RAPPORT : ARCHITECTURE FULLSTACK, FONCTIONNALITES AVANCEES ET CONTENEURISATION D'UNE APPLICATION TODO

Réalisé : par Hachem Squalli El Houssaini
Dirigé : Pr. Y. IDRISSI KHAMLICHI

Introduction :

Dans le cadre du cursus Développement Web Fullstack avec Spring Boot et Angular , ce projet vise à concevoir, développer et déployer une application de gestion de tâches ("ToDo App") robuste et moderne. L'objectif n'est pas seulement de créer une liste de tâches, mais d'appliquer les principes d'ingénierie logicielle modernes : une architecture découplée, une gestion d'état réactive côté client, et une chaîne de déploiement conteneurisée.

Ce rapport détaille les choix techniques effectués sur le backend (Spring Boot), le frontend (Angular 17+), l'ajout d'une fonctionnalité de gamification ("Mode Zen") pour améliorer la productivité utilisateur, et enfin, l'infrastructure de déploiement via Docker.

Architecture Backend: Spring Boot & PostgreSQL

Le backend repose sur l'écosystème Spring Boot 3, privilégiant la convention sur la configuration pour accélérer le développement tout en maintenant une structure rigoureuse.

Structure en Couches

Couche	Composant	Rôle Technique
Présentation	TodoController	Expose les endpoints REST (/api/todos). Gère la sérialisation JSON et les codes HTTP.
Service	TodoService	Contient la logique métier. Fait le pont entre le contrôleur et la base de données.
Accès aux Données	TodoRepository	Interface étendant JpaRepository. Abstrait les requêtes SQL complexes via Hibernate.
Modèle	Todo (Entity)	Représentation objet de la table todos dans la base de données PostgreSQL.

Gestion des Entités et Persistance

L'entité `Todo` utilise les annotations JPA standard (`@Entity`, `@Id`, `@GeneratedValue`) pour le mapping ORM. L'utilisation de la bibliothèque **Lombok** (`@Data`, `@NoArgsConstructor`) permet de réduire considérablement le code "boilerplate" (getters, setters, constructeurs), rendant le code plus lisible et maintenable.

La validation des données est assurée par l'API Jakarta Validation (`@NotBlank`), garantissant qu'aucune tâche sans titre ne puisse être persistée en base, renforçant ainsi l'intégrité des données avant même qu'elles n'atteignent le SGBD.

Contrôleurs et Gestion des Exceptions

Le `TodoController` expose une API RESTful complète (CRUD). Un point d'attention particulier a été porté à la gestion des erreurs via le `GlobalExceptionHandler`. Au lieu de laisser les exceptions Java "fuiter" vers le client (ce qui serait une faille de sécurité potentielle en exposant la stack trace), nous utilisons `@RestControllerAdvice` pour intercepter les exceptions et renvoyer un objet `ProblemDetail` . Cela permet au frontend de recevoir des erreurs structurées et prévisibles.

Injection de Dépendances et Inversion de Contrôle (IoC)

Le cœur de l'architecture backend repose sur le principe d'Inversion de Contrôle (IoC) géré par le conteneur Spring. Plutôt que d'instancier manuellement les objets (via `new TodoService()`), nous déléguons cette responsabilité au framework.

Dans notre code, nous utilisons l'**Injection par Constructeur**, considérée comme la meilleure pratique actuelle (par rapport à l'injection par champ `@Autowired`). L'annotation Lombok `@RequiredArgsConstructor` joue un rôle clé ici : elle génère automatiquement un constructeur pour tous les champs déclarés final.

Flux d'injection :

1. Spring détecte TodoRepository (interface étendant JpaRepository).
2. Il injecte ce repository dans le constructeur de TodoService.
3. Il injecte ensuite l'instance de TodoService dans TodoController.

Cela garantit que nos composants sont immutables et testables unitairement (on peut facilement passer un "mock" de service dans le constructeur du contrôleur lors des tests).

2.5 Sécurité et Configuration CORS

L'architecture étant découplée (Frontend sur le port 80/4200 et Backend sur le port 8080), nous sommes confrontés à la politique de sécurité des navigateurs : le **Same-Origin Policy**. Par défaut, le navigateur bloque les requêtes HTTP provenant d'un domaine différent.

Pour résoudre cela, le contrôleur utilise l'annotation : `@CrossOrigin(origins = "http://localhost:4200", allowCredentials = "true")`

Cette configuration injecte les en-têtes HTTP nécessaires (`Access-Control-Allow-Origin`) dans les réponses du serveur, autorisant explicitement notre application Angular à consommer l'API, tout en rejetant les requêtes provenant d'autres sources non autorisées.

Architecture Frontend : Angular 17+ & Signals

Le frontend marque une rupture avec les anciennes versions d'Angular en adoptant les dernières innovations du framework : les **Standalone Components** et les **Signals**.

Modernité du Framework : Standalone Components

Contrairement aux architectures classiques basées sur les `NgModule`, cette application utilise des composants autonomes (`standalone: true`). Cela simplifie considérablement l'arbre de dépendances : chaque composant (`App`, `Header`, `ZenFocusModal`) importe directement ce dont il a besoin (`CommonModule`, `MatIconModule`). Cela réduit la taille du bundle final (Tree Shaking plus efficace) et facilite les tests unitaires.

Gestion d'État Réactive avec Signals

L'innovation majeure de ce projet réside dans l'abandon partiel de Zone.js au profit des **Signals** pour la gestion de l'état local.

Caractéristique	Approche Classique (RxJS / Zone.js)	Approche Moderne (Signals)	Avantage dans le projet
Déclenchement	Détection de changement globale	Mise à jour granulaire (fine-grained)	Performances accrues
Syntaxe	{{ variable }}	{{ variable() }}	Distinction claire lecture/écriture
Dépendances	Besoin de subscribe / async pipe	Réactivité automatique via computed	Code plus lisible et moins de fuites mémoire

Dans `app.ts`, la liste des tâches est gérée par `todos = signal<Todo[]>([])`. Lorsqu'une tâche est ajoutée ou modifiée, le signal notifie la vue instantanément sans avoir besoin de parcourir tout l'arbre des composants. De plus, l'utilisation de `computed` pour la variable `hasPending` permet de dériver l'état (savoir si des tâches sont en attente) sans recalcul manuel à chaque cycle.

Communication Client-Serveur et Typage Strict

L'une des forces de cette architecture est l'utilisation rigoureuse de TypeScript pour garantir la cohérence des données entre le back et le front. L'interface Todo.model.ts agit comme un contrat :

```
// TypeScript
2. export interface Todo {
3.     id?: number;
4.     title: string;
5.     completed: boolean;
6. }
7.
```

Le service `TodoService` utilise le module `HttpClient` d'Angular pour effectuer les appels REST. Contrairement aux Promesses (Promises) classiques de JavaScript, Angular utilise des **Observables** (via la bibliothèque RxJS). Cela permet de traiter les flux de données de manière asynchrone et de réagir aux événements (succès, erreur) de façon fluide dans les composants via la méthode. `subscribe`().

Nouvelle Syntaxe de Contrôle de Flux (Control Flow)

Le projet exploite la toute nouvelle syntaxe de template introduite dans Angular 17 (`@if`, `@for`), qui remplace les directives structurelles historiques `(*ngIf, *ngFor)`. Cette évolution n'est pas seulement esthétique, elle améliore les performances de rendu du DOM.

Fonctionnalité Phare : Le "Mode Zen" (Gamification)

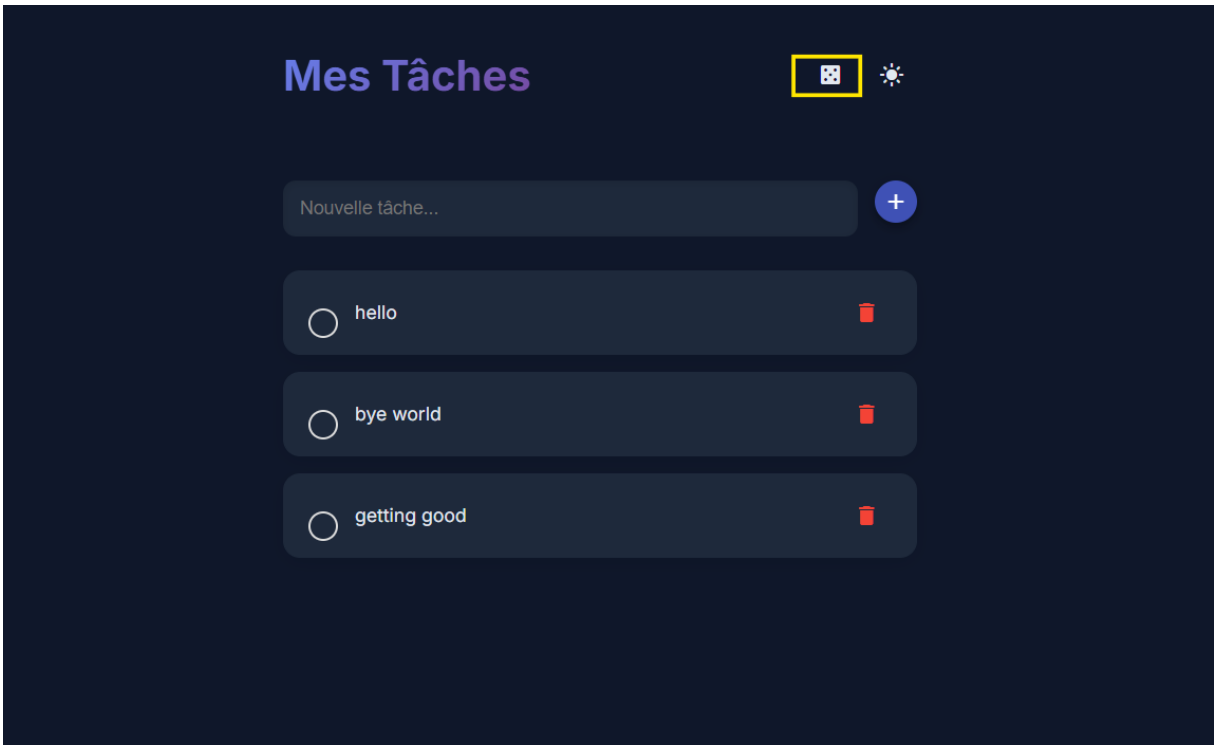
En tant qu'ingénieur sensible à l'expérience utilisateur (UX) et aux mécaniques de jeu, j'ai implémenté une fonctionnalité inédite : le **Mode Zen**.

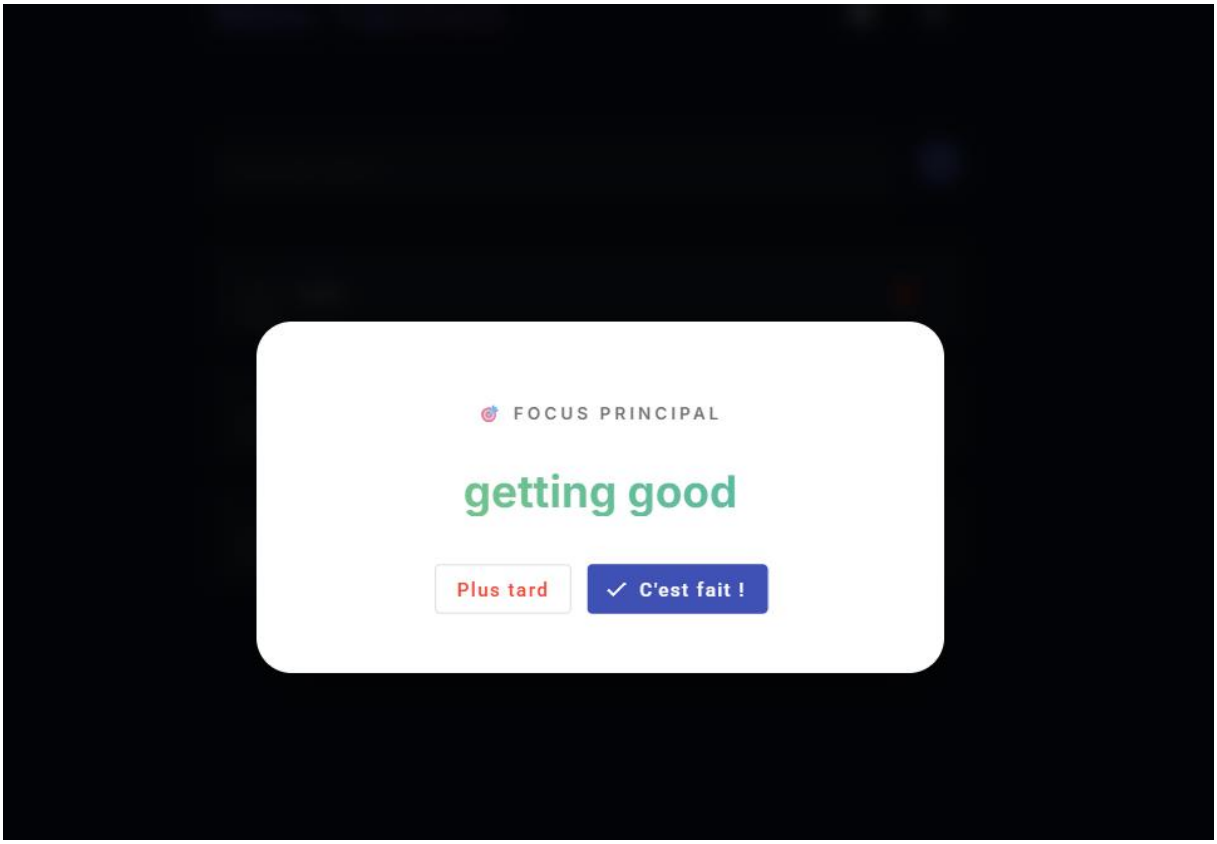
Concept et Objectif

L'utilisateur face à une longue liste de tâches peut souffrir de "paralysie décisionnelle". Le Mode Zen introduit une mécanique de hasard (RNG - Random Number Generation) inspirée des jeux vidéo (type "Gacha" ou coffre au trésor). L'objectif est de forcer la concentration sur **une seule tâche** choisie aléatoirement par le système, éliminant la charge mentale du choix.

Implémentation Technique

1. **Logique (App.ts) :** La méthode `spinTheWheel`() filtre d'abord les tâches non complétées. Elle utilise `Math.random`() pour sélectionner un index, puis injecte cette tâche dans le signal `focusedTodo`.
2. **Composant Modal (ZenFocusModal) :**
    - Ce composant est isolé et reçoit la tâche via `@Input`.
    - Il utilise un overlay sombre avec un effet de flou (`backdrop-filter: blur)` pour masquer visuellement le reste de la liste, renforçant psychologiquement l'idée de "Focus".
    - Il communique avec le parent via `@Output` pour signaler soit la fermeture ("Plus tard"), soit la complétion de la tâche ("C'est fait !").
3. **Design System :** L'utilisation de Material Design et d'animations CSS (`popIn`, `fadeIn`) rend l'interaction fluide et gratifiante.


Mes Tâches
Nouvelle tâche...
hello
bye world
getting good


FOCUS PRINCIPAL
getting good
Plus tard
C'est fait !

Stratégie DevOps : Conteneurisation

J’ai conçu l'infrastructure de déploiement en mettant l'accent sur la sécurité, la légèreté et la séparation des environnements.

Approche Multi-Stage Build

Pour les Dockerfiles (Frontend et Backend), j'ai appliqué le modèle **Multi-Stage Build**. Cette technique est cruciale pour la sécurité et la performance.

Backend (backend/Dockerfile) :

- **Stage 1 (Build) :** Image `maven:3.9.6-eclipse-temurin-17`. Contient tous les outils de compilation (Maven, JDK complet). Le code est compilé ici.
- **Stage 2 (Runtime) :** `Image eclipse-temurin:17-jre-alpine`. Ne contient que le JRE (Java Runtime Environment) nécessaire à l'exécution, sur une base Alpine Linux ultra-légère (environ 5MB pour l'OS).
- **Bénéfice :** L'image finale ne contient ni le code source, ni Maven, réduisant la surface d'attaque potentielle.

Frontend (frontend/Dockerfile) :

- **Stage 1 (Build) :** Image `node:22-alpine`. Installe les dépendances et compile l'Angular en fichiers statiques (HTML/CSS/JS).
- **Stage 2 (Serve) :** Image `nginx:alpine`. Sert uniquement les fichiers statiques générés.

5.2 Sécurité et Optimisation des Images

Dans le Dockerfile du backend, une mesure de sécurité explicite a été ajoutée :

```
// Dockerfile
2. RUN addgroup -S spring && adduser -S spring -G spring
3. USER spring:spring
4.
```

Par défaut, les conteneurs s'exécutent en `root`. En créant un utilisateur spécifique (`spring`) et en basculant dessus, nous limitons les dégâts potentiels si un attaquant parvenait à exécuter du code arbitraire via une faille de l'application (RCE). C'est un principe fondamental de "moindre privilège".

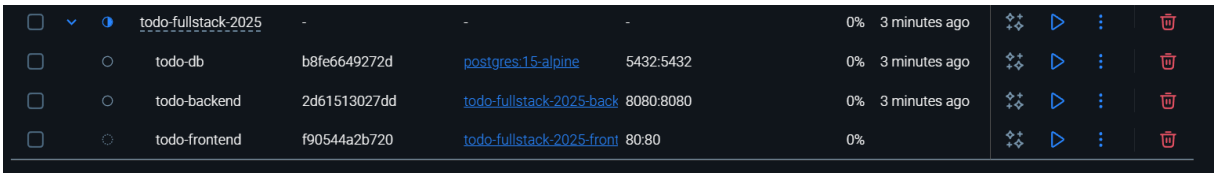
Pour le frontend, une configuration Nginx personnalisée (`nginx.conf`) a été mise en place pour gérer le routage SPA (Single Page Application). Elle redirige toutes les routes inconnues vers `index.html`, permettant à Angular de gérer le routing interne sans erreur 404 du serveur.

Orchestration avec Docker Compose

Le fichier `docker-compose.yml` orchestre trois services :

1. **Postgres :** Base de données isolée dans un réseau privé.
2. **Backend :** Connecté à la base via des variables d'environnement (`SPRING_DATASOURCE_URL`) qui surchargent la configuration locale.
3. **Frontend :** Expose le port 80.

Le réseau **`app-network`** de type **`bridge`** assure que les conteneurs peuvent communiquer entre eux par leurs noms de service (**`postgres`**, **`backend`**) sans exposer la base de données directement sur internet.



Conclusion

Ce projet démontre la maîtrise d'une stack technologique complète et moderne. Le backend assure une gestion des données robuste et sécurisée avec Spring Boot. Le frontend exploite la puissance des Signals d'Angular pour offrir une interface réactive et performante. L'ajout du "Mode Zen" prouve la capacité à traduire un besoin utilisateur (productivité) en fonctionnalité technique innovante. Enfin, l'architecture Docker mise en place garantit un déploiement reproductible, optimisé et sécurisé, répondant aux standards industriels actuels.