D&D CEAMS – SQLite Documentation

**This is not a complete document! If it seems incomplete, that’s because it is!**

**I will continue to update this document and indicate here when it is complete.**

**ABOUT THIS DOCUMENT**

This document is intended for use of Team Strikeforce developers in developing the project ‘D&D CEAMS’. The purpose of this document is to explain how I (Steven Rohr) have created the database system and how other developers should use it.

**ABOUT SQLITE**

This system uses SQLite and has been developed in Visual Studio 2012. You can obtain the required header file and implementation used to interface with the database from the following URL: <http://www.sqlite.org/download.html>

You can also obtain the required SQLite files as well as my own interaction header/implementation from the Team Strikeforce Google docs where you likely found this file.

We are using SQLite as it is open source and free to use for all purposes. It is a relational, offline database language.

SQLite’s documentation is available here: <http://www.sqlite.org/docs.html>

**WHAT YOU (DEVELOPER) NEED TO KNOW**

I (Steven Rohr) will act as the primary caretaker of this database. At the time of writing this document, I am making the database and filling it. Ideally, any issues with the database will be resolved by myself, however, in the event that someone else must resolve some issue, I will do my best to outline here how to do so.

Files:

Besides all the .h and .cpp files I have included for the different classes we use in this project (Race, Character, Weapon, etc.), there are also four files needed for any Database interaction:

sqlite3.h - Header file included in SQLite.

sqlite3.c - Implementation of the above header.

DBIO.h - My Database Interaction header.

DBIO.cpp - Implementation of the above header.

Without the first two files, it is impossible for you as a developer to access the Database, and without the bottom two files, it is much more difficult. The bottom two files, DBIO.h and DBIO.cpp, contain functions that I have designed and written to allow you to easily interact with the database. These functions include:

* bool AddToCEAMS(object)
  + Adds an object to the database.
  + Call like “AddToCEAMS(myCharacter);”
* bool RemoveFromCEAMS(object)
  + Removes the specified object from the database.
  + Call like “RemoveFromCEAMS(myCharacter);”
* vector<Class Name>\* Load<Class Name, plural>()
  + Returns a pointer to a vector of all elements found in the database of that class type. Returns NULL if nothing found.
  + Call like “vector<Spell> \*ptr = LoadSpells();”
* void Complete\_\_\_\_\_\_\_\_ (vector<\_\_\_\_\_\_\_\_\_\_>)
  + There are several of these, as detailed below. They are used to flesh out data for certain fields which will not be fleshed out due to restrictions of SQL.

|  |  |
| --- | --- |
| Data Class Type | Function |
| Character | CompleteItems(vector<Item>) |
|  | CompleteFeats(vector<Fear>) |
|  | CompleteSkills(vector<Skill>) |
|  | CompleteSpells(vector<Spell>) |
|  | CompleteClasses(vector<Class>) |
| Class | CompleteSpellsCanLearn(vector<Spell>) |
| Feat | CompletePrereqs(vector<Feat>) |

* + You must pass in a vector containing all the spells/feats/skills/etc, which I assume you will obtain from the corresponding “Load” function.
  + This works by using string’s operator == to compare the name of the field given by the database and comparing it to the data in the vector passed in, then fleshing out the fields by copying data from the passed in vector.

All Boolean functions above return true on success, false on failure (will also print error to standard out if unsuccessful connection).

All functions are overloaded unless otherwise noted, and work for the following object types: Class, Race, Character, Feat, Skill, Item, Weapon, Spell (unless otherwise noted).

We are using string::stoi, and thus, we are using C++11, please remember this if you should for some reason attempt to run my code on Linprog.

Due to how relational databases are designed, upon loading characters the fields denoting the spells, classes, items, feats which the character has will ONLY contain the names of such fields. There will be member functions in the Character class which accept a vector of spells/classes/items/feats and fill out the respective Character vectors with all data.

This is a flaw in relational databases and I cannot come up with a better way to do it. Sorry in advance. (Though it should just be a few function calls in the program loading section, I doubt it’ll be too bad.)