajoute \$ au nom de la variable et on indique le type entre chevrons <>
 - ▶ nom.observable\$: Observable<type.observable>
- subscribe() : permet de "souscrire" à un Observable dans le typescript
- unsubscribe() : ne pas oublier "d'annuler la souscription" quand on n'a plus besoin de l'Observable
 - ▶ Sinon, risque de fuite mémoire
- async : permet de "souscrire" à un Observable dans le template

Les observables - opérateurs

- On manipule les Observables à l'aide d'**opérateurs**
- Deux types d'opérateurs principaux
 - Opérateurs de **bas niveau** : agissent sur les émissions de l'Observable
 - Opérateurs de **haut niveau** : agissent directement sur l'Observable

Les observables - opérateurs

- On manipule les Observables à l'aide d'opérateurs
- Deux types d'opérateurs principaux
 - Opérateurs de **bas niveau** : agissent sur les émissions de l'Observable
 - Opérateurs de **haut niveau** : agissent directement sur l'Observable
- pipe() : permet d'appliquer les opérateurs à un Observable

```
const modifiedObservable$ = originalObservable$.pipe(  
    firstOperator(),  
    secondOperator(arguments),  
    thirdOperator  
) ;
```

Les opérateurs bas niveau

- Les opérateurs sont appliqués dans l'ordre de passation à la fonction `pipe()`
- Exemples d'opérateurs :
 - `map()` : permet de transformer les émissions d'un Observable
 - `filter()` : permet de filtrer les émissions d'un Observable
 - `tap()` : permet d'ajouter des effets secondaires à un Observable
- Pour aller plus loin : <https://www.learnrxjs.io/>

Les opérateurs haut niveau

- Un Observable haut niveau est un Observable qui souscrit à d'autres Observables
- l'Observable qui souscrit est appelé l'Observable extérieur
- les Observables qui sont souscrits sont appelés les Observables intérieurs
- Les opérateurs haut niveau servent à gérer les situations où une nouvelle émission arrive de l'Observable extérieur alors que la souscription précédente à l'Observable intérieur n'a pas encore complété

Les opérateurs haut niveau

- Quatre opérateurs de haut niveau :
 - ▶ mergeMap : n'attend pas qu'un Observable intérieur complète pour souscrire au suivant (mise en parallèle)
 - ▶ concatMap : attend que l'Observable intérieur complète avant de souscrire au suivant (mise en série)
 - ▶ exhaustMap : ignore toute nouvelle émission de l'Observable extérieur tant qu'il y a une souscription active à un Observable intérieur.
 - ▶ switchMap : reçoit une nouvelle émission de l'Observable extérieur, s'il y a une souscription active à un Observable intérieur, il l'annule et souscrit au suivant.

Fuites mémoires

- Lorsqu'un Observable complète, il est détruit => pas de fuite mémoire
- Deux cas principaux :
 - ▶ On connaît le nombre d'émissions voulues de l'Observable
 - On utilise l'opérateur de bas niveau **take()**
 - Syntaxe : `take(nb_voulu)`
 - ▶ On a besoin des émissions de l'Observable pendant la durée de vie du component

Fuites mémoires

- Deux cas principaux :
 - ▶ On connaît le nombre d'émissions voulues de l'Observable
 - ▶ On a besoin des émissions de l'Observable pendant la durée de vie du component
 - Observable détruit en même temps que le component
 - On utilise `ngOnDestroy()` du lifecycle hooks du component (<https://angular.io/guide/lifecycle-hooks>)
- Les Observables "subscribe" avec le pipe `async` sont "unsubscribe" automatiquement par Angular lors de la destruction du component

Les formulaires (suite)

Reactive forms & Observables



Reactive forms & Observables

- Il est possible de coupler des formulaires et des Observables pour obtenir, par exemple, un aperçu du traitement du formulaire
- On lie un Observable aux changements de valeur du formulaire grâce à son attribut valueChanges
- On peut utiliser :
 - ▶ l'opérateur map() pour transformer les émissions du formulaire
 - ▶ Le pipe async et le mot clé as pour afficher l'aperçu dans le template
- On peut modifier le déclenchement de valueChanges pour émettre uniquement lorsque l'utilisateur change de champ, lors du blur des différents champs

Reactive forms & Observables

```
// new-surf-spot.component.ts
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
import { map, Observable } from "rxjs";
import { SurfSpotModel } from "../models/surf-spot.model";

@Component({selector: 'app-new-surf-spot'...})

export class NewSurfSpotComponent implements OnInit {

  surfSpotForm!: FormGroup;
  surfSpotPreview$!: Observable<SurfSpotModel>;

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit(): void {
    this.surfSpotForm = this.formBuilder.group( controls: {
      name: [null, [Validators.required]],
      location: [null, [Validators.required]],
      imageUrl: [null, [Validators.required]],
      description: [null],
    }, options: {
      updateOn: 'blur'
    });
    this.surfSpotPreview$ = this.surfSpotForm.valueChanges.pipe(
      map( project: formData => ({
        ...formData,
        createdDate: new Date(),
        id: 0,
        like: 0
      }))
    );
  }

  onSubmitForm() {
    console.log(this.surfSpotForm.value);
  }
}
```

```
ngOnInit(): void {
  this.surfSpotForm = this.formBuilder.group( controls: {
    name: [null, [Validators.required]],
    location: [null, [Validators.required]],
    imageUrl: [null, [Validators.required]],
    description: [null],
  }, options: {
    updateOn: 'blur'
  });
  this.surfSpotPreview$ = this.surfSpotForm.valueChanges.pipe(
    map( project: formData => ({
      ...formData,
      createdDate: new Date(),
      id: 0,
      like: 0
    }))
  );
}
```

Reactive forms & Observables

```
// new-surf-spot.component.ts
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
import { map, Observable } from "rxjs";
import { SurfSpotModel } from "../models/surf-spot.model";

@Component({selector: 'app-new-surf-spot'...})

export class NewSurfSpotComponent implements OnInit {

  surfSpotForm!: FormGroup;
  surfSpotPreview$!: Observable<SurfSpotModel>;

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit(): void {
    this.surfSpotForm = this.formBuilder.group({
      controls: {
        name: [null, [Validators.required]],
        location: [null, [Validators.required]],
        imageUrl: [null, [Validators.required]],
        description: [null],
      },
      options: {
        updateOn: 'blur'
      }
    });
    this.surfSpotPreview$ = this.surfSpotForm.valueChanges.pipe(
      map((project: FormGroup) => {
        ...project,
        createdDate: new Date(),
        id: 0,
        like: 0
      })
    );
  }

  onSubmitForm() {
    console.log(this.surfSpotForm.value);
  }
}
```

```
<!-- new-surf-spot.component.html -->


<h2>NOUVEAU SPOT</h2>
  <form ...>
</div>


<h2>{{ surfSpot.name | titlecase }}</h2>
  <p>Pays : {{ surfSpot.location }}</p>
  <img [src]="surfSpot.imageUrl" [alt]="surfSpot.name" />
  <p>{{ surfSpot.description }}</p>
  <p>Dernière session le {{surfSpot.createdDate | date: 'd MMMM yyyy, à HH:mm'}}</p>
</div>


```