# Angular

Un framework pour créer des applications web et mobile

# Typescript

Il s'agit de **javascript typé** c'est à dire dont on précise le type (string, boolean, …) des variables et paramètres.

## 1 - Les types de base

```ts
let isDone: boolean = false;
let age: number = 6;
let name: string = 'Emile Zola';
let list: number[] = [7, 21, 8, 4];

enum Color {
  Red = 1,
  Green = 2,
  Blue = 4
}
let c: Color = Color.Green;

let notSure: any = 4;
notSure = 'maybe a string instead';
notSure = false;

let unusable: void = undefined;
```

## 2 - Les fonctions

On précise le type des variables d'entrée et de la valeur de sortie

```ts
function isMajor(age: number): boolean {
  return (age >= 18) ? true : false;
}

function getUrl(webSite: string, protocol = 'https'): string {
  return `${protocol}://${webSite}`;
}
// return  https://www.site.com
let secureUrl = getUrl('site.com')
// return  http://www.site.com
let insecureUrl = getUrl('site.com', 'http');
```

On utilise les fonctions fléchées

```ts
let square1 =
  (n: number) => n * n;

let square2 =
  (n: number) => {
    return n * n;
  }
```

```ts
setTimeout(
  () => {
      // do something
  },
  1000
)
```

## 3 - Les interfaces

Elles permettent de créer d'autres types

```ts
interface User {
  pseudo: 'string';
  age?: number; // Cette propriété est optionnelle
}
let user1: User = {
  pseudo: 'Jean',
  age: 25
}
let user2: User = {
  name: 'Sylvie'
}
```

On peut créer une interface en « extends » une autre. La nouvelle possède toutes les propriétés et méthodes de l'ancienne plus les siennes.

```ts
interface Employee extends User {
  company: string;
}

let employee1: Employee = {
  pseudo: 'Virginie',
  age: 30,
  company: 'Google'
}
```

## 4 - Les classes

Elles permettent de construire des objets ayant certaines propriétés et méthodes

```ts
class Animal {
  name: string;
  nbLegs: number;

  constructor(n: string, nbL: number) {
    this.name = n;
    this.nbLegs = nbL;
  }
}

let dog = new Animal('chien', 4);
```

On peut également « extends » une classe pour en créer une nouvelle. Le « constructor » de la classe mère est appelée avec le mot clé « super ».

```ts
class Bird extends Animal {
  canFly: boolean;

  constructor(name: string, canFly: boolean) {
    super(name, 2);
    this.canFly = canFly;
  }
}

let ostrich = new Bird('autruche', false);
```

# Le data-binding

Le **data-binding** (liaison de données) permet de **synchroniser** la vue (html) et les données de l'application (ts)

## 1 - Interpolation

L'interpolation permet de transmettre les valeurs du contrôleur (ts) à la vue (html). Elle évalue ce qui se trouve à l'intérieur des **{{ … }}**, le transforme en string et l'affiche dans la vue html

### html

```html
<!-- Affiche 3 -->
{{ 2 + 1 }}
<!-- Affiche "Bonjour monde" -->
Bonjour {{ str }}
<!-- Affiche "Bonjour monde" -->
{{ 'Bonjour ' + str }}
<!-- Affiche "Age : 10" -->
Age : {{ age }}
<!-- Affiche "rue Emile Zola à Troyes -->
{{ address.street }} à {{ address.city }}
```

### ts

```ts
str: string = 'monde';
age: number = 10;
address: any = {
  street: 'rue Emile Zola',
  city: 'Troyes'
}
```

## 2 - Property binding

Tout comme l'interpellation, le **property binding** permet de transmettre des données du contrôleur à la vue. Mais il s'applique sur des **propriétés** des éléments et n'est pas transformé en string.

### a - Sur un élément HTML

```html
<!-- Redirige vers la page de google -->
<a [href]="link">Aller sur google</a>
<!-- Affiche l'image à l'adresse https://site.com/image.jpg -->
<img [src]="imageUrl" />
<!-- L'élément ne sera pas affiché (display: none) -->
<div [hidden]="1 + 1 === 2">On ne me voit pas</div>
```

```ts
link: string = 'https://google.com';
imageUrl: string = 'https://site.com/image.jpg';
```

### b - Sur un attribute directive

```html
<!-- Applique la classe CSS active car isSelected est true -->
<div [ngClass]="isSelected ? 'active' : 'inactive'"></div>
<!-- Ajoute du style à l'élément -->
<div [ngStyle]="{'color': '#ff00ff', 'font-weight': 'bold'}"></div>
```

```ts
isSelected: boolean = true;
```

### c - Sur une propriété d'un composant

```html
<!-- Transmet theUser au child-component -->
<child-component [user]="theUser"></child-component>
```

```ts
theUser: any = {
  firstName: 'Emile',
  lastName: 'Zola'
}
```

## 3 - Event binding

Les utilisateurs sont également susceptible d'interagir avec l'application en effectuant des actions (clic sur un bouton, déplacement de la souris, remplissage d'un champ texte,…). Angular permet de capter les interactions de son choix grâce à l'**event binding**.

### a - Sur un élément HTML

```
<!-- La méthode doSomething est appelée à chaque fois
que l'on appuie sur le bouton -->
<button (click)="doSomething()">Cliquer</button>

<!-- La méthode keyUpEvent est appelée à chaque fois que l'on
relache une touche et passe grâce au mot clé $event un
évenement comportant les informations sur cet'évenement -->
<input type="text" (keyup)="keyUpEvent($event)" />

<!-- Dans $event on récupère la valeur de l'input et on attribue
cet valeur à notre variable val -->
<input type="text" (keyup)="val = $event.target.value" />

<!-- Ici deleteValue sera appelée lorsque la touche esc sera enfoncée -->
<input type="text" (keydown.esc)="deleteValue()" />
```

```
val = '';

doSomething() {
  // do something...
}

keyUpEvent(event: any) {
  this.val = event.target.value;
}

keyUpValue(value: string) {
  this.val = value;
}

deleteValue() {
  this.val = '';
}
```

### b - Sur un attribute directive

```
<div (myClick)="customClick($event)" clickable>Clic</div>
```

```
customClick(isClicked: boolean) {
  // ...
}
```

### c - Sur une propriété d'un composant

```
1  <!-- L'évenement update est envoyé depuis le child-component avec la
2  valeur $event et appelle la méthode updateValue dans ce composant -->
3  <child-component (update)="updateValue($event)"></child-component>
```

```
updateValue(event: any) {
  // Do something with event
}
```

## 4 - Two-way data binding

Parfois, on souhaite **à la fois** transmettre une donnée du contrôleur à la vue (**data binding**) et modifier cette donnée après une action (**event binding**). Dans ce cas on utilise le **two-way data binding.** C'est le cas par exemple d'un input avec une valeur initiale qui met à jour cette valeur à chaque modification du champ texte. La notation est **[( … )]**

### a - Sur un élément un élément de formulaire (input, textarea, select)

On utilise la directive **ngModel**

```
<input [(ngModel)]="firstName">
<!-- est équivalent à  -->
<input [ngModel]="firstName" (ngModelChange)="firstName = $event">
<!-- est équivalent à  -->
<input [value]="firstName" (input)="firstName = $event.target.value">
```

### b - Sur un composant

```
{{ firstName }}

<app-child-comp [(name)]="firstName"></app-child-comp>
<!-- est équivalent à  -->
<app-child-comp [name]="firstName" (nameChange)="firstName = $event"></app-child-comp>
```

# Les modules

Toute application Angular est constituée d'au moins un module : l'**AppModule** qui est responsable du lancement de l'application.

Un **module** permet d'**ajouter des fonctionnalités à l'application**.
Il existe des modules développés par Angular et que l'on peut utiliser.

- **FormsModule** : pour la gestion et la validation des formulaires
- **RouterModule** : pour la navigation à travers les pages de l'application
- **HttpClientModule** : pour effectuer des requêtes HTTP

```ts
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 { }
```

@NgModule est un décorateur qui permet à Angular d'identifier qu'il s'agit d'identifier qu'il s'agit d'un module

**declarations** : les composants, directives, pipes utilisées dans ce module
**imports** : les modules dont les composants ou directives seront utilisés dans ce module
**exports** : des éléments de déclarations qui seront utilisables dans les modules qui importeront celui-ci
**providers** : créateur de services qui seront utilisés dans ce module

On ajoute des modules externes à notre application via **npm**

Un module comporte des **composants**, **directives**, **pipes** et **services**

# Les composants

Un composant est formé de :

- un ficher html qui comporte sa structure
- un fichier scss qui apporte le style propre à ce composant sans interférer avec les autres composants
- un fichier typescript qui comporte toute la logique du composant : valeurs des variables, création d'une requête http,…

Angular identifie un composant grâce au décorateur **@Component**

**element.component.scss**

```scss
:host {
  display: block;
  background-color: #ddd;
}

.top {
  font-size: 20px;
}
```

**element.component.ts**

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

@Component({
  selector: 'app-element',
  templateUrl: './element.component.html',
  styleUrls: ['./element.component.scss']
})
export class ElementComponent implements OnInit {

  title: string;

  constructor() { }

  ngOnInit() {
    this.title = `Le titre de l'élément`;
  }

}
```

**element.component.html**

```html
<div class="top">
  {{ title }}
</div>
<div>
  Contenu de l'élément
</div>
```

Angular crée un tag HTML que l'on utilise pour afficher le composant

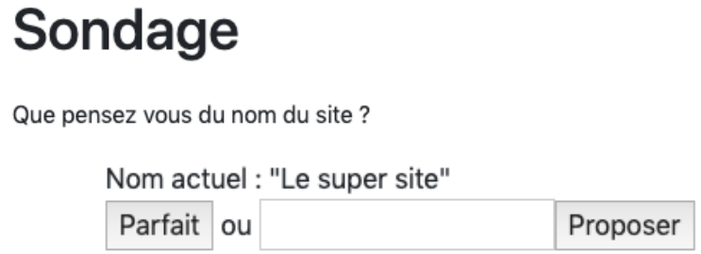
```html
<div>
  La vue du composant vient en dessous :
</div>

<app-element></app-element>
```

## Composant parent

### discussion.component.html

```html
<h1>Sondage</h1>
<p>
  Que pensez vous du nom du site ?
  <app-choose-value
    [value]="websiteName"
    (accept)="acceptName($event)"
    (propose)="proposeName($event)">
  </app-choose-value>
</p>
```




### discussion.component.ts

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

@Component({
  selector: 'app-discussion',
  templateUrl: './discussion.component.html',
  styleUrls: ['./discussion.component.scss']
})
export class DiscussionComponent implements OnInit {

  websiteName: string;

  constructor() { }

  ngOnInit() {
    this.websiteName = 'Le super site';
  }

  acceptName(event: any) {
    // Do something
  }

  proposeName(name: string) {
    // Do something with name
  }

}
```

## Composant enfant

### choose-value.component.html

```html
<div>
  Nom actuel : "{{ value }}"
</div>
<div>
  <button (click)="accept.emit()">Parfait</button> ou
  <input #valueInput type="text">
  <button (click)="clickOnPropose(valueInput.value)">Proposer</button>
</div>
```

### choose-value.component.ts

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

@Component({
  selector: 'app-choose-value',
  templateUrl: './choose-value.component.html',
  styleUrls: ['./choose-value.component.scss']
})
export class ChooseValueComponent implements OnInit {

  @Input() value: string;
  @Output() accept: EventEmitter<void> = new EventEmitter();
  @Output() propose: EventEmitter<string> = new EventEmitter();

  constructor() { }

  ngOnInit() {
  }

  clickOnPropose(val: string) {
    this.propose.emit(val);
  }

}
```

**@Input** pour obtenir une propriété du composant parent
**@Output** pour envoyer un événement au composant parent

# Les directives

Elles permettent de **modifier le comportement** des éléments HTML **ou d'en ajouter**

## 1 - Les directives structurelles

Elles modifient la structure de la page : elles ajoutent, modifient ou suppriment des éléments HTML. Elles commencent par un astérisque *

**component.ts (extrait)**

```ts
categories: string[] = ['Musique', 'Cinéma', 'Chanson'];
isAuthenticated: boolean = true;
```

**component.html**

```html
<div *ngIf="2 + 2 === 5">
  Ce div ne s'affichera pas car la condition est évaluée à false
</div>

<ul>
  <li *ngFor="let item of ['Accueil', 'Profil', 'Contact']">
    {{ item }}
  </li>
</ul>

<p *ngIf="isAuthenticated">
  Bienvenue
</p>

<table class="table table-bordered">
  <tr *ngFor="let category of categories; let i = index;">
    <td>{{ i + 1 }}
    <td>
      {{ category }}
    </td>
  </tr>
</table>
```

**Affichage**

- Accueil
- Profil
- Contact

Bienvenue

| | |
|---|---|
| 1 | Musique |
| 2 | Cinéma |
| 3 | Chanson |

Directives structurels fournies par Angular :

***ngIf**
***ngFor**
***ngSwitch**

## 2 - Les directives d'attribut

Elles modifient l'apparence ou le comportement d'un élément HTML

**component.html**

```
<!-- ngClass accepte une liste de classes CSS séparées par des espaces -->
<div [ngClass]="'first second'"></div>
<!-- ngClass accepte un tableau de classes CSS -->
<div [ngClass]="['first', 'second']"></div>
<!--  ngClass accepte un objet dans les clés sont les classes CSS. Si la condition
est évaluée à true, la classe est ajoutée à l'élément sinon elle est retirée  -->
<div [ngClass]="{'first': true, 'second': 2 + 3 === 5, 'third': someVariable}"></div>

<!-- ngClass accepte un objet dans lequel chaque clé est un propriété CSS -->
<div [ngStyle]="{'font-size': '20px', 'max-width.px': 100, 'min-height.%': 50}"></div>

<!-- ngModel permet de faire du data-binding (liaison de données)
entre les éléments HTML d'un formulaire et l'application' -->
<input type="text" ngModel #firstName="ngModel">
<input type="text" [(ngModel)]="lastName">
<p>
  Votre nom est {{ firstName.value }} {{ lastName }}
<p>

<!-- routerLink permet de naviguer dans l'application -->
<a routerLink="/contact">Aller à la page contact</a>
```

# 3 - Créer sa propre directive

## string-max.directive.ts

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

@Directive({
  selector: '[appStringMax]'
})
export class StringMaxDirective implements AfterViewInit {

  @Input() appStringMax: number;
  @Input() withPoints: boolean = true;

  constructor(private el: ElementRef) {}

  ngAfterViewInit() {
    // Ajouter une valeur par défaut
    this.appStringMax = this.appStringMax || 5;
    // Garder les premiers caractères
    let str = this.el.nativeElement.textContent.substring(0, this.appStringMax);
    // Ajouter "..." si nécessaire
    if (this.withPoints) {
      str += '...';
    }
    this.el.nativeElement.textContent = str;
  }

}
```

Exemple : Créer une directive qui permet de couper un texte en fonction du nombre de caractères

## component.html

```html
<div appStringMax>Voici une chaine de caractères un peu longue</div>
<div [appStringMax]="strMax1">Voici une chaine de caractères un peu longue</div>
<div [appStringMax]="strMax2" [withPoints]="false">Voici une chaine de caractères un peu longue</div>
```

## Affichage

Voici...
Voi...
Voici une ch

## component.ts (extrait)

```ts
    strMax1 = 3;
    strMax2 = 12;
```

# Les services

Les **services** permettent de **structurer l'application** de façon à séparer la vue et les données.
Ils sont également utiles pour des fonctions spécifiques (réduction d'une image, calcul d'un prix TTC,…) et pour la communication entre les composants.

```ts
import { Component, OnInit } from '@angular/core';
import { CategoryService } from './services/category.service';
import { NewsletterService } from './services/newsletter.service';

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

export class AppComponent {

  categories: string[];

  constructor(
    private categoryService: CategoryService,
    private newsletterService: NewsletterService
  ) {}

  ngOnInit() {
    this.categories = this.categoryService.getCategories();
  }

  inscriptionToNewsletter(email: string) {
    this.newsletterService.inscription(email);
  }

}
```

```ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CategoryService {

  categories: string[] = [
    'Musique',
    'Cinéma',
    'Chanson'
  ];

  constructor() { }

  getCategories(): string[] {
    return this.categories;
  }
}
```

```ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class NewsletterService {

  constructor(private http: HttpClient) { }

  inscription(email: string) {
    return this.http.post('http://site.com/api/newletter', {
      email: email
    }).subscribe();
  }
}
```

# Le routage

Le router d'Angular permet de simuler l'existence de pages différentes bien qu'en réalité il n'y en ait qu'une seule : index.html (SPA). L'objectif est d'obtenir les comportements classiques d'un site internet, à savoir :

- Être redirigé directement vers la bonne vue à l'entrée de l'url http://site.com/contact dans la barre de navigation ou lors d'un clic sur un lien
- Pouvoir cliquer sur les boutons suivant et précédent du navigateur.

Pour cela il va falloir importer le **RouterModule** d'Angular et configurer toutes les **routes** de l'application.

Grâce à javascript, on a la possibilité de modifier l'historique de navigation : « Browser's History Journal ». Ainsi on va pouvoir ajouter des pages à ce journal pour permettre le bon comportement des boutons précédent et suivant du navigateur. C'est également le role du **RouterModule.**

Pour éviter que le navigateur n'effectue réellement une nouvelle requête, on ne va pas utiliser la propriété **href**, ce qui aurait pour effet de recharger complètement la page, mais une directive **routerLink**

Il faut faire attention à bien configurer le serveur sur lequel l'application finale sera hébergée. Il faut rediriger toutes les urls vers index.html qui s'occupera grâce au router d'afficher la bonne vue

**app.module.ts**

```ts
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'home', component: HomeComponent},
  {path: 'contact', component: ContactComponent, data: {
    title: 'Contact page'}
  },
  {path: 'users', component:UsersComponent, children: [
    {path: '', redirectTo: 'list', pathMatch: 'full'},
    {path: 'list', component: UsersListComponent},
    {path: 'detail/:id', component: UserDetailComponent},
    {path: '**', redirectTo: 'list'}
  ]},
  {path: '**', component: Error404Component}
];

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes)
  ],
  declarations: [
    AppComponent,
```

- Les routes sont composés d'un **path** et d'un **component** qui sera utilisé lorsque l'url correspondra au chemin indiqué.
- Le **router** cherche l'url de la première route correspondante en partant du premier élément. Dés que le **path** d'une route correspond, la recherche s'arrête et la vue s'affiche. C'est la stratégie du **first match win**
- Le path vide '' correspond au chemin de base, par exemple http://site.com
- Le path ** est une carte blanche qui permet d'attraper tous les chemins. Compte tenu du **first match win** il faut le placer en dernier
- Il est également possible de créer des **sous routes**. Ici on a par exemple http://site.com/users/list. La propriété **redirectTo** permet de faire une redirection. Ici http://site.com/users redirige vers http://site.com/users/list
- On peut également ajouter des **data** à une route.
- La navigation vers une url comportant un paramètre est également possible en utilisant **detail/:id**

**app.component.html**

```html
Le contenu des pages au dessous
<router-outlet></router-outlet>
```

Pour afficher la vue il suffit de placer une balise **router-outlet**.
Le contenu de la vue correspondante se placera au niveau de cette balise.

**users.component.html**

```html
<h1>Les utilisateurs</h1>
<ul>
  <li><a routerLink="/">Home</a></li>
  <li><a routerLink="/contact">Contact</a></li>
  <li><a routerLink="list" routerLinkActive="active">liste</a></li>
  <li><a routerLink="detail/1" routerLinkActive="active">Utilisateur 1</a></li>
  <li><a [routerLink]="['detail', userId]">Utilisateur {{ userId }}</a></li>
</ul>
<router-outlet></router-outlet>
```

- La directive **router-link** permet d'indiquer le chemin de redirection après le clic. Elle joue le role de href.
- Il est possible d'indiquer un chemin absolu en ajoutant un '/', sinon le chemin est relatif. **/contact** correspond à l'url http://site.com/contact alors que **detail/1** correspond à http://site.com/users/detail/1
- la directive **routerLinkActive** permet d'ajouter une classe (ici 'active') lorsque l'on se trouve sur la route. Ce qui peut être utilise pour modifier l'affichage.

**user-detail.component.ts**

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

@Component({
  selector: 'app-user-detail',
  templateUrl: './user-detail.component.html',
  styleUrls: ['./user-detail.component.scss']
})
export class UserDetailComponent implements OnInit {

  userId: string;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.route.paramMap.subscribe(params => {
      this.userId = params.get('id');
    })
  }

}
```

**user-detail.component.html**

```html
<h2>id user : {{ userId }}</h2>
```

**page site.com/users/detail/32**

Le contenu des pages au dessous

# Les utilisateurs

- Home
- Contact
- liste
- Utilisateur 1
- Utilisateur 32

## id user : 32

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

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent {

  constructor(private router: Router) { }

  goToContact() {
    this.router.navigate(['/contact']);
  }

  goToDetail() {
    let userId = '42';
    this.router.navigate(['/users/detail', userId]);
  }

}
```

Il est également possible de changer de route dans le contrôleur.

Certaines fois, il peut être intéressant de ne pas charger toutes les « pages » de l'application, notamment pour une grosse application, dans le but de réduire le temps de chargement de la première page. Dans ce cas on peut **découper l'application en plusieurs modules** correspondant aux pages et dont les fichiers javascript seront téléchargés « ultérieurement ».
Dans ce cas, il faut paramétrer les **routes** à l'intérieur du nouveau module.

## app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';


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

## home.module.ts

```ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';

const routes: Routes = [
  {path: '', component: HomeComponent}
];

@NgModule({
  declarations: [HomeComponent],
  imports: [
    CommonModule,
    RouterModule.forChild(routes)
  ]
})
export class HomeModule { }
```

## app-routing.module

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

const routes: Routes = [
  {path: '', redirectTo: '/home', pathMatch: 'full'},
  {path: 'home', loadChildren: './home/home.module#HomeModule'}
];

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

# Les formulaires

Un formulaire est composé de balises « input », « textarea », « select » qui permettent à l'utilisateur d'interagir avec l'application pour lui transmettre des données (qui pourront ensuite être envoyées au serveur, par exemple).

## input, textarea, select

C'est la directive **ngModel** et le two-way data binding qui va permettre à Angular de repérer les changement dans les balises du formulaire et de leur donner une valeur par défaut.

```
<input type="text" name="firstName" [(ngModel)]="firstName" #fName="ngModel" required>
```

Il est également possible d'être informé de l'état de cet input. Angular va ajouter des propriétés au **formControl** « fName » :

| | OUI | NON |
|---|---|---|
| Le control a été visité | touched: true<br>untouched: false | touched: false<br>untouched: false |
| La valeur a changé | dirty: true<br>pristine: false | dirty: false<br>pristine: true |
| Le control est valide | valid: true<br>invalid: false | valid: false<br>invalid: true |

Dans la même idée, des classes **ng-touched**, **ng-untouched**, **ng-dirty**, **ng-pristine**, **ng-valid**, **ng-invalid** à l'élément input ou select.

## La validation

Il est possible d'ajouter des directives de validation pour indiquer les comportements requis de l'input, select, par exemple : **required**, **email**, **minlength='3'**, …
Le formControl « fName » dans notre exemple a également une propriété **errors** qui vaut **null** lorsque tout est correct et, par exemple, {required: true} si le champ est vide.
C'est plus utile que les propriétés **valid** ou **invalid** pour spécifier les messages d'erreurs

## La balise <form>

Un formulaire est toujours délimité par des balises **<form></form>**. Mais comme on est en **SPA**, **pas de balise action**. Si des données sont à envoyer au serveur, il faudra faire une requête à part.
La balise form crée un **formGroup** qui regroupe tous les **formControl** à l'intérieur. Ainsi le formGroup a également des propriétés **valid**, **invalid** qui décrivent la validité globale de tout le formulaire.

```
<form #testForm="ngForm" (ngSubmit)="submitForm(testForm)">
  <input type="text" name="firstName" [(ngModel)]="firstName" #fName="ngModel" required>
  <div *ngIf="fName.errors?.required" class="error">
    Le prénom est obligatoire
  </div>
  <select name="sel" [(ngModel)]="...">...</select>
  <input type="submit" [disabled]="testForm.invalid">
</form>
```

Le **template input variable** « testForm » est associé par la directive **ngForm** à tout le formulaire. Il est donc possible de désactiver le bouton de soumission lorsque le formulaire n'est pas valide.

L'événement **ngSubmit** est déclenché après la soumission du formulaire (appui sur l'input de type submit). C'est le moment pour effectuer des opérations sur ce formulaire (envoyer les données au serveur,…).

# Liste de commandes Angular

Créer une application :
ng new myApp

Créer un composant :
**ng generate component pages/home**

Créer une classe :
**ng generate class myClass —skipTests**

Créer un service :
**ng generate service myService**

Lancer l'application et l'ouvrir dans le navigateur :
**ng serve -o**

Construire l'application pour la production :
**ng build —prod**

# Le cycle de vie d'une application angular

| | |
|---|---|
| **constructor** | Appelé une seule fois lors de la création |
| **ngOnChanges** | Appelé lorsque des propriétés @Input sont définies ou modifiées |
| **ngOnInit** | Appelé une seule fois lorsque les propriétés sont définies. Idéal pour initialiser les valeurs, appeler les services,… |
| **ngDoCheck** | Appelé après chaque détection de changement (clic, requête http,…) |
| **ngAfterContentInit** | Appelé une fois après que le contenu à l'intérieur des balises <ng-content> soit initialisé |
| **ngAfterContentChecked** | Appelé après que le contenu à l'intérieur des balises <ng-content> soit vérifié par le mécanisme de détection de changement |
| **ngAfterViewInit** | Appelé une fois lorsque la vue et les vues enfants sont initialisées. Idéal pour modifier un élément de la vue (par exemple sa taille) |
| **ngAfterViewChecked** | Appelé après qu'une une vue ou une vue enfant soit vérifiée par le mécanisme de détection de changement |
| **ngOnDestroy** | Appelé avant la destruction du composant. Idéal pour se désabonner (unsubscribe) des observables, effacer les timers,… |

## Quelques commandes cordova :

Créer une nouvelle application cordova :
**cordova create my-app**

Ajouter une plateforme :
**cordova platform add android**

Ajouter un plugin :
**cordova plugin add cordova-plugin-camera@^2.0.0**

Supprimer un plugin :
**cordova plugin rm StatusBar**

Voir la liste des plugins :
**cordova plugin ls**

Voir l'état des exigence pour le développement sur une plateforme :
**cordova requirements**

Lancer l'application dans un émulateur :
**cordova emulate android**

Lancer l'application sur le mobile connecté :
**cordova run android**

Construire l'application pour l'OS du mobile (pour android crée un .apk)
**cordova build android**

## Quelques commandes ionic :

Créer une nouvelle application ionic :
**ionic start my-app**

Ajouter une page :
**ionic generate page contact**

Ajouter un composant :
**ionic generate component contact/form**

Ajouter un service :
**ionic generate service api/user**

Ajouter une plateforme :
**ionic cordova platform add android**

Ajouter un plugin :
**ionic cordova plugin add cordova-plugin-camera**
**npm install @ionic-native-camera**

Supprimer un plugin :
**ionic cordova plugin rm cordova-plugin-camera**

Lancer l'application dans un émulateur :
**ionic cordova emulate android**

Lancer l'application sur le mobile connecté :
**ionic cordova run android**

Construire l'application pour l'OS du mobile (pour android crée un .apk)
**cordova build android**

# Le cycle de vie d'une application Ionic