

Project 4.A: Book search engine

Sylvain Sivacoumar, Oussama Bouibker, Alexandre Em

February 6, 2022

Abstract

Un moteur de recherche est une application web permettant à un utilisateur de trouver des ressources, que ce soit des pages web, documents texte, audio ou vidéo, à partir d'une requête de mots clés. Nous allons donc concevoir le long de ce projet une application web/mobile permettant d'effectuer une recherche approfondie des livres de la base de Gutenberg avec un système de pertinence lors des recherches et de suggestion selon les précédentes recherches de l'utilisateur et comprendra donc un système d'authentification.

Keywords — Moteur de recherche, API Rest, Application web/mobile

1 Introduction

À la suite du projet de recherche Regex: "Offline", le sujet de ce projet final consiste à concevoir un moteur de recherche suivi d'une application web et mobile qui permettrait à leurs utilisateurs de pouvoir rechercher rapidement un livre en fonction de son contenu et de leur proposer une liste de livres recommandé selon leurs précédentes recherches. Nous aurons au final une architecture ressemblant à la *Figure 1*, mais qui sera différente de celle que l'on présentera lors de ce rapport et de la vidéo par des limites d'espace allouées dans les différents services en ligne.

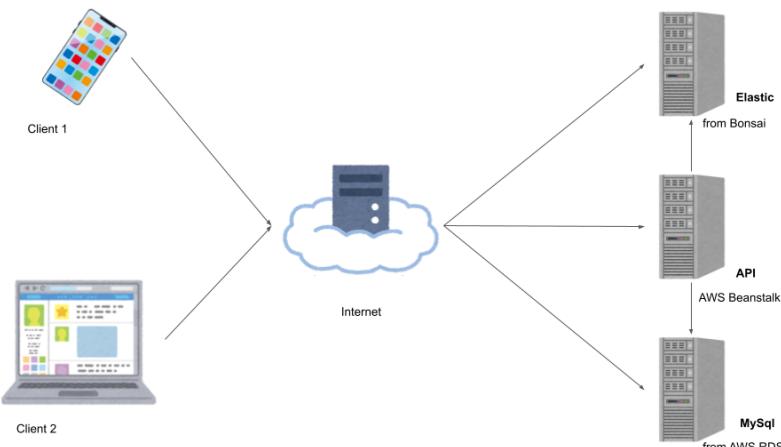


Figure 1: Architecture final du projet en environnement de "production"

Comme nous l'avons précisé précédemment, l'architecture durant ce rapport et la vidéo de présentation présente quelques légères différences. En effet l'API et les deux bases de données seront servis au sein d'une seule machine. Les données étant accessibles qu'en local, les communications entre les machines clientes et le serveur s'effectueront donc au sein d'un seul et même réseau. Mais reste valide et disponible (API) depuis ce *lien*. Par la suite, nous allons voir en détail les structure de données ainsi que les algorithmes utilisées pour implémenter le moteur de recherche de notre projet.

2 Algorithme et Structure utilisés

2.1 Base de données

Pour ce projet nous avons choisi une base de données relationnel SQL, qui gérera toutes les informations des livres ainsi que les informations utilisateurs, dont le schéma de relation est en *Figure 10*. Mais aussi une base données Elasticsearch, qui stockera le contenu des livres et permettra d'effectuer une recherche sur leur contenu. Pour chaque table, il y a un id privé qui est un nombre entier et connu uniquement par SQL et une id unique public (uuid) connu par notre app qui sera mappé avec l'id privé, ce qui nous permettra de les rendre un peu plus sécurisé.

2.2 Table d'indexation

La table d'indexation est une structure de donnée permettant d'accélérer les recherches par l'association de clés, on pourra par la suite y accéder avec des structures optimisées pour la recherche au lieu d'y accéder de manière séquentiel. On a indexé chaque livres par rapport à leurs mots présents avec leur occurrence dans le texte tout en retirant les mots vides. La pertinence sera donc calculé en fonction du nombre d'occurrence du mot indexé dans un livre. Nous avons fait le choix d'implémenter cette structure afin d'effectuer des recherches rapides selon les mots clés, comme le système de relation facilite la tâche. L'inconvénient de la table d'indexage c'est que cela prend beaucoup d'espace en mémoire comme on peut observer sur la *Figure 2*, mais aussi que cela prend énormément de temps de mettre tous les mots d'un livre dont la taille est supérieur à 10000.

テーブル	操作	行	タイプ	照合順序	サイズ	オーバーヘッド
auth0_user	表示 構造 検索挿入 空にする 削除	3	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
auth0_user_read_list	表示 構造 検索挿入 空にする 削除	2	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
authors	表示 構造 検索挿入 空にする 削除	1,067	InnoDB	utf8mb4_0900_ai_ci	208.0 KiB	-
authors_books	表示 構造 検索挿入 空にする 削除	1,954	InnoDB	utf8mb4_0900_ai_ci	192.0 KiB	-
books	表示 構造 検索挿入 空にする 削除	1,968	InnoDB	utf8mb4_0900_ai_ci	544.0 KiB	-
hibernate_sequence	表示 構造 検索挿入 空にする 削除	1	InnoDB	utf8mb4_0900_ai_ci	16.0 KiB	-
word	表示 構造 検索挿入 空にする 削除	534,944	InnoDB	utf8mb4_0900_ai_ci	100.8 MiB	-
word_index	表示 構造 検索挿入 空にする 削除	12,024,081	InnoDB	utf8mb4_0900_ai_ci	1.1 GiB	-
8 テーブル	合計	12,564,020	InnoDB	utf8mb4_0900_ai_ci	1.2 GiB	0 バイト

Figure 2: Nombre de ligne par table avec leurs poids

2.3 Elasticsearch indexation

ElasticSearch est une base de donnée NoSQL, qui nous permettra de stocker et indexer du texte.

Cette technologie nous permet donc sur ce projet d'avoir un moteur de recherche efficace et configurable que nous interrogeons en temps réel.

ElasticSearch utilise Lucene, un logiciel OpenSource utilisant Apache pour avoir une fonctionnalité de moteur de recherche.

Un noeud est un processus exécuté sur une machine et un cluster est un ensemble de noeud et permet donc l'indexation collective et la distribution des requêtes sur tout les noeuds qu'il possède.

Un index est un ensemble de document et une partition est un morceau d'un index. Un document est un ensemble de champs et propriétés définis en JSON. Ce dernier appartient à un type et est contenu dans un index.

Un type est un ensemble de document qui possèdent des similarités, c'est à dire qu'ils possèdent des mêmes champs dans le même indice.

Un indice est un conteneur logique de document en fonction des types.

Afin d'indexer des documents, nous spécifions leur type lors de l'ajout des documents dans ElasticSearch.

La force d'ElasticSearch est que ce dernier indexe nativement des documents dont les structures JSON ne sont pas les mêmes d'un document à l'autre et ceci est possible avec le typage.

Une seconde spécificité d'ElasticSearch est qu'un utilisateur peut faire plusieurs types de recherches: une recherche sur une propriété spécifique ou une recherche de façon générale.

Lorsqu'une recherche est enclenché dans ElasticSearch, ce dernier le transfert à tout les noeuds possédant une partition de l'index.

La seconde étape consiste sur chacun des noeuds, une instance d'Apache Luence qui va permettre la recherche sur la partition et donc renvoyer les résultats correspondants.

3 Implémentation

3.1 Authentification

Pour la gestion d'utilisateur avec authentification, nous avons fait le choix d'utiliser un web service tiers Auth0 qui est assez connu dans le milieu de l'authentification et donc utilisé par de nombreuses entreprises dans de nombreuses applications, ce qui permet d'instaurer une confiance entre l'utilisateur et nous par la sécurité des données qui seront entrées par celui ci. Elle propose aussi un tableau de bord intuitif qui nous permet de gérer les bases de données, les rôles, etc. et d'avoir l'activité des utilisateurs sur l'application ainsi que les accès aux logs concernant la partie authentification. Elle prend également en charge plusieurs compte tiers tel que Google, Microsoft, Apple, etc. et est totalement personnalisable (*voir Figure 3a*).

Pour le cas de notre application, nous avons implémenté un système de rôle avec permission mais nous avons aussi inclus des interactions avec notre API que nous allons détaillé par la suite.



Figure 3: Auth0

3.1.1 Rôle et permission

Nous avons donc implémenté pour notre application un système de rôle *Admin* et *User* qui comprennent des permissions prédéfinies par nous même. Ces permissions permettront ensuite à l'utilisateur d'effectuer des actions sur notre API. Nous vérifions donc les permissions attribuées à un utilisateur en décodant le token qui a été généré lorsqu'il se connecte. Bien évidemment il est possible d'attribuer des rôles manuellement afin que celui ci ait les accès à certaines fonctionnalités sans toutes les avoir ou alors de créer un rôle répondant aux besoins de genre d'utilisateur.

Dans le cas de notre API, les routes ne nécessitant aucune authentification, contiennent en général sur leur url `/public/` et sont des routes qui modifient pas nos base de données. Les routes réservées au rôle User permettent de modifier les données liées à lui même tandis que les routes réservées à l'Admin permettent de modifier l'intégralité de la base de données. L'attribution de ce rôle se fait manuellement pour une raison de sécurité.

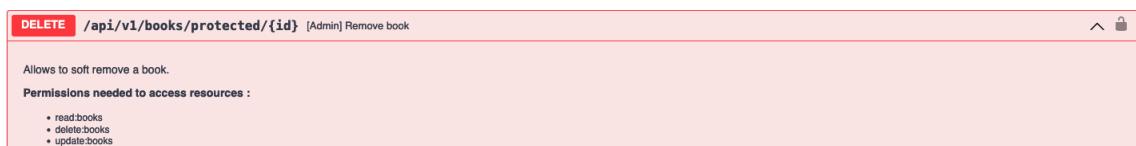


Figure 4: Endpoint protégé avec les permissions nécessaires pour accéder à la route

3.1.2 Flux

Nous avons aussi implémenté différents flux afin d'automatiser certaines actions telle que l'assignation du rôle *voir Figure 3b User* à chaque inscription et l'enregistrement d'un utilisateur dans notre base de données grâce à un appel à notre API (*voir le code 6.3*).

3.2 Migration des données

3.2.1 Méthode d'obtention des données

Afin de récupérer des livres nous avons utilisé la librairie de Gutenberg, qui propose des eBooks gratuitement. Nous allons donc en télécharger 2000 en format text afin de pouvoir les mettre sur Elasticsearch. Nous avons donc trouver un script python [2] qui permet de les télécharger puis de les extraire de leurs archives automatiquement. Une fois téléchargés, nous les filterons avec un deuxième script bash afin de n'avoir que des livres possédant plus de 10000 mots.

3.2.2 Migration vers nos bases de données

Une fois les fichiers récupérés nous allons extraire les informations des livres et créer la table d'indexation grâce à différents scripts python qui généreront par la suite plusieurs fichiers JSON qui nous permettront de remplir nos bases de données.

- **all_words.json** est une array qui contient tous les mots (hors stop words) de tous les livres
- **books_words_occ.json** qui contient tous les mots d'un livre avec leurs occurrences
- **books.json** qui contient toutes les informations d'un livre récupéré depuis le fichier text ainsi que le chemin absolu (local) vers le fichier.

Avec toutes ces metadata, nous allons donc ensuite les migrer en utilisant l'API (script située dans le répertoire test de l'API) afin de formaliser les données et d'éviter les erreurs d'incompatibilités avec notre API.

3.3 Moteur de recherche

Nous avons choisi de proposer notre moteur de recherche sous forme d'une API Restful, qui est une interface proposant différentes méthodes et qui va permettre de modifier les données selon la requête mise en entrée par l'utilisateur. Elle a été implémenté en Java Spring Boot pour sa simplicité et sa structure permettant d'avoir un code propre et ordonné. Elle a été structuré de manière à pouvoir ajouter facilement de nouvelles fonctionnalités. Nous l'afficherons ensuite dans un Swagger afin de montrer les différentes fonctionnalités de notre API.

3.3.1 Pertinence des résultats

Afin de proposer un moteur de recherche qui répond aux besoins d'un utilisateur, nous avons besoin de critère de pertinence lors de la sélection des livres qui pourrait lui satisfaire. Ainsi nous proposons deux routes de recherches:

- une route utilisant la table d'indexation précédemment défini, où nous calculons l'ordre d'importance des livres par la somme des occurrences des mots clés entrés par l'utilisateur et en filtrant les livres précédemment lu par l'utilisateur.
- une route utilisant Elasticsearch, qui calcule l'ordre des livres à affichés à l'utilisateur avec un système de scoring. On utilise pour cela les fonctions score données par Elasticsearch, qui permet d'attribuer un certain nombre de points selon des critères que l'on aura prédefini. Nous ferons la somme de ces points à la fin pour affiché en premier les livres qui ont répondu à plus de critère et donc un score élevé.

Tout cela est définie au niveau de la requête SQL/Elasticsearch.

3.3.2 Suggestion des livres

Nous allons aussi proposer à l'utilisateur un système de recommandation de 10 livres qui seront choisi selon les préférences de l'utilisateur. Ici nous avons fait le choix de regarder les livres des auteurs dont l'utilisateur aurait déjà lu un ou plusieurs livres sans bien évidemment affiché les livres qu'il aurait lu. Nous aurions pu ajouter le critère de genre de livre si nous avions des informations dessus. Cette liste serait compléter si besoin par les précédentes recherches.

3.4 Partie Front-end

Expo est un framework de React Native qui permet de développer rapidement des applications web et mobile tout en conservant le même code. Pouvant être écrit en Javascript ou en Typescript, il est facile à abordé et ne requiert presque aucune configuration Android ou IOS pour être build. Expo nous a donc paru le choix le plus pertinent car il répond bien aux besoin mobile, que ce soit en PWA (Progressive Web Application) ou en Application Native tout en gagnant en temps de codage par son côté hybride.

3.4.1 Services

Les services nous permettent de stocker les fonctions d'appels au Back-end, nous permettant d'interagir avec ce dernier. Il existe deux types de routes :

Les routes sécurisés, qui avec un token de connexion permet d'accéder au ressource de la route.

Les routes publique, qui est une ressource commune entre les utilisateurs anonymes et les utilisateurs connectés.

3.4.2 Pages

Afin de pouvoir afficher un rendu front sur notre application mobile/web, nous avons implémenter un système de page permettant l'affichage structuré de nos données de nous récupérons à partir des services.

Pour permettre une navigation entre nos pages, nous avons implémenter un système de route entre les pages permettant un changement de page optimale entre elles. Une barre de navigation en bas de l'écran à été ajouté pour permettre à l'utilisateur un meilleur confort de navigation.

Nous verrons ici les différentes pages et leurs utilité ici :

- La page Home contient une liste de route de plusieurs page qui seront affiché dans son render
- La page Profil contient l'historique de lecture de l'utilisateur
- La page Book contient la liste des suggestions utilisateur (suggestions calculé en fonctions des livres déjà lu susceptible de plaire) et la liste de tout les livres disponible dans le back-end
- La page Search permet la recherche des livres en fonction d'une date de publication, auteur, titre, défaut et regex (sur les mots du contenu des différents livre)
- La page Author contient une liste des auteurs, avec leurs livres associés
- La page Authentification contient deux boutons pour permettre d'accéder au front: un bouton continue permettra à l'utilisateur de pouvoir profiter du contenu publique et de façon anonyme mais ne pourras pas accéder a la lecture des livres et avoir une suggestion de livre proposé en fonction des dernières lecture. Il n'aura pas aussi une page de profil associé (anonyme).
Le second bouton permet quant à lui de faire une connexion ou inscription avec Auth0.

3.4.3 Composants

Pour permettre le partitionnement de code, nous avons créer des composants que nous utilisons sur nos pages. Ces composants nous font gagner beaucoup de temps, de lisibilité et plus de performance coté développement. Le composant CardAuthor permet l'affichage des différents auteurs, avec leurs livres associés sur une fenêtre pop-up. Le composant CardBook permet l'affichage des livres avec l'image de couverture du livre, son titre et son auteur. A savoir qu'il faut être connecté pour pouvoir accéder au contenu du livre et que ce dernier est affiché lorsque l'utilisateur clique sur le bouton "Voir plus", une fenêtre pop-up apparaît avec le contenu du livre.

Il existe deux types de CardBook: les cartes pour la page de profil et les cartes pour la liste des livres suggérer/tout les livres. Cette différence est notable car les deux cartes affiche toutes deux les spécifications de livres mais pas de la même

disposition afin de rendre l'utilisation du site plus agréable et moins redondante. Le composant SearchBar permet l'affichage d'une barre de recherche. Elle permet aussi d'enregistrer les mots que l'utilisateur entre.

3.5 Déploiement

Afin de pouvoir proposer notre application à nos utilisateurs, nous devons nous assurer qu'aucun bug ne sorte et nous fasse perdre la dernière version stable et donc l'accès à l'application. Cela devra donc passer par des phases de test unitaires, fonctionnels et d'intégration, mais aussi à la vérification du build de l'application. Nous avons donc mis en place une pipeline CI/CD grâce à Git Action, qui vérifie tout cela et la déploie une fois tous les checks passés sur AWS elastic beanstalk qui nous informera l'état de santé de notre application en modifiant le statut du Git Action. Cette pipeline sera executé à chaque push git ou merge de PR qui seront effectués sur une branche spécifique *preview*. Nous n'avons pour le moment pas eu le temps d'écrire tous les tests afin de compléter la pipeline. Nous n'avons pas eu également eu le temps d'écrire de pipeline pour le déploiement de la partie cliente de l'application. Elle serait basé sur le même process, mais serait publié sur les différentes plateforme PlayStore pour la version Android et Vercel, Firebase ou autre pour la version web, Expo rendant simple le déploiement par des configurations simplifiés par rapport à React Native Vanilla par exemple.

4 Phase de test

4.1 Test de performance

Afin de tester la performance de nos différentes routes de recherche, nous allons effectué plusieurs phase de test: recherche d'un mot existant dans la base de donnée, recherche de plusieurs mots et recherche d'un mot aléatoire. Nous avons donc pour cela créer un notebook Jupyter qui va appeler les différentes routes de recherche (SQL/Elasticsearch) que nous allons stocker dans deux dictionnaires distinctes qui contiendront chacun le mot recherché en clé et un tableau de temps mesuré en valeur des dictionnaires.

Pour chaque phase de test, nous avons effectué $N=100$ (par soucis de temps) recherches sur chaque mot réparti sur plusieurs processus, puis nous calculons la moyenne ainsi que son écart type. Pour la première recherche (simple), nous avons repris et modifier le premier projet *Offline* afin d'effectuer les recherche sur l'intégralité des livres et de comparer les résultats obtenu.

Remarque: Les diagrammes que nous allons présenter ci dessous seront tous en échelle logarithmique, pour une question de visibilité.

4.1.1 Recherche d'un mot simple

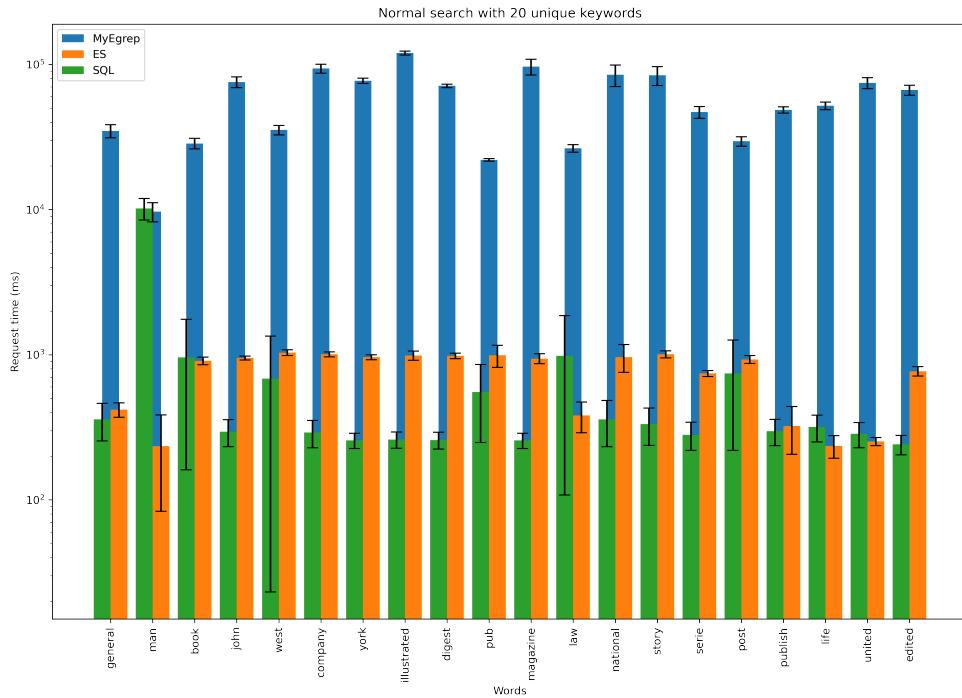


Figure 5: Diagramme du temps d'exécution moyen d'une recherche d'un mot

4.1.2 Recherche de plusieurs mots

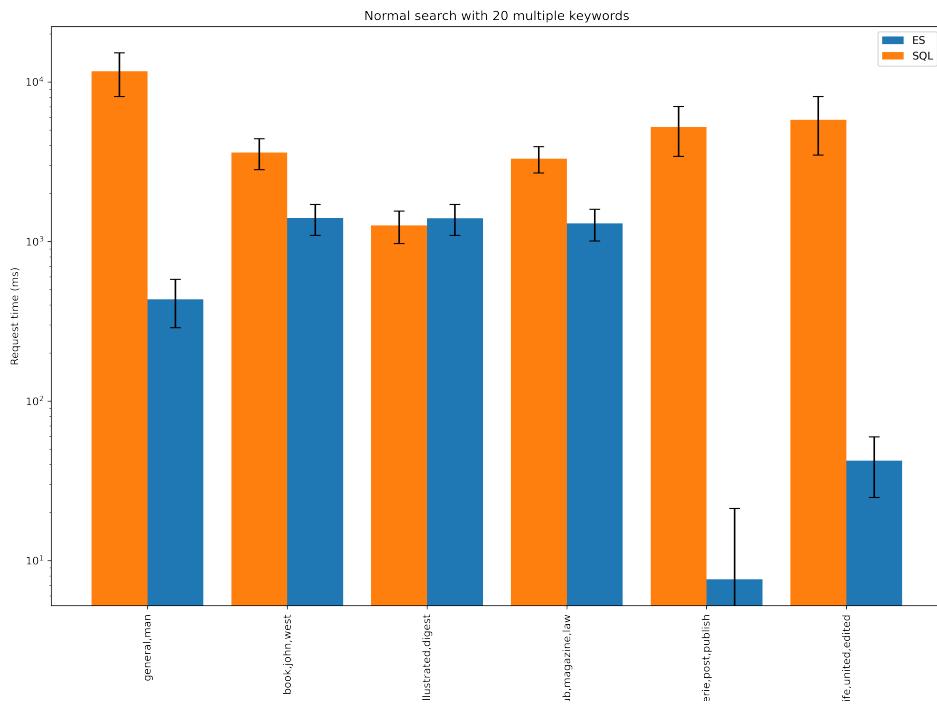
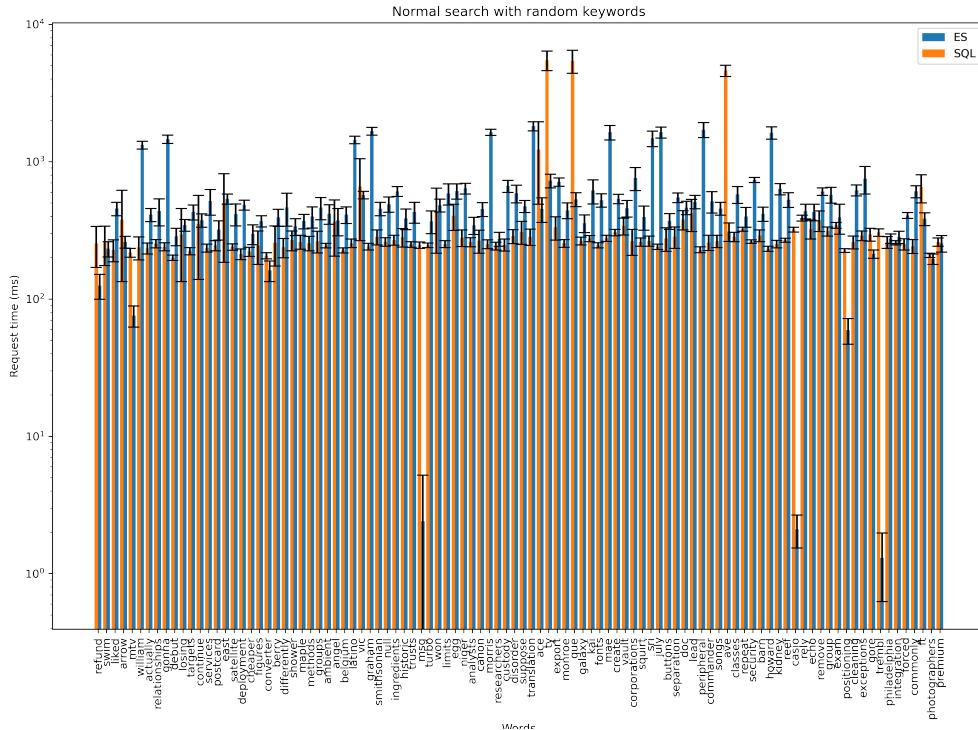
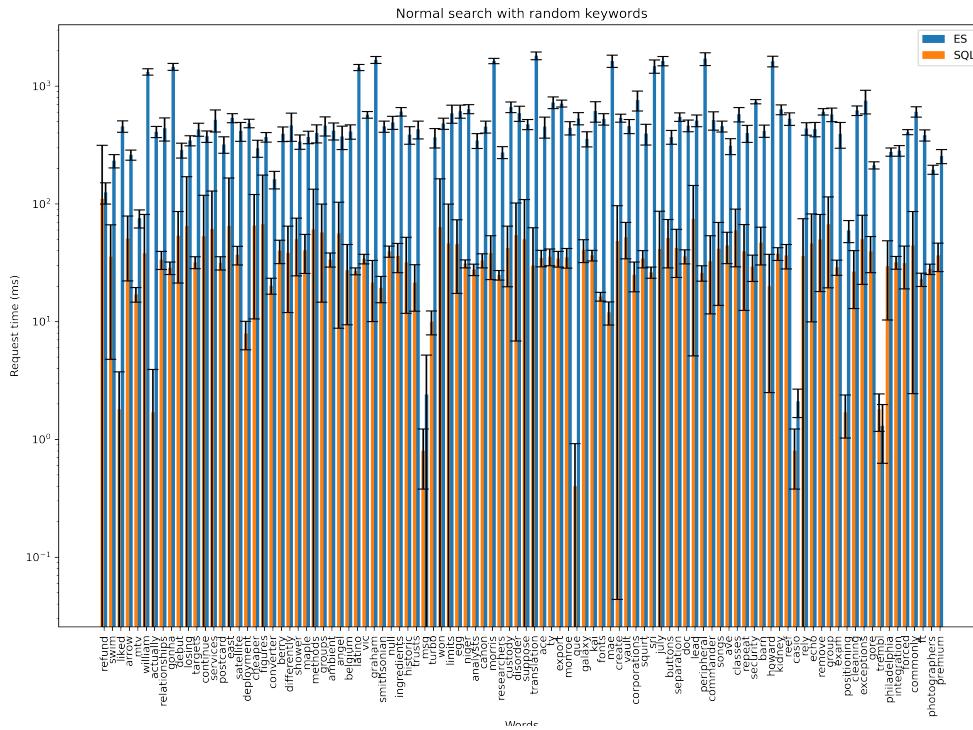


Figure 6: Diagramme du temps d'exécution moyen d'une recherche à plusieurs mots

4.1.3 Recherche simple de 100 mots aléatoire



(a) Diagramme du temps d'exécution moyen pour des requête SQL avec *contains*



(b) Diagramme du temps d'exécution moyen pour des requête SQL avec *equal*

Figure 7: Test de 100 mots aléatoire, qui donc peuvent ne pas exister dans un des livres

4.1.4 Temps moyen des recherche en général

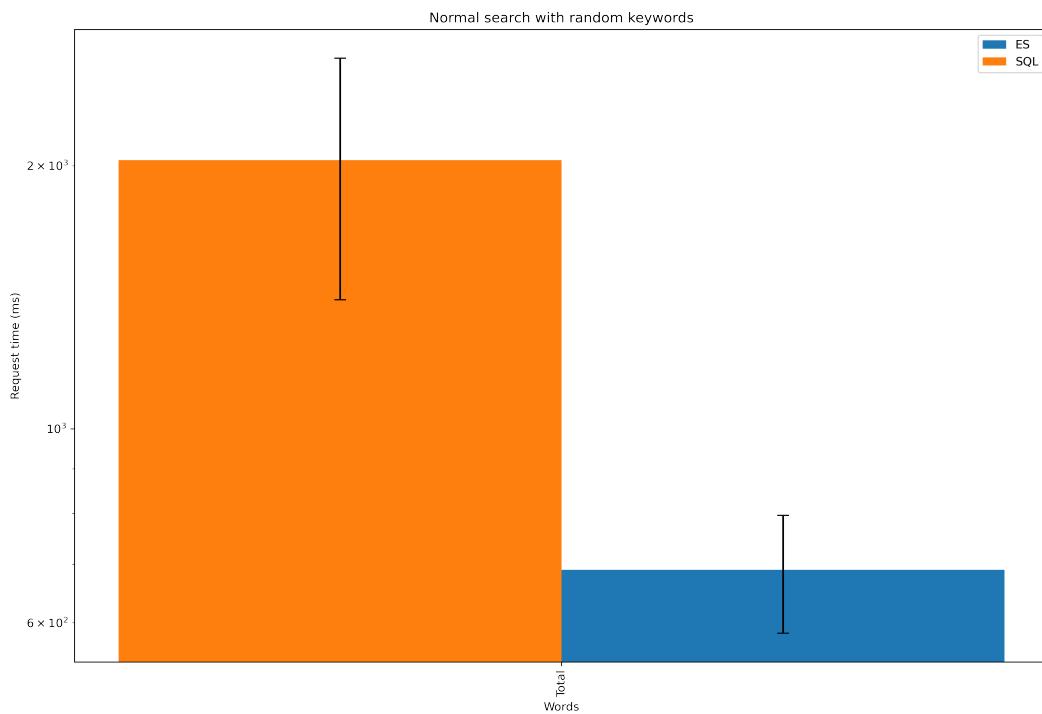


Figure 8: Moyenne de tous les diagrammes générés précédemment

4.2 Observation

4.2.1 Recherche d'un mot simple

Nous pouvons déjà voir que la recherche avec le projet 1 *Offline*, qui reprend les algorithmes du livre *Aho - Compilers - Principles, Techniques, and Tools* n'est pas du tout efficace. On pourrait aussi essayer de répartir les recherches sur plusieurs threads et observer le temps d'exécution, mais on peut dire que la recherche d'un mot sans indexation permet d'avoir une base de données plus légère mais avec un traitement par requête beaucoup plus long qu'avec indexation. Comme nous pouvons le constater pour les recherches effectuées avec SQL et Elasticsearch. On peut voir que les temps de recherche d'un mot simple peuvent varier selon le mot dans le cas des requêtes SQL. En effet, on peut voir qu'il a beaucoup plus de mal avec le mot *man*, qui peut s'expliquer par le fait que ce mot puisse s'intégrer sur plusieurs mots (en tant que préfixe et/ou suffixe) et donc si on effectue une recherche exacte (donc sans *contains*), on aurait un temps d'exécution beaucoup moins important du fait que les mots soient indexés mais qui retournerait beaucoup moins de résultat. À part ce genre de cas, on peut voir qu'en général les traitements des requêtes SQL (avec *contains*) sont moins longs que ceux d'Elasticsearch dans le cas d'une recherche avec un seul mot clé, mais que les traitements des recherches d'Elasticsearch sont plus constants.

On peut aussi remarquer que l'écart type de certaines recherches SQL sont énormes,

si on observe les temps récupérés on peut constater que le temps de traitement de la première requête est toujours beaucoup plus longue que les suivantes, ce qui explique cette écart.

4.2.2 Recherche de plusieurs mots

Cette fois ci on peut voir que le temps de traitement pour les recherches de plusieurs mots avec SQL prennent beaucoup plus de temps que celles effectuées avec Elasticsearch qui varie selon le nombre de mots clés et donc plus il y a de mots clés dans la recherche plus il match rapidement avec les documents. Tandis que le traitement avec SQL dépend juste de l'intégration des mots clés dans d'autres mot. On aurait donc un temps nettement inférieur si les recherches étaient pour des mots exactes.

4.2.3 Recherche simple de 100 mots aléatoire

Dans ce cas d'observation, nous allons comparer les temps de traitement de recherche exacte ou non. Nous allons utilisé une API externe qui retourne un certain nombre de mots (anglais) aléatoire. Ces mots pourront alors ne pas être indexé.

Nous allons donc tester sur 100 mots aléatoire dont 10 requêtes par mots.

On peut voir en général que les requêtes effectuées par SQL prennent moins de temps que les requêtes Elasticsearch mais qu'encore une fois, pour les cas des mots dont le nombre de mots où il peut être un suffixe et/ou un préfixe est élevé, le temps de traitement est multiplier par environ 10ms. En revanche lorsque l'on fait des requêtes de recherche SQL avec *equal*, on peut voir que le temps de requête baisse considérablement.

4.2.4 Temps moyen des recherche en général

On peut donc voir qu'en général, Elasticsearch excelle dans le domaine du moteur de recherche mais selon le type de requête que l'utilisateur souhaiterait effectuer, il pourrait être plus pertinent d'alterner avec SQL dans le cas où le/les mot recherché sont exactes.

5 Conclusion

MyLibrary est une application mobile et web permettant d'effectuer des recherches sur la base de Gutenberg. Ce projet très intéressant nous a permis de manipuler plusieurs technologies tels qu'Elasticsearch, Expo, AWS, etc mais aussi de voir différents algorithmes et structures de données utilisés par la majorité des moteurs de recherche. Plutôt efficace avec un temps moyen d'une seconde par recherche pour des cas de recherches complexes et accessible. Nous pouvons donc dire que le but de ce projet à été atteint. Maintenant la table d'indexation est fixe et ne se remplit pas lors d'ajout de livres dans la base de données, nous devons donc trouver un moyen d'automatiser le processus de mise à jour de celle ci avec une planification avec cron par exemple, mais aussi distribuer les tâches dans différents processus afin de gagner en temps. Et pourquoi pas automatiser aussi l'ajout des nouveaux livres de Gutenberg afin d'avoir une base toujours à jour.

6 Annexe

6.1 Gestion de projet

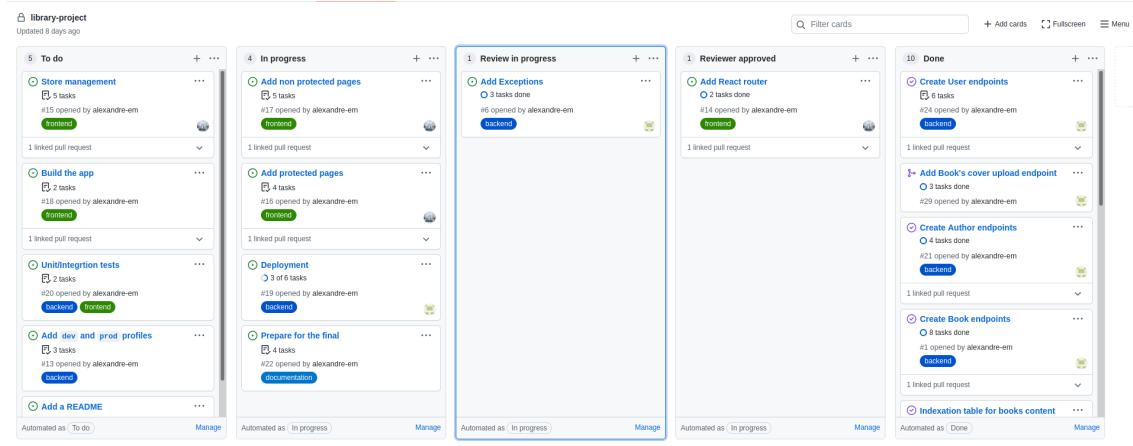


Figure 9: Tableau de bord Github pour la répartition des tâches

6.2 Base de données SQL

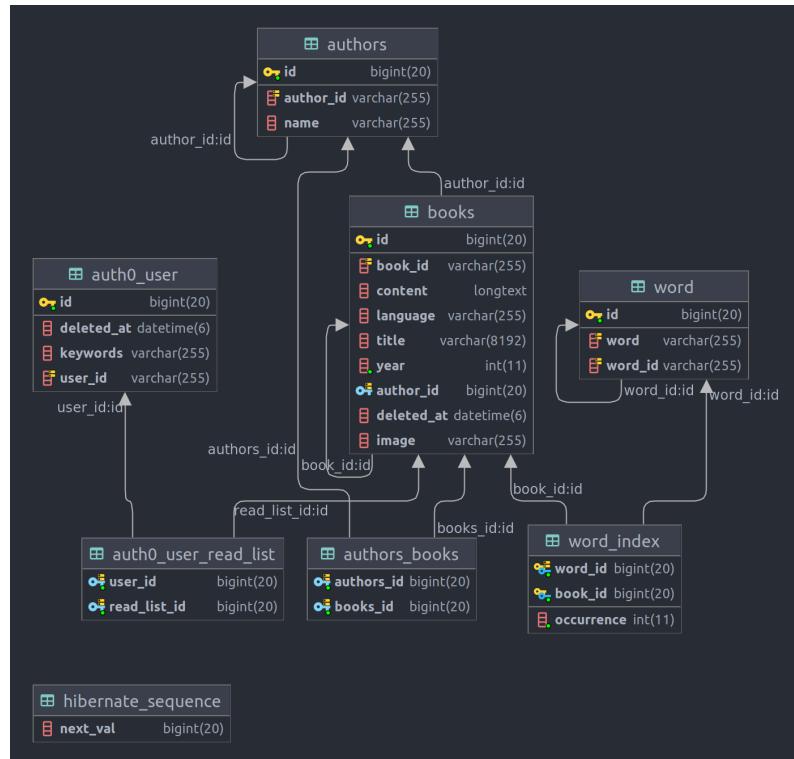


Figure 10: Diagramme entité-relation de la base de données mylibrary

6.3 Authentification

Voici les scripts implémentés dans le flux d'Auth0, qui ont permis l'automatisation d'ajout de rôle User à un nouvel utilisateur ainsi que l'ajout de l'utilisateur dans notre base de donnée afin de pouvoir gérer les suggestions.

```
// Inject User's role rule
function (user, context, callback) {
    const ManagementClient = require('auth0@2.27.0').ManagementClient;

    const management = new ManagementClient({
        token: auth0.accessToken,
        domain: auth0.domain
    });

    const count = context.stats && context.stats.loginsCount ?
        context.stats.loginsCount : 0;
    if (count > 1) {
        return callback(null, user, context);
    }

    const params = { id : user.user_id };
    const data = { "roles" : ["rol_l1zHEj10uyAcjDyZ"] };

    management.users.assignRoles(params, data, function (err, user) {
        if (err) {
            console.log(err);
        }
        callback(null, user, context);
    });
}
```

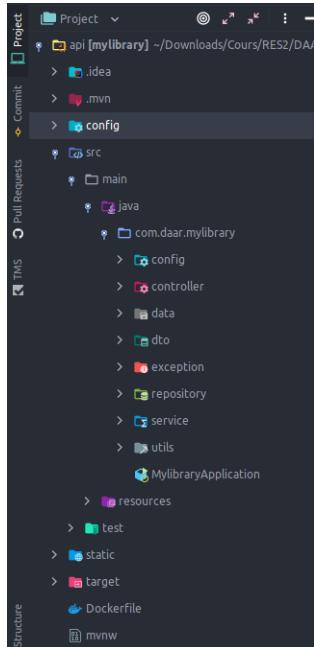
Code 1: Script permettant l'injection automatique du rôle user lors de la première connexion

```
// Register an user on our db
const axios = require('axios');

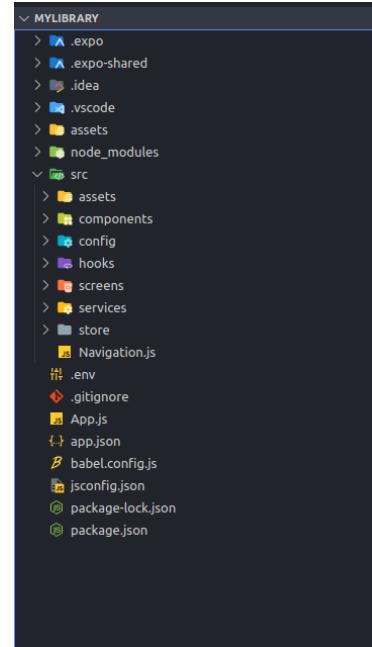
exports.onExecutePostUserRegistration = async (event) => {
    await axios.post(`#${API_URL}/api/v1/user/${event.user.user_id}`);
}
```

Code 2: Script permettant l'enregistrement automatique de l'utilisateur dans notre base de donnée

6.4 Structure du code



(a) Structure du code du backend



(b) Structure du code du frontend

Figure 11: Structure du code

6.5 Swagger

Portail proposé à l'utilisateur afin d'utiliser les services proposés par notre API

Figure 12: Swagger de l'API

6.6 Screen de l'application Front

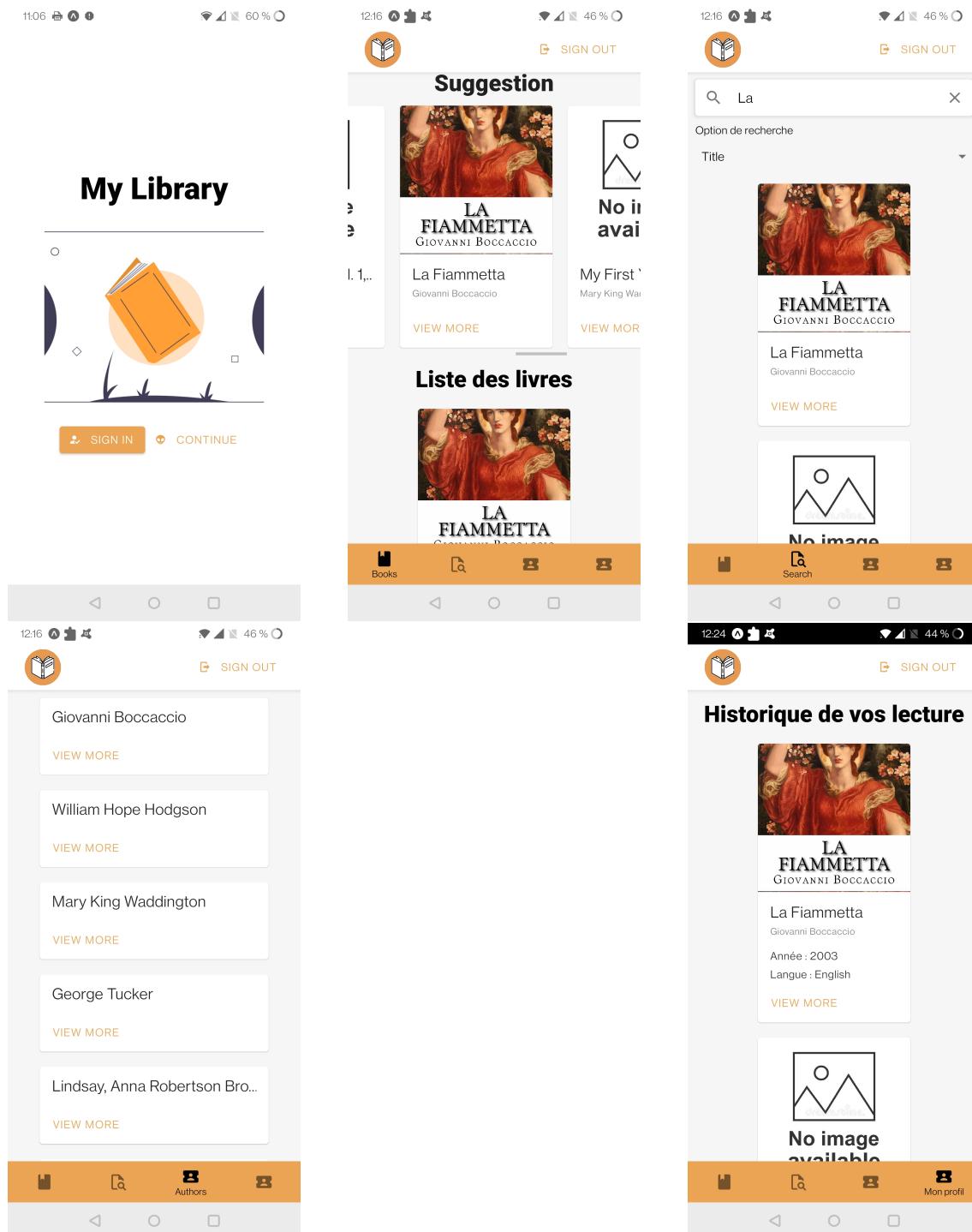


Figure 13: Différentes vue de l'application *My Library*

6.7 Pipeline de Déploiement

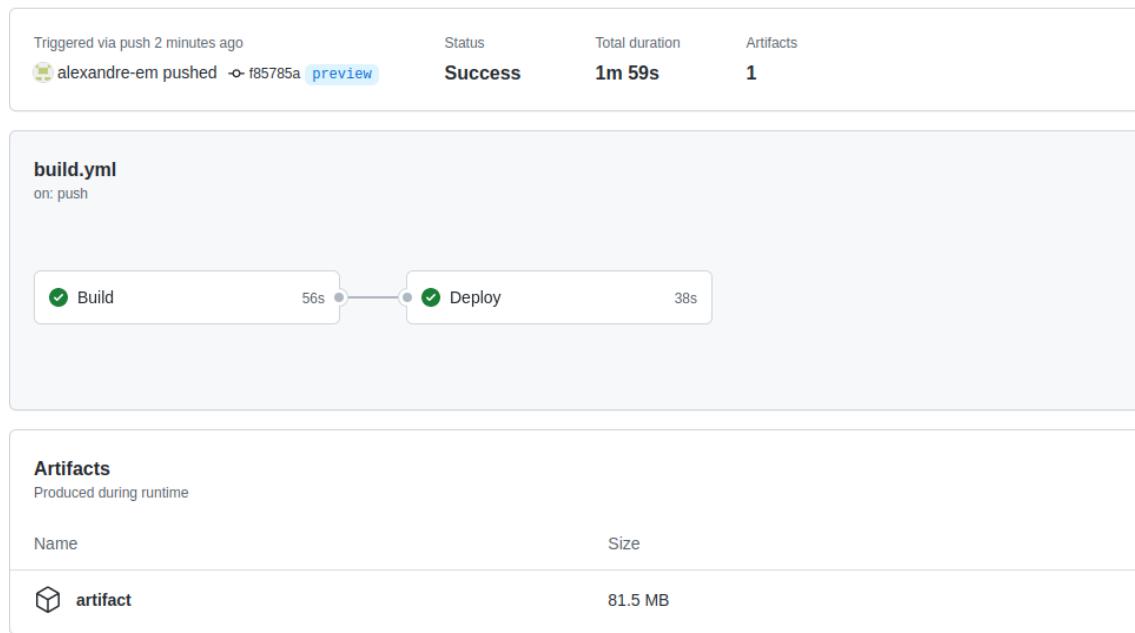


Figure 14: Pipeline pour le déploiement automatique de l'API sur AWS BS

References

- [1] Project Gutenberg - www.gutenberg.org
- [2] Script to download the gutenberg's ebooks
- [3] Stop words list - countwordsfree.com/stopwords
- [4] Interrogez efficacement vos bases de données - Mukesh Mithrakumar
- [5] Fonctionnement Elasticsearch - Openclassroom
- [6] Elastic vs Lucene - Overstack
- [7] Swagger de notre API