AUTODESK UNIVERSITY

AULON64368

Getting Started with Dynamo Custom Node Development Using C# and Zero Touch Import

Thomas Mahon, Bimorph & Alex Faur, Fabrit Global

Class Description

Have you ever thought about coding but never tried because you think you can't do it? Are you an experienced Dynamo user but find yourself at the limits of visual programming? Or maybe you're an IronPython wiz who feels strong-typed languages are out-of-reach.

If the above resonates then you should attend this lab, because in 75 minutes these barriers will be broken. You will learn how to develop Zero Touch nodes for Dynamo in C# and set yourself up on a path to develop relevant and highly-prized skills for 21 Century construction professionals. Hosted by Thomas Mahon, a computational design/BIM expert and creator of BimorphNodes, one of Dynamo's most popular packages - you will learn from one of the most experienced developers in the Dynamo community. By the end of the lab, delegates will be able to apply this knowledge to their existing workflows, and enter the top tier of developers capable of controlling Dynamo for Revit without limitation!

Why learn Zero Touch? There are numerous reasons to create Zero Touch nodes in C#:

- Fully integrated in an IDE (Visual Studio), providing IntelliSense and debugging which make coding easier than any other option
- Speed - Zero Touch nodes execute rapidly making them ideal for complex problems on large projects
- Full access to the Revit API, Dynamo API and ability to communicate with external applications
- Code is more secure and easier to protect if IP is a concern

The workshop is delivered through practical exercises covering:

- Zero Touch basics and Visual Studio environment configuration
- Key C# Concepts
- Creating namespaces, classes, methods and properties to understand how Dynamo consumes Zero Touch libraries
- Compiling Zero Touch projects and Custom Package creation

About the Speakers:

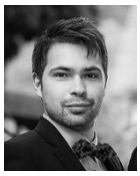


Thomas is the Founding Director of Bimorph and a computational design/BIM specialist. With a background in Architecture, he has worked with leading architectural firms including Foster + Partners and Rogers Stirk Harbour + Partners on a diverse range of high-profile projects.

He has continued to push the boundaries of digital technology, specializing in computational design/BIM and software engineering to pioneer new paradigms for building design and delivery. His achievements include developing ground-breaking workflow-automation tools for leading construction companies, using cutting-edge procedural modelling techniques to deliver geometrically complex buildings, and more recently, developing widely-used Dynamo Node Packages and Revit add-ins.

Thomas has grown Bimorph rapidly since launching in 2015, with projects ranging from Revit and Dynamo add-in development which significantly improve workflow efficiency (such as BimorphNodes), to solving highly complex building design problems for major construction companies spanning from Europe, to South East Asia.

bimorph
DIGITAL ENGINEERING



Alexandru Faur, Fabrit Global

Alex is a co-founder of Fabrit Global, with a background in Software Development, Software Architecture and Business Analysis. He has started his career as a software engineer for the Oil & Gas sector in Scotland and has successfully delivered software solutions for companies from USA, UK and Southeast Asia.

Working in a complex and challenging environment, he realized his ideas and practices are widely applicable and could be transferred to other sectors. With Fabrit Global, he thrives to help businesses improve processes and be ahead of their competition through technology and software innovation. Alex has been collaborating for several years with top BIM experts and digital consultants to become a specialist for the Construction industry, using Revit API and Dynamo custom nodes. He has been developing sophisticated workflows and design methodologies that provide his clients with a competitive advantage, helping cut expenses and offer more scalable, data-driven design solutions.

fabrit | global

ZERO TOUCH INTRODUCTION

Zero Touch development in Dynamo for Revit provides a simple workflow for writing custom nodes in C# using classes and importing library of classes into Dynamo without the need to configure the software or manually configure the node – it therefore requires little effort to begin using your program once its compiled, hence the term ‘Zero Touch’.

Writing custom nodes in C# has many advantages over Python; C is a strongly-typed language which improves type-safety and robustness of code. In addition, it can be written within powerful IDEs, such as Visual Studio, which aid the developer and enable the development of program that would be difficult to achieve using Python and Dynamo’s text editor. In addition, as programs written in C# have to be compiled before runtime, they execute quicker than Python, making it an ideal option for projects that demand a high degree of performance.

The workshop covers the following topics:

1. Visual Studio set-up
2. Environment configuration
3. Core C# Concepts
4. Writing custom nodes:
    - HelloWorld (intro)
    - Using the Revit API
    - Compiling and debugging

PREREQUISITES

Before beginning the lab, ensure the following software is installed:

1. Revit 2020
2. Dynamo 2.1
3. Visual Studio Community 2017 – Use the Visual Studio Installer and select/install the **.NET Desktop Development Workload** (https://visualstudio.microsoft.com/vs/community/)
4. Revit Lookup (https://github.com/jeremytammik/RevitLookup)

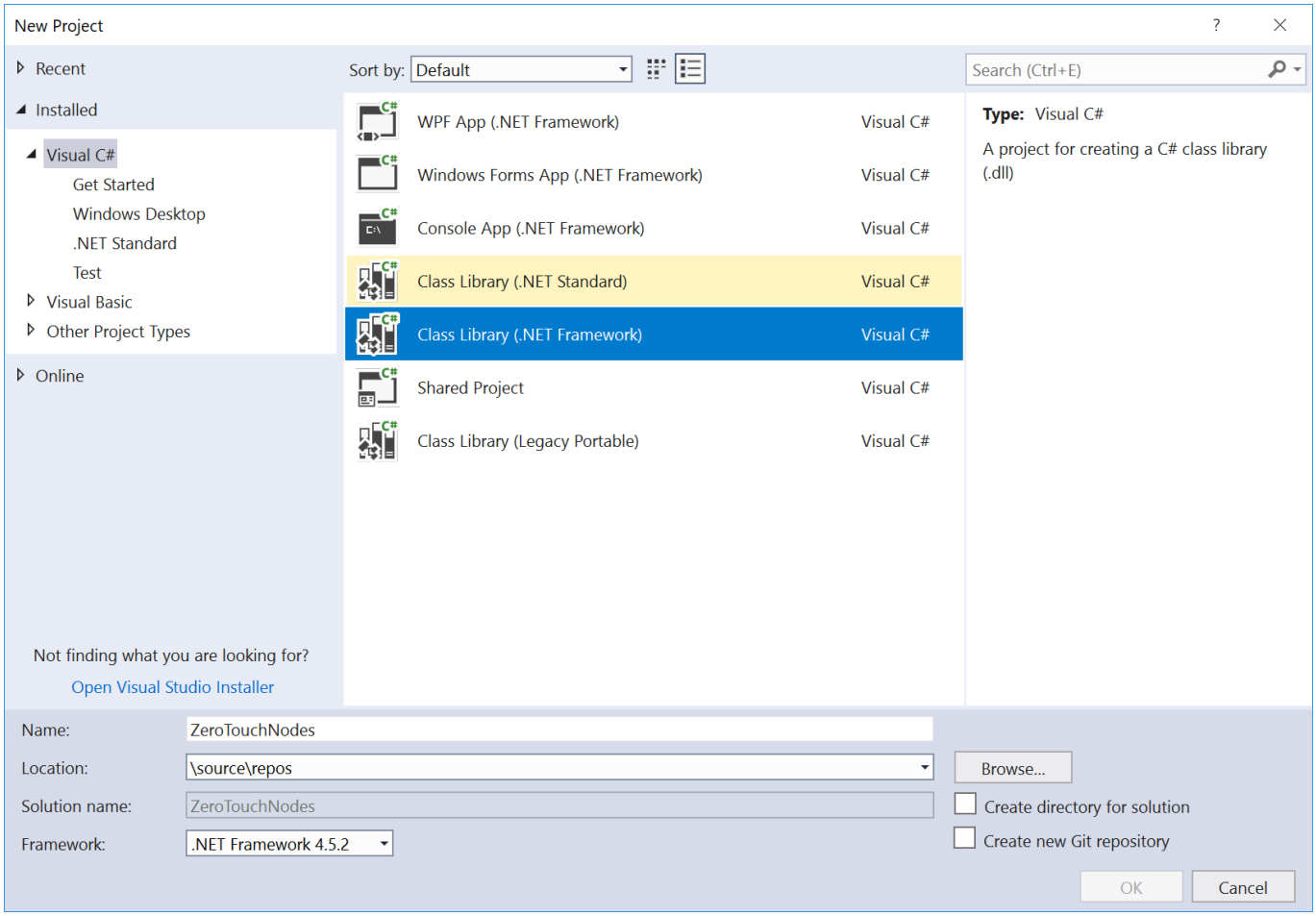
1. VISUAL STUDIO SETUP

Starting a Zero Touch Project in Visual Studio

The following steps demonstrate how easy it is to setup a new Class Library project in Visual Studio.

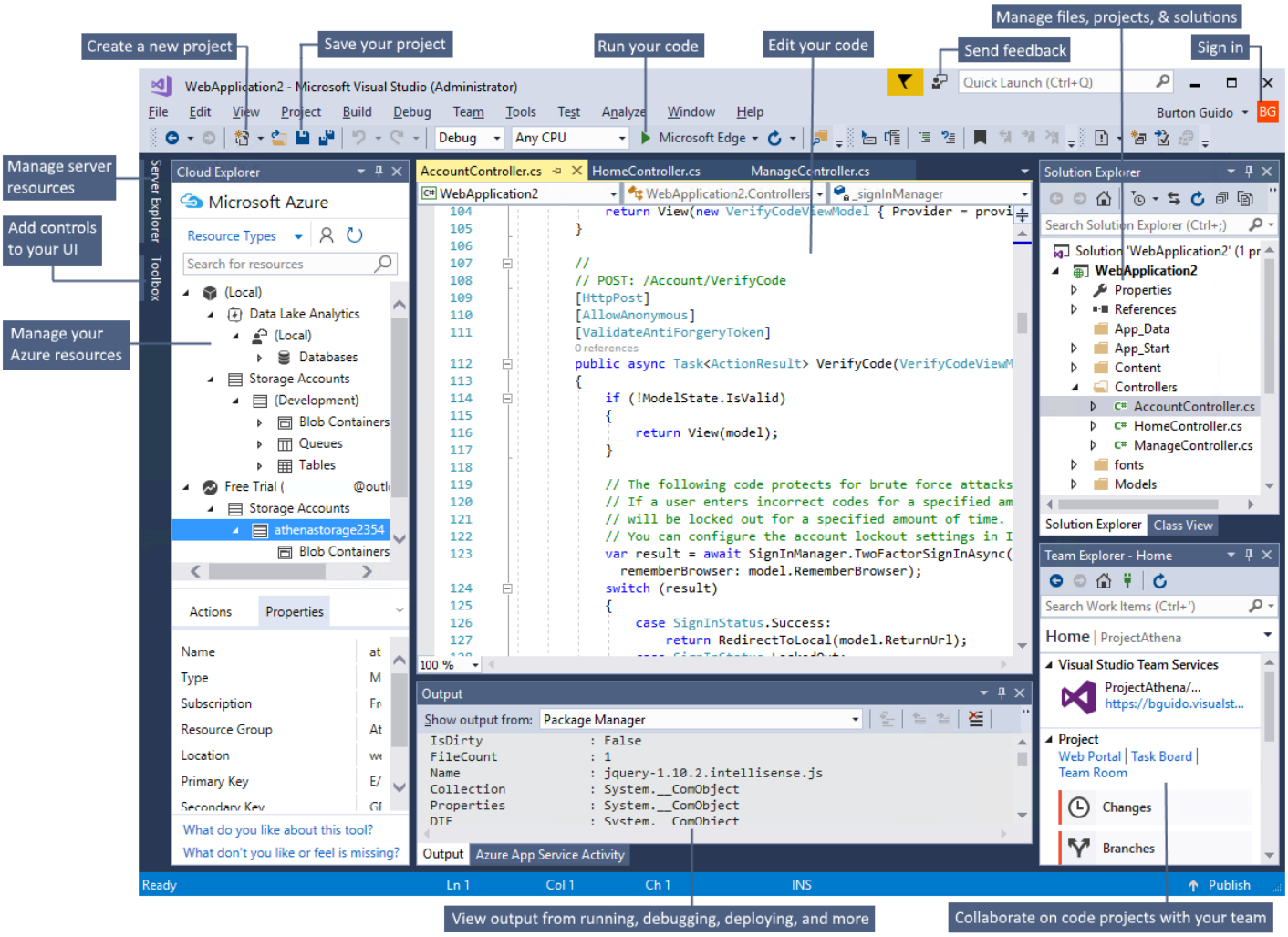
Step 1: Create a New Project

Open Visual Studio and create a new project by clicking File > New > Project. In the dialog window that opens, select Class Library and name the project ZeroTouchNodes and uncheck the nearby checkboxes. Click the OK button to create the new project:



Step 2: Tour of the IDE

Referenced from https://docs.microsoft.com/en-us/visualstudio/ide/visual-studio-ide?view=vs-2017
Now the project is setup, its worth taking a high-level visual overview of Visual Studio to get family with its appearance. The following image shows Visual Studio with an open project and several key tool windows you'll likely use:



Solution Explorer (top right) lets you view, navigate, and manage your code files. Solution Explorer can help organize your code by grouping the files into solutions and projects.

The editor window (center), where you'll likely spend a majority of your time, displays file contents. This is where you can edit code or design a user interface such as a window with buttons and text boxes.

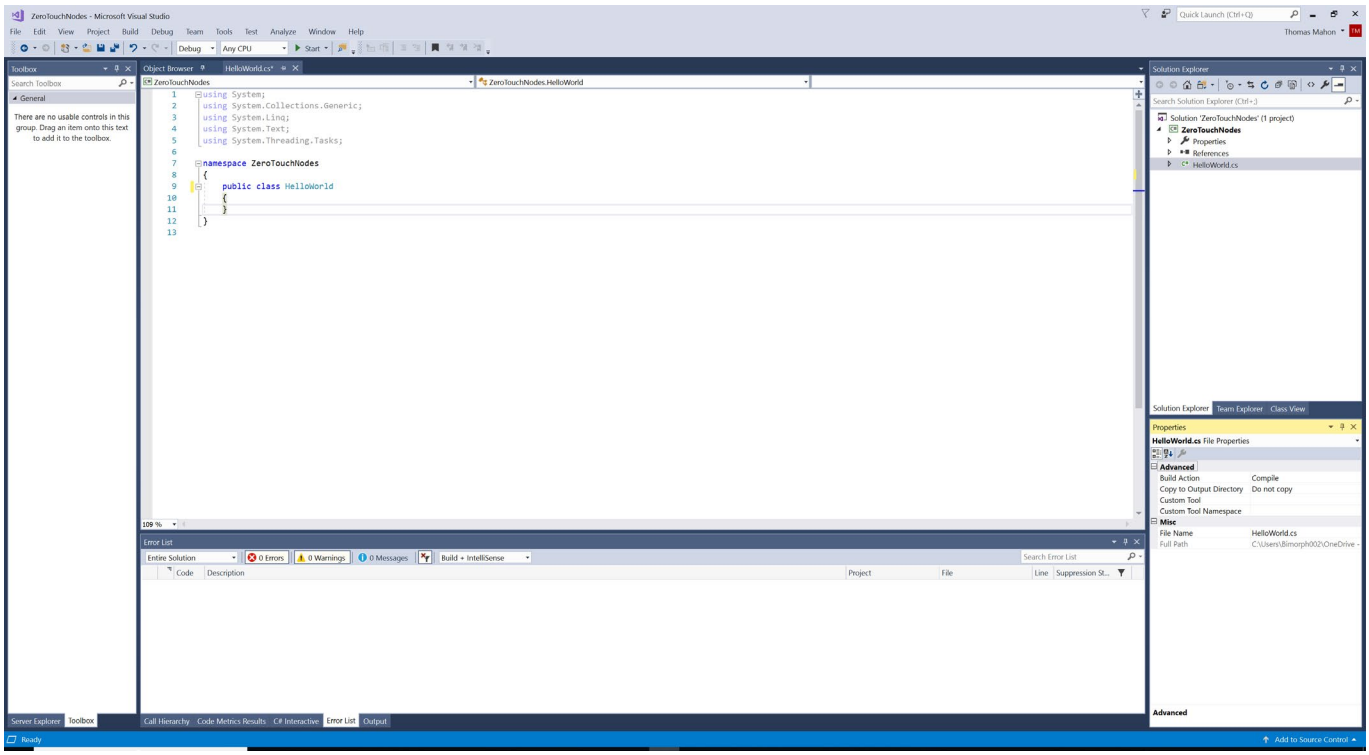
The Output window (bottom center) is where Visual Studio sends notifications such as debugging and error messages, compiler warnings, publishing status messages, and more. Each message source has its own tab.

Team Explorer (bottom right) lets you track work items and share code with others using version control technologies such as Git and Team Foundation Version Control (TFVC).

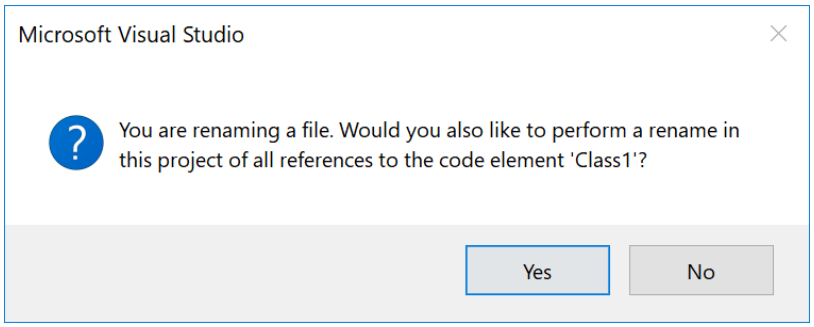
Step 3: Rename the Class

You have now created your first Visual Studio developer project. You should see a new class (by default, named Class1.cs) and a new namespace called ZeroTouchNodes. The next step is to rename the class from the default name to `HelloWorld` – this will be the class name of our new custom Zero Touch node.

To rename the class and ensure all references are updated, click on the Class1.cs file in the Solution Explorer panel, and below, in the Properties panel, enter HelloWorld into the File Name property. Ensure the class is marked public otherwise it won't be visible in the Dynamo when it is imported:



When you rename a class in this fashion, the following warning message appears. Click Yes to accept the change:

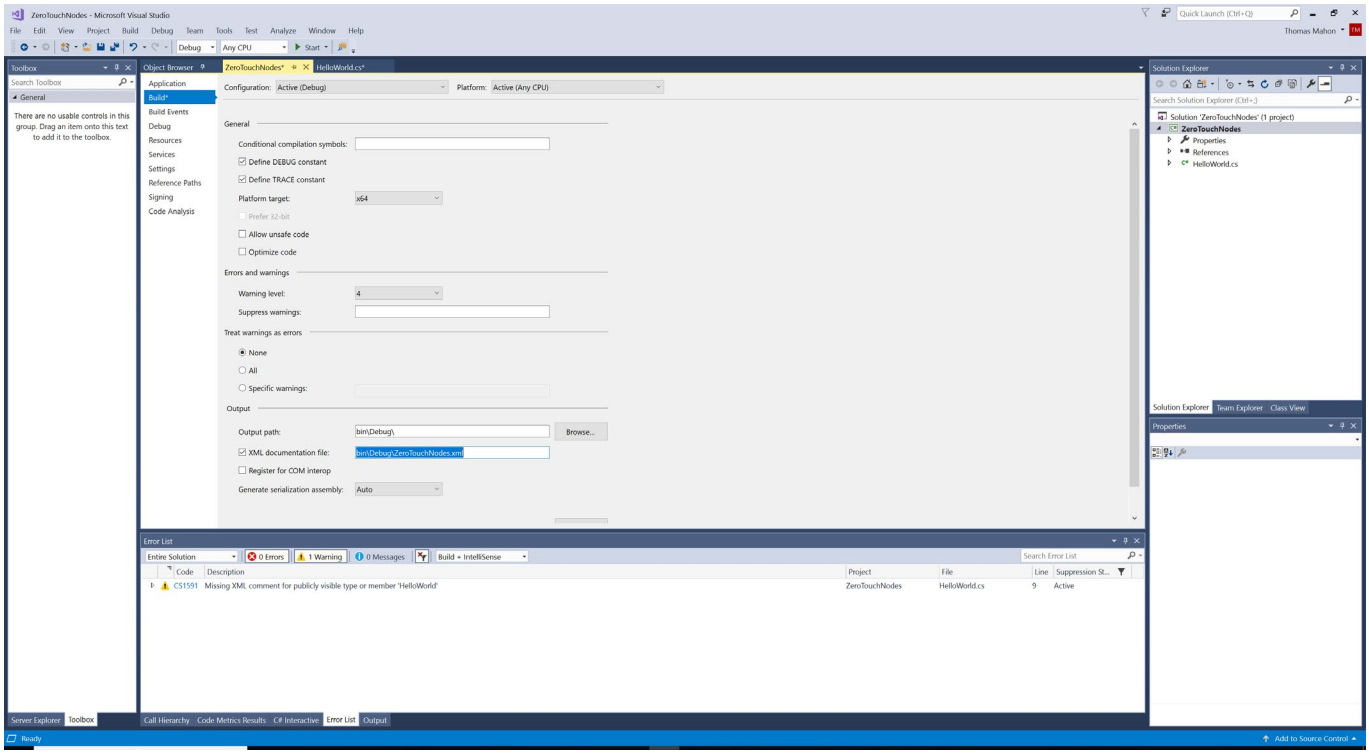


2. ENVIRONMENT CONFIGURATION

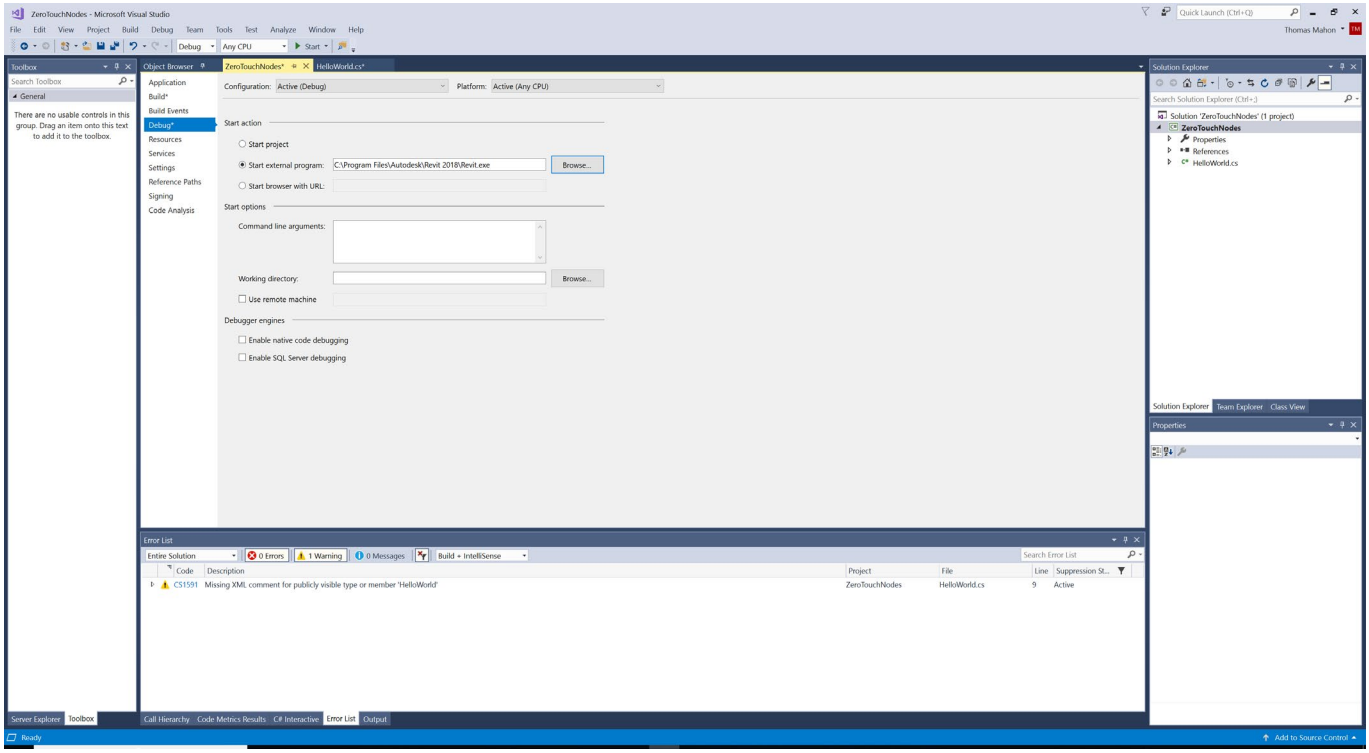
Step 1 Build Configuration

Once the project is set-up, it needs to be configured to automate actions that aid with the development of Zero Touch custom node packages. The first step is to associate an external application to our project for the purposes of debugging and testing. Since Dynamo is dependent on Revit to operate, the external application needs to be set to Revit.exe. To do this, click on the Project tab > ZeroTouchNodes Properties, from the menu. A new panel will open displaying the project properties.

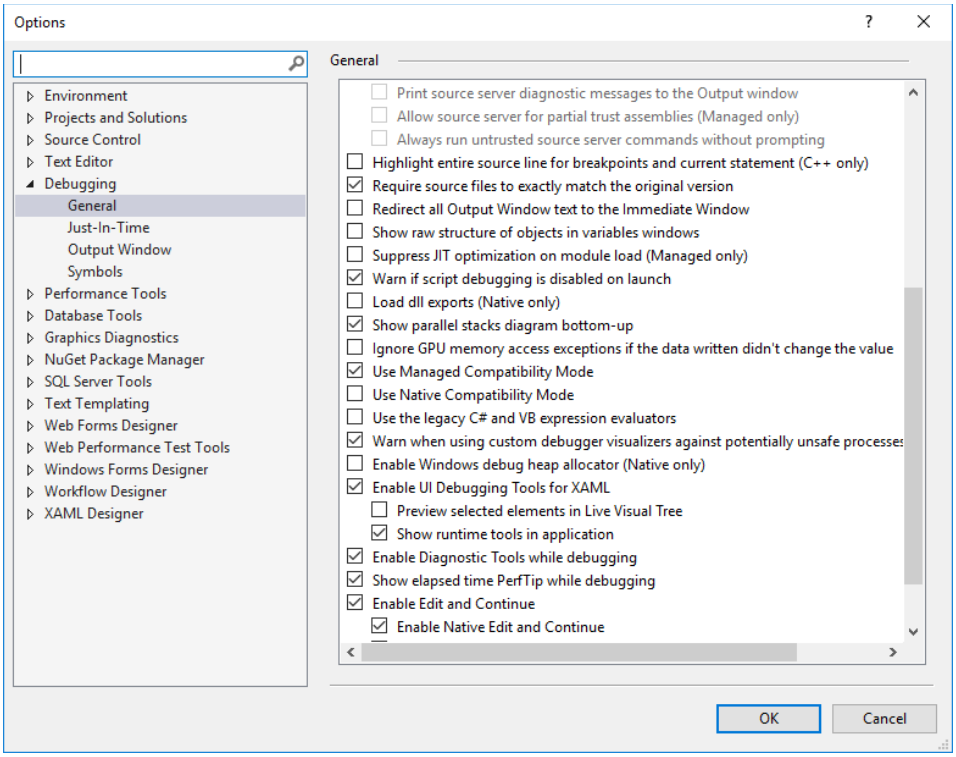
Click on the Build tab on the left-hand side and select x64 from the drop-down next to Platform Target. This step is required whenever referencing a library built on the x64 platform, such as the Revit API:



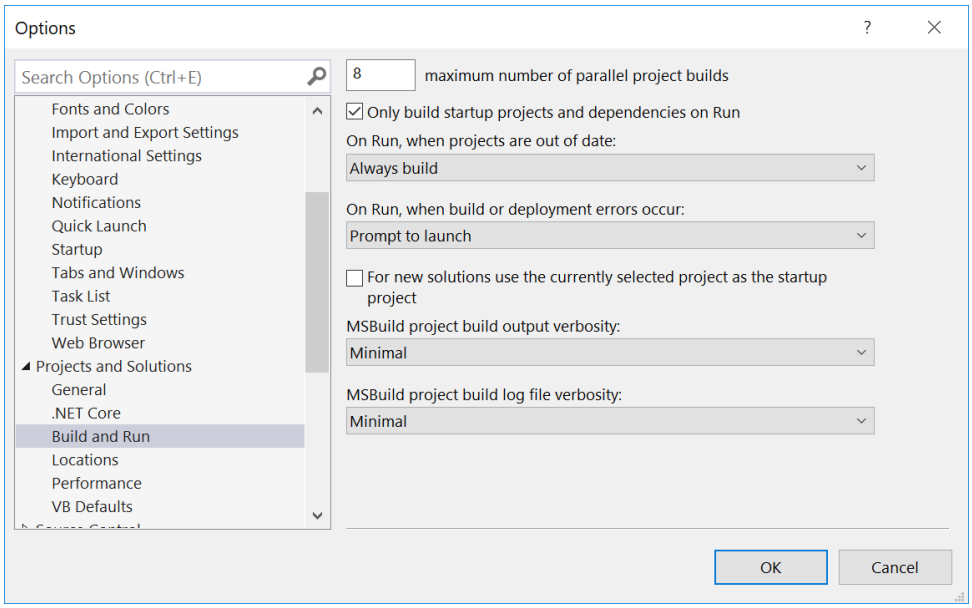
Next, click on the Debug tab and activate the ‘Start External Program’ radio button. Click on the square browse button next to the input field, and browse to your Revit root folder (typically this is C:\Program Files\Autodesk\Revit 2018) and locate/select the Revit.exe file. Click OK to select it and assign it to the external program setting:



To ensure Revit launches when we run the application, click on the Tools tab > Options. In the Options dialog, expand the Debugging tab and click on General. Find ‘Use Managed Combability Mode’ and ensure it is checked:

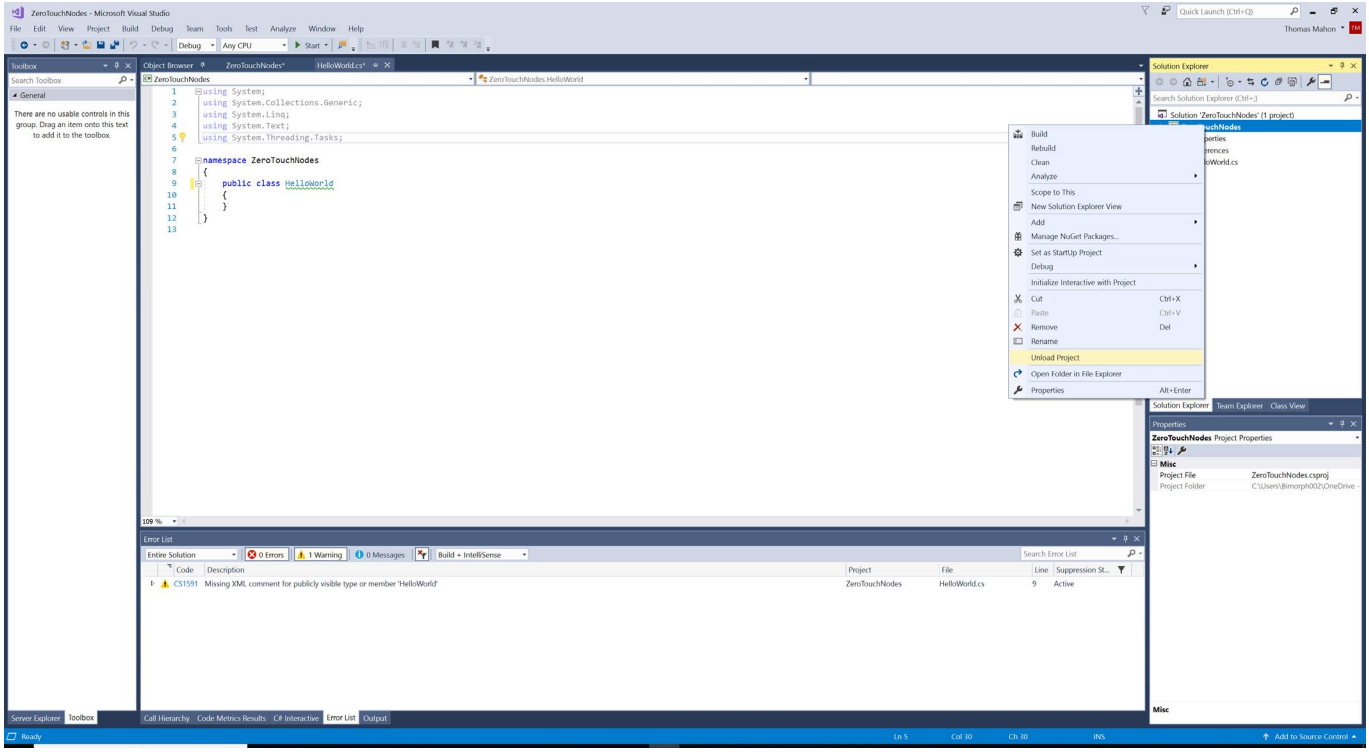


Click on Tools > Options > Project Solutions > Build and Run, ensure ‘Always build’ option is selected from the drop-down under ‘On Run, when projects are out of date’:



Step 2 Configure .csproj File

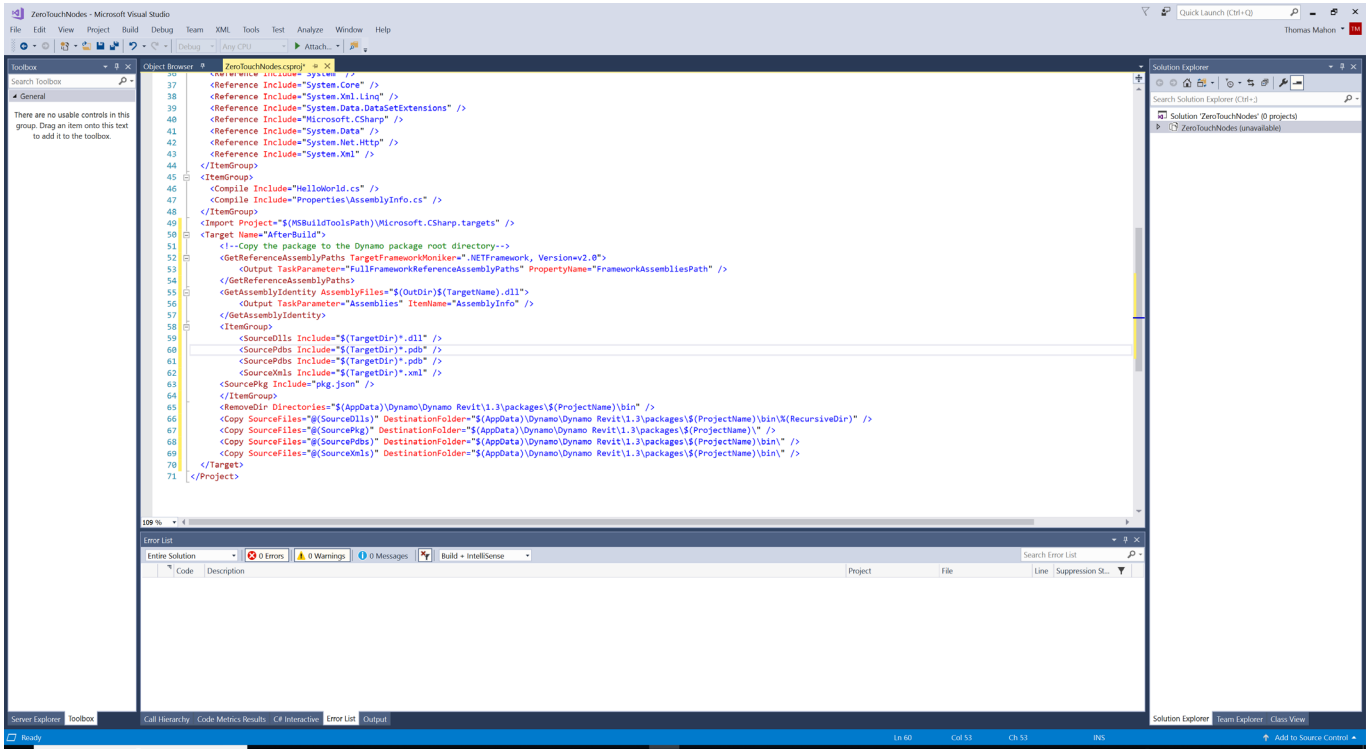
To complete the configuration the .csproj file requires some additional instructions to automate the creation of the folders and copying of the files required to package our Zero Touch nodes in Dynamo’s root package directory so they appear in the library. This avoids the need to manually copy over our new library to this location every time the project is compiled. First, right-click on the ZeroTouchNodes solution file in the Solution Explorer, and click Unload Project:



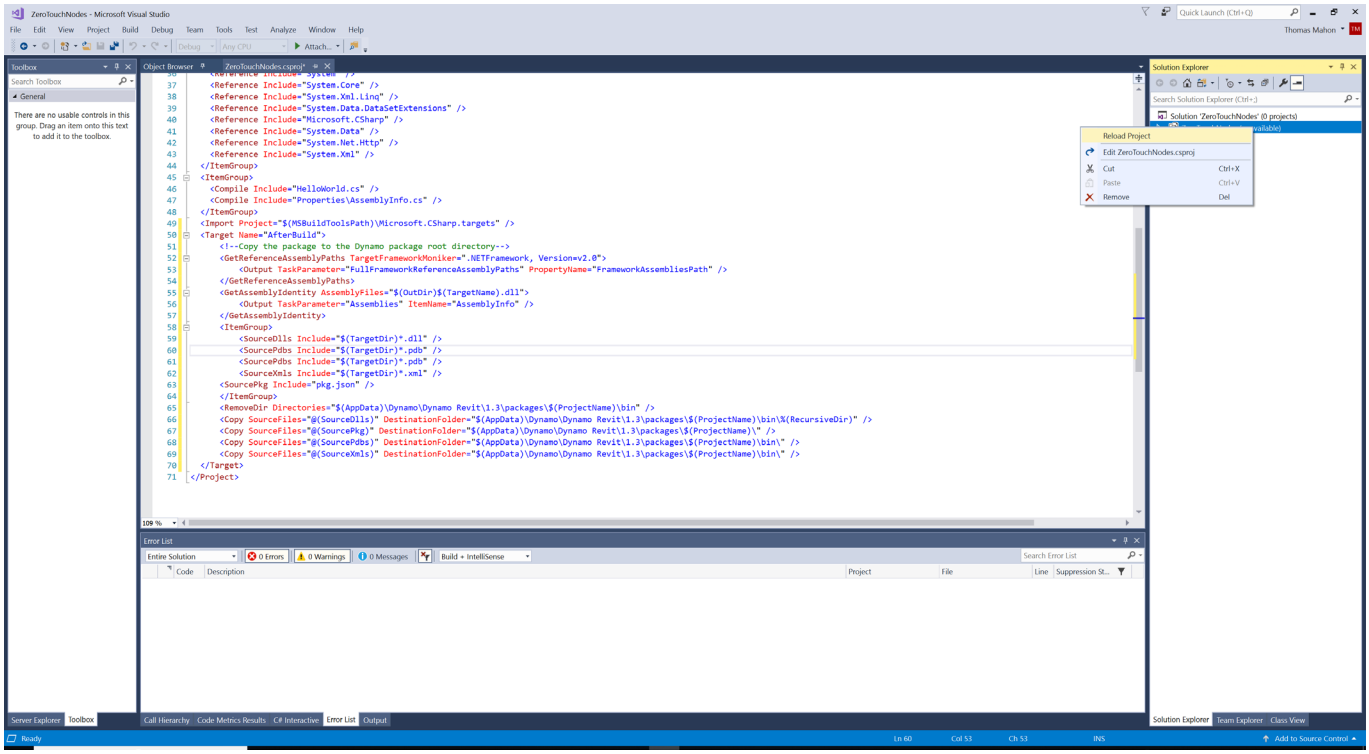
Next, right-click the ZeroTouchNodes solution file in the Solution Explorer again, and click on Edit ZeroTouchNodes.csproj from the drop-down menu. The csproj file will be displayed. Scroll down to the bottom of the file and copy and paste the XML below before the closing </project> tab:

```
<Target Name="AfterBuild">
    <!--Copy the package to the Dynamo package root directory-->
    <GetReferenceAssemblyPaths TargetFrameworkMoniker=".NETFramework, Version=v2.0">
        <Output TaskParameter="FullFrameworkReferenceAssemblyPaths"
PropertyName="FrameworkAssembliesPath" />
    </GetReferenceAssemblyPaths>
    <GetAssemblyIdentity AssemblyFiles="$(OutDir)$(TargetName).dll">
        <Output TaskParameter="Assemblies" ItemName="AssemblyInfo" />
    </GetAssemblyIdentity>
    <ItemGroup>
        <SourceDlls Include="$(TargetDir)*.dll" />
        <SourcePdbs Include="$(TargetDir)*.pdb" />
        <SourcePdbs Include="$(TargetDir)*.pdb" />
        <SourceXmls Include="$(TargetDir)*.xml" />
    <SourcePkg Include="pkg.json" />
    </ItemGroup>
    <RemoveDir Directories="$(AppData)\Dynamo\Dynamo Revit\2.0\packages\$(ProjectName)\bin" />
    <Copy SourceFiles="@(SourceDlls)" DestinationFolder="$(AppData)\Dynamo\Dynamo
Revit\2.0\packages\$(ProjectName)\bin\%(RecursiveDir)" />
    <Copy SourceFiles="@(SourcePkg)" DestinationFolder="$(AppData)\Dynamo\Dynamo
Revit\2.0\packages\$(ProjectName)\" />
    <Copy SourceFiles="@(SourcePdbs)" DestinationFolder="$(AppData)\Dynamo\Dynamo
Revit\2.0\packages\$(ProjectName)\bin\" />
    <Copy SourceFiles="@(SourceXmls)" DestinationFolder="$(AppData)\Dynamo\Dynamo
Revit\2.0\packages\$(ProjectName)\bin\" />
</Target>
```

The code snippet automates the creation of the folders Dynamo requires all custom node packages include (bin, dyf and extra), as well as the copying all dll, pdb, xml and json files found in the bin folder of the ZeroTouchNodes project to the Dynamo root package folder whenever we compile. It targets Dynamo v1.3, however any version can be targeted by changing the version number in the directory addresses to the version you have installed.



Save the change by clicking on the save icon and re-open the solution by right-clicking on the project in the Solution Explorer, and select Reload Project:



Step 3 Create Custom Package JSON

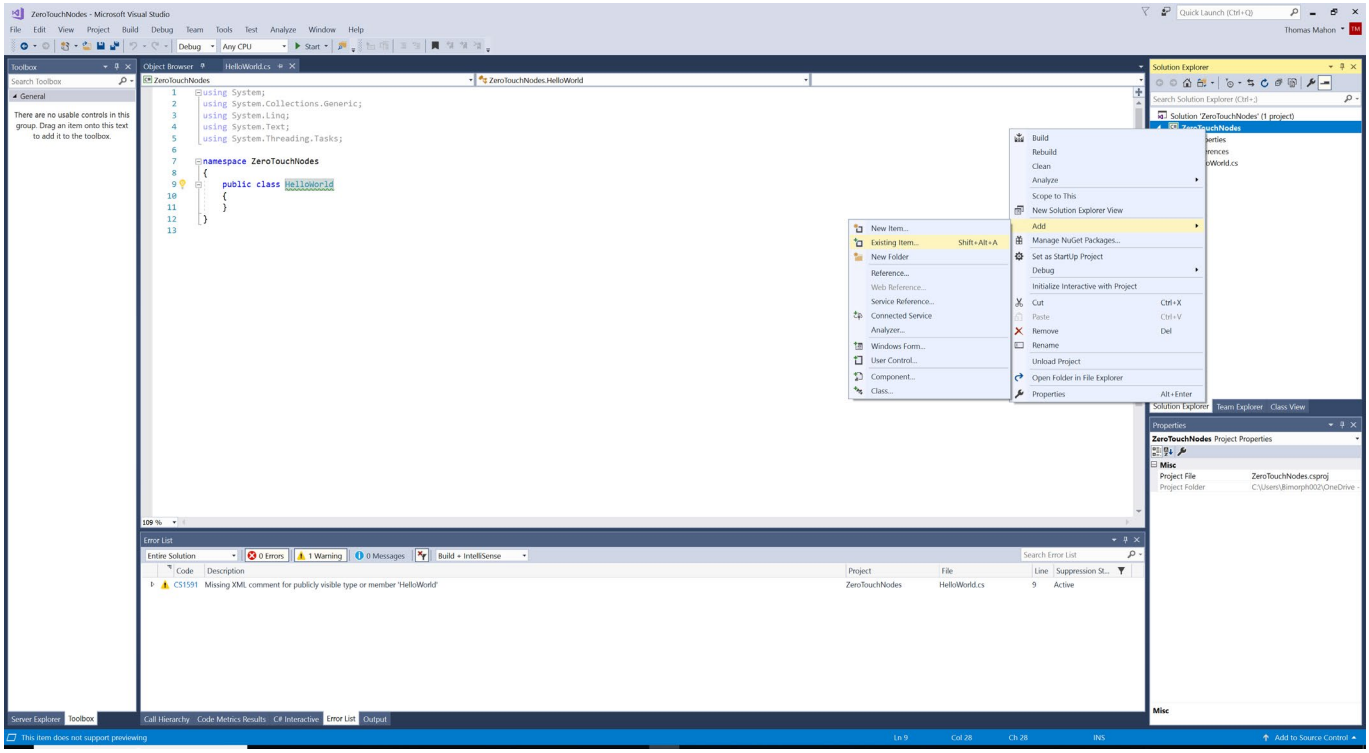
Now the configuration of our ZeroTouchNode project is complete in Visual Studio, we need to create a JSON file to accompany the folders and files copied to Dynamo's root package directory. Dynamo relies on this JSON file to provide details about the package and point to any libraries which the package utilises; in our case, it's the ZeroTouchNode library we are in the process of writing.

The JSON file can be created from a new text file (txt). Copy the JSON code below and paste it into the text file using any text editor. It includes all the attributes Dynamo requires for custom packages, including the reference to our new ZeroTouchNodes library in the node_libraries in the root folder of the ZeroTouchNodes Visual Studio project and copy/paste the code shown below. Rename the extension of the file to json and ignore any Windows warning about changing file extensions.

The ZeroTouchNodes package will comprise a single dll named ZeroTouchNodes and the node_libraries property of the json file must point to this file otherwise Dynamo will not load the dll and no nodes will appear in the library:

```
{
  "license": "MIT",
  "file_hash": null,
  "name": "ZeroTouchNodes",
  "version": "1.0.0",
  "description": "A collection of Zero Touch Dynamo nodes.",
  "group": "",
  "keywords": [ "ZeroTouch", "dynamo" ],
  "dependencies": [],
  "contents": "Zero Touch Nodes - A collection of simple Zero Touch Nodes Dynamo nodes.",
  "engine_version": "1.0.0",
  "engine": "dynamo",
  "engine_metadata": "",
  "site_url": "",
  "repository_url": "",
  "contains_binaries": true,
  "node_libraries": [
    "ZeroTouchNodes, Version=1.0.0, Culture=neutral, PublicKeyToken=null"]
}
```

For convenience, add the json file to the Visual Studio project so edits can easily be made. To do this, right-click on the ZeroTouchNodes solution file in the Solution Explorer, and click Add > Existing Item, (ensure all file types is select from the browse window file types drop-down) then selection and open the JSON file:



The environment configuration of the project is complete. Now it is time to start creating new custom nodes. In the next step, we look at creating new constructors, methods and properties of the `HelloWorld` class.

3. KEY C# CONCEPTS

C# Syntax Introduction

Before we begin writing our first custom Zero Touch node, it is important to understand the basics of C# syntax, the architecture of a simple program (in this case, the 'program' is our custom nodes) and key C# concepts as this will make writing our nodes in the next few exercises easy to follow and understand.

Note that the explanations and concepts introduced throughout this document are orientated specifically for introductory-level custom node development; details which are not relevant (but important to know) have therefore been omitted. Delegates are therefore encouraged to continue developing their understanding of C# concepts following the lab.

Let's start with some basics syntax concepts, (referenced from https://docs.microsoft.com):

1. **Value and reference types**
   Variables that are based on value types directly contain values (the memory address stores the value). Assigning one value type variable to another copies the contained value.

   This differs from the assignment of reference type variables, which copies a reference to the object but not the object itself (a reference to the object, and the object itself, are both allocated memory addresses).

   a. Example value types: `bool`, `int` (integer), `double` (decimal number)
   b. Example reference type: `HelloWorld`

   Further reading: Research the memory 'heap' and 'stack' to understand how value and reference types are stored and accessed.

2. **Keywords**
   Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program. Fortunately, Visual Studio will highlight keywords to aid you as you write code to avoid any potential conflicts. To see the list of keywords, visit https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/.



	as	base	bool
eak	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
	explicit	extern	false

3. **Modifiers**
   Modifiers are used to modify declarations of types and type members. Zero touch node development typically requires the use of the `static` modifier if your class returns objects different from its own type. In most cases, it's the only modifier (excluding Access Modifiers) you'll need to use.

4. **Access Modifiers (aka Access Specifiers)**
   Access modifiers are keywords used to specify the declared accessibility of a member or a type. The accessibility levels in C# are: `public`, `private`, `internal`, `protected`, `protected internal` and `private protected`. For the purposes of the lab we will only use 2:

   a. Public – Access is not restricted. You can call public members or types anywhere in your program. So can any external application, i.e. Dynamo
   b. Private – Access is limited to the containing type. You can only call members or types within their scope (typically the class). External applications cannot access nor call private members or types.

5. **Statements**
   The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

   A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block in C# is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

```
static void Main()
{
    // Declaration statement.
    int counter;

    // Assignment statement.
    counter = 1;

    // Error! This is an expression, not an expression statement.
    // counter + 1;

    // Declaration statements with initializers are functionally
    // equivalent to  declaration statement followed by assignment statement:
    int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
    const double pi = 3.14159; // Declare and initialize  constant.

    // foreach statement block that contains multiple statements.
    foreach (int radius in radii)
    {
        // Declaration statement with initializer.
        double circumference = pi * (2 * radius);

        // Expression statement (method invocation). A single-line
        // statement can span multiple text lines because line breaks
        // are treated as white space, which is ignored by the compiler.
        System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                 counter, radius, circumference);

        // Expression statement (postfix increment).
        counter++;

    } // End of foreach statement block
} // End of Main method body.
```

6. **Using directive**

   The using directive has three uses:

   a. To allow the use of types in a namespace so that you do not have to qualify the use of a type in that namespace:

   ```
   using Autodesk.Revit.DB;
   ```

   b. To allow you to access static members and nested types of a type without having to qualify the access with the type name:

   ```
   using static System.Math;
   ```

   c. To create an alias for a namespace or a type. This is called a using alias directive:

   ```
   using ProtoCurve = Autodesk.DesignScript.Geometry.Curve;
   ```

   The scope of a using directive is limited to the file in which it appears and can appear:

   a. At the beginning of a source code file, before any namespace or type definitions.
   b. In any namespace, but before any namespace or types declared in this namespace.

   Otherwise, you will get a compiler error.

7. **Dot operator**
   The dot operator (.) is used for member access. The dot operator specifies a member of a type or namespace. For example, the dot operator is used to access specific methods within the .NET Framework class libraries

```
Line dynamoLine = Line.ByStartPointEndPoint(startPoint, endPoint);
```

   It is used use to access members of classes (and class instances), such as its properties:

```
double lineLength = dynamoLine.Length;
```

   And the dot is used to form qualified names, which are names that specify the namespace or interface, for example, to which they belong:

```
Autodesk.Revit.DB.Wall wall = Autodesk.Revit.DB.Wall.Create(document, profile, false);
```

8. **Comments**
   Always add comments throughout your code to document the purpose of your program and provide more meaning. All good developers comment their code – it makes it easier for you and others to comprehend the logic, flow and structure of your program. The C# compiler ignores comments, so you can include them anywhere in your program without any adverse effects.

   To create a single-line comment, use a double forward-slash:

```
double lineLength = dynamoLine.Length; //Get the total length of the ProtoGeometry Line
```

   A multi-line comment can be achieved by containing the text within and forward slash and asterisk /* and terminates with an asterisk and forward slash:

```
/*Get the total length of the ProtoGeometry Line
The length is measured in mm*/
double lineLength = dynamoLine.Length;
```

Components of a Simple Program

Programs written in C# are structured into hierarchal structures that comprise a library, which is organised into namespaces, classes, and class members. Class members include (but are not limited to) constructors, fields, properties and methods. Custom nodes written in C# confirm to these same hierarchies.

1. **Namespace**
   A namespace essentially a container with a meaningful name which the developer can use to add more legibility and structure to their program by grouping related classes together. Our `HelloWorld` class includes a namespace called ZeroTouchNodes, and within this namespace, the HelloWorld class is declared.

Namespace declaration comprises the `namespace` keyword followed by a name of your choice – we will use ZeroTouchNodes. Following the rules of C# syntax, we use open '{' and closed '}' curly brackets to define the scope of the namespace:

```
namespace ZeroTouchNodes
{

}
```

2. Class

Classes are one of the basic constructs of the common type system in the .NET Framework. It is essentially a data structure that encapsulates a set of data and behaviours that belong together as a logical unit. The data and behaviours are the members of the class, and they include its methods, properties, and events, and so on.

A class declaration is like a blueprint that is used to create instances or objects at run time. If you define a class called `HelloWorld`, `HelloWorld` is the name of the type. Multiple instances of the same `HelloWorld` type can be created, and each instance can have different values in its properties and fields.

A class is a reference type. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an instance of the class by using the `new` operator, or assign it an object of a compatible type that may have been created elsewhere.

a. Declaring a Class

Classes are declared by using the class keyword followed by a unique identifier as shown in the following example.

Tip: your classes (i.e. your custom nodes) should use the `public` access modifier if they need to appear in the Dynamo node library after Zero Touch import.

```
public class HelloWorld
{

}
```

The `class` keyword is preceded by the access level. Because `public` is used in this case, anyone can create instances of this class. The name of the class follows the `class` keyword. The name of the class must be a valid C# identifier name. The remainder of the definition is the class body between the open '{' and closed '}' curly

brackets, where the behaviour and data are defined. Fields, properties, methods, and events on a class are collectively referred to as class members.

b. Creating Objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the `new` keyword followed by the name of the class that the object will be based on, like this:

```
HelloWorld hello = new HelloWorld("Hello BILT Euro 2018");
```

3. Constructor

Whenever a class is created, its constructor is called. A class may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

a. Default Constructors

If you don't provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values as listed in the Default Values Table.

Tip: Zero Touch import will convert default constructors into a node visible in the Dynamo library. To prevent this, you should always declare a default constructor explicitly and change its access level to `private`:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        //Default constructor set to private
        private HelloWorld() { }
    }
}
```

b. Constructor syntax

A constructor is a method whose name is the same as the name of its type. Its method signature includes only the method name and its parameter list , which is always encased in curved brackets; it does not include a return type. The following example shows the constructor for our `HelloWorld` class:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        private string message;

        //HelloWorld constructor
```

```
        public HelloWorld(string myMessage)
        {
            message = myMessage;
        }
    }
}
```

4. Fields, Properties and Methods

Classes can also include fields, properties and methods. Fields are variables used to store values relevant to a class instance. Properties are used to access a class instances fields and either get (return) or set its values. Methods are declared with a return type (or use the `void` keyword if they do not) and include statements which perform a procedure relevant to the class. In the following exercises, we explore fields, properties and methods, their implementation and how they are used.

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        //HelloWorld field
        private string message;
        private int length;

        //Property which
        public int Length
        {
            get
            {
                return length;
            }
        }

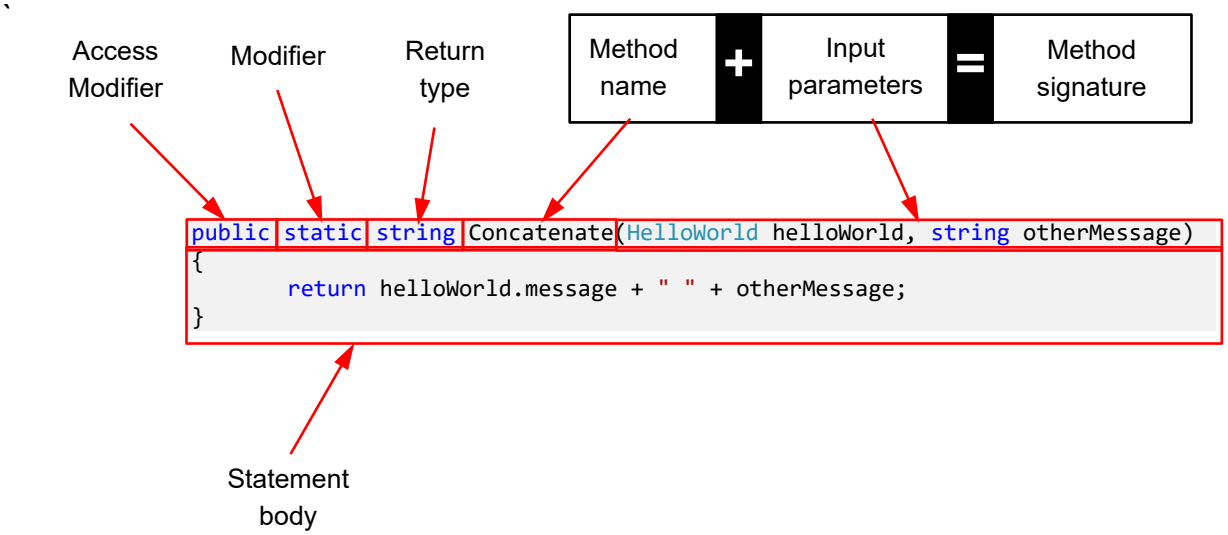
        //A constructor - 'creates' and instance of HelloWorld class
        public HelloWorld(string myMessage)
        {
            message = myMessage;
            length = myMessage.Length;
        }

        //A method (And member of) MyClass
        public static string Concatenate(HelloWorld helloWorld, string otherMessage)
        {
            return helloWorld.message + " " + otherMessage;
        }
    }
}
```

Syntax of a Method

The following diagram summarises the basic syntax structure of a method. Dynamo Zero Touch import will convert `public` class members, which include methods, into nodes. The modifiers are both optional; without any access modifier for example, the method is set to `private` by default. In cases where the method doesn't return anything, replace the return type with the `void` keyword:

`


Access Modifier
Modifier
Return type
Method name
+
Input parameters
=
Method signature
public static string Concatenate(HelloWorld helloWorld, string otherMessage)
{
return helloWorld.message + " " + otherMessage;
}
Statement body

4. WRITING CUSTOM NODES – THE BASICS

HelloWorld Custom Nodes

Now the basics concepts are understood, its time to begin practically applying this knowledge to develop our HelloWorld class nodes. We are going to create a default constructor, add new members to our class, and compile our program (i.e. our custom nodes) to begin testing it.

Create a Constructor

The first step involves creating a default constructor. Applying the concepts learned in the previous section, we need to ensure our class is `public` so it displays in the Dynamo library after Zero Touch import, and we need to declare a default constructor following the syntax, so our program should look like this once these steps are complete:

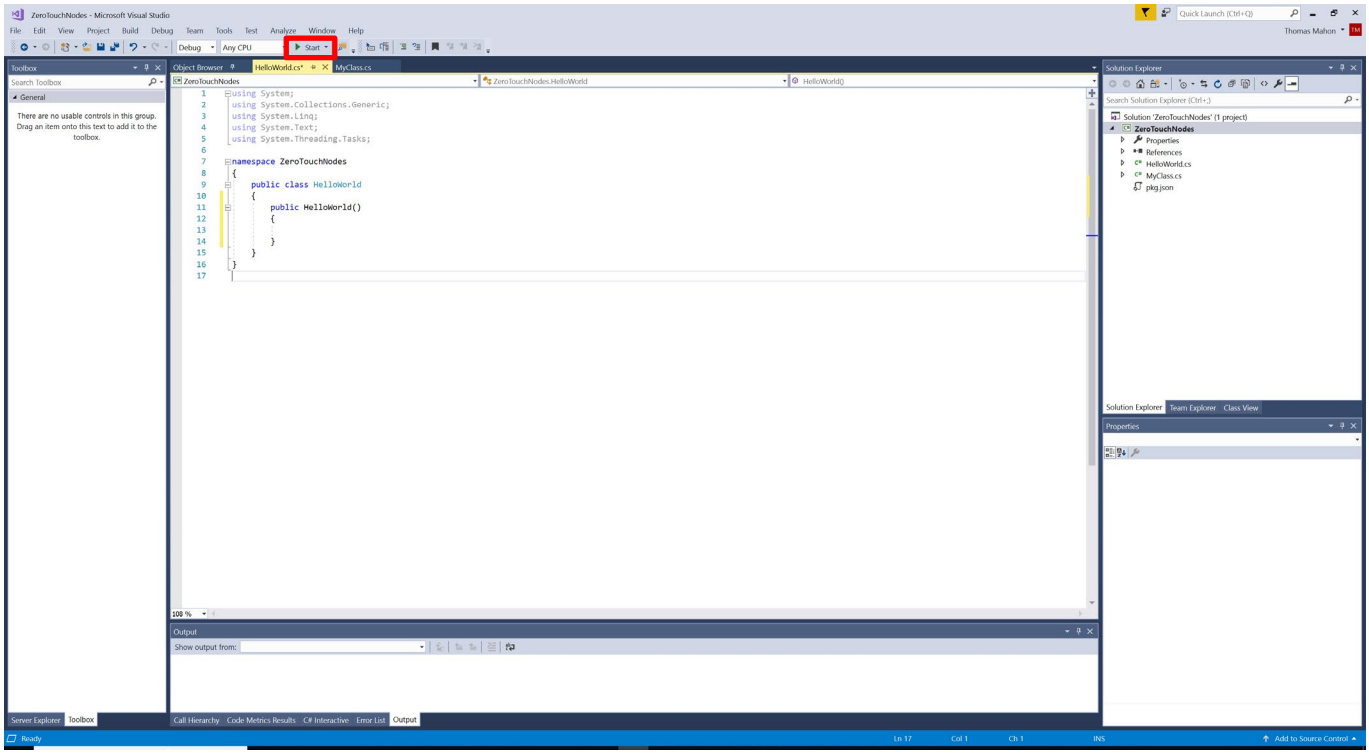
```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        public HelloWorld()
        {

        }
    }
}
```

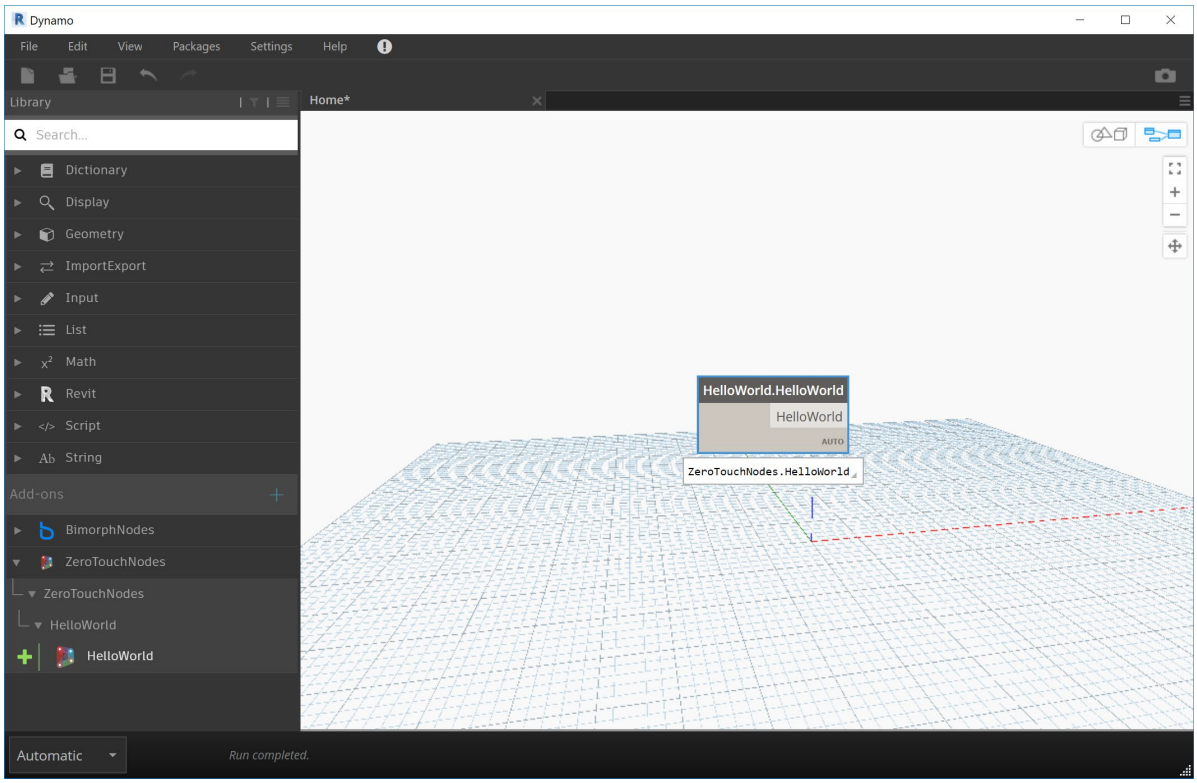
Compile the Solution and Test the Node

We've just written a basic program. Now lets compile our code into a library (which is output as a dll file – a 'Dynamically Linked Library') and test the results in Dynamo. To do this, simply click the start button in visual studio to begin the compile process and Revit will automatically launch so we can begin testing. Note that the solution can also be compiled using Build tab > Build Solution (or Rebuild Solution) if you don't want to test in debug mode.

Note that as we've already configured our environment to export the library files and custom node JSON file to the Dynamo root package folder, there are no intermediate steps involved; our nodes should automatically appear in Dynamo's node library.



Launch Dynamo (v2.0 since this is our target platform) and our ZeroTouchNode and place our new HelloWorld node on the graph:



You will see that our `HelloWorld` class constructor node is correctly returning an instance of the `HelloWorld` class. However, right now it is not doing much and serves no purpose. In the next step we will look at developing the constructors method signature and create fields and properties in our class to store the values.

Add Constructor Inputs, Fields and Properties

To add more functionality to our `HelloWorld` class, we will now add an input to the constructor enabling a message to be passed into the constructor as a `string` (text). To do this we need to enter the input type followed by a name within the curved brackets of the constructor declaration. C# is strong-typed language, so variable are always preceded by their type. Add the input using the name message:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        public HelloWorld(string message)
        {

        }
    }
}
```

Now the input is declared, the next consideration is what should happen with the message once its input. If we leave the constructor in its current state, the message does nothing as we haven't told the compiler what to do with it.

This is where we can add a field to our class to store the input message value. We can then access the value stored in the field whenever required using the '.' (dot) operator. Declare a new field called above the constructor named _message. As this field is storing our message, the type of this field must match, so precede the name with `string` to define its type.

We also want to control its access level; typically fields are always set to private following C# encapsulation standards, and our program will also adhere to this principle. Our code should look like this once complete:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        private string _message;

        public HelloWorld(string message)
        {

        }
    }
}
```

Next, we need to assign the message that's input into the constructor to the _message field. To do this we have to specify it in the statement body of our constructor using the '=' (assignment) operator as shown below:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
```

```
        private string _message;

        public HelloWorld(string message)
        {
            _message = message;
        }
    }
}
```

The final step involves creating a `public` property so the field value is accessible at every level, and crucially, shows as a node in the Dynamo library. Properties use the `get` and `set` keywords and adhere to the following syntax:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        private string _message;

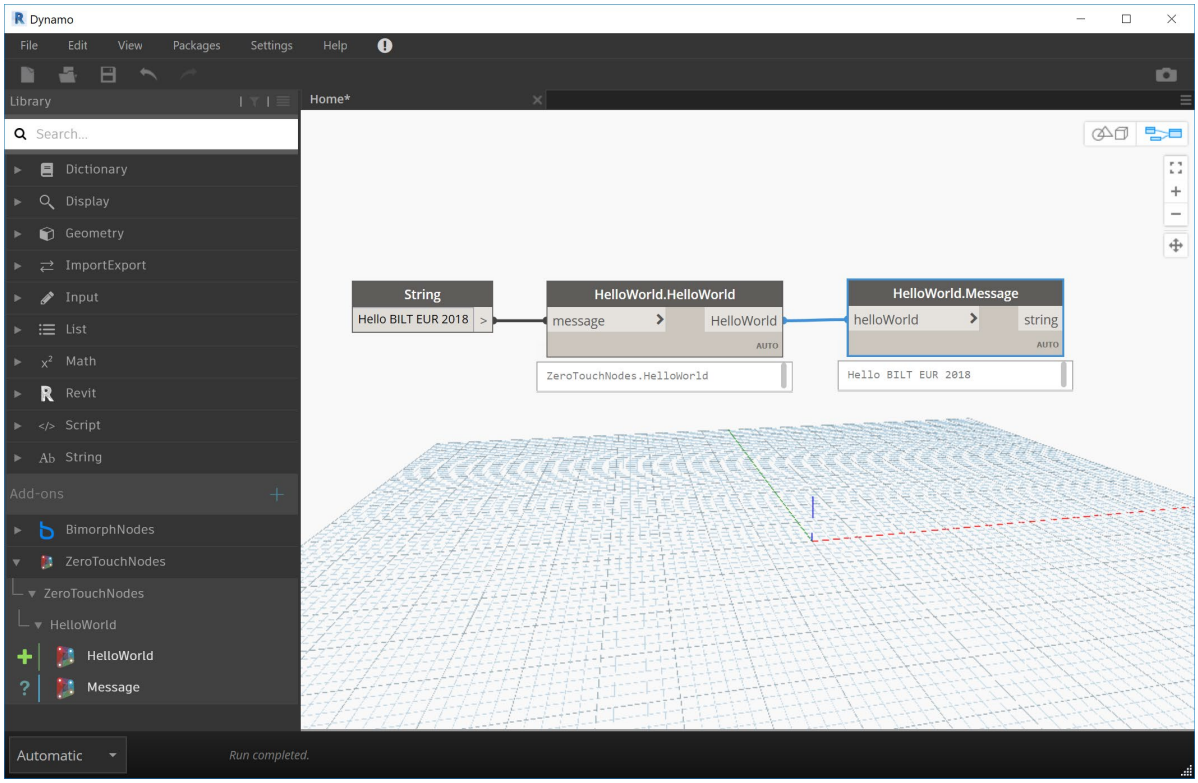
        public string Message
        {
            get { return _message; }
            set { message = value; }
        }

        public HelloWorld(string message)
        {
            _message = message;
        }
    }
}
```

Tip: properties can implement one or the other to restrict modification (`set`) or access (`get`) to the field; just omit the one you don't need. E.g.

```
public string Message
{
    get { return _message; }
}
```

Once these steps are complete, compile the solution again and test the nodes in Dynamo. If successful, you should now see two nodes in the library and our `HelloWorld` constructor will include a `string` input called message:



Add a Method to the Class

To compete the exercise, we will now add a new `public` method to our `HelloWorld` class which can be used to query if a character or word appears in the message of a `HelloWorld` instance, returning `true` if the query is found, or `false` if it's not.

Methods require a return type and optional modifiers. Since our method is querying the message value for a substring, the return type is either `true` or `false` which are values of the `bool` type. Hence, the return type of our method is `bool`.

Now we have established the method signature, declare the method within our class with the name Contains() and precede the name with the return type `bool`.

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        private string _message;

        public string Message
        {
            get { return _message; }
            set { message = value; }
        }

        public HelloWorld(string message)
        {
            _message = message;
        }
```

```
        bool Contains()
        {

        }
    }
}
```

If the access level of the method (or any member of class, or a class itself) is not specified, its set to `private` by default. Since we want the method to show as a node in our library after Zero Touch import, we need to declare the access level as `public`.

The method also requires a string input – the query string – so create an input the method called substring preceded by `string` as the type. You method should now look like this:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        private string _message;

        public string Message
        {
            get { return _message; }
            set { message = value; }
        }

        public HelloWorld(string message)
        {
            _message = message;
        }

        public bool Contains(string subString)
        {

        }
    }
}
```

Next, we need to write the instructions in our method to perform the query action we require. These instructions are declared within the statement body of the method (within the two curly brackets below the method signature).
Since we are returning a `bool` value, we can declare a new `bool` variable, perform the query on the `HelloWorld` Message property, then `return` the result. Declare a new `bool` value called contains adhering to the same syntax rules we've already used in the previous steps.

We are going to assign the result of the query to this value. We use = operator for assignment and it's the Message property we are querying, so we need to write the following line in our method:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        //HelloWorld field
        private string _message;

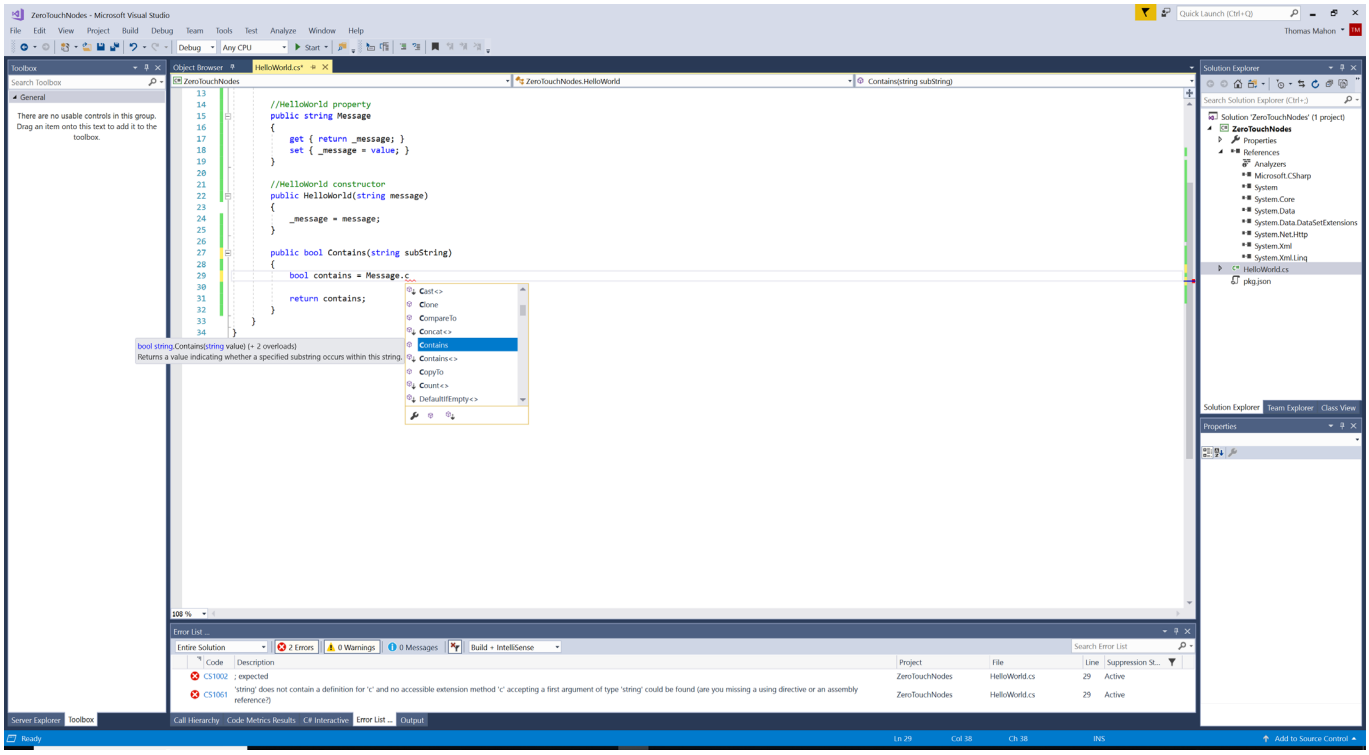
        //HelloWorld property
```

```
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        //HelloWorld constructor
        public HelloWorld(string message)
        {
            _message = message;
        }

        public bool Contains(string subString)
        {
            bool contains = Message
        }
    }
}
```

To perform the query we can utilise the Contains method from the .NET Common Runtime Language (which returns a `bool` value) using the '.' (dot) operator on the Message property. The dot operator will display the IntelliSense list where we can browse all the methods which can be called on the object the dot is used on. Locate the Contains method and double click it to add it into your method:



To complete this step, we need to input a string into the .NET Contains method. This string is the same string input into our method, so pass subString into the curved brackets of the method. Remember, in C#, statements terminate with a semi-colon:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
```

```
        //HelloWorld field
        private string _message;

        //HelloWorld property
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        //HelloWorld constructor
        public HelloWorld(string message)
        {
            _message = message;
        }

        public bool Contains(string subString)
        {
            bool contains = Message.Contains(substring);
        }
    }
}
```

Finally, our method needs to return the result of the query, otherwise the query executes with the result we need but will be disposed (removed from memory) after execution which isn't very useful, as we need the result! Instead we can use the `return` statement which outputs/returns a value or reference type from a method. Therefore, we need to insert the `return` statement. The syntax is always `return` followed by the name of value or reference to output; in our case, it's the contains value:

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        //HelloWorld field
        private string _message;

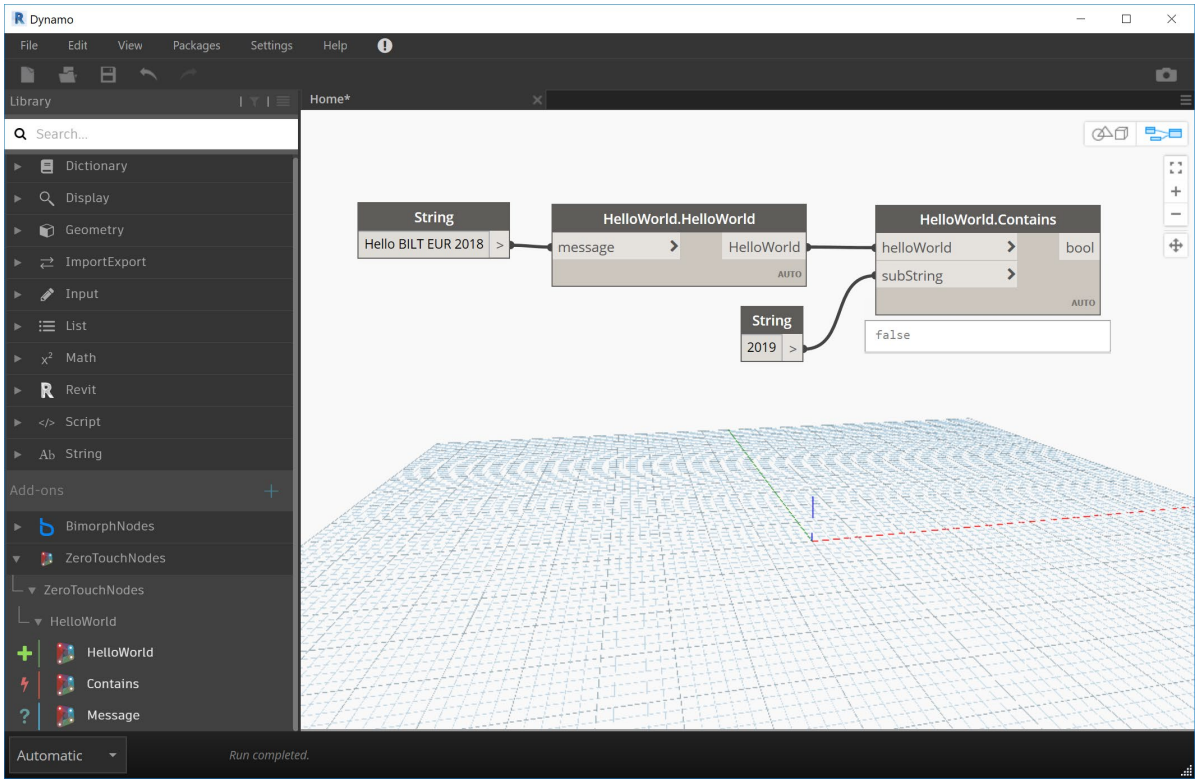
        //HelloWorld property
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        //HelloWorld constructor
        public HelloWorld(string message)
        {
            _message = message;
        }

        //HelloWorld method
        public bool Contains(string subString)
        {
            bool contains = Message.Contains(subString);

            return contains;
        }
    }
}
```

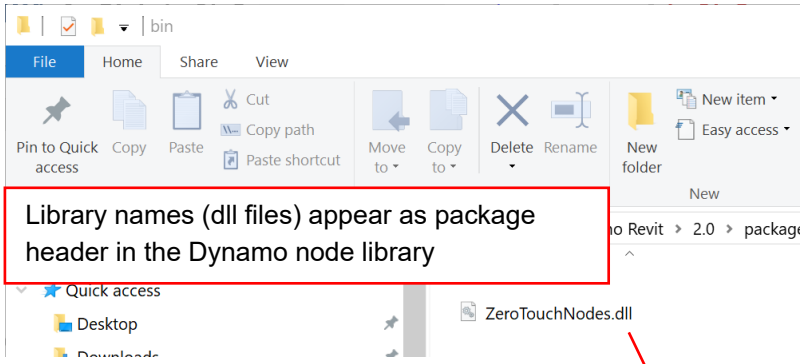
Recompile the solution and test the result in Dynamo. You should see three nodes in the ZeroTouchNodes library. If the new Contains method has been developed correctly, you will see two inputs; one for a `HelloWorld` instance to query, and a another for the string with which to search. The output from the node should be either true or false:



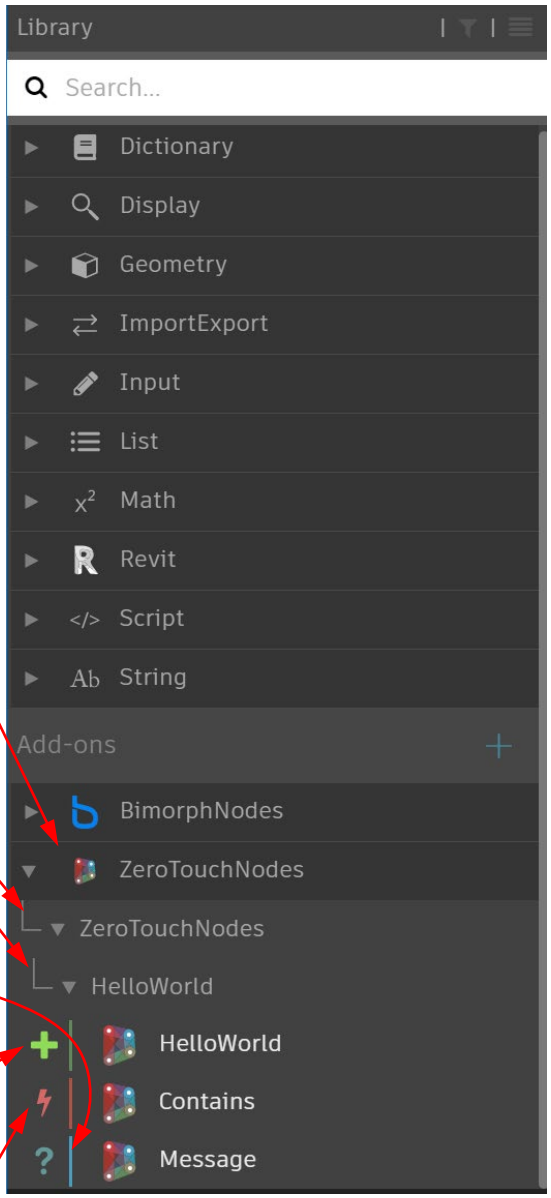
Tip: Do not create custom Zero Touch nodes if the same functionality already exists with a single, or combination of, out-of-the-box node. Dynamo already includes a OOTB node called String.Contains for example and the `HellowWorld`.Message property can be used to provide the input for this node (i.e. a combination of nodes achieves the same end-goal). The reason then for adding a method called Contains to our `HelloWorld` class is purely for the educational purposes of the lab.

Zero Touch Import – How Dynamo Consumes your Library

We've successfully completed the creation of our first set of custom Zero Touch nodes. The way our program (i.e. our custom nodes) is consumed by Dynamo during Zero Touch import should now be clear. To summarise:



Library names (dll files) appear as package header in the Dynamo node library



Namespace and class names appear as sub-categories in the Dynamo node library

```
namespace ZeroTouchNodes
{
    public class HelloWorld
    {
        private string _message;
```

Properties appear as 'Query' nodes in the Dynamo node library

```
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }
```

Constructors appear as 'Create' nodes in the Dynamo node library

```
        public HelloWorld(string message)
        {
            _message = message;
        }
```

Methods appear as 'Action' nodes in the Dynamo node library

```
        public bool Contains(string subString)
        {
            bool contains = Message.Contains(subString);

            return contains;
        }
    }
}
```

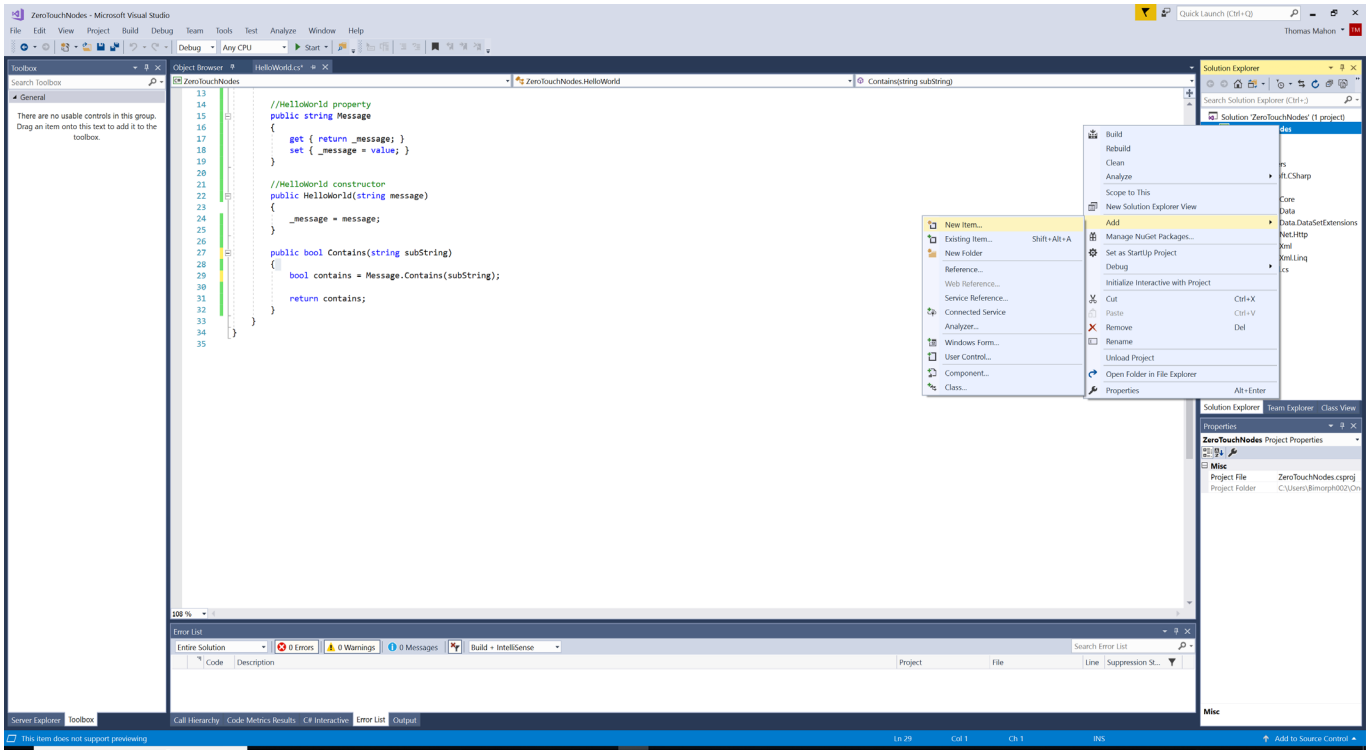
5. WRITING CUSTOM NODES WITH THE REVIT API

Create Revit Elements Programmatically

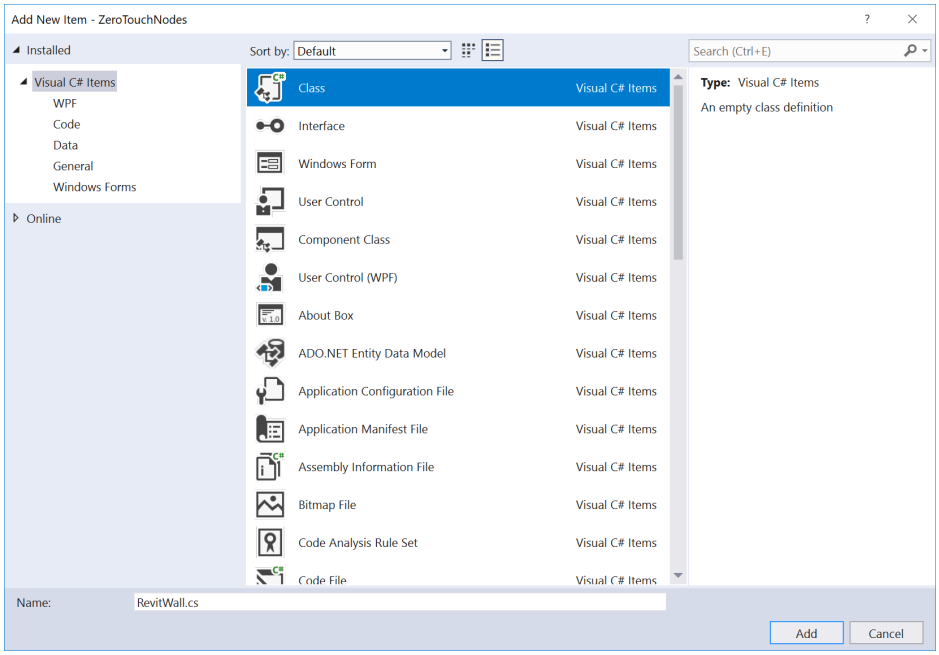
In the final exercise, we are going to create a new class named `RevitWall` which includes a simple method that programmatically creates a new wall element using the Revit API. While the inputs and output of this method is not particularly useful, the purpose of this exercise is to convey the steps needed to implement the Revit API and output Dynamo-compatible elements from our custom nodes. These steps can be applied to create, modify or even delete any element type found in Revit.

Create the Wall Class

The first step involves creating a new class in the project. To achieve this, right-click the solution name in the Solution Explorer, and select Add > New File:



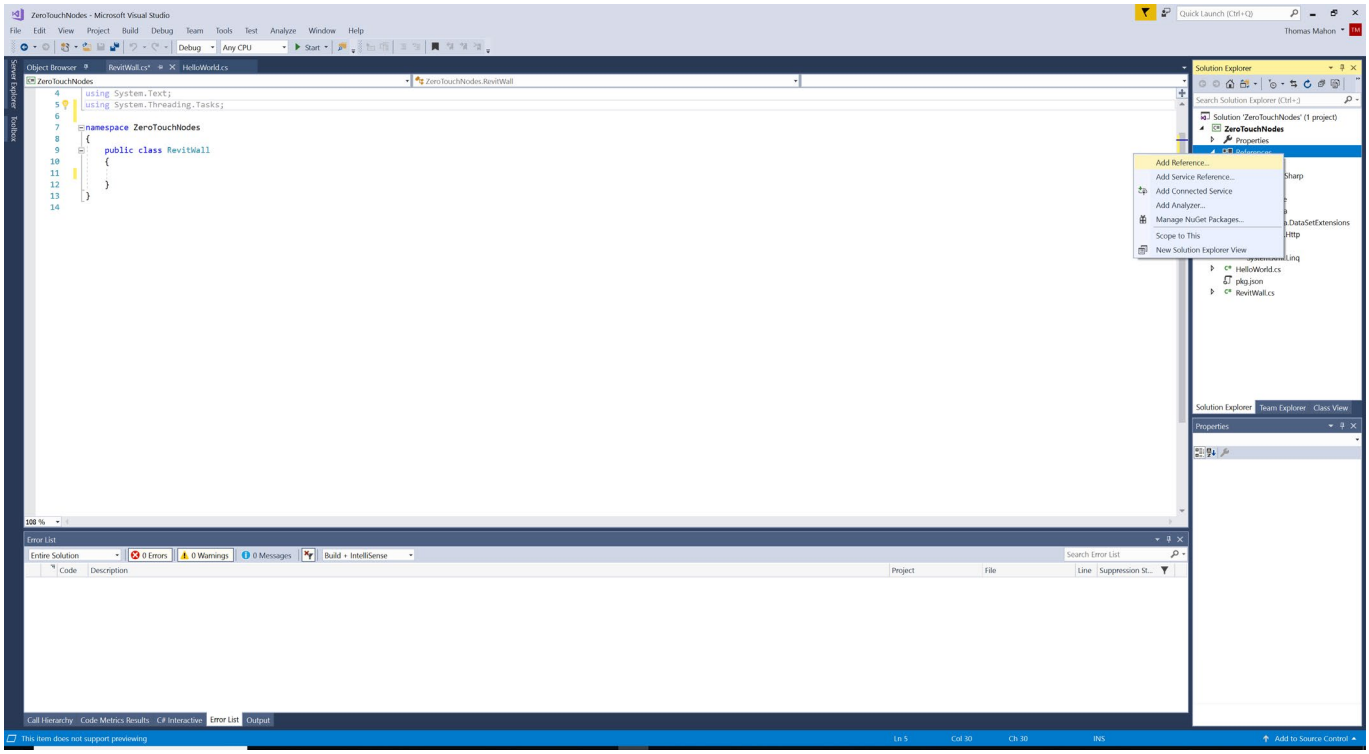
In the dialog box that opens, select Class from the menu and Name it RevitWall.cs. Click the Add button to add the new class to the project:



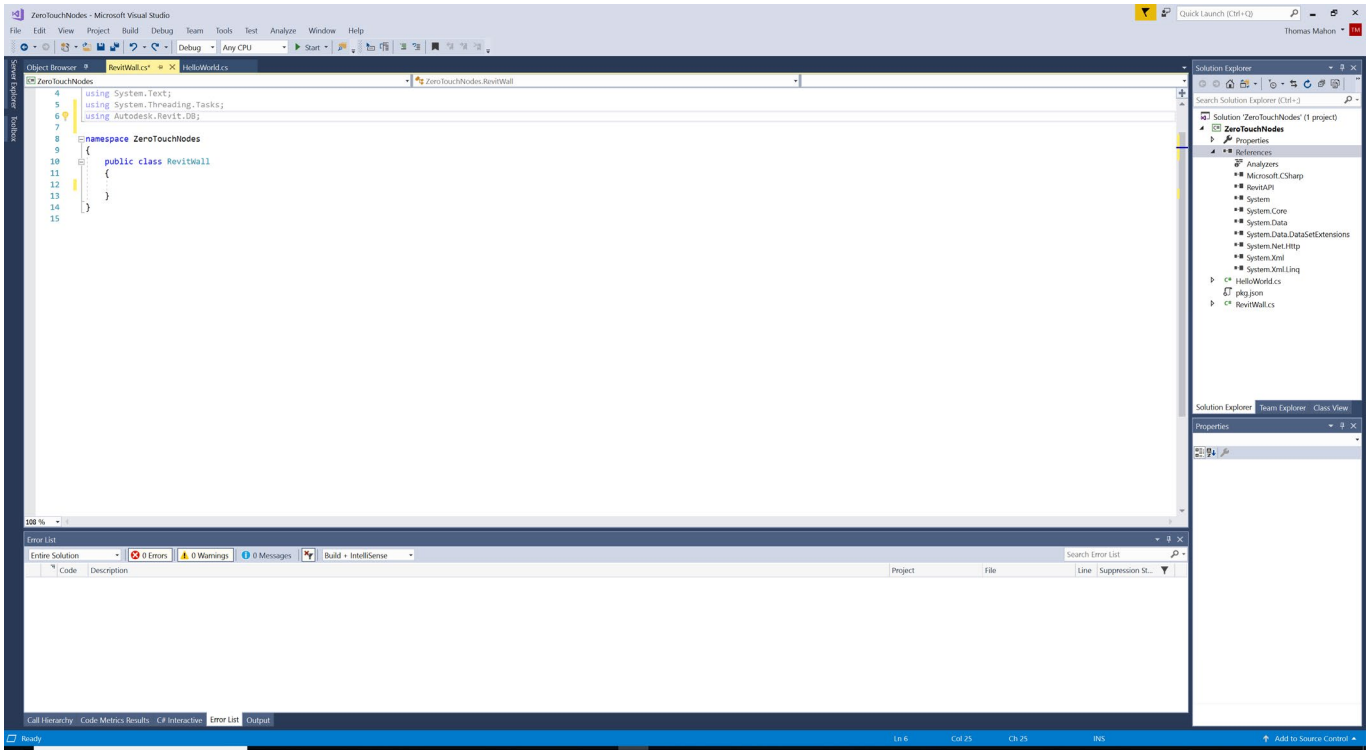
Once the `RevitWall` class is added to the project, ensure its access level is declared as `public` so it is visible in the Dynamo node library after Zero Touch import.

Add a Reference to the Revit API

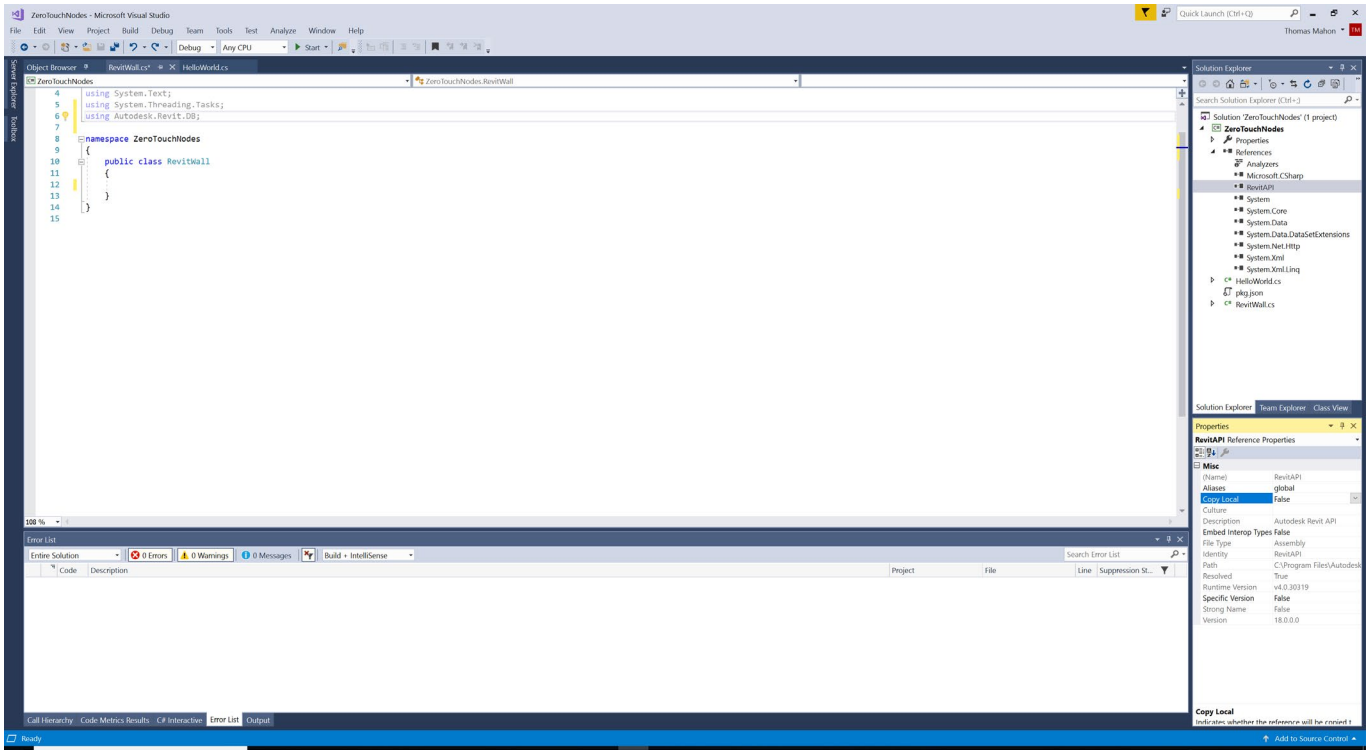
To be able to make calls to the Revit API (or any external library) we need to take two actions. The first involves adding it as a reference in our project. To do this, right-click on References in the Solution Explorer panel and select Add Reference. Using the dialog box that opens, browse to your installation location of Revit (typically: C:\Program Files\Autodesk\Revit 2018) and locate the file **RevitAPI.dll**. Once located, select it and Click the Add button followed by the OK button to add it to your project:



The second involves declaring the `using` directive so we don't need to fully qualify the API calls (i.e. declare the namespace to access the class). Following the guidance from Section 3, add the `using` directive at the top of the `RevitWall` class document followed by Autodesk.Revit.DB to use the primary namespace in the Revit API:

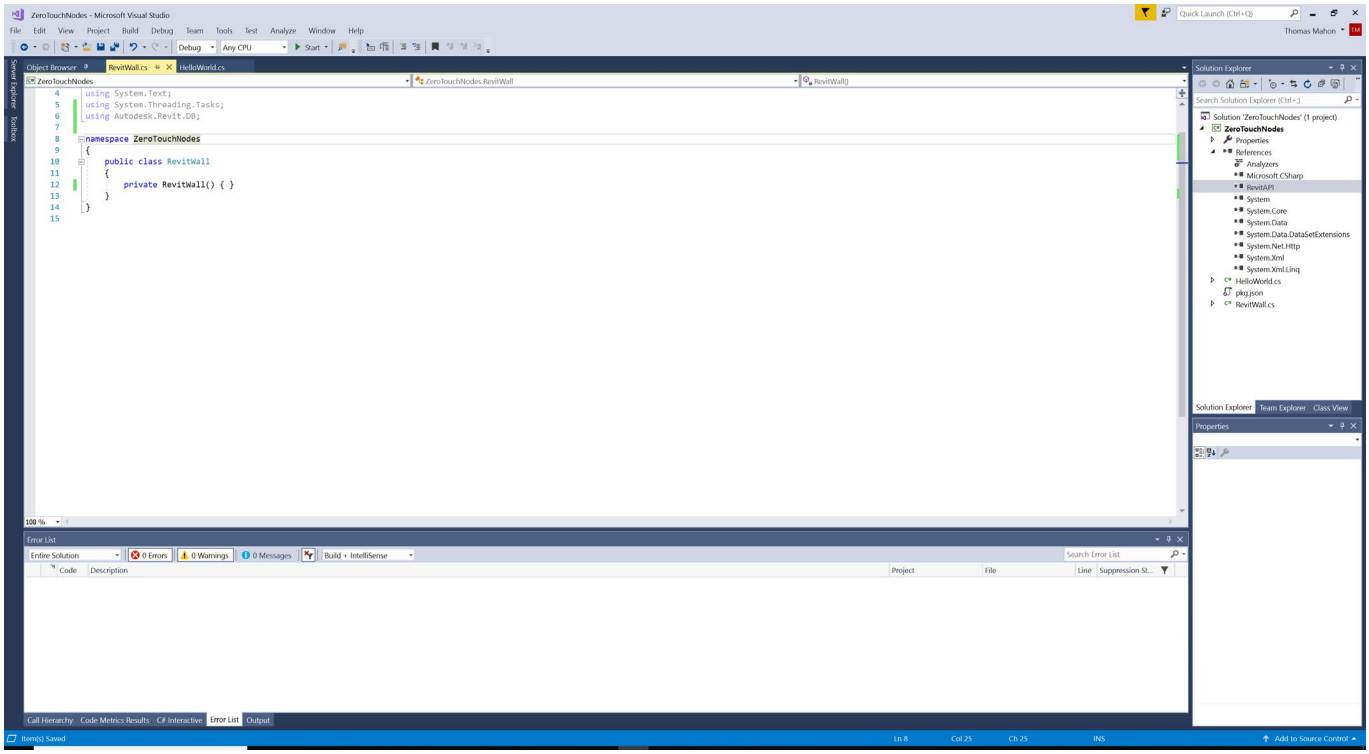


To avoid copying all the Revit API dll's into your project when you compile, click the RevitAPI listing from the References list in the Solution Explorer and set the 'Copy Local' property in the Properties dialog to False:



Create a Default Constructor

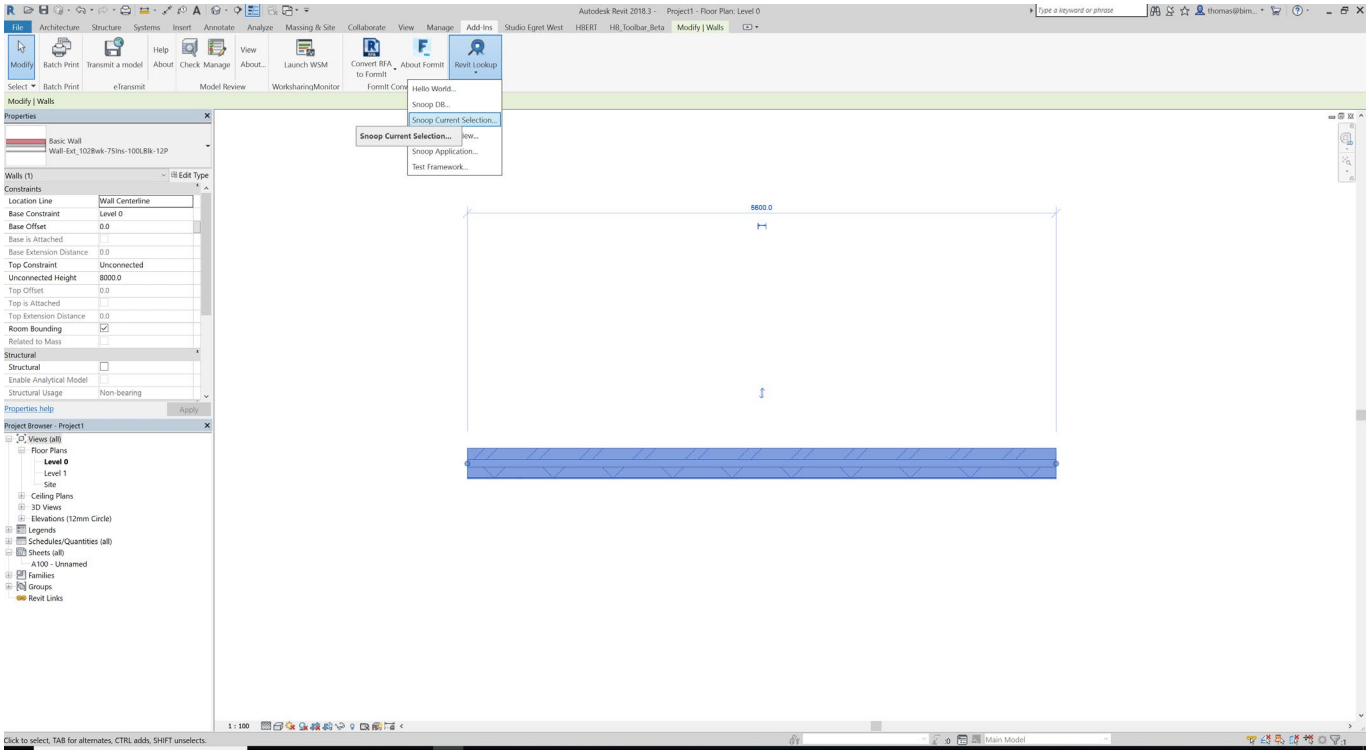
Create a default constructor and set its access level to `private` to prevent any unwanted nodes from appearing in the node library:



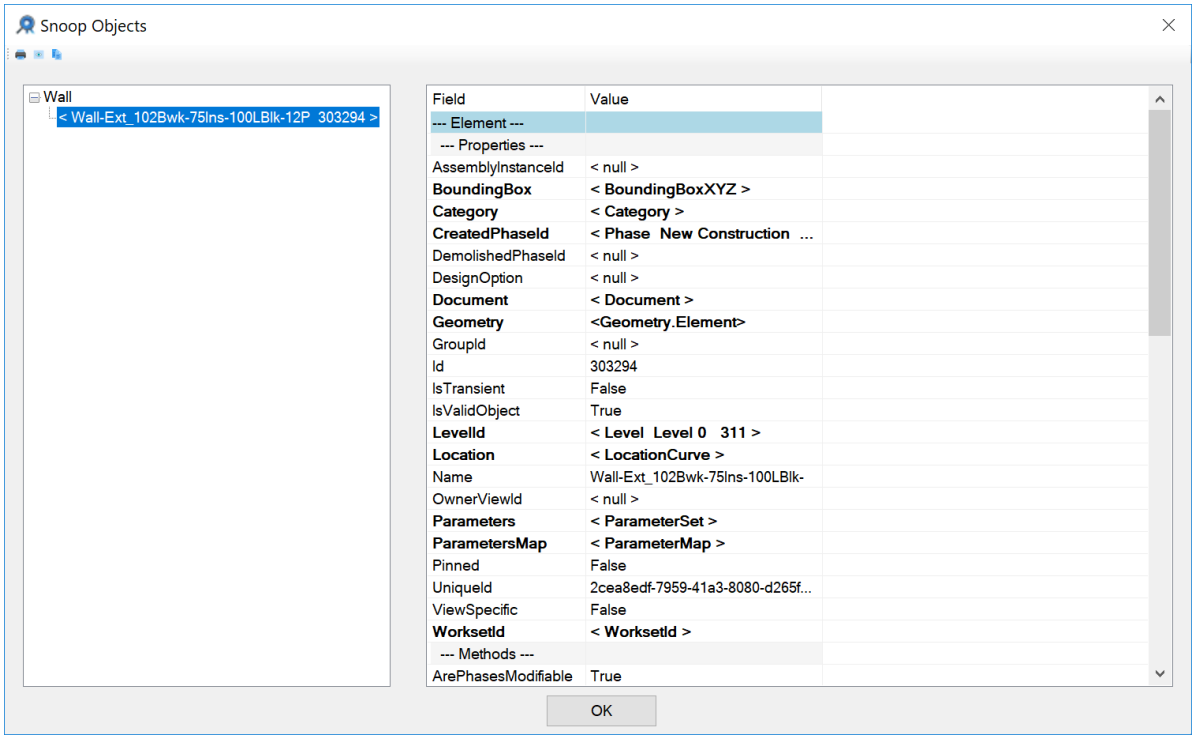
Declare the new Method

Declare a new method in the Wall class called Create() with a `public` access modifier. Next, we need to establish the methods return type. An effective way to do this is to open Revit, create an element of the type we are looking to create (i.e. a Autodesk.Revit.BD.`Wall`) and use Revit Lookup to interrogate its properties and type.

Start by manually creating a wall in Revit, select it, and click Add-Ins tab > Revit Lookup > Snoop Current Selection:



A dialog appears which enables detailed interrogation of the element. We simply want to know its class, which is listed at the top of the hierarchy tree on the left side of the dialog – in this case, it's the Revit API `Wall` class:



Now the class – or rather, the return type of the method – is identified, we can complete the method signature declaration. Our code should look like this:

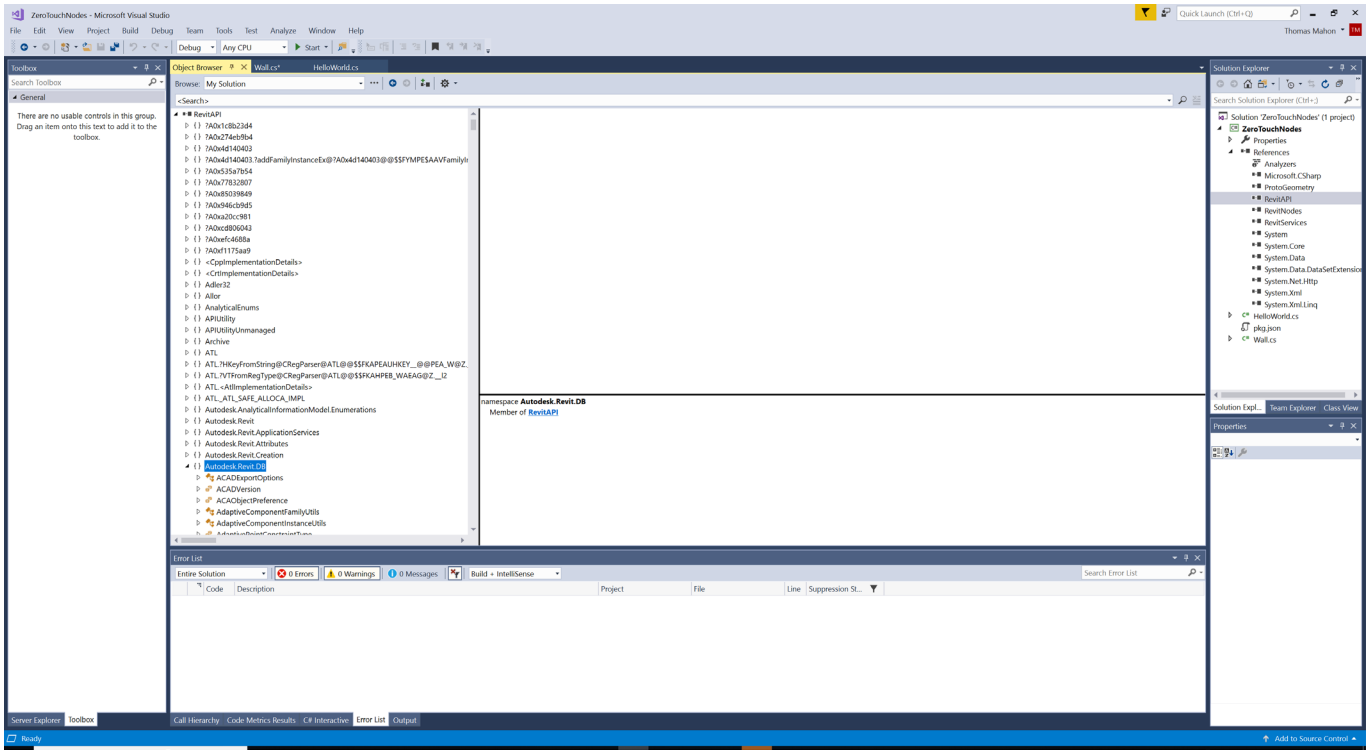
```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create()
        {

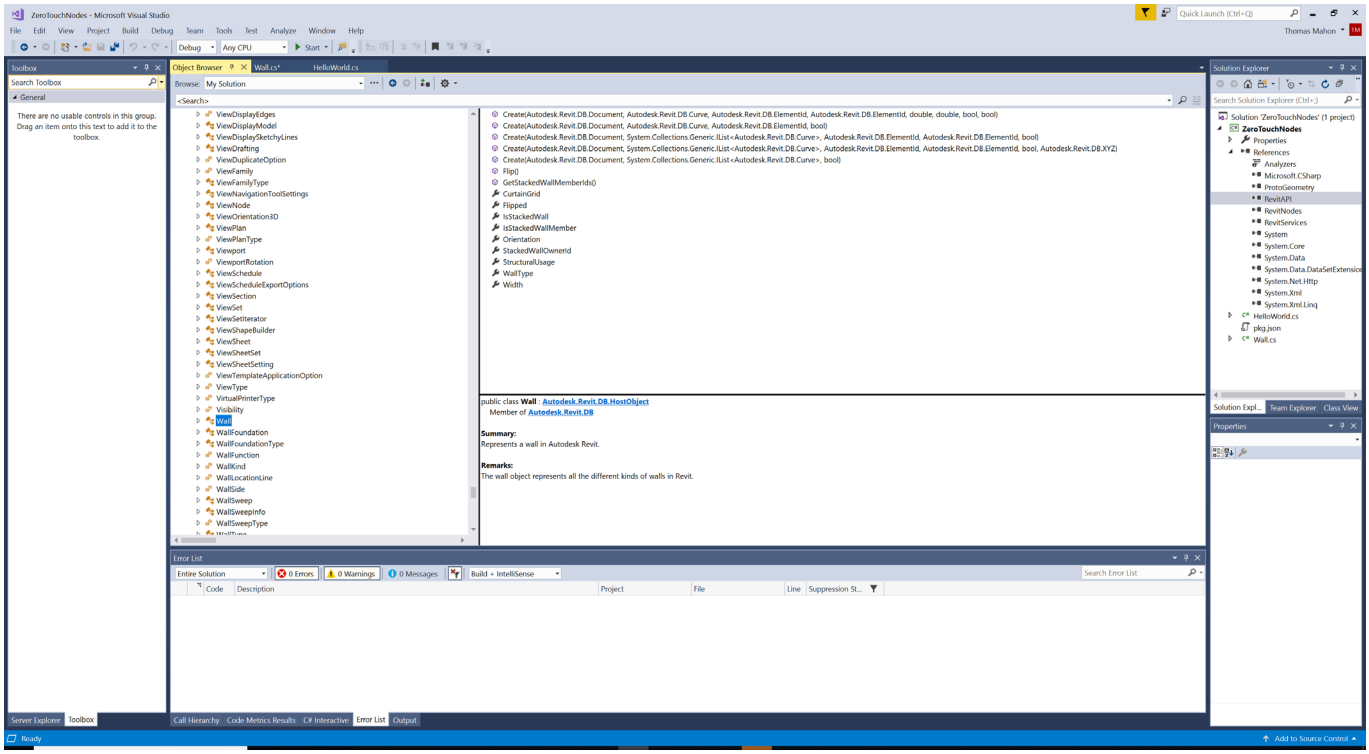
        }
    }
}
```

Identify the Revit API Call

Revit Lookup has also provided us with a line of enquiry for finding an appropriate `Wall` constructor in the Revit API. To begin exploring the API, double click the RevitAPI listed under References in the Solution Explorer. The Object Browser panel will display; find and expand the RevitAPI library, then locate and expand its Autodesk.Revit.DB namespace. You are now looking at the Revit API:



Locate the `Wall` class from this namespace and click it to displays its members in the central dialog box:



We can see that there are a number of overloaded wall constructors; for the purposes of the lab, we are going to use the simplest method, which takes a document, curve, level Id, and boolean:

Create(Autodesk.Revit.DB.Document *document*, Autodesk.Revit.DB.Curve *curve*, Autodesk.Revit.DB.ElementId *levelId*, bool *structural*)

Add the constructor to your method and assign it to a new variable named newWall adhering to C# syntax grammar. A convenient way of implementing a method from the API is to copy/paste its name and signature from the description box directly into your code then update its inputs accordingly:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create()
        {
            Wall newWall = Wall.Create(Autodesk.Revit.DB.Document document, Autodesk.Revit.DB.Curve
curve, Autodesk.Revit.DB.ElementId levelId, bool structural);
        }
    }
}
```

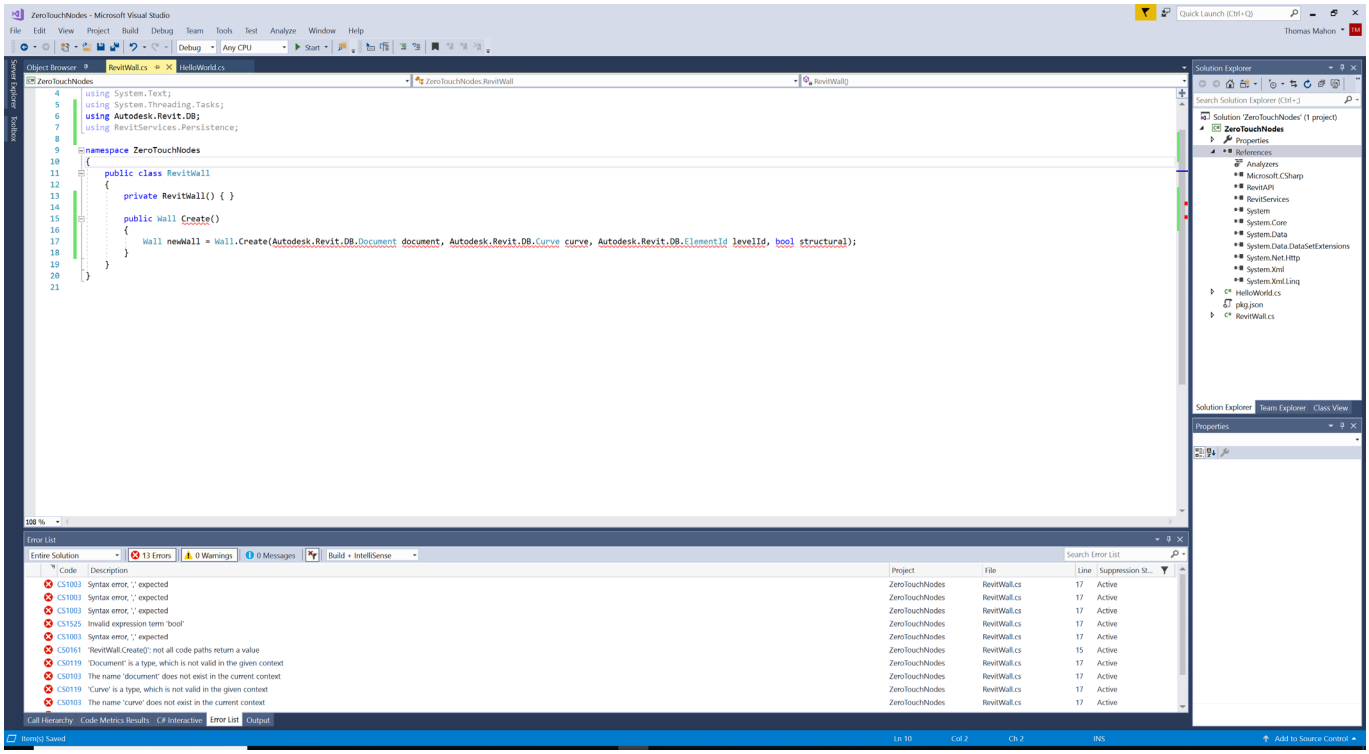
Tip: The `Wall`.Create() method is static so it's called on its type, not an instance, which means the use of the `new` keyword is unneeded. Visual Studio will tell you if the method is static on the first line of the method signature in the description box, enabling you to determine how to call the member:



Adding Inputs for the Wall Constructor

The next step involves creating variables within the methods scope to satisfy the inputs for the `Wall`.Create() method. The first input is an instance of the active `Document` object. Conveniently, we can utilise the RevitServices.dll library (which is included with Dynamo's install) for Revit interop and instantiation of this object. Start by adding the library (typically found in C:\Program Files\Autodesk\Revit 2020\AddIns\DynamoForRevit\Revit) to the project in the same way we added the RevitAPI.dll.

Add the `using` directive to the top of your document followed by RevitServices.Persistence to target the required namespace from this library:



Create a new variable in our method called doc and use the `static` `DocumentManager` class from RevitServices.dll to instantiate a new `Document` object. We can update the first input of the Create() constructor from the API by passing the doc variable into the first of its inputs:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create()
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            Wall newWall = Wall.Create(doc, Autodesk.Revit.DB.Curve curve, Autodesk.Revit.DB.ElementId
levelId, bool structural);
        }
    }
}
```

For the next input we need to create a `Curve` object. For this we will use the Revit API `Line` class and its CreateBound() constructor, which takes two XYZ's (Revit API points/vectors) as inputs:

CreateBound(**Autodesk.Revit.DB.XYZ** *endpoint1*, **Autodesk.Revit.DB.XYZ** *endpoint2*)

Begin by copy/pasting the `Line`.CreateBound() method signature into your method and assign it to a new variable called lnLocationCurve. Since it's a static constructor, we call it directly on the `Line` type without the `new` keyword:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create()
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            Line lnLocationCurve = Line.CreateBound(Autodesk.Revit.DB.XYZ endpoint1,
Autodesk.Revit.DB.XYZ endpoint2);

            Wall newWall = Wall.Create(doc, Autodesk.Revit.DB.Curve curve, Autodesk.Revit.DB.ElementId
levelId, bool structural);
        }
    }
}
```

We want to be able to control the length of this line (and therefore the length of the new Wall) as an input to our method. Therefore, we need to add a new input called lengthInFt of type `double`.

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall(double lenghtInFt) { }

        public Wall Create()
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            Line lnLocationCurve = Line.CreateBound(Autodesk.Revit.DB.XYZ endpoint1,
Autodesk.Revit.DB.XYZ endpoint2);

            Wall newWall = Wall.Create(doc, Autodesk.Revit.DB.Curve curve, Autodesk.Revit.DB.ElementId
levelId, bool structural);
        }
    }
}
```

Tip: The Revit API uses decimal feet as its internal units system. This cannot be changed, so if you prefer working in other unit systems (i.e. metric) or if your inputs are not decimal feet, you will have to perform the units conversion within your code. It is also worth mentioning for angles, it uses radians, not degrees.

To satisfy the inputs for the `Line`.CreateBound() constructor, create two new `XYZ` objects called ptStart and ptEnd above lnLocationCurve. Instantiate the new `XYZ`'s using the `new` keyword, and assign them to these variables. Use the lengthInFt variable as the X dimension of ptEnd:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create(double lenghtInFt)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;
```

```
            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(Autodesk.Revit.DB.XYZ endpoint1,
Autodesk.Revit.DB.XYZ endpoint2);

            Wall newWall = Wall.Create(doc, Autodesk.Revit.DB.Curve curve, Autodesk.Revit.DB.ElementId
levelId, bool structural);
        }
    }
}
```

Pass the new `XYZ`'s into the `Line`.CreateBound() inputs to complete the instantiation of the new `Line` object and pass the lnLocationCurve into the `Wall`.Create() methods second input:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create(double lenghtInFt)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

            Wall newWall = Wall.Create(doc, lnLocationCurve, Autodesk.Revit.DB.ElementId levelId, bool
structural);
        }
    }
}
```

There are two more inputs that need to be satisfied. The final input is not structural, so we can simply input `false` as the wall doesn't need to be structural. For the remaining input '*levelId*' we want this to be an input into our method so the user can provide the level. However, the `ElementId` type can be used as the type as it is not supported by Dynamo. Instead, we can provide the level Id as an integer value using Dynamo's Element.Id node, and instantiate the `ElementId` object within our method using the integer. Therefore, create a new input called levelId of the type `int`:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);
```

```
            Wall newWall = Wall.Create(doc, lnLocationCurve, Autodesk.Revit.DB.ElementId levelId,
false);
        }
    }
}
```

To provide the `ElementId` object, create a new variable of this type called levelElementId and instantiate it using the `new` keyword with the `ElementId` constructor which takes an `int` as an input. Pass this into the `ElementId` input of the `Wall`.Create() method to satisfy its inputs:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

            ElementId levelElementId = new ElementId(levelId);

            Wall newWall = Wall.Create(doc, lnLocationCurve, levelElementId, false);
        }
    }
}
```

Return the Wall Element

To return an object from a method, the `return` statement is used. The syntax is always the `return` keyword followed by the value or object to return with a semi-colon to complete the statement. In our case, that object is the newWall element:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public Wall Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

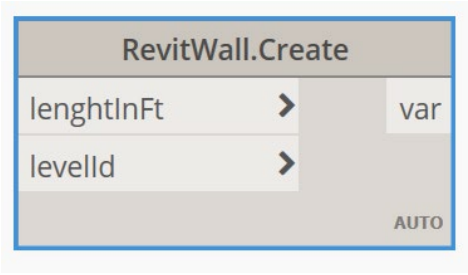
            ElementId levelElementId = new ElementId(levelId);

            Wall newWall = Wall.Create(doc, lnLocationCurve, levelElementId, false);

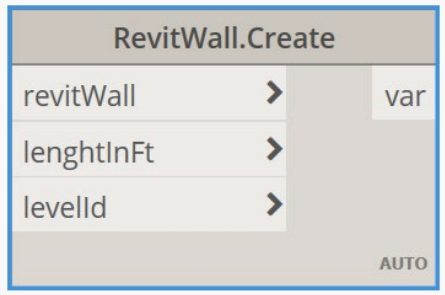
            return newWall;
        }
    }
}
```

Make the Method Static

We can add the `static` modifier to our methods input signature as there is no need to call it on an instance of the type. It is also worth mentioning what would happen in Dynamo after Zero Touch import, if we left out method as non-static; an unnecessary input would appear expecting an instance of the `RevitWall` class, and there would be no means to satisfy the input as the `RevitWall` constructor is set to `private` (i.e. so no `RevitWall` 'create' nodes will display in Dynamo's node Library):



Static method



Non-static method – requires type instance input

Modifiers are added immediately after access modifiers, so our method should look like this once complete:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public static Wall Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

            ElementId levelElementId = new ElementId(levelId);

            Wall newWall = Wall.Create(doc, lnLocationCurve, levelElementId, false);

            return newWall;
        }
    }
}
```

Add a Transaction

There is an additional step we need to take before we can begin testing out method; adding a transaction to our method. A transaction is a context required in Revit in order to make any modifications to a model – this includes the creation or deletion of elements. Without a

transaction in out method would throw an exception as we are attempting to modify the document.

We can utilise Dynamo's RevitServices library again to declare the transaction. We use its `static TransactionManager` class and use its EnsureInTransaction() method to start the transaction. The transaction must be closed using the TransactionTaskDone() method. The modifications to the document must be declared between these two statements, as shown below:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public static Wall Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

            ElementId levelElementId = new ElementId(levelId);

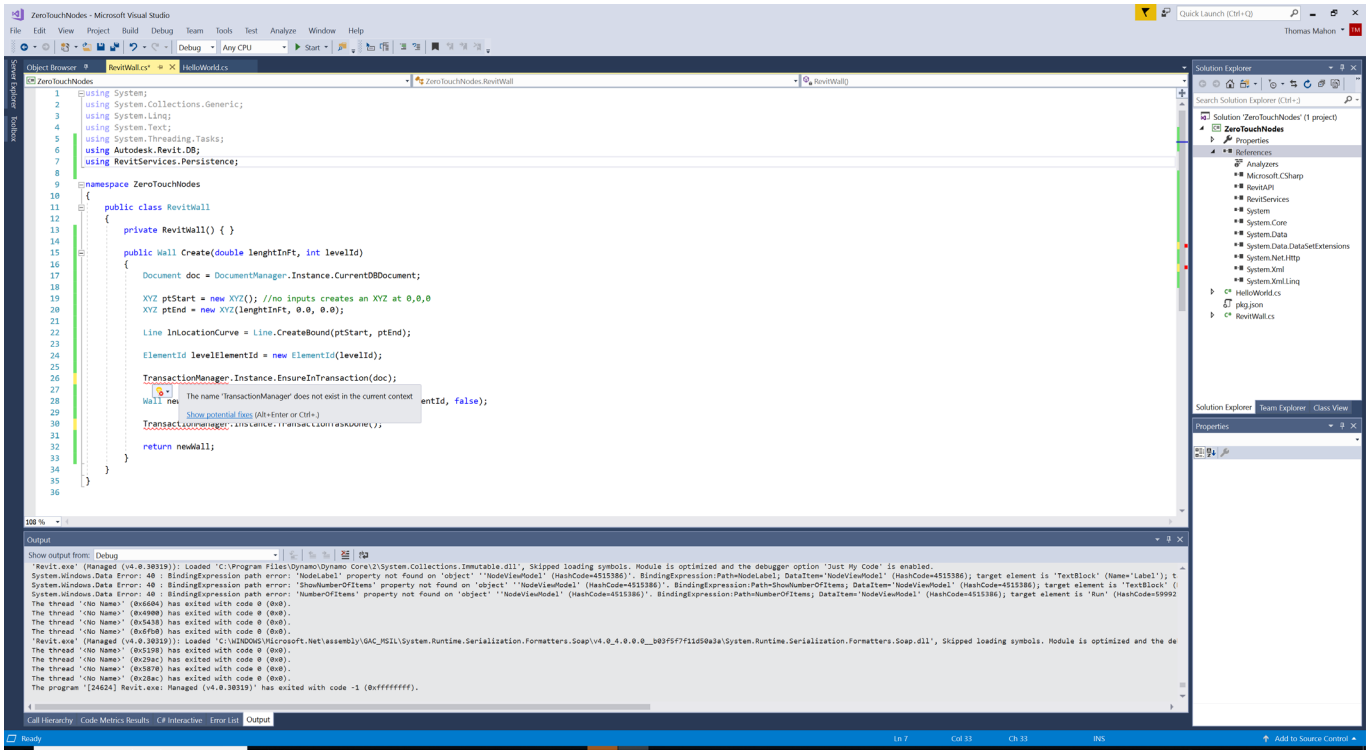
            TransactionManager.Instance.EnsureInTransaction(doc);

            Wall newWall = Wall.Create(doc, lnLocationCurve, levelElementId, false);

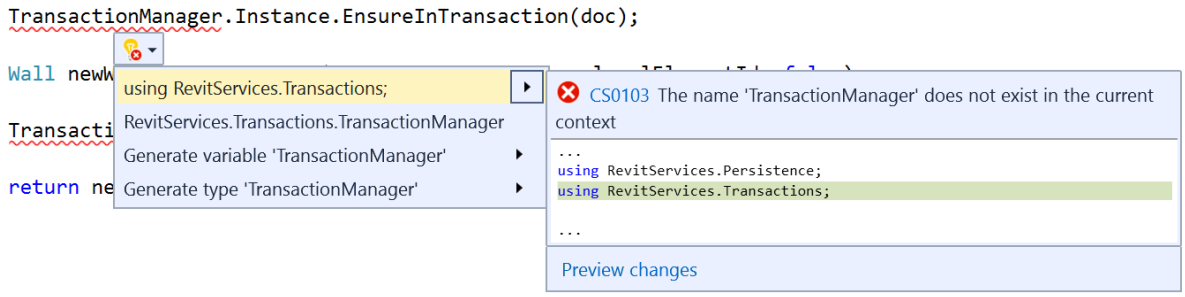
            TransactionManager.Instance.TransactionTaskDone();

            return newWall;
        }
    }
}
```

To utilise this class, we also need to add a `using` directive to point to the namespace containing the `TransactionManager` class. Visual Studio can assist us here; simply hover your cursor over the unknown class and click on 'Show potential fixes':



It will suggest adding the required directive, which you can confirm by clicking on the suggestion:

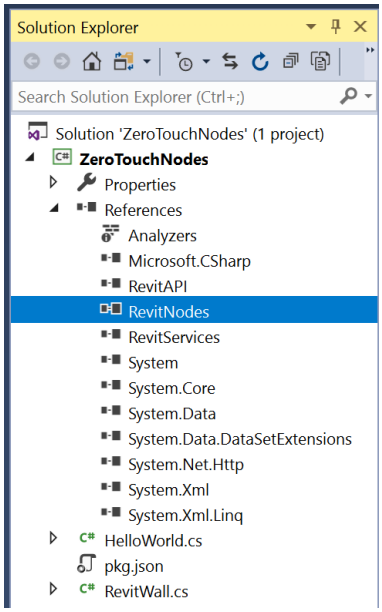


Tip: It is possible to use the Revit API's `Transaction` class for the same purpose, however it is recommended to use the `TransactionManager` class from Dynamo's RevitServices library as It saves you the effort of having to dispose your objects (aka garbage collection), plus it helps to keep both Revit and Dynamo in more consistent states.

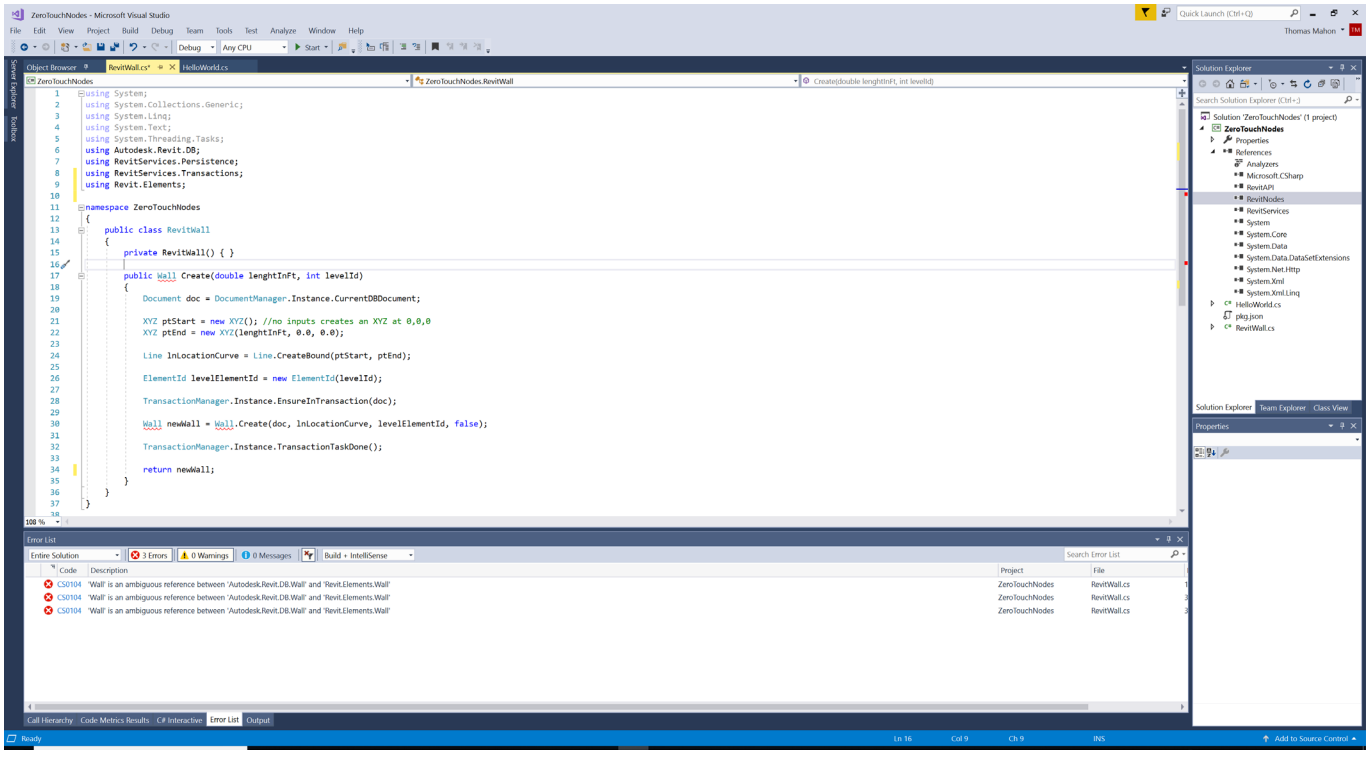
Wrap the Wall Element and Update the Methods Return Type

Before we compile and test, there is an important step we need to take to make the Revit wall element Dynamo-compatible: we need to wrap it in Dynamo's Revit.Elements.`Element` wrapper class. This library is an interop library which delivers all of Dynamo's Revit nodes you see in its node library, as well as helper functions to wrap and unwrap elements from Revit.

The method we call to 'wrap' Revit elements into the Dynamo wrapper class is ToDSType(). It takes a `bool` input, where `true` is used if the element exists in Revit, or `false` if the element is being instantiated by our code. In our case, we are creating a new wall element so we need to input `false`. The wrapper method is available in Dynamo's RevitNodes.dll library. Add the library by right-clicking on References in the Solution Explorer and locate the library here: C:\Program Files\Autodesk\Revit 2020\AddIns\DynamoForRevit\Revit:



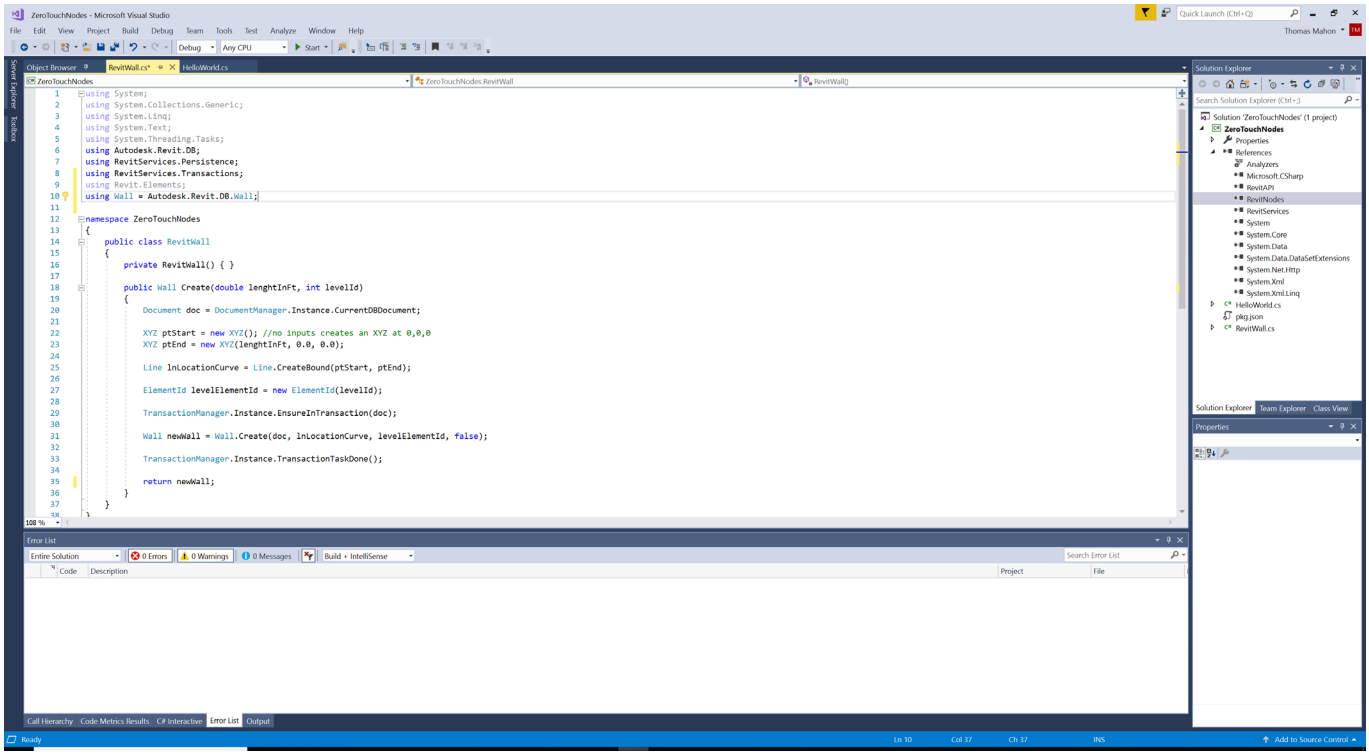
Add the following `using` directive followed by the target namespace Revit.Elements; to the top of the `RevitWall` document:



You will notice Visual Studio now displays a warning of an ambiguous reference between the Revit API `Wall` class and the `Wall` class in Dynamo's Revit.Elements library. We can resolve

this quickly using an alias in our using directives. Aliases provide convenient shorthand's for the developer to use in their code to avoid having to fully qualify classes or resolve issues such as ambiguous references. The simplest solution is to add a new alias to our using directives and qualify the intended `Wall` class – in our case, it's the `Wall` class from the Revit API – with the following syntax:

```
using Wall = Autodesk.Revit.DB.Wall;
```



We can now call ToDSType() on our newWall object. For convenience we can call it as the wall element is returned from the method, as shown below.

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public static Wall Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

            ElementId levelElementId = new ElementId(levelId);

            TransactionManager.Instance.EnsureInTransaction(doc);

            Wall newWall = Wall.Create(doc, lnLocationCurve, levelElementId, false);

            TransactionManager.Instance.TransactionTaskDone();
```

```
            return newWall.ToDSType(false);
        }
    }
}
```

ToDSType() also has another purpose; it creates a binding (known as 'Element binding') between the element created in our code, and its representation in Revit. It essentially creates a dependency between the node and the Revit element(s) it outputs. To illustrate this behaviour; if our method inputs change (length or level changes), the change modifies the same Wall instance. Without element binding, we would get a new wall every time the node's inputs change, or if the Dynamo file is closed/re-opened and run! It is therefore essential to call ToDSType() on any Revit Element instantiated in your code.

Tip: Visit https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration for detailed information of Dynamo's wrapper classes and geometry conversion utilities.

Update the Methods Return Type

ToDSType() returns an Element from Dynamo's Revit.Elements.`Element` class. However, our method currently returns a `Wall` class object from the Revit API and you will see a warning in Visual Studio that it cannot convert between the two types. The final step then is to update our methods return type. To avoid any ambiguous references with the `Element` class in the Revit API, qualify the target class:

```
namespace ZeroTouchNodes
{
    public class RevitWall
    {
        private RevitWall() { }

        public static Revit.Elements.Element Create(double lenghtInFt, int levelId)
        {
            Document doc = DocumentManager.Instance.CurrentDBDocument;

            XYZ ptStart = new XYZ(); //no inputs creates an XYZ at 0,0,0
            XYZ ptEnd = new XYZ(lenghtInFt, 0.0, 0.0);

            Line lnLocationCurve = Line.CreateBound(ptStart, ptEnd);

            ElementId levelElementId = new ElementId(levelId);

            TransactionManager.Instance.EnsureInTransaction(doc);

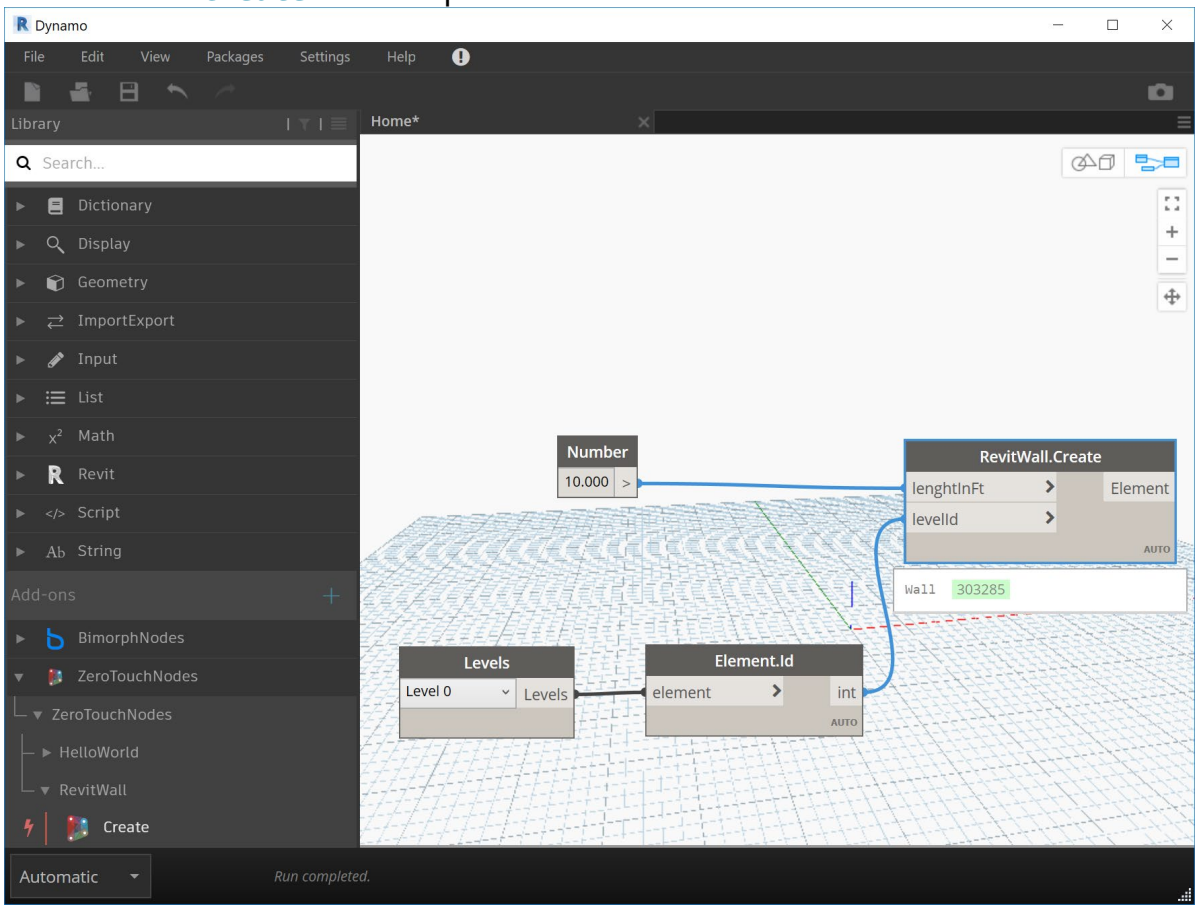
            Wall newWall = Wall.Create(doc, lnLocationCurve, levelElementId, false);

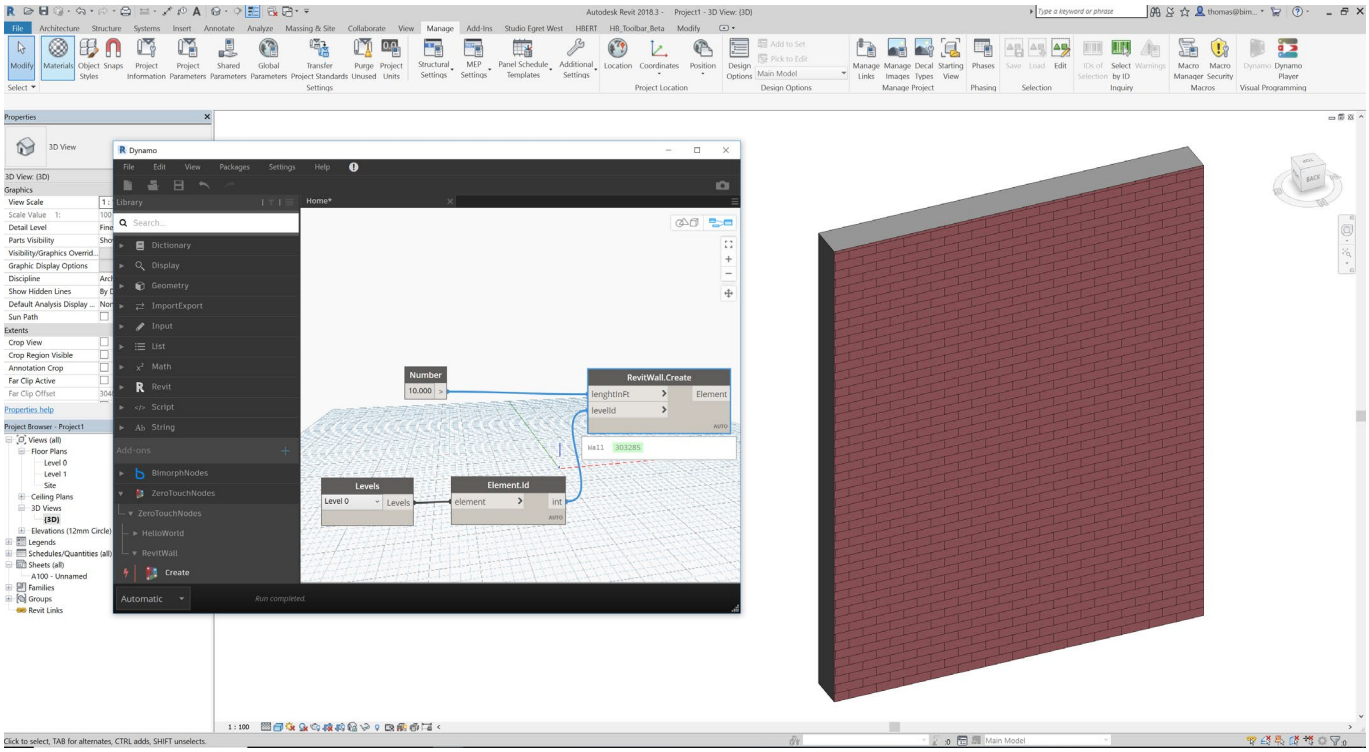
            TransactionManager.Instance.TransactionTaskDone();

            return newWall.ToDSType(false);
        }
    }
}
```

Compile and Test the Final Solution

We can now test the final solution and check if our ode is working as expected. If you have completed the exercise successfully, you should see a new wall appear in Revit after satisfying the RevitWall.`Create` node's inputs:





6. DO YOU WANT TO LEARN MORE?

Book a Bimorph 2-day Zero Touch Workshop

If you've enjoyed this lab and would like to deepen your understanding of C#, the Revit API and custom node development we provide a range of Dynamo Python and C# workshops based in London or at your site if booking in groups.

Gain expert knowledge on Dynamo's Revit wrapper class, element binding, wrapping and unwrapping, inheritance, garbage collection, interoperability techniques, data management, custom package creation, its ProtoGeometry, RevitNodes + geometry conversion libraries, in-depth C# skill development, XML documentation and more.

To enquire and book, visit: https://bimorph.com/dynamo-bim-workshops/