

PLATEFORME WEB D'APPRENTISSAGE DE LA PROGRAMMATION

Mohamed-Khalil Mehalli

Encadrants : Nicolas BAUDRU, Nicolas DURAND
¹Aix-Marseille Université

Résumé

Ce projet présente le développement d'une plateforme web dédiée à l'apprentissage de la programmation. Face à la complexité de configuration des environnements de développement locaux, cette solution propose un espace où les étudiants peuvent écrire, compiler et tester leur code directement dans le navigateur. La plateforme supporte les langages C, Java et Python. L'ensemble est déployable via Docker Compose pour un test rapide.

Mots clés : Plateforme web, Apprentissage programmation, API REST, Compilation multi-langages, FastAPI, Angular.

1. Introduction

L'apprentissage de la programmation constitue un enjeu majeur dans la formation des étudiants en informatique. Cependant, pour les étudiants débutants, la configuration d'un environnement de développement local représente souvent une barrière technique décourageante avant même d'avoir écrit la première ligne de code. Personnellement, en première année de licence, l'installation d'un compilateur C sous Windows s'est avérée particulièrement difficile à cause de la configuration des variables d'environnement et des chemins d'accès.

Ce projet vise à répondre à cette problématique en développant une plateforme web d'apprentissage de la programmation. L'objectif principal est de fournir un environnement accessible où les étudiants peuvent résoudre des exercices mis à disposition par leurs professeurs, écrire, compiler et tester leur code directement dans le navigateur, sans aucune installation préalable.

La plateforme permet :

- La création complète d'un exercice avec plusieurs fichiers, des séries de tests et des indices qui se révèlent progressivement.
- La création de modules, cours ainsi que la modification de ces derniers ainsi que les exercices.

— La résolution de ces mêmes exercices.

Ce rapport détaille l'architecture retenue, les choix technologiques, la modélisation des données, ainsi que les mécanismes de compilation et de test qui constituent le coeur technique de la plateforme.

2. État de l'art

Plusieurs plateformes d'apprentissage de la programmation existent sur le marché, chacune avec ses spécificités.

LeetCode et **HackerRank** proposent des environnements d'entraînement orientés vers la préparation aux entretiens techniques.

Replit offre un environnement de développement cloud complet et polyvalent.

France-IOI, une plateforme française, propose une approche similaire à ce projet avec des exercices progressifs et une exécution en ligne.

3. Architecture générale

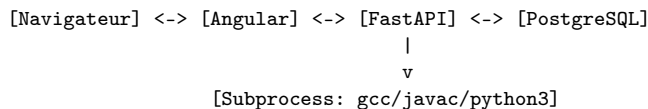
La plateforme repose sur une architecture web classique en trois couches : frontend, backend et base de données.

Le **frontend** constitue l'interface utilisateur. Développé en Angular, il est responsable de l'affichage des pages, de l'édition du code source et de la communication avec le backend via des appels HTTP REST.

Le **backend** implémente la logique métier de l'application : gestion des exercices, traitement des soumissions de code, orchestration de la compilation et de l'exécution des tests. Il joue également le rôle d'intermédiaire avec la base de données.

La **base de données** relationnelle PostgreSQL stocke l'ensemble des informations persistantes : utilisateurs, structure pédagogique, exercices, soumissions et résultats.

L'architecture de la plateforme est représentée comme suit :



L'architecture adoptée n'est pas modulaire comme une architecture en microservices, mais le code a été structuré de manière modulaire pour faciliter la maintenance et l'ajout de nouvelles fonctionnalités.

4. Choix technologiques

Les technologies retenues ont été sélectionnées selon plusieurs critères : productivité de développement, adéquation avec les besoins du projet ainsi que mes compétences et mes envies personnelles.

4.1. Frontend : Angular

Angular a été choisi pour sa capacité à structurer le code de manière claire et maintenable grâce à son architecture en composants. J'ai aussi choisi Angular pour me permettre de l'apprendre, ayant raté son cours l'année dernière à cause de la mobilité.

L'architecture a été pensée pour la production d'un code maintenable et lisible :

- L'interface de l'éditeur de texte s'adapte dynamiquement au rôle de l'utilisateur (configurations **STUDENT** vs **TEACHER**) sans duplication de code
- L'utilisation des décorateurs **@Input** et **@Output** garantit une communication parent-enfant claire et un flux de données maîtrisé, facilitant la maintenance.
- Un service dédié à la navigation (**NavigationInformationService**) intègre un mécanisme de cache, réduisant drastiquement les appels réseaux inutiles lors des changements de pages.

Concernant l'éditeur de code, une approche minimaliste a été privilégiée pour l'instant : un simple élément **HTML textarea**. Ce choix m'a permis de me concentrer sur des éléments critiques de ce projet tout en restant suffisant pour les exercices pédagogiques visés.

4.2. Backend : Python avec FastAPI

Le backend repose sur une API REST développée en Python avec le framework FastAPI. Python a été préféré à Node.js pour sa syntaxe claire permettant une implémentation rapide et une bonne lisibilité du code.

FastAPI offre plusieurs avantages significatifs :

- Documentation automatique des endpoints, qui s'est avérée très utile pour tester mes endpoints et la résolution d'erreurs
- Validation automatique des entrées grâce à l'intégration native de Pydantic
- Support natif de l'asynchronisme pour les opérations d'entrée/sortie
- Performances satisfaisantes

La validation des données entrantes est assurée par **Pydantic**. Cette bibliothèque agit comme un filtre de sécurité à l'entrée de l'API, elle vérifie automatiquement que les informations envoyées correspondent exactement au format attendu. Cela permet de rejeter immédiatement les données incorrectes avant qu'elles ne soient traitées, garantissant ainsi que le cœur de l'application ne manipule que des informations valides.

4.3. Base de données : PostgreSQL avec SQLAlchemy

PostgreSQL a été choisi pour sa fiabilité et mon expérience passée. L'accès à la base depuis le backend est réalisé via **SQLAlchemy**, un ORM permettant de manipuler les données sous forme d'objets Python tout en sécurisant les requêtes contre les injections SQL (qui sera utile dans le futur avec un système d'authentification).

Alembic complète cet ensemble en tant qu'outil de migration de base de données, permettant de gérer proprement l'évolution du schéma au fil du développement.

4.4. Conteneurisation : Docker

L'ensemble du projet est conteneurisé via Docker et orchestré par Docker Compose. Cette approche permet de lancer la plateforme complète sans installation préalable autre que Docker lui-même, répondant ainsi à l'exigence de déploiement aisé mentionnée dans le cahier des charges.

5. Conception du système

5.1. Gestion et stockage des exercices

La conception du système de stockage des exercices a nécessité plusieurs semaines de réflexion et d'échanges avec mes encadrants. La solution retenue est à la fois

simple d'utilisation pour les enseignants et optimale pour le stockage des données.

5.1.1. Création et balisage (Côté Enseignant)

Lorsqu'un enseignant crée un exercice, il définit les zones à compléter en insérant des balises directement dans le code source. Ces balises utilisent la syntaxe `<complete id="...">` pour l'ouverture et `</complete>` pour la fermeture. Elles doivent être placées en commentaire afin de respecter la syntaxe du langage de programmation utilisé.

Listing 1 : Exemple de balisage en C

```
int addition(int a, int b){
    // <complete id="1">
    return a + b;
    // </complete>
}
```

5.1.2. Analyse et séparation (Côté Backend)

Lors de la sauvegarde, le backend analyse le fichier source et le scinde en deux entités distinctes dans la base de données :

- **Le Template** (table `exercise_files`) : contient la structure immuable du code. Les balises de l'enseignant sont remplacées par des marqueurs (`// TODO: 1 ... // END TODO: 1`) destinés à l'affichage étudiant.
- **Les Marqueurs** (table `exercise_markers`) : stockent la solution attendue (le contenu original situé entre les balises), ainsi que l'identifiant du marqueur et la référence au fichier parent.

5.1.3. Optimisation des soumissions étudiants

Grâce à cette architecture, le processus côté étudiant est fortement optimisé :

- Le frontend reçoit uniquement les templates avec les zones TODO à compléter.
- Lors de la soumission, seul le code rédigé entre les marqueurs est enregistré dans la table `submission_marker`. Cette approche réduit considérablement le volume de données stockées.

5.2. Modularité du compilateur

L'un des défis techniques du projet était de supporter plusieurs langages de programmation sans multiplier les conditions et le code dupliqué. La solution retenue utilise une configuration centralisée, définissant les commandes spécifiques à chaque langage.

Listing 2 : Configuration multi-langages COMPILER_CONFIG

```
COMPILER_CONFIG = {
    "c": {
        "compiler_cmd": ["gcc", "{input_files}",
            "-o", "app"],
        "run_cmd": ["/app"],
        "extension": "c",
        "main_name": "main"
    },
    "python": {
        "compiler_cmd": ["python3", "-m",
            "py_compile", "main.py"],
        "run_cmd": ["python3", "main.py"],
        "extension": "py",
        "main_name": "main"
    },
    "java": {
        "compiler_cmd": ["javac",
            "{input_files}"],
        "run_cmd": ["java", "Main"],
        "extension": "java",
        "main_name": "Main"
    }
}
```

Cette architecture présente un avantage majeur : l'ajout d'un nouveau langage (par exemple C++) se résume à l'ajout de quelques lignes de configuration, sans modification de la logique de compilation elle-même.

5.3. Pipeline d'exécution et sécurité

L'exécution d'une solution étudiante suit un processus rigoureux en trois phases :

1. **Phase 1 - Extraction** : La fonction `extract_student_solutions()` analyse le code soumis par l'étudiant et extrait les solutions présentes entre les marqueurs TODO. Une validation vérifie que tous les marqueurs attendus sont présents.
2. **Phase 2 - Reconstruction** : La fonction `inject_markers_into_template()` injecte les solutions extraites dans les templates originaux de l'enseignant. Cette étape garantit que l'étudiant n'a pas pu altérer les parties critiques du code.
3. **Phase 3 - Exécution** : Les fichiers reconstitués sont écrits dans un répertoire temporaire isolé, puis compilés et exécutés via `subprocess`.

Limite reconnue : L'exécution se fait via subprocess local, sans véritable environnement sandboxé. Un code malveillant pourrait théoriquement accéder au système de fichiers. Cette limitation résulte des contraintes de temps du projet. L'exécution dans un environnement isolé constitue une amélioration future identifiée.

5.4. Système de tests

La méthode d'exécution des tests a aussi fait l'objet d'une réflexion en concertation avec mes encadrants. La solution retenue se distingue par sa simplicité d'implémentation technique tout en offrant une grande flexibilité aux enseignants.

Cette approche repose sur l'utilisation des arguments de la ligne de commande (**argv**) pour injecter les données d'entrée, et sur la capture de la sortie standard (**stdout**) pour valider le résultat.

Concrètement, lors de la conception d'un exercice, l'enseignant rédige un fichier principal (main) chargé de lancer le test.

Reprenons l'exemple de la fonction addition définie précédemment. Pour évaluer le travail de l'étudiant, l'enseignant fournira le fichier main.c suivant :

Listing 3 : Fichier main.c pour tester la fonction addition

```
#include <stdio.h>
#include <stdlib.h>
#include "fonction.h"

int main(int argc, char ** argv) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int c = addition(a, b);
    printf("%d", c);
    return 0;
}
```

Pour définir un cas de test, l'enseignant spécifie simplement les arguments d'entrée et le système lui montrera la solution. Il pourra ensuite valider le test si le résultat lui convient.

6. Modélisation des données

La base de données comprend 10 tables principales organisées en deux catégories. Vous pouvez retrouver le schéma complet (voir 1)

6.1. Tables de contenu

Table	Description
User	Comptes utilisateurs avec rôle
Unit	Modules pédagogiques
Course	Cours contenant des exercices
Exercise	Définition d'un exercice
ExerciseFile	Fichiers sources (templates)
ExerciseMarker	Solutions de référence
TestCase	Cas de test (entrée/sortie)
Hint	Indices

TABLE 1 : Tables de contenu

6.2. Tables de suivi étudiant

Table	Description
SubmissionHistory	Historique des soumissions
SubmissionMarker	Code soumis par marqueur
SubmissionResult	Résultat par cas de test
ExerciseProgress	Progression de l'étudiant
HintView	Indices consultés

TABLE 2 : Tables de suivi étudiant

7. Déploiement

L'ensemble du projet est conteneurisé via Docker Compose, orchestrant trois services :

- **PostgreSQL** : Image `postgres:15`, la base est initialisée via le fichier `init.sql` pour faciliter les tests.
- **Backend** : Image basée sur `python:3.10`, incluant `gcc`, `build-essential` et `openjdk-17`.
- **Frontend** : Application Angular servie par Nginx (un serveur web léger).

Cette configuration permet de lancer la plateforme avec une unique commande :

```
docker-compose up --build
```

8. Conclusion

8.1. État d'avancement

Les fonctionnalités suivantes sont pleinement opérationnelles :

- Création, édition et suppression d'exercices multi-fichiers
- Système de marqueurs avec extraction et injection automatiques
- Compilation et exécution multi-langages (C, Java, Python)
- Exécution des tests avec comparaison automatique des sorties
- Navigation hiérarchique (modules → cours → exercices)
- Système d'indices avec déblocage progressif
- Déploiement Docker complet

8.2. Difficultés rencontrées

La conception du système de marqueurs et de tests a représenté les principaux défis du projet. Plusieurs itérations ont été nécessaires pour aboutir à une solution satisfaisant les contraintes de simplicité pour l'enseignant.

8.3. Améliorations futures

Plusieurs fonctionnalités n'ont pas pu être implémentées dans le temps imparti :

- **Authentification** : Les modèles de données sont présents, mais l'interface de connexion n'a pas été développée. La priorité a été donnée au cœur fonctionnel.
- **Chat temps réel** : Aurait nécessité l'implémentation de WebSockets, ajoutant une complexité significative.
- **Tableau de bord analytique** : Les données de suivi sont collectées, mais l'interface de visualisation n'a pas été développée.
- **Sandbox d'exécution** : L'exécution dans un conteneur Docker isolé renforcerait la sécurité.

Ces améliorations constituent des perspectives pour une version ultérieure de la plateforme.

Annexe : Base de données

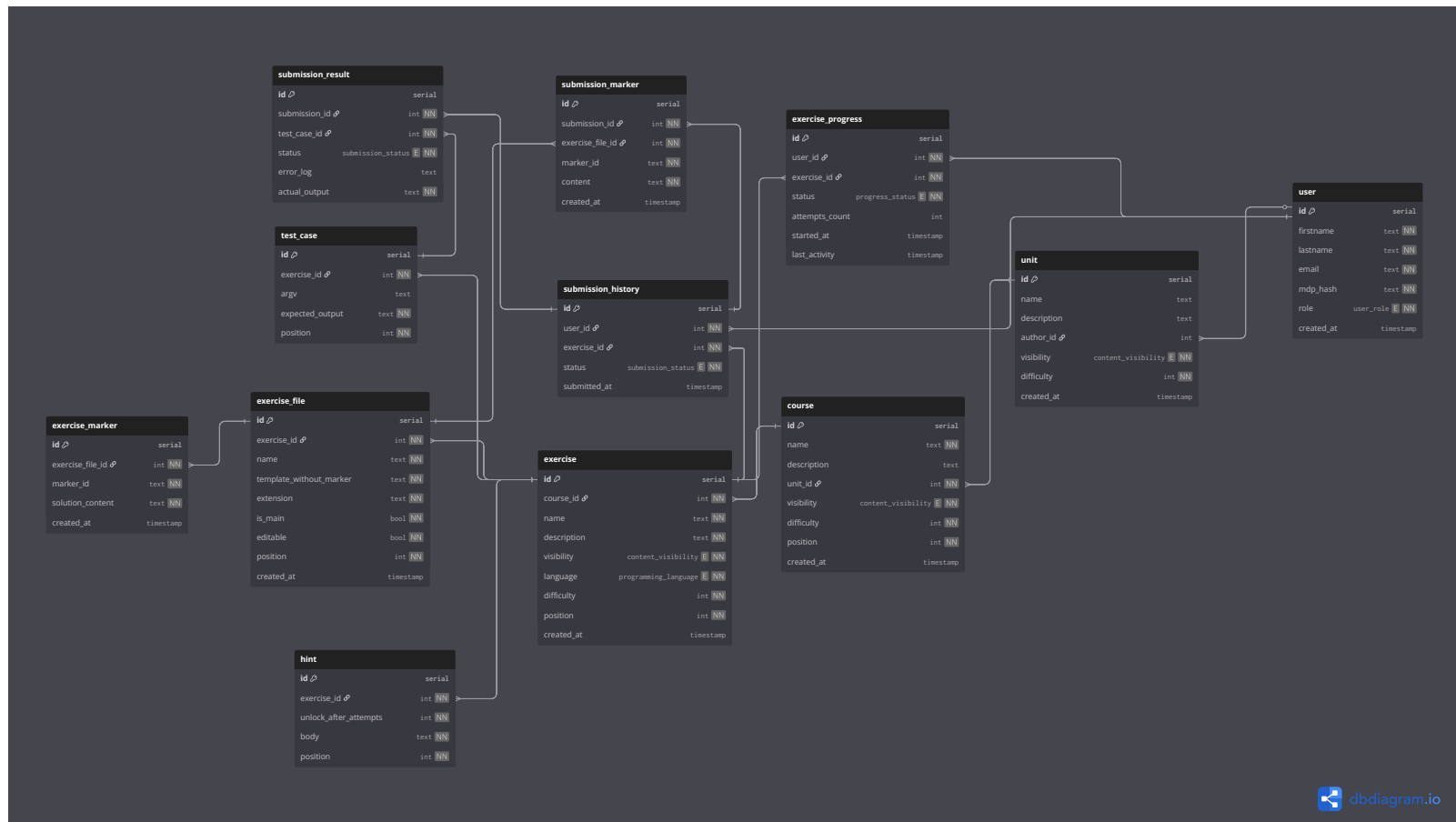


FIGURE 1 : Schéma de la base de données