



Formation Angular

Romain Bohdanowicz

Twitter : @bioub

<http://formation.tech/>



Introduction



- Romain Bohdanowicz

Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle

- Expérience

Formateur/Développeur Freelance depuis 2006

Plus de 8000 heures de formation animées

- Langages

Expert : HTML / CSS / JavaScript / PHP / Java

Notions : C / C++ / Objective-C / C# / Python / Bash / Batch

- Certifications

PHP 5 / PHP 5.3 / PHP 5.5 / Zend Framework 1

- Particularités

Premier site web à 12 ans (HTML/JS/PHP), Triathlète à mes heures perdues

- Et vous ?

Langages ? Expérience ? Utilité de cette formation ?



TypeScript



- TypeScript : JavaScript + Typage statique
 - TypeScript est un langage créé par Microsoft, construit comme un sur-ensemble d'ECMAScript
 - Pour pouvoir exécuter le code il faut le transformer en JavaScript avec un compilateur
 - A quelques exceptions près et selon la configuration, le JavaScript est valide en TypeScript
 - Le principal intérêt de TypeScript est l'ajout d'un typage statique

TypeScript - Installation



- Installation
 - `npm install -g typescript`
- Création d'un fichier de configuration
 - `tsc --init`
- Compilation
 - `tsc`

TypeScript - Typage statique



- Le principal intérêt de TypeScript est l'introduction d'un typage statique

```
const firstName: string = 'Romain';  
  
function hello(firstName: string): string {  
    return `Hello ${firstName}`;  
}
```

- Types basiques :

- *boolean*
- *number*
- *string*

```
const lastName: string = 'Bohdanowicz';  
const age: number = 32;  
const isTrainer: boolean = true;
```

TypeScript - Typage statique

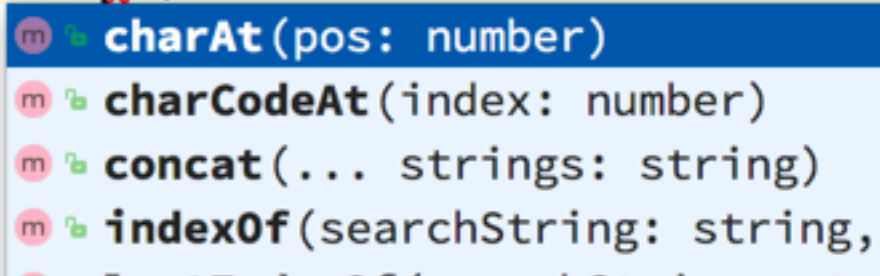


▸ Avantages

- Complétion

```
const firstName: string = 'Romain';

function hello(firstName: string): string {
  return `Hello ${firstName.}`;
}
```




- Détection des erreurs

```
const firstName: string = 'Romain';

function hello(firstName: string): string {
  return `Hello ${firstName}`;
}

hello({
  firstName: 'Romain',
});
```



TypeScript - Typage statique



▸ Tableaux

```
const firstNames: string[] = ['Romain', 'Edouard'];  
const colors: Array<string> = ['blue', 'white', 'red'];
```

▸ Tuples

```
const email: [string, boolean] = ['romain.bohdanowicz@gmail.com', true];
```

▸ Enum

```
enum Choice {Yes, No, Maybe}  
  
const c1: Choice = Choice.Yes;  
const choiceName: string = Choice[1];
```

▸ Never

```
function error(message: string): never {  
    throw new Error(message);  
}
```

TypeScript - Typage statique



▸ Any

```
let anyType: any = 12;  
anyType = "now a string string";  
anyType = false;  
anyType = {  
  firstName: 'Romain'  
};
```

▸ Void

```
function withoutReturn(): void {  
  console.log('Do something')  
}
```

▸ Null et undefined

```
let u: undefined = undefined;  
let n: null = null;
```

TypeScript - Assertion de type



- Le compilateur ne peut pas toujours déterminer le type adéquat :

```
const formElt = document.querySelector('form.myForm');  
const url = formElt.action; // error TS2339: Property 'action' does not exist on  
type 'Element'.
```

- Il faut alors lui préciser, 3 syntaxes possibles

```
let formElt = <HTMLFormElement> document.querySelector('form.myForm');  
const url = formElt.action;
```

```
let formElt = document.querySelector<HTMLFormElement>('form.myForm');  
const url = formElt.action;
```

```
let formElt = document.querySelector('form.myForm') as HTMLFormElement;  
const url = formElt.action;
```

TypeScript - Inférence de type



- TypeScript peut parfois déterminer automatiquement le type :

```
const title = 'First Names';  
console.log(title.toUpperCase());  
  
const names = ['Romain', 'Edouard'];  
for (let n of names) {  
    console.log(n.toUpperCase());  
}
```



- Pour documenter un objet on utilise une interface
 - Anonyme

```
function helloInterface(contact: {firstName: string}) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```

- Nommée

```
interface ContactInterface {  
    firstName: string;  
}  
  
function helloNamedInterface(contact: ContactInterface) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```



- Les propriétés peuvent être :
 - optionnelles (ici *lastName*)
 - en lecture seule, après l'initialisation (ici *age*)
 - non déclarées (avec les crochets)

```
interface ContactInterface {  
    firstName: string;  
    lastName?: string;  
    readonly age: number;  
    [propName: string]: any;  
}  
  
function helloNamedInterface(contact: ContactInterface) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```



- Quelques différences avec JavaScript sur le mot clé class
 - On doit déclarer les propriétés
 - On peut définir une visibilité pour chaque membre : *public*, *private*, *protected*

```
class Contact {  
    private firstName: string;  
  
    constructor(firstName: string) {  
this.firstName = firstName;  
    }  
  
    hello(): string {  
return `Hello my name is ${this.firstName}`;  
    }  
}  
  
const romain = new Contact('Romain');  
console.log(romain.hello()); // Hello my name is Romain
```



- Une classe peut
 - Hériter d'une autre classe (comme en JS)
 - Implémenter une interface
 - Être utilisée comme type

```
interface Writable {  
    write(data: string): void;  
}  
  
class FileLogger implements Writable {  
    write(data: string): Writable {  
        console.log(`Write ${data}`);  
        return this;  
    }  
}
```




- Permet de paramétrer le type de certaines méthodes

```
class Stack<T> {  
    private data: Array<T> = [];  
    push(val: T) {  
        this.data.push(val);  
    }  
    pop(): T {  
        return this.data.pop();  
    }  
    peek(): T {  
        return this.data[this.data.length - 1];  
    }  
}  
  
const strStack = new Stack<string>();  
strStack.push('html');  
strStack.push('body');  
strStack.push('h1');  
console.log(strStack.peek().toUpperCase()); // H1  
console.log(strStack.pop().toUpperCase()); // H1  
console.log(strStack.peek().toUpperCase()); // BODY
```



Angular CLI



- Angular introduit un programme en ligne de commande permettant d'interagir avec l'application :
 - créer un projet
 - builder
 - lancer le serveur de dev
 - générer du code
 - générer les fichiers de langue
 - ...

Angular CLI - Introduction



- Installation

- `npm install -g @angular/cli`

- Documentation

- <https://cli.angular.io/>

- <https://github.com/angular/angular-cli/wiki>

- `ng help`

- `ng help COMMANDE`



- Complet (avec tests et conventions)
 - `ng new CHEMIN_VERS_MON_PROJET`
- Minimal
 - `ng new --minimal CHEMIN_VERS_MON_PROJET`
- Autres options
 - `--skip-commit` : ne fait pas de commit initial
 - `--routing` : créer un module pour les routes (Single Page Application)
 - `--prefix` : change le préfixe des composant (par défaut *app*)
 - `--style` : change type de fichier CSS (css par défaut ou *sass*, *scss*, *less*, *stylus*)
 - `--service-worker` : ajouter un service worker pour le mode hors-ligne

Angular CLI - Squelette minimal



- **.angular-cli.json**
Fichier de configuration du programme *ng*, permet de renommer des répertoires, des fichiers
- **tsconfig.json — src/tsconfig.app.json**
Configuration du compilateur TypeScript
- **src/app**
Le code source de l'application
- **src/assets**
Les fichiers statiques non-buildés (images...)
- **src/environments**
Configuration de l'application
- **src/index.html — src/main.ts**
Points d'entrées de l'application
- **src/polyfills.ts**
Chargement des polyfills (core-js, ...)
- **src/style.css**
CSS global
- **src/typings.d.ts**
Documentation TypeScript des sources JavaScript (interfaces...)

```
— .angular-cli.json
— .git
— .gitignore
— node_modules
— package-lock.json
— package.json
— src
  — app
    — app.component.ts
    — app.module.ts
  — assets
    — .gitkeep
  — environments
    — environment.prod.ts
    — environment.ts
  — index.html
  — main.ts
  — polyfills.ts
  — styles.css
  — tsconfig.app.json
  — typings.d.ts
— tsconfig.json
```



- Compiler l'application Angular
 - `ng build`
- Options intéressantes :
 - `--prod` : minifie le code avec UglifyJS et active les options `--aot`, `--environment=prod`, `--extract-css`, `--build-optimizer`...
 - `--environment=NOM` : permet de charger un fichier de configuration particulier (staging, test...)
 - `--vendor-chunk` : pour que le code de `node_modules` soit dans un fichier séparé



► Gains d'un build avec *--prod*

```
Angular 4 : ng build
Date: 2017-11-02T09:02:41.042Z
Hash: 1d2842c3e0ac46a944f0
Time: 6349ms
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB [entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 18.1 kB {vendor} [initial] [rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 199 kB {inline} [initial] [rendered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 11.3 kB {inline} [initial] [rendered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 1.98 MB [initial] [rendered]
```

```
Angular 5 : ng build
Date: 2017-11-02T09:07:47.401Z
Hash: d1a929eaad03e8e746bb
Time: 4937ms
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB [entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 17.9 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 199 kB [initial] [rendered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 11.3 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.29 MB [initial] [rendered]
```

```
Angular 4 : ng build --prod
Date: 2017-11-02T09:03:29.639Z
Hash: cb067f695303856c2315
Time: 6228ms
chunk {0} polyfills.14173651b8ae6311a4b5.bundle.js (polyfills) 61.4 kB {4} [initial] [rendered]
chunk {1} main.f5677287cea9969f6fb6.bundle.js (main) 8.39 kB {3} [initial] [rendered]
chunk {2} styles.d41d8cd98f00b204e980.bundle.css (styles) 0 bytes {4} [initial] [rendered]
chunk {3} vendor.43700a281455e3959c70.bundle.js (vendor) 217 kB [initial] [rendered]
chunk {4} inline.6b5a62abf05dccccf24d7.bundle.js (inline) 1.45 kB [entry] [rendered]
```

```
Angular 5 : ng build --prod --vendor-chunk
Date: 2017-11-02T09:10:58.444Z
Hash: cf4dd52226e15e33c748
Time: 11487ms
chunk {0} polyfills.ad37cd45a71cb38eee76.bundle.js (polyfills) 61.1 kB [initial] [rendered]
chunk {1} main.2c7fbf970f7125d9617e.bundle.js (main) 7.03 kB [initial] [rendered]
chunk {2} styles.d41d8cd98f00b204e980.bundle.css (styles) 0 bytes [initial] [rendered]
chunk {3} vendor.719fe92af8c44a7e3dac.bundle.js (vendor) 167 kB [initial] [rendered]
chunk {4} inline.220ce59355d1cb2bcb28.bundle.js (inline) 1.45 kB [entry] [rendered]
```


Angular CLI - Serveur de développement



- Lancer le serveur de dev
 - `ng serve`
- Options intéressantes :
 - `--port` : changer le port
 - `--target=production` : sert les fichiers dans la config de prod



- Générateurs

Angular CLI contient un certain nombre de générateurs : application, class, component, directive, enum, guard, interface, module, pipe, service, universal, appShell

- Dry run

Chaque générateur peut se lancer avec l'option *--dry-run* ou *-d* qui va afficher le résultat de la commande sans rien créer, sachant qu'il n'y a pas de retour automatique possible une fois les fichiers créés.

- Afficher la doc d'un générateur

- `ng help generate NOM_DU_GENERATEUR`



- Générer un module
 - `ng generate module CHEMIN_DEPUIS_APP`
 - `ng g m CHEMIN_DEPUIS_APP`
- Autres options
 - `--routing`: génère un 2e module pour les routes
 - `--flat`: ne crée pas de répertoire



- Générer un composant
 - `ng generate component CHEMIN_DEPUIS_APP`
 - `ng g c CHEMIN_DEPUIS_APP`
- Autres options
 - `--flat` : ne crée pas de répertoire
 - `--export` : ajoute une entrée dans les exports du module



- Lancer les tests Karma + Jasmine
 - `ng test`
- Lancer les tests Protractor
 - `ng e2e`
- Vérifier les conventions de code
 - `ng lint`
 - `ng lint --fix --type-check`



Composants

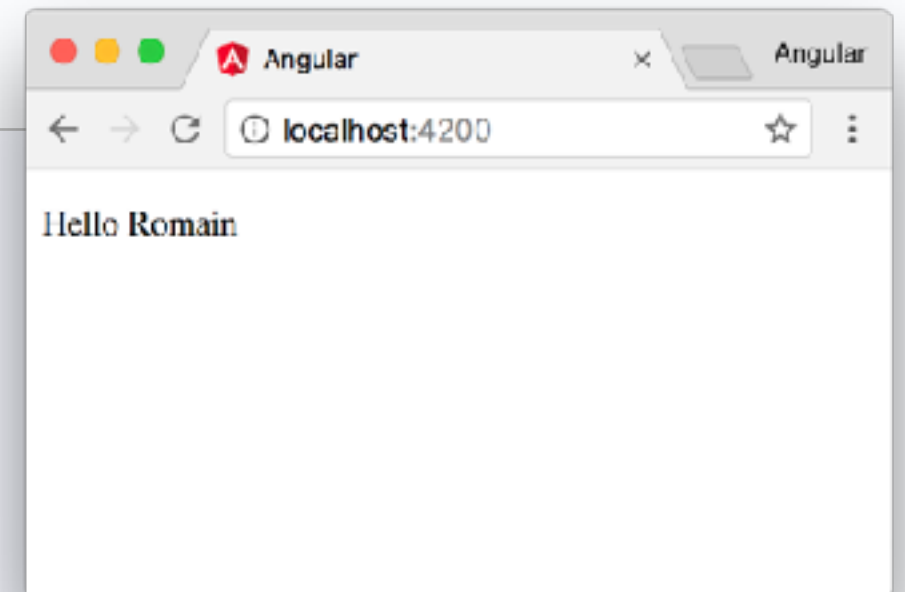
Composants - Introduction



- 2 parties
 - code TypeScript
 - template
- Compilation
 - Les 2 sont compilés dans un code optimisé pour la VM JavaScript
 - Le template peut être compilé en JIT (par le browser) ou en AOT (au moment du build)

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

@Component({
  selector: 'my-hello',
  template: '<p>Hello {{name}}</p>',
})
export class HelloComponent {
  public name = 'Romain';
}
```



Composants - Lifecycle Hooks



Composants - Lifecycle Hooks



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

@Component({
  selector: 'hello-lifecycle',
  template: `
    {{ now | date:'HH:mm:ss' }}
  `
})
export class LifecycleComponent implements OnInit, OnDestroy {

  public now = new Date();
  private intervalId: number;

  ngOnInit() {
    this.intervalId = setInterval(() => {
      this.now = new Date();
    }, 1000)
  }

  ngOnDestroy() {
    clearInterval(this.intervalId);
  }
}
```



Templates



▸ Templates

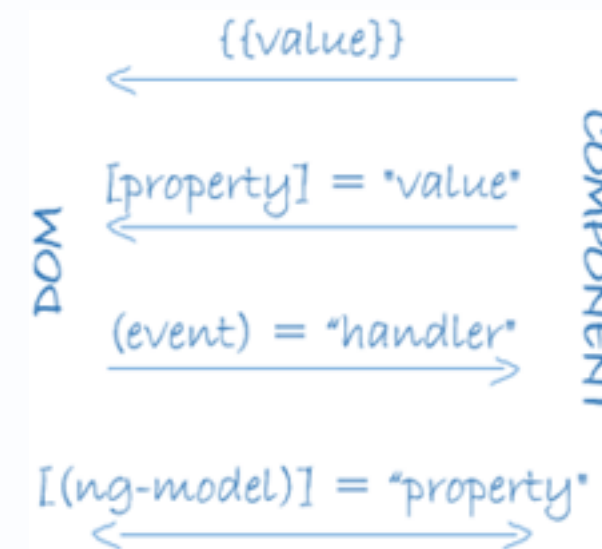
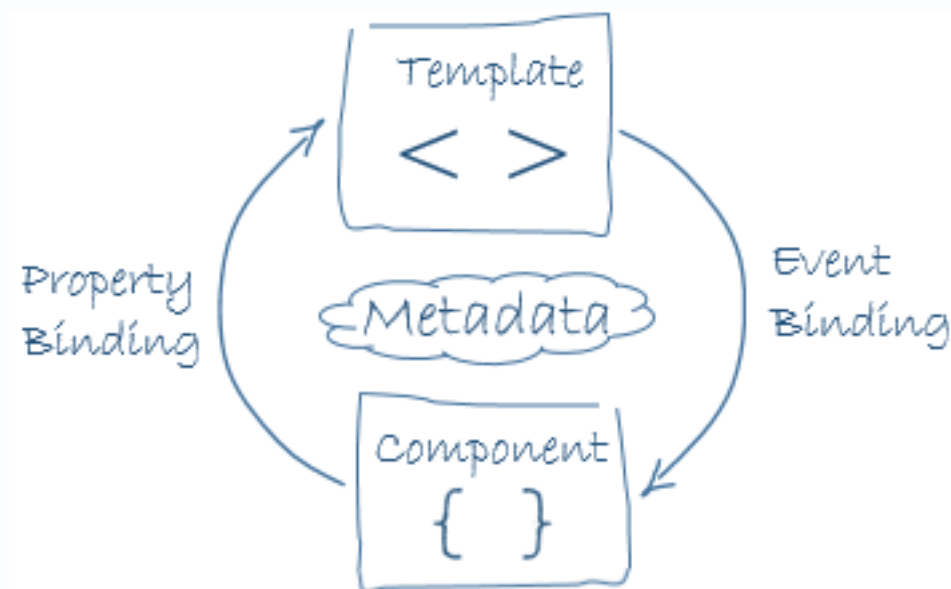
- Comme dans AngularJS, on décrit l'interface de manière déclarative dans des templates
- Chaque template est compilé par le compilateur d'Angular, soit en amont (mode AOT pour Ahead Of Time Compilation), soit dans le browser (mode JIT pour Just In Time Compilation)
- Les templates sont ainsi transformé en du code optimisé pour la VM/Moteur JavaScript

Templates - Data binding



► Data binding

- Sans data binding ce serait au développeur de maintenir les changements à opérer sur le DOM à chaque événement
- Dans jQuery par exemple, cliquer sur un bouton peut avoir pour conséquence de rafraîchir une balise, de lancer un indicateur de chargement...
- Avec Angular le développeur décrit l'état du DOM en fonction de propriétés qui constitue le Modèle, ainsi un événement n'a plus qu'



Templates - Property Binding



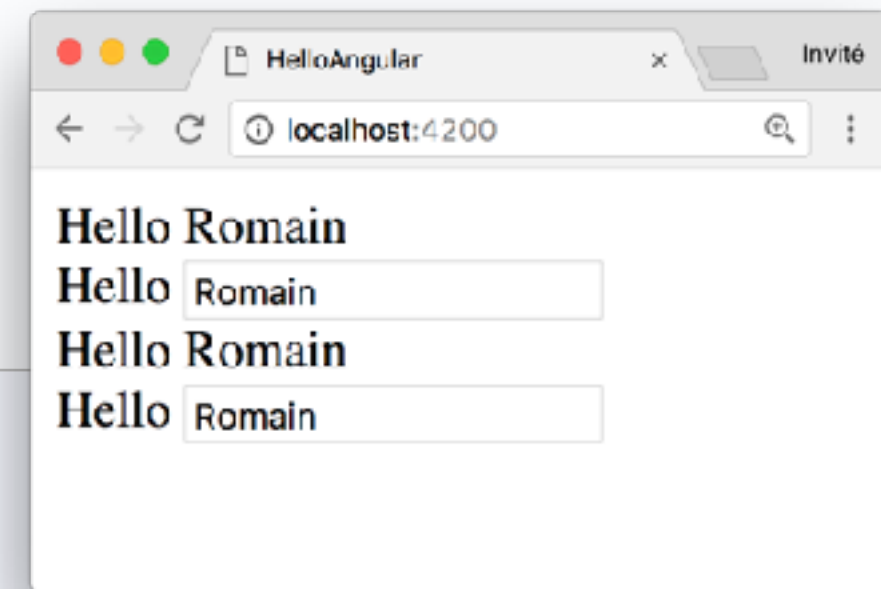
▸ 2 syntaxes

Pour synchroniser le DOM avec le modèle (les propriétés publiques du composant dans Angular)

- `bind-nomDeLaPropDuDOM="propDuComposant"`
- `[nomDeLaPropDuDOM]="propDuComposant"`

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

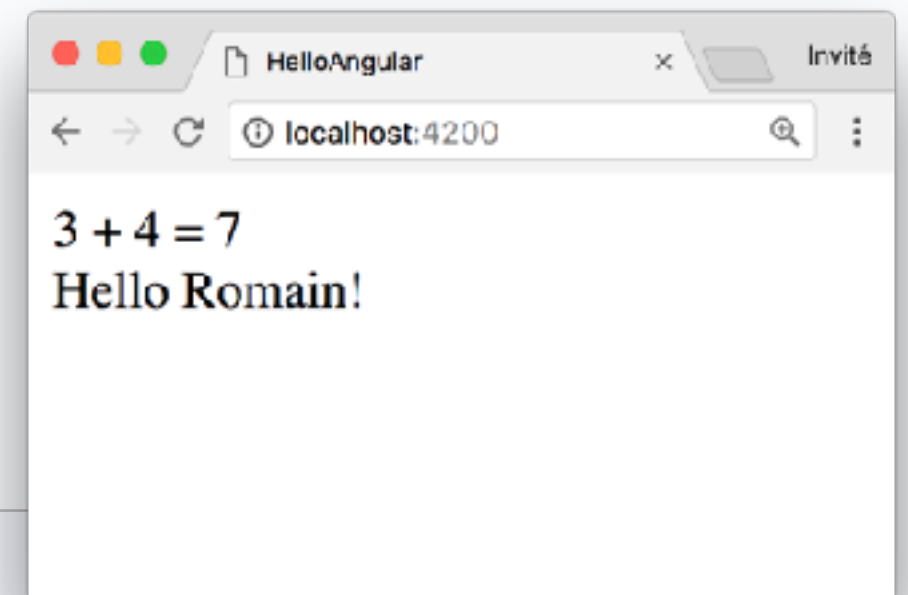
@Component({
  selector: 'hello-property-binding',
  template: `
    <div>Hello <span bind-textContent="prenom"></span></div>
    <div>Hello <input bind-value="prenom"></div>
    <div>Hello <span [textContent]="prenom"></span></div>
    <div>Hello <input [value]="prenom"></div>
  `
})
export class PropertyBindingComponent {
  public prenom = 'Romain';
}
```



Templates - Expressions



- Dans un property binding il est possible d'utiliser des noms de propriétés ou des expressions, sauf les expressions ayant des effets de bords :
 - affectations (`=`, `+=`, `-=`, ...)
 - `new`
 - expressions chaînées avec `;` ou `,`
 - incrementation et décrémentation (`++` et `--`)



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

@Component({
  selector: 'hello-prenom',
  template: `
    <div>3 + 4 = <span [textContent]="3 + 4"></span></div>
    <div>Hello <span [textContent]="prenom + '!' "></span></div>
  `,
})
export class PrenomComponent {
  public prenom = 'Romain';
}
```

Templates - Interpolation

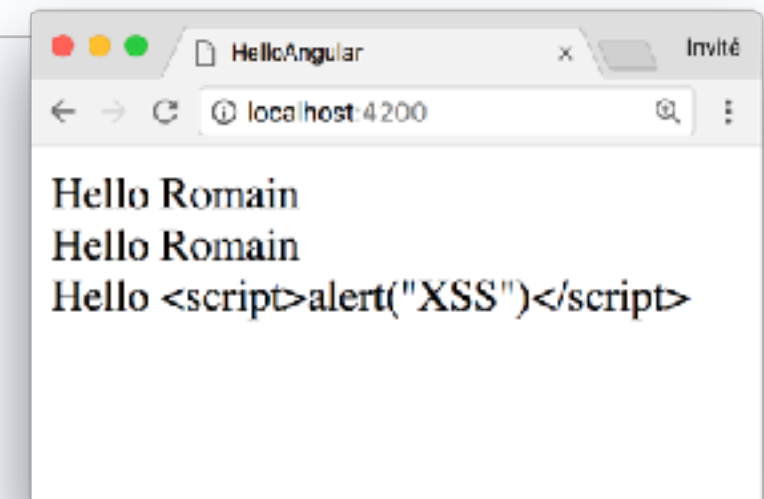


▸ Interpolation

- Plutôt que bind-innerHTML sur une balise span, on peut utiliser la syntaxe aux doubles accolades {{ }}
- A privilégier car cette syntaxe échappe les entrées, évitant ainsi que des balises contenues dans les entrées se retrouvent dans le DOM (faille XSS)

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

@Component({
  selector: 'hello-interpolation',
  template: `
<div>Hello <span [innerHTML]="prenom"></span></div>
<div>Hello {{prenom}}</div>
<div>Hello {{xssAttack}}</div>
`
})
export class InterpolationComponent {
  public prenom = 'Romain';
  public xssAttack = '<script>alert("XSS")</script>';
}
```



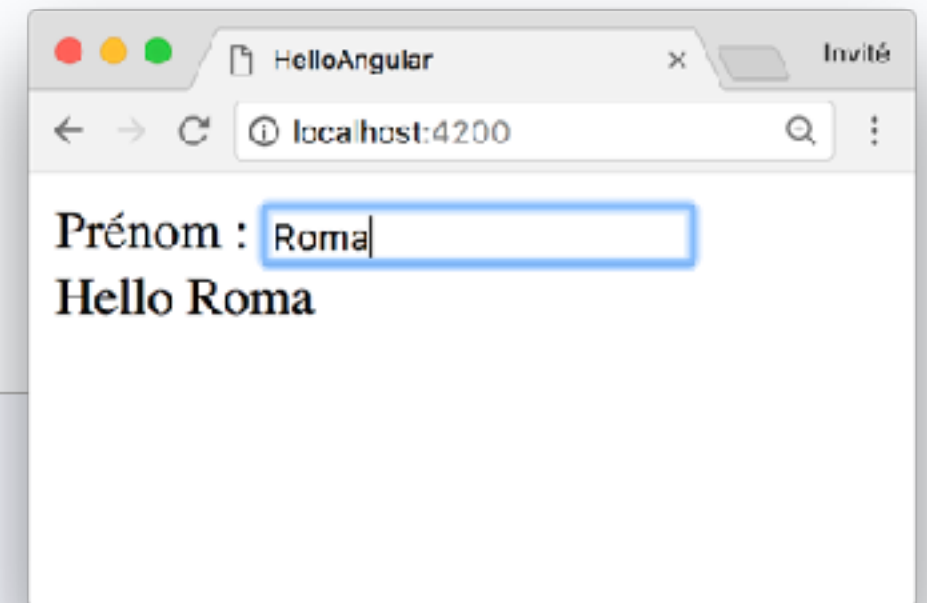
Templates - Event Binding



▸ 2 syntaxes

Pour synchroniser le DOM avec le modèle (les propriétés publiques du composant dans Angular), on utilise des événements

- `on-nomEvent="methodeDuComposant()"`
- `(nomEvent)="methodeDuComposant()"`



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

@Component({
  selector: 'hello-event-binding',
  template: `
    <div>Prénom : <input on-input="updatePrenom($event)"></div>
    <div>Prénom : <input (input)="updatePrenom($event)"></div>
    <div>Prénom : <input (input)="prenom = $event.target.value"></div>
    <div>Hello {{prenom}}</div>
  `,
})
export class EventBindingComponent {
  public prenom = '';

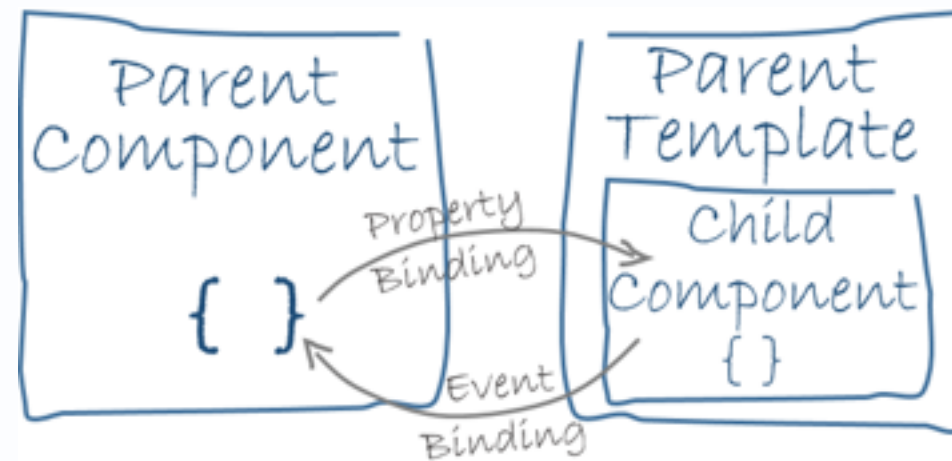
  public updatePrenom(e) {
    this.prenom = e.target.value;
  }
}
```




- Data-binding dans les 2 sens
- TODO
- `[()]` = BANANA IN A BOX



- Communication inter-composant



todo-container

['Pain', 'Lait', 'Beurre']

todo-form

Beurre

Add

todo-list

todo-details

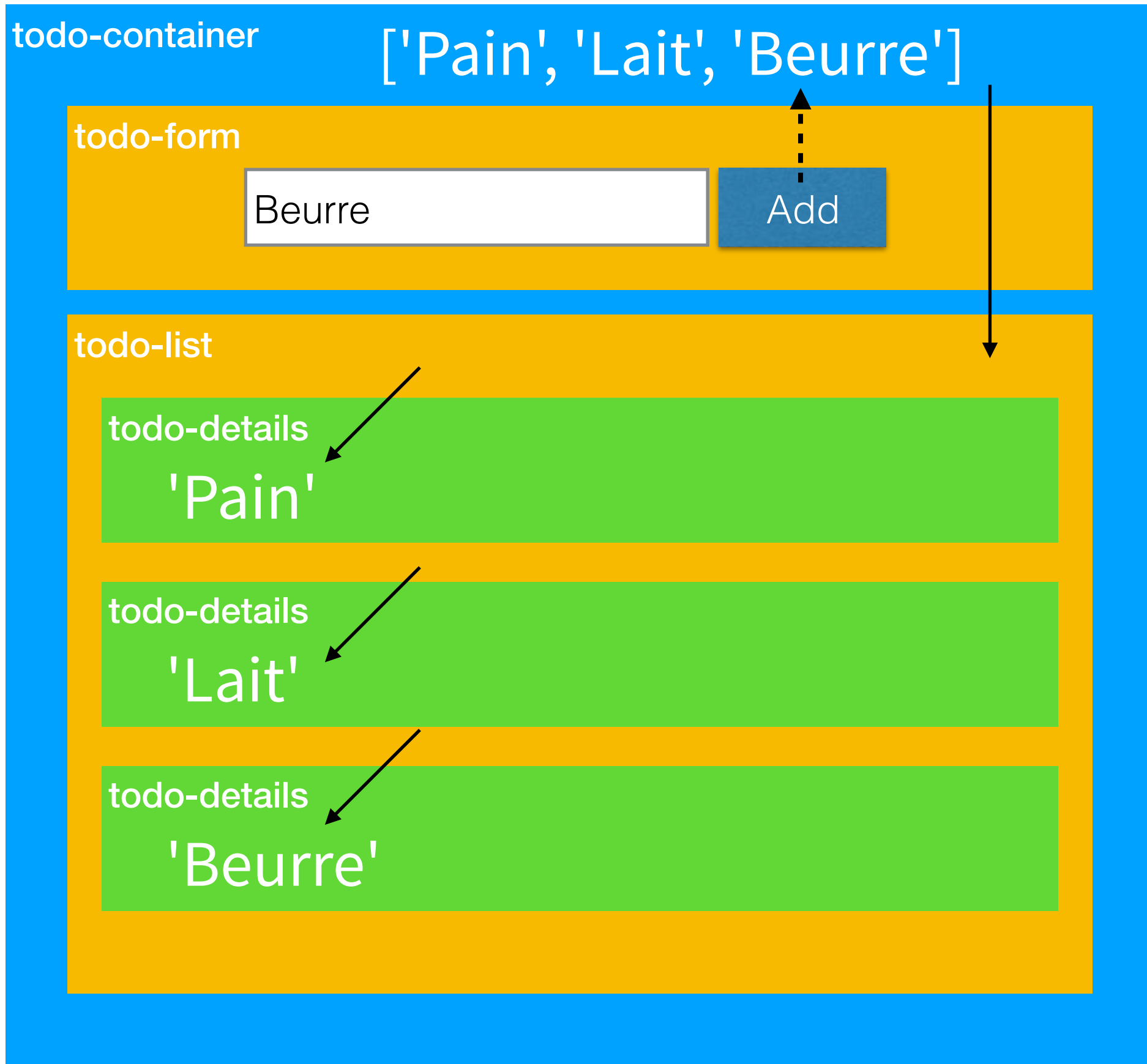
'Pain'

todo-details

'Lait'

todo-details

'Beurre'





Modules



- 2 notions de modules
 - NgModule (class décorée avec @NgModule)
 - Module ES6 (import / export de fichiers)
- Jusqu'à la RC d'Angular 2, la notion de NgModule n'existait pas
- Intérêt d'avoir des NgModules :
 - Pouvoir importer un ensemble de composants / directives / pipes...
 - Pour configurer la portée d'un service
 - Permettre de charger des blocs de code par lazy-loading (après le chargement initial)
 - ...



▸ Principaux Modules

- `AppModule` : le module racine
- `CommonModule` : le module qui inclut toutes les directives Angular de base comme *NgIf*, *NgForOf*, ...
- `BrowserModule` : exporte *CommonModule* et contient les services permettant le rendu DOM, la gestion des erreurs, la modification des balises *title* ou *meta*...
- `FormsModule` : le module qui permet la validation des formulaires, la déclaration de la directive *ngModel*...
- `HttpClientModule` : contient les composants pour les requêtes HTTP
- `RouterModule` : permet de manipuler des routes (associer des composants à des URL)

▸ Open-Source

La première chose à faire après l'installation d'une bibliothèque *Angular* via *npm* sera d'importer un module



▸ Déclaration

- Pour qu'un composant, directive ou pipe existe dans l'application il faut le déclarer dans un module
- Ne jamais déclarer 2 fois la même classe dans 2 modules différents

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { HelloComponent } from './hello/hello.component';

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



► Erreur courante

Si un composant, directive ou pipe n'est pas déclaré dans un module, ou bien que le module dans lequel il est déclaré n'est pas importé par le module qui l'utilise

```
Uncaught Error: Template parse errors:  
'app-title' is not a known element:  
1. If 'app-title' is an Angular component, then verify that it is part of this  
NgModule.  
2. If 'app-title' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the  
'@NgModule.schemas' of this component to suppress this message.
```




- Bonnes pratiques
 - Créer un module CoreModule global
 - Créer un module SharedModule global

Best Practices - Linters



Best Practices - Style Guide



Best Practices - Build

