This tutorial walks you through the steps for creating Lua Script and interacting with the Component Entities through Event Bus (EBus), including how to create Component Entities, creating and assigning Lua Scripts, and working with EBus to execute events that interacts with our Component Entities. At the end of this tutorial, you will have learned how to connect and use EBus through Lua Script.

You will learn how to do the following:

- Create and Prepare Component Entities for Lua Scripting
- Create Lua Script
- Sending and Receiving events with EBus

Step 1: Creating and Preparing Component Entities for Lua Scripting
The first step in the tutorial is to create Component Entities to manipulate through Lua Scripting.

To create the Component Entities and Prepare it for Lua Scripting

1) Open the Lumberyard Editor.

2) In the Welcome to Lumberyard Editor screen, click the New level button.
   a. If the welcome screen isn’t displayed, click File > New.
3) In the **New Level** dialog, name your level *LuaScripting10_EBusTutorial* and set the Heightmap Resolution to 128x128, as shown below.

![New Level Dialog](image)

4) Right-click on the ground plane of your level and select **Create new component entity**.

![Create New Component Entity](image)
5) With the newly created component entity selected, navigate to the Entity Inspector.
   a. If the Entity Inspector window is not visible or part of your UI layout, you can find it in Tools > Entity Inspector.

6) On the Entity Inspector, click on Add Component... > Rendering > Static Mesh.
7) A Static Mesh component should appear in the Entity Inspector window, under the Transform component.

8) Next, add a static mesh called transform_helper under the Static Mesh Component > Static asset by clicking on the “…” button or Browse.

9) In the resulting Preview window, navigate to Objects > default, select transform_helper.cgf and hit Open. You should now see transform_helper in the Static asset slot.
10) Scale up the `transform_helper` mesh and also move it along the Z-Axis to get the mesh out of the ground with Scale and Transform tools.

11) For this demo, we will need two static meshes, in which case, duplicate the component entity we've just created by selecting the entity and hitting Ctrl + D.
   a. Alternatively, you can open up the **Entity Outliner** window via **View > Open View Pane > Entity Outliner**, right-click on the existing entity and selecting **Duplicate**.

![Entity Outliner window](image)

12) Move the duplicated entity away from the original entity and change the static mesh to `primitive_cube` by browsing for `primitive_cube` where we added `transform_helper` in Steps 8 and 9.

13) Lastly, with the `primitive_cube` component entity selected, add a Lua Script Component by going back to **Entity Inspector > Add Component… > Scripting > Lua Script**

![Entity Inspector window](image)
You have now created **Component Entities** and prepared it to interact with **Lua Scripts**.
Step 2: Creating the Lua Script

Now that we have our Component Entities prepared, let’s go ahead and create the Lua Script, which will drive our Component Entity events.

To create Lua script

1) Open Lumberyard’s internal Lua Editor by going to the Lua Script Component added to the `primitive_cube`, at the end of Step 1 and clicking on the “{ }” button.
   a. Alternatively, you can find the executable for this software located in the versioned Bin64 folder under your Lumberyard dev folder as `LuaIDE.exe`.
      i. For instance, if you built a profile for Visual Studio 2013, you would find this executable in `/Bin64vc120/`
2) Once the **Lua Editor** launches, make sure the Target and Debugging buttons show green indicators/icons.

   a. If the indicators are not green, click on the Target button and select Editor. Debugging button should automatically switch to the “ON” state with a green indicator.

3) Within the **Lua Editor**, select either the New File button or navigate to **File > New**.
4) Next, a **Save File** window should appear. You can save this file wherever you’d like.
   a. For this demo, we’ll create a folder called **luascripts** under
      /dev/SamplesProject/Levels/LuaScripting10_EBusTutorial/, and save our new Lua file
      with “tutorial_tick” as the file name.

   ![Save File Window]

   b. Once complete, you should notice that script file open and available for you to edit.

5) Within your Lua Script, create a local table to act as your primary table for this Lua Script file.
   a. You can do this with the line:
      ```lua
      local tutorial_tick = {}
      ```

   b. This creates a local table called, “tutorial_tick”. The reason for that particular name is
      because today we will be demonstrating EBus usage with the Tick Bus, but ultimately
      the name is up to you.

6) Next up, we’ll need to expose values to Users within Lumberyard so that they can manipulate
   variables without having to edit the Lua Script itself.

7) Within our primary local table, let’s create the sub-table called, **Properties**.
   a. Please note this name is important and must be spelled exactly the same in order for
      this lua script to work properly.
   b. After you have added the properties sub-table, your local table should look like this:
c. This local table’s, “Properties”, will expose all valid sub-types created within it to the Lua Script Component in Lumberyard. We'll return to this table later.

8) Now, we will need to return this primary local table at the end of the Lua Script file.
   a. This is done simply by adding the line: `return tutorial_tick` to the end of your Lua Script File.
   b. This is outside of the table itself and when done properly, your Lua Script should now look like:

   ```lua
   local tutorial_tick = 
   [ ]
   [ ]
   [ ]
   [ ]
   [ ]
   [ ]
   return tutorial_tick
   ```

9) After your primary local table is defined but before you return that table, add the following functions:
   a. `function tutorial_tick:OnActivate()
      end`
      i. The OnActivate function will trigger as soon as Component Entity with the respective Lua Script & Lua Script Component spawns mid-game.
   b. `function tutorial_tick:OnDeactivate()
      end`
      i. The OnDeactivate function will trigger as soon as the Lua Script is shut down by the Component Entity with the respective Lua Script & Lua Script Component being deleted mid-game.

With those native functions in place, your Lua Script is now ready to edit so that it can begin manipulating other Components who are exposed to Lua in Lumberyard. Exposing functions to Lua from other areas of code is done through EBuses.
Step 3: Sending and Receiving events with EBus

For this section of tutorial, we’ll show you the 3 use cases for EBuses:

- Connecting to a Handler / Notification EBus so that a Script may receive messages from other areas in code
- Sending out Event functions to a specific Component Entity.
- Sending out Broadcasts so that all Components Entities receive function calls.

To demonstrate these 3 cases, we’ll be using the Tick (gameplay clock) and Transform EBuses.

Let’s start off by setting up our connection to the Tick EBus

1) The first thing we’ll need to do is connect to the Tick EBus. Doing so will allow our Lua Script to hear back from C++ code and allow us to perform an action when a Gameplay Tick occurs. Although framerate dependent, these Ticks generally happen 30 Ticks to a single real life second.
   a. To connect to the Tick EBus, write the following call in your primary local table’s OnActivate function:
      
      ```lua
      TickBus.Connect(self, 0)
      ```
   b. This will connect us to the TickBus, ensuring that this Lua Script (designated by ‘self’) will hear back from C++ code whenever a Tick occurs with a priority of 0, which is designated by the number 0 in `TickBus.Connect(self, 0)`. Lua Scripts connected to that Bus with a value greater than 0 will receive their Tick ping after the Lua Scripts connected to the bus with priority 0.
      i. Essentially, the greater the priority number, the lesser it’s priority. Think of this as executing a specific event or events for every new number you read starting from 0 and onward.

2) Now, just connecting to the TickBus isn’t enough. Without a solid reference in code, this connection will be thrown out as soon as Lua performs data garbage collection. This garbage collection happens fairly frequently, but with some random variance.
   a. To store this connection to TickBus, we’ll need to add the following line of code before the TickBus.Connect call.
      ```lua
      self.TickBus =
      ```
   b. This should result in the following code:
self.TickBus = TickBus.Connect(self, 0)

c. This will store the TickBus connection inside your primary local table’s variable of, "TickBus".

3) Now that we are connected to the TickBus, you can now declare an OnTick function for your Lua Script that will get pinged every gameplay Tick. So, similar to what you did for creating the OnActivate and OnDeactivate functions for your primary local table, add the following line to your Lua Script between those 2 aforementioned functions:

```
function tutorial_tick:OnTick(deltaTime, timePoint)
end
```

4) Because your Script is receiving this OnTick message from C++ code, it also comes with the two variables deltaTime and timePoint.
   a. Each EBus you Connect to receives call backs from elsewhere in C++ code that have their own unique data associated with them, so please be aware beforehand of what information you’ll be getting back from the EBuses you are connecting to.
5) Now that we’ve established the connection to the Tick EBus, we next need to make sure we properly Disconnect from it when our Script is deactivated.
   a. To do so, simply write the following line into your primary local table’s OnDeactivate function:
      
      ```lua
      self.TickBus:Disconnect()
      ```
   b. As you may recall, we stored our connection to the Tick Bus within our own variable, TickBus. So now we merely call that variable and request it to perform its Disconnect function in the event of our Lua Script deactivating.
      i. Failure to disconnect from Ebuses you are connected to will result in odd and broken behavior, so please ensure that you always disconnect from an Ebus whenever you no longer have need for hearing back from it.

Now that we’ve properly setup a connection to the Tick EBus and can perform actions over time; let’s go ahead and use the Transform Ebus to rotate our two Component Entities every Gameplay Tick.

1) Before we get started pinging the Transform Ebus, let’s add a new variable to our primary local table’s Properties sub-table. The Properties sub-table will expose itself to the Lua Script Component in the Entity Inspector, allowing us to dynamically change our rotation rates from Entity Inspector rather than having to continuously alter our Lua Script further down the road.
   a. To add this new variable, simply write it out in the Properties sub-table like so:
      
      ```lua
      SpinRate = 1.0
      ```
   b. This will create the variable, “SpinRate”, with a default value of 1.0. Once this is added, your initial primary local table should look like so:
2) Next, we'll send an Event on the Transform Bus to change the rotation of the Component Entity that is currently hosting this Lua Script every Tick.
   a. We do this by adding the following line of code to the primary local table’s OnTick function:

```
TransformBus.Event.RotateByZ(self.entityId, self.Properties.SpinRate * 0.001)
```

3) **Transform EBus** sends an Event call for the **RotateByZ** function, rotating your specific Component Entity by a rate of SpinRate * 0.001. The reason for the 0.001 is that we will be doing this for every Tick and so want to keep each rotation fairly small to compensate for how much rotating we’ll be doing.
   - Any time you send an Event, the function you are pinging will require the additional parameter of what Component Entity (represented by an `entityId`) you are asking to perform this action on before all its standard parameters.
   - All EBus(es) you’ll directly communicate to with Events or Broadcasts will have a set amount of functions associated to it, reflected to Lua from elsewhere in C++ code.
   - While Lumberyard’s own **Lua Editor** can show you these reflected functions in the Classes Reference section, if you are using an external Lua Script editor, please consult your Project’s additional code base for reference on what functions are exposed to Lua.

4) Now we’ll want to demonstrate the EBus’ Broadcast functionality.
   a. Right after our Event call, **TransformBus.Event.RotateByZ**, add the following line of code:

```
TransformBus.Broadcast.RotateByZ(self.Properties.SpinRate * 0.001)
```
   b. **Transform EBus** will Broadcast out to all Transform Components in your game, the command to **RotateByZ** by the rate of SpinRate * 0.001.
5) With that last line of code, your Lua Script should be fully complete and so now let’s return to Lumberyard to see it in action!
   a. Once back in Lumberyard, go back to the **primitive_cube** Component Entity, you earlier assigned the Lua Script Component to and ensure that it now references your Lua Script.
   
   b. If it does not, simply browse for the Lua Script from the path we saved out to, when we opened the **Lua Editor**.
      i. For this tutorial, the path was:
         
         /dev/SamplesProject/Levels/LuaScripting10_EBusTutorial/luascripts/tutorial_tick.luac

ii. Your Lua Script Component should now look like so:

![](image1)

6) Run the game and you should see both Component Entities slowly rotating, with the Component Entity hosting the Lua Script, rotating at double the rate of the Component Entity that lacks the Lua Script Component.
   
   a. This is because that while the Component Entity without the Lua Script Component is receiving a singular Broadcast command to rotate; the Component Entity with the Lua Script Component is receiving 2 commands to rotate: one command from the EBus Event and the other from the EBus Broadcast.
   
   b. To increase the rotation rate of your objects, go to the Component Entity that has the Lua Script Component on it and you should notice an additional field in it: SpinRate. This is the SpinRate variable we created earlier within the Properties sub-table. Increase or decrease this number in the Lua Script Component to cause faster or slower rotations respectively.
Congratulations! You have successfully interacted with Component Entities through EBus via Lua Script.

Want to learn more about Lumberyard? Visit the complete Lumberyard Tutorials Page.

We’d love to hear from you! Head to our Tutorial Discussion forum to share any feedback you have, including what you do or don’t like about our tutorials or new content you’d like to see in the near future.