BENECH Jean
LEHUEN Nicolas

20/05/1997

Axone

Serveur HTTP
Connecté à un SGBDR
Avec interpréteur de script

INSA de Rouen
Département Génie Mathématique et Informatique
1996-1997

Table des matières

1 Présentation générale du projet

1.1 Besoins

Ce projet a pour but de permettre la génération automatique de pages HTML à partir d'un part de données brutes se trouvant dans une base de données, d'autre part de modèles de présentation (*templates*).

Les modèles de présentation permettent de décrire la forme générale d'un document. Par exemple, on peut décider que le titre d'un document apparaît toujours dans une police donnée, à coté d'une icône dépendant du type de document.

Les pages HTML sont ensuite générées à la demande en fonction de ce modèle et des données formant les documents (par exemple : titre, auteur, type du document, contenu du document, document suivant, document précédent, etc.). S'il est nécessaire de changer la mise en page des documents, il suffit de le faire au niveau du modèle de présentation.

Ainsi on obtient une forte indépendance entre le fond et la forme d'un document, ainsi qu'une facilité de mise en page et d'administration accrue.

Un tel système demande que le format des fichiers modèles de présentation soit simple, pratique mais surtout polyvalent, afin de couvrir une large gamme de besoins. On peut en effet imaginer qu'un serveur de document doive fournir un moteur de recherche, par exemple. Ce moteur devant afficher ses résultats dans un document HTML, il peut être intéressant d'avoir un système de modèles de documents suffisamment performant pour gérer des informations stockées dans une base de données aussi bien que des informations obtenues par interaction avec un client.

Afin de faciliter l'administration du serveur, il est intéressant de pouvoir gérer une majeure partie de sa configuration non pas par des fichiers de configuration complexes, mais par des formulaires renvoyés par le serveur au client via le Web. Ainsi, l'administration du serveur peut se faire de tout point du globe, simplement et rapidement.

Ceci entraîne des impératifs de sécurisation du serveur, afin d'éviter que tout client puisse en changer la configuration. Certains documents du serveur ne doivent être accessibles que par l'administrateur du serveur. Il faut donc gérer l'identification de la personne effectuant une requête sur ces documents sensibles, et décider d'autoriser l'accès ou non suivant l'identité obtenue.

Un tel système de contrôle d'accès peut d'ailleurs être étendu à tout document présent sur le serveur, afin de pouvoir limiter l'accès des documents du serveur aux seules personnes concernées (données sensibles ou privées dans le cas d'un Intranet, par exemple).

L'identification d'un client nécessite enfin une dernière caractéristique pour le serveur : la gestion du suivi de sessions de consultation. En effet, les échanges d'information sur le Web se font en mode non connecté, sous la forme requête - réponse. Tel quel, le protocole HTTP ne permet pas de savoir si deux requêtes consécutives proviennent de la même personne, par exemple.

Un système de suivi de sessions permet donc de connaître la provenance de chaque requête faite au serveur, ce qui permet entre autres d'éviter une procédure d'identification du client à chaque requête le nécessitant : l'identification ne se fait qu'à la première requête sensible, puis l'identité de la personne est stockée dans la session ouverte. Un tel système a un autre avantage : il permet la persistance de données partagées par plusieurs documents au cours d'une même session, en particulier des résultats de requêtes SQL.

1.2 Solution envisagée

Lors de l'étude des besoins engendrés par ce projet, il nous est apparu qu'afin d'accélérer les échanges d'information au sein du serveur il serait nécessaire d'obtenir un logiciel serveur homogène.

En effet, le recours à des solutions classiques telles que le développement de programmes CGI pour des serveurs HTTP déjà existants (NCSA, CERN, Apache, Netscape,…) poserait des problèmes de performance, accrus par l'interfaçage avec un SGBDR.

Des solutions plus efficaces existent pour pallier les déficiences des programmes CGI : Fast-CGI, ISAPI, NSAPI, etc. Cependant, elles ont pour inconvénient de dépendre fortement du matériel utilisé pour le serveur, voire du logiciel serveur utilisé.

Nous avons donc décidé de développer un logiciel serveur complet en Java intégrant toutes les fonctionnalités nécessaires. Les performances a priori plus faibles de l'interprétation de bytecode Java sont en effet compensées par les limitations de bande passante sur Internet. On atteint ainsi des taux de transfert d'information sur Internet similaires entre une application développée en Java et une application développée en C++, par exemple (le facteur limitant le taux de transfert étant la bande passante, et non la rapidité d'exécution de l'application).

De plus, le fait d'avoir une architecture homogène (toute en Java) permet d'éviter des pertes de performances dues à des échanges d'informations inter-processus, tels qu'il en existe pour l'interface CGI ou Fast-CGI.

2 Fonctionnement interne du projet

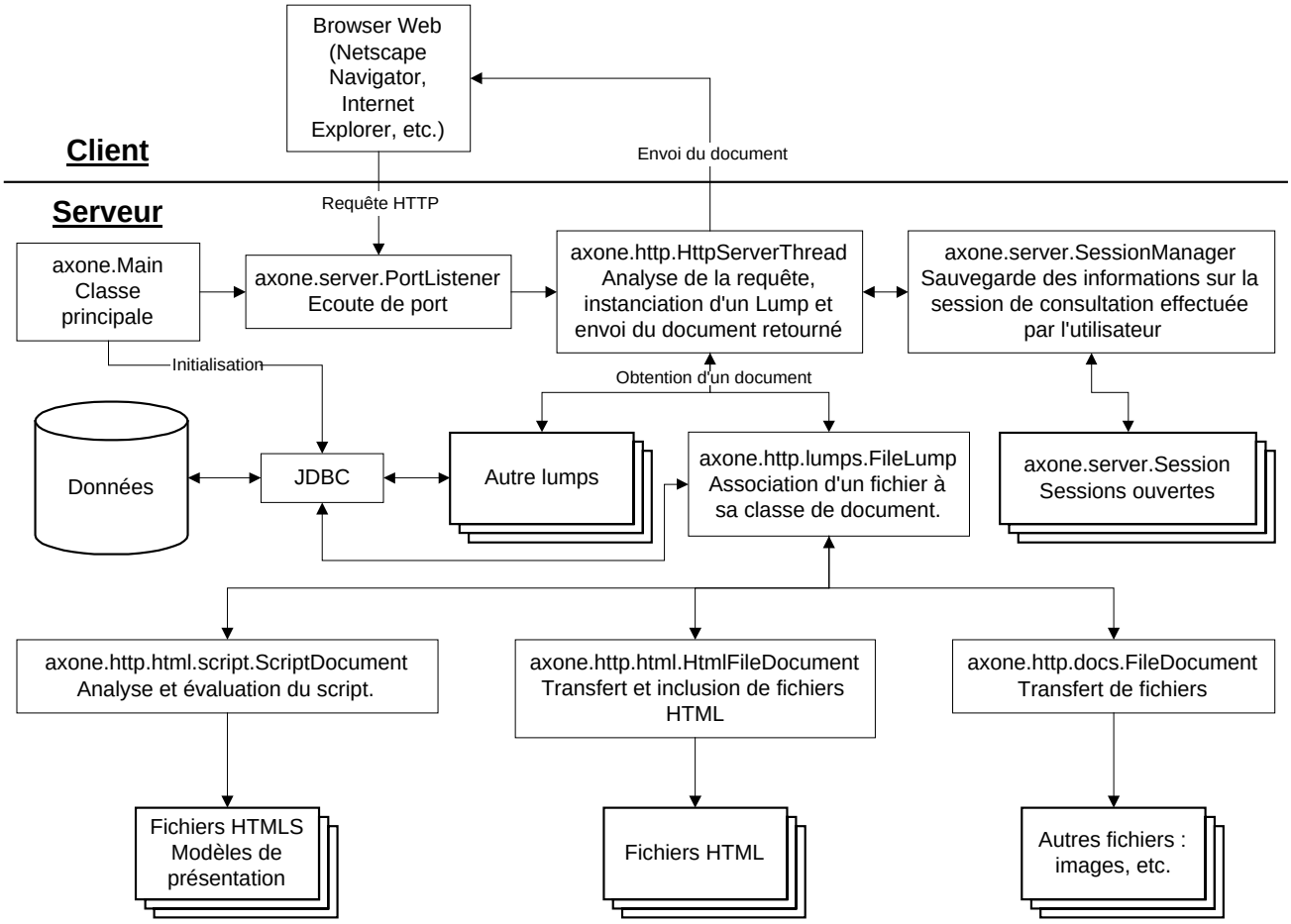

Figure 1 - Structure du projet

2.1 La partie serveur HTTP

Cette partie gère les aspects du protocole HTTP en entrée.

2.1.1 Gestion de la connexion

Au lancement du serveur (lancement de la classe `axone.Main`), on instancie la classe `axone.server.PortListener`, chargée d'écouter les demandes de connexion sur un numéro de port donné (ce numéro de port est habituellement 80 pour un serveur HTTP).

A chaque demande de connexion, le `PortListener` instancie une `axone.http.HttpServerThread`, une thread chargée de répondre à la requête. Cette thread est démarrée, et le `PortListener` est prêt à accepter une nouvelle connexion. Ainsi, toutes les requêtes envoyées au serveur sont traitées en parallèle.

2.1.2 Analyse de la requête

La `HttpServerThread` chargé de répondre à la requête construit d'abord une instance de la classe `HttpRequest`. Cette classe se construit en lisant les informations envoyées par le client selon le protocole HTTP.

2.1.3 Gestion de la session de la requête

On détermine ensuite la session à laquelle appartient la requête en regardant si elle contient un cookie HTTP appelé `SESSIONID`. Ce cookie est envoyé au client par `HttpServerThread` à chaque requête.

Si le cookie est absent, cela signifie qu'il s'agit d'une requête appartenant à une nouvelle session, que l'on ouvre avec le `axone.server.session.SessionManager` du serveur (il y a alors instanciation de `axone.server.session.Session`).

Si le cookie est présent, `HttpServerThread` demande au `SessionManager` l'instance de `Session` correspondante. Là encore, s'il n'y a pas de Session correspondante, on en ouvre une.

`Session` contient de nombreuses informations, principalement l'identité du client ; l'historique de la session (l'ensemble des requêtes effectuées par le client), des résultats de formulaires, des connexions à des bases de données et des résultats de requêtes.

Initialement, l'identité du client n'est pas fixée. Elle ne le sera que si besoin est, au cours de l'étape suivante.

2.1.4 Instanciation d'un lump, obtention d'un document et envoi du document.

Les différents champs de la requête (instance de la classe `axone.http.HttpRequest`) sont alors analysés, en particulier le champ `reference`, qui contient l'URL à retourner.

La `HttpServerThread` détermine ensuite le lump qui va traiter la requête. Pour cela, elle examine la référence de la requête. Si elle est de la forme `/~(Nom de Lump)/(suite de la référence)`, alors le nom du lump devant traiter la requête est obtenu facilement. Sinon, c'est le lump `axone.http.lumps.FileLump` qui est utilisé par défaut.

`HttpServerThread` instancie alors le lump requis, et vérifie auprès de lui si l'identité du client est requise. Le cas échéant, la requête est transférée au lump `axone.http.lumps.IdentifyLump` (chargé d'identifier le client).

Une fois le problème de l'identité du client réglé, `HttpServerThread` demande au lump s'il accepte de traiter la requête, sachant maintenant à qui il s'adresse. Si oui, on récupère le document généré par le lump. Sinon, la requête est transférée à `axone.http.lumps.AccessDeniedLump`, qui renvoie un document précisant que l'accès à la référence est refusé.

Tout lump renvoie un document, que le `HttpServerThread` n'a plus qu'à envoyer au client. Tout document dérive de la classe abstraite `axone.http.docs.Document`, qui contient une méthode `send` destinée à envoyer le document. Ainsi, le `HttpServerThread` n'a pas à se préoccuper de la manière d'envoyer le document, ni d'ailleurs de son type.

Une fois le document envoyé au client, `HttpServerThread` se termine.

2.2 Les lumps

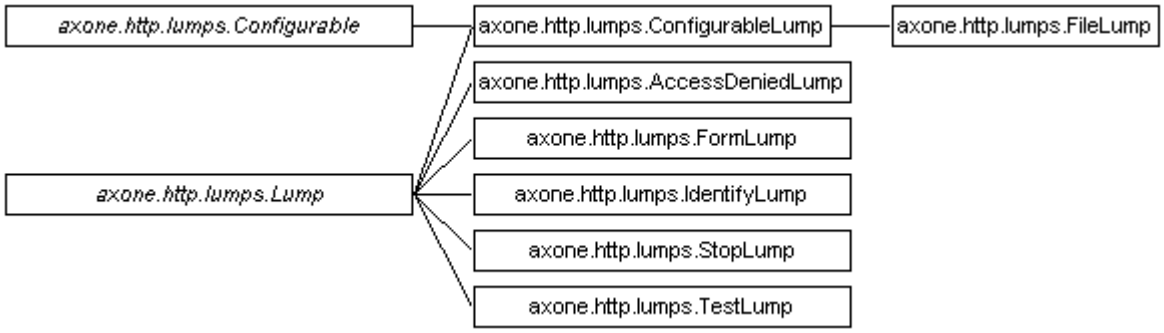

Figure 2 - Hiérarchie des lumps

2.2.1 Principe général

Les lumps, baptisés ainsi car ce sont les briques élémentaires du serveur (lump = morceau de sucre en anglais, dans l'esprit café !), ont pour but de retourner un document (instance d'une classe dérivée de `Document`) à partir d'une requête du client.

Au niveau programmation, un lump implémente l'interface `axone.http.lumps.Lump`, qui définit entre autres la méthode `process`. Cette méthode prend en argument le `HttpServerThread` qui a instancié le lump, la `Session` ouverte et la `HttpRequest` en cours. Elle doit retourner un `Document`, donc une instance d'une de ses classes dérivées puisque `Document` est abstraite.

2.2.2 Droits d'accès

Comme expliqué plus haut, les lumps peuvent demander l'identification du client, et accepter ou refuser l'accès à une référence. Ceci est implémenté par deux méthodes de `Lump`, `needIdentity` et `checkAccess`.

Ces deux méthodes prennent en argument la `HttpRequest` devant être traitée et la `Session` ouverte. La méthode `needIdentity` n'est appelée que si l'identité du client est inconnue, et doit retourner `true` si le lump désire la connaître. La méthode `checkAccess` est toujours appelée, et doit retourner `true` si le lump accepte de traiter la requête.

Il est ainsi possible de définir des droits d'accès au niveau de chaque lump. En particulier, le lump `FileLump` peut très bien définir des droits d'accès de manière précise sur chacun des fichiers du serveur (bien que cela ne soit pas encore implémenté).

2.2.3 Lumps configurables

Ces lumps dérivent de la classe abstraite `axone.html.lumps.ConfigurableLumps`, et correspondent à des lumps qui peuvent se configurer en leur donnant la référence `/config`. Ainsi, le lump configurable `FileLump` peut se configurer en donnant pour URL `http://(serveur)/~FileLump/config`.

Le but est que tout lump configurable fournisse un ou plusieurs documents contenant des formulaires permettant de le configurer via un simple browser Web. Ainsi, la configuration de `FileLump`, le principal lump du serveur, se fait de cette manière.

Les lumps configurables peuvent utiliser les services de la classe `axone.server.config.ConfigManager`, qui permet de stocker des éléments de configuration de manière persistante. En effet, le `ConfigManager` est chargé de lire les configurations des lumps depuis un fichier au démarrage du serveur, puis de sauver régulièrement ces configurations (notamment à l'arrêt du serveur).

Le `ConfigManager` peut stocker la configuration de tout objet implémentant l'interface `axone.http.lumps.Configurable`. Cette interface définit la méthode `getDefaultConfiguration`

chargée de renvoyer une configuration par défaut pour l'objet. Une configuration dérive d'`Object`, donc n'importe quel objet peut être stocké par le `ConfigManager`.

L'intérêt du `ConfigManager` est de libérer les lumps de la charge de gestion des fichiers de configuration. On n'a plus qu'un seul fichier de configuration géré par le `ConfigManager`. Pour pouvoir stocker tout type d'objet dans le fichier de configuration, on utilise le mécanisme de sérialisation d'objets introduit par le JDK 1.1.

2.2.4 Le lump par défaut : `FileLump`

Ce lump est le lump par défaut. Il est chargé de construire un document à partir d'un fichier présent dans le système de fichier. Quatre points sont configurables : les alias, le document par défaut, les classes de documents et les types de contenu.

Les alias permettent de définir des "raccourcis" vers des répertoires. Si on définit que l'alias `demo` pointe vers `/pub/www/demo`. Ainsi, l'appel à l'URL `http://(serveur)/demo/index.html` renverra le document du serveur `/pub/www/demo/index.html`.

Le document par défaut indique quel fichier charger lorsque le `FileLump` reçoit une URL sur un répertoire local. Par exemple si le document par défaut est index.html, l'appel à l'URL `http://(serveur)/demo` renverra le document du serveur `/pub/www/demo/index.html`.

Les classes de documents permettent d'associer une extension de fichier à une classe de document. La classe de document utilisée par défaut est `axone.http.docs.FileDocument`. Cette classe n'effectue aucun traitement sur le contenu du fichier.

Les documents contenant du script à interpréter, par exemple, doivent être associés à la classe `axone.http.html.script.ScriptDocument` si l'on désire envoyer une version interprétée du document au client (sinon le document serait renvoyé tel quel !). On associe donc les fichiers portant l'extension `htmls` à la classe `ScriptDocument`, et tous les fichiers portant cette extension seront interprétés avant l'envoi.

Plus simplement, on associe les documents HTML à la classe `axone.http.html.HtmlFileDocument`, afin de permettre l'insertion de fichiers HTML à l'intérieur d'autres documents HTML (voir 0 ci-dessous).

Les types de contenu permettent d'associer aux fichiers n'étant pas associés à une classe de document précise (i.e. associés à FileDocument) une information de type de contenu, nécessaire au protocole HTTP. Ainsi les fichiers GIF auront pour type de contenu `image/gif`.

2.2.5 Le lump d'identification : `IdentifyLump`

Ce lump permet d'identifier le client. Pour cela il affiche un formulaire d'identification, où on demande au client son *login* et son mot de passe.

La réponse du client est alors envoyée à `IdentifyLump` comparée aux entrées présentes dans la base de données du projet, via une requête SQL.

Si l'identité est correcte, `IdentifyLump` l'inclus dans la session en cours et `IdentifyLump` renvoie un document de confirmation contenant un lien vers le document initialement demandé.

Si l'identité est incorrecte, `IdentifyLump` renvoie un document le spécifiant.

2.2.6 Le lump de stockage des formulaires : `axone.http.lumps.FormLump`

Ce lump, utilisé pour traiter le résultat d'un formulaire, permet de stocker ce résultat dans une variable de la session en cours. Il est possible de spécifier une URL vers laquelle le `FormLump` réorientera la requête.

2.3 Les documents

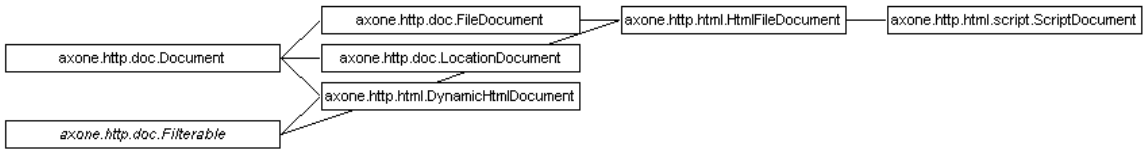

Figure 3 - Hiérarchie des classes de documents

2.3.1 Principe général

Un document dérive de la classe abstraite `Document`. Cette classe demande l'implémentation de trois méthodes : `getContentType`, qui doit retourner le `Content-Type` du document au format MIME, `getInputStream` qui doit renvoyer un flux d'entrée sur le contenu du document, et `getContentLength`, qui doit renvoyer la longueur du contenu ou -1 si cette longueur est inconnue.

A partir de ces trois méthodes, il est possible d'implémenter des classes de documents très différentes dans leur comportement, dont nous allons voir trois exemples ci-dessous, et un quatrième au 2.4.1 ci-dessous.

2.3.2 Les documents HTML dynamiques : `axone.http.html.DynamicHtmlDocument`

Cette classe de document permet aux lumps de construire des documents en mémoire, sans aucun support de fichier. C'est une classe très utile à la plupart des lumps, dans la mesure où beaucoup ont à créer des documents HTML à la volée.

2.3.3 Les documents issus de fichiers : `FileDocument`

La méthode `getInputStream` de cette classe renvoie un `InputStream` sur le fichier auquel l'instance de la classe est associée. Ainsi, la lecture et l'envoi du `FileDocument` revient à la lecture et l'envoi du fichier associé. Ceci permet d'avoir un accès rapide aux fichiers du serveur.

La méthode `getContentType` de cette classe renvoie le type de contenu associé à l'extension du fichier, tel qu'il a été configuré (cf. 2.2.4 ci-dessus).

La méthode `getContentLength` renvoie tout simplement la taille du fichier associé.

La classe `FileDocument` agit donc comme un encapsulateur de fichier ; son action est très réduite, par conséquent l'envoi de fichiers du serveur vers le client est tout particulièrement rapide, malgré l'impression de complexité que donne l'architecture.

2.3.4 Les documents HTML : `HtmlFileDocument`

Cette classe agit comme `FileDocument`, à l'exception que sa méthode `getContentType` retourne automatiquement `text/html`.

L'intérêt de cette classe réside dans le fait qu'elle implémente `axone.http.doc.Filterable`. Cette interface permet de fournir une version du document expurgée de son entête et de son épilogue, afin de permettre son insertion à l'intérieur d'un document contenant du script (cf. 2.4). Pour ce faire, on enlève toute la partie précédant le tag `<BODY>` et la partie suivant `</BODY>`.

2.4 L'interpréteur de script

2.4.1 Intégration au projet, la classe `axone.http.html.script.ScriptDocument`

Les documents contenant du script doivent être associés à la classe de documents `ScriptDocument`, afin que le script soit interprété avant l'envoi du document au client.

Le rôle de `ScriptDocument` est de lancer l'analyse du script lors de la construction du document, à partir du contenu du fichier. La méthode `getInputStream` de `ScriptDocument` renvoie ensuite le résultat de l'évaluation de l'expression résultant de cette analyse.

2.4.2 La grammaire reconnue

L'intérêt de notre système de script est de s'intégrer de manière totalement transparente à l'intérieur d'un fichier HTML. En effet, les instructions et expressions supportés par l'interpréteur de script sont toutes sous forme de tags HTML. Ainsi, à l'interprétation du script, chaque expression ou instruction renvoie une valeur sous forme de texte ou de code HTML, qui s'insère exactement à l'endroit ou elle se trouvait. Un exemple de fichier script et de son interprétation est donné au 3.2.4.

L'interpréteur de script reconnaît actuellement les instructions ou expressions suivantes :

Instruction / Expression	**Action / Valeur**
Gestion de base de données	
<CONNECT NAME=*Expression* URL=*Expression* USERNAME=*Expression* PASSWORD=*Expression*>	Connexion à une base de donnée.
<REQUEST NAME=*Expression* VALUE=*Expression*>	Création d'une requête.
<FIELD *Expression Expression*>	Récupération de la valeur d'un champ d'une requête.
<CLOSEREQUEST *Expression*>	Fermeture d'une requête.
<NEXT *Expression*>	Passage à l'enregistrement suivant d'une requête.
<PREVIOUS *Expression*>	Passage à l'enregistrement précédent d'une requête.
<FOR *Expression*>*Expression*</FOR>	Boucle sur les enregistrements d'une requête.
<REQUESTTABLE REQUEST=*Expression*>	Affichage de la table d'une requête.
<EOR *Expression*>	Renvoie TRUE si la requête a été entièrement lue.
<NOTEOR *Expression*>	L'inverse de EOR.
<ISEMPTY *Expression*>	Renvoie TRUE si le résultat d'une requête est vide.
<RECNUM *Expression*>	Renvoie le numéro d'enregistrement en cours dans une requête.
Gestion de fichiers, de formulaires	
<INSERT *Expression*>	Insertion d'un document.
<VAR *Expression*>	Valeur d'un champ de formulaire sauvegardé.
<ARG *Expression*>	Valeur d'un argument de requête HTTP.
<DATE>	Date et heure.
<THIS>	Référence du document actuellement interprété.
<DIRECTORY *Expression*>	Renvoie le répertoire contenant le fichier désigné par l'expression.
Expressions conditionnelles	
<WHILE *Expression*>*Expression*<WEND>	Evalue l'expression donnée tant que l'expression conditionnelle s'évalue en TRUE.
<IF *Expression*> *Expression* <ELSE> *Expression* <ENDIF>	Renvoie la première expression si l'expression conditionnelle s'évalue en TRUE, la seconde sinon.
<NOT *Expression*>	Renvoie TRUE si l'expression s'évalue en FALSE, FALSE sinon.
<EQUALS *Expression Expression*>	Renvoie TRUE si les deux expressions s'évaluent de la même manière.
<NOTEQUALS *Expression Expression*>	Renvoie FALSE si les deux expressions s'évaluent de la même manière.
<EQUALSIGNORECASE *Expression Expression*>	Renvoie TRUE si les deux expressions s'évaluent de la même manière, sans tenir compte de la casse.
<NOTEQUALSIGNORECASE *Expression Expression*>	Renvoie FALSE si les deux expressions s'évaluent de la même manière, sans tenir compte de la casse.
<AND *Expression Expression*>	ET logique entre les deux expressions.
<OR *Expression Expression*>	OU logique entre les deux expressions.

2.4.3 La phase d'analyse syntaxique

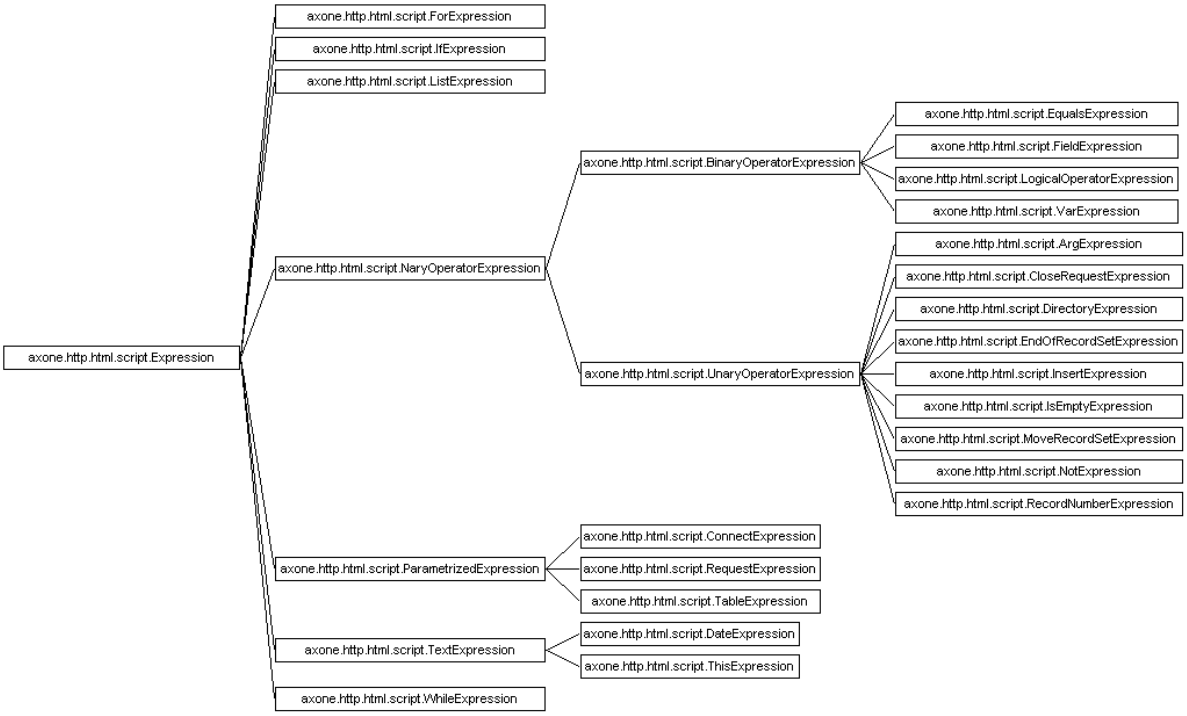

Figure 4 - Hiérarchie des classes d'expressions

L'analyseur syntaxique que nous avons développé est basé sur un algorithme prédictif de descente récursive à gauche. Cet algorithme relativement simple a l'avantage de s'adapter à notre grammaire. L'analyseur est implémenté dans la classe `axone.http.html.script.Parser`.

On commence par construire une expression de la classe `ListExpression` (liste d'expression). On parcourt ensuite le document à la recherche d'un mot clé de notre grammaire.

On stocke alors le texte lu jusqu'alors dans une expression de classe `TextExpression` (le texte lu étant du code HTML sans mots clés de notre grammaire) et on appelle le constructeur d'expression correspondant au mot clé lu. Ce constructeur appelle alors récursivement le `Parser` afin d'analyser les expressions contenues dans l'expression construite.

2.4.4 La phase d'évaluation

Une fois l'arbre des expressions du script construit, on peut l'évaluer à la demande, de manière toute aussi récursive. Afin d'évaluer une expression, il faut en effet évaluer les expressions qu'elle contient.

L'évaluation d'une expression peut soit se faire par écriture directe sur un flux de sortie (`java.io.OutputStream`), par obtention d'un lecteur (`java.io.Reader`) ou d'une chaîne de caractères, suivant les besoins.

3 Guide de l'utilisateur

3.1 Configuration

3.1.1 Configuration initiale : le fichier `axone.properties`

Ce fichier permet de définir quelques points de configuration nécessaires au lancement su serveur. C'est un fichier au format `properties` du Java. Ce fichier doit se trouver à la racine des fichiers du projet. Ce fichier est facultatif, s'il est absent, des valeurs par défaut sont prises qui équivalent au fichier suivant :

```
# Nom de la base de données requise par le serveur. Voir 3.1.2.
database.name=jdbc:odbc:axone
# Nom et mot de passe de l'administrateur de la base de données système
database.admin.username=Administrateur
database.admin.password=
# Nom du fichier contenant les configurations gérées par le ConfigManager
config.filename=axone.config
```

3.1.2 Configuration de la base de données système

Dans le cas où le pont JDBC-ODBC est utilisé pour l'accès à la base de données système, il est nécessaire de configurer l'ODBC de manière à définir un alias reliant cette base de données au nom précisé dans le fichier `axone.properties`. Sous Windows 95 ou Windows NT, cela s'effectue en appelant le panneau de configuration, en sélectionnant "ODBC" (ou "ODBC 32 bits"), puis en créant une entrée DSN.

La base de données système ne doit contenir pour l'instant qu'une seule table : la table `Users`, contenant les champs `USERNAME` (nom de login de l'utilisateur), `PASSWORD` (mot de passe), `UID` (numéro unique d'utilisateur), `REALNAME` (nom "en clair" de l'utilisateur) et `EMAIL`.

Cette base de donnée devra bien sûr être enrichie selon les besoins. Il est fort probable par exemple que l'on aie besoin d'un système de groupes d'utilisateurs, afin de pouvoir définir plus aisément des droits d'accès sur les fichiers du serveur.

N'oubliez pas de créer un utilisateur dans la table `Users` ou bien il vous sera impossible de configurer certains points du serveur, ou même de l'arrêter correctement !

3.1.3 Lancement

Placez-vous à la racine des fichiers du projet, c'est à dire dans le répertoire contenant le sous répertoire `axone` (celui contenant le fichier `Main.class`). Lancez ensuite la classe `axone.Main` :

```
D:\Axone>java axone.Main
axone.Main : Starting a PortListener for axone.http.HttpServerThread on port 80
```

Le serveur est alors fonctionnel sur le port 80. Si vous désirez lancer le serveur sur un autre port :

```
D:\Axone>java axone.Main axone.http.HttpServerThread:8080
axone.Main : Starting a PortListener for axone.http.HttpServerThread on port 8080
```

3.1.4 Configuration du `FileLump`

Une fois le serveur lancé pour la première fois, il vous faut configurer le `FileLump`. La majeure partie de la configuration est déjà faite par défaut, mais quelques points restent à définir, comme les alias.

Ouvrez un browser Web, et ouvrez l'URL `http://(votre serveur)/~FileLump/config`.

Le serveur vous demande alors de vous identifier, puis un menu apparaît vous permettant de configurer au choix les alias, les types de contenus ou les classes de documents. Suivez le guide !

3.1.5 Arrêt du serveur

L'arrêt du serveur se fait interactivement, en appelant l'URL `http://(votre serveur)/~StopLump`. Une vérification de votre identité est faite si nécessaire, puis un document de confirmation vous est retourné. Confirmez votre demande et le serveur s'arrête.

3.2 Ecriture de scripts

3.2.1 Principe général

Toute expression de script insérée dans un document est exécutée et/ou remplacée par sa valeur au moment de l'exécution, du coté serveur. La plupart des expressions renvoient des données destinées à l'affichage, comme l'expression DATE qui renvoi la date et l'heure.

Ainsi, le document de script suivant affiche la date et l'heure du moment :

```
<HTML>
	<HEAD><TITLE>Date et heure</TITLE></HEAD>
	<BODY>
		<H1><DATE></H1>
	</BODY>
</HTML>
```

Certaines autres instructions ne sont destinées qu'à effectuer une action, et sont en général remplacées par des commentaires HTML (entre `<!--` et `-->`) permettant de savoir si l'instruction s'est correctement exécutée ou non.

Attention, afin d'être interprété, un document contenant du script doit être associé à la classe de documents `ScriptDocument`. Pour cela, dans la configuration de `FileLump`, vérifiez que l'extension du document correspond bien à la classe `axone.http.html.script.ScriptDocument`. Evitez cependant d'associer par exemple les fichiers portant l'extension `html` à cette classe. En effet, ces documents sont censés ne pas contenir de script, et leur envoi au client serait plus lent.

Il est donc préférable de ne faire figurer du script que dans des fichiers portant l'extension `htmls`, par exemple.

Pour une référence rapide des différentes instructions ou expressions, voyez le tableau du 2.4.2.

3.2.2 Utiliser des requêtes SQL

3.2.2.1 Connexion à une base de données

Le principe est de créer une connexion à une base de données, puis de la stocker dans une variable de connexion. On utilise pour cela l'instruction `CONNECT` :

`<CONNECT NAME=`*`(Nom de la variable de connexion)`* `URL=`*`(URL de la connexion au format JDBC)`* `USERNAME=`*`(Nom de login pour la connexion)`* `PASSWORD=`*`(Mot de passe pour la connexion)`* *…*`>`

Tous les arguments à fournir peuvent l'être sous la forme d'expressions.

Le nom de variable de la connexion est totalement libre, pourvu qu'il ne contienne pas d'espaces. L'URL de la connexion est de la forme `jdbc:odbc:demo` si vous désirez vous connecter à l'alias ODBC appelé `demo`. Il est possible de fournir des arguments en sus de `USERNAME` et de `PASSWORD`, si la base de données le nécessite, en utilisant la forme PARAMETRE=VALEUR.

La variable de connexion est stockée dans la session du client. Ainsi, il est possible de conserver une connexion d'un document à l'autre, mais pas d'un utilisateur à l'autre.

Si une demande d'ouverture est faite pour une variable de connexion déjà ouverte, la connexion reste dans son état précédent. Ainsi on économise le temps de connexion à la base de données. De manière générale, essayez de reconstruire la connexion à chaque document en ayant besoin. Ainsi, vous êtes certain que la connexion est toujours ouverte, sans perte de temps si elle l'est déjà.

L'instruction connexion renvoie un commentaire HTML indiquant le bon déroulement de la connexion.

3.2.2.2 Création et fermeture d'une requête SQL

Une fois la connexion créée, on construit une requête SQL et on la stocke dans une variable de requête à l'aide de l'instruction `REQUEST` :

`<REQUEST NAME=`*`(Nom de la variable de requête)`* `VALUE=(`*`Libellé de la requête SQL)`* `[BIDIR=`*`(Expression retournant TRUE ou FALSE)`*`]>`

Là encore, le nom de la variable comme la requête SQL peuvent être des expressions à évaluer (il en va de même pour toutes les instructions ou expressions du langage de script).

Le fait que la requête SQL puisse être le résultat d'une expression permet d'ailleurs une grande richesse fonctionnelle, comme on le verra dans l'exemple.

L'argument BIDIR permet de préciser si l'on désire un résultat bidirectionnel, c'est à dire un résultat que l'on pourra parcourir dans les deux sens. Un résultat unidirectionnel ne peut être parcouru qu'en avant. Un résultat bidirectionnel peut être parcouru en avant comme en arrière, mais il prend plus de mémoire.

Une requête SQL peut ensuite être fermée à l'aide de l'instruction `CLOSEREQUEST` :

```
<CLOSEREQUEST (Nom de la variable de requête)>
```

La fermeture d'une requête permet d'économiser de la mémoire sur le serveur.

3.2.2.3 Consultation des résultats d'une requête

Une méthode simple de consulter le résultat d'une requête est d'en afficher la table à l'aide de l'expression `REQUESTTABLE` :

```
<REQUESTTABLE REQUEST=(Nom de variable de requête)>
```

Une méthode plus souple consiste à combiner une expression `FOR` et des expression `FIELD` :

```
<FOR (Nom de variable de requête)>
       …
       <FIELD (Nom de variable de requête) (Nom de champ 1)>
       …
       <FIELD (Nom de variable de requête) (Nom de champ 2)>
       …
       …
</FOR>
```

L'expression `FOR` exécute une itération sur tous les enregistrements du résultat d'une requête. L'expression `FIELD` permet de récupérer la valeur d'un champ de l'enregistrement en cours de ce résultat. En combinant ces deux expressions il est donc possible de mettre dans n'importe quelle forme le résultat d'une requête.

Si l'expression `FOR` n'est pas assez souple (on veut ajouter une condition d'arrêt supplémentaire, par exemple), on peut utiliser une expression `WHILE` combinée à une expression `NOTEOR`, par exemple :

```
<WHILE <AND <NOTEOR DOC> <NOTEQUALS <FIELD DOC TITLE> "Essai">>>
       <FIELD DOC REF>
       <FIELD DOC TITLE>
       <A HREF="mailto:<FIELD DOC EMAIL>">Mail à l'auteur</A><BR>
       <NEXT DOC>
<WEND>
```

Ceci affiche une liste de document tant qu'il reste des enregistrements dans la requête et que le titre du document n'est pas `"Essai"`.

Les expressions NEXT et PREVIOUS permettent de passer d'un enregistrement à l'autre dans le résultat d'une requête SQL. L'expression PREVIOUS n'a d'effet que si la requête à laquelle on l'applique est bidirectionnelle.

3.2.3 Utiliser des résultats de formulaires ou des résultats de requêtes.

Il est possible de récupérer les arguments de la requête HTTP ayant requis un document de script avec des expressions `ARG` :

```
<ARG (Nom de l'argument dont on veut récupérer la valeur)>
```

Il est ainsi possible de créer des documents de script répondant à un formulaire :

(Dans un premier fichier)

```
<FORM ACTION=Bonjour.htmls METHOD=POST>
<INPUT TYPE=TEXT NAME=NOM>
<INPUT TYPE=SUBMIT>
</FORM>
```

(Dans `Bonjour.htmls`) `<H1>Bonjour <ARG NOM> !</H1>`

Si ce n'est pas suffisant, il est possible de sauvegarder les résultats d'un formulaire dans une variable de formulaire à l'aide du lump `FormLump` :

```
(Dans un premier fichier)
<FORM ACTION=/~FormLump METHOD=POST>
<INPUT TYPE=HIDDEN NAME=FORMNAME VALUE=MYFORM>
<INPUT TYPE=HIDDEN NAME=ANSWERWITH VALUE=Bonjour2.htmls>
<INPUT TYPE=TEXT NAME=NOM>
<INPUT TYPE=TEXT NAME=PRENOM>
<INPUT TYPE=SUBMIT>
</FORM>
```

Notez que l'on passe deux paramètres cachés à `FormLump` : le premier, `FORMNAME`, permet de fixer le nom de la variable de formulaire dans laquelle seront stockés les résultats. Le second, `ANSWERWITH`, permet de préciser à quel référence le `FormLump` passe la main une fois les données stockées.

```
(Dans Bonjour2.htmls)
<H1>Bonjour <VAR MYFORM PRENOM> <VAR MYFORM NOM> !</H1>
```

Notez que dans ce cas précis, `<VAR MYFORM PRENOM>` et `<ARG PRENOM>` sont strictement équivalents car `FormLump` réémet entièrement la requête HTTP vers la référence indiquée une fois le résultat du formulaire stocké.

L'intérêt de `FormLump` réside donc dans la capacité de stocker les résultats de plusieurs formulaires. Ainsi on peut imaginer un sondage qui se ferait sur plusieurs pages, plutôt qu'en une seule grande page, sans que cela pose de problème au niveau de la rédaction de script de traitement.

3.2.4 Un exemple : une base de données de documents, et son moteur de recherche, le tout en script.

On stocke des documents dans une table `Docs`, contenant les champs suivants : `REF` (référence du document), `Title` (titre du document), `Author` (numéro d'auteur du document), `Logo` (URL d'une image servant de logo au document), et `Text` (texte du document).

Les informations concernant les auteurs sont stockées dans la table `Authors`, avec les champs `AUTHREF` (numéro d'auteur), `FirstName`, `LastName` et `Email`.

Enfin, on a chaîné les documents, afin de préciser pour chaque document quel est le document précédent et le document suivant. Cette chaîne est stockée dans la table `Chain`, avec les champs `REF` (référence du document), `PREF` (réf. du document précédent) et `NREF` (réf. du document suivant). Les champs `NREF` et `PREF` sont mis à -1 s'il n'y a pas de suivant ou de précédent.

Il est donc possible dans un premier temps d'écrire un script définissant le modèle de présentation d'un document. Ce modèle est appelé soit directement, soit par un formulaire, et prend comme argument une variable `DOC` contenant la référence du document à afficher.

3.2.4.1 Modèle de présentation de document

Fichier DocView.htmls

On commence par ouvrir et stocker une connexion à la base de données.

```
<CONNECT NAME=DEMO URL=jdbc:odbc:demo USER=Administrateur PASSWORD="">
```

Puis on ouvre les différentes requêtes.

```
<REQUEST NAME=THISDOC VALUE="SELECT * FROM Docs WHERE REF=<ARG DOC>;">
<REQUEST NAME=THISAUTHOR VALUE="SELECT * FROM Authors WHERE AUTHREF=<FIELD
THISDOC Author>;">
```

```
<REQUEST NAME=THISCHAIN VALUE="SELECT * FROM Chain WHERE REF=<ARG DOC>;">

On affiche maintenant le document.
<HTML><HEAD><TITLE><FIELD THISDOC Title></TITLE></HEAD>
<BODY>
<H1><FIELD THISDOC Title></H1>
<IMG SRC="<FIELD THISDOC Logo>">
<H6>Auteur : <A HREF="mailto:<FIELD THISAUTHOR Email>">
<FIELD THISAUTHOR FirstName> <FIELD THISAUTHOR LastName></A></H6>
<HR>
<FIELD THISDOC Text>

On affiche une barre de navigation.
<HR>
<IF <NOTEQUALS <FIELD THISCHAIN PREF> -1>>
       On construit des liens vers ce document en passant la valeur appropriée pour DOC.
       <A HREF=<THIS>?DOC=<FIELD THISCHAIN PREF>>[Précédent]</A>
<ELSE><ENDIF>
<IF <NOTEQUALS <FIELD THISCHAIN NREF> -1>>
       <A HREF=<THIS>?DOC=<FIELD THISCHAIN NREF>>[Suivant]</A>
<ELSE><ENDIF><BR>
<A HREF=DocSelect.htmls>Retour au menu</A>
</BODY></HTML>

On ferme les requêtes maintenant inutiles.
<CLOSEREQUEST THISDOC>
<CLOSEREQUEST THISAUTHOR>
<CLOSEREQUEST THISCHAIN>
```

3.2.4.2 Système de sélection et de recherche de document

On peut fournir un menu permettant de sélectionner quel document on désire visualiser ou même de lancer une recherche par mots clés.

```
Fichier DocSelect.htmls
On ouvre la connexion si nécessaire.
<CONNECT NAME=DEMO URL=jdbc:odbc:demo USER=Administrateur PASSWORD="">

<HTML><HEAD><TITLE>Sélection du document</TITLE></HEAD>
<BODY>
<H1>Sélection d'un document par son titre</H1>

On crée un formulaire avec une liste déroulante.
<FORM ACTION=DocView.htmls METHOD=GET>

Il est tout à fait possible de ne créer la requête qu'à ce point du document !
<REQUEST NAME=DOCS VALUE="SELECT * FROM Docs;">
<P><SELECT NAME=DOC>

On fait une boucle FOR pour mettre une option pour chaque document.
<FOR DOCS>
       <OPTION VALUE=<FIELD DOCS REF>><FIELD DOCS Title>
</FOR>

On ferme la requête
<CLOSEREQUEST DOCS>

</SELECT>
<INPUT TYPE=SUBMIT VALUE="Visualiser"></P>
</FORM>

On crée un formulaire destiné à lancer une recherche par mots clés.
<H1>Sélection d'un document par mot clé</H1>
<FORM ACTION=DocSearch.htmls METHOD=GET>
```

```
<P>Mot clé : <INPUT TYPE=TEXT NAME=KEYWORD>
<INPUT TYPE=SUBMIT Value="Chercher"></P></FORM>
</BODY>
</HTML>
```

3.2.4.3 Document effectuant la recherche et affichant les résultats

```
<HTML><HEAD><TITLE>Résultat de la recherche</TITLE></HEAD>
<BODY>
<H1>Résultat de la recherche</H1>
<CONNECT NAME=DEMO URL=jdbc:odbc:demo USER=Administrateur PASSWORD="">
```

On effectue une recherche assez simple du mot clé dans le titre et dans le texte du document.

```
<REQUEST NAME=SEARCH VALUE="SELECT REF, Title FROM Docs WHERE ((Text Like
'%<ARG KEYWORD>%') OR (Title Like '%<ARG KEYWORD>%'));">
```

Si le résultat de la requête est vide…

```
<IF <ISEMPTY SEARCH>>
       <B>Pas de document trouvé !</B>
<ELSE>
```

… sinon on affiche le résultat, avec des liens permettant de visualiser les fichiers.

```
       <FOR SEARCH>
       <A  HREF="/demo/DocView.htmls?DOC=<FIELD  SEARCH  REF>"><FIELD  SEARCH
       Title></A><BR>
       </FOR>
<ENDIF>
<CLOSEREQUEST SEARCH>
</BODY></HTML>
```