Workspace Desktop Edition Developer's Guide

Best Practices for Views

12/14/2025

Best Practices for Views

Purpose: To provide a set of recommendations that are required in order to implement a typical view within Interaction Workspace.

Contents

Keyboard Navigation

TAB Key--Every control in a window has the ability to have focus. Use the TAB key to move from one control to the next, or use SHIFT+TAB to move the previous control. The TAB order is determined by the order in which the controls are defined in the Extensible Application Markup Language (XAML) page. **Access Keys**--A labeled control can obtain focus by pressing the ALT key and then typing the control's associated letter (label). To add this functionality, include an underscore character (_) in the content of a control. See the following sample XAML file:
[XAML]

```
<Label Content="_AcctNumber" />
```

Focus can also be given to a specific GUI control by typing a single character. Use the WPF control AccessText (the counterpart of the TextBlock control) to modify your application for this functionality. For example, you can use the code in the following XAML sample to eliminate having to press the ALT key:
[XAML]

```
<AccessText Text="_AcctNumber" />
```

Shortcut Keys--Trigger a command by typing a key combination on the keyboard. For example, press CTRL+C to copy selected text. **Alarm Notification**--Interaction Workspace can be configured to emit a sound when an unsolicited event occurs.

Branding

To replace trademark logos, icon images and text, modify the `Rebranding.xml` file. The `Rebranding.xml` file is similar to a language dictionary and enables you customize the appearance of your application. For example, you can replace the embedded splashscreen resource, `pack://application:,,,/Genesyslab.Desktop.WPFCommon;component/Images/Splash.png`, with a local image such as `MySplash.png` or `MyImagesFolder/MySplash.png`. The `Rebranding.xml` file is shown in the following example:
[XML]

```
<Dictionary>
    <Value Id="Application.SplashScreen"
Source="pack://application:,,,/Genesyslab.Desktop.WPFCommon;component/Images/Splash.png"/>
    <Value Id="Window.AboutWindow.Logo" Source="MonLogo.png"/>
    <Value Id="Window.LoginWindow.Logo" Source="Branding/Logo2.png"/>
    <Value Id="Windows.AboutWindow.TextBlockInteractionWorkspace" Text="Custom Interaction
Workspace"/>
</Dictionary>
```

Note: You must include a path that is relative to the application when you replace the splashscreen.

Localization

To dynamically change the language in your view, modify the XAML by using the following sample:
[XAML]

```
<UserControl xmlns:loc="http://schemas.tomer.com/winfx/2006/xaml/presentation">
  <Expander>
    <Expander.Header>
      <TextBlock loc:Translate.Uid="DispositionCodeView.TextBlockDisposition"
          Text="{loc:Translate Default=The Disposition}" />
    </Expander.Header>
    <Button/>
  </Expander>
</UserControl>
```

Refer to `DispositionCodeView.TextBlockDisposition` in the language XML file. For English, modify the `Genesyslab.Desktop.Modules.Windows.en-US.xml` file as shown in the following example:
[XML]

```
<Dictionary EnglishName="English" CultureName="English" Culture="en-US">
  <Value Id="DispositionCodeView.TextBlockDisposition" Text="The Disposition"/>
</Dictionary>
```

For French, modify the `Genesyslab.Desktop.Modules.Windows.fr-FR.xml` file as shown in the following example:
[XML]

```
<Dictionary EnglishName="French" CultureName="France" Culture="fr-FR">
  <Value Id="DispositionCodeView.TextBlockDisposition" Text="La Disposition"/>
</Dictionary>
```

The language can also be changed within the code itself, as shown in the following example:
[C#]

```
string text =
LanguageDictionary.Current.Translate("DispositionCodeView.TextBlockDisposition", "Text");
```

Parameterization

Interaction Workspace is configured as a role-based application. For example, if an agent is assigned the task of `TeamCommunicator`, the Click-Once group file that is related to this task is downloaded when the application starts up and the associated module is loaded in RAM. The GUI that is specific to this task is then displayed only to the agents that are assigned the `TeamCommunicator` task. The task section in the following example enables you to download and execute a custom module extension. If the task name (`InteractionWorkspace.TeamCommunicator.canUse`) is configured in Configuration Manager, the required group of files (`TeamCommunicator`) is downloaded, and the module (`TeamCommunicatorModule`) are executed. This parameterization functionality is configured in the `InteractionWorkspace.exe.config` file, as shown in the following example:
[XML]

```
<configuration>
  ...
  <tasks>
    ...
```

```
    <task name="InteractionWorkspace.Features.TeamCommunicator"
      clickOnceGroupsToDownload="TeamCommunicator"
      modulesToLoad="TeamCommunicatorModule" />
    ...
  </tasks>

  <modules>
    ...
    <module assemblyFile="Genesyslab.Desktop.Modules.TeamCommunicator.dll"
      moduleType="Genesyslab.Desktop.Modules.TeamCommunicator.TeamCommunicatorModule"
      moduleName="TeamCommunicatorModule"
      startupLoaded="false"/>
    ...
  </modules>
  ...
</configuration>
```

Parameterization functionality can also be accomplished by loading a custom module conditioned with a task. In the Configuration Manager, a role must be configured with the name of the task. In this example, the task is named `InteractionWorkspace.ExtensionSample.canUse` and assigned to the agent. This custom parameterization functionality is configured in the `ExtensionSample.module-config` file, as shown in the following example:
[XML]

```
<configuration>
  <configSections>
    <section name="tasks"
      type="Genesyslab.Desktop.Infrastructure.Config.TasksSection,
Genesyslab.Desktop.Infrastructure" />
    <section name="modules"
      type="Microsoft.Practices.Composite.Modularity.ModulesConfigurationSection,
Microsoft.Practices.Composite" />
  </configSections>
  <tasks>
    <task name="InteractionWorkspace.ExtensionSample.canUse"
      clickOnceGroupsToDownload="ExtensionSample"
      modulesToLoad="ExtensionSampleModule" />
  </tasks>
  <modules>
    <module assemblyFile="Genesyslab.Desktop.Modules.ExtensionSample.dll"
      moduleType="Genesyslab.Desktop.Modules.ExtensionSample.ExtensionSampleModule"
      moduleName="ExtensionSampleModule"
      startupLoaded="false"/>
  </modules>
</configuration>
```

Internationalization

WPF and .NET work with Unicode strings, so internationalization does not normally require extra coding. However, there are some potential issues to consider when creating your custom code, such as:

- Strings coming from the server might not be in true Unicode.
- The language might not be read/written from left to right (for example, Arabic languages).
- The correct font must be installed on the agents system.

Screen Reader Compatibility

The Microsoft UI Automation API is used for WPF applications that require accessibility functionality. The following two tools are available to assist you in developing applications that are compliant with accessibility software, such as Job Access With Speech (JAWS):

- `UISpy.exe` (Microsoft Windows SDK)--Displays the GUI controls tree along with the UIAutomation properties of the controls (such as AccessKey, Name, and others)
- Narrator (Microsoft Windows)--Reads the content of a window

Use the following code sample to add a name to a GUI control in the XAML file:
[XAML]

```
<TextBox Name="textBoxUserName" AutomationProperties.Name="UserName" />
```

The `AutomationProperties.Name` of the TextBox control is automatically set with the content value of a Label control. If a GUI control already has a Label control the XAML file looks similar to the following example:
[XAML]

```
<Label Target="{Binding ElementName=textBoxUserName}" Content="_UserName" />
<TextBox Name="textBoxUserName" />
```

Note: The `AutomationProperties.Name` must be localized.

Themes

Genesys recommends that you place the control styles and color resources that are used in the application into an XAML file containing a WPF ResourceDictionary. This enables you to modify and extend an existing theme. To make the themes extensible, use ThemeManager to register all the available themes in the application. When a theme is changed, ThemeManager copies this ResourceDictionary to the global application ResourceDictionary. All previously copied styles and brushes are overwritten with the new ones. **Note:** The XAML file that you create to contain the control styles and color resources is not a Microsoft Composite Application Library (CAL) module.

Loosely-coupled Application Library and Standard Controls

Interaction Workspace is a modular Windows Presentation Foundation (WPF) client application and uses the standard WPF controls. This section provides information about these controls. The Loosely-coupled Application Library is part of the Composite Application Guidance which aims to produce a flexible WPF client application that is loosely coupled. The following graphical tree shows a typical composite application built with loosely-coupled applications:

```
Shell
 Region1
  View11
  View12
 Region2
```

```
  View21
   Region21
     View211
     View212
Shell
```

The typical GUI is composed of a shell, region(s), and view(s). The shell is the main window of the application where the primary user interface (UI) content is contained. The shell is usually a single main window that contains multiple views. The shell can contain named regions where modules can add views. A region is a rectangular graphical area that is embedded in a shell or a view and can contain one or more views. Views are the composite portions of the user interface that are contained in the window(s) of the shell. Views are the elementary pieces of UI, such as a user control that defines a rectangular portion of the client area in the main window.

Views

A view contains controls that display data. The logic that is used to retrieve the data, handle user events, and submit the changes to the data is often included in the view. When this functionality is included in the View, the class becomes complex, and is difficult to maintain and test. You can resolve these issues by using Presentation Patterns and Data Binding.

Presentation Patterns

Use patterns to separate the responsibilities of the display and the behavior of the application into different classes, named the `View` and the `View Model`. Genesys suggests the following presentation patterns:

- Model-View-ViewModel (MVVM)
- Model-View-PresentationModel (Presentation Model)

The MVVM pattern is used in Genesys samples.

- The Model is similar to having several data sources (InteractionService from Enterprise SDK, Statistics from the Platform SDK, or any other data).
- The View is a stateless UserControl; a graphical interface with no behavior.
- The ViewModel is an adaptation layer between the Model and the View. It offers the Model data to the View. The behavior of the View is defined in this layer. For instance, the View launches the commands, but the commands are implemented in the ViewModel.

Each view consists of several classes. The VoiceView is described in the following table:

Roles	Classes/Interfaces	Files	Description
View	IVoiceView	IVoiceView.cs	The interface
View	VoiceView	VoiceView.xaml VoiceView.xaml.cs	The implementation of the IVoiceView. VoiceView.xaml is the XAML file that describes the view and

Roles	Classes/Interfaces	Files	Description
			VoiceView.xaml.cs contains the code behind.
ViewModel	IVoiceViewModel	IVoiceViewModel.cs	The interface
ViewModel	VoiceViewModel	VoiceViewModel.cs	The implementation of the IVoiceViewModel.

Data Binding

When you use presentation patterns in application development you have the option of using the data-binding capabilities that are provided by the WPF. Data-binding is used to bind elements to application data. The bound elements automatically reflect changes when the data changes its value. For example, if the DataContext property of the VoiceView class is set to an instance of the VoiceViewModel class, then the Text property of a TextBlock control can have a DataBinding toward the PhoneNumber property of the VoiceViewModel class. By default it is a two-way binding. If the value of either the VoiceViewModel.PhoneNumber or the TextBlock display changes then the other changes as well. The following example also shows how the command VoiceViewModel.AnswerCallCommand can be initiated from the VoiceView:

```
<TextBlock Text="{Binding PhoneNumber}"/>
<Button Command="{Binding AnswerCallCommand}">Answer Call</Button>
```

Note: Modularity requires that each interface is registered in the module initialization. See Customize Views and Regions for details on how to register an interface.

Tips and Tricks

When you need to control several Views, you can use a Controller class to coordinate the activities of multiple Views (and others controllers). The ViewModel is created by the View, and the Views are created and managed by the Controllers. The following logical tree is a depiction of the relationship between the instantiated classes:

```
Controller1
 Controller11
  View111
   ViewModel111
  View112
   ViewModel112
 View12
  ViewModel12
Controller2
 View21
  ViewModel21
 View22
  ViewModel22
```

Use the information provided in this section along with the information in the Customizing Interaction Workspace topic to create your own view.