Université Mohammed V - Rabat
École Nationale d'Informatique
et d'Analyse des Systèmes

Rapport du projet Projet MDE

Filière

Génie Logiciel

Sujet

DSL for generating Spring Boot RESTful APIs - VS Code Extension

---

Réalisé par

Yassine OUHADI

Badreddine HOUSNI

Hamza ALAMI IBN JAMAA

Mehdi CHARIFE

Encadré par

M. MAHMOUD EL HAMLAOUI

Année Universitaire 2023-2024

Table des matières

Introduction générale

L'ingénierie dirigée par les modèles (MDE) représente une approche novatrice dans le développement logiciel, visant à améliorer la qualité, la compréhension et la maintenance des systèmes en utilisant des metamodels pour représenter les différents aspects d'un model. Cette méthodologie repose sur le principe des transformations modèle vers modèle (M2M) et modèle vers texte (M2T), permettant la manipulation et la génération automatique de code à partir des metamodels.

Eclipse, fort de composants puissants tels qu'EMF, a constitué un pilier essentiel de la MDE. EMF a fourni une infrastructure robuste permettant la création et la manipulation de metamodels, tandis que des outils tels que Xtext et Epsilon ont simplifié la création des DSL. Cependant, notre orientation se tourne désormais vers l'exploitation de L'architecture MDA dans Llenvironnement Visual Studio Code.

La transition vers VS Code s'inscrit dans une démarche d'adaptation aux besoins actuels des développeurs. Alors qu'Eclipse a prospéré dans le contexte de l'Ingénierie Dirigée par les Modèles, VS Code offre une approche plus légère, modulaire et largement adoptée. Cette transition représente une opportunité de réimaginer les pratiques d'ingénierie dirigée par les modèles dans un environnement moderne.

Dans ce contexte, notre projet se positionne comme une extension pour VS Code, mettant l'accent sur le développement d'un Domain Specific Language (DSL) dédié à la génération de code. En capitalisant sur le concept de transformation modèle vers texte (M2T), nous cherchons à exploiter les avantages de l'architecture MDA dans l'écosystème de VS Code.

Chapitre 1

Présentation du sujet

Dans ce chapitre, nous débuterons en explorant le contexte de notre choix, en introduisant l'idée à la base de notre projet.

1.1 Contexte

Notre idée est d'introduire les concepts éprouvés de l'Ingénierie Dirigée par les Modèles dans l'environnement Visual Studio Code. En développant un DSL spécifique à la génération de code, nous offrons aux développeurs la possibilité d'appliquer ces principes directement dans leur éditeur préféré. Cette extension offrira des fonctionnalités avancées s'intégrant harmonieusement à l'écosystème VS Code.

En utilisant les fonctionnalités du protocole Language Server (LSP), cette extension s'intègre de manière transparente les fonctionnalités de l'Ingénierie Dirigée par les Modèles (MDE) dans Visual Studio Code. Les utilisateurs sont ainsi habilités à créer des instances de modèles à partir de notre méta-modèle, bénéficiant d'une validation syntaxique, d'une complétion de code et d'une validation de code, ce qui facilite grandement la génération de code.

1.2 Jgen Extension

L'objectif consiste actuellement à élaborer un DSL basé essentiellement sur les concepts fondamentaux de l'architecture REST de Spring Boot. Il équivaut à la création d'un méta-modèle pour une partie de base de l'API REST de Spring Boot. Nous avons également l'intention d'exploiter une extension pour Visual Studio Code dédiée à ce DSL. Cette extension facilitera la génération de code pour les API RESTful de Spring Boot, tout en fournissant une interface pour simplifier l'instanciation de ce méta-modèle.

Fig. 1.1 : Jgen Extension

Le schéma ci-dessus expose l'architecture anticipée pour le développement de notre extension Jgen, illustrant également les différentes sous-fonctionnalités de l'extension :

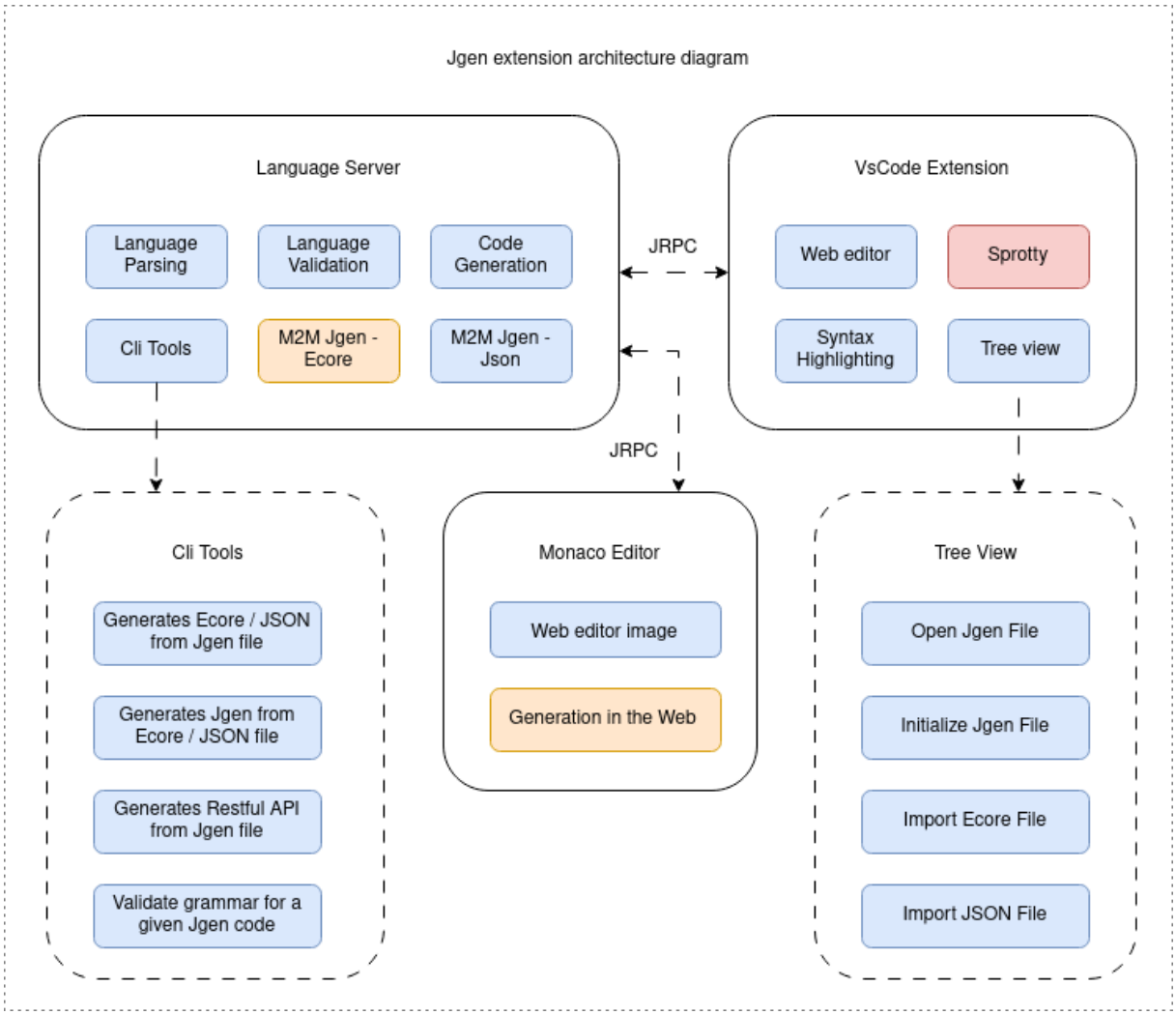

Fig. 1.2 : Jgen architecture diagram

1.3 Jgen metaModel

En ce qui concerne notre DSL, il est nécessaire au préalable de disposer d'un méta-modèle Ecore décrivant de manière exhaustive les différentes composantes de notre méta-modèle ainsi que les relations qui existent entre elles. L'utilisation de ce méta-modèle Ecore offre de nombreux avantages et services fournis par Eclipse, notamment par EMF.

En se basant sur ce méta-modèle, nous procédons ensuite à la génération d'une grammaire Xtext dédiée à notre DSL. Ainsi, la présence d'un méta-modèle Ecore pour notre DSL s'avère indispensable, que ce soit pour faciliter la génération de code ou pour effectuer des transformations entre le modèle Jgen et un autre modèle (Ecore, JSON, ..), ou inversement, selon les besoins spécifiques du projet.

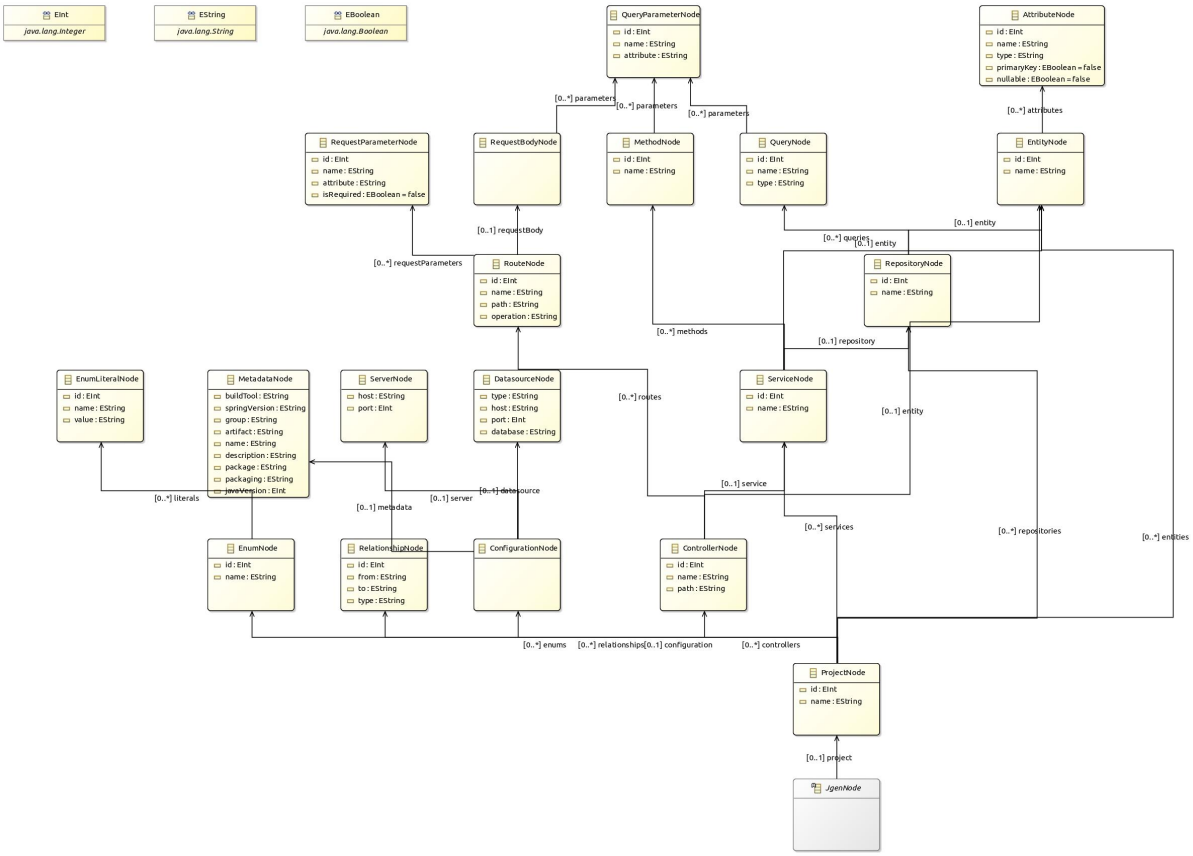

Fig. 1.3 : Jgen metaModel

1.4 Language Server Protocol

Le Language Server Protocol (LSP) est un protocole standard pour les serveurs de langage de programmation. Il permet aux éditeurs de code et aux environnements de développement intégrés (IDE) de communiquer efficacement avec les serveurs de langage pour fournir des fonctionnalités telles que la complétion de code, la vérification de la syntaxe, la navigation de code, et la recherche de définitions.

La conception d'un LSP consiste à définir les méthodes, les paramètres et les événements qui seront utilisés pour communiquer entre l'éditeur de code et le serveur de langage. Il est important de prendre en compte les besoins spécifiques de la langue de programmation cible, ainsi que les exigences en matière de performances et de fiabilité.

Il est également important de considérer la compatibilité avec les différents éditeurs de code et les IDE qui prennent en charge le protocole LSP, en utilisant des implémentations standard pour faciliter l'intégration.

Enfin, il est important de fournir une documentation claire et détaillée pour les développeurs qui souhaitent implémenter ou utiliser le serveur de langage.

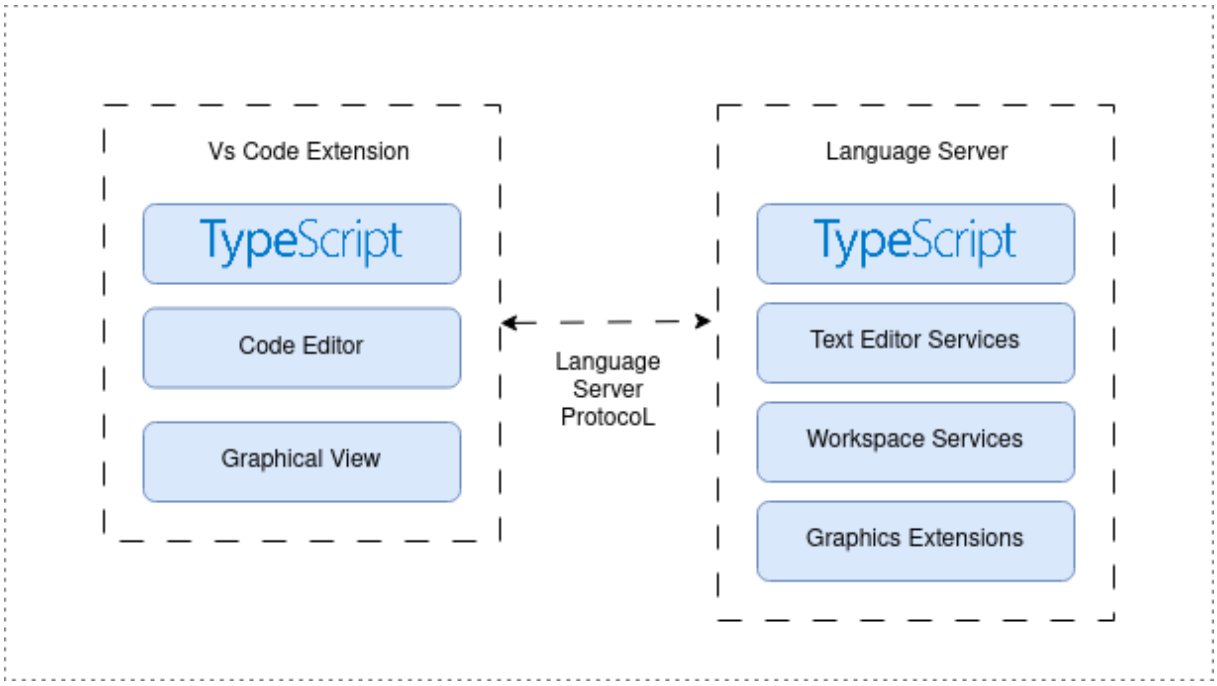

Fig. 1.4 : Language Server Protocol

Language server s'exécute en tant que processus distinct, et les outils de développement communiquent avec le serveur à l'aide du protocole de langage via JSON-RPC. Voici un exemple de la manière dont un outil et un serveur de langage communiquent au cours d'une session d'édition standard :

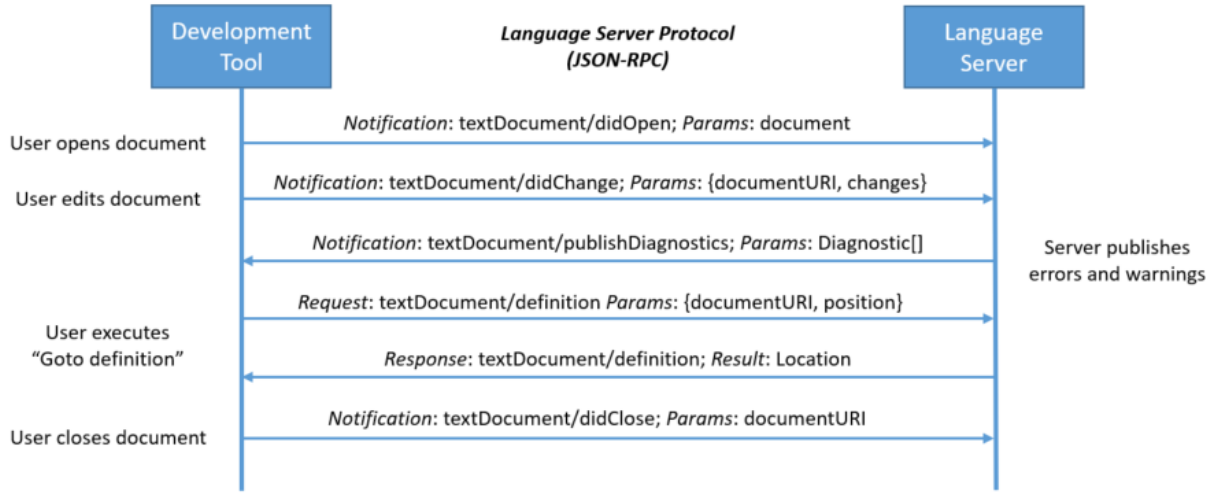

Fig. 1.5 : JSON-RPC

C'est généralement ainsi que se déroule la communication entre le language server pour notre DSL Jgen et l'éditeur d'extension, ou bien le Monaco Editor.

Chapitre 2

Réalisation

Dans ce chapitre, on présente le choix des technologies et des outils de travail utilisés pour réaliser notre DSL - extension VSCode.

2.1 Outils utilisés

2.1.1 Eclipse EMF

Eclipse Modeling Framework (EMF) est un framework de modélisation basé sur Eclipse et une fonction de génération de code pour la création d'outils et d'autres applications basés sur un modèle de données structurées.

2.1.2 VSCode extension api

L'API d'extension de Vs Code est un ensemble de fonctionnalités qui permet aux développeurs de créer des extensions pour l'éditeur de code Visual Studio Code. Ces extensions peuvent ajouter de nouvelles fonctionnalités à l'éditeur, comme des commandes personnalisées, des raccourcis clavier, des colorations de syntaxe, des débogueurs, des outils de développement, des thèmes, des snippets, etc.

2.1.3 Lsp

LSP (Language Server Protocol) est un protocole open-source pour les services de langage développé par Microsoft. Il permet aux éditeurs de code et aux IDEs de communiquer avec des services de langage externes pour fournir des fonctionnalités telles que la coloration syntaxique, la complétion de code, la navigation de code, la vérification de code, la modification de code, etc.

2.1.4 Langium

Langium un outil pour l'ingénierie linguistique qui prend en charge le protocole Language Server et est open source. Il est écrit en TypeScript et fonctionne sous Node.js.

Langium peut être utilisé pour diverses tâches telles que l'analyse de code, la génération de code et l'autocomplétion. Le protocole Language Server est un protocole ouvert qui permet aux différents éditeurs de code et aux environnements de développement intégrés (EDI) de s'intégrer aux outils d'analyse de code et de débogage, tels que Langium.

2.2 Création de notre DSL

Concernant l'écriture de notre DSL, nous avons suivi un processus étape par étape en utilisant différentes technologies.

2.2.1 Écrire la grammaire

Tout d'abord, nous avons créé notre premier méta-modèle Ecore à l'aide d'Eclipse EMF. Ce méta-modèle a été conçu pour décrire les notions de base de la création d'APIs RESTful avec Spring Boot en Java. À partir de ce méta-modèle, nous avons généré le fichier Genmodel.

Ensuite, nous avons utilisé ce Genmodel pour créer la grammaire initiale Xtext correspondante. Cette étape a permis de définir la syntaxe concrète du langage. Cependant, en basant notre grammaire sur Xtext, cependant que Langium partageait une grammaire similaire. Nous avons donc personnalisé la grammaire Xtext pour s'aligner sur nos besoins spécifiques.

Une fois la grammaire définie, nous avons utilisé la commande suivante pour générer le code TypeScript correspondant à notre grammaire en utilisant Langium :

```
$ npm run langium:generate
```

Après l'exécution de cette commande, la génération produit un modèle sémantique sur lequel les AST peuvent être mappés, ainsi qu'un parseur capable de reconnaître notre langage.

2.2.2 Implémenter la validation

Dans le cadre de notre DSL, la validation joue un rôle crucial en garantissant la conformité des programmes créés. Par exemple, nous pouvons implémenter des validations pour détecter la duplication d'éléments, telles que des noms non uniques pour les définitions ou les paramètres. De plus, la validation peut signaler des références inexistantes à des objets, des relations incorrectes entre deux entités ou une mauvaise implémentation de services. Ces contraintes sont essentielles pour garantir la cohérence et la fiabilité des programmes créés avec notre DSL. En utilisant le registre de validation et en associant des fonctions de validation à des nœuds spécifiques de notre AST, nous pouvons mettre en place des mécanismes robustes pour améliorer la qualité des programmes générés tout en facilitant le développement dans notre environnement DSL.

2.2.3 Personnaliser la CLI

La personnalisation de la CLI offre une accessibilité accrue et facilitant l'intégration de fonctionnalités avancées. Notre CLI a été configurée pour prendre en charge plusieurs actions clés, permettant une utilisation flexible de notre langage. Ci-dessous, nous présentons les principales actions disponibles dans notre CLI, ainsi qu'une brève explication de chacune :

1. `generateEcore` : Génère un modèle Ecore à partir du code Jgen.
2. `generateJson` : Génère du code Json à partir du code Jgen.
3. `generateJgen` : Génère du code Jgen à partir du code Json ou Ecore.
4. `validateGrammar` : Valide la grammaire pour un code Jgen donné.
5. `generateRESTfulAPI` : Génère une API RESTful à partir du code Jgen.

```
yassine@yassine-HP-EliteBook-840-G3:~/MDE/JGEN$ ./bin/cli
Usage: cli [options] [command]

Options:
  -V, --version                   output the version number
  -h, --help                      display help for command

Commands:
  generateEcore [options] <file>  generates Ecore from Jgen code
  generateJson [options] <file>   generates Json from Jgen code
  generateJgen [options] <file>   generates Jgen from Json/Ecore code
  validateGrammar <file>          validate grammar for a given Jgen code
  generateRESTfulAPI [options]    generates Restful API from Jgen code
  help [command]                  display help for command
yassine@yassine-HP-EliteBook-840-G3:~/MDE/JGEN$ ./bin/cli generateRESTfulAPI --help
Usage: cli generateRESTfulAPI [options]

generates Restful API from Jgen code

Options:
  -d, --destination <dir>  destination directory of generating
  -c, --content <content>  jgen content directly
  -p, --path <file>        source file (possible file extensions: ${fileExtensions})
  -h, --help               display help for command

Examples:

  To generate code for a given DSL file with a specified destination from file path:
    $ ./bin/cli generateRESTfulAPI -d <destination-path> -p <file>

  To generate code for a given DSL file with a specified destination from direct content:
    $ ./bin/cli generateRESTfulAPI -d <destination-path> -c "<jgen-content>"

Replace <destination-path> with the desired directory path where you want to store the generated code.
```

FIG. 2.1 : Jgen CLI

2.2.4 Generation de code

La génération de code implique la transformation d'un AST représentant une instance de notre DSL en un code cible. Pour la génération de code Ecore, il s'agit d'une transformation M2M. Il est nécessaire de connaître le métamodèle d'Ecore afin de réaliser cette transformation correctement, en utilisant un mappage d'un métamodèle à un autre.

Pour la génération de RESTful APIs, il s'agit de créer des templates vides. Ce processus est similaire à la génération avec Acceleo dans Eclipse.

2.3 Construire l'extension Vs Code

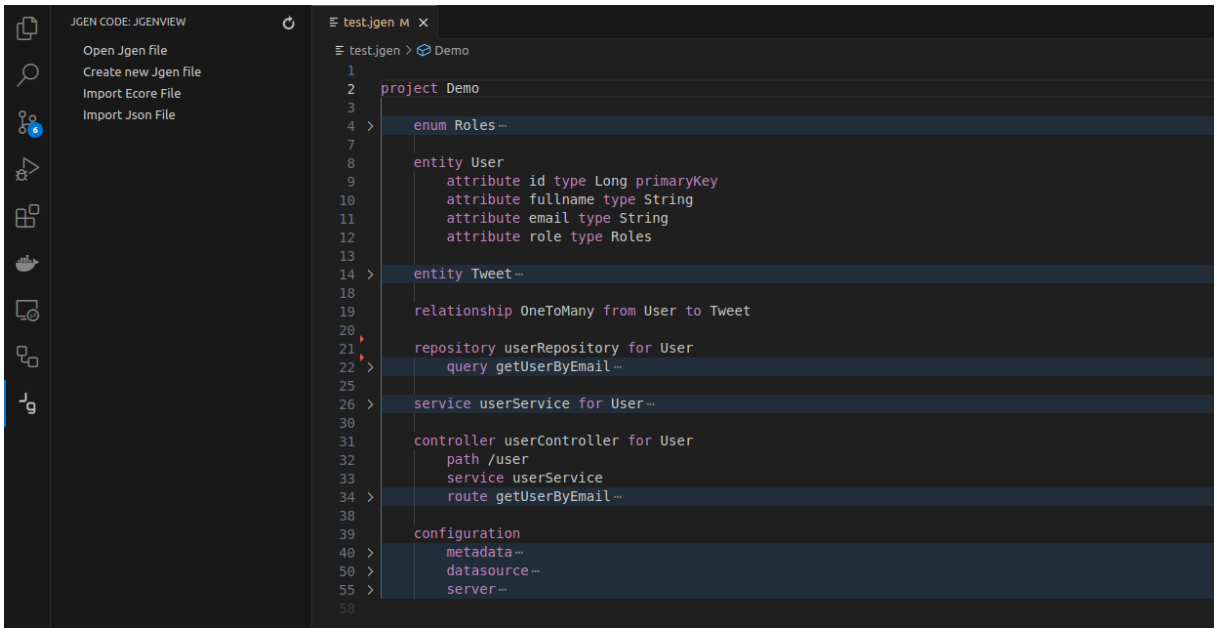

Fig. 2.2 : Jgen Editor

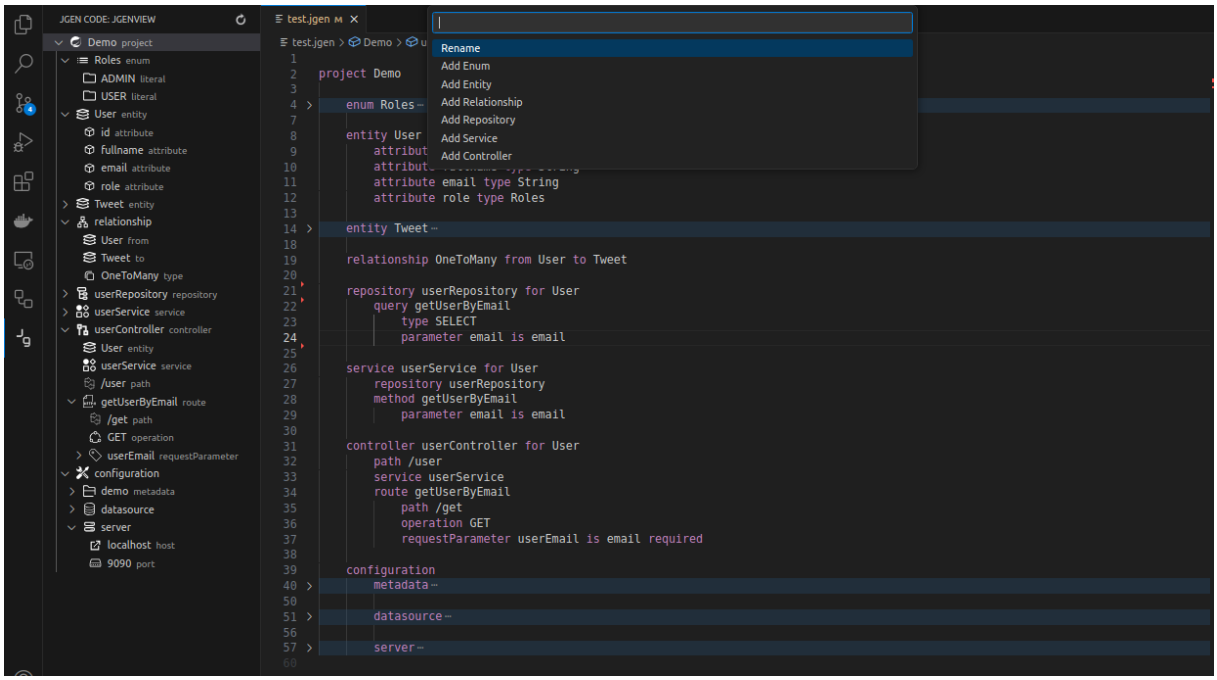

Fig. 2.3 : Jgen Treeview

2.4 Génération sur le Web

2.4.1 L'éditeur Monaco

Nous avons réussi à intégrer notre DSL avec l'éditeur web Monaco. Cette intégration ouvre la porte à des fonctionnalités avancées telles que l'accès aux fonctionnalités de Langium dans un environnement web, offrant ainsi une solution complète de développement pour notre DSL.

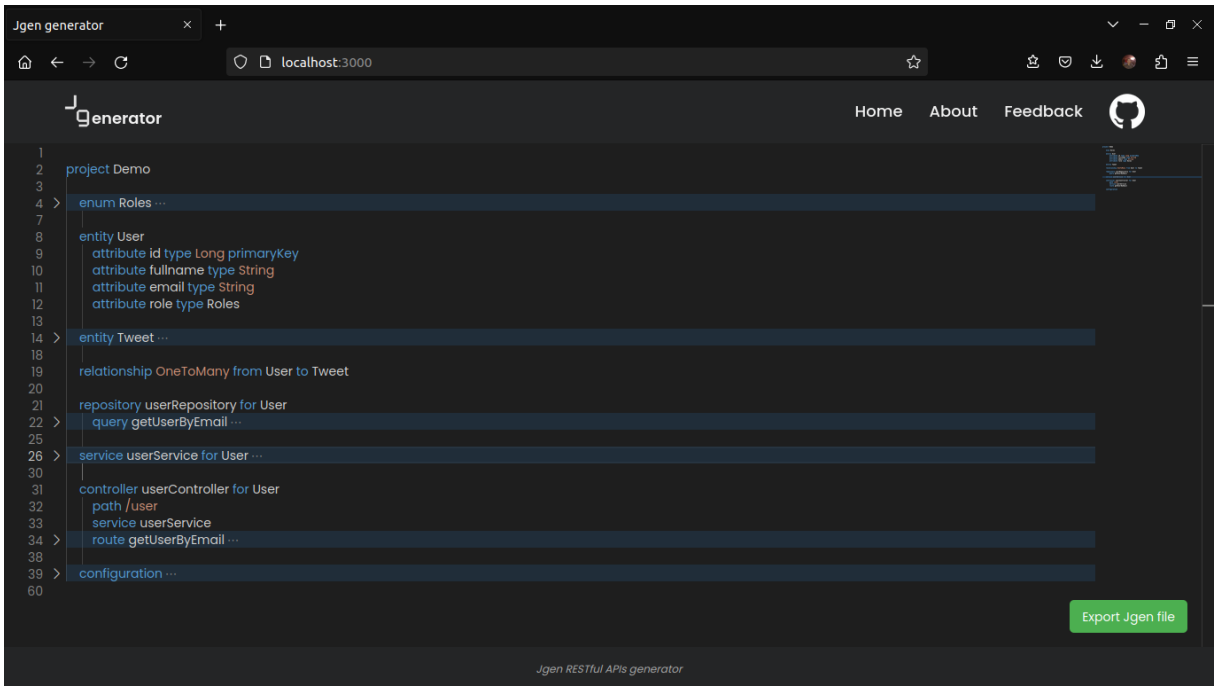

FIG. 2.4 : Monaco Editor

2.4.2 Génération sur le Web

Nous explorons l'intégration réussie de la génération de code dans Monaco Editor en utilisant le langage Jgen. Le processus repose sur l'architecture Language Server Protocol pour la communication entre le langage server et l'éditeur Monaco.

2.4.2.1 Langage Server Implementation

Dans le langage server, nous avons étendu notre application en écoutant les notifications générées lors de la validation des documents Jgen. Ces notifications, encapsulant des informations cruciales telles que l'URI du document, son contenu et d'éventuelles diagnostics, sont envoyées au client via le LSP. La génération de code est effectuée uniquement si le document est entièrement validé et ne contient aucune erreur.

2.4.2.2 Monaco Editor Configuration

Du côté du client, notre configuration Monaco Editor est personnalisée pour le langage Jgen. Nous avons créé un wrapper autour du client MonacoEditorLanguageClient pour établir la connexion avec le langage server Jgen.

Pour compléter cette intégration, nous avons ajouté un bouton d'exportation permet à l'utilisateur de télécharger le code actuel du projet Jgen au format de fichier approprié, facilitant ainsi le partage et la sauvegarde des projets.

2.5 CI/CD workflow with Github Actions

2.5.1 Extensions Tests

Dans notre pipeline de développement orchestré par GitHub Actions, nous assurons la fiabilité et la fonctionnalité de notre extension de langage Jgen grâce à une série de tests automatisés.

Pour évaluer la fonctionnalité de l'extension, un fichier de test dédié, 'tests/generateRestfulAPI.test.ts', est exécuté. Ce test se concentre sur la fonction 'generateRestfulAPIAction', vérifiant qu'elle génère avec succès du code API Restful. Toutes les erreurs rencontrées au cours de ce processus sont traitées de manière appropriée dans un bloc try-catch.

Ces tests rigoureux garantissent que toutes les modifications apportées à l'extension de langage Jgen sont soumises à une validation approfondie, maintenant ainsi un haut niveau de qualité et de fonctionnalité du code.

2.5.2 Déploiement sur Docker Hub

Pour rationaliser le processus de déploiement et permettre aux utilisateurs d'utiliser facilement notre environnement Jgen, nous exploitons Docker et GitHub Actions pour la conteneurisation et le déploiement automatisés.

Ce pipeline garantit que nos composants CLI et web sont efficacement conteneurisés, permettant aux utilisateurs de déployer et d'utiliser l'environnement Jgen avec l'intégration de l'éditeur Monaco directement depuis Docker Hub. Les GitHub Actions orchestrées fournissent une approche robuste et automatisée pour maintenir des versions conteneurisées cohérentes et fiables de notre environnement de langage Jgen.

Conclusion et perspectives

La génération de code à partir de notre DSL a démontré l'efficacité du processus, avec la possibilité de transformer un AST représentant notre DSL en un code cible, répondant ainsi aux besoins spécifiques du développement RESTful avec Spring Boot.

Pour les perspectives à court terme, nous visons à enrichir davantage l'intégration de l'éditeur Monaco avec Langium. Dans cette intégration web, nous envisageons :

1. Gestion des Projets Utilisateur : Cette fonctionnalité permettra aux utilisateurs de générer, visualiser et gérer leurs projets Jgen de manière centralisée.

2. Exportation Avancée : Le bouton d'exportation sera étendu pour inclure des options avancées. En plus de télécharger le code actuel du projet Jgen, l'utilisateur pourra choisir parmi plusieurs formats de fichier, tels que le format Jgen, JSON ou Ecore. Cette flexibilité favorisera une intégration transparente avec d'autres outils et plates-formes.

3. Importation de Fichiers : Les utilisateurs auront également la possibilité d'importer des fichiers Jgen, JSON ou Ecore directement dans l'éditeur. Cela ouvrira la porte à une collaboration simplifiée, permettant aux utilisateurs de partager et d'importer des projets Jgen avec leurs collègues et partenaires.

4. Gestion des Versions : Un aspect crucial de la gestion de projet est la gestion des versions. Nous envisageons d'implémenter une fonctionnalité de suivi des versions pour chaque projet, permettant aux utilisateurs de revenir à des versions précédentes du code généré, offrant ainsi un meilleur contrôle et une meilleure traçabilité.

En envisageant des perspectives futures, il serait intéressant de généraliser notre DSL pour y intégrer davantage de notions liées au développement d'API REST. Cette extension permettrait une couverture plus large des cas d'utilisation et offrirait une plus grande flexibilité aux développeurs.

Par ailleurs, l'automatisation pourrait être améliorée en intégrant un diagramme synchronisé avec notre DSL, en utilisant des outils tels que Sprotty. Cela permettrait une visualisation graphique des entités et de leurs relations, offrant ainsi une compréhension plus intuitive du modèle.