uTomate Manual

Ancient Light Studios

Version 1.5.0
2015-03-17

Table of Contents

uTomate manual 1
- Overview 1
- Terminology 1

Automation plans 2
- Overview 2
- Creating automation plans 2
- Editing automation plans 2

Automation plan entries 4
- Overview 4
- Run Action 4
- Decision 5
- For Each 5
- For Each File 6
- Run Plan 7
- Note 7

Actions 8
- Creating actions 8
- Editing actions 8
- Expression mode 9
- File sets and wildcards 10

Expressions 12
- Overview 12
- Expression syntax 12
- Expression typing 13
- Runtime properties 14

The uTomate main window 15
- Overview 15
- Settings and plan view 16
- Runner view 17
- Project properties view 18
- Editor properties view 19

Advanced Topics 20
- Running uTomate from the command line 20
- Script extensions 22

Support of older Unity versions 24

uTomate manual

Overview

uTomate is an automation extension for Unity3D which allows you to automate many processes during development that otherwise require a lot of manual work. This manual is explaining uTomate in detail. If you just want a quick start we recommend that you watch our quickstart video. If you want more details about all the available uTomate actions have a look at the actions reference page. There is also a tutorials page which contains various tutorials that help you getting common tasks done quickly.

A quick start

Before going into the details of all of uTomate's options, let's start with a simple automation. First, create a new automation plan, by right-clicking in your project tree and selecting *Create uTomate Automation plan*. This will create a new automation plan. Now you can open the automation plan in uTomate's automation plan editor by right-clicking it and selecting *Edit in automation plan editor*. Dock the editor at some convenient. Next let's create a simple action which simply prints "Hello World". Right-click in the project tree and select *Create uTomate General Echo*. A new action will be created in the project tree. Select the action. The action will now be shown in Unity's inspector window. In the *Text* setting enter `Hello World!`. Now drag the action from the project tree into the automation plan editor. Congratulations - you have created your first automation plan! Run it by pressing the *Run this plan* button of the automation plan editor.

Terminology

Throughout this manual, the following terms are used:

Automation Plan

an automation plan controls the order in which actions are being performed. It consists of automation plan entries. Automation plans are edited in the uTomate automation plan editor window.

Automation Plan Entry

an automation plan entry is a control structure of an automation plan. An entry can be anything from as simple as "Execute a single action" up to "Execute another plan". There are also entries for making decisions or running things in a loop.

Action

an action is a piece of work that can be done independently. You can create actions and configure them in the Unity3D inspector window.

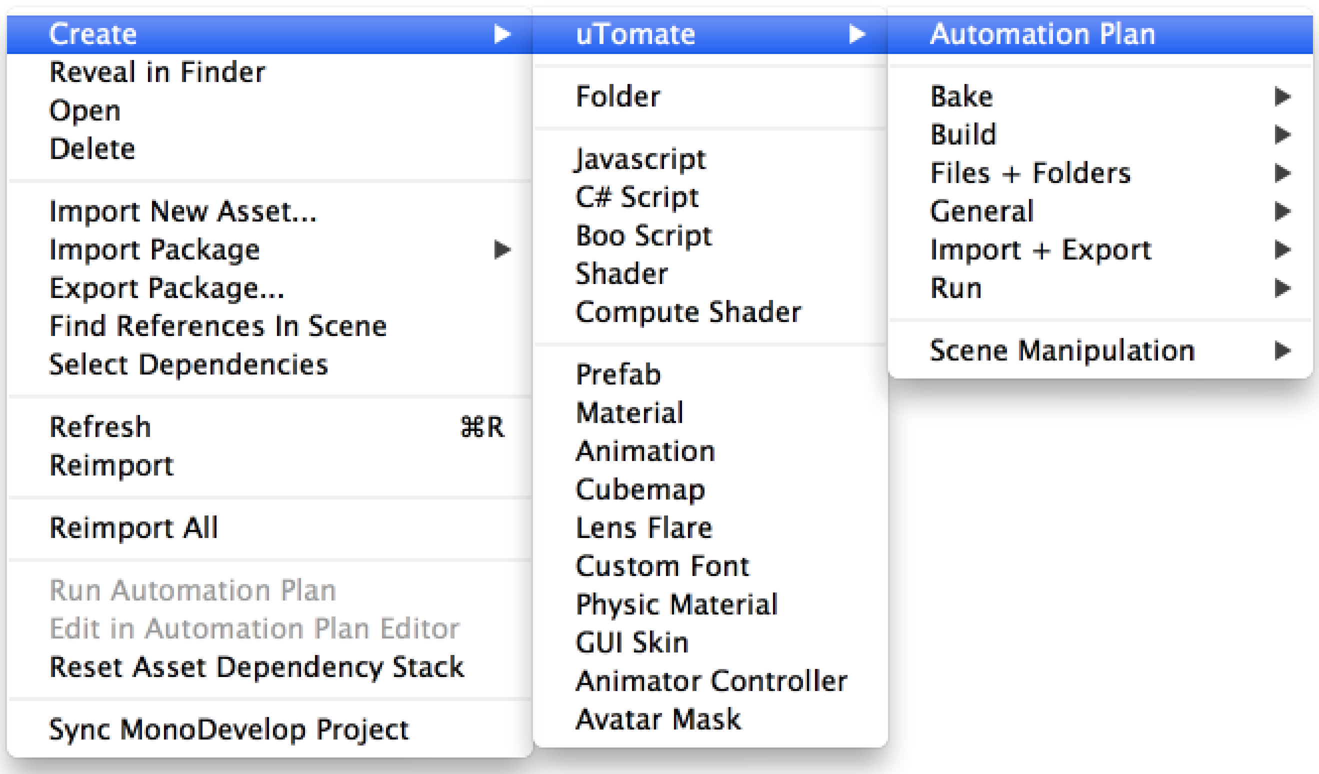
Automation plans

Overview

Automation plans consist of a set of automation plan entries. Using the entries you can control the order in which your actions are running. Each entry can be connected to another entry. When you run an automation plan, uTomate will start executing the entry which is marked as the first entry. Once an entry has finished executing, the entry that is connected to this entry as as *Next Entry* will be executed. Once an entry has no more next entry, the automation plan is finished. There are specialised entry types which allow you to add decisions or loops to your automation plan, therefore enabling you to visually script the flow of your automation.

Creating automation plans

To create an automation plan, right-click the folder in which you want to create the plan, then select *Create* *uTomate* *Automation plan.*

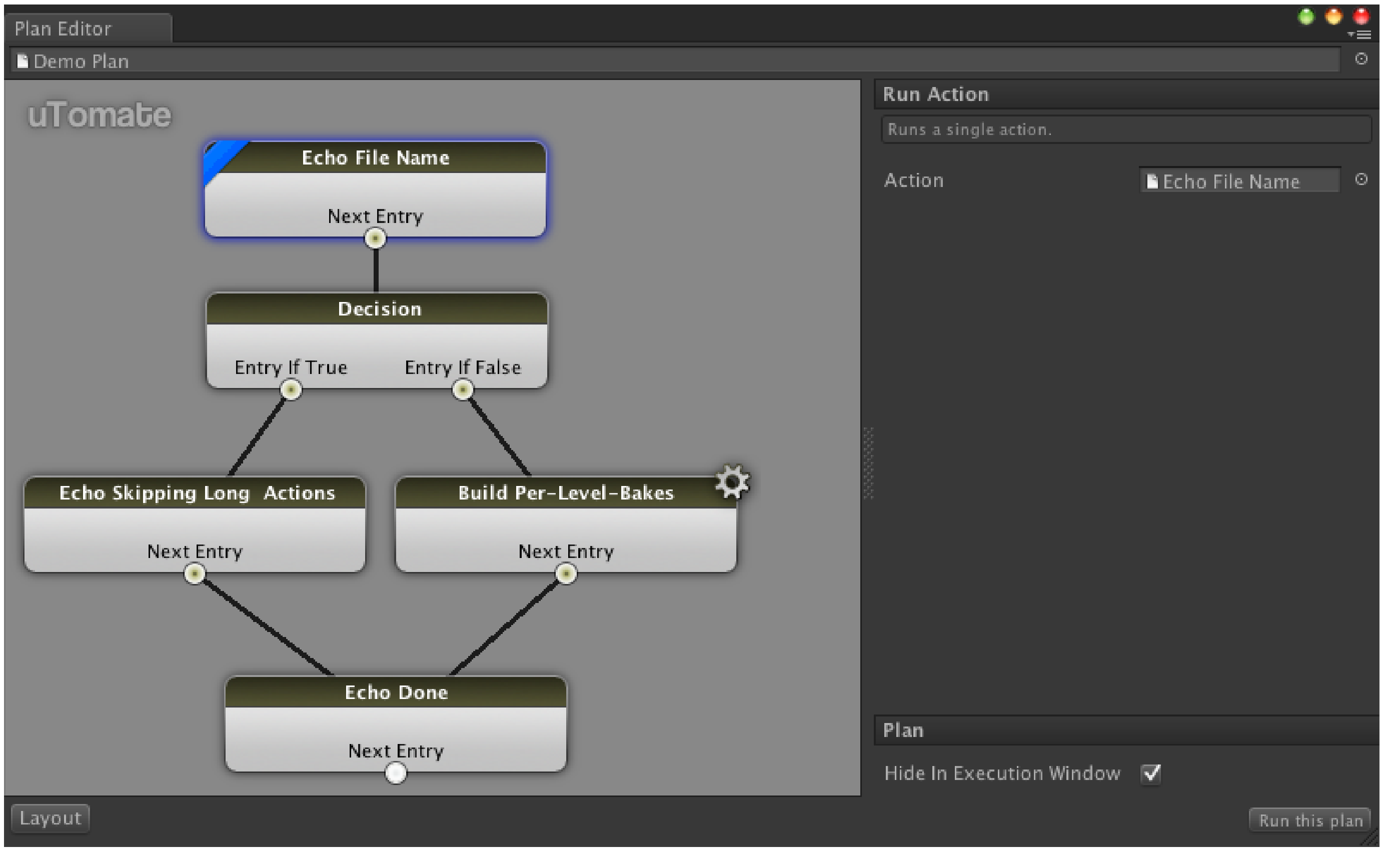


Alternatively you can use the main menu *Assets* *Create* *uTomate* *Automation plan.*

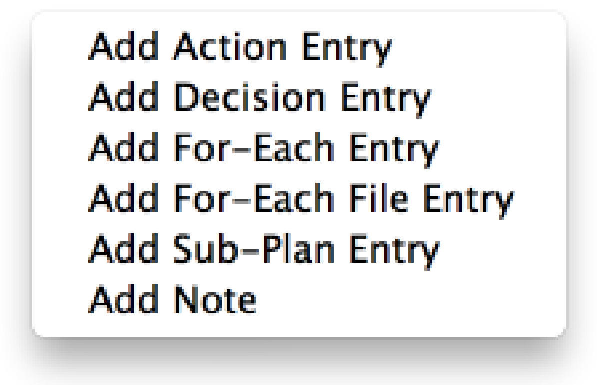
Editing automation plans

Automation plans are not edited in the Unity inspector. Instead we have provided a specialised visual editor for editing them. To open the automation plan editor go to the main menu and select *Window* *uTomate Automation Plan Editor*.

At the top of the automation plan editor you can see the automation plan selector. You can select a plan for editing by clicking on the dot icon at the right of the plan selector. Alternatively you can drag an automation plan from the project tree on the plan selector. The rest of the automation plan editor is divided in two sections. On the left side you see the entries that are part of the automation plan in a node view. On the right side you see the entry inspector which allows you to modify the properties of the currently selected entry.



In the node view you can see which entries are currently part of your automation plan. Each node in the node view represents a single entry of the automation plan. The first entry of the automation plan is highlighted with a blue ribbon. When you run an automation plan it will start with the node marked with that blue ribbon. To set an entry as the first entry, right-click that entry and select the *Set As First Node* option in the context menu. To delete an entry, right-click the entry and select the *Delete Node* option in the context menu. This will automatically delete all incoming and outgoing connections of that entry as well. You can add new entries to an automation plan by right-clicking on a free space in the node view. This will open a context menu from which you can select which type of entry you'd like to add. Please look at the Entry types section for more information about the available entry types.



The automation plan entries can be connected with each other using connectors. These connections control which entry is executed after which. Each connector can only be connected to a single entry, however multiple connectors can connect to the same entry. Most entries have only a single connector named *Next Entry* which should be connected to the entry that you want to be executed after the originating entry. To connect two entries simply drag a connection from the connector to another entry (click on the connector circle, keep the mouse button down and then drag into the direction of the other entry. When you are above the other entry, release the mouse button). To delete a connection, right-click the connector circle of the entryfrom where the connection starts and then select the *Delete Connection* option in the context menu. To connect an already connected connector to a different entry, simply drag a new connection from the connector to the new entry. It will replace the old connection.

CAUTION Be careful when connecting entries. uTomate cannot safely determine whether or not your connected entries form a circle (especially when using loop or decision entries with expressions). When the entries form a circle, your automation plan might loop forever (again depending on the used expressions) and you will have to manually stop it.

Automation plan entries

Overview

uTomate comes with 6 types of automation plan entries, which you can combine into automation plans.

Run Action

The *Run Action* entry will execute a single action and is the bread and butter automation plan entry that you will use most of the time. You can add a *Run Action* entry using the context menu or by simply dragging an action from your project tree into the node view.

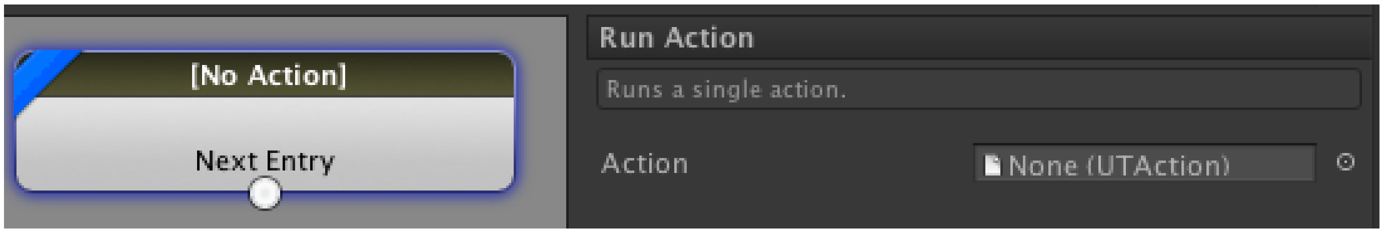


Table 1. Run Action properties

Name	Required	Description
`Action`	optional	The action that should be run. If no action is specified, the entry will do nothing.

Decision

The decision entry allows you to add decisions to your automation plan. For example you might not want to run parts of your plan on certain platforms or you might want to skip expensive parts of your plan without having to modify the plan over and over again. The decision entry has two connectors *Entry If True* and *Entry If False*. In the inspector of the decision entry you can specify the decision to be made. By default this is a simple check box. If you tick the check box the entry that is connected to *Entry If True* will be executed next. If the checkbox is not ticked the entry that is connected to *Entry If False* is connected. You can switch the decision property to expression mode to dynamically evaluate an expression when the entry is executed. If this expression evaluates to `true` the entry connected to *Entry If True* will be executed next, otherwise the entry connected to *Entry If False* will be executed next.

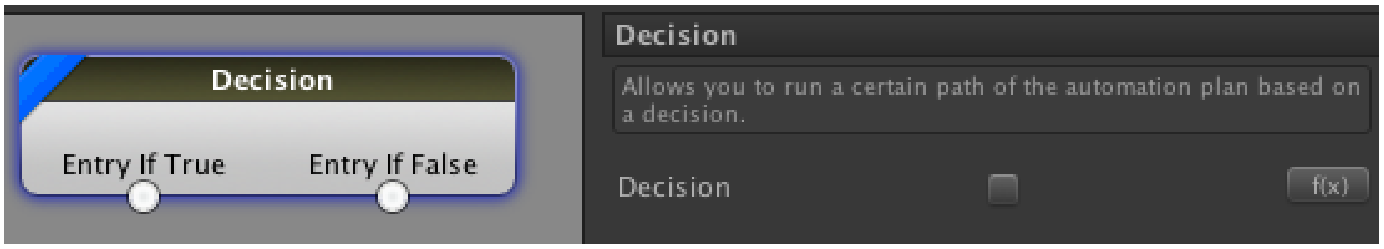
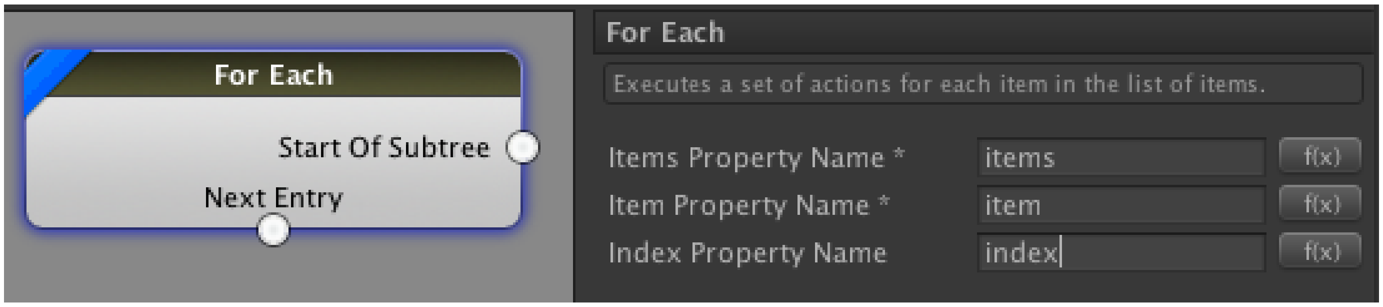

Decision
Entry If True
Entry If False
Decision
Allows you to run a certain path of the automation plan based on a decision.
Decision
f(x)

Table 2. Decision properties

Name	Required	Description
`Decision`	required	The decision to be made. You will almost always want to put this into expression mode.

For Each

The *For Each* entry repeats a part of the automation plan for each entry in a given list or array. When the *For Each* entry is running it will read a list of items from the configured `Items` property. Then it will write the currently iterated item into a configured `Item` property and run the automation plan entry that the *Start Of Subtree* connector is pointing to. Once the last entry in the subtree has been executed, the *For Each* entry will write the next item from the `Items` list into the `Item` property and then again run the entry to which the *Start Of Subtree* connector is pointing. This will be repeated until every item in the list has been iterated over.



Executes a set of actions for each item in the list of items.

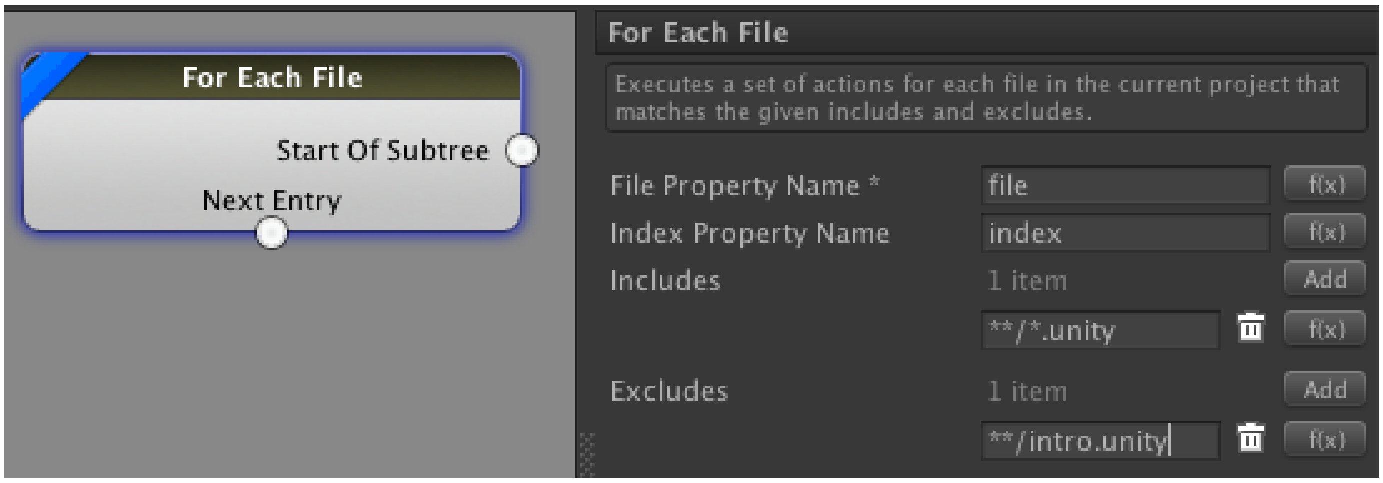
Items Property Name * items
Item Property Name * item
Index Property Name index

Table 3. For Each properties

Name	Required	Description
`Items Property Name`	required	The name of the property which contains the list of items.
`Item Property Name`	required	The name of the property which should be filled with the currently iterated item.
`Index Property Name`	optional	The name of the property which should be filled with the current index in the list of items.

For Each File

This entry works similar to the For Each entry, but instead of a list of items, it works on a file set. It has properties for specifying files to be included and excluded from the file set (just like a lot of uTomates actions have). The currently iterated file will be written into a configured property- When you run this entry, it will first build the file set based on the specification. Then it will put the name of the first file in the file set into the property given in `File Property Name`. Then the entry that the *Start Of Subtree* connector is pointing to will be run. After the sub-tree is executed, the whole process will be repeated for the next file in the file set until all files in the file set have been iterated over.



Executes a set of actions for each file in the current project that matches the given includes and excludes.

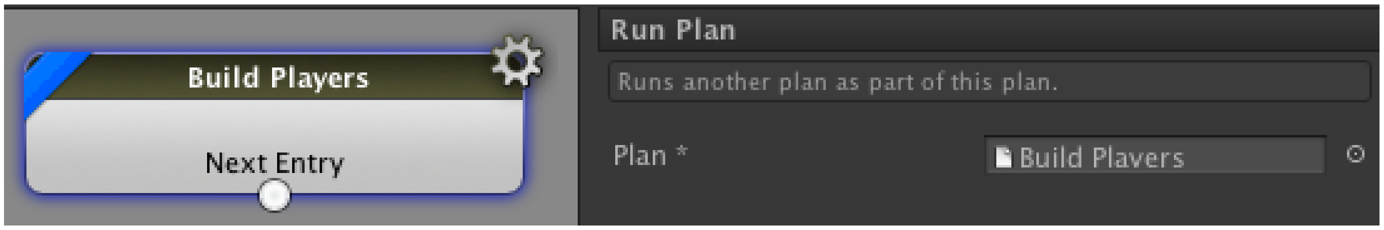
File Property Name * file
Index Property Name index
Includes 1 item
**/*.unity
Excludes 1 item
**/intro.unity

Table 4. For Each File properties

Name	Required	Description
`File Property Name`	required	The name of the property which should be filled with the current file name.
`Index Property Name`	optional	The name of the property which should be filled with the index of the current file in the file list.
`Includes`	optional	The files to be included into the file set.
`Excludes`	optional	The files to be excluded from the file set.

Run Plan

The *Run Plan* entry allows you to run another automation plan as part of this automation plan. This is useful if you want to re-use the logic from one plan in another.



Runs another plan as part of this plan.

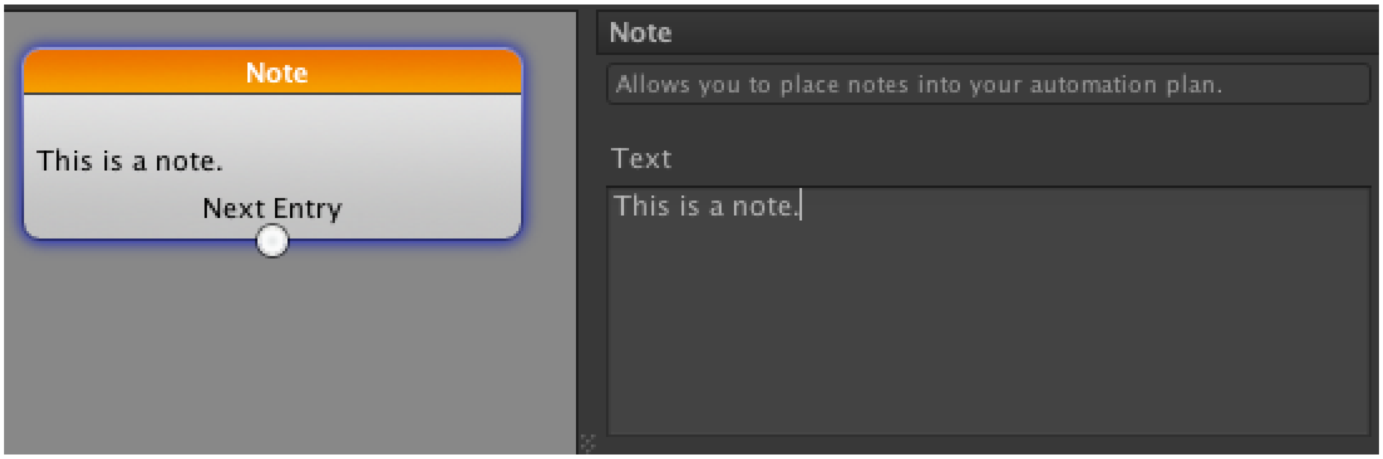
Plan * Build Players

Table 5. Run Plan properties

Name	Required	Description
`Plan`	required	The automation plan to be run.

Note

The *Note* automation plan entry allows you to place notes on your plan. This is useful for documenting parts of your automation that might not be self-explanatory. It can also be used to document prerequisites of the plan (e.g. some editor properties that need to be set before the plan can be run). This entry is special in that it has no connectors, cannot be set as the first entry and therefore not be executed. It is for documentation purposes, only.



Note

Allows you to place notes into your automation plan.

Text

This is a note.

Table 6. Note properties

Name	Required	Description
`Text`	optional	A text note to document the automation plan.

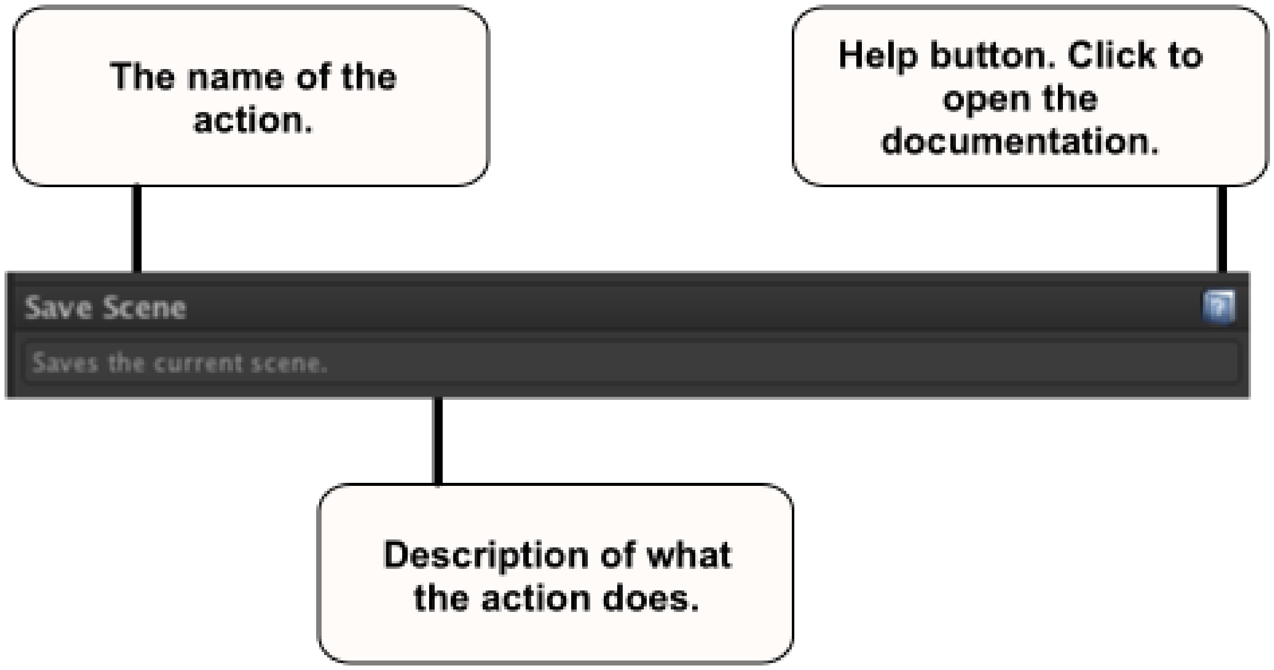
Actions

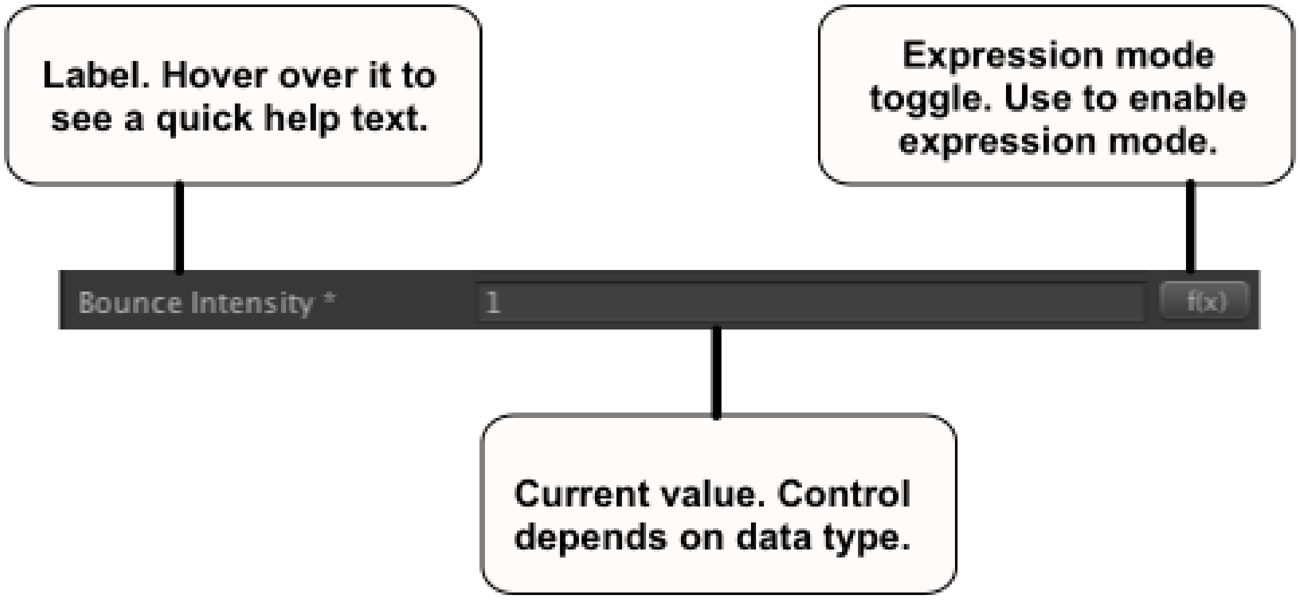
Creating actions

To automate any work using uTomate you use actions. To create an action simply right-click the folder in your project where you want to add the action. Then select *Create* *uTomate* and select the kind of action you want to create. You can find a description of all actions at our uTomate actions reference pages.

Editing actions

When you select an action in the project view, the action’s properties will be displayed in Unity’s inspector panel. The inspector works the same way as the usual Unity3D inspector, so for each property of an action you will see a line in the inspector which enables you to modify this property. In addition to this, there are a few other elements which are described in the following image:


The name of the action.
Help button. Click to open the documentation.
Save Scene
Saves the current scene.
Description of what the action does.


Label. Hover over it to see a quick help text.
Expression mode toggle. Use to enable expression mode.
Bounce Intensity *
1
f(x)
Current value. Control depends on data type.

Expression mode

Each property of an action can be switched into expression mode by clicking the expression mode toggle. This will hide the data type specific input (e.g. the checkbox, select box, etc.) and replace it with a text field into which you can enter an expression. When your action is run the value of this property will then be determined by evaluating the expression. The expression can be an arbitrary UnityScript expression. See the section on expressions for more details about how expressions work.

File sets and wildcards

Wildcards are intended for collecting sets of files easily. You use them with actions that do something for a set for files.

Many actions use file sets as they work on multiple files, for example the *Build Player* action or the *Copy Files* action.

Every action which supports wildcards usually has an *Includes* and an *Excludes* property. You combine these two to select the files you want to have affected by the action. E.g. if you want to select all png files in your project you would set *Includes* to `**/*.png`. The `**` stands for "anywhere below", the `*` stands for "exactly below" or a part of a file/folder name. So for example if you only wanted to select pngs in your `/Textures` folder and all subfolders of it then the wildcard would be `Textures/**/*.png`. If you only want to select pngs directly in your textures folder but not in subfolders, the wildcard would be `Textures/*.png`.

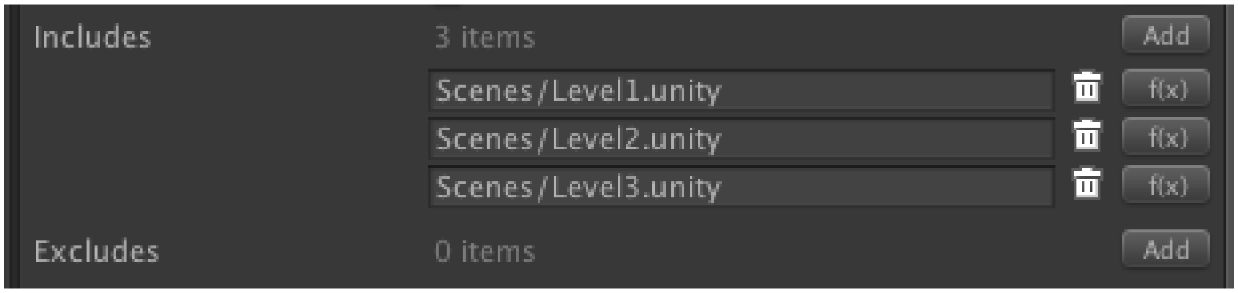
Wildcards can also match files by file names, so if you want to select all PNGs starting with `robot` (e.g. `robot01.png`, `robot02.png`, etc.) the wildcard would be `**/robot*.png`. You can also combine this so e.g. if you want to select all png files anywhere below the textures folder starting with `robot` then the wildcard would be `Textures/**/robot*.png`.

The *Excludes* property allows you to reduce the selection once you made it. You usually use this if your selection criteria sounds like "anything like this, except that". So if you want to select all PNGs starting with `robot` except the one’s ending with `05`, then you can do this by setting *Includes* to `**/robot*.png` and Excludes to `**/*05.png`.

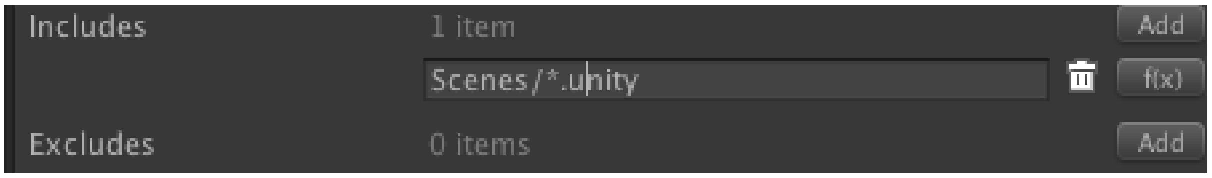
Using wildcards is preferable over selecting files directly, as they allow you to select a lot of files quickly with just a few keystrokes instead of having to do a lot of typing and clicking. Let’s assume you want to build a player and your scenes are stored in a `Scenes` folder inside your project:

- `Scenes`
  - `Level1.unity`
  - `Level2.unity`
  - `Level3.unity`
  - etc.

Then you could enter these scenes one by one into the action:


Includes 3 items
Scenes/Level1.unity
Scenes/Level2.unity
Scenes/Level3.unity
Excludes 0 items

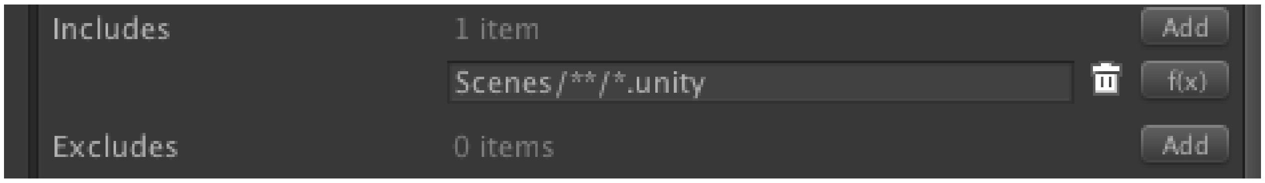
That of course is rather cumbersome and doesn't scale up very well when you have 150 scenes in your project. As shown in the example above, you use a wildcard character `*` to select more than one file with a single line:


Includes 1 item
Scenes/*.unity
Excludes 0 items

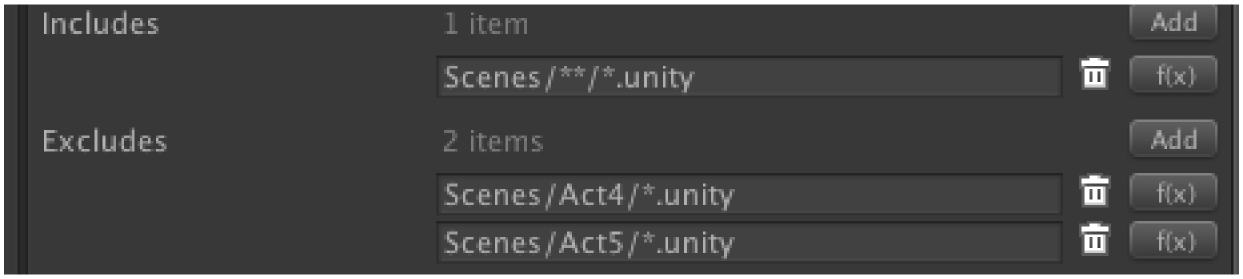
This will select all Unity scenes within the `Scenes` folder. You can also have a fully recursive search using the `**` wildcard character. For example if you have your scenes layouted like this:

- `Scenes`
  - `Act1`
    - `Level1.unity`
    - `Level2.unity`
  - `Act2`
    - `Level1.unity`
    - `Level2.unity`

Then you can still select all of your scenes with a single line:


Includes 1 item
Scenes/**/*.unity
Excludes 0 items

By combining *Includes* and *Excludes* you have complex file sets with just a few lines of setup. For example if you want to include all your scenes within your `Scenes` folder but not the one's in the `Act4` and `Act5` subfolders, you can combine *Includes* and *Excludes* to have this:



Includes: `Scenes/**/*.unity`

Excludes: `Scenes/Act4/*.unity`, `Scenes/Act5/*.unity`

Wildcard selections have the nice behaviour of automatically adapting to your project. So if you for example decide to add some more scenes to the folder, you don't need to change the action setup to match new scenes. Also if you choose to rename or delete scenes, your action doesn't need to be updated. Since it is using the wildcard selection, it will automatically match all scenes within your folder structure. So you should almost always use wildcard selections over hardcoding full paths into your actions.

CAUTION Using the `**` wildcard for the *Includes* or *Excludes* field might incur performance problems if your project is very large. If you can, avoid expressions having `**` at the start. For example if all your textures are below the `Textures` folder, you could of course select them with `**/*.png` as this would include the `Textures` folder as well. However the whole folder tree of your project would be scanned. If you use `Textures/**/*.png` instead, then scanning will be limited to the `Textures` folder, which will be much faster. In general be as specific as you can when using wildcards to avoid unneccessary directory scans.

Expressions

Overview

Expressions are a powerful feature of uTomate that can be used almost everywhere where a value is required. Expressions allow you to have a certain value calculated when a plan is running instead of specifying it beforehand. To use an expression for any action property, simply switch the property to expression mode by toggling the expression mode button.

Expression syntax

Expressions are specified in an extended UnityScript syntax. Basically everything that can be done in UnityScript can be used as an expression in uTomate. Within expressions you have full access to the .Net API and Unity's own API. It is also possible to define and use utility functions that you can call from an expression using script extensions. uTomate comes with several built-in utility functions. See the link:http://www.ancientlightstudios.com/utomate/documentation/scriptextensiondocs/index .html[script extension reference page] for more details. You can access runtime-properties by typing a `$` sign followed by the name of the property. Property names are case-sensitive. A few examples:

```
$foo
$Foo // this is different from $foo
$foo:bar
$_foo23

string.IsNullOrEmpty($foo) // uses .net API
$ut:file.Exists($project:assets + '/My.asset') // uses utility functions

// concatenate two strings
"foo" + "bar" // yields "foobar"

// calculate a number
2 * 2 // yields 4

// multiply a string
"2" * 4 // yields "2222"

$foo + "bar" // yields "barbar" if $foo is set to "bar"
$foo * 2 // yields 4 if $foo is set to 2, yields "22" if $foo is set to "2"
```

NOTE Due to technical limitations you currently cannot use `$<property>` inside some string. For example if you want to have a string that reads like `"$foo"` (verbatim) you need to concatenate it using `"$" + "foo"`. Otherwise the preprocessor will replace the `$foo` reference and the output will not look as you would expect it to be. We hope to overcome this limitation in a later version of uTomate.

Whenever you try to access a runtime property that is not known, the plan execution will be aborted. If you don't want this to happen, add a question mark to the properties name. If the runtime-property is not known this expression will return the `null` value instead of aborting the plan execution.

```
$iMightNotExist?
```

CAUTION Usually you want your plan execution to halt on unknown properties instead of silently ignoring them. Silently ignoring things yields erratic and hard to debug behavior in many cases, so use this feature sparingly.

Expression typing

All expressions are strongly typed. You therefore need to do type conversions and casting if required. For example if an action property reqires an `int` value, but your runtime-property is of type `float` you need to invoke .net's `Math .round` function:

```
Math.round($myFloatProperty)
```

NOTE All project and editor properties that can be set in the main window are of `string` type.

If the expression type does not match the action property type, the plan execution will be aborted. To make things easier, there are some implicit built-in casts and conversions available. The following table lists these:

Table 7. Built-in type conversions

Target type	Expression type	Conversion Rule	Example
`string`	`any`	If the expression's value is null the string value will be "null" otherwise the `ToString()` function will be called to convert the expression to a string.	* `null` `"null"` * `10` `"10"` * `SomeEnum` `"SomeEnum"`
`bool`	`string`	If the string value is "true" (in any letter case) the bool value will be true, otherwise false.	* `"true"` `true` * `"True"` `true` * `"false"` `false` * `"foo"` `false` * `null` `false`
`any enum type`	`string`	The string value will be de-nicified and matched against the known enum members. If any member fits, the expression will be converted to that enum member.	Let the target type be `UnityEngine.LightmapsMode`: * `"Single"` `LightMapsMode.Single`
`int`	`string`	The string value will be parsed into an int.	* `"10"` `10` * `"001"` `1`
`float`	`string`	The string value will be parsed into a float.	* `"10.0"` `10.0f` * `"100"` `100.0f`

NOTE When you enable debug mode all implicit casts and conversions are being logged to the console. This is useful to check and debug the results of the conversion.

Runtime properties

Runtime properties allow to transfer data between actions. They can be set using the *Set Property action* or using the project properties or editor properties dialogs.

uTomate comes with a set of useful built-in runtime properties. You can extend this set using script extensions.

Table 8. Built-in runtime properties

Property	Value
`$project:root`	The root folder of the currently open project.
`$project:assets`	The Assets folder of the currently open project.

`$os:platform`	The platform that this editor is running on. One of:
	
	`"Win32NT"`
	`"Unix"`
	`"MacOSX"`

`$os:pathSeparatorType`	The kind of path separator used on the current platform:
	
	`"Windows"`
	`"Unix"`

`$os:pathSeparatorChar`	The actual path separator character that is used on the current platform:
	
	`"/"`
	`"\"`

`$user:home`	The full path to the current user’s home directory.
`$user:desktop`	The full path to the current user’s desktop.
`$utomate:debugMode`	`true` if the debug mode is enabled in uTomate’s preferences, `false` otherwise. Available since version 1.3.0.
`$unity:isUnityPro`	`true` if this is running on Unity Pro, `false` otherwise.
`$unity:supportsAndroid`	`true` if building for Android is supported by Unity, `false` otherwise. Available since version 1.2.0.
`$unity:supportsIos`	`true` if building for iOS targets is supported by Unity, `false` otherwise. Available since version 1.2.0
`$unity:version`	Contains the version of Unity. Available since version 1.2.0
`$unity:selectionAtPlanStart`	Contains the object that was selected when the automation plan started. This can be used to create plans that work on the currently selected object. Available since version 1.4.3.

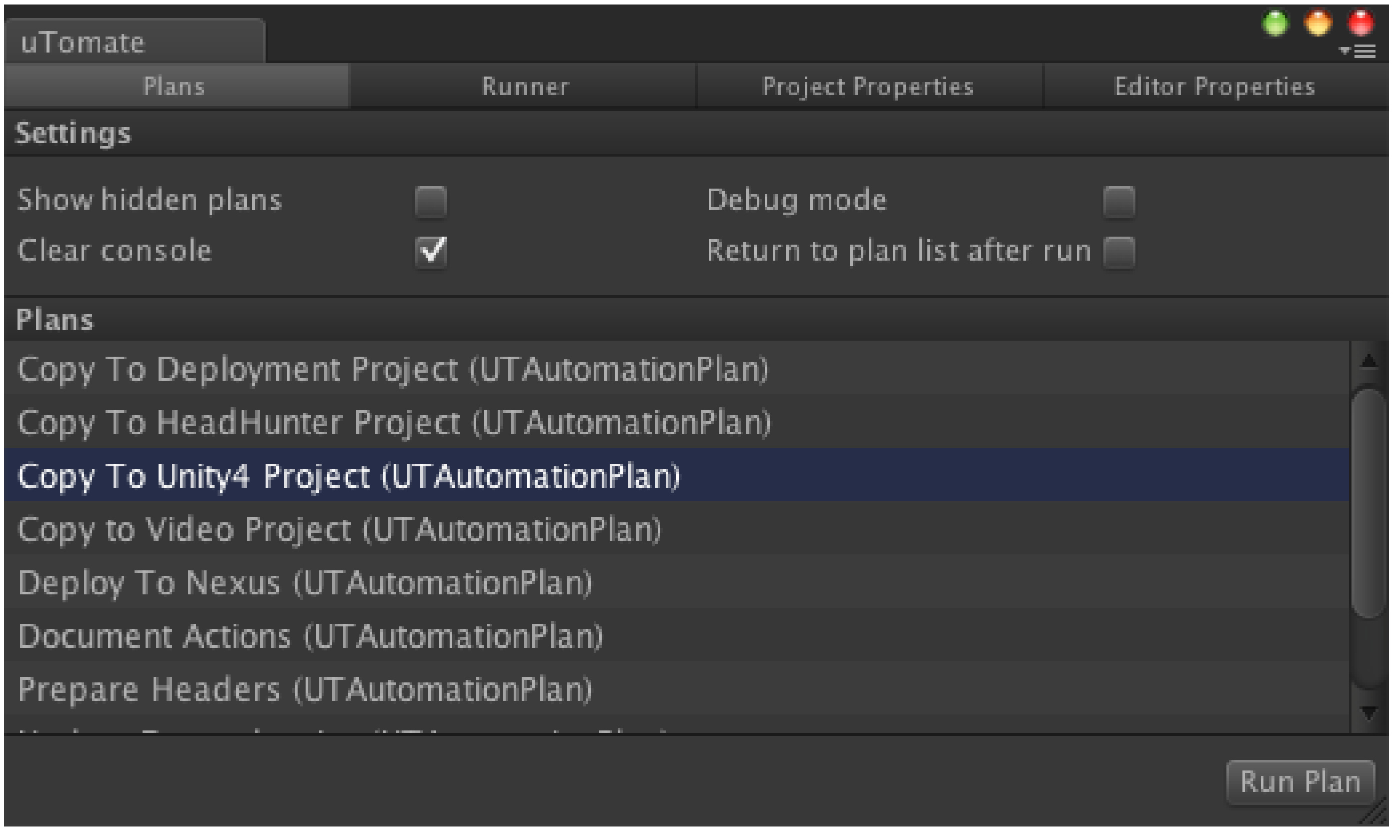
The uTomate main window

Overview

uTomate’s main window provides several views that allow you to quickly run a plan from your project and preset custom runtime-properties. You can open the main window using the main menu *Window* *uTomate* *Main Window*.

Settings and plan view

The settings and plan view shows a list of all plans that are currently available in your project.



Settings

Show hidden plans [ ]
Clear console [✓]
Debug mode [ ]
Return to plan list after run [ ]

Plans

Copy To Deployment Project (UTAutomationPlan)
Copy To HeadHunter Project (UTAutomationPlan)
Copy To Unity4 Project (UTAutomationPlan)
Copy to Video Project (UTAutomationPlan)
Deploy To Nexus (UTAutomationPlan)
Document Actions (UTAutomationPlan)
Prepare Headers (UTAutomationPlan)

You can run a plan from here by selecting it in the list and then clicking on the *Run Plan* button. This will automatically switch the main window to the runner view. Above the plan list there are a few settings, which are described in the following table:

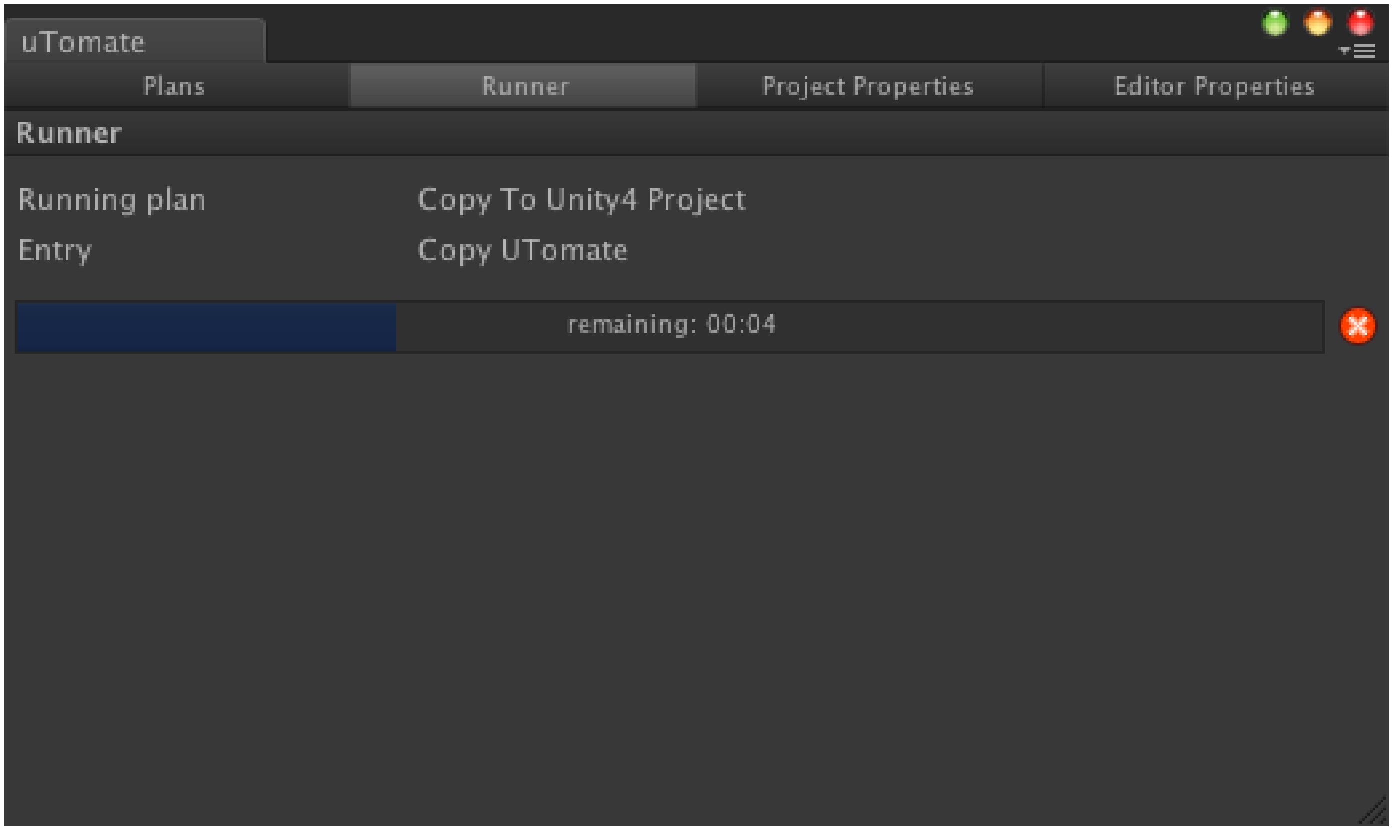
Table 9. uTomate Settings properties

Setting	Description
Show hidden plans	When this checkbox is ticked, the plan list also shows plans that were hidden by setting the "Hide in Execution Window" in the plan editor.
Clear Console	When this checkbox is ticked, the console will be automatically cleared whenever you run a plan.
Debug Mode	When this checkbox is ticked, all actions provider more verbose output that is useful when debugging your automation plans.

Setting	Description
Return to plan list after run	When this checkbox is ticked, the main window will return to the plan list after running a plan. Otherwise it will stay at the runner view.

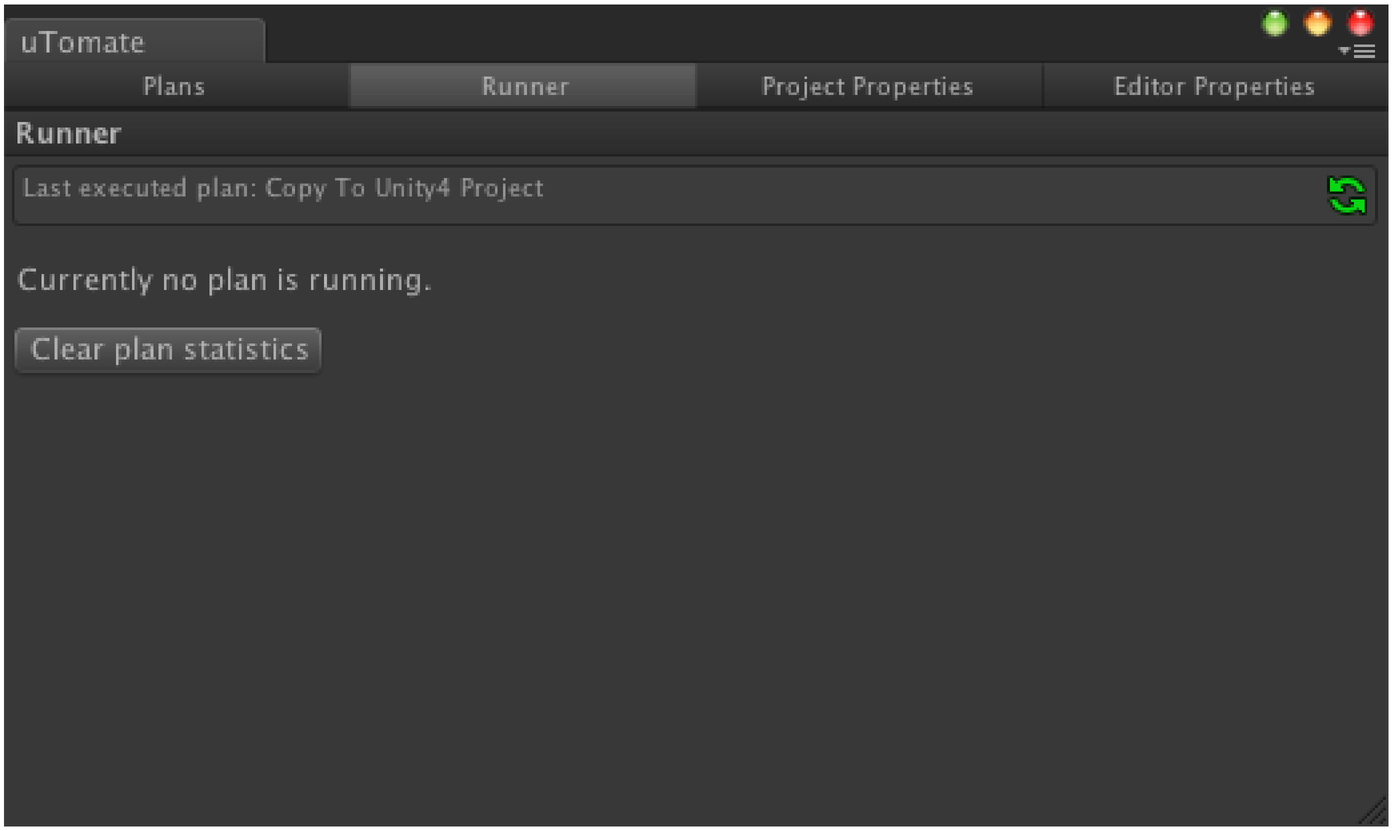
Runner view

The runner view has two states. The running state is active whenever you execute a plan.



Running plan Copy To Unity4 Project
Entry Copy UTomate
remaining: 00:04

In this state the view will show the currently executed plan and action and an estimation of how long this plan will take to execute. That estimation is based on statistics that have been collected of previous runs of the plan, so it will be more accurate with each run of the plan. Using the cancel icon next to the progress bar you can cancel the run. Cancelling will usually finish the currently running action, though some long running actions (such as "Bake Lightmaps") have special support for cancellation and will cancel as fast as possible without finishing their task.



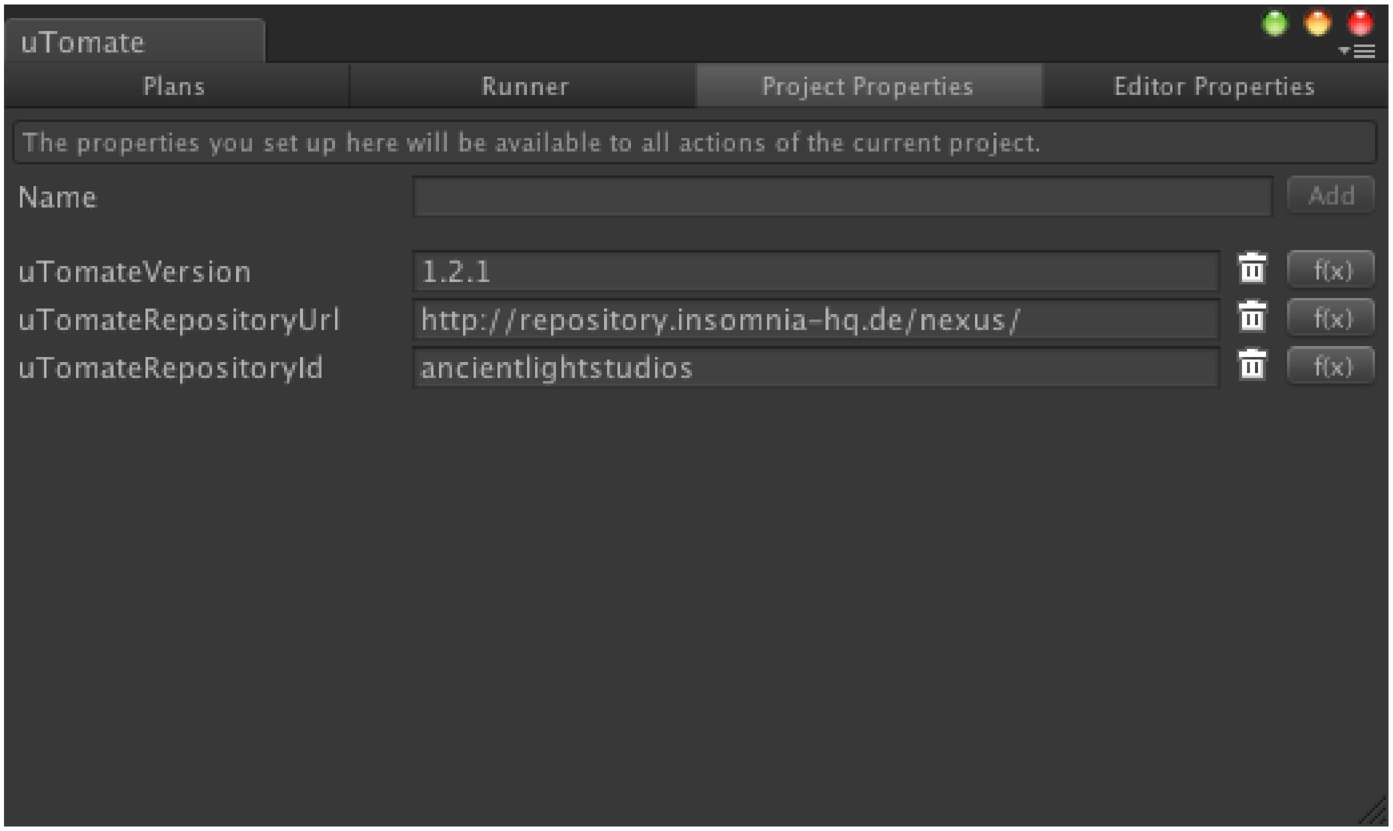
Last executed plan: Copy To Unity4 Project

Currently no plan is running.

In idle state the runner view shows the plan that was executed last. By clicking on the repeat icon next to the plan's name you can run this plan again quickly without having to return to the plan view. There is also a button labeled *Clear Plan Statistics*. With this button you can reset the plan statistics, which are used to project how long a plan will take to execute. This button is useful if you have made significant changes to one or more plans and want these statistics to be re-calculated.

Project properties view

The project properties view allows you to pre-set runtime properties. These properties will be set before any plan in the current project is being run. This is useful if you have common settings that are reused by many actions and that you may want to change later without having to visit each action using this setting. They will be persisted into a special asset named `uTomateProjectProperties` which you can (and should) add to version control. Using this file you can share the properties with the whole team working on that project.



The properties you set up here will be available to all actions of the current project.

Name [Add]

uTomateVersion 1.2.1

uTomateRepositoryUrl http://repository.insomnia-hq.de/nexus/

uTomateRepositoryId ancientlightstudios

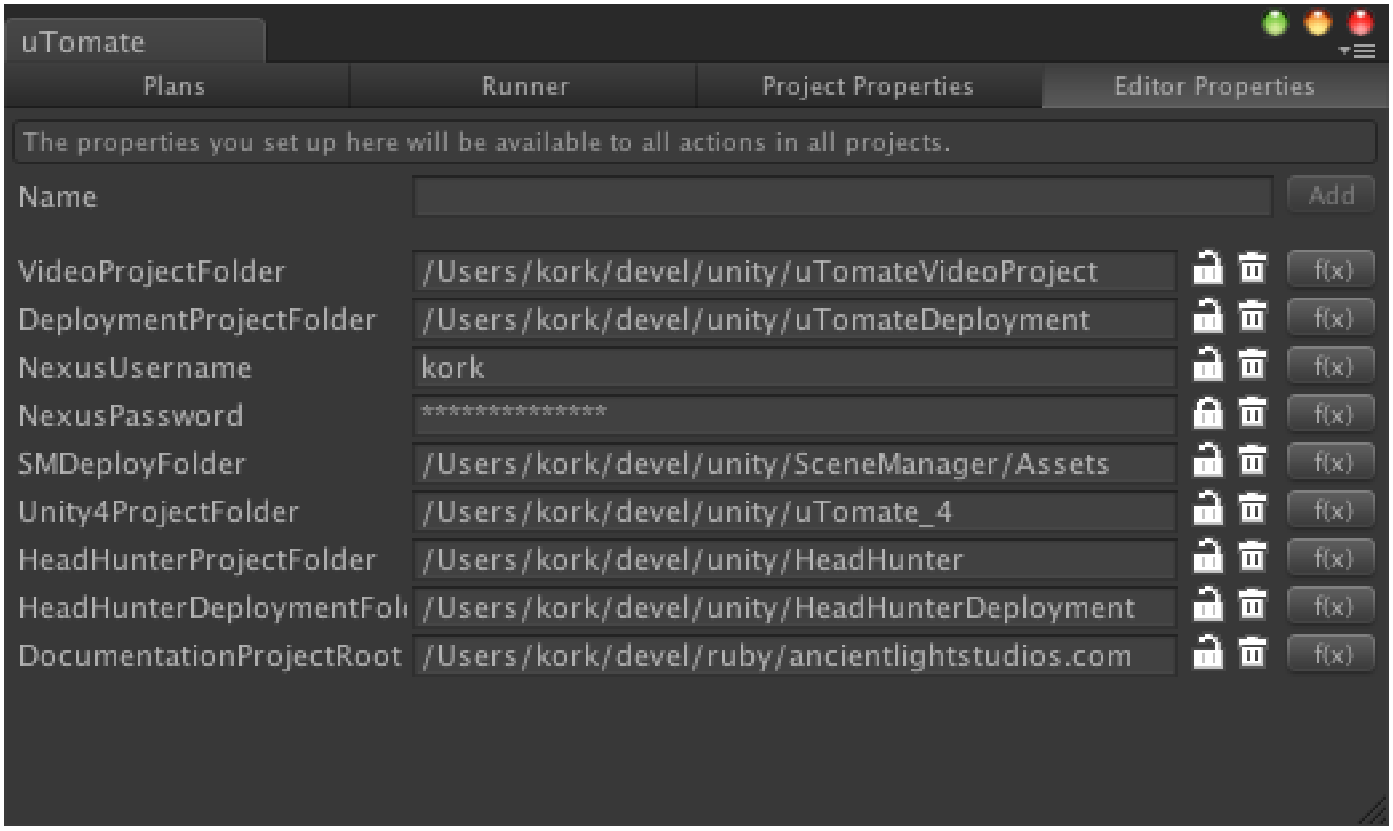
To add a property, type it's name into the text field labeled *Name* and then click the *Add* button. The new property will be added to the list of properties. You can then specify its value by simply typing it into the property's text field. You may also specify an expression for this property by switching the expression mode on. To delete a property, click the trashcan icon next to the property.

NOTE Property names must start with a letter or underscore followed by zero or more letters, numbers colons or underscores.

CAUTION Properties are defined in the order in which they appear in the list, so when using expressions you can only reference properties that are defined before the point at which you reference them.

Editor properties view

The editor properties view allows you to specify runtime-properties that are local to your editor. All properties specified in this view work just like the project properties, but they will not be shared to the team. Use this view to specify properties that are different on each machine (e.g. paths to certain projects) or contain confidential information like passwords. Also the editor properties are not specified per-project but are rather shared between projects (you could think of them as a kind of environment variables).



uTomate

Plans	Runner	Project Properties	Editor Properties

The properties you set up here will be available to all actions in all projects.

Name	
VideoProjectFolder	/Users/kork/devel/unity/uTomateVideoProject
DeploymentProjectFolder	/Users/kork/devel/unity/uTomateDeployment
NexusUsername	kork
NexusPassword	***************
SMDeployFolder	/Users/kork/devel/unity/SceneManager/Assets
Unity4ProjectFolder	/Users/kork/devel/unity/uTomate_4
HeadHunterProjectFolder	/Users/kork/devel/unity/HeadHunter
HeadHunterDeploymentFol	/Users/kork/devel/unity/HeadHunterDeployment
DocumentationProjectRoot	/Users/kork/devel/ruby/ancientlightstudios.com

In addition to the controls for adding, removing and toggling expression mode there is a lock icon next to each editor property. When you click the lock, the input field is masked with asterisk characters. This is useful to hide passwords from the eyes of bystanders when showing things on your machine.

CAUTION All data is stored unencrypted in the editor preferences! Ticking the lock icon will not encrypt the property value it will simply mask it in the view.

Advanced Topics

Running uTomate from the command line

uTomate comes with a support script that allows you to run uTomate plans from the command line. That way uTomate plans can be used in continuous integration environments. To run a plan from the command line, invoke Unity3D's command line. Under MacOS, you can launch Unity from the Terminal by typing:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity
```

..while under Windows, you should type

```
"C:\Program Files (x86)\Unity\Editor\Unity.exe"
```

..at the command prompt. The following table shows the known command line arguments and their meaning:

Table 10. uTomate command line arguments

Argument	Required	Description
`-projectPath <Path>`	Optional	Full path to the project that contains the uTomate automation plan.
`-batchmode`	Optional	Suppresses all popup windows that would require manual intervention.
`-executeMethod AncientLightStudios.uTomate.UTExternalRunner.RunPlan`	Required	Instructs Unity3D to execute uTomate's command line runner.
`-plan <Plan Name>`	Required	The plan that should be run.
`-prop <Name>=<Value>`	Optional	Sets a runtime-property before executing the plan. This option can be repeated for each property you would like to set.
`-debugMode <true/false>`	Optional	If set to true, this enables Debug Mode which will print more verbose output to Unity's console.
`-nographics`	Optional	(Windows only)When running in batch mode, do not initialize graphics device at all. This makes it possible to run your automated workflows on machines that don't even have a GPU. If you are using a continuous integration system adding this option might be necessary depending on your agent setup.
`-logFile <filename>`	Optional	Setting this option allows you to redirect the log output to file. This is useful for continuous integration builds where you would probably like to inspect the log in case of an error.

NOTE There are more command line arguments available on Unity's command line facility. Have a look at Unity's documentation for the command line.

NOTE When you are using uTomate with Unity 4 or below, you need to remove the namespace from the `-executeMethod` argument. In this case use
`-executeMethod UTExternalRunner.RunPlan`
instead of
`-executeMethod AncientLightStudios.uTomate.UTExternalRunner.RunPlan`

CAUTION Do not specify the `-quit` argument. uTomate is driven by the UI thread of Unity3D so it will not block the UI while it is running. However the `-quit` argument assumes that command line runners block until they are done. Therefore if you specify the `-quit` argument Unity3D will quit right after uTomate has initialized but before the plan is actually executed. uTomate will quit Unity3D automatically once the plan has finished (or failed) so specifying the `-quit` argument is not necessary.

One Example:

```
# This runs the plan 'My Plan' in the Unity project
# at C:\projects\myproject, sets the properties
# - 'Foo' to 'Bar'
# - 'some:prop' to 'some value'
# and enables debug mode

"C:\Program Files (x86)\Unity\Editor\Unity.exe"
    -executeMethod AncientLightStudios.uTomate.UTExternalRunner.RunPlan
    -projectPath "C:\projects\myproject"
    -plan "My Plan"
    -prop "Foo=Bar"
    -prop "some:prop=some value"
    -debugMode true

# On OSX, this would run the plan 'My Plan' in the
# Unity project at /Users/MyUser/myproject

/Applications/Unity/Unity.app/Contents/MacOS/Unity
    -executeMethod AncientLightStudios.uTomate.UTExternalRunner.RunPlan
    -projectPath "/Users/MyUser/myproject"
    -plan "My Plan"
    -prop "Foo=Bar"
    -prop "some:prop=some value"
    -debugMode true
```

When the plan has been finished successfully, the editor will exit with exit code 0. If the plan has been canceled or failed, the editor will exit with exit code 1. Using the exit code you can decide whether or not your continuous integration has failed.

Script extensions

Since uTomate 1.2.0 it is possible to extend the default set of runtime properties and to add utility functions to expressions by using script extensions.A script extension is basically a class which gets instanciated whenever an automation process is started. The class instance will then be made available to all actions under a chosen property name.

```
// the UTScriptExtension annotation marks this
// class as script extension
// "example" is the name under which it will be
// made available in the scripting context.
[UTScriptExtension("example")]
public class ExampleScriptExtension {
  public string KnockKnock() {
          return "Who's there?";
     }
}
```

This example script extension will be registered under the property name example in the scripting context. You can access it from every action by calling:

```
$example.KnockKnock() // will return "Who's there?"
```

CAUTION Script extensions are intended to complement actions, not to replace them. As a rule of thumb, every script extension you write should only contain functions that do reasonably simple things, like converting or calculating values.

Using script extensions you can add utility functions in a pluggable way and use them from every action. It is also possible that a script extension prepares several properties in the scripting context.

In fact all built-in runtime properties are prepared by a script extension:

```
// you don't need to specify a variable name. If it's not there
// the script extension itself will not be injected in the
// scripting context.
// the 0 is the load order priority. Since this extension provides
// the built in properties it is loaded first.

[UTScriptExtension(0)]
// When a ScriptExtension implements UTIContextAware
// it get's the scripting context injected once
// it is initialized
public class UTBuiltInVariablesScriptExtension : UTIContextAware
{
    public UTContext Context {
        set {
            SetupVariables (value);
        }
    }

    private void SetupVariables (UTContext context)
    {
        context ["project:root"] = UTFileUtils.ProjectRoot;
        context ["project:assets"] = UTFileUtils.ProjectAssets;
        // and so on...
    }
}
```

Once the script extension has been initialized, the properties are available in the scripting context and can be used by actions.

NOTE All you need to do to register a script extension is annotating it with the `UTScriptExtension` attribute. It will automatically be picked up by uTomate.

Support of older Unity versions

In general, uTomate tries to hide differences between Unity versions from you. Depending on which Unity version you are using, some actions may show different configuration entries. Some actions, will not be available in older Unity versions, either because the functionality these actions provide was not available in the older Unity version (e.g. the new asset bundle system introduced in Unity 5 is not available in Unity 4) or because we didn’t backport this action to the older Unity version. In general we provide new actions only for the latest version of Unity and only backport them to older Unity versions if there is significant demand for a backport and the amount of work required for the backport is within reasonable limits. You can use the Unity version switch in the available uTomate actions documentation to see which action is available in which Unity version.