Rapport projet fil rouge
François Lamothe

API
Serverless :
- doc : https://ln2e8uzlpa.execute-api.eu-west-1.amazonaws.com/dev/openapi/ui/
- endpoint : https://ln2e8uzlpa.execute-api.eu-west-1.amazonaws.com/dev/openapi/

EC2 FreeBSD (si indisponible, me le signaler afin que j'active l'instance) :
- doc : http://34.253.131.212:8080/openapi/ui/
- endpoint : http://34.253.131.212:8080/openapi/

Gestionnaire d'API
RedYarnProject : https://52.51.220.151:9443/devportal/apis/a567f6cd-bd96-45b2-bdbe-e34f130c4b2e/overview

Code source
Le code source, ainsi que ce rapport et des fichiers de test, sont disponibles sur le dépôt GitHub public suivant : https://github.com/robolth/redyarn

1. Objectifs

Le projet fil rouge rassemblant les évaluations de plusieurs modules, voici un point des différents objectifs assignés et de leur degré d'avancement (il ne s'agit là que d'une synthèse, les points couverts sont détaillés dans la suite du rapport, la page de la partie impliquée est à chaque fois indiquée à droite) :

Module	Fonctionnalité demandée	Résultat	Commentaires	Page
Python	Dépôt de tout type de fichier et restitution au format JSON	Fait	Tous les fichiers sont acceptés, et un JSON est systématiquement renvoyé	6
	API de type RESTfull	Fait	L'API respecte la norme OpenAPI 3	4
	Formats traités : texte, nombre et images au minimum	Fait	Le serveur traite les formats suivants : - plain text (txt, markdown, python, etc.) - csv - images (jpg, gif, png, etc.) - pdf - Microsoft Office document - epub - mp3 - mp4 (expérimental) En outre, tous les autres formats sont supportés à hauteur d'un jeu de métadonnées minimal.	6
	Envoi d'une erreur en cas de format non-supporté	Fait		
	Service implémenté en Python 3.x avec Flask comme moteur web	Fait	Python 3.6 et Flask	4
IPv4	Machine accessible en ssh	Fait	Cela s'est fait sans difficulté particulière avec un certificat fourni par AWS. Procédure en annexe.	17
	Mise en place d'un *packet filter*	Partiel	Fait mais uniquement sur le serveur FreeBSD, je suis parvenu à rediriger le ssh et la page de documentation de l'API mais les requêtes n'aboutissent pas.	14
	Utilisation du protocole https	Partiel	Fait uniquement en *serverless*	14
	Identification des usagers	Fait	Réalisé grâce à des clefs API, avec deux niveaux d'identification : utilisateur et administrateur	9
	Format d'API respectant la norme OpenAPI 3.0	Fait	Le fichier YAML de définition de l'API (disponible sur le dépôt GitHub du projet) respecte la norme OpenAPI 3.0 et a été créé avec Swagger Editor	4
	Démonstration d'un usage de l'API	Fait		

SOA	Mini application visuelle permettant d'utiliser l'API (prise en charge par l'équipe des clients) en plus d'un usage via curl	Fait	La page de documentation de l'API propose une interface en ligne de test de l'API, ainsi que les commande curl nécessaires à son utilisation	4
	Modèle d'architecture de la solution/du développement	Fait	Architecture « Modèle - Vue - Controlleur »	4
	Déclaration de l'API dans un API manager	Fait	L'API a été publiée sur un gestionnaire d'API WSO2 créé par un camarade de classe : il s'agit de l'API RedYarnProject.	12
IaaS	Document succinct avec les consignes pour l'utilisation de l'API utilisant curl, ainsi que du résultat attendu	Fait	Une documentation est proposée en annexe. (La page de documentation de l'API propose également une interface en ligne de test de l'API, ainsi que les commande curl nécessaires à son utilisation. Elle comprend également des exemples de résultats.)	15
	Fichiers tests fournis	Fait	Des fichiers de test sont fournis sur le dépôt GitHub du projet.	1
	Accès à l'API depuis internet	Fait	https://ln2e8uzlpa.execute-api.eu-west-1.amazonaws.com/dev/openapi/ui/	1
	Redéploiement possible de l'application à l'aide de *serverless*	Fait	Possible avec Zappa, cf. procédure détaillée en annexe	16
	4 arguments variables retournés	Fait	Le nombre varie en fonction du type de fichier, mais comprend au minimum 5 arguments : nom, extension, taille, type MIME et type de fichier. La majorité des fichiers ont beaucoup d'autres arguments.	6
	Archivage du fichier dans S3	Fait	Le fichier ainsi que les métadonnées extraites sont archivées dans des buckets S3 distincts.	12
	Suppression du fichier au bout d'un an – théorie	Fait	Les fichiers ainsi que les métadonnées sont automatiquement supprimés au bout d'un an grâce à une règle de cycle de vie AWS Bucket.	12
	Idem – pratique	Fait		
	Utilisation d'un service AWS	Fait	Utilisation du service AWS Rekognition	8
IPv6 / cryptographie	Mise en place d'un VPC	Fait	Un VPC a été mis en place pour protéger la version *serverless* du serveur.	14
	Hashage	Fait	- enregistrement des *hashes* des mots de passe - identification des documents chargés sur le serveur à l'aide d'une empreinte numérique unique	9
	Chiffrement	Fait	Utilisation de mots de passe et de clefs API pour authentifier les utilisateurs et les administrateurs.	9

2. Conception

Le travail de conception de l'application ne s'est pas déroulé d'un seul tenant avant le développement, mais s'est au contraire réparti sur toute la durée du projet en fonction des contraintes et des idées supplémentaires apparues tout le long du projet.

2.1.Architecture

Le modèle client-serveur étant une donnée du sujet, le choix du *framework* Flask pour son implémentation a plus ou moins conditionné l'architecture générale de l'application. Il s'agit d'un *microframework* ne nécessitant l'installation d'aucun composant externe ni d'aucune librairie, et n'imposant aucune couche d'abstraction pour la gestion d'une base de données, ce qui est idéal dans notre cas dans la mesure où nous ne souhaitons pas inclure une interface web évoluée.

(Bien entendu, des librairies supplémentaires seront malgré tout utilisées pour les fonctionnalités du serveur à proprement parler : elles sont détaillées plus bas.)

2.2.Design pattern

La logique du code a été divisée selon le motif de conception classique en trois modules "modèle-vue-controller". Il s'agit du *design pattern* le plus répandu pour les applications web.

La classe définissant le format du fichier JSON généré (métadonnées de base et éléments supplémentaires à extraire en fonction du type de fichier) sont regroupés dans le module "Modèle" : il s'agit de la représentation des données devant être gérées et renvoyées aux utilisateurs.
Les scripts permettant la manipulation des données sont regroupés dans le module "Contrôleurs" : ils permettent d'extraire les métadonnées des fichiers chargés par l'utilisateur, mais également d'accéder aux métadonnées précédemment stockées et de les afficher ou de les supprimer. Ils contiennent également les scripts nécessaires à la création et à la gestion des clefs API utilisateurs.

Enfin, la fonction "Vue" est assurée par le serveur Flask qui gère le renvoi des fichiers JSON en réponse aux requêtes des utilisateurs –dans la mesure où il n'y a pas d'interface utilisateur (à part la page de documentation automatiquement générée à partir du fichier yaml de définition), ce module ne nécessite pas de code supplémentaire.

2.3.Spécifications API

L'architecture de l'API devant respecter les spécifications OpenAPI, mon choix s'est rapidement porté sur l'utilisation de l'outil Swagger Editor pour la génération du fichier de spécifications au format YAML.

J'ai décidé d'implémenter les requêtes suivantes :

- une requête POST sur le chemin d'accès /file/ permettant de charger un document sur le serveur et renvoyant un fichier JSON contenant les métadonnées et les éventuelles données extraites du fichier,

- une requête GET sur le chemin d'accès /metadata/ permettant de renvoyer le fichier JSON généré par une requête précédent,

- une requête GET sur le chemin d'accès /metadata/ascii-art/ permettant de renvoyer le contenu d'une image converti en texte simple (méthode généralement connue sous le nom d'*ASCII art* (ce contenu est également intégré au JSON de la requête normale, mais cette requête spécifique permet de l'afficher directement sans décalage de ligne afin que l'usager reconnaisse immédiatement l'image),

- une requête GET sur le chemin d'accès /metadata/ls/ renvoyant un page html listant l'ensemble des métadonnées déjà extraites avec des liens pour y accéder plus facilement,

- une requête DELETE sur le chemin d'accès /metadata/ pour effacer un fichier JSON de métadonnées précédemment extrait,

- une requête DELETE sur le chemin d'accès /metadata/ls/ pour effacer l'ensemble des métadonnées précédemment extraites,

- une requête POST sur le chemin /api_key/ permettant d'enregistrer un nouvel usager en lui allouant une clef API,

- une requête DELETE sur le chemin d'accès /api_key/ permettant de révoquer une clef API précédemment allouée.

Ces différentes requêtes doivent fournir un service adapté et sécurisé abordant différents domaines dans le cadre du projet fil rouge. Elles sont regroupées dans un fichier yaml de définition qui a été généré en ligne à l'aide de l'éditeur Swagger (cf. annexes).

Note : les pages de documentation générées à partir du fichier YAML proposent de tester les requêtes en ligne (bouton « Try it out »). Les commandes curl ainsi générées fonctionnent sans souci, mais utilisent des arguments supplémentaires inutiles. La documentation en annexe propose les commandes curl minimales.

3. Développement

Le développement de l'application Red Yarn a représenté environ les deux tiers du travail consacré au projet fil rouge. Il a principalement été réalisé en local, sur mon ordinateur personnel (MacBook Pro). L'approche a d'abord été ascendante, avec l'ajout graduel de fonctionnalités de plus en plus complètes sur un petit noyau de base, mais a basculé sur une approche descendante une fois que l'implémentation d'OpenAPI 3 a débuté.

3.1.Serveur

J'avais à l'origine développé un premier prototype de serveur Flask répondant aux attentes du projet, sur la base de travaux pratiques réalisés en cours de Python. J'ai ensuite cherché à adapter ce code aux spécifications OpenAPI suite aux cours de *Services Oriented Architecture* (SOA). Néanmoins cette approche ascendante s'est rapidement avérée fastidieuse, source d'erreurs et surtout d'un code brouillon.
Afin d'optimiser la conformité du code avec les normes de qualité Python ainsi que les bonnes pratiques en vigueur, j'ai donc décidé d'adopter une approche descendante en générant automatiquement l'architecture du serveur Flask à partir de mon fichier de spécifications grâce à l'outil open source OpenAPI Generator.

Néanmoins cela a conduit à des difficultés. En premier lieu, le code généré par le programme OpenAPI generator souffre de plusieurs bugs, comme par exemple des lignes de code en Java au milieu du code Python. Cela vient probablement du fait qu'il s'agit d'un générateur polyvalent pouvant produire du code en de multiples langages, des erreurs de ce genre sont donc compréhensibles. Leur correction fut relativement aisée.

Par ailleurs, le système d'import des *packages* dans l'arborescence du code généré est adaptée des conventions de Python 2, avec des fichiers **ini.py** et un fichier **main.py**. Cela ne pose pas de souci pour le développement (il suffit de lancer le serveur avec la commande python -m), mais pour les déploiements sur AWS, cela peut poser problème. Pour régler cela j'ai simplement déplacé le code du lancement du serveur dans un script app.py un niveau plus haut dans l'arborescence : ce n'est pas aussi propre que les conventions Python 3 le permettent, mais cela suffit amplement.

3.2.Fonctionnalités

Tous fichiers :
Le serveur est capable de renvoyer certaines métadonnées de base de l'intégralité des fichiers qui lui sont envoyés. Outre le **nom**, **l'extension** et la **taille**, l'application détecte également le **type MIME** (*Multipurpose Internet Mail Extensions*) en deux parties (type et sous-type) : cette métadonnée sert de base aux opérations ultérieures en permettant au serveur de réaliser des opérations adaptées en fonction du type de fichier.

J'ai également utilisé la librairie python-magic pour obtenir plus de détails sur le **type de fichier**. Cette méthode s'est révélée extrêmement efficace et donne beaucoup d'informations sur quasiment tous les fichiers envoyés.

Fichiers texte :
Tous les fichiers en **texte simple** (.txt, .md, .py, .sh, etc.) sont pris en charge, le **contenu** étant intégré dans le JSON renvoyé.
Fichiers pdf :
Pour les **fichiers pdf**, le serveur extrait diverses informations liées au document (pour peux qu'ils soient renseignés) :

- logiciels employés pour l'édition et la génération du pdf,
- auteur,
- nombre de pages,
- thème,
- titre.

Le texte est également extrait du pdf et intégré au JSON, à condition bien entendu qu'il ne s'agisse pas d'un simple scan d'un texte. Cette fonctionnalité utilise la librairie PyPDF2.

Fichiers epub :
Les **fichiers epub** (livres électroniques) sont également pris en compte et permettent l'extraction de plusieurs informations, entre autres :

- auteur,
- description,
- version,
- éditeur et identifiant de l'éditeur,
- langue,
- date de publication,
- thème de l'ouvrage,
- titre,
- source.

Cette fonctionnalité utilise la librairie Python epub-meta.

Fichiers csv :
Pour les **fichiers csv**, le serveur peut extraire ces données :

- délimiteurs,
- terminateurs de ligne,
- nombre de lignes et de colonnes,
- et bien entendu le contenu.

Cette fonctionnalité utilise le module csv de la librairie normalisée Python.

Documents Microsoft Office :
Les **documents de la suite Microsoft Office** sont analysés et diverses informations peuvent être extraites :

- auteur,
- nombre de caractères, de mots, de lignes, de paragraphes et de pages,
- entreprise,
- date de création,
- durée d'édition,
- nom de la dernière personne ayant édité le document,
- Nombre de révisions,
- titre.

Le contenu texte n'est malheureusement pas extrait de ces documents. Cette fonctionnalité utilise le module ElementTree ainsi que le module zipfile de la librairie normalisée Python.

Fichiers image :

Pour les **images**, le serveur extrait les éléments suivants:

- données EXIF pour les photos,
- objets identifiés par le service aws rekognition,
- personnalités identifiées par le service aws rekognition,
- contenu au format « *ascii art* ».

Voici un exemple de contenu au format *ASCII art* :

Note : l'activation du service aws rekognition est nécessaire pour l'obtention des résultats associés. Sur RosettaHub ce service ne peut pas être activé en continu, il reste disponible jusqu'à 10h après son activation et doit être relancé à l'issue. Il convient donc de demander son activation au préalable pour tester cette fonctionnalité.

La conversion au format « *ascii art* » est réalisée grâce à la librairie python sci-kit image. Quand aux fonctionnalités de reconnaissance d'image et de célébrités, elles sont implémentés grâce à la librairie boto3.

Fichiers mp3 et mp4 :

Les **fichiers mp3** sont pris en compte, le serveur extrait les informations id3 lorsqu'elles sont disponibles :

- album,

- artiste,
- catégorie,
- n° de piste,
- titre

Enfin, les **fichiers mp4** sont partiellement pris en charge, mais seules les vidéos iTunes sont prises en compte.

Les fonctionnalités de prise en compte des mp3 et des mp4 utilisent la librairie Python mutagen.

3.3.Authentification par chiffrement

Pour sécuriser l'application, j'ai décidé d'implémenter deux types de clefs API : les clefs utilisateur et les clefs administrateur. Ces requêtes servent à sécuriser le serveur en évitant les requêtes malveillantes, ainsi qu'en limitant le nombre de requêtes par jour pour chaque utilisateur afin d'éviter les abus pouvant surcharger le serveur.

Les requêtes GET ne nécessitent aucune clef : n'importe qui peut consulter l'ensemble des métadonnées extraites des fichiers chargés sur le serveur.

Les **clefs utilisateur** sont nécessaires pour les requêtes POST sur /api/file/ et ne permettent aucune autre action. Une clef donnée ne peut effectuer que 10 requêtes de ce genre par jour : cette limitation doit permettre d'éviter toute saturation du serveur.

Les **clefs administrateur** sont nécessaires aux requêtes DELETE sur les chemins d'accès /metadata/ (pour effacer un JSON donné de métadonnées) et /metadata/ls (pour tous les effacer). Cela signifie qu'un simple utilisateur ne peut pas effacer les métadonnées extraites d'un fichier qu'il a chargé. Elles sont également nécessaires aux requêtes POST et DELETE sur le chemin d'accès /api_key/ afin de générer ou désactiver de nouvelles clefs.

Le système n'utilise pas d'identifiant, mais uniquement des phrases de sécurité, notamment afin de s'assurer qu'aucune donnée à caractère personnel ne soit stockée sur le serveur et ainsi rester en conformité avec le RGPD.

Le serveur ne stocke que des empreintes numériques (*hashes*) générés lors de la génération d'une clef API, afin d'éviter qu'il soit possible de pirater les clefs en cas d'accès non autorisé à ces empreintes. La librairie employée pour le hashage est werkzeug.

Enfin, pour plus de sécurité, les empreintes numériques sont stockées sur un bucket S3 distinct ce qui permet de les isoler des métadonnées.

3.4.Consultation des résultats

L'API autorise la consultation des résultats passés, et ce jusqu'à un an après leur dépôt sur le serveur. Pour ce faire, les fichiers chargés ainsi que les métadonnées générées (JSON) sont stockées sur le serveur (cf. plus bas pour les modalités).

L’accès ultérieur implique un moyen d’identifier les fichiers et leurs métadonnées par un identifiant unique, or l’utilisation du nom de fichier n’est pas satisfaisante à cette fin. En effet, le risque que plusieurs fichiers ayant le même nom soient chargés n’est pas négligeable.

La solution retenue fut de générer une empreinte numérique unique à partir du fichier lui-même. Le module hashlib a permis de générer cette empreinte ; pour éviter de bloquer le serveur avec un fichier trop long ou malicieux, l’empreinte n’est générée qu’à partir des 65536 premiers bits du fichier (j’ai estimé que cela serait suffisant pour garantir l’unicité des empreintes).

4. Déploiement

Une fois le code testé en local (sur l'ip localhost), je me suis tourné vers le déploiement. J'ai décidé de tester le déploiement le plus vite possible dès mon premier prototype pour anticiper les problèmes éventuels, quitte à faire beaucoup d'allers-retour entre mon installation en local et mes instances sur AWS.

4.1.IaaS : instances EC2

La première plateforme de déploiement que j'ai testée fut une simple instance EC2 tournant sur Unbuntu, en d'autres termes un service "*Infrastructure as a service*". L'étape préalable au déploiement est bien entendu la configuration de l'instance sur la console AWS, en particulier le contrôle des ouvertures de port pour sécuriser les connexions entrantes.

Ce déploiement a été considérablement facilité par l'emploi d'un environnement virtuel pour réinstaller toutes les librairies nécessaire : il a ensuite suffit de transférer le code par SCP puis de se connecter à l'instance en SSH pour lancer le serveur. (Le service RosettaHub met automatiquement à l'arrêt les instances EC2 après un certain temps d'inactivité, il convient donc de relancer l'instance puis le serveur pour chaque test.)
Une fois ce premier test effectué, j'ai décidé de déployer de nouveau le serveur, cette fois-ci sur une instance EC2 tournant sur FreeBSD. Les images FreeBSD disponibles sur le catalogue d'AWS Marketplace nécessitant un abonnement payant, j'ai installé une image gratuite disponible sur le site de la communauté.

Le seul souci que j'ai rencontré fut l'installation de certaines librairies Python (Scikit image et pythonmagic) indisponibles avec pip sur FreeBSD. J'ai du me tourner vers pkg pour les installer, et utiliser des versions moins à jour, mais dans l'ensemble cette migration fut assez facile.
Enfin, par commodité, j'ai alloué une adresse ip elastique statique à l'instance EC2 pour ne plus avoir besoin de récupérer la nouvelle adresse ip à chaque lancement de l'instance EC2.

Voici l'adresse de l'API sur le serveur FreeBSD : http://34.253.131.212:8080/openapi/ui/

(Note : ce serveur FreeBSD est désactivé en temps normal pour économiser les crédits RosettaHub, il faut me demander son activation avant test ; il est en revanche disponible en permanence sur l'adresse *serverless*, voir ci-dessous.)

4.2.PaaS: Elastic Beanstalk

Dans l'optique de simplifier le déploiement, un test sur AWS Elastic Beanstalk a également été effectué. Il s'agit en l'occurence d'un "*Platform as a service*" permettant de déléguer à AWS la gestion du stockage et de la mise à l'échelle.

Le déploiement s'est avéré extrêmement simple grâce à l'environnement virtuel utilisé pour le développement : il suffit de fournir la liste des librairies nécessaires et le déploiement s'effectue tout seul et automatiquement. Sur l'ensemble des déploiements testés, ce fut clairement le plus simple à mettre en oeuvre.

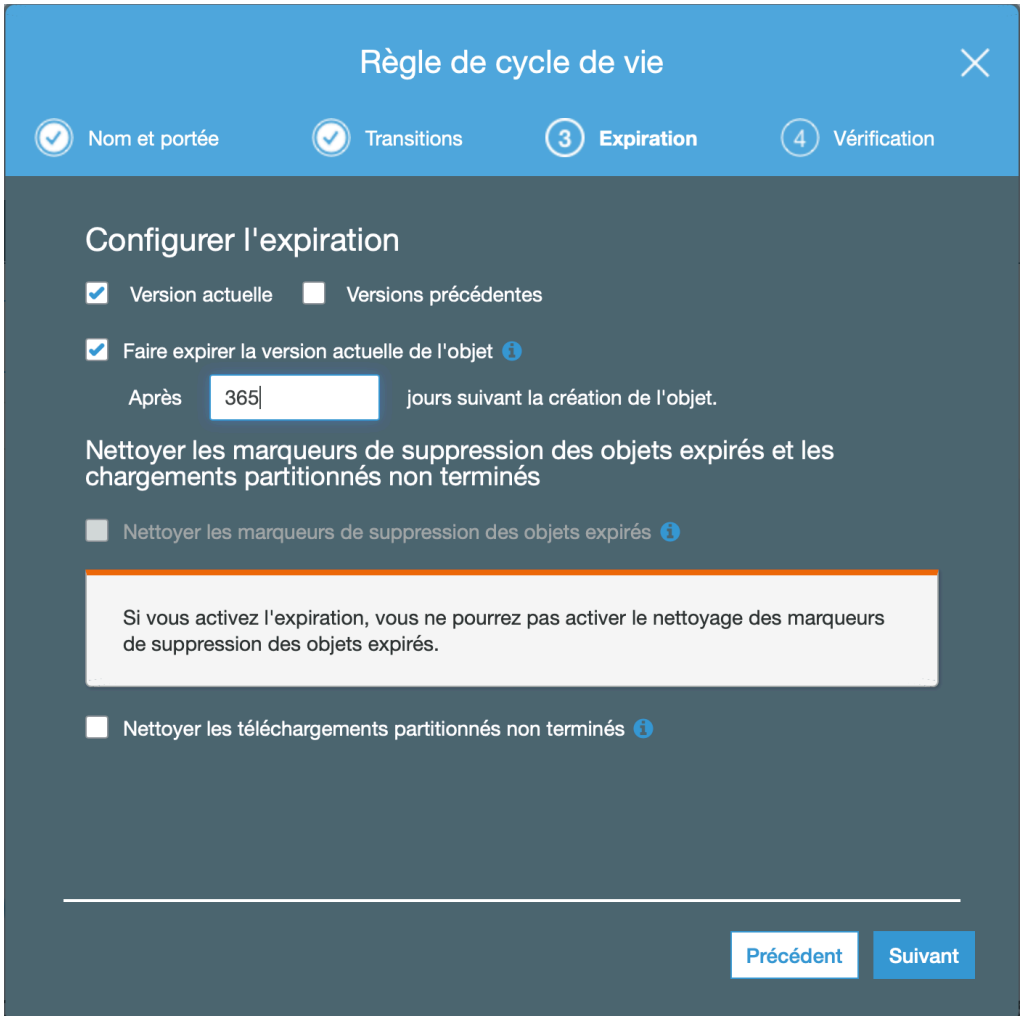
4.3.FaaS : *serverless*

Le déploiement en *serverless* sur AWS Lambda permet de faire tourner notre service sans avoir à s'occuper du serveur. Il s'agit en d'autre termes d'un déploiement "*Function as a service*". Outre un coût moins élevé, cela permet d'externaliser la mise à l'échelle des capacités du serveur à Amazon : chaque requête est gérée indépendamment des autres, et il ne peut donc y avoir de surcharge du service.

Plutôt que de rebâtir le programme en partant de zéro, j'ai décidé d'utiliser le programme zappa pour déployer mon serveur Flask existant. Cet outil est relativement simple à utiliser mais a toutefois nécessité quelques ajustements du projet.

En premier lieu, le programme doit être complètement *stateless* pour que le serveur n'aie plus à être géré. Cela implique de ne plus rien stocker en "local" (jusqu'à présent le stockage s'effectuait sur les EC2). Le stockage des données et des métadonnées a donc été basculé dans des *buckets* S3 grâce à la librairie boto3.

Pour effacer automatiquement les métadonnées et les données chargées au bout d'une année, une règle de gestion a été ajoutée sur le bucket S3 approprié :



J'ai choisi d'isoler les métadonnées des empreintes de phrases secrètes afin de faciliter cette règle de gestion, j'ai donc défini des buckets S3 distincts pour le stockage des empreintes, des données et des métadonnées. Cela permet de sécuriser les données en les compartimentant.

J'ai également du utiliser une option de zappa appelée slim handler pour déployer mon service en raison d'une taille trop importante, probablement liée au nombre de librairies utilisées pour extraire les métadonnées d'une grande variété de types de fichiers.
Il faut faire attention à ce que l'arborescence du serveur définie dans le fichier yaml de l'API corresponde à celle du répertoire du serveur. Cela peut poser problème car, contrairement à une instance EC2, une url Lambda ne donne aucune souplesse pour remonter l'arborescence.

Enfin, lors du déploiement un problème a fait apparaître une incompatibilité entre Zappa et RosettaHub : en effet, Zappa nécessite l'activation du service KMS (Key Management Service) pour le chiffrement des variables d'environnement, et ce même en l'absence d'emploi de ces variables par le programme. Hors, RosettaHub ne permet pas l'activation de KMS. La solution a donc été de modifier le code de l'un des scripts de Zappe (suppression des lignes 1038 et 1039 du fichier core.py) pour que le déploiement ne soit pas bloqué.

Par commodité et pour éviter les problèmes de dépendances de librairies Python, j'ai décidé de réinstaller toutes les librairies de l'environnement virtuel sur une instance EC2 "propre" et d'effectuer le déploiement zappa depuis cette instance.

Une fois ces procédures effectuées, le service est disponible en *serverless* à l'url suivante : https://ln2e8uzlpa.execute-api.eu-west-1.amazonaws.com/dev/openapi/ui/

Note importante : l'API déployée en serverless ne permet pas le dépôt de fichiers trop volumineux. Cela pose essentiellement problème pour les fichiers mp3 et mp4, pour lesquels il vaut mieux utiliser le serveur sur l'instance EC2.

4.4.Gestionnaire d'API

En plus des déploiements sur une EC2 FreeBSD et en *serverless*, l'API a également été publiée sur un gestionnaire d'API WSO2 créé par un camarade de classe : il s'agit de l'API RedYarnProject.

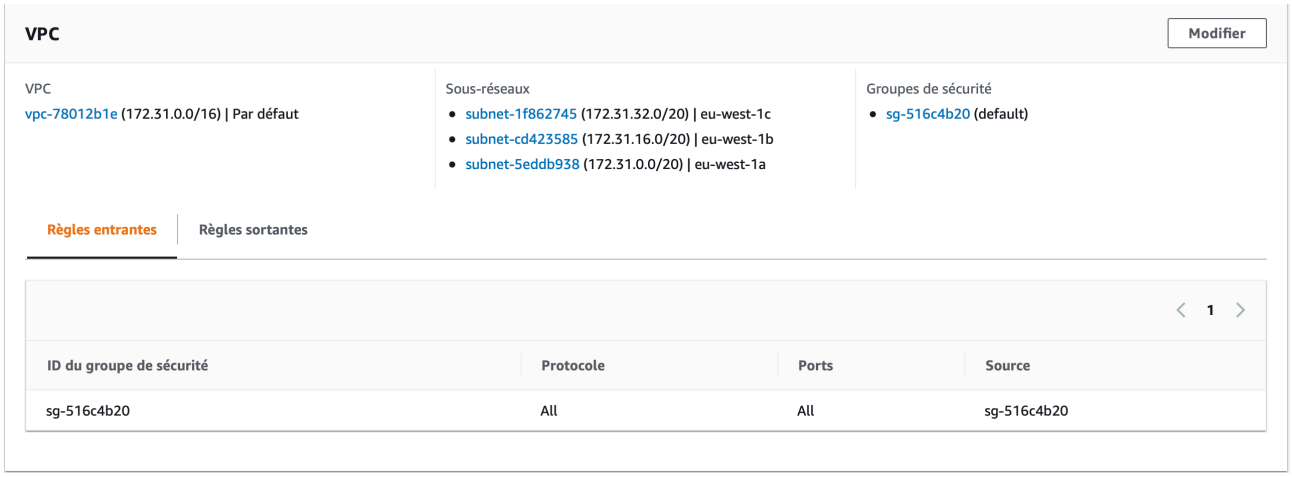
Le référencement a été relativement aisé : j'ai attribué le *endpoint* du serveur EC2 FreeBSD à la version Sandbox, et celui du serveur AWS Lambda (*serverless*) à la version Production.

L'utilisation ne demande aucune configuration particulière, il faut simplement prendre soin de générer un *token* (jeton) correspondant à la version que l'on souhaite utiliser (cela se fait sur l'onglet Applications, et il faut choisir l'application RedYarn). Le gestionnaire choisira ensuite automatiquement quel *endpoint* utiliser en fonction du *token* fourni. (Si la version Sandbox ne fonctionne pas, me demander d'activer l'EC2 FreeBSD.)

4.5.Sécurité

Pour l'utilisation d'instances EC2, les accès aux ports sont configurés manuellement depuis la console AWS. Cela n'est en revanche pas nécessaire avec Lambda, qui se charge de cette configuration automatiquement.

Pour la fonction Lambda en *serverless*, AWS permet la configuration d'un VPC assorti de règles entrantes et sortantes :



Un *packet filter* a également été installé, mais uniquement sur le serveur FreeBSD (je ne suis pas parvenu à le mettre en œuvre sur Lambda.)

Le packet filter fonctionne avec l'utilitaire pfctl et un fichier de configuration pf.conf. Je suis parvenu à :
- rediriger les connexions du port 8585 vers le port 8080,
- rediriger les connexions du port 7515 vers le ssh.

La connexion ssh est correctement re-dirigée, ainsi que la page de documentation de l'API. Malheureusement, les requêtes sont bloquées et je ne suis pas parvenu à les rediriger, cette fonctionnalité est donc incomplète.

Enfin, un avantage de l'utilisation d'AWS Lambda est que l'on bénéficie automatiquement d'une **adresse HTTPS**, ce qui permet d'améliorer la sécurité du service, sans pour autant avoir besoin de payer pour louer un nom de domaine.

5. Annexes

5.1. Documentation API

Cette annexe décrit les commandes curl possibles ainsi que les résultats attendus. Les arguments devant être modifiés ou personnalisés sont en rouge.

Chaque requête est présentée avec un exemple de résultat.

Pour rappel, voici les *endpoints* disponibles :
- https://ln2e8uzlpa.execute-api.eu-west-1.amazonaws.com/dev/openapi/
- http://34.253.131.212:8080/openapi/

(Si le deuxième *endpoint* ne fonctionne pas, me prévenir pour que j'active l'EC2.)

Dépôt de fichier pour analyse :

```
$ curl -X POST "endpoint/file" -H "passphrase: ma-passphrase" -F
"data=@monfichier.txt"


{
  "dateCreated": "2020-03-23",
  "extension": "txt",
  "filename": "monfichier.txt",
  "filetype": "ASCII text, with no line terminators",
  "idFile": "idFile",
  "mimetype": "text/plain",
  "size": 199,
  "text": "Lorem ipsum"
}
```

(Cette requête requiert une clef API utilisateur.)

Récupération des métadonnées d'un fichier déjà chargé :

```
$ curl -X GET "endpoint/metadata/?idFile=idFile" -H "accept:
application/json"


"{'ascii_art': None,\n 'csv_info': None,\n 'date_created':
'2020-03-23',\n 'epub_info': None,\n 'exif_data': None,\n
'extension': 'txt',\n 'filename': 'monfichier.txt',\n 'filetype':
'ASCII text, with no line terminators',\n 'id3_info': None,\n
'id_file': 'idFilE',\n 'image_recognition': None,\n 'mimetype':
'text/plain',\n 'mp4_info': None,\n 'office_info': None,\n
'pdf_info': None,\n 'size': 199,\n 'text': 'Lorem ipsum'}"
```

Suppression d'un jeu de métadonnées :

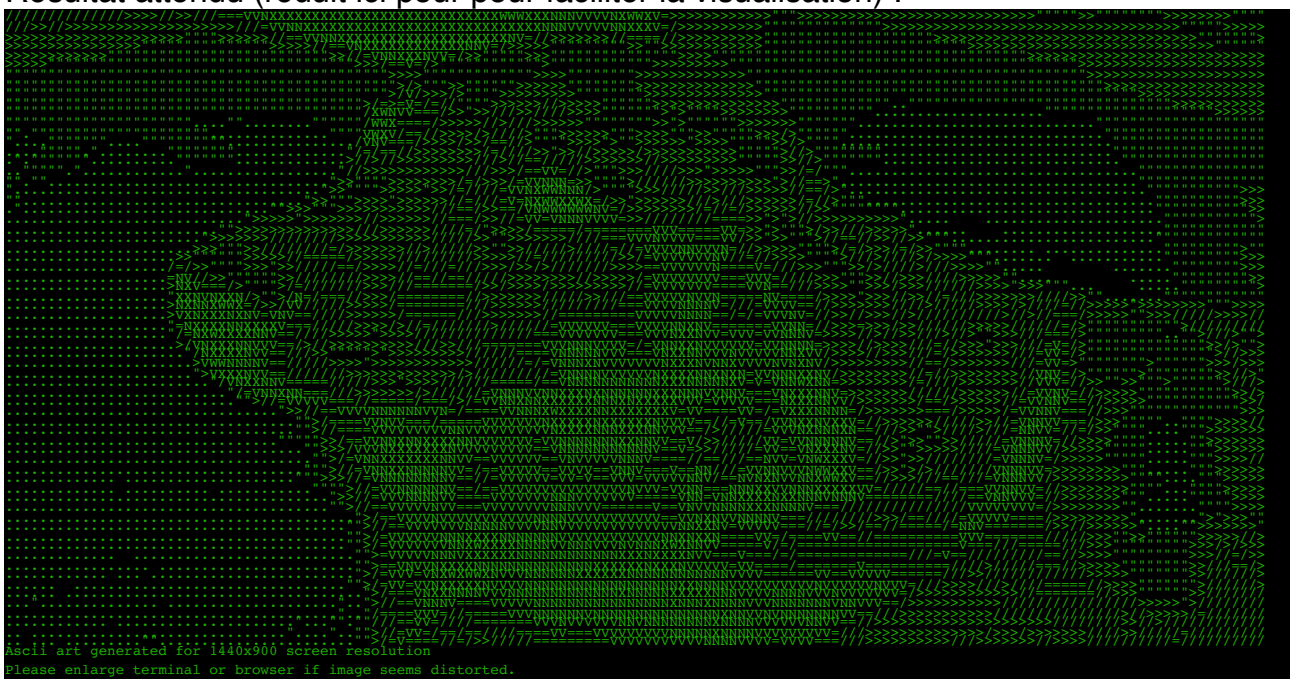
```
$ curl -X DELETE "endpoint/metadata/?idFile=idFile" -H
"admin_passphrase: ma-passphrase-admin"


{
  "detail": "File metadata deleted",
  "status": 200,
  "title": "successful operation",
  "type": "about:blank"
}
```

(Cette requête requiert une clef API administrateur.)

Récupération de l'ascii-art d'un fichier image déjà chargé :

```
$ curl -X GET "endpoint/metadata/ascii-art/?idFile=idFile"
```

Résultat attendu (réduit ici pour pour faciliter la visualisation) :



Affichage de l'ensemble des métadonnées disponibles :

```
$ curl endpoint/metadata/ls


<!DOCTYPE html><html><body><h1>Metadata available</h1><p>Click on
a metadata ID to open its JSON file. (For images, click on
mimetype to open ascii art.)</p><p><a href="..../?idFile.. …
```

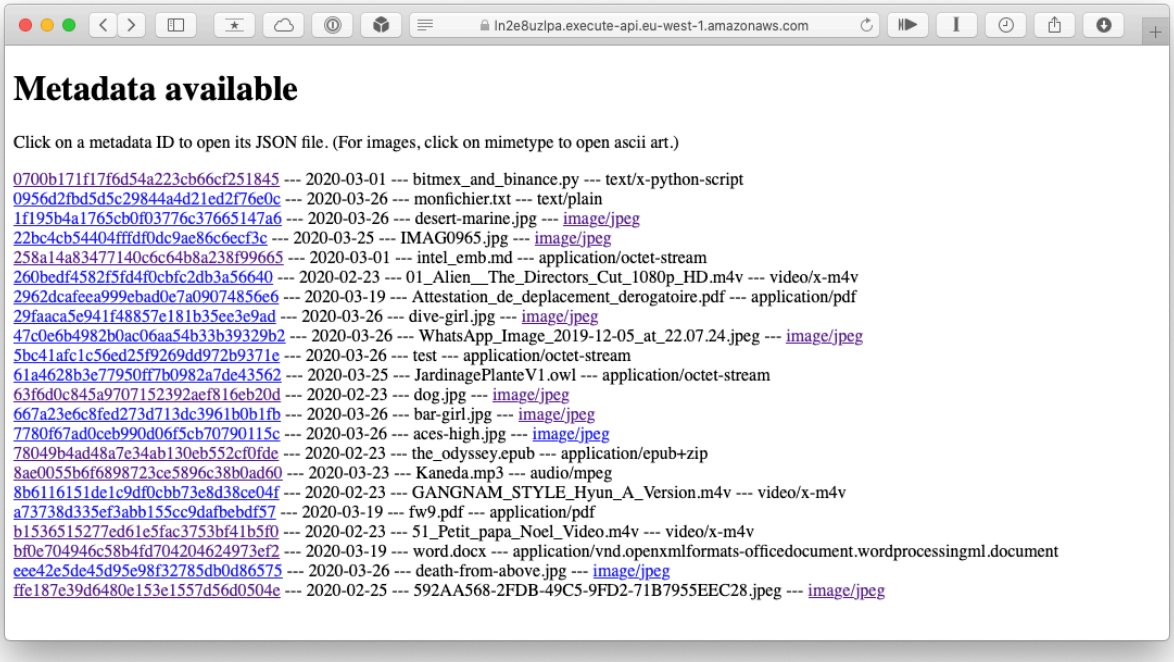
Cette commande affiche le code source d'une page HTML récapitulant l'ensemble des métadonnées disponibles avec leurs identifiants, la date de chargement, le nom du fichier et le type de fichier. (Les identifiants sont des liens qui renvoient vers le JSON

directement ; en outre, pour les fichiers image, le type de fichier est également un lien qui renvoie vers l'ASCII art.)

Une version plus lisible dans un terminal peut être obtenue avec lynx :

```
$ lynx endpoint/metadata/ls


                              Metadata available

Click on a metadata ID to open its JSON file. (For images, click on mimetype to open
ascii art.)

0700b171f17f6d54a223cb66cf251845 --- 2020-03-01 --- script.py --- text/x-python-script
1f195b4a1765cb0f03776c37665147a6 --- 2020-03-26 --- image.jpg --- image/jpeg
22bc4cb54404fffdf0dc9ae86c6ecf3c --- 2020-03-25 --- music.mp3 --- audio/mpeg
```

Et il est bien entendu possible de charger cette url directement dans un navigateur normal :



(L'adresse est simplement « endpoint/metadata/ls »)

Suppression de l'ensemble des métadonnées disponibles :

```
$ curl -X DELETE "endpoint/metadata/ls/" -H "accept: */*" -H
"admin_passphrase: ma-passphrase-admin"


{
  "detail": "All file metadata deleted",
  "status": 200,
  "title": "successful operation",
```

```
  "type": "about:blank"
}
```

(Cette requête requiert une clef API administrateur.)

Enregistrement d'une nouvelle phrase de sécurité utilisateur :

```
$ curl -X POST "endpoint/api_key?new_passphrase=nouveau-mot-de-
passe" -H "admin_passphrase: mon-motdepasse-admin"


{
  "detail": "api key created for uid user3",
  "status": 200,
  "title": "successful operation",
  "type": "about:blank"
}
```

(Cette requête requiert une clef API administrateur.)

Suppression d'une clef API utilisateur :

```
$ curl -X DELETE "endpoint/api_key?uid=mot-de-passe" -H "accept:
*/*" -H "admin_passphrase: mon-motdepasse-admin"


{
  "detail": "api key deleted for uid user3",
  "status": 200,
  "title": "successful operation",
  "type": "about:blank"
}
```

(Cette requête requiert une clef API administrateur.)

5.2.Procédure pour le déploiement serverless

Cette annexe détaille la procédure nécessaire au déploiement de l'API sur le service AWS Lambda au moyen de Zappa.

Installations nécessaires :

```
$ pip install awscli
$ pip install boto
$ pip install –upgrade boto
$ pip install boto3
$ pip install –upgrade boto3
$ pip install botocore
$ pip install –upgrade botocore
```

Configurer le dossier .aws si ce n'est pas déjà fait :

```
$ aws configure (Il faudra entrer les clefs RosettaHub)
```

Si ce n'est pas déjà fait, créer un environnement virtuel Python (nécessaire pour Zappa) :

```
$ python3 -m venv venv
```

Activer son environnement virtuel :

```
$ source venv/bin/activate
```

Installations nécessaire pour le déploiement *serverless* :

```
$ pip install connexion[swagger-ui]
$ pip install zappa
```

Puis faire tourner l'app dans le venv et installer toutes les librairies nécessaires.

Une fois que tout est prêt :

```
$ zappa init
```

Modifier le fichier zappa_settings.json qui a été généré en ajoutant :
"slim_handler": true

(Cela permet de déployer des serveurs de grande taille, cf. explications dans le rapport.)

Modifier venv/lib/python3.6/site-packages/zappa/core.py : Commenter les lignes 1038 et 1039 :

```
        Environment={'Variables': aws_environment_variables},
        KMSKeyArn=aws_kms_key_arn,
```

(Ces deux lignes servent à contourner une limitation de RosettaHub)

À ce stade tout devrait être prêt pour le déploiement :

```
$ zappa deploy dev
```

Si tout se déroule correctement, on obtient l'url de l'API.

5.3.Connexion en SSH à l'instance EC2 FreeBSD

Se connecter à AWS *via* RosettaHub.
Sélectionner Services > EC2.
Dans la colonne de gauche, dans la partie Réseaux et sécurité, sélectionner « Paire de clés ».
Cliquer sur « Créer un nouvelle clé » et sauvegarder le fichier .pem ainsi généré dans un répertoire de son ordinateur.
Puis dans un terminal, à partir de ce répertoire :

```
$ ssh -i "mon-fichier.pem" ec2-user@34.253.131.212
```

(L'adresse ip a été allouée par aws elastic ip. Le nom d'utilisateur peut changer en fonction de l'image d'OS installée, ici il s'agit d'une FreeBSD.)

De même, il est possible d'utiliser scp pour envoyer des fichiers vers ou depuis son instance EC2 :

```
$ scp -i "mon-fichier.pem" -r ~/path/on/my/computer ec2-
user@34.253.131.212:/path/on/ec2/instance
```

5.4.Lancement du serveur sur l'instance EC2 FreeBSD

Une fois connecté à l'instance EC2, pour lancer le serveur :

```
$ cd redyarnserver
$ python app.py
```