Main types of development for DigitalEdge solution.

# Introduction

### This document presents and explains standard paths for developing new, or modifying existing, functionalities on DigitalEdge web application.

Intermediate knowledge about MVC 5 framework and toolset is required, and assumed by the document creator.

# UI FLOW Introduction (Digital Edge process)

Ui Flows in DigitalEdge is a set of complete functionalities which enables the application to track stages of multistep flows which asks the user, through multiple states, to fill in the data needed for a single or multiple actions he requested to perform.

Most common flows are structured like this:

**Enter data screen/s** – user is asked to input all the data required for the action he is performing. User can be presented with multiple steps/stages/states if the data he is asked to enter is too numerous to fit on one screen, or there are multiple paths in the flow, which is dependent on the data he inputted.

**Review entered data screen** – this is step two of the flow, where the user can review the data he filled out, and check if it’s alright. He can then go back and edit the data or continue to the next step. If additional authorization is requested for that action, **subprocess** of authorization will be triggered, where user will be asked to use another factor of authentication. **Subprocesses** are explained later in this document

**Result/status and additional actions screen** – User is informed about the competition of his action, and presented with additional options. Those options can be related to the action he just performed, like “repeat” or “add geolocation to transfer”, or it can be related to his next point of navigation on the application, such as “go to homepage” or “view transfer list”

# UI FLOW Model and definition (Digital Edge process)

There are two main types of UI FLOWs in Digital Edge solution.

Coded UI Flow and Generic UI Flow. Classes used for each are named:

CodedUIProcess<T> : Process

GenericUIProcess : Process

Generic UI flow is using the same structure as Coded UI flow, but the process of development is different.

Generic UI flow is loaded from the database, whole definition is being injected into process object in memory and then executed as-is. Generic flow definition is created and edited through “Digital Edge Studio” tool which allows the flow creator to use visual elements, through drag&drop mechanism, to organize and define the whole process.

In this document, we will talk only about Coded UI flows, and the documentation about Generic Ui flows should be contained in a separate document.

Coded UI flow is self-explanatory. It is a whole process definition written in code.

The main building block of processes is its definition. How everything will look and work is defined in one file. Digital Edge solution has a whole infrastructure developed to support this definition.

As seen above, Coded UI Process derives from the class “Process” and this is where most of the process definition lies. Let’s look at the structure of the “Process” class:

|  |  |  |
| --- | --- | --- |
| UiContentPlacement.Placement | Property:Enum | Where this process will be displayed. Processes can be placed Inline or in a modal popup (wide, narrow or side popup) |
| string OnDoneUtility | Property: String | Which “Utility” (JavaScript function) will be executed after the flow is done |
| int ProcessId | Property:Int | Process identifier |
| string WorkItemId | Property:String | Unique flow identfier |
| string ContainerId | Property:String | Id of the html element which will contain the process markup |
| MessagesDisplay MessagesDisplay | Property:MessagesDisplay | Used for setting the way of displaying messages, warnings during the UI Flow execution |
| bool UsesFlowHistory | Property:Boolean | UiFlow system has an automated way of remembering the history of state changes. Enabling this will enable history tracking so using “back button” is seamlessly easy to implement. This is not always possible, so the developer can decide whether or not to use this functionality |
| string ShortName | Property:String | Short name of the process, which will be translated on the UI. This should be populated with the localization key which has his localization strings defined. |
| string Description | Property:String | Description of the, which will be translated on the UI. This should be populated with the localization key which has his localization strings defined. |
| RouteValues RouteValues | Property:RouteValues | Whats the url of the process. Controller, Area and Action should be defined here |
| Messages Messages | Property:Messages | Object that contains all the messages that should be displayed to the user during the process |
| List<State> ProcessStates | Property:State | List of states objects. State is another complex object which defines one screen, one step of the process/flow |
| List<Form> Forms | Property:Form | List of forms. Every state has its form, which is used to display input fields to the user and bind it to the process object, more precisely, to the process workitem object, through the binding paths |
| ProcessActivity OnInitActivity | Property:ProcessActivity | Definition of the microsoft foundation activity (xaml) which will be executed on process initialization (when the user enters the flow page) |
| CommonStatesDefinition CommonStatesDefinition | Property:CommonStatesDefinition | This object contains properties informing the flow about some of the steps/states meaning. Basically, states can present a state where user is abandoning the process, or one if the states is a state where process ends. All of this should be defined here |
| int CurrentState | Property:Int | Id of the state which is currently active on the users UI |
| SubProcessInfo SubProcessInfo | Property:SubProcessInfo | Information about the subprocesses related to this current process. Digital Edge solution has an ability to run multiple UI Flows in the same time, and start each of them based on a trigger defined in this object. |
| FlowHistory FlowHistory | Property:FlowHistory | This Is being automatically populated if flow history is enabled on the process, which we talked about above |
| List<VariableField> | Property:VariableField | This is a place where some of the custom values can be defined, and then transferred, through reflection and binding paths to the workitem object. Query string values are also mapped to predefined variable fields, by matching query string key and variable field key defined. This means we can enter the process with different values provided |
| List<ProcessOutput> ProcessOutputs | Property:ProcessOutput | Definition of the values one flow is expected to return. This values can be later used by other processes. Perfect example is that additional authorization has an output of OTP code which user has entered. That otp is actually used by the process which started authorization subprocess, and it’s transferred through the ProcessOutputs definition |
| Form ActiveForm | Property:Form | Returns the form connected to the state that is currently active in the flow and on the UI of the user |
| State ActiveState | Property:State | Returns the currently active state in the flow and on the UI of the user |
| NavigateToFormId | Method:Void | Sets a state as active in the flow, which has the form with the given ID |
| void StartSubProcess | Method:Void | Starts the subprocess, from the list of subprocesses, with the given identifier |
| void AddErrorMessage | Method:Void | Adds a new entry to the Messages object, which will be displayed to the user as an error after the current postback |
| void AddInfoMessage | Method:Void | Adds a new entry to the Messages object, which will be displayed to the user as an information after the current postback |
| void AddWarningMessage | Method:Void | Adds a new entry to the Messages object, which will be displayed to the user as a warning after the current postback |

Ui Flow is a set of ordered states. User goes through the states until the flow has finished. State object has a reference to Form which is placed in process object, and has list of actions. Actions are basically buttons on the page, but can be used in other ways. Action leads to another state or refreshes the current one with some additional code execution in-between, such as api calls or some custom code with mapping or decisions.

Let’s look at the State model:

|  |  |  |
| --- | --- | --- |
| int? StateId | Property:Int | Unique identifier of the state |
| bool CanGoBackUsingFlow | Property:Boolean | Determines if the next state can go back to this state using the generic ui flow history. Also requires UsesFlowHistory property of the process to be set to True |
| HotSaveLevel HotSaveLevel | Property:HotSaveLevel | The weight of importance for saving the current state of the process to the database. Digital Edge system allows user to continue his flow even after he exited it, at any time, and this property will determine if current state of the flow should be saved and how often. |
| UiStateType StateType | Property:UiStateType | Type of the state, which defines its usage |
| string ShortName | Property:String | Short name of the state which will be translated on the UI. This should be populated with the localization key which has his localization strings defined. |
| string Description | Property:String | Description of the state which will be translated on the UI. This should be populated with the localization key which has his localization strings defined. |
| string FormId | Property:String | Identifier of the form, as a reference to the form that’s being used on the state |
| List<Action> Actions | Property:Action | List of actions user can perform on the state. Not all actions are visible to the user and not all actions are executed by the user. Some of them are executed automatically. |
| public Messages Messages | Property:Messages | Object that contains all the messages that should be displayed to the user while he’s on specific state |
| List<StateAction> StateActions | Property:StateAction | This is not same as actions, since it’s not executed from the user in any case. These actions are UI actions executed automatically when the user is on a specific state. They can be configured with when and how they should be executed, by using StateAction model. |
| void AddErrorMessage | Method:Void | Adds a new entry to the Messages object, which will be displayed to the user as an error after the current postback |
| void AddInfoMessage | Method:Void | Adds a new entry to the Messages object, which will be displayed to the user as an information after the current postback |
| void AddWarningMessage | Method:Void | Adds a new entry to the Messages object, which will be displayed to the user as a warning after the current postback |

CodedUiProcess wraps around the Process object, and defines Generic type **T,** which is actually type of the workitem, and is different on every new process definition.

**Workitem** is a only object in this ui flow model which is not predefined – developer has freedom to add any property he wants in a work item. This is where we store any of the variables we need for our current flow, api calls or decisions for path of the flow.

# Developing an UI Flow in Digital Edge Solution

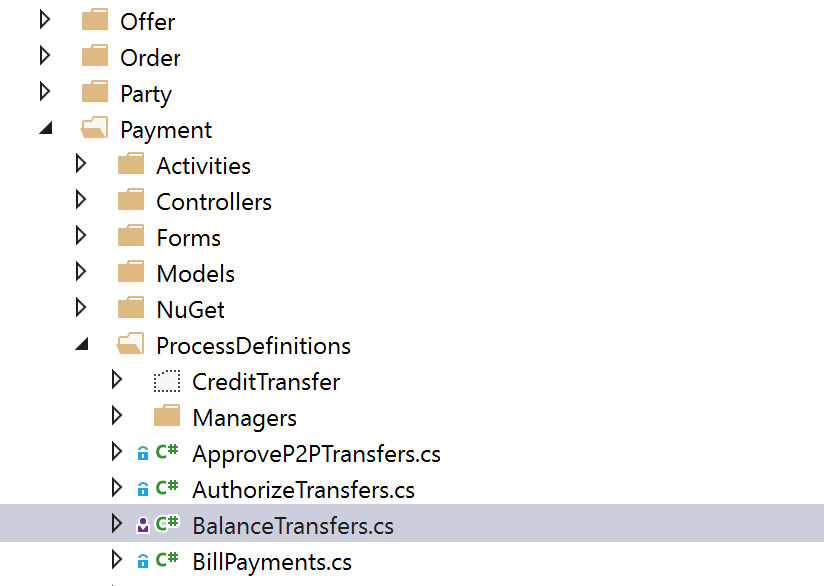
Let’s go by example. We will look at the balance transfer process, and explain step by step.

First we need to determine where will we put files with the code/definition.

Digital Edge web project’s mvc areas are organized the same way DEWEB API is organized, meaning that we have 1 area that has all of the functionalities from 1 api of the same name.

So, since it’s a balance transfer process and the API endpoint is located in the Payment API, balance transfer process definition should be in payment area.

First we create ProcessDefinition, WorkItem definition and forms definition, and the most common way of doing that is to create a .cs file named same as the name of the process you’re creating, in the folder “ProcessDefinitions” in the appropriate area, in this case its Areas\Payment\ProcessDefinitions\BalanceTransfers.cs



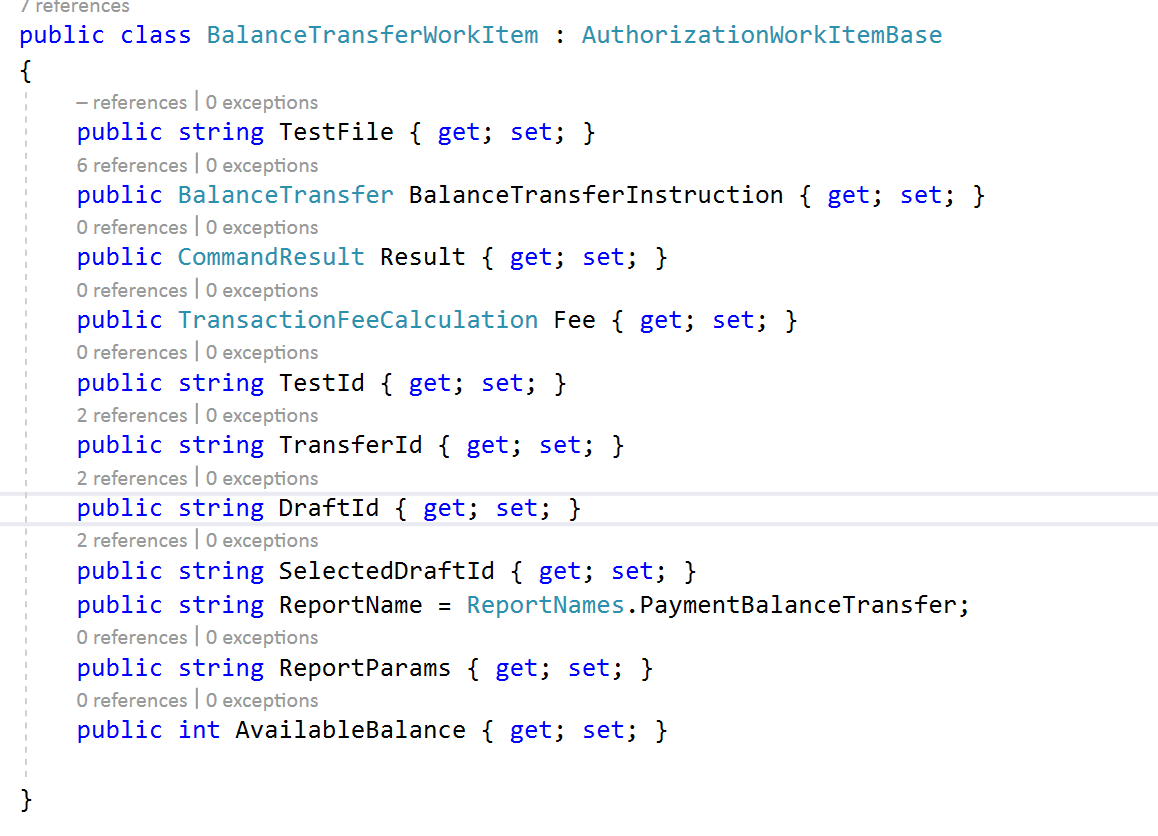
## Workitem Definition

In this file, we create our WorkItem, in this case its BalanceTransferWorkItem, which inherits from AuthorizationWorkItemBase.

We inherit from this base workitem in case we need additional authorization to be performed, and since many of the flows require so, AuthorizationWorkItemBase which has all required properties used for additional authorization, is used throughout the solution, in order to prevent code duplication

public class BalanceTransferWorkItem : AuthorizationWorkItemBase

Here we defined all the variables we need to store in memory, so we can use them in the Flow.



## Forms in UI Flows

Next up, we create our forms, which will be used for 3 different states we need for our process.

Form model is pretty simple for people used to Web programming. One form can contain fields and field groups. Every field has a property “Group” which is used to define which field belongs to which group

Model used for fields is a complex object with multiple inheritance/abstraction levels. You can define label for a field, validation rules, whether It’s displayed or hidden**, ReviewMode, Display order** and **width percentage** and also, you can define it’s **BindingPath.**

As said above, we always tend to give a preview of his inputted data to the user, and for this purpose, we use ReviewMode property of InputField, which will render the label and value in a read-only mode.

DisplayOrder and WidthPercentage is used for generic view rendering. Digital Edge solution has a functionality that will organize and display fields, buttons, and everything else on the UI by generic field rendering.

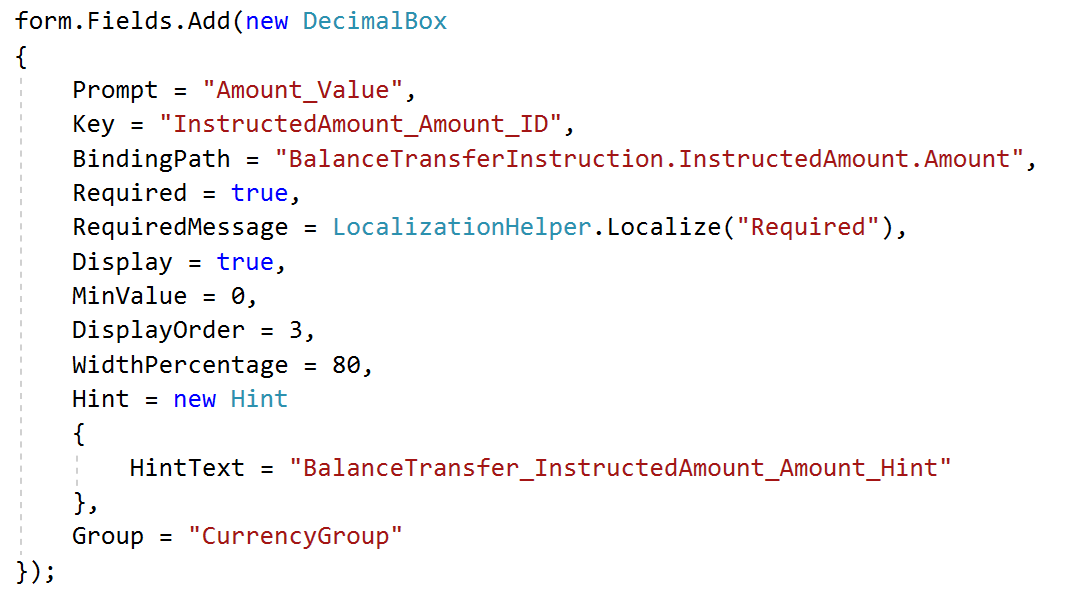
## Generic rendering of forms

Form has a property named “ViewPath”, and when populated, it will use that .cshtml name in order to render a view to the user. That view, can further redirect to generic rendering, where system will generate a page solely on forms definition, or it can have custom code if it’s the only way to generate some specific HTML. We tend to have at least as possible of those “Custom” .cshtml files, so have in mind that, when developing some page on Digital Edge, always try to use exusting generic view rendering. If “ViewPath” property of the form is not populated, system will automatically try to render it by generic way

## Binding paths

Every input field has a property named “BindingPath”. Binding path is used for mapping values, using reflection, from the fields to the **WorkItem**, and in reverse, from the **WorkItem** to a specific **InputField.**

Let’s look at the Amount field on BalanceTransfer:



Binding path is a string property which is populated with some property’s path, using the dot notation, just like it’s used for accessing object members in c#. This specific path is from the property of the above mentioned BalanceTransferWorkItem named “BalanceTransferInstruction”. Since this binding path is only used for field-to-workitem and workitem-to-field binding, it is not mandatory to specify “WorkItem.” part, because it’s assumed you are asking for a property in the workitem.

In this particular situation, value of this DecimalBox will be automatically mapped to:

**BalanceTransferWorkItem.BalanceTransferInstruction.InstructedAmount.Amount**

Binding paths functionality helps us lessen the amount of code needed to remap and use values that the user has entered, with the use of this component everything is done automatically.

## Process definition

Now, in the same file, we also create method that returns ProcessDefinition. We set everything that’s needed for this flow based on the table with explanations written above, in this document.

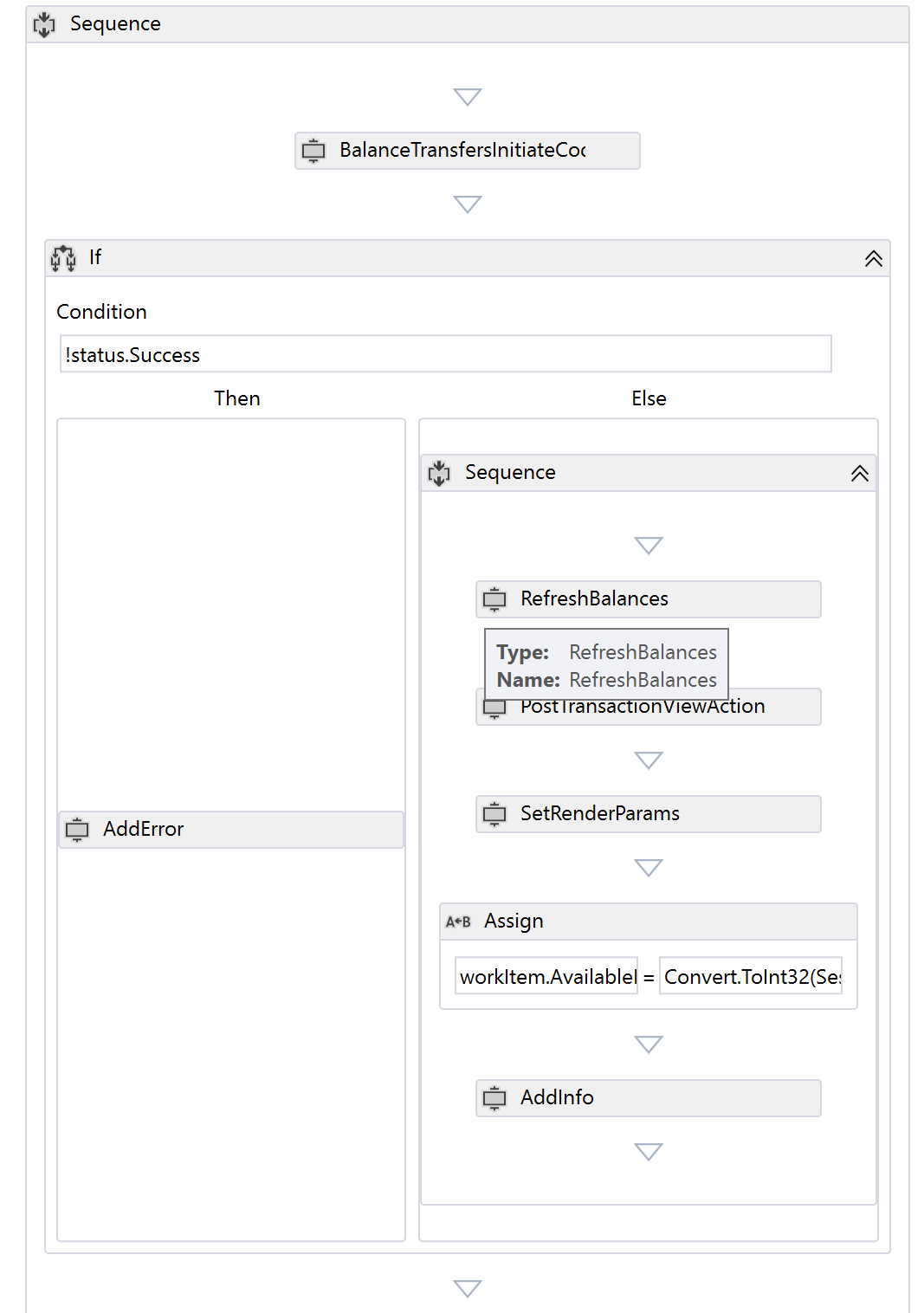
For this particular process, we stated things like this:

* We knew it will be a 3 step process, so we added three states with three forms. First is for data input, second is filled with fields with ReviewMode set to true, and the third which is fairly empty because there are no fields, just a status screen. Since it’s a 3 step process, we set the CommonStatesDefinition to EndStateId = 3
* We set the placement to the Inline
* We added AuthorizationSubProcess and other subprocesses to the subprocess definition
* We added the corresponding actions to the states which will let the user navigate through the UI Flow, and for those that had some additional functionalities that needed to be executed, we set the **“Transitions” property to a list of transitions** and set the paths to them

## Transitions across states using actions

Digital Edge solution uses .xaml activity files to invoke custom methods. These transitions are linked to the actions on the states, so when user executes, for example, “Pay” action on the second state of this UI Flow we are working on, system will look for “ActivityPath” in that particular action object, find it on file system and execute it.

In .xaml files we can execute wide range of actions. Most commonly, it’s used for executing API calls/commands.



.Xaml file looks like this. This one is used for transition from the Review State of balance transfer to the Status State of the balance transfer process.

It first invokes method which will take data from the work item, populate and send request to the API. This particular activity is a generated activity. Digital edge solution references a library AssecoSEE.Rest.Activities which has all the methods needed to invoke all of the web api endpoints.

The developer should use these methods for calling the api, because it’s cleaner and easier to do, but also custom code activity method can be created and then invoked from the .xaml file.

“ActivityPath” in action is populated only with the file name of the transition that needs to be executed, so in order for this to work, .xaml files are put in a predefined location by convention:

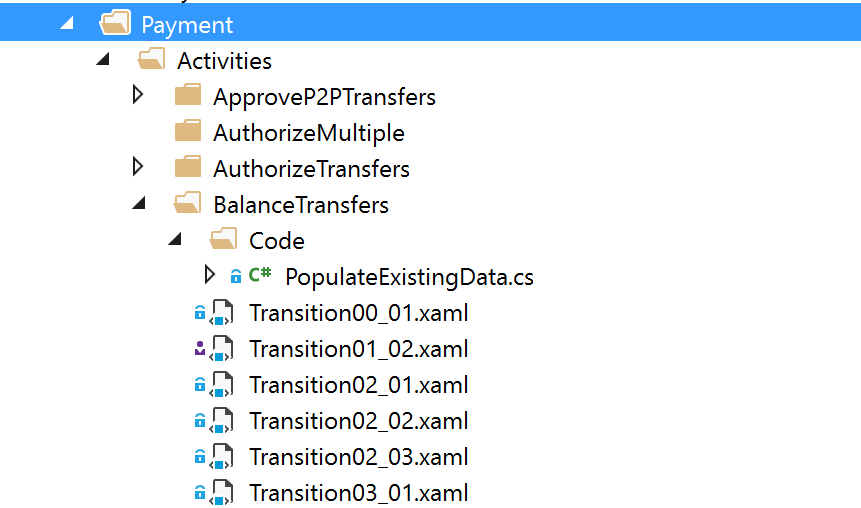
Areas/{Area}/Activities/{ProcessName}/Transition02\_03.xaml

As you can see, by convention, the developer should name the transition in the similar fashion. In the name of the transition above, we can see that it’s a set of methods that are executed in a transition from state 02 to 03.

Custom Code activities are placed in the following location:

Areas/{Area}/Activities/{ProcessName}/Code/MyCustomCodeActivity.cs

Example:



## Process Managers

Every process has to have his manager. Manager is basically a wrapper around the process definition, and it’s used for dependency injection – they inherit from its own interface and from ProcessManager

public class BalanceTransferProcessManager : ProcessManager, IBalanceTransferProcessManager

IBalanceTransferProcessManager inherits from IProcessManager, which is used to easily configure which process definition will be used for the same route, through unity dependency injection.

Dependency injection here is important for easy switching between implementations of different processes used on the same route values – based on the particular requests for the client we are developing for, which will be explained later in the document.

By convention, process managers are placed and named as: Areas/{Area}/ProcessDefinitions/Managers/{NameOfTheProcess}ProcessManager.cs

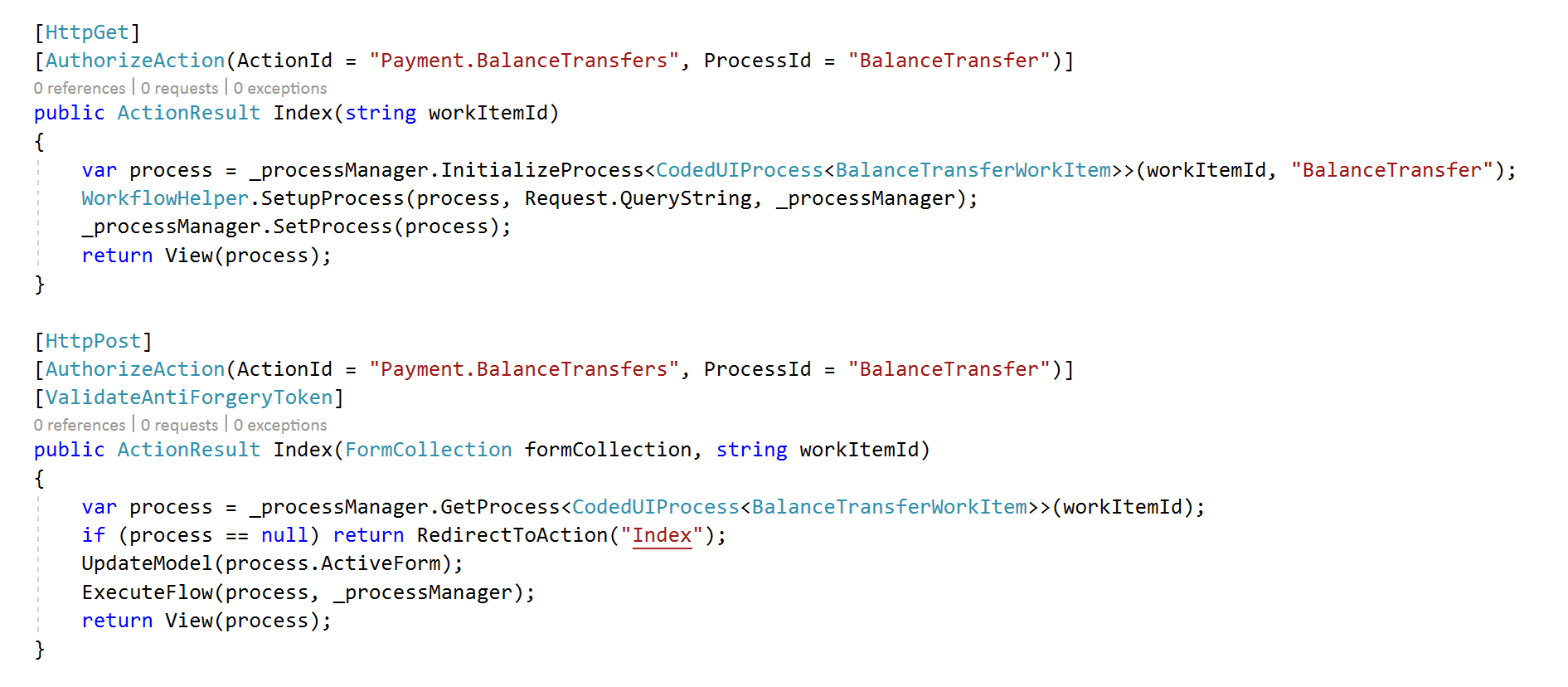
## Controllers

For each new process a new MVC controller should be created.

By convention it should be named same as the process it is working with, i.e.:

{NameOfTheProcess}Controller.cs

Controller that handles with the process, should always contain the same code. No custom code should be written in the controllers in Digital Edge solution. For example:

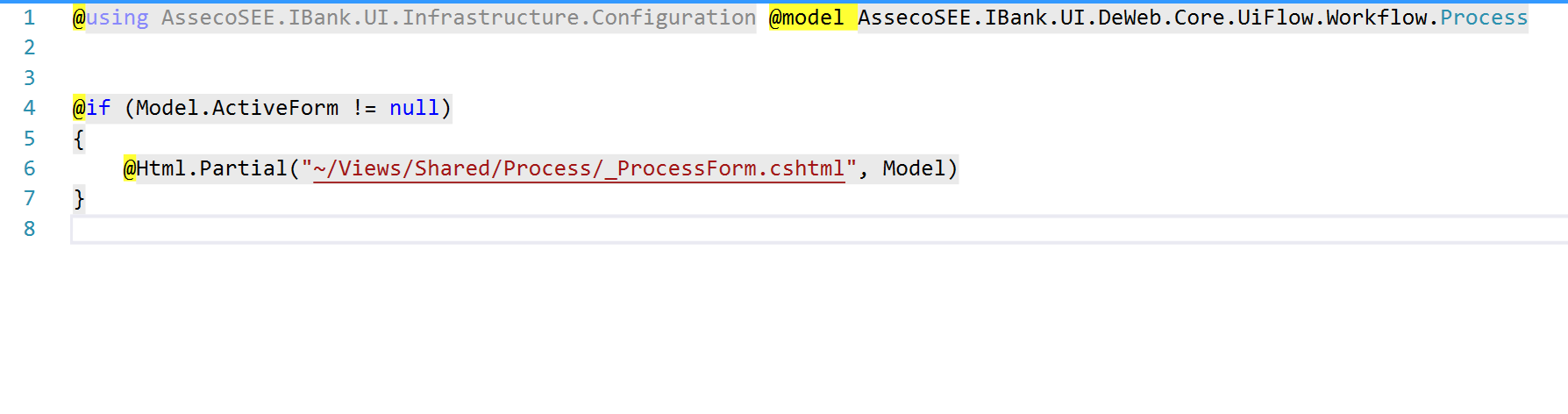


## Views

Views folder solution structure should be organized, again, by MVC framework convention, but the steps views used for rendering different states of the process should be placed in “Steps” folder. i.e.:

Areas\{Area}\Views\{ControllerName}\Steps\{NameOfTheTemplateDefinedInFormDefinition}.cshtml

If the rendering of the process will be generic (which it should be, with exception to some extreme cases), “Step” cshtml file should contain only the following code:

Everything else is handled by Digital Edge framework.

# Adaptation - implementation of Digital Edge solution

Digital edge uses nuget packaging system to deploy and version different parts of the solution.

Nuget packaging and how to use the project which has Digital Edge nuget packages is explained in other document, named ‘Configuring and working on a web project that uses DeWeb nuget packages’