GameStorage Client Asset

Contents

Intro

The Game Storage Client Asset is an asset that allows multiple sets of user-defined data (models) to be stored in a single place.

The location where data should be stored or retrieved from and it's format is also user-definable. This ranges from the UCM's Game Storage Server (GSS for short), locally on a device, in-game (only retrieval) and no storage (i.e. transient data).

The definition of the data is stored independently from the data so saving and restoring to and from multiple locations is possible.

Internals

The asset uses a Node class to build a tree of named nodes. This allows access by name.

Each Node can define its own storage location or inherit it from its parent. This minimized defining a model.

A Node has only a few properties:

- **Children** – A List of Child Nodes, if empty or null the node is a leaf.
- **Name** – The name of the node.
- **Parent** – The Node's Parent Node.
- **Value** – The actual Node Value

The following properties are computed when accessed:

- **StorageLocation** – The storage location (it's set by the constructor if not inheriting from its parent).
- **Count** – The number of Children.
- **IsRoot** – If the Parent is null, the node is considered to be the root of the tree
- **Path** – A dotted string of names, used for addressing and serialization.

The structure and data are stored independently so after the structure is restored, data can be restored from multiple locations.

The serialization for persistence is performed either by the build-in XmlSerializer class or by an external JSON library because of lack of JSON support in .Net 3.5 (Mono). When storing data in the Game Storage Server, JSON is the only supported format as the service is MongoDB based and only accepts JSON at the moment.

Json (de)serialization is not a simple process of making a single call to a JSON library as one would expect because the data types are unknown at compile time (normally JSON libraries take a class as template of which to restore).

The result is that although a single call serialization would work, deserialization would be lossy in terms of data types. JSON libraries normally try to guess the best data type for values. So a byte, short, int and long will all end up as a 64 bit integer (as it always fits).

Matter is further complicated by the differences between the Unity3D JSON support (which is fast but simplistic as it only restores fields and very simple classes) and Json.Net popular and often used in Xamarin code.

The .Net XmlSerializer is slightly better in its jobs as it saves the data types automatically. However it translates all generic lists into an array and will restore them as array.

So serialization is formatted as an array of simple nodes objects where object contains the path, value and datatype of the node.

The asset contains code to cache XmlSerializers as the first time they are created it can be quite slow. After caching them, XmlSerialization performs just as well and binary serialization. The same seems to hold for JSON libraries as the popular Newtonsoft Json.Net.

Binary serialization has been disabled for now as its support is very limited in PCL assemblies for Xamarin.

Configuration

The configuration consists of a few settings:

- Host – The hostname or IP address of the RAGE A2 service (which also acts as a proxy to the Game Storage Service).
- Port – The port on which the A2 Authentication & Authorization service can be accessed.
- A2Port – The port on which the Game Storage Web service can be accessed.
- BasePath – The Web Service Api path, typically '/api/' (note the trailing /)
- Secure – If true the HTTPS protocol is used if false HTTP is used instead.
- UserToken – Default 'a:' for anonymous access. This value will be overwritten with the actual Authorization token once a user has logged in.

Defining a Model

The asset contains a storage property that contains the models.

Depending on the functionality, the API methods are located in the GameStorageClientAsset class or in the Node class.

Models are added by using the AddModel() method like:

```
GameStorageClientAsset storage = new GameStorageClientAsset();

storage.AddModel("User");
```

Note: storage has a string indexed so models root node can be accessed like:

```
storage["User "]
```

Note: the root node is not serialized but re-created and should not contain data.

A Node contains a number of methods

- Clear() – Clears a model
- AddChild() – Add a Child node (this method has some overloads).
- RegisterTypes() - For registering types for XmlSerialization at startup.

Adding a server based value:

- `storage["User"].AddChild("Server", String.Empty, StorageLocations.Server);`

Adding a game based 'virtual' value (read-only):

- `storage["User"].AddChild("Virtual", StorageLocations.Game);`

Adding a transient value (not stored):

- `storage["User"].AddChild("Transient", Double.MaxValue, StorageLocations.Transient);`

Adding values with inherited storage locations:

- `storage["User"].AddChild("Name", "Me");`
- `storage["User"].AddChild("Age", 25);`
- ```
  storage["User"].AddChild("STRUCT", new DemoClass
      { a = 10, b = "elf", c = DateTime.Now });
  ```

Note: Data types added must off-course be serializable. In case of **Unity3D** and **Json** this implies that only classes and fields can be used (**Unity3D** does not support serialization of stucts).

Adding a ChildNode (E) to another Node (D):

- `Node D = B.AddChild("D", DateTime.Now);`
- `D.AddChild("E", (Byte)5);`

Note: AddChild() returns the newly added Node.
  ```

Addressing a Node by path:

- `storage["Wiki"]["F"].Value = 42;`
- `Int32 age = storage["Wiki"]["F"].Value;`

Note: Full dotted path addressing will be added in a upcoming release.

Persisting

Saving the structure to the Game Storage Server requires a connection to be made like:

```
if (storage.CheckHealth())
{
    Log(storage.Health);

    if (storage.Login(user, pass))
    {
        Log("Logged-in");
    }
}
```

After this saving the structure looks like:

```
if (storage.Connected)
{
     storage.SaveData("Wiki", StorageLocations.Server);
}
else
{
     Log("Not Connected");
}
```

Saving the structure locally (using IDataStorage):

```
storage.SaveStructure("Wiki", StorageLocations.Local);
```

Note: The StorageLocations parameter both specifies the location and the data to be filtered for saving.

Asset Interfaces

The asset uses the **IDataStorage** interface for local storage and retrieval of settings.

For communication with the GameStorage Server, the **IWebServiceRequest** interface is used.

For retrieval of Game based values the **IVirtualProperties** interface is used.

Optionally the **ILog** interface (when present) is used for diagnostic logging.

In can optionally use the **IDefaultSettings** interface, to allow defaults to be compiled as a resource into the game (as opposed to configuring in game code).

IDataStorage, **IWebServiceRequest** and **ILog** are described in the Asset Base Manual document.

IVirtualProperties

This interface has a single method to be implanted, it has the model name and a key as parameter and expects a return value (although defined as an Object it should off-course match the definition in the model.

```
/*
 * Copyright 2016 Open University of the Netherlands
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * This project has received funding from the European Union's Horizon
 * 2020 research and innovation programme under grant agreement No 644187.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
namespace AssetPackage
{
    using System;

    /// <summary>
    /// Interface for virtual properties.
    /// </summary>
    public interface IVirtualProperties
    {
        #region Methods

        /// <summary>
        /// Looks up a given key to find its associated value.
        /// </summary>
        ///
        /// <param name="model"> The model. </param>
        /// <param name="key">   The key. </param>
        ///
        /// <returns>
        /// An Object.
        /// </returns>
        Object LookupValue(String model, String key);

        #endregion Methods
    }
}
```