**CXHibernate: A Hibernate framework for C++**

**What?**

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++ framework, it uses C++ objects. These objects can be stored, retrieved, updated or deleted from SQL databases that are interfaced through 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).

Other languages such as Java and C# .NET have had a hibernate framework for many years, to ease the pain of programmers in dealing with databases. You can find the documentation of those frameworks on the internet (Java Hibernate: <https://hibernate.org/orm/documentation> and C# .Net NHibernate <https://nhibernate.info/doc/>)

**Why?**

Working with a database can be a difficult and time-consuming task. Not only is there 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 its drivers. CXHibernate does support a number of database platforms, datatypes and Object Relational Mappings. As such, it isan 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'.



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:



**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(); |

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 paragraph.

As you can see: 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.

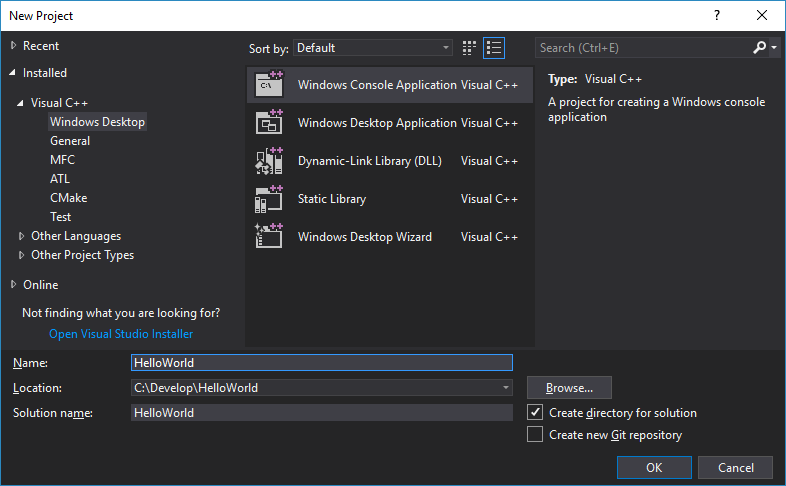
Business keys – and so primary keys – can be made up of multiple columns, but one of these columns is assigned to the sequence generator with the ‘generator=”true”’ attribute.

**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”.

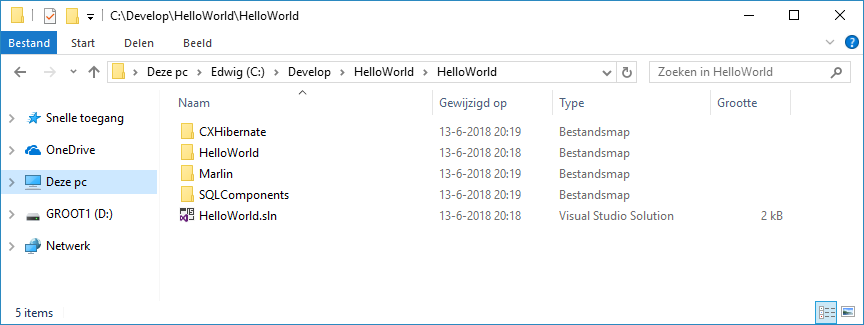
Be sure to ‘unselect’ the ‘Create new Git repository’ option.



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

* CXHibernate
* SQLComponents
* Marlin

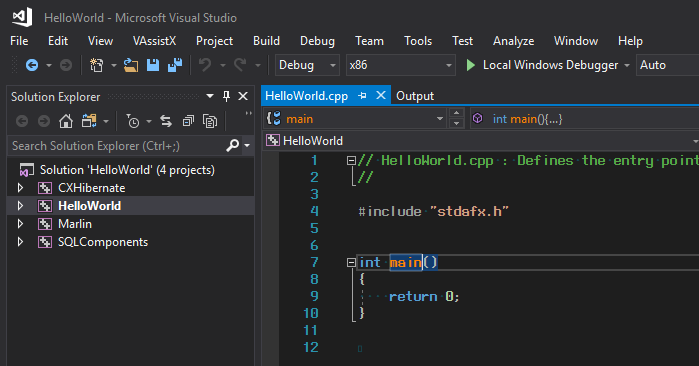
After this inclusions the solution directory should look something like:



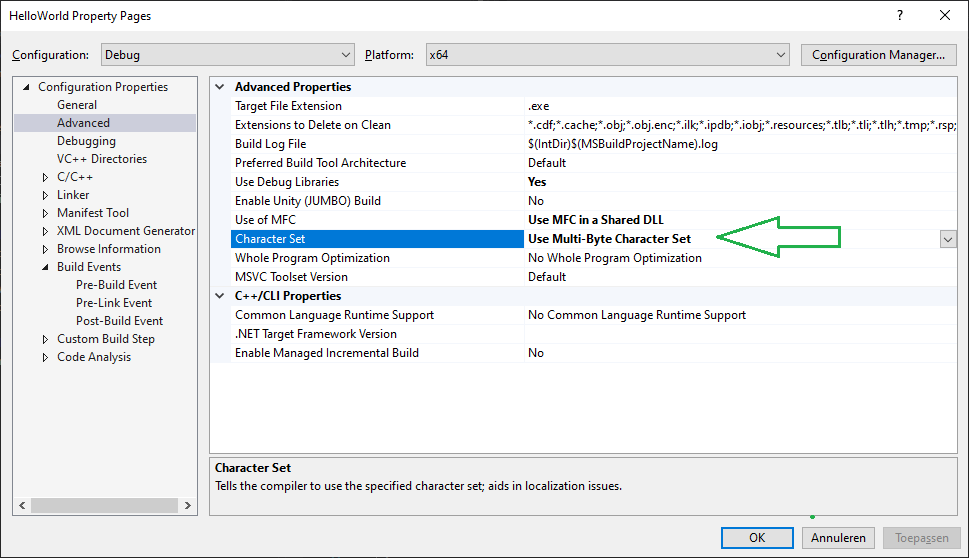
(Sorry for the Dutch explorer, and yes ‘GROOT’ is not a walking tree, it does mean ‘BIG’)

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. The first setting we need to make is to change the use of the “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.

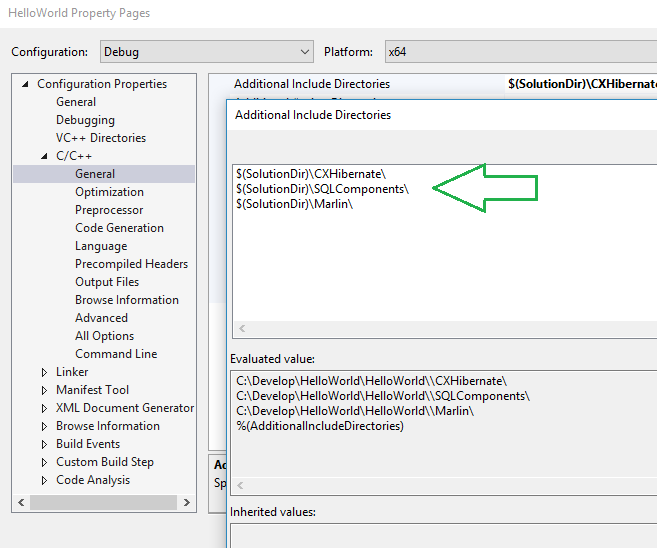
Secondly the whole Framework was built in Western Europe with no need or emphasis on Unicode and further Internationalisation. So everything currently only works under the MBCS character set.

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

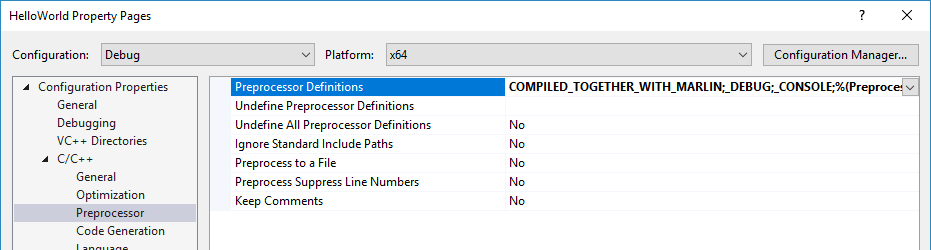
We proceed with the include paths needed for our project. The extra components and their header files need to be found by the compiler so we add the following paths:

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

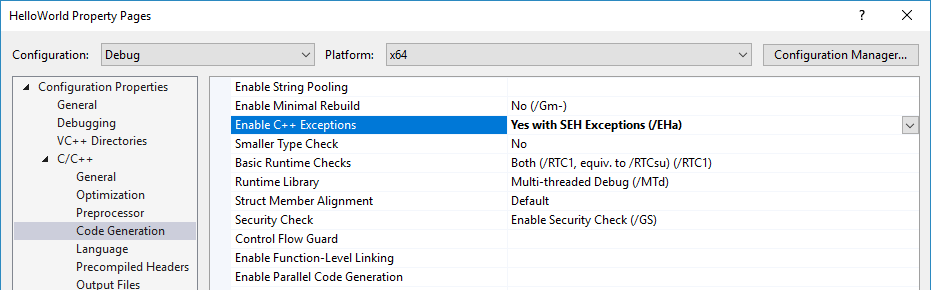
This is done on the C++ General properties page, on the first line “Additional Include Directories”:



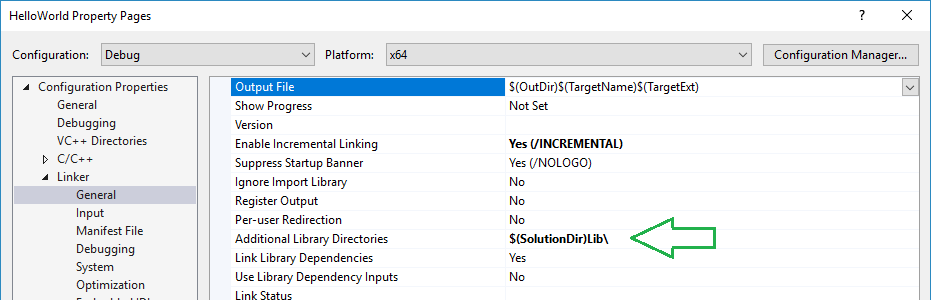
On the “Preprocessor” page we add “COMPILED\_WITH\_MARLIN” to the preprocessor definitions:



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:

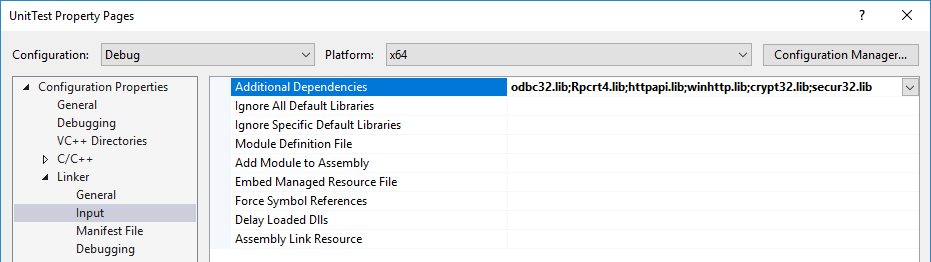
|  |
| --- |
| 1>------ Build started: Project: HelloWorld, Configuration: Debug x64 ------  1>stdafx.cpp  1>Automatically linking with SQLComponents\_x64D.lib  1>Automatically linking with CXHibernate\_x64D.lib  1>Automatically linking with Marlin\_x64D.lib  1>country.cpp  1>country\_cxh.cpp  ==========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

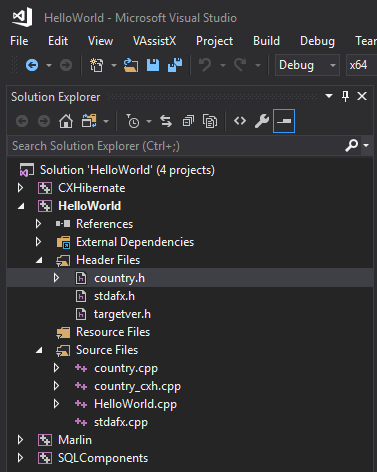
The extra components are:

* 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.

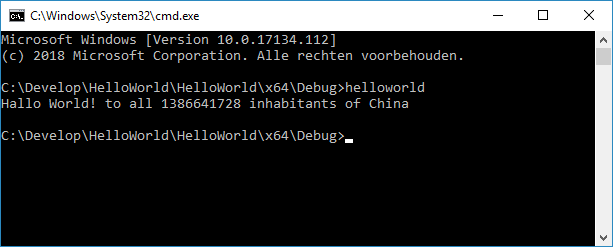
Of course you should also have an ODBC connection named “hibtest” to the test database of the CX-Hibernate project. In this case a Firebird 3.0 database, with the “COUNTRY” table in it.

(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 😊

**Basic Operations**

Now that we have introduced you to CX-Hibernate in the last chapter with a “Hello World” example, lets look at the basic Hibernate operations and their derivates. The basic operations are:

* Load: Get an object from an external store (database, webservice, filestore);
* Insert: Place a new object in hibernation in an external store;
* Update: Change the object in the hibernation store, so that it reflects the one in our app;
* Save: Inserts new objects, or updates existing loaded ones;
* Delete: Delete the object from the hibernation store, and most likely from our application too.

The basic operations are methods of the general CXSession class. Each application’s user must have an active session, in order to be able to perform basic operations on the external store.

Per basic operation, a number of variants and requirements are described in the paragraphs below.

**Loading an object**

Loading an object has various overloads in the session. This is done to make it easy on the programmer to take the shortest route to an easy load of the object. The load methods go from simple one parameter loads to a load with a complex set of filters returning a set of objects. These are the load methods of the CXSession:

CXObject\* Load(CString p\_className,int p\_primary);

CXObject\* Load(CString p\_className,CString p\_primary);

CXObject\* Load(CString p\_className,SQLVariant\* p\_primary);

CXObject\* Load(CString p\_className,VariantSet& p\_primary);

CXResultSet Load(CString p\_className,SQLFilter\* p\_filter);

CXResultSet Load(CString p\_className,SQLFilterSet& p\_filters);

There are so many “Load” operations, as to accomodate the situations that the database objects have primary keys that exist out of more than one database column. In many cases where we have just one “id” column, the first “Load” operation with an “integer” primary key is quite sufficient. In cases where we have compound primary keys, the variants with a “VariantSet” and a “SQLFilterSet” the way to go. And in fact, these variants get called internally by the other “Load” operations.

A second thing to notice is the fact that all the “Load” operations have the name of the class as a first parameter. This parameter can be filled with a static string like e.g. “classname” (case-insensitve!!), but it’s easier and more natural to use the “ClassName” from the object factory of the class. We have already seen such an example in the HelloWorld program of the previous chapter as in:

|  |
| --- |
| ...  Country\* land = (Country\*)session->Load(Country::ClassName(),1);  ... |

**Inserting an object**

This is a two-step procedure. Inserting an object requires the creation of an object of a certain persistent class first. After filling the object with data, and perhaps a partial primary key, we can ask Hibernate to insert the object into the external store.

Because a ‘Cat’ is a derived class from ‘Animal’ the object is stored in two tables (cat and animal).

Here is an example of such a two fase insertion:

|  |
| --- |
| ...  CXSession\* session = hibernate.CreateSession();  ...  Cat\* pussy = (Cat\*) session->CreateObject(Cat::ClassName());  // Fill in our object with reasonable values  pussy->SetAnimalName("Silvester");  pussy->SetHas\_claws(true);  pussy->SetLikesBirds("Tweety");  // Go save it in the database in two tables (Animal and Cat)  bool result = session->Insert(pussy);  ... |

Instead of “Insert” you may also use “Save” as an equivalent.

The “CreateObject” method of the session is called instead of “new”-ing the object. This assures us that the object was created by the object factory and that all necessary operations for the Hibernate framework have been taken care off. **Never “new” your object!**

A side effect of inserting the object into an external store is that the primary key in the base class CXObject will get filled in, which in it’s turn changes the state of the object from “transient” to “persistent”.

Another side effect is that the object is referenced in the object cache. You need not keep track of it in your application for free-ing. The hibernate framework does that for you.

**Updating an object**

Updating a single object is quite simple. Just call the session’s “Update” method. That’s it.

Here is an example:

|  |
| --- |
| ...  CXSession\* session = hibernate.CreateSession();  ...  Cat\* pussy = (Cat\*) session->Load(Cat::ClassName(),42);  // change the state of the object  pussy->SetColor("black-and-white");  // Go update the external store  bool result = session->Update(pussy);  ... |

**Deleting an object**

Deleting a single object is quite simple. Just call the session’s “Delete” method. That’s it.

Here is an example:

|  |
| --- |
| ...  CXSession\* session = hibernate.CreateSession();  ...  // Load a certain object  Cat\* pussy = (Cat\*) session->Load(Cat::ClassName(),13);  // Go delete from the external store  bool result = session->Delete(pussy);  ... |

**Please remember:** if the deletion goes well (resulting in a ‘true’ return value), the object is removed from the hibernate caches AND it is destroyed by a “delete” action of the C++ language. Your pointer to the object is then no longer valid!

Only when the “Delete” operation returns a ‘false’ (whether the object gets not deleted from the database or from the cache) the pointer is still intact. After optionally logging this fact, the caller (you!) must call “Delete” on the object.

Mind you, there is **\*NO\*** need to use the C++ “delete <pointer>” operator on the pointer. That’s done in the Delete(CXObject\*) method of the CXSession!

**The default implementation**

When you are following the default implementation. E.g. by generating the class implementation with the CFG2CPP tool, you will get:

* A header file with all data members and default setters and getters for all the data members. The class declaration then contains one (1) extra macro for the declaration of the serialization factory ("DECLARE\_CXO\_SERIALLIZATION")
* A factory implementation file, containing the serialization and de-serialization of the object for both the database and SOAP messages. It also contains the create-object factory that the hibernate framework uses to create a new object of this class.

Mostly, you can leave this implementation file alone. Only when you add new attributes to your class (and columns to the database) need you go here and add those attributes. You can either do one of the following:

1. Create a new set of files with the “cfg2cpp” tool, after you changed the config file and pluck out the \*.cxh.cpp file;
2. Add the attribute by adding lines to the file. Great care has been taken to make the four implementations alike, so that it’s easy to add the lines for a new attribute.

An implementation file with just a constructor and a destructor. You can now start to begin the implementation of your class right away in this file. No visible overhead in your way. So this is really the C++ way: Zero overhead.

**More**

The full documentation (see the documention directory) contains a lot more info about:

* Persistent classes and there possibilities
* O/R Mapping methods
* Supported database assocation between classes (and tables)
* Filters and how to express the search for objects at load time
* Transaction and batch processing
* Interception events like “OnLoad” and “OnSave” and so on
* Datatypes and SQLVariants (and fun with those)
* More info on the tools to generate \*.cpp, \*.h and \*\_chx.cpp files
* Apendices on datatypes, operators and type conversions that you can use in this library

In fact, the full manual is much longer than this article.

**Have fun**

So: Have fun with Hibernation in C++