

**CXHibernate**

**Hibernate for C++**

*Version: 0.5.0*

*Status: Alpha*

*Author: ir. W.E. Huisman*

Copyright 2018, ir. W.E. Huisman

All rights reserved. Published under the MIT software license

Contents

[INTRODUCTION 3](#_Toc513037906)

[ARCHITECTURE 4](#_Toc513037907)

# INTRODUCTION

CXHibernate is a database framwork to communicate with a persistent object store. Most commonly this is a database, but also a filestore or a vanilla store on the internet are possibilities to persist objects. Because CXHibernate is a C++ framwork, it uses C++ objects. These objects can be stored, retrieved, updated or deleted from SQL databases that are interfaced throug the general ODBC standard. All SQL databases have such a general [Open Database Connectivity](https://en.wikipedia.org/wiki/Open_Database_Connectivity) layer as defined by the [Microsoft ODBC standard](https://msdn.microsoft.com/en-us/library/s9ds2ktb.aspx).

# ARCHITECTURE

CONFIGURATION

PERSISTENT CLASSES

BASIC O/R MAPPING

ASSOCIATION MAPPINGS

INHERITANCE MAPPING

# WORKING WITH OBJECTS

Objects of classes in the CXHibernate framework that need to be persisted, need to be inherited from the central CXObject class. All classes need a general constructor that have the CXTable as their only parameter, as the objects of the class will belong to this table. In general the interface definition looks like:

|  |
| --- |
| #once  #include <CXObject.h>  class Person : public CXObject  {  public:  Person(CXTable\* p\_table);  virtual ~Person();  // (De-)Serialize methods  void Serialize (SQLRecord\* p\_record,int p\_mutID);  void DeSerialize(SQLRecord\* p\_record);  // Other methods  ...  private:  // Database persistent attributes  int m\_id;  CString m\_first\_name;  CString m\_surname;  SQLDate m\_birthday;  Int m\_address\_id;  // Other attributes  ...  };  DECLARE\_CXO\_FACTORY(Person); |

Things to be aware of here are:

1. The class is inherited from CXObject;
2. The general constructor has a CXTable pointer as its parameter;
3. The class has overrides for the “Serialize” and “De-Serialize” methods
4. The persistent attributes (that will fill in our database are explicitly separated in the private section from the other attributes. We need them here together so that we can write the (De-)Serialize methods;
5. We declare a factory for creating new objects of this type of class.

As a bare minimum we need to write at least de database serialization and deserialization methods. But it’s better to write four of them instead. 2 for the database and 2 for the SOAP/XML message that will be needed to store objects on a filestore or to send them through the internet to a webservice endpoint.

These are the standard serialization methods.

|  |
| --- |
| // Bring the contents of the class to a SOAPMessage or a SQLRecord  virtual void Serialize(SOAPMessage& p\_message,XMLElement\* p\_entity);  virtual void Serialize(SQLRecord& p\_record, int p\_mutation = 0);  // Read the contents of an object from a SOAPMessage or a SQLRecord  virtual void DeSerialize(SOAPMessage& p\_message,XMLElement\* p\_entity);  virtual void DeSerialize(SQLRecord& p\_record); |

We really do need these methods as C++ has no ‘reflection’ kind of system like C#.NET, so we must do our plumbing ourselves and explicitly write which internal attributes will be persistent on the database store.

To begin writing an implementation we need at least:

|  |
| --- |
| #include "stdafx.h"  #include "Person.h"  #include <CXTable.h>  #include <SQLRecord.h>  #ifdef \_DEBUG  #define new DEBUG\_NEW  #undef THIS\_FILE  static char THIS\_FILE[] = \_\_FILE\_\_;  #endif  DEFINE\_CXO\_FACTORY(Person);  Person::Person(CXTable\* p\_table)  :CXObject(p\_table)  {  } |

Some points to be aware of here are:

1. We use the standard MFC implementation structure (stdafx.h and the debugger redirections so that we can detect memory leaks);
2. We begin by defining our factory to create new objects of type “Person”. This is a general macro from CXObject that creates a simple factory function that creates new objects of type “Person”;
3. The constructor passes on the CXTable pointer to the general main class (CXObject).

FILTERS

TRANSACTIONS

INTERCEPT EVENTS

BATCH PROCESSING

# DATATYPES

All standard fundamental C++ datatypes can be used in the CXHibernate framwork. Apart from these a number of extensions are created that are needed to communicate with the ODBC interface of the database. All datatypes that can be streamed from a to a database are so encapsulated in our own datatypes.

The following datatypes can be used and are supported by CXHibernate:

|  |  |  |
| --- | --- | --- |
| Datatype | ODBC type | Explanation |
| int | SQL\_LONG | Standard 32 bits integer in the range from  -2,147,483,648 to 2,147,483,647 |
| long | SQL\_LONG | Standard 32 bits integer in the range from  -2,147,483,648 to 2,147,483,647 |
| \_\_int64 | SQL\_BIGINT | 64 bits integer in the range from  -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| short | SQL\_SHORT | 16 bits integer from -32768 to 32767 |
| char | SQL\_TINYINT | 8 bits integer from -128 to 127 |
| bool | SQL\_BIT | True or false (0 or 1) |
| unsigned int | SQL\_ULONG | 32 bits unsigned integer in the range from 0 (zero) to 4,294,967,295 |
| unsigned long | SQL\_ULONG | 32 bits unsigned integer in the range from 0 (zero) to 4,294,967,295 |
| unsigned \_\_int64 | SQL\_UBIGINT | 64 bits integer in the range from 0 (zero) to 18,446,744,073,709,551,615 |
| unsigned short | SQL\_USHORT | 16 bits integer from 0 to 65535 |
| unsigned char | SQL\_UTINYINT | 8 bits integer from 0 (zero) to 255 |
| float | SQL\_FLOAT | Approximate floating point number as defined by the IEEE. Upto 3.4E +/- 38 (7 digits) |
| double | SQL\_DOUBLE | Approximate floating point number as defined by the IEEE. Upto 1.7E +/- 308 (15 digits) |
| const char \* | SQL\_CHAR | Pointer to a string of characters |
| CString | SQL\_CHAR | MFC String type |
| void\* + size | SQL\_BINARY | Pointer to a binary buffer. Can be used with database types as BLOB and CLOB |
| SQL\_NUMERIC\_STRUCT | SQL\_NUMERIC | Exact floating point number upto 38 decimal places |
| DATE\_STRUCT | SQL\_DATE | Year, month, day according to the Gregorian calendar. |
| TIME\_STRUCT | SQL\_TIME | Hour, minutes and seconds on a day |
| TIMESTAMP\_STRUCT | SQL\_TIMESTAMP | Full timestamp including fractions of a second |
| SQL\_INTERVAL\_STRUCT | SQL\_INTERVAL | All 13 interval types according to the SQL 9075 ISO standard. |
| SQLGUID | SQL\_GUID | Microsoft GUID compatible type |
| bcd | SQL\_NUMERIC | High level exact floating point number upto 40 decimal places + math functions + operators |
| SQLDate | SQL\_DATE | High level Gregorian date + subfunctions + operators on dates, time(stamp) and intervals |
| SQLTime | SQL\_TIME | High level time type + subfunctions + operators |
| SQLTimestamp | SQL\_TIMESTAMP | High level timestamp type + subfunctions + operators on dates, time(stamps) and intervals |
| SQLInterval | SQL\_INTERVAL | High level interval type for all 13 types |
| SQLGuid | SQL\_GUID | Microsoft GUID compatible type |
| SQLVariant | All above !! | Variant class, encapsulating all other types in this list!! |

QUERY LANGUAGE AND NATIVE SQL

XML MAPPINGS

TOOLS