ORION

MDD

Choix techniques et Architecture MVP

Diffusion : OpenClassrooms – Mentor : Jeoffrey Hernandez - Evaluateur : Jordan Charpentier

Version	***Date***	***Auteur***	***Motif***
1	20/5/2025	Christophe Pierrès	Création du document
2			

Contenu **Page**

1. Introduction 3

2. Diagrammes de use-cases des besoins du MVP 5

2.1. Utilisateur non authentifié 5
2.2. Utilisateur authentifié 5
2.2.1. Proposition d'amélioration : notifications en temps réel (SSE) 5

3. Choix techniques 6

3.1. Comparaison modèle Servlet/bloquant versus Réactive 6
3.1.1. Architecture bloquante (JEE ou Spring MVC avec JPA et JDBC) 6
3.1.2. Architecture réactive - Spring WebFlux avec R2DBC 6
3.1.3. Architecture générale 7
3.1.3.1. Disponibilité 7
Si besoin : scalabilité horizontale possible avec Kubernetes 8
Schéma conceptuel : disponibilité avec Kubernetes 8
3.2. Backend 9
3.2.1. Service REST : Spring Webflux 9
3.2.2. Persistance : base Postgresql v17 avec driver R2DBC 9
3.3. Frontend : Angular v19 9
3.3.1. Librairie de composants visuels 10
3.4. Sécurité 11
3.5. Testing 13
3.5.1. Front-end testing 13
3.5.2. Back-end testing 14

4. Dossier d'architecture 15

4.1. Architecture globale 15
4.2. Architecture Reactive du Backend 15
4.3. Architecture du Frontend 16
4.4. Modèle de Données 16
4.5. Diagramme de Déploiement (Docker-compose) 17

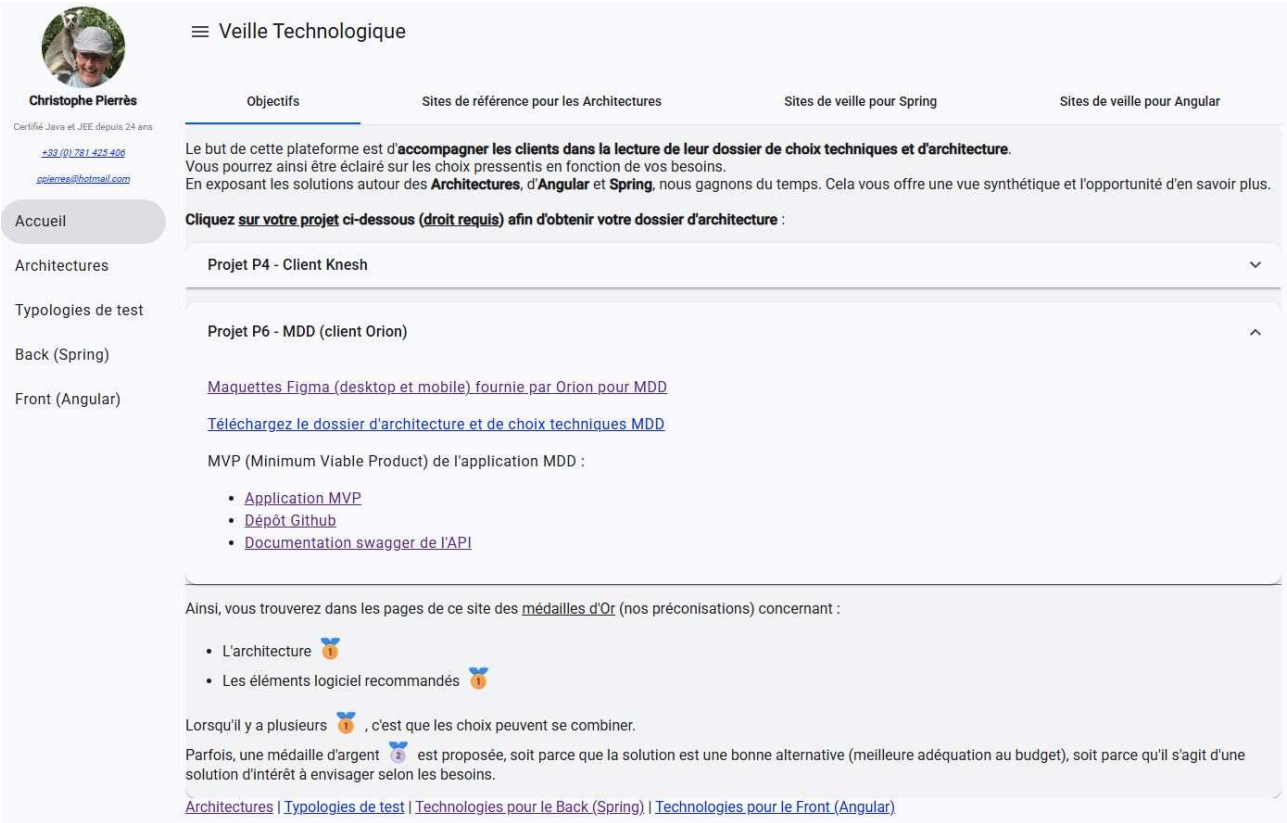
5. Annexes 18

5.1. Stack Spring Data JPA/MVC versus Spring Webflux 18

1. Introduction

Ce document est destiné à justifier les choix techniques par rapport aux besoins du projet MDD.

En parallèle de la lecture de ce document, nous vous inviterons à utiliser notre site de veille technologique dédié aux Architectures, ainsi qu'à Spring et Angular : https://veille.cpierres.dscloud.me/



Les sections suivantes vous guideront sur les choix effectués, en utilisant cette application de veille technologique. Ceci vous permettra ainsi d'avoir une vision comparative et documentée par rapport à notre proposition. Vous y trouverez des liens pour chaque technologie (sites officiels et autres liens utiles).

Les fonctionnalités demandées dans ce MVP (Minimum Viable Product) sont basiques. Il n'empêche que ce MVP doit permettre de choisir une base solide et évolutive, afin que l'application puisse toujours répondre aux nouvelles exigences techniques en toute circonstance.

Le MVP est accessible sur internet (browser ou mobile) : https://mdd.cpierres.dscloud.me/

Pour qu'un site de réseau social orienté développeurs puisse être viable, il faut que celui-ci réponde aux besoins exprimés aujourd'hui :

- Accepter une très forte montée en charge, sans ralentissement,
- Avoir le respect des développeurs (public exigeant) en démontrant les capacités les plus modernes sur le site de réseau social lui-même (actualisation instantanée de l'IHM pour tous les utilisateurs quel que soit le device – browser ou mobile)
- Proposer une IHM responsive avec des temps de réponse le plus rapide possible (quelque soit la complexité fonctionnelle)
- Être fortement résilient (tolérance aux pannes)

Mais, pour attirer et avoir du succès, un réseau social ne peut pas se contenter d'être géré comme une « application de gestion à l'ancienne ». Une application vraiment moderne se doit, non seulement d'être responsive, mais doit proposer des actualisations en temps réel pour tous les utilisateurs et sur tous les canaux (mobiles, browsers), bien entendu en respectant les contraintes précédentes. Par exemple,

- Réaliser facilement des chats en temps réel (dans le cas présent : afficher les nouveaux Posts/Articles ou commentaires en temps réel, ou encore actualiser en temps réel les statistiques de popularité).

Également un réseau social doit répondre à ce que font les concurrents déjà bien implantés :

- Gérer les canaux de communication modernes, tels que des dialogues vidéo entre développeurs par exemple (sans nécessité un outil tierce),
- Télécharger des grosses vidéos sur des sujets techniques et les diffuser en streaming (sans scotcher les threads bien entendu !)
- Si l'application venait à beaucoup s'étendre fonctionnellement, permettre d'évoluer vers une architecture micro-services, grâce à une bonne gestion des latences entre services,
- Intégrer une logique événementielle pour créer et partager des événements en continu (vers un micro-service spécialisé sur la communication – mail, sms ou autre) ; permettre la diffusion de flux multi-media (via kafka par exemple).

Pour répondre à ces exigences, les maitres mots sont de bien gérer l'asynchronisme et ne pas monopoliser les ressources du ou des serveurs du fait « d'attentes bloquantes ».

Côté frontend Angular permet depuis longtemps de créer des applications « responsive », en évitant des blocages de l'IHM grâce à une bonne gestion de l'asynchronisme. Cependant, derrière Angular côté backend, on propose souvent une architecture bloquante. De ce fait, même si Angular gère l'asynchronisme, il ne pourra pas éviter un ralentissement général de l'application du fait d'une surcharge d'un serveur par exemple (par exemple, thread monopolisé pour gérer le téléchargement d'une grosse vidéo).

Côté backend, Spring Webflux a été introduit avec Spring Framework 5.0, sorti en septembre 2017.

Cinq années d'existence avec une nouvelle version majeure 6 procurent une **maturité suffisante pour être choisi en confiance en 2025**. C'est une technologie d'avenir exigeant néanmoins des développeurs sénior !

Webflux est aujourd'hui une réponse mature pour permettre de l'asynchronisme de bout en bout (backend associé à base de données R2DBC) afin d'être fluide sur toute la chaine, sans monopoliser les précieux threads.

Nos premières préconisations sont donc :

- Frontend : Angular 19 (axé sur des améliorations de performance, une plus grande maintenabilité et facilité). L'API Signal a été introduite dans Angular 16 par exemple et est mature en v19.
- Backend : SpringBoot 3.4.4 (Spring 6) et Spring Webflux
- Database : Postgresql v17 avec un driver R2DBC (Reactive Relational Database Connectivity)

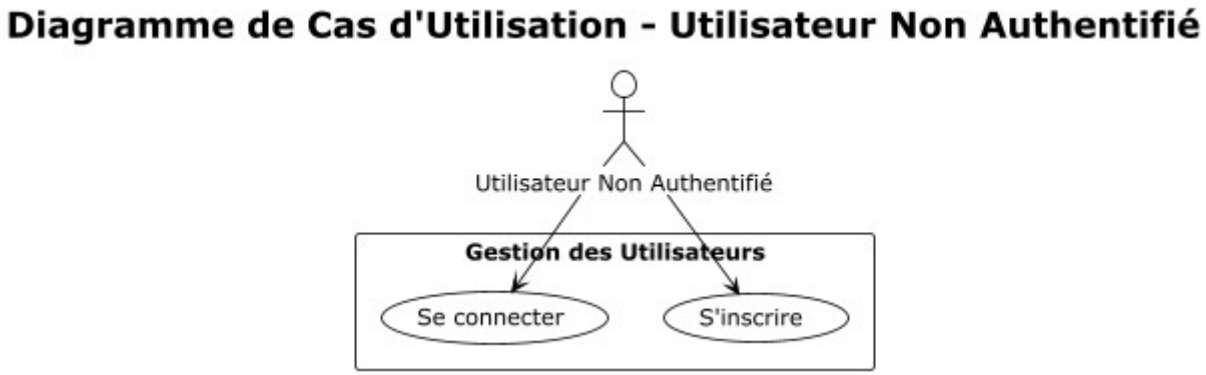
Pour la mise à jour en temps réel, nous gérerons cela efficacement via des SSE (Server Side Event) qui sont des flux unidirectionnels (plus légers que les WebSockets), et qui fonctionnent avec le protocole HTTP standard (ok avec les proxy). Les SSE s'intègrent nativement au HTML5 (API EventSource).

Vous trouverez dans les sections ci-dessous :

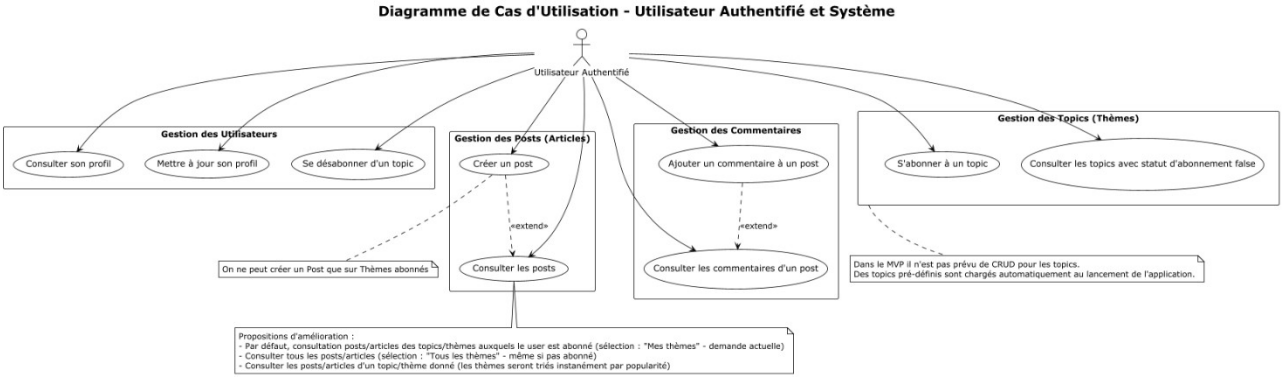
- Des diagrammes de use-case pour synthétiser les besoins du MVP
- Les motivations pour les choix techniques
- Le dossier d'architecture du MVP

2. Diagrammes de use-cases des besoins du MVP

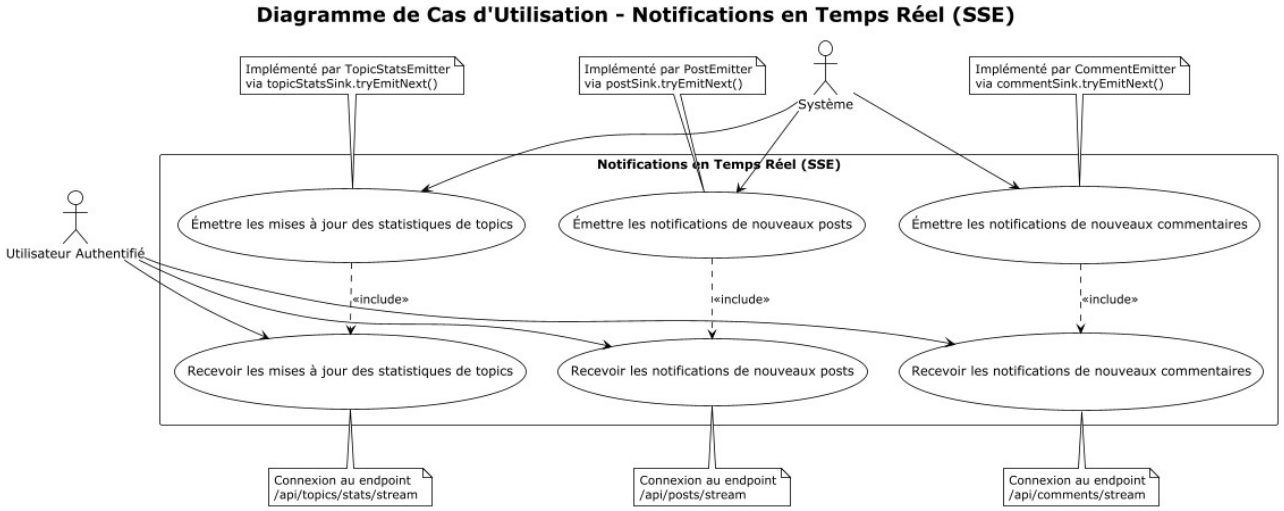
2.1. Utilisateur non authentifié



2.2. Utilisateur authentifié



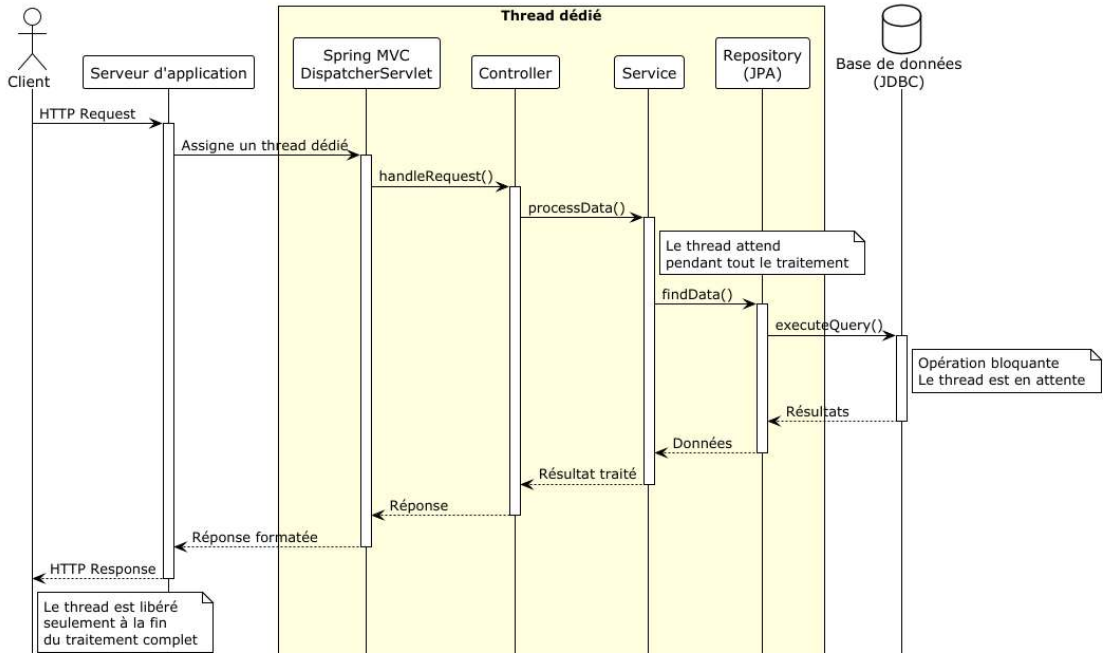
2.2.1. Proposition d'amélioration : notifications en temps réel (SSE)



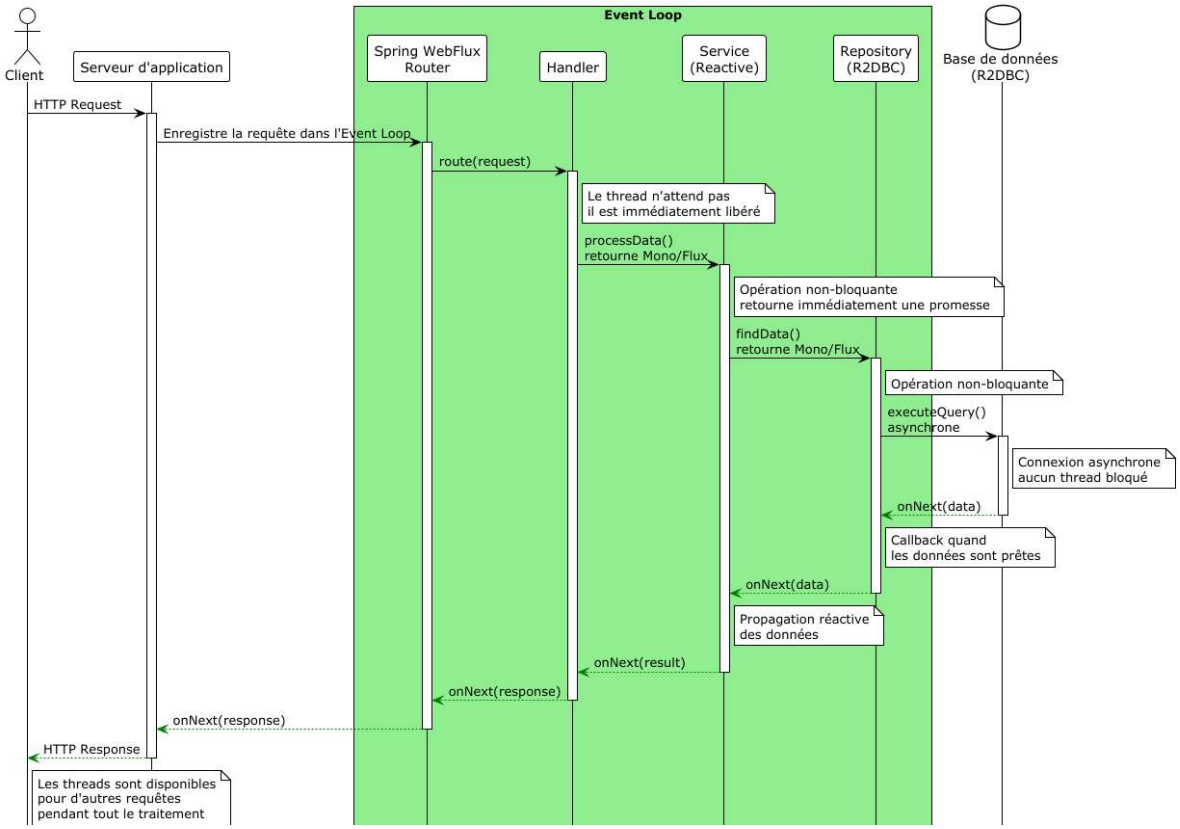
3. Choix techniques

3.1. Comparaison modèle Servlet/bloquant versus Réactive

3.1.1. Architecture bloquante (JEE ou Spring MVC avec JPA et JDBC)



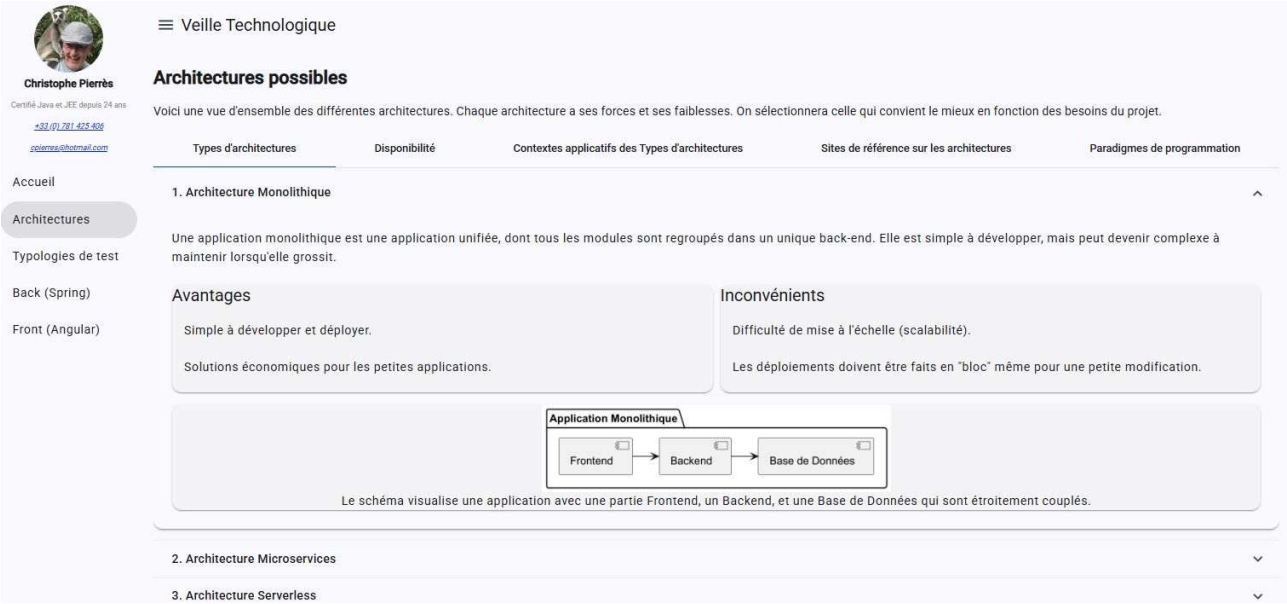
3.1.2. Architecture réactive - Spring WebFlux avec R2DBC



3.1.3. Architecture générale

Pour une introduction aux architectures possibles, cliquez sur Architectures puis sur chaque Type :

Etant donné la faible complexité de l'application, l'architecture Monolithique est celle retenue car suffisante pour ce premier lot :



Le backend est découplé du frontend en communiquant avec du REST sur protocole HTTP.

L'inconvénient de mise à l'échelle (scalabilité) est minoré du fait des choix techniques préconisés.

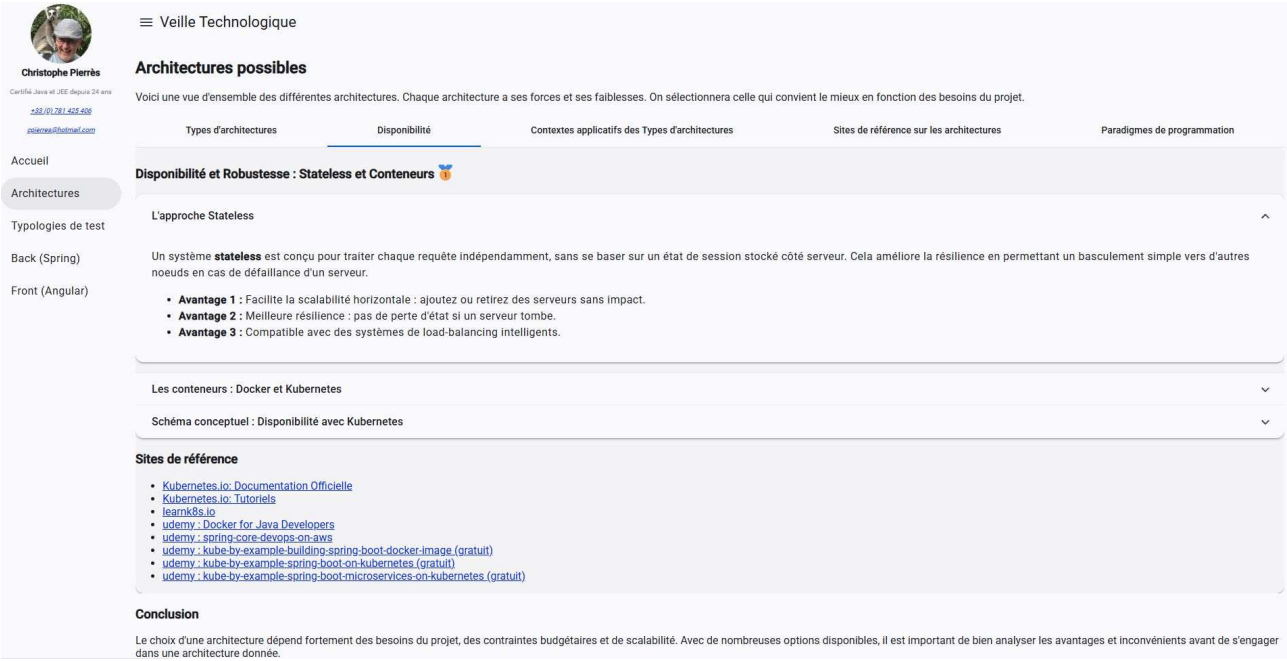
A ressource égale, on peut avoir une montée en charge beaucoup plus forte avec une architecture réactive de bout en bout.

3.1.3.1. Disponibilité

La disponibilité est un critère important. Le développement en architecture REST (REpresentational State Transfer) a l'avantage d'être stateless, ce qui permet d'assurer la robustesse et une disponibilité horizontale supplémentaire via la containerisation (Docker / Kubernetes).

Dans un premier temps, l'application sera déployée via DockerCompose (pile de containers docker représentant nos trois couches). Ce déploiement devrait répondre d'une manière satisfaisante pour un long moment du fait des technologies réactives et peu consommatrices de ressources et de Threads que nous avons choisies. La scabilité verticale (ressources d'une machine) est largement améliorée grâce à nos choix.

Dans un second temps, et seulement si besoin, nous pourrons encore améliorer grâce à une scalabilité horizontale.



Si besoin : scalabilité horizontale possible avec Kubernetes

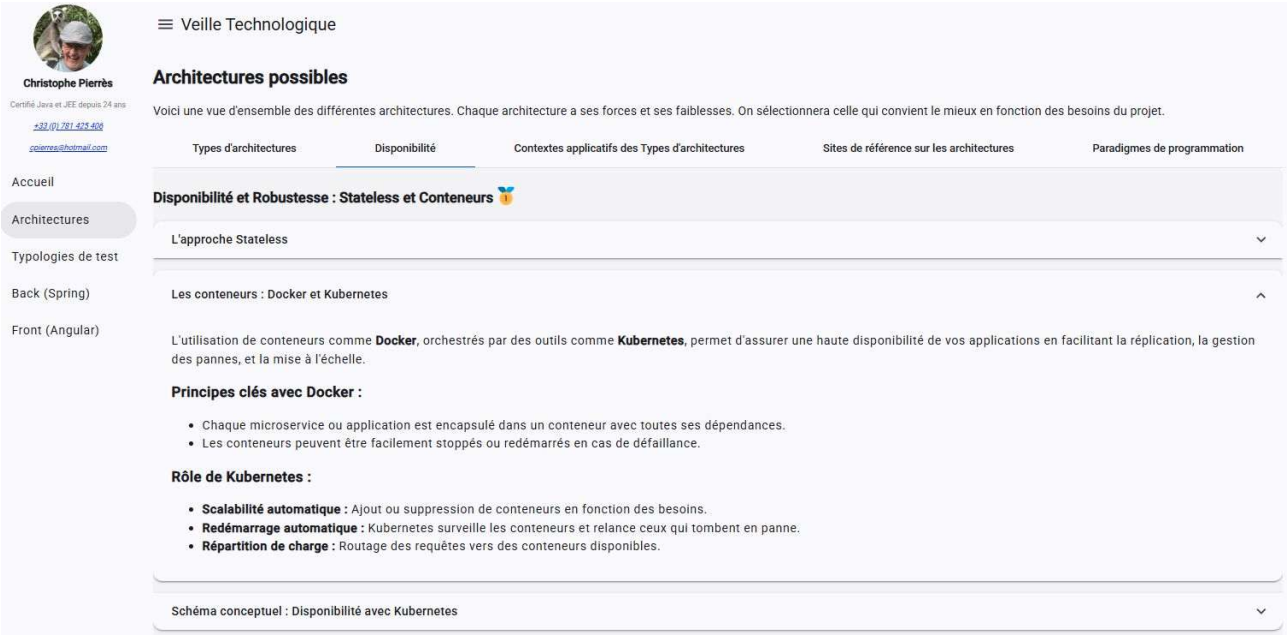
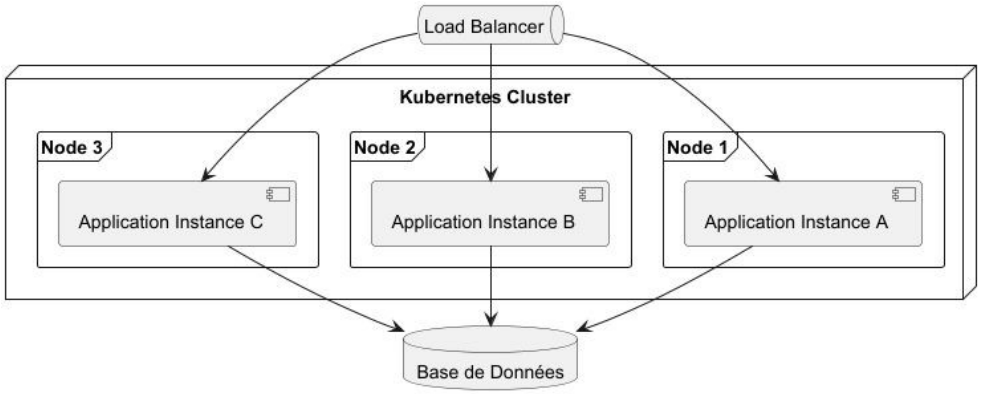


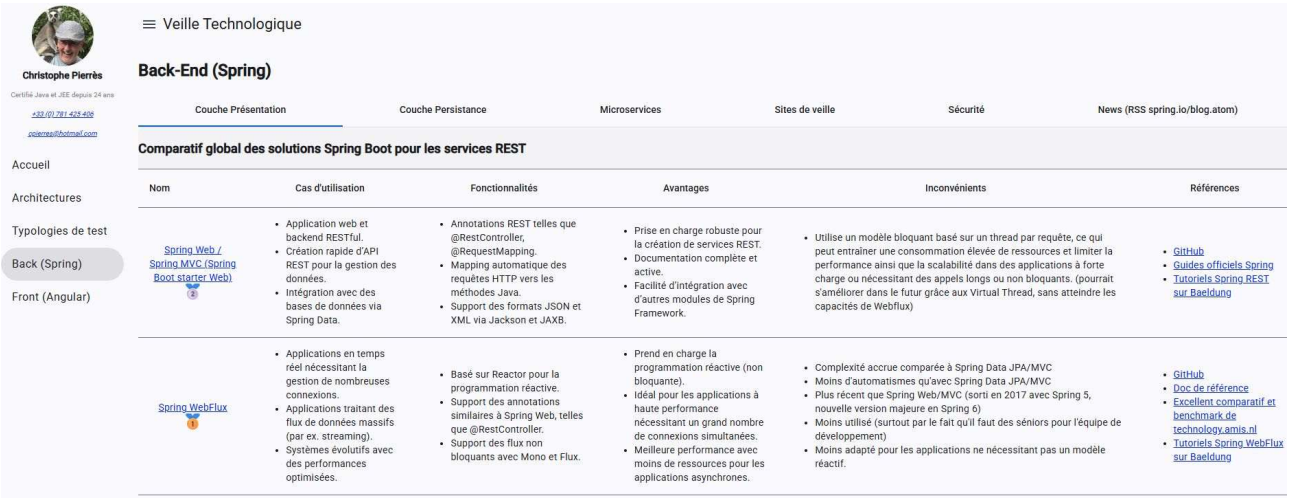
Schéma conceptuel : disponibilité avec Kubernetes



3.2. Backend

3.2.1. Service REST : Spring Webflux

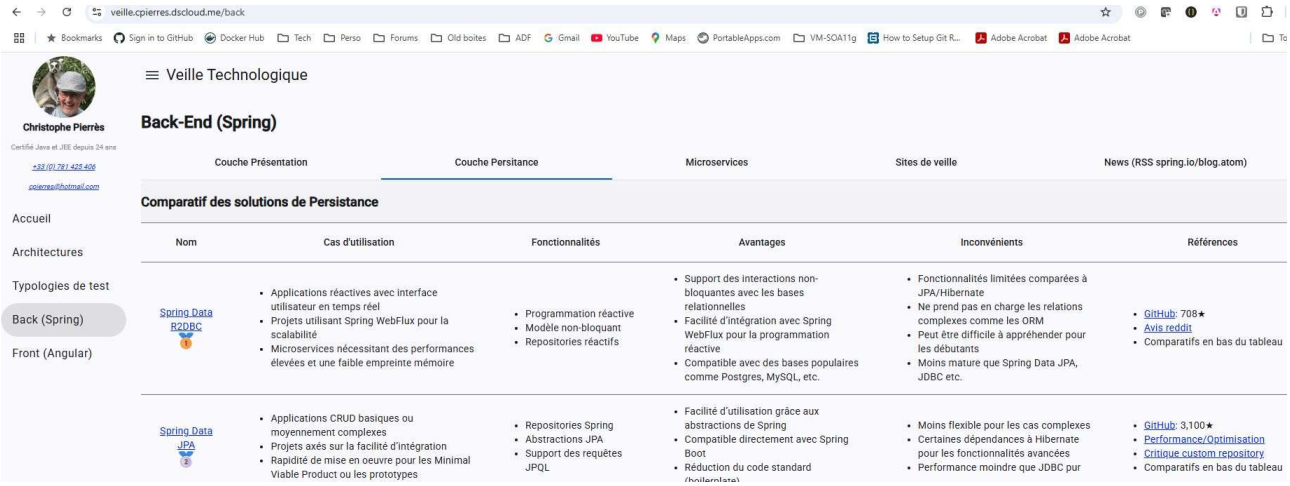
Cliquez sur : https://veille.cpierres.dscloud.me/back (ou bien Back (Spring) / Couche Présentation / REST)



Tout en bas de ce tableau internet, vous trouverez également des sites de formation et d'expertises ainsi qu'un comparatif entre Spring Data JPA/MVC et Spring Webflux (points communs et différences).

3.2.2. Persistance : base Postgresql v17 avec driver R2DBC

Cliquez sur le lien : Couche Persistance



En bas de ce tableau internet, vous trouverez également d'autres liens utiles.

3.3. Frontend : Angular v19

Nous choisissons cette version récente d'Angular car les technologies avancées sur lesquelles elle se base ont été introduites depuis la version 14 ou 16 et on peut les considérer aujourd'hui comme matures.

Il est bien temps d'utiliser tous les concepts modernes d'Angular qui apportent : performance, meilleure maintenabilité (standalone components) et plus de simplicité.

Citons l'API Signal : Lorsqu'une valeur contenue dans un Signal est modifiée, Angular réagit automatiquement aux mises à jour de manière optimisée, sans nécessiter un cycle complet de détection des changements.

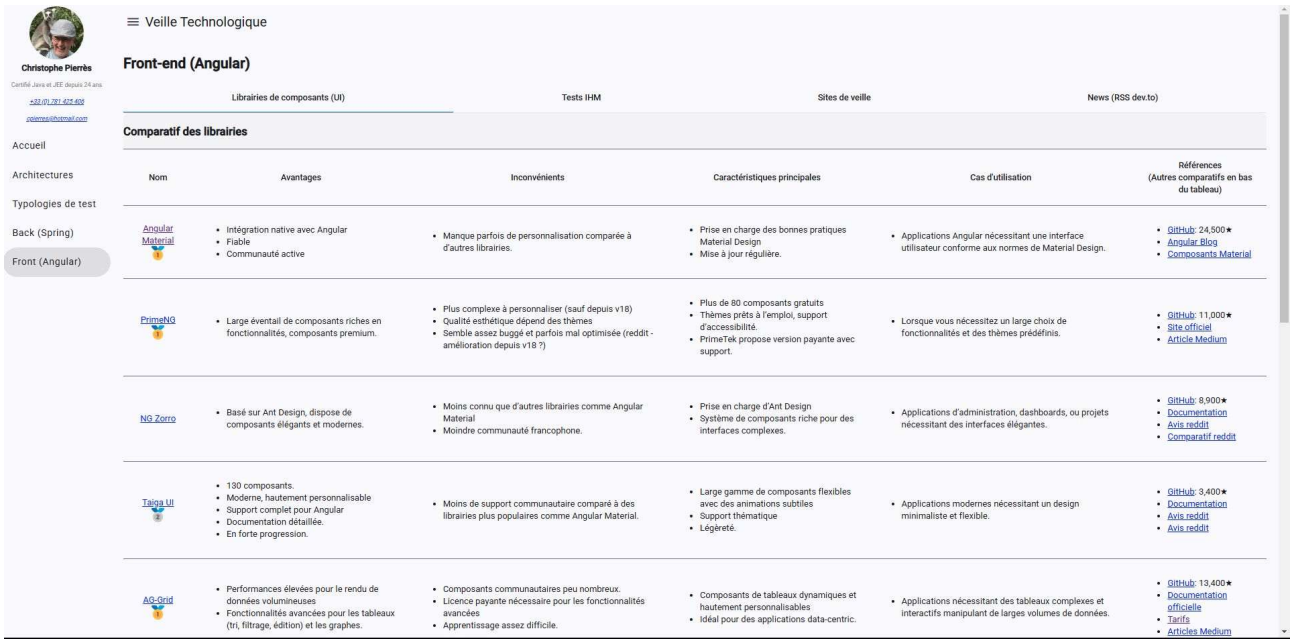
Les performances du **runtime Angular** et du nouveau **compilateur Ivy** permettent une amélioration significative des temps de chargement des applications.

3.3.1. Librairie de composants visuels

Une analyse des librairies de composants est disponible sur : https://veille.cpierres.dscloud.me/front

La solution incluse dans Angular est : **Angular Materials** 🥇. Nous la préconisons car robuste et intéressante.

Les composants de cette librairie ne couvrent pas toujours tous les besoins (ex : diagrammes de Gantt). Pour compléter les manques éventuels, nous proposons **PrimeNG** 🥇. Cette solution est la plus populaire d'après l'historique des stars. Néanmoins, ce tableau pourrait permettre d'affiner ce choix car si vous avez des demandes spécifiques (grosse volumétrie sur les tableaux ou composants spécifiques par exemple), d'autres solutions peuvent être décidées entre **AG-Grid** (payante) et la prometteuse librairie **Taigai UI** :



Christophe Pierrès
Certifié Java et JEE depuis 24 ans
+33 (0) 781 425 406
cpierres@hotmail.com

Accueil
Architectures
Typologies de test
Back (Spring)
Front (Angular)

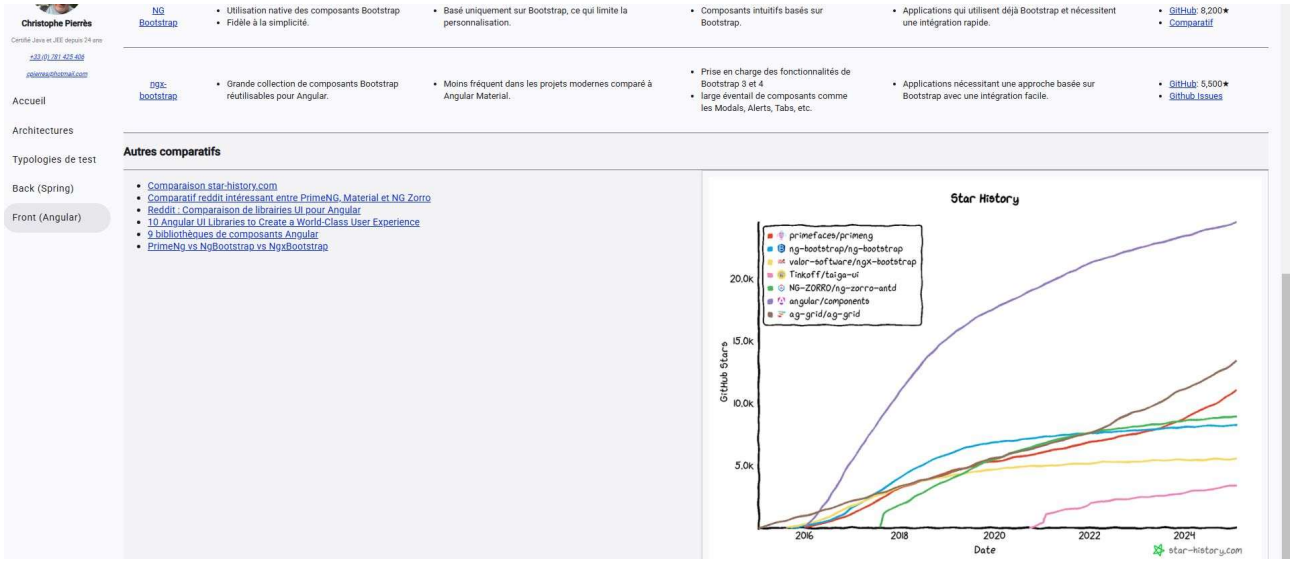
Veille Technologique

Front-end (Angular)

Librairies de composants (UI) | Tests IHM | Sites de veille | News (RSS dev.to)

Comparatif des librairies

Nom	Avantages	Inconvénients	Caractéristiques principales	Cas d'utilisation	Références (Autres comparatifs en bas du tableau)
Angular Material 🥇	• Intégration native avec Angular • Fiable • Communauté active	• Manque parfois de personnalisation comparée à d'autres librairies.	• Prise en charge des bonnes pratiques Material Design • Mise à jour régulière.	• Applications Angular nécessitant une interface utilisateur conforme aux normes de Material Design.	• GitHub: 24,500★ • Angular Blog • Composants Material
PrimeNG 🥇	• Large éventail de composants riches en fonctionnalités, composants premium.	• Plus complexe à personnaliser (sauf depuis v18) • Qualité esthétique dépend des thèmes • Semble assez buggé et parfois mal optimisée (reddit - amélioration depuis v18 ?)	• Plus de 80 composants gratuits • Thèmes prêts à l'emploi, support d'accessibilité. • PrimeTek propose version payante avec support.	• Lorsque vous nécessitez un large choix de fonctionnalités et des thèmes prédéfinis.	• GitHub: 11,000★ • Site officiel • Article Medium
NG Zorro	• Basé sur Ant Design, dispose de composants élégants et modernes.	• Moins connu que d'autres librairies comme Angular Material • Moindre communauté francophone.	• Prise en charge d'Ant Design • Système de composants riche pour des interfaces complexes.	• Applications d'administration, dashboards, ou projets nécessitant des interfaces élégantes.	• GitHub: 8,900★ • Documentation • Avis reddit • Comparatif reddit
Taiga UI 🥈	• 130 composants. • Moderne, hautement personnalisable • Support complet pour Angular • Documentation détaillée. • En forte progression.	• Moins de support communautaire comparé à des librairies plus populaires comme Angular Material.	• Large gamme de composants flexibles avec des animations subtiles • Support thématique • Légèreté.	• Applications modernes nécessitant un design minimaliste et flexible.	• GitHub: 3,400★ • Documentation • Avis reddit • Avis reddit
AG-Grid 🥇	• Performances élevées pour le rendu de données volumineuses • Fonctionnalités avancées pour les tableaux (tri, filtrage, édition) et les graphes.	• Composants communautaires peu nombreux. • Licence payante nécessaire pour les fonctionnalités avancées • Apprentissage assez difficile.	• Composants de tableaux dynamiques et hautement personnalisables • Idéal pour des applications data-centric.	• Applications nécessitant des tableaux complexes et interactifs manipulant de larges volumes de données.	• GitHub: 13,400★ • Documentation officielle • Tarifs • Articles Medium



Christophe Pierrès
Certifié Java et JEE depuis 24 ans
+33 (0) 781 425 406
cpierres@hotmail.com

Accueil
Architectures
Typologies de test
Back (Spring)
Front (Angular)

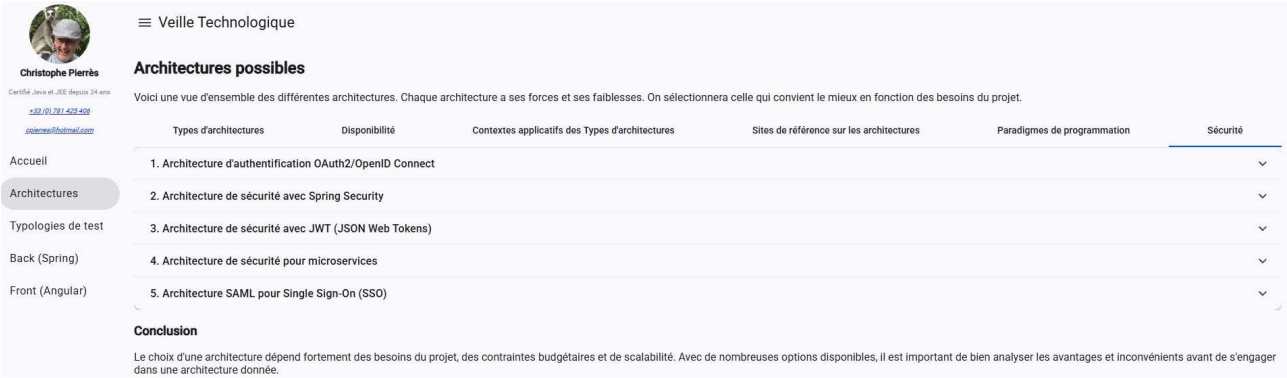
NG Bootstrap	• Utilisation native des composants Bootstrap • Fidèle à la simplicité.	• Basé uniquement sur Bootstrap, ce qui limite la personnalisation.	• Composants intuitifs basés sur Bootstrap.	• Applications qui utilisent déjà Bootstrap et nécessitent une intégration rapide.	• GitHub: 8,200★ • Comparatif
ngx-bootstrap	• Grande collection de composants Bootstrap réutilisables pour Angular.	• Moins fréquent dans les projets modernes comparé à Angular Material.	• Prise en charge des fonctionnalités de Bootstrap 3 et 4 • large éventail de composants comme les Modals, Alerts, Tabs, etc.	• Applications nécessitant une approche basée sur Bootstrap avec une intégration facile.	• GitHub: 5,500★ • Github Issues

Autres comparatifs

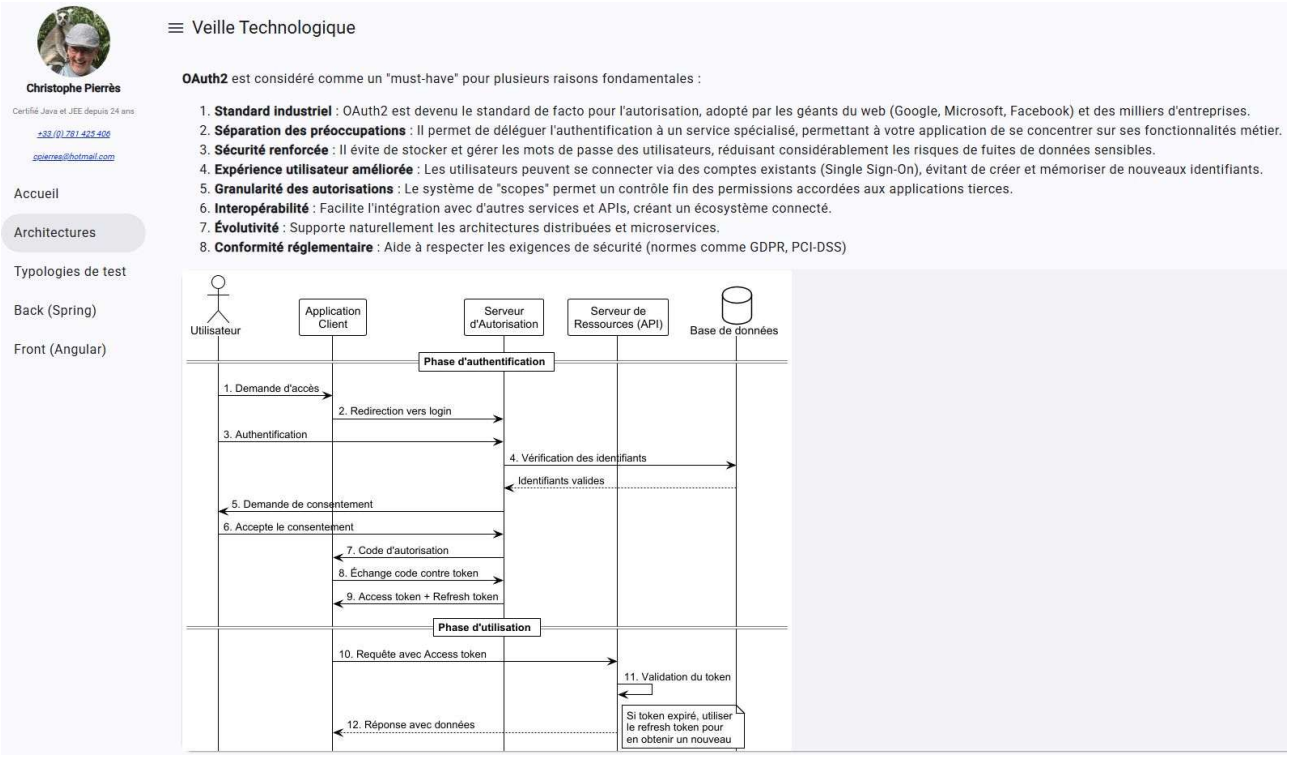
- Comparaison star-history.com
- Comparatif reddit intéressant entre PrimeNG, Material et NG Zorro
- Reddit : Comparaison de librairies UI pour Angular
- 10 Angular UI Libraries to Create a World-Class User Experience
- 9 bibliothèques de composants Angular
- PrimeNg vs NgBootstrap vs NgxBootstrap

3.4. Sécurité

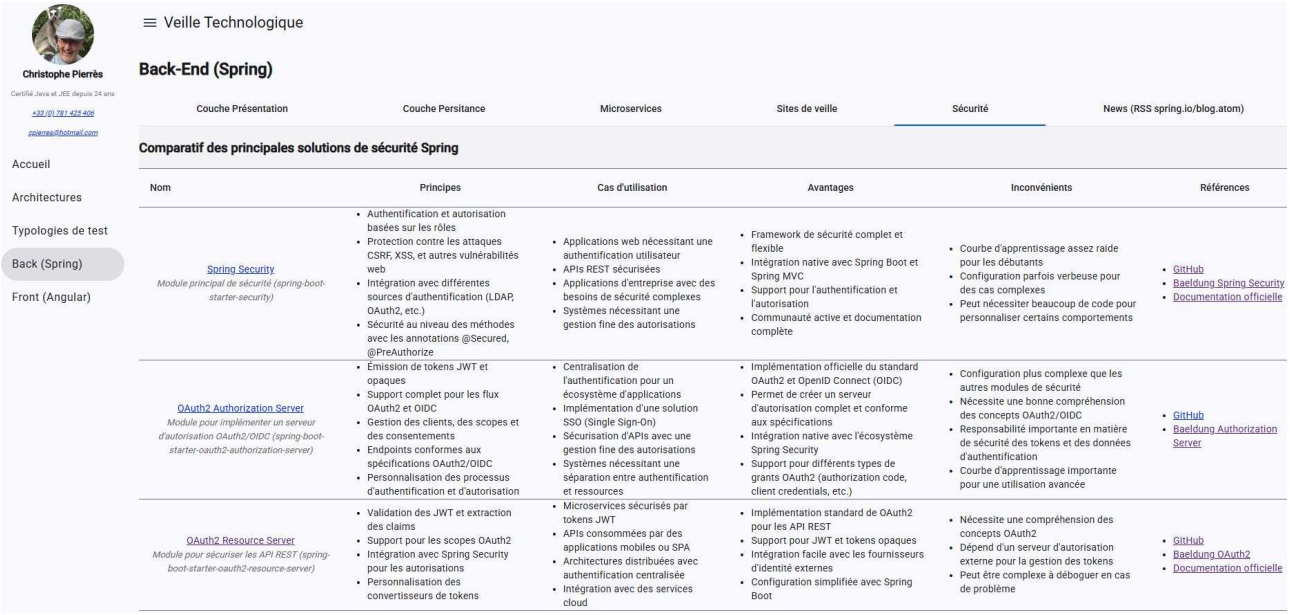
Reportez-vous au menu Architectures puis à l'onglet Sécurité :



Dépliez "1. Architecture d'authentification OAuth2/OpenID Connect", qui est l'architecture de sécurité générale retenue de facto pour MDD :



Maintenant, accédez au menu Back (Spring) puis Sécurité. Voici les modules de sécurité « classiques » :



Christophe Pierrès
Certifié Java et JEE depuis 24 ans
+33 (0) 781 425 406
cpierres@hotmail.com

Accueil
Architectures
Typologies de test
Back (Spring)
Front (Angular)

≡ Veille Technologique

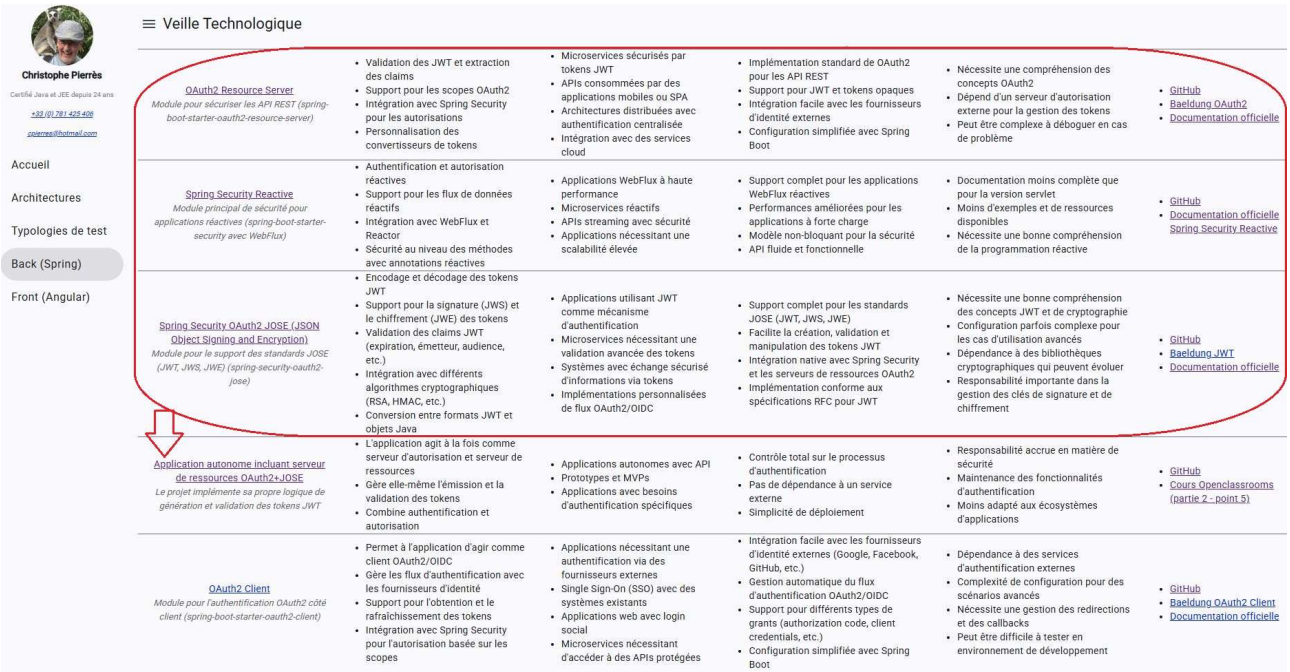
Back-End (Spring)

Couche Présentation | Couche Persitance | Microservices | Sites de veille | Sécurité | News (RSS spring.io/blog.atom)

Comparatif des principales solutions de sécurité Spring

Nom	Principes	Cas d'utilisation	Avantages	Inconvénients	Références
Spring Security *Module principal de sécurité (spring-boot-starter-security)*	• Authentification et autorisation basées sur les rôles • Protection contre les attaques CSRF, XSS, et autres vulnérabilités web • Intégration avec différentes sources d'authentification (LDAP, OAuth2, etc.) • Sécurité au niveau des méthodes avec les annotations @Secured, @PreAuthorize	• Applications web nécessitant une authentification utilisateur • APIs REST sécurisées • Applications d'entreprise avec des besoins de sécurité complexes • Systèmes nécessitant une gestion fine des autorisations	• Framework de sécurité complet et flexible • Intégration native avec Spring Boot et Spring MVC • Support pour l'authentification et l'autorisation • Communauté active et documentation complète	• Courbe d'apprentissage assez raide pour les débutants • Configuration parfois verbeuse pour des cas complexes • Peut nécessiter beaucoup de code pour personnaliser certains comportements	• GitHub • Baeldung Spring Security • Documentation officielle
OAuth2 Authorization Server *Module pour implémenter un serveur d'autorisation OAuth2/OIDC (spring-boot-starter-oauth2-authorization-server)*	• Émission de tokens JWT et opaques • Support complet pour les flux OAuth2 et OIDC • Gestion des clients, des scopes et des consentements • Endpoints conformes aux spécifications OAuth2/OIDC • Personnalisation des processus d'authentification et d'autorisation	• Centralisation de l'authentification pour un écosystème d'applications • Implémentation d'une solution SSO (Single Sign-On) • Sécurisation d'APIs avec une gestion fine des autorisations • Systèmes nécessitant une séparation entre authentification et ressources	• Implémentation officielle du standard OAuth2 et OpenID Connect (OIDC) • Permet de créer un serveur d'autorisation complet et conforme aux spécifications • Intégration native avec l'écosystème Spring Security • Support pour différents types de grants OAuth2 (authorization code, client credentials, etc.)	• Configuration plus complexe que les autres modules de sécurité • Nécessite une bonne compréhension des concepts OAuth2/OIDC • Responsabilité importante en matière de sécurité des tokens et des données d'authentification • Courbe d'apprentissage importante pour une utilisation avancée	• GitHub • Baeldung Authorization Server
OAuth2 Resource Server *Module pour sécuriser les API REST (spring-boot-starter-oauth2-resource-server)*	• Validation des JWT et extraction des claims • Support pour les scopes OAuth2 • Intégration avec Spring Security pour les autorisations • Personnalisation des convertisseurs de tokens	• Microservices sécurisés par tokens JWT • APIs consommées par des applications mobiles ou SPA • Architectures distribuées avec authentification centralisée • Intégration avec des services cloud	• Implémentation standard de OAuth2 pour les API REST • Support pour JWT et tokens opaques • Intégration facile avec les fournisseurs d'identité externes • Configuration simplifiée avec Spring Boot	• Nécessite une compréhension des concepts OAuth2 • Dépend d'un serveur d'autorisation externe pour la gestion des tokens • Peut être complexe à déboguer en cas de problème	• GitHub • Baeldung OAuth2 • Documentation officielle

Pour ce qui concerne MDD, les modules que nous utiliserons seront les suivants ; ils seront embarqués dans notre application qui sera autonome, pour plus de simplicité :



Christophe Pierrès
Certifié Java et JEE depuis 24 ans
+33 (0) 781 425 406
cpierres@hotmail.com

Accueil
Architectures
Typologies de test
Back (Spring)
Front (Angular)

≡ Veille Technologique

OAuth2 Resource Server *Module pour sécuriser les API REST (spring-boot-starter-oauth2-resource-server)*	• Validation des JWT et extraction des claims • Support pour les scopes OAuth2 • Intégration avec Spring Security pour les autorisations • Personnalisation des convertisseurs de tokens	• Microservices sécurisés par tokens JWT • APIs consommées par des applications mobiles ou SPA • Architectures distribuées avec authentification centralisée • Intégration avec des services cloud	• Implémentation standard de OAuth2 pour les API REST • Support pour JWT et tokens opaques • Intégration facile avec les fournisseurs d'identité externes • Configuration simplifiée avec Spring Boot	• Nécessite une compréhension des concepts OAuth2 • Dépend d'un serveur d'autorisation externe pour la gestion des tokens • Peut être complexe à déboguer en cas de problème	• GitHub • Baeldung OAuth2 • Documentation officielle
Spring Security Reactive *Module principal de sécurité pour applications réactives (spring-boot-starter-security avec WebFlux)*	• Authentification et autorisation réactives • Support pour les flux de données réactifs • Intégration avec WebFlux et Reactor • Sécurité au niveau des méthodes avec annotations réactives	• Applications WebFlux à haute performance • Microservices réactifs • APIs streaming avec sécurité • Applications nécessitant une scalabilité élevée	• Support complet pour les applications WebFlux réactives • Performances améliorées pour les applications à forte charge • Modèle non-bloquant pour la sécurité • API fluide et fonctionnelle	• Documentation moins complète que pour la version servlet • Moins d'exemples et de ressources disponibles • Nécessite une bonne compréhension de la programmation réactive	• GitHub • Documentation officielle Spring Security Reactive
Spring Security OAuth2 JOSE (JSON Object Signing and Encryption) *Module pour le support des standards JOSE (JWT, JWS, JWE) (spring-security-oauth2-jose)*	• Encodage et décodage des tokens JWT • Support pour la signature (JWS) et le chiffrement (JWE) des tokens • Validation des claims JWT (expiration, émetteur, audience, etc.) • Intégration avec différents algorithmes cryptographiques (RSA, HMAC, etc.) • Conversion entre formats JWT et objets Java	• Applications utilisant JWT comme mécanisme d'authentification • Microservices nécessitant une validation avancée des tokens • Systèmes avec échange sécurisé d'informations via tokens • Implémentations personnalisées de flux OAuth2/OIDC	• Support complet pour les standards JOSE (JWT, JWS, JWE) • Facilite la création, validation et manipulation des tokens JWT • Intégration native avec Spring Security et les serveurs de ressources OAuth2 • Implémentation conforme aux spécifications RFC pour JWT	• Nécessite une bonne compréhension des concepts JWT et de cryptographie • Configuration parfois complexe pour les cas d'utilisation avancés • Dépendance à des bibliothèques cryptographiques qui peuvent évoluer • Responsabilité importante dans la gestion des clés de signature et de chiffrement	• GitHub • Baeldung JWT • Documentation officielle
Application autonome incluant serveur de ressources OAuth2+JOSE *Le projet implémente sa propre logique de génération et validation des tokens JWT*	• L'application agit à la fois comme serveur d'autorisation et serveur de ressources • Gère elle-même l'émission et la validation des tokens • Combine authentification et autorisation	• Applications autonomes avec API • Prototypes et MVPs • Applications avec besoins d'authentification spécifiques	• Contrôle total sur le processus d'authentification • Pas de dépendance à un service externe • Simplicité de déploiement	• Responsabilité accrue en matière de sécurité • Maintenance des fonctionnalités d'authentification • Moins adapté aux écosystèmes d'applications	• GitHub • Cours Openclassrooms (partie 2 - point 5)
OAuth2 Client *Module pour l'authentification OAuth2 côté client (spring-boot-starter-oauth2-client)*	• Permet à l'application d'agir comme client OAuth2/OIDC • Gère les flux d'authentification avec les fournisseurs d'identité • Support pour l'obtention et le rafraîchissement des tokens • Intégration avec Spring Security pour l'autorisation basée sur les scopes	• Applications nécessitant une authentification via des fournisseurs externes • Single Sign-On (SSO) avec des systèmes existants • Applications web avec login social • Microservices nécessitant d'accéder à des APIs protégées	• Intégration facile avec les fournisseurs d'identité externes (Google, Facebook, GitHub, etc.) • Gestion automatique du flux d'authentification OAuth2/OIDC • Support pour différents types de grants (authorization code, client credentials, etc.) • Configuration simplifiée avec Spring Boot	• Dépendance à des services d'authentification externes • Complexité de configuration pour des scénarios avancés • Nécessite une gestion des redirections et des callbacks • Peut être difficile à tester en environnement de développement	• GitHub • Baeldung OAuth2 Client • Documentation officielle

Spring Security Reactive est le module principal pour les applications réactives.

Oauth2 Resource Server est une configuration qui permet à une application de se comporter comme un serveur de ressources OAuth 2.0. Un serveur de ressources est responsable de protéger les ressources (données, API) et de vérifier que les requêtes entrantes contiennent des tokens valides avant d'autoriser l'accès.

Dans l'architecture OAuth 2.0 complète, on distingue généralement :

- Le serveur d'autorisation : responsable de l'authentification des utilisateurs et de l'émission des tokens
- Le serveur de ressources : responsable de la validation des tokens et de la protection des ressources

Dans le cas de P6MDD, l'application jouera les deux rôles à la fois, d'une manière autonome :

- Elle émet des tokens JWT (rôle de serveur d'autorisation) via les endpoints /api/auth/login et /api/auth/register (utilisation du **module JOSE**)
- Elle valide ces tokens pour protéger ses propres ressources (rôle de serveur de ressources – conféré grâce au module **Oauth2 Resource Server**). Un serveur de ressources est responsable de protéger les ressources (données, API) et de vérifier que les requêtes entrantes contiennent des tokens valides avant d'autoriser l'accès

Cette approche "tout-en-un" est courante pour les applications qui n'ont pas besoin de partager leur système d'authentification avec d'autres applications. Elle simplifie l'architecture tout en bénéficiant des avantages de sécurité du protocole OAuth 2.0 et des JWT.

Nous utiliserons une clé symétrique avec encryptage HS256. Une clé asymétrique se justifierait dans le cas où il y aurait un partage de la sécurité entre plusieurs services distribués, ce qui n'est pas le cas pour MDD.

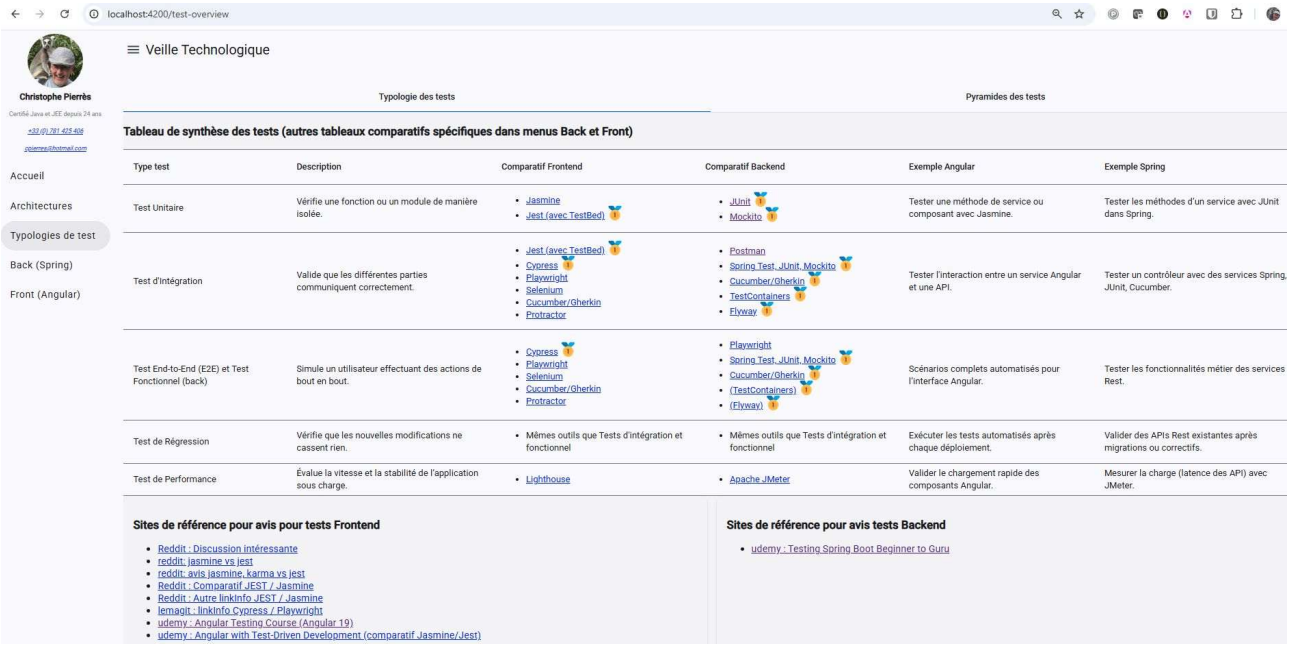
Dans un second temps (non demandé dans le MVP), il serait intéressant de permettre « l'authentification sociale » (via son compte google, github ou facebook par exemple). Cela éviterait à l'utilisateur d'avoir à s'enregistrer avec un compte spécifique et de devoir mémoriser un mot de passe. Dans ce cas, il faudra ajouter le module **OAuth2 client**. Cela fait partie des must-have pour un réseau social pour les développeurs.

3.5. Testing

Cette problématique ne fait pas partie du scope du MVP. Cependant, ces choix devront arriver vite dès la confirmation du lancement du projet. Par conséquent, voici nos préconisations.

Ce tableau présente une synthèse des choix de solutions de testing pour le front et le back :

https://veille.cpierres.dscloud.me/test-overview



Les deux sous-sections suivantes présentent les justifications de ces choix.

3.5.1. Front-end testing

Pour une analyse détaillée des choix de librairies, consultez le lien : https://veille.cpierres.dscloud.me/front ou menu **Front (Angular)**, puis onglet **Tests IHM** :

Christophe Pierrès

Certifié Java et JEE depuis 24 ans

+33 (0) 781 424 406

spierres@hotmail.com

Accueil

Architectures

Typologies de test

Back (Spring)

Front (Angular)

localhost:4200/front

Veille Technologique

Front-end (Angular)

Librairies de composants (UI) | Tests IHM | Sites de veille | News (RSS dev.to)

Comparatif des librairies de test

Nom	Type de Test	Caractéristiques	Points forts	Points faibles	Références (comparatifs bas du tableau)
Jasmine	• Unit Testing	• Framework de test unitaire populaire • Écrit en JavaScript • Approche BDD (Behavior Driven Development)	• Intégré par défaut dans Angular • Syntaxe simple et facile à comprendre • Bien adapté aux tests frontend • Écosystème mature et bien documenté	• Pas conçu pour exécuter les tests dans les navigateurs ou directement avec Node.js • Pas conçu pour gérer les rythmes de test asynchrone avancés	• GitHub: 15,800★ • StackShare • Comparatifs en bas du tableau
Karma	• Test Runner	• Permet l'exécution des tests dans des navigateurs réels • Compatible avec plusieurs frameworks (Jasmine, Mocha, etc.) • Intégration avec CI/CD (Continuous Integration)	• Intégré par défaut dans Angular • S'exécute sur plusieurs navigateurs • Prend en charge de nombreux frameworks de test	• Configuration complexe pour certains projets • Performance plus lente par rapport aux solutions modernes comme Jest	• GitHub: 12,000★ • StackShare • Reddit : Avis et linkinfo • Comparatifs en bas du tableau
TestBed	• Unit Testing (Angular Test Framework)	• Solution native pour le test unitaire dans Angular • Permet de tester les composants, modules et services Angular • Par défaut a besoin de Jasmine et Karma pour l'exécution • Ou bien de Jest	• Solution introduite et maintenue directement par Angular • Excellente intégration avec Angular • Prend en charge les tests unitaires et semi-intégrés	• Dépend de Karma pour l'exécution, ce qui peut être lent (mais Jest peut remplacer Karma) • Limité aux tests d'unités ou composés liés à Angular	• GitHub: 97,000★ • Github issues • StackShare • Testing Angular Guide • Tutoriel intéressant
Jest	• Unit Testing • Integration Testing	• Framework tout-en-un : assertion, mocking, backend intégrés • Compatible avec JavaScript, TypeScript, Angular, React, Vue.js • Approche moderne avec tests parallèles automatiques	• Exécution des tests rapide grâce au parallélisme • Le meilleur choix pour les projets modernes utilisant Angular ou React • Excellent support des tests d'instantanés (snapshot testing)	• Léger apprentissage requis pour les développeurs habitués à Jasmine/Karma • Plus difficile à configurer pour les environnements multi-navigateurs	• GitHub: 44,500★ • StackShare • Comparatifs en bas du tableau
Protractor	• End-to-end (E2E) Testing	• Framework de test automatisé pour les applications Angular • Support natif des synchronisations Angular • Compatible avec Selenium et WebDriver	• Conçu spécifiquement pour Angular • Large support grâce aux API Selenium • Bonne documentation officielle	• Déprécié depuis 2020 (recherche d'alternatives nécessaire) • Capacité limitée pour les applications non-Angular	• GitHub: 8,700★ • StackShare
Cypress	• E2E Testing	• Tests end-to-end interactifs pour les applications web modernes • Rapports intégrés et faciles à configurer • Moins dépendant d'environnements externes par rapport à d'autres outils	• Excellente expérience utilisateur pour les tests frontend • Facile à intégrer avec CI/CD • Documentation bien écrite et claire	• Principalement conçu pour tester les navigateurs (pas d'autre type d'applications)	• GitHub: 46,000★ • StackShare • Avis sur Reddit (vs playwright) • Autre avis sur Reddit • OpenClassrooms : Cypress
Playwright	• E2E Testing	• Framework moderne pour le test E2E • Compatible avec les navigateurs modernes (Chromium, Firefox, WebKit) • Supporte les tests via plusieurs langages (JavaScript, TypeScript, Python, Java, C#)	• Performances rapides grâce à des sessions de navigateur headless • Fonctionnalités avancées comme les tests parallèles, les fixtures et les traces interactives • Documentation riche et exemples faciles à suivre • Automatisation multiplateforme (Windows, macOS, Linux)	• Pas conçu pour les tests unitaires • Configuration initiale plus complexe pour les débutants • Maîtrise requise des promesses pour gérer les tests asynchrones	• GitHub: 69,200★ • StackShare • Reddit : avis Playwright (vs cypress et selenium)

Autres comparatifs et sites de référence

- Reddit : Discussion intéressante
- reddit: jasmine vs jest
- reddit: avis jasmine, karma vs jest
- Reddit : Comparatif JEST / Jasmine
- Reddit : Autre linkinfo JEST / Jasmine
- lemagit : linkinfo Cypress / Playwright
- udemy : Angular Testing Course (Angular 19)
- udemy : Angular with Test-Driven Development (comparatif Jasmine/Jest)

Angular intègre par défaut Jasmine, Karma et TestBed. Cependant, notre veille technologique montre que beaucoup se sont tournés vers **Jest** et **Cypress,** que nous retenons. Dans un post récent sur reddit, on apprend que l'équipe d'Angular elle-même envisage de changer leurs propositions par défaut (éventuellement vers Jest), ce qui confirme notre proposition. Par ailleurs, **TestBed** fonctionne parfaitement avec Jest.

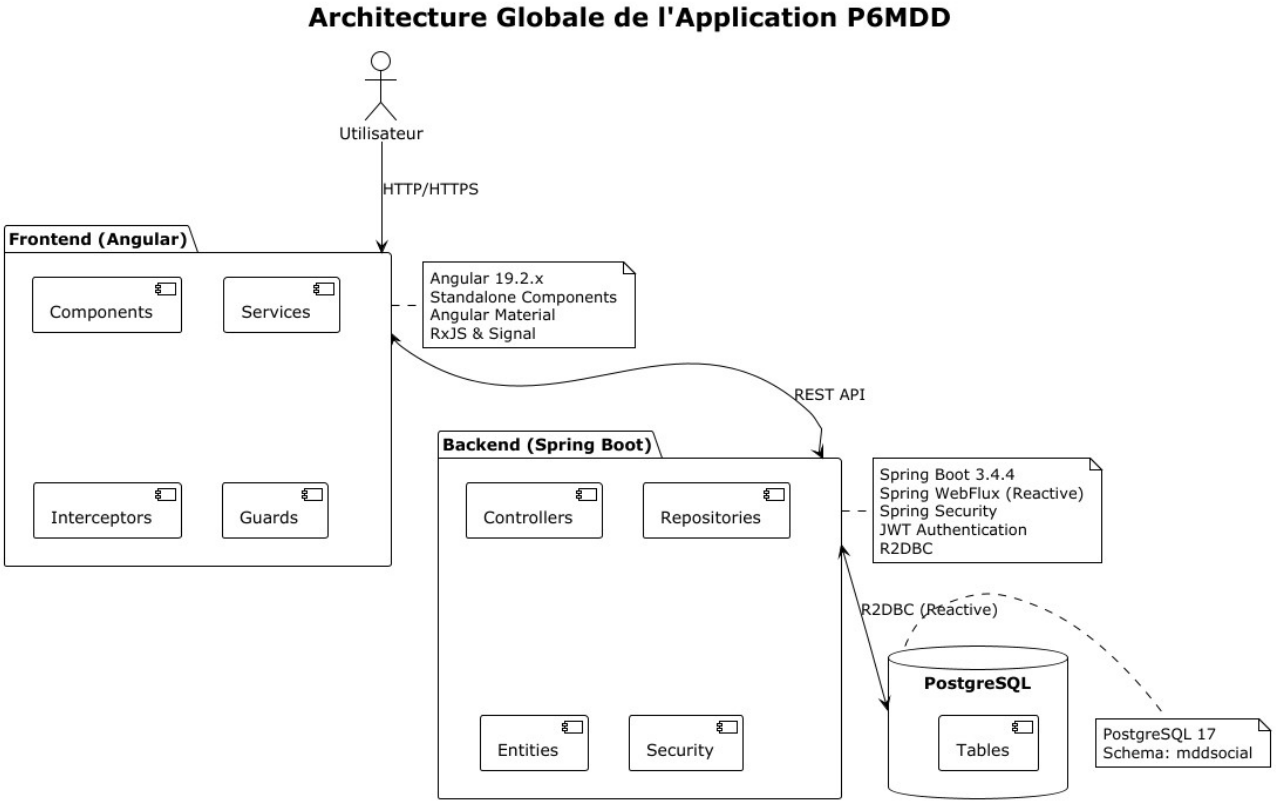
3.5.2. Back-end testing

Pour Spring, l'intégration des capacités de test est particulièrement bien soignée et les propositions par défaut sont excellentes. Spring permet d'intégrer avec facilité des briques optionnelles que nous avons retenues (TestContainers). Le tableau vu précédemment au début de cette section confirme les choix suivants :

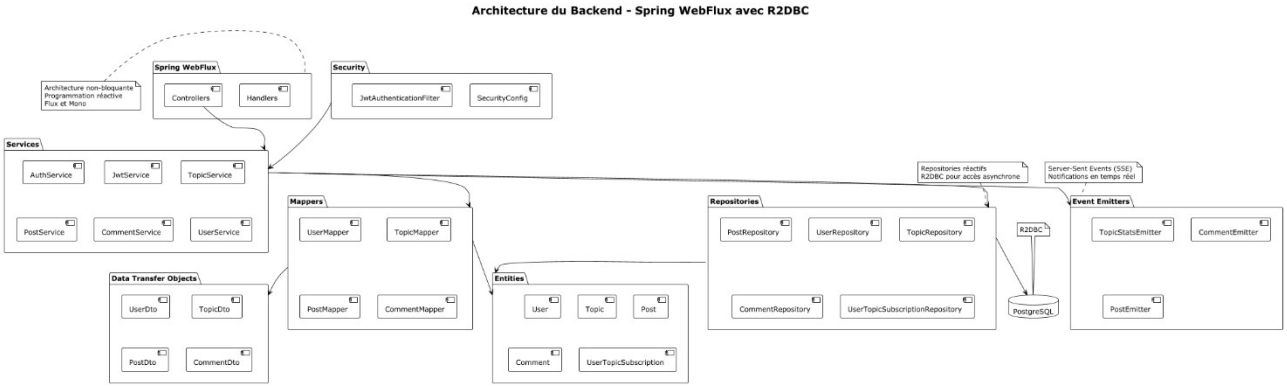
SpringTest (Junit, **Mockito)**, **Cucumber/Gherkin** (expression des tests en langage naturel permettant aux fonctionnels de mieux comprendre le résultat des tests d'intégration ou fonctionnels), **TestContainers** et **Flyway** pour la mise en œuvre de containers réalistes si besoin pour des jeux de tests plus sophistiqués/poussés que de simples mock.

4. Dossier d'architecture

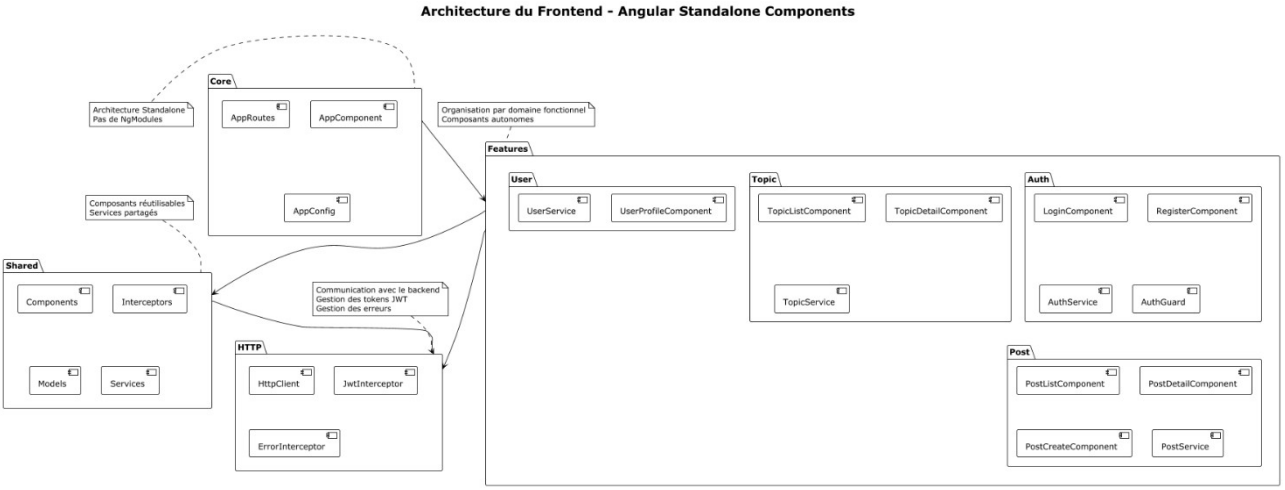
4.1. Architecture globale



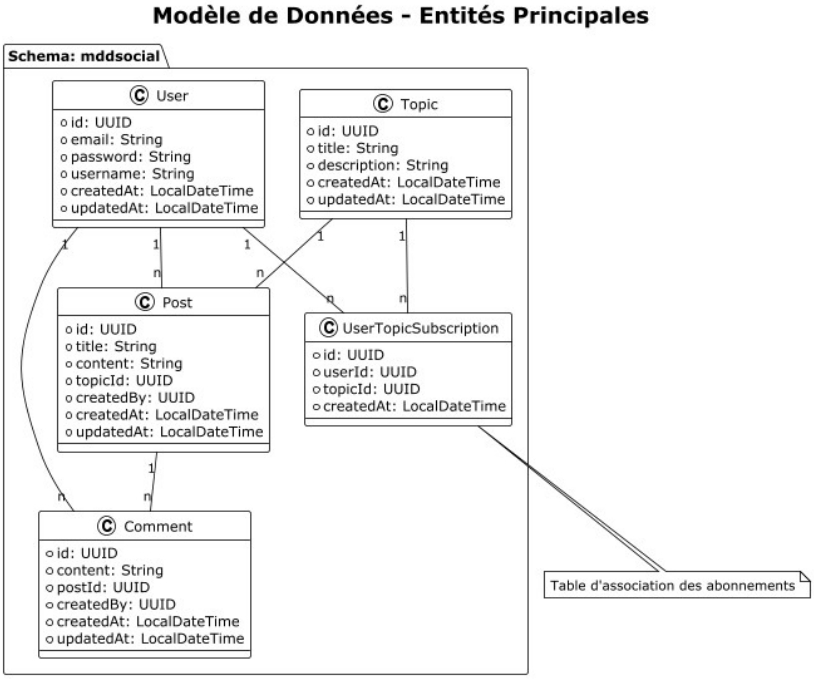
4.2. Architecture Reactive du Backend



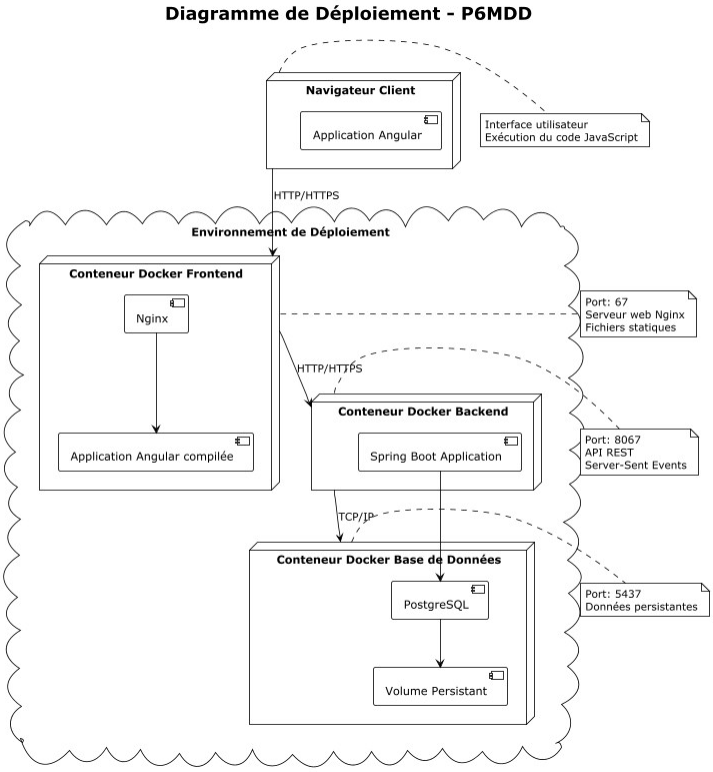
4.3. Architecture du Frontend



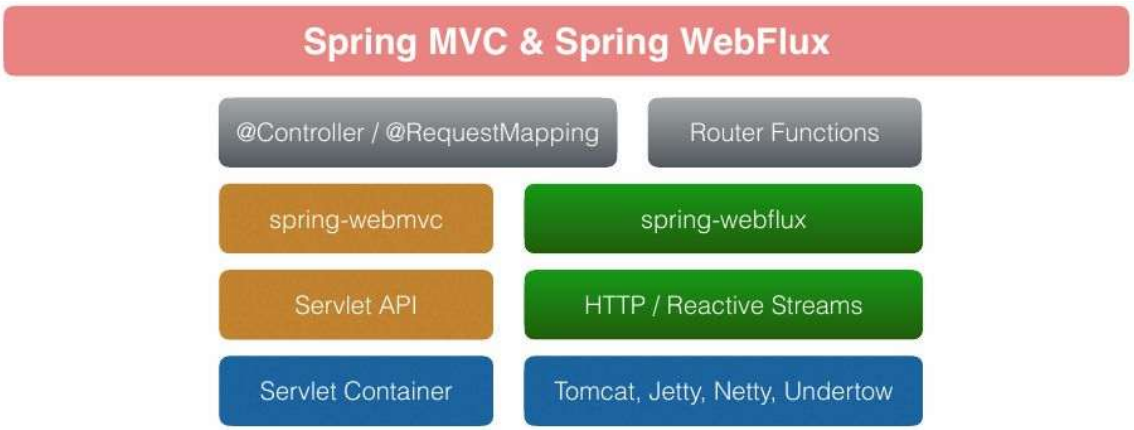
4.4. Modèle de Données



4.5. Diagramme de Déploiement (Docker-compose)



5. Annexes

5.1. Stack Spring Data JPA/MVC versus Spring Webflux

Webflux n'utilise pas de container de servlets. On utilise à la place l'API des Streams réactifs non bloquant. Le stack sous-jacent est donc différent.