Université de Sherbrooke

Approches orientées objets
Projet de session
Rapport

Calliope

Élèves :
Gabin Granjon (*grag3893*)
Victor Levrel (*levv1279*)
Yann Dionisio (*dioy6143*)
Adrien Larousse (*lara6256*)

Enseignant :
Hubert Kenfack Ngankam

15 avril 2025

Table des matières

1 Introduction **2**

2 Architecture **3**
2.1 Architecture générale . . . 3
2.1.1 Docker . . . 3
2.1.2 Environnement . . . 4
2.2 Frontend . . . 5
2.2.1 Modèle de classe . . . 6
2.3 Backend . . . 6
2.4 API IA . . . 8
2.5 Service de base de données . . . 9

3 Les Design Pattern **10**
3.1 Client . . . 10
3.1.1 Command . . . 10
3.1.2 Singleton . . . 11
3.1.3 Decorateur . . . 11
3.1.4 Adaptateur . . . 12
3.1.5 Observer . . . 13
3.2 API IA . . . 14
3.2.1 Singleton . . . 14
3.2.2 Strategy . . . 15
3.2.3 Decorator . . . 16
3.2.4 Facade . . . 17

4 API **19**
4.1 POST `/transcribe` . . . 19
4.2 POST `/summarize` . . . 21

5 Guide de déploiement **23**

6 Tests **26**

7 Conclusion **27**

Chapitre 1

Introduction

Ce projet a été réalisé dans le cadre du cours IFT785 - Approches orientés objets du semestre d'hiver à l'université de Sherbrooke.

Pour ce projet nous avons choisi de créer un site web pour servir d'assistant de prise de notes automatique. Cet assistant intelligent capture des discussions (réunions, conférences, cours) sous forme de texte structuré et génère automatiquement des résumés, points clés et actions à prendre. Il vise à faciliter la prise de notes et l'organisation des informations pour les étudiants et les professionnels.

Les fonctionnalités principales sont :

- Transcription automatique de l'audio en texte.
- Gestionnaire de note : création de note, édition de texte...
- Résumé automatique du contenu transcrit.

Chapitre 2

Architecture

Ce chapitre présente l'architecture globale de notre projet.

2.1 Architecture générale

Le projet est basé sur une application multi-service. La figure 2.1 représente l'architecture globale de notre projet.

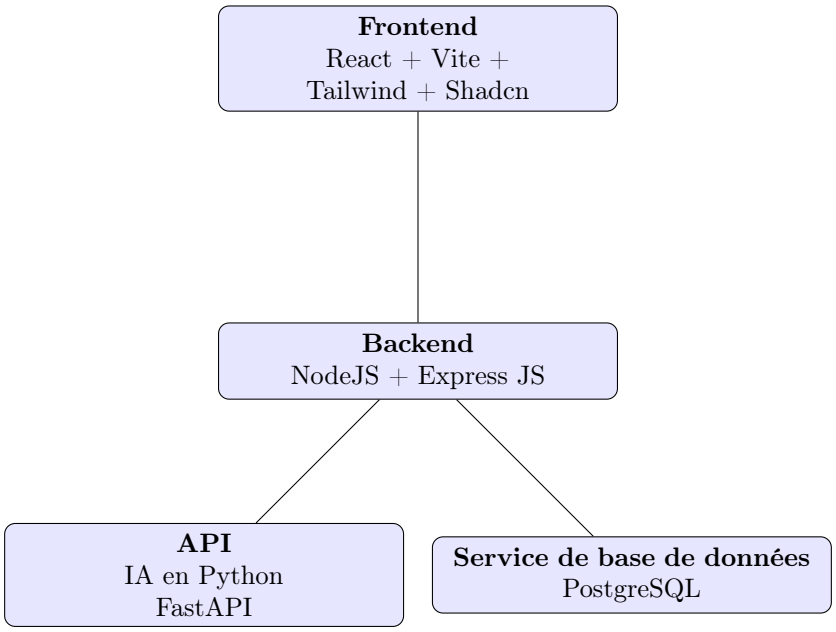


FIGURE 2.1 – Architecture du Système

Il y a 4 blocs qui vont permettre le bon fonctionnement du projet, la séparation des responsabilités, la limitation des accès. Nous allons donc voir en détail le contenu de chaque bloc.

2.1.1 Docker

Dans ce projet, nous utilisons la *conteneurisation* afin d'isoler chaque service, de faciliter la gestion des environnements de développement, de test et de production, et de

simplifier le processus de développement. Pour cela, nous avons recours à Docker pour créer les services, et à Docker Compose pour orchestrer leur exécution.

Les différents services – qui communiquent via des API – sont :

- Le service client, construit avec le bundler `Vite`, utilise le framework `React`.
- Le backend de l'application est un service `NodeJS` basé sur la librairie `ExpressJS`.
- Le service d'intelligence artificielle, chargé de la transcription des fichiers audio, est un service `Python` utilisant `FastAPI` pour l'exposer sous forme d'API.
- Le service de base de données repose sur le système de gestion `PostgreSQL`, avec un volume partagé pour garantir la persistance des données.

Pour gérer la communication et l'organisation de ces conteneurs, nous utilisons Docker Compose, qui permet de définir et de configurer l'ensemble des services dans un fichier unique (docker-compose.yml).Cela facilite le développement, et un futur déploiement et une mise en production.

2.1.2 Environnement

Pour simplifier le développement, nous avons mis en place 3 environnements différents : développement, test et production. Pour cela, nous utilisons différents fichiers Docker comme `Dockerfile.dev` et `docker-compose.dev.yaml`.

2.2 Frontend

```
calliope-client/
├── Dockerfile
├── package-lock.json
├── package.json
├── public/
│   └── Images/
├── src/
│   ├── App.tsx
│   ├── components/ ..... Composant de l'interface graphique
│   │   └── Composants de l'UI
│   ├── global.css
│   ├── hooks/
│   │   └── use-mobile.ts
│   ├── layouts/
│   │   └── AuthentificationLayout.tsx
│   ├── lib/
│   │   └── utils.ts/
│   ├── main.tsx ..... Point d'entrée du client
│   ├── models/ ..... Modèle de classe du client
│   │   ├── NavItem.ts
│   │   ├── User.ts
│   │   └── UserAdapter.ts
│   ├── pages/ ..... Plus haut niveau d'abstraction pour l'affichage des pages
│   │   ├── LandingPage.tsx
│   │   ├── LibraryPage.tsx
│   │   │   └── Page.tsx
│   ├── services/ ..... Communication avec le backend
│   │   ├── AudioService.ts
│   │   └── UserService.ts
│   └── vite-env.d.ts
├── tests/ ..... Tests du client
├── tsconfig.app.json
├── tsconfig.json
├── tsconfig.node.json
└── vite.config.ts
```

FIGURE 2.2 – Structure du client

La figure 2.2 représente la structure de fichiers du frontend. à la racine du dossier calliope, nous retrouvons tous les fichiers de configuration pour les dockers, les dépendances à installer.

Ensuite il y a deux dossiers : src pour le code et public pour les ressources publics – comme certaines images – , accessible depuis le client.

Le src va contenir différents dossiers :

- components : Contient tous les composants visuels de la **librairie graphique shadnc**
- layouts : Layouts pour les différentes pages : authentification etc...
- pages : fichier contenant les .tsx des différentes pages du front. A savoir login, register, home....
- services : dossier comprenant les fichiers qui se chargent de la récupération des données et de toute autre information remplie par l'utilisateur

2.2.1 Modèle de classe

2.3 Backend

La figure 2.3 représente la structure de fichiers du backend.

```
calliope-backend
    config
        ai.config.js
        jwt.config.js
        pg.config.js
        swagger.config.js
    controllers
        audio.controller.js
        auth.controller.js
    coverage
        clover.xml
        coverage-final.json
        lcov-report
            base.css
            block-navigation.js
            controllers
                audio.controller.js.html
                index.html
            favicon.png
            index.html
            middleware
                index.html
                multerErrorHandler.middleware.js.html
                validateAudio.middleware.js.html
            prettify.css
            prettify.js
            routes
                audio.routes.js.html
                index.html
            sort-arrow-sprite.png
            sorter.js
        lcov.info
    eslint.config.mjs
    index.js
    middleware
        jwtMiddleware.js
        multerErrorHandler.middleware.js
        user.middleware.js
        validateAudio.middleware.js
    models
        user.model.js
    routes
        audio.routes.js
        auth.routes.js
```

FIGURE 2.3 – Structure du backend

2.4 API IA

La figure 2.4 représente la structure de fichiers de l'API.

```
calliope-ia:
    Dockerfile
    api
        routes.py
    app.py
    pytest.ini
    requirement.txt
    service
        api_handler.py
        audio_preprocessing
            audio_file.py
            audio_file_decorator.py
            basic_audio_file.py
            mono_conversion_decorator.py
            resampling_decorator.py
        audio_segmentation
            audio_processor.py
            fixed_duration_segment.py
            segmentation_strategy.py
            silence_based_segmentation.py
        data_manager.py
        model_manager.py
        preprocess_audio.py
    setup.py
```

FIGURE 2.4 – Structure de l'API

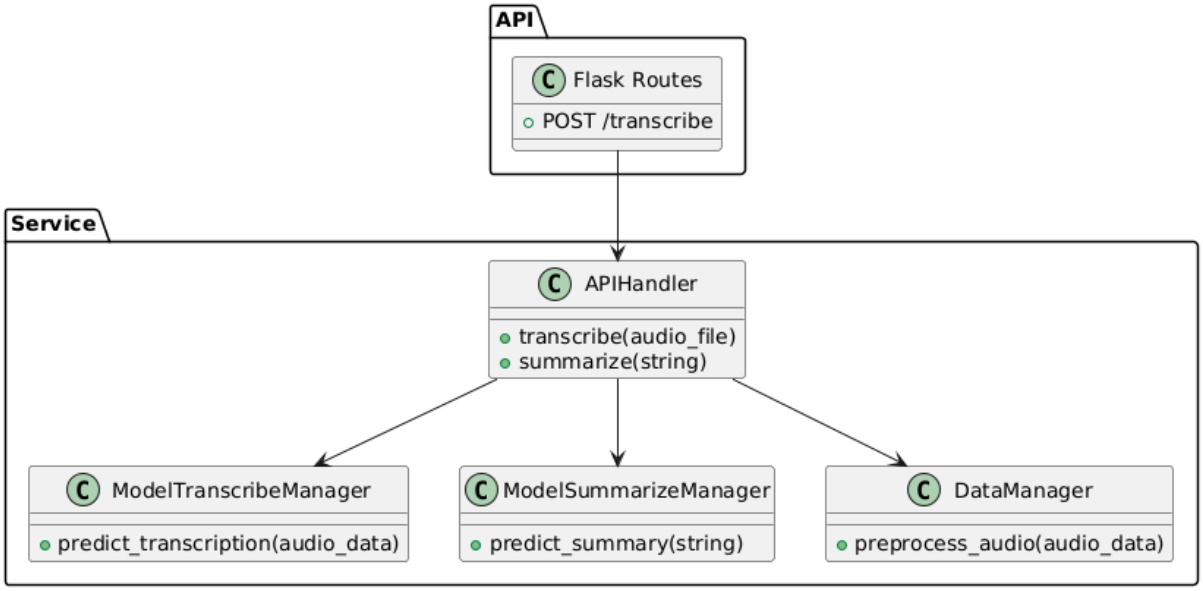


FIGURE 2.5 – Fonctionnement de l'API

2.5 Service de base de données

La figure 2.6 représente la structure de fichiers de la base de données.

```
calliope-db:
    .env
    .env.sample
    init-scripts
        functions
            utilisateur
                create_user.sql
                get_user.sql
                get_user_public_id.sql
                login_user.sql
                signin_user.sql
                verify_user.sql
        init.sh
        schemas
            create_tables.sql
```

FIGURE 2.6 – Structure de la db

Chapitre 3

Les Design Pattern

Ce chapitre décrit les différents design pattern que l'on a utilisé.

3.1 Client

3.1.1 Command

Le premier patron de conception que nous avons implémenté côté client est le Command Pattern. Son principal intérêt est de définir une interface commune permettant d'exécuter des actions de manière uniforme à travers l'application. En particulier, ce patron nous permet de centraliser la logique liée à la création de ressources et à la mise à jour des composants.

Précisément, nous implémentons une fonction `execute()` asynchrone, car nos commandes orchestrent des appels asynchrones vers le service *backend*.

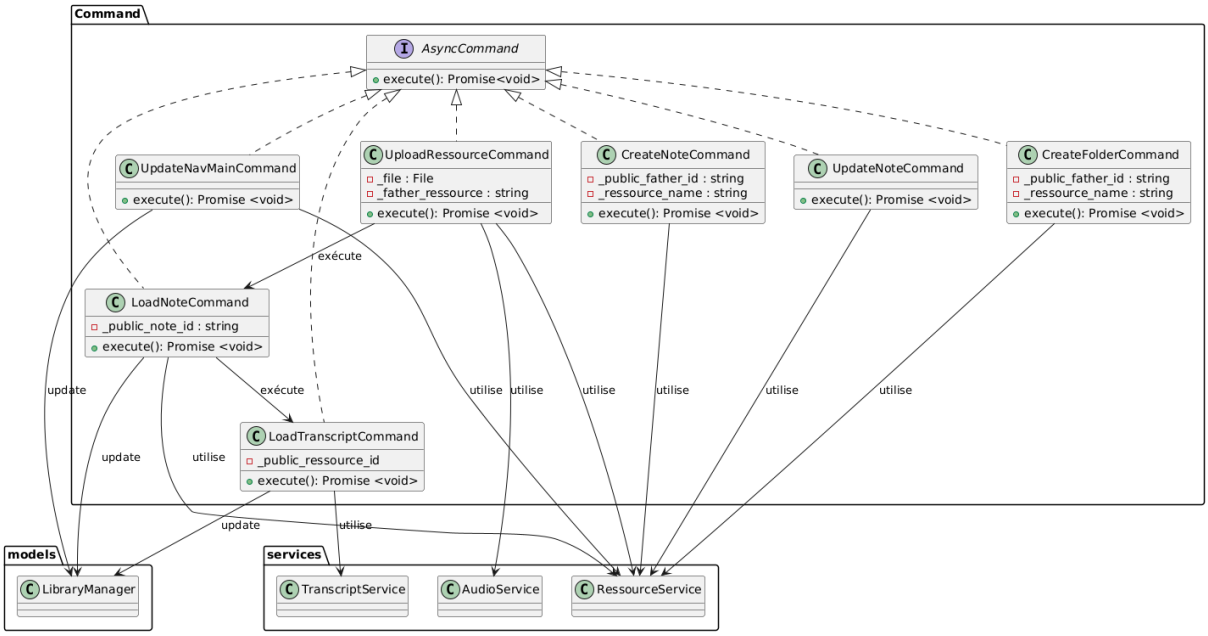


FIGURE 3.1 – Diagramme UML représentant l'utilisation du patron de conception `Command`

3.1.2 Singleton

Pour gérer les données afficher sur notre client, nous utilisons le patron de conception Singleton. Ce patron de conception permet de gérer une seule instance de `LibraryManager`. La classe `LibraryManager` nous permet d'accéder n'importe ou dans l'application à la référence de l'éditeur Markdown. En plus de cela, elle permet de centraliser une instance de `Library` qui contient la note courante, le transcript courant et l'arborescence courant

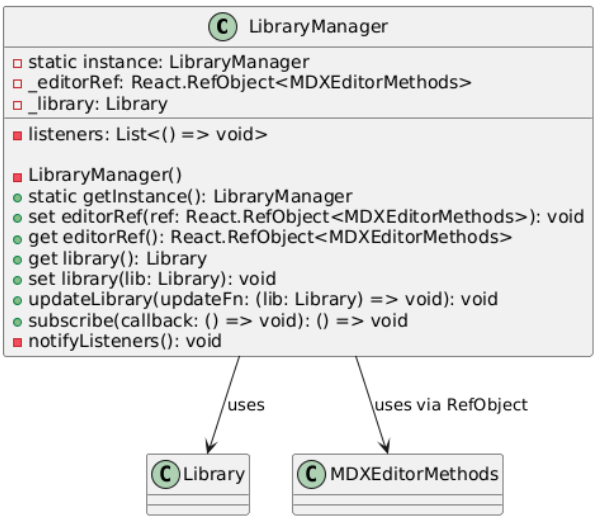


FIGURE 3.2 – Diagramme UML représentant l'utilisation du patron de conception `Singleton`

3.1.3 Decorateur

Pour informer l'utilisateur de l'état des commandes exécutées (création de note, téléversement d'un fichier audio), nous avons implémenté le patron de conception *Décorateur*, appliqué à l'interface `AsyncCommand` présentée précédemment.

Les deux décorateurs que nous avons définis permettent d'afficher des *toasters*, c'est-à-dire de petites modales visibles côté client, comme illustré à la Figure 3.3.

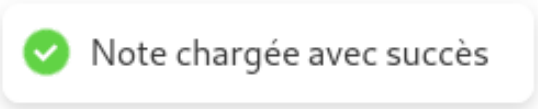


FIGURE 3.3 – Toaster issue du décorateur `ToastSuccessErrorCommandDecorator`

Nos décorateurs permettent d'utiliser deux types de *toasters*, en fonction du contexte :

- `ToastSuccessErrorCommandDecorator` : utilisé pour les commandes asynchrones de courte durée. Ce décorateur affiche un message en cas de succès ou d'échec de la commande.
- `ToastPromiseCommandDecorator` : utilisé pour les commandes asynchrones longues, comme la transcription. Il permet d'afficher un *toast* indiquant que la commande

est en cours d'exécution, puis, une fois l'opération terminée, un message de succès ou d'erreur est affiché.

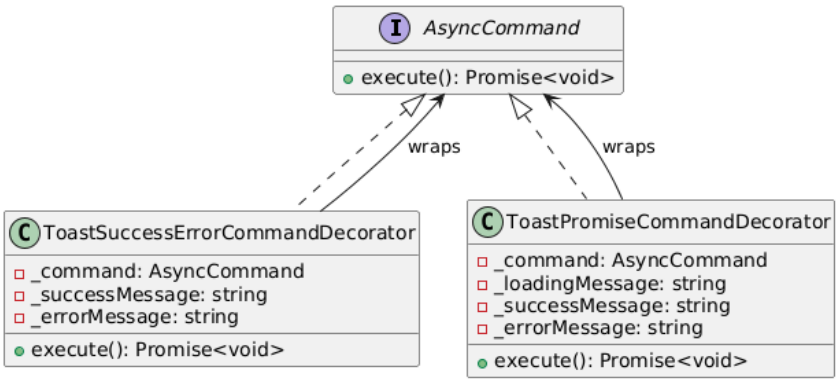


FIGURE 3.4 – Diagramme UML représentant l'utilisation du patron de conception `Decorateur`

3.1.4 Adaptateur

Dans notre cas, l'utilisation du patron de conception *Adaptateur* nous a permis de convertir facilement les réponses des requêtes en objets manipulables dans l'application. Par exemple, la classe `UserAdapter` permet de transformer un objet `JSON` reçu du backend en une instance de la classe `User`. De même, la classe `NavItemAdapter` convertit un objet JSON, tel qu'illustré à le Listing 3.1, en un objet `NavItem`, que l'on retrouve à le Listing 3.2.

Listing 3.1 – Exemple de réponse JSON retourné par `RessourceService.getArborescence`

```
{
    root: {
        public_root_ressource: "root_123",
        root_name: "Root",
    },
    arborescence: [
    {
        child_ressource_id: "folder_1",
        child_ressource_name: "Folder 1",
        child_ressource_nature: "dossier",
        parent_ressource_id: "root_123",
        parent_ressource_name: "Root",
    },
    {
        child_ressource_id: "note_1",
        child_ressource_name: "Note 1",
        child_ressource_nature: "note",
        parent_ressource_id: "root_123",
        parent_ressource_name: "Root",
    },
    ],
```

```
}
```

Listing 3.2 – Objet `NavItem` construit à partir du JSON

```
const navItem = new NavItem("Root", "dossier", "root_123", [
    new NavItem("Folder␣1", "dossier", "folder_1", []),
    new NavItem("Note␣1", "note", "note", [])
]);
```

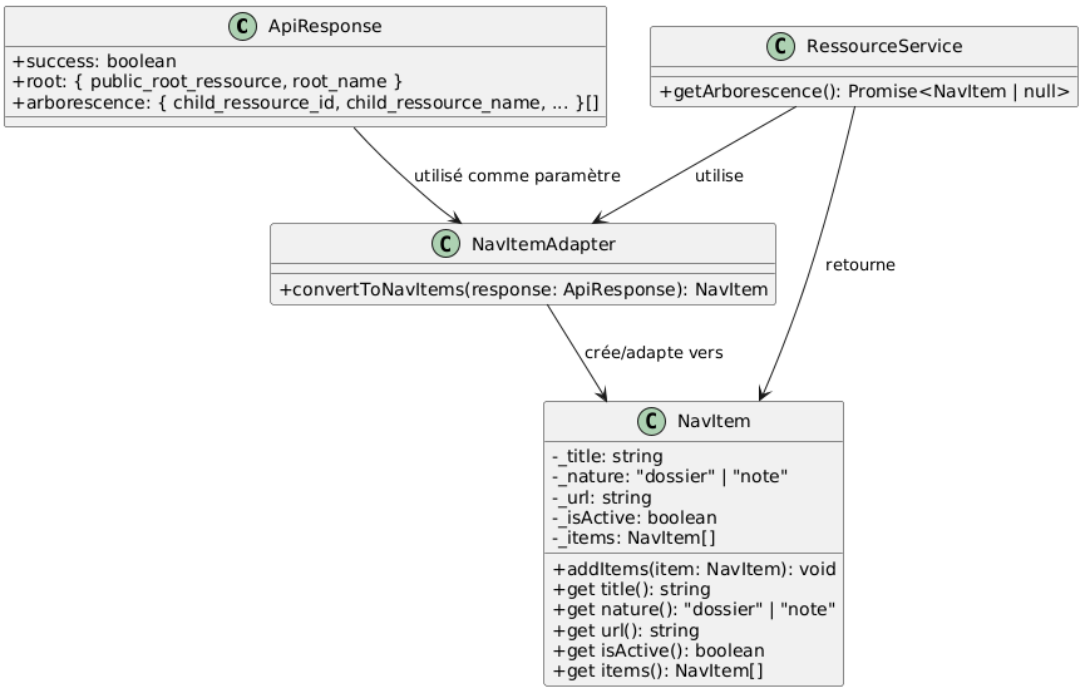


FIGURE 3.5 – Diagramme UML représentant l'utilisation du patron de conception `Adapter` pour `NavMain`

3.1.5 Observer

Le patron de conception *Observer* est utilisé dans notre client pour assurer la mise à jour automatique de l'interface utilisateur en réponse aux modifications de l'état de l'application. Il permet à différents composants de s'abonner à un objet central, le `LibraryManager`, qui agit en tant que *Subject*. Lorsqu'une modification est apportée à la bibliothèque de composants (par exemple, après une action de l'utilisateur ou une mise à jour des données), le `LibraryManager` notifie automatiquement tous les observateurs enregistrés, déclenchant ainsi un rafraîchissement de l'affichage.

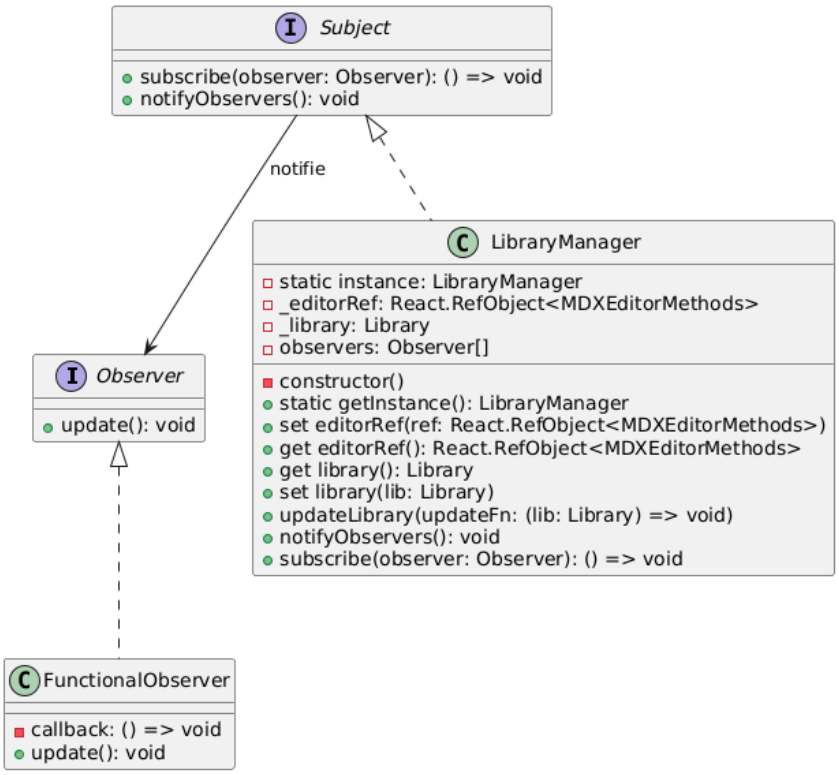


FIGURE 3.6 – Diagramme UML représentant l'utilisation du patron de conception `Observer` pour la mise à jour de l'affichage dans le client

3.2 API IA

3.2.1 Singleton

Nous utilisons le patron de conception singleton pour la création du modèle de transcription. Le modèle étant lourd (utilisation de whisper), cela permet de ne le charger qu'une seule fois, engendrant ainsi un gain de temps considérable lorsque l'on veut transcrire plusieurs fichiers audio à la suite.

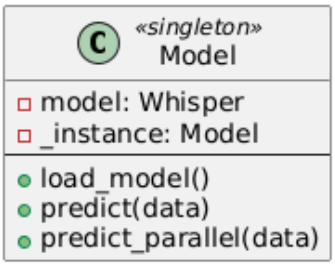


FIGURE 3.7 – Diagramme UML représentant l'implémentation du patron de conception `Singleton` pour la création de Model

3.2.2 Strategy

Le design pattern Strategy a été utilisé dans notre projet dans l'API. Il permet de créer une famille d'algorithme interchangeable en fonction du contexte souhaité.

Dans notre projet nous nous en sommes servi pour résoudre le problème de comment segmenter l'audio reçu de l'utilisateur. En effet, Il existe de nombreuses manières de segmenter un audio en portion pour le retranscrire. En fonction du contexte, ces méthodes doivent être changées.

Les méthodes implémentées :

- **Segmentation à durée fixe** : la segmentation la plus simple, on découpe l'audio en segments d'une durée fixe indépendemment du son ou du fichier audio. Cela permet une transcription rapide et "en direct" portion par portion. Cependant cela peut poser problème si l'audio est coupé au milieu d'un mot. La transcription peut alors être impactée.
- **Segmentation basée dur les silences** : la segmentation est effectuée lorsque l'audio contient un silence d'une durée pré-définie. L'algorithme supprime alors le silence et sépare l'audio en deux portions. Cela est particulièrement utile lors de la prise de parole de personnes différentes qui sont bien définies. L'environnement est calme est primordiale pour que cette méthode soit efficace.

La figure 3.8 représente le diagramme UML du design pattern utilisé dans notre cas.

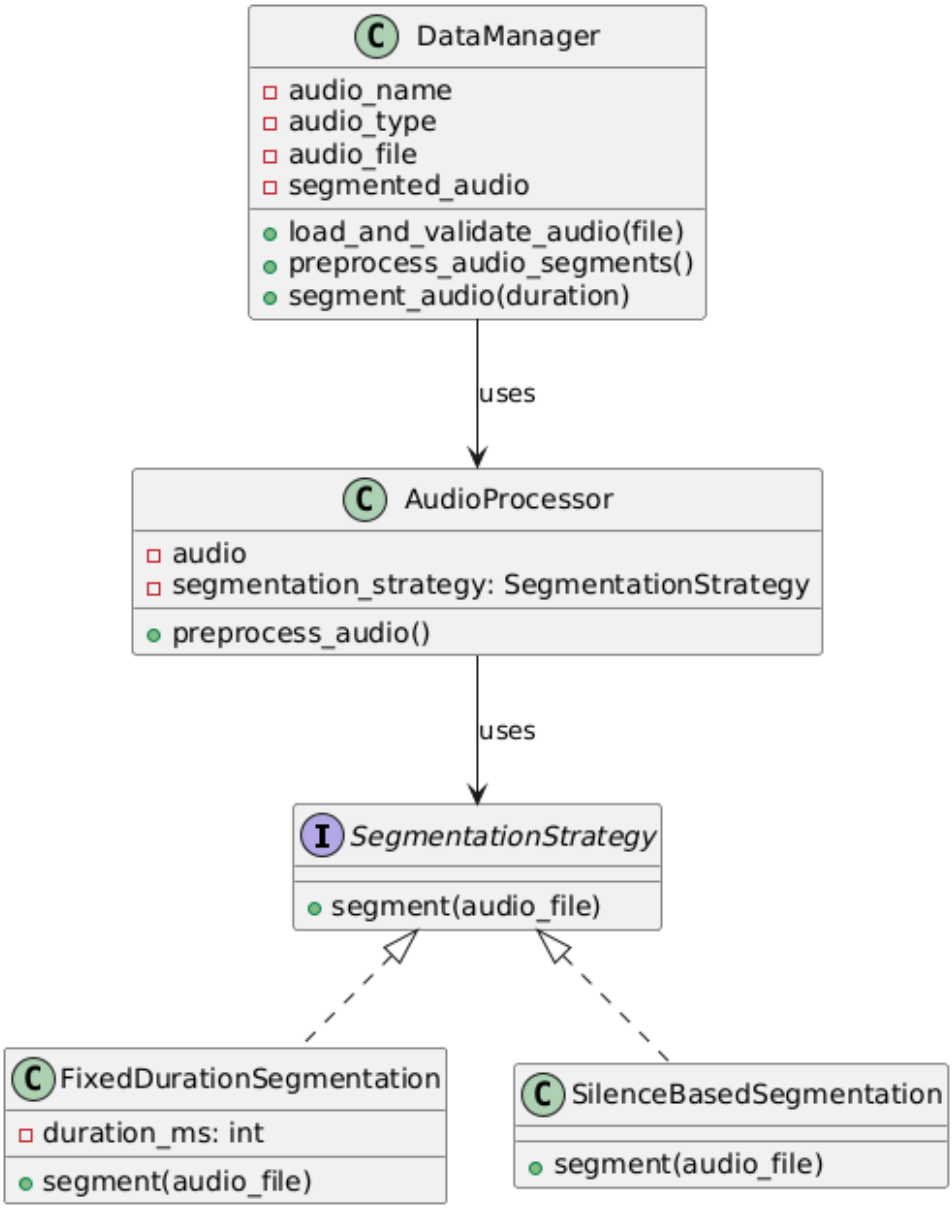


FIGURE 3.8 – Diagramme UML représentant l'implémentation du patron de conception Strategy

3.2.3 Decorator

Le design pattern Decorator a été utilisé dans notre projet dans l'API. C'est un design pattern structurel qui nous permet d'ajouter des comportements à des objets en ajoutant des couches au fur et à mesure.

Le pattern décorateur se prêtait particulièrement bien à notre problème pour le pré traitement des signaux audio. En effet, en ajoutant des couches de pré-traitement on peut pré traiter le signal donné en entrer de manière plus exacte. Par exemple un signal audio en stéréo devra être changé en mono. Mais un signal déjà en mono n'aura pas besoin d'être modifié.

Les différents décorateurs sont :

- **MonoConversionDecorator** : Convertir l'audio de stéréo à mono.

- **ResamplingDecorator** : Changer la fréquence d'échantillonage à 16kHz.

La figure 3.9 représente le diagramme UML du Design Pattern Decorateur utilisé dans notre cas.

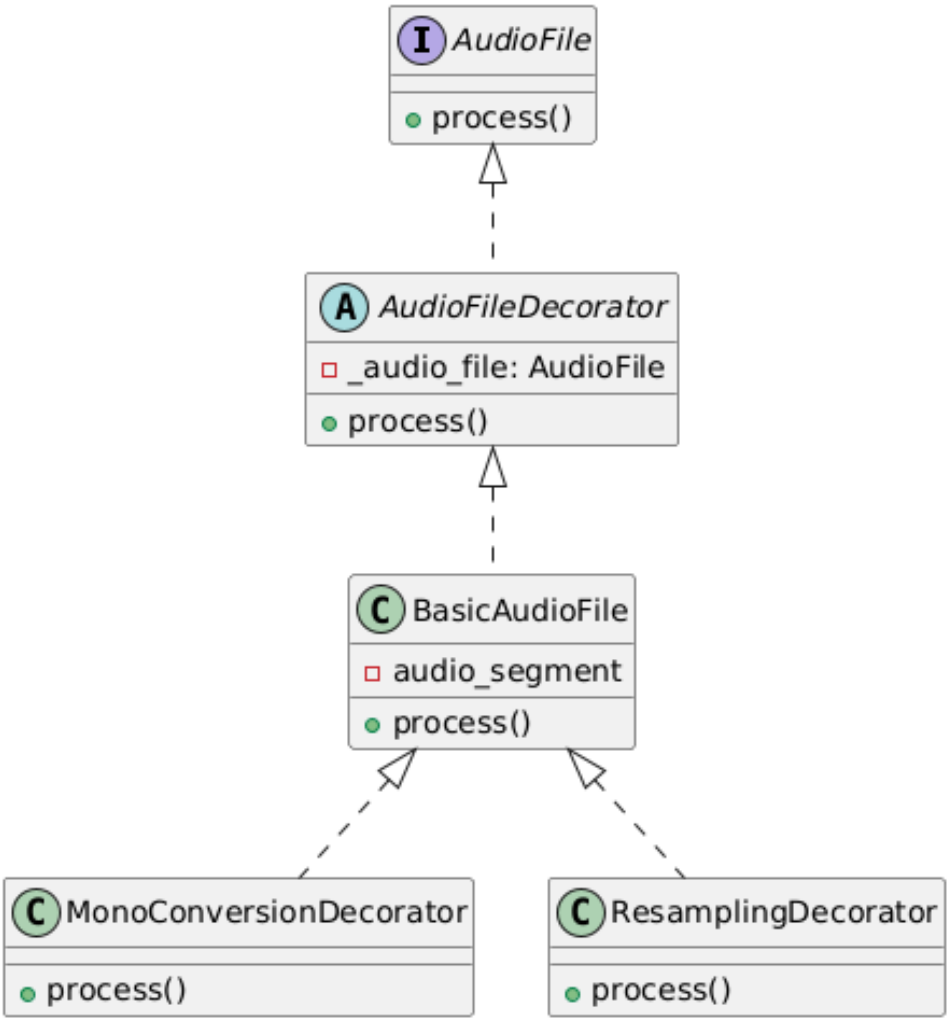


FIGURE 3.9 – Diagramme UML représentant l'implémentation du patron de conception Decorator

3.2.4 Facade

Pour la transcription d'un fichier, plusieurs étapes sont nécessaires auparavant :

- Le chargement et la validation de l'audio (format, taille, etc.)
- Le prétraitement de l'audio (resampling, segmentation, etc.)
- L'appel du modèle via le ModelManager pour prédire la transcription

Nous utilisons alors le patron de conception Facade pour centraliser toutes ces étapes au sein d'une même classe, APIHandler, afin de simplifier l'interaction avec l'API. La figure 3.10 illustre l'implémentation du patron de conception Facade dans notre cas.

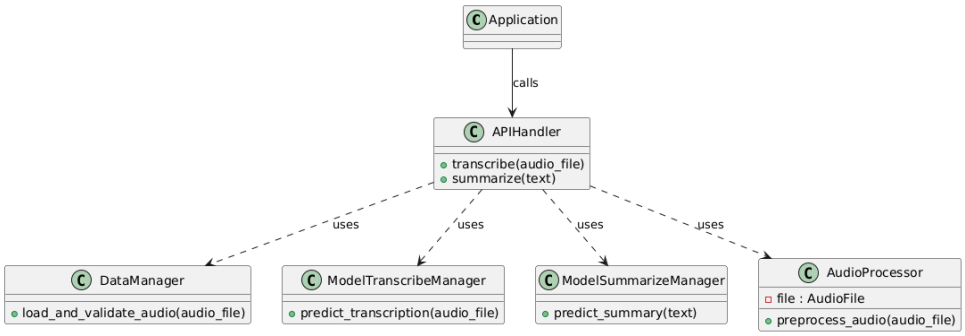


FIGURE 3.10 – Diagramme UML représentant l'implémentation du patron de conception Facade

Chapitre 4

API

Aperçu du module

Ce module définit les routes FastAPI pour notre service de transcription et de résumé de texte. Il est intégré à l'application principale FastAPI pour fournir les fonctionnalités suivantes :

- Transcription de fichiers audio (`POST /transcribe`)
- Résumé automatique de textes en français (`POST /summarize`)

Dépendances

- `FastAPI`
- `APIHandler` depuis `service.api_handler`

4.1 POST `/transcribe`

Description

Transcrit un fichier audio fourni par l'utilisateur. Le fichier doit être envoyé via un formulaire multipart.

Méthode

HTTP Method : `POST`

URL : `/transcribe`

Type de contenu attendu : `multipart/form-data`

Paramètres

`file` *(requis)* : Fichier audio à transcrire (ex. : `.mp3`, `.wav`).

Réponses possibles

`200 OK` Transcription réussie. Retourne un JSON contenant le texte transcrit.

`400 Bad Request` Aucun fichier audio fourni.

`415 Unsupported Media Type` Format de fichier non pris en charge ou erreur lors de la transcription.

Exemples de réponses

Succès (200 OK)

```
{
  "transcript": "Bonjour, ceci est un exemple de transcription."
}
```

Erreur (400 Bad Request)

```
{
  "detail": "Audio file is required"
}
```

Erreur (415 Unsupported Media Type)

```
{
  "detail": "Unsupported file type"
}
```

4.2 POST `/summarize`

Description

Résumé automatique d'un texte écrit en français. Le texte est envoyé dans le corps de la requête sous forme de JSON.

Méthode

HTTP Method : `POST`

URL : `/summarize`

Type de contenu attendu : `application/json`

Corps de la requête

`text` *(requis)* : Le texte brut à résumer (champ JSON).

Réponses possibles

`200 OK` Résumé généré avec succès.

`400 Bad Request` Texte vide ou manquant.

`415 Unsupported Media Type` Échec de la génération du résumé.

Exemples de requêtes et réponses

Requête (exemple JSON)

```
{
  "text": "L'intelligence artificielle est un domaine en pleine expansion
      qui..."
}
```

Réponse (200 OK)

```
{
  "summary": "L'intelligence artificielle est en forte croissance."
}
```

Erreur (400 Bad Request)

```
{
  "detail": "Text is required"
}
```

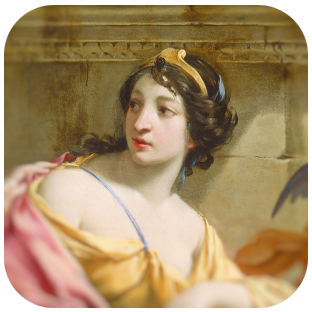
Erreur (415 Unsupported Media Type)

```
{
  "detail": "Summarization failed"
}
```

Chapitre 5

Guide de déploiement

Retrouvez le guide de déploiement également dans notre README.md sur github



Calliope

Application de transcription de note audio

À propos du projet

Contexte

Le projet Calliope a été réalisé dans le cadre du cours **IFT785 - Approches orientées objet**. Le présent répertoire contient le fruit de notre travail de session. Ce projet avait pour objectif de renforcer nos compétences en programmation orientée objet, en utilisation de patrons de conception ainsi qu'en mise en place de tests.

Construit avec

Frontend	Backend	API	Base de données	Outils
React	Node.js	FastAPI	PostgreSQL	Docker
Vite	JavaScript	Python		ESLint
TypeScript				Pylint
				Prettier

Utilisation

Pré-requis

Ce projet repose entièrement sur **Docker**, **Docker Compose** ainsi que sur des fichiers d'environnement `.env` pour fonctionner correctement.

Docker et Docker Compose

L'utilisation de **Docker** et **Docker Compose** est **obligatoire**. Le projet ne peut pas être lancé sans ces outils, aucune installation manuelle des dépendances n'est prévue.

Assure-toi d'avoir installé :

- Docker (version 20.x ou supérieure recommandée)
- Docker Compose (intégré à Docker Desktop ou installé séparément)

Vérification :

```
docker --version
docker compose version
```

Fichiers `.env`

Avant de lancer le projet, assure-toi que tous les fichiers d'environnement sont présents.
Fichiers à créer :

- `.env` basé sur `.env.sample` (calliope-db)
- `.env.dev` basé sur `.env.dev.sample` (calliope-backend)
- `.env.prod` basé sur `.env.prod.sample` (calliope-backend)
- `.env.test` basé sur `.env.test.sample` (calliope-backend)

Lancement du projet

Commande pour lancer en mode production :

```
make lint-frontend
make lint-backend
make lint-api
```

Développement, tests et couverture

Mode développement

```
make docker-up-dev
```

Exécution des tests

Pour tous les tests :

```
make tests
```

Tests ciblés :

```
make tests-frontend
make tests-backend
make tests-api
```

Couverture des tests

```
make coverage
```

Le rapport sera disponible dans le répertoire `coverage`.

Analyse statique (Linters)

Analyse automatique :

```
make lint
```

Commandes spécifiques :

```
make lint-frontend
make lint-backend
make lint-api
```

Configuration et accès local

Accès aux services :

- Frontend : http://localhost:5173
- Backend : http://localhost:5000
- API : http://localhost:8000
- Base de données : `localhost:5432`

Contributeurs

- Gabin Granjon
- Victor Levrel
- Yann Dionisio
- Adrien Larousse

Chapitre 6

Tests

Rapport de test avec métriques de couverture

Chapitre 7

Conclusion

Ce projet nous a permis de mettre en œuvre une application complète, combinant des approches orientées objets, des concepts avancés de design logiciel, ainsi qu'une intégration fluide entre différents services. Grâce à une architecture modulaire reposant sur Docker, nous avons pu développer, tester et déployer de manière structurée une solution permettant la transcription automatique de fichiers audio et le résumé automatique de textes.

L'utilisation de design patterns, tels que Singleton, Facade, Strategy, Decorator, etc. nous a permis de concevoir un code à la fois robuste, maintenable et évolutif, respectant les principes SOLID. En particulier, l'implémentation du pattern Facade au sein de l'API IA a simplifié considérablement l'utilisation des composants internes, tout en masquant leur complexité.

En somme, Calliope constitue une base solide pour une application d'assistance intelligente à la prise de notes, avec un fort potentiel d'évolution. Des pistes d'amélioration sont envisageables, telles que l'ajout d'un système de détection des intervenants dans l'audio, ou encore l'amélioration de la génération de résumés, en implémentant un formatage plus attrayant en markdown.