Dossier d'architecture

Diffusion : OpenClassrooms – Mentor : Joeffrey Hernandez – Soutenance : Jordi Charpentier

Version	***Date***	***Auteur***	***Motif***
1	12/02/2025	Christophe Pierrès	Création du document
2			

Contenu ***Page***

1. Introduction 2

2. Choix et justification de l'architecture technique 2

3. Choix et justification des librairies..... 5

3.1. Librairies de test 6
3.1.1. Front-end testing 6
3.1.2. Back-end testing 7
3.2. Librairie de composants visuels 7

4. Choix et justification des paradigmes de programmation 8

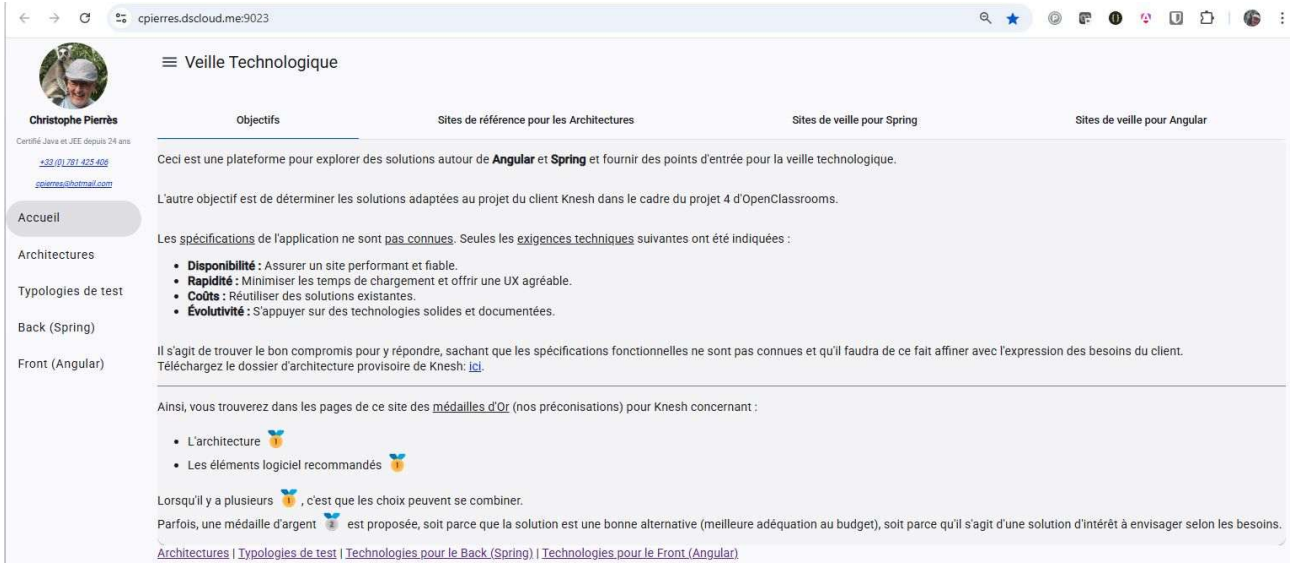
4.1. Frontend 10
4.2. Module backend de présentation (couche REST) 11
4.3. Module backend de persistance 11

5. Conclusion..... 12

1. Introduction

Ce dossier d'architecture répond aux exigences techniques du client Knesh. Il s'appuie sur notre site de veille technologique : https://cpierres.dscloud.me:9023/

La page d'accueil rappelle les besoins de Knesh et décrit la méthodologie :

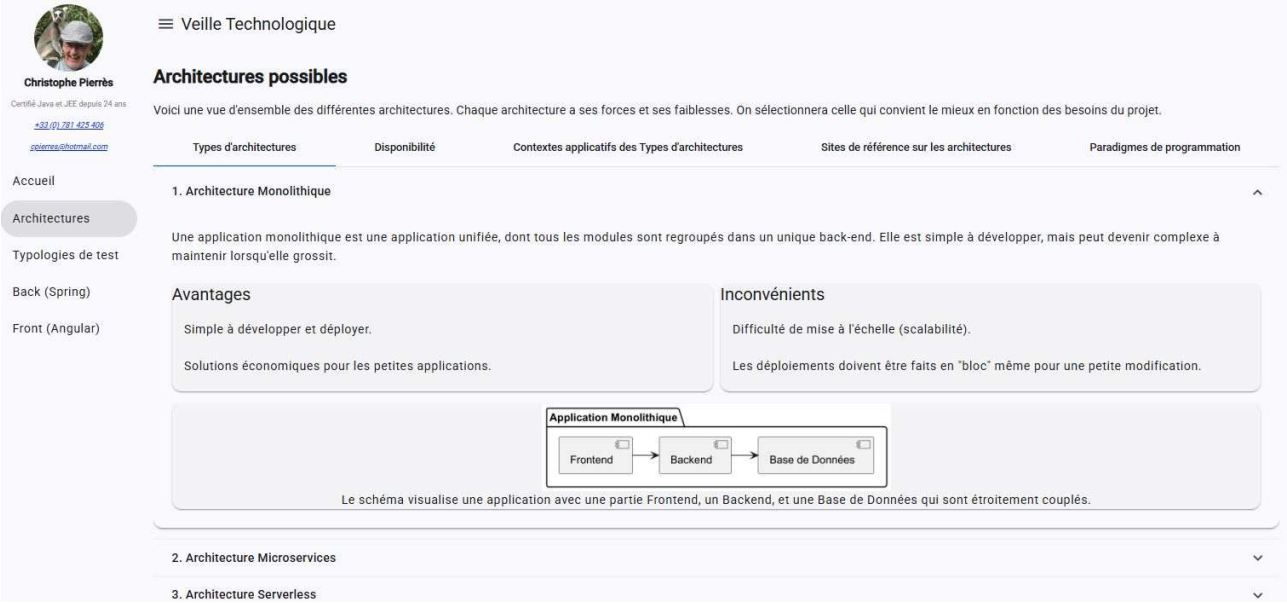


Nous proposons des choix mais le site peut permettre d'affiner selon les besoins précisés, si nécessaire.

2. Choix et justification de l'architecture technique

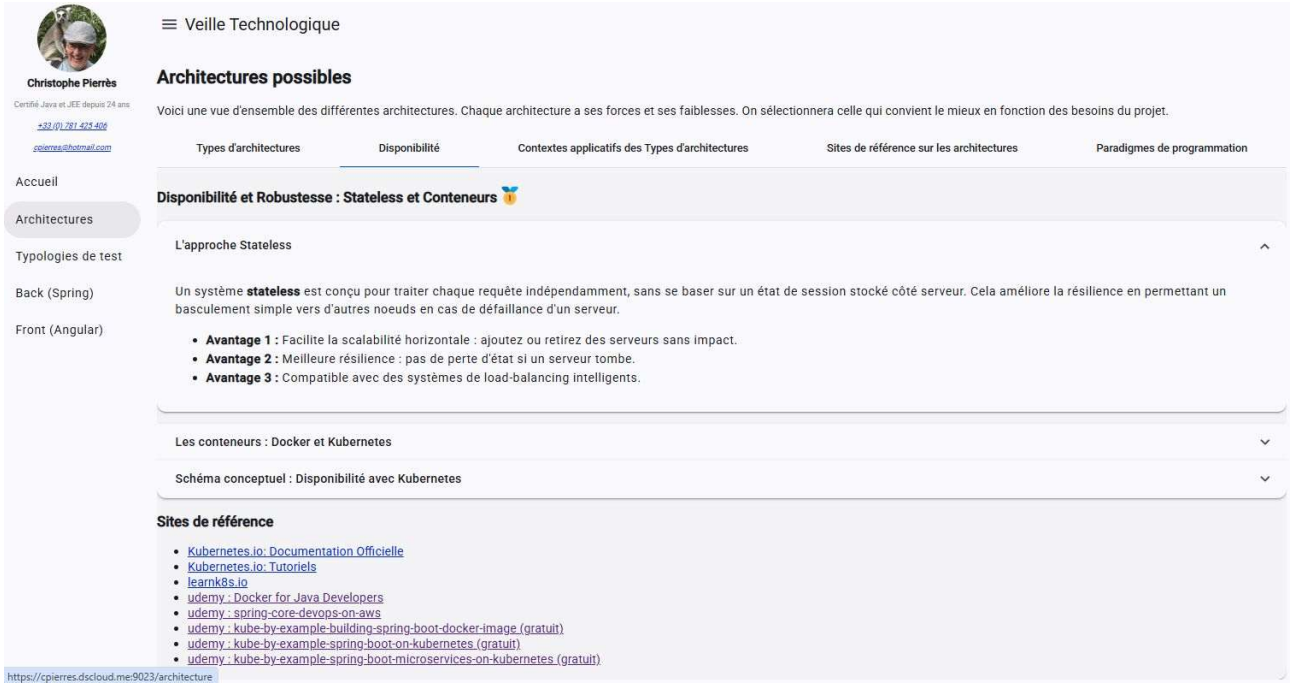
L'écran suivant décrit les types d'architecture : https://cpierres.dscloud.me:9023/architecture

<u>Cliquez sur chaque titre pour afficher une description synthétique avec les avantage et inconvénients</u> :

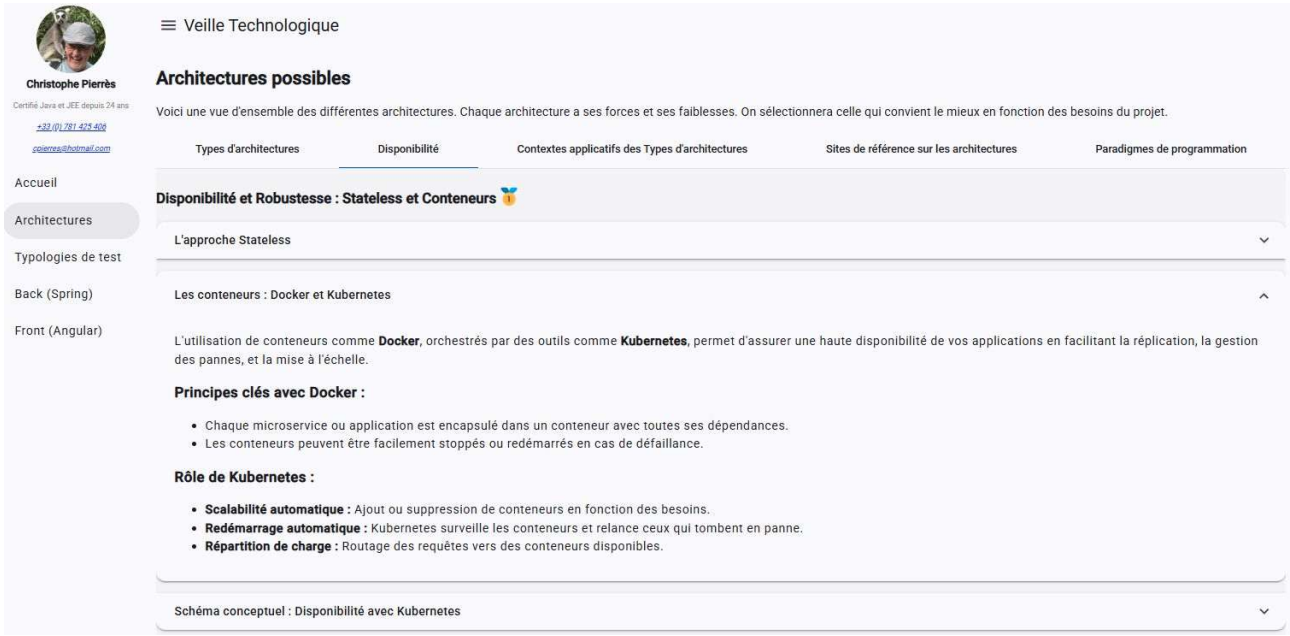


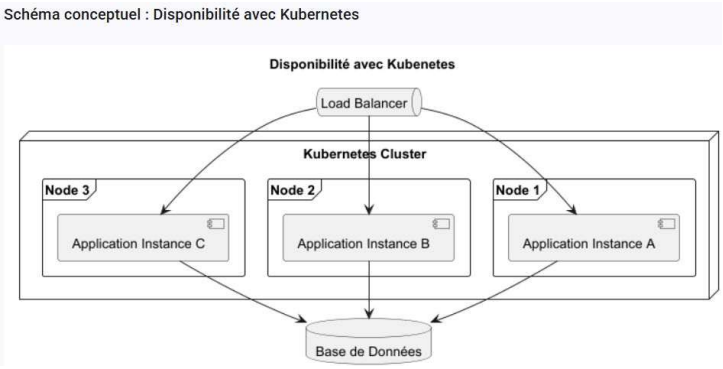
Nous vous conseillons de lire les différents types d'architecture afin de vérifier l'adéquation à vos besoins. D'après les exigences exprimées, la notion de **<u>coût</u>** est importante. **L'architecture monolithique** est retenue car **<u>simple à mettre en œuvre</u>**.

La **<u>disponibilité</u>** est un critère important. Le développement en architecture REST (Representational State Transfer) a l'énorme avantage d'être stateless, ce qui permet d'assurer la robustesse et la disponibilité :



Nous proposons d'adopter la mise en œuvre de Docker et **Kubernetes** afin d'assurer une disponibilité totale et une scalabilité horizontale. Les principes sont décrits ci-après :



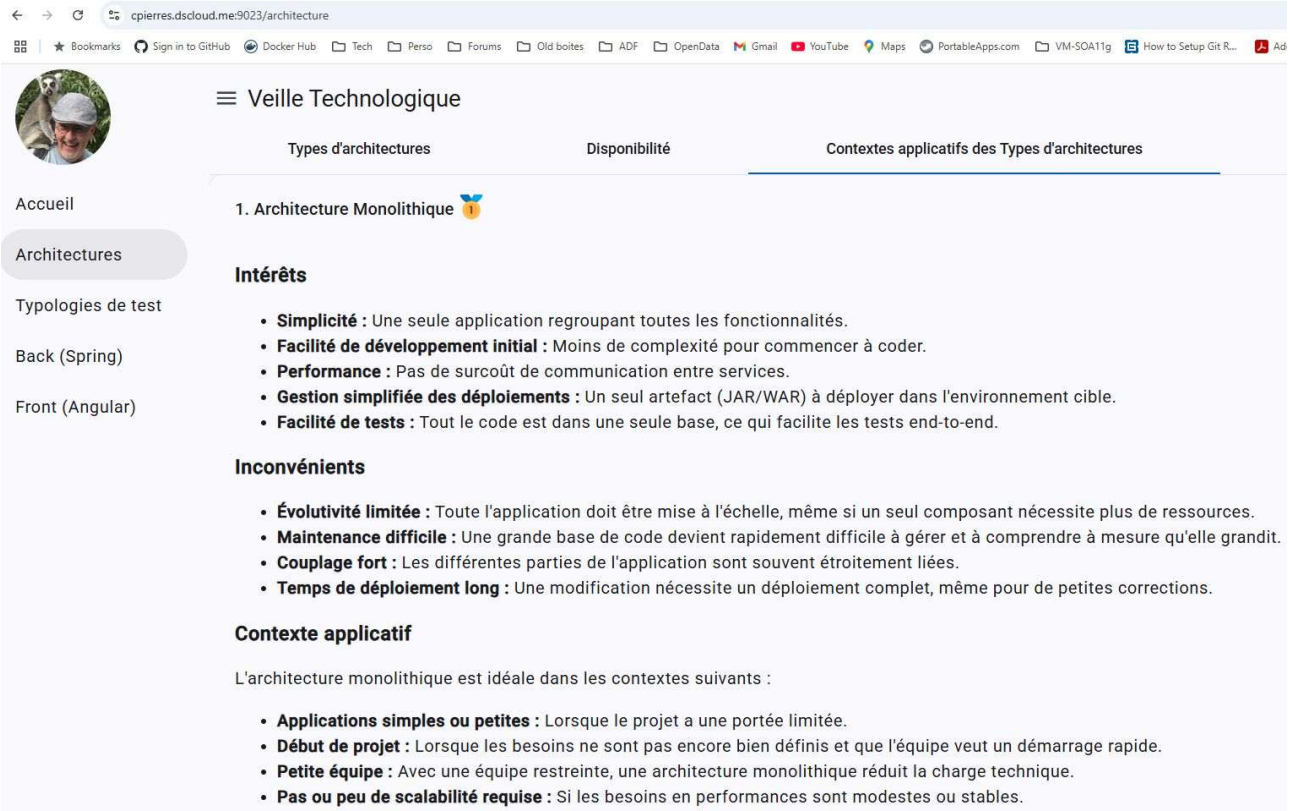


Pour confirmer que notre proposition est conforme à vos attentes, veuillez consulter les différentes sections de l'onglet **Contextes applicatifs** :

L'architecture principale sélectionnée est l'architecture **Monolithique**. Cette architecture peut intégrer les principes **Event-Driven** qui permettent d'effectuer des traitements asynchrones sur la base d'événements.

L'architecture Micro-services est potentiellement intéressante pour un gros développement mais plus complexe et coûteuse à mettre en œuvre. Il restera possible de migrer après coup vers cette architecture si nécessaire. Nous vous conseillons la lecture du contexte d'application de ces autres architectures.

Ci-dessous, la justification plus détaillée :



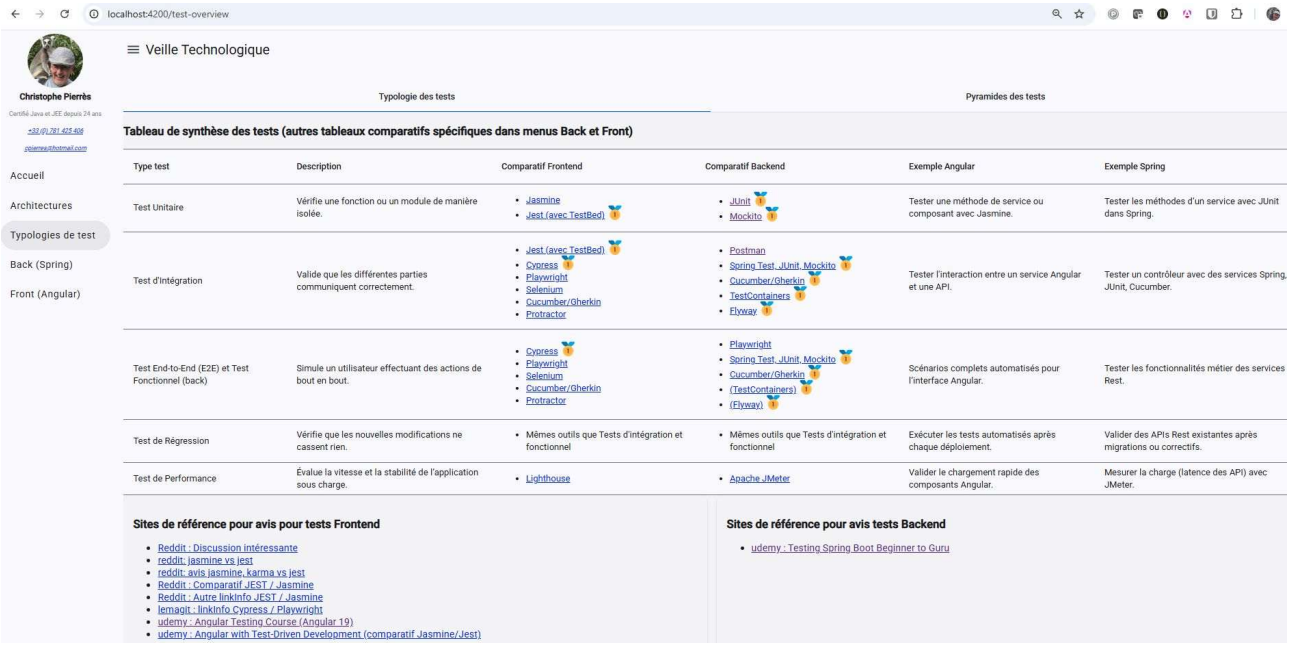
Grâce à l'approche **Kubernetes**, la disponibilité et la scalabilité peuvent être assurées. Veuillez consulter néanmoins les autres sections afin de comprendre les impacts budgétaires des autres possibilités.

3. Choix et justification des librairies

3.1. Librairies de test

https://cpierres.dscloud.me:9023/test-overview

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



localhost:4200/test-overview

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

Typologie des tests | Pyramides des tests

Tableau de synthèse des tests (autres tableaux comparatifs spécifiques dans menus Back et Front)

Type test	Description	Comparatif Frontend	Comparatif Backend	Exemple Angular	Exemple Spring
Test Unitaire	Vérifie une fonction ou un module de manière isolée.	• Jasmine • Jest (avec TestBed)	• JUnit • Mockito	Tester une méthode de service ou composant avec Jasmine.	Tester les méthodes d'un service avec JUnit dans Spring.
Test d'Intégration	Valide que les différentes parties communiquent correctement.	• Jest (avec TestBed) • Cypress • Playwright • Selenium • Cucumber/Gherkin • Protractor	• Postman • Spring Test, JUnit, Mockito • Cucumber/Gherkin • TestContainers • Flyway	Tester l'interaction entre un service Angular et une API.	Tester un contrôleur avec des services Spring, JUnit, Cucumber.
Test End-to-End (E2E) et Test Fonctionnel (back)	Simule un utilisateur effectuant des actions de bout en bout.	• Cypress • Playwright • Selenium • Cucumber/Gherkin • Protractor	• Playwright • Spring Test, JUnit, Mockito • Cucumber/Gherkin • (TestContainers) • (Flyway)	Scénarios complets automatisés pour l'interface Angular.	Tester les fonctionnalités métier des services Rest.
Test de Régression	Vérifie que les nouvelles modifications ne cassent rien.	• Mêmes outils que Tests d'intégration et fonctionnel	• Mêmes outils que Tests d'intégration et fonctionnel	Exécuter les tests automatisés après chaque déploiement.	Valider des APIs Rest existantes après migrations ou correctifs.
Test de Performance	Évalue la vitesse et la stabilité de l'application sous charge.	• Lighthouse	• Apache JMeter	Valider le chargement rapide des composants Angular.	Mesurer la charge (latence des API) avec JMeter.

Sites de référence pour avis pour tests Frontend

- 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)

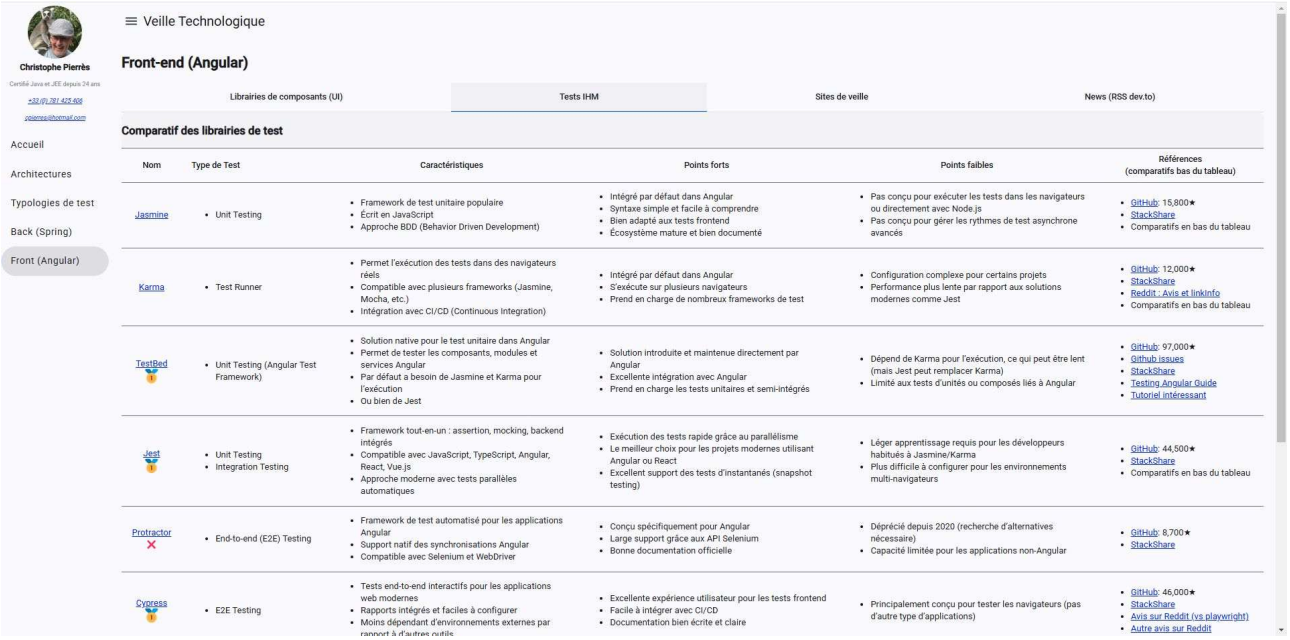
Sites de référence pour avis tests Backend

- udemy : Testing Spring Boot Beginner to Guru

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

3.1.1. Front-end testing

Pour une analyse détaillée des choix de librairies, consultez le lien : https://cpierres.dscloud.me:9023/front sur l'onglet **Tests IHM** :



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 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

rapport à d'autres outils

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.

3.1.2. Back-end testing

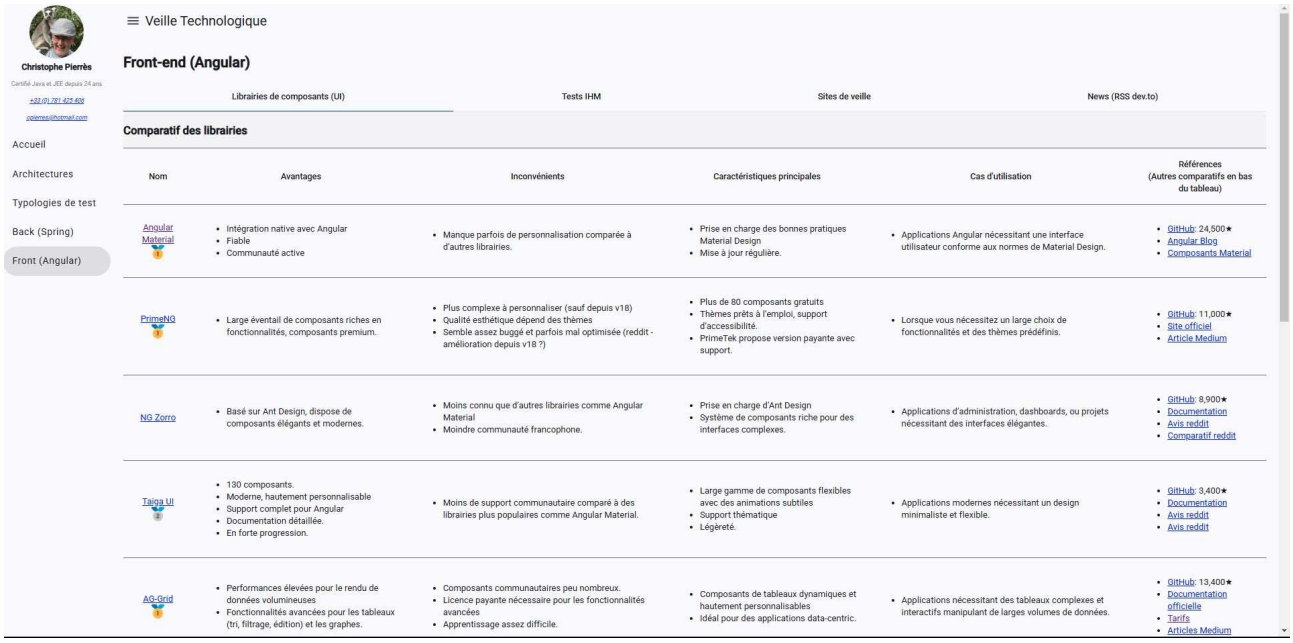
Pour Spring, l'intégration des capacités de test est particulièrement bien soignée et les propositions par défaut sont excellentes. Spring permet d'intégrer avec facilité des briques optionnelles que nous avons retenues (TestContainers). Le tableau vu précédemment au début de cette section confirme les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

3.2. Librairie de composants visuels

Une analyse des librairies de composants est disponible sur : https://cpierres.dscloud.me:9023/front.

La solution par défaut incluse dans Angular est : **Angular Materials** 🥇 . Nous préconisons cette solution très populaire car robuste et intéressante. Angular prévoie de faire évoluer les librairies sous-jacentes à Angular Materials en restant rétro-compatible. Ceci dit, les composants de cette librairie ne couvrent pas tous les besoins potentiels (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 nouvelle et prometteuse librairie **Taigai UI** :



Christophe Pierrès
Certifié Java et JEE depuis 24 ans
+33 (0) 781 425 409
cpierres@hotmail.com

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

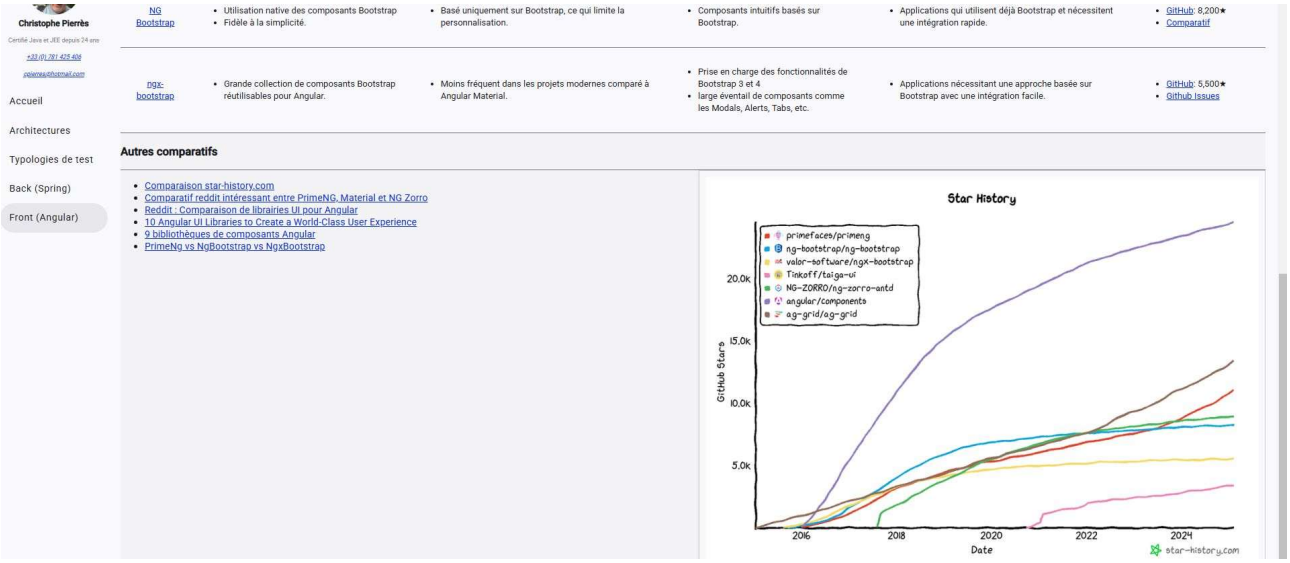
≡ Veille Technologique

Front-end (Angular)

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

Comparatif des librairies

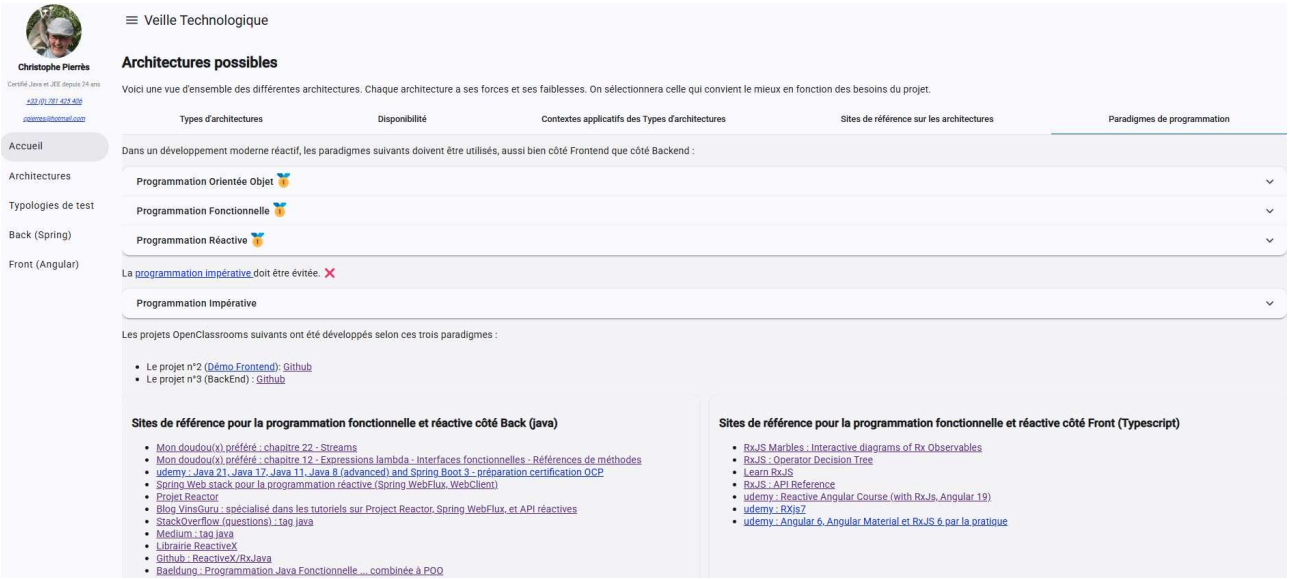
Nom	Avantages	Inconvénients	Caractéristiques principales	Cas d'utilisation	Références (Autres comparatifs en bas du tableau)
Angular Material 🥇	• Intégration native avec Angular • Fiable • Communauté active	• Manque parfois de personnalisation comparée à d'autres librairies.	• Prise en charge des bonnes 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



4. *Choix et justification des paradigmes de programmation*

Le choix des paradigmes de programmation a été placé sur le lien https://cpierres.dscloud.me:9023/architecture , onglet **Paradigmes de programmation**.

Ces choix induisent des choix pour le frontend et surtout le backend que nous aborderons ensuite.



<u>Vous pouvez cliquer sur chaque section pour la justification</u> :

Types d'architectures | Disponibilité | Contextes applicatifs des Types d'architectures | Paradigmes de programmation

Dans un développement moderne réactif, les paradigmes suivants doivent être utilisés, aussi bien côté Frontend que côté Backend :

Programmation Orientée Objet

La programmation orientée objet (POO) est basée sur l'organisation du code autour des objets, qui contiennent des données (« propriétés ») et des comportements (« méthodes »). Elle repose sur quatre concepts clés :

- **Encapsulation** : Protéger les données et fournir des accès via des méthodes.
- **Héritage** : Réutiliser ou spécialiser des comportements en dérivant des classes.
- **Polymorphisme** : Capacité à traiter différents objets au travers d'une interface commune.
- **Abstraction** : Cacher les détails de l'implémentation pour se focaliser sur les fonctionnalités.

En Angular et Spring, la POO est prépondérante, utilisée sous forme de composants, services, et beans (en Spring).

Programmation Fonctionnelle

Programmation Réactive

La programmation impérative doit être éviter sauf pour des cas simples. ❌

Programmation Impérative

Programmation fonctionnelle :

Types d'architectures | Disponibilité | Contextes applicatifs des Types d'architectures | Paradigmes de programmation

Dans un développement moderne réactif, les paradigmes suivants doivent être utilisés, aussi bien côté Frontend que côté Backend :

Programmation Orientée Objet

Programmation Fonctionnelle

La programmation fonctionnelle repose sur des fonctions pures pour manipuler des données. Elle privilégie la **déclarativité**, l'**immutabilité**, ainsi que des fonctions d'ordre supérieur.

Grâce aux **Streams** (comme en Java ou RxJS dans Angular), ce paradigme devient extrêmement concis et efficace :

- **Concision :** Les Streams réduisent considérablement le code requis par rapport à des boucles impératives.
- **Efficacité :** Approche basée sur l'exécution paresseuse (lazy evaluation), ne calculant que ce qui est nécessaire.
- **Chaînage :** Les opérations comme `map`, `filter`, `reduce` clarifient la logique.

Utilisation typique avec Angular et Spring :

- En Angular, RxJS permet un chaînage clair des actions sur des Observables, parfait pour traiter des données dynamiques.
- En Spring, depuis Java 8, les Streams permettent de filtrer et transformer les collections de manière concise et performante.

Programmation Réactive

La programmation impérative doit être éviter sauf pour des cas simples. ❌

Programmation réactive :

Types d'architectures | Disponibilité | Contextes applicatifs des Types d'architectures | Paradigmes de programmation

Dans un développement moderne réactif, les paradigmes suivants doivent être utilisés, aussi bien côté Frontend que côté Backend :

Programmation Orientée Objet

Programmation Fonctionnelle

Programmation Réactive

La programmation réactive met l'accent sur la gestion des flux de données **asynchrones**. Ce paradigme est particulièrement utile pour traiter des événements, ou des flux en temps réel.

- Les Observables (côté Angular via RxJS) et les Publishers/Flux (côté Spring via Reactor) permettent de manipuler et transformer des flux dynamiques de manière similaire aux Streams, mais avec une gestion automatique de l'asynchronisme.
- **Exemple pratique Angular :** Utiliser RxJS pour filtrer, regrouper ou transformer des événements utilisateurs avant de les afficher.
- **Exemple pratique Spring :** Spring WebFlux propose une gestion réactive des requêtes et des réponses HTTP.

La combinaison des paradigmes fonctionnels et réactifs est particulièrement puissante pour optimiser les performances des applications modernes.

La programmation impérative doit être éviter sauf pour des cas simples. ❌

Programmation Impérative

La programmation impérative doit être évitée comme expliqué ci-après :

Types d'architectures | Disponibilité | Contextes applicatifs des Types d'architectures | Paradigmes de programmation

Dans un développement moderne réactif, les paradigmes suivants doivent être utilisés, aussi bien côté Frontend que côté Backend :

Programmation Orientée Objet

Programmation Fonctionnelle

Programmation Réactive

La programmation impérative doit être éviter sauf pour des cas simples. ❌

Programmation Impérative

La programmation impérative repose sur des instructions et des structures de contrôle comme les boucles, conditions, etc. Elle est souvent inefficace et difficile à maintenir dans un contexte moderne :

- **Moins concise :** Nécessite souvent plus de lignes de code en manipulant directement les structures de données.
- **Mutable :** Conduit souvent à des erreurs dues à la modification de l'état des variables.

Toutefois, elle peut être utile pour des scripts ou des tâches simples mais doit être évitée dans des applications professionnelles modernes.

Les projets OpenClassrooms suivants ont été développés selon ces trois paradigmes :

- Le projet OpenClassrooms n°2 (Démo Frontend): Github
- Le projet OpenClassrooms n°3 (BackEnd) : Github

Les projets OpenClassrooms suivants ont été développés selon ces trois paradigmes :

- projet n°2 (Démo Frontend): Github
- projet n°3 (BackEnd) : Github

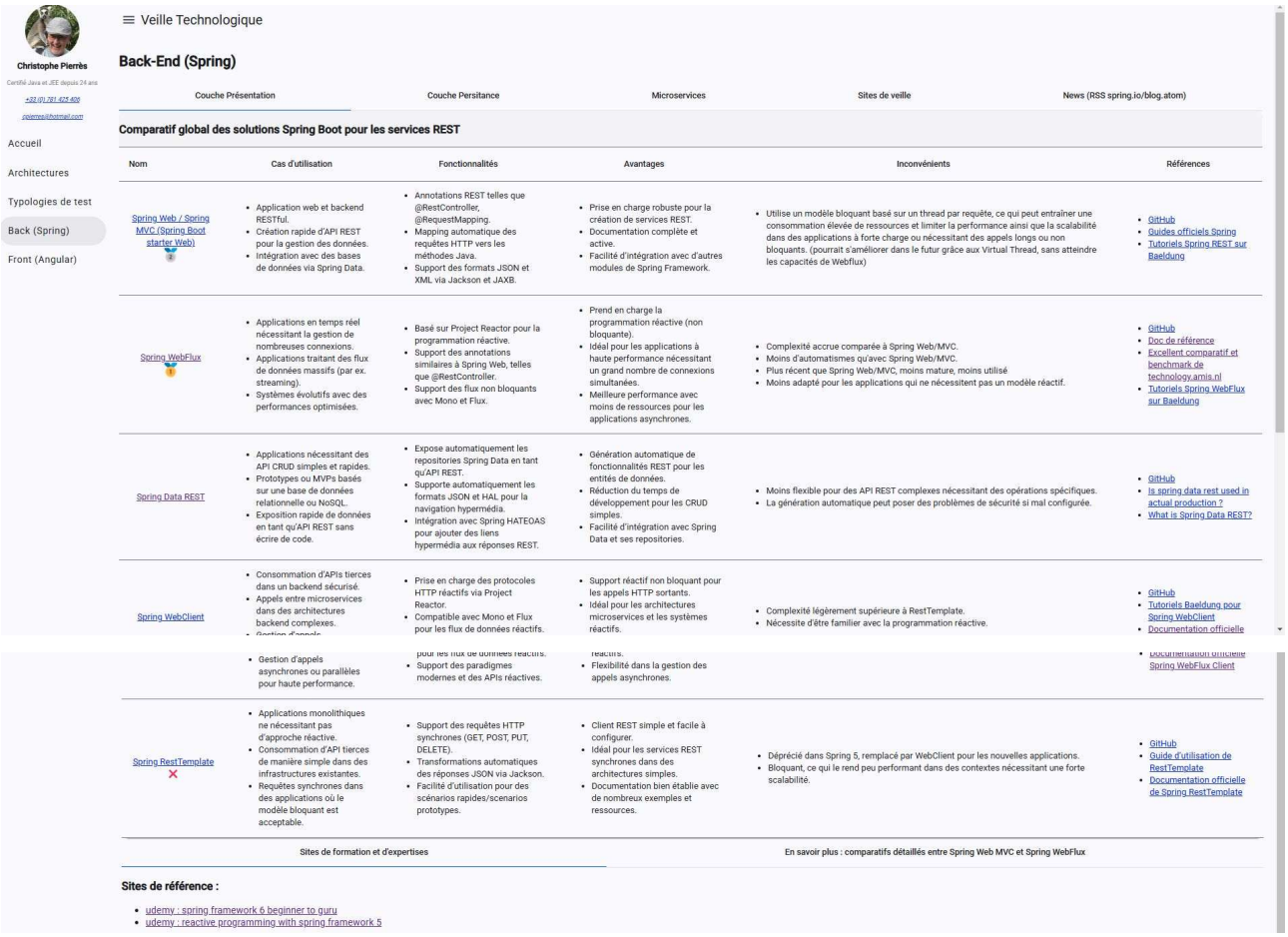
4.1. Frontend

De fait, Angular grâce à Javascript/Typescript, RxJS, les signaux permet d'appliquer la programmation objet, fonctionnelle et réactive.

Concernant les modules backend et base de données, il y a des choix permettant d'appliquer les principes réactifs d'une manière plus ou moins poussée.

4.2. Module backend de présentation (couche REST)

Spring Webflux associé à une base de données **R2DBC** permettent d'obtenir un système full réactif de bout en bout, et sans limite, au prix d'un développement légèrement plus complexe, donc plus coûteux.



Veille Technologique

Christophe Pierrès
Certifié Java et JEE depuis 24 ans
+33 (0) 781 425 409
cpierres@hotmail.com

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

Back-End (Spring)

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

Comparatif global des solutions Spring Boot pour les services REST

Nom	Cas d'utilisation	Fonctionnalités	Avantages	Inconvénients	Références
Spring Web / Spring MVC (Spring Boot starter Web)	• Application web et backend RESTful. • Création rapide d'API REST pour la gestion des données. • Intégration avec des bases de données via Spring Data.	• Annotations REST telles que @RestController, @RequestMapping. • Mapping automatique des requêtes HTTP vers les méthodes Java. • Support des formats JSON et XML via Jackson et JAXB.	• Prise en charge robuste pour la création de services REST. • Documentation complète et active. • Facilité d'intégration avec d'autres modules de Spring Framework.	• Utilise un modèle bloquant basé sur un thread par requête, ce qui peut entraîner une consommation élevée de ressources et limiter la performance ainsi que la scalabilité dans des applications à forte charge ou nécessitant des appels longs ou non bloquants. (pourrait s'améliorer dans le futur grâce aux Virtual Thread, sans atteindre les capacités de Webflux)	• GitHub • Guides officiels Spring • Tutoriels Spring REST sur Baeldung
Spring WebFlux	• Applications en temps réel nécessitant la gestion de nombreuses connexions. • Applications traitant des flux de données massifs (par ex. streaming). • Systèmes évolutifs avec des performances optimisées.	• Basé sur Project Reactor pour la programmation réactive. • Support des annotations similaires à Spring Web, telles que @RestController. • Support des flux non bloquants avec Mono et Flux.	• Prend en charge la programmation réactive (non bloquante). • Idéal pour les applications à haute performance nécessitant un grand nombre de connexions simultanées. • Meilleure performance avec moins de ressources pour les applications asynchrones.	• Complexité accrue comparée à Spring Web/MVC. • Moins d'automatismes qu'avec Spring Web/MVC. • Plus récent que Spring Web/MVC, moins mature, moins utilisé • Moins adapté pour les applications qui ne nécessitent pas un modèle réactif.	• GitHub • Doc de référence • Excellent comparatif et benchmark de technology.amis.nl • Tutoriels Spring WebFlux sur Baeldung
Spring Data REST	• Applications nécessitant des API CRUD simples et rapides. • Prototypes ou MVPs basés sur une base de données relationnelle ou NoSQL. • Exposition rapide de données en tant qu'API REST sans écrire de code.	• Expose automatiquement les repositories Spring Data en tant qu'API REST. • Supporte automatiquement les formats JSON et HAL pour la navigation hypermédia. • Intégration avec Spring HATEOAS pour ajouter des liens hypermédia aux réponses REST.	• Génération automatique de fonctionnalités REST pour les entités de données. • Réduction du temps de développement pour les CRUD simples. • Facilité d'intégration avec Spring Data et ses repositories.	• Moins flexible pour des API REST complexes nécessitant des opérations spécifiques. • La génération automatique peut poser des problèmes de sécurité si mal configurée.	• GitHub • Is spring data rest used in actual production ? • What is Spring Data REST?
Spring WebClient	• Consommation d'APIs tierces dans un backend sécurisé. • Appels entre microservices dans des architectures backend complexes. • Gestion d'appels asynchrones ou parallèles pour haute performance.	• Prise en charge des protocoles HTTP réactifs via Project Reactor. • Compatible avec Mono et Flux pour les flux de données réactifs. • Support des paradigmes modernes et des APIs réactives.	• Support réactif non bloquant pour les appels HTTP sortants. • Idéal pour les architectures microservices et les systèmes réactifs. • Flexibilité dans la gestion des appels asynchrones.	• Complexité légèrement supérieure à RestTemplate. • Nécessite d'être familier avec la programmation réactive.	• GitHub • Tutoriels Baeldung pour Spring WebClient • Documentation officielle Spring WebFlux Client
Spring RestTemplate ✕	• Applications monolithiques ne nécessitant pas d'approche réactive. • Consommation d'API tierces de manière simple dans des infrastructures existantes. • Requêtes synchrones dans des applications où le modèle bloquant est acceptable.	• Support des requêtes HTTP synchrones (GET, POST, PUT, DELETE). • Transformations automatiques des réponses JSON via Jackson. • Facilité d'utilisation pour des scénarios rapides/scenarios prototypes.	• Client REST simple et facile à configurer. • Idéal pour les services REST synchrones dans des architectures simples. • Documentation bien établie avec de nombreux exemples et ressources.	• Déprécié dans Spring 5, remplacé par WebClient pour les nouvelles applications. • Bloquant, ce qui le rend peu performant dans des contextes nécessitant une forte scalabilité.	• GitHub • Guide d'utilisation de RestTemplate • Documentation officielle de Spring RestTemplate

Sites de formation et d'expertises | En savoir plus : comparatifs détaillés entre Spring Web MVC et Spring WebFlux

Sites de référence :

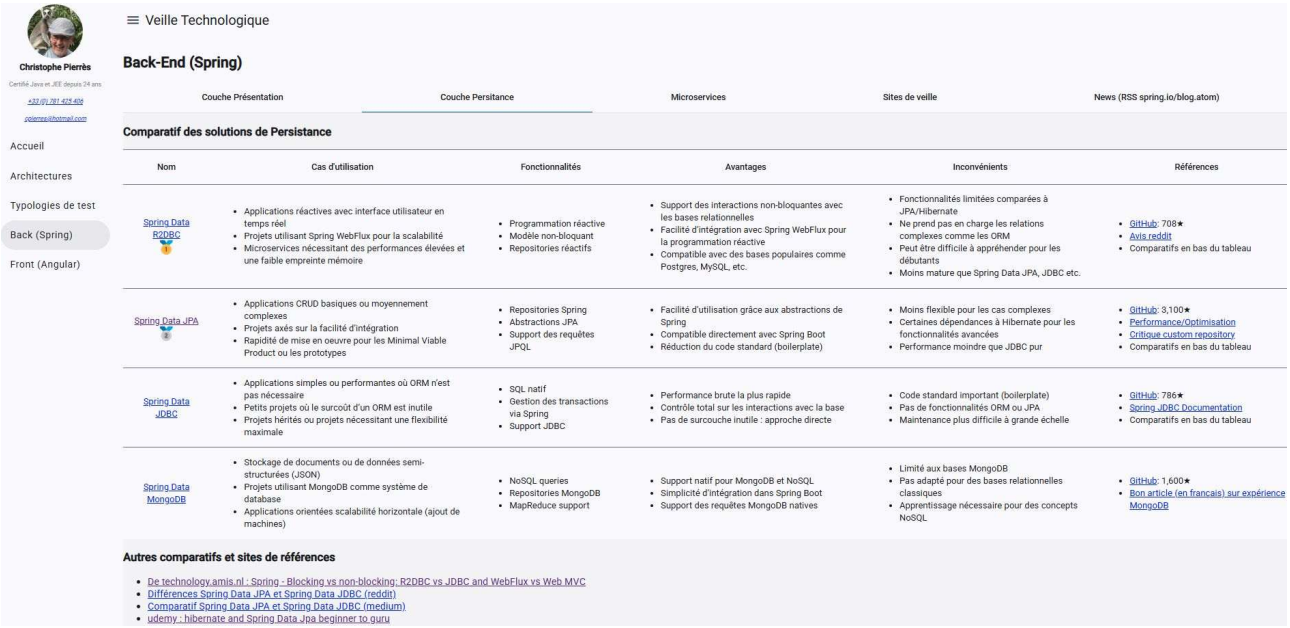
- udemy : spring framework 6 beginner to guru
- udemy : reactive programming with spring framework 5

Le second choix serait **Spring Web MVC**, plus simple en développement. Il correspond au modèle bloquant des Servlets (1 thread par requête) lequel monte moins en charge. Cet inconvénient pourrait être atténué par le déploiement **Kubernetes** qui permet de répartir la charge d'un modèle REST Stateless.

4.3. Module backend de persistance

Si **Spring Webflux** est retenu, il faut choisir une base de données de type **R2DBC** qui est elle-même réactive : Postgresql est le choix suggéré. Si **Spring Web MVC** est retenu, alors notre choix va vers **Spring Data JPA**.

Les justifications se trouvent dans le menu **Back (Spring)** puis onglet **Couche Persistance** :



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 | News (RSS spring.io/blog.atom)

Comparatif des solutions de Persistance

Nom	Cas d'utilisation	Fonctionnalités	Avantages	Inconvénients	Références
Spring Data R2DBC	• Applications réactives avec interface utilisateur en temps réel • Projets utilisant Spring WebFlux pour la scalabilité • Microservices nécessitant des performances élevées et une faible empreinte mémoire	• Programmation réactive • Modèle non-bloquant • Repositories réactifs	• Support des interactions non-bloquantes avec les bases relationnelles • Facilité d'intégration avec Spring WebFlux pour la programmation réactive • Compatible avec des bases populaires comme Postgres, MySQL, etc.	• Fonctionnalités limitées comparées à JPA/Hibernate • Ne prend pas en charge les relations complexes comme les ORM • Peut être difficile à appréhender pour les débutants • Moins mature que Spring Data JPA, JDBC etc.	• GitHub: 708★ • Avis reddit • Comparatifs en bas du tableau
Spring Data JPA	• Applications CRUD basiques ou moyennement complexes • Projets axés sur la facilité d'intégration • Rapidité de mise en oeuvre pour les Minimal Viable Product ou les prototypes	• Repositories Spring • Abstractions JPA • Support des requêtes JPQL	• Facilité d'utilisation grâce aux abstractions de Spring • Compatible directement avec Spring Boot • Réduction du code standard (boilerplate)	• Moins flexible pour les cas complexes • Certaines dépendances à Hibernate pour les fonctionnalités avancées • Performance moindre que JDBC pur	• GitHub: 3,100★ • Performance/Optimisation • Critique custom repository • Comparatifs en bas du tableau
Spring Data JDBC	• Applications simples ou performantes où ORM n'est pas nécessaire • Petits projets où le surcoût d'un ORM est inutile • Projets hérités ou projets nécessitant une flexibilité maximale	• SQL natif • Gestion des transactions via Spring • Support JDBC	• Performance brute la plus rapide • Contrôle total sur les interactions avec la base • Pas de surcouche inutile : approche directe	• Code standard important (boilerplate) • Pas de fonctionnalités ORM ou JPA • Maintenance plus difficile à grande échelle	• GitHub: 786★ • Spring JDBC Documentation • Comparatifs en bas du tableau
Spring Data MongoDB	• Stockage de documents ou de données semi-structurées (JSON) • Projets utilisant MongoDB comme système de database • Applications orientées scalabilité horizontale (ajout de machines)	• NoSQL queries • Repositories MongoDB • MapReduce support	• Support natif pour MongoDB et NoSQL • Simplicité d'intégration dans Spring Boot • Support des requêtes MongoDB natives	• Limité aux bases MongoDB • Pas adapté pour des bases relationnelles classiques • Apprentissage nécessaire pour des concepts NoSQL	• GitHub: 1,600★ • Bon article (en français) sur expérience MongoDB

Autres comparatifs et sites de références

- De technology.amis.nl : Spring - Blocking vs non-blocking: R2DBC vs JDBC and WebFlux vs Web MVC
- Différences Spring Data JPA et Spring Data JDBC (reddit)
- Comparatif Spring Data JPA et Spring Data JDBC (medium)
- udemy : hibernate and Spring Data Jpa beginner to guru

5. Conclusion

En espérant que ce site de veille vous a permis d'avoir les idées claires sur les possibilités d'architecture et les choix retenus, adaptés à vos besoins et votre budget.

N'hésitez pas à consulter les sites de références qui sont les sources de notre analyse (au minimum 3, voire plus !). Il y a également, dans chaque section, des flux RSS permettant d'avoir les dernières actualités sur chaque thème.