

Chat Conversation

Note: This is purely the output of the chat conversation and does not contain any raw data, codebase snippets, etc. used to generate the output.

User Input

pour que je sois bien prepare a mon exam, donne moi la facon la plus memorisable pour ecrire un code sur papier , si on nous a demander d'ajouter un component, html, ts, de creation simple (inclusant tt les type vu dans le cours , routage, pipes , directives...)

*Listed directory [app](file:///home/sami/Desktop/frontend/EmiShop2/src/app) *

*Viewed [app.ts](file:///home/sami/Desktop/frontend/EmiShop2/src/app/app.ts) *

Planner Response

Pour réussir ton examen sur papier, la clé est d'avoir une "**Structure Réflexe**" que tu peux dessiner mécaniquement. Voici la méthode la plus mémorisable, découpée en 3 blocs logiques.

⌚ La "Recette Magique" (3 Blocs)

Imagine ton composant comme un être vivant :

1. **L'Entête (Le Cerveau)** : Les imports et le décorateur `@Component`.
 2. **La Classe (L'Action)** : Les variables, les `@Input/@Output` et les fonctions.
 3. **Le Template (Le Visage)** : Le HTML avec les directives et les pipes.
-

1. Le Bloc TS (TypeScript) - Le Cerveau

C'est ici que tu gères les données et la logique.

```
// 1. Les Imports
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { CommonModule } from '@angular/common'; // Obligatoire pour *ngIf, *ngFor

@Component({
  selector: 'app-user',
  standalone: true, // Très probable dans ton cours actuel
  imports: [CommonModule], // Liste ce que tu utilises dans le HTML (Pipes, Directives)
  templateUrl: './user.html'
})
export class UserComponent {
  // 2. Les Propriétés (Types de données)
  @Input() name: string = "Inconnu";           // Donnée entrante (du parent)
  @Output() onClick = new EventEmitter();        // Donnée sortante (vers le parent)

  status: boolean = true;                      // Variable locale pour le *ngIf
  items: string[] = ['A', 'B', 'C'];            // Pour le *ngFor
  prix: number = 25.5;                          // Pour les Pipes

  // 3. Les Fonctions (Logique)
  toggleStatus() {
    this.status = !this.status;
    this.onClick.emit(this.status); // Envoyer un signal au parent
  }
}
```

2. Le Bloc HTML - Le Visage

Regroupe tout ce qui touche à l'affichage (Directives, Pipes, Bindings).

```
<!-- INTERPOLATION : Afficher une variable -->
<h1>Bonjour {{ name | uppercase }}</h1> <!-- PIPE intégré -->

<!-- *NGIF : Affichage conditionnel -->
<p *ngIf="status">Le profil est actif.</p>

<!-- *NGFOR : Boucle -->
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>

<!-- BINDINGS (Les parenthèses et crochets) -->
<img [src]="'assets/logo.png'"> <!-- [Property] : Entrée de donnée -->
<button (click)="toggleStatus()"> <!-- (Event) : Sortie d'action -->
  Changer le prix : {{ prix | currency:'EUR' }} <!-- Autre PIPE -->
</button>

<!-- TWO-WAY BINDING (Si FormsModule est importé) -->
<input [(ngModel)]="name"> <!-- La "banane dans la boîte" [( )] -->
```

3. Le Bloc Routage - La Carte

Souvent demandé en une seule ligne dans les fichiers app.routes.ts.

```
// Structure simple d'une route
{ path: 'user/:id', component: UserComponent }
```

Astuces de Mémorisation pour le Papier

1. **Le Mnémonique des Bindings :**
 - [] = PropRIÉTÉ (Les crochets ressemblent à une boîte qui contient la donnée).
 - () = ÉVÉNEMENT (Les parenthèses sont rondes comme un bouton sur lequel on clique).
 - [()] = Les deux (La banane dans la boîte).
2. **L'astuce de l'étoile *** : Souviens-toi que l'étoile (*ngIf, *ngFor) signifie que tu modifies la **structure** du HTML (tu enlèves ou ajoutes des éléments).
3. **L'import CommonModule** : Sur papier, on oublie souvent l'import. Retiens : "Si j'utilise une directive de base (If/For), je dois importer CommonModule ou la directive spécifique".

Checklist Rapide à vérifier avant de rendre ta copie :

- J'ai mis le @ devant Component, Input, Output.
- J'ai ouvert/fermé toutes les balises HTML.
- Mes fonctions dans le TS n'ont pas le mot function (car c'est une classe).
- J'ai bien mis les deux points : pour les types (name: string).

User Input

DE MEME, GUIDE POUR FAIRE UN FORMULAIRE TRES SIMPLE

Planner Response

Voici le guide pour créer un **formulaire simple** sur papier. On utilise généralement l'approche "**Template-driven**" (avec ngModel) car c'est la plus rapide et la plus visuelle pour un examen.

Le Formulaire "Réflexe" (Le guide 1-2-3)

1. Le Bloc TS : L'importation CRUCIALRE

N'oublie jamais d'importer le module des formulaires, sinon `ngModel` ne marchera pas !

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms'; // ⚡ INDISPENSABLE

@Component({
  selector: 'app-contact',
  standalone: true,
  imports: [FormsModule], // On l'ajoute ici
  templateUrl: './contact.html'
})
export class ContactComponent {
  // Un seul objet pour regrouper les données
  user = {
    nom: '',
    email: ''
  };

  onSubmit() {
    console.log("Formulaire envoyé !", this.user);
    // Ici tu mets ta logique (ex: alert, appel service...)
  }
}
```

2. Le Bloc HTML : La structure "Banane"

C'est ici que tu lies tes champs à ton objet TS.

```
<!-- 1. Référence locale #f="ngForm" pour surveiller l'état du formulaire -->
<form #f="ngForm" (ngSubmit)="onSubmit()">

  <label>Nom :</label>
  <!-- 2. [(ngModel)] + l'attribut 'name' (obligatoire !) -->
  <input
    name="monNom"
    [(ngModel)]="user.nom"
    required
    #n="ngModel">

  <!-- Petit bonus : message d'erreur si le champ est vide et touché -->
  <span *ngIf="n.invalid && n.touched" style="color:red">Nom requis</span>

  <br>

  <label>Email :</label>
  <input
    name="monEmail"
    [(ngModel)]="user.email"
    required
    email>

  <br>

  <!-- 3. Désactiver le bouton si le formulaire est invalide -->
  <button type="submit" [disabled]="f.invalid">
    Envoyer
  </button>
```

```
</button>
```

```
</form>
```

💡 Les 3 Règles d'Or pour ne rien oublier sur papier :

1. **L'attribut name** : Sur un `<input>`, si tu mets `[(ngModel)]`, tu **dois** mettre un `name="..."`. C'est l'erreur n°1 en examen.
2. **La référence #f="ngForm"** : Elle te permet de demander à Angular : "Hé, est-ce que tout le formulaire est valide ?" via `f.invalid`.
3. **La "Banane dans la boîte" [()]** :
 - `[]` : Angular envoie la donnée vers l'input.
 - `()` : L'input renvoie la modification au TS.
 - Ensemble, ils font la synchronisation en temps réel.

🚀 Version "Flash" (Si tu es pressé par le temps) :

Si le prof demande juste "Comment lier un input à une variable ?", dessine juste ceci :

```
<input [(ngModel)]="maVariable" name="v1">
```

*Et n'oublie pas de dire à l'oral (ou d'écrire à côté) : "Il faut importer **FormsModule** dans le composant".*

User Input

DE MEME POUR UNE CONFIGURATION DE ROUTAGE SIMPLE

Planner Response

Pour le routage, il faut voir ça comme un "**GPS**" : tu définis des destinations (les URLs) et les composants correspondants. Voici la configuration minimale à retenir pour le papier.

🌐 Le Routage "Réflexe" (3 Étapes)

1. Définir les routes (`app.routes.ts`)

C'est le tableau de correspondance entre l'URL et le Composant.

```
import { Routes } from '@angular/router';
import { Home } from './home';
import { Details } from './details';

export const routes: Routes = [
  // 1. Route par défaut (Redirection)
  { path: '', redirectTo: 'home', pathMatch: 'full' },

  // 2. Route simple
  { path: 'home', component: Home },

  // 3. Route avec PARAMÈTRE (l'id)
  { path: 'product/:id', component: Details },

  // 4. Route "404" (toujours à la fin !)
  { path: '**', component: NotFound }
];
```

2. Afficher et Naviguer (app.html)

Dans ton fichier HTML principal, tu dois dire à Angular où afficher les composants et comment changer de page.

```
<nav>
  <!-- ▲ Pas de href ! On utilise routerLink -->
  <a routerLink="/home">Accueil</a>

  <!-- Navigation avec paramètre dynamique -->
  <a [routerLink]="['/product', 5]">Voir Produit 5</a>
</nav>

<!-- ▲ LE MARQUEUR : Détermine où le composant s'affiche -->
<router-outlet></router-outlet>
```

3. Récupérer un paramètre (details.ts)

Si tu as une route comme `/product/:id`, voici comment récupérer l'ID dans le code.

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

@Component({ ... })
export class Details {
  // On "injecte" le service de la route active
  route = inject(ActivatedRoute);
  productId: string | null = null;

  ngOnInit() {
    // On récupère le paramètre 'id' (celui défini dans app.routes.ts)
    this.productId = this.route.snapshot.paramMap.get('id');
  }
}
```

💡 Astuces pour ne pas se tromper sur papier :

1. **Le router-outlet** : C'est comme un cadre photo vide. Si tu l'oublies, tes composants ne s'afficheront jamais !
2. **Le path: '*'** : C'est le joker. Il doit être **le dernier** du tableau, sinon il "mange" toutes les autres routes.
3. **L'importation** : Dans le fichier `app.ts` (le composant racine), n'oublie pas d'importer `RouterOutlet` et `RouterLink` dans le tableau `imports: [...]`.
4. **href vs routerLink** :
 - `href="/home"` : **MAUVAIS** (recharge toute la page).
 - `routerLink="/home"` : **BON** (navigation fluide sans rechargeement).

🚀 Memo express (Le strict minimum) :

- **Tableau**: `[{ path: '...', component: ... }]`
- **Lien**: ``
- **Sortie**: `<router-outlet></router-outlet>`
- **Paramètre**: `path: 'user/:id' → inject ActivatedRoute`

User Input

DE MEME POUR UNE CONFIGURATION SERVICE , AVEC LES DIRECTIVES INCLUSES , ET LES PIPES AUSSI (AVEC LE MAX DES COMMENTAIRES , JE SUIS UN STUPIDE)

Planner Response

 Guide : Service + Directive + Pipe (Angular – “Tout-en-un”)

⚠ Ce mémo est pensé pour être copié à la main sur papier; chaque bloc comporte le maximum de commentaires afin que rien ne te glisse-pas.

1 Service – La logique métier réutilisable**Fichier:** `src/app/services/data.service.ts`

```
/* -----
   SERVICE : DataService
-----
  • Fournit les données (API, mock, localStorage...) à tout le
    projet.
  • @Injectable({ providedIn: 'root' }) → singleton global.
----- */

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

/* -----
   Le décorateur indique à Angular que ce service peut être injecté.
   'root' = disponible partout, pas besoin de le déclarer dans un
   module.
----- */

@Injectable({
  providedIn: 'root' // <-- singleton global
})
export class DataService {

  // Exemple d'URL d'API (à remplacer par la tienne)
  private readonly API_URL = 'https://dummyjson.com/products';

  // Le HttpClient est injecté dans le constructeur → ready-to-use
  constructor(private http: HttpClient) {}

  /* -----
     Méthode GET : récupère la liste des produits.
     Retourne un Observable que le composant pourra souscrire.
----- */

  getProducts(): Observable<any> {
    return this.http.get<any>(this.API_URL);
  }

  /* -----
     Méthode POST : ajoute un produit (exemple d'écriture).
----- */

  addProduct(product: any): Observable<any> {
    return this.http.post<any>(this.API_URL, product);
  }

  /* -----
     Méthode utilitaire : filtre local (sans appel serveur).
----- */

  filterByCategory(products: any[], category: string): any[] {
    return products.filter(p => p.category === category);
  }
}
```

À retenir

- `@Injectable({providedIn: 'root'})` = le service est créé une fois et partagé.
- `HttpClient` → nécessite d'importer `HttpClientModule` dans le module racine (`app.module.ts`).
- Toutes les méthodes `return` un `Observable`; le composant les `subscribe`.

2 Directive – Le comportement réutilisable (ex.:highlight on hover)

Fichier: `src/app/directives-hover-highlight.directive.ts`

```
/*
 * -----
 * DIRECTIVE : HoverHighlightDirective
 *
 * -----
 * • Ajoute une classe CSS lorsqu'on survole l'élément.
 * • Utilisation : <div appHoverHighlight></div>
 * -----
 */
import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appHoverHighlight]'    // <-- attribut à mettre sur l'élément
})
export class HoverHighlightDirective {

  // Couleur par défaut (peut être sur-chargée via @Input si besoin)
  private readonly hoverColor = '#ffeb3b';

  constructor(
    private el: ElementRef,           // référence native de l'élément DOM
    private renderer: Renderer2       // API sécurisée pour manipuler le DOM
  ) {}

  /*
   * -----
   * HostListener écoute l'événement DOM 'mouseenter' (souris entre).
   * -----
   */
  @HostListener('mouseenter')
  onMouseEnter(): void {
    // Ajoute la couleur de fond
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', this.hoverColor);
  }

  /*
   * -----
   * HostListener écoute l'événement 'mouseleave' (souris sort).
   * -----
   */
  @HostListener('mouseleave')
  onMouseLeave(): void {
    // Réinitialise la couleur de fond (vide = valeur par défaut)
    this.renderer.removeStyle(this.el.nativeElement, 'backgroundColor');
  }
}
```

À retenir

- `selector: '[appHoverHighlight]'` → on l'utilise comme attribut.
- `HostListener` = écouteur d'événement intégré, pas besoin de `addEventListener`.
- `Renderer2` évite les problèmes de sécurité (XSS) et fonctionne côté serveur (SSR).

3 Pipe – Transformation de donnée (ex.:truncate text)

Fichier: `src/app/pipes-truncate.pipe.ts`

```

/*
----- PIPE : TruncatePipe -----
• Coupe une chaîne à N caractères et ajoute « ... » si besoin.
• Utilisation : {{ longText | truncate:30 }}
----- */

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

@Pipe({
  name: 'truncate'           // <-- nom à appeler dans le template
})
export class TruncatePipe implements PipeTransform {

/*
----- transform(value, limit) → renvoie la chaîne tronquée.
- value : texte d'entrée.
- limit : nombre max de caractères (par défaut 20).
----- */

transform(value: string | null | undefined, limit = 20): string {
  if (!value) return '';           // protection contre null/undefined
  return value.length > limit
    ? value.substring(0, limit) + '...' // ajoute l'ellipse
    : value;                         // sinon renvoie tel quel
}
}

```

À retenir

- Un **Pipe** doit implémenter **PipeTransform** et définir **transform**.
- Les **pipes pure** (par défaut) sont ré-exécutés uniquement quand l'entrée change – très performant.

■ Mise en place dans le module racine

Fichier: `src/app/app.module.ts` (ou `app.config.ts` si vous utilisez le mode *standalone*)

```

/*
----- MODULE : AppModule (ou AppConfig si standalone)
-----
• Déclare les directives et pipes.
• Importe HttpClientModule pour le service.
----- */

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http'; // <-- service HTTP

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

// Import de nos artefacts
import { HoverHighlightDirective } from './directives/hover-highlight.directive';
import { TruncatePipe } from './pipes/truncate.pipe';

@NgModule({
  declarations: [
    AppComponent,
    HoverHighlightDirective, // <-- directive déclarée
    TruncatePipe           // <-- pipe déclaré
  ],
  imports: [
    BrowserModule,
    HttpClientModule // <-- indispensable pour DataService
  ],
}

```

```

    providers: [],           // le service est déjà fourniIn: 'root'
    bootstrap: [AppComponent]
})
export class AppModule { }

```

⚡Si vous utilisez les **standalone components** (Angular≥14), vous n'avez plus besoin du **declarations**; il suffit d'ajouter **HoverHighlightDirective** et **TruncatePipe** dans le tableau **imports** du composant racine.

5 Exemple d'utilisation dans un composant

Fichier: src/app/components/product-list/product-list.component.ts

```

import { Component, inject, OnInit } from '@angular/core';
import { DataService } from '../../../../../services/data.service';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css'],
  // Si vous êtes en mode standalone, ajoutez les imports ici :
  // imports: [CommonModule, HoverHighlightDirective, TruncatePipe]
})
export class ProductListComponent implements OnInit {

  // Injection du service (singleton)
  private dataSrv = inject(DataService);

  // Tableau qui recevra les produits depuis le service
  products: any[] = [];

  ngOnInit(): void {
    // Souscription à l'Observable du service
    this.dataSrv.getProducts().subscribe(res => {
      this.products = res.products;    // dépend du format de l'API
    });
  }
}

```

Template: product-list.component.html

```

<!-- -----
  TEMPLATE : utilisation du service, de la directive et du pipe
-->
<ul>
  <!-- *ngFor = boucle sur le tableau -->
  <li *ngFor="let p of products"
      appHoverHighlight          <!-- ← directive appliquée -->
      style="margin: 8px; padding: 8px; border: 1px solid #ddd;">

    <!-- Utilisation du pipe truncate pour le titre long -->
    <strong>{{ p.title | truncate:30 }}</strong>

    <!-- Affichage du prix (exemple de pipe intégré) -->
    <span> - {{ p.price | currency:'EUR' }}</span>
  </li>
</ul>

```

Récapitulatif ultra-court (à recopier en bas de page)

Élément	Où le créer	Décorateur / Sélecteur	Points clés
Service	src/app/services/...	@Injectable({providedIn: 'root'})	Singleton, HttpClient, retourne Observable.
Directive	src/app/directives/...	@Directive({selector: '[appHoverHighlight]'}))	HostListener, Renderer2, ajoute/retire style.
Pipe	src/app/pipes/...	@Pipe({name: 'truncate'})	Implémente PipeTransform, pure par défaut.
Module	app.module.ts (ou app.config.ts)	imports: [HttpClientModule] + declarations: [HoverHighlightDirective, TruncatePipe]	Déclare les artefacts, importe HttpClientModule.
Composant	src/app/components/...	inject(DataService) + appHoverHighlight + truncate	Injection, boucle *ngFor, utilisation directive & pipe.

👉 Dernier conseil «stupide» (mais efficace)

Dessine toujours le diagramme suivant sur ton brouillon:

```
[Component] <-- inject --> [Service]
      |           |
      |---- uses --> [Directive]    |
      |---- uses --> [Pipe]        |
```

Cela te rappelle l'ordre d'injection et t'assure de ne rien oublier quand tu écris le code à la main. Bon courage pour ton examen! 🚀

User Input

FAIT DE MEME POUR CES PARTIES: 📝 1. Le Bloc TS (TypeScript) - Le Cerveau C'est ici que tu gères les données et la logique.

```
typescript // 1. Les Imports import { Component, Input, Output, EventEmitter } from '@angular/core'; import { CommonModule } from '@angular/common'; // Obligatoire pour *ngIf, *ngFor @Component({ selector: 'app-user', standalone: true, // Très probable dans ton cours actuel imports: [CommonModule], // Liste ce que tu utilises dans le HTML (Pipes, Directives) templateUrl: './user.html' }) export class UserComponent { // 2. Les Propriétés (Types de données) @Input() name: string = "Inconnu"; // Donnée entrante (du parent) @Output() onClick = new EventEmitter(); // Donnée sortante (vers le parent)
```

```
status: boolean = true; // Variable locale pour le *ngIf items: string[] = ['A', 'B', 'C']; // Pour le *ngFor prix: number = 25.5; // Pour les Pipes // 3. Les Fonctions (Logique) toggleStatus() { this.status = !this.status; this.onClick.emit(this.status); // Envoyer un signal au parent } } 🎉 2. Le Bloc HTML - Le Visage Regroupe tout ce qui touche à l'affichage (Directives, Pipes, Bindings).
```

html

Bonjour {{ name | uppercase }}

Le profil est actif.

- {{ item }}

 Changer le prix : {{ prix | currency:'EUR' }}   3. Le Bloc Routage - La Carte Souvent demandé en une seule ligne dans les fichiers app.routes.ts .

typescript // Structure simple d'une route { path: 'user/:id', component: UserComponent }  Astuces de Mémorisation pour le Papier
Le Mnémonique des Bindings :  PropRIÉTÉ (Les crochets ressemblent à une boîte qui contient la donnée).  = ÉVÉNEMENT (Les parenthèses sont rondes comme un bouton sur lequel on clique).  = Les deux (La banane dans la boîte). L'astuce de l'étoile * : Souviens-toi que l'étoile (*ngIf, *ngFor) signifie que tu modifies la structure du HTML (tu enlèves ou ajoutes des éléments). L'import CommonModule : Sur papier, on oublie souvent l'import. Retiens : "Si j'utilise une directive de base (If/For), je dois importer CommonModule ou la directive spécifique". Checklist Rapide à vérifier avant de rendre ta copie : J'ai mis le @ devant Component, Input, Output. J'ai ouvert/fermé toutes les balises HTML. Mes fonctions dans le TS n'ont pas le mot function (car c'est une classe). J'ai bien mis les deux points : pour les types (name: string).

Edited relevant file

Planner Response

Voici le **cheat-sheet complet** que tu peux recopier à la main: il couvre le **bloc TS (le cerveau)**, le **bloc HTML (le visage)**, le **routage**, ainsi que les **astuces de mémorisation** et une **check-list** rapide.

Cheat-Sheet:UserComponent (Angular)

1 BlocTS – Le Cerveau

```
// -----  
// [1] Imports - toujours les premiers lignes  
// -----  
import { Component, Input, Output, EventEmitter } from '@angular/core';  
import { CommonModule } from '@angular/common'; // Nécessaire pour *ngIf, *ngFor, etc.  
  
// -----  
// [2] Décorateur @Component  
// -----  
@Component({  
  selector: 'app-user', // <app-user></app-user>  
  standalone: true, // composant autonome (cours actuel)  
  imports: [CommonModule], // modules utilisés dans le template  
  templateUrl: './user.html' // fichier HTML du composant  
})  
export class UserComponent {  
  // -----  
  // [3] Propriétés - typage Angular  
  // -----  
  @Input() name: string = "Inconnu"; // donnée reçue du parent  
  @Output() onClick = new EventEmitter<boolean>(); // événement vers le parent  
  
  status: boolean = true; // pour *ngIf  
  items: string[] = ['A', 'B', 'C']; // pour *ngFor  
  prix: number = 25.5; // pour les pipes (currency)  
  
  // -----  
  // [4] Méthodes - logique du composant  
  // -----  
  toggleStatus(): void {  
    this.status = !this.status; // bascule true/false  
    this.onClick.emit(this.status); // informe le parent  
  }  
}
```

2 BlocHTML – Le Visage

```

<!-- -->
<!-- [1] Interpolation + Pipe intégré -->
<h1>Bonjour {{ name | uppercase }}</h1>

<!-- -->
<!-- [2] *ngIf - affichage conditionnel -->
<p *ngIf="status">Le profil est actif.</p>

<!-- -->
<!-- [3] *ngFor - boucle sur un tableau -->
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>

<!-- -->
<!-- [4] Bindings - [prop] et (event) -->
<img [src]="'assets/logo.png'"> <!-- [Property] -->
<button (click)="toggleStatus()">
  Changer le prix : {{ prix | currency:'EUR' }}
</button>

<!-- -->
<!-- [5] Two-Way Binding - [(ngModel)] (FormsModule requis) -->
<input [(ngModel)]="name" placeholder="Nom" />

```

[3] Bloc Routage – La Carte

```

// Dans app.routes.ts (ou un module de routing dédié)
import { Routes } from '@angular/router';
import { UserComponent } from './components/user/user';

export const routes: Routes = [
  // Route simple avec paramètre d'ID
  { path: 'user/:id', component: UserComponent }
];

```

Astuces de mémorisation (papier)

Astuce	Mnémotechnique
Bindings	[] = Prop (boîte qui contient la donnée). () = Event (bouton qui se déclenche). [()] = les deux → <i>banane dans la boîte</i> .
Étoile *	Signale structure (*ngIf, *ngFor). Elle modifie le DOM (ajoute / enlève des éléments).
CommonModule	Obligatoire dès que tu utilises *ngIf, *ngFor, ngClass, ... → ne l'oublie jamais.
Standalone	Si standalone: true → le composant n'a pas besoin d'être déclaré dans un module.
EventEmitter	Toujours typer l'événement (new EventEmitter<Type>()).
RouterLink vs href	routerLink = navigation Angular (pas de rechargement). href = page reload.

✓ Checklist rapide avant de rendre la copie

- ☒ @Component, @Input, @Output précédés du @.
- ☒ Toutes les balises HTML ouvertes sont fermées.
- ☒ Pas de mot-clé function dans les méthodes de classe (Angular utilise la syntaxe methodName() {}).
- ☒ Les types sont bien indiqués (: string, : boolean, : number).

- **CommonModule** importé dès qu'on utilise `*ngIf` / `*ngFor`.
 - **routerLink** utilisé pour la navigation, pas `href`.
 - **FormsModule** importé si on utilise `[(ngModel)]`.
-

Ce cheat-sheet est pensé pour être recopié à la main; chaque section est clairement séparée, très commentée et accompagnée d'une petite checklist pour éviter les oubli classiques.

User Input

REGROUPE LE TOUT , DANS UN FICHIER .md FULL ARSENAL FOR EXAM

Edited relevant file

Planner Response

✓Artifact créé

Le fichier demandé a été généré:

- **Chemin complet:** `file:///home/sami/.gemini/antigravity/brain/f7b90978-3d19-40ea-bc2a-97fb39652d12/full_exam_arsenal.md`

Il contient:

- Le composant **UserComponent** (TS + HTML)
- La configuration de **routing**
- Un **service** générique, une **directive** de survol, et un **pipe** de troncature
- Astuces de mémorisation et checklist de vérification

Vous pouvez maintenant ouvrir le fichier et le recopier à la main pour votre examen.

User Input

CP LE DANS LE CHEMIN DU DOSSIER DE L'APP

Edited relevant file

`*Viewed [full_exam_arsenal.md](file:///home/sami/.gemini/antigravity/brain/f7b90978-3d19-40ea-bc2a-97fb39652d12/full_exam_arsenal.md) *`

`*Viewed [full_exam_arsenal.md](file:///home/sami/.gemini/antigravity/brain/f7b90978-3d19-40ea-bc2a-97fb39652d12/full_exam_arsenal.md) *`

Edited relevant file

Planner Response

Le fichier **full_exam_arsenal.md** a été copié dans le répertoire de ton application Angular :

Chemin complet

`/home/sami/Desktop/frontend/EmiShop2/src/app/full_exam_arsenal.md`

Il contient :

- Le composant **UserComponent** (TS)
- Le template HTML correspondant
- La configuration de **routing**
- Un **service** générique (`DataService`)
- Une **directive** de survol (`HoverHighlightDirective`)
- Un **pipe** de troncature (`TruncatePipe`)
- Astuces de mémorisation et checklist de vérification

Tu peux maintenant ouvrir ce fichier dans ton éditeur et t'en servir comme référence pendant ton examen. Bonne révision !

