

**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 3](#_Toc517202513)

[1. ARCHITECTURE 4](#_Toc517202514)

[2. CONFIGURATION 6](#_Toc517202515)

[3. A BASIC “HELLO WORLD” EXAMPLE 7](#_Toc517202516)

[4. BASIC OPERATIONS 15](#_Toc517202517)

[4.1 Loading an object 15](#_Toc517202518)

[4.1.1 Load from an integer identifier 15](#_Toc517202519)

[4.1.2 Load from a string identifier 16](#_Toc517202520)

[4.1.3 Load from a single identifier (not being integer or string) 16](#_Toc517202521)

[4.1.4 Load from a compound set of values 16](#_Toc517202522)

[4.1.5 Load of a set of objects from a single condition filter 16](#_Toc517202523)

[4.1.6 Load of a set of objects from multiple condition filters 17](#_Toc517202524)

[4.2 Inserting an object 18](#_Toc517202525)

[4.3 Updating an object 18](#_Toc517202526)

[4.4 Deleting an object 19](#_Toc517202527)

[5. PERSISTENT CLASSES 20](#_Toc517202528)

[5.1 Rules 20](#_Toc517202529)

[5.2 The default implementation 21](#_Toc517202530)

[5.3 Overridable virtual functionality 21](#_Toc517202531)

[6. BASIC O/R MAPPING 22](#_Toc517202532)

[7. ASSOCIATION MAPPINGS 23](#_Toc517202533)

[8. FILTERS 24](#_Toc517202534)

[9. TRANSACTIONS AND BATCHES 25](#_Toc517202535)

[10. INTERCEPTION EVENTS 26](#_Toc517202536)

[11. DATATYPES 27](#_Toc517202537)

[12. QUERY LANGUAGE AND NATIVE SQL 28](#_Toc517202538)

[13. XML MAPPINGS 29](#_Toc517202539)

[14. TOOLS 30](#_Toc517202540)

[APPENDIX 1: FUN WITH SQLVariant’s 31](#_Toc517202541)

# 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:

* Chapter 4: The Hibernate basic operations (Load, Insert, Update, Delete)
* Chapter 5: The rules for persistent classes
* Chapter 6: On basic O/R mapping and inheritance
* Chapter 7: Mapping associations in CX-Hibernate
* Chapter 8: Filters and how to use them to load objects
* Chapter 9: Transactions and larger batches
* Chapter 10: Interception events on the objects themselves
* Chapter 11: Fundamental datatypes, and how they are supported by SQLComponents
* Chapter 12: On native SQL and queries through the use of SQLComponents
* Chapter 13: Details of the config.xml mapping files
* Chapter 14: Helpful tools

And at last there are appendices:

* Appendix 1: Fun with SQLVariant’s

# 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.

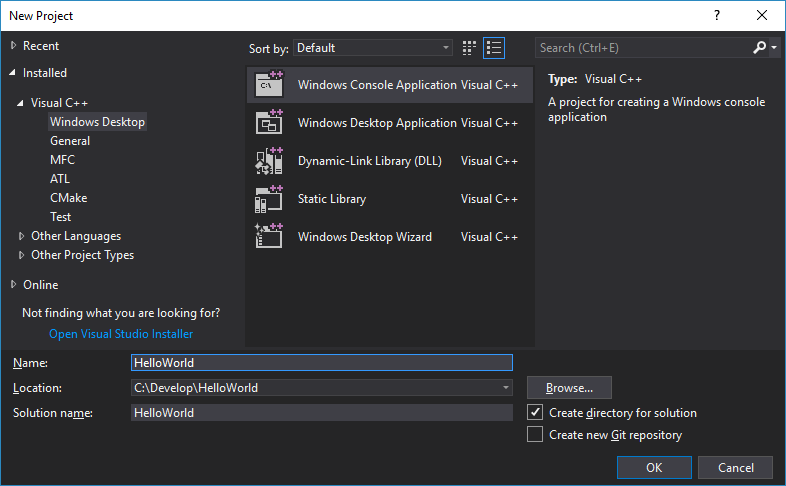
Business keys – and so primary keys – can be made up of multiple columns, but one of these columns is assigned to the sequence generator with the ‘generator=”true”’ attribute.

# 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”.

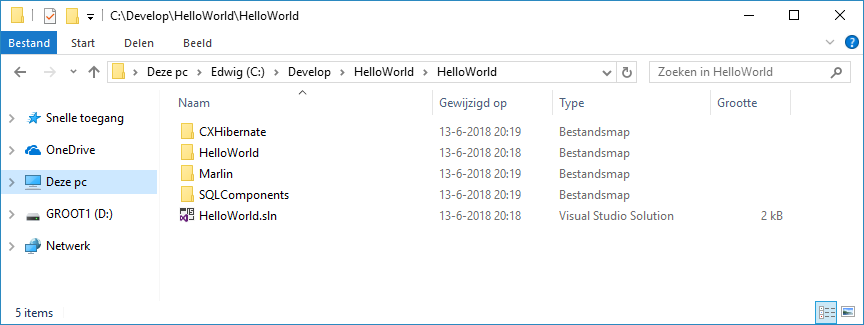
Be sure to ‘unselect’ the ‘Create new Git repository’ option.



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

* CXHibernate
* SQLComponents
* Marlin

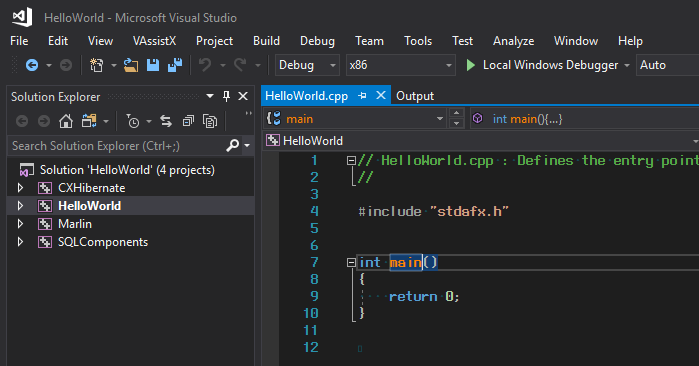
After this inclusions the solution directory should look something like:



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

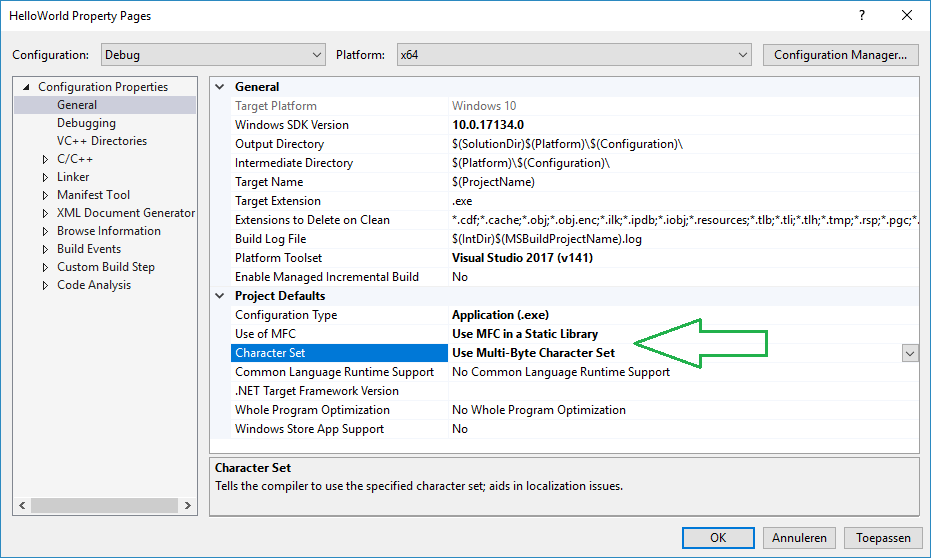
After we have copied the three component directories, we can include the project files of these components in our solution. Just use the “Add…” and “Existing project…” options on the solution level of the “Hello World” solution

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. 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.

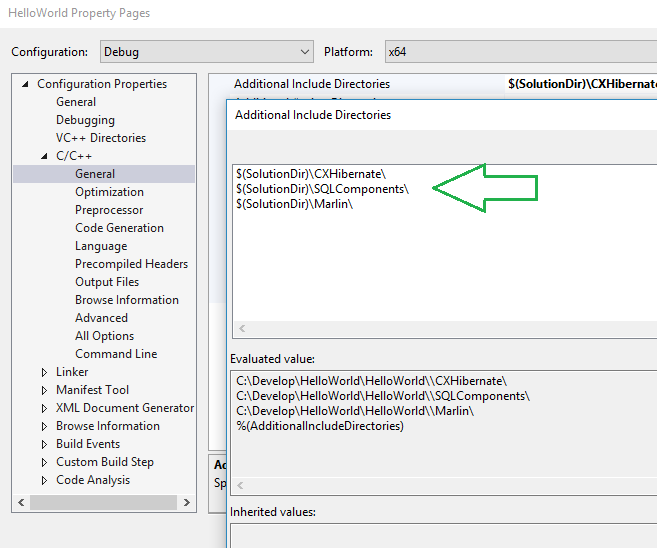
Secondly the whole Framework was built in Western Europe with no need or emphasis on Unicode and further Internationalisation. So everything currently only works under the MBCS character set.

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

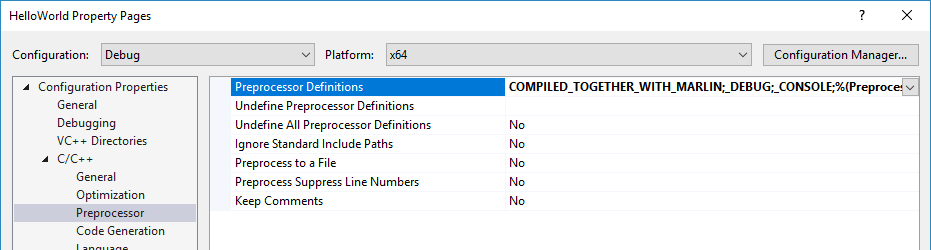
We proceed with the include paths needed for our project. The extra components and their header files need to be found by the compiler so we add the following paths:

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

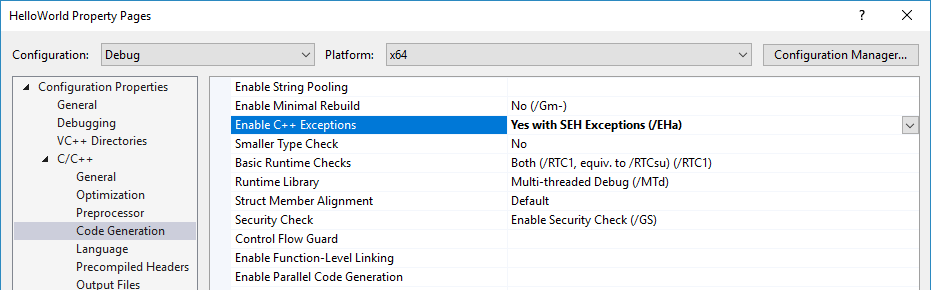
This is done on the C++ General properties page, on the first line “Additional Include Directories”:



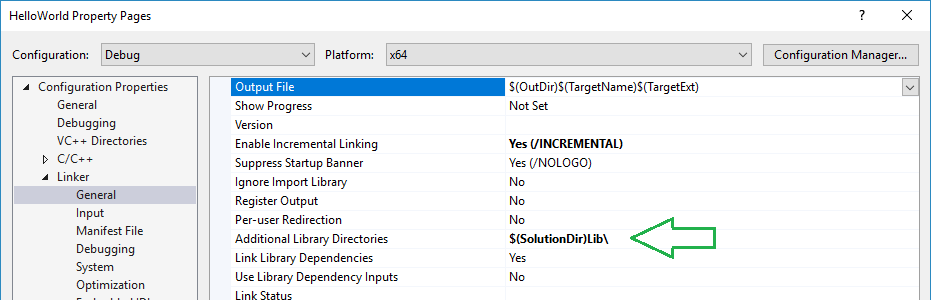
On the “Preprocessor” page we add “COMPILED\_WITH\_MARLIN” to the preprocessor definitions:



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:

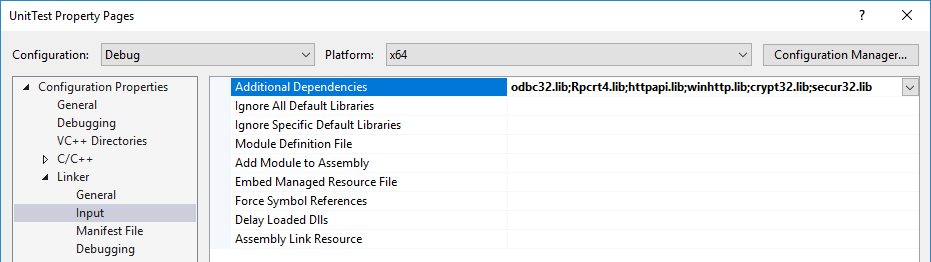
|  |
| --- |
| 1>------ Build started: Project: HelloWorld, Configuration: Debug x64 ------  1>stdafx.cpp  1>Automatically linking with SQLComponents\_x64D.lib  1>Automatically linking with CXHibernate\_x64D.lib  1>Automatically linking with Marlin\_x64D.lib  1>country.cpp  1>country\_cxh.cpp  ==========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

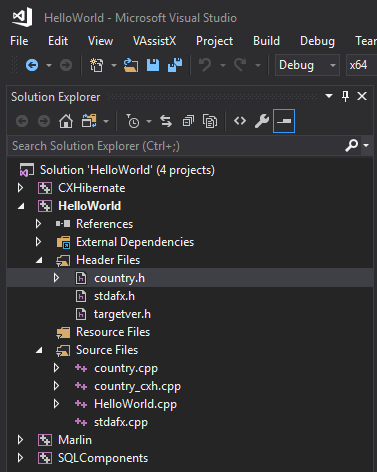
The extra components are:

* 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.

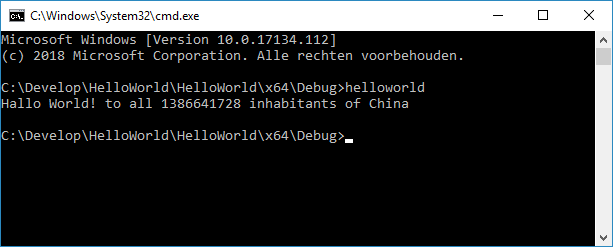
Of course you should also have an ODBC connection named “hibtest” to the test database of the CX-Hibernate project. In this case a Firebird 3.0 database, with the “COUNTRY” table in it.

(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 😊

# 4. 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.

## 4.1 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);  ... |

Here are all the load operations:

### 4.1.1 Load from an integer identifier

The most common method to identify an object in the database – and mostly the fastest – is by using an integer “id” column. If you – your application – happen to now the id, the load is most easily done by using the “Load” variant with the integer primary key as a second parameter, as in the example above.

In general, a network database with all de-personalized “id” numbered identifiers is by far the most performant way of putting a database application together.

That’s why this type of load is recommended;

**CXObject\* Load(CString p\_classname,int p\_primaryKey);**

### 4.1.2 Load from a string identifier

In the case where you have a single string column as the primary key of a class of objects, the optimized form with a CString as second parameter is the quickest way to load an object:

**CXObject\* Load(CString p\_classname,CString p\_primaryKey);**

Notice that code tables with (“string identifier”, “string tekst\_info”) pairs often have this type of format in the database. For such code tables this type of load is the easiest.

### 4.1.3 Load from a single identifier (not being integer or string)

In cases where the primary key column has another datatype than “integer” or “string”, you can load an object through a SQLVariant. SQLVariants can hold any datatype that can be put into an ODBC database or can be retrieved from it. For a more elaborate discussion about SQLVariants, please lookup the chapter on “Datatypes” and the “Fun with Variants” appendix.

**CXObject\* Load(CString p\_classname,SQLVariant\* p\_primaryKey);**

In general this works by first assigning a value to a variant, and then load the object through the variant. For instance, if we supply a GUID in the form of a “p\_guid” parameter, we can:

|  |
| --- |
| ...  SQLVariant myGuid(p\_guid)  Hardware\* pc = (Hardware\*)session->Load(Hardware::ClassName(),&myGuid);  ... |

Load the object through the use of a SQLVariant, supplied as the primary key of the object.

### 4.1.4 Load from a compound set of values

CX-Hibernate is especially not restricted to objects with a single identifier primary key. In fact, most relational database have tables with compound primary keys. That’s why it’s made easy to work with those objects. If you want to load an object with a primary key, you shall have to supply a compound set of values to load the object. This is done through the VariantSet. In fact it’s a simple “std::vector” of SQLVariant’s. Loading is done by

**CXObject\* Load(CString p\_classname,VariantSet& p\_primaryKey);**

This makes it easy to refer to a compound primary key. In fact, internally CX-Hibernate works always on VariantSet’s of values. In the CXObject’s primary key, in search functions and in all Load functions.

We can directly load an object from a compound key:

|  |
| --- |
| ...  SQLVariant myCompany(p\_company);  SQLVariant myLedger(p\_ledger);  SQLVariant myTransaction(p\_transID);  VariantSet key;  Key.push\_back(myCompany);  Key.push\_back(myLedger);  Key.push\_back(myTransaction);  Payment\* paymentToday = (Payment\*)session->Load(Hardware::ClassName(),key);  ... |

This looks akward, but in a real application, we already will have a loaded “VariantSet p\_key” somewhere around, and the load will only take one line of code.

### 4.1.5 Load of a set of objects from a single condition filter

In many cases we will want to look for objects, not by their primary key, but by some candidate key. A search form where we fill in one or more filter values is a perfect example of such a load. That’s why we can load objects by one (this paragraph) or by a set of filters (next paragraph). Filters can be defined through the “SQLFilter” object and used as the second parameter in a “Load” operation.

This is the form of a “Load” with one filter:

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

A few things to keep in mind:

* A “CXResultSet” is in fact a “std:vector<CXObject\*>” container. It gives you a bunch of pointers to objects as a result of the load from the datastore.
* A “SQLFilter” has in general an attribute, an operator and a value. For more information on filters, please refer to the chapter “Filters”.

We can specify a filter and then load a set of objects. For instance we could in the “HelloWorld” application load all the countries with more than 10 million inhabitants:

|  |
| --- |
| ...  SQLFilter filter("inhabitants",OP\_Greater,10000000)  CXResultSet set = session->Load(Country::ClassName(),&filter);  ... |

Once we loaded the objects, we can simply step through them with a for loop. Because we know the data class beforehand, we can simply cast the CXObject’s to the desired class, as in:

|  |
| --- |
| ...  for(auto& object : set)  {  Country\* country = reinterpret\_cast<Country\*>(object)  .. do something interessing with the country object ...  }  ... |

### 4.1.6 Load of a set of objects from multiple condition filters

An extension of loading objects through a filter, is loading a set of objects through a set of filters. This works exactly as loading through one (1) filter. But instead a “SQLFilterSet” is used.

**CXResultSet\* Load(CString p\_classname,SQLFilterSet& p\_filter);**

This makes it possible to provide multiple search conditions at once to the load operation. In fact: this overload of the “Load” operation is more in use then the “Load” from just one filter. We can e.g.:

|  |
| --- |
| ...  SQLFilter filter1("inhabitants",OP\_Greater,10000000);  SQLFilter filter2("continent", OP\_Equal, "Asia");  SQLFilterSet filters;  filters.push\_back(filter1);  filters.push\_back(filter2);  CXResultSet set = session->Load(Country::ClassName(),filters);  ... |

… load all the countries on the continent of “Asia” having more inhabitants than 10 million.

## 4.2 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.

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.

## 4.3 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);  ... |

## 4.4 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.

# 5. PERSISTENT CLASSES

In Hibernate ‘Persistent classes’ are referred to as those classes that consists of objects that reside in an external store and ‘do hibernate’ there when the application is not running. In general the objects are stored in one or more – related – database tables. In CX-Hibernate the external store can also be a filestore or an internet webservice server that is internally serviced by CX-Hibernate as well.

In order for CX-Hibernate to cooperate with these persistent classes, they must obey to a set of special rules in general. Some of these rules are found in other Hibernate products as well. Other rules are native to CX-Hibernate, for no other reason that the C++ language does not support meta data and a reflection mechanism like Java and C#. Advocates of the C++ language however do not recon that to be a drawback, but a winning point, as reflection is quite slow.

## 5.1 Rules

Classes must obey to the following rules:

1. Each class must be derived from the general “CXObject” class.

Reason for this rule, is the fact that a lot of general functionality is already in the CXObject base class, and deriving from this general ‘object’ type of class makes 90 % of reflection unnecessary. Creating, loading, tracking changes, inserting, updating and deleting all works through the general CXObject class interface.

1. Each class must have a basic constructor without any parameters.

The parameter-less constructor is used by Hibernate to create instances of your class. This works through a generalized object factory mechanism that is created at startup link time of your application.

1. Each class must either have no destructor, or a “virtual” destructor.

A lot of functionality in the CXObject class are overridable by your own class. So there exist already a good number of virtual methods. This creates the necessity of a virtual destructor. Any other destructor will not be called by the C++ language.

1. It is highly recommended to declare the “DECLARE\_CXO\_SERIALIZATION” macro in the interface declaration in the “<class>.h” file.

This macro defines a lot of virtual overrides for your class for the object factory, the generator mechanism and the serialization and de-serialization of information out of your database, or out of XML/SOAP messages. It’s an easy way of declaring everything at once without forgetting everything.

1. Each class must implement an “object factory”

The object factory is an small automated class does the “new” of a new object of the class. The object factories are put by the linker in a special loading segment and started by the “hibernate” static object at program startup time, before the “main” is called. As such the programmer is assured that the object factory can be called.

*Also: Please be very aware that only classes that define an object factory can be named in the “hibernate.cfg.xml” configuration file*

1. Each class must implement a database and/or a soap serializator and deserializator.

In order to assign fields from a database record to data member attributes of a class object, you must implement deserializators. See below under the “BEGIN\_DBS\_DESERIALIZE” and the “BEGIN\_XML\_DESERIALIZE” macros.

In order to bring data from class object members to the database you must implement serializators. See below under the “BEGIN\_DBS\_SERIALIZE” and “BEGIN\_XML\_SERIALIZE” macros.

1. Classes that have auto generating primary keys in the database must implement a “generator deserializator”

In order to accept the value of a database sequence or generator, the class must have a specialized de-serializator to upate it’s view of the business key with the new value of a generator after the insert operation in the database. Otherwise following updates and deletes on the object will not work.

## 5.2 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
* 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

## 5.3 Overridable virtual functionality

# 6. 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*

# 7. ASSOCIATION MAPPINGS

# 8. FILTERS

Filters & FilterSets

Operators

More than one value

AND and OR

# 9. TRANSACTIONS AND BATCHES

SQLTransactions

Mutation numbers

Mutation stacks on fields

Committing at the end

# 10. 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.

# 11. 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.

# 12. QUERY LANGUAGE AND NATIVE SQL

# 13. XML MAPPINGS

# 14. TOOLS

# APPENDIX 1: FUN WITH SQLVariant’s