# Contraintes techniques d'implémentation (Back-End)

Projet MMIPlatform par Glenn Guillard, Alexandre Guedes, Thomas Henry, Jerôme Fabre et Morgane Le Normand

## Contexte

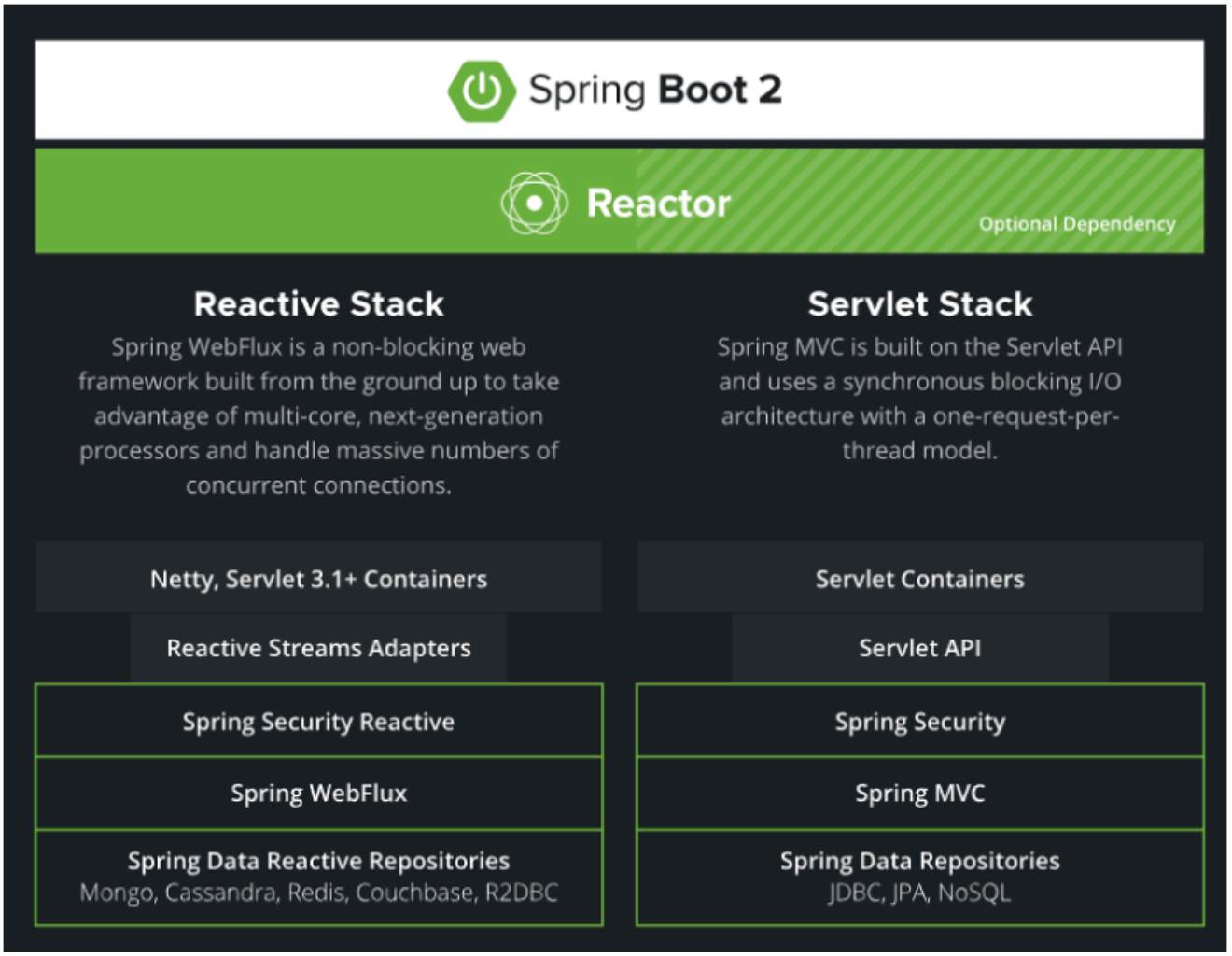
L’IUT de Meaux ne dispose actuellement d'aucune application pour gérer efficacement les notes, les absences et la visualisation de l’établissement. En réponse à ce besoin, le projet **MMIPlatform** a été proposé par notre équipe. Cette application vise à centraliser et simplifier la gestion académique et administrative grâce à des outils modernes, notamment une vue interactive 3D des salles et une gestion avancée des matrices Excel.

## 1. Contrainte technologique

Pour ce projet, du fait que nous travaillons en microservices et que nous développons une solution d'export de fichiers massifs, nous avons dû changer le framework proposé, qui était Symfony, pour utiliser **Spring en Java**.

Pour l'utilisation de Java, nous avons choisi de partir sur une **architecture hexagonale**, capable de rendre le code Java scalable et maintenable en séparant infrastructure, domaine métier et composants applicatifs.

Pour notre solution d'export Excel et pour d'autres requêtes demandant un temps de résolution important, nous avons combiné deux modes de fonctionnement de Spring. Nous avons mélangé la **Servlet stack** et la **Reactive stack** au sein de l'application.




L'intérêt étant que, pour tout ce qui concerne les requêtes simples d'API, comme l'authentification, la création d'utilisateurs et les actions CRUD liées aux notes, nous utilisions la **Servlet stack**. Cette dernière permet d'obtenir un comportement fluide pour ce type d'action grâce à un composant synchrone bloquant : si une partie de la requête n'est pas raccord, elle bloque tout le processus. Avec un modèle une requête par thread, le client doit attendre la validation de sa requête pour effectuer d'autres actions (mode synchrone bloquant).

Mais pour l'export Excel, c'est différent : le client n'a pas besoin de suivre en continu la création de son Excel. Il attend seulement le téléchargement du fichier généré par du code Java. C'est pour cela que nous utilisons la **Reactive stack**, qui permet de créer des actions web non bloquantes et de garantir la livraison d'une réponse en mode asynchrone, avec la possibilité d'effectuer du multitâche intégré à un Mono ou un Flux. Grâce à un système de cache avancé, l'ouverture d'un flux mobilise des ressources encapsulées dans le cache, ce qui permet de ne pas perturber l'expérience côté client.

## Différence entre Mono et Flux

- **Mono** : C'est comme une commande en boulangerie. Une fois prête, le boulanger donne la commande complète au client en un seul élément. Cela correspond à une promesse : "Je te donne ta commande quand elle est prête."
  Un Mono représente un unique objet ou rien du tout (0 ou 1).

- **Flux** : C'est comme un client qui commande un paquet de cookies. Dès que le premier cookie est prêt, on le lui donne immédiatement, puis les autres suivent au fur et à mesure qu'ils sortent du four. Cela correspond à une promesse : "Je te donne les cookies un par un dès qu'ils sont prêts."
  Un Flux représente une suite d'éléments (1 ou n).

---

## Stockage des données

Pour stocker les données relatives aux notes, aux étudiants, etc., nous avons créé un modèle de données relationnelles robuste, ce qui nous a conduits à utiliser des bases de données relationnelles.

Bases de données utilisées :

1. MySQL
2. Oracle SQL
3. PostgreSQL
4. H2 in-memory pour les tests en local (elle se construit et se détruit en fonction de l'état de lancement de l'application)

## Persistance des données

Pour gérer la persistance de nos données en Spring, nous utilisons plusieurs composants et librairies :

- Hibernate
- Jakarta EE
- Spring JPA
- WebClient DataBuffer

## Génération d'Excels

Pour gérer la génération de fichiers Excel, nous avons utilisé :

- **Apache POI for WebFlux**

## Sécurité

Pour garantir la sécurité de l'application, nous utilisons plusieurs librairies :

- **Spring Security** (gestion des accès aux endpoints et FilterChain du JWT)
- **JsonWebToken.io** (gestion et génération des JWT en accord avec les règles de Spring Security)

- **BouncyCastle**

### Conteneurisation

Nous utilisons Docker avec la configuration suivante :

- Maven 3.8.8
- OpenJDK 21
- Un conteneur Java Spring
- Un conteneur SQL lié à un volume MySQL
- Une network faisant office de load balancer

### Outils divers

- **Lombok** : pour la génération automatique des getters et setters ainsi que l'injection des dépendances sans constructeur
- **netty-resolver-dns-native-macos** : pour gérer les adressages DNS pour les utilisateurs macOS
- **Tests unitaires** : Spring Boot Starter Test (embarquant Mockito et JUnit)

---

## 2. Compatibilité et interopérabilité

- **Environnements cibles** : L'application doit fonctionner sous des environnements spécifiques (Windows, Linux, macOS, Cloud, etc.).
- **Interopérabilité** : L'application doit pouvoir s’intégrer avec d’autres systèmes ou respecter des API existantes.

---

## 3. Performance et scalabilité

- **Exigences de performance** : Le système doit répondre dans un délai inférieur à X millisecondes pour une requête donnée.
- **Gestion de charge** : Le système doit pouvoir gérer un certain nombre de requêtes simultanées (scalabilité horizontale et verticale).
- **Temps de disponibilité** : Le système doit garantir un taux de disponibilité de 99,9 % (SLA).

---

## 4. Contraintes liées à la sécurité

- **Chiffrement** : Toutes les communications entre le client et le serveur doivent être sécurisées via HTTPS (TLS).
- **Authentification** : Mise en place d’une authentification sécurisée via JWT.
- **Confidentialité des données** : Conformité avec les réglementations comme le RGPD (protection des données des utilisateurs).

---

## 5. Contraintes de développement

- **Méthodologie** : Utilisation d’une méthodologie agile (Scrum, Kanban).
- **Normes de codage** : Respect des règles de qualité de code, comme les normes imposées par SonarQube.
- **Tests** : Mise en place d’un cadre de tests automatisés (JUnit, Mockito) pour garantir la qualité logicielle.
- **Normes d'architecture** : Respect des normes d'architecture dans le développement de l'application (architecture hexagonale).

---

## 6. Contraintes liées au déploiement

- **Infrastructure** : L’application doit être déployée dans un environnement cloud (AWS, Azure, GCP) ou sur des serveurs on-premise.
- **Conteneurisation** : Utilisation obligatoire de conteneurs Docker avec orchestration Kubernetes.
- **CI/CD** : Mise en place d’une pipeline CI/CD pour automatiser les déploiements (GitHub Actions, Jenkins, GitLab CI).

---

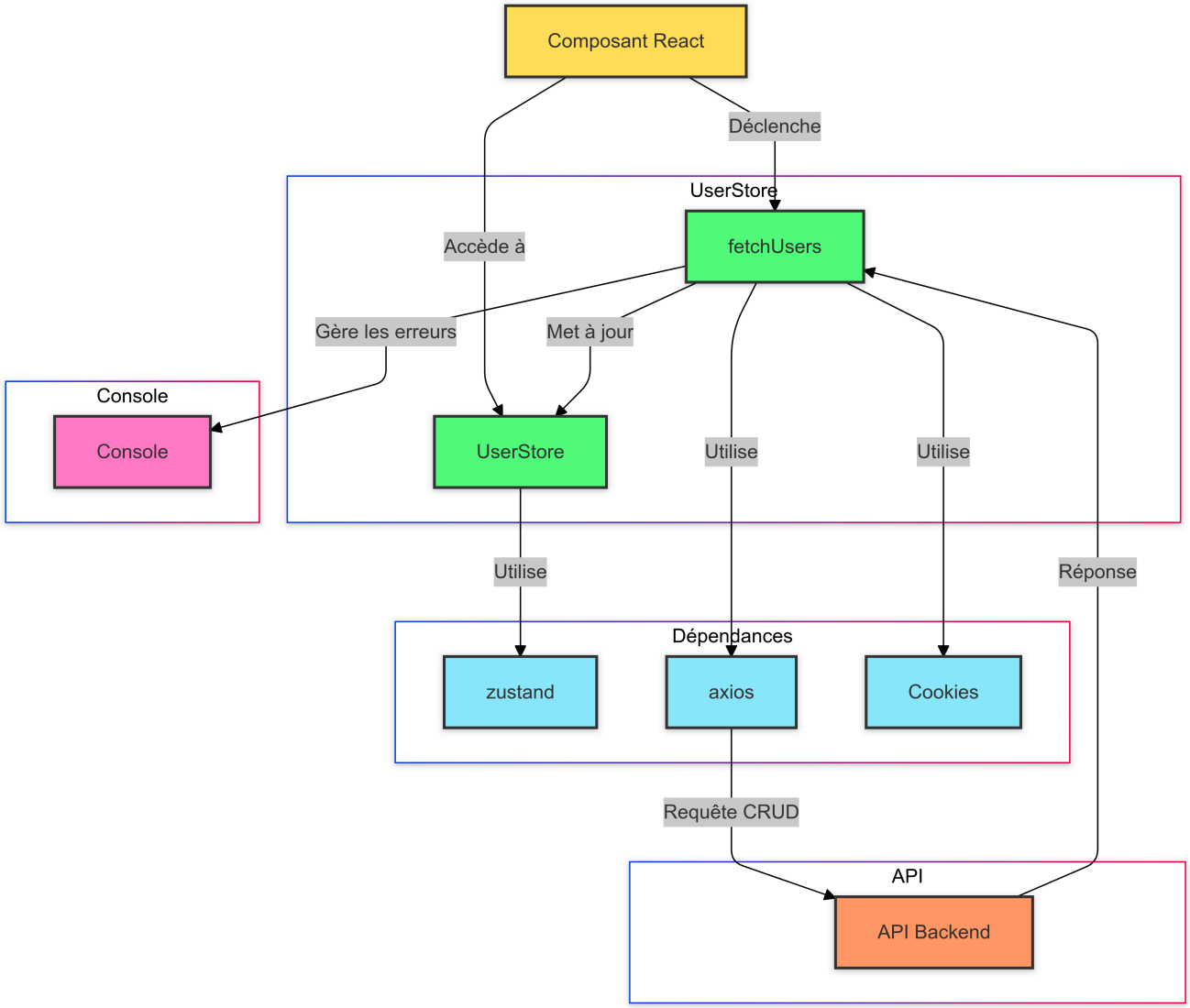
## 7. Limites imposées par le matériel ou l’environnement

- **Ressources matérielles** : L’application doit fonctionner sur des machines avec X Go de RAM et Y processeurs.
- **Connexion réseau** : Optimisation pour des environnements avec des connexions réseau limitées (bande passante réduite).

---

## 8. Communication avec le back-end depuis le client

Voici un exemple d'architecture dans un usage type comme la récupération des users




Nous avions comme contrainte d'utiliser du React pour notre application client.

Pour gérer l'état de nos données dans l'entièreté des parcours clients nous utilisons différentes librairies :

- **Zustand**
- **JsCookies**
- **Axios**

## Qu’est-ce que Zustand ?

**Zustand** est une librairie légère de gestion d’état pour les applications JavaScript et React. Contrairement à des solutions plus lourdes comme Redux, Zustand est conçue pour être simple et intuitive tout en restant suffisamment puissante pour gérer des cas complexes. Elle utilise une approche basée sur des hooks (crochets) et offre une API minimaliste. Nous permettant de gérer la state des retours de nos micro services back-end.

## Usage principal de Zustand

1. **Gestion de l’état global**
   Zustand permet de stocker et gérer l’état global des objets dans l'application React,

tout en offrant une simplicité proche de l’état local géré avec `useState` ou `useReducer`.

2. **État réactif**
   Les composants s’abonnent automatiquement aux changements des données, ce qui signifie qu’ils seront automatiquement re-rendus lorsque l’état auquel ils accèdent est mis à jour.
3. **Remplacement de solutions comme Redux**
   Zustan peut être utilisé comme une alternative à Redux sans nécessiter de boilerplate ou de configurations complexes comme les reducers ou les middlewares.

## Axios

**Axios** est une bibliothèque JavaScript qui facilite nos requêtes HTTP avec une API simple et intuitive. Compatible avec `async/await`, elle gère automatiquement les en-têtes, les cookies et les données JSON. Elle se distingue par ses fonctionnalités comme l’interception des requêtes/réponses, la configuration globale et l’annulation des requêtes, offrant une alternative plus flexible et puissante que Fetch.

## JsCookies

JsCookies nous permet à travers une gestion simplifié des cookies, de gérer un cookie contenant notre JWT délivré par l'authentification côté back-end. Cela permet d''identifier l'utilisateurs dans ses parcours et de lui délivrer des autorisations lorsqu'il appelle nos micro services.