Creating a professional Revit add-in for the Autodesk App Store requires a structured approach that goes beyond just writing code. You'll need to focus on a robust development process, adherence to Autodesk's specific requirements, and a polished user experience.

**Development Process**

Start with a solid foundation by using **C#** and **Microsoft Visual Studio**, as this is the standard for Revit API development.1 The core of your add-in will be the **IExternalCommand** interface for commands that interact with the Revit model and the **IExternalApplication** interface for applications that run in the background.

* **Project Structure**: Organize your code into a well-defined project.2 A common approach is to separate your main logic from the projects that do the compiling, allowing you to target multiple Revit versions with the same core code.3 This makes it much easier to support multiple Revit releases without code duplication.
* **Best Practices**: Incorporate robust error handling using try-catch blocks to prevent unexpected crashes.4 Use **transactions** to manage changes to the Revit database, ensuring that operations are either fully completed or completely rolled back.5
* **User Interface**: Design a user-friendly interface. Autodesk requires a **ribbon button** to access your main command.6 If your add-in has multiple functions, a custom ribbon tab with your add-in's name is recommended.7 The UI should be intuitive and guide the user.
* **Documentation and Help**: Create a professional-looking HTML quick-start page that will be included with your app's download.8 This page should clearly explain the add-in's purpose and how to use it. You may also want to include more detailed documentation and video tutorials.

**App Store Submission Requirements**

The Autodesk App Store has specific technical and content requirements for Revit add-ins.9 Failing to meet these can result in your app being rejected.

* **Add-in Manifest**: You must use an **.addin** manifest file as the loading mechanism.10 This XML file tells Revit which DLLs to load and where to find your command's class.11
* **Installer**: The Autodesk App Store will create an installer for you based on the information you provide. You must organize your files in a **.bundle** folder with a **PackageContents.xml** file.12 If you have custom installation needs, you can provide Windows Installer Merge Modules (.msm) files.
* **Stability and Behavior**: Your add-in must be stable and not interfere with standard Revit functionality or other add-ins.13 This includes preventing data loss and not blocking other add-ins from functioning.
* **Content and Marketing**: Prepare high-quality marketing materials, including a descriptive title (under 128 characters), a compelling description, and a thumbnail image and screenshots. Your product's value proposition should be clear from the start.
* **Legal and Licensing**: You can include your own **End-User License Agreement (EULA)**.14 It's also a good practice to code-sign your DLLs. While not always a strict requirement for free apps, it's highly recommended as it provides an extra layer of trust and security for users and ensures Revit only displays one security pop-up for all your signed add-ins.

**The first step** to building your professional Revit add-in is to **set up your development environment**. This involves installing the necessary software and tools that will allow you to write, compile, and test your code.

Here's a breakdown of what you'll need:

1. **Autodesk Revit**: You'll need a licensed version of Autodesk Revit installed on your machine. The version you choose will influence the Revit API version you'll be developing against.
2. **Microsoft Visual Studio**: This is the primary Integrated Development Environment (IDE) for C# development.
   * **Community Edition**: The free Community edition of Visual Studio is more than sufficient for developing Revit add-ins.
   * **.NET Development Workload**: During Visual Studio installation, ensure you select the ".NET desktop development" workload.
3. **Revit API SDK (Software Development Kit)**: Although much of the API documentation is available online, the SDK provides valuable resources, including example code and the necessary DLLs. You can typically find this within your Revit installation directory or download it from the Autodesk Developer Network (ADN).
4. **Revit API Documentation**: Familiarize yourself with the Revit API documentation. Autodesk provides comprehensive online resources and help files that are crucial for understanding the available commands and data structures.
5. **Referencing Revit DLLs**: In your Visual Studio project, you'll need to add references to key Revit API DLLs, primarily RevitAPI.dll and RevitAPIUI.dll. These are usually located in the Revit installation folder.

Once these are set up, you can begin creating a new **Class Library** project in Visual Studio, which will serve as the foundation for your add-in.

**1. Install the Revit API**

You can't create an add-in without the tools to talk to Revit. The Revit API is a set of libraries that allow your code to interact with Revit's data model.

* Go to the Autodesk developer website or find a reliable source to download the **Revit SDK (Software Development Kit)** for the specific Revit version you want to target (e.g., Revit 2024, Revit 2025). The SDK contains valuable samples and documentation.
* Once you've downloaded the SDK, you'll find the **Revit API DLLs**. These are the crucial files you'll need to reference in your project. The most important ones are:
  + RevitAPI.dll
  + RevitAPIUI.dll

**2. Create Your Project in Visual Studio**

Now, you'll set up the project in Visual Studio. We'll create a Class Library project, which is the standard starting point for Revit add-ins.

1. Open Visual Studio.
2. Select **"Create a new project."**
3. In the search bar, type **"Class Library."** Choose the **"Class Library"** template for **C#** and click **"Next."** Make sure you select the one for a .NET Framework project, not a .NET Core or .NET 5/6/7 project, as the Revit API is not compatible with those.
4. In the configuration window, name your project **"RevitAddins"** and choose a location on your computer.
5. Set the **.NET Framework version**. This is critical. You must match the .NET Framework version used by your target Revit version. For example, Revit 2022 and later require **.NET Framework 4.8**.
6. Click **"Create."**

**3. Add References to the Revit API DLLs**

With your project created, you need to tell it where to find the Revit API.

1. In the **Solution Explorer** on the right side of Visual Studio, right-click on your **RevitAddins** project.
2. Select **"Add" > "Reference."**
3. In the Reference Manager window, click the **"Browse"** button at the bottom.
4. Navigate to the directory where you found the Revit API DLLs.
   * A common location for these DLLs is C:\Program Files\Autodesk\Revit [Version]\.
5. Select **RevitAPI.dll** and **RevitAPIUI.dll** and click **"Add."**
6. Make sure the two DLLs are checked in the Reference Manager window and click **"OK."**

Great! Knowing you're working with Revit 2023 is crucial for setting the correct .NET Framework version.

Here's what you need to do next, building upon the foundation you've already laid:

**4. Set the Target .NET Framework Version (Crucial for Revit 2023)**

Revit 2023 specifically requires **.NET Framework 4.8**. If you didn't select this during project creation, or want to ensure it's correct:

1. In the **Solution Explorer**, right-click on your **RevitAddins** project.
2. Select **"Properties."**
3. In the Project Properties window, go to the **"Application"** tab.
4. Under **"Target framework,"** make sure **".NET Framework 4.8"** is selected. If not, choose it from the dropdown.
5. Save your changes (File > Save All).

**5. Create Your First Revit Command**

Now, let's get some actual code in there. The simplest way to start is by creating a command that will appear on the Revit ribbon. This involves implementing the IExternalCommand interface.

1. In the **Solution Explorer**, right-click on your **RevitAddins** project.
2. Select **"Add" > "New Item."**
3. Choose **"Class"** and name it something descriptive, like HelloWorldCommand.cs. Click **"Add."**
4. Replace the default code in HelloWorldCommand.cs with the following:

C#

using Autodesk.Revit.Attributes;

using Autodesk.Revit.DB;

using Autodesk.Revit.UI;

namespace RevitAddins // This should match your project name

{

[Transaction(TransactionMode.Manual)] // Defines how transactions are handled

[Regeneration(RegenerationOption.Manual)] // Defines when the model is regenerated

public class HelloWorldCommand : IExternalCommand

{

public Result Execute(

ExternalCommandData commandData,

ref string message,

ElementSet elements)

{

// This is where your command's logic will go.

// For now, we'll just show a simple message box.

TaskDialog.Show("Hello World", "Welcome to your first Revit Add-in!");

// Return success if the command executed without errors

return Result.Succeeded;

}

}

}

**Explanation of the attributes:**

* + **[Transaction(TransactionMode.Manual)]**: This tells Revit that your command will manage its own transactions. For simple read-only operations or commands that don't modify the model, Manual is fine. For commands that *do* change the model, you'll need to open and commit transactions within your Execute method.
  + **[Regeneration(RegenerationOption.Manual)]**: This indicates that Revit doesn't need to automatically regenerate the model after your command finishes. Manual is generally preferred for performance.

**Explanation of the Execute method:**

* + **commandData**: This object provides access to the current Revit application, document, and other useful information.
  + **message**: A string you can use to pass back an error message if your command fails.
  + **elements**: An ElementSet you can use to pass back specific Revit elements that your command might have interacted with.
  + **Result.Succeeded**: This tells Revit that your command completed successfully. Other possible return values are Result.Cancelled and Result.Failed.

**6. Build Your Project**

Before you can test your add-in, you need to compile it.

1. In Visual Studio, go to **"Build" > "Build Solution"** (or press F6).
2. Check the **Output** window at the bottom of Visual Studio for any errors. If it says "Build succeeded," you're good to go!

You've now created a basic Revit add-in that, once loaded into Revit, will display a "Hello World" message. The next step will be to create the necessary files to load this add-in into Revit.

Let me know when you're ready for the next step, which involves creating the .addin manifest file and loading it into Revit!

Excellent! With Revit 2023 and the correct .NET Framework version (.NET Framework 4.8) established, you're on a solid track. You've successfully created the RevitAddins project and implemented your first HelloWorldCommand.

The next logical step is to make this command accessible within Revit. To do this, you need to:

1. **Create an Add-in Manifest File (.addin)**: This file tells Revit about your add-in, including its name, the assembly (your DLL) it should load, and the command class to execute.
2. **Configure your Project to Output the Manifest**: Ensure the .addin file is copied alongside your DLL when you build.
3. **Install the Add-in into Revit**: Place the DLL and the .addin file in the correct Revit Add-Ins folder.

Let's dive into that.

**7. Create the Add-in Manifest File (.addin)**

This is a crucial XML file that Revit reads to discover and load your add-in.

1. **In Visual Studio**:
   * Right-click on your **RevitAddins** project in the Solution Explorer.
   * Select **"Add" > "New Item..."**
   * Choose **"Text File"** from the list.
   * Name this file **RevitAddins.addin**. **Make sure it has the .addin extension.**
   * Click **"Add."**
2. **Edit the .addin File**: Open the RevitAddins.addin file and paste the following XML content into it. **You'll need to adjust the Assembly path to point to where your compiled DLL will be located.**

XML

<?xml version="1.0" encoding="utf-8"?>

<ExternalApplication>

<Name>RevitAddins Application</Name>

<Assembly>C:\Path\To\Your\RevitAddins\bin\Debug\RevitAddins.dll</Assembly>

<ClientId>YOUR\_UNIQUE\_CLIENT\_ID\_HERE</ClientId>

<FullClassName>RevitAddins.HelloWorldCommand</FullClassName>

<VendorId>YOUR\_VENDOR\_ID\_HERE</VendorId> <VendorDescription>Your Company Name</VendorDescription> <Version>1.0.0</Version> </ExternalApplication>

**Key elements and what to do:**

* + **<Name>**: This is the name that will appear in the Revit ribbon's "Add-Ins" tab. You can keep it as "RevitAddins Application" or change it.
  + **<Assembly>**: **This is the most important path.** It *must* point to the full path of your add-in's DLL file (RevitAddins.dll) that will be generated when you build your project.
    - For **Debug** builds, this will typically be something like: C:\YourSolutionFolder\RevitAddins\bin\Debug\RevitAddins.dll
    - For **Release** builds, it would be: C:\YourSolutionFolder\RevitAddins\bin\Release\RevitAddins.dll
    - **For now, use the Debug path. We'll automate this better later.**
  + **<ClientId>**: This needs to be a **Globally Unique Identifier (GUID)**. Each add-in *must* have a unique GUID.
    - To generate one: In Visual Studio, go to **Tools > Create GUID**. Select **"Registry Format"** and click **"New GUID"**. Copy the generated GUID (it will look something like {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}).
    - **Remove the curly braces {}** when pasting it into the ClientId tag.
  + **<FullClassName>**: This is the fully qualified name of your command class: YourNamespace.YourClassName. In your case, it's RevitAddins.HelloWorldCommand.
  + **<VendorId>**, **<VendorDescription>**, **<Version>**: These are optional for local testing but become important if you plan to publish to the Autodesk App Store. For now, you can fill them with placeholders or leave them if you prefer.

**8. Configure Project Properties to Copy Output Files**

You need to ensure that both your RevitAddins.dll and your RevitAddins.addin file are placed in the same output directory when you build, making it easy to deploy.

1. **Copy the DLL**:
   * In the **Solution Explorer**, right-click on your **RevitAddins** project and select **"Properties."**
   * Go to the **"Build"** tab.
   * Under **"Output path,"** ensure it's set to bin\Debug\ (or bin\Release\ if you plan to build a release version).
   * Crucially, find the references to RevitAPI.dll and RevitAPIUI.dll in the **"Dependencies"** (or **"References"**) section. Right-click on each and select **"Properties."** Set **"Copy Local"** to True. This ensures your project's DLL doesn't rely on finding these in the Revit installation path during deployment.
2. **Copy the .addin File**:
   * In the **Solution Explorer**, right-click on the RevitAddins.addin file you added.
   * Select **"Properties."**
   * Find the **"Copy to Output Directory"** property.
   * Change its value to **"Copy if newer"** or **"Copy always."** This will ensure the .addin file is placed in your bin\Debug\ folder alongside the RevitAddins.dll when you build.

**9. Build and Install Your Add-in**

Now, let's get your add-in into Revit.

1. **Build the Solution**: Go to **"Build" > "Build Solution"** (or press F6). If there are no errors, navigate to your project's output folder (e.g., C:\YourSolutionFolder\RevitAddins\bin\Debug\). You should find RevitAddins.dll and RevitAddins.addin here.
2. **Install into Revit**:
   * Locate the Revit Add-Ins Folder: The standard location for add-ins is:

C:\ProgramData\Autodesk\ApplicationPlugins\

Note: ProgramData is a hidden folder. You might need to enable "Show hidden items" in your File Explorer view.

* + Create a Folder for Your Add-in: Inside the ApplicationPlugins folder, create a new folder named RevitAddins. The path would look like:

C:\ProgramData\Autodesk\ApplicationPlugins\RevitAddins\

* + **Copy Your Files**: Copy *both* RevitAddins.dll and RevitAddins.addin from your Visual Studio output folder (e.g., bin\Debug) into this newly created RevitAddins folder in ProgramData.
  + **Important Note on .addin File Path**: When you placed the .addin file directly in the ApplicationPlugins\RevitAddins folder (alongside the DLL), Revit will find it. The <Assembly> path in your .addin file can often be simplified to just the DLL name (RevitAddins.dll) if it's in the same folder. For local development, manually updating the full path in the .addin file is common, but for deployment, relative paths or just the filename is preferred. For now, leave the full path as you set it up.

1. **Launch Revit 2023**: Start Revit.
2. **Check the Ribbon**: Look for the **"Add-Ins"** tab on the Revit ribbon. You should see a panel with the name you specified in your .addin file (e.g., "RevitAddins Application"), and within that panel, a button labeled "Hello World."
3. **Test Your Command**: Click the "Hello World" button. You should see your "Welcome to your first Revit Add-in!" message box appear.

If you see the message, congratulations! You've successfully created, built, and loaded your first Revit add-in.

The next logical steps would involve:

* Making the command modify the Revit model using transactions.
* Creating a more complex UI if needed.
* Preparing for distribution.

**Introduction**

This document provides a comprehensive guide for creating a professional Revit add-in. It covers the full development lifecycle, from project setup to a working, deployable application. The documentation also includes a detailed account of potential issues encountered during development and their resolutions.

**Phase 1: Foundational Setup (Sprint 1)**

This phase establishes the core structure of the add-in.

1. Project Structure:

A well-organized project is key to maintainability. The recommended structure is:

* src/: Main C# project folder.
  + Application/: Contains the IExternalApplication class.
  + Commands/: Contains all IExternalCommand classes.
  + Resources/: Stores images and other assets.
* Publish/: Holds final build and deployment files (.addin, .dll).

2. IExternalApplication Class:

The entry point for the add-in. The OnStartup method is used to create the ribbon UI.

C#

using System.Reflection;

using Autodesk.Revit.UI;

namespace RevitAddins

{

public class App : IExternalApplication

{

public Result OnStartup(UIControlledApplication a)

{

string tabName = "Your App Name";

a.CreateRibbonTab(tabName);

RibbonPanel panel = a.CreateRibbonPanel(tabName, "My Tools");

string assemblyPath = Assembly.GetExecutingAssembly().Location;

PushButtonData button1Data = new PushButtonData("Command1Id", "Hello\nWorld", assemblyPath, "RevitAddins.Commands.Command1");

panel.AddItem(button1Data);

return Result.Succeeded;

}

public Result OnShutdown(UIControlledApplication a)

{

return Result.Succeeded;

}

}

}

3. IExternalCommand Class:

The code that executes when a ribbon button is clicked.

C#

using Autodesk.Revit.DB;

using Autodesk.Revit.UI;

using Autodesk.Revit.Attributes;

namespace RevitAddins.Commands

{

[Transaction(TransactionMode.Manual)]

public class Command1 : IExternalCommand

{

public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)

{

TaskDialog.Show("My App", "Hello, Revit World!");

return Result.Succeeded;

}

}

}

4. .addin Manifest File:

A simple XML file that tells Revit where to find your add-in.

XML

<?xml version="1.0" encoding="utf-8" standalone="no"?>

<RevitAddIns>

<AddIn Type="Application">

<Name>My Revit Application</Name>

<Assembly>RevitAddins.dll</Assembly>

<AddInId>YOUR-UNIQUE-GUID-HERE</AddInId>

<FullClassName>RevitAddins.App</FullClassName>

<VendorId>YourCompany</VendorId>

</AddIn>

</RevitAddIns>

**Deployment for Testing:** Manually copy the compiled .dll and .addin file to the C:\Users\[YourUsername]\AppData\Roaming\Autodesk\Revit\Addins\[RevitVersion]\ folder.

**Phase 2: Building Features and Professional Polish (Sprint 2)**

This phase expands the add-in with more commands and a better user interface.

1. Creating Multiple Commands:

New IExternalCommand classes are created for specific functions (e.g., counting doors, getting document titles).

**2. Adding Icons:**

* Add image files (e.g., .png, 32x32px) to the src/Resources/ folder.
* In Visual Studio, select each image file and change its **Build Action** property to **Resource**. This embeds the image directly into the compiled DLL.
* Reference the embedded images in App.cs using the pack URI scheme: new BitmapImage(new Uri("pack://application:,,,/RevitAddins;component/Resources/door\_icon.png"));

**Potential Issue 1: IOException: Cannot locate resource** ❌

* **Problem:** The add-in loaded, but the button icons were not displayed, leading to a resource not found error.
* **Reason:** The **Build Action** for the image files was incorrectly set to **Content**, not **Resource**. Revit's BitmapImage class can only load images that are embedded as a resource.
* **Resolution:** The **Build Action** was changed to **Resource**, and the project was cleaned and rebuilt to force the changes to take effect. This ensured the icons were correctly packaged within the DLL.

**Phase 3: Professional Deployment (Sprint 3)**

This phase packages the add-in into a standard, distributable format.

1. The .bundle Folder Structure:

A self-contained folder that standardizes add-in distribution.

* YourAddinName.bundle/
  + PackageContents.xml
  + Contents/
    - RevitAddins.addin
    - RevitAddins.dll
    - Any other required files.

2. The PackageContents.xml Manifest:

This file is the new entry point for Revit, directing it to the correct .addin file. It's crucial for professional deployments and required by the Autodesk App Store.

**Potential Issue 2: Bundle Not Recognized** ❌

* **Problem:** The add-in failed to load, with no error message, even after the .bundle folder was correctly placed in C:\ProgramData\Autodesk\Revit\Addins\[RevitVersion]\.
* **Reason:** The initial PackageContents.xml was too simple and lacked the RuntimeRequirements element, which explicitly tells Revit which versions the add-in is compatible with. Without this tag, Revit ignores the bundle.
* **Resolution:** The PackageContents.xml was updated to include the necessary RuntimeRequirements tags. The template was based on a professional, multi-version add-in (AlphaBIM), and it revealed the need for explicit version declarations. The final, working template included these tags for Revit 2023 and 2024.

XML

<?xml version="1.0" encoding="utf-8"?>

<ApplicationPackage SchemaVersion="1.0" AutodeskProduct="Revit" ProductType="Application" Name="Your Addin Name" AppVersion="1.0.0" SeriesMin="R2023" SeriesMax="R2024">

<Components>

<ComponentEntry AppName="Your Addin Name" ModuleName="./Contents/RevitAddins.addin" Version="1.0.0"/>

</Components>

</ApplicationPackage>

**Final Deployment:** The entire .bundle folder is copied to C:\ProgramData\Autodesk\ApplicationPlugins. This is the designated location for bundle-format add-ins.

You're absolutely right to focus on professionalism and adherence to Autodesk's standards for submission. The "Hello World" is a great start, but a truly professional add-in for the Autodesk App Store requires a more structured approach.

Let's outline the professional way to build and prepare your Revit add-in for submission.

## **1. Robust Project Setup and Code Practices**

This is the bedrock of a professional add-in.

### Core Development Environment

* **Visual Studio (Community Edition is fine):** Use C# and the .NET Framework.
* **Revit API SDK:** Always have the SDK for your target Revit version installed. It contains essential samples, documentation, and tools.
* **.NET Framework Version:** For Revit 2023, ensure your project targets **.NET Framework 4.8**. This is critical. For future versions, check Autodesk's API documentation for the correct .NET version.
* **Target Architecture (x64):** As we discussed, always target **x64** to match Revit's architecture and avoid runtime issues.

### Code Best Practices

* **Error Handling:** Implement comprehensive try-catch blocks around all Revit API calls. Provide user-friendly error messages.
* **Transactions:** Use Transaction or TransactionGroup correctly. For operations that modify the Revit model, always wrap them in transactions.
  + TransactionMode.Manual: Use this if your command needs to manage transactions itself.
  + TransactionMode.Automatic: Revit handles transaction creation and committing.
  + TransactionMode.ReadOnly: For commands that only read data.
* **Resource Management:** Properly dispose of objects, especially those interacting with Revit's database.
* **Modularity:** Break down your code into logical classes and methods. Avoid monolithic code blocks.
* **Revit API Guidelines:**
  + **IExternalCommand vs. IExternalApplication**: Understand when to use each. IExternalCommand is for user-initiated actions (ribbon buttons), while IExternalApplication is for background processes, event handling, or setting up initial application states.
  + **UI Elements**: Use standard Revit UI patterns. Task Dialogs are preferred over WinForms MessageBox for user feedback.
  + **Performance**: Optimize your code, especially when dealing with large models or numerous elements. Avoid unnecessary element reloads or iterations.

## 2. Add-in Manifest (.addin file) Best Practices

This file is how Revit discovers and loads your add-in.

### Structure and Content

* **XML Format:** It's an XML file with a .addin extension.
* **ExternalApplication Element:** Use this for add-ins that run in the background or manage UI elements.
* **ExternalCommand Element:** Use this for commands that are triggered by a ribbon button.
* **Required Tags:**
  + **Assembly**: The **full path** to your compiled DLL. For deployment, this should ideally be a relative path or just the DLL name if placed correctly.
  + **ClientId**: A **unique GUID** generated specifically for your add-in. Never reuse GUIDs. Use Tools > Create GUID in Visual Studio.
  + **FullClassName**: The complete namespace and class name of your entry point (e.g., YourNamespace.YourCommandClass).
  + **Name**: The display name of your add-in (appears in the ribbon).
  + **Text**: The text for the ribbon button.
  + **Description**: Tooltip text for the button.
* **Optional but Recommended Tags for Professionals:**
  + **VendorId**: A unique identifier for your company. Autodesk often recommends using a reversed domain name (e.g., com.yourcompany).
  + **VendorDescription**: Your company's name.
  + **VisibilityMode**: Controls when the command is visible (e.g., AlwaysVisible, NotVisibleInProject).
  + **LargeImage / SmallImage**: Paths to your ribbon button icons.
  + **LongDescription**: For detailed tooltips.

### Example .addin Manifest Snippet for a Command

XML

<?xml version="1.0" encoding="utf-8"?>

<RevitAddIns>

<AddIn Type="Command">

<Name>Your Add-in Name</Name>

<Assembly>RevitAddins.dll</Assembly> <ClientId>YOUR\_UNIQUE\_GUID\_HERE</ClientId>

<FullClassName>RevitAddins.HelloWorldCommand</FullClassName>

<Text>Hello World</Text>

<Description>Displays a simple welcome message.</Description>

<VendorId>com.yourcompany</VendorId>

<VendorDescription>Your Company Name</VendorDescription>

<VisibilityMode>AlwaysVisible</VisibilityMode>

</AddIn>

</RevitAddIns>

## 3. Packaging for Distribution and the App Store

Autodesk has specific requirements for how your add-in should be packaged.

### The .bundle Structure

Instead of just dropping DLLs and .addin files into a folder, professional add-ins use a .bundle structure.

* **Root Folder:** Your add-in's main folder will be named <YourAddinName>.bundle.
* **PackageContents.xml:** This file within the .bundle describes the contents of your add-in. It's used by Autodesk's installer.
* **Contents Folder:** This subfolder holds your actual add-in files:
  + Your .addin manifest file.
  + Your compiled DLL(s).
  + Any other required assemblies or resources.
  + Icons, images, HTML help files, etc.

**Example Structure:**

YourAddinName.bundle/

├── PackageContents.xml

└── Contents/

├── RevitAddins.addin

├── RevitAddins.dll

├── AnotherLibrary.dll

├── Icons/

│ ├── YourAddinLarge.png

│ └── YourAddinSmall.png

├── Help/

│ └── QuickStart.html

└── EULA.rtf (or .txt)

Autodesk's App Store submission process will help you generate the necessary installer files based on this structure.

### Installer

* For the App Store, you generally don't create your own installer (like an MSI). You provide the files, and Autodesk's system creates an installer for your users.
* The key is to provide the correct structure within your submission package.

## 4. Autodesk App Store Submission Requirements

This is where professionalism truly shines.

### Key Areas to Prepare

1. **Marketing Materials:**
   * **Product Title:** Clear, concise, and under 128 characters.
   * **Description:** A compelling summary of what your add-in does and the problems it solves.
   * **Thumbnail Image:** A professional, high-resolution image (e.g., 256x256 px).
   * **Screenshots:** Showcase your add-in in action within Revit. Highlight key features.
   * **Video (Optional but Recommended):** A short demo video can significantly boost interest.
2. **Documentation:**
   * **HTML Quick-Start Guide:** Autodesk requires this. It's generated from information you provide during submission. It should clearly explain installation and basic usage.
   * **Full Documentation (Optional):** You can link to your own website for more detailed manuals, tutorials, or FAQs.
3. **Compatibility:**
   * Specify which Revit versions your add-in supports (e.g., Revit 2023, 2024, 2025, 2026).
   * Ensure your add-in runs on supported Windows operating systems.
4. **Product Stability:**
   * **No Data Loss:** Your add-in must not cause users to lose data.
   * **No Interference:** It shouldn't block Revit's standard functionality or the functionality of other add-ins.
   * **Thorough Testing:** Test extensively on different Revit versions and in various project scenarios.
5. **Licensing and EULA:**
   * You can provide your own **End-User License Agreement (EULA)**. This is typically a .rtf or .txt file included in your package.
6. **Digital Signing (Highly Recommended):**
   * Digitally sign your DLLs using a code-signing certificate. This adds a layer of trust for users, as Revit will only show one security warning for all your signed add-ins, rather than prompting for each one. This is a significant professionalism boost.

### Submission Process

1. **Register as a Publisher:** Go to the Autodesk App Store Publisher Center.
2. **Prepare Your Package:** Create the .bundle structure with all necessary files.
3. **Fill Out Submission Forms:** Provide all marketing, technical, and licensing information.
4. **Upload Your Package:** Upload your .bundle or source files.
5. **Review and Approval:** Autodesk reviews your submission. This can take some time.

**Step 1: Create the .bundle Directory Structure**

This is the standard, professional folder structure for a Revit add-in that you would submit to the Autodesk App Store. It encapsulates everything the installer needs in a single, organized place.

1. **In your project's main directory** (the one that contains your RevitAddins.sln file and the RevitAddins project folder), create a new folder.
2. Name this folder **RevitAddins.bundle**. The .bundle extension is critical.
3. Inside the RevitAddins.bundle folder, create a new folder named **Contents**.
4. Inside the Contents folder, create a new folder named **2023**. This folder will hold all your files for the Revit 2023 version.

Your folder structure should now look like this:

RevitAddins.sln

...

RevitAddins/

...

RevitAddins.bundle/

├── Contents/

│ └── 2023/

└── PackageContents.xml

**Step 2: Create the PackageContents.xml File**

This is the most important file for App Store submission. It tells the Autodesk installer what to do, what files to copy, and which versions of Revit your add-in supports.

1. **In Visual Studio**, right-click on your **RevitAddins** project in the Solution Explorer.
2. Select **"Add" > "New Item..."**.
3. Choose **"Text File"** and name it **PackageContents.xml**.
4. Paste the following content into the file.

XML

<?xml version="1.0" encoding="utf-8"?>

<ApplicationPackage SchemaVersion="1.0"

AutodeskProduct="Revit"

ProductType="Application"

Name="RevitAddins"

Description="A professional Revit add-in."

AppVersion="1.0.0"

OnlineDocumentation="http://yourwebsite.com/docs/RevitAddins.html"

Author="Your Company Name"

Icon="Contents/YourIcon.ico">

<CompanyDetails

Name="Your Company Name"

Url="http://yourwebsite.com"

Email="support@yourwebsite.com"/>

<RuntimeRequirements

OS="Win64"

Platform="Revit"

SeriesMin="R2023"

SeriesMax="R2023"/>

<Components>

<ComponentEntry

AppName="RevitAddins"

ModuleName="./Contents/2023/RevitAddins.addin"

AppDescription="Professional Revit Add-in"

Version="1.0.0"

Guid="YOUR\_UNIQUE\_GUID\_HERE"/>

</Components>

</ApplicationPackage>

**Critical adjustments for you:**

* + **Name**: Give your add-in a good, descriptive name.
  + **AppVersion**: Use a standard versioning format like 1.0.0.
  + **OnlineDocumentation**: This is for your external documentation link.
  + **Author** and **CompanyDetails**: Fill in your company's information.
  + **SeriesMin** and **SeriesMax**: Since you're targeting only Revit 2023, set both to R2023.
  + **Guid**: Generate a new, unique GUID for your add-in and paste it here. **Do not use the one from your previous .addin file.** This new GUID will be the one that represents your entire add-in bundle on the App Store.

**Step 3: Create the .addin Manifest inside the Bundle**

This is the manifest file that Revit actually reads. It is now located *inside* the bundle, as specified by the PackageContents.xml.

1. In Visual Studio, right-click on your **RevitAddins** project and select **"Add" > "New Item..."**.
2. Choose **"Text File"** and name it **RevitAddins.addin**.
3. Paste the following code, which is the proper .addin format for a professional deployment.

XML

<?xml version="1.0" encoding="utf-8"?>

<RevitAddIns>

<AddIn Type="Command">

<Assembly>RevitAddins.dll</Assembly>

<AddInId>YOUR\_PREVIOUSLY\_GENERATED\_GUID</AddInId>

<FullClassName>RevitAddins.HelloWorldCommand</FullClassName>

<VendorId>YOUR\_VENDOR\_ID</VendorId>

<VendorDescription>Your Company Name</VendorDescription>

</AddIn>

</RevitAddIns>

**Critical adjustments for you:**

* + **<Assembly>**: This path is now **relative to the .addin file**. The DLL will be in the same 2023 folder, so you only need the filename.
  + **<AddInId>**: You will use a **different** GUID here than the one in PackageContents.xml. This GUID identifies the command itself.
  + **FullClassName**: This is the namespace and class name of your command: RevitAddins.HelloWorldCommand.

**Step 4: Configure Visual Studio to Build the Bundle**

This is the final, crucial step. We will automate the process of copying your compiled DLL and the .addin file into the correct location inside the .bundle folder.

1. **Configure Output Path**:
   * Right-click your **RevitAddins** project in Solution Explorer and select **"Properties."**
   * Go to the **"Build"** tab.
   * Change the "Output path" to a path that points to your bundle. It should look like this:

..\RevitAddins.bundle\Contents\2023\

* + This tells Visual Studio to put the compiled DLL directly in the correct folder within the bundle.

1. **Configure .addin and .xml File Properties**:
   * In the Solution Explorer, find your new PackageContents.xml and RevitAddins.addin files.
   * Click on **PackageContents.xml**. In the Properties window, change **"Copy to Output Directory"** to **"Copy if newer."**
   * Click on **RevitAddins.addin**. In the Properties window, change **"Copy to Output Directory"** to **"Copy if newer."**

**Step 5: Final Review**

* Go to **"Build" > "Rebuild Solution."**
* After a successful build, open your RevitAddins.bundle folder.
* The Contents/2023 folder should now contain your **RevitAddins.dll** and **RevitAddins.addin** files.
* The root of the .bundle folder should contain the **PackageContents.xml** file.