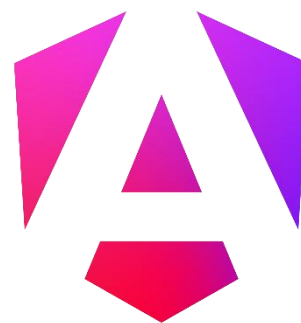


Une journée d'initiation à Angular

Sébastien Chassande-Barrioz

sebastien.chassande-barrioz@cgi.com



- Rappel les supports de cours sont dispo sur :
- <https://im2ag-moodle.univ-grenoble-alpes.fr>
 - Ou <http://chassande.fr/formation/ecom/>

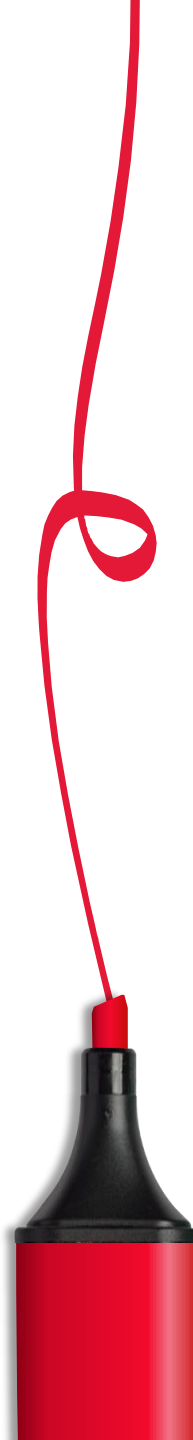
Agenda

Partie 1

- Javascript + TypeScript
- Boite à Outils
- TP initialisation du projet
- Basiques Angular
 - Les composants
 - Les templates
 - Les structures de contrôle
- TP Premier pas
- Les signaux
 - Cours + TP

Partie 2

- Les observables
 - Cours + TP
- Les services
 - Cours + TP
- Le routage
 - Cours + TP
- Les formulaires
 - Cours + TP



JavaScript





Javascript : Généralités

- JavaScript != Java
- Orienté Objet
- Faiblement typé et dynamique
- Programmation OO, Impérative et fonctionnelle
- ECMAScript 5 (ES5) = JS
- Nouvelle version : ECMAScript 2015/ES6
 - Nouveauté : Classes, Constantes, Arrow functions, Modules
 - Transpileur vers ES5



Javascript ES6 : VAR/LET/Const

- Déclarer une variable avec **var** provoque le hoisting : déclaration de la variable tout en haut de la fonction.
- Nouveau mot clé : **let** : déclare la variable à l'endroit prévu
- Nouveau mot clé : **const** : déclare une constante à l'endroit prévu

```
function addAdminSuffix(user) {  
  if (user.isAdmin) {  
    var n = user.name + '_ADMIN';  
    return n;  
  }  
  console.log(n); //not defined or undefined ???  
}  
addAdminSuffix({name: 'Gauthier'});
```

Javascript ES6 : les Classes

- Déclaration de classe
- Héritage
- Méthode static
- Constructeur
- Mot clé this obligatoire pour accéder aux membres
- New pour instancier
- Déclaration de constante avec *readonly*

➔ Un peu comme en Java !!

```
class Person {
  age() { return 30; }
}
class User extends Person {
  f1;
  login;
  private readonly version = 0.8;

  static defaultRole() { return 'ROLE_USER'; }

  constructor(login) {
    super();
    this.login = login;
    this.f1 = 12;
  }
  toString() { return 'Utilisateur : ' + this.login; }
}
```

```
let jntakpe = new User('jntakpe');
console.log(jntakpe.toString());
console.log(jntakpe.age());
console.log(User.defaultRole());
```

Javascript ES6 : GetTER et SetTER implicite

- Déclaration du get et/ou du set
 get monChamp
→ déclaration implicite du champ avec _ en prefix
- Le champ monChamp est accessible de l'extérieur. C'est le getter et le setter qui sont appelés.

```
class User {  
  constructor(login) { this._login = login; }  
  get login() {  
    return this._login.toLowerCase();  
  }  
  set login(login) {  
    if (login) {  
      this._login = login.trim();  
    }  
  }  
  
  toString() {  
    return 'Login : \'' + this._login + '\'';  
  }  
}  
  
const jntakpe = new User();  
jntakpe.login = ' JNtakpe';  
console.log(jntakpe.login);
```

Javascript ES6 : Arrow function

- Syntaxe raccourcie pour déclarer une fonction
- Plusieurs paramètres :
(p1,p2,p3) => { console.log(p1, p2, p3); }
- this = le parent

```
const a = [  
  "We're up all night 'til the sun",  
  "We're up all night to get some",  
  "We're up all night for good fun",  
  "We're up all night to get lucky"];
```

```
// Sans la syntaxe des fonctions fléchées  
const a2 = a.map( function(s){ return s.length } );  
// [31, 30, 31, 31]
```

```
// Avec, on a quelque chose de plus concis  
const a3 = a.map( s => s.length );  
// [31, 30, 31, 31]
```

Javascript ES6 : Destructuration

Extraire des valeurs d'un objet

Objet initial

```
const user = {  
  username: 'gpeel',  
  authority: 'admin',  
  address: { city: 'Aix' }  
};
```

Création des
constantes
username et role

```
// renaming field  
const { username, authority: role } = user;  
console.log(username);  
console.log(role);
```

Création des
constantes selon
un chemin

```
// cascading  
const { address: { city } } = user;  
console.log(city);
```

Extraire des valeurs d'un tableau

Objet initial

```
const arr = ['Jocelyn', 'Gauthier', 'Benjamin'];
```

```
const [a, b] = arr;  
console.log(a);  
console.log(b);
```

Création des
constantes a et
b



Javascript ES6 : REST

- Utilisation de ... pour indiquer le reste
- Dans les paramètres des fonctions

```
const formateurs = [];
```

```
function newPush(...formateursToPush) {  
  for (let formateur of formateursToPush) {  
    formateurs.push(formateur);  
  }  
}
```

```
newPush('Jocelyn', 'Gauthier', 'Sébastien');  
console.log(formateurs);
```

- Dans les tableaux

```
const [firstForm, ...others] = formateurs;  
console.log('First : ' + firstForm + ' ||| Rest: ' + others);
```



Javascript ES6 : String Template

- Les string template
- Evaluation du contenu des `${ ... }`
- Eviter de faire des +
- Multi ligne

```
const gpeel = {  
    firstname: 'Gauthier',  
    lastname: 'PEEL'  
};
```

```
const fullname = 'Monsieur ' + gpeel.firstname + ' ' + gpeel.lastname;  
console.log(fullname);
```

```
const tplFullname = `Monsieur ${gpeel.firstname} ${gpeel.lastname}`;  
console.log(tplFullname);
```

```
const multiLine = `

<h1>Multi</h1>  
    </div>`;  
console.log(multiLine);


```

TypeScript



TS

TypeScript : Généralités

- Typage statique et optionnel
- Superset de Javascript
- Injection par type
- Linter : analyse statique de code

**Typer son code permet de détecter
des problèmes à l'écriture**

→ C'est donc un impératif !

TypeScript : Interface

- Une interface classique

```
interface Writer {  
  write(message: string): void;  
}
```

```
interface User {  
  name: string;  
  age?: number;  
}
```

- Une interface anonyme

```
function updateAge(user: {birthdate: Date, age: number}): void {  
  if (moment().dayOfYear() === moment(user.birthdate).dayOfYear()) {  
    user.age++;  
  }  
}
```

TypeScript : Les variables

```
class TodoHelper {
  private loading = false;

  private todos: Todo[] = []

  faireQQchose() {
    const todo: Todo = { task : 'boire une biere', completed: false };
    let newTodo: Todo
  }
}

interface Todo {
  task: string;
  completed: Boolean
}
```

Typage implicite à boolean grâce à l'affectation d'une valeur

Variable de type tableau de Todo

Création d 'une constante de type Todo

Création d 'une variable de type Todo

TypeScript : Les fonctions

Typage du retour d'une fonction

```
function isWeekEnd(day: Days): boolean {  
    return day === Days.Saturday || day === Days.Sunday;  
}
```

Paramètre optionnel

```
function setAge(age: number?): void {  
    this.user.age = age;  
}
```

Paramètre par défaut

```
function setAge(age: number=20): void {  
    this.user.age = age;  
}
```

TypeScript : Les décorateurs

- Ajoutés à TypeScript pour Angular
- Permettent de modifier la cible
- Utilisés en Angular pour les metadonnées
- Proposés dans ES7

```
@Component({selector: 'cpm-hello'})  
export class HelloComponent {  
  
    @Input() name: string;  
  
    constructor() {  
        console.log(`Hello ${name}`);  
    }  
  
}
```



Quelques outils pour Angular





Les outils à installer

- Node : <https://nodejs.org/fr/> ➔ LTS version
- IDE : Visual Studio Code <https://code.visualstudio.com>
- `npm install -g typescript`



Angular CLI

- Boite à outils pour les projets Angular
- Github : <https://github.com/angular/angular-cli>
- `npm install @angular/cli -g`
- Génération d'un template de projet

```
ng new PROJECT-NAME  
cd PROJECT-NAME
```

- Lancement du serveur / simulateur

```
ng serve  
➔ http://localhost:4200/
```

- Génération de composant
ng g c path/MyComponent

Scaffold	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Guard	<code>ng g guard my-new-guard</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>



Angular CLI Tool

- Une autre CLI
- GitHub project : <https://github.com/littleuniversestudios/angular-cli-tools>
- `npm install angular-cli-tools -g`
- Template de projet :

Basic	https://github.com/littleuniversestudios/ng2-basic-seed
Bootstrap	https://github.com/littleuniversestudios/ng2-bootstrap-seed
Material Design	https://github.com/littleuniversestudios/ng2-material-seed
Firebase	https://github.com/littleuniversestudios/ng4-firebase-seed

- Custom projet template



Projet TODOLIST : création du squelette du projet





Projet TODOLIST

1. Installer NodeJS (Version ≥ 20) (ré-ouvrir un terminal pour la suite)
2. Vérifier votre version de NPM: `npm --version`
3. Installer les modules suivants :
 - `npm install -g @angular/cli`
 - `npm install -g typescript`
4. Créer votre projet todolist : `ng new todolist`
5. Se placer dans le répertoire : `cd todolist`
6. Lancer le simulateur : `npm start`
7. Visualiser l'application créer dans votre navigateur: <http://localhost:4200>
8. Aller voir le code source
 1. Fichier `index.html`
 2. Fichier `app.component.html`
9. Arrêter le simulateur et compiler l'application : `npm run build`
 - ➔ Regarder le résultat produit (html, js ...)

Sur un pc perso (mac ou Linux),
les « `npm install -g` » doivent être fait en root



Comment installer Node sur un linux en non ROOT ?

- Télécharger le binaire linux 64bits (tar.xz) de NodeJS sur <https://nodejs.org/en/download/current>
- Extraire le tar.xz dans un répertoire de votre home → `~/node-v22.6.0-linux-x64`
- Ajouter le répertoire bin du répertoire extrait dans votre PATH
Comment : Dans le fichier `~/.bashrc` ajouter la suivante en fin de fichier :

```
export PATH=~/ node-v22.6.0-linux-x64/bin:$PATH
```

- Ouvrir un nouveau terminal et taper
node --version

pour vérifier que vous avez la bonne version d'installée.

- Si vous avez des problèmes d'install de package (npm install ...)
`npm cache clean --force`



Les basiques





Introduction

- Framework web Javascript
- Open source
- Développé par Google
- depuis septembre 2016
- Multi plateformes
- Rapide
- Généralement en TypeScript
 - Transpileur vers ES5 (JS supporté par les navigateurs)
 - Simulateur/serveur web



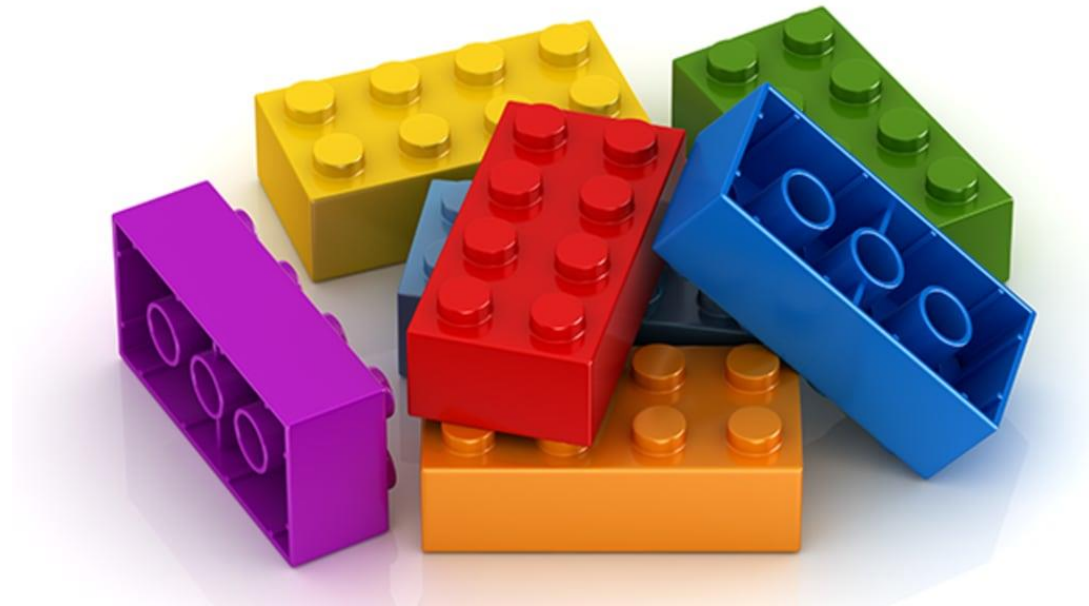
- 1 version tous les 6 mois → migrer régulièrement
 - Version 17 : beaucoup de nouveautés !
 - Version 20 actuellement





Des composants

- Orienté composant (comme ReactJS, Aurelia, Vue.js)
- Application = assemblage de composants
- Intégration des web components
- Mais aussi des services, des directives, des guards ...





Composant

- Composant =
 - Une classe (fichier .ts)
 - + Un template html
 - [+ des css]
- Template = fichier html séparé | inline
- CSS = fichiers css séparés | inline
- selector : nom pour utiliser le composant depuis le template d'un autre composant
 - ➔ Utilisation d'un composant par un autre template
- Les composants doivent être déclarés dans le module. (Auto via la CLI)

Fichier de la **classe** struct-component.ts

```
@Component({  
  selector: 'app-struct',  
  templateUrl: './struct.component.html',  
  styleUrls: ['./struct.component.css']  
})  
export class StructComponent {  
  colors: string[] = ['red', 'blue', 'green'];  
}
```

Fichier **template** struct.component.html

```
<div class="too">  
  <h2>Colors</h2>  
  <ul>  
    <li>{{ colors[0] }}</li>  
    <li>{{ colors[1] }}</li>  
  </ul>  
</div>
```

Fichier **css** struct.component.css

```
.too {  
  text-align: center;  
}
```



Composant : Cycle de vie

- Le composant implémente une interface pour être prévenu de son cycle de vie.
- Ci-contre un tableau des plus importants
- Initialisation du composant : Pas dans le constructor Mais dans `ngOnInit()` { ... }

Interface	Méthode	Description
OnChange	<code>ngOnChange()</code>	Un des paramètres du composant a changé. Peut être appelé plusieurs fois.
OnInit	<code>ngOnInit()</code>	Le composant est initialisé. Est appelé une seule fois.
AfterViewInit	<code>ngAfterViewInit()</code>	Les vues du composant et de ses enfants sont initialisées.
OnDestroy	<code>ngOnDestroy()</code>	Le composant va être détruit par Angular.

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

@Component({
  selector: 'app-struct',
  templateUrl: './struct.component.html',
  styleUrls: ['./struct.component.css']
})
export class StructComponent implements OnInit {
  colors: string[] = [];

  public constructor(private myService: MyService) {}

  public ngOnInit() {
    this.colors = this.myService.getColors();
  }
}
```



Template d'un composant

- Les doubles moustaches: **{{ }}**
 - Expression JS évaluée
- Les champs/méthodes private de la classe ne sont pas visibles par le template
- Utilisation de **?** pour l'accès aux champs non encore chargés (lazy loading) ou optionnel

Exemple avec template et css **inlinés**

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-interpolation',
  template: `<h1>{{titre}}</h1>
    <h3>Bonjour {{user.firstName + ' ' + user.lastName}}</h3>
    <p>Votre numéro de téléphone : {{user.contact?.phone}}</p>
  styles: ['h1 { font-weight: normal; }']
})
export class InterpolationComponent {
  titre: string = 'Interpolation';
  user: any = {
    firstName: 'Sebastien',
    lastName: 'Chassande'
  };
  colors: string[] = ['rouge', 'vert', 'bleu'];
  toto: string;
}
```



Boucle @for

- Dans un template de composant
- Le bloc contenu dans le @for sera dupliqué autant de fois qu'il y a d'élément dans un tableau
- Track : définit quelle partie de élément, Angular doit surveiller le changement.
- Possibilité de connaitre certaines informations en déclarant des variables dans le for
 - Exemple l'index: ;let myIndexVar = index;
 - D'autres infos : \$first, \$last, \$odd, \$even,\$count
- Migration depuis l'ancienne syntaxe (ngFor)
Directive pour dupliquer un composant (html ou custom)

```
@Component({
  template: `<ul>
    @for (color of colors; track color; let idx = $index) {
      <li>{{idx}}:{{ color }}</li>
    }
    @empty {
      <li>No items found</li>
    }
  </ul>`,
})
class ExampleComponent {
  colors = ["Red", "Blue", "White"];
}
```



@if

- Dans un template de composant
- Pour afficher ou non des composants (html ou custom)
- Le contenu du @If sera créer ou non selon la valeur de l'expression
- Possibilité d'écrire des @else et des @else if

```
@Component({  
  template: `  
    @if (showHello) {  
      <h2>Hello</h2>  
    }  
    @else if (showGoodbye) {  
      <h2>Goodbye</h2>  
    }  
    @else {  
      <h2>See you later</h2>  
    }  
  `,  
})  
class Test {  
  showHello: boolean = true;  
  showGoodbye: boolean = false;  
}
```



Organisation du code

Réutilisation

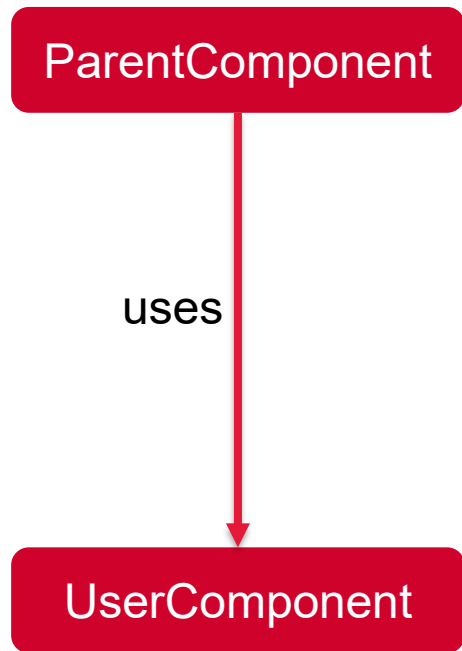
- Composant = unité de réutilisation dans une application
- Module = unité de réutilisation entre des applications

Organisation du code source

- Un sous module par grand partie de l'application
- Si pas inline alors un répertoire par composant (html, .ts, .css) (Comportement de la CLI)
- Un répertoire pour tous les composants
- Un répertoire pour tous les services
- Un répertoire pour les classes/interface du modèle de données



Utilisation d'un composant par un autre composant



```
import {Component} from '@angular/core';
@Component({
  selector: 'app-interpolation',
  template: `

# {{titre}}

 <app-user-component [myUser]="user"></ app-user-component>`})
export class ParentComponent {
  titre: string = 'Interpolation';
  user: User;
  ngOnInit() { this.user = { firstName: 'Sebastien', lastName: 'Chassande' }; }
}
```

Instanciación du composant enfant

Affectation de l'attribut **myUser** avec la valeur de la variable **user**

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-user-component',
  template: `

### Bonjour {{myUser.firstName}} {{myUser.lastName}}</h3>`}) export class UserComponent { @Input() myUser: User; }


```

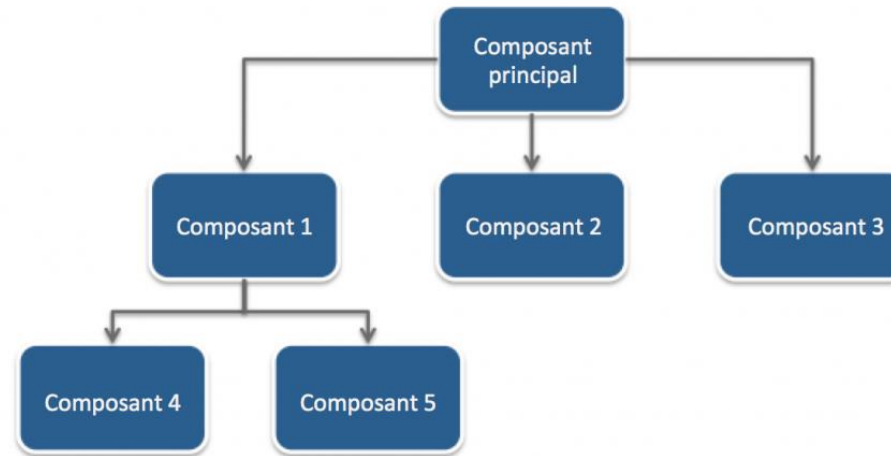
Nom de la balise html pour instancier le composant enfant

Utilisation du champ 'myUser' dans le template

La structure d'une application : **Un arbre**

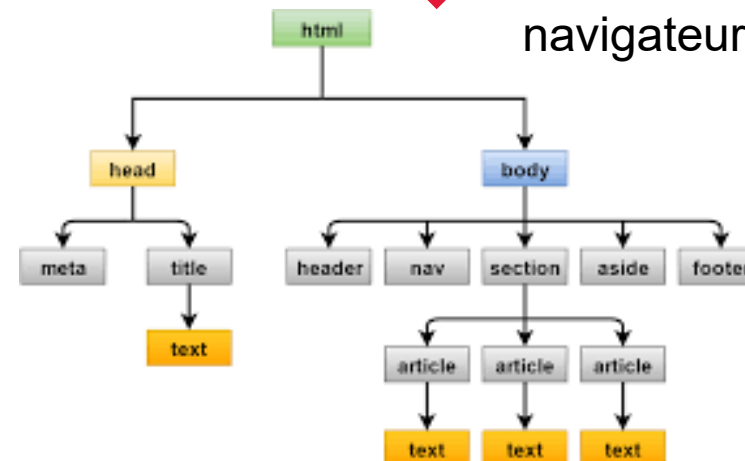
Un arbre de composants :

- Des composants faits maison
- Des balises html
- Des web component



Le javascript modifie l'arbre DOM dans le navigateur

L'arbre DOM représentant les nœuds réels



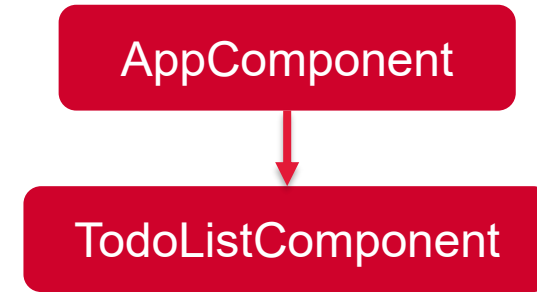


Projet TODOLIST : Mes premiers composants



Etape 1 : Affiche une todo Liste

1. Créer l'interface Todo comme modèle:
 1. `ng g class model/Todo`
Cela crée le fichier `src/app/model/Todo.ts`
 2. Changer le type de class en interface
 3. Dans l'interface ajouter les champs suivants:
`id?: number; // le ? Rend optionnel le champ.`
`name: string;`
`completed:boolean`
2. Créer un composant TodoList
 1. `ng g c component/TodoList`
Cela crée les fichiers `src/app/component/todo-list/todo-list-component.*` (ts, html, css, spec.ts)
3. Vider le template principal de l'application fichier (`app.component.html`) puis y ajouter l'utilisation du composant 'app-todo-list' :
`<app-todo-list></app-todo-list>`



4. Compléter le composant TodoList, partie .ts
 1. Créer une variable
`todos: Todo[] = []`
 2. Initialiser la variable todos dans un `ngOnInit` avec un tableau de 3 Todo de votre choix
5. Compléter le composant TodoList, partie .html
 1. Dans le template, afficher la liste des todos en appliquant un `ngFor` sur un composant Todo

Étape 2 : Créer et utiliser un composant dédié pour afficher chaque TODO



1. Créer le composant Todo
ng g c component/Todo
Cela crée les fichiers
src/app/component/todo/todo-component.* (ts, html, css, spec.ts)
2. Ajouter une variable dans la classe (.ts) du composant créé
3. Modifier la vue du composant (.html) pour afficher le nom de la tâche et d'une checkbox pour changer le flag completed

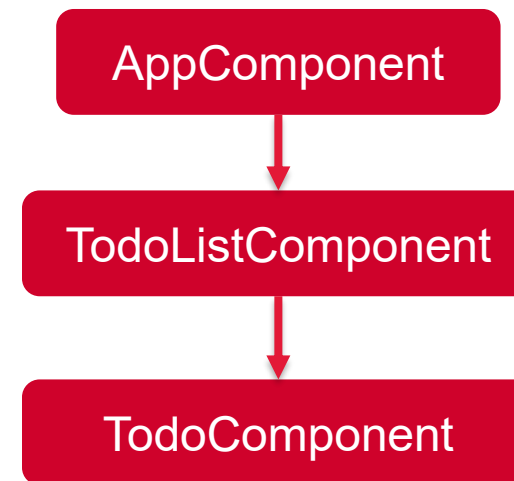
Tips: Pour réagir à un click sur une balise:

```
<li (click)="myMethod()">
```

La méthode est à implémenter dans la partie typescript.

4. Depuis la vue du composant (.html) TodoList ajouter l'appel au composant Todo dans la boucle @for

```
</ul>  
  @for(t of todos, track t.id) {  
    <li>  
      <app-todo [myTodo]="t"> </app-todo>  
    </li>  
  }  
</ul>
```





C'est quoi un signal ?

Définition :

« Conteneur » de données permettant de signaler la modification des données aux utilisateurs de la donnée.

Intérêts / Bénéfice :

- ➔ Permet de signaler à la vue (html) que la donnée a été modifiée (ts).
- ➔ Limite le recalcul de ce qui doit changer dans la vue.
- ➔ Performance / fluidité

Remplacement de Zone.js qui n'est pas performant.

A utiliser le + possible !!!

<https://blog.angular-university.io/angular-signal-components/>

Les signaux de base

Signal

```
const count = signal(0);

// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());

count.set(3);

// Increment the count by 1.
count.update(value => value + 1);
```

- Il faut toujours appeler la fonction pour obtenir la valeur
- Modifiable par set ou update

Effect

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

- L'effet ne renvoie pas de valeur.
- Il permet de réagir. Idem que computed, il est appelé lors qu'un signal a change
- Il peut agir sur l'application

Calculé

```
const showCount = signal(false);
const count = signal(0);
const conditionalCount = computed(() => {
  if (showCount()) {
    return `The count is ${count()}.`;
  } else {
    return 'Nothing to see here!';
  }
});
```

- Il est recalculé automatiquement lorsqu'un signal utilisé à l'intérieur change par une fonction
- Ne peut être modifié autrement que par le recalcul
- /\ aux chemins (if)
→ mettre l'appel de tous les signaux au début

Les signaux entrant d'un composant: **Input**

Enfant :

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

@Component({
  selector: "book",
  standalone: true,
  template: `<div class="book-card">
    <b>{{ book().title }}</b>
    <div>{{ book().synopsis }}</div>
  </div> `,
  styles: ``,
})
class BookComponent {
  book = input.required<Book>();
}
```

Parent :

```
<book [book]="angularBook" />
```

- Paramètre d'entrée d'un composant sous la forme d'un signal
- Non modifiable
- *Required* pour indiquer qu'il est requis

Les signaux sortant d'un composant: **Output**

Enfant :

```
@Component({
  selector: "book",
  standalone: true,
  template: `<div class="book-card">
    <b>{{ book()?.title }}</b>
    <div>{{ book()?.synopsis }}</div>
    <button (click)="onDelete()">Delete Book</button>
  </div>`,
})
class BookComponent {
  deleteBook = output<Book>();
  book = input.required<Book>();
  onDelete() {
    this.deleteBook.emit({
      title: "Angular Deep Dive",
      synopsis: "A deep dive into Angular core concepts",
    });
  }
}
```

Parent :

HTML `<book (deleteBook)="deleteBookEvent($event)" />`

TS

```
deleteBookEvent(book: Book) {
  console.log(book);
}
```

- Permet diffuser un **événement** à son **parent**
- Réaction à un événement graphique (ici click)
- Emission d'un objet
- La méthode du composant « parent » est appelée avec l'objet
- Typage !

Les signaux entrant/sortant d'un composant: **Model**

Enfant :

```
@Component({
  selector: "book",
  standalone: true,
  template: `<div class="book-card">
    <b>{{ book()?.title }}</b>
    <div>{{ book()?.synopsis }}</div>
    <button (click)="changeTitle()">
      Change title
    </button>
  </div> `,
  styles: `
`,
})
export class BookComponent {
  book = model<Book>();

  changeTitle() {
    this.book.update((book) => {
      if (!book) return;
      book.title = "New title";
      return book;
    });
  }
}
```

Parent :

```
@Component({
  selector: "booklist",
  standalone: true,
  template: `
    <div>
      <book [(book)]="book" />
    </div>
    <div>
      <b>Parent</b> <br />
      <div>{{ book().title }}</div>
      <div>{{ book().synopsis }}</div>

      <button (click)="changeSynopsis()">
        Change Synopsis
      </button>
    </div>
  `,
  styles: ``,
  imports: [BookComponent],
})
export class BookListComponent {
  book = signal<Book>({
    title: "Angular Core Deep Dive",
    synopsis: "Deep dive to advanced features of Angular",
  });
}
```

Le nom de la variable book dans BookComponent (Enfant) est donc le nom de l'attribut dans l'appel fait par le parent

L'attribut est valorisé par la donnée contenue dans un attribut de la classe du parent. Ici un signal « book »

- Les modifications sont propagées dans les 2 sens
 - Du parent vers l'enfant
 - De l'enfant vers le parent



Projet TODOLIST : Passage en signal

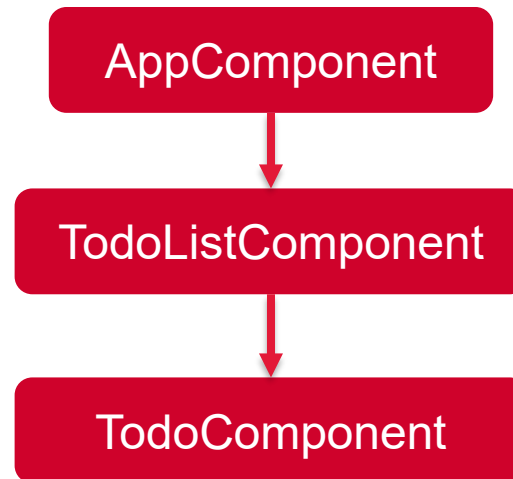


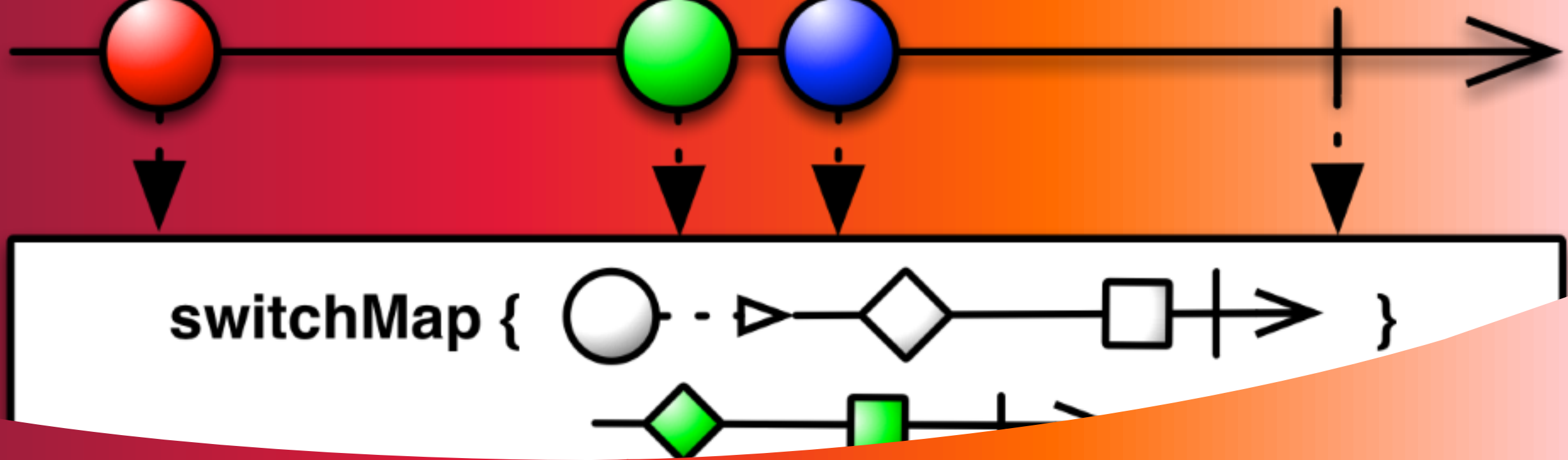
Passage en signal



1. Transformer les variables des parties .ts en signaux

1. La liste des todos das TodoListComponent
2. La variable myTodo TodoComponent





Les observables



RXJS

- Reactive extensions - Programmation réactive
- Flux événements
 - une valeur,
 - une erreur ou
 - une terminaison
- Asynchrone
- Existent dans de nombreux langage (JS, Java, .NET ...)
- Observables != Promises
 - Ça ressemble
 - Un observable peut renvoyer plusieurs valeurs
 - Un observable peut « rester en vie » pour envoyer un flux « infini de données »

```
interface Observable<T> {  
    Subscription subscribe(Observer s);  
}
```

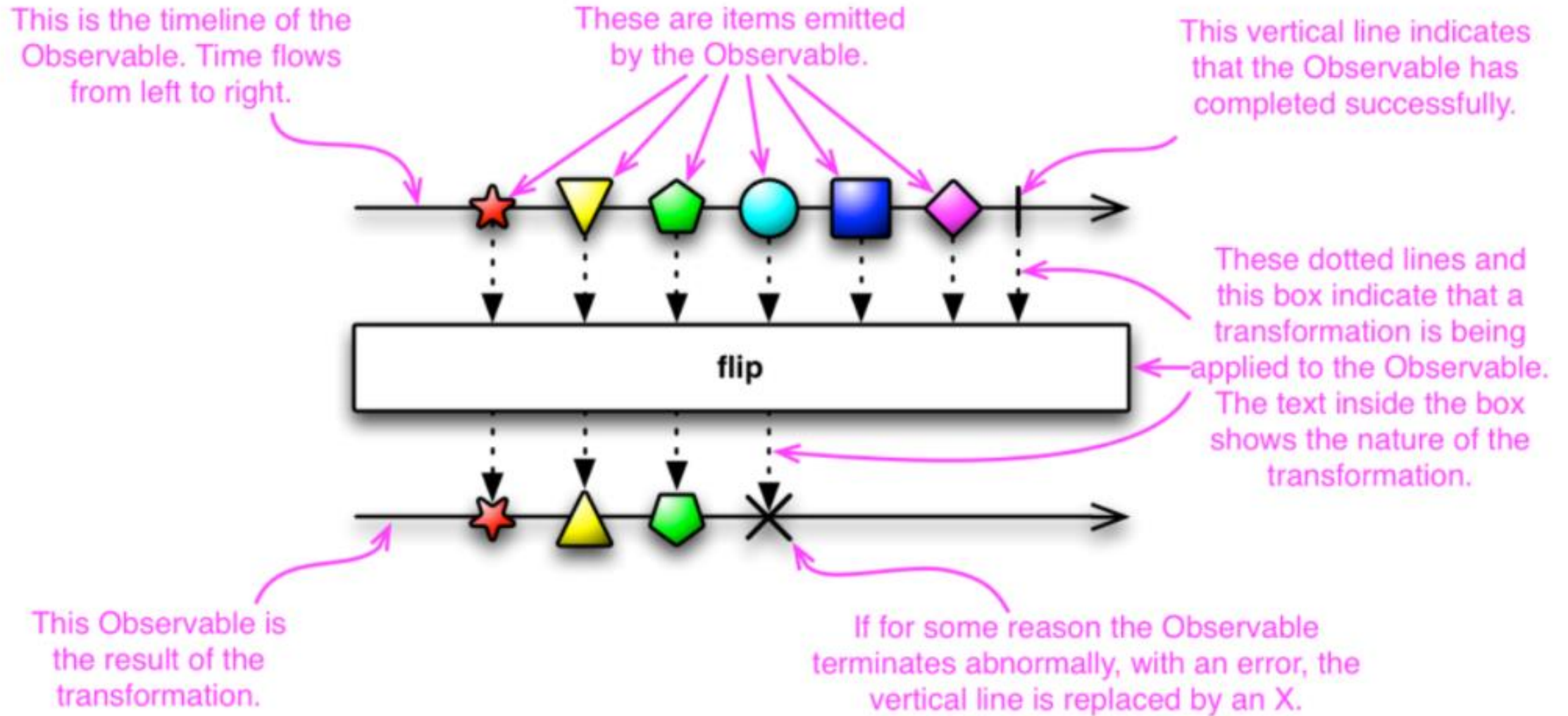
```
interface Observer<T> {  
    void onNext(T t);  
    void onError(Throwable t);  
    void onCompleted();  
}
```



Les marbles

- Marbles : documentation des opérateurs pour manipuler le flux d'événements
- Des exemples d'opérateur pour manipuler les observables
 - Création d'un observable: From, Of, interval, timer
 - Combinaison : concat, merge, combineLatest
 - Filtrage : distinct, filter, first, last, elementAt, find
 - Opération mathématique : count, max, min, reduce
 - Transformation : map, repeat,
 - Utilitaire : delay
- <http://rxmarbles.com/>

Comprendre un diagramme Marble





Opérateur : MAP

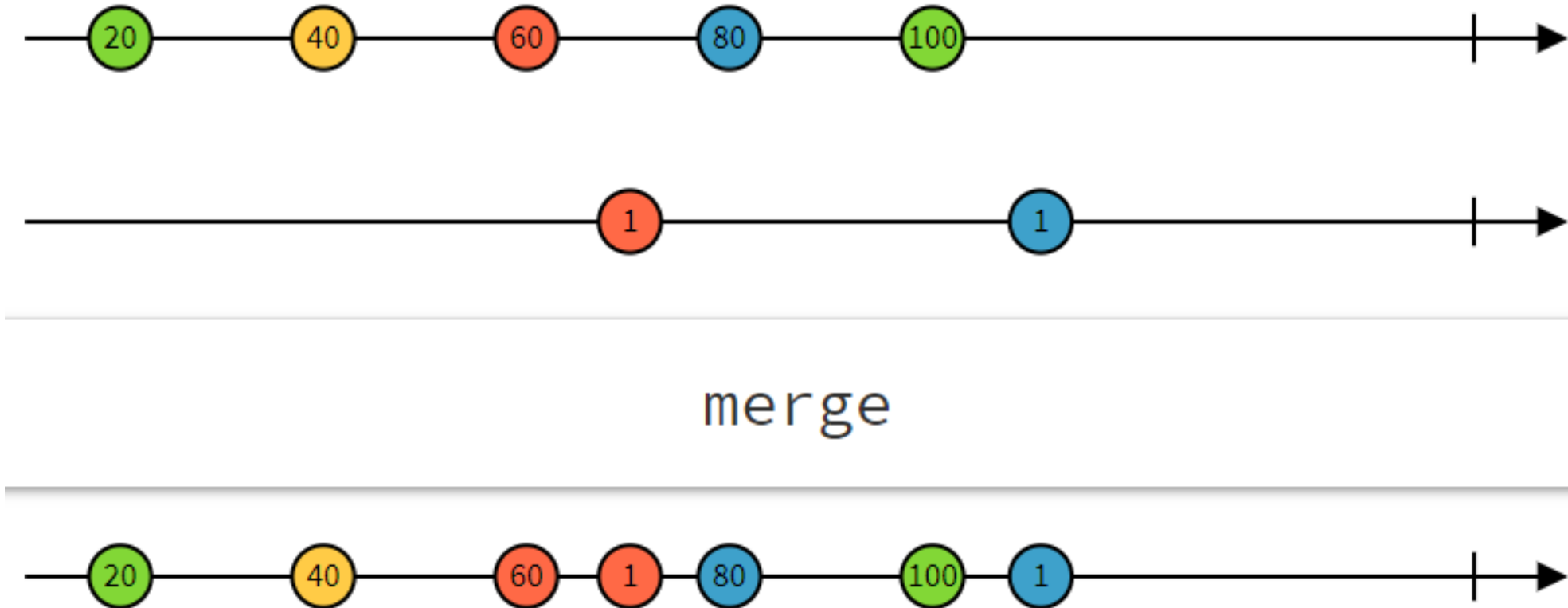


```
map(x => 10 * x)
```



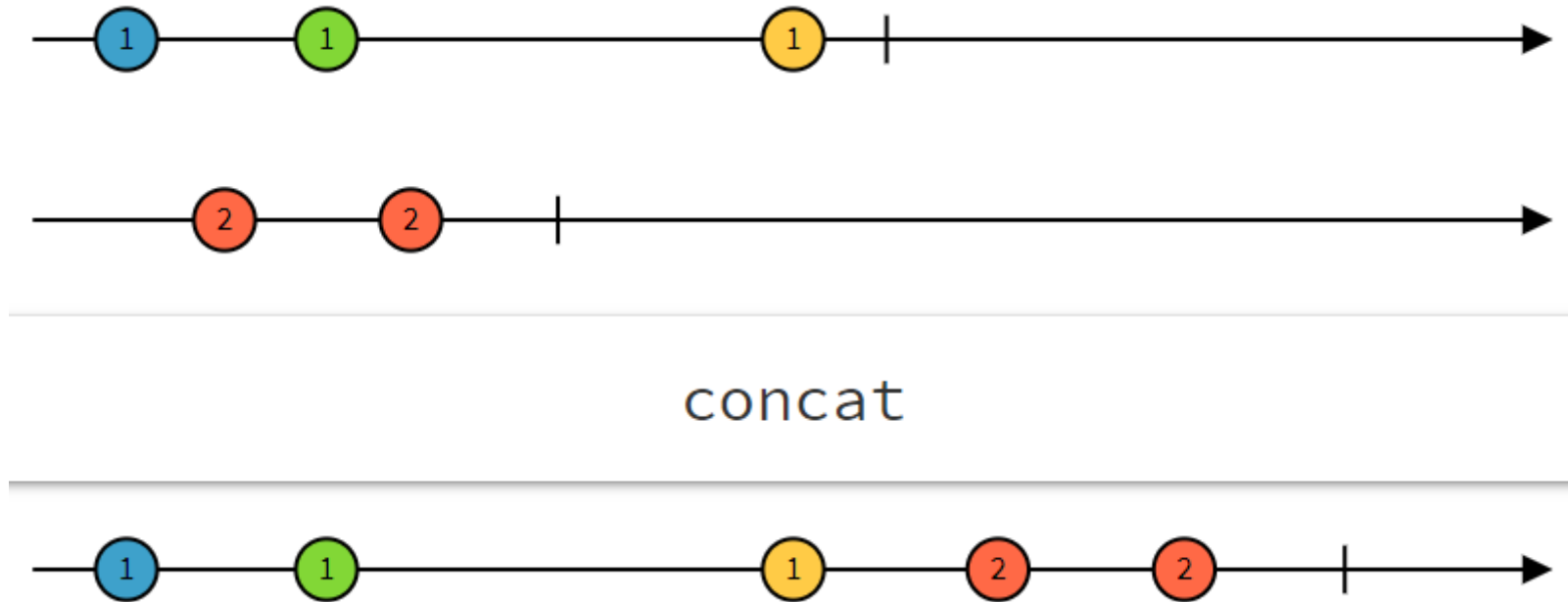


Opérateur : MERGE





Opérateur : Concat





Quand faut-il utiliser un observable ?

Dès qu'une opération est asynchrone !

- Exemple : Appel du serveur

Le nombre de données à obtenir est inconnu au départ

➔ En angular, les Services



Comment faut-il utiliser un observable ?

- Il faut souscrire/s'abonner via la méthode `subscribe` sinon rien de ne se passe !
- Il y a 3 fonctions à donner en paramètres
 1. La fonction pour traiter un événement
 2. La fonction pour traiter le cas d'erreur
 3. La fonction pour traiter la fin de l'observable

```
// La création d'un observable
this.data = new Observable(observer => {
  setTimeout(() => { observer.next(42); }, 1000);
  setTimeout(() => { observer.next(43); }, 2000);
  setTimeout(() => { observer.complete(); }, 3000);
});

// L'inscription et la consommation d'un observable
let subscription = this.data.subscribe(
  value => this.values.push(value),
  error => this.anyErrors = true,
  () => this.finished = true );
}
```

Comment composer des actions avec observables ?

- Appel d'une 1^{ère} fonction async puis utilisation du pipe pour chaîner les actions suivantes.
- Le **pipe** renvoie l'observable qui chaine les actions
- L'opérateur **map** renvoie une valeur
- L'opérateur **flatMap** (mergeMap) renvoie un observable de valeurs
- L'opérateur **catchError** attrape les erreurs

```
private loadCompetition(): Observable<Competition> {  
    this.loading = true;  
    // load id from url path  
    return this.route.paramMap.pipe(  
        // load competition from the id  
        flatMap((paramMap) => this.competitionService.get(paramMap.get('id'))),  
        map((rcompetition) => {  
            this.competition = rcompetition.data;  
            if (!this.competition) {  
                // the competition has not been found => create it  
                this.createCompetition();  
            }  
            return this.competition;  
        })),  
        // load referees  
        flatMap(() => this.loadReferees()),  
        // load coaches  
        flatMap(() => this.loadCoaches()),  
        catchError((err) => {  
            console.log('loadCompetition error: ', err);  
            this.loading = false;  
            return of(this.competition);  
        })),  
        map(() => {  
            this.loading = false;  
            return this.competition;  
        })  
    );  
}
```

Chainage avec le pipe

Fonction synchrone

Fonction asynchrone renvoyant un observable



Les services





Un Service Angular C'est quoi ?

- Service = simple classe
- Singleton
- Déclaré dans un module
- Rôle
 - Stockage de données (Panier)
 - Appel du backend
 - Algorithme métier
- Injectable dans les composants ou d'autres services grâce à **@Injectable()**

Exemple d'un service qui appelle une API

```
import {Injectable} from '@angular/core';  
import { Observable, of } from 'rxjs';  
import { map } from 'rxjs/operators';  
import { HttpClient, HttpResponse } from '@angular/common/http';
```

Import de
l'opérateur **map**

@Injectable()

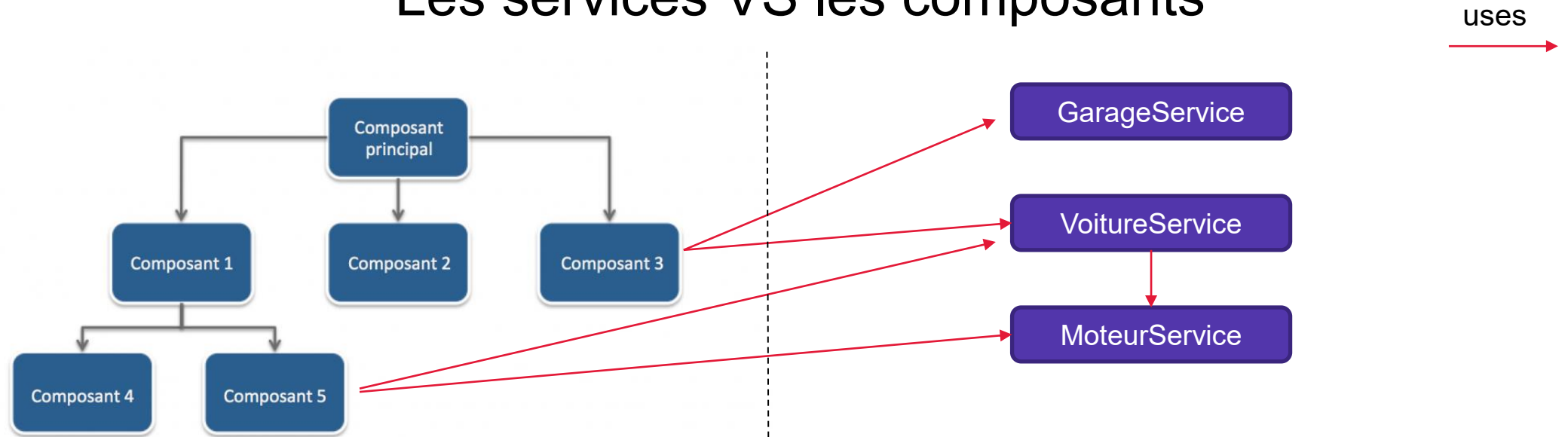
```
export class GithubService {
```

Injection du service
HttpClient dans le
service **Githubservice**

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

```
  findReposForUser(username: string): Observable<String> {  
    return  
    this.http.get(`http://api.github.com/users/${username}/repos`)  
      .pipe(  
        map((body:any) => body.name)  
      );  
  }  
}
```

Les services VS les composants



- Un composant est un objet graphique
- Il peut être instancié plusieurs fois dans l'arbre des composants qui représente la page
- La liaison entre les composants est via le code HTML majoritairement : utilisation de la balise html.

- Un service est un objet contenant seulement du code métier ou d'appel de service web
- Un service est singleton: 1 seule instance
- Les services sont injectés dans les composants ou d'autres services via le constructeur



Le service HTTP

- Service Angular nommé HttpClient permettant de récupérer des données
- Réalise les requêtes AJAX en utilisant XMLHttpRequest
- Renvoi des observables
- Méthode : get, post, put, delete, patch, head
- Transformation du résultat grâce aux opérateurs d'Observable

```
import {Injectable} from '@angular/core';  
import { Observable, of } from 'rxjs';  
import { map } from 'rxjs/operators';  
import { HttpClient, HttpResponse } from '@angular/common/http';
```

```
@Injectable()  
export class GithubService {  
  
    constructor(private http: HttpClient) {}  
  
    findReposForUser(username: string): Observable<any> {  
        return this.http.get(  
            `http://api.github.com/users/${username}/repos`  
        ).pipe(  
            map((body:any) => body.address)  
        );  
    }  
}
```

!! Dans le module de l'application (app.module.ts), il faut ajouter provideHttpClient() dans la section providers.

```
@NgModule({  
  imports: [  
    BrowserModule,  
    // Remove the module  
    ...  
  ],  
  declarations: [  
    AppComponent,  
    ...  
  ],  
  providers: [provideHttpClient()],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule {}
```



Service HTTP : les paramètres dans le Header

```
import { of } from 'rxjs';

public getRequestOptions(): RequestOptions {
  const headers: any = {
    'Content-Type': 'application/json',
    'version': '1.0',
    'X-Auth-Token': 'ZERF43534R4RCZ4TRTZERTZER434'
  }
  return new RequestOptions({ headers: new Headers(headers) });
}

public askResetPassword(email: string): Observable<Response> {
  this.log.d('askResetPassword(', email, ')');
  return this.http.get(`${env.serverUrl}/users/resetPassword/${email}`,
    this.getRequestOptions()).pipe(
    map(body => { return { errorCode: 0 }; }),
    catchError(err => { return of({ errorCode: err.json() }); })
  );
}
```



Projet TODOLIST : Mise en place d'un service



Avec un backend de stockage des TODOs



Le backend REST

- Une simple application Java spring boot avec une base de données mémoire (H2)
- Disponible en téléchargement sur <https://im2ag-moodle.univ-grenoble-alpes.fr> ^{ou} <http://chassande.fr/formation/ecom/>
- Télécharger le Jar exécutable backend-0.0.1-SNAPSHOT.jar
!! Prendre la version compatible avec votre version Java
- Lancer l'exécutable sur votre ordinateur: `java -jar backend-0.0.1-SNAPSHOT.jar`

Lorsque l'application est lancée :

- URL de l'API : <http://localhost:8080/todos>
- Méthode disponible : GET, POST, DELETE
- Documentation de l'API en Swagger :
 - Java 17+: <http://localhost:8080/swagger-ui/index.html#/Todo%20Entity>
 - Java 11: <http://localhost:8080/swagger-ui.html#/Todo32Entity>
 - Permet aussi de faire des requêtes pour peupler la base (qui est vide à chaque démarrage)

Ajouter un service Angular



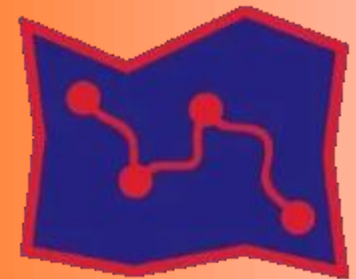
- Ajouter un service angular pour gérer les Todo: `ng g s service/Todo`
- Déclarer ce service dans le `app.module.ts` dans le tableau des 'providers'
- Créer une fonction asynchrone dans le service qui utilise le service `http` pour appeler le backend

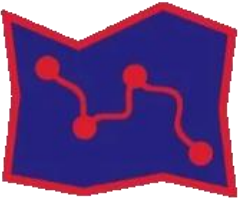
```
public all(): Observable<Todo[]> { ... }
```
- Dans le composant de liste, utiliser le service
 - Dans le constructeur : injecter le service en déclarant un paramètre du constructeur
 - Dans le `ngOnInit()` : appeler le service pour récupérer les données. Attention l'appel est asynchrone !
- Remplir votre base en effectuant qq requêtes `POST /todos` via Swagger



Angular Component Router

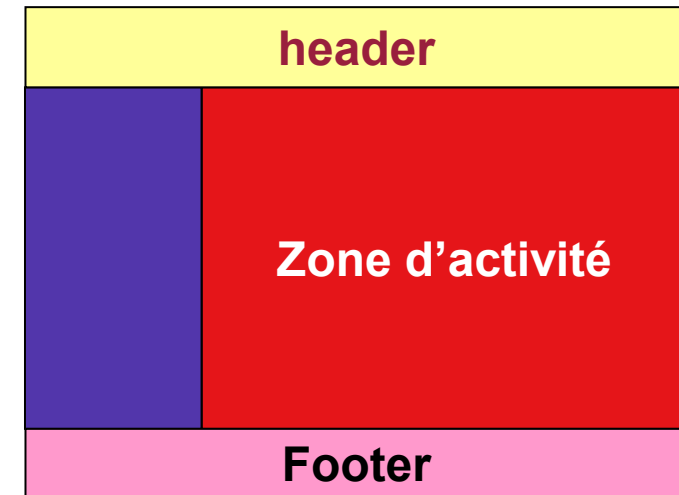
Routing





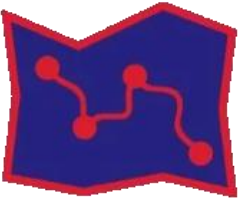
Routage entre « Page »

- SPA : Single Page association
→ Une seule page qui s'auto modifie via du JS
- Une application = { activités }
- Une seule activité en même temps
- Routage d'activité en activité
- Utilisation de l'URL après le # pour identifier l'activité et ses paramètres
 - En prod on configure le serveur pour une SPA afin d'éviter le #.
- <https://angular.io/docs/ts/latest/guide/router.html>



Dans le html (sous le composant principal app.component.html)

`<router-outlet></router-outlet>`



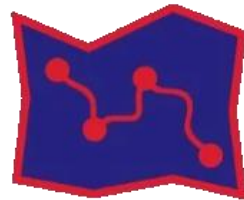
Configuration des routes (mode standalone)

- Dans le fichier app.routes.ts
- Association des routes avec des composants

```
export const routes: Routes = [  
  { path: 'home', component: HomeComponent },  
  
  { path: 'user/login', component: UserLoginComponent },  
  { path: 'user/create', component: UserCreateComponent },  
  
  { path: 'tournement', component: TournamentListComponent },  
  
  { path: "", redirectTo: '/home', pathMatch: 'full' }  
];
```

!! les urls ici ne sont pas déclarées avec le / de début

Cas avec module



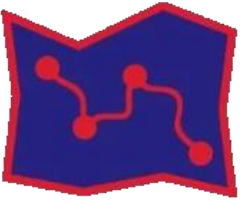
Configuration des routes

Dans le app-routing.module.ts
(sinon app.modules.ts)

- Association des routes avec des composants
Note: Les urls ici ne commencent pas par /
- Déclaration du module

```
import { RouterModule, Routes } from '@angular/router';
export const routes: Routes = [
  { path: 'foo',      component: FooComponent },
  { path: 'bar/:id', component: BarDetailComponent },
  { path: '**', redirectTo: 'foo' }
];

@NgModule({
  declarations: [ AppComponent, ... ],
  imports: [ BrowserModule,
            FormsModule,
            HttpClientModule,
            RouterModule.forRoot(routes),
            bootstrap: [AppComponent] })
export class AppModule { }
```



Navigation

Dans un template

```
<a routerLink="/bar/12">Bar 12</a>
```

Dans du code .ts (composant/service)

- *Injection du router dans le component*
`constructor(private router: Router) {}`

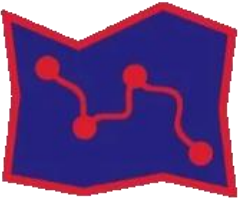
- *Puis utilisation dans une méthode*

```
this.router.navigate(['/bar', 12]);
```

- Contrôler la navigation avec un Guard

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate() {
    console.log('AuthGuard#canActivate called');
    return Math.random() * 10 > 5;
  }
}
```

```
import { AuthGuard } from '../auth-guard.service';
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard]
  }
];
```



Navigation : URL avec paramètres

Déclarer la route avec les paramètres

```
const appRoutes: Routes = [  
  {path: 'resetPassword/:userId/token/:token', component: MyComponent},  
  {path: '**', redirectTo: 'accueil'}  
];
```

Récupérer la valeur des paramètres dans le composant

```
import { ActivatedRoute, Params } from '@angular/router';  
export class ResetPasswordComponent implements OnInit {  
  public userId = 0;  
  public token: string;  
  
  constructor(private activatedRoute: ActivatedRoute) { }  
  
  ngOnInit() {  
    this.activatedRoute.params.subscribe((params: Params) => {  
      this.userId = params['userId'];  
      this.token = params['token'];  
    });  
  }  
}
```



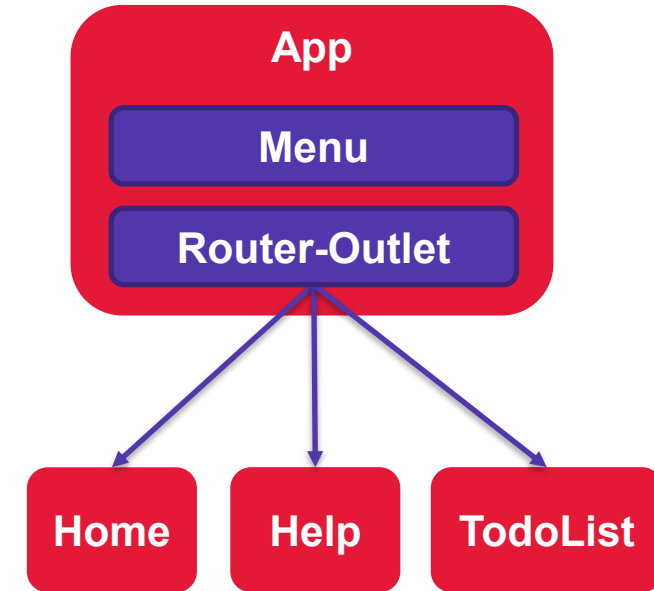
Projet TODOLIST : Plusieurs pages avec du routage





Mettre en place plusieurs pages

- Ajouter un composant Home: `ng g c component/Home`
- Ajouter un composant Help: `ng g c component/Help`
- Dans le composant principal (`app.component`) ajouter :
 - Un menu qui permet de naviguer sur 3 pages : home, help TodoList
Des simples liens `<a ...></a>` suffisent
 - Une zone de service `<router-outlet>`
- Configurer le routage dans le `app-routing.module.ts`
 - Déclarer les routes/urls vers les composants





Les formulaires



Un formulaire en Angular c'est du HTML

- Utilisation des balises input, form ...
- Problématique : Lire et injecter des données dans les balises HTML
- Solutions :
 - Dirigée par le template : form
 - Dirigée par le template : Two way binding  Solution classique
 - Formulaire réactif



Formulaire dirigé par le **template** : Two way Binding

Le template fait le lien avec le modèle accessible depuis les champs de la classe
[(ngModel)]="a.b.monChamp"

```
<h2>Sign up template form</h2>
<form (ngSubmit)="register()">
  <div class="form-group">
    <label for="username">Nom d'utilisateur</label>
    <input type="text" class="form-control" id="username"
      name="username" [(ngModel)]="user.name">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password"
      name="password" [(ngModel)]="user.password">
  </div>
  <button type="submit">Se connecter</button>
</form>
```

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

@Component({...})
export class SignupFormComponent {
  user: User;
  register() {
    console.log(user.name);
    // ...
  }
}

export interface User {
  name: string;
  password: string;
}
```



Formulaire dirigé par le **template** : La validation

La validation des champs est définie dans le template par des attributs HTML5

Définition d'une variable pour accéder à l'objet FormControl de chaque champ :

`#myfield="ngModel"`

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

<div *ngIf="name.invalid && (name.dirty || name.touched)" class="alert alert-danger">
  <div *ngIf="name.errors.required">Name is required.</div>
  <div *ngIf="name.errors.minlength">Name must be at least 4 characters long.</div>
</div>
```



Formulaire

Un formulaire est représenté par un *FormGroup*

- *valid* : si le formulaire est valide
- *errors* : objet contenant les erreurs du formulaire
- *dirty* : false jusqu'à ce que l'utilisateur modifie un des champs du formulaire
- *pristine* : opposé de dirty
- *touched* : false jusqu'à ce que l'utilisateur soit entré et sorti d'un des champs du formulaire
- *untouched* : opposé de touched
- *value* : la valeur du formulaire sous forme de clé/valeur avec le nom du champ en clé
- *valueChanges* : un observable emettant la valeur du formulaire à chaque changement sur un des champs



Champ de formulaire

Un champ de formulaire est un *FormControl* :

- *valid* : si le champ est valide avec les validations et contraintes qui lui sont appliquées
- *errors* : objet contenant les erreurs du champ
- *dirty* : false jusqu'à ce que l'utilisateur modifie le champ
- *pristine* : opposé de dirty
- *touched* : false jusqu'à ce que l'utilisateur soit entré et sorti du champ
- *untouched* : opposé de touched
- *value* : la valeur du champ
- *valueChanges* : un observable emettant la valeur du champ à chaque changement



Projet TODOLIST : une page d'édition création dun TODO



Projet TODOLIST : une page d'édition/création d'un TODO



Ajouter une page d'édition d'un Todo existant ou nouveau contenant formulaire de Todo : le nom et completed

Consignes :

- Créer un composant dédié qui sait faire l'édition et la création en même temps
- La page est activable selon 2 urls :
 - Pour la création: /todo
 - Pour l'édition: /todo/:todold
- Dans le ngOnInit
 - Récupérer le todold dans l'URL s'il y en a un
 - S'il y a un todold alors appeler le backend pour récupérer l'objet via la méthode GET (voir swagger du backend)
 - Faire une méthode get(todold) dans le service TodoService
- Permettre la sauvergarde du todo (nouveau ou existant) en appelant le backend
 - Création ➔ Appel du serveur /todo, Méthode http POST
 - Modification ➔ Appel du serveur /todo/\${todo.id}, Méthode http PUT
 - Faire une seule méthode save(Todo) dans le service TodoService qui s'adapter
- Dans la page TodoList, ajouter les liens pour éditer un Todo existant et un lien pour ajouter un nouveau todo



Conclusion



Qu'est-ce qu'on a appris ?

- Des évolution JS : des classes
- Ecrire du code TS qui est compilé en JS pour être exécuté par le navigateur
- Ecrire des composants qui s'appellent entre eux
- Utiliser la CLI pour programmer et tester
- Un peu de programmation réactive
- Accéder à des services distants en http
- Mettre en place de routage entre pages
- Faire des formulaires basiques