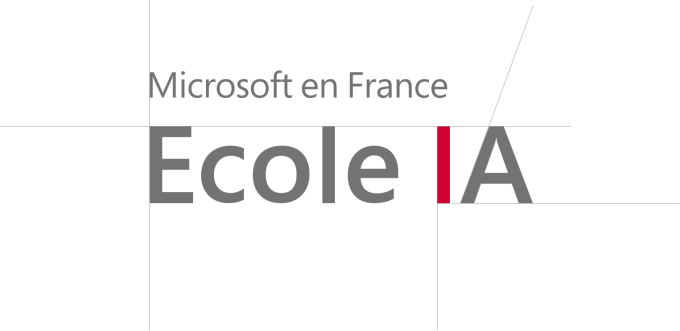
**FLORENT VANHOLLEBEKE Session mai 2023**

**PROJET CHEF D’ŒUVRE : E1**

**VOKALPEDIA**

****

****

**PLAN**

**Introduction**

**Partie I : La conception du projet**

**A) Compréhension du besoin client**

**B) Etat de l’art**

**C) Choix techniques opérés**

**Partie II : La mise en œuvre du projet**

**A) Gestion de projet**

**B) Situation du livrable**

**Partie III : Le bilan du projet**

**A) Retours d’expérience**

**B) Améliorations envisageables**

**Conclusion**

**Annexes**

**Introduction** :

En France, plus de 200 000 personnes souffrent de déficience visuelle sévère, et plus de 900 000 personnes sont classifiées comme malvoyants moyens.

Dans le monde, le nombre de malvoyants est supérieur à 30 millions de personnes, chiffre appelé à tripler en raison de l’expansion démographique et du vieillissement de la population.

Toujours en France, près de 850 000 personnes souffrent d’un handicap moteur.

A une époque où le numérique prend une place de plus en plus prépondérante, l’incapacité à observer ou interagir avec un écran se révèle être un facteur encore plus discriminant.

Wikipédia.org, encyclopédie universelle multilingues et open source, a été créée en 2001. Elle constitue aujourd’hui une source majeure d’informations, et une véritable ouverture vers le monde.

Fonctionnant sur le principe du collaboratif, son accès gratuit, est alimenté et contrôlé par ses propres contributeurs et utilisateurs.

Wikipédia comprend des articles dans plus de 300 langues, et si l’anglais est la langue la plus utilisée, il n’en demeure pas moins que des millions de pages existent également en français.

Les articles de Wikipédia comportent du texte mais également des images.

Porteuse des valeurs d’universalité et d’inclusivité, Wikipédia doit donc pouvoir s’adresser à tous et notamment aux personnes souffrant d’un handicap.

**Partie I : La conception du projet**

Dans cette première partie, nous aborderons les différentes étapes qui ont conduit à l’élaboration de l’application Vokalpédia, hors cadre de travail.

**A) Compréhension du besoin client**

**1) Problématique**

Les assistants personnels, tels que Google Home ou Amazon Alexa, se démocratisent et tendent à améliorer la vie de ses utilisateurs.

Néanmoins, quiconque a déjà testé l’option d’accessibilité de Microsoft Windows s’aperçoit de la difficulté à obtenir un résultat précis dans la consultation de sites web.

Toutefois qu’en est-il des personnes malvoyantes ou souffrant d’handicaps moteur ?

A priori, malvoyants et écrans paraissent des concepts antinomiques.

Fort heureusement, et depuis déjà un certain temps, les éditeurs de logiciels travaillent à l’inclusivité des malvoyants ou handicapés moteur dans la sphère numérique. A ce titre, certains systèmes d’exploitation (comme les Microsoft Windows par exemple) et navigateurs web s’attèlent à inclure des mécanismes d’aide aux malvoyants et autres handicaps. Nous parlons des fonctionnalités d’accessibilité.

Ainsi, il semble qu’il existe une place pour le développement de solutions informatiques spécialisées en *Speech-to-Text* et *Text-to-Speech*.

Le segment utilisateur privilégié (appelé aussi « persona » en méthode Scrum, voir infra) porte donc sur les personnes malvoyantes ou handicapés moteur.

C’est dans cette optique qu’est né le projet d’application Vokalpédia. Cette application vise à offrir aux utilisateurs, en particulier handicapés, la faculté de parcourir l’encyclopédie Wikipédia avec le minimum d’interactions.

L’objectif de l’application, comme le jeu de mots l’intitulant, est de parcourir Wikipédia par le biais vocal.

Qui dit vocal, signifie en termes de technologie d’intelligence artificielle, la synthèse vocale (*Text-to-Speech*) ainsi que la reconnaissance automatique de la parole (*Speech-to-Text*).

**2) Formalisation du besoin client**

Pour développer une solution informatique, encore faut-il connaitre le besoin à remplir pour satisfaire un utilisateur. Il s’agit du but et de l’utilité de l’application.

Nous traiterons donc ici des différents besoins que peuvent rencontrer un utilisateur.

Il y a différentes méthodes pour formaliser le besoin utilisateur. J’ai choisi d’utiliser les méthodes Agiles, et notamment le principe du *Product Backlog* regroupant des *user stories*. Nous ne rentrerons pas maintenant dans le détail, la méthode de gestion de projet utilisée sera expliquée en infra (ie, en Partie II : A).

A cet effet, il est courant de formaliser les besoins de la façon suivante : « en tant qu’utilisateur… ».

Ainsi, voici les principaux besoins utilisateurs recensés :

*« En tant qu’utilisateur, je dois pouvoir me connecter au site et m’authentifier ».*

*« En tant qu’utilisateur, je dois pouvoir être rassuré quant à la sécurisation de mes données personnelles ».*

*« En tant qu’utilisateur, je dois pouvoir obtenir de l’aide dans l’utilisation de l’application ».*

*« En tant qu’utilisateur, je dois pouvoir utiliser l’application avec la voix ».*

*« En tant qu’utilisateur, je dois pouvoir choisir la page désirée ».*

*« En tant qu’utilisateur, je dois pouvoir consulter uniquement les grandes sections du sommaire ».*

*« En tant qu’utilisateur, je dois pouvoir consulter le sommaire dans son intégralité ».*

*« En tant qu’utilisateur, je dois pouvoir consulter uniquement un article au sein d’une page ».*

*« En tant qu’utilisateur, je dois pouvoir consulter une page dans son intégralité ».*

*« En tant qu’utilisateur, je dois pouvoir mettre en pause et reprendre la lecture audio ».*

*« En tant qu’utilisateur, je dois pouvoir obtenir une description des images présentes sur la page ».*

*« En tant qu’utilisateur, je dois pouvoir bénéficier d’un échange vocal fluide en obtenant des retours sous délai raisonnable ».*

Recenser les besoins de l’utilisateur nous permet désormais de définir la solution technique qui va pouvoir répondre auxdits besoins.

**B) Etat de l’art**

Ayant recensé les besoins, il convient désormais de rechercher la technologie optimale pour notre projet parmi le large éventail possible, et ce eu égard à nos compétences. Cela commence par recenser les technologies existantes sur le sujet nous préoccupant.

L’application doit avoir recours à plusieurs modèles d’intelligence artificielle. Son fonctionnement doit reposer sur la combinaison d’un *Text-to-Speech*, d’un *Speech-to-Text* mais aussi du *Captioning image* (description du contenu d’une image).

1. Text to speech

Le text to speech est une technologie visant à convertir à l’oral un contenu écrit. De nombreuses applications modernes utilisent cette technologie telles que les assistants vocaux ou les livres audio.

Au cours du temps, les modèles de text to speech ont évolué. Les premiers fonctionnaient sur des techniques de synthèse de formants, puis par synthèse par concaténation. Les derniers modèles plus évolués utilisent les réseaux de neurones. Les voix rendues sont plus naturelles et expressives.

Parmi les modèles open source basés sur des réseaux de neurones, nous pouvons citer :

* Tacotron développé par Google (le réseau de neurones se2seq). Il génère des spectrogrammes de la parole à partir d’une séquence de caractères. Ces spectrogrammes sont ensuite converties en ondes sonores par le biais d’un vocodeur (exemple, WaveRNN).
* WaveNet développé par DeepMind. Il repose sur un modèle génératif de pointe, et utilise un réseau de neurones convolutifs profonds, afin de produire des ondes sonores directement issu du texte. En réalité, Tacotron et WaveNet sont souvent associés.
* FastSpeech repose sur la durée prédictive pour générer la parole. Il est plus rapide que Tacotron car n’utilise par le mécanisme de l’attention.

En dehors de l’open source, de nombreuses entités commerciales proposent des sources de text to speech.

Nous pouvons citer Google Cloud Text-to-Speech, Amazon Polly, Microsoft Azure Cognitive Services Speech et IBM Watson.

1. Speech to text
2. Captioning image

**C) Choix techniques opérés**

Comme vu précédemment, l’application doit comporter trois modèles d’intelligence artificielle. Ils vont être localisés côté *back-end*. Elle a également besoin d’avoir en sus du *back-end,* un *front-end* ainsi qu’une base de données pour remplir sa mission.

Le *back-end* est la partie non visible de l’application, celle qui opère côté serveur, alimentée dans notre cas par une base de données et la possibilité de requêter le site Wikipédia pour l’agrandir. La partie *back-end* est le cœur de l’application. Celle où nous requêtons Wikipédia, où nous délivrons les informations obtenues après divers traitements, dont celui des modèles d’intelligence artificielle.

Le *front-end* est la partie visible de l’application, celle que l’utilisateur apprécie visuellement et avec laquelle il peut interagir directement.

La partie *front-end* de Vokalpédia sera volontairement minimaliste, elle se destine prioritairement à des malvoyants et n’a donc par définition pas vocation à être excessivement surchargée (elle tentera au contraire d’être efficiente au niveau de l’UX, c’est-à-dire l’expérience utilisateur). Le *front-end* se doit d’être utilisable avec le moins d’interactions possibles.

Après étude des différentes solutions techniques envisageables, divers choix ont été opérés dans une optique constante d’efficacité et de fluidité de l’application.

1) Langages de programmation et frameworks

a) Généralités

Pour les langages de programmation, j’ai retenu **Python** pour le *back-end* en raison de sa simplicité et de sa grande familiarité avec les domaines de l’intelligence artificielle. Il dispose également de très nombreuses bibliothèques facilitant le code et la connexion à diverses API.

Javascript aurait pu être également candidat à la gestion du *back-end* en ayant recours à Node.js. L’application aurait été entièrement écrite en Javascript. Les API que j’ai choisies d’utiliser auraient pu fonctionner avec Javascript.

Toutefois, je maitrise mieux Python et le framework Django m’était connu contrairement à Node.js.

Pour le *front-end*, naturellement le **Javascript** a dû être retenu. Eu égard à l’objet de l’application, son recours ne s’est pas limité à quelques effets visuels ou interactivités superfétatoires pour l’utilisateur. Le Javascript et certaines de ses bibliothèques natives ont été employées activement pour mener à bien le projet.

b) Le *front-end*

Le *front-end* est réalisé en **Javascript vanilla, HTML et SCSS**. Je n’ai pas retenu de framework *front-end* tels que React /AngularJS / VueJS pour un unique fichier Javascript. Cela aurait inutilement rallongé le temps de développement dans ce cas. Toutefois, un framework était initialement prévu, j’expliquerai dans la dernière partie du rapport pourquoi il n’en est finalement rien.

En revanche, l’application utilise des bibliothèques natives à Javascript, en particulier les classes *SpeechRecognition* et *SpeechSynthesisUtterance* qui permettent la reconnaissance automatique de la parole et la synthèse vocale. Ces classes seront revues un peu plus en détails en infra.

c) Le *back-end*

Pour faciliter le développement et donner comme son nom l’indique un cadre de travail, le framework **Django** écrit en Python a été utilisé.

Django est un framework dit MVT (Model-View-Template), il s’agit d’une légère originalité à la traditionnelle architecture MVC (Model-View-Controler). Avec l’architecture MVT, la vue est ce que l’on nomme contrôleur (dans le MVC), et le Template du MVT est l’équivalent de la vue du MVC. Au-delà de ces faux-amis en termes de nommage d’architecture, l’idée générale demeure identique.

Un des avantages du framework Django réside dans son implémentation native d’une importante couche de sécurité, notamment par son fonctionnement avec les jetons d’authentification (*CSRF tokens* dans les formulaires par exemple), et de ses modèles utilisateurs et groupes natifs.

De surcroit, Django dispose dès sa création d’une base de données extrêmement légère : SQLite3. Elle est très simple à mettre en place et repose sur le principe du SQL. La conception de la base de données à partir de classes (UML) plutôt qu’en pur SQL est également une simplification. Les migrations se font relativement aisément et le requêtage de la base de données grandement simplifiée par le biais de l’**ORM** (*Object Relational Mapping*) de Django.

Dans le cas présent, la base de données étant relativement restreinte, **SQLite3** est amplement suffisant. Dans le cadre d’un projet plus important et/ou une mise en production sur le marché d’une telle application, j’aurais utilisé une base de données plus robuste telle que PostgreSQL par exemple.

Une alternative à Django aurait été le framework Flask, il est plus léger que Django, mais il n’a pas nativement : un système d’authentification avec utilisateurs/groupes, une base de données et un ORM.

Il est possible d’utiliser un ORM comme SQLAlchemy avec Flask, mais in fine cela revient à complexifier le projet en interfaçant plusieurs bibliothèques, alors que Django remplit en soi tous les outils nécessaires.

Pour toutes ces raisons, le choix du framework Django paraissait naturel.

2) Les bibliothèques et API externes utilisées

L’application repose sur le recours à plusieurs API (*Application Programming Interface*) externes. Une API permet la communication entre différents programmes.

Vokalpédia utilise principalement trois bibliothèques et API externes.

Chronologiquement dans le parcours utilisateur, la première utilisée est la bibliothèque de *Speech-to-Text* de Javascript, plus précisément la **Web Speech API de Mozilla**. Elle s’appelle API, mais concrètement il s’agit d’une sorte de bibliothèque native à Javascript utilisable directement dans le navigateur web. Elles offrent notamment les classes *SpeechRecognition* et *SpeechSynthesisUtterance.* La première permet la reconnaissance automatique de la parole tandis que la seconde permet la synthèse vocale. Elles constituent une part très importante du script Javascript écrit pour l’application puisque l’essentiel de l’application repose littéralement sur un dialogue vocal entre l’utilisateur et le *back-end*. Les autres méthodes Javascript que j’ai écrites n’ont pour but que de permettre ce dialogue.

La seconde utilisée est celle de Wikipédia. La communauté Wikipédia a développé une API, téléchargeable avec le gestionnaire de paquets pip. Il y a en réalité deux bibliothèques **(wikipedia, et wikipediaapi**, ce qui nécessite deux imports de bibliothèques) permettant de requêter le site Wikipédia. Ces bibliothèques offrent différentes classes et méthodes afin d’obtenir le contenu de la manière le plus aisée possible.

L’alternative à ces bibliothèques était de scrapper le site Wikipédia avec des bibliothèques telles que Beautifulsoup ou Sélénium. En réalité, j’utilise aussi **Beautifulsoup** à certains moments de l’application car les bibliothèques wikipédia ne se suffisent pas à elles seules pour mener à bien le projet comme je le souhaitais. Sélénium était un choix possible mais Beautifulsoup parait plus adaptée au simple *scrapping*, alors que Sélénium semble plus tournée vers une automatisation approfondie.

Ensuite, le contenu qui est obtenu est utilisé dans le cadre de l’utilisation d’une seconde API. Il s’agit de celle de **Microsoft Azure Cognitive Services Speech**. Le cloud de Microsoft Azure héberge un modèle entrainé en *Text-to-Speech* pour transformer une chaine de caractères entrante en un fichier .wav en sortie qui contient de l’audio (la transformation audio de la chaine de caractères).

Etant donné que l’application Vokalpédia est capable d’analyser une image et d’en faire une description, une autre API est utilisée. Il s’agit de **Microsoft Azure Computer Vision**. Cette fois, l’url d’une image est envoyée à Microsoft Azure, et en sortie est récupérée une chaine de caractères. Cette chaine de caractères est ensuite transmise à Microsoft Azure Cognitive Services Speech pour obtenir le fichier .wav en audio.

J’ai choisi d’avoir recours à Microsoft Azure pour plusieurs raisons. Tout d’abord, j’avais la nécessité d’avoir une grande fluidité dans le workflow de l’application, ensuite j’ai pu étudier Microsoft Azure durant ma formation.

J’ai développé et entrainé des modèles « *custom*» pour *le Text-to-Speech* et le *Captioning Image*. Malheureusement, la qualité de prédiction et surtout le temps d’exécution pour générer le fichier .wav était beaucoup trop long pour une utilisation fluide de l’application. A titre de démonstration du travail accompli, j’ai choisi de proposer un modèle « *custom* » de *Captioning Image* pour le projet, mais j’ai décidé de n’y avoir recours qu’en utilisation secondaire en cas d’échec de Microsoft Azure, lequel doit rester le canal privilégié de fonctionnement de l’application.

A noter enfin que seulement trois images sont prises de manière aléatoire sur les pages Wikipédia demandées. Je requête l’intégralité des images mais je n’en envoie que trois prises aléatoirement à Microsoft Azure Computer Vision. Par souci d’économie financière (API payante) et de rapidité. Il faut quelques secondes pour faire le traitement auprès de Computer Vision, et ensuite le résultat de Computer Vision est envoyé à Microsoft Azure Cognitive Services Speech qui lui-même a son propre temps de traitement. Pour garder une certaine fluidité, seulement trois images sont donc gardées. Il s’agit d’un choix arbitraire aisément modifiable.

**Partie II : La mise en œuvre du projet**

**A) Gestion de projet**

Le projet a été conduit sous l’égide des méthodes Agiles, en particulier la méthode **Scrum**.

L’architecture des méthodes Agiles vise à un système itératif, incrémental et adaptatif, à rebours des méthodes traditionnelles. Un système itératif car il s’agit de cycles répétitifs, généralement assez courts (quelques jours ou semaines) visant à présenter au client une fonctionnalité terminée. Un système incrémental car le produit est réalisé petit à petit à partir d’ajouts de fonctionnalités terminées, et n’est au contraire pas réalisé d’une traite pour être présenté au client. Enfin, un système adaptatif car le but est d’associer pleinement le client en recueillant ses remarques, en s’améliorant perpétuellement à chaque itération, et en construisant le produit à l’aune desdites remarques.

La méthode Scrum implique la construction du projet autour de trois artefacts : le « Product Backlog », le « Sprint Backlog », et « l’incrément ». Par ailleurs, elle s’articule autour de quatre cérémonies : le « Sprint Planning », le « Daily Scrum », la « Sprint Review » (ou dit aussi « Démo »), et enfin la « Sprint Rétrospective ». Plusieurs outils opérationnels accompagnent ces artefacts et cérémonies.

Le projet a été mené seul, toutefois je me suis attaché autant que faire se peut, eu égard aux circonstances, à essayer de suivre ce cadre de travail. Le *Product Backlog* a été vu en supra dans la partie relative au besoin client.

Par le biais du site Notion.so, j’ai réalisé un tableau Kanban qui fait office de *Sprint Backlog*. En annexes, nous pouvons y retrouver une photographie (état d’avancement) au 26 novembre 2022, au 26 janvier 2023 ainsi qu’au 26 mars 2023 (deux mois d’intervalle).

Il ne s’agit pas du délai de Sprint Planning. En effet, je me suis plutôt référé au temps de ma formation, qui est du rythme de trois semaines entreprise pour une seule en école.

**B) Situation du livrable**

1 ) Le workflow de l’application

L’application Vokalpédia tente de répondre aux besoins utilisateur en offrant une solution relativement efficiente en termes d’interactions.

Dans cette version de l’application, l’utilisateur doit commencer par se connecter avec un compte utilisateur (*user*) et un mot de passe. Cette phase d’authentification est malheureusement nécessaire car la solution repose sur des appels API auprès de Microsoft Azure. Or, Microsoft Azure n’est pas une solution open source, le coût financier est certes raisonnable, mais néanmoins existant. Un contrôle de l’utilisation de l’application est partant nécessaire.

Si l’utilisateur n’a pas déjà de compte utilisateur, alors il est renvoyé vers la page web « signup ». Il s’agit d’une page d’inscription préétablie par Django. Tous les champs présents et les contrôles associés proviennent de Django, à l’exception de la date de naissance que j’ai décidé d’ajouter.

Une fois l’authentification opérée, l’utilisateur fait face à un bouton « Démarrer » sur la page web « home ».

Cette page est le cœur de l’application, tout ou presque s’y réalise.

En cliquant sur ledit bouton, l’utilisateur lance l’application. Un *Speech-to-Text* de Javascript (Web Speech API) se lance afin de questionner l’utilisateur.

L’utilisateur est donc invité à répondre selon les modalités attendues. Une aide est fournie à l’utilisateur en cas de première utilisation ou d’oubli de son fonctionnement.

Concrètement, l’utilisateur doit dire « recherche » suivi d’un thème désiré.

Une fois l’information fournie par l’utilisateur, elle est transmise par Javascript au serveur web côté Django.

Au niveau affichage, l’application comprend un logo, une phrase de présentation, le statut de l’intéressé (statut connecté). Lorsque l’utilisateur a prononcé sa demande, un calcul de précision est réalisé quant à la compréhension du thème par l’application. La demande de l’utilisateur est également affichée.

L’application dit vocalement la demande de l’utilisateur pour que ce dernier puisse s’assurer que le contenu demandé est bien celui compris par Vokalpédia.

La vue Django « Wikispeech » traite la chaine de caractères récupérée selon son contenu.

En effet, tout dépend si l’utilisateur a demandé « lecture », « aide », « sommaire », « section », « image », ou le nom d’un article en particulier en sus du thème, lequel doit systématiquement être renseigné. Ainsi :

* « Aide » comme son nom l’indique est une aide vocale expliquant les différentes commandes possibles.
* « Lecture » consiste à lire l’intégralité de la page demandée.
* « Sommaire » renvoie le sommaire dans son intégralité, grands titres comme sous-titres.
* « Section » renvoie seulement les grands titres du sommaire (donc sans les sous-parties).
* « Image » renvoie la description audio de trois images aléatoires de la page à titre illustratif.

La vue Django renvoie alors soit une url de fichier audio enregistrée en base de données, le fichier étant lui-même enregistré dans le répertoire « média » de l’application, soit une chaine de caractères transmise en JSON. Ce dernier cas ne se réalise que dans le cas du mode « lecture ».

En effet, le mode « lecture » adresse une chaine de caractères à Javascript. Ce dernier par le biais du Web Speech API et sa fonctionnalité *Text-to-Speech* convertit alors la chaine de caractères en audio. Il est lu directement à l’utilisateur. Par ailleurs, un prompt est ouvert pour afficher le contenu de la chaine de caractères sur la page web.

En revanche, et dans tous les autres cas (y compris requête « image »), la vue Django « Wikispeech » ne fait que renvoyer une url de fichier audio. Le fichier audio qui s’y trouve est alors lu par le lecteur audio du navigateur web. Je transmets une url plutôt que le fichier lui-même par efficience.

Les requêtes de l’utilisateur sont enregistrées en base de données afin ne pas requêter à chaque fois Wikipédia pour des demandes qui auraient déjà été faites. Cela fait un gain de temps à l’utilisation, on y gagne plusieurs secondes, et de surcroit cela évite de faire des appels à Microsoft Azure, lequel rappelons-le, est une solution payante.

Il est possible de connaitre les requêtes par utilisateur. Il est à noter que la suppression de l’utilisateur ne doit pas en cascade supprimer la ressource générée.

Si l’utilisateur veut mettre en pause l’application, il le peut en cliquer sur la touche « espace » du clavier. En la retapant, il pourra relancer la lecture audio.

De même, si l’utilisateur appuie sur « Alt », alors il relance la lecture audio depuis son début.

Une fois la lecture terminée, il convient de cliquer à nouveau sur le bouton pour relancer un cycle de l’application.

2) La base de données réalisée

La base de données repose donc sur SQLite3 et son fonctionnement est opérée par le biais de l’ORM. Je l’ai conçue selon la méthode des diagrammes de classes (UML).

La base comporte quatre tables : *User, Search, UserSearch, SpeechResult*.

L’idée derrière chaque table est la suivante :

* **User**: Utilisateur en français, elle est limpide à comprendre. J’ai besoin d’un utilisateur, je ne peux autoriser quiconque sans restriction à requêter l’application eu égard à l’utilisation d’un service cloud. Dans notre cas, j’ai utilisé ce que nativement Django fournit, mais je l’ai revanche étendue en lui adjoignant un champ de date de naissance.
* **Search** : Une recherche en français. L’utilisateur effectue des requêtes de recherche de page Wikipédia. Il y a trois champs : *text* (concrètement le nom du thème demandé), *type* (le type de retour en soit cela fait référence à une commande : article, help, navigation, image), et *date* pour horodater la recherche.
* **UserSearch** : Il s’agit d’une table intermédiaire servant de liaison entre *User* et *Search* puisqu’un utilisateur peut faire une ou plusieurs recherches, et une recherche peut concerner un à plusieurs utilisateurs (relation *many to many*). Table nécessaire pour préserver l’intégrité de la base.
* **SpeechResult** : Elle vise à l’enregistrement de la prédiction, le contenu final obtenu. Il y a une clef étrangère avec la table *Search* (relation *one to one*), et ensuite deux champs. Le champ *file\_name* traite du nom de la recherche, il s’agit du thème associé à une commande particulière (ex : section/sommaire/article désiré) avec un timestamp d’horodatage ce qui le rend forcément unique sauf cas d’extrême improbabilité. Et enfin, un champ *file\_path* qui vise à pointer le lieu d’enregistrement du fichier (concrètement dans le un répertoire nommé média).

En sus, un fichier managers.py existe et lié à la table User par le champ *objects*. Il s’agit de méthodes fournies par Django permettant de créer un utilisateur, le mettre éventuellement en *superuser* (super utilisateur ayant des droits d’administration).

Avec les modèles actuels, si nous supprimons un utilisateur (table *User*), les entrées associées dans la table *UserSearch* seront également supprimées. Cependant, les entrées dans la table *Search* ne seront pas supprimées automatiquement.

La raison en est que la relation entre *User* et *Search* est définie par la table *UserSearch*, qui a des clés étrangères (*ForeignKey*) vers *User* et *Search*. Lorsqu'un utilisateur est supprimé, les entrées correspondantes dans *UserSearch* sont également supprimées en raison de la propriété *on\_delete=models.CASCADE* pour la clé étrangère *user*. Cependant, cela n'affecte pas directement la table *Search*.

J’ai tout de même rajouter une méthode manuelle de suppression des recherches lorsque nous souhaitons supprimer un utilisateur. Il s’agit de la méthode delete dans la table *User*. Elle va d’abord récupérer les entrées dans *UserSearch* associées à l’utilisateur,ensuite pour chaque entrée elle va supprimer ladite entrée *UserSearch* puis la *Search* associée, et in fine à la fin l’utilisateur est lui-même supprimé.

3) Les tests unitaires réalisés

J’ai utilisé le framework de test **Pytest** pour réaliser les tests unitaires de l’application. C’est un outil spécialisé dans les tests de code en langage python.

Précisément, j’ai écrit des tests pour des éléments essentiels au bon fonctionnement de l’application. Il s’agit d’une classe en particulier, et les deux méthodes de Microsoft Azure employées.

La classe *Navigation\_sommaire\_wikipedia* est prépondérante dans l’application puisqu’elle permet de récupérer soit le sommaire intégral, soit seulement les grandes sections. Elle répond donc à la problématique de deux des six commandes possibles (rappel : lecture, sommaire, section, image, aide, et un article en particulier à préciser).

Les cinq tests réalisés sont validés par Pytest et le résultat visible en annexe.

4) Journalisation des logs/erreurs

Un fichier django.log (défini dans le fichier de settings pour les paramètres) récupère tous les logs d’activité, y compris les erreurs ce qui est utile pour le débogage. Il est possible d’envoyer des emails en cas d’erreur 500, ce dernier point reste inactif en l’état du projet. Voir annexe.

**Partie III : Le bilan du projet**

**A) Retours d’expérience**

**1 ) Retour d’expérience sur les bogues et tentatives de solution**

a) Les boutons « Rafraichir » et « Arrêter »

J’ai dû rajouter deux autres boutons, « Rafraichir » et « Arrêter ». Initialement ils n’existaient pas et « Démarrer » était l’unique bouton. Malheureusement, les phases de test m’ont fait comprendre qu’en l’état l’application pouvait connaitre des bugs. Rajouter ces boutons évitent de rencontrer des difficultés à stopper la lecture et pouvoir relancer « proprement » une nouvelle recherche. Ces boutons sont donc facultatifs dans la majorité des cas, mais salvateurs pour le moment en cas de bugs.

C’est en l’état un regret de ma part puisque je cherche à obtenir l’expérience utilisateur (UX) la plus simple et épurée possible.

b) La problématique des formats d’image

La description des images s’avère régulièrement problématique. En effet, Wikipédia utilise plusieurs formats d’image. Tous ne sont pas parfaitement compatibles avec Microsoft Azure Computer Vision, lequel préfère les JPEG et PNG. A cela se rajoute des limites de taille des images dans l’option la moins onéreuse de Microsoft Azure. J’ai tenté de contourner ces limites en essayant de ne retenir que les images en JPEG ou PNG et en tentant de réduire la taille des images.

Je rappelle que je requête l’ensemble des images de la page et n’en retient que trois aléatoirement. Sur ces trois, je tente de ne récupérer donc que des JPEG ou PNG. Toutefois, malgré tous ces traitements, il arrive tout de même souvent que le processus casse. Du coup, le programme ne va pas au bout de ses traitements et en cas d’un échec n’est pas capable de rebondir correctement. Il n’y a donc pas de lecture audio qui se fait.

c) Les demandes complexes

J’ai réussi après maintes difficultés à obtenir le requêtage des recherches en mots composés (exemple : New York, donc deux mots). Par contre, les algorithmes et le code que j’ai réalisés ne peuvent en l’état aller plus loin. C’est-à-dire que je fais en sorte de récupérer la première sortie proposée par une liste de l’API de Wikipédia. Du coup, pour une recherche de « New York », c’est « New York City » qui sera récupérée car la première mise en avant par l’API. Toutefois, vouloir consulter l’Etat de New York ou la série policière « New York Police Criminelle » sera impossible.

Le problème se posera aussi naturellement avec les homonymes. Exemple, il est impossible de demander les pages de « Troyes » (ville en Champagne), « Troie » (ville de l’Antiquité) ou « Trois » le chiffre. Vous n’aurez en l’état pas le choix de ce que vous souhaitez consulter.

d) Les demandes gérées par Wikipédia en anglais

En certains cas, il est possible que certaines demandes n’arrivent pas à être traitées par l’API de Wikipédia en français, j’ai donc codé l’application de telle sorte que ce soit la version anglaise qui prenne le relais. Malgré cette démarche, il est possible que des recherches n’aboutissent pas correctement par défaut de *matching*.

**2) Retour d’expérience sur la vie du projet**

Comme vu en supra, le développement a été réalisé dans le cadre des méthodes Agiles.

En dépit d’un cadre de travail fourni par la méthode, il n’en demeure pas moins que pour l’élaboration d’une telle application en étant seul, et avec mes compétences d’apprenant, les surprises furent nombreuses au cours du développement.

Tout d’abord, j’avais sous-estimé l’importance qu’aurait Javascript au sein du projet. Je ne maitrisais pas Javascript (hors syntaxe) avant de commencer. Il m’a fallu acquérir la logique inhérente à Javascript qui est très différente des autres langages en raison de son caractère asynchrone.

Au final, Javascript représente un aspect important du total du code écrit. Ce qui est somme toute logique puisqu’il permet l’interaction vocale avec l’utilisateur et dispose même de fonctionnalités d’intelligence artificielle qui lui sont propres grâce à la Web Speech API.

Ensuite, bien que je disposais de bases en Django, il m’a fallu les perfectionner. Par ailleurs, j’ai rencontré de nombreuses difficultés au niveau du *scrapping*. Là encore, si j’avais déjà étudié quelque peu le *scrapping*, il m’a fallu me renseigner et tester d’innombrables tentatives d’algorithmes pour obtenir le résultat final dans le requêtage de Wikipédia. Les choses furent simplifiées lorsque j’ai découvert les API de Wikipédia. Cependant, elles restent à manipuler intelligemment et les méthodes de ces API ne délivrent pas forcément toujours le résultat souhaité.

Enfin, le développement de modèles « custom » fut laborieux, surtout concernant le modèle de Captioning Image. La raison principale est d’ordre technique, faire tourner le modèle demande une machine puissante capable de tourner longtemps, chose que j’ai fini par acquérir. En effet, j’ai voulu utiliser le jeu de données MS COCO qui est particulièrement lourd (plus de 130 000 images). Malgré un grand nombre d’*epochs,* la qualité du modèle obtenu au cours de l’entrainement ne me satisfait pas. M’étant documenté de diverses sources et tutoriels sur ce sujet, et après plusieurs tentatives d’entrainement, aucun ne me semble fournir une qualité proche d’un modèle déjà entrainé par un service de cloud type Microsoft Azure.

A rebours, il existe des modèles « custom » pour le *Text-to-Speech* ou *Speech-to-Text*, mais ils sont le plus souvent très longs à l’exécution dès que nous dépassons une utilisation dépassant la chaine de caractères de taille modeste. Cela est antinomique avec la lecture d’une page Wikipédia.

Il m’a donc fallu abandonner mes projets initiaux d’utiliser que des modèles « custom » sur chaque sujet par des modèles entrainés par un cloud pour obtenir une application viable.

Naturellement, la conduite du projet au regard des méthodes Agiles en fut impactée. Difficile d’estimer correctement son temps lorsque l’on apprend Javascript en codant l’application. L’idée générale fut toutefois d’essayer de mener à bien une tâche dans le minimum de temps acceptable. La construction d’une application est comme un Légo, il est souvent indispensable de coder une autre brique afin de pouvoir en imbriquer deux ensemble. On ne peut développer tout le *back-end* ou tout le *front-end* d’une traite sans faire des aller/retour régulier entre les deux pour les enrichir et les faire s’interconnecter correctement.

**B) Améliorations envisageables**

1) La suppression des boutons « Rafraichir » et « Arrêter »

Il s’agit évidemment d’une réponse au bug vu en supra. L’idéal serait la suppression de ces boutons pour n’en avoir plus qu’un.

Il s’agit d’une révision du code probablement envisageable avec un peu plus de temps.

2) La création d’un véritable site et l’hébergement de l’application

En réalité, le travail a déjà été commencé. Je suis propriétaire du nom de domaine  « vokalpedia.com ». Je devais présenter le projet avec l’hébergement effectué. Seulement, j’ai rencontré des soucis techniques avec l’hébergeur. La version de Linux utilisée par l’hébergeur bloque certains paquets de mon environnement de travail, et ils sont indispensables puisqu’il s’agit de Microsoft Azure. A l’écriture de ces lignes, je n’ai pas encore eu l’opportunité de trouver la solution avec cet hébergeur (Namecheap).

3) Le développement d’une application mobile

Initialement, j’avais à l’esprit de développer une application mobile pour Vokalpédia. Le projet se destine aux malvoyants et je souhaitais une utilisation 100% vocal. Le recours à une application mobile était donc idéal. Seulement le temps étant compté, je n’ai pu développer cette application.

Surtout, j’ai découvert après recherche une incompatibilité entre Web Speech API de Javascript et la technologie Apache Cordova alors que j’escomptais l’utiliser au niveau de la webview.

Il aurait donc fallu revoir le projet en utilisant une autre technologie, probablement React Native. Seulement, je ne connais pas bien la bibliothèque/framework React, cela aurait nécessité une courbe d’apprentissage importante. Et il faudrait vérifier avec certitude que Web Speech API fonctionne avec React.

Une autre possibilité serait d’abandonner complètement le recours à Web Speech API en Javascript, et passer intégralement l’application sous l’égide de Microsoft Azure. Il est possible de faire du *Text-to-Speech* et du *Speech-to-Text* avec Microsoft Azure en Javascript. Ce serait la meilleure solution si nous acceptons d’avoir une API payante. Il restera là encore à vérifier la pleine compatibilité de toutes les bibliothèques et API.

Enfin, se défaire de Web Speech API permettrait aussi d’utiliser d’autres navigateurs web que Google Chrome. En effet, tous les navigateurs ne supportent pas en l’état l’application Vokalpédia. Il n’est par exemple pas conseillé d’essayer de l’utiliser sur Brave (pourtant basé sur Chromium), ni sur Firefox par exemple.

4) L’utilisation des options d’accessibilité

Certains systèmes d’exploitation et navigateurs web offrent des options d’accessibilité pour les personnes handicapées.

Le navigateur Chrome sur lequel a été développé l’application en dispose. La fondation Mozilla avec Firefox dispose aussi d’outils d’accessibilité.

J’avais fait des tests avec Microsoft Windows 10, toutefois je n’ai pas trouvé mes tentatives fructueuses.

Il serait donc intéressant d’essayer de modifier l’application Vokalpédia en effectuant des recherches poussées sur le sujet de l’accessibilité pour rendre l’application la plus inclusive possible.

5) Utilisation de technologies alternatives

a) Docker et un service cloud hébergeur

L’utilisation de Docker permettrait de tenir compte de la problématique de scalabilité. Il faudrait idéalement l’associer à l’utilisation d’un service cloud qui hébergerait la solution et fournirait les ressources hardware nécessaires pour s’adapter à la demande des utilisateurs. Plus il y aurait d’utilisateurs, plus il y aurait de requêtages et donc nécessité d’accroitre les capacités hardware pour y faire face.

b) Passage à Node.js

Le passage à un *back-end* tournant sur Node.js permettrait aussi d’améliorer la situation en cas de forte sollicitation du serveur web. Node.js est basé sur Javascript, gère l’asynchrone, et est extrêmement efficace pour gérer de fortes surcharges, à la différence de Python/Django qui ne sont pas réputés pour être rapides en exécution.