



Module D314 – Document de conception

|  |  |  |
| --- | --- | --- |
| Suivi des versions | | |
| Date | **Version** | **Objet de la version** |
| 19/11/2015 | 01 | Création du document |
| 24/11/2015 | 02 | Ajout de la partie « Détails de la partie WS » |
|  |  |  |

Sommaire

[Description 3](#_Toc435562236)

[Architecture générale 4](#_Toc435562237)

[Détails de la partie Web-service 5](#_Toc435562238)

[Architecture 5](#_Toc435562239)

[Principaux fichiers du projet 5](#_Toc435562240)

[Classe Dimension 5](#_Toc435562241)

[Ce type complexe (Au sens XSD) sera utilisé en tant que paramètre d’appel ou de réponse du web-service, sous forme de liste d’objet. 5](#_Toc435562242)

[Méthodes exposées (Web méthodes) 5](#_Toc435562243)

[Méthodes de combinaisons 5](#_Toc435562244)

[Algorithmes de calcul 6](#_Toc435562245)

[Signatures et type SOAP-XML 8](#_Toc435562246)

[Gestion du cache 8](#_Toc435562247)

[Logs applicatives 10](#_Toc435562248)

[Logs techniques 11](#_Toc435562249)

[Installation et configuration 11](#_Toc435562250)

[Environnement utilisé 11](#_Toc435562251)

[Description du projet NetBeans 11](#_Toc435562252)

[Librairies tierces 11](#_Toc435562253)

[Tests et mise en œuvre 11](#_Toc435562254)

[Build et déploiement 11](#_Toc435562255)

[Méthodes de tests 13](#_Toc435562256)

[Détails de la partie client 14](#_Toc435562257)

[Présentation de l’interface de l’optimiseur ODE 14](#_Toc435562258)

[Détails des objets 14](#_Toc435562259)

[Exemple d’utilisation de l’optimiseur 15](#_Toc435562260)

[Etape 1 : Sélection du server 15](#_Toc435562261)

[Etape 2 : Appui sur le bouton Connexion 15](#_Toc435562262)

[Etape 3 : Sélection cube de données 16](#_Toc435562263)

[Etape 4 : Sélection espace alloué 16](#_Toc435562264)

[Etape 5 : Sélection Algorithme Metropolis ou Matérialisation 17](#_Toc435562265)

[Etape 6 : Fin d’optimisation 17](#_Toc435562266)

[Mode déconnecté 18](#_Toc435562267)

[Mode Full-WebService 18](#_Toc435562268)

[Bouton Purge Cache 19](#_Toc435562269)

[Mise en œuvre Technique 19](#_Toc435562270)

[Ajout de la référence au Webservice 19](#_Toc435562271)

[Ajout de l’adresse du Webservice 19](#_Toc435562272)

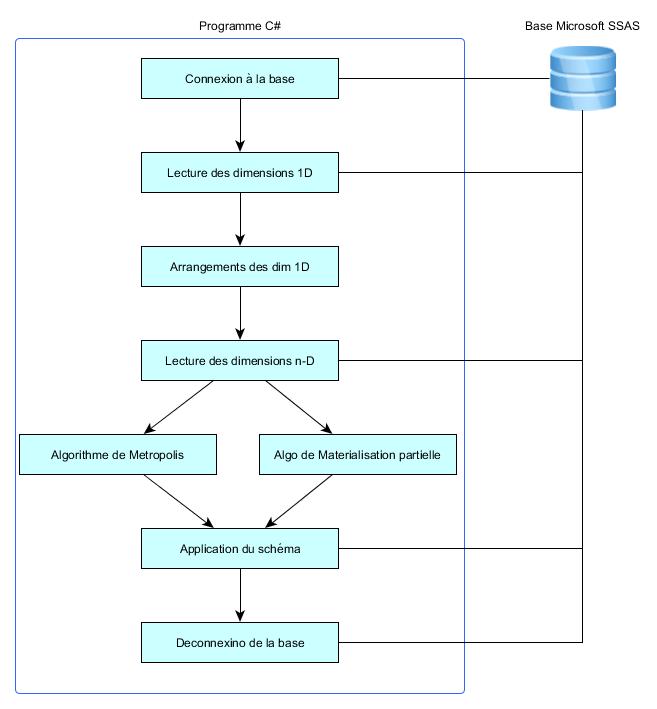
[Implantation dans le code C# 20](#_Toc435562273)

# Description

Réalisé dans le cadre de notre projet d’étude de Master 2 SIID, le projet « **Optimisation des Données de l’Entrepôt (ODE)** » consiste à utiliser les techniques mathématiques vues dans le Master afin de construire l'entrepôt de données de manière optimal, en termes de temps de réponse à l’interrogation des cubes et d’occupation disque :

<https://github.com/M2SIID-ODE/Projet_ODE>

Dans le projet ODE, un seul programme en C# se charge à la fois des interactions avec la base de données SQL Server OLAP (Microsoft SSAS) et des calculs d’optimisations du schéma de base de données :



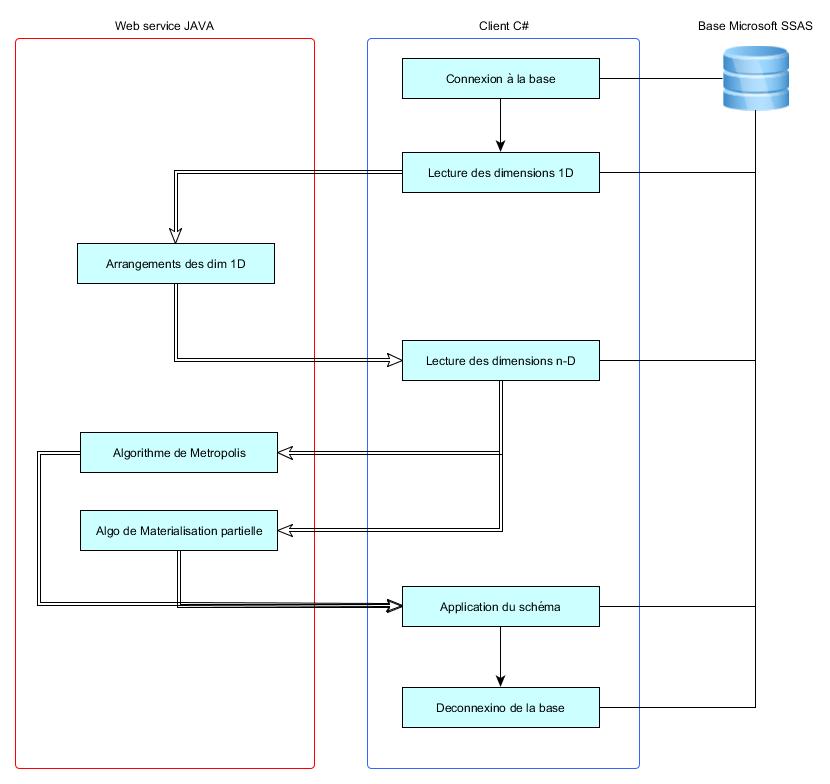
Dans le cadre du module D314 Web-service, nous avons migré les fonctionnalités « cœur de calcul » vers un Web-service Java. Ainsi, les processus consommateurs de ressources machine (Principalement charge CPU) peuvent être déportés sur un serveur pourvu d’une forte puissance de calcul, tandis que la base OLAP reste sur une machine plutôt orientée accès disque et mémoire RAM.

De plus, la portabilité de la JVM Oracle permettra une exploitation sur différents systèmes d’exploitation, par exemple sur la plateforme Amazon Web Service AWS- EC2 (Instances types C3 et C4 orientées calculs intensifs) tandis que la base OLAP reste cantonnée à l’environnement Windows.

# Architecture générale

Quatre méthodes ont été migrées vers un web-service Java :

* **Calcul de schéma optimal par la méthode de Metropolis**
* **Calcul de schéma optimal par la méthode de matérialisation partielle**
* **Arrangement des dimensions du cube relationnel de la base de données OLAP** (2 Méthodes liées)



# Détails de la partie Web-service

## Architecture

### Principaux fichiers du projet

|  |  |
| --- | --- |
| Nom | Description |
| Dimension.java | Définition de la classe Dimension, identique au client C# |
| DimensionUtils.java | Méthodes exposées |
| DimensionTest.java | Méthode stub pour le test des méthodes exposées |
| OdeService.java | Interface du Web-service |
| OdeServiceImpl.java | Implémentation de l’interface du Web-service |
| SqliteSql.java | Méthodes liées au cache de résultats |
| CacheWebServiceOde.db | Fichier de base de données SQLite du cache de résultats du web-service |
| log4j.properties | Fichier de configuration du logger log4j |

### Classe Dimension

Dans le source ***Dimension.java***, on définit la classe Dimension par :

* Un nom, sous la forme ***<Dimension 1D> \* <Dimension 1D> \* … \* <Dimension 1D>***
* Un compteur du nombre d’items dans cette dimension
* Une taille unitaire d’un item de cette dimension
* Un ordre N de la dimension N-D

La définition de classe est identique à celle du client C#, afin de ne pas introduire d’opération de conversion superflue.

### Ce type complexe (Au sens XSD) sera utilisé en tant que paramètre d’appel ou de réponse du web-service, sous forme de liste d’objet.

|  |
| --- |
| *NOTE : Afin d’être visibles dans le XSD du web-service, les champs de la classe Dimension ont dû être spécifiés PUBLIC, rendant inutile les getters / setters de la classe.* |

### Méthodes exposées (Web méthodes)

Dans le source ***OdeService.java***, on déclare les méthodes exposées par le Web-service :

|  |  |  |  |
| --- | --- | --- | --- |
| Nom | Arguments | Type de retour | Description |
| GetCombinaisons | List<Dimension> | List<Dimension> | Cf. section Méthodes de combinaisons |
| TestGetCombinaisons | - | boolean | Cf. section Méthodes de test |
| GetCombinaisonsIndex | List<Dimension> | List<String> | Cf. section Méthodes de combinaisons |
| TestGetCombinaisonsIndex | - | boolean | Cf. section Méthodes de test |
| Metropolis | List<Dimension>  double  Integer | List<Integer> | Cf. section Algorithmes de calcul |
| TestMetropolis | - | boolean | Cf. section Méthodes de test |
| MaterialisationPartielle | List<Dimension>  double | List<Integer> | Cf. section Algorithmes de calcul |
| TestMaterialisationPartielle | - | boolean | Cf. section Méthodes de test |
| CleanCache | - | boolean | Cf. section Gestion du cache |

### Méthodes de combinaisons

**Méthodes de GetCombinaisons**

Dans le source ***DimensionUtils.java***, on définit une méthode « chapeau » ***GetCombinaisons***.



Cette méthode est le point d’entrée pour la méthode ***FunctionGetCombinaisons***, qui sera appelée de façon récursive pour construire une liste de combinaisons de dimensions ≥ 1 (Paramètre In / Out ***listCuboides***) ainsi qu’une liste de noms (Paramètre In / Out ***index\_cuboides***) à partir d’une liste de dimension 1-D (Paramètre In ***listDim1D***)



Les autres paramètres ***profCourante***, rang et ***prefix\_index*** sont utilisés seulement pour la récursivité.

La méthode ***GetCombinaisons*** va retourner la liste des combinaisons possibles de dimensions ≥ 1-D

**Méthodes de GetCombinaisonsIndex**

Dans le source ***DimensionUtils.java***, on définit une méthode « chapeau » ***GetCombinaisonsIndex***.



L’initialisation puis l’appel de la méthode unique ***FunctionCombinaisonIndex*** seront les mêmes que pour ***GetCombinaison***, mais l’argument retourné sera la liste de String ***index\_cuboides***

Côté client, l’appel successif (Ou simultanée en asynchrone) des deux méthodes ***GetCombinaisons*** et ***GetCombinaisonsIndex*** doit toujours aller de pair.

|  |
| --- |
| *NOTE : Idéalement, une seule méthode aurait suffi puisque les deux méthodes GetCombinaisons et GetCombinaisonsIndex font les mêmes opérations, et ne diffèrent que par leur type de retour.*  *Mais la complexité de gestion des multiples retours pour une seule web-méthode (Metaclasse englobant un String et un objet de type Dimension) nous a contraints à cette scission.* |

### Algorithmes de calcul

**Méthodes de Metropolis**

Dans le source ***DimensionUtils.java***, on définit une méthode « Chapeau » Metropolis.



La méthode calcul une solution optimale avec l’algorithme de matérialisation partielle, appliqué aux paramètres d’entrée de la méthode :

* Liste des dimensions ***listCuboides*** qui a été générée par les web-méthodes ***GetCombinaisons*** puis retravaillée par le client pour renseigner les champs manquant au travers d’une requête sur la base OLAP
* Un seuil de poids ***seuil\_poids*** (En Octets) pour « arrêter » le calcul lorsque la taille mémoire maximale a été alloué.
* Un nombre maximal ***nb\_boucle*** d’itération de boucle, car cet algorithme va générer une suite de nombres aléatoires.

Cette méthode est le point d’entrée pour la méthode ***FunctionMetropolis*** qui implémente le cœur de l’algorithme de calcul.



|  |
| --- |
| *NOTE : La méthode FunctionMetropolis est PRIVATE car elle ne doit pas être accessible de l’extérieur. En effet, un ensemble d’initialisation et de conversion est obligatoire, et sont effectuées dans la méthode exposée Metropolis.* |

La méthode ***Metropolis*** va retourner sous forme de liste de flags booléens la « meilleure » solution qui a été trouvée dans les conditions fixées de nombre de boucle et d’occupation mémoire.

**Exemple**



Equivaut à dire que la solution optimale consiste à matérialiser uniquement les dimensions N-D suivantes : D3 et DN

|  |
| --- |
| *NOTE : Le lecteur pourra se reporter au document de conception du projet ODE, section Optimiseur utilisant l’algorithme de Metropolis pour connaitre le fonctionnement détaillé de cet algorithme (Cliquer sur le bouton « RAW » pour lancer le téléchargement du fichier en local)*  [*https://github.com/M2SIID-ODE/Projet\_ODE/blob/master/Livrables/Documentation/ProjetODE\_Livrable\_Document\_Conception.docx*](https://github.com/M2SIID-ODE/Projet_ODE/blob/master/Livrables/Documentation/ProjetODE_Livrable_Document_Conception.docx) |

**Méthodes de MaterialisationPartielle**

Dans le source ***DimensionUtils.java***, on définit une méthode « Chapeau » ***MaterialisationPartielle***.





La méthode calcul une solution optimale avec l’algorithme de matérialisation partielle, appliqué aux paramètres d’entrée de la méthode :

* Liste des dimensions ***listCuboides*** qui a été générée par les web-méthodes ***GetCombinaisons*** puis retravaillée par le client pour renseigner les champs manquant au travers d’une requête sur la base OLAP
* Un seuil de poids ***seuil\_poids*** (En Octets) pour « arrêter » le calcul lorsque la taille mémoire maximale a été alloué.

Cette méthode est le point d’entrée pour la méthode ***FunctionMaterialisationPartielle*** qui implémente le cœur de l’algorithme de calcul.

|  |
| --- |
| *NOTE : La méthode FunctionMaterialisationPartielle est PRIVATE car elle ne doit pas être accessible de l’extérieur. En effet, un ensemble d’initialisation et de conversion est obligatoire, et sont effectuées dans la méthode exposée MaterialisationPartielle.* |

La méthode ***MaterialisationPartielle*** va retourner sous forme de liste de flags booléens la « meilleure » solution qui a été trouvée dans les conditions fixées de nombre de boucle et d’occupation mémoire, comme la méthode ***Metropolis***.

|  |
| --- |
| *NOTE : Le lecteur pourra se reporter au document de conception du projet ODE, section Optimiseur utilisant l’algorithme de matérialisation partielle pour connaitre le fonctionnement détaillé de cet algorithme (Cliquer sur le bouton « RAW » pour lancer le téléchargement du fichier en local)*  [*https://github.com/M2SIID-ODE/Projet\_ODE/blob/master/Livrables/Documentation/ProjetODE\_Livrable\_Document\_Conception.docx*](https://github.com/M2SIID-ODE/Projet_ODE/blob/master/Livrables/Documentation/ProjetODE_Livrable_Document_Conception.docx) |

### Signatures et type SOAP-XML

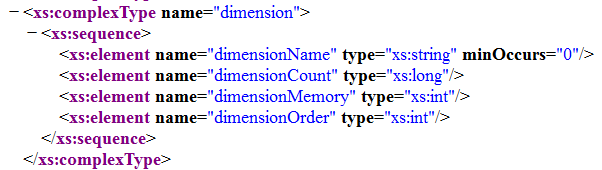
Les « signatures » des méthodes du web-service (Noms des méthodes, paramètres et retours) sont accessibles dans un unique fichier WSDL, auto-généré par le serveur Glassfish :

<http://127.0.0.1:8080/WebServiceOde/OdeServiceImplService?WSDL>

De plus, la description des types et des structures utilisés par le web-service est faite dans le schéma XSD, référencé dans le WSDL :

<http://127.0.0.1:8080/WebServiceOde/OdeServiceImplService?xsd=1>

On y voit notamment la définition de la classe « Dimension », traité comme un type complexe, avec les membres PUBLIC définis dans les sources Java de la classe éponyme :



### Gestion du cache

#### Présentation

En fonction du nombre de dimensions (Et du nombre de tirages aléatoires pour Metropolis) les calculs peuvent être longs, alors que le résultat attendu est déterministe. Même les tirages au sort sont « constants » entre deux exécutions, car le germe du générateur aléatoire est identique. Pour chaque algorithme : « Même entrées, même sorties »…

#### Fonctionnement

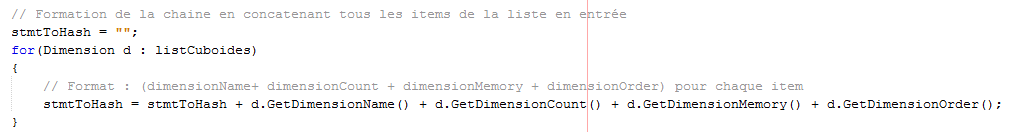
Le mécanisme de cache vient s’intercaler entre les appels aux méthodes de calcul Metropolis ou matérialisation partielle (Icone ***Web Service request*** du schéma) et l’envoi de la réponse par le web-service (Icone ***Web Service response*** du schéma)

Comme vu précédemment (cf. Section Méthodes exposées) chaque enregistrement d’un objet de type dimension contient notamment un nom, chaine de caractère qui est la concaténation de toutes les dimensions 1-D qui la constitue. Si on prend un exemple de 10 dimensions 1-D, la chaine la plus longue fera plusieurs centaines d’octets.

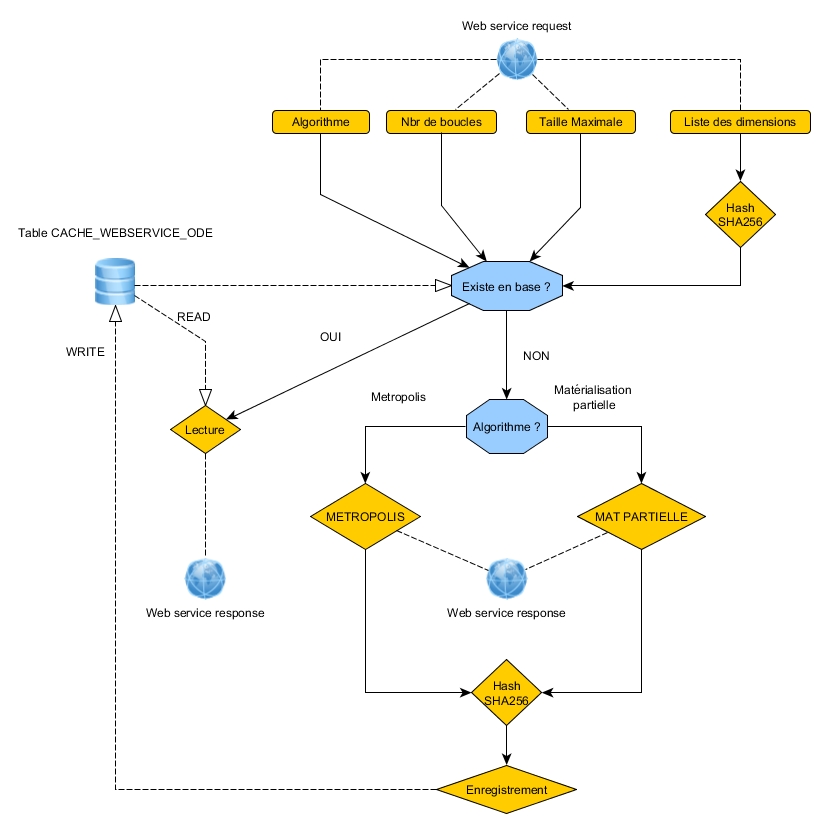
L’appel des méthodes Metropolis ou Matérialisation partielle se faisant avec une liste de ces objets (55 en restant avec 10 dimensions 1-D) on arrive à quelques kilo-octets par liste, à stocker dans la cache pour chaque jeu de paramètre d’appel.

Plutôt que de stocker l’ensemble de la liste des dimensions, on réalise donc un hash SHA256 de leur concaténation pour la « compresser », de façon destructive mais unique (Opération ***Hash SHA256*** du schéma) Ainsi, deux listes de dimensions qui diffèrent d’un seul caractère (Ou propriété) auront des hash différents, et pour un « prix » mémoire de seulement 256 bits par liste de dimensions, soit 32 octets (Au lieu de plusieurs kilo-octets)

***Source****: SqliteSql.Java*



Voici une vue d’ensemble de la logique du cache que nous gérons dans le web-service (Ce schéma est commenté à la suite) :



Les paramètres d’entrées sont cherchés dans la base de données du cache :

* Directement pour le type d’algorithme
* Directement pour le nombre de boucles (Toujours à 0 pour la matérialisation partielle, car non-pertinent pour cet algorithme)
* Directement pour la taille maximale allouée
* Après calcul du hash SHA256 pour la liste de dimensions

Si ces 4 éléments ne se retrouvent pas à l’identique dans la table CACHE\_WEBSERVICE\_ODE, alors on lance le calcul adéquat selon l’algorithme demandé, puis on enregistre le résultat en table (« Initialisation »)

Sinon, on ne déroule pas le calcul, mais on retourne la valeur de résultat lue en base, enregistrée lors d’une initialisation précédente.

#### Remise à zéro du cache

Une fonction de remise à zéro du cache a été prévue dans le web-service afin de remettre à zéro manuellement le stockage du cache : La méthode ***CleanCache*** réalise un DELETE de l’ensemble de la table CACHE\_WEBSERVICE\_ODE. Dans le client C#, la même méthode peut-être appelée à la demande de l’utilisateur.

#### Accès à la base de données du cache

L’accès à la base de données SQLite peut se faire à partir d’un client graphique, comme ***SQLiteDatabaseBrowser*** présenté dans le devoir du D314 :

<http://sqlitebrowser.org/>

Pour mieux voir le mécanisme de cache lors de l’exécution du web-service, il faut se connecter au fichier de base de données ***CacheWebServiceOde.db*** déployé sur Glassfish, et non celui du projet NetBeans. Dans notre configuration de déploiement (Serveur Glassfish local sur Windows 7 Pro) ce fichier est stocké dans :

***\Sources\WebService\WebServiceOde\build\web\WEB-INF\classes***

### Logs applicatives

On utilise l’API existante ***log4j 1.2*** (Et non 2.X) afin de générer et gérer facilement les traces du programmes, car les sorties « Console » de NetBeans ne sont pas adaptés au contexte de production.

<https://logging.apache.org/log4j/1.2/>

Le niveau de « finesse » des logs générées et leur présentation peut être changée au travers du fichier ***log4j.properties***  situé :

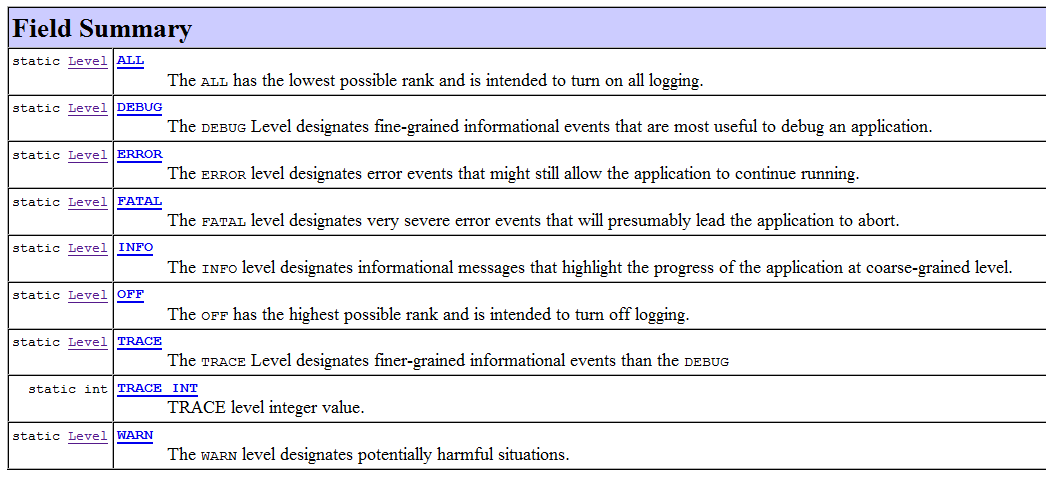
* Dans la section ***Web pages/WEB-INF/Classes*** du projet NetBeans.
* Dans ***\Sources\WebService\WebServiceOde\build\web\WEB-INF\classes*** du projet déployé

Par exemple, nous avons réalisé le débogage avec le niveau DEBUG, dans un fichier de sortie :



Dans l’API log4j, 4 niveaux de finesse sont disponibles, cf. sa Javadoc sur :

<https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html>



En situation de production, le niveau de finesse doit être augmenté à ERROR ou WARN pour ne pas surcharger le serveur (Même si la taille des logs applicatives est réduite à 5 Mo, les logs au niveau DEBUG vont nécessiter des ressources I/O disques importantes, au détriment des performances du Web-service)

Le fichier de logs applicatives est actuellement généré à l’emplacement définit par le paramètre suivant du ***log4j.properties*** :



|  |
| --- |
| *NOTE :* *Le processus de Build & Deploy décrit ci-dessous va écraser le fichier existant par celui du projet NetBeans.* |

### Logs techniques

Indépendamment des logs applicatives, les logs techniques du serveur Glassfish contiennent tous les évènements du serveur (Lancement / Arrêt du serveur, Déploiement / Retraits de .WAR) ainsi que les logs complètes en cas d’exception Java. Les logs applicatives y sont également retranscrits, mais noyées sous un flot d’informations techniques, ce qui rend leur utilisation délicate.

Dans notre configuration de déploiement (Serveur Glassfish local sur Windows 7 Pro) ces logs techniques sont stockées dans :

C:\Users\olivier.essner\AppData\Roaming\NetBeans\8.0.2\config\GF\_4.1\domain1\logs\

|  |
| --- |
| *NOTE : Les fichiers cachés de Windows doivent être affichés.* |

## Installation et configuration

### Environnement utilisé

* **OS** : Microsoft Windows 7 Pro 64 bits FR
* **IDE** : Oracle NetBeans IDE 8.0.2 for Java EE
* **JDK** : Oracle JDK 1.8.0 u 60
* **Serveur applicatif** : Oracle Glassfish 4.1 (Fourni avec NetBeans)

### Description du projet NetBeans

* **Catégorie de projet** : Java Web
* **Projet** : Web application
* **Nom du projet** : OdeWebService
* Pas de répertoire dédié aux librairies
* **Serveur** : Glassfish Server 4.1
* **Java EE version** : Java EE 7 Web
* Pas de framework

### Librairies tierces

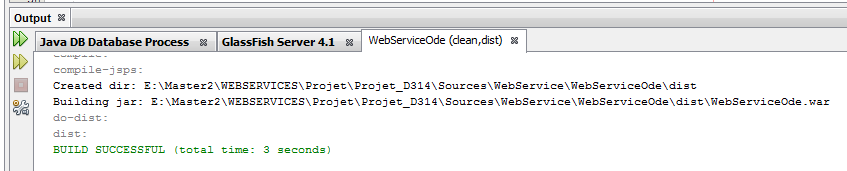
* **Log4j** < log4j-1.2.17> : Pour les logs applicatifs.
* **SQLite** < sqlite-jdbc-3.8.11.2> : Base de données embarquée.

## Tests et mise en œuvre

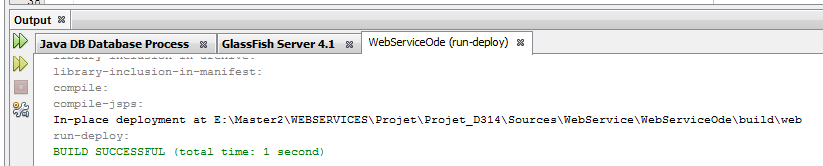
### Build et déploiement

Lancer NetBeans, puis ouvrir le projet ***\Sources\WebService\WebServiceOde***

Sur l’explorateur de solution de NetBeans, faire un clic-droit sur la racine ***WebServiceOde*** puis choisir « Clean & Build ». La fenêtre « Output » de NetBeans doit signaler un succès :



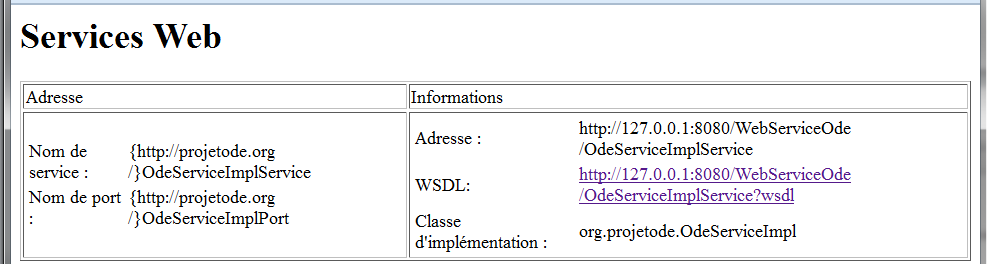
Sur l’explorateur de solution de NetBeans, faire un clic-droit sur la racine ***WebServiceOde*** puis choisir « Deploy ». La fenêtre « Output » de NetBeans doit signaler un succès, avec démarrage du serveur Glassfish si c’est le premier déploiement depuis le lancement de NetBeans :



On peut s’assurer du bon déploiement du Web-service en accédant à la page HTML auto-générée par Glassfish pour décrire les Web-service qui sont déployés et démarrés :

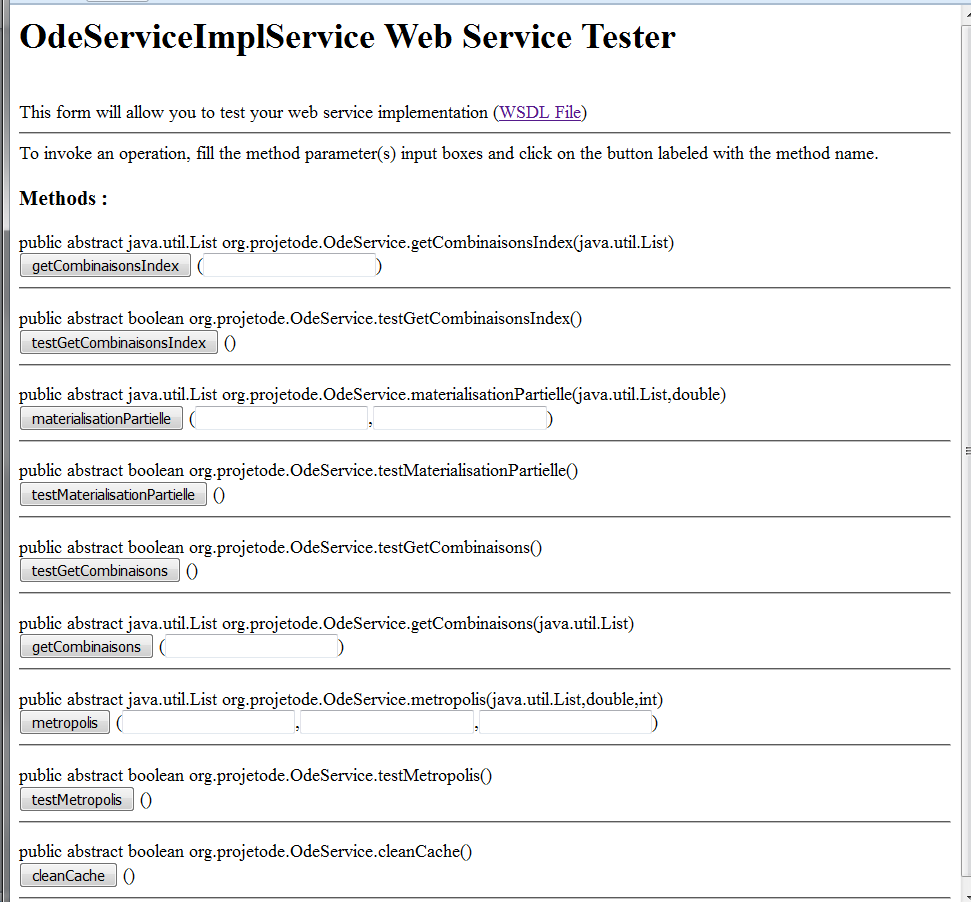
Résumé des web-services déployés sur le serveur applicatif :

<http://127.0.0.1:8080/WebServiceOde/OdeServiceImplService>



Liste détaillée des méthodes exposées sur :

<http://127.0.0.1:8080/WebServiceOde/OdeServiceImplService?tester>



En cas de problème (Page inaccessible ou service non-déployé) consulter les logs techniques, cf. Section « Logs techniques » plus haut.

### Méthodes de tests

Chaque méthode du Web-service (A l’exception du nettoyage de cache) dispose d’une fonction de test, qui simule la partie cliente : Les méthodes du Web-services sont appelées avec des valeurs de test prédéfinies.

Dans le source ***DimensionTest.java***, on déclare les 4 méthodes de test suivantes :

|  |  |
| --- | --- |
| Méthode exposée | Méthode de test associée |
| getCombinaisonsIndex(java.util.List) | testGetCombinaisonsIndex() |
| materialisationPartielle(java.util.List,double) | testMaterialisationPartielle() |
| getCombinaisons(java.util.List) | testGetCombinaisons() |
| metropolis(java.util.List,double,int) | testMetropolis() |

Ces méthodes sont utiles pour tester le fonctionnement du web-service en l’absence de client, à partir de la seule page HTML de test générée par Glassfish sur :

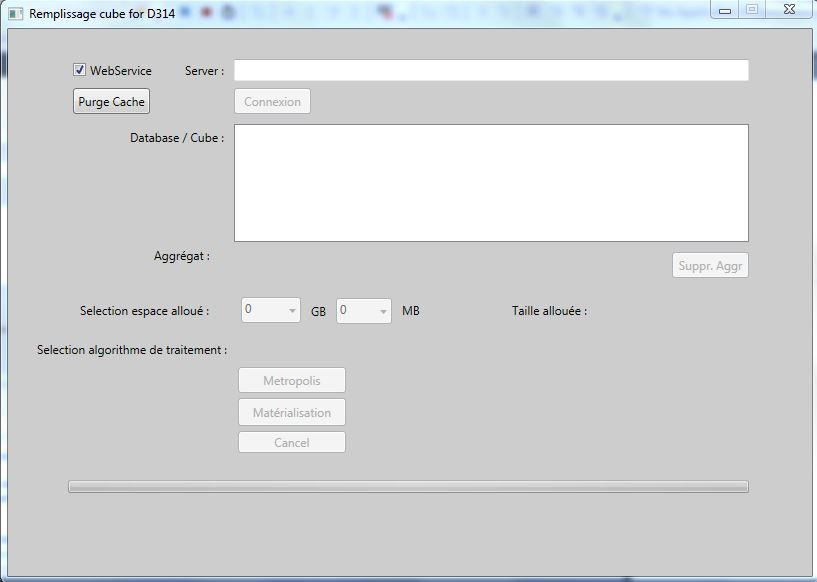
<http://127.0.0.1:8080/WebServiceOde/OdeServiceImplService?tester>

Ces méthodes de test n’ont pas d’argument, car tout est prédéfini, et ne retournent qu’un booléen : 1 si tout s’est bien passé, 0 sinon. Le débogage du Web-service se fait alors au travers de ses logs applicatifs, cf. Section « Logs applicatives » plus haut.

# Détails de la partie client

L’optimiseur ODE est lancé via le fichier « Optimiseur ODE » présent dans le répertoire « Source/Client/ ».

## Présentation de l’interface de l’optimiseur ODE

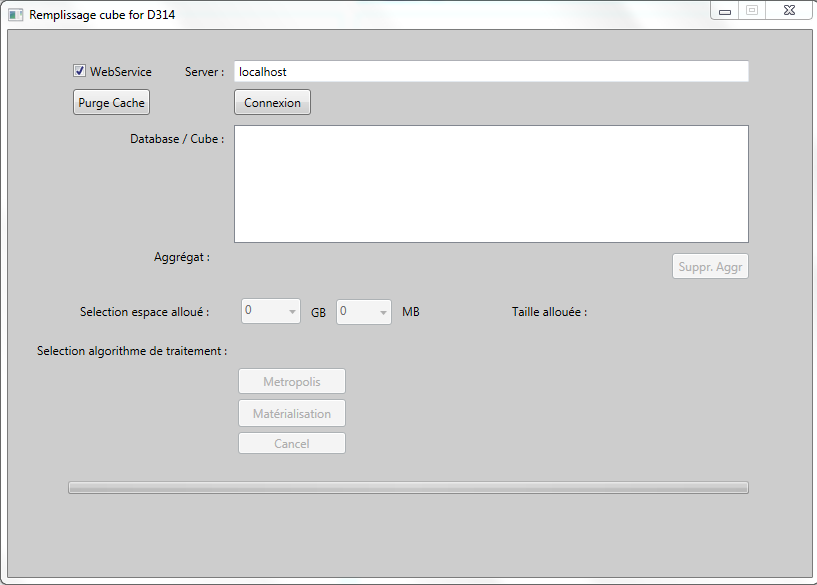


### Détails des objets

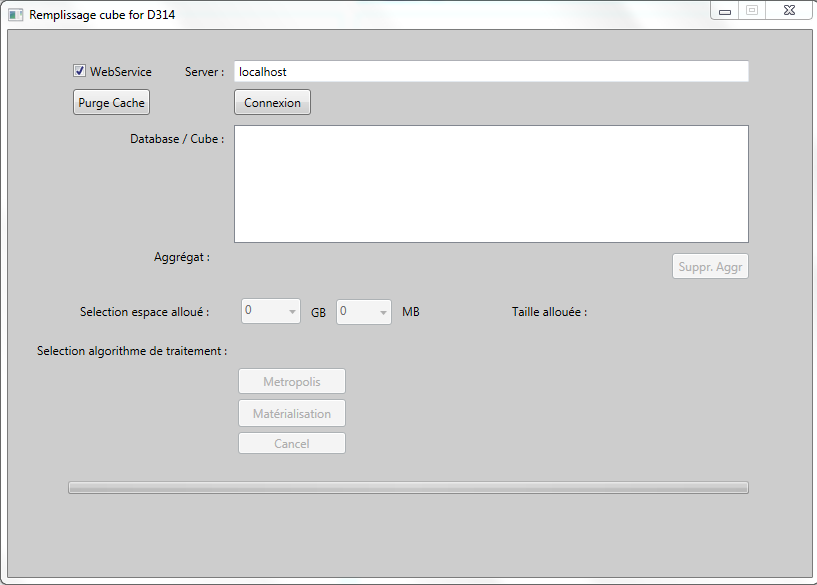
* **Check-box Webservice** : Cette fonction n’était pas présente dans la version initiale de projet. Il permet de basculer en mode Full-Webservice.
* **Textbox Server :** Permet de spécifier le server contenant les cubes de données à traiter. A noter que pour les besoins de ce projet, nous avons ajouté la gestion d’un mode déconnecté que nous détaillerons plus loin.
* **Bouton Purge Cache :** Cette fonction n’était pas présente dans la version initiale de projet. Il permet d’effectuer le vidage de la cache via un Webservice dédié.
* **Bouton Connexion :** Permet la connexion au server.
* **ListBox Database/Cube :** Permet de sélectionner le cube de données à traiter.
* **Bouton Suppr. Aggr :** Permet de supprimer les aggrégats présents sur le cube de données sélectionné.
* **Selection espace alloué :** Permet de spécifier l’espace disque alloué au traitement de Métropolis ou de Matérialisation.
* **Bouton Matérialisation :** Permet de lancer l’optimisation via l’algorithme de Matérialisation Partielle.
* **Bouton Cancel :** Permet de stopper le traitement.

## Exemple d’utilisation de l’optimiseur

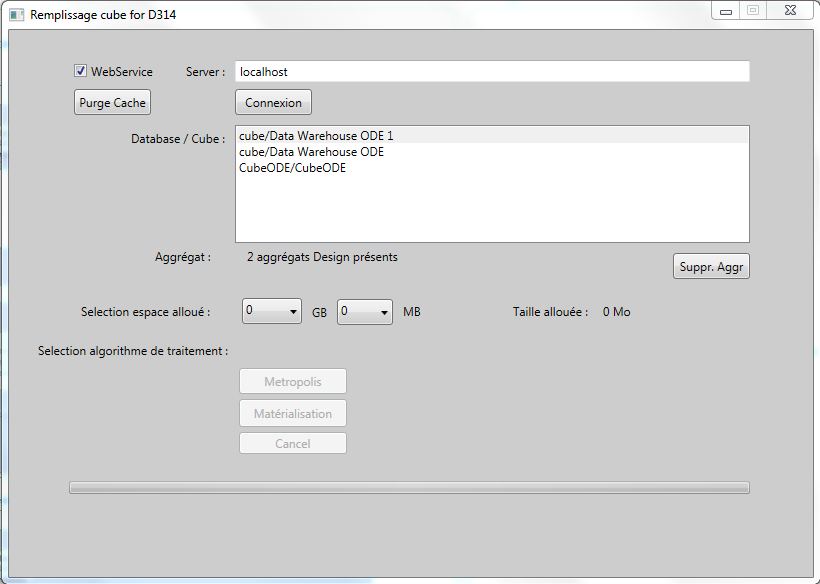
### Etape 1 : Sélection du server



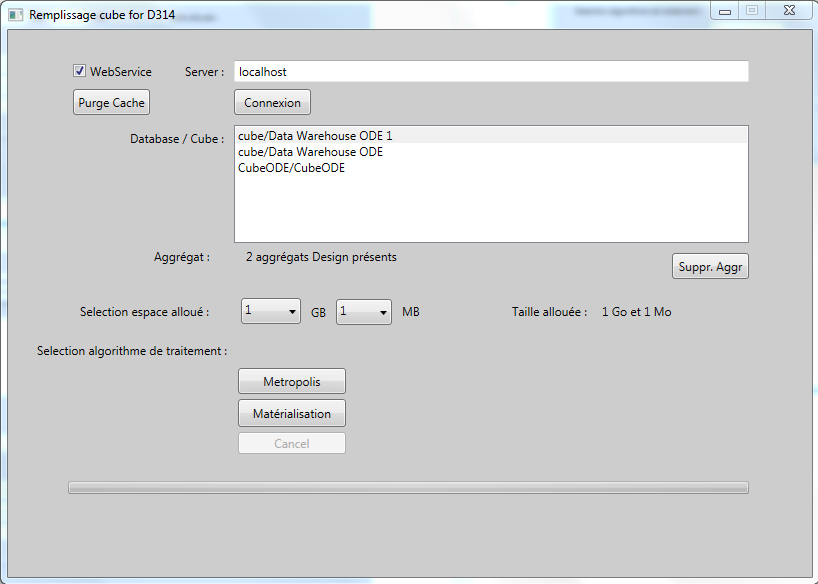
### Etape 2 : Appui sur le bouton Connexion



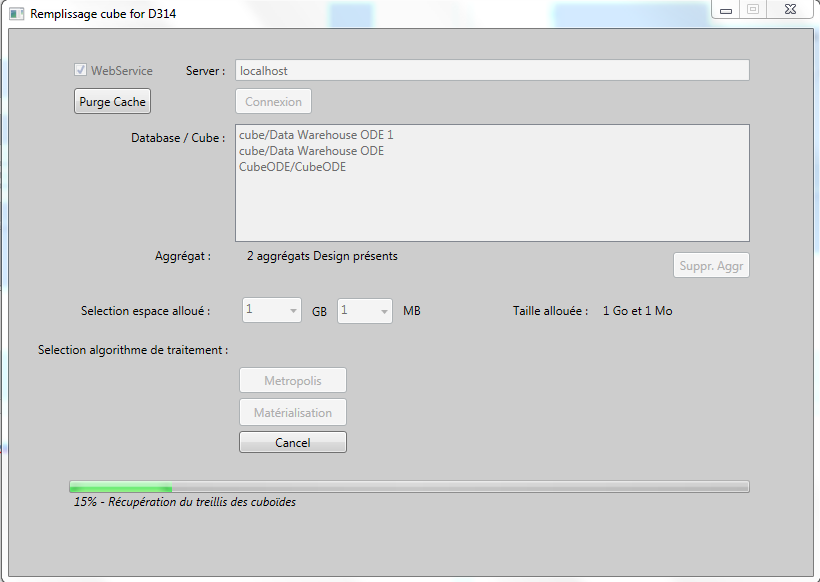
### Etape 3 : Sélection cube de données



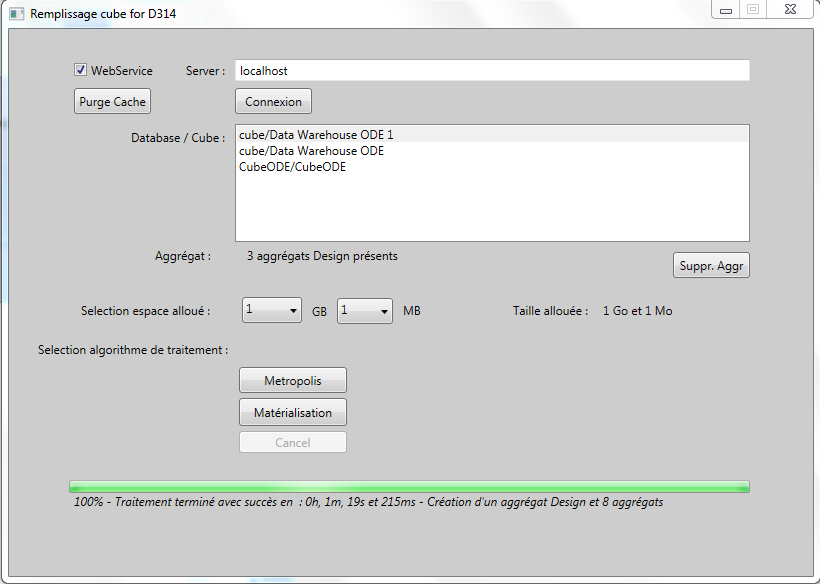
### Etape 4 : Sélection espace alloué



### Etape 5 : Sélection Algorithme Metropolis ou Matérialisation



### Etape 6 : Fin d’optimisation

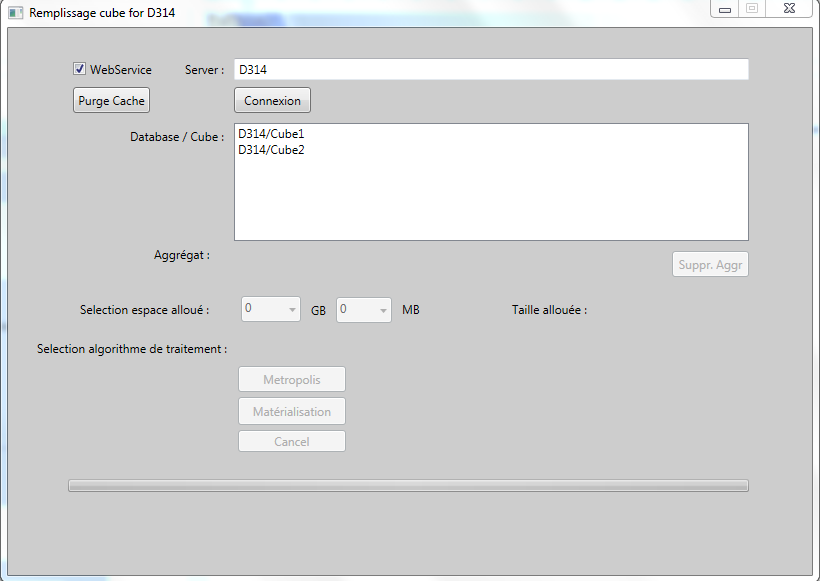


## Mode déconnecté

Ce mode a été ajouté afin de permettre une utilisation en mode déconnecté de l’optimiseur c’est-à-dire sans présence de l’ensemble de la base OLAP, il sera donc utilisé dans le cadre de ce projet.

L’activation de ce mode se fait uniquement en sélectionnant « D314 » en tant que Server.

L’affichage de l’interface est alors la suivante après connexion :



Dans ce mode, les données issues des cubes de données sont directement forcées dans les procedures :

* **Bouton\_Connexion\_Click :** forçage des noms de cubes et de leurs nombres d’aggrégats
* **WS\_GetDimension1DProperties :** forçage des dimensions et taille mémoire
* **WS\_GetPoidsCuboides :** forçage count et espace mémoire

## Mode Full-WebService

Ce mode a été ajouté afin de permettre une utilisation en mode Webservice, il sera donc utilisé dans le cadre de ce projet. Les procédures suivantes ont ainsi été créées :

|  |  |
| --- | --- |
| Mode Webservice | Projet initial |
| WS\_GetCombinaisons | GetCombinaisons |
| WS\_Metropolis | Metropolis |
| WS\_MaterialisationPartielle | MaterialisationPartielle |

|  |
| --- |
| *NOTE : Lors de nos tests sur les Webservices, nous nous sommes positionné en mode connecté. Ensuite nous avons vérifié que les aggrégats créés avec les algorithmes de Métropolis ou de Matérialisation étaient identiques en mode Full-Webservice et en version intiale.* |

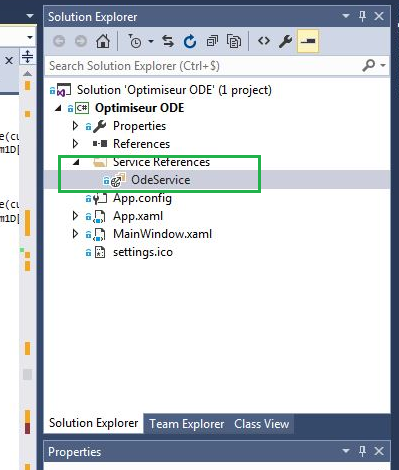
## Bouton Purge Cache

Ce bouton a été ajouté afin de permettre une purge de la cache. Il fera appel à la procédure « Bouton\_Purge\_Click ».

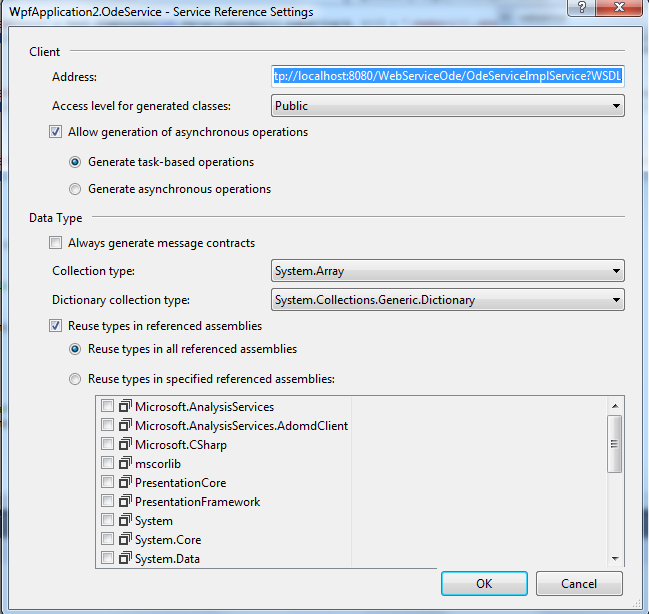
Après confirmation de la demande de suppression, cette procédure fera appel au Webservice avec la fonction « CleanCache ».

## Mise en œuvre Technique

### Ajout de la référence au Webservice



### Ajout de l’adresse du Webservice



### Implantation dans le code C#

L’ensemble des procedures ont été codées de la façon suivante :

* Instanciation de la classe proxy permettant l'appel distant au Webservice.

OdeService.OdeServiceClient Webservice = new OdeService.OdeServiceClient();

* Création d’un objet list de dimension au format Webservice (dimension et non Dimension).

List<OdeService.dimension> listDim1D\_Temp = new List<OdeService.dimension>();

* Création des objets list de dimension au format Webservice afin de gérer les objets retournés pas le Webservice.

List<OdeService.dimension> listCuboides\_Temp = new List<OdeService.dimension>();

List<String> index\_cuboides\_Temp = new List<String>();

* Peuplement de la liste des objets du Webservice par transfert de notre objet list de Dimension vers l’objet List de dimension.

for (int i = 0; i < listDim1D.Count; i++)

{

listDim1D\_Temp.Add(new OdeService.dimension() { dimensionName = … });

}

* Appel du Webservice

listCuboides\_Temp = Webservice.GetCombinaisons(listDim1D\_Temp.ToArray()).ToList();

index\_cuboides\_Temp = Webservice.GetCombinaisonsIndex(listDim1D\_Temp.ToArray()).ToList();

* Alimentation de la list Dimension à partir de la list dimension retournée par le Webservice.

for (int i = 0; i < listCuboides\_Temp.Count; i++)

{

listCuboides.Add(new Dimension(…));

listCuboides[i].SetDimensionMemory(listCuboides\_Temp[i].dimensionMemory);

index\_cuboides.Add(index\_cuboides\_Temp[i]);

}

|  |
| --- |
| *NOTE : Il est nécessaire d’effectuer un traitement particulier entre les listes C# et celles en java attendues en entrée des Webservices car nous avons une incompatibilité entre les classes Dimensions (C#) et dimensions (Java).* |