



Université de Montpellier
Master informatique

UE : TER

**Projet : REFACTORING D'APP WEB SUIVANT LE MODÈLE
DES MICRO-SERVICES**

Rapport - GROUPE LAM

Auteurs :

Meryam RHADI (GL)
Lamyae KHAIROUN (GL)
Awa SECK (IASD)

Encadrants :

Chouki Tibermacine
Vincent Berry

Remerciements

Avant tout développement, il apparaît opportun de commencer ce rapport de ter par des remerciements.

Nos remerciements les plus sincères vont être adressés à Monsieur Chouki Tibermacine ainsi qu'à Monsieur Vincent Berry nos encadrants pour leurs directives, leurs conseils, leur soutien tout au long de notre projet mais également leur disponibilité pour l'aboutissement de notre travail.

Nous vous présentons, chers messieurs, notre reconnaissance mais aussi un immense joie d'avoir travailler avec vous tout au long du projet.

Résumé

Le présent rapport est le fruit d'un travail effectué au sein de la faculté de sciences de Montpellier, dans le cadre de l'unité d'enseignement TER (Travail d'Étude et de Recherche) du programme du Master 1 Informatique.

L'objectif de ce projet est de proposer un refactoring de l'application suivant une architecture micro-services.

Notre projet vise aussi à étudier les différentes méthodes de migration d'une application Web monolithique vers une architecture en micro-service, ainsi qu'à étudier les diverses approches pour trouver des micro-services candidats en s'appuyant sur le schéma de la base de données, particulièrement celle de "Unsupervised Machine Learning" nommé Clustering.

Abstract

The present report is the result of an effort carried out at the Faculty of Sciences in Montpellier, in the TER module of the Master in Computer Science.

This project aims to study the different methods of migrating from a monolith web application to a micro-service architecture, and choose one of them, and to familiar with the clustering approach to find one or more micro-services candidates based on the database schema.

ملخص

هذا التقرير هو ثمره مجهودات التي تم تنفيذها في جامعه العلوم بمونبيليه بماده البحث والدراسه في اطار برنامج الماستر

للمعلومات

هذا المشروع يهدف الى دراسه مختلف الطرق للهجره من تطبيق يضم الكود في مشروع واحد الى هندسه تنقسم الى عدده وحدات وكان علينا اختيار واحده من بين هذه الطرق وايضا التعود على الكلاستورن لايجاد هندسه الوحدات التي يمكننا القيام بها بالاعتماد على قاعده البيانات

Mots-clé : architecture micro-services, monolithe, migration, clustering, docker, gitlab, requête, refactoring.

Sommaire

Sommaire	2
1 L'application web ShellOnYou	9
1.1 Objectif de l'application initiale	9
1.2 Structure de l'application	9
1.3 Base de données	10
1.4 Analyse de besoin et problématique	11
2 Organisation du travail	12
2.1 Le diagramme de Gant	12
2.2 Les tâches réalisées durant le projet	13
2.2.1 Les tâches de l'étude de l'architecture et la migration vers micro-services	13
2.2.2 Les tâches de Clustering	14
2.2.3 Les tâches globales	14
2.3 Méthode de communication	14
3 Technologies et l'architecture utilisées	15
3.1 Technologies utilisées	16
3.1.1 Docker et docker-compose	16
3.1.2 Visual studio	16
3.1.3 Gitlab	16
3.1.4 Node	16
3.1.5 WSL ubuntu	17
3.1.6 Axios	17
3.1.7 Google Drive	17
3.1.8 Google Colaboratory	17
3.1.9 Postman	17
3.1.10 Express	17
3.1.11 Overleaf	18
3.2 Schéma d'architecture entre les différentes parties de l'ap- plication	18
3.2.1 Diagramme de la façon de communication en utili- sant <i>axios</i>	18

3.2.2	Diagramme de la façon de communication en utilisant <i>APIGateway</i>	19
4	Prise en main et installation des technologies	20
4.1	Installation de WSL	21
4.2	Installation et configuration de Docker pour WSL 2	21
4.3	Installation de ShellOnYou	21
5	Migration vers les microservices	23
5.1	Monolithe	24
5.2	Architecture micro-service	25
6	Partie technique : migration du monolithe vers les microservices	26
6.1	Création du micro-service <i>MSExercice</i>	26
6.1.1	Objectif de <i>MSExercice</i>	26
6.1.2	Création et configuration de conteneur	27
6.1.3	Utilisation d'axios	31
6.2	Création du microservice <i>API Gateway</i>	36
6.2.1	Objectif de <i>APIGateway</i>	36
6.2.2	Création et configuration de conteneur	37
6.2.3	Exemple de la nouvelle version de l'application avec <i>API Gateway</i>	39
7	Découpage en clusters	45
7.1	Construction et pré-traitement des données	45
7.1.1	Construction	45
7.1.2	Pré-traitement	46
7.2	Clustering	46
7.2.1	Mesure de similarité	46
7.2.2	Vectorisation des documents	48
7.2.3	Identification du nombre de clusters	48
7.2.4	Méthode de clustering	49
7.3	Inférence de nom du micro-service	50
7.4	Conclusion	50
8	Conclusion	52
8.1	Analyse générale	52
8.2	Problèmes rencontrés	52
8.3	Perspectives	53

Liste des figures

1.1	Schéma de base de données de l'application	10
2.1	Diagramme de Gantt de projet	12
2.2	Les canaux utilisée pour la communication	14
3.1	Schéma de l'architecture avec axios	18
3.2	Schéma de l'architecture avec API Gateway	19
4.1	Page d'accueil	22
5.1	Exemple illustratif de l'architecture monolithique VS l'architecture microservices[5].	23
5.2	Architecture du monolithe [12].	24
5.3	Exemple d'architecture micro-service [12].	25
6.1	Le fichier de configuration ConfMsEx.js	27
6.2	Extrait du code du fichier package.json	28
6.3	Extrait de code du fichier de configuration Dockerfile	28
6.4	Définition du service MSExercice dans le fichier yml	29
6.5	Extrait de code du serveur de MSExercice	30
6.6	Test de la méthode GET qui renvoie le formulaire de création d'un exercice.	31
6.7	Exemple d'utilisation d'axios pour rediriger vers MSExercice dans nodedock	32
6.8	Route de détail d'un exercice dans MSExercice	33
6.9	La méthode read() de controller exercise (extrait de code 1)	33
6.10	La méthode read() permet de controller un exercice (extrait de code 2).	34
6.11	Liste des exercice affiché depuis le monolithe	35
6.12	Detail de l'exercice nommé <i>jour 13</i>	36
6.13	Définition du service Gateway dans le fichier yml	37
6.14	Des lignes de fichier index de Gateway	38
6.15	Page d'accueil de l'application avec API Gateway	39
6.16	Liste des exercices dans le microservice MSExercice	40

6.17	Formulaire de modification d'un exercice	41
6.18	Liste des exercices dans <i>MSExercice</i>	42
6.19	Détails de l'exercice nommé "jour 11"	42
6.20	Formulaire de creation d'un exercice dans <i>MSExercice</i> . .	43
6.21	Création d'un nouveau exercice nommé "Exercice 15" . . .	44
7.1	Méthodes de prétraitement	47
7.2	Résultat de Elbow	49
7.3	Les différents clusters	50

Introduction

Dans le cadre de notre projet d'étude TER (Travail d'Etude et de Recherche), nous sommes chargés de faire la refactorisation d'une application web monolithique en une application sous forme de micro-services.

Le but de ce travail est de faire une partie d'étude pour s'approfondir dans la notion de l'architecture microservice avec ces différentes méthodes de migration, ainsi de choisir une de ces dernières pour réaliser la refactorisation du monolithe vers les microservices, ainsi que l'implémentation d'une méthodologie de refactoring en utilisant une approche du machine learning non supervisé notamment le *clustering* pour découper la base de données en différents clusters afin que chaque cluster soit un micro-service candidat.

L'application en question est écrite en *Node.js*, adossée par une base de données Postgres et hébergée dans des conteneurs Dockers et déployée sur l'adresse <https://shellonyou.fr>. Elle permet aux étudiants de se former au Shell Linux par la pratique (Learning by doing) en mettant en avant des compétences gagnées au fur et à mesure des exercices réalisés.

Dans les lignes qui suivent, nous allons d'abord analyser cette application en donnant sa structuration, ensuite nous parlerons de comment nous avons organisé le travail, et après nous explicitons les différentes technologies utilisées pour faire le refactoring de cette dernière, puis nous aborderons la prise en main et installation des outils que nous avons travaillé avec, ensuite nous allons parler des inconvénients des applications monolithiques qui emmènent à utiliser des micro-services avant d'entamer le vif du projet à savoir la création des micro-services et le découpage en clusters.

Enfin, nous terminons par une analyse globale ainsi que les difficultés rencontrés avant de conclure.

Chapitre 1

L'application web ShellOnYou

1.1 Objectif de l'application initiale

Chaque année, des milliers d'étudiants dans le monde souhaitent au terme de leur formation, continuer leur parcours dans le domaine de l'informatique. À l'issue de ce parcours, leurs niveaux sont hétéroclites, certains de ces parcours mettant l'accent sur des connaissances théoriques, d'autres purement techniques ou alors n'approfondissant pas assez certains domaines.

Pour faire face à ce problème, il est important de connaître le niveau de chaque étudiant, lorsqu'il intègre un parcours et de pouvoir lui fournir un outil permettant la consolidation de ses compétences par la pratique. C'est pourquoi le réseau *Polytech* a décidé de mettre en place une plateforme, l'application **Shell On You**, qui met à disposition des étudiants divers exercices dans l'optique d'évaluer leur capacités mais aussi de favoriser l'acquisition de compétences sur les systèmes d'exploitation UNIX (ou ses variantes).

1.2 Structure de l'application

Par rapport au back-end de l'application *Shell On You*, il est monolithique, s'organise en adoptant l'architecture *Modèle-Vue-Contrôleur* (MVC), pour séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts.

L'application est programmée avec *Node.js* et le framework *Express*.

Les vues de *Shell On You* sont écrites en **EJS** (Embedded JavaScript templates) qui est un langage de modèle simple qui permet de générer un balisage HTML avec du JavaScript.

L'application en question est adossée à une base de données *Postgres*

1.4 Analyse de besoin et problématique

D'une part cette application réponds parfaitement au besoin dont elle a été créée, mais d'autre part l'application suit une architecture monolithique dans laquelle tous les modules pertinents et services proposés sont regroupés sous la forme d'une seule unité, donc avec le temps il sera difficile et complexe de gérer et faire évoluer cette dernière, à cause du nombre de lignes de codes qui augmente. Ainsi que les différentes parties du système ne peuvent pas être mises à l'échelle indépendamment, car elles sont liées par un couplage fort.

Suite à cette problématique, il a été proposé de travailler avec une architecture en micro-services, afin d'obtenir une application sous forme de services autonomes, ayant un couplage faible et peuvent évoluer indépendamment en fonction des besoins des utilisateurs.

Chapitre 2

Organisation du travail

2.1 Le diagramme de Gant

C'est un outil utilisé en gestion de projet, qui permet de visualiser dans le temps les diverses tâches composant un projet, la figure 2.1 représente le diagramme de gant qu'on a réalisé pour organisé ce projet.

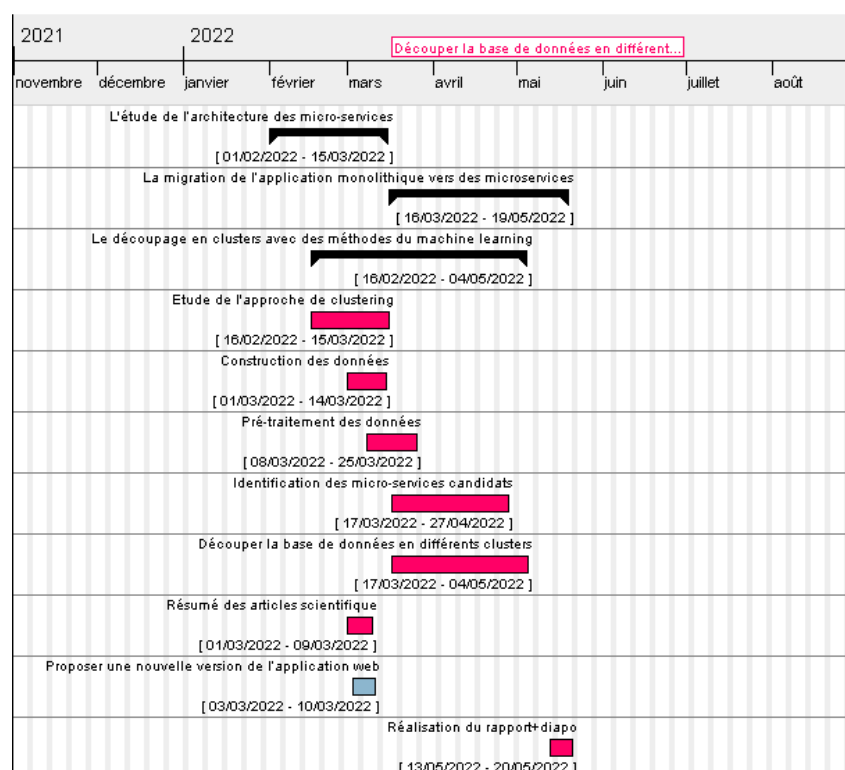


Figure 2.1. Diagramme de Gantt de projet

En effet, le diagramme de Gantt fournit un planning prévisionnel. Nous l'avons établi au début de projet et nous sommes fixé comme objectif de le respecter au maximum durant le projet. Alors nous avons pu faire toutes les tâches qui concernent tout ce qui est étude, ainsi que nous avons fait les tâches de la partie technique, qui s'agit de la mise en place de la réfactorisation de l'application et la répartition de la base de données en clusters. Mais il y a toujours des perspectives qui peuvent être réalisées dans le futur comme indiqué dans la section 8.3.

2.2 Les tâches réalisées durant le projet

Le travail a été scindé en deux grandes parties :

- une partie directement liée au Génie Logiciel : l'étude de l'architecture des micro-services (MSA) de l'application **ShellOnYou** monolithique et sa migration vers des microservices suivant un découpage décidé manuellement. Ce travail a été fait par **Meryam RHADI** et **Lamyae KHAIROUN**.
- une partie plus liée au Machine Learning : la proposition de micro-services pour **ShellOnYou** par un découpage en clusters. Ce travail a été fait par **Awa SECK**

2.2.1 Les tâches de l'étude de l'architecture et la migration vers micro-services

Cette partie comporte plusieurs sous-tâches qui sont :

- Étude bibliographique des micro-services.[10]
- Choix des types d'architecture pour MSA.
- Étude des méthodes de communication des microservices.
- S'initier à la technologie Docker et l'orchestration de conteneurs par Docker-Compose.
- Analyse des avantages et inconvénients de l'architectures monolithe et microservice.
- Proposition d'un découpage adéquat au refactoring de l'application visée.
- Création du microservice **MSExercice**, qui communique avec le monolithe en utilisant la méthode axios.
- Création du microservice **APIGateway**.
- Introduction d'une API gateway
- Mise en place de communication entre le monolithe et le microservice par l'API gateway.

2.2.2 Les taches de Clustering

Cette phase est divisible en deux étapes : identification du micro-service et refactoring détaillée dans le chapitre 7.

2.2.3 Les tâches globales

- Résumé des articles scientifiques : **Meryam RHADI** [9] [1], **Lamyae KHAIROUN** [7], **Awa SECK** [17].
- Proposer l'architecture adéquate pour le refactoring.
- Proposer une nouvelle version de l'application web.
- Réalisation du diagramme de gant.
- Réalisation du rapport de fin de projet et des diapositives de soutenance.

2.3 Méthode de communication

Pour échanger avec nos encadrants, nous utilisons le serveur *Mat-termost*, avec des canaux nommés par thème, comme il est représenté dans la figure 2.2 , et les réunions se font soit en distanciel sous *Zoom* soit en présentiel à Polytech (Bat 31), pour déterminer l'avancement du projet et/ou les éventuelles difficultés rencontrées.

En ce qui concerne les membres groupe, nous nous réunissons en présentiel puisqu'il n'y a pas la contrainte de distance.

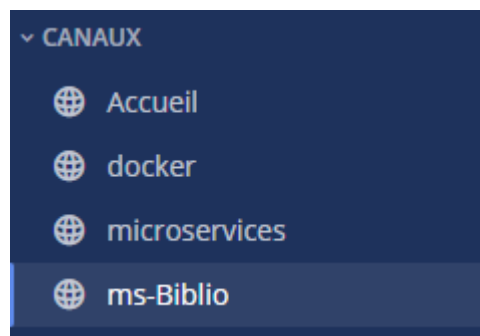


Figure 2.2. Les canaux utilisée pour la communication

Chapitre 3

Technologies et l'architecture utilisées



Ce projet nous a donné l'occasion de découvrir et utiliser plusieurs technologies.

3.1 Technologies utilisées

3.1.1 Docker et docker-compose

Docker est une plateforme, qui permet d'empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel serveur, qu'il que soit physique ou virtuel[2].

Docker Compose est un outil qui permet de décrire (dans un fichier YAML) et gérer (en ligne de commande) plusieurs conteneurs comme un ensemble de services inter-connectés.

3.1.2 Visual studio

Visual Studio est un ensemble complet d'outils de développement permettant de générer des applications web ASP.NET, des services web XML, des applications bureautiques et des applications mobiles [16].

3.1.3 Gitlab

GitLab est une plateforme de développement collaborative open source éditée par la société américaine du même nom. Elle couvre l'ensemble des étapes du DevOps.

Se basant sur les fonctionnalités du logiciel Git, elle permet de piloter des dépôts de code source et de gérer leurs différentes versions. Son usage est particulièrement indiqué pour les développeurs qui souhaitent disposer d'un outil réactif et accessible [8].

3.1.4 Node

Node JS est une technologie qui permet de faire du JavaScript côté serveur. Node permet de développer et déployer des applications.

Il est principalement utilisé pour créer des applications web qui permettent les échanges de données, par exemple une application de chat, des sites de streaming multimédia, etc [11].

3.1.5 WSL ubuntu

Le Windows Subsystem for Linux (WSL) est une variante d'Ubuntu proposée officiellement par Microsoft et Canonical, et qui se déploie nativement sur Windows 10 (uniquement) au moyen de la couche de compatibilité permettant d'exécuter des fichiers ELF (binaires exécutables) pour Linux à partir d'un système Windows [14].

3.1.6 Axios

Axios est une bibliothèque JavaScript fonctionnant comme un client HTTP. Elle permet de communiquer avec des API en utilisant des requêtes. Comme avec les autres clients HTTP, il est possible de créer des requêtes avec la méthode POST, GET, PUT, et DELETE [3].

3.1.7 Google Drive

Google Drive est un service de stockage et de partage de fichiers dans le cloud lancé par la société Google. Google Drive, qui regroupe Google Docs, Sheets, Slides et Drawings, est une suite bureautique permettant de modifier des documents, des feuilles de calcul, des présentations, des dessins, des formulaires, etc. Les utilisateurs peuvent rechercher les fichiers partagés publiquement sur Google Drive par l'entremise de moteurs de recherche Web [15].

3.1.8 Google Colaboratory

Google Colab ou Colaboratory est un service cloud, offert par Google (gratuit), basé sur Jupyter Notebook et destiné à la formation et à la recherche dans l'apprentissage automatique. Cette plateforme permet d'entraîner des modèles de Machine Learning directement dans le cloud ce qui nous a permis de faire la partie *Clustering*[4].

3.1.9 Postman

Postman est un logiciel gratuit qui permet d'effectuer des requêtes API sans coder, pour répondre à une problématique de test d'API partageable [13].

3.1.10 Express

Express est une infrastructure d'applications Web Nodejs minimaliste et flexible, qui fournit un ensemble de fonctionnalités robuste pour les applications Web et mobiles. Express apporte une couche fine de fonctionnalités d'application Web fondamentales, sans masquer les fonctionnalités de Nodejs [6].

3.1.11 Overleaf

Overleaf est un éditeur LaTeX en ligne, collaboratif en temps réel. C'est avec cette technologie que nous avons pu écrire ce rapport.

3.2 Schéma d'architecture entre les différentes parties de l'application

3.2.1 Diagramme de la façon de communication en utilisant *axios*

Ce schéma donne une illustration de la façon dont les différentes technologies utilisées communiquent entre elles en utilisant *axios*, comme montre la figure 3.1, cette partie sera détaillé dans section 6.1.

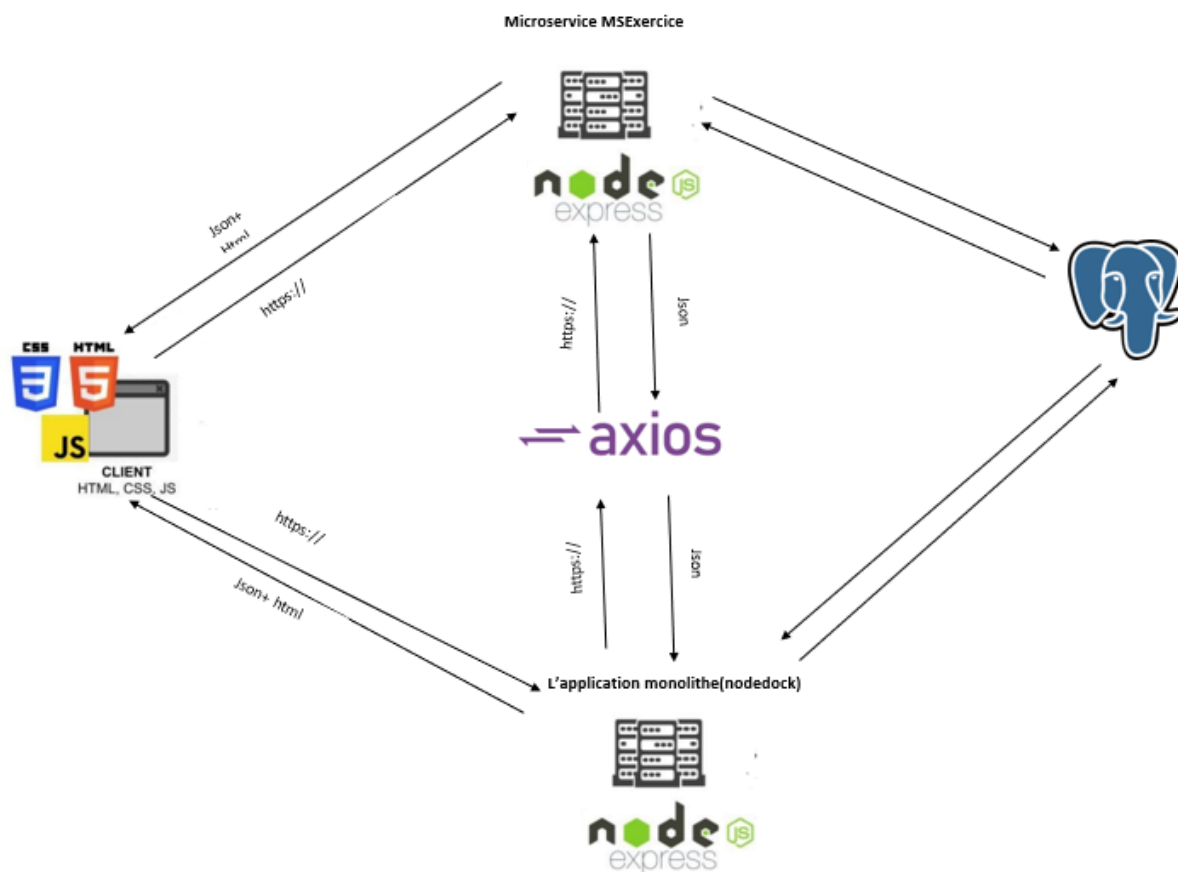


Figure 3.1. Schéma de l'architecture avec axios

3.2.2 Diagramme de la façon de communication en utilisant *API-Gateway*

Ce schéma donne une illustration de la façon dont les différentes technologies utilisées communiquent entre eux en utilisant *Api Gateway*, comme montre la figure 3.2, cette partie sera détaillé dans section 6.2.2 .

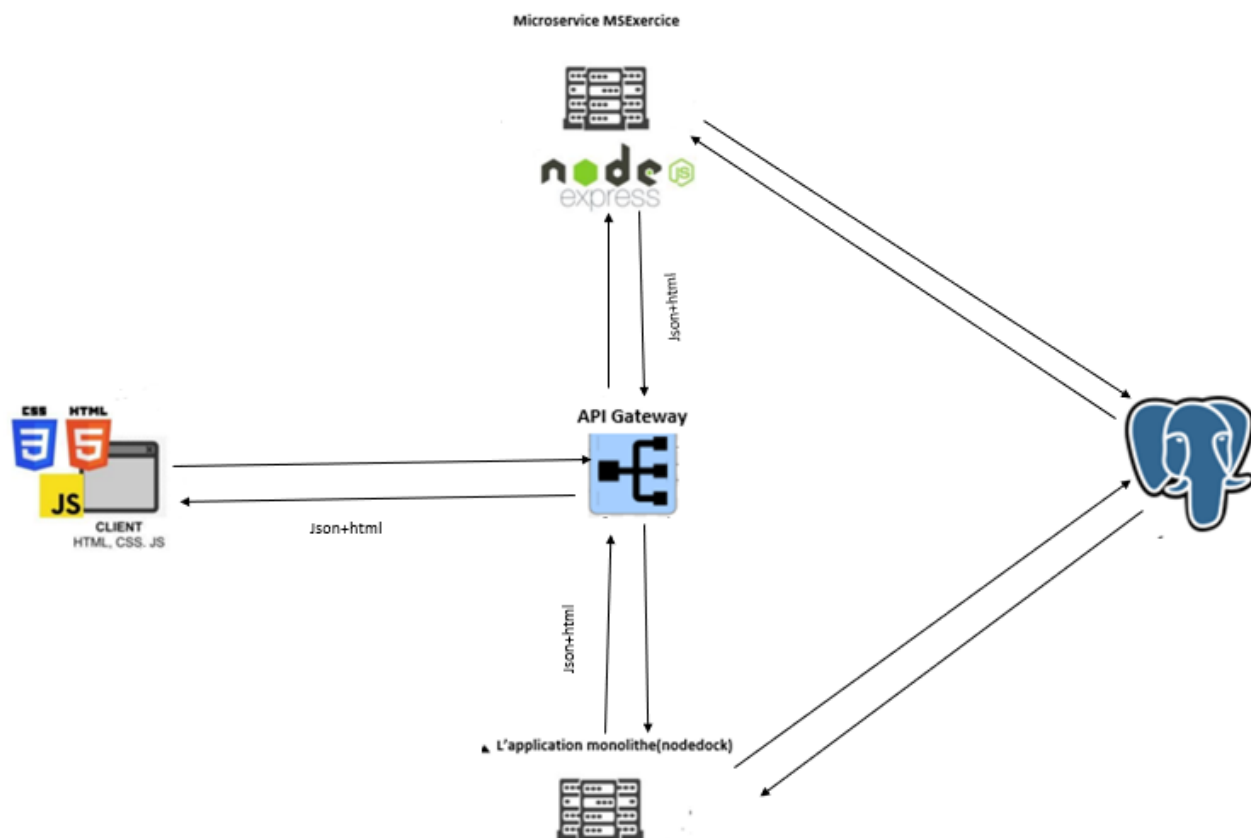


Figure 3.2. Schéma de l'architecture avec API Gateway

Chapitre 4

Prise en main et installation des technologies

À cause des contraintes qui sont liées à la performance de nos machines, on ne pouvait pas installer Ubuntu en double boot sur nos machines ou de créer des machines virtuelles qui sont lourdes et fastidieuses à utiliser, et on ne pouvait même pas utiliser nos sessions distantes sur les ordinateurs de la faculté à travers X2GO à cause des contraintes de droit d'installation.

Donc nos encadrants, nous ont proposé d'utiliser **WSL** pour travailler avec node, docker, visual studio et gitlab.

Dans les sections qui suivent nous allons expliquer comment on a procédé pour utiliser ces techniques dans notre projet.

4.1 Installation de WSL

On a installé WSL qui nous a permet de travailler avec la distribution Ubuntu de Linux dans Windows 10. Pour ce faire alors il était nécessaire de suivre plusieurs étapes qui se résument comme suit :

- Activer la fonctionnalité « Sous-système Windows pour Linux.
- Définir WSL 2 comme version par défaut.
- Installer la distribution Ubuntu de Linux, depuis le Microsoft Store.
- Exécutez ubuntu via le menu Démarrer.
- Saisissez le nom d'utilisateur puis le mot de passe du compte utilisateur.

Une fois la distribution Ubuntu de Linux est installée sur Windows 10, là on pouvait commencer à l'utiliser avec docker.

4.2 Installation et configuration de Docker pour WSL 2

Depuis Windows, nous pouvons exécuter des containers Linux grâce à une distribution Linux qui tourne avec WSL 2.

Pour ce faire il faut d'abord installer et configurer Docker pour WSL 2 en passant par les étapes suivantes :

- Télécharger de "Docker Desktop" (environ 500 Mo) sur le site officiel de Docker.
- Démarrer de l'installation, et cocher "Install required Windows components for WSL 2" lors de l'installation.
- Après la fin de l'installation, il faut redémarrer la machine.
- Finalement on ouvre Docker Desktop, on accède à "WSL Integration", et on coche la case pour que notre distribution Linux bénéficie de Docker.

Là on était prêts à créer et lancer notre premier container, qui est fait avec docker dans visual studio.

4.3 Installation de ShellOnYou

Après avoir installé WSL 2 et docker, on peut ouvrir visual studio avec la commande **code . &** puis on ajoute un terminal de type Ubuntu WSL, qui nous permet de cloner le projet depuis le dépôt GitLab de l'application avec une clé ssh avec la commande **git clone**.

Une fois que le projet est cloné on change de branche vers celle qui a été créé pour les microservices "ms".

Là on est prêt pour faire des *commit* dans le dépôt gitlab du projet.

Avant de lancer l'application ShellOnYou, on doit changer dans les variables d'environnements et on exécute la commande **npm install** pour installer les modules nécessaires au sein des conteneurs ce qui prends un

peu plus de temps dans la première fois (au début on avait que le conteneur nodedock qui est le monolithe et postgres qui représente la base de données). Puis pour démarrer l'application, on doit se mettre dans le dossier principal plage et on exécute la commande : **docker-compose up**.

Ensuite pour tester, on peut utiliser soit l'outil Postman soit un navigateur. Le serveur node.js du monolithe écoute sur le port HTTP 5001. La figure 4.1 un test qui présente la page d'accueil de l'application.

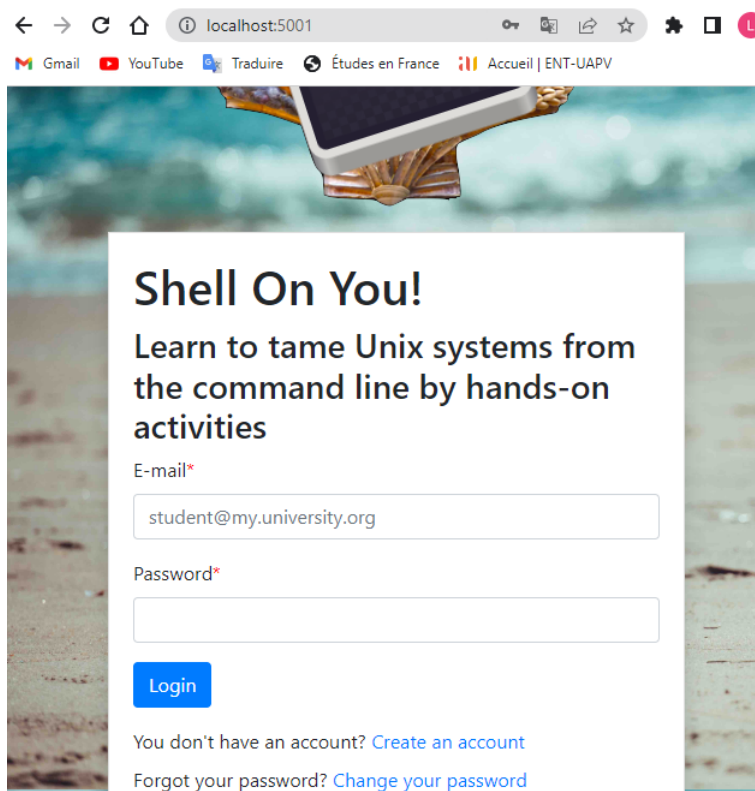


Figure 4.1. Page d'accueil

Chapitre 5

Migration vers les microservices

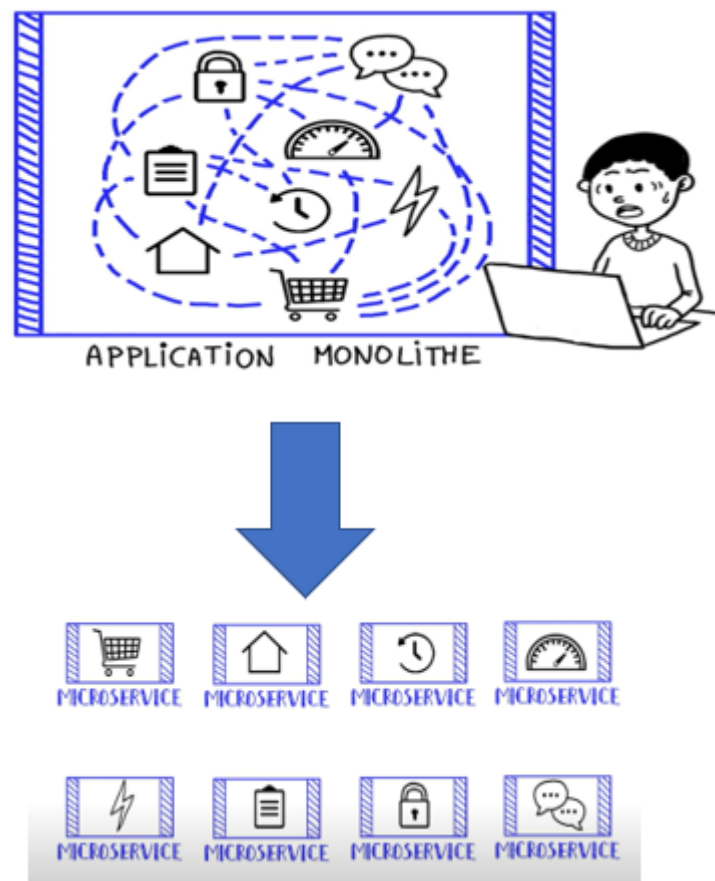


Figure 5.1. Exemple illustratif de l'architecture monolithique VS l'architecture microservices[5].

5.1 Monolithe

Une application monolithique est généralement un système d'applications dans lequel tous les modules pertinents sont regroupés sous la forme d'une seule unité déployable d'exécution [12].

Une application monolithique classique utilise une conception en couches, avec des couches distinctes pour l'interface utilisateur, la logique d'application et l'accès aux données, comme montre la figure 5.2.

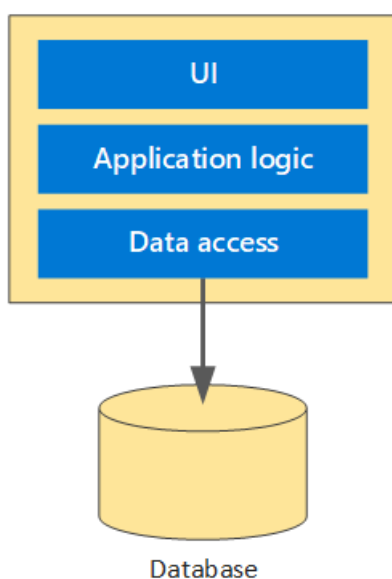


Figure 5.2. Architecture du monolithe [12].

Une application monolithique est généralement obtenue quand on commence une application par l'implémentation de petites fonctionnalités, auxquelles sont ajoutées de nouvelles fonctionnalités et ainsi de suite. Avec le temps l'application a tendance à grossir de plus en plus. Une application monolithique peut alors commencer à subir les problèmes suivants :

- La grande taille du code qui augmente avec le temps, et ce code devient complexe et difficile à comprendre et à faire évoluer.
- Les procédures de test deviennent plus difficiles, ce qui augmente la probabilité d'introduire des vulnérabilités.
- Les différentes parties du système ne peuvent pas être mises à l'échelle (*scaling*) indépendamment, car elles sont étroitement couplées.
- Toute modification, même triviale, nécessite le déploiement d'une nouvelle version de l'application entière.

5.2 Architecture micro-service

Une architecture de micro-service se compose d'un ensemble de petits services autonomes, chaque service doit implémenter une fonctionnalité unique dans un contexte borné.

Les micro-services sont généralement des unités d'exécution décentralisées et faiblement couplées.

Le figure 5.3 illustre une architecture de micro-services classique, où chaque micro-service (par exemple *produits*, *commandes*, etc) accède indépendamment à sa base de données. Avoir des bases de données séparées est considéré comme une bonne pratique, bien que dans certaines applications une base de données commune soit employée.

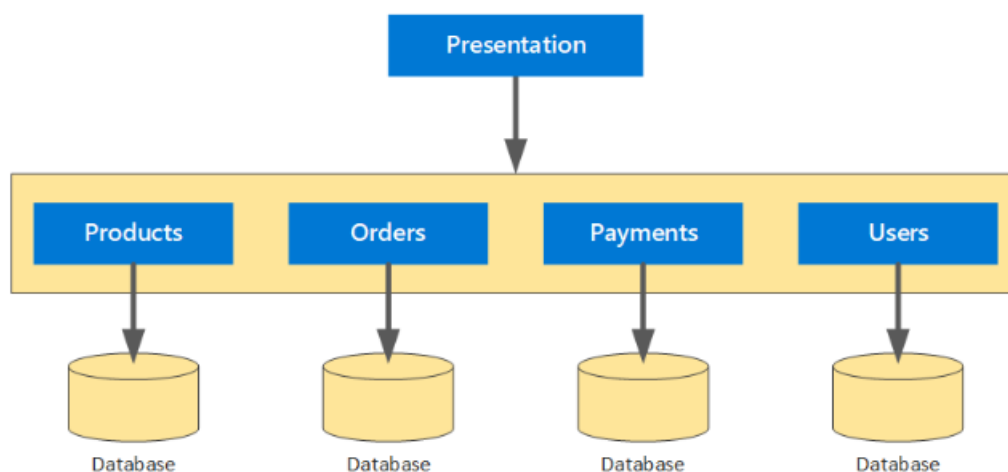


Figure 5.3. Exemple d'architecture micro-service [12].

La migration d'un monolithe vers un micro-service nécessite un temps et un investissement, donc il est important de comprendre les avantages et les défis que posent les micro-services :

- Les services peuvent évoluer indépendamment en fonction des besoins des utilisateurs.
- Au fil du temps, les cycles de développement sont plus rapides, car les fonctionnalités peuvent être mises en production plus rapidement.
- Les services sont isolés et sont plus tolérants aux défaillances : un service unique qui échoue n'entraîne pas la défaillance de l'ensemble de l'application.
- Les services peuvent être déployés de manière indépendante. Une équipe peut mettre à jour un service existant sans avoir à recréer et redéployer l'application entière.

Chapitre 6

Partie technique : migration du monolithe vers les microservices

6.1 Création du micro-service *MSExercice*

Suivant les études qu'on a fait concernant la migration du monolithe vers des micro-services, on a trouvé que la partie qui dépend des exercices et des scripts python est la partie qui connaît plus de pression par les utilisateur. C'est pour cela on a procédé à ressortir cette partie du monolithe et de lui créer un micro-service y propre, qui est totalement indépendant des autres parties de l'application, contient que les fichiers qui concernent les exercices.

6.1.1 Objectif de *MSExercice*

Ce service permet de stocker dans la base de données des scripts Python et des archives *tar.gz*. Pour simplifier l'utilisation des fichiers, ils sont convertis dans un modèle spécifique avant leur stockage. Ce modèle permet de stocker le nom, la taille, les données et le MD5 du fichier dans un seul attribut. Ce service permet la réalisation des différentes tâches suivante :

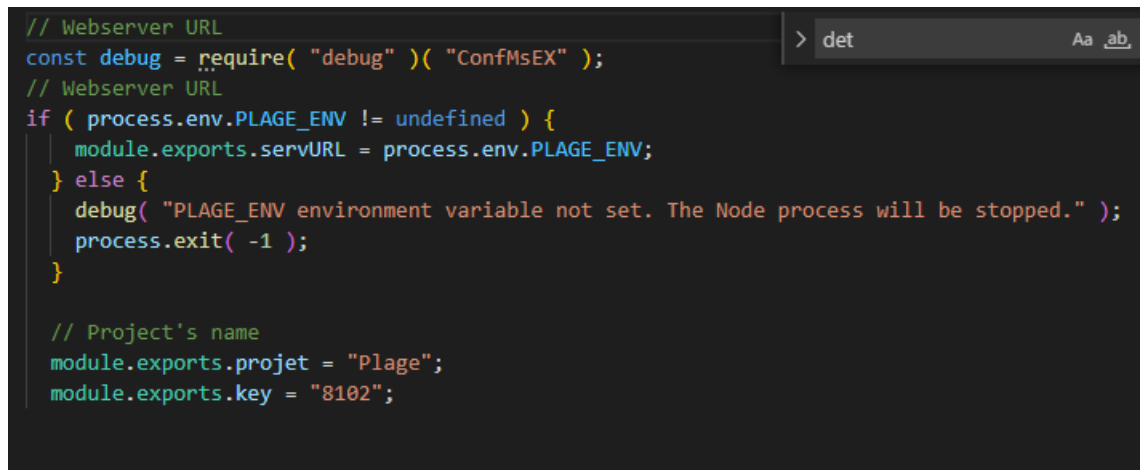
- La création des exercices selon des conditions spécifiques qui dépendent du rôle de l'utilisateur (seuls certains utilisateurs ont le droit d'ajouter des exercices).
- La visualisation de la liste des exercices.
- L'affichage des détails de chaque exercice, avec la possibilité de téléchargement des fichiers *tar.gz*.
- La possibilité de modifier les informations, qui concernent les exercices, soumis à des conditions.
- La possibilité de la suppression des exercices, selon le rôle de l'utilisateur.

- D'autres méthodes qui s'exécutent en arrière plan, lors de l'exécution des méthodes principales citées précédemment.

6.1.2 Création et configuration de conteneur

Ce microservice a été conçu selon le principe du MVC, c'est à dire qu'on a gardé la même architecture que l'application principale en lui extrayant tous les contrôleurs, modèles, vues et aussi les routes concernant la parties exercices et scripts python.

Ensuite on ajouté à notre microservice, des fichiers de configuration pour pouvoir accéder à des ressources externe (si besoins), comme par exemple le fichier `ConfMsEx.js` qui permet d'accéder au serveur du monolithe qui écoute sur le port 5001, comme montre la figure 6.1 (Ces fichiers de configuration sont dans le répertoire **config**).



```
// Webserver URL
const debug = require( "debug" )( "ConfMsEX" );
// Webserver URL
if ( process.env.PLAGE_ENV !== undefined ) {
  module.exports.servURL = process.env.PLAGE_ENV;
} else {
  debug( "PLAGE_ENV environment variable not set. The Node process will be stopped." );
  process.exit( -1 );
}

// Project's name
module.exports.projet = "Plage";
module.exports.key = "8102";
```

Figure 6.1. Le fichier de configuration ConfMsEx.js

Puis pour configurer notre microservice et créer le conteneur docker qui est associé on passe par les étapes suivantes :

- créer un fichier `package.json` avec les dépendances de notre service en exécutant la commande `npm init`.

```
{
  "name": "plage",
  "version": "0.0.0",
  "description": "A simple application for Shell Week @ IG Polytech Mtp",
  "main": "index.js",
  "repository": "",
  "author": "...",
  "scripts": {
    "start": "node index.js",
    "start-watch": "nodemon index.js",
    "lint": "eslint index.js server.js controller --ext .js ",
    "lint-fix": "eslint index.js server.js controller --ext .js --fix",
    "test": "NODE_ENV=test APP_ENV=test jest",
    "test-watch": "NODE_ENV=testjest --watch",
    "ejs-lint": "./node_modules/.bin/ejslint view"
  },
  "dependencies": {
    "archiver": "^5.3.0",
    "async": "^3.2.3",
  }
}
```

Figure 6.2. Extrait du code du fichier `package.json`

- Installer des dépendances en exécutant la commande `npm install` qui permet d'installer tous les modules nécessaires.
- Définir l'environnement de l'application avec un `Dockerfile` qui permet de construire une image Docker adaptée à nos besoins.

```
FROM node:10

# Installs pip and psycopg2-binary necessary for some python exercises
RUN apt-get update && apt-get install -y python-pip
RUN pip install psycopg2-binary

# Create app directory and default directory to put things
# (This way we do not have to type out full file paths but
# can use relative paths based on the working directory. )
WORKDIR /usr/src/app

COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install

# If you are building your code for production
```

Figure 6.3. Extrait de code du fichier de configuration `Dockerfile`

- Définir le service *MSExercice* dans le fichier `docker-compose.yml` avec les champs nécessaires afin qu'il puisse être exécuté en même temps avec les autres conteneurs, mais dans un environnement isolé.

```
MSExercice:
  volumes:
    - ./MSExercice:/usr/src/app
  build: ./MSExercice
  image: "exercice/exercice:21"
  container_name: MSExercice
  depends_on:
    - postgres
  ports:
    - 5021:5021
  networks:
    - plagenet
  env_file:
    - variables.env
```

Figure 6.4. Définition du service MSExercice dans le fichier yml

- Ajouter un serveur Node.js appelé `index.js` qui écoute les requêtes reçues sur le port 5021.

```
i18n.expressBind(app, {  
  // setup some locales - other locales default to vi silently  
  locales: ["en", "fr"],  
  // set the default locale  
  defaultLocale: "en",  
  session: true,  
  directory: "./locale",  
  extension: ".json",  
});  
  
//Set proper Headers on Backend  
app.use((req, res, next) => {  
  res.setHeader("Access-Control-Allow-Origin", "*");  
  res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");  
  res.setHeader("Access-Control-Allow-Headers", "Content-Type, Authorization");  
  next();  
});  
app.use("/css", express.static(path.join(__dirname + '/static/css')));  
app.use("/img", express.static(path.join(__dirname + '/static/img')));
```

Figure 6.5. Extrait de code du serveur de MSExercice

- Exécuter la commande "docker build nom_de_notre_image ." pour construire l'image de notre conteneur "MSExercice".
 - Exécutez la commande "docker-compose up" qui démarre et exécute l'intégralité de l'application.
- Dans la figure 6.6 on représente un exemple de test avec la méthode GET qui renvoie le formulaire de création d'un exercice.

localhost:5021/exercice

uTube Traduire Études en France Accueil | ENT-UAPV [L3 Projet 2019-21]... Mon Drive - Googl... Izly Accueil | EN

[e](#) [S'inscrire à une session](#) [Profils](#) [Exercices](#) [Créer un exercice](#) [Séquences](#) [Créer une séquence](#) [Se](#)

Créer un nouvel exercice

Exercise name :

jouons avec ls

Énoncé de l'exercice :

Fichier Editer Voir Format Outils

← → Paragraphe ▾ **B** *I* [List icons]

Figure 6.6. Test de la méthode GET qui renvoie le formulaire de création d'un exercice.

6.1.3 Utilisation d'axios

Après avoir créé et testé toutes les méthodes http de notre micro-service, on avait comme but de rediriger les requêtes http arrivant aux monolithe et qui concernent la partie exercice vers le micro-service qu'on a créé précédemment.

Comme c'est marqué dans la sous-section 3.1.6, **axios** est une bibliothèque qui permet de communiquer entre des API, ça nous a fait penser à l'utiliser pour faire la redirection des requêtes reçus par le monolithe vers MSExercice.

Pour ce faire, on a décidé de faire la redirection dans les routes de node-dock (monolithe) dans les fichiers concernant les routes de exercices. Lorsque on envoie une requête à MSExercice à l'aide d'axios, ce dernier renvoie une réponse. L'objet de réponse se compose de :

- **data** : les données renvoyées par le serveur.
- **status** : le code HTTP renvoyé par le serveur.
- **statusText** : le statut HTTP renvoyé par le serveur.
- **headers** : en- têtes obtenus du serveur.
- **config** : la configuration de la requête d'origine.
- **request** : l'objet de la requête.

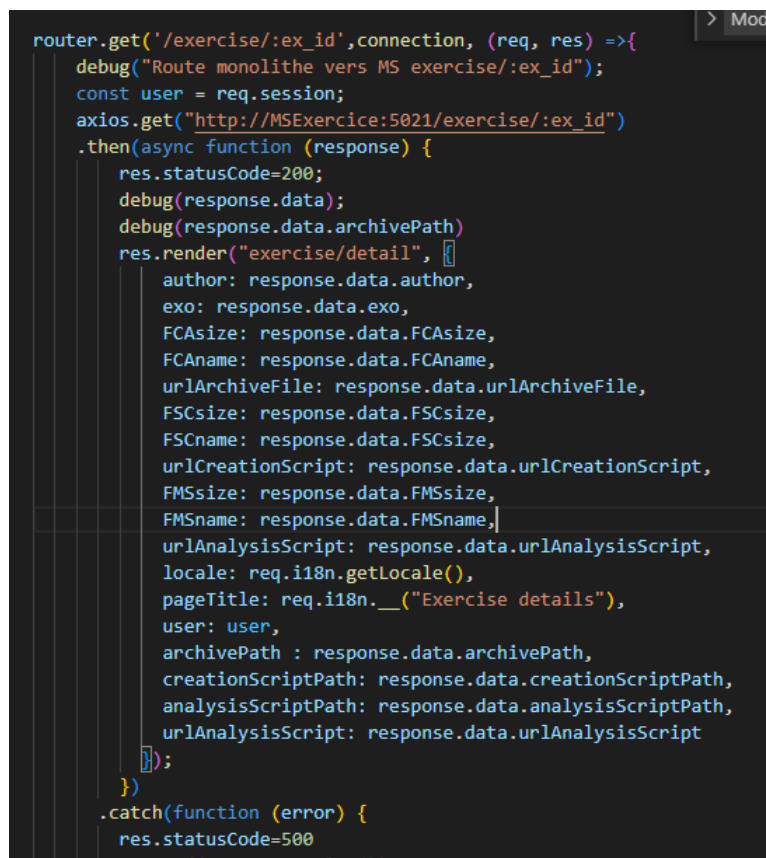
Cette réponse est traitée et sérialisée dans une méthode des contrôleurs de notre microservice.

Puis au niveau de nodedock, les données renvoyées par le microservice sont stockées dans **response.data**, qu'on peut utiliser après si besoin pour renvoyer une réponse aux clients.

La redirection vers les vues maintenant se fait au niveau des routes de nodedock au lieu de ses contrôleurs.

Exemple de redirection de la requête de demandant les détails d'un exercice :

La capture 6.7 représente un extrait de code qu'on a utilisé pour faire la redirection avec axios vers le microservice, ainsi que l'envoi de la vue en question au client en utilisant **render** au niveau de la route de détail d'un exercice dans le monolithe.



```
router.get('/exercise/:ex_id',connection, (req, res) =>{
  debug("Route monolithe vers MS exercise/:ex_id");
  const user = req.session;
  axios.get("http://MSExercise:5021/exercise/:ex_id")
    .then(async function (response) {
      res.statusCode=200;
      debug(response.data);
      debug(response.data.archivePath)
      res.render("exercise/detail", {
        author: response.data.author,
        exo: response.data.exo,
        FCAsize: response.data.FCAsize,
        FCAname: response.data.FCAname,
        urlArchiveFile: response.data.urlArchiveFile,
        FSCsize: response.data.FSCsize,
        FSCname: response.data.FSCsize,
        urlCreationScript: response.data.urlCreationScript,
        FMSsize: response.data.FMSsize,
        FMSname: response.data.FMSname,
        urlAnalysisScript: response.data.urlAnalysisScript,
        locale: req.i18n.getLocale(),
        pageTitle: req.i18n.__( "Exercise details"),
        user: user,
        archivePath : response.data.archivePath,
        creationScriptPath: response.data.creationScriptPath,
        analysisScriptPath: response.data.analysisScriptPath,
        urlAnalysisScript: response.data.urlAnalysisScript
      });
    })
    .catch(function (error) {
      res.statusCode=500
    })
})
```

Figure 6.7. Exemple d'utilisation d'axios pour rediriger vers MSExercise dans nodedock

La figure 6.8 représente la route de détail d'un exercice au niveau de MSError, qui invoque la méthode `read()`, qui renvoie un objet contenant les informations d'un exercice par son id.

```
// Details
router.get('/exercise/:ex_id', function (req, res) {
  ControllerExercise.read(req, res)
})
```

Figure 6.8. Route de détail d'un exercice dans MSError

Les deux figures 6.9, 6.10 montre un extrait de code de la méthode `read()` que l'on appelle dans la route de détail d'un exercice, cette méthode renvoie un objet réponse sérialisé à l'aide de la méthode **stringify** de l'objet **JSON**.

```
module.exports.read = async function (req, res) {
  debug("Want to get an exercise with its files");
  //id de l'user qui esr connecter pour recuperer Les exo qui a écrit
  const user = req.session;
  const exo = await ModelExercise.readById('1');
  if (exo) {
    const resExo = new ModelExercise(exo);
    let fCA, fSCS, fMS;
    let archivePath, urlArchiveFile;
    let creationScriptPath, urlCreationScript;
    let analysisScriptPath, urlAnalysisScript;
    const empty = { name: "Aucun fichier", size: 0, md5: 0, data: 0 };
    if (resExo.template_archive !== undefined) {
      fCA = new ModelFile(JSON.parse(resExo.template_archive));
      // new
      let fileName = fCA.name;
      let randomFolderName = crypto.randomBytes(20).toString('hex');
      archivePath = "static/files/"+randomFolderName+"/"
      fs.mkdir(archivePath, { recursive: true }, function (err) {
        if (!err) {
          archivePath += fileName
          fs.writeFile(archivePath, Buffer.from(fCA.data), function (err) {
            if (!err) {
              debug("archive file put in folder "+archivePath)
            }
          })
        }
      })
    }
  }
}
```

Figure 6.9. La méthode `read()` de controller exercise (extrait de code 1)

```
res.status(200).send(JSON.stringify({
  author: author,
  exo: resExo,
  FCAsize: fCA.size,
  FCAname: fCA.name,
  urlArchiveFile: urlArchiveFile,
  FSCsize: fSCS.size,
  FSCname: fSCS.name,
  urlCreationScript: urlCreationScript,
  FMSsize: fMS.size,
  FMSname: fMS.name,
  urlAnalysisScript: urlAnalysisScript,
  locale: req.i18n.getLocale(),
  pageTitle: req.i18n.__("Exercise details"),
  user: user,
  archivePath :archivePath,
  creationScriptPath: creationScriptPath,
  analysisScriptPath: analysisScriptPath,
  urlAnalysisScript: urlAnalysisScript

}));
} else {
  res.status(500).end();
}
```

Figure 6.10. La méthode `read()` permet de contrôler un exercice (extrait de code 2).

La figure 6.12 est un exemple de résultat de redirection depuis le monolithe, si on clique sur le bouton *Détails* de l'exercice nommé *jour 13* dans la liste des exercices.

[S'inscrire à une session](#) [Profils](#) [Exercices](#) [Créer un exercice](#) [Séquences](#) [Créer une séquence](#) [Sessions](#) [Créer un](#)

Liste des exercices

Nom	Auteur	État	Actions
jour 13	Admin	Rédaction en cours	Modifier Supprimer Détails
ms exo	Admin	Rédaction en cours	Modifier Supprimer Détails
Exercice de commande	Admin	À tester	Modifier Supprimer Détails
Exercice shell linux	Admin	À tester	Modifier Supprimer Détails
Test test	Admin	Disponible	Modifier Supprimer Détails

[Télécharger la dernière version de la bibliothèque Python](#)

[Langue](#)   © Copyright 2018-2022 Polytech 

Figure 6.11. Liste des exercices affichée depuis le monolithe.

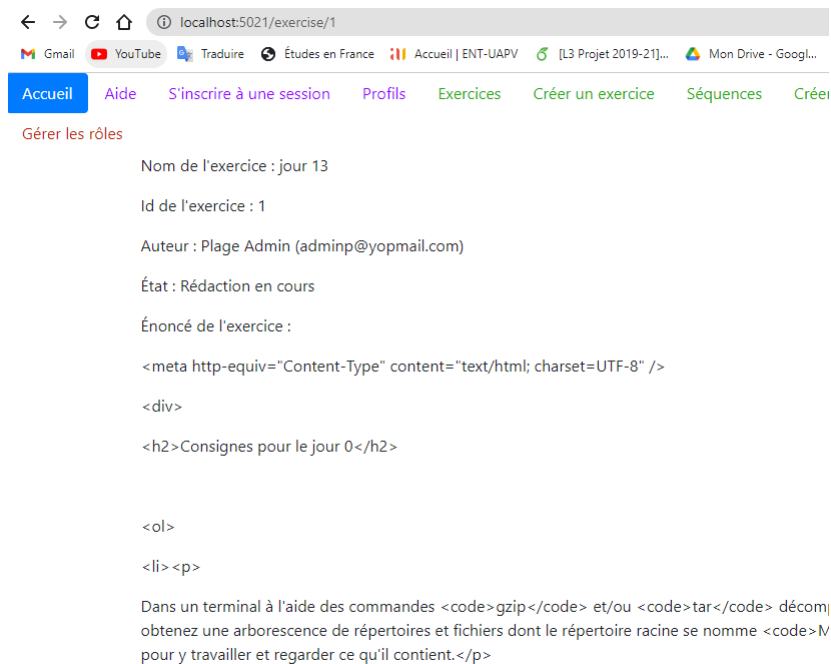


Figure 6.12. Detail de l'exercice nommé *jour 13*.

6.2 Création du microservice *API Gateway*

6.2.1 Objectif de *APIGateway*

Ce microservice a été créé pour jouer le rôle de la passerelle d'API qui est le point d'entrée à l'application. On appelle la passerelle d'API, qui transfère l'appel aux services appropriés sur le back-end.

Nous avons décidé de créer ce microservice pour remplacer la bibliothèque *axios*, qu'on a déjà utilisé pour gérer la communication entre le monolithe et le microservice, suite au problème généré lors de la redirection. En fait le monolithe parse le body (contenu) de la requête avant d'appeler le microservice, donc on ne peut pas accéder aux informations saisies par l'utilisateur dans un formulaire, ou récupérer les données saisies dans l'URL, etc.

Donc cet API gateway serait le premier à recevoir les requêtes des clients et sans toucher aux requêtes reçues, les redirige soit vers le monolithe soit vers le microservice *MSExercice*.

6.2.2 Création et configuration de conteneur

La création du microservice *APIGateway* fait essentiellement appel à un module npm appelé **express-http-proxy**, pour configurer notre microservice et créer le conteneur docker y associé on a passé par les étapes suivantes :

- créer un fichier **package.json** avec les dépendances de notre service en exécutant la commande **npm init**.
- Installer des dépendances en exécutant la commande **npm install** qui permet d'installer tout les modules nécessaires.
- Définir l'environnement de l'application avec un **Dockerfile** qui permet de construire une image Docker adaptée à nos besoins.
- Définir le service *Gateway* dans le fichier **docker-compose.yml** avec les champs nécessaires afin qu'il puisse être exécuté en même temps avec les autres conteneurs mais dans un environnement isolé.

```
Gateway:
  volumes:
    - ./Gateway:/usr/src/app
  build: ./Gateway
  image: "gateway/gateway:25"
  container_name: gateway
  depends_on:
    - postgres
  ports:
    - 5022:5022
  networks:
    - plagenet
  env_file:
    - variables.env
```

Figure 6.13. Définition du service Gateway dans le fichier yml

- Ajouter un serveur proxy Node.js appelé **index.js** qui écoute les requêtes reçues sur le port 5022, et qui gère la redirection vers monolithe et le microservice en utilisant le module **express-http-proxy** qui permet à ce serveur de jouer le rôle d'intermédiaire entre ces services (nodedock et MSExercice).

```
const path = require('path');
const request = require('request');
const cors = require('cors');
const proxy = require('express-http-proxy');
app.use(cors());
app.use(express.json());

// les autres routes sont envoyées au monolithe :
var httpProxy = require('http-proxy');
var apiProxy = httpProxy.createProxyServer();
app.use("/Ms", function(req, res) {

  apiProxy.web(req, res, { target: 'http://MSExercice:5021'})
});
app.use("/", function(req, res) {

  apiProxy.web(req, res, { target: 'http://node:5001'})
});
app.listen(5022, ()=>{
  console.log('ShellOnYou **API gateway** listening on port 5022');
});
```

Figure 6.14. Des lignes de fichier index de Gateway

- Exécuter la commande "docker build nom_de_notre_image ." pour construire l'image de notre conteneur *APIGateway*.
- Exécution de la commande **docker-compose up**, pour démarré l'intégralité de l'application.

Dans le cas de l'ajout de d'un nouveau micro-services, *APIGateway* peut gérer la redirection de ce dernier facilement.

6.2.3 Exemple de la nouvelle version de l'application avec API Gateway

Comme c'est indiqué précédemment, le serveur de l'API Gateway écoute les requêtes reçues sur le port **5022**, et redirige soit vers le monolithe soit vers le microservice *MSExercice*, alors dans les figures ci-dessous on montre un exemple de ceci.

- La figure 6.15 représente la page d'accueil de l'application que l'on obtient après une redirection de la part de notre API Gateway.

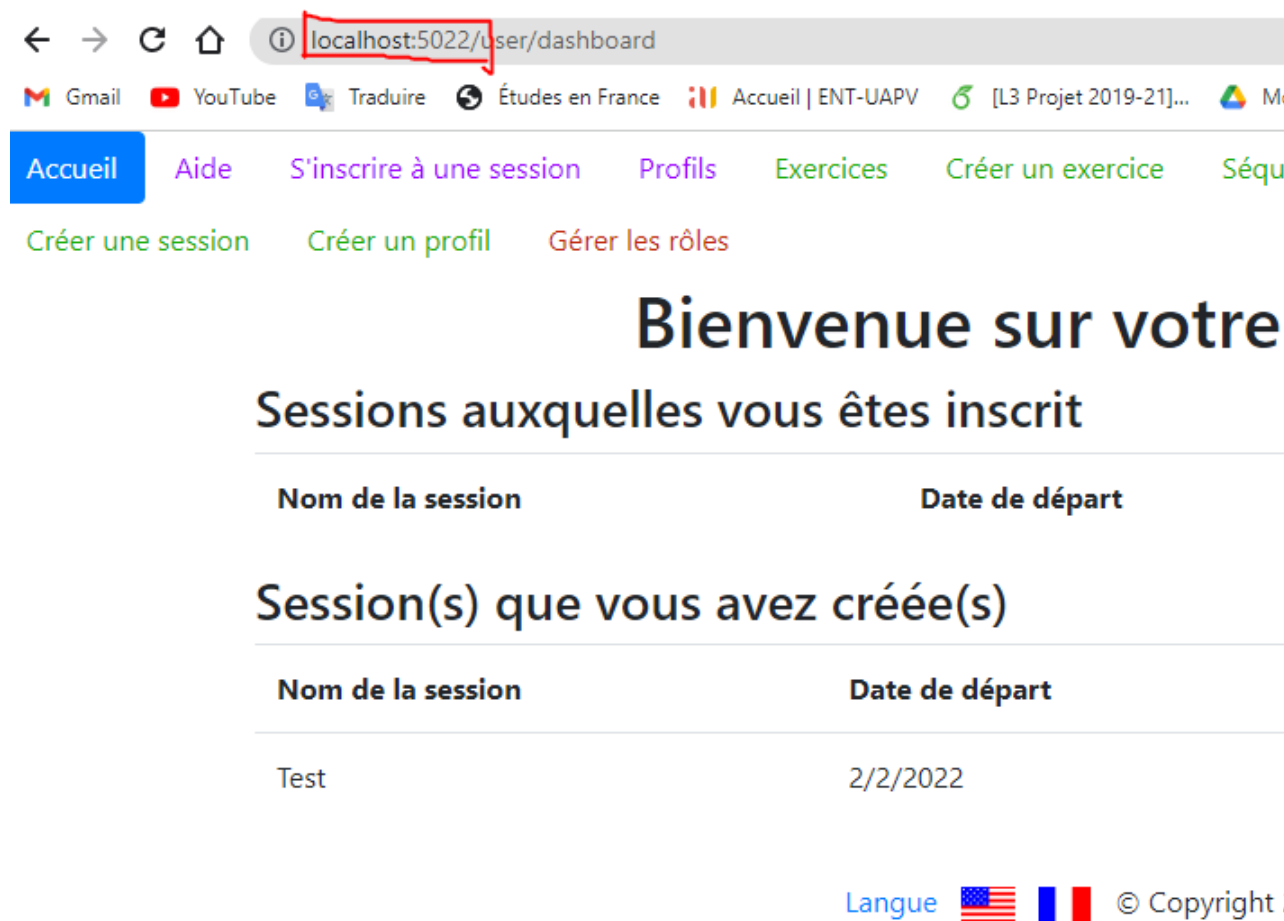


Figure 6.15. Page d'accueil de l'application avec API Gateway

– La figure 6.16 montre un exemple de redirection de la requête demandant la liste des exercices vers MSeXercice.

localhost:5022/Ms/exercise/list

Traduire Études en France Accueil | ENT-UAPV [L3 Projet 2019-21]... Mon Drive - Googl...

S'inscrire à une session Profils **Exercices** Créer un exercice Séquences Créer un

Liste des exercices

Nom	Auteur	État
jour 13	Admin	Rédaction en cours
ms exo	Admin	Rédaction en cours
Exercice de commande	Admin	À tester
Exercice shell linux	Admin	À tester
Test test	Admin	Disponible

[Télécharger la dernière version de la bibliothèque Python](#)



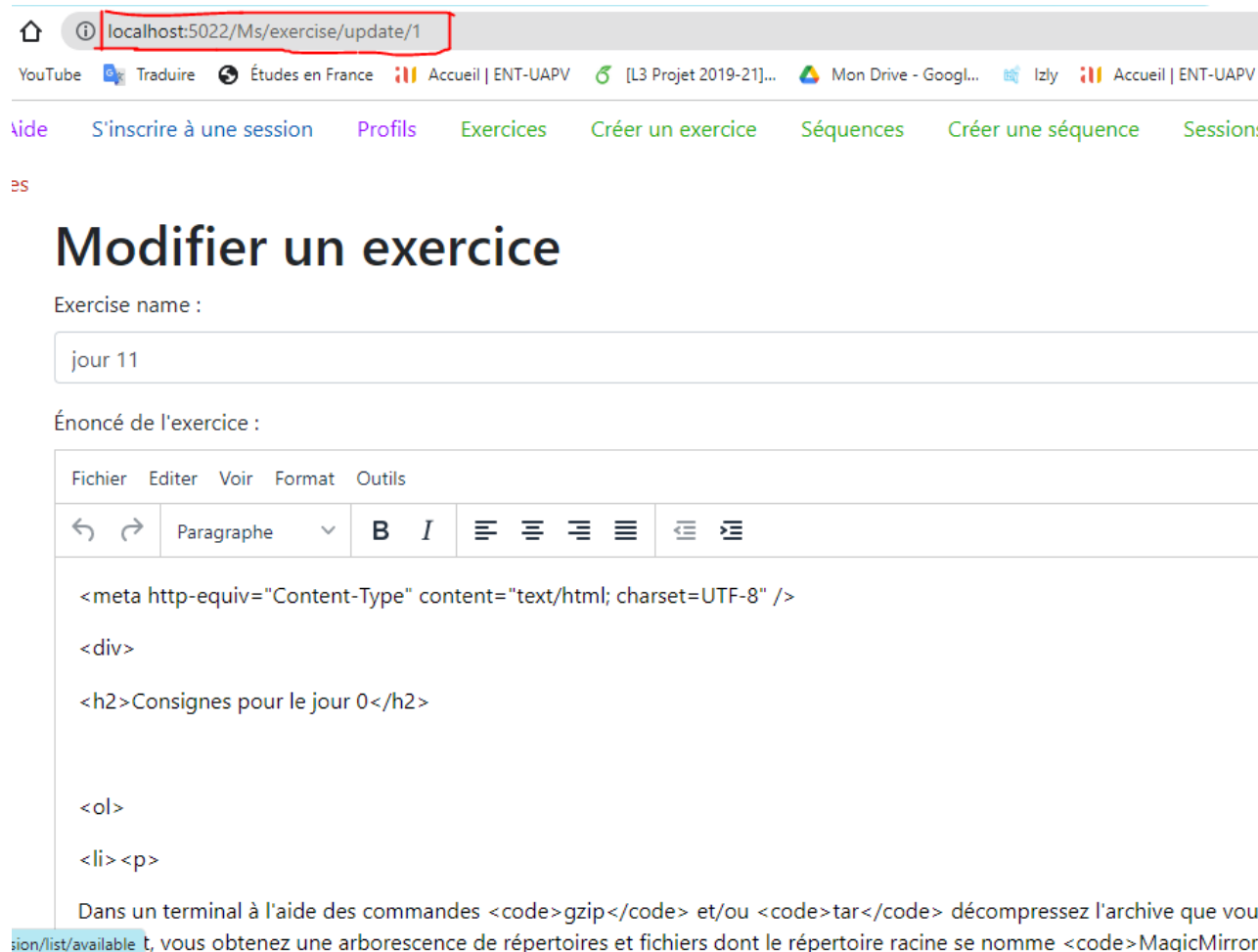
Lingue   © Copyright 2018-2022 Polytech

Figure 6.16. Liste des exercices dans le microservice MSeXercice

Si on clique sur le bouton **Modifier** pour modifier les informations qui concerne l'exercice dont le nom est "jour 13", l'API Gateway nous redirige vers la vue de modification d'un exercice dans *MSExercice*, comme c'est représenté dans la figure 6.17.



localhost:5022/Ms/exercise/update/1

YouTube Traduire Études en France Accueil | ENT-UAPV [L3 Projet 2019-21]... Mon Drive - Google Izly Accueil | ENT-UAPV

[Aide](#) [S'inscrire à une session](#) [Profils](#) [Exercices](#) [Créer un exercice](#) [Séquences](#) [Créer une séquence](#) [Session](#)

Modifier un exercice

Exercise name :

jour 11

Énoncé de l'exercice :

Fichier Editer Voir Format Outils

↶ ↷ Paragraphe ▾ **B** *I* ☰ ☷ ☷ ☷ ☷ ☷ ☷

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<div>
<h2>Consignes pour le jour 0</h2>

<ol>
<li><p>
Dans un terminal à l'aide des commandes <code>gzip</code> et/ou <code>tar</code> décompressez l'archive que vous trouvez à l'adresse <a href="#">session/list/available</a>, vous obtenez une arborescence de répertoires et fichiers dont le répertoire racine se nomme <code>MagicMirror
```

Figure 6.17. Formulaire de modification d'un exercice

Après avoir changer le nom de l'exercice en question, si on regarde dans la liste des exercice on remarque que le nom de l'exercice est bien changé, comme c'est marqué dans la figure 6.18.

Liste des exercices

Nom	Auteur	État	Actions
jour 11	Admin	Rédaction en cours	Modifier Supprimer Détails

Figure 6.18. Liste des exercices dans MSExercice

En cliquant sur le bouton **Détails** de cette exercice, une requête demandant les information de cet exercice sera envoyé à API Gateway qui lui même va la rediriger vers MSExercice qui y traite et renvoie une réponse, la figure 6.19 est une capture qui montre les détails d'un exercice dans MSExercice.

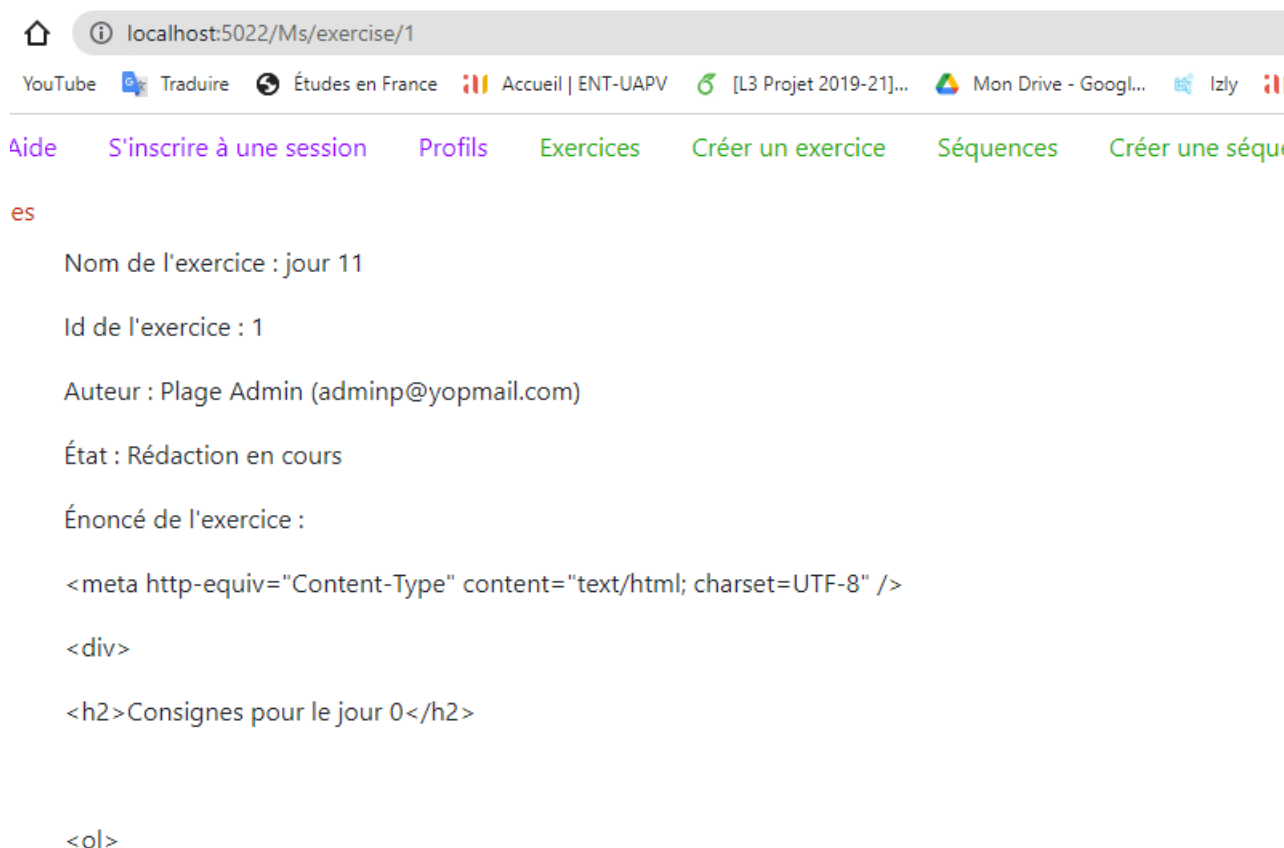


Figure 6.19. Détails de l'exercice nommé "jour 11"

Les requêtes http demandant la création d'un exercice sont aussi redirigées vers le microservice. La figure 6.20 est un exemple de création d'un nouveau exercice par le microservice après avoir reçu une requête redirigée par API Gateway.

localhost:5022/Ms/exercice

Tube Traduire Études en France Accueil | ENT-UAPV [L3 Projet 2019-21]... Mon Drive - Googl... Izly

S'inscrire à une session Profils Exercices **Créer un exercice** Séquences Créer une séquence

Créer un nouvel exercice

Exercise name :

Exercice 15

Énoncé de l'exercice :

Fichier Editer Voir Format Outils

↶ ↷ Paragraphe B I ☰ ☷ ☹ ☺ ☰ ☷

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<div>
<h2>Consignes pour le jour 0</h2>

<ol>
<li><p>
Dans un terminal à l'aide des commandes <code>gzip</code> et/ou <code>tar</code> décompressez l'arc
résultat, vous obtenez une arborescence de répertoires et fichiers dont le répertoire racine se nomme <code>
dans ce répertoire pour y travailler et regarder ce qu'il contient </code>
```

Figure 6.20. Formulaire de creation d'un exercice dans MSExercise

Si on demande l'affichage de la liste des exercices dans le microservice, on voit bien que l'exercice qu'on a créé est bien ajouté dans la liste, comme l'indique la figure 6.21

Exercice 15	Admin	Rédaction en cours	Modifier Supprimer Détails
-------------	-------	--------------------	--

Figure 6.21. Création d'un nouveau exercice nommé "Exercice 15"

Chapitre 7

Découpage en clusters

L'architecture basée sur les microservices est un style architectural exploité pour développer des systèmes logiciels avec la principale préoccupation de maintenabilité, d'indépendance, de déployabilité et d'évolutivité. Ces fonctionnalités importantes dans le développement de logiciels modernes ont conduit de nombreuses entreprises à migrer leurs systèmes logiciels monolithiques existants vers des architectures basées sur les microservices.

Le processus de migration est une tâche plus ou moins fastidieuse car cela nécessite de diviser le système en parties cohérentes qui représentent l'ensemble des microservices. Les travaux existants portent principalement sur les aspects fonctionnels de ce découpage. Nous discutons dans ce travail qu'il serait avantageux de commencer ce fractionnement en décomposant la base de données en clusters, où les données de chaque cluster sont associées à leur propre base de données indépendante.

Communément appelé le modèle "database-per-service" dans l'architecture de micro-services, ce splitting comporte trois étapes.

7.1 Construction et pré-traitement des données

7.1.1 Construction

Basé sur le modèle de schéma de base de données du système logiciel monolithique, nous créons un ensemble de documents. Chaque document représente une collection de symboles dérivés d'une table relationnelle dans le modèle de données. Un symbole peut être le nom d'une table, l'attribut d'une table ou une relation entre deux tables. Chaque symbole est modélisé selon son importance, où :

- Le nom de la table est répété plusieurs fois dans le document. Cette répétition lui donne un poids plus lourd, puisqu'il s'agit d'un symbole

important caractérisant une structure de données donnée.

- Une relation entre les tables représente l'existence d'une clé étrangère qui est un symbole significatif qui décrit une relation fonctionnelle ou structurelle entre les tableaux.

Remarque : On distingue deux types de clés étrangères :

- La première, appelée clé étrangère singleton, relie une table source vers une seule table cible. Ce qui décrit une forte relation entre les deux. Nous le modélisons donc en dupliquant la clé étrangère et le nom du document source plusieurs fois (quatre fois dans notre cas) dans le document cible. De même, nous dupliquer le nom du document cible dans le document source.
- Le deuxième type est une clé étrangère non singleton, elle relie la table source à plusieurs tables cibles. Ce type indique une relation moins forte entre les documents par rapport au premier type. Par conséquent, nous le modélisons en répétant la clé étrangère et le nom du document source moins de fois que le premier type dans les documents cibles. On fait de même pour le nom des documents cibles dans le document source.

7.1.2 Pré-traitement

Après l'étape précédente, les symboles du document sont traités et nettoyés en utilisant la technique du NLP (Natural Language Processing) comme suit :

- **Tokénisation :** cette procédure prend chaque symbole de document et le divise en mots séparés (jetons). La séparation est basé sur l'existence de certains caractères entre les jetons comme '-' et '_' ou des lettres majuscules.
- **Lemmatisation :** c'est le processus de transformation de chaque mot en son lemme (racine) à l'aide d'une analyse morphologique. Pour exemple, les mots "*studies*" ou "*studying*" sont transformés en "*study*", qui est leur racine.

La figure **7.1** détaille bien comment nous avons procédé au prétraitement du schema de la base de données.

7.2 Clustering

7.2.1 Mesure de similarité

Dans cette partie, nous classons tous les symboles en clusters, où chaque cluster contient les symboles les plus similaires. Cette mesure de similarité est faite en utilisant **Jaccard Index (ou Jaccard Similarity**

```
▼ Preprocessing

[21] # stop words
stop_words = set(stopwords.words('english'))
# punctuation
punctuation = set(string.punctuation)
# lemmatization
lemmatization = WordNetLemmatizer()

[22] # function clean
def clean(documents):
    # split documents and remove stop words

    split_doc = " ".join([i for i in documents.lower().split() if i not in stop_words])

    # remove punctuation
    punc_doc = ''.join([j for j in split_doc if j not in punctuation])

    # normalize the text
    normalized = " ".join([lemmatization.lemmatize(word) for word in punc_doc.split()])

    return normalized

# Using the Jaccard index to get the word similarity
def jaccard_similarity(x,y):
    """ returns the jaccard similarity between two lists """
    intersection_cardinality = len(set.intersection(*[set(x), set(y)]))
    union_cardinality = len(set.union(*[set(x), set(y)]))
    return intersection_cardinality/float(union_cardinality)
```

Figure 7.1. Méthodes de prétraitement

Coefficient) défini comme étant le rapport de la taille de l'intersection et de la taille de l'union de deux éléments que l'on souhaite connaître leur similarité.

7.2.2 Vectorisation des documents

C'est la procédure de transformation de chaque symbole en un modèle d'espace vectoriel. Cette procédure fournit un poids pour chaque symbole, ce qui indique la valeur de ce symbole dans son document et aussi en considérant également la collecte de tous les documents.

Nous transformons d'abord les documents prétraités en un ensemble de vecteurs. Nous avons utilisé la méthode TF-IDF pour vectoriser nos documents. Nous avons choisi cette méthode car cela nous a donné de bons résultats avec de nombreux exemples de tests.

Le TF-IDF est mesuré à l'aide de deux scores :

- **TF(t)** = (Nombre de fois que le terme/symbole t apparaît dans un document) / (Nombre total de termes dans le document).
- **IDF(t)** = \log (Nombre total de documents / Nombre de documents contenant le terme t).

Ensuite, TF-IDF est calculé comme suit : **TF-IDF(t) = TF(t)*IDF(t)**.

Enfin, nous obtenons un ensemble de vecteurs qui représentent les poids des termes dans chaque document.

7.2.3 Identification du nombre de clusters

Plusieurs techniques de clustering nécessitent comme entrée le nombre de clusters souhaités. Afin d'identifier le nombre optimal de clusters, nous avons exploité une méthode largement utilisée nommée *Elbow*. Cette méthode calcule la fonction de coût générée à l'aide de plusieurs valeurs de k (nombre de clusters) à la somme des distance au carré entre les documents et leur assignation centroïdes de cluster (cela correspond à une mesure de la variance dans les données d'entrée que nous voulons minimiser grâce au clustering).

En règle générale, nous choisissons comme nombre de clusters là où la courbe de la somme de la distance au carré commence à former un coude (elbow, d'où le nom).

La figure **7.2** montre la courbe obtenue avec la méthode *Elbow*. Sur la base de ce résultat, nous sélectionnons 6 comme nombre de clusters.

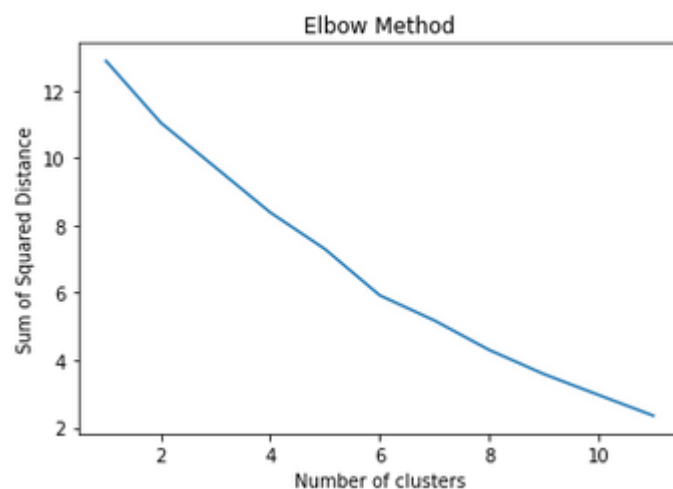


Figure 7.2. Résultat de Elbow

7.2.4 Méthode de clustering

L'algorithme de clustering k-means a été utilisé pour regrouper les documents représentés par les vecteurs générés. K-means est une méthode simple pour regrouper une collection d'observations selon un nombre spécifié de clusters (k).

L'idée de base de cet algorithme est de sélectionner au hasard k centroïdes qui correspondent au nombre de clusters, puis il attribue chaque observation au centre le plus proche en fonction de la similarité. Puis l'algorithme itère en mettant à jour les centroïdes jusqu'à ce qu'aucune autre modification du partitionnement ne soit visible. La figure 7.3 montre le résultat du clustering où chaque cluster est constitué d'un ensemble de documents, ainsi qu'un ensemble de sujets qui symbolisent chaque cluster à partir duquel le nom du microservice est sélectionné.

Remarque : Comme on peut le constater, le cluster 1 est un peu particulier car regroupe les classes "exercice_production", "plage_session", "student_statement" et "user_session" qui n'ont pas l'air similaire. Cela est dû au fait que tous les utilisateurs de l'application, que ce soit étudiants ou enseignants, travaillent sur la classe "exercice_production" qui est liée à la classe "user_session", début d'utilisation de l'application. Par conséquent, les classer dans un seul microservice assurera la scalabilité.

```
cluster 0 --> [0, 'acquiredskill', 'skill', 'theme']
cluster 1 --> [1, 'exerciseproduction', 'plagesession', 'studentstatement', 'usersession']
cluster 2 --> [2, 'profile', 'profilelevel', 'sequencelist']
cluster 3 --> [3, 'exercise', 'exerciselevel', 'nam']
cluster 4 --> [4, 'session']
cluster 5 --> [5, 'userplage', 'userrole']
```

Figure 7.3. Les différents clusters

7.3 Inférence de nom du micro-service

Nous distribuons les tables de la base de données dans leurs clusters correspondants. Le cas le plus simple est celui où le nom d'une table et tous ses attributs/colonnes se trouvent dans le même cluster. Dans ce cas, nous attribuons cette table à ce cluster. Mais parfois, le clustering peut poser des problèmes si, par exemple, pour une table donnée certains de ses attributs sont dans un cluster et d'autres dans un cluster différent. Dans ce cas, il appartient au développeur de décider si :

- Il ignore cette répartition d'attributs dans différents clusters et les regroupons ainsi dans le cluster où le nom de la table apparaît,
- Il découpe la table en tables distinctes selon le résultat du clustering.

À la fin de cette étape, chaque table de la base de données sera affectées à son clusters. Par conséquent, chaque micro-service potentiel sera nommé avec le symbole dominant dans le cluster associé. Par convention, si le nom du symbole correspond à une entité bien identifiée, comme *user_plage* ou *user_role* dans notre cas, le micro-service sera nommé en ajoutant le symbole avec le mot *ManagementService*, comme *UserManagementService*.

7.4 Conclusion

Nous avons présenté un processus centré sur les données pour l'identification des micro-services candidats comme première étape de la migration de systèmes logiciels dans une architecture basée sur des micro-services. Le processus s'appuie sur la modélisation thématique appliquée à des documents structurés à partir des symboles du modèle de données et enrichis par des mots sémantiquement liés. Bien que nous ayons utilisé le coefficient Jaccard Index, les méthodes *Elbow* et *K-means* dans le processus d'identification de micro-services, le processus proposé est générique et d'autres dictionnaires et techniques de clustering peuvent être utilisés.

Chapitre 8

Conclusion

8.1 Analyse générale

Cette unité d'enseignement très intéressant s'avère aussi très bénéfique. Il nous donne l'opportunité de booster nos recherches et de nous auto-former afin d'améliorer nos compétences.

Le but de ce projet était le refactoring d'application web suivant le modèle des micro-services. Nous avons utilisé deux approches pour décider du périmètre :

- Refactorisation manuelle d'un goulot d'étranglement du monolithe (gestion des exercices).
- Application d'une méthode d'identification des micro-services par une méthode automatisée de clustering (exploitant le schéma de la base de données).

En effet, la réalisation de ce projet nous a permis d'appréhender la notion de Refactoring d'applications web monolithique en des micro-services divers et variées, ainsi de se familiariser au phénomène de Clustering afin d'identifier des micro-services candidats.

Les précieux conseils et les fructueuses suggestions de nos encadrants ont permis de réaliser ce travail qui est le fruit d'un travail d'équipe.

Nous avons pu extraire un micro-service nommé *Exercice* et créer l'*API Getway*. Du côté automatique, on a identifié divers microservices candidats en utilisant les méthodes de machine learning.

8.2 Problèmes rencontrés

Tout au long des phases de développement et de débogage, nous avons pu appréhender plus précisément les limites de notre stratégie. Certaines de ces limites sont inhérentes au projet, ce sont ses faiblesse que nous avons listé ci-dessous :

- Les techniques utilisés dans ce projet nécessitent d'avoir un ordinateur puissant, et ce n'était pas notre cas, donc c'était difficile de travailler avec des machines qui buggent et cela demande pas mal de temps pour refonctionner.
- Le manque de temps surtout que c'était la première fois que nous travaillions sur les micro-services, ce qui nécessite une auto-formation pour bien comprendre.
- Des difficultés rencontrés pour comprendre le code de l'application monolithe, à cause du grand nombre des fichiers, près d'une centaine, organisés par catégorie (M - V - C) puis par fonctionnalité.
- Le résumé d'article scientifique écrit en anglais n'était pas une tâche facile pour nous car on devait le faire en français ce qui nécessite une autre tâche de la traduction.
- Les merges et conflits créés par Gitlab quand on travaille sur la même branche et même code depuis des machines différentes, de ce fait certains membres du groupe était obligés de bosser avec la même machine, ce qui ralentit le projet.
- La migration d'un monolithe vers un micro-service nécessite un temps et un investissement, ce qui est difficile de réaliser par un groupe de trois personnes et une durée limitée.

8.3 Perspectives

Ce projet peut avoir un avenir et continuer à être développé, il n'est pas fini dans sa globalité. Il existe des tâches ouvertes pour avoir une application 100% flexible et évolutive, parmi ces tâches on peut citer :

- La réalisation de plusieurs autres micro-services.
- La migration définitives vers les micro-services.
- L'inclusion de RabbitMQ qui est un système permettant de gérer des files de messages afin de permettre à différents clients de se communiquer très simplement.

Bibliographie

- [1] Alessandra Levcovitz, Ricardo Terra, Marco Tulio Valente. *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems*. Research gate. 2016. url : https://www.researchgate.net/publication/302892908_Towards_a_Technique_for_Extracting_Microservices_from_Monolithic_Enterprise_Systems.
- [2] Antoine Crochet-Damais. *Docker : définition, Docker Compose, Docker Hub, Docker Swarm...* Journal du net. 2020. url : <https://www.journaldunet.fr/web-tech/guide-de-l-entreprise-digitale/1146290-docker-definition-docker-compose-docker-hub-docker-swarm-160919/>.
- [3] Axios. *Qu'est-ce qu'Axios ?* Axios. 2020. url : <https://axios-http.com/fr/docs/intro>.
- [4] Colab. *Bienvenue dans Colaboratory*. Colab. 2018. url : <https://colab.research.google.com/>.
- [5] Cookie connecté. *Shéma de monolithe vs microservices*. Cookie connecté. 2020. url : https://www.youtube.com/watch?v=ucHwp1jUS2w&ab_channel=Cookieconnect%C3%A9.
- [6] Express. *Express infrastructure Web minimaliste souple et rapide pour nodejs*. Express. 2020. url : <https://expressjs.com/fr/>.
- [7] Florian Auer,Valentina Lenarduzzi, Michael Felderer, Davide Taibi. *From monolithic systems to Microservices : An assessment framework*. Science direct. 2021. url : <https://www.sciencedirect.com/science/article/pii/S0950584921000793>.
- [8] JDN. *GitLab : tout savoir sur la plateforme de DevOps open source*. Journal du net. 2022. url : <https://www.journaldunet.fr/web-tech/guide-de-l-entreprise-digitale/1443814-gitlab-tout-savoir-sur-la-plateforme-de-devops-open-source/>.
- [9] Jonas Fritzsche,Justus Bogner Alfred Zimmer, mannStefan Wagner. *From Monolith to Microservices : A Classification of Refactoring Approaches*. Software Engineering. 2018. url : <https://arxiv.org/abs/1807.10059>.

- [10] Lamyae KHAIRON. *Résumé d'article scientifique*. Ue TER. 2022. url : <https://drive.google.com/file/d/1JsIf4X8E3But-0wxPgeWxt0B06gRHSa6/view?usp=sharing>.
- [11] MDN contributors. *Node*. mdn web docs. 2022. url : <https://developer.mozilla.org/fr/docs/Web/API/Node>.
- [12] Microsoft. *Migrer une application monolithique vers des microservices à l'aide d'une conception pilotée par domaine*. Microsoft. 2020. url : <https://docs.microsoft.com/fr-fr/azure/architecture/microservices/migrate-monolith>.
- [13] Postman. *Postman (logiciel)*. Postman. 2020. url : [https://fr.wikipedia.org/wiki/Postman_\(logiciel\)](https://fr.wikipedia.org/wiki/Postman_(logiciel)).
- [14] roschan, Revoxandco, krodela bestiole. *Sous-système Windows pour Linux : Ubuntu sur Windows*. wiki-ubuntu-fr. 2021. url : <https://doc.ubuntu-fr.org/wsl>.
- [15] Wikipedia. *Google drive*. Wikipedia. 2020. url : https://fr.wikipedia.org/wiki/Google_Drive.
- [16] wikipedia. *Visual Studio - Définition et Explications*. techno-science. 2018. url : <https://www.techno-science.net/glossaire-definition/Visual-Studio.html>.
- [17] Yamina Romani, Okba Tibermacine, Chouki Tibermacine. *Towards Migrating Legacy Software Systems to Microservice-based Architectures : a Data-Centric Process for Microservice Identification*. Lirmm. 2022. url : https://www.lirmm.fr/~tibermacin/papers/2022/YRetAl_ICSA_NEI_2022.pdf.