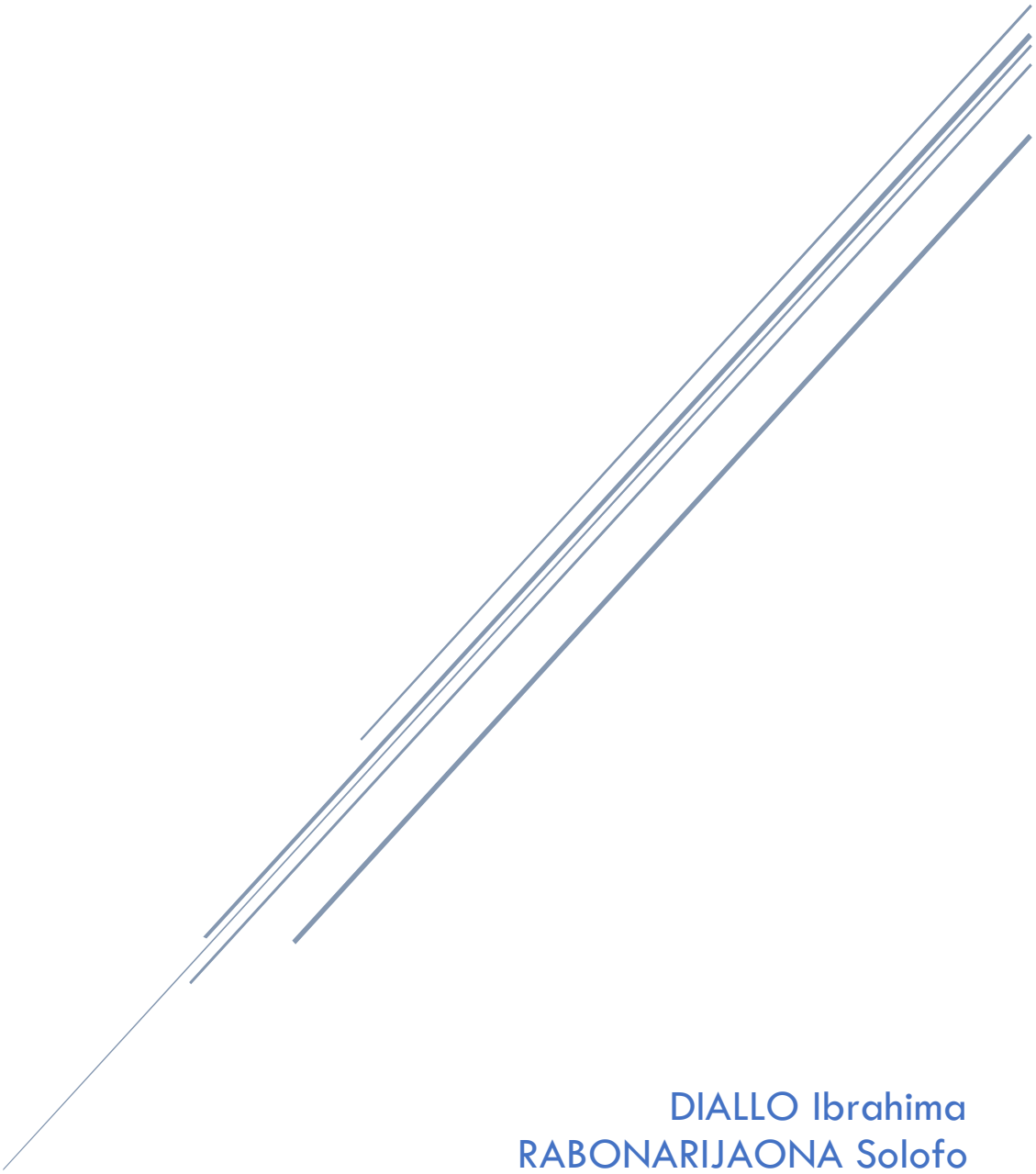
Rapport

Projet documents numériques



DIALLO Ibrahima
RABONARIJAONA Solofo
ROMDAN Elias
TROTTA Nicolas

1 Introduction

1.1 Formulation du problème

De nos jours, à l'intérieur des organisations des milliers et des milliers d'informations s'échangent chaque jour entre les différents intervenants, qui peuvent être des humains mais aussi des machines. Pour avoir une trace de toutes ces informations qui circulent, l'idée est de construire une base de données et de la modifier au fur et à mesure en fonction du flux d'informations. Un être humain ne pourra jamais gérer une telle quantité d'informations, d'où l'utilité de créer un programme qui fera l'intermédiaire entre les informations et la base de données.

Plus spécifiquement, dans notre projet, nous devons créer une application Web qui permettra d'enregistrer toutes les transactions qui se produisent dans une entreprise de vente d'avions où les échanges de données sont assez importants. En entrée, l'application récupérera des fichiers de type XML qui contiennent des informations, ces fichiers passeront par une série de contrôles de validité, avant que la base de données soit mis à jour avec le contenu de ces fichiers. En sortie, une interface graphique s'occupera d'afficher le contenu de la base de données.

1.2 Découpage en sous-tâches

Nous avons découpé le projet en plusieurs sous-tâches pour que chaque membre du groupe puisse avancer sur une partie de façon indépendante. Bien évidemment à la fin du développement, toutes les parties seront amenées à communiquer entre elles pour assurer le bon fonctionnement du produit.

Pendant le découpage, nous avons commencé par fixer un modèle général de ce que le produit sera à l'état final. Ensuite, nous avons isolé les différents modules qui composent le produit. Enfin, nous avons assigné à chaque membre du groupe, un module pour le réaliser selon ses compétences et son aisance avec les technologies qui seront utilisées pour le développement de ce module.

Nos discussions ont émergé les principales sous-tâches suivantes :

- La réalisation du module du scrutateur.
- La réalisation du module du validateur.
- La réalisation du module du parseur.
- La création d'une interface graphique.
- La création d'une base de données.
- La création des fichiers XML de tests.
- La création d'un fichier Log des transactions sur la base de données.

2 Description de la réalisation des principales sous-tâches

2.1 Module du scrutateur

Le module scrutateur est un programme Java qui utilise la bibliothèque WatchService pour scruter le répertoire docnum/projet/data/XML/. Le scrutateur traite uniquement les nouveaux fichiers créés dans le répertoire et ignore les mises à jour et suppressions de fichiers.

Le programme scrutateur est lancé en parallèle avec l'application web développé avec le Framework Spring en utilisant un Thread qui sera lancé au démarrage du serveur.

2.2 Module du validateur

Le module du validateur se partage entre un fichier Java et un fichier XSD. Le code Java s'occupe de récupérer le fichier XML détecté par le scrutateur, de vérifier que la taille du fichier est inférieure ou égale à 5 Ko, de vérifier que le type du fichier est XML, puis de passer la main au fichier XSD.

À la différence du code Java, le fichier XSD s'occupe de vérifier le contenu du fichier XML. Les balises du fichier XML doivent respecter à la lettre la structure et les règles définies par le fichier XSD.

Si le fichier XML échoue sur l'une de ces vérifications, le code Java rejettera le fichier puis s'occupera de notifier l'utilisateur à travers l'interface graphique. Sinon, le fichier XML passera du module du validateur vers le module du parseur.

2.3 Module du parseur

Pour rappel, un parseur XML est un outil qui décompose les données contenues dans un document XML et les transforme en entités capables d'être lues et traitées. En Java, ces informations XML sont converties en objets facilement manipulables, tel n'importe quel attribut déclaré dans une classe.

Dans les premières versions du prototype de l'application, nous avons décidé de tester 3 types de parseur : DOM (Document Object Model), SAX (Simple API for XML) et Jackson XML Parser. Nos 3 concurrents n'ont pas démérité mais au moment de trancher, nous n'avons pas pu rester sourds face aux arguments massues du parseur DOM :

- Une documentation et des ressources fournies afin de comprendre son fonctionnement.
- Une facilité d'utilisation issues de sa structure en arbre qui rend aisée la navigation, de la racine jusque dans les nœuds de la moindre sous-branche.
- Nos fichiers XML étant encore de taille raisonnable, le problème de la contrainte d'espace lors du stockage en mémoire du document ne se pose pas.

Loin d'être une usine à gaz, la classe « DocumentBuilderFactory » nous permet de construire un objet de type « DocumentBuilder » nous donnant accès à toutes les

méthodes nécessaires à l'extraction et à l'exploitation des données contenues dans un document XML. En quelques lignes, voici le principe du parseur DOM appliqué à notre propre projet :

```
File inputFile = new File("data/XML/" + fileName);
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(inputFile);
```

- Nous récupérons dans un premier temps de type File fichier dans le dossier du scrutateur déjà passé par le scanner scrupuleux du « XSDValidator » afin de s'assurer que le XML soit conforme.
- Nous construisons une nouvelle instance de « DocumentBuilderFactory ».
- Nous créons un nouveau document de type « DocumentBuilder ».
- Puis enfin nous parsons les informations du XML dans ce nouvel objet.

Comme expliqué plus haut, le parser DOM transforme le fichier xml en arbre que l'on aura tout loisir de parcourir depuis sa racine jusqu'à la moindre feuille. Ici, l'élément <action> est la racine depuis laquelle nous pouvons accéder à ses branches, les éléments <avion> et tout leur contenu, pour rappel voici la structure du fichier XML :

```
<action intitule idFic nomVerif checksum dateAction>
    <avion id>
        <name></name>
        <weight></weight>
        <moteur type>
            <puissance></puissance>
            <nombre></nombre>
        </moteur>
    </avion>
</action>
```

L'intégralité des avions sera stockée dans un objet de type « NodeList » afin de pouvoir parcourir chacun d'eux à l'aide d'une boucle et ainsi accéder à leur contenu respectif :

```
/* pour chacune des branches */
for (int temp = 0; temp < nList.getLength(); temp++) {
    Node nNode = nList.item(temp);
    /* on vérifie que le noeud parcouru soit un élément */
    if (nNode.getNodeType() == Node.ELEMENT_NODE) {
       Element eElement = (Element) nNode;
       /* traitement des informations */
       /* ... */
     }
}
```

2.4 Sécurité

Chaque fichier XML destiné à être parsé, une fois délacé vers le dossier surveillé par le scrutateur, va être contrôlé par le module du **FileChecker** avant d'être confié aux bons soins du module du **FileParser**. Nous ne détaillerons pas la phase de validation du fichier XML car il dispose d'une partie dédiée mais il est important de noter que lorsqu'un fichier parvient jusqu'au parser, c'est qu'il est considéré comme conforme.

Cependant être d'une structure conforme ne signifie pas être automatiquement valide, c'est une condition nécessaire mais pas suffisante pour que les informations contenues dans le fichier soient traitées et inscrites en base de données. Une bonne partie des informations désormais extraites par le parser doit être analysé pour vérifier qu'elles sont conformes aux contraintes imposées par le cahier des charges de l'application.

Depuis le **FileParser**, nous allons devoir retourner des données au **FileChecker** afin de les vérifier et le cas échéant, d'annuler le traitement en cours. La classe action contient à elle seule 5 attributs dont 4 sont liés à la sécurité de l'opération :

- **intitule** : add, update ou delete
    - S'occupe d'aiguiller le traitement de chaque <action>
- **idFic** : identifiant unique du fichier
    - Chaque idFic de fichier est stocké en BDD, ainsi aucun ne peut être utilisé plus d'une fois
- **nomVerif** : nom du validateur
    - Faute de temps, cet attribut n'a pas pu être exploité
- **checksum** : nombre déclaré de conteneurs d'information (ici chaque balise <avion>)
    - Si le checksum ne correspond pas au nombre réel de conteneurs, alors le fichier est rejeté

- Ce nombre est vérifié en comptant chaque nœud de la liste d'<avion>

- **dateAction** : date déclarée de la création du fichier
  - Si une information date de plus de 3 mois alors le fichier est rejeté
  - La date est ici en format dateTime

En dehors des attributs de chaque attribut, il nous faudra aussi tester l'intégrité et la conformité des informations contenues dans chaque avion :

```
if(intituleAction.matches("add")){
    // Véfiriction unitaire des infos
    boolean isValidAddInfos =
        checker.checkInfo("'name'" + " dans le fichier " + fileName, avionName) &&
        checker.checkInfo("'weight'" + " dans le fichier " + fileName, avionWeight) &&
        checker.checkInfo("'type'" + " dans le fichier " + fileName, moteurType) &&
        checker.checkInfo("'puissance'" + " dans le fichier " + fileName, moteurPuissance);

    if(isValidAddInfos) {
     restClient.addNewAvion(avion, idAvion);
    }
}
else if(intituleAction.matches("update")){
    // Véfiriction unitaire des infos
    boolean isValidUpdateInfos =
        checker.checkInfo("'name'" + " dans le fichier " + fileName, avionName) ||
        checker.checkInfo("'weight'" + " dans le fichier " + fileName, avionWeight) ||
        checker.checkInfo("'type'" + " dans le fichier " + fileName, moteurType) ||
        checker.checkInfo("'puissance'" + " dans le fichier " + fileName, moteurPuissance);
    if (isValidUpdateInfos) {
     restClient.updateNewAvion(avion, idAvion);
    }
}
else if(intituleAction.matches("delete") && idAvion > 0L){
   restClient.deleteNewAvion(avion, idAvion);
}
```

Le rôle du parseur vis-à-vis de cette question va consister à transmettre au **Controller** les données vérifiées, cependant il est à noter qu'une dernière série

de tests aura lieu dans le **Controller** afin de ne pas créer de doublons dans la BDD ou bien tenter d'intervenir sur des champs ou des tables qui n'existent pas.

2.5 Interface graphique

Nous avons utilisé les technologies HTML5, la librairie de style Picnic.css, l'API Fetch et le Framework JavaScript Vue.js pour développer l'interface graphique de l'application.

Le contenu de la base de données, qui sera ensuite affiché sur l'interface graphique de l'application, est récupéré en utilisant l'API fetch. La mise à jour dynamique et asynchrone de l'interface graphique est gérée par le Framework Vue.js.

2.6 Base de données

Pour la persistance des données, nous avons opté pour la base de données en mémoire H2, vu la facilité et la rapidité à déployer et gérer une telle base. Pour une première version de notre application, notre base de données se compose uniquement de 3 tables.

La table Avion pour gérer les transactions d'avions. La table Moteur pour gérer les types de moteurs qui sont équipés aux avions. La table Message pour avoir une trace des fichiers détectés par le scrutateur.

La communication entre le serveur Spring et la base de données H2 est assuré par l'interface de programmation JPA qui fera office du langage des requêtes SQL.

2.7 Les fichiers XML

Tous les fichiers XML qui passeront par l'application, doivent respecter une certaine structure pour pouvoir être traité. Nous avons essayé de faire en sorte que la structure de ces fichiers soit la plus générique possible, pour éviter d'écrire du code redondant dans la partie du parseur.

La balise <action> qui se trouve en début de chaque fichier XML, décidera de l'action à réaliser par la suite dans la base de données. L'action pourra être un ajout défini par l'intitulé "add", une modification définie par l'intitulé "update" ou une suppression définie par l'intitulé "delete".

La balise <avion> contienne les informations à propos d'un avion qui seront transmises à la base de données. L'attribut "id" désignera l'identifiant unique de l'avion. Toutes les autres balises embarquées dans la balise <avion>, désigneront les propriétés d'un avion comme le nom, poids et type de moteur.

2.8 Le fichier Log

Chaque événement lié à l'entrée d'un fichier dans le répertoire gardé par le scrutateur et aux différents scénarios possibles, donne lieu à une consignation scrupuleuse dans un **fichier de log** de format textuel. Nous rappellerons seulement que si le fichier de log n'existe pas, celui-ci est créé, s'il existe, les données sont naturellement inscrites à la fin du fichier en cours d'écriture. A chaque détection d'un nouveau fichier, le parser fera appel à la classe **Scribe** chargée d'écrire

l'entête de chaque nouvelle événement grâce aux informations contenues dans les 5 attributs de la classe <action> du fichier XML (intitule, idFic, nomVerif, checksum et dateAction).

```
/* inscription des infos dans le log */
Scribe.logMemoire(
    "Intitule de l'action          : " + intituleAction + "\n" +
    "Identifiant unique du fichier : " + idFic + "\n" +
    "Nom du vérificateur           : " + nomVerif + "\n" +
    "Checksum déclaré du fichier   : " + checksum + "\n" +
    "Date de l'action déclarée     : " + dateAction + "\n\n"
);
```

Le reste des inscriptions dans le journal se fera depuis les autres classes de l'application dès qu'une information sera jugée non conforme ou bien au contraire lorsque des données qui auront passées tous les barrages de sécurité seront enfin inscrites avec succès dans la base de données. Ci-dessous, un exemple d'un fichier Log rempli par une succession d'actions sur la base de données :

```
Fichier de log du scrutateur crée le 2019-11-28 à 07:23:25

Intitule de l'action          : add
Identifiant unique du fichier : add_1
Nom du vérificateur           : XSDValidator
Checksum déclaré du fichier   : 5
Date de l'action déclarée     : 2019-11-23T10:20:15

AJOUT confirmé : L'avion AirbusA310 a été ajouté dans la BDD
AJOUT confirmé : L'avion Soukhoi Su-27 a été ajouté dans la BDD
AJOUT confirmé : L'avion Cenadair CL-415 a été ajouté dans la BDD
AJOUT confirmé : L'avion Lockheed C-130 Hercules a été ajouté dans la BDD
AJOUT confirmé : L'avion Lockheed SR-71 Blackbird a été ajouté dans la BDD



------------------------------

Modification en date du 2019-11-28 à 07:23:33

Intitule de l'action          : update
Identifiant unique du fichier : update_1
Nom du vérificateur           : XSDValidator
Checksum déclaré du fichier   : 5
Date de l'action déclarée     : 2019-11-24T05:23:15

UPDATE confirmé : Les infos de l'avion Lockheed C-130 Hercules ont été mises à jour
UPDATE confirmé : Les infos de l'avion Lockheed SR-71 Blackbird ont été mises à jour
UPDATE confirmé : Les infos de l'avion AirbusA310 ont été mises à jour
UPDATE confirmé : Les infos de l'avion Soukhoi Su-27 ont été mises à jour
UPDATE confirmé : Les infos de l'avion Canadair CL-415 ont été mises à jour
```

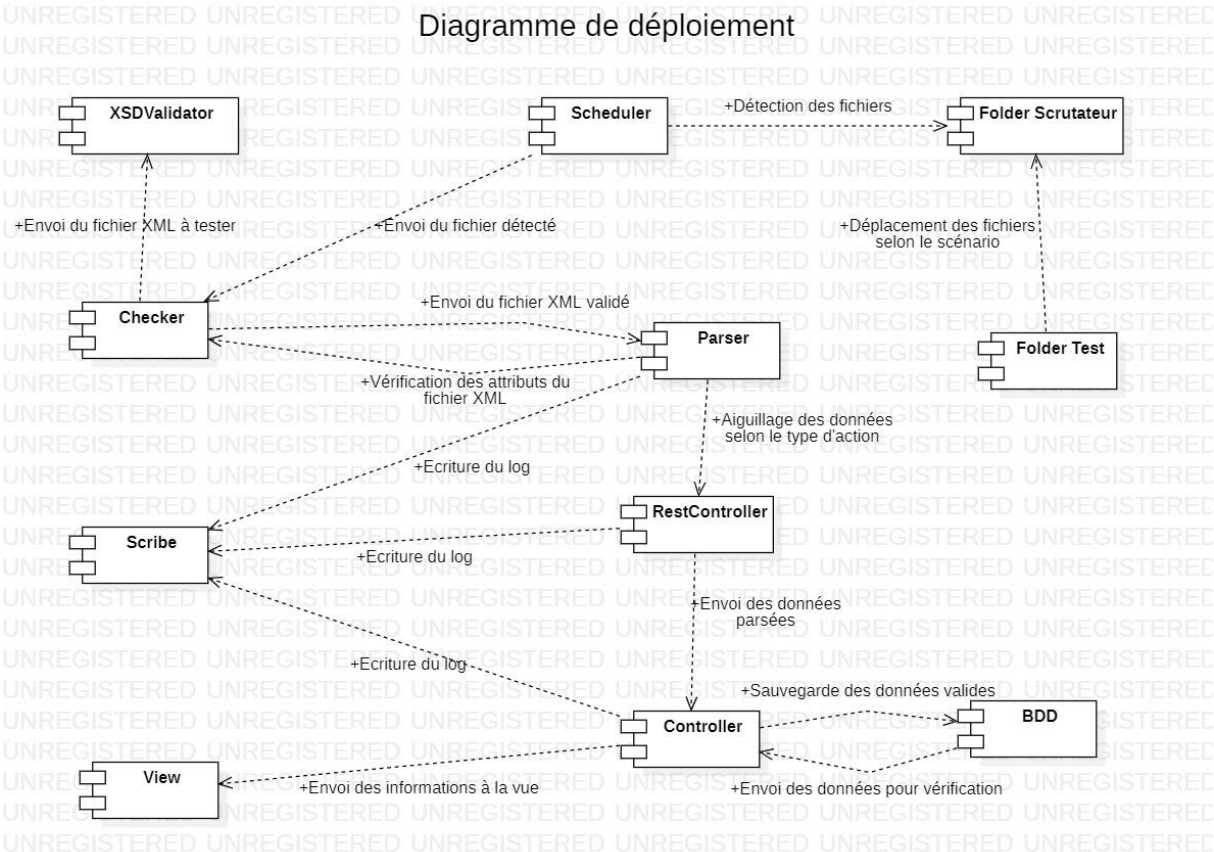
3 Intégration

3.1 Architecture générale

Nous avons fait le choix de développer ce projet en langage Java sur Spring Boot. Comme tous les Frameworks, celui-ci est une surcouche de la technologie sur laquelle il est construit et sur laquelle il propose toute une série de fonctions et de modèles pour standardiser le développement d'une application.

Un projet Spring Boot s'articule tout autour d'un module intitulé Contrôleur et annoté @Controller. Il intercepte les données transmises depuis les différents modules de traitement et les renvoie aux différents modules en charge de la partie vue. Chacune des pages Web de l'application est traitée par le contrôleur par l'intermédiaire d'une méthode qui lui est dédiée et qui renvoie selon les besoins une adresse URL ou bien le traitement d'une tâche spécifique.

Objectif principal de ce projet, la sauvegarde des informations stockées dans les fichiers XML arrive comme il se doit en fin de parcours une fois que toutes les vérifications ont été effectuées et se fait dans le **Controller** de l'application. La transmission des données au **Controller** se fera par l'intermédiaire de la classe **RestController**. Ci-dessous un diagramme de déploiement qui explique l'interaction entre les grands modules constituant notre application :



3.2 Préparation et réalisation de l'intégration

Pour assurer une intégration solide, nous avons dés le début commencé à utiliser l'outil de gestion de versions : Git. D'un côté, on gardera un historique de toute

l'évolution du code et d'un autre, on garantira une intégration fluide. Les sous-tâches étaient partagé de façon à éviter que 2 personnes ou plus se retrouvent à coder la même partie, mais à un moment donné, ces parties doivent se communiquer entre elles. Git interviendra dans ces situations, où il s'occupera de fusionner le code qui se trouve dans le dépôt avec le code en local. Vu que chaque développeur fera ses propres tests pour s'assurer que sa partie est bien intégré aux autres parties, on aura une assurance que le code qui se retrouve sur le dépôt Git est fonctionnel est bien intégré. En cas de conflit lors de l'intégration, les codeurs impliqués dans les parties en conflit décideront des modifications à faire pour résoudre ce conflit, qui est bien récurent quand il s'agit d'utiliser un espace de travail en commun.

4 Interface utilisateur

L'interface utilisateur se compose en 2 grandes sous-parties :

- Message du vérificateur : Il se charge de notifier l'utilisateur à propos de l'état du fichier détecté. Si le ficher est valide, son nom sera affiché en vert. Si le fichier est invalide, son nom avec la raison de son refus seront affichés en rouge. Cette sous-partie gardera un historique de tous les fichiers détectés depuis le lancement de l'application.
- Les tables dans la base de données : Il se charge d'afficher le contenu des tables Avion et Moteur qui se trouvent dans la base de données.

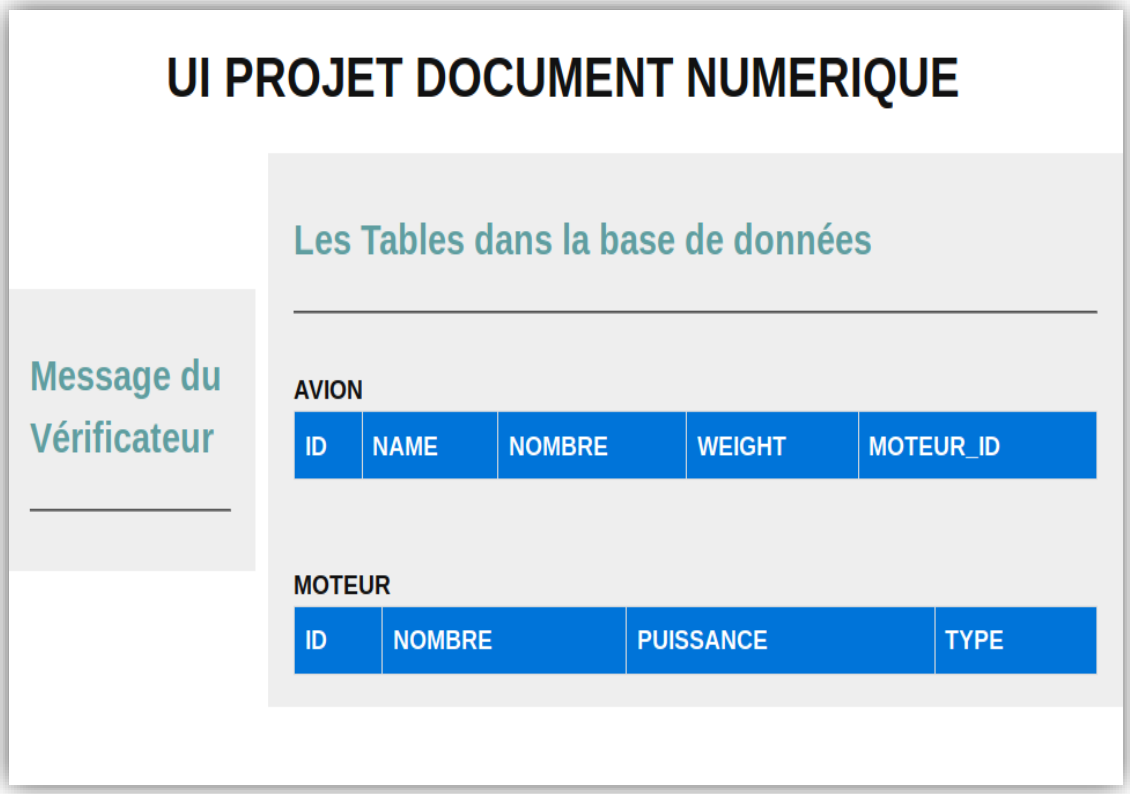


Figure 1 : Interface graphique au lancement

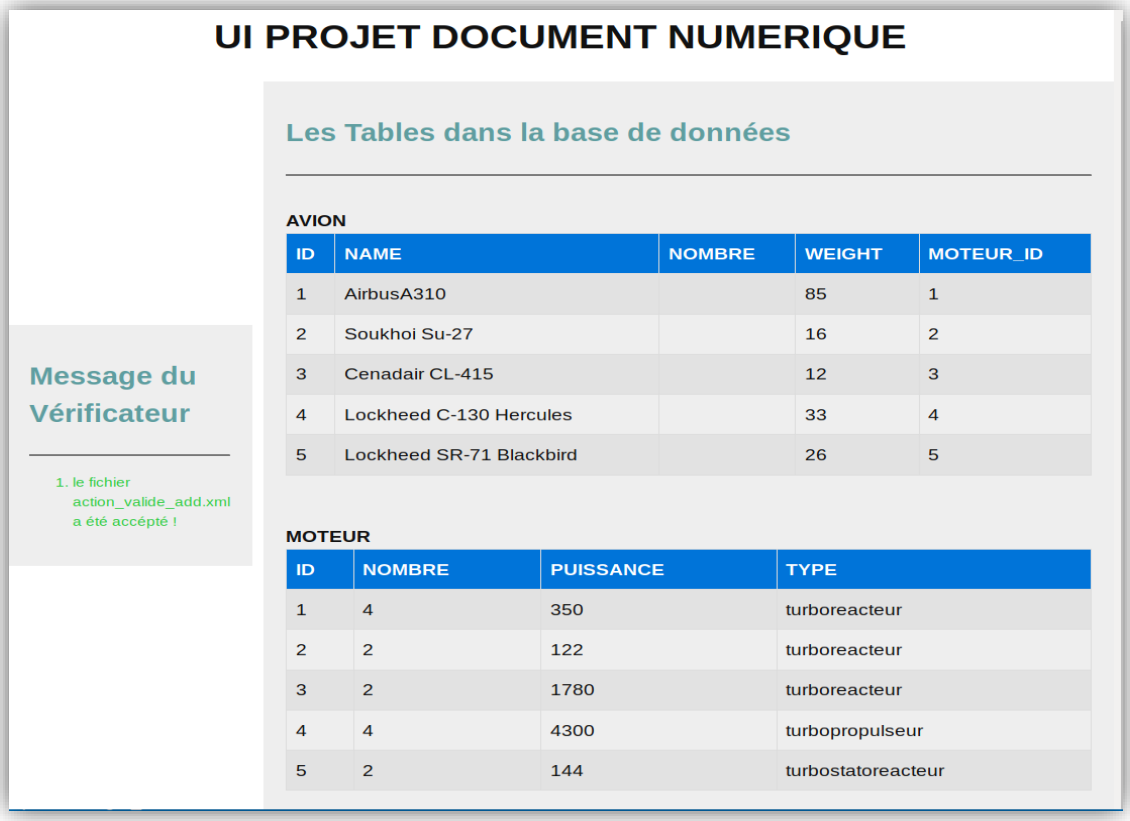


Figure 2 : Interface graphique à la détection d'un fichier d'ajout

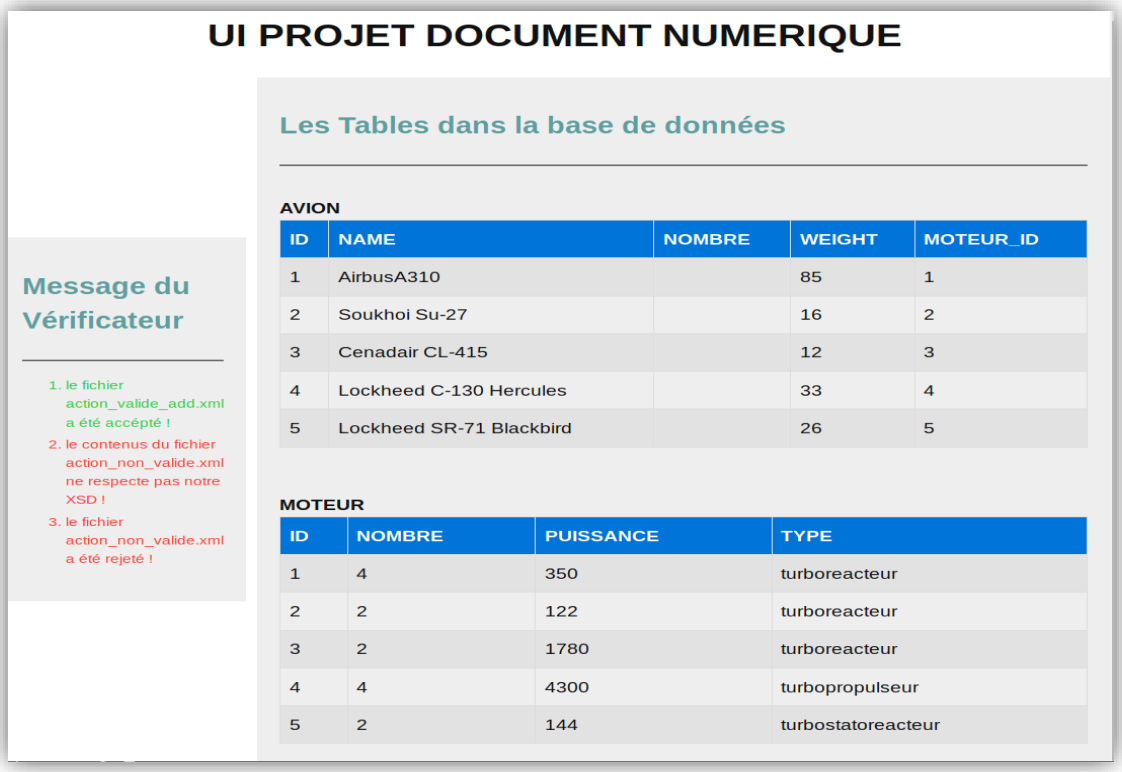


Figure 3 : Interface graphique à la détection d'un fichier non valide

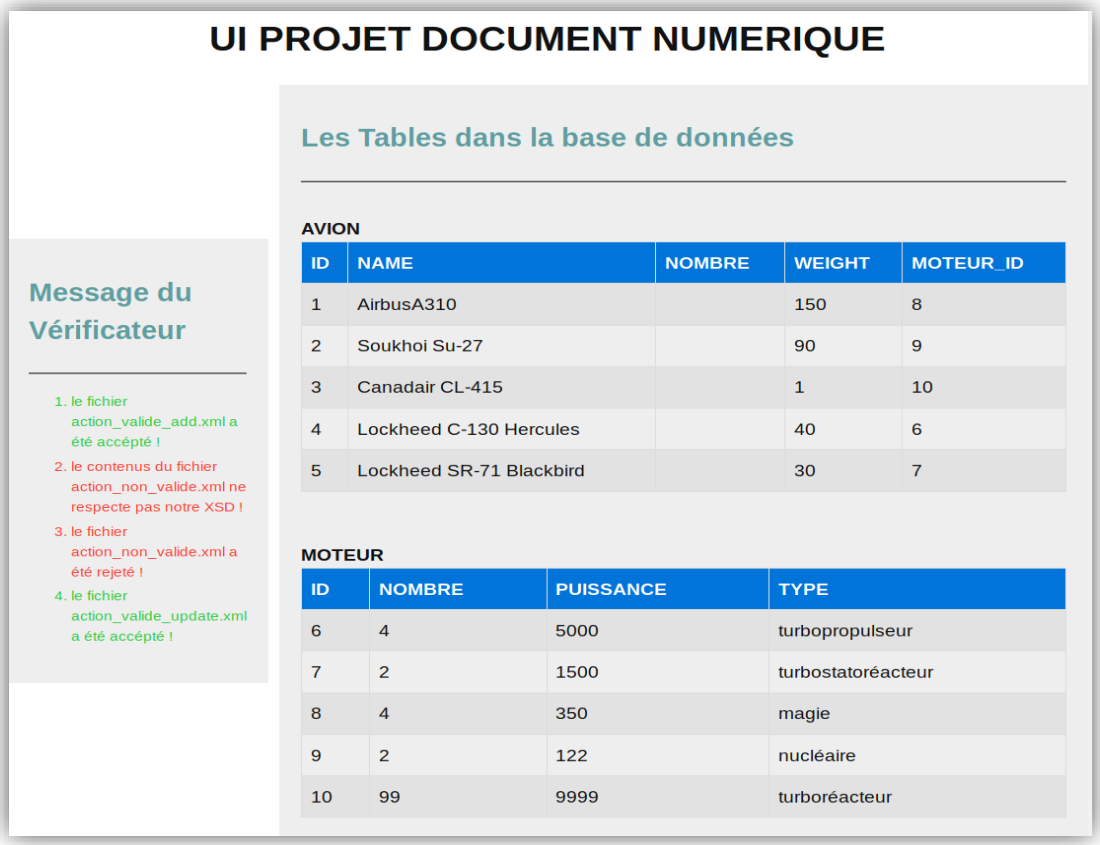


Figure 4 : Interface graphique à la détection d'un fichier de modification

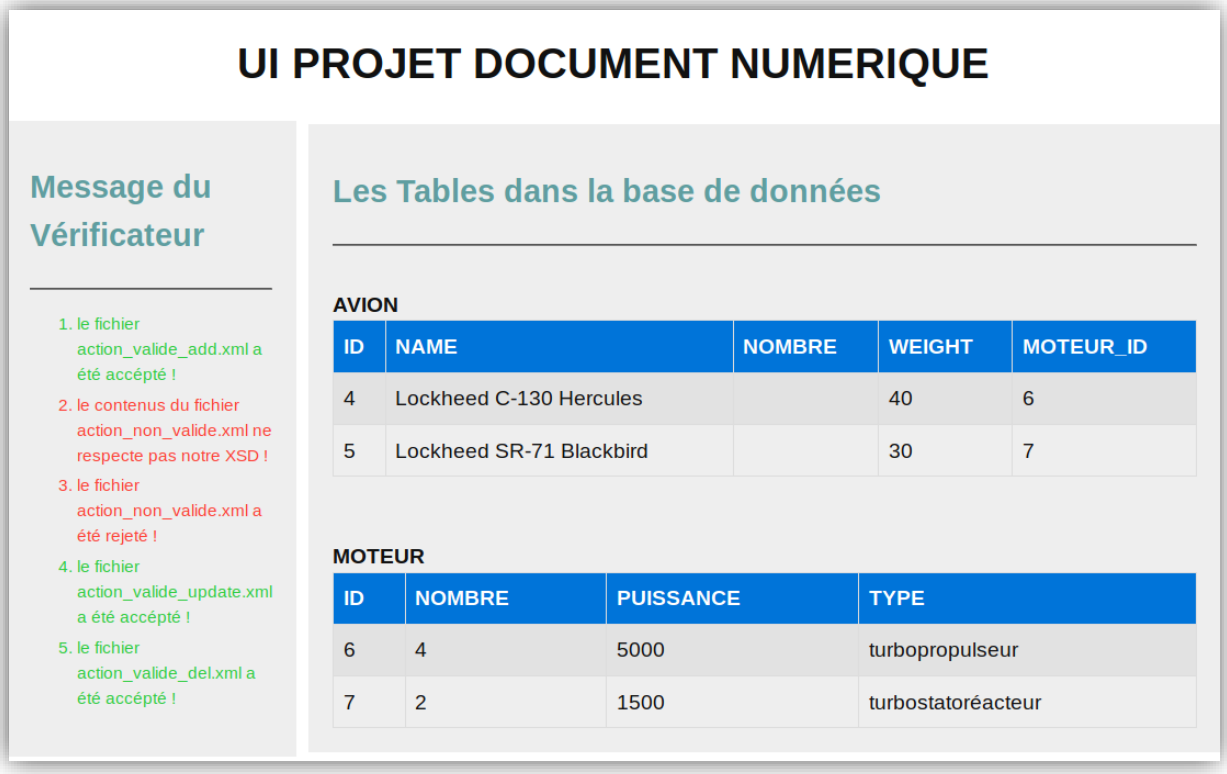


Figure 5 : Interface graphique à la détection d'un fichier de suppression

5 Plan et réalisation des tests

AVANT PROPOS

Objet des fichiers de test

Voici les fichiers de tests qui nous ont permis de prouver que :

- Le xsdValidator fait bien son travail
- Nos différentes sécurités ne laissent passées que les fichiers qui :
    - Répondent aux critères requis
    - Sont bien formés
- Le FileChecker fait bien son travail
- Les évènements sont bien enregistrés dans le fichier de log

Contenu du dossier "testdata"

- 4 fichiers xml valides
- 7 fichiers xml non valides

Usage des fichiers de test

Déplacer selon l'action souhaitée chaque fichier xml dans le répertoire :

- "projet/data/XML"

Protocole de test pour la présentation du 28/11/2019

Afin de mieux démontrer que le cahier des charges a bien été respecté, les fichiers XML doivent être ajouté au dossier " projet/data/XML " en respectant cet ordre général :

- Ajouter les fichiers valides puis les fichiers non valides
- Pour les fichiers non valides, l'ordre a moins d'importance puisque :
    - Chaque erreur est indépendante
    - Une seule erreur entraine le rejet du fichier

Dans le détail, il sera souhaitable et utile à la démonstration de procéder ainsi :

- Ajout des fichiers valides dans cet ordre :
    - valide_add.xml
        - Insertion des 5 avions de base
        - Permet toutes les manipulations ultérieures sur la BDD
    - valide_del.xml
        - Efface ce qui vient d'être ajouté
    - valide_update.xml
        - Met à jour les avions présents

        - Ne crée pas en doublons des avions non présents dans la BDD
    - valide_add.xml (à nouveau)
        - A ce moment-là, le fichier add_1 est déjà en mémoire
        - Permet de démontrer que la contrainte de l'identifiant unique est respectée
    - valide_add_idFic.xml
        - Comme le fichier avec l'identifiant unique add_1 est déjà présent en mémoire
            - Si on essaie de la rajouter à nouveau
                - Une erreur sera signalée et le fichier sera rejeté
            - Si on met un fichier d'ajout avec un idFic différent (add_2)
                - Le fichier doit passer sans problème
                - Les avions qui ne sont pas déjà présents sont rajoutés
                    - Ceux déjà présent ne sont pas impactés
- Ajout des fichiers non valides dans l'ordre de son choix (aucune importance) :
    - non_valide_Latin.xml
    - non_valide_Blank.xml
    - non_valide_add_structure.xml
    - non_valide_add_date.xml
    - non_valide_add_Checksum.xml
    - non_valide_add_1000carac.xml
    - non_valide_add_1carac.xml

FICHIERS VALIDES

valide_add.xml

- Fichier de test valide - ajout
    - Doit passer sans problème s'il n'est pas déjà en mémoire

valide_del.xml

- Fichier de test valide - delete
    - Doit passer sans problème s'il n'est pas déjà en mémoire

valide_update.xml

- Fichier de test valide - update

    - Doit passer sans problème s'il n'est pas déjà en mémoire

valide_add_idFic.xml

- 2ème fichier de test valide - ajout - idFic modifié pour permettre second ajout
    - Doit passer sans problème s'il n'est pas déjà en mémoire
    - Sert à prouver que l'identifiant unique du fichier "idFic" est bien contrôlé
    - Contenu identique au fichier "valide_add.xml" en dehors de l'attribut idFic

FICHIERS NON VALIDES

non_valide_Latin.xml

- Fichier de test non valide - présence de caractère non ASCII
    - Soukhoï Su-27 non accepté
    - Doit rejeter le fichier et retourner une erreur de type de caractère

non_valide_Blank.xml

- Fichier de test non valide - structure du fichier qui ne respecte pas le validator.xsd
    - Doit rejeter le fichier et retourner une erreur de fichier invalide car complètement vide

non_valide_add_structure.xml

- Fichier de test non valide - structure du fichier qui ne respecte pas le validator.xsd
    - Doit rejeter le fichier et retourner une erreur de fichier invalide

non_valide_add_date.xml

- Fichier de test non valide - date déclarée supérieure à 3 mois (par rapport au 2019-11-28)
    - Doit rejeter le fichier et retourner une erreur de date supérieure à 3 mois

non_valide_add_Checksum.xml

- Fichier de test non valide - checksum déclaré mis a 1 au mieu de 5
    - Doit rejeter le fichier et retourner une erreur de checksum

non_valide_add_1000carac.xml

- Fichier de test non valide - champs d'avion qui a plus de 1000 caractères
    - Doit rejeter le fichier et retourner une erreur du nombre de caractères égal supérieur à 1000

- (A ...) 1006 caractères pour être précis
- Le fichier fait également **plus de 5ko**, l'erreur doit également être signalée

non_valide_add_1carac.xml

- Fichier de test non valide - champs d'avion qui n'a qu'un seul caractère (A)
  - Doit rejeter le fichier et retourner une erreur du nombre de caractères égal à 1

6 Perspectives

Cette première version de notre application fait bien l'essentiel de ce qu'on lui demande de faire, c'est une version émergée par un effort collectif, un ensemble de ressources et une contrainte de temps. Ce résultat ne nous empêche pas de regarder vers le futur sur des améliorations qui peuvent être appliqué dans le but de rendre l'application plus performante :

- Natural Language Processing (NLP) : Au lieu de se limiter uniquement sur des fichiers XML, on pourra intégrer une IA qui s'occupera de traiter les différents types d'informations : images, sons, vidéos, etc.
- Détection simultanée de plusieurs fichiers XML : Pour l'instant, le scrutateur ne détecte qu'un seul fichier à la fois. Mettre en place un mécanisme qui s'occupera de traiter plusieurs fichiers à la fois pourra aider à gagner du temps pour l'utilisateur.
- Un fichier XML avec plusieurs actions : Pour l'instant, le validateur limite les fichiers XML d'avoir une unique balise d'action. L'idée est de modifier les contraintes pour laisser passer les fichiers XML contenant plusieurs balises d'action.

7 Conclusion et bilan du projet

Le lancement du projet était long vu notre focalisation sur la structure des fichiers XML, mais on a rapidement rattrapé ce retard, on se concentrant sur le programme qui traitera ces fichiers XML et qui constituera le cœur du projet. Le début de la construction du programme avec ses différents modules, nous a permis d'avoir une vue claire de la structuration de nos fichiers XML. Une fois qu'on a fixé ces détails, le restant du projet est passé de façon fluide.

Scruter, valider, parser, stocker, écrire un log et afficher les données, ce projet nous a appris à manipuler des technologies, qu'on n'a jamais utilisé auparavant, dans la réalisation d'un projet issu d'une idée voulant s'adapter aux évolutions des méthodes de travail dans les entreprises.