

**CX-Hibernate**

**Hibernate for C++**

**REFERENCE MANUAL**

*Version: 0.9.0*

*Status: Alpha*

*Author: ir. W.E. Huisman*

Copyright © 2018, ir. W.E. Huisman

All rights reserved. Published under the MIT software license

Contents

[INTRODUCTION 4](#_Toc516679148)

[1. ARCHITECTURE 5](#_Toc516679149)

[2. CONFIGURATION 6](#_Toc516679150)

[PERSISTENT CLASSES 7](#_Toc516679151)

[BASIC O/R MAPPING 8](#_Toc516679152)

[ASSOCIATION MAPPINGS 9](#_Toc516679153)

[INHERITANCE MAPPING 10](#_Toc516679154)

[WORKING WITH OBJECTS 11](#_Toc516679155)

[FILTERS 13](#_Toc516679156)

[TRANSACTIONS 14](#_Toc516679157)

[INTERCEPTION EVENTS 15](#_Toc516679158)

[BATCH PROCESSING 16](#_Toc516679159)

[DATATYPES 17](#_Toc516679160)

[QUERY LANGUAGE AND NATIVE SQL 18](#_Toc516679161)

[XML MAPPINGS 19](#_Toc516679162)

[TOOLS 20](#_Toc516679163)

# 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).

Working with a database can be a difficult and time-consuming task. Not only is their the task of mapping object-oriented classes, but also all the details of programming the low level operations of SELECT-ing, INSERT-ing, UPDATE-ing and DELETE-ing the objects in/from the database.

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. It acts as a go between layer between your application and the database and it’s drivers. CXHibernate does support a number of database platforms, datatypes and Object Relational Mappings. As such, it is a ORM (= Object Relational Mapper).

If you are new tot CXHibernate suggested reading is at least

* Chapter 1: The architecture of Hibernate
* Chapter 2: The configuration files
* Chapter 3: A basic “Hello World” example

After you have familiarized yourself with the basic, suggested reading continues with:

# 1. 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'.



Objects that are not kept track of are referenced as ‘transient’ objects. Meaning that they will ‘go away’ when the program closes and are not persisted in a database, internet or file storage layer.

Handeling the objects is no different in each of these three cases.

The config.xml file (default ‘hibernate.cfg.xml’) describes the data classes in your application and in the storage layers

You can chain two applications together to form a 'cloudstore'. The client side will request objects from the server side that resides 'somewhere-in-the-cloud'. Besides the configuration of the application, there is no difference from storing and retrieving objects from a database. This cloudstore configuration is described in the following image:



# 2. CONFIGURATION

The standard configuration of your application is in general contained in the “hibernate.cfg.xml” file in the root directory your application. 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 is transparent when you use the default name.

|  |
| --- |
| CXSession\* session = hibernate.GetSession(“ses”); |

Any other name can be loaded with the general interface when requesting a new working session. This works by requesting an explicit session from an alternate configuration, as in:

|  |
| --- |
| CXSession\* session = hibernate.LoadConfiguration(“ses”, “C:\Path-to-app\My\_config.xml”); |

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>  <logfile>C:\TMP\My\_hibernate\_logfile.txt</logfile>  <loglevel>6</loglevel>  <database\_use>use</database\_use>  <class>  <name>country</name>  <schema>data</schema>  <table>country</table>  <discriminator>cty</discriminator>  <attributes>  <attribute name="id" datatype="int" generator="true"isprimary="true" />  <attribute name="name" datatype="string" maxlength="100" />  <attribute name="inhabitants" datatype="int" />  <attribute name="continent" datatype="string" maxlength="20" />  </attributes>  <identity name="pk\_country">  <attribute name="id" />  </identity>  <generator name="country\_seq" start="1" />  </class>  </hibernate> |

In fact: this is al that is needed for the example in the next chapter.

As you can ses: the configuration file has a few general settings, and then contains one or more classes and there structure. Whether this be stand-alone classes without any object-oriënted hierarchy or complex hierarchies, class associations, indices and the rest.

Most basic is the fact that the class description names all transient attributes in your class (and thusly in the database table). Of course, your application’s objects can have more data members than just these attributes, but these are the one that will get persisted in the database.

Special care goes to the primary key column (in this case “id”). New instances of objects are created by the generator (starting with the number ‘1’). In the database this column will be part of the primary key, thus forming the identity of the object and of the record in the database table.

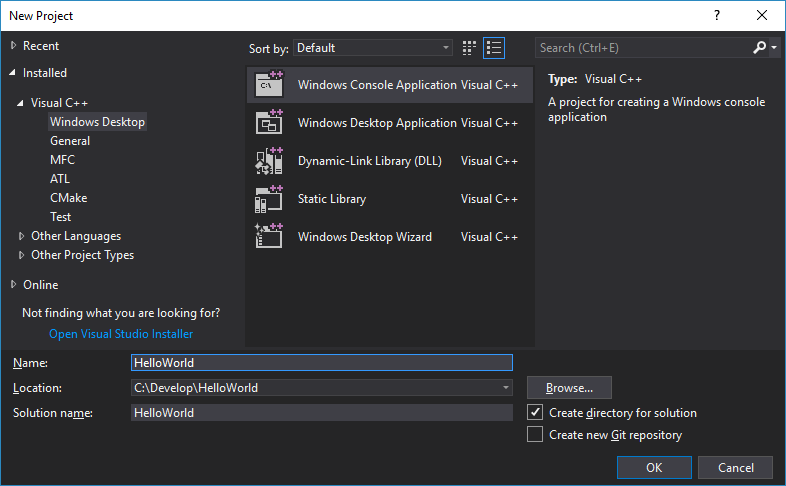
Business keys – and so primary keys – can be made up of multiple columns, but one of these columns is assigned to the sequence generator with the ‘generator=”true”’ attribute.

# 3. A BASIC “HELLO WORLD” EXAMPLE

After a long standing tradition of introducing programmers to a new paradigma, we will program a database version of ‘Hello World!” with CXHibernate.

This walk through begins with a new solution directory “HelloWorld” and a solution file in Visual Studio 2017 (any version of Visual Studio will do). We begin with a standard “Windows Console Application”.

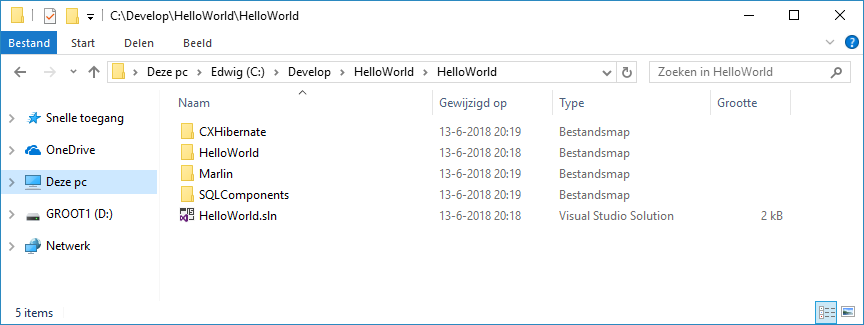
Be sure to ‘unselect’ the ‘Create new Git repository’ option.



From github at <https://github.com/edwig/cxhibernate> we add the following component directories:

* CXHibernate
* SQLComponents
* Marlin

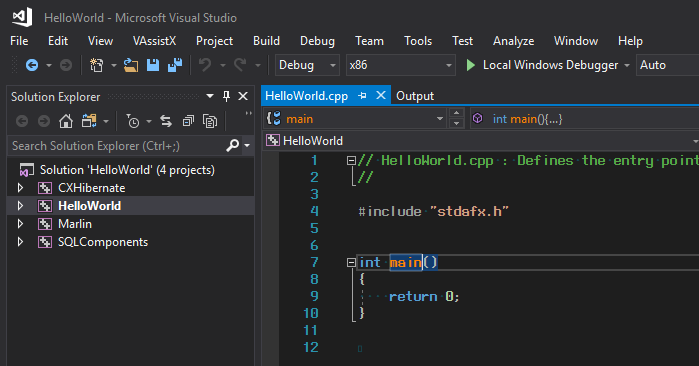
After this inclusions the solution directory should look something like:



(Sorry for the Dutch explorer, and yes ‘GROOT’ does means ‘BIG’)

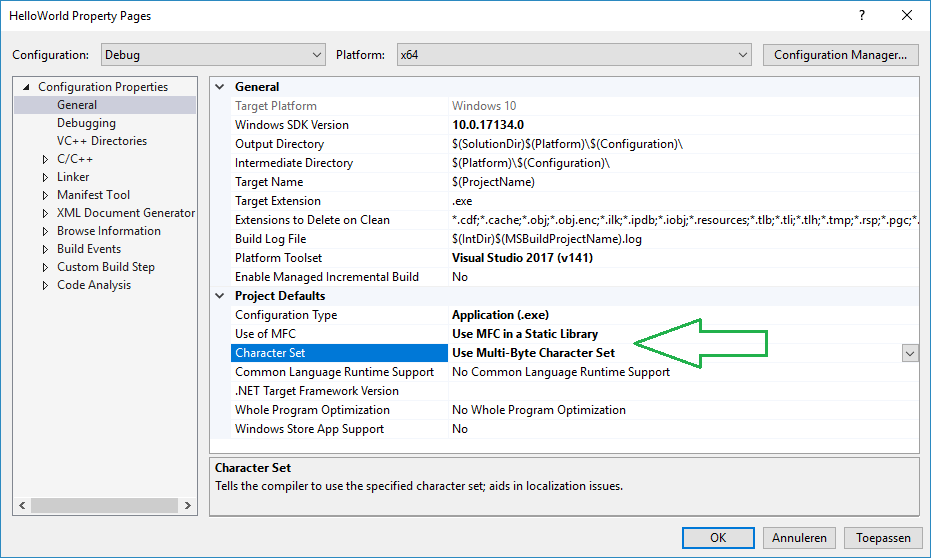
After we have copied the three component directories, we can include the project files of these components in our solution. Just use the “Add…” and “Existing project…” options on the solution level of the “Hello World” solution

After the inclusion of the three project files, your solution should loke something like:



Before we can now begin programming in our “HelloWorld.cpp” file, we need to change some of the project settings, to be able to use the three added components. These are the settings we need to make:

1. Change the “Use of MFC” to “Use MFC in a Static Library”
2. Change “Character Set” to “Use Multi-Byte Character Set”



The Hibernate modules are all compiled to be used as static linked libraries. This was done to escape from the ‘DLL Hell’ when installing an application. But you can change that of course at your own leasure if you so please.

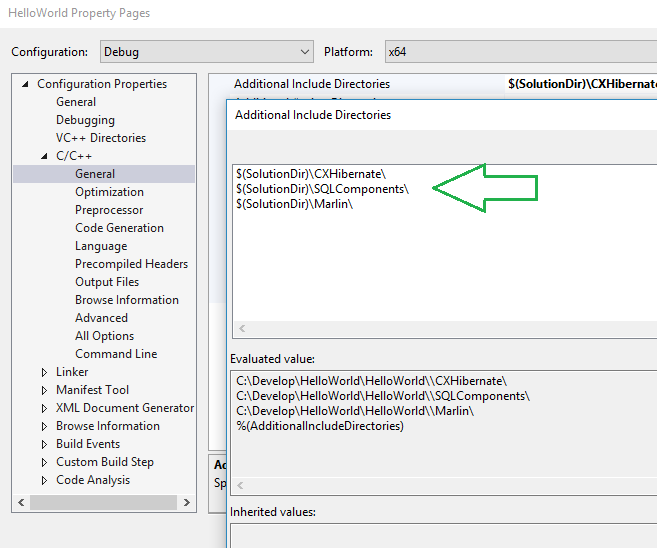
Secondly the whole Framework was built in Western Europe with no need or emphasis on Unicode and further Internationalisation. So everything currently only works under the MBCS character set.

*A Unicode UTF-8 or UTF-16 version is on the whish list.*

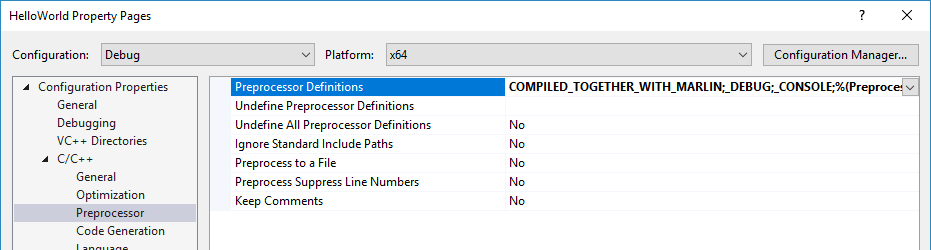
We proceed with the include paths needed for our project. The extra components and their header files need to be found by the compiler so we add the following paths:

* $(SolutionDir)CXHibernate\
* $(SolutionDir)SQLComponents\
* $(SolutionDir)Marlin\

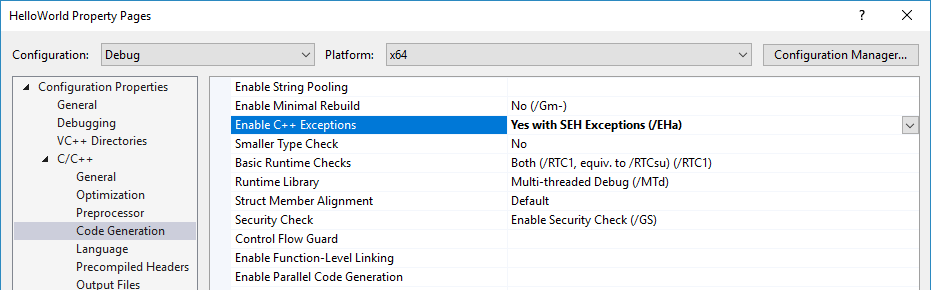
This is done on the C++ General properties page, on the first line “Additional Include Directories”:



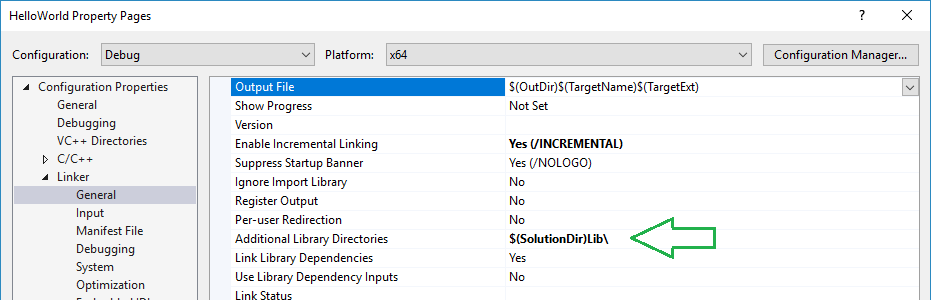
On the “Preprocessor” page we add “COMPILED\_WITH\_MARLIN” to the preprocessor definitions:



On the “Code Generation” page we enable the asynchroneous exceptions for the SQLComponents and Marlin type exceptions:



A last step is to add the path to the “Lib” directory for linking with the resulting libraries of the “Marlin”, “SQLComponents” and “CXHibernate” modules. Go to the “Linker / General” page and fill in the “$(SolutionDir)Lib\” path at the “Additional Library Directories” setting:



Ok. We’re good to go. You can now in essence compile the application, but first we need to add code to our “main()” function, and add a persistent class called “Country” to a our application.

To create a persistent class real quick, add the configuration file from chapter 2, to our “Hello World” directory and run the “CXH2CPP” utility against it from the command line with the option:

**CXH2CPP Country**

This wil generate the “country.h”, “country.cpp”and “country\_cxh.cpp” files. Include these files in your “Hello world project”.

Before you can compile them you need to make one more modification, in this case to your “stdafx.h” file. This is what you must add at the end of the file:

|  |
| --- |
| #include <afx.h>  #include <SQLComponents.h>  #include <CXHibernate.h>  #include <Marlin.h> |

Not only will this allow you to use MFC, but also CXHibernate. Also the names of the specific libraries to your configuration and platform will be automatically configured in Visual Studio.

Compiling one single file or multiple files will now result in auto linking to the libraries:

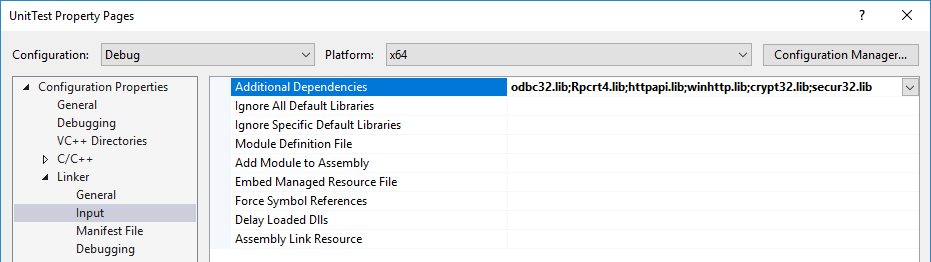
|  |
| --- |
| 1>------ Build started: Project: HelloWorld, Configuration: Debug x64 ------  1>stdafx.cpp  1>Automatically linking with SQLComponents\_x64D.lib  1>Automatically linking with CXHibernate\_x64D.lib  1>Automatically linking with Marlin\_x64D.lib  1>country.cpp  1>country\_cxh.cpp  ========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ========== |

Still, we cannot build the needed runtimer if we do not specify an extra mandatory set of MS-Windows components that are needed by the Marlin and SQLComponents framework. Otherwise we would get a bunch of “Unresolved external symbol” errors from the system linker.

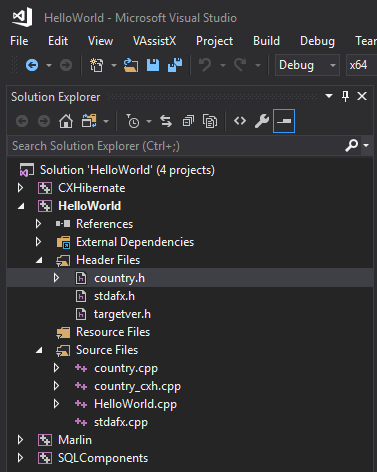
The extra components are:

* odbc32.lib For ODBC and ODBC-Manager functions
* Rpcrt4.lib Needed for the generation of Microsoft GUID’s
* httpapi.lib Needed for the server access to the HTTP service protocol
* winhttp.lib Needed for the client access to the HTTP protocol
* crypt32.lib Needed for encrypted webservices
* secur32.lib Needed for

Add them to the “Linker / Input” page of the project file



OK, now we have everything. Our project should look like:



And everything should compile fine, but for the fact that it does not do anything (yet).

But first let take a peek at the generated files for our “country” class.

This is the implementation \*.CPP file

|  |
| --- |
| // Implementation file for class: country  // Automatically generated by: CX-Hibernate  //  #include "stdafx.h"  #include "country.h"  #ifdef \_DEBUG  #define new DEBUG\_NEW  #undef THIS\_FILE  static char THIS\_FILE[] = \_\_FILE\_\_;  #endif  // CTOR for class  country::country()  {  // Things to do in the constructor  }  // DTOR for class  country::~country()  {  // Things to do in the destructor  } |

The interface file (\*.H)

|  |
| --- |
| // Interface definition file for class: country  // Automatically generated by: CX-Hibernate  // File: country.h  //  #pragma once  #include <CXObject.h>  #include <bcd.h>  #include <SQLDate.h>  #include <SQLTime.h>  #include <SQLTimestamp.h>  #include <SQLInterval.h>  #include <SQLGuid.h>  #include <SQLVariant.h>  class country : public CXObject  {  public:  // CTOR of an CXObject derived class  country();  // DTOR of an CXObject derived class  virtual ~country();  // Serialization of our persistent objects  DECLARE\_CXO\_SERIALIZATION;  // GETTERS  int GetId() { return m\_id; };  CString GetName() { return m\_name; };  int GetInhabitants() { return m\_inhabitants; };  CString GetContinent() { return m\_continent; };  protected:  // Database persistent attributes  int m\_id { 0 };  CString m\_name ;  int m\_inhabitants { 0 };  CString m\_continent ;  private:  // Transient attributes go here  }; |

And on the following page we find the generated “country\_cxh.cpp” file. This is the place where we do our serialization and deserialization. This comes in the place for where other variants of Hibernate can do reflection. C++ has no metadata, so the serialization is done by these macro’s.

|  |
| --- |
| // (De-)Serializing factories for class: country  // Generated by CX-Hibernate cfg2cpp tool  //  #include "stdafx.h"  #include "country.h"  #include <SQLRecord.h>  #include <SOAPMessage.h>  #ifdef \_DEBUG  #define new DEBUG\_NEW  #undef THIS\_FILE  static char THIS\_FILE[] = \_\_FILE\_\_;  #endif  BEGIN\_XML\_SERIALIZE(country,CXObject)  CXO\_XML\_SERIALIZE(int ,m\_id ,"id" ,XDT\_Integer);  CXO\_XML\_SERIALIZE(CString ,m\_name ,"name" ,XDT\_String);  CXO\_XML\_SERIALIZE(int ,m\_inhabitants ,"inhabitants" ,XDT\_Integer);  CXO\_XML\_SERIALIZE(CString ,m\_continent ,"continent" ,XDT\_String);  END\_XML\_SERIALIZE  BEGIN\_XML\_DESERIALIZE(country,CXObject)  CXO\_XML\_DESERIALIZE(int ,m\_id ,"id" ,XDT\_Integer);  CXO\_XML\_DESERIALIZE(CString ,m\_name ,"name" ,XDT\_String);  CXO\_XML\_DESERIALIZE(int ,m\_inhabitants ,"inhabitants",XDT\_Integer);  CXO\_XML\_DESERIALIZE(CString ,m\_continent ,"continent" ,XDT\_String);  END\_XML\_DESERIALIZE  BEGIN\_DBS\_SERIALIZE(country,CXObject)  CXO\_DBS\_SERIALIZE(int ,m\_id ,"id" ,XDT\_Integer);  CXO\_DBS\_SERIALIZE(CString ,m\_name ,"name" ,XDT\_String);  CXO\_DBS\_SERIALIZE(int ,m\_inhabitants ,"inhabitants",XDT\_Integer);  CXO\_DBS\_SERIALIZE(CString ,m\_continent ,"continent" ,XDT\_String);  END\_DBS\_SERIALIZE  BEGIN\_DBS\_DESERIALIZE(country,CXObject)  CXO\_DBS\_DESERIALIZE(int ,m\_id ,"id" ,XDT\_Integer);  CXO\_DBS\_DESERIALIZE(CString ,m\_name ,"name" ,XDT\_String);  CXO\_DBS\_DESERIALIZE(int ,m\_inhabitants ,"inhabitants",XDT\_Integer);  CXO\_DBS\_DESERIALIZE(CString ,m\_continent ,"continent" ,XDT\_String);  END\_DBS\_DESERIALIZE  BEGIN\_DESERIALIZE\_GENERATOR(country)  CXO\_DBS\_DESERIALIZE(long, m\_id, "id", XDT\_Integer);  END\_DESERIALIZE\_GENERATOR  // Static factory to create a new object if this class  DEFINE\_CXO\_FACTORY(country); |

Now finally we can get some work done. We can now start to fill in our “main()” function of the application. Request a session from the global “hibernate” object and load a first country with the id=1 into memory. If all goes well, we can directly begin calling methods of the object, and print a “Hello world” on the console.

After we have done our work, optionally we can now close the session.

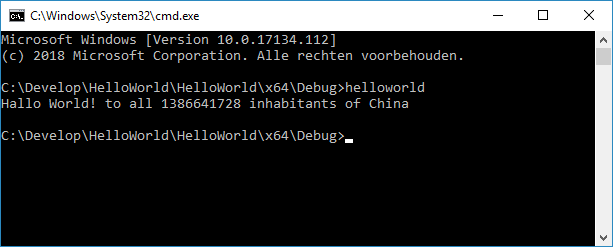
Of course you should also have an ODBC connection named “hibtest” to the test database of the CX-Hibernate project. In this case a Firebird 3.0 database, with the “COUNTRY” table in it.

(Filled from: <https://simple.wikipedia.org/wiki/List_of_countries_by_population> )

|  |
| --- |
| // HelloWorld.cpp : Defines the entry point for the console application.  //  #include "stdafx.h"  #include "country.h"  #include <CXHibernate.h>  #include <CXSession.h>  int main()  {  CXSession \* session = hibernate.CreateSession();  if(session)  {  // Set a database session  session->SetDatabaseConnection("hibtest","sysdba","altijd");  country\* land = (country\*)session->Load(country::ClassName(),1);  if(land)  {  printf("Hallo World! to all %d inhabitants of %s\n"  ,land->GetInhabitants()  ,land->GetName().GetString());  }  else  {  printf("Cannot find a country with id = %d\n",1);  }  // And close our session  session->CloseSession();  }  return 0;  } |

Now compile, copy the “hibernate.cfg.xml” to the runtime directory and run it!

And low and behold!



This concludes our “Hello World” first example. Of course you can go off now and create all kinds of extra test to this simple first program. Suggestions are that you try to:

* Add an extra parameter to the program to request a different country
* Add a load with a filter to request a set of countries and print them all
* Count all the inhabitants in the world and say “Hello” to all of them, spreading ‘peace and happiness’ to the world 😊

# 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\_table. 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.

*Coming in a later version:*

* *MapStrategy::Strategy\_classtable. This is the strategy where every class have there own table. Even for super- and subclasses. This strategy is also kwown as the “union-table-strategy”, because it takes a “SELECT union SELECT” construct, to get all the relevant records from the database when querying for an object.*

*The advantage of this strategy is that it wastes no space, and that every attribute in every class can have the full swing of the database help like NOT-NULL constraints and such.*

*The drawback on the other hand is that it takes quite complex multiple SELECT statements chained together with a UNION construct. These are inherently slower than the select state­ments in the other strategies.*

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 currently the same for all classes in the application!

*Planned for a later version is the configuration where each class hierarchy can have it’s own mapping strategy. For now the strategy is fixed for the complete 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

In application development – and especially in large applications – it’s always handy to have a mechanism to intercept the base operations. Even if it’s in the very last stages of a transaction. Multiple programmers may work in a large scale project with different knowledge of the lay of the land, or you just want to have a way to do a quick and dirty solution for the duration of a quick patch.

Precisly because of these reasons CX-Hibernate has four classes of interception events on the base operations of an object. These events are declared in “CXObject.h” and the are:

* **void OnLoad()**: Just after CX-Hibernate loads an object or a set of objects, the virtual overridable “OnLoad” of that object is called. This event cannot stop the application in loading this object, but it can perform extra actions on the just loaded object. The return type is ‘void’, because it cannot stop the load;
* **bool OnInsert()**: Before inserting an object in the datastore, this trigger gets called. Be aware that the trigger gets called before the generator value can be assigned to the designated attribute / property of the object. The primary key / business key of this object is bound to be incomplete upon calling this trigger. But as with the other triggers, we can perform extra actions or even stop the insert by returning ‘false’ from the trigger;
* **bool OnUpdate()**: Before updating an object in the datastore, this trigger gets called. Here you can add extra actions to perform before every update, or you can even stop the update by returning ‘false’ from this trigger. Be sure to return ‘true’ if you want the update to continue;
* **bool OnDelete()**: Before deleting an object from the datastore, this trigger gets called. Here you can add extra actions to perform before every deletion, or you can even stop the deletion by return ‘false’ from this trigger. Be sure to return ‘true’ if you want the deletion to continue.

Here are a few more facts to keep in mind about all of the above triggers:

* Triggers on single objects are carried out in relation to the action, but triggers on sets of objects are carried out in one bunch. So if all objects are loaded from the datastore, then the triggers for all objects are fired in a tight loop, all at once;
* Triggers are called for all datastores. It does not matter that the object goes to and from a database, a filestore or a webservice on the internet;
* Triggers are generally called before the logging of the object, so that if the trigger changes the object, the changes are reflected in the logfile. Exception to this rule is the OnDelete trigger, where we log the object to delete before we do anything else;
* Triggers from the framework to the application are called from within a try … catch() loop. The loop will intercept the “StdException” exception only and log the fact that the trigger misbehaved in the logfile at the ‘error-level’;
* Overloaded events are NOT generated by the “CXH2CPP” utility. In general it’s unlikely that you will need these interception events right at the start of an application;
* It is a good custom to keep the code of the events in the <classname>.cpp file and not in the accompanying “<classname>\_cxh.cpp” file. Reasons for this rule / custom is that we can later on re-generate the \*\_cxh.cpp” file from scratch – e.g. with the CXH2CPP utility – and then replace it on the file system level, without touching the trigger code.

# BATCH PROCESSING

SQLTransactions

Mutation numbers

Mutation stacks on fields

Committing at the end

# 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