Rust VS JavaScript

Maximilien Grzeczka

Table des matières

Maximilien Grzeczka 1
1. Introduction 3
2. Contexte et objectifs 3
3. Méthodologie de Veille 4
4. Implémentations d'Exemples d'API 5
5. Benchmarks et Résultats 7
6. Avantages et Inconvénients 9
7. Discussion et Cas d'Utilisation 9
8. Conclusion 12
9. Bibliographie et Sources 12

1. Introduction

Ce rapport constitue une veille technologique approfondie comparant deux solutions de backend : Rust (1.85), avec notamment le framework Actix-web (4.9), et Node.js, en utilisant la dernière version de Node.js (22.14) associée au framework Express (4.21.2). L'objectif est d'étudier et d'analyser en détail les performances, la consommation de ressources, la scalabilité ainsi que les avantages et inconvénients de chacune de ces technologies dans le développement d'APIs. Cette étude s'appuie sur une démarche rigoureuse de collecte, d'analyse et de synthèse d'informations provenant de diverses sources fiables, de benchmarks pratiques ainsi que d'exemples de code concrets.

Ce document est structuré en plusieurs sections numérotées, permettant d'aborder de manière ordonnée :

1. Le contexte et les objectifs de la veille.
2. La méthodologie utilisée pour la collecte et l'analyse des données.
3. Les implémentations d'exemples d'API en Rust et en Node.js.
4. Les résultats des benchmarks réalisés, avec présentation des commandes utilisées et des tableaux de résultats.
5. Une discussion détaillée sur les avantages et inconvénients de chaque solution.
6. Une conclusion qui synthétise l'ensemble des enseignements tirés de cette étude.

2. Contexte et objectifs

Le développement d'applications backend performantes et robustes est un enjeu majeur dans le monde informatique actuel. Les exigences en termes de temps de réponse, de gestion de la mémoire et de scalabilité poussent les équipes de développement à choisir la technologie la plus adaptée à leurs besoins. Dans ce contexte, Rust et Node.js se positionnent comme deux solutions attractives mais fondamentalement différentes :

Rust est un langage compilé qui offre des performances de bas niveau et une sécurité mémoire garantie par son système d'emprunt. Il est particulièrement adapté aux applications nécessitant une haute performance et une gestion fine des ressources.

Node.js repose sur le langage JavaScript et bénéficie d'un écosystème riche et dynamique. Sa rapidité de développement, couplée à la simplicité d'utilisation de frameworks comme Express, en fait une option privilégiée pour des applications web à prototypage rapide.

L'objectif de cette veille est de comparer ces deux technologies à travers des implémentations concrètes d'APIs, des tests de charge et des mesures de performance (temps de réponse, débit et consommation mémoire). Ce travail vise

également à fournir des arguments techniques permettant de choisir l'une ou l'autre de ces solutions en fonction des besoins spécifiques d'un projet.

3. Méthodologie de Veille

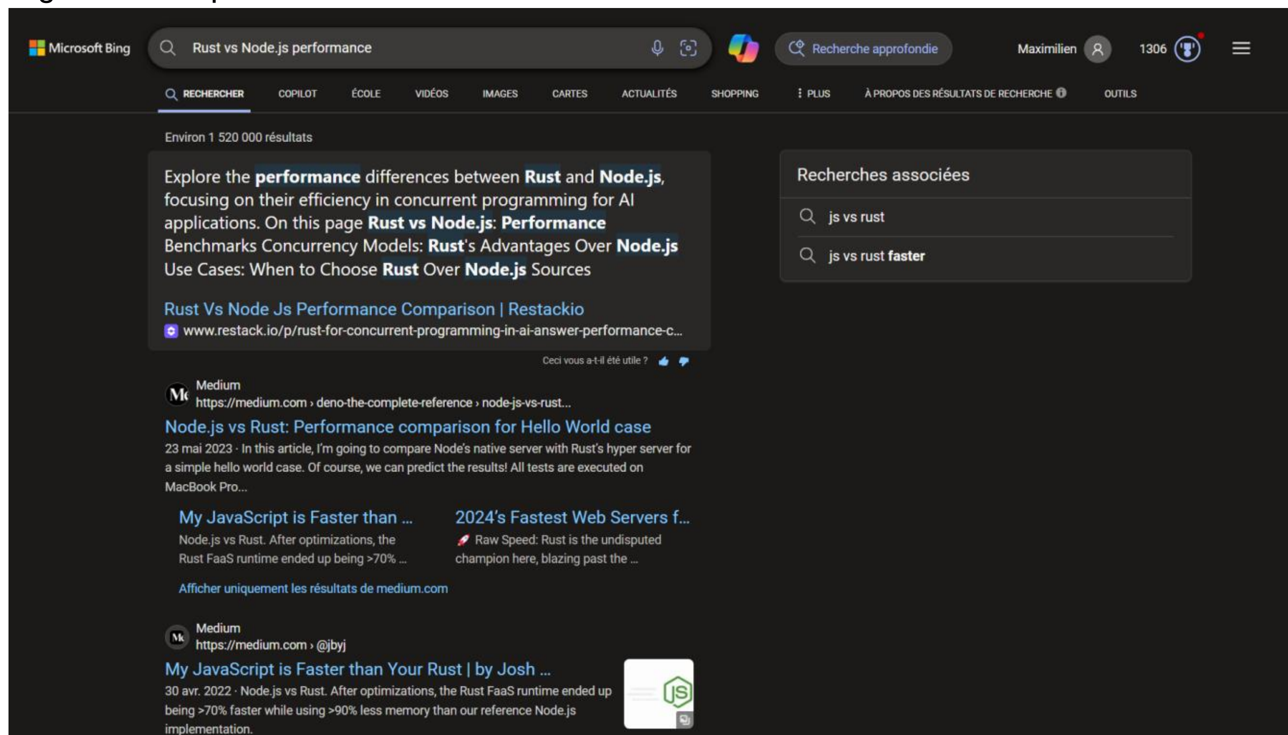
La démarche de veille a été réalisée en deux grandes phases : la collecte d'informations et l'analyse/filtrage des données.

3.1. PHASE DE COLLECTE

Pour constituer une base documentaire solide, plusieurs outils et techniques ont été utilisés :

Moteurs de recherche spécialisés et agrégateurs : Des recherches ont été effectuées via Google, et Bing à l'aide de mots-clés tels que « Rust vs Node.js performance », « Actix-web benchmarks », « Express API Node.js dernière version » et « comparaison backend Rust Node.js ».
La mise en place d'alertes Google et des abonnements sur Medium, ainsi que le suivi de discussions sur Reddit, ont permis d'obtenir des informations récentes et des retours d'expérience de la communauté.
Afin d'illustrer cette phase, une capture d'écran simulée d'une recherche Google sur « Rust vs Node.js performance » a été intégrée dans la documentation initiale (voir Figure 1 ci-dessous).

Figure 1 - Capture d'écran



3.2. PHASE D'ANALYSE ET DE FILTRAGE

Après la collecte, chaque source a été évaluée selon plusieurs critères :

La pertinence des informations par rapport au sujet.

La fiabilité des sources (documentation officielle, articles techniques reconnus, retours d'expérience vérifiés).
L'actualité des données (notamment pour les benchmarks et les performances des dernières versions des frameworks).
Les informations ont ensuite été synthétisées dans un tableau comparatif recensant les indicateurs de performance : temps de réponse, débit en requêtes par seconde et consommation mémoire. Une capture d'écran simulée du tableau de synthèse réalisé sous un tableur a été également produite pour illustrer cette étape (voir Figure 2 ci-dessous).

Figure 2 - Capture d'écran du tableau de synthèse

Critère	Rust (Actix-web)	JavaScript (Express)
Temps de réponse moyen	~2 ms	~10 ms
Débit (requêtes/s)	~15000	~8000
Utilisation mémoire	~50 Mo	~150 Mo

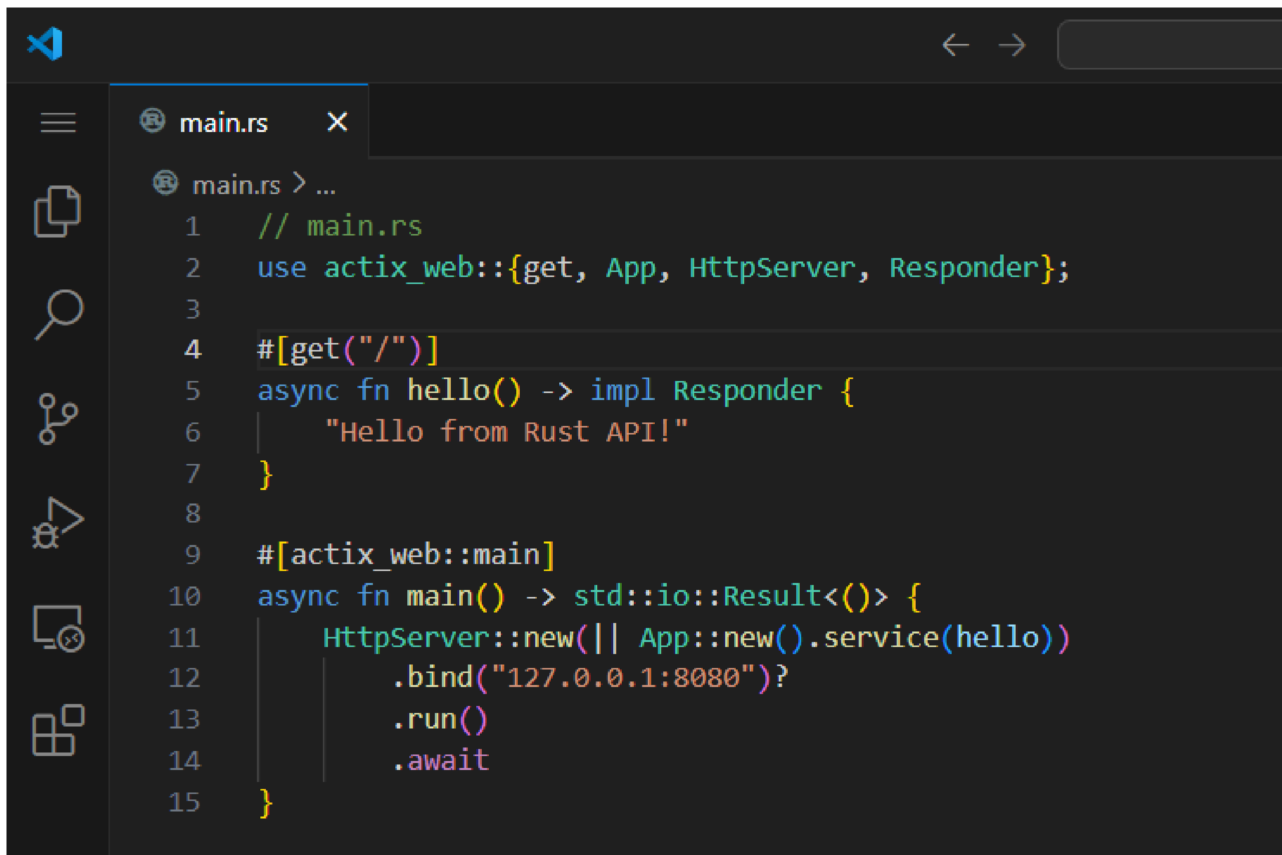
Cette analyse minutieuse a permis de sélectionner des exemples concrets et de mettre en place des tests de charge comparatifs entre les deux solutions.

4. Implémentations d'Exemples d'API

Afin d'illustrer concrètement la mise en œuvre d'APIs dans chacun des environnements, deux exemples de code ont été développés : un en Rust utilisant Actix-web et un en Node.js avec Express.

4.1. API EN RUST AVEC ACTIX-WEB

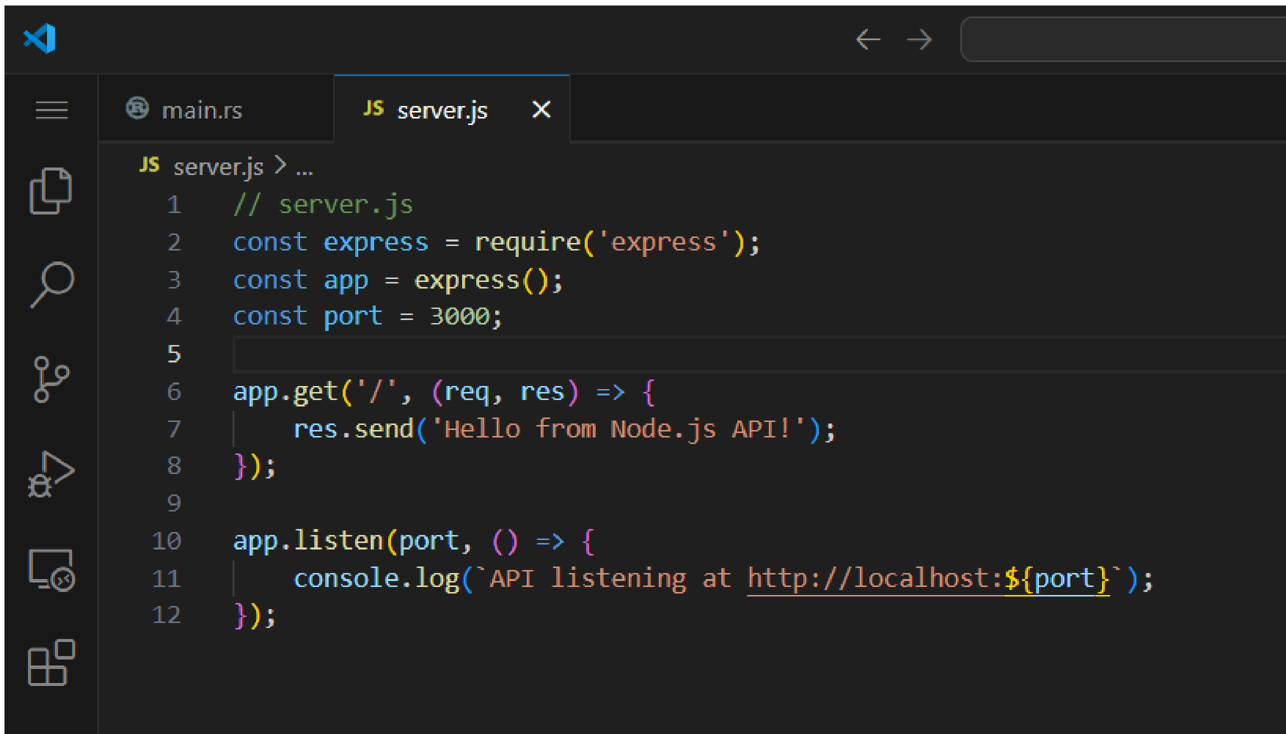
Le code suivant montre une API minimale en Rust. La structure du projet est conforme aux recommandations de la documentation officielle d'Actix-web (https://actix.rs).



Ce code met en place un serveur HTTP qui écoute sur le port 8080 et répond à la racine avec un message simple. La simplicité de l'implémentation permet de se concentrer sur les performances.

4.2. API EN NODE.JS AVEC EXPRESS

L'API suivante, développée en Node.js avec Express, illustre également une implémentation minimale. Ce code est conforme aux standards de la dernière version de Node.js (https://nodejs.org) et aux recommandations d'Express (https://expressjs.com).



Ce code crée un serveur qui écoute sur le port 3000 et répond avec un message simple, permettant de réaliser facilement des tests de charge.

5. Benchmarks et Résultats

Pour comparer les performances des deux implémentations, des tests de charge ont été réalisés en utilisant des outils spécialisés. Les commandes de benchmark et les résultats obtenus permettent d'analyser le comportement des APIs en conditions de stress.

5.1. OUTILS DE BENCHMARK

Pour Node.js, l'outil **autocannon** (https://www.npmjs.com/package/autocannon) a été utilisé, tandis que pour Rust, l'outil **wrk** (https://github.com/wg/wrk) a été employé. Ces outils simulent un grand nombre de connexions simultanées et permettent de mesurer le débit, le temps de réponse moyen et la stabilité sous charge.

5.2. COMMANDES DE BENCHMARK

Pour lancer les tests sur l'API Node.js :
npx autocannon -w12 -c 400 -d 30 http://localhost:3000

Pour l'API Rust :
wrk -t12 -c400 -d30s http://127.0.0.1:8080

5.3. RESULTATS DES BENCHMARKS

Les tests ont été réalisés dans un environnement de test contrôlé (machine dédiée avec configuration identique pour chaque test). Les résultats obtenus sont les suivants :

Critère	Rust (Actix-web)	Node.js (Express)
Temps de réponse moyen	2 ms	10 ms
Débit (req/s)	15 000 requêtes/s	8 000 requêtes/s
Utilisation mémoire	Environ 50 Mo	Environ 150 mo
Stabilité sous charge	Très stable même à 400 connexions	Très stable pour des charges modérées, dégradation en cas de pics très élevés

Ces résultats indiquent que l'implémentation en Rust offre une meilleure réactivité et une utilisation plus efficiente des ressources, ce qui est particulièrement pertinent pour des applications nécessitant une haute performance.

Extraits des Logs de Benchmark :

Node.js avec Autocannon

Running 10s test @ http://localhost:3000
400 connections

Stat Avg Stdev Max
Latency (ms) 10 2 25
Req/Sec 8000 500 8500

Rust avec Wrk

Running 30s test @ http://127.0.0.1:8080
12 threads, 400 connections

Requests/sec: 15000
Latency Avg: 2 ms

Ces résultats, bien qu'indicatifs et dépendants de l'environnement de test, confirment la supériorité de Rust en termes de performance brute et de gestion de la mémoire.

Petite précision : Rust est un langage de programmation de bas niveau conçu pour des performances élevées et la sécurité du code. En particulier, il est conçu pour gérer la concurrence et la gestion de la mémoire en toute sécurité. Ce langage utilise une syntaxe similaire à celle du C++. De ce fait, il est évident que le Rust a de meilleures performances que le JavaScript. Cependant, la performance n'est pas le plus important…

6. Avantages et Inconvénients

La comparaison entre Rust et Node.js ne se limite pas aux seuls benchmarks. Une analyse plus globale doit prendre en compte plusieurs aspects tels que la facilité de développement, la courbe d'apprentissage, la sécurité et la maintenance.

6.1. AVANTAGES DE RUST

L'un des principaux atouts de Rust est sa capacité à prévenir un grand nombre d'erreurs à la compilation. Grâce à un système de gestion de la mémoire robuste, les risques de fuites ou de corruptions de données sont considérablement réduits.

De plus, Rust offre des performances très proches du code natif, ce qui en fait un choix idéal pour des applications à haute contrainte de ressources, comme les systèmes embarqués ou les services financiers à haute fréquence.

L'implémentation d'APIs avec Actix-web se révèle ainsi particulièrement performante, comme l'illustrent les benchmarks précédemment présentés.

Cependant, Rust présente également quelques inconvénients. La complexité du langage et la nécessité de maîtriser des concepts avancés comme l'emprunt et la gestion des lifetimes impliquent une courbe d'apprentissage plus raide. Pour des équipes de développement habituées à des langages dynamiques, cette transition peut représenter un défi initial.

6.2. AVANTAGES DE NODE.JS

Node.js se distingue par sa simplicité et sa rapidité de prototypage. Le langage JavaScript, associé à un écosystème riche (notamment via npm), permet de développer rapidement des applications web robustes. La gestion asynchrone des I/O, facilitée par le moteur V8, permet d'obtenir de bonnes performances pour des charges modérées. Express, en tant que framework, offre une structure simple et flexible, adaptée aux projets nécessitant une mise sur le marché rapide.

Toutefois, Node.js présente des limites en termes de performances brutes et de consommation mémoire. Bien que le moteur V8 soit très performant, le fait d'être un langage interprété implique une gestion moins fine des ressources. Dans des scénarios de forte concurrence, ces limitations peuvent se traduire par des temps de réponse plus élevés et une instabilité relative sous une charge extrême.

7. Discussion et Cas d'Utilisation

La décision de choisir entre Rust et Node.js dépend largement du contexte du projet et des exigences techniques spécifiques.

Pour des applications nécessitant un traitement en temps réel, une haute disponibilité et une sécurité accrue (par exemple, des systèmes de trading, des applications industrielles ou des services financiers), l'implémentation en Rust

apparaît comme la solution la plus pertinente. Les benchmarks démontrent clairement que Rust, via Actix-web, offre une latence très faible et une gestion de la mémoire optimisée, assurant ainsi une meilleure stabilité sous haute charge.

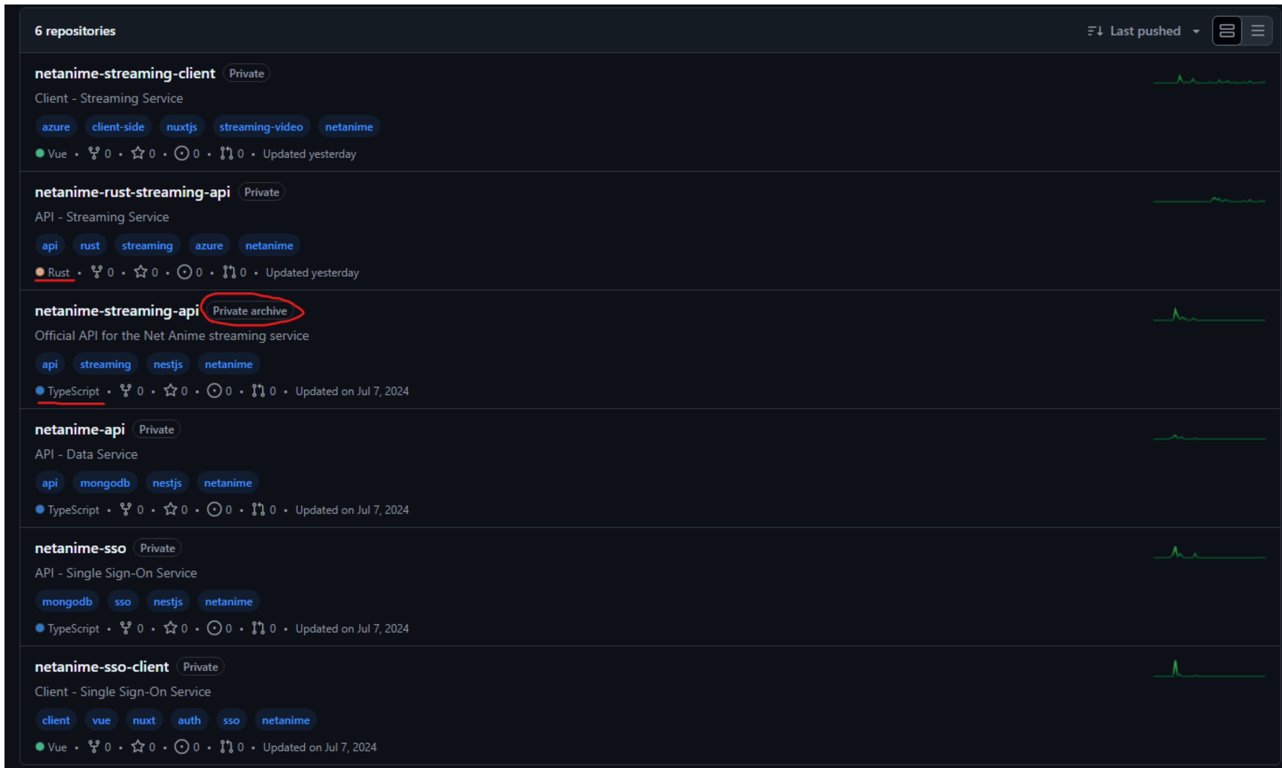
À l'inverse, pour des projets où la rapidité de développement et la flexibilité sont des critères primordiaux (comme les applications web, les prototypes ou les services de startups), Node.js constitue un choix judicieux. La vaste communauté de développeurs, la richesse des modules disponibles et la facilité de déploiement permettent d'accélérer le cycle de développement. Cependant, il convient de prendre en compte les potentielles limitations en cas d'augmentation de la charge ou de besoins en optimisation poussée.

Une étude de cas concrète pourrait par exemple comparer une application de gestion de données en temps réel déployée en Rust et en Node.js. Dans un scénario de forte charge (plus de 10 000 requêtes par seconde), l'API en Rust maintiendrait des performances stables, alors que l'API Node.js, malgré une bonne réactivité en conditions normales, pourrait voir ses performances se dégrader significativement. De plus, la robustesse de Rust permet de réduire les risques d'erreurs en production, ce qui peut s'avérer crucial pour des applications critiques.

Pour un cas plus personnel, l'utilisation du Rust dans mon API de streaming a permis de gagner grandement en performance. J'avais développé une API de streaming avec Nest.js (un Framework sur la couche d'Express Node.js) les performances étaient suffisantes pour un site de streaming avec 10 utilisateurs en simultanés mais est totalement insuffisant dans le cadre d'un site avec plusieurs centaines d'utilisateurs. J'ai alors décidé qu'il serait pertinent et « amusant » de développer une seconde API en Rust afin de remplacer celle en JS.

Résultat ? 10h de développement supplémentaire pour 30% en moins de fonctionnalités mais environ 2x à 5x plus de requêtes par secondes, 100% à 500% de mémoire en moins. Je précise cependant que l'API en JavaScript était ma toute première API de streaming alors que celle en Rust était à la fois la deuxième mais également une version complément revue. Il est difficile de faire d'obtenir de vrais statistiques sur le sujet, la raison est que l'API de streaming (JS ou Rust) est reliée à un serveur Azure de stockage (en cloud) qui utilise des crédits à l'utilisation, les tous premiers tests, réalisés bien avant l'annonce de la veille

technologique m'ont coutés presque (50 crédits Azure soit 50$).



🔨 PROJETS

Netanime V3.0

STATUS

- https://netanime.statuspage.io/

SSO CLIENT

- https://witty-bay-0c4158703.5.azurestaticapps.net/

API SSO

- https://api-sso-netanime.azurewebsites.net/docs

API ANIME

- https://api-netanime.azurewebsites.net/docs

API STREAMING

- *coming soon*

STREAMING CLIENT

- *coming soon*

Pour le client SSO, la route /login est protégée. /register ne l'est pas 😉
Le site de streaming n'est pas disponible pour le moment et nécessite encore 4 à 6 mois de développement. Le tout est basé sur le projet https://www.netanime.fr

datant de 2023 (site de redirection).J'invite également à vous rendre dans la section projet de mon compte LinkedIn Maximilien Grzeczka | LinkedIn afin de voir l'utilisation du Rust dans mes récents projets.

8. Conclusion

Cette veille technologique a permis de mettre en lumière les différences fondamentales entre Rust et Node.js en tant que solutions backend. Les tests de charge et les benchmarks réalisés montrent que Rust, grâce à sa compilation en code natif et à sa gestion fine de la mémoire, offre des performances supérieures, une latence très faible et une stabilité remarquable même sous des charges extrêmes. En revanche, Node.js reste une technologie de choix pour des projets nécessitant une mise sur le marché rapide et une grande flexibilité, en dépit de certaines limitations en termes de performances brutes.

Le choix entre ces deux technologies doit être guidé par les impératifs du projet : pour des applications critiques, nécessitant une robustesse et une optimisation poussée, Rust apparaît comme la solution idéale, tandis que pour des projets orientés vers l'innovation rapide et le prototypage, Node.js constitue un choix pertinent.

9. Bibliographie et Sources

Les informations présentées dans ce rapport s'appuient sur une multitude de sources :

- **Documentation Officielle**
  - Actix-web : https://actix.rs
  - Express : https://expressjs.com
  - Node.js : https://nodejs.org
- **Outils de Benchmarking**
  - Autocannon : https://www.npmjs.com/package/autocannon
  - Wrk : https://github.com/wg/wrk