UGESystem Custom Command Guide

> 🛑 **[CRITICAL WARNING] Data Loss on Package Update**
>
> This guide involves **modifying core scripts** (e.g., `CommandType.cs`, `UGEGameEventController.cs`) inside `Assets/UGESystem/` and **creating new files** within the package folder.
>
> **Updating the UGESystem package will RESET (DELETE/OVERWRITE) all changes in this folder.**
>
> - **New Files**: **BACK UP** your files before updating, or create them outside the package folder (`Assets/MyGame/Scripts/`) from the start.
> - **Core Modifications**: You MUST **re-apply these changes** manually after every package update.

This guide explains how to add new commands to the UGESystem to fit your project requirements.

> 📢 **IMPORTANT NOTICE: Web Story Maker**
>
> - The **Web Story Maker** is a complimentary service provided by the developer for user convenience.
> - Users **cannot modify or customize the source code** of the web tool.
> - Therefore, **custom commands added via this guide will NOT appear and cannot be edited in the Web Story Maker.**
> - Custom commands can **ONLY be configured and edited within the Unity Editor's Inspector window.**

Goal: What are we building?

As an example, we will create a command called **`DebugLogCommand`** that prints a message to the Console window while the game is running.

Step 1: Register Command Type

First, we need to tell the system that a new command type exists by giving it a unique name.

1. **Open File**: `Assets/UGESystem/Core/Scripts/UGESystem/GameEvents/Enums/CommandType.cs`
2. **Action**: Add a comma (`,`) after the last item in the list, then add your new name.

```
public enum CommandType
{
    // ... existing items ...
    TriggerEvent, // Make sure there is a comma here!

    // [Add] New command name
    DebugLog
}
```

Step 2: Define Data Structure (Command Class)

Create a script to hold the settings (data) for your command, such as the message text.

1. **Folder**: `Assets/UGESystem/Core/Scripts/UGESystem/GameEvents/Data/Commands/` (or your custom folder)
2. **Create File**: `DebugLogCommand.cs` (C# Script)
3. **Write Code**: Copy and paste the code below.

```
using UnityEngine;
using System;

namespace UGESystem
{
    [Serializable] // Required for saving/loading.
    public class DebugLogCommand : EventCommand
    {
        // 1. Data field to be edited in the Unity Inspector
        [SerializeField] public string LogMessage;

        public DebugLogCommand()
        {
            // 2. Link the name registered in Step 1
            CommandType = CommandType.DebugLog;
        }

        // Returns null because the Web Tool does not support custom commands.
        public override IEventCommandDto ToDto()
        {
            return null;
        }
    }
}
```

Step 3: Implement Execution Logic (Handler Class)

Define **what actually happens** when this command is executed in the game.

1. **Folder**: `Assets/UGESystem/Core/Scripts/UGESystem/GameEvents/Managers/Runners/Handlers/` (or your custom folder)
2. **Create File**: `DebugLogCommandHandler.cs` (C# Script)
3. **Write Code**: Copy and paste the code below.

```
using System.Collections;
using UnityEngine;

namespace UGESystem
{
    // Implement ICommandHandler so the system recognizes this script.
    public class DebugLogCommandHandler : ICommandHandler
    {
        public IEnumerator Execute(IGameEventCommand command, UGEGameEventController controller)
        {
            // 1. Cast the generic command to our specific type (DebugLogCommand).
            var cmd = command as DebugLogCommand;

            if (cmd != null)
            {
                // 2. Write the actual logic here. (e.g., Print log)
                Debug.Log({{content}}quot;[UGESystem Log] {cmd.LogMessage}");
            }

            // 3. Wait for one frame or finish immediately.
            yield return null;
        }
    }
}
```

Step 4: Register to System Controller

Finally, connect your new Data (`Command`) and Logic (`Handler`) to the main system (`Controller`). This does not change existing logic but simply registers the new parts.

1. **Open File**: `Assets/UGESystem/Core/Scripts/UGESystem/GameEvents/Managers/UGEGameEventController.cs`
2. **Edit Code**: Find the `InitializeCommandHandlers` function and add the lines marked below.

```
private void InitializeCommandHandlers()
{
    // ... existing handler creation ...
    var triggerEventHandler = new TriggerEventCommandHandler();

    // [Add 1] Create an instance of your handler
    var debugLogHandler = new DebugLogCommandHandler();

    _commandHandlers = new Dictionary<GameEventType, Dictionary<Type, ICommandHandler>>
    {
        {
            GameEventType.Dialogue, new Dictionary<Type, ICommandHandler>
            {
                // ... existing registrations ...
                { typeof(TriggerEventCommand), triggerEventHandler },

                // [Add 2] Map the data type to the handler
                // "When DebugLogCommand data arrives, let debugLogHandler handle it."
                { typeof(DebugLogCommand), debugLogHandler },
            }
        },
        // Optional: Add to CinematicText mode if needed
        {
            GameEventType.CinematicText, new Dictionary<Type, ICommandHandler>
            {
                // ... existing registrations ...
                // [Optional] Add here if you want it to work in cinematic mode too
                { typeof(DebugLogCommand), debugLogHandler },
            }
        },
    };
}
```

Step 5: Verify

1. Return to the Unity Editor and wait for compilation to finish.
2. Right-click in the Project window > `Create` > `UGESystem` > `New Game Event`.
3. In the Inspector window, click the **Add (+)** button in the `Commands` list. You should see `DebugLog` in the list.
4. Enter a message and run the game to see the log in the Console window.