Sterling Object-Oriented Database for Silverlight and Windows Phone 7

Version 0.1 (alpha)

For the latest documentation, please visit the website at:

http://sterling.codeplex.com/

Sterling is designed to be a fast and easy to use local object-oriented database for Silverlight and Windows Phone 7. This page will help you get started with Sterling and handle indexes, keys, and custom types for serialization and deserialization to isolated storage within your projects.

Table of Contents

Quick Start

Let's get started right away with Sterling. Assume we have a contact class that we want to save and use the email as the key. Our class looks like this:

```
public class Contact
{
   public string Email { get; set; }
   public string PhoneNumber { get; set; }
   public string LastName { get; set; }
   public string FirstName { get; set; }
}
```

Defining your Class as a Table

Add the Sterling project or DLL to your solution and reference it. It is recommended you create a project specifically for the Sterling database, for example, "ContactDatabase." Create a class called "MyDatabase" and inherit from BaseDatabaseInstance. Sterling supports multiple databases, and the only requirement for an application is that they each have a unique name. There are two overrides you must implement:

```
public class MyDatabase : BaseDatabaseInstance
{
   public override string Name { get { return "My Database"; } }
   public override List<ITableDefinition> _RegisterTables()
   {
   }
}
```

The _RegisterTables method is where you define how to save your class. The key should be unique. Sterling assumes any save with the same key is an update to that record. We'll define our table using the base CreateTableDefinition method like this:

```
public override List<ITableDefinition> _RegisterTables()
{
 return new List<ITableDefinition>
                       {
                           CreateTableDefinition<Contact,string>(c => c.Email)
                       }
}
```

That's all there is to it - the only requirement is that the key is a serializable type (see below). You can create a custom serializer if you wish to use a complex key. Now we need to make the database available to our application.

Using Sterling

To use Sterling, you will create an instance of the Sterling Engine and then dispose of it. It is recommended you create the instance during the Application startup and dispose it during the Application exit.

1. Create the engine: SterlingEngine engine = new SterlingEngine();
2. Register any custom loggers or serializers *before* activating the engine (see below)
3. Activate the engine: engine.Activate();
4. Register your database with the engine: ISterlingDatabaseInstance myDatabase = engine.SterlingDatabase.RegisterDatabase<MyDatabase>();
5. Now the engine is ready for you to use. If you register the same database twice, it will throw an exception.
6. Unhook any loggers from the engine (see below)
7. When you are done with the engine, dispose it: engine.Dispose();

Here is an example service that can be added to your App.xaml page from the Sterling Phone example project:

```
public class DatabaseService : IApplicationService, IApplicationLifetimeAware
{
    public static DatabaseService Current { get; private set; }

    public ISterlingDatabaseInstance Database { get; private set; }

    private SterlingEngine _engine;

    public void StartService(ApplicationServiceContext context)
    {
        Current = this;
        _engine = new SterlingEngine();
    }

    public void StopService()
    {
        _engine = null;
    }

    public void Starting()
    {
        _engine.Activate();
        Database = _engine.SterlingDatabase.RegisterDatabase<PhoneDatabase>();
    }

    public void Started()
    {
        return;
    }

    public void Exiting()
    {
        _engine.Dispose();
    }

    public void Exited()
    {
        return;
    }
}
```

Saving Your Class

Saving your class is easy. Sterling will automatically map your class based on the type. Stering does not auto-generate keys, but it will pass back the key from the instance as the result of a save operation:

```
var key = myDatabase.Save(myContact);
```

Loading Your Class

To load a class, simply give Sterlng the class type and the key:

```
var contact = myDatabase.Load<Contact>("johndoe@somedomain.com");
```

Querying by Key

Sterling supports full LINQ to Object queries on keys. Keys are returned as a TableKey<TType,TKey> where TType is the type of class, and TKey is the type of key. The "Key" is the key value, and "LazyValue" is a lazily-loaded value for the key.

This query will not de-serialize anything from disk - you can bind it to a list and then simply reference item.LazyValue.Value to get the full entity:

```
var query = from key in myDatabase.Query<Contact,string>() orderby key select key;
```

Note you simply pass the class type and key type to get the correct query from the database.

Deleting your Class

To delete a class, simply pass an instance:

```
myDatabase.Delete(myContact);
```

```
```

or a type and a key:

```
myDatabase.Delete(typeof(Contact), "johndoe@somedomain.com");
```

Truncating a Table

Truncate will wipe the table, indexes, and keys from disk and provide a clean copy. It is non-reversible. To truncate a table, simply:

```
myDatabase.Truncate(typeof(Contact));
```

Purging the Database

Purging the entire database will wipe it clean, including all tables, indexes, and keys. This is not reversible. Simply call purge:

```
myDatabase.Purge();
```

Understanding the Sterling Engine

The Sterling engine is used to help coordinate some of the synchronized features of the database. Sterling only supports a few basic types to remain lightweight. If there are other types you wish to serialize, you can create a custom serializer (see below) and then register this with the engine. You may also have an unlimited number of loggers that you provide to Sterling. Sterling assigns each logger a Guid you can save and use to unhook from Sterling at any time. All of this registration must be done before activation of the engine.

Once the engine is activated, you can register your databases. The engine will pass the loggers and serializers to the databases. At this time, there is no way to have a different set of loggers or serializers per database.

Sterling is fast for querying keys and indexes because it stores these in memory. While saves update the in-memory copies, they are not persisted to disk unless you manually flush them:

```
myDatabase.Flush();
```

Or when the engine is disposed. Because of the way the disposable pattern works, it is recommended that you implement the engine as an IApplicationService that is IApplicationLifetimeAware and implements IDisposable like this:

```
public void Dispose()
{
    if (_engine != null)
    {
        _engine.Dispose();
    }
    GC.SuppressFinalize(this);
}
```

You can then explicitly call dispose on the application exiting event.

Advanced Sterling

Sterling is more than simple serialization. Sterling supports extensibility so you can serialize absolutely any type of structure or class. It also supports having indexes. In some cases, classes may have covered indexes and can be queried rapidly without pulling from isolated storage.

Adding Indexes to Tables

To add an index to a table, use the WithIndex extension method. You may add an index with one or two index values. Each index by default also includes the key. An index with two values returns those as a Tuple. If you wanted to index three or more values, you could simply implement your own Tuple and provide a custom serializer for it.

Here we add an index that covers last name and first name (and because it is the key, e-mail as well). Indexes require a name that must be unique per table. It is suggested you provide these as public constants on the database definition so they can be consistently referenced elsewhere in the application. Here is an example of adding the index, then loading the user's name and email and sorting by last name. The class type is always first, and the key type is always last. So in this case, the Contact class is first, the two strings are for the name tuple, and the final string is for the email key. If the key were an integer, then it would be registered as <Contact,string,string,int> instead:

```
public const string CONTACT_BY_LAST_NAME = "ContactLastName";

public override List<ITableDefinition> _RegisterTables()
{
 return new List<ITableDefinition>
                       {
                           CreateTableDefinition<Contact,string>(c => c.Email)
.WithIndex<Contact,string,string,string>(CONTACT_BY_LAST_NAME, c =>
Tuple.Create(c.LastName, c.FirstName))
                        }
}

var query = from index in
myDatabase.Query<Contact,string,string,string>(MyDatabase.CONTACT_BY_LAST_NAME) orderby
index.Index.Item1 select new
         {
            LastName = index.Index.Item1,
            FirstName = index.Index.Item2,
            Email = index.Key
        };
```

Using the Default Logger

The Silverlight version of Sterling comes with a default logger that automatically attaches to the Sterling engine and writes log entries to the Debugger. It will only do this if the debugger is attached, and is useful for watching debug data as you run the Silverlight project in debug mode.

To use the default logger, simply create and reference an instance of it *before the engine is activated*, and pass in the minimum level to log:

```
private SterlingDefaultLogger _logger;

...

_logger = new SterlingDefaultLogger(SterlingLogLevel.Verbose);
```

Just before disposing the engine, deatch the logger by calling the detach method:

```
_logger.Detach();
```

Adding a Custom Logger

You can add as many custom loggers as you like. The reference application for Silverlight shows a logger attached to the UI. You must attach a logger prior to activating the engine.

Use the *RegisterLogger* method to register the logger. You must pass an Action delegate that accepts a severity, message, and exception object. When you call the method, a GUID is returned to track the registration. You can later use the *UnhookLogger* method to detach it. Here is an example:

```
Guid _myLoggerId;
_myLoggerId =
_engine.SterlingDatabase.RegisterLogger((severity,message,exception)=>MyLogger.WriteLine(
"{0}{1}{2}",severity,message,exception));
...
_engine.SterlingDatabase.UnhookLogger(_myLoggerId);
_engine.Dispose();
```

Building a Custom Serializer

By default, Sterling will do one of two things with every property on a class. If the property is another class with a table definition, it will serialize the key and then update the value the key for the "foreign table" points to. Otherwise, it will serialize based on any of the existing BinaryWriter overloads. These include: bool, byte, byte[], char, double, float, int, long, sbyte, short, string, uint, ulong, and ushort. Sterling automatically handles IList of any of the previous types, and has a special serializer for DateTime, Guid, and Uri (Guid support is not present for the Windows Phone 7 due to the lack of the Parse method).

Sterling can handle any type of serialization, however, as long as you provide a serializer. The serializer must be based on the *BaseSterlingSerializer* class.

The methods include *CanSerialize* which is passed a type. Your class must return true if it can handle serialization and deserialization of that type, otherwise false. The *Serialize* method is passed an object (or struct) and a BinaryWriter. You simply serialize to the writer however you like. The *Deserialize* method is passed a type and a reader. You can instantiate the type and then deserialize it from the stream. Here is an example for handling the custom struct that is in the reference application:

```
public class FoodSerializer : BaseSerializer
{
    public override bool CanSerialize(Type targetType)
    {
        return targetType.Equals(typeof (NutrientDataElement));
    }

    public override void Serialize(object target, BinaryWriter writer)
    {
        var data = (NutrientDataElement)target;
        writer.Write(data.NutrientDefinitionId);
        writer.Write(data.AmountPerHundredGrams);
    }

    public override object Deserialize(Type type, BinaryReader reader)
    {
        return new NutrientDataElement
                    {
                        NutrientDefinitionId = reader.ReadInt32(),
                        AmountPerHundredGrams = reader.ReadDouble()
                    };
    }
}
```

It is fine to handle multiple types in the same serializer or even to serialize based on derived types. Once you have defined your serializer, you simply register it with the engine *before* activation, like this:

```
_engine.SterlingDatabase.RegisterSerializer<FoodSerializer>();
_engine.Activate();
```

The serializer is then available to all databases defined.