Realtime Storage for Unity3d

About

Realtime Cloud Storage is a fully managed NoSQL database service based on Amazon DynamoDB that provides fast and predictable performance with seamless scalability.

If you are a developer, you can use Realtime Cloud Storage to create database tables that can store and retrieve any amount of data, and serve any level of request traffic.

Realtime Cloud Storage is integrated with Realtime Messaging to provide real time notification when data is added, removed or changed. This means that you don't need to worry about synchronizing data between users.

Before you dive into your preferred SDK please take a few minutes to read this starting guide. For your convenience it's divided into 13 small chapters that will guide you through the main concepts and best practices for optimal use of your Realtime Cloud Storage.

Requirements

Unity 4.6 or greater, Realtime Messaging for Unity3d

Supported Platforms

Standalone, Android, iOS

Main Documentation

http://storage-public.realtime.co/documentation/starting-guide/1.0.0/main-concepts.html

Installation

The Realtime Storage Client for Untiy3d is distributed as a **.unityPackage**. Here are the steps needed to install the Realtime Storage Client.

- Storage requires the Realtime Messaging client. Before proceeding, please download and install the Realtime Messaging client.
- Download the Realtime Storage Client **.unityPackage**
- Open Unity. It is suggested you start a new project.
- Double click the **.unityPackage**
- You will be prompted with an import dialog. Import the package.
- Thats it! The Realtime storage client has been imported and is ready to be configured.

Configuration

Now that the product has been installed, it is time to configure it. Configuration is managed by a scriptable object. This object may be accessed by using the **Realtime/Storage Settings** main menu command.

Account

- **Application Key** : Identifies your application. This may be acquired from the your account information on the Realtime website.
- **Private Key** : A secret token used for authorizing clients. If you want to use network security and wish to authenticate directly from the client you will need this.

Storage Url

- **Service Url** : The location of the realtime storage server
- **Is Cluster** : Is this server a cluster instance ?
- **Is Https** : Use SSL for communication ?

Messenger Url

- **Service Url** : The location of the realtime messenger instance on the storage server
- **Is Cluster** : Is this server a cluster instance ?
- **Is Https** : Use SSL for communication ?

Storage Repository

The storage repository is a service for pure data access. In other words, it does not manage objects or receive change notifications. Use the repository when you need C.R.U.D. access without worrying about a persistence.

Object Annotations

The **[StorageKey]** annotation is required for objects to be used by the storage system. This annotation identifies the table, primary key and secondary key of the object.

```
[StorageKey("Scores", "UserName", "Points")]
class Score
{
    public string UserName { get; set; }
    public int Points { get; set; }
    public int Points2 { get; set; }
}
```

C.G.U.D. (Create, Get, Update, Delete).

Once your object model is set, it is time to access your data. The storage repository exposes a number of tasks for data access. These tasks include create, get, update, delete, increment, and decrement.

The returning tasks are long running operations. You should call these operations inside coroutines. Use the WaitRoutine() IEnumerator method to pause the coroutine while the operation completes.

The tasks return a generic object of type Storage Response. This object has a data property with the returning data as well as an error property with information regarding any errors.

Example Use of a Storage Repository

```
IEnumerator ExampleCoroutine() {

       // Create a new Object to use
       var score = new Score ();

       // Create a new repository
       Repository = new StorageRepository();
       // get task
       var result1 = Repository.Create(score);
       // wait for the task to complete
       yield return StartCoroutine(result1.WaitRoutine());
       // handle client error
       result1.ThrowIfFaulted();
       // handle server error
       if (result1.Result.hasError)
           throw new Exception(result1.Result.error.message);

       //GET
       var result2B = Repository.Get<Score>(score.UserName, score.Points);
       yield return StartCoroutine(result2B.WaitRoutine());
       result2B.ThrowIfFaulted();
       if (result2B.Result.hasError)
              throw new Exception(result2B.Result.error.message);
       var score2 = result2B.Result.data;
       score2.Points += 100;

       //UPDATE
       var result3 = Repository.Update(score2);
       yield return StartCoroutine(result3.WaitRoutine());
       result3.ThrowIfFaulted();
       if (result3.Result.hasError)
              throw new Exception(result3.Result.error.message);
       var score3 = result3.Result.data;


        //DELETE
       var resultd = Repository.Delete(score3);
       yield return StartCoroutine(resultd.WaitRoutine());
       resultd.ThrowIfFaulted();
       if (resultd.Result.hasError)
              throw new Exception(resultd.Result.error.message);
}
```

For more examples, please see the Demos/StorageTests.cs script included with the project.

Storage Controller

The storage controller utilizes the Realtime messenger to give the storage repository awareness of any changes. In other words, if another user makes a change to a storage object or table, the controller will receive change notification and update any storage, table or item references in memory.

Construct

If an authentication token is passed to the StorageController's constructor, the Controller will connect and authorize for change notification. You may wait for this connection by running the WaitForConnect coroutine.

```
// Construct
Context = new StorageController(AuthKey);

// Wait for Connect
yield return StartCoroutine(Context.WaitForConnect());
```

Disconnection

Be sure to disconnect the controller when it is no longer of use. For instance, when the game enters into sleep mode. You may reconnect the controller when the game wakes up again.

Create a Table Reference

Once connected, you may now create a Table Reference. A Table reference is an is a specialized collection which is wired to receive table change notification.

```
// get score table
var tableTask = Context.Table<Score>();
yield return StartCoroutine(tableTask.WaitRoutine());
tableTask.ThrowIfFaulted();
// Use score table reference
var table = tableTask.Result;
```

Manual Listening

The Table Reference exposes a number of methods for handling change notification manually.

```
        //Wire listeners
        int callbacks = 0;
        table.On(StorageEventType.DELETE, response =>
        {
            Terminal.Log("DELETE " + response.Val<Score>().UserName);
        });

        table.On(StorageEventType.PUT, response =>
        {
            Terminal.Log("PUT " + response.Val<Score>().UserName);
        });

        table.On(StorageEventType.UPDATE, response =>
        {
            Terminal.Log("UPDATE " + response.Val<Score>().UserName);
        });
```

C.G.U.D. (Create, Get, Update, Delete).

Once connected and ready, the Table Reference exposes a number of tasks for data access. These tasks include create, get, update, delete, increment, and decrement.

Example Use of a TableReference for Data Access

```
    var score = new Score
    {
           Points = Strings.RandomNumber(100, 10000),
           Points2 = Strings.RandomNumber(100, 10000),
           UserName = Strings.RandomString(10),
    };

    var tresult1 = Context.Repository.Create(score);
    yield return StartCoroutine(tresult1.WaitRoutine());
    tresult1.ThrowIfFaulted();

    var tresult2 = Context.Repository.Update(score);
    yield return StartCoroutine(tresult2.WaitRoutine());
    tresult2.ThrowIfFaulted();

    var tresult3 = Context.Repository.Delete(score);
    yield return StartCoroutine(tresult3.WaitRoutine());
    tresult3.ThrowIfFaulted();
```