Custom TypeEditor Template

Since .NET 3.1 started to support the WinForms Runtime, a new WinForms designer was needed to support .NET applications. The 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, thus we called the new designer – WinForms Out of Process Designer (or OOP designer for short). We call the existing process – the process that Visual Studio runs in – the client process, and the process which shows the actual Form at Design time, the server process, which we call *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 problem created by introducing this DesignToolsServer as a dedicated server process, let’s look at a typical type editor for any image processing control property 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 it *with* runs in the context of Visual Studio. That in turn means there is some considerable communication between the two processes necessary which custom Type Editors for the OOP-Designer need to take care of. In addition, Type Editors also need to provide a NuGet package which gets partly loaded into the Visual Studio process and partly into the server process. And to that end, this NuGet needs to have a special structure (see below for details). [This blog post](https://devblogs.microsoft.com/dotnet/state-of-the-windows-forms-designer-for-net-applications/) describes the concept of the OOP designer and 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:

* **\_Solution Items:** This is a solution folder, which holds this readme, the Directory.Build Target which determines the NuGet package version for the Designer SDK and the NuGet.config setting.
* **CustomControlLibrary.Client** This is project of the same target framework as Visual Studio, and which 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 the server process and controls the client-based UI 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 excludively by the server process, they are not. The rendering of the UI in the context of Visual Studio is completely covered behind the scenes, and the Designer take 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, but 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:** The project holds 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 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 was a straightforward procedure:

* 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, 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).
* 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 the 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 this exact context, because otherwise Windows message processing queues of different processes would run concurrently and quickly dead-lock each other, so that the 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 custm property. The value, which the type editor returns, is now the assigned to the property by the property browser.

And here now is the all-important difference to the Out-Of-Process scenario: When the property browser asks the UITypeEditor to display the visual representation of the value, that type’s value is not available. The reason: The property browser runs in a different process than the process that defines the type. And the reason for that again is that Visual Studio is targeting a different .NET (framework) version altogether: 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 there must be different processes for this dilemma to be resolved. So, instead of giving the UITypeEditor the control’s custom/special property’s value directly, its handing it a so-called proxy object. Let’s take a look at what infrastructure components of the OOP-Designer we need to understand, before we talk about the workflow for setting the value in the OOP scenario:

* **Creating ViewModels:** The class `CustomTypeEditorVMClient` provides a static method `Create`. You pass it the service provider and the proxy object the client-side type editor just got from the property browser. Don’t confuse view models in the context of the WinForms Designer with View Models you might know from XAML languages. They are only remote relatives. Yes, they are controlling the UI in a way. But no, 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.
* **Sessions and the DesignToolsClient:** For the communication with the DesignToolsServer server process, we need the other endpoint on the client side. The `DesignToolsClient` class represents this 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:** The view model returned from the server is not the actual server-side view model instance (it can’t, because 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, almost pointer to the actual server-side hosted instance – in the example the instance of the server-side view model. The client-side view model will need this 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 value of what he go passed from the client and eventually assigns it the property of the user control.

Now, with this important basics in mind, here is the workflow for setting a property value via a type editor in the OOP 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.
* And 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, 2 view model types to control the edit procedure need 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. 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’s 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 the problem, 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 processes bounderies. 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 now 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 process to the server. This method now sends the `CustomTypeEditorOKClickRequest` to the server and passes the induvial retrieved data from the user’s input in 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 the `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. The question now is: How does the ViewModel property find the way back to the control’s property? That is done client-side: 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 ProxyObject. When the user clicks OK in the editor, that code flow is returned to the the TypeEditor (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 TypeEditor 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.

# Extending the solution by additional type editors

(TBD.)