N-Storage

Developer manual

Email: lrndlrnd@mail.ru
URL: www.Sergey-L-Gladkiy.narod.ru

Contents

Introduction

N-Storage library provides saving and reading (serialization and deserialization) of object data for .NET programs. Data saved in Text format as object structures those are easily readable for humans. The library developed for using with minimal efforts for code writing and with maximum serialization possibilities. The storage algorithms provide automatically supported version compatibility (forward and backward) when data structure changed (data removed or added). The system provides error diagnostics with pointing error data elements and reasons of the errors.

ADVANTAGES of N-Storage library:

1. 100% C# source code.
2. Simple text file format, human readability.
3. Simple in use with minimal code writing.
4. Automatically supports the most .NET types (including generic types).
5. Forward and backward version compatibility for object data changes.
6. Error diagnostics when sawing and reading data.
7. Reading data with noncritical errors.
8. Any type automatically serialized (without Serializable attribute).
9. No problem with Events.
10. No problem with mutually dependent objects.
11. Streams and memory-mapped files serialization.

Serialization concepts

Serialization is a process of storing objects' state (program data) with converting them into a data stream with the possibility of total reconstruction of initial state (deserialization).

Main concepts of data serialization, used in N-Storage system, aim to such realization of storage system that provides the simple use and automatically supported forward and backward version compatibility with data changes.

- ***Serialization must store ALL data as is.*** As serialization must store the total state of the object, all object's data must be stored for guaranteed object reconstruction. Data must not be marked with Serializable attribute to be stored.

- ***Deserialization must reconstruct data but not execute methods.*** Together with data objects also contains 'properties'. Properties are not data, they are interfaces for accessing data. The access interface can hide complicated logic execution under the setting data values. In many cases this execution cannot be done before the total state of the object reconstructed. So, serialization MUST NOT be based on properties, it must be based on data fields only.

- ***Objects may be reconstructed partially.*** During deserialization process all data that can be reconstructed must be reconstructed. Corruption of some part of data must not lead the reconstruction of another part being impossible (if file format is not corrupted).

- ***Automatically support of forward and backward version compatibility***. With internal program data representation changing the forward and backward compatibility of stored data must be AUTOMATICALLY implemented in storage system if it is can be done without special knowledge. For an example, if in new program version some data added it have not being marked with special attribute. The storage system has enough information to handle this satiation. If newer program reads data stored with older version, the absence of some part of data must not cause error generation. Similarly, when older program reads data written with newer version, the presence of superfluous data mast not cause problems – this data just not needed for the program. Developer must not mark every new field with special attribute or write special code – storage system is responsible for this.

1. File format

NET Object Data Format (NODF) is intended for storing data conforming to the NET Framework types, including fundamental types, arrays, generics, enumerations, structures and classes.

File format

NODF file is a text file containing strings in Unicode character encoding. Culture dependent data string representation (floating numbers, dates) is formatted as Invariant Culture.

File structure

The NODF file divided into separate blocks. Each block has the following format

[**BLOCK**:block_name]
Block Data
[**BLOCKEND**:block_name]

BLOCK, **BLOCKEND** – keywords, block_name – name of the block, Block data – data, specific for the block.

NODF file contains the following three obligatory blocks:

1. FORMAT – information about format and data version;
2. INFORMATION – information about contained data (number of objects, types and so on);
3. OBJECTS – stored object data.

In addition there can be nonobligatory TYPES block following after INFORMATION one and containing the dictionary of data types.

FORMAT and INFORMATION blocks

FORMAT and INFORMATION blocks contains data written in simples format – sequence of the following records

NAME=VALUE

where NAME is the stored property name and VALUE is its string value representation (conforming Invariant Culture). In these blocks only one line strings allowed for each property (multiline records not allowed). The number of properties can differ for various versions. Only 'public' properties stored in these blocks, storage implemented with simple serialization algorithm.

TYPES block

The TYPES block contains records of the following format:

TYPE~ID64=TYPENAME

where **TYPE** – keyword;
ID64 – 64-bit unsigned integer number (number of type which is the unique identifier);
TYPENAME – type description.

Type description (TYPENAME) has the following syntax:

Name<Parameter1:Parameter2:…:ParameterN>

Name – type name;
ParameterX – type parameters depending on its specific properties (present not for all types).

For generic types all parameters are also type descriptions and their format is the same as TYPENAME.
For array names the following scheme used: array type name consists of the name of base type (type of array elements) followed by brackets '[]'. Regular and irregular arrays supported. Brackets format is the same as for C# arrays declaration (see appendix).

OBJECTS block

Objective data block contains the sequence of object records. Each object record is data for one object and has the following format:

OBJECT~ID64:TYPE
{
Object data
}

OBJECT – keyword;
ID64 – 64-bit unsigned integer number (number of object which is the unique identifier);

TYPE – object's type.
Object Data – object data.

Object's type can be represented with two ways. If the is TYPES block in the file, the object's type is the reference to one of the dictionary entities. And then its format is:

TYPE~ID64

where **TYPE** – keyword;
ID64 – 64-bit unsigned integer – type's identifier (key) in the dictionary.

If there is not TYPES block in the file, then object's type is the full type description TYPENAME.

Object data is a sequence of the member records (excluding objects of string and value types (see later). Each member record has the following format:

```
MEMBERID Name:TYPE
/
        Member Value
\
```

MEMBERID – identifier of object's element data and can be one of two: **DATA** (data element), **PROPERTY** (property element);
Name – name of the element;
TYPE – type of the element (the same as for object record);
Member value – value of the member.

Member value has the following syntax:

FORMATID<PARAMETERS>=value

FORMATID – identifier of the member format;
PARAMETERS – data parameters, not obligatory and depend on the member format;
value – the member value (can be multiline).

The syntax of data parameters: PARAMETERS are the sequence of values separated with ':', number and sense of values depend on data format. For binary data (streams) they are 2 64-bit integers – size of stream in bytes and the current stream position. For array data parameters are its sizes. Regular and irregular arrays supported. For regular arrays the number of parameters equals its dimension and each value if the size of one dimension. For irregular arrays each parameter represents the inner dimensions for each element (see appendix).

Format identifier can be one of the following:

- **VALUE** – data value is string representation of member value of type TYPE (for Invariant Culture). This format is used for all primitive value types (int, double and so on, excluding struct types), enumerated types and other.
- **REF** (reference) – data value is a 64-bit unsigned integer – reference to the object's number in the file's object list, also it can be **NULL**. This format is used for all reference types (class), struct types and string values.
- **LIST** – data value is the list of simple values separated with spaces. Each simple value is the string representation of one primitive type value (for Invariant Culture). This format is used for arrays of primitive type values.
- **REFLIST** – data value if the list of 64-bit unsigned integers separated with spaces (identifiers of objects in the file's object list) or **NULL** values. Used for storing arrays of reference types.
- **BINARY** – data value is the stream of bytes. Used for storing streams.
- **EXTERNAL** – data value is a path for the external file, containing the data for this member.

Object Data for String values and objects of primitive types (value type excludinge struct) have the following format:

VALUE<PARAMETERS>= data value

PARAMETERS – data parameters, not obligatory and depend on the type;
data value – string representation of the value.
For type 'String' the only parameter is the length of string, data value begins from the next line.
For primitive type data value is string representation of value (for Invariant Culture).

2.N-Storage realization

Realization.

1. File format corresponds to the description of NODF.
2. Only **DATA** members(fields) supported.
3. Value formats realized: **VALUE**, **REF**, **LIST**, **REFLIST**, **BINARY**.

4. Supported data types: classes, structures, generics, value types, strings, nullables, arrays (regular and irregular), interfaces, streamsпотоки, memory-mapped files.

Constraints.
1. Mixed arrays **NOT** supported (irregular with inner regular).
2. Arrays of streams **NOT** supported.
3. Streams with 'read-only' position NOT supported.
4. Virtual fields **NOT** supported.

Notes.
1. Auto-properties (properties with implicit getter and setter) force creation of special fields. These fields stored by property's name, not by field's name.
2. String representation of floating point values can lead to the precision lost due to the forward and backward conversion. So, the initial and reconstructed values can be NOT equal after deserialization.

3. Using N-Storage

Saving and reading data

The main class for serializing and deserializing data is FileSerializer. This class has two constructors:
For serialization:

```
FileSerializer(object data, FormatAttributes fa, FileStoreOptions opt, string fileName)
```

For deserialization:

```
FileSerializer(FormatAttributes fa, string fileName)
```

object data – root object for serialization;
FormatAttributes fa – information about data format and version;
FileStoreOptions opt – options for storing data;
string filename – full file path.

Parameter 'FileStoreOptions opt' allows set up some options for data storage process. For an example, one option turns on or off using the TYPES block while storing data.

Class FormatAttributes contains a set of public properties which stored in FORMAT block. A descendant of the class can be used and it can contain additional public properties which automatically stored during serialization.

Serialization.
Serialization is implemented in 2 steps (calling 2 methods):
1. Calling method
   bool PrepareSerialization()
   This method creates internal data representation structure. It returns true if the structure created.
2. Calling method
   bool Serialize()
   This method stores data into the file. Returns true if data stored.

Deserialization.
Deserialization is implemented in 2 steps (calling 2 methods):
1. Calling method
   bool PrepareDeserialization()
   This method opens the file and reads header data – format and version attributes and information (number of stored objects, types and so on). Returns true if header data read successfully.
2. Calling method
   bool Deserialize()
   This method reads objective data from the file into internal representation structure and reconstructs original objects. Returns true if even one object read and there is no critical error (when file structure error fixed).

During deserialization process there can be noncritical errors, when the deserialization process can be continued. The errors are: superfluous data presented, errors of some objects creation and so on. After deserialization the information about these errors can be requested with calling method

```
List<BaseObjectDataItem> GetErrorInformation()
```

The method returns list of object those have been read with errors.

Serializing special data

Constrains for the serialization data caused by the current program realization.

For deserialization all stored classes must implement 'public' default (parameterless) constructor. If some class has no such constructor, the instances of the class are not created and marked as error objects in the data structure.

NOTE: Why 'public' constructor required. If some class has no public constructor, this is probably caused some reasons and the class developer does not intend it to be created with external objects by common method. It is probably intended for special creation process. Common example of such objects is 'singleton'.

If some data field is not needed to be stored, it can be marked with NonSerialized attribute.

If there is no possibility to provide default public constructor for some class (for an example, this class is from an external library), then a generating object of Creator class can be registered for solving this problem. This class has the following constructor

```
Creator(Type type, Create func)
```

Type type – type of created instances;
Create func – delegate for providing instantiation of objects.
The generating object must be registered by calling *SetCreator* method of the Instantiator class.

If some class cannot be serialized itself by some reasons, the storing class can be registered for solving the problem. Storing class must be a descendant of base class Storer. This class used in general case of substituting one object data instead of another. For realizing storing class the descendant of Storer class must be created and 2 abstract methods overridden:

```
void FromObject(object wrapped)
void ToObject(ref object wrapped)
```

the former "copies" data from original object to storing one (called before serialization), the later "copies" data from storing object to original one (after deserialization).
The storing class must implement 2 constructors:

```
Storer(object stored)
Storer()
```

for serialization and deserialization accordingly.

For registering storing class a descendant of StorerReg must be implemented and 2 abstract methods overridden:

```
Type GetStoredClass()
Type GetDataClass()
```

The former must return type of original object class, the later – storing class. The descendant of StorerReg must be registered with SetStorer method of the TypeRepository class.

Appendix A. NODF data examples

Block FORMAT

```
[BLOCK:FORMAT]
Format=TEST
Version=1.0
[BLOCKEND:FORMAT]
```

Block INFORMATION

```
[BLOCK:INFORMATION]
Root=TestClasses.General
Types=39
Objects=45
[BLOCKEND:INFORMATION]
```

Block TYPES

```
[BLOCK:TYPES]
TYPE~0=TestClasses.Structured
TYPE~1=String
TYPE~2=DateTime
TYPE~3=UInt64
TYPE~4=TestClasses.Struct
TYPE~5=Double
TYPE~6=TestClasses.Enumerated
TYPE~7=Object
TYPE~8=TestClasses.Primitive
TYPE~9=Int32
[BLOCKEND:TYPES]
```

Type descriptions TYPENAME

Simple

```
Int32
Double
String
DateTime
TestClasses.Enumerated
```

Arrays

```
Object[,]
Object[][]
Double[,]
String[][]
```

Generics

```
Nullable<Double>
System.Collections.Generic.List<Double>
System.Collections.Generic.Dictionary<String:Object>
```

Data values

Values (VALUE)

```
VALUE=1
VALUE=2.2
VALUE=Three
```


References (REF)

```
REF=3
REF=NULL
```

Lists (LIST)

```
LIST<4:5>=
0.564041984064524 -0.524071000294793 0.917046885898824 0.742994267373809 -0.255815873507325
-0.874795514100602 -0.672671720233127 -0.0310939024347318 0.862441429804285 -0.246964344404156
0.0394569481906746 -0.644531836567694 0.47184890204661 0.581974272421549 0.762016043887481
0.0081122987941431 -0.925377672503413 0.76113086648338 -0.000737712253228606 -0.20026077386004

LIST<5:<3:1:4:NULL:4>:<<3:2:3>:<2>:<1:2:3:4>:<1:2:NULL:NULL>>>=
1.11 1.12 1.13
1.21 1.22
1.31 1.32 1.33
2.11 2.12
3.11
3.21 3.22
3.31 3.32 3.33
3.41 3.42 3.43 3.44
5.11
5.21 5.22

LIST<3:<2:NULL:3>>=
One Two
One Two Last
```

Reference lists (REFLIST)

```
REFLIST<3:2>=
12 NULL
NULL 13
14 16

REFLIST<4:<3:1:NULL:4>:<<3:2:3>:<2>:<1:2:NULL:NULL>>>=
17 NULL 18
19 20
NULL NULL NULL
21 22
23
24 25
```

Binary (BINARY)

```
BINARY=NULL

BINARY<0:0>=

BINARY<722:123>=
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789
012345678901234567890123456789

01234567890123456789

012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

0123456789
```

Member records

```
DATA _size:Int32
/
VALUE=3
\

DATA strValue:String
```



```
/
REF=4
\

DATA regularDoubles:Double[,]
/
LIST<3:4>=
1.1 1.2 1.3 1.4
2.1 2.2 2.3 2.4
3.1 3.2 3.3 3.4
\

DATA dateData:TYPE~3
/
VALUE=634609728000000000
\

DATA doubleField:TYPE~5
/
VALUE=1.1
\

DATA data:TYPE~11
/
REFLIST<4:5>=
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
\
```

Object records

```
OBJECT~14:TestClasses.Simple
{
DATA intValue:Int32
/
VALUE=1
\
DATA strValue:String
/
REF=15
\
DATA doubleValue:Double
/
VALUE=3.3
\
}

OBJECT~15:String
{
VALUE<1>=
2
}

OBJECT~16:DateTime
{
DATA dateData:UInt64
/
VALUE=32
\
}

OBJECT~6:TYPE~8
{
DATA privateInt:TYPE~9
/
VALUE=1
\
DATA privateDouble:TYPE~5
/
VALUE=2.2
\
}
```

