Angular : composants et routage

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Angular

Pour créer un nouveau composant

```
ng generate component component-name
```

Angular

Pour créer un nouveau composant

```
ng generate component component-name
```

Ou utiliser le raccourci

```
ng g c component-name
```

Angular

Pour créer un nouveau composant

```
ng generate component component-name
```

Ou utiliser le raccourci

```
ng g c component-name
```


Pour placer un composant dans un répertoire x

```
ng g c x/component-name
```

Angular

Autres options de la commande `ng g c`

- `-inline-style` : permet de définir les styles **CSS** du composant directement dans le fichier **TypeScript** du composant, plutôt que dans un fichier de style **CSS** séparé.
- `-inline-template` : permet de définir le template **HTML** du composant directement dans le fichier **TypeScript** du composant, plutôt que dans un fichier de template **HTML** séparé.
- `-skip-tests` : permet de créer un composant sans générer de fichiers de test unitaires.
- `-dry-run` : permet d'exécuter la commande **CLI** comme une simulation sans effectuer réellement de modifications sur le système de fichiers.
- ...

Angular

Pour appliquer certaines propriétés sur tous les composants générés, il faut modifier la section `schematics` **de** `angular.json`

```
"schematics": {
  "@schematics/angular:component": {
    "inlineTemplate": true,
    "inlineStyle": true
  }
},
```

Angular

Pour la suite, on va créer 3 composants

- `adresse`
- `stagiaire`
- `menu`

Angular

Commençons par créer le premier composant `adresse` dans un répertoire `components`

```
ng g c components/adresse
```

Angular

Commençons par créer le premier composant `adresse` dans un répertoire `components`

```
ng g c components/adresse
```

Résultat

```
CREATE src/app/components/adresse/adresse.spec.ts (558 bytes)
CREATE src/app/components/adresse/adresse.ts (200 bytes)
CREATE src/app/components/adresse/adresse.css (0 bytes)
CREATE src/app/components/adresse/adresse.html (23 bytes)
```

Angular

Ensuite `stagiaire`

```
ng g c components/stagiaire
```

Angular

Ensuite `stagiaire`

```
ng g c components/stagiaire
```

Résultat

```
CREATE src/app/components/stagiaire/stagiaire.spec.ts (572 bytes)
CREATE src/app/components/stagiaire/stagiaire.ts (208 bytes)
CREATE src/app/components/stagiaire/stagiaire.css (0 bytes)
CREATE src/app/components/stagiaire/stagiaire.html (25 bytes)
```

Angular

Et enfin `menu`

```
ng g c components/menu
```

Angular

Et enfin `menu`

```
ng g c components/menu
```

Résultat

```
CREATE src/app/components/menu/menu.spec.ts (537 bytes)
CREATE src/app/components/menu/menu.ts (188 bytes)
CREATE src/app/components/menu/menu.css (0 bytes)
CREATE src/app/components/menu/menu.html (20 bytes)
```

Angular

Constats : quatre fichiers créés pour chaque composant `x`

- `x.ts`
- `x.html`
- `x.css`
- `x.spec.ts`

Angular

Pour afficher le contenu de ces trois composants dans `app.html`

```
<app-stagiaire></app-stagiaire>
<app-adresse></app-adresse>
<app-menu></app-menu>
```

Angular

Pour afficher le contenu de ces trois composants dans `app.html`

```
<app-stagiaire></app-stagiaire>
<app-adresse></app-adresse>
<app-menu></app-menu>
```

Remarques

- En pratique, on évite généralement d'afficher tous les composants directement dans le composant principal.
- On préfère associer chaque composant à une route spécifique.
- Ainsi, le composant est chargé et affiché dans le composant principal uniquement lorsque son chemin correspond à l'URL de la requête **HTTP**.

Module de routage

- À la création du projet, on a eu un fichier de routage : `app.routes.ts`
- Ce fichier permet d'assurer le mapping chemin/composant
- Il contient un tableau vide de type `Routes`
- Chaque route peut avoir comme attributs (`path`, `component`, `redirectTo`, `children`...)

Contenu de `app.routes.ts`

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

export const routes: Routes = [];
```

Angular

Définissons les routes dans le tableau de routes du fichier `app.routes.ts` :

```
import { Routes } from '@angular/router';
import { Stagiaire } from './components/stagiaire/stagiaire';
import { Adresse } from './components/adresse/adresse';

export const routes: Routes = [
    { path: 'stagiaire', component: Stagiaire },
    { path: 'adresse', component: Adresse },
];
```

Angular

Définissons les routes dans le tableau de routes du fichier `app.routes.ts` :

```
import { Routes } from '@angular/router';
import { Stagiaire } from './components/stagiaire/stagiaire';
import { Adresse } from './components/adresse/adresse';

export const routes: Routes = [
    { path: 'stagiaire', component: Stagiaire },
    { path: 'adresse', component: Adresse },
];
```


Aucune route n'est définie pour le composant `Menu`, car celui-ci doit être affiché quel que soit le chemin demandé.

Angular

Remarques

- Les URLs saisies auront la forme : `localhost:4200/adresse` ou `localhost:4200/stagiaire`.
- Faut-il ajouter le préfixe `/` dans les valeurs de l'attribut `path` ?
- Non, car le préfixe `/` est déjà défini dans `index.html`, via la balise `<base href="/">`.

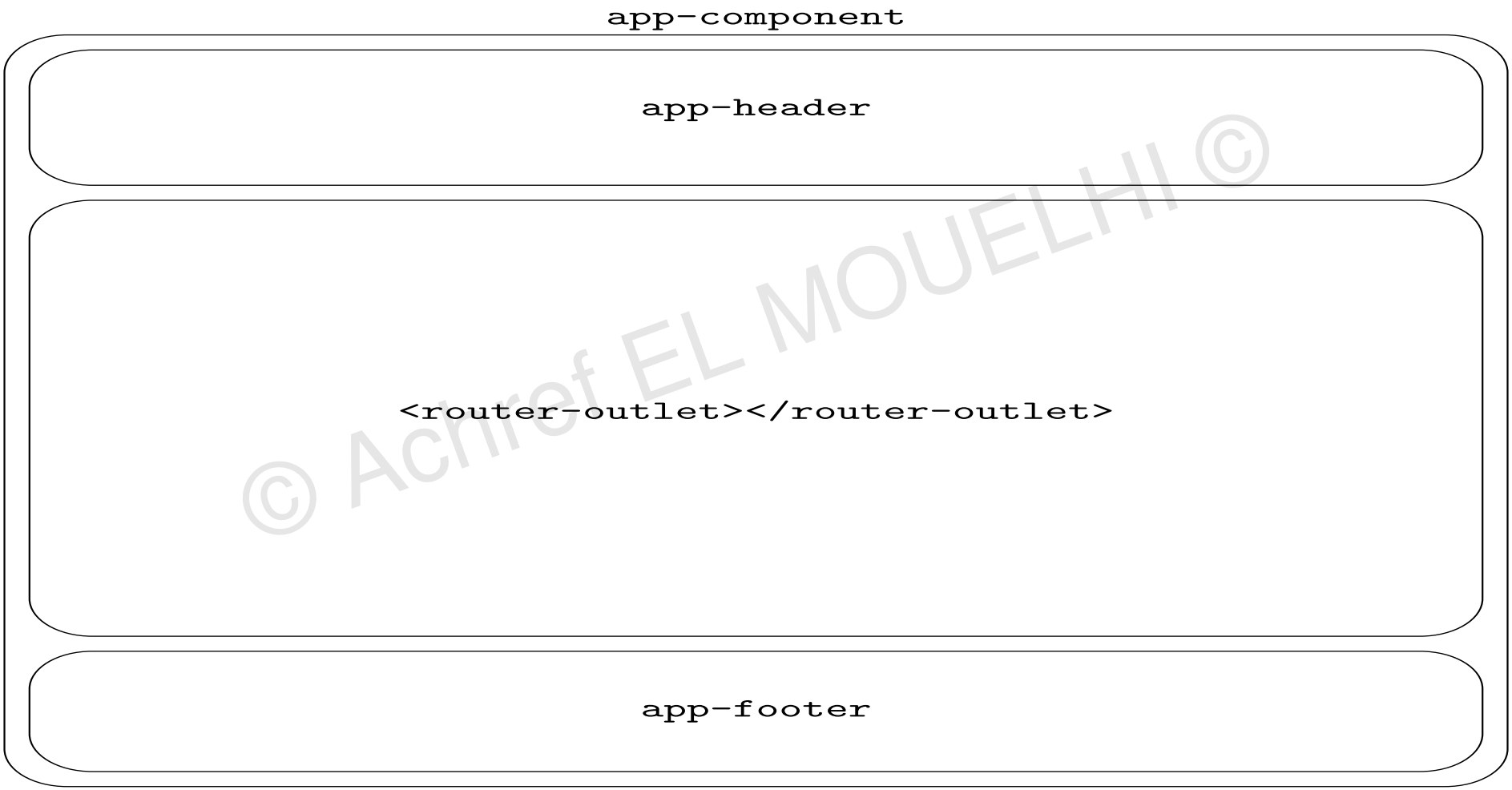
Angular

Où afficher les composants définis dans les routes ?

On doit indiquer un emplacement dans `app.html`, mais ce n'est pas le sélecteur du composant qui est utilisé.

Angular

On utilise la balise spéciale `<router-outlet>` dans `app.html`



Angular

Remarques

- Pour accéder à un composant, l'utilisateur doit connaître son chemin défini dans le tableau de routes (or ceci n'est pas vraiment très pratique)
- On peut plutôt définir un menu contenant des liens vers nos différents composants

Angular

Commençons par définir le menu suivant dans `menu.html`

```
<nav>
    <ul>
        <li><a href=''> Accueil </a></li>
        <li><a href='stagiaire'> Stagiaire </a></li>
        <li><a href='adresse'> Adresse </a></li>
    </ul>
</nav>
```

Angular

Dans `app.html`, on ajoute le menu et on indique l'emplacement des composants à afficher

```
<app-menu></app-menu>
<router-outlet></router-outlet>
```

Angular

Dans `app.html`, on ajoute le menu et on indique l'emplacement des composants à afficher

```
<app-menu></app-menu>
<router-outlet></router-outlet>
```

Dans `app.ts`, importons `RouterModule` et le composant `Menu`

```
import { RouterModule } from '@angular/router';
import { Menu } from "./components/menu/menu";

@Component({
  selector: 'app-root',
  imports: [CommonModule, RouterModule, Menu],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
```

Angular

Remarque

Avec `href`, chaque clic recharge toute la page : ce n'est pas le principe d'une application monopage (**SPA**).

Solution : utiliser `routerLink` à la place de `href`

```
<nav>
    <ul>
        <li><a routerLink=''> Accueil </a></li>
        <li><a routerLink='stagiaire'> Stagiaire </a></li>
        <li><a routerLink='adresse'> Adresse </a></li>
    </ul>
</nav>
```

Solution : utiliser `routerLink` à la place de `href`

```
<nav>
    <ul>
        <li><a routerLink=''> Accueil </a></li>
        <li><a routerLink='stagiaire'> Stagiaire </a></li>
        <li><a routerLink='adresse'> Adresse </a></li>
    </ul>
</nav>
```

Sans oublier d'importer `RouterLink` dans `menu.ts`

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

@Component({
  selector: 'app-menu',
  imports: [RouterLink],
  templateUrl: './menu.html',
  styleUrl: './menu.css'
})
export class Menu {}
```

Angular

Pour afficher la route active en gras

```
<ul>
    <li><a routerLink=''> Accueil </a></li>
    <li routerLinkActive=active>
        <a routerLink='stagiaire'> Stagiaire </a>
    </li>
    <li routerLinkActive=active>
        <a routerLink='adresse'> Adresse </a>
    </li>
</ul>
```

Angular

Pour afficher la route active en gras

```
<ul>
    <li><a routerLink=''> Accueil </a></li>
    <li routerLinkActive=active>
        <a routerLink='stagiaire'> Stagiaire </a>
    </li>
    <li routerLinkActive=active>
        <a routerLink='adresse'> Adresse </a>
    </li>
</ul>
```


Dans `menu.css`**, il faut définir la classe** `active`

```
.active {
    font-weight: bold;
}
```

Angular

Sans oublier d'importer `RouterLinkActive` **dans** `Menu.ts`

```
import { Component } from '@angular/core';
import { RouterLink, RouterLinkActive } from '@angular/router';

@Component({
  selector: 'app-menu',
  imports: [RouterLink, RouterLinkActive],
  templateUrl: './menu.html',
  styleUrl: './menu.css'
})
export class Menu {}
```

Angular

Si on ajoute `routerLinkActive=active`, il sera en gras quelle que soit la page visitée, pour cela on ajoute `[routerLinkActiveOptions]="{ exact: true }"` pour que la classe soit uniquement ajoutée lorsque la route correspond exactement à la valeur de `routerLink`

```
<ul>
    <li
        routerLinkActive=active
        [routerLinkActiveOptions]="{ exact: true }"
    >
        <a routerLink=''> Accueil </a>
    </li>
    <li routerLinkActive=active>
        <a routerLink='stagiaire'> Stagiaire </a>
    </li>
    <li routerLinkActive=active>
        <a routerLink='adresse'> Adresse </a>
    </li>
</ul>
```

Angular

Deux formes de paramètres de route

- `/chemin/param1/param2`
- `/chemin?var1=value1&var2=value2`

Angular

Deux formes de paramètres de route

- `/chemin/param1/param2`
- `/chemin?var1=value1&var2=value2`

Pour ces deux formes de paramètre

- Deux manières différentes de définir les routes
- Deux objets différents permettant de récupérer les valeurs respectives
  - `paramMap` et `params` pour `/chemin/param1/param2`
  - `queryParamMap` et `queryParams` pour `/chemin?var1=value1&var2=value2`

Angular

Définissons une route de la forme `/chemin/param1/param2` **dans** `app.routes.ts`

```
import { Routes } from '@angular/router';
import { Stagiaire } from './components/stagiaire/stagiaire';
import { Adresse } from './components/adresse/adresse';

export const routes: Routes = [
    { path: 'stagiaire', component: Stagiaire },
    { path: 'stagiaire/:nom/:prenom', component: Stagiaire },
    { path: 'adresse', component: Adresse },
];
```

Angular

Pour récupérer les paramètres d'une route ayant la forme `stagiaire/:nom/:prenom`, il faut :

- aller dans le composant concerné (ici, `stagiaire.ts`)
- faire une injection de dépendance de la classe `ActivatedRoute` (comme paramètre de constructeur)
- utiliser un objet de cette classe dans la méthode `ngOnInit()`
  - soit par l'intermédiaire d'un objet `paramMap` (Map)
  - soit par l'intermédiaire d'un objet `params` (Object)
- choisir une des deux solutions suivantes
  - solution asynchrone avec les observables
  - solution synchrone avec les snapshot

Angular

Pour récupérer les paramètres d'une route de la forme `stagiaire/:nom/:prenom`**, dans** `stagiaire.ts`

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

@Component({
  selector: 'app-stagiaire',
  templateUrl: './stagiaire.html',
  styleUrl: './stagiaire.css'
})
export class Stagiaire {
  nom = "";
  prenom = "";

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.route.paramMap.subscribe(res => {
      this.nom = res.get('nom') ?? "";
      this.prenom = res.get('prenom') ?? "";
      console.log(res);
    });
  }
}
```

Angular

Contenu de `stagiaire.html`

```
<h2>Stagiaire</h2>
<p> Bonjour {{ prenom }} {{ nom }} </p>
```

Angular

Pour récupérer les paramètres d'une route dans un objet JavaScript, on utilise `params`

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

@Component({
  selector: 'app-stagiaire',
  templateUrl: './stagiaire.html',
  styleUrl: './stagiaire.css'
})
export class Stagiaire {
  nom = "";
  prenom = "";

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.route.params.subscribe(res => {
      this.nom = res['nom']
      this.prenom = res['prenom']
      console.log(res)
    })
  }
}
```

Angular

La deuxième solution avec `snapshot` **et** `paramMap`

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

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

  nom = "";
  prenom = "";

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.nom = this.route.snapshot.paramMap.get('nom') ?? ''
    this.prenom = this.route.snapshot.paramMap.get('prenom') ?? ''
  }
}
```

Angular

Ou `snapshot` **et** `params`

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

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

  nom = "";
  prenom = "";

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.nom = this.route.snapshot.params['nom'];
    this.prenom = this.route.snapshot.params['prenom'];
    console.log(this.nom + ' ' + this.prenom);
  }
}
```

Angular

Pour récupérer les paramètres d'une route ayant la forme `adresse?ville=value1&rue=value2&codepostal=value3`, il faut :

- aller dans le composant concerné (ici, `adresse.ts`)
- faire une injection de dépendance de la classe `ActivatedRoute`
- utiliser un objet cette classe dans la méthode `ngOnInit()`
  - soit par l'intermédiaire d'un objet `queryParamMap` (Map)
  - soit par l'intermédiaire d'un objet `queryParams` (Object)
- choisir une des deux solutions suivantes
  - solution asynchrone avec les observables
  - solution synchrone avec les snapshot

Angular

Pas besoin de définir une route pour récupérer les paramètres `rue`, `codepostal` **et** `ville`

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

@Component({
  selector: 'app-adresse',
  imports: [],
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse implements OnInit {
  rue = '';
  codePostal = '';
  ville = '';

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.route.queryParamMap.subscribe(
      res => {
        this.ville = res.get('ville') ?? '';
        this.rue = res.get('rue') ?? '';
        this.codePostal = res.get('codepostal') ?? '';
      }
    );
  }
}
```

Angular

Dans `adresse.html`

```
<h2>Adresse</h2>
<ul>
    <li>Rue : {{ rue }} </li>
    <li>Code Postal : {{ codePostal }} </li>
    <li>Ville : {{ ville }} </li>
</ul>
```

Angular

Ou avec l'objet `queryParams`

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

@Component({
  selector: 'app-adresse',
  imports: [],
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse implements OnInit {
  rue = '';
  codePostal = '';
  ville = '';

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.route.queryParams.subscribe(res => {
      this.rue = res['rue']
      this.codePostal = res['codePostal']
      this.ville = res['ville']
    })
  }
}
```

Angular

Une première solution synchrone avec `snapshot` et `queryParamMap`

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

@Component({
  selector: 'app-adresse',
  imports: [],
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse implements OnInit {
  rue = '';
  codePostal = '';
  ville = '';

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.rue = this.route.snapshot.queryParamMap.get('rue') ?? ''
    this.ville = this.route.snapshot.queryParamMap.get('ville') ?? ''
    this.codePostal = this.route.snapshot.queryParamMap.get('codePostal') ?? ''
  }
}
```

Angular

Ou `snapshot` **et** `queryParams`

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

@Component({
  selector: 'app-adresse',
  imports: [],
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse implements OnInit {
  rue = '';
  codePostal = '';
  ville = '';

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.ville = this.route.snapshot.queryParams['ville'];
    this.rue = this.route.snapshot.queryParams['rue'];
    this.codePostal = this.route.snapshot.queryParams['codepostal'];
  }
}
```

Angular

Avantages de `paramMap` et `queryParamMap`

- Plus robuste : évite les erreurs si la clé n'existe pas.
- `paramMap` et `queryParamMap` utilisent les mêmes méthodes (`get`, `getAll`, `has`), ce qui facilite la lecture du code.
- Recommandé par l'équipe **Angular** pour le typage strict.

Angular

Depuis Angular 14, `inject` a été introduit pour injecter les dépendances sans passer par le constructeur

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

@Component({
  selector: 'app-adresse',
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse implements OnInit {
  rue = '';
  codePostal = '';
  ville = '';

  route = inject(ActivatedRoute)


  ngOnInit(): void {
    this.ville = this.route.snapshot.queryParams['ville'];
    this.rue = this.route.snapshot.queryParams['rue'];
    this.codePostal = this.route.snapshot.queryParams['codepostal'];
  }
}
```

Angular

Injection de dépendance par constructeur

- Design pattern classique utilisé dans de nombreux frameworks frontend et backend.
- Pratique standard dans **Angular**, largement adoptée par la communauté.
- Facilite les tests unitaires : les dépendances peuvent être aisément remplacées par des mocks.
- Les dépendances sont déclarées explicitement, ce qui améliore la lisibilité et la maintenabilité du code.
- Peut paraître moins intuitif lorsque certaines dépendances ne sont pas utilisées immédiatement.

Utilisation de `inject()`

- Permet d'injecter des dépendances directement dans le corps d'une classe, d'une méthode ou d'un hook.
- Peut être utilisé dans des services fonctionnels ou des fonctions autonomes.
- Moins répandu et donc parfois moins compris, ce qui peut rendre le code moins accessible à d'autres développeurs.
- Peut compliquer les tests unitaires et rendre plus difficile le remplacement des dépendances par des mocks.

Angular

Exercice

- Considérons un composant `calcul` (à créer)
- Ce composant est accessible via le chemin `calcul/:op` (à définir dans `app.routes.ts`)
- Les valeurs possibles de `op` sont `plus`, `moins`, `fois` et `div`
- Si l'adresse saisie dans la barre d'adresse contient `/calcul/plus?value1=2&value2=5`, la réponse attendue dans le template est `2 + 5 = 7`

Une solution possible

```
export class CalculComponent implements OnInit {
  value1 = 0
  value2 = 0
  operateur = ''
  resultat = 0
  op = ''
  erreur: string | null = null

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    const operators = {
      'plus': '+',
      'moins': '-',
      'fois': '*',
      'div': '/'
    }
    combineLatest([
      this.route.paramMap,
      this.route.queryParamMap
    ]).subscribe(([params, query]) => {
      this.op = params.get('op') ?? 'plus'
      this.value1 = Number(query.get('value1'))
      this.value2 = Number(query.get('value2'))
      if (!Object.keys(operators).includes(this.op)) {
        this.erreur = 'Opérateur invalide'
      } else {
        this.operateur = Object.values(operators)[Object.keys(operators).indexOf(this.op)]
        this.resultat = calculer(this.value1, this.value2, this.operateur)
      }
  }
}
```

Angular

Et la fonction `calculer`

```
function calculer(a: number, b: number, op: string): number {
  switch (op) {
    case '+': return a + b
    case '-': return a - b
    case '/': return a / b
    default: return a * b
  }
}
```

Angular

Remarque

`combineLatest` permet de réagir simultanément à plusieurs observables (ici `paramMap` et `queryParamMap)` sans avoir à les imbriquer, et en gardant un code plus clair, plus court et plus performant.

Angular

`combineLatest` **nous évite les imbrications suivantes**

```
this.route.paramMap.subscribe(
  params => {
    this.op = params.get('op') ?? 'plus'
    this.route.queryParamMap.subscribe(query => {
      this.value1 = Number(query.get('value1'))
      this.value2 = Number(query.get('value2'))
      if (!Object.keys(operators).includes(this.op)) {
        this.erreur = 'Opérateur invalide'
      } else {
        this.operateur = Object.values(operators)[Object.keys(operators).indexOf(this.op)]
        this.resultat = calculer(this.value1, this.value2, this.operateur)
      }
    })
  }
)
```

Angular

Comparaison : deux `subscribe` vs `combineLatest`

Aspect	**Deux `subscribe` imbriqués**	**Avec `combineLatest`**
Structure du code	Deux abonnements imbriqués, difficiles à lire et à maintenir.	Un seul abonnement, code plus lisible et concis.
Synchronisation des flux	Nécessite de gérer manuellement la dépendance entre les observables.	`combineLatest` fournit automatiquement la dernière valeur de chaque observable.
Réactivité	Le calcul n'est relancé que si on le code explicitement dans chaque `subscribe`.	Le calcul est relancé dès qu'un des flux (`paramMap` ou `queryParamMap`) émet une nouvelle valeur.
Performances et propreté	Plusieurs abonnements simultanés $\Rightarrow$ risque de fuites mémoire.	Un seul flux géré proprement $\Rightarrow$ code plus sûr et plus efficace.
Testabilité	Plus complexe à tester à cause de la logique imbriquée.	Plus simple à simuler et à tester (un flux unique).

Angular

Création de liens avec paramètres : deux méthodes

- On prépare le lien dans la classe et on l'affiche dans le template.
- On prépare le lien et on l'affiche dans le template.

Angular

Commençons par ajouter le lien suivant dans `app.html`

```
<a routerLink='{{ lienStagiaire }}'> Stagiaire avec params 1 </a>
```

Angular

Commençons par ajouter le lien suivant dans `app.html`

```
<a routerLink='{{ lienStagiaire }}'> Stagiaire avec params 1 </a>
```

Explication

Logiquement, `{{ lienStagiaire }}` est un attribut de la classe `AppComponent`.

Angular

Commençons par ajouter le lien suivant dans `app.html`

```
<a routerLink='{{ lienStagiaire }}'> Stagiaire avec params 1 </a>
```

Explication

Logiquement, `{{ lienStagiaire }}` est un attribut de la classe `AppComponent`.

Dans `app.ts`**, on ajoute un attribut** `lienStagiaire`

```
param1 = 'wick';
param2 = 'john';
lienStagiaire = '';
constructor() {
    this.lienStagiaire = `/stagiaire/${this.param1}/${this.param2}`;
}
```

Angular

Modifions les liens dans le menu pour les styler si seulement si les routes sans paramètres sont actives

```
<ul>
    <li routerLinkActive=active [routerLinkActiveOptions]="{exact: true}">
        <a routerLink=''> Accueil </a>
    </li>
    <li routerLinkActive=active [routerLinkActiveOptions]="{exact: true}">
        <a routerLink='stagiaire'> Stagiaire </a>
    </li>
    <li routerLinkActive=active [routerLinkActiveOptions]="{exact: true}">
        <a routerLink='adresse'> Adresse </a>
    </li>
</ul>
```

Angular

Une deuxième écriture avec le one way binding (property binding)

```
<a [routerLink]='lienStagiaire'> Stagiaire avec params 2 </a>
```

Angular

Une deuxième écriture avec le one way binding (property binding)

```
<a [routerLink]='lienStagiaire'> Stagiaire avec params 2 </a>
```


Remarque

Rien à changer dans la `app.html`.

Angular

Une troisième écriture

```
<a [routerLink]="['/stagiaire',param1, param2]">
    Stagiaire avec params 3
</a>
```

Angular

Une troisième écriture

```
<a [routerLink]="['/stagiaire',param1, param2]">
    Stagiaire avec params 3
</a>
```


Remarque

Rien à changer dans la `app.html`.

Angular

Pour construire un chemin de la forme `/chemin?param1=value1&param2=value2`

```
<a [routerLink]="['/adresse']" [queryParams]="{ ville: 'Marseille',
   codepostal: '13000', rue: 'paradis'}">
    Adresse avec params
</a>
```

Angular

Exercice

- Dans `stagiaire.html`, construisez un lien vers la route `/stagiaire` avec deux paramètres
- Dans `stagiaire.ts`, utilisez la solution snapshot puis observable pour récupérer les paramètres. Dans `stagiaire.html`, on affiche les paramètres.
- Vérifier, en cliquant sur le lien, que les nouveaux paramètres sont affichés

Angular

Conclusion

- Si la valeur initiale de paramètre est utilisée seulement à l'initialisation du composant et ne risque pas de changer, utilisez les snapshot.
- Si la route risque de changer tout en restant dans le même composant, utilisez les observables. L'initialisation du composant (`ngOnInit()`) ne serait donc pas appelée à nouveau, l'observateur sera notifié lorsque l'URL est modifiée.

Angular

Exercice

- Créez un nouveau composant `tableau`.
- Dans ce composant, déclarez un tableau `numbers = [2, 3, 8, 5, 1]` comme attribut.
- Définissez une route `tableau/:indice` dans `app.routes.ts` (`indice` étant l'indice de l'élément à afficher).
- Ajoutez deux liens `suivant` et `précédent` qui permettent de naviguer respectivement sur l'élément suivant et précédent de `numbers`.
- Les deux liens `suivant` et `précédent` permettront une navigation circulaire.

Angular

Remarque

Depuis **Angular 16**, le décorateur `@Input` permet de récupérer directement les paramètres de route.

Angular

Simplifions `stagiaire.ts` **avec** `@Input`

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

@Component({
  selector: 'app-stagiaire',
  templateUrl: './stagiaire.html',
  styleUrl: './stagiaire.css'
})
export class Stagiaire {

  @Input() nom?: string;
  @Input() prenom?: string;

}
```

Angular

Simplifions `stagiaire.ts` avec `@Input`

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

@Component({
  selector: 'app-stagiaire',
  templateUrl: './stagiaire.html',
  styleUrl: './stagiaire.css'
})
export class Stagiaire {

  @Input() nom?: string;
  @Input() prenom?: string;

}
```

Pas de changement dans `stagiaire.html`

Angular

De même pour `adresse.ts`

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

@Component({
  selector: 'app-adresse',
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse {

  @Input() rue?: string;
  @Input() codePostal?: string;
  @Input() ville?: string;

}
```

Angular

De même pour `adresse.ts`

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

@Component({
  selector: 'app-adresse',
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse {

  @Input() rue?: string;
  @Input() codePostal?: string;
  @Input() ville?: string;

}
```

Pas de changement dans `adresse.html`

Angular

Il nous reste d'activer ce mode de récupération de paramètre dans `app.config.ts`

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners, provideZoneChangeDetection }
   from '@angular/core';
import { provideRouter, withComponentInputBinding } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideZoneChangeDetection({ eventCoalescing: true }),
    // provideRouter(routes)
    provideRouter(routes, withComponentInputBinding())

  ]
};
```

Angular

Remarque

La récupération des paramètres avec `@Input` permet une récupération asynchrone des données.

Angular

Pour modifier le titre en fonction du composant demandé, on utilise le service `Title`

```
import { Component, Input, OnInit } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'app-adresse',
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse implements OnInit {
  @Input() rue?: string;
  @Input() codePostal?: string;
  @Input() ville?: string;

  constructor(private title: Title) { }

  ngOnInit(): void {
    this.title.setTitle('Adresse')
  }
}
```

Angular

On peut aussi le faire dans le fichier de routage

```
{ path: 'stagiaire', component: StagiaireComponent, title: 'Trainee' },
{ path: 'adresse', component: AdresseComponent, title: 'Address' },
```

Angular

Pour remettre un titre par défaut pour tous les autres composants (code à placer dans le fichier de routage)

```
import { Routes } from '@angular/router';
import { Stagiaire } from './components/stagiaire/stagiaire';
import { Adresse } from './components/adresse/adresse';

export const routes: Routes = [
    { path: 'stagiaire', component: Stagiaire, title: 'Trainee' },
    { path: 'stagiaire/:nom/:prenom', component: Stagiaire },
    { path: 'adresse', component: Adresse },
];

routes
    .filter(elt => !elt.title)
    .map(elt => elt.title = 'CoursAngular');
```

Angular

Pour tester

- visitez la route `/adresse` et vérifiez que le `title` affiché est `Adresse`.
- visitez la route `/stagiaire` et vérifiez que le `title` affiché est `CoursAngular`.

Angular

Étapes

- Injecter la classe `Router` dans le constructeur de notre classe
- Utiliser l'objet de cette classe injectée dans n'importe quelle méthode de notre classe (`.ts`) pour réorienter vers un autre chemin

Angular

Quelques méthodes de la classe `Router`

- `createUrlTree` construit simplement un arbre d'**URL** (une représentation de la route) sans effectuer la navigation et retourne un objet `UrlTree`.
- `navigateByUrl` navigue vers une URL complète sous forme de chaîne ou `UrlTree`.
- `navigate` navigue en construisant l'**URL** à partir de segments ou paramètres (souvent relatifs à une route).
- ...

Angular

Dans adresse.ts

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

@Component({
  selector: 'app-adresse',
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse {

  @Input() rue?: string;
  @Input() codePostal?: string;
  @Input() ville?: string;

  nom = 'wick';
  prenom = 'john';

  constructor(private router: Router) { }

  goToStagiaire(): void {
    this.router.navigateByUrl('/stagiaire/' + this.nom + '/' + this.prenom);
  }
}
```

Angular

Dans adresse.html

```
<h2>Adresse</h2>
<ul>
    <li>Rue : {{ rue }} </li>
    <li>Code Postal : {{ codePostal }} </li>
    <li>Ville : {{ ville }} </li>
</ul>
<div>
    <button (click)="goToStagiaire()">
        Aller dans stagiaire
    </button>
</div>
```

Angular

Dans `adresse.html`

```
<h2>Adresse</h2>
<ul>
    <li>Rue : {{ rue }} </li>
    <li>Code Postal : {{ codePostal }} </li>
    <li>Ville : {{ ville }} </li>
</ul>
<div>
    <button (click)="goToStagiaire()">
        Aller dans stagiaire
    </button>
</div>
```


En cliquant sur le bouton, la méthode `goToStagiaire()` sera appelée et on sera redirigé vers `/stagiaire/john/wick`.

Angular

On peut aussi utiliser la méthode `navigate()`

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

@Component({
  selector: 'app-adresse',
  templateUrl: './adresse.html',
  styleUrl: './adresse.css'
})
export class Adresse {

  @Input() rue?: string;
  @Input() codePostal?: string;
  @Input() ville?: string;

  nom = 'wick';
  prenom = 'john';

  constructor(private router: Router) { }

  goToStagiaire() {
    this.router.navigate(['/stagiaire', this.nom , this.prenom]);
  }
}
```

Angular

Pour rediriger vers une route acceptant des `queryParams`

```
gotoMarseille() {
    this.router.navigate(
        ['/adresse'],
        {
            queryParams:
            {
                ville: 'Marseille',
                codePostal: '13008'
            }
        }
    )
}
```

Angular

L'option queryParamsHandling accepte

- merge : conserve les anciens paramètres et ajoute les nouveaux.
- preserve : garde les paramètres existants sans en ajouter.

Angular

L'option queryParamsHandling accepte

- merge : conserve les anciens paramètres et ajoute les nouveaux.
- preserve : garde les paramètres existants sans en ajouter.

Exemple

```
this.router.navigate(
    ['/adresse'],
    {
        queryParams: { ville: 'Toulon' },
        queryParamsHandling: 'merge'
    }
);
```

Angular

On peut rediriger vers un chemin existant

```
const routes: Routes = [
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:param1/:param2', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' }
];
```

Angular

Ou pour un lien avec paramètres

```
const routes: Routes = [
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:param1/:param2', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' },
  { path: 'trainee/:param1/:param2', redirectTo: '/stagiaire/:param1/:
    param2' },
];
```

Angular

On peut créer un chemin vide pour que l'URL `localhost:4200` soit accessible

```
const routes: Routes = [
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:param1/:param2', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' },
  { path: '', redirectTo: '/stagiaire', pathMatch: 'full' }
];
```

Angular

On peut créer un chemin vide pour que l'URL `localhost:4200` soit accessible

```
const routes: Routes = [
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:param1/:param2', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' },
  { path: '', redirectTo: '/stagiaire', pathMatch: 'full' }
];
```


Remarque

- Sans la partie `pathMatch: 'full'` (pour les chemins vides), toutes les routes déclarées après cette dernière ne seront pas accessibles.
- `pathMatch: 'full'` ne laisse donc passer que les requêtes dont le chemin correspond exactement au chemin vide
- La deuxième valeur possible pour `pathMatch` est `'prefix'`

Angular

On peut créer un composant `error` et l'afficher en cas de chemin inexistant

```
const routes: Routes = [
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:nom/:prenom', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' },
  { path: 'error', component: ErrorComponent },
  { path: '**', redirectTo: '/error' }
];
```

Angular

On peut créer un composant `error` et l'afficher en cas de chemin inexistant

```
const routes: Routes = [
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:nom/:prenom', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' },
  { path: 'error', component: ErrorComponent },
  { path: '**', redirectTo: '/error' }
];
```


Le chemin `**` doit être le dernier. Autrement, toutes les requêtes seront redirigées vers le composant `error`.

Angular

Remarques

- En cliquant sur `Accueil` dans le menu, c'est le composant `Error` qui s'affiche.
- La route vide n'existe plus.

Angular

Remarques

- En cliquant sur `Accueil` dans le menu, c'est le composant `Error` qui s'affiche.
- La route vide n'existe plus.

Problème

Si on définit la route `{ path: '', component: AppComponent }`, le composant principal sera affiché deux fois en demandant la route vide.

Angular

Remarques

- En cliquant sur `Accueil` dans le menu, c'est le composant `Error` qui s'affiche.
- La route vide n'existe plus.

Problème

Si on définit la route `{ path: '', component: AppComponent }`, le composant principal sera affiché deux fois en demandant la route vide.

Solution

Créer un composant `Home` et l'associer à la route vide.

Angular

Commençons par créer le composant `home`

```
ng g c components/home
```

Angular

Commençons par créer le composant `home`

```
ng g c components/home
```

Résultat

```
CREATE src/app/components/home/home.spec.ts (537 bytes)
CREATE src/app/components/home/home.ts (188 bytes)
CREATE src/app/components/home/home.css (0 bytes)
CREATE src/app/components/home/home.html (20 bytes)
```

Angular

Ensuite

Déplaçons les contenus respectifs de `app.ts` et `app.html` dans `home.ts` et `home.html`.

Dans `app.html`, gardons seulement le contenu suivant

```
<app-menu></app-menu>
<router-outlet></router-outlet>
```

Angular

Créons ensuite le composant `error`

```
ng g c components/error
```

Angular

Créons ensuite le composant `error`

```
ng g c components/error
```

Résultat

```
CREATE src/app/components/error/error.spec.ts (544 bytes)
CREATE src/app/components/error/error.ts (192 bytes)
CREATE src/app/components/error/error.css (0 bytes)
CREATE src/app/components/error/error.html (21 bytes)
```

Angular

Contenu de `error.html`

```
<h2>Page d'erreur</h2>
<p>La page demandée n'existe pas</p>
```

Angular

Assurez vous d'avoir le tableau `routes` suivant dans `app-routing.module.ts` avant de tester l'URL `localhost:4200`

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'stagiaire', component: StagiaireComponent },
  { path: 'stagiaire/:nom/:prenom', component: StagiaireComponent },
  { path: 'adresse', component: AdresseComponent },
  { path: 'trainee', redirectTo: '/stagiaire' },
  { path: 'error', component: ErrorComponent },
  { path: '**', redirectTo: '/error' }
];
```

Angular

Deux modes de chargement de composants avec **Angular**

- **Eager loading** : chargement de tous les composants
- **Lazy loading** : chargement sur demande avec les promesses

Angular

Pour charger dynamiquement un composant, il faut l'importer avec les promesses

```
{
  path: 'adresse',
  loadComponent: () => import('./components/adresse/adresse.component')
    .then(c => c.AdresseComponent)
},
```

Angular

Lazy loading : avantages

- **Amélioration des performances** : En retardant le chargement de ressources non essentielles, on réduit le temps de chargement initial de la page.
- **Optimisation des ressources** : En ne chargeant que les ressources nécessaires au fur et à mesure que l'utilisateur fait défiler la page ou interagit avec elle, on optimise l'utilisation des ressources système, notamment la puissance du processeur et la mémoire.
- **Économie de bande passante** : En retardant le chargement des ressources non essentielles, cela peut être particulièrement bénéfique pour les utilisateurs sur des connexions Internet lentes ou limitées.
- ...

Angular

Lazy loading : inconvénients

- **Complexité du code** : Gérer le chargement asynchrone des ressources nécessite une écriture de code plus détaillée et peut être difficile à comprendre.
- **Problème de référencement et de classement** : Les moteurs de recherche peuvent avoir des difficultés à indexer correctement le contenu chargé de manière asynchrone car les robots d'exploration peuvent ne pas attendre le chargement des ressources différées.
- **Compatibilité du navigateur** : Bien que la plupart des navigateurs modernes prennent en charge le lazy loading, des problèmes de compatibilité peuvent survenir avec des versions plus anciennes ou moins courantes.
- ...