# UMG to Lua translator user guide

## Table of Contents



### *Overview:*

The UMG to Lua translator tool lets a user take a UMG widget in UE4 and translate it to a Lua file which outputs the same UI as the UMG widget. This tool can be accessed from within the UMG editor in the Unreal editor. The final translation can be given as either a new Menu file, a new Lua Class, which can then be reused in other translations, or it can be translated as raw widgets.

### *Limitations:*

Since not every Widget class available in UMG has an equivalent class in Lua, not every class is open for translation. Only classes that are named “Translatable <WidgetType>”, such as “Translatable Image”, are translatable.

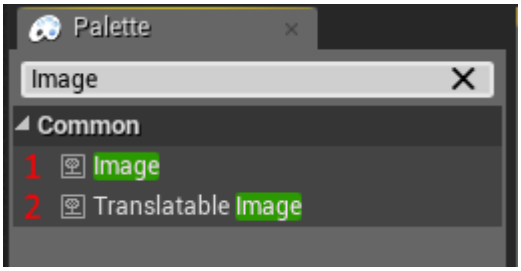


*Illustration 1: Translatable and non-translatable widgets.*

As seen in illustration 1, there are 2 types of image Widgets:

1. The standard Image, that comes in UE4 by default. This Widget cannot be translated.
2. “Translatable Image”, which is an extended version of Image which can be translated.

In most cases the Translatable types are functionally the same as their regular counterpart. The exception is the “Translatable Button”, which is modified to be more similar to its Lua counterpart.

### *Interface:*

## Opening the editor

When opening the UMG editor, a new button is present in the toolbar, by default at the top of the editor, as seen below in Illustration 2.


*Illustration 2: UMG Editor toolbar displaying the “Lua Translator” button, highlighted red.*

When the button is pressed, the Lua Translator tool window will open. In this window you can set several variables for the translation, as seen below in Illustration 3.


*Illustration 3: Lua Translator tool window, with labeled variables and commands.*

## The editor window

Illustration 3 shows the variables and commands present for the Translator. They are:

1. Select save directory command. When pressed, this button will open the windows file explorer, and allow you to select the folder where the eventual translation file will be placed. Use of this is optional, as it is also possible to manually set the file path for the save directory (as seen in 4).
2. Translate Widget command. When pressed, this button will begin the translation process

with the current translation Target Widget and translation variables.

3. Target Widget. This is the name of the Widget that was open in the UMG editor where the tool button was pressed. This shows what widget will be translated when the translate widget command is executed. This is set when the tool window is opened and cannot be changed from the tool window. If you wish to translate a different widget, please open the tool from a UMG editor with that widget open for editing.
4. Save Directory. The file path of the folder where the translation file will be saved. If this file path does not exist, it will be created. If it does exist but is inaccessible, an error will be returned when translation is attempted. This can be set manually by entering a file path, or can be set using the “Select save directory” command, as seen in 1. The default path set here is the current project's root folder. If this variable is left empty, it will default to the project's root folder.
5. File Name. This is the name the Lua file will be given after translation. This defaults to the Widget name of the widget open for editing, or the Lua Class name (see “Lua Class Translations” section) for this widget if one is set. If a file with this name is already present in the target save directory, the translator will return an error when translation is attempted, unless “Allow Overwriting” is ticked (see 7). User caution with that option is advised, as overwriting will completely overwrite the previous file content, unless “Insert at Line nr” is used (see 8).
6. Translation Type. What the widget should be translated as. Possible options are Menu, Class, and Widgets.
   “Menu” will translate the file as a new Lua menu, adding all necessary formatting for the file to be usable as a new menu directly.
   “Class” will translate the file as a new Lua class, adding all necessary formatting for the registering of that new class. This makes the widget reusable for translation in other user widgets. Be careful not to introduce circular dependencies between userwidgets when using this.
   “Widgets” will translate the widgets without any formatting around them. This is useful when you wish to add to an existing Lua menu without having to recreate the entire menu in UMG first.
7. Allow overwriting (CAUTION: advanced option). When checked, this will allow the translator to overwrite the target file if a file with the desired name is already present in the target save directory. When used with “Insert at line nr” (see 8), this can be used to insert new widget data into existing Lua files.
   This is a dangerous option as it allows the translator to completely overwrite the file content of an existing file. Using a version control system or creating backups of your file is recommended.
8. Insert at line nr (advanced option). When “Allow Overwriting” is checked (see 7), this variable can be edited. If “Allow Overwriting” is checked and this variable is set to a positive, non-zero number then the translation will be inserted into the pre-existing file at the given line nr (if one is present, otherwise it will create a new file as normal). Although the translator will allow this to be used with any Translation Type (see 6), it is mostly useful when translating as “Widgets” to add to existing menus or classes.

## Lua translation interface

There are a few more interface elements that were either added for, or are required for translating Classes. These will be discussed further in the “Lua Class Translations” section, but briefly they are:

1. A “Lua class name” entry, when selecting a UserWidget or the root component of the

current UserWidget, as seen below in illustration 4.


*Illustration 4: Lua Class Name translation variable*

2. Variables created in the Graph editor for a UMG Widget. These variables can be translated on instances of that userwidget in other userwidgets. This will be discussed further in the "Lua Class Translations" section.

## *Lua class translations:*

### Lua class name

For the available "Class" translation type, some additional setup is required, and a few additional options are available. First of all, to translate a widget as a Lua class, it needs to be given a Lua Class name. This will be the name of the translation file for this class, for the class definition within that file, and will be the name used to create instances of this class in other translated UserWidgets. The interface for it is discussed in the "Lua translation interface" paragraph of the "Interface" section. Once set this variable should not be changed, since that will invalidate translations in other UserWidgets that use it. If you do wish to change it, you will need to:

1. Change the variable on the root of the UserWidget that is translated as a Lua class, called the Class Widget for now.
2. Translate the Class Widget, overwriting any previously existing Lua file for that class.
3. Go to every UserWidget that uses the Class Widget within it and translate that UserWidget, overwriting any previously existing Lua file for that UserWidget.

If no other UserWidgets use this Class Widget yet, you can skip the third step. Note that if files have been edited after translation, those changes will be lost when overwritten.

### Translating editable Class Widget variables

Additionally, there is an option available for the translation of variables within UserWidgets that you wish to reuse. This option is very exact in its use, and is prone to user error due to the difficulty faced when implementing it.
UserWidgets may have variables you wish to edit on instances of that UserWidget within other UserWidgets. For ease, we will call UserWidgets used within other UserWidgets "Class Widgets". For example, say you have a Class Widget that implements a bar at the top of the screen that shows some player information, as well as the name of the current menu you are in. When you reuse that Class Widget, you may wish to edit a variable on that instance to display the name of your new menu, rather than the default text. This is possible, and is done in two parts (we will use the Menu Name as our example for this explanation).

The first part is creating the variable to be translated. The second part is to get this variable to update your Class Widget in real-time whenever it is changed. For translation, only the first part is mandatory, but the second part makes it much more user friendly and is highly recommended.

### Setting up the variable

To create a translatable variable, you will need to go to the Graph editor for your Class Widget.

There you will add a variable of the desired type, a “Text” in our example. The detail view for example is seen below in illustration 5.

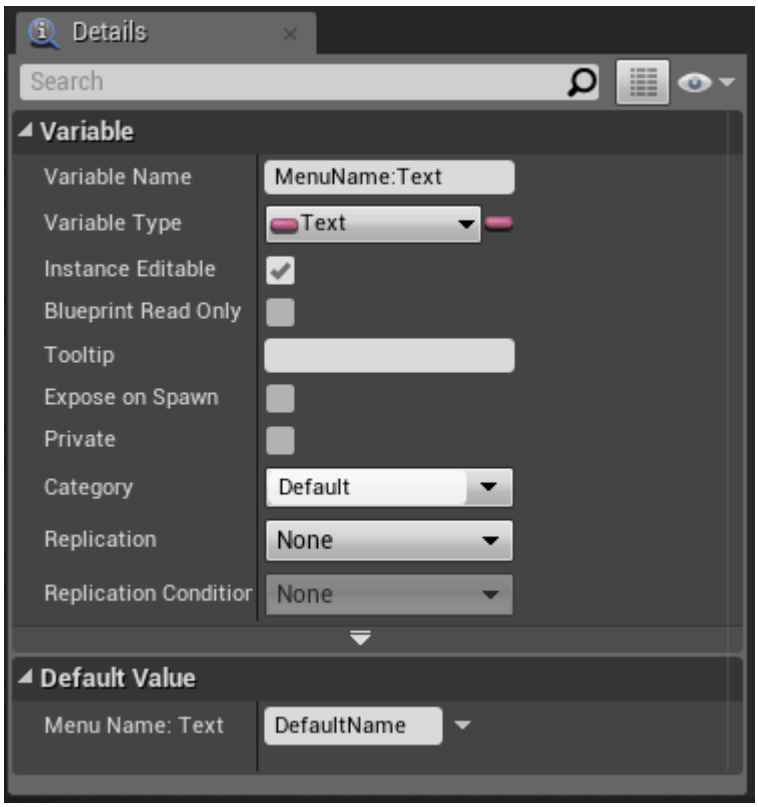

*Illustration 5: The MenuName variable detail view*

There are two things you need to do when setting up this variable, and one recommended step:

1. The variable must be Instance Editable. This will allow you to change this variable on instances of this Class Widget within other User Widgets.
2. The variable name must be very precisely set to one of two things. It must be named either SourceWidget,Variablename or SourceWidget:VariableName, the difference being the use of a comma or a colon in the name.
   When using the comma (e.g. MenuName,Text) this variable will be translated as a direct variable assignment, as ClassWidget.SourceWidget.VariableName = Value, e.g. MyMenuBar.MenuName.Text = DefaultName.
   When using the colon (e.g. MenuName:Text) this variable will be translated using a Set function, as ClassWidget.SourceWidget:SetVariableName(value), e.g. MyMenuBar.MenuName:SetText(DefaultName).
3. Recommended step: Set a default value. This is the value that will be used if nothing is set on instances of this Class Widget, and can be a useful reminder to assign a proper value.

Any variables not adhering to this naming scheme will not be translated.
Note that the Comma (“,”) is translated as a full stop (“.”), unfortunately the UMG Graph editor does not allow for full stops in its variable names, otherwise that would've been more sensible.
Once this setup is completed, the variable can be edited on instances of this class widget, and will be translated when translating UserWidgets that use it.
However, changes to this variable are currently not reflected in the instances of this Class Widget, it is simply some data. So lets see how to reflect changes to this variable.

## Reflecting variable changes

In order to see changes to your created variable, it needs to be hooked up to the property of your Source Widget within the Class Widget you want to change. To do this 2 steps are required:

1. Make the Source Widget within your Class Widget “Variable”. This is done by checking the checkbox labelled “Is Variable” at the top right of the details view for this Source Widget, as seen in Illustration 6 below.

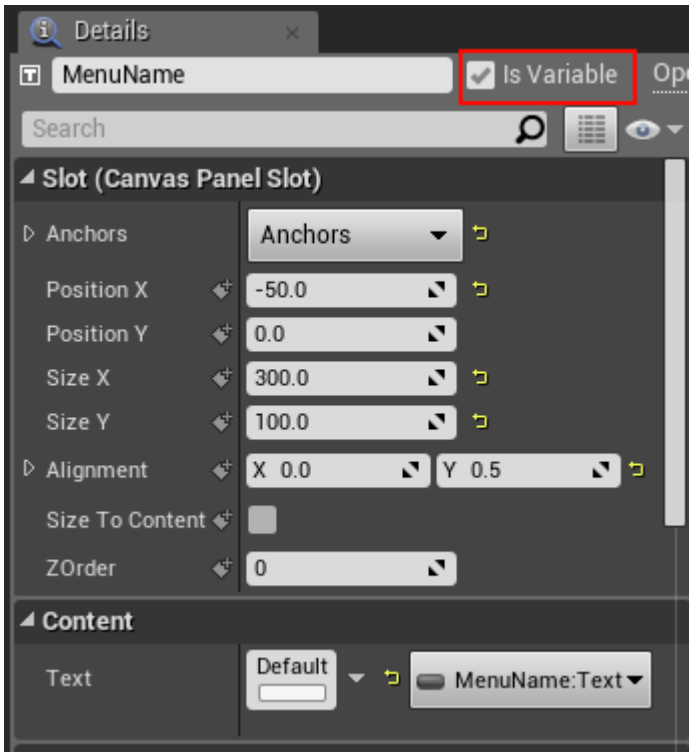


*Illustration 6: Detail view, showing the “is variable” option in red.*

2. You must then go to the Graph editor for this Class Widget, and link a few nodes to the “Event On Synchronize Properties”. These nodes must set the value within your Source Widget, which is now available as a variable in the graph editor, to the value of the Translatable variable you created earlier. An example can be seen in Illustration 7 below.

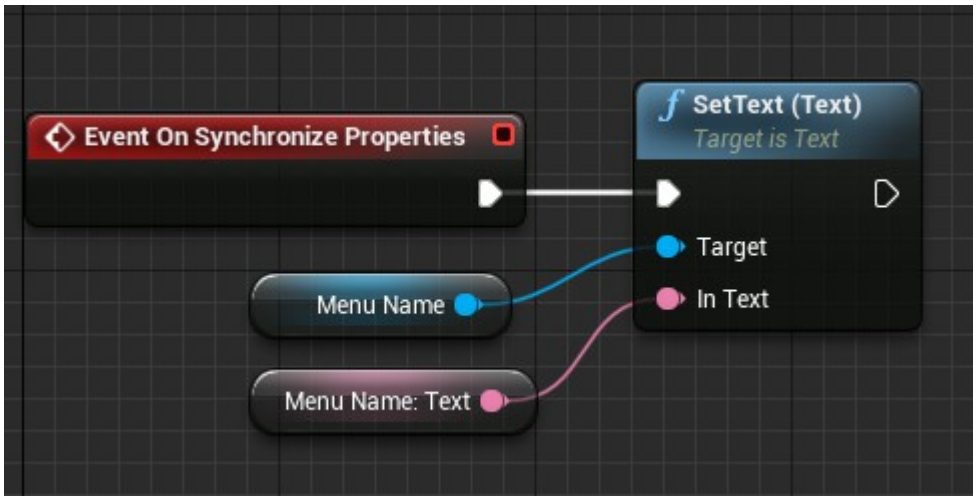


*Illustration 7: Graph for variable reflection nodes*

Note that if the “Event On Synchronize Properties” is not already present, it should be easy enough

to find by right clicking and using the search function.
When this is all hooked up, any time you change the variable on an instance of your Class Widget it should be reflected in the UMG editor immediately.

### *Example walkthrough*

Now lets look at an example of everything we just mentioned. We'll create a menu in UMG using a few Source Widgets, and create a Lua translation for that menu. Then we'll create a new Lua class Widget,  add it to that menu as an additional Source Widget, and update our translation. Finally we'll add a variable to that Lua Class Widget, set its value on our Menu Widget, and update our translation again.

## Creating the Menu widget

Let's create a simple user widget to translate as a menu. Create a new Widget Blueprint, in our example we call it MyMenu. Then add a few translatable source widgets to it. For our example we'll add an Image, a TextBlock, a Button and a BackgroundBlur.

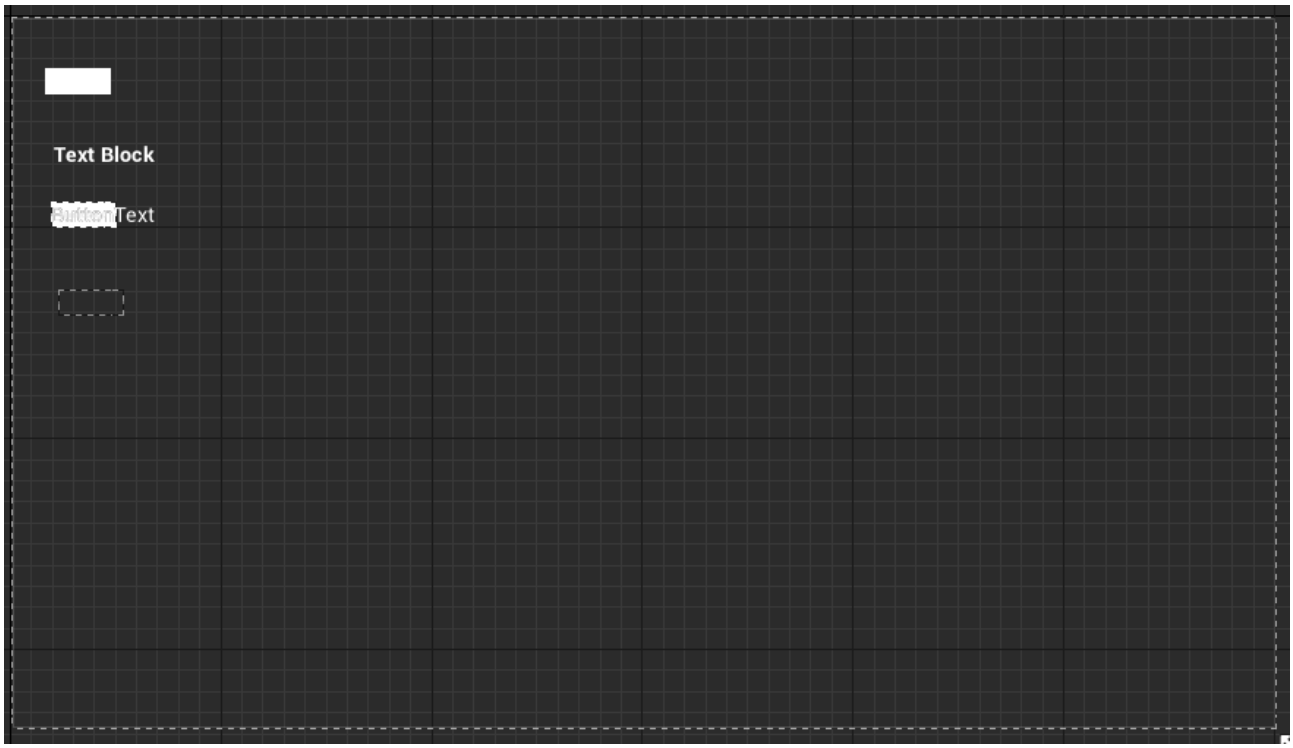

*Illustration 8: Initial widget placement*

Now that we have some widgets, lets set a few values on them.
On the text block we:
Set the X size to 300 and the Y size to 50, move it to the center of our menu, set its color to red, increase its font size to 30 and set the justification to center. We set the text to “My Menu Text”.
On the button we:
Set the X size to 250 and the Y size to 80, move it to below our text block, change the label to “Example Button”, and change the outline size to 2.
On the backgroundBlur we:
Set the X and Y size to 300, move it to below our button, set the blur strength to 10 and set its Z-order to -1.

On the Image:
Set the X and Y size to 250, move it on top of the backgroundblur, set a texture on its brush image, and set its Z-order to -1.

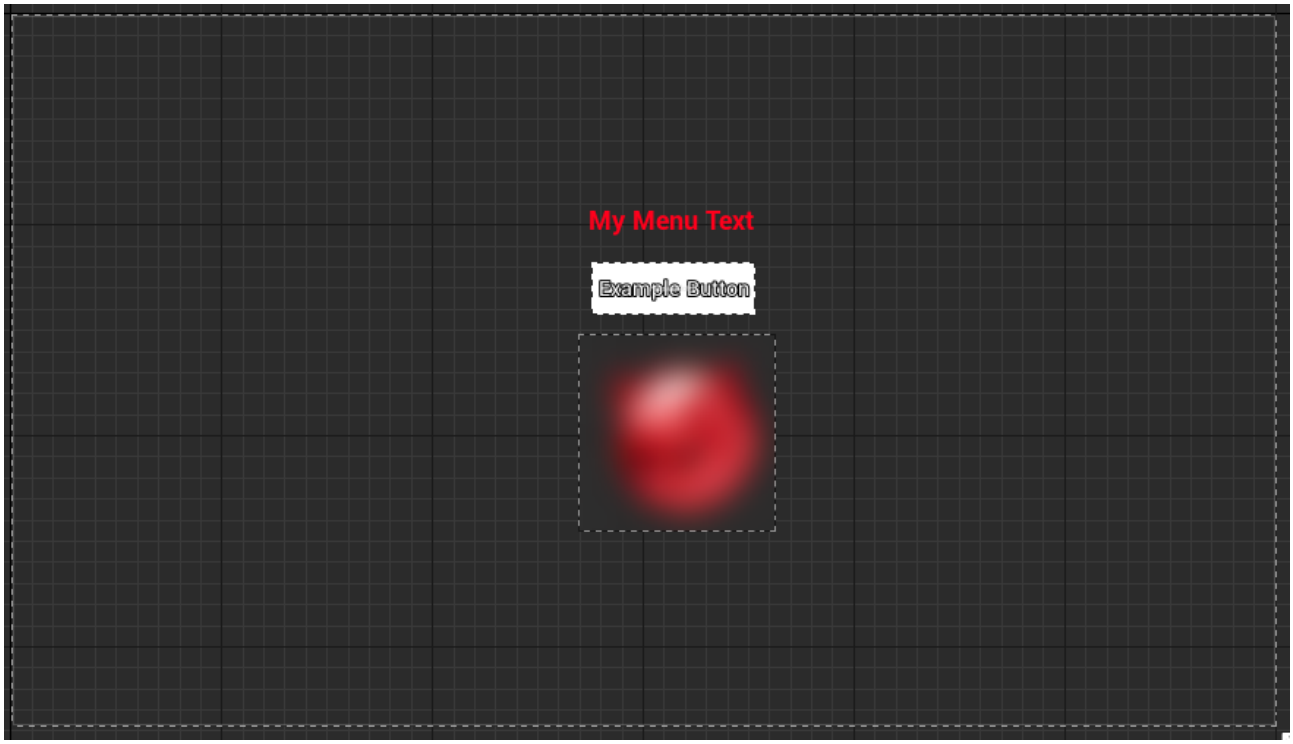


*Illustration 9: What our example now looks like in the UMG editor.*

Now that we have a few widgets set up, let's create the first menu translation. Open the Lua Translator by pressing the tool button at the top of the UMG editor.
Now lets set our translation variables. First of all, check that the Target Widget is set to the name of your Widget Blueprint, MyMenu in our case. Then set the TranslationType to “Menu” if it is not set to that already.
Now select a file directory where you want to save your translated menu file. You can either enter the path manually or use the “select save directory” button to open the windows file explorer. This directory should most likely be where you plan to store all your Lua menu files. You can always move the file after translating.

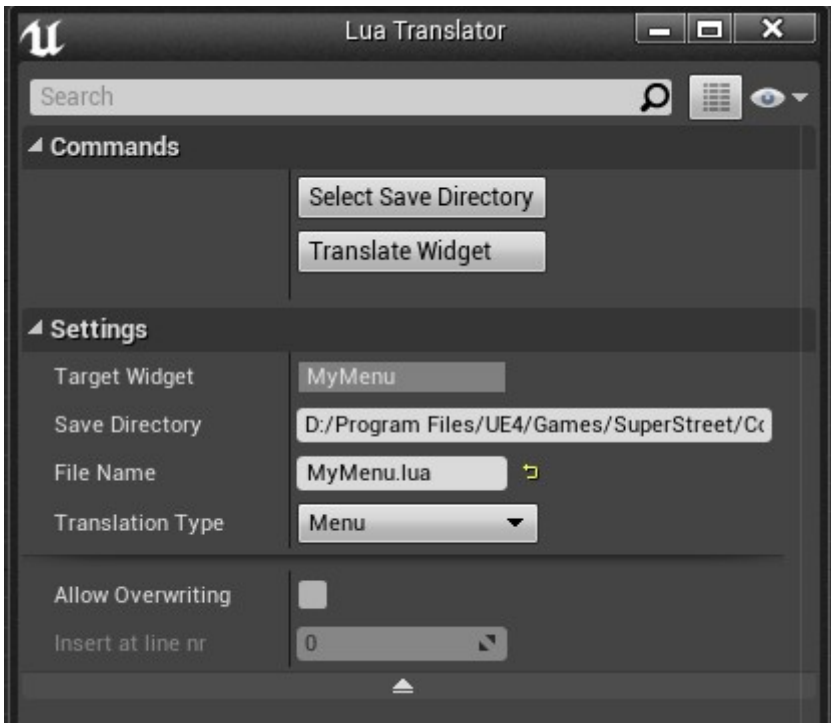

*Illustration 10: Our translation settings*

Optionally you can change the file name for your translated file. If the two settings at the bottom don't show up for you they are hidden in the advanced tab, which you can open by pressing the arrow. We don't need those settings for now though.

Once everything is set, press the Translate Widget button. You should get a message pop up telling you your translation has succeeded! If something has gone wrong you may receive an error telling you that the translation has failed, along with the reason for failing. Some frequent causes for failure are that a file with the desired name already existed or that the target directory is read only.

*Illustration 11: The translation has succeeded, and returned a result message.*

Lets take a quick look at the translated menu file.

```lua
DefineClass("MyMenu", BaseMenu, function(self, Outer, ...)
    self.Super(Outer, ...)
end)

function MyMenu:AddElements()
    self.Super:AddElements()

    local TranslatableTextBlock_64 = UILabel(self.Outer)
    TranslatableTextBlock_64:SetText("My Menu Text")
    TranslatableTextBlock_64:SetColor(Color(1.0, 0.0, 0.009549, 1.0))
    TranslatableTextBlock_64.Font = TranslatableTextBlock_64:MakeFont(
    nil, Resources:LoadSync("/Engine/EngineFonts/Roboto.Roboto"),
    "Bold", 30, 0, false, nil, Color(0.0, 0.0, 0.0, 1.0))
    TranslatableTextBlock_64:SetFont(TranslatableTextBlock_64.Font)
    TranslatableTextBlock_64:CheckFormatting()
    TranslatableTextBlock_64:SetJustification(Justification.Center)
    TranslatableTextBlock_64:SetRenderTransformPivot(Vector2D(0.500, 0.500))
    TranslatableTextBlock_64:SetAnchors(Vector2D(0.000, 0.000), Vector2D(0.000, 0.000))
    TranslatableTextBlock_64:SetSize(300.0, 50.0)
    TranslatableTextBlock_64:SetPos(852.0, 288.0)

```

*Illustration 12: The start of our menu translation file*

In illustration 12 you can see the start of our menu translation file. It defines a class for the menu so that it can be opened from other Lua code, and begins adding the source widgets one by one. In the example image you can see it adding the TextBlock (as a UILabel) and setting all the relevant values on it. Similar blocks will be present for all other source widgets we had.

```lua

    self:AddChild(TranslatableTextBlock_64)
    self:AddChild(TranslatableImage_59)
    self:AddChild(TranslatableBackgroundBlur_43)
    self:AddChild(TranslatableButton)
    self.TranslatableTextBlock_64 = TranslatableTextBlock_64
    self.TranslatableImage_59 = TranslatableImage_59
    self.TranslatableBackgroundBlur_43 = TranslatableBackgroundBlur_43
    self.TranslatableButton = TranslatableButton

end

```

*Illustration 13: End of the menu translation file, showing the parenting.*

And in illustration 13 you can see the end of our example file. Here all the created widgets get parented to the root canvas of our menu, and member variables are set for this menu so that the widgets can be accessed later. Note that if you re-parented any widgets in the UMG editor, they will not be parented to the root canvas but will have their parents set correctly here.

## *Creating a new Lua class*

Next we'll look at translating a Widget Blueprint as a new Lua class, and how to use it in our Menu. For our example we'll create a grey bar to go at the top of our screen, with a textblock on it to display the name of whatever menu we are in. We can then reuse this in all of our other menus.

We add an image to our canvas, stretch it across the whole canvas and set its color to a slightly transparent grey. We then also add a text block to the canvas at the top of the screen.

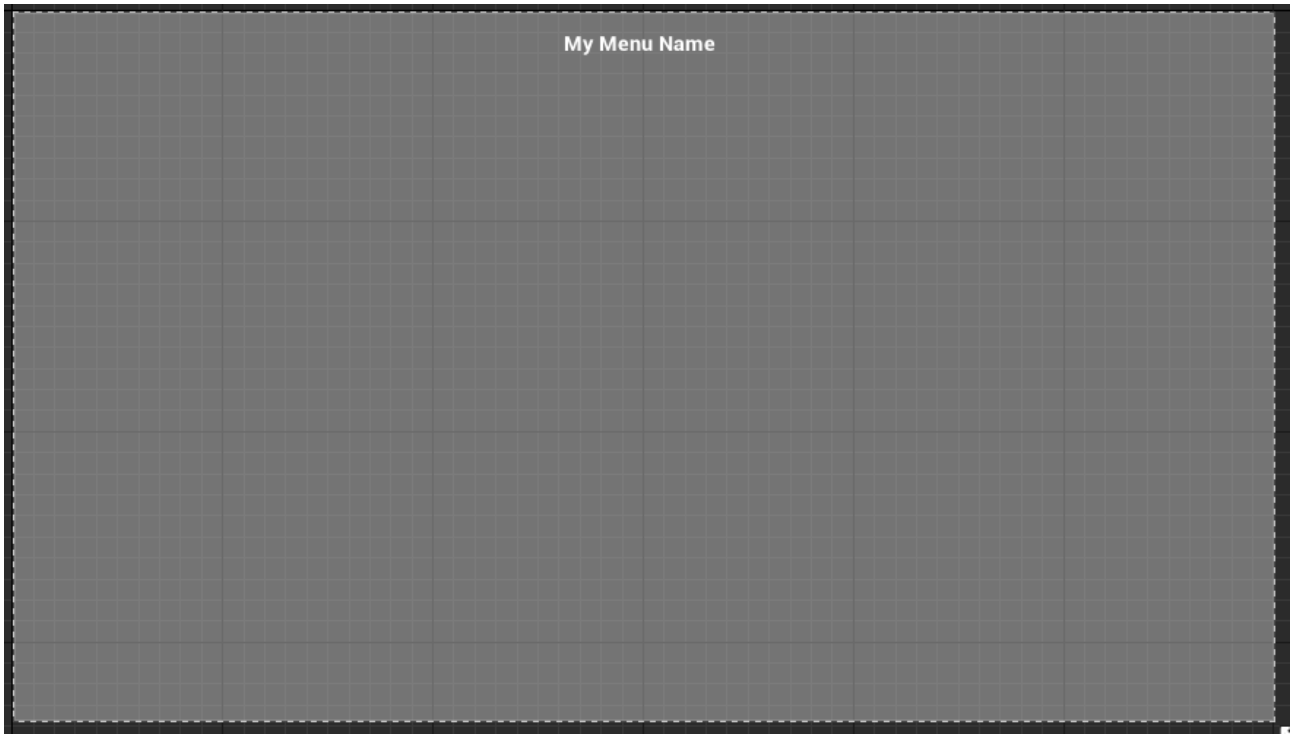


*Illustration 14: Our grey image and textblock on top of it.*

After that is done, we set the LuaClass variable on the widget root to MenuTopBar.

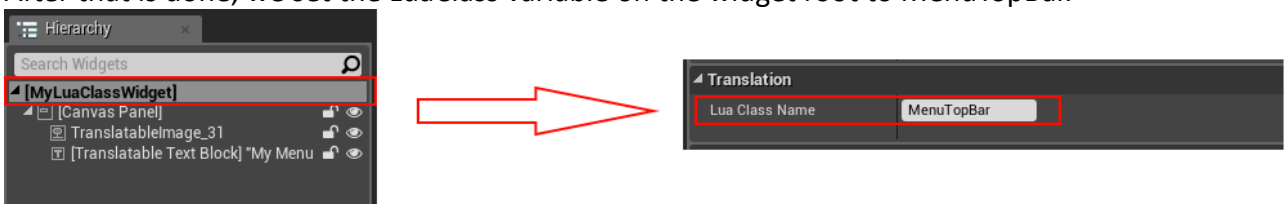


*Illustration 15: Select the widget root and find the LuaClassName variable in the details view*

Now lets translate the class. Follow the same procedure as before, but make sure to select “Class” as your translation type this time. You probably also want to change the target folder to wherever you are going to store your Lua classes.
Also notice how the File Name automatically got set to what you entered as your LuaClass variable. You should not change this, as this determines what the class definition will be in the translated file. If you do change it, make sure to go back and change the variable afterwards (or just translate again and overwrite the file).

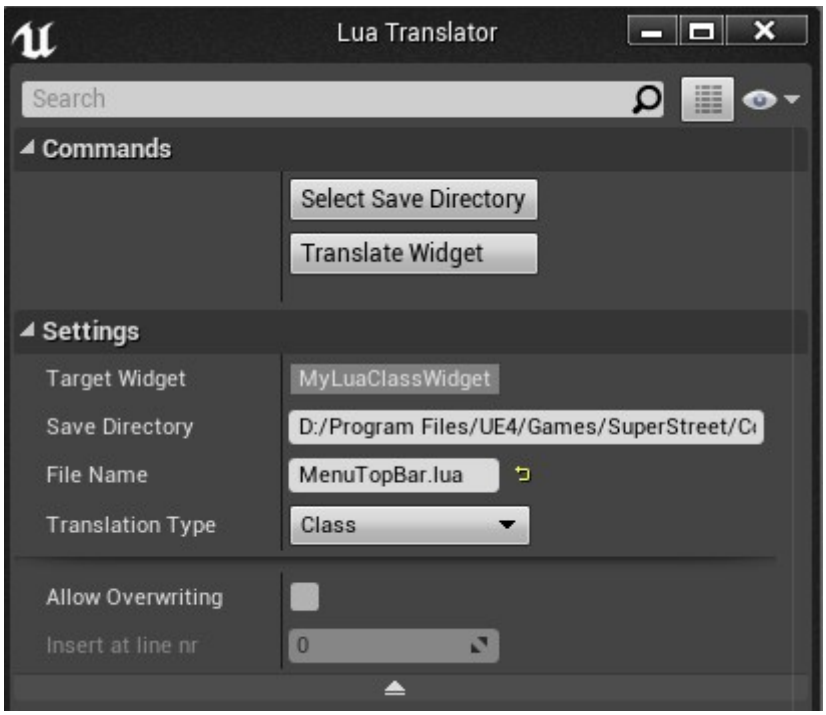


*Illustration 16: Our example settings for Class translation.*

And again, lets take a look at our translation file.

```lua
require "Engine/UI/UICanvas"

DefineClass("MenuTopBar", UICanvas, function(self, Outer)
    self.Super(Outer)

    local TranslatableImage_31 = UIImage(self.Outer)
    TranslatableImage_31:SetColor(Color(0.7, 0.7, 0.7, 0.4))
    TranslatableImage_31:SetBrush(TranslatableImage_31:MakeBrush(
    nil, Color(1.0, 1.0, 1.0, 1.0), Vector2D(32.000, 32.000), Margin(0.0, 0.0, 0.0, 0.0), 3, 0))
    TranslatableImage_31:SetRenderTransformPivot(Vector2D(0.500, 0.500))
    TranslatableImage_31:SetAnchors(Vector2D(0.000, 0.000), Vector2D(1.000, 1.000))

    local TranslatableTextBlock_30 = UILabel(self.Outer)
    TranslatableTextBlock_30:SetText("My Menu Name")
    TranslatableTextBlock_30.Font = TranslatableTextBlock_30:MakeFont(
    nil, Resources:LoadSync("/Engine/EngineFonts/Roboto.Roboto"),
    "Bold", 24, 0, false, nil, Color(0.0, 0.0, 0.0, 1.0))
    TranslatableTextBlock_30:SetFont(TranslatableTextBlock_30.Font)
    TranslatableTextBlock_30:CheckFormatting()
    TranslatableTextBlock_30:SetJustification(Justification.Center)
    TranslatableTextBlock_30:SetRenderTransformPivot(Vector2D(0.500, 0.500))
    TranslatableTextBlock_30:SetAnchors(Vector2D(0.000, 0.000), Vector2D(0.000, 0.000))
    TranslatableTextBlock_30:SetSize(250.0, 40.0)
    TranslatableTextBlock_30:SetPos(828.0, 28.0)

    self:AddChild(TranslatableImage_31)
    self:AddChild(TranslatableTextBlock_30)
    self.TranslatableImage_31 = TranslatableImage_31
    self.TranslatableTextBlock_30 = TranslatableTextBlock_30
end)

```

*Illustration 17: The lua class translation for our MenuBar.*

Now that we have the translated Lua Class, lets add it to our menu. For our example we add it to

our menu widget, set it to stretch horizontally at the top of the screen, and set its Y size to 100.

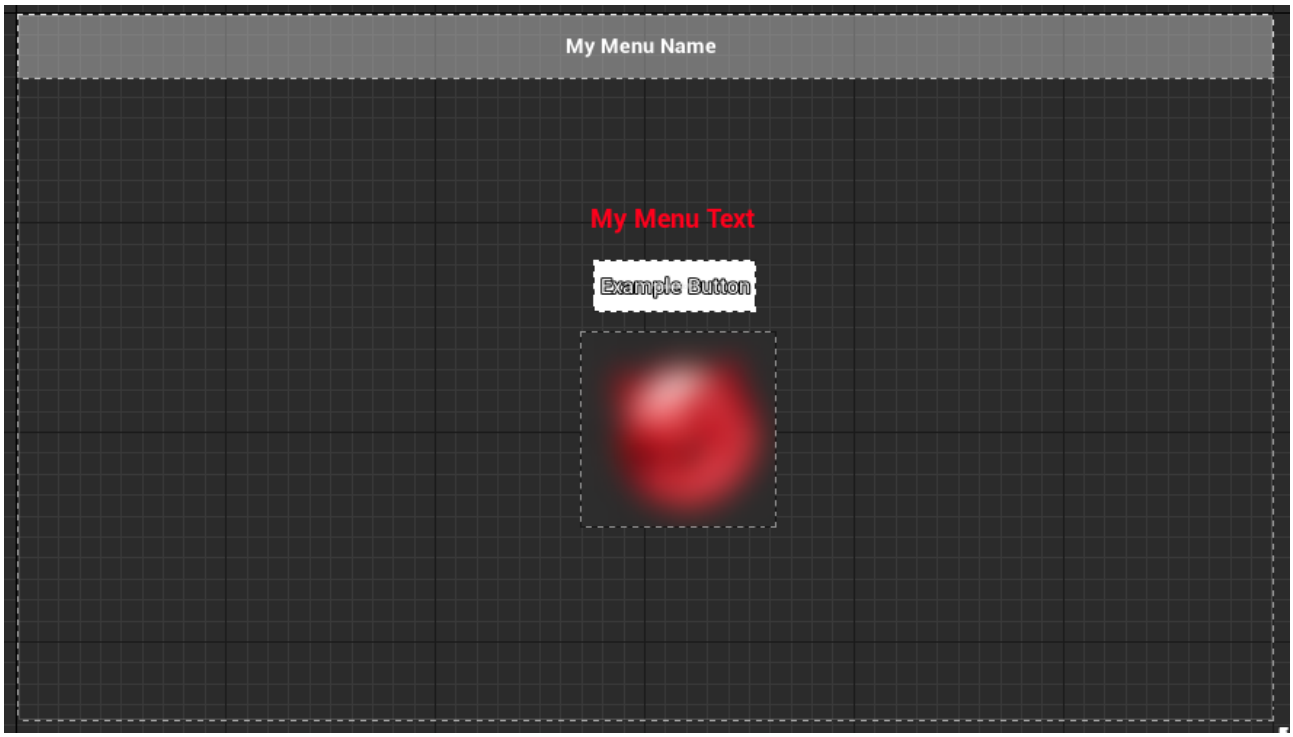


*Illustration 18: Our menu widget with the added bar at the top.*

Now lets translate our menu file again, since it's been changed. This time, in the translators advanced options tick the “Allow Overwriting” option. This will allow us to overwrite our previous menu file without having to go find it and delete it first. Note that this option can be dangerous, as it will overwrite ALL file content, not just the lines that have changed!

```
55
56      local MyLuaClassWidget = MenuTopBar(self.Outer)
57      MyLuaClassWidget:SetRenderTransformPivot(Vector2D(0.500, 0.500))
58      MyLuaClassWidget:SetAnchors(Vector2D(0.000, 0.000), Vector2D(1.000, 0.000))
59      MyLuaClassWidget:SetSize(0.0, 100.0)
60
```

*Illustration 19: The added lines in our menu translation file*

In the menu translation file we can now see a new MenuTopBar being created (or whatever you named your Lua Class). Any changes to that class will now automatically apply to all files that use it.
Now this was all well and good, but there's one thing we're still missing...

## Translating UMG variables

The menu name textblock we added to our example top bar is the same in every menu. Thankfully we have a way to change this.
In the UMG editor for your Lua Class Widget, rename the widget you want to change a property of to something distinct. In our example we'll rename our textblock to “MenuName”. Also tick the “Is Variable” box at the top of the details view, this will be useful later.
Now go to the graph editor. In there, create a new variable of the same type as the property you want to change, in our case a Text.
Now the variable you create must have a few properties set on it.

First of all, it must be *instance editable*.
Secondly, you should set a default value for it. In our case we set that to *Default Menu Name*.
Finally, you must name the variable. This must be done very precisely to one of two formats: *SourceWidgetName,PropertyName* or *SourceWidgetName:PropertyName*.
Refer to the explanation earlier in this document for when to use either format. In our example we name our variable MenuName:Text.

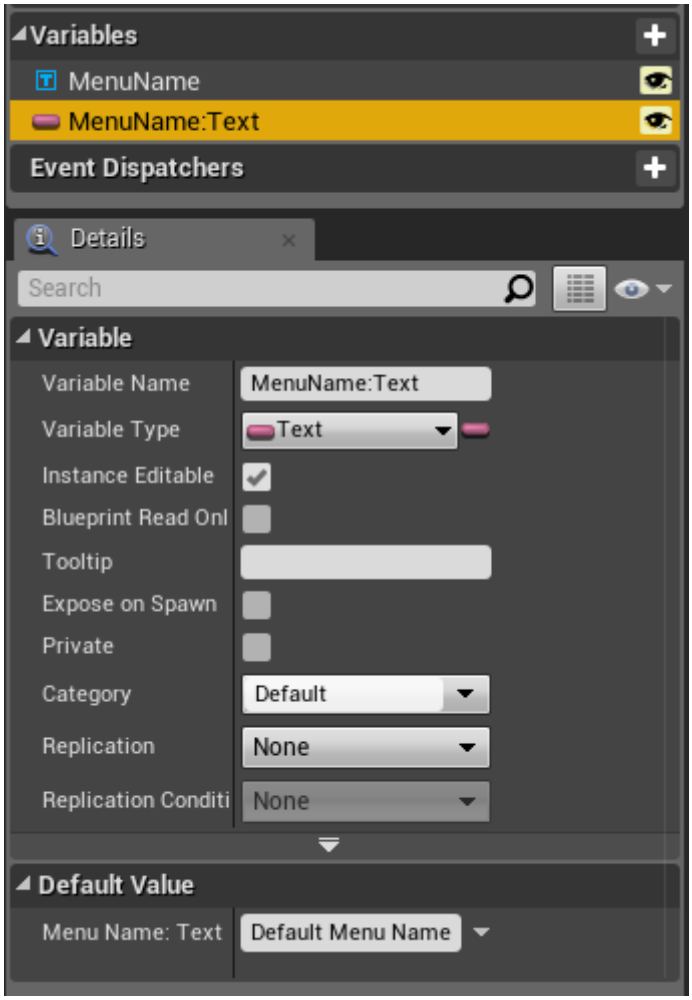


*Illustration 20: Our UMG variable setup*

Now if you go to the menu widget, you should see a variable added under the “Default” category (or a different one if you set one on the variable). Here you can change the value, and it will be translated and set when you translate your menu again. However, before we do that there is one more thing we should do. If you change the value now you will notice that it will not update the instance of your Class Widget in your Menu Widget. We can change that very easily by going back to the graph editor we were at before.
Once there, you have to add only a few nodes. Add the “On Synchronize Properties” event, and from there set the property on your Source Widget to the value in your created variable. It should look something like this:

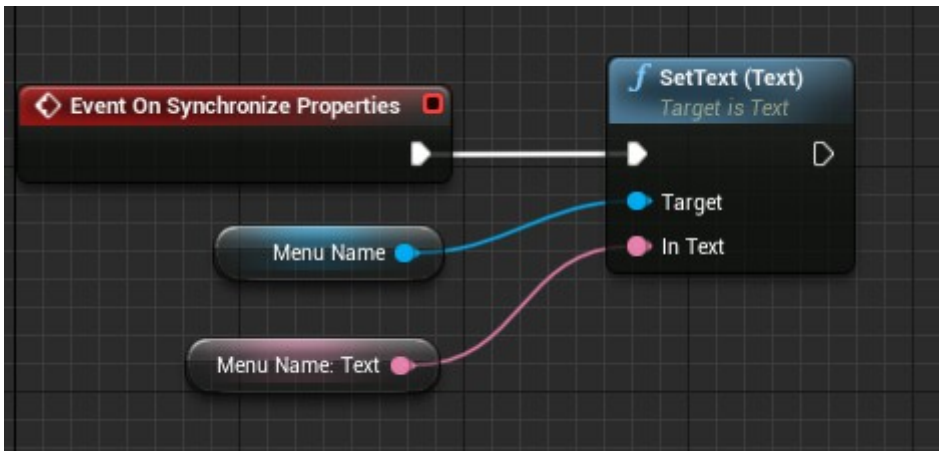

*Illustration 21: Our synchronize nodes*

Now if you change the value of your created variable on an instance of your Class Widget it will immediately update and be displayed, so you can see what you're editing. Lets translate both our files one last time, starting with the Lua Class (to update the variable names we may have changed), and then translate the menu file again. Remember to select the right translation type each time, and tick the Allow Overwriting box if you still have the previous file.

I have cut the end result image from this guide, as it showed in-game footage that I'm not sure I can show. It was originally included in this documentation as it was made for internal use at Team6.