

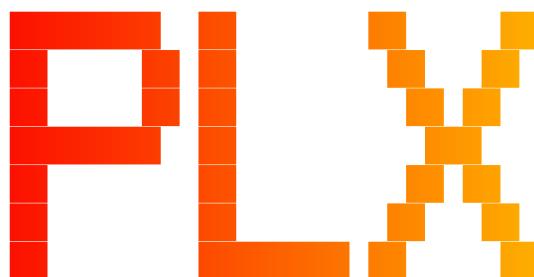


Département des Technologie de
l'information et de la communication (TIC)
Informatique et systèmes de communication
Informatique logicielle

Travail de Bachelor

Concevoir une expérience d'apprentissage interactive à la programmation avec PLX

Ou comment permettre aux enseignants de programmation
de concevoir des cours orientés sur la pratique et le feedback.



Étudiant

Samuel Roland

Enseignant responsable

Bertil Chapuis

Année académique

2024-25

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Vincent Peiris
Chef de département TIC

Authentification

Le soussigné, Samuel Roland, atteste par la présente avoir réalisé ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées

Yverdon-les-Bains, le 23.07.2025

Samuel Roland

Table des matières

Préambule	2
Authentification	3
Introduction	7
Contexte	7
Problèmes de l'expérience actuelle	8
L'approche de PLX	10
Nouveaux défis	13
Défi 1: Comment les enseignant·es peuvent voir le code et les résultats en temps réel ? ..	13
Défi 2: Comment faciliter la rédaction et la maintenance des exercices ?	14
Solutions existantes	20
Glossaire	23
Planification	24
Déroulement	24
Planification initiale	24
Planification finale	25
État de l'art	26
Format de données humainement éditables existants	26
KHI - Le langage de données universel	27
Bitmark - le standard des contenus éducatifs digitaux	27
NestedText – Un meilleur JSON	29
SDLang - Simple Declarative Language	30
KDL - Cuddly Data language	31
Conclusion	31
Librairies de parsing en Rust	33
Les serveurs de langage	34
Adoption	35
Librairies disponibles	35
Choix final	36
POC de serveur de langage avec <code>lsp-server</code>	36
Surlignage du code	37
Textmate - surlignage syntaxique	38
Tree-Sitter - surlignage syntaxique	38
Surlignage sémantique	39
Choix final	40
POC de surlignage de notre syntaxe avec Tree-Sitter	41
Protocoles de communication bidirectionnels et formats de sérialisation	45
JSON	45

WebSocket	45
Formats binaires	46
Choix final	47
POC de synchronisation de messages JSON via WebSocket avec tungstenite	47
Développement du serveur de session live	50
Définition du protocole	51
Vue d'ensemble du protocole	51
Définition des sessions live	51
Définition, identifiants et configuration du client	52
Transport, sérialisation et gestion de la connexion	53
Messages	54
Diagrammes de séquence	58
Vue d'ensemble de l'implémentation	62
La librairie <code>plx-core</code> et son module <code>live</code>	62
Les processus en jeu	63
Typage des commandes Tauri	63
Partage des types	64
Implémentation du client	65
Implémentation du tableau de bord	65
Implémentation du serveur	66
Lancement	66
Gestion de la concurrence	66
Tâches tokio	68
Tests de bouts en bouts	69
Développement de la syntaxe DY	70
Vue d'ensemble	70
Librairies implémentées	71
Lignes directrices de conception	72
Commentaires	73
Support du Markdown	73
Définition et contraintes des clés	73
Implémentation de la librairie <code>dy</code>	74
Catégorisation des lignes	75
Construction d'un arbre de blocs	76
Conversion vers la struct T	77
Implémentation de <code>plx-dy</code>	77
Modèle de données de PLX et choix des clés	77
Définition d'une hiérarchie de clés en Rust	78
Intégration de <code>plx-dy</code>	80
Structures de fichiers DY	80
Intégration au CLI	81
Détection d'erreurs	84
Tests unitaires	87
Conclusion	89
Bibliographie	90
Annexes	98

Outils utilisés	98
Usage de l'intelligence artificielle	98
Outils techniques	98
Logo	98
Cahier des charges original	99
Concevoir une expérience d'apprentissage interactive à la programmation avec PLX	99

Introduction

Contexte

L'informatique et particulièrement la programmation, sont des domaines **abstraits et complexes** à apprendre. Dans la majorité des universités, l'informatique est enseignée sur des cours composés d'une partie théorique, dispensée par un·e professeur·e, et d'une partie pratique, sous forme de laboratoires, encadrée par des assistant·es. Les sessions théoriques sont souvent données sous forme **magistrale**: une présentation durant 2 périodes pour présenter différents concepts, morceaux de code et études de cas. Les étudiant·es ont **rarement la possibilité d'être actif·ves**, ce qui limite fortement la concentration et la rétention de l'information. Une grande partie de l'auditoire décroche et préfère travailler sur des laboratoires ou réviser d'autres cours.

Lors des rares sessions d'exercice en classe et durant la révision en dehors des cours, un temps important est perdu à mettre en place les exercices et les vérifications manuelles. Ce **processus fastidieux** se fait au détriment de la pratique délibérée, concept popularisé par le psychologue Anders Ericsson (1) dans ses recherches en expertise (2). Il a étudié les points communs de l'entraînement des champion·nes et expert·es dans de nombreux domaines: sport, médecine, psychologie, échecs, armée, musique, ... En bref, leurs entraînements consistent à travailler de manière concentrée sur des sous-compétences spécifiques. Cette méthode demande de recevoir un feedback rapide et régulier, afin de corriger et affiner constamment son modèle mental. La solidité du modèle mental construit par l'expérience, permet d'atteindre un haut niveau d'expertise.

Ce travail de Bachelor s'inscrit dans ce contexte en poursuivant le projet PLX, application desktop qui accompagne les étudiant·es dans leur apprentissage du code. L'acronyme PLX signifie **Practice programming exercises in a deliberate Learning eXperience**. Le projet vise à **redéfinir l'expérience d'apprentissage et d'enseignement** de la programmation, en s'inspirant de la pratique délibérée.

Problèmes de l'expérience actuelle

Pour mieux comprendre à quel point le processus actuel d'entraînement est fastidieux, regardons un exercice concret de C pour débutant. Une enseignante qui suit une classe de 40 étudiant·es, fournit la consigne suivante sur un serveur, comme premier exercice de la session.

Salut-moi

Un petit programme qui te salue avec ton nom complet.

Assure-toi d'avoir la même sortie que ce scénario, en répondant `John` et `Doe` manuellement.

```
> ./main
Quel est ton prénom ? John
Salut John, quel est ton nom de famille ? Doe
Passe une belle journée John Doe !
>
```

Démarre avec ce bout de code.

```
int main(int argc, char *argv[]) {
    // ???
}
```

Vérifie que ton programme ait terminé avec le code de fin 0, en lançant cette commande.

```
> echo $?
0
```

Un titre, une consigne et un scénario pour tester le bon fonctionnement sont fournis. L'enseignante annonce un temps alloué de dix minutes. Une fois la consigne récupérée et lue par un étudiant, il prend le code de départ et crée un nouveau fichier dans ses fichiers personnels. L'étudiant ouvre ensuite son IDE favori dans le dossier de l'exercice et configure la compilation avec CMake. Après trois minutes de mise en place, il peut enfin commencer à coder.

Une première solution est développée après deux minutes et peut être testée. Il lance un terminal, compile le code, rentre `John` et `Doe` et s'assure du résultat. Après relecture de l'output générée, il se rend compte d'une erreur sur `Passe une belle journée Doe` : seul le nom de famille s'affiche, le prénom a été oublié. Deux minutes pour tester son code se sont écoulées. Après une minute de correction, l'étudiant retourne dans son terminal et recommence le processus de validation. L'exercice est terminé juste à la fin du temps alloué et l'étudiant peut suivre la correction. S'il avait eu une erreur de plus, il aurait eu besoin de quelques minutes de plus. Certain·es étudiant·es à ses côtés n'ont pas eu le temps de finir et doivent s'arrêter.

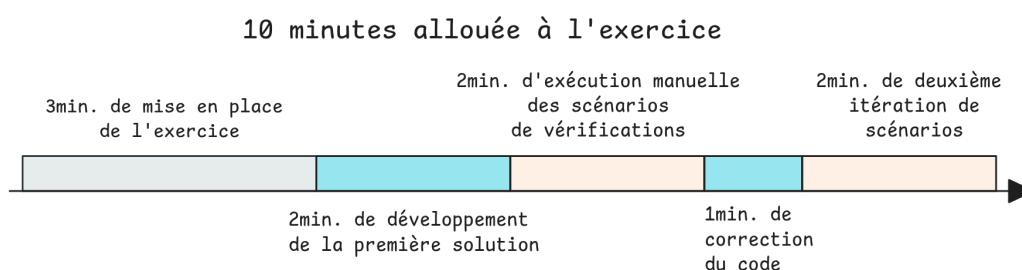


Figure 1 – Résumé visuel du temps estimé passé sur l'exercice par un étudiant débutant

En résumé, sur les dix minutes seulement trois ont été utilisées pour de l'écriture de code. Tout le reste a été perdu sur des tâches « administratives » autour de l'exercice.

Durant la correction, l'enseignante va présenter sa solution et demander s'il y a des questions. Certain·es étudiant·es les plus avancé·es poseront peut-être des questions sur l'approche ou une fonction spécifique. Une partie des étudiant·es penseront avoir tout compris puisque la sortie de leur programme est correcte. En réalité, les subtilités de la solution présentée n'auront pas forcément été intégrées. Ces incompréhensions pourraient impacter la suite de l'apprentissage qui s'appuierait alors sur une base fragile ou incomplète.

Une autre partie des étudiant·es pourraient être resté·es bloqué·es sur le début de l'exercice et auraient manqué une occasion de s'entraîner, sans que l'enseignante sans rendre compte.

Pour tous ces cas, il est facile de penser qu'il suffirait que l'étudiant·e pose une question et le problème serait résolu. En pratique, il peut être intimidant de poser des questions dans une grande classe ou selon la réaction des enseignant·es face aux questions basiques. Parfois, c'est tout un sujet qui est flou ou mal amené et pourtant il est rare d'entendre une question du type « *Je suis complètement paumé·e, vous pouvez réexpliquer ce que fait cette fonction ?* » ou encore « *Je ne sais pas ce qui est flou, mais je n'ai vraiment pas compris votre solution.* ».

Faire fonctionner le programme n'est que la première étape. Faire du code robuste, modulaire, lisible et performant demande des retours humains pour pouvoir progresser. Les étudiant·es moins expérimenté·es ne savent pas immédiatement si la compétence est acquise, comme le feedback n'arrive que dans les corrections des évaluations notées, plusieurs semaines plus tard.

Du côté de l'enseignante, en comptant uniquement sur les questions des étudiant·es, savoir si le message de l'exercice est passé reste un challenge, tout comme comprendre à quel point les concepts sous-jacents ont été acquis. Il est difficile aussi de savoir quand l'exercice doit se terminer. Peut-être qu'il aurait fallu 5 minutes de plus pour qu'une majorité ait le temps de finir ? Pour avoir accès aux réponses, elles doivent être manuellement rendues sur un serveur. Ce rendu prend à nouveau du temps pour chaque étudiant·e. Pour l'enseignante, récupérer, ouvrir et fermer 40 fichiers, prendrait trop de temps en classe.

Une autre approche serait de coder dans un fichier Google Docs partagé à toute la classe. L'enseignante a maintenant un moyen de relire au fur et à mesure, détecter les incompréhensions, mais les étudiant·es ont perdu toute l'expérience du développement en local. Dans Google Docs, il n'y a pas de couleur sur le code, pas d'auto-complétion et pas d'erreur de compilation visible dans le code. Tous les raccourcis, le formatage automatique et les informations au survol manquent terriblement. Pour tester leur programme, les étudiant·es doivent constamment copier leur code dans un fichier local.

En conclusion, le problème est que l'entraînement est fastidieux pour les étudiants, ce qui implique moins d'exercices effectués, moins de motivation à avancer et freine l'apprentissage en profondeur. Le manque de retour ralentit également la progression des compétences autour de la qualité du code produit. Les enseignant·es n'ont pas accès aux réponses des étudiant·es, ce qui empêche d'avoir une vision précise des incompréhensions et de donner de feedbacks.

L'approche de PLX

Ce travail de Bachelor vise à poursuivre le développement du projet PLX (3), application desktop écrite en Rust, VueJS (4) et TypeScript. Cette application permet aux étudiant·es de se concentrer pleinement sur l'écriture du code. PLX est inspiré de Rustlings (Terminal User Interface (TUI) pour apprendre le Rust), permettant de s'habituer aux erreurs du compilateur Rust et de prendre en main la syntaxe (5). PLX fournit actuellement une expérience locale similaire pour le C et C++.

Pour commencer à s'entraîner, les étudiant·es clonent un repository Git contenant tous les exercices. Ensuite, ils et elles peuvent travailler localement dans leur IDE favori, qui s'exécute en parallèle de PLX. Les scénarios de vérifications, exécutés auparavant manuellement, sont lancés automatiquement à chaque sauvegarde de fichier. Ces suites de tests automatisées, appelées « checks », permettent d'apporter à l'étudiant·e un feedback automatisé immédiat, riche et continu. Au lieu de perdre sept minutes sur dix sur des tâches « administratives », PLX en automatise la majorité et permet à l'étudiant·e de réduire ce temps à une minute.

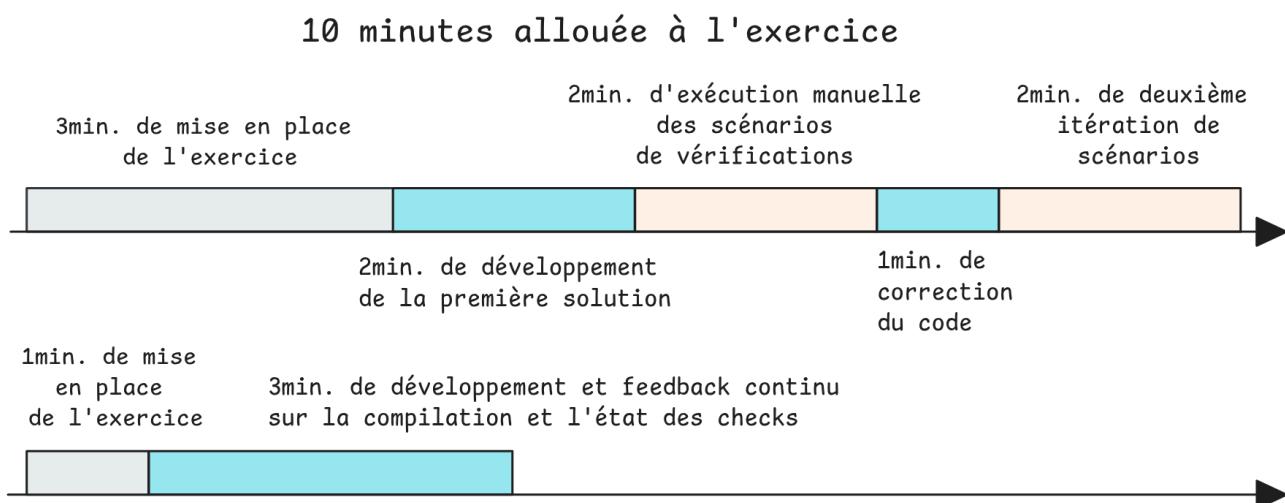


Figure 2 – Comparaison du temps nécessaire estimé sans et avec PLX

Ces checks restent pour l'instant assez primitifs, seulement l'output est comparée à celui attendu. D'autres vérifications plus avancées pourront être supportées dans le futur. Les enseignant·es rédigent le titre, la consigne, ainsi que les détails des checks dans des fichiers texte en format TOML.

Skills	Exos	
1 Introduction	1.1 Basic arguments usage	InProgress
2 Pointers	1.2 Basic output printing	InProgress
3 Parsing		
4 Structs		
5 Enums		

Figure 3 – Dans PLX, l’aperçu des listes de compétences et exercices dans un cours fictif, il est possible de parcourir les exercices et d’en démarrer un

```

intro/basic-args/main.c: In function 'main':
intro/basic-args/main.c:4:3: error: too few arguments to function 'printf'
  4 |   printf();
     | ^~~~~~
In file included from intro/basic-args/main.c:1:
/usr/include/stdio.h:363:12: note: declared here
 363 | extern int printf (const char * __restrict __format, ...);
     | ^~~~~~
```

Figure 4 – Une fois cet exercice de C lancé, le titre et la consigne sont visibles. Les erreurs de compilation sont directement affichés dans PLX, en préservant les couleurs

PLX - Full fictive course No live session

Basic arguments usage

The 2 first program arguments are the **firstname** and **number of legs of a dog**. Print a full sentence about the dog. Make sure there is at least 2 arguments, print an error if not.

Checks

C1: Joe + 5 legs

Arguments: Joe 5

Expected

```
The dog is Joe and has 5 legs
```

Diff

```
- the dog joe and has 5 legs
+ The dog is Joe and has 5 legs
```

C2: No arg -> error

Expected

```
Error: missing argument firstname and legs number
```

Diff

```
- error: missing argument firstname and legs number
+ Error: missing argument firstname and legs number
```

C main.c M intro/basic-args/m ...

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 3)
        printf(format: "error: missing
                     argument firstname and legs
                     number");
    else
        printf(format: "the dog joe
                     and has 5 legs");
}
```

Figure 5 – 2 checks qui échouent, avec la différence d'output pour facilement comprendre ce qui n'est pas correcte. L'IDE s'est ouvert automatiquement en parallèle.

plx

PLX - Full fictive course No live session

Basic arguments usage

The 2 first program arguments are the **firstname** and **number of legs of a dog**. Print a full sentence about the dog. Make sure there is at least 2 arguments, print an error if not.

Checks

C1: Joe + 5 legs

C2: No arg -> error

C3: One arg -> error

Figure 6 – Une fois tous les checks passés, tout passe au vert et l'exercice est terminé

Nouveaux défis

Le besoin de feedback humain pour les étudiant·es en plus du feedback automatisé, et celui de permettre aux enseignant·es d'accéder aux réponses, ne sont pas encore résolus par PLX. Ces nouveaux défis sont le point de départ des deux extensions majeures qui seront développées dans le cadre de ce travail.

Défi 1: Comment les enseignant·es peuvent voir le code et les résultats en temps réel ?

Comme mentionné précédemment, le rendu manuel d'exercices prend un peu de temps et ne sera pas fait fréquemment durant un entraînement. De plus, avoir accès à une archive de fichiers de code, demanderait encore de les compiler et lancer localement avant de pouvoir construire des statistiques de l'état des checks.

Comme l'application fonctionne localement et s'exécute à chaque sauvegarde, le code et les résultats des checks sont déjà connus par PLX. Il suffirait d'avoir un serveur central, qui héberge les sessions d'entraînement synchrones (appelées « sessions live »). A chaque changement, PLX pourrait ainsi envoyer le code et l'état des checks. Ces informations pourraient être transférées par le serveur vers le client PLX de l'enseignant·e, pour les afficher sur un tableau de bord dédié.

Ce tableau de bord permettra aux enseignant·es de rapidement comprendre les lacunes des étudiant·es, en relisant les différentes réponses affichées. Grâce à l'état des checks, il sera facile de voir si la classe a terminé l'exercice ou de filtrer les réponses pour concentrer sa relecture. Il sera possible de sélectionner certaines réponses particulières pour les afficher au beamer, pouvoir les commenter ou demander à la classe de critiquer constructivement le code choisi.

Pour accéder aux exercices, les étudiants doivent cloner le repository Git du cours sur leur machine à travers l'interface de PLX pour qu'il puisse avoir une liste de cours disponibles localement. Nous prenons en exemple le cours de PRG2, cours de C à la HEIG-VD. Une session live peut être démarrée par un·e enseignant·e pour un cours donné (le cours étant unique par l'origine du repository Git) et les étudiant·es peuvent la rejoindre. En ouvrant le cours dans PLX, la liste des sessions ouvertes liées au repository est affichée et les étudiant·es peuvent choisir celle de leur enseignant·e. Durant la session, l'enseignant·e définit une liste d'exercice et les lance l'un après l'autre au rythme choisi.

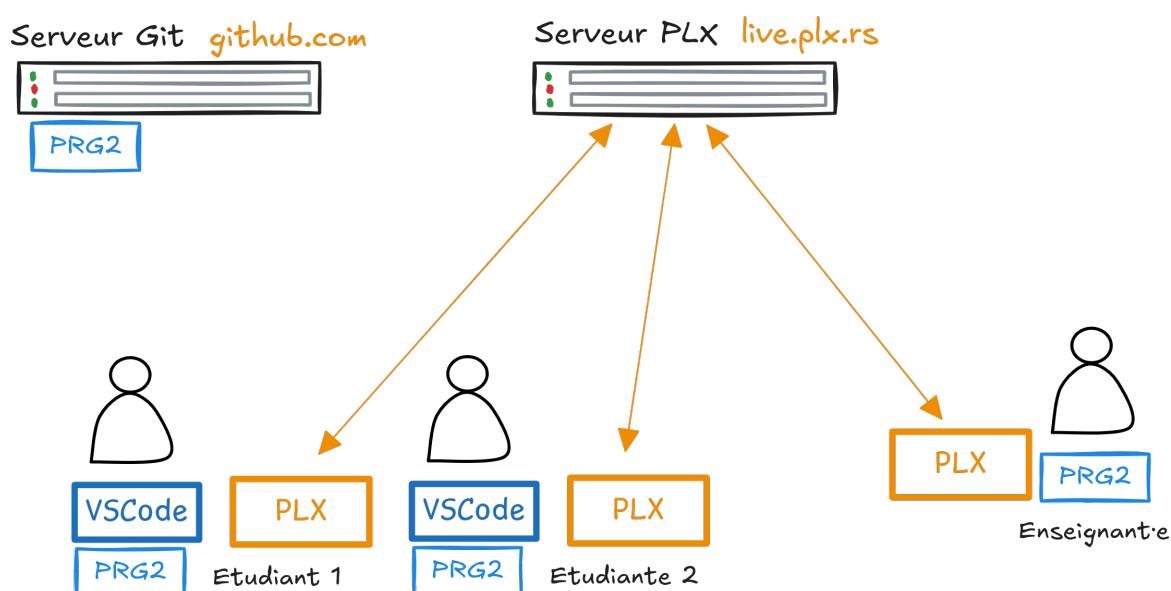


Figure 7 – L'enseignant·e et les étudiant·es sont connectés à une session live sur un serveur PLX, du repository « PRG2 »

L'exercice en cours est affiché sur tous les clients PLX. À chaque sauvegarde d'un fichier de code, le code est compilé et les checks sont lancés. Les résultats des checks et le code modifié seront envoyés à l'enseignant de la session.

Ce premier défi nécessite le développement d'un serveur central et la spécification du protocole de communication entre clients et serveurs PLX. Ce protocole s'appuiera sur un autre protocole de communication bidirectionnel pour permettre cette expérience en direct en classe.

Défi 2: Comment faciliter la rédaction et la maintenance des exercices ?

La rédaction de contenu sous forme de fichier textes, au lieu de l'approche classique de formulaires, semble particulièrement plaire en informatique. En effet, de nombreux enseignant·es à la HEIG-VD rédigent une part de leur contenu (exercices, slides, consignes de laboratoires, évaluations) dans divers formats textuels.

Un exemple d'usage du Markdown est le recueil d'exercices du cours de PRG2 (6). On note également l'usage de balises HTML `<details>` et `<summary>`, pour rendre disponible la solution tout en la cachant par défaut. Pour combler le manque de mise en page du Markdown, d'autres enseignant·es utilisent Latex ou Typst (7).

Pour faciliter l'adoption de PLX, nous avons besoin d'un format de données simple à prendre en main, pour décrire les différents types d'exercices supportés. Si on reprend l'exercice présenté plus tôt, qu'on le rédige en Markdown, en y ajoutant la solution dans le même style du recueil de PRG2 (8), cela donne le Snippet 1.

```
# Salut-moi
Un petit programme qui te salue avec ton nom complet.

Assure-toi d'avoir la même sortie que ce scénario, en répondant `John` et `Doe`
manuellement.
```
> ./main
Quel est ton prénom ? John
Salut John, quel est ton nom de famille ? Doe
Passe une belle journée John Doe !
>
```

Démarre avec ce bout de code.
```
c
int main(int argc, char *argv[]) {
 // ???
}
```

Vérifie que ton programme ait terminé avec le code de fin 0, en lançant cette commande.
```
sh
> echo $?
0
```

<details>
<summary>Solution</summary>

```
c
#include <stdio.h>

#define NAME_MAX_SIZE 100
int main(int argc, char *argv[]) {
 char firstname[NAME_MAX_SIZE];
 char lastname[NAME_MAX_SIZE];

 printf("Quel est ton prénom ? ");
 fflush(stdout);
 scanf("%s", firstname);

 printf("Salut %s, quel est ton nom de famille ? ", firstname);
 fflush(stdout);
 scanf("%s", lastname);

 printf("Passe une belle journée %s %s !\n", firstname, lastname);
 return 0;
}
```

</details>
```

Snippet 1 – Exemple d'exercice de programmation, rédigé en Markdown

Ce format en Snippet 1 est pensé pour un document lisible par des humains. Cependant, si on voulait pouvoir automatiser l'exécution du code et des étapes manuelles de rentrer le prénom, le nom et de vérifier l'output, nous aurions besoin d'extraire chaque information sans ambiguïté. Hors cette structure, bien que reproductible manuellement sur d'autres exercices, n'est pas assez standardisée pour une extraction automatique.

En effet, sans comprendre le langage naturel, comment savoir que `John` et `Doe` sur l'extrait du Snippet 2 doivent être rentrés à la main et ne font pas partie de l'output ?

```
Assure toi d'avoir la même sortie que ce scénario, en répondant `John` et `Doe` manuellement.
```
> ./main
Quel est ton prénom ? John
Salut John, quel est ton nom de famille ? Doe
Passe une belle journée John Doe !
>
```

```

Snippet 2 – Extrait 1 du Snippet 1 décrivant le scénario à tester

Et si on avait différents scénarios, comment pourrait-on les décrire et différencier ? Comment distinguer la consigne utile du reste des instructions génériques ? La partie « *en répondant John et Doe manuellement* » ne devrait pas apparaître si le scénario a pu être automatisé. L'étudiant·e pourra comprendre le scénario simulé à travers l'affichage du check.

Sur le Snippet 3, comment le parseur peut détecter qu'on parle du code d'exit du programme et que ce code doit valoir zéro ?

```
Vérifie que ton programme ait terminé avec le code de fin 0, en lançant cette commande.
```
sh
> echo $?
0
```

```

Snippet 3 – Extrait 2 du Snippet 1 décrivant le code de fin

Le découpage mentale des informations peut sembler simple en tant qu'humain avec le langage naturel, mais devient une tâche impossible pour un parseur qui doit être fiable. Le langage naturel peut être compris par des modèles de langages mais on exclut l'usage de l'intelligence artificielle pour ce parseur, car nous avons besoin qu'il soit prédictible et rapide.

De plus, ce format possède plusieurs parties qui demandent plus de travail à la rédaction. Le code de la solution est développé dans un fichier `main.c` séparé et doit être copié manuellement. Une partie du texte sur Snippet 2 comme *Assure toi d'avoir le même output que ce scénario* est générique et doit pourtant être constamment répétée à chaque exercice pour introduire le snippet. L'output est à maintenir à jour avec le code de la solution, si celle-ci évolue, on risque d'oublier de mettre à jour la consigne de l'exercice.

Maintenant qu'il est clair que le Markdown seul n'est pas adapté, regardons du côté des formats structurés. L'option la plus rapide et facile à mettre en place serait simplement de définir un schéma JSON à respecter. On aurait d'abord un champ pour le titre (sous la clé `exo` pour raccourcir le mot `exercice`) et la consigne.

Ensuite une liste de checks serait fournie. Chaque check serait défini par un titre et une séquence d'opérations à effectuer. Chaque opération serait de type `see` (ce que l'on s'attend à « voir » dans l'output), `type` (ce qu'on tape dans le terminal) et finalement `exit` (pour définir le code d'exit attendu). Il serait pratique de définir cette séquence dans un objet, avec en clé `see`, `type` ou `exit` et en valeur, un paramètre. Comme les clés des objets en JSON n'ont pas d'ordre et doivent être uniques (9), nous ne pourrions pas répéter plusieurs étapes `see`. Nous devons décrire la séquence comme un tableau `[]` d'objets `{}`. Voici un exemple d'usage de ce schéma sur le Snippet 4.

```
{
  "exo": "Salut-moi",
  "instruction": "Un petit programme qui te salue avec ton nom complet.",
  "checks": [
    {
      "name": "Il est possible d'être salué avec son nom complet",
      "sequence": [
        { "kind": "see", "value": "Quel est ton prénom ?" },
        { "kind": "type", "value": "John" },
        { "kind": "see", "value": "Salut John, quel est ton nom de famille ?" },
        { "kind": "type", "value": "Doe" },
        { "kind": "see", "value": "Passe une belle journée John Doe !" },
        { "kind": "exit", "code": 0 }
      ]
    }
  ]
}
```

Snippet 4 – Equivalent JSON de l'exercice défini sur le Snippet 1

Cet exemple d'exercice en Snippet 4 est minimal, mais montre clairement que rédiger dans ce format serait fastidieux. Si la consigne s'étalait sur plusieurs lignes, nous aurions du remplacer manuellement les retours à la ligne par des `\n`. Au-delà du texte brut, tous les guillemets, deux points, crochets et accolades nécessaires demande un effort de rédaction important.

Un autre format plus léger à rédiger est le YAML, regardons ce que cela donne:

```
exo: Salut-moi
instruction: Un petit programme qui te salue avec ton nom complet.
checks:
  - name: Il est possible d'être salué avec son nom complet
    sequence:
      - kind: see
        value: Quel est ton prénom ?
      - kind: type
        value: John
      - kind: see
        value: Salut John, quel est ton nom de famille ?
      - kind: type
        value: Doe
      - kind: see
        value: Passe une belle journée John Doe !
      - type: exit
        value: 0
```

Snippet 5 – Equivalent YAML de l'exercice défini sur le Snippet 1

Le YAML nous a permis ici de retirer tous les guillemets, les accolades et crochets. Cependant, malgré sa légereté, il contient encore plusieurs points de friction:

- Les tirets sont nécessaires pour chaque élément de liste et les deux points pour chaque clé
- Pour avoir plus d'une information par ligne, il faut ajouter une paire d'accolades autour des clés (`- { kind: see, value: Passe une belle journée John Doe ! }`)
- Les tabulations sont difficiles à gérer dès qu'on dépasse 3-4 niveaux, elles sont aussi nécessaires pour du contenu multiligne
- Certaines situations nécessitent encore des guillemets autours des chaînes de caractères

L'intérêt clair du YAML, tout comme le JSON est la possibilité de définir des paires de clés/valeurs, ce qui n'est pas possible en Markdown. On pourrait définir une convention par dessus Markdown:

définir qu'un titre de niveau 1 est le titre de l'exercice, qu'un bloc de code sans langage défini est l'output ou encore que le texte entre le titre et l'output est la consigne.

Quand on arrive sur des champs plus spécifiques aux exercices de programmation, cette idée de convention au dessus du Markdown ne fonctionne plus vraiment. Comment définir le code d'exit attendu? Comment définir la commande pour stopper un programme? Ou encore définir les parties de l'output qui sont des entrées utilisateurs ?

Pour résoudre ces problèmes, nous proposons une nouvelle syntaxe, nommée DY, à mi-chemin entre le Markdown et le YAML, concise et compacte. Voici un exemple en Figure 8.

```
exo Salut-moi
Un petit programme qui te salue avec ton nom complet.

check Il est possible d'être salué avec son nom complet
see Quel est ton prénom ?
type John
see Salut John, quel est ton nom de famille ?
type Doe
see Passe une belle journée John Doe !
exit 0
```

Figure 8 – Equivalent de l'exercice du Snippet 1, dans une version préliminaire de la syntaxe DY

Dans cette syntaxe DY, nous reprenons les idées de `see`, `type`, et `exit`. Nous avons gardé les clés du YAML mais retiré le superflu: les tabulations, les deux points, les tirets et les accolades. Les différentes informations sont séparées par la fin de ligne avant une autre clé valide. La consigne est définie dans la suite du titre et peut s'étendre sur plusieurs lignes. Le Markdown est toujours supporté dans le titre et la consigne.

Ce deuxième défi demande d'écrire un parseur de cette nouvelle syntaxe. Ce n'est que la première étape, car lire du texte structuré blanc sur fond noir sans aucune couleur, sans feedback sur la validité du contenu, mène à une expérience un peu froide. En plus du parseur, il est indispensable d'avoir un support solide dans les IDE modernes pour proposer une expérience d'édition productive.

```

exo.py

1 exo Salut-moi
2 Un petit programme qui te salue avec ton nom complet.
3
4 check             a name for the check is required
5 see Quel est ton prénom ?
6 type John
7 see Salut John, qu'est est ton nom de famille ?
8 type Doe
9 see Passe une belle journée John Doe !
10 exit 0
11
12 ch|check Define a new automated check in this exercise
13 exo
14
15
16
17
18
19
20

```

Figure 9 – Aperçu de l’expérience souhaitée de rédaction dans un IDE

On voit dans la Figure 9 que l’intégration inclut deux fonctionnalités principales

1. le surlignage de code, qui permet de coloriser les clés et les propriétés, afin de bien distinguer les clés du contenu
2. l’intégration avancée des erreurs du parseur et de la documentation à l’éditeur. On le voit en ligne 4, après la clé `check` une erreur s’affiche pour le nom manquant. En ligne 19, l’auto-complétion facilite la découverte et rédaction en proposant les clés valides à cette position du curseur.

Pour convaincre les plus perplexes des lecteur·ices, il peut être intéressant de comprendre la réflexion stratégique derrière ce projet, maintenant que les solutions standards ont pu être comparées. Là où certain·es auraient simplement pris le YAML, TOML ou un autre format connu par habitude, ne faisons le choix de ne pas se contenter de l’existant. Dans un contexte professionnelle, il aurait peut-être été difficile de justifier le développement d’une solution, « juste pour optimiser le YAML et le Markdown », dans un contexte académique, nous avons la chance d’avoir du temps.

La conception de la syntaxe DY est similaire à celle de l’éditeur de texte Neovim (fork moderne de Vim) (10). Prendre en main Neovim, le personnaliser et s’y habituer prend un temps conséquent. De nombreux raccourcis d’édition du texte sont très différents des autres éditeurs. Dans Neovim au lieu de `ctrl+c` pour copier, on utilise `y`. Pour sélectionner un mot à droite et le supprimer, plutôt que `ctrl+shift+droite` puis `supprimer`, on tape simplement `dw` (**d**elete **w**ord). L’outil a été entièrement conçu pour être optimisé en définissant des raccourcis facile et rapide à taper. Les premières semaines d’usage de l’outil sont pénibles, ce n’est qu’en suite que l’on prend goût à la rapidité et l’agilité d’édition.

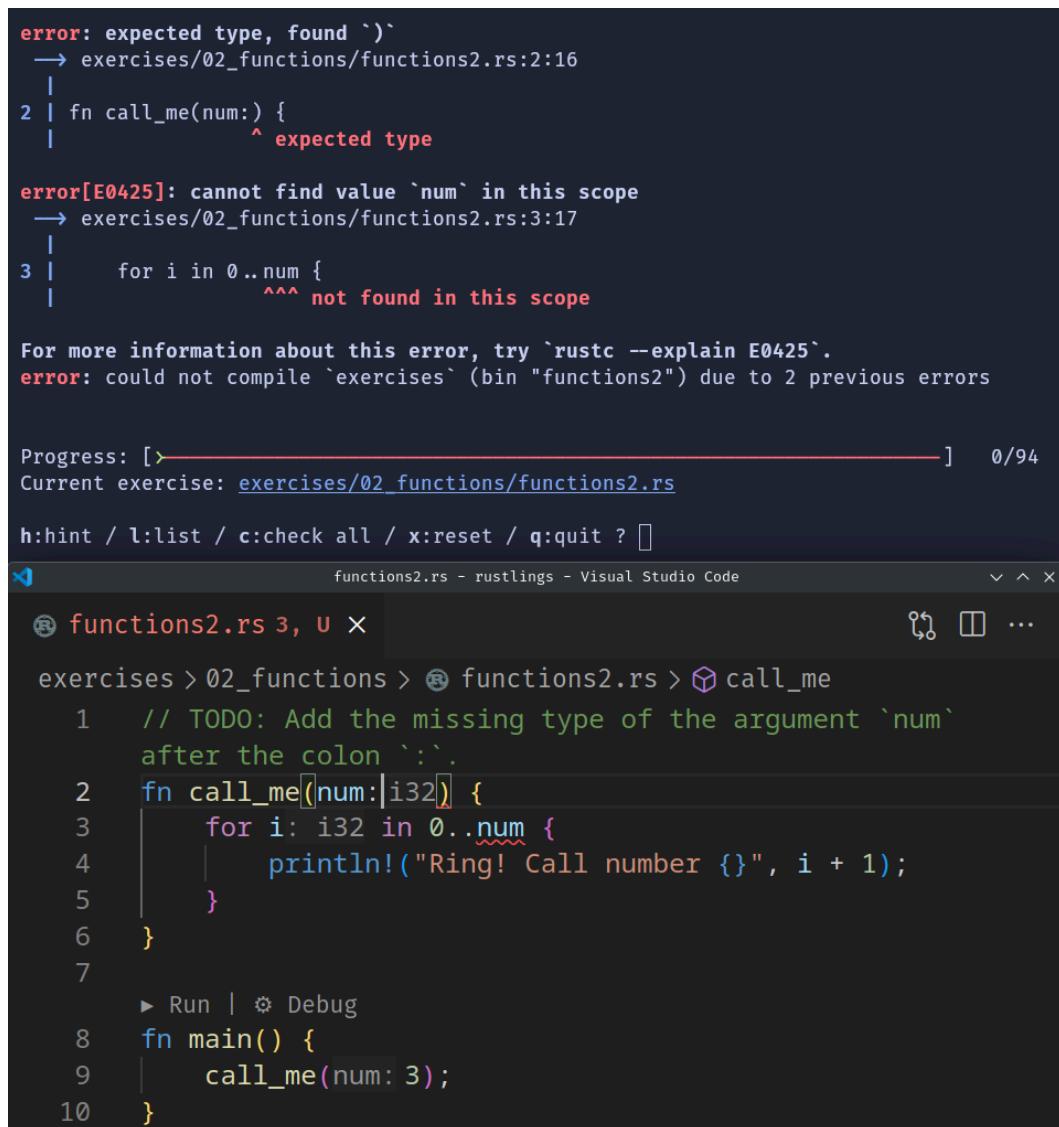
Sur plusieurs années, les enseignant·es passeront des centaines d’heures à retranscrire, modifier ou créer des exercices. Nous préférons passer du temps au développement, à l’intégration et à la documentation, pour optimiser la rédaction à long terme. A long terme, nous faisons le pari

qu'un·e enseignant·e non initié·e y gagnera, par rapport au temps de mise en place de l'outil et d'apprentissage durant la première heure. Dans certains contextes, si le YAML n'est pas connu, la syntaxe DY pourrait être même plus simple à prendre en main. Une fois initié à la syntaxe, la rédaction facilitée encourage à rédiger souvent et rapidement de nouveaux exercices, ce qui améliore la quantité et la qualité de la pratique pour les étudiant·es.

Cette nouvelle syntaxe, son parseur et support d'IDE permettront de complètement remplacer le format TOML actuellement utilisé par PLX.

Solutions existantes

Faire des exercices de programmation couverts par des suites de tests automatisées n'est pas une idée nouvelle en soit. Comme mentionné dans l'introduction, PLX est inspiré de Rustlings. Cette TUI propose une centaine d'exercices de Rust avec des erreurs de compilation à corriger ou des tests unitaires à faire passer. Le site web de Rustlings recommande de faire ces exercices en parallèle de la lecture du *Rust book* (la documentation officielle) (5).



The screenshot shows a Visual Studio Code interface with a dark theme. At the top, there's a terminal window displaying the output of a rustc compilation. The error messages indicate that the code is missing type annotations and variable declarations. Below the terminal, the main code editor shows a file named 'functions2.rs'. The code contains a function 'call_me' that prints numbers from 0 to num. A cursor is visible at the end of the line 'fn call_me(num:i32) {'. The status bar at the bottom shows 'Progress: [] 0/94' and 'Current exercise: exercises/02_functions/functions2.rs'.

```

error: expected type, found `)`
→ exercises/02_functions/functions2.rs:2:16
|
2 | fn call_me(num:) {
|           ^ expected type

error[E0425]: cannot find value `num` in this scope
→ exercises/02_functions/functions2.rs:3:17
|
3 |     for i in 0..num {
|           ^^^ not found in this scope

For more information about this error, try `rustc --explain E0425`.
error: could not compile `exercises` (bin "functions2") due to 2 previous errors

Progress: [ ] 0/94
Current exercise: exercises/02_functions/functions2.rs

h:hint / l:list / c:check all / x:reset / q:quit ? 
```

functions2.rs - rustlings - Visual Studio Code

functions2.rs 3, U X

exercises > 02_functions > functions2.rs > call_me

```

1 // TODO: Add the missing type of the argument `num`
2 fn call_me(num:i32) {
3     for i: i32 in 0..num {
4         println!("Ring! Call number {}", i + 1);
5     }
6 }
7
8 ► Run | ⚙ Debug
9 fn main() {
10     call_me(num: 3);
11 }
```

Figure 10 – Rustlings en action dans le terminal en haut et l'IDE VSCode en bas

De nombreux autres projets se sont inspirées de ce concept, `clangs` pour le C (11), `golings` pour le Go (12), `ziglings` pour Zig (13) et même `haskellings` pour Haskell (14) ! Ces projets incluent une suite d'exercice et une TUI pour les exécuter pas à pas, afficher les erreurs de compilation ou les cas de tests qui échouent, pour faciliter la prise en main des débutant·es.

Chaque projet se concentre sur un langage de programmation et crée des exercices dédiés. PLX prend une approche différente: en plus d'être une application desktop, il n'y a pas d'exercice proposé et PLX supporte de multiples langages. Le contenu sera géré indépendamment de l'outil, permettant aux enseignant·es d'intégrer leur propre contenu.

Plusieurs plateformes web similaires existent, comme CodeCheck (15), qui permet de configurer l'exercice en ajoutant des commentaires directement dans le code de la solution. Par exemple, un commentaire `//HIDE` va cacher une ligne, `//EDIT` va définir un bloc éditable, `//ARGS` indique des arguments à passer au programme ou encore `//CALL 1 2 3` pour appeler une fonction avec les arguments 1, 2 et 3.

Complete the following program to compute the average length of the words. Invoke the appropriate method to compute the length of each word, compute the sum of the five results, and divide by 5.0.

AverageTester.java

```

1  public class AverageTester
2  {
3      public static void main(String[] args)
4      {
5          String word1 = "Mary";
6          String word2 = "had";
7          String word3 = "a";
8          String word4 = "little";
9          String word5 = "lamb";
10         int length1 = word1.length();
11         int length2 = word2.length();
12         // todo
13         double average = (length1 + length2) / 2.0;
14         System.out.println(average);
15         System.out.println("Expected: 3.6");
16     }
17 }
18 }
```

[CodeCheck](#)

[Reset](#)

[Download](#)

Running program

AverageTester.java:

Actual	Expected
3.5	3.6
Expected: 3.6	Expected: 3.6

Figure 11 – Aperçu d'un exercice de Java sur CodeCheck, avec un code qui compile mais un résultat erroné (16)

Le code est exécuté sur le serveur et l'édition se fait dans le navigateur dans un éditeur simplifié. L'avantage est la simplicité d'usage et le système de pseudo commentaires pour configurer l'exercice depuis la solution directement. Comme désavantage par rapport à PLX c'est le temps de compilation qui est plus lent qu'une compilation en local et l'expérience d'édition en ligne reste trop minimale pour passer des heures sur des exercices. Chaque exercice a son propre URL pour l'édition et un autre pour l'entraînement, ce qui peut rendre fastidieux le déploiement de dizaines d'exercices à la chaîne.

Ces solutions existantes sont intéressantes mais ne couvrent qu'une partie des besoins de PLX. Le plus gros manque est l'impossibilité de faire des sessions live.

Glossaire

L'auteur de ce travail se permet un certain nombre d'anglicismes quand un équivalent français n'existe pas. Certaines constructions de programmations bien connues comme les `strings` au lieu d'écrire `chaînes de caractères` sont également utilisées. Certaines sont spécifiques à certains langages et sont décrites ci-dessous pour aider à la lecture.

- `POC` : *Proof Of Concept*, preuve qu'un concept fonctionne en pratique. Consiste ici en un petit morceau de code développé juste pour démontrer que le concept est fonctionnel, sans soin particulier apporté à la qualité de l'implémentation. Ce code n'est pas réutilisé par la suite, il sert seulement d'inspiration pour l'implémentation réelle.
- `output` : flux de texte de la sortie standard du programme / texte affiché dans la console par le programme
- `exo` : abréviation familière de `exercice`. Elle est utilisée dans la syntaxe DY, le code et le protocole pour rendre la rédaction plus concise.
- `check` : nom choisi pour décrire un ou plusieurs tests unitaires ou vérifications automatisées du code
- `Cargo` : le gestionnaire de dépendances, de compilation et de test des projets Rust
- `crate` : la plus petite unité de compilation avec Cargo, concrètement chaque projet contient un ou plusieurs dossiers avec un `Cargo.toml`, ce sont des crates locales. Les dépendances sont également des crates qui ont été publié sur le registre officiel.
- `Cargo.toml` , configuration de Cargo dans un projet Rust définit les dépendances (les crates) et leurs versions minimum à inclure dans le projet, équivalent du `package.json` de NPM
- `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
- `parsing` ou `désérialisation` : processus d'un parseur, visant à extraire de l'information brute vers une forme structurée facilement manipulable
- `sérialisation` : inverse du processus du parseur, qui vise à transformer une structure de données quelconque en une forme brute (une string par exemple) afin de la stocker sur le disque ou l'envoyer à travers le réseau
- `struct` : structure de données regroupant plusieurs champs, disponible en C, en Rust et d'autres langages inspirés
- `backtick` : caractère accent grave utilisé sans lettre, délimiteur fréquent de mention de variable ou fonction dans un rapport en Markdown
- `README` ou `README.md` : Point d'entrée de la documentation d'un repository Git, généralement écrit en Markdown, affiché directement sur la plupart des hébergeurs de repository Git
- `regex` : raccourcis pour les expressions régulières
- `snippet` : court morceau de code ou de données
- `querystring` : partie d'un URL après le `?` tel que `?action=send&id=23` , qui se termine au premier `#` rencontré

Planification

Déroulement

Le travail commence le 17 février 2025 et se termine le 24 juillet 2025. Sur les 16 premières semaines, soit du 17 février 2025 au 15 juin 2025, la charge de travail représente 12h par semaine. Les 6 dernières semaines, soit du 16 juin 2025 au 24 juillet 2025, ce travail sera réalisé à plein temps.

Un rendu intermédiaire noté est demandé le 23 mai 2025 avant 17h et le rendu final est prévu pour le 24 juillet 2025 avant 17h.

La défense sera organisée entre le 25 août 2025 et le 12 septembre 2025.

Planification initiale

Note: cette planification est reprise du cahier des charges original en annexe, avec quelques corrections mineures.

En se basant sur le calendrier des travaux de Bachelor, voici un aperçu du découpage du projet pour les différents rendus.

Rendu 1 - 10 avril 2025 - Cahier des charges

- Rédaction du cahier des charges.
- Analyse de l'état de l'art des parseurs, des formats existants de données humainement éditables, du surlignage de code et des serveurs de langages.
- Analyse de l'état de l'art des protocoles bidirectionnels temps réel (WebSocket, gRPC...) et des formats de sérialisation (JSON, Protobuf, ...).
- Prototype avec les librairies disponibles de parsing et de serveurs de langages en Rust, choix du niveau d'abstraction espéré et réutilisation possible.

Rendu 2 - 23 mai 2025 - Rapport intermédiaire

- Rédaction du rapport intermédiaire.
- Définition de la syntaxe DY à parser, des clés liés à PLX, la liste des vérifications et des erreurs associées.
- Définition d'un protocole de synchronisation du code entre les participants d'une session.
- Prototype d'implémentation de cette synchronisation.
- Prototype des tests automatisés sur le serveur PLX.
- Définition du protocole entre les clients PLX et le serveur pour les entraînements live.

Moitié des 6 semaines à temps plein - 4 juillet 2025

- Écriture des tests de validation du protocole et de gestion des erreurs.
- Développement du serveur PLX.
- Rédaction du rapport final par rapport aux développements effectués.

Rendu 3 - 24 juillet 2025 - Rapport final

- Développement d'une librairie `dy`.
- Intégration de cette librairie à PLX.
- Rédaction de l'affiche et du résumé publiable.
- Rédaction du rapport final.

Planification finale

La rédaction du rapport de l'état de l'art a pris beaucoup de temps au début du projet, en plus de la finition du cahier des charges, pour bien creuser les cinq sujets concernés par ce travail. Le nombre important de technologies à investiguer, en parallèle du développement des POCs, a retardé la spécification du protocole de communication et de la syntaxe DY.

De manière générale, la rédaction du rapport a pris important dans ce travail. Pour apprendre petit à petit à mieux contextualiser, à expliquer d'abord les problèmes puis les solutions, il a fallu passer par de multiples relectures et éditions, demander des retours à d'autres personnes et intégrer ces retours.

L'écriture des tests de validation du protocole, qui était prévue avant de commencer le serveur, s'est finalement intégrée au développement du serveur. Cela avait plus du sens d'écrire les tests au fur et à mesure que les nouveaux messages du protocole étaient définis, pour s'adapter aux nombreuses ajustements des structures de données et de l'architecture.

Après la préparation du développement du serveur qui a permis de spécifier le protocole et le comportement attendu du client et du serveur, le développement a été plus rapide que prévu. Nous pensions passer deux semaines de développement et une semaine pour les tests et de rapport. Au final, l'historique Git nous montre que c'est plutôt en une semaine, entre le 24 juin et le 30 juin, que la majorité du serveur a pu être mise en place. Cela n'a pas permis de prendre de l'avance sur le programme, car l'intégration dans l'application desktop de PLX n'a pas été évidente.

Heureusement, la deuxième partie de développement autour de notre syntaxe DY a été également plus courte que prévue. Entre le 13 et 18 juillet, le développement du parseur de son intégration dans PLX desktop et dans un CLI ont pu être menés à bien.

Malgré ces décalages, nous avons réussi à développer et documenter tous les éléments planifiés.

Ce que l'on peut retenir comme apprentissage de cette planification, c'est que le développement, lorsqu'il est bien préparé en amont, peut aller plus vite que prévu. Au contraire, le temps de rédaction et raffinage du rapport est souvent le double ou le triple du temps estimé au départ.

Le fait de fixer des dates de relectures externes avec des collègues ou de proposer de montrer notre application à une personne qui pourrait être intéressé de l'utiliser, est un vrai moteur pour avancer plus rapidement et se concentrer sur les parties les plus importantes.

État de l'art

Cette section explore l'état de l'art de cinq sujets liés aux deux défis de ce travail. Avant de développer la syntaxe DY, une recherche est faite autour des **syntaxes existantes moins répandues** qui ont des objectifs proches à la nôtre. Les **librairies de parsing** en Rust sont discutées ensuite pour comprendre si elles peuvent nous aider à l'implémentation du parseur. Pour inclure la compréhension du parseur directement dans les IDE, nous verrons comment **les serveurs de langage** permettent d'améliorer grandement l'expérience d'édition. Nous parlerons également des techniques de **surlignage de code**, dans les IDEs et sur le web, qui permettent de rendre notre syntaxe agréable à lire.

Pour conclure ces recherches, le défi du serveur de session live a demandé d'explorer les **protocoles de communication bidirectionnels**, pour nous permettre d'envoyer et de recevoir des messages en temps réel. Ce dernier sujet inclut aussi une comparaison entre **formats textes et binaires de sérialisation** des messages.

En plus de la comparaison des solutions existantes, quelques **POCs** sont développés pour découvrir et tester le fonctionnement des solutions choisies. L'implémentation est disponible dans le dossier `pocs` du repository Git de la documentation du projet. Ce dossier est accessible sur <https://github.com/samuelroland/tb-docs/tree/main/pocs>

Format de données humainement éditables existants

Avant de commencer ce travail conséquent de créer une nouvelle syntaxe, il est nécessaire de s'assurer qu'il n'existe pas d'autres librairies qui existent déjà et qui pourraient apporter la même expérience, simplicité et rapidité de rédaction. Nous avons aussi besoin d'avoir une intégration Rust puisque PLX est développé en Rust. Nous cherchons aussi une validation du contenu intégrée à l'éditeur, pour éviter des allers-retours constants entre l'éditeur et l'affichage d'erreurs de rédaction dans PLX.

Les parseurs JSON vérifient que le document est correcte, mais le choix des clés et des valeurs n'est pas vérifié. C'est pour cette raison que le projet JSON Schema (17) existe. Un schéma JSON définit un ensemble de clés valides, les types attendus pour chaque valeur, les champs requis et optionnels. L'intégration de ce projet dans l'IDE permet d'intégrer des erreurs lorsque des structures ne respecte pas le schéma et facilite la rédaction avec l'autocomplétion des clés et des valeurs. Nous cherchons une solution qui mixe dans un seul outil la définition de la syntaxe et sa validation.

La recherche se concentre sur les projets qui visent à créer des meilleures alternatives aux formats bien répandus ou qui ont un lien avec l'éducation. On ignore aussi les projets dont la spécification ou l'implémentation n'est pas encore utilisable en production. Ainsi, le langage de balise pour les recettes de cuisines Cooklang (18) n'est pas présenté. La recherche n'est pas évidente comme il existe de nombreuses manières de nommer ce que l'on cherche: langage de balise (*markup language*), format de donnée, syntaxe, langage de donnée, langage spécifique à un domaine (*Domain Specific Language - DSL*), ... La recherche a principalement été faite en anglais avec les mots-

clés suivants la barre de recherche de Google, Github.com et de crates.io: `data format`, `syntax`, `human friendly`, `alternative to YAML`, `human writable`, et `human readable`.

KHI - Le langage de données universel

D'abord nommée UDL (*Universal Data Language*) (19), 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, les balises, les structures, les listes, les tuples, les tables/matrices, les énumérations et les arbres hiérarchiques sont supportés.

```
{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 6 – Un exemple simplifié de KHI tiré du README (20) décrivant un exemple d'article d'encyclopédie.

Les objectifs sont la polyvalence, un format source (fait pour être rédigé à la main), l'esthétisme et la simplicité. Le Snippet 6 permet de percevoir l'intérêt qu'une combinaison plus légère de toutes ces structures de données peut avoir. Cependant, PLX n'a pas besoin d'autant de possibilités, les différents séparateurs `@`, `{`, `,`, `;`, `[`, `\`, etc. sont nécessaires pour que le parseur puisse différencier ces structures, mais créent une charge mentale additionnelle durant la rédaction. De plus, une implémentation en Rust est proposée (21) mais son dernier commit date du 11.11.2024, en plus du fait que le projet contient encore de nombreux `todo!()` dans son code.

Bitmark - le standard des contenus éducatifs digitaux

De nombreux formats de données existent pour décrire du contenu éducatif digital au vu de toutes les plateformes existantes autour de l'éducation et de l'enseignement. Cette diversité de formats rend l'interopérabilité très difficile, freine l'accès à la connaissance. Ces limites restreignent les créateurs de contenus et les éditeurs dans les possibilités de migration entre plateformes ou de publication du même contenu à différents endroits.

Bitmark est un standard open-source (22), qui vise à uniformiser tous ces formats pour améliorer l'interopérabilité (23). Leur stratégie est de définir un format basé sur le contenu (*Content-first*) plutôt que son rendu (*Layout-first*) permettant de supporter un affichage sur un maximum de plateformes, incluant les appareils mobiles (23). C'est la Bitmark Association en Suisse à Zurich qui développe ce standard, notamment à travers des Hackatons organisés en 2023 et 2024 (24).

Le standard permet de décrire du contenu statique comme des articles et du contenu interactif sur des quiz de divers formats. Deux équivalents sont définis : le *bitmark markup language* et le

bitmark JSON data model (25). La partie quiz 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, des audios à enregistrer et de nombreux autres types d'exercices.

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

Snippet 7 – Un exemple de question à choix multiple tiré de leur documentation (26).

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 8 – Equivalent de Snippet 7 dans le Bitmark JSON data model (26)

La plateforme Open Taskpool, qui met à disposition des exercices d'apprentissage de langues (27), fournit une API JSON utilisant le bitmark JSON data model.

```
curl "https://taskpool.taskbase.com/exercises?translationPair=de→en&word=school&exerciseType=bitmark.cloze"
```

Snippet 9 – Requête HTTP à Open Taskpool pour demander une paire de mots de l'allemand vers l'anglais, autour du mot `school` et de format `cloze` (texte à trou).

```

"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 10 – Extrait de la réponse au Snippet 9, respectant le standard Bitmark (28). La phrase `There used to be a __ here.` doit être complétée par `school`, en s'aider du texte en allemand.

Une autre plateforme, Classtime, utilise Bitmark pour son système d'import et export de questions (29). On voit dans leur documentation (30) que le système de création d'exercices reste basé sur des formulaires.

Ces deux 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 et la présence d'un équivalent JSON, la spécification du *markup* n'a peut-être pas été optimisée pour la rédaction à la main directement. En plus, Bitmark ne spécifie pas de type d'exercices programmation nécessaires à PLX. On salue au passage l'envie de standardiser le format des différentes plateformes, à long-terme cela ne peut que simplifier la vie des enseignant·es dans la gestion de leur contenu et augmenter la qualité de la pratique des étudiant·es.

NestedText – Un meilleur JSON

NestedText se veut *human-friendly*, similaire au JSON, mais pensé pour être facile à modifier et à visualiser par les humains. Le seul type de donnée scalaire supporté est la chaîne de caractères, afin de simplifier la syntaxe et retirer les guillemets (31). En plus des types de données restreints, l'autre différence avec le YAML est la facilité d'intégrer des morceaux de code sans échappements ni guillemets, les caractères de données ne peuvent pas être confondus avec NestedText (32).

```

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 11 – Exemple tiré de leur README (31)

Les tabulations restent nécessaires pour définir la hiérarchie. Tout comme le JSON la validation du contenu n'est géré que par des librairies externes qui vérifient la validité à l'aide d'un schéma (33). De plus, l'implémentation officielle est en Python et il n'en existe pas pour le Rust. Il existe une crate réservée qui est restée vide (34).

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 (35)). SDLang définit également différents types de nombres (32 bits, 64 bits, entiers, flottants...), quatre valeurs de booléens (`true`, `false`, `on`, `off`), 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 12](#) – Exemple de SDLang tiré de leur site web (35)

Cet exemple en Snippet 12 est intéressant par le faible nombre de caractères réservés et la densité d'information. Il s'approche de ce qui avait été imaginé sur la syntaxe DY, dans l'introduction en Figure 8. En YAML, trois lignes auraient été nécessaires à définir l'auteur avec son nom, email et un attribut booléen. En SDLang une seule ligne suffit: `author "Peter Parker" email="peter@example.org" active=true`. Les neuf valeurs de la matrice sont définies sur seulement cinq lignes, avec l'espace comme séparateur.

Il est regrettable que les strings doivent être entourées de guillemets. Le texte brut sur plusieurs lignes (comme du code) doit être entouré de backticks ` (36). De même, la définition de la hiérarchie d'objets nécessite d'utiliser une paire d'accolades.

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 13](#) – Exemple de KDL simplifié tiré de leur site web (37)

Si l'exemple en Snippet 13 paraît proche de SDLang, c'est normal puisque KDL est un fork de SDLang. Les améliorations qui nous intéressent concernent la possibilité de retirer des guillemets pour les strings sans espace (`person name=Samuel` au lieu de `person name="Samuel"`). Cette simplification n'inclut malheureusement pas le texte multiligne, qui demande d'être entourée par `"""`. Le problème d'intégration de morceaux de code entre ``` pour certains langages qui utilisent ce symbole (comme Bash), a été relevé par l'auteur du fork dans la FAQ. Le texte brut est ainsi supporté avec un `#` ajouté autour des guillemets, par exemple `regex #"\d{3} "[^/"]+###` ou dans la Snippet 13 avec le noeud `build`. Une répétition des `#` permet d'inclure ce caractère littéral pour éviter tout besoin d'échappement. Par exemple `msg ##"hello#"john##` contient un noeud `msg` avec la valeur `hello#"john` (37).

En dehors des autres désavantages déjà évoqués pour SDLang, il reste toujours le problème des types de nombres qui peuvent créer des ambiguïtés avec le texte. Ce noeud `version 1.2.3` est interprété comme nombre à virgule avec une erreur de format, il a besoin de guillemets `version "1.2.3"` pour indiquer que ce n'est pas un nombre.

Conclusion

En conclusion, au vu du nombre de tentatives/variantes existantes, qui va au-delà de ce qui a été documenté dans ce rapport, on voit que la verbosité des formats largement répandus du XML, JSON et même du YAML est un problème identifié par plusieurs personnes. Même Apple a fait son propre format de configuration, le Pkl qui mixe des constructions de programmation et de données pour la validation des fichiers (38).

La diminution de la verbosité des syntaxes décrites précédemment est intéressante: on évite des guillemets, des accolades, et d'autres séparateurs ce qui facilite la rédaction. Le contenu multiligne devient plus facile à intégrer en évitant d'échapper des caractères particuliers. Le problème reste que pour supporter des structures de données avancées, elles sont toujours obligées d'avoir un minimum de séparateurs. Souvent ne proposent pas de validation intégrée.

Ces améliorations sont déjà un gain important, mais il est possible d'aller encore plus loin, en sacrifiant une partie des structures de données. Notre syntaxe DY propose une approche encore plus légère. En se limitant à un ensemble de clés autorisées avec une hiérarchie définie, nous pouvons exprimer la hiérarchie du document sans nécessiter de tabulations ni d'accolades et valider le document directement durant le parsing.

Librairies de parsing en Rust

Après s'être intéressé aux syntaxes existantes, nous nous intéressons maintenant aux solutions pour simplifier le développement du parseur DY. Après quelques recherches avec le tag `parser` sur crates.io (39), j'ai trouvé la liste de librairies suivantes :

- `nom` (40), utilisé notamment par `cexpr` (41)
- `winnow` (42), fork de `nom`, utilisé notamment par le parseur Rust de KDL (43)
- `pest` (44)
- `combine` (45)
- `chumsky` (46)

À noter aussi l'existence de la crate `serde` (47), un framework de sérialisation et désérialisation très populaire dans l'écosystème Rust (selon le site lib.rs (48)). Il est notamment utilisé pour les parseurs JSON `serde_json` et TOML `toml`. Ce n'est pas une librairie de parsing mais un modèle de donnée basée sur des traits Rust (des interfaces) pour faciliter le passage d'un arbre syntaxique abstrait (AST) aux structures de données Rust. Le modèle de données de Serde (49) supporte 29 types de données. Trois raisons nous poussent à ne pas construire un parseur compatible avec `serde` :

1. Seulement les strings, listes et structs sont utiles pour PLX. Par exemple, les 12 types de nombres sont inutiles à différencier.
2. La sérialisation (structure Rust vers syntaxe DY) n'est pas prévue, seul la désérialisation nous intéresse
3. L'association des clés et les propriétés vers les attributs des structs Rust n'est pas du 1:1. La valeur après `exo` contient le nom de l'exercice puis la consigne, ce qui signifie une seule string pour deux champs `name` et `instruction` dans la structure `Exo` finale.

Parser des simples expressions de math comme `((23+4) * 5)` est idéal pour ces outils: les débuts et fin de chaque partie sont claires, une combinaison de fonctions de parsing permettraient facilement d'identifier les expressions parenthésées, les opérateurs puis les nombres. Elles semblent bien adaptées à ignorer des espaces, extraire les nombres tant qu'ils contiennent des chiffres, extraire des opérateurs et les deux opérandes autour...

Pour DY, l'aspect multiligne et le fait qu'une partie des clés est optionnelle, rend compliquée l'approche de combinaisons de parseurs.

```
exo Dog struct
Consigne très longue

en *Markdown*
sur plusieurs lignes

check la struct a la bonne taille
see sizeof(Dog) = 12
exit 0
```

Figure 12 – Exemple d'exercice PLX en DY, avec une consigne en Markdown sur plusieurs lignes

Le Figure 12 nous montre une consigne qui démarre après la ligne `exo` et continue sur plusieurs lignes jusqu'à qu'on trouve une autre clé (ici `check`). Le problème se pose aussi avec la clé `see`, qui est aussi multiligne, dont la valeur s'arrête au prochain `see`, `type`, `exit` ou `check`.

La syntaxe DY est relativement simple à parser et sa nature implicite rend compliqué l'usage de librairies pensées pour des formats avec beaucoup de séparateurs. Après ces recherches et quelques essais avec `winnow`, nous avons décidé que l'écriture manuelle du parseur sans librairie serait plus simple.

Les serveurs de langage

Par défaut, avec un nouveau langage, il faut manuellement lancer le compilateur ou le parseur sur son fichier, voir les erreurs et de revenir dans l'éditeur pour les corriger. Certaines opérations répétitives, comme renommer une fonction à chaque appel, doivent être faites à la main. Pour ces raisons, il devient très intéressant d'intégrer un nouveau langage aux différents IDE utilisés dans le monde, mais cela posent de nombreux challenges.

Le support d'un éditeur consiste à intégrer les erreurs du parseur, l'autocomplétion, les propositions de corrections, des informations au survol... et de nombreuses petites fonctionnalités qui améliorent l'expérience de rédaction. L'avantage d'avoir les erreurs de compilation directement soulignées dans l'éditeur, c'est de pouvoir voir les erreurs dans leur contexte et de corriger immédiatement les problèmes. Supporter chaque éditeur de code indépendamment signifie travailler avec des API légèrement différentes pour supporter la même fonctionnalité et utiliser plusieurs langages de programmation différents.

Les développeur·euses de nouveaux IDE font face à un défi similaire, mais encore plus difficile, celui de supporter des centaines de langages pour qu'un maximum de monde puisse développer avec. Microsoft était face au même problème pour son éditeur VSCode et a inventé un protocole, nommé `Language Server Protocol (LSP)` (50). Ce protocole définit un pont commun entre un client LSP implémenté à l'interne de chaque IDE et un serveur LSP, appelé serveur de langage (*language server*). L'IDE peut ainsi demander de manière unique des informations, tel que *Donne moi les résultats d'autocomplétion pour le curseur à cette position* sans devoir supporter des détails du langage édité. Le projet a pour but de simplifier la vie des développeur·euses de nouveaux langages et des nouveaux éditeurs qui peuvent intégrer rapidement des centaines de langages en implémentant « juste » un client LSP.

Les serveurs de langages tournent dans des processus séparés de l'éditeur, ce qui permet de ne pas imposer de langage de programmation. Le client LSP se charge de lancer le processus du serveur, de lancer des requêtes et d'intégrer les données des réponses dans leur interface visuelle. Les serveurs de langage n'ont aucune idée de l'éditeur qui leur demande des informations et ils n'en ont pas besoin puisque le protocole définit les réponses attendues en retour.

The screenshot shows a Neovim editor window with the following code:

```

1 fn main() {
2     let name: &str = "    John    ".trim();
fn(&self) -> &str

```

A tooltip provides documentation for the `trim` method:

Returns a string slice with leading and trailing whitespace removed.
 'Whitespace' is defined according to the terms of the Unicode Derived Core Property `White_Space`, which includes newlines.

The cursor is at the end of the word `trim`. A completion dropdown is open, listing various `trim` methods:

- f trim()
- f trim_matches(...)
- f trim_right_matches(...)
- f trim_ascii()
- f trim_ascii_start()
- f trim_start_matches(...)
- f trim_ascii_end()
- f trim_end_matches(...)
- f trim_start()
- f trim_end()

At the bottom of the editor, there is a section titled `# Examples` containing:

```

let s = "\n Hello\tworld\t\n";

```

Figure 13 – Exemple d'autocomplétion dans Neovim, générée par le serveur de langage `rust-analyzer` sur l'appel d'une méthode sur les `&str`

Un serveur de langage n'a pas besoin d'implémenter toutes les fonctionnalités du protocole. Un système de capacités (*Capabilities*) est défini pour annoncer les méthodes implémentées (51). Nous pourrons ainsi implémenter que la petite partie du protocole qui nous intéresse.

Le protocole **JSON-RPC** (JSON Remote Procedure Call) est utilisé comme protocole de communication. 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 optionnellement `params` (52). L'`id` sert à associer une réponse à une requête. Il est aussi possible d'envoyer une notification, c'est-à-dire une requête qui n'attend pas de réponse. Le champ `method` va indiquer l'action à appeler. Le transport des messages JSON-RPC peut se faire en `stdio` (flux standards d'entrée/sortie), sockets TCP ou même en HTTP.

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

, caption: [Exemple de requête JSON-RPC du client pour demander des propositions d'autocomplétion (`textDocument/completion`). Tiré de la spécification (53)],)

Quelques exemples de serveurs de langages implémentés en Rust

- `rust-analyzer`, serveur de langage officiel du langage Rust
- `tinymist`, serveur de langage de Typst (système d'édition de document, concurrent du Latex, utilisé pour la rédaction de ce rapport)
- `asm-lsp` (54), 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 langage pour Java implémenté en Java (55)
- `tailwindcss-language-server`, le serveur de langage pour le framework TailwindCSS, implémenté en TypeScript (56)
- `typescript-language-server` pour TypeScript, implémenté en TypeScript également (57)
- et beaucoup d'autres...

Adoption

Selon la liste sur le site de la spécification (58), 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 (59) quant à elle, contient plus de 200 projets, dont 40 implémentés en Rust! Ce large support et ces nombreux exemples faciliteront le développement de ce serveur de langage et son intégration dans différents IDE.

Librairies disponibles

Pour ne pas devoir réimplémenter la mise en place d'un serveur, il existe plusieurs crates qui prennent en charge une partie des parties du protocole commune à tous les langages, comme l'initialisation de la communication.

En cherchant à nouveau sur `crates.io` sur le tag `lsp`, on trouve différents projets dont `async-lsp` (60) utilisée par la même auteure dans `nix` (61) (un serveur de langage pour le système de configuration de NixOS).

Le projet `tinymist` a extrait une crate `sync-ls`, mais le README déconseille son usage et conseille `async-lsp` à la place (62). En continuant la recherche, on trouve encore une autre crate `tower-lsp` et un fork `tower-lsp-server` (63)... `rust-analyzer` a également extrait une crate `lsp-server`. Une crate commune à plusieurs projets est `lsp-types` (64) qui définit les structures de données, comme `Diagnostic`, `Position`, `Range`. Ce projet est utilisé par `lsp-server`, `tower-lsp` et d'autres (65).

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.

Le choix de `lsp-types` est facilement choisi mais les quantités des autres crates ne rend pas le choix immédiat. Les 2 projets les plus utilisés (en termes de *reverse dependencies* sur crates.io) sont `lsp-server` (56 projets) (66) et `tower-lsp` (85 projets) (67). 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. L'autre argument est que le projet `tower-lsp` est basée sur des abstractions asynchrones, nous préférions partir sur la version synchrone pour simplifier l'implémentation.

Cette partie est un *nice-to-have*, nous espérons avoir le temps de l'intégrer dans ce travail. Après quelques heures sur le POC suivant, cela semble être assez facile et rapide.

POC de serveur de language avec `lsp-server`

La crate `lsp-server` contient un exemple de `goto_def.rs` (68) qui implémente la possibilité de `Aller à la définition` (*Go to definition*), généralement accessible dans l'IDE par un `Ctrl+clic` sur une fonction. Nous avons modifié et exécuté cet exemple puis créé un petit script `demo.fish` qui simule un client et affiche chaque requête. Le client va simplement demander la définition d'une position dans `/tmp/another.rs` et le serveur va lui renvoyer une autre position.

Le code du serveur qui gère la requête du type `GotoDefinition` est visible en Snippet 14. Le serveur répond avec un `GotoDefinitionResponse` qui contient une position dans le code (de type `Location`) sur le fichier `/tmp/another.rs` à la ligne 3 entre les caractères 12 et 25 (la plage est décrite avec le type `Range`).

```
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 14 – Extrait de `goto_def.rs` modifié qui retourne un emplacement `Location` dans une réponse `GotoDefinitionResponse`

Maintenant que nous avons un petit serveur fonctionnel, nous pouvons lancer notre client. Sur la Figure 14 les lignes après `CLIENT:` sont envoyées en `stdin` et celles après `SERVER:` sont reçues en `stdout`.

1. Durant l'initialisation, le serveur nous indique qu'il supporte un *fournisseur de définition* avec `definitionProvider` à `true`.
2. Le client envoie ensuite une requête `textDocument/definition`, pour le symbole dans un fichier `/tmp/test.rs` sur la ligne 7 au caractère 23.
3. Le serveur lui répond comme attendu.
4. Puis, le client a terminé et demande au serveur de s'arrêter.

```

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": {"character": 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}

```

Figure 14 – Output du script `demo.fish` avec les détails de la communication entre un client et notre serveur LSP

Surlignage du code

Par défaut un nouveau langage avec une extension de fichier dédiée reste en noir en blanc dans l'IDE. Pour faciliter la lecture, nous souhaitons pouvoir coloriser la majorité du contenu de notre syntaxe, tout en groupant les couleurs par type d'éléments surlignés. Pour notre syntaxe DY, on aimerait que toutes les clés aient la même couleur, tout comme les propriétés qui doivent être toutes colorisées d'une seconde couleur. Les commentaires doivent être grisés.

Le bout de C `printf("salut");` est vu par un système de surlignage de code comme une suite de morceaux, des tokens d'une certaine catégorie. Ce bout de code pourrait être subdivisé avec les tokens suivants `printf` (identifiant), `(` (séparateur), `"` (séparateur), `salut` (valeur littérale), `,` , `)` et `;` (séparateur).

Les IDE modernes intègrent des systèmes de surlignage de code (*code highlighting*) et définissent leur propre liste de catégories de tokens, par exemple: séparateur, opérateur, mot clé, variable, fonction, constante, macro, énumération, ... Une fois la catégorie attribuée, il reste encore à définir quel couleur concrète est utilisée pour chaque catégorie. C'est le rôle des thèmes comme Monokai, Darcula, Tokioynight et beaucoup d'autres. Les systèmes de surlignage supportent parfois un rendu web via une version HTML contenant des classes CSS spécifiques à chaque type de token. Des thèmes écrits en CSS peuvent ainsi appliquer leurs couleurs. Le surlignage peut être de type

yntaxique (*syntax highlighting*), avec une analyse purement basée sur la présence et l'ordre des tokens, ou sémantique (*semantic highlighting*) après une analyse du sens du token.

Textmate - surlignage syntaxique

TextMate est un IDE pour macOS qui a introduit un concept de grammaires. Ces grammaires permettent de définir la manière dont le code doit être séparé en tokens, à l'aide d'expressions régulières issues de la bibliothèque C Oniguruma (55) (69). VSCode s'appuie sur ces grammaires TextMate (70), tout comme IntelliJ IDEA, qui les utilise pour le Swift, C++ ou Perl qui ne sont pas supportés nativement (71).

Le Snippet 15 montre un exemple de grammaire Textmate décrivant un langage nommé `untitled` avec quatre mots clés (`if`, `while`, `for`, `return`) et des chaînes de caractères entre guillemets. Les expressions régulières données en `match`, `begin` et `end` permettent de trouver les tokens dans le document et leur attribuer une catégorie (comme un mot clé avec `keyword.control.untitled`).

```
{
  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 = '\\\\.';
        }
      );
    },
  );
}
```

Snippet 15 – Exemple de grammaire Textmate tiré de leur documentation (72).

Tree-Sitter - surlignage syntaxique

Tree-Sitter (73) se définit comme un « *outil de génération de parser et une librairie de parsing incrémentale. Il peut construire un arbre de syntaxe concret (CST) depuis un fichier source et mettre à jour cet arbre efficacement, quand le fichier source est modifié.* » (73) (Traduction personnelle). Tree-Sitter permet aux éditeurs de fournir plusieurs fonctionnalités, dont le surlignage syntaxique.

Tree-Sitter est supporté dans Neovim (74), dans le nouvel éditeur Zed (75), ainsi que d'autres. Tree-Sitter a été inventé par l'équipe derrière Atom (76) 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) (77).

The screenshot shows a GitHub repository interface with a code editor and a sidebar. The code editor displays a Rust file named `exo.rs` with 400 lines and 377 loc. The sidebar is titled "Symbols" and lists various symbols found in the code, such as `ExoInfo`, `ExoStateInfo`, `Exo`, `Exo`, `from_dir`, `find_exo_and_solution_files`, `check_exo_solutions`, `get_main_file`, `compiler`, `test`, `test_parse_full_intro_basi...`, `test_exo_done`, and `test_exo_favorite`. The symbols are categorized by type (struct, func, impl, mod) and name.

```

main plx / src / models / exo.rs
ode Blame 400 lines (377 loc) · 14.1 KB
96 impl Exo {
125     // Check every solution file and check that it corresponds to an actual exo file
126     fn check_exo_solutions(
127         exo_files: &Vec<std::path::PathBuf>,
128         solution_files: &Vec<std::path::PathBuf>,
129         warnings: &mut Vec<ParseWarning>,
130     ) {
131         if solution_files.is_empty() {
132             return;
133         }
134         for solution_file in solution_files {
135             //try to get solution file name and solution last extension
136             match (solution_file.file_stem(), solution_file.extension()) {
137                 (Some(file_name), Some(extension)) => {
138                     //try to parse file name to string
139                     match (file_name.to_str(), extension.to_str()) {
140                         (Some(file_name), Some(extension)) => {
141                             // associated exo file should be of format <file_name>.<extension>
142                             // This essentially removes the .sol part
143                             let exo_target_name =
144                                 format!("{}.{})", file_name.replace(".sol", ""), extension);
145                             let exo_exists = exo_files
146                             .iter()
147                             .find(|exo_file| exo_file == exo_target_name)
148                         }
149                     }
150                 }
151             }
152         }
153     }
154 }

```

Figure 15 – Liste de symboles sur un exemple de Rust sur GitHub, générée par Tree-Sitter

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 compilée puis utilisée via le CLI `tree-sitter` durant le développement. Pour la production, comme elle n'a pas de dépendance externe, elle pourra être intégrée ou chargée dynamiquement (73, 78). Pour permettre du surlignage syntaxique, il reste encore à définir des fichiers de requêtes qui sélectionnent des noeuds dans l'arbre généré et attribue des catégories à ces tokens, qui sont compatible avec l'IDE.

Surlignage sémantique

Les deux solutions de surlignage syntaxique présentées précédemment sont déjà satisfaisantes, mais ne tiennent pas compte de nombreuses informations sémantiques qui permettraient d'améliorer encore la colorisation.

Sur le Snippet 16 surligné avec Tree-Sitter (surlignage syntaxique pour rappel), on voit que les appels de `HEY` et `hi` dans le `main` ont les mêmes couleurs alors que l'un est une macro, l'autre une fonction. À la définition, les couleurs sont bien différentes, parce que le `#define` permet de différencier la macro d'une fonction. À l'appel, il n'est pas possible de les différencier avec une analyse syntaxique, car les tokens extraits seront les mêmes (identifiant, parenthèse, chaîne littérale, etc). La notation en majuscules de l'identifiant de la macro ne peuvent pas être utilisés pour différencier l'appel, car ce n'est qu'une convention, ce n'est pas requis par le 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;
}

```

Snippet 16 – Exemple de code C `hello.c`, avec macro et fonction surligné de la même manière à l'appel dans le `main`

Sur la Figure 16, l'arbre syntaxique concret généré par Tree-Sitter nous montre que les appels de `hi` et `HEY` sont catégorisés comme des fonctions.

```
(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]
```

Figure 16 – CST généré par `tree-sitter parse hello.c`

Pour différencier les appels, le serveur de langage possèdent plus de contexte sémantique et peut nous aider à améliorer la colorisation. Le surlignage sémantique est une extension du surlignage syntaxique, où un serveur de langage fournit des tokens sémantiques. Un serveur de langage est plus lent et ne fournit une information supplémentaire que pour une partie des tokens, il n'a pas pour but de remplacer le surlignage syntaxique. (79)

Si on inspecte l'état de Neovim avec le fichier du Snippet 16 ouvert, le serveur de langage `clangd` a réussi à préciser la notion de macro au-delà du 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
```

Snippet 17 – Extrait de la commande `:Inspect` dans Neovim avec le curseur sur le `HEY`

```
1 #include <stdio.h>
2 |
3 const char *HELLO = "Hey";
4 #define HEY(name) printf("%s %s\n", HELLO, name)
5 void hi(char *name) { printf(format: "%s %s\n", HELLO, name); }
6 |
7 int main(int argc, char *argv[]) {
8   |   hi(name: "Samuel");
9   |   HEY("Samuel");
10  |   return 0;
```

Figure 17 – Une fois `clangd` lancé, l'appel de `HEY` prend une couleur différente que l'appel de fonction mais la même couleur que celle attribuée sur sa définition

En voyant la liste des tokens sémantiques définis dans la spécification LSP (80), cela peut aider à comprendre l'intérêt et les possibilités d'un surlignage avancé. Par exemple, on trouve des tokens sémantiques comme `macro`, `regexp`, `typeParameter`, `interface`, `enum`, `enumMember`, qui seraient difficiles de détecter au niveau syntaxique.

Choix final

L'auteur a ignoré l'option du système de SublimeText, pour la simple raison qu'il n'est supporté nativement uniquement par SublimeText, probablement parce que cet IDE est propriétaire (81). Ce système utilise des fichiers `.sublime-syntax`, qui ressemblent à TextMate (82), mais qui sont rédigés en YAML.

Si le temps le permet, une grammaire sera développée avec Tree-Sitter pour supporter du surlignage dans Neovim. Le choix de ne pas explorer plus les grammaires Textmate, laisse penser que nous délaissions complètement VSCode. Ce choix peut paraître étonnant comme VSCode est régulièrement utilisé par 73% des 65,437 répondant·es au sondage de StackOverflow 2024 (83).

Cette décision se justifie notamment par la roadmap de VSCode: entre mars et mai 2025 (84, 85), des employé·es de Microsoft ont commencé un travail d'investigation autour de Tree-Sitter pour explorer les grammaires existantes et l'usage de surlignage dans VSCode (86). Des premiers efforts d'exploration avaient d'ailleurs déjà eu lieu en septembre 2022 (87).

La version de VSCode de mars 2025 (1.99) supporte de manière expérimentale le surlignage avec Tree-Sitter des fichiers CSS et des expressions régulières dans les fichiers TypeScript. (88)

L'usage du surlignage sémantique n'est pas au programme de ce travail, mais pourra être exploré dans le futur si certains éléments sémantiques pourraient en bénéficier.

POC de surlignage de notre syntaxe avec Tree-Sitter

Ce POC vise à prouver que l'usage de Tree-Sitter fonctionne pour coloriser les clés et les propriétés. L'exemple du Snippet 18 est un exercice de choix multiples. Ce format n'est pas supporté par PLX, mais cela nous permet de coloriser un exemple minimaliste incluant des clés `exo` (titre) et `opt` (options) avec en plus des propriétés `.ok` et `.multiple`.

```
// 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 18 – Affichage noir sur blanc ce qui rend la lecture difficile
avec une question à choix multiples dans un fichier `mcq.dy`

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 règles construites via des fonctions fournies par Tree-Sitter et des expressions régulières. La documentation **The Grammar DSL** de la documentation explique toutes les options possibles en détails (89). Pour avoir un aperçu, la fonction `seq` qui indique une liste de tokens qui viendront en séquence et `choice` permet de tester plusieurs options à la même position. On remarque également les clés et propriétés insérés dans les tokens de `key` et `property`.

```

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

```

Snippet 19 – Résultat de la grammaire minimaliste `grammar.js`, définissant un ensemble de règles sous `rules`.

On observe dans le Snippet 19 plusieurs règles:

- `source_file` : décrit le point d'entrée d'un fichier source, défini comme une répétition de ligne.
- `_line` : une ligne est une séquence d'un choix entre 4 types de lignes, chacune décrites en dessous, puis un retour à la ligne
- `line_withkey` : une ligne avec une clé consiste en une séquence de token composé d'une clé, ensuite de zéro à plusieurs propriétés. Elle se termine optionnellement par un contenu qui commence après un premier espace
- `commented_line` définit les commentaires comme `//`, puis un reste
- `list_line`, `dash` et le reste des règles suivent la même logique

Après avoir appelé `tree-sitter generate` pour générer le code du parseur C et `tree-sitter build` pour le compiler, on peut maintenant parser un fichier donné et afficher le CST (Concrete Syntax Tree). Dans cet arbre qui démarre avec un noeud racine `source_file`, on retrouve des noeuds du même type que les règles définies précédemment, avec le texte extrait dans la plage de caractères 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 propriété `.ok` et le tiret de la règle `dash`.

```

dy> tree-sitter parse -c mcq.dy
0:0 - 7:0    source_file
0:0 - 0:19   commented_line `// Exercice basique`"
0:19 - 1:0   "\n"
1:0 - 1:16  line_withkey
1:0 - 1:3   key `exo`
1:3 - 1:4   "
1:4 - 1:16  content `Introduction`"
1:16 - 3:0   "\n"
3:0 - 3:13  line_withkey
3:0 - 3:3   key `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"

```

Figure 18 – CST généré par la grammaire définie sur le fichier `mcq.dy`

Pour voir notre exercice en couleurs, il nous reste deux éléments à définir. Le premier 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 ressemblent à `@variable`, `@constant`, `@function`, `@keyword`, `@string` ... ou des versions plus spécifiques comme `@string.regexp`, `@string.special.path`. Ces noms sont ensuite utilisés par les thèmes pour appliquer un style.

```

(key) @keyword
(commented_line) @comment
(content) @string
(property) @property
(dash) @operator

```

Snippet 20 – Aperçu du fichier `queries/highlights.scm`

Le deuxième élément est la couleur exacte pour chaque nom de surlignage. Le CLI `tree-sitter` supporte directement la configuration d'un thème via son fichier de configuration, un exemple est visible en Snippet 21.

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

Snippet 21 – Contenu du fichier de configuration de Tree-Sitter présent sur Linux au chemin `~/.config/tree-sitter/config.json`

```
// 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
```

Figure 19 – Screenshot du résultat de la commande
`tree-sitter highlight mcq.dy` avec notre exercice surligné

La réalisation du POC s'inspire de l'article **How to write a tree-sitter grammar in an afternoon** (90).

Le résultat de ce POC est encourageant, même s'il faudra probablement plus de temps 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.

Le surlignage sémantique pourrait être utile en attendant l'intégration de Tree-Sitter dans VSCode. L'extension `tree-sitter-vscode` en fait déjà une intégration avec cette approche, qui s'avère beaucoup plus lente qu'une intégration native, mais permettrait d'avoir une solution fonctionnelle temporaire. À noter que l'extension n'est pas triviale à installer et ni à configurer, son usage est encore expérimental. Elle nécessite d'avoir un build WebAssembly de notre parseur Tree-Sitter (91).

```
≡ mcq.dy
1 // Basic MCQ exo
2 exo Introduction
3
4 opt .multiple
5 - C is an interpreted language
6 - .ok C is a compiled language
7 - C is mostly used for web applications
8
9
```

Figure 20 – Screenshot dans VSCode une fois l'extension `tree-sitter-vscode` configurée pour notre grammaire Tree-Sitter

Protocoles de communication bidirectionnels et formats de sérialisation

Le serveur de sessions live a besoin d'un système de communication en temps réel pour être capable transmettre le code et les résultats des étudiants. Ces messages seront transformés dans un format standard, facile à sérialiser et désérialiser en Rust. Cette section explore les formats textuels et binaires disponibles, ainsi que les protocoles disponibles qui fonctionnent sur TCP.

JSON

Contrairement à toutes les critiques relevées précédemment sur le JSON et d'autres formats pour leur rédaction manuelle, le JSON est une option solide pour la communication client-serveurs. Il est très populaire pour les API REST, les fichiers de configuration et s'intègre bien en Rust.

En Rust, avec `serde_json`, il est simple de parser du JSON dans une struct. Une fois la macro `Deserialize` appliquée, on peut directement appeler `serde_json::from_str(json_data)`.

```
#[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();
```

[Snippet 22](#) – Exemple simplifié de parsing de JSON, tiré de leur documentation (92).

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

[Snippet 23](#) – Exemple de sérialisation en JSON d'une structure arbitraire, tiré de leur documentation (93).

WebSocket

Le protocole WebSocket, défini dans la RFC 6455, permet une communication bidirectionnelle 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 deux sens. On évite ainsi d'ouvrir plusieurs connexions HTTP à chaque requête. La technologie a été pensée pour être utilisée par des applications dans les navigateurs, mais fonctionne également en dehors (94).

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 longueur variable) et de distinguer le texte des données binaires. (95) Le protocole supporte ainsi d'envoyer un type de message pour le texte qui doit être de l'UTF8 et un autre pour le binaire (94).

Dans l'écosystème Rust, il existe plusieurs crates qui implémentent le protocole, parfois côté client, côté serveur ou les deux. Il existe plusieurs approches synchrone et asynchrone (`async/await`).

La crate `tungstenite` propose une abstraction du protocole qui permet de facilement interagir avec des `Message`, leur écriture `send()` et leur lecture `read()` est simple à utiliser (96). Elle passe la

Autobahn Test Suite (suite de tests de plus de 500 cas pour vérifier une implémentation Websocket) (97).

Comme nous avons besoin de gérer des milliers de clients simultanés, les threads natifs ne sont pas adaptés et nous avons besoin d'un système asynchrone. L'approche asynchrone requiert un runtime, qui ressemble à l'ordonnanceur d'un OS, mais qui gère des tâches asynchrones en dehors de l'espace noyau, pour que ces threads virtuels puissent laisser la place à d'autres dès qu'une interaction avec le réseau les fait attendre.

En Rust, `tokio` est la solution la plus populaire à ce problème (48). Une version `async` autour de `tungstenite` existe pour le `runtime Tokio` et s'appelle `tokio-tungstenite` (98). Nous avons aussi besoin de pouvoir écrire et lire sur un socket en même temps (pour attendre des messages tout en continuant d'en envoyer) et `tokio-tungstenite` supporte ce besoin.

Il existe une crate `websocket` avec une approche sync et `async`, qui est dépréciée et dont le README (99) conseille l'usage de `tungstenite` ou `tokio-tungstenite` à la place (99). Pour conclure cette section, il est intéressant de relever qu'il existe d'autres crates tel que `fastwebsockets` (100) à disposition, qui semblent demander de travailler à un plus bas niveau.

Formats binaires

Protocol Buffers, dit Protobuf, est un format binaire développé par Google pour sérialiser des données structurées, de manière compacte, rapide et simple. L'idée est de définir un schéma dans un style qui ne dépend pas d'un langage de programmation. À partir de ce schéma du code de sérialisation est généré pour interagir avec ces structures depuis du C++, Java, Go, Ruby, C# et d'autres. (101)

```
edition = "2023";

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

Snippet 24 – Un exemple de description d'une personne en ProtoBuf, de leur site web (101).

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

Snippet 25 – Son usage en Java avec les classes autogénérées à la compilation, de leur site web (101).

Le langage Rust n'est pas officiellement supporté, mais un projet du nom de PROST! existe (102) et permet de générer du code Rust depuis des fichiers Protobuf.

gRPC est un protocole inventé par Google basé sur Protobuf. Ce système de Remote Procedure Call (RPC - un système d'appel de fonctions à distance) est universelle, performant et supporte le streaming bidirectionnel sur HTTP2. En plus des définitions des messages en Protobuf déjà présentés, il est possible de définir des services, avec des méthodes avec un type de message et un type de réponse. (103)

Un article de 2019, intitulé **The state of gRPC in the browser** (104) 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 entre gRPC-Web et gRPC.

De nombreux autres formats binaires existent: un framework RPC pour Rust nommé **tarpc** (105), MessagePack (106), Cap'n Proto (107) qui tente d'être plus rapide que Protobuf ou encore Apache Thrift (108).

Choix final

gRPC n'est pas une bonne option pour notre projet comme l'application desktop de PLX contient une partie web, les limites de gRPC-Web et des navigateurs risquent de ralentir ou compliquer le développement.

Par soucis de facilité d'implémentation et d'intégration, nous avons choisi de rester sur un format textuel et d'implémenter la sérialisation en JSON via la crate `serde_json`. Notre expérience existante du WebSocket, sa possibilité d'utiliser autant du texte que du binaire, l'usage possible en Rust et son support dans les navigateurs, en font une solution adaptée pour ce travail. Nous utiliserons la crate `tungstenite` et `tokio-tungstenite`.

Quand l'usage de PLX dépassera des dizaines/centaines d'étudiants connectés en même moment et que la latence sera trop forte ou que les coûts d'infrastructures deviendront trop élevés, les formats binaires plus légers seront une option à considérer. Au vu des nombreux choix, mesurer la taille des messages, le temps de sérialisation et la facilité d'intégration sera nécessaire pour faire un choix.

POC de synchronisation de messages JSON via WebSocket avec `tungstenite`

Pour vérifier la faisabilité technique d'envoyer des messages en temps réel en Rust via WebSocket, un petit POC a été développé dans le dossier `pocs/websockets-json`. Le code et les résultats des checks doivent être transmis depuis le client PLX des étudiant·es vers celui de l'enseignant·e, en passant par le serveur de session live. À cause de sa nature interactive, il n'est pas évident de retranscrire ce qui s'y passe quand on lance le POC dans trois shells 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. Notre POC contient un sous dossier `fake-exo` contenant l'exercice fictif à implémenter.

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

Snippet 26 – Code Rust de départ de l'exercice fictif à compléter par l'étudiant·e

Le protocole définit pour permettre cette synchronisation est découpé en 2 étapes. La première partie consiste en une mise en place de la connexion et l'annonce de son rôle. La deuxième partie consiste en la compilation régulière et l'envoi du résultat du check vers le serveur, qui ne fait que de transmettre au socket associé au `teacher`.

Annonce des clients

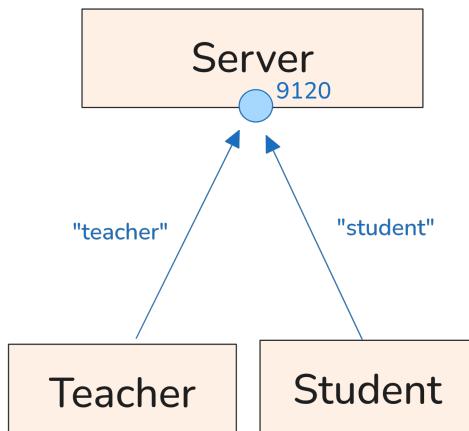
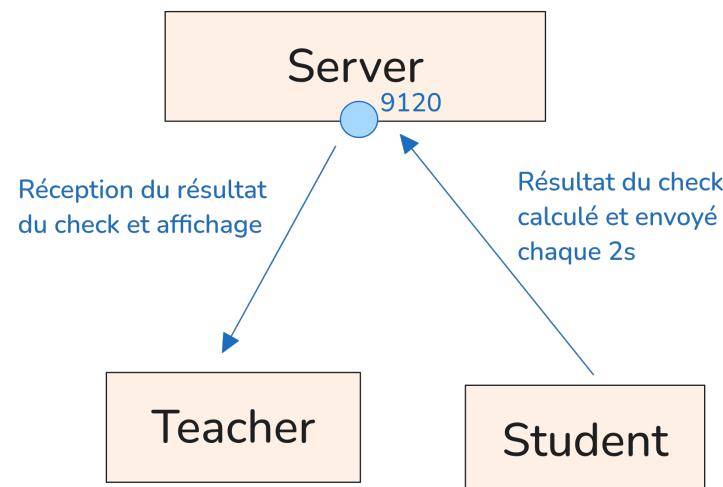


Figure 21 – Mise en place

Transfert des résultats des checks



Exo check on file main.rs, with given code:

```

fn main() {
    println!("Hello, world!");
}
Built but output is incorrect
Hello, world!
  
```

```
{"path":"main.rs","status": {"IncorrectOutput":{"stdout":"Hello, world!\n"}}, "code":"println()..."}
```

Figure 22 – Envoi du check vers le `server` qui transmet au `teacher`

Nous devons démarrer 3 shells pour lancer ce POC.

```

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

Snippet 27 – Shell 1 (S1)

```

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

Snippet 28 – Shell 2 (S2)

```

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

Snippet 29 – Shell 1

On lance le serveur sur le S1, qui attend des connexions sur le port 9120. On lance ensuite le `teacher` sur le shell 2 (S2), connexion au serveur et envoi d'un premier message avec le texte « `teacher` » pour annoncer son rôle. Le serveur a bien reçu la connexion d'un client et a détecté le rôle de `teacher`.

Dans le shell 3 (S3), on lance finalement le `student`.

```

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
  
```

Snippet 30 – Le `student` compile le code de `fake-exo`, exécute le check, puis envoie le résultat.

Le message envoyé, visible en Snippet 31 contient le chemin du fichier, un statut `IncorrectOutput` qui contient la sortie éronnée et le contenu du code.

```
{
  "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"
}
```

Snippet 31 – Le résultat éronné puisque l'exercice n'est pas encore implémenté.

Le serveur sur le S1 affiche `Forwarded one message to 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!");
}
Built but output is incorrect
Hello, world!
```

Snippet 32 – Le `teacher` a bien reçu le message et peut l'afficher

Si l'étudiant introduit une erreur de compilation, un message avec un statut différent est envoyé, incluant les erreurs de compilation comme contexte de l'échec.

```
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 33 – Le `teacher` a bien reçu le code actuel avec l'erreur de compilation de Cargo

Seul la crate `tungstenite` est utilisée pour ce POC, car `tokio-tungstenite` n'était pas certain d'être utilisé au départ. Cette deuxième crate exposant simplement une API autour, devrait être similaire au niveau de la facilité d'accès aux éléments du protocole.

Le système de synchronisation en temps réel fonctionne et permet d'envoyer différents messages du `student` au serveur qui le retransmet immédiatement au `teacher`. Même si cet exemple est minimal puisqu'il n'y a qu'un·seule étudiant·e et enseignant·e impliqué·e, nous avons démontré que la crate `tungstenite` fonctionne et peut être utilisée comme base de notre serveur de session live.

Développement du serveur de session live

Cette partie documente l'architecture et l'implémentation du serveur de session live, l'implémentation d'un client dans PLX et le protocole défini entre les deux.

La Figure 23 montre la vue d'ensemble des composants logiciels avec trois clients. Le serveur de session live est accessible par tous les clients. Les clients des étudiant·es transmettent et reçoivent d'autres informations que les clients des enseignant·es.

Tous les clients ont accès à tous les exercices, stockés dans des repository Git. Le parseur s'exécute sur les clients pour extraire les informations du cours, des compétences et des exercices. Le serveur n'a pas besoin de connaître les détails des exercices, il ne sert que de relai pour les participant·es d'une même session. Le serveur n'est utile que pour participer à des sessions live, PLX peut continuer d'être utilisé sans serveur pour l'entraînement seul·e.

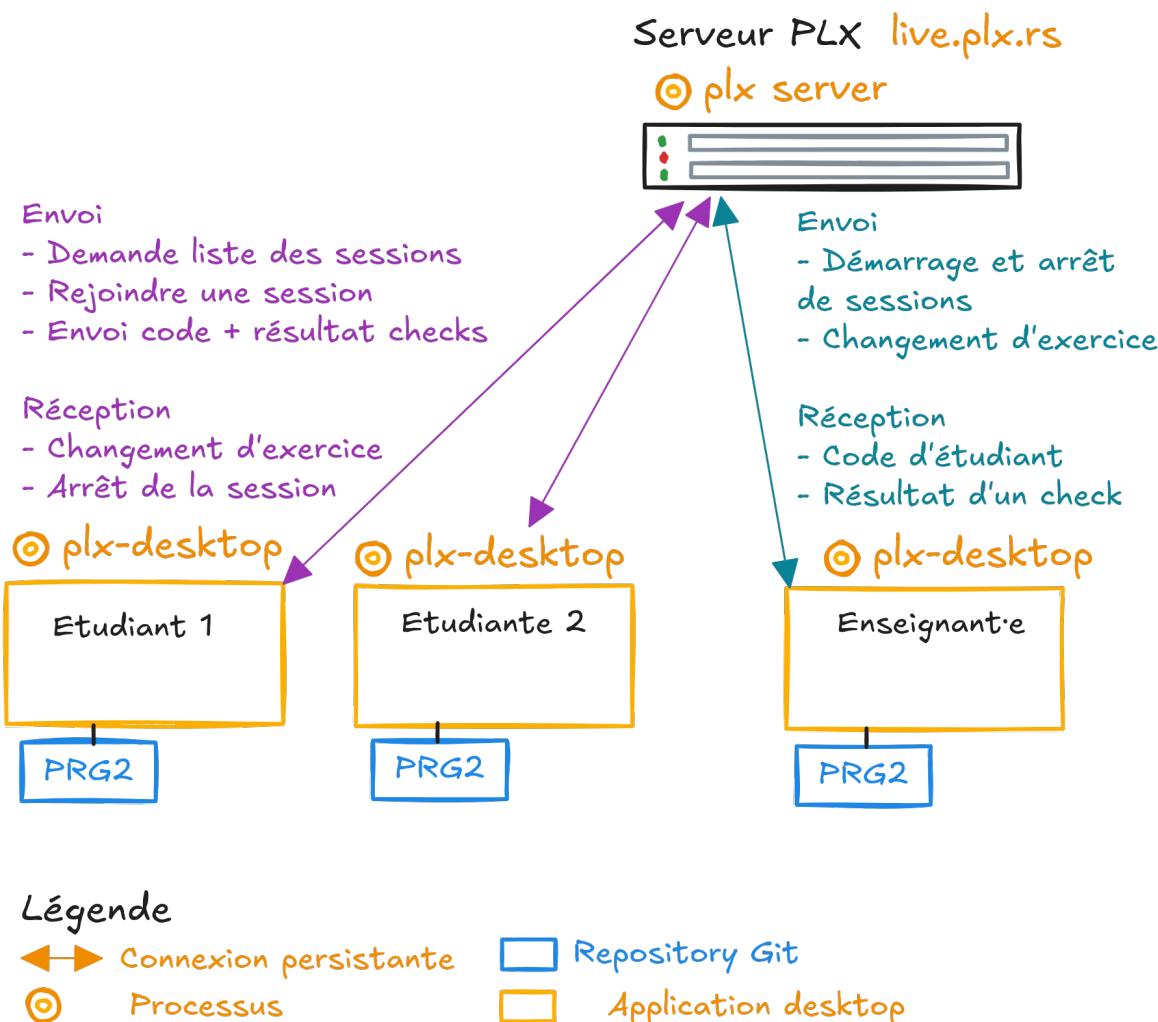


Figure 23 – Vue d'ensemble avec le serveur de session live, des clients, et notre parseur

Définition du protocole

Ce chapitre définit le protocole de communication qui régit les interactions entre les clients PLX et un serveur PLX.

Vue d'ensemble du protocole

Sur le plan technique, il fonctionne sur le protocole WebSocket pour permettre une communication bidirectionnelle. Trois parties composent notre protocole: la gestion de la connexion, la gestion des sessions et le transfert du code et résultats d'un exercice. La particularité du protocole est qu'il n'inclut pas d'authentification. Les clients sont néanmoins identifiés par un identifiant unique (`client_id`) permettant de reconnaître un client après une déconnexion temporaire.

Le protocole définit deux types de messages: les clients envoient des actions au serveur (message `Action`) et le serveur leur envoie des événements (message `Event`). La plupart des événements sont envoyés du serveur en réponse à des actions reçues.

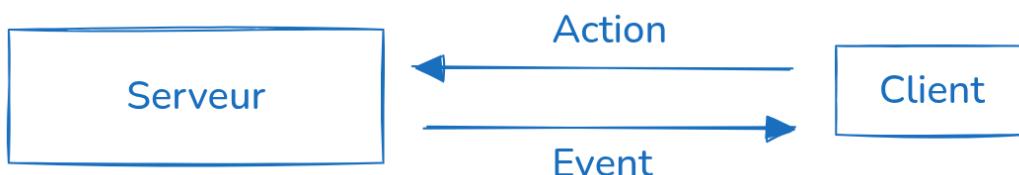


Figure 24 – Les deux types de messages ne sont envoyés que dans une direction

Ne pas avoir de système de compte implique que tous les clients sont égaux par défaut. Pour éviter que n'importe quel client puisse contrôler une session en cours et arrive à changer d'exercice ou arrêter la session, un système de rôle est défini. Nous aurions pu définir un rôle enseignant·e et étudiant·e, mais cela exclut d'autres contextes: lorsque des assistant·es sont présent·es ou qu'un groupe d'étudiant·es révisent ensemble, en dehors des cours. Nous avons besoin de définir deux rôles qui permettent de distinguer les clients qui gèrent une session et les autres qui y participent. Nous choisissons ainsi de les nommer respectivement **leader** et **follower**. La personne qui crée la session devient leader et les autres qui rejoignent deviennent followers. Le serveur peut ainsi vérifier à chaque `Action` reçue que le rôle du client autorise l'action.

Ce rôle est attribué à chaque client dans une session, avoir un rôle en dehors d'une session ne fait pas de sens. Les clients followers suivent les exercices lancés par les clients leaders et envoient le code et les résultats des checks à chaque changement. Les clients leaders ne participent pas aux exercices, mais le serveur leur transfère chaque modification envoyée par les clients followers. Le protocole n'empêche pas d'avoir plusieurs leaders par session, pour permettre certains contextes avec plusieurs enseignant·es ou assistant·es présent·es, pour aider à relire tous les morceaux de code envoyés.

Un système de gestion des pannes du serveur et des clients est défini, pour éviter d'engendrer de la confusion dans l'expérience finale. Les instabilités de Wifi, la batterie vide ou un éventuel crash de l'application ne devrait pas impacter le reste des participant·es de la session. Quand le serveur s'éteint, en cas de panne ou de mise à jour, les clients doivent pouvoir afficher une alerte dans leur interface. Un·e enseignant·e qui se déconnecte involontairement, n'impacte pas la présence de la session, qui continuera d'exister sur le serveur.

Définition des sessions live

Le protocole tourne autour du concept de *session live*, qui peut être vu comme un endroit virtuel temporaire où plusieurs personnes s'entraînent sur les mêmes exercices au même moment. Une partie des personnes ne participent pas directement, mais observent l'entraînement des autres.

Une personne démarre une session pour un repository qui contient des exercices pour PLX et d'autres rejoignent pour faire ces exercices. Une fois une sélection d'exercices préparée, les exercices sont lancés l'un après l'autre au rythme imposé par le leader. La session vit jusqu'à que le leader l'arrête ou qu'un temps d'expiration côté serveur décide de l'arrêter après un certain temps d'inactivité. L'arrêt d'une session fait quitter tous les clients connectés mais ne coupe pas les connexions WebSocket.

Une session est définie par un nom et un ID de groupe (`group_id`), cette combinaison est unique sur le serveur. Cet ID textuel de groupe est complètement arbitraire. Par défaut, le client PLX va prendre le lien HTTPS du repository Git pour regrouper les sessions du même cours. Cette ID peut paraître inutile, mais elle présente deux intérêts importants: une simplification de l'accès à une session et une limitation du spam.

Si 100 sessions live tournent en même temps, il serait difficile dans l'interface graphique de trouver une session en particulier parmi une liste de 100 entrées. Ainsi, grâce au regroupement par `group_id`, seules les sessions du cours seront listées. Si 1-6 enseignant·es enseignent un cours en même temps, la liste ne sera longue que de 1-6 entrées et le nom de la session sera suffisant pour que les étudiant·es puissent trouver celle qui les intéressent.

Comme notre serveur n'a pas de compte et ni de gestion de classe ou d'école, il n'est pas possible de s'assurer les participant·es font bien parti·es de l'école. Ainsi des personnes externes pourraient tout à fait se connecter et tenter d'envoyer du spam pour perturber les cours. Un premier type d'attaque qui pourrait tromper les étudiant·es, serait de lancer la création automatisée d'autres sessions avec des noms très proches des sessions légitimes. Les étudiant·es pourraient rejoindre la mauvaise session. Une autre attaque plus ennuyante serait la pollution d'une session, simplement en envoyant plein de morceaux de code aléatoires pour des centaines de clients fictifs. Cette attaque rendrait le tableau de bord des enseignant·es inutilisable, comme le code des 30 étudiant·es seraient noyés au milieu de centaines d'autres.

Les deux attaques mentionnées restent possible mais leur accès est rendu plus difficile pour un·e attaquant·e externe à l'école, grâce au `group_id`. Il n'est pas possible de lister toutes les sessions en cours du serveur sans connaître tous les `group_id` utilisés. Nous considérons que ce genre d'attaque n'a dans tous les cas pas tellement d'intérêt pour un·e attaquant·e. Le seul intérêt potentiel serait de récupérer du code des étudiant·es sur les exercices de PLX mais ceux-ci n'ont rien de sensible ni de personnel. Si la pollution du tableau de bord devient un problème, nous pourrions implémenter un système de modération pour ignorer ou bannir certains clients d'une session.

Définition, identifiants et configuration du client

Un « client » est défini comme la partie logicielle de PLX qui se connecte à un serveur de session live. Un client n'a pas besoin d'être codé dans un langage ou pour une plateforme spécifique, le seul prérequis est la capacité d'utiliser le protocole WebSocket. Chaque client est anonyme (le nom n'est pas envoyé, il ne peut pas être connu de l'enseignant·e facilement), mais s'identifie par un `client_id`, qu'il doit persister. Cet ID doit rester secrète entre le client et serveur, sinon il devient possible de se faire passer pour un autre client, ce qui devient problématique pour un client leader.

Par souci de simplicité, les clients PLX génèrent un UUID version 4 (exemple `1be216e1-220c-4a0e-a582-0572096cea07`) (109). Le protocole ne définit pas de contrainte sur le contenu de cet identifiant, un autre format de plus grande entropie pourrait facilement être utilisé plus tard, si une sécurité plus accrue devenait nécessaire.

Les clients leader ont besoin d'identifier la source du message transféré. Quand un nouveau bout de code arrive dans l'interface, soit le bout de code se met en bas de la liste pour un nouveau client, soit il met à jour le code existant du client existant. Le `client_id` ne peut pas être utilisé car il

doit rester secret, il ne doit pas être envoyé vers un autre client. Nous avons besoin d'un autre identifiant, différent à chaque session pour un client donné pour garder l'intérêt de l'anonymat.

La solution choisie consiste à générer un numéro de client `client_num`, valeur entière incrémentale (partant de zéro), attribué par le serveur dans l'ordre d'arrivée dans la session. L'ordre d'arrivée étant très souvent différent, chaque client devrait généralement avec un numéro différent. La deuxième utilité de ce numéro est de permettre aux participant·es de mentionner à l'oral un code spécifique, par exemple: *Je ne comprends pas l'erreur, est-ce que vous pouvez me dire pourquoi mon code ne compile pas, en numéro 8 ?* Ou encore *Sur le code 23 que vous aviez montré avant, est-ce que l'approche était meilleure que ce code 12 ?*

Un client ne peut pas se connecter plusieurs fois simultanément au même serveur. Cela peut arriver lorsque l'on démarre l'application deux fois, le même `client_id` sera utilisé sur deux connexions WebSocket distinctes. Lors de la deuxième connexion, la première est fermée par le serveur après l'envoi d'une erreur. Une fois connecté, chaque client ne peut rejoindre qu'une session à la fois.

Pour qu'un client puisse se connecter au serveur, un repository d'un cours PLX doit contenir à sa racine un fichier `live.toml` avec les entrées visibles sur le Snippet 34.

```
# This is the configuration used to connect to a live server
domain = "live.plx.rs"
port = 9120
group_id = "https://github.com/samuelroland/plx-demo.git"
```

Snippet 34 – Exemple de configuration `live.toml`

Le `port` et le `group_id` sont optionnels. La valeur par défaut du port du protocole est utilisée et le `group_id` par défaut peut être récupéré via l'origine du repository cloné.

Transport, sérialisation et gestion de la connexion

Ce protocole se base sur le protocole Websocket RFC 6455 (94) qui est basé sur TCP. Il utilise le port **9120** par défaut, qui a été choisi parmi la liste des ports non assignés publiés par l'IANA (110). Ce port est également configurable si nécessaire. Les messages, transmis dans le type de message `Text` du protocole WebSocket, sont transmis sous forme de JSON sérialisé en chaîne de caractères.

```
{
  "type": "SendFile",
  "content": {
    "path": "main.c",
    "content": "\n#include <stdio.h>\n\nint main(int argc, char *argv[]) {\n    printf(\"hello world!\\n\");\n    return 0;\n}"
  }
}
```

Snippet 35 – Un exemple de message en format JSON, ici l'action `SendFile`

Pour que le serveur et les clients connectés puissent savoir s'ils communiquent avec une version compatible, il est nécessaire d'envoyer un numéro de version de ce protocole à la première connexion. C'est le serveur qui sera souvent le plus à jour et décidera d'accepter ou refuser la connexion, en renvoyant un code HTTP 400 s'il la refuse.

Comme le montre le Snippet 36, les clients doivent se connecter en indiquant la version du protocole supportée par le client (`live_protocol_version`) et le `client_id` présenté précédemment

(`live_client_id`). Si cette première requête ne contient pas ces informations, le serveur la refusera également.

```
ws://liveplx.rs:9120?live_protocol_version=0.1.0&live_client_id=e9fc3566-32e3-4b98-99b5-35be520d46cb
```

Snippet 36 – Lien de connexion en WebSocket

Les navigateurs web ne pouvant pas définir des entêtes HTTP via l'API `WebSocket`, il est nécessaire de passer via la querystring.

Pour ce numéro de version on utilise le Semantic Versioning 2.0.0 (111). Le numéro actuel est `0.1.0` et restera sur la version majeur zéro (`0.y.z`) durant la suite du développement, jusqu'à que le protocole ait pris de la maturité.

La connexion WebSocket devrait se terminer comme le protocole WebSocket le définit, c'est à dire en fermant proprement la connexion WebSocket avec un message de type `Close`.

Messages

Voici les actions définies, avec l'événement associé en cas de succès de l'action. Tous les champs et le message final en JSON doivent être encodés en UTF-8. Toutes les dates sont générées par le serveur en UTC. Les dates sont sérialisées sous forme de timestamp¹.

Identifiant	Clients autorisés	But	Événement associé	Événement envoyé
Action::StartSession	tous	Démarrer une session	Event::SessionJoined	même client
		<pre>{ "type": "StartSession", "content": { "name": "PRG2 Jack", "group_id": "https://github.com/prg2/prg2.git" } }</pre>		<pre>{ "type": "SessionJoined", "content": 4 }</pre> <p>Le numéro retourné est le <code>client_num</code>.</p>
Action::GetSessions	tous	Lister les sessions ouvertes pour un <code>group_id</code> donné	Event::SessionsList	au même client
		<pre>{ "type": "GetSessions", "content": { "group_id": "https://github.com/prg2/prg2.git" } }</pre>		<pre>{ "type": "SessionsList", "content": [{ "name": "PRG2 Jack", "group_id": "https://github.com/prg2/prg2.git" }, { "name": "PRG2 Alissa", "group_id": "https://github.com/prg2/prg2.git" }] }</pre>
Action::JoinSession	tous	Rejoindre une session en cours	Event::SessionJoined	au même client
		<pre>{ "type": "JoinSession", "content": { "name": "PRG2 Alissa", "group_id": "https://github.com/prg2/prg2.git" } }</pre>		<pre>{ "type": "SessionJoined", "content": 4 }</pre> <p>C'est le même message pour les leaders et followers.</p>
Action::LeaveSession	tous	Quitter la session en cours	Event::SessionLeaved	au même client

¹Le nombre de secondes depuis l'époque Unix (1er janvier 1970).

<pre>{ "type": "LeaveSession" }</pre>			<pre>{ "type": "SessionLeaved" }</pre>	
Action::StopSession	le leader qui a démarré la session	Arrêter la session en cours	Event :: SessionStopped	à tous les clients de la session
<pre>{ "type": "StopSession" }</pre>			<pre>{ "type": "SessionStopped" }</pre>	
Action::SendFile	followers	Envoyer une nouvelle version d'un fichier	Event :: ForwardResult	aux clients leaders de la session
<pre>{ "type": "SendFile", "content": { "path": "main.c", "content": "\n#include <stdio.h>\n\nint main(int argc, char *argv[]) {\n printf(\"hello world!\\n\");\n return 0;\n}\n" } }</pre>			<pre>{ "type": "ForwardFile", "content": { "client_num": 23, "file": { "path": "main.c", "content": "\n#include <stdio.h>\n\nint main(int argc, char *argv[]) {\n printf(\"hello world!\\n\"); return 0;\n}\n\n" }, "time": 1751632509 } }</pre>	
Tout le contenu du fichier de code est envoyé peu importe la nature de la modification.				
Action::SendResult	followers	Envoyer le résultat d'un check	Event :: ForwardResult	aux clients leaders de la session
<pre>{ "type": "SendResult", "content": { "check_result": { "index": 3, "state": { "type": "Passed" } } } }</pre>			<pre>{ "type": "ForwardResult", "content": { "client_num": 12, "result": { "check_result": { "index": 0, "state": { "type": "Passed" } }, "time": 1751632509 } } }</pre>	
Autres exemples de Action::SendResult				
<pre>{ "type": "SendResult", "content": { "check_result": { "index": 0, "state": { "type": "BuildFailed", "content": "main.c: In function 'main':\nmain.c:4:5: error: 'a' undeclared" } } } }</pre>			<pre>{ "type": "SendResult", "content": { "check_result": { "index": 1, "state": { "type": "CheckFailed", "content": "Hello Doe !" } } } }</pre>	
<pre>{ "type": "SendResult", "content": { "check_result": { "index": 1, "state": { "type": "RunFailed", "content": "Hello\\nsegfault" } } } }</pre>				
Action::SwitchExo	leaders	Changer d'exercice actuel de la session, identifié par un chemin relatif	Event :: ExoSwitched	à tous les clients de la session
<pre>{ "type": "SwitchExo", "content": { "path": "structs/hello-dog" } }</pre>			<pre>{ "type": "ExoSwitched", "content": { "path": "intro/salue-moi" } }</pre>	



Il reste encore des événements indépendants. L'événement `Stats` sur le Snippet 37 est envoyé aux leaders à chaque fois qu'un client rejoint ou quitte la session, excepté quand le leader créateur rejoint. L'événement `ServerStopped` sur le Snippet 38 est envoyé à tous les clients lorsqu'il doit s'arrêter.

```
{
  "type": "Stats",
  "content": {
    "followers_count": 32,
    "leaders_count": 2
  }
}
```

```
{
  "type": "ServerStopped"
}
```

Snippet 38 – Message `Event::ServerStopped`Snippet 37 – Message `Event::Stats`,
reçu uniquement par les clients leaders

Pour conclure cette liste de messages, voici la liste des types d'erreurs qui peuvent être reçues du serveur via un `Event::Error`, contenant différents types de `LiveProtocolError`. Ces erreurs peuvent arriver dans différents contextes et ne sont pas toujours liées à une action précise. Une partie des erreurs ne peuvent pas arriver si le client gère correctement son état et ne tente pas des actions non autorisées par son rôle.

```
{
  "type": "Error",
  "content": {
    "type": "FailedToStartSession",
    "reason": "There is already a session with the same group id and name combination."
  }
}
```

Event :: Error(LiveProtocolError::FailedToStartSession)

```
{
  "type": "Error",
  "content": {
    "type": "FailedToJoinSession",
    "reason": "No session found with this name in this group id"
  }
}
```

Event :: Error(LiveProtocolError::FailedToJoinSession)

```
{
  "type": "Error",
  "content": {
    "type": "FailedSendingWithoutSession"
  }
}
```

Event :: Error(LiveProtocolError::FailedSendingWithoutSession)

```
{
  "type": "Error",
  "content": {
    "type": "FailedToLeaveSession"
  }
}
```

Event :: Error(LiveProtocolError::FailedToLeaveSession)

```
{
  "type": "Error",
  "content": {
    "type": "SessionNotFound"
  }
}
```

Event :: Error(LiveProtocolError::SessionNotFound)

```
{
  "type": "Error",
  "content": {
    "type": "CannotJoinOtherSession"
  }
}
```

Event :: Error(LiveProtocolError::CannotJoinOtherSession)

```
{
  "type": "Error",
  "content": {
    "type": "ForbiddenSessionStop"
  }
}
```

Event :: Error(LiveProtocolError::ForbiddenSessionStop)

```
{
  "type": "Error",
  "content": {
    "type": "ActionOnlyForLeader",
    "reason": "switch of exo"
  }
}
```

Event :: Error(LiveProtocolError::ActionOnlyForLeader)

La gestion de plusieurs clients leaders, promu par le leader créateur, n'est pas encore supportée dans le protocole. Il suffirait d'ajouter deux messages pour ajouter ou retirer le rôle de leader à un client donné.

Diagrammes de séquence

Maintenant que les différents types de messages sont connus, voici quelques diagrammes de séquence pour mieux comprendre le déroulement d'une session et l'ordre des messages.

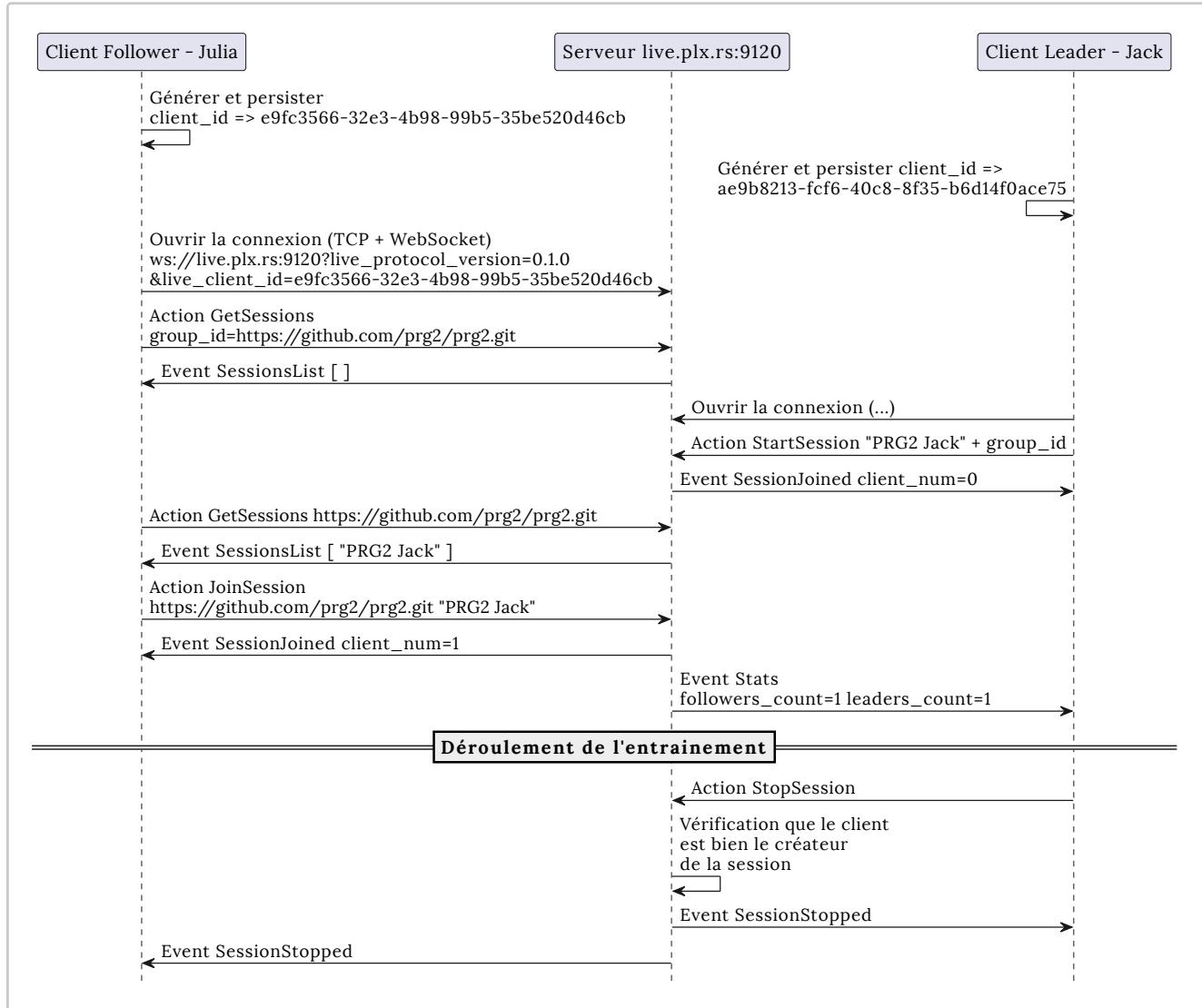


Figure 25 – Exemple de communication avec gestion d'une session

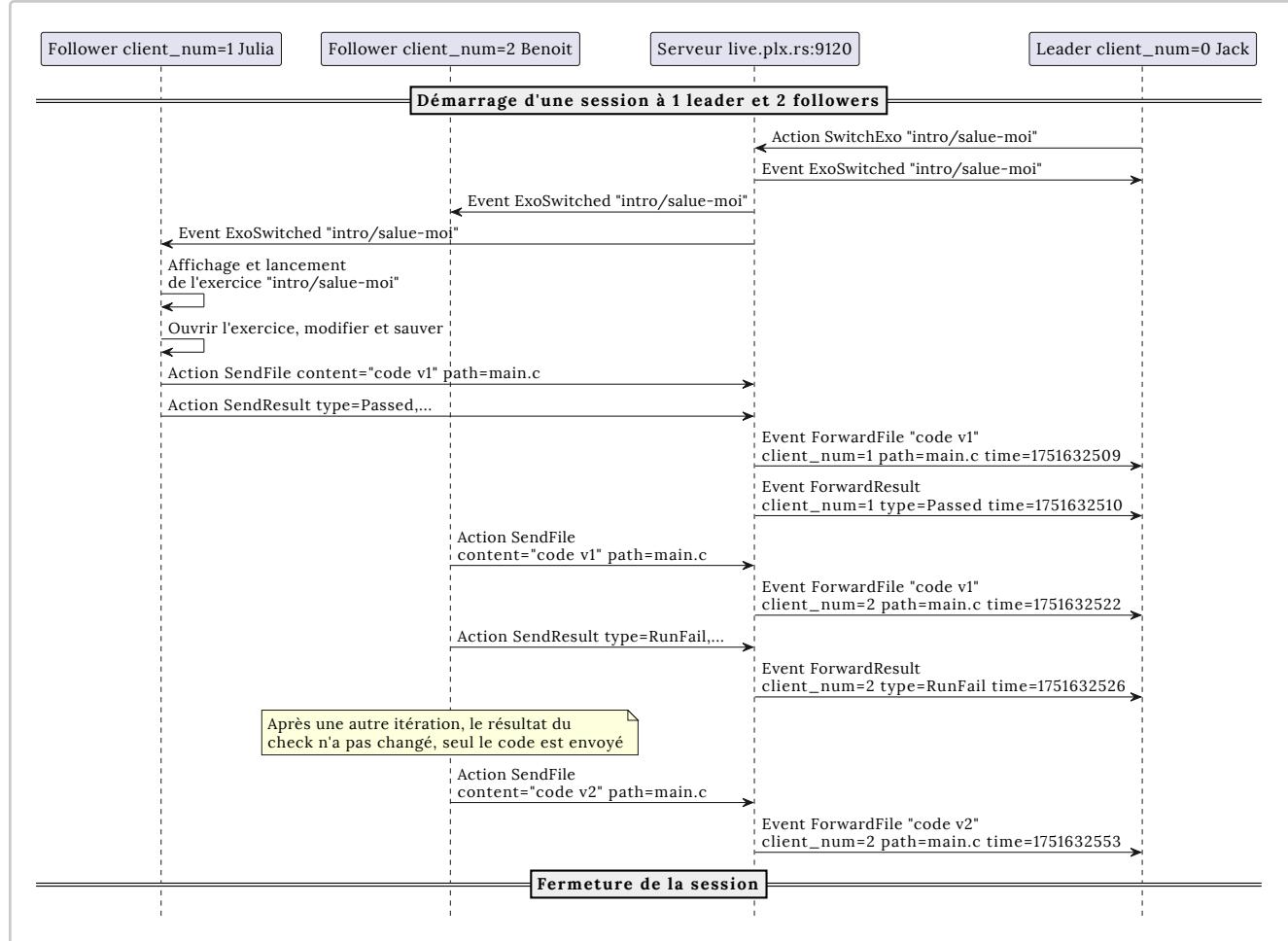


Figure 26 – Exemple de communication avec transferts des bouts de code et des résultats

Lors de la réception d'un signal d'arrêt (par exemple lors d'un `Ctrl+c`), le processus du serveur ne doit pas quitter immédiatement. Les sessions en cours doivent être arrêtées et tous les clients doivent recevoir un `Event :: ServerStopped` qui informe de l'arrêt du serveur.

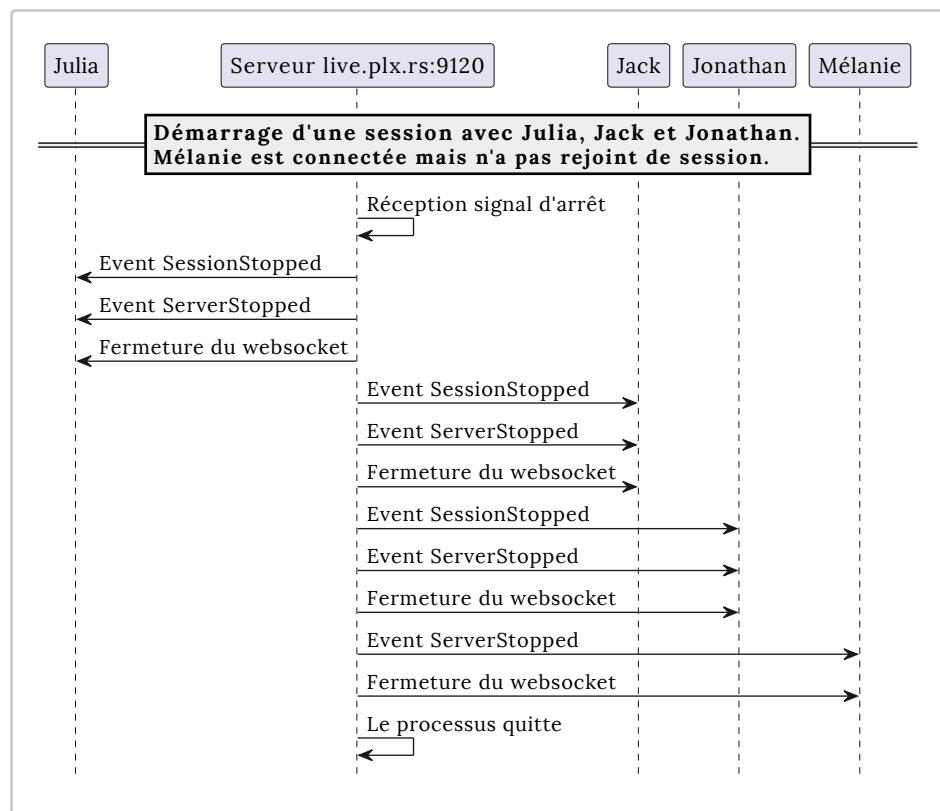


Figure 27 – Exemple de communication qui montre l'arrêt du serveur, avec différents clients dans une session ou en dehors

Gestion des pannes

Nous avons fait le choix que le serveur de garder les données du serveur en mémoire vive uniquement, les données des sessions en cours ne sont pas persistées. La première raison est la simplicité d'implémentation: pas de base de données ni d'état sur le disque à synchroniser. La deuxième est que la majorité des informations sont temporaires comme le serveur agit principalement comme relai. Les cas de crash devraient être très rares grâce aux garanties de sécurité mémoire de Rust. Le serveur deviendrait indisponible en cas de surcharge ou de redémarrage pour des mises à jour.

En cas de redémarrage, les participant·es devraient recréer ou rejoindre les nouvelles sessions à la main. Le tableau de bord des leaders prendraient quelques minutes à retrouver son état précédent en attendant que tous les participant·es aient sauvés et envoyés leur fichier une première fois.

Pour limiter la fréquence des dérangements, les mises à jour du serveur seront faites en dehors des heures de cours. Le conteneur Docker du serveur sera lancé en mode redémarrage automatique, afin de rendre le serveur à nouveau disponible après un crash, en espérant que cela ne se produise pas en boucle.

Pour simplifier le développement et la logique de reconnexion, les clients n'ont pas besoin de persister l'état de la session, comme l'identifiant de l'exercice en cours. Durant la connexion d'un client, leader ou follower, le serveur doit renvoyer le dernier message `ExoSwitched` qu'il a envoyé dans cette session. Pour un client leader, le serveur doit aussi lui renvoyer tous les derniers `Event :: ForwardFile` et `Event :: ForwardResult` pour chaque client follower. Ce transfert est nécessaire pour que l'interface reprenne le même état qu'avant déconnexion².

²Autrement, le leader devrait attendre les prochains envois de ces événements pour chaque follower afin d'avoir une interface à jour.

Pour un follower déconnecté temporairement, son leader ne devrait pas voir 2 versions du même code avant et après redémarrage, mais uniquement la dernière version. Pour permettre cette expérience, un client qui se reconnecte à une session doit récupérer le même `client_num` qu'avant déconnexion. Le serveur doit maintenir pour chaque session un lien entre `client_id` et `client_num` pour chaque client.

Evolutivité

Le concept de session lancée par des leaders et de transfert de données provenant de followers vers des leaders, peut facilement être étendu à d'autres contextes d'apprentissage. Si on souhaite pour entraîner en live d'autres types d'exercice, comme des choix multiples, il suffirait d'ajouter une nouvelle action `Action :: SendChoice` pour envoyer une réponse et un événement associé (`Event :: ForwardChoice`), pour renvoyer cette réponse vers les clients leaders.

Dans le futur, de nouveaux formats d'exercices seront supportés par PLX. Si cela implique de changer trop souvent la structure des résultats dans le champ `content.check_result` dans le message `Event :: SendResult`, une solution serait de ne pas spécifier la structure exacte de ce sous champ et laisser les clients gérer les structures non définies ou partielles. Cela pourrait éviter de régulièrement devoir augmenter le numéro de version majeure à cause de *breaking change*.

Vue d'ensemble de l'implémentation

Nous avons implémenté un nouveau module Rust nommé `live` dans la librairie existante de PLX. Cette librairie est prévue pour un usage interne actuellement et n'est pas pensée pour être réutilisée par d'autres projets. Ce module `live` contient plusieurs fichiers pour implémenter le protocole et le serveur.

La librairie `plx-core` et son module `live`

Dans la Figure 28, l'application desktop et le CLI dépendent de cette librairie `plx-core`. Le CLI contient une sous-commande `plx server` pour démarrer le serveur. L'application desktop dépend du code Rust des modules existants `app`, `core` et `models` qui rendent possible l'entraînement local. Elle dépend aussi de `LiveConfig` pour charger un fichier `live.toml`.

Le fichier `protocol.rs` contient toutes les structures de données autour des messages du protocole: `Session`, `ClientNum`, `ClientRole`, les messages `Action` et `Event` et les types d'erreurs `LiveProtocolError`. Le reste des fichiers implémente les différentes tâches concurrentes gérées par le serveur. Le point d'entrée du serveur est la structure `LiveServer`. Le module `live` dépend aussi de `tokio` pour gérer la concurrence des tâches et `tokio-tungstenite` pour l'implémentation WebSocket.

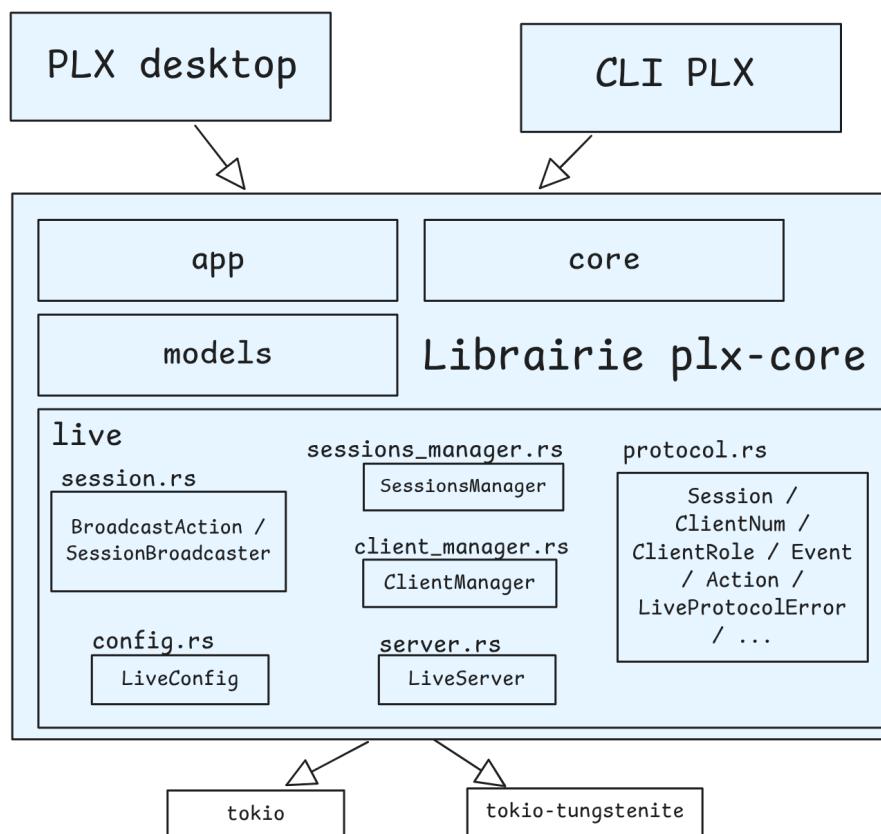


Figure 28 – Aperçu du nouveau module `live` de la librairie `plx-core`

PLX est développé avec Tauri (112), framework permettant de créer des applications desktop en Rust. Tauri est une alternative à ElectronJS (113) et permet de créer une application desktop basé sur une application web. La partie frontend est écrit en VueJS (4) et reste isolée dans une « fenêtre de navigateur », tandis que la partie backend en Rust permet d'accéder aux fichiers et aux commandes systèmes. Tauri permet ainsi de définir des fonctions Rust exposée au frontend, appelée commandes Tauri (114).

Les processus en jeu

Sur la Figure 29, on voit les différentes communications réseau et processus en jeu. Le binaire `plx-desktop` lance deux processus. Le backend et frontend discutent via le système de communication inter-processus (IPC) de Tauri (qui utilise JSON-RPC (52)). Les commandes Tauri utilisent notre librairie. Les commandes Tauri sont appelées depuis `commands.ts` et `shared.ts` contient des types communs.

Le serveur PLX peut être déployé dans un conteneur Docker via la commande `plx server`. Le client est implémenté en TypeScript dans `client.ts` et se connecte au serveur.

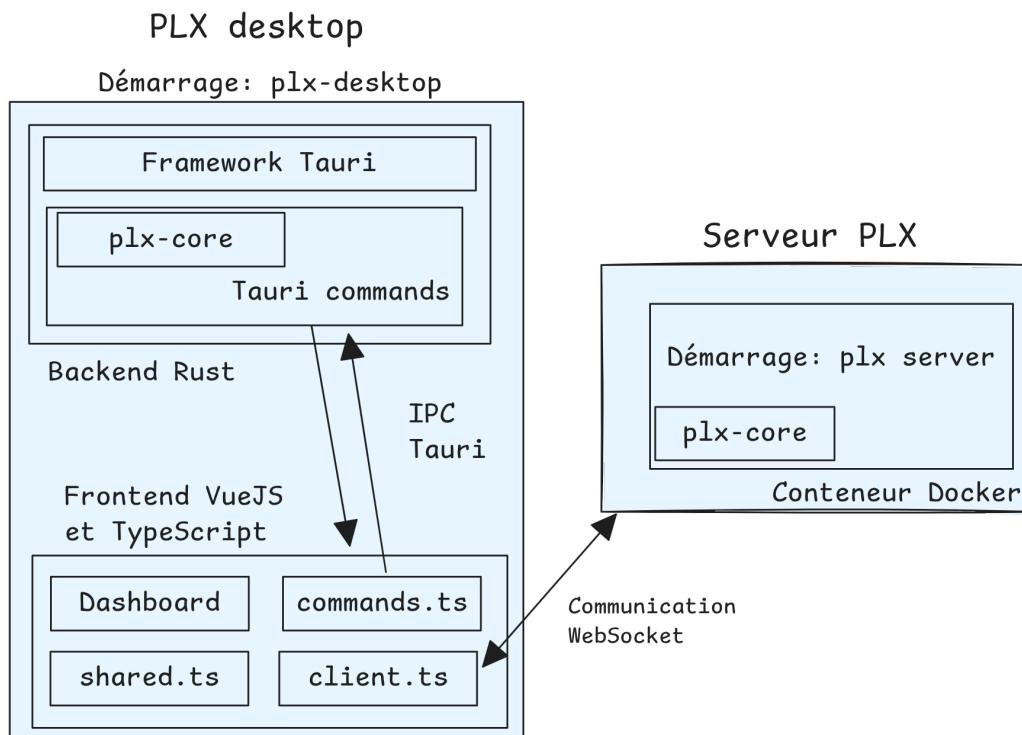


Figure 29 – Aperçu du réseau et processus qui composent le projet PLX

Typage des commandes Tauri

Pour les commandes Tauri mises à disposition du frontend, l'appel d'une commande se fait par défaut via une fonction `invoke` faiblement typée: le nom de la commande est une `string` et les paramètres sont mis dans un objet, comme montré sur le Snippet 39. Les types de ces valeurs ne sont pas vérifiés à la compilation, seule l'exécution permet de trouver des erreurs dans la console de la fenêtre du frontend. En cas de changement de signature en Rust, nous pourrions oublier d'adapter le code du frontend sans s'en rendre compte.

```
#[tauri::command]
pub async fn clone_course(repos: String) -> bool {
    let base = get_base_directory();
    GitRepos::from_clone(&repos, &base).is_ok()
}
```

```
import { invoke } from "@tauri-apps/api/core";
const success = await invoke("clone_course", {
    repos: "https://github.com/samuelroland/plx-demo"
});
```

Snippet 39 – Une commande Tauri en Rust pour cloner le repository d'un cours et son appel faiblement typé en TypeScript.

Pour résoudre ce problème, le projet `tauri-specta` (115) nous permet de générer une définition une fonction bien typée de l'appel à la commande, après avoir annoté la fonction Rust et les types des paramètres.

```

rust
#[derive(Serialize, Debug, specta::Type)]
pub struct CourseWithConfig {
    course: Course,
    config: Option<LiveConfig>,
}
#[tauri::command]
#[specta::specta]
pub async fn get_local_courses() -> Vec<CourseWithConfig> {
    // ...
}

```

Figure 30 – Exemple d'annotation avec `tauri-specta` sur la commande Tauri et sur les structures associées

Le Figure 30 démontre comment annoter une commande Tauri avec `#[specta::specta]`. Cette commande permet de récupérer les cours PLX locaux et retourne un vecteur de `CourseWithConfig`. Notre structure `CourseWithConfig` et les types utilisés dans ses champs, tels que `Course` et `LiveConfig`, ont également été annotées avec `specta::Type`.

```

// Commande TypeScript autogénérée par tauri-specta
export const commands = {
    async cloneCourse(repos: string): Promise<boolean> {
        return await invoke("clone_course", { repos });
    }
}
// Exemple d'appel dans Home.vue
const success = await commands.cloneCourse("https://github.com/samuelroland/plx-demo")

```

Snippet 40 – Différence d'appel des commandes grâce à `tauri-specta`, par rapport à Snippet 39

Si la commande en Rust changeait de nom, de type des paramètres ou de valeur de retour, maintenant que l'appel est typé, le frontend ne compilera plus et le changement nécessaire en TypeScript ne pourrait pas être oublié. Le fichier généré est `desktop/src/ts/commands.ts`.

Partage des types

Les structures de données comme `Action`, `Event`, `LiveProtocolError`, `Session`, `CheckStatus` sont également utiles du côté du client TypeScript. On aimerait éviter de devoir définir des types TypeScript manuellement en doublon des types Rust afin de faciliter le développement et les changements du protocole. Il existe plusieurs solutions pour exporter les types Rust vers des types TypeScript équivalent. Le CLI `typeshare` (116) a permis d'exporter automatiquement les types communs, en activant l'export via une annotation `#[typeshare]`. Le fichier généré est `desktop/src/ts/shared.ts`.

Prenons un exemple avec le résultat d'un check. L'attribut `#[serde ...]` demande que le `CheckStatus` soit sérialisé avec un champ discriminant `type` et son contenu sous un champ `content`. Cette conversion est nécessaire pour permettre de générer un équivalent TypeScript.

```

#[derive(Serialize, Deserialize, Eq, PartialEq, Clone,
Debug)]
#[serde(tag = "type", content = "content")]
#[typeshare]
pub enum CheckStatus {
    Passed,
    CheckFailed(String),
    BuildFailed(String),
    RunFailed(String),
}
#[derive(Serialize, Deserialize, Eq, PartialEq, Clone,
Debug)]
#[typeshare]
pub struct ExoCheckResult {
    pub index: u16,
    pub state: CheckStatus,
}

```

```

export type CheckStatus =
| { type: "Passed", content?: undefined }
| { type: "CheckFailed", content: string }
| { type: "BuildFailed", content: string }
| { type: "RunFailed", content: string };

export interface ExoCheckResult {
    index: number;
    state: CheckStatus;
}

```

Snippet 42 – Equivalent TypeScript des 2 types.
Les types `u16` et `String` ont été pu converti vers `number` et `string`

Snippet 41 – 2 types Rust pour décrire le résultat d'un check.

Implémentation du client

Le client a été développé dans le *frontend* de PLX pour simplifier l'implémentation. En effet, une partie des messages pourrait être envoyée depuis la librairie, par exemple dès qu'un résultat de check est disponible. Et une autre partie viendrait de l'interface graphique via les boutons de gestion de sessions. Nous ne pouvons pas avoir deux connexions WebSocket séparées, les messages du *frontend* devrait donc passer par des commandes Tauri pour arriver sur le socket géré en Rust. Pour les actions de sessions, cela peut fonctionner mais cela devient difficile à gérer lorsqu'il faut attendre des événements comme `ForwardFile` et `ForwardResult` sur le websocket et en même temps de continuer de pouvoir envoyer des actions comme `StopSession`.

Après avoir tenté l'approche précédente, gérer tout le client dans le *frontend* s'est révélé beaucoup plus simple. Au final, l'entraînement dans une session live ne fait de différence que dans l'interface. Le backend de l'application desktop n'a pas besoin de se préoccuper de savoir si l'exercice est fait seul-e ou dans une session live. A la réception de résultats des checks ou d'erreurs de compilation, le *frontend* se charge d'envoyer des actions `SendFile` et `SendResult`.

Implémentation du tableau de bord

Avant de présenter l'implémentation technique, voici un aperçu du tableau de bord réalisé pour les clients leaders et des changements d'interface pour les clients followers.

Screenshots à venir, quand l'interface aura été un poil amélioré et que le switch d'exo fonctionnera.

todo création de session

todo rejoindre la session

todo les stats

todo choix des exos

todo switch d'exos

todo lancement d'un exo étudiant, erreur de build

todo code actuel et erreur de build disponible dans le dashboard

Implémentation du serveur

Lancement

Une fois lancé via la commande `plx server`, si on y connecte un client et qu'on envoie des actions, on peut directement voir sur le Snippet 43 des logs pour visualiser les messages reçus et envoyés.

```
> plx server
Started PLX server on port 9120
ClientManager for new client
SERVER: Received from 4cd31b74-0192-4900-8807-70912cc9d5d8: {
    "type": "GetSessions",
    "content": {
        "group_id": "https://github.com/samuelroland/plx-demo"
    }
}
SERVER: Sending to 4cd31b74-0192-4900-8807-70912cc9d5d8: {
    "type": "SessionsList",
    "content": []
}
SERVER: Received from 4cd31b74-0192-4900-8807-70912cc9d5d8: {
    "type": "StartSession",
    "content": {
        "name": "jack",
        "group_id": "https://github.com/samuelroland/plx-demo"
    }
}
SERVER: Sending to 4cd31b74-0192-4900-8807-70912cc9d5d8: {
    "type": "SessionJoined",
    "content": 0
}
```

Snippet 43 – Sortie console du serveur à la réception de l'action `GetSessions`, répondue par un `SessionsList`, puis un `StartSession` reçu ce qui génère un `SessionJoined`.

Gestion de la concurrence

L'exemple précédent ne comportait qu'un seul client, en pratique nous en auront des centaines connectés en même temps, ce qui pose un défi de répartition du travail sur le serveur. En effet, le serveur doit être capable de faire plusieurs choses à la fois, dont une partie des tâches qui sont bloquantes:

1. Réagir à la demande d'arrêt, lors d'un `Ctrl+c`, le serveur doit s'arrêter proprement pour fermer les sessions et envoyer un `Event :: ServerStopped`.
2. Attendre de futurs clients qui voudraient ouvrir une connexion TCP
3. Attendre de messages sur le socket pour chaque client
4. Parser le JSON des messages des clients et vérifier que le rôle permet l'action
5. Parcourir la liste des clients d'une session pour leur broadcaster un message l'un après l'autre
6. Envoyer un `Event` pour un client donné
7. Gérer les sessions présentes, permettre de rejoindre ou quitter, de lancer ou d'arrêter ces sessions

Une approche basique serait de lancer un nouveau *thread* natif (fil d'exécution, géré par l'OS) à chaque nouveau client pour que l'attente sur le socket des messages envoyés puisse se faire sans bloquer les autres. Cette stratégie pose des problèmes à large échelle, car un *thread* natif possède un coût non négligeable. L'ordonnancement de l'OS, qui décide sur quel coeur du processeur pourra travailler chaque *thread* et à quel moment, a un certain cout. Si on démarre des centaines de *threads* natifs, l'ordonnanceur va perdre beaucoup de temps à constamment ordonner tous ces *threads* et les mettre en place.

Une solution à ce problème est de passer vers du Rust `async`. Concrètement, les fonctions asynchrones sont préfixées du mot-clé `async` et des appels de ces fonctions suffixés de `.await`). Grâce au runtime `Tokio`, librairie largement utilisée dans l'écosystème Rust, le code devient asynchrone

grâce au lancement de threads virtuels, appelés « tâches Tokio » (117). Au lieu d'être soumis à un ordonnancement préemptif de l'ordonnanceur de l'OS, les tâches Tokio ne sont pas préemptées mais redonnent le contrôle au *runtime* à chaque `.await`. Cette asynchronisme permet d'attendre le résultat du réseau sans bloquer le *thread* natif sur laquelle est exécutée la tâche. Seul la tâche tokio sera mis dans un fil d'attente géré par le *runtime* pour être relancée plus tard une fois le résultat arrivé. Le *runtime* lui même ordonne ses tâches sur plusieurs *threads* natifs, pour permettre un parallélisme en plus de la concurrence existante.³

Ce runtime de threads virtuelles permet ainsi de lancer des milliers de tâches tokio avec un faible cout mémoire ou du temps nécessaire à leur ordonnancement qui est plus léger. Tokio est donc une solution bien adaptée aux applications en réseau avec de nombreux clients concurrents mais aussi beaucoup d'attente sur des entrées/sorties. (117)

³Plus de détails sur les tâches Tokio sont disponibles dans sa documentation (117)

Tâches tokio

Sur la Figure 31, nous observons 5 clients connectés, les clients 1 à 4 sont connectés dans la session 1 et le 5ème n'est pas connecté à une session. Le client en orange est le leader de la session 1. Chaque rectangle à l'intérieur du serveur (sauf le `SessionsManagement`), correspond à une tâche Tokio indépendante, avec laquelle la communication se fait uniquement par *channel* (système de messages aussi appelé *message passing*).

Le point d'entrée est le `LiveServer`. Il attend sur le port 9120 dans l'attente de nouveaux clients qui veulent se connecter en TCP. Il doit ensuite gérer l'initialisation de la connexion WebSocket (*handshake*), qui inclut la vérification du numéro de version et de la présence du `client_id`. Une fois le client connecté avec succès, le `LiveServer` lance un nouveau `ClientManager` dans une tâche Tokio séparée. Cette tâche devient propriétaire de l'instance du WebSocket et reçoit aussi une référence partagée sur le `SessionsManagement`. Le `LiveServer` peut ainsi continuer d'attendre d'autres clients.

Le `SessionsManagement` n'est qu'un état partagé au serveur qui implémente différentes méthodes pour changer cet état. Il est initialisé par le `LiveServer` et accédé par les `ClientManager` et s'occupe de stocker les informations relatives aux sessions en cours.

Le `ClientManager` étant de seul à accéder au socket de son client, il doit s'occuper de recevoir des messages `Action` du client. Il doit aussi transmettre sur le socket les `Event` que le `SessionBroadcaster` lui transfère.

Tous les `ClientManager` ont un *channel* (flèche violette) vers les `SessionBroadcaster`. Comme leur nom l'indique, ils ne servent qu'à broadcaster un message à tous les leaders ou tous les clients de la session. Leur état stocke la partie transmission du *channel* vers chaque `ClientManager` de la session (les flèches vertes).

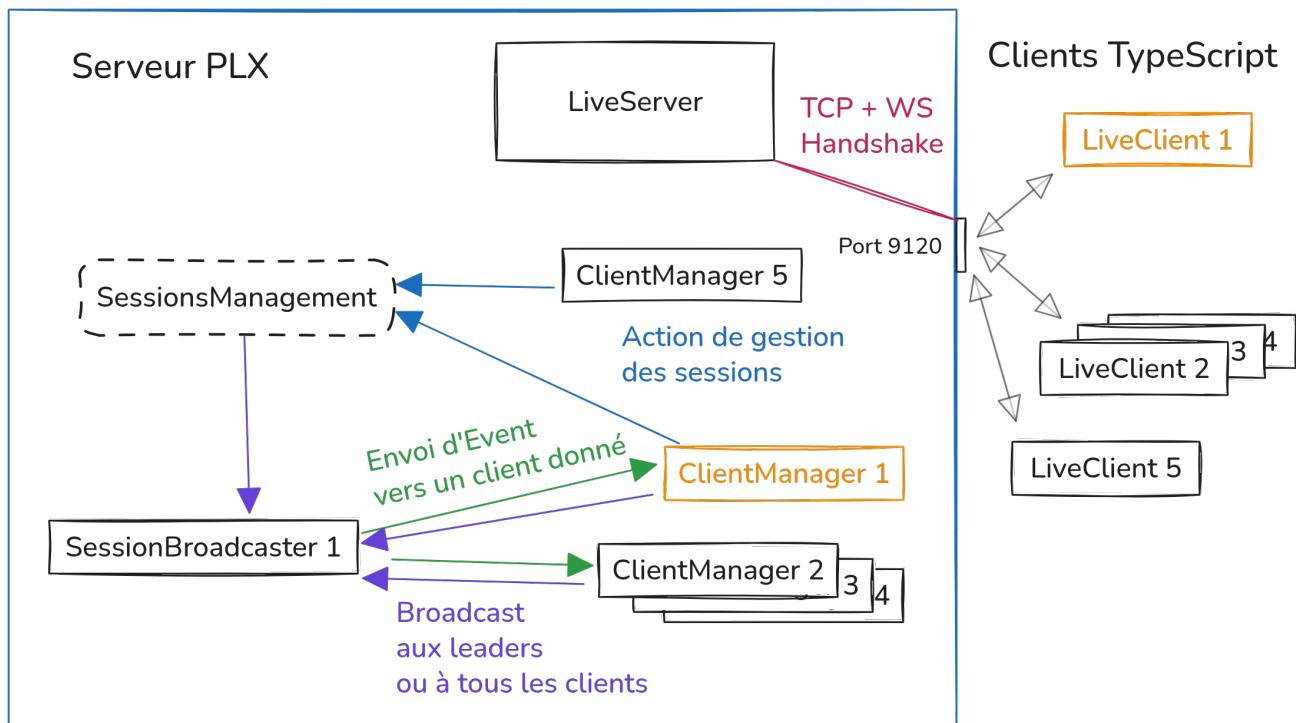


Figure 31 – Aperçu des tâches tokio lancées et interactions possibles pour 5 clients et 1 session en cours.

Le `ClientManager`, lors de la réception d'un message, doit parser le JSON du message vers la structure `Action`. Une fois le message extrait, il doit vérifier que le rôle permet l'action ou alors renvoyer une erreur directement. Si la demande est autorisée et concerne la gestion de sessions alors il

peut l'effectuer en utilisant une des méthodes de `SessionsManagement`, comme `get_sessions()` par exemple. Dans le cas d'un `SendFile`, si le client est bien dans une session, il peut directement créer le `ForwardFile` et l'envoyer au `SessionBroadcaster` pour qu'il puisse l'envoyer à tous les leaders.

Le `SessionsManagement` possède deux `HashMap` (tables de hachage avec clés/valeurs): la première contient des sessions regroupées par `group_id`. Chaque session contient évidemment le nom et `group_id`, mais également le `client_id` du créateur de la session et le dernier `client_num` attribué. Une seconde liste existe pour lier `client_id` de leader vers la session créée pour facilement retrouver la session dans la première liste.

TODO besoin de voir les actions effectuées de bout en bout pour un message `SendFile` pour mieux se représenter les interactions ou déjà clair ??

Tests de bouts en bouts

TODO

```
#[test]
#[ntest::timeout(4000)]
fn cannot_create_same_session_twice() {
    let c = &mut spawn_server_and_n_clients(2);
    c[0].send_msg(Action::StartSession {
        name: NAME.to_string(),
        group_id: GROUP_ID.to_string(),
    });
    assert_eq!(
        c[0].wait_on_next_event().unwrap(),
        Event::SessionJoined(ClientNum(0))
    );
    c[1].send_msg(Action::StartSession {
        name: NAME.to_string(),
        group_id: GROUP_ID.to_string(),
    });
    assert_eq!(
        c[1].wait_on_next_event().unwrap(),
        Event::Error(LiveProtocolError::FailedToStartSession(
            "There is already a session with the same group id and name
            combination.".to_string()
        ))
    );
}
```

Snippet 44 – Exemple de tests de bout en bout

Développement de la syntaxe DY

Cette partie documente les besoins de PLX, la définition et l'implémentation de la syntaxe DY, son parseur, l'intégration dans PLX et son usage via le CLI.

Vue d'ensemble

Tout l'enjeu de cette syntaxe DY est d'arriver à convertir un bout de texte vers une struct Rust.

```
// Définition du cours
course Programmation 1
code PRG1
goal Apprendre des bases solides du C++
```

Figure 32 – Définition d'un cours PLX en syntaxe DY

```
struct Course {
    name: String,
    code: String,
    goal: String,
}
```

Snippet 45 – Struct Rust
d'un cours PLX

Dans la Figure 32, les clés sont `course`, `code` et `goal`, chaque clé introduit une valeur. Le but est de remplir la struct Rust du Snippet 45 avec ces valeurs.

Après avoir défini les lignes directrices de conception, nous sommes parti du modèle de données de PLX, pour lister les clés nécessaires. Cette idée de syntaxe DY pourrait être utile à d'autres projets ou d'autres d'exercices, nous ne voulons pas construire un parseur uniquement pour PLX. Nous cherchons à mettre en place une abstraction qui nous permet rapidement de définir d'autres clés et convertir ces données dans une struct Rust associée. La syntaxe DY se base sur une hiérarchie de clés qui permet de l'autre côté, au cœur du parseur, d'extraire le contenu et de le valider en partie.

Pour des erreurs plus spécifiques que ce qui est possible via les contraintes définies sur la spécification d'une clé, il est possible de définir en Rust des validations plus avancées.

Pour la détection d'erreurs, si on adoptait l'approche des compilateurs de langages de programmation qui échoue la compilation à la moindre erreur, l'expérience serait très frustrante. Au moindre exercice mal retranscrit parmi une centaine présents, tout le cours serait inaccessible dans l'interface de PLX. Nous préférons au contraire accepter d'avoir des objets partielles (un exercice avec un titre vide, mais une consigne et des checks valides par exemple) et d'afficher les erreurs dans l'interface pour avertir des erreurs présentes. Les parties éronnées ne sont pas extraites pour ne pas impacter le reste des données valides.

Le parseur prend en entrée une `String` directement et n'est pas responsable d'aller lire un fichier. Ceci nous permet de parser du contenu sans avoir de fichier sous jaçant, notamment dans des snippets de DY intégrée à une documentation web. Cela laisse aussi le choix de choisir les noms de fichiers par le projet qui intègre la syntaxe. Nous verrons quels fichiers PLX a choisi d'utiliser pour stocker son modèle de données.

Tout le développement du parseur s'est fait en *Test Driven Development* (TDD), qui s'est révélé très facile à mettre en place comme chaque étape possède des entrées et sorties bien définies.

L'extension de fichier recommandée est `.dy`. Ces fichiers doivent être encodés en UTF8 et le caractère de retour à la ligne doit être le `\n`.

Librairies implémentées

Nous ne voulons pas créer de parseur spécifique aux données d'un cours ni même des données de PLX. A la place, nous souhaitons pouvoir utiliser une abstraction qui nous permet de facilement définir des nouveaux objets en DY et avec le minimum de code pour l'extraire dans une struct Rust dédiée. Nous ne pouvons pas tout implémenter au même endroit, car cela demanderait de constamment changer la logique du parseur pour s'adapter à de nouvelles clés.

L'implémentation est donc divisée en deux parties très claires: le cœur du parseur et les spécifications DY (appelées par la suite « spec DY »). Les deux sont indispensables et leur combinaison permet de parser du contenu défini par un spec DY.

Le projet PLX a maintenant 3 librairies. En plus de `plx-core` déjà présenté pour le développement du serveur, nous avons créé deux crate: le cœur du parseur dans la crate librairie `dy` et la spec DY pour PLX dans la crate librairie `plx-dy`. Cet dernière peut ainsi être utilisé par le CLI avec des fonctions haut-niveaux comme `parse_course()`.

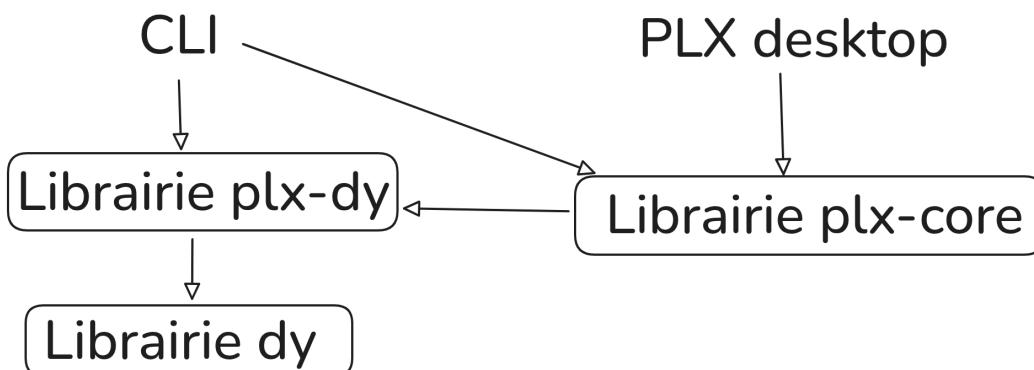


Figure 33 – Aperçu des librairies en jeu, et des dépendances entre librairie et du CLI et de l'application desktop.

Le cœur du parseur ne connaît que le concept de clé, de leur hiérarchie, des propriétés et des commentaires. Il ne sait pas savoir quelles clés et propriétés seront utilisées au final. Une spec DY définit des clés et leur hiérarchie. Elle définit en Rust comment construire la struct final à partir du résultat du cœur du parseur. Elle peut aussi générer d'autres erreurs spécifiques si nécessaire.

Pour faire un parallèle avec le JSON, son parseur est indépendant de son validateur de schéma (du projet JSON Schema (17)), il est facile de déterminer où sont les clés et valeurs en JSON grâce au nombreux séparateurs (`:` et `"`). En DY, le cœur du parseur est incapable de déterminer si une ligne commence par une clé ou non, comme un clé n'est pas distinguable des autres mots tant que ce mot n'est pas définie comme une clé.

C'est comme si on avait fusionné le code du parseur JSON et de son validateur, tout en ne donnant le schéma à valider qu'au moment de parser un document.

En DY, la spec DY est le schéma et se définit directement en Rust au lieu d'être dans un fichier texte comme pour les schémas JSON. Cette spec DY est un paramètre du cœur du parseur qui va mixer l'extraction du contenu et une partie de sa validation en se basant sur cette spec.

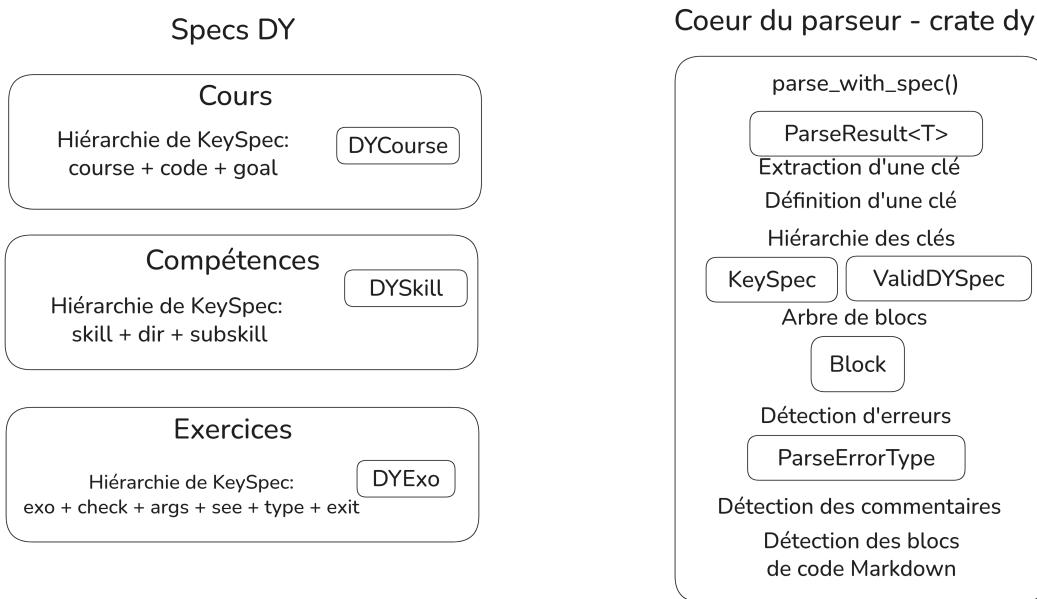


Figure 34 – Séparation claire entre spec DY et cœur du parseur

Lignes directrices de conception

Ces lignes directrices permettent de mieux comprendre certains choix de clés, de syntaxe ou de stratégie.

- Privilégier la facilité plutôt de rédaction, plutôt que la facilité d'extraction:** quand de nouveaux éléments syntaxiques sont ajoutés, l'optimisation de la rédaction est la priorité, face
- Pas de tabulations ni d'espace en début de ligne.** Cela introduit le fameux débat des espaces versus tabulations. En utilisant des espaces, le nombre d'espaces devient configurable. Les espaces complexifient un peu la collaboration comme les changements dans Git deviennent plus difficile à lire si deux enseignant·es n'utilisent pas les mêmes réglages.
- Une seule manière de définir un objet.** Au lieu d'ajouter différentes variantes pour arriver au même résultat juste pour satisfaire différents style, ne garder qu'une seule possibilité. Dès que des variantes sont introduites, cela complexifie le parseur et l'apprentissage. Pour garder un style commun, il faut discuter pour se mettre d'accord sur le style ou accepter de mixer les variantes dans un cours.
- Peu de caractères réservés, simple à taper et restreint à certains endroits.** Le moins de caractère réservés doivent être définis car cela pourra rentrer en conflit avec le contenu. Ils doivent toujours restreint à zones spécifiques du texte, pour que ces contraintes puissent être détournées si nécessaire.
- Pas de caractères en pairs.** Les caractères réservés ne doivent pas être `()`, `{}`, ou `[]` ou d'autres caractères qui vont toujours ensemble pour délimiter le début et la fin d'un élément. Surtout durant la retranscriptions, ces pairs requièrent des mouvements de curseur plus complexe, pour aller une fois au début et une fois à la fin de la zone délimitée.
- Utiliser des clés de préférence courtes.** Si une clé est utilisée très souvent et que son nom est long (plus de 5 lettres), il peut être judicieux de choisir une alternative plus courte. L'alternative peut être un surnom (`exercise -> exo`) ou le début (`directory -> dir`). Une clé devrait au minimum avoir deux lettres.
- Une erreur ne doit pas empêcher le reste de l'extraction.** Le parseur doit détecter les erreurs mais faire comme si elle n'était pas là. Une information manquante prend ainsi une valeur par défaut, pour permettre un usage ou aperçu limité à la place de perdre le reste de l'information.
- La struct Rust des objets extraits ne doit pas contraindre la structure de rédaction.** Par exemple, pour une struct `Exo` avec deux champs `name` et `instruction` (consigne), ne doit pas contraindre la rédaction à l'usage de deux clés séparées.

9. Le seul type natif autorisé est la string. Les projets qui intègre la syntaxe DY ne récupèrent que des strings en valeurs après les clés définies. Contrairement au YAML, TOML et d'autres, aucun type de dates, nombre entiers ou flottants n'est défini pour garder une base minimaliste et éviter toute ambiguïté d'interprétation, comme mentionné plusieurs fois dans l'état de l'art. Cela n'empêche pas que ces projets décident de supporter des nouveaux types et s'occupent eux-même du parsing et de la validation de ces valeurs.

Dans l'état actuel, la syntaxe DY n'a pas de tabulations ni d'espace, aucun caractère réservé (la fin de ligne et l'espace sont des séparateurs mais ne sont pas réservés à la syntaxe). Les clés réservées peuvent être échappées pour écrire le mot littéral.

Commentaires

Pour permettre de communiquer des informations supplémentaires durant la rédaction, les commentaires sont supportés et ne sont visibles que dans le fichier directement. Le parseur les ignore et ne se rappelle pas de leur position. Les commentaires ne peuvent être définis que sur une ligne dédiée, les 2 premiers caractères de la ligne doivent être `//` et le reste de la ligne est complètement libre. Le texte tel que `exo intro // c'est basique` n'est pas considéré comme un incluant commentaire, contrairement au C et d'autres langages.

Support du Markdown

Tous les champs supportent le Markdown, cela signifie que les snippets de code en Markdown sont considérées comme du contenu par notre parseur. La zone concernée commence par ````` ou `~~~`, elle est ignorée jusqu'à trouver le même marqueur. Ainsi, de potentielles clés ou commentaires présents ne sont pas considérés, ce qui permet de préserver les commentaires de code.

```
exo Parser la syntaxe DY

Faites en sorte d'extraire en C++ l'exercice suivant défini dans la syntaxe de PLX.

```
// A simple comment
exo print hello world
just in C++
check output is correct
see Hello, World !
```

check L'exercice a pu être parsé, la structure JSON est affichée
see { "name": "print hello world", "instruction": "just in C++",
"checks": [ { "name": "output is correct", "args": [], "exit": null,
"sequence": [ { "See": "Hello, World !" } ] } ] }
```

Figure 35 – Exercice avec une consigne qui contient un bloc de code contenant lui-même un autre exercice en DY.

Le contenu du bloc de code n'est pas interprété et la consigne se termine avant le `check` colorisé

Définition et contraintes des clés

Les clés sont tirés du concept de clé/valeur du JSON. Une clé est une string en minuscule, contenant uniquement des caractères alphabétiques. Elle doit se trouver tout au début d'une ligne sans espace. Si un caractère existe après la clé, il ne peut être que l'espace ou le retour à la ligne `\n`. Ainsi `course` ne contient pas la clé `course`. Les clés introduisent une valeur et parfois le début d'un objet si elles contiennent d'autres clés enfants.

Dans l'exemple de la Figure 36, les clés sont `course`, `code` et `goal`. La clé `course` introduit une valeur (le nom du court) et un objet (le cours) qui contient les valeurs tirées des clés enfants (`code`

et `goal`). Les types de valeurs introduites peuvent être soit sur une ligne ou multilignes. La clé `code` introduit la valeur `PRG1`, qui ne peut être définie que sur une ligne, car un code raccourci ne peut pas contenir de retour à la ligne. Tandis que la valeur de la clé `goal` peut s'étendre sur plusieurs lignes. Si une ligne suffit, la valeur est aussi valide.

```
course Programmation 1
code PRG1
goal Apprendre des bases solides du C++
parce que c'est vraiment *important* pour la suite du Bachelor !
```

Figure 36 – Exemple d'usage de clés et de leur hiérarchie avec un cours PLX

Si du contenu devait contenir un mot qui est aussi utilisé pour une clé il suffit de ne pas le placer au début d'une ligne. Ajouter un espace devant le mot suffit à respecter cette contrainte comme le démontre Figure 37. Cet espace supplémentaire n'aura pas d'impact sur l'affichage si le Markdown est interprété en HTML, puisque le rendu graphique d'un navigateur ignore les double espaces.

```
exo Fibonnaci suite

Print the Fibonnaci suite for 100 values. Make sure to
check the code is correctly formatted.

check ...
```

Figure 37 – Exemple

Note: les mentions de « clés parents » sur des clés à la racine, concerne le document lui-même.

Les clés, créées dans une spec DY en Rust à l'aide de la struct `KeySpec`, possèdent les attributs suivants

1. `id` : le texte de la clé (exemple `course`), qui doit être unique pour toute la spec DY
2. `desc` : Une description qui sert à documenter le but de la clé, qui sera utile pour la documentation au survol et l'autocomplétion pour le futur serveur de langage
3. `subkeys` : un vecteur de sous clés possibles, qui peut être vide.
4. `vt` : Un type de valeur, soit ligne simple soit multilignes, défini via l'enum `ValueType`.
5. `once` : champ booléen qui définit si la clé ne peut se retrouver qu'une seule fois dans chaque objet défini par la clé parent.
6. `required` : si la clé doit exister au moins une fois dans tout objet de la clé parent et si une valeur est requise pour la clé. Si ces contraintes ne sont pas respectées des erreurs `MissingRequiredKey` ou `MissingRequiredValue` sont générées.

Une valeur ne peut être que de type string (ce qui nous permet d'éviter les guillements ou certaines ambiguïtés). Elle commence après la clé et se termine dès qu'une autre clé valide est trouvée ou que la fin du fichier est atteint. Si le type de valeur (attribut `vt`) est une ligne simple, alors le contenu s'arrête à la fin de la ligne, les lignes suivantes seront ignorées. Si celles-ci ne sont pas vides, cela causera une erreur de `InvalidMultilineContent`.

Implémentation de la librairie dy

Dans la Figure 38, la vue d'ensemble des étapes haut niveaux est présentées. Avec le type généré le contenu est découpé en ligne,

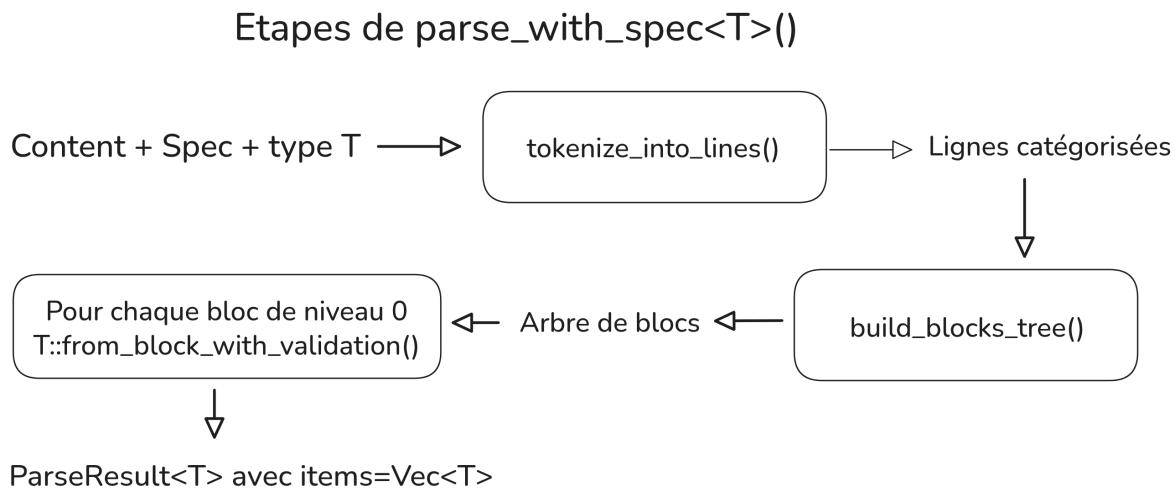


Figure 38 – Etapes haut-niveau du cœur du parseur, via son point d’entrée `parse_with_spec<T>`

```

pub struct ParseResult<T> {
    pub items: Vec<T>,
    pub errors: Vec<ParseError>,
    pub some_file_path: Option<String>,
    pub some_file_content: Option<String>,
}
  
```

Snippet 46 – La struct décrivant un vecteur de résultat de type générique, incluant les erreurs trouvées. Le nom du fichier, si disponible et contenu sont inclus pour permettre l'affichage des erreurs

```

pub trait FromDYBlock<'a> {
    fn from_block_with_validation(block: &Block<'a>) -> (Vec<ParseError>, Self);
}
  
```

Snippet 47 – Un trait (interface) `FromDYBlock` qui impose d’implémenter `from_block_with_validation`, utilisé sur chaque struct final associée à une spec DY

```

pub fn parse_with_spec<'a, T>(
    spec: &'a ValidDYSPEC,
    some_file: &Option<String>,
    content: &'a str,
) -> ParseResult<T>
where
    T: FromDYBlock<'a> {...}
  
```

Snippet 48 – La définition de `parse_with_spec`, avec la contrainte que `T` doit implémenter le trait `FromDYBlock`

Catégorisation des lignes

La première étape consiste à prendre le fichier brut, d’itérer sur chaque ligne et de catégoriser la ligne pour définir si elle contient une clé (`WithKey`) ou non (`Unknown`). Une liste de toutes les clés présente dans l’arbre `ValidDYSPEC` est extraite. Pour ne pas comparer à chaque fois toutes les clés, elles sont regroupées par longueur dans une `HashMap`. A chaque ligne, on extrait le premier mot et on regarde uniquement les clés de même longueur que ce mot.

```

exo Salue-moi
Un petit programme qui te salue avec ton nom complet.

check Il est possible d'être salué avec son nom complet
see Quel est ton prénom ?
type John
see Salut John, quel est ton nom de famille ?
type Doe
see Passe une belle journée John Doe !
exit 0

```

Figure 39 – Reprenons l'exercice Salue-moi en exemple

```

Line { index: 0, slice: "exo Salue-moi", lt: WithKey(KeySpec 'exo')},
Line { index: 1, slice: "Un petit programme qui te salue avec ton nom complet.", lt: Unknown},
Line { index: 2, slice: "", lt: Unknown},
Line { index: 3, slice: "check Il est possible d'être salué avec son nom complet", lt: WithKey(KeySpec 'check')},
Line { index: 4, slice: "see Quel est ton prénom ?", lt: WithKey(KeySpec 'see') },
Line { index: 5, slice: "type John", lt: WithKey(KeySpec 'type') },
Line { index: 6, slice: "see Salut John, quel est ton nom de famille ?", lt: WithKey(KeySpec 'see')},
Line { index: 7, slice: "type Doe", lt: WithKey(KeySpec 'type',) },
Line { index: 8, slice: "see Passe une belle journée John Doe !", lt: WithKey(KeySpec 'see')},
Line { index: 9, slice: "exit 0", lt: WithKey(KeySpec 'exit')}},

```

Snippet 49 – Liste de lignes avec un index, la référence vers le morceau de texte de la ligne, ainsi que type de ligne `lt`

Construction d'un arbre de blocs

Cet arbre représente la hiérarchie des clés et valeurs trouvées et respecte en tout temps la hiérarchie de `ValidDYSPEC`. Un bloc contient le texte extrait (sous forme de vecteur de lignes), la plage du contenu concerné (`Range`) et une référence vers la définition de la clé associée au bloc. Les erreurs rencontrées au fur et à mesure n'impacte pas cet arbre et sont insérées dans une liste d'erreurs séparées. Les commentaires ne sont pas inclus.

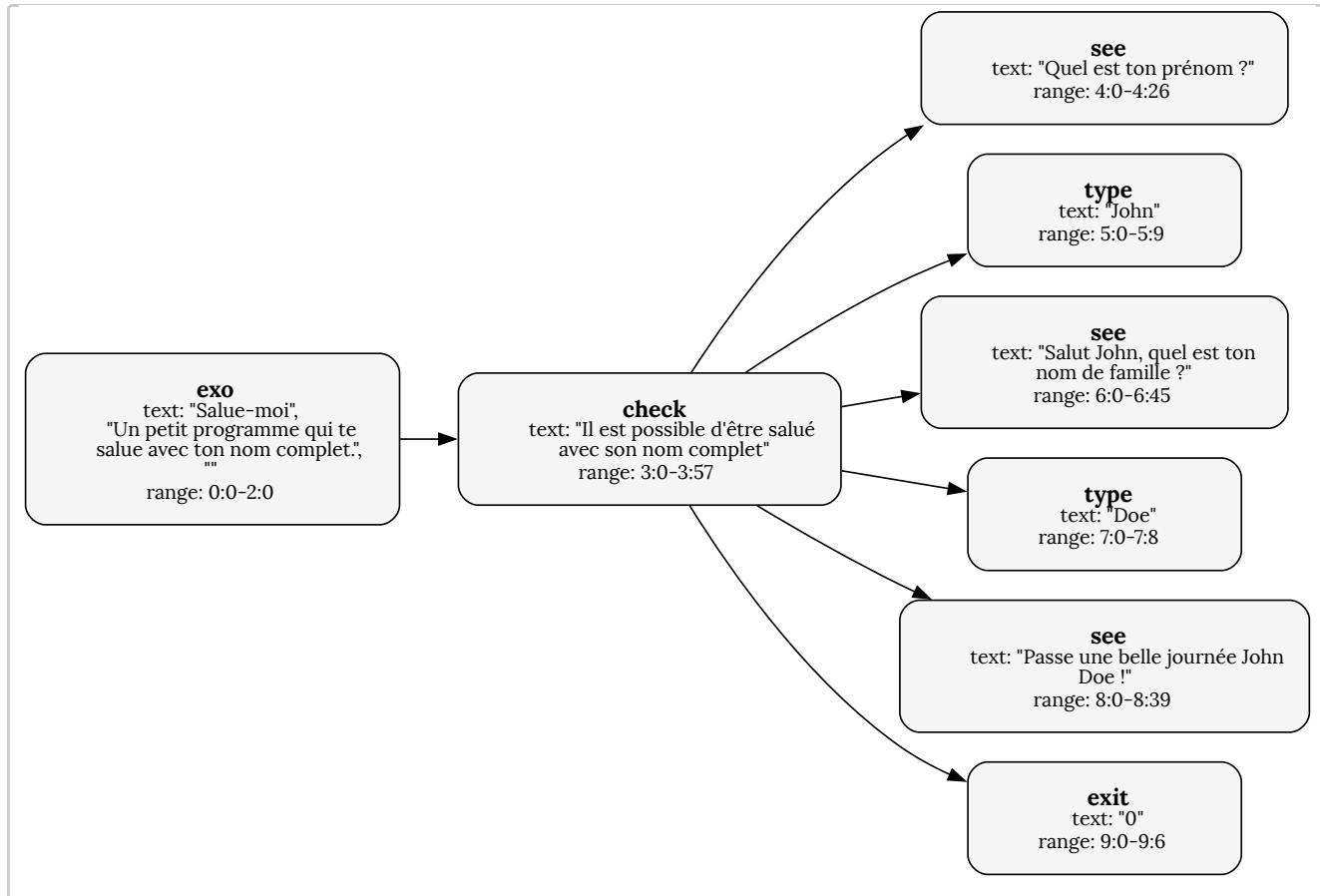


Figure 40 – Arbre de blocs généré à partir des lignes du Snippet 49, les blocs sont ordrés de haut en bas.

Conversion vers la struct T

Grâce à l'interface `FromDYBlock`, nous pouvons donner chaque bloc racine à la méthode `T::from_block_with_validation(&block)` afin qu'il puisse être converti en type `T` et générer d'autres erreurs si nécessaires. La liste d'erreur créé durant la construction de l'arbre de blocs

Pour finir l'exemple de



Snippet 50 –

Implémentation de `plx-dy`

Modèle de données de PLX et choix des clés

Pour que l'application desktop de PLX fonctionne, nous avons besoin de décrire un cours, divisé en compétences, qui regroupent des exercices. Un exercice définit un ou plusieurs checks. Voici une liste des informations associés à ces quatres objets.

- Un cours:** un nom (par exemple `Programmation 2`), un code (souvent il existe un raccourci du nom, comme `PRG2`) et une description de l'objectif du cours. Une liste de compétences.
- Une compétence:** un nom, une description et un ensemble d'exercices. Une compétence peut aussi être une sous compétence, pour subdiviser une grande compétence en sous compétences plus spécifiques.
- Un exercice:** un nom, une consigne et un ou plusieurs checks pour vérifier le comportement d'un programme.

4. **Un check:** un nom, des arguments à passer au programme, un code d'exit attendu et une séquence d'actions/assertions à lancer. Une action peut être ce qu'on tape au clavier et une assertion concerne la vérification que l'output est correct.

Nous avons ensuite défini la liste de clés et leur hiérarchie pour le modèle de données précédent, ainsi que les structs finales à remplir.

Un cours

- `course` est le nom du cours
 - `code` donne un raccourci du nom du cours
 - `goal` pour l'objectif du cours, sur plusieurs lignes

```
pub struct DYCourse {
    pub name: String,
    pub code: String,
    pub goal: String,
}
```

Snippet 51 – Simple struct Rust pour un cours

Une compétence

- `skill` définit un nom la même ligne et une description optionnel sur les lignes suivantes.
 - `dir` est le dossier dans lequel sont définis les exercices de cette compétences
 - `subskill`: une sous compétence, pour découper en compétences plus spécifiques

```
pub struct DYSkill {
    pub name: String,
    pub description: String,
    pub directory: String,
    pub subskills: Vec<DYSkill>,
}
```

Snippet 52 – Le nom et description sont séparés en deux champs pour les compétences

Un exercice

- `exo` définit un nom sur la même ligne et une consigne optionnel sur les lignes suivantes.
 - `check` introduit le début d'un check avec un titre
 - `args` définit les arguments du programme de l'exercice
 - `see` demande à voir une ou plusieurs lignes en sortie standard. L'entrée peut être sur plusieurs lignes.
 - `type` simule une entrée au clavier
 - `exit` définit le code d'exit attendu, valeur optionnelle

```
pub enum TermAction {
    See(String),
    Type(String),
}
pub struct Check {
    pub name: String,
    pub args: Vec<String>,
    pub exit: Option<i32>,
    pub sequence: Vec<TermAction>,
}
pub struct DYExo {
    pub name: String,
    pub instruction: String,
    pub checks: Vec<Check>,
}
```

Snippet 53 – Définition d'un exercice, avec des checks et la séquence d'action

Définition d'une hiérarchie de clés en Rust

Après avoir présenté les attributs de la struct `KeySpec`, voici un exemple concret de définition en Rust de spec DY. Nous avons défini sur le Snippet 54 une constante par clé, puis regroupés les clés `goal` et `code` en sous clés de `course`.

```

const GOAL_KEYSPEC: &KeySpec = &KeySpec {
    id: "goal",
    desc: "The goal key describes the learning goals of this course.",
    subkeys: &[],
    vt: ValueType::Multiline,
    once: true,
    required: true,
};

const CODE_KEYSPEC: &KeySpec = &KeySpec {
    id: "code",
    desc: "The code of the course is a shorter name of the course, under 10 letters usually.",
    subkeys: &[],
    vt: ValueType::SingleLine,
    once: true,
    required: true,
};

const COURSE_KEYSPEC: &KeySpec = &KeySpec {
    id: "course",
    desc: "A PLX course is grouping skills and exos related to a common set of learning goals.",
    subkeys: &[CODE_KEYSPEC, GOAL_KEYSPEC],
    vt: ValueType::SingleLine,
    once: true,
    required: true,
};

pub const COURSE_SPEC: &DYSpec = &[COURSE_KEYSPEC];

```

Snippet 54 – Exemple de définition en Rust de la spec DY des cours PLX, avec 3 KeySpec pour les 3 clés

Le tableau de `KeySpec` final (alias de type `DYSpec`) de toutes les clés autorisées à la racine, doit encore être validé. Un type wrapper `validDYSPEC` permet de valider que chaque id de clé est bien unique.

En suivant la même logique que précédemment, nous avons défini 3 hiérarchie de clés, avec les structures et contraintes de clés suivantes.

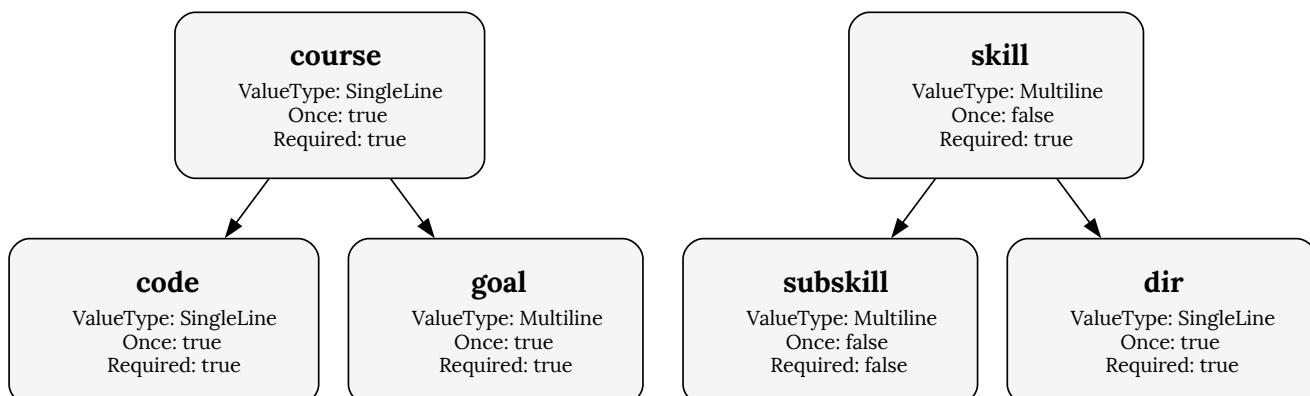


Figure 41 – Aperçu graphique de la spec DY des cours.

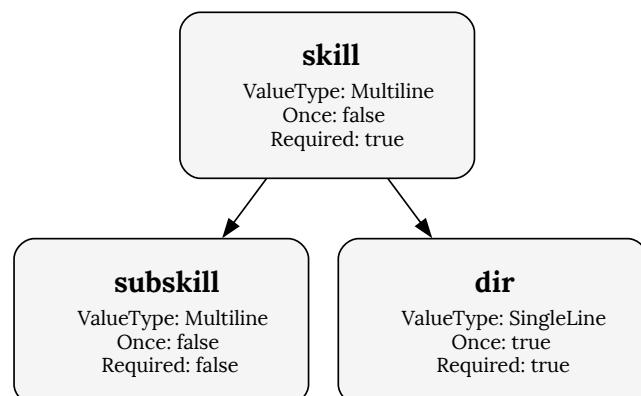


Figure 42 – Aperçu graphique de la spec DY des compétences.

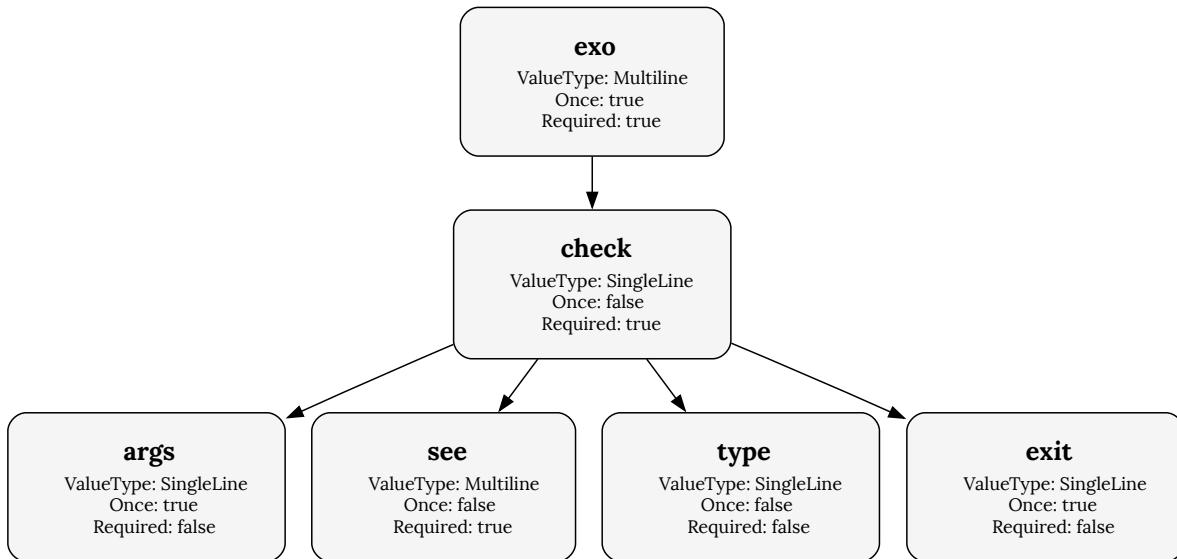


Figure 43 – Aperçu graphique de la spec DY des exercices PLX. La clé `exo`, `check` et `see` sont obligatoires (`required=true`). `args` et `exit` ne peut apparaître qu'une seule fois par `check` (`once=true`). Seuls `exo` et `see` peuvent être donnés sur plusieurs lignes (`ValueType=Multiline`).

Fonctions publiques

Chacune des 3 spécifications donne accès à une fonction haut niveau comme `parse_course`, `parse_skills` et `parse_exo`. Ces fonctions font simplement appel à `parse_with_spec`, comme le montre l'exemple en .

```

pub fn parse_course(some_file: &Option<String>, content: &str) -> ParseResult<DYCourse> {
    parse_with_spec::<DYCourse>(
        &ValidDYSPEC::new(COURSE_SPEC).expect("COURSE_SPEC is invalid !"),
        some_file,
        content,
    )
}
  
```

Snippet 55 – Exemple d'usage de `parse_with_spec` pour définir la fonction `parse_course` dans la spec DY des cours

Intégration de `plx-dy`

Structures de fichiers DY

La crate `plx-dy` définit des constantes pour des fichiers dans lesquels se trouvent les définitions de nos 3 objets. Le cours est décrit dans `course.dy`, les compétences dans `skills.dy` et chaque exercice dans un fichier `exo.dy` dans son dossier à côté du code et de ses solutions. Le fichier `live.toml` est attendue à la racine également.

```
plx-demo> tree
.
├── README.md
├── course.dy
├── skills.dy
└── live.toml
└── intro
    ├── basic-args
    │   ├── exo.dy
    │   ├── main.c
    │   └── main.sol.c
    ├── salut-moi
    │   ├── debug.log
    │   ├── exo.dy
    │   └── main.c
    └── structs
        └── meeting-participants
            ├── exo.dy
            ├── main.cpp
            └── main.sol.cpp
```

Snippet 56 – Exemple de structure de fichiers pour un cours PLX de démonstration, avec 2 compétences et 3 exercices.

TODO fix headings level

TODO

Intégration au CLI

La première intégration a été faite dans le CLI (qui permet aussi de démarrer le serveur, pour rappel). Ce CLI est utile pour que les enseignant·es puissent vérifier que le contenu du contenu d'un cours PLX est valide. Dans le futur, il pourrait aussi servir à d'autres outils qui pourraient réutiliser le JSON généré, par exemple pour insérer les exercices dans une base de données.

```
> plx parse -h
Parse the given DY file or parse the course.dy inside given folder

Usage: plx parse [OPTIONS] <PATH>

Arguments:
<PATH> A PLX file with a .dy extension, or a folder with a course.dy

Options:
--full  Enable the full course parsing in PLX's format. Only valid with a folder
```

Snippet 57 – Aide de la sous commande `plx parse`

```
// Définition du cours
course Programmation 1
code PRG1
goal Apprendre des bases solides du C++
```

Figure 44 – Définition d'un cours PLX dans un fichier `course.dy`

```
> plx parse course/course.dy
Found 1 items in course/course.dy with no error!
{
  "name": "Programmation 1",
  "code": "PRG1",
  "goal": "Apprendre des bases solides du C++"
}
```

Figure 45 – Équivalent extrait du parseur et affiché en JSON

```
skill Introduction
dir intro
skill Enumerations
dir enums
skill Structures
dir structs
skill Pointers and memory
dir pointers
skill Parsing
dir parsing
```

Figure 46 – TODO skills.dy

```
> plx parse skills/skills.dy
Found 5 items in skills/skills.dy with no error!
[
  {
    "name": "Introduction",
    "description": "",
    "directory": "intro",
    "subskills": []
  },
  {
    "name": "Enumerations",
    "description": "",
    "directory": "enums",
    "subskills": []
  },
  {
    "name": "Structures",
    "description": "",
    "directory": "structs",
    "subskills": []
  },
  {
    "name": "Pointers and memory",
    "description": "",
    "directory": "pointers",
    "subskills": []
  },
  {
    "name": "Parsing",
    "description": "",
    "directory": "parsing",
    "subskills": []
  }
]
```

Figure 47 – TODO

```

exo Salue-moi
Un petit programme qui te salue avec ton nom complet.

check Il est possible d'être salué avec son nom complet
see Quel est ton prénom ?
type John
see Salut John, quel est ton nom de famille ?
type Doe
see Passe une belle journée John Doe !
exit 0

```

Figure 48 – TODO `exo.dy`

```

> plx parse exo/exo.dy
Found 1 items in exo/exo.dy with no error!
{
  "name": "Salue-moi",
  "instruction": "Un petit programme qui te salue avec ton nom complet.",
  "checks": [
    {
      "name": "Il est possible d'être salué avec son nom complet",
      "args": [],
      "exit": 0,
      "sequence": [
        {
          "See": "Quel est ton prénom ?"
        },
        {
          "Type": "John"
        },
        {
          "See": "Salut John, quel est ton nom de famille ?"
        },
        {
          "Type": "Doe"
        },
        {
          "See": "Passe une belle journée John Doe !"
        }
      ]
    }
  ]
}

```

Figure 49 – TODO

Détection d'erreurs

```

// Pas tout à fait le bon ordre
code PRG1
course Programmation 1

```

Figure 50 – Définition incorrecte d'un cours PLX dans un fichier `course.dy`.

Le `goal` manque et le `code` doit être placé après la clé `course`.

```
> plx parse course-error/course.dy
Found 1 item in course-error/course.dy with 3 errors.

Error at course-error/course.dy:1:0
code PRG1
^^^^ The 'code' key can be only used under a `??`

Error at course-error/course.dy:2:0
course Programmation 1
| Missing required key 'goal'

Error at course-error/course.dy:2:0
course Programmation 1
| Missing required key 'code'
```

Figure 51 – Les erreurs ont été détectées par le parseur

```
skill Introduction
skill Enumerations
dir enums
enums/test
skill
dir parsing
```

Figure 52 – TODO skills.dy

```
> plx parse skills-error/skills.dy
Found 3 items in skills-error/skills.dy with 3 errors.

Error at skills-error/skills.dy:0:0
skill Introduction
| Missing required key 'dir'

Error at skills-error/skills.dy:3:0
enums/test
^^^^^^^^^ Invalid multiline content found after the 'dir' key which is single line

Error at skills-error/skills.dy:4:5
skill
| Missing a value for the required key 'skill'
```

Figure 53 – TODO

```
exo
Exo tout cassé...

check Salue la personne donnée en argument
args John Doe
args John
// see Quel âge as-tu ?
exit one
```

Figure 54 – TODO exo.dy

```
> plx parse exo-error/exo.dy
Found 1 item in exo-error/exo.dy with 3 errors.

Error at exo-error/exo.dy:3:0
check Salue la personne donnée en argument
| Missing required key 'see'

Error at exo-error/exo.dy:5:0
args John
^^^^ The 'args' key can only be used once at this level

Error at exo-error/exo.dy:7:5
exit one
    ^^^ Couldn't parse the given value as the program's exit code (signed 32bits integer)
```

Figure 55 – TODO

Tests unitaires

```
> cargo test
running 22 tests
test parser::tests::test_line_into_parts ... ok
test semantic::tests::test_can_build_blocks_for_complex_skills ... ok
test semantic::tests::test_can_detect_content_out_of_key ... ok
test semantic::tests::test_can_build_blocks_with_multiline_keys_ignoring_comments ... ok
test parser::tests::test_can_tokenize_comments_and_empty ... ok
test semantic::tests::test_can_detect_duplicated_key_error ... ok
test parser::tests::test_can_tokenize_and_ignore_anything_inside_code_blocks ... ok
test semantic::tests::test_can_detect_invalid_multiline_content ... ok
test parser::tests::test_line_starts_with_key ... ok
test semantic::tests::test_can_detect_wrong_key_positions ... ok
test semantic::tests::test_required_key_also_work_at_root ... ok
test semantic::tests::test_missing_keys_and_values_with_required_keys_are_detected ... ok
test semantic::tests::test_empty_lines_are_present_in_block_text ... ok
test semantic::tests::test_can_extract_complex_exos_blocks_with_errors_ignorance ... ok
test semantic::tests::test_strange_exo_parsing_can_correctly_ignore_error ... ok
test parser::tests::test_can_tokenize_lines_with_invalid_keys_and_empty_lines ... ok
test parser::tests::test_can_tokenize_basic_lines ... ok
test semantic::tests::test_can_build_blocks_for_simple_course ... ok
test spec::tests::test_can_validate_valid_spec ... ok
test spec::tests::test_empty_spec_is_invalid ... ok
test spec::tests::test_spec_with_duplicated_key_at_root ... ok
test spec::tests::test_spec_with_duplicated_key_deeply ... ok

running 10 tests
test course::tests::test_can_parse_simple_valid_course ... ok
test course::tests::test_parse_result_display_is_also_correct ... ok
test course::tests::test_parse_result_display_can_highlight_unknown_content ... ok
test course::tests::test_parse_result_display_is_correct ... ok
test exo::tests::test_can_extract_args_by_space_split ... ok
test exo::tests::test_can_error_on_invalid_exit_code ... ok
test exo::tests::test_detect_empty_args_error_but_ignores_empty_type ... ok
test skill::tests::test_can_detect_subskill_missing_value ... ok
test exo::tests::test_can_parse_a_simple_exo ... ok
test skill::tests::test_can_parse_simple_skills ... ok
```

[Snippet 58](#) – Aperçu des 32 tests unitaires développés pour les crates `dy` et `plx-dy`

Pour mieux se représenter à quoi ressemble ces tests. Ce test `test_can_error_on_invalid_exit_code` petit test pour s'assurer que l'implémentation de `FromDYBlock` pour `DYExo` détecte bien l'erreur d'un code d'exit non numérique et qu'il prend sa valeur par défaut (`None`). On s'assure aussi que le reste de l'exercice est extrait correctement (titre, check, see).

```

#[test]
fn test_can_error_on_invalid_exit_code() {
    let text = "exo thing
check test
see hello
exit blabla
";
    let some_file = &Some("exo.dy".to_string());
    assert_eq!(
        parse_exos(some_file, text),
        ParseResult {
            some_file_path: some_file.clone(),
            some_file_content: Some(text.to_string()),
            items: vec![DYExo {
                name: "thing".to_string(),
                instruction: "".to_string(),
                checks: vec![Check {
                    name: "test".to_string(),
                    args: vec![],
                    exit: None,
                    sequence: vec![TermAction::See("hello".to_string())],
                }]
            }],
            errors: vec![ParseError {
                range: range_on_line_part(3, 5, 11),
                error: ParseErrorType::ValidationErrors(
                    ERROR_CANNOT_PARSE_EXIT_CODE.to_string()
                )
            }]
        }
    )
}

```

Snippet 59 – Aperçu des 32 tests unitaires développés pour les crates `dy` et `plx-dy`



Snippet 60 – Aperçu des 32 tests unitaires développés pour les crates `dy` et `plx-dy`

Conclusion

Ce rendu intermédiaire termine ainsi les recherches et la rédaction sur l'état de l'art, de nombreuses technologies ont été parcourue. Pour le choix des différentes librairies, une approche basée sur la réduction de la complexité a été privilégiée. Des POC ont été développé pour mieux comprendre et prouver le fonctionnement de Tree-Sitter, la crate `lsp-server` et l'usage de WebSocket en Rust via la crate `tungstenite` ainsi que l'envoi des messages en JSON.

TODO

Bibliographie

Avertissement: le format de cette bibliographie n'est pas encore tout à fait correct, notamment sur la gestion des auteurs et des contributeurs. Il manque certains nom d'auteurs ou dates de consultation. Cela sera corrigé par la suite avant la rendu final.

1. K. Anders Ericsson - Wikipedia. En ligne. 2025. Disponible à l'adresse: https://en.wikipedia.org/wiki/K._Anders_Ericsson
2. ERICSSON, Anders et POOL, Robert. International Edition. Penguin Canada, 2017.
3. PLX. PLX website. En ligne. 2025. Disponible à l'adresse: <https://plx.rs/>
4. Vue.js - The Progressive JavaScript Framework. En ligne. 2025. Disponible à l'adresse: <https://vuejs.org/>
5. rustlings. En ligne. 2025. Disponible à l'adresse: <https://rustlings.rust-lang.org/>
6. HEIG-VD. GitHub - PRG2-HEIGVD/PRG2_Recueil_Exercices at 1027428583bdfa68b6638d90eaca215ec0647317. En ligne. 22 mai 2025. Disponible à l'adresse: https://github.com/PRG2-HEIGVD/PRG2_Recueil_Exercices/tree/1027428583bdfa68b6638d90eaca215ec0647317
7. Typst: Compose papers faster. En ligne. 2025. Disponible à l'adresse: <https://typst.app/>
8. HEIG-VD. PRG2_Recueil_Exercices/C5_Chaines_de_caracteres/05-03-Opérations_sur_les_buffers.md at 1027428583bdfa68b6638d90eaca215ec0647317. En ligne. 19 mars 2024. Disponible à l'adresse: https://github.com/PRG2-HEIGVD/PRG2_Recueil_Exercices/blob/1027428583bdfa68b6638d90eaca215ec0647317/C5_Chaines_de_caracteres/05-03-Op%C3%A9rations_sur_les_buffers.md
9. CROCKFORD, Douglas et BRAY, Tim. RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format - Section 1 Introduction. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc7159.html#section-1>
10. ????. Home - Neovim. En ligne. 2025. Disponible à l'adresse: <https://neovim.io/>
11. GitHub - danwritecode/clings: rustlings for C....clings. En ligne. 2025. Disponible à l'adresse: <https://github.com/danwritecode/clings>
12. CONTRIBUTEURS. GitHub - mauricioabreu/golings": " rustlings but for golang this time. En ligne. 2025. Disponible à l'adresse: <https://github.com/mauricioabreu/golings>
13. ziglings/exercises: Learn the ⚡ Zig programming language by fixing tiny broken programs. - Codeberg.org. En ligne. 2025. Disponible à l'adresse: <https://codeberg.org/ziglings/exercises>
14. GitHub - MondayMorningHaskell/haskellings: An automated tutorial to teach you about Haskell!. En ligne. 2025. Disponible à l'adresse: <https://github.com/MondayMorningHaskell/haskellings>
15. HORSTMANN, Cay. CodeCheck. En ligne. 2025. Disponible à l'adresse: <https://horstmann.com/codecheck/>
16. HORSTMANN, Cay. AverageTester.java exercice on CodeCheck. En ligne. 2025. Disponible à l'adresse: <https://codecheck.io/files/wiley/codecheck-bj-4-object-102>
17. JSON Schema - Build more. Break less. Empower others. En ligne. 2025. Disponible à l'adresse: <https://json-schema.org/>
18. DUBOVSKOY, Alexey. Cooklang - Recipe Markup Language. En ligne. 2025. Disponible à l'adresse: <https://cooklang.org/>
19. TORM. udl v0.3.1 - Parser for UDL (Universal Data Language). En ligne. 2023. Disponible à l'adresse: <https://crates.io/crates/udl>
20. TORM. The Khi data language. En ligne. 2024. Disponible à l'adresse: <https://github.com/khilang/khi>
21. TORM. Rust Khi parser & library. En ligne. 2024. Disponible à l'adresse: <https://github.com/khilang/khi.rs>
22. About and Contact - bitmark Documentation - License. En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/about/#license>

23. bitmark Association website. En ligne. 2025. Disponible à l'adresse: <https://www.bitmark-association.org/>
24. BITMARK ASSOCIATION. bitmark Hackathon. En ligne. 2025. Disponible à l'adresse: <https://www.bitmark-association.org/bitmarkhackathon>
25. ASSOCIATION. bitmark Documentation. En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/>
26. BITMARK ASSOCIATION. Quizzes - .multiple-choice, .multiple-choice-1. En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/quizzes/#multiple-choice-multiple-choice-1>
27. TASKBASE. open-taskpool - 12,000 UK 🇺🇦 → DE 🇩🇪 & DE 🇩🇪 → EN 🇬🇧 learning tasks ready for you to use. En ligne. 2025. Disponible à l'adresse: <https://github.com/taskbase/open-taskpool>
28. BITMARK ASSOCIATION. Quizzes - .cloze (gap text). En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/quizzes/#cloze-gap-text>
29. TODO. How bitmark helps Classtime to import and export learning content efficiently. En ligne. 4 avril 2022. Disponible à l'adresse: <https://youtu.be/XegSepmkSnU?t=96>
30. CLASSTIME. Créer la première question / le premier jeu de questions. En ligne. 2024. Disponible à l'adresse: <https://help.classtime.com/fr/comment-commencer-a-utiliser-classtime/creer-la-premiere-question-le-premier-jeu-de-questions>
31. KUNDERT, Ken et KUNDERT, Kale. NestedText – A Human Friendly Data Format. En ligne. 2025. Disponible à l'adresse: <https://github.com/KenKundert/nestedtext>
32. KUNDERT, Ken et KUNDERT, Kale. NestedText – A Human Friendly Data Format – Structured Code. En ligne. 2025. Disponible à l'adresse: <https://nestedtext.org/en/latest/#structured-code>
33. KUNDERT, Ken et KUNDERT, Kale. NestedText documentation – Schemas. En ligne. 2025. Disponible à l'adresse: <https://nestedtext.org/en/latest/schemas.html>
34. BOB22Z. docs.rs – Crate nestedtext. En ligne. 2025. Disponible à l'adresse: <https://nestedtext.latest/nestedtext/>
35. LUDWIG, Sönke. SDLang, Simple Declarative Language. En ligne. 2025. Disponible à l'adresse: <https://sdlang.org/>
36. LUDWIG, Sönke. Language Guide - String Literals. En ligne. 2020. Disponible à l'adresse: <https://github.com/dlang-community/SDLang-D/wiki/Language-Guide#string-literals>
37. KAT MARCHÁN (ZKAT), et contributeurs. KDL, a cuddly document language. En ligne. 2025. Disponible à l'adresse: <https://kdl.dev/>
38. Pkl :: Pkl Docs. En ligne. 2025. Disponible à l'adresse: <https://pkl-lang.org/>
39. All Crates for keyword 'parser'. En ligne. 2025. Disponible à l'adresse: <https://crates.io/keywords/parser>
40. COUPRIE, Geoffroy (Geal). nom v8.0.0 A byte-oriented, zero-copy, parser combinators library. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/nom>
41. Reverse dependencies of nom crate. En ligne. 2025. Disponible à l'adresse: https://crates.io/crates/nom/reverse_dependencies
42. PAGE, Ed (epage). winnow v0.7.8 A byte-oriented, zero-copy, parser combinators library. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/winnow>
43. Dependencies of kdl crate. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/kdl/6.3.4/dependencies>
44. pest v2.8.0 The Elegant Parser. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/pest>
45. (MARWES), Markus Westerlind. combine v4.6.7 Fast parser combinators on arbitrary streams with zero-copy support. En ligne. 2024. Disponible à l'adresse: <https://crates.io/crates/combine>
46. JOSHUA BARRETTO (ZESTERER), Rune Tynan (CraftSpider), et contributeurs. chumsky v0.10.1 A parser library for humans with powerful error recovery. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/chumsky>
47. Overview - Serde. En ligne. 2025. Disponible à l'adresse: <https://serde.rs/>
48. Most popular Rust libraries. En ligne. 2025. Disponible à l'adresse: <https://lib.rs/std>
49. Serde data model. En ligne. 2025. Disponible à l'adresse: <https://serde.rs/data-model.html>
50. MICROSOFT, et contributeurs. Language Server Protocol. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/>
51. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Capabilities. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities>

52. JSON-RPC WORKING GROUP. JSON-RPC 2.0 Specification. En ligne. 2013. Disponible à l'adresse: <https://www.jsonrpc.org/specification>
53. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Content part. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#contentPart>
54. BERGERCOOKIE, et contributeurs. asm-lsp v0.10.0 Language Server for x86/x86_64, ARM, RISCV, and z80 Assembly Code. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/asm-lsp>
55. ORGANISATION, et contributeurs eclipse-jdtls. GitHub - eclipse-jdtls/eclipse.jdt.ls: Java language server. En ligne. 2025. Disponible à l'adresse: <https://github.com/eclipse-jdtls/eclipse.jdt.ls>
56. TAILWINDLABS, et contributeurs. GitHub - tailwindlabs/tailwindcss-intellisense: Intelligent Tailwind CSS tooling for Visual Studio Code. En ligne. 2025. Disponible à l'adresse: <https://github.com/tailwindlabs/tailwindcss-intellisense>
57. ORGANISATION, et contributeurs typescript-language-server. GitHub - typescript-language-server/typescript-language-server: TypeScript & JavaScript Language Server. En ligne. 2025. Disponible à l'adresse: <https://github.com/typescript-language-server/typescript-language-server>
58. MICROSOFT, et contributeurs. Implementations - Tools supporting the LSP. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/implementors/tools/>
59. MICROSOFT, et contributeurs. Implementations - Language Servers. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/implementors/servers/>
60. OXALICA, et contributeurs. async-lsp v0.2.2 Asynchronous Language Server Protocol (LSP) framework based on tower. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/async-lsp>
61. OXALICA, et contributeurs. nil/crates/nil/Cargo.toml - NIx Language server, an incremental analysis assistant for writing in Nix. En ligne. 2025. Disponible à l'adresse: <https://github.com/oxalica/nil/blob/577d160da311cc7f5042038456a0713e9863d09e/crates/nil/Cargo.toml#L11>
62. MYRIAD-DREAMIN, contributeurs. sync-ls - Synchronized language service inspired by async-lsp, primarily for tinyminst. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/sync-ls>
63. ORGANISATION, et contributeurs tower-lsp-community. tower-lsp-server v0.21.1 Language Server Protocol implementation based on Tower. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/tower-lsp-server>
64. lsp-types v0.97.0 Types for interaction with a language server, using VSCode's Language Server Protocol. En ligne. 2024. Disponible à l'adresse: <https://crates.io/crates/lsp-types>
65. ORGANISATION GLUON-LANG, et contributeurs. Reverse dependencies of lsp-types crate. En ligne. 2024. Disponible à l'adresse: https://crates.io/crates/lsp-types/reverse_dependencies
66. ORGANISATION, et contributeurs rust-lang. Reverse dependencies of lsp-server crate. En ligne. 2024. Disponible à l'adresse: https://crates.io/crates/lsp-server/reverse_dependencies
67. EYAL KALDERON, et contributeurs. Reverse dependencies of tower-lsp crate. En ligne. 2023. Disponible à l'adresse: https://crates.io/crates/tower-lsp/reverse_dependencies
68. ORGANISATION RUST-LANG, ET CONTRIBUTEURS. rust-analyzer/lib/lsp-server/examples/goto_def.rs at master · rust-lang/rust-analyzer · GitHub. En ligne. 2025. Disponible à l'adresse: https://github.com/rust-lang/rust-analyzer/blob/master/lib/lsp-server/examples/goto_def.rs
69. En ligne. 2025. Disponible à l'adresse: https://macromates.com/manual/en/regular_expressions
70. En ligne. 2025. Disponible à l'adresse: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>
71. En ligne. 2025. Disponible à l'adresse: <https://www.jetbrains.com/help/idea/textmate.html>
72. LTD, MacroMates. Language Grammars – TextMate 1.x Manual - Example Grammar. En ligne. Disponible à l'adresse: https://macromates.com/manual/en/language_grammars#example_grammar
73. CONTRIBUTEURS DE TREE-SITTER. Introduction - Tree-sitter. En ligne. 2025. Disponible à l'adresse: <https://treesitter.github.io/tree-sitter/>
74. Neovim Documentation - Treesitter. En ligne. 2025. Disponible à l'adresse: <https://neovim.io/doc/user/treesitter.html>
75. Language Extensions - Grammar. En ligne. 2025. Disponible à l'adresse: <https://zed.dev/docs/extensions/languages?#grammar>
76. Creating a Grammar. En ligne. 2025. Disponible à l'adresse: <https://flight-manual.atom-editor.cc/hacking-atom/sections/creating-a-grammar/>

77. GITHUB, et contributeurs. Navigating code on GitHub. En ligne. 2025. Disponible à l'adresse: <https://docs.github.com/en/repositories/working-with-files/using-files/navigating-code-on-github>
78. CONTRIBUTEURS DE TREE-SITTER. Creating Parsers - Getting Started - Tree-sitter. En ligne. 2025. Disponible à l'adresse: <https://tree-sitter.github.io/tree-sitter/creating-parsers/1-getting-started.html>
79. MICROSOFT, et contributeurs. Semantic Highlight Guide | Visual Studio Code Extension API. En ligne. 2025. Disponible à l'adresse: <https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide>
80. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Semantic Tokens. En ligne. 2025. Disponible à l'adresse: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_semanticTokens
81. SUBLIMEHQ. SublimeHQ - End User License Agreement. En ligne. 2025. Disponible à l'adresse: <https://www.sublimehq.com/eula>
82. SUBLIMEHQ. Syntax Definitions. En ligne. 2025. Disponible à l'adresse: <https://www.sublimetext.com/docs/syntax.html>
83. STACK EXCHANGE INC. Technology | 2024 Stack Overflow Developer Survey - Integrated development environment. En ligne. 2025. Disponible à l'adresse: <https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment>
84. MICROSOFT, et contributeurs. Iteration Plan for March 2025 · Issue #243015 · microsoft/vscode · GitHub. En ligne. 2025. Disponible à l'adresse: <https://github.com/microsoft/vscode/issues/243015>
85. MICROSOFT, et contributeurs. Iteration Plan for May 2025 · Issue #248627 · microsoft/vscode · GitHub. En ligne. 2025. Disponible à l'adresse: <https://github.com/microsoft/vscode/issues/248627>
86. MICROSOFT, et contributeurs. Explore using tree sitter for syntax highlighting · Issue #210475 · microsoft/vscode · GitHub. En ligne. 2025. Disponible à l'adresse: <https://github.com/microsoft/vscode/issues/210475>
87. MICROSOFT, et contributeurs. [Exploration] Tree-sitter tokenization exploration (Fixes #161256) by aiday-mar · Pull Request #161479 · microsoft/vscode · GitHub. En ligne. 2022. Disponible à l'adresse: <https://github.com/microsoft/vscode/pull/161479>
88. March 2025 (version 1.99) - Tree-Sitter based syntax highlighting (Preview). En ligne. mars 2025. Disponible à l'adresse: https://code.visualstudio.com/updates/v1_99#_treesitter-based-syntax-highlighting-preview
89. CONTRIBUTEURS DE TREE-SITTER. The Grammar DSL - Tree-sitter. En ligne. 2025. Disponible à l'adresse: <https://tree-sitter.github.io/tree-sitter/creating-parsers/2-the-grammar-dsl.html>
90. SIRAPHOB, Ben. How to write a tree-sitter grammar in an afternoon | siraben's musings. En ligne. 2022. Disponible à l'adresse: <https://siraben.dev/2022/03/01/tree-sitter.html>
91. GitHub - AlecGhost/tree-sitter-vscode: Bring the power of Tree-sitter to VSCode. En ligne. 2025. Disponible à l'adresse: <https://github.com/AlecGhost/tree-sitter-vscode>
92. serde_json - Parsing JSON as strongly typed data structures. En ligne. 2025. Disponible à l'adresse: https://docs.rs/serde_json/latest/serde_json/index.html#parsing-json-as-strongly-typed-data-structures
93. serde_json - Constructing JSON values. En ligne. 2025. Disponible à l'adresse: https://docs.rs/serde_json/latest/serde_json/index.html#constructing-json-values
94. FETTE, Ian et MELNIKOV, Alexey. RFC 6455: The WebSocket Protocol. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc6455>
95. FETTE, Ian et MELNIKOV, Alexey. RFC 6455: The WebSocket Protocol - 1.5. Design Philosophy. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc6455#section-1.5>
96. CONTRIBUTEURS, Snapview GmbH et. Lightweight stream-based WebSocket implementation. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/tungstenite>
97. CONTRIBUTEURS. GitHub - crossbario/autobahn-testsuite: Autobahn WebSocket protocol testsuite. En ligne. 2025. Disponible à l'adresse: <https://github.com/crossbario/autobahn-testsuite>
98. CONTRIBUTEURS, Snapview GmbH et. tokio-tungstenite. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/tokio-tungstenite>
99. CONTRIBUTEURS. websocket. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/websocket>
100. CONTRIBUTEURS. fastwebsockets. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/fastwebsockets>
101. GOOGLE, et contributeurs. Protocol Buffers Documentation. En ligne. 2025. Disponible à l'adresse: <https://protobuf.dev/>

102. CONTRIBUTEURS. GitHub - tokio-rs/prost: PROST! a Protocol Buffers implementation for the Rust Language. En ligne. 2025. Disponible à l'adresse: <https://github.com/tokio-rs/prost>
103. AUTHORS. gRPC - A high performance, open source universal RPC framework. En ligne. 2025. Disponible à l'adresse: <https://grpc.io/>
104. BRANDHORST, Johan. The state of gRPC in the browser | gRPC. En ligne. 2019. Disponible à l'adresse: <https://grpc.io/blog/state-of-grpc-web>
105. GOOGLE. GitHub - google/tarpc: An RPC framework for Rust with a focus on ease of use. En ligne. 2025. Disponible à l'adresse: <https://github.com/google/tarpc>
106. CONTRIBUTEURS, Benoist Claassen et. MessagePack: It's like JSON. but fast and small. En ligne. 2025. Disponible à l'adresse: <https://msgpack.org/>
107. Cap'n Proto: Introduction. En ligne. 2025. Disponible à l'adresse: <https://capnproto.org/>
108. Apache Thrift - Home. En ligne. 2025. Disponible à l'adresse: <https://thrift.apache.org/>
109. DAVIS, Kyzer, PEABODY, Brad et LEACH, Paul. RFC 9562: Universally Unique IDentifiers (UUIDs) - 5.4. UUID Version 4. En ligne. 2025. Disponible à l'adresse: <https://www.ietf.org/rfc/rfc9562.html#name-uuid-version-4>
110. IANA. Service Name and Transport Protocol Port Number Registry. En ligne. 18 juin 2025. Disponible à l'adresse: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=unassigned&page=5>
111. PRESTON-WERNER, Tom. Semantic Versioning 2.0.0 | Semantic Versioning. En ligne. 2025. Disponible à l'adresse: <https://semver.org/>
112. Tauri 2.0 | Create small, fast, secure, cross-platform applications. En ligne. 2025. Disponible à l'adresse: <https://tauri.app/>
113. Build cross-platform desktop apps with JavaScript, HTML, and CSS. En ligne. 2025. Disponible à l'adresse: <https://www.electronjs.org/>
114. Inter-Process Communication | Tauri - Commands. En ligne. 2025. Disponible à l'adresse: <https://tauri.app/concept/inter-process-communication/#commands>
115. BEAUMONT, Oscar et ALLAN, Brendan. tauri-specta - Completely typesafe Tauri commands. En ligne. mai 2023. Disponible à l'adresse: <https://crates.io/crates/tauri-specta>
116. Overview - The Typeshare Book. En ligne. octobre 2024. Disponible à l'adresse: <https://1password.github.io/typeshare/>
117. tokio::task - Asynchronous green-threads. En ligne. 2025. Disponible à l'adresse: <https://docs.rs/tokio/latest/tokio/task/>
118. LEARNEAO. Free AI Grammar Checker - LanguageTool. En ligne. 2025. Disponible à l'adresse: <https://languagetool.org/>
119. GitHub - DACC4/HEIG-VD-typst-template-for-TB: This template is a typst version of a LaTeX template for the travail de bachelors (TB) used at the HEIG-VD. En ligne. 2025. Disponible à l'adresse: <https://github.com/DACC4/HEIG-VD-typst-template-for-TB>
120. JONASLOOS. BibTeX to Hayagriva. En ligne. 2025. Disponible à l'adresse: <https://jonasloos.github.io/bibtex-to-hayagriva-webapp/>
121. PLX. Development - PLX docs - Logo design. En ligne. 2025. Disponible à l'adresse: <https://plx.rs/book/dev.html#logo-design>

Figures

Fig. 1	Résumé visuel du temps estimé passé sur l'exercice par un étudiant débutant	8
Fig. 2	Comparaison du temps nécessaire estimé sans et avec PLX	10
Fig. 3	Dans PLX, l'aperçu des listes de compétences et exercices dans un cours fictif, il est possible de parcourir les exercices et d'en démarrer un	11
Fig. 4	Une fois cet exercice de C lancé, le titre et la consigne sont visibles. Les erreurs de compilation sont directement affichés dans PLX, en préservant les couleurs	11
Fig. 5	2 checks qui échouent, avec la différence d'output pour facilement comprendre ce qui n'est pas correcte. L'IDE s'est ouvert automatiquement en parallèle.	12
Fig. 6	Une fois tous les checks passés, tout passe au vert et l'exercice est terminé	12
Fig. 7	L'enseignant·e et les étudiant·es sont connectés à une session live sur un serveur PLX, du repository « PRG2 »	13
Fig. 8	Equivalent de l'exercice du Snippet 1, dans une version préliminaire de la syntaxe DY ..	18
Fig. 9	Aperçu de l'expérience souhaitée de rédaction dans un IDE	19
Fig. 10	Rustlings en action dans le terminal en haut et l'IDE VSCode en bas	20
Fig. 11	Aperçu d'un exercice de Java sur CodeCheck, avec un code qui compile mais un résultat erroné (16)	21
Fig. 12	Exemple d'exercice PLX en DY, avec une consigne en Markdown sur plusieurs lignes ..	33
Fig. 13	Exemple d'autocomplétion dans Neovim, générée par le serveur de langage <code>rust-analyzer</code> sur l'appel d'une méthode sur les <code>&str</code>	34
Fig. 14	Output du script <code>demo.fish</code> avec les détails de la communication entre un client et notre serveur LSP	37
Fig. 15	Liste de symboles sur un exemple de Rust sur GitHub, générée par Tree-Sitter	39
Fig. 16	CST généré par <code>tree-sitter parse hello.c</code>	40
Fig. 17	Une fois <code>clangd</code> lancé, l'appel de <code>HEY</code> prend une couleur différente que l'appel de fonction mais la même couleur que celle attribuée sur sa définition	40
Fig. 18	CST généré par la grammaire définie sur le fichier <code>mcq.dy</code>	43
Fig. 19	Screenshot du résultat de la commande <code>tree-sitter highlight mcq.dy</code> avec notre exercice surligné	44
Fig. 20	Screenshot dans VSCode une fois l'extension <code>tree-sitter-vscode</code> configurée pour notre grammaire Tree-Sitter	44
Fig. 21	Mise en place	48
Fig. 22	Envoi du check vers le <code>server</code> qui transmet au <code>teacher</code>	48
Fig. 23	Vue d'ensemble avec le serveur de session live, des clients, et notre parseur	50
Fig. 24	Les deux types de messages ne sont envoyés que dans une direction	51
Fig. 25	Exemple de communication avec gestion d'une session	58
Fig. 26	Exemple de communication avec transferts des bouts de code et des résultats	59
Fig. 27	Exemple de communication qui montre l'arrêt du serveur, avec différents clients dans une session ou en dehors	60
Fig. 28	Aperçu du nouveau module <code>live</code> de la librairie <code>plx-core</code>	62
Fig. 29	Aperçu du réseau et processus qui composent le projet PLX	63

Fig. 30 Exemple d'annotation avec <code>tauri-specta</code> sur la commande Tauri et sur les structures associées	64
Fig. 31 Aperçu des tâches tokio lancées et interactions possibles pour 5 clients et 1 session en cours.	68
Fig. 32 Définition d'un cours PLX en syntaxe DY	70
Fig. 33 Aperçu des librairies en jeu, et des dépendances entre librairie et du CLI et de l'application desktop.	71
Fig. 34 Séparation claire entre spec DY et cœur du parseur	72
Fig. 35 Exercice avec une consigne qui contient un bloc de code contenant lui-même un autre exercice en DY. Le contenu du bloc de code n'est pas interprété et la consigne se termine avant le <code>check</code> colorisé	73
Fig. 36 Exemple d'usage de clés et de leur hiérarchie avec un cours PLX	74
Fig. 37 Exemple	74
Fig. 38 Etapes haut-niveau du cœur du parseur, via son point d'entrée <code>parse_with_spec<T></code>	75
Fig. 39 Reprenons l'exercice Salut-moi en exemple	76
Fig. 40 Arbre de blocs généré à partir des lignes du Snippet 49, les blocs sont ordrés de haut en bas.	77
Fig. 41 Aperçu graphique de la spec DY des cours.	79
Fig. 42 Aperçu graphique de la spec DY des compétences.	79
Fig. 43 Aperçu graphique de la spec DY des exercices PLX. La clé <code>exo</code> , <code>check</code> et <code>see</code> sont obligatoires (<code>required=true</code>). <code>args</code> et <code>exit</code> ne peut apparaître qu'une seule fois par <code>check once=true</code> . Seuls <code>exo</code> et <code>see</code> peuvent être donnés sur plusieurs lignes (<code>ValueType=Multiline</code>).	80
Fig. 44 Définition d'un cours PLX dans un fichier <code>course.dy</code>	81
Fig. 45 Equivalent extrait du parseur et affiché en JSON	82
Fig. 46 TODO <code>skills.dy</code>	83
Fig. 47 TODO	83
Fig. 48 TODO <code>exo.dy</code>	84
Fig. 49 TODO	84
Fig. 50 Définition incorrecte d'un cours PLX dans un fichier <code>course.dy</code> . Le <code>goal</code> manque et le <code>code</code> doit être placé après la clé <code>course</code>	84
Fig. 51 Les erreurs ont été détectées par le parseur	85
Fig. 52 TODO <code>skills.dy</code>	85
Fig. 53 TODO	85
Fig. 54 TODO <code>exo.dy</code>	86
Fig. 55 TODO	86

Tables

Annexes

Outils utilisés

Usage de l'intelligence artificielle

L'auteur de ce travail a utilisé l'IA

- pour chercher des syntaxes humainement éditables, comme certains projets ne sont pas bien référencés sur Google, en raison d'une faible utilisation ou décrits avec d'autres mots-clés
- pour trouver la raison de certaines erreurs d'exécution ou de compilation dans les POC fait en Rust
- pour mieux comprendre les règles de précédence de Tree-Sitter et avoir des exemples
- avec LanguageTool pour trouver les fautes d'orthographes ou de grammaire et les corriger, basé sur des règles logiques et sur l'IA (118)

copilot for some bugs in Rust

reverso pour rephraser certaines tournures

aide de Bertil sur 4-5 phrases de l'introduction

Outils techniques

- Neovim pour l'édition du rapport et l'écriture du code
- Template Typst `HEIG-VD typst template for TB` (119)
- Convertisseur de BibTex vers Hayagriva (120)

Logo

Le logo de PLX utilisé sur la page de titre a été créé par l'auteur de ce travail au commencement du projet PLX, bien avant ce travail de Bachelor (121).

Cahier des charges original

Concevoir une expérience d'apprentissage interactive à la programmation avec PLX

Contexte

Ce travail de Bachelor vise à développer le projet PLX (voir [plx.rs](#)), application desktop écrite en Rust, permettant de faciliter la pratique intense sur des exercices de programmation en retirant un maximum de friction. PLX vise également à apporter le plus vite possible un feedback automatique et riche, dans l'idée d'appliquer les principes de la pratique délibérée à l'informatique. PLX peut à terme aider de nombreux cours à la HEIG-VD (tels que PRG1, PRG2, PCO, SYE, ...) à transformer les longs moments de théorie en session d'entraînement dynamique, et redéfinir l'expérience des étudiants sur ces exercices ainsi que les laboratoires. L'ambition est qu'à terme, cela génère un apprentissage plus profond de modèles mentaux solides, pour que les étudiants aient moins de difficultés avec ces cours.

Problème

Le projet est inspiré de Rustlings (Terminal User Interface pour apprendre le Rust), permettant de s'habituer aux erreurs du compilateur Rust et de prendre en main la syntaxe. PLX fournit actuellement une expérience locale similaire pour le C et C++. Les étudiants克lonent un repos Git et travaillent localement sur des exercices afin de faire passer des checks automatisés. À chaque sauvegarde, le programme est compilé et les checks sont lancés. Cependant, faire passer les checks n'est que la 1ère étape. Faire du code qualitatif, modulaire, lisible et performant demande des retours humains pour pouvoir progresser. De plus, les exercices existants étant stockés dans des PDF ou des fichiers Markdown, cela nécessite de les migrer à PLX.

Défis

Ce TB aimerait pousser l'expérience en classe plus loin pour permettre aux étudiants de recevoir des feedbacks sur leur réponse en live, sur des sessions hautement interactives. Cela aide aussi les enseignants à mesurer l'état de compréhension et les compétences des étudiants tout au long du semestre, et à adapter leur cours en fonction des incompréhensions et des lacunes.

Pour faciliter l'adoption de ces outils et la rapidité de création/transcription d'exercices, on souhaiterait avoir une syntaxe épurée, humainement lisible et éditabile, facilement versionnable dans Git. Pour cette raison, nous introduisons une nouvelle syntaxe appelée DY. Elle sera adaptée pour PLX afin de remplacer le format TOML actuel.

Voici un exemple préliminaire de la syntaxe DY qui permettra de décrire un exercice de programmation dans PLX. Elle contient 2 checks pour vérifier le comportement attendu. Le premier cas décrit un check de succès et le deuxième cas décrit une situation d'erreur.

```
exo Just greet me

checks
name Can enter the full name and be greeted
see What is your firstname ?
type John
see Hello John, what's your lastname ?
type Doe
see Have a nice day John Doe !
exit 0

name Stops if the name contains number
see What is your firstname ?
type Alice23
see Firstname cannot contain digits.
exit 1
```

Ces 2 défis impliquent :

1. Une partie serveur de PLX, gérant des connexions persistantes pour chaque étudiant et enseignant connecté, permettant de recevoir les réponses des étudiants et de les renvoyer à l'enseignant. Une partie client est responsable d'envoyer le code modifié et les résultats après chaque lancement des checks.
1. Le but est de définir une syntaxe et de réécrire le parseur en Rust en s'a aidant d'outils adaptés (TreeSitter, Chumsky, Winnow, ...).

Le projet, les documents et les contributions de ce TB, seront publiés sous licence libre.

Objectifs et livrables

1. Livrables standards : rapport intermédiaire ; rapport final ; résumé ; poster.
1. Un serveur en Rust lancé via le CLI plx permettant de gérer des sessions live.
1. Une librairie en Rust de parsing de la syntaxe DY.
1. Une intégration de cette librairie dans PLX.

Objectifs fonctionnels

Les objectifs fonctionnels posent l'hypothèse du cas d'utilisation où un professeur lance une session live pour plusieurs étudiants. Il n'y a cependant pas de rôle spécifique attribué au professeur par rapport aux étudiants, il y a seulement une distinction des permissions entre le créateur de la session et ceux qui la rejoignent.

1. Les professeurs peuvent lancer et stopper une session live via PLX liée au repository actuel, via un serveur défini dans un fichier de configuration présent dans le repository. Il peut exister plusieurs sessions en même temps pour le même repository (afin de supporter plusieurs cours en parallèle dans plusieurs classes). Ils donnent un nom à la session, afin que les étudiants puissent l'identifier parmi les sessions ouvertes. Un code de vérification unique est généré par session permettant de distinguer 2 sessions du même nom dans le même repos.
1. En tant qu'étudiant, une fois le repository cloné, il est possible de lancer PLX, de lister les sessions ouvertes et de rejoindre une session en cours en s'assurant du code de vérification. Un numéro unique incrémental est attribué à chaque étudiant pour la session.
1. Le professeur peut choisir une série d'exercices parmi ceux affichés par PLX, lancer un exercice et gérer le rythme d'avancement de la classe. Cet exercice sera affiché directement chez les étudiants ayant rejoint.

1. Une vue globale permet au professeur d'avoir un aperçu général de l'état des checks sur tous les exercices. En sélectionnant un exercice, il est possible de voir la dernière version du code édité ainsi que les résultats des checks pour ce code, pour chaque étudiant.
1. L'intégration de la librairie `dy` dans PLX permet de décrire les informations d'un cours, des compétences et des exercices. Elle détecte les erreurs spécifiques à PLX.
1. L'intégration dans PLX permet d'utiliser uniquement des fichiers `.dy` pour décrire le contenu. Elle doit aussi afficher les erreurs dans une liste sur une commande dédiée (par ex. `plx check`).

Objectifs non fonctionnels

1. Une session live doit supporter des déconnexions temporaires, le professeur pourra continuer à voir la dernière version du code envoyé et le client PLX essaiera automatiquement de se reconnecter. Le serveur doit pouvoir supporter plusieurs sessions live incluant au total 300 connexions persistantes simultanées.
2. Une session live s'arrête automatiquement après 30 minutes après déconnexion du professeur, cela ne coupe pas l'affichage de l'exercice en cours aux étudiants
3. Pour des raisons de sécurité, aucun code externe ne doit être exécuté automatiquement par PLX. Seule une exécution volontaire par une action dédiée peut le faire.
4. Le temps entre la fin de l'exécution des checks chez l'étudiant et la visibilité des modifications par l'enseignant ne doit pas dépasser 3s.
5. Le code doit être le plus possible couvert par des tests automatisés, notamment par des tests end-to-end avec de multiples clients PLX.
6. Le parseur DY doit être assez capable de parser 200 exercices en < 1s.
7. Retranscrire à la main un exercice existant du Markdown en PLX DY ne devrait pas prendre plus d'une minute.

Objectif nice to have

1. La librairie `dy` permettrait d'intégrer le parseur et les erreurs spécifiques à un language server permettant une expérience complète d'édition dans VSCode et Neovim.
2. La librairie `dy` serait également capable de générer des définitions TreeSitter pour supporter le syntax highlighting via ce système.

Calendrier du projet

En se basant sur le calendrier des travaux de Bachelor, voici un aperçu du découpage du projet pour les différents rendus.

Rendu 1 - 10 avril 2025 - Cahier des charges

- Rédaction du cahier des charges.
- Analyse de l'état de l'art des parsers, des formats existants de données humainement éditables, du syntax highlighting et des languages servers.
- Analyse de l'état de l'art des protocoles bidirectionnels temps réel (websockets, gRPC, ...) et des formats de sérialisation (JSON, protobuf, ...).
- Prototype avec les librairies disponibles de parsing et de language servers en Rust, choix du niveau d'abstraction espéré et réutilisation possible.

Rendu 2 - 23 mai 2025 - Rapport intermédiaire

- Rédaction du rapport intermédiaire.
- Définition de la syntaxe DY à parser, des préfixes et flags liés à PLX, et la liste des vérifications et des erreurs associées.
- Définition d'un protocole de synchronisation du code entre les participants d'une session.
- Prototype d'implémentation de cette synchronisation.
- Prototype des tests automatisés sur le serveur PLX.
- Définition du protocole entre les clients PLX et le serveur pour les entraînements live.

Moitié des 6 semaines à temps plein - 4 juillet 2025

- Écriture des tests de validation du protocole et de gestion des erreurs.
- Développement du serveur PLX.
- Rédaction du rapport final par rapport aux développements effectués.

Rendu 3 - 24 juillet 2025 - Rapport final

- Développement d'une librairie `dy`.
- Intégration de cette librairie à PLX.
- Rédaction de l'affiche et du résumé publiable.
- Rédaction du rapport final.