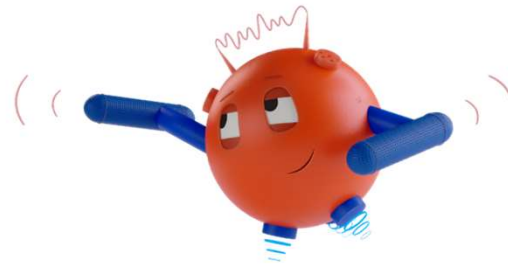


July 13, 2023

Mastering Custom Activities



Agenda



- **Introductory Notions**
 - .NET
 - UiPath Project Compatibility
 - WF vs Core WF
- **Custom Activities intro - source**
 - What is a custom activity?
 - Types of Custom Activities
- **How To Create a Custom Activity using code**
 - Adding dependencies
 - Understanding .csproj files
 - Implement the logic
 - Build
- **How to publish a Custom Activity**
- **How to debug a custom Activity?**
- **How To Create a Custom Activity using the Activity Creator**
 - Understanding folder structure
 - Create a simple activity
 - Create a scope activity
 - Understanding activity structure
 - Understanding activity design
- **Useful tips**
- **Targeting multiple .NET Runtimes**
 - Prerequisites
 - Step by step migration

Introductory Notions

.NET



- UiPath Automations are developed and executed using **.NET**
- Depending on the UiPath Project **Compatibility Type** you use, internally UiPath can use a different “**version**” of **.NET**:
 - In 2002, Microsoft released **.NET Framework**, a proprietary development platform for creating Windows applications.
 - In 2014, Microsoft introduced **.NET Core** as a cross-platform (**Windows, Linux, MacOS**), open-source successor to .NET Framework. Subsequently, the name of this framework was changed simply to **.NET**
 - **These 2 are not compatible!!!**



.NET != .NET Framework!!!

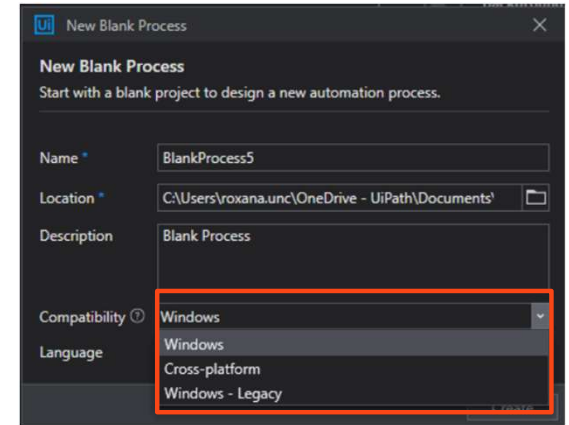
(important!)

UiPath Project Compatibility



Compatibility Type	.NET Runtime	Target Framework Moniker (.csproj file)
Windows Legacy	.NET Framework 4.6.1	net461
Windows	.NET 6 – Windows Specific	net6.0-windows
Cross-platform / Windows	.NET 6	net6.0

- **Windows Legacy** is deprecated since UiPath Studio 22.10. It should not be used to create new projects.
- **Cross-Platform** automation projects are designed to run cross-platform on Windows, Linux, and macOS operating systems (great for **Serverless Robots**). **Windows specific features are not available** (like desktop applications automation OR some excel activities)
- **Windows** automation projects are still using .NET but are platform-specific. All windows specific features and packages can be used.



When building custom activities, we must target the **appropriate runtime** for the **UiPath Compatibility type** we plan on using.

Windows Workflow Foundation and Core WF

Windows Workflow Foundation:

- The foundation **of the workflow designing and executing experience** in UiPath Studio is built using the [Windows Workflow Foundation Framework](#).
- **This Framework was released as part of .NET Framework** (not compatible with **.NET Core**)
- This framework has both:
 - **A run-time engine** that allows us to execute workflows.
 - **A rehostable designer** to implement said workflows.

Core WF:

- UiPath created a ported version of Windows Workflow Foundation to .NET 6, called [CoreWF](#)
- This is the version **used for Windows** and **Cross-Platform** UiPath Projects
- It offers a very similar experience to Workflow Foundation, both when used in UiPath Studio and when building custom activities.

Custom activities intro



What is a custom activity?

Activities are the building blocks used in UiPath Studio to design automation projects.



Custom Activities are .Net classes that inherit one of the WF specific activity classes (NativeActivity, CodeActivity, AsyncCodeActivity)



Custom activities benefits

Create reusable components and share between multiple projects

Easy change/update the logic by simply update in one place

Integrate complex logic in your RPA projects

Extend the use of RPA by creating new ways of interacting with different applications

Custom Activities lifecycle

1

Phase One

Create Custom Activity

- Create a library project in Visual Studio
- Edit **.csproj** files
- Create activities
- Implement the logic
- Design activities
- Build

2

Phase Two

Publish Custom Activity

- Generate NuGet package
- Add NuGet package to a NuGet Feed accessible to our robots/developers

3

Phase Three

Consume Custom Activity

- Create/Open a project in UiPath Studio
- Add NuGet package in phase two as dependency
- Drag-drop the activity
- Update parameters
- Run the project

Custom Activities types

CodeActivity

- Helps us to **create basic activities** that execute simple pieces of logic.
- This is used to create very simple, linear activities.

NativeActivity

- Helps us to create more complex activities, like **scope activities**.
- More advanced control flow for its execution or the execution of its child activities
- With a Native activity we can Abort, Cancel or Schedule the Child activity.

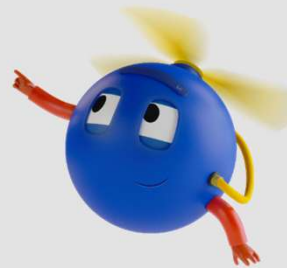
AsyncCodeActivity

- Enables derived activities to implement **asynchronous execution logic**.
- Leverages the **Asynchronous Programming Model** (**BeginAsync** & **EndAsync** methods).

ContinuableAsyncCodeActivity & ContinuableAsyncNativeActivity

- Only with the **UiPath Activity Creator Package**
- These leverage the **Task-based Asynchronous Pattern** (**async** keyword, **Task** class, etc.)

How to create Custom Activities using code

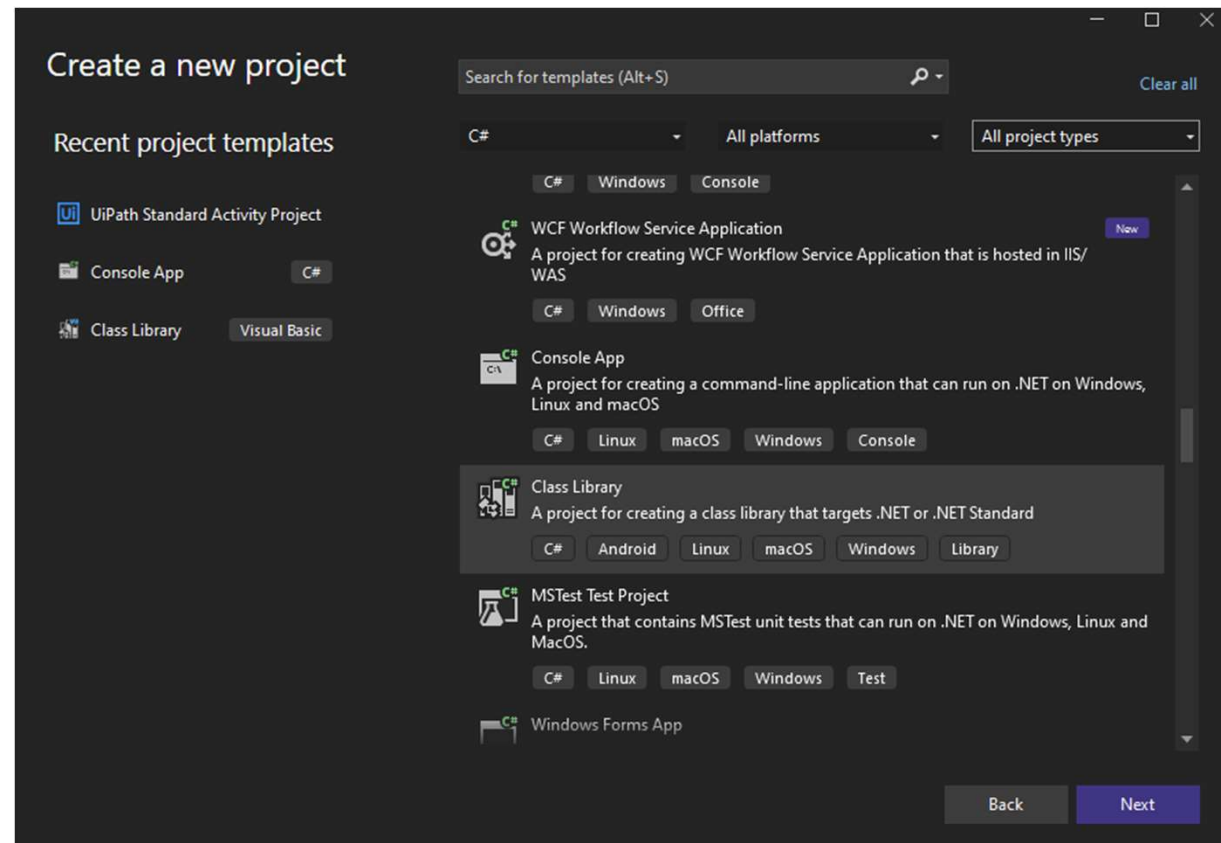
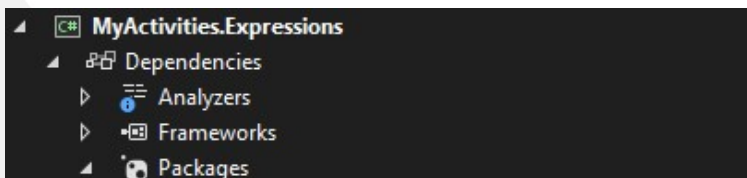


Prerequisites:

- ✓ Visual Studio Community/Professional/Enterprise 2022 with the .NET desktop development workload installed.
- ✓ UiPath's Marketplace Feed (<https://gallery.uipath.com/api/v3/index.json>) as a package source in Visual Studio. If this feed is not available in Visual Studio during development, the activities ***will not build successfully***.
- ✓ NET 6
- ✓ UiPath Studio >=22.10.x

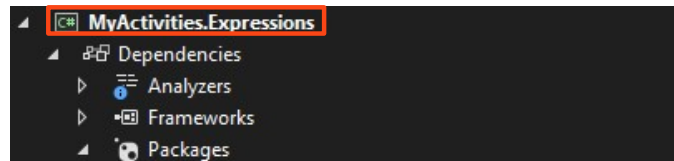
1. Create project in Visual Studio

1. Open Visual Studio >> Create New Project >> **Class Library**
2. **Select framework .NET6.0** in order to be compatible with UiPath Studio >=22.10
3. **Create Project.**



2.a. Add Dependencies

1. Start by Editing the .csproj file (double click project item)



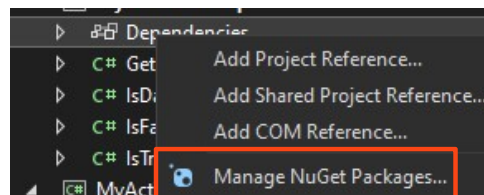
These packages are also used internally by UiPath Studio, to avoid conflicts we must mark them as private:
PrivateAssets="All"

2. Add the latest versions of the following packages ([you can copy them from here](#)):

```
<ItemGroup>
  <PackageReference Include="UiPath.Workflow.Runtime" Version="6.0.0-20220401-03" PrivateAssets="All" />
  <PackageReference Include="UiPath.Workflow" Version="6.0.0-20220401-03" PrivateAssets="All" />
  <PackageReference Include="System.Activities.Metadata" Version="6.0.0-20220318.2" PrivateAssets="All" />
</ItemGroup>
```

3. Add any 3rd party NuGet packages, too.

! Alternatively, use the NuGet Package Manager to add the packages without editing the .csproj:



2.b. Understanding .csproj files

At least these properties need to be updated:

- **Project SDK**
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop" >
- **Target frameworks** [\(see slide 5 for values\)](#)
<TargetFrameworks>net6.0-windows</TargetFrameworks>
- **OutputPath** is the where the project will be built.
<OutputPath>..\..\Output</OutputPath>
- Any packages we need to use are referenced using a **PackageReference** XML Node. These are grouped using **ItemGroup** nodes:
 <ItemGroup>
 <PackageReference Include="ClosedXML" Version="0.97.0">
 ...
 </ItemGroup>

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop" ToolsVersion="Current">
  <PropertyGroup>
    <TargetFrameworks>net6.0-windows</TargetFrameworks>
    <RootNamespace>MyActivities.Excel.Activities</RootNamespace>
    <AssemblyName>MyActivities.Excel.Activities</AssemblyName>
    <OutputPath>..\..\Output</OutputPath>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <PackageId>MyActivities.Excel.Activities.Implementation</PackageId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="ClosedXML" Version="0.97.0" />
    <PackageReference Include="System.Activities.Metadata" Version="6.0.0-20220318.2" PrivateAssets="All" />
    <PackageReference Include="UiPath.Workflow.Runtime" Version="6.0.0-20220401-03" PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\MyActivities.Excel\MyActivities.Excel.csproj" PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <Compile Update="Properties\Resources.Designer.cs">
      <DesignTime>True</DesignTime>
      <AutoGen>True</AutoGen>
      <DependentUpon>Resources.resx</DependentUpon>
    </Compile>
  </ItemGroup>
  <ItemGroup>
    <EmbeddedResource Update="Properties\Resources.resx">
      <Generator>PublicResXFileCodeGenerator</Generator>
      <LastGenOutput>Resources.Designer.cs</LastGenOutput>
    </EmbeddedResource>
    <EmbeddedResource Update="Properties\Resources.*.resx">
      <DependentUpon>Resources.resx</DependentUpon>
    </EmbeddedResource>
  </ItemGroup>
  <Import Project="..\..\Shared\UiPath.Shared.Activities\UiPath.Shared.Activities.projitems" Label="Shared" />
</Project>
```

Extra content may be present here, in case **additional resources** or **Satellite Assemblies** (useful for localization) are needed.

3. Create Activity class

1. **Add class.** Right click on project >> Add >> New Item... >> C# Class
2. Set a **Class name**. The class name will be the activity name.
3. Add using **System.Activities** to the new class.
4. The class needs to inherit one of the WF specific activity classes (NativeActivity, CodeActivity, AsyncCodeActivity)

```

0 references
public class GetDate : CodeActivity
{
    [Category("Output")]
    1 reference
    public OutArgument<DateTime> Date { get; set; }
    0 references
    protected override void Execute(CodeActivityContext context)
    {
        var date = DateTime.Now;
        Date.Set(context, date);
    }
}
  
```

4. Implement the logic

Define Parameters

- Depending on the type of parameter, we should use the Generic classes **InArgument<T>**, **OutArgument<T>** or **InOutArgument<T>** to define our arguments
- Attributes such as **Category**, **RequiredArgument**, **Description**, **DisplayName** and many others can be used to customize the look and behavior of each one of our arguments

```
[DisplayName("Output Date")]
[Category("Output")]
[Description("Simple Output Parameter representing the output date time")]
[RequiredArgument]
public OutArgument<DateTime> Date { get; set; }
```

Define Activity Logic

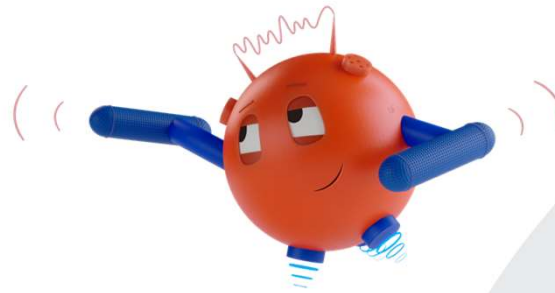
- For **CodeActivity** and **NativeActivity**, we need to overwrite the **Execute** method.
- For **AsyncCodeActivity** we need to overwrite **BeginExecute** and/or **EndExecute** methods.
- By using **NativeActivity** we can **Abort**, **Cancel** or **Schedule** child activity by using one of the following methods: **AbortChildInstance**, **CancelChild**, **CancelChildren**, **ScheduleAction**.

```
protected override void Execute(CodeActivityContext context)
{
    var date = DateTime.Now;
    Date.Set(context, date);
}
```

5. Build the solution

- ! The generated .dll file will have the same name as the project.
- ! The generated .dll file will be used to create the .nupkg.
- ! The .dlls will be created in **OutputPath** specified in the .csproj file.
- ! **Unit Tests** can be configured to run after build, thus helping eliminate obvious bugs
- ! Once your DLL(s) are generated, you can use the **NuGet Package Manager** [to build the package](#). However, it's best to configure your solution OR use a CI/CD flow to automatically build your package.

How to create a Scope Activity using code



1. Create ScopeActivity class

1. Create a new class and **inherit NativeActivity**
2. Override the **Execute** Method
3. Add a constructor
4. Create the necessary arguments

```
public sealed class NotepadAppScope : NativeActivity
{
    // Define an activity input argument of type string
    [LocalizedCategory("Input")]
    public InArgument<string> FilePath { get; set; }

    public NotepadAppScope()
    {
    }

    // If your activity returns a value, derive from CodeActivity<TResult>
    // and return the value from the Execute method.
    protected override void Execute(NativeActivityContext context)
    {
    }
}
```



2. Create the Scope Property

1. NativeActivities can use a property of type **ActivityAction<T>** to store the Child Activities
 - **T** is the type of the object we want to pass as argument to child activities

```
// A tag used to identify the scope in the activity context
internal static string ParentContainerPropertyTag => "ScopeActivity";

public NotepadAppScope()
{
    Body = new ActivityAction<string>
    {
        Argument = new DelegateInArgument<string>(ParentContainerPropertyTag),
        Handler = new Sequence { DisplayName = "Do" }
    };
}
```

```
public sealed class NotepadAppScope : NativeActivity
{
    [Browsable(false)]
    public ActivityAction<string> Body { get; set; }
```

2. In the constructor, we initialize our **Body** property with an **empty sequence**
3. The **argument** passed to the Children also **needs to have a name**.
 - Children activities will use this name to extract it from their context.



3. “Invoke” child activities

1. In the execute method, use the **ScheduleAction** method to Invoke the **Child Activities** located in the body property.
 - Here is where you also **pass the argument to the child activities**.
2. Optionally, you can also add **OnFaulted/OnCompleted** callbacks

```
protected override void Execute(NativeActivityContext context)
{
    // Obtain the runtime value of the Text input argument
    string text = context.GetValue(this.FilePath);

    context.ScheduleAction<string>(Body, text);
}
```

```
protected override void Execute(NativeActivityContext context)
{
    // Obtain the runtime value of the Text input argument
    string text = context.GetValue(this.FilePath);

    context.ScheduleAction<string>(Body, text, OnCompleted, OnFaulted);
}

private void OnFaulted(NativeActivityFaultContext faultContext,
    Exception propagatedException, ActivityInstance propagatedFrom)
{
    //TODO
}

private void OnCompleted(NativeActivityContext context,
    ActivityInstance completedInstance)
{
    //TODO
}
```

4. Create a child activity

1. Create another class that overrides **CodeActivity**.
2. The only extra action is **retrieving the argument passed by the Parent Activity to the ActivityDelegate<T>**
 - This property can be retrieved from the **CodeActivityContext**
 - We can identify it using the name we set to the **DelegateInArgument<T>** in the scope activity

```
public sealed class ReadText : CodeActivity
{
    // Define an activity input argument of type string
    [RequiredArgument]
    [Category("Output")]
    public OutArgument<string> Text { get; set; }

    // If your activity returns a value, derive from CodeActivity<TResult>
    // and return the value from the Execute method.
    protected override void Execute(CodeActivityContext context)
    {
        //Retrieve property from parent scope
        //(this will throw an exception if activity is not inside the appropriate scope)
        var property = context.DataContext.
            GetProperties() [NotepadAppScope.ParentContainerPropertyTag];
        string filePath = property.GetValue(context.DataContext) as string;

        // Obtain the runtime value of the Text input argument
        context.SetValue(Text, File.ReadAllText(filePath));
    }
}
```

```
Body = new ActivityAction<string>
{
    Argument = new DelegateInArgument<string>
        (ParentContainerPropertyTag),
```



```
//Retrieve property from parent scope
//(this will throw an exception if activity is not inside the appropriate scope)
var property = context.DataContext.
    GetProperties() [NotepadAppScope.ParentContainerPropertyTag];
string filePath = property.GetValue(context.DataContext) as string;
```


5.b. Custom Activities Design Phase



CacheMetadata is a method that is called **at the design time** of your activity by the **Workflow Runtime**.

When is CacheMetadata called?

- When dragging your activity into a workflow.
- When modifying an argument of your activity.
- When running your workflow, before calling your Execute method.
- When moving your activity inside of your workflow.

Why use CacheMetadata?

- By default, this method exposes parameters, variables, activity delegates, etc. to the Workflow Runtime.
- **Without this method, the runtime has no idea about what's inside our activity!**
- We can also use it to implement more complex validation logic or add constraints



If we **override CacheMetadata**, we must also **invoke its base class implementation**, otherwise our workflow will throw an exception at runtime.


8.b. Custom Activities Design Phase



1. We can **override** the **CacheMetadata** method for **any type of custom activity**, just like in the example to the right.
2. Method **AddValidationError** can be used to add a custom error message if needed.
3. But we must also **invoke its base class implementation**, otherwise our workflow will throw an exception at runtime.

```
protected override void CacheMetadata(NativeActivityMetadata metadata)
{
    //Dummy Validation Error
    if(FilePath == null)
        metadata.AddValidationError("File Path Argument is mandatory!!!");

    base.CacheMetadata(metadata);
}
```

 **CacheMetadata** is **invoked at design time**, this means we will not be able to access the value of InArguments or InOutArguments, since the **values only become available at runtime**. But **we can check if these are null or not**.

How to publish a Custom Activity

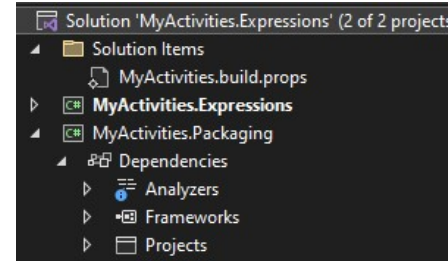


1. Create packaging project

1. Add a new C# Class Library to the existing solution.
This will be our **packaging project**.
2. Create {ProjectName}.build.props file containing Company, Copyright, VersionPrefix, Package Title and other related properties. Usually, this file is created in the solutions folder.
3. **Import {ProjectName}.build.props file** by using the following line in the .csproj file:
 - **<Import Project="../MyActivites.build.props" />**
4. Add .dll files. These files will be present in the generated .nupkg file.
5. Delete .cs file. Keep your **publishing project as simple as possible!**



Solution



MyActivities.build.props

```
<Project>
  <PropertyGroup>
    <Company>ACME</Company>
    <Copyright>© ACME</Copyright>
    <VersionPrefix>23.2.4</VersionPrefix>
    <PackageTags>MyActivities</PackageTags>
    <PackageTitle>MyActivities.Expressions.Activities</PackageTitle>
    <PackageId>MyActivities.Expressions.Activities</PackageId>

  </PropertyGroup>
</Project>
```

1. Create packaging project - example



MyActivities.Packaging.csproj

```
<Project Sdk="Microsoft.NET.Sdk">
  <Import Project="..\MyActivities.build.props" />
  <PropertyGroup>
    <TargetFrameworks>net6.0-windows</TargetFrameworks>
    <UseWPF>true</UseWPF>
    <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <OutputPath>..\Output</OutputPath>
    <TargetsForTfmSpecificBuildOutput>$(TargetsForTfmSpecificBuildOutput);AddDlls</TargetsForTfmSpecificBuildOutput>
  </PropertyGroup>
  <Target Name="RemoveMetaDll" AfterTargets="BuiltProjectOutputGroup">
    <ItemGroup>
      <BuiltProjectOutputGroupOutput Remove="@{(BuiltProjectOutputGroupOutput)" />
    </ItemGroup>
  </Target>
  <Target Name="AddDlls">
    <ItemGroup>
      <BuildOutputInPackage Include="$(OutputPath)MyActivities.Expressions.dll" />
    </ItemGroup>
  </Target>
  <ItemGroup>
    <ProjectReference Include="..\MyActivities.Expressions\MyActivities.Expressions.csproj" PrivateAssets="All" />
  </ItemGroup>
</Project>
```

Note: When using activity creator, all three dlls must be imported. Example:

```
<Target Name="AddDlls">
  <ItemGroup>
    <BuildOutputInPackage Include="$(OutputPath)MyActivities.Excel.dll" />
    <BuildOutputInPackage Include="$(OutputPath)MyActivities.Excel.Activities.dll" />
    <BuildOutputInPackage Include="$(OutputPath)MyActivities.Excel.Activities.Design.dll" />
  </ItemGroup>
</Target>
```

2. Edit .csproj file – part 1



1. Set **GeneratePackageOnBuild** to **True**

2. Update other .csproj properties, like:

- TargetFrameworks
- UseWPF
- etc.

```
<Project Sdk="Microsoft.NET.Sdk">
  <Import Project="..\MyActivities.build.props" />
  <PropertyGroup>
    <TargetFrameworks>net6.0-windows</TargetFrameworks>
    <UseWPF>>true</UseWPF>
    <AutoGenerateBindingRedirects>>true</AutoGenerateBindingRedirects>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <OutputPath>..\Output</OutputPath>
    <TargetsForTfmSpecificBuildOutput>$(TargetsForTfmSpecificBuildOutput);AddDlls</TargetsForTfmSpecificBuildOutput>
  </PropertyGroup>

  <Target Name="RemoveMetaDll" AfterTargets="BuiltProjectOutputGroup">
    <ItemGroup>
      <BuiltProjectOutputGroupOutput Remove="@ (BuiltProjectOutputGroupOutput)" />
    </ItemGroup>
  </Target>
</Project>
```

3. Edit .csproj file – part 2

1. Add dlls

```
<TargetsForTfmSpecificBuildOutput>$(TargetsForTfmSpecificBuildOutput);AddDlls</TargetsForTfmSpecificBuildOutput>
```

```
<Target Name="AddDlls">
  <ItemGroup>
    <BuildOutputInPackage Include="$(OutputPath)MyActivities.Excel.dll" />
    <BuildOutputInPackage Include="$(OutputPath)MyActivities.Excel.Activities.dll" />
    <BuildOutputInPackage Include="$(OutputPath)MyActivities.Excel.Activities.Design.dll" />
  </ItemGroup>
</Target>
```

2. Add dependencies in .csproj by using ItemGroup

```
<ItemGroup>
  <PackageReference Include="UiPath.Workflow" Version="6.0.0-20220401-03" PrivateAssets="All" />
  <PackageReference Include="System.Activities.Core.Presentation" Version="6.0.0-20220318.2" PrivateAssets="All" />
</ItemGroup>
```

2. Add a **Project Reference** to the other solution projects so far, so that they will be rebuilt every time you build your Packaging project:

```
<ItemGroup>
  <ProjectReference Include="..\MyActivities.Expressions\MyActivities.Expressions.csproj" PrivateAssets="All" />
</ItemGroup>
```



Project References and references to packages used internally by UiPath should be marked as private, lest your project fails to compile: **PrivateAssets="All"**

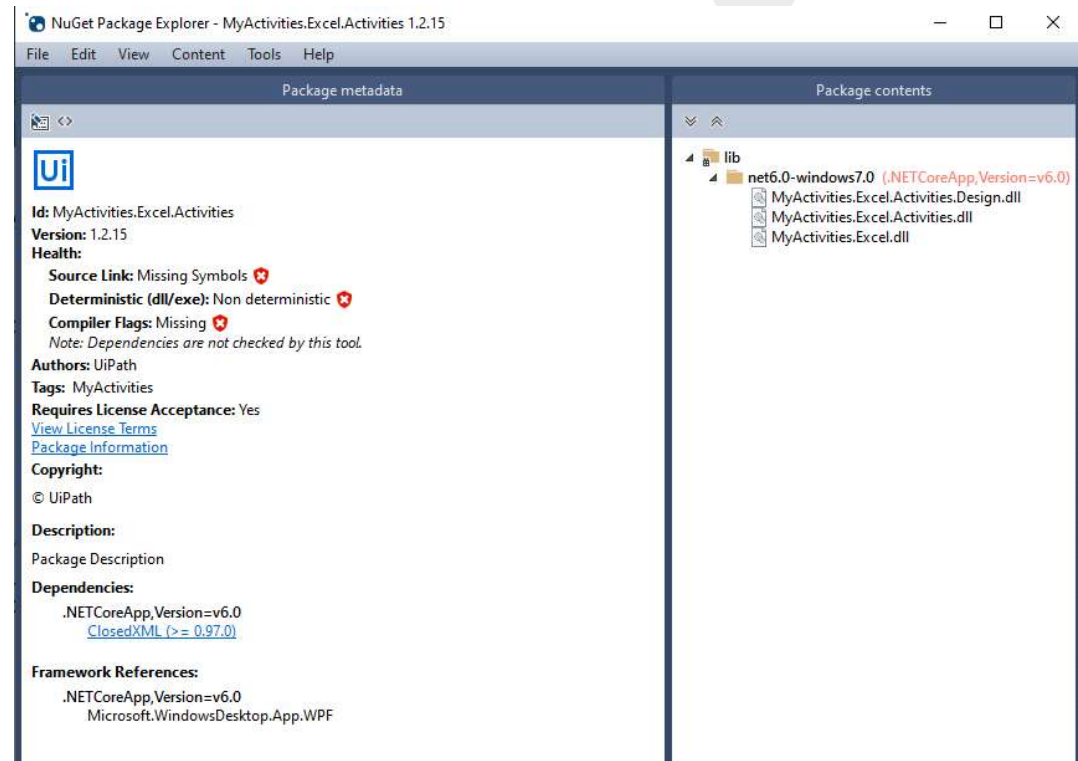
4. Build the project and open package

BUILD

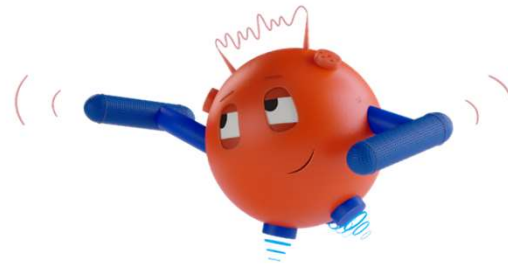
- Right click on packaging project. Click build.
- A .nupkg file will be created in OutputPath.

OPEN

- Install Nuget Package Explorer.
- Right-click on .nupkg file. Open With -> Nuget Package Explorer.



How to debug a Custom Activity?



1. Debug Custom Activity - Prerequisites

- ✓ Build your solution in Visual Studio in order to be able to generate your .nupkg.
- ✓ Create your .nupkg and install it in UiPath Studio.
- ✓ Keep both Visual Studio and UiPath Studio opened because we'll use them.

2. Debug Custom Activity



1. Create a workflow in UiPath Studio
2. Insert your custom activity
3. Add a breakpoint in UiPath Studio and run your workflow in Debug Mode.
4. Add some breakpoints in Visual Studio.
5. Attach to process in Visual Studio (Debug->Attach To Process -> Select "UiPath.Executor.exe" process from Available Processes).
*Only first time you must do these steps. After that, you can directly click on "Reattach to process" (Shift+ Alt + P). Visual Studio stores your selected process.
6. Click Continue in UiPath Studio and your execution continues.

How to create a Custom Activity using the Activity Creator





Prerequisites:

- ✓ Visual Studio Community/Professional/Enterprise 2022 with the .NET desktop development workload installed.
- ✓ UiPath's Marketplace Feed (<https://gallery.uipath.com/api/v3/index.json>) as a package source in Visual Studio. If this feed is not available in Visual Studio during development, the activities will not build successfully.
- ✓ UiPath Activity Creator Extension installed in Visual Studio
- ✓ NET 6
- ✓ UiPath Studio >=22.10.x



Version 4.0 of the Activity Creator only works with Visual Studio 2022. Likewise, the activities produced target .NET 6 Windows projects. To create activities compatible with older versions of Visual Studio (>= 2019.) or .NET, please use version 3 of the extension.



1. Create project in Visual Studio

Open Visual Studio >> Create New Project >> UiPath Standard Activity Project.

project name convention: <Your company's name>.<Your product's name>

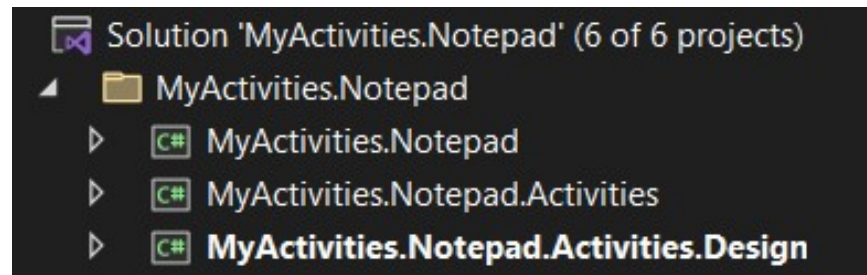
The screenshot shows the "Configure your new project" dialog box in Visual Studio. The title bar says "Configure your new project". Below it, the project type is "UiPath Standard Activity Project". The "Project name" field contains "MyActivities.Excel". The "Location" field shows "C:\CustomActivities\" with a dropdown arrow and a browse button "...". The "Solution name" field, which has an information icon, also contains "MyActivities.Excel". There is a checkbox labeled "Place solution and project in the same directory" which is currently unchecked. The "Framework" dropdown menu is set to ".NET Framework 4.6.1".

If the Framework cannot be changed when we create the project, this can be changed by editing the metadata for every project (edit .csproj file).

```
<TargetFrameworks>net6.0-windows</TargetFrameworks>
```

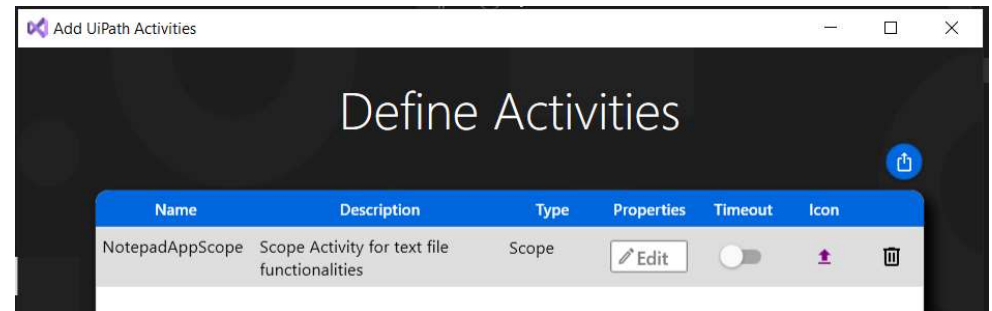
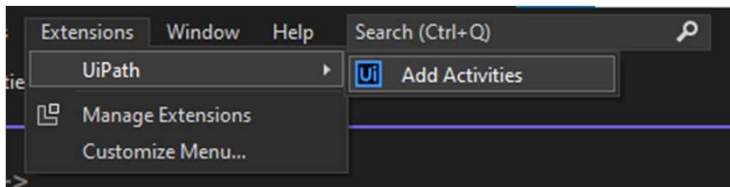
2. Understanding folder structure – main folder

- **Auxiliary project** (MyActivities.Notepad) contains all the custom classes which can be used in subsequent projects. Examples: custom datatypes, interfaces, anumes, auxiliary classes, etc. **This is optional!**
- **Activity project** (MyActivities.Notepad.Activities) contains the logic of our activities. Each activity has a corresponding file in activity project. In this file we can customize the activity properties (Category, Name, Description, Direction, Type, Required)
- **Designer project** contains all the UI elements for your activities (Custom controls, themes, converters and the activity UIs).
 - Each activity can have a corresponding Xaml and Xaml.cs in designer project.
 - The Design of our activities is built using WPF (**Windows Presentation Foundation** Controls)



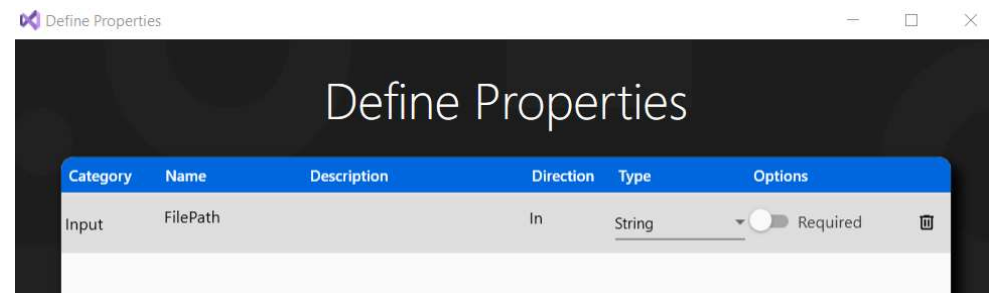
2. Create a scope activity

Open “Add Activities” Wizard:



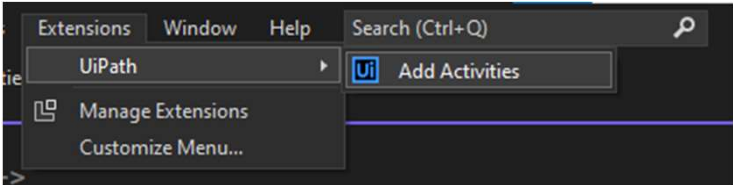
Create the following activity:

- **Name:** NotepadAppScope
- **Type:** Scope
- **Input arguments:**
 - FilePath - InArgument<string>



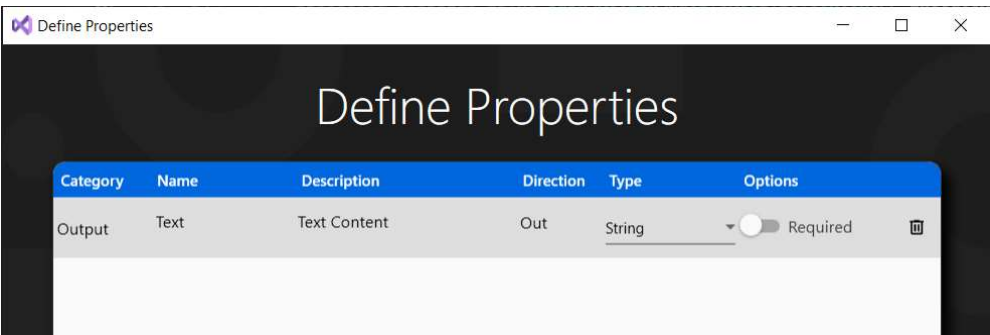
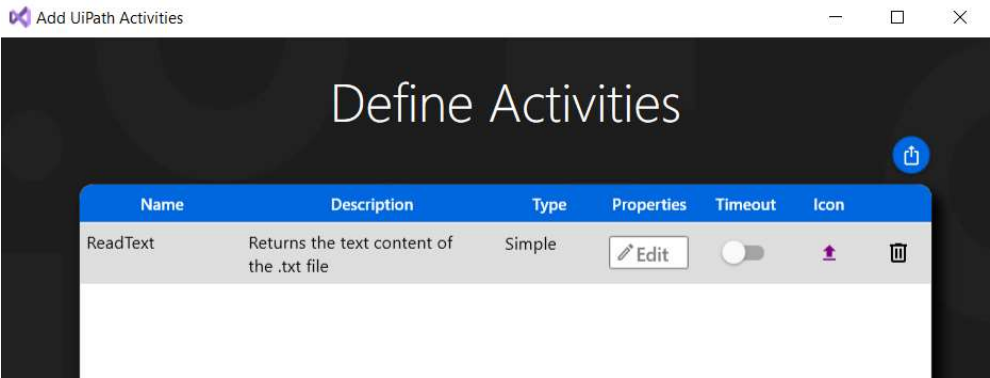
3. Create a simple activity

Open “Add Activities” Wizard:



Create the following activity:

- **Name:** ReadText
- **Type:** Simple
- **Output arguments:**
 - Text - OutArgument<string>



4. Understanding Activity Structure

- This scope activity implements **ContinuableAsyncNativeActivity** that inherits the **NativeActivity** class. **ContinuableAsyncNativeActivity** is used to improve the performance of using the WF specific activity classes in our project.
- By using **ExecuteAsync** method, the parent activity starts the execution of child activity (calling **ScheduleAction**) and pass the value for the DelegateArgument.
- All other components (CacheMetadata, Events, Helpers) are similar to those used in creating activities using code.

```
protected override async Task<Action<NativeActivityContext>> ExecuteAsync(
    NativeActivityContext context, CancellationToken cancellationToken)
{
    // Inputs
    var filepath = FilePath.Get(context);
    _objectContainer.Add(filepath);

    return (ctx) => {
        // Schedule child activities
        if (Body != null)
            ctx.ScheduleAction<IObjectContainer>(Body, _objectContainer,
                OnCompleted, OnFaulted);

        // Outputs
    };
}
```



5. Understanding Activity Design

- The **ActivityDesigner** control is the container of the UI elements we want to encapsulate.
- WorkflowItemPresenter** is used to Bind the Body Property to an empty activity container in the page. The binding is done by using the databinding option offered by WPF.



At runtime, the DesignerMetadata class will be used to load the designer metadata.

```
sap:ActivityDesigner x:Class="MyActivities.Notepad.Activities.Design.Designers.NotepadAppScopeDesigner"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:s="clr-namespace:System;assembly=mscorlib"
xmlns:sa="http://schemas.microsoft.com/netfx/2009/xaml/activities"
xmlns:p="clr-namespace:MyActivities.Notepad.Activities.Design.Properties"
xmlns:sharedres="clr-namespace:UiPath.Shared.Localization"
xmlns:sap="clr-namespace:System.Activities.Presentation;assembly=System.Activities.Presentation"
xmlns:sapv="clr-namespace:System.Activities.Presentation.View;assembly=System.Activities.Presentation"
xmlns:sapc="clr-namespace:System.Activities.Presentation.Converters;assembly=System.Activities.Presentation"
xmlns:converters="clr-namespace:UiPath.Shared.Activities.Design.Converters"
xmlns:uip="clr-namespace:UiPath.Shared.Activities.Design.Controls" >

<sap:ActivityDesigner.Resources>
  <ResourceDictionary>
    <sapc:ArgumentToExpressionConverter x:Key="ArgumentToExpressionConverter" />
    <converters:ActivityIconConverter x:Key="ActivityIconConverter" />
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="..\Themes\Generic.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</sap:ActivityDesigner.Resources>

<sap:ActivityDesigner.Icon>
  <DrawingBrush Stretch="Uniform" Drawing="{Binding Path=ModelItem, Converter={StaticResource ActivityIconConverter}, ConverterParameter=pack://>
</sap:ActivityDesigner.Icon>

<ui:ActivityDecoratorControl Style="{StaticResource ActivityDecoratorStyle}">
  <DockPanel LastChildFill="True">

    <StackPanel DockPanel.Dock="Top">
      <uip:FilePathControl ModelItem="{Binding ModelItem}" HintText="{x:Static p:Resources.FilePathHintText}" DockPanel.Dock="Top"
        Expression="{Binding Path=ModelItem.FilePath, Converter={StaticResource ArgumentToExpressionConverter}, ConverterParameter=p:Resources.FilePath} Title="{x:Static p:Resources.NotepadPathDialogTitle}"
        MaxWidth="{Binding ElementName=ItemsPresenter, Path=ActualWidth}"
        AutomationProperties.Name="{x:Static p:Resources.FilePathHintText}" />
    </StackPanel>

    <sap:WorkflowItemPresenter x:Uid="sad:WorkflowItemPresenter_1"
      AutomationProperties.AutomationId="Activity"
      DockPanel.Dock="Bottom"
      MinWidth="400"
      Margin="0,10,0,0"
      Item="{Binding Path=ModelItem.Body.Handler, Mode=TwoWay}"
      AllowedItemType="{x:Type sa:Activity}"
      HintText="{x:Static p:Resources.DropActivityHere}" />
  </StackPanel>
</DockPanel>
</ui:ActivityDecoratorControl>
</sap:ActivityDesigner>
```

7. Build the solution



We need to create a .dll for each project from the first folder as follows:

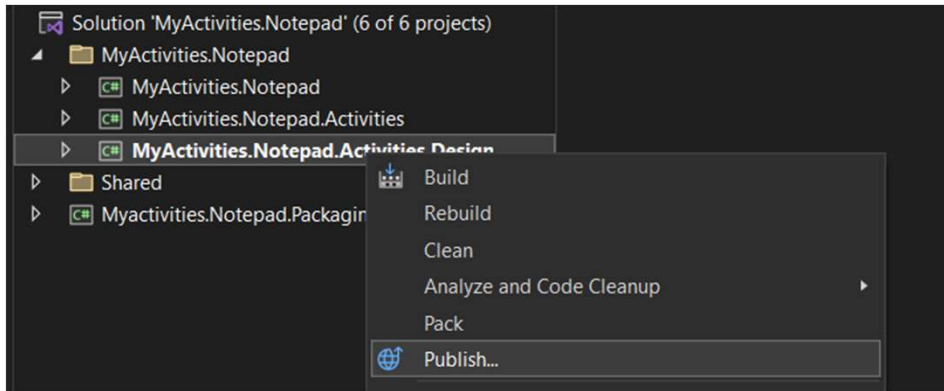
Project Name	DLL
MyActivities.Notepad	MyActivities.Notepad.dll
MyActivities.Notepad.Activities	MyActivities.Notepad.Activities.dll
MyActivities.Notepad.Activities.Design	MyActivities.Notepad.Activities.Design.dll



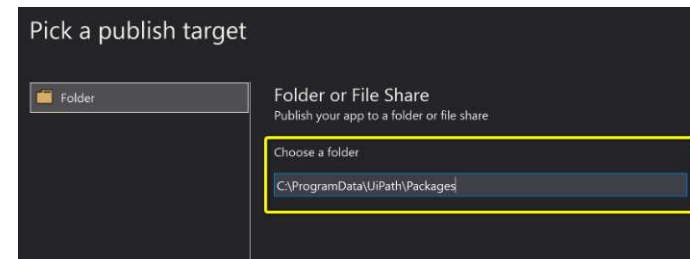
The .dlls will be created in `OutputPath` specified in `PropertyGroup` (in .csproj file).

7. Publishing the solution

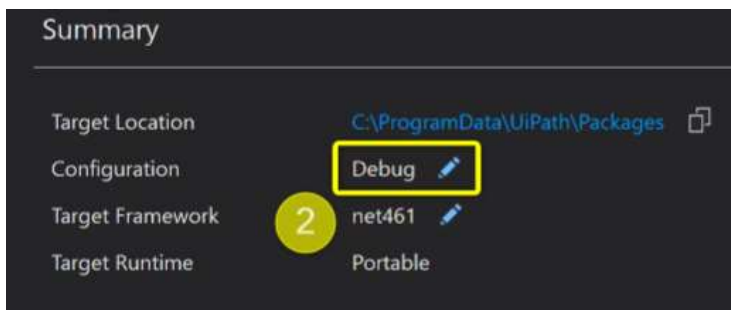
1. Right-click your Design project and select Publish



2. Create Profile and pick a publish target



3. Configure the rest of the properties and Publish

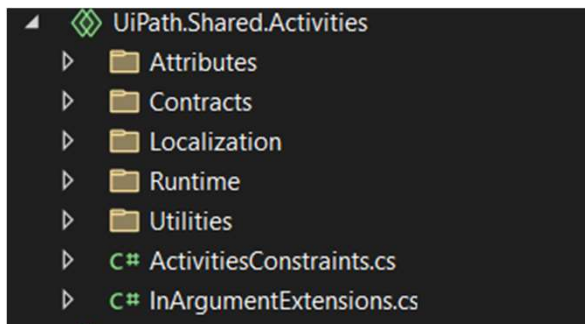


Detailed instructions can be found here:
<https://docs.uipath.com/developer/docs/building-activity-packages>

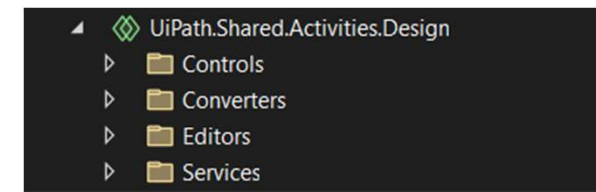
8. Understanding folder structure – shared folder

Contains handy classes and controls that can be used to speed up development.

Activities shared project contains classes/interfaces which can be used in Activities Project from main folder. Usually, the activity class inherits one of classes from Runtime folder, which are classes derived from one of the WF specific activity classes.



Design shared project contains classes/interfaces which can be used in Design Project from main folder.



- ⚠ The Shared Project does not get compiled on its own. When referenced by another project, the code is effectively compiled as *part* of that project.
- ⚠ Shared projects cannot reference any other project type (including other Shared Projects)

Useful tips – how to add drop-down list property

```
5 references
public enum GT
{
    OffForRowsAndColumns,
    OnForRowsAndColumns,
    OnForRowsOnly,
    OnForColumnOnly
}

[RequiredArgument]
[LocalizedDisplayName(nameof(Resources.DesignPivotTable_GrandTotals_DisplayName))]
[LocalizedDescription(nameof(Resources.DesignPivotTable_GrandTotals_Description))]
[LocalizedCategory(nameof(Resources.Input_Category))]
1 reference
public GT GrandTotals { get; set; }
```

Input	
GrandTotals	OffForRowsAndColumns ▾
PivotTableName	OffForRowsAndColumns
SheetName	OnForRowsAndColumns
ShowSubtotals	OnForRowsOnly
	OnForColumnOnly
Misc	
Private	<input type="checkbox"/>

```
2 references
protected override async Task<Action<AsyncCodeActivityContext>> ExecuteAsync(AsyncCodeActivityContext context)
{
    // Inputs
    var grandTotals = GrandTotals;
```

Useful tips – how to add checkbox property

```
[Category("Input")]  
[DisplayName("IncludeRangeLimits")]  
[RequiredArgument]  
1 reference  
public bool IncludeRangeLimits { get; set; }
```

Input	
DateToCheck	mydate ...
IncludeRangeLimits	<input checked="" type="checkbox"/>
Left	left ...
Right	mydate ...

```
0 references  
protected override void Execute(CodeActivityContext context)  
{  
    var includerangelimits = IncludeRangeLimits;
```

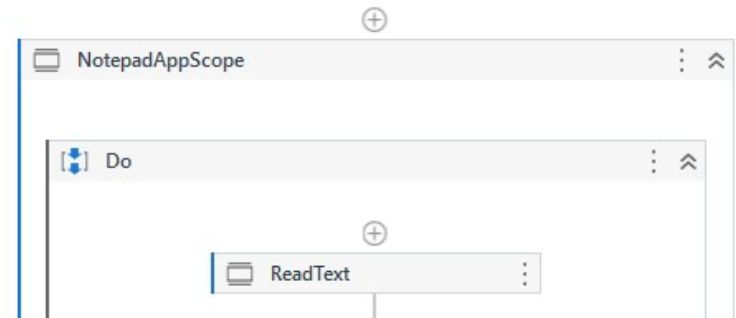
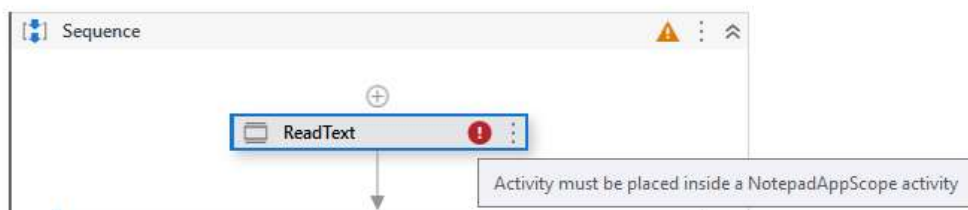

Enforce Child Activity is inside parent scope

- The **ActivityConstraints** static class in **UiPath.Shared.Activities** project can provide a constraint that loops through the parents of a specific activity and checks if at least one of them is of a specific type.
- We can add this Constraint in our activity's **constructor**:

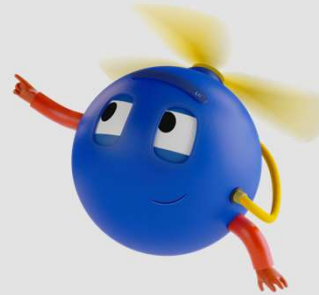
```
#region Constructors

public ReadText()
{
    base.Constraints.Add(ActivityConstraints.HasParentType<ReadText, NotepadAppScope>
        (Resources.NotepadAppScope_ChildActivityConstraint));
}

#endregion
```



Targeting multiple .NET Runtimes





Prerequisites:

- ✓ Visual Studio Community/Professional/Enterprise 2022 with the .NET desktop development workload installed.
- ✓ UiPath's Marketplace Feed (<https://gallery.uipath.com/api/v3/index.json>) as a package source in Visual Studio. If this feed is not available in Visual Studio during development, the activities ***will not build successfully***.
- ✓ UiPath Activity Creator Extension installed in Visual Studio.
- ✓ NET 6
- ✓ UiPath Studio >=22.10.x

Background Information



- Ideally, when we build an activity package, we want it to be usable across all Compatibility Types in UiPath Studio.
 - To achieve this, we need to **use multiple Target Frameworks**.
- The **dependencies we use might not be compatible with all the Target Frameworks** selected.
 - We need to always check the compatibility of our dependencies and even **use different versions or different packages altogether** to achieve the desired level of compatibility.
- *****.***.Activities.Design** packages use WPF, this means that they can only be used on Windows machines and they're **incompatible with the “.net6.0” target**.
 - When targeting Cross-Platform projects, do not include the *****.***.Activities.Design** dll.

Configuring .csproj



Auxiliary project (MyActivities.Notepad) & Activities project (MyActivities.Notepad.Activities) :

1. Add all the target frameworks you want to support in the .csproj file, separated by “;”

```
<TargetFrameworks>net6.0;net461</TargetFrameworks>
```

2. If using both .NET Framework 4.6.1 and .NET 6.0, you will need to use **different WF references for each target framework**. We can do this using a **Condition attribute**. Replace the Existing References with the following:

```
<ItemGroup Condition=" '$(TargetFramework)' == 'net461' ">
  <Reference Include="System.Activities.Presentation" />
  <Reference Include="System.Activities" />
</ItemGroup>
<ItemGroup Condition=" '$(TargetFramework)' == 'net6.0' ">
  <PackageReference Include="System.Activities.Metadata" Version="6.0.0-*" PrivateAssets="All" />
  <PackageReference Include="UiPath.Workflow.Runtime" Version="6.0.0-20220909-01" PrivateAssets="All" />
</ItemGroup>
```

Configuring .csproj



Designer project (MyActivities.Notepad.Activities.Design):

1. Add all the target frameworks you want to support in the .csproj file, separated by “;”. Designer Projects use WPF, so we cannot use “net6.0”, we need to **use the platform specific “net6.0-windows” target**:

```
<TargetFrameworks>net461;net6.0-windows</TargetFrameworks>
```

2. If using both .NET Framework 4.6.1 and .NET 6.0 Windows, you will need to use **different WF references for each target framework**. We can do this using a **Condition attribute**. Replace the Existing References with the following:

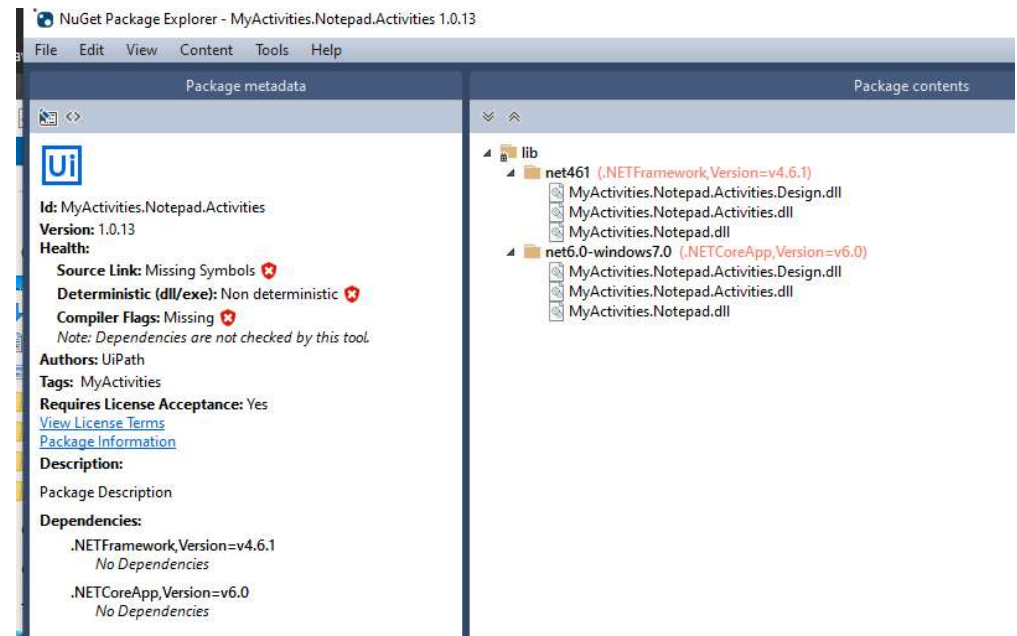
```
<ItemGroup Condition=" '$(TargetFramework)' == 'net461' ">
  <Reference Include="System.Activities" />
  <Reference Include="System.Activities.Presentation" />
  <Reference Include="System.Activities.Core.Presentation" />
</ItemGroup>

<ItemGroup Condition=" '$(TargetFramework)' == 'net6.0-windows' ">
  <PackageReference Include="UiPath.Workflow" Version="6.0.0-20220909-01" PrivateAssets="All" />
  <PackageReference Include="System.Activities.Core.Presentation" Version="6.0.0-*" PrivateAssets="All" />
</ItemGroup>
```

Understanding package contents



- ✓ **One** .NET folder/ .NET Framework folder will be created for **each target framework**.
- ✓ When the .nupkg is added to the UiPath Studio project, the required dlls (from net461 folder or from net6.0-windows7.0 folder) will be **imported**, based on the **UiPath Studio project compatibility**.
- ✓ Dependencies must be runtime specific.



References & useful links



- Targeting .NET Frameworks: <https://learn.microsoft.com/en-us/dotnet/standard/frameworks>
- **Windows Workflow Foundation**
 - Documentation: <https://learn.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/>
 - Samples: <https://learn.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/samples/>
- Windows Workflow Foundation port to .NET 6 done by UiPath: <https://github.com/UiPath/CoreWF>
- !!! Using the UiPath Activity Creator: <https://docs.uipath.com/developer/docs/using-activity-creator>
- !!! Migrating Activities to .NET 6: <https://docs.uipath.com/developer/docs/migrating-activities-to-net#step-1-migrate-the-project-to-the-new-sdk-style-format-and-add-the-net60-windows-target>
- Understanding the Project File: <https://learn.microsoft.com/en-us/aspnet/web-forms/overview/deployment/web-deployment-in-the-enterprise/understanding-the-project-file>
- Exposing data with **CacheMetadata**: <https://learn.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/exposing-data-with-cachemetadata>
- Windows Presentation Foundation: <https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-6.0>
- Managing Activities Packages: <https://docs.uipath.com/studio/docs/managing-activities-packages>