Table of Contents

TODO: inclure ce document dans le rapport plus large

todo: ajouter screenshots de PLX

todo: parler de rustlings et d'autres projets autour

Note: ce brouillon demande encore de nombreuses finitions et relecture avant d'être rendu le 23 mai.

Dictionnaire

- `Cargo.toml` définit les dépendances (les crates) et leur versions minimum à inclure dans le projet, équivalent du `package.json` de NPM
- `crate` : la plus petite unité de compilation avec cargo, concrètement chaque projet contient un ou plusieurs dossiers avec un `Cargo.toml`
- `crates.io` : le registre officiel des crates publiée pour l'écosystème Rust, l'équivalent de `npmjs.com` pour l'écosystème Javascript, ou `mvnrepository.com` pour Java

Etat de l'art

Format de données existant orienté humainement éditable

Ces recherches ignorent les formats de données largement supporté et répandu tel que le XML, JSON, YAML et TOML. Ils sont tout à fait adapter pour des configurations, de la sérialisation et de l'échange de donnée et sont pour la plupart facilement lisible. Cependant la quantité de séparateurs et délimiteurs en plus du contenu qu'ils n'ont pas été optimisé pour la rédaction par des humains. Le YAML et le TOML, bien que plus léger que le JSON, inclue de nombreux types de données autre que les strings, des tabulations et des guillemets, ce qui rend la rédaction plus fastidieuse qu'en Markdown.

On cherche quelque chose du niveau de simplicité du Markdown en terme de rédaction, mais avec une validation poussée customisable par le projet qui définit le schéma.

TODO: continuer markdown inspiration + besoin

Ces recherches se focalisent sur les syntaxes qui ne sont pas spécifique à un domaine ou qui seraient complètement déliée de l'informatique ou de l'éducation. Ainsi, l'auteur ne présente pas Cooklang (1), qui se veut une langage de balise pour les recettes de cuisines, même si l'implémentation du parseur en Rust (2) pourra servir pour d'autres recherches. On ignore également les projets qui créent une syntaxe très proche du Rust, comme la Rusty Object Notation (RON) (3), de par leur nécessité de connaître un peu la syntaxe du Rust et surtout parce qu'elle ne simplifie pas vraiment l'écriture comparé à du YAML. On ignore également les projets dont la spécification ou l'implémentation est en état de « brouillon » et n'est pas encore utilisable en production.

Contrairement aux languages de programmation qui existent par centaines, les syntaxes de ce genre ne sont pas monnaies courantes. Différentes manières de les nommer existent: language de balise (markup language), format de donnée, syntaxes, langage de donnée, language spécifique à un domaine (de l'anglais Domain Specific Language - DSL), … Les mots-clés utilisés suivants ont été utilisés sur Google, la barre de recherche de Github.com et de crates.io: `data format`, `human friendly`, `human writable`, `human readable`.

KHI - Le langage de données universel

D'abord nommée UDL (Universal Data Language) (4), cette syntaxe a été inventée pour mixer les possibilités du JSON, YAML, TOML, XML, CSV et Latex, afin de supporter toutes les structures de données modernes. Plus concrètement le markup, les structs, les listes, les tuples, les tables/matrices, les enums, les arbres hiérarchiques sont supportés. Les objectifs sont la polyvalence, un format source (fait pour être rédigé à la main), l'esthétisme et la simplicité.

```
{article}:
uuid: 0c5aacfe-d828-43c7-a530-12a802af1df4
type: chemical-element
key: aluminium
title: Aluminium
description: The <@element>:{chemical element} aluminium.
tags: [metal; common]

{chemical-element}:
symbol: Al
number: 13
stp-phase: <Solid>
melting-point: 933.47
boiling-point: 2743
density: 2.7
electron-shells: [2; 8; 3]

{references}:
wikipedia: \https://en.wikipedia.org/wiki/Aluminium
snl: \https://snl.no/aluminium
```

Snippet 1. – Un exemple simplifié de KHI de leur README (5), décrivant un exemple d'article d'encyclopédie.

Une implémentation en Rust est proposée (6). Son dernier commit sur ces 2 repositorys date du 11.11.2024, le projet a l'air de ne pas être fini au vu des nombreux `todo!()` présent dans le code. La large palette de structures supportées implique une charge mentale additionnelle pour se rappeler, ce qui en fait une mauvaise option pour PLX.

Bitmark - le standard des contenus éducatifs digitaux

Bitmark est un standard open-source, qui vise à uniformiser tous les formats de données utilisés pour décrire du contenu éducatif digital sur les nombreuses plateformes existantes (7). Cette diversité de formats rend l'interropérabilité très difficile et freine l'accès à la connaissance et restreint les créateurs de contenus et les éditeurs dans les possibilités de migration entre plateformes. La stratégie est de définir un format basé sur le contenu (Content-first) plus que basé sur son rendu (layout-first) permettant un affichage sur tous type d'appareils incluant les appareils mobiles (7). C'est la Bitmark Association en Suisse à Zurich qui développe ce standard, notamment à travers des Hackatons organisés en 2023 et 2024 (8).

Le standard permet de décrire du contenu statique et interactif, comme des articles ou des quiz de divers formats. 2 formats équivalents sont définis: le bitmark markup language et le bitmark JSON data model (9)

La partie quizzes du standard inclut des textes à trous, des questions à choix multiple, du texte à surligner, des essais, des vrai/faux, des photos à prendre ou audios à enregister et de nombreux autres type d'exercices.

```
[.multiple-choice-1]
[!What color is milk?]
[?Cows produce milk.]
[+white]
[-red]
[-blue]
```

Snippet 2. – Un exemple de question à choix multiple tiré de leur documentation (10). L'option correcte `white` est préfixée par `+` et les 2 autres options incorrectes par `-`. Plus haut, `[! ... ]` décrit une consigne, `[? ... ]` décrit un indice.


```
{
    "markup": "[.multiple-choice-1]\n[!What color is milk?]\n[+white]\n[-red]\n[-blue]",
    "bit": {
        "type": "multiple-choice-1",
        "format": "text",
        "item": [],
        "instruction": [ { "type": "text", "text": "What color is milk?" } ],
        "body": [],
        "choices": [
            { "choice": "white", "item": [], "isCorrect": true },
            { "choice": "red", "item": [], "isCorrect" : false },
            { "choice": "blue", "item": [], "isCorrect" : false }
        ],
        "hint": [ { "type": "text", "text": "Cows produce milk." } ],
        "isExample": false,
        "example": []
    }
}
```


Snippet 3. – Equivalent de Snippet 2 dans le Bitmark JSON data model (10)

Open Taskpool, projet qui met à disposition des exercices d'apprentissage de langues (11), fournit une API JSON utilisant le Bitmark JSON data model.

Demander à Open Taskpool des exercices d'allemand vers anglais autour du mot `school` de format `cloze` (texte à trou), se fait avec cette simple requête: `https://taskpool.taskbase.com/exercises?translationPair=de→en&word=school&exerciseType=bitmark.cloze`.

```
...
"cloze": {
    "type": "cloze",
    "format": "text",
    "instruction": "Gegeben: \"Früher war hier eine Schule.\", schreiben Sie das fehlende
Wort",
    "body": [
        { "type": "text", "text": "There used to be a " },
        {
            "type": "gap",
            "solutions": [ "school" ],
            "answer": { "text": "" }
        },
        { "type": "text", "text": " here." }
    ]
},
...
```

Snippet 4. – Extrait simplifié de la réponse JSON, respectant le standard Bitmark (12). La phrase `There used to be a ___ here.` doit être complétée par le mot `school` en s'aidant du texte en allemand.

Un autre exemple d'usage se trouve dans la documentation de Classtime (13), on voit que le système de création d'exercices est basé sur des formulaires. Ces 2 exemples donnent l'impression que la structure JSON est plus utilisée que le markup. Au vu de tous séparateurs et symboles de ponctuations à se rappeler, la syntaxe n'a peut-être pas été imaginée dans le but d'être rédigée à la main directement. Finalement, Bitmark ne spécifie pas de type d'exercices programmation nécessaire à PLX.

NestedText — Un meilleur JSON

NestedText se veut human-friendly, similaire au JSON mais pensé pour être facile à modifier et visualiser par les humains. Le seul type de donnée scalaire supporté est la chaîne de caractères, afin de simplifier la syntaxe et retirer le besoin de mettre des guillemets. La différence avec le YAML, en plus des types de données restreint est la facilité d'intégrer des morceaux de code sans échappements ni guillemets, les caractères de données ne peuvent pas être confondus avec NestedText (14).

```
Margaret Hodge:
    position: vice president
    address:
        > 2586 Marigold Lane
        > Topeka, Kansas 20682
    phone: 1-470-555-0398
    email: margaret.hodge@ku.edu
    additional roles:
        - new membership task force
        - accounting task force
```

Snippet 5. – Exemple tiré de leur README (14)

Ce format a l'air assez léger visuellement et l'idée de faciliter l'intégration de blocs multi-lignes sans contraintes de charactères réservée serait utile à PLX. Cependant, tout comme le JSON la validation du contenu n'est pas géré directement par le parseur mais par des librairies externes qui vérifient le schéma (15). De plus, l'implémentation officielle est en Python et il n'y a pas d'implémentation Rust disponible; il existe une crate réservée mais vide (16).

SDLang - Simple Declarative Language

SDLang se définit comme « une manière simple et concise de représenter des données textuellement. Il a une structure similaire au XML: des tags, des valeurs et des attributs, ce qui en fait un choix polyvalent pour la sérialisation de données, des fichiers de configuration ou des langages déclaratifs. » (Traduction personnelle de leur site web (17)). SDLang définit également différents types de nombres (32bit, 64bit, entier, flottant, …), 4 valeurs de booléans (`true` , `false` , `on` , `off`) comme en YAML, différents formats de dates et un moyen d'intégrer des données binaires encodées en Base64.

```
// This is a node with a single string value
title "Hello, World"

// Multiple values are supported, too
bookmarks 12 15 188 1234

// Nodes can have attributes
author "Peter Parker" email="peter@example.org" active=true

// Nodes can be arbitrarily nested
contents {
  section "First section" {
    paragraph "This is the first paragraph"
    paragraph "This is the second paragraph"
  }
}

// Anonymous nodes are supported
"This text is the value of an anonymous node!"

// This makes things like matrix definitions very convenient
matrix {
  1 0 0
  0 1 0
  0 0 1
}
```

Snippet 6. – Exemple tiré de leur site web (17)

Ce format s'avère plus intéressant que les précédents de part le faible nombre de caractères réservés et la densité d'information: avec l'auteur décrit par son nom, email et un attribut booléan sur une seule ligne ou la matrice de 9 valeurs définie sur 5 lignes. Il est cependant regrettable de voir de les strings doivent être entourées de guillemets et les textes sur plusieurs lignes doivent être entourés de backticks `` ` ``. De même la définition de la hiéarchie d'objets définis nécessite d'utiliser une paire `{ }`, ce qui rend la rédaction un peu plus lente.

KDL - Cuddly Data language

```
package {
  name my-pkg
  version "1.2.3"

  dependencies {
    // Nodes can have standalone values as well as
    // key/value pairs.
    lodash "^3.2.1" optional=#true alias=underscore
  }

  scripts {
    // "Raw" and dedented multi-line strings are supported.
    message """
      hello
      world
      """
    build #"""
      echo "foo"
      node -c "console.log('hello, world!');"
      echo "foo" > some-file.txt
      """#
  }
}
```

Snippet 7. – Exemple simplifié tiré de leur site web (18)

Est-ce que cela paraît proche de SDLang vu précédemment ? C'est normal puisque KDL est basé sur SDLang avec quelques améliorations. Celles qui nous intéressent concernent la possibilité d'utiliser des guillemets pour les strings sans espace (`person name=Samuel` au lieu de `person name="Samuel"`). Cette simplification n'inclue malheureusement des strings multilines, qui demande d'être entourée par `"""`. Le problème d'intégration de morceaux de code est également relevé, les strings brutes sont supportées entre `#` sur le mode une ou plusieurs lignes, ainsi pas d'échappements des backslashs à faire par ex.

En plus des autres désavantages restant de hiéarchie avec `{ }` et guillemets, il reste toujours le problème des types de nombres qui posent soucis avec certaines strings si on ne les entoure pas de guillemets. Par exemple ce numéro de version `version "1.2.3"` a besoin de guillemets sinon `1.2.3` est interprété comme une erreur de format de nombre à virgule.

Conclusion

En conclusion, au vu du nombre de tentatives/variantes trouvées, on voit que la verbosité des formats largement répandu du XML, JSON et même du YAML est un problème qui ne touche pas que l'auteur. Le gain de verbosité des syntaxes listées est réel mais reste ciblé sur un usage plus avancé de structure de données et types variés. L'auteur pense pouvoir proposer une approche encore plus légère et plus simple, inspirée du style du Markdown en évitant une partie des charactères non explicites.

TODO finish + merge intro above

Librairies existantes de parsing en Rust

Après s'être intéressé aux syntaxes existantes, nous nous intéressons maintenant aux solutions existantes pour simplifier ce parsing de cette nouvelle syntaxe en Rust.

Après quelques recherches avec le tag `parser` sur crates.io (19), j'ai trouvé la liste de librairies suivantes:

- `winnow` (20), fork de `nom`, utilisé notamment par le parseur Rust de KDL (21)
- `nom` (22), utilisé notamment par `cexpr` (23)
- `pest` (24)
- `combine` (25)
- `chumsky` (26)

A noter aussi l'existance de la crate `serde`, un framework de sérialisation et desérialisation très populaire dans l'écosystème Rust (selon lib.rs (27)). Il est notamment utilisé pour les parseurs JSON et TOML. Ce n'est pas une librairie de parsing mais un modèle de donnée basée sur les traits de Rust pour faciliter son travail. Au vu du modèle de données de Serde (28), qui supporte 29 types de données, ce projet paraît à l'auteur apporter plus de complexités qu'autre chose pour trois raisons:

- Seulement les strings, listes et structs sont utiles pour PLX. Par exemple, les 12 types de nombres sont inutiles à différencier et seront propre au besoin de la variante.
- La sérialisation (struct Rust vers syntaxe DY) n'est pas prévue, seulement la desérialisation est utile.
- Le mappage des préfixes et flags par rapport aux attributs des structs Rust qui seront générées, n'est pas du 1:1, cela dépendra de la structure définie pour la variante de PLX.

Après ces recherches et quelques essais avec `winnow`, l'auteur a finalement décidé qu'utiliser une librairie était trop compliqué pour le projet et que l'écriture manuelle d'un parseur ferait mieux l'affaire. La syntaxe DY est relativement petite à parser, et sa structure légère et souvent implicite rend compliqué l'usage de librairies pensées pour des langages de programmation très structuré.

Par exemple, une simple expression mathématique `((23+4) * 5)` paraît idéale pour ces outils, les débuts et fin sont claires, une stratégie de combinaisons de parseurs fonctionnerait bien pour les expressions parenthésées, les opérateurs et les nombres. Elles semble bien adapter à exprimer l'ignorance des espaces, extraire les nombres tant qu'il contiennent des chiffres, extraires des opérateurs et les 2 opérandes autour…

Pour DY, l'aspect multilignes et qu'une partie des préfixes optionnel, complique l'approche de définir le début et la fin et d'appeler combiner récursivement des parseurs comme on ne sait pas facilement où est la fin.

```
exo Dog struct
Consigne très longue

en *Markdown*
sur plusieurs lignes

xp 20
checks
 ...
```

Snippet 8. – Exemple d'un début d'exercice de code, on voit que la consigne se trouve après la ligne `exo` et continue sur plusieurs lignes jusqu'à qu'on trouve un autre préfixe (ici `xp` qui est optionnel ou alors `checks`). size

Les serveurs de language et librairies Rust existantes

Une part importante du support d'un langage dans un éditeur, consiste en l'intégration des erreurs, l'auto-complétion, les propositions de corrections, des informations au survol… et de nombreuses fonctionnalités qui améliorent la compréhension ou l'interaction. L'avantage d'avoir les erreurs de compilation directement soulignées dans l'éditeur, permet de voir et corriger immédiatement les problèmes sans lancer une compilation manuelle dans une interface séparée.

Contrairement au surlignage de code, ces fonctionnalités demandent une compréhension beaucoup plus fine, ils sont implémentés dans des processus séparés de l'éditeur (aucun langage de programmation n'est ainsi imposé). Ces processus séparés sont appelés des serveurs de langage (language server en anglais). Les éditeurs qui intègre Tree-Sitter développe un client LSP qui se charge de lancer ce serveur, de lancer des requêtes et d'intégrer les données des réponses dans leur interface visuelle.

La communication entre l'éditeur et un serveur de langage démarré pour le fichier en cours, se fait via le `Language Server Protocol (LSP)`. Ce protocole inventé par Microsoft pour VSCode, résoud le problème des développeurs de langages qui doivent supporter chaque éditeur de code indépendamment avec des APIs légèrement différentes pour faire la même chose. Le projet a pour but également de simplifier la vie des nouveaux éditeurs pour intégrer rapidement des dizaines de langages via ce protocole commun et standardisé (29).

Fig. 1. – Exemple d'auto-complétion dans Neovim, générée par le serveur de language `rust-analyzer` sur l'appel d'une méthode sur les `&str`

Les points clés du protocole à relever sont les suivants:

- **JSON-RPC** (JSON Remote Procedure Call) est utilisé comme format de sérialisation des requêtes. Similaire au HTTP, il possède des entêtes et un corps. Ce standard définit quelques structures de données à respecter. Une requête doit contenir un champ `jsonrpc`, `id`, `method` et optionnelement `params` (30). Il est possible d'envoyer une notification (requête sans attendre de réponse). Par exemple, le champ `method` va indiquer l'action qu'on tente d'appeler, ici une des fonctionnalités du serveur. Voir Snippet 9
- Un serveur de langage n'a pas besoin d'implémenter toutes les fonctionnalités du protocole. Un système « Capabilities » est défini pour annoncer les méthodes implémentées (31).

- Le transport des messages JSON-RPC peut se faire en `stdio` (flux standard entrée et sorties), sockets TCP ou même en HTTP.

```
Content-Length: ...\r\n
\r\n
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "textDocument/completion",
    "params": {
        ...
    }
}
```

Snippet 9. – Exemple de requête en JSON-RPC envoyé par le client pour demander des propositions d'auto-complétion à une position de curseur données. Tiré de la spécification (32)

Quelques exemples de serveurs de langages implémentés en Rust
- `tinymist`, serveur de langage de Typst (système d'édition de document, utilisé pour la rédaction de ce rapport)
- `rust-analyzer`, serveur de language officiel du langage Rust
- `asm-lsp` (33), permet d'inclure des erreurs dans du code assembleur

D'autres exemples de serveurs de langages implémentés dans d'autres langages
- `jdtls` le serveur de language pour Java implémenté en Java (34)
- `tailwindcss-language-server`, le serveur de langage pour le framework CSS TailwindCSS, implémenté en TypeScript (35)
- `typescript-language-server` et pour finir celui pour TypeScript, implémenté en TypeScript également (36)
- et beaucoup d'autres projets existent…

Une crate commune à plusieurs projets est `lsp-types` (37) qui définit les structures de données, comme `Diagnostic`, `Position`, `Range`. Ce projet est utilisé par `lsp-server`, `tower-lsp` et d'autres (38).

Adoption

Selon la liste sur le site de la spécification (39), la liste des IDE qui supportent le LSP est longue: Atom, Eclipse, Emacs, GoLand, Intellij IDEA, Helix, Neovim, Visual Studio, VSCode bien sûr et d'autres. La liste des serveurs LSP (40) quand à elle, contient plus de 200 projets, dont 40 implémentés en Rust! Ce large support et ces nombreux exemples va grandement faciliter le développement de ce serveur de language et son intégrations dans différents IDE.

Librairies disponibles

En cherchant à nouveau sur `crates.io` sur le tag `lsp`, on trouve différent projets dont `async-lsp` (41) utilisée dans `nil` (42) (un serveur de langage pour le système de configuration de NixOS) et de la même auteure.

Le projet `tinymist` a extrait une crate `sync-ls`, mais le README déconseille son usage et conseille `async-lsp` à la place (43). En continuant la recherche on trouve encore un autre `tower-lsp` et un fork `tower-lsp-server` (44)… `rust-analyzer` a également extrait une crate `lsp-server`.

Choix final

L'auteur travaillant dans Neovim, l'intégration se fera en priorité dans Neovim pour ce travail. L'intégration dans VSCode pourra être fait dans le futur et devrait être relativement simple.

Les 2 projets les plus utilisés (en terme de reverse dependencies sur crates.io) sont `lsp-server` (45) (56) et `tower-lsp` (85) (46). L'auteur a choisi d'utiliser la crate `lsp-server` étant développé par la communauté Rust, la probabilité d'une maintenance long-terme est plus élevée, et le projet `tower-lsp` est basée sur des abstractions asynchrones, l'auteur préfère partir sur la version synchrone pour simplifier l'implémentation.

Cette partie est un nice-to-have, l'auteur espère avoir le temps de l'intégrer dans ce travail. Après quelques heures sur le POC suivant, on voit cela semble être assez facile et la possibilité d'ajouter progressivement le support de fonctionnalités est aussi un atout.

POC sur lsp-server

TODO: bien d'avoir cette section séparée pour ce POC ?

L'auteur a modifié et exécuté l'exemple de `goto_def.rs` fourni par la crate `lsp-server` (47). Il a aussi créé un script `demo.fish` permettant de lancer la communication en stdin et attendre entre chaque requête. Cet exemple minimaliste mais clair démontre la communication qui se produit quand on clique sur un `Aller à la définition` dans un IDE. L'IDE va lancer le serveur de langage associé au fichier édité en lancant simplement le processus et en communication via les flux standards. Il y a d'abord une phase d'initialisation et d'annonces des capacités puis l'IDE peut envoyer des requêtes.

```
CLIENT: Content-Length: 85

{"jsonrpc": "2.0", "method": "initialize", "id": 1, "params": {"capabilities": {}}}
SERVER: Content-Length: 78

{"jsonrpc":"2.0","id":1,"result":{"capabilities":{"definitionProvider":true}}}
CLIENT: Content-Length: 59

{"jsonrpc": "2.0", "method": "initialized", "params": {}}

CLIENT: Content-Length: 167

{"jsonrpc": "2.0", "method": "textDocument/definition", "id": 2, "params":
{"textDocument": {"uri": "file:///tmp/test.rs"}, "position": {"line": 7, "character": 23}}}
SERVER: Content-Length: 144

{"jsonrpc":"2.0","id":2,"result":[{"range":{"end":{"character":25,"line":3},"start":{"char
acter":12,"line":3}},"uri":"file:///tmp/another.rs"}]}
CLIENT: Content-Length: 67

{"jsonrpc": "2.0", "method": "shutdown", "id": 3, "params": null}
SERVER: Content-Length: 38

{"jsonrpc":"2.0","id":3,"result":null}
CLIENT: Content-Length: 54

{"jsonrpc": "2.0", "method": "exit", "params": null}
```

Fig. 2. – Exemple de discussion en LSP une demande de `textDocument/definition`, output de `fish demo.fish` dans le dossier `pocs/lsp-server-demo`.
Les lignes après `CLIENT:` sont envoyés en stdin et celles après `SERVER` sont reçues en stdout.

L'initialisation nous montre que le serveur se présente comme supportant uniquement les « aller à la définition » (go to definition) puisque `definitionProvider` est à `true`. Le client envoie ensuite une demande de `textDocument/definition`, en précisant que celle-ci doit être donnée sur le symbole dans fichier `/tmp/test.rs` sur la ligne 7 au charactère 23.

L'auteur a codé en dur une liste de `Location` (positions dans le code pour cette définition), dans `/tmp/another.rs` sur la `Range` de la ligne 3 du charactère 12 à 25. Une fois la réponse envoyée, le client demande au serveur de s'arrêter.

Le code qui gère cette requête du type `GotoDefinition` se présente ainsi.

```
match cast::<GotoDefinition>(req) {
    Ok((id, params)) => {
        let locations = vec![Location::new(
            Uri::from_str("file:///tmp/another.rs")?,
            Range::new(Position::new(3, 12), Position::new(3, 25)),
        )];
        let result = Some(GotoDefinitionResponse::Array(locations));
        let result = serde_json::to_value(&result).unwrap();
        let resp = Response { id, result: Some(result), error: None };
        connection.sender.send(Message::Response(resp))?;
        continue;
    }
    ...
};
```

Snippet 10. – Extrait de `goto_def.rs` modifié pour retourner un `Location` dans la réponse `GotoDefinitionResponse`

Cette communication permet de visualiser les échanges entre l'IDE et un serveur de langage. En pratique après avoir implémenté une logique de résolution des définitions un peu plus réaliste cette communication ne serait pas visible mais bénéficiera à l'intégration dans l'IDE. Si on l'intégrait dans VSCode, , la fonctionnalité du clic droit + Aller à la définition fonctionnerait.

Systèmes de surglignage de code

Les IDEs modernes supportent possèdent des systèmes de surglignage de code (syntax highlighting en anglais) permettant de rendre le code plus lisible en colorisant les mots, charactères ou groupe de symboles de même type (séparateur, opérateur, mot clé du langage, variable, fonction, constante, …). Ces systèmes se distinguent par leur possibilités d'intégration. Les thèmes intégrés aux IDE peuvent définir directement les couleurs pour chaque type de token. Pour un rendu web, une version HTML contenant des classes CSS spécifiques à chaque type de token peut être générée, permettant à des thèmes écrits en CSS de venir appliquer les couleurs. Les possibilités de génération pour le HTML pour le web implique parfois une génération dans le navigateur ou sur le serveur directement.

Un système de surlignage est très différent d'un parseur. Même s'il traite du même langage, dans un cas, on cherche juste à découper le code en tokens et y définir un type de token. Ce qui s'apparente seulement à la premier étape du lexer/tokenizer généralement rencontré dans les parseurs.

Textmate

Textmate est un IDE pour MacOS qui a inventé un système de grammaire Textmate. Elles permettent de décrire comment tokeniser le code basée sur des expressions régulières. Ces expressions régulières viennent de la librairie C Oniguruma (48). VSCode utilise ces grammaires Textmate (49). Intellij IDEA l'utilise également pour les langages non supportés par Intellij IDEA comme Swift, C++ et Perl (50).

Exemple de grammaire Textmate permettant de décrire un language nommé `untitled` avec 4 mots clés et des chaines de charactères entre guillemets, ceci matché avec des expressions régulières. Tiré de leur documentation (51).

```
{  scopeName = 'source.untitled';
   fileTypes = ( );
   foldingStartMarker = '\{\s*$';
   foldingStopMarker = '^\s*\}';
   patterns = (
      {  name = 'keyword.control.untitled';
         match = '\b(if|while|for|return)\b';
      },
      {  name = 'string.quoted.double.untitled';
         begin = '"';
         end = '"';
         patterns = (
            {  name = 'constant.character.escape.untitled';
               match = '\\.';
            }
         );
      },
   );
}
```

La documentation précise un choix important de conception: « A noter que ces regex sont matchées contre une seule ligne à la fois. Cela signifie qu'il n'est pas possible d'utiliser une pattern qui matche plusieurs lignes. La raison est technique: être capable de redémarrer le parseur à une ligne arbitraire et devoir reparser seulement un nombre minimal de lignes affectés par un changement. Dans la plupart des situations, il est possible d'utiliser le model `begin` / `end` pour dépasser cette limite. » (51) (Traduction personnelle, dernier paragraphe section 12.2).

Tree-Sitter

Tree-Sitter (52) se définit comme un « outil de génération de parser et une librairie de parsing incrémentale. Il peut construire un arbre de syntaxe concret (CST) pour depuis un fichier source et efficacement mettre à jour cet arbre quand le fichier source est modifié. » (52) (Traduction personnelle)

Rédiger une grammaire Tree-Sitter consiste en l'écriture d'une grammaire en Javascript dans un fichier `grammar.js`. Le cli `tree-sitter` va ensuite générer un parseur en C qui pourra être utilisé directement via le CLI `tree-sitter` durant le développement et être facilement embarquée comme librarie C sans dépendance dans n'importe quelle type d'application (52, 53).

Tree-Sitter est supporté dans Neovim (54), dans le nouvel éditeur Zed (55), ainsi que d'autres. Tree-Sitter a été inventé par l'équipe derrière Atom (56) et est même utilisé sur GitHub, notamment pour la navigation du code pour trouver les définitions et références et lister tous les symboles (fonctions, classes, structs, etc) (57).

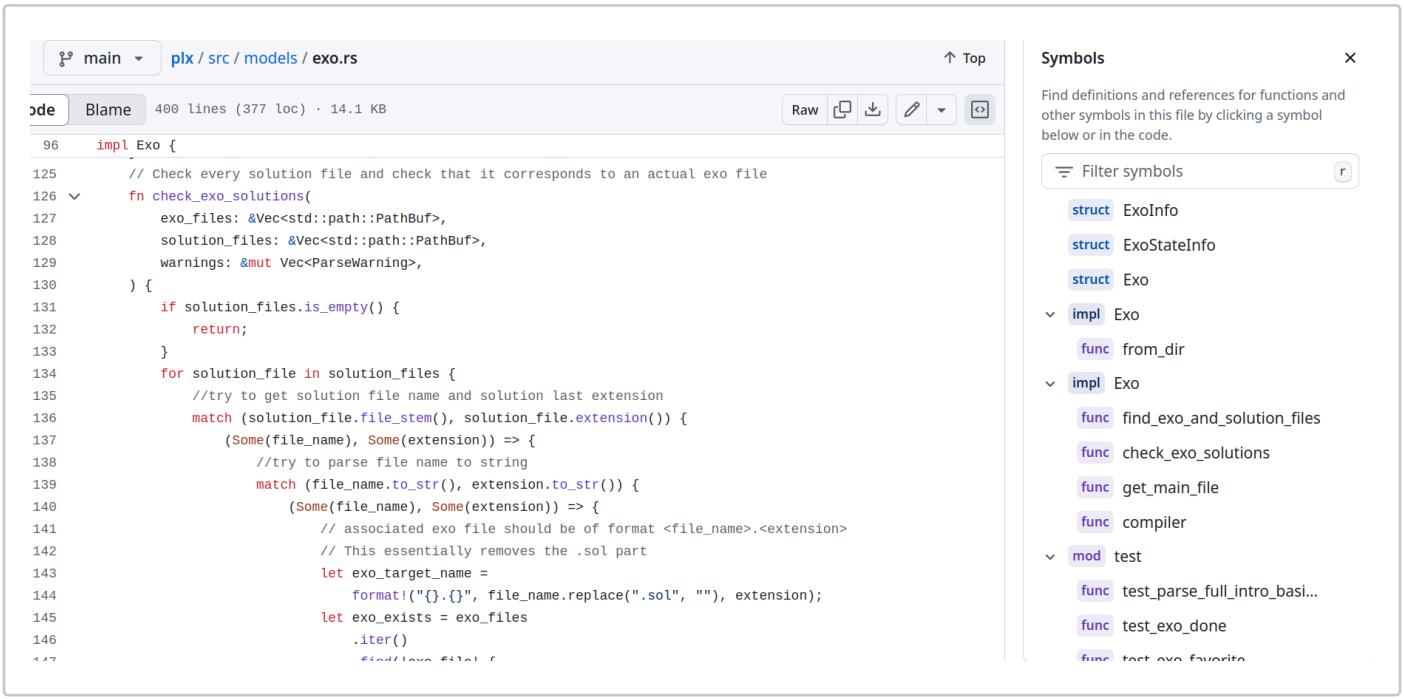
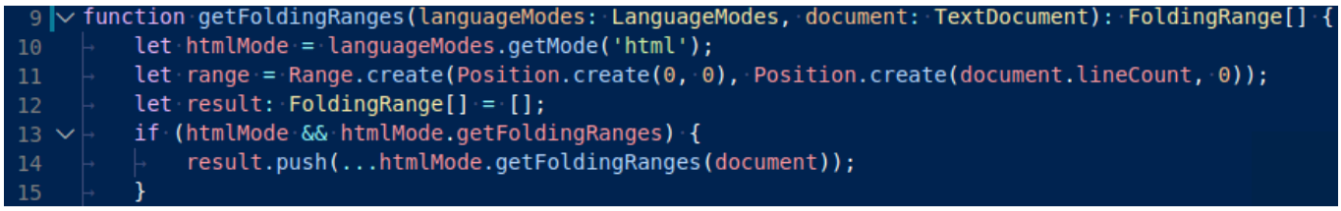


Fig. 3. – Liste de symboles générées par Tree-Sitter, affichés à droite du code sur GitHub pour un exemple de code Rust de PLX

Semantic highlighting

Le surlignage sémantique est une extension du surlignage syntaxique. Les serveurs de language peuvent ainsi fournir des tokens sémantiques qui apportent une classification plus fine du language, que les systèmes syntaxiques ne peuvent pas détecter. (58)

Without semantic highlighting:



With semantic highlighting:

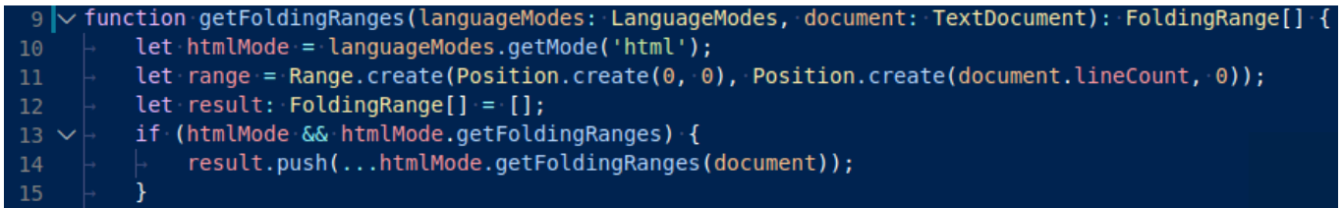


Fig. 4. – Exemple tiré de la documentation de VSCode, démontrant quelques améliorations dans le surglignage. Les paramètres `languageModes` et `document` sont colorisés différemment que les variables locales. `Range` et `Position` sont colorisées commes des classes. `getFoldingRanges` dans la condition est colorisée en tant que fonction ce qui la différencie des autres propriétés. (58)

En voyant la liste des tokens sémantiques possible dans la spécification LSP (59), on comprend mieux l'intérêt et les possibilités de surlignage avancé. Par exemple, on trouve des tokens `macro`, `regexp`, `typeParameter`, `interface`, `enum`, `enumMember`, qui seraient difficile de différencier durant la tokenisation mais qui peuvent être surligné différement pour mettre en avant leur différence sémantique.

Sur cette exemple de C, surgligné ici uniquement grâce à Tree-Sitter, sans surlignage sémantique, on voit que les appels de `HEY` et `hi` dans le `main` ont les mêmes couleurs.

```c
#include <stdio.h>

const char *HELLO = "Hey";
#define HEY(name) printf("%s %s\n", HELLO, name)
void hi(char *name) { printf("%s %s\n", HELLO, name); }

int main(int argc, char *argv[]) {
    hi("Samuel");
    HEY("Samuel");
    return 0;
}
```

Via la commande `:InspectTree` dans Neovim qui permet d'afficher l'arbre généré par Tree-Sitter, on voit que les 2 lignes `hi` et `HEY` sont catégorisés sans surprise comme des fonctions (noeuds `function`, `arguments`, ...).

```
(expression_statement ; [7, 4] - [7, 17]
  (call_expression ; [7, 4] - [7, 16]
    function: (identifier) ; [7, 4] - [7, 6]
    arguments: (argument_list ; [7, 6] - [7, 16]
      (string_literal ; [7, 7] - [7, 15]
        (string_content))))) ; [7, 8] - [7, 14]
(expression_statement ; [8, 4] - [8, 18]
  (call_expression ; [8, 4] - [8, 17]
    function: (identifier) ; [8, 4] - [8, 7]
    arguments: (argument_list ; [8, 7] - [8, 17]
      (string_literal ; [8, 8] - [8, 16]
        (string_content))))) ; [8, 9] - [8, 15]
```

Extrait de la commande `:Inspect` dans Neovim avec le curseur sur le `HEY`, qui nous montre que le serveur de langage (`clangd` ici), a réussi à préciser la notion de macro au-delà simple appel de fonction.

```
Semantic Tokens
  - @lsp.type.macro.c links to PreProc   priority: 125
  - @lsp.mod.globalScope.c links to @lsp   priority: 126
  - @lsp.typemod.macro.globalScope.c links to @lsp   priority: 127
```

Ainsi dans Neovim une fois `clangd` lancé, l'appel de `HEY` prend ainsi la même couleur que celle attribuée sur sa définition.

Choix final

L'auteur a ignoré l'option du système de SublimeText. pour la simple raison qu'il n'est supporté nativement que dans SublimeText, probablement parce que cet IDE est propriétaire (60). Ce système utilisent des fichiers `.sublime-syntax`, qui ressemble à TextMate (61) mais rédigé en YAML.

Si le temps le permet, une grammaire développée avec Tree-Sitter permettra de supporter du surglignage dans Neovim. Le choix de ne pas explorer plus les grammaires Textmate sur le long terme se justifie également par la roadmap de VSCode: entre mars et mai 2025 (62, 63), du travail d'investigation a été fait pour explorer les grammaires existantes et l'usage de surlignage de code (64). Des premiers efforts d'exploration avait d'ailleurs déjà eu lieu en septembre 2022 (65). L'usage du Semantic highlighting n'est pas au programme de ce travail mais pourra être explorer dans le futur si certains éléments sémantiques pourraient en bénéficier.

POC de Tree-Sitter

Ce POC vise à prouver que l'usage de Tree-Sitter fonctionne pour coloriser les préfixes et les flags de Snippet 11 pour ne pas avoir cette affichage noir sur blanc qui ne facilite pas la lecture.

```
// Basic MCQ exo
exo Introduction

opt .multiple
- C is an interpreted language
- .ok C is a compiled language
- C is mostly used for web applications
```

Snippet 11. – Un exemple de question choix multiple dans un fichier `mcq.dy`, décrite avec la syntaxe DY. Les préfixes sont `exo` (titre) et `opt` (options). Les flags sont `.ok` et `.multiple`.

Une fois la grammaire mise en place avec la commande `tree-sitter init`, il suffit de remplir le fichier `grammar.js`, avec une ensemble de régle construites via des fonctions fournies par Tree-Sitter et des expressions régulières.

```
module.exports = grammar({
  name: "dy",
  rules: {
    source_file: ($) => repeat($._line),
    _line: ($) =>
      seq( choice($.commented_line, $.prefixed_line, $.list_line, $.content_line), "\n"),
    prefixed_line: ($) =>
      seq($.prefix, optional(repeat($.property)), optional(seq(" ", $.content))),
    commented_line: (_) => token(seq(/\/\/ /, /.+/)),
    list_line: ($) => seq($.dash, repeat($.property), optional(" "),
optional($.content)),
    dash: (_) => token(prec(2, /- /)),
    prefix: (_) => token(prec(1, choice("exo", "opt"))),
    property: (_) => token(prec(3, seq(".", choice("multiple", "ok")))),
    content_line: ($) => $.content,
    content: (_) => token(prec(0, /.+/)),
  },
});
```

On observe dans cet exemple un fichier source, découpé en une répétition de ligne. Il y a 4 types de lignes qui sont chacunes décrites avec des plus petits morceaux. `seq` indique une liste de tokens qui viendront en séquence, `choice` permet de tester plusieurs options à la même position. On remarque également la liste des préfixes et flags insérés dans les tokens de `prefix` et `property`. La documentation **The Grammar DSL** de la documentation explique toutes les options possibles en détails (66).

Après avoir appelé `tree-sitter generate` pour générer le code du parser C et `tree-sitter build` pour le compiler, on peut demander au CLI de parser un fichier donné et afficher le CST. Dans cet arbre qui démarre avec son noeud racine `source_file`, on y voit les noeuds du même type que les règles définies précédemment, avec le texte extrait dans la plage de charactères associée au noeud. Par exemple, on voit que l'option `C is a compiled language` a bien été extraite à la ligne 5, entre le byte 6 et 30 (`5:6 - 5:30`) en tant que `content`. Elle suit un token de `property` avec notre flag `.ok` et le tiret de la règle `dash`.

```
dy> tree-sitter parse -c mcq.dy
0:0  - 7:0    source_file
0:0  - 0:16     commented_line `// Basic MCQ exo`
0:16 - 1:0      "\n"
1:0  - 1:16     prefixed_line
1:0  - 1:3        prefix `exo`
1:3  - 1:4        " "
1:4  - 1:16       content `Introduction`
1:16 - 3:0      "\n"
3:0  - 3:13     prefixed_line
3:0  - 3:3        prefix `opt`
3:3  - 3:13       property ` .multiple`
3:13 - 4:0      "\n"
4:0  - 4:30     list_line
4:0  - 4:2        dash `- `
4:2  - 4:30       content `C is an interpreted language`
4:30 - 5:0      "\n"
5:0  - 5:30     list_line
5:0  - 5:2        dash `- `
5:2  - 5:5        property `.ok`
5:5  - 5:6        " "
5:6  - 5:30       content `C is a compiled language`
5:30 - 6:0      "\n"
6:0  - 6:39     list_line
6:0  - 6:2        dash `- `
6:2  - 6:39       content `C is mostly used for web applications`
6:39 - 7:0      "\n"
```

Fig. 5. – Concrete Syntax Tree généré par la grammaire définie sur le fichier `mcq.dy`

La tokenisation fonctionne bien pour cette exemple, chaque élément est correctement découpé et catégorisé. Pour voir ce snippet en couleurs, il nous reste deux choses à définir. La première consiste en un fichier `queries/highlighting.scm` qui décrit des requêtes de surlignage sur l'arbre (highlights query) permettant de sélectionner des noeuds de l'arbre et leur attribuer un nom de surlignage (highlighting name). Ces noms ressembles à `@variable`, `@constant`, `@function`, `@keyword`, `@string` etc... et des versions plus spécifiques comme `@string.regexp`, `@string.special.path`. Ces noms sont ensuite utilisés par les thèmes pour appliquer un style.

```
> cat queries/highlights.scm
(prefix) @keyword
(commented_line) @comment
(content) @string
(property) @property
(dash) @operator
```

Le CLI supporte directement la configuration d'un thème via son fichier de configuration, on reprend simplement chaque nom de surlignage en lui donnant une couleur.

```
> cat ~/.config/tree-sitter/config.json
{
    "parser-directories": [ "/home/sam/code/tree-sitter-grammars" ],
    "theme": {
        "property": "#1bb588",
        "operator": "#20a8c3",
        "string": "#1f2328",
        "keyword": "#20a8c3",
        "comment": "#737a7e"
```

```
    }
}
```

```
// Basic MCQ exo
exo Introduction

opt .multiple
- C is an interpreted language
- .ok C is a compiled language
- C is mostly used for web applications
```

Fig. 6. – Screenshot du résultat de la commande `tree-sitter highlight mcq.dy` avec notre exercice surligné

L'auteur de ce travail s'est inspiré de l'article **How to write a tree-sitter grammar in an afternoon** (Ben Siraphob) (67) pour ce POC. Le résultat de ce POC est encourageant, même s'il faudra probablement plus que quelques heures pour gérer les détails, comprendre, tester et documenter l'intégration dans Neovim, cette partie nice to have a des chances de pouvoir être réalisée dans ce travail au vu du résultat atteint avec ce POC.

Protocoles de synchronisation et formats de sérialisation existants

Le serveur de gestion de sessions live a besoin d'un système de communication bidirectionnelle en temps réel, afin de transmettre le code et les résultats des étudiants. Ces messages seront transformées dans un format standard, facile à sérialiser et désérialiser en Rust. Cette section explore les formats textuels et binaires disponibles, ainsi que les protocoles de communication bi-directionnelle.

JSON

Contrairement à toutes les critiques relevées précédemment sur le JSON et d'autres formats, dans leur usage en tant que format source, JSON est une option solide pour la communication entre clients-serveurs. Le format JSON est très populaire pour les APIs REST, les fichiers de configuration, et d'autres usages.

En Rust, avec `serde_json`, il est plutôt facile de parser du JSON dans une `struct`. Exemple simplifié tiré de leur documentation (68). Une fois la macro `Deserialize` appliquée, on peut directement appeler `serde_json::from_str(json_data)`.

```rust
use serde::{Deserialize, Serialize};
use serde_json::Result;

#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: u8,
    phones: Vec<String>,
}
// ...
let data = r#" {
        "name": "John Doe",
        "age": 43,
        "phones": [ "+44 1234567", "+44 2345678" ]
    }"#;
let p: Person = serde_json::from_str(data).unwrap();
println!("Please call {} at the number {}", p.name, p.phones[0]);
```

Autre exemple pour montrer qu'il est facile de générer un objet JSON de structure quelconque. Egalement tiré de leur documentation (69).

```rust
use serde_json::json;

fn main() {
    // The type of `john` is `serde_json::Value`
    let john = json!({
        "name": "John Doe",
        "age": 43,
        "phones": [ "+44 1234567", "+44 2345678" ]
    });
    println!("first phone number: {}", john["phones"][0]);
    println!("{}", john.to_string());
}
```

Protocol Buffers - ProtoBuf

Parmi les formats binaires, on trouve ProtoBuf, un format développé par Google pour sérialiser des données structurées, de manière compacte, rapide et simple. L'idée est de définir un schéma dans

un style non spécifique à un langage de programmation, puis de génération automatiquement du code pour interagir avec ces structures depuis du C++, Java, Go, Ruby, C# et d'autres. (70)

Un simple exemple de description d'une personne en ProtoBuf tiré de leur site web (70).

```
edition = "2023";

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

Et son usage en Java avec les classes autogénérées à la compilation, exemple tiré de leur site web (70).

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

Le langage Rust n'est pas officiellement supporté mais un projet du nom de PROST! existe (71) et permet de générer du code Rust depuis des fichiers Protobuf.

MessagePack

Le slogan de MessagePack, format binaire de sérialisation: « C'est comme JSON, mais rapide et léger » (Traduction personnelle). Une implémentation en Rust du nom de RPM existe (72).

Websocket

Le protocole Websocket, définie dans la RFC 6455 (73), permet une communication bi-directionnelle entre un client et un serveur. A la place de l'approche de requête-réponses du HTTP, le protocole Websocket définit une manière de garder une connexion TCP ouverte et un moyen d'envoyer des messages dans les 2 sens. On évite ainsi d'ouvrir plusieurs connexions HTTP, une nouvelle à chaque fois qu'un événement se produit ou que le client veut vérifier si le serveur n'a pas d'événements à transmettre. La technologie a été pensée pour être utilisée par des applications dans les navigateurs, mais fonctionne également en dehors (73).

La section **1.5 Design Philosophy** explique que le protocole est conçu pour un « minimal framing » (encadrement minimal autour des données envoyées), juste assez pour permettre de découper le flux TCP en frame (en message d'une durée définie) et de distinguer le texte des données binaires. Le texte doit être encodé en UTF-8. (74)

La section **1.3. Opening Handshake**, nous explique que pour permettre une compatibilité avec les serveurs HTTP et intermédiaires sur le réseau, l'opening handshake (l'initialisation du socket une fois connecté) est compatible avec le format des entêtes HTTP. Cela permet d'utiliser un serveur websocket sur le même port qu'un serveur web, ou d'héberger plusieurs serveurs websocket sur différents routes par exemple `/chat` et `/news`. (75)

Dans l'écosystème Rust, il existe plusieurs crate qui implémente le protocole, parfois côté client, côté serveur ou les deux. Il existe plusieurs approches sync et async, nous nous concentrons ici sur

une approche sync avec gestion des threads natifs manuelle pour simplifier l'implémentation et les recherches.

La crate `tungstenite` propose une abstraction du protocole qui permet de facilement interagir avec des `Message`, leur écriture `send()` et leur lecture `read()` de façon très simple (76). Elle passe la Autobahn Test Suite (suite de tests de plus de 500 cas) (77).

```rust
use std::net::TcpListener;
use std::thread::spawn;
use tungstenite::accept;

/// A WebSocket echo server
fn main () {
    let server = TcpListener::bind("127.0.0.1:9001").unwrap();
    for stream in server.incoming() {
        spawn (move || {
            let mut websocket = accept(stream.unwrap()).unwrap();
            loop {
                let msg = websocket.read().unwrap();

                // We do not want to send back ping/pong messages.
                if msg.is_binary() || msg.is_text() {
                    websocket.send(msg).unwrap();
                }
            }
        });
    }
}
```

Snippet 12. – Exemple de serveur echo en WebSocket avec la crate `tungstenite`. Tiré de leur README (76)

Une version async pour le runtime tokio existe également, elle s'appelle `tokio-tungstenite`, si le besoin de passer à un modèle async avec Tokio se fait sentir, nous devrions pouvoir y migrer (78).

Il existe une crate `websocket` avec une approche sync et async, qui est dépréciee et dont le README (79) conseille l'usage de `tungstenite` ou `tokio-tungstenite` à la place (79).

A noter qu'il existe d'autres crates tel que `fastwebsockets` (80) à disposition, qui ont l'air de permettre de travailler a un plus bas niveau. Pour faciliter l'implémentation nous les ignorons pour ce travail.

gRPC

gRPC est un protocole basé sur ProtoBuf, inventé par Google. Il se veut être un système de Remote Procedure Call (RPC - un système d'appel de fonctions à distance), universelle et performant qui supporte le streaming bi-directionnelle sur HTTP2. La possibilité de travailler avec plusieurs langages reposent sur la génération automatique de code pour les clients et serveurs permettant de gérer la sérialisation en Protobuf et gérant le transport.

En plus des définitions des messages en Protobuf déjà présentée, il est possible de définir des services, avec des méthodes avec un type de message et un type de réponse.

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Snippet 13. – Exemple de fichier .proto définissant 2 messages et un service permettant d'envoyer un nom et de recevoir des salutations en retour. Tiré de leur documentation d'introduction (81)

Comme Protobuf, Rust n'est pas supporté officiellement mais une implémentation du nom de Tonic existe (82), elle utilise Prost! mentionnée précédemment pour l'intégration de Protobuf.

Un article de 2019, intituté **The state of gRPC in the browser** (83) montre que l'utilisation de gRPC dans les navigateurs web est encore malheureusement mal supportée. En résumé, « il est actuellement impossible d'implémenter la spécification HTTP/2 gRPC dans le navigateur, comme il n'y a simplement pas d'API de navigateur avec un contrôle assez fin sur les requêtes. » (Traduction personnelle). La solution a été trouvée à ce problème est le projet gRPC-Web qui fournit un proxy entre le navigateur et le serveur gRPC, faisant les conversions nécessaires entres gRPC-Web et gRPC.

Il reste malheureusement plusieurs limites: le streaming bi-directionnelle n'est pas possible, le client peut faire des appels unaires (pour un seul message) et peut écouter une server-side streams (flux de messages venant du server). L'autre limite est le nombre maximum de connexions en streaming simultanées dans un navigateur sur HTTP/1.1 fixées à 6 (84), ce qui demande de restructurer ses services gRPC pour ne pas avoir plus de 6 connexions en server-side streaming à la fois.

tarpc

tarpc également développé sur l'organisation GitHub de Google sans être un produit officiel, se définit comme « un framework RPC pour Rust, avec un focus sur la facilité d'utilisation. Définir un service peut être fait avec juste quelques lignes de code et le code boilerplate du serveur est géré pour vous. » (Traduction personnelle) (85)

tarpc est différent de gRPC et Cap'n Proto « en définissant le schéma directement dans le code, plutôt que dans un langage séparé comme Protobuf. Ce qui signifie qu'il n'y a pas de processus de compilation séparée et pas de changement de contexte entre différent langages. » (Traduction personnelle) (85)

Choix final

Par soucis de facilité de debug, d'implémentation et d'intégration, l'auteur a choisi de rester sur un format textuel et d'implémenter la sérialisation en JSON via la crate mentionnée précédemment `serde_json`. L'expérience existante des websockets de l'auteur, sa possibilité de choisir le format de données, et son solide support dans les navigateurs (au cas où PLX avait une version web un jour), font que ce travail utilisera la combinaisons Websockets + JSON.

gRPC aurait pu aussi être une option comme PLX est en dehors du navigateur, il ne serait pas touché par les limites exprimées. Cependant, cela renderait plus difficile un support d'une version web de PLX si le projet en avait besoin dans le futur.

Quand l'usage de PLX dépassera une dizaines/centaines d'étudiants connectés en même moment et que la latence sera trop forte ou que les coûts d'infrastructures deviendront un soucis, les formats binaires plus légers seront une option à creuser. Au vu des nombreux choix, mesurer la taille des messages, la latence de transport et le temps de sérialisation sera important pour faire un choix. D'autres projets pourraient également être considéré comme Cap'n Proto (86) qui se veut plus rapide que Protobuf, ou encore Apache Thrift (87). Ces dernières options n'ont pas été explorée dans cet état de l'art principalement parce qu'elles proposent un format binaire.

POC de synchronisation de messages JSON via websockets avec tungstenite

Pour vérifier la faisabilité technique d'envoyer des messages en temps réel en Rust via websockets, un petit POC a été développé dans le dossiers `pocs/websockets-json`. Le code et les résultats des checks doivent être transmis des étudiants depuis le client PLX des étudiants vers ce lui de l'enseignant, en passant par le serveur de session live.

De part sa nature interactive, il n'est pas évident de retranscrire ce qui s'y passe quand on lance le POC dans 3 shell côte à côte, le mieux serait d'aller compiler et lancer à la main. Nous documentons ici un aperçu du résultat.

Ce petit programme en Rust prend en argument son rôle (`server`, `teacher` ou `student`), tout le code est ainsi dans un seul fichier `main.rs` et un seul binaire.

Ce programme a la structure suivante, le dossier `fake-exo` contient l'exercice à implémenter.

```
.
├── Cargo.lock
├── Cargo.toml
├── fake-exo
│   ├── Cargo.lock
│   ├── Cargo.toml
│   ├── compare_output.txt
│   └── src
└── src
    └── main.rs
```

Snippet 14. – Structure de fichiers du POC.

```rust
// Just print "Hello <name> !" where <name> comes from argument 1
fn main() {
    println!("Hello, world!");
}
```

Snippet 15. – Code Rust de départ de l'exercice fictif à compléter par l'étudiant

Le mini protocole définit pour permettre cette synchronisation est découpé en 2 étapes.

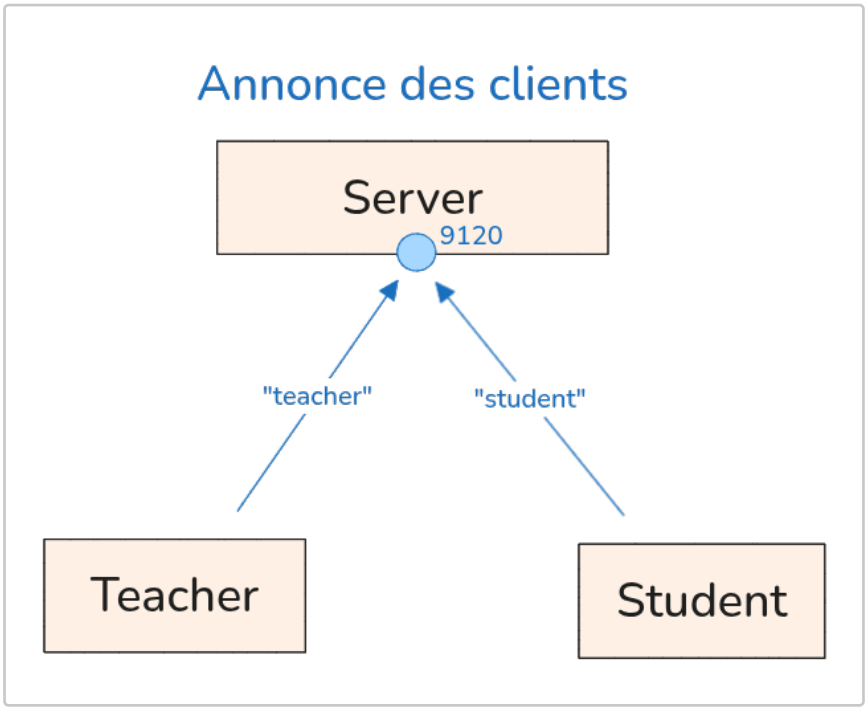


Fig. 7. – La première partie consiste en une mise en place par la connexion et l'annonce des clients de leur rôle, en se connectant puis en envoyant leur rôle en string.

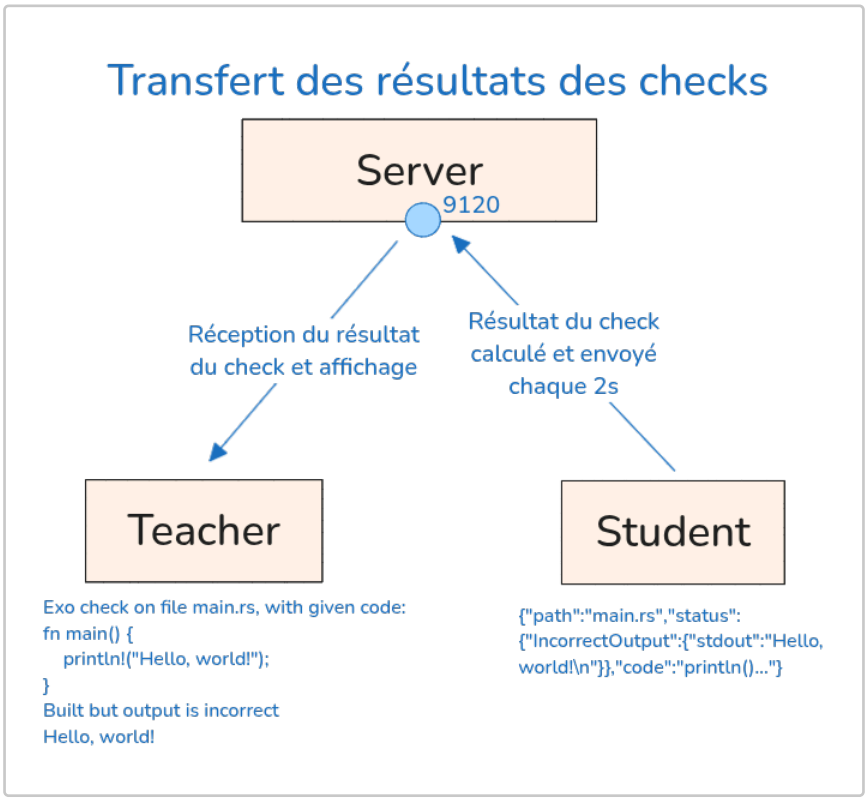


Fig. 8. – La deuxième partie consiste en l'envoi régulier du client du résultat du check vers le serveur, qui ne fait que de transmettre au socket associé au `teacher`.

Dans un premier shell (S1), nous lançons en premier lieu le serveur:

```
websockets-json> cargo run -q server
Starting server process ...
Server started on 127.0.0.1:9120
```

Snippet 16. – Lancement du serveur et attente de connexions sur le port 9120.

Dans un deuxième shell (S2), on lance le `teacher`:

```
websockets-json> cargo run -q teacher
Starting teacher process...
Sending whoami message
Waiting on student's check results
```

Snippet 17. – Lancement du `teacher`, connexion au serveur et envoi d'un premier message littéral `teacher` pour annoncer son rôle

Dans S1, on voit que le serveur a bien reçu la connexion et a détecté le rôle de `teacher`.

```
...
Teacher connected, saved associated socket.
```

Snippet 18. – `teacher` est bien connecté au serveur

Dans S3, on lance finalement le rôle de l'étudiant:

```
websockets-json> cargo run -q student
Starting student process...
Sending whoami message
Starting to send check's result every 2000 ms
Sending another check result
{"path":"fake-exo/src/main.rs","status":{"IncorrectOutput":{"stdout":"Hello, world!
\n"}},"code":"// Just print \"Hello <name> !\" where <name> comes from argument 1\nfn
main() {\n    println!(\"Hello, world!\");\n}\n"}
```

On y voit le message envoyé contient le résultat du check après compilation et exécution.


```
{
  "path": "fake-exo/src/main.rs",
  "status": {
    "IncorrectOutput": {
      "stdout": "Hello, world!\n"
    }
  },
  "code": "// Just print \"Hello <name> !\" where <name> comes from argument 1\nfn main()
{\n    println!(\"Hello, world!\");\n}\n"
}
```


Snippet 19. – Le message envoyé avec un chemin de fichier, le code et le statut. Le statut est une enum définie à « output incorrect », puisque l'exercice n'est pas encore implémenté.

Le serveur sur le S1, on ne voit que le `Forwarded one message to teacher`. Sur le S2, on voit immédiatement ceci:

```
Exo check on file fake-exo/src/main.rs, with given code:
// Just print "Hello <name> !" where <name> comes from argument 1
fn main() {
    println!("Hello, world!");
}
Built but output is incorrect
Hello, world!
```

Snippet 20. – Le `teacher` a bien reçu le message et peut l'afficher, la synchronisation temps réel a fonctionné.

Si l'étudiant introduit une erreur de compilation, un message avec un statut différent est envoyé, voici ce que reçoit le `teacher` :

```
Exo check on file fake-exo/src/main.rs, with given code:
// Just print "Hello <name> !" where <name> comes from argument 1
fn main() {
    println!("Hello, world!", args[3]);
}
failed build with error
  Compiling fake-exo v0.1.0
error: argument never used
--> src/main.rs:3:31
  |
3 |     println!("Hello, world!", args[3]);
  |              ----------------  ^^^^^^^ argument never used
  |              |
  |              formatting specifier missing
```

Snippet 21. – Le `teacher` a bien reçu le code actuel avec l'erreur et l'output de compilation de Cargo

Le système de synchronisation en temps réel permet ainsi d'envoyer différents messages au serveur qui le retransmet directement au `teacher`. Même si cet exemple est minimale puisqu'il ne vérifie pas la source des messages, et qu'il n'y a qu'un seul étudiant et enseignant impliqué, nous avons démontré que la crate `tungstenite` fonctionne.

TODO: cette bibliographie ne respecte pas encore tous les standards de la HEIG-VD, encore en rôdage avec Typst et le guide de la bibliothèque sur la norme ISO-690…

Bibliographie

1. DUBOVSKOY, Alexey. Cooklang – Recipe Markup Language. Online. 2025. Available from: https://cooklang.org/
2. DUBOVSKOY, Alexey. Canonical Cooklang parser in Rust. Online. 2025. Available from: https://github.com/cooklang/cooklang-rs
3. RON, Contributeurs de. Rusty Object Notation. Online. 2025. Available from: https://github.com/ron-rs/ron
4. TORM. udl v0.3.1 - Parser for UDL (Universal Data Language). Online. 2023. Available from: https://crates.io/crates/udl
5. TORM. The Khi data language. Online. 2024. Available from: https://github.com/khilang/khi
6. TORM. Rust Khi parser & library. Online. 2024. Available from: https://github.com/khilang/khi.rs
7. ASSOCIATION. bitmark Association website. Online. 2025. Available from: https://www.bitmark-association.org/
8. ASSOCIATION. bitmark Hackathon. Online. 2025. Available from: https://www.bitmark-association.org/bitmarkhackathon
9. ASSOCIATION. bitmark Documentation. Online. 2025. Available from: https://docs.bitmark.cloud/
10. ASSOCIATION. Quizzes - .multiple-choice, .multiple-choice-1. Online. 2025. Available from: https://docs.bitmark.cloud/quizzes/#multiple-choice-multiple-choice-1
11. TASKBASE. open-taskpool - 12,000 UK 🇺🇦 → DE 🇩🇪 & DE 🇩🇪 → EN🇬🇧 learning tasks ready for you to use. Online. 2025. Available from: https://github.com/taskbase/open-taskpool
12. ASSOCIATION. Quizzes - .cloze (gap text). Online. 2025. Available from: https://docs.bitmark.cloud/quizzes/#cloze-gap-text
13. CLASSTIME. Créer la première question / le premier jeu de questions. Online. 2024. Available from: https://help.classtime.com/fr/comment-commencer-a-utiliser-classtime/creer-la-premiere-question-le-premier-jeu-de-questions
14. KUNDERT, Ken. NestedText — A Human Friendly Data Format. Online. 2025. Available from: https://github.com/KenKundert/nestedtext
15. KEN et KUNDERT, Kale. NestedText documentation - Schemas. Online. 2025. Available from: https://nestedtext.org/en/latest/schemas.html
16. BOB22Z. docs.rs - Crate nestedtext. Online. 2025. Available from: https://nestedtext/latest/nestedtext/
17. LUDWIG, Sönke. SDLang, Simple Declarative Language. Online. 2025. Available from: https://sdlang.org/
18. KAT MARCHÁN (ZKAT), et contributeurs. KDL, a cudlly document language. Online. 2025. Available from: https://kdl.dev/
19. All Crates for keyword 'parser'. Online. 2025. Available from: https://crates.io/keywords/parser
20. (EPAGE), Ed Page. winnow v0.7.8 A byte-oriented, zero-copy, parser combinators library. Online. 2025. Available from: https://crates.io/crates/winnow
21. Dependencies of kdl crate. Online. 2025. Available from: https://crates.io/crates/kdl/6.3.4/dependencies
22. (GEAL), Geoffroy Couprie. nom v8.0.0 A byte-oriented, zero-copy, parser combinators library. Online. 2025. Available from: https://crates.io/crates/nom
23. Reverse dependencies of nom crate. Online. 2025. Available from: https://crates.io/crates/nom/reverse_dependencies
24. pest v2.8.0 The Elegant Parser. Online. 2025. Available from: https://crates.io/crates/pest

25. (MARWES), Markus Westerlind. combine v4.6.7 Fast parser combinators on arbitrary streams with zero-copy support. Online. 2024. Available from: https://crates.io/crates/combine

26. JOSHUA BARRETTO (ZESTERER), et contributeurs, Rune Tynan (CraftSpider). chumsky v0.10.1 A parser library for humans with powerful error recovery. Online. 2025. Available from: https://crates.io/crates/chumsky

27. Most popular Rust libraries. Online. 2025. Available from: https://lib.rs/std

28. Serde data model. Online. 2025. Available from: https://serde.rs/data-model.html

29. MICROSOFT, et contributeurs. Language Server Protocol. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/

30. GROUP, JSON-RPC Working. JSON-RPC 2.0 Specification. Online. 2013. Available from: https://www.jsonrpc.org/specification

31. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Capabilities. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities

32. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Content part. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#contentPart

33. BERGERCOOKIE, et contributeurs. asm-lsp v0.10.0 Language Server for x86/x86_64, ARM, RISCV, and z80 Assembly Code. Online. 2025. Available from: https://crates.io/crates/asm-lsp

34. ORGANISATION, et contributeurs eclipse-jdtls. GitHub - eclipse-jdtls/eclipse.jdt.ls: Java language server. Online. 2025. Available from: https://github.com/eclipse-jdtls/eclipse.jdt.ls

35. TAILWINDLABS, et contributeurs. GitHub - tailwindlabs/tailwindcss-intellisense: Intelligent Tailwind CSS tooling for Visual Studio Code. Online. 2025. Available from: https://github.com/tailwindlabs/tailwindcss-intellisense

36. ORGANISATION, et contributeurs typescript-language-server. GitHub - typescript-language-server/typescript-language-server: TypeScript & JavaScript Language Server. Online. 2025. Available from: https://github.com/typescript-language-server/typescript-language-server

37. LSP-TYPES, Contributeurs de. lsp-types v0.97.0 Types for interaction with a language server, using VSCode's Language Server Protocol. Online. 2024. Available from: https://crates.io/crates/lsp-types

38. , et contributeurs Organisation gluon-lang. Reverse dependencies of lsp-types crate. Online. 2024. Available from: https://crates.io/crates/lsp-types/reverse_dependencies

39. MICROSOFT, et contributeurs. Implementations - Tools supporting the LSP. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/implementors/tools/

40. MICROSOFT, et contributeurs. Implementations - Language Servers. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/implementors/servers/

41. OXALICA, et contributeurs. async-lsp v0.2.2 Asynchronous Language Server Protocol (LSP) framework based on tower. Online. 2025. Available from: https://crates.io/crates/async-lsp

42. OXALICA, et contributeurs. nil/crates/nil/Cargo.toml - NIx Language server, an incremental analysis assistant for writing in Nix. Online. 2025. Available from: https://github.com/oxalica/nil/blob/577d160da311cc7f5042038456a0713e9863d09e/crates/nil/Cargo.toml#L11

43. MYRIAD-DREAMIN, et contributeurs. sync-ls - Synchronized language service inspired by async-lsp, primarily for tinymist. Online. 2025. Available from: https://crates.io/crates/sync-ls

44. ORGANISATION, et contributeurs tower-lsp-community. tower-lsp-server v0.21.1 Language Server Protocol implementation based on Tower. Online. 2025. Available from: https://crates.io/crates/tower-lsp-server

45. ORGANISATION, et contributeurs rust-lang. Reverse dependencies of lsp-server crate. Online. 2024. Available from: https://crates.io/crates/lsp-server/reverse_dependencies

46. EYAL KALDERON, et contributeurs. Reverse dependencies of tower-lsp crate. Online. 2023. Available from: https://crates.io/crates/tower-lsp/reverse_dependencies

47. ORGANISATION, et contributeurs rust-lang. rust-analyzer/lib/lsp-server/examples/goto_def.rs at master · rust-lang/rust-analyzer · GitHub. Online. 2025. Available from: https://github.com/rust-lang/rust-analyzer/blob/master/lib/lsp-server/examples/goto_def.rs

48. Online. 2025. Available from: https://macromates.com/manual/en/regular_expressions

49. Online. 2025. Available from: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide

50. Online. 2025. Available from: https://www.jetbrains.com/help/idea/textmate.html

51. LTD, MacroMates. Language Grammars — TextMate 1.x Manual - Example Grammar. Online. Available from: https://macromates.com/manual/en/language_grammars#example_grammar

52. TREE-SITTER, Contributeurs de. Introduction - Tree-sitter. Online. 2025. Available from: https://tree-sitter.github.io/tree-sitter/

53. TREE-SITTER, Contributeurs de. Creating Parsers - Getting Started - Tree-sitter. Online. 2025. Available from: https://tree-sitter.github.io/tree-sitter/creating-parsers/1-getting-started.html

54. Neovim Documentation - Treesitter. Online. 2025. Available from: https://neovim.io/doc/user/treesitter.html

55. Language Extensions - Grammar. Online. 2025. Available from: https://zed.dev/docs/extensions/languages?#grammar

56. Creating a Grammar. Online. 2025. Available from: https://flight-manual.atom-editor.cc/hacking-atom/sections/creating-a-grammar/

57. GITHUB, et contributeurs. Navigating code on GitHub. Online. 2025. Available from: https://docs.github.com/en/repositories/working-with-files/using-files/navigating-code-on-github

58. MICROSOFT, et contributeurs. Semantic Highlight Guide | Visual Studio Code Extension API. Online. 2025. Available from: https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide

59. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Semantic Tokens. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_semanticTokens

60. SUBLIMEHQ. SublimeHQ - End User License Agreement. Online. 2025. Available from: https://www.sublimehq.com/eula

61. SUBLIMEHQ. Syntax Definitions. Online. 2025. Available from: https://www.sublimetext.com/docs/syntax.html

62. MICROSOFT, et contributeurs. Iteration Plan for March 2025 · Issue #243015 · microsoft/vscode · GitHub. Online. 2025. Available from: https://github.com/microsoft/vscode/issues/243015

63. MICROSOFT, et contributeurs. Iteration Plan for May 2025 · Issue #248627 · microsoft/vscode · GitHub. Online. 2025. Available from: https://github.com/microsoft/vscode/issues/248627

64. MICROSOFT, et contributeurs. Explore using tree sitter for syntax highlighting · Issue #210475 · microsoft/vscode · GitHub. Online. 2025. Available from: https://github.com/microsoft/vscode/issues/210475

65. MICROSOFT, et contributeurs. [Exploration] Tree-sitter tokenization exploration (Fixes #161256) by aidaymar · Pull Request #161479 · microsoft/vscode · GitHub. Online. 2022. Available from: https://github.com/microsoft/vscode/pull/161479

66. TREE-SITTER, Contributeurs de. The Grammar DSL - Tree-sitter. Online. 2025. Available from: https://tree-sitter.github.io/tree-sitter/creating-parsers/2-the-grammar-dsl.html

67. SIRAPHOB, Ben. How to write a tree-sitter grammar in an afternoon | siraben's musings. Online. 2022. Available from: https://siraben.dev/2022/03/01/tree-sitter.html

68. serde_json - Parsing JSON as strongly typed data structures. Online. 2025. Available from: https://docs.rs/serde_json/latest/serde_json/index.html#parsing-json-as-strongly-typed-data-structures

69. serde_json - Constructing JSON values. Online. 2025. Available from: https://docs.rs/serde_json/latest/serde_json/index.html#constructing-json-values

70. GOOGLE, et contributeurs. Protocol Buffers Documentation. Online. 2025. Available from: https://protobuf.dev/

71. CONTRIBUTEURS. GitHub - tokio-rs/prost: PROST! a Protocol Buffers implementation for the Rust Language. Online. 2025. Available from: https://github.com/tokio-rs/prost

72. rmp - The Rust MessagePack Library. Online. 2025. Available from: https://docs.rs/rmp/latest/rmp/

73. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol. Online. 2025. Available from: https://www.rfc-editor.org/rfc/rfc6455

74. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol - 1.5. Design Philosophy. Online. 2025. Available from: https://www.rfc-editor.org/rfc/rfc6455#section-1.5

75. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol - 1.3. Opening Handshake. Online. 2025. Available from: https://www.rfc-editor.org/rfc/rfc6455#section-1.3

76. CONTRIBUTEURS, Snapview GmbH et. Lightweight stream-based WebSocket implementation. Online. 2025. Available from: https://crates.io/crates/tungstenite

77. CONTRIBUTEURS. GitHub - crossbario/autobahn-testsuite: Autobahn WebSocket protocol testsuite. Online. 2025. Available from: https://github.com/crossbario/autobahn-testsuite

78. CONTRIBUTEURS, Snapview GmbH et. tokio-tungstenite. Online. 2025. Available from: https://crates.io/crates/tokio-tungstenite

79. CONTRIBUTEURS. websocket. Online. 2025. Available from: https://crates.io/crates/websocket

80. CONTRIBUTEURS. fastwebsockets. Online. 2025. Available from: https://crates.io/crates/fastwebsockets

81. AUTHORS. Introduction to gRPC | gRPC. Online. 2025. Available from: https://grpc.io/docs/what-is-grpc/introduction/

82. CONTRIBUTEURS. GitHub - hyperium/tonic: A native gRPC client & server implementation with async/await support. Online. 2025. Available from: https://github.com/hyperium/tonic

83. BRANDHORST, Johan. The state of gRPC in the browser | gRPC. Online. 2019. Available from: https://grpc.io/blog/state-of-grpc-web

84. CONTRIBUTORS, MDN. EventSource - Web APIs | MDN. Online. 2025. Available from: https://developer.mozilla.org/en-US/docs/Web/API/EventSource

85. CONTRIBUTEURS. GitHub - google/tarpc: An RPC framework for Rust with a focus on ease of use. Online. 2025. Available from: https://github.com/google/tarpc

86. Cap'n Proto: Introduction. Online. 2025. Available from: https://capnproto.org/

87. Apache Thrift - Home. Online. 2025. Available from: https://thrift.apache.org/