Rapport de stage

Implémentation Java d'un VOSpace

Grégory ADAM
Promotion 2019/2020

CENTRE DE DONNÉES ASTRONOMIQUES DE STRASBOURG

Observatoire **astronomique** de Strasbourg | ObAS

IUT Robert Schuman
Institut universitaire de technologie
Université de Strasbourg

Implémentation Java d'un VOSpace

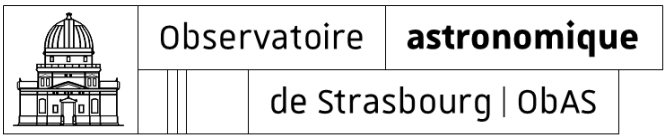
Grégory ADAM
Promotion 2019/2020

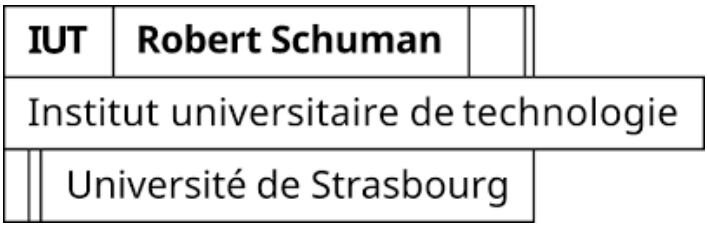
Stage du 08/04/2020 au 23/06/2020.

Tuteur en entreprise : Grégory Mantelet
Tuteur pédagogique : Julien Haristoy

Observatoire astronomique de Strasbourg
11 Rue de l'Université,
67000 Strasbourg

IUT Robert Schuman Illkirch
72 Route du Rhin,
67411 Illkirch-Graffenstaden





Remerciements

Je souhaite remercier en premier lieu Grégory MANTELET pour m'avoir accompagné tout au long du stage et aidé à réaliser le projet.

Je souhaite également remercier André SCHAAFF qui s'occupe de la mise en place des stages et qui a organisé des visioconférences afin que l'on puisse découvrir l'Observatoire même en étant en télétravail.

Je remercie tous les intervenants qui ont présenté le Centre de Données astronomique de Strasbourg et ses services.

Merci également au service informatique de l'Observatoire qui a permis que le stage se déroule en bonne condition en télétravail en mettant en place notamment des outils de communication.

Enfin je souhaite remercier le directeur de l'Observatoire, Pierre-Alain DUC, et le directeur du Centre de Données astronomique de Strasbourg, Mark ALLEN, pour m'avoir accueilli dans leur établissement.

Je remercie également l'IUT Robert SCHUMAN et plus spécifiquement le département informatique de m'avoir permis d'effectuer ce stage dans de bonnes conditions même en ces temps difficiles.

Pour finir, je remercie Julien HARISTOY, l'enseignant responsable de mon stage, de m'avoir accompagné.

Table des matières

Introduction

Pour le bon déroulement de la recherche astronomique, il est important que différents organismes et chercheurs collaborent. C'est pour cela qu'il faut mettre en place des services permettant de facilement communiquer avec d'autre. En effet la plupart des services déjà existant peuvent être interconnectés afin de croiser les informations qu'elles contiennent. Par exemple : récupérer les coordonnées d'un objet astronomique dans un service à partir d'un de ces noms, afficher la position de l'objet dans un atlas du ciel et avoir des références concernant cet objet dans un autre service.

C'est dans cet environnement que ce déroule mon stage, il a pour but de réaliser un standard précisant comment peuvent être échangées des données permettant ainsi une interopérabilité entre plusieurs services. Ce standard a été mise en place par un organisme regroupant plusieurs établissements et notamment des Observatoires. Son but est de créer des standards afin qu'il soit simple de collaborer, que ce soit d'un Observatoire à l'autre ou encore entre chercheurs.

Ces données pourront être stocké sous n'importe quelle forme. Le standard ne précise que la communication entre le client et le serveur qui permet de manipuler et accéder à des données. Pour des questions de simplicité dans mon implémentation, les données seront stockées sous forme de fichier.

La première itération d'un mot défini dans le glossaire est suivie par un astérisque.

1 Contexte du stage

1.1 Présentation de l'entreprise

L'Observatoire astronomique de Strasbourg est un établissement de recherche et d'enseignement fondé en 1881. Situé le sur campus de l'Université de Strasbourg, il est une unité mixte de recherche du CNRS (Centre National de la Recherche Scientifique) et de l'Université de Strasbourg, Il dépend donc de ces deux organismes. La principale fonction de celui-ci n'est pas l'observation du ciel mais la distribution, l'analyse et le traitement de données astronomiques. Il est composé de deux équipes scientifiques :

GALHECOS (*Galaxies, High Energy, Cosmology, Compact Objects & Stars*) :

Cette équipe étudie les galaxies, notamment leur formation et évolution.

Le CDS (Centre de Données astronomiques de Strasbourg) :

Le principal objectif du CDS est de répertorier, stocker et distribuer des données astronomiques en proposant des services permettant l'accès et l'utilisation de ces données, celles-ci peuvent être des images, des liens vers des articles ou encore des catalogues d'objets astronomiques. Il s'occupe également de faire des recherches à l'aide de ces données. Ainsi cette équipe doit pouvoir gérer un grand nombre de données que cela soit en termes de stockage, traitement ou encore pouvoir répondre aux requêtes faites par les utilisateurs.

L'Observatoire compte 87 personnes dont 38 au sein du CDS, service dans lequel j'ai effectué mon stage. Le directeur de l'Observatoire est Pierre-Alain DUC.

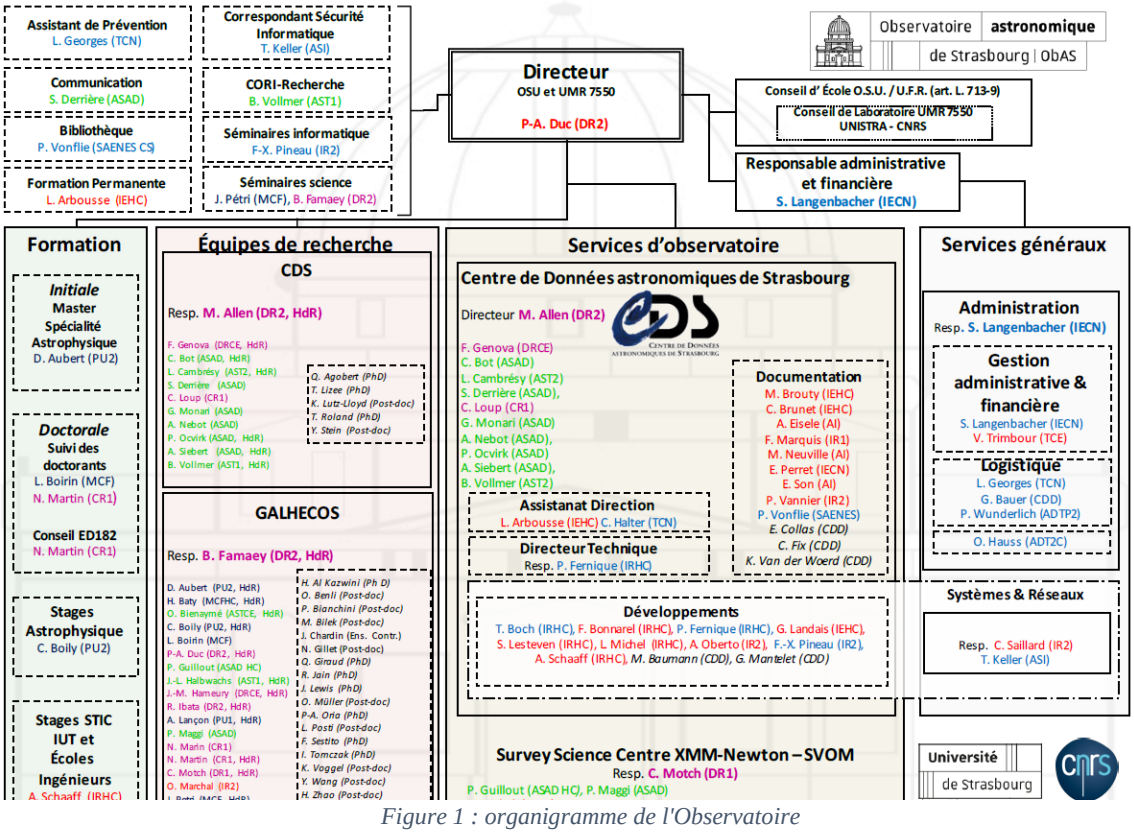


Figure 1 : organigramme de l'Observatoire

Cet établissement de recherche est aussi membre de l'IVOA (*International Virtual Observatory Alliance*). Il s'agit d'une organisation scientifique composée de plusieurs organismes du monde entier. Sa principale mission est de définir des standards afin d'assurer une interopérabilité entre les services de ces organismes. L'ensemble des services issus de cette collaboration constitue ce que l'on appelle l'Observatoire Virtuel.

Figure 2 : logo de l'IVOA

Les services

Le CDS propose plusieurs services à la communauté astronomique et au grand public :

SIMBAD :

Figure 3 : logo SIMBAD

Une base de données d'objets astronomiques contenant des informations sur des objets hors du système solaire. Elle est mise à jour quotidiennement par identification croisée. Cela permet de pouvoir retrouver un objet en utilisant n'importe quel identifiant le représentant. Il est également possible de retrouver un objet en utilisant sa position dans le ciel. Les documentalistes doivent alors regrouper toutes les informations concernant le même objet en prêtant attention aux identifiants qui doivent correspondre au bon objet. Cette base compte plus de 10 millions d'objets astronomiques et plus de 35 millions d'identifiants.

VizieR :

Figure 4 : logo VizieR

Une base de données de catalogues astronomiques. Un catalogue est un ensemble de données provenant d'observations effectuées au sol ou depuis l'espace, et stocké sous la forme d'une ou plusieurs tables. Il permet aux utilisateurs d'extraire des informations en utilisant certains critères de sélection. VizieR répertorie actuellement 19765 catalogues.

Aladin :

Figure 5 : logo Aladin

C'est un atlas interactif du ciel réalisé en Java. Il permet de visualiser et localiser des objets du ciel en précisant un nom d'objet. Il est possible de zoomer sur ces objets et d'appliquer différents filtres. SIMBAD et VIZIER peuvent être couplés avec Aladin afin d'afficher des informations de ces bases en plus des images d'Aladin. Il existe aussi une version web codée en Javascript appelée Aladin lite mais avec un nombre réduit de fonctionnalités.

1.2 Le standard VOSpace*

Le standard VOSpace est une interface* mise en place par l'IVOA* qui spécifie comment accéder à des stockages de données afin de conserver, modifier et récupérer des informations. Cela permet d'avoir une manière unique et universelle pour mettre en place un système d'échange de données : le client n'a pas à faire attention à la manière dont une donnée est stockée ; la réponse à la requête d'un client sera la même quel que soit le système de stockage derrière l'interface. Grâce à ce service un client peut :

- Ajouter ou supprimer des éléments.
- Manipuler et accéder à des métadonnées* telles que : une description, un sujet.
- Accéder à des données via un URI*

Chaque donnée est représentée sous la forme d'un nœud*. À ce nœud sont associées des métadonnées descriptives.

Le modèle utilisé par IVOA pourrait être comparé avec un système de fichier comme dans n'importe quel système d'exploitation comme par exemple : Windows, MacOS ou encore Linux, un nœud dans VOSpace représentant un fichier, un répertoire ou un lien vers un autre nœud.

Ce standard recommande aussi une manière asynchrone* d'effectuer les transferts et certaines manipulations de données. Cela signifie, par exemple, que lors d'une copie ou d'un déplacement d'un nœud, le client n'a pas besoin de rester connecté au service jusqu'à ce que l'action soit terminée. De même lors d'un transfert, la génération de l'URL vers laquelle le client pourra transférer des données pourra se faire de manière asynchrone mais le transfert en lui-même vers l'URL générée sera fait en synchrone*. Pour ce faire, VOSpace spécifie d'utiliser UWS* (*Universal Worker Service*), un autre standard de l'IVOA qui définit comment gérer l'exécution d'actions de manière asynchrone.

VOSpace est également décrit comme un service RESTful*. RESTful est une architecture logicielle définissant des contraintes telles que :

- Les réponses aux requêtes doivent être formatées (par exemple JSON*, XML*)

- Les responsabilités sont séparées entre le client et le serveur. Ainsi, le stockage des données et l'interface utilisateur sont indépendants. Il devient alors facile d'avoir plusieurs interfaces utilisateur sur plusieurs plateformes différentes qui utilisent le même service côté serveur. Cela permet aussi de faire évoluer soit le service soit l'interface sans impacter son pair.
- Le service doit être sans état, c'est-à-dire que le serveur ne conserve aucune information concernant la connexion précédente du client. Le client doit alors envoyer toutes les informations nécessaires afin d'opérer des actions. Cela permet notamment de ne pas conserver une connexion continue entre le client et le serveur.
- Le service doit être une interface uniforme, c'est-à-dire que chaque donnée est identifiée. Les requêtes où il y a des données doivent contenir assez d'informations pour pouvoir modifier ou supprimer cette donnée. Les messages du service doivent contenir assez d'informations afin de pouvoir manipuler la réponse et l'utiliser.

En utilisant HTTP*, il faudra utiliser des URI afin d'accéder à des nœuds en utilisant des méthodes HTTP comme par exemple : GET, PUT, POST, DELETE.

Dans le cadre de mon stage nous avons décidé d'implémenter VOSpace sous la forme d'un micro-service*. Cela signifie qu'il doit être petit et léger et qu'il y aura un faible couplage entre les services de l'application. Cela signifie qu'il y a une indépendance entre les différents services. Par exemple, une interface graphique et la partie requête vers le serveur, chacun pour évoluer sans que l'autre ait à être modifié. Mon implémentation de VOSpace est une implémentation de bibliothèque générique permettant une installation rapide et facile d'un VOSpace. La généricité permet au service de pouvoir être installé sur n'importe quel système, que cela soit un système avec des fichiers déjà existant ou non. La généricité passe aussi par la possibilité que cette bibliothèque puisse être facilement étendue grâce à des classes abstraites et interfaces, pour par exemple supporter de nouveaux systèmes de stockage tel qu'un cloud*. La mise en place du service se fait automatiquement lors du lancement en utilisant le fichier de configuration. Si la base de données et le dossier n'existe pas alors ils sont créés, si un dossier existe et qu'il contient déjà des fichiers alors des nœuds sont créés en fonction de ces fichiers. Le service, une fois le fichier de configuration rempli, est directement utilisable.

Ce service sera utilisé par les astronomes par l'intermédiaire d'une interface graphique afin de ne pas à avoir à connaître l'utilisation de la ligne de commande. Il sera principalement utilisé par l'Observatoire de Strasbourg mais d'autres Observatoires qu'il soit français ou étrangers pourront l'utiliser pour leur propre système.

2 Utilisation du service

VOSpace impose plusieurs fonctionnalités dans son standard :

Considérons qu'un nœud est un fichier stocké en local (c'est-à-dire dans un simple système de fichier/dossier) associé avec des métadonnées. Ce fichier pouvant stocker des images astronomiques, des articles ou encore des tables de base de données astronomiques. Les métadonnées eux sont des informations sur le fichier, par exemple : la taille du fichier, son auteur, une description ou encore la liste des nœuds fils si le fichier est un dossier.

***GetViews/Properties/Protocols* :**

Cette fonctionnalité permet d'obtenir la liste de toutes les vues*/propriétés*/protocoles* fournis, acceptés ou encore utilisés dans le service (une vue est un format de fichier par exemple : PNG, ZIP. Les propriétés sont des informations supplémentaires sur le fichier, par exemple : l'auteur, une description ou une date de création. Un protocole est une méthode de communication entre un client et un serveur, par exemple : HTTPGET). « Un élément fourni » correspond à un élément que le client peut récupérer tandis qu'un « élément accepté » signifie que le client peut utiliser ces éléments pour envoyer des informations.

Pour récupérer cette liste, il suffit d'envoyer une requête HTTP-GET vers la ressource VOSpace correspondante.

Par exemple, voici comment récupérer la liste des propriétés avec l'outil cURL :

```
curl -v "localhost:8080/vospace/properties"
```

La réponse suivante apparaît alors :

```
<vos:properties xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.1">
    <vos:accepts>
        <vos:property uri="ivo://ivoa.net/vospace/core#contributor"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#coverage"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#creator"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#date"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#description"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#groupread"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#groupwrite"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#identifier"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#language"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#publicread"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#publisher"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#quota"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#relation"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#rights"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#source"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#subject"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#title"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#type"/>
    </vos:accepts>
    <vos:provides>
        <vos:property uri="ivo://ivoa.net/vospace/core#availableSpace"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#btime"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#ctime"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#format"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#length"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#mtime"/>
    </vos:provides>
    <vos:contains>
        <vos:property uri="ivo://ivoa.net/vospace/core#btime"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#ctime"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#description"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#length"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#mtime"/>
        <vos:property uri="ivo://ivoa.net/vospace/core#subject"/>
    </vos:contains>
```

Il est possible d'observer les propriétés acceptées, fournies et utilisées, comme par exemple :

- Description (*description*) et titre (*title*) sont des propriétés acceptées. Ainsi cela permet au client de donner une description et un titre.
- La taille du nœud (*length*) et la date de création du nœud (*btime*) sont des propriétés fournies, c'est-à-dire que ces propriétés sont écrites par le service et que le client ne peut que les lire.
- Enfin la description (*description*) et sujet (*subject*) sont des propriétés utilisées autrement dit, il y a au moins un nœud où cette propriété est présente.

Ici les réponses sont sous le format XML, comme spécifié par le standard VOSpace. Mais notre librairie permet également de les retourner dans le format JSON. Pour cela, il suffit de

renseigner l'en-tête HTTP « Accept » avec le type MIME du format de réponse souhaitée : ici, « application/json ».

Cela peut se faire en utilisant le paramètre -H de la commande cURL :

```
curl -v -H "accept: application/json" "localhost:8080/vospace/properties"
```

Pour obtenir les informations des autres métadonnées du service il suffit de remplacer *properties* par *views* ou *protocols.*

***GetNode* :**

Cette fonctionnalité est utilisée pour obtenir les métadonnées concernant un nœud, comme par exemple son type, ses propriétés ou encore les vues qui peuvent être utilisées pour télécharger ou envoyer des données.

Par exemple, avec la requête suivante, le client demande d'obtenir les informations du nœud *folder*

```
curl -v "localhost:8080/vospace/nodes/folder"
```

Voici un exemple de réponse :

```
<vos:node xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
nstance" uri="vos://cds-vospace.com!vospace/folder" version="2.1" xsi:type="vos:ContainerNode">
    <vos:properties>
        <property uri="ivo://ivoa.net/vospace/core#btime">06/02/2020 03:26:16</property>
        <property uri="ivo://ivoa.net/vospace/core#ctime">06/05/2020 10:56:07</property>
        <property uri="ivo://ivoa.net/vospace/core#mtime">06/02/2020 03:26:16</property>
        <property uri="ivo://ivoa.net/vospace/core#length">177414</property>
        <property uri="ivo://ivoa.net/vospace/core#subject">subject</property>
    </vos:properties>
    <vos:accepts>
        <view uri="ivo://ivoa.net/vospace/core#anyview"/>
    </vos:accepts>
    <vos:nodes>
        <vos:node uri="vos://cds-vospace.com!vospace/folder/anotherFolder" xsi:type="ContainerNode"/>
        <vos:node uri="vos://cds-vospace.com!vospace/folder/test_copy1" xsi:type="DataNode"/>
    </vos:nodes>
</vos:node>
```

On peut observer les propriétés associées avec ce nœud ainsi que leur valeur, les vues acceptées et les enfants (puisque le nœud est un nœud conteneur).

Il y a aussi des paramètres optionnels, tel qu'un URI qui précise à partir de quel fils la liste des enfants va commencer, ou encore *limit* qui précise le nombre de fils qui doit être affiché. Ces deux paramètres sont utilisés lorsque le nœud est un nœud conteneur (*ContainerNode*). Un autre paramètre est disponible : *detail.* Il sert à préciser le niveau de détails voulu. Il peut être mis à « min » (minimum), « max » (maximum) ou encore « *properties* » (pour afficher seulement les propriétés).

***SetNode* :**

SetNode permet d'ajouter, modifier et supprimer des propriétés. Cette fonctionnalité doit envoyer un fichier au format XML lors de la requête.

Par exemple, voici comment modifier les propriétés du nœud « folder » avec la commande cURL :

```
curl -v -X POST -d @setNode.xml -H "Content-type: text/xml" "http://localhost:8080/vospace/nodes/folder"
```

La requête doit être envoyée avec la méthode HTTP POST. Les propriétés du nœud à modifier doivent être décrites dans un fichier XML. Il faut alors aussi préciser le fichier à envoyer - ici, setNode.xml – ainsi que son type, sinon la requête sera refusée par le service.

Voici à quoi le fichier setNode.xml pourrait ressembler :

```
<vos:node xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0"
version="2.1" xsi:type="vos:ContainerNode" uri="vos://cds-vospace.com!vospace/folder">
<vos:properties>
<vos:property uri="ivo://ivoa.net/vospace/core#description" xsi:nil="true"></vos:property>
<vos:property uri="ivo://ivoa.net/vospace/core#title" readOnly="true">title</vos:property>
</vos:properties>
</vos:node>
```

Dans le ficher il faut donner l'URI du nœud à modifier et son type, puis les propriétés. Ici, un titre, qui sera seulement accessible en lecture et ne pourra donc être ni supprimé ni modifié. La propriété « description » sera supprimée grâce à l'attribut xsi:nil="true". L'URI du nœud et son type ne peuvent être changés et doivent être donnés de manière strictement identique à leur valeur dans le VOSpace.

La réponse est la représentation du nœud avec les modifications faites. On peut observer que la propriété « *description* » a été supprimée et la propriété « *title* » ajoutée.

```
<vos:node xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" uri="vos://cds-vospace.com
!vospace/folder" version="2.1" xsi:type="vos:ContainerNode">
    <vos:properties>
        <property uri="ivo://ivoa.net/vospace/core#btime">06/02/2020 03:26:16</property>
        <property uri="ivo://ivoa.net/vospace/core#ctime">06/10/2020 11:39:34</property>
        <property uri="ivo://ivoa.net/vospace/core#mtime">06/02/2020 03:26:16</property>
        <property uri="ivo://ivoa.net/vospace/core#length">177414</property>
        <property uri="ivo://ivoa.net/vospace/core#subject">subject</property>
        <property uri="ivo://ivoa.net/vospace/core#title">title</property>
    </vos:properties>
    <vos:accepts>
        <view uri="ivo://ivoa.net/vospace/core#anyview"/>
    </vos:accepts>
    <vos:nodes>
        <vos:node uri="vos://cds-vospace.com!vospace/folder/anotherFolder" xsi:type="ContainerNode"/>
        <vos:node uri="vos://cds-vospace.com!vospace/folder/test_copy1" xsi:type="DataNode"/>
    </vos:nodes>
</vos:node>
```

***CreateNode* :**

CreateNode est utilisé afin de créer de nouveaux nœuds sans données associées. Cette fonctionnalité, utilise aussi un fichier XML : ici, newLink.xml. Mais cette fois, comme spécifié par RESTful, la méthode PUT de HTTP doit être utilisée afin de créer une nouvelle ressource (ici, un nouveau nœud).

Voici par exemple comment créer un nouveau lien (*LinkNode*) vers un nœud VOSpace :

```
curl -v -X PUT -d @newLink.xml -H "Content-type: text/xml" "http://localhost:8080/vospace/nodes/link"
```

Le fichier newLink.xml :

```
<node xmlns="http://www.ivoa.net/xml/VOSpace/v2.0" version="2.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="vos:LinkNode" uri="vos://cds-vospace.com!vospace/link">
    <target>vos://cds-vospace.com!vospace/folder</target>
    <properties>
        <property uri="ivo://ivoa.net/vospace/core#description">description</property>
    </properties>
</node>
```

S'agissant d'un lien à créer, il faut alors préciser la cible. Pour d'autres types de nœud, tels que des nœuds de données (*DataNode*) ou conteneurs (*ContainerNode*), il n'y a pas de cible à donner. Des propriétés peuvent être directement données au moment de la création du nœud.

Comme précédemment, la réponse contient la représentation du nœud créé, une propriété pour la date de création (*btime*), date de modification des métadonnées (*ctime*) et modification des données (*mtime*) sont automatiquement insérées. Il en va de même pour la taille des données qui est initialisée à 0 puisque aucune donnée n'est actuellement associée.

```
<vos:node xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" uri="vos://cds-vospace.
com!vospace/link" version="2.1" xsi:type="vos:LinkNode">
    <vos:properties>
        <property uri="ivo://ivoa.net/vospace/core#btime">06/11/2020 01:39:02</property>
        <property uri="ivo://ivoa.net/vospace/core#ctime">06/11/2020 01:39:02</property>
        <property uri="ivo://ivoa.net/vospace/core#mtime">06/11/2020 01:39:02</property>
        <property uri="ivo://ivoa.net/vospace/core#length">0</property>
        <property uri="ivo://ivoa.net/vospace/core#description">description</property>
    </vos:properties>
    <vos:target>vos://cds-vospace.com!vospace/folder</vos:target>
</vos:node>
```

***DeleteNode* :**

Cette fonction permet de supprimer un nœud ainsi que ses données associées. Si le nœud est un conteneur alors tous les sous nœuds seront eux aussi supprimés.

La méthode DELETE de HTTP est utilisée. La réponse est vide, avec un code 204 *No Content*.

```
curl -v -X DELETE "localhost:8080/vospace/nodes/link"
```

***Move/CopyNode* :**

Elles sont utilisées pour déplacer ou copier des nœuds déjà existants. Ces deux fonctionnalités s'exécutent de manière asynchrone, afin de mettre en place une négociation entre le client et le serveur à l'aide d'UWS.

```
curl -v -X POST -d @moveJob.xml -H "Content-type: text/xml" "http://localhost:8080/vospace/transfers"
```

Tout d'abord il faut envoyer en requête POST vers la route* s'occupant des transferts les informations de l'action à faire. Ces informations sont stockées dans un fichier XML : ici, moveJob.xml :

```
<vos:transfer xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" version="2.1">
<vos:target>vos://cds-vospace.com!vospace/folder/anotherFolder</vos:target>
<vos:direction>vos://cds-vospace.com!vospace/anotherFolder</vos:direction>
<vos:keepBytes>false</vos:keepBytes>
</vos:transfer>
```

Dans ce fichier, la cible doit être précisée. Elle correspond au nœud qui sera déplacé ou copié. Puis la direction c'est-à-dire l'endroit où sera stocké le nouveau nœud, doit être donnée. Enfin *keepByte* sert à informer le service si l'action est un déplacement ou une copie. S'il est à *True* cela signifie que le transfert sera une copie, et sinon cela sera un déplacement. Une fois la requête envoyée, le serveur répond avec une URL qui correspond à l'action. Par exemple :

```
Location: http://localhost:8080//vospace/transfers/1591877479593
```

```
curl -v "http://localhost:8080//vospace/transfers/1591877479593"
```

Avec cet URL il est possible d'obtenir des informations sur l'action.

Grâce au résultat de cette requête, nous obtenons des informations sur l'action telles que : sa date de création et son statut (c'est-à-dire s'il est en cours d'exécution ou non ou s'il est terminé avec une erreur ou sans).
Maintenant il faut lancer l'action. Pour cela il faut faire la requête suivante :

```
curl -v -d "PHASE=RUN" "http://localhost:8080/vospace/transfers/1591877479593/phase"
```

Cela permet de notifier le serveur que l'action doit commencer. À tout moment, l'état de l'action peut être vérifié avec la requête suivante :

```
curl -v "http://localhost:8080/vospace/transfers/1591877479593/phase"
```

La réponse spécifiera l'état actuel de l'action par exemple : complétée ou erreur. Une fois complétée, l'action a eu lieu et la copie ou le déplacement est terminé.

***PushToVoSpace/PullFromVoSpace* :**

Ces deux fonctionnalités permettent respectivement d'envoyer des données vers VOSpace et de récupérer des données depuis VOSpace. Elles aussi se déroulent de manière asynchrone. Le principe de fonctionnement est semblable à celui du déplacement : les requêtes sont les mêmes, seul le fichier XML change :

```
<vos:transfer xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" version="2.1">
<vos:target>vos://cds-vospace.com!vospace/mydata2</vos:target>
<vos:direction>pushToVoSpace</vos:direction>
<vos:view uri="ivo://ivoa.net/vospace/core#anyview"/>
<vos:protocol uri="ivo://ivoa.net/vospace/core#httpput" />
</vos:transfer>
```

L'attribut *keepBytes* n'est plus présent. Dans ce cas, c'est l'attribut « direction » qui précise l'action : ici *pushToVoSpace* (c'est-à-dire envoyer vers VOSpace ; la destination des données est la cible). Il faut aussi préciser une vue et un protocole. Actuellement le service ne supporte que *anyview* comme vue (c'est-à-dire, une qui accepte n'importe quel type de fichier et le stocke dans le système de données tel qu'il est envoyé) et le protocole httpput. Pour la récupération de données, il suffit de changer la direction en *pullFromVoSpace*.

Une fois toutes ces étapes faites, il faut récupérer l'URL qui permettra l'envoie ou la récupération. Pour cela, il faut récupérer le résultat de l'action nommé *transferDetails* :

```
curl -v "http://localhost:8080/vospace/transfers/1591881229175/results/transferDetails"
```

Ce qui retourne un document XML comme celui-ci :

```
<vos:transfer xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" version="2.1">
    <vos:target>vos://cds-vospace.com!vospace/mydata2</vos:target>
    <vos:direction>pushToVoSpace</vos:direction>
    <vos:protocol uri="ivo://ivoa.net/vospace/core#httpput">
        <vos:endpoint>http://localhost:8080//vospace/upload/mydata2</vos:endpoint>
    </vos:protocol>
</vos:transfer>
```

En réponse à cette requête le serveur va donc renvoyer une liste d'URLs possibles où le client peut transférer des données.

Ici, par exemple, il y a une URL où les données peuvent être envoyées. Maintenant il n'y a plus qu'à utiliser cette adresse pour faire le transfert.

Pour envoyer des données, Il faut donner le fichier que l'on veut envoyer :

```
curl -v -T mydata2 "http://localhost:8080//vospace/upload/mydata2"
```

Recevoir des données, se fait en précisant le nom de fichier où seront stockées les données réceptionnées :

```
curl -v -o mydata2 "http://localhost:8080//vospace/download/mydata2"
```

3 Mise en place du service

Afin de mettre en place le service, j'ai utilisé Gradle qui sert notamment à automatiser certaines tâches comme l'import de nouvelles bibliothèques ou l'ajout de plugins* et la vérification de prérequis tel que la version Java. Eclipse qui est l'environnement de développement* que j'ai utilisé durant mon stage, intègre un plugin pour Gradle permettant de préparer le projet en enregistrant automatiquement les chemins d'accès vers les bibliothèques et autres dépendances. Gradle permet aussi de s'occuper de la compilation du projet ainsi que l'exécution des tests unitaires

3.1 Architecture du service

Afin d'avoir une idée de l'architecture du service j'ai réalisé un diagramme UML*. Le diagramme de classe final est disponible en annexe. Afin d'avoir un service générique et extensible, plusieurs possibilités d'extension ont été mises en place. Il est ainsi facilement possible d'ajouter de nouvelles vues et de les intégrer au service en créant une nouvelle fabrique qui permet d'instancier une vue en fonction de son URI. De la même manière il y a une fabrique pour les nœuds qui peut être étendue afin de supporter de nouveaux types de nœuds. Il est aussi possible d'ajouter de nouveaux formats de représentation des nœuds. Actuellement les formats supportés sont : JSON et XML. Enfin pour la gestion du système de stockage, il est précisé dès la création du VOSpace. Pour l'instant, seul le système de fichier local est supporté. Mais il est simple d'implémenter et d'intégrer un nouveau système de stockage.

Les types de nœuds supportés par le service sont les nœuds de données qui correspondent aux fichiers, les nœuds de type conteneur qui sont équivalents à des dossiers, et les liens qui sont des nœuds qui pointent vers d'autres nœuds.

3.2 Implémentation du service

3.2.1 Gestion de la base de données (métadonnées)

La gestion de la base de données est gérée par H2. Il s'agit d'un système de gestion de base de données léger. Il permet de créer facilement une base de données avec une interface graphique web. La base de données correspond à un simple fichier stocké en local.

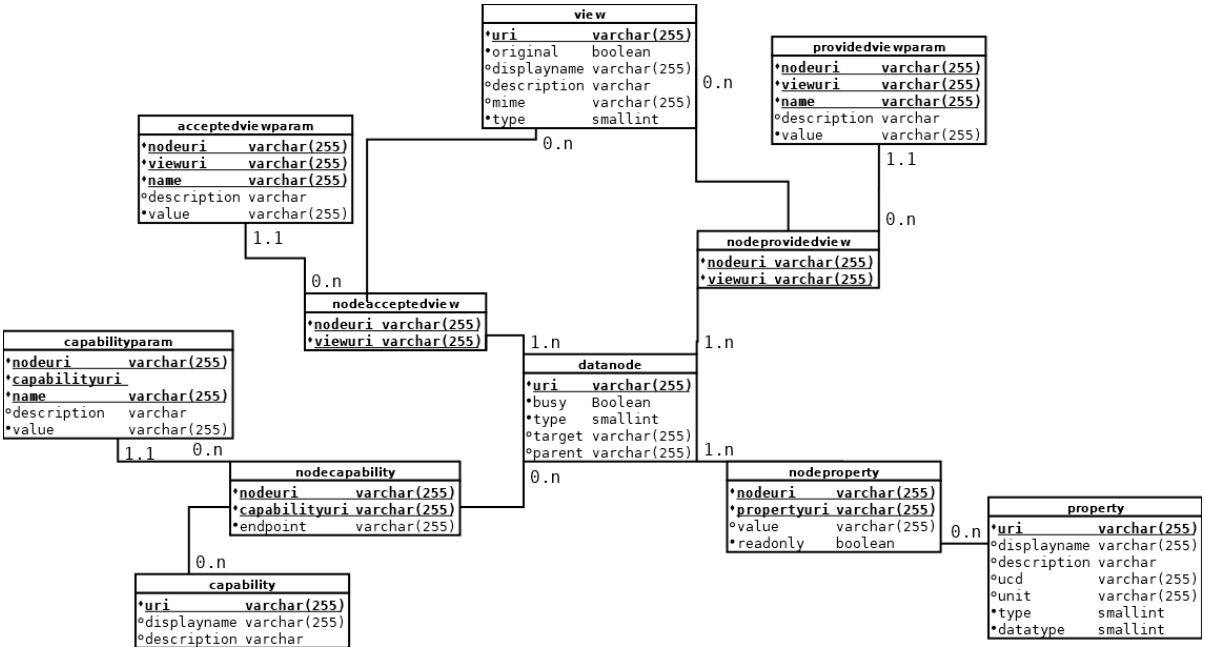


Figure 6 : MCD de la base de données des métadonnées

Toutes les métadonnées d'un nœud sont conservées dans cette base :

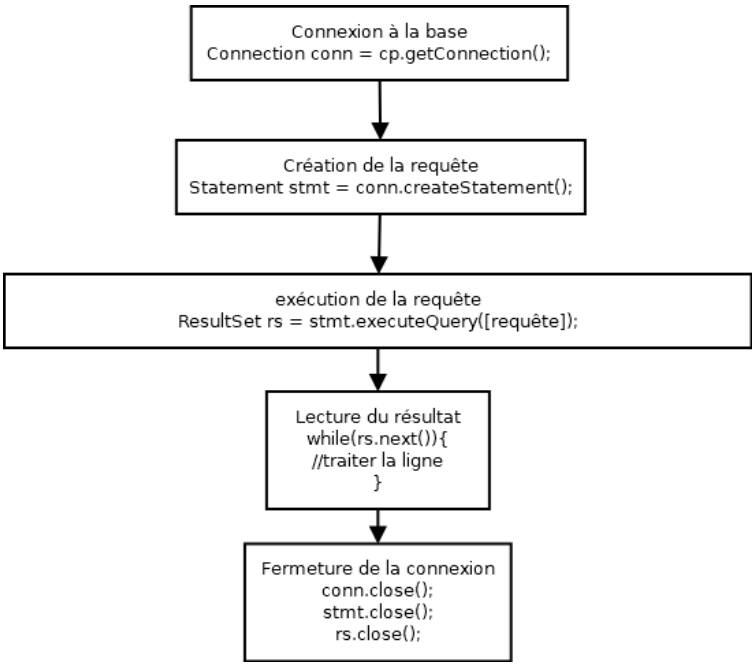
- La table *datanode* regroupe tous les nœuds quel que soit leur type, elle contient les colonnes suivantes :
  - « uri » qui permet d'identifier le nœud et constitue donc la clé primaire de cette table,
  - « *busy* » est un booléen qui indique si le nœud est actuellement accessible ou non. Le nœud peut ne pas être accessible si un transfert est en cours d'exécution.
  - Le type de nœud (par exemple : lien, nœud de données ou encore conteneur) est représenté par un nombre.
  - *Target* est l'uri de la cible d'un lien et est donc vide si le nœud n'est pas un lien. La cible devant exister, *target* est donc une clé étrangère sur la colonne « uri » de la table *datanode* elle-même.
  - « parent » est aussi une clé étrangère sur la colonne « uri » de la même table. Il identifie le parent direct du nœud, c'est-à-dire le conteneur dans lequel est ce nœud. Si le nœud est directement situé à la racine alors cette colonne est vide.
- La table *view* regroupe toutes les vues qui peuvent être utilisées dans le service, elles sont elles aussi identifiées par un URI. Elle est constituée des colonnes suivantes :
  - *original* indique si la vue conserve les données telles quelles ou non.
  - *displayname* est un nom destiné à l'affichage.

    - *mime* est le type de données correspondant à la vue : text/xml ou text/json par exemple. Le type permet de préciser si la vue peut être acceptée, fournie ou les deux.
- *Nodeacceptedview* et *nodeprovidedview* font le lien entre un nœud et une vue en récupérant en clé étrangère l'uri du nœud et celui de la vue. Ces deux attributs forment ensemble la clé primaire afin qu'un nœud n'ait pas deux fois la même vue. Grâce à cette table il est simple de récupérer toutes les vues associées avec un nœud.
- *Acceptedviewparam* et *providedviewparam* donnent des paramètres associés avec un couple vue, nœud, et des informations sur ce paramètre comme un nom, une description ainsi que sa valeur.
- La table *property* contient toutes les propriétés du service. Chacune d'elles est identifiée par un URI.
    - Tout comme les vues, elles ont un type qui désigne si la propriété est acceptée, fournie ou les deux.
    - Elles ont également un type de données (*datatype*). Par exemple : une chaîne de caractère (un texte), une date.
    - *displayname*, *description*, ucd (qui n'est actuellement pas utilisé) et *unit* sont des informations supplémentaires sur la propriété.
- *Nodeproperty* a le même fonctionnement que *nodeacceptedview* avec des colonnes en plus. Pour chaque propriété associée avec le nœud, il faut donner la valeur de la propriété dans la colonne « *value* ». La colonne « *readonly* » indique si la propriété est en lecture seule ou non.
- *Capability* regroupe toutes les interfaces tierces permettant d'interroger un nœud d'une manière particulière, par exemple : des requêtes SQL* sur un nœud conteneur avec des tables dans celui-ci. Ces interfaces sont identifiées par un URI et la table peut contenir des informations supplémentaires comme un nom d'affichage ou une description.
- *Capabilityparam* a le même fonctionnement que *acceptedviewparam.*
- *Nodecapability* est semblable à *nodeproperty* mais il n'y a pas de valeur ni de booléen mais une URL (*endpoin*t) qui permet d'envoyer les requêtes pour exploiter le contenu de ce nœud.

Afin que cette base soit utilisable par le service chaque table, a un équivalent sous la forme d'une classe Java. Ainsi lorsque les données sont récupérées depuis la base elles peuvent directement être stockées dans un objet qui a les mêmes attributs que la table. Ce procédé est appelé JavaBean.

Les requêtes vers la base se font par l'intermédiaire d'une classe qui s'occupera de récupérer toutes les informations et les renvoyer sous la forme d'un objet. Grâce à cela, dans le système il ne restera plus qu'à manipuler ces objets.

Pour chaque fonctionnalité de VOSpace qui a un lien avec les métadonnées, il y a une méthode dans le gestionnaire de base de données. En Java les requêtes vers la base de données se font en plusieurs étapes.



Tout d'abord, il faut se connecter à la base. Une fois connecté il faut créer la requête qui permettra d'interroger la base en SQL avec les informations nécessaires. Par exemple : "select * from datanode". Cette requête permet de récupérer tous les attributs de tous les nœuds dans la base. Chaque ligne retournée correspond à un nœud. Chaque colonne d'une ligne représente un attribut du nœud représenté. Il faudra alors traiter le résultat de la requête ligne par ligne, puis colonne par colonne afin de récupérer les informations de tous les nœuds du VOSpace.

Lorsque j'ai commencé la gestion de la base de données, j'ai utilisé une connexion unique, mais cela pouvait poser un problème. En effet, si plusieurs clients ont besoin d'accéder à la base de données en même temps il y aura un conflit : un des clients utilisera la connexion et les autres ne pourront pas pendant ce temps-là. Pour avoir plusieurs connexions à la base initialisées et disponibles en permanence, il faut utiliser ce que l'on appelle un *pool* de connexions. Lors de sa création, un *pool* de connexions crée directement plusieurs connexions qui pourront être récupérées (première étape). Une fois qu'une connexion est réinitialisée et rendue au *pool* de connexions, elle peut être réutilisée et est à nouveau disponible dans le *pool* de connexions. Le *pool* de connexions permet ainsi d'avoir de meilleures performances en n'ayant pas à recréer une nouvelle connexion à chaque requête. La création étant coûteuse en temps.

Un autre problème qui est apparu est le risque de fuite mémoire. En effet, je ne fermais la connexion à la base de données que lors d'un déroulement normal. Ce faisant, lors d'une erreur, la connexion n'était pas fermée. Afin d'éviter d'avoir à faire plusieurs vérifications pour fermer la connexion, j'ai utilisé un concept en Java appelé *try-with-ressources* que l'on pourrait traduire littéralement par « essayer avec les ressources ». Cela permet de ne pas avoir à fermer les connexions manuellement. Quand le traitement est terminé ou qu'il y a une erreur, les ressources sont automatiquement supprimées. Ci-dessous un exemple d'utilisation de ce concept :

```
try(
        Connection conn = cp.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select uri from datanode");
){
    //traitement des resultats
}
catch(Exception e) {
    //gestion des erreurs
}
```

Comme illustré ci-dessus, contrairement à un bloc « *try-catch* » ordinaire, les ressources à fermer automatiquement dès la sortie du bloc sont déclarées et initialisées entre parenthèses au niveau du « *try* ». Cela est rendu possible par le fait que les classes *Connection*, *Statement* et *ResultSet* implémentent l'interface java.lang.AutoCloseable. Une fois la connexion à la base de données récupérée et la requête exécutée, son résultat peut être traité, comme d'habitude, dans le bloc de traitement « *try* » (entre les accolades). De même, les erreurs sont toujours traitées dans le bloc « *catch* ».

Maintenant que les requêtes sont exécutées sans risque de fuite mémoire, un autre risque persiste : celui de l'injection SQL. Il s'agit d'une faille qui permet d'exécuter des requêtes qui ne sont pas initialement prévues par les développeurs : par exemple en récupérant un nœud, il pourrait être possible d'en même temps supprimer ou modifier un autre. Afin de se prémunir contre cette faille, j'ai changé la plupart des requêtes en requêtes préparées. Cela permet de neutraliser les caractères permettant de faire ces injections. Avec une requête non-préparée, la requête est créée par concaténation de chaîne de caractères. Cela signifie que les variables ne subissent aucune vérification lors de leur retranscription. Il est donc possible de fermer la requête prévue et d'en écrire une nouvelle qui modifierait ou détruirait en partie ou en totalité la base de données. Il pourrait aussi être possible de détourner des informations confidentielles. Avec une requête préparée, aucune concaténation n'est utilisée lors de la création de la requête. Elle est écrite en totalité avec « ? » qui substitue les variables. Les « ? » pourront être remplacés par les variables nécessaires grâce à des méthodes. L'exemple ci-dessous montre l'utilisation d'une requête préparée.

```
try (
        Connection conn = cp.getConnection();
        PreparedStatement stmt = conn.prepareStatement("select * from datanode where uri = ?");
    ){
        stmt.setString(1, uri);

        try(ResultSet rs = stmt.executeQuery();){
            //traitement des resultats
        }
}
catch(Exception e) {
    //gestion des erreurs
}
```

Pour insérer la variable, on utilise une méthode de type *set* en précisant en premier le numéro du point d'interrogation ainsi que la variable qu'il faut insérer. Il y a différents types de méthodes *set* qui permettent d'insérer différents types de variable : par exemple, des entiers, des chaînes de caractères. Ici c'est une chaîne de caractères.

Dans cet exemple, on peut observer qu'il y a un deuxième *try-with-resources*, en effet la ligne où la variable est insérée n'est pas une déclaration de ressource et donc ne peut pas être mis avant.

Maintenant comment peut-on récupérer un nœud avec toutes ses informations depuis la base de données en utilisant son URI ?

Tout d'abord il faut récupérer les informations de base. Pour cela, il faut interroger les informations de la table *datanode* puis en créer sa représentation objet avec ses données. En fonction du type du nœud, des métadonnées supplémentaires sont ajoutées. Si c'est un lien, alors il faut récupérer la cible. Si c'est un conteneur alors il faut récupérer ses fils, en récupérant tous les nœuds qui ont pour parent le nœud demandé. Si c'est un conteneur ou un nœud de données alors il faut récupérer les vues et aptitudes (*capability*) à l'aide des tables *nodeacceptedview*, *nodeprovidedview* et *nodecapability*. Toutes les propriétés sont récupérées de la même manière que les vues mais avec la table *nodeproperty*.

C'est aussi à ce niveau que le paramètre *detail* de la méthode GetNode est traité, s'il est à « min » alors seules les informations importantes sont conservées : type de données, cible si c'est un lien et fils si c'est un conteneur. S'il est à « *properties* » alors en plus des informations importantes, il y aura toutes les propriétés. Sinon toutes les informations seront renvoyées.

Comment se déroule une opération sur la base de données ? (Exemple de setNode)

Comme vu précédemment setNode permet d'ajouter ou de modifier des propriétés déjà existantes. Pour ce faire il faut avoir les métadonnées du nœud non modifié et les nouvelles métadonnées. Avant de mettre à jour la base de données, les anciennes et nouvelles métadonnées sont comparées entre elles :

- Si la nouvelle propriété existe déjà et qu'elle peut être modifiée (c'est-à-dire, pas en lecture seule) alors c'est une mise à jour de la propriété dans la base.
- Si la nouvelle propriété n'existe pas au niveau du nœud alors elle est créée dans la base de données avec la valeur correspondante.
- Si la propriété n'est pas supportée par le service alors une erreur est envoyée.

En plus de cela la propriété indiquant la date de modification des métadonnées est mise à jour à la date et heure actuelle.

3.2.2 Stockage des nœuds

Le stockage actuel des fichiers se fait sous la forme d'un simple système de fichier local où les nœuds de données sont des fichiers et les nœuds conteneur sont des dossiers. Les liens n'ont pas de représentation physique en dehors de la base de données. Lors de la création d'un nœud de données, un fichier vide est créé (éventuellement avec ses nœuds parents si non-existant). Par exemple : le nœud « vos://cds-vospace.com!vospace/folder/test_copy1 » a comme dossier parent « vos://cds-vospace.com!vospace/folder ». Donc dans le système de fichier, *test_copy1* sera dans un dossier nommé *folder*.

Il y a une fonction pour chaque fonctionnalité de VOSpace qui utilise les fichiers. Lorsqu'il y a un transfert, UWS est utilisé.

Puisque certaines actions doivent pouvoir être interrompues, il n'est pas possible d'utiliser les fonctions natives de copie/déplacement de Java qui ne peuvent être stoppées telles quelles. J'ai donc dupliqué les fichiers en les lisant et en écrivant un nouveau fichier bloc par bloc à l'endroit indiqué lors de la requête. Il est alors possible de vérifier à chaque itération l'état actuel de l'action. Lorsqu'elle est interrompue, la copie est arrêtée et le fichier qui était en cours de création est supprimé. Le déplacement d'un fichier se fait de la même manière mais avec une suppression du fichier source à la fin de la copie. Les copies se font de manière récursive afin de préserver la hiérarchie des dossiers dupliqués.

En ce qui concerne les transferts d'envoi ou de réception de fichiers, la négociation avec UWS permet d'obtenir des adresses vers lesquelles les données peuvent être envoyées. Dans notre système actuel la génération est simple, il y a une URL ressemblant à "http://-localhost:8080/vospace/upload/[path]" pour l'envoi de fichier et "http://localhost:8080-/vospace/download/[path]" pour la réception. Une fois l'action terminée, l'URL est générée et peut être récupérée. Maintenant il ne reste plus qu'à envoyer ou réceptionner les fichiers. Dans le cas où l'on utilise une URL qui n'a pas été générée, une erreur sera produite ; il n'est en effet pas autorisé d'envoyer ou de recevoir des fichiers sur une URL qui n'a pas été préparée auparavant.

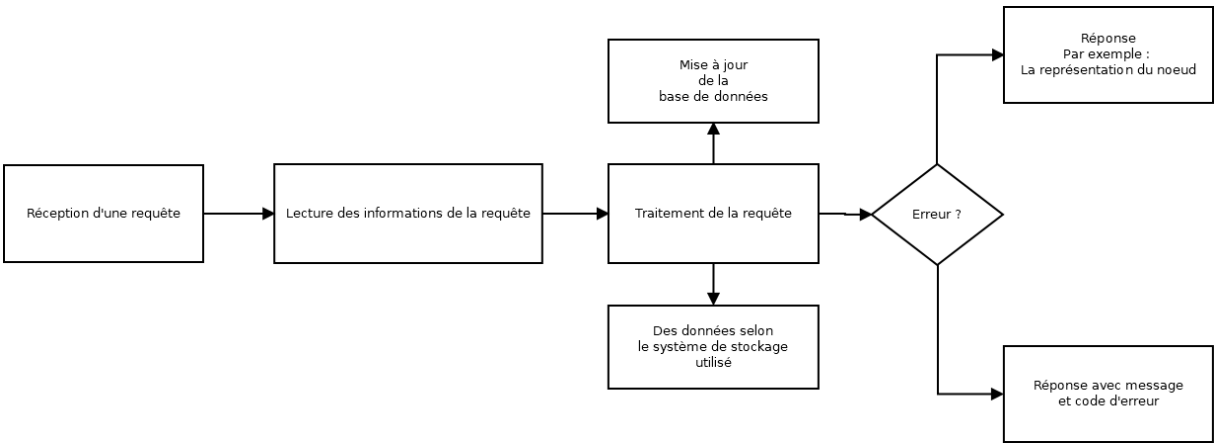
Le calcul de la taille des fichiers et dossiers se fait aussi à ce niveau. C'est donc au gestionnaire des données que revient cette tâche. Dans le cas des répertoires, ce calcul est fait de manière récursive.

Pour pouvoir ajouter de nouvelles manières de stocker des données, une interface Java est disponible. Elle permet d'imposer l'implémentation de toutes les fonctions qui sont nécessaires lors des transferts. Ainsi lors de la création d'un VOSpace, un nouveau système de stockage peut être renseigné et utilisé de manière parfaitement transparente.

Afin d'utiliser UWS, il faut implémenter la classe abstraite *JobThread* de la bibliothèque UWS-lib. Elle permet d'identifier et d'exécuter l'action à réaliser en fonction des informations stockées dans la requête : une copie, un déplacement, un *upload* ou encore un *download*.

3.2.3 Gestion des requêtes des clients vers VOSpace

Toutes les requêtes vers le service se font de la même façon :



Une fois la requête réceptionnée, les informations qu'elle contient (par exemple : type de fichier envoyé, destination, type de réponse attendu) sont lues et interprétées. Dès lors, en fonction de l'URL et de la méthode HTTP utilisées, la requête peut être exécutée. Le système va alors mettre à jour la base de données ou récupérer des données depuis celle-ci, et ainsi de même pour le système de stockage. À chaque étape, il y a une vérification s'il y a eu une erreur ou non. Dans le cas d'une erreur, le service répondra avec un code et un message d'erreur en fonction de la provenance de celle-ci.

Afin de gérer l'API HTTP, j'ai utilisé Vert.X (sélectionné au début du stage parmi les *frameworks* : DropWizard, Jersey et Vert.X). Il a une manière très intéressante de créer des URL vers lesquelles envoyer les requêtes. Il utilise un principe appelé fonction anonyme. Il s'agit d'une fonction qui n'a pas de nom. Par exemple :

```
this.router.route("/vospace/nodes/*").handler(routingContext -> {

    //traitement de la requête

});
```

Ici la route : « nomDeDomaine/vospace/nodes/[chemin] » est créée. Elle traitera toutes les requêtes vers cet URI quel que soit « [chemin] ». De cette manière, il est également possible d'ajouter des contraintes de type, comme par exemple « n'accepter que des fichiers XML, ou que des requêtes de type *Get* », …

Vert.X permet aussi d'avoir une gestion de l'envoi et de la réception de fichier. Lors de la réception de fichiers, ils sont stockés dans un dossier temporaire en attente de traitement. C'est à ce moment que le gestionnaire de fichier s'occupe de déplacer le fichier au bon endroit.

Les réponses peuvent être formatées en XML et JSON. Il est possible d'ajouter de nouveaux formats de réponse simplement à l'aide de deux classes. La première est une classe abstraite qui permet d'ajouter un nouveau format. La seconde classe est en réalité une interface permettant d'implémenter une fabrique qui sera utilisée pour instancier les formats en fonction du *header* « Accept ». Une fois instancié, un format est utilisé pour convertir un nœud ou encore d'autres données. La possibilité d'ajouter de nouveau format de conversion montre en quoi cette bibliothèque est générique et extensible.

Pour la génération des formats, j'avais d'abord utilisé une concaténation de chaînes de caractères mais cela n'était pas optimisé et il y avait un grand risque d'oubli de parenthèses ou balises. J'ai alors utilisé un arbre DOM* pour générer les sorties XML. Cela permet, à l'aide de fonctions, de créer un nouveau document XML qui peut alors être converti en texte puis envoyé en réponse au client. Par ailleurs, des vérifications sont effectuées afin de s'assurer de la validité des XML.

De même, pour le format JSON j'ai utilisé Vert.X qui contient des fonctions de création et manipulation de JSON.

Il y a aussi une méthode qui fait le processus inverse pour les XML. Ainsi le XML envoyé par un client peut être converti en nœud par exemple. Il n'y a alors pas à manipuler de XML au sein de VOSpace mais simplement des nœuds.

3.2.4 Gestion des erreurs

VOSpace indique que les erreurs devront être renvoyées au client sous la forme d'un message et d'un code d'erreur :

InternalFault : Indique qu'une erreur interne est survenue.
- *NodeNotFound* : Indique que le nœud demandé n'a pas été trouvé.
- *PermissionDenied* : Le client n'a pas les permissions pour accéder au nœud ou il essaie de modifier/supprimer une propriété qui est en lecture seule.
- *TypeNotSupported* : Indique que le type de nœud donné n'est pas supporté par le service.
- *ViewNotSupported* : La vue donnée n'est pas supportée par le service.
- *DuplicateNode* : Cette exception a lieu lorsqu'un client essaie d'ajouter un nouveau nœud qui a un URI qui existe déjà.
- *ContainerNotFound* : Indique qu'un nœud parent n'existe pas.
- Etc.

Toutes ces erreurs sont utilisées dans VOSpace et remontées jusqu'au gestionnaire des requêtes qui va s'occuper d'envoyer les messages et codes d'erreur en fonction de l'erreur qui a eu lieu.

Afin de vérifier les erreurs : *ContainerNotFound, ViewNotSupported* ou NodeNotFound, la base de données est interrogée. Si aucune ligne ne correspond au nœud/vue alors c'est que celui-ci n'existe pas. Il faut faire attention au fait que la base de données doit être synchronisée avec le système de fichier car la base de données est utilisée pour faire les vérifications et il n'y a alors pas besoin de faire de vérification d'existence ou non côté système de stockage.

Dans le code chacune de ces erreurs correspond à une exception Java qui hérite de l'exception de base de VOSpace qui est *VOSpaceException*. *VOSpaceException* correspond à *InternalFault* qui est une erreur interne au serveur. Le code et le message d'erreur renvoyés en réponse sont stockés dans ces exceptions.

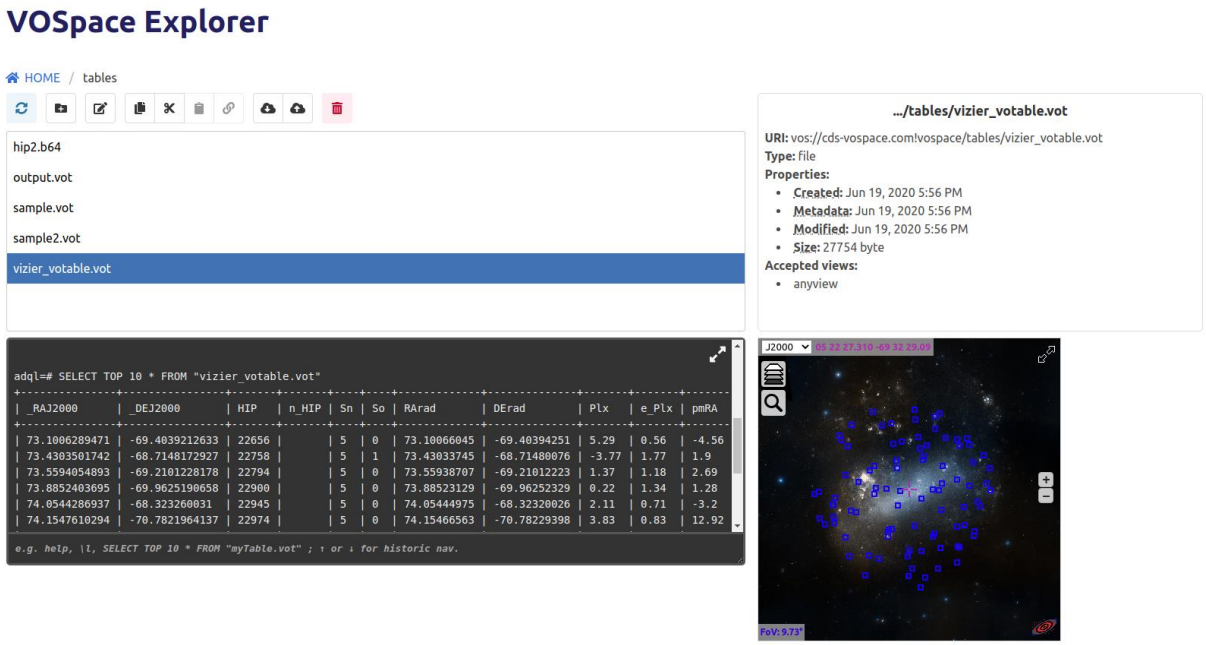
4 Perspectives

4.1 Nouvelles fonctionnalités

Le service actuel n'est pas encore totalement utilisable. La suite de l'implémentation sera faite par : soit un nouveau stagiaire, soit mon tuteur en entreprise Grégory Mantelet. Il manque encore certaines fonctionnalités précisées par VOSpace. C'est notamment le cas de « *pullToVoSpace* » et « *pushFromVoSpace* » qui concernent les transferts d'un VOSpace à l'autre. Les transferts client-serveur (*pushToVoSpace* et *pullFromVoSpace*) sont utilisables mais il n'est pas encore possible d'utiliser des formats d'imports et d'exports c'est-à-dire les vues : l'envoi et la réception se font sans modification du format.

Une autre fonctionnalité importante de VOSpace qui n'est pas encore présente est l'utilisation de *capabilities* qui permettent de relier certains nœuds du VOSpace à d'autre service : par exemple, un dossier contenant des VOTable (un format astronomique de table) pourrait être interrogé grâce à une interface de commande SQL à la même manière que sur une base de données. Il serait aussi possible d'afficher des points du ciel stockés sur VOSpace dans l'atlas du ciel Aladin lite.

Voici un exemple d'interface graphique avec l'intégration d'une interface de commande ADQL (équivalent astronomique de SQL) et Aladin lite :



Le système actuel est anonyme ce qui n'est pas souhaitable dans un système de stockage tel que VOSpace. Il faudrait mettre en place un système d'authentification afin de pouvoir reconnaître sur le service les personnes y accédant. De plus cette authentification permettra de pouvoir mettre en place une gestion des permissions afin d'avoir des droits d'accès aux nœuds ; tout le monde ne pourra alors pas accéder à tous les nœuds et les modifier. Il sera alors aussi possible d'utiliser la propriété « auteur » (*author*) afin d'avoir plus d'informations sur les nœuds.

Actuellement les données sont stockées en local sont la forme d'un système de fichier. Mais lorsqu'il y aura un grand nombre de données à stocker, il serait intéressant d'avoir un système de stockage dans un cloud.

Une autre fonctionnalité qui pourrait être intéressante serait la mise en place d'une corbeille afin d'avoir une possibilité de récupérer un fichier supprimé par mégarde.

4.2 Améliorations possibles

Le service peut avoir des améliorations. En effet, la gestion des erreurs pourrait être mieux optimisée en utilisant un outil de Vert.X qui permet de pouvoir rediriger vers une autre route s'occupant des erreurs. Actuellement chaque erreur est traitée par la route appelée lors de la requête.

Par ailleurs, le système de log ne donne pas assez d'information en cas d'erreur, il est difficile de trouver la source de celle-ci. C'est pour cela qu'il serait intéressant d'ajouter des logs plus explicites.

Enfin, il faudrait ajouter le support d'autres types de nœuds du standard. Pour le moment seuls les nœuds de données, les nœuds de type conteneur et les liens sont supportés. Or dans le standard certains nœuds plus spécifiques sont décrits, notamment : des nœuds de données non structurées (*unstructuredDataNode*) qui sont des nœuds dont le format est inconnu par le service et des nœuds de données structurées (*structuredDataNode*) qui eux ont un format reconnu. Cette amélioration va de pair avec la fonctionnalité des vues.

Il y a d'autres améliorations avec les transferts afin de les protéger face aux accès concurrents ainsi qu'une génération unique d'URL afin que deux personnes n'aient pas la même URL pour faire des transferts.

Conclusion

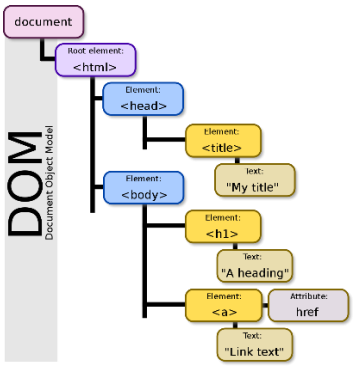
La librairie que j'ai implémentée met en place les bases du standard, mais grâce à son extensibilité et sa généricité, elle peut évoluer et être améliorée simplement et sans grandes contraintes. Elle est, en l'état actuel, fonctionnelle en tant que prototype mais pas utilisable tel quel car il manque encore certaines fonctionnalités importantes telles que : la possibilité de relier le VOSpace à d'autres services ou encore la gestion de format. Cette librairie, une fois totalement opérationnelle, pourra permettre une meilleure collaboration et occupera une place importante au sein de l'Observatoire virtuel.

Ce stage m'a permis d'avoir une première expérience dans le monde du travail et plus précisément dans l'informatique. De plus, avec les circonstances actuelles, j'ai dû télétravailler, et cela m'a permis de découvrir cette méthode de travail de plus en plus utilisée. J'ai aussi acquis de nouvelles compétences en utilisant notamment de nouvelles bibliothèques. J'ai également appris à utiliser de nouveaux outils : par exemple Gradle. Enfin j'ai pu appliquer ce que j'ai appris en cours comme par exemple : les *Design Pattern* afin de répondre à des problèmes de conception spécifique, la modélisation UML pour avoir une réflexion sur l'architecture du logiciel avant de commencer à l'implémenter ou encore la création de base de données. Enfin j'ai pu participer à des réunions régulières avec mon tuteur en entreprise afin de présenter mon avancée et d'en discuter. J'ai ainsi pu avoir une expérience de communication en entreprise.

Enfin, grâce à ce stage j'ai pu confirmer mon envie de travailler dans un secteur tel que l'informatique et de poursuivre mes études en école d'ingénieur. Ce stage m'a aussi montré l'éventualité de travailler dans un secteur de recherche qui n'est pas une possibilité que je rejette.

Glossaire

Arbre DOM : *Document Object Model* est une interface de programmation qui permet de lire et modifier des documents structurés de manière arborescente tels les documents formatés en que XML ou HTML. Il permet donc par extension de créer des documents XML. DOM permet de gérer une hiérarchie avec des nœuds sous forme d'arbre. Ci-dessous, un exemple montrant un arbre avec toute la hiérarchie de nœud :



Asynchrone : Une requête asynchrone est une requête qui se déroule sans que le client ne soit connecté au service.

Cloud : Cela correspond à un service de stockage distant (par exemple AWS chez Amazon).

Environnement de développement : ensemble d'outils qui permet de développer des logiciels (par exemple : Eclipse, Visual Studio).

HTTP : L'*HyperText Transfer Protocol* est un protocole de communication entre un client et un serveur. HTTP met en place différentes méthodes qui permettent de préciser ce que doit faire la requête. Les méthodes utilisées dans VOSpace sont : GET, POST, PUT et DELETE.

Interface : Dispositif permettant l'interaction entre différents acteurs. L'interface donne une manière unique d'accéder à certaines ressources. Ainsi aucun des acteurs n'a à faire attention à ce qu'il se passe de l'autre côté de l'inferface utilisée.

IVOA : *International Virtual Observatory Alliance*, un consortium qui met en place des standards afin d'avoir une interopérabilité entre les services.

Le site de l'IVOA : http://ivoa.net/

JSON : *JavaScript Object Notation* est un format de données qui permet de stocker de l'information de manière structurée. Il contient un ensemble de couples clé-valeur avec une hiérarchie. Voici un exemple de JSON :

```
{
  "uri" : "vos://cds-vospace.com!vospace/",
  "type" : "ContainerNode",
  "nodes" : [ {
    "uri" : "vos://cds-vospace.com!vospace/uws",
    "type" : "ContainerNode"
  } ]
}
```

On peut observer chaque couple clé-valeur, la valeur pouvant être un tableau d'autres clé-valeur.

Métadonnée : Informations associées à une donnée. Par exemple : une description, un sujet, …

Micro-service : C'est un service petit et léger ayant un faible couplage entre les différentes fonctionnalités.

Nœud : C'est le modèle de donnée de base au sein de VOSpace Il peut s'agir d'un lien, d'un fichier ou encore d'un dossier.

Plugins : Module permettant d'ajouter de nouvelles fonctionnalités à un outil ou programme.

Propriété : Une propriété est une information permettant de décrire un aspect particulier d'un nœud (description, date de création, taille, …).

Protocole : Un protocole spécifie des règles de communication entre un client et un serveur.

RESTful : RESTful est une architecture logicielle définissant des contraintes, dont notamment :
- les réponses doivent être formatées (par exemple : XML, JSON).
- Les responsabilités sont partagées entre serveur et client.
- Pas de conservation de l'état d'une connexion une fois déconnecté.
- les réponses du serveur doivent contenir assez d'information pour manipuler les éléments.
- Les requêtes doivent contenir assez d'information pour que le serveur puisse exécuter la manipulation.

Route : Une route est un chemin à partir duquel il est possible d'exécuter des requêtes.

Synchrone : Une requête synchrone est une requête qui se déroule alors que le client est connecté au service tout au long de celle-ci.

UML : L'*Unified Modeling Language* est un langage de modélisation graphique permettant de visualiser/concevoir l'architecture d'un logiciel informatique d'une manière normalisée.

URI : *Uniform Resource Identifier*. Il s'agit d'une chaîne de caractères permettant d'identifier de manière unique une ressource. C'est de cette manière que VOSpace a choisi d'identifier un nœud, une propriété ou une vue.

UWS : L'*Universal Work Service* est un standard de l'IVOA qui spécifie comment exécuter des tâches asynchrones. Dans VOSpace, il est utilisé pour gérer la négociation de certaines requêtes (ex : téléchargement de fichier).

Document du standard UWS :
http://www.ivoa.net/documents/UWS/20101010/REC-UWS-1.0-20101010.pdf

UWS-lib :
http://cdsportal.u-strasbg.fr/uwstuto/index.html

VOSpace : Standard de l'IVOA spécifiant comment accéder et manipuler des données.

Le document du standard :
http://www.ivoa.net/documents/VOSpace/20180620/REC-VOSpace-2.1.pdf

Vue : Dans VOSpace une vue correspond à un format de fichier supporté par le service (Par exemple : VOTable, FITS qui sont des formats de fichier astronomique).

XML : L'*Extensible Markup Language* est un format de données qui stocke l'information de manière structurée à l'aide de balises. Voici un exemple de fichier XML :

```
<node xmlns="http://www.ivoa.net/xml/VOSpace/v2.0" version="2.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="vos:LinkNode" uri="vos://cds-vospace.com!vospace/[path]">
    <target>vos://cds-vospace.com!vospace/[path]</target>
    <properties>
        <property uri="ivo://ivoa.net/vospace/core#description">description</property>
    </properties>
</node>
```

On peut y observer la hiérarchie : *node* a pour fils *target* et properties. *Properties* a pour fils *property*. Le balisage est visible. Des schémas peuvent être précisés afin de pouvoir vérifier la forme du XML.

Bibliographie/sitographie

Vert.X : Gestion du côté API HTTP
https://vertx.io/

H2 : Gestion de la base de données
https://www.h2database.com/html/main.html

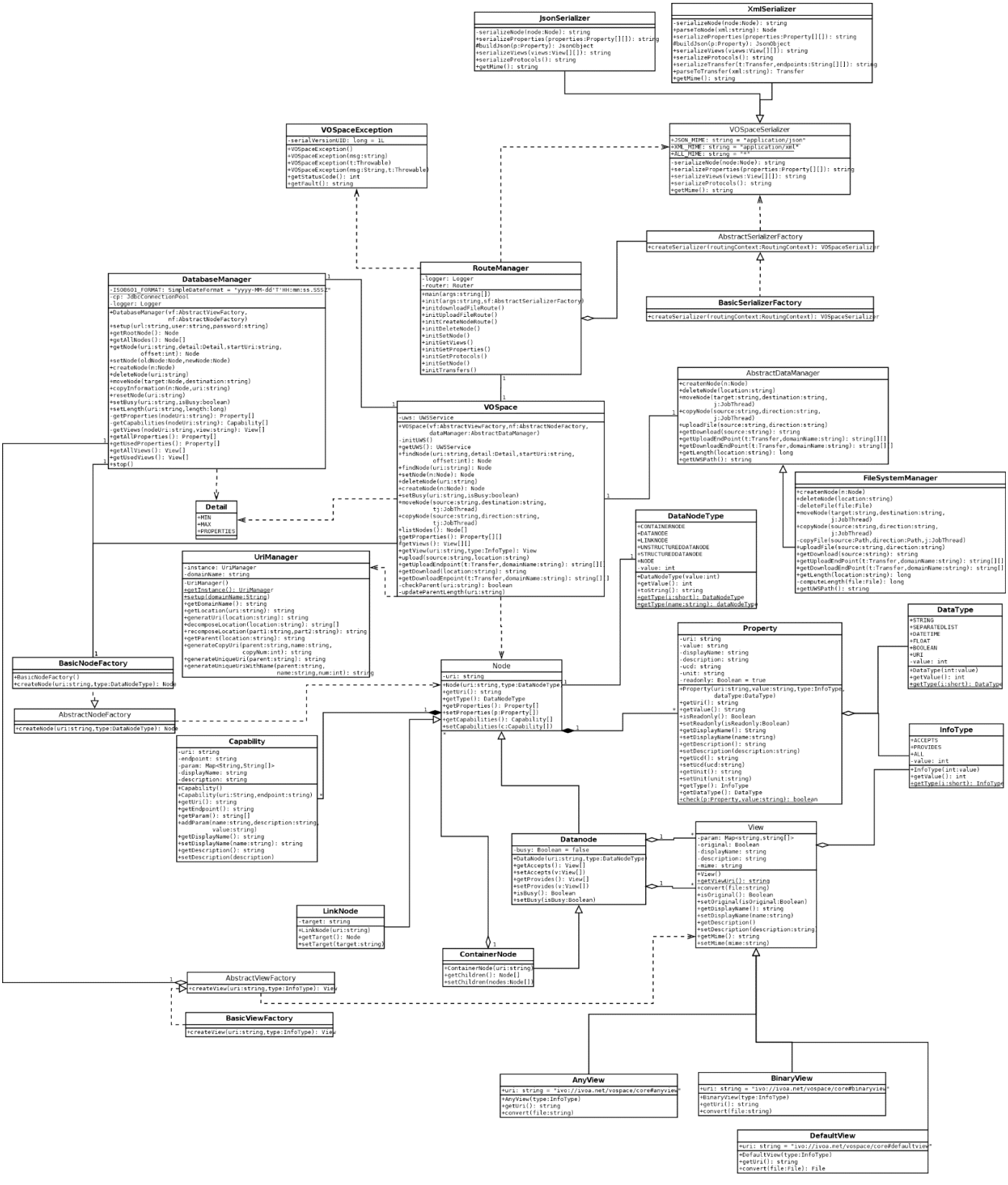
Gradle : Gestion des dépendances, bibliothèques et tests unitaires
https://gradle.org/

Bibliothèque UWS-lib : Gestion asynchrone des transferts
http://cdsportal.u-strasbg.fr/uwstuto/index.html

Le standard VOSpace :
http://www.ivoa.net/documents/VOSpace/20180620/REC-VOSpace-2.1.pdf

Le site de l'IVOA :
http://ivoa.net/

Annexe



Annexe 1 : Diagramme de classe du VOSpace

Ce diagramme montre toute l'architecture du service que j'ai implémenté. On peut notamment observer toutes les classes abstraites et interfaces (classes qui n'ont pas leur nom en gras) qui permettent l'extensibilité du service. Certains liens « *use* » et les classes héritant de « *VOSpaceException* » ont été supprimés pour des raisons de lisibilité.