

**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](#_Toc513731789)

[ARCHITECTURE 4](#_Toc513731790)

[CONFIGURATION 5](#_Toc513731791)

[PERSISTENT CLASSES 6](#_Toc513731792)

[BASIC O/R MAPPING 7](#_Toc513731793)

[ASSOCIATION MAPPINGS 8](#_Toc513731794)

[INHERITANCE MAPPING 9](#_Toc513731795)

[WORKING WITH OBJECTS 10](#_Toc513731796)

[FILTERS 12](#_Toc513731797)

[TRANSACTIONS 13](#_Toc513731798)

[INTERCEPT EVENTS 14](#_Toc513731799)

[BATCH PROCESSING 15](#_Toc513731800)

[DATATYPES 16](#_Toc513731801)

[QUERY LANGUAGE AND NATIVE SQL 17](#_Toc513731802)

[XML MAPPINGS 18](#_Toc513731803)

[TOOLS 19](#_Toc513731804)

# INTRODUCTION

CXHibernate is a database framework 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).

Hibernate is a paradigm that greatly simplifies the tasks of dealing with a database from the program­mers perspective. Although it does not exempt him or her from dealing with database details, the standard workflow of working with persistent objects is quite easy.

CXHibernate does support a number of database platforms, datatypes and Object Relational Mappings. As such, it is a ORM (= Object Relational Mapper).

# ARCHITECTURE

The central working object of the CXHibernate architecture is the "session". A session is your unit of work that gives you access to the object caches, the database, the filestore and (through the internet) other datastores at a different network location.

Objects that are made persistent can be handled directly from the application as if the where 'regular' objects. They can be 'found' through the session. The session wil try to find the objects in the cache at first, and in a second attempt at a different stored 'location'.



# CONFIGURATION

The standard configuration of your application is in general contained in the “configuration.cxh” file in the root directory of the filestore. This is a general XML file with the definition of all of the classes in your application, their attributes and there associations. Loading this file with the general interface:

|  |
| --- |
| hibernate.LoadConfiguration() |

Is enough to get you going. Alternatively you may specify a different file as an argument to this call. The \*.cxh extension of this file is merely a convention, instead of a requirement. The XML configuration file holds the general para meters for the application and the sessions, and also the definition of all classes. This looks like:

|  |
| --- |
| <hibernate>  <strategy>standalone</strategy>  <default\_catalog>test</default\_catalog>  <default\_schema>data</default\_schema>  <class>  <name>invoice</name>  <table>invoice</table>  <attributes>  <attribute name=”id” type=“int” generator=”true” primary=”true” not-null=”true” />  <attribute name=”description” type=”string” />  <attribute name=”total” type=”bcd” />  <attribute name=”vat” type=”bcd” dbs\_name=”total\_vat” />  </attributes>  </class>  …  </hibernate> |

# PERSISTENT CLASSES

# BASIC O/R MAPPING

CXHibernate knows about three different types of object-relational mapping. The current mapping can be found by querying the “hibernate.GetStrategy()” interface. The mappings are:

* MapStrategy::Strategy\_standalone. This is the default strategy, where every object class has exactly one database table, and no class inheritance takes place. All object transactions are always carried out on a 1:1 basis as standard SELECT, INSERT, UPDATE and DELETE actions against the database
* MapStrategy::Strategy\_one\_table. This is the strategy where you can have linear class inheritance. All attributes of the super class and all derived classes of an object are stored in one record of one database table. This strategy is also known as the “table-per-class-hierarchy” mapping.

A drawback of this strategy is that it wastes some database space, and that the attributes of the subclasses cannot have a NOT-NULL constraint in the database.

The advantage of this strategy is that all database operations are against one record of one table and thus gain in performance.

* MapStrategy::Strategy\_sub\_tables. This is the strategy where every class (super class and subclass alike) have each there own table. This strategy is also kwown as the “joined-table-strategy”.

The advantage of this strategy is that it does not waste any database space, and that mandatory attributes can have a NOT-NULL constraint in the database.

The drawback on the other hand is that SELECT statements require “LEFT OUTER JOIN” links to the tables of the subclasses, and that the other statements (insert, update, delete) have to be repeated against multiple database tables.

The mapping strategy can be set by your application by calling “hibernate.SetStrategy()”, but \*\*ONLY\*\* before all classes and configurations are loaded, either by loading the configuration.cxh file, or by loading the table definitions. As soon as there are sessions and classes defined, the strategy is fixed and cannot be changed again. This also means that the strategy is the same for all classes in the application!

# 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

# INTERCEPTION 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 |
| bigint | 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) |
| uint | SQL\_ULONG | 32 bits unsigned integer in the range from 0 (zero) to 4,294,967,295 |
| ulong | SQL\_ULONG | 32 bits unsigned integer in the range from 0 (zero) to 4,294,967,295 |
| ubigint | SQL\_UBIGINT | 64 bits integer in the range from 0 (zero) to 18,446,744,073,709,551,615 |
| ushort | SQL\_USHORT | 16 bits integer from 0 to 65535 |
| uchar | 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) |
| 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 |
| 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!! |
| var |  | Same as the SQLVariant class |

The following should be taken into account:

* Some of these datatypes are elementary C++ types, but the typedef defines (ushort for unsigned short) should be used instead of the elementary types. The reason for this rule is that the typenames are used in the serialization code for objects;
* SQLVariant (var) is an encapsulation of all other datatypes, and is used in the communication with the ODBC driver. Data must be encapsulated in this datatype to be carried to- and from the database driver.

# QUERY LANGUAGE AND NATIVE SQL

# XML MAPPINGS

# TOOLS