HP Connect-It

Version du logiciel : 3.90

SDK

Date de publication de la documentation : 15 May 2008
Date de publication du logiciel : May 2008

Avis juridiques

Table des matières

Introduction

Le Kit de Développement Connect-It permet de développer et d'implémenter vos propres connecteurs. Ce kit de développement spécifie une interface Java basée sur la norme J2EE Connector Architecture (1.5). Le standard JCA définit un ensemble d'interfaces Java permettant de simplifier l'intégration des applications d'entreprise telles qu'un ERP, une base de données.

A qui s'adresse ce manuel

Ce manuel s'adresse à des développeurs ayant des connnaissances sur Java et la norme JCA. Pour obtenir plus d'information sur cette norme, consultez le site suivant : J2EE Connector Architecture [http://java.sun.com/j2ee/connector].

Terminologie

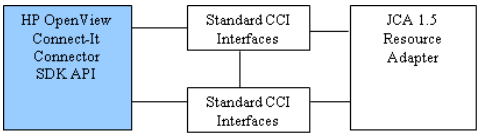
Les acronymes suivants sont utilisés dans ce manuel :

- JCA : J2EE Connector Architecture
- RA : Resource Adapter
- EIS : Enterprise Information System
- CCI : Common Client Interface
- SPI : Service Provider Interface

- JDBC : Java Database Connectivity

Généralités

L'API définit une extension à l'API JCA 1.5 de façon à supporter l'intégration du connecteur au sein de l'applicatif. Le schéma suivant illustre ce comportement :



Le mode de communication avec le connecteur dépend du système d'information (EIS) sur lequel il est connecté.

On distingue deux types possibles de communication :

- Communication outbound (synchrone)

  C'est le client qui initie l'échange de données. Dans ce cas, il s'agit par exemple de l'exécution d'une requête sur une base de données.
- Communication inbound (asynchrone)

  C'est l'EIS qui initie la communication. Le connecteur est alors en écoute d'événements. Un exemple est une messagerie.

Extensions SPI

Le SDK fournit une extension des classes SPI afin de supporter la description des métadonnées.

Extensions CCI

Le SDK fournit une couche cliente de façon à gérer l'accès à un système qu'il soit une base de données relationnelle ou non. Cette extension regroupe des fonctionnalités provenants de l'API CCI standard et de l'API JDBC.

1 Echange de données

Le but d'un connecteur construit à partir du SDK est de standardiser les échanges avec le système d'information. Ces échanges se font à travers l'envoi et la réception de données.

Données et types de donnée

Avant toute échange de données, il est nécessaire de connaître la structure de ces données. C'est ce que l'on appelle les métadonnées. Le SDK repose sur le fait que la structure d'une donnée doit être connue pour pouvoir être manipulée. Ceci se fait au travers des interfaces :

com.hp.ov.cit.connector.cci.ObjectRecord - représente une donnée particulière.

et

com.hp.ov.cit.connector.cci.ObjectType - représente la structure qu'un ensemble de données de même nature doit avoir.

Comme il doit être possible de décrire chaque donnée émise ou reçue, une instance *ObjectRecord* est liée à sa description *ObjectType*.

Les données fournies au travers du connecteur sont généralement organisées au sein d'une hiérarchie ou graphe, leurs métadonnées sont donc également hiérarchiques. Une métadonnée est dite 'complexe' lorsqu'elle contient d'autres métadonnées. Ces données filles forment les champs de ladite donnée. A contrario, une métadonnée est dite 'simple' lorsqu'elle ne contient aucune autre métadonnée. Elle ne contient donc aucun champ.

Un graphe d' *ObjectRecord* consiste en :

- Une seule donnée *ObjectRecord* racine.
- L'ensemble des *ObjectRecord*s qui peuvent être accédés en traversant les champs récursivement.

Modèles

Le SDK propose 2 modèles de données distincts décrits par le couple *ObjectType*, *ObjectRecord* qui sont le modèle *Classe/Instance* et le modèle *XMLSchema/XML*. Un seul modèle est possible au sein d'un connecteur.

Modèle Classe/Instance

Ce modèle est une représentation objet d'une structure de données. Ce modèle repose sur les notions de classe et d'instance Java.

Classe

Une classe possède un nom et appartient à un package qui forme son espace de nommage. Elle est formée de champs qui eux-mêmes sont associés à des classes.

Au sein de ce modèle une classe forme les métadonnées. Elle se représente donc au travers de l'interface *ObjectType*, et se caractérise par les méthodes :

```
public String getName();
public String getNamespace();

public Class getObjectClass();
public boolean isSimple();
public Field getField(String fieldName);
public Field[] getFields();
```

Un champ représenté par l'interface *com.hp.ov.cit.connector.cci.Field* regroupe alors ses informations propres ainsi que celles relatives à sa classe. Un champ possède les caractéristiques suivantes :

- il peut être modifiable,
- il peut posséder une valeur par défaut,
- il peut apparaître plusieurs fois et donc contenu dans une liste, on dit alors qu'il est indexé,
- il peut exiger qu'il possède une valeur.

Ceci se traduit alors au travers de l'interface *Field* par les méthodes:

```
public String getName();
public ObjectType getType();
public Object getDefault();
```

```
public boolean isIndexed();
public boolean isReadOnly();
public boolean isRequired();
```

Instance

Une instance est associée quant à elle à une classe et contient des valeurs pour un ou plusieurs de ses champs.

Au sein de ce modèle, une instance forme une donnée. Elle se représente donc au travers de l'interface *ObjectRecord*, et se caractérise par les méthodes :

```
public Object get(String fieldName);
public Object get(String fieldName, int fieldIndex);
public void set(String fieldName, Object value);
public void set(String fieldName, int fieldIndex, Object value);
public void remove(String fieldName);
public void remove(String fieldName, int fieldIndex);
```

Types simples

Le tableau suivant résume les types simples Java supportés par le SDK.

java.lang.Boolean
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double
java.lang.String
java.util.Date
byte[]
char[]

Exemple

Considérons le modèle de données ci-dessous:

```
A
|- String
|- int
|- B
|- String
|- C*
|- boolean
```

La représentation des classes métier est alors :

```
public class A
{
private String stringField = "This is a string";
private int intField;
```

```
private B bField;
}

public class B
{
private String stringField;
private List<C> listOfCField;
}

public class C
{
private boolean booleanField;
}
```

Les opérations sur les types peuvent se faire comme suit :

```
ObjectType objectTypeA = ...;
Field field = objectTypeA.getField("stringField");
boolean isSimple = field.getType().isSimple(); // true
Object defaultValue = field.getDefault(); // "This is a string"
...
field = objectTypeA.getField("bField");
isSimple = field.isSimple(); // false
ObjectType objectTypeB = field.Type();
field = objectTypeB.getField("listOfCField");
boolean isIndexed = field.isIndexed(); //true;
```

Les opérations de population de données peuvent se faire commme suit :

```
ObjectRecord objectA = ...;
ObjectRecord objectB = ...;
ObjectRecord objectC = ...;

objectA.set("intField", 5);
objectA.set("bField", objectB);

java.util.List<C> list = new java.util.ArrayList<C>();
list.add(objectC);
objectB.set("listOfCField", list);
```

Modèle XMLSchema/XML

Dans le cas de systèmes manipulant des données XML, il peut être intéressant d'utiliser directement ce mode de représentation. Une métadonnée est alors formée d'un ensemble de schémas XML indépendants. Ce modèle restreint l'utilisation des interfaces décrites plus haut. Ainsi, les seules méthodes pertinentes de l'interface *ObjectType* sont :

```
public String getName();
public String getNamespace();

public boolean isXSD();
public org.w3c.dom.ls.LSInput[] getXSD();
```

Ce modèle suppose également qu'une métadonnée identifiée par son nom et son espace de nommage est toujours simple. C'est-à-dire qu'il ne peut contenir de champ, mais seulement un ou plusieurs schémas XML.

De la même façon les données proprement dites sont accessibles au sein de l'interface *ObjectRecord* par les méthodes :

```
public void readXML(org.w3c.dom.ls.LSInput input);
public void writeXML(org.w3c.dom.ls.LSOutput output);
```

Ces dernières permettent d'importer ou d'exporter la représentation XML :

- org.w3c.dom.ls.LSInput - représente une source d'entrée pour la donnée XML.
- org.w3c.dom.ls.LSOutput - représente une destination de sortie pour la donnée XML.

2 Design-Time

Cette section décrit les éléments utilisés par un connecteur pour se connecter à un EIS et découvrir ses métadonnées.

Interface DesignTimeFactory

L'interface *com.hp.ov.cit.connector.spi.designtime.DesignTimeFactory* centralise les informations nécessaires à :

- l'obtention d'une connexion
- la description structurelle des données échangées avec l'EIS

Mode de communication

Les méthodes

public boolean supportsOutbound()

et

public boolean supportsInbound()

permettent de reconnaître le mode communication utilisé par l'EIS. Au sein de Connect-It ces 2 modes sont exclusifs. Ce qui veut dire que l'implémentation d'un connecteur ne peut supporter ces 2 modes à la fois.

Connexion *design-time*

Quel que soit le mode de communication, il est nécessaire de décrire les types de données échangées. Pour cela on utilise la notion de connexion, qu'elle soit réelle ou non. Il est également possible dans le cas d'une communication outbound que cette connexion soit différente de la connexion utilisée par la suite lors de l'échange proprement dite. Par exemple, dans le cas d'un web service, les métadonnées sont décrites par un fichier WSDL qui peut être accessible par une connexion ftp alors que l'interaction avec le web service se fait sur un protocole http.

L'API de la classe *DesignTimeFactory* propose les méthodes suivantes:

- **public boolean requiresSeparateMetaDataConnection()**

  Détermine si l'EIS distingue les 2 types de connexions. Il est à noter que dans le cas d'une communication inbound, cette méthode n'est pas utilisée.

- **public javax.resource.cci.ConnectionSpec createMetaDataConnectionSpec()**

  Cette méthode retourne une implémentation JavaBean de l'interface *ConnectionSpec*. L'objet contient des informations telles que "utilisateur", "mot de passe", ... spécifiques à un client, pour se connecter lors de la phase de design-time.. Dans le cas d'une communication outbound ne différenciant pas la connexion design-time de celle du run-time, la méthode concernée est alors

- **public javax.resource.cci.ConnectionSpec createConnectionSpec()**

Par exemple, si l'accès aux métadonnées se fait par une connexion http, il est peut-être nécessaire de connaître l'url pour s'y connecter :

```
package com.myeis;

import java.net.URL;
import javax.resource.cci.ConnectionSpec;

public class MyEISConnectionSpec implements ConnectionSpec
{
private URL url;

public URL getUrl()
{
return url;
}

public void setUrl(String url)
{
this.url = url;
}
}
```

Une fois, les informations de connexion obtenues, il est alors possible de décrire les métadonnées proprement dites.

Obtention des métadonnées

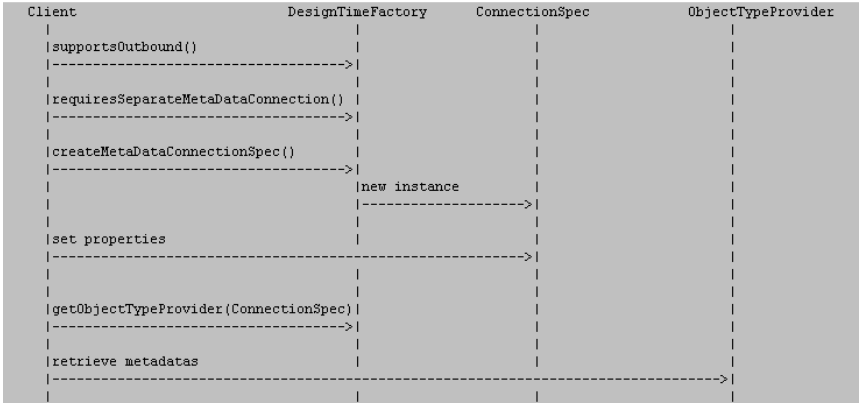
La méthode

public ObjectTypeProvider getObjectTypeProvider(javax.resource.cci.ConnectionSpec metaDataConnSpec)

retourne un objet permettant d'obtenir la description des données qui seront échangées avec l'EIS. Elle prend en paramètre les informations de connexion nécessaires.

Exemple

Les opérations décrites ci-dessus peuvent se résumer par le diagramme suivant dans le cas d'une communication outbound nécessitant une connexion spécifique pour l'accès aux métadonnées :



Un client demande à la **DesignTimeFactory** si le connecteur supporte la communication outbound. Dans l'affirmative, il lui demande alors de préciser si la connexion aux métadonnées est distincte de celle utilisée pour échanger des données. Suivant la réponse, le client fait appel alors soit à la méthode *createMetaDataConnectionSpec*, soit à la méthode *createConnectionSpec* afin d'obtenir la description d'une connexion. Le client remplit alors les propriétés de cette dernière et fait appel à la *DesignTimeFactory* pour obtenir l'objet *ObjectTypeProvider* permettant de décrire les métadonnées.

Interface ObjectTypeProvider

L'interface *com.hp.ov.cit.connector.spi.designtime.ObjectTypeProvider* permet de décrire les types de données de l'EIS. Il est possible que cette description soit infinie. Par exemple, un type de donnée A peut contenir un type de donnée B qui lui même contient un type de donnée A. Pour éviter ce problème de récursivité, au lieu de décrire d'un bloc les types de données, l'interface se base sur un modèle navigable. Ainsi seuls les types de données de premier niveau sont retrouvés, puis par appels successif à l'interface, les niveaux inférieurs peuvent être décrit. Comme ces types sont obtenus à travers une connexion, l'appel à la méthode

public void close()

permet de libérer celle-ci.

Les méthodes appelées pour décrire les métadonnées de premier niveau sont listées ci-dessous :

```
public java.util.List<ObjectType> getReceivedTypes();
public java.util.List<ObjectType> getRequestTypes();
public java.util.List<ObjectType> getResponseTypes();
```

Suivant le type d'EIS et le mode de communication considéré (inbound ou outbound) ces méthodes devront alors être ou non supporté. Dans le cas d'une méthode supportée, l'implémentation de celles-ci se fait de la manière suivante :

```
public java.util.List<ObjectType> getXXXTypes()
{
java.util.List<ObjectType> types = new java.util.ArrayList<ObjectType>();
types.add(new MyEISObjectType());
...
return types;
}
```

Dans le cas d'une méthode non supportée :

```
public java.util.List<ObjectType> getXXXTypes() throws javax.resource.NotS
upportedException
{
throw new javax.resource.NotSupportedException();
}
```

Communication Inbound

Dans ce mode, seule la méthode suivante est supportée :

```
public java.util.List<ObjectType> getReceivedTypes()
```

Cette méthode doit retourner la liste des événements susceptibles d'être reçus depuis l'EIS.

Communication Outbound

Deux styles d'échange de données peuvent être supportés :

- le style requête /réponse (au sens requête HTTP)
- le style interrogation (au sens requête SELECT SQL)

Les types de données dans le cas de requêtes s'obtiennent par :

```
public java.util.List<ObjectType> getRequestTypes()
```

Cette méthode doit retourner la liste des types de requêtes susceptibles d'être envoyées vers l'EIS.

Lorsqu'une requête produit une réponse, par exemple l'appel à une fonction **getPurchaseOrder(int id)** retournant un objet "PurchaseOrder", voici la méthode qu'il faut implémenter pour décrire le type de réponse attendu :

```
public java.util.List<ObjectType> getResponseTypes()
```

Lorsqu'une requête induit sa réponse, par exemple l'appel à une fonction "getPurchaseOrders(PurchaseOrderType)" retournant un objet "PurchaseOrder", on considère que c'est alors la méthode

```
public java.util.List<ObjectType> getReceivedTypes()
```

qui est éligible. En effet, au lieu d'effectuer une requête contenant des données, il s'agit plutôt d'interroger l'EIS pour retrouver des éléments d'après leur métadonnées.

Il est à noter que la méthode **getResponseTypes()** ne peut être supportée seule. Il n'est évidemment pas possible de recevoir une réponse de l'EIS sans au préalable lui envoyer une requête

Navigation

Une fois les types de premier niveau obtenus, l'appel à la méthode suivante permet de retrouver niveau par niveau les sous-types :

```
public ObjectType getType(String namespace, String name)
```

En utilisant les informations du couple *namespace, name* passé en paramètre par l'appelant, il est ainsi possible de connaître le niveau que l'on doit décrire. Lorsqu'un niveau est terminal, la méthode doit retourner *null*.

3 Runtime

Cette section décrit les éléments utilisés par un connecteur pour se connecter à un EIS et échanger des données.

Communication Outbound

Configuration

La classe clé de ce mode de communication est celle implémentant l'interface *javax.resource.spi.ManagedConnectionFactory*. Il est obligatoire que cette classe implémente également l'interface *javax.resource.spi.ResourceAdapter*. D'après les spécifications JCA, il est également obligatoire que cette classe soit un JavaBean. Les champs de cet objet JavaBean représentent les informations nécessaires à la connexion indépendamment du client. Par exemple, pour à une base de donnée via une connexion ODBC, le nom de cette base est nécessaire quel que soit le client.

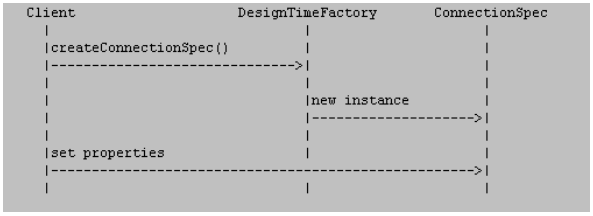
```
package com.mycompany.myeis;

import javax.resource.spi.ManagedConnectionFactory;
import javax.resource.spi.javax.resource.spi.ResourceAdapter

public class MyEISManagedConnectionFactory implements ManagedConnectionFac
tory, ResourceAdapter
{
private String dataSourceName;
```

```
public String getDataSourceName()
{
return dataSourceName;
}

public void setDataSourceName(String dataSourceName)
{
this.dataSourceName = dataSourceName;
}

...
}
```

Connexion

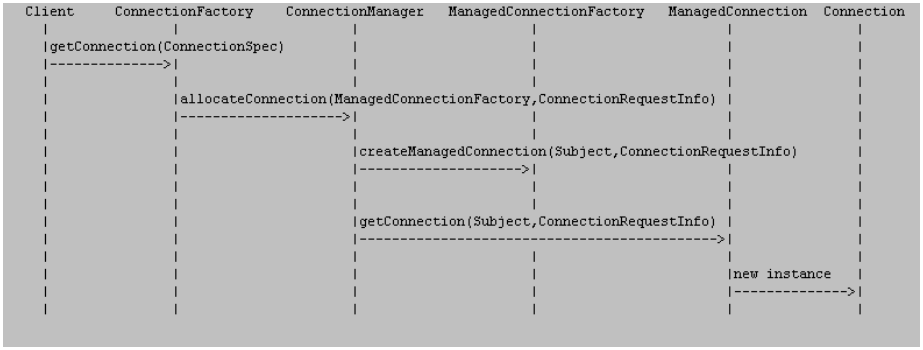
L'obtention d'une connexion cliente au sein d'un connecteur construit sur le SDK respecte la norme JCA.

Il faut au préalable obtenir un objet *javax.resource.cci.ConnectionSpec* représentant les informations de connexion. Cette obtention est résumée par le schéma suivant :



Le client accède alors à l'EIS via l'interface *javax.resource.cci.ConnectionFactory* pour créer un connexion à partir des informations fournies. Pour des raisons

de simplicité certains détails ont été omis (pooling de connexion, enregistrement de listener sur la connexion).



Toute implémentation doit obligatoirement retourner un objet connexion du type *com.hp.ov.cit.connector.cci.Connection*.

Echange

Une fois la connexion établie, l'application cliente (Connect-It) est capable d'échanger des données avec le système externe. A ce niveau, deux styles d'échange sont possibles en accord avec les informations de design-time :

- requête avec ou sans réponse
- interrogation

style Requête/ Réponse

La plupart des échanges avec un EIS se regroupent sous cette catégorie. Par exemple, insertion d'un record dans une base de données relationnelle. L'accès à cette fonctionnalité se fait par la méthode :

public Interaction createInteraction()

L'interface *com.hp.ov.cit.connector.cci.Interaction* est la suivante:

```
public interface Interaction
{
...
public ObjectRecord execute(ObjectRecord request) throws ResourceException
;
}
}
```

On fournit alors une donnée en entrée et l'on obtient (ou non) une réponse en retour.

Style Interrogation

Il s'agit d'interroger l'EIS en lui fournissant un prototype des données attendues. Par analogie, une requête SQL SELECT spécifie en entrée quelles colonnes sont attendues dans les enregistrements reçus.

L'accès à cette fonctionnalité se fait par la méthode:

public Statement createStatement()

L'interface *com.hp.ov.cit.connector.cci.Statement* est la suivante:

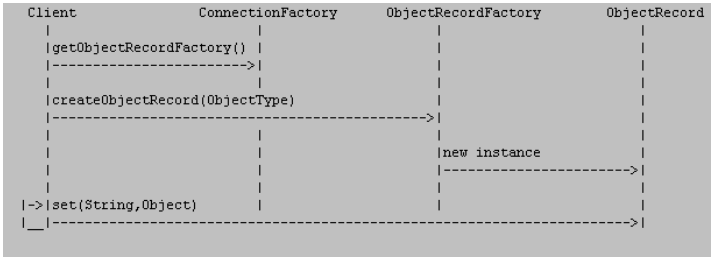
```
public interface Statement
{
...
public ObjectResultSet executeQuery(ObjectRecord prototype) throws Resourc
eException;
}
}
```

On itère alors sur le result set retourné pour retrouver les différentes données résultats en utilisant les méthodes **next()** et **getObjectRecord()**.

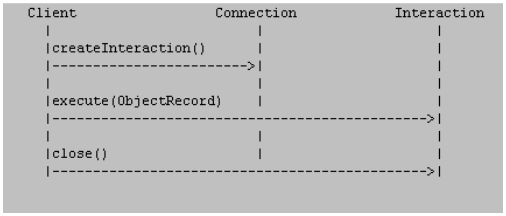
```
public interface ObjectResultSet
{
public boolean next();
public ObjectRecord getObjectRecord();
public void close() throws ResourceException;
}
```

Diagrammes

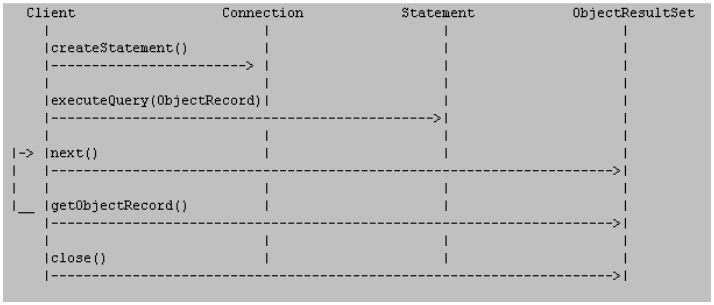
Création d'une donnée à partir des métadonnées du design-time :



Création d'une interaction avec la donnée obtenue précédemment :



Interrogation à partir d'un prototype de donnée obtenue précédemment :



Communication Inbound

Configuration

La classe clé de ce mode de communication est celle implémentant l'interface *javax.resource.spi.ResourceAdapter*. D'après les spécifications JCA, il est également obligatoire que cette classe soit un JavaBean. Les champs de cet objet JavaBean représentent les informations nécessaires à la connexion indépendamment du client.

Connexion

C'est la classe implémentant l'interface *javax.resource.spi.ActivationSpec* qui représente les informations nécessaires à l'établissement d'une connexion cliente. Comme la classe *javax.resource.spi.ResourceAdapter*, ce doit être obligatoirement un objet JavaBean.

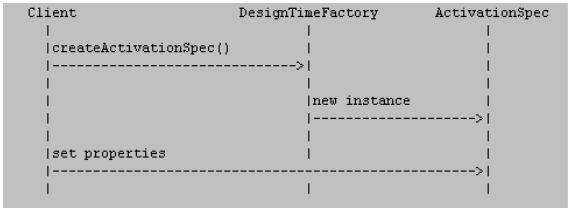
Echange

L'échange dans ce cas est à l'initiative de l'EIS. Le connecteur fonctionne alors comme un écouteur d'évènements. Sur réception de ces évènement le connecteur notifie le client grâce à l'objet *javax.resource.spi.endpoint.MessageEndPointFactory* passé en paramètre lors de son démarrage. Cette fabrique lui permet de créer un objet *ConnectionListener* dont l'interface est la suivante :

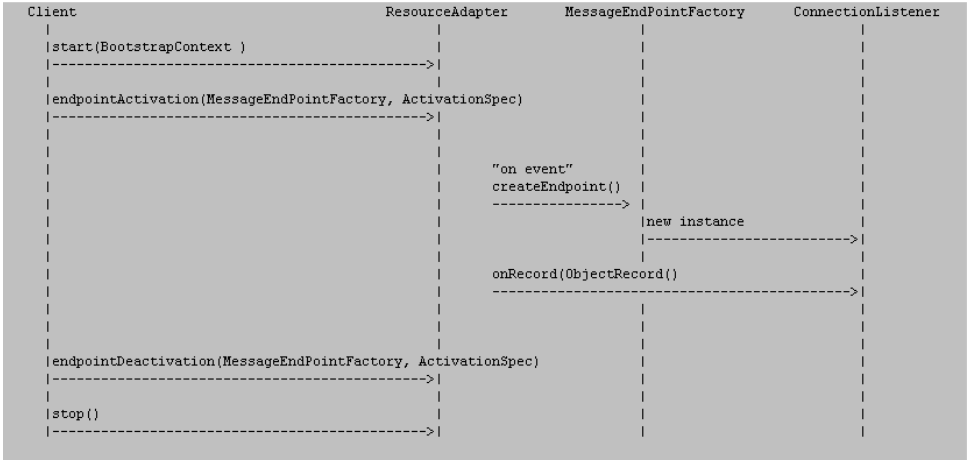
```
public interface ConnectionListener extends MessageListener
{
public void onException(Exception exception);
public ObjectRecord onRecord(ObjectRecord record);
}
```

Diagrammes

Obtention au préalable des informations de connexion (design-time) :



Cycle de vie de la classe ResourceAdapter :



4 Déploiement

Pour utiliser le connecteur au sein de l'application Connect-It vous devez premièrement créer un fichier de déploiement. Le SDK n'utilise pas le descripteur `ra.xml` de la norme JCA, mais son propre descripteur de déploiement. Ce fichier XML est basé sur la notion de contexte introduite par le framework *Spring*. Il doit être obligatoirement nommé `designtime-beans.xml` et se trouver à la racine de l'archive JAR du connecteur.

Les informations contenues sont :

- Le nom complet de la classe *com.hp.ov.cit.connector.spi.designtime.DesignTimeFactory*.
- Le nom complet de la classe *javax.resource.spi.ResourceAdapter*. Dans le cas d'une communication outbound, il s'agit de l'implementation de la classe *javax.resource.spi.ManagedConnectionFactory*.

  Un exemple est donné ci-dessous :

```
<beans>

<bean id="designTimeFactory" class="com.mycompany.myeis.MyEisDesignTime
Factory">
<property name="resourceAdapter">
<ref bean="resourceAdapter"/>
</property>
</bean>

<bean id="resourceAdapter" class="com.mycompany.myeis.MyEisManagedConne
ctionFactory"/>
```

```
</beans>
```

5 Configuration

Un certain nombre de fichier de configuration sont nécessaires à l'application *Connect-It* pour pouvoir utiliser le connecteur. Il est à noter que ce nom doit être unique parmi tous les autres connecteurs existants dans *Connect-It*. Il est donc demandé de suivre la convention de nommage des packages "à la Java". Considérons par exemple le nom *com.mycompany.myeis* pour notre connecteur.

Fichier description

C'est le fichier central pour la description proprement-dite du connecteur. Il regroupe l'ensemble des propriétés afférentes au connecteur telles que son nom unique, les références aux noms des fichiers définits ci-après, ainsi que sa clé d'activation. Ce fichier doit obligatoirement posséder l'extension `.dsc`. Il est conseillé de le nommer `myeis.dsc`.

Exemple:

```
{CONNECTORDESC
InternalName=com.mycompany.myeis
ParentInternalName=Application_connectors
Name=My EIS
HTMLHelp=This is a description of my connector
Key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXX
IconFile=myeis.bmp
Sched.CanUsePointer=0
Cnx.HasCnx=1
```

```
Wizard.File=myeis-wizard.xml
Java.Class=com.hp.ov.cit.container.RAContainer
Java.Configuration.File=myeis-config.xml
Java.JVMConfiguration.File=myeis-jvmconf.xml
Java.HasOptions=1
}
```

Fichier icône

Pour visualiser une icône, dans l'arbre de navigation des connecteurs, vous devez fournir un fichier bitmap au format 16x16. Le nom de ce fichier est par exemple `myeis.bmp`.

Fichier de configuration

Ce fichier comporte l'ensemble des propriétés JavaBeans qui doivent être configurées par l'utilisateur. Ce fichier permet également de spécifier quelles seront les propriétés comprises dans l'export de la configuration du scénario par la ligne de commande :

```
conitsvc -export[:<property file>] <scenario>
```

Exemple d'un fichier *myeis-config.xml* :

```
<configuration>
<property name="ra_url" type="String" export="true">
<definition>
<default/>
</definition>
<export>
<description>URL</description>
</export>
</property>
<property name="cs_userName" type="String" export="true">
<definition>
<default/>
</definition>
<export>
<description>User</description>
</export>
</property>
</configuration>
```

Fichier wizard

C'est un fichier de définition de wizard pour le connecteur au format XML.

Il permet de décrire l'enchaînement des pages permettant de configurer le connecteur au sein de *Connect-It*. Il contient par exemple une page de définition de la connexion. Les contrôle graphiques y sont également décrit en termes de notion (texte, case à cocher, bouton, ...), de label, de positionnement, ...

L'ensemble des propriétés JavaBeans qui doivent être configurées par l'utilisateur doivent y figurer. Il faut cependant respecter la règle de nommage suivante :

- Toute propriété relative à l'implémentation de la l'interface *javax.resource.spi.ResourceAdapter* doit être préfixée par *ra_*
- Toute propriété relative à l'implémentation de l'interface *javax.resource.cci.ConnectionSpec* pour le designtime (métadonnées) doit être préfixée par *mdcs_*
- Toute propriété relative à l'implémentation de l'interface *javax.resource.cci.ConnectionSpec* pour le runtime doit être préfixée par *cs_*
- Toute propriété relative à l'implémentation de l'interface *javax.resource.spi.ActivationSpec* doit être préfixée par *as_*

Exemple d'un fichier `myeis-wizard.xml` :

```
<wizard>

<page name="pgConnector">
<title>Connection</title>

<description>Configure connection to MyEIS</description>

<description>Enter the URL</description>
<control type="Textbox" name="ra_url">
<Value>$(GetValue[ra_url])</Value>
<label>URL</label>
<XOffset>2500</XOffset>
<labelLeft>1</labelLeft>
<Mandatory>1</Mandatory>
<MandatoryMsg>You must specify an URL value</MandatoryMsg>
<bind>Value</bind>
</control>

<description>Enter the user name</description>
<control type="Textbox" name="cs_userName">
<Value>$(GetValue[cs_userName])</Value>
<label>User</label>
<XOffset>2500</XOffset>
<labelLeft>1</labelLeft>
<bind>Value</bind>
</control>
```

```
<Transition>
<To script="true">{trConnector}</To>
</Transition>
</page>

</wizard>
```

Fichier de configuration de la JVM

Vous devez fournir à l'application un fichier de configuration du classpath pour démarrer la JVM.

Connect-It nécessite une configuration minimum quel que soit le connecteur construit sur le SDK. Cette configuration est décrite dans le fichier situé sous *CONNECT-IT_HOME*/*config*/*shared*/*jca-container-jvmconf.xml*. Vous devez donc par conséquent l'inclure dans votre propre fichier de configuration de la JVM.

Exemple d'un fichier `myeis-jvmconf.xml` :

```
<jvmConfiguration id="com.mycompany.myeis">
<jarLocation>./com.mycompany.myeis</jarLocation>
<jars>
<jar groupId="com.mycompany.myeis" optional="false" provided="true"
version="1.00" versionNeeded="true">myeis</jar>
</jars>
<import>../shared/jca-container-jvmconf.xml</import>
</jvmConfiguration>
```

6 Packaging

Le packaging final du connecteur au sein de l'instalation de Connect-It doit être le suivant :

```
Connect-It/
|
|- lib/
|  |- com.mycompany.myeis/
|     |- myeis-1.00.jar
|     |- myeis-3rdparty1.jar
|     |- myeis-3rdparty2.jar
|     |- ...
|
|- config/
|- com.mycompany.myeis/
|- myeis.bmp
|- myeis-jvmconf.xml
|- myeis.dsc
|- myeis-wizard.xml
|- myeis-config.xml
```

Note :

Pour une question d'unicité de nom, les répertoires de configuration et d'archives du dit connecteur doivent suivre la convention de nommage des packages "à la Java". D'où le nom *com.mycompany.myeis* utilisé dans l'exemple ci-dessus.

Archive Java

Le contenu de l'archive `myeis-1.00.jar` doit avoir la forme suivante :

```
myeis-1.00.jar
|
|- designtime-beans.xml
|
|- com/
|  |- mycompany/
|     |- myeis/
|        |- MyEisDesignTimeFactory.class
|        |- MyEisManagedConnectionFactory.class
|        |- MyEisConnectionManager.class
|        |- ...
|
|- META-INF/
|  |- Manifest.mf
```

7 Extension

Interface com.hp.ov.cit.connector.spi.ContainerContext

Lorsque le connecteur est instancié au travers de Connect-It, l'application fournit à l'implémentation de celui-ci une spécialisation de la classe <javax.resource.spi.BootstrapContext> permettant l'accès à des fonctionnalités spécifiques du conteneur. Cette classe de contexte fournit les possibilités ci-dessous:

Ecoute d'évènement

Il est possible d'être notifié des évènements d'exécution du scénario Connect-It. Ceci se fait au travers de l'enregistrement d'une classe d'écoute en utilisant les méthodes :

```
public void addContainerListener(ContainerListener listener);
public void removeContainerListener(ContainerListener listener);
```

Le SDK introduit à ce jour 2 types de classes d'écoute :

- com.hp.ov.cit.connector.spi.ExecutionListener : écoute des notifications de démarrage et d'arrêt du scénario.
- com.hp.ov.cit.connector.spi.SessionListener : écoute des notifications d'ouverture et de fermeture de session au sein de l'exécution d'un scénario.

Accès au chemin du scénario

Il est possible d'obtenir le chemin complet du scénario exécuté au travers de l'appel :

```
public String getScenarioAbsolutePath();
```

Si vous êtes intéressé par ces fonctionnalités, vous aurez certainement à écrire au sein de votre implémentation <javax.resource.spi.ResourceAdapter> le code suivant :

```
public void start(BootstrapContext bootstrapContext) throws ResourceAdapte
rInternalException
{
if (bootstrapContext instanceof ContainerContext)
{
//store this CIT context for use
}
else
{
//who is my container?
throw new ResourceAdapterInternalException();
}
}
```

Classe com.hp.ov.cit.connector.spi.designtime.ObjectTypeProviderEx

Cette classe est une implémentation spécifique de l'interface <com.hp.ov.cit.connector.spi.designtime.ObjectTypeProvider>. Elle permet à toute implémentation l'utilisant, de fournir au conteneur des informations supplémentaires concernant les types supportés.

L'interface <com.hp.ov.cit.connector.cci.ObjectType> donne le type de la classe Java pour contenir des données simples (valeurs entières, booléennes,..). Cependant pour certains types - spécialement les dates - une classe Java peut-être insuffisante pour décrire entièrement la sémantique d'un type (ex: date, date/heure ou heure). Cette classe particulière addresse cette problématique en donnant au conteneur une information additionnelle sur un type simple considéré au travers de la méthode :

```
public String getXSDBuiltinDatatype(ObjectType simpleType)
```

Cette méthode retourne le nom d'un type "built-in" de la spécification XML Schema [http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#built-in-datatypes] de façon à compléter la description d'un type simple.

Par défaut la classe de base n'apporte aucune information additionnelle sur les types simples manipulés.

Une utilisation type serait :

```
public class MyObjectTypeProvider extends ObjectTypeProviderEx
{
....

@Override
public String getXSDBuiltinDatatype(ObjectType simpleType)
{
if( simpleType instanceof MyDateObjectType)
{
return "date";
}
else if( simpleType instanceof MyDatetimeObjectType)
{
return "dateTime";
}
else if( simpleType instanceof MyTimeObjectType)
{
return "time";
}
else
{
//sorry ...no additionnal info along the Java class type
return null;
}
}
}
```

8 Utilisation

Le fonctionnement d'un connecteur développé à l'aide du SDK est lié à :

- une déclaration explicite de l'accès au SDK dans le certificat d'utilisation de Connect-It
- la génération d'une clé pour le connecteur créé à l'aide du SDK

Certificat d'utilisation

Le certificat d'utilisation active :

- le runtime permettant l'utilisation d'un connecteur créé à l'aide du SDK
- le menu permettant la génération d'une clé pour le connecteur nouvellement créé, clé utilisée par le runtime.

▶ Manuel Connect-It - Utilisation, Installation, Saisir le certificat d'utilisation.

Générer une clé

Une clé permet au connecteur de fonctionner.

Pour générer une clé :

1 Lancez l'éditeur de scénarios de Connect-It

2 Sélectionnez **Java/ Générer une clé d'activation SDK**

3 Dans la fenêtre qui s'affiche, renseignez :
   - Le nom de votre connecteur créé
   - Son comportement (mode production, mode consommation)
4 La clé générée doit être copiée dans le fichier de description
   ▶ Manuel Connect-It - SDK, section Fichier de description [page 65].

Cette clé est liée au certificat d'utilisation qui lui seul permet l'activation et le fonctionnement du connecteur.

I Annexe

9 Fichier Wizard

Cette section fournit la référence de la syntaxe utilisée pour le fichier XML de l'assistant de configuration du connecteur (wizard).

Structure générale

Un wizard est composé de pages. Chaque page peut posséder des contrôles de saisie, des labels et des descriptions. Chaque page définit une transition vers la page suivante.

```
<wizard>

<include/>
<property/>
<page>
<transition/>
</page>

</wizard>
```

Elément wizard

L'élément racine doit être *wizard*.

Les sous-éléments possibles sont :

Elément	Optionnel	Description
include	oui	Permet d'inclure des définitions provenant de fichiers externes.
property	oui	Permet de définir des propriétés scriptées.
page	oui	Définit les pages constituant le wizard.

Elément include

Permet d'inclure un fichier. La syntaxe est la suivante :

```
<include type="..." [basedir="..."]>the file name</include>
```

Attribut	Optionnel	Description
type	non	Définit le type d'inclusion
basedir	oui	Définit le répertoire du fichier à inclure
included	oui	Permet d'ignorer ou non l'élément

Les types d'inclusions sont les suivant :

- string
- wizard

Inclusion de type string

Permet d'inclure un fichier de ressources (chaînes de localisation). Par défaut, le chemin du fichier à inclure est relatif au fichier courant.

Les chaînes définies dans ce fichier sont accessibles par la syntaxe: $(IDS_*NAME_OF_THE STRING*).

Par exemple :

Considérons le fichier `myeisstrings.str`

```
EIS_TITLE, "Title for the EIS"
EIS_DESCRIPTION, "Description of the EIS"
....
```

L'utilisation de ressources dans le fichier wizard sera alors :

```
<wizard>
<include type="string">eisstrings.str</include>
<title>$(IDS_EIS_TITLE)</title>
```

```
...
</wizard>
```

Note :

- L'accès aux ressources n'est effectif que pour tout élément défini après cette inclusion.
- L'inclusion est prise en compte lors de la génération du wizard. Sa valeur ne peut donc pas être scriptée.

Inclusion de type wizard

Permet d'inclure un autre fichier wizard. Les éléments pouvant spécifier des inclusions de ce type sont *wizard* et *page*.

Il est possible de passer des paramètres au wizard inclus, qui seront alors accessibles au travers de la syntaxe :

```
$(GetValue[NAME_OF_THE PARAMETER])
```

Par exemple, pour passer le paramètre *myParameter* ayant pour valeur *myValue* au wizard `myIncludedWizard.xml`, la syntaxe sera alors:

```
<include type="wizard" myParameter="myValue>myIncludedWizard.xml</include>
```

Elément page

Un wizard est composé d'une succession de pages. Les attributs possibles sont :

Attribut	Optionnel	Description
name	non	Définit le nom de la page. Chaque nom de page est unique.
included	oui	Permet d'ignorer ou non l'élément.

Les sous-éléments possibles sont :

Elément	Optionnel	Description
transition	non	Définit la transition vers la page suivante.
description	oui	Permet d'ajouter une description pour la page, une section de la page ou un contrôle.

Elément	Optionnel	Description
title	oui	Définit le titre de la page.
property	oui	Permet de définir des propriétés scriptées.
control	oui	Définit les contrôles constituants le page.
linebreak	oui	Définit un saut de ligne.
separator	oui	Définit un séparateur horizontal.

```
<page name="..." included="...">


<title/>
<image/>
<description/>
<property/>
<control/>
<linebreak/>

<separator/>
<transition/>

</page>
```

Note :

La première page du wizard d'un connecteur doit obligatoirement être nommée **pgConnector.**

Elément property

Une propriété est une valeur de type basique tel que *string* ou *long*. Les attributs possibles sont :

Attribut	Optionnel	Description
name	non	Définit le nom de la propriété.
included	oui	Permet d'ignorer ou non l'élément.
script	oui	Permet de spécifier un contenu scripté.

Exemple :

```
<page name="myPage">
<property name="IsVisible" type="Long" script="true">RetVal = 1</property>
</page>
```

```
<property name="DelimString" script="true">RetVal = ""</property>
```

L'utilisation d'une propriété se fait suivant la syntaxe *property full path* référençant le chemin complet (sans la racine) de la propriété dans l'arborescence XML.

Exemple :

```
<visible script="true">{myPage.IsVisible} &lt;&gt; 1</visible>
<value script="true">{DelimString}</value>
```

Elément control

Permet de définir un contrôle graphique. Les attributs possibles sont :

Attribut	Optionnel	Description
name	non	Définit le nom de la propriété.
type	non	Définit le type de contrôle. Celui-ci doit être unique au sein d'un page.
included	oui	Permet d'ignorer ou non l'élément.
script	oui	Permet de spécifier un contenu scripté.

Les sous-élements possibles quel que soit le type du contrôle considéré sont :

Elément	Optionnel	Type	Description
visible	oui	booléen	Spécifie si le contrôle est visible ou non.
enabled	oui	booléen	Spécifie si le contrôle est grisé ou non.
readonly	oui	booléen	Spécifie si le contrôle est éditable ou non.
mandatory	oui	booléen	Spécifie si le contrôle nécessite ou non une valeur.
mandatorymsg	oui	string	Spécifie le message d'erreur si aucune valeur n'est spécifiée alors que l'attribut mandatory est présent et égal à 1.
label	oui	string	Définit texte au dessus du contrôle.

Elément	Optionnel	Type	Description
labelleft	oui	booléen	Si '1' ou 'true', positionne le label à gauche de celui-ci.
xoffset	oui	long	Définit l'espace à gauche du contrôle.
bind	oui	string	Spécifie quelles sont les éléments du contrôle dont les valeurs prises en compte lors de la validation de la page associée. Exemple : <bind>value</bind> permet de prendre en compte la valeur de l'élément <value>.
property	oui	string	Permet de définir des propriétés scriptées.

D'autres sous-éléments sont disponibles suivant le type du contrôle considéré. Les principaux contrôles et leur sous-éléments sont :

Type de contrôle	Sous-élément	Type	Description
textbox	value	string	Valeur d'entrée du texte.
	multiline	long	0 = simple ligne sinon pourcentage du poids du contrôle
	password	boolean	Valeur spécifiant s'il s'agit d'un champ crypté. 1 = champ crypté
checkbox	value	boolean	Valeur spécifiant si le contrôle est coché ou non.
	caption	string	Label du contrôle.
combobox	value	string	Valeur de l'item sélectionné.
	values	string	Liste des items (label=valeur) possibles séparés par des virgules. Exemple : <values>English=en,French=fr<values>

Type de contrôle	Sous-élément	Type	Description
numbox	value	long	Valeur numérique pour le contrôle.
	minvalue	long	Spécifie la valeur minimum.
	maxvalue	long	Spécifie la valeur maximum.
label	caption	string	Label du contrôle.
fileedit	value	string	Chemin du fichier sélectionné.
	openmode	long	Définit le type d'édition : ■ 1 = OPEN ■ 2 = SAVE ■ 4 = OPEN_DIR ■ 8 = SAVE_DIR ■ 16 = APPEND
	filters	string	Définit un filtre fichier. Exemple : <filters>XML files (*.xml)|*.xml|XML-Schema files (*.xsd)|*.xsd|</filters>
	defext	string	Extension par défaut à utiliser. Exemple : <defext>txt</defext>
	serializationId	string	Définit l'id de ce contrôle de sélection de fichier. Plusieurs contrôles peuvent utiliser le même id. Cet id est utilisé pour sauvegarder l'emplacement du dernier fichier sélectionné.
optionbuttons	value	sting	Valeur de l'item sélectionné.

Type de contrôle	Sous-élément	Type	Description
	values	string	Liste des items (label=valeur) possibles séparés par des virgules. Exemple : <values>ISO-8859-1=0,UTF-8=1,Shift-JIS=2<values>
	border	boolean	Spécifie si le contrôle comporte un cadre ou non.

Exemple :

```
<control type="TextBox" name="Server">
<value>$(GetValue[Server])</value>
<caption>$(IDS_SERVER_LABEL)</caption>
<xoffset>2500</xoffset>
<bind>value</bind>
</control>
```

Attribut bind

L'attribut *bind* est l'attribut permettant de relier un contrôle à une propriété de configuration du connecteur. Pour tous les contrôles actuellement supportés par le SDK, seule la valeur *value* est valide. Lorsqu'il est spécifié sur un contrôle de nom 'cs_myprop', cela veut dire que la valeur de l'élément <value> du contrôle sera passée au connecteur comme valeur pour la propriété de configuration 'cs_myprop', c'est-à-dire comme la valeur de la propriété 'myprop' de la propriété **ConnectionSpec** du connecteur.

Gestion des mots de passe

La gestion des propriétés de configuration de type Mot de passe nécessite un traitement particulier dans les wizards. Si la propriété contenant le mot de passe est 'cs_password', le nom du contrôle wizard pour cette propriété doit être 'clearcs_password'.

Exemple :

```
<control type="TextBox" name="clearcs_password">
<value>$(GetValue[cs_password])</value>
<password>1</password>
<label>$(IDS_PASSWORD_LABEL)</label>
<xoffset>2500</xoffset>
<labelleft>1</labelleft>
<bind>value</bind>
</control>
```

Eléments linebreak et separator

Ces éléments sont utilisés pour mettre en forme la page d'un wizard. Les attributs possibles sont :

Attribut	Optionnel	Description
included	oui	Permet d'ignorer ou non l'élément.

Elément transition

Toute page doit comporter un élément de transition. Cet élément permet de spécifier quelle sera la page suivante. Les attributs possibles sont :

Attribut	Optionnel	Description
script	oui	Permet de spécifier un contenu scripté.

Exemples :

```
<transition><to>nextPage</to></transition>

<transition>
<to script="true">
if( $(GetValue[ShowAdvancedWiz]) = 1 ) then
RetVal = "pgAdvanced"
else
RetVal = {trConnector}
end if
</to>
</transition>
```

Note :

La transition de la dernière page du wizard d'un connecteur doit être égale à la valeur scriptée *{trConnector}*.

Attribut script

Les wizards supportent des scripts simples empruntant la syntaxe du Basic. Ces scripts sont évalués à l'exécution du wizard.

L'attribut *script* est disponible sur tous les éléments contenant une valeur. Il permet de spécifier la valeur de l'élément sous la forme d'une expression scriptée qui sera évalué lorsque l'attribut prend la valeur *true*.

Exemple:

```
<... script="true">
if {Protocol.Value} = "ftp" or {Protocol.Value} = "http" then
RetVal = 1
else
RetVal = 0
end if
</...>
```

Dans les scripts Basic des wizards, la syntaxe {...} permet de référencer la valeur d'un contrôle ou d'un propriété du wizard. Ces valeurs sont référencées à l'aide du chemin complet (sans la racine) de la propriété dans l'arborescence XML.

Attribut included

Cet attribut est disponible sur la plupart des éléments. Il est optionnel. Il contient une valeur booléenne, informant si l'élément considéré doit être ignoré ou non.

Les différentes valeurs que peuvent prendre cet attribut sont :

- *0* ou *1* (ou tout autre valeur que *0*)
- *false* ou *true*
- Une expression composite utilisant les opérateurs *and*, *or* et *not*

Lorsque que la valeur de cet attribut est *false*, le contenu de l'élément auquel il appartient sera ignoré.

 Note :

La valeur de cet attribut est évaluée lors de la génération du wizard, et non lors de son exécution. L'inclusion d'un élément ne peut donc pas dépendre de la valeur d'un contrôle ou de tout autre expression scriptée. La valeur de cet attribut est donc en général évaluée à l'aide de la fonction **GetValue**.

Fonctions

Les fonctions définies ici ne sont pas des fonctions de script Basic, mais des fonctions évaluées lors de la génération du wizard, et non lors de son exécution.

Elles se présentent sous la forme :

```
$(FunctionName[param1,param2<,optionalparam>,...])
```

Fonction GetValue

Cette fonction permet de retrouver dynamiquement une valeur au sein du wizard. C'est la fonction la plus utilisée dans un wizard, car c'est elle qui permet de récupérer la valeur courante d'une propriété de configuration du connecteur.

La syntaxe est la suivante :

```
$(GetValue[name,default])
```

Le paramètre *name* spécifie le nom de la valeur à retrouver. Le paramètre *default* définit une valeur par défaut si celle-ci n'est pas trouvée.

Il existe plusieurs valeurs dont le nom est pré-défini :

- OSUnix : retourne *1* si la plateforme est Unix, *0* sinon.
- OSWindows : retourne *1* si la plateforme est Windows, *0* sinon.
- WizardDir : retourne le chemin complet du répertoire wizard d'installation (CONNECT-IT_HOME/config/wiz).
- NameID : retourne le nom du connecteur
- ShowAdvancedWiz : retourne *1* si le wizard est en mode avancé, *0* sinon.
- ConfigDir : retourne le chemin complet du répertoire de configuration du connecteur.

Lorsque la fonction *GetValue* est appelée, la recherche de la valeur se fait sur :

1. Les valeurs spécifiques définies dans le fichier de description.
2. Les propriétés de configuration du connecteur.
3. Les valeurs prédéfines.

Exemple :

```
<value>$(GetValue[mylogin])</value>

<property name="trConnector" script="true">
if( $(GetValue[Cnx.HasCnx, 1]) = 1 then
RetVal = "pgConnection"
else
...
</property>

<control type="checkbox" name="UseWindowsRegistry" included="$(GetValue[OS
Windows])">
<value>$(GetValue[UseWindowsRegistry])</value>
<caption>$(IDS_SERVER_LABEL)</caption>
<xoffset>2500</xoffset>
<bind>value</bind>
</control>
```

Fonction Dump

Cette fonction permet de formater une chaîne de caractères pour l'utiliser dans un script. Elle échappe les guillemets contenus dans la chaîne, et entoure la chaîne de guillemets. Cette fonction peut être très utile lors de l'utilisation dans un script d'une chaîne récupérée à l'aide de la fonction **GetValue** ou d'une chaîne provenant d'un fichier `.str`. La syntaxe est la suivante :

```
$(Dump[string])
```

Exemple :

```
<value script="true">RetVal = $(Dump[$(GetValue[theValue])])</value>
```

Fonction EspaceCommas

Cette fonction permet d'échapper les virgules d'un chaîne. Cette fonction peut être utile lorsque la chaîne est un sous-élément d'une chaîne utilisant la virgule comme caractère de séparation (l'élément values du contrôle *optionbuttons* par exemple). La syntaxe est la suivante :

```
$(EscapeCommas[string])
```

Fonction File

Cette fonction permet de récupérer le chemin complet d'un fichier. La syntaxe est la suivante :

```
$(File[name,basedir])
```

Le paramètre *name* spécifie le nom du fichier. Le paramètre *basedir* définit le répertoire du fichier. Par défaut, il s'agit du répertoire du fichier wizard considéré.

Exemple :

```
<image>$(File[myfile.bmp])</image>
```

10 Fichier de configuration

Cette section fournit la référence de la syntaxe utilisée pour le fichier de configuration.

La structure générale du fichier est la suivante :

```
configuration>

<property>
<definition>
<default/>
</definition>
<export>
<description/>
</export>
</class>
</property>

<property>
<definition>
<default/>
</definition>
<export>
<description/>
</export>
</class>
</property>

</configuration>
```

Elément configuration

L'élément raçine doit être *configuration*. Les sous-éléments possibles sont :

Elément	Optionnel	Description
property	oui	Définit la ou les propriétés nécessaires au code Java du connecteur.

Elément property

Spécifie une propriété de configuration Java.

Les attributs possibles sont :

Attribut	Optionnel	Type	Description
name	non	string	Définit le nom de la propriété.
type	non	string	Spécifie le type de la propriété.
export	oui	boolean	Indique si la propriété doit être prise en compte lors de l'export (option -export).

Les sous-éléments possibles sont :

Elément	Optionnel	Description
definition	oui	Définition de la propriété.
export	oui	Définition de l'export.
class	oui	Définition de la classe Java correspondante.

Elément definition

Possède les sous-éléments :

Elément	Optionnel	Type	Description
default	oui	string	Spécifie la valeur par défaut utilisée lors de l'initialisation du wizard.

Elément export

Possède les sous-éléments :

Elément	Optionnel	Type	Description
description	oui	string	Indique la description utilisée lors de l'exportation de la propriété. Apparaît en commentaire dans le fichier de propriétés exporté.

Elément class

A chaque type de propriété est implicitement associé une classe Java. Cet élément permet de surcharger la classe implicite du type de la propriété.

L'exemple ci-dessous déclare une propriété de type *String* qui correspondra à une propriété JavaBean de classe *java.net.URI*.

```
<property name="myURIProperty" type="String" export="true">
<class>java.net.URI</class>
</property>
```

Type de propriétés

Le tableau suivant décrit les types de propriétés supportés ainsi que leur type de propriété JavaBean par défaut.

Type	Type JavaBean
Boolean	java.lang.Boolean
Byte	java.lang.Byte
Short	java.lang.Short
Long	java.lang.Integer

Type	Type JavaBean
LingInt	java.lang.Long
Float	java.lang.Float
Double	java.lang.Double
String	java.lang.String
Memo	java.lang.String
Date	java.util.Date
Time	java.sql.Time
Timestamp	java.sql.Timestamp
Password	java.lang.String
File	java.io.File
Url	java.net.URL

Pour connaître l'ensemble des types JavaBean supportés, se référer à la documentation sur les JavaBeans.

11 Fichier de configuration de la JVM

Cette section fournit la référence de la syntaxe utilisée pour le fichier de configuration de la JVM.

La structure générale du fichier est la suivante :

```
<jvmConfiguration>

<jarLocation/>
<jarLocation/>

<jars>
<jar/>
<jar/>
<jar/>
</jars>

<jvmOptions>
<jvmOption/>
<jvmOption/>
</jvmOptions>

<import/>
<import/>

</jvmConfiguration>
```

Elément jvmConfiguration

L'élément racine doit être *jvmConfiguration*.

Les attributs possibles sont :

Attribut	Optionnel	Type	Description
id	non	string	Définit un identifiant unique pour la configuration considérée.

Les sous-éléments possibles sont :

Elément	Optionnel	Type	Description
jarLocation	oui	string	Définit les chemins du classpath utilisé par le connecteur.
jars	oui		Définit les archives utilisées par le connecteur.
import	oui	string	Permet d'inclure un classpath provenant d'un fichier externe.
jvmOptions	oui		Permet de définir les options de la JVM.

Elément jarLocation

Le classpath du connecteur est formé d'un ou plusieurs chemins référençant les diverses archives (fichiers .jar ou .zip) nécessaires à l'exécution du code. Pour chaque connecteur il est possible de définir ces chemins de recherche des archives. La valeur d'un chemin est soit relative au répertoire lib d'installation de Connect-It, soit un chemin absolu. Il est à noter que la recherche des archives se fait dans l'ordre dans lequel sont déclarés ces chemins.

Exemple :

```
<jarLocation>./com.mycompany.myeis</jarLocation>
<jarLocation>c:/myEIS/myEISPath</jarLocation>
```

Par défaut, si aucun élément *jarLocation* n'est spécifié, le chemin par défaut est celui du répertoire **lib** d'installation de Connect-It.

Elément jars

Les sous-éléments possibles sont :

Elément	Optionnel	Type	Description
jar	oui	string	Définit une entrée d'archive pour le class-path.

Elément jar

Les attributs possibles sont :

Attribut	Optionnel	Type	Défaut	Description
groupId	non	string		Définit un identifiant de groupe pour l'archive.
provided	oui	boolean	true	Indique si l'archive considérée provient des chemins de recherche pour le class-path ou doit être fournie par l'utilisateur. La valeur 'true' indique qu'elle est fournie par l'installation (chemins de recherche), l'attribut 'optional' est alors ignoré.

Attribut	Optionnel	Type	Défaut	Description
optional	oui	boolean	false	Spécifie si l'archive est optionnelle. La valeur 'false' indique que l'archive doit obligatoirement exister dans les chemins de recherche pour le classpath, ou dans le classpath additionnel définit soit dans l'application (entrée 'Configurer la JVM' du menu 'Java') soit au niveau du connecteur (page 'Configurer la JVM' du Wizard).
version	oui	string		Permet de suffixer l'archive par sa version. Le nom complet de l'archive est alors `name-version.jar`. Si cette extension n'est pas trouvée, une nouvelle recherche se fera en utilisant le nom complet `name-version.zip`.

Attribut	Optionnel	Type	Défaut	Description
versionNeeded	oui	boolean	true	Indique si le suffixe de version pour la recherche de l'archive est obligatoire. ■ La valeur 'true' indique que la recherche se fait uniquement sur le nom versionné. ■ La valeur 'false' indique que la recherche se fait d'abord sur le nom versionné, puis sur le nom brut et ce chemin par chemin.

La valeur doit référencer le nom de l'archive que l'on souhaite ajouter.

Exemple d'entrée de classpath pour la librairie `xercesImpl-2.6.2.jar` fournie par l'installation de l'application :

```
<jar groupId="xerces" optional="false" provided="true" version="2.6.2" ver
sionNeeded="true">xercesImpl</jar>
```

Elément jvmOptions

Cet élément permet de définir des options supplémentaires à prendre en compte par la JVM.

Les sous-éléments possibles sont :

Elément	Optionnel	Type	Description
jvmOption	oui	string	Définit une option de la JVM.

Exemple :

```
<jvmOptions>
<jvmOption>-Xmx125m</jvmOption>
<jvmOption>-Dcom.sun.management.jmxremote</jvmOption>
</jvmOptions>
```

Elément import

En plus de la configuration propre au connecteur, il est possible de spécifier une (ou plusieurs) configuration additionnelle pour la JVM. Ces déclarations proviennent alors d'un (ou plusieurs) fichier supportant une syntaxe identique. Suivant l'endroit ou la déclaration d'import est faite, les déclarations viennent se superposer avant ou après les définitions courantes. La valeur doit référencer le chemin relatif au fichier que l'on doit importer.

Exemple :

```
<import>../shared/jca-container-javaconf.xml</import>
```

12 Fichier de description

Cette section fournit la référence de la syntaxe utilisée pour le fichier de description.

Structure du fichier

La structure générale du fichier est la suivante :

```
{CONNECTORDESC
//property list
//property name=property value
Name=
InternalName=
...
}
```

Propriétés

Le tableau suivant résume les propriétés des connecteurs :

Propriété	Type	Optionnel	Valeur par défaut	Description
Propriétés communes				

Propriété	Type	Optionnel	Valeur par défaut	Description
Name	string	non		Nom affiché du connecteur.
InternalName	string	non		Nom interne du connecteur (unique).
ParentInternal-Name	string	oui		Nom du noeud parent auquel il appartient.
HTMLHelp string oui Description au format html.				
Key	string	non		Clé d'activation.
Icône				
IconFile	string	non		Chemin relatif de l'icône du connecteur (.bmp)
Programmateurs				
Sched.CanUsePointer	boolean	oui	true	Le SDK ne fournit pas de support pour les pointeurs. Cette valeur doit être explicitement mise à 0. *Sched.CanUsePointer=0*
Cache				
Cache.Support-Cache	boolean	oui	true	Le SDK ne fournit pas de support pour le cache de metadonnées. Cette valeur doit être explicitement mise à 0. *Cache.Support-Cache=0*
Timezone				

Propriété	Type	Optionnel	Valeur par défaut	Description
Tmz.HandleServerDelay	boolean	oui	true	Le SDK ne fournit pas de support pour la gestion du décalage avec le serveur. Cette valeur doit être explicitement mise à 0. *Tmz.HandleServerDelay=0*
External formats				
ExtFmt.Use	boolean	oui	true	Le SDK ne fournit pas de support pour la gestion des formats étendus. Cette valeur doit être explicitement mise à 0. *ExtFmt.Use=0*
Wizard				
Wizard.File	string	oui		Chemin relatif du fichier Wizard.
Java				
Java.Class	string	non		Spécifie la classe Java façade du connecteur. Doit être : `Java.Class=com.hp.ov.cit.container.RAContainer`
Java.Configuration.File	string	oui		Chemin relatif du fichier de configuration.
Java.JVMConfiguration.File	string	oui		Chemin relatif du fichier de configuration de la JVM.
Java.HasOptions	string	oui	false	Il est conseillé de mettre cette valeur à 1. Java.HasOptions=1

Propriété	Type	Optionnel	Valeur par défaut	Description
Java.Support-Proxy	boolean	oui	false	La configuration d'un serveur mandataire est-elle supportée ?
Java.PriorTo-Jdk15ProxyRegi-stration	boolean	oui	false	L'enregistrement du serveur mandataire utilise-t-il la fonctionnalité introduite par la classes <java.net.ProxySelector> ?
Divers				
RedeployOnCha-nge	string	oui		Spécifie la liste des propriétés qui nécessitent de redéployer le connecteur si elles ont été modifiées. Exemple : `RedeployOnChange=cs_CacertsFile cs_KeystoreFile as_KeystorePassword`

Exemple

Voici un exemple d'un fichier de configuration pour un connecteur de type *outbound* construit sur le SDK.

```
{CONNECTORDESC
Name=MyEIS
InternalName=com.mycompany.myeis
ParentInternalName=com.mycompany
HTMLHelp=Connector to interact with my eis
Key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXX
IconFile=myeis.bmp

EventDriven=0
SupportParallelization=1
Transac.CanSupportTransactions=1
Cnx.HasCnx=1
Cnx.CanDisableReconnection=0
Wizard.File=myeis-wizard.xml
```

```
Java.Class=com.hp.ov.cit.container.RAContainer
Java.Configuration.File=myeis-config.xml
Java.JVMConfiguration.File=myeis-jvmconf.xml
Java.HasOptions=1

// The following properties must always have these values
Sched.CanUsePointer=0
Cache.SupportCache=0
Tmz.HandleServerDelay=0
ExtFmt.Use=0
}
```

Information additionnelles

Multiples descriptions

Un fichier de description peut en fait contenir plusieurs descriptions, chacune correspondant à une section *CONNECTORDESC*. Bien qu'il soit en général conseillé d'écrire un fichier de description par connecteur, cette possibilité peut parfois s'avérer utile pour définir des catégories de connecteur, ou pour gérer différentes versions d'un même EIS.

Hiérarchie des connecteurs

La propriété *ParentInternalName* permet de spécifier le nom interne du noeud parent, ou catégorie, dans la hiérarchie des connecteurs. Les catégories sont également définies dans des fichiers de description, mais de manière simplifiée :

```
{CONNECTORDESC
InternalName=...
ParentInternalName=...
Name=...
HTMLHelp=...
IconFile=...
}
```

Si aucun *ParentInternalName* n'est spécifié, la catégorie (ou le connecteur) sera située à la racine de la hiérarchie.

Un certain nombre de catégories sont prédéfinies dans Connect-It :

Catégorie	Nom interne
Connecteurs applicatifs	Application_connectors
Connecteurs de type protocole	Protocol_connectors
Connecteurs ERP	ERP_connectors
Connecteurs Inventaire	Gateways

13 Code Java

JavaBeans

Types supportés

Un certain nombre d'interfaces provenant des spécifications JCA doivent être implémentées sous la forme de JavaBeans. Parmi ces interfaces, celles utilisées par le SDK sont les suivantes :

```
javax.resource.spi.ManagedConnectionFactory
javax.resource.spi.ResourceAdapter
javax.resource.cci.ConnectionSpec
javax.resource.spi.ActivationSpec
```

Le tableau suivant résume les types de valeurs autorisés pour leurs propriétés :

java.lang.Boolean
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Byte
java.lang.Short
java.lang.Long
java.lang.Float

java.lang.Character

Le SDK étend cette liste à d'autres types fréquemment utilisés. Le tableau suivant résume ces types supportés :

java.util.Date
java.sql.Time
java.sql.Timestamp
java.io.File
java.net.URL
java.net.URI

Validation

On peut imaginer par exemple qu'une propriété d'un objet JavaBean ne peut prendre que certaines valeurs en fonction d'une autre propriété. Comme l'objet ne contrôle pas l'ordre dans lequel les propriétés seront mises à jour, le SDK fournit une alternative à ce problème au travers de l'interface :

```
public interface ValidatingBean
{
public void validate() throws InvalidPropertyException;
}
```

Cette interface permet de gérer une phase de validation ou d'initialisation sur l'objet JavaBean l'implémentant une fois que toutes ses propriétés ont été mises à jour.

Logging

Le SDK utilise le framework *Jakarta Commons Logging (JCL)* pour enregistrer des messages au journal de Connect-It (logs). Pour utiliser cette fonctionnalité à partir d'une classe Java, vous devez inclure le code suivant :

```
package com.mycompany.myeis;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyEISClass
{
private static final Log log = LogFactory.getLog(MyEISClass.class);

...
}
```

JCL définit un niveau de priorité associé à chaque message. Les niveaux utilisés par le journal de Connect-It sont les suivants :

- error - Messages d'erreurs.
- info - Messages d'information.
- warn - Messages d'avertissement.
- debug - Messages de débogage. Seulement visibles dans le journal lorsque le mode 'debug' est activé.

Pour enregistrer un message au journal de Connect-It, utilisez les méthodes de l'interface *org.apache.commons.logging.Log* :

```
log.error(Object message);
log.error(Object message, Throwable t);
log.warn(Object message);
log.warn(Object message, Throwable t);
log.info(Object message);
log.info(Object message, Throwable t);
log.debug(Object message);
log.debug(Object message, Throwable t);
```

Support Log4J

Le framework JCL permet d'unifier l'accès à une implémentation de logging : Log4J, JDK Logging, etc.

Par défaut Connect-It repose sur une configuration de la librairie Log4J. Ainsi tout les messages loggués par la couche Log4J par appel direct ou via l'API JCL seront pris en compte par Connect-It.

Support JDK logging

Connect-It ajoute le support pour le framework de logging fournit par le JDK grâce une configuration statique. La configuration statique par défaut est décrite par le fichier `<JRE_HOME>\lib\logging.properties`.

Lors de l'instanciation du connecteur, le niveau de logging du logger racine du JDK est modifié de façon à ce qu'il corresponde à celui configuré pour l'application Connect-It. L'ensemble des évènements de log, obtenus par appel direct au framework JDK logging, sont alors redirigés vers l'application.

Internationalisation

Le SDK repose sur le mécanisme standard d'internationalisation de Java. Vous pouvez utiliser ce mécanisme au sein de votre code pour loguer vos messages.

Il faut pour cela écrire un (ou plusieurs) fichier(s) au format *properties* qui va (vont) contenir les chaînes de caractères éligibles à l'internationalisation.

Exemple d'utilisation

Fichier `com/mycompany/myeis/i18n/mymessages.properties`

```
connection.error = Erreur de connexion.
execution.failed = Echec de l'exécution.
```

Fichier `com/mycompany/myeis/i18n/mymessages_en.properties`

```
connection.error = Connection error.
execution.failed = Execution failed.
```

Fichier `com/mycompany/myeis/MyEISClass.java`

```
package com.mycompany.myeis;

import java.util.ResourceBundle;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyEISClass
{
private static final ResourceBundle bundle = ResourceBundle.getBundle("com
.mycompany.myeis.i18n.mymessages");
private static final Log log = LogFactory.getLog(MyEISClass.class);

public void execute()
{
try
{
...
}
catch(Exception e)
{
log.error(bundle.getString("execution.failed"), e);
}
}
}
```