U++ SQL

U++ SQL Basics

Schema description files (.sch) and sql script files (.sql)

Each database should be described in a schema description file, which is a file with extension ".sch" and . describes what tables you want to have in the database. U++ SQL packages will take the schema, create ".sql" script files from it (containing DDL statements like 'create table'...) and execute them against the database to realize your database table structure (you do not need to write sql statements, U++ will generate them for you.)

To add this file to your Upp package, make sure your package is selected in the package list (located at the top-left of TheIDE), right-click on the white-space in the file list (underneath the package list at bottom-left), and select "Insert package directory file(s)". Then give the new name of your file (eg. "MyDatabase.sch") and click "Open". The new file will show up in your file list.

Schema Update/Upgrade

The SQL packages also allow for database updates without losing your stored data.

Basic Description and Use

For this section, the example used will be oriented to PostgreSQL use. See the SQL example packages provided in the Upp examples for using MySQL and SQLite as well.

The Schema description file (.sch file)

In each schema description file, you describe the table and column layout of your database.

Note: In U++, we tend to use uppercase for all database names.

Postgresql Example ("person_db.sch"):

```
TABLE_ (PERSON)
    SERIAL_ (PERSON_ID) PRIMARY_KEY
    STRING_ (NAME, 25)
    DATE_  (BIRTH_DATE)
    INT_   (NUM_CHILDREN)
    DATE_  (DATE_ADDED) SQLDEFAULT(CURRENT_DATE)
END_TABLE

TABLE_ (EMPLOYEE)
    SERIAL_ (EMPLOYEE_ID) PRIMARY_KEY
    STRING_ (DEPARTMENT, 50)
    STRING_ (LOCATION, 50)
    DATE_  (START_DATE)
    BOOL_  (IS_SUPERVISOR)
    TIME_  (WORKDAY_START)
    TIME_  (WORKDAY_END)
    INT64  (PERSON_ID) REFERENCES(PERSON.PERSON_ID)
END_TABLE
```

In this schema, we have described a 'person' table and an 'employee' table, with theforeign key 1 to 1 relationship "an employee is a person".

The different types mentioned in this example map to SQL types. More information about types should be referenced by looking at the source code header files for the database type. In this example, all of the types referenced are found defined in the file "PostgreSQLSchema.h" from the "PostgreSQL" U++ package.

Each type declaration has 2 variants; one with an underscore "_" and one without. When an underscore is used, an `SqlId` object is automatically created for use as a variable in your source files. When not used, you must manually define the `SqlID` object in your source. Reference the SqlId objects section below for further explanation.

Note: if you use a name more than once, you should use an underscore ***only the first time*** you declare the name, otherwise you will get "already defined" compilation errors. This is shown in the above example where the column name "PERSON_ID" is used twice; there is an underscore only the first time it is used.

Source Files (for PostgreSQL example)

Header file includes/defines ("person.hpp"):

```
#include <PostgreSQL/PostgreSQL.h>
#define SCHEMADIALECT <PostgreSQL/PostgreSQLSchema.h>
#define MODEL <MyPackage/person_db.sch>
```

```
#include "Sql/sch_header.h"
```

Source file includes ("person.cpp"):

```
#include "person.hpp"
#include <Sql/sch_schema.h>
#include <Sql/sch_source.h>
```

Session objects:

```
PostgreSQLSession m_session;
```

The session object is used to control the connection and session information. Each database dialect will have its own session object to use.

Database connection using session:

```
bool good_conn = m_session.Open("host=localhost dbname=persons user=user1 password=pass1")
```

The `Open()` function returns a true or false value depending on success of connecting to database.

SqlId objects:

SqlId objects aid the formation of sql statements by mapping database field/column names to local variables.

```
SqlId all("*");
SqlId person_name("NAME");
```

We will now be able to use "all" and "person_name" in our sql CRUDstatements in our code.

As mentioned previously, all of the declarations in our schema file that end in an underscore will automatically be declared as SqlId variables we can access in our source code.

Example use of SqlId variables:

```
sql * Insert(PERSON)(NAME, "John Smith") (BIRTH_DATE, Date(1980,8,20)) (NUM_CHILDREN, 1)
```

The variables `PERSON, NAME, BIRTH_DATE, NUM_CHILDREN`were available to us even though we didn't define them in our source. We could have also used the variable `person_name` instead of `NAME` as we defined it ourselves.

Sql objects

Sql objects are used for CRUD operations on the database; they operate on a session.

```
Sql sql(m_session); //define Sql object to act on Session object m_session.
```

Select example:

```
sql * Select(all).From(PERSON).Where(person_name == "John Smith");
```

Note: Here we can use "all" because we defined it as an `SqlId` variable above (same goes for "person_name").

Exceptions vs Checking for errors.

There 2 multiple ways to make sql statements.

1. Manual error checking.

   Manual error checking uses the **asterisk ("*") operator** when writing SQL statements.

```
sql * Select(all).From(PERSON).Where(NAME == "John Smith");
if(sql.IsError()){
    Cout() << m_session.GetErrorCodeString() << "\n";
}
```

2. Exception handling.

   Specify exception handling by using the **ampersand ("&") operator** when writing SQL statements.

```
try{
    sql & Select(all).From(PERSON).Where(NAME == "John Smith");
}catch(SqlExc& err){
    Cout() << err << "\n";
    // Or we can get the error from the session too...
    Cout() << m_session.GetErrorCodeString() << "\n";
}
```

*Remember: SqlExc is a subclass of Exc, which is a subclass of String, so it can be used as a string to get its error.