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; }
    }
 }
```

Revision #2
Created 2023-02-15 07:53:02 UTC by Tsuyoshi.Kato
Updated 2024-08-30 07:49:47 UTC by Tsuyoshi.Kato