Octobre 2025

Rapport de veille technologique

Grégoire Marousé Développeur Java

BUT Informatique 2e année (2B1)

IUT de Vannes - 8 Rue Michel de Montaigne, 56000 Vannes

Accompagnement

Maître d'alternance (CA-TS) : **R. Hannaoui** (rachid.hannaoui@ca-ts.fr)

Tuteur (IUT) : **JF.Kamp** (jean-francois.kamp@univ-ubs.fr)

Enseignants responsables de l'alternance à l'IUT

G.Kerbellec (goulven.kerbellec@univ-ubs.fr)

T.Godin (thibault.godin@univ-ubs.fr)

Table des matières

Introduction & Mission en entreprise

Pour mon alternance, j'ai rejoint le Crédit Agricole Technologies et Services (aussi appelé CA-TS) de Vannes pour un poste de développeur Java. Les principales missions de la filiale sont la conception, la fabrication et la maintenance du système d'information bancaire. Pour ma part, je fais partie de l'équipe Paiement mobile qui gère l'application de paiement mobile du Crédit Agricole.

Si je le souhaite, on m'a également proposé de travailler avec deux autres équipes :

- Option Carte : elle gère le parcours de gestion des cartes, la liste des cartes et les options associées.
- Cycle de Vie Carte : elle est chargée du suivi de la carte, de sa création jusqu'à sa récupération par la banque.

Pour ce rapport, je vais me concentrer sur l'équipe de paiement mobile. La mission de l'équipe est de gérer le développement de l'application de paiement mobile du Crédit Agricole côté back-end.

Les besoins

Pour mener à bien cette mission avec l'équipe, les principaux enjeux concernent avant tout la sécurité, qui doit être garantie en permanence. Un suivi utilisateur régulier est également nécessaire afin de corriger les éventuels problèmes, maintenir et assurer la disponibilité de l'application ainsi que son bon fonctionnement. Enfin, la mission comprend aussi une dimension de conception et de mise en place de nouvelles fonctionnalités potentielles.

Les compétences

Afin de répondre efficacement aux besoins de la mission, je vais être amené à développer et renforcer les compétences suivantes :

- Techniques : maîtrise de Java avec Spring Boot, développement et intégration d'API Rest, analyse du code existant, conception de solutions techniques, participation à l'élaboration de plans de tests.
- Collaboration et organisation : travail en équipe, communication interpersonnelle, pratique des méthodes Agiles.
- Analyse et résolution : compréhension et analyse des besoins, proposition de solutions adaptées, résolution de problèmes et mise en place de correctifs.
- Veille et amélioration continue : suivi des évolutions technologiques et intégration des bonnes pratiques.
- Documentation et support : rédaction de documentation technique et fonctionnelle, apport de support technique à l'équipe et aux utilisateurs.

Les technologies

De façon générale, je vais travailler sur du back-end en Java lié à l'application avec Spring Boot et des API Rest.

L'entreprise dispose également de plateformes internes pour la gestion des tickets ainsi que pour la documentation et l'accompagnement, afin de préserver la sensibilité et la confidentialité des informations. Les projets sont développés sous Eclipse, sur des PC équipés de Windows, et l'utilisation d'un outil de gestion de version tel que Git est probablement mise en place pour faciliter le travail en équipe.

Java est un langage de programmation de haut niveau, orienté objet, publié en 1995 par les fondateurs et ingénieurs de Sun Microsystems (aujourd'hui acquis par Oracle). Sa particularité repose sur la JVM (Java Virtual Machine), qui permet d'exécuter le même programme sur différents systèmes d'exploitation, faisant de Java un langage multi-plateforme par excellence.

Reconnu pour sa robustesse et sa fiabilité, Java est massivement utilisé dans des secteurs critiques tels que la banque, l'assurance ou encore les télécommunications. Son fonctionnement repose sur une compilation en bytecode, qui combine portabilité et performances correctes.

D'autre part, Spring Boot est un framework basé sur Spring, qui simplifie le développement en fournissant une configuration automatique, une gestion centralisée des dépendances et une intégration native des bonnes pratiques (sécurité, data, cloud, tests). Spring est créé au début des années 2000, et visait à

simplifier le développement en Java, qui était jugé trop lourd avec les standards de l'époque. C'est un framework open-source qui fournit une infrastructure complète pour le développement d'applications Java. Même si Spring a simplifié Java, il restait parfois complexe à configurer (fichiers XML, dépendances nombreuses, gestion manuelle des serveurs). Spring Boot, lancé en 2014, a été conçu pour réduire cette complexité et accélérer le développement.

Pourquoi Spring Boot est central dans un environnement Java + API Rest ?

- Il permet de créer rapidement des APIs Rest en exposant des contrôleurs web.
- Il assure une sécurité robuste (Spring Security), essentielle dans le contexte des paiements mobiles.
- Il offre une base standardisée et éprouvée, réduisant les risques d'erreurs et accélérant la mise en production.

Enfin, une API REST est une interface de programmation d'application (API) qui respecte les principes de conception du style d'architecture REST, abréviation de « REpresentational State Transfer ». Ce style regroupe un ensemble de règles et de consignes pour la création d'API web reposant sur HTTP, il permet l'interopérabilité et la communication entre systèmes de manière standardisée. Dans l'environnement Java + Spring Boot, elle est particulièrement bien prise en charge grâce à Spring MVC (contrôleurs Rest, gestion JSON intégrée).

Sans Spring Boot et Rest, le développement en Java serait :

- plus complexe (configuration manuelle lourde),
- moins rapide à mettre en place,
- moins interopérable (chaque système devrait inventer son propre protocole de communication).

En conclusion, c'est un écosystème bien pensé ou les 3 parties sont assez indispensable pour l'efficacité et la productivité aujourd'hui.

Réflexions sur l'écosystème

Dans cette partie nous allons nous intéresser aux alternatives à l'écosystème actuel avec les avantages et inconvénients de chacun tout en regardant les besoins liés à ma mission. Pour commencer reprenons les avantages et inconvénients de l'environnement Java, Spring Boot, API Rest. Vous retrouverez un tableau de synthèse à la fin des analyses.

Java Spring Boot & API Rest

Avantages	Inconvénients
Maturité et robustesse : éprouvé dans l'industrie depuis des décennies. Sécurité intégrée (Spring Security, gestion des transactions). Écosystème riche : librairies, frameworks, outils de monitoring, intégrations cloud. Scalabilité : adapté aux microservices et architectures distribuées. Standardisation : API Rest largement comprise et supportée par toutes les plateformes. Disponibilité des compétences : grand nombre de développeurs formés sur Java/Spring.	Performance relative : Java consomme plus de ressources que des environnements compilés natifs (C, Go). Verbosité : le langage Java reste parfois lourd comparé à Kotlin ou Python. Évolution lente : environnement stable mais perçu comme moins innovant que d'autres stacks modernes.

.NET Core (C# + ASP.NET Core + gRPC/Rest)

L'écosystème .NET est né au début des années 2000 sous l'impulsion de Microsoft, avec pour objectif de concurrencer directement Java dans le développement d'applications d'entreprise. Initialement limité à l'univers Windows, il s'est ouvert avec l'arrivée de .NET Core en 2016, version multiplateforme et open source qui a marqué un tournant. ASP.NET Core, la brique web de la plateforme, permet de développer des applications exposant des APIs Rest ou gRPC, dans une logique très proche de celle de Spring Boot.

La force de .NET Core réside dans ses performances et son intégration naturelle avec l'écosystème Microsoft, en particulier le cloud Azure et les services de sécurité ou de gestion d'identité comme Active Directory. Ces atouts en font une alternative solide pour des applications critiques, tout en bénéficiant d'un support industriel massif. En revanche, cet environnement souffre parfois de l'image d'un produit encore trop lié à Microsoft, ce qui peut freiner son adoption dans des environnements très hétérogènes.

Dans le cadre d'une application de paiement mobile, .NET Core représente un écosystème très solide et mature, capable de rivaliser avec Java/Spring Boot. Idéal

si l'organisation est déjà intégrée à l'écosystème Microsoft (cloud Azure, infrastructures Windows). Cependant, dans un contexte bancaire multi-partenaire, Java conserve un avantage en termes de standardisation et reste plus universel.

L'écosystème JavaScript : Node.js et l'évolution vers Deno

JavaScript, longtemps limité au navigateur, a connu une véritable révolution avec la création de Node.js en 2009. Pour la première fois, ce langage pouvait être exécuté côté serveur, ouvrant la voie à des architectures full-stack unifiées où le front-end et le back-end partagent le même langage. Des frameworks comme Express.js (léger et minimaliste) ou NestJS (plus structuré et orienté entreprise) ont ensuite apporté un cadre plus robuste au développement serveur. Dans le même esprit, GraphQL a introduit une nouvelle façon d'interroger les données, particulièrement efficace pour les applications mobiles nécessitant une communication rapide et flexible.

Node.js s'est imposé par sa rapidité de développement et la richesse de son écosystème NPM, permettant de créer des APIs ou des prototypes en un temps record. Cependant, cette agilité se fait parfois au détriment de la robustesse et de la sécurité. Le modèle mono-threadé de Node et la gestion complexe des dépendances peuvent devenir des limites dans des environnements exigeants, notamment lorsqu'il s'agit d'applications critiques.

Pour répondre à ces enjeux, Deno est créé en 2020 par Ryan Dahl, le concepteur de Node.js, comme une alternative plus moderne et sécurisée. Écrit en Rust, Deno vise à corriger les faiblesses structurelles de Node en intégrant TypeScript nativement et en exécutant le code dans une sandbox sécurisée, où chaque accès au système ou au réseau doit être explicitement autorisé. Personnellement pour mes projets en web j'utilise Le framework Fresh, construit sur Deno. Il pousse encore plus loin cette approche en favorisant un rendu côté serveur et en minimisant le JavaScript exécuté côté client.

Dans le cadre d'une application de paiement mobile, ces technologies peuvent être intéressantes pour développer des interfaces interactives, des services web légers ou des composants front-end réactifs. Toutefois, leur usage comme socle technique principal reste limité par des questions de maturité, de sécurité certifiée et de garanties industrielles.

En comparaison, l'écosystème Java/Spring Boot/Rest conserve une avance décisive dans ce type de projet. Sa stabilité, sa conformité aux standards du secteur bancaire et sa robustesse éprouvée en production en font une base plus adaptée pour gérer la logique métier et les transactions sensibles d'une application de paiement mobile, là où les environnements JavaScript restent davantage orientés vers la présentation et la réactivité du front-end.

Go (Go + Rest/gRPC)

Go, créé par Google à la fin des années 2000, est un langage pensé pour répondre aux défis de performance et de simplicité rencontrés dans les infrastructures cloud et distribuées. Sa syntaxe épurée, proche de C dans l'esprit mais beaucoup plus accessible, et sa compilation rapide en font un langage très apprécié pour les microservices. Go excelle dans les environnements nécessitant rapidité et efficacité, et il est aujourd'hui adopté par plusieurs grands acteurs du cloud.

Cependant, son écosystème reste plus jeune et moins riche que celui de Java ou .NET. Les frameworks de sécurité, de gestion des transactions ou de conformité bancaire y sont moins développés. Pour une fintech cherchant à innover vite et à construire des services très performants en environnement cloud, Go peut représenter une option séduisante. En revanche, dans une banque traditionnelle, il impose un effort de formation et de gouvernance important, avec un risque d'isolement technologique.

Dans le cas d'une application de paiement mobile, Go pourrait être une bonne solution pour développer des services périphériques très performants, mais il apparaît encore trop immature pour porter seul un cœur applicatif bancaire.

Environnements sans API standardisée (Monolithes, SOAP, RPC maison)

Avant l'essor de Rest et des API ouvertes, les systèmes bancaires reposaient sur des architectures monolithiques ou sur des standards comme SOAP. Ces approches avaient leur logique : le monolithe garantissait une cohérence interne, tandis que SOAP proposait un protocole fortement typé et sécurisé, qui a longtemps été la norme dans le monde bancaire. Pourtant, ces technologies se sont révélées peu adaptées à l'ère des applications mobiles et de l'ouverture des services imposée par les régulateurs (comme la directive européenne PSD2). SOAP est lourd, verbeux, difficile à intégrer avec des front-end modernes. Quant aux architectures monolithiques, elles manquent de souplesse pour évoluer dans un environnement où la modularité et la scalabilité sont devenues essentielles.

Dans le cadre d'une application de paiement mobile, persister avec des environnements sans API standardisée serait un frein majeur. La banque ne pourrait ni offrir une expérience fluide à ses utilisateurs, ni s'interconnecter facilement avec des partenaires tiers. C'est précisément cette exigence d'ouverture et de compatibilité qui a contribué à imposer l'écosystème Java/Spring Boot/Rest comme un standard dans le secteur.

Tableau de synthèse des comparaisons

Écosystème	Coût de mise en œuvre	Performance et Scalabilité	Stabilité et Maturité	Sécurité & Conformité	Facilité de développement	Interopérabilité & ouverture
Java + Spring Boot + Rest	Moyen à élevé (compétences expertes, infra plus lourde)	Bonne mais consommatrice de ressources	Très élevée (standard bancaire mondial)	Excellente (frameworks éprouvés)	Moyen	Excellente (Rest standardisé, large écosystème)
.NET Core (C# + ASP.NET Core)	Moyen (plus compétitif si écosystème Microsoft déjà en place)	Très bonne (runtime performant, gRPC efficace)	Élevée mais adoption bancaire plus limitée	Très bonne (intégration sécurité Microsoft)	Relativement accessible	Bonne (surtout avec Azure, moins universelle que Java)
Node.js (Express/NestJS + Rest/GraphQL)	Faible à moyen (équipe facile à recruter, infra légère)	Moyenne (mono-thread, adapté aux apps légères)	Moyenne (moins éprouvé pour systèmes critiques)	Moyenne (besoin d'efforts supplémentaires)	Très bonne (prototypage rapide)	Bonne (flexible, GraphQL, APIs modernes)
Deno	Faible au départ, élevé en expertise	Bonne performance, peu de recul industriel	Immature, encore en évolution	Excellente sécurité, conformité limitée	Très bonne, TypeScript natif, écosystème restreint	Standards modernes, compatibilité limitée
Go (Golang + Rest/gRPC)	Moyen (expertise rare, mais infra légère)	Excellente (performances proches du natif)	Moyenne (langage jeune)	Moyenne à bonne (sécurité encore en construction)	Bonne (syntaxe simple, rapide à prendre en main)	Bonne (gRPC performant, interopérable)
Sans API standardisée (SOAP/monolithes)	Élevé (maintenance lourde, compétences rares)	Faible à moyenne (rigidité, peu scalable)	Ancienne mais peu adaptée au moderne	Bonne historiquement, mais inadaptée au mobile	Faible (développement lent, verbeux)	Faible (interopérabilité limitée)

Conclusion

L’analyse du tableau de synthèse montre que chaque environnement possède ses forces propres : .NET Core séduit par ses performances et son intégration Microsoft, Node.js par sa rapidité et sa flexibilité, et Go par son efficacité et sa légèreté. Deno, plus récent, propose une approche moderne et sécurisée mais encore peu mature. À l’inverse, les solutions plus anciennes comme SOAP apparaissent désormais inadaptées aux besoins du développement mobile moderne.

Dans le cadre du développement d’une application de paiement mobile côté back-end, l’écosystème Java + Spring Boot + Rest apparaît comme le choix le plus pertinent. Sa maturité, sa stabilité éprouvée et son adoption massive dans le secteur bancaire en font une solution de confiance, parfaitement adaptée aux exigences de sécurité et de conformité. Même si son coût de mise en œuvre peut être supérieur et ses performances parfois moins optimales que des alternatives plus récentes, ces limites sont compensées par la richesse de son écosystème, sa robustesse et son aspect universelle.

D’autre part le CA-TS utilise également des environnement C# pour d’autres services avec des besoins légèrement différents. Les environnement C# et Java sont les deux écosystèmes qui se démarquent du lot à travers cette veille technologique. Selon les futures évolutions du domaine informatique, on peut imaginer le Go ou une alternative sécurisée de Node.js déloger un de ces géants après avoir atteint une certaine maturité nécessaire aux milieux critiques.

Sources

https://www.axopen.com/technos/versus

https://fr.wikipedia.org/wiki/Java_(langage)

https://fr.wikipedia.org/wiki/Spring_(framework)

https://fr.wikipedia.org/wiki/Spring_Boot

https://spring.io/projects/spring-boot

https://www.ibm.com/fr-fr/think/topics/rest-apis

https://www.redhat.com/fr/topics/api/what-is-a-rest-api

https://fr.wikipedia.org/wiki/.NET

https://learn.microsoft.com/fr-fr/aspnet/core/overview?view=aspnetcore-9.0

https://fr.wikipedia.org/wiki/ASP.NET_Core

https://fr.wikipedia.org/wiki/Go_(langage)

https://nodejs.org/fr

https://en.wikipedia.org/wiki/Node.js

https://techvify.com/best-programming-language-for-banking-software/

https://www.techempower.com/benchmarks/

https://roshancloudarchitect.me/benchmarking-giants-asp-net-core-8-vs-node-js-vs-go-a-performance-analysis-964808bc6013