Online Account System

..for Unity3D

by

Konstantinos Vasileiadis

• Documentation

Although server-side, back-end, front-end and example codes are well documented here's a more detailed view of the key classes and functions.

Contents:

• Workflow

When designing this asset, we had in mind the following flow:

– The user Registers
0) Download the registration Form
1) Add the user's input to the form fields
2) Attempt to Register with the provided input

- The user Logs In
3) Prompt the user to input their username and password to Login
4) Attempt to Login with the given username and password

- The user Manages his Account
5) Download their account Information
6) Alter their account Info (change username, custom fields, ..)
7) Upload the account Info back to the database

So we created single – line commands for you to do each of the above. You can check that at /Account System/C# Scripts/Back End/AS_ShortDemo.cs.

• Introduction

The main class of AccountSystem is AS_AccountInfo (located at /Account System/C# Scripts/Back End/AS_AccountInfo.cs).

It is essentially an array of AS_MySQLFields, each holding a MySQL field's Name, stringValue, Type, and two booleans - "If it must be unique" and "If it can be left null".

In plain English, an instance of AS_AccountInfo holds all of the information associated with that user, and serves as a middle-man between Unity and the Database.

For the rest of this tutorial, assume we have declared an **AS_AccountInfo** instance named "**accountInfo**".

That can be done with the following code:

C#:

```
AS_AccountInfo accountInfo = new AS_AccountInfo();
```

Developer's Note:

I understand that most functions will appear very simple for what they are supposed to do.

Don't worry that something's missing.

The truth is that after your initial set-up, everything is coded to execute without requiring you to specify any further information, allowing you to focus on developing your game. :)

• Background Knowledge

(What is a **Callback**?)

Before we begin, we have to briefly explain the concept of a **Callback**. If you are already familiar with callbacks, feel free to skip this short and crude explanation.

When a function takes **a long time to complete** (*for instance when it needs to connect to a MySQL server and receive data from it*) but we want the rest of the program to **keep executing**, we can declare that function as a **Co-Routine**.

After calling our Co-Routine, the program will **continue executing "in parallel"** with the Co-Routine.

The only problem is that we can't know in advance when our Co-Routine will have finished executing, so we use a technique called Callback to let the program know what we want to be executed **after** our Co-Routine has finished!

A **Callback** is just an ordinary **function**/method that we pass as an argument to the Co-Routine, so that it may be called from inside the Co-Routine when we want.

In our case that Callback function takes as input a string argument, so we can return a string message from our Co-Routine..!

That string is formatted in a specific way, and you can check if it's an error message with "message.IsAnError();".

Developer's Note:

For your convenience, all Co-Routines are being called indirectly.

For example, when you call the TryToLogin() function, that function creates a temporary Game Object and that Game Object in turn calls the TryToLogin() Co-Routine, calling your callback and then destroying itself when the Co-Routine has finished executing.

Poor guy :(

• How To's

• Creating a simple Registration GUI:

1. First we need to download the **registration form**.

We do this by filling the **accountInfo** with the required field **names**, **types** and which of them are **required** and must be filled by the user.

The **stringValues** are left blank for now.

C#:

```
accountInfo.TryToDownloadRegistrationForm ( OnFormDownloaded );
```

As a callback we pass any function that we want executed after the download has finished - as long as it has a single string parameter

C#:
```
void OnFormDownloaded ( string messageFromCoRoutine ) {
        // Do meaningful stuff with the downloaded form (it's stored at accountInfo)
}
```

2. When the download has finished, we go through each field of accountInfo, print out its name, wether or not it's required and then ask the user for his input, storing it as the field's stringValue.

C#:
```
foreach( AS_MySQLField field in accountInfo.fields )
{
        // Print field.name
        // Prompt user to fill field.stringValue
        // If field.isRequired is true
                // Verify that the user filled its field.stringValue
}
```

3. When the user has filled in the form and submits it, we just have to call

C#:
```
accountInfo.TryToRegister ( onRegistrationAttempted )
```

Note that our callback function onRegistrationAttempted can be called without having successfully registered the user (in case his username was taken for example) so we need to handle any errors.

C#:
```
void onRegistrationAttempted ( string messageFromCoRoutine ) {

        // Check if messageFromCoRoutine contains errors
                // Error Handling
        // Otherise
                // Prompt the user to Log In with his new account
}
```

- Attempting to Login:

Assuming we have two strings (**username** and **password**) Loging In is a process as simple as

C#: **username**.TryToLogin (**password**, **LoginCallback**);

As usual, if the login fails, **LoginCallback** will be called with an error message – otherwise it will be called with the user's unique **accountId**.

C#:
```
void LoginCallback ( string returnedMessage ) {

        if ( returnedMessage.IsAnError () )
                // Error Handling
        else
                int accountId = System.Convert.ToInt32( returnedMessage );
}
```

- Attempting to Recover a Lost Password:

Just as easily, you simply use the following:

C#:
```
emailAddress.TryToRecoverPassword ( RecoverPasswordCallback );
```

If all goes well, an email will be sent to the provided emailAddress - otherwise the callback will be called with an error message - perhaps "email not found".

C#:
```
void RecoverPasswordCallback ( string messageFromCoRoutine ) {

        // Check if messageFromCoRoutine contains errors
                // Error Handling
        // Otherise
                // Let the user know everything went alright
}
```

• Managing Account Information In-Game:

It's not uncommon for a developer to need access to a user's account information inside the game. Perhaps to display his name, update his score or check if he has unlocked a level.

Doing that with Online Account System is a three part process -
Downloading the account Info, **Messing around** with it and **Uploading** it back.

We assume you have the user's unique **accountId** - that's provided on successful login.

• Downloading Information:

The following code

C#:
```
accountInfo.TryToDownload ( onDownloadAttempted  );
```

Does just that.. use **onDownloadAttempted** if you want to take an action when the download is complete.

If all goes well, each of the **fields** in **accountInfo** will have its **stringValue** filled with the **value** of the according **database table field**.

• Getting/Setting Values:

To **get** a value, use:

C#:
```
string fieldValue = accountInfo.GetFieldValue ( fieldName );
```

If we could not find the specified fieldName, the return value will be **null**

To **set** a value, use:

C#:
accountInfo.SetFieldValue (**fieldName**, **fieldValue**)

This code returns a bool value - **true** if we found the fieldName and updated its value correctly - **false** if there was an error (wrong **fieldName**)

- Uploading Information:

C#:
accountInfo.TryToUpload (**onUploadAttempted**);

And yet again, as simple as that.
onDownloadAttempted will be called either when the upload has finished or if an error occured.

• Storing / Retrieving Custom Information

Sometimes storing every bit of **additional** player information as a custom fields in your database seems like an **overkill** – and it is.

Wouldn't it be easier to handle the entire information as a single compact class?

We have a special field (named customInfo) in your **accounts table** just for that.

CustomInfo is a class we provide for easily storing/accessing additional player information in your database. As long as you keep it [Serializable], you can modify it to suit your needs :)

We have an instance of that class in accountInfo.

C#:
```
accountInfo.customInfo = new CustomInfo();
```

Maybe CustomInfo has an attribute named lastLoadedLevel, that stores the player's last loaded level. After each level you can update its value:

C#:
```
accountInfo.customInfo.lastLoadedLevel = currentLevel;
```

And before the user logs out, perhaps you want to store that value – it will be done automatically once we upload accountInfo to the database just like we did before:

C#:
```
accountInfo.TryToUpload ( onUploadAttempted );
```

Then, you can use it when the user logs back in, by downloading the entire account info like we did before:

C#:
```
accountInfo.TryToDownload ( onDownloadAttempted  );
```

and then retrieving your custom info again

C#:
```
int levelToLoad = accountInfo.customInfo.lastLoadedLevel;
```

For any Queries / Suggestions,
reach me at mr.kon.vas@gmail.com