Introduction au module Angular

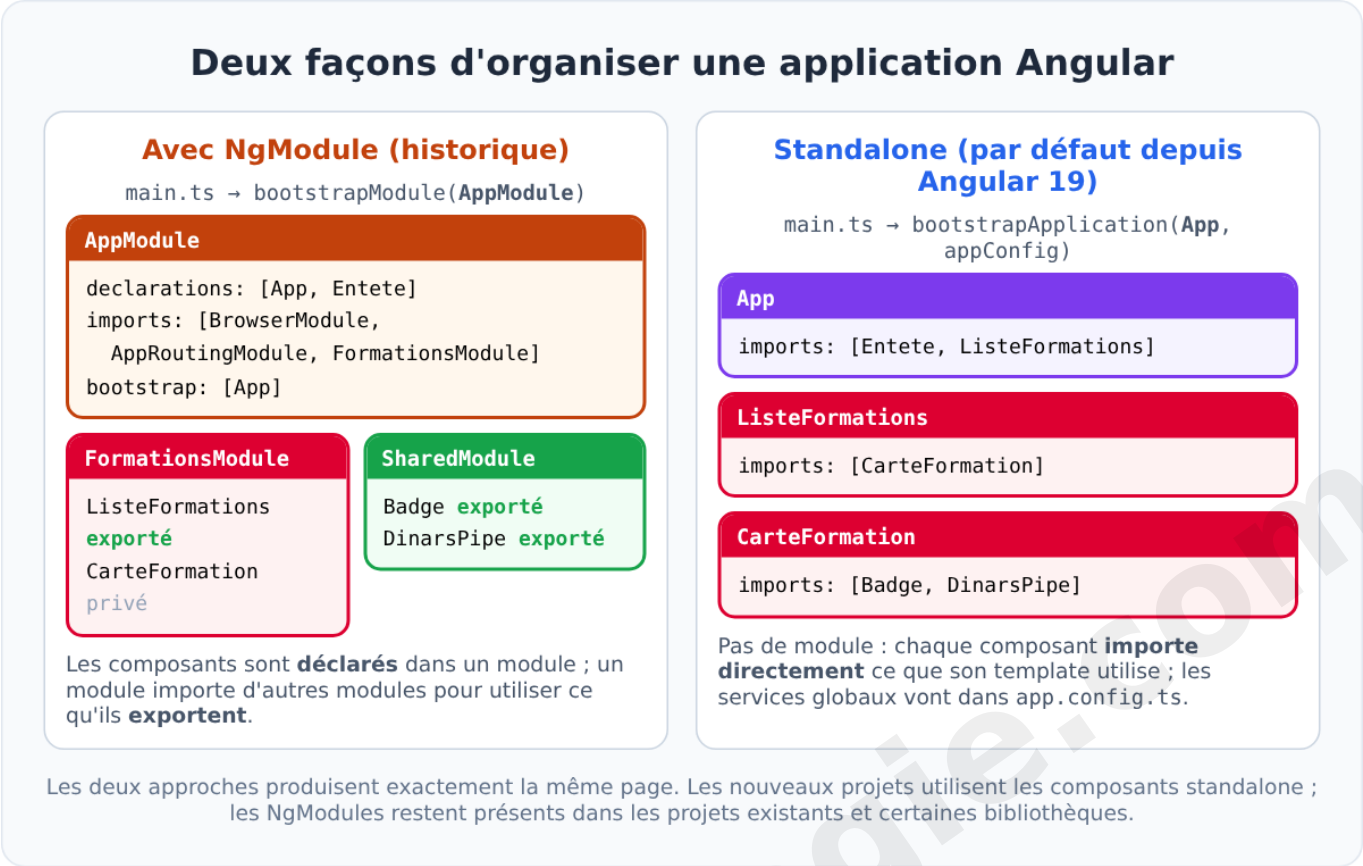
1. Objectifs

- Comprendre ce qu'est un **module Angular** (`NgModule`) et pourquoi il a longtemps été au cœur de toute application Angular
- Lire et écrire le **module racine** `AppModule`: `declarations`, `imports`, `exports`, `providers`, `bootstrap`
- Découper une application en **modules de fonctionnalité**, en **module partagé** et en **module de routage**
- Comprendre les règles de **visibilité**: ce qu'un module déclare, exporte et importe
- Fournir un **service** à l'application et charger un module à la demande (*lazy loading*)
- Faire le lien avec les **composants standalone**, utilisés par défaut depuis Angular 19, et migrer un projet à modules
- Reconnaître et corriger les erreurs typiques : `NG8001`, `NG8004`, `NG6007`, `NG6008`, `NG0201`

2. Présentation

- **Angular** est un framework écrit en **TypeScript**, livré sous forme de bibliothèques (`@angular/core`, `@angular/common`, `@angular/router`...) que l'on importe dans son application.
- Une application Angular contient plusieurs sortes de « briques » :
  - des **composants**: la partie visible de l'application (voir Introduction au component Angular) ;
  - des **services**: la logique et l'accès aux données, partagés entre composants grâce à l'**injection de dépendances**;
  - des **directives**: elles modifient le comportement ou l'apparence d'une balise ;
  - des **pipes**: ils mettent en forme une valeur dans un template (date, montant...).
- Il faut indiquer à Angular **quelles briques chaque template a le droit d'utiliser**. Pendant des années (Angular 2 à 14), on le faisait uniquement avec des **modules**: des classes décorées avec `@NgModule` qui regroupent des briques et les partagent.
- Depuis Angular 14, un composant peut être **autonome** (*standalone*) : il importe lui-même ce dont il a besoin, sans module. C'est le comportement **par défaut depuis Angular 19**, et c'est ce que génère `ng new` aujourd'hui.
- **Pourquoi apprendre les modules ?** Parce que vous les rencontrerez partout : dans les projets existants des entreprises, dans les tutoriels, et dans des bibliothèques qu'on importe encore sous forme de modules (`FormsModule`, `ReactiveFormsModule`...). Savoir lire un `NgModule` et le transformer en standalone fait partie du métier de développeur Angular.

apcpedagogie.com



1. Définition

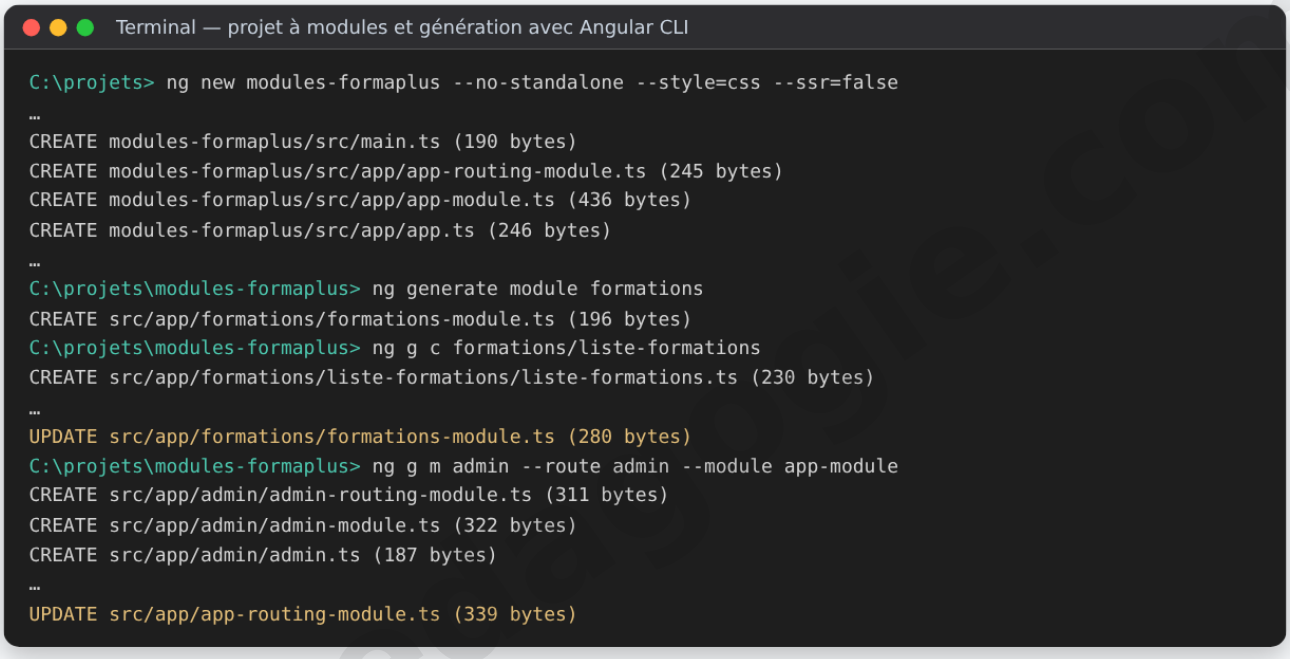
> *Un **module Angular** est une classe décorée avec `@NgModule` qui regroupe des composants, des directives et des pipes liés à une même fonctionnalité, indique de quels autres modules ils ont besoin, et choisit ce qu'il rend utilisable par le reste de l'application.*

- On peut comparer une application à modules à un **puzzle**: chaque module est une pièce qui regroupe une partie de l'application (l'accueil, les formations, l'administration…), et les pièces s'emboîtent par leurs `imports` et leurs `exports`.
- Un module joue trois rôles :
  - **regrouper**: il déclare les composants, directives et pipes qui lui appartiennent ;
  - **contrôler la visibilité**: ce qu'il déclare est privé, sauf ce qu'il exporte ;
  - **configurer**: il peut fournir des services et indiquer le composant de démarrage.
- **Ne pas confondre :** un **module Angular** (`@NgModule`) et un **module JavaScript** (un fichier avec `import` / `export`, voir Les classes et les modules ES6). Les deux coexistent : un fichier `formations-module.ts` est un module JavaScript qui contient un module Angular.

apcpedagogie.com

- **Créer un projet à modules**
  - Pour étudier les modules, on crée un projet avec l'option `--no-standalone`. La CLI génère alors un module racine au lieu de `app.config.ts`:

```
ng new modules-formaplus --no-standalone --style=css --ssr=false
cd modules-formaplus
code .
```



Dans un projet à modules, la CLI **déclare automatiquement** le nouveau composant dans le module le plus proche (ligne UPDATE).

  - Le démarrage se fait désormais par un **module**, et non plus par un composant :

```
// src/main.ts (projet à modules)
import { platformBrowser } from '@angular/platform-browser';
import { AppModule } from './app/app-module';

platformBrowser().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

```
// src/app/app.ts : un composant NON autonome
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: false,
  styleUrl: './app.css',
  templateUrl: './app.html',
})
export class App {}
```

- `standalone: false` signifie : « ce composant n'importe rien lui-même, il appartient à un module ». Les composants d'un projet à modules n'ont **pas** de tableau `imports`: c'est leur module qui fournit tout ce que leur template utilise.

Le module racine AppModule

- Toute application à modules a un **module racine**, appelé par convention `AppModule`. Voici celui de notre projet, une fois les composants ajoutés :

Anatomie du module racine AppModule

src/app/app-module.ts

```
import { NgModule, provideBrowserGlobalErrorListeners } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing-module';
import { FormationsModule } from './formations/formations-module';
import { App } from './app';
import { Entete } from './layout/entete/entete';

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

- **declarations** les composants, directives et pipes qui appartiennent au module (chacun dans un seul module)
- **imports** les autres modules dont les éléments exportés sont utilisés dans les templates du module
- **providers** les services et fournisseurs mis à disposition de l'application
- **bootstrap** le composant racine à afficher au démarrage (module racine seulement)
- **exports** (modules de fonctionnalité) ce que le module rend utilisable par les modules qui l'importent

- Il commence par importer (au sens JavaScript) `NgModule`, le décorateur, et `BrowserModule`, puis décrit le module avec les propriétés suivantes.

- **declarations**
  - La liste des **composants, directives et pipes** qui appartiennent à ce module. Ils peuvent s'utiliser entre eux dans leurs templates.
  - Chaque élément est déclaré dans **un seul** module (sinon erreur `NG6007`). On n'y met **jamais** de module ni de service, et jamais de composant standalone (erreur `NG6008`).

- **imports**
  - La liste des **autres modules** dont ce module a besoin. Les templates des composants déclarés peuvent alors utiliser tout ce que ces modules **exportent**.
  - Exemples courants :

Module importé	Ce qu'il apporte	Où l'importer
`BrowserModule`	le nécessaire pour exécuter l'application dans un navigateur (il réexporte `CommonModule`)	le module racine **uniquement**
`CommonModule`	les directives et pipes de base : `*ngIf`, `*ngFor`, `ngClass`, `date`, `currency`...	tous les autres modules
`FormsModule`	les formulaires pilotés par le template (`ngModel`)	là où il y a des formulaires
`ReactiveFormsModule`	les formulaires réactifs (`formGroup`, `formControlName`)	là où il y a des formulaires réactifs
`AppRoutingModule`	la configuration des routes et `<router-outlet>`	le module racine
vos modules (`FormationsModule`, `SharedModule`...)	ce qu'ils exportent	là où on en a besoin

  - **Les blocs `@if` et `@for`** font partie du langage des templates : ils fonctionnent sans `CommonModule`. Ce module reste nécessaire pour les anciennes directives (`*ngIf`, `*ngFor`...) et pour les pipes comme `date`: sans lui, `{{ d | date }}` donne `NG8004: No pipe found with name 'date'`.
  - **Code ancien :** vous verrez encore `imports: [BrowserModule, FormsModule, HttpModule]`. `HttpModule` a été supprimé depuis Angular 8, puis son remplaçant `HttpClientModule` est devenu obsolète : aujourd'hui, le client HTTP se fournit avec `provideHttpClient()` dans les `providers`.

apcpedagogie.com

- **exports**
  - Le **sous-ensemble des déclarations** (ou des modules importés) que ce module rend utilisable par les modules qui l'importent. Tout ce qui n'est pas exporté reste **privé**.
  - Le module racine n'exporte rien : personne ne l'importe. Les `exports` servent dans les modules de fonctionnalité et le module partagé.

- **providers**
  - Les **services** et fournisseurs que le module met à disposition. Dans le module racine, ils sont disponibles dans toute l'application. On y place aussi les fonctions de configuration d'Angular, comme `provideBrowserGlobalErrorListeners()` (générée par la CLI) ou `provideHttpClient()`.
  - Aujourd'hui, la plupart des services n'ont plus besoin d'être listés ici : ils se fournissent eux-mêmes (voir plus bas).

- **bootstrap**
  - Le **composant racine** à créer au démarrage et à insérer à la place de `<app-root>` dans `index.html`. Seul le module racine possède cette propriété.

- ## Un module de fonctionnalité
  - Un **module de fonctionnalité** (*feature module*) regroupe tout ce qui concerne une partie de l'application. On en crée un pour les formations, puis ses composants : la CLI les **déclare automatiquement** dans le module (ligne `UPDATE`).

```
ng generate module formations
ng g c formations/liste-formations
ng g c formations/carte-formation
ng g s formations/catalogue
ng g interface models/formation
```

  - Le module déclare ses deux composants, importe ce dont ils ont besoin, et n'**exporte que la liste**: la carte est un détail interne.

apcpedagogie.com

```
// src/app/formations/formations-module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { SharedModule } from '../shared/shared-module';
import { ListeFormations } from './liste-formations/liste-formations';
import { CarteFormation } from './carte-formation/carte-formation';

@NgModule({
  declarations: [ListeFormations, CarteFormation],
  imports: [CommonModule, SharedModule],
  exports: [ListeFormations],
})
export class FormationsModule {}
```

```
// src/app/models/formation.ts
export interface Formation {
  id: number;
  titre: string;
  niveau: 'Débutant' | 'Intermédiaire' | 'Avancé';
  dureeHeures: number;
  prix: number;
  nouveau: boolean;
}
```

```
// liste-formations.ts
import { Component, inject } from '@angular/core';
import { Catalogue } from '../catalogue';

@Component({
  selector: 'app-liste-formations',
  standalone: false,
  styleUrl: './liste-formations.css',
  templateUrl: './liste-formations.html',
})
export class ListeFormations {
  private readonly catalogue = inject(Catalogue);
  protected readonly formations = this.catalogue.lister();
}
```

apcpedagogie.com

```
<!-- liste-formations.html -->
<section>
  <h2>Nos formations ({{ formations.length }})</h2>
  <div class="cartes">
    @for (f of formations; track f.id) {
      <app-carte-formation [formation]="f" />
    }
  </div>
</section>
```

```
// carte-formation.ts
import { Component, input } from '@angular/core';
import { Formation } from '../../models/formation';

@Component({
  selector: 'app-carte-formation',
  standalone: false,
  styleUrl: './carte-formation.css',
  templateUrl: './carte-formation.html',
})
export class CarteFormation {
  formation = input.required<Formation>();
}
```

```
<!-- carte-formation.html -->
<article class="carte">
  <h3>
    {{ formation().titre }}
    @if (formation().nouveau) {
      <app-badge />
    }
  </h3>
  <p>{{ formation().niveau }} — {{ formation().dureeHeures }} heures</p>
  <p class="prix">{{ formation().prix | dinars }}</p>
</article>
```

- **`input.required<Formation>()`** permet à la liste de transmettre une formation à chaque carte avec `[formation]="f"`. Ce mécanisme est expliqué dans Communication entre composants ; ici, retenez seulement que `<app-badge>` et le pipe `dinars` viennent du module partagé.

apcpedagogie.com

- ## Un module partagé

  - Le **module partagé** (*shared module*) regroupe les petites briques réutilisées un peu partout : un badge, un pipe de mise en forme, un bouton… Il les **déclare et les exporte** toutes.

```
ng generate module shared
ng g c shared/badge
ng g pipe shared/dinars
```

```
// src/app/shared/shared-module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Badge } from './badge/badge';
import { DinarsPipe } from './dinars-pipe';

@NgModule({
  declarations: [Badge, DinarsPipe],
  imports: [CommonModule],
  exports: [Badge, DinarsPipe],
})
export class SharedModule {}
```

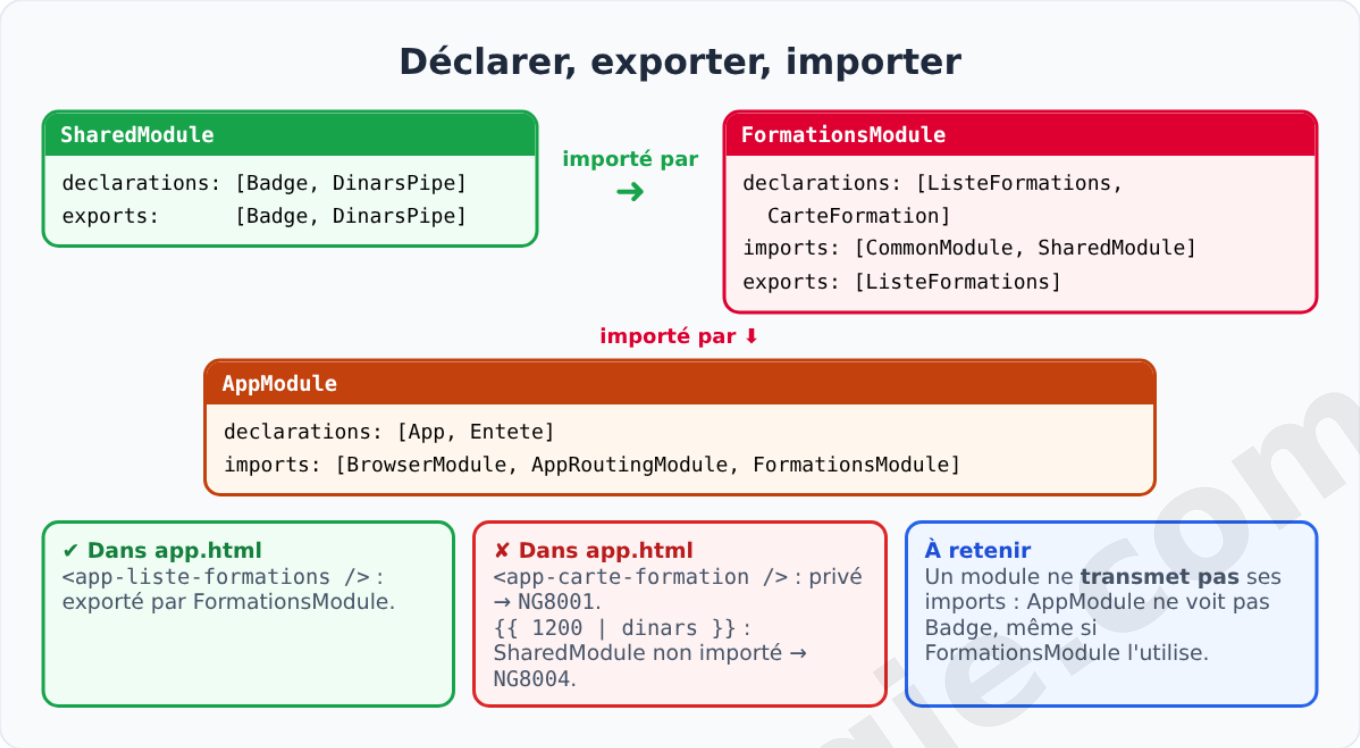
```
// src/app/shared/dinars-pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'dinars',
  standalone: false,
})
export class DinarsPipe implements PipeTransform {
  transform(montant: number): string {
    return `${montant.toFixed(3)} DT`;
  }
}
```

  - Le schéma suivant résume qui voit quoi. La règle essentielle : **un module ne transmet pas ses imports**. `FormationsModule` importe `SharedModule`, mais `AppModule` ne voit pas pour autant `Badge`. Pour lui donner accès, il faudrait importer aussi `SharedModule` dans `AppModule`, ou ajouter

apcpedagogie.com

`SharedModule` aux `exports` de `FormationsModule` (réexportation).



- Il reste à compléter le module racine et le template de `App` :

```
// src/app/app-module.ts
import { NgModule, provideBrowserGlobalErrorListeners } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing-module';
import { FormationsModule } from './formations/formations-module';
import { App } from './app';
import { Entete } from './layout/entete/entete';

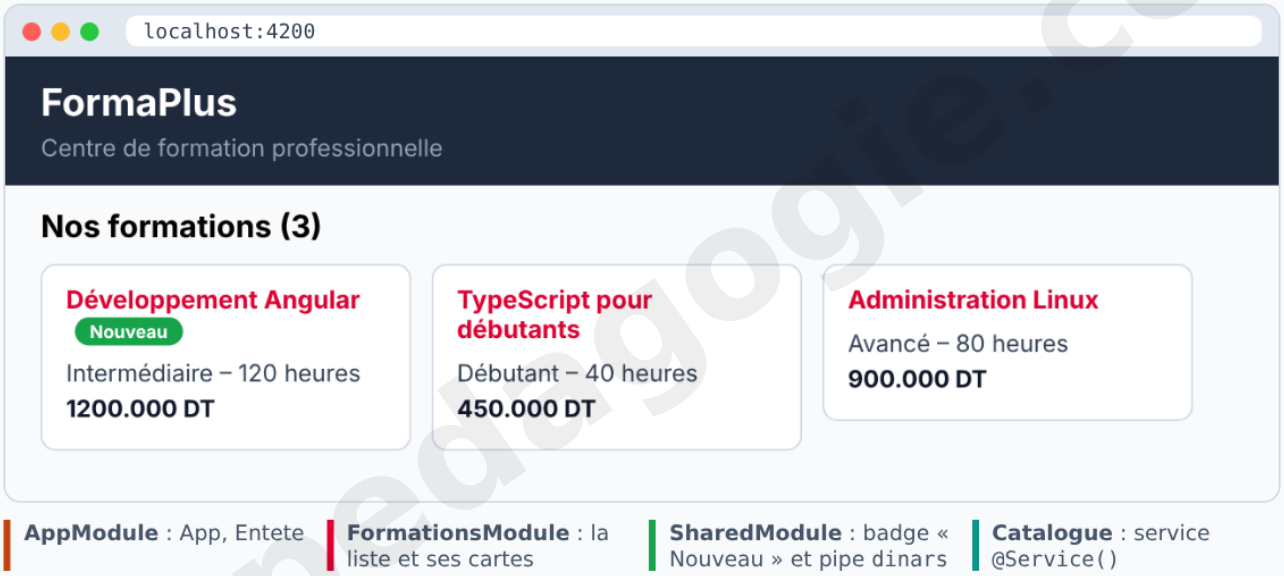
@NgModule({
  declarations: [App, Entete],
  imports: [BrowserModule, AppRoutingModule, FormationsModule],
  providers: [provideBrowserGlobalErrorListeners()],
  bootstrap: [App],
})
export class AppModule {}
```

apcpedagogie.com

```
<!-- src/app/app.html -->
<app-entete />
<main>
  <app-liste-formations />
</main>
<router-outlet />
```

```
ng g c layout/entete
ng serve
```

Résultat : l'application à modules



- **D'où vient `<router-outlet>`?** `AppRoutingModule` importe `RouterModule` et le **réexporte** (`exports: [RouterModule]`). Comme `AppModule` importe `AppRoutingModule`, les templates de ses composants peuvent utiliser `<router-outlet>`.

- **Fournir un service**
  - Le service `Catalogue` contient les données des formations. Avec Angular 22, la CLI le génère avec le décorateur `@Service()`, qui le rend disponible **automatiquement** dans toute l'application, sans l'ajouter à aucun `providers`. Dans les versions précédentes, on écrit `@Injectable({ providedIn: 'root' })`, qui a le même effet.

apcpedagogie.com

```
// src/app/formations/catalogue.ts (Angular 22)
import { Service } from '@angular/core';
import { Formation } from '../models/formation';

@Service()
export class Catalogue {
  private readonly formations: Formation[] = [
    { id: 1, titre: 'Développement Angular', niveau: 'Intermédiaire',
      dureeHeures: 120, prix: 1200, nouveau: true },
    { id: 2, titre: 'TypeScript pour débutants', niveau: 'Débutant',
      dureeHeures: 40, prix: 450, nouveau: false },
    { id: 3, titre: 'Administration Linux', niveau: 'Avancé',
      dureeHeures: 80, prix: 900, nouveau: false },
  ];

  lister(): Formation[] {
    return this.formations;
  }
}
```

```
// Écriture équivalente, avant Angular 22
import { Injectable } from '@angular/core';

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

// Fourni par un module : le service
// n'est pas fourni automatiquement…
@Service({ autoProvided: false })
export class Catalogue { … }

// … il faut donc l'ajouter aux providers
@NgModule({
  …
  providers: [provideBrowserGlobalErrorListeners(), Catalogue],
})
export class AppModule {}
```

- Si un service n'est fourni nulle part, la compilation réussit, mais l'application plante au démarrage. La console du navigateur affiche : `NG0201: No provider found for `_Catalogue`. Source: _AppModule.`

apcpedagogie.com

- Les services seront étudiés en détail dans le chapitre sur les services et l'injection de dépendances.

Le module de routage et le chargement à la demande

- Un **module de routage** contient uniquement la configuration des routes. Le module racine utilise `RouterModule.forRoot(routes)` ; un module de fonctionnalité utilise `RouterModule.forChild(routes)`.
- Un module de fonctionnalité peut être **chargé à la demande** (*lazy loading*) : son code n'est téléchargé que lorsque l'utilisateur visite sa route, ce qui allège le démarrage. La CLI prépare tout avec une seule commande :

```
ng g m admin --route admin --module app-module
```

```
// src/app/app-routing-module.ts (mis à jour par la CLI)
const routes: Routes = [
  {
    path: 'admin',
    loadChildren: () =>
      import('./admin/admin-module').then((m) => m.AdminModule),
  },
];

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

```
// src/app/admin/admin-routing-module.ts
const routes: Routes = [{ path: '', component: Admin }];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class AdminRoutingModule {}
```

apcpedagogie.com

```
# ng build : le module admin est un fichier séparé
Lazy chunk files  | Names        | Raw size
chunk-ERQRZJ7R.js | admin-module | 633 bytes
```

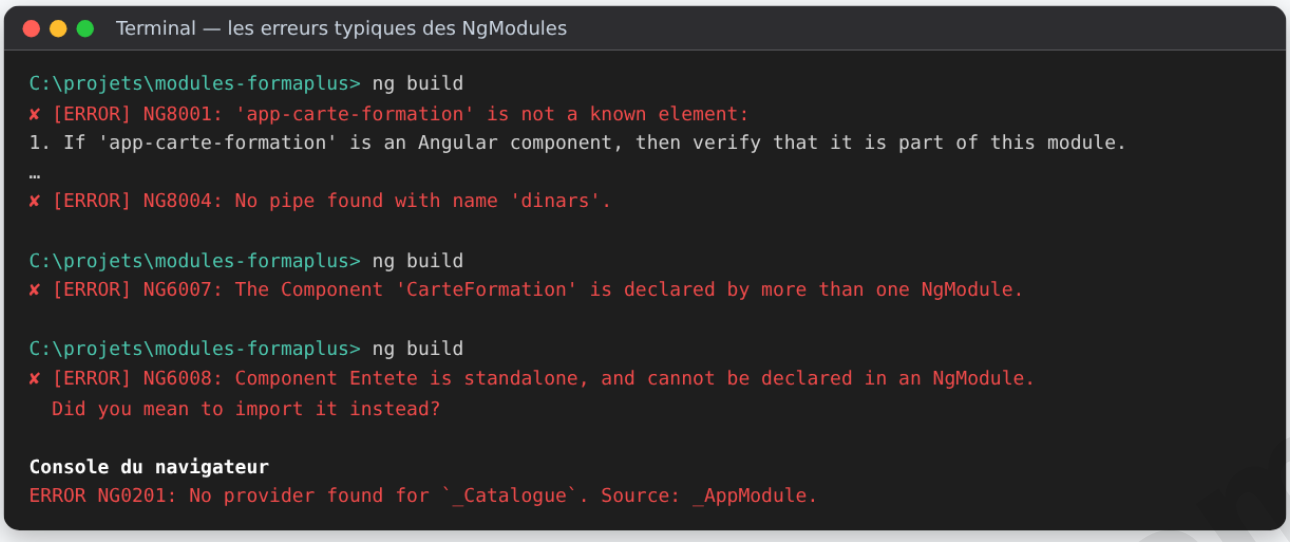
- Le routage sera étudié en détail dans la leçon Routage et navigation avec Angular. En standalone, on charge directement un composant avec `loadComponent: () => import('./admin/admin').then((m) => m.Admin).`

- ## Les types de modules

  - La documentation d'Angular classait les modules en plusieurs catégories selon leur rôle. Ces catégories restent utiles pour lire un projet existant, et elles ont toutes un équivalent standalone :

Type	Rôle	Exemple	Équivalent standalone
Module racine	démarre l'application	`AppModule`	`bootstrapApplication(App, appConfig)`
Module de fonctionnalité (*feature*)	regroupe une partie de l'application	`FormationsModule`, `AdminModule`	un dossier par fonctionnalité, composants autonomes
Module de routage (*routing*)	définit les routes d'un module	`AppRoutingModule`	`app.routes.ts` et `provideRouter(routes)`
Module de service	fournit uniquement des services	l'ancien `HttpClientModule`	fonctions `provide…()` : `provideHttpClient()`
Module de widgets	bibliothèque de composants d'interface	modules d'Angular Material, de ng-bootstrap	import direct de chaque composant
Module partagé (*shared*)	briques réutilisées partout	`SharedModule`	import direct de chaque composant ou pipe

1. ## Erreurs fréquentes



Les quatre premières erreurs bloquent la compilation ; NG0201 apparaît à l'exécution, quand un service n'est fourni nulle part.

Message	Cause	Solution
NG8001: 'app-x' is not a known element … verify that it is part of this module.	Composant non déclaré dans le module, ou déclaré dans un autre module qui ne l'exporte pas, ou module non importé	Déclarer le composant, l'ajouter aux exports de son module, et importer ce module
NG8004: No pipe found with name 'x'.	Même cause pour un pipe (date sans CommonModule, dinars sans SharedModule)	Importer le module qui déclare et exporte le pipe
NG8002: Can't bind to 'x' since it isn't a known property of 'app-y'.	Le composant app-y n'est pas connu du module (souvent affiché avec NG8001)	Comme pour NG8001
NG6007: The Component 'X' is declared by more than one NgModule.	Composant présent dans les declarations de deux modules	Le déclarer dans un seul module, l'exporter, et importer ce module ailleurs
NG6008: Component X is standalone, and cannot be declared in an NgModule.	Composant standalone placé dans declarations	Le mettre dans imports, ou ajouter standalone: false au composant
NG0201: No provider found for `X`. (dans la console)	Service qui n'est fourni nulle part	@Service() ou providedIn: 'root', ou l'ajouter aux providers

1. ## Des modules aux composants standalone

   - Voici la même application écrite en standalone. Chaque élément du module trouve sa place ailleurs :

Dans un NgModule	En standalone
`platformBrowser().bootstrapModule(AppModule)`	`bootstrapApplication(App, appConfig)` dans `main.ts`
`declarations: [...]`	disparaît : chaque composant est autonome (`standalone: false` est supprimé)
`imports: [...]` du module	`imports: [...]` de **chaque composant**, limité à ce que son template utilise
`exports: [...]`	disparaît : tout composant peut être importé là où on en a besoin
`providers: [...]` du module racine	`providers: [...]` de `app.config.ts`
`AppRoutingModule` avec `RouterModule.forRoot`	`app.routes.ts` et `provideRouter(routes)`
`bootstrap: [App]`	premier argument de `bootstrapApplication`

```
// src/main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { App } from './app/app';

bootstrapApplication(App, appConfig)
  .catch((err) => console.error(err));
```

```
// src/app/app.config.ts
import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideBrowserGlobalErrorListeners(), provideRouter(routes)],
};
```

apcpedagogie.com

```
// src/app/app.ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { Entete } from './layout/entete/entete';
import { ListeFormations } from './formations/liste-formations/liste-formations';

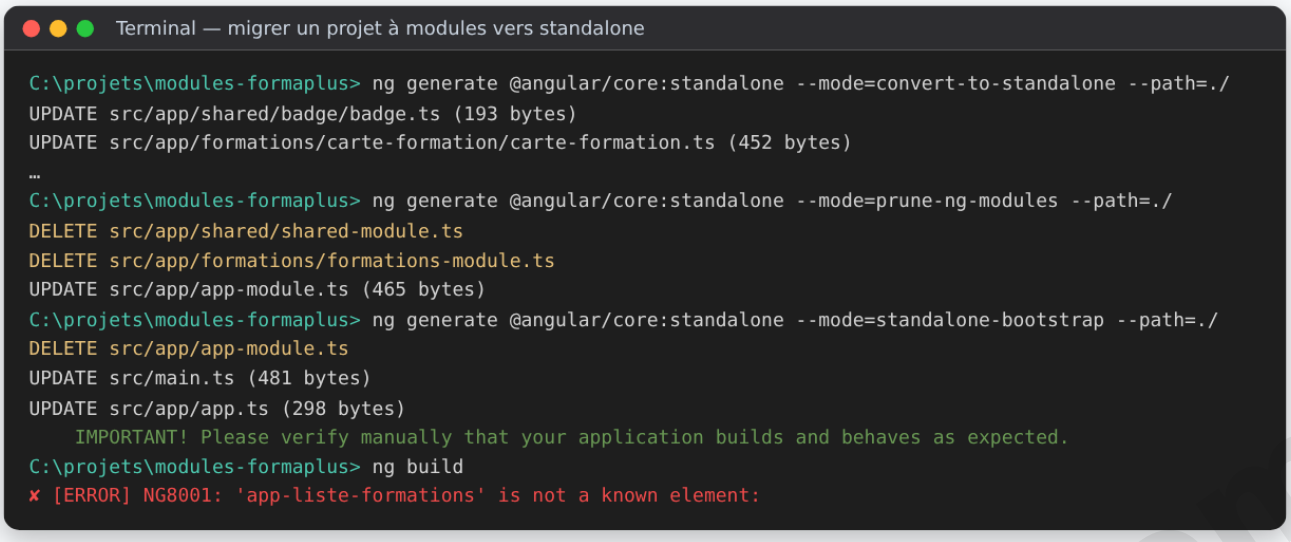
@Component({
  selector: 'app-root',
  styleUrl: './app.css',
  templateUrl: './app.html',
  imports: [Entete, ListeFormations, RouterOutlet],
})
export class App {}
```

```
// src/app/formations/carte-formation/carte-formation.ts
@Component({
  selector: 'app-carte-formation',
  styleUrl: './carte-formation.css',
  templateUrl: './carte-formation.html',
  imports: [Badge, DinarsPipe],
})
export class CarteFormation { … }
```

1. **Migrer automatiquement**
    - Angular fournit une migration automatique, à lancer en **trois étapes**, avec un commit Git entre chaque étape :

```
ng generate @angular/core:standalone --mode=convert-to-standalone --path=./
ng generate @angular/core:standalone --mode=prune-ng-modules --path=./
ng generate @angular/core:standalone --mode=standalone-bootstrap --path=./
ng build
```

apcpedagogie.com



La migration se fait en trois étapes (commit entre chaque). Ici, il a fallu ajouter `ListeFormations` aux `imports` de App à la main : toujours compiler et tester après la migration.

- Sur notre projet, la migration a converti les composants et supprimé les trois modules, mais elle a oublié d'importer `ListeFormations` dans `App` (le composant arrivait jusque-là par l'export de `FormationsModule`). Il a suffi de l'ajouter à la main. Elle laisse aussi un `main.ts` avec `importProvidersFrom(BrowserModule, AppRoutingModule)`, que l'on peut réécrire proprement comme ci-dessus.
- **Les deux mondes se mélangent :** un composant standalone peut être placé dans les `imports` d'un `NgModule`, et un `NgModule` (par exemple `FormsModule`) peut être placé dans les `imports` d'un composant standalone. On peut donc migrer un gros projet petit à petit.

Activité 1 : créer et lire un projet à modules

- Créer le projet `modules-formaplus` avec `--no-standalone`, puis comparer avec votre projet `forma-plus` du TP 1 : citer les fichiers qui n'existent que dans l'un des deux projets, et comparer les deux fichiers `main.ts`.
- Répondre en commentaire dans `app-module.ts`: à quoi servent `BrowserModule`, `AppRoutingModule` et `bootstrap: [App]`?

Activité 2 : le module des formations

- Reproduire `FormationsModule`, `SharedModule`, le service `Catalogue` et le composant `Entete` de la leçon, puis afficher la page.
- Ajouter une quatrième formation (« Réseaux informatiques », Intermédiaire, 60 h, 700 DT, nouvelle) et vérifier qu'elle s'affiche avec son badge.

- **Activité 3 : la visibilité**
  - Provoquer une à une les erreurs suivantes, noter le code et la première ligne du message en commentaire, puis corriger :
    - écrire `<app-carte-formation />` dans `app.html` ;
    - écrire `{{ 1200 | dinars }}` dans `app.html` ;
    - retirer `exports` de `SharedModule` ;
    - déclarer `CarteFormation` à la fois dans `FormationsModule` et dans `AppModule`.
  - Trouver deux façons différentes de pouvoir afficher `{{ 1200 | dinars }}` dans `app.html`, et expliquer laquelle vous préférez.

- **Activité 4 : un nouveau module de fonctionnalité**
  - Générer un module `contact` avec un composant `formulaire-contact` (nom, e-mail, message, bouton) et l'afficher sous la liste des formations. Quels fichiers faut-il modifier ? Quelles propriétés de quels modules ?
  - Utiliser le pipe `dinars` dans ce formulaire (par exemple « Frais d'inscription : 50.000 DT »).

- **Activité 5 : services et chargement à la demande**
  - Remplacer `@Service()` par `@Service({ autoProvided: false })` dans `Catalogue`, lancer l'application et relever l'erreur affichée dans la console (F12). La corriger avec les `providers` d'`AppModule`.
  - Générer un module `admin` chargé à la demande (`ng g m admin --route admin --module app-module`). Lancer `ng build` et repérer le fichier *Lazy chunk*. Ouvrir `http://localhost:4200/admin`, puis, dans l'onglet *Réseau* des outils de développement, observer le moment où ce fichier est téléchargé.

- **Activité 6 : migrer vers standalone**
  - Initialiser Git dans `modules-formaplus` et faire un commit. Lancer les trois étapes de la migration automatique, avec un commit après chacune.
  - Compiler, corriger les erreurs éventuelles, puis réécrire `main.ts` avec `app.config.ts` et `app.routes.ts`, comme dans la leçon.
  - Avec `git diff` ou VS Code, résumer en cinq lignes ce que la migration a changé. Publier le dépôt sur GitHub.

apcpedagogie.com

- ## À retenir

  - Un **NgModule** est une classe décorée avec `@NgModule` qui regroupe des composants, directives et pipes, et contrôle ce qui est visible.
  - `declarations`: ce qui appartient au module ; `imports`: les modules dont il a besoin ; `exports`: ce qu'il partage ; `providers`: les services ; `bootstrap`: le composant racine.
  - Tout ce qui est déclaré est **privé** sauf s'il est exporté, et un module ne transmet pas ses imports.
  - `BrowserModule` dans le module racine seulement, `CommonModule` dans les autres.
  - Modules de fonctionnalité, partagé, de routage : une organisation par rôle, avec chargement possible à la demande (`loadChildren`).
  - Depuis Angular 19, les composants sont **standalone** par défaut : chaque composant importe ce qu'il utilise, et `app.config.ts` remplace les `providers` du module racine.
  - La migration `ng generate @angular/core:standalone` transforme un projet à modules, en trois étapes à vérifier.

> *Les modules ont structuré Angular pendant des années ; les composants standalone les rendent facultatifs. Comprendre les deux, c'est pouvoir travailler aussi bien sur un projet neuf que sur une application existante.*

Sommaire du cours Angular

Source : https://apcpedagogie.com/introduction-au-module-angular · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com

apcpedagogie.com