SharpMedia Database Design

© 2007 Žiga Osolin – SharpMedia project, document version: DRAFT

Contents

[About 2](#_Toc167024843)

[Goals 2](#_Toc167024844)

[Deployment Structure 2](#_Toc167024845)

[Namespace Structure 3](#_Toc167024846)

[*Namespace* Database 3](#_Toc167024847)

[*Interface* IDatabase 3](#_Toc167024848)

[*Interface* IEvent 3](#_Toc167024849)

[*Interface* INode 3](#_Toc167024850)

[*Interface* ITypedStream 6](#_Toc167024851)

[Managed implementation 9](#_Toc167024852)

[Helper Classes 10](#_Toc167024853)

[*Namespace* Database.Memory 11](#_Toc167024854)

[*Namespace* Database.Reference 11](#_Toc167024855)

[*Class* Query 12](#_Toc167024856)

[*Namespace* Database.Transactions 13](#_Toc167024857)

[Usage Cases 14](#_Toc167024858)

[Linking nodes 14](#_Toc167024859)

[Using Helpers 15](#_Toc167024860)

[Queries 16](#_Toc167024861)

# About

SharpMedia Database is object storage abstraction interface, responsible for storing serializable objects in typed manner.

# Goals

Database system has unique goals not utialized by other database. The database should actually represent a filesystem and database, all in one. It should allow fast and safe read and write operations. One of the design strategies is that we are writing objects, not byte data. This allows certain optimizations and better organisation.

The goal is to provide very dynamic and extandable database. It should allow capability based security information to be applied to it, multiple objects (of different types) to be stored at one node and various extensions.

# Deployment Structure

Database system is very common and is deployed as a part of SharpMedia core library as other common systems. The library can be System aware (system can provide security for database) but System is not required in order to use database interfaces.

# Namespace Structure

Namespace structure consists of basic abstraction that resides in Database namespace and a memory implementation of database.

## *Namespace* Database

Database namespace contains basic abstractions; this includes interfaces that allow access and manipulation of database.

### *Interface* IDatabase

The database only allows high level information about the system (capacity and free space) and allows all further functionality through returned ***INode*** root. Free space and device storage return approximate number of bytes. This values cannot be used to compute the actual number of objects we can place in database.

### *Interface* INode

This interface allows all sorts of node manipulation. Each node can have any number of named children, each child's name must be unique (it is a string). Besides that, each node has at least one typed stream attached, one is default. The actual data is stored in typed streams. Each typed stream can contain any number of objects[[1]](#footnote-2). Besides all enumerated, the node also supports versioning, e.g. can track previous versions. New version is created only on demand, explicitally. First version is always index 0. A version contains it's own typed streams and inheritance but children are shared between all versions. Nodes also allow inheritance[[2]](#footnote-3). If node inherits other node, this means that is containes all typed streams of base node. A node is an essentially heirarchical, versioned element in database.

The node implementation must implement the interface with those members:

1. This value can be 0 if you cannot support it. It is only an approximation.
2. New version is previous plus 1. The bool argument specifies whether a copy must be created or
3. Implementor of database driver must implement only one level (own children) search, managed node takes care of deeper searches.
4. This information is not required, can return 0.
5. Physical location is informative and not required to be correct.
6. Children and typed streams are returned only for current node, not for it's base. Use helpers for that.
7. The base node, linking to most current version. It is not driver implemented. It can be null (and usually is null). If base node is non-null, the following rules apply:
   1. Base's default typed stream does not have an effect on the inherited node;
   2. If base and current node have the stream of the same type, you can manipulate only current node's typed streams;
   3. Bases do not support versioning; all versions of node will essentially link to latest version of base node;
   4. Adding typed stream always adds stream to node and not it's base;
   5. Removing typed streams from base is not allowed and throws an exception. If you want to remove typed streams from base, use direct base manipulation;
   6. Children of nodes are also shared, inherited node overwrites base's children;
   7. Through node you can obtain any kind of access to base nodes, as long as security on base node permits that.
8. Base with version is the same as normal base, only that we bind base to current version, meaning that creating a new version of base node will not change the link to new version.

All operations are restrictive. Following operations throw exceptions:

* Opening non-existent stream;
* Adding existent stream;
* Removing non-existent stream;
* Changing default stream to non-existent or null stream;
* Deleting non-existent child;
* Creating child with the same name or invalid name;
* Removing a typed stream from base node.

Following operations do not throw an exception, return null or 0:

* Finding a node;
* Checking byteoverhead if not support returns 0;
* Checking time stamps returns invalid times (null time) if not supported;
* Obtaining non-existant version.

The scheme represents the node-child-typed stream hierarchy. The versioning is not included.

#### Driver implementation

Drivers should implement only subset of functionality. The subset is contained in ***IDriverNode*** interface. This class includes only methods that must be implemented and some signatures of methods are different. This means that you pass types as names, not as classes. This ensures that we don't have to load assemblies into system application domain.

Managed node also uses driver implementation as a second interface implementation, to support system wrapper.

### *Interface* ITypedStream

Typed stream is interface that enables access to actual data. Most of typed streams will contain small number of elements, usually 0 or 1. Implementations should use this information to optimize small number of elements scenarios. At the same time, typed streams may be filled with thousand of small (less than few hundred bytes) objects. Too much memory must not be wasted to store so many elements and seach times (indexing) must allow fast read access.

Drivers should implement single-threaded data access. Managed implementation however permits multiple read and one write stream scenarios through locking (access is never simultanious).

1. Flags are (more or less) stable hints and specifiers of stream. Implementation may use them. The following flags are supported:
   1. Compression hint (can be ignored);
   2. Maximum one object (must not allow more objects to be inserted, object can only reside at index 0);
   3. Allow derived types allows other object types to reside in database if their derive from the type of typed stream;
   4. Interleaved tries to make writing streams interleaved, the order of writes will affect the order of reads.
2. The object type is the same as the type of the stream. It can be different only if derived types are allow at typed stream. This is better than reading the whole (potencially big object) and checking it's type.
3. Sequential read/write operations are special because drivers can optimize them.
4. Erases object from offset and count. The last flag specifies if place is left empty or object indexes following offset are substracted by number of object that were to be erased.
5. Number of residing elements that are alive, not the maximum index number.

It is important that typed stream implements good indexing techniques for fast (random) object search. The database is expected to work with smaller objects. We can (almost) always split data into smaller logical components; for example video can be split into stream of frames and stream of sound tracks, document can be split on images, paragraphs … This type of storage is efficient because we can access only part of actual data (paragraphs only, video only (no sound), images only).

#### Driver implementation

Because drivers reside in different application domain, we do not have access to types. The ***IDriverTypedStream*** is a special interface that exposes only required subset of functionality in non-object form (types as strings, objects as byte[] blocks and type information as strings).

Driver implementation is also used as second interface for managed typed stream.

### Managed implementation

Because we want only one common implementation, we added a managed layer over actual database. This includes managed database implementation (**DatabaseManager : IDatabase**), managed node implementation (**ManagedNode : INode, IProxyNode**) and typed stream (**ManagedTypedStream : ITypedStream, IProxyTypedStream**).

Additional security layer is implemented by System to ensure safety in some configurations of SharpMedia. This is described in System design document.

Managed implementation takes care of thread safety, garanties that maximum one **INode** and **ITypedStream** will exist for specific location, resolves complex paths (with '/', '.' …) and provides all checks (creating child with the same name as one that already exists, trying to access previous version that does not exist etc.). This means that driver do not need to do any of those because they are **always** accessed through managed nodes.

#### Direct Managed Access

Direct managed access to node can be made only inside System application domain. This is most useful in game runtime in SharpMedia is configured to not use processes. The access is possible through **INode** and is the same in any scenario.

#### Access Via System

Accessing via System can be from any SharpMedia process. The node and typed stream behave as **INode** and **ITypedStream** states. System nodes are actually only lightweight classes that point to **ManagedNode**s. The system typed stream is also a lightweight class that points to **ManagedTypedStream**s.

Mixed access (direct and via system) is permitted.

#### Managed Node Cleanup

There is **always** only one managed node per **driver node**. We keep the hierarchy (cached) in memory. Only back (parent) links are hard, children links are weak references. This allows GC to cleanup the unused managed nodes when it wants (or when we request it). Because both **ManagedNode** and **IDriverNode** are relativelly small in size, cleanups are not very frequent.

Managed nodes share the same version node and it is cleaned when all managed nodes are cleaned. Version has only weak backlinks on managed nodes.

### Helper Classes

Helper classes reside in **Database** namespace and help you manipulate typed streams and nodes.

#### TypedStream<T>

**TypedStream<T>** is a template class that allows type-safe access to objects in certain typed stream. It can be constructed either using a node (where only one stream of T exists) or using a **ITypedStream** reference. It provides all functionality offered by typed streams and a lot more:

* Indexers for object get and set access;
* Helper methods for manipulation of single object typed streams;
* Implements **ITypedStream**, so such access is also possible.

#### TypedStreamHelper

TypedStreamHelper contains *static* methods that help you manipulating **ITypedStream** object. It is heavily used by TypedStream. Some methods are template methods and they allow automatic template resolving.

#### Node

A node helper contains *static* as well as *dynamic* methods for node manipulation. A node also allows various seraches, also those returning multiple nodes. Searches can be based on regex expression or wildcards. Valid wildcards search expressions are any expressions that consist of characters and additional '\*'s. '\*' can represent any name, but does not include '/' character. *We suggest using wildcards because they are more limited and result in faster searches.*

Node can also search for typed streams or several versions. A generic search expression is like this:

»Node/Path/To/Sth@Version#TypedStream«

If no typed stream is specified, default is used. If No version is specified, current is used. Wildcards or regular expressions can be used only in node part. In Version part, additional Version+ can be used and for TypedStream, additional »:« can be used (meaning that it derives from this type).

You can also find an object (or objects) directly by applying a »[index]« operator.

## *Namespace* Database.Memory

Namespace contains in-memory implementation of database. Database serializes objects to RAM[[3]](#footnote-4) buffers and then deserializes them from them. Because the implementation is very short, it is distributed together with this library.

The in-memory database is useful for holding non-persistant data such as process communication data (I/O ports can actually be opened) and other stuff that must reside in RAM.

## *Namespace* Database.Reference

Namespace contains reference implementation of database. Reference database allows holding objects as references. Serialization is actually never performed, only objects are served, if necessary also as marshalled by reference.

It is doubtful if such database can exist for non-system processes and also if it will ever exist.

### *Class* Query

Moved to OS.Database!

Since database holds data in organised, typed way, we can submit queries on such database. Query is a process of searching for nodes/types streams that have all properties requested by queries. The different between finds and queries is that query actually inspects objects in typed stream while finder only follows the pattern.

#### Relationships

#### Matchers

## *Namespace* Database.Transactions

Moved to OS.Database!

Transactions are operations that either fully succeed or do not alter the state of database, they are atomic operations. If two objects need to be both written to database, we cannot garantie that the system will succesfully write the second when the first one is already written[[4]](#footnote-5).

This is useful, when a number of objects written to the database have references to one another, and thus we could write an object that would either not be referenced (resulting in garbage) or one that references an object that has not yet been written the the database.

Transactions can work upon any managed or system entities.

### *Class* Transaction

The most important class is the **Transaction** class. It can be constructed without arguments or with the specified temporary storage (otherwise, storage is specified by system through it's policies). Transaction API is based on **ITypedStream** and **INode** interfaces. This means that you can:

* Add a node and a proxy node is returned, tracking all changes;
* Add a typed stream and a proxy typed stream is returned, tracking all changes;
* Use node proxy to add new nodes to transaction (by using Find on children or anything like it);
* Use proxy node to add new typed streams or edit existing.

It should be ensured that most of the operations are validated at execution time and not commit time. Reading newly created typed streams or newly inserted objects should also be available. The transaction should also be thread safe.

**Note :When creating a proxy node, the actual node is locked and cannot be accessed (so accessing it from same thread will result in dead-lock or better, an exception thrown).**

**Note: Transaction and proxies stay alive (e.g. also lock internal nodes) until transaction is disposed. Try using locks to ensure proper disposing.**

#### Comitting

Commiting is a process of executing all operations previously specified on proxies. When all operations are *atomically* executed, the operations are cleared and new transaction can continue. If commiting fails somewhere in between the process, the system will automatically either fully complete (if possible) the transaction or it will be fully reverted.

#### Reverting

Reverting is a process of discarding all changes made until previous commit or transaction construction. Reverting is simple operation and does not require special synhronizations.

### Custom Operations

Custom operations can be employed as transaction. Such operations are not specified through proxies but directly on transaction, using **Transaction.AddCustom(Operation).** Some custom operations are also a part of transaction library. This includes:

* ObjectCopy (ITypedStream, uint -> ITypedStream, uint);
* TypedStreamCopy (ITypedStream -> ITypedStream);

Custom operations are useful because they allow less data to be stored to special transaction streams. For example, object copy would require copying an object to this special storage and then copying it to actual typed stream. This is an unnecessary overhead which can be overcome using custom operations.

# Usage Cases

## Linking nodes

A node linking is represented here:

public void LinkNodes(DatabaseManager manager)

{

INode node = manager.Find("/Some/Path/To/Node");

INode linkTo = manager.Find("/Some/Path/To/Link");

// We can link to node.

node.Base = linkTo;

// We can link to specific version.

node.BaseWithVersion = linkTo;

// Or we can even link to version 2 (if not exist, we link to null).

node.BaseWithVersion = linkTo[2];

// Once nodes are linked, we can access link through node.

using (ITypedStream s = node.GetTypedStream(OpenMode.Read, typeof(string)))

{

// the node can exist in node or linkTo and it will be valid.

// Do sth. with it.

}

// We could also use the helper trick, automatically resolving to string typed stream.

using (TypedStream<string> ts = new TypedStream<string>(node))

{

// Use it.

}

}

## Using Helpers

Helpers can be very useful because they can make code more readable and they implement a lot of useful methods and properties:

public void UsingHelpers(DatabaseManager manager)

{

INode node = manager.Find("/Some/Path/To/Node");

// Can access all objects.

ICollection<string> allObjects = TypedStreamHelper.GetAllObjects<string>(node);

foreach (string x in allObjects)

{

// Do some work.

}

// We can list all typed stream, including those in bases.

// Node implements static methods for INode.

List<Type> typedStreams = Node.AllTypedStreams(node);

foreach (Type t in typedStreams)

{

using (ITypedStream s = node.GetTypedStream(OpenMode.Read, t))

{

// Do some work ...

}

}

// We can also access well typed streams using template helper.

using (TypedStream<object> helper = new TypedStream<object>(node))

{

// Do some work with it, more functionality than normal ITypedStream.

}

// And much more ...

}

## Queries

Moved to OS.Database!

Queries can be useful to query for non-simple relations. This usage case includes finder features (that are also a part of queries) and relations that can be defined by queries.

## Transactions

Moved to OS.Database!

Transactions can make your life simple by making operations atomic.

public void Commit()

{

DatabaseManager manager = new DatabaseManager();

manager.Mount("/Root", Memory.MemoryDatabase.CreateNewDatabase("MyDB"));

INode root = manager.Find("Root");

// Now we want to create 3 nodes atomically.

using (Transaction transaction = new Transaction())

{

// A proxy is returned.

root = transaction.AddNode(root);

root.CreateChild("Child1", typeof(string), StreamOptions.Interleaved);

root.CreateChild("Child2", typeof(object), StreamOptions.Interleaved);

INode last = root.CreateChild("Child3", typeof(string), StreamOptions.Interleaved);

// We can also use last which must be a proxy.

last.Base = root;

last.AddTypedStream(typeof(object), StreamOptions.Interleaved);

// We can now commit it, all proxies still valid.

transaction.Commit();

// We can issue more transactions.

root.Base = root.Find("Child1");

// We are not happy with it.

transaction.Revert();

// Disposing makes all proxies invalid and unlocks nodes.

}

}

1. There are also special one object typed streams. This allows special optimizations on some implementations. [↑](#footnote-ref-2)
2. Inheritance is implemented by managed streams and does not require driver awareness. [↑](#footnote-ref-3)
3. Random Access Memory [↑](#footnote-ref-4)
4. Most database drivers garantie that one object is either written or not, e.g. the atomicy of write operation. This is needed to make the database stable (no loss of information or currupt data). [↑](#footnote-ref-5)