Angular 2/4/5

Gilbert NZEKA
pro@gilbertnzeka.com
https://www.linkedin.com/in/nzeka

Qui suis-je ?

Gilbert NZEKA

- Formateur
  - En entreprise
  - Université
  - Ecoles supérieures
- Entrepreneur
  - Solution de paiement
  - Billetterie en ligne
  - Daily deals dans le sport
  - Legaltech
- Auteur et co-auteur

Et vous?

Vos attentes de la formation

Programme

- Introduction – Découvrez l'environnement et les principes d'Angular
- ES2015 (ES6) et TypeScript – Appréhendez le futur du Web
- Gérez les Composants et directives efficacement
- Pipes – Utilisez les transformateurs fournis ou créez vos propres pipes
- Formulaires – Créez et validez des formulaires avec Angular

Programme

- Services et injection de dépendances – Maitrisez les bonnes pratiques
- Asynchronicité – Formez-vous à la programmation réactive avec Angular
- Routage – Maitrisez la navigation sous Angular
- Serveurs et communication HTTP – Envoyez et recevez des données par HTTP
- Allez plus loin…

Introduction – Découvrez l'environnement et les principes d'Angular

Les technologies de développement Web

Qu’est-ce qu’un framework?

Qu'est-ce qu'un framework?

Un **framework** désigne en programmation informatique un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application. C'est le **framework**, ou structure logicielle en français, qui établit les fondations d'un logiciel ou son squelette.

Il existe aujourd'hui différents types de framework. Il y a eux dits d'infrastructure système, permettent le développement des systèmes d'exploitation et des interfaces graphiques. Il y en a également qui sont spécifiques aux applications développées par des entreprises pour un usage interne ou sur le Web. Certains, appellent également framework les solutions de gestion de contenu, qui ont pour mission de créer, collecter, classer, stocker et publier des biens numérisés.

Qu'en est-il dans le Web?

Au commencement, les sites Internet étaient simples ! Votre navigateur envoyait une demande à un serveur et ce dernier répondait avec de l'HTML. Le navigateur transformait cette réponse en ce que vous voyez dans le navigateur.

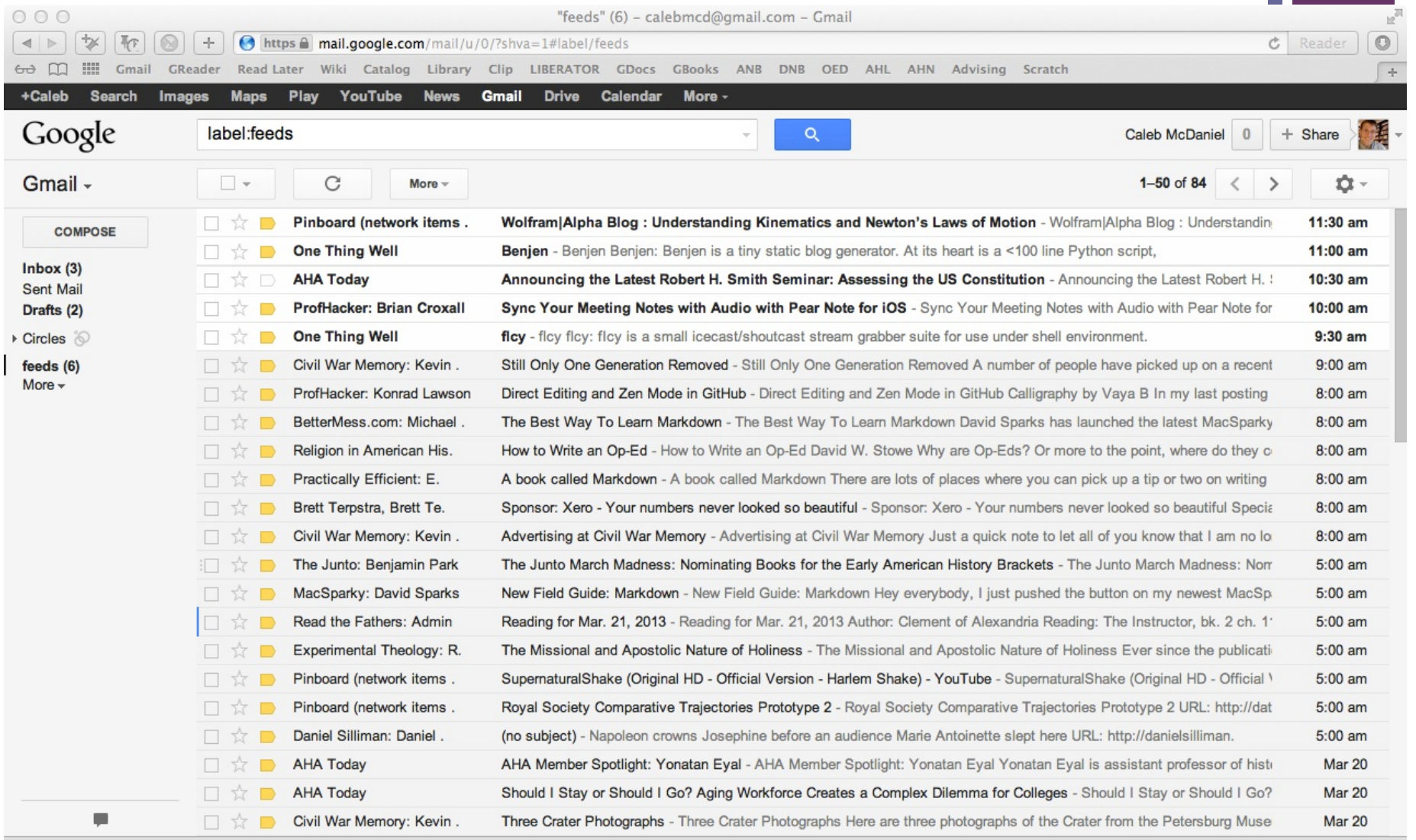
Qu'en est-il dans le Web?

Aujourd'hui, c'est beaucoup plus compliqué ! Votre navigateur commence toujours par envoyer une demande, le serveur commence toujours par envoyer une réponse. Une fois que cette réponse est revenue à votre navigateur, le véritable travail commence : Ajax et contenus asynchrones, CSS, JS, etc.

Qu'en est-il dans le Web?

Bootstrap et Angular sont tous les 2 des **front-end frameworks**. Bootstrap vous aide à mettre en page votre site visuellement. Angular vous aide à gérer l'intéractivité nécessaires des applications web professionnelles (cf. Gmail et l'url routing des emails).

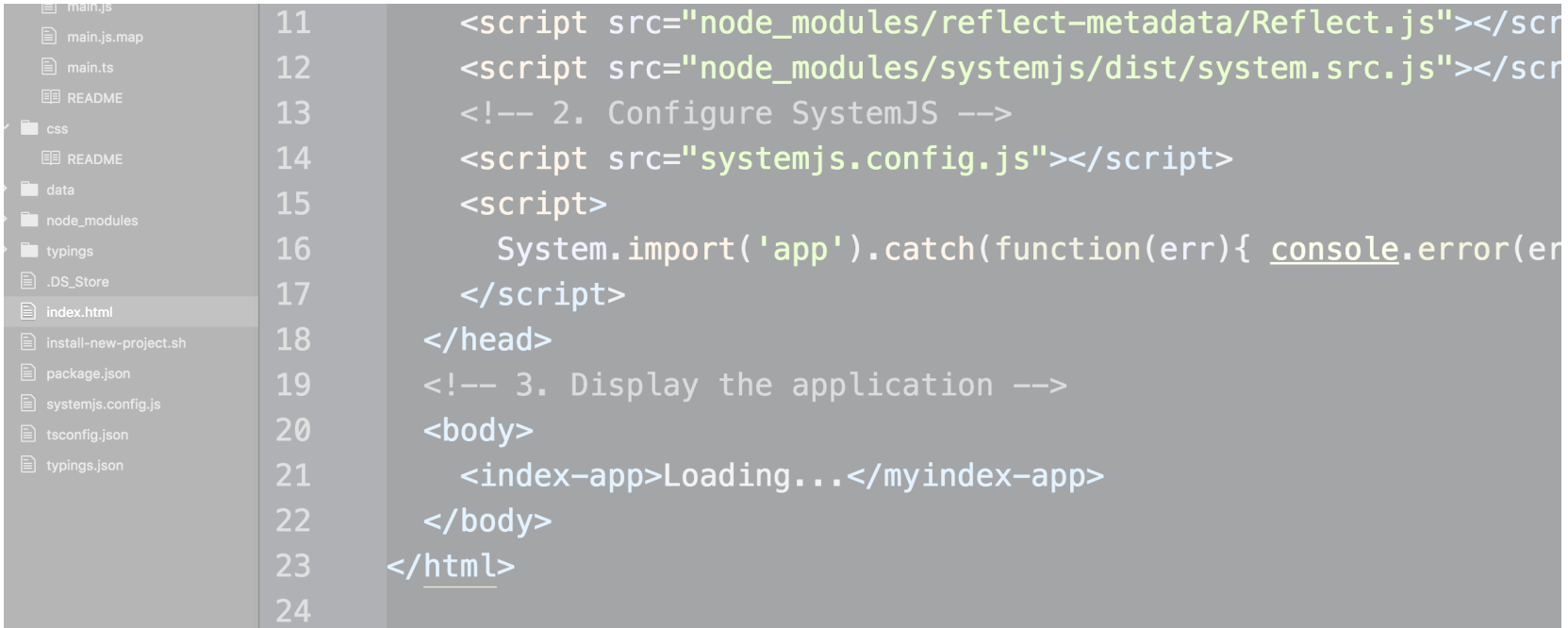
La message Gmail : un exemple concret de ***Single-Page Application***



Qu'est-ce que Angular?

Angular est le framework JavaScript libre et open-source développé par Google.

Angular est fondé sur l'extension du langage HTML par de nouvelles balises (tags) et attributs pour aboutir à une définition déclarative des pages web.



Qu’est-ce que Angular?

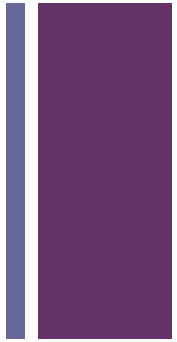
Le code HTML étendu représente alors la partie « vue » du patron d'architecture MVC (modèle-vue-contrôleur) auquel Angular correspond, via des components permettant de prototyper des actions en code JavaScript natif.

Angular utilise une boucle de dirty-checking (qui consiste à surveiller et à détecter des modifications sur un objet JavaScript) pour réaliser un data-binding bidirectionnel permettant la synchronisation automatique des modèles et des vues.

Angular embarque un sous-ensemble de la bibliothèque open source jQuery appelé jQLite, mais peut aussi utiliser jQuery si elle est chargée.

Installation de notre environnement Angular

Installation de notre environnement



sudo npm install --unsafe-perm --verbose -g @angular/cli

Installation de notre environnement

ng -v

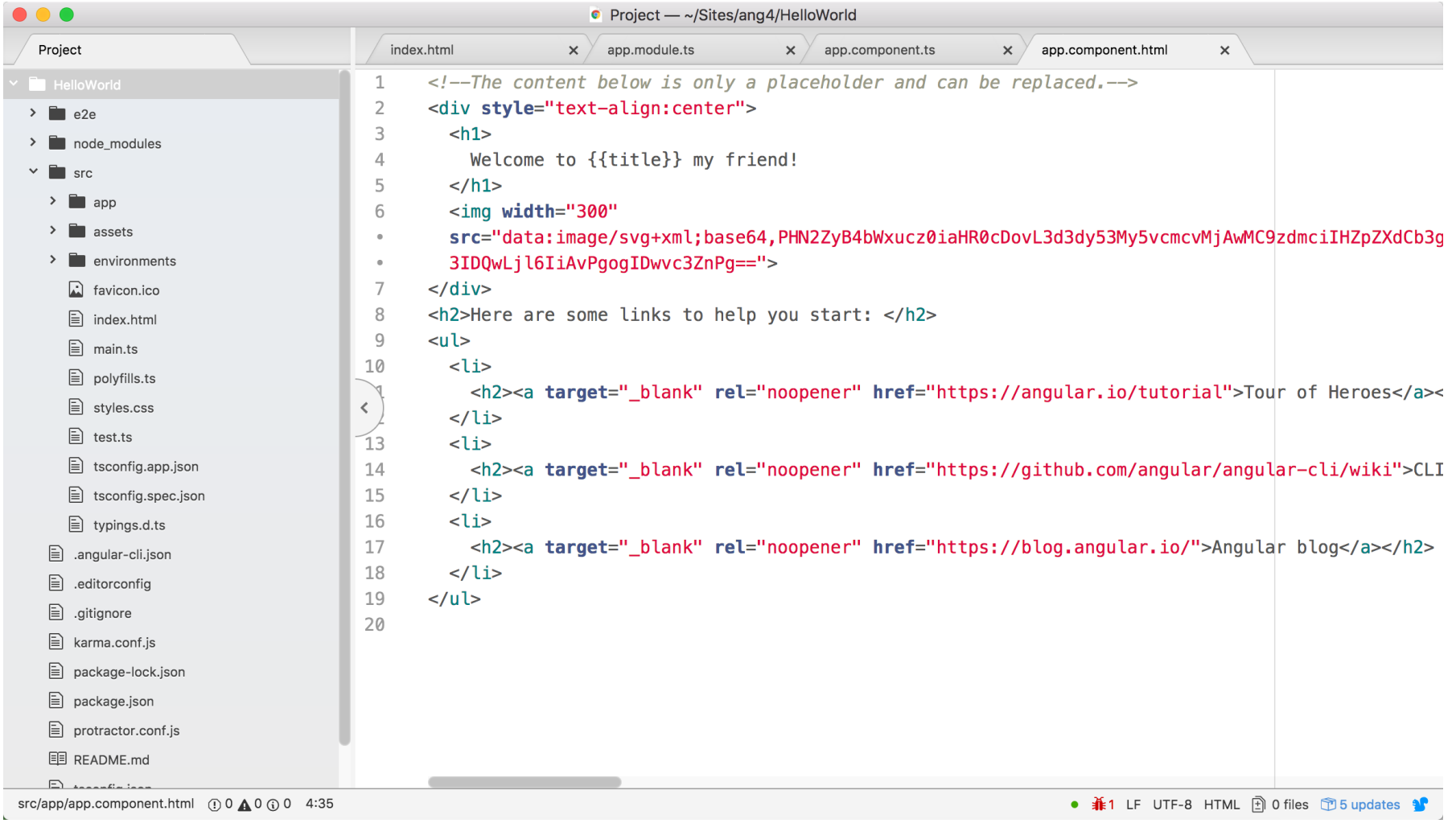
Installation de notre environnement

ng new HelloWorld

Installation de notre environnement

ng generate component [NAME]

Installation de notre environnement



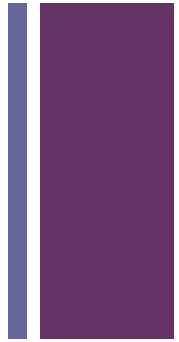
Installation de notre environnement

ng -v

Installation de notre environnement

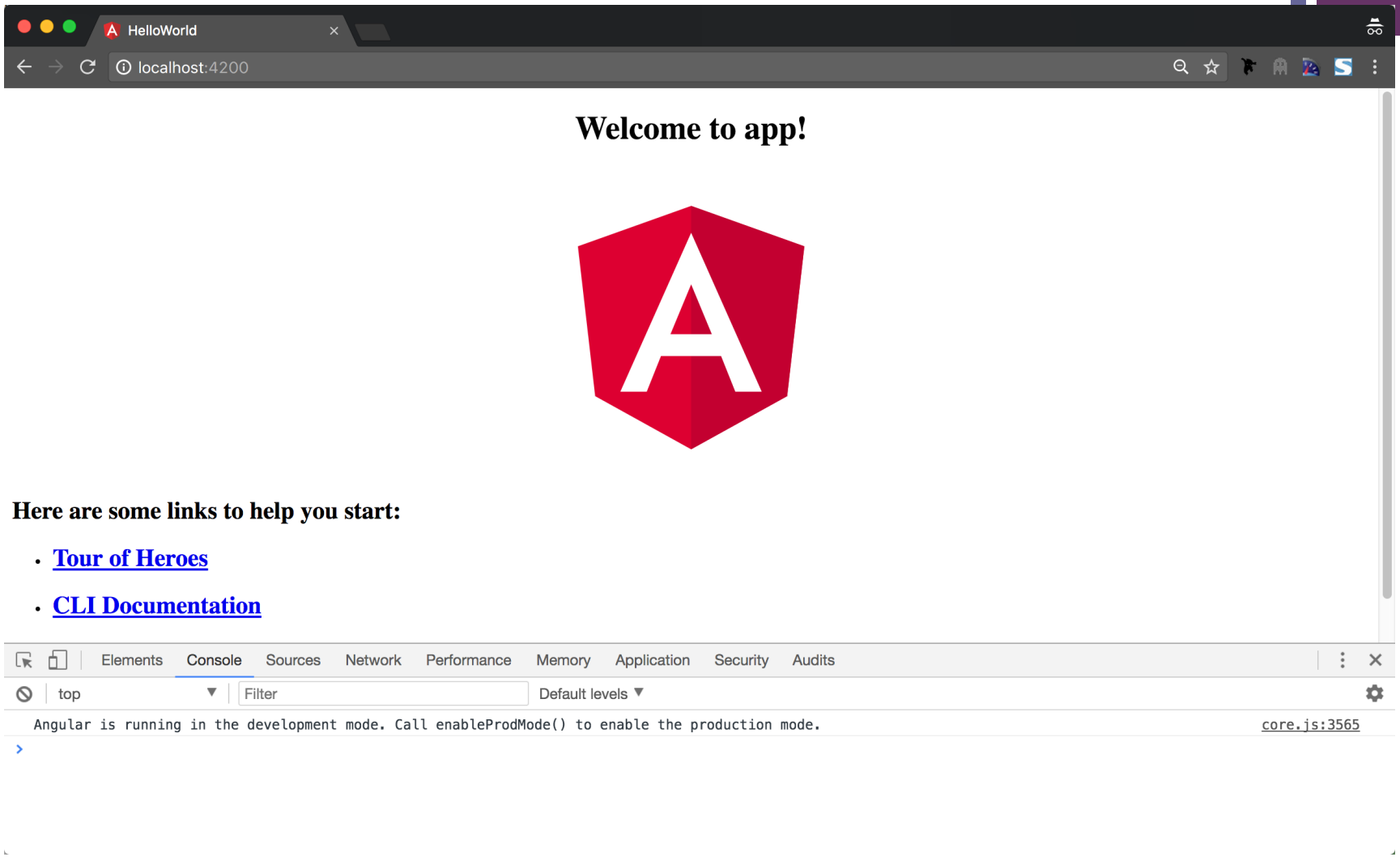
ng serve

Installation de notre environnement



http://localhost:4200/

Installation de notre environnement



Installation de notre environnement

Commands	Description
ng help	returns all commands with flags they can take as a param
ng new [project-name]	create a brand new angular project with live server BANG!
ng init	grabs name from folder that already exist
ng build	builds the production version of project
ng build -prod --aot	builds and gzips file and should do tree shaking
ng serve	creates the local version of project with live reload server
ng serve -p 8080	serve with different port number
ng github-pages:deploy	push site to gitHub pages
ng lint	lints files in project
ng generate	takes the arguments below
ng g component my-new-component	Component
ng g directive my-new-directive	Directive
ng g pipe my-new-pipe	Pipe
ng g service my-new-service	Service
ng g class my-new-class	Class
ng g interface my-new-interface	Interface
ng g enum my-new-enum	Enum
ng g module my-module	Module

Définition : Angular

Angular est un framework Javascript créé par Google qui permet de créer et structurer des applications "One Page" avec de l'HTML, CSS et surtout Javascript en créant des briques appelées composants qui s'imbriqueront entre eux.

Définition : Composants

Le concept central d'une application Angular est le composant. En effet, toute l'application peut être modélisée comme un arbre de ces composants.

Un composant contrôle un ensemble d'éléments graphiques visibles sur le navigateur. Il permet de définir leur aspect visuel mais également comment ils vont réagir à diverses actions.

Définition : Directives

Une directive modifie le DOM pour modifier l'apparence, le comportement ou la mise en page des éléments DOM. Les directives sont l'un des principaux éléments constitutifs des applications Angular pour créer des applications. En fait, les composants angulaires sont en grande partie des directives avec des modèles. On a des Atributes Directives comme NgStyle ou NgClass mais aussi des structural Directives comme NgFor et NgIf.

Définition : Interpolation

Interpolation permet d'afficher le contenu de variables définies dans le "controller" dans une vue.

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero}}</h2>
  `
```

Définition : Pipes

Un pipe est un élément du framework Angular permettant d'effectuer des transformations directement dans le template. Ils sont utilisés pour mettre en forme une date ou un nombre. Ils peuvent également être utilisés pour effectuer des filtres.

Définition : Services

Si un élément de code, des données, etc. est requis par de nombreux composants de notre application. Une bonne pratique consiste à créer un Service réutilisable. Puis par un mécanisme de d'injection de dépendances, on pourra dire aux composants de notre choix d'accéder à tel ou tel Services et aux données/fonctions/etc. qu'il propose.

Définition : Routing

Le routage nous permet d'exprimer certains aspects de l'état de l'application dans l'URL. Contrairement aux solutions frontales côté serveur, cela est facultatif: nous pouvons créer une application complète sans jamais modifier l'URL. L'ajout de routage, cependant, permet à l'utilisateur d'accéder directement à certains aspects de l'application. Ceci est très pratique, car il peut garder votre application pouvant être liée et personnalisable et permettre aux utilisateurs de partager des liens avec d'autres.

Définition : Promises

Une promesse est un objet représentant le résultat futur d'une action exécutée de manière asynchrone, pouvant donc terminer à n'importe quel moment. Il nous est possible d'y greffer des callbacks de succès et d'erreur, respectivement lorsque la promesse est résolue ou rejetée, afin de pouvoir traiter le résultat renvoyé par la promesse.

Définition : Observables

Reprend le principe des Promises mais va plus loin avec la possibilité d'annuler une requete en cours…

Différences entre les versions d'Angular

- Angular 4 : une meilleure compilation des JS et des vues (jusque 60% de compression)
- Angular 4 : La "directive" **template** est désormais appelée **ng-template**.
- Angular 4 : ngIf possède désormais la condition **ELSE**.
- Angular 4 : Rajoute le PIPE **titlecase** (eq. du ucwords en PHP)
- Angular 4 : Modifie légèrement l'écriture des requêtes GET

```
http.get(`${baseUrl}/api/movies`, { params: { sort: 'ascending' } });
```

- Angular 5 : Ajout du support pour les Progressive Web Apps

Site mobile ou appli mobile ? Le débat fait rage depuis des années. Il existe désormais une solution qui allie les points forts des 2 options pour optimiser l'expérience utilisateur : les progressive web apps (ou PWA).

Les Progressive Web Apps

- Les Progressive Web Apps utilisent les dernières technologies pour combiner les meilleures applications Web et mobiles. Pensez-y comme un site Web construit à l'aide de technologies Web, mais qui agit et se sent comme une application.
- **Progressive :** S'adapte à n'importe quel device et s'améliore progressivement suivant les fonctionnalités du device
- **Découverte :** accessible sans téléchargement et depuis un simple navigateur
- **Liens :** Les URLs indiquent l'état actuel de l'application. Rechargement, partage, etc. simplifiés.
- **App-like :** Une PWA doit ressembler à une application native et fonctionner de la même manière

Les Progressive Web Apps

- **Connectivité indépendante :** la PWA doit fonctionner dans des zones de faible connectivité voire hors ligne
- **Ré-engageable :** Grâce aux notifications et d'autres fonctionnalités, possibilité de fidéliser l'utilisateur
- **Installable :** Possibilité d'installer la PWA sur l'écran d'accueil de l'appareil
- **Actualisée :** Lorsque de nouveaux contenus apparaissent et que l'utilisateur est connecté à Internet, l'application se met à jour.
- **Sûr :** HTTPS, HTTPS, HTTPS…
- *Exemples de PWA : https://pwa.rocks/*

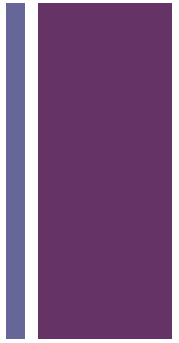
ES2015 (ES6) et TypeScript – Appréhendez le futur du Web

Introduction à TypeScript

What is TypeScript?

- A superset of JavaScript created by Microsoft
- Transpiled into JavaScript
- Focuses on:
  - Application-scale language which adds type safety
  - Modern features
    - ES6 – classes, arrow functions
    - ES7 – async/await
    - ES?

Main reason why TypeScript is great



"TypeScript doesn't try to NOT be JavaScript – it just tries to make JavaScript better"

–Spencer

Types

- Everything derives from object!
- number
- string
- boolean
- Array

Types, cont'd

- "any" type
- Means anything goes

```
function sayHi(sayer: any) {
    console.log("Hi from " + sayer);
}

sayHi("Spencer");
sayHi(42);
```

Quick note on strings

- You can still declare strings using double quote or single quote:

```
let hisName = "George";
let herName = 'Tracy';
```

- But you can also use template strings!
- Multiline AND interpolation… just use backticks!

```
let name = `
Hi from ${hisName}
and ${herName}!
`
```

Variables

- Largely the same as JavaScript

```
var name = "Spencer";
var name: string = "Spencer";
```

- TypeScript adds type hints

```
name = 3;
```

- TypeScript is strongly typed

Variables : let or var?

- The scope of a variable defined with var is function scope or declared outside any function, global.
- The scope of a variable defined with let is block scope.

```
function varvslet() {
  console.log(i); // i is undefined due to hoisting
  // console.log(j); // ReferenceError: j is not defined

  for( var i = 0; i < 3; i++ ) {
    console.log(i); // 0, 1, 2
  };

  console.log(i); // 3
  // console.log(j); // ReferenceError: j is not defined

  for( let j = 0; j < 3; j++ ) {
    console.log(j);
  };

  console.log(i); // 3
  // console.log(j); // ReferenceError: j is not defined
}
```

Variables, cont'd

- Notice something strange?

```
var name = "Spencer";
var name: string = "Spencer";
```

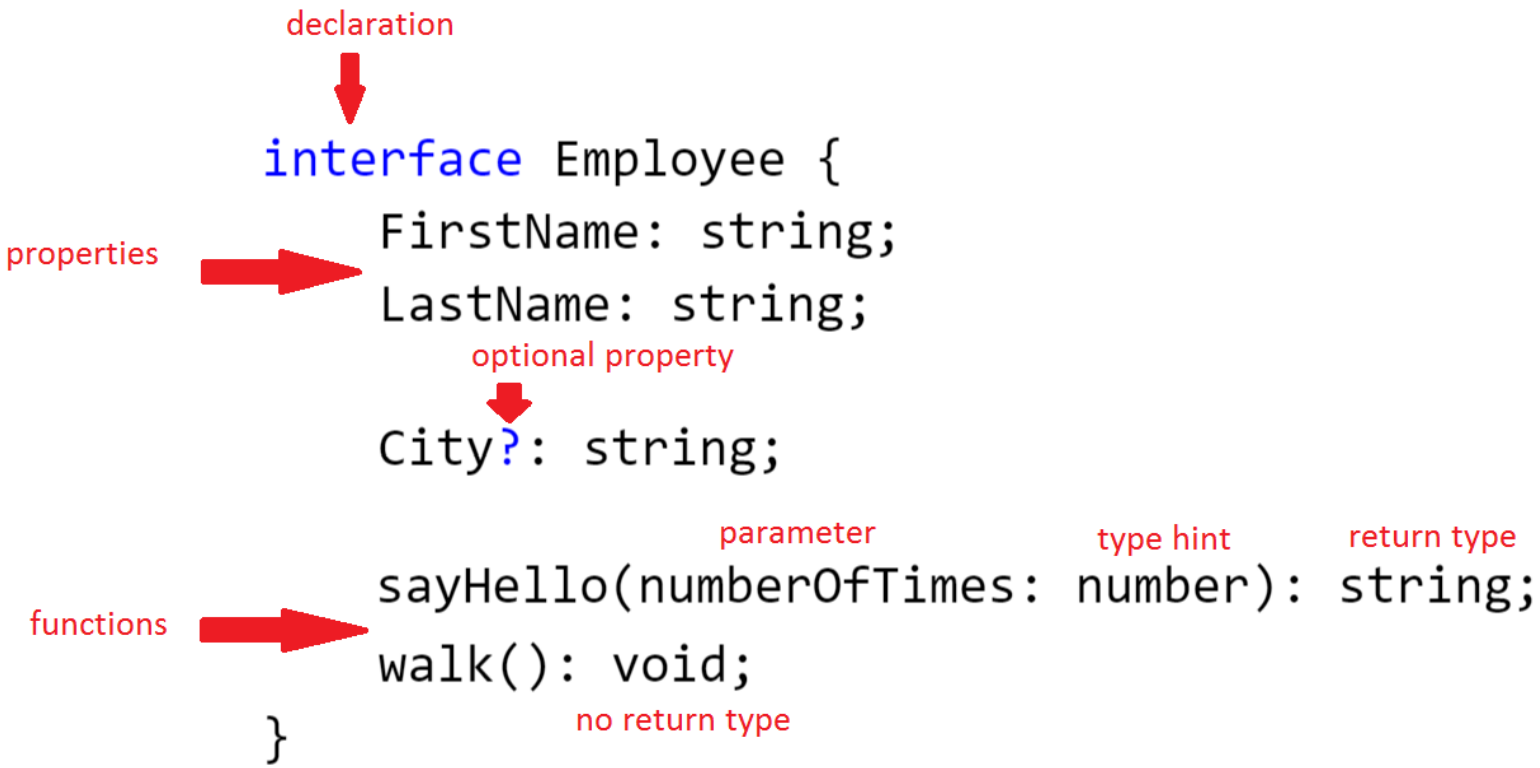
- Yes, it's declared twice – and that's totally valid with variables declared with "var" keyword
- Use let or const instead – it's safer

```
let name = "Spencer";
let name: string = "Spencer";
```

- Not valid with variables declared with "let" keyword

Interfaces

- Classes without functionality – just like C#



Interfaces, cont'd

```
interface Employee {
    FirstName: string;
    LastName: string;

    City?: string;
}
```

- Won't let you compile with this type hint if you don't fulfill the entire contract!

```
let employee: Employee = {
    FirstName: "Spencer",
    City: "St. Louis"
};
```

More interfaces

- Duck typing: "If it walks like a duck and quacks like a duck… it's a duck"

```
function sayHello(emp: Employee) {
    console.log("hello " + emp.FirstName);
}

sayHello({FirstName: "John", LastName: "Lackey"});
```

- This is valid! And good! Respects JavaScript while adding safety

Classes

- Classes can define behaviors AND properties

```
class Animal {
    name: string;
    constructor(theName: string) {
        this.name = theName;
    }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

Classes, cont'd

- Inheritance is a thing

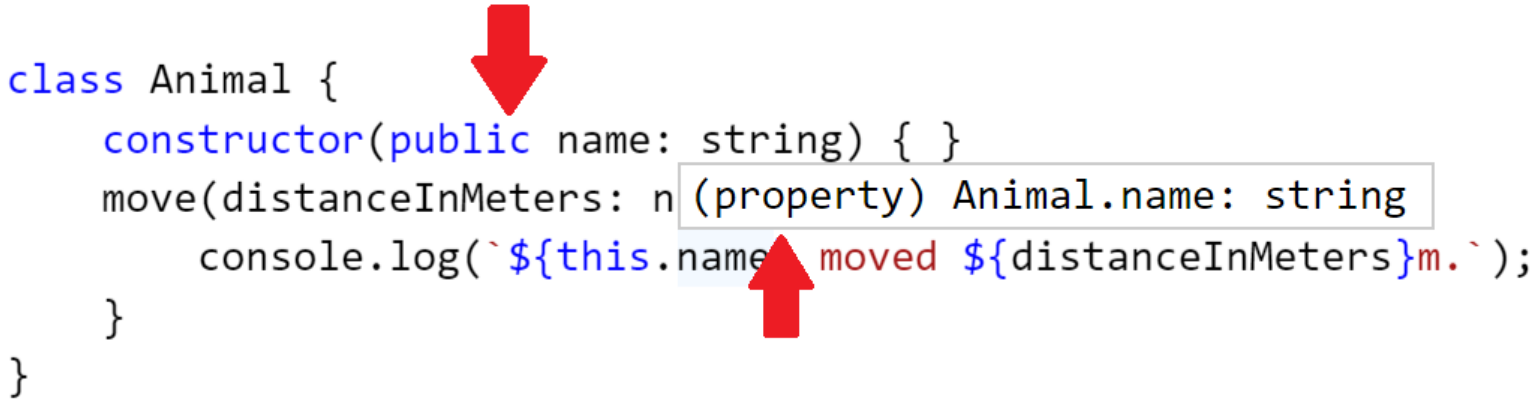
```
class Snake extends Animal {
    constructor(name: string) {
        super(name);
    }

    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
```

Using classes

```
let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

Constructor/property shorthand

- Adding an accessibility modifier to a parameter in the constructor automatically converts it to a property



Functions

- Perform actions

```
// Named function
function add(x: number, y: number): number {
    return x + y;
}

// Anonymous function
let myAdd = function(x: number, y: number): number {
    return x+y;
};

// Anonymous function declared as an arrow function
let anotherMyAdd = (x: number, y: number) => x + y;
```

Generics

- Like C# and VB.NET generics
- Most common generic: Array type

```
class Employee {
    FirstName: string;
    LastName: string;
}

let employees: Employee[] = [];
let moreEmployees: Array<Employee> = [];
```

```
push(...items: Employee[]): number
...items: Employee[] New elements of the Array.
employees.push()
```

this

- Lots of power, lots of heartache, as `this` can be changed function by function
- Our main use for it: know that you need to use this to access variables in class scope

```
class Employee {
    FirstName: string;
    LastName: string;

    sayHello() {
        console.log(`Hello from ${this.FirstName} ${this.LastName}`);
    }
}
```


Decorators

- Basically, attributes in C#/VB.NET
- Considered experimental feature – can only be turned on with compiler option

```
@Component({
    selector: 'expense-app',
    templateUrl: './app/app.component.html'
})
export class AppComponent {  }
```

Decorators

- They are an "annotation" mechanism that give meanings and additional features to the elements they are attached to
- @Component is an annotation that tells Angular, that the class, which the annotation is attached to, is a component.
- They are Class Decorators, Property Decorators (like @Input, @Output), Method Decorators and Parameter Decoration

import/export

- Like `using` statements in C#/VB.NET
- `import` statements allow us to get code from other TypeScript declarations
- `export` simply means public

person.ts katas\4\app

```
1   export interface Person {
2       FirstName: string;
3       LastName: string;
4   }
```

```
import {Person} from "./person"

var matt: Person = {
    FirstName: "Matt",
    LastName: "Carpenter"
}
```

Truthy/falsy

- TypeScript has null and undefined, like JavaScript
- And they are NOT the same thing!
- You don't check for null, you check for truthiness/falsiness
- Falsy values:
  - false
  - null
  - undefined
  - 0
  - "" (empty string)
  - NaN (which means not a number)

Premiers pas sous Angular

Components

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

@Component({
    moduleId: module.id,
    templateUrl: 'app.component.html',
    selector: 'my-app'
})
export class AppComponent {}
```

Modules

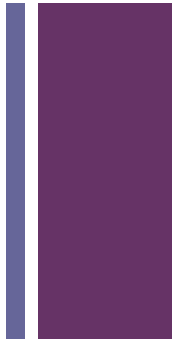
```
@NgModule({
  imports: [
    routing, BrowserModule, FormsModule, KataModule
  ],
  declarations: [
    AppComponent, HomeComponent
  ],
  providers: [
    appRoutingProviders
  ],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule {}
```

Running your app

- Import your root NgModule
- Bootstrap it!

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule }              from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Steps to setting up your app



1. Write your module
2. Write your component
3. Write “main” method
4. Import the proper scripts
5. Choose your module loader
6. Run your app!

Components and Templating

Components, components, components…

- Components in Angular are awesome
- Composable and easy to maintain
- Lots of smaller components > Fewer big components
- Follow the Web Component ideal
  - Abstracting a large chunk of HTML into a single element
  - "Building blocks"
  - Functionality is fully encapsulated – expose only what needs exposing

Intro to Templates

- Templates contain the HTML that is rendered by your component
- Can be placed inline inside of Component declaration OR in a separate file

```
@Component({
    selector: 'my-app',
    template: '<h4>{{successString}}</h4>'
})
export class AppComponent {
    successString = "If you see this, your Angular 2 app is working!";
}
```

Ways to bind component to view

- Interpolation expressions
- Property binding
- Event binding
- ngModel

Interpolated expressions

- {{ }} syntax
- Put your expression inside the brackets
- Accepts non-side-effect-JavaScript (no new, assignments, etc.)
- Best use case – bind pure properties to it when possible

```
@Component({
    selector: 'my-app',
    template: '<h4>{{successString}}</h4>'
})
export class AppComponent {
    successString = "If you see this, your Angular 2 app is working!";
}
```

Interpolated expressions, cont'd

- What if you have something like this?

```
<div>
    {{ employee.FirstName }}
</div>
```

```
@Component({})
export class MyComponent {
    employee: Employee = undefined;
}
```

- This will throw an error. And this sucks ->

```
<div>
    {{ employee && employee.FirstName }}
</div>
```

- Elvis to the rescue! (…I don't know why they call it the Elvis operator. It's called "monadic null checking")

```
<div>
    {{ employee?.FirstName }}
</div>
```

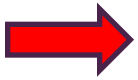
Property binding

- Bind directly to properties on the DOM model using []
- The power and convenience of this cannot be understated!
- Allows something like this:

```
let customerName = document.getElementById("#customerName");
customerName.style.color = "red";
```

- To become this:

```
export class MyComponent {
    myColor = "red";
    customer = {Name: "David"};
}
```



```
<div [style.color]="myColor">
    {{ customer.Name }}
</div>
```

Property binding, cont'd

- You can also bind to attributes by using [attr.name]
- Example: aria (accessibility) attributes don't have DOM model property bindings

Directives

- Used to modify behavior of DOM elements
- Structural directives (prefixed with *) can add/remove DOM elements

```
<ul>
  <li *ngFor="#user of users">
    {{ user.name }} is {{ user.age }} years old.
  </li>
<ul>
```

Directives

- Used to modify behavior of DOM elements
- Structural directives (prefixed with *) can add/remove DOM elements

```
<div *ngIf="isValid;else other_content">
    content here ...
</div>

<ng-template #other_content>other content here...</ng-template>
```

Exercices

Veuillez reprendre créer un HelloWorld en vous basant sur le nutshell. L'objectif est d'afficher "*Hello, my name is XXXX*" au lancement de l'application dans le navigateur. Merci de renommer le composant MyName.component.ts également.

Event binding

- Bind directly to events on the DOM model using ()
- Again, the power and convenience cannot be understated
- Support for custom events

```
@Component({
    selector: 'my-app',
    template: `<button (click)="showMeAlert()">Click me!</button>`
})
export class AppComponent {
    showMeAlert() {
        alert("Show me the money");
    }
}
```

ngModel

- Allows two-way binding
- Follows unidirectional data flow concepts

Two essential directives

- `*ngIf`
  - Optionally renders an element based on the truthiness of an expression
- `*ngFor`
  - Looping directive
  - Replaces `ng-repeat` from Angular 1

Exercices

Reprenons notre HelloWorld et rajoutons un <INPUT> qui va permettre de modifier le nom affiché à l'écran en fonction de la valeur saisie dans le <INPUT>.

Child components

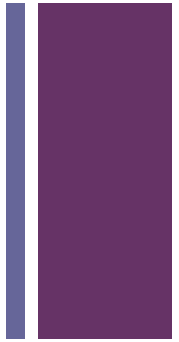
What is a child component?

- A child component is a component that load a subpart of a bigger component
- I know, pretty stunning right?

Ok, well how do you make one?

- Easy: you make a component!
- Only a couple of steps to get it into a parent component

Step 1: Declare your component



```
@Component({
    selector: "my-child",
    template: "This is a child component"
})
export class ChildComponent {}
```

Step 2: Add child selector to markup

```
@Component({
    selector: "my-child",
    template: "This is a child component"
})
export class ChildComponent {}
```

```
@Component({
    selector: "app",
    template: "<my-child></my-child>",

})
export class AppComponent {}
```

You're done…

- Your child component is totally isolated – it can't interact with your parent!
- True encapsulation!
- …
- Ok, maybe we need to add some more stuff to make it useful in The Real World™

@Input properties

- Means of binding data from the parent to the child
- Requires two steps
  - Add property to child and decorate it with @Input
  - Bind the parent property to the child using [] syntax

Step 1: Add the input property

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

@Component({
    selector: "my-child",
    template: "This is a child component with a message: {{ message }}"
})
export class ChildComponent {
    @Input() message: string;
}
```

Step 2: Bind the parent property to that property

```
@Component({
    selector: "app",
    template: `<my-child [message]="marketingPhrase"></my-child>`,
    directives: [ChildComponent]
})
export class AppComponent {
    marketingPhrase = "Drink more Ovaltine";
}
```

Event emitting and @Output

- Parents can communicate with child components easily via input properties
- How does the child communicate back up?
- Simple: @Output properties and EventEmitter!
- A little more to it than @Input properties, but very easy

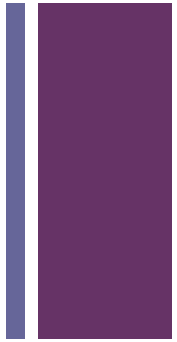
Steps involved

1. Import the EventEmitter and Output types
2. Add property to child of type EventEmitter decorated with @Output
3. Write subscriber function in parent
4. Bind subscriber function to event using () syntax
5. Emit an event!

Steps 1 and 2

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

@Component({
    selector: "my-child",
    template: `This is a child component with a message: {{ message }}`
})
export class ChildComponent {
    @Input() message: string;
    @Output() signaledIsHungry = new EventEmitter<string>();
}
```

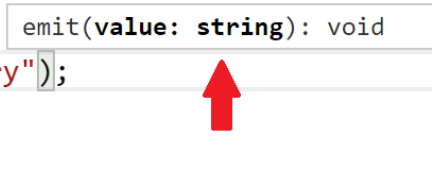
Steps 3 and 4



```
@Component({
    selector: "app",
    template: `<my-child [message]="marketingPhrase"
                         (signaledIsHungry)="didSignalIsHungry($event)"></my-child>`,
    directives: [ChildComponent]
})
export class AppComponent {
    marketingPhrase = "Drink more Ovaltine";

    didSignalIsHungry(message: string) {
        console.log(message);
    }
}
```

Step 5 – Emit your event

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

@Component({
    selector: "my-child",
    template: `This is a child component with a message: {{ message }}

    <button (click)="complainOfHunger()">Say I'm hungry</button>`
})
export class ChildComponent {
    @Input() message: string;
    @Output() signaledIsHungry = new EventEmitter<string>();

    complainOfHunger() {
        this.signaledIsHungry.emit("I'm hungry");
    }
}
```



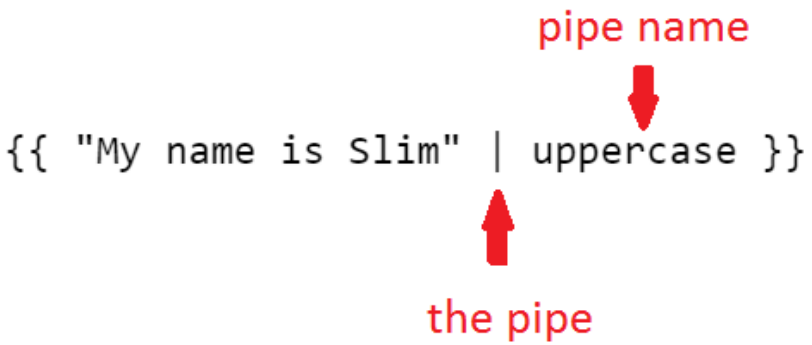
Exercices

Maintenant ajoutons un <IMG> et un <BUTTON>. Choisissez 3 images sur Google et tentons de créer un slider qui change les images à chaque fois que l'on clique sur le <BUTTON>. Le parent détient le bouton qui permet de passer d'une image à l'autre et le composant fils contient la liste des images (le slider).

Pipes

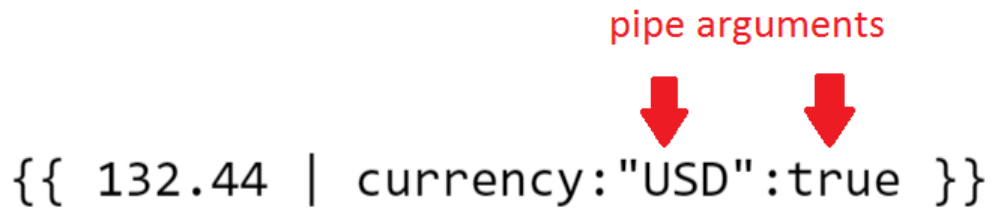
Pipes

- Filters in Angular 1
- Alters the display of an interpolated expression



Pipes, cont'd

- Pipes can take arguments/parameters



- Parameters are expressions like everything else
- Separated by colons :

Some built-in pipes

- number
- date
- currency
- percent
- json
- uppercase/lowercase
- async
  - But that's another show

Exercices

Veuillez afficher votre nom, dans l'exercice précédent, en majuscule. N'hésitez pas à explorer l'ensemble des Pipes sur https://angular.io/docs/ts/latest/guide/pipes.html et https://angular.io/docs/ts/latest/api/.

Architecture des composants

Atomic Design - L'introduction

En informatique, la programmation modulaire reprend l'idée de fabriquer un produit (le programme) à partir de composants (les modules).

Elle décompose une grosse application en modules, groupes de fonctions, de méthodes et de traitement, pour pouvoir les développer et les améliorer indépendamment, puis les réutiliser.

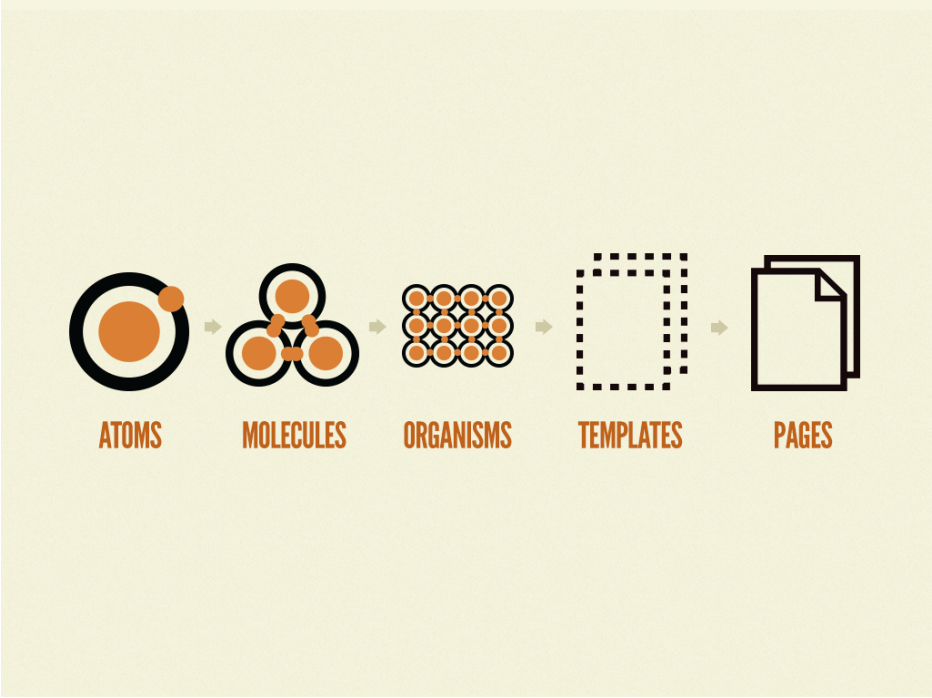
Le développement du code des modules peut être attribué à des (groupes de) personnes différentes pour accélérer les développements.

Atomic Design - L'introduction

Cette logique de programmation est reprise par Brad Frost pour concevoir nos modules en cinq niveaux différents. Nous partirons de l'abstrait (les éléments HTML basiques qu'on appellera atome) au concret (le template de notre site). Cette méthode permet de créer des systèmes évolutifs fonctionnant par assemblages d'éléments de différents niveaux.

En plus d'améliorer la maintenabilité du code, cela aura une influence sur l'homogénéité de l'interface de l'application créée et augmentera la satisfaction du client sur le site

Atomic Design



Atomic Design - Définitions

Atomes

Le premier niveau correspond aux atomes. On part de l'abstrait : des éléments simples comme les balises de titre (h1 à h6), de formulaire (input, label, button), constitueront ce niveau. Dans ce niveau, on associe des styles à des éléments HTML simples, sans l'utilisation de classes ou d'ids.

Atomic Design - Définitions

Molécules

Dans le niveau suivant, on assemble nos atomes pour créer les différentes sections de nos molécules. En plus d'assembler des atomes entre eux, les molécules peuvent prendre des propriétés qui leurs seront propres.

Atomic Design - Définitions

Organismes

De la même façon, les organismes imbriquent les molécules entre elles et pourront appliquer des styles spécifiques aux molécules ou à l'organisme.

Atomic Design - Définitions

Templates

On arrive au concret : le groupement des différents organismes forme le template du site.

Atomic Design - Définitions

Pages

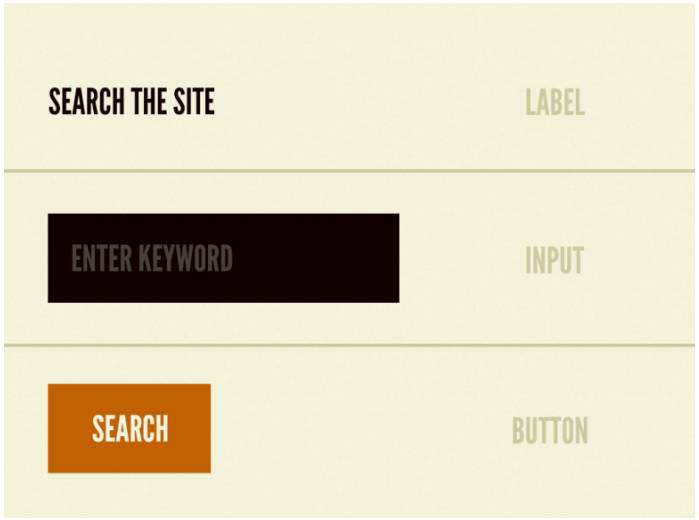
Ce dernier niveau correspond à l'instance spécifique de notre template et de l'ensemble de nos niveaux. La structure de chaque page pourra être modulée en intervenant sur les styles des molécules, des organismes, et des templates en fonction de la spécificité de la page.

Atomic Design - Définitions

Intégrons ensemble un moteur de recherche dans le “header“ de la page comme le ferait un développeur ou un UX suivant les préceptes de l’Atomic Design...

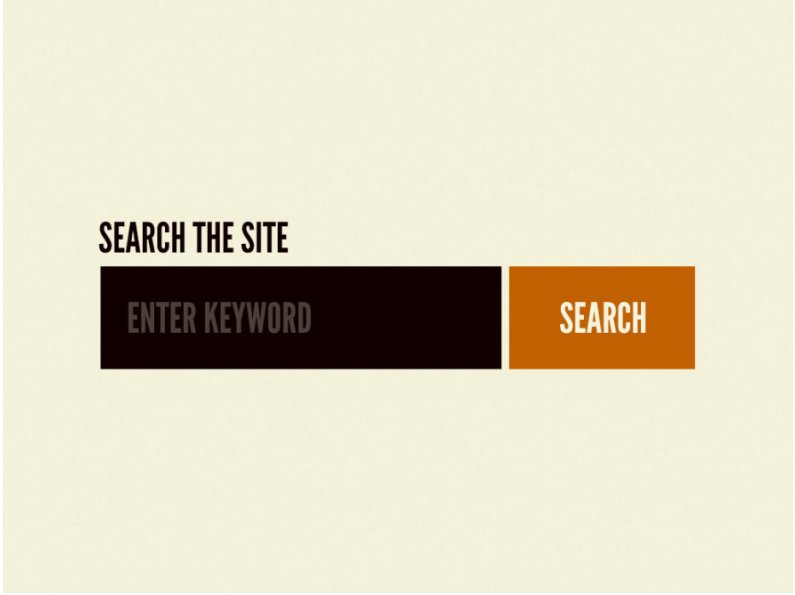
Atomic Design

Première étape : On créé les atomes ! Un label HTML, un champ texte HTML et un bouton HTML. Créés séparément, on peut les utiliser comme on le souhaite où on le souhaite !



Atomic Design

Deuxième étape : Ces atomes sont combinés pour créer la molécule “formulaire de recherche”. Combinés autrement, on aurait également pu créer une autre molécule “formulaire de connexion”, etc.

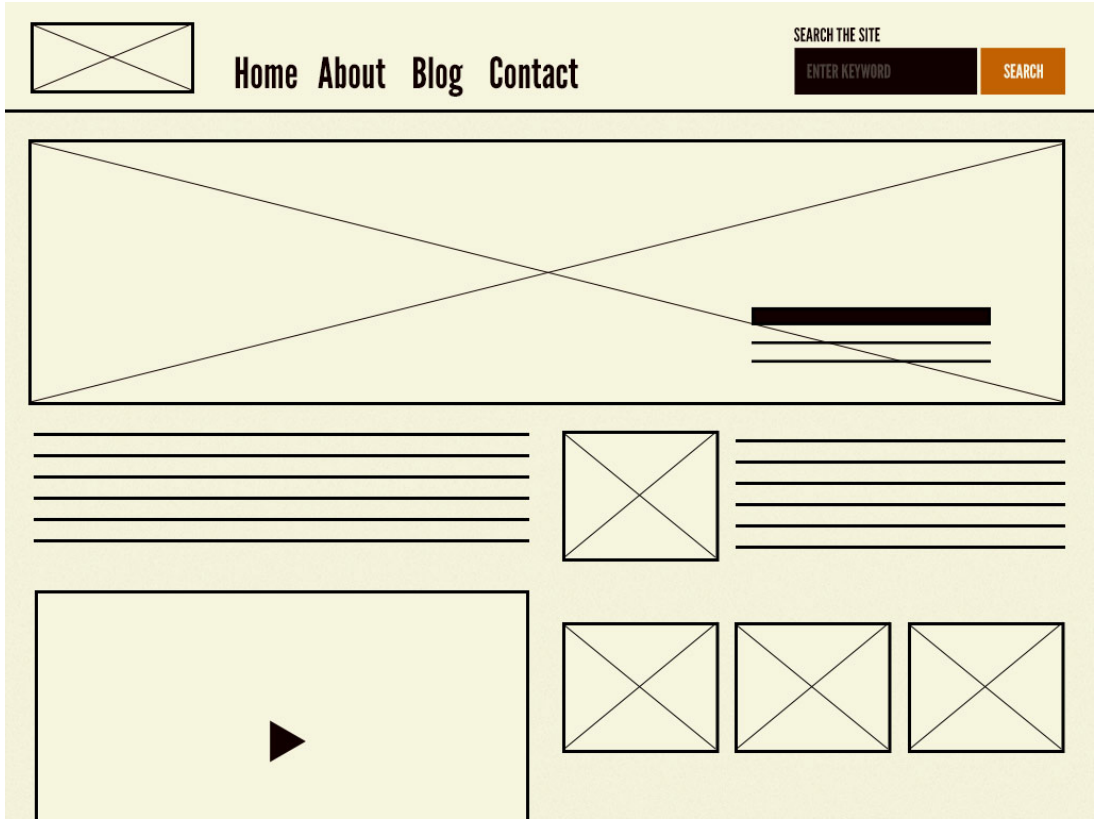


Atomic Design

Troisième étape : On intègre le formulaire dans l'organisme "header". Cet organisme contient également d'autres molécules qui contiennent elles-même des atomes, etc.

Atomic Design

Dernière étape : La combinaison des organismes permet de créer notre template de site !!!



Services

What are Services?

- Services are best used to create, read, update and delete data
- Enforces separation of concerns
- Think of it like this: you don't want your database in your view!

Guess what? A service is just a class

- Are you surprised?

Créer des Services en ligne de commande

ng generate service [NAME]

Ready for refactoring

```
@Component({
    selector: "app",
    template: `
    <div *ngFor="let employee in employees">
        {{ employee.LastName }}, {{ employee.FirstName }}
    </div>
    `
})
export class AppComponent {
    employees = [
        {FirstName: "Martin", LastName: "Marr"},
        {FirstName: "Liz", LastName: "Garcia"}
    ];
}
```

Steps to declaring and using a service

1. Write your class and add some data to it
2. Import Injectable and decorate your service class with it
3. Import the service in your component
4. Add the service to the component providers
5. Add it to the component as a property and in the constructor

Step 1 – Make class with data

```
export class EmployeeService {
    getEmployees() {
        return [
            {FirstName: "Martin", LastName: "Marr"},
            {FirstName: "Liz", LastName: "Garcia"}
        ];
    }
}
```

Step 2 – Import @Injectable and decorate

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

@Injectable()
export class EmployeeService {
    getEmployees() {
        return [
            {FirstName: "Martin", LastName: "Marr"},
            {FirstName: "Liz", LastName: "Garcia"}
        ];
    }
}
```

Steps 3 and 4 – Import and register

```
import {EmployeeService} from "./employee.service";

@Component({
    selector: "app",
    template: `
    <div *ngFor="let employee in employees">
        {{ employee.LastName }}, {{ employee.FirstName }}
    </div>
    `,
    providers: [EmployeeService]
})
export class AppComponent {
```

This is dependency injection

- A single instance will flow from parent down to child if the child requests an instance in its constructor
- Dependencies are auto-resolved – if one service is dependent on another, the "parent" service will be injected into the dependent service
- Allows for modularity and better testing

Step 5 – Add as property and to constructor

```
@Component({
    selector: "app",
    template: `
    <div *ngFor="let employee in employees">
        {{ employee.LastName }}, {{ employee.FirstName }}
    </div>
    `,
    providers: [EmployeeService]
})
export class AppComponent {
    constructor(private employeeService: EmployeeService){}
}
```

Hey wait a second…

```
@Component({
    selector: "app",
    template: `
    <div *ngFor="let employee in employees">
        {{ employee.LastName }}, {{ employee.FirstName }}
    </div>
    `
})
export class AppComponent {
    employees = [
        {FirstName: "Martin", LastName: "Marr"},
        {FirstName: "Liz", LastName: "Garcia"}
    ];
}
```

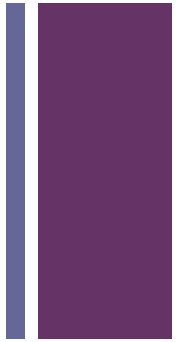
Good time to introduce… Lifecycle hooks!

- Lifecycle hooks allow us to run code after a component is created and/or destroyed
- Focusing on two:
  - OnInit
  - OnDestroy
- But really only one:
  - OnInit

OnInit

- OnInit – run right after a component is constructed
- If a component has the "ngOnInit" function on it, the OnInit lifecycle hook will execute
- There's also a helpful interface that tells you that you haven't implemented it yet if you mean to implement it

Step 1: Import OnInit



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

Step 2: Implement

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

export class AppComponent implements OnInit {
    employees: any[];
    constructor(private employeeService: EmployeeService){}

    ngOnInit() {
        this.employees = this.employeeService.getEmployees();
    }
}
```

Exercices

Passez l'ensemble des data sur les précédentes images venant de Google dans un Service !

+

RxJS, HTTP and Observables

History

- Promises
  - A function that promises to return at some time
  - Async functions in JavaScript
  - Extremely popular for a long time in Angular, jQuery… pretty much everything
- Limitations of promises
  - Promises have a very simple API surface – both good and bad
  - Can't be easily cancelled in flight

Enter RxJS

- RxJS (or Reactive Extensions) is a library created by Microsoft
- Allows for asynchronous, event-based applications
- Much more powerful than promises…
- …but a little harder to understand
- …with a large API surface

RxJS is in Angular

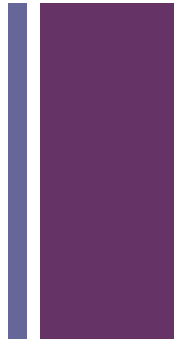
- Google went all in with RxJS
- Promises aren't a thing of the past, but RxJS is definitely a glimpse into the future

Observable

- Like a promise on steroids
- Promises are one and done
- Observables can be repeated

- Think of it like downloading a movie file and then watching it vs. streaming on Netflix

Observables are ridiculous



- Really outside the scope of this workshop, but worth mentioning
- …because the Angular HTTP library uses Observables

Angular's HTTP Service

- Allows us to perform HTTP requests
- Easy to use!
- Simply add to your service, register with the application, and start making requests!

Steps

1. ~~Register HTTP_PROVIDERS with your application~~
2. Add the Http service to your service
3. Call HTTP methods and return their results

Step 1 – Add to your bootstrap method

```
import {bootstrap}    from '@angular/platform-browser-dynamic';
import {AppComponent} from './app.component';
import {HTTP_PROVIDERS} from "@angular/http";

bootstrap(AppComponent, [HTTP_PROVIDERS]);
```

- This is like a super provider property in our component

Step 2 – Add Http to your service

```
import {Http} from "@angular/http";

@Injectable()
export class EmployeeService {
    constructor(private http: Http) {}

    getEmployees() {
        return [
            {FirstName: "Martin", LastName: "Marr"},
            {FirstName: "Liz", LastName: "Garcia"}
        ];
    }
}
```

Step 3: Call some Http methods!

```
import {Http} from "@angular/http";
import {Observable} from "rxjs";
import "rxjs/add/operator/map";

@Injectable()
export class EmployeeService {
    constructor(private http: Http) {}

    getEmployees() {
        return this.http.get("./api/Employees")
                        .map(response => response.json());
    }
}
```

Breaking it down

- This imports the "map" operator, which we can use to convert our "Response" object to something useful

```
import "rxjs/add/operator/map";
```

- In this case, we're telling it that our HTTP response will come in the form of JSON, so deserialize the JSON to a usable object

```
getEmployees() {
    return this.http.get("./api/Employees")
                    .map(response => response.json());
}
```

What we're getting back

- An Observable<any>, which means that this Observable says at some point it will return an object of type any

```
(method) EmployeeService.getEmployees(): Observable<any>
getEmployees() {
    return this.http.get("./api/Employees")
                    .map(response => response.json());
}
```

We can make it more useful with a type hint

- Adding a type hint to the method makes it return something more useable in an application

```
(method) EmployeeService.getEmployees(): Observable<Employee[]>
getEmployees() : Observable<Employee[]>
    return this.http.get("./api/Employees")
                    .map(response => response.json());
}
```

Still a bit more work to do

```
export class AppComponent implements OnInit {
    employees: Employee[];
[ts] Type 'Observable<Employee>' is not assignable to type 'Employee[]'.
  Property 'length' is missing in type 'Observable<Employee>'.
(property) AppComponent.employees: Employee[]
        this.employees = this.employeeService.getEmployees();
    }
}
```

Two easy choices

- We can either subscribe to the response…

```
export class AppComponent implements OnInit {
    employees: Employee[];
    constructor(private employeeService: EmployeeService){}

    ngOnInit() {
        this.employeeService.getEmployees()
            .subscribe(emps => this.employees = emps);
    }
}
```

Or use the async pipe

```
@Component({
    selector: "app",
    template: `
    <div *ngFor="let employee in employees | async">
        {{ employee.LastName }}, {{ employee.FirstName }}
    </div>
    `,
    providers: [EmployeeService]
})
export class AppComponent implements OnInit {
    employees: Observable<Employee[]>;
    constructor(private employeeService: EmployeeService){}

    ngOnInit() {
        this.employees = this.employeeService.getEmployees();
    }
}
```

Me? I prefer subscribing

- But that's just me

Exercices

Veuillez passer l'ensemble de vos images dans un fichier JSON et ajouter du networking dans votre Service !

Formulaires

Remember ngModel?

```
<h2>Inside Form</h2>
<div>
  <label>Change min value:</label>
  <input [(ngModel)]="minValue">
</div>
<div>
  <label>Change max value:</label>
  <input [(ngModel)]="maxValue">
</div>
```

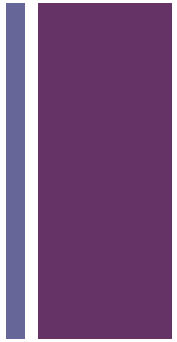
```
counterValue = 3;
minValue = 0;
maxValue = 12;
```

Let's go further…

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

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

Documentation



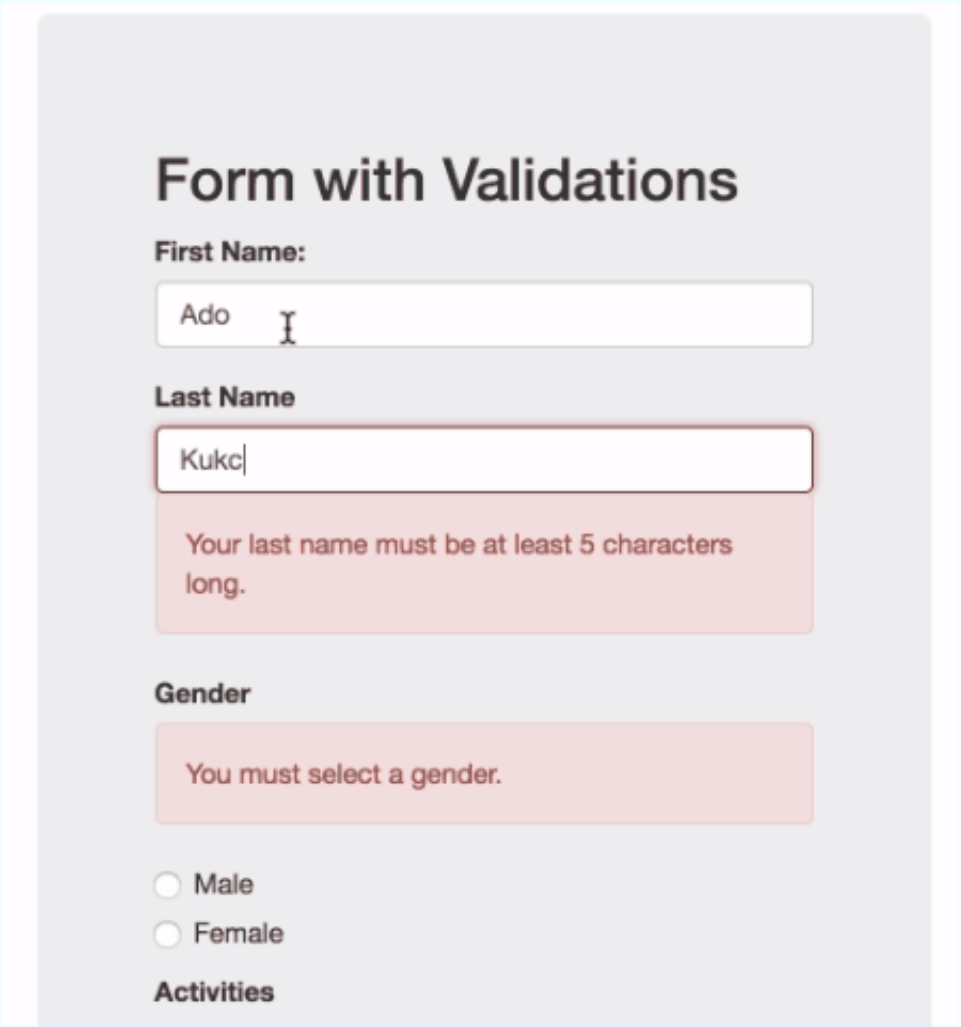
https://angular.io/docs/ts/latest/guide/forms.html

Demo

Exercices

Veuillez intégrer un formulaire permettant de saisir les données d'un nouvel employé (ID unique, nom, prénom, adresse, ville, pays, date de naissance, photo, poste, etc.). Créez ensuite une page factice permettant de voir le detail d'un employé.

Model driven or reactive forms

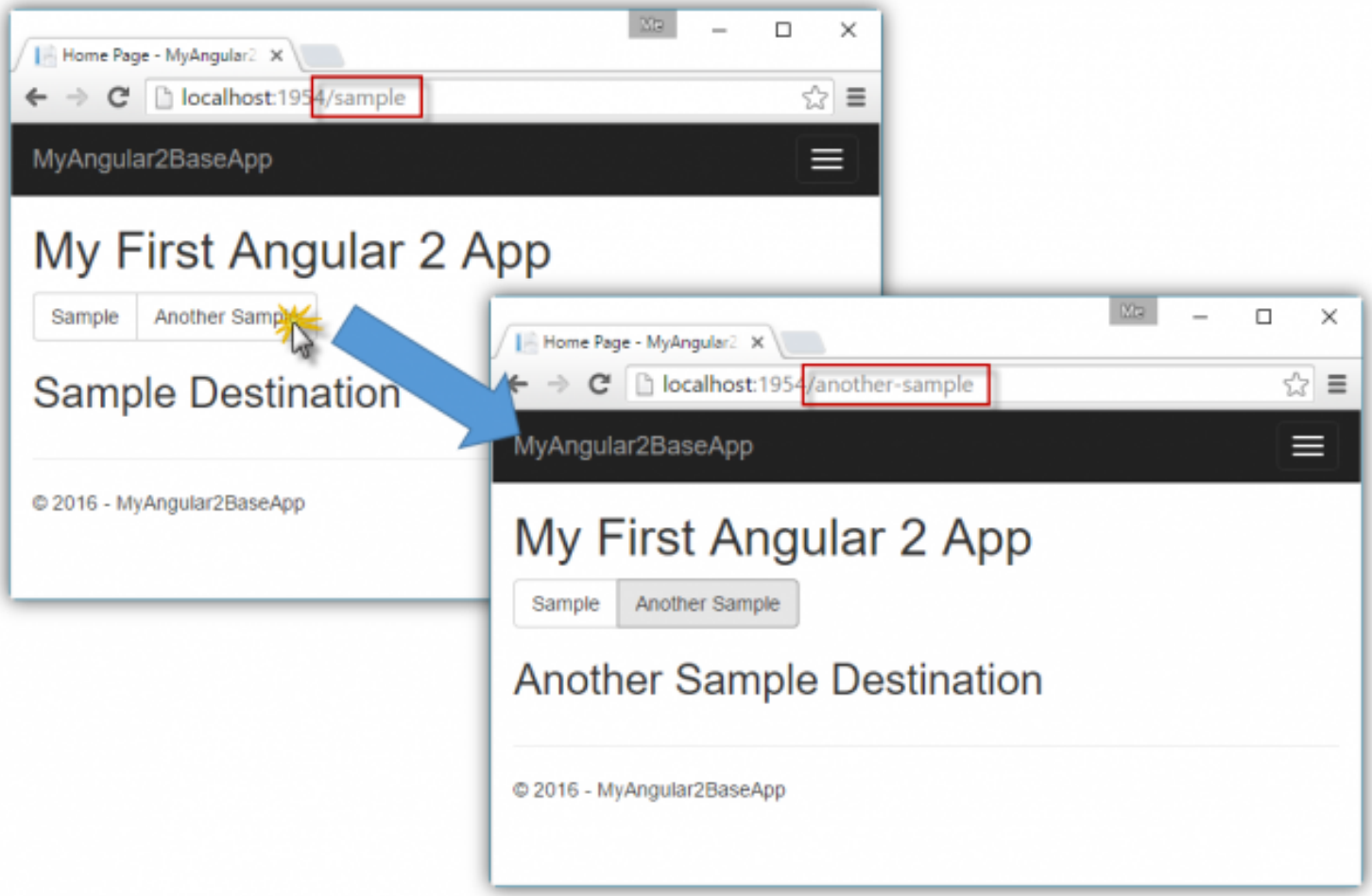


- Ajouter plus de fonctionnalités au formulaire et aux champs
- Mapper le formulaire à un objet/model
- Valider en direct les champs du formulaire

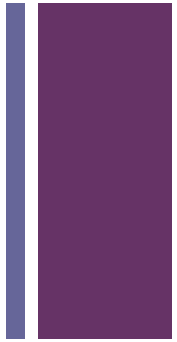
Demo

Routage

Using URLs to navigate the app



Documentation



https://angular.io/docs/ts/latest/tutorial/toh-pt5.html

Demo

Comment tester mon application?

E2E Test Framework

https://angular.io/docs/ts/latest/guide/testing.html

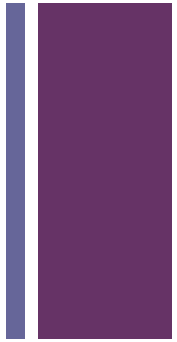
Introduction à Protractor

- Protractor est un framework de test de bout en bout pour les applications Angular. Protractor exécute des tests sur votre application exécutée dans un vrai navigateur, interagissant avec lui comme le ferait un utilisateur.

```
describe('angularjs homepage todo list', function() {
  it('should add a todo', function() {
    browser.get('https://angularjs.org');

    element(by.model('todoList.todoText')).sendKeys('write first protractor test');
    element(by.css('[value="add"]')).click();

    var todoList = element.all(by.repeater('todo in todoList.todos'));
    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('write first protractor test');

    // You wrote your first test, cross it off the list
    todoList.get(2).element(by.css('input')).click();
    var completedAmount = element.all(by.css('.done-true'));
    expect(completedAmount.count()).toEqual(2);
  });
});
```

Déployer son application Angular

Rappels



sudo npm install -g angular-cli

Rappels

ng new HelloWorld

Compilation

ng build

ng build --prod

Des questions? Je reste joignable sur nzeka@nachosmedia.com !