LiteDB for Unity

Woodbine Software

Introduction

LiteDB for Unity allows a developer to easily store their data in a document database which is saved to a local file. This can be useful for saving settings, creating game saves or checkpoints, item databases, or any number of other uses.

LiteDB for Unity is based on the LiteDB database tool. It has been rebuilt to work with Unity and tested on many popular platforms, including Android and iOS. Additionally, it has been extended to support Unity data types, such as Quaternion and Vector3.

Getting started

The most common way to use LiteDB for Unity is to create scripts that utilize the `LiteDatabase` and `LiteCollection<T>` classes. The developer will create POCO classes which represent the data they wish to save in the database. Usually, each collection in the database will contain one type of document, which is represented by a POCO class. However, there is not a requirement that all documents in a collection are of the same type.

To get started, we'll show a quick example of inserting and querying data.

First, a LiteDatabase instance is created by specifying the path to the file where the database will be stored. It is recommended that one of the paths provided by Unity's Application class is used to make sure the file is placed in an appropriate location that is available cross platform. Typically, `Application.persistentDataPath` is used.

```
string dataFile = Path.Combine(Application.persistentDataPath, "MyDatabase.db");
var db = new LiteDatabase(dataFile);
```

Next, an instance of `LiteCollection<T>` is retrieved from the `LiteDatabase` instance. If the collection doesn't exist yet, a collection will be created automatically this first time a document is inserted.

```
LiteCollection<Customer> col = db.GetCollection<Customer>();
```

The instance of LiteCollection can be used to insert, update, delete, or query data.

```
// Create a customer and insert it into the collection.
var customer = new Customer
{
    Name = "Tim Johnson",
    Phones = new[] { "555-5555" },
    Age = 29,
};

col.Insert(customer);

// Query for a Customer with an Id of 1.
Customer queryResult = col.FindOne(Query.EQ("Id", 1));
```

An index, which is used to help speed up queries, can be created with the `EnsureIndex()` method:

```
// Create an index on the Name property.
col.EnsureIndex("Name");
```

The following POCO class is what we've been using in these examples:

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string[] Phones { get; set; }
}
```

Supported platforms

LiteDB for Unity has been tested and works on many of the major platforms that Unity supports:

- Windows
- Mac
- Android
- iOS
- *Support for UWP and WebGL will be coming soon!*

Be cautious when querying for data for on a platform that uses IL2CPP, such as iOS. IL2CPP does not support the ability to compile code on the fly, which is a needed platform feature when Expressions are used to query data. Instead of querying with a LINQ Expression, use the Query class, which does not require just-in-time compilation.

Instead of using an `Expression` like this:

```
var results = col.Find(c => c.Age > searchAge);
```

Use a `Query` like this:

```
var results = col.Find(Query.GT("Age", searchAge));
```

Supported Unity data types

LiteDB for Unity has support for several common Unity data types:

- Vector2
- Vector3
- Vector4
- Quaternion
- Matrix4x4

These data types can be used in documents stored to the database without any additional work if the default global `BsonMapper` is used. In the database, these data types are stored as a byte array. This allows for fast, compact storage and for these data types to be indexed. However, if it is desirable to

store these data types in a different format, they can be easily overridden by registering custom serialization methods. This can be done by calling the `BsonMapper.RegisterType` method:

```
// Register a customized serialization method for the Vector3 type.
BsonMapper.Global.RegisterType<Vector3>(MyCustomSerializeMethod,
    MyCustomDeserializeMethod);
```

Similarly, if a custom `BsonMapper` instance is being used, the support for Unity data types can be easily added to it by calling `RegisterType()` with the serialization methods defined in the `UnityTypes` class. For example, to register serialization for a Quaternion on a custom `BsonMapper`:

```
bsonMapper.RegisterType<Quaternion>(UnityTypes.QuaternionSerialize,
    UnityTypes.QuaternionDeserialize);
```

IDs in LiteDB for Unity

All documents stored in a database need to have an identifier. The most common way to define an ID on a POCO class is to include an integer Id property.

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

LiteDB will automatically recognize this as the identifier.

Alternatively, identifiers can be a different data type. These data types can be used: `ObjectId`, `Guid`, `Int32` (int), `Int64` (long), and `DateTime`.

It is not a requirement that the property in the POCO class be named “Id”. If it is desirable to have a different name, use the [BsondId] attribute on the property to indicate that it should be used as the document’s id:

```
[BsonId]
public int PlayerId { get; set; }
```

References to other documents

It is possible for documents in LiteDB to reference other documents stored in the same database. This is desirable when one document should be linked to another document, rather than embedding a copy of the linked document.

The classic example of this is the Order and Customer scenario. In this scenario, an Order document represents an order that a customer has placed. The Order document should reference the Customer document, so it can be determined who placed the order; the Customer document should not be stored directly in the Order document so that the Customer information can be updated independently of Orders.

In LiteDB, this can be easily implemented by adding a BsonRef attribute to the Customer property.

```
public class Order
{
    public int Id { get; set; }
    public DateTime OrderPlaced { get; set; }

    // BsonRef indicates that Customer is a referenced document.
    [BsonRef]
    public Customer Customer { get; set; }
}

public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

LiteDB will now recognize that the Customer property is a reference, and the Order document will be stored with a reference to the Customer document in the Customer collection, rather than having the Customer stored embedded inside of the Order document.

Examples Overview

Three example scenes are provided with the Asset Store package to demonstrate usage of LiteDB for Unity. The scripts, scenes, and related item are all located inside the **LiteDB for Unity/Examples** folder.

Website Example Scene

This example demonstrates 2 simple usage scenarios, similar to the one described in the "Getting Started" section. There scenarios are derived from the www.litedb.org home page and are designed to quickly introduce the usage of LiteDB for Unity. The focus of this example is the code itself, which can be found in the **WebsiteExamples.cs** script.

Scene file: LiteDB for Unity/Examples/Scenes/Website Simple Examples.unity
Code file: LiteDB for Unity/Examples/Scripts/WebsiteExamples.cs

Settings Example

The Settings Example demonstrates how LiteDB for Unity could be used to save the settings of a game. It features a range of data types being persisted, and provides a template for save and loading data from the database into the UI and back again.

Scene file: LiteDB for Unity/Examples/Scenes/Settings Example.unity
Code file: LiteDB for Unity/Examples/Scripts/SettingsExample.cs

Save Game Example

The Save Game Example demonstrates a method of creating and loading game save states. This is the most complicated of the examples, and includes a mini-game, which is driven by several scripts. The support for Unity data types is employed in this example to store the locations of game objects.

The most important script for the purposes of saving and loading data is the GameManager.cs script. To discover how LiteDB for Unity is utilized, examine these methods: `SaveGame()`, `LoadLastSave()`, and `LoadSelectedSave()`.

Scene file: LiteDB for Unity/Examples/Scenes/Save Game Example.unity
Code folder: LiteDB for Unity/Examples/Scripts/Save Game Example Scripts/
Primary script: GameManager.cs

FAQ

Where should I store my database files?

It is recommended to store your files in one of the folders provided by Unity's `Application` class, either `Application.persistentDataPath` or `Application.temporaryCachePath`. `Application.persistentDataPath` will be used in most common scenarios.

How is date and time stored in a LiteDB database?

LiteDB supports the storage of `DateTime` objects. A `DateTime` object will be converted to UTC when it is stored and converted back to local time when it is retrieved.

What version of LiteDB is LiteDB for Unity based on?

LiteDB for Unity is based on version 3.1.4 of LiteDB. It will be continuously updated with the latest patches and fixes.

What about v4.0 of LiteDB?

Unfortunately, versions 4.0 and above of LiteDB rely heavily on just-in-time compilation of Expressions for performing queries, which renders it unsuitable for IL2PP platforms like iOS. As a result, we can't list it on the Asset Store.

However, we do have v4.0+ versions of LiteDB for Unity available! Just be aware that it may not work with iOS or other IL2CPP builds. Send us a request to support@woodbine.software and include your invoice number, and we will happily send you a copy!

Additional Support

If you encounter issues, have questions, or would like to leave feedback, we'd like to hear from you so we can help!

Support email: support@woodbine.software

Website: http://woodbine.software

If you're enjoying LiteDB for Unity, please consider leaving a good review on the Unity Asset Store!