

Kévin DESPOULAINS

Gaël GENDRON

Corentin GUILLOUX

Valentin FOUCHER

Enzo CRANCE

Charlotte RICHARD

Laure DU MESNILDOT

Timothée NEITTHOFFER

Elèves ingénieurs de l'INSA Rennes

Année universitaire 2018/2019

Rapport de conception

Projet 4INFO

Logiciel de génération de données d'apprentissage pour la reconnaissance d'écriture manuscrite

Encadrants

Bertrand COÜASNON (*IRISA*)

Erwan FOUCHÉ & Julien BOUVET (*Sopra Steria*)



Projet en collaboration avec

Jean-Yves LE CLERC

(*Archives départementales d'Ille-et-Vilaine*)

Sophie TARDIVEL (*Doptim*)



Table des matières

1	Introduction	1
2	Architecture Générale et technologies utilisées	2
2.1	Architecture générale	2
2.2	Interactions	5
2.3	Technologies	6
3	Client	7
3.1	La communication client-serveur	7
3.2	Les différents composants de l'interface	8
3.3	Interactions entre les différents composants de l'interface	10
4	Serveur	11
4.1	Traitemet des données	11
4.2	Base de données	12
4.3	Interface avec le reconnaiseur	13
4.4	Contrôleur	15
5	Conclusion	16
6	Annexes	17

Chapitre 1

Introduction

Ce projet nous a été proposé par l'équipe [IntuiDoc](#) de l'[IRISA](#), en collaboration avec la startup [Doptim](#) et avec le soutien de Jean-Yves LE CLERC, conservateur du patrimoine aux [archives départementales](#) d'Ille-et-Vilaine. Tout au long de l'année, nous serons encadrés par Bertrand COÜASNON, enseignant-chercheur membre d'IntuiDoc, Erwan FOUCHÉ, chef de projet chez [Sopra Steria](#) et Julien BOUVET, ingénieur chez Sopra Steria également. Nous serons aussi accompagnés par Sophie TARDIVEL, responsable et *data scientist* chez Doptim.

Ce rapport définira la structure de notre projet, ainsi que les moyens de conception mis en oeuvre pour répondre aux spécifications définies dans les précédents rapports. Certains choix de notre conception sont justifiés par le contexte du projet, que nous rappellerons au besoin. Le chapitre 2 du rapport présentera l'architecture logicielle générale de l'application, ses différentes parties, ainsi que leurs interactions, tandis que les parties suivantes présenteront ces parties de l'architecture plus en détail. La solution technique du projet étant basée sur un modèle client / serveur, nous présenterons le côté client dans le chapitre 3 et le côté serveur dans le chapitre 4. La partie présentant le côté client traitera de l'organisation de l'IHM, tandis que celle qui présente le côté serveur définira sa structure détaillée pour chaque module individuellement. Nous expliquerons également les interactions entre ces modules. Enfin, nous décrirons le fonctionnement du projet à l'aide de diagrammes de séquence avant de conclure ce rapport. Nous avons décidé de conduire ce projet sur deux itérations. Nous aborderons donc pour chaque partie ce qui est intégré à la première itération, et ce qui sera rajouté dans la seconde.

Chapitre 2

Architecture Générale et technologies utilisées

Ce projet a pour but de fournir un programme permettant de concevoir des bases d'apprentissage automatiquement pour l'entraînement de systèmes de reconnaissance d'écriture manuscrite. Il sera notamment exploité par les [archives d'Ille-et-Vilaine](#) ainsi que la startup [Doptim](#). Les reconnaiseurs utilisés pouvant être multiples, il faut que ce projet puisse facilement évoluer, qu'une partie du projet puisse être remplacée par un morceau plus adapté au reconnaisseur choisi. Ainsi, tous les modules de notre projet et non seulement l'interface avec le reconnaiseur doivent pouvoir être remplacés par l'implémentation choisie par l'utilisateur. Par exemple, nous avons choisi une base de données intégrée avec *SQLite* mais celle-ci ne peut gérer facilement l'accès concurrentiel, ou gérer efficacement une grande quantité de données. Ainsi, l'utilisateur pourrait choisir d'utiliser une autre base de données comme *MySQL* ou *MongoDB*. Nous avons donc dû prendre en compte dans l'architecture l'aspect interchangeable de nos modules. Notre *back-end* est, nous le rappelons, écrit en Scala, et utilise des bibliothèques externes pour la gestion du JSON et des images (*OpenCV* par exemple). Nous utiliserons un serveur basé sur [Grizzly](#), une technologie utilisée dans un projet précédent cette année, et qui nous a paru simple d'utilisation, ainsi que [Jersey](#) pour l'API REST, pour les mêmes raisons.

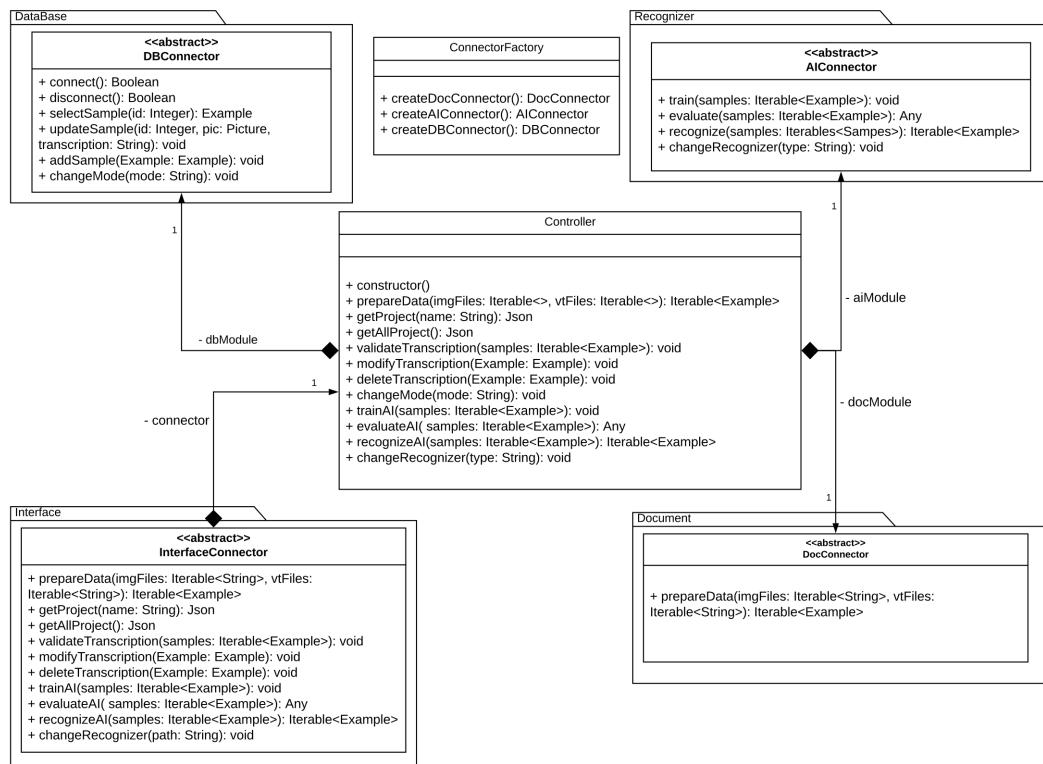
Architecture générale

Architecture du serveur

Le serveur de ce projet est composé de trois principaux modules représentant les différents besoins du projet. Ainsi, il nous faut traiter les données d'entrée fournies par l'utilisateur sous la forme d'un document scanné et possiblement d'une vérité terrain afin de les transformer en données utilisables par les reconnaiseurs. Il nous faut également pouvoir stocker les bases d'apprentissage qui constituent le cœur de notre projet. Enfin, ces bases ne serviraient à rien s'il n'était pas possible d'interfacer notre projet avec le reconnaiseur de l'utilisateur.

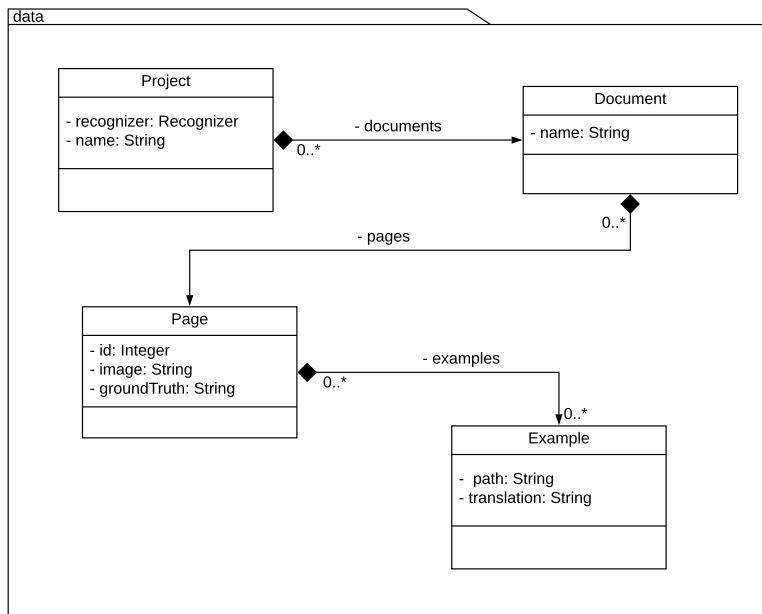
Notre projet étant composé de parties bien distinctes, la mise en place de modules indépendants et pouvant être remplacés par l'utilisateur n'a donc pas posé de problème. Nous avons créé une structure constituée des différents *packages* correspondant aux fonctionnalités ainsi qu'une interface faisant le lien entre tous. De cette manière, chaque partie est détachée de l'ensemble global, et l'interface centrale qu'on appellera *Controller* fera appel aux méthodes nécessaires des différents *packages*, afin de répondre aux demandes de l'utilisateur. Les *packages* auront alors une interface à implémenter permettant une utilisation indépendante de l'implémentation.

Figure 1 : Architecture des modules avec le connecteur



Dans notre projet, nous aurons également besoin de représenter les objets avec lesquels nous travaillons. Ainsi, nous avons choisi d'implémenter des classes de données pour représenter les exemples d'apprentissage (Example), les pages des documents utilisés (Page), lesdits documents (Document) et enfin les projets (Project) car on peut imaginer que l'utilisateur puisse vouloir avoir un projet sur des archives paroissiales et un autre sur des textes arabes anciens.

Figure 2 : Structure du package de données



L'attribut `Recogniser` est converti en chaîne de caractères dans la base de données en passant par un `enum` recensant les différents reconnaiseurs existants. Ces classes de données sont liées à la structure de la base de données et seront donc expliquées plus profondément dans cette partie.

Architecture du client

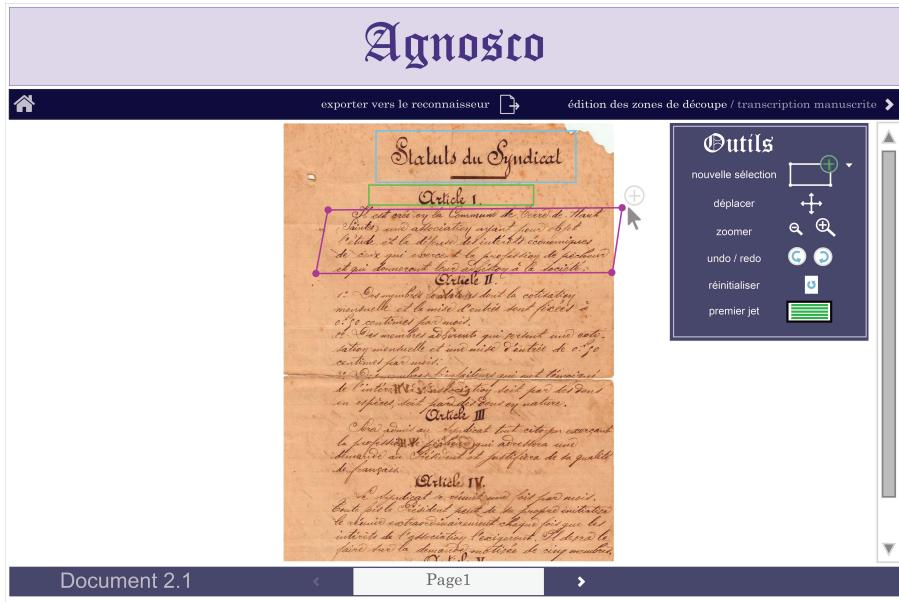
L'interface est composée de cinq pages distinctes. Il y a tout d'abord la page d'accueil, où l'utilisateur choisit le projet sur lequel il veut travailler ou en crée un nouveau en choisissant les documents qui le composent. La page d'accueil est représentée par la maquette de la figure 3.

Figure 3 : Maquette de la page d'accueil de l'IHM



L'interface possède également une page de découpe des zones ou des paragraphes des pages des documents à l'aide d'outils graphiques dédiés. La maquette de cette page est présentée à la figure 4.

Figure 4 : Maquette de la page de découpe des zones



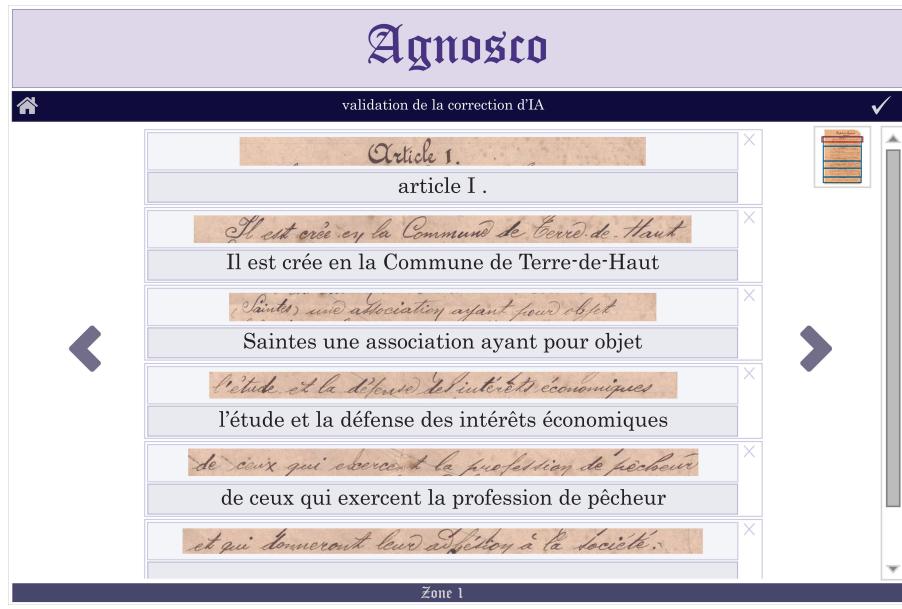
Ensuite, l'utilisateur peut choisir d'annoter lui-même le document scanné en tapant à la main les transcriptions des imagettes ou de faire passer le document par un reconnaiseur d'écriture manuscrite. L'interface propose donc une page d'annotation manuelle ainsi qu'une page de visualisation et de correction des transcriptions proposées par le reconnaiseur. Ces deux pages sont sensiblement similaires et leur maquette est présentée à la figure 5.

Figure 5 : Maquette de la page d'annotation manuelle



Enfin, la dernière page de l'interface est la page de validation des transcriptions où l'utilisateur fait une dernière relecture des annotations et les valide rapidement. La maquette de cette page est présentée à la figure 6.

Figure 6 : Maquette de la page de validation des annotations



Interactions

Avant d'aller plus en avant dans le projet, il est important de comprendre comment les données circulent au sein de l'application.

Tout d'abord, l'utilisateur choisit en entrée la Page avec laquelle il souhaite travailler. Elle est déjà enregistrée dans la base de données mais elle n'a pas été traitée. La Page, qui est associée à une vérité terrain, passe alors dans le module *Préparation des données* afin d'être découpée en imagettes et de lier à celles-ci les transcriptions correspondantes. Une fois cela effectué, les Examples ainsi générés vont être stockés dans la base de données en étant indiqués comme non validés. L'utilisateur va ensuite pouvoir valider les ensembles d'Examples, et une fois cela fait, ils sont envoyés au Recogniser qui va les utiliser comme ensemble d'entraînement.

Technologies

Précisons à présent les technologies que nous avons choisi d'utiliser pour ce projet.

Serveur

Le serveur étant un ensemble de modules, chaque module utilise une technologie différente en lien avec sa fonction. Nous aborderons en premier les technologies nécessaires à la préparations des données, puis sur celles liées à leur stockage. Nous verrons ensuite celles utilisées par l'API REST développée pour communiquer avec le client et enfin, celles de l'interface avec le reconnaiseur. Le langage principalement utilisé côté serveur est Scala.

Préparation des données

Pour gérer la découpe des images, nous avons décidé d'utiliser la bibliothèque [OpenCV](#), car celle-ci fournit des outils de découpe et de traitement d'image adaptés à ce que nous souhaitions faire.

Stockage des données

Pour stocker les données, nous avions décidé de nous orienter vers un système de gestion de base de données. Nous avions décidé, par simplicité et du fait que peu de contraintes (demandes d'accès simultanés, lourd nombre d'image) allaient s'imposer à ce système, de choisir un gestionnaire de base de données n'utilisant pas de serveur et local, au moins pour la première itération du projet, ce qui nous a fait porter notre choix sur [SQLite](#), qui est une technologie simple d'utilisation avec une bibliothèque d'interfacage avec la JVM facile à prendre en main.

API REST

Ayant fait le choix d'utiliser un client Web, nous avons alors dû réfléchir à développer une API REST afin que le client puisse effectuer ses requêtes sur le serveur. Il nous fallait tout d'abord établir un serveur qui serait en mesure de recevoir des requêtes HTTP en provenance du client. Notre choix s'est alors porté sur [Grizzly](#) et [Jersey](#), que nous avons utilisés dans un projet précédent en Java. Il nous fallait également de quoi construire et déconstruire nos objets locaux au serveur en JSON afin de pourvoir les transmettre au client. Nous avons alors choisi d'utiliser la bibliothèque [org.json](#) qui répond à ces problèmes.

Interface avec le reconnaiseur

Nous avons décidé de permettre à l'utilisateur d'utiliser un reconnaiseur afin de proposer une première transcription des imagettes afin de permettre plus d'ergonomie dans l'utilisation de l'application. Pour cela, il nous a été proposé d'utiliser [Laia](#) qui est un système de reconnaissance d'écriture manuscrite. Celui-ci peut être utilisé sous forme de conteneur Docker afin de ne pas télécharger sur sa machine toutes les dépendances du système. [Laia](#) permet notamment de créer et de gérer un reconnaiseur paramétrable selon le type de problème et selon les données d'entrée.

Client

Concernant le client, nous avons opté pour une application web afin de permettre une évolution vers un possible contexte multiutilisateur dans le cas où notre projet serait déployé sur un serveur distant et auquel les utilisateurs accèderaient au travers d'une page web. De nombreux frameworks web existent et nous avons choisi d'utiliser [Angular 7](#), soit la dernière version de celui-ci. Nous avons déjà utilisé ce framework durant un projet précédent, ce qui ne nous oblige pas à réapprendre un nouvel outil, mais à consolider les bases que nous avions dessus afin d'avoir au plus vite une version fonctionnelle. De plus, il possède une bonne documentation et de nombreux guides existent.

Chapitre 3

Client

La communication client-serveur

L'interface affichera les informations de la base de données (les exemples, ou les couples imagettes - transcriptions) et les modifiera. Pour ce faire, elle sera reliée au connecteur central via sa ressource REST. Le connecteur traitera les demandes de l'IHM et les enverra au connecteur de la base de données. Celui-ci exécutera les ordres de l'utilisateur sur la base et renverra les imagettes et les transcriptions au connecteur central qui les fera suivre à l'IHM. Les modifications (ajout ou modification d'une transcription ou suppression d'un exemple) seront enregistrées localement puis envoyées à la base de données lors du changement de page et du chargement d'une nouvelle page.

Les communications entre le client et le serveur se feront via des appels à une API REST dont nous définissons les différentes requêtes dans la liste ci-dessous. Chaque composant du client nécessitant des requêtes REST différentes, la liste est donc organisée par composant et l'organisation des appels sera détaillée par la suite.

Général

GET /base/projectsAndDocuments

Renvoie la liste des noms des projets, contenant pour chaque projet une liste des noms des documents qui le composent.

POST /base/createNewProject/{project_name}/{list_docs}

Crée un nouveau projet à partir du nom du projet et de la liste des noms des documents passée en paramètre.

DELETE /base/deleteDocument/{name}

Supprime le document de la base qui porte le nom donné, ainsi que son contenu.

GET /base/availableRecognisers

Renvoie la liste des noms des reconnaiseurs disponibles sur le serveur.

POST /base/exportRecogniserExamples/{name}

Récupère tous les exemples de la base qui sont contenus dans des projets utilisant le reconnaiseur dont le nom est passé en paramètre. Les exemples sont triés, ne sont retenus que ceux qui sont définis comme utilisables et validés. Ensuite, ces exemples sont exportés en tant que lot d'entraînement, vers le format d'entrée associé au reconnaiseur en question.

Annotation / Validation

GET /base/documentPages/{name}

Renvoie la liste des identifiants en base de données des pages qui composent un document dont le nom est donné en paramètre.

GET /base/pageData/{id}

Renvoie l'image associée à la page dont l'identifiant est donné en paramètre, ainsi que la liste des imagettes et des transcriptions des exemples qui la composent.

POST /base/saveExampleEdits

Enregistre dans la base de données les modifications de transcriptions décrites par l'objet JSON associé à la requête.

PUT /base/disableExample/{id}

Rend l'exemple dont l'identifiant est donné en paramètre inutilisable, i.e. l'utilisateur juge que l'image est trop parasitée ou que l'exemple n'est pas pertinent.

PUT /base/enableExample/{id}

Rend l'exemple dont l'identifiant est donné en paramètre utilisable (principalement utilisé pour annuler l'action décrite ci-dessus).

POST /base/validateExamples

Valide tous les exemples dont les identifiants sont fournis dans une liste JSON associée à la requête.

Découpe

GET /base/documentPagesWithImages/{name}

Renvoie la liste des identifiants et des images associés aux pages qui composent le document portant le nom donné.

POST /base/addDocumentGroundtruth/{name}

Ajoute à la base la vérité terrain donnée dans l'objet PiFF (donc JSON) associé à la requête, et l'associe au document dont le nom est donné en paramètre.

GET /base/recogniseImages/{name}

Envoie la liste des imagettes contenues dans le document donné au reconnaiseur associé au projet contenant ce document. La réponse à cette requête est une liste de transcriptions trouvées par le reconnaiseur.

Les différents composants de l'interface

Angular permet d'organiser les interfaces web par composants. Notre interface ne contiendra que des composants indépendants interagissant les uns avec les autres. Tous les composants seront regroupés dans le composant de *routing* app-root.

Première itération

Pour la première itération, l'interface est réduite à sa forme la plus simple qui répond au cahier des charges, à savoir ouvrir un document de travail et valider ou invalider les transcriptions. L'IHM aura donc deux composants : la page d'accueil et la page de validation des annotations.

La page d'accueil permet à l'utilisateur de choisir le projet sur lequel il veut travailler, ainsi que d'en créer de nouveaux en choisissant les documents qui le composent. Pour afficher la liste des projets existants, le composant accueil appellera la requête GET /base/projectsAndDocuments et, pour créer un nouveau projet, il appellera POST /base/createNewProject/{project_name}/{list_docs}. L'utilisateur a également la possibilité de supprimer un document existant en faisant appel à la requête DELETE /base/deleteDocument/{name}.

La page de validation, comme son nom l'indique, permettra de valider ou d'invalider les transcriptions de la base de données relatives au document ouvert. Ces transcriptions ayant été réalisées par des humains, elles seront considérées comme vraies, et l'utilisateur pourrait les invalider seulement s'il manque des mots, ou si l'imagette est jugée peu pertinente pour l'apprentissage (par exemple, si le texte est presque illisible ou si l'imagette ne contient pas de texte du tout).

Pour ce faire, le composant validation réalisera différents appels à l'API REST. Tout d'abord, il faut pouvoir identifier quelles pages composent le document ouvert par l'utilisateur. Cela sera effectué par l'appel à la requête GET /base/documentPages/{name}. Pour obtenir la liste des imagettes et des transcriptions qui composent une page, le composant appellera GET /base/pageData/{id}. Lorsque l'utilisateur changera de page après avoir parcouru toutes les transcriptions d'une page, on réalisera un appel à la requête POST /base/saveExampleEdits afin d'enregistrer en base de données les modifications effectuées sur les transcriptions. De plus, un exemple peut être invalidé en appelant la requête PUT /base/disableExample/{id} ou au contraire redevenir valide après l'appel à la requête PUT /base/enableExample/{id}. Enfin, l'appel à POST /base/validateExamples permet de valider tous les exemples valides de la page.

Deuxième itération

Pour la deuxième itération, nous implémenterons les fonctionnalités supplémentaires décrites dans les rapports précédents.

Dans la page d'accueil, nous rajouterais la possibilité d'interagir avec un reconnaiseur d'écriture manuscrite. Pour cela, le composant appellera la requête GET /base/availableRecognisers afin d'obtenir la liste des noms des reconnaiseurs disponibles sur le serveur pour que l'utilisateur fasse son choix. De plus, l'appel à la requête POST /base/exportRecogniserExamples/{name} permettra de récupérer tous les exemples de la base contenus dans des projets utilisant le reconnaiseur dont le nom est passé en paramètre. Les exemples seront triés pour ne retenir que ceux définis comme utilisables et validés. Ensuite, ces exemples seront exportés en tant que lot d'entraînement, vers le format d'entrée associé au reconnaiseur en question.

Nous proposerons également une page de découpe des zones ou des paragraphes du document à l'aide d'outils graphiques. Pour ce faire, nous aurons besoin de récupérer l'image de la page scannée afin de la découper, ce qui sera effectué via l'appel à la requête GET /base/documentPagesWithImages/{name} qui renvoie la liste des identifiants et des images associés aux pages qui composent le document portant le nom *name*. En outre, ce composant devra permettre d'ajouter dans la base de données la vérité-terrain générée dans cette page de l'IHM. Ceci sera effectué via la requête POST /base/addDocumentGroundtruth/{name}, *name* étant le nom du document ouvert. Enfin, il avait été spécifié dans le précédent rapport que la page de découpe devra posséder un bouton permettant d'envoyer la liste des imagettes de la page découpée au reconnaiseur associé au projet contenant le document ouvert pour qu'il propose des transcriptions. Ainsi, un *click* sur ce bouton fera appel à la requête REST GET /base/recogniseImages/{name}.

Lors de l'ouverture d'un nouveau document de travail, il devra être possible d'utiliser des manuscrits scannés qui ne possèdent pas de vérité terrain. L'utilisateur aura alors la possibilité d'annoter le manuscrit manuellement ou de le faire passer par un reconnaiseur d'écriture après avoir découpé les zones du manuscrit via la page de découpe de l'interface.

L'interface possèdera donc une page d'annotation manuelle ainsi qu'une page de visualisation et de correction des résultats de l'apprentissage du reconnaiseur. Les deux composants seront sensiblement les mêmes, à quelques différences près décrites dans le rapport de spécification.

Les appels REST effectués par ces composants seront les mêmes et seront identiques à ceux effectués par le composant de la page de validation, soit GET /base/documentPages/{name} pour obtenir les identifiants des pages composant le document ouvert, GET /base/pageData/{id} qui renvoie la liste des exemples qui composent une page, POST /base/saveExampleEdits afin d'enregistrer en base de données les modifications effectuées sur les transcriptions, ainsi que PUT /base/disableExample/{id} ou PUT /base/enableExample/{id} pour cacher ou réhabiliter un exemple. Enfin, on pourra valider tous les exemples valides grâce à la requête POST /base/validateExamples.

Enfin, la dernière page de l'interface de la deuxième itération sera la page de validation qui a déjà été réalisée lors du premier rendu. Deux modifications y seront apportées pour la deuxième itération.

Tout d'abord, une fenêtre sera ajoutée pour visualiser la page courante en miniature, avec ses zones de découpe affichées par dessus, ainsi qu'un rectangle montrant la position courante de l'utilisateur, afin que ce dernier ait constamment accès à sa position dans le document. Nous obtiendrons l'image de la page entière grâce à la requête GET /base/pageData/{id}.

Ensuite, la bannière en haut de la page affichera si les transcriptions ont été réalisées manuellement par un humain ou par le reconnaiseur d'écriture, car l'attention fournie par l'utilisateur devrait être moindre pour un manuscrit annoté manuellement où les fautes sont théoriquement moins nombreuses. Savoir si la transcription a été faite manuellement ou par le reconnaiseur permettrait d'augmenter la rapidité de la validation finale. Cette information sera également obtenue grâce à l'appel REST précédent.

En résumé, à la fin de la deuxième itération, nous aurons cinq composants différents : la page d'accueil, la page de découpe des zones du document, la page d'annotation manuelle, la page de visualisation de la reconnaissance et la page de validation finale des transcriptions.

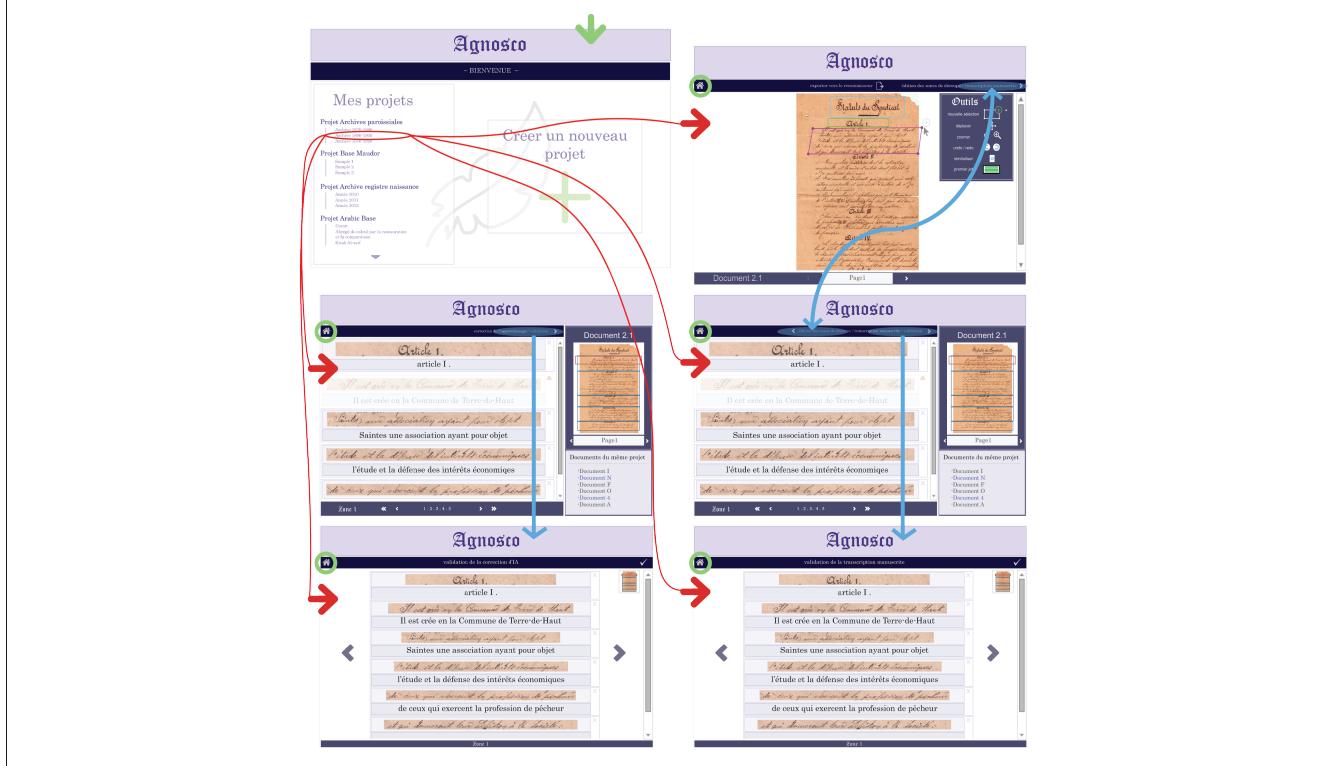
Interactions entre les différents composants de l'interface

L'utilisateur peut naviguer entre les différents composants via les boutons de l'interface. Tout d'abord, une icône permettant de revenir à la page d'accueil sera présente en haut à gauche de chaque page - à l'exception de la page d'accueil. Sur cette dernière, le choix du document ouvre celui-ci dans la page de l'interface qui lui correspond, soit :

- la page de découpe si aucun travail n'a été réalisé au préalable sur ce document ;
- la page d'annotation manuelle si les découpes ont déjà été réalisées et que l'utilisateur n'a pas exporté le document vers le reconnaiseur ;
- la page de visualisation des transcriptions du reconnaiseur si une reconnaissance a été réalisée sur le document ;
- ou la page de validation finale si toutes les transcriptions ont déjà été renseignées.

Des boutons de navigation dans les différentes pages sont également présents sur certaines pages, comme décrit dans la partie précédente. Les interactions sont visibles dans le schéma de la figure 7.

Figure 7 : Schéma des interactions entre les différents composants de l'IHM



Chapitre 4

Serveur

Notre serveur est comme expliqué précédemment divisé en plusieurs modules : un module de traitement des données d'entrée, un module de gestion de la base de données contenant les différents projets sur lesquels travaille l'utilisateur, l'interface avec le reconnaiseur d'écriture manuscrite, ainsi que le contrôleur qui permet de diriger ces sous-parties du logiciel.

Traitement des données

Pour ce qui est du traitement des données, il nous faut un *package* pour chacune des deux tâches concernées, à savoir la lecture des fichiers d'entrée ainsi que la découpe d'image.

Lecture des fichiers d'entrée : *package* input

Comme expliqué dans le dernier rapport, nous avons choisi de ne traiter qu'un seul format dans notre logiciel, le format PiFF. Pour pouvoir lire les fichiers d'entrée, on doit construire une représentation des données contenues dans ces derniers sous la forme d'objets. Ceci constituera le *package* piff. Il définit une classe PiFF, qui contient des pages (PiFFPage), qui elles-mêmes contiennent des portions de texte (PiFFEelement). Ces classes peuvent être converties au format JSON (bibliothèque org.json), ce qui permet d'exporter les objets vers un fichier PiFF afin de répondre à la spécification PR_F0_1 pour l'utilisation du format PiFF en interne.

L'utilisateur doit toutefois pouvoir utiliser d'autres formats afin de permettre l'évolution de notre logiciel comme spécifié dans les précédents rapports (GEN_EVO). Pour cette raison, nous avons un *package* converters contenant une interface, PiFFConverter, qui permet de convertir un fichier quelconque en objet PiFF. Nous en fournissons une implémentation, l'objet GEDIToPiFFConverter, qui comme précisé dans les précédents rapports (PR_FO_2), permet au logiciel de lire le format GEDI. Celui-ci est utilisé notamment par la base de données Maurdor, présentée dans les précédents rapports, et à laquelle nous avons accès pour nos tests. L'utilisateur peut rajouter autant d'implémentations qu'il le souhaite.

En plus de ces deux *packages*, nous proposons un objet PiFFReader (singleton), qui permettra d'ouvrir un fichier, et d'utiliser les concepts présentés ci-dessus. Plus précisément, il permettra de lister des implémentations de PiFFConverter, et de les appeler une par une sur le fichier d'entrée pour réussir à le lire.

Découpe des images : *package* processing

Pour la découpe d'image, il nous faut une classe ImageProcessing, qui appelle les méthodes de la bibliothèque OpenCV, que nous avons choisie précédemment. Cette bibliothèque nous permet de découper les images afin d'obtenir les imagettes selon les lignes ou les paragraphes (PR_TR_2 et PR_TR_4). Après la découpe, les imagettes sont associées au texte pour former des exemples, que nous avons modélisés par une classe Example (spécifications PR_RE_1 et PR_RE_2).

La phase de découpe étant automatique, il nous faut faire appel à un détecteur de lignes. C'est la fonction du package `linedetection`. Nous y plaçons une interface `LineDetector`, qui permet de trouver les lignes de texte dans un objet PiFF (spécification `PR_TR_1`). Le détecteur de lignes utilisé pour le projet est fourni par l'encadrant, et fonctionne sous Linux. Les membres du groupe peuvent pourtant travailler sous Windows ou macOS. Pour faciliter le développement du logiciel, nous avons donc choisi de fournir une implémentation, la classe `BlurLineDetector` (nom dû à la méthode de détection), basée sur des connexions réseau, afin de pouvoir faire fonctionner le serveur sous n'importe quel système d'exploitation, avec un petit module serveur sous Linux qui appelle simplement l'exécutable. Ce petit module ne fait pas partie du cahier des charges mais sera développé pour les tests. L'utilisateur, ici aussi, pourra rajouter ses propres méthodes de détection de lignes s'il le souhaite.

Planification Cette partie du projet sera intégralement développée pour la première itération. En effet, on a besoin des fonctionnalités de découpe d'image et de traitement des vérités terrain dès la première itération que nous avons définie.

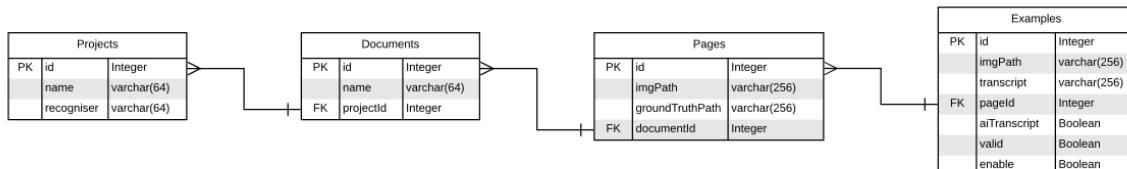
Base de données

Le Système de Gestion de Bases de Données (SGBD) choisi est SQLite. La bibliothèque Java SQLite, une implémentation de l'API JDBC (Java DataBase Connectivity), nous servira à appeler des commandes SQL directement depuis notre programme. Il nous faut cependant remanier les relations entre les tables de la base de données, car en nous concertant avec l'équipe à propos de l'ergonomie à apporter ainsi que de la forme que devaient prendre les données, nous avons remarqué que notre précédente organisation ne pouvait fournir tous les éléments nécessaires. En effet, dans les précédents rapports, la forme que devait prendre l'IHM quand aux données venant du serveur et dont elle aurait besoin était plutôt floue. C'est en se penchant plus en avant lors de ce rapport que nous l'avons remarqué.

Architecture

La nouvelle base de données se base sur les classes de données présentées dans la première partie. Elles sont donc composées d'une table `Projects` qui contient les projets existants ainsi que le `Recogniser` associé. Celui-ci est représenté sous la forme d'une chaîne de caractères car un `enum` sera mis en place afin de constituer les différents reconnaiseurs possibles et qu'on peut restorer l'`enum` facilement à partir d'une chaîne de caractères. Une autre table `Documents` représente les différents documents qui constituent notre base, par exemple, "Archives paroissiales de 1870-1872". Ceux-ci sont donc représentés par un nom et sont liés à un `Project`. Une troisième table nous permet de conserver les `Pages` des `Documents`. Elles sont donc liées à une image (le scan de la page) et à une vérité terrain (la description de l'image) qui sont stockées au travers de leur chemin dans le système de fichier pour les raisons expliquées dans les rapports précédents (`STO_VER`). Les `Pages` sont également associées au `Document` dont elles sont issues. Enfin, une dernière table `Examples` stocke les `exemples` générés par la découpe. Ils sont constitués du chemin vers l'imagette correspondante ainsi que la transcription de celle-ci, sachant que cette transcription peut être nulle dans le cas où on souhaite utiliser le `reconnaisseur` pour proposer une transcription (`STO_REC`), ou quand la vérité terrain n'a pas été fournie et qu'il faut la construire (`STO_USR`). Les `Examples` sont également liés à la `Page` d'où ils ont été découpés.

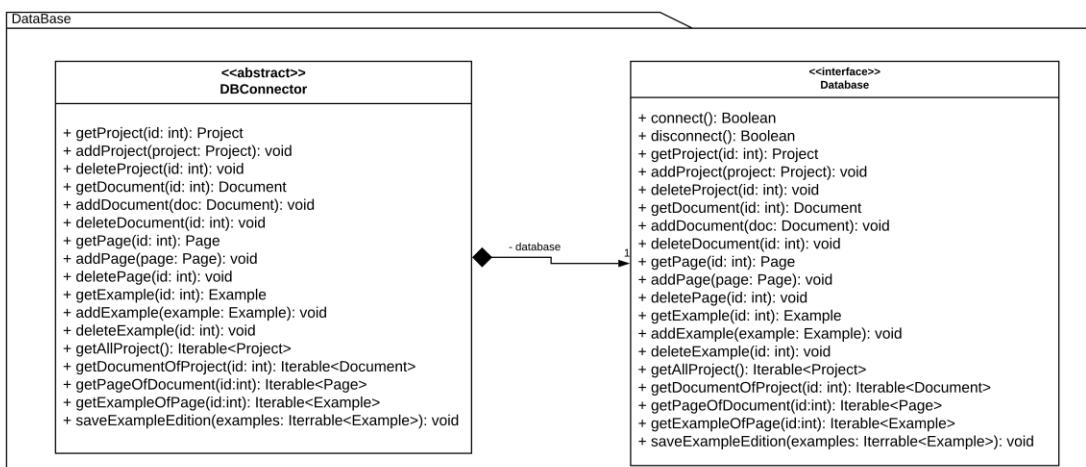
Figure 8 : Structure de la base de données



Cette structuration nous permet de hiérarchiser les exemples d'apprentissage générés par notre logiciel et donc de pouvoir fournir une plus grande ergonomie au sein de l'application client.

Le module de la base de données est constitué d'une interface représentant cette base de données ainsi que les opérations qui seront effectuées dessus (*STO_SEL*, *STO_UPD*, *STO_INS*, *STO_DEL*). Ainsi, il est possible de récupérer les différents instances de la hiérarchie de données. Nous avons préféré garder les requêtes sur la base de données simples et élémentaires afin de permettre plus de flexibilité sur les opérations qui les utiliseront ensuite. De cette manière, plusieurs fonctions du Controller pourront appeler la même fonction dans la base plutôt que d'avoir des demandes précises sur la base de données. De cette manière, l'évolution du projet en est simplifiée en permettant l'établissement simple de nouvelles fonctions dans le Controller sans avoir à rajouter de requêtes dans la base de données.

Figure 9 : Architecture de la base de données

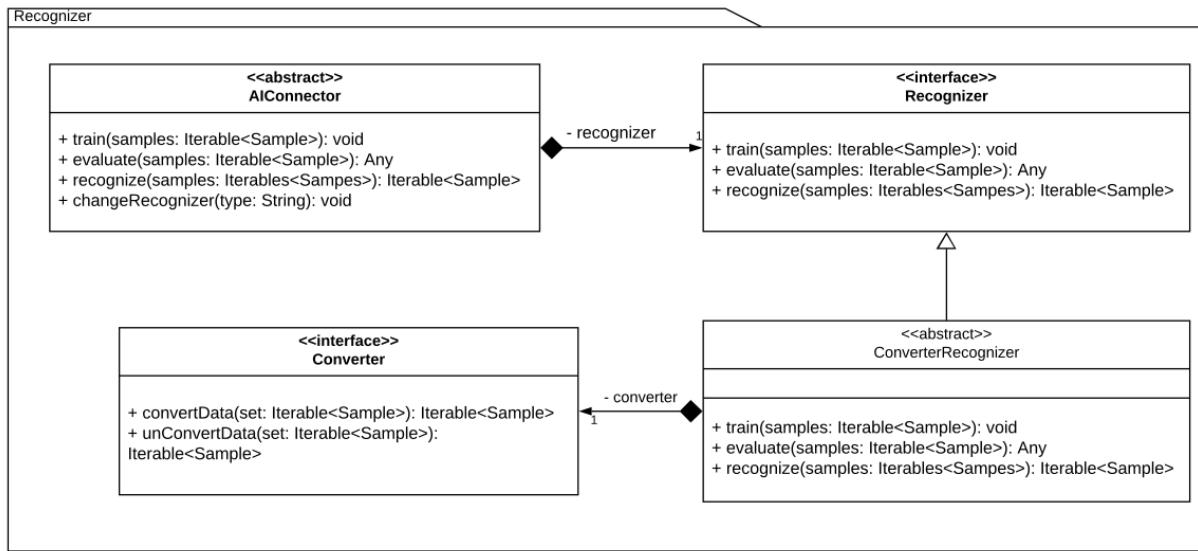


Planification Comme l'ensemble des méthodes de la base de données sont nécessaires pour les appels de l'API REST, toute la classe devra être implémentée. L'ensemble des tables qu'elle manipule seront aussi créées. Pour la seconde itération de notre projet, de nouvelles méthodes ne seront pas nécessaires comme nous réutiliserons celles déjà créées pour les ajouts de méthodes dans le contrôleur. En revanche, pour cette seconde partie, le serveur et le client pouvant être sur des machines distinctes, nous devrons envisager un envoi des images au client, et non pas seulement des chemins de fichiers. La solution technique exacte n'est pas encore fixée, nous y réfléchissons en se basant sur les conseils de nos encadrants, qui ont eu des situations similaires dans leurs projets professionnels.

Interface avec le reconnaiseur

Cette partie du projet a pour objectif de lier la base d'apprentissage de notre logiciel avec le reconnaiseur choisi par l'utilisateur. Ainsi, cette partie étant très liée à l'utilisateur, il faut lui permettre de pouvoir facilement attacher le reconnaiseur de son choix au logiciel. C'est avec cette contrainte en tête que nous avons conçu l'architecture de cette partie.

Figure 10 : Architecture de l'interface avec le reconnaiseur



Architecture Cette partie du projet contient comme toutes les autres un connecteur qui la lie au Connector principal. Ce connecteur possède un **Recogniser** afin de pouvoir effectuer les opérations classiques d'entraînement, d'évaluation de celui-ci ainsi que lui demander d'effectuer une transcription. Il permet également de changer de reconnaiseur quand l'utilisateur souhaite utiliser un autre type de reconnaiseur.

Le reconnaiseur est représenté par une interface **Recogniser** qui permet trois actions possibles : entraîner le reconnaiseur à partir d'un ensemble d'apprentissage, évaluer ce même reconnaiseur sur un ensemble de validation, et enfin, lui demander de transcrire un ensemble non étiqueté. Ces fonctions ont été construites afin de répondre aux spécifications *IR_AP*, *IR_EV* ainsi que *IR_TR*. Cette interface permet notamment à l'utilisateur de connecter son propre reconnaiseur, et même s'il le souhaite, d'en créer un au sein de l'application. Elle est également la seule qu'il est obligé d'implémenter pour faire fonctionner l'ensemble. Cependant, il faut dans certains cas convertir les données dans la base de données en données compréhensibles par le reconnaiseur. Ainsi, deux autres interfaces sont nécessaires : une interface **Converter** permettant la conversion des données entrantes et sortantes du reconnaiseur (*IR_CV*) ainsi qu'une classe abstraite **ConverterRecogniser** héritant de **Recogniser** et possédant un convertisseur adapté au reconnaiseur utilisé. En effet, le reconnaiseur peut avoir besoin d'un formatage des images en entrée, mais également de la transcription associée. Par exemple, nous implémenterons la structure décrite précédemment pour un reconnaiseur particulier : **Laia**. Ce reconnaiseur demande de transformer les images afin qu'elles aient la même hauteur en pixels, et que la transcription soit traduite dans les symboles qu'il peut reconnaître et organisée d'une certaine façon. Par exemple, il faut remplacer les ' ' par '<space>' et que l'identifiant de l'image correspondante à la transcription soit indiqué avant celle-ci dans un fichier. Le convertisseur va donc se charger de mettre en forme les données et de créer ce qui est nécessaire.

Planification Dans la première itération délivrée le 27 Février, la possibilité d'entraîner le reconnaiseur doit être implantée ainsi que celle de changer de reconnaiseur. Dans un second temps, il sera possible de faire transcrire les imagettes par le reconnaiseur et de faire l'évaluation de celui-ci. Les données remontées lors de l'évaluation sont laissées libres à l'utilisateur puis la fonction rend un objet non défini à l'avance qui sera sous la forme d'un *JSON* et qui contiendra les informations choisies par l'utilisateur dans son implémentation. Il pourra alors obtenir le nombre et la fréquence des erreurs, sur quelles lettres celles-ci surviennent et ainsi de suite.

Contrôleur

Architecture

Le rôle du contrôleur est de mettre en relation les 4 parties du projet : le *client*, i.e. l'interface homme-machine, la base de données, ainsi que le traitement des données. Pour ce faire, nous avons 4 classes abstraites, un connecteur par *package* du projet, chacun possédant des méthodes générales, qui appellent d'autres méthodes de leurs *packages* respectifs. Cela permet d'avoir un objet *Controller* central qui manipule tous les éléments du projet avec un code très lisible. Nous fournissons une implémentation de chaque connecteur, pour le bon fonctionnement du projet. Une fabrique est également prévue, permettant de choisir l'implémentation des connecteurs de chaque partie sans avoir à modifier le code du connecteur central. En effet, le contrôleur récupèrera les instances des connecteurs à partir de cette fabrique. Cela permet à l'utilisateur de changer d'instance de connecteur en changeant une ligne de code seulement.

Pour la première itération de notre projet, le *Controller* devra permettre de relier les différents modules au travers de leurs interfaces de connection ainsi que permettre les exécutions des demandes en provenance de l'utilisateur, c'est à dire le client.

Interactions

Le connecteur central reçoit donc, au travers du connecteur en relation avec le client les demandes de l'utilisateur et, afin d'exécuter ces demandes, appelle au travers des connecteurs des autres modules les méthodes nécessaires. Le *Controller* sert alors d'intermédiaire entre l'utilisateur et les différents modules. Il permet notamment cette structure modulaire en détachant les appels de méthodes des classes réelles qui peuvent alors être changées selon le bon vouloir de l'utilisateur.

Chapitre 5

Conclusion

Dans le cadre de notre projet de 4e année à l'INSA, nous avons pour objectif de conduire un projet depuis la rédaction du cahier des charges jusqu'à sa livraison finale en passant par les spécifications, la conception et le développement. Une fois le cahier des charges et les spécifications réalisées, nous avons rédigé le présent rapport, détaillant la conception de notre projet.

Nous avons découpé ce rapport en deux parties principales : le côté client et le côté serveur, après avoir rappelé le contexte de notre projet de génération de bases d'apprentissage pour la reconnaissance d'écriture.

Dans la première partie, nous avons expliqué nos choix des technologies utilisées pour la réalisation de l'interface. Nous avons ensuite détaillé les différents composants de l'IHM pour la première itération puis pour la deuxième, avant de décrire les interactions entre les pages de l'IHM et entre le *front-end* et le *back-end*.

La deuxième partie était axée sur le côté serveur de l'application. Nous avons tout d'abord décrit l'architecture générale de notre projet. Nous avons ensuite détaillé la conception des blocs de la découpe des images, de la base de données et de l'interface avec le reconnaiseur, en précisant dans chaque partie l'architecture correspondant aux deux itérations successives ainsi que les interactions de chaque bloc avec le reste de l'application.

Ce rapport est émaillé de schémas d'architecture, de diagrammes de classes ainsi que de maquettes de l'interface pour la clarté et la lisibilité, afin de permettre une meilleure compréhension à la lecture.

Le projet se poursuivra par une phase de développement afin de produire un premier rendu fonctionnel pour le 27 février.

Chapitre 6

Annexes

Vocabulaire utile

Le **deep learning** est une méthode d'apprentissage automatique basée sur un réseau de neurones. Ainsi, un algorithme utilisant le *deep learning* apprend par lui-même et devient de plus en plus performant au fur et à mesure qu'il accumule les exemples. Il suffit de lui spécifier les paramètres du problème qu'il doit résoudre et lui donner des exemples sur lesquels s'entraîner.

Une **base d'apprentissage (ou base d'entraînement)** est un ensemble d'exemples que l'on fournit à un algorithme de *deep learning* afin que celui-ci puisse apprendre.

La **vérité terrain** est, dans le cadre de notre projet, un ensemble de documents numériques correspondant à des documents manuscrits. Ces documents numériques contiennent la transcription des documents manuscrits ainsi que diverses informations telles que la position des paragraphes dans ces documents ou encore les numéros de ligne par paragraphe. Cette vérité terrain a été établie au préalable par des humains et non de manière automatique.

Une **imagette** correspond, dans le cadre de notre projet, à une partie de texte manuscrit découpée au format image. Elle peut correspondre à une ligne ou à un paragraphe du document.

Une **retranscription** est, dans le cadre de notre projet, la transcription tapée d'un texte manuscrit.

Les formats **GEDI** et **PiFF** sont des formats de description d'images. Ils contiennent une image et des métadonnées (comme la position des paragraphes par exemple). Le format GEDI est le format de la base qui nous est donnée pour nourrir notre base d'apprentissage. Le format PiFF est un format de description d'images qui tend à être universel.

Codes de specification

Bloc 1 : Préparation des données	
Spécification	Description
PR_FO_1	Traiter le format PiFF en interne dans le logiciel
PR_FO_2	Fournir un convertisseur du format GEDI vers PiFF
PR_TR_1	Intégrer une fonction de détection de lignes au logiciel
PR_TR_2	Permettre un découpage des images en lignes
PR_TR_3	Localiser les paragraphes
PR_TR_4	Permettre un découpage des images en paragraphes
PR_RE_1	Associer les images à leur transcription
PR_RE_2	Associer la vérité terrain à une transcription
PR_RE_3	Permettre de générer une vérité terrain si besoin, grâce à un reconnaiseur

Bloc 2 : Stockage des données	
Spécification	Description
STO_VER	Stocker des imagettes associées à une vérité terrain
STO_USR	Stocker des imagettes associées à une transcription générée par l'utilisateur
STO_REC	Stocker des imagettes associées à une transcription générée par un reconnaisseur
STO_SEL	Fournir des méthodes pour accéder aux données stockées
STO_UPD	Fournir des méthodes pour modifier les données stockées
STO_INS	Fournir des méthodes pour pouvoir insérer des données à stocker
STO_DEL	Fournir des méthodes pour pouvoir supprimer des données stockées

Bloc 3 : Interface avec le reconnaiseur	
Spécification	Description
IR_CV	Convertir les données au format d'entrée du reconnaiseur
IR_AP	Fournir les données au reconnaiseur
IR_EV	Pouvoir lancer une évaluation du reconnaiseur
IR_TR	Pouvoir lancer une transcription d'un document par le reconnaiseur

IR_CV	Convertir les données au format d'entrée du reconnaiseur
IR_AP	Fournir les données au reconnaiseur
IR_EV	Pouvoir lancer une évaluation du reconnaiseur
IR_TR	Pouvoir lancer une transcription d'un document par le reconnaiseur

Bloc 4 : Interface avec l'utilisateur	
Spécification	Description
PEA_GEN_1	Valider un ensemble d'annotations
PEA_GEN_2	Éditer manuellement les transcriptions
PEA_GEN_3	Corriger les annotations proposées par un reconnaisseur externe à l'application
PEA_GEN_4	Envoyer les modifications à la base de données lorsque la vérité-terrain d'une imagette est modifiée
PEA_GEN_5	Ignorer un couple imagette-transcription s'il n'est pas pertinent
PEA_GEN_6	Regrouper les documents en projets
PEA_GEN_7	Sélectionner d'abord le projet puis le document sur lequel l'utilisateur veut travailler à l'ouverture de l'application
PEA_GEN_8	Créer un nouveau projet
PEA_GEN_9	Basculer vers la page de découpe des zones
PEA_GEN_10	Basculer vers la page d'édition des annotations
PEA_GEN_11	Basculer vers la page de validation des transcriptions
PDEC_OD_1	Créer une nouvelle zone à l'aide d'un rectangle (outil "nouvelle sélection")
PDEC_OD_2	Pouvoir modifier la position des sommets des rectangles
PDEC_OD_3	Rajouter des sommets à la zone
PDEC_OD_4	Changer le type de la zone avec un menu déroulant
PDEC_OD_5	Déplacer la zone sélectionnée sur le document (outil "déplacer")
PDEC_OD_6	Zoomer et dézoomer sur le document (outils "zoom +" et "zoom -")
PDEC_OD_7	Annuler la dernière action (outil "annuler")
PDEC_OD_8	Refaire l'action annulée précédemment (outil "refaire")
PDEC_OD_9	Supprime toutes les zones de la page pour retourner au document vierge (outil "réinitialiser")
PDEC_OD_10	Applique un détecteur de lignes sur la zone sélectionnée (outil "appliquer la détection de lignes sur la zone")
PDEC_OD_11	Continuer la découpe du document sur la page suivante
PDEC_OD_12	Passer à l'édition des annotations sur la page qu'il vient de découper
PDEC_OD_13	Exporter la page découpée au format PiFF afin de soumettre les données à un reconnaisseur externe à l'application
PDEC_OD_14	Posséder un bouton de retour au menu principal
PDEC_OD_15	Permettre à l'utilisateur de se déplacer sur le manuscrit à l'aide d'un scroll horizontal et vertical
PEMA_1	Placer le curseur sur la première imagette ne possédant pas de transcription
PEMA_2	Positionner le curseur sur l'annotation suivante en appuyant sur Entrée
PEMA_3	Proposer un raccourci clavier permettant de basculer vers la prochaine imagette sans vérité-terrain
PEMA_4	Posséder un bouton intitulé "modifier les zones du manuscrit"
PEMA_5	Afficher la liste des imagettes du document découpé
PEMA_6	Ignorer un couple imagette-transcription s'il n'est pas pertinent
PEMA_7	Basculer vers la page de validation des transcriptions
PCORIA_1	Valider les transcriptions zone par zone et passer à la zone suivante avec un simple appui sur Entrée
PCORIA_2	Pouvoir modifier une annotation fausse en cliquant dessus pour y positionner son curseur et en effectuant ses modifications manuellement
PCORIA_3	La zone de visualisation des imagettes se présente de la même manière que sur la page d'édition manuelle des transcriptions
PCORIA_4	Posséder également la fonctionnalité de mise à l'écart d'un couple imagette-transcription
PCORIA_5	Basculer vers la page de validation des transcriptions

Spécification	Description
PVAL_1	Accéder à la page de validation depuis le menu principal
PVAL_2	Accéder à cette page de validation depuis les pages d'édition manuelle des annotations et de correction des transcriptions proposées par le reconnaiseur
PVAL_3	Valider les transcriptions zone par zone et passer à la zone suivante avec un simple appui sur Entrée
PVAL_4	Pouvoir modifier une annotation fausse en cliquant dessus pour y positionner son curseur et en effectuant ses modifications manuellement
PVAL_5	Indique si les transcriptions ont été fournies manuellement par un humain ou si elles proviennent d'un reconnaiseur
PVAL_6	Faire figurer une fenêtre montrant la page entière découpée en zones avec la zone courante dans une couleur différente
PVAL_7	Valider le travail pour de bon et fermer le document à l'aide d'un bouton prévu à cet effet

Bloc 5 : Lien entre les blocs précédents	
Spécification	Description
LINK_PR_STO	Envoyer les données en entrée vers le système de stockage
LINK_STO_IHM	Extraire les données pour les fournir à l'IHM
LINK_STO_IR	Extraire les données pour les fournir au système de reconnaissance
LINK_IHM_STO	Envoyer les demandes de l'IHM au système de stockage
LINK_IHM_IR	Envoyer les résultats du reconnaiseur vers le système de stockage
LINK_COH	Fournir un logiciel composés de blocs communiquant entre eux de manière fonctionnelle et cohérente

Bloc 6 : Général	
Spécification	Description
GEN_ERGO	Ergonomie de l'application
GEN_ERGO	Concevoir un logiciel évolutif
GEN_ERGO	Fournir un logiciel open source



INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839
35708 Rennes Cedex 7
Tél. +33 [0] 2 23 23 82 00
Fax +33 [0] 2 23 23 83 96

www.insa-rennes.fr

INSA

UNIVERSITE
BRETAGNE
LOIRE

Cti
Commission
des Titres d'Ingénieur

