Sedna XML:DB API Documentation

15th February 2009

Sedna XML:DB API Features....3

XML:DB API Emulations....3

Hierarchical Collections....3

Binary Resources....3

XUpdate Language....3

Getting started, the basics....3

Registering the driver....3

URIs for the XML:DB DatabaseManager....3

Making a connection....4

Inserting and Retrieving Resources....4

XML Resources....4

DOM Document Retrieval....5

Text XML Document Retrieval....5

SAX Document Retrieval....5

Inserting a DOM Document....5

Inserting a SAX Document....5

Binary Resources....6

Binary BLOB Content Retrieval....6

Inserting a Binary BLOB....6

Java Object Retrieval....6

Inserting a Java Object....6

Querying....6

Querying against a resource....6

Querying against a collection....7

Querying against the entire database....7

Using Resource Sets Effectively....7

Setting XQuery variables from within Java....7

Querying local and nonpersistent XML data....7

Supported Java Types & corresponding XSD types....7

Updating via Sedna XQuery Update....8

Update example....8

Updating via XUpdate....8

Updating a particular document....8

Updating a collection....8

Updating with neither Update language....9

Updating a DOM Document....9

Updating a Text XML Document....9

Updating a Binary BLOB Resource....9

Updating a Java Object....9

Transactions....9

Inserting multiple documents within a transaction....9

Updating using a Transaction....10

Manipulating the DB Connection Pool....10

Manipulating Sedna XML:DB API XML usage....10

Extending with Custom Service plugins....11

Good practices....11

Loading enormous XML documents....11

Embedding user input free text into XQuery....11

Sedna XML:DB API Features

- Meets all the requirements for "XML:DB API Core Level 1" compliance.
- Via this API, Sedna supports Binary BLOBS as well as Java Object storage.
- Via this API, Sedna is hierarchical collections friendly.
- Via this API, Sedna can execute a XQuery / XPath directly against a resource or a collection.
- Via this API, Sedna supports the XUpdate standard for updating data[1].
- Zero dependencies. Other than xmldb.jar interface APIs (9kb) this package requires nothing other than a JVM.
- All XML processing is 100% JAXP based.
- Compiled binary jar file is very light weight (under 50kb).
- Makes full use of Sedna's ACID Transactions capability. Manual / Auto - Commit / Rollback 100% supported.
- Very small memory footprint and very fast execution, the server carries the burden wherever possible.
- Extensible, supports custom XML:DB Service plugins on XML:DB Collections.
- Stands up to immense usage stress, underlying Database Connection Pooling Manager.

XML:DB API Emulations

Hierarchical Collections

Sedna does not support hierarchical collections, it supports one collection deep from the root database and nothing further.

XML:DB API states "An XML database MAY expose collections as a hierarchical set of parent and child collections."

Sedna XML:DB API emulates this "hierarchical collections" functionality.

As usual for Sedna, all first level collections will be known by their respective name, e.g. "myCollection".

If however, the XML:DB API user wishes to create a child collection in the collection "myCollection" called "mySub", Sedna internally understands this as a new first level collection known as "myCollection / mySub".

When this XML:DB API implementation connects to the root collection or what Sedna understands as the database root, it does not see "myCollection / mySub" as a child collection.

Binary Resources

Sedna does not natively support storing binary objects.

However, this feature has been emulated in the Sedna XML:DB API. Sedna XML:DB API does this by encoding binary data into Base 64 format, which is then stored inside an XML document, ready for the database.

Unlike other XML:DB API implementations, this implementation allows one to store Java based Objects, under the condition they implement `java.io.Serializable` as well as byte arrays (`byte[]`).

Storing large binary objects like images in any database, relational or tree based is a bad idea. Databases are designed for data that is to be queried. Storing binary objects will only over stress any database, needlessly. Despite the Sedna XML:DB API supporting binary resources it is not recommended.

Storing Java Objects is another method, they are naturally small and one can insert, retrieve and update custom "View" Objects quickly based on their name within a collection which could act as an id reference. A downside to this method is the intrinsic data cannot be queried via XQuery. If one wishes to store Java Objects inside an XML database, it is recommended that one investigates Sun's JAXB [2]

XUpdate Language

Currently there is a clear standard of querying XML Databases (XQuery). However different proposals currently exist about updating existing data. In 2000 the XUpdate working group created a draft outlining a language called "XUpdate" to update data within XML Documents / Databases. Despite being an XML language that is not easy for programmers or humans alike, it is actually the most widely adopted update language among XML Databases today. The W3C are finalising the XQuery Update Facility which will certainly super cede the XUpdate language and become "the standard".

Sedna currently supports a declarative node level update language language which is based upon the XQuery update proposal by Patrick Lehti[3], this is an intuitive and powerful update language.

There is an XUpdate / Sedna Update language translator for use within the Sedna XML:DB API. All XUpdate commands and elements are supported, bar three. Go to page 8 for more details.

Getting started, the basics

Registering the driver

```
import org.xmldb.api.base.Database;
import org.xmldb.api.DatabaseManager;

Database sednaDatabase;
Class clazz = Class.forName("net.cfoster.sedna.DatabaseImpl");
sednaDatabase = (Database)(clazz.newInstance());
DatabaseManager.registerDatabase(sednaDatabase);
```

URIs for the XML:DB DatabaseManager

The URI format for connecting to a Sedna database instance is described in the following format

```
xmldb:sedna://[hostAddress]/[database]/[collectionpath]
```

1 All XUpdate commands are supported bar xupdate:if, xupdate:cdata and xupdate:update.
2 Java API for XML Binding. https://jaxb.dev.java.net/
3 Design and Implementation of a Data Manipulation Processor for an XML Query Language - August 2001

Any of the following URIs are acceptable for the Sedna XML:DB API DatabaseManager

```
xmldb:sedna://127.0.0.1/mydb
xmldb:sedna://127.0.0.1:5050/mydb
xmldb:sedna://www.host.com:1234/mydb/my-collection
xmldb:sedna://somehost/bigdb/a-collection/sub-collection
xmldb:sedna://abc/db/a/b/c/d/e/f/g
```

If the given URI does not state a port, the XML:DB API assumes the port is 5050 as this is the default Sedna port.

Making a connection

Creating a connection to a Sedna Database instance and disconnecting.

```
import org.xmldb.api.DatabaseManager;
import org.xmldb.api.Collection;

Collection collection =
DatabaseManager.getCollection("xmldb:sedna://myhost/mydatabase",
"username", "password");

// database transaction code

collection.close(); // MUST close collection
```

When calling `DatabaseManager.getCollection(String uri, String user, String pass)`, this implementation may create a new underlying Socket to a server and is the only way of creating underlying Socket connections, it returns a `Collection` object.

When calling `collection.getChildCollection(String name)` to get another Collection object, or any other method which returns a new Collection object from the initial one does not create a new underlying Socket, it reuses the initial underlying Socket connection.

Calling `close()` on the original `Collection` or any `Collection` object spawned from the original will notify the Database Connection Pool that the underlying Socket connection has no further use. It is then the Database Connection Pool's decision to either close the underlying socket or add it to an "idle" list, ready to be reused / recycled.

Due to the behaviour of the Database Pooling mechanism, it is possible that when calling `DatabaseManager.getCollection(String uri, String user, String pass)` a Collection will be returned which uses a "recycled" Database Socket connection. "Recycled" in the sense that it was either:

- Abandoned (i.e. a user forgot to call `close()` on a `Collection` object previously spawned from `DatabaseManager`.
- Notified by the XML:DB API user that it has no further use, i.e. a user has called `close()` on a `Collection` object which was previously spawned from `DatabaseManager`.

It is important that a developer always closes a collection when they have finished all necessary transactions. As this will notify the Sedna XML:DB API Database Connection Pool that the underlying connection instance is no longer required for active use.

It is possible to alter how the Database Connection Pool works, this is covered later in the documentation, go to page 10 for further details.

Inserting and Retrieving Resources

A Resource is a loose term for either an XML fragment or a binary object. If it is a XML fragment, it may well be an entire XML document, or a subset (fragment) of an original XML document. If it is a binary object, it is binary data, whether that be an image, CSV text file or even a Java object.

In XML:DB API, two classes exist called `XMLResource` and `BinaryResource`, they are both derived from the same interface class `Resource`.

XML Resources

Sedna XML:DB API's implementation of `XMLResource` was developed with both memory efficiency and performance in mind. By default, it prefers to store XML data in a String instead of a DOM structure representation to save memory. However if one were to set or get the content of the resource via the methods `getContentAsDOM()` or `setContentAsDOM(Node n)`, then the instance will store a DOM representation of the XML data for faster subsequent access.

DOM Document Retrieval

Retrieving a document from the database using a known ID, ultimately working with the result as a DOM Document object.

```
String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.getResource(id);

Document doc = (Document) resource.getContentAsDOM();
```

Text XML Document Retrieval

Retrieving a document from the database using a known ID, ultimately working with the result as text XML.

```
String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.getResource(id);

String doc = (String) resource.getContent();
```

SAX Document Retrieval

Retrieving a document from the database using a known ID, utlimately using a SAX content handler to handle the document.

```
String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.getResource(id);

// A custom SAX Content Handler is required
// to handle the SAX events
ContentHandler handler = new MyContentHandler();

resource.getContentAsSAX(handle);
```

Inserting a DOM Document

Storing a new DOM document into the database using a known ID.

```
// Document is assumed to be a valid DOM document.
Document document;

String id = "gladiator-2000";
XMLResource resource =
(XMLResource)collection.createResource(id,
XMLResource.RESOURCE_TYPE);

resource.setContentAsDOM(document);
collection.storeResource(resource);
```

Inserting a SAX Document

Sometimes, it may be convenient to create an `XMLResource`'s content via SAX.

Using a SAX ContentHandler to store a new document into the database using a known ID.

```
// File containing the XML to be inserted
String fileName = "gladiator-2000.xml";

String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.createResource(id,
XMLResource.RESOURCE_TYPE);

ContentHandler handler = resource.setContentAsSAX();

XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setContentHandler(handler);
reader.parse(new InputSource(fileName));

collection.storeResource(resource);
```

Note, that by default an XMLReader does not parse Comments, CDATA or DTD Declarations. However the Sedna XML:DB API can (with a tweak).

```
// File containing the XML to be inserted
String fileName = "gladiator-2000.xml";

String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.createResource(id,
XMLResource.RESOURCE_TYPE);

ContentHandler handler = resource.setContentAsSAX();

XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setContentHandler(handler);
reader.setProperty("http://xml.org/sax/properties/lexical-handler",
handler);
reader.parse(new InputSource(fileName));

collection.storeResource(resource);
```

Binary Resources

Binary BLOB Content Retrieval

Retrieving a binary BLOB from the database. The BLOB is identified with a known ID. The database will need to determine the data is binary and return the proper Resource type.

```
String id = "gladiator-2000-img";
BinaryResource resource =
(BinaryResource) collection.getResource(id);

// Return value of getContent must be defined in the specific
language mapping
// for the language used. For Java this is a byte array.
byte[] img = (byte[]) resource.getContent();
```

Inserting a Binary BLOB

Insert a binary BLOB into the database. The BLOB is identified with a known ID.

```
String id = "image.gif";
BinaryResource resource =
(BinaryResource) collection.createResource(id,
BinaryResource.RESOURCE_TYPE);

FileInputStream in = new FileInputStream(id);
byte[] b = new byte[in.available()];
in.read(b);
in.close();

resource.setContent(b);
collection.storeResource(b);
```

Java Object Retrieval

Retrieving a Java Object from the database. The Java Object is identified with a known ID. When casting, one must cast to the appropriate datatype or a ClassCastException will be thrown.

```
String data[];
String id = "my-string.string-array";

BinaryResource resource =
(BinaryResource) collection.getResource(id);

// Must cast the data back to the appropriate object type
data = (String[])resource.getContent();
```

Inserting a Java Object

Inserting a Java Object that implements `java.io.Serializable` into the database. The Java Object is identified with a known ID.

```
String data[] = { "alpha", "beta", "gamma", "delta" };
String id = "my-string.array";
BinaryResource resource =
(BinaryResource)collection.createResource(id,
BinaryResource.RESOURCE_TYPE);

resource.setContent(data);
collection.storeResource(b);
```

Querying

Querying Collections or XML documents using either “XPath” or “XQuery” can be accomplished via either `XPathQueryService` or `XQueryService`.

Querying against a resource

Querying a document in a collection for all movies with the title of Gladiator, ultimately working with the results as Document DOM Nodes.

```
// note: XQuery is perfectly fine in this instance also:
String xpath = "/movie/title='Gladiator'";
String id = "films.xml";

XQueryService service =
(XQueryService)collection.getService("XQueryService", "1.0");

ResourceSet resultSet = service.queryResource(id, xpath);

ResourceIterator results = resultSet.getIterator();
while (results.hasMoreResources())
{
XMLResource resource =
(XMLResource) results.nextResource();
Document result = (Document)resource.getContentAsDOM();
}
```

Querying against a collection

Querying a collection for all documents where an id code is between certain values, ultimately working with the results as Document DOM Nodes.

```
String query = "for $x in //aRecord \n"+
"where $x/id>100 and $x/id<200 \n"+
"return $x";

XQueryService service =
(XQueryService)collection.getService("XQueryService", "1.0");

ResourceSet resultSet = service.query(query);

ResourceIterator results = resultSet.getIterator();
while (results.hasMoreResources())
{
XMLResource resource =
(XMLResource) results.nextResource();
Document result = (Document)resource.getContentAsDOM();
}
```

Querying against the entire database

The previous two methods work on the assumption that the developer is either querying a resource or a collection, so XPath `doc()` and `collection()` functions don't need to be written.

However, if a developer wishes to query the database as a whole, for instance querying multiple documents and collections and concatenating the results, they can explicity state which `doc()`/`document()` and/or `collection()` they wish to use inside the XQuery statement itself. They can use either of the two methods shown above (`queryResource(String id, String query)` or `query(String query)`), because if one explicitly states these facts in the XQuery, it will make no difference to the ResourceSet returned.

```
String query = "collection(\"films\")/movie/title='Gladiator'";

XQueryService service =
(XQueryService)collection.getService("XQueryService", "1.0");

ResourceSet resultSet = service.query(query);

ResourceIterator results = resultSet.getIterator();
while (results.hasMoreResources())
{
XMLResource resource =
(XMLResource) results.nextResource();
Document result = (Document)resource.getContentAsDOM();
}
```

Using Resource Sets Effectively

When the Sedna XML:DB API sends a query, the client downloads each resource result incrementally, the client does not download the entire set initially. The Sedna network protocol does not state how many resources it is sending, therefore when calling something a method like `getSize()` will force the downloading of all resources into a local memory cache.

Also, for each resource the Sedna server does send, it does not state whether or not it is the last resource in the set. The only way the client understands whether or not it has received the last resource is by making the mistake of asking the server for a resource that doesn't exist, only to be told that it has reached the end of the set. Because of this, `hasMoreResources()` actually downloads the next available resource as well as `nextResource()` as that internally calls `hasMoreResources()`

This best way to use this implementation is to get a `ResourceIterator` and iterate through the results.

However, this technique may not applicable for certain situations, so the following list of `ResourceSet` methods gives an insight into the inner workings of Sedna XML:DB API.

`getSize()`
: Forces the downloading of all resources.

`getMembersAsResource()`
: Forces the downloading of all resources.

`removeResource(long index)`
: May download all resources up till index.

`addResource(Resource res)`
: Does not download anything.

`getResource(long index)`
: May download all resources up till index.

`clear()`
: Notifies the server no more resources are required and clears the local memory cache.

Setting XQuery variables from within Java

The method `void declareVariable(String qname, Object initialValue)` within `XQueryService` allows one to set an XQuery variable directly from Java.

This allows one to insert data from Java into an XQuery statement in both a convenient manner and removing room for human error, see the following example

```
xqService.declareVariable("a","hello world");
xqService.declareVariable("b",new int[] { 1,2,3 });
xqService.declareVariable("c",new Date());

xqService.query("$a,$b,$c");
```

Querying local and nonpersistent XML data

One can query against XML documents that are not stored within the database. For instance documents stored locally or taken directly from a Web Service message. This allows a developer to utilize Sedna's high performance XQuery engine without having to actually store data within it first. See the following example.

```
Document myDoc = loadFromLocalXML();
xqService1.declareVariable("myDoc", myDoc);

xqService1.query(
"for $x in $myDoc/a/b/c return $x/string()");

Document[] collection = loadLotsOfXMLDocs();
xqService2.declareVariable("myData", collection);

xqService2.query(
"for $y in $myData return $y/people/person");
```

Supported Java Types & corresponding XSD types

In this implementation of the XML:DB API, the following Java types are supported and map to the corrosponding XML Schema data types.

Java Data Type	XML Schema Type
Boolean, Boolean[], boolean[]	xs:boolean, xs:boolean*, xs:boolean*
Byte, Byte[], byte[]	xs:byte, xs:byte*, xs:base64Binary
Integer, Integer[], int[]	xs:int, xs:int*, xs:int*
Short, Short[], short[]	xs:short, xs:short*, xs:short*
Long, Long[], long[]	xs:long, xs:long*, xs:long*
Float, Float[], float[]	xs:float, xs:float*, xs:float*
Double, Double[], double[]	xs:double, xs:double*, xs:double*
String, String[]	xs:string, xs:string*
Date, Date[]	xs:date, xs:date*
Document, Document[]	document(), document()*
Element, Element[]	element(), element()*
Attr, Attr[]	attribute(), attribute()*
Comment, Comment[]	comment(), comment()*
ProcessingInstruction, ProcessingInstruction[]	processing-instruction(), processing-instruction()*
Object[][4]	item()*

Updating via Sedna XQuery Update

Sedna supports a powerful and intuitive update language that is based on the XQuery update proposal by Patrick Lehti with a number of improvements[5].

Explaining how to use this update language and its syntax is outside the scope of this document, but explaining how to apply it via this XML:DB API is within scope.

To execute a Sedna update statement, one must use the custom Sedna XML:DB API Service `SednaUpdateService` class, found in the `net.cfoster.sedna` package.

Update example

When using the Sedna Update Service, unlike `XQueryService` the developer must explicitly state which document and/or collection they wish to update against via the appropriate XPath `document()` and `collection()` functions.

```
import net.cfoster.sedna.SednaUpdateService;

String updateStr = "UPDATE\n"+
"insert <warning>High Blood Pressure!</warning>\n"+
"preceding "+
"document(\"hospital.xml\")//blood_pressure[systolic>180]";

SednaUpdateService service =
(SednaUpdateService)collection.getService("SednaUpdateService",
"1.0");

service.update(updateStr);
```

Updating via XUpdate

Sedna does not natively support the XUpdate language, however this XML:DB API can translate XUpdate language into the Sedna Update language before executing, all XUpdate elements/commands are supported, except 3.

Command/Element	Supported
xupdate:insert-before	YES
xupdate:insert-after	YES
xupdate:append	YES
xupdate:remove	YES
xupdate:rename	YES
xupdate:variable	YES
xupdate:value-of	YES
xupdate:update	NO[6]
xupdate:cdata	NO[7]
xupdate:if	NO[8]

For the following examples, consider the following XUpdate statement can be returned as a String via the method `getMyUpdateStatement()`

```
<xupdate:modifications version="1.0"
xmlns:xupdate="http://www.xmldb.org/xupdate">

<xupdate:insert-before
select="/addresses/address[@id = 1]/phone[@type='home']">

<xupdate:element name="warning">
<xupdate:attribute name="warning-level">
serious
</xupdate:attribute>

High blood pressure
</xupdate:element>
</xupdate:insert-after>
</xupdate:modifications>
```

Updating a particular document

Updating a particular resource document with a given XUpdate Statement.

```
String id = "hospital.xml";

XUpdateQueryService service =
(XUpdateQueryService)collection.getService("XUpdateQueryService"
, "1.0");

String updateStr = getMyUpdateStatement(); // above

service.updateResource(id, updateStr);
```

Updating a collection

```
XUpdateQueryService service =
(XUpdateQueryService)collection.getService("XUpdateQueryService"
, "1.0");

String updateStr = getMyUpdateStatement(); // above

service.update(updateStr);
```

4 Object array may contain different data types, but each array item's data type must be a data type already listed.
5 Sedna Update Language - http://modis.ispras.ru/sedna/progguide/ProgGuidesu6.html
6 Sedna does not support direct atomic (element/attribute content) level updates in this manner [22nd July 2007].
7 Sedna does not support the updating and appending of CDATA [22nd July 2007].
8 Usage of this command is ambiguous, ill-defined by the XUpdate Working Group [22nd July 2007].

Updating with neither Update language

Updating a DOM Document

Update an existing DOM document stored within the database.

```
String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.getResource(id);

Document document =
(Document)resource.getContentAsDOM();

// Change document ...

resource.setContent(document);
collection.storeResource(resource);
```

Updating a Text XML Document

Update an existing Text XML document stored within the database.

```
String id = "gladiator-2000";
XMLResource resource =
(XMLResource) collection.getResource(id);

String document = (String) resource.getContent();

// Change document ...

resource.setContent(document);
collection.storeResource(resource);
```

Updating a Binary BLOB Resource

Update an existing Binary BLOB stored within the database.

```
String id = "image.gif";
BinaryResource resource =
(BinaryResource)collection.getResource(id);

byte[] data = (byte[])resource.getContent();

// change binary data ...

resource.setContent(data);
collection.storeResource(resource);
```

Updating a Java Object

Update an existing Java Object which implements `java.io.Serializable` stored within the database.

```
String id = "my-java-object.obj";
BinaryResource resource =
(BinaryResource)collection.getResource(id);

MyObject obj = (MyObject)resource.getContent();

obj.setValueX(23423.234d);
obj.setValueY("a","b",'c');
// etc...

resource.setContent(obj);
collection.storeResource(resource);
```

Transactions

Sedna allows one to carry out Transactions (The ability to perform multiple inserts, updates and deletes then choosing either to "commit" or "rollback" all changes). Fortunately the XML:DB API specifies a `Transaction` service, which is implemented in the Sedna XML:DB API.

At the time of writing this document, OZONE is the only other database that provides an XML:DB API driver that supports Transactions [22nd July 2007]. So Sedna is now the second XML Database to provide this functionality via an XML:DB API Implementation.

By default, the Sedna XML:DB API is in an "auto-commit" state, every statement commits automatically. But the developer can take control.

Inserting multiple documents within a transaction

```
// Documents are assumed to be valid DOM documents.
Document document1;
Document document2;

String id1 = "gladiator-2000";
String id2 = "gonein60seconds-2000";

TransactionService transaction =
(TransactionService)collection.getService("TransactionService",
"1.0");

transaction.begin(); // this turns auto-commit off!

XMLResource resource1 =
(XMLResource)collection.createResource(id1,
XMLResource.RESOURCE_TYPE);

resource1.setContentAsDOM(document1);
collection.storeResource(resource1);

XMLResource resource2 =
(XMLResource) collection.createResource(id2,
XMLResource.RESOURCE_TYPE);

resource2.setContentAsDOM(document2);
collection.storeResource(resource2);

transaction.commit();
// now back into an auto-commit state.
```

When calling `begin()` on a TransactionService the client breaks out of the `auto-commit` state and enters the start of a user defined transaction.

Subsequently, when calling `commit()` or `rollback()` the client reverts back into the `auto-commit` state.

Updating using a Transaction

Any operation, including inserts, updates and even queries are considered as part of a transaction.

```
import net.cfoster.sedna.SednaUpdateService;

String updateStr1 = "UPDATE\n"+
"insert <warning>High Blood Pressure!</warning>\n"+
"preceding //blood_pressure[systolic>180]";

String updateStr2 = "UPDATE\n"+
"insert <critical>Extra high blood pressure!</critical>\n"+
"preceding //blood_pressure[systolic>250]";

SednaUpdateService service =
(SednaUpdateService)collection.getService("SednaUpdateService",
"1.0");

TransactionService transaction =
(TransactionService)collection.getService("TransactionService",
"1.0");

service.updateCollection(updateStr1);

service.updateCollection(updateStr2);

transaction.rollback();
```

Note that despite this property being set on a Collection object, Transactions are executed in the context of a "socket connection". A developer cannot run two unique transaction operations against two different collections unless they have created them from two different `Database.getCollection(String uri, String user, String pass)` calls which is essentially two different connections to the database.

Manipulating the DB Connection Pool

The Database Connection Pool, manages the underlying Socket connections with the Sedna server. By default a Sedna Server can cope with a maximum of 10 concurrent connections.

The Sedna XML:DB API, depending on it's configuration will either:

- Wait forever, until a connection can be created and returned.
- Wait a set amount of milliseconds to return a connection before throwing an XMLDBExecption

The Database Connection Pool also uses connection recycling, if a user wishes to get a connection to a Sedna instance and an idle connection already exists which was previously created with the same username/password it may well reuse / recycle that connection.

The Database Connection Pool's behaviour can be modified by calling `setProperty(String name, String value)` on a `Database` object, and viewed by calling `getProperty(String name)` on a Database object.

The values which one can set/get and their meanings are as follows:

`conn-max-active`
: Maximum number of concurrent database connections the Sedna XML:DB API can use. Set 0 for no limit.

`conn-max-idle`
: Maximum number of concurrent idle database connections to retain in the pool. Set -1 for no limit.

`conn-max-wait`
: Maximum time to wait for a database connection to become available in milliseconds. An XMLDBException is thrown if this timeout is exceeded. Set to -1 to wait indefinitely.

`conn-remove-abandoned-timeout`
: Amount of connection inactivity time in milliseconds before a connection is considered abandoned and thus will either turn idle or be closed.

`conn-remove-abandoned`
: Whether or not to close abandoned connections or keep them open indefinitely.

`conn-log-abandoned`
: Whether or not Sedna XML:DB API should print to the standard output stream when a connection has been identified as abandoned.

`conn-abandoned-check-interval`
: Amount of time in milliseconds between a batch check of all connections within the pool, to see if any have been abandoned in order to be recycled or closed.

When creating a console application with Sedna XML:DB API, it would be best to set `conn-max-active` to 0. Otherwise when calling `close` on a Collection object, the connection will remain open and thus the application will remain open.

```
Database sednaDatabase;
Class clazz = Class.forName("net.cfoster.sedna.DatabaseImpl");
sednaDatabase = (Database)(clazz.newInstance());

sednaDatabase.setProperty("conn-max-idle","0");
```

When creating software which remains in a persistent state (like a Servlet) and may well be hit over and over in quick succession, it is recommended to pool database connections by increasing `conn-max-active`.

```
Database sednaDatabase;
Class clazz = Class.forName("net.cfoster.sedna.DatabaseImpl");
sednaDatabase = (Database)(clazz.newInstance());

sednaDatabase.setProperty("conn-max-idle","4");
sednaDatabase.setProperty("conn-log-abandoned","true");
sednaDatabase.setProperty("conn-max-wait","-1");
```

Manipulating Sedna XML:DB API XML usage

It is possible to change how Sedna XML:DB API internally works with XML content. It is important to outline these settings despite no obvious need to alter them. Apart from extraordinary circumstances, it is not recommended to change these values. To change/view them, one can call `setProperty(String name, String value)` and `getProperty(String name)` methods on a `Database` object.

`db-namespace-aware`
: Document Builder, namespace aware

`db-validating`
: Document Builder, validate document(s)

`db-ignore-element-content-whitespace`
: Document Builder, ignore element content whitespace

`db-expand-entity-references`
: Document Builder, expand entity references

`db-ignore-comments`
: Document Builder, ignore Comments

`db-coalescing`
: Document Builder, coalescing

`db-resolve-entities`
: Document Builder, resolve and include entities (e.g. DTDs)

`sp-namespace-aware`
: SAX Parser, namespace aware

`sp-validating`
: SAX Parser, validate document(s)

`sp-resolve-entities`
: SAX Parser, resolve and include entities (e.g. DTDs)

`xmlresource-to-dom-returns-document`
: Whether `getContentAsDOM()` within `XMLResource` returns a `Document` Object (cast to `Node`) or a genuine `Node` Object

Extending with Custom Service plugins

In Sedna XML:DB API, each `Collection` object by default comes with the following Services

`XQueryService`	`XPathQueryService`
`XUpdateQueryService`	`TransactionService`
`SednaUpdateService`	`IndexManagementService`
`ModuleManagementService`	`RoleManagementService`
`UserManagementService`	

It is possible to create and add custom Service plugin components to be used in `Collection` objects. To do this, see the following example.

```
import net.cfoster.sedna.DatabaseImpl;

DatabaseImpl.registerPluginService("com.mypackage.MyCustomService");
```

The full list of methods that exist on `DatabaseImpl` which one can use for `Service` plugin manipulation are as follows

`static void registerPluginService(String fullClassName)`
: Register a custom Sedna XML:DB API Service plugin class which implements the XML:DB `Service` class.

`static void deregisterPluginService(String fullClassName)`
: Deregister a custom Sedna XML:DB API Service plugin class.

`static String[] getPluginServices()`
: Retrieve a list of current custom custom Sedna XML:DB API Service plugins.

Good practices

Loading enormous XML documents

When inserting a very large XML document into Sedna, it is possible to stream the document direct from it's location via a valid URI, whether that be an ftp or http server or the local disk, see the following example.

```
import net.cfoster.sedna.SednaUpdateService;

SednaUpdateService sus =
(SednaUpdateService)collection.getService("SednaUpdateService",
"1.0");

sus.update("LOAD \"http://domain.org/xxl-doc1.xml\"
\"xxl-doc1.xml\"");

sus.update("LOAD \"ftp://domain.org/xxl-doc2.xml\"
\"xxl-doc2.xml\"");
```

Embedding user input free text into XQuery

When inserting text into an XQuery which may well of been typed in from a end user, it is not recommended to try to insert this into the XQuery directly, instead it is recommended to bind a Java variable to an XQuery variable, see the following example.

```
XQueryService xqs =
(XQueryService)collection.getService("XQueryService", "1.0");

xqs.declareVariable("user", userIdString);
xqs.declareVariable("pass", passwordString);

xqs.query("doc(\"users.xml\")//account[@user=$user and
@pass=$pass]");
```