Before We Start (Read This Once)

A quick word on what this guide actually is.

This guide is built around observation, not memorization.

You won't be expected to recall everything after one pass. That's not how learning works, and it's definitely not how Revit API works.
Revit API starts with something small. Something manageable. This is that starting point.

Do I need to know advanced C# before using Revit API?

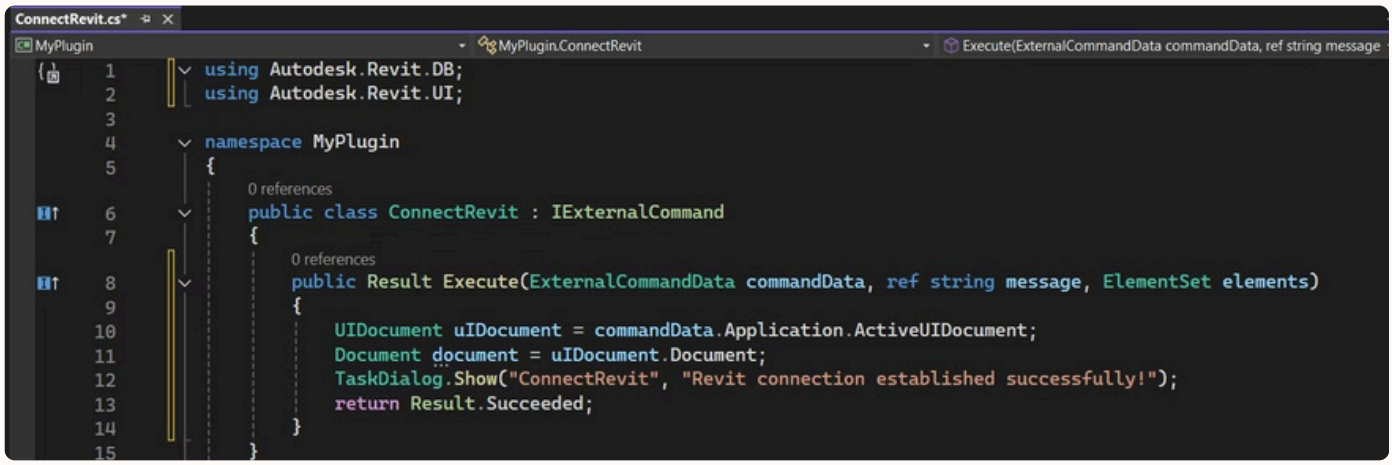
Not at all.

You need just enough C# to understand what you're looking at. The depth comes later, through repeated use and application.

There are no screenshots on this page intentionally. We're setting the frame first.

The Smallest Working Revit Add-In

This is a complete,functional Revit command. Focus on the overall structure and the minimal amount of code.



> Smaller than you expected? That's normal.

Many expect Revit API to require hundreds of lines, but as you see, that's not the case.

If This Looks New, That's Perfectly Fine

> "What if I don't understand half of this code?"

This initial unfamiliarity is normal. Understanding comes through practical use, not just analysis.

The syntax may feel unfamiliar, but this discomfort is temporary.

We'll focus only on what matters—nothing extra.

CHECKPOINT

You're not falling behind. You're at the exact point where everyone else started. The difference is you're about to see the pattern, and once you see it, everything else becomes predictable.

What Is IExternalCommand? (No Theory Needed)

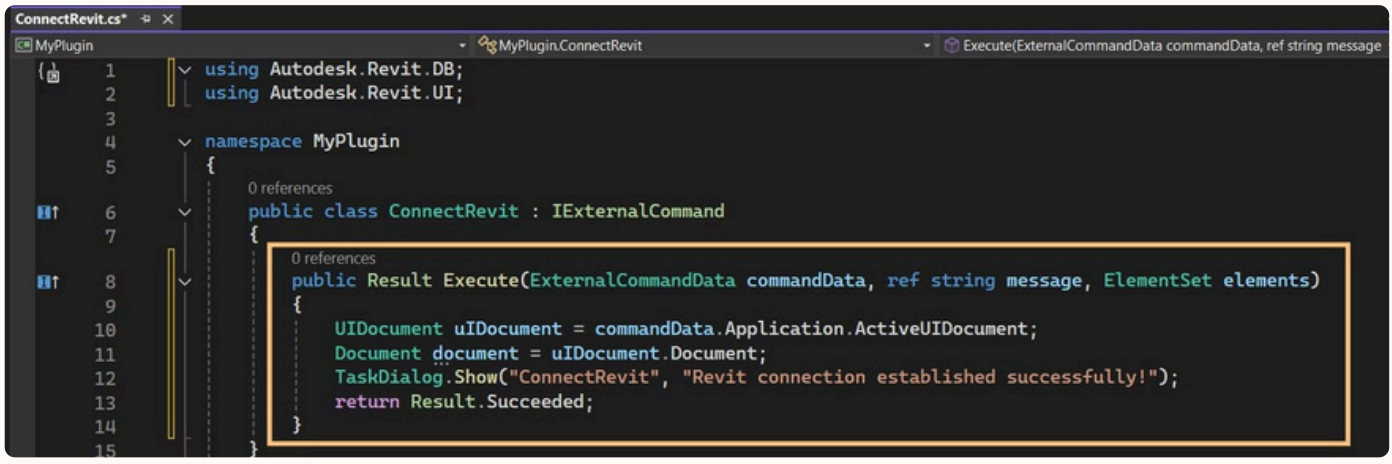
> Why does this interface exist?

Revit uses this interface to identify runnable commands. It's a marker that tells Revit: "This can be run." When you build a command, you're essentially marking your code as executable with **IExternalCommand**.

> Note: We're skipping general interface definitions and SOLID principles. They aren't necessary to get started.

The One Method Revit Actually Cares About

> Which part of this code actually runs?



Only the `Execute()` method runs. Everything else provides structural support, but the core logic, the "actual work," lives inside `Execute()`. Understanding this method means understanding the command.

Why Do We Return Result.Succeeded?

"What happens if this isn't returned?"

Revit expects a status signal:

Did the command complete successfully or not?

1

At the conclusion of your Execute() method, you return one of three values: **Result.Succeeded**

2

Result.Failed

3

or **Result.Cancelled**.

Consider it a status report back to Revit.

Revit checks this return value to determine whether everything proceeded as expected. Success? Return Result.Succeeded. Failure? Return Result.Failed.

We're keeping error handling simple for now. This is about confirming successful execution, not managing edge cases.

How Do We Find Elements in Revit?

"How does Revit API locate walls, pipes, or ducts?"

Through FilteredElementCollector. Think of it as a search mechanism—nothing more complex.

Say you need to find all walls in your project. Or all mechanical equipment. Or all doors on a specific level.
FilteredElementCollector is how you instruct Revit: "Locate these elements and return them." You'll use this repeatedly; it's one of the most frequently used tools in Revit API development.

Search for Walls

Find all wall elements in the active document.

Search for Pipes

Locate specific pipe elements based on criteria.

Search for Ducts

Identify duct systems or individual duct components.

Example: Collecting Walls (or Pipes / Ducts)

Here's a practical application. This is how you collect all walls in the active document:

```
namespace MyPlugin
{
    0 references
    public class ConnectRevit : IExternalCommand
    {
        0 references
        public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
        {
            UIDocument uIDocument = commandData.Application.ActiveUIDocument;
            Document document = uIDocument.Document;
            List<Wall> allWalls = new FilteredElementCollector(document)
                .OfClass(typeof(Wall)).Cast<Wall>().ToList();
            TaskDialog.Show("ConnectRevit", "Revit connection established successfully!");
            return Result.Succeeded;
        }
    }
}
```

> "Do I need to master all filter types right now?"
> **Not at all.** Focus on understanding this example; deeper knowledge will come with repeated use and applic ation.

Understanding one example reveals the underlying pattern. Once you grasp how to collect one element type, the same approach applies to anything—doors, windows, ducts, pipes, families, views, schedules.

How Do We Access the Current Revit Model?

"How does thecode know which projectis open?"

Revit passes access to the active document via **ExternalCommand** when **Execute()** is called.

The Document Reference

Think of **Document** as a reference to the file currently open in Revit.

This 'Document' reference is your entry point to all model elements like walls, floors, systems, and views.

```
namespace MyPlugin
{
    0 references
    public class ConnectRevit : IExternalCommand
    {
        0 references
        public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
        {
            UIDocument uIDocument = commandData.Application.ActiveUIDocument;
            Document document = uIDocument.Document;
            List<Wall> allWalls = new FilteredElementCollector(document)
                .OfClass(typeof(Wall)).Cast<Wall>().ToList();
            TaskDialog.Show("ConnectRevit", "Revit connection established successfully!");
            return Result.Succeeded;
        }
    }
}
```

Let's Prove This Command Actually Works

"How do I confirm the add-in ran successfully?"

Confirm add-in execution in Revit using the **TaskDialog** to display a message.

Your First Interaction

```
ConnectRevit.cs*
MyPlugin                    MyPlugin.ConnectRevit                    Execute(ExternalCommandData commandData, ref string message
 1  using Autodesk.Revit.DB;
 2  using Autodesk.Revit.UI;
 3
 4  namespace MyPlugin
 5  {
        0 references
 6      public class ConnectRevit : IExternalCommand
 7      {
            0 references
 8          public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
 9          {
10              UIDocument uIDocument = commandData.Application.ActiveUIDocument;
11              Document document = uIDocument.Document;
12              TaskDialog.Show("ConnectRevit", "Revit connection established successfully!");
13              return Result.Succeeded;
14          }
15      }
```

This is your first programmatic interaction with Revit.

This dialog provides visual confirmation that your code executed successfully.

Key Interfaces & Core Methods

Key Interfaces

IEXTERNALCOMMAND

The interface Revit uses to identify executable commands. When you implement this, Revit knows your code can be triggered as a command.

Core Method

EXECUTE()

The only method that actually runs when your command is triggered. This is where your logic lives.

Return Values

- Result.Succeeded — Command completed successfully
- Result.Failed — Command encountered an error
- Result.Cancelled — User cancelled the operation

Essential Classes

Document

Reference to the open Revit file

UIDocument

Access to the user interface and active view

FilteredElementCollector

Tool for finding elements in the model

TaskDialog

Simple message boxes for user feedback

Watch the Video in This Module

To understand how to create your first plugin and see what it looks like in action.

The video walkthrough demonstrates the complete process—from creating a new project to seeing your add-in run inside Revit.

WHAT YOU'LL SEE IN THE VIDEO

01

Setting up a new Revit API project in Visual Studio

02

Writing the IExternalCommand implementation

03

Creating the .addin manifest file

04

Building and deploying to Revit

05

Testing the command and seeing TaskDialog in action

Seeing it work brings everything together.

Theory becomes real when you watch the code execute in Revit.