BORDEAUX INP Enseirb-Matmeca

Rapport de stage

Création et Déploiement d'un Entrepôt de Fichiers

Pavillon de la Mutualité
Mutualité Française Gironde

Membre de vyv3

Lucas Senis
Informatique 2e année

Résumé

Mon stage s'est déroulé dans le service informatique au sein du service informatique du Pavillon de la Mutualité - Mutualité Française Gironde, membre du groupe VYV[3], dans les locaux de la Clinique Mutualiste Arnaud Duben, à Pessac. Le sujet du stage comportait deux parties :

- la mise en place d'un système de stockage de fichier centralisé pour y déposer les fichiers de la Clinique Mutualiste
- l'utilisation de ce système pour produire des statistiques concernant la Clinique Mutualiste, par le biais des fichiers contenus dans ce système de stockage

Le système de stockage de fichiers, nommé entrepôt, devait être accessible au moins en lecture par une interface web, et en lecture/écriture par ligne de commande. De plus, le langage majoritairement utilisé à la Clinique Mutualiste pour le traitement de données étant le R, l'entrepôt devait également être accessible en lecture avec une bibliothèque R.

Lors de l'ajout d'un fichier à l'entrepôt, un identifiant unique lui est attribué. Pour pouvoir manipuler les fichiers efficacement, on associe à chaque fichier de contenu un autre fichier nommé manifeste qui contient des métadonnées relatives au fichier que l'on ajoute à l'entrepôt (par exemple identifiant, auteur, type, date de création...). Afin de pouvoir déplacer les fichiers plus simplement en cas de transfert d'entrepôt à entrepôt, un autre format de fichier a été défini : le bundle. Celui-ci contient les informations du manifeste mais aussi le fichier de contenu.

Pour stocker les fichiers et leur manifeste, l'entrepôt s'appuie principalement sur une arborescence de fichiers basée sur l'identifiant. Par exemple, un fichier d'identifiant `0x12345678` sera situé (ainsi que son manifeste) dans le dossier `files/0x/1/2/3/4/5/6/7/8/`. Cependant, rechercher un fichier dont on ne connaît pas l'identifiant peut être très coûteux (il faudrait parcourir tous les manifestes un à un jusqu'à trouver celui qu'on recherche). C'est pourquoi les manifestes sont stockés dans un cache sous la forme d'une base de données SQLite en plus de l'arborescence de fichiers. Ainsi, on peut obtenir l'identifiant du fichier souhaité en faisant une recherche dans la base de données, ce qui permet de connaître l'emplacement de ce fichier. Un système de gestion de versions a également été implémenté pour pouvoir retracer l'évolution d'un fichier.

L'entrepôt est programmé en Python et définit les trois interfaces suivantes :

- interface de requête : accès aux fichiers de l'entrepôt en lecture
- interface d'intégration : accès aux fichiers de l'entrepôt en écriture, possibilité d'ajouter des fichiers et d'en supprimer
- interface d'administration : accès aux composantes internes de l'entrepôt, manipulation de l'arborescence de fichiers et du cache

L'interface par ligne de commande est la seule à utiliser ces trois interfaces, l'interface web et la bibliothèque R n'ont accès qu'aux fonctions de requête et d'intégration (bien que la bibliothèque R n'utilise pas de fonction d'intégration pour le moment).

L'interface web s'appuie sur une API de type REST définie en Python et accessible depuis le réseau de la Clinique Mutualiste. Afin d'être utilisable depuis plusieurs machines, la bibliothèque R s'appuie également sur cette API.

Des conventions de codes d'erreur ont été établies pour l'interface par ligne de commande (valeur de retour de la commande) ainsi que pour l'API REST (code d'erreur HTTP).

À la fin du stage, je me suis chargé du déploiement de l'entrepôt et de l'API associée sur le réseau de la Clinique Mutualiste de Pessac.

Sommaire

Introduction

J'ai effectué mon stage au sein du Pavillon de la Mutualité - Mutualité Française Gironde, dans le service informatique. Ce stage s'est déroulé à la Clinique Mutualiste Arnaud Duben à Pessac.

La Clinique Mutualiste Arnaud Duben est un établissement privée à but non lucratif offrant des services de soins dans tout le département. Les Cliniques Mutualistes situées à Pessac et à Lesparre-Médoc font partie de cette entreprise, qui est elle-même membre du groupe VYV^3. VYV^3 est un groupe rassemblant plusieurs entreprises dans le domaine médical et totalisant environ 1700 d'établissement de soins et d'accompagnement.

1 Mission technique et solutions

Les objectifs de ce stage étaient divisés en deux parties.

La première partie du stage consistait à concevoir et réaliser un entrepôt de données capable de stocker des fichiers.

La deuxième partie du stage consistait à rédiger des scripts de traitement des données pour le département d'information médicale afin de l'aider à produire diverses statistiques. Cette étape réalisait la recette fonctionnelle finale du projet.

1.1 Organisation modulaire du projet

En début de stage, nous avons pu établir un schéma modulaire du projet avec le client. Ce schéma a évolué au cours du développement, selon ce qui nous a semblé plus pertinent au regard de l'implémentation. Dans ce schéma, les formes grises représentent les fonctionnalités déjà existantes sur lesquelles s'appuie le projet, et les formes jaunes représentent les modules constituant la bibliothèque Python, que l'on nommera *entrepôt*. Une flèche noire pleine d'un module vers un autre signifie que le premier module s'appuie sur le deuxième.

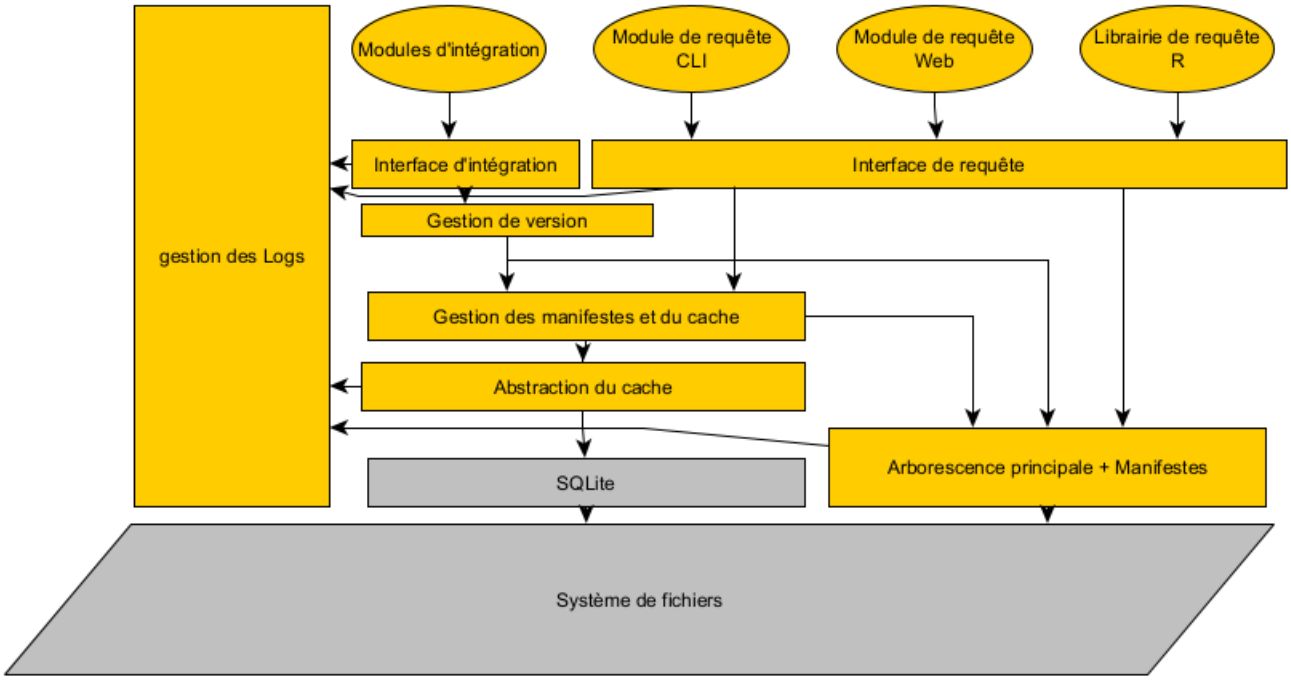


Figure 1: Schéma modulaire initial du projet

La solution retenue pour stocker les fichiers et y accéder efficacement est de stocker les fichiers en eux-mêmes sur le disque dans une arborescence de dossiers nommée *arborescence principale*. À chacun de ces fichiers est associée une liste de métadonnées permettant de manipuler ce fichier. Ces métadonnées constituent le *manifeste* du fichier, et sont stockées en deux endroits : dans l'arborescence principale sous la forme d'un fichier `.MANIFEST` situé au même emplacement que le fichier désigné, et dans une base de données SQLite (que l'on nommera *cache*) pour pouvoir effectuer une recherche plus efficacement qu'en parcourant toute l'arborescence principale.

Le *gestionnaire de manifestes* est le module central de l'entrepôt. Il définit une interface pour que les couches supérieures du projet puissent accéder aux fichiers et manifestes dans l'arborescence principale ou le cache comme s'ils n'étaient stockés qu'à un endroit. C'est dans ce module qu'est assurée la cohérence entre l'arborescence principale et le cache. La *gestion de version* est une surcouche du gestionnaire de manifestes permettant d'associer un nouveau fichier à un fichier déjà existant pour pouvoir retrouver les versions antérieures d'un fichier lors d'une requête.

Afin de garder une trace des différentes opérations effectuées, nous avons également prévu un système de logs sous la forme d'une interface permettant à la quasi-totalité des modules de l'entrepôt d'écrire dans un fichier commun.

J'ai donc commencé à programmer ce projet en suivant cette architecture. Cependant, la gestion des versions représentant une très petite part du projet, j'ai suggéré de l'inclure dans le gestionnaire de manifestes. De plus, l'organisation des modules utilisant l'entrepôt était assez vague dans le schéma modulaire initial. Nous avons donc précisé celle-ci et aboutit au schéma modulaire suivant. Dans ce schéma, un ovale représente un module utilisant l'entrepôt.

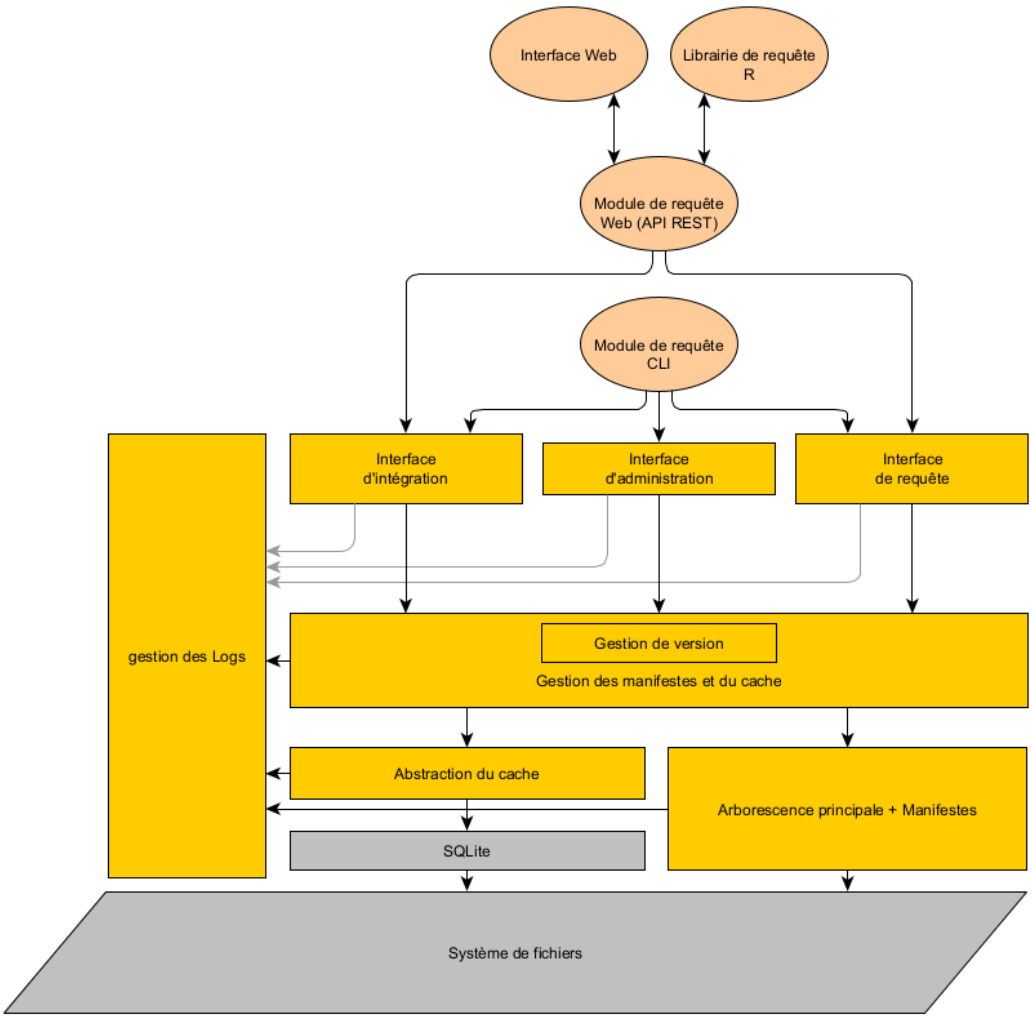


Figure 2: Schéma modulaire final du projet

On constate sur ce schéma la présence de trois interfaces. L'interface de requête permet d'accéder aux fichiers et manifestes en lecture et l'interface d'intégration permet d'ajouter, supprimer, modifier ou remplacer des fichiers ou des manifestes. Une interface d'administration a également été ajoutée pour effectuer des opérations privées sur l'entrepôt, comme par exemple resynchroniser le cache avec l'arborescence principale en cas de problème avec la base de données SQLite. Un autre module viendra s'y ajouter mais était hors périmètre du stage : une gestion de droits d'utilisateurs et de jetons, en particulier pour l'accès distant.

Deux modules ont été programmés pour interagir directement avec l'entrepôt : un module en ligne de commandes, ainsi qu'une API de type REST pour accéder à l'entrepôt depuis le réseau de la clinique. Cette API est utilisée par une interface web et par une bibliothèque R afin de faciliter le traitement des données.

1.2 Manipulation des manifestes

Nous avons vu que les informations concernant un fichier sont stockées dans le manifeste correspondant. Ce dernier peut exister sous deux formes différentes :

- un fichier `.MANIFEST`

  Il s'agit d'un fichier contenant du texte, et dont chaque ligne respecte le format suivant :

```
<propriete> = <valeur>
```

Dans l'exemple ci-dessus, `<propriete>` est une chaîne de caractères non vide et ne contenant pas la séquence `" = "`. `<valeur>` est une chaîne de caractères potentiellement vide mais ne contenant pas la séquence `" = "`.

- un objet Python de la classe `Manifest`

Puisqu'un manifeste est associé à un fichier, ils sont manipulés ensemble dans le code. C'est dans ce but qu'a été créée la classe `Entity`. Une *entité* est un objet contenant un manifeste ainsi qu'un fichier (ou les informations nécessaires pour y accéder).

L'équivalent d'une entité du point de vue de l'arborescence de fichiers est un *bundle*. Un bundle est un format de fichier propre à l'entrepôt qui contient les informations du manifeste et le contenu du fichier associé au manifeste. Ce format a été conçu pour pouvoir déplacer les fichiers plus simplement. Un fichier bundle est la concaténation du manifeste et du fichier de contenu, séparés par un saut de ligne :

```
<propriete1> = <valeur1>
<propriete2> = <valeur2>

<contenu>
```

Les contraintes sur les couples (`<propriete>`,`<valeur>`) sont les mêmes que celles énoncées précédemment, et il n'y a aucune contrainte sur `<contenu>`, qui peut être textuel ou composé de données binaires.

2 Réalisations

2.1 Structure d'un manifeste

Un objet de la classe `Manifest` contient un attribut `corrupted` et un attribut `properties`. `corrupted` indique s'il y a eu une erreur lors de la manipulation du manifeste. Les erreurs surviennent généralement lors de l'extraction d'un fichier si celui-ci ne respecte pas le format indiqué en Sous-section 1.2. L'attribut `properties` contient un dictionnaire dont les couples (clé,valeur) correspondent aux couples (`<propriete>`,`<valeur>`), pour reprendre l'exemple précédent.

En premier lieu, nous avons établi une liste des champs nécessaires dans le manifeste d'un fichier :

- `id`

  Un identifiant unique est attribué à chaque fichier lors de son ajout à l'entrepôt.

- `file.originalName`, `file.type` et `file.author`

  Ces champs contiennent respectivement le nom du fichier, son type et son auteur. Ces informations doivent être précisées par l'utilisateur lors de l'ajout d'un fichier.

- `file.creationDate`

  Ce champ désigne la date de création du fichier et peut être précisée par l'utilisateur lors de l'ajout d'un fichier. Si il n'est pas précisé, le champ prend pour valeur la date d'ajout du fichier à l'entrepôt.

- *file.isMostRecent (booléen)*

  Ce champ a été retiré par la suite car cette information pouvait être retrouvée facilement en faisant une recherche dans le cache sur le champ `file.previousVersion`.

- `file.previousVersion`

  Ce champ contient l'identifiant d'un autre fichier présent dans l'entrepôt. Il ne peut pas être modifié directement par l'utilisateur lors d'un ajout, il faut utiliser une fonction différente de l'ajout simple d'un fichier pour que le gestionnaire de manifestes définisse la valeur de ce champ dans le cadre de la gestion de version.

- `file.deleted` (booléen)

  Afin de ne pas perdre des fichiers, ceux-ci ne sont pas supprimés directement mais plutôt "marqués" comme supprimés avec le champ `file.deleted`.

- `file.comment`

  Ce champ contient du texte relatif au fichier et n'est pas interprété par l'entrepôt.

- *file.signature* (`file.signature.type` et `file.signature.value`)

  Le champ contenait initialement un hash md5 du fichier afin de vérifier qu'un manifeste n'a pas été associé au mauvais fichier. Par la suite, il a été scindé en deux champs : `file.signature.type` qui contient le nom de la méthode de hachage utilisée et `file.signature.value` qui contient le hash en lui-même. Cela permet d'utiliser d'autres méthodes que md5.

- `file.error`

  Contient un message d'erreur si le fichier ou le manifeste n'est pas cohérent. Il est impossible d'ajouter un fichier avec le champ file.error.

- `content.type.name` et `content.type.version`

  Ces champs contiennent le type interne d'un fichier, à ne pas confondre avec `file.type` : un fichier texte (de type externe `txt`) peut contenir des informations sous différents formats. Ces formats sont précisés dans le type interne du fichier `content.type.name`. Le champ `content.type.version` désigne les différentes normes d'un type, car celles-ci peuvent évoluer avec le temps.

- `content.year`, `content.month` et `content.status`

  Ce sont des propriétés liées au fichiers produits pour le PMSI (Programme de Médicalisation des Systèmes d'Information) qui se déclinent par année, mois et version.

2.2 Arborescence de l'entrepôt

L'ensemble des fichiers manipulés par l'entrepôt se situent dans le dossier `files`. Ce dossier principal contient trois sous-dossiers : `0x`, `sqlite_database` et `tmp`.

Le dossier `files/0x` est l'arborescence principale. Lorsqu'un fichier est ajouté à l'entrepôt, un identifiant aléatoire lui est automatiquement associé. Cet identifiant se présente sous la forme d'un entier à 8 chiffres en base hexadécimale. Cette limite est arbitraire, il est envisageable d'élargir l'alphabet à tout caractère du système de fichiers en modifiant un paramètre de l'arborescence principale. Il est également possible de définir une arborescence principale avec des identifiants plus longs. Supposons par exemple que l'on souhaite ajouter un fichier `data.txt` à l'entrepôt. Un identifiant aléatoire lui est associé, par exemple `0xdeadbeef`. Ce fichier et son manifeste sont alors stockés à l'emplacement `files/0x/d/e/a/d/b/e/e/f/` respectivement sous le nom `0xdeadbeef` et `0xdeadbeef.MANIFEST`. Lors du déploiement de l'entrepôt, le dossier `files/0x/` est vide car il n'est pas pertinent de créer plusieurs milliards ($\sum_{i=1}^{8} 16^i \approx 4 \times 10^9$) de dossiers dont la plupart ne seront jamais utilisés. Ainsi, le chemin vers l'emplacement d'un fichier est créé par le module responsable de l'arborescence principale lors de l'ajout de ce fichier.

Le dossier `files/sqlite_database/` contient le cache de manifestes sous la forme d'un fichier `data.db` unique.

Enfin, le dossier `files/tmp/` contient des fichiers temporaires créés lors de certains ajouts de fichiers.

Afin d'assurer l'intégrité de l'entrepôt, il s'est avéré nécessaire d'ajouter une composante de sécurité en restreignant les droits d'accès aux différents fichiers du projet. En effet, il ne doit pas être possible d'accéder aux fichiers de l'entrepôt directement depuis l'arborescence de fichiers. Pour cela, un utilisateur nommé `entrepot` a été créé. Cet utilisateur a les droits de lecture et d'écriture (et d'exécution) sur tout le dossier `files` et ses sous-dossiers, et en est propriétaire. Aucun autre utilisateur ne peut modifier ou même accéder au contenu de `files` (mode `rwx------`). Pour interagir avec l'entrepôt, l'utilisateur est donc contraint d'utiliser les interfaces à sa disposition : une commande dédiée (voir Sous-section 2.5) ou une interface de type REST (voir Sous-section 2.6).

2.3 Gestion du cache

Le cache de manifestes est stocké dans le fichier `files/sqlite_database/data.db` sous la forme d'une base de données SQLite. Elle est manipulée grâce à la bibliothèque Python `sqlite3`. Dans un souci de modularité, le module de gestion du cache est scindé en deux parties. Le fichier source `sqliteAbstraction.py` définit un ensemble de fonctions pour permettre aux autres modules d'interagir avec la base de données sans utiliser directement `sqlite3`. En utilisant ces fonctions, le fichier `manifestCache.py` définit une interface basique pour le gestionnaire de cache : `get`, `set`, `remove` et `clean`. La méthode `clean` est utilisée lorsqu'il faut resynchroniser le cache avec l'arborescence principale.

2.4 Contraintes lors de l'ajout de fichiers

Certaines vérifications sont systématiquement effectuées sur le fichier et le manifeste :

- format incorrect

  Si le manifeste fourni en entrée (qu'il soit sous la forme d'un fichier séparé ou d'un bundle) ne respecte pas le format établi par convention, alors l'objet qui en résulte est qualifié de corrompu et l'ajout du fichier sera refusé.

  Lors d'un ajout par bundle, le programme vérifie que le fichier contient bien les parties correspondant au manifeste et au contenu. Si ce n'est pas le cas, l'entité qui en résulte est qualifiée de corrompue et ne sera pas ajoutée à l'entrepôt.

- champs obligatoires et interdits

  Le manifeste de tout fichier doit impérativement contenir les champs `file.originalName`, `file.type` et `file.author`. Les champs interdits sont `file.deleted` et `file.error`, ainsi que `file.previousVersion` car ce champ ne peut être modifié que par l'entrepôt lors de la gestion des versions.

- valeurs de champs

  Les champs `id` et `file.creationDate` ont des formats particuliers qui sont vérifiés à l'aide d'expressions régulières. C'est également le cas pour le champ `file.previousVersion` (le même format que `id`, mais ce dernier est interdit lors de l'ajout d'un fichier donc sa vérification n'est pas nécessaire).

- signature

  Si le manifeste fourni en entrée contient le champ `file.signature.value`, alors la signature du fichier de contenu est recalculée pour vérifier que celle-ci correspond bien. Cette vérification est effectuée séparément car c'est la seule qui nécessite une lecture du fichier de contenu. De plus, elle s'appuie sur la valeur de deux champs : `file.signature.name` et `file.signature.value`.

- duplication de fichiers

  Même si un fichier répond à toutes les exigences nécessaires pour être ajouté à l'entrepôt, une dernière vérification est faite pour savoir si un fichier identique est déjà stocké. Pour cela, le programme recherche la signature du fichier dans le cache. Par défaut, l'ajout d'un fichier déjà présent est autorisé, l'utilisateur est simplement informé de la duplication.

2.5 Interface par ligne de commande

Pour interagir avec l'entrepôt depuis un terminal, il faut utiliser l'exécutable `src/exec/entrepot` qui est un programme Python. Celui-ci contient en réalité plusieurs commandes, dont voici la liste :

1. `entrepot help`

   Sert de manuel pour l'utilisation de `entrepot`, peut donner la liste des commandes où des informations plus précises sur les commandes. L'aide disponible pour chaque fonction est stockée dans la variable `__doc__`, qui permet d'accéder au docstring d'une fonction depuis le programme.

2. `entrepot add`

   Permet d'ajouter un nouveau fichier à l'entrepot. Si un fichier identique est déjà présent, l'utilisateur en est informé afin d'éviter les doublons.

3. `entrepot get`

   Affiche un fichier, un manifeste ou les deux sur la sortie standard.

4. `entrepot get-property`

   Affiche la valeur d'une propriété dans le manifeste d'un fichier sur la sortie standard.

5. `entrepot set-property`

   Modifie la valeur d'une propriété dans le manifeste d'un fichier. Toutes les propriétés ne sont pas modifiables.

6. `entrepot update`

   Ajoute une nouvelle version d'un fichier à l'entrepot. Cette commande est similaire à `entrepot add` mais implique la gestion de versions.

7. `entrepot replace`

   Remplace un fichier déjà existant par un autre qui prend le même identifiant. Cette commande n'a vocation à être utilisée que pour corriger des erreurs d'ajout.

8. `entrepot delete`

   Marque un fichier comme supprimé.

9. `entrepot find`

   Permet de retrouver un manifeste en fonction de la valeur de ses champs.

10. `entrepot recover-manifests`

    Resynchronise le cache avec l'arborescence principale.

11. `entrepot status`

    Afficher des informations relatives au nombre de fichiers dans l'entrepôt.

Lors de l'ajout de fichiers par les commandes `entrepot add`, `entrepot update` et `entrepot replace`, il est possible d'effectuer cet ajout de différentes façons :

- chemin vers le fichier + chemin vers le manifeste
- chemin vers le fichier

  Dans ce cas, le programme cherche un fichier de même nom que le fichier à ajouter mais d'extension `.MANIFEST` dans le dossier du fichier à ajouter.

- chemin vers le manifeste + fichier écrit dans l'entrée standard
- chemin vers le bundle
- bundle écrit dans l'entrée standard

Lors d'un ajout par bundle, le fichier n'est pas stocké tel quel. En effet, le contenu et le manifeste sont stockés sous la forme de deux fichiers séparés comme cela aurait été le cas pour un ajout classique.

Lorsque l'on ajoute un fichier, celui-ci est copié dans le dossier `files/tmp` sous la forme d'un fichier temporaire ayant pour nom une suite aléatoire de 20 chiffres et pour extension `.tmp`.

De plus, il est possible d'empêcher l'ajout d'un fichier déjà présent dans l'entrepôt avec l'option `--check--duplicate`, que ce soit lors d'un ajout par `entrepot add`, `entrepot update` ou `entrepot replace`.

Comme expliqué plus haut, l'utilisateur n'a pas les droits de lecture et d'écriture sur les fichiers de l'entrepôt. Pour qu'il puisse y accéder en passant par `entrepot`, j'ai souhaité activer le SUID sur cet exécutable. Initialement, cet exécutable était un simple script précédé de `#!/usr/bin/python3`. Cependant, le SUID n'étant pas applicable aux fichiers non compilés, j'ai compilé le programme en un nouvel exécutable avec `pyinstaller`. J'ai ainsi pu lui appliquer le SUID (mode `--s--x--x`). De cette manière, n'importe quel utilisateur sur la machine peut utiliser la commande `entrepot` et celle-ci sera exécutée avec les droits de l'utilisateur `entrepot`, ce qui lui permettra d'interagir avec les données contenues dans l'entrepôt.

2.6 Interface par API REST

L'API a été développée à l'aide de la bibliothèque python `fastAPI`. Afin d'être disponible en permanence, l'API REST tourne sur un serveur `uvicorn`. De plus, j'ai rédigé un script appelé à intervalle régulier grâce à `crontab` pour relancer l'applicatif en cas de panne (limitation technique temporaire liée à la politique de sécurité de l'établissement. À terme, il conviendrait de privilégier le déploiement en tant que service). J'ai choisi `fastAPI` car cette bibliothèque effectue une vérification automatique du type des données lors de la réception de requêtes et de l'envoi de réponses. Cette vérification est simple à implémenter car elle s'appuie sur les annotations de types en Python. De plus, `fastAPI` génère une documentation automatique à l'adresse `/docs` :

Figure 3: Liste des requêtes définies générée par fastAPI

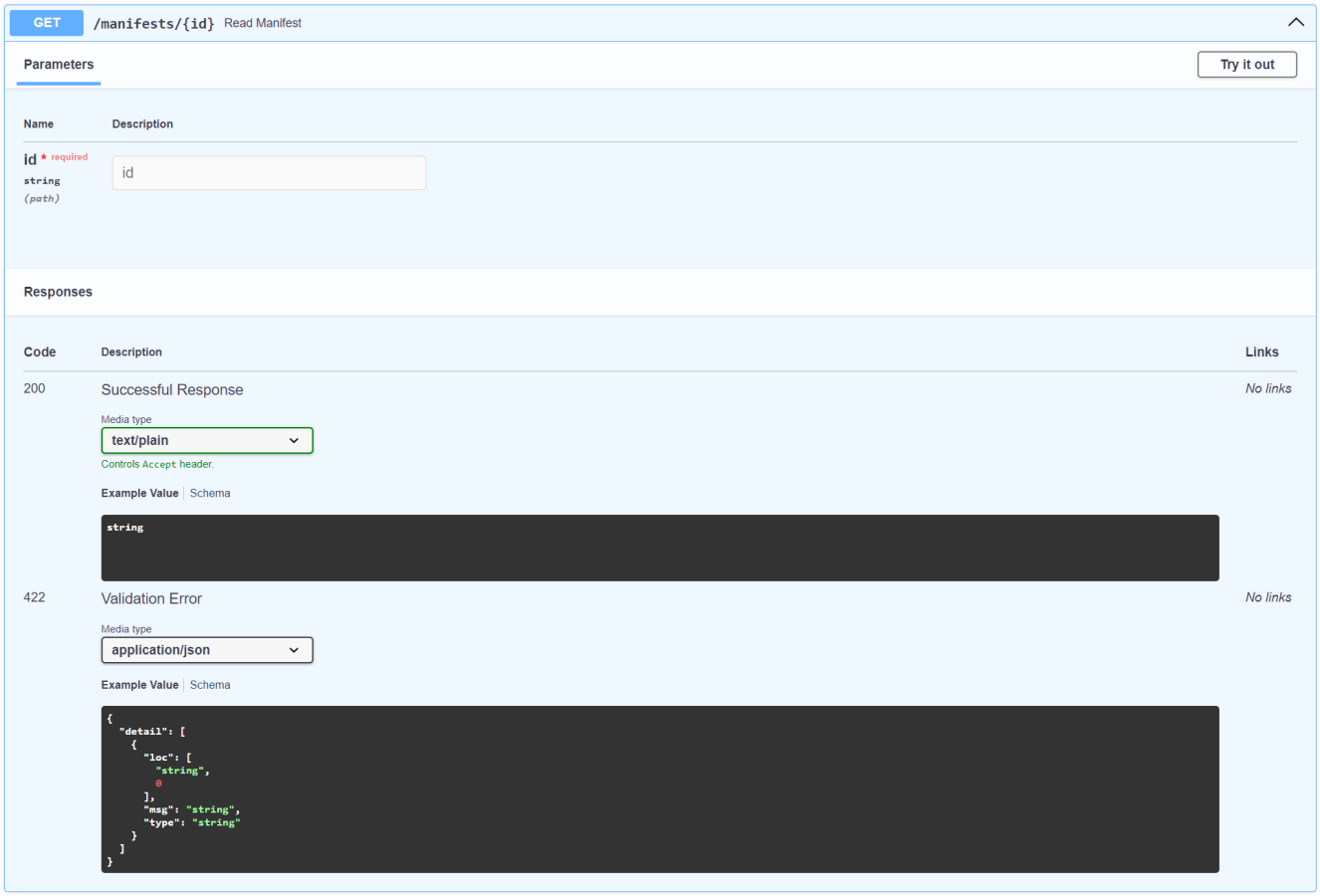


Figure 4: Exemple de documentation générée pour une requête

L'API REST n'ayant pas accès à l'interface d'administration (voir Figure 2), les fonctions définies s'appuient principalement sur les interfaces de requête et d'intégration :

- GET : `/isonline`

  Cette requête est différente des autres car elle n'a pas pour but d'interagir avec l'entrepôt. En effet, quand le serveur reçoit une requête `/isonline`, il répond systématiquement `true`. Dans le cadre de l'interface web, cette requête est envoyée à l'ouverture de la page pour que l'utilisateur soit informé si la connexion a échoué (voir Figure 8b).

- GET : `/manifests/{id}`, `/files/{id}` et `/bundles/{id}`

  Les requêtes `/manifests/{id}` et `/files/{id}` sont l'équivalent de la commande `entrepot get` évoquée plus haut. Elles retournent respectivement le manifeste et le fichier d'identifiant `id` sous un format binaire.

  La requête `/bundles/{id}` génère le fichier bundle correspondant à l'identifiant `id` et retourne également celui-ci sous forme de `bytes`.

- GET : `/bundles/{id}/json` et `/manifests/{id}/json`

  Ces deux requêtes retournent respectivement le manifeste et le bundle sous forme d'un dictionnaire `json` plutôt que des `bytes`. Elles sont utilisées lorsque l'interface web manipule les propriétés d'un manifeste. Les requêtes `/bundles/{id}` et `/manifests/{id}` sont plutôt utilisées dans le cas du téléchargement de fichiers.

- GET : `/find/{mode}/{values}`

Cette requête permet de faire une recherche par propriétés dans l'ensemble des manifestes de l'entrepôt. `values` contient les conditions que le manifeste doit respecter selon un certain format. Par exemple, le motif `file.author=un_auteur&file.type=txt` correspond à l'ensemble des fichiers de type `txt` rédigés par `un_auteur`. Si l'argument `values` vaut `all`, alors tous les manifestes correspondent au motif.

L'argument `mode` peut prendre deux valeurs : `include` pour obtenir les manifestes correspondant au motif recherché et `exclude` pour obtenir ceux qui ne correspondent pas au motif (voir Figure 9).

- POST : `/add/file_and_manifest`

  Cette requête permet d'ajouter un fichier et son manifeste à l'entrepôt en transmettant les deux éléments sous la forme de fichiers.

- POST : `/add/manifest` et `/add/file`

  Ces deux requêtes fonctionnent ensemble et sont indissociables. Elles sont utilisées lorsque le manifeste du fichier que l'on souhaite ajouter a été rempli manuellement via un formulaire.

  Une première requête HTTP de type `application/json` est envoyée pour transmettre le manifeste, et une deuxième requête de type `multipart/form-data` est envoyée avec le fichier de contenu. Afin de faire le lien entre ces deux requêtes, la première génère un identifiant temporaire et y associe le manifeste dans un dictionnaire. Cet identifiant est transmis à l'utilisateur dans la réponse de la première requête, et il doit le renvoyer avec le fichier dans la deuxième requête. Ainsi, le serveur associe au fichier le manifeste ayant l'identifiant temporaire correspondant et transmet l'entité créée à l'interface d'intégration. Le manifeste est alors retiré du dictionnaire dans lequel il était stocké et l'identifiant temporaire peut être réutilisé.

  Cette séparation permet d'assurer que le manifeste fourni est au bon format avant d'autoriser la transmission du fichier de contenu, qui peut être coûteuse en terme de transmission et de traitement.

- POST : `/add/bundle`

  Avec cette requête, l'utilisateur peut ajouter un fichier et son manifeste à l'entrepôt en un seul fichier bundle.

2.7 Bibliothèque R

Initialement, l'API avait pour seule vocation d'être interrogée par l'interface web. Finalement, elle a aussi servi de moyen d'accès à l'entrepôt pour l'interface en R. Cette interface a une vocation d'outil pour réaliser des statistiques sur les fichiers de l'entrepôt. La plupart des fonctions définies manipulent des `data.frame`, car c'est le principal format utilisé pour manipuler des données en R au département d'information médicale. Diverses fonctions ont donc dû être implémentées :

- `entrepot.ping`

  De manière similaire à l'interface web, cette fonction envoie une requête `/isonline` à l'entrepôt pour vérifier que celui-ci est bien accessible. Si ce n'est pas le cas, une erreur est générée pour avertir l'utilisateur.

- `entrepot.init`

  La valeur de retour de cette fonction est une variable représentant l'entrepôt et contenant son URL. Bien qu'une URL par défaut ait été implémentée, il est possible d'en préciser une autre en argument de cette fonction dans le cas où l'entrepôt devrait être déplacé, ou encore si plusieurs entrepôts venaient à exister simultanément.

- `entrepot.getFileURL` et `entrepot.getManifestURL`

Ces fonctions n'envoient pas de requête à l'entrepôt, elles calculent simplement l'adresse à laquelle le fichier et le manifeste peuvent être téléchargés en fonction de l'URL de l'entrepôt et de l'identifiant fourni.

- `entrepot.getFile`

  `entrepot.getFile` retourne le contenu brut du fichier associé à un identifiant donné.

- `entrepot.getManifest` et `entrepot.get`

  Ces deux fonctions retournent un `data.frame`. Celui retourné par `entrepot.getManifest` contient les informations présentes dans le manifeste du fichier. `entrepot.get` retourne le même `data.frame` mais avec une colonne supplémentaire : l'URL à laquelle on peut télécharger le fichier de contenu.

- `entrepot.v.getFileURL`, `entrepot.v.getFile`, `entrepot.v.getManifestURL`, `entrepot.v.getManifest` et `entrepot.v.get`

  Ces fonctions sont similaires à celles évoquées précédemment, mais sont vectorisées. Ainsi, si on leur passe en argument un vecteur d'identifiants, elles retourneront un `data.frame` (usuellement représenté sous la forme d'un tableau en deux dimensions) dont chaque ligne correspondra à un identifiant.

```
e <- entrepot.init()

entrepot.get(e, "0x00000000")
//      id      | file.author | ...
//--------------|-------------|-----
//  0x00000000  |   auteur1   | ...

entrepot.get(e, "0x00000001")
//      id      | file.author | ...
//--------------|-------------|-----
//  0x00000001  |   auteur2   | ...

entrepot.v.get(e, c("0x00000000","0x00000001"))
//      id      | file.author | ...
//--------------|-------------|-----
//  0x00000000  |   auteur1   | ...
//  0x00000001  |   auteur2   | ...
```

Figure 5: Exemple d'utilisation de la vectorisation (fonction `entrepot.get`)

- `entrepot.find` et `entrepot.v.find`

  La fonction `entrepot.find` effectue une recherche de manifeste par propriétés similaire à celle effectuée par `entrepot find` dans l'interface par ligne de commande. Elle prend en argument une liste nommée.

  `entrepot.v.find` permet d'ajouter l'opérateur logique OU à la recherche de manifestes, qui jusqu'ici ne permettait de filtrer qu'avec l'opérateur ET. Pour cela, elle prend en argument un vecteur de listes nommées, correspondant chacune à une clause du OU.

- `entrepot.readFile`

  Cette fonction permet d'interpréter un fichier en fonction d'un format précisé et retourne son contenu sous la forme d'un `data.frame`.

- `entrepot.getRSSByDate` et `entrepot.v.getRSSByDate`

  Ces fonctions appliquent les fonctions "outils" développées précédemment. `entrepot.getRSSByDate` utilise `entrepot.find` sur les champs `file.type`, `content.date.month` et `content.date.year` pour trouver le fichier de type RSS demandé, puis `entrepot.readFile` pour l'extraire sous la forme d'un `data.frame`. `entrepot.v.getRSSByDate` est une vectorisation de cette fonction, similaire à celles vues précédemment.

2.8 Gestion d'erreurs

Si la commande `entrepot` n'est pas utilisée correctement, elle retournera un code d'erreur et affichera un message décrivant l'erreur. La signification des différents codes d'erreur définis est également répertoriée dans un fichier `error_codes.txt`. J'ai établi certaines règles générales pour l'attribution des codes d'erreur. Premièrement, les chiffres des centaines et dizaines indiquent quelle commande a été utilisée. Les commandes sont numérotés dans l'ordre présenté en Sous-section 2.5. Voici la signification des chiffres des unités :

- 0 : succès
- *1 : nombre d'arguments invalide
- *2 : option inconnue
- *3 : option invalide
- *4 : incompatibilité d'options
- *5 : argument invalide
- *6 : fichier non trouvé dans l'entrepôt
- *8 : erreur inconnue lors de l'interprétation des options
- *9 : erreur interne

Il est à noter que le succès d'une commande aboutira toujours à une valeur de retour de 0 (il n'y a pas de chiffre des centaines ou des dizaines pour indiquer la fonction utilisée). Afin d'éviter toute confusion, les multiples de 10 (autres que 0) ne sont pas utilisés en codes d'erreur.

Les codes d'erreurs terminant par 9 ont été utiles durant la phase de développement pour repérer des bugs. Dans la version finale du projet, un code d'erreur se terminant par 9 signifie généralement que la base de données est défectueuse et qu'il faut effectuer une opération `entrepot recover-manifests`.

Les codes d'erreurs terminant par 8 sont théoriquement impossibles, mais ont été définis dans l'éventualité où une erreur interne à la bibliothèque `getopt`, responsable des options, surviendrait.

Enfin, on remarque que la signification du chiffre des unités 7 n'est pas définie dans la liste précédente. En effet, le 7 a été conservé pour décrire des erreurs spécifiques à une commande et non généralisables à toutes :

- 27 (`entrepot add`) : fichier en entrée invalide
- 47 (`get-property`) : le fichier existe, mais n'a pas la propriété demandée
- 57 (`set-property`) : la propriété ne peut pas être modifiée/ajoutée
- 67 (`entrepot update`) : fichier en entrée invalide
- 77 (`entrepot replace`) : fichier en entrée invalide
- 87 (`entrepot delete`) : fichier déjà supprimé

De même, l'interface par API REST utilise les codes d'erreur HTTP pour informer l'utilisateur d'un problème avec la requête, la plus courante étant l'erreur 404. Cette erreur survient lorsque l'utilisateur demande un fichier qui n'est pas dans l'entrepôt, ou bien dans le cadre de la requête `/add/file` si l'identifiant temporaire ne correspond à aucun manifeste en mémoire. D'autres codes d'erreurs possibles sont 400 et 500 lorsque la requête en elle-même pose un problème ou si la base de données est corrompue. Dans tous les cas, l'exception HTTP contient un message pour informer l'utilisateur de l'origine du problème.

Puisque les erreurs générées par l'interface de requêtes arrivent principalement lorsqu'un fichier n'a simplement pas été trouvé, la gestion des erreurs avec cette interface est basique. Dans le cas de l'interface d'intégration, les erreurs concernent des fichiers entiers et peuvent donc être plus complexes. C'est pourquoi un `enum` a été ajouté à l'interface d'intégration pour informer les modules l'utilisant de manière plus précise. Il s'agit de la classe `IntegrationErrno`, visible en Figure 6.

Conclusion

Sur les deux parties du stage, la première a été intégralement réalisée : l'entrepôt de données en lui-même est fonctionnel et permet de stocker n'importe quel type de fichier. De plus, l'interface par ligne de commande est terminée et sécurisée. L'interface par API REST est également opérationnelle et le serveur est actif. L'interface web est également achevée en ce qui concerne la lecture et la recherche de fichiers. Pour l'ajout de fichiers via l'interface web, une interface générale (voir Figure 10c) permet d'ajouter n'importe quel type de fichiers. À la demande de mon tuteur, une interface spécialisée dans l'ajout de fichiers de type RSS (voir Figure 10f) a été développée. Cela permet d'assurer la présence de certains champs nécessaires dans un fichier RSS mais qui ne le seraient pas pour un autre type de fichiers. Cette interface destinée à un seul type de fichiers est facilement reproductible, car il suffit de changer une seule variable dans le code pour modifier la liste des propriétés obligatoires. Cette page a été demandée vers la fin du stage en plus du projet initial.

La deuxième partie du projet concernait la bibliothèque R, et celle-ci a pu être implémentée en accord avec les standards de la clinique, notamment la manipulation des données sous la forme de `data.frame`.

Le déploiement de l'ensemble de la biliothèque et plus particulièrement l'API REST sur un serveur a nécessité l'aide d'un administrateur système et la mise à disposition d'une machine au sein de la Clinique. De plus, l'utilisation d'un `crontab` était une solution astucieuse pour contourner les limitations techniques de la machine. Durant les derniers jours du stage, j'ai d'ailleurs rédigé un script externe au projet et associé à un `crontab` pour automatiser la récupération d'un fichier mis en ligne régulièrement.

Il aurait été possible d'implémenter une autre interface par ligne de commande, cette fois-ci pour interagir à distance avec l'entrepôt. En effet, l'interface actuelle n'est disponible que sur la machine sur laquelle l'entrepôt est stocké et il faut s'y connecter en `ssh` pour y accéder. Une interface par ligne de commande à distance aurait été possible en adaptant l'API REST pour émuler l'interface par ligne de commande locale. Bien que le développement d'une telle interface aurait été intéressant, cela aurait dépassé le cadre du stage et n'aurait pas été possible dans le temps imparti.

Bilan personnel

Au cours de ce stage, j'ai pu mettre en pratique les connaissances accumulées dans le cadre de mon cursus à l'ENSEIRB-Matméca, qu'il s'agisse de compétences en programmation (par exemple les langages Python, JavaScript et les scripts bash) ou de méthodes de travail comme l'approche orientée objet et la gestion de projet. En effet, l'actualisation du schéma modulaire (voir Figure 6) tout au long du projet m'a guidé dans mon organisation de travail, de même que le diagramme de Gantt (voir Figure 7). Ce stage a été très enrichissant et m'a également permis d'acquérir de nouvelles compétences, comme la gestion d'une API, l'automatisation d'un script à l'aide de `crontab` et surtout le langage R. De plus, le cadre de travail était agréable et les moyens mis à ma disposition (locaux, prêt d'un ordinateur portable sur toute la durée du stage et attribution d'un serveur pour y installer l'entrepôt) m'ont permis de travailler dans de bonnes conditions.

Durant ce stage, j'ai mené un projet complet, de la définition du cahier des charges à la mise en production. J'ai eu l'occasion de travailler en autonomie, en particulier lors des phases de documentation, d'auto-formation et de recherche de solutions. De plus, j'ai pu me familiariser avec les méthodes agiles en adoptant une démarche incrémentale et en faisant des points hebdomadaires avec le client.

Annexes

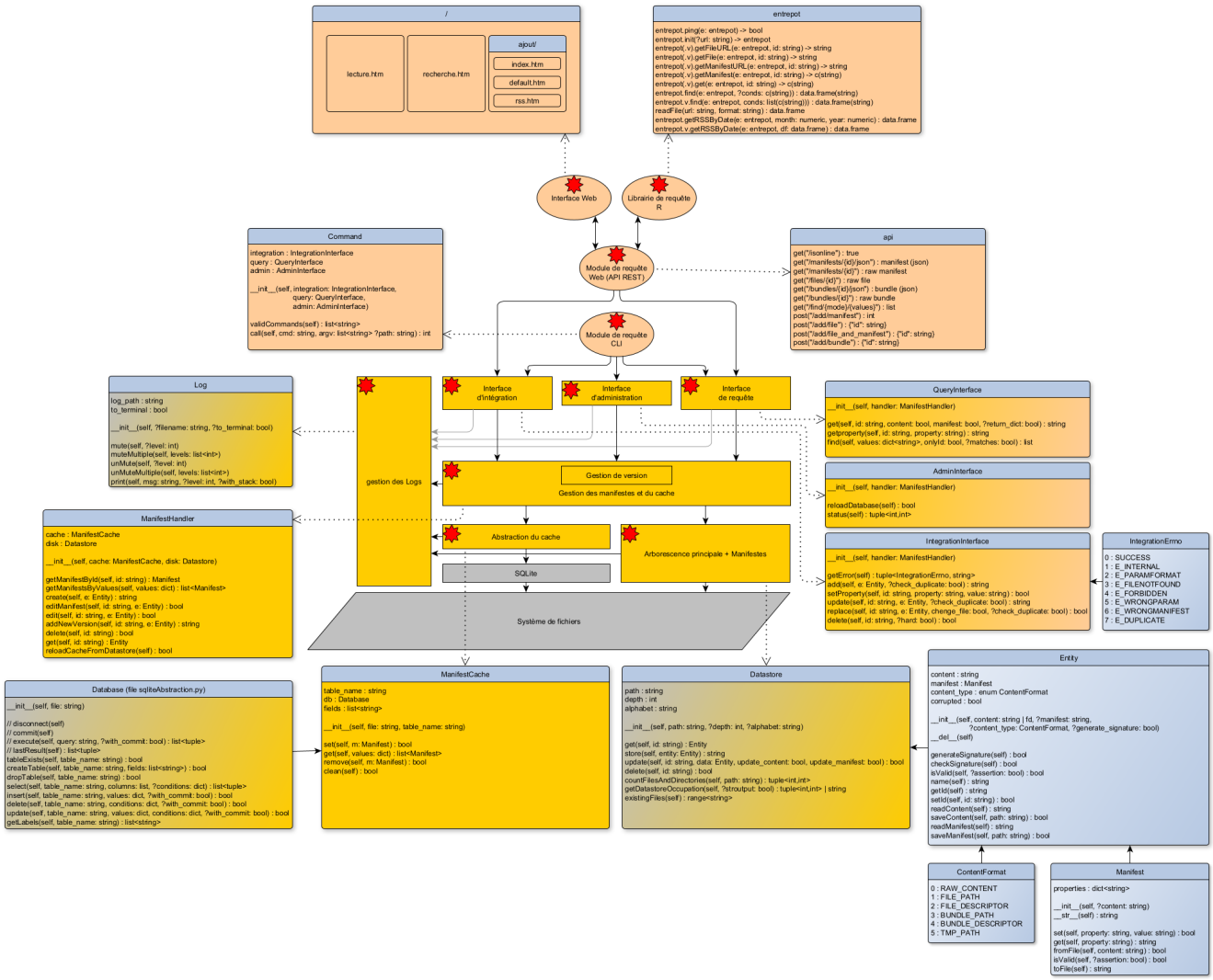


Figure 6: Schéma modulaire du projet avec les interfaces

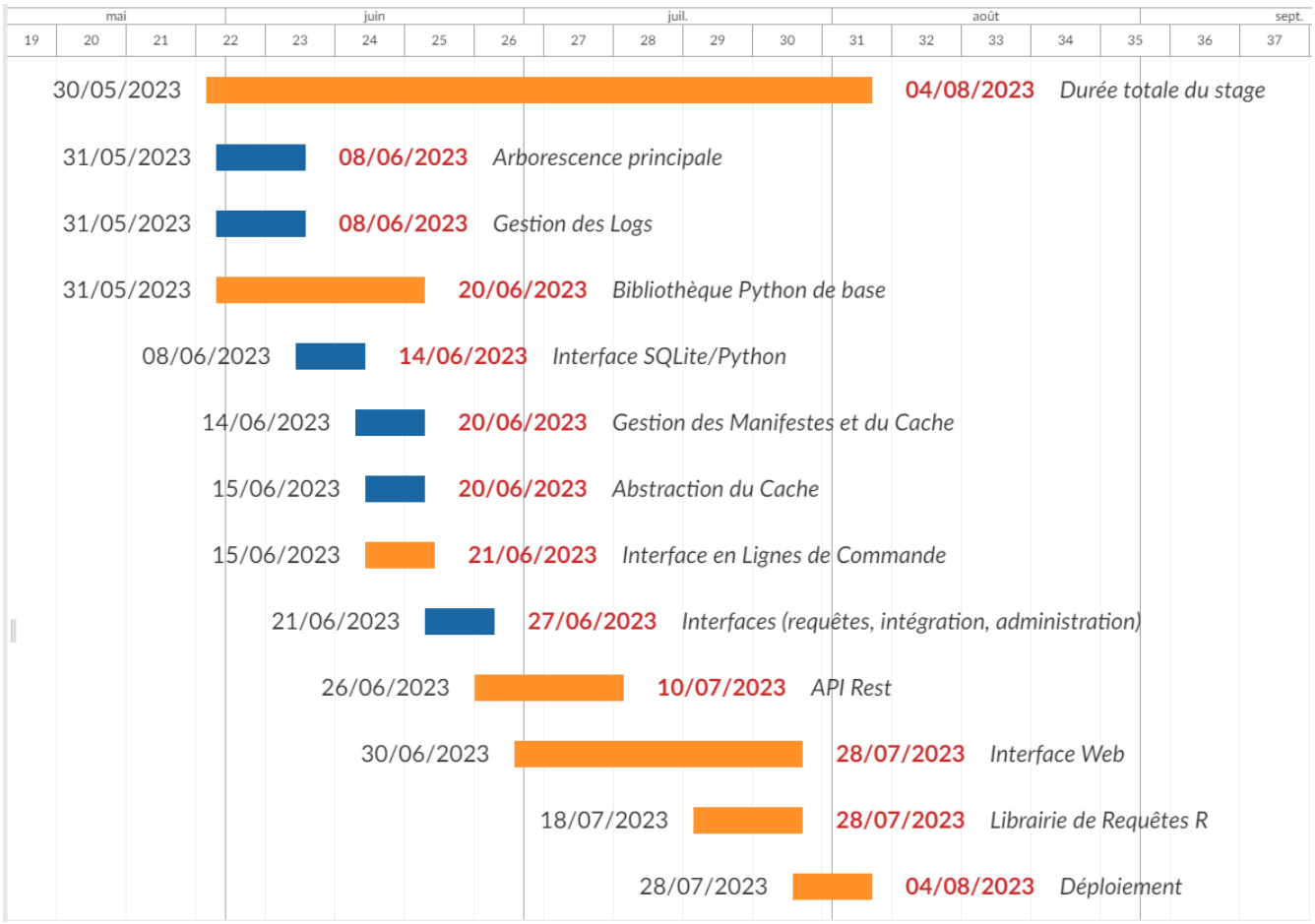
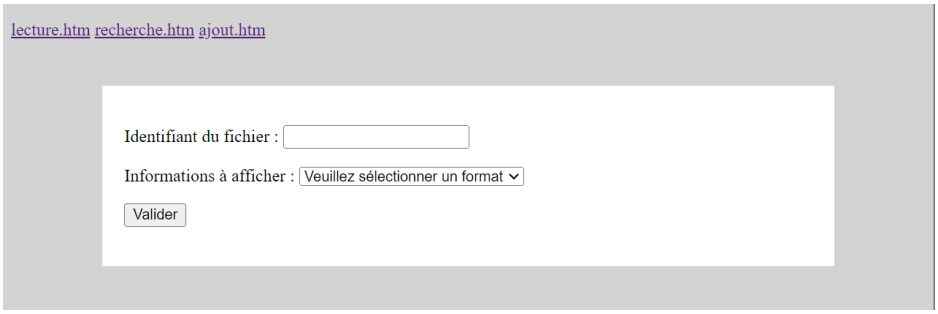
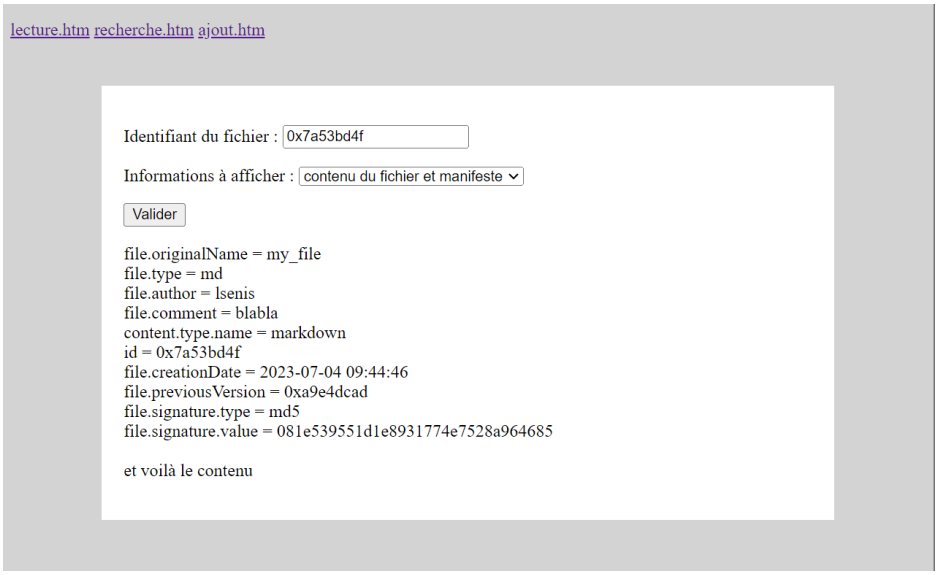


Figure 7: Diagramme de Gantt du projet

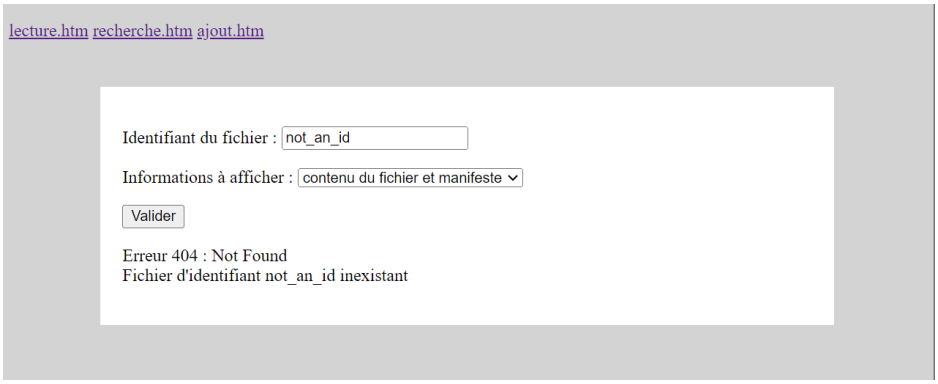


(a) Apparence initiale

(b) Apparence lorsque le serveur est inaccessible

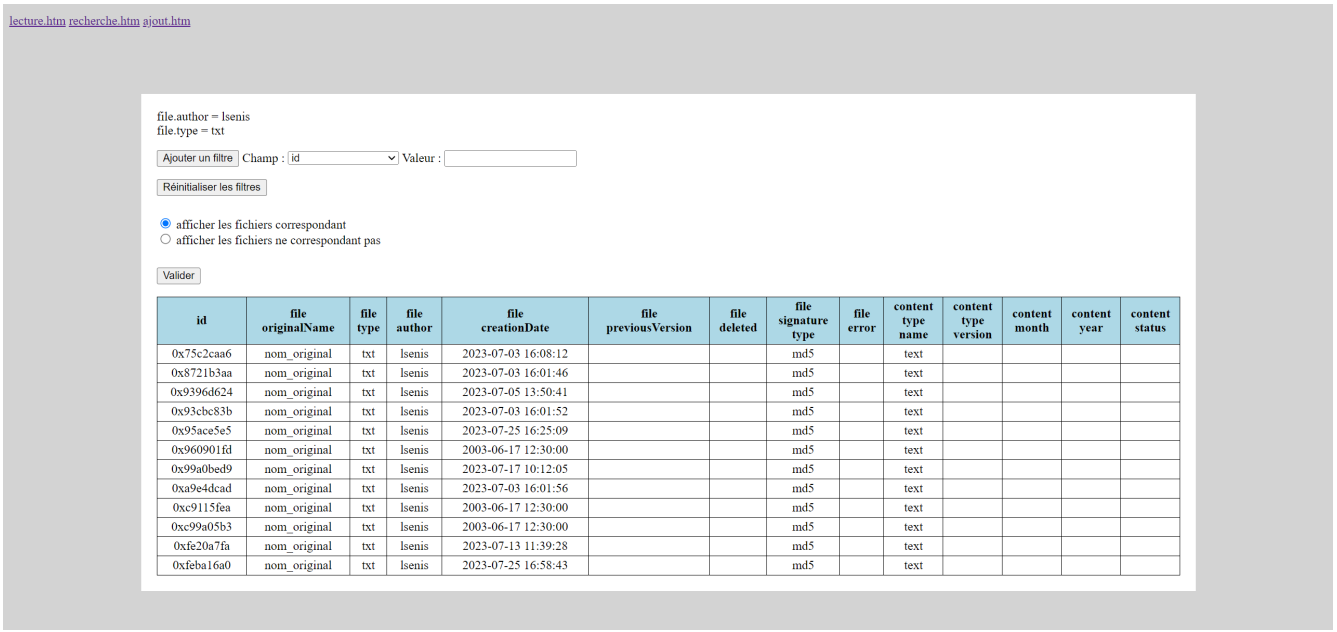


(c) Utilisation standard



(d) Requête invalide

Figure 8: Page de lecture



lecture.htm recherche.htm ajout.htm

file.author = lsenis
file.type = txt

Ajouter un filtre Champ : id Valeur :

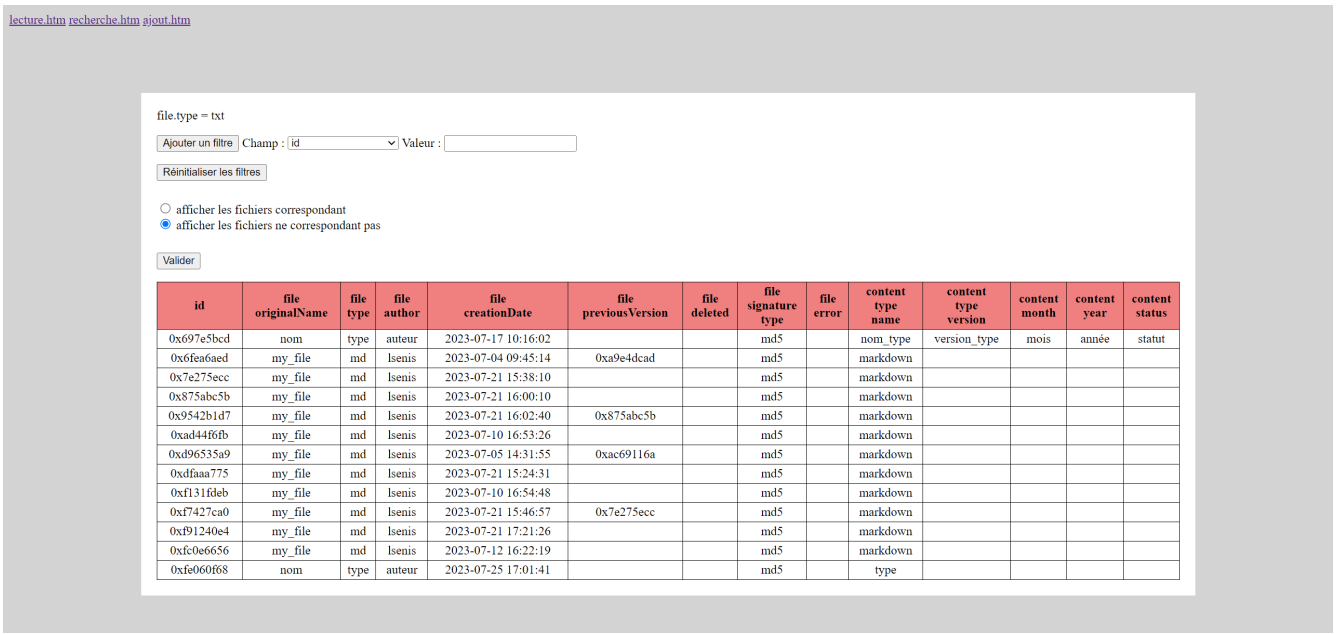
Réinitialiser les filtres

afficher les fichiers correspondant
afficher les fichiers ne correspondant pas

Valider

id	file originalName	file type	file author	file creationDate	file previousVersion	file deleted	file signature type	file error	content type name	content type version	content month	content year	content status
0x75c2caa6	nom_original	txt	lsenis	2023-07-03 16:08:12			md5		text				
0x8721b3aa	nom_original	txt	lsenis	2023-07-03 16:01:46			md5		text				
0x9396d624	nom_original	txt	lsenis	2023-07-05 13:50:41			md5		text				
0x93cbc83b	nom_original	txt	lsenis	2023-07-03 16:01:52			md5		text				
0x95ace5e5	nom_original	txt	lsenis	2023-07-25 16:25:09			md5		text				
0x960901fd	nom_original	txt	lsenis	2003-06-17 12:30:00			md5		text				
0x99a0bed9	nom_original	txt	lsenis	2023-07-17 10:12:05			md5		text				
0xa9e4dcad	nom_original	txt	lsenis	2023-07-03 16:01:56			md5		text				
0xc9115fea	nom_original	txt	lsenis	2003-06-17 12:30:00			md5		text				
0xc99a05b3	nom_original	txt	lsenis	2003-06-17 12:30:00			md5		text				
0xfe20a7fa	nom_original	txt	lsenis	2023-07-13 11:39:28			md5		text				
0xfeba16a0	nom_original	txt	lsenis	2023-07-25 16:58:43			md5		text				

(a) Recherche par inclusion



lecture.htm recherche.htm ajout.htm

file.type = txt

Ajouter un filtre Champ : id Valeur :

Réinitialiser les filtres

afficher les fichiers correspondant
afficher les fichiers ne correspondant pas

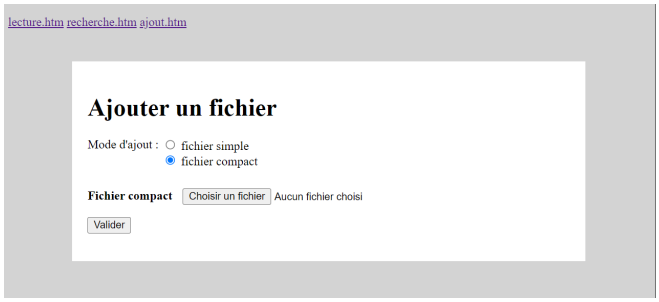
Valider

id	file originalName	file type	file author	file creationDate	file previousVersion	file deleted	file signature type	file error	content type name	content type version	content month	content year	content status
0x697e5bcd	nom	type	auteur	2023-07-17 10:16:02			md5		nom_type	version_type	mois	année	statut
0x6fea6aed	my_file	md	lsenis	2023-07-04 09:45:14	0xa9e4dcad		md5		markdown				
0x7e275ecc	my_file	md	lsenis	2023-07-21 15:38:10			md5		markdown				
0x875abc5b	my_file	md	lsenis	2023-07-21 16:00:10			md5		markdown				
0x9542b1d7	my_file	md	lsenis	2023-07-21 16:02:40	0x875abc5b		md5		markdown				
0xad44f6fb	my_file	md	lsenis	2023-07-10 16:53:26			md5		markdown				
0xd96535a9	my_file	md	lsenis	2023-07-05 14:31:55	0xac69116a		md5		markdown				
0xdfaaa775	my_file	md	lsenis	2023-07-21 15:24:31			md5		markdown				
0xf131fdeb	my_file	md	lsenis	2023-07-10 16:54:48			md5		markdown				
0xf7427ca0	my_file	md	lsenis	2023-07-21 15:46:57	0x7e275ecc		md5		markdown				
0xf91240e4	my_file	md	lsenis	2023-07-21 17:21:26			md5		markdown				
0xfc0e6656	my_file	md	lsenis	2023-07-12 16:22:19			md5		markdown				
0xfe060f68	nom	type	auteur	2023-07-25 17:01:41			md5		type				

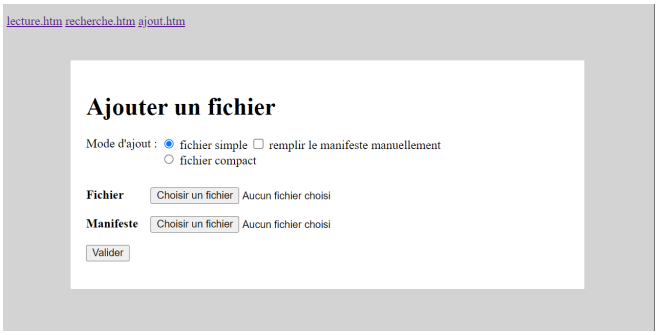
(b) Recherche par exclusion

Figure 9: Page de recherche

(a) Page d'accueil de l'interface d'ajout



(b) Ajout par bundle



(c) Ajout par fichier et manifeste séparés

(d) Remplissage manuel du manifeste

(e) Résultat lors de l'ajout d'un fichier

(f) Interface dédiées aux fichiers de type RSS

Figure 10: Pages d'ajout