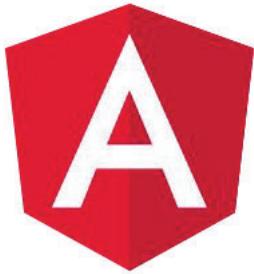


Angular, maîtriser le framework Front-End de Google

CFD

Du mardi 01 au vendredi 04 juin 2021

Angular



whoami

Frédéric GAURAT

Développeur Freelance & Formateur

Objectifs de formation

Objectifs de formation

Organiser, modulariser et tester ses développements JavaScript

Maîtriser les fondamentaux du Framework Angular

Créer rapidement des applications Web complexes

Savoir intégrer les tests unitaires au développement

Connaître les bonnes pratiques de développement et de mise en production

Travaux pratiques

Composition modulaire d'une application avec Angular.

Sommaire

Sommaire

Développement JavaScript : rappels

Migrer d'AngularJS 1.x à Angular

L'utilitaire ng ou @angular/cli

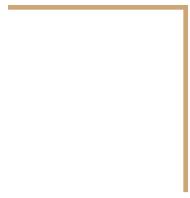
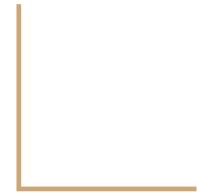
Définition de composants

Classifications des composants applicatifs

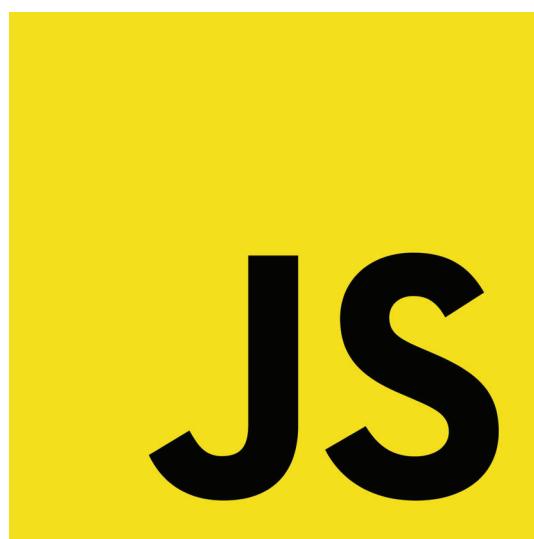
Gestion des formulaires, "Routing" et requête HTTP

Tests unitaires. Bonnes pratiques et outils

Développement JavaScript : rappels



ES5 / ES6 / ES7 / ... présentation générale



Bonnes pratiques ECMAScript 5

Les fonctions

Les portées

Les closures

L'API Array

Nouveautés syntaxiques

Les portées

Fonctions fléchées

Les Class

Template string

Déstructuration de tableau et d'objet

Le développement Objet avec la syntaxe de class

Syntaxe de Base

```
class Rectangle {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
}  
  
const p = new Rectangle();
```

Le développement Objet avec la syntaxe de class

Héritage

```
class Animal {  
    constructor(nom) {  
        this.nom = nom;  
    }  
  
    parle() {  
        console.log(`${this.nom} fait du bruit.`);  
    }  
}  
  
class Chien extends Animal {  
    constructor(nom) {  
        super(nom); // appelle le constructeur parent avec le paramètre  
    }  
    parle() {  
        console.log(`${this.nom} aboie.`);  
    }  
}
```

Le développement Objet avec la syntaxe de class

Membres Statics

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    static distance(a, b) {  
        const dx = a.x - b.x;  
        const dy = a.y - b.y;  
        return Math.hypot(dx, dy);  
    }  
  
    const p1 = new Point(5, 5);  
    const p2 = new Point(10, 10);  
  
    console.log(Point.distance(p1, p2));
```

Le développement Objet avec la syntaxe de class

Membres Getters / Setters

```
class Rectangle {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
  
    get area() {  
        return this.calcArea();  
    }  
    set largeur(largeur){  
        this.largeur = largeur;  
    }  
    get largeur(){  
        return this.largeur;  
    }  
  
    calcArea() {  
        return this.largeur * this.hauteur;  
    }  
  
    const carré = new Rectangle(10, 10);  
    console.log(carré.area);
```

Déstructuration de tableau

```
● ● ●

let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: Array [30,40,50]
```

Déstructuration d'objet

```
● ● ●

const o = {p: 42, q: true};
const {p, q} = o;

console.log(p); // 42
console.log(q); // true

// Assign new variable names
const {p: toto, q: truc} = o;

console.log(toto); // 42
console.log(truc); // true
```

Rest operator

```
● ● ●

function maFonction(a, b, ... autres);
    console.log(a);
    console.log(b);
    console.log(autres);
}

maFonction("un", "deux", "trois");
// affichera ceci dans la console
// "un"
// "deux"
// ["trois"]
```

Rest operator - tableaux

```
● ● ●

const arr = ["toto", "titi", "tata", "tutu"]

let [a, b, ... c] = arr;

console.log(a, b, c)
// "toto", "titi", ["tata", "tutu"]
```

Rest operator - objets

```
const obj = {  
    name: "DURAND",  
    firstname: "Robert",  
    login: "rdurand",  
}  
  
let theobj = { ... obj, name: "MARTIN"}  
  
console.log(theobj)  
// { name: 'MARTIN', firstname: 'Robert', login: 'rdurand' }
```

Le pattern observer/observable

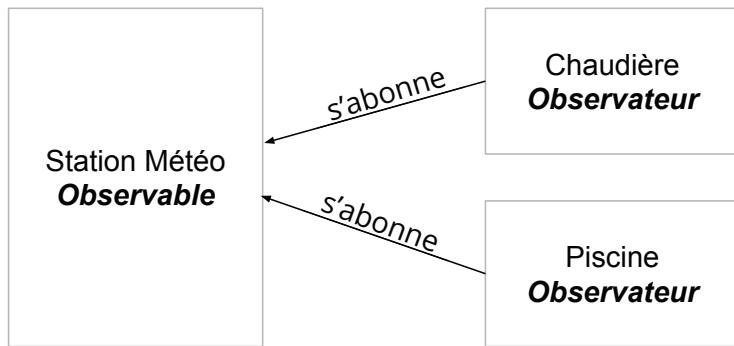
Le pattern **observateur** est un **patron de conception**.

Il est utilisé pour **envoyer un signal** à des modules qui jouent le rôle d'observateurs.

En cas de notification, **les observateurs effectuent l'action adéquate** en fonction des informations qui parviennent depuis les modules qu'ils observent : **les observables**.

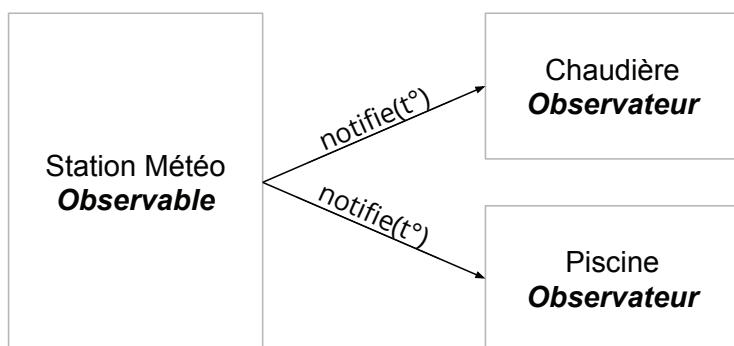
Le pattern observer/observable

Phase 1 : les observateurs s'abonnent



Le pattern observer/observable

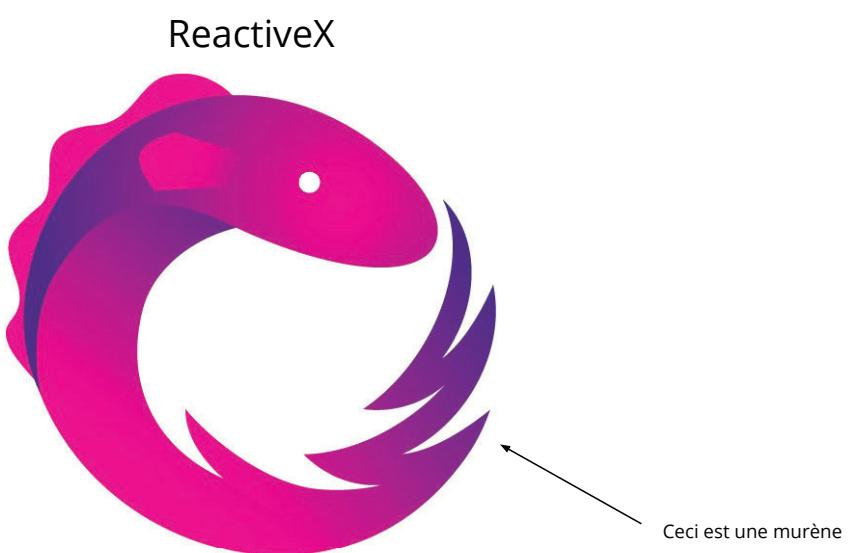
Phase 2 : l'observable notifie



La programmation Reactive

Paradigme de programmation **visant à conserver une cohérence d'ensemble** en **propageant les modifications d'une source réactive** (modification d'une variable, entrée utilisateur, etc.) **aux éléments dépendants de cette source.**

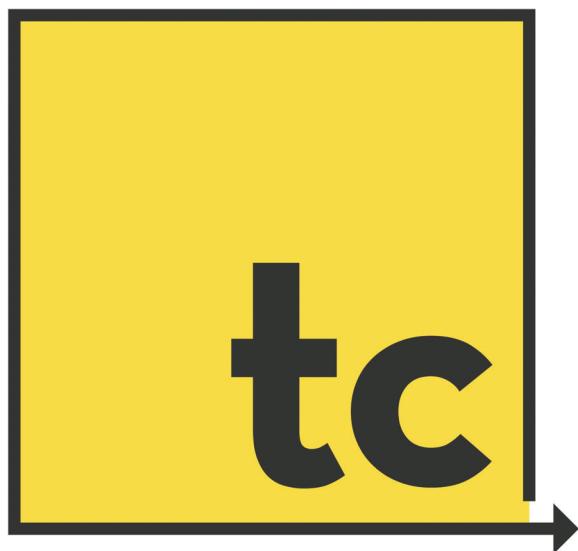
ReactiveX - RxJS



Outils indispensables : Babel, Traceur et Typescript



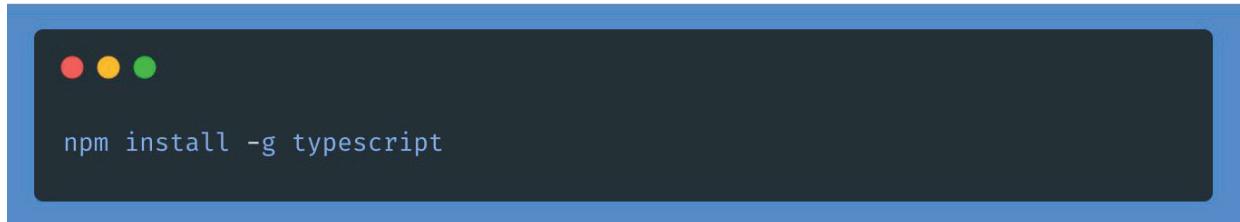
Outils indispensables : Babel, Traceur et Typescript



Outils indispensables : Babel, Traceur et Typescript



TypeScript en détail, configuration



A screenshot of a terminal window with a dark background and a blue header bar. The header bar contains three colored dots (red, yellow, green). The main area of the terminal shows the command `npm install -g typescript` in white text.

TypeScript en détail, configuration

```
function greeter(person) {
    return "Hello, " + person;
}

let user = "Jane User";

const hello = greeter(user);

console.log(hello)
```

TypeScript en détail, configuration

```
tsc greeter.ts
```

TypeScript en détail, configuration

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = "Jane User";

const hello = greeter(user);

console.log(hello)
```

TypeScript en détail, configuration

```
interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person: Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = { firstName: "Jane", lastName: "User" };

const hello = greeter(user);

console.log(hello)
```

TypeScript en détail, configuration



```
class Student {
  fullName: string;
  constructor(
    public firstName: string,
    public middleInitial: string,
    public lastName: string
  ) {
    this.fullName = firstName + " " + middleInitial + " " + lastName;
  }
}

interface Person {
  firstName: string;
  lastName: string;
}

function greeter(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

const hello = greeter(user);

console.log(hello)
```

ES6/2015 approche modulaire

Les applications Javascript sont devenues volumineuses

Nous devons découper nos applications en modules,

MAIS

Les applications Javascript s'exécutent côté serveur (Node) => on importe un fichier

Les applications Javascript s'exécutent côté client (Browser) => on ne peut pas importer de fichier (HTTP)

ES6/2015 approche modulaire

Les Solutions :

CommonJS, AMD, RequireJS

Webpack et Babel

ES7 gestion de l'asynchronicité : await async.

Le callback hell (pyramid of doom)



ES7 gestion de l'asynchronicité : les promises

L'objet Promise est utilisé pour réaliser des traitements de façon asynchrone.

Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.

ES7 gestion de l'asynchronicité : les promises

la méthode then() et catch()

```
function getTodo(id) {
    const p = new Promise((resolve, reject) => {
        setTimeout(() => {
            const todo = { id, title: `title ${id}` }
            resolve(todo);
        }, 1000);
    });
    return p;
}

pTodo
    .then(todo =>{
        console.log(todo)
        return getTodo(todo.id+1)
    })
    .then(todo =>{
        console.log(todo)
        return getTodo(todo.id+1)
    })
    .catch(e => console.log(e))
```

ES7 gestion de l'asynchronicité : les promises

La méthode all()

```
function getTodo(id) {  
  
    const p = new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const todo = { id, title: `title ${id}` }  
            resolve(todo);  
        }, 1000);  
  
    });  
  
    return p;  
}  
  
const pTodo1 = getTodo(1);  
const pTodo2 = getTodo(2);  
const pTodo3 = getTodo(3);  
const pTodo4 = getTodo(4);  
Promise.all([pTodo1,pTodo2,pTodo3,pTodo4]).then(arr => console.log(arr))
```

ES7 gestion de l'asynchronicité : await async.

La déclaration **async function** définit une fonction asynchrone : elle renvoie une promesse comme valeur de retour

```
function getTodo(id) {  
  
    const p = new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const todo = { id, title: `title ${id}` }  
            resolve(todo);  
        }, 1000);  
  
    });  
  
    return p;  
}  
  
async function main() {  
    const todo1 = await getTodo(1);  
  
    console.log(todo1)  
    const todo2 = await getTodo(2);  
    console.log(todo2)  
    const todo3 = await getTodo(3);  
    console.log(todo3)  
    const todo4 = await getTodo(4);  
    console.log(todo4)  
}  
  
main()
```

Travaux pratiques !

Mise en œuvre de l'environnement
avec TypeScript.

Eléments à retenir

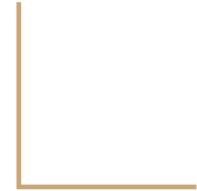
Le développement orienté objet est plus facile avec ES6

Il y a beaucoup d'outils de build mais Webpack est le plus adapté au dev Web

L'asynchronicité est facilité grâce aux promises et aux async/await

TypeScript est un sur-ensemble de ES6, il ajoute les types

Migrer d'AngularJS 1.x à Angular



Comparaison et "topographie" des concepts

Les concepts AngularJS

Controllers

Templates

Components

Directives

Services

Les concepts Angular

Non

Non

Components + "Controller" + Template

Directives

Services

Préparer la migration

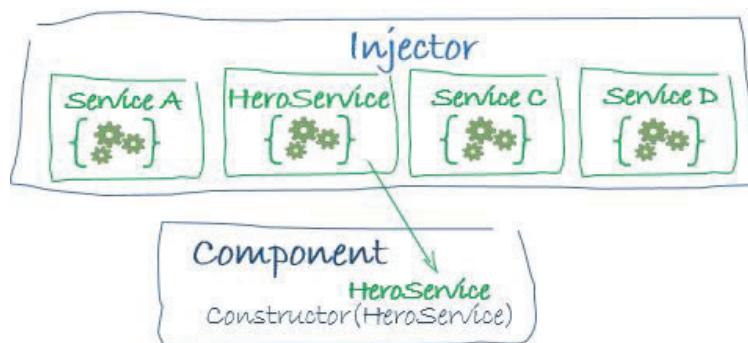
Appliquez les bonnes pratiques AngularJS (John Papa)

Utilisez un module loader

Migrer vers Typescript (*.js vers *.ts)

Utilisez la directive component dans tout votre projet

Principe de l'injection de dépendance



Démo !

Migrer d'AngularJS 1.x à Angular

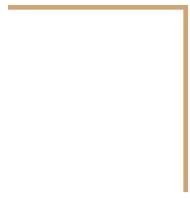
Eléments à retenir

Une application AngularJS doit respecter le coding-style guide pour faciliter la migration.

Conversion des fichiers .js en .ts

Utilisez la library ngUpgrade

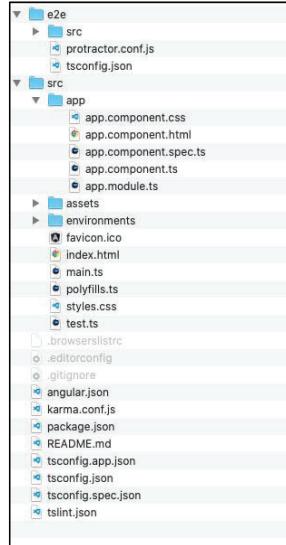
L'utilitaire ng ou @angular/cli



Utilisation de l'utilitaire en ligne de commande

```
● ● ●  
npm install -g @angular/cli  
ng new my-dream-app  
cd my-dream-app  
ng serve
```

Scaffold un projet : anatomie et dépendance



Configuration et commandes clés

Le fichier angular.json

Les commandes ng

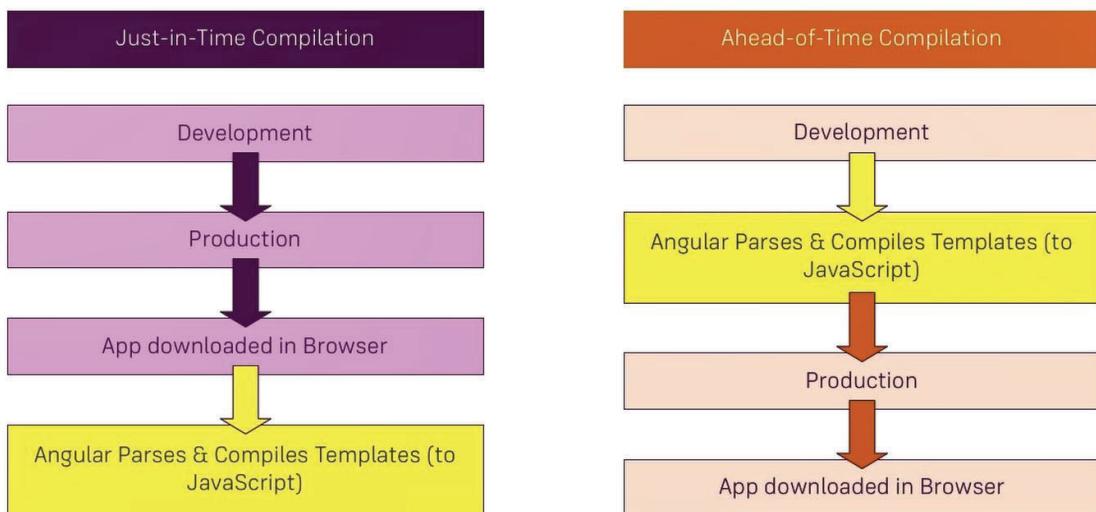
```
fleur@imac-de-fred-1:~/local_dev/formations
~/local_dev/formations » ng help
Available Commands:
  add Adds support for an external library to your project.
  analytics Configures the gathering of Angular CLI usage metrics. See https://angular.io/cli/usage-analytics-gathering.
  build (b) Compiles an Angular app into an output directory named dist/ at the given output path. Must be executed from within a workspace directory.
  deploy Invokes the deploy builder for a specified project or for the default project in the workspace.
  config Retrieves or sets Angular configuration values in the angular.json file for the workspace.
  doc (d) Opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.
  e2e (e) Builds and serves an Angular app, then runs end-to-end tests using Protractor.
  generate (g) Generates and/or modifies files based on a schematic.
  help Lists available commands and their short descriptions.
  lint (l) Runs linting tools on Angular app code in a given project folder.
  new (n) Creates a new workspace and an initial Angular app.
  run Runs an Architect target with an optional custom builder configuration defined in your project.
  serve (s) Builds and serves your app, rebuilding on file changes.
  test (t) Runs unit tests in a project.
  update Updates your application and its dependencies. See https://update.angular.io/
  version (v) Outputs Angular CLI version.
  xi18n (i18n-extract) Extracts i18n messages from source code.

For more detailed help run "ng [command name] --help"
~/local_dev/formations »
```

Lancer un server de dév et builder pour la prod

```
// Lancer un serveur de développement  
ng serve  
// builder une application avec information de debug  
  
ng build  
  
// builder une application pour la prod  
ng build --prod
```

Compilation Ahead of Time. La notion de "Tree Shaking"



Gestion des modules : bonnes pratiques

John Papa et le “coding style guide”: <https://angular.io/guide/styleguide>

Le nom des modules

LIFT = Locate, Identify, Flat, Try to be DRY

Travaux pratiques !

Structurer, "scaffolder" un projet
d'application

Eléments à retenir

il faut maîtriser la commande ng !

Définition de composants

Comprendre les Web Components

Découpage en components

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Comprendre les Web Components : les specs

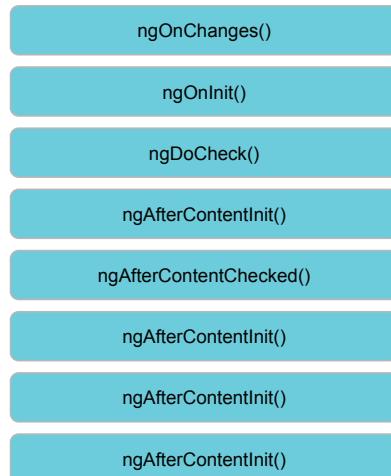
Les composants Web consistent en 3 technologies qui peuvent être utilisées ensemble pour créer des éléments réutilisables, encapsulés, versatiles et sans risquer une collision avec d'autre morceaux de code.

Custom Elements: pour créer et enregistrer de nouveaux éléments HTML et les faire reconnaître par le navigateur.

HTML Templates: squelette pour créer des éléments HTML instanciables.

Shadow DOM: permet d'encapsuler le JavaScript et le CSS des éléments.

Cycle de vie dans l'application



Angular Compiler : Change Detection

Zone.js et NgZone

Le moteur ivy

Syntaxe des templates - Interpolation



Syntaxe des templates - Bindings

Type	Syntax	Category
Interpolation Property Attribute Class Style	<pre>{} [target]="expression" bind-target="expression"</pre>	One-way from data source to view target
Event	<pre>(target)="statement" on-target="statement"</pre>	One-way from view target to data source
Two-way	<pre>[[(target)]="expression" bind-on-target="expression"</pre>	Two-way

Syntaxe des templates - Template reference variables



```
<input #phone placeholder="phone number" />
```

Directives Structurelles : ngIf, ngFor, ngSwitch...

Les directives structurelles construisent ou modifient la **structure du DOM** en ajoutant, supprimant ou manipulant les éléments de l'arbre.

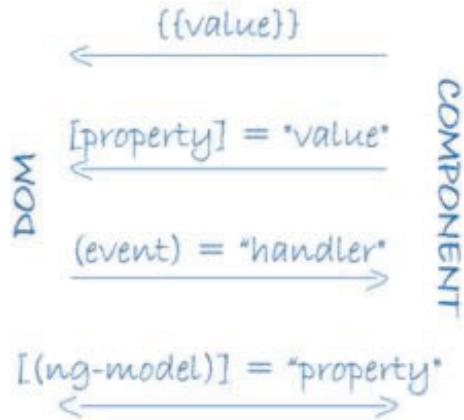
Définition syntaxique, le symbole (*)

```
● ● ●  
<div *ngIf="hero" class="name">{{hero.name}}</div>  
  
<ng-template [ngIf]="hero">  
  <div class="name">{{hero.name}}</div>  
</ng-template>
```

Variables de Template

```
● ● ●  
<input #phone placeholder="phone number" />
```

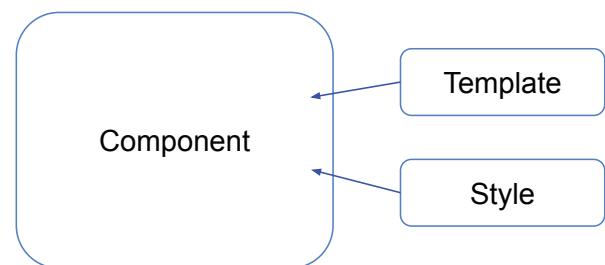
Data bindings



Classe de composants et directives de configuration

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

export class AppComponent {
  title = 'my-dream-app';
}
```

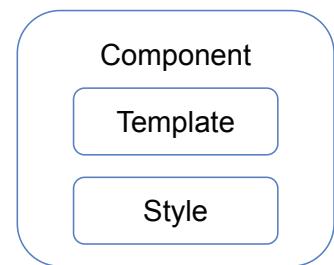


Classe de composants et directives de configuration

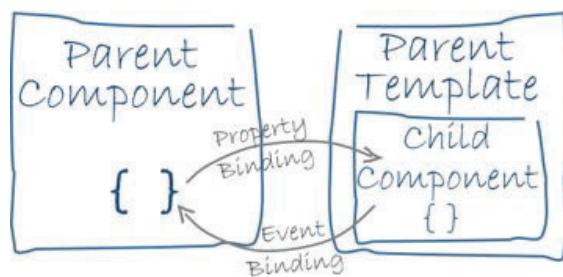
```
@Component({
  selector: 'app-root',
  template: '<h1>{{title}}</h1>',
  styles: ['h1 { font-weight: normal; }']

})

export class AppComponent {
  title = 'my-dream-app';
}
```



Événements utilisateur et événements logiques personnalisés



Événements utilisateur et événements logiques personnalisés

Parent

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

@Component({
  selector: 'app-vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <app-voter *ngFor="let voter of voters"
      [name]="voter"
      (voted)="onVoted($event)">
    </app-voter>
  `
})
export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Narco', 'Celeritas', 'Bombasto'];

  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}
```

Enfant

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

@Component({
  selector: 'app-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)" [disabled]="didVote">Agree</button>
    <button (click)="vote(false)" [disabled]="didVote">Disagree</button>
  `
})
export class VoterComponent {
  @Input() name: string;
  @Output() voted = new EventEmitter<boolean>();
  didVote = false;

  vote(agreed: boolean) {
    this.voted.emit(agreed);
    this.didVote = true;
  }
}
```

Travaux pratiques !

Création de composants

Eléments à retenir

La syntaxe des templates est simple

Il y a 3 types de bindings :

property binding: []

event binding: ()

Two-way binding: [()] (banana in a box)

Classifications des composants applicatifs

Les Modules

Les modules (NgModule) sont des conteneurs.

Ils permettent de stocker au même endroit des components, services, etc... relatifs au même domaine applicatif

Pour organiser son code il existe 2 types de modules :

Les Shared Modules

Les Feature Modules

Les Shared Module

Vous pouvez regrouper les éléments (components,..) les plus courants dans un seul module, puis importer ce module là où vous en avez besoin

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CustomerComponent } from './customer.component';
import { NewItemDirective } from './new-item.directive';
import { OrdersPipe } from './orders.pipe';

@NgModule({
  imports:      [ CommonModule ],
  declarations: [ CustomerComponent, NewItemDirective, OrdersPipe ],
  exports:      [ CustomerComponent, NewItemDirective, OrdersPipe,
                  CommonModule, FormsModule ]
})
export class SharedModule { }
```

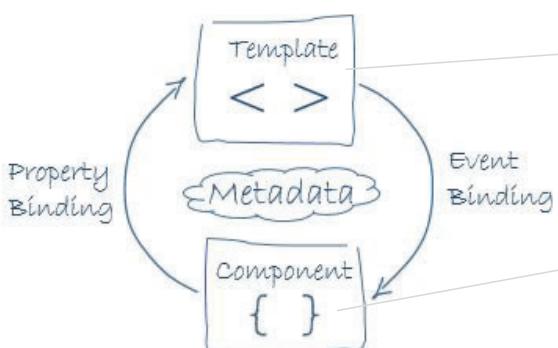
Les Feature modules

Un feature module fournit un ensemble cohérent de fonctionnalités axées sur un besoin applicatif spécifique.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';// import the new component
import { CustomerDashboardComponent } from './customer-dashboard/customer-dashboard.component';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CustomerDashboardComponent
  ],
})
```

Les Components

Un Component contrôle une partie d'écran appelé view.



```
@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers: [ HeroService ]
})

export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

Les Directives

Il existe trois types de directives :

Les Components : directive avec un template

Les Directives Structurelles : modifient la disposition du DOM en ajoutant et en supprimant des éléments du DOM

Les Directives d'Attributs : modifient l'apparence ou le comportement d'un élément, d'un composant ou d'une autre directive.

Les Pipes

Les pipes permettent de transformer des données dans le template

```
<p>Today is {{today | date}}</p>

<p>The date is {{today | date:'fullDate'}}</p>

<p>The time is {{today | date:'shortTime'}}</p>
```

Les Guards

Les Guards sont associés aux routes.

Ils permettent de contrôler l'accès à une **route** (autorisation) ou le départ depuis une **route** (enregistrement ou publication obligatoire avant départ).

Les Services et l'injection de dépendance

Un service est une vaste catégorie regroupant tout élément fonctionnel dont une application a besoin

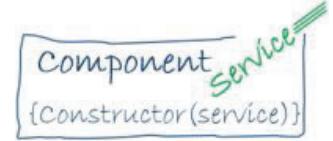
C'est une simple classe déclarée **Injectable**

Il s'utilise en le déclarant en paramètre (dépendance) du constructeur d'un autre élément Angular (Component, Directive, autre Service,..)

Principe de l'injection de dépendances

L'injection de dépendance (DI) est un pattern important.

Angular possède sa propre implémentation.



Les dépendances sont des services ou des objets dont une classe a besoin pour remplir sa fonction.

L'injection de dépendance est un modèle de codage dans lequel une classe demande des dépendances à des sources externes plutôt que de les créer elle-même, Angular lui fournit via un Injector.

<https://angular.io/guide/dependency-injection>

Création de services injectables. Classification des services

```
1 import { Component, OnInit } from '@angular/core';
2 import { Observable } from 'rxjs';
3 import { User } from '../user';
4 import { UserService } from '../user.service';
5
6 @Component({
7   selector: 'app-user-list',
8   templateUrl: './user-list.component.html',
9   styleUrls: ['./user-list.component.css']
10 })
11 export class UserListComponent implements OnInit {
12
13   list$: Observable<User[]>
14
15   constructor(private userService:UserService) { }
16
17   ngOnInit(): void {
18     this.list$ = this.userService.findAll()
19   }
20
21 }
```

Travaux pratiques !

Création de composants et de directives personnalisées

Eléments à retenir

Les modules organisent le code (root, feature, shared)

Les composants contrôlent des parties de l'écran

Les services se chargent des responsabilités transversales

Les pipes formatent l'affichage

Gestion des formulaires, "Routing" et requête HTTP

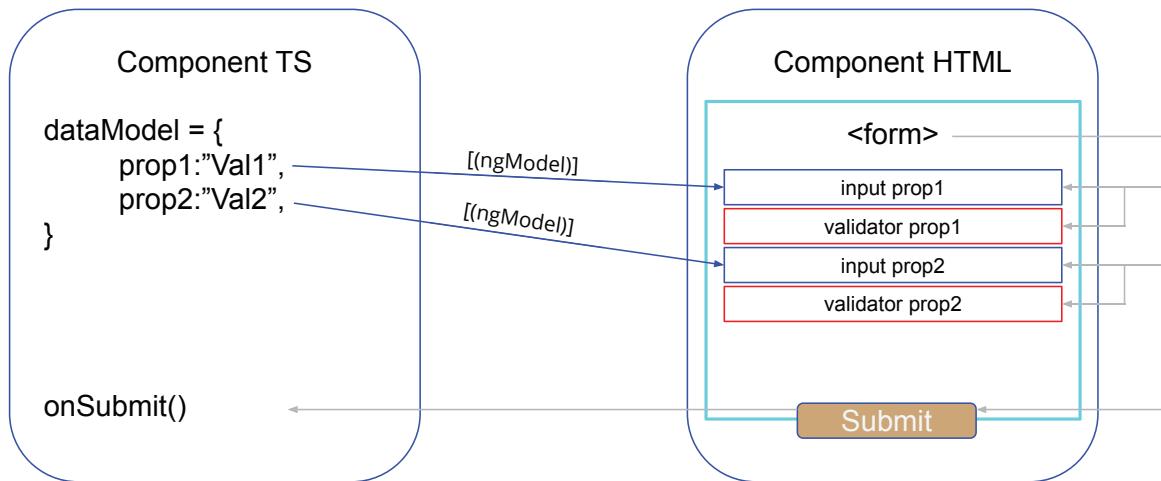
Template Driven Forms vs Reactive Forms

Les **Reactive forms** offrent un accès direct et explicite au modèle objet des formulaires sous-jacents.

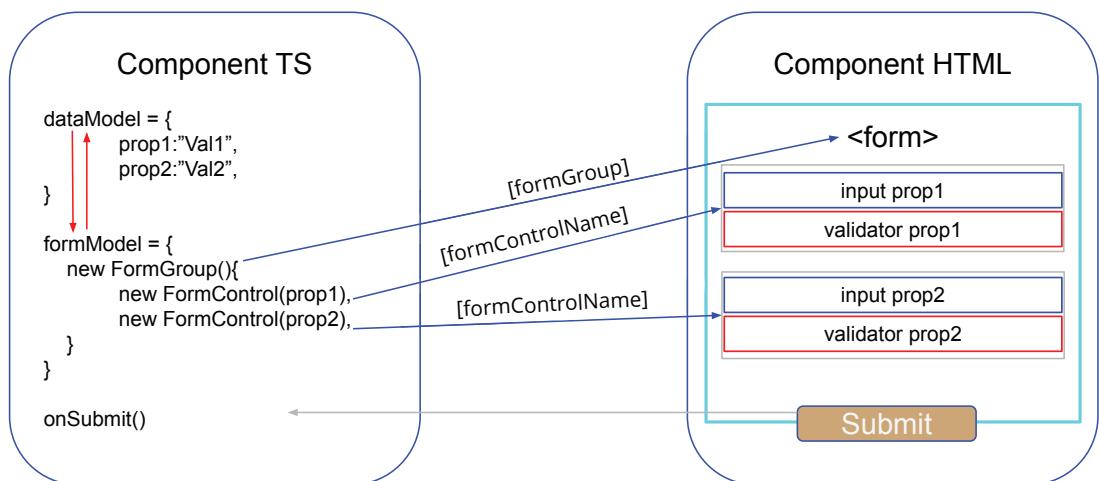
Par rapport aux formulaires basés sur des modèles, ils sont plus robustes : ils sont plus évolutifs, réutilisables et testables.

Les **Template-driven forms** basés sur des modèles s'appuient sur des directives dans le modèle pour créer et manipuler le modèle d'objet sous-jacent. Ils sont faciles à ajouter à une application, mais ils ne sont pas aussi évolutifs que les **Reactive forms**.

Template Driven Form



Reactive Forms



Liaison de données via HTTP

La récupération de donnée se fait via le service HttpClient proposé par le HttpClientModule.

Les méthodes du service HttpClient renvoie des Observables

Programmation Réactive avec RxJS

La programmation réactive est un paradigme de programmation visant à conserver une cohérence d'ensemble en **propageant les modifications d'une source** réactive (modification d'une variable, entrée utilisateur, etc.) **aux éléments dépendants** de cette source.

ReactiveX combine le **pattern Observer** avec le **pattern Iterator** et la **programmation fonctionnelle** avec **les collections** pour gérer des séquences d'événements.

Concepts essentiels de RxJS

Observable: représente une collection invoquable de valeurs ou d'événements.

Observateur: est une collection de callback qui écoute les valeurs fournies par l'Observable.

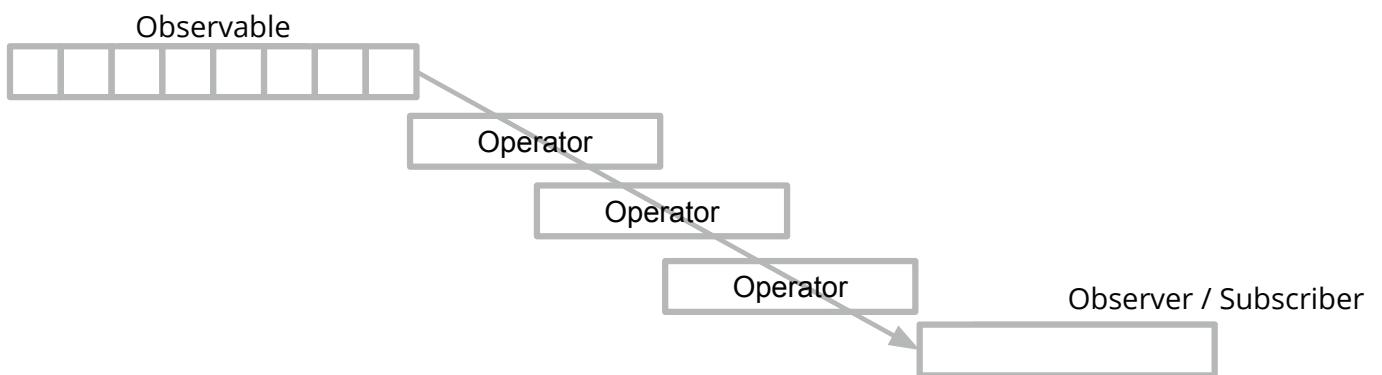
Subscriber: représente l'exécution d'un observable.

Operators: sont des fonctions pures qui permettent un style de programmation fonctionnel pour traiter les collections avec des opérations comme la cartographie, le filtrage, la concaténation, la réduction, etc.

Sujet: est l'équivalent d'un EventEmitter, et le seul moyen de diffuser une valeur ou un événement à plusieurs Observateurs (multicasting).

Schedulers: sont des dispatchers centralisés pour contrôler la concurrence, nous permettant de coordonner lorsque le calcul se produit sur, par exemple, setTimeout ou requestAnimationFrame ou autres.

Concepts essentiels de RxJS



Gestion et configuration des échanges HTTP

Le fichier environment

Environnement : gestion des conf de build - dev

```
// environment.ts
// This file can be replaced during build by using the `fileReplacements` array.
// `ng build --prod` replaces `environment.ts` with `environment.prod.ts`.
// The list of file replacements can be found in `angular.json`.

export const environment = {
  production: false
};
```

Environnement : gestion des conf de build - prod



```
// environment.prod.ts
export const environment = {
  production: true
};
```

Création de routes



```
ng new routing-app --routing
```

Création de routes Shared Module

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router'; // CLI imports router

// sets up routes constant where you define your routes
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
];

// configures NgModule imports and exports
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Ciblage, "router-outlet" événements de routage

```
<h1>{{ title }} app is running!</h1>
<router-outlet></router-outlet>
```

Intercepter les paramètres de routage et wildcard

```
● ● ●

<h1>{{ title }} app is running!</h1>
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router'; // CLI imports router

const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '', redirectTo: '/first-component', pathMatch: 'full' }, // redirect to `first-component`
  { path: '**', component: PageNotFoundComponent }, // Wildcard route for a 404 page
];

// configures NgModule imports and exports
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Configuration des "Guard" pour l'initialisation des routes

```
● ● ●

ng generate guard your-guard
-- 
export class YourGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    // your logic goes here
  }
}
-- 
//Routing module (excerpt)
{
  path: '/your-path',
  component: YourComponent,
  canActivate: [YourGuard],
}
```

Travaux pratiques !

Mise en œuvre des cycles de validation de formulaire.
Consommation d'une API REST.

Eléments à retenir

La programmation réactive permet de réagir aux changement ayant lieu dans l'application (data, events,...)

Il a 2 types de formulaires :

- les reactive forms piloté par le code
- les templates forms piloté par le template

Les requêtes HTTP sont observables

la configuration des urls se fait dans les fichiers "environment"

Les routes sont déclarées par un mapping entre URL et Component

Tests unitaires. Bonnes pratiques et outils

Test Driven Development

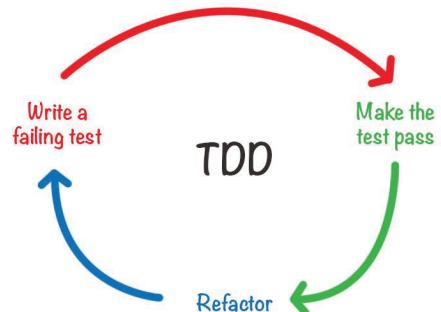
Test-Driven Development (TDD): une méthode de développement de logiciel, qui consiste à concevoir un logiciel par petits pas, de façon itérative et incrémentale, en écrivant chaque test avant d'écrire le code source et en remaniant le code continuellement.

Red: Rédiger un test : Le code correspondant n'existe pas encore donc ce test va échouer et donc rester rouge...

Green: Écrire le code qui permet de faire passer le test, peu importe la qualité du code (répétitions,...)

Refactor: Refactoriser le code et les tests.

Améliorer le code. Supprimer les duplications, optimisations, single-responsability,...



Ecrire des tests avec Jasmine



Exécuter des tests avec Karma



UnitTest avec Jasmine

```
describe("A suite with some shared setup", function() {
  var foo = 0;

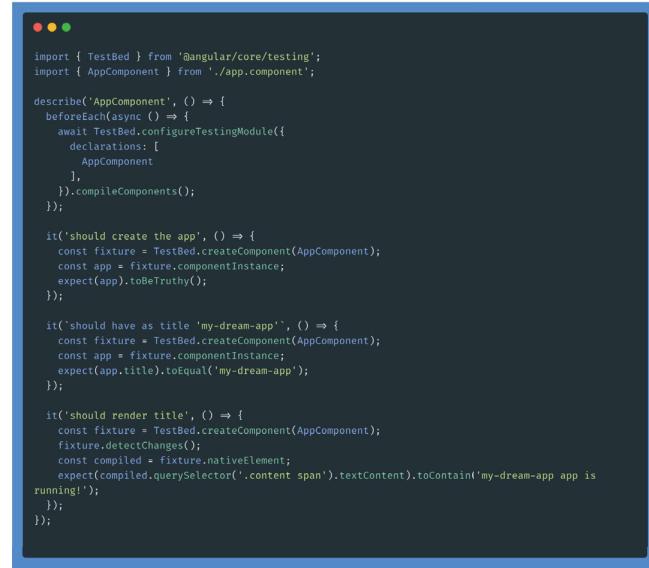
  beforeEach(function() {
    foo += 1;
  });

  afterEach(function() {
    foo = 0;
  });

  beforeAll(function() {
    foo = 1;
  });

  afterAll(function() {
    foo = 0;
  });
});
```

UnitTest pour Angular avec Jasmine



```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });

  it(`should have as title 'my-dream-app'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app.title).toEqual('my-dream-app');
  });

  it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('.content span').textContent).toContain('my-dream-app app is
running!');
  });
});
```

Ecrire des tests d'intégration avec protractor

Le projet Selenium

Les tests e2e (End-to-End)

Ecrire des tests d'intégration avec protractor

```
● ● ●

import { AppPage } from './app.po';
import { browser, logging } from 'protractor';

describe('workspace-project App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getTitleText()).toEqual('my-dream-app app is running!');
  });

  afterEach(async () => {
    // Assert that there are no errors emitted from the browser
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });
});
```

Angular "Coding guide Style"

John Papa et le coding style guide: <https://angular.io/guide/styleguide>

Travaux pratiques !

Développement d'une application à partir de tests unitaires.
Mise en œuvre du Test Driven Development.

Eléments à retenir

Il faut écrire les tests avant d'écrire le code

Grâce à la commande ng, un test est généré pour chaque élément créé

TDD pour les tests unitaires avec Jasmine et Karma

BDD pour les tests fonctionnels avec Protractor

Conclusion