SD225402

Cross-Product and .NET App Development

Ben de Vries
Aurecon

Learning Objectives

- Learn how to set up visual studio projects to target multiple versions of Autodesk products
- Learn how to create and deploy a bundle
- Learn how to use C# language features
- Learn how code more efficiently

Description

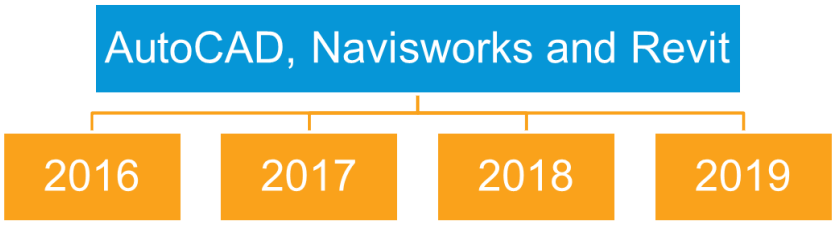
Learn how to set up and configure your visual studio projects to target multiple .NET platforms and versions of AutoCAD software, Navisworks software, and Revit software, as well as explore some tips and tricks for coding and deployments of WPF (Windows Presentation Foundation) applications add-ins.

Speaker(s)

I have been working in the engineering field for 22 years. With a background in bridge drafting, I've always been interested in software and anything technical. I now find myself a full-time developer for the past 15 years working across the entire Aurecon business developing applications and automation in C# across all the Autodesk product range.

Introduction

This class will demonstrate how to create a Cross Product application in WPF for AutoCAD, Navisworks and Revit targeting versions 2016 to 2019. It will present how to setup visual studio to target these versions, show how to code more efficiently across these versions and create a deployable *.bundle file.



Cross Product Development

One of the challenges to creating .NET applications for AutoCAD, Navisworks and Revit is that you must program against the application programming interface (API). These API's work great when you are only targeting one platform e.g. AutoCAD but when you have an application that can span across different platforms, problems can occur such as .NET versions and tightly bound\coupled code.

The main objective to creating a cross product application is a good separation of concerns. What this means it that you want to separate out all aspects of your code into distinct sections, such that each section addresses a separate concern.

Design Patterns

Application design patterns can be used to help with this separation of concerns and in the supplied demo (AwesomeAppIdea) a numbers of design patterns are used, these are listed below

- IoC – Inversion of Control \ Dependency Injection
- MVVM – Model-view-viewmodel

Dependency Injection

Dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it.

https://en.wikipedia.org/wiki/Dependency_injection

MVVM

MVVM facilitates a separation of development of the graphical user interface – be it via a markup language or GUI code – from development of the business logic or back-end logic (the data model). The view model of MVVM is a value converter, meaning the view

model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the view model is more model than view, and handles most if not all the view's display logic. The view model may implement a mediator pattern, organizing access to the back-end logic around the set of use cases supported by the view.
https://en.wikipedia.org/wiki/Model–view–viewmodel

Frameworks

Additional API frameworks can be implemented into your applications to help with the above-mentioned design patterns.

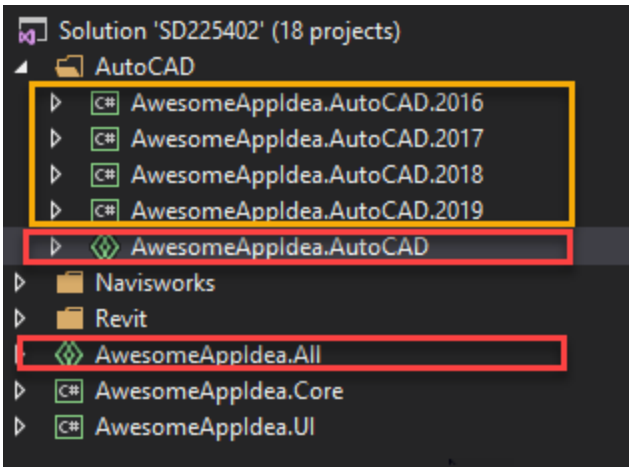
Prism Library

The demo app uses the Prism framework v6.3.0.

Prism is a framework for building loosely coupled, maintainable, and testable XAML applications in WPF. Prism provides an implementation of a collection of design patterns that are helpful in writing well-structured and maintainable XAML applications, including MVVM, dependency injection, commands, EventAggregator, and others.
https://prismlibrary.github.io/

Visual Studio 2017 Setup

The project setup required to target multiple products and .NET versions is a class library with a common shared project. The idea of the shared project is that you store all your code with in it and your main class libraries reference this shared project. Each class library has an entry point into each application using their respective api's.



Example showing 4 class libraries that target their respective frameworks and product api's in Yellow and their referenced shared projects in Red.

The shared project called AwesomeAppIdea.All is referenced into all the product class libraries. This contains the bootstrapper that wires everything up.

The class library called AwesomeAppIdea.Core contains all the core business logic across all products (Zero references to any product API.

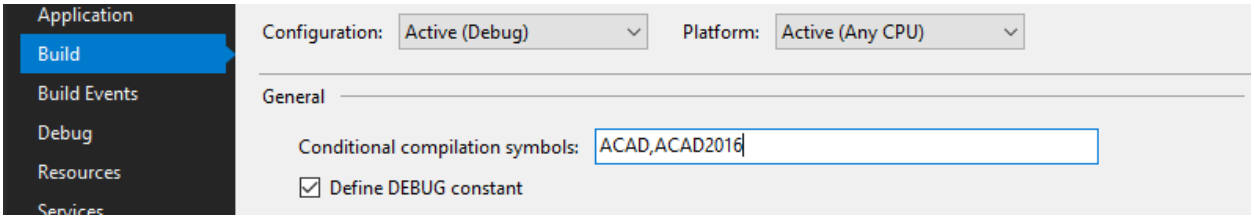
The class library called AwesomeAppIdea.UI contains all the version independent UI (View and View Models) for your application.

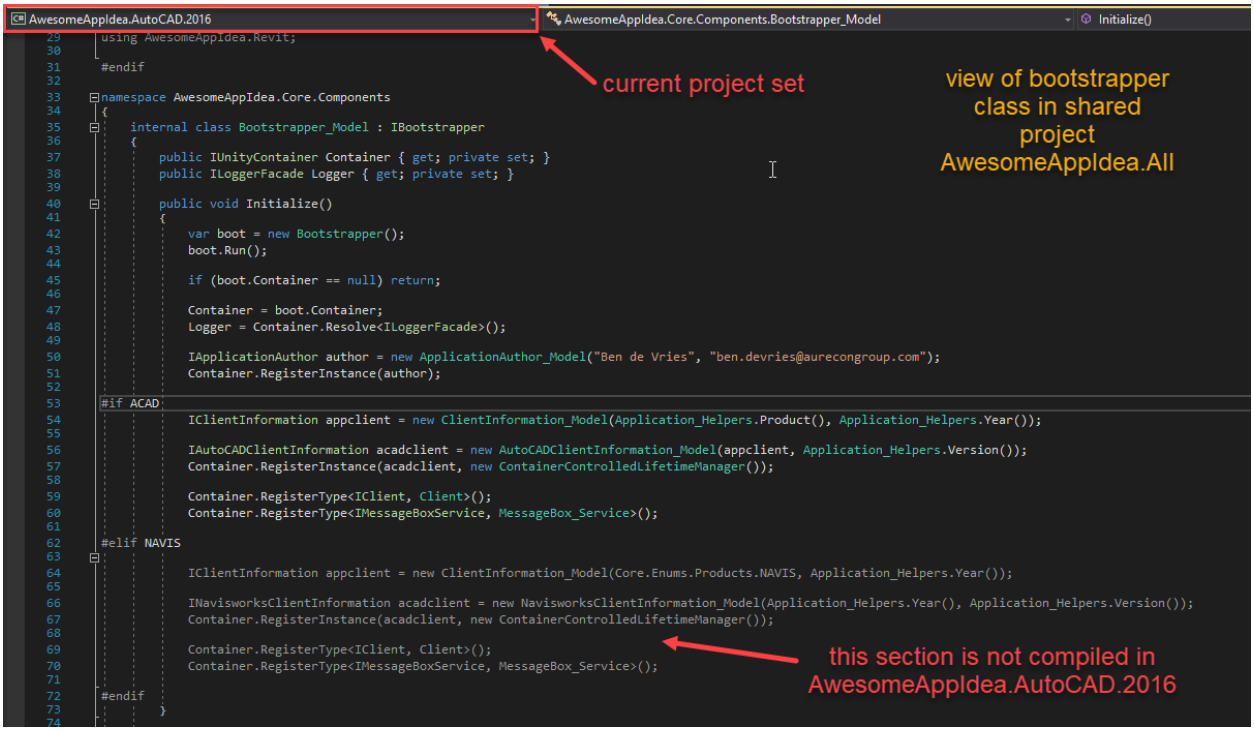
Making you application smart

What does this mean? Given that the main application\UI is in a separate dll and has zero references to any Autodesk application. The dll must understand what product it is being loaded into. Why? UI theming and any possible extra UI or services implementations.

Conditional compilation symbols

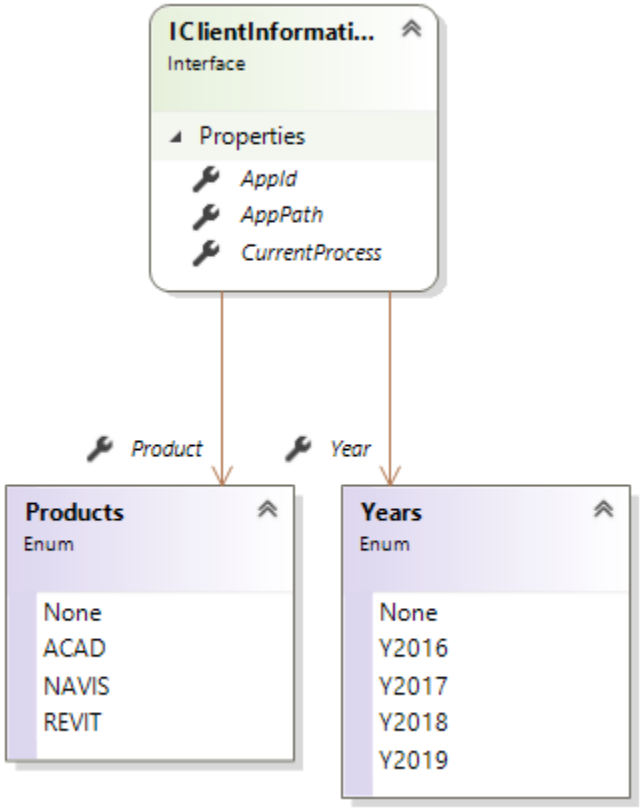
As part of each project you can set conditional compilation symbols. What this means is you can isolate sections of code to only compile if it meets the correct symbol.





IClientInformation Contract

The IClientInformation interface is a common contract that all products must implement on start-up. When a product hits its entry point, the client information is recorded and registered into the container.



AwesomeAppIdea and the Prism framework

What is a Bootstrapper?

A bootstrapper is a class that is responsible for the initialization of an application built using the Prism Library. By using a bootstrapper, you have more control of how the Prism Library components are wired up to your application.

The Prism Library includes a default abstract Bootstrapper base class that can be specialized for use with any container. Many of the methods on the bootstrapper classes are virtual methods. You can override these methods as appropriate in your own custom bootstrapper implementation.

In the case of the AweseomeAppIdea, a custom bootstrapper is used to reconfigure the default logger and viewmodellocator.

Containers and Dependency Inject

Dependency injection containers reduce the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime based on the configuration of the container. During the objects creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first. In some cases, the container itself is resolved as a dependency.

In the case of the AwesomeAppIdea, the standard unity container is used but you can also reconfigure the bootstrapper to implement other containers such as Castle Windsor, StructureMap, and Spring.NET and MEF.

AwesomeAppIdea and Product Entry points

Entry Points and Dependencies

The entry points into AutoCAD, Navisworks and Revit can be made in a few different ways.

AutoCAD

- IExtensionApplication
- CommandMethod

Navisworks

- EventWatcherPlugin
- DockPanePlugin

Revit

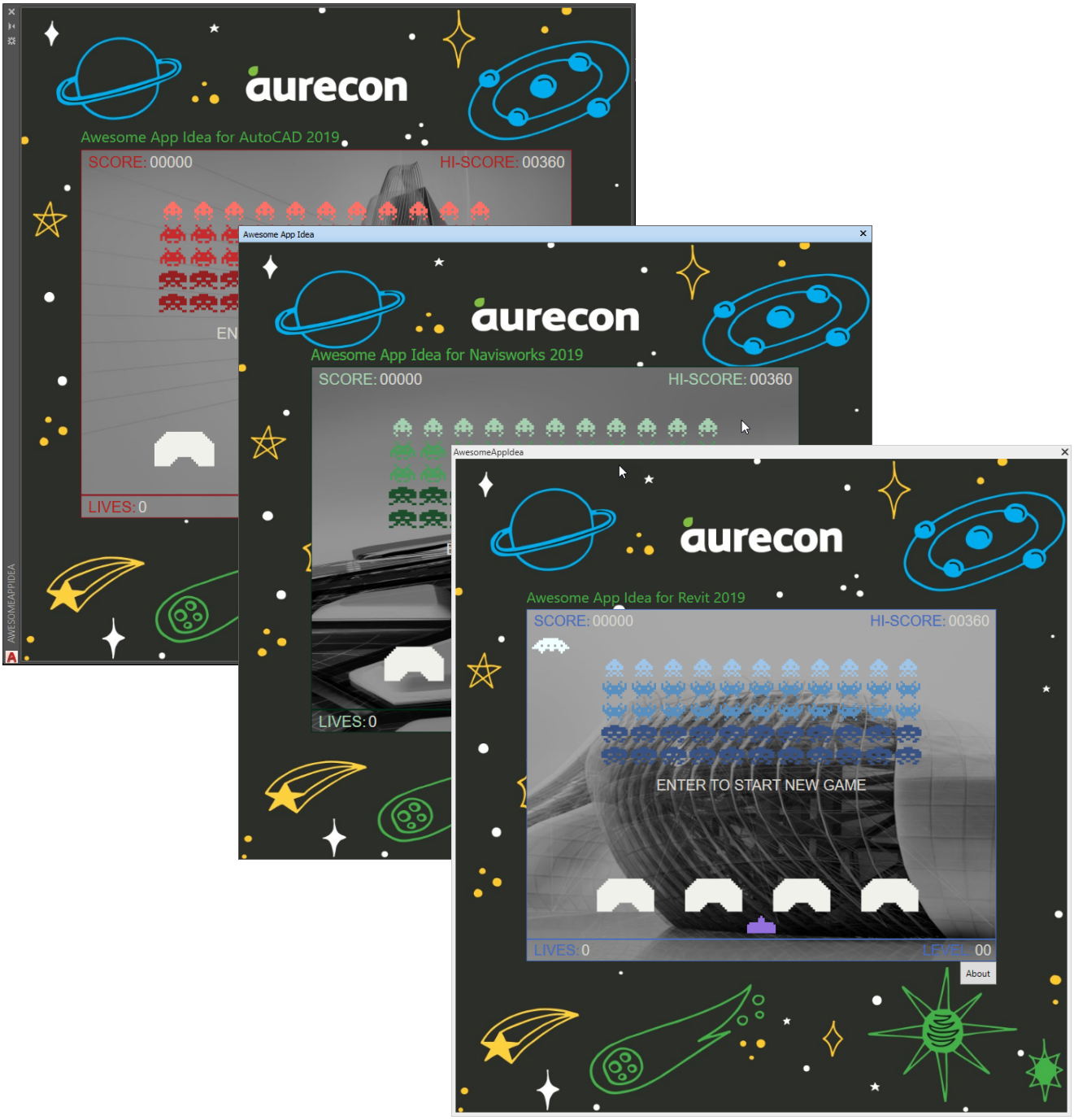
- IExternalApplication
- IExternalCommand

Whichever entry point you decided to use the first thing you need to do is make sure you load any dependencies into the current AppDomain. As this application uses some extra WPF functions the `System.Windows.Interactivity.dll` is required prior to loading any UI. A simple static method can be used.

```
public static void LoadDependancies()
{
   var currentdirectory = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
   var dependancy = new FileInfo(Path.Combine(currentdirectory, "System.Windows.Interactivity.dll"));
   if (dependancy.Exists)
   {
      Assembly.Load(AssemblyName.GetAssemblyName(dependancy.FullName));
   };
}
```

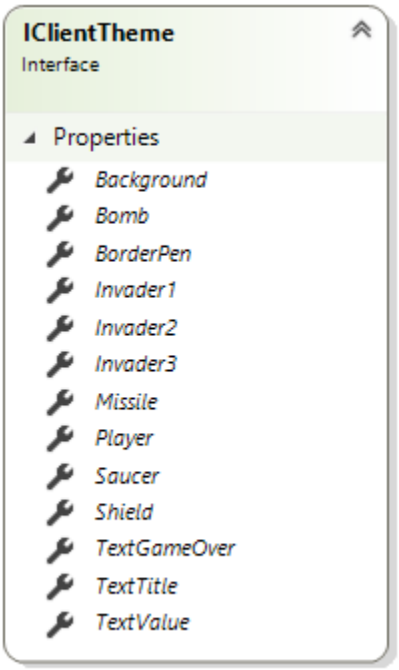
AwesomeAppIdea User Interface

So here we get to the fun part, the UI. The UI I chose for this demo is somewhat irrelevant to what you would normally see in these applications. What is does show, is that if everything is setup correctly you can implement whatever you want. Below are 3 images of the same AwesomeAppIdea loaded in AutoCAD, Navisworks and Revit, branded per product and yes this is a working version of space invaders!

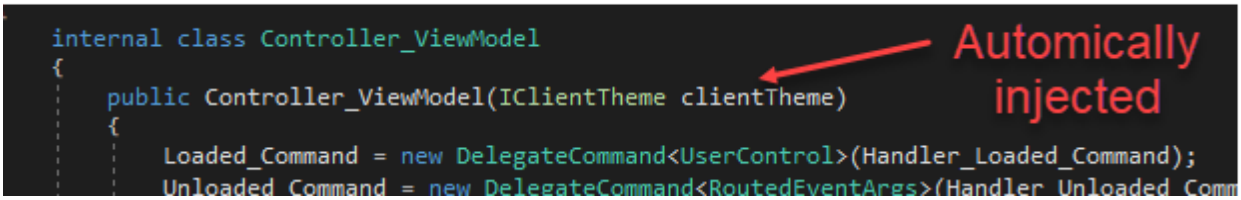


How does all this work?

As we have made our AwesomeAppIdea smart, it now understands what application it is loaded into. Now theming can be applied by implementing the IClientTheme interface. Each product version can implement its own theme or use a common theme.



This is registered with the container during startup and is injected into the ViewModel of the UI when it is required. See image below.



Commanding and Messaging

The about button command property is data bound to a DelegateCommand property in the view model. See below.

The DelegateCommand is then wired up to is a IMessageBoxService. The reason for doing this is that our application UI is generic and has no references to any product. It will communicate

with our products using the IMessageBoxService. Below is the code for the Main_ViewModel and you can see that the IApplicationInfo and IMessageService are injected automatically.

```
using AwesomeAppIdea.Core.Contracts;
using Prism.Commands;
using Prism.Mvvm;
using System;

namespace AwesomeAppIdea.UI.ViewModel
{
    public class Main_ViewModel : BindableBase
    {
        public Main_ViewModel(IApplicationInfo applicationInfo, IMessageBoxService
dialogService)
        {
            _messageboxService = dialogService;
            Info = applicationInfo;
            About_Command = new DelegateCommand(Handler_About_Command);
        }

        private void Handler_About_Command()
        {
            var messagecontent = $"Developed by:
{Info.Author.Name}{Environment.NewLine}Email Address:
{Info.Author.EmailAddress}{Environment.NewLine}Version: {Info.Version}";
            _messageboxService.Show("About", messagecontent);
        }

        private readonly IMessageBoxService _messageboxService;

        public IApplicationInfo Info { get; }

        public DelegateCommand About_Command { get; }
    }
}
```

Each product will implement an IMessageBoxService

AwesomeAppIdea.bundle

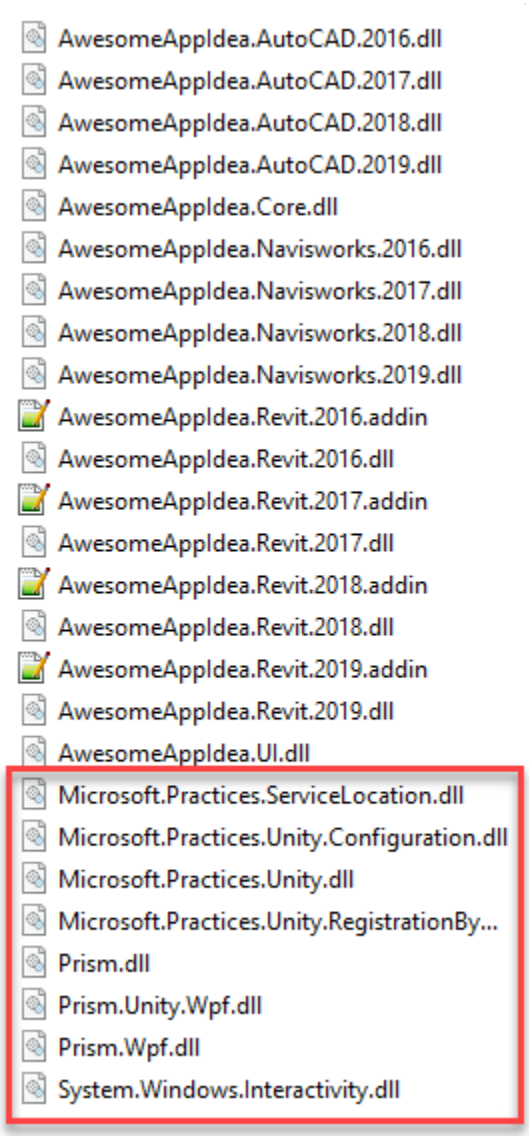
To help with debugging you can include a MSBuild script that will copy the output files to a bundle for testing. The script (AwesomeAppIdea.AfterBuild.Targets) looks like this

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
        <PropertyGroup>
        <DebugBundle>C:\ProgramData\Autodesk\ApplicationPlugins\AwesomeAppIdeaDEBU
G.Bundle\Contents</DebugBundle>
        </PropertyGroup>
        <Target Name="AfterBuild" Condition=" '$(Configuration)'=='Debug' ">
                <Copy SourceFiles ="$(TargetDir)$(TargetFileName)"
DestinationFolder="$(DebugBundle)" ContinueOnError="true"/>
        </Target>
</Project>
```

You can wire this up in your proj files with the following

```
    <CustomAfterMicrosoftCommonTargets>$(MSBuildProjectDirectory)\..\MS
Build\AwesomeAppIdea.AfterBuild.Targets</CustomAfterMicrosoftCommonTargets>
```

Every time you make a change to you code you can build and run AutoCAD, Navisworks or Revit. The folder it copied the assemblies into is *C:\ProgramData\Autodesk\ApplicationPlugins\AwesomeAppIdeaDEBUG.bundle\Contents*



The box highlighted in RED are the Prism assemblies that will have to be copied into your bundle contents folder manually.

The Revit add-in files need to be created manually, although you could automate this if you want to.

Github Repo

For all the code associated with this demo please follow the links below

https://github.com/devriesb13/AwesomeAppIdea.git

https://github.com/devriesb13/AwesomeAppIdea

Conclusion

Now that the core projects are setup and working, you can now explore the possibilities of expansion.

- Connect to Forge
- Connect to SharePoint or other cloud services
- Data and Analytics

The possibilities are endless…

Finally, thank you for attending this session, I hope you found this enjoyable and valuable.