ANGULAR(2) (BUILD: 6.0 / 3 MAY 2018)

BUILD APPS

Charles JABBOUR GEDEON

SOMMAIRE

INTRODUCTION

OUTILS & IDE

- VISUAL STUDIO CODE
- CHROME
- AUGURY

NPM

- The Node Package Manager
- Besoin de node.js
- https://www.npmjs.com/get-npm
- npm install npm@latest -g
  - Update now !

WEBPACK

- Pour packager votre application… en entier !
- https://webpack.js.org/
- https://webpack.js.org/guides/getting-started/

ANGULAR CLI

- Side project lancé avec angular 2
- Permet de gérer une application angular 2+
  - Composant, services, etc…
- https://cli.angular.io/
- Test !
  - ng new myApp & serve

TYPESCRIPT & ES6

TYPESCRIPT

- Typescript est un langage entre Java et Javascript.
  - Java: Typecheck à la compilation, Objet
  - Javascript: Asynchrone

- Typescript est un projet Microsoft. Google à choisit de s'en servir pour developper Angular(2+).

TYPAGE DES VARIABLES

```
let variable: type;
```

```
const counter: number = 0;
const counterName: string = ‘a string’;
```

Type facultatif lorsque le compilateur peut inférer

TYPAGE DES VARIABLES

Type custom

```
const varName: MyPersonalType = new MyPersonalType();
```

Type générique !!!

```
const ponies: Array< MyPersonalType > = [new MyPersonalType()];
```

Lors d'un type inconnu, le type *any* (même si… ce n'est pas le but !) est TRES utile

```
let changing: any = 2;
changing = true;
```

TYPAGE DES VARIABLES

Valeurs énumérées

```
enum MyStatus {Ready, Started, Done}

    const MyType = new MyType();
    race.status = MyType.Ready;
```

```
enum MyStatus {Ready = 1, Started, Done}
```

Utilisation dans angular:

Pour plus tard

Tableau:

```
    this.enums = Object.keys(MyEnum);
```

Utilisation dans un select:

```
<option *ngFor="let s of enums; let i of index"  value="MyEnum[i]">{{MyEnum[i]}}</option>
```

TYPAGE DES FONCTIONS

Retour de fonctions & arguments optionnels

```
function startRace(race?: Race): Race {
  race.status = RaceStatus.Started;
  return race;
}
```

Passage d'argument par "forme"

```
function addPointsToScore(player: { score: number; }, points: number): void {
  player.score += points;
}
```

CLASSES ET INTERFACES

CLASSES

Declaration - implementation

```
class Car implements Movable {
  run(meters) {
    logger.log(`car move ${meters}m`);
  }
}
```

INTERFACES

Declaration

```
Interface Movable {
  run(meters: number): void;
}
```

Utilisation

```
function startRunning(car: Movable): void {
      car.run(10);
};

const myCar = {
      run: (meters) => logger.log(`car runs ${meters}m`)
};


startRunning(car);
```

OBJECT ORIENTED: TYPESCRIPT

```
class MyClass
```

```
class MyClass {
  constructor(public name: string, private speed: number) {
  }
  run() {
  }
}
```

```
interface IRunnable
```

```
class MyClass {
  public name: string;
  private speed: number;

  constructor(name: string, speed: number) {
    this.name = name;
    this.speed = speed;
  }

  run() {
  }
}
```

ARROW FUNCTIONS

```
getUser(login)
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

```
(param1, param2) => {

  //Body

  return ...

}
```

Pas de nouveau scope pour this.

COMPOSANTS

INTRO

- Custom elements ("éléments personnalisés”)
- Shadow DOM ("DOM de l’ombre”)
- Template
- HTML imports

INTRO

```
// Creation d'un element
var Component = document.registerElement('mon-composant');
// Insertion dans le body depuis JS
document.body.appendChild(new Component());
```

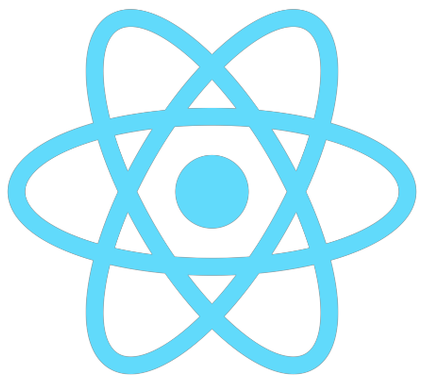
INTRO

```
// Utilisation
var ComponentProto = Object.create(HTMLElement.prototype);
// callback sur la creation d'un element.
ComponentProto.createdCallback = function() {
  this.innerHTML = '<h1> my main title </h1>';
};

// Instance
var Component = document.registerElement('ns-', {prototype: ComponentProto});
// insertion avec l'api standard
document.body.appendChild(new Component());
```

INTRO

- Angular est un framework orienté composant
- Les composants sont organisés de façon arborescente



UNE PREMIERE APPLICATION

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

@Component({
  selector: ‘my-elem',
  templateUrl: ‘mon-composant.html’
})
export class MyComposant {


  constructor() {}
}
```

```
<div>
  <h2>{{ race.name }}</h2>
  <div>{{ race.status }}</div>
  <div *ngFor="let elem of obj.elements”>
    <my-elem [elem]=“elem"></ns-elem>
  </div>
</div>
```

UNE PREMIERE APPLICATION

- Avec angular 2, il est possible d'importer très facilement des composants.
- Travail en équipe !!
- La puissance des webcomponents, la DI en plus !
- Two way binding

UNE PREMIERE APPLICATION

- Pratique:
  - Bootstrap d’une app angular à la main

LETS GO

UNE PREMIERE APPLICATION - ANGULAR CLI

- Generation de l'application
- Generation d'un composant
- Demarrage

UNE PREMIERE APPLICATION - ANGULAR CLI

- class
- component
- directive
- enum
- guard
- interface
- module
- pipe
- service
- application
- library
- universal

Une commande pour chaque composant

ex: ng generate component User

ANGULAR 6

AJOUT SUR LES DÉCLARATIONS

- Depuis angular 6, Angular CLI va déclarer les services, pipes etc… au niveau de l'annotation et plus au niveau du module racine (ou sous module)

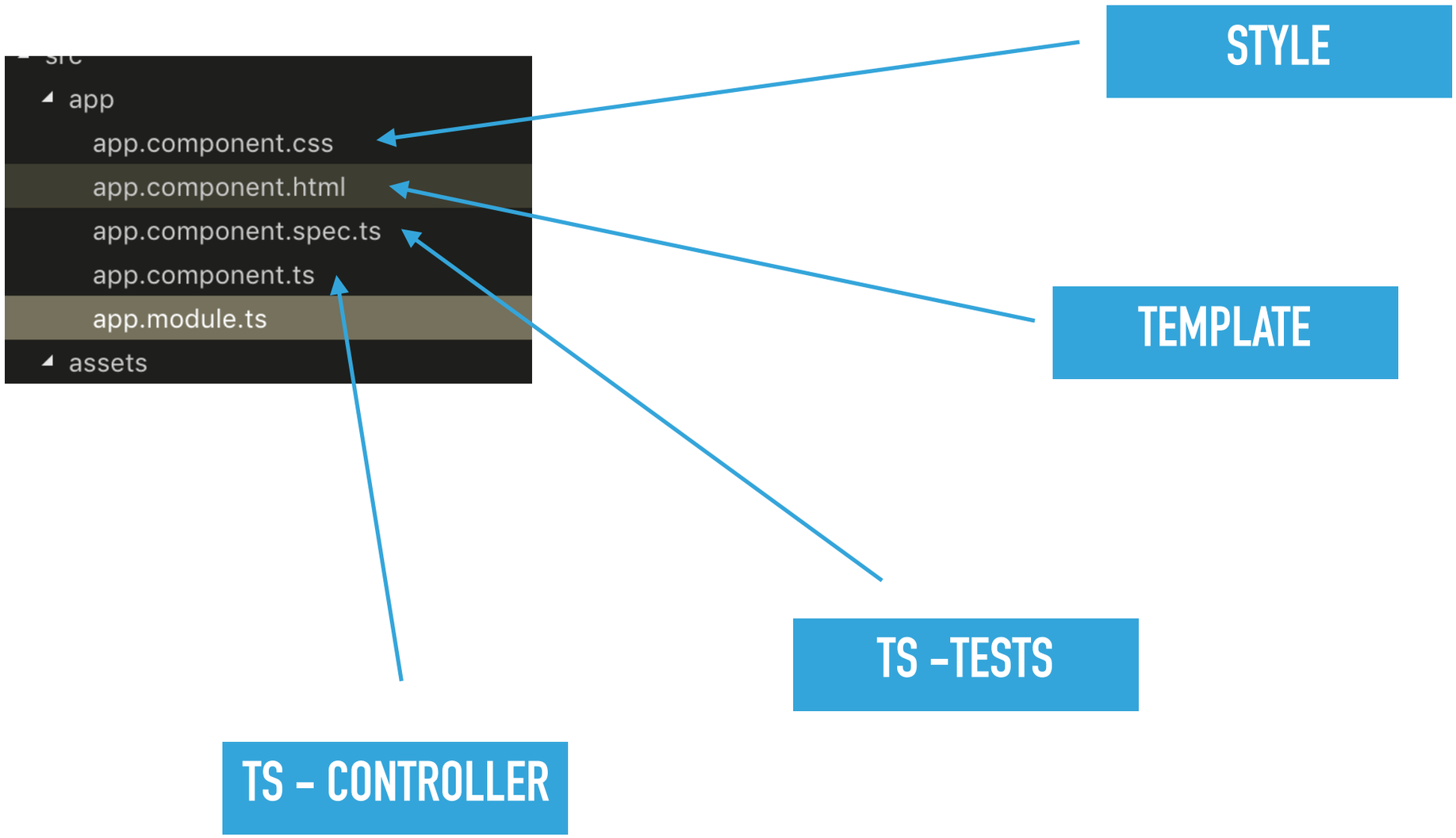
COMPOSANTS - SYNTAXE

COMPOSANTS - SYNTAXE

- A chaque composant correspond un template.
- Ils sont liés, et utilise le shadow dom

Voyons quelques elements de syntaxes

COMPOSANTS - SYNTAXE



Le composant permet d'encapsuler toute la logique de présentation autour, souvent, d'une classe métier

INTERPOLATION

{{ c'est tout }}

Si l'interpolation échoue, on affiche rien. Pas d'erreur...

INTERPOLATION

{{ c'est tout . Comme d'habitude }}

Erreur lors du chargement si l'interpolation échoue

{{ c'est tout ?.oupas}}

Si la propriété doit être chargé depuis le serveur, on utilisera "?"

Le safe navigation operator

INTERPOLATION

```
<p [textContent]="user.name"></p>
```

=

```
<p> {{user.name}} </p>
```

EVENT

- Plus de syntaxes spécifiques à apprendre.
- https://developer.mozilla.org/fr/docs/Web/Events

Tout est là !

EVENT

Un objet ‘event’ peut être passé directement à une instruction

```
onButtonClick($event)
```

Il sera du type de l’événement déclenché

EVENT

- Pratique
  - Ajout d’un objet dans une liste lors d’un click. Affichage.

VARIABLES

En plus des variables public de l'instance du composant
on pourra déclarer une variable directement dans le
template

```
<input type="text" #name>
```

Et être utilisé comme cela:

```
<button (click)="name.focus()">Focus the input</button>
```

DIRECTIVES STRUCTURELLES

*ngIf

*ngFor

[ngSwitch]

*ngSwitchCase

DIRECTIVES STANDARDS

*NgStyle

[ngStyle]="{fontWeight: fontWeight, color: color}"

*NgClass

<div [ngClass]="{'awesome-div': isAnAwesomeDiv(), 'colored-div': isAColoredDiv()}"

SERVICES

LES SERVICES BUILT IN

Deux petits services build-in Angular

Title

Meta

Charles JABBOUR GEDEON

LES SERVICES, CREATION

On peut surtout, écrire les notre.

Les services, sont des singletons

Il suffit... de déclarer une classe. Un service peut lui-même être injectable avec

@Injectable

DI

L'annotation @Injectable()
sur la classe… qui peut être injecté

Ajout dans le module racine.
providers

{ provide: PasteService, useClass: PasteServices },

{ provide: ApiService, useClass: ApiService }

DI - SUITE

useFactory

Pratique pour le mocking. Et dans pleins d'autres cas !

useValue

Plus élégant qu'une constante.

LES PIPES

Les pipes ont le même intérêt que ceux du bash.
Ils transforment un flux

Formatage: json,, uppercase, lowercase, titlecase

Découpage array: slice

Internationalisation: number, percent, currency, date

Chargement asynchrone, observable): async

LES PIPES

Creation d'un nouveau pipe avec PipeTransform

Override de la methode: transform(value, args)

@Pipe

LES PIPES

Creation d'un nouveau pipe avec PipeTransform

Override de la methode: transform(value, args)

@Pipe

LES DIRECTIVES ET COMPOSANTS

DECLARATION DIRECTIVES

Les directives sont des “briques” minimaliste permettant d’ajouter des comportements.
Tout ce qui est vrai pour les directives est vrai pour les composants

Les composants héritent des directives.

@Directive

```
@Directive({
  selector: ‘[doNothing]'
})
export class DoNothingDirective {

constructor() {
console.log('Do nothing directive');

} }
```

Exemple de selecteur:

selector: div.[doNothing]’

DIRECTIVES

Passer des données aux directives/composants enfants

```
inputs: ['text: logText']
```

```
@input(logText)
text: string
```

DIRECTIVES

Passer des données aux directives/composants parents

```
@Output() itemSelected = new EventEmitter<MyType>();
```

CYCLE DE VIE

- ngOnInit sera appelée une seule fois après le premier changement
- ngOnDestroy est appelée quand le composant est supprimé.

CYCLE DE VIE

- ngAfterViewInit est appelée quand tous les bindings du template ont été vérifiés pour la première fois.
- ngAfterViewChecked est appelée quand tous les bindings du template ont été vérifiés, même s'ils n'ont pas changé. Cela peut être utile si on a un composant ou une directive attend qu'un élément particulier soit disponible pour en faire quelque chose.
- ngAfterContentInit est appelée quand tous les contenus projetés du composant ont été vérifiés pour la première fois. (transclusion)
- ngAfterContentChecked est appelée quand tous les contenus projetés du composant ont été vérifiés, même s'ils n'ont pas changé. (transclusion)

CYCLE DE VIE

```
class CycleCompTestComponent implements
OnInit,
OnDestroy,
AfterViewInit,
AfterViewChecked
```

```
  ngOnInit() {
  }
  ngOnDestroy(): void {
    console.log('ngOnDestroy');
  }
  ngAfterViewInit(): void {
    console.log('ngAfterViewInit');
  }
  ngAfterViewChecked(): void {
    console.log('ngAfterViewChecked');
  }
```

DETECTION DES MODIFICATIONS

- ngOnChanges: Est la méthode appelée quand la valeur d'une propriété (via @Input) est modifiée.

- ngDoCheck: Est la méthode appelée quand une valeur d'un composant est modifié.

Cette méthode est déclenchée également lors d'un changement sur la référence.

Dans la plupart des cas, nous n'avons pas besoin de nous servir de ces fonctions.

C'est utile dans un cas
ou une variable change énormément et que l'on veut implementer son propre mécanisme de détection.
(en couplage avec OnPush)

CHANGE DETECTION STRATEGY

OnPush

On push provoque la désactivation du ChangeDetector d'angular et fait en sorte que le dom soit ré-évalué dans les cas suivants:

- Lors d'un changement de la référence d'un objet passé via @Input (pas seulement un de ses attributs)
- Lors de la génération d'un événement qui provoque le rechargement du DOM au niveau de l'enfant (click, keyup, subscribe...)

DETECTION DES MODIFICATIONS

MARKFORCHECK

Utilisation avec ChangeDetectionStrategy.OnPush afin de ré évaluer le dom même lors d'un simple changement d'attribut d'un objet

DETECTCHANGES()

Lorsque le ChangeDetector est détaché, les valeurs affichés ne correspondent pas aux valeurs dans les objets:
Il est dit "détaché dans deux cas":

Apres un appel à detach()

ou

Apres qu'un changement ait eu lieu en dehors du monde Angular.

FORMULAIRES

DEUX MÉTHODES

- Template driven
- Code driven

CODE DRIVEN – FORM CONTROL

- valid : si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.
- invalid : si le champ est invalide, au regard des contraintes et des validations qui lui sont appliquées.
- errors : un objet contenant les erreurs du champ.
- dirty : false jusqu'à ce que l'utilisateur modifie la valeur du champ.
- pristine : l'opposé de dirty.
- touched : false jusqu'à ce que l'utilisateur soit entré dans le champ.
- untouched : l'opposé de touched.
- value : la valeur du champ.
- valueChanges : un Observable qui émet à chaque changement sur le champ.

CODE DRIVEN – FORM CONTROL

```
const password = new FormControl('charles');
console.log(password.value); // logs “charles"
```

CODE DRIVEN – FORM GROUP

- valid : si tous les champs sont valides, alors le groupe est valide.
- invalid : si l'un des champs est invalide, alors le groupe est invalide.
- errors : un objet contenant les erreurs du groupe, ou null si le groupe est entièrement valide. Chaque erreur en constitue la clé, et la valeur associée est un tableau contenant chaque contrôle affecté par cette erreur.
- dirty : false jusqu'à ce qu'un des contrôles devienne "dirty".
- pristine : l'opposé de dirty.
- touched : false jusqu'à ce qu'un des contrôles devienne "touched".
- untouched : l'opposé de touched.
- value : la valeur du groupe. Pour être plus précis, c'est un objet dont les clé/valeurs sont les contrôles et leur valeur respective.
- valueChanges : un Observable qui émet à chaque changement sur un contrôle du groupe.

CODE DRIVEN – FORM GROUP

```
const form = new FormGroup({
  username: new FormControl(‘charles'),
  password: new FormControl()
});
console.log(form.dirty); // logs false until the user enters a value
console.log(form.value); // logs Object {username: “charles", password: null}
console.log(form.get('username')); // logs the Control
```

TEMPLATE DRIVEN

```
<form (ngSubmit)=“submit()">
    <button type="submit">Register</button>
  </form>
```

```
<h2>Sign up</h2>
<form (ngSubmit)="submit(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel>
  </div>
  <button type="submit">Register</button>
</form>
```

```
[(ngModel)]="user.password"
```

```
[ngModel]="user.username" (ngModelChange)="user.username = $event">
```

CODE DRIVEN

- Declaration artisanale... ou

```
constructor(fb: FormBuilder) {
}
```

CODE DRIVEN

- Ex

```
export class RegisterFormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control(''),
      password: fb.control('')
    });
  }

  register() {
    console.log(this.userForm.value);
  }
```

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
  </div>
  <button type="submit">Register</button>
</form>
```

CODE DRIVEN

- Validators.required pour vérifier qu'un contrôle n'est pas vide ;
- Validators.minLength(n) pour s'assurer que la valeur entrée a au moins n caractères ;
- Validators.maxLength(n) pour s'assurer que la valeur entrée a au plus n caractères ;
- Validators.email() (disponible depuis la version 4.0) pour s'assurer que la valeur entrée est une adresse email valide
- Validators.pattern(p) pour s'assurer que la valeur entrée correspond à l'expression régulière p définie.

CODE DRIVEN

- Validators

```
this.userForm = fb.group({
      username: fb.control('', [Validators.required, Validators.minLength(3)]),
      password: fb.control('', Validators.required)
    });
```

Equivalent template driven

```
<label>Username</label><input name="username" ngModel required minlength="3">
```

ERREUR: CODE DRIVEN

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]=“userForm">
  <div>
    <label>Username</label>
      <input formControlName="usernameCtrl">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is required</div>
      <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('minlength')">Username should be 3 </div>
  </div>
  <div>
    <label>Password</label>
      <input type="password" formControlName="passwordCtrl">
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

ERREUR: TEMPLATE DRIVEN

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required #username="ngModel">
    <div *ngIf="username.dirty && username.hasError('required')">Username is required</div>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required #password="ngModel">
    <div *ngIf="password.dirty && password.hasError('required')">Password is required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

REAGIR AU CHANGEMENT

- valueChanges : Observable

```
    this.contentCtrl
    .valueChanges
    .pipe(
      debounceTime(400),
      distinctUntilChanged()
    ).subscribe(newValue => console.log(newValue));
```

REAGIR AU CHANGEMENT – FOCUS – SOUMISSION

Angular 5

Code driven

```
this.usernameCtrl = new FormControl('', Validators.required);
this.passwordCtrl = new FormControl('', {
  validators: Validators.required,
  updateOn: 'blur'
});
```

```
this.userForm = new FormGroup({
  username: this.usernameCtrl,
  password: this.passwordCtrl
}, {
  updateOn: 'blur'
});
```

Template driven

```
  [(ngModel)]="user.username" [ngModelOptions]="{ updateOn: 'blur' }" required>
```

```
<form (ngSubmit)="register()" [ngFormOptions]="{ updateOn: 'blur' }">
```

REAGIR AU CHANGEMENT – FOCUS – SOUMISSION

Code driven

```
static isOldEnough(control: FormControl) {
  const birthDate = new Date(control.value);
  birthDate.setFullYear(birthDate.getFullYear() + 18);
  return birthDate < new Date() ? null : { tooYoung: true };
}
```

Association dans le controller

```
this.birthdateCtrl = fb.control('', [
  Validators.required,
  UserFormComponent.isOldEnough
]);
```

Dans le template

```
<div>
  <label>Date</label><input type="date" formControlName="birthdate">
  <div *ngIf=" birthdateCtrl.dirty && birthdateCtrl.hasError('tooYoung')">You're way too young !!</div>
</div>
```

HTTP

COMPOSANT BUILT-IN

- Simple injection à HTTP
- Prévu pour les tests
- Basé sur… Rx !

COMPOSANT BUILT-IN

- Get / put / post / delete / patch / head / jsonp

```
http.get('/mon/super/service')
  .subscribe((response: Array<Paste>) => {
    console.log(response);
    console.log(response.status); // logs 200

 });
```

COMPOSANT BUILT-IN

Retry

```
raceService.list()
 .retry(3)
 .subscribe(races => {
  this.pastes = pastes;
 });
```

COMPOSANT BUILT-IN

Params

```
const params = {
  'sort': 'ascending',
  'page': '1'
};

http.get('/mon/super/service', { params })
// will call the URL ${baseUrl}/api/races?sort=ascending&page=1
  .subscribe(response => {
    // will return the pastes sorted
    this.pastes = response;
  });
```

COMPOSANT BUILT-IN

Auth

```
const headers = { 'Authorization': `Bearer ${token}` };

http.get('/mon/super/service', { headers })
  .subscribe(response => {
    // will return the pastes visible for the authenticated user
    this.races = response;
  });
```

ROUTING

LES MODULES

- Meta données
  - Compilation des components
  - Injections
  - Dépendances
  - Pipes, guards, resolvers, etc…
- Element de base d’une application Angular
- Génération: ng generate module …

DECLARATION

- Composant optionnel
- Declarations dans app.module ou fichier dédié. app.routes

```
export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  { path: ‘Other', component: OtherComponent }


  { path: ‘**’, component: ErrorComponent }
];
```

DECLARATION

```
@NgModule({
  imports: [BrowserModule, RouterModule.forRoot(ROUTES)],
  declarations: [……],
  bootstrap: […]
})
export class AppModule {
}
```

DECLARATION

Ajout d’un simple <router-outlet>

```
<main>
  <router-outlet></router-outlet>
  <!-- the component's template will be inserted here-->
</main>
```

NAVIGATION

Ajout des ‘router link’

```
<a href="" routerLink="/">Home</a>
<!-- same as -->
<a href="" [routerLink]="['/']">Home</a>
```

Css

```
<a href="" routerLink="/" routerLinkActive="selected-menu">Home</a>
```

NAVIGATION

Depuis le code:

```
    this.router.navigate(['']);
```

Passage de paramètres:

```
export class MyComp implements OnInit {
  elem: any;

  constructor(private service: Service, private route: ActivatedRoute) {
  }

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('myId');
    this.service.get(id).subscribe(elem => this.elem = elem);
  }
}
```

NAVIGATION

Depuis le code:

```
    this.router.navigate(['']);
```

Passage de paramètres:

```
export class MyComp implements OnInit {
  elem: any;

  constructor(private service: Service, private route: ActivatedRoute) {
  }

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('myId');
    this.service.get(id).subscribe(elem => this.elem = elem);
  }
}
```

NAVIGATION

Redirection:

```
{ path: '', pathMatch: 'full', redirectTo: '/breaking' },
```

Ordre d'interprétations

```
{ path: 'facebook/:id', component: FbComponent },
{ path: 'facebook/new', component: FbNewProfilComponent }
```

VS

```
{ path: 'facebook/new', component: FbNewProfilComponent }
{ path: 'facebook/:id', component: FbComponent },
```

NAVIGATION ++

Lors de la déclaration de composants complexes, mais devant avoir leurs propres “URL” (Path)

```
{
{ path: ‘facebook/:id’,
  component: FbComponent
  children: [
    { path: ‘time-line', component: TimeLineComponent },
    { path: 'menu', component: LeftMenuPanelComponent },
    { path: 'Ads', component: AdsComponent }
  ]
}
```

Soit

```
{
{ path: ‘facebook/:id’,
  component: FbComponent
  children: [
    { path: '', pathMatch: 'full', redirectTo: 'time-line' },
    { path: ‘time-line', component: TimeLineComponent },
    { path: 'menu', component: LeftMenuPanelComponent },
    { path: 'Ads', component: AdsComponent }
  ]
}
```

```
{
{ path: ‘facebook/:id’,
  component: FbComponent
  children: [
    { path: ‘’, component: TimeLineComponent },
    { path: 'menu', component: LeftMenuPanelComponent },
    { path: 'Ads', component: AdsComponent }
  ]
}
```

NAVIGATION ++

- appliquer des guards à plusieurs routes à la fois
- appliquer des resolvers à plusieurs routes à la fois

NAVIGATION ++

Gestion de la vue par profils: Les guards:

"rien n'empeche un utilisateur d'accéder à une "page" interdite" par l'URL.

LES GUARDS SONT LÀ POUR NOUS Y AIDER:

CanActivate: guard "simple"

CanActivateChild: guard sur elements enfants (wildcard)

CanLoad: Lazy loading

CanDeactivate: Empecher un changement de route

NAVIGATION ++

APPLICATION D'UNE GUARD

```
{ path: 'maroute', component: MyComponent, canActivate: [LoggedInGuard] }
```

DÉCLARATION D'UNE GUARD SIMPLE

```
@Injectable()
export class LoggedInGuard implements CanActivate {

  constructor(private router: Router, private userService: UserService) { }

  canActivate(
   route: ActivatedRouteSnapshot,
   state: RouterStateSnapshot): Observable<boolean>|Promise<boolean>|boolean {
    const loggedIn = this.userService.isLoggedIn();
    if (!loggedIn) {
      this.router.navigate(['/login']);
    }
    return loggedIn;
  }
}
```

NAVIGATION ++

Même principe que les resolvers “à l’ancienne” coté serveur

Avantage d’une application One Page

1. la navigation vers la nouvelle page a l’air plus rapide

2. l’utilisateur peut être perturbé si le chargement du contenu est trop long, parce que la page apparaît vide, ce qui ressemble à un bug
3. le template doit être codé avec soin, parce qu’il doit fonctionner correctement pendant la courte période où la course est `null` ou `undefined`
4. le template peut cependant donner un feedback immédiat en affichant un message ou une animation indiquant que le contenu est en court de chargement
5. Même si le chargement échoue (à cause d’une perte de connexion par exemple), au lieu de rester sur la page courante, la navigation est effectuée et l’URL change, bien que la page ne puisse afficher aucune donnée

NAVIGATION ++

Le meilleur du monde OPS et SSR

DECLARATION

```
@Injectable()
export class MsgResolver implements Resolve<MessageModel> {

  constructor(private mService: msgService) { }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
   Observable<MessageModel> | Promise<MessageModel> | MessageModel {
     return this.raceService.get(+route.paramMap.get('raceId'));
   }
}
```

APPLICATION À LA ROUTE

```
{
  path: 'messages/:msgId',
  component: MsgComponent,
  resolve: {
    msg: MsgResolver
  }
}
```

UTILISATION DANS LE COMPOSANT

```
export class MsgComponent implements OnInit {

  msg: MessageModel;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.msg = this.route.snapshot.data[‘msg’];
    // OU mieux:
    this.route.data.subscribe(data => this.msg = data['msg']);

   }
}
```

NAVIGATION ++

On pourra s'inscrire aux events générés par le router !

- NAVIGATIONSTART
- NAVIGATIONEND
- NAVIGATIONERROR
- NAVIGATIONCANCEL

NAVIGATION ++

```
ngOnInit() {
  this.router.events
    .subscribe((event) => {
      // example: NavigationStart, RoutesRecognized, NavigationEnd
      console.log(event);
    });
}
```

```
this.router.events
  .subscribe((event) => {
    if (event instanceof NavigationEnd) {
      console.log('NavigationEnd:', event);
    }
  });
```

NAVIGATION

Depuis le code:

```
    this.router.navigate(['']);
```

LAZY LOADING

FONCTIONNEMENT

- Les applications deviennent de plus en plus grande et Angular nous permet de définir des modules pour gérer la complexité et les dépendances.
- Angular nous permet, en plus de la gestion fine du routing, de charger les modules à la demande.
- Angular nous permet également de charger des modules en arrière plan, grace à la “preloadingStrategy”

PRÉPARATION

On va devoir définir un module qui sera chargé à la demande.

Ce module sera le point de départ du système de chargement à la demande.

Angular s'occupera lui même de générer les "chunks" javascript.

```
export const ADMIN_ROUTES: Routes = [
  { path: 'administrator', component: AdminComponent }
];
```

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(ADMIN_ROUTES)
  ],
  declarations: [AdminComponent]
})
export class AdminModule { }
```

On notera l'appel à "forChild"

PRÉPARATION

Il nous reste simplement à définir les routes dans le module racine:

```
{ path: 'admin', loadChildren: './admin/admin.module#AdminModule' }
```

Lors de la construction de l'application, Angular va analyser les routes, et détecter les modules lazy loadé.

OBSERVABLES & RXJS

PROMISES - RAPPEL

```
const getUser = function (login) {
  return new Promise(function (resolve, reject) {
    // Appels réseaux etc…
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};
```

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
  .catch(function (error) {
    console.log(error);
```

REACTIONS !

Dans la programmation réactive, toute donnée entrante sera dans un flux

... et même devenir un nouveau flux que l'on pourra aussi écouter.

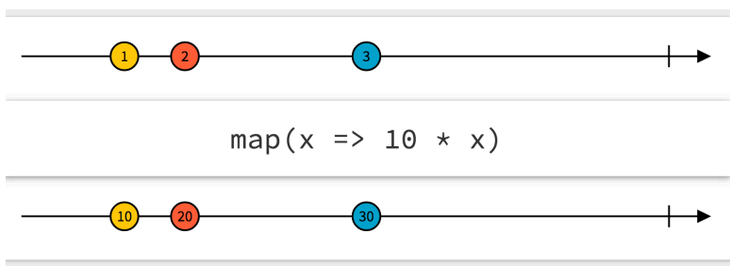
Un flux est une séquence ordonnée d'événements, qui représentent des valeurs

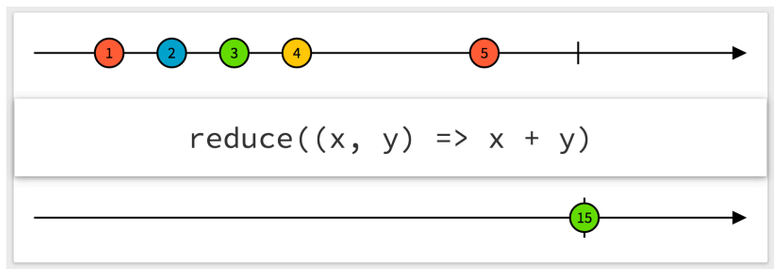
Un pattern bien connu: Observer

listener sera appelé un observer, et le flux, un observable

REACTIONS !

- Les observables permettent de gérer l'asynchronisme, et de profiter en plus d'une riche API permettant de modifier le flux (de l'observable), ou d'en générer un autre par chainage.





- Pas vraiment un concurrent aux promises

REACTIONS AVEC ANGULAR

```
const emitter = new EventEmitter();

emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
emitter.emit('there');
emitter.complete();
```

```
const emitter = new EventEmitter();

const subscription = emitter.subscribe(
value => console.log(value),
error => console.log(error),
() => console.log('done')

);

emitter.emit('hello');
subscription.unsubscribe(); // unsubscribe
emitter.emit('there');
```

REACTIONS AVEC ANGULAR

```
import { Component } from '@angular/core'
import { Observable } from 'rxjs/Rx'

@Component({
  selector: 'my-app',
  template: `
    <ul>
      <li *ngFor="let item of items | async">

      </li>
    </ul>
  `
})
export class AppComponent {
  public items = Observable.of([1, 2, 3])
}
```

```
  constructor(private http: HttpClient) { }

  public callRestService(): Observable<User[]> {
    return this.http
      .get<User[]>('https://jsonplaceholder.typicode.com/users');
  }

}
```

Depuis HttpClient plus besoin de map JSON

+ LES FONCTIONS FILTER, MAP, REDUCE... DES ARRAYS JS DISPO

DIFFERENCE ENTRE ‘OF’ ET ’FROM’

- Les observables sont des ensembles.
- Ces ensemble peuvent être créer de deux façon:
  - `Observable.of([1, 2, 3]).subscribe(x => console.log(x));`
  - `Observable.from([1, 2, 3]).subscribe(x => console.log(x));`

- Deux types d’observables
  - Les froids: Qui ne produiront aucune valeur s’ils ne sont pas écoutés ou observés via la fonction subscribe.
  - Les chauds: *Qui* produisent des valeurs même si le flux n’a pas de souscription.
- Dans tous les cas les observables sont lazy, rien ne se passe tant que “o.subscribe()” n’est pas appelée

PIPE ASYNC

On pourra directement utiliser un observable<[]> dans un *ngFor
grâce à " | async "

Ce pipe permet "d'attendre" la donnée, et de ne pas faire planté notre directive.

```
<div *ngFor="let user of users | async">{{user.user}}</div>
```

Users est de type Observable<User[]>

SERVICE WORKER

INTRODUCTION

- Specification existante
- Offline first & progressive app
- Fonctionne comme un proxy local
- Tab =! Service worker

INTERÊT

- La mise en cache d'une application est similaire à l'installation d'une application native. L'application est mise en cache en tant qu'unité et tous les fichiers sont mis à jour ensemble.
- Une application en cours continue de fonctionner avec la même version de tous les fichiers. Il ne commence pas soudainement à recevoir des fichiers en cache d'une version plus récente, qui sont probablement incompatibles.
- Lorsque les utilisateurs actualisent l'application, ils voient la dernière version entièrement mise en cache. Les nouveaux onglets chargent le dernier code en cache.
- Les mises à jour ont lieu en arrière-plan, relativement rapidement après la publication des modifications. La version précédente de l'application est servie jusqu'à ce qu'une mise à jour soit installée et prête.
- Les ressources ne sont téléchargées que si elles ont changé.

FONCTIONNEMENT

- Angular charge un “manifest” (héritage du *cache manifest)*
- Le manifest précise:
  - Resources en cache (hash)
  - Compare lors d’un update
- Le SW s’occupe de télécharger la nouvelle version et gère la mise en cache (ngsw-config.json)

INSTALLATION

yarn add @angular/service-worker

npm install @angular/service-worker

Configuration de cli .json afin de généré le manifest au Build-time

ng set apps.0.serviceWorker=true

I18N

INTRO

- Angular intègre un mécanisme d'internationalisation
- Basé sur une valeur injectable: LOCALE_ID
- @Inject(LOCALE_ID) public locale: string)

CHANGEMENT DE LA LOCAL AU DÉMARRAGE

```
import { registerLocaleData } from '@angular/common';
import localeFr from '@angular/common/locales/fr';

registerLocaleData(localeFr);
```

UTILISATION

Exemple dans un composant

```
@NgModule({
  imports: [BrowserModule],
  declarations: [CustomLocaleComponent], // and other components
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR'}
  ]
  // ...
})
export class AppModule { }

@Component({
  selector: 'ns-locale',
  template: `
    <p>The locale is {{ locale }}</p>
    <!-- will display 'fr-FR' -->

    <p>{{ 1234.56 | number }}</p>
    <!-- will display '1 234,56' -->
  `
})
class CustomLocaleComponent {
  constructor(@Inject(LOCALE_ID) public locale: string) { }
}
```

FONCTIONNEMENT

- Angular transforme les composants en"javascript" (c'est en partie grace à ca qu'il gère sa "change detection strategy"
- C'est pendant cette phase de compilation du HTML en JavaScript que la traduction est réalisée. Cela a des conséquences importantes
- On ne peut pas changer la locale (et donc le texte affiché dans l'application) pendant l'exécution. L'application entière doit être rechargée et redémarrée pour changer de langue
- Une fois démarrée, l'application est plus rapide, parce qu'elle ne doit pas traduire les clés encore et encore ;
- Si on utilise la compilation AOT, on doit construire et servir autant de versions de l'application que de locales supportées.

https://angular.io/guide/aot-compiler

DEMO

- On marque les parties des templates qui doivent être traduites en utilisant un attribut i18n
- On exécute une commande permettant d'extraire ces parties marquées vers un fichier, par exemple messages.xlf. Deux formats standards sont gérés
- On demande à un traducteur de fournir une version traduite de ce fichier, par exemple messages.fr.xlf ;
- On construis l'application en fournissant la locale ('fr' par exemple) et ce fichier contenant les traductions françaises. (messages.fr.xlf). Le compilateur Angular et la CLI remplacent les parties marquées via l'attribut i18n par les traductions trouvées dans le fichier, et configurent l'application avec le LOCALE_ID fournit

TESTING

TOOLING

- Jasmine & Karma
- Mocking
- Système built-in
- spec
- Gestion de protractor

TEST D'UN OBJET METIER

```
describe('MyClass', () => {
  let obj: MyClass;

  beforeEach(() => {
    obj = new MyClass('arg1', 10);
  });

  it('should have a name', () => {
    expect(obj.name).toBe('arg1');
  });

  it('should have a value', () => {
    expect(obj.value).not.toBe(1);
    expect(obj.value).toBeGreaterThan(9);
  });
});
```

TEST D'UN COMPOSANT - 1

JAVASCRIPT

```
@Component({
  selector: 'ns-pony',
  template: `<img [src]="'/images/pony-' + pony.color.toLowerCase() +
'.png'" (click)="clickOnPony()">`
})
export class PonyComponent {

  @Input() pony: PonyModel;
  @Output() ponyClicked = new EventEmitter<PonyModel>();

  clickOnPony() {
    this.ponyClicked.emit(this.pony);
  }

}
```

TEST D'UN COMPOSANT – 2 – EVALUATION DU DOM

JAVASCRIPT

```
import { TestBed } from '@angular/core/testing';

import { PonyComponent } from './pony_cmp';

describe('PonyComponent', () => {

  it('should have an image', () => {
    TestBed.configureTestingModule({
      declarations: [PonyComponent]
    });
    const fixture = TestBed.createComponent(PonyComponent);
    // given a component instance with a pony input initialized
    const ponyComponent = fixture.componentInstance;
    ponyComponent.pony = { name: 'Rainbow Dash', color: 'BLUE' };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have an image with the correct source attribute
    // depending of the pony color
    const element = fixture.nativeElement;

expect(element.querySelector('img').getAttribute('src')).toBe('/images/po
ny-blue.png');
  });
});
```

TEST D'UN COMPOSANT – 3 – EVENTS

```
it('should emit an event on click', () => {
  TestBed.configureTestingModule({
    declarations: [PonyComponent]
  });
  const fixture = TestBed.createComponent(PonyComponent);

  // given a pony
  const ponyComponent = fixture.componentInstance;
  ponyComponent.pony = { name: 'Rainbow Dash', color: 'BLUE' };

  // we fake the event emitter with a spy
  spyOn(ponyComponent.ponyClicked, 'emit');

  // when we click on the pony
  const element = fixture.nativeElement;
  const image = element.querySelector('img');
  image.dispatchEvent(new Event('click'));

  // and we trigger the change detection
  fixture.detectChanges();

  // then the event emitter should have fired an event
  expect(ponyComponent.ponyClicked.emit).toHaveBeenCalled();
});
```

TEST D'UN SERVICE - 1

JAVASCRIPT

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  constructor(private localStorage: LocalStorageService) {
  }

  list(): Array<RaceModel> {
    return this.localStorage.get('races');
  }
}
```

TEST D'UN SERVICE - 2

JAVASCRIPT

```
import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      { provide: LocalStorageService, useClass: FakeLocalStorage }
    ]
  }));

  it('should return 2 races from localStorage', () => {
    const service: RaceService = TestBed.get(RaceService);
    const races = service.list();
    expect(races.length).toBe(2);
  });
});
```

TEST D'UN SERVICE – 3

JAVASCRIPT

```
import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {

  const localStorage = jasmine.createSpyObj('LocalStorageService',
['get']);

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      { provide: LocalStorageService, useValue: localStorage }
    ]
  }));

  it('should return 2 races from localStorage', () => {
    localStorage.get.and.returnValue([{ name: 'Lyon' }, { name: 'London'
}]);

    const service: RaceService = TestBed.get(RaceService);
    const races = service.list();

    expect(races.length).toBe(2);
    expect(localStorage.get).toHaveBeenCalledWith('races');
  });
})
;
```

PROTRACTOR

- Jasmine/Assert: Test “unitaire”
  - Test d’une fonction métier
  - Retour d’une requête REST
  - ...
- Protractor:
  - Test impliquant de l’UI
    - Simulation de click, navigation ...
    - Test de présence d’elements sur la page

PROTRACTOR

```
import { browser, by, element } from 'protractor';

describe('Protractor Demo', () => {

  it('create Paste button should work',() => {

    expect(element(by.id('source-modal')).isPresent()).toBeFalsy("The modal window shouldn't appear right now ");
    element(by.buttonText('create Paste')).click();
    expect(element(by.id('source-modal')).isPresent()).toBeTruthy('The modal window should appear now');

  });
});
```

Fragmentation en 2 fichiers: ObjetPage / Tests

```
export class Pastebin extends Base {

  /* Pastebin Heading */
  getPastebinHeading(): promise.Promise<string> {
      return this.getPastebin().element(by.css("h2")).getText();
  }
```

```
it('should display the heading Pastebin Application', () => {

  expect(mainPage.getPastebinHeading()).toEqual("Pastebin Application");


});
```