

**Workflows**

© CourtView Justice Solutions Inc.

All Rights Reserved.

Information contained in this document is proprietary to CourtView Justice Solutions Inc, and may be used or disclosed only with written permission from CourtView Justice Solutions Inc. This document, or any part thereof, may not be reproduced without the prior written permission of CourtView Justice Solutions Inc. This document refers to numerous products by their trade names. In most, if not all cases these designations are claimed as Trademarks or Registered Trademarks by their respective companies.

**ShowCase, CourtView, DAMION** and **JuryView** areregistered trademarks of CourtView Justice Solutions Inc.

This document and the related software described in this manual are supplied under license or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. The information in this document is subject to change without notice and does not represent a commitment on the part of CourtView Justice Solutions Inc.

The names of companies and individuals used in the sample database and in examples in the manuals are fictitious and are intended to illustrate the use of the software. Any resemblance to actual companies or individuals, whether past or present, is purely coincidental.

Table of Contents

[Introduction 4](#_Toc372539738)

[Workflow Orientation 4](#_Toc372539739)

[.NET Workflows – 3.0 and 3.5 5](#_Toc372539740)

[CallMethodActivity 8](#_Toc372539741)

[Xoml and rules files 9](#_Toc372539742)

[.NET Workflows – 4.0 + 10](#_Toc372539743)

[ShowCase Class Design 11](#_Toc372539744)

[Loading, Clearing and Saving 14](#_Toc372539745)

[Assemblies and namespaces 15](#_Toc372539746)

[Hands on - Creating a ShowCase Workflows 17](#_Toc372539747)

[Workflow Workshop Project 17](#_Toc372539748)

[.NET 3.5 workflow 17](#_Toc372539749)

[.NET 4.0 workflow 19](#_Toc372539750)

[Workflow Management in ShowCase 21](#_Toc372539751)

[Smart Docketing 22](#_Toc372539752)

[Docket Codes 23](#_Toc372539753)

[Docket Fields 24](#_Toc372539754)

[SD Functions 26](#_Toc372539755)

[Putting it all together 28](#_Toc372539756)

[Data Exchange Services 30](#_Toc372539757)

[ExchangeRequest and ExchangeResponse 31](#_Toc372539758)

[GetShellCase 32](#_Toc372539759)

[DES Request Authentication 32](#_Toc372539760)

[DES Agent Workflows 34](#_Toc372539761)

[Summary 36](#_Toc372539762)

# Introduction

**W**orkflow means orchestration. It is the orchestration of tasks, processes, actions etc. The term workflow is abstract and applies as a general idea to every process. A workflow is made up of individual components that are clearly defined and have a specific purpose. These components are grouped together in a sequence, in parallel, in a loop etc. These individual components and orchestration tools together make up a workflow system.

The term workflow is used quite loosely in every line of business (LOB) application. This does not mean that the application has a good workflow system. A good workflow system:

1. Allows definition of clearly defined atomic components.
2. Provides various tools to orchestrate the components.
3. Provides a graphical view of the components and how they are orchestrated in a workflow.
4. Provides capabilities to dynamically change the definition of a workflow.

A clear understanding of what workflow means and the capabilities of a workflow system is very important for any one creating or using a LOB application.

# Workflow Orientation

Traditional software design methodologies cover all major areas of an application such as User Interface, Business or middle Layer, Database design etc. But they do not cover the best practices in how these areas of an application interact to make the application. After all, majority of the application code is about defining business processes that involves orchestration of all these areas. Workflow orientation is an addition to the design thinking of traditional software development.

With this shift in design thinking, the creators of an application are putting significant thought into how to create a good workflow system for the application. This will make the application flexible and adaptable to changing business practices and rules.

Workflow orientation also requires a comprehensive understanding of frameworks that provide the tools to create a good workflow system. .NET framework for example provides Windows Workflow Foundation. This foundation provides the graphical tools and classes that allow any application based on .NET framework utilize the workflow principles. For detailed history on the evolution of workflow orientation refer to this Wikipedia article: <http://en.wikipedia.org/wiki/Business_Process_Execution_Language>

Microsoft originally introduced Windows Workflow Foundation in version 3.0. Later, this foundation was completely revamped from ground up in version 4.0. ShowCase uses workflows created in both the versions of .NET framework. The workflows created in version 3.0 and 3.5 are compatible with the newer versions of the framework.

In the next section we will learn about how workflows are designed and constructed in both the .NET framework versions.

# .NET Workflows – 3.0 and 3.5

.NET frameworks 3.0 and 3.5 provided 2 different types of workflows - Sequential Workflow and State Machine Workflow. Sequential workflows are used to implement a process that requires a simple step by step execution in a sequence. Notice the style below. This style of execution is similar to how code is written in a typical application.

Execute A

Execute B

IF cond = True THEN

Execute C

ELSE

Execute D

END IF

State Machine workflows are used to implement a process that requires transitions between different states. Notice the style below in Figure 1. This is quite different than a sequential flow, because the transitions between states can make up quite complex patterns. Such complex patterns are sometimes impossible to represent in a sequential workflow.

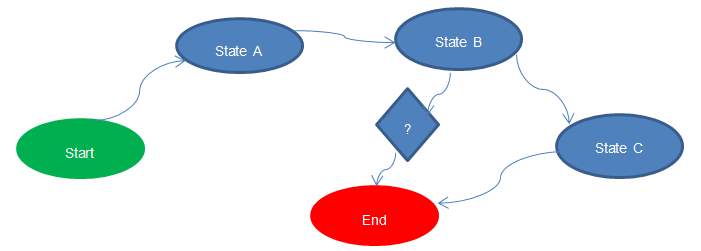


Figure-: A state machine workflow

ShowCase primarily uses Sequential Workflows due to the nature of the processes it automates. These processes follow a simple sequential style of execution and are best fit for sequential workflows. Each step in a sequential workflow that is executed is called an **Activity**. An activity represents a block of work that is implemented using few lines of code. An activity is also constructed in a re-usable fashion so it could be used in several workflows. An activity is what we consider an atomic component. A workflow is simply an orchestration of such activities. A large number of such activities in an application will help in automating business processes of an application.

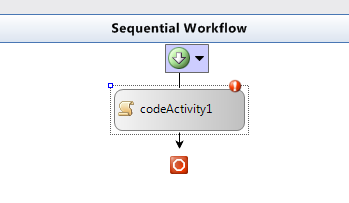
In .NET framework 3.0 and 3.5 such activities can only be built with code. There is also a generic **CodeActivity** that allows the developer to write any code when it is used in a sequential workflow. In Visual Studio’s workflow designer, the CodeActivity can be dragged and dropped onto the designer and looks like shown in the Figure-2.

Figure -2: Code Activity

You can right click on the activity and chose the option “View Code” to generate a class file that holds the code for this activity. The code block for this activity will look like Figure-3.

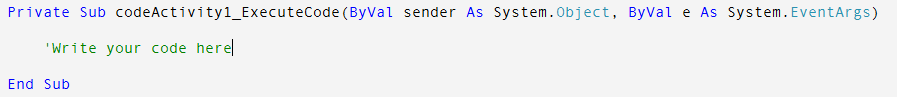


Figure-3: Code behind for a CodeActivity

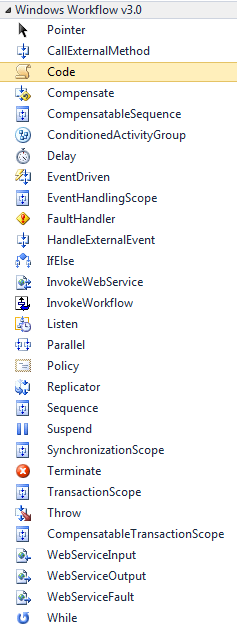
There are several other activities that can be used in a .NET 3.5 workflow that provide programming constructs to define a sequential workflow. Figure-4 lists most of those activities. Each activity provided out of the box helps to provide a specific programming construct that is typically available to vb or c# code. For example:

Figure-4: Activity toolbox

**IfElseActivity** provides the “IF…THEN…ELSE” construct.

**WhileActivity** provides the “WHILE” loop construct.

**InvokeWebServiceActivity** provides the programming construct to create a web service client and invoke a web service method.

**ParallelActivity** provides the programming construct to execute several child activities in parallel.

**TerminateActivity** terminates the execution of a workflow.

**ThrowActivity** allows you to throw an exception from the workflow.

A detailed description of each activity is beyond the scope of this discussion, but several resources and textbooks can be found online to gain a thorough understanding of workflow foundation in .NET framework 3.0 and 3.5.

Once a workflow is constructed using .NET 3.5 it is compiled into the application code. The framework provides classes under **System.Workflow** namespace to execute the workflows. The workflows are executed in an isolated runtime that is different from the runtime of your application. But both the application and workflows are executed in the same process or app domain of your application.

.NET 3.5 provided us with all necessary tools to create a good workflow system, except for one. If you recall from the introduction section, there are at least 4 characteristics of a good workflow system. The 4th characteristic was the ability to dynamically change the definition of a workflow. This is clearly not provided by the framework. Any workflow would have to be compiled into the application code to be executed by the workflow runtime. This means each time the definition of a workflow is changed; it required a new version deployment of the application.

Even though the framework did not provide an out of the box solution to be able to construct a dynamically changeable workflow, it provided us with tools that we could use to create the solution. To create such workflow, it needed to meet the following criteria:

1. The workflow should not be expressed in code. It should only be expressed in xml.
2. The workflow should not be compiled into code. Because changes to a workflow compiled into application code will require a new version of the application to be released.
3. The application needs a management system to store and retrieve the definition of a workflow and also execute it in the workflow runtime.

A .NET 3.5 workflow is expressed in xml, specifically **xoml – eXtensible Object Markup Language.** This xml follows the BPEL (Business Process Execution Language) standard. Only if the workflow utilizes activities that require code to be written, a code file is created and the workflow would need to be compiled. For example, when a new workflow is created, its file name could be something like “TestWorkflow1.xoml”. This is an xml file. But when we use a code based activity in this workflow, Visual Studio will automatically create a companion file called “TestWorkflow1.xoml.vb” to hold the code that defines that activity.

To be able to express a workflow in pure xoml (meaning no code beside file), we need a custom activity that can be used to dynamically execute pre-existing code. In the .NET world, this can be done via *Reflection*. Any method of any class can be invoked at run time using reflection. This technique can be used to build a custom activity to use in workflows.

## CallMethodActivity

The *CallMethodActivity* is a custom built activity that is used to invoke methods on classes using reflection. Since it is a custom workflow activity, when used in a workflow, it is fully expressed in xoml format. This activity is one of the basic building blocks of ShowCase workflows. All ShowCase workflows built using .NET 3.5 framework were possible only due to this activity.

The properties exposed by CallMethodActivity can be seen in the Figure-5 below.

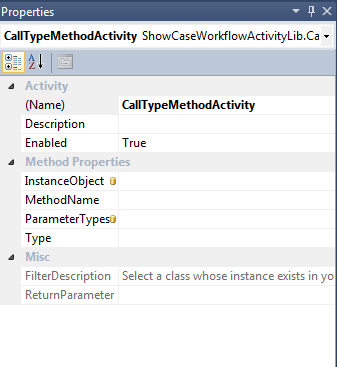
**Type** – This is the name of the class including the assembly name. For example, ShowCaseClassLib.CCase.

Figure-5: Properties of a CallMethodActivity

**ParameterTypes** – A comma separated list of input parameters that the method accepts. This is only needed if you are calling an overloaded method (a method that has the same name but different input parameters).

**MethodName** – This is the name of the method to invoke on the class specified above.

**InstanceObject** – This is the reference to the object of the specified type.

**Parameters** – This is a list of parameters that are required for the method. This list is dynamically populated in the designer when the method name is selected.

When the workflow is executed, this activity simply invokes the method on the class specified with the included parameters.

## Xoml and rules files

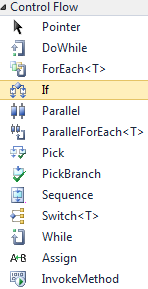
The workflows created using .NET 3.5 framework will have a file extension of “.xoml”. Additionally a file with extension “.rules” is also created in some instances. In a workflow, the activities such as IfElseActivity, PolicyActivity, WhileActivity will require code to evaluate conditions. Such code is automatically translated to xml and is maintained in a separate companion file with same name but with “.rules” extension.

The ability to express conditional statements in code as xml is specific to .NET framework, specifically .NET rules engine. This is another tool in the workflow system that provides us the ability to construct a pure xoml workflow without any code file associated.

# .NET Workflows – 4.0 +

In .Net framework version 4.0 the windows workflow foundation was completely revamped. Several improvements in the Architecture, Designer and runtime were made. As a result the entire workflow development experience is different. We will discuss some of the major changes done to the foundation in this section.

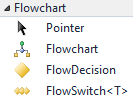
A workflow in the new version is called an Activity. An Activity can contain any number of child activities. This is the primary architectural difference between the 2 versions. There are no Sequential or State Machine workflows like in the old version, everything is an Activity. Several new activities are added to the toolbox to improve the way workflows are implemented. Take a look at the Figure-6 below that lists some of the basic control flow activities.

The most basic activity is the **SequenceActivity** which simply allows multiple other activities to be addedto the designer to form a sequence. This is similar to the SequentialWorkflow in the previous version.

**ForEachActivity** is a generic activity that provides the “For” loop construct in typical vb.net or c# code. Similarly you also have a **ParallelForEachActvity** that allows each loop to be executed in parallel.  
The **PickActivity** provide us with the “IF … THEN … ELSEIF … ELSEIF … ENDIF” construct.

**SwitchActivity** provides us with the “SELECT CASE” construct.

**AssignActivity** is used for assignment of variables; x = y where x is the variable and y could be any function call or simply another variable. The call to a function is specified using vb.net syntax.   
  
The **InvokeMethodActivity** is used to execute any method on any class using reflection. This is similar to the custom built **CallMethodActivity** that we discussed earlier.

**FlowchartActivity** is in a way similar to SequenceActivity which is used to group several activities together. But it provides a way to group activities using familiar flow charting techniques.

Improvements and modifications have been made to the designer that will now allow for an easier workflow development experience. Workflow files in .NET 4.0 have an extension of “.xaml”, which stands for **“eXtensible Application Markup Language**”. There are no other companion files along with a xaml file. All code and business rules are expressed in xaml. These workflows can be compiled into the application or simply loaded and executed in the workflow runtime dynamically. This framework is perfect for our definition of a good workflow system.

Figure -6: WF4.0 Activities

So far we have discussed the concepts of a workflow system and the tools that make it. A good workflow system enhances the flexibility of an enterprise application and provides a robust mechanism to understand, implement and adapt to business requirements. The enterprise application itself should be designed in a way to allow for such flexibility; specifically, the middle layer of the application.

A workflow is an orchestration of existing atomic components. In an application, the atomic components live as methods or functions in the classes defined for the middle layer. A workflow therefore simply orchestrates necessary functions or methods of certain classes. In the next section we will take a detailed look at the ShowCase class design.

# ShowCase Class Design

The middle layer of ShowCase is constructed as an ORM (Object Relational Mapping) layer. ShowCase’s ORM layer is a simple 1 to 1 model with respect to database model. For every table in the database, there is a corresponding class with exact same definition and associated collection class. Take a look at the Figure-7 below:

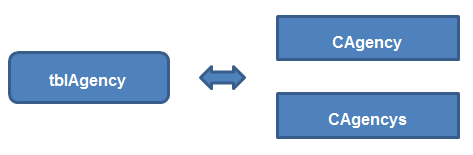


Figure-7

A database table called “**tblAgency**” is represented as 2 different classes in ShowCase – **CAgency** which is called a single class, and **CAgencys** which is called a collection class. The single class contains a property definition for every column in the corresponding table. The collection class is used to hold multiple instances of a single class. A collection class’ name is constructed by simply appending the letter “s” to the single class name. You will therefore notice collection class names that are spelled incorrectly, for example CPartys. The letter “C”, which denotes a class, is a prefix for every class name in the ShowCase ORM layer. Similarly the word “tbl” is prefixed to every table name and “vew” is prefixed to every database view name.

Every single class and collection class is derived from 2 base classes that hold majority of the logic to load and save its data into the database, as shown in Figure-8.

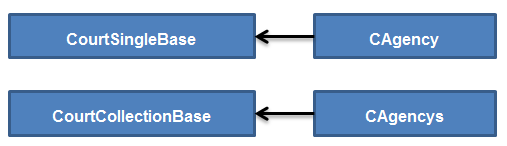
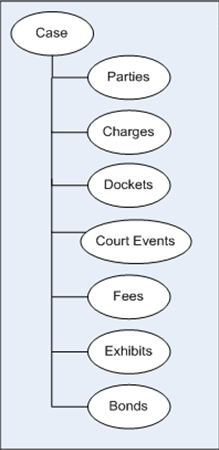


Figure-8

The foreign key relationships in the database are also represented and maintained in the class design using simple navigation properties. Navigation property is named the same as the child object’s collection class name without the prefix “C”. For example, the **Charges** property on the class **CCase** is used to represent the “one to many” relationship between the case and charge tables. Accessing the Charges property will return an instance of the class **CCharges** that hold one more **CCharge** objects.

Take a look at the Figure-9 to see some of the different entities that are related to a case. In ShowCase, a vast majority of entities have a “one to many” relationship to the case. Due to this the CCase is considered the root class in the design hierarchy. As a result all case related entities can only be populated and accessed by navigating through the CCase object.

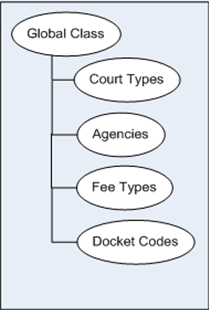
ShowCase classes are designed using a strict parent-child relationship structure. Child entities are accessed by navigating through the parent entity. This is called top-down navigation. The design also allows bottom-up navigation which is to get to the parent entity from a child entity. For example to navigate to the 5th docket on a case, one would use the following code:

Figure-9

*CDocket d =* ***CCase.Dockets[4];***

To navigate back to the parent case from a docket object, the following code would be used:   
*CDocket d = CCase.Dockets[x];  
CCase c =* ***d.ParentDockets.ParentCase****; OR* ***d.Parent.Parent****;*

So far in this section we have established that the CCase class as the root of many entities in the ShowCase class design.

But ShowCase has several other classes that represent the lookup tables used in the system. There are over 100 lookup tables used by ShowCase. These classes are not accessed via the CCase class. Instead all lookup table classes are accessed via a special class called CGlobal, also called the Global Class. The primary purpose of the global class is to make the lookup tables available to general ShowCase programming. Take a look at Figure-10.

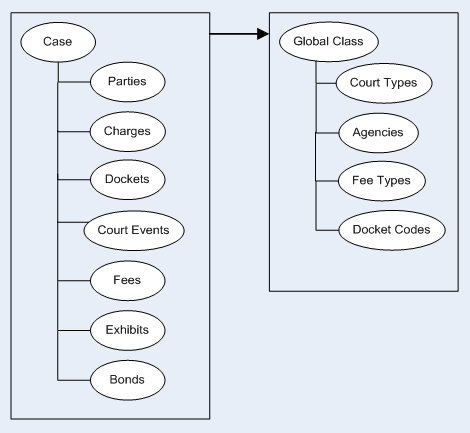
All lookup table classes are rooted at the global class. So to access a lookup table from within any other class, a reference to the global class is necessary. The ORM layer of ShowCase is designed in such a way that every class that represents a database entity (that is not a lookup table) contains a navigation property called **GlobalClass** that is used to obtain the reference to the CGlobal object and there by navigating to the necessary lookup table. Take a look at the Figure-11 that explains this hierarchy.

Figure-11

Figure-10

So for example, to access the fee types lookup table from the Fee class (CCaseFee) the navigation code will look like **this.GlobalClass.FeeTypes;** where the keyword **this** represents an instance of CCaseFee class.

The details learnt so far are the fundamental concepts of the ShowCase class design. These concepts are visible in every area of the ShowCase application and therefore are very important to comprehend.

## Loading, Clearing and Saving

This section explains how classes in ShowCase are loaded with data from the database, persisted to the database, and how they are cleared when necessary.

To load a single class in ShowCase, simply call the **BaseLoad** method by passing the database identity value of a row in the table. This function is defined in the base class CourtSingleBase.

|  |
| --- |
| **Public Function BaseLoad(ByVal ID As Integer) As Boolean** |

To load a collection class in ShowCase, call the overridable **Load** method defined on the base collection class CourtCollectionBase. Many classes have their own overloaded versions of this Load method. There is also an overloaded version of this method that can be used to load a collection class from a dataset.

|  |
| --- |
| **Public Overridable Function Load() As Boolean** |
| **Public Overridable Function Load(ByVal pDS As DataSet) As Boolean** |

In practice, the need to load a specific object in ShowCase is rare. Majority of ShowCase programming starts off with the CCase object. When working with this root object, the first thing to do is to load the case. This is done in code as shown below:

|  |
| --- |
| **CCase myCase = new CCase(); myCase.BaseLoad(12345); OR myCase.Load(12345)** |

The Load method of the CCase class automatically loads all its related entities such as Parties, Dockets, Charges etc. All objects necessary are already pre-loaded when the case is loaded which is why the need to load a specific object is pretty rare. This automatic loading of related entities is called eager loading. The alternative to this is lazy loading which only loads child objects when the corresponding navigation properties are accessed. ShowCase does not directly support lazy loading yet; although the class design can be used to simulate this type of loading behavior.

In some situations it is necessary to clear object variables so they can be re-loaded with different data. To clear an object’s state in memory simply call the overridable Clear method defined in both the single and collection base classes.

|  |
| --- |
| **Public Overridable Sub Clear()** |

Finally, to persist an object’s state to the database, call the overridable Save() method on CourtSingleBase. The Save() method is not available for collection classes. Majority of ShowCase classes override this method to perform special business logic before and after the object is saved into the database. In some situations, when an object needs to be saved to the database without executing any of the business logic, call the BaseSave() method.

|  |
| --- |
| **Public Overridable Function Save() As Boolean** |
| **Public Function BaseSave() As Boolean** |

## Assemblies and namespaces

There are a large number of classes that provide functionality to different areas of ShowCase. These classes are organized into related namespaces and assemblies. In addition to understanding the basics of ShowCase class design, it is very important to understand what assemblies are available and their functional aspects. The table below lists the basic assemblies needed to create workflows and a brief description of their purpose. Any User Interface specific assemblies are not listed here.

|  |  |
| --- | --- |
| **Assembly** | **Description** |
| BaseClassLib | Holds all the base classes such as CourtSingleBase, CourtCollectionBase, CGlobal and other utility classes. |
| BaseSQLDAO | Contains core database access classes and utility classes that are used for database operations. |
| ShowCaseClassLib | This is the largest assembly in ShowCase. This assembly contains the classes that make up the ORM layer of ShowCase. |
| ShowCaseCCB | This assembly contains extension methods that act as delegates to invoke user interface forms from inside a workflow. |
| ShowCaseDAL | This assembly contains classes and interfaces that define the data access layer of ShowCase. |
| ShowCaseUtil | This assembly holds utility and helper classes for features such as Logging, Serialization etc. |
| ShowCaseWorkflowActivityLib | This is the main assembly for all custom workflow activities such as CallMethodActivity, SendEmailActivity etc. |
| ShowCaseWorkflowLib | Contains a single class ActivityRuntimeManager that is used to execute WF 4.0 workflows. |
| ImagingClient | This assembly contains all the classes necessary to work with images on a case. |
| ImagingUtil | This assembly contains all the utility classes that can be used to manipulate an image. |
| Security.Authorization.Claims | This contains classes built to handle roles and permissions which are expressed as Claims by the Authentication service. This assembly is needed when creating workflows that authenticate against the ShowCase system. |

So far, we’ve learned quite a bit about ShowCase, the internal design as well as some of the important assemblies that are required to build workflows. Now let’s take a look at the current architecture of ShowCase and the components that make up the system. Refer to the Figure-12 below.

**ShowCase Client**

**Figure-11: ShowCase evolution as an SOA framework**

A thick client application is continuously evolving into a service oriented distributed enterprise system. With this evolution the inner workings of ShowCase are susceptible to change. The goal is to ensure this evolution is least disruptive and to make sure the product is backward compatible.

# Hands on - Creating a ShowCase Workflows

There are 2 different kinds of workflows that can be created for ShowCase. One which is used by the client application in features such as Smart Docketing, Judge/Division Assignments, Auto Docketing, InCourt Processing etc. The other kind is workflows that are used by the DES (Data Exchange Services) sub-system. Let us look at the workflows used by ShowCase in the next few sections. DES workflows will be discussed later in a separate section.

ShowCase workflows are used to automate case level processing by using methods and functions defined in various classes and they always work on a specific case. A CCase object is always passed in to the workflow as an input parameter and the workflow can have additional parameters.

A Visual Studio Solution called WorkflowWorkshop is provided to developers that enables them to create new workflows or edit existing ones. This solution can be used to work with both WF 3.5 and WF 4.0 style of workflows.

## Workflow Workshop Project

The WorkflowWorkshop solution contains 2 child projects - **WorkflowLib** that contains references to all necessary ShowCase assemblies and **WorkflowTestBed** which is a simple console application that can be used to test and debug workflows.

|  |
| --- |
| **Note: Only WF 4.0 workflows allow debugging** |

## .NET 3.5 workflow

To edit a WF 3.5 workflow, simply double click on the file with a “.xoml” extension inside the WorkflowLib project. This opens up the designer which allows you to modify the workflow. You can also right click on the file and open it in an xml editor to see the xml definition of the workflow. Please note some WF 3.5 workflow will have a companion file with same name but a “.rules” extension. This file contains the xml definition of conditional statements defined inside the workflow.

To create a new WF 3.5 workflow, copy the file with name “TemplateWorkflow.xoml” inside the **TemplateWorkflows** folder and paste it into any necessary sub folder. The template workflow is a pre-constructed workflow that helps reduce certain mundane steps required to create a new workflow. Rename the new workflow file name. Additionally you must also provide the same name in the xml definition of the workflow. Right click on the new workflow file and open it in an xml editor. Identify the xml attribute “**x:Name**” and change its value to the appropriate workflow name.

A WF 3.5 ShowCase workflow is derived from **ShowCaseWorkflowActivityLib.BaseWorkflow** class. This is the class that provides all functionality necessary to maintain a workflow’s definition in pure xoml. The BaseWorkflow class contains the **Case** and **GlobalClass** properties that are 2 root classes of ShowCase class design. These 2 properties will be automatically populated by the ShowCase client application with the case the user is working on. Listed below are all the properties defined for WF 3.5 ShowCase workflow.

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| Description | String | A brief description of the workflow |
| Case | ShowCaseClassLib.CCase | The root case object that this workflow is executed for. This is automatically supplied by the ShowCase framework when this workflow is executed. |
| GlobalClass | ShowCaseClassLib.CGlobal | The root global class that contains references to all lookup tables. |
| InputParametersFormat | String | A comma separated list of input parameters that the workflow requires. This must be set in the format – “param1:System.Int32, param2:System.DateTime” |
| EnableCaseTransaction | Boolean | Set to true if all the actions done by the workflow should be wrapped inside a database transaction. If the workflow throws an exception the entire transaction is rolled back. |
| Result | System.Object | The return value of the workflow. This property can be set to indicate a return value after the workflow executes. |

There are other important properties that are useful while creating the workflow.

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| WorkflowLocals | ShowCaseWorkflowActivityLib.WorkflowLocals | A dictionary object to hold temporary local values inside a workflow |
| DefaultDateTime | System.DateTime | Returns DateTime.MinValue |
| DefaultDecimal | Decimal | Returns 0D |
| DefaultFalse | Boolean | Returns False |
| DefaultTrue | Boolean | Returns True |
| DefaultInteger | Integer | Returns 0 |
| DefaultString | String | Return String.Empty |

The WorkflowLocals object contains a dictionary object to hold local variables and their values while the workflow is executing. This is analogous to defining temporary variables inside a method or function definition. The variables inside a function do not persist and are not accessible beyond the scope of the function.

The “Default” properties are a way to specify default values to certain variables in a workflow. WF 3.5 workflows do not provide an elegant way to define default values. Hence the need for such default properties.

## .NET 4.0 workflow

WF 4.0 workflows are a lot different from the previous version. But they are a logical evolution. A lot of concepts we have learned still apply. Many of the workarounds we used in the previous version are supplied out of the box in the newer version.

To create a new workflow, add a new item to the project. In the Add New Item form, select the “Workflow” category on the left and choose the “Activity” item. This will create the workflow and will open in the designer.

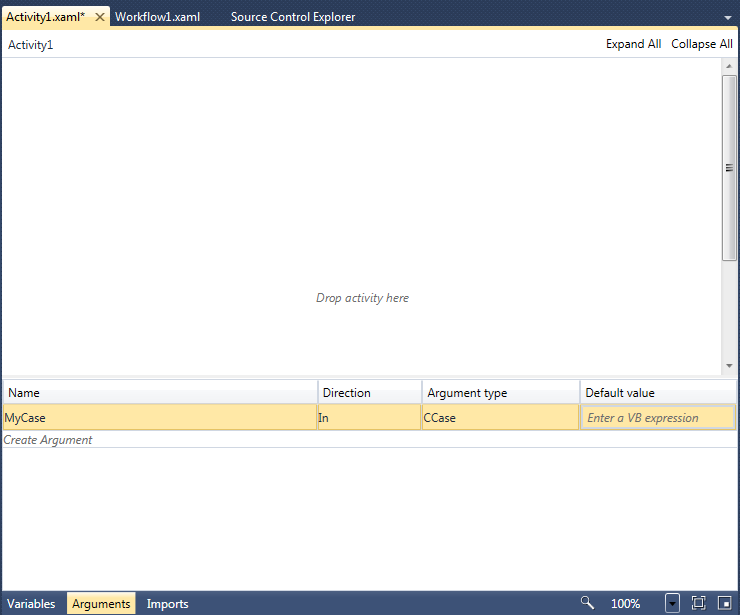


Figure-13: WF 4.0 Designer

A WF 4.0 workflow contains a section where input arguments can be specified. An input argument of type ShowCaseClassLib.CCase must be defined for every ShowCase workflow. This was automatically done in WF 3.5 version, but in the newer version this is a manual step.

Any local variables that are going to be used in the workflow are added to the Variables section. Compare this to how we defined the local variables using the WorkflowLocals property of a WF 3.5 workflow.

A workflow file’s BuildAction property must always be set to “None” as we do not want the workflow definitions to be built into the application. This applies to both versions of the workflow file. In fact, if a WF 3.5 workflow file’s BuildAction is not set to None it will cause compile time errors.

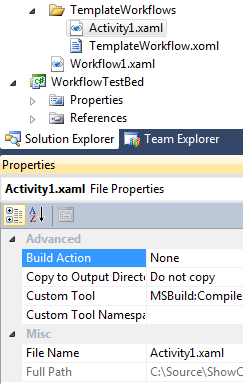


Figure-14: Workflow file BuildAction property

# Workflow Management in ShowCase

All the workflows created for ShowCase must be added to the ShowCase database. This is done using the workflow management tab in the Admin form. You can access the form by navigating to Admin -> Admin menu item in ShowCase. The form is as shown in the Figure-15 below.

A workflow is ShowCase is called a **ProcessWorkflow**. A ProcessWorkflow is categorized or assigned to a ProcessWorkflowCategory. This categorization helps ShowCase organize all the workflows into respective functional areas. The form shown below provides functionality to add new categories and new workflows to the system. An existing workflow’s definition can be saved to the local system using the “Export Definition” button. Process Workflows that are specifically designed to be called from ShowCase code will have their Program Control (an enumeration value) set.

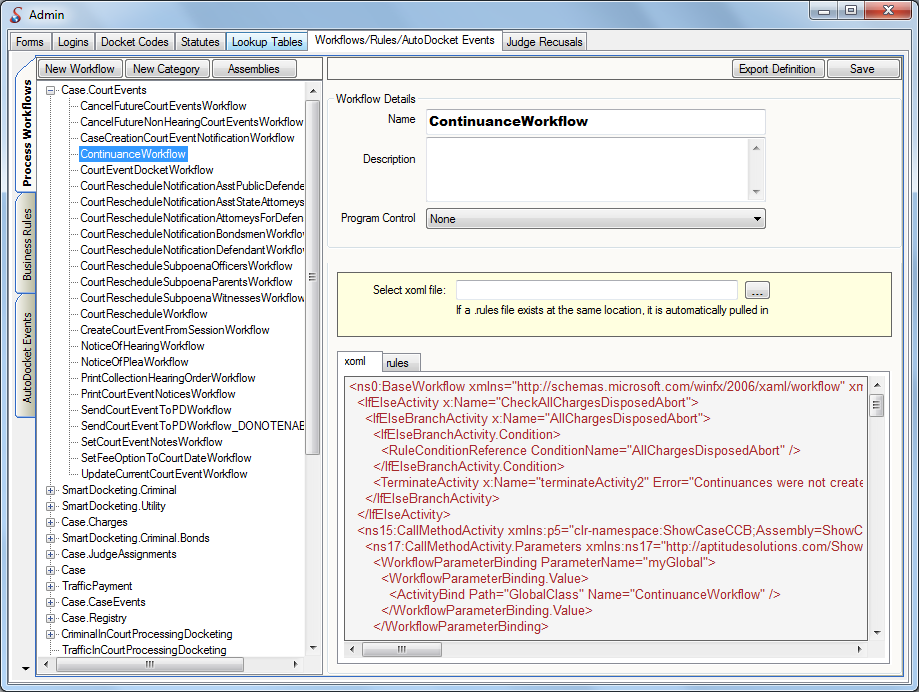


Figure-15: Workflow management form in ShowCase

# Smart Docketing

“Docketing” is analogous to logging of actions performed in a system. In ShowCase users can manually create docket lines when they take certain actions. ShowCase also automatically dockets after certain pre-defined actions such as scheduling a court date, or adding a charge to a case. Smart docketing is the exact opposite of this process. Smart Docketing is executed with automatic actions when certain docket entries are entered and/or created.

Workflows play an important role in the smart docketing process because that is where the actions are defined. To define a smart docket, we first need to define a docket code.

## Docket Codes

To enable a docket code for smart docketing, the flag “Smart Docket Active” must be set to true. Furthermore, it must be linked to one or more actions or functions that will need to be executed. These actions are referred to as “Smart Docket Functions” or simply SD Functions. Take a look at the Figure-16 below to see how a docket code is setup as a smart docket.

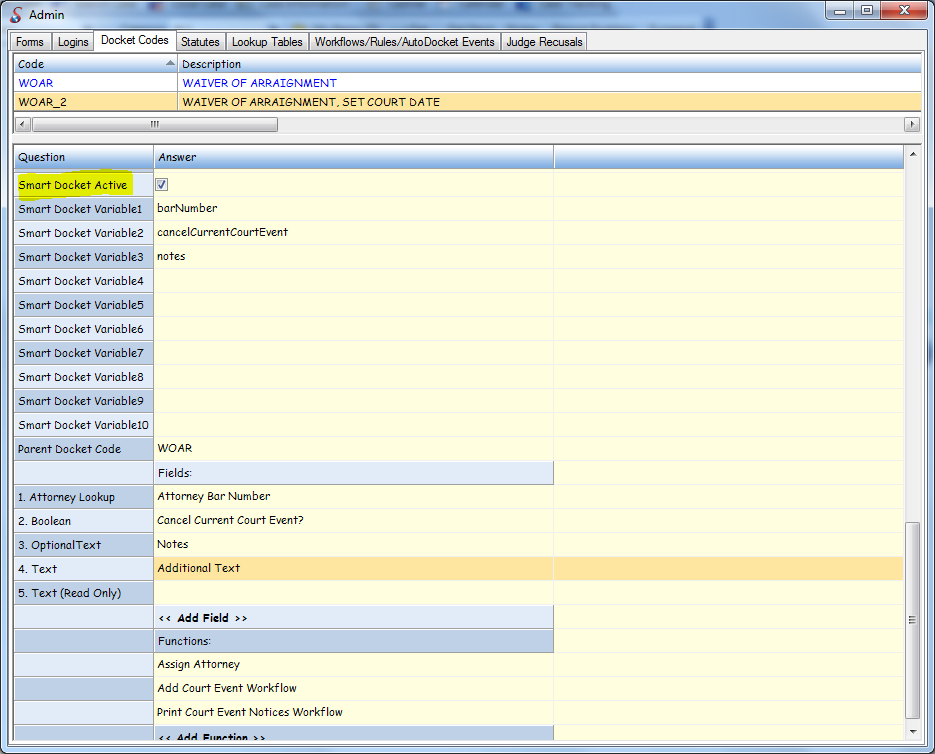


Figure-16: Docket Code Admin Form

Any number of SD Functions can be added to a docket code. All the attached functions are executed sequentially in the order they appear.

A docket code can also be configured with docket fields. A docket field serves as a mechanism to capture input values from the user so they can be supplied to the SD Functions attached to the docket code. For example, in the figure above, the first function “Assign Attorney” requires an attorney’s bar number to be entered by the user. For this purpose, a docket field of type “Attorney Lookup” is configured for this docket code. We will take a detailed look at the SD Functions and Docket Fields in the next few sections.

## Docket Fields

A docket field is a system defined field that can be used to capture input from the user for any smart docket. Common fields such as a “Text”, “Boolean”, “DateTime” etc are readily available. There are several other system defined fields available to use in smart docketing. Table below lists the most commonly used fields. For a full list of all the fields, please refer to the Docket Fields lookup table in the Admin section of ShowCase. The behavior of a docket field is dependent on the type definition of the field – specifically the CalcType. The CalcType dictates how a field behaves when used in smart docketing. There are 15 different types of docket fields defined by ShowCase. Table below provides the definition of these types. New docket fields can be added to the system as long as they belong to one of these pre-defined CalcTypes.

|  |  |
| --- | --- |
| **CalcType (Field Type)** | **Description** |
| 0 | Plain text field |
| 1 | Lookup field. User input is validated against a lookup table based on following criteria:  Calc1 – Name of the lookup table to search e.g, *Agencys*  Calc2 – Function (on the lookup table collection class) that does the lookup  Calc3 – Property (on the lookup table class) that serves as return value of the field |
| 2 | Format Date – user input is formatted to a date time. An error is thrown if the input cannot be converted to a date time. |
| 3 | Static List – Provides user with a static list of values to pick from.  Calc1 – A pipe delimited list of static values. Putting a pipe character at the beginning of the string will add an empty item in the dropdown. |
| 4 | Static Default Text – The value typed in Cacl1 field is the return value of the field. |
| 5 | Calculated Default Text – Performs a calculation to identify the default text.  Calc1 – name of the class  Calc2 – method or property name  Calc3 – Specify if Calc2 is a method or property. 1 for method, 2 for property |
| 6 | Same as CalcType 3, but if the selected value is the same as that specified in Calc2, the ImageRequired property of the docket code will be set to true. This means if the user selects a desired value the system will make sure a document is scanned before executing any functions. |
| 7 | Create a word merge document if the user asks for it..  Calc1 – A static list, typically “Yes|No”.  Calc2 – One of the options in Calc1, typically “Yes”. This means if the user chooses “Yes”, then run the word merge specified in Calc3.  Calc3 – Program control name of a word merge document. |
| 8 | Provides the user with a dropdown list of values. The list is populated using a dictionary object obtained from:  Calc1 – name of the class  Calc2 – name of the function that returns a dictionary.  The return value of this field is the Key value used in the Dictionary. |
| 9 | Boolean |
| 10 | Same as calc type 8, with one of the values pre-selected.  Calc1 – name of the class  Calc2 – name of the function that returns a dictionary.  Calc3 – Name of a docket field of CalcType 5. The field defined in Calc3 is evaluated to identify the default value to pre-select. |
| 11 | Provides the user with a dropdown list of values. The list is populated using a dictionary object obtained from:  Calc1 – name of the class  Calc2 – name of the function that returns a dictionary.  Calc3 – Calc10 – input values to the function defined in Calc2. These input values can be captured from the user input or default values can be specified. A default value is anything that is enclosed in double quotes e.g., “1”, “Hello World” etc. To read the user input a variable name must be used. Then this variable name must be plugged into one of the SDVariable1 – SDVariable10 columns of a docket code. |
| 12 | Provides the user with a dropdown list of values used in a pre-defined enumeration.  Calc1 – name of the enumeration e.g., ShowCaseClassLib.eCaseBalanceLedgerTypes |
| 13 | NOT USED |
| 14 | Returns the value of a ShowCase setting.  Cacl1 – name of the showcase setting |
| 15 | Decimal |

## SD Functions

Smart docket functions are used to express actions that can be executed dynamically. These actions could be simple function calls, popping up user interfaces or executing workflows. No matter what the action is, it must be defined in the lookup table SD Functions. Table below looks at the different elements needed to define a smart docket function.

|  |  |
| --- | --- |
| **Name** | **Description** |
| Description | The name and/or brief description of the SD Function. |
| SD Class Map | The name and path to a class |
| SD Class Function | Name of the function to execute |
| SD Variable 1 – SD Variable 10 | Input parameters to the function. There are 3 different kinds of parameters that can be supplied.  ***CCase*** – This is a hard coded parameter which instructs ShowCase to pass the case object as the input parameter  Static parameter – A default or static value to be supplied as a parameter value. This parameter is entered as string value inside double quotes e.g., “11”, “Hello World”.  Variable parameter – The value of the parameter is unknown at design time, but can be captured from the user when they are ready to execute the smart docket. This parameter is entered as a string that represents a variable name, e.g., *barNumber.* Then this variable name must be entered in one of the SDVariable1 – SDVariable10 fields of the Docket Code itself. |

Table below shows how to define an SD function to execute a workflow.

|  |  |
| --- | --- |
| **Name** | **Description** |
| Description | The name and/or brief description of the SD Function. |
| SD Class Map | ***CGlobal.ProcessWorkflows*** |
| SD Class Function | ***ExecuteWorkflowByName*** |
| SD Variable 1 | ***CCase*** |
| SD Variable 2 | Category name of the workflow e.g., ***“SmartDocketing.Criminal”***. This is a static parameter hence it must be inside double quotes. |
| SD Variable 3 | Name of the workflow e.g., ***“ContinuanceWorkflow”***. This is a static parameter hence it must be inside double quotes. |
| SD Variable 4 – SD Variable 10 | Any input parameters to the workflow |

## Putting it all together

Smart docketing is a versatile process that allows the definition and automation of new processes without the need for versioning the product or expensive customization requests. So how does it all look after the entire setup is done? Take a look at the Figure-17 below.

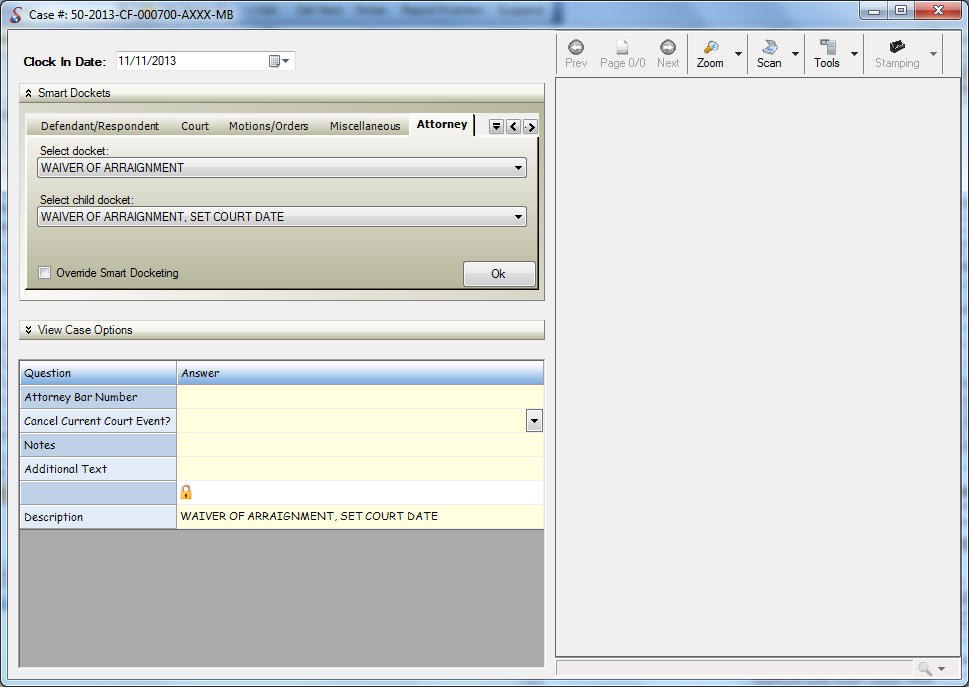


Figure-17: Smart Docketing form

Smart Docketing is a document driven process. To get to this form, Open a case, navigate to the dockets tab and click on the “Smart Docketing” button. The idea is to allow the system to automatically run required processes with minimal data input from the user after scanning a document. A document is identified by a docket Code and the functions attached to the docket code drive the document driven process.

As seen in the figure above, all smart dockets are organized into Categories called Docket Categories. These are user defined and can be assigned to a docket code. Every category is represented in the smart docketing form as a tab. The docket code descriptions appear inside the dropdown list provided under each tab.

A smart docket can also have a parent-child relationship with other smart dockets. The figure above shows such scenario. When a smart docket is selected in the first dropdown, if it has any child smart dockets, they will appear in the second dropdown list. This hierarchy is established by simply setting the ParentDocketCode property of every child docket code.

Once the smart docket is selected, any docket fields configured are displayed in the grid below. These serve as questions to the user to provide answers for. The values entered in this grid are mapped to the input values to the functions and finally by the click of the Ok button, any configured smart docket functions are executed. The checkbox “Override Smart Docketing” can be used to disable the execution of any configured smart docket functions.

# Data Exchange Services

The DES sub-system was created to provide a standard and uniform framework to create data exchange interfaces within the ShowCase system. A data exchange is defined as sharing of information between 2 or more systems. Typically such exchanges are implemented using web services and the systems involved needed to create and maintain service contracts to these web services. When exchanges and the need to communicate with different systems grow, it gets very hard to maintain and track the information flow between systems. It is also a complex process to maintain the service contracts between all the systems.

An exchange in DES is also accessed as a service call. But it is implemented using the REST (Representational State Transfer) protocol, which means there is no need for external systems to create a service contract to access the exchange. REST is the way World Wide Web works. It dictates that every entity can be treated as a resource and 4 different actions can be performed on a specific resource, namely, GET, POST, PUT and DELETE. These actions are called http verbs. The GET verb is used to only get information about the resource and it should not modify the resource. POST is used to create a resource and PUT can be used for both creating and updating the resource. DELETE is used to delete the resource. This is an idealistic REST philosophy and is known as Pure REST. DES does not follow Pure REST, but follows what’s called Practical REST. With Practical REST, like a SOAP web service, the http verbs are used to represent actions, not necessarily a resource. The verb GET can be used to get information about a resource and also to perform certain actions such as deletes and updates. The verb POST can be used to create, update and delete information. The framework that is used to implement REST (WCF REST for example) does not restrict the developer to use Pure REST or Practical REST. Both of these are just design ideologies of how restful services can be implemented.

An exchange in DES is implemented using a workflow and is accessed via a simple URL. For example, an exchange to get citation information provided a citation number can be accessed via a URI pattern such as: ***/GetCitationInformation?CitationNumber={citnum}***. Multiple such exchanges are defined using unique URI patterns and workflows in the DES system.

Take a look at the table below to understand how a DES exchange is configured:

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Description** | **Example** |
| Name | A friendly name for the data exchange. 100 character limit. | Get Citation Information |
| Description | A brief description of the purpose of the data exchange. 100 character limit. | Gets the Citation Information for Processing Citation Payment |
| UriTemplate | This defines how the url should look like for accessing this data exchange. All the input parameters that need to be in the url are also defined here. 1000 character limit. | /GetCitationInformation? CitationNumber={CitationNumber} |
| IsRealTime | Identifies if the data exchange needs to be executed in a real time or a non-real time fashion. | True |
| ProcessWorkflow | The workflow that implements the data exchange | GetCitationInformationWorkflow |
| Priority | Priority of this data exchange compared to others. | 3 |
| Worker Address | Non-real time exchanges can be executed by several worker services called EWS. Specify the worker service that executes this type of exchange requests. |  |

## ExchangeRequest and ExchangeResponse

A DES request is made by using the web service **Exchange Point Service** (EPS). This service is considered the entry point into the DES system. When a request for an exchange is received by EPS, it internally constructs an instance of the class ExchangeRequest. This object contains all the information about the exchange and the input values received. Next the workflow configured for the exchange is executed by passing in the ExchangeRequest object. Therefore a DES workflow must accept an input argument of this type. Currently the name of this argument must be “ExchangeRequest”.

When the workflow finishes executing, EPS expects a return value to be included as a response to the request received. This return value must be an xml document. Therefore a DES workflow must also contain an output argument of type *System.Xml.XmlDocument*. Currently the name of this output argument must be “Result”. The restriction on the name of the input and output argument may be relaxed in later versions of DES.

## GetShellCase

In the world of ShowCase programming the Case object is the most important component. Almost all of the business logic in ShowCase is intended for a Case. The Case object is the root of a number of ShowCase entities and each of those entities cannot be instantiated without the root. The Case object also needs a properly initialized Global Class object to allow access to all the lookup tables. There is a significant amount of code required to properly initialize a root Case object. For ShowCase workflows, the client application constructs the root Case object for use which is convenient. But DES workflows do not have such a convenience. A DES workflow is intended to work with an ExchangeRequest object primarily. So when a DES workflow needs to work with the Case object, the method GetShellCase needs to be used to obtain an empty case object that has all required references to its child classes and the global class. To get a handle to a well-defined CCase object:

1. Authenticate against the ShowCase system using the Authentication Service.
2. Call the ShowCaseDAL.CCaseHelper.GetShellCase method.

## DES Request Authentication

DES exchanges can be implemented to require authentication. This is especially needed when the exchange uses ShowCase classes to read or update information. In this scenario, the caller of the exchange should be authenticated prior to the DES request being made. Let us consider few scenarios where this authentication is automatically done for the caller.

**Calling DES Real-Time Exchange from inside a ShowCase Workflow**

DES exchanges can be executed from within a ShowCase workflow. To do this the following functions can be used on ***ShowCaseUtil.HttpHelper*** class:

|  |
| --- |
| public static string GetXML(string url) |
| public static string GetXML(string url, bool passClaimCookie) |
| public static string PostXML(string url, string data) |
| public static string PostXML(string url, string data, bool passClaimCookie) |

The GetXML overloaded functions can be used to execute a DES request that relies on the http verb GET. These are called “GET requests”. A GET request can be executed by simply passing in the URL. Optionally a Boolean parameter can be specified whether or to embed the authentication cookie in the request. Since a user is already authenticated in ShowCase, this authentication cookie can simply be forwarded to DES by using the 2nd overload of GETXML and PostXML functions. Remember that these functions are simply forwarding the user’s authentication cookie to the DES. These methods do not perform any authentication prior to the issuing the request.

**Calling DES Real-Time Exchange from other areas**

Another way to execute a DES request is to use the methods on the class ***Exchange.ClientLib.ExchangeData***:

|  |
| --- |
| public static string RequestData(string epsURL) |
| public static string RequestData(string epsURL, string userName, string password, string friendlyDBName) |
| public static string RequestData(string epsURL, bool includeAuthCookie) |
| public static string PostData(string epsURL, XmlDocument data) |
| public static string PostData(string epsURL, XmlDocument data, string userName, string password, string friendlyDBName) |
| public static string PostData(string epsURL, string data) |
| public static string PostData(string epsURL, string data, bool includeAuthCookie) |
| public static string PostData(string epsURL, string data, string userName, string password, string friendlyDBName) |

These methods allow you to use RequestData (Get requests) or PostData (Post requests) to the DES system. Each of these methods have an overload that takes in the Boolean paramaeter “includeAuthCookie” that can be used to simply forward the user’s authentication cookie to DES. This is similar to the GetXML and PostXML methods defined above. Additionally, the RequestData and PostData methods also have overloads that take in “username”, “password” and “friendlyDBName” parameters. These overloads will perform the authentication and pass the resulting user’s auth cookie to the DES.

To learn more about authenticating against ShowCase, refer to the white paper on ShowCase’s authentication sub-system.

**Calling DES Non-Realtime Exchanges**

The same classes and functions defined above are used to invoke non-real time exchanges.

Non-realtime exchanges in DES are executed by the worker service called EWS (Exchange Worker Service). This service is pre-configured with a ShowCase user account which is going to be used for execution when needed. Currently there is no way to pass the credentials of the caller to the EWS service.

## DES Agent Workflows

ShowCase Agent is a specialized process that is used to execute most of the nightly jobs such as TCATS, D6, CCIS etc. This is built as a windows service that is capable of executing several jobs simultaneously. The agent service has the ability to:

1. Create and configure jobs.
2. Schedule jobs at certain intervals or at specific times during any day.
3. Send notifications (emails) upon successful execution or failures of jobs.

Each job in the Agent process is defined as a workflow and is implemented using WF 3.0. It is possible to re-define an Agent job as a DES exchange. DES can be used to create and configure jobs or exchanges as we discussed above. But they are always invoked by a caller and hence are not readily available for scheduling. To allow a DES exchange to be scheduled, we can use a SQL Server Agent to define a scheduled job. The job itself will simply invoke a non-real time exchange defined in DES using a power shell script as shown in the Figure 18 below. The exchange will then need to take care of the implementation and also the notifications when the exchange has finished execution.

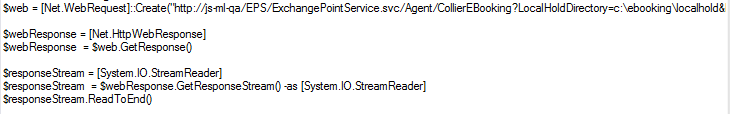


Figure 18 Powershell script to execute a DES exchange

To simplify the process of creating a new DES agent workflow, a custom activity called **AgentActivity** is created. This is essentially a sequence activity with certain pre-defined parameters. This activity is defined in the **Exchange.WorkflowLib.Desgin** assembly. Make sure to add the activity from this assembly to the toolbox to create your own DES Agent Exchange. Take a look at the Figure 19 below that shows the CCIS Export agent job defined as a DES exchange.

The agent activity is a sequence activity with certain pre-defined parameters. See the “Properties” pane to the right. An agent activity can be used to set the authentication information as seen in the **Authentication** section of properties window. This is used to allow the exchange to automatically authenticate against ShowCase before executing any activities inside the workflow. The **Notifier** section of the properties window can be used to configure the email addresses which should be notified on the execution status of the workflow.

The **Arguments** to a DES Agent workflow are similar to a DES workflow. An input parameter named “ExchangeRequest” of type ExchangeRequest is required. Also an output parameter named “Result” of type XmlDocument is required. Any additional parameters that the job requires can be defined as arguments to the workflow. The definition of the job itself can be expressed inside the AgentActivity. For example, as seen in the Figure 19, the Flowchart activity will hold all the logic for the CCIS Export agent activity.

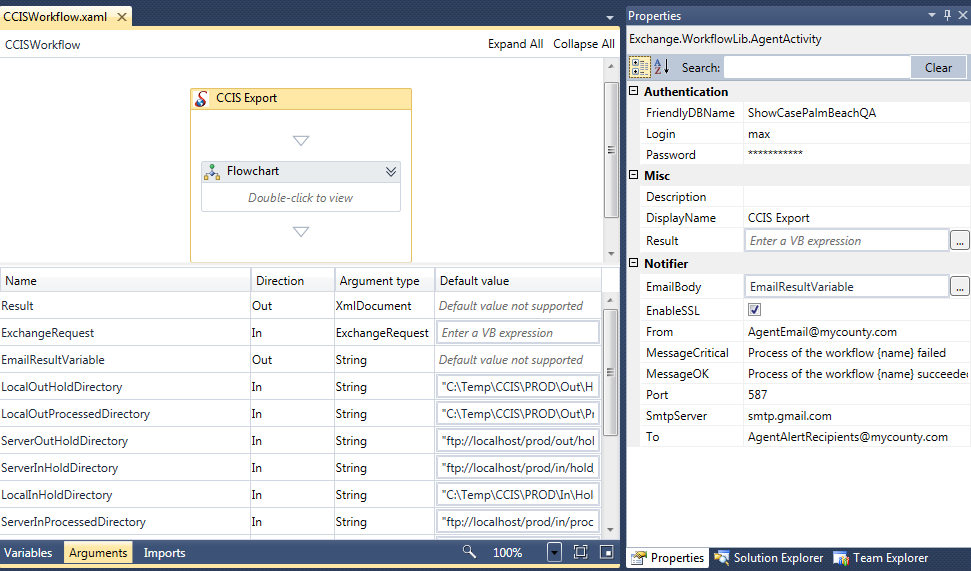


Figure 19 CCIS Agent Workflow

# Summary

We have reviewed the multi-faceted role of workflows in the ShowCase ecosystem. They are used in the implementation of several features of the product. We discussed how to create and manage workflows in different versions of .NET framework. We also learned a good deal about the differences between ShowCase workflows and DES workflows. We discussed how to create DES Agent workflows using the AgentActivity.

The power of workflow orientation is more visible in Smart Docketing. It is a flagship feature of ShowCase and is a power tool to create custom processes and logic beyond what is originally provided in the product. The DES system also benefits from the principles of workflow orientation. New exchanges can be defined and existing exchanges can be modified dynamically due to the no-code approach of these workflows.