SimpleSQL

SQLite manager for Unity3D

echo17.com

Table of Contents

1.

Overview

SimpleSQL is a plugin for Unity3D that allows you to easily query or modify data in a database.

Its primary advantage over other plugins is that you don't have to concern yourself with the specialized dll's necessary for connecting to a database. Also, it simplifies publishing to mobile devices by doing all the work of moving data to your application's working directory for you.

Some examples of how you can use a database in a game or project include (but are not limited to):

- Keep track of player stats
- Store inventory information, such as items' weight, cost, damage, armor, etc.
- Keep track of progress and visited areas in a game
- Store maps of your world with interlinking connections
- Store dialog for an RPG with quick lookups based on person and place
- And many more!

SimpleSQL uses SQLite as the database format for its data storage. You can easily create and modify databases programmatically or with third party tools. An example of a tool that can be used is:

- DB Browser for SQLite: https://sqlitebrowser.org/

These tools are not supported by echo17, so you should check with the above providers if you have questions regarding their products. Training on these products is outside of the scope of echo17 and this

documentation.

2.

Workflow

SimpleSQL only works with databases in your application's working directory so the databases in your project will never be updated at runtime.

You can use this workflow to your advantage by making your project database a template that can create one or many working databases.

If you do not have a working database in existence then SimpleSQL will copy the project database to your working directory. If you do have a working database in existence, then SimpleSQL will only copy the project database over to the working directory if you tell it to.

If you are using data statically (not making changes at runtime), then you can overwrite your working database(s) without any consequences. This keeps your working databases in sync with what you have set up in your project.

If you are using data dynamically (making changes at runtime), then you **DO NOT** want to overwrite your working database since this would wipe out any changes since the last time the application was run. If you need to make changes to your working database(s)' structure or static data, then you will need to follow an upgrade path as outlined in Upgrading Databases.

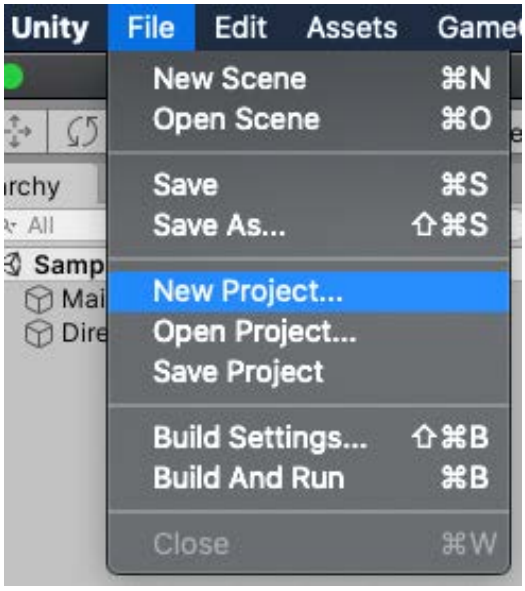
You may want to divide your data into multiple databases. For instance, you could keep all your static data such as room layout, maps, unit properties, etc. in one database and all your dynamic data such as player stats and achievements in another database. This allows you to overwrite the static database without worrying about wiping out dynamic data.

3.

Setting Up Your Project

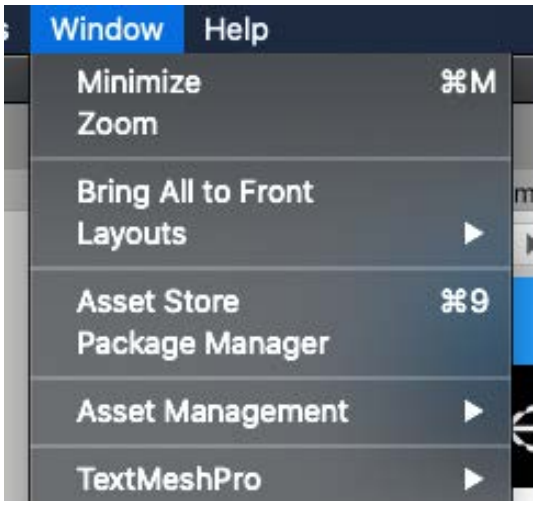
To use SimpleSQL in your project, first create a new project in Unity. It's best to create a new project for any version changes so you can have a history of the plugin in case something goes wrong and you want to revert to an earlier version.

Figure 3-1 New Project



You can then import the SimpleSQL plugin from the asset store.

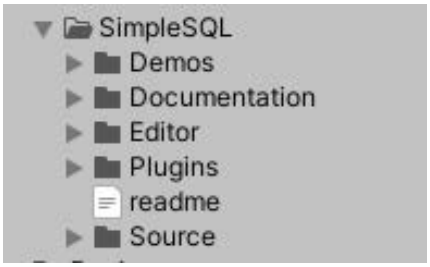
Figure 3-2 Asset Store



If you have already purchased SimpleSQL, you will either be shown a download or import link, depending on if you have the latest version.

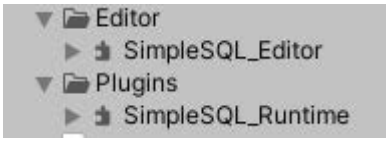
This will download the entire SimpleSQL package, demo and all, to your new project.

Figure 3-5 Editor Plugin Folders



Your final game project layout should have the two dll's in their respective directories: SimpleSQL_Editor.dll in the Editor directory and SimpleSQL_Runtime.dll in the Plugins directory.

Figure 3-6 Dlls



4.

Creating a Database

Before you can use SimpleSQL in Unity, you will first need to create a database. There are a couple of ways to do this:

1. Create an empty database in Unity using SimpleSQL, then add tables and data programmatically at runtime or modify the database with a third party tool.
2. Create a database in a third party tool such as the DB Browser for SQLite.

Databases for SimpleSQL have to have the extension ".bytes". You can tell SimpleSQL to change the file name after it has copied the file to your application's working directory, but Unity requires all non-standard assets (such as a SQLite database) that will be streamed to have this extension within your project.

4.1 Create an Empty Database using SimpleSQL in Unity

There are a few ways to create an empty database using SimpleSQL

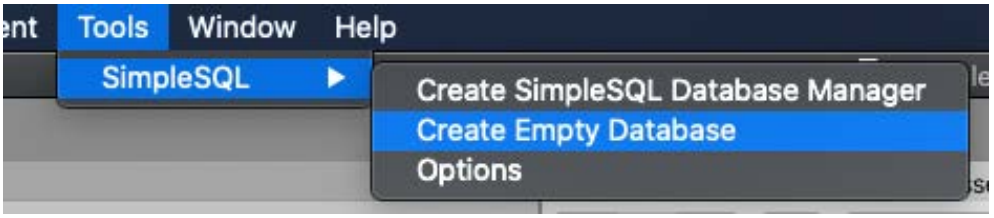
1. Create from the unity menu
2. Create from the project menu

3. Create from the asset menu

Create from the Unity Menu

To create a database from the Unity menu, select Tools > SimpleSQL > Create Empty Database

Figure 4-1 Create Database From Unity Menu



This will add a new empty database file to the currently selected folder (or the folder of the currently selected object). SimpleSQL automatically assigns the ".bytes" extension to your new file, which is not visible from within Unity, but can be seen in your OS's folder browser.

Figure 4-2 New Empty Database

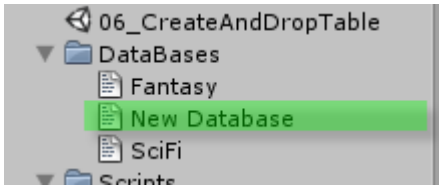
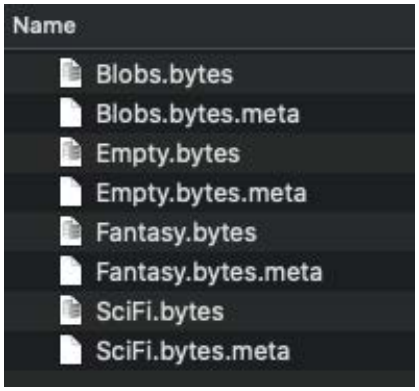


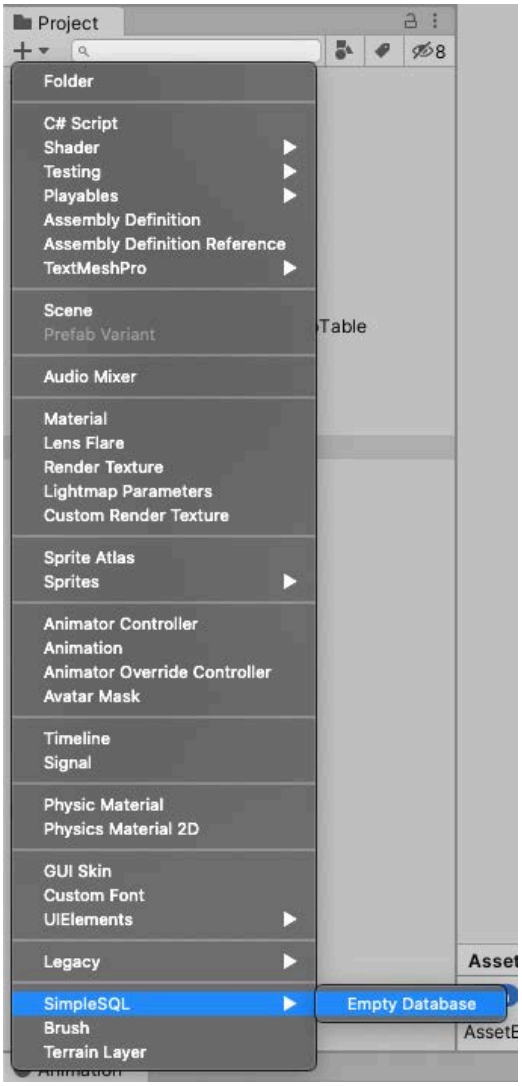
Figure 4-3 Bytes Files



Create From the Project Menu

You can also create a new database from the Project window menu by clicking on the Create dropdown and going to SimpleSQL > Empty Database.

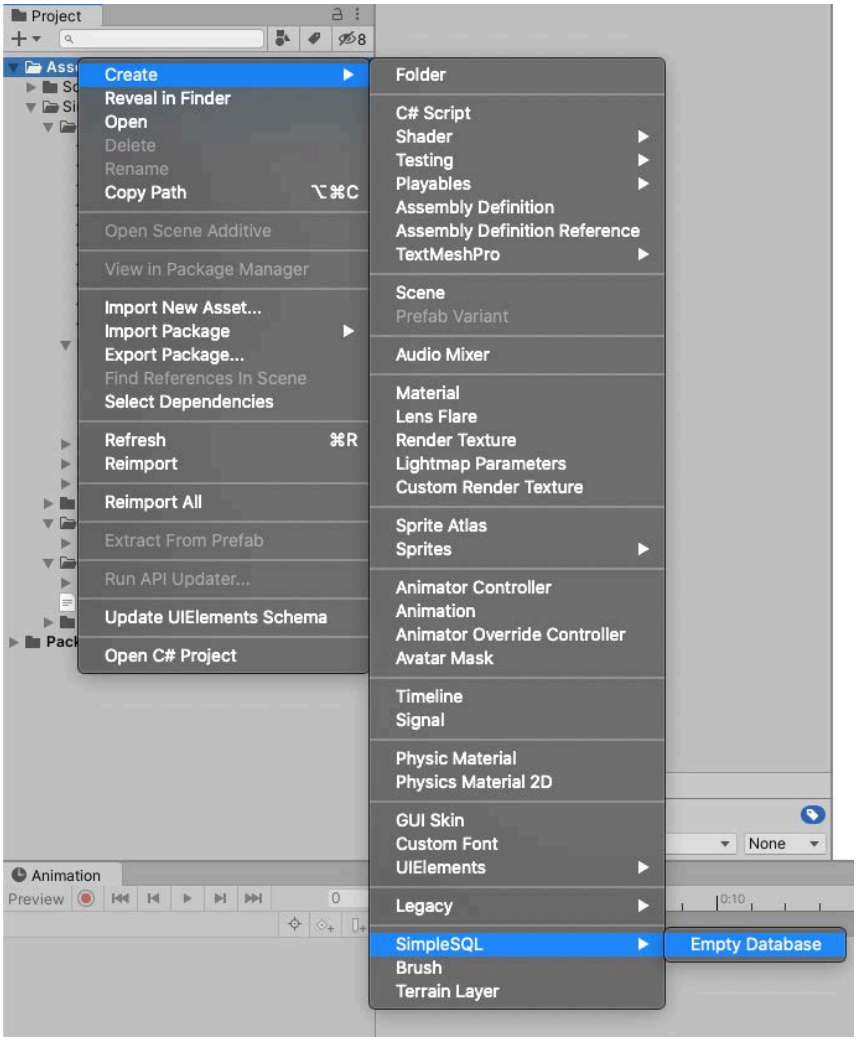
Figure 4-4 Create Database From Project Menu



Create From the Asset Menu

You can also create a new database by right-clicking in your project window and going to Create > SimpleSQL > Empty Database. This allows you to specify the exact sub-folder of the database without having to move it there.

Figure 4-5 Create Database From Asset Menu



4.2 Create a Database with a Third Party Tool

It is beyond the scope of this document on how you can create a database using a third party tool such as DB Browser for SQLite. The key thing to keep in mind here is make sure your database has the extension of ".bytes".

You can change the extension of the database with the SimpleSQLManager discussed later by changing the database's working name. The ".bytes" extension is necessary for your project.

4.3 Making Changes to Your Database

You can make changes to your database through third party tools or through code programmatically at runtime. If you are setting up a database structure or data that will persist, it is probably better to set up your changes in a third party tool. If your database structure will change dynamically then you may want to make your changes in code.

If your database's data will be updated then you will need to make any structure changes in code through an upgrade path. Please see Upgrading Databases for more information.

5.

Creating a Database Manager

Once you have your database files ready to go, you are now ready to use the data in a scene.

Each database that you will reference in a scene will need a SimpleSQLManager script to interact with it. You can attach this script to any object, but it is usually cleaner to have this on its own GameObject.

You can have multiple managers in a single scene, each accessing a database. You may want to divide out your static data into a separate database from your dynamic data for instance. This way you can overwrite your static database with any new settings or changes without worrying about overwriting dynamic data.

5.1 Creating a SimpleSQLManager

This will show you how to set up a SimpleSQLManager that will use only the ORM data class structure. If you are interested in including System.Data structures, see Create a SimpleSQLManager with System.Data.

There are a few ways to create a SimpleSQLManager in a scene:

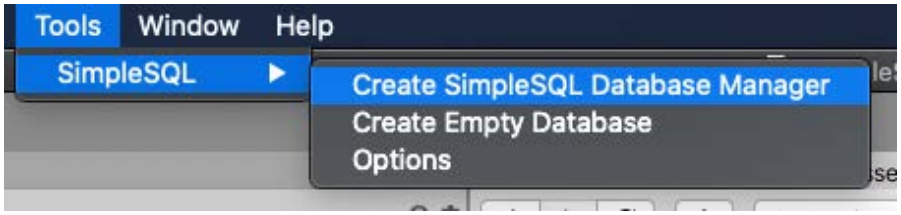
1. Create through the unity menu

2. Create through the Hierarchy menu
3. Attach to an existing GameObject

Create Through the Unity Menu

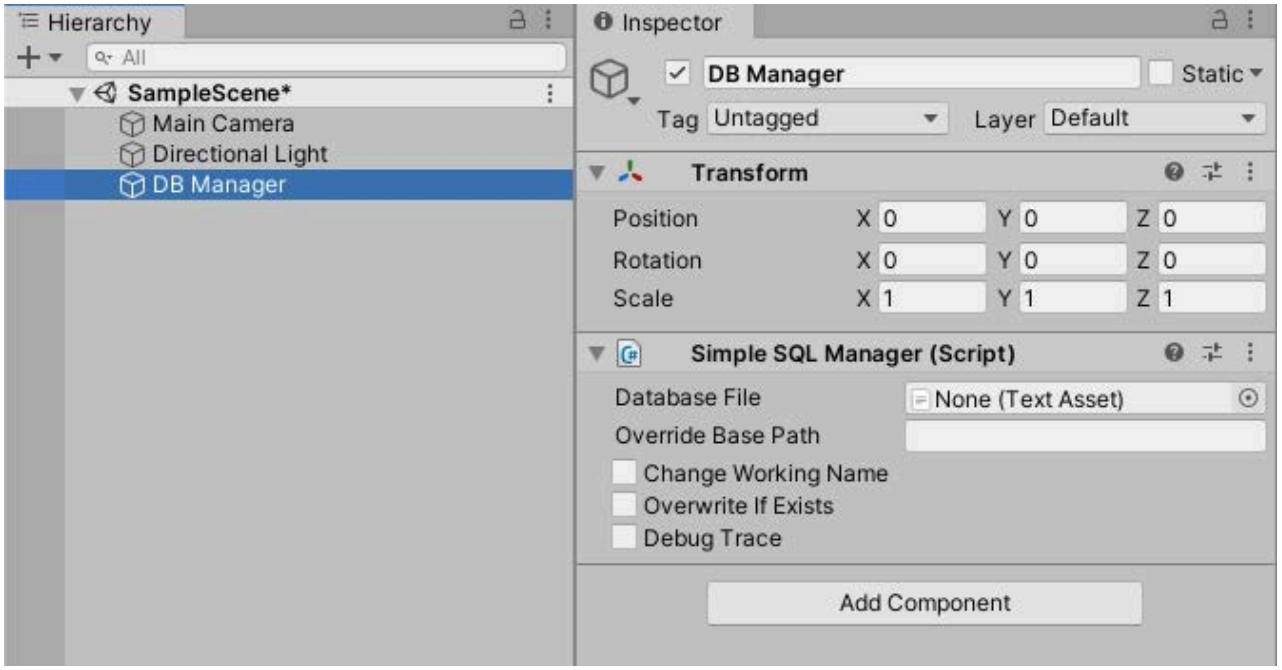
To create a new SimpleSQLManager through the Unity menu, click on the Tools > SimpleSQL > Create SimpleSQL Database Manager.

Figure 5-1 Create SimpleSQLManager Through Unity Menu



This creates a new DB Manager object in the scene with the SimpleSQLManager script attached.

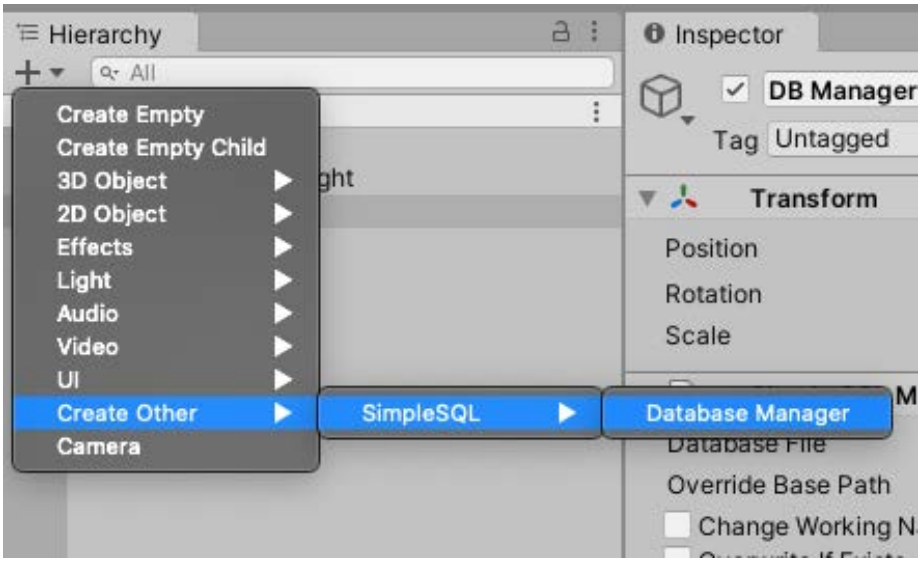
Figure 5-2 New Database Manager



Create Through the Hierarchy Menu

To create a DB Manager through the Hierarchy menu, Click the Hierarchy window's Create button and select SimpleSQL > Database Manager.

Figure 5-3 Create SimpleSQLManager Through Hierarchy Menu



Attach to an Existing GameObject

You can attach the SimpleSQLManager script to an existing GameObject by:

1. Dragging the script to the GameObject
2. Attaching the script through the Unity menu

Dragging the Script to the GameObject

1. Select the GameObject you want the script to be attached to.
2. Expand the SimpleSQL_Runtime dll in your project window. You should see the SimpleSQLManager script located there.
3. Drag this script onto the currently selected GameObject.

Figure 5-4 Select GameObject

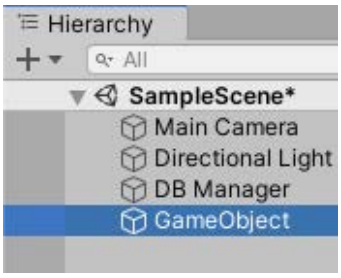
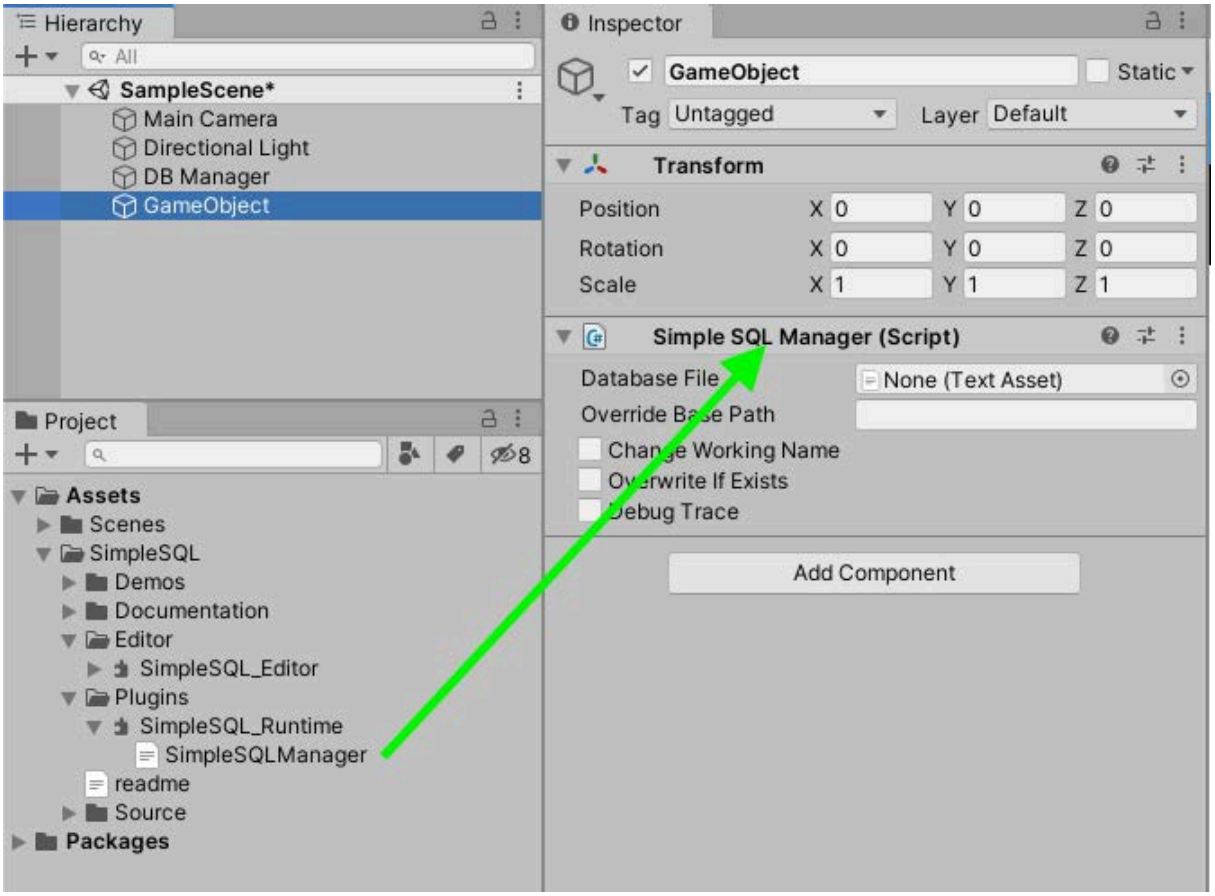


Figure 5-5 Drag Script to GameObject



Attaching the Script Through the Unity Menu

You can attach the script through the Unity menu by:

1. Select the GameObject you want the script to be attached to.
2. In the Unity menu, go to Component > Scripts > Simple SQLManager.

Figure 5-6 Select GameObject

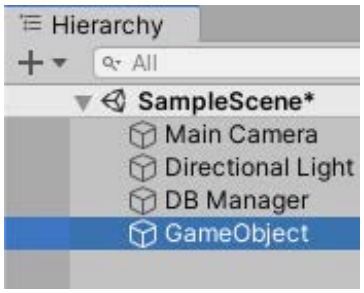
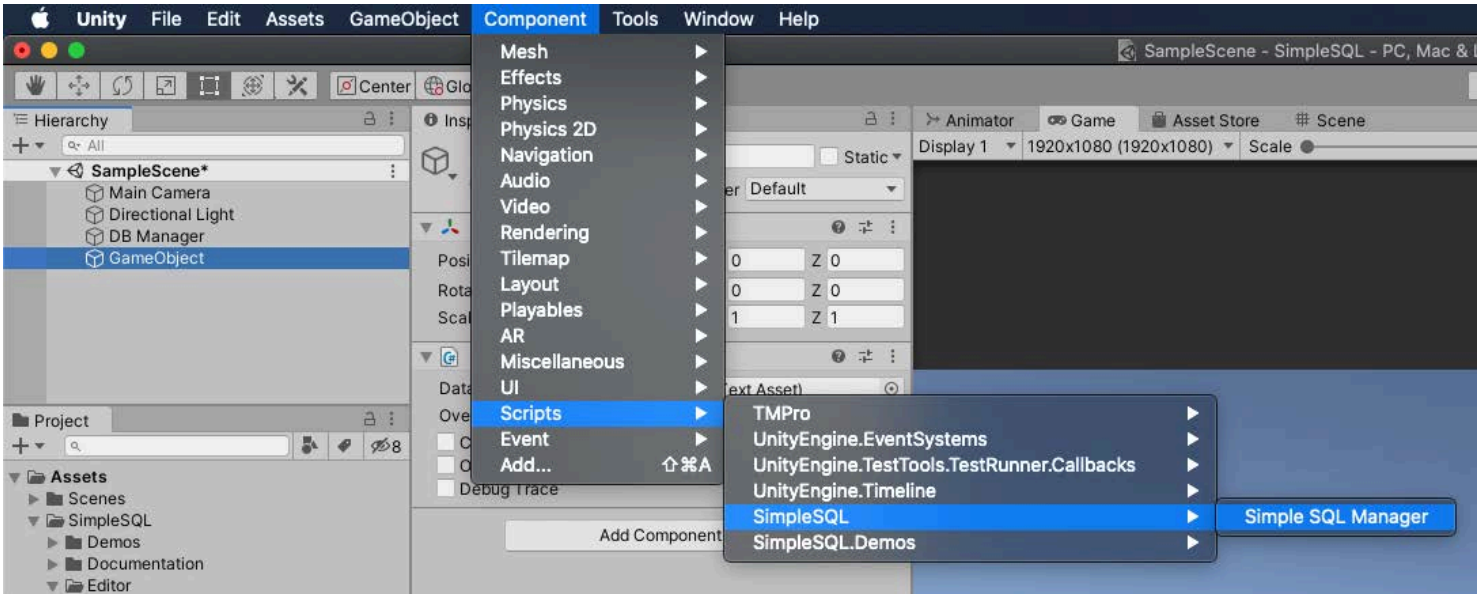


Figure 5-7 Attach SimpleSQLManager Script Through Menu



5.2 Create a SimpleSQLManager with System.Data

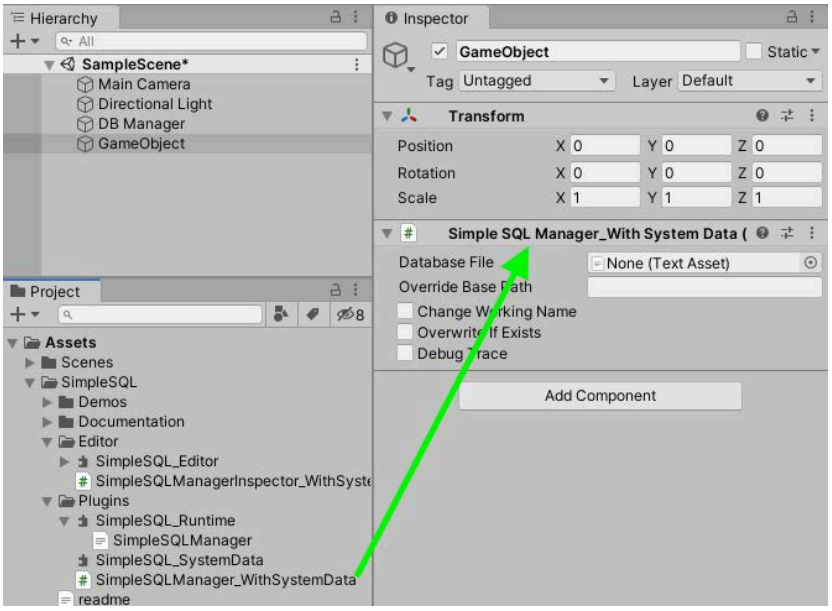
If you want to use System.Data with your SimpleSQLManager, then you'll need to set up your project to accommodate. See Optimize Data Library on how to set your project to use System.Data.

Figure 5-8a Set to use System Data

You can still use the ORM classes alongside the System.Data structures.

Once you have your project set to use the System.Data library, you can drag the SimpleSQLManager_WithSystemData script from the Plugins folder to your gameobject.

Figure 5-8b SimpleSQLManager_WithSystemData



5.3 Manager Settings

Once you have your manager set up in a GameObject, you can specify its settings:

- Database File - This is the SQLite file that will be used by the manager.
- Override Base Path - This is the base path your project will use to store the database. Note that on mobile devices, this path is limited to the app's sandbox.
- Change Working Name - Using this value lets you change the file's working name from the project's name to whatever you wish.
- Overwrite if Exists - This will overwrite the database stored in your application's working directory with the database in your project. Use this with extreme caution as it can wipe out any changes made by your application.
- Debug Trace - This will output the SQL statements used by the manager to the debug console window for better debugging in the editor.

Database File

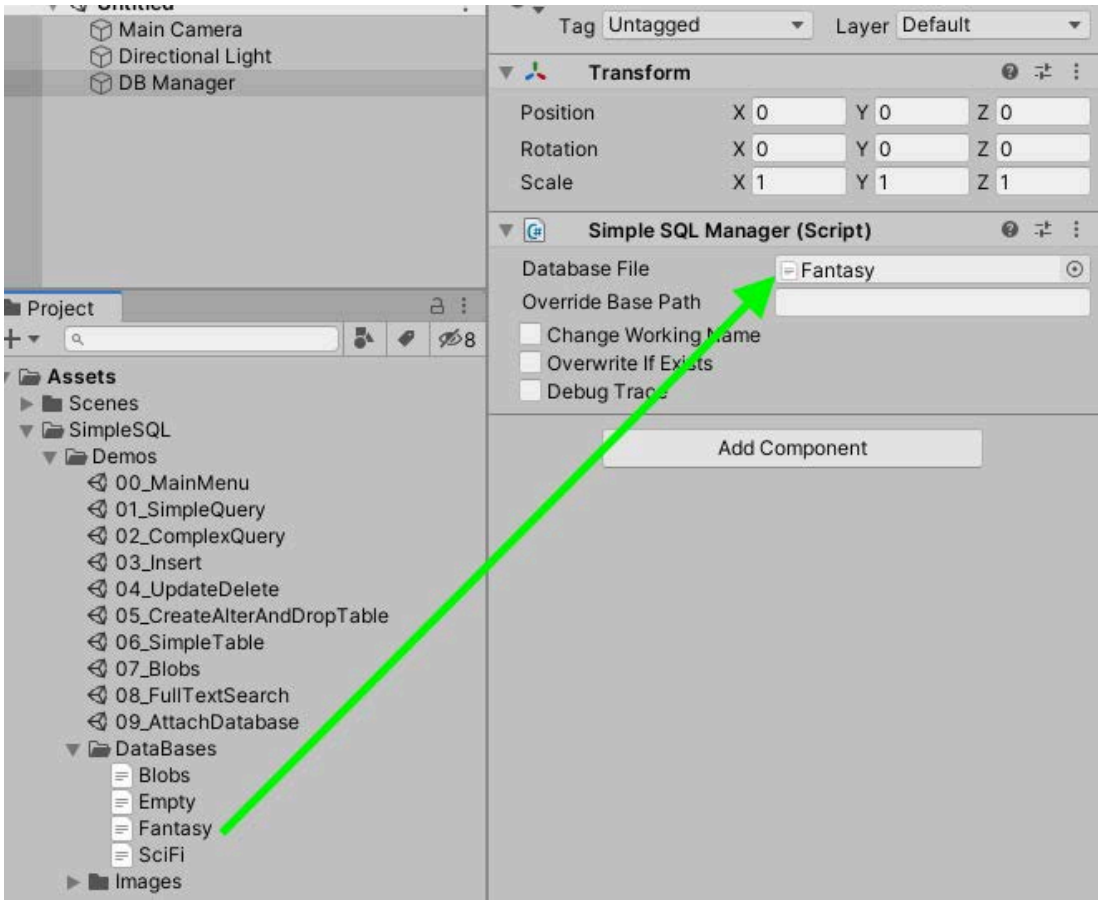
To assign a database file to the manager, you can do one of the following:

1. Drag the database asset to the the database file field of the inspector
2. Click on the circle to the right of the database file field of the inspector and browse for your database

You can store your databases in whatever folder structure that you choose as long as the file has the extension ".bytes".

Drag Database Asset to the Inspector

Figure 5-9 Database File



Select the Database from the Object Browser

Figure 5-10 Select Database File Browser

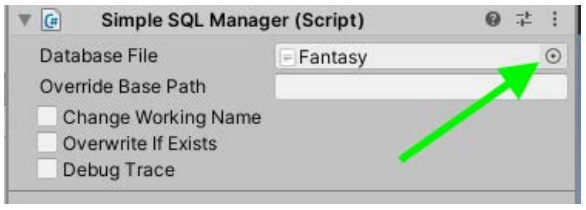
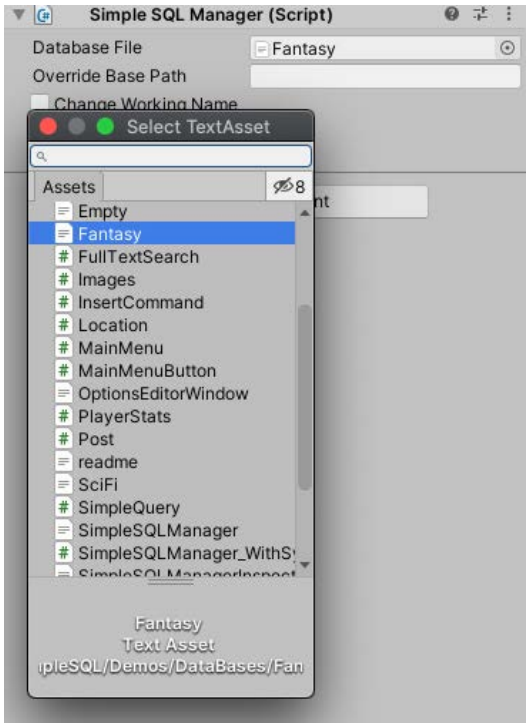


Figure 5-11 Select Database from Browser



Change Working Name

You may wish to have a different name for your working database other than the project database's name. This can be useful if you are using the same project database template to create multiple working directories or if you just prefer your database to have a different extension, perhaps. You can change the working database's name by checking the Change Working Name toggle and filling in the name field.

Figure 5-12 Change Extension

If you don't toggle the Change Working Name property on, then your database will be copied over with the same name as the project database with the ".bytes" extension.

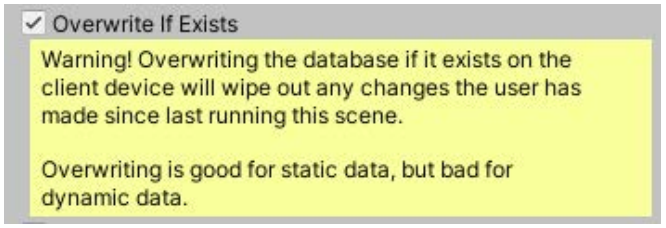
If you do toggle the Change Working Name property on, then you must supply a name or you will get an error.

Overwrite if Exists

SimpleSQL will first copy your database from the project to your application's working directory if it does not already exist. All interaction with the database is done from the working directory. The project's database will remain untouched.

If you want to copy the database as it is in your project to your application's working directory, then you will want to toggle this on.

Figure 5-13 Overwrite If Exists



You will be warned that checking this property on will overwrite the database in your working directory.

Static database are good for having the Overwrite If Exists property on. Since they do not change at runtime, there is no harm overwriting the working database.

Dynamic databases should **NOT** have the Overwrite If Exists property on since this will wipe out any changes made during runtime with the database from the project.

See Workflow for an explanation of how SimpleSQL uses databases.

If you need to make changes to a database's structure in the working directory, but you don't want to overwrite the database and wipe out any changes in data, then you should follow an upgrade path explained in Upgrading Databases.

Debug Trace

This setting is only used in the editor and has no effect on your runtime code within your target device. Setting this value will allow you to see the SQL statements that pass through the manager. This can help you debug your SQL by showing the statements in the debug console.

Figure 5-14 Debug Trace

6.

Data Structure

SimpleSQL allows you to optionally choose to use .NET data structures such as DataTable, DataRow, and DataView or use a lighter-weight class-based ORM (Object Relational Mapping).

Some advantages of using the ORM method:

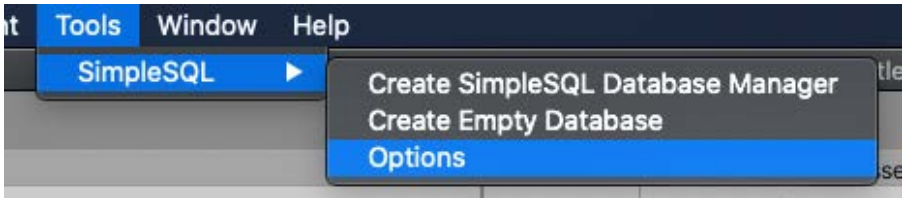
- Much smaller memory footprint. The System.Data dll that is required for the standard .NET data structures takes up a full megabyte of storage. In addition to this, you also have to use the full .NET Unity library for this to work properly, further bloating your application.
- Strongly-typed field casting that is set up once. You don't need to constantly cast the data you retrieve from your database to the proper types, cutting down on the risk of typos and bugs.
- Simple calls to update and modify your database with little need for SQL syntax. Queries and complex modifications can still be called with SQL statements, if you prefer.

SimpleSQL makes use of the Generic library to be able to store your arrays without having to cast them each time they are referenced. Be sure your scripts have a reference to the `System.Collections.Generic` library.

6.1 Using .NET's System.Data

If you decide you want to use .NET's System.Data library, then you can set this in the Options by going to the Unity menu SimpleSQL > Options.

Figure 6-1 Options



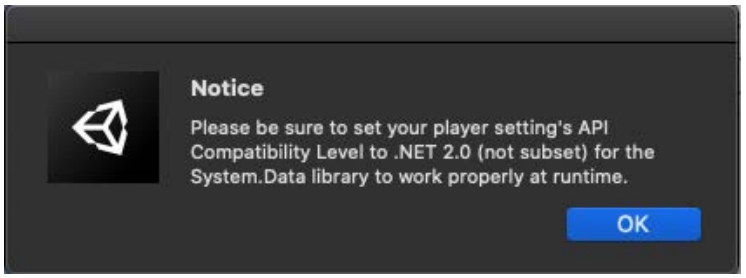
From there you can choose the Data Library Optimization setting. If you wish to use System.Data, you'll need to select the appropriate option:

Figure 6-2 Use System Data

When you choose to use the System.Data library, you will be alerted to the fact that you will need to set your project's player setting's API Compatibility Level to the full .NET library (not just the subset).

Newer versions of Unity come with System.Data as part of the package. You can safely delete the System.Data dll that is added to the Plugins folder in this case.

Figure 6-3 Use Full .NET



Using System.Data requires the full .NET library in your player settings. This will increase your final package size.

You can still use the ORM classes alongside the System.Data structures.

6.2 Using the ORM

If you decide to use the ORM, then you can turn off the System.Data library to streamline your application. Note that you can leave the System.Data library in your application without harm, it will just make the final package larger.

Figure 6-4 No System.Data

You will be asked to confirm removing the System.Data library and warned that doing so may cause errors if you have scripts or gameobjects in your scene referencing this library. Be sure you remove all these references before turning off System.Data to avoid errors.

Figure 6-5 Confirm System.Data Removal

Sample Classes

Below is an example of a simple class used to store weapon information in your game. This structure is designed around the results of a query, so it actually has more fields than the Weapon table in the database. You can find this class along with the corresponding database in the demo project included with SimpleSQL.

```
using SimpleSQL;

public class Weapon
{
    // The WeaponID field is set as the primary key in the SQLite
database,
    // so we reflect that here with the PrimaryKey attribute
    [PrimaryKey]
```

```
    public int WeaponID { get; set; }

    public string WeaponName { get; set; }

    public float Damage { get; set; }

    public float Cost { get; set; }

    public float Weight { get; set; }

    public int WeaponTypeID { get; set; }

    public string WeaponTypeDescription { get; set; }
}
```

The actual Weapon table in the database uses all these fields except for the WeaponTypeDescription field which is pulled in by using a JOIN in a query.

Attributes

You can specify attributes for your fields that will allow you to communicate traits to your database. In the above example, the WeaponID field has a [PrimaryKey] attribute added that signifies that this field is a unique key field in the data table. Other attributes that you can add to your fields include:

- PrimaryKey - Table's unique key field
- Indexed - Table is indexed based on this field
- NotNull - This field cannot be null
- Default(value) - If no value is provided when inserting, then the default value set here will be used
- MaxLength(value) - The maximum length of a string field

Here is an example data structure class that uses multiple attributes on several fields:

```
using SimpleSQL;

public class StarShip
{
    // StarShipID is the primary key, which automatically gets the
NotNull attribute
    [PrimaryKey]
    public int StarShipID { get; set; }

    // The starship name will have an index created in the database.
    // It's max length is set to 60 characters.
    // The name cannot be null.
    [Indexed, MaxLength(60), NotNull]
    public string StarShipName { get; set; }

    // The home planet name's maximum length is set to 100
characters.
    // The default value is set to Earth
    [MaxLength(100), Default("Earth")]
    public string HomePlanet { get; set; }

    // The range cannot be null.
    [NotNull]
    public float Range { get; set; }

    // The armor's default value is set to 120
    [Default(120.0f)]
    public float Armor { get; set; }

    // Firepower has no restrictions
    public float Firepower { get; set; }
}
```

In the next chapter, you will see how these classes can be used in quick and simple retrieval of data from your database.

7.

Queries

The most common task of a database is to quickly retrieve data. SimpleSQL passes a SQL statement to the database and returns a list of data stored in the format of your custom classes that you set up.

7.1 Query Examples

Most of these samples can be found in the demo that comes with the SimpleSQL plugin so only the relevant information will be highlighted. Please refer to these demos for complete implementations.

Retrieve All Data from a Table

This sample shows how easy it is to retrieve data from the Weapon table of your database and store the results in a generic list of your Weapon class.

```
public SimpleSQL.SimpleSQLManager dbManager;

void Start ()
{
```

```
 string sql = "SELECT * FROM Weapon";
 List<Weapon> weapons = dbManager.Query<Weapon>(sql);
}
```

Note that somewhere in your script you will need to set a public reference to the SimpleSQLManager object in your scene that controls the database you are wishing to query.

From now on we will leave out the lines that show the reference to the manager object since it is implied that one is needed.

Iterating the Results

Once you have your data stored, you can access it by looping through the list of results.

```
foreach (Weapon weapon in weapons)
{
    Debug.Log(weapon.WeaponName + " " + weapon.Damage);
}
```

Notice the simplicity here as compared to accessing the data in a .NET DataTable where you would need to cast each field to its appropriate type before using. Predefining a class once with types simplifies the amount of work required for every task afterward.

Retrieve Data from a Table Join

```
string sql = "SELECT " +
    "W.WeaponID, " +
```

```
    "W.WeaponName, " +
    "T.Description AS WeaponTypeDescription " +
    "FROM Weapon W " +
    "JOIN WeaponType T " +
    "ON W.WeaponTypeID = T.WeaponTypeID ";

List<Weapon> weapons = dbManager.Query<Weapon>(sql);
```

If you look at the Weapon class where we are storing the results of the query, you'll notice that it has a field for the WeaponTypeDescription even though the Weapon table does not carry this field. We put this field in the class so that we can store the joined value.

You do not need to fill every field of a class. In the above example only three of the Weapon class' fields are being populated with data.

Using Linq to Retrieve a Table's Data

You can quickly retrieve all the data in a table using Linq without writing any SQL syntax. Be sure you have a reference to the `System.Linq` library in your scripts if you choose to use this method.

```
List<Weapon> weapons =
    new List<Weapon> (from w dbManager.Table<Weapon> ()
    select w);
```

You can also easily filter down the results by using a where clause with Linq.

```
List<Weapon> weapons =
    new List<Weapon> (from w in Table<Weapon> ()
            where w.WeaponName == "Sword"
            select w).FirstOrDefault ();
```

7.2 System.Data Query Examples

Retrieve data from a table and store in a DataTable

This example shows how you can retrieve data from a SQLite table and store it in a DataTable. You can then process the table by iterating over the rows, filter using a DataView, or anything else the System.Data library allows.

```
public SimpleSQLManager_WithSystemData manager;
    public GUIText text;

    void Start ()
    {
        int row = 0;

        text.text = "";

        DataTable dt = manager.Query("SELECT " +
                                        "W.WeaponID, " +
                                        "W.WeaponName, " +
                                        "W.Damage, " +
                                        "W.Cost, " +
                                        "W.Weight, " +
                                        "W.WeaponTypeID, " +
                                        "T.Description AS
WeaponTypeDescription " +
                                    "FROM " +
                                        "Weapon W " +
                                        "JOIN WeaponType T " +
                                            "ON W.WeaponTypeID =
```

```
T.WeaponTypeID " +
                                    "ORDER BY " +
                                        "W.WeaponID ");
        foreach (DataRow dr in dt.Rows)
        {
            text.text += "Row: " + row.ToString() + " ";
            for (int c=0; c<dt.Columns.Count; c++)
            {
                text.text += dt.Columns[c].ColumnName + "=" +
dr[c].ToString() + " ";
            }
            text.text += "\n";

            row++;
        }
    }
```

One advantage to querying your data using this method is that you do not need to know the fields or their types beforehand. The DataTable stores each cell generically as an object type which will need to be cast when using the data later.

8.

Inserting Records

To get data into your database, you use insert commands.

8.1 Insert Record Examples

Insert with SQL Statement

You can insert into a database using SQL statements. You can also bind parameters using the ? value in your sql statement. To insert, you call the Execute function in your manager.

```
string sql = "INSERT INTO PlayerStats " +
    "(PlayerName, TotalKills, Points) " +
    "VALUES (?, ?, ?)";

dbManager.Execute(sql, "New Player", 3, 50000);
```

Note that we had three ? parameters and passed three values in our execute command. The number of parameters must match the number of values.

You do not need to use parameters at all if you prefer. You could just as easily built up a SQL string using concatenation.

Insert with Class Definition

You can also quickly insert into a database using the class definition of your table. We first create an instance of our class definition, fill it in with data, and pass the instance to our manager using Insert.

```
PlayerStats playerStats = new PlayerStats { PlayerName = "New
Player", TotalKills = 3, Points = 50000};

dbManager.Insert(playerStats);
```

For this method to work, your class definition must match the table definition with no extra fields. For example, the Weapon class could not be used like this because it has a definition of a field populated by a join with another table (WeaponTypeDescription). You may need to create separate classes for your table definitions and your query results.

9.

Updating Records

To change existing data in your database, you use update commands.

9.1 Update Record Examples

Update with SQL Statement

```
string sql = "UPDATE PlayerStats " +
    "SET PlayerName = ?, " +
    "TotalKills = ?, " +
    "Points = ? " +
    "WHERE " +
    "PlayerID = ?";

dbManager.Execute(sql, "Updated Player Name", 55, 120321, 2);
```

Note that you can use parameter binding with the ? in your SQL statement to simplify and reuse your query.

Update with Class Definition

```
PlayerStats playerStats = new PlayerStats { PlayerID = 2, PlayerName
= "Updated Player Name", TotalKills = 55, Points = 120321};

dbManager.UpdateTable(playerStats);
```

To call the UpdateTable function in your manager, you set all the values of your class instance, including the key field. The UpdateTable function will use the key field to look up the record. For this to work, you must specify the primary key in your class definition.

10.

Deleting Records

To remove records from your database, you will call delete commands.

10.1 Delete Record Examples

Delete With SQL Statement

```
string sql = "DELETE FROM PlayerStats WHERE PlayerID = ?";

dbManager.Execute(sql, 2);
```

Note that you can use parameter binding with the ? to simplify and reuse your SQL statements.

Delete With Class Definition

```
PlayerStats playerStats = new PlayerStats { PlayerID = 2};

dbManager.Delete<PlayerStats>(playerStats);
```

To delete with the class definition, you set up your instantiated class object with the primary key set. Then call Delete with the object, casting to the appropriate class structure. Your class must have a primary key attribute for this to work.

11.

Transactions

A transcation is a collection of database modification commands that will be run all at once. Transactions vastly improve performance of Insert, Update, and Delete functions on a database. If you are calling many commands all in a row, you will see an improvement in performance by using a transaction.

To start a transaction, simply call before your statements you want to batch:

```
dbManager.BeginTransaction();
```

When you are done with your statements, you can then commit the transaction to the database, making it run all the commands:

```
dbManager.Commit();
```

11.1 Transaction Examples

Transactions with SQL Statements

```
string sql = "INSERT INTO PlayerStats (PlayerName, TotalKills,
Points) VALUES (?, ?, ?)";

dbManager.BeginTransaction();

dbManager.Execute(sql, "Player 1", 2, 100);
dbManager.Execute(sql, "Player 2", 11, 1584);
dbManager.Execute(sql, "Player 3", 0, 0);

dbManager.Commit();
```

You can see in this example why setting up your SQL statement with parameter bindings using the ? can come in handy. You only have to specify the SQL statement once and then bind it multiple times.

Note that your commands will not process until you call the Commit function if you have started a Transaction.

Transactions With Class Definitions

In this example we first set up a list of PlayerStats. We will pass this list to the manager using the InsertAll function which starts and commits a transaction for us. We populate the list with instantiated objects of our PlayerStats class.

```
PlayerStats playerStats;

List<PlayerStats> playerStatsCollection = new List<PlayerStats>();
```

```
playerStats = new PlayerStats { PlayerName = "Player 1", TotalKills
= 2, Points = 100};
playerStatsCollection.Add (playerStats);

playerStats = new PlayerStats { PlayerName = "Player 2", TotalKills
= 11, Points = 1584};
playerStatsCollection.Add (playerStats);

playerStats = new PlayerStats { PlayerName = "Player 3", TotalKills
= 0, Points = 0};
playerStatsCollection.Add (playerStats);

dbManager.InsertAll(playerStatsCollection);
```

12.

Creating, Altering, and Dropping Tables

You can create and drop tables programmatically at runtime.

12.1 Create Table Examples

Create Table With SQL Statements

This example shows how to create a table and an index for the table. You can set the attributes of each field (such as primary key, not null, etc.) using the SQL statement.

```
string sql;

sql = "CREATE TABLE \"StarShip\" " +
                "(\"StarShipID\" INTEGER PRIMARY KEY  NOT NULL, " +
                "\"StarShipName\" varchar(60), " +
                "\"HomePlanet\" varchar(100), " +
                "\"Range\" FLOAT, " +
                "\"Armor\" FLOAT, " +
                "\"Firepower\" FLOAT)";
dbManager.Execute(sql);
```

```
sql = "CREATE INDEX \"StarShip_StarShipName\" on
\"StarShip\"(\"StarShipName\")";
dbManager.Execute(sql);
```

Create Table With Class Definition

This example shows how powerful SimpleSQL is when creating a table from a predefined class definition. See the Data Structure Chapter to see the StarShip class definition.

```
dbManager.CreateTable<StarShip>();
```

12.2 Alter Table Examples

Alter Table With SQL Statements

This example shows how to add a column to a table

```
string sql;

sql = "ALTER TABLE \"LocationMapping\" ADD COLUMN \"NewField\"
INTEGER";

dbManager.Execute(sql);
```

Alter Table With SQL Statements # 2

Though not technically an alter statement, this example shows how to drop a column from a table.

This example removes a column by first renaming a table to a temporary location, then creating a new table with the original name, then copying the data from the temp table to the new table, and finally removing the temp table. All this is necessary because you cannot simply drop a column.

This method can also be used to change column names, ordering, or types as well.

Note that we use a transaction here to group all the commands into a single call for efficiency and performance.

```
string sql;

// start a transaction to speed up processing
dbManager.BeginTransaction();

// rename our table to a backup name
sql = "ALTER TABLE \"StarShip\" RENAME TO \"Temp_StarShip\"";
dbManager.Execute(sql);

// create a new table with our desired structure, leaving out the
dropped column(s)
sql = "CREATE TABLE \"StarShip\" " +
            "(\"StarShipID\" integer PRIMARY KEY  NOT NULL , " +
            "\"StarShipName\" varchar(60) NOT NULL , " +
            "\"HomePlanet\" varchar(100) DEFAULT Earth , " +
            "\"Range\" float NOT NULL , " +
            "\"Armor\" float DEFAULT 120 , " +
```

```
            "\"Firepower\" float) ";
dbManager.Execute (sql);

// copy the data from the backup table to our new table
sql = "INSERT INTO \"StarShip\" " +
            "SELECT " + "" +
            "\"StarShipID\", " +
            "\"StarShipName\", " +
            "\"HomePlanet\", " +
            "\"Range\", " +
            "\"Armor\", " +
            "\"Firepower\" " +
            "FROM \"Temp_StarShip\"";
dbManager.Execute(sql);

// drop the backup table
sql = "DROP TABLE \"Temp_StarShip\"";
dbManager.Execute (sql);

// commit the transaction and run all the commands
dbManager.Commit();
```

12.3 Drop Table

To drop a table and/or index programmatically, you call the SQL statement like this:

```
string sql;

sql = "DROP INDEX \"StarShip_StarShipName\"";
dbManager.Execute(sql);

sql = "DROP TABLE \"StarShip\"";
dbManager.Execute(sql);
```

13.

Upgrading Databases

13.1 Database Workflow

SimpleSQL uses the database located in your application's working directory at runtime. This directory is not the same as your project directory, where you set up the link to the database. This allows SimpleSQL to make modifications to the database. It also allows you to create multiple working databases from a single project database. Using an Object Oriented Programming analogy, you can think of the project database as the template or class and the working databases as the object or instantiated class.

If your database's data will change during runtime, then your database is said to have dynamic data. If you are only using the database for settings and values that will not change during runtime, then your database is said to have static data. You may also have some data that is dynamic and some that is static.

If you are using only static data, then you can safely check the Overwrite if Exists property on the SimpleSQLManager. This will completely wipe out the database that exists in your device's working directory and replace it with the database in your project. Since nothing changes in your working directory, wiping it out will have no consequence.

If you are using dynamic only data or a mixture of static and dynamic data, then you **DO NOT** want to check the Overwrite if Exists property on the SimpleSQLManager. Overwriting the data would wipe out any changes the user makes at runtime and would not be desireable.

So how do you update a dynamic database if you can't overwrite it?

13.2 Upgrade Path

The most common method for keeping a working database in sync with your project database is to keep track of your database version (not the same as your project version). You can then upgrade your tables and table structures based on a set of steps between the working database's current version and the project database's version. This is known as the upgrade path.

Typically you will store the database version inside each of your databases so that you know what upgrade path to take for each.

For example:

Let's say your project database is at version 3.0. You may have three working databases in your application's working directory that are at versions 1.0, 2.0, and 3.0, all created from this project database.

In order to get all your databases in sync with your project database, you will need to create an upgrade path for each version of your database. At the beginning of your application's life cycle, you would call the upgrade path on each of your databases. Something like this (psuedocode):

```
if (dbVersion == 1.0)
{
    // add tables, change table structure, or modify data to get to
version 2.0

    dbVersion = 2.0;
    // update dbVersion in database to 2.0
}
```

```
if (dbVersion == 2.0)
{
    // add tables, change table structure, or modify data to get to
version 3.0

    dbVersion = 3.0;
    // update dbVersion in database to 3.0
}
```

Your database at version 1.0 will enter the first logic block, upgrading to 2.0. It will then enter the second logic block since it is now at 2.0 and upgrade to 3.0.

Your database at version 2.0 will skip the first logic block and enter the second, upgrading to 3.0.

Your database at version 3.0 won't enter any upgrade logic blocks since it is already up-to-date.

13.3 Redundancy

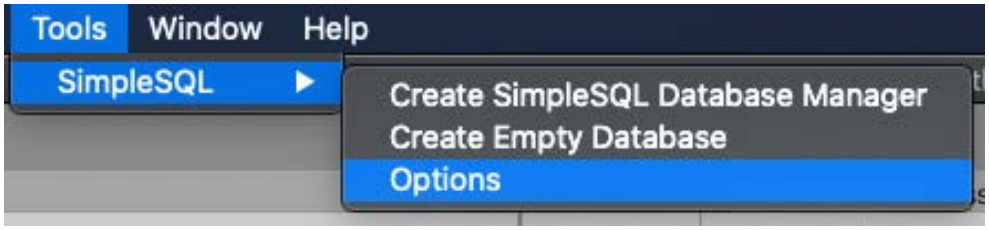
You may note that using an upgrade path can introduce redundancy. For example, let's say in the logic blocks above for upgrading from 1.0 to 2.0 you add a table called "TableA". Then in the logic block for upgrading 2.0 to 3.0 you delete the table "TableA". If your database is at version 1.0, it will add then immediately delete "TableA", which on the surface seems pointless, but gives you complete control over a database's changes from any starting version to the final version.

14.

Options and Optimization

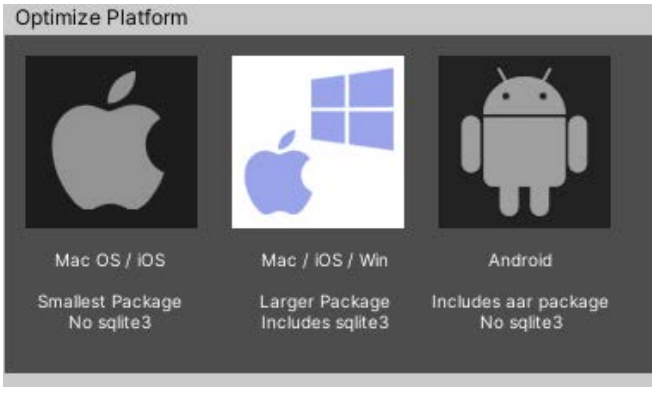
To allow more flexibility and optimization, SimpleSQL has the ability to use different DLLs for different platforms and data structure libraries. To access the options window, go to the Unity menu SimpleSQL > Options.

Figure 14-1 Options



14.1 Optimize Platform

Figure 14-2 Optimize Platform



If you will be running your application only on Mac OS or iOS, then the first platform option is the best solution. It strips out the sqlite3 library which already exists on these platforms, making your final package much smaller.

If you will be running your application on Windows Mac OS, or iOS, then the Universal option is the best solution. It will include the sqlite3 libraries required by Windows, but will also make your final package a little larger.

Android requires a special sqlite package to be included. Choosing the Android option will copy this package into your project.

14.2 Optimize Data Library

Figure 14-5 Optimize Data Library

If you want to use the .NET System.Data library which includes data structures such as DataTable, DataRow, and DataView, then you'll need to select Use System.Data. If you do not need these structures and prefer to use the ORM classes, then you'll want to select No System.Data.

If you switch from using System.Data to not using it, then you will be warned that you should remove all references to System.Data and SimpleSQLManager_WithSystemData from your scripts and gameobjects. Failure to do so may cause errors or Unity to crash.

Figure 14-6 Confirm System.Data Removal

If you switch from not using System.Data to using it, you will be notified that you'll need to set the full

.NET API in the player settings for the library to work properly at runtime.

Figure 14-7 Use Full .NET

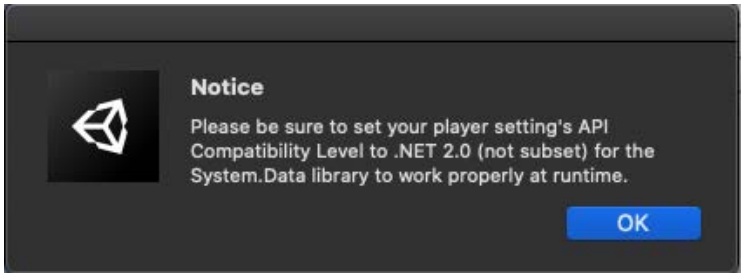


Figure 14-8 Full .NET API

For more information about using System.Data see Using .NET's System.Data and System.Data Query Examples.

You can still use the ORM classes alongside the System.Data structures.

15.

FAQ and Troubleshooting

Check out echo17.com to find links to resources, including forum support.