Task Automator

by Ramdal Games



Thanks

First, thank you so much for purchasing Task Automator. Your support is truly appreciated. A lot of time, care, and effort went into building it, and that support means a lot. Hopefully, you find it useful.

Contents

BRIEF **4**

SETUP **5**

IMPORT TASK AUTOMATOR 5

GET STARTED **6**

TASK AUTOMATOR **7**

QUICK OVERVIEW **8**

PRESETS **9**

TASKS EXECUTION **10**

DEFAULT TASKS **11**

FILES 11

1) Clone Files: *11*

Use cases: 11

2) Create Files: *12*

3) Delete Files: *12*

4) Move Files: *12*

PACKAGES 13

1) Import Package *13*

2) Remove Package: *13*

PROJECT SETTINGS 14

1) Change Input Type: *14*

2) Change Project Identity: *14*

CREATE YOUR TASKS **15**

WHAT A CUSTOM TASK NEEDS 15

Example: 16

REQUIRED API 17

1) Execute (TaskProgress progress) *17*

2) Progress (TaskProgress progress) *17*

3) TaskResult *17*

Example: 18

4) TaskProgress API *18*

progress.TaskName 18

progress.StartTime 18

progress.ElapsedTime 18

progress.AppendLog(...) 18

Survivable State 19

Substates 20

OPTIONAL API 21

1) MenuItemPath *21*

2) OnResume *21*

3) OnAllTasksFinished *22*

4) ScheduleRestart *22*

5) HasScheduledRestart *22*

6) CancelScheduledRestart *22*

OPTIONAL CUSTOM UI 23

Example: 23

SUPPORT AND USEFUL LINKS **24**

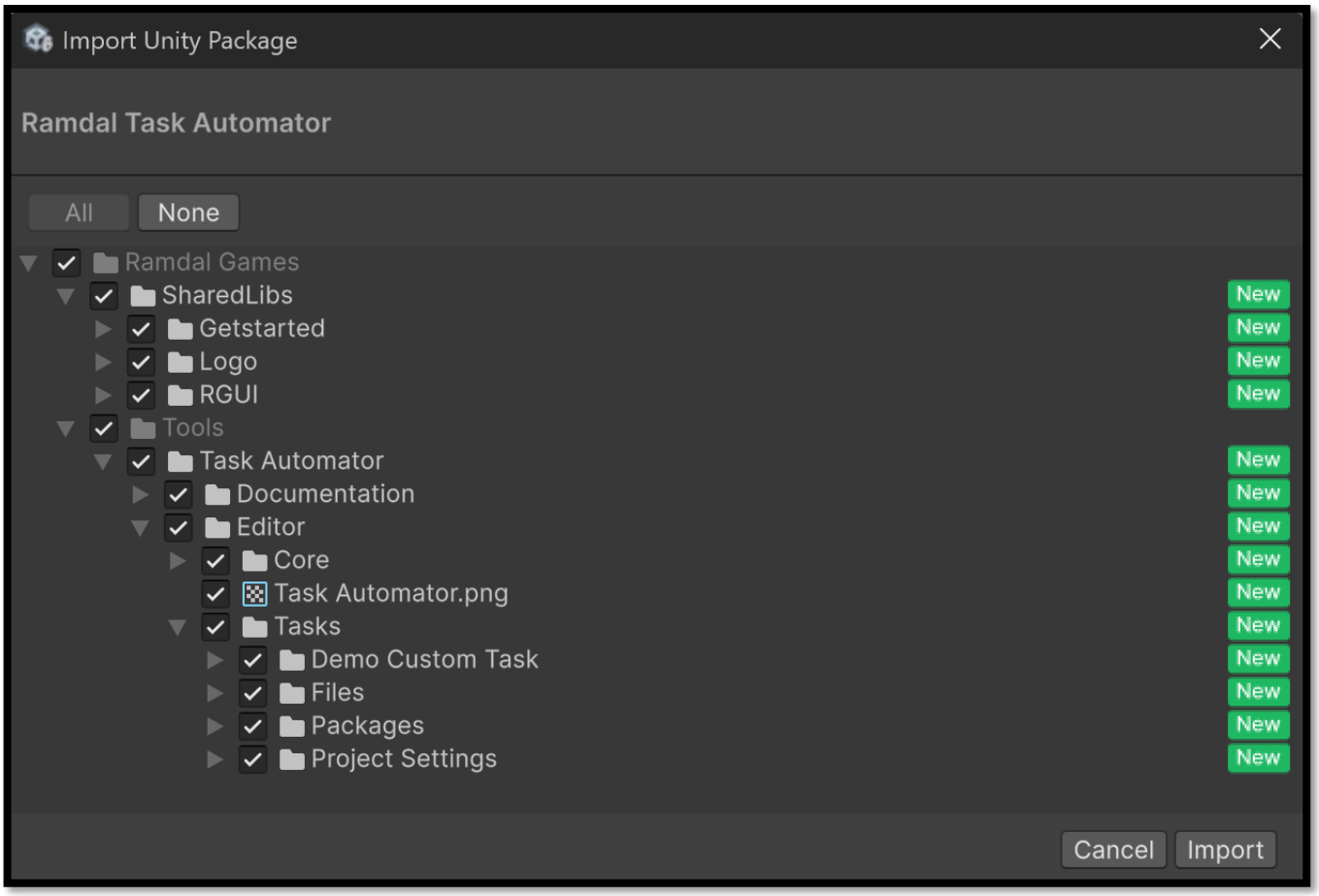
Brief

Task Automator lets you automate repetitive tasks and workflow steps in Unity by turning them into reusable presets that can be run in just a few clicks. By reducing manual work, it saves time and cost, lowers the chance of mistakes, keeps you focused on your main work, and can be expanded with your own custom tasks.

Setup

Import Task Automator

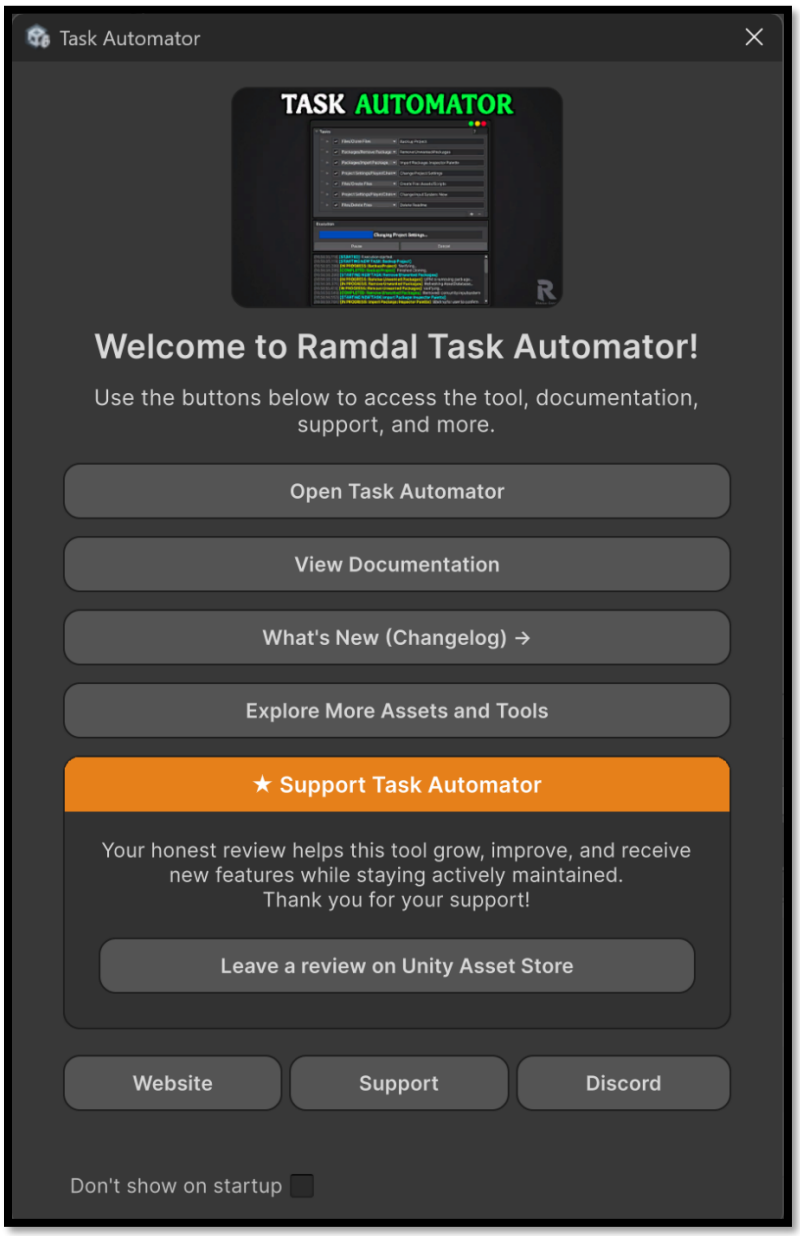
Download the asset, then import everything, and you should be good to go!



Get Started

You should see the Get Started popup automatically.

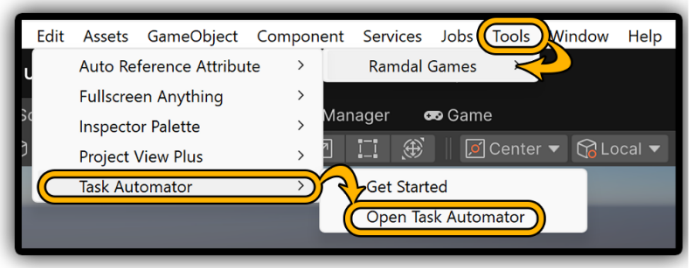
If you do not, you can always open it from **Tools > Ramdal Games > Task Automator > Get Started**.

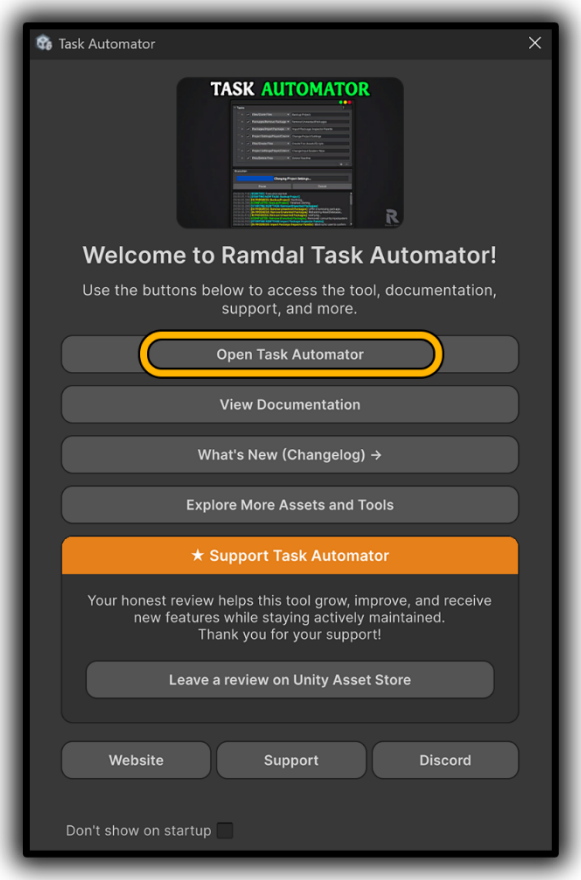


- It can help you access useful links and documents.
- You can disable it from showing automatically on startup or remove it completely.

Task Automator

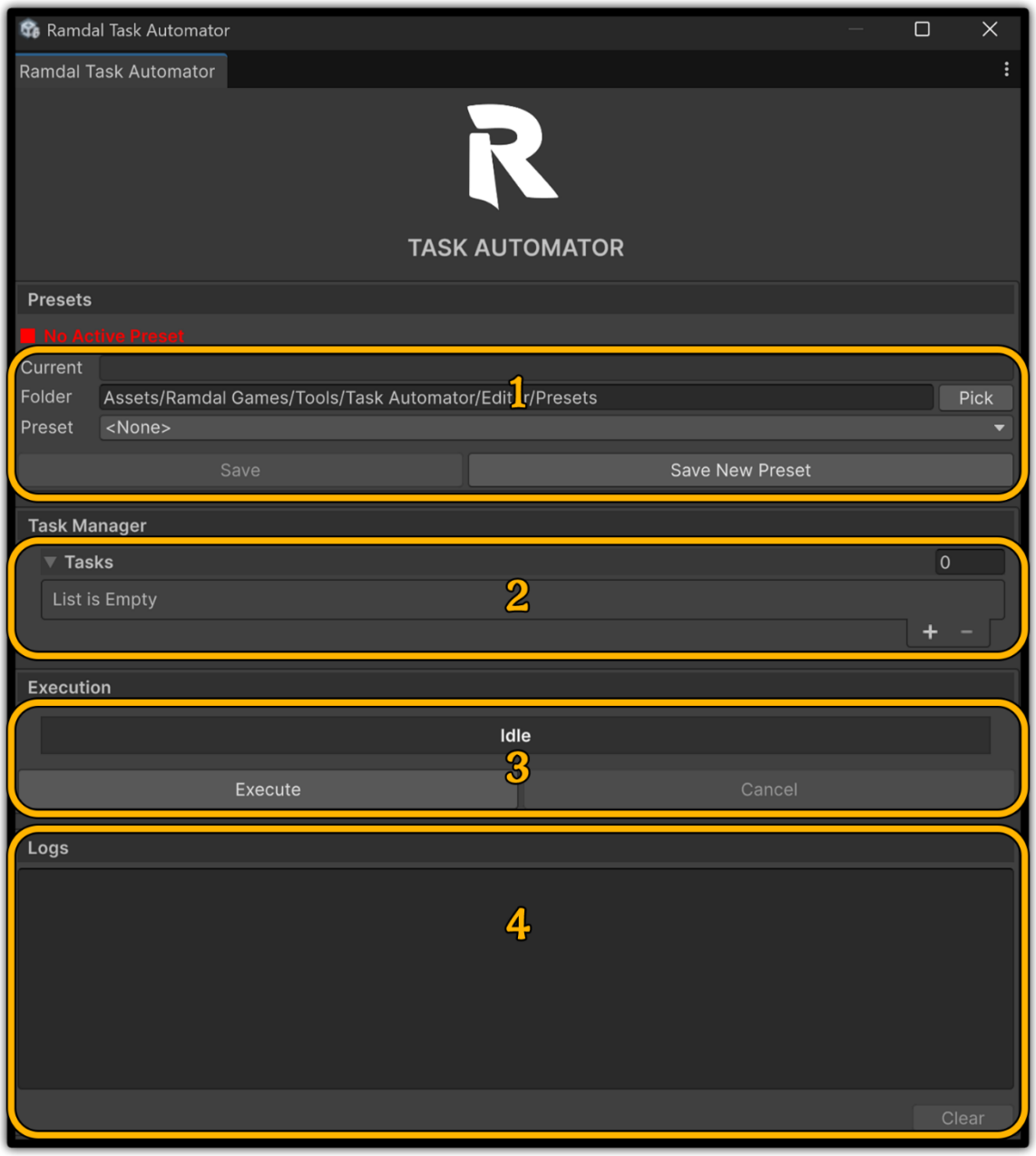
You can open Task Automator by clicking Open Task Automator in the Get Started window, or from **Tools > Ramdal Games > Task Automator > Open Task Automator**.





Quick Overview

1) **Presets**: lets you create, load, save, and switch presets easily.
2) **Task Manager**: contains all the tasks that will be executed.
3) **Execution**: lets you start, manage, and track task execution.
4) **Logs**: shows the full progress of the last job, helping you identify any problems during execution.



Presets

By default, Task Automator comes with no presets, and you can easily create your own:

1) Select where you want to save the presets.

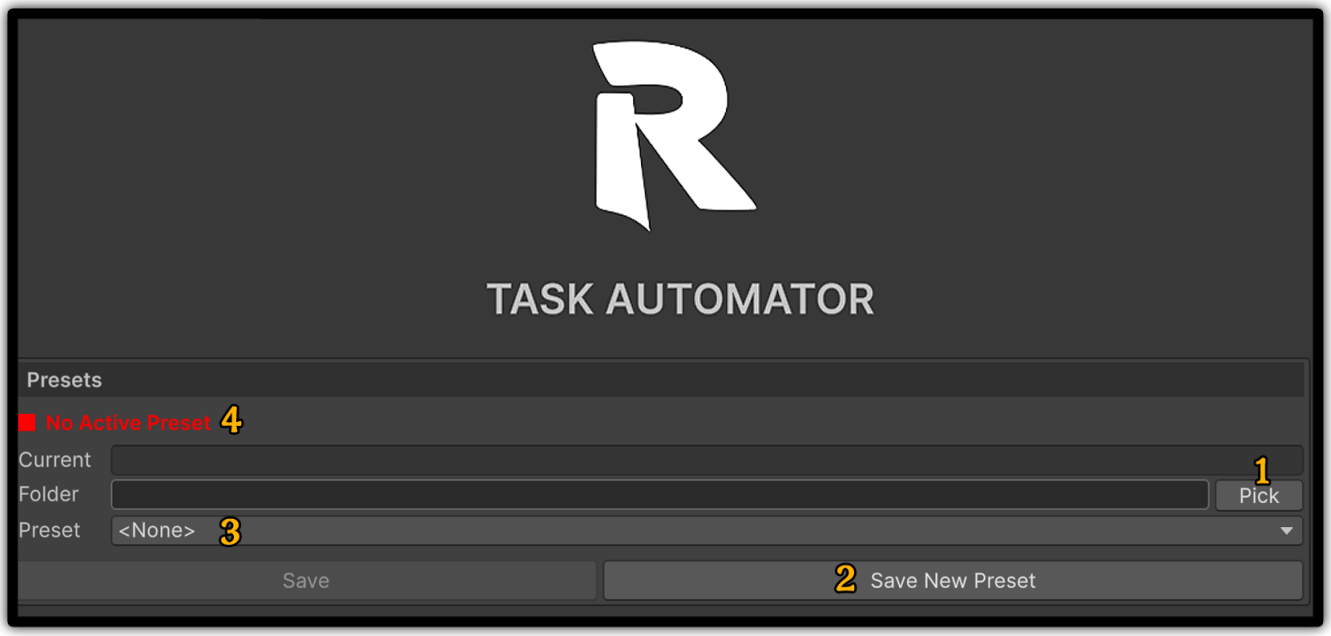
2) Click Save New Preset to save your current setup.

3) Switch between saved presets easily.

4) An indicator always shows whether your preset is saved or has unsaved changes, so make sure to save it.

Notes:

- By default, presets are saved as JSON files, so you can easily preview, edit, or share them across multiple projects and users.
- If you have unsaved changes and try to switch presets, Task Automator will ask whether you want to save first so you do not lose your current setup.



Tasks Execution

Out of the box, Task Automator comes with multiple handy tasks that can help automate daily setup and shared workflows.

1) Add tasks by clicking the + button.

2) Choose the task you want.

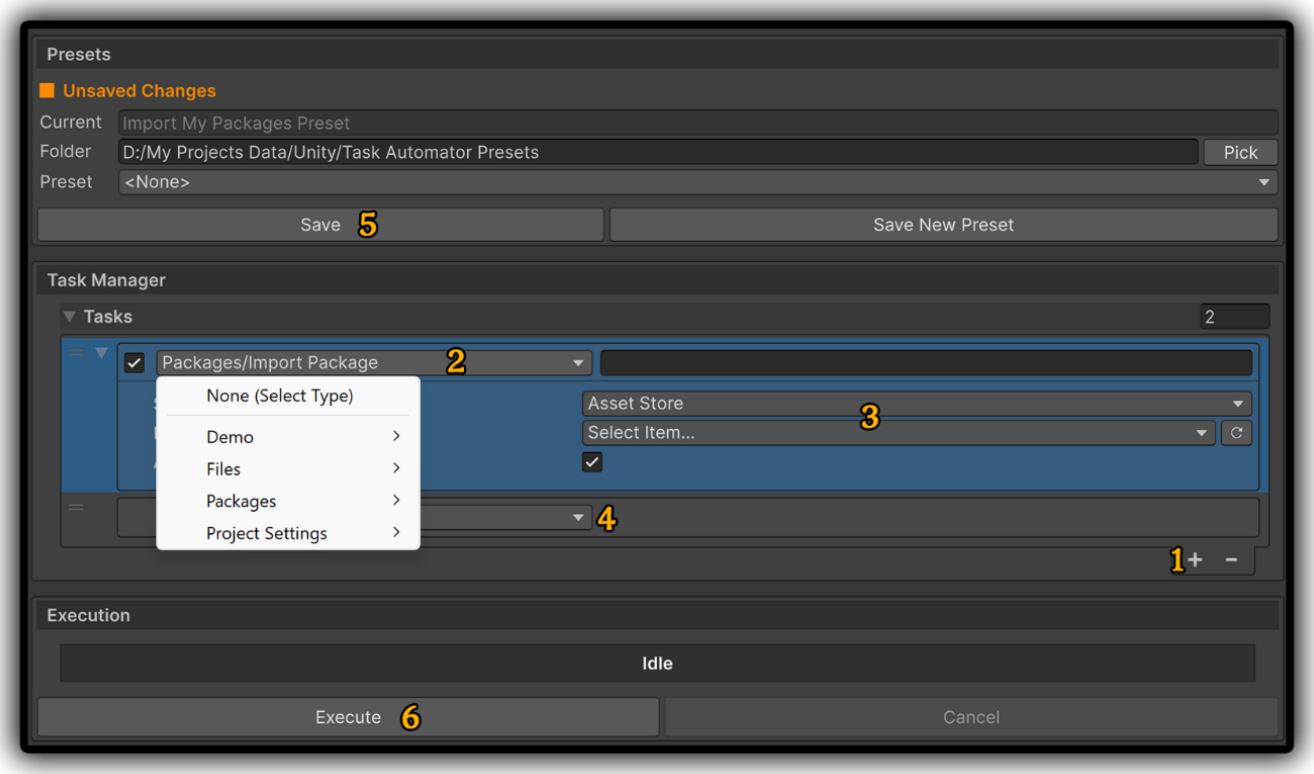
3) Configure it.

4) Repeat the same for each task you want to add.

5) Save your configuration.

6) When you are ready, click Execute.

- All enabled tasks are executed synchronously from top to bottom.
- You can track each task status in real time through the logs and progress bar.
- If a restart is prompted during execution, you can safely accept it. The job will resume automatically after all tasks have been completed.



Default Tasks

By default, Task Automator comes with multiple useful tasks:

Files

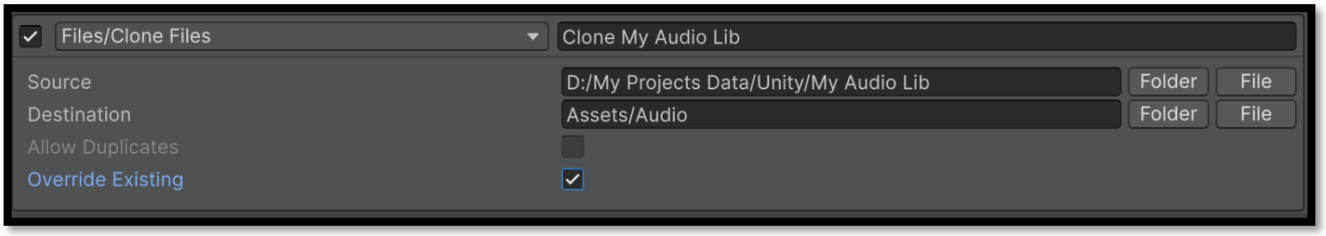
Allows you to perform file and folder related tasks.

1) Clone Files:

- Allows you to clone files, folders, and assets from one place to another.
- During cloning, the tool skips all .meta files.
- You can allow duplicates in the destination, override existing files, or skip the operation if neither option is enabled.

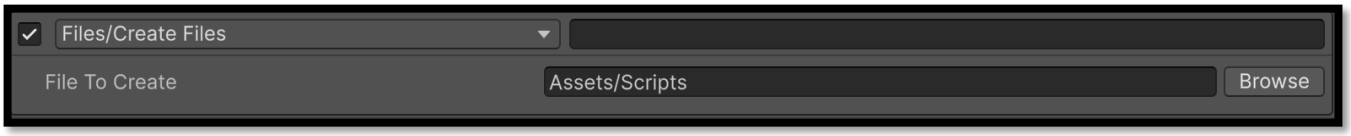
Use cases:

- Clone a project structure from one project to another.
- Create a backup copy of specific elements or an entire project.



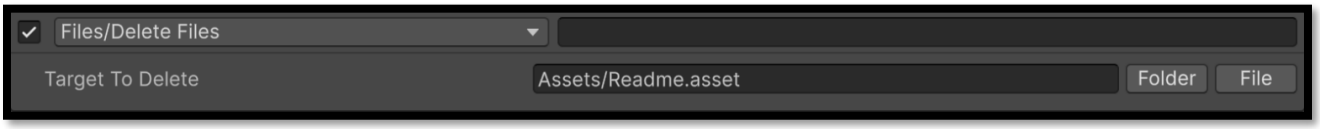
2) Create Files:

- Allows you to create new directories and folders.
- If a file or folder with the same name already exists, it will be skipped.



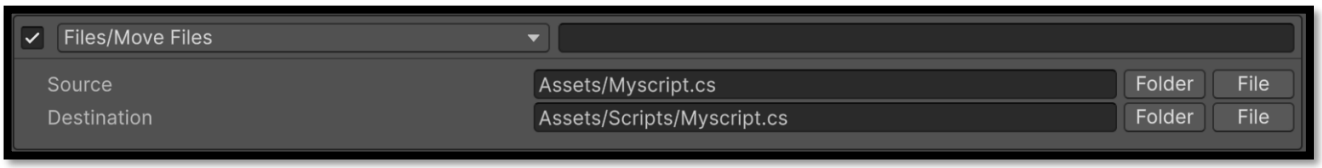
3) Delete Files:

- Allows you to delete the selected file or folder.
- If no valid path is provided, it will be skipped.



4) Move Files:

- Allows you to move files or folders.
- If no valid path is provided, it will be skipped.



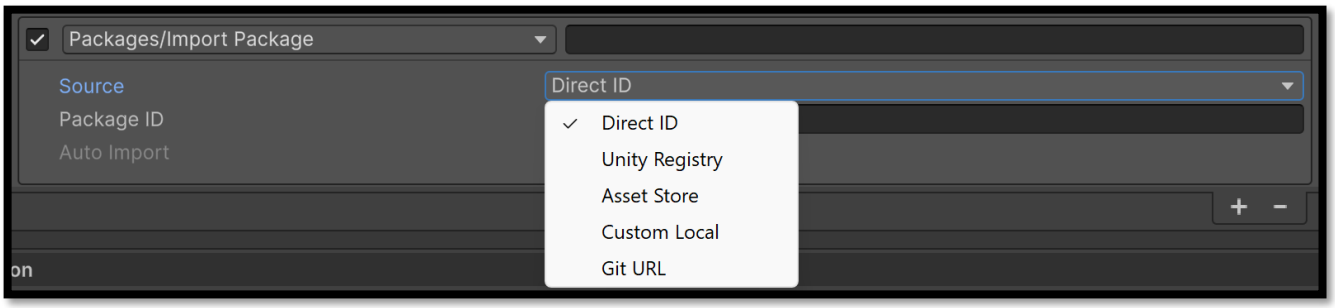
Packages

Allows you to perform package-related tasks.

1) Import Package

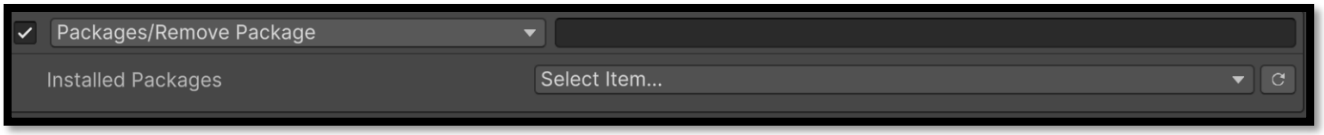
You can import packages from different sources:

- **Direct ID:** import a package using its package ID.
- **Unity Registry:** import packages from the Unity Registry.
- **Asset Store:** import any downloaded Asset Store assets you own.
- **Custom Local:** import local packages from your disk.
- **Git URL:** import and clone packages directly from Git using a URL.



2) Remove Package:

- You can remove all imported and registered packages from the manifest.



Project Settings

Allows you to change project settings.

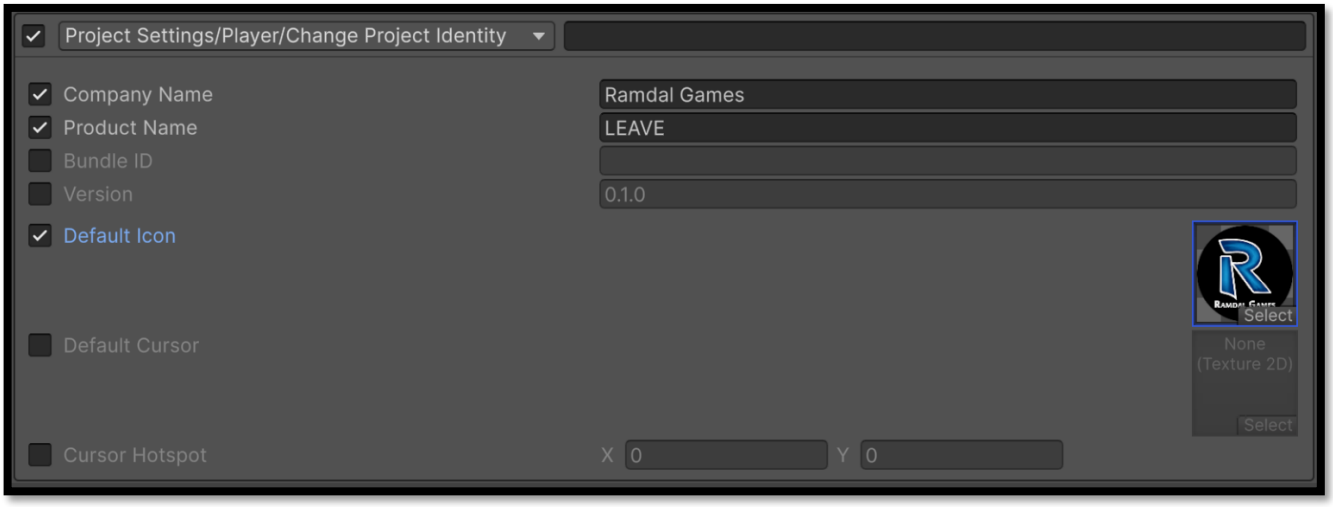
1) Change Input Type:

- Allows you to change the current input handler to Old, New, or Both.
- Automatically imports and removes packages and dependencies.
- If Unity asks you to restart the editor, you can safely accept. Otherwise, it will prompt you to restart the editor after all tasks have been executed.



2) Change Project Identity:

- Allows you to change main project settings such as Company Name, Product Name, Bundle ID, Version, Default Icon, Default Cursor, and Cursor Hotspot.
- Changes are applied only to the enabled elements.



Create Your Tasks

Task Automator fully supports creating and executing your own custom tasks out of the box.
It is designed from the ground up to be expandable, so creating your own tasks is a core part of how the system works.

Inside the package, you will find a full example called **DemoTestTask.cs**.
This example shows the complete flow of a custom task, including:

- How a task starts.
- How it keeps running over time.
- How it survives reloads and recompiles.
- How it resumes correctly.
- How to return progress, completion, or failure.
- How to create a custom drawer for the task UI.

What a Custom Task Needs

A custom task is simply a class that:

- is marked as **[Serializable].**
- inherits from **Task**.
- implements the required task methods.

Example:

```
using System;
using UnityEngine;

namespace RamdalGames.Tools.TaskAutomator {
    [Serializable]
    0 references
    internal class MyCustomDemoTask : Task {
        [SerializeField] private string myCustomField = "Hello World!";
        [SerializeField] private float myCustomFloat = 99f;

        2 references
        public override void Execute(TaskProgress progress) {
            // Here you can implement the logic of your custom task.
        }

        2 references
        public override TaskResult Progress(TaskProgress progress) {
            // Here you can implement the logic to check the progress
            // of your custom task and return the appropriate TaskResult.

            return new TaskResult();
        }
    }
}
```



Required API

These are the required parts every custom task must implement.

1) Execute (TaskProgress progress)

```
public override void Execute(TaskProgress progress) { }
```

- Execute is where you run your main task logic.
- Called once when the task starts.
- Use it to begin your task, prepare data, add logs, and create any survivable state you need.

2) Progress (TaskProgress progress)

```
public override TaskResult Progress(TaskProgress progress) { }
```

- Called repeatedly while the task is running.
- Use it to keep the task moving, check its state, and return whether it is still running, completed, or failed.
- Once you return TaskStatus.Completed or Failed the task will be marked as Finished.

3) TaskResult

- Returned by Progress() to tell Task Automator the current task state.
- Main statuses you will usually return:
  - TaskStatus.InProgress
  - TaskStatus.Completed
  - TaskStatus.Failed

Example:

```
return new TaskResult (TaskStatus.InProgress, "Running...");
return new TaskResult (TaskStatus.Completed, "Task finished
successfully.");
return new TaskResult (TaskStatus.Failed, "Something went wrong." );
```

- You can also optionally override the verification time after completion or failure:

```
return new TaskResult (
    status: TaskStatus.Completed,
    additionalNotes: "Task finished successfully.",
    verificationTime: 1);
```

4) TaskProgress API

TaskProgress is passed into task callbacks and gives you access to task runtime data and helper functions.

progress.TaskName

Returns the task type name, or the custom task name if the user set one.

progress.StartTime

Returns the time when the task started, in EditorApplication.timeSinceStartup seconds.

progress.ElapsedTime

Returns how many seconds have passed since the task started.

progress.AppendLog(...)

Adds a log entry to the built-in execution log.

Parameters:

status

The log status label. You can use either a TaskStatus enum value or your own custom string status.

message

The main log message to display.

skipDuplicates

If true, repeated identical logs can be skipped. Default is true.

statusColor

Optional custom color for the status label using a hex color string.

Example:

```
progress.AppendLog(
    TaskStatus.InProgress,
    "Unity recompiled! Resuming Running task...",
    skipDuplicates: true,
    statusColor: "#00FFFF");
```

Survivable State

A survivable state is simply a persistent string key you add while the task is still executing.

Its role is to store persistent task states that you can check later while the task is still running, even across reloads, recompiles, resumes, or editor restarts.

It stays alive for the entire task execution, then gets cleared when the task finishes.

Create a survivable state

```
SurvivableState survivableState = progress.AddSurvivableState(key);
```

Check if a survivable state exists

```
bool hasState = progress.HaveSurvivableState(key);
```

Get an existing survivable state

```
SurvivableState survivableState = progress.GetSurvivableState(key);
```

Remove a survivable state

```
progress.RemoveSurvivableState(key);
```

Clear all survivable states

```
progress.ClearAllSurvivableStates();
```

Substates

- A substate is a smaller state stored inside a survivable state.
- It is useful for tracking extra details inside the main state, such as selected paths, temporary markers, handled steps, conditions, or progress flags.

Add a substate

```
SurvivableState survivableState = progress.GetSurvivableState(key);
survivableState.AddSubstate(key);
```

Check if a substate exists

```
bool hasSubstate = survivableState.HaveSubstate(key);
```

Remove a substate

```
survivableState.RemoveSubstate(key);
```

Clear all substates

```
survivableState.ClearSubstates();
```

Dump all substates

```
string allSubstates = survivableState.DumpAllSubstates(string separator = "");
```

Optional API

These are optional overrides you can implement when needed.

1) MenuItemPath

Lets you place the task under a custom path in the Add Task menu.

```
public override string MenuItemPath() => "Demo";
```

2) OnResume

- Called once every time Unity recompiles or reloads scripts while the task is still running.
- Use it if your task needs to restore logic or report that it resumed correctly.

```
public override void OnResume(TaskProgress progress)
```

3) OnAllTasksFinished

- Called once after all tasks have finished.
- Use it for final cleanup, restart handling or additional logic.

```
public override void OnAllTasksFinished()
```

4) ScheduleRestart

Schedules an editor restart that will execute after all tasks have finished.

```
ScheduleRestart();
```

5) HasScheduledRestart

Returns true if an editor restart is already scheduled.

```
bool scheduledRestart = HasScheduledRestart();
```

6) CancelScheduledRestart

Cancels the scheduled editor restart.

```
CancelScheduledRestart();
```

Optional Custom UI

You can also create a custom drawer for your task.

If you want to keep the default Task Automator header, inherit from **TaskDrawer.**

Example:

```
[CustomPropertyDrawer(typeof(DemoTestTask))]
internal class DemoTestTaskDrawer : TaskDrawer {
public override void OnGUI(…){
base.OnGUI(…); // To draw the default task header.
//Draws your custom task UI.
}
public override float GetPropertyHeight(…){
//Return the height of your custom drawer.
}
```

Support and Useful Links

Support: support@ramdalgames.com

Contact: contact@ramdalgames.com

Ramdal Games: https://ramdalgames.com

Mohamed Haftari: https://mohamedhaftari.com

Join Our Discord Community

Explore More Assets and Tools