GeoTools Data Store
Performance and Recommendations

Submitted To: Program Manager
GeoConnections
Victoria, BC, Canada

Submitted By: Jody Garnett
Refractions Research Inc.
Suite 400 –1207 Douglas St.
Victoria, BC, V8W
jgarnett@refractions.net
Phone: (250) 383-3022
Fax: (250) 383-2140

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

This document describes the performance and scalability characteristics of the new Geotools2 DataStore API.

The design of the GeoTools DataSource API was reviewed as part of the Validating Web Feature Server design process. We have taken an active role in the development of the new DataStore API allowing us to address our previous concerns.

This work has resulted in:

- A clean design that meets our requirements
- Dramatic gains in efficiency and scalability

We have been very pleased with this process and the results we have achieved. The GeoTools Team and our partners at The Open Planning Project have been very helpful throughout this process.

2 DESIGN OF DATASTORE API

The goals of the GeoTools DataStore API are:

- Scalability by way of providing a Streaming API
- Clean mapping to File access (in addition to Database access)
- Fine granularity to allow efficient filters and joins
- Transaction support
- Strong transaction support by way of Feature Locking

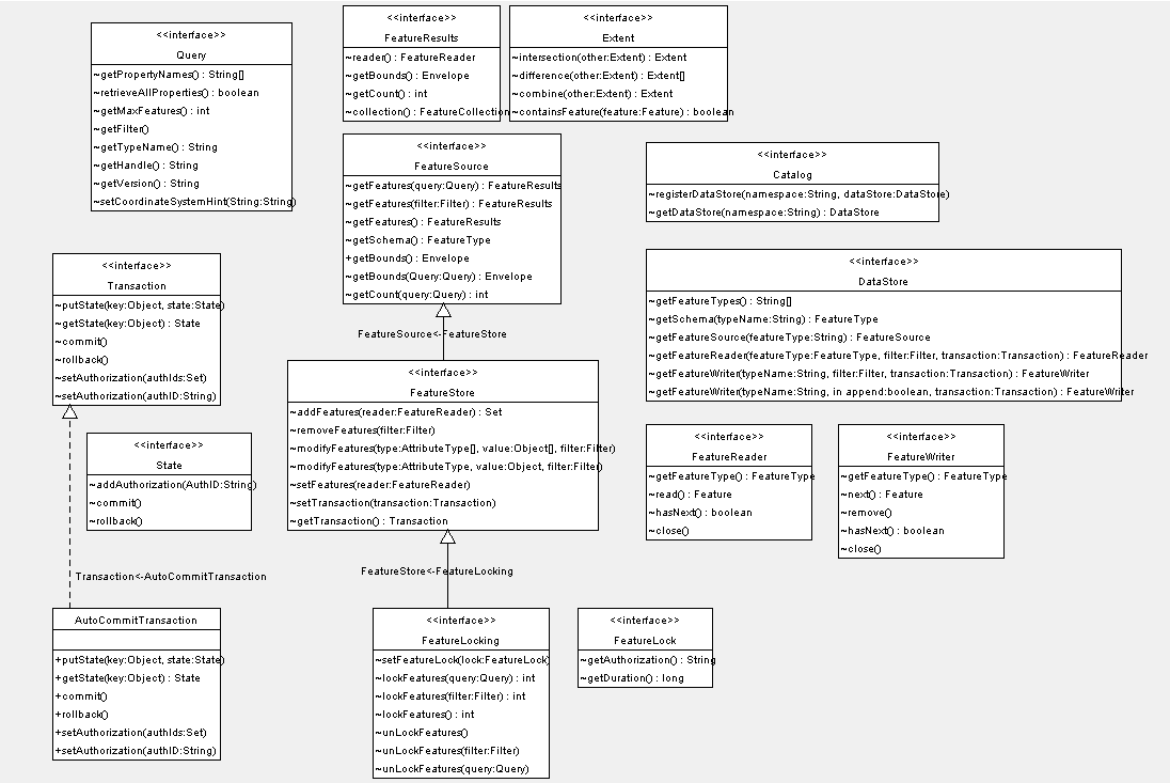

Figure 1: DataStore API

2.1 Original GeoTools DataSource API

The original DataSource GeoTools API:

- Represents a Single Table or File
- Is modeled after a Database Connection
- Implements Transactions as a State of the DataSource
- Operates by Loading Spatial Data into a "FeatureCollection" in memory
- Does not support Feature Locking

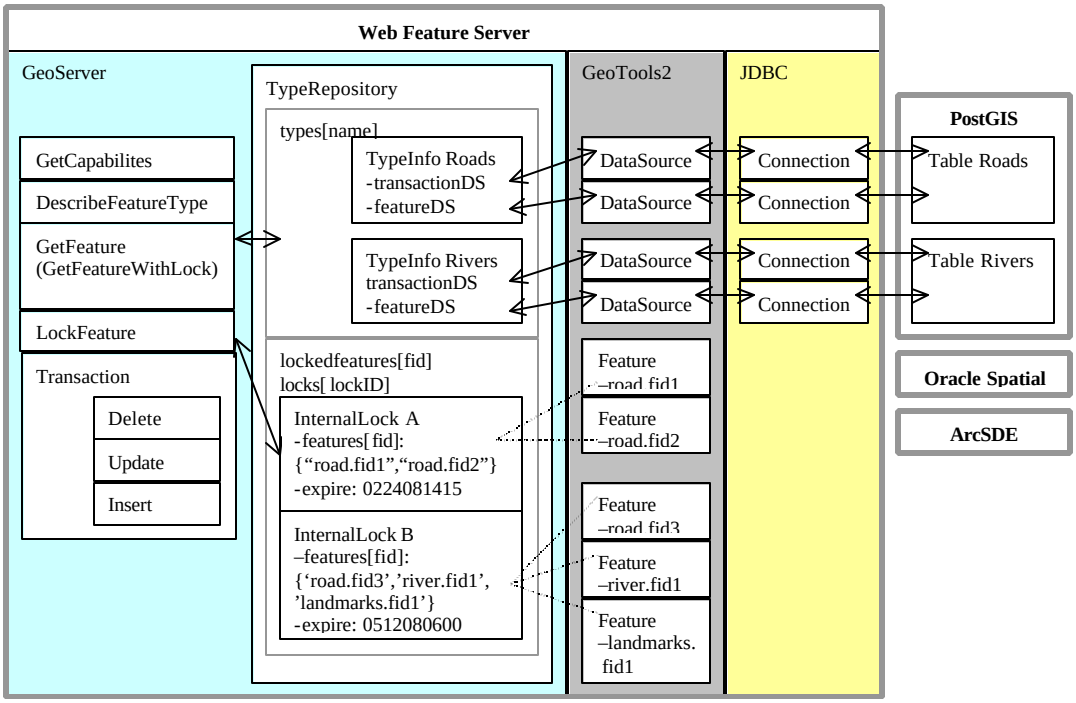

Figure 2: GeoServer 1.0 and Original GeoTools DataSource

The interaction of GeoServer with the DataSource API occurs through the TypeRepository. The TypeRepository maintains two DataSource instances for each table. The TypeRepository also maintains InternalLock instances recording Feature Lock information.

2.1.1 Limitations

The original DataSource Design has several limitations:

- Inefficient use of database connections
- Scalability Limited by Memory
- Lacks Strong Transaction Support

2.2 Milestone 1 DataSource Design Proposal

In the document “Transaction Web Feature Server Design”, an extension to the DataSource API was proposed.

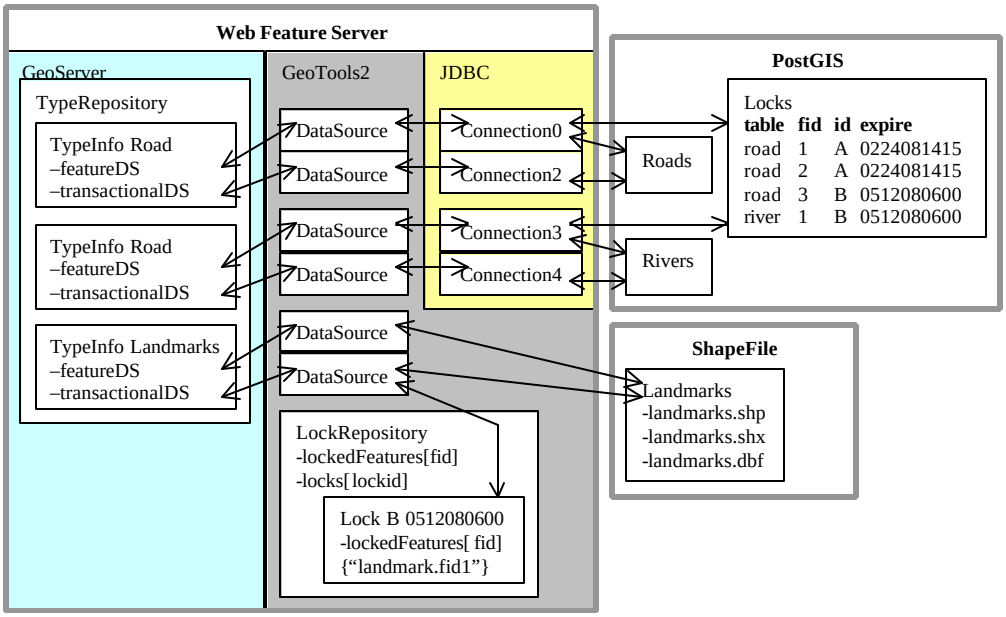

Figure 3: Milestone 1 Proposal

In this proposal strong transaction support is implemented using a Locking Table in PostGIS. DataSources without native lock support make use of a lock repository provided by GeoTools.

2.2.1 Limitations

The Milestone 1 DataSource Design Proposal maintains several limitations:

- Inefficient use of database connections
- Scalability Limited by Memory

The Milestone 1 DataSource Design Proposal was a strong influence in the final design of the New DataStore API.

2.3 Final GeoTools DataStore API

The new GeoTools DataStore API:

- Is modeled after Accessing a "Stream" of Spatial Data
- Implements Transactions separately from a DataStore
- Operates by providing a "FeatureReader" to iterate through Spatial Data
- Implements Feature Locking as part of the API (not as an extension)

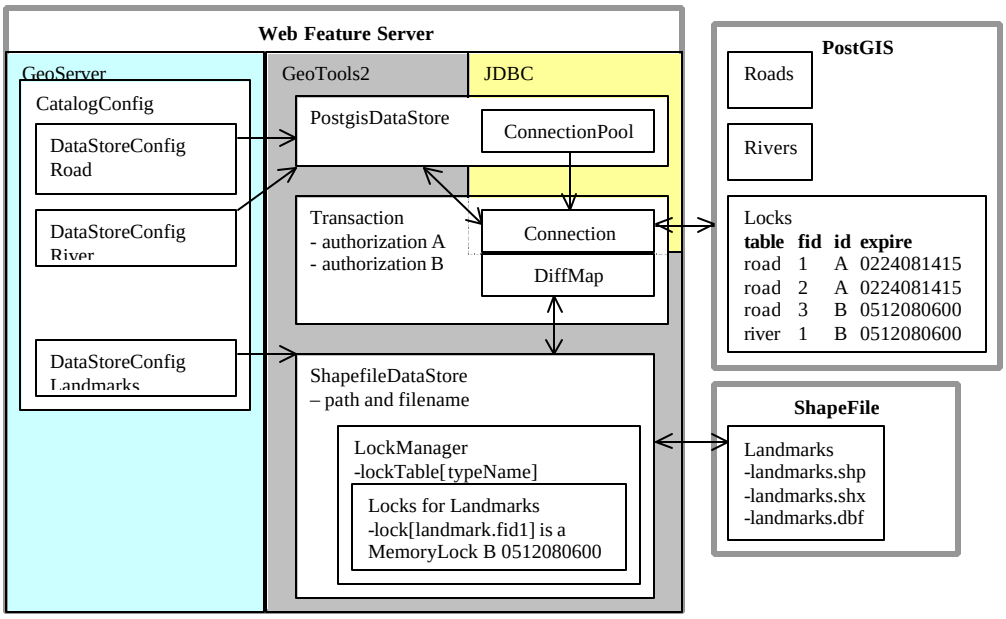

Figure 4: GeoServer WMS Branch and New GeoTools DataStore

This design has several benefits for GeoServer:

- Transaction allows shared use database connection
- Strong Transaction Support is provided
- Transaction and Locking are provided for file DataStore
- Scalability is not Limited by Memory

3 PERFORMANCE TESTING

Geotools2 currently has new DataStore implementations for the following:

DataStore/Contact	Optimizations
PostgisDataStore Chris Holmes Jody Garnett	- count(Filter.NONE) - addFeatures() – direct SQL - modifyFeatures() – direct SQL - removeFeatures() – direct SQL
OracleDataStore Sean Geoghegan	- count(Filter.NONE)
ShapeFileDataStore Ian Schneider	no optimizations
ArcSDEDataStore Gabriel Roldán	no optimizations

These implementations are currently in active development. This document will focus on the PostgisDataStore as it represents a stable implementation with limited optimization.

3.1 Testing Process

We will perform the following tests:

- Read All contents
- Calculate Bounding Box
- Read a Single Feature
- Lock a Single Feature
- Remove a Single Feature
- Lock all Features
- Attempt to remove a Feature without permission

These tests have been performed with different volumes of Data to assess scalability and performance.

These tests were performed with the current branch of GeoServer as it has been switched over to the new DataStore API. A public release is forthcoming.

For comparison, the GeoServer 1.0 release based on the DataSource API has been used.

3.2 Performance Results

We have found that with the limited optimizations present in the current implementations performance is bound by the use of FeatureReader.

The following graph indicates the time in seconds to read a dataset vs. the size of the dataset.

Results are presented for:

- The new GeoTools DataStore: FeatureReader access
- The original DataSource: FeatureCollection Itterator access

Size is measured as the number of features in the dataset times the number of attributes for each feature.

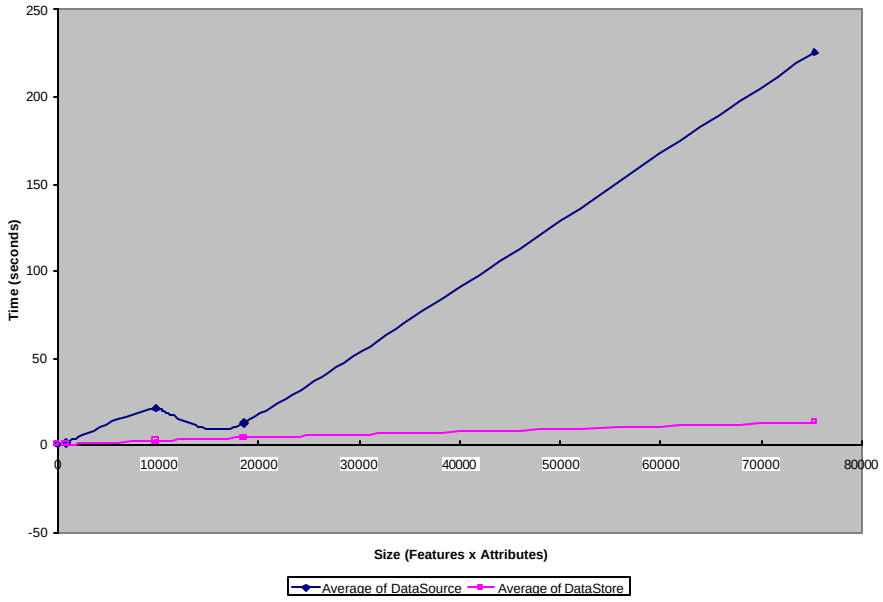

Figure 5: DataStore / DataSource Performance Comparison

We should note that:

- DataStore is significantly faster
- DataStore has resolved the our Scalability Concerns
- DataStore performance is Linear

The DataStore API has been designed to allow for optimization. We expect the PostGIS DataStore to be optimized as the implementation matures and looking forward to even better performance.

3.2.1 PostGIS DataStore FeatureReader Performance

These results are expressed in terms of:

- API – DataSource and DataStore are compared
- Size – represents the total number of attributes.
  (number of Features) multiplied by (number of Attributes in each Feature)
- N – the number of features

API \ Size	14	819	9800	18567	75335	1826340	N
Old DataSource	0.7018						7
New DataStore	0.5736						7
Old DataSource			21.123				49
New DataStore			2.5966				49
Old DataSource		1.351					117
New DataStore		0.519					117
Old DataSource				12.9546			2063
New DataStore				4.485			2063
Old DataSource					225.6575		15067
New DataStore					13.1505		15067
Old DataSource						out of memory	91317
New DataStore						137.895	91317

These tests show a remarkable speed improvement over Data Source when working with the DataStore API. They also illustrate the ability of the DataStore API to smoothly scale to large data sets.

4 PERFORMANCE CONSIDERATIONS

To ease adoption of the DataStore API, an extremely capable AbstractDataStore has been provided to implementers.

The goal of AbstractDataStore is to allow for full functionality; optimizations for speed must be left to specific implementations.

DataStore Performance Considerations:

Class	Purpose
FeatureReader	Sequential access
FeatureWriter	Sequential modification, and append
FeatureSource	Spatial Query by Type
FeatureResults	Results of a Spatial Query
FeatureLocking	Locking

This section introduces the call stack as maintained by a DataStore when providing content. Each layer of this call stack is managed by a “wrapper” class that builds functionality on the layer preceding it.

By taking advantage of native functionality, these layers may be replaced to improve performance or take advantage of native transaction or locking support.

For Example – optimizations for the new PostGIS DataStore:

- FeatureWriter:
  Using a FeatureWriter to add new content to a DataStore may be implemented as O(1), rather than O(N) by through the use of SQL INSERT
- FeatureSource:
  The getFeatures(Filter) method may be optimized to O(lgN), as opposed to O(N), by the use of SQL SELECT in conjunction with PRIMARY KEY.
  Further optimization is possible through the use of GEO Spatial Indexes
- FeatureStore:
  May add, remove and modify requests can be reduced to simple SQL statements that do not require a FeatureWriter for post processing.
- FeatureResults:
  May cache the results of getCount(), and getBounds() to prevent recalculation
- FeatureResults:
  May cache the ResultSet produced by a SQL query for increased performance at the cost of memory use.

When implementing caching care must be taken maintain cache contents in the presents of modification. DataStore provides a FeatureEvent notification service allowing implementations to handle cache updates.

4.1 Baseline

The AbstractDataStore read access is built on the concept of a FeatureReader.

We will use a simple example of a DataStore built on top of an array to illustrate performance considerations.

Example of a Simple read-only DataStore:

```
final FeatureType featureType = featureArray[0].getFeatureType();
return new AbstractDataStore(){
  public String[] getTypeNames() {
    return new String[]{ featureType.getTypeName() };
  }
  public FeatureType getSchema(String typeName) throws IOException {
    if( typeName != null && typeName.equals( featureType.getTypeName() ) ){
      return featureType;
    }
    throw new IOException( typeName + " not available");
  }
  protected FeatureReader getFeatureReader(String typeName)
    throws IOException
  {
    return DataUtilities.reader( featureArray );
  }
};
```

4.1.1 getFeatureReader(schema, filter, transaction)

The use of a FeatureReader to iterate or stream over the contents of the DataStore is by definition of O(N).

FeatureReader methods:

- next() – read the next Feature
- hasNext() – check for more Features

Overhead for FeatureReader.next():

FeatureReader	Performance Overhead on next()
DataUtilities.reader	O(1) – base reader supplied by subclass
DiffFeatureReader	O(1) – memory bound
FilteringFeatureReader	O(1) – filter overhead
ReTypeFeatureReader	O(1) – Change Schema

Overhead for FeatureReader.hasNext():

FeatureReader	Performance Overhead on hasNext()
DataUtilities.reader	O(1) – base reader supplied by subclass
DiffFeatureReader	O(1) – memory bound
FilteringFeatureReader	O(N) – needs to search for the next match
ReTypeFeatureReader	O(0) – Change Schema

4.1.2 FeatureSource

The FeatureSource Interface provides a high-level API, specifically for optimization. AbstractDataStore uses DefaultFeatureLocking which is not optimized.

Performance of DefaultFeatureStore:

Method	Performance Considerations
GetCount(Query)	O(1) – no optimization implemented
GetBounds(Query)	O(1) – no optimization implemented
GetFeatures(Query)	See FeatureResults

4.1.3 FeatureResults

The FeatureResults Interface provides access to the results of a query. The FeatureResults class is allows implementations a chance to cache the results of a query, or may make use of optimizations.

DefaultFeatureStore uses DefaultFeatureResults to implement its getFeatures(Query) operation.

Performance of DefaultFeatureResults:

Method	Performance Considerations
reader()	O(N) – uses getFeatureReader
getCount()	O(N) – uses reader
GetBounds()	O(N) – uses reader

4.2 FeatureWriter Performance Considerations

The AbstractDataStore write access is built on the concept of a FeatureWriter. Similar in design to FeatureReader, the expected performance of FeatureWriter is O(N).

FeatureWriter methods:

- next() – read the next Feature
- remove() – remove the last feature returned by next()
- write() – write any modifications to the last feature returned by next()
- hasNext() – check for more features

4.2.1 getFeatureWriter(typeName, filter, transaction):

Overhead for FeatureWriter.next():

FeatureWriter	Performance Overhead on next()
(base writer)	O(1)
DiffFeatureWriter	O(1) – memory bound
CheckedFeatureWriter	O(1) – memory bound
FilteringFeatureWriter	O(1)

Overhead for FeatureWriter.remove():

FeatureWriter	Performance Overhead on remove()
(base writer)	O(1)
DiffFeatureWriter	O(1)
CheckedFeatureWriter	O(1)
FilteringFeatureWriter	O(1)

Overhead for FeatureWriter.write():

FeatureWriter	Performance Overhead on write()
(base writer)	O(1)
DiffFeatureWriter	O(1) – memory bound
CheckedFeatureWriter	O(1) – memory bound
FilteringFeatureWriter	O(1)

Overhead for FeatureWriter.hasNext():

FeatureWriter	Performance Overhead on hasNext()
(base writer)	O(1)
DiffFeatureWriter	O(1) – memory bound
CheckedFeatureWriter	O(1) – memory bound
FilteringFeatureWriter	O(N) – needs to search ahead for the next match

4.2.2 getFeatureWriter(typeName, transaction)

This method returns a FeatureWriter that allows the addition of new content. Expected FeatureWriter performance is in accordance with the previous section

4.2.3 getFeatureWriterAppend(typeName, transaction)

This method returns a FeatureWriter that only supports the addition of new content. The very first write is O(N), subsequence write operations are O(1).

4.2.4 FeatureStore

The FeatureStore Interface extends the high-level Query API provided by FeatureSource with the addition of writing operations. AbstractDataStore uses DefaultFeatureLocking which is not optimized.

Performance of DefaultFeatureStore: (N is size of DataStore, M is size of change)

Method	Performance Considerations
AddFeatures	O(N) + O(M) – depends on getFeatureWriterAppend
RemoveFeatures	O(N)
ModifyFeatures	O(N)
SetFeatures	O(N) + O(M)

4.2.5 FeatureLocking

The FeatureLocking Interface provides additional locking operations to FeatureStore. AbstractDataStore uses DefaultFeatureLocking, which is not optimized.

DefaultFeatureLocking makes use of an InProcessLockingManager that maintains Feature Lock information in memory.

Performance of DefaultFeatureLocking:

Method	Performance Considerations
LockFeatures(Query)	O(N) + O(M)
LockFeatures(Filter)	O(N) + O(M)
LockFeatures()	O(N)
unLockFeatures(Query)	O(N)
unLockFeatures(Filter)	O(M)
unLockFeatures()	O(N)

5 DESIGN RECOMMENDATIONS

There are a limited number of design recommendations at this time.

5.1 Replace Use of FeatureType in Requests

Currently the high-level FeatureSource interface makes use of a Query object to represent a request for information.

The DataStore API makes use Schema type and Filter to issue a query. It has been requested that the DataStore API make use of the Query object already created for this purpose.

This is a mistake, and renders the class difficult to use. This problem is compounded by Schema information being immutable.

Planned Change:

```
old: DataStore.getFeatureReader( FeatureType, Filter, Tranasaction )
new: DataStore.getFeatureReader( Query, Transaction )
```

This change has been requested, with good reason, and is already underway.

5.2 Limit Change Notification

The new DataStore API contains an event notification system for content changes. Event notification allows user interfaces and caches to stay informed of DataStore changes as they occur.

User interface designers have requested that changes be sent less often, for fear of poorly designed applications flooding them with redraw events.

No changes are planned to the design, although a delayed event listener may be written to address these concerns.

5.3 Catalog Support

Currently GeoServer is maintaining a lot of metadata in order to operate. It would be nice to fold some of this information into the geotools2 library.

- Track OFC Catalog Specification
- Provide Cross DataStore Lock Management
- Generate XML Catalog Schema information

A Catalog class has been added with this intention.