Centre de données astronomiques

Documentation Registry

CDD 2019

Alexis Guyot
20/08/2019

Table des matières

Factory ... 2
I. Fonctionnement général ... 2
II. Récupération des données ... 2
a. Présentation générale ... 2
b. Métadonnées des ReadMe ... 3
c. Métadonnées des tables META de VizieR ... 3
d. Métadonnées récupérées localement ... 3
e. Cas spéciaux : Les fichiers statiques ... 4
III. Construction de la réponse ... 4
a. Présentation générale ... 4
b. ResourceFactory ... 4
c. Modèles de ressources et pyxb ... 4
d. Builders : Construction de la ressource ... 5
e. De l'objet ressource au dictionnaire bien formé ... 5
f. Inclusion dans l'enveloppe OAI-PMH ... 6
IV. Lancement de la factory ... 6
V. Mise à jour des UCDs ... 6
Validator ... 7
Harvester ... 8
I. Fonctionnement général ... 8
II. Récupération des réponses ... 8
a. Réponses pré-générées ... 8
b. ListIdentifiers et ListRecords ... 8
III. Utilisation du Harvester ... 9
Factory ... 10
Harvester ... 12
Factory ... 14
I. Mettre à jour les schémas IVOA ... 14
Validator ... 15
I. Ajouter de nouveaux validateurs ... 15

FONCTIONNEMENT

Factory

I. Fonctionnement général

La factory est un programme qui a pour objectif de pré-construire quasiment toutes les réponses possibles aux requêtes OAI-PMH des utilisateurs/programmes qui se servent du registry VizieR. Ces réponses sont générées au format XML et respectent le format précédemment cité. La factory pré-génère les réponses aux requêtes de type **Identify**, **ListSets**, **ListMetadataFormats**, qui retournent des résultats statiques, ainsi qu'aux requêtes de type **GetRecord** qui sont construites à partir de métadonnées récupérées dynamiquement sur les services du CDS. Il y a, pour ce dernier type de requête, une réponse par catalogue visible sur VizieR (liste récupérée via la table METAcat). Les résultats sont enregistrés en respectant le format de métadonnées **ivo_vor**. Les fichiers sont générés dans le dossier dont le chemin est indiqué dans le champ XMLLocation du fichier **Factory/venv/setup.json**. Il est plutôt conseillé d'utiliser des chemins absolus pour éviter les problèmes, mais l'utilisation de chemins relatifs n'est pas exclue. Le chemin doit alors être relatif au fichier **factory.sh** (voir ci-après). Lors de la génération, un dossier dont le nom correspond à la date de création est créé et les résultats sont placés à l'intérieur. Cela permet de garder plusieurs versions de générations.

La génération des fichiers s'exécute en deux étapes distinctes : La récupération des métadonnées puis la construction du résultat XML.

II. Récupération des données

a. Présentation générale

La récupération des métadonnées est initiée lors de la création du « serveur OAI-PMH » (**Server/Server.py**), entité du programme qui utilise la bibliothèque pyoai pour s'assurer de la conformité du résultat par rapport au standard. Elle est effectuée en pratique par les classes présentes dans le module Loaders. Les métadonnées sont principalement récupérées de trois sources différentes : **ElasticSearch** (exploitation des fichiers ReadMe), **VizieR** (exploitation des métadonnées de la base) et depuis des **fichiers stockés localement**. La base ElasticSearch du CDS étant, au moment du développement du Registry, un outil en alpha (limite pré-alpha), une sécurité est assurée pour gérer le cas des catalogues non-gérés par ce dernier grâce à l'API ReadMe, permettant d'obtenir ces derniers au format JSON. A part pour les données récupérées via ElasticSearch ou celles génériques (présentes dans les fichiers locaux), la récupération des autres est faite dans des threads différents afin d'optimiser le temps d'exécution. La fusion des métadonnées récupérées des trois différentes sources se fait dans le constructeur de la classe **Builder** (Builders/Builder.py).

b. Métadonnées des ReadMe

Le premier réflexe de l'application est de récupérer ces métadonnées via la base **ElasticSearch**. Cette dernière présente l'avantage de pouvoir toutes les récupérer en une seule requête plutôt qu'avec un système d'une requête par catalogue, comme ce serait le cas en utilisant **l'API JSON** des ReadMe. Si la récupération via **ElasticSearch** échoue, pour un catalogue particulier ou pour tous, c'est cette dernière qui prendra le relais pour récupérer les métadonnées contenues dans les ReadMe. Malgré sa condition d'outil en alpha, la base **ElasticSearch** reste une option très intéressante pour l'élaboration des résultats puisqu'elle permet de gagner 5 à 10 minutes sur le temps de génération.

Cette étape permet de récupérer les métadonnées suivantes : **title**, **shortname**, **creator**, **description**, **relationships** (de type related-to), **version** (du curator), une date de création alternative (au cas où aucune ne serait présente dans VizieR), **subject**, **source**, **altIdentifiers**, source alternative (au cas où aucun bibcode ne serait présent dans VizieR), **contributor** et **capability** (pour les Associated Data).

Cette partie concerne les fichiers **Loaders/Loader.py** et **Loaders/LoaderReadMe.py**.

c. Métadonnées des tables META de VizieR

La récupération des métadonnées via VizieR se base sur le fichier **Utilitaries/Database.py** qui permet de se connecter directement à la base et de la requêter via SQL, et se fait dans le fichier **Loaders/LoaderVizieR**.

Cette étape permet de récupérer les métadonnées suivantes : **subject**, **waveband** (de coverage), **facility**, **updated** (ainsi que date/@Updated), **relationships** (de type IsPreviousVersionOf et IsServedBy ConeSearch), **tableset**, **footprint** (de coverage) et **capability** (de type ConeSearch).

Chaque table qui contient des coordonnées engendre une capability de type ConeSearch. On les détecte via le champ **cooframe** de METAtab, quand ce dernier vaut entre 1 et 6.

d. Métadonnées récupérées localement

Il s'agit des données qui restent constantes d'une génération à une autre. Elles sont contenues dans des fichiers JSON stockés dans le dossier **LocalData**, et chargées dans le fichier **Loaders/LoaderLocal.py**.

Cette étape permet de récupérer les métadonnées suivantes : **referenceURL**, **footprint**, **identifier**, **capability** (de type WebBrowser et ParamHTTP), une date d'update alternative (la date courante), **contact**, **status**, **validationLevel**, **publisher**, **type**, **contentLevel**, **relationships** (de type IsServedBy TAP) et **rights**.

e. Cas spéciaux : Les fichiers statiques

Les fichiers statiques réponses à **Identify**, **ListMetadataFormats** et **ListSets** sont générés automatiquement grâce à la bibliothèque **pyoai** (fichier Server/Server.py). Il existe 11 ressources spéciales réponses à **GetRecord** qui sont stockées localement de manière statique. Ces dernières ne nécessitent donc pas une récupération de données et sont directement reconstruites à chaque fois par rapport à un modèle stocké dans le dossier **LocalData/LocalResources** par le builder **Builders/BuilderLocal.py**.

III. Construction de la réponse

a. Présentation générale

Une fois toutes les métadonnées nécessaires chargées, l'application se lance dans la génération des fichiers XML. D'abord, elle construit les réponses statiques. Dans l'ordre elle va donc générer le fichier résultat à **Identify**, puis à **ListMetadataFormats**, puis à **ListSets**, et enfin aux 11 ressources statiques de **GetRecord**. Une fois cela fait, elle va lancer la construction des fichiers XML pour les ressources dynamiques. Pour chaque identifiant de catalogue récupéré sur VizieR, elle va d'abord demander au serveur de construire une version au format **etree** de lxml (bibliothèque python qui permet de construire des objets représentant une structure XML) de la réponse, puis va la sérialiser dans le dossier indiqué dans le fichier **setup.json**.

b. ResourceFactory

Pour construire sa réponse, le serveur va faire appel à une **ResourceFactory**, qui va construire un objet représentant un type de ressource particulier. Cette dernière est déclarée dans le fichier **Resource/ResourceFactory.py**. Dans les faits, cette manipulation parait un peu futile aujourd'hui car toutes les ressources générées par cette méthode sont de type **CatalogService**. Cependant, si l'application venait à évoluer et voulait se diversifier pour proposer d'autres types, la mise à jour se ferait très facilement, sans modifier beaucoup de code. Nous reviendrons plus tard sur les façons de faire évoluer simplement la factory. L'application se sert ensuite des **Builders** et des métadonnées récupérées par le serveur pour construire la réponse en respectant le format imposé par le type de ressource.

c. Modèles de ressources et pyxb

Afin de s'assurer dès la conception de la validité des réponses vis-à-vis du format **VORegistry**, nous avons décidé d'utiliser une bibliothèque, **pyxb**, qui permet de générer des classes python construites pour représenter les structures et les contraintes imposées par le standard IVOA via des schémas **XSD**[1]. Ces derniers sont stockés dans le dossier **Schemas**. Cette bibliothèque fonctionne de manière très simple de la façon suivante : Elle prend en entrée le chemin vers un fichier XSD et un nom pour le fichier python qui va être

[1] XSD (XML Schema Description) est un langage de description de documents XML

généré, puis crée dans ce dernier un ensemble de classes ainsi que des fonctions « **BuildAutomaton** » qui vérifient, lors de la construction d'un objet utilisant un des types du fichier, que tous les attributs et éléments obligatoires sont renseignés et que les valeurs entrées sont du bon type et à la bonne multiplicité. Les fichiers générés par ce procédé ont été placés dans le dossier **Models**. Chaque classe représente un type de balise complexe[2]. Quand un type de balise est une spécification d'un autre type déclaré dans le même schéma ou dans un autre, la classe hérite de celle qui définit le type étendu. Les fichiers du dossier Models qui commencent par un « _ » (underscore) sont des fichiers complémentaires générés par pyxb quand une classe en étend une autre, définie dans un schéma différent de celui passé en entrée. La syntaxe de la commande pyxb est la suivante :

pyxbgen -u <CheminVersLeSchemaXSD> -m <NomDuFichierAGenerer>[3]

d. Builders : Construction de la ressource

En se basant sur les classes générées par pyxb, les **Builders** construisent les réponses en utilisant les métadonnées récupérées précédemment. Le **BuilderVOR** construit les éléments et attributs propres aux ressources de type **VOResource** et le **BuilderVOData** ceux propres aux ressources de type **VODataService**, qui sont des extensions du premier type. Les ressources de type CatalogService sont de type VODataService. Pour des raisons de lisibilité, deux builders supplémentaires ont été ajoutés : Un pour construire les capabilities des différents types possibles, et un pour construire les interfaces de tous les types à l'intérieur de ces dernières. Le **BuilderCapability** peut construire des capabilités de type **ConeSearch**, **SIA**, **SSA** et **SLAP**, même si en pratique seul le type **ConeSearch** est actuellement utilisé. Le **BuilderInterface** peut construire des interfaces de type **WebBrowser**, **WebService** et **ParamHTTP**. Les builders ont juste pour tâche de construire un objet dont le type est défini et approuvé par les classes et fonctions générées par pyxb à partir des schémas. Une fois cela fait, ils retournent leur objet à l'appelant, c'est-à-dire soit à d'autres builders, soit aux classes **R_<type>** instanciées par la **ResourceFactory** et déclarées dans le même dossier.

e. De l'objet ressource au dictionnaire bien formé

Ces dernières ont un rôle d'intermédiaires entre la **ResourceFactory** et les **Builders**. Elles ont la charge de transformer les **objets ressources**[4] en **objets xml** puis en **dictionnaires** afin d'être ajoutés à l'enveloppe OAI-PMH par la bibliothèque **pyoai**. Le passage par le format xml est inclus et géré par la bibliothèque pyxb et permet de s'assurer que le résultat retourné par les builders est conforme au format xml (bien formé).

---

[2] Balises qui contiennent d'autres sous-balises ou des attributs

[3] Il n'est pas nécessaire d'indiquer l'extension .py ici, seul le nom du fichier est important.

[4] Objets dont le type est déclaré dans les fichiers générés par pyxb.

f. Inclusion dans l'enveloppe OAI-PMH

Le serveur récupère enfin de la ResourceFactory un dictionnaire dont la structure fait écho à ce à quoi va ressembler l'intérieur du champ « metadata » de la réponse OAI-PMH. Il encapsule ce dernier dans une balise **metadata** (grâce aux fonctionnalités offertes par le fichier **common** de pyoai), construit l'intérieur de la balise **header** puis fait de même avec cette dernière. Enfin, il retourne à la bibliothèque deux objets (trois en pratique mais le dernier vaut None puisque non utilisé dans notre cas), un de type **common.Header** et l'autre de type **common.Metadata**. La bibliothèque se charge de renvoyer un objet de type **lxml.etree** (« type xml » de python) qui se sérialise très facilement.

IV. Lancement de la factory

Pour faciliter son utilisation, un script shell a été développé. Sobrement intitulé factory.sh, le script décore le lancement de la Factory de deux fonctionnalités supplémentaires :

- Préciser une limite pour le nombre de catalogues à traiter :
    - En précisant un nombre entier en paramètre au script.
    - Par défaut il n'y a pas de limite (génération complète).
    - Permet de faire des tests avant une grosse génération.
- Définir un nombre maximum de versions de génération dans le dossier OAI-PMH.
    - Evite de surcharger ce dossier.
    - Nombre à modifier en dur en modifiant la valeur de la variable MAX_REP.
    - Supprime automatiquement la plus vieille génération jusqu'à descendre en dessous de la valeur de MAX_REP.

Le lancement de la Factory se fait de la manière suivante ([*] = optionnel) :

./factory.sh [Limite]

V. Mise à jour des UCDs

Lors de la construction des résultats aux requêtes **GetRecord** sur des catalogues de VizieR, la Factory utilise à certains moments les **UCD** (notamment lors de la construction de l'élément **tableset**). Pour gagner du temps lors de la génération, les correspondances entre les identifiants et les noms sont stockées en local dans le fichier **LocalData/ucd.json**. Puisque ces derniers évoluent peu dans le temps, cela reste intéressant. Cependant, un script a tout de même été mis en place afin de recharge la liste des UCD depuis VizieR, **updateUCD.sh**. Pour ce faire, il suffit d'exécuter ce dernier et tout se fait automatiquement.

Validator

Le Validator est une application développée dans le but d'effectuer une batterie de tests sur les fichiers générés par la Factory. Elle produit un rapport au format XML dans le dossier dans lequel se trouve le fichier validator.sh.

Son fonctionnement est simple et en trois étapes principales : **identifier la génération la plus récente** à analyser, **effectuer les tests**, **sérialiser le rapport** XML. Comme la Factory, il faut au préalable indiquer au Validator où se trouve le dossier OAI-PMH en éditant le fichier **Validator/setup.json**.

Au moment de la rédaction de cette documentation, seul un test a été mis en place. Il analyse le fichier de log créé par la génération et établit un rapport des catalogues dont la génération d'un résultat OAI-PMH a échoué, et qui sont donc indisponibles via le Harvester. Il identifie par la même occasion la cause de cet échec et l'explique de façon compréhensible par un être humain normal pour les causes d'erreurs les plus connues. Pour les erreurs survenues lors de la récupération des données, il fournit également des liens pour aller vérifier soi-même si le service avec qui la communication a échoué est disponible ou non.

Le lancement du Validator s'exécute de la façon suivante :

./validator.sh

Harvester

I. Fonctionnement général

Le Harvester est la partie du projet avec lequel les utilisateurs (humains ou machines) vont interagir pour obtenir les métadonnées générées par la Factory. L'application doit être exposée sur un serveur, se requête via URL et est accompagnée d'un script php qui fait le lien entre le projet python et le point d'entrée URL. Elle possède comme les trois sous-projets précédents un fichier **Harvester/venv/setup.json** qui contient le chemin vers le dossier **OAI-PMH** qui contient les fichiers XML à partir du script php **index.php**.

Son fonctionnement est simple. D'abord, l'URL est récupérée puis parsée pour ne garder que les paramètres. En fonction du verbe renseigné, une fonction est appelée. Celle-ci a la charge de s'assurer que les paramètres renseignés dans l'URL sont corrects et tous présents, si non de lever les exceptions OAI-PMH proposées par la bibliothèque pyoai, et si oui de faire appel à un « serveur ». Ceux-ci sont déclarés dans le fichier **Interface.py** et ont pour mission de charger la réponse XML générée par la Factory. Le résultat est ensuite retourné au script php, puis affiché sur la page html.

II. Récupération des réponses

a. Réponses pré-générées

Pour les réponses pré-générées par la Factory, le Harvester va se contenter de charger le bon fichier XML en fonction des paramètres fournis par l'utilisateur. Une fois cela fait, elle va mettre à jour le header OAI-PMH, notamment les champs **responseDate** et **request**, pour correspondre au contexte dans lequel l'utilisateur récupère les données (bonne date et bons paramètres).

Dans le cas de **GetRecord**, si le format de métadonnées précisé n'est pas **ivo_vor**, l'application va se charger de faire la conversion vers le bon format. Le fichier contenant les fonctions de conversion de **ivo_vor** vers **oai_dc** est **Converter.py**.

b. ListIdentifiers et ListRecords

Dans le cas de ces deux verbes, il était impossible lors de la génération des fichiers par la Factory d'inclure une réponse statique. En effet, les champs **from**, **until** et **set** sont problématiques puisque la liste des résultats va varier en fonction de leur valeur, et que le nombre de possibilités pour ces dernières est infini. Le Harvester est donc obligé de construire lui-même la réponse aux requêtes de ces types. D'un point de vue conceptuel, ce n'est pas trop gênant puisque la construction de ces deux listes ne nécessite pas de métadonnées supplémentaires que celles générées pour les autres verbes (on peut les construire à partir des fichiers déjà générés).

La génération des réponses des deux verbes suit le même fonctionnement : Chargement des chemins vers tous les fichiers générés, traitement, renvoi à pyoai d'une liste (soit de headers, soit de ressources) et d'un resumptionToken (éventuellement vide s'il n'y a plus rien à afficher). Concernant le traitement, il est également relativement similaire d'un verbe à l'autre. Dans tous les cas, pour chaque fichier on charge et parse le XML, on récupère le header OAI-PMH (et la ressource pour ListRecords), on vérifie que le datestamp et le setSpec correspondent aux attributs passés en paramètres et si oui on ajoute le résultat du fichier courant dans la liste à retourner. Comme pour la Factory, pyoai se charge d'ajouter l'enveloppe OAI-PMH et retourne l'objet XML obtenu.

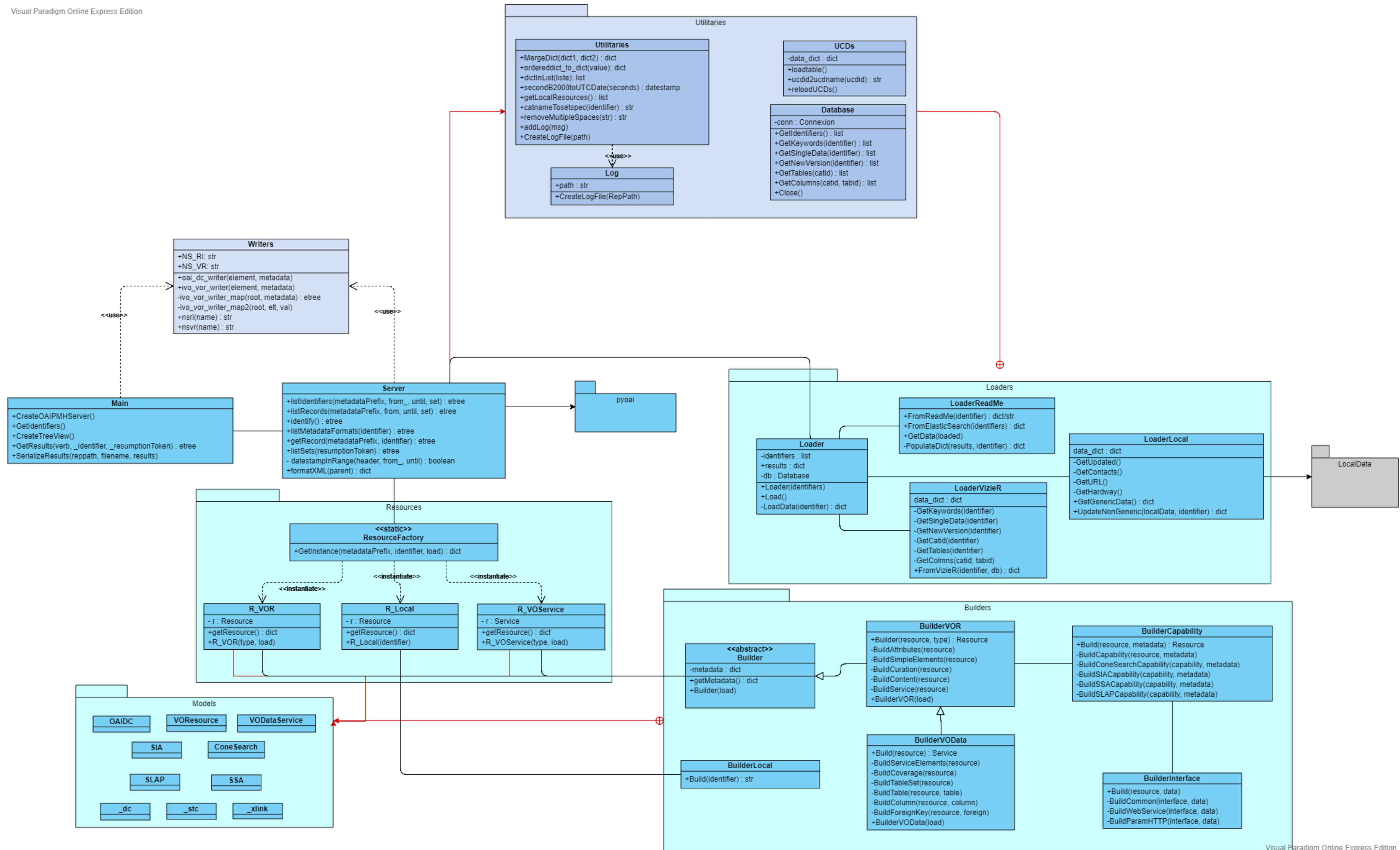
III. Utilisation du Harvester

Pour pouvoir utiliser le Harvester, il faut placer à l'emplacement de notre choix sur le serveur le dossier **Harvester**, le fichier **index.php** (impérativement au même endroit) et le dossier **OAI-PMH** dans lequel les résultats ont été pré-générés par la Factory (si on souhaite le mettre dans un autre dossier, il faut mettre à jour le fichier **setup.json**). Il suffit ensuite d'accéder depuis un navigateur au serveur, et de préciser dans l'URL les paramètres souhaités.

Exemple :
http://monserveur.fr/?verb=GetRecord&metadataPrefix=ivo_vor&identifier=ivo://CDS.VizieR/I/325

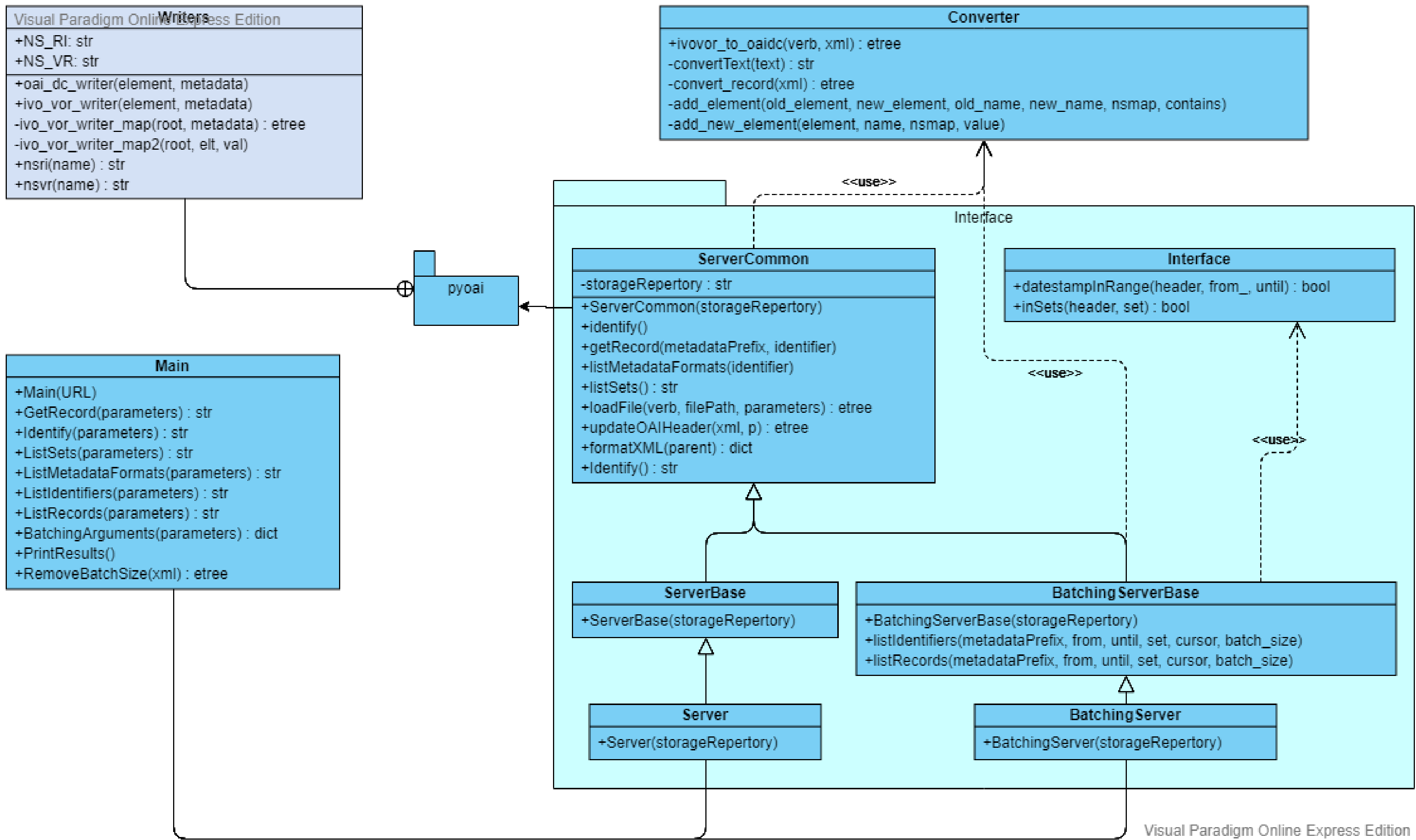
STRUCTURE

Factory



Visual Paradigm Online Express Edition
Utilitaries
Utilitaries
+MergeDict(dict1, dict2) : dict
+ordereddict_to_dict(value): dict
+dictInList(liste): list
+secondB2000toUTCDate(seconds) : datestamp
+getLocalResources() : list
+catnameTosetspec(identifier) : str
+removeMultipleSpaces(str) : str
+addLog(msg)
+CreateLogFile(path)
<<use>>
Log
+path : str
+CreateLogFile(RepPath)
UCDs
-data_dict : dict
+loadtable()
+ucdid2ucdname(ucdid) : str
+reloadUCDs()
Database
-conn : Connexion
+GetIdentifiers() : list
+GetKeywords(identifier) : list
+GetSingleData(identifier) : list
+GetNewVersion(identifier) : list
+GetTables(catid) : list
+GetColumns(catid, tabid) : list
+Close()
Writers
+NS_RI: str
+NS_VR: str
+oai_dc_writer(element, metadata)
+ivo_vor_writer(element, metadata)
-ivo_vor_writer_map(root, metadata) : etree
-ivo_vor_writer_map2(root, elt, val)
+nsri(name) : str
+nsvr(name) : str
<<use>>
<<use>>
Main
+CreateOAIPMHServer()
+GetIdentifiers()
+CreateTreeView()
+GetResults(verb, _identifier, _resumptionToken) : etree
+SerializeResults(reppath, filename, results)
Server
+listIdentifiers(metadataPrefix, from_, until, set) : etree
+listRecords(metadataPrefix, from, until, set) : etree
+identify() : etree
+listMetadataFormats(identifier) : etree
+getRecord(metadataPrefix, identifier) : etree
+listSets(resumptionToken) : etree
- datestampInRange(header, from_, until) : boolean
+formatXML(parent) : dict
pyoai
Loaders
LoaderReadMe
+FromReadMe(identifier) : dict/str
+FromElasticSearch(identifiers) : dict
+GetData(loaded)
-PopulateDict(results, identifier) : dict
Loader
-identifiers : list
+results : dict
-db : Database
+Loader(identifiers)
+Load()
-LoadData(identifier) : dict
LoaderLocal
data_dict : dict
-GetUpdated()
-GetContacts()
-GetURL()
-GetHardway()
+GetGenericData() : dict
+UpdateNonGeneric(localData, identifier) : dict
LoaderVizieR
data_dict : dict
-GetKeywords(identifier)
-GetSingleData(identifier)
-GetNewVersion(identifier)
-GetCatid(identifier)
-GetTables(identifier)
-GetColmns(catid, tabid)
+FromVizieR(identifier, db) : dict
LocalData
Resources
<<static>>
ResourceFactory
+GetInstance(metadataPrefix, identifier, load) : dict
<<instantiate>>
<<instantiate>>
<<instantiate>>
R_VOR
- r : Resource
+getResource() : dict
+R_VOR(type, load)
R_Local
- r : Resource
+getResource() : dict
+R_Local(identifier)
R_VOService
- r : Service
+getResource() : dict
+R_VOService(type, load)
Models
OAIDC
VOResource
VODataService
SIA
ConeSearch
SLAP
SSA
_dc
_stc
_xlink
Builders
<<abstract>>
Builder
-metadata : dict
+getMetadata() : dict
+Builder(load)
BuilderVOR
+Builder(resource, type) : Resource
-BuildAttributes(resource)
-BuildSimpleElements(resource)
-BuildCuration(resource)
-BuildContent(resource)
-BuildService(resource)
+BuilderVOR(load)
BuilderCapability
+Build(resource, metadata) : Resource
-BuildCapability(resource, metadata)
-BuildConeSearchCapability(capability, metadata)
-BuildSIACapability(capability, metadata)
-BuildSSACapability(capability, metadata)
-BuildSLAPCapability(capability, metadata)
BuilderVOData
+Build(resource) : Service
-BuildServiceElements(resource)
-BuildCoverage(resource)
-BuildTableSet(resource)
-BuildTable(resource, table)
-BuildColumn(resource, column)
-BuildForeignKey(resource, foreign)
+BuilderVOData(load)
BuilderLocal
+Build(identifier) : str
BuilderInterface
+Build(resource, data)
-BuildCommon(interface, data)
-BuildWebService(interface, data)
-BuildParamHTTP(interface, data)
Visual Paradigm Online Express Edition

Harvester


Visual Paradigm Online Express Edition
Writers
+NS_RI: str
+NS_VR: str
+oai_dc_writer(element, metadata)
+ivo_vor_writer(element, metadata)
-ivo_vor_writer_map(root, metadata) : etree
-ivo_vor_writer_map2(root, elt, val)
+nsri(name) : str
+nsvr(name) : str
Converter
+ivovor_to_oaidc(verb, xml) : etree
-convertText(text) : str
-convert_record(xml) : etree
-add_element(old_element, new_element, old_name, new_name, nsmap, contains)
-add_new_element(element, name, nsmap, value)
<<use>>
Interface
pyoai
ServerCommon
-storageRepertory : str
+ServerCommon(storageRepertory)
+identify()
+getRecord(metadataPrefix, identifier)
+listMetadataFormats(identifier)
+listSets() : str
+loadFile(verb, filePath, parameters) : etree
+updateOAIHeader(xml, p) : etree
+formatXML(parent) : dict
+Identify() : str
Interface
+datestampInRange(header, from_, until) : bool
+inSets(header, set) : bool
<<use>>
<<use>>
Main
+Main(URL)
+GetRecord(parameters) : str
+Identify(parameters) : str
+ListSets(parameters) : str
+ListMetadataFormats(parameters) : str
+ListIdentifiers(parameters) : str
+ListRecords(parameters) : str
+BatchingArguments(parameters) : dict
+PrintResults()
+RemoveBatchSize(xml) : etree
ServerBase
+ServerBase(storageRepertory)
BatchingServerBase
+BatchingServerBase(storageRepertory)
+listIdentifiers(metadataPrefix, from, until, set, cursor, batch_size)
+listRecords(metadataPrefix, from, until, set, cursor, batch_size)
Server
+Server(storageRepertory)
BatchingServer
+BatchingServer(storageRepertory)
Visual Paradigm Online Express Edition

MAINTENIR/FAIRE EVOLUER

Factory

I. Mettre à jour les schémas IVOA

Si un jour les schémas de l'IVOA venaient à évoluer (très probable), il ne sera pas très difficile de les modifier pour garder une application cohérente et fonctionnelle. Voici les étapes à suivre :

1. Placer le nouveau schéma xsd dans le dossier Schémas des sources de la Factory.
2. A l'aide de pyxb, créer un nouveau fichier python (voir ici ou sur Internet).
   2.1. Il est conseillé de ne pas directement générer le fichier dans le dossier Models. En effet, des fichiers de dépendances (ceux qui commencent par un « _ » (undescore) vont peut-être être créés et risquent de rentrer en conflit avec ceux déjà présents. Pour éviter les problèmes, il vaut mieux faire ça en deux étapes, comme décrit ici.
3. Déplacer le ou les fichiers créés par pyxb dans le dossier Models, sauf les éventuels fichiers de dépendances qui sont déjà présents dans ce dernier.
   3.1. Si un fichier de dépendance pour un namespace déjà généré en tant que Model principal est créé (exemple : _vr = VOResource, _vs = VODataService, voir ici chapitre QName), ne pas l'inclure non plus.
4. Légèrement modifier le principal fichier python obtenu.
   4.1. Dans la section « Import bindings for namespaces imported into schema » (dans les imports), mettre à jour les chemins vers les modules importés. Ce chemin est relatif au fichier main.py. Pour une meilleure compréhension de cette sous-partie, voir fichier VODataService.py.
   4.2. En dessous de la ligne « Namespace = pyxb.namespace.NamespaceForURI(…) », ajouter la ligne « Namespace.setPrefix(<prefix>) pour indiquer le préfixe associé au schéma généré.
5. Si besoin modifier/ajouter des Builders pour correspondre aux nouveaux types générés à partir du schéma.

Validator

I. Ajouter de nouveaux validateurs

Méthode :

1. Créer une nouvelle classe pour le nouveau validateur.
2. Ajouter une fonction principale et autant de fonctions intermédiaires que souhaité pour appliquer les traitements nécessaires.
    2.1. Cette fonction devra prendre en entrée le chemin vers la génération la plus récente du dossier OAI-PMH.
    2.2. Elle devra retourner un objet lxml.etree qui respecte la forme XML suivante :
    <NouveauValidateur>
    <description>Ce que valide ce nouveau validateur.</description>
    <resultat>Le test est-il est passé et si non quels éléments ont coincé et pourquoi
    </resultat>
    </NouveauValidateur>
3. Dans le main, ajouter dans la partie « Builds the XML report » la ligne suivante (adaptée en fonction des besoins) :
    element.append(NouveauValidateur().FonctionPrincipale(path))