

Formation Angular

Ihab ABADI / UTOPIOS

Programme Partie 1

- **Rappel Es6+ Javascript**
- **TypeScript et typage en Javascript**
- **Philosophie du Framework Angular**
- **Notion de templating**
- **Angular CLI et environnement de dev**
- **Structure d'une application Angular**
- **Notion de composant**
- **Passer des données à un composant**
- **Utilisation des événements dans un composant**
- **TwoWay Binding**
- **Utilisation des projections**
- **Notion de directives**
- **Directive d'attribut**
- **Directive de structure**

Rappel ES6 Javascript

- **Introduction des classes :**
 - ES6 a introduit la possibilité d'instanciation d'objet à partir d'une classe
 - La possibilité d'utilisation d'héritage comme tout autre langage de POO
- **Arrow Functions:**
 - La possibilité d'écrire des fonctions sans état avec une écriture fléchée
- **Template Strings**
 - Mécanisme d'interpolation des chaînes de caractères
- **Variable muable et mutable et block scoped**
- **Opérateur ...spread et ...rest**
 - La possibilité de récupérer les éléments d'un tableau, objet ou passer un nombre illimité de paramètres
- **Déstructuration des objets**
 - La possibilité d'extraire des variables à partir d'un objet ou tableau plus rapidement
- **Utilisation des modules**
 - Utilisation des modules donne la possibilité d'exporter d'importer des fonctionnalités, objet, variable

TypeScript

- **TypeScript est un transpiler de Javascript**
- **TypeScript offre la possibilité d'utiliser un langage fortement typé**
- **TypeScript peut utiliser plusieurs type (boolean, number, string, [], {}, undefined, enum, any...)**
- **TypeScript donne la possibilité d'utiliser des classes**
- **TypeScript donne la possibilité d'utiliser des interfaces**
- **TypeScript donne la possibilité d'utiliser des Decorators (Annotation)**
- **Démo**

Philosophie Angular

- **Angular est un (Vrai) Framework développé par Google**
- **Pourquoi utiliser Angular ?**
- **TypeScript**
- **Facilité d'apprentissage**
- **Une multitude de fonctionnalité disponible directement dans le framework**
 - **Routage**
 - **Formulaire**
 - **Observable**
 -

Angular Templating

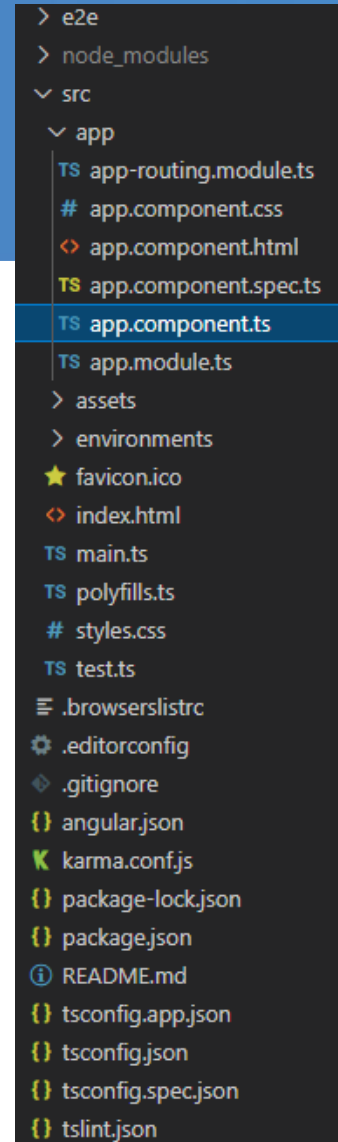
- **La gestion du rendu dans angular se fait directement dans le DOM réel**
- **Angular utilise le moteur rendu Ivy**
- **La compilation et la génération du code en AOT**
- **L'amélioration des debug**
- **Diminution du poids des Js**

Environnement de dev et Angular CLI

- **Pour le développement en Angular, on utilisera :**
 - **Un ide comme visual studio**
 - **NodeJS**
- **Angular fournit une commande Line interface pour la simplification de développement**
- **Installation du CLI se fait à l'aide d'un package npm**
- **Quelque exemple de commande**

Structure d'un projet Angular

- Dossier node_module
- Fichier de configuration (node, typescript)
- Fichier de configuration de test
- Fichier Modules
- Fichier composants

A screenshot of a file explorer window showing the structure of an Angular project. The 'src' directory is expanded, showing sub-directories 'app' and 'assets', and files like 'environments', 'favicon.ico', 'index.html', 'main.ts', 'polyfills.ts', 'styles.css', and 'test.ts'. The 'app' directory is further expanded, showing 'app-routing.module.ts', 'app.component.css', 'app.component.html', 'app.component.spec.ts', and 'app.component.ts' (which is highlighted). Other files in the root of the project include '.browserslistrc', '.editorconfig', '.gitignore', 'angular.json', 'karma.conf.js', 'package-lock.json', 'package.json', 'README.md', 'tsconfig.app.json', 'tsconfig.json', 'tsconfig.spec.json', and 'tslint.json'.

```
> e2e
> node_modules
v src
  v app
    TS app-routing.module.ts
    # app.component.css
    <> app.component.html
    TS app.component.spec.ts
    TS app.component.ts
    TS app.module.ts
  > assets
  > environments
  ★ favicon.ico
  <> index.html
  TS main.ts
  TS polyfills.ts
  # styles.css
  TS test.ts
≡ .browserslistrc
⚙ .editorconfig
🔒 .gitignore
{} angular.json
📄 karma.conf.js
{} package-lock.json
{} package.json
📖 README.md
{} tsconfig.app.json
{} tsconfig.json
{} tsconfig.spec.json
{} tslint.json
```


Notion de composants

- Le concept de base de toute application angular est le composant.
- En effet, toute l'application peut être modélisée sous la forme d'un arbre de ces composants.
- Fondamentalement, un composant est tout ce qui est visible par l'utilisateur final et qui peut être réutilisé plusieurs fois dans une application.
- Création d'un composant en Angular se fait à l'aide de l'annotation `@Component`
- Un composant est défini par plusieurs méta-data (selector, template,...)
- Un Composant peut partagé ses propriétés avec le template

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

@Component({
  selector: "rio-hello",
  template: "<p>Hello, {{name}}!</p>",
})
export class HelloComponent {
  name: string;

  constructor() {
    this.name = "World";
  }
}
```

Notion de composants

Un moyen utile de conceptualiser la conception d'applications angulaires consiste à la considérer comme une arborescence de composants imbriqués, chacun ayant une portée isolée

```
<rio-todo-app>
  <rio-todo-list>
    <rio-todo-item></rio-todo-item>
    <rio-todo-item></rio-todo-item>
    <rio-todo-item></rio-todo-item>
  </rio-todo-list>
  <rio-todo-form></rio-todo-form>
</rio-todo-app>
```

Utilisation des événements dans un composant

- Un gestionnaire d'événements est spécifié à l'intérieur du modèle à l'aide de crochets pour désigner la liaison d'événements.
- Ce gestionnaire d'événements est ensuite codé dans la classe pour traiter l'événement

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

@Component({
  selector: 'rio-counter',
  template: `
    <div>
      <p>Count: {{num}}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class CounterComponent {
  num = 0;

  increment() {
    this.num++;
  }
}
```

Utilisation des évènements dans un composant

- Dans Angular, on peut créer nos propres évènements pour remonter des données à un composant parent.
- La mise en place d'un évènement se fait par une propriété bindable et annoter avec **@Output** de type **EventEmitter**

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

@Component({
  selector: 'rio-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent implements OnChange {
  num = 0;
  parentCount = 0;

  ngOnChange(val: number) {
    this.parentCount = val;
  }
}
```

```
<div>
  Parent Num: {{ num }}<br>
  Parent Count: {{ parentCount }}
  <rio-counter [count]="num" (result)="ngOnChange($event)">
  </rio-counter>
</div>
```

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

@Component({
  selector: 'rio-counter',
  templateUrl: 'app/counter.component.html'
})
export class CounterComponent {
  @Input() count = 0;
  @Output() result = new EventEmitter<number>();

  increment() {
    this.count++;
    this.result.emit(this.count);
  }
}
```

```
<div>
  <p>Count: {{ count }}</p>
  <button (click)="increment()">Increment</button>
</div>
```

Two way binding

- Two way binding combine à la fois le Input et OutPut avec la directive ngModel

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

Utilisation des projections

- La projection est un concept très important dans Angular.
- Il permet aux développeurs de créer des composants réutilisables et de rendre les applications plus évolutives et flexibles.
- Pour illustrer cela, supposons que nous ayons un ChildComponent comme

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

@Component({
  selector: 'rio-child',
  template: `
    <div style="border: 1px solid blue; padding: 1rem;">
      <h4>Child Component</h4>
      <ng-content></ng-content>
    </div>
  `
})
export class ChildComponent {
}
```

```
...
<rio-child>
  <p>My <i>projected</i> content.</p>
</rio-child>
...
```

Directives

- Une directive modifie le DOM pour changer l'apparence, le comportement ou la disposition des éléments DOM.
- Les directives sont l'un des principaux blocs de construction utilisés par Angular pour créer des applications. En fait, les composants angular sont en grande partie des directives avec des modèles.
- Il existe trois principaux types de directives dans Angular:
- Composant - directive avec un modèle.
- Directives d'attribut - directives qui modifient le comportement d'un composant ou d'un élément mais n'affectent pas le modèle
- Directives structurelles - directives qui modifient le comportement d'un composant ou d'un élément en affectant la façon dont le modèle est rendu

Directives d'attribut

- **Les directives d'attribut sont un moyen de modifier l'apparence ou le comportement d'un composant ou d'un élément DOM natif. Idéalement, une directive devrait fonctionner d'une manière indépendante des composants et non liée aux détails de mise en œuvre.**
- **Par exemple, Angular a des directives d'attribut intégrées telles que `ngClass` et `ngStyle` qui fonctionnent sur n'importe quel composant ou élément**
- **demo**

Directives de structure

- Les directives structurelles sont un moyen de gérer le rendu d'un composant ou d'un élément via l'utilisation de la balise de template.
- Cela nous permet d'exécuter du code qui décide de la sortie finale rendue. Angular a quelques directives structurelles intégrées telles que `ngIf`, `ngFor` et `ngSwitch`
- demo

Programme Partie 2

- **Injection de dépendance en Angular, Injection et utilisation de services**
- **Utilisation des observables**
- **Utilisation de module en Angular**
- **Module de routing**
- **Module ReactiveForms**

Injection de dépendance en Angular

- L'injection de dépendances (DI) était une fonctionnalité de base dans Angular.
- DI est un concept de programmation antérieur à Angular.
- Le but de DI est de simplifier la gestion des dépendances dans les composants logiciels. En réduisant la quantité d'informations dont un composant a besoin pour connaître ses dépendances, les tests unitaires peuvent être simplifiés et le code a plus de chances d'être flexible.
- Le système DI d'Angular est (principalement) contrôlé via `@NgModule`. Plus précisément, via un tableau de providers

```
@NgModule({  
  declarations[ ChatWidget, AuthWidget ]  
  providers: [ AuthService, ChatSocket ],  
})
```

Injection de dépendance en Angular

- L'interaction avec DI Angular se fait à l'aide d'annotation `@Inject` et `@Injectable`
- `@Inject ()` est un mécanisme manuel pour faire savoir à Angular qu'un paramètre doit être injecté.
- `@Injectable ()` fait savoir à Angular qu'une classe peut être utilisée avec l'injecteur de dépendances.
- `@Injectable ()` n'est pas strictement requis si la classe a d'autres décorateurs angulaires dessus ou n'a pas de dépendances. Ce qui est important, c'est que toute classe qui va être injectée avec Angular soit décorée.
- La meilleure pratique consiste à décorer les injectables avec `@Injectable ()`, car cela a plus de sens pour le lecteur

```
constructor(@Inject(ChatWidget) private chatWidget) { }
```

```
@Injectable()  
export class ChatWidget {  
  constructor(  
    public authService: AuthService,  
    public authWidget: AuthWidget,  
    public chatSocket: ChatSocket) { }  
}
```

Injection de dépendance en Angular

- **DI Angular renvoie un singleton si l'injection se fait au niveau du module principale**
- **Angular offre la possibilité de définir le scope d'injection au niveau d'un composant, un sous module, ..**

```
@Component({  
  selector: "child-own-injector",  
  template: `<span>{{ value }}</span>`,  
  providers: [Unique],  
})
```

Utilisation des observables

- Une nouvelle fonctionnalité intéressante utilisée avec Angular est l'Observable.
- Il ne s'agit pas d'une fonctionnalité spécifique à Angular, mais plutôt d'une norme proposée pour la gestion des données asynchrones.
- Les observables ouvrent un canal de communication continu dans lequel plusieurs valeurs de données peuvent être émises au fil du temps.
- À partir de là, nous obtenons un modèle de traitement des données en utilisant des opérations de type tableau pour analyser, modifier et maintenir les données.
- Angular utilise largement les observables

Utilisation des observables

- Création d'un observable à l'aide d'une librairie RxJS

```
this.data = new Observable(observer => {
  setTimeout(() => {
    observer.next(42);
  }, 1000);

  setTimeout(() => {
    observer.next(43);
  }, 2000);

  setTimeout(() => {
    observer.complete();
  }, 3000);
});

let subscription = this.data.subscribe(
  value => this.values.push(value),
  error => this.anyErrors = true,
  () => this.finished = true
);
```

Utilisation des observables VS Promise

- Les promesses et les observables nous fournissent des abstractions qui nous aident à gérer la partie asynchrone de nos applications.
- Cependant, il existe des différences importantes entre les deux:
- Les observables peuvent définir à la fois les aspects de configuration et de démontage du comportement asynchrone.
- Les observables sont annulables.
- De plus, Observables peut être retenté à l'aide de l'un des opérateurs de relance fournis par l'API, tels que `retry` et `retryWhen`.
- les promesses exigent que l'appelant ait accès à la fonction d'origine qui a renvoyé la promesse afin d'avoir une capacité de nouvelle tentative

Utilisation de module

- Dans Angular, un module est un mécanisme pour regrouper des composants, des directives, des pipes et des services qui sont liés, de manière à pouvoir être combinés avec d'autres modules pour créer une application.
- Une application angulaire peut être considérée comme un puzzle où chaque pièce (ou chaque module) est nécessaire pour pouvoir voir l'image complète.
- Pour créer un module, on utilise l'annotation @NgModule

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

@NgModule({
  imports: [ ... ],
  declarations: [ ... ],
  bootstrap: [ ... ]
})
export class AppModule { }
```


TP Complet Module

- Créer un module ainsi que tous les composants nécessaires pour réaliser l'application suivante

[Demo](#) [Home](#) [About](#) [Contact](#)

[Books](#) [Toys](#) [Music](#)

ID	Title	Price	Shopping Cart	
1	Book 1	400	- <input type="text" value="0"/> +	Edit Remove
2	Book 2	250	- <input type="text" value="0"/> +	Edit Remove
3	Book 3	650	- <input type="text" value="0"/> +	Edit Remove



Angular et le routing

- **Le routage (Ou le routing) nous permet d'exprimer certains aspects de l'état de l'application dans l'URL.**
- **Contrairement aux solutions frontales côté serveur, cela est facultatif.**
- **Nous pouvons créer l'application complète sans jamais changer l'URL.**
- **L'ajout d'un routage permet à l'utilisateur d'accéder directement à certains aspects de l'application.**
- **Ceci est très pratique car cela permet de garder votre application en lien et en signet et permet aux utilisateurs de partager des liens avec d'autres.**
- **Le routage vous permet de:**
 - **Maintenir l'état de l'application**
 - **Mettre en œuvre des applications modulaires**
 - **Implémenter l'application en fonction des rôles (certains rôles ont accès à certaines URL)**

Angular et le routing (Configuration)

- La configuration du routing commence par définir les Routes
- Le type Routes est un tableau de routes qui définit le routage de l'application.
- C'est dans Routes que nous pouvons configurer les chemins attendus, les composants que nous voulons utiliser et ce que nous voulons que notre application les comprenne.
- Chaque itinéraire peut avoir des attributs différents; certains des attributs communs sont:
 - chemin - URL à afficher dans le navigateur lorsque l'application se trouve sur la route spécifique
 - composant - composant à rendre lorsque l'application est sur l'itinéraire spécifique
 - redirectTo - redirige la route si nécessaire; chaque route peut avoir un attribut de composant ou de redirection défini dans l'itinéraire (abordé plus loin dans ce chapitre)
 - children - tableau d'objets de définitions d'itinéraire représentant les itinéraires enfants de cet itinéraire (abordés plus loin dans la formation)

```
const routes: Routes = [  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

Angular et le routing (Configuration)

- Les routes sont à importer dans le module de base en utilisant le module RouterModule

```
...  
import { RouterModule, Routes } from '@angular/router';  
  
const routes: Routes = [  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];  
  
export const routing = RouterModule.forRoot(routes);
```

```
...  
import { routing } from './app.routes';  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    routing  
  ],  
  declarations: [  
    AppComponent,  
    ComponentOne,  
    ComponentTwo  
  ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule {  
}
```

Angular et le routing (RouterOutlet)

- RouterOutlet permet de réserver un espace au composant
- Angular ajoute dynamiquement le composant de la route en cours d'activation dans l'élément <router-outlet> </router-outlet>

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

@Component({
  selector: 'app',
  template: `
    <nav>
      <a routerLink="/component-one">Component One</a>
      <a routerLink="/component-two">Component Two</a>
    </nav>

    <router-outlet></router-outlet>
    <!-- Route components are added by router here -->
  `
})
export class AppComponent {}
```

Angular et le routing (Utilisation du redirectTo)

- Lorsque notre application démarre, elle redirige vers la route par défaut.
- Nous pouvons configurer le routeur pour rediriger vers une route nommée

```
export const routes: Routes = [  
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

Angular et le routing (Navigation entre route)

- La navigation peut se faire à l'aide de la directive RouterLink

```
<a routerLink="/component-one">Component One</a>
```

- La navigation peut se faire à l'aide un objet Router

```
this.router.navigate(['/component-one']);
```


Angular et le routing (Paramètres de route)

- La déclaration de paramètres se fait dans la déclaration de route
- La navigation vers une route avec un paramètre peut se faire avec la directive routerLink
- La navigation vers une route avec un paramètre peut également se faire avec un objet router
- Récupération des paramètres se fait à l'aide d'un objet de type ActivatedRoute et de l'observable queryParams

```
{ path: 'product-details/:id', component: ProductDetails }
```

```
[routerLink]="['/product-details', product.id]"
```

```
this.router.navigate(['/product-details', id]);
```

```
this.sub = this.route.params.subscribe(params => {  
  this.id = +params['id']; // (+) converts string 'id' to a number  
  
  // In a real app: dispatch action to load the details here.  
});
```

Angular et le routing (Children)

- Lorsque certaines routes ne peuvent être accessibles et affichées que dans d'autres routes, il peut être approprié de les créer comme routes enfants.
- Par exemple: la page des détails du produit peut avoir une section de navigation par onglets qui affiche la présentation du produit par défaut. Lorsque l'utilisateur clique sur l'onglet "Spécifications techniques", la section affiche les spécifications à la place
- product-details/3/overview

```
export const routes: Routes = [  
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },  
  { path: 'product-list', component: ProductList },  
  { path: 'product-details/:id', component: ProductDetails,  
    children: [  
      { path: '', redirectTo: 'overview', pathMatch: 'full' },  
      { path: 'overview', component: Overview },  
      { path: 'specs', component: Specs }  
    ]  
  }  
];
```

Angular et le routing (Lazy loading)

- En configurant l'intégralité du Routing de l'application dans le module principal, on serait amené à importer tous les modules de l'application avant son démarrage.
- A titre d'exemple, plus l'application sera riche, plus la page d'accueil sera lente à charger par effet de bord.
- Pour éviter ces problèmes de "scalability", Angular permet de charger les modules à la demande (i.e. : "Lazy Loading") afin de ne pas gêner le chargement initial de l'application.
- Seul le module AppRoutingModuleModule importe le module RouterModule avec la méthode statique forRoot afin de définir le "Routing" racine et la configuration du router via le second paramètre.
- Les "Child Routing Modules" importent le RouterModule avec la méthode forChild.

```
export const appRouteList: Routes = [  
  {  
    path: 'landing',  
    component: LandingComponent  
  },  
  {  
    path: 'book',  
    loadChildren: './views/book/book-routing.module#BookRoutingModule'  
  },  
  {  
    path: '**',  
    redirectTo: 'landing'  
  }  
];
```

```
export const bookRouteList: Routes = [  
  {  
    path: 'search',  
    component: BookSearchComponent  
  },  
  {  
    path: '**',  
    redirectTo: 'search'  
  }  
];
```

Angular et le routing (Route Guards)

- Les "Guards" permettent de contrôler l'accès à une "route" (e.g. autorisation) ou le départ depuis une "route" (e.g. enregistrement ou publication obligatoire avant départ).
- Les "Guards" sont ajoutés au niveau de la configuration du "Routing"
- Une "Guard" d'activation est un service qui implémente l'interface CanActivate. Ce service doit donc implémenter la méthode canActivate. Cette méthode est appelée à chaque demande d'accès à la "route" ; elle doit alors retourner une valeur de type boolean ou Promise<boolean> ou Observable<boolean> indiquant si l'accès à la "route" est autorisé ou non. Il est donc possible d'attendre le résultat d'un traitement asynchrone pour décider d'autoriser l'accès ou non.

```
{  
  path: 'cart',  
  loadChildren: './views/cart/cart-routing.module#CartRoutingModule',  
  canActivate: [  
    IsSignedInGuard  
  ]  
},
```

```
export class IsSignedInGuard implements CanActivate {  
  
  constructor(private _session: Session) {  
  }  
  
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {  
    return this._session.isSignedIn();  
  }  
}
```

Suite TP

- **Ajouter un module Login pour accéder au module panier**

Gestion Formulaire en Angular

- Il existe différentes façon d'implémenter les formulaires avec Angular.
- **Template-driven Forms (à éviter) :** Cette approche a de nombreuses limitations et s'avère rapidement fastidieuse à implémenter, peu extensible et peu efficace.
- **Reactive Forms (à adopter) :** cette approche vient appuyer le paradigme "Reactive Programming" qui fait parti des fondements d'Angular avec : une meilleure séparation de la logique du formulaire et de la vue, une meilleure testabilité, des Observables, la génération dynamique de formulaires

Gestion Formulaire (ReactiveForms)

- **Angular offre une approche originale et efficace nommée "Reactive Forms" présentant les avantages suivants :**
- **La logique des formulaires se fait dans le code TypeScript.**
- **Le formulaire devient alors plus facile à tester et à générer dynamiquement.**
- **Les "Reactive Forms" utilisent des Observables pour faciliter et encourager le "Reactive Programming"**

Gestion Formulaire (ReactiveForms, Création Formulaire)

- Pour créer un formulaire avec ReactiveForms, on utilise le module ReactiveFormsModule
- Chaque champ est un objet de type FormControl
- Les FormControl peuvent être regrouper avec un FormGroup

```
export class BookFormComponent {  
  
  bookForm = new FormGroup({  
    title: new FormControl(),  
    description: new FormControl()  
  });  
  
  submitBook() {  
    console.log(this.bookForm.value);  
  }  
  
}
```

```
<form  
  [formGroup]="bookForm"  
  (ngSubmit)="submitBook()">  
  
  <input  
    type="text"  
    formControlName="title">  
  
  <textarea formControlName="description"></textarea>  
  
  <button type="submit">SUBMIT</button>  
  
</form>
```


Gestion Formulaire (ReactiveForms,Validation)

- Les constructeurs des "controls" (FormControl, FormGroup et FormArray) acceptent en second paramètre une liste de fonctions de validation appelées "validators".
- Les "validators" natifs d'Angular sont regroupés sous forme de méthodes statiques dans la classe Validators
- démo

Suite TP

- **Ajouter un module pour Ajouter et modifier des produits de notre application, seulement si on est correctement connecté**

Programme Partie 3

- **Module HttpClient**
- **Cycle de vie d'un composant**
- **Pipe**
- **Testing en Angular**

Angular HttpClient

- Dans une application Angular, la plupart des données proviennent d'API ReST où les échanges de données se font via des requêtes HTTP.
- Pour produire des requêtes HTTP, Angular fournit un service HttpClient qui est un wrapper de la classe native XMLHttpRequest
- Le service HttpClient a pour avantages :
 - simplifier l'implémentation d'échanges HTTP,
 - fournir les outils nécessaires pour faciliter l'implémentation des "tests unitaires",
 - se baser sur les Observables permettant ainsi d'annuler les requêtes si nécessaire, de suivre la progression d'"upload" et de "download".

Angular HttpClient

- **HttpClient** est un service Angular ; on peut donc le récupérer avec la **Dependency Injection**.
- Le service **HttpClient** fournit des méthodes pour exécuter des requêtes en fonction du verb HTTP (get, post...)
- Chaque requête est un observable qui permet de récupérer la réponse de la requêtes

```
constructor(private _httpClient: HttpClient) { imports: [  
}                                             HttpClientModule,
```

```
this._httpClient.get<GoogleVolumeListResponse>(this._bookListUrl)  
  .subscribe(googleVolumeListResponse => {  
  
    this.bookCount = googleVolumeListResponse.totalItems;  
    this.bookList = googleVolumeListResponse.items.map(item => new Book({  
      title: item.volumeInfo.title  
    }));  
  
  });
```

Angular HttpClient

- **HttpClient** est un service Angular ; on peut donc le récupérer avec la **Dependency Injection**.
- Le service **HttpClient** fournit des méthodes pour exécuter des requêtes en fonction du verb HTTP (get, post...)
- Chaque requête est un observable qui permet de récupérer la réponse de la requêtes

```
constructor(private _httpClient: HttpClient) { imports: [  
}                                             HttpClientModule,
```

```
this._httpClient.get<GoogleVolumeListResponse>(this._bookListUrl)  
  .subscribe(googleVolumeListResponse => {  
  
    this.bookCount = googleVolumeListResponse.totalItems;  
    this.bookList = googleVolumeListResponse.items.map(item => new Book({  
      title: item.volumeInfo.title  
    }));  
  
  });
```

Angular HttpClient

- **HttpClient** est un service Angular qui peut être injecter à l'intérieur d'un notre service
- **Démo**

Cycle de vie d'un composant

- Un composant a un cycle de vie géré par Angular lui-même.
- Angular gère la création, le rendu, les propriétés liées aux données.
- Il propose également des hooks qui nous permettent de répondre aux événements clés du cycle de vie.
- Voici l'inventaire complet de l'interface hook du cycle de vie:
 - `ngOnChanges` - appelé lorsqu'une valeur de liaison d'entrée change
 - `ngOnInit` - après les premiers `ngOnChanges`
 - `ngDoCheck` - après chaque exécution de détection de changement
 - `ngAfterContentInit` - après l'initialisation du contenu du composant
 - `ngAfterContentChecked` - après chaque vérification du contenu du composant
 - `ngAfterViewInit` - après l'initialisation des vues du composant
 - `ngAfterViewChecked` - après chaque vérification des vues d'un composant
 - `ngOnDestroy` - juste avant la destruction du composant
 - Démo

Pipe dans Angular

- Les Pipes sont des filtres utilisables directement depuis la vue afin de transformer les valeurs lors du « binding »
- Le Pipe async est un Pipe capable de consommer des Observables (ou Promise) en appelant implicitement la méthode subscribe (ou then) afin de "binder" les valeurs contenus dans l'Observable (ou la Promise).

```
<div>{{ user.firstName | slice:0:10 | lowercase }}</div>
```

```
<div>{{ user.firstName | lowercase }}</div>
```

Pipe dans Angular

- On peut créer nos propres pipe
- Pour créer un Pipe personnalisé, il faut :
- implémenter une classe suivant l'interface PipeTransform,
- décorer cette classe avec le décorateur @Pipe() en indiquant le nom du Pipe.
- ajouter la classe aux declarations (et exports) du module associé.

```
@Pipe({
  name: 'price'
})
export class PricePipe implements PipeTransform {

  transform(price: Price): string {

    if (price == null) {
      return null;
    }

    const amount = price.coefficient * Math.pow(10, price.exponent);

    return `${amount} ${price.currency}`;

  }

}
```

Pipe dans Angular Testing (Test unitaire)

- **Grâce à Angular CLI, tous les outils nécessaires à l'implémentation et l'exécution des tests unitaires sont installés et pré-configurés dès la création de l'application**
- **Pour déclencher les tests, on peut utiliser la commande du CLI**
- `ng test --watch`

Pipe dans Angular Testing (Test unitaire)

- Jasmine <https://jasmine.github.io/> est un framework (produit par Pivotal <https://pivotal.io/>, cf. <https://www.pivotaltracker.com>) extensible dédié aux tests sur "browser" et sur NodeJS.
- C'est le JUnit du JavaScript.
- Il inclut tout le nécessaire pour :
 - définir des suites de tests (fonctions describe et it),
 - implémenter des assertions de toute sorte (fonction expect),
 - implémenter rapidement des "Spies" (alias mocks) (fonctions createSpy, createSpyObj et spyOn).

Pipe dans Angular Testing (Test unitaire)

- Pour ajouter un test, il suffit de créer un fichier avec l'extension `.spec.ts` dans le dossier `src`.
- Plus exactement, la convention est de créer ce fichier dans le même dossier que le fichier contenant le code source à tester.
- Il suffit d'utiliser les 3 fonctions suivantes pour implémenter un premier test :
 - `describe` : pour définir une suite (ou groupe) de "specs".
 - `it` : pour définir une "spec" (ou un test).
 - `expect` : pour implémenter les assertions.

```
import { Calculator } from './calculator';

describe('Calculator', () => {

  it('should evaluate 2 + 3 + 4 to 9', () => {

    const calculator = new Calculator();

    expect(calculator.evaluate('2 + 3 + 4')).toEqual(9);

  });

});
```

Pipe dans Angular Testing (Test unitaire)

- Comme dans tous les frameworks de tests unitaires, on peut définir des logique de "setup" et de "tear down" avec respectivement les fonctions `beforeEach` et `afterEach`.
- `beforeEach` : permet d'inscrire une fonction de "setup" qui sera appelée avant chaque "spec". Les fonctions de "setup" permettent de préparer un environnement sain pour chaque "spec".
- `afterEach` : permet d'inscrire une fonction de "tear down" qui sera appelée après chaque "spec".
- Les fonctions de "tear down" permettent de nettoyer l'environnement ou encore exécuter des assertions pour s'assurer que les "tests" n'ont pas d'effets de bord (requête HTTP "mocked" non exécutées ou sans réponse).

```
import { Calculator } from './calculator';

describe('Calculator', () => {

  let calculator: Calculator;

  beforeEach(() => {
    calculator = new Calculator();
  });

  it('should evaluate 2 + 3 + 4 to 9', () => {
    expect(calculator.evaluate('2 + 3 + 4')).toEqual(9);
  });

});
```