UNIVERSITÉ CÔTE D'AZUR

POLYTECH NICE SOPHIA

Rapport de stage

IOmentum

EPU 4

Tom Niget

RAPPORT INTERMÉDIAIRE

Nature du stage	Stage
Nom de l'entreprise	IOmentum
Ville de l'entreprise	La Madeleine
Pays de l'entreprise	France
Type de présence	Distanciel
Date de début	2022-05-16
Date de fin	2022-07-31
Titre du sujet	Développement d'outils d'analyse de code et de test automatisé

Résumé

Français

J'ai effectué un stage de 11 semaines dans l'entreprise *IOmentum*, agence de développement spécialisée dans la conception et la réalisation d'applications web et backend. Durant ce stage, j'ai été amené à travailler sur deux projets OSS (open-source software) de l'entreprise, *cargo-breaking* et *restest*, permettant respectivement de détecter les changements cassants entre deux versions d'une base de code, et de tester automatiquement de manière déclarative des endpoints d'APIs REST.

Mon stage s'est entièrement déroulé en télétravail, avec au moins une réunion hebdomadaire pour évaluer l'avancement de mon travail et mon bien-être dans l'entreprise, et les communications s'effectuaient via la plateforme de discussion instantanée interne. Ayant déjà pu expérimenter le télétravail par le passé lors de précédentes expériences professionnelles, je n'ai pas eu de problèmes en découlant.

J'ai été principalement suivi par M. Cédric Bonaudo, ingénieur, et M. Jérémy Lempereur, directeur, mais ai pu faire la connaissance de nombre d'autres collaborateurs durant ma période passée au sein de l'entreprise.

Anglais

I was an intern for 11 weeks at IOmentum, a software development company specializing in web and backend applications design and implementation. During this internship, I worked on two OSS (open-source software) projects of the company, cargo-breaking and restest, which respectively allow detecting breaking changes between two versions of a code base, and automatically testing REST API endpoints in a declarative way.

My internship was entirely done on a remote basis, with a minimum of one meeting each week to evaluate my performance and my well-being in the company, and communications were done via the in-house instant messaging platform. Having experienced remote work in the past, I experienced no difficulties stemming from it.

I was mainly managed by Mr. Cédric Bonaudo, engineer, and Mr. Jérémy Lempereur, head of the company, but I was able to meet a lot of other collaborators during my time with the company.

TABLE DES MATIÈRES

REMERCIEMENTS

Je tiens tout d'abord à remercier mon maître de stage, Jérémy Lempereur, qui en m'offrant ce stage, m'a permis de découvrir un monde professionnel et une manière de travailler très agréables que je ne suis pas près d'oublier.

Merci à toute l'équipe IOmentum, de manière générale, pour son accueil et sa gentillesse.

Pour terminer, merci à l'administration de l'Université, dont l'efficacité n'a d'égal que son respect à l'égard des étudiants.

MISE EN CONTEXTE

IOmentum est une société de conseil en informatique, dont l'activité est centrée principalement sur le développement de services et applications Web pour le compte d'autres entreprises. Elle emploie des salariés en CDI, certains consultants à plein temps pour des clients, d'autres ne travaillant que sur des projets internes, ainsi que des étudiants alternants et stagiaires, venant d'école d'ingénieurs en majorité.

En plus de son activité directement commerciale, IOmentum est particulièrement active dans l'écosystème Rust[1], langage dépendant aujourd'hui de la fondation éponyme mais originellement publié il y a une dizaine d'années par la fondation Mozilla. Pour de nombreuses raisons, ce langage a rassemblé autour de lui une communauté assez importante et investie, reposant beaucoup plus sur le bouche-à-oreille et la réputation que sur une entreprise importante dirigeant le navire en apportant fonds et avocats[2]. La gestion est transparente et communautaire, et le processus d'évolution du langage est entièrement dirigé par la fondation et ouverte aux contributions.

Depuis 10 ans, le langage a gagné en popularité[3], et de nombreuses entreprises[4] investissent dans la fondation Rust et dans le développement du langage, en vue de l'aider à durablement remplacer le C et le C++ (entre autres) pour certains usages. La recherche en cybersécurité, depuis des années, a continué de démontrer que les erreurs d'accès mémoire restent l'origine principale de failles de sécurité dans les logiciels[5], découlant souvent du laxisme du C, mais en somme facilement évitées via un langage qui les supprime à la racine, comme le Rust.

Étant compilé (à l'inverse du JavaScript ou du Python) vers des binaires natifs (à l'inverse du Java ou du C#) et basant sa gestion mémoire sur le RAII plutôt que le *garbage collection* (à l'instar du C++ et à l'inverse du Go), le Rust se classe régulièrement en haut du podium en matière de performances, notamment de frameworks Web[6]. En outre, il occupe depuis 7 ans la place du langage le plus apprécié dans la *Stack Overflow Developer Survey*[7], un sondage annuel effectué par le site du même nom, dont les résultats sont chaque année scrutés par le monde du développement.

---

[1] (IOmentum, 2022)

[2] Le Java, par exemple, est entièrement sous la houlette d'Oracle, et le « Java Community Process », permettant à des parties tierces de suggérer des changements au langage ou aux spécifications, n'attend en pratique de suggestions que de grandes entreprises, clientes d'Oracle. Depuis le rachat de Sun, les problèmes liés aux droits d'auteurs et aux contrats de licence exigeants d'Oracle ont également diminué l'attractivité du langage (Clark, 2022), au profit par exemple de Kotlin ou Clojure.

[3] Plus récemment, il est même devenu le premier langage autre que le C à être officiellement autorisé dans la base de code de Linux (LKML, 2021).

[4] Microsoft fournit des bindings pour l'API Win32 en Rust ; AWS (Amazon) emploie à plein-temps des contributeurs au langage ; Mozilla a réécrit une partie de Firefox en Rust.

[5] (ZDnet, 2019)

[6] (TechEmpower, 2022)

[7] (Stack Exchange, 2022)

Tout ceci, bien sûr, n'est possible que grâce à l'investissement de la communauté qui développe et maintient la partie la plus importante du langage : l'écosystème. Pas de Java sans Spring ou Apache Commons, pas de JavaScript sans Node.js ou React.

Dans ce but, IOmentum développe de nombreux projets dits OSS (open-source software), tels que des outils de développement, des librairies, ou même des plateformes servant de base à des sites Web, et qui sont fournis gratuitement à la communauté, sous une licence open-source. Ces projets ne génèrent pas directement de revenus pour l'entreprise, mais démontrent à la communauté l'investissement qu'elle fournit dans l'écosystème. La finalité est qu'au sein de la communauté, une réputation s'établit et peut fournir à l'entreprise à la fois des clients travaillant dans la même éthique, et de la main-d'œuvre à la recherche d'un employeur investi.

TRAVAIL DEMANDÉ ET RÉALISÉ

Mon rôle dans tout cela a été de travailler sur deux projets OSS de l'entreprise ; sans milestones précises (les projets n'ayant pas de scope déterminé), simplement en les améliorant du mieux que j'ai pu durant le temps de mon stage.

Projet 1 – cargo-breaking

a. Contexte

Le premier projet s'appelle cargo-breaking, et est destiné à permettre au développeur d'une librairie (Rust) de détecter automatiquement à chaque nouvelle version les changements cassants effectués au niveau du code, afin de déterminer de manière systématique le numéro à attribuer à la prochaine version.

En effet, l'écosystème Rust suit, comme une grande partie du monde du développement aujourd'hui, de manière stricte la convention Semantic Versioning [8] (ci-après semver), qui consiste à former les versions des projets (librairies ou applications) sous la forme A.B.C, correspondant respectivement au numéros majeur, mineur et de correctif. Des règles précises régissent la modification de ces numéros. Par exemple, une version 1.6.2 devient 1.6.3 en cas de correctif mineur, 1.7.0 en cas d'ajout non cassant et 2.0.0 en cas de modification cassante (changement de signature de fonction, suppression, dépréciation, …).

Une conséquence directe de ce système est une facilitation drastique d'un problème rencontré par tout système de dépendances : l'unification.

Imaginons le cas suivant : un projet a deux dépendances, notées 1 et 2, qui dépendent chacune de leur côté d'une troisième, notée 3. Les dépendances 1 et 2 ont chacune des critères différents quant à la version de la dépendance 3 (Figure 1). Un système naïf arrêterait la résolution ici – 1 et 2 dépendent de deux versions différentes de 3, on ne peut pas les concilier. Conséquence : si 1 produit un objet venant de 3, cet objet ne pourra pas être utilisé par 2 qui utilise une autre version de 3. Ce cas de figure, appelé dépendances en doublon, est un comportement indésirable d'un système de résolution de dépendances.

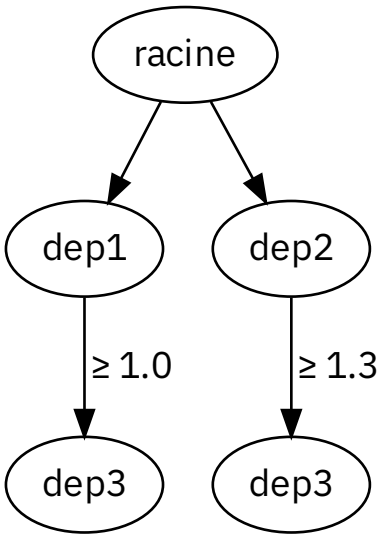

Figure 1 : Arbre original

Aussi, un système plus malin, prenant en compte le semver, pourra remarquer que le critère ≥1.3 est un sous-ensemble de ≥1.0, et pourra unifier les deux critères vers le plus spécifique des deux, pour ne plus avoir qu'une

[8] (Preston-Werner, Gestion sémantique de version, 2013)

seule instance de 3 dans le graphe (Figure 2). Ainsi, 1 et 2 utiliseront la même version de 3, et il n'y a plus de problèmes de compatiblité.

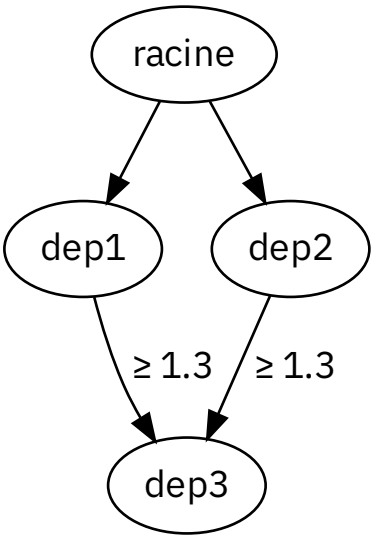

Figure 2 : Arbre réduit

Dans ce raisonnement, la réduction a été rendue possible par une hypothèse cruciale : le fait que ≥1.3 soit un sous-ensemble de ≥1.0. C'est une garantie que fournit semver, mais l'Histoire est remplie d'exemples de logiciels ne respectant pas ce principe[9]. Souvent délibérément (numéro de version à usage plus commercial que technique, contraintes d'ordre légal[10]), mais parfois involontairement (changement accidentellement cassant). Il y a également une inertie terriblement humaine liée aux numéros majeurs de version : il est courant que des développeurs rechignent à passer à une nouvelle version majeure, dans des cas où des changements *techniquement* majeurs sont publiés mais où ça ne justifie pas, à leurs yeux, d'incrémenter le numéro majeur[11].

La réalisation d'un outil permettant d'automatiser le processus de sélection de version s'impose naturellement comme solution idéale.

b. Étude de l'existant

Cargo-breaking n'est ni le premier ni le seul sur ce terrain : rust-semverver (Babushkin, 2017) et plus récemment cargo-semver-check (Gruevski, 2022) remplissent un rôle similaire – bien qu'ayant une finalité différente : leur but est seulement de détecter les changements, là où cargo-breaking calcule en plus le numéro de version suivante.

Ces trois outils diffèrent néanmoins sur leur fonctionnement. Là où la comparaison de texte brut (« diff ») est un problème pour ainsi dire résolu (Miller & Myers, 1985; Myers, 1986), et dont la mise en pratique est à la base de systèmes modernes tels que Git, notre besoin réside ici dans la comparaison de code en tant que tel, afin d'observer par exemple non pas des ajouts de caractères ou d'octets mais d'objets de langage tels que des fonctions ou des types.

[9] Les premières versions de Java étaient numérotés en 1.A, où A correspondait au numéro majeur. Il y a ainsi eu des ajouts cassants en Java 1.2, 1.4 et 1.5 (ajout de mots-clés réservés, cassant le code les ayant utilisés comme identificateurs auparavant). Depuis la version 5, le « 1. » est omis pour les noms commerciaux, mais les versions internes continuent malgré tout de commencer par 1.
Les versions du noyau NT sont également réputées pour être particulièrement chaotiques ; bien que la surface d'API publique soit supposément stable, il y a par exemple eu des suppressions (*de facto* cassantes) entre Vista (6.0), 7 (6.1), 8 (6.2) et 8.1 (6.3), alors que le numéro majeur ne changeait pas.
[10] (Siegel, 2022)
[11] Un problème tellement important que l'auteur du semver y a dédié un article il y a quelques mois (Preston-Werner, 2022). Encore un coup de l'islamo-gauchisme.

L'approche initiale, utilisée dans la première version du projet, consistait à effectuer la comparaison au niveau de l'AST (Abstract Syntax Tree, arbre de syntaxe abstrait) du programme, à l'aide de la librairie syn qui permet d'analyser du code Rust et d'en générer un arbre.

La comparaison d'arbres de syntaxe est étudiée depuis des années pour des usages aussi divers que variés, allant de l'analyse d'évolution de code source (Neamtiu, Foster, & Hicks, 2005) à la détection automatisée de plagiat (Feng, Cui, & Xia, 2013), et des algorithmes très rapides permettent aujourd'hui d'effectuer une telle comparaison avec un coût mémoire et temporel réduit.

Une fois les arbres générés, ils étaient comparés et les éléments supprimés, modifiés ou ajoutés étaient accumulés afin de déterminer si des changements cassants étaient présents.

Cette approche présente un problème important : la charge de l'analyse du code, pour un langage qui peut évoluer au fil du temps. En effet, des fonctionnalités peuvent apparaître, perturbant l'analyse syntaxique, et on se retrouve dans tous les cas à effectuer du travail que le compilateur pourrait effecuter à notre place. Ainsi, une deuxième approche a été mise sur la table : se brancher directement dans la pipeline[12] du compilateur pour récupérer le code sous une forme plus facilement analysable. On évite l'analyse syntaxique, on gagne beaucoup de temps, mais on a un nouveau problème, similaire au précédent : on est bloqué sur une version précise du compilateur. L'API interne n'est en effet pas stable et peut changer à tout moment ; une mise à jour du compilateur peut donc casser notre code.

c. Travail effectué

Ma nouvelle approche, celle adoptée pour le projet, découle d'un adage courant en programmation : on est jamais mieux servi que par les autres (Vaillant, 2017; Scott, 2013; Lester, 2019).

Parmi la panoplie d'outils associés au compilateur Rust, en figure un bien pratique : rustdoc. Il s'agit du générateur de documentation, qui sert à générer un ensemble de pages Web (au format HTML) reliées entre elles, listant la documentation des types et fonctions du projet[13]. Cet outil dépend du compilateur et est mis à jour en tandem avec ce dernier. Il se trouve qu'en plus de l'HTML, cet outil peut émettre une sortie au format JSON, facilement lisible par un programme, contenant en fait tout ce dont on a besoin pour analyser la surface publique de l'API d'une librairie. C'est donc sur cette sortie qu'on base désormais la comparaison, avec au final très peu de traitement préalable.

[12] Un compilateur, de nos jours, est une œuvre d'ingénierie qui peut se révéler particulièrement complexe. La compilation se divise généralement en de nombreuses phases : analyse lexicale, syntaxique, sémantique, lowering / désugarisation, et bien d'autres. On appelle « pipeline » le flot de données du compilateur, à travers ces différentes phases.
[13] (Rust Foundation, 2022)

Le résultat est immédiat : plus de problèmes liés aux évolutions du langage, vu que le compilateur peut être mis à jour ; plus de problèmes liés à une dépendance particulière sur une version, car on traite la sortie JSON qui, elle, ne change pas[14]. Le seul inconvénient réside justement dans le fait d'exécuter le compilateur (à travers rustdoc) pour effectuer la comparaison, car la chaîne de compilation est lente à démarrer et à l'exécution[15]. C'est un compromis, qui se vaut ici car cargo-breaking n'est pas voué à être exécuté souvent.

En tout, mon travail sur cargo-breaking aura abouti à un backend stable reposant sur rustdoc, et aura permis de trouver des bugs dans le compilateur Rust[16] et dans une librairie de compilation très utilisée, cc-rs[17].

Subsidiairement, j'ai également amélioré (en pratique, réécrit de zéro) la documentation du programme, qui en explique le fonctionnement (qui n'était logiquement plus d'actualité).

[14] Assez peu, en tout cas, pour qu'on n'ait pas à s'en soucier.
[15] Il s'agit d'un défaut reconnu des outils de compilation du Rust, langage somme toute très complexe, et dont la compilation a toujours été plus lente que d'autres langages de la même catégorie. À l'autre bout du spectre, on trouve le Pascal, qui a dès le départ été conçu pour être compilable en une passe.
[16] (Niget, ICE when running rustdoc in JSON mode on clap, 2022)
[17] (Niget, Fix location of atlmfc directory on MSVC 15+, 2022)

Projet 2 – restest

a. Contexte

IOmentum développe beaucoup d'applications Web et d'APIs en Rust, et il est intéressant dans tout projet de ce type de pouvoir facilement et efficacement tester la partie backend, autrement dit les endpoints qui seront appelés par les clients.

Pour une API suivant le modèle SOAP, ce n'est pas différent des tests pour le reste du code, vu que le client est d'emblée exposé comme un objet programmatique, mais pour des APIs HTTP classiques, suivant simplement le modèle du REST moderne[18] la question est très différente. Les tests peuvent porter sur le statut HTTP, sur le corps de réponse – qui peut lui-même contenir différents types d'objets. Les langages dynamiques, tels que le JavaScript ou le Python, sont bien équipés pour consommer de telles APIs, dont le schéma n'est pas forcément spécifié ou fixe, mais les systèmes de typage statique posent plus de contraintes.

Il existe donc un besoin, pour les développeurs de services Web en Rust, d'un outil permettant de tester, depuis le code, des endpoints d'API, à la manière de tests unitaires et de façon à pouvoir être utilisé comme un test « normal ».

b. Étude de l'existant

La librairie la plus utilisé en Rust pour développer des APIs REST est Warp, et cette dernière possède un système de tests assez complet (Zupan, 2020). D'autres librairies offrent des outils similaires, mais à chaque fois spécifiques et conçues pour être utilisés dans la même base de code. En pratique, seule une poignée de librairies permettent de tester « en brut » des endpoints HTTP, sans dépendre de celle utilisée pour implémenter lesdits endpoints.

IOmentum a donc commencé en 2021 le développement de restest, afin de faciliter le test des projets internes en premier lieu, mais comme pour cargo-breaking, sous forme d'un outil open-source mis à la disposition de la communauté.

c. Travail effectué

Une partie bien plus courte de mon stage a été dédiée à restest qu'à cargo-breaking – en pratique, pour la simple raison que restest était à mon arrivée déjà fonctionnel, là où cargo-breaking était en suspens depuis plusieurs mois en raison des problèmes de backend évoqués précédemment.

[18] REST fait historiquement référence à une séries de conventions de conception d'API (Fielding, 2000), dont la plus grande partie n'a pas survécu à l'épreuve du temps. De nos jours, REST ne désigne guère plus qu'une API prenant des paramètres via un corps HTTP et répondant en JSON. Ce glissement de définition a été étudié et critiqué, notamment par l'inventeur du terme REST (Fielding, REST APIs must be hypertext-driven, 2008). Toutefois, la définition originale du REST a elle-même fait l'objet de critiques, notamment quant à sa réalisabilité (Begemann, 2018).

Ainsi, j'ai surtout passé du temps à communiquer avec les utilisateurs existants de restest (membres de l'équipe IOmentum) pour analyser leurs besoins et les fonctionnalités manquantes qui pourraient les aider dans leur vie quotidienne.

J'ai notamment ajouté le support des verbes HTTP DELETE et PUT (suppression et modification sur place), et des corps de réponse vides (cas rare ; se produit par exemple dans le cas d'une suppression réussie, auquel cas on va simplement renvoyer le code 200 sans données).

Planning

Du fait du mode de fonctionnement de l'entreprise, je n'ai pas eu de planning strict lors de mon stage. En effet, en dehors d'une réunion de suivi hebdomadaire avec un responsable de projet, généralement le vendredi, j'étais libre de répartir mon temps de travail sur les différents projets qui m'étaient affectés. En pratique, je n'ai eu à travailler sur restest que les quelques premières semaines du stage, et le reste de mon temps a été dédié à cargo-breaking.

J'aimerais de tout cœur remplir un peu plus cette feuille, mais c'est vraiment tout ce qu'il y a à dire sur le planning de mon stage.

BIBLIOGRAPHIE

Babushkin, I. (2017). *rust-semverver*. Récupéré sur https://github.com/rust-lang/rust-semverver

Begemann, O. (2018). *Roy Fielding's REST dissertation*. Récupéré sur https://oleb.net/2018/rest/

Clark, L. (2022). *Oracle contracts and pricing a 'challenge', says Gartner*. Récupéré sur The Register: https://www.theregister.com/2022/04/21/oracle_contracts_pricing_gartner/

Feng, J., Cui, B., & Xia, K. (2013). A Code Comparison Algorithm Based on AST for Plagiarism Detection. *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, (pp. 393-397). Récupéré sur https://sci.bban.top/pdf/10.1109/EIDWT.2013.73.pdf

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Récupéré sur https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fielding, R. T. (2008). *REST APIs must be hypertext-driven*. Récupéré sur https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Gruevski, P. (2022). *cargo-semver-check*. Récupéré sur GitHub: https://github.com/obi1kenobi/cargo-semver-check

IOmentum. (2022). Récupéré sur GitHub: https://github.com/iomentum

Lester, A. (2019). *9 Things You Should Never Code Yourself*. Récupéré sur New Relic: https://newrelic.com/blog/best-practices/7-things-never-code

LKML. (2021). *[PATCH 00/13] [RFC] Rust support*. Récupéré sur Linux Kernel Mailing List: https://lkml.org/lkml/2021/4/14/1023

Miller, W., & Myers, E. W. (1985). A file comparison program. *Software: Practice and Experience, 15*, 1025-1040. Récupéré sur https://publications.mpi-cbg.de/Miller_1985_5440.pdf

Myers, E. W. (1986). An O(ND) Difference Algorithm and Its Variations. *Algorithmica, 1*, 251-266. Récupéré sur http://www.xmailserver.org/diff2.pdf

Neamtiu, I., Foster, J. S., & Hicks, M. (2005). Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Softw. Eng. Notes, 30*(4), 1-5. Récupéré sur https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.3726&rep=rep1&type=pdf

Niget, T. (2022). *Fix location of atlmfc directory on MSVC 15+*. Récupéré sur GitHub: https://github.com/rust-lang/cc-rs/pull/699

Niget, T. (2022). *ICE when running rustdoc in JSON mode on clap*. Récupéré sur GitHub: https://github.com/rust-lang/rust/issues/97432

Preston-Werner, T. (2013). *Gestion sémantique de version*. Récupéré sur https://semver.org/lang/fr/

Preston-Werner, T. (2022). *Major Version Numbers are Not Sacred*. Récupéré sur https://tom.preston-werner.com/2022/05/23/major-version-numbers-are-not-sacred.html

Rust Foundation. (2022). *What is Rustdoc?* Récupéré sur https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html

Scott, T. (2013). The Problem with Time & Timezones. Récupéré sur https://www.youtube.com/watch?v=-5wpm-gesOY

Siegel, D. (2022). *Lawyer says bumping software's minor version would cost $2k to "refile" copyright paperwork. Is this typical?* Récupéré sur https://law.stackexchange.com/a/82651

Stack Exchange. (2022). *Stack Overflow Developer Survey*. Récupéré sur https://survey.stackoverflow.co/2022/

TechEmpower. (2022). *Web Framework Benchmarks, Round 21*. Récupéré sur https://www.techempower.com/benchmarks/#section=data-r21

Vaillant, L. (2017). *Rolling Your Own Crypto*. Récupéré sur https://loup-vaillant.fr/articles/rolling-your-own-crypto

ZDnet. (2019). *Microsoft: 70 percent of all security bugs are memory safety issues*. Récupéré sur https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Zupan, M. (2020). *End-to-end testing for Rust web services*. Récupéré sur LogRocket: https://blog.logrocket.com/end-to-end-testing-for-rust-web-services/