# UI Examples

- A simple configurable plugin
- A plugins that shows a custom dialog
- A plugins that defines a custom menu
- A simple plugin that consumes events

# A simple configurable plugin

Creating a plugin that registers a configuration and uses the default configuration editor.

The Plugin class.

The first step is to create a class that implements the IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
public class SimpleConfigurablePlugin : IPluginService
{
}
```

## Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Name
{
  get { return return this.GetType().Name; }
}

public string Description
{
   get { return "A plugin that has configuration"; }
}

public string Version
{
  get { return "1.1.1"; }
}

public IEnumerable<int> FeatureIds
{
  get { return null; }
}
```

## The Initialize method

First we need to register a custom cinfiguration interface and register to recieve an event when the configuration is loaded.

```
  ...
  var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterface<I
  loader.ConfigLoaded += loader_ConfigLoaded;
  ...
```

Then we need to registter a defualt configuration definition of our custom type with the UI definition manager

```
unityContainer.Resolve<UIDefinitionManager>().RegisterUIElement
(
  new DefaultConfigDefinition<ISimpleConfiguration>(category: OptionCategories.Plugins, name: "
);
```

The full code for the example with extra info:

## SimpleConfigurablePlugin.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.UIComposition.UIConfiguration;
using FTC.SDKInterface.Plugins;
using FTC.SDKInterface.Services;
using Feat.UIComposition.Config;
using System.ComponentModel;
using Feat.UIComposition;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Menus;
using FTC.SDKInterface.Constants;
using Feat.Metadata.PropertyGrid;

namespace UIPlugins
{

   /// <summary>
   /// This examples demos a simple plugin that registers a configuration with FTC
   /// and uses the default configuration editor in scene composer.
   /// </summary>
   public class SimpleConfigurablePlugin : IPluginService
   {
      public string Description
      {
         get { return "A plugin that has configuration"; }
      }
      public string Name
      {
         get { return this.GetType().Name; }
      }

      public string Version
      {
         get { return "1.1.1"; }
      }

      public IEnumerable<int> FeatureIds
      {
```

```
      get { return null; }
   }

   public void Initialize(IUnityContainer unityContainer)
   {
      // We register the configuration interface with the configuration manager.
      // The object that is returned allow access to the config instance, allows the saving
      // and allows the registration of events pertinent to the configuration, such as the
      // Explicit loading or saving of the configuration is not necessary, as FTC will load
      // it is accessed, and will save it when the application closes.
      // The plugin developer can however explicitly save the configuration or reload it fr
      var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInt

      //We register to receive events when the configuration is loaded.
      loader.ConfigLoaded += loader_ConfigLoaded;


      // We register with the ui definition manager a default configuration definition with
      // We also specify the category in which to place the configuration, and the name to
      unityContainer.Resolve<UIDefinitionManager>().RegisterUIElement
      (
         new DefaultConfigDefinition<ISimpleConfiguration>(category: OptionCategories.Plugi
      );
   }

   /// <summary>
   /// Event handler for when the configuarion is loaded.
   /// </summary>
   /// <param name="sender"></param>
   /// <param name="e"></param>
   void loader_ConfigLoaded(object sender, ConfigLoadedEventArgs<ISimpleConfiguration> e)
   {
      // If the configuration was not loaded successfully, either this is the first time it
      // and there was no configuration to load, or the configuration file was corrupted.
      // Either way we initialize complex configuration properties, that could not be initi
      if (!e.ConfigLoadedSuccessfully)
      {
         e.Instance.OtherParameters = new List<string>
         {
            "Param1 "
         };
      }
   }


}

/// <summary>
/// The configuration interface. This interface will be implemented by Scene Composer and c
/// when it is registered with the Configuration manager service.
/// </summary>
/// All properties declared  in the configuration interface must conform to standard .NET X
///
```

```
    /// If we use the default editor for configurations then we can control how the property gr
    ///
    /// The CategoryDisplayInfo attribute specifies that the default category should have no he
    ///
    /// The DefaultValue attribute specifies the default value to be used for the property, eit
    /// when the configuration is first created. \n
    ///
    /// The Category attribute dictates the category in which the property grid will display th
    ///
    /// The DisplayName attribute specifies what name the property grid should use for this pro
    /// Scene composer will use the actual property name.\n
    ///
    /// The Browsable attribute allows the hiding of properties in the property grid.
    [CategoryDisplayInfo("", StyleKey = PropertyGridConstants.NoHeaderCategory)]
    public interface ISimpleConfiguration : IConfiguration
    {

       [DefaultValue("format C:")]
       string CommandToExecute { get; set; }

       [DefaultValue(true)]
       [DisplayName("Show confirmations")]
       [Category("Options")]
       bool Confirm { get; set; }

       [DefaultValue(false)]
       [Category("Options")]
       bool Silent { get; set; }

       [Browsable(false)]
       List<string> OtherParameters { get; set; }
    }
}
```

# A plugins that shows a custom dialog

## Creating a plugin that displays a custom dialog

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll, Feat.UIComposition.dll and Feat.Core.dll. The resulting dll from your project (along with any non .NET dependecies) must be placed in the Plugins directory in the FTC install location.

### The Plugin class.

The first step is to create a class that implements the IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
public class PluginWithMenuAndDialogPlugin : IPluginService
{

}
```

### Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Description
{
  get { return "A plugin with a menu item and dialog"; }
}

public string Name
{
  get { return this.GetType().Name; }
}

public string Version
{
  get { return "1.1.1"; }
}

public IEnumerable<int> FeatureIds
{
   get { return null; }
}
```

The initliaze method will register a menu command and a custom dialog Registering Commands: To register a command first an interface that derives from ICommandContainer must be created.

The interface will contain properties for each of the commands that will be registered. The interface will be implemented internally by Scene Composer. The plugin can then register the commands using the UICommandManager service. This can be done either directly or using the helper class CommandDefinitionCollection that simplifies the registering of multiple commands.

The command can be invoked from code, (see other more complex examples for a demonstration), or can be displayed in the menu or used as a shortcut. This is done by attaching metadata to the command, as displayed in the example.

```
public void Initialize(IUnityContainer unityContainer)
{
  var uiDefManager = unityContainer.Resolve<UIDefinitionManager>();

  //Creates a collection of ui element definitions.
  var uiElementCollection = new UIDefinitionCollection()
  {
      new DialogDefinition<ICustomDialogViewModel, CustomDialogViewModel, ICustomDialogView, Cu
  };
  // We register the elements of the collection with the UIDefinitionManager
  uiElementCollection.RegisterAll(uiDefManager);

  // We create the command collection and add the command definitions.
  var cmdCollection = new CommandDefinitionCollection<IPluginCommands>()
  {
     {
        c => c.ContextCommand,
        new ManagedCommand<IBitmap>()
        {
           ExecuteHandler = o=>
           {
           },
           Metadata = new MetadataCollection()
           {
              new ContextMenuOption()
              {
                  MenuPath = new MenuPath() { "Here" },
                  PanelViewModel = typeof(ITreeViewModel<>)
              },
           new AutoInvalidateOnSelectionInterface<ISelectedSolution>()
           }
        }
     },
     {
        c=>c.ShowDialogCommand,
        new ManagedCommand()
        {
          ExecuteHandler = (o)=>
          {
            var dlgSrv = unityContainer.Resolve<IDialogSvc>();
            var dialogVM = unityContainer.Resolve<ICustomDialogViewModel>();
            dialogVM.Message = "Custom Dialog";
            if (dlgSrv.ShowDialog(dialogVM) == true)
```

```
            {
              dlgSrv.ShowMessageBox(dialogVM.Message);
            }
          },
          Metadata = new MetadataCollection()
          {
            new MainMenuOption()
            {
              MenuPath = new MenuPath() { MenuConstants.File_2, "My Menu Option", 350 },
            },
            new ToolTip("Invokes a custom dialog from a plugin"),
            new KeyShortcut() { Key = Key.B, Modifiers= ModifierKeys.Control | ModifierKeys.Shi
          }
        }
    }
  };
  cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
 }
```

## The dialog window

This custom dialog will contain a button that will exacute a command to display a messagebox.

```
this.ShowMessageBox = new ManagedCommand()
{
  ExecuteHandler = o =>
  container.Resolve<IDialogSvc>().ShowMessageBox
  (
  "Message box from a plugin",
  "Message box title",
  MessageBoxButton.OK,
  MessageBoxImage.Exclamation
  )
 };
```

The full code for the example:

## CustomUIConfigPlugin.cs

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
 using FTC.SDKInterface.Plugins;
 using FTC.SDKInterface.Selections;
 using FTC.SDKInterface.Services;
 using Feat.UIComposition;
 using Feat.UIComposition.Common;
 using Feat.UIComposition.Config;
 using Feat.UIComposition.Paneling;
 using FTC.SDKInterface.Constants;
 using UIPlugins.PluginWithMenuAndDialog.CustomDialog;
```

```
using Feat.UIComposition.Dialogs;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Menus;
using Feat.UIComposition.Shortcuts;
using System.Windows.Input;
using Feat.CAL.Services;
using Feat.UIComposition.ContextMenus;
using Feat.Core;
using FTC.SDKInterface.Solution;

namespace UIPlugins.PluginWithMenuAndDialog
{
   /// <summary>
   /// This is an example of a plugin that registers a command and displays a custom dialog ar
   /// </summary>
   public class PluginWithMenuAndDialogPlugin : IPluginService
   {
      public string Description
      {
         get { return "A plugin with a menu item and dialog"; }
      }
      public string Name
      {
         get { return this.GetType().Name; }
      }

      public string Version
      {
         get { return "1.1.1"; }
      }

      public IEnumerable<int> FeatureIds
      {
         get { return null; }
      }

      /// <summary>
      /// The initialization method has the role of registering with Scene composer different
      /// </summary>
      /// In this example the initliaze method will register a menu command and a custom dialo
      ///
      /// Registering Commands:
      /// To register a command first an interface that derives from ICommandContainer must be
      /// The interface will contain properties for each of the commands that will be register
      /// The interface will be implemented internally by Scene Composer.
      /// The plugin can then register the commands the UICommandManager service. This can be
      /// directly or using the helper class CommandDefinitionCollection that simplifies the r
      /// commands. \n
      ///
      /// The command can be invoked from code, (see other more complex examples for a demonst
      /// or can be displayed in the menu or used as a shortcut. This is done by attaching met
      /// as displayed in the example.
      /// <param name="unityContainer"></param>
```

```
public void Initialize(IUnityContainer unityContainer)
{
   var uiDefManager = unityContainer.Resolve<UIDefinitionManager>();

   //Creates a collection of ui element definitions.
   // In this case we just register a dialog.
   // A ui element has 4 elements :
   // - a view model interface
   // - a view model class implementation
   // - a view interface
   // - a view class implementation
   var uiElementCollection = new UIDefinitionCollection()
   {
      new DialogDefinition<ICustomDialogViewModel, CustomDialogViewModel, ICustomDialog\
   };
   // We register the elements of the collection with the UIDefinitionManager
   uiElementCollection.RegisterAll(uiDefManager);

   // We create the command collection and add the command definitions.
   var cmdCollection = new CommandDefinitionCollection<IPluginCommands>()
   {
      {
         c => c.ContextCommand,
         new ManagedCommand<IBitmap>()
         {
            ExecuteHandler = o=>
            {
               //unityContainer.Resolve<FTC.SDKInterface.Selections.ISelectedScene>().Va
            },
            Metadata = new MetadataCollection()
            {
               new ContextMenuOption()
               {
                  MenuPath = new MenuPath() { "Here" },
                  PanelViewModel = typeof(ITreeViewModel<>)
               },
               new AutoInvalidateOnSelectionInterface<ISelectedSolution>()
            }

         }
      },
      {
         // We specify the command for which we are registering an implementation,
         // The type of c will be inferred to be IPluginCommands, so we don't have to sp
         c=>c.ShowDialogCommand,
         //
         new ManagedCommand()
         {
            ExecuteHandler = (o)=>
            {
               var dlgSrv = unityContainer.Resolve<IDialogSvc>();
               // Invokes a dialog that was previously registered with the UIDefinitionM
               // We first create the viewmodel for the dialog
```

```
                    var dialogVM = unityContainer.Resolve<ICustomDialogViewModel>();
                    // We configure the view model properties
                    dialogVM.Message = "Custom Dialog";

                    // We show the dialog. The dialog window will be constructed based on the
                    // UIDefinitionManager service.
                    if (dlgSrv.ShowDialog(dialogVM) == true)
                    {
                       dlgSrv.ShowMessageBox(dialogVM.Message);
                    }
                 },
                 Metadata = new MetadataCollection()
                 {
                    // The main menu option to be created.
                    // The menu path is specifies the location where the menu should be displ
                    // The menu constant class contains the main menu definitions used by sce
                    // The plugin can an other main menu options too.
                    // The tokens of the path can be specified either with commas, or using m
                    // The each token of a path can be followed by an integer that represents
                    // in the menu. The option present in the scene composer menu, have order
                    // so plugins can add items in between the existing items
                    // A menu can be devided in several groups, in the example below, the ite
                    new MainMenuOption()
                    {
                       MenuPath = new MenuPath() { MenuConstants.File_2, "My Menu Option", 35
                    },
                    // Specifies a tooltip for the command, used for the menu items and the k
                    new ToolTip("Invokes a custom dialog from a plugin"),
                    //Specifies a shortcut key for the command.
                    new KeyShortcut() { Key = Key.B, Modifiers= ModifierKeys.Control | Modifi
                 }
              }
           }
        };
        // We register the commands.
        cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
     }
  }

  /// <summary>
  /// The interface that contains the command definitions.
  /// </summary>
  /// The main role of this interface is to uniquely identify the command and to provide an e
  /// No implementation of this interface is required, Scene COmposer will generate a concret
  public interface IPluginCommands : ICommandContainer
  {
     ManagedCommand ShowDialogCommand { get; }
     ManagedCommand<IBitmap> ContextCommand { get; }
  }
}
```

## ICustomDialogViewModel.cs

```
using System;
using System.Collections.Generic;
using Feat.Core;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   /// <summary>
   /// The dialog view model interface. This interface can be used to exchange information bet
   /// </summary>
   public interface ICustomDialogViewModel
   {
      string Message { get; set; }

   }
}
```

## ICustomDialogView.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.Core;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   /// <summary>
   /// The interface for the dialog view. It contains no methods as the view does not communic
   /// </summary>
   public interface ICustomDialogView : IDialogView
   {
   }
}
```

## CustomDialogViewModel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows;
using FTC.SDKInterface.Services;
using Feat.Core;
using System.Windows.Input;
using Feat.UIComposition.Commanding;
using Feat.CAL.Services;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   class CustomDialogViewModel : ICustomDialogViewModel
   {
      public CustomDialogViewModel(IUnityContainer container)
```

```
        {
            // We create a simple command to demo using a Scene Composer message box
            // This is the way to show message box so that it will conform to the Scene Composer
            this.ShowMessageBox = new ManagedCommand()
            {
                ExecuteHandler = o =>
                    container.Resolve<IDialogSvc>().ShowMessageBox
                    (
                        "Message box from a plugin",
                        "Message box title",
                        MessageBoxButton.OK,
                        MessageBoxImage.Exclamation
                    )
            };
        }

        /// <summary>
        /// The command for showing a dialog box, that will be bound to the view.
        /// </summary>
        public ICommand ShowMessageBox { get; set; }

        /// <summary>
        /// The message that will be retuned by the viewmodel.
        /// </summary>
        public string Message { get; set; }
    }
}
```

## CustomDialogView.xaml

```
<!--
    The actual UI for the view. Most of the look and feel for the application is inherited by t
    The dialog window must however specify the Style property to ensure consistent look and fee
    The DialogWindow style resource is defined within Scene Composer
-->

<ui:DialogWindow
    x:Class="UIPlugins.PluginWithMenuAndDialog.CustomDialog.CustomDialogView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:ui="clr-namespace:Feat.Core;assembly=Feat.Core"
    Title="Plugin dialog"
    Style="{DynamicResource DialogWindow}" >
   <StackPanel Margin="15">
      <Label Content="This is a custom dialog defined in a plugin"></Label>
      <StackPanel>
         <Label Content="Message:"></Label>
         <!--We use databiding to edit the MessageProperty-->
         <TextBox Text="{Binding Message}"></TextBox>
      </StackPanel>
      <!--We  use databinding to assign the ShowMessageBox command to the button-->
      <Button Command="{Binding ShowMessageBox}" Content="Show message Box" Margin="0,30,0,10"
      <StackPanel Orientation="Horizontal" Margin="10">
```

```
            <Button Click="OnOk" Margin="10" Content="OK"></Button>
            <Button Click="OnCancel" Margin="10" Content="Cancel"></Button>
        </StackPanel>
    </StackPanel>
</ui:DialogWindow>
```

# A plugins that defines a custom menu

## Creating a plugin that displays a custom menu

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll, Feat.UIComposition.dll and Feat.Core.dll. The resulting dll from your project (along with any non .NET dependecies) must be placed in the Plugins directory in the FTC install location.

### The Plugin class.

The first step is to create a class that implements the IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
 public class CustomMenuPlugin : IPluginService
 {

 }
```

### Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Name
{
  get { return "Custom Menu Definition"; }
}

public string Description
{
   get { return "A plugin that defines a custom menu"; }
}

public string Version
{
  get { return "1.0"; }
}
public void Initialize(IUnityContainer unityContainer)
{
    //register the menu config interface
    var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterface
    ...
}
```

### Config and commands interfaces

We will need to provide an interface derived from IConfiguration and an interface derived from ICommandContainer that defines the available commands.

```
public interface IMenuPluginConfig : IConfiguration
{
  ObservableCollection<string> Data { get; set; }
}

public interface ICustomPluginCommands : ICommandContainer
{
  ManagedCommand AddItem { get; }
  ManagedCommand RemoveItem { get; }
}
```

Then in the Initialize method:

Register the menu config interface

```
...
var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterface<IMe
loader.ConfigLoaded += loader_ConfigLoaded;
...
```

Register the menu commands

```
 var cmdCollection = new CommandDefinitionCollection<ICustomPluginCommands>()
 {
  {
     c=>c.AddItem,
     new ManagedCommand()
     {
        ExecuteHandler = o=>
        {
           var data = loader.Instance.Data;
           data.Add("Item " + data.Count);
        },
        Metadata = new MetadataCollection()
        {
           new MainMenuOption()
           {
              MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Add" },
           }
         }
     }
  },
  {
     c=>c.RemoveItem,
     new ManagedCommand()
     {
        CanExecuteHandler = o=> loader.Instance.Data.Any(),
        ExecuteHandler = o=>
        {
```

```
            var data = loader.Instance.Data;
            data.RemoveAt(0);
         },
         Metadata = new MetadataCollection()
         {
            new MainMenuOption()
            {
               MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Remove" },
            },
         }
      }
   }
};
cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
```

And register a custom menu definition to hold the item list

```
  var menuDefMgr = unityContainer.Resolve<MenuDefinitionManager>();
  menuDefMgr.RegisterMenuDefinition(new CustomMenuDefinition()
  {
     MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "RegItems" },
  });

  loader.Instance.Data.CollectionChanged += Data_CollectionChanged;
```

The full code for the example:

## CustomMenuPlugin.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using FTC.SDKInterface.Plugins;
using System.Collections.ObjectModel;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Config;
using Feat.UIComposition;
using Feat.UIComposition.Menus;
using FTC.SDKInterface.Constants;

namespace UIPlugins.CustomMenu
{
  public class CustomMenuPlugin : IPluginService
  {
     public string Name
     {
        get { return this.Description; }
     }

     public string Version
     {
```

```
      get { return "1.0"; }
   }

   public string Description
   {
      get { return "Custom Menu Definition"; }
   }

   public IEnumerable<int> FeatureIds
   {
      get { return null; }
   }

   public FTC.SDKInterface.Services.IUnityContainer UnityContainer { get; set; }

   public void Initialize(FTC.SDKInterface.Services.IUnityContainer unityContainer)
   {
      var loader = unityContainer.Resolve<IConfigurationManager>().RegisterConfigurationInterf
      loader.ConfigLoaded += loader_ConfigLoaded;
      this.UnityContainer = unityContainer;

      var cmdCollection = new CommandDefinitionCollection<ICustomPluginCommands>()
      {
         {
            c=>c.AddItem,
            new ManagedCommand()
            {
               ExecuteHandler = o=>
               {
                  var data = loader.Instance.Data;
                  data.Add("Item " + data.Count);
               },
               Metadata = new MetadataCollection()
               {
                  new MainMenuOption()
                  {
                     MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Add" },
                  }
               }
            }
         },
         {
            c=>c.RemoveItem,
            new ManagedCommand()
            {
               CanExecuteHandler = o=> loader.Instance.Data.Any(),
               ExecuteHandler = o=>
               {
                  var data = loader.Instance.Data;
                  data.RemoveAt(0);
               },
               Metadata = new MetadataCollection()
               {
```

```
                    new MainMenuOption()
                    {
                       MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "Remove" },
                    },
                 }
              }
           }
        };
        cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());

        var menuDefMgr = unityContainer.Resolve<MenuDefinitionManager>();
        menuDefMgr.RegisterMenuDefinition(new CustomMenuDefinition()
        {
           MenuPath = new MenuPath() { MenuConstants.File_1, "Items", "RegItems" },
        });

        loader.Instance.Data.CollectionChanged += Data_CollectionChanged;
     }

     void Data_CollectionChanged(object sender, System.Collections.Specialized.NotifyCollectionC
     {
        var cmdContainer = this.UnityContainer.Resolve<UICommandManager>().GetCommandContainer<I
        cmdContainer.RemoveItem.RaiseCanExecuteChanged();
     }

     void loader_ConfigLoaded(object sender, ConfigLoadedEventArgs<IMenuPluginConfig> e)
     {
        if (!e.ConfigLoadedSuccessfully)
        {
           e.Instance.Data = new ObservableCollection<string>();
        }
     }
  }

  public interface IMenuPluginConfig : IConfiguration
  {
     ObservableCollection<string> Data { get; set; }
  }

  public interface ICustomPluginCommands : ICommandContainer
  {
     ManagedCommand AddItem { get; }
     ManagedCommand RemoveItem { get; }
  }
}
```

## CustomMenuDefintion.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.UIComposition.Menus;
```

```
using System.Windows.Controls;
using Feat.UIComposition.Config;
using System.Collections.ObjectModel;
using Feat.UIComposition.Commanding;
using Feat.CAL.Services;
using Feat.Utils;
using System.Windows;

namespace UIPlugins.CustomMenu
{
  class CustomMenuDefinition : MenuDefinitionBase
  {
     protected override IMenuItemManager CreateMenuItemManger(MenuManager menuManger)
     {
        return new CustomMenuManager(this);
     }

     public override string UniqueName
     {
        get { return this.GetType().AssemblyQualifiedName + this.MenuPath; }
     }
  }

  class CustomMenuManager : IMenuItemManager
  {
     public ObservableCollection<string> items;
     private IDialogSvc dialogSvc;

     public CustomMenuDefinition Definition { get; set; }

     public CustomMenuManager(CustomMenuDefinition cmDef)
     {
        this.Definition = cmDef;
        this.dialogSvc = this.Definition.UnityContainer.Resolve<IDialogSvc>();
        this.items = this.Definition.UnityContainer.Resolve<IConfigurationManager>().
           GetConfiguration<IMenuPluginConfig>().Data;
        items.CollectionChanged += items_CollectionChanged;
     }

     void items_CollectionChanged(object sender, System.Collections.Specialized.NotifyCollectior
     {
        this.Invalidate();
     }

     MenuItem rootMenuItem;
     public void Attach(MenuManager menuMgr, ItemsControl parentMenuItem, object startingSiblinc
     {
        this.rootMenuItem = new MenuItem()
        {
           Header = "ItemList"
        };
        parentMenuItem.Items.Add(rootMenuItem);
        this.Invalidate();
```

```
    }

    public bool IsItemVisible
    {
       get
       {
          return this.items.Any();
       }
    }

    public event EventHandler HasItemsChanged;

    public void Invalidate()
    {
       this.rootMenuItem.Items.Clear();
       items.
          Select(s => new MenuItem()
          {
             Header = s,
             Command = new ManagedCommand()
             {
                ExecuteHandler = o =>
                   this.dialogSvc.ShowMessageBox(s, s, MessageBoxButton.OK, MessageBoxImage.Inf
             }
          }).
          ForEach(i => this.rootMenuItem.Items.Add(i));

       if (this.HasItemsChanged != null)
       {
          this.HasItemsChanged(this, EventArgs.Empty);
       }
    }
  }
}
```

# A simple plugin that consumes events

## Creating a plugin that consumes events

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll and Feat.Core.dll. The resulting dll from your project (along with any non .NET dependecies) must be placed in the Plugins directory in the FTC install location.

### The Plugin class.

The first step is to create a class that implements the IEventListener interface. This is the type that will be instantiated by Scene Composer.

```
public class SimpleEventsConsumer : IEventListener
{
}
```

### Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Name
{
  get { return "SC event consumer plugin"; }
}

public string Description
{
   get { return "A plugin that consumes SC events and shows a message box on each event"; }
}

public string Version
{
  get { return "1.1.0"; }
}
public void Initialize(IUnityContainer unityContainer)
{
   //resolve the IDialogSvc interface. We will use it to show simple message boxes.
  dlgSvc = unityContainer.Resolve<IDialogSvc>();
}
```

### Specifying what events should be handled by the plugin.

We implement the EventsToSubscribe property of the IEventListener interface. This property specifies a set of events that Scene Composer should pass to the plugin.

```
public EventId EventsToSubscribe
{
  get { return EventId.AppShutdown | EventId.AssetGeneration | EventId.ImportFinished | EventIc
}
```

## Implementing the event handlers of IEventListener

Next we implement event handler methods.

```
public void OnAssetGeneration(string assetFile, bool isTargetAsset, OperationStatus status)
{
  ShowMessage(String.Format("Asset generation for '{0}'  {1}.", assetFile, status));
}

public void OnImportFinished()
{
  ShowMessage("Import finished.");
}

public void OnShutdown()
{
  ShowMessage("App shutting down.");
}

public void OnSolutionChanged(FTC.SDKInterface.Solution.ISolution solution)
{
  ShowMessage("Solution changed!");
}
```

The ShowMessage method uses IDialogSvc to show a messagebox.

```
private void ShowMessage (string message)
{
  Application.Current.Dispatcher.BeginInvoke(new Action( () =>
  {
    dlgSvc.ShowMessageBox(message, "EventsPlugin");
  }), null );
}
```

The full code for the example:

## SimpleEventsConsumer.cs

```
using Feat.CAL.Services;
using FTC.SDKInterface.Plugins;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;
using System.Windows;

namespace SCEventsPlugin
{
  public class SimpleEventsConsumer : IEventListener
  {
     private IDialogSvc dlgSvc;

     private void ShowMessage(string message)
     {
        Application.Current.Dispatcher.BeginInvoke(new Action(() =>
        {
           dlgSvc.ShowMessageBox(message, "EventsPlugin");
        }), null);
     }

     public EventId EventsToSubscribe
     {
        get { return EventId.AppShutdown | EventId.AssetGeneration | EventId.ImportFinished | E
     }

     public void OnAssetGeneration(string assetFile, bool isTargetAsset, OperationStatus status
     {
        ShowMessage(String.Format("Asset generation for '{0}'  {1}.", assetFile, status));
     }

     public void OnImportFinished()
     {
        ShowMessage("Import finished.");
     }

     public void OnShutdown()
     {
        ShowMessage("App shutting down.");
     }

     public void OnSolutionChanged(FTC.SDKInterface.Solution.ISolution solution)
     {
        ShowMessage("Solution changed!");
     }

     public string Description
     {
        get { return "A plugin that consumes SC events and shows a message box on each event";
     }

     public IEnumerable<int> FeatureIds
     {
        get { return new int[0]; }
     }

     public void Initialize(FTC.SDKInterface.Services.IUnityContainer unityContainer)
     {
```

```
         //resolve the IDialogSvc interface. We will use it to show a message boxes.
         dlgSvc = unityContainer.Resolve<IDialogSvc>();
      }

      public string Name
      {
         get { return "SC event consumer plugin"; }
      }

      public string Version
      {
         get { return "1.1.0"; }
      }
   }
}
```