retengr

ANGULAR
PRISE EN MAIN

Durée : 4 jours
Objectif : Maîtriser les bases d'Angular

Historique du support

Version	Date	Commentaire	Auteur
1.0.0	17/04/2020	Création support + Angular 9	Vincent Roth

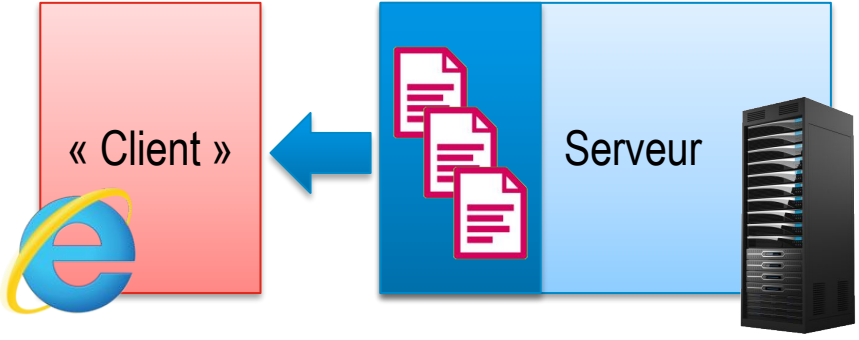
SOMMAIRE

SOMMAIRE

INTRODUCTION À L'**ARCHITECTURE WEB** D'AUJOURD'HUI

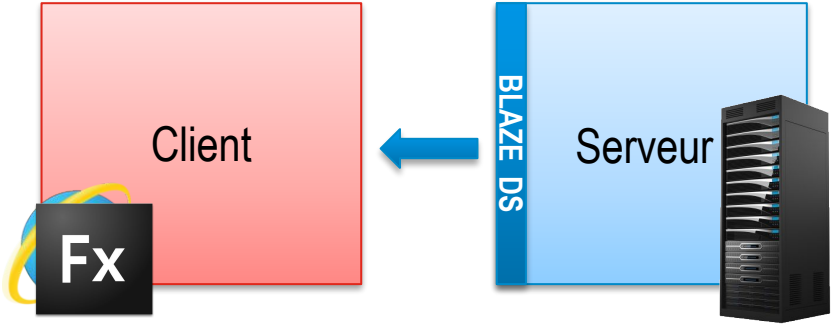
SINGLE PAGE APPLICATION

ALTERNATIVES



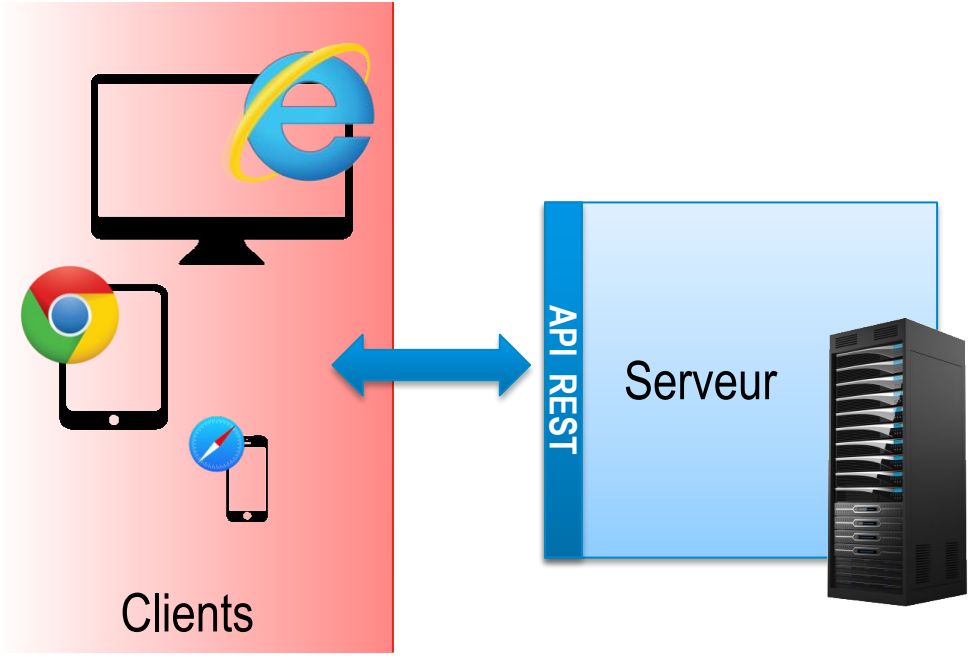
IHM GÉNÉRÉE ET GÉRÉE CÔTÉ SERVEUR

IHM CÔTÉ CLIENT, NON STANDARD



SINGLE PAGE APPLICATION

L'IHM CÔTÉ CLIENT BASÉE SUR LES STANDARDS DU WEB



- Application Web constituée d'une seule page HTML (et de ressources Javascript, CSS...)
- Routage, changement de vue géré côté client
- Echanges avec le serveur limités aux données

SINGLE PAGE APPLICATION

AVANTAGES

- Expérience utilisateur
- Performance
- Compétences dédiées
- Interopérable
- URL-friendly

REST

EXPOSITION DE SERVICES SIMPLE ET INTEROPÉRABLE

- REST (REpresentational State Transfer) n'est pas un protocole ou un standard, mais un style d'architecture pour concevoir des applications pour les systèmes hypermédia.
- REST repose sur les standards URI et HTTP (et liens hypertextes, média types, etc.)

Le Web est un espace d'information dans lequel les données -Ressources- sont désignés par des identificateurs globaux -URI- ("Uniform Resource Identifier"). Les ressources sont manipulées à l'aide des verbes HTTP (GET, POST, PUT, DELETE...)

- La technologie d'implémentation d'un serveur (ou d'un client) REST n'importe pas (.NET, Java, PHP, C, Ruby, Python, Node.js, etc.). Le respect des principes d'architecture permet aux architectures REST d'être interopérables.

PRÉSENTATION D'**ANGULAR**

HISTORIQUE ET OBJECTIFS

- Angular est un framework open-source créé par Google
- Il utilise le langage TypeScript, un sur-ensemble de JavaScript
- Il a pour but de faciliter la création d'applications web modernes, à page unique
- Il s'appuie sur des principes déclaratifs et propose d'étendre le langage HTML
- Angular se veut extensible et pleinement testable
- Successeur d'AngularJS (v1.x.x), il a débuté en 2009
  - 13 Juin 2012 AngularJS 1.0.0
  - 5 Février 2016 AngularJS 1.5.0 <- Prépare à l'architecture Angular
  - 14 Septembre 2016 Angular v2.0.0
  - 23 Mars 2017 Angular v4.0.0
  - Une version majeure tous les 6 mois (environs)
  - 6 Février 2020: Angular v9.0.0

PRINCIPES

DÉCLARATIF VS IMPÉRATIFS

- **Impératif** : principe de programmation décrivant les opérations de façon séquentielle afin qu'elles soient exécutées dans un certain ordre par la machine pour modifier l'état du programme
  - Exemple : **jQuery**

- **Déclaratif:** principe de programmation consistant à créer des applications sur la base de composants unitaires indépendant du contexte global, mais dépendant uniquement des arguments qui lui sont passés
  - Exemple : **AngularJS**

La programmation déclarative permet de décrire le « quoi » (texte, titres, paragraphes), quand la programmation impérative permet de décrire le « comment » (positionnement, couleurs, polices de caractères)

PRINCIPES

MV*

- AngularJS se base sur un pattern d'architecture MVVM (Modèle-Vue-VueModèle)
  - Le concept a été défini à l'origine par Microsoft pour WPF (Windows Presentation Foundation) et Silverlight en 2005
  - Comme tout pattern MV*, il permet de séparer la vue, de la logique et de l'accès aux données
  - Il se base sur des bindings et des évènements pour faire communiquer les différentes couches

La catégorisation des patterns a assez peu d'importance, ce qui compte étant la séparation des logiques métier et de présentation de manière à faciliter la maintenance des applications.

PRINCIPES

INJECTION DE DÉPENDANCES

- L’injection de dépendance (DI) est un pattern qui permet d’injecter de façon dynamique des dépendances, cela permet de :
  - Ne plus être responsable de la création des instances
  - Faciliter l’écriture des tests unitaires
  - Changer ou charger les dépendances dynamiquement au runtime ou à la compilation
  - Améliorer la maintenabilité et la compréhension du code

PRINCIPES

MODULES

- La plupart des applications sont conçues avec un point d'entrée principal qui instancie, câble, et démarre l'application.
- Pour Angular, l'approche est différente : l'application est constituée de modules. Chaque module est responsable d'un périmètre technique ou fonctionnel; il ya plusieurs avantages à cette approche :
  - Le principe de déclaration rend le code plus simple à comprendre pour la maintenabilité
  - Pas besoin de charger tous les modules lors des tests unitaires, ce qui facilite leur production
  - Facilite le packaging de code sous forme de composants externes réutilisables pour la capitalisation
  - Le chargement des modules peut se faire dans n'importe quel ordre ou même être chargés en parallèle (de par la nature asynchrone de l'exécution des modules)

PRINCIPES

Ecosystème

- Angular mets à disposition des packages permettant d'enrichir rapidement l'application de fonctionnalité :
  - localize : traduction statique de l'application
  - material : framework graphique basé sur le Material Design
  - pwa : mécanisme PWA
  - universal : rendu côté serveur

- Une CLI est aussi disponible et maintenu par Angular, il permet de :
  - initialiser une application Angular
  - construire l'application
  - générer des squelettes de composants, modules, …
  - lancer les tests unitaires et de bout en bout (end-to-end e2e)

PRINCIPES

TypeScript

- TypeScript est un langage de programmation, **sur-ensemble** de JavaScript.

- Gratuit et open-source
- Développé et maintenu par Microsoft

- Notions plus fortes de **typage** et de programmation orienté objet

- **Transpilé** en JavaScript
- Non interprétable directement par le navigateur.

MISE EN PLACE DE L'**ENVIRONNEMENT**

ENVIRONNEMENT

Logiciels

- Node.js

  https://nodejs.org/

- Visual Studio Code (ou autre IDE surtout TS) :

  https://code.visualstudio.com/Download

- Angular CLI :

  https://www.npmjs.com/package/@angular/cli

ANGULAR CLI

Commandes

- ng new <projet>

  Crée une application Angular de nom donné en paramètre
- ng serve

  Démarre l'application pour l'aperçu en développement
- ng generate <type> <chemin-local/nom>

  Génère les fichiers propres au type d'objet demandé (module, component, service)
- ng help

  Fournit la liste des commandes disponibles

  --help en paramètre d'une commande pour afficher une aide de la commande

EXERCICE

Création d'une application

- Créer une application Angular avec la CLI

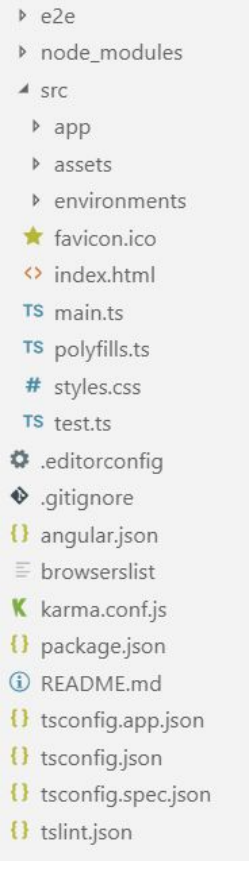
  Avec routage et SCSS

- Nous ferons pas à pas une petite application pour gérer des animaux et leur vétérinaire dans une clinique vétérinaire.
  - Fiche d'un animal (nom, espèce, vétérinaire, commentaire)
  - Liste des animaux
  - Création/Modification de la fiche d'un animal
  - Ajout d'un vétérinaire (nom et prénom)

  - Un animal est suivi par un vétérinaire
  - Un vétérinaire suit plusieurs animaux

ANGULAR

Arborescence d'un projet Angular



src/app/	Fichiers de l'application Angular
src/assets/	Fichiers statiques servant à l'application
node_modules/	Les dépendances NPM, résolues avec *npm install*
angular.json	Configuration pour Angular CLI
package.json	Info du projet, dépendances et scripts pour NPM
tsconfig.json	Configuration de la transpilation TS en JS
tslint.json	Configuration du linter TS
karma.conf.js	Configuration pour *karma*, gérant les tests unitaires
e2e/protractor.conf.js	Configuration pour *protactor*, gérant les tests e2e

MODULE ANGULAR

MODULE ANGULAR

- Un **module** Angular décrit une partie de l'application Angular et comment elles s'intègrent entre elles.

  Toute application Angular à au moins un module, défini avec le décorateur **NgModule**. Il défini l'amorce (bootstrap) de l'application.

- À ne pas confondre avec les modules TypeScript

  Les modules TypeScript (fichier .ts) définissent un périmètre de code TypeScript (mot-clefs export/import).

  Les modules Angular couvrent plusieurs fichiers, celui du module et les composants qui lui appartiennent.

MODULE ANGULAR

Définition

- Décorateur **@NgModule**
- declarations

  Objets-vues (**composant**, **pipe** et **directive**) appartenant au module.
- imports

  Autres modules dont les objets-vues sont utilisés dans le template des composants du module.
- exports

  Les objets-vues que le module autorise à utiliser s'il est importé.
- providers

  Instance de **services** que le module fourni à la collection globale de services de l'application. Ces instances sont disponibles dans toute l'application.
- bootstrap

  Liste des composants qui seront à la racine de l'application, c'est-à-dire ceux qui contiennent les autres.

```
import { BrowserModule } from
'@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

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

MODULE ANGULAR

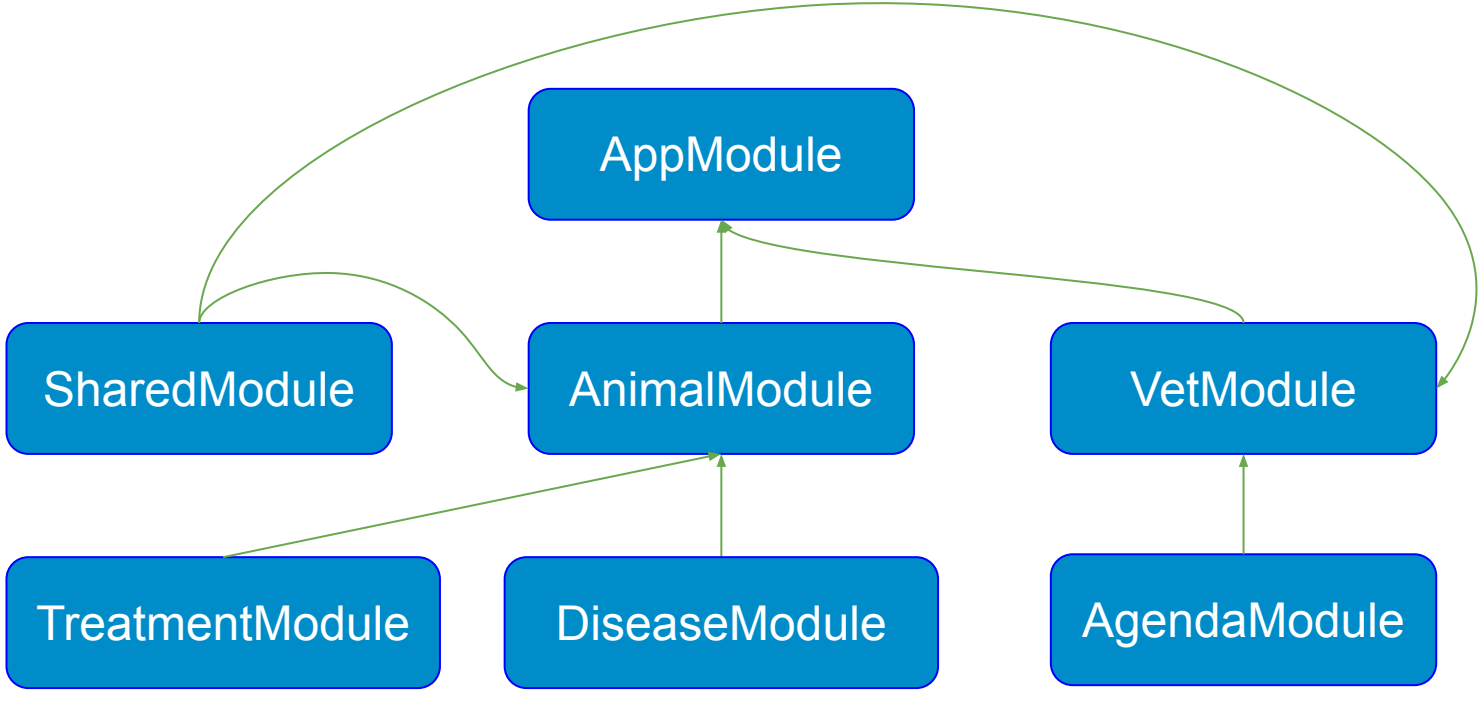
Hiérarchie

- Angular recommande une approche fonctionnel de l'application :
  - Chaque fonctionnalité à son propre **répertoire**
  - Chaque fonctionnalité à son propre **module Angular**
  - Chaque fonctionnalité à son propre **composant principal**
  - Chaque fonctionnalité à sa propre **navigation**

- Une fonctionnalité peut être découpées en plusieurs fonctionnalités
- Ce qui est commun à l'application ou à une fonctionnalité est placé dans un **module partagé** (SharedModule).
- Chaque objet Angular ne doit être responsable que d'une seule chose.

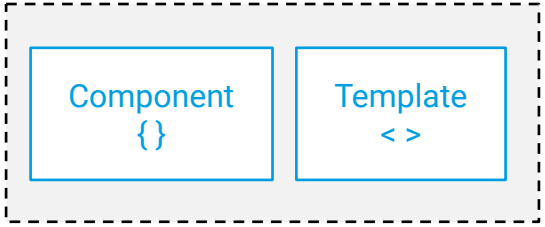
MODULE ANGULAR

Hiérarchie



COMPOSANT

COMPOSANT



Définition

- Un **composant** Angular contrôle une partie de l'écran, appelé **vue**.

- Décorateur **@Component**
- selector

  Balise où l'instance de ce composant est inséré dans le HTML.
- templateUrl

  Chemin vers le template **HTML** du composant.
- styleUrls

  Tableau de chemins des styles **CSS** affectant le template du composant.

  Le sélecteur **:host** permet d'affecter le style du composant au niveau de sa balise initiale.
- providers

  Tableau d'**instances isolées de service** qu'Angular fournira au composant.

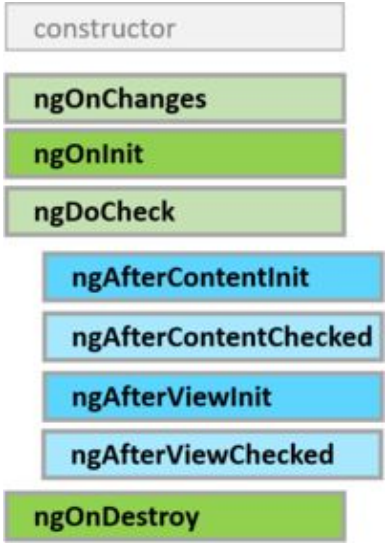
```
import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
    providers: []
})
export class AppComponent {
    title = 'app';
}
```

COMPOSANT

Cycle de vie

- Un **composant** a un cycle de vie géré par Angular :
  - création
  - initialisation
  - rendu de la vue
  - destruction



- Angular peut recycler une instance de composant plutôt que d'en créer un nouveau, il vaut mieux utiliser l'initialisation (**ngOnInit**) plutôt que la création (**constructor**) pour initialiser l'état du composant.

EXERCICE

Module et composant

- Créer un type (classe ou interface) pour la fiche d'un animal.

  Il caractérise ce qui est contenu dans la fiche : nom, espèce, vétérinaire, commentaire.

- Utiliser la CLI pour créer un module animal.

- Utiliser la CLI pour créer un composant animal dans le module animal.

- Utiliser le composant animal dans le composant app.

- Afficher un animal dans le composant animal.

  Interpolation dans le template des attributs du composant avec **{{ }}**

SERVICE

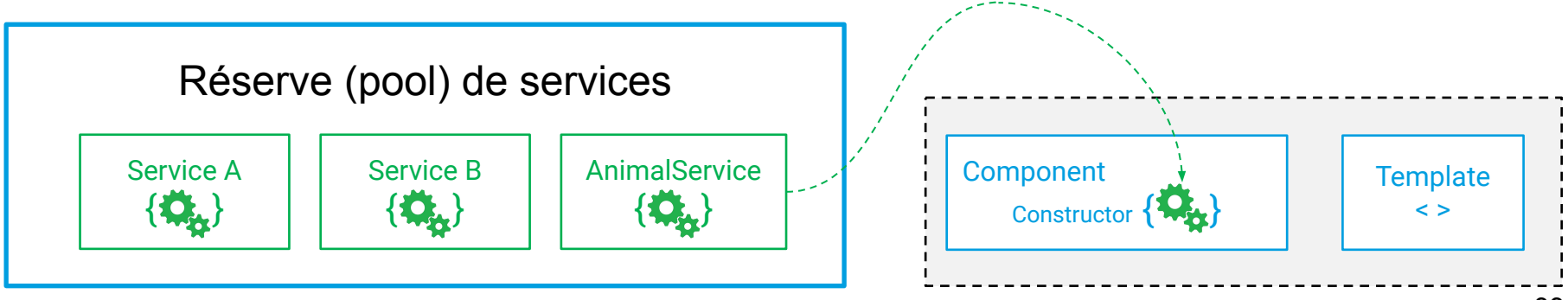
SERVICE

Définition

- Un **service** Angular est une classe avec une responsabilité définie.
  Elle fait quelque chose de spécifique et le fait bien.

- Décorateur **@Injectable**

  Il indique à Angular de gérer les instances de cette classe et de les injecter dans les composants, directives et services en dépendant (injection de dépendance).



SERVICE

Fournir un service

- Il y a 3 manières de fournir une instance de service :
  - Par la décorateur **@Injectable du service** avec la propriété **providedIn**

    La valeur "root" sera la plupart du temps utilisée, et rend l'instance disponible dans toute l'application.

  - Par un **module** via la propriété **providers** du décorateur **@NgModule**

    L'instance est alors disponible pour toute l'application.

  - Par un **composant** via la propriété **providers** du décorateur **@Component**

    L'instance est disponible uniquement pour ce composant.

EXERCICE

Service

- Utiliser la CLI pour créer un service animal.

  Il fournira les données relatives aux animaux.

- Créer une méthode dans le service pour obtenir une fiche.

- Utiliser ce service depuis le composant pour afficher la fiche consultée.

DIRECTIVE

DIRECTIVE

Définition

- Il existe 3 types de directives dans Angular :
  - **Directive structurelle** : modifie la structure du DOM en ajoutant et supprimant des éléments.
  - **Directive d'attribut** : modifie le comportement d'un élément, d'un composant ou d'une autre directive.
  - **Composant** : une directive avec un template

DIRECTIVE

Directive structurelle

- Elle manipule la structure du DOM, en ajoutant, supprimant ou modifiant des éléments.
- Un astérisque * précède le nom de la directive.

```
<ng-container *ngIf="animal">{{ animal.name }}</ng-container>

<ul>
    <li *ngFor="let animal of animals">{{ animal.name }}</li>
</ul>

<ng-container [ngSwitch]="animal?.species">
    <app-cat *ngSwitchCase="'cat'" [animal]="animal"></app-cat>
    <app-dog *ngSwitchCase="'dog'" [animal]="animal"></app-dog>
    <app-animal *ngSwitchDefault [animal]="animal"></app-animal>
</ng-container>
```

- ng-container est un élément Angular qui n'est pas placé dans le DOM.
- ?. est un opérateur Angular qui arrête le parcours de l'objet si la valeur précédente est nulle (safe navigator operator)

DIRECTIVE

Directive d'attribut

- Elle modifie le comportement de l'élément, du composant ou d'une autre directive.
- Décorateur **@Directive**
- selector

```
<p appLowercase>{{ animal.name }}</p>
```

Attribut, entre crochets [...], qui associe le HTML à cette directive.

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
    selector: '[appLowercase]'
})
export class LowercaseDirective {
  constructor(private el: ElementRef<HTMLElement>) {}

  @HostListener('mouseenter') onMouseEnter(): void {
    this.tranforsm('lowercase');
  }

  private tranforsm(textTransform: string): void {
    this.el.nativeElement.style.textTransform = textTransform;
  }
}
```

EXERCICE

Directive

- Créer un composant responsable de l'affichage de la liste des fiches.

  Afficher ce composant dans le composant app

- Utiliser une directive structurelle pour afficher la liste des fiches.

- Créer une directive d'attribut qui affichera le nom de l'animal de la fiche en majuscule au survol de la souris.

  Ceci est une directive simple pour l'exemple, on privilégiera l'usage du CSS pour un comportement modifiant le style de l'élément.

PIPE

PIPE

Définition

- Transforme une valeur.
- Principalement utilisé dans le template.

```
import { Pipe, PipeTransform} from '@angular/core';

@Pipe({
    name: 'lowercase'
})
export class LowercasePipe implements PipeTransform {
  tranforsm(text: string): string {
    return text.toLowerCase();
  }
}
```

```
<p>{{ animal.name | lowercase }}</p>
```

EXERCICE

Directive

- Créer un pipe pour tronquer le commentaire d'une fiche d'un animal à 177 caractères et ajouter "…".

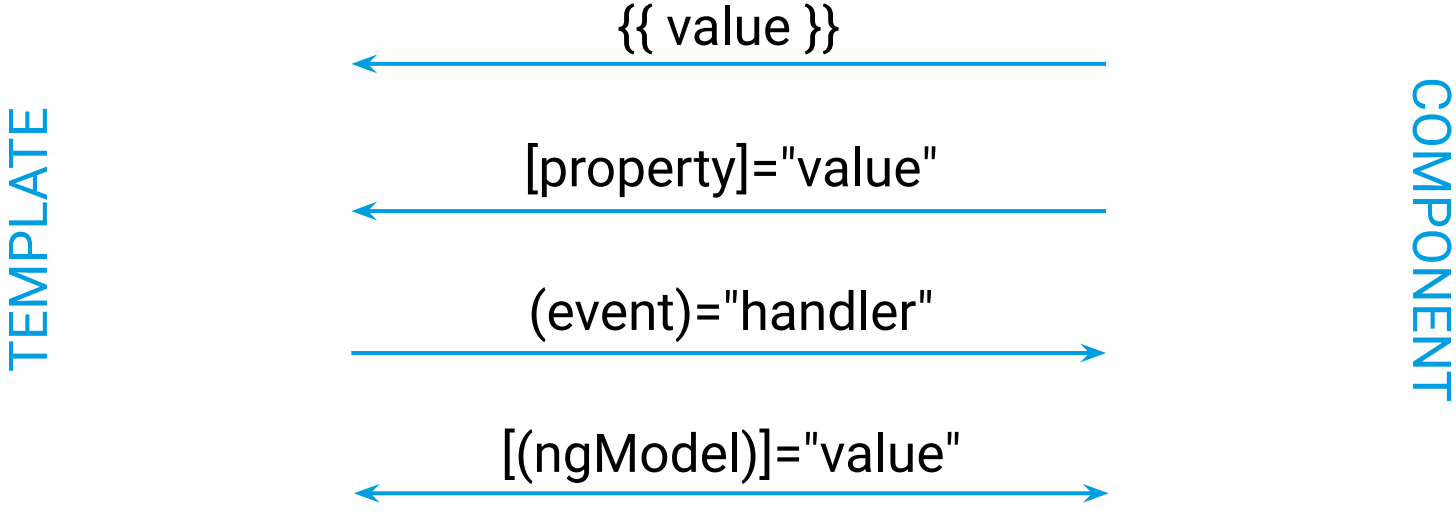
  Si le commentaire est plus court que 180 caractères, il doit être affiché entièrement.
- Faire passer en argument du pipe le nombre de caractère à tronquer.

LIAISON DE DONNÉES
DATA BINDING

LIAISON DE DONNÉES

Définition

- La liaison de données est un mécanisme qui lie une partie du template à une partie d'un composant.
- Une syntaxe spécifique dans le template HTML indique à Angular la direction de la liaison.

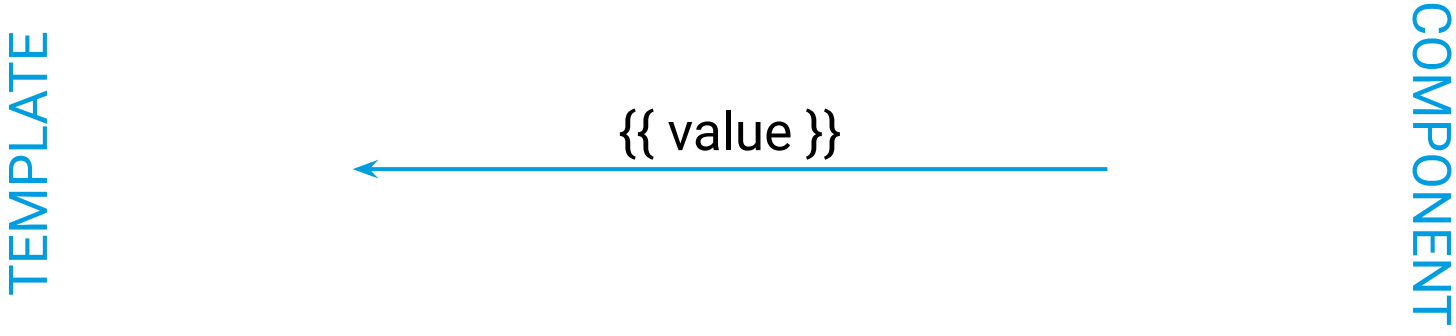


LIAISON DE DONNÉES

Interpolation

- Interprète du code au sein du template HTML.
- Accède à tout attribut et méthode publiques du composant.
- Opération JS/TS possible.

```
<label>
    {{ animal.name }}
</label>
```



LIAISON DE DONNÉES

Liaison de propriété (property binding)

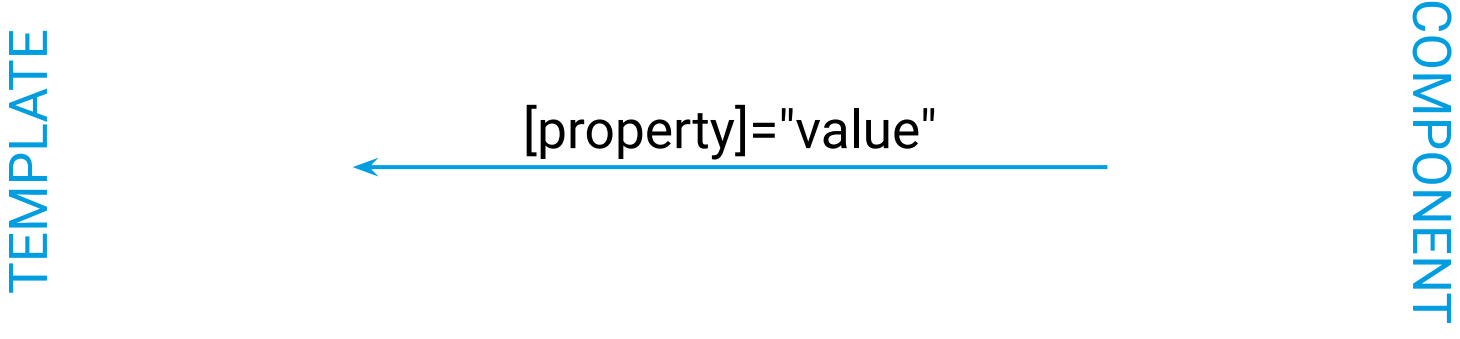
- Transmet une valeur du composant parent à l'attribut d'un composant enfant.

Template parent :

```
<app-animal [model]="animal" *ngFor="let animal of animals">
</app-animal>
```

Composant enfant :

```
@Input() model: Animal;
```

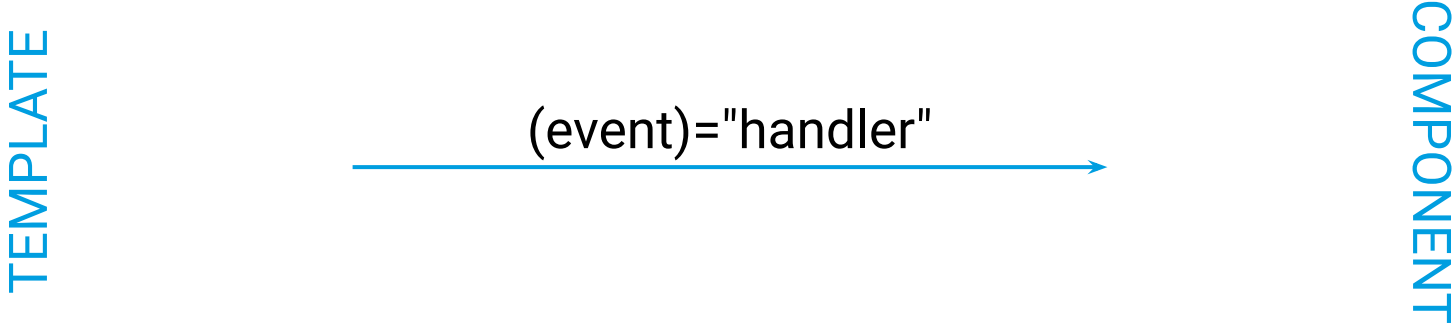


LIAISON DE DONNÉES

Liaison d'événement (event binding) du DOM

- Appelle une méthode du composant lorsque l'événement se produit sur l'élément.
- L'événement peut être passé à la méthode en utilisant **$event**.

```
<button (click)="onNext($event)">Next</button>
```



LIAISON DE DONNÉES

Liaison d'événement (event binding) d'un composant

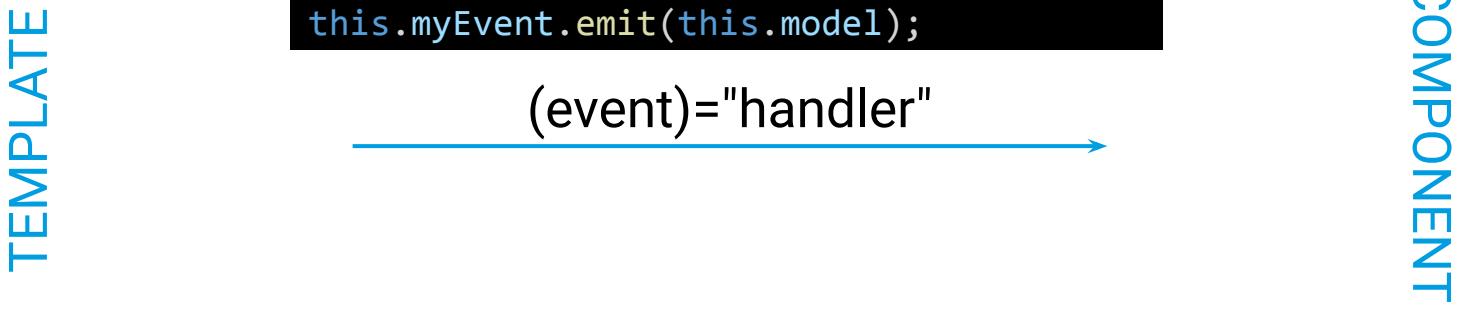
- Appelle une méthode du composant lorsque l'événement est émis par le composant enfant.
- Ce qu'émet le composant peut être passé à la méthode en utilisant **$event**.

Template parent :

```
<app-animal [model]="animal" (myEvent)="onEvent($event)">
</app-animal>
```

Composant enfant :

```
@Output() myEvent: EventEmitter<Animal>;
...
this.myEvent.emit(this.model);
```

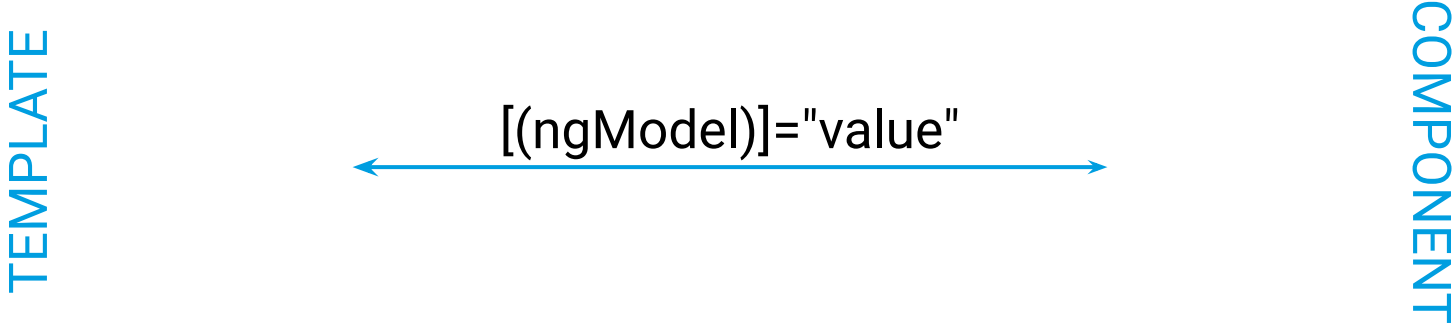


LIAISON DE DONNÉES

Liaison de données bidirectionnelle (two-way data binding)

- Combine les liaisons de propriété et d'événement.
- La valeur est transmise au template et mise à jour lorsqu'elle évolue dans la vue.
- Notamment utilisé pour les champs de formulaire.

```
<input [(ngModel)]="animal.name">
```



EXERCICE

Liaison de données

- Créer un composant responsable de l'affichage d'une fiche de la liste et des actions relatives à une fiche.

- Utiliser une liaison de propriété pour fournir la fiche entre le composant de liste et celui d'affichage.

- Utiliser une liaison d'événement pour supprimer une fiche de la liste.

  Le bouton de suppression est géré par le composant en charge de l'affichage d'une fiche de la liste.

COMMUNICATION SERVEUR
HTTP CLIENT

COMMUNICATION SERVEUR

Service HttpClient

- Service fourni par Angular avec le module **HttpClientModule** (@angular/common/http)

- Fournit une API pour les requêtes HTTP, basé sur *XMLHttpRequest*

  - Typage des requêtes et des réponses
  - Gestion d'erreur basé sur les Observables
  - Support pour les tests
  - Support pour les intercepteurs de requête et de réponse

COMMUNICATION SERVEUR

Service HttpClient - Typage

```
httpClient.get<Animal>('/api/animals/1')
.subscribe({
  next: animal => this.name = animal.name
});
```

COMMUNICATION SERVEUR

Service HttpClient - Observable

- Usage de tous les opérateurs de la librairie RxJS :

map	reduce
filter	tap
retry	...

```
httpClient.get<Animal[]>('/api/animals')
.pipe(retry(3))
.subscribe({
  next: animals => this.animals= animals
});
```

COMMUNICATION SERVEUR

Service HttpClient - Test

1 - Exécution de la requête.

2 - Vérification que la requête attendue a bien été faite et fournit l'objet de requête.

3 - Fournit une réponse à la requête.

4 - Vérification qu'aucune autre requête a été réalisée.

```
(service: AnimalService, httpMock: HttpTestingController) => {
1     service.get(1).subscribe(animal => expect(animal.name).toEqual('Teto'));
2     const req = httpMock.expectOne('/api/animals/1');
      expect(req.request.method).toEqual('GET');
3     req.flush({ name: 'Teto' });
4     httpMock.verify();
}
```

COMMUNICATION SERVEUR

Service HttpClient - Intercepteur

- Un intercepteur peut intervenir à deux moments de la requête :
  - Avant son envoi au serveur.
  - Après réception de la réponse.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

    constructor(private auth: AuthService) {}

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        // Obtient l'en-tête d'authentification du service.
        const authHeader = this.auth.getAuthorizationHeader();
        // Clone la requête pour ajouter le nouvel en-tête.
        const authReq = req.clone({headers: req.headers.set('Authorization', authHeader)});
        // Passe la requête clonée à la place de la requête d'origine.
        // Retourne un Observable qui peut être écouté pour la réponse à la requête.
        return next.handle(authReq);
    }
}
```

COMMUNICATION SERVEUR

Service HttpClient - Intercepteur

- Un intercepteur est un **service** particulier qui doit être fourni par un module.

```
import { NgModule } from '@angular/core';
import { HTTP_INTERCEPTORS } from '@angular/common/http';

import { AuthInterceptor } from './auth.interceptor';

@NgModule({
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ]
})
export class AuthModule { }
```

EXERCICE

Mise en place du serveur

- Installer json-server : https://www.npmjs.com/package/json-server
- Définir une commande dans le package.json pour lancer json-server avec le fichier JSON contenant les données.
- Vérifier que le serveur json-server fonctionne
- Configurer le proxy d'Angular CLI pour que les requêtes aillent sur le serveur json-server : https://angular.io/guide/build#proxying-to-a-backend-server
- Vérifier que le serveur json-server est accessible depuis l'*host* Angular

EXERCICE

Communication serveur

- Modifier le service animal pour utiliser le service HttpClient et obtenir les fiches d'animaux depuis le serveur.

- Créer une méthode pour supprimer une fiche existante sur le serveur.

- Écrire un intercepteur pour mesurer le temps que prend chaque requête.

ROUTAGE

ROUTEUR

Définition

- Permet la navigation d'une **vue** à une autre à mesure que l'utilisateur fait des actions.
  - Interprète l'URL du navigateur pour naviguer vers une **vue générée côté client**.
  - Gère les **paramètres d'URL** et les fournit au composant.
  - Utilise des **liens** pour naviguer vers la vue correspondante.

Vue ≠ Page

Une vue est une partie de l'écran dont un composant est responsable.

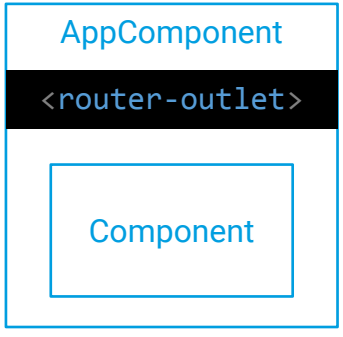
ROUTEUR

Routes

- Une application Angular utilisant le routage a une **seule instance** (singleton) du service **Router**.
  - Quand l'URL client change, il vérifie l'existence d'une **Route** qui déterminera le composant à afficher.
  - Le composant est placé après le **RouterOutlet** du template

```
const appRoutes: Routes = [
    { path: '', component: AnimalListComponent },
    { path: 'animals/:id', component: AnimalComponent },
    { path: 'veterinarians', component: VetComponent }
]
```

```
RouterModule.forRoot(appRoutes)
```



ROUTEUR

Liens

- La directive **RouterLink** sur une balise ancre **<a>** donne le contrôle de cet élément au routeur.
  - Si le chemin de navigation est **fixe**, on peut directement donner la valeur au RouterLink.

```
<a routerLink="/">Home</a>
```

  - Si le chemin de navigation est **dynamique**, il faut fournir un tableau au RouterLink.

```
<a [routerLink]="['animals',animal.id]">{{animal.name}}</a>
```

ROUTEUR

Paramètre d'URL

- Le service **ActivatedRoute** permet d'accéder aux informations de la route courante.
  - La propriété **paramMap** est un Observable qui émet une valeur à chaque changement de route.

```
this.activatedRoute.paramMap.subscribe((params: ParamMap) =>
  this.service.get(+params.get('id')).subscribe(animal => this.animal = animal)
);
```

  - La propriété **snapshot** contient les valeurs de la dernière route.

```
const id = this.activatedRoute.snapshot.paramMap.get('id');
this.service.get(+id).subscribe(animal => this.animal = animal));
```

EXERCICE

Routeur

- Configurer les routes vers la liste et le détail d'une fiche.
- Ajouter un lien pour chaque élément de la liste des fiches pour naviguer vers le détail de la fiche.
- Utiliser le paramètre d'URL pour afficher la fiche demandée.
- Ajouter un lien pour revenir à l'accueil de l'application.

  Ici notre accueil est la liste des fiches.

ROUTE ENFANT

Route par fonctionnalité

- Chaque fonctionnalité gère ses propres routes.

AppRoutingModule :

```
const appRoutes: Routes = [
  { path: '', redirectTo: 'animals', pathMatch: 'full' }
]
```

AnimalRoutingModule :

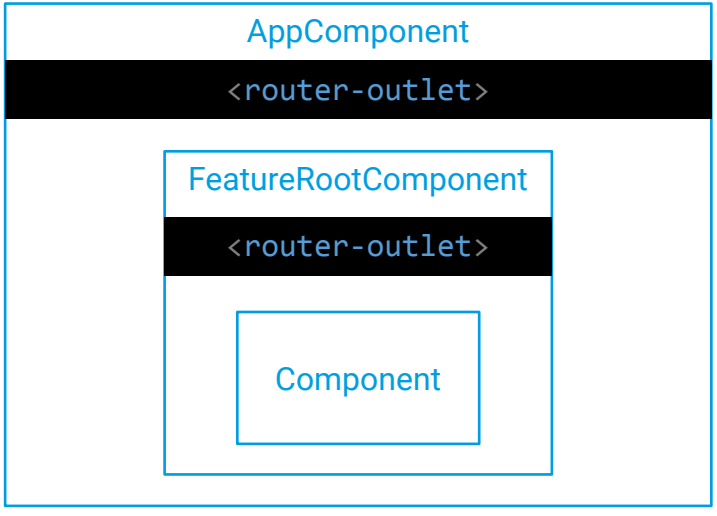
```
const animalRoutes: Routes = [
  { path: 'animals', component: AnimalListComponent },
  { path: 'animals/:id', component: AnimalComponent }
]
```

```
RouterModule.forChild(animalRoutes)
```

ROUTE ENFANT

Route enfant

- Par rapport aux routes principales, une route enfant :
  - étend le chemin définie par la route parent.
  - le composant de la route est placé au niveau du RouterOutlet du composant de la route parent.




```
const animalRoutes: Routes = [{
  path: 'animals', component: AnimalRootComponent,
  children: [
    {
      path: '',
      component: AnimalListComponent
    },
    {
      path: ':id',
      component: AnimalComponent
    }
  ]
}];
```

EXERCICE

Route enfant

- Créer un composant racine pour la fonctionnalité “animal”.

- Créer un module de routage pour la fonctionnalité “animal” et y gérer les routes propres à la fonctionnalité.

ROUTE ASYNCHRONE

Définition

- Charge le module de la route de manière asynchrone.
  - Le module n'est chargé que si l'utilisateur accède à la route.
  - Peut améliorer le temps de chargement de l'application.
  - L'application peut être étendue sans alourdir le chargement initial.
  - La somme de la taille des modules est supérieure à une application sans asynchronisme.

```
const appRoutes: Routes = [
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin.module').then(mod => mod.AdminModule)
  }
];
```

EXERCICE

Route asynchrone

- Créer un module vétérinaire avec son composant racine.
- Ajouter une route asynchrone vers ce module.
- Créer un composant de liste et de détail et y afficher les vétérinaires provenant d'un service dédié.

GUARD

Définition - canActivate

- Permet de valider la navigation vers une route.
- Peut stopper la navigation.

```
const routes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard]
  }
];
```

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    // Do some logic
    return true;
  }
}
```

GUARD

Définition - canDeactivate

- Permet de valider la navigation depuis une route.
- Peut stopper la navigation.

```
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

@Injectable({ providedIn: 'root' })
export class CanDeactivateGuard
  implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(component: CanComponentDeactivate) {
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

```
const routes: Routes = [
  {
    path: ':id',
    component: MyFormComponent,
    canDeactivate: [CanDeactivateGuard]
  }
];
```

RESOLVE

Définition

- Permet de charger des données pour le composant d'une route.
- Peut aussi manipuler la navigation.

```
import { Injectable } from '@angular/core';
import { Resolve, RouterStateSnapshot, ActivatedRouteSnapshot } from '@angular/router';
...

@Injectable({ providedIn: 'root' })
export class AnimalDetailResolverService implements Resolve<Animal> {
  constructor(private service: AnimalService) {}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Animal> {
    const id = route.paramMap.get('id');
    return this.service.get(+id);
  }
}
```

RESOLVE

Définition

```
const routes: Routes = [
  {
    path: ':id',
    component: AnimalDetailComponent,
    resolve: {
      animal: AnimalDetailResolverService
    }
  }
];
```

```
ngOnInit() {
  this.activatedRoute.data.subscribe((data: { animal: Animal }) => {
    this.animal = data.animal;
  });
}
```

EXERCICE

Resolve

- Utiliser un Resolve pour obtenir l'animal pour le composant de détail de la fiche d'un animal.

    Ceci remplace l'usage de l'ID dans le composant

FORMULAIRES

FORMULAIRE PILOTÉ PAR LE TEMPLATE
TEMPLATE-DRIVEN

FORMULAIRE PILOTÉ PAR LE TEMPLATE

Définition

- Utilise les contrôles de formulaire HTML classique, tel que <input> ou <select>.
- Liaison bidirectionnelle avec les attributs du composant avec la directive **NgModel**, du module **FormsModule** (@angular/forms).
- Liaison d'événement **ngSubmit** pour l'envoi du formulaire.

  Réagit à la touche Entrée et au clic sur le bouton sans attribut type ou de type "submit".

```
<form (ngSubmit)="onSubmit()" >
  <input placeholder="Name" name="name" [(ngModel)]="model.name">
  <button type="submit">Submit</button>
</form>
```

```
onSubmit() {
  this.animalService.create(this.model).subscribe();
}
```

EXERCICE

Formulaire piloté par le template

- Créer un composant de formulaire pour une fiche d'un animal.
- Avoir un attribut correspondant au modèle dans ce composant.
- Définir le formulaire dans le template.
- Faire la liaison entre le modèle et le formulaire.
- Envoyer la nouvelle fiche au serveur.

FORMULAIRE PILOTÉ PAR LE TEMPLATE

Validation

- Utilise des directives de validation dont le nom est le même que les validateurs HTML 5.

pattern	required
min	minlength
max	maxlength
step	

```
<input name="name" [(ngModel)]="model.name" required minlength="4">
```

FORMULAIRE PILOTÉ PAR LE TEMPLATE

Gestion des erreurs

- Usage de variable de référence du template avec **#**.
- Le formulaire et chaque contrôleur de champ peut être référencé.

```
<form #animalForm="ngForm" (ngSubmit)=onSubmit()>
  ...
  <button type="submit" [disabled]="!animalForm.form.valid">Submit</button>
</form>
```

```
<input name="name" [(ngModel)]=model.name #name="ngModel" required>
<p *ngIf="name.errors?.required">Please fill a title.</p>
```

EXERCICE

Formulaire piloté par le template - Validation

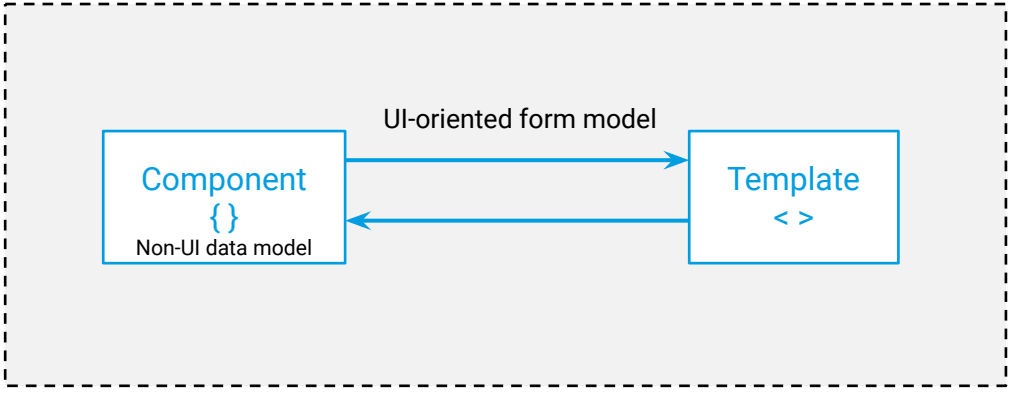
- Ajouter des validations pour les champs obligatoires.
- Afficher un message d’erreur pour chaque champ avec une validation.
- Envoyer la nouvelle fiche uniquement si le formulaire est valide.

FORMULAIRE RÉACTIF
REACTIVE FORM

FORMULAIRE RÉACTIF

Définition

- Formulaire piloté par le modèle.
- Gestion explicite entre un modèle de donnée (souvent en provenance d'un serveur) et un modèle de formulaire.
- Usage du module **ReactiveFormsModule**.



FORMULAIRE RÉACTIF

Modèle de formulaire

- Contient l'état et les valeurs des contrôleurs HTML.

```
this.vetForm = new FormGroup({
  firstName: new FormControl(),
  lastName: new FormControl()
});
```

```
<form [formGroup]="vetForm" (ngSubmit)="onSubmit()">
  <input placeholder="Last Name" formControlName="lastName">
  <input placeholder="First Name" formControlName="firstName">
  <button type="submit">Submit</button>
</form>
```

FORMULAIRE RÉACTIF

Gestion explicite du flux de donnée

- Il est de la responsabilité du composant de gérer les modèles.

```
this.vetForm = new FormGroup({
  firstName: new FormControl(vet.firstName),
  lastName: new FormControl(vet.lastName)
});
```

```
onSubmit() {
  const formModel = this.vetForm.value;
  const toSaveVet: Veterinarian = {
    firstName: formModel.firstName as string,
    lastName: formModel.lastName as string
  };
  this.vetService.create(toSaveVet).subscribe();
}
```

EXERCICE

Formulaire réactif

- Créer un composant de formulaire pour un vétérinaire.
- Avoir un attribut correspondant au modèle de formulaire dans ce composant.
- Définir le formulaire dans le template.
- Faire la liaison entre le modèle de formulaire et le formulaire.
- Envoyer le nouveau vétérinaire au serveur.

FORMULAIRE RÉACTIF

Validation

- Défini pour chaque **FormControl** dans le modèle de formulaire en utilisant des functions **Validators**.

required	requiredTrue
min	minLength
max	maxLength
email	pattern

```
this.vetForm = new FormGroup({
  name: new FormControl(
    '',
    [
      Validators.required,
      Validators.minLength(4)
    ]
  )
});
```

FORMULAIRE RÉACTIF

Gestion des erreurs

- Accès direct au **FormGroup** pour la validation du formulaire.
- Accès au **FormControl** par la méthode **get** du FormGroup.

```
<form [formGroup]="vetForm" (ngSubmit)="onSubmit()">
  ...
  <button type="submit" [disabled]="!vetForm.valid">Submit</button>
</form>
```

```
<input formControlName="firstName">
<p *ngIf="vetForm.get('firstName').errors?.required">Fill a first name.</p>
```

EXERCICE

Formulaire réactif - Validation

- Ajouter des validations pour les champs obligatoires.
- Afficher un message d’erreur pour chaque champ avec une validation.
- Envoyer la nouvelle fiche uniquement si le formulaire est valide.

FORMULAIRES

Comparaison

	Piloté par le template	Réactif
Préparation	Moins explicite, créé par des directives	Plus explicite, créé dans le composant
Modèle de donnée	Non structuré	Structuré
Prédictibilité	Asynchrone	Synchrone
Validation	Directives	Fonctions
Mutabilité	Mutable	Immuable
Scalabilité	Abstraction au dessus des APIs	Accès bas niveau des APIs

MOTEUR DE RENDU

MOTEUR DE RENDU

Pourquoi un moteur de rendu ?

- Les navigateurs n'interprètent pas directement les composants d'Angular.
- Ces composants doivent être construits pour être compréhensible.

MOTEUR DE RENDU

Modes du moteur Angular

- Juste à temps, Just-in-time (JIT), construit l'application dans le navigateur.

  Utilisé en développement jusqu'à Angular 8

- En avance sur son temps, Ahead-of-time (AOT), construit l'application une seule fois à sa construction.
  - Rendu plus rapide
  - Moins de requête asynchrone
  - Taille du framework réduit
  - Détecte les erreurs de template plus tôt
  - Meilleur sécurité

  Utilisé en production, et en développement depuis Angular 9 (moteur Ivy)

ENVIRONNEMENT

ENVIRONNEMENT

- Le fichier **environment.ts** est utilisé pour définir des valeurs dépendantes de l'environnement.

- La CLI peut construire l'application avec un des fichiers d'environnement :
  - paramètre **--prod** pour construire une application avec le fichier **environment.prod.ts** et un profil de construction orienté production.
  - paramètre **--configuration=xyz** pour construire une application avec le profil de construction xyz, défini dans le fichier angular.json.

TESTS

TESTS UNITAIRES

TESTS UNITAIRES

Introduction

- Angular utilise la librairie **Karma** pour les tests unitaires.
- Les fichiers ***.spec.ts** définissent les tests à jouer et est généré par la CLI.

- Seul le composant ou le service testé est chargé.
- Toutes les dépendances doivent être définies dans le module de test.
- Les **RouterModule** et **HttpClientModule** ont des modules spécifiques pour les tests.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [AnimalListComponent, AnimalItemComponent],
    imports: [RouterTestingModule, HttpClientTestingModule],
    providers: [AnimalService]
  }).compileComponents();
}));
```

TESTS UNITAIRES

Tester un composant

- Seul les attributs et méthodes publiques de la classe sont accessible.
- Les entrées d'un composant (notamment @Input) peuvent être initialisé avant le test.

```
beforeEach(() => {
  fixture = TestBed.createComponent(AnimalItemComponent);
  component = fixture.componentInstance;
  component.model = new Animal();
  component.model.name = 'Teto';
  fixture.detectChanges();
});

it('should render name of animal, () => {
  const compiled = fixture.debugElement.nativeElement;
  expect(compiled.querySelector('#name').textContent).toContain(
    component.model.title
  );
});
```

TESTS DE BOUT EN BOUT
END-TO-END TEST

TESTS DE BOUT EN BOUT

- Angular utilise la librairie **Protractor** pour les tests de bout en bout.
- Les répertoire **e2e** définit la configuration et les tests à jouer.
- Chaque ***.e2e.spec.ts** vise à tester un comportement de l'application.
- Toute l'application est chargée et est manipulée à travers le navigateur.

```
navigateTo() {
  return browser.get(browser.baseUrl) as Promise<any>;
}

getTitleText() {
  return element(by.css('app-root h1')).getText() as Promise<any>;
}
```

```
it('should display welcome message', () => {
  page.navigateTo();
  expect(page.getTitleText()).toEqual('Welcome to Angular App!');
});
```

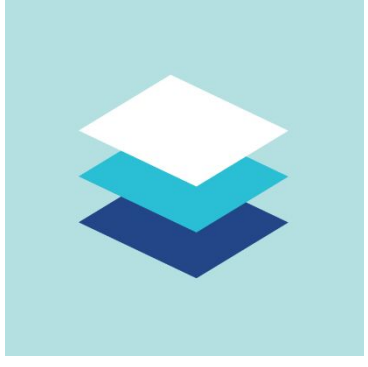
LIBRAIRIES

ANGULAR MATERIAL

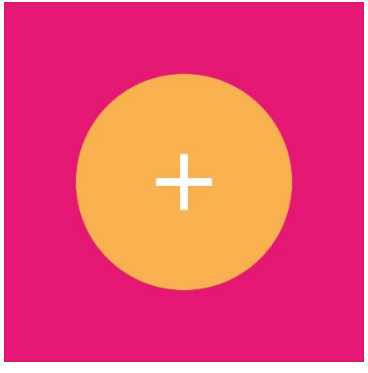
ANGULAR MATERIAL

Définition

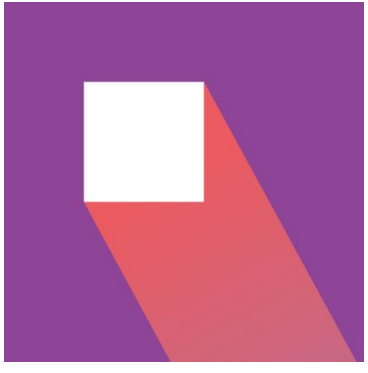
- Librairie de composants graphiques pour Angular.
- Basée sur une IU/XU de conception matérielle (Material Design), language visuel créé par Google en 2014, s'inspirant du rendu matériel tel le papier, et avec des principes réactifs (responsive).



Une métaphore matérielle



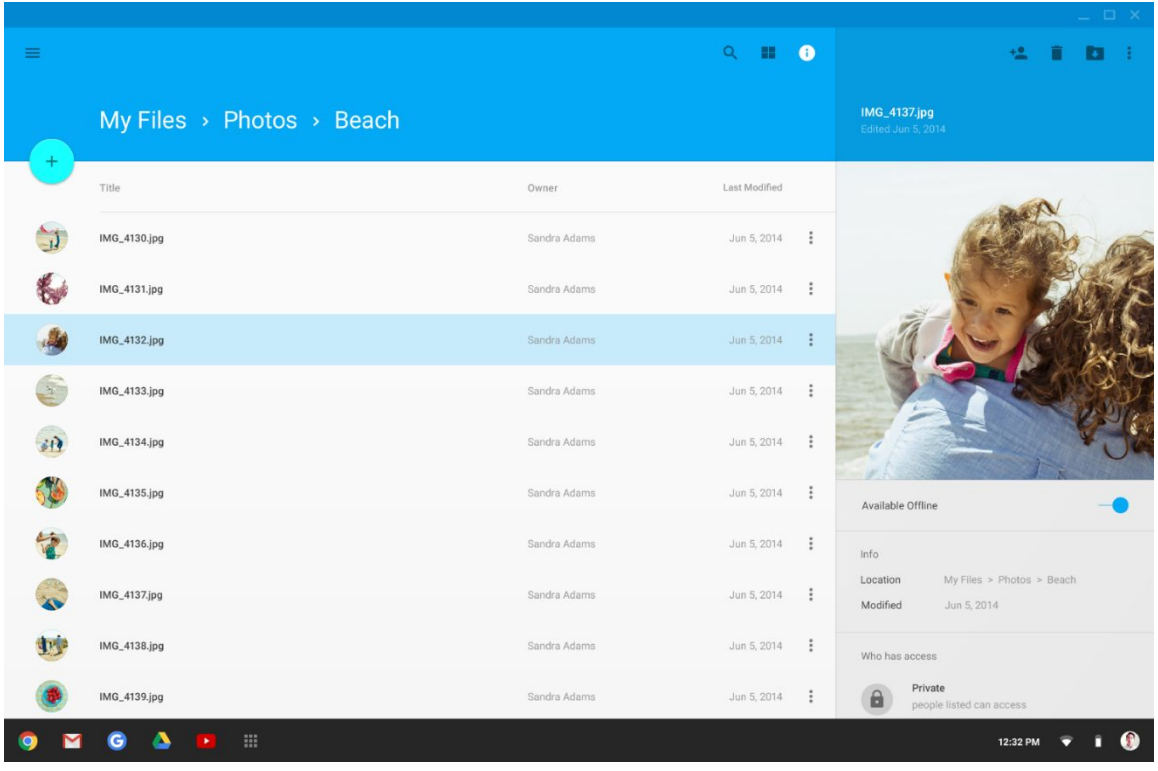
Audacieux, graphique, intentionnel



Le mouvement apporte du sens

ANGULAR MATERIAL

Exemple



EXERCICE

Angular Material

- Ajouter la librairie @angular/material avec *ng add*.
  https://material.angular.io/guide/getting-started

- Ajouter une barre d'outils (Toolbar)

- Utiliser des cartes (Card) pour la liste des fiches des animaux.

INTERNATIONALISATION

ANGULAR I18N

- L'outils d'internationalisation d'Angular aide à faire des application disponible dans plusieurs langues.

- La traduction se fait en 4 étapes :
  - Marquage du texte statique dans le template
  - Extraction du texte dans un fichier standardisé
  - Traduire ou faire traduire le fichier standardisé
  - Construire l'application avec le fichier traduit, remplaçant le texte marqué

- Une application doit être construite et déployée pour chaque langue supportée.

TRADUCTION DYNAMIQUE

- Plus simple d'utilisation, notamment pour le code TypeScript
- Ne requiert qu'une seule application.
- Plus lent, et plus complexe à tester.
- Changement à la demande de la langue.

- Exemple de librairie : @ngx-translate/core

EXERCICE

i18n avec ngx-translate

- Installer avec NPM les librairies @ngx-translate/core et @ngx-translate/http-loader
  - https://www.npmjs.com/package/@ngx-translate/core
  - https://www.npmjs.com/package/@ngx-translate/http-loader

- Créer un fichier fr.json dans le répertoire src/assets/i18n

- Charger le fichier fr.json et traduire dynamiquement du texte de l'application

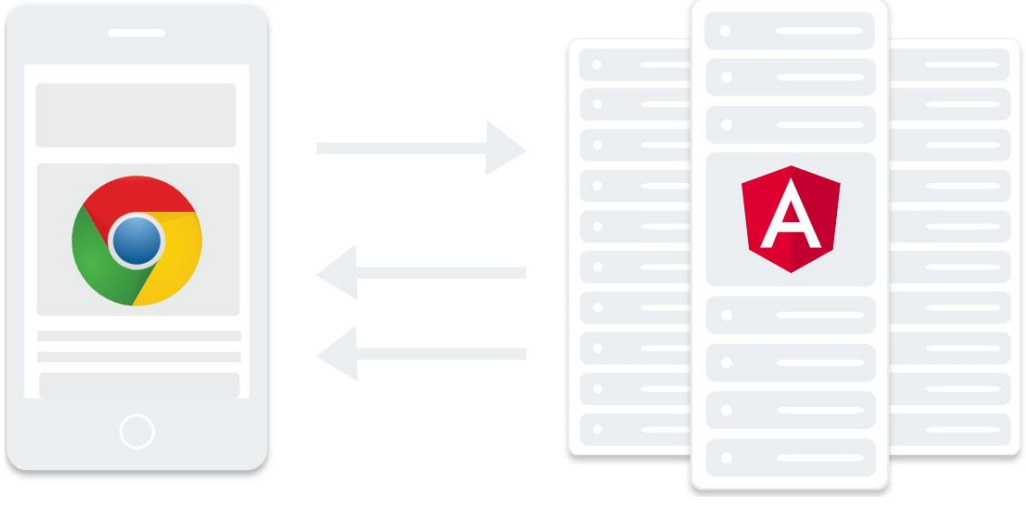
  Par exemple tout le texte dans le template du composant d'une fiche d'un animal dans la liste.

  Suivre la documentation de @ngx-translate/core.

PRÉ-RENDU CÔTÉ SERVEUR
ANGULAR UNIVERSAL

ANGULAR UNIVERSAL

- Pré-rendu côté serveur de l'application Angular



ANGULAR UNIVERSAL

Avantages

- Meilleure performance perçue

  Les utilisateurs de l'application verront instantanément une vue rendue par le serveur qui améliore considérablement les performances perçues et l'expérience utilisateur globale.

- Optimisé pour les moteurs de recherche (SEO)

  Le pré-rendu côté serveur permet de garantir que tous les moteurs de recherche peuvent accéder au contenu de l'application sans avoir à interpréter JavaScript.

- Aperçu du site

  Assure que les applications de médias sociaux affichent correctement une image d'aperçu de votre application.

APPLICATION WEB PROGRESSIVE (PWA)

ANGULAR PWA

- Une PWA est une application Web qui peut être installée et manipulée comme une application native pour smartphone.
- Utilise des API standardisées du navigateur Web pour stocker des données et interagir avec l'appareil (comme le GPS).
- Angular fournit un module pour rendre l'application PWA.

EXERCICE

PWA

- Ajouter la librairie @angular/pwa avec *ng add*.
  https://angular.io/guide/service-worker-getting-started

FIN

SOURCES

Angular
- angular.io
- angular.io/guide/cheatsheet
- CLI : github.com/angular/angular-cli
- Version sémantique : semver.org

Material
- material.angular.io

I18n
- angular.io/guide/i18n
- npmjs.com/package/@ngx-translate/core

HTML 5 validation de formulaire
- MDN HTML5 constraint validation

ES6
- es6-features.org

TypeScript
- typescriptlang.org

RxJS
- rxjs-dev.firebaseapp.com/

ANNEXES

JAVASCRIPT
ECMA SCRIPT 6

FONCTION FLECHE

Arrow function

- Raccourci pour définir une fonction avec la syntaxe **=>**.
- Sans corps de fonction **{ }** et **return** implicite pour les expressions.
- Supporte les déclarations classiques avec corps de fonction **{ }** et **return** éventuel.
- Le contexte de la fonction (**this**) est partagé à l'expression ou code de function.

FONCTION FLECHE

Exemples


```
const ages = [10, 12, 13, 14, 19, 20, 27];
const minors = ages.filter(age => age < 18);
const agesMetada = ages.map(age => ({ age, minor: age < 18 }));
const minorAgesSum = minors.reduce(((acc, age) => acc + age), 0);
```

```
var ages = [10, 12, 13, 14, 19, 20, 27];
var minors = ages.filter(function (age) {
   return age < 18;
});
var agesMetada = ages.map(function (age) {
   return { age: age, minor: age < 18 };
});
var minorAgesSum = minors.reduce(function (acc, age)
{
   return acc + age;
}, 0);
```

FONCTION FLECHE

Contexte partagé

```
function Person(age) {
   this.age = age;
   const log = () => { console.log(this.age); };
   log();
}
new Person(10);
```


```
function Person(age) {
   this.age = age;

   var log = function log() {
      console.log(this.age);
   }.bind(this);

   log();
}
new Person(10);
```

LITTERAUX DE GABARITS

Template literals

- Chaîne de caractère avec support multi-ligne et interpolation.
- Délimité par l’accent grave ``.
- Interpolation avec la syntaxe **${}**.

LITTERAUX DE GABARITS

Exemples


```es6
const agent = {
   firstName: 'James',
   lastName: 'Bond'
};
const catchPhrase = `My name is ${agent.lastName} ...
${agent.firstName} ${agent.lastName}`;
```


```es5
var agent = {
   firstName: 'James',
   lastName: 'Bond'
};
var catchPhrase = 'My name is ' + agent.lastName + ' ...\n' + agent.firstName + ' ' + agent.lastName;
```

TYPESCRIPT

TYPES

- Types primitifs :
  - boolean
  - number
  - object
  - string
  - symbol
  - void
- Typage des variables, paramètre et retour de fonction
- Inférence de type par rapport à la première affectation
- Assertion de type avec **<**type**>** ou **as** type

TYPES

Variable

Type

```
let isDone: boolean = false;
```

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b0101;
let octal: number = 0o744;
```

```
let color: string = "blue";
color = 'red';
```

Assertion

```
let someValue: any = 'This is a string';
let stringLength = (<string>someValue).length;
```

```
let someValue: any = 'This is a string';
let stringLength = (someValue as string).length;
```

TYPES

Fonction

Typage

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

Paramètre optionnel

```
function buildName(firstName: string, lastName?: string) {
   if (lastName) {
      return `${firstName} ${lastName}`;
   }
   return firstName;
}
```

Paramètre avec valeur par défaut

```
function buildName(firstName: string, lastName = '-') {
   return `${firstName} ${lastName}`;
}
```

Reste des paramètres

```
function buildName(firstName: string, ...rest: string[]) {
   return `${firstName} ${rest.join('')}`;
}

let name = buildName('Joseph', 'Lucas', 'Macleod');
```

CLASSES

- Définition de classe comme ES6 mais avec :
  - une définition des attributs
  - visibilité des attributs et des méthodes
  - des accesseurs
- Visibilité :
  - public : visibilité par défaut, comme JavaScript
  - private : uniquement accessible depuis la classe elle-même
  - protected : accessible aussi par les classes enfants

CLASSES

Exemples

```
class Greeter {
   private greeting: string;

   constructor(message: string) {
      this.greeting = message;
   }

   greet(): string {
      return `Hello, ${this.greeting}`;
   }
}

let greeter = new Greeter('World');

greeter.greet();
```

Définition et assignation des paramètres de constructeur comme attributs de classe.

```
class Greeter {
   constructor(private greeting: string) {
   }

   greet(): string {
      return `Hello, ${this.greeting}`;
   }
}
```

CLASSES

Accesseur

```
class Employee {
   private _fullName: string;

   get fullName(): string {
      return this._fullName;
   }

   set fullName(newName: string) {
      // Do some logic
      this._fullName = newName;
   }
}

let employee = new Employee();
employee.fullName = 'John Smith';
```

MODULE

- Définition de module comme ES6 :
  - périmètre du module à lui-même et non un périmètre global
  - la relation entre modules est déclarative : les exports et imports sont explicites

MODULE

Exemple

Fichier string-validator.ts

```
export interface StringValidator {
   isAcceptable(s: string): boolean;
}
```

Fichier post-code-validator.ts

```
import { StringValidator } from './string-validator';

const postCodeRegexp = /^[0-9]{5}$/;

export class PostCodeValidator implements StringValidator {
   isAcceptable(postCode: string) {
      return postCodeRegexp.test(postCode);
   }
}
```

Fichier main.ts

```
import { PostCodeValidator } from './post-code-validator';

let myValidator = new PostCodeValidator();
```

RXJS

RXJS

Introduction

- Rx = ReactiveX = Reactive Extensions
- API pour la programmation événementielle et asynchrone avec des flux
- Plus qu'une API, c'est une nouvelle manière de programmer
- Plusieurs implémentations : RxJava, RxJS, Rx.NET...

RXJS

Définition

- RxJS
  - Objet principal : Observable
  - Objets complémentaires : Observer, Subject, Scheduler
  - Opérateurs : map, filter, reduce, every, …
- Combine :
  - pattern Observer
  - pattern Iterator
  - programmation fonctionnelle
- Gestion des séquences d'évènements et de flux de données

RXJS

Fonctionnalités

- Facilite la gestion du cycle d'un flux
  - Création du flux
  - Composition / transformation / combinaison de flux, avec les opérateurs
  - Ecoute, avec la souscription

- Usage de fonction pure
  - Le résultat est le même pour les mêmes paramètres
  - Pas d'effet de bord dans les opérateurs
  - Effet de bord uniquement dans la souscription finale

OBSERVABLE

Exemple

```
const observable = Observable.create((observer) => {
  console.log('Hello');
  observer.next(42);
});

observable.subscribe({
  next: (x) => {
    console.log(x);
  },
});

observable.subscribe({
  next: (y) => {
    console.log(y);
  },
});
```

OPERATEURS

Sur un flux existant

```
import { map, filter, scan, tap, bufferCount } from 'rxjs/operators';

const output: Observable<number[]> = inputObservable.pipe(
  map((x) => x * 5),
  filter((x) => x < 45),
  scan((acc, x) => acc + x),
  tap((x) => console.log(x)),
  bufferCount(3)
);
```

OPERATEURS

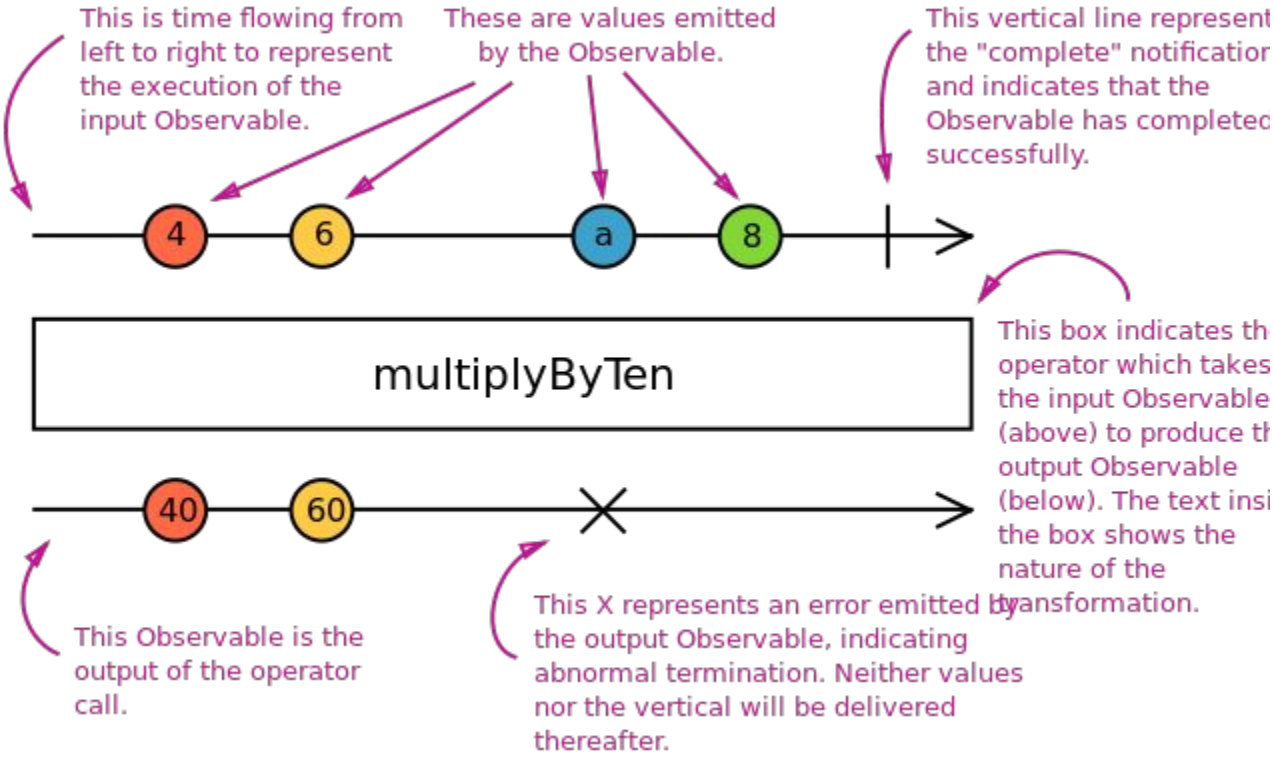
Création de flux

```
import { of, interval, from, fromEvent, merge } from 'rxjs';

of(1, 2, 3, 4);
interval(1000);
from([1, 2, 3, 4]);
fromEvent(buttonElement, 'click');
from(promise);
concat(observable1, observable2);
merge(observable1, observable2);
```

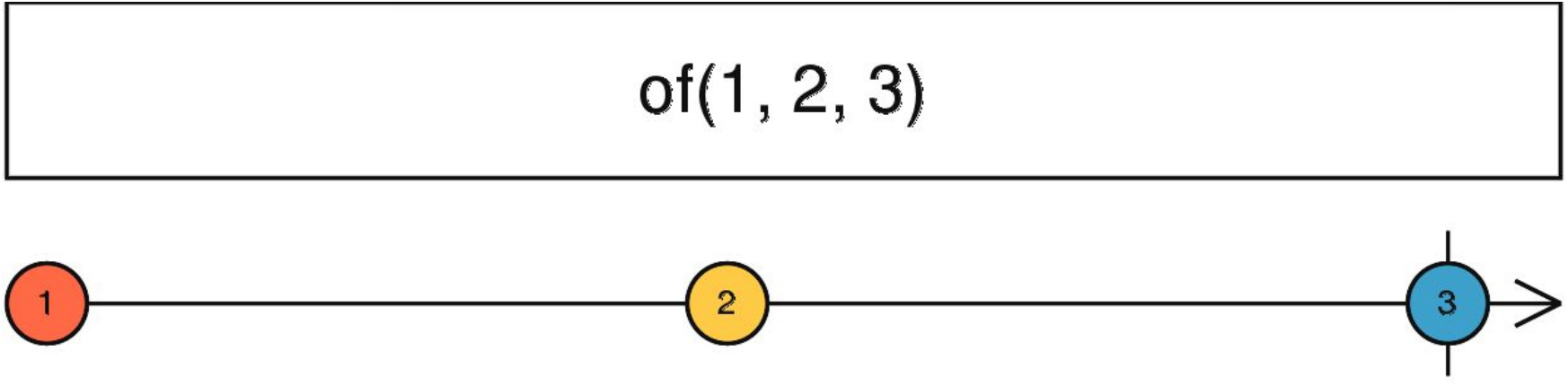
OPERATEURS

Diagramme de Marble



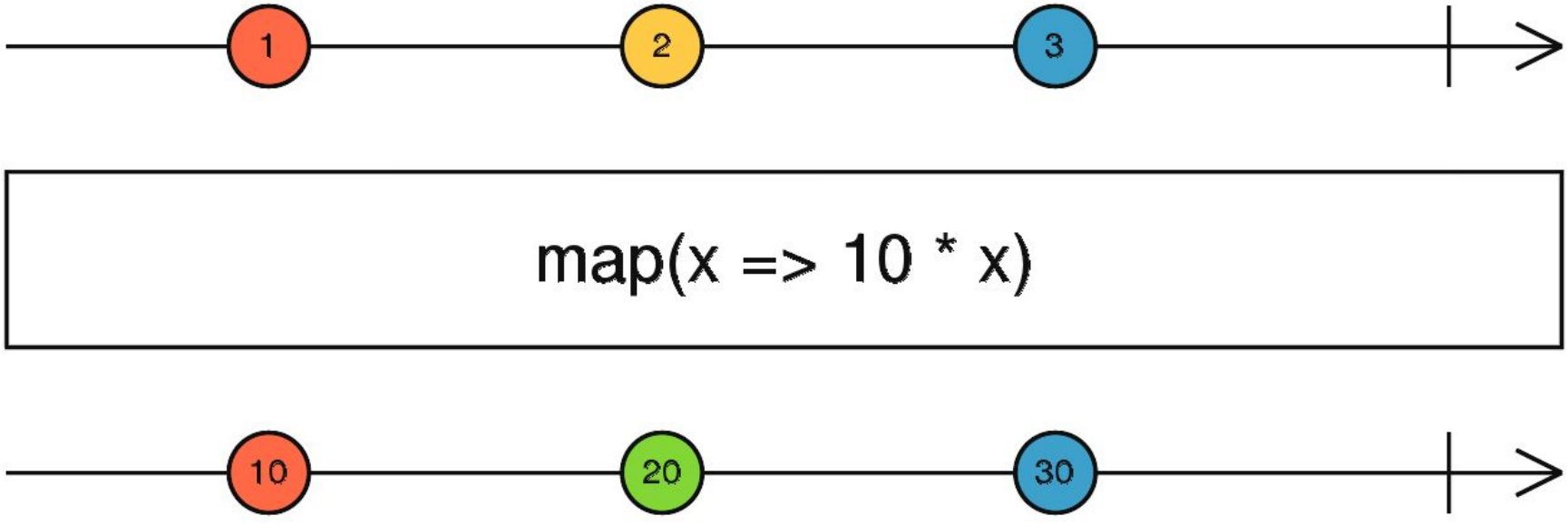
OPERATEURS

Diagramme de Marble - opérateur of



OPERATEURS

Diagramme de Marble - opérateur map



OPERATEURS

Diagramme de Marble - opérateur mergeAll