L.I.D.E

Management de projet

M2 ACDI - Groupe 3

Cahier des charges

Auteurs :
Julien Fontaine
Louis Marchand
Paulin Violette

Professeur responsable :
David Genest

28 septembre 2018

Table des matières

1 Introduction

L'objectif de ce projet est de développer un environnement de développent en ligne accessible depuis un navigateur web classique (l'application doit au moins fonctionner sur Firefox et Chrome). Cet environnement doit permettre l'édition, la compilation et l'exécution de code simple directement depuis le navigateur sans avoir à installer un quelconque outil sur les postes des utilisateurs. Toute la partie concernant la compilation et l'exécution du code est donc effectuée sur un serveur et le résultat rendu à l'utilisateur dans son navigateur. Tout ceci doit être transparent pour l'utilisateur.

Les cibles de l'application sont principalement les étudiants de L1 de l'Université d'Angers, qui dans le cadre des enseignements sont amenés à faire du développement. Dans la mesure où ces étudiants peuvent ne pas disposer d'un environnement de développement sur leur poste personnel (ou même ne possèdent pas d'ordinateur personnel), ils doivent pouvoir continuer leur projet en dehors de l'université. La cible peut être poussée jusqu'aux étudiants de L3 voire M1 si les fonctionnalités proposées par l'application sont suffisamment avancées pour justifier une utilisation par ces étudiants.

L'application a été commencée l'an dernier. Dans l'état actuel, l'application est une preuve de concept : en effet, des changements techniques sont nécessaires pour faire bien fonctionner l'application, mais les bases de l'implémentation sont posées.

2 État actuel du projet

L'application actuelle implémente les fonctionnalités suivantes :

- L'édition de code, avec coloration syntaxique en fonction du langage.
- La gestion de multiples fichiers, ainsi que la possibilité d'importer et d'exporter ces fichiers. La création des fichiers peut être faite à partir de modèles spécifiques au langage utilisé.
- La compilation et l'exécution dans des conteneurs Docker.
- Une interface d'administration permettant d'ajouter des langages de programmation et des modèles de fichiers dans l'application.

2.1 Points à améliorer du projet actuel

2.1.1 Communication Docker/Client

Lors de l'exécution d'un programme depuis l'IDE on peut observer des "reliquats" non-désirés s'afficher lorsque le résultat de l'exécution est renvoyé au client. C'est une anomalie liée à la nature de la communication entre le client et le serveur métier. En effet, le client effectue une requête HTTP au serveur métier pour lancer la compilation et l'exécution de son code, et le résultat lui est renvoyé via une nouvelle requête. Cependant, le temps écoulé entre ces deux requêtes est problématique car il y a potentiellement une perte d'information.

2.1.2 Création automatisée des images Docker

La création automatisée des images Docker est une fonctionnalité intéressante pour les enseignants car elle leur permettra de soumettre leur propre Dockerfile pour personnaliser les environnements en fonction des exercices et des cours. Actuellement, les images Docker des environnements d'exécution (C,C++,Java) doivent être crées à la main sur le serveur métier, ce qui n'est pas commode (car les administrateurs de l'application doivent disposer d'un accès à ce serveur pour déployer de nouvelles images docker) et peu sécurisé.

3 Évolutions prévues

3.1 Évolutions liées aux problèmes de l'application actuelle

3.1.1 Déploiement

Afin de faciliter le déploiement de l'application, il nous semble également important d'ajouter une fonctionnalité permettant d'automatiser le déploiement des images docker liées aux environnements ajoutés par un administrateur.

3.2 Communication entre le client et les programmes exécutés

Afin de résoudre les problèmes existant lors de la communication entre le client et son programme en cours d'exécution, il est nécessaire d'adopter une architecture permettant une communication bi-directionnelle entre le client et le serveur.

3.3 Évolutions souhaitées par le client

Les fonctionnalités suivantes sont souhaitées par le client :

- Une aide avancée à la saisie du code
- Un système de gestion de projets, avec sauvegarde des fichiers sur le serveur. Le support d'outils de build (tel que CMAKE pour le C/C++) serait souhaitable.
- L'ajout d'un débogueur simple (point d'arrêt, avancement pas à pas, consultation des variables)
- Un système permettant de gérer des devoirs, accompagné d'un outil d'analyse permettant d'automatiser des tests sur des devoirs rendus par les étudiants.

4 Solutions proposées

4.1 Architecture

La nouvelle architecture (voir figure 1) se décompose en deux entités distinctes :

- Une première entité qui accueillera toute la partie applicative : authentification des utilisateurs, l'administration/gestion de compte, et la mise à disposition des ressources web (HTML/CSS/JS).
- La seconde entité accueillera la partie métier du projet, à savoir tout ce qui touche à la conteneurisation (compilation/exécution du code) au moyen d'un serveur WebSocket, ainsi que le stockage des fichiers des utilisateurs, l'API de gestion des projets et le module de création automatisée des images Docker des administrateurs.

Nous avons fait le choix d'utiliser un serveur WebSocket pour assurer une communication bi-directionnelle synchrone entre l'utilisateur et le serveur métier. L'inconvénient majeur de l'ancienne architecture était que les fichiers

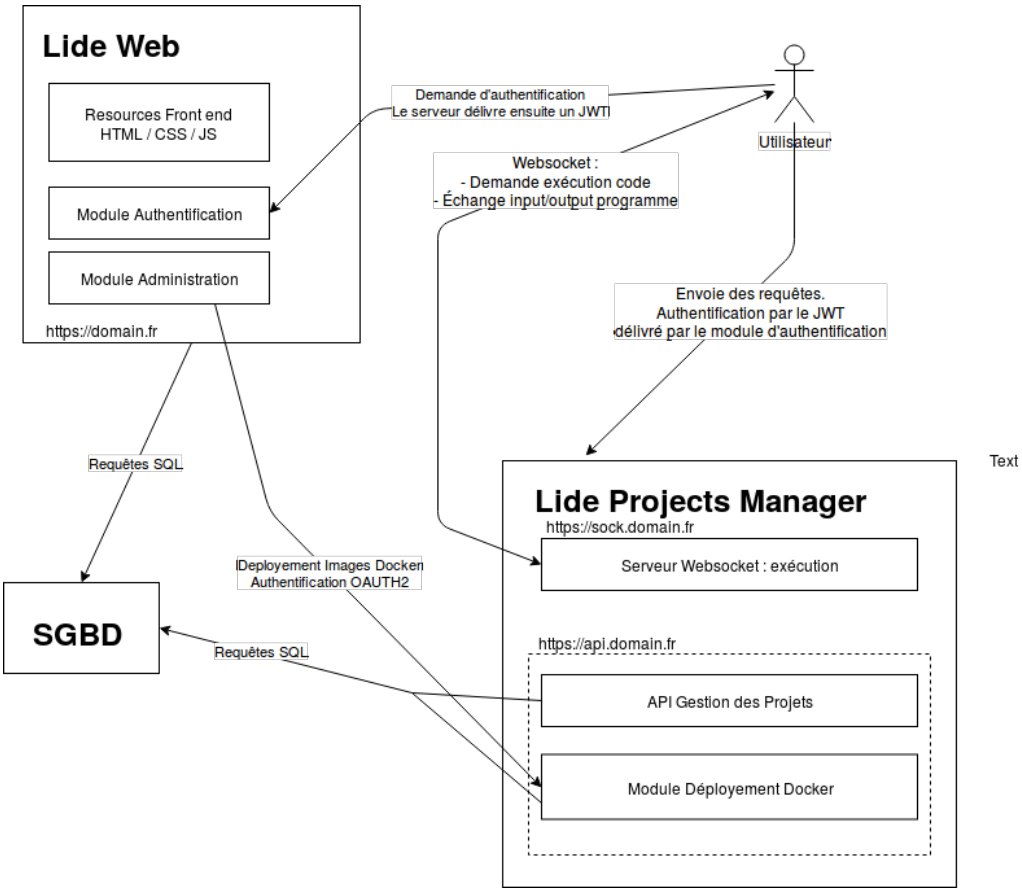

FIGURE 1 – Architecture à mettre en place

sources de l'utilisateur étaient envoyés sur la partie applicative puis étaient récupérés par le conteneur lors de son instanciation au moyen d'un "WGET", ce qui posait des problèmes de sécurité et d'optimisation dans l'utilisation des technologies. Désormais, une fois l'utilisateur authentifié auprès du serveur applicatif, ses fichiers seront directement transférés sur le serveur métier et montés dans son conteneur. Grâce à ces changements dans l'architecture nous avons pu séparer nettement la partie applicative du projet de la partie métier.

4.2 Technologies

4.3 Base de données

Pour permettre l'évolution de l'architecture de l'application, la base de données est modifiée. La table user est conservée mais modifiée. La table `langage` de l'ancien modèle de données est renommée en `environments` et agrémentée de plusieurs colonnes. De nouvelles tables font leur apparition pour permettre la mise en place du système de projet et le stockage de plusieurs fichiers sources (`project_file`, `project`). Plusieurs fichiers modèles associés à un environnement sont désormais disponibles (`file_model`, `environments_file_models`). Un table pour le suivi des déploiements des environnements est ajoutée (`environment_deployement_logs`). Voir la figure 2.

Table `users` : Un utilisateur de l'application est unique. Les utilisateurs sont donc identifiés par un `id` unique. Cette table est gérée en grande partie par l'extension FosUserBundle. Chaque utilisateur possède également une adresse `mail` unique et valide pour qu'il puisse être contacté. Cette adresse permet également de valider l'inscription. Un utilisateur possède un `nom` qui sera utilisé au sein de l'application. Il possède bien sûr un `mot de passe`. Deux colonnes booléean sont utilisés pour :

- `is_admin` : identifier les utilisateurs possédant les droits d'administration sur l'application
- `is_super` : identifier les utilisateurs possédant les droits de super utilisateur.

Le champ json `configuration` contient des valeurs de configuration de l'interface : thème de l'éditeur, de la console, taille de la police, etc...

Table `projects :` Dans cette table sont représentés les projets des utilisateurs dans l'application. Chaque projet est unique et possède donc un `id` unique. Un projet possède un nom (colonne `name`). Un projet appartient à un et un seul utilisateur et correspond à un et un seul environnement : respectivement les colonnes `user_id` et `environmen_id`. Un projet peut être archivé et ainsi ne plus être modifiable : `is archived`. L'archivage d'un projet est définitif, il est néanmoins toujours possible de le dupliquer.

Il peut également être partagé avec d'autres utilisateurs selon la volonté de son propriétaire : cette propriété est représentée dans la colonne booléenne `is_public`. On note également les dates de création et de dernière modification du projet : `created_at` et `updated_at`. La colonne `updated_at` est également mise à jour à chaque modification d'un fichier. Le champ `uuid` est un UUID [1] spécifique au projet. Il est utilisé pour référencer les projets lors de la duplication.

Table `project_files` Un projet peut être composé de plusieurs fichiers répartis dans des répertoires d'une arborescence sur le serveur. Chaque fichier d'un projet est donc renseigné dans la table PROJECT FILES. Un `id` unique lui est attribué. Il est donc rattaché à un et un seul projet : `project id`. Il possède évidemment un nom et son chemin d'accès est également stocké : `name` et `path`. La date de création et de dernière modification sont également stockées : `created at` et `updated at`.

Table `environments :` Pour chaque langage de programmation proposé aux utilisateurs, un environnement est défini. Ces environnements sont paramètrables. Chacun possède un `label` unique et une `description`. La date de création et de dernière modification sont stockées : `created at` et `updated at`. À chaque environnement est associé un docker, stocké sur le serveur WebSocket, qui contient les librairies et les compilateurs correspondants. Ce docker est identifié par son nom unique : `docker name`.

Table `environment_deployement_logs :` Les environnements ont vocation à être déployés par les administrateurs de l'application directement depuis l'interface. À chaque déploiement d'un environnement est associé un log de déploiement. Les administrateurs peuvent consulter via une page les informations relatives à un déploiement spécifique :

1. Universal Unique Indentifier

— `start_date` : l'horodatage du démarrage du déploiement
— `end_date` : l'horadatage de la fin du déploiement
— `status` : le statut du déploiement : cette colonne peut prendre trois valeurs : `running` (le déploiement est démarré mais non terminé), `success` (le déploiement s'est terminé avec succès), `error` (le déploiement à échoué)
— `logs` : un champs json stockant les sorties (`stdout` et `stderrr`) sous forme d'un tableau d'objet, dans l'ordre chronologique.
— `dockerfile_path` : le chemin vers le fichier dockerfile. Dans l'interface, l'administrateur pourra télécharger ce fichier.

Table `file_model` : Des fichiers modèles sont mis à la disposition des utilisateurs selon les environnements. Plusieurs fichiers modèles peuvent être disponibles pour un environnement. Chaque fichier modèle est identifié dans cette table par un `id` unique, il possède un `nom`, une `description`, un `path`, sa date de création et de dernière modification sont également stockées : `created_at` et `updated_at`.

Les modèles de fichiers sont reliés via la table pivot `environments_file_models` avec les environnements. Ainsi, un modèle de fichier peut être réutilisé dans plusieurs environnements (par exemple, un modèle de classe Java peut être réutilisé dans un environnement java 7 et java 8).

Les tables suivantes (voir figure 3) sont destinées à l'organisation de devoirs et les outils d'analyse les accompagnant. Ces tables seront probablement modifiées après une étude plus profonde.

Table `user_user_groups_map` : Cette table sert à faire le lien entre les utilisateurs et les groupes d'utilisateurs. Un utilisateur appartient à au moins un groupe.

Table `user_groups` : Les groupes d'utilisateurs. Par exemple le groupe des utilisateurs étant en L1 MPCIE. Un groupe d'utilisateurs est identifié par un `id` unique. Il possède un nom (colonne `name`, chaîne de 255 charactères maximum) unique et une `description` (texte sans limite de taille).

Table `assignements` : Les enseignants peuvent définir des devoirs (assignements). Un devoir est basé sur un modèle de projet (`template_project_id`)

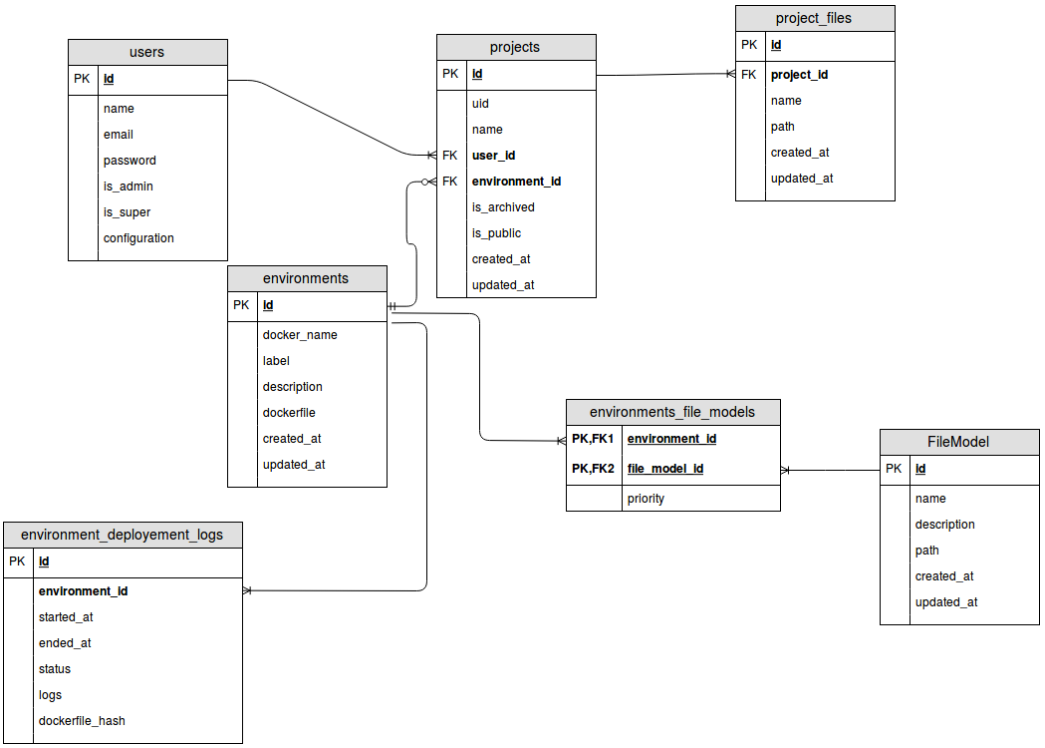

FIGURE 2 – Modèle des données : partie 1

que les utilisateurs pourront récupérer. Un devoir est destiné à un groupe d'utilisateurs (`group_id`). Un devoir a une date de début (`start_time`) et une date de rendu (`due_time`).

Table `assignement_return :` Lorsqu'un utilisateur (`user_id`) participe à un devoir (`assignement_id`) il rend un `projet_id` à une certaine date (temps) : `return_time`.

Table `assignement_tests :` Pour un devoir les enseignants peuvent créer des tests unitaires qui serviront à vérifier les résultats des utilisateurs. Plusieurs tests peuvent être définis pour un devoir (assignement). Un test est attribué à un et un seul devoir : `assignement_id`. Un test prend une entrée et une sortie attendue (la sortie attendue peut être une expression régulière) : `input` et `expected_output`.

Table `assignement_test_results :` Les tests associés à un devoir sont exécutés sur le projet rendu par un utilisateur. Pour chaque test effectué, son résultat est stocké dans une colonne json (`result`). Le format de ce champs reste à définir.

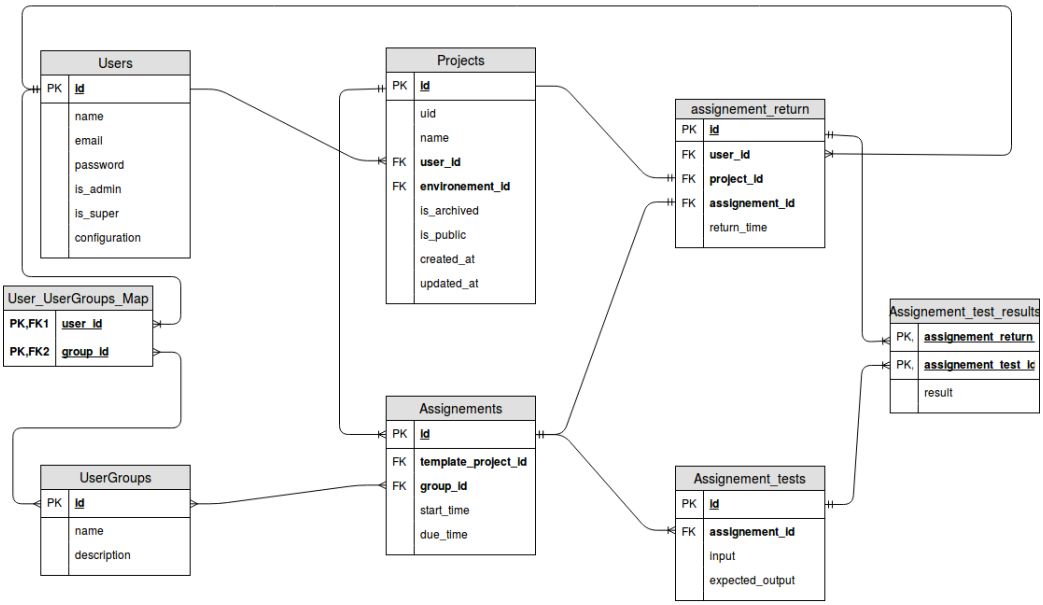

FIGURE 3 – Modèle des données : partie 2

4.4 Partie applicative : Lide Web

L'application LIDE Web reprendra le développement fait l'an dernier.

La partie backend s'appuie sur le framework Symfony. Le bundle FOSUserBundle [2] est utilisé pour la gestion des utilisateurs. La partie administration s'appuie sur le bundle EasyAdmin [3]. Une interface permettant la consultation de la table de logs des déploiements devra être ajouté.

La partie permettant la gestion des conteneurs sera déplacée sur le serveur Websocket.

Ce serveur gère également l'authentification des utilisateurs. Pour cela, il délivre un JWT [4] signé. La charge utile du JWT contient :

— `sub` : id de l'utilisateur sujet du token
— `exp` : timestamp d'expiration du token.
— `user_name` : nom de l'utilisateur

2. FosUserBundle : page sur la documentation officielle Symfony : `https://symfony.com/doc/current/bundles/FOSUserBundle/index.html`

3. Voir la page de la documentation Symfony (en anglais) : `https://symfony.com/doc/master/bundles/EasyAdminBundle/index.html`

4. Json Web Token : voir `https://jwt.io/`

Le bundle symfony `web-token/jwt-bundle` [5] permet la génération et la lecture de JWT intégré à Symfony. Des tests seront nécessaires afin de valider son utilisation.

4.4.1 Frontend

L'interface actuelle de l'application s'appuie sur le framework Bootstrap, et la bibliothèque javascript JQuery. Ces technologies ont l'avantage d'être facilement abordable. Néanmoins, au vu des fonctionnalités demandées et du besoin grandissant d'intéractivité dans l'application, la possibilité de migrer la partie IDE de l'application vers VueJS est une possibilité. Ce framework permettrait d'accélérer les développements et de faciliter l'organisation du code. Mais il a aussi le désavantage d'être plus difficile à prendre en main au départ. Cette possibilité est toujours à l'étude, des discussions incluant les développeurs (élèves de M1) sont nécessaires.

4.5 Serveur métier : Lide Project Manager App

Nous avons pris la décision de rester sur le framework Symfony pour cette application, afin de ne pas augmenter le nombre de technologies utilisées par le projet.

Les utilisateurs se connectant au serveur websocket ou effectuant des requêtes sur l'API de gestion de fichier seront identifiés par le JWT donné par le serveur Lide Web. L'API de déploiement des images docker doit être accessible uniquement par Lide-Web : pour cela, nous pensons utiliser une authentification OAUTH2, mais des discussions et analyses sont encore nécessaires.

Le serveur WebSocket Il s'appuiera sur la bibliothèque Rachet [6], qui permet la création de serveurs websocket en PHP, qui fonctionne grâce à une boucle d'évènement. Le serveur sera lancé via une commande Symfony, ce qui permettra d'utiliser les fonctionnalités de Symfony (principalement l'injection de dépendances). Ce serveur se chargera de lancer l'exécution du code d'un projet dans un conteneur Docker, et ensuite d'assurer la communication avec le programme lancé.

5. Voir `https://web-token.spomky-labs.com/symfony-bundle`
6. Site officiel de Ratchet : `http://socketo.me/`

Gestion des projets Un bundle Symfony sera créé afin de permettre la gestion des projets des utilisateurs. Ce bundle définira une API REST permettant :

- D'accéder aux projets et leur fichiers,
- De modifier les informations relatives au projet,
- De créer des fichiers et répertoires dans un projet, ainsi que de les modifier ou les supprimer,
- De dupliquer un projet existant.

Déploiement automatisé des environnements Les environnements sont composés d'une image docker et d'un script bash permettant le lancement de la compilation et l'exécution des programmes des utilisateurs. Les dockerfiles et scripts sont définis par les administrateurs de l'application via l'interface d'administration

Une route sécurisée avec une authentification OAUTH2 permettra à l'application Lide Web de communiquer avec la partie déploiement des environnements. À cette fin, trois routes seront nécessaires :

- Une route permettant de connaître les images Docker présentent sur le serveur
- Une route permettant de supprimer une image docker
- Une route permettant de lancer le déploiement d'une image docker. Cette route devra prendre l'identifiant de l'environnement à déployer. Un appel à cette route lancera un processus qui lancera la commande construisant l'image, et s'assura de mettre le résultat de cette exécution dans la table de logs de la base de données.

Autocomplétion Cette partie devra faire l'objet de recherche afin de trouver la meilleur solution. Une solution envisagée est l'utilisation de la librairie LLVM, qui pourrait être intégrée au client web en passant par une compilation en WebAssembly.

5 Lide Web Server

5.1 Interfaces Utilisateurs Étudiants

5.1.1 Interface de gestion de projet

Cette interface (voir figure 4 et 5) permet à l'utilisateur de visualiser et de gérer ses projets. Elle permet d'accéder à un projet existant, de créer un nouveau projet ou de cloner un projet existant (soit un projet lui appartenant ou un projet public d'un autre utilisateur).

Lorsqu'un utilisateur n'a aucun projet créé, nous lui proposons un lien vers un manuel d'utilisation. Un manuel d'utilisation existe déjà dans la version produite l'an dernier (sous format PDF). Il sera nécessaire d'extraire et de mettre à jour les informations de ce manuel d'utilisation, afin de le proposer sous la forme d'une page web intégrée à l'application.

Création d'un nouveau projet Chaque utilisateur peut créer autant de projet qu'il le souhaite, dans la limite de l'espace de stockage alloué par les administrateurs. Chaque projet est défini par un nom (unique par utilisateur), et d'un environnement d'exécution pour le code. Cet environnement peut être changé plus tard si nécessaire, via l'interface de configuration du projet sur la page de développement. Une fois un projet créé, l'utilisateur est redirigé vers la page de développement.

L'utilisateur peut également cloner des projets existants. Pour cela, il doit disposer d'un code correspondant au projet. Il peut ensuite saisir ce code. Après la saisie du code, des informations sur le projet (nom, environnement) lui sont affiché, ou alors si le code donné n'est pas valide, un message d'erreur. Si le code est valide, il peut ensuite cloner le projet, ce qui l'amènera sur la page de développement.

5.2 Interfaces d'administration

Les interfaces d'administration ne changent pas dans leur design et organisation par rapport à l'an dernier. Les formulaires seront adaptés afin de correspondre aux modifications du modèle de données. Un bouton permettant de déployer un environnement sera ajouté, ainsi qu'une section permettant de consulter les logs des déploiements des environnements.

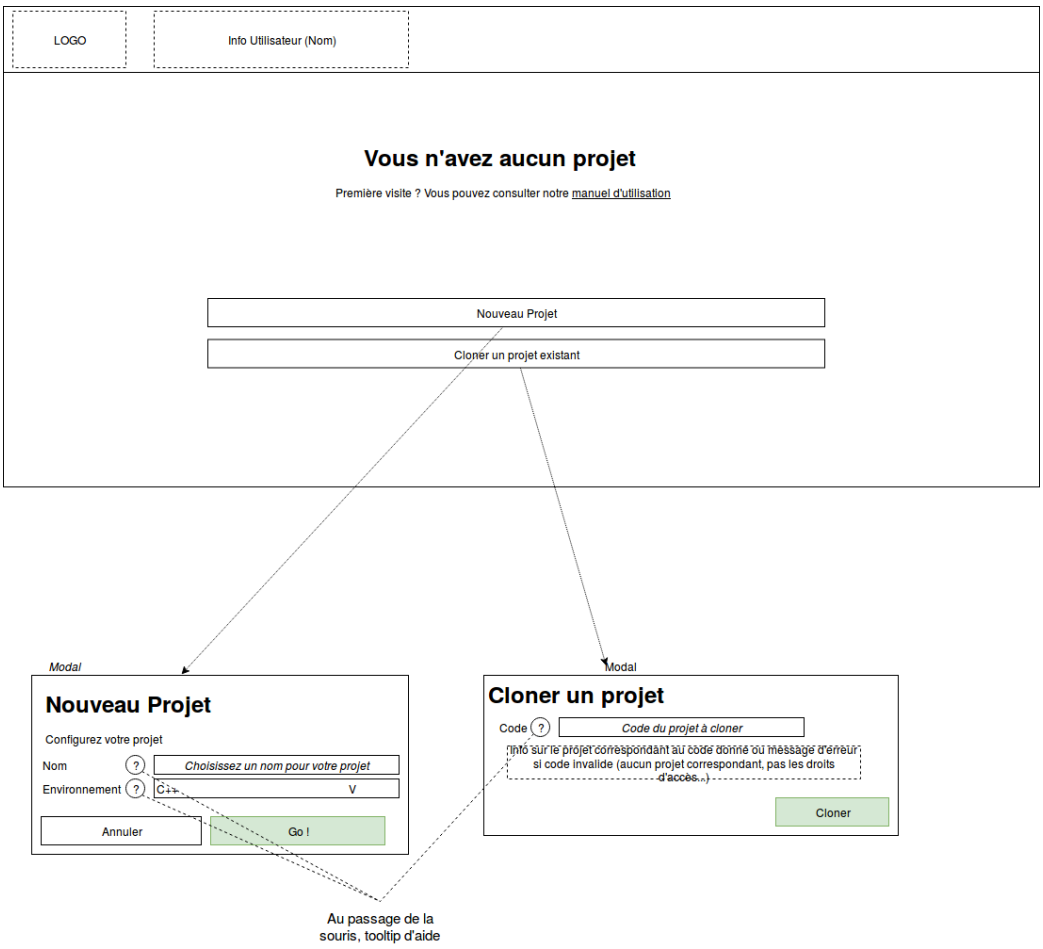

FIGURE 4 – Interface de gestion de projet pour un utilisateur n'ayant aucun projet

6 Application des gestions de projet : Lide Project Manager

6.1 Sécurité de l'exécution des programmes

Cette application va exécuter du code provenant d'utilisateurs. Il est nécessaire de se protéger d'éventuelles attaques. Pour cela, les programmes sont exécutés dans des conteneurs, les isolant du reste du système. L'exécution est également limitée en temps (le processus est automatiquement tué après un

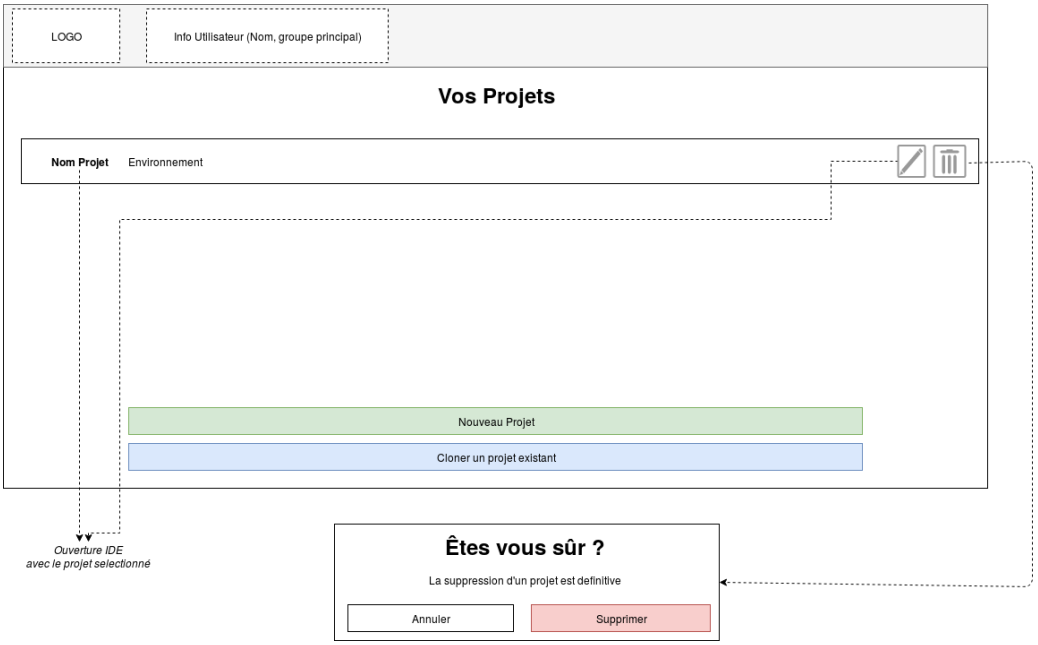

FIGURE 5 – Interface de gestion de projet pour un utilisateur ayant des projets existants

temps défini dans les paramètres de l'application), en mémoire allouée, et en nombre de CPU accessible.

Nous avons également identifié les vecteurs d'attaques suivants :

— *Fork Bomb* : un programme lançant des processus à l'infini. Pour cela, il faut limiter les nombres de processus pouvant être exécutés dans le conteneurs.

— Surcharge de la sortie : un programme écrivant un nombre important de données sur la sortie pourrait causer une surcharge de la mémoire. Pour cela, il faut limiter la taille de la sortie.

— Écriture d'un grand nombre de fichiers : un programme pourrait créer un grand nombre de fichiers, ou de fichiers de grande taille et saturer la mémoire des serveurs. Pour cela, l'espace de stockage est limité pour chaque utilisateur.

6.2 API pour la gestion des projets

Toutes les routes renvoient une réponse JSON. Les dates renvoyées dans les réponses sont au format ISO 8061.

Ces spécifications peuvent être modifiées. Une version actualisée des spécifications est disponible à l'adresse suivante : `https://gitlab.com/ua-lide/specs`

L'api de gestion de projets intègre les routes suivantes :

6.2.1 Récupérer tous les projets selon des filtres

`GET /projects?{query}`

Avec les paramètres suivants :

— `user_id` : id de l'utilisateur propriétaire du projet
— `name` : nom du projet
— `page` : numéro de la page (si la pagination est implémentée)

Seul les projets visibles par l'utilisateur effectuant la requête (c'est à dire ses projets et les projets publics des autres utilisateurs) sont retournés.

La réponse retournée est de la forme :

```
{
    "data": [
        {
            "id": "1233",
            "uuid": "ad4e9634-c278-11e8-a355-529269
               fb1459"
            "name": "project_name",
            "user_id": "1",
            "environnement_id": "{environnement_id}"
               ,
            "is_public": "true", //or "false"
            "is_archived": "false", //or "false"
            "created_at": "1977-04-22T01:00:00-05:
               00",
            "updated_at": "1977-04-22T01:00:00-05:
               00"
        },
        {
```

```
            "id": "1234",
            "uuid": "76025f19-7101-47d8-b9ee-
              c27e9671b798"
            "name": "project_name",
            "user_id": "1",
            "environnement_id": "1",
            "is_public": "true",
            "is_archived": "false",
            "created_at": "1977-04-22T01:00:00-05:
              00",
            "updated_at": "1977-04-22T01:00:00-05:
              00"
        },
        ...
    ],
    "meta": {
        "page": "2",
        "next": "/project?page=1&{filtres}",
        "previous" : "/project?page=3&{filtres}",
    }
}
```

La pagination n'est pas prioritaire : elle serait néanmoins souhaitable si on autorise cette requête sans préciser d'utilisateurs, le nombre de projets présents dans l'application pouvant être très important.

6.2.2 Création d'un nouveau projet

`POST /api/project`

Avec le contenu de requête suivant :

```
{
    "data": {
        "name": "new_project_name",
        "envionnement_id": "{environnement_id}",
        "is_public": false
    }
}
```

Les champs `name` et `environnemnt_id` sont requis. Le champ `environment_id` doit réferencer un environnement existant. Le champ `is_public` prend la valeur `false` par défaut. Le nom d'un projet est unique par utilisateur. Si les valeurs données pour ces champs ne sont pas valides, une réponse json contenant un tableau d'objet décrivant les erreurs est envoyé par le serveur. Cette réponse a un code HTTP 422.

Si la requête est valide, la réponse retournée est de la forme (code HTTP 200) :

```
{
    "data": {
        "id": "12345",
        "uuid": "b9653515-7beb-425a-aa4f-2
           a7840124a0e",
        "name": "new_project_name",
        "user_id": "321",
        "environnement_id": "1",
        "is_public": true,
        "is_archived": false,
        "created_at": "1977-04-22T01:00:00-05:00",
        "updated_at": "1977-04-22T01:00:00-05:00"
    }
}
```

6.2.3 Cloner un projet existant

Le client effectue la requête :

```
POST /project/clone/{uuid projet}
```

Si le projet n'est pas accessible par l'utilisateur effectuant la requête, un code d'erreur 403 est renvoyé.

Sinon le serveur effectue la duplication du projet (informations et fichiers), puis renvoie la même réponse que pour une création.

6.2.4 Modifier les informations d'un projet

```
PUT /project/{idProject}
```

Seul l'utilisateur propriétaire du projet est autorisé à utiliser cette route.
Avec le contenu de requête suivant :

```
{
    "data": {
        "name": "updated_project_name",
        "envionnement_id": "{environnement_id}",
        "is_public": false, //or "true",
        "is_archived": true
    }
}
```

Aucun champ n'est requis. Les champs non renseignés garde la valeur persistée en base de données.

Le retour est le même que pour la création.

6.2.5 Supprimer un projet

```
DELETE /project/{idProject}
```

Seul l'utilisateur propriétaire du projet est autorisé à utiliser cette route.
Le serveur doit retourner un code http 200 après la suppression du projet.
La suppression d'un projet entraîne la suppression de tous les fichiers liés.

6.2.6 Récupérer la liste des fichiers d'un projet

```
GET /project/{idProject}/files
```

Si l'utilisateur n'est pas autorisé à accéder au projet (c'est à dire projet non public n'appartenant pas à l'utilisateur), le serveur doit retourner un code d'erreur 403 (accès refusé).

Si le projet n'existe pas, le serveur doit retourner un code d'erreur 404.

Le retour doit être de la forme :

```
{
    "data": [
        {
            "id": "1",
            "name": "file_name",
            "path": "/path/in/project",
```

```
            "created_at": "YYYY-MM-DD HH:mm:ss",
            "updated_at": "YYYY-MM-DD HH:mm:ss"
        },
        {
            "id": "3",
            "name": "file_name2",
            "path": "/path/in/project",
            "created_at": "YYYY-MM-DD HH:mm:ss",
            "updated_at": "YYYY-MM-DD HH:mm:ss"
        },
        ...
    ]
}
```

Le contenu des fichiers n'est pas retourné, afin d'éviter des réponses trop lourdes.

6.2.7 Récupérer le contenu d'un fichier

`GET /api/project/{idProject}/files/{idFile}`

Si l'utilisateur n'est pas autorisé à accéder au projet (c'est à dire projet non public n'appartenant pas à l'utilisateur), le serveur doit retourner un code d'erreur 403 (accès refusé).

Si le projet ou fichier n'existe pas, le serveur doit retourner un code d'erreur 404.

Le retour doit être de la forme :

```
{
    "data": {
        "id": "1",
        "name": "file_name",
        "path": "/path/in/project",
        "content": "contenu du fichier",
        "created_at": "YYYY-MM-DD HH:mm:ss",
        "updated_at": "YYYY-MM-DD HH:mm:ss"
    }
}
```

6.2.8 Ajouter un fichier à un projet

`POST /project/{idProject}/files`

Avec un contenu de requête de la forme :

```
{
    "data":{
        "name": "file_name",
        "path": "/path/in/project",
        "content": "content"
    }
}
```

Tous les champs sont requis. Le couple (`name`, `path`) est unique dans un projet.

Si la requête est valide, le fichier est créé et la réponse est la même que pour la récupération d'un fichier (cf 6.2.7).

6.2.9 Modifier un fichier

`PUT /project/{idProject}/files`

Si l'utilisateur n'est pas autorisé à accéder au projet (c'est à dire projet non public n'appartenant pas à l'utilisateur), le serveur doit retourner un code d'erreur 403 (accès refusé).

Le contenu de la requête doit être de la forme :

```
{
    "data":{
        "name": "file_name",
        "path": "/path/in/project",
        "content": "content"
    }
}
```

Aucun champ n'est requis. Les champs manquants seront traités avec la valeur existante sur le serveur. Les champs présents ont les mêmes règles de validations que pour la création. Si la requête est valide, les données sont modifiées et la réponse est la même que pour une requête GET.

6.2.10 Supprimer un fichier

`DELETE /project/{id projet}/files`

Si l'utilisateur n'est pas autorisé à accéder au projet (c'est à dire projet non public n'appartenant pas à l'utilisateur), le serveur doit retourner un code d'erreur 403 (accès refusé).

Après la suppression du fichier, le serveur doit renvoyer un code 200.

6.3 Serveur websocket pour l'exécution

Le serveur websocket permet à l'utilisateur de lancer ses programmes, d'en recevoir la sortie et d'y envoyer des entrées.

L'authentification se fait via un JWT (émis par Lide Web) inclus dans la requête HTTP de connexion au serveur websocket (pour rappel, une connexion websocket est une requête http qui change de protocole). La reqûete http se fait en SSL sur une url de la forme :

`GET /ws/{id projet}?jwt={JWT token}`

L'id du projet doit correspondre à un projet de l'utilisateur (sinon la connexion est refusée).

Les messages envoyés sont des objets json ayant deux champs : un champ `type` indiquant le type de message, et un champ `data` contenant des informations spécifiques au type de message.

6.3.1 Lancer l'exécution

L'utilisateur envoie un message de la forme :

```
{
    "type": "execute",
    "data": {
        "compile_options": "",
        "launch_options": "",
    }
}
```

Le champ `compile_options` contient les options de compilation à passer au compilateur : le champ `launch_options` contient les arguments à passer au programme.

Lorsque le serveur reçoit ce message, il lance un conteneur exécutant le code du projet défini par l'utilisateur à la connexion, seulement si l'utilisateur n'a pas déjà un conteneur lancé.

Si l'utilisateur a déjà un conteneur en cours d'exécution, le message suivant est envoyé :

```
{
    type: "status",
    data: {
        "is_running": true
    }
}
```

6.3.2 Obtenir l'état du programme

Le client peut demander au serveur l'état de son programme via le message :

```
{
    type: "get_status",
    data: {
        "is_running": true
    }
}
```

Le serveur renvoie un message de la forme :

```
{
    type: "status",
    data: {
        "is_running": true
    }
}
```

Le champ `is_running` prend la valeur `true` si l'utilisateur a un programme en cours d'exécution, ou `false` dans le cas contraire.

Des informations supplémentaires (tel que le temps d'exécution) pourraient être ajoutées.

6.3.3 Envoi de la sortie standard (stdout) du programme

Le serveur envoie un message de la forme :

```
{
    "type": "stdout"
    "data": {
        "output": "output programme",
    }
}
```

6.3.4 Envoi de la sortie d'erreur (stderr) du programme

Le serveur envoie un message de la forme :

```
{
    "type": "stdout"
    "data": {
        "output": "output programme",
    }
}
```

6.3.5 Envoi d'une entrée (stdin) au programme

Le client envoie un message de la forme :

```
{
    "type": "input",
    "data": {
        "input": "input utilisateur"
    }
}
```

Le message est ignoré si l'utilisateur à l'origine du message n'a pas de conteneur en cours d'exécution.

6.3.6 Fin de l'exécution

Le serveur envoie un message de la forme :

```
{
    "type": "end"
    "data": {
        "return": "0"
```

```
    }
}
```

Le champ `return` est le code retourné par le programme.

Des informations supplémentaires (tel que le temps d'exécution) pourraient être ajoutées.

6.3.7 Forcer l'arrêt

Pour forcer l'arrêt d'un programme, le client envoie le message :

```
{
    "type": "force_stop",
    "data": {}
}
```

Le serveur va ensuite forcer l'arrêt du conteneur, et enverra le message de fin d'exécution une fois que l'action sera effectuée. Le message est ignoré si l'utilisateur n'a pas de programme en cours d'exécution.