Custom Type Editor Template

# Building the Solution Template Package

A Type Editor for the modern out-of-process WinForms Designer is built from several projects. The section *Introduction to the Template Solution* gives all the necessary background information in detail. What’s important for building the templates: There are 2 Visual Studio Solutions in the repo which represent a working type editor, both in C# and in Visual Basic. This solution is in the path *winforms-designer-extensibility\Templates\TypeEditor\src\TemplateSolutions*.

Text

Description automatically generated  
(templateSolutionItems.png)

These solutions provide:

* A WinForms .NET 6 custom control project named *CustomControlLibrary* which holds the actual custom control. The custom control’s only purpose is to render the content of a composite type named *CustomPropertyStore*, which is just the composition of a bunch of properties of different types.  
    
  Graphical user interface, application, Word

  Description automatically generated
* Three projects which make up the Control Designer:
  + The *CustomControlLibrary.Client* project which targets the same .NET Framework version as Visual Studio (in the sample .NET Framework 4.7.2). It holds the actual WinForms type editor, the type editor’s UI, and the client-side view model.
  + The *CustomControlLibrary.Server* project, which targets .NET 6. It holds the actual control designer, a custom CodeDom serializer which determines the necessary steps to generate custom property code for `InitializeComponent`, a designer action list implementation for the control (see screenshot above) and finally a couple of methods in the server-side view model which are called by the client to control aspects of the UI.
  + The *CustomControlLibrary.Protocol* project which holds all the classes which are necessary to handle the communication between the client and the server process.
* A Package project named *CustomControlLibrary.Package*, which packs the binaries of all those projects in a special structure as a NuGet project, so they can be loaded by the WinForms Designer in the individual client and server processes.
* A .NET 6 WinForms project named *CustomTypeEditorTest* to test the control and its design-time functionality.

The procedure for building the actual templates from the template solutions is as follows:

* Make sure the template solutions work as planned if you made any modifications.
* From the command line, change the current directory to *Templates*.
* Run the prepareTemplates.bat batch file.

This copies the relevant project files from the template solution to the templates folder. The batch file then calls *dotnet pack* to create the solution template package and also installs the package with *dotnet new install*. You should see the result of that operation:

D:\Git\NetControlDesigners\src\Templates\Templates>dotnet new install .\bin\Debug\Microsoft.WinForms.Designer.TypeEditorTemplate.1.1.0-prerelease-preview3.nupkg

The following template packages will be installed:

D:\Git\NetControlDesigners\src\Templates\Templates\bin\Debug\Microsoft.WinForms.Designer.TypeEditorTemplate.1.1.0-prerelease-preview3.nupkg

Success: Microsoft.WinForms.Designer.TypeEditorTemplate::1.1.0-prerelease-preview3 installed the following templates:

Template Name Short Name Language Tags

--------------------- ------------------ -------- ------------------------------------------------------------------

WinForms .NET Cust... WinFormsTypeEditor [C#] WinForms/Designer/TypeEditor/ActionList/CodeDomSerializer/Solution

WinForms .NET Cust... WinFormsTypeEditor VB WinForms/Designer/TypeEditor/ActionList/CodeDomSerializer/Solution

# Using the Type Editor Template

After building the templates they are ready to use from the CLI as well as from Visual Studio.

## Creating a new Type Editor Solution within Visual Studio

* Start Visual Studio, and klick on *Create a new Project*.
* In the Dialog, type “winforms” in the filter textbox.
* Pick one of the available type editor templates, either for C# or Visual Basic.  
    
  A screenshot of a computer

  Description automatically generated with medium confidence
* Click *Next*.
* In the *Configure your Project* page, specify the following options:
  + **Project name:** This will become the base name of the Project.
  + **Location:** This is the path where the solution and the respective projects will be created.
  + **Solution:** You can choose here if the projects should be created as a new or be added to an existing solution.
  + **Solution name:** The name of the solution.
* Click *Next*.
* On the *Additional Information* page, specify the following options:
  + **Framework:** Pick the .NET Version you want the server components (control library, control designer project) targeted against. Note that at this point the client projects will always target classic .NET Framework 4.7.2, as this is the Visual Studio target framework version. **NOTE:** The type editor templates support .NET Versions from 6.0 on.
  + **PropertyTypeName:** This is the name of the individual custom property the type editor will offer the editing functionality for. In the sample project the templates are based on, this is the `CustomPropertyStore` type. Every reference to this type name or file name will be renamed to the class name you’re entering here.
  + **Type Editor Name:** This is the name of the type editor. In the sample project the templates are based on, this is the `CustomTypeEditor` type. Every reference to this type name or file name will be renamed to the class name you’re entering here.
  + **CustomControlName:** This is the name of the custom control. In the sample project the templates are based on, this is the `CustomControl` type. And again, every reference to this type name or file name will be renamed to the class name you’re entering here.
* Click *Create* to create the solution.

## Creating a new Type Editor Solution from the dotnet CLI

After installing the templates, you are using the type editor solution templates like every other Visual Basic or C# templates from the CLI. Refer to the help option for the exact parameter names. The parameters are the same as in the above *additional options* description:

# Introduction to the Template Solution

Since .NET 3.1 started to support the WinForms Runtime, a new WinForms designer was needed to support .NET applications. This work required a near-complete rearchitecting of the designer, as we responded to the differences between .NET and the .NET Framework based WinForms designer everyone knows and loves.

Until we added support for .NET Core applications there was only a single process, devenv.exe, that both the Visual Studio environment and the application being designed ran within. But .NET Framework and .NET Core can’t both run together within devenv.exe, and as a result we had to take the designer out of process. We call the existing process – the process that Visual Studio runs in – the client process or the Visual Studio process, and the process which shows the actual Form at Design time, the server process or short *DesignToolsServer*.

## Enter the DesignToolsServer

While simple control designer scenarios like type converters, action lists or CodeDom serializers don’t need any substantial rewrites, type editors are a different beast altogether.

To illustrate the additional requirements which arose by introducing different processes, let’s look at a typical type editor for any property of type `Image` like the Button’s *BackgroundImage* property: While the actual Image that you picked will be rendered on a button in the server process, the dialog you picked the Image *with* runs in the context of Visual Studio. That in turn means there is communication between the two processes necessary which custom type editors for the modern WinForms Designer need to take care of. In addition, type editor solutions also need to provide a NuGet package whose individual assemblies gets partly loaded into the Visual Studio process and partly into the DesignToolsServer. And to that end, this NuGet needs to have a special structure, which a dedicated Package project takes care of. If you want to learn more about the concept of the modern WinForms Designer: [This blog post](https://devblogs.microsoft.com/dotnet/state-of-the-windows-forms-designer-for-net-applications/) describes the concept of the different processes in greater detail.

## Projects which the templates create

Setting all these things up manually means coordinating a lot of moving parts, and there is a huge potential that things go wrong. The individual projects created by this template help to prevent falling into those traps. The templates create a series of projects and important Solution Folders, depending on your needs for both C# and Visual Basic. Let’s look at the projects which are part of the template solution in detail:

* **\_Solution Items:** This is not really a project, but rather a solution folder, which holds this readme, the *Directory.Build* Target which determines the NuGet package version for the used Designer SDK version and the *NuGet.config* setting. If at any point you would need to change the SDK version which are used throughout the solution, you would only need to change them in one spot: here.
* **CustomControlLibrary.Client** This is a project of the same target framework version as Visual Studio, and it holds the actual type editor UI running in the context of Visual Studio. It also contains the so-called client *view model*, which is a UI controller class. It communicates on the one side with its pendant in the server process and controls the client-based UI (a modal WinForms Dialog) on the other side based on the server-provided data.
* **CustomControlLibrary.Server:** This project holds every aspect of the control designer, which needs to be executed in the context of the server process. Those are
  + The server-side view model, which provides the necessary data to the client-side view model.
  + The factory class, which generates the server-side view model.
  + A custom CodeDom serializer for the custom property type the type editor is handling, should one be needed.
  + A custom designer action list which can be accessed at design time through the designer action glyph of the control. Please note, that although these classes are hosted purely in the server-side designer assembly, the UI for the respective action list is still shown in the context of Visual Studio. But the communication to that end is done completely behind the scenes by the Designer SDK. So, even if it looks like the designer action lists are handled exclusively by the server process, they are not. The rendering of the UI in the context of Visual Studio is completely done behind the scenes, and the WinForms Designer takes care of the communication between the client and the server process completely on its own.
  + The actual control designer, which – as one example – paints the adornments for the controls. This is the only part of the UI which is actually rendered server-side, and although it looks like this rendering is done in the context of Visual Studio, it is not. The rendering of the Form and all its components at design time is done by the DesignToolsServer and just projected on the client-area of Visual Studio Design surface.
* **CustomControlLibrary.Protocol:** This project contains all the classes which are necessary for the communication between the client and the server process via [JSON-RPC](https://www.jsonrpc.org/). The Designer SDK provides a series of base classes which are optimized for transferring WinForms-typical data types between the client- and the server-process. A typical protocol library for a control designer builds on those classes.
* **CustomControlLibrary:** This is the project, which contains your actual custom control(s).
* **CustomControlLibrary.Package:** This is the project which creates the NuGet package. This NuGet package organizes the individual control designer components for the DesignToolsServer server process and the Visual Studio client process in respective folders, so that the required parts are available for the processes at design time.

## Invoking Type Editors, In Process vs. Out-Of-Process

In the classic framework, invoking of a type editor is a straightforward procedure. Here is what happens, when the user triggers to edit a value of a property by opening a type editor via the Visual Studio’s property browser:

* The user wants to set a value for a property of a control which either doesn’t have a default string representation (like an image or a sound file) or is a composite property type, which demands a more complex user interaction. A type editor for that property type is defined by the `EditorAttribute` (see class `CustomPropertyStore` in the template project as an example).
* The custom type editor class, which is usually provided along with the type the custom control provides for that special property, is instantiated when the user clicks on the …-Button in the property’s cell of Visual Studio’s property browser.
* The property browser now calls the `EditValue` method of the type editor and passes the value of the property to set. In other words: The type editor receives the instance of the custom property. In the example of the `BackgroundImage` property of the Button control, the instance would be the actual image. In our template example, that instance would be of type `CustomPropertyStore`.
* The type editor now gets the `UIDialogService`, which enables the type editor to display a modal (WinForms) dialog in the context of Visual Studio. It is important to show the dialog in the context of Visual Studio, because otherwise Windows message processing queues of different processes would run concurrently and quickly dead-lock each other, so that Visual Studio would freeze.
* The UI converts the value in an editable format, gets the updates from the users, and then converts the edits back to the type of that control’s custom property. The value, which the type editor returns, is now assigned by the property browser to the property.

And here now is the all-important difference compared to the out-of-process Designer scenario: When the property browser asks the UITypeEditor to display the visual representation of the value, that type’s value is not available in the context of Visual Studio. The reason: The property browser runs in a process targeting a different .NET version than the process that defines the type. Visual Studio runs, for example, against .NET Framework 4.7.2 while the custom control library you are developing is e. g. targeting .NET 7. There is simply no way that .NET Framework can deal with types defined in or based on types defined .NET 7. So, instead of giving the UITypeEditor the control’s custom/special property’s value directly, it’s handing it via a so-called *proxy object*.

The concept of proxy objects in the client (Visual Studio) process does require a special infrastructure for handling user inputs in custom type editors. Let’s look at what infrastructure components of the Designer we need to understand, before we talk about the workflow for setting the value in the OOP scenario:

* **Using View Models:** View models are for controlling aspects of a UI without having a direct reference to the UI specific components. Don’t confuse view models in the context of the WinForms Designer with view models you might know from XAML languages, though! They are only remote relatives. Yes, they are controlling the UI in a UI-technology independent way – and that’s the main aspect of a view model. But no, in contrast to XAML, they are not doing this by direct data binding. View models in the context of the Designer are rather used, to sync certain conditions of the UI between the client and the server process. The class `CustomTypeEditorVMClient` provides a static method `Create`, which is the dedicated way to create a view model. You pass it the service provider and also the proxy object of the instance of the property value to edit, which the client-side type editor just got from the property browser.
* **Sessions and the DesignToolsClient:** For the communication with the DesignToolsServer server process, the Designer not only need the endpoint on the server, but also on the client side. The `DesignToolsClient` class represents that client-side endpoint and provides the basic mechanisms for communication with the server. To separate the concerns of each open designable WinForms document within Visual Studio, each open Designer is associated with a session. The `Create` method in the sample shows how to retrieve a session along with and the `DesignToolsClient` through the service provider, and can now, with both objects, talk to the server – in this case to create the respective *server-side* view model.
* **Proxy classes:** These classes solve the basic challenge: Representing objects of server-side ,NET version types which are not known to the client. If you select a component in the Designer, what the property browser “sees” is a proxy object which kind of points to the real object in the server process, not the actual one. A value of a property of a complex type is also represented by a proxy object, since – again – its type only exists on the server, because it’s targeting a different .NET version. And yet again: Also, the view model returned from the server is not the actual server-side view model instance (it can’t, because, again, it might contain or be based on types that are not existing in the client-side target framework) but what we call a proxy class, so, the pointer to the actual server-side hosted instance. The client-side view model will need this view model-proxy to synchronize necessary data across the process boundaries.
* **Data transport and remote procedure calls:** The communication between client and server is always synchronous, so blocking: You define endpoints in the server-process, which the client calls. Basically, each endpoint needs three dedicated classes: A *Request* class, defined in the Protocol project (see below), which transports necessary data to the DesignToolsServer. A *Response* class, which transports result data back to the client process – also defined in the Protocol project. And lastly a *Handler* class, which is the server-side remote-procedure to call, if you will. In this template, two endpoints are already predefined: `CreateCustomTypeEditorVM` creates the server-side view model, whose instance is then hosted as a proxy-object in the client-side view model, so the communication and data exchange can be simplified over those two instances. And then there is also the `TypeEditorOKClick` endpoint: This method is called when the user clicked the OK button of the type editor during design time to indicate that they changed the value passed by the property browser. Since the custom property type only exists in the DesignToolsServer, the client can only pass over the individual data fragments from what the user entered in the dialog to the server process. But it is the server which then creates the actual instance of the value of what it got passed from the client. And it eventually assigns that value to the property of the user control.

Now, with these important basics in mind, here is the workflow for setting a property value via a type editor in the out-of-process Designer scenario in detail:

* As in the classic in-process-Scenario, the user wants to set a value for a custom property. And again, a type editor for that property type is defined by the `EditorAttribute` (see class `CustomPropertyStore` in the template project). The first important difference: Since the type in question might not be available in the client process’ target framework, the type can only be defined as a string. Also as before, the custom type editor class is instantiated when the user clicks on the …-Button in the property’s cell of the property browser. Now, here is a first exciting challenge that the modern designer faces: When the custom control lives only in the server process, and the actual type editor lives only in the client, how does the WinForms Designer finds the type editor on the client side? This is where an important component in the client designer project comes into play: the `TypeRoutingProvider`. It holds a table of `TypeRoutingDefinition` ibjects and assigns the names of the editors to the actual types. That means, if you were ever to add additional type editors for other property types or controls to your control library solution, this table must be ament and maintained accordingly. It’s best practice to use the `EditorNames` definitions in the Protocol project to that end, since it minimizes typos by providing IntelliSense support.
* Now, yet again, the property browser calls the `EditValue` method of the type editor and passes the value of the property to set. But the value now is not the actual value of the property. It’s rather the proxy object, which points to the actual instance of the value in the server process. This also means, processing the value must be happening in the server-process. To this end, the two view model types to control the edit procedure need now to be used: one on the client side (`CustomTypeEditorVMClient`), and one on the server side (`CustomTypeEditorVM`). The template creates both classes for you, along with the infrastructure methods to set them up.
* The static `Create` method of the client-side view model has now all the information to create the actual client-side view model by calling the `CreateViewModelClient` method of the Designer service provider, and for that it passes the server-side proxy to the server view model.
* The type editor’s main task is to edit the value of type `CustomPropertyStore`. To keep the example simple, this is just a composite type, composed of a `string`, a `DateTime`, a list of `string` elements and a custom Enum. As a reminder: since this type only exists server-side, the UI (being in the context of Visual Studio) cannot use this type. This is where the Protocol project/assembly comes into play. The Protocol project defines all the transport classes, which can be used in either process. It is defined as a .NET standard library, so all its types can be projected and used in both .NET and .NET Framework projects equally. So, to solve this requirement, we mirror the `CustomPropertyStore` type with a special data class we define in the Protocol project named `CustomPropertyStoreData`. This type also provides the necessary methods to convert the data its hosting into the JSON format and back from it, which is needed to transport it across the process’s boundaries. With that, the response class for the endpoint to create the server-side view model not only takes the proxy of the server-side view model, but also the original values of the types, the custom property type is composed of. And this data we now use to populate the type editor’s dialog client side.
* The user now edits the values.
* When the user clicks OK, we validate the data on the client inside the `CustomTypeEditorDialog`. And if the validation passes, the dialog’s returns `DialogResult.OK`, and we call the `ExecuteOKCommand` method of the client view model, to kick of the data transfer to the server. This method now sends the `CustomTypeEditorOKClickRequest` to the server and passes the individual retrieved data from the user’s input of the dialog along. The endpoint’s handler gets those data and passes - in turn - that data to the server-side view model, which then again calls its `OnClick` method, composes the actual instance of the custom control’s property type, and stores it in the `PropertyStore` property of the server-side view model. And with that the call chain seems to end here. So, the server-side ViewModel now holds the edited and committed result. But wait! How does the ViewModel property find the way back to the control’s property? That last step is done client-side, and it’s kind of subtle: Remember? When the client-side view model got created, it not only triggered the creation of the server-side view model. It also requested the proxy of that view model to be returned to the client side. On the client, the client-side ViewModel holds the reference to server-side view model’s PropertyStore property over a proxy object. When the user clicks OK in the editor, that code flow is returned to the type editor (running in the context of Visual Studio), which opened the modal dialog to begin with. Now, back in the actual type editor class, it is where the assignment from this ViewModel to the actual Property of the Control happens:

var dialogResult = editorService.ShowDialog(\_customTypeEditorDialog);

if (dialogResult == DialogResult.OK)

{

// By now, the UI of the Editor has asked its (client-side) ViewModel

// to run the code which updates the property value. It passes the data to

// the server, which in turn updates the server-side ViewModel.

// When it's time to return the value from the client-side ViewModel back to the

// Property Browser (which has called the type editor in the first place), the client-side

// ViewModel accesses its PropertyStore property, which in turn gets the required PropertyStore

// proxy object directly from the server-side ViewModel.

value = viewModelClient.PropertyStore;

}

The PropertyStore property of the ViewModelClient doesn’t have a dedicated backing field to hold the value. Rather, it uses the infrastructure of the proxy to communicate with the server-side view model to get the just created proxy of the server-side view model’s PropertyStore content directly. And the proxy object is what we need here: Again, since the client doesn’t know the type, it can only deal with the proxy objects which point and represent the server types instead.