Présentation Architecture Angular

Projet réalisé par Valentin CIRCOSTA

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

EFREI
ANGULAR

Table des matières

Point d'entrée

src/main.ts : Bootstrap de l'application

Ce fichier est le point d'entrée de l'application Angular.
Il initialise le projet, configure les services principaux et démarre le composant racine.

Rôles principaux :

- Démarrage : bootstrapApplication(AppComponent) lance l'application avec l'approche standalone (plus moderne qu'AppModule).
- Routing : provideRouter(routes) enregistre les routes définies dans app.routes.ts.
- HTTP : provideHttpClient active HttpClient et applique deux intercepteurs :
    - authInterceptor → ajoute automatiquement le JWT dans les requêtes.
    - errorInterceptor → gère les erreurs globalement (ex. mauvais mot de passe, accès interdit).
- Composant racine : AppComponent contient le <router-outlet> pour afficher les pages.
- Gestion des erreurs : .catch(err => console.error(err)) log les erreurs critiques au lancement.
- Standalone Components
    - Depuis Angular 14+, plus besoin de NgModule.
    - Un composant peut être *standalone* (standalone: true), et importer directement ses dépendances (imports: [CommonModule, RouterModule]).
    - Avantage : plus simple, plus clair, meilleure modularité.
    - Dans ton projet : bootstrapApplication(AppComponent) remplace AppModule.

En résumé : main.ts est le centre de configuration qui lance l'app, active le routing, sécurise les appels HTTP et gère les erreurs globales.

src/app/app.config.ts : Configuration globale

Rôles principaux :

- Zone.js : provideZoneChangeDetection({ eventCoalescing: true }) → optimise la détection des changements en regroupant les événements pour de meilleures performances.
- Routing : provideRouter(routes) → enregistre toutes les routes définies dans app.routes.ts.
- HTTP Client : provideHttpClient(...) → configure les appels HTTP avec deux intercepteurs globaux :
    - authInterceptor : ajoute automatiquement le JWT aux requêtes.
    - errorInterceptor : centralise la gestion des erreurs (ex. mauvais mot de passe, accès interdit).

En résumé : ce fichier définit la configuration principale de l'app Angular (perf, routes, HTTP + sécurité/erreurs).

src/app/app.component.ts : Composant racine

Rôle principal : c'est le point d'entrée visuel de l'app, celui qui contient le router-outlet pour afficher les différentes pages selon les routes.

Imports :

- RouterOutlet & RouterModule → gestion de la navigation.
- CommonModule & HttpClientModule → fonctionnalités Angular de base et appels HTTP.

Services injectés :

- AuthService → gestion de l'authentification.
- Router → permet de rediriger l'utilisateur.

Fonction logout() :

- Déconnecte l'utilisateur via auth.logout().
- Redirige automatiquement vers la page de connexion /auth/login.

En résumé : AppComponent est le composant racine qui gère l'affichage des pages et permet la déconnexion avec redirection vers le login.

Routing et Navigation

src/app/app.routes.ts : Routes principales

- auth → charge les routes liées à l'authentification (/auth/...).
- reservations → accessible uniquement si l'utilisateur est connecté (authGuard).
- admin → réservé aux administrateurs (adminGuard).
- '' → redirige automatiquement vers la page de login (/auth/login).

En résumé : ce fichier définit la navigation principale de l'application, avec une sécurité basée sur des guards pour protéger certaines pages (auth et admin).

src/app/features/*/routes.ts : Routes par feature

Auth (AUTH_ROUTES)

- Routes : /login, /register
- Composants associés : LoginComponent, RegisterComponent
- Rôle : gérer l'accès à l'application (connexion / inscription).

Reservations (RESERVATIONS_ROUTES)

- Routes : /reservations, /reservations/new, /reservations/:id
- Composants associés : ReservationListComponent, ReservationFormComponent, ReservationDetailComponent
- Rôle : gérer le cœur fonctionnel côté utilisateur (consulter, créer, voir les détails d'une réservation).

Admin (ADMIN_ROUTES)

- Routes : /admin, /admin/users, /admin/computers
- Composants associés : AdminDashboardComponent, UserManagementComponent, ComputerManagementComponent
- Rôle : offrir une interface de gestion réservée aux administrateurs (utilisateurs et ordinateurs).

La gestion des routes dans ce projet suit une logique de modularité par feature, ce qui permet de bien séparer les responsabilités :

- Auth assure l'entrée et l'accès sécurisé à l'application,
- Reservations gère les fonctionnalités principales offertes aux utilisateurs,
- Admin fournit des outils de supervision et de gestion réservés aux administrateurs.

Cette organisation rend l'application plus claire, plus maintenable et plus évolutive, car chaque module est indépendant et peut être développé, testé ou modifié sans impacter directement les autres.

Lazy Loading : Comment ça fonctionne

Le Lazy Loading consiste à ne charger les composants ou les modules que lorsqu'ils sont nécessaires, c'est-à-dire quand l'utilisateur navigue vers leur route.

- Réduit le bundle initial → application plus rapide à démarrer.
- Améliore la performance et la réactivité de l'application.

```ts
// app.routes.ts
import { Routes } from '@angular/router';
import { authGuard } from './core/guards/auth.guard/auth.guard';
import { adminGuard } from './core/guards/admin.guard/admin.guard';

export const routes: Routes = [
  {
    path: 'auth',
    loadChildren: () => import('./features/auth/auth.routes').then(m => m.AUTH_ROUTES)
  },
  {
    path: 'reservations',
    canActivate: [authGuard],
    loadChildren: () => import('./features/reservations/reservations.routes').then(m => m.RESERVATIONS_ROUTES)
  },
  {
    path: 'admin',
    canActivate: [adminGuard],
    loadChildren: () => import('./features/admin/admin.routes/admin.routes').then(m => m.ADMIN_ROUTES)
  },
  { path: '', redirectTo: '/auth/login', pathMatch: 'full' }
];
```

- Page d'accueil
    - Angular charge uniquement le code nécessaire pour la route par défaut (/auth/login).
    - Les modules reservations et admin ne sont pas inclus dans le bundle initial.
- Navigation vers une feature
    - Quand l'utilisateur clique sur /reservations :
        - Angular importe dynamiquement RESERVATIONS_ROUTES.
        - Seuls les composants de reservations sont chargés.
    - Pareil pour /admin avec ADMIN_ROUTES.
- Guards
    - authGuard et adminGuard vérifient si l'utilisateur a le droit d'accéder à la feature avant de la charger.
    - Si le guard bloque l'accès, Angular ne charge même pas le module.

Architecture DDD

src/app/features/ : Logique métier par domaine

- auth/ → Contient tout ce qui concerne l'authentification des utilisateurs :
    - login.component : formulaire et logique de connexion.
    - register.component : formulaire et logique d'inscription.
    - auth.routes.ts : routes spécifiques à l'authentification (/auth/login, /auth/register).
- reservations/ → Gère les opérations liées aux réservations :
    - reservation-list : affichage de toutes les réservations.
    - reservation-form : création ou modification d'une réservation.
    - reservation-detail : affichage détaillé d'une réservation.
    - reservations.routes.ts : routes spécifiques (/reservations, /reservations/new, /reservations/:id).
- admin/ → Contient les fonctionnalités d'administration, accessibles aux utilisateurs avec le rôle admin :
    - admin-dashboard.component : tableau de bord global.
    - user-management.component : gestion des utilisateurs.
    - computer-management.component : gestion des ordinateurs.
    - admin.routes.ts : routes spécifiques à l'administration (/admin, /admin/users, /admin/computers).

Chaque domaine encapsule ses composants et ses routes, ce qui rend l'application modulaire, maintenable et adaptée au lazy loading.

src/app/shared/ : Composants réutilisables

components/ → Composants réutilisables dans toute l'application

- button.component → Boutons personnalisables (label, type, désactivation).
- loading-spinner.component → Affiche un indicateur de chargement.
- modal.component → Fenêtre modale réutilisable pour alertes, formulaires ou confirmations.
- navbar.component → Barre de navigation commune à l'application.

Ces composants sont standalone et peuvent être importés dans différents modules ou domaines.

directives/ → Comportements réutilisables sur les éléments HTML

- highlight.directive → Ajoute un effet visuel sur un élément (ex. survol ou focus).
- role-based.directive → Affiche ou masque des éléments selon le rôle de l'utilisateur.

pipes/ → Transformations réutilisables des données

- date-format.pipe → Formate les dates pour l'affichage.
- role-label.pipe → Transforme un rôle technique (admin, user) en libellé lisible.

services/ → Logique métier réutilisable et globale

- error.services.ts → Gestion centralisée des notifications d'erreurs, warnings et informations.
- D'autres services : API, authentification, notifications, gestion des réservations ou utilisateurs.

Résumé : shared/ contient tout ce qui peut être utilisé dans plusieurs domaines de l'application : composants UI, directives, pipes et services. Cela favorise la réutilisabilité et la cohérence dans toute l'app.

src/app/core/ : Services globaux

guards/ → Protection des routes

- auth.guard → Vérifie si l'utilisateur est connecté avant d'accéder à certaines routes.
- admin.guard → Vérifie si l'utilisateur a le rôle admin pour accéder aux routes administratives.

interceptors/ → Intercepteurs HTTP globaux

- auth.interceptor → Ajoute automatiquement le token d'authentification aux requêtes HTTP.
- error.interceptor → Intercepte les erreurs HTTP et les transmet au service de notification.

models/ → Définition des types de données

- user.model → Structure d'un utilisateur (id, email, rôle…).
- reservation.model → Structure d'une réservation (id, date, utilisateur…).
- randomuser.model → Modèle pour les utilisateurs générés aléatoirement (si utilisé).

services/ → Services globaux

- api.service → Service générique pour les appels HTTP vers l'API.
- auth.service → Gestion de l'authentification (login, logout, récupération utilisateur courant).
- notification.service → Gestion des notifications et erreurs globales.
- random-user.service → Gestion des utilisateurs aléatoires (utile pour tests ou données factices).
- reservations.service → Gestion des réservations (CRUD, récupération liste ou détails).

Résumé : core/ contient tout ce qui est central et global pour l'application : sécurité (guards), communication avec l'API (services et interceptors), et définition des modèles. Ces éléments ne dépendent pas d'un domaine métier spécifique, contrairement à features/.

Configuration

angular.json : Configuration du projet

Architect / Targets

Le projet définit plusieurs cibles de build et développement.

a) build

- **Builder** : @angular/build:application
- **Entrée principale** : src/main.ts
- **Polyfills** : zone.js
- **Fichiers de style** : src/styles.scss
- **Assets** : tout ce qui est dans le dossier public
- **Configurations** :
    - **production** :
        - Optimisation et hash des fichiers pour le cache
        - Budgets pour taille initiale (< 1 MB) et styles (< 8 KB)
    - **development** :
        - Pas d'optimisation
        - Source maps activés
        - Pas d'extraction de licences

b) serve

- Utilise le builder @angular/build:dev-server
- Se base sur le build target angular-projetFinal:build pour démarrer le serveur
- Par défaut, la configuration development est utilisée

c) test

- Builder : @angular/build:karma
- Polyfills : zone.js + zone.js/testing
- Style : scss

- Assets et styles identiques au build principal

d) lint

- Builder : @angular-eslint/builder:lint
- Pattern pour linting : tous les .ts et .html du dossier src

angular.json centralise toute la configuration Angular :

- Structure du projet
- Cibles de build, dev server, tests et lint
- Configuration production vs développement
- Gestion des styles, assets et polyfills

C'est le cœur de la configuration du projet, qui permet de construire, servir, tester et maintenir l'application Angular de manière cohérente.

package.json : Dépendances et scripts

Scripts principaux

- ng : raccourci pour la CLI Angular (ng)
- start : lance le serveur de dev (ng serve)
- build : construit le projet pour production (ng build)
- watch : build en continu pour dev (ng build --watch --configuration development)
- test : lance les tests unitaires avec Karma (ng test)
- test:ci : tests unitaires en mode CI, Chrome headless
- lint : vérifie la qualité du code avec ESLint (ng lint)
- prepare : hook pour Husky (git hooks)

Dépendances (dependencies)

- Angular core : @angular/core, @angular/common, @angular/forms, @angular/router, @angular/platform-browser
- RxJS : pour la programmation réactive (rxjs)
- Zone.js : gestion des zones pour Angular (zone.js)
- tslib : helpers TypeScript pour le runtime (tslib)

Ce sont les packages nécessaires pour que ton application fonctionne en production.

Dépendances de développement (devDependencies)

- Angular build & CLI : @angular/cli, @angular/compiler, @angular/compiler-cli, @angular/build, @angular/platform-browser-dynamic
- Tests : karma, karma-chrome-launcher, karma-jasmine, jasmine-core, jest-environment-jsdom
- Lint / formatage : eslint, angular-eslint, @typescript-eslint/*, lint-staged, husky, prettier
- CSS / Tailwind : tailwindcss, postcss, autoprefixer
- TypeScript : typescript, typescript-eslint

Ces packages ne sont nécessaires que pour le développement, tests, linting et build du projet.

En résumé :

- Scripts : gestion dev, build, test, lint, CI
- Dependencies : tout ce dont Angular a besoin pour tourner
- DevDependencies : outils de dev, tests, lint et styling

tsconfig.json : Configuration TypeScript

Héritage

- "extends": "./tsconfig.json" → hérite des réglages globaux TypeScript définis dans tsconfig.json.

Cela évite de répéter toute la configuration TS spécifique à Angular.

Compiler Options

- "outDir": "./out-tsc/app" → dossier de sortie pour les fichiers compilés TypeScript.
- "types": [] → pas de types supplémentaires globalement inclus (ex. pas de Node.js ou Jasmine par défaut).

Fichiers inclus

- "include": ["src/**/*.ts"] → compile tous les fichiers .ts de src/.

Fichiers exclus

- "exclude": ["src/**/*.spec.ts"] → ignore les fichiers de test (.spec.ts) pour la compilation de l'application.

En résumé :
Ce fichier configure la compilation TypeScript pour l'application Angular, en utilisant les réglages globaux, en excluant les tests et en définissant où sortir le code compilé.

Styling et Assets

src/styles.scss : Styles globaux + Tailwind

Styles globaux

- C'est le fichier principal de styles pour ton application Angular.
- Tout ce que tu mets ici s'applique globalement (à toutes les pages et composants).

Intégration TailwindCSS

- @tailwind base; → importe les styles de base de Tailwind (reset CSS, normalisation des éléments HTML).
- @tailwind components; → importe les composants Tailwind prédéfinis (boutons, formulaires, etc.).
- @tailwind utilities; → importe toutes les classes utilitaires Tailwind (marges, couleurs, flex, grid…).

En résumé :
styles.scss sert de point d'entrée global pour les styles et charge Tailwind pour pouvoir utiliser ses classes utilitaires et composants dans toute l'application.

tailwind.config.js : Configuration Tailwind

content: ["./src/**/*.{html,ts}"]

- Indique à Tailwind quels fichiers scanner pour détecter les classes CSS utilisées.
- Ici, toutes les templates HTML et les fichiers TypeScript dans src/ seront analysés.
- Permet à Tailwind de purger les classes inutilisées lors de la production.

theme: { extend: {} }

- Sert à personnaliser le thème : couleurs, polices, espacements, etc.
- extend: {} signifie qu'aucune personnalisation n'a encore été ajoutée, Tailwind garde ses valeurs par défaut.

plugins: []

- Permet d'ajouter des extensions Tailwind ou des plugins tiers pour ajouter des utilitaires supplémentaires.

tailwind.config.js configure Tailwind pour ton projet, en précisant où chercher les classes, et en laissant la possibilité d'étendre le thème ou ajouter des plugins si nécessaire.

src/assets/ : Ressources statiques

- Contient tous les fichiers statiques de ton projet : images, icônes, polices, JSON, fichiers de configuration, etc.
- Ces fichiers sont copiés tels quels dans le build final et accessibles via une URL relative dans ton application.
- Permet de séparer le contenu statique du code TypeScript ou des composants.

En résumé : C'est un dossier pour tout ce qui est statique et réutilisable dans l'application.