Support
support@visualcomponents.com

Community
community.visualcomponents.net

Adding settings and configurations

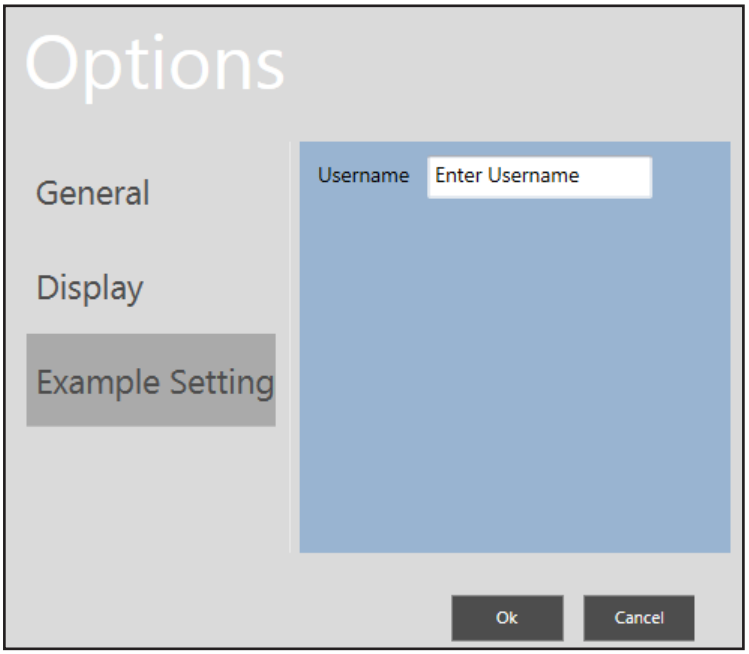
Next Generation | Version: November 20, 2015 | Example: Available upon request

Essentials can be extended to have additional settings and configurations.

The setup for creating settings and configurations is a straightforward process that incorporates the MVVM design pattern. A configuration file is created and exported as a type of ISection interface. That allows the configuration section to define its own type for containing configuration elements. The configuration settings are implemented in a ViewModel that is exported as a type of ISettingViewModel interface. That creates a new screen section of controls for Essentials located in the Options tab. The handling of requests and edits to configuration settings is performed by the IApplicationSettingService interface.

The topics covered in this tutorial include:

- Minimal setup for adding settings and configurations to Essentials.
- Implementation of custom configuration file and application service.



Creating a project in Visual Studio

Visual Studio 2013 and Visual C# are used to add a new section of settings in the backstage area of Essentials. Those settings will be linked to a configuration file in order to store and retrieve data for properties.

1. Run **Visual Studio** as an **administrator**.

2. Create a new **Class Library** project, and name the project **UX.CustomSettings** to id your assembly as an extension of the VisualComponents.UX namespace.

3. Access the properties of your project.

4. In the **Build** tab, set **Platform target** to either **x86** or **x64**.

5. Set **Output path** to be the path to your **Visual Components program files** in order for the MEF to discover and load your assembly at runtime.

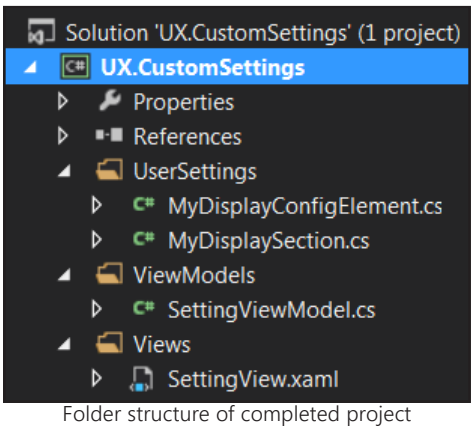
6. In the **Debug** tab, set **Start external program** to run the **.exe** file for Essentials.

7. Add references to the following assemblies and namespaces:

   - **Caliburn.Micro** that is available in your Essentials program files.
   - **Create3D.Shared** that is available in your Essentials program files.
   - **System.ComponentModel.Composition** that is available in the .NET Framework.
   - **System.Configuration** that is available in the .NET Framework.
   - **System.Windows.Interactivity** that is available in your Essentials program files.
   - **System.Xaml** that is available in the .NET Framework.
   - **UX.Shared** that is available in your Essentials program files.

8. In your project, add three **folders** to contain your **ViewModels**, **Views** and **configurations**.

9. (Optional) Delete **Class1.cs**.



Folder structure of completed project

Creating ViewModel

Your ViewModel class needs to derive from the Caliburn.Micro.Screen class and implement the ISettingViewModel interface. To be discovered by the MEF and loaded at runtime, your class must be exported as a type of ISettingViewModel. Metadata can be used to define the order and group of your ViewModel in the Options tab.

1. In your **ViewModels** folder, add a **Class** item and name that class **SettingViewModel**.

2. Type the following code to define your ViewModel.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Caliburn.Micro;
using System.ComponentModel.Composition;
using VisualComponents.Create3D;
using VisualComponents.UX.Shared;

namespace UX.CustomSettings.ViewModels
{
    [Export(typeof(ISettingViewModel))]
    [ExportMetadata("Order", 30)]
    public class SettingViewModel: Screen, ISettingViewModel
    {
        public void CancelApplyingSetting()
        {
            //throw new NotImplementedException();
        }

        public IHeaderModel Header
        {
            get { return new HeaderModel("Example Setting", ""); }
        }

        public string Id
        {
            get { return "ExampleSetting_Options_Backstage"; }
        }

        public void RefreshSettings()
        {
            //throw new NotImplementedException();
        }

        public void SaveAndApplySetting()
        {
            //throw new NotImplementedException();
        }
    }
}
```

Creating View

You can now setup a view of user controls that are relevant to the settings you want to implement in Essentials.

1. In your **Views** folder, add a **WPF User Control** item and name that control **SettingView**.

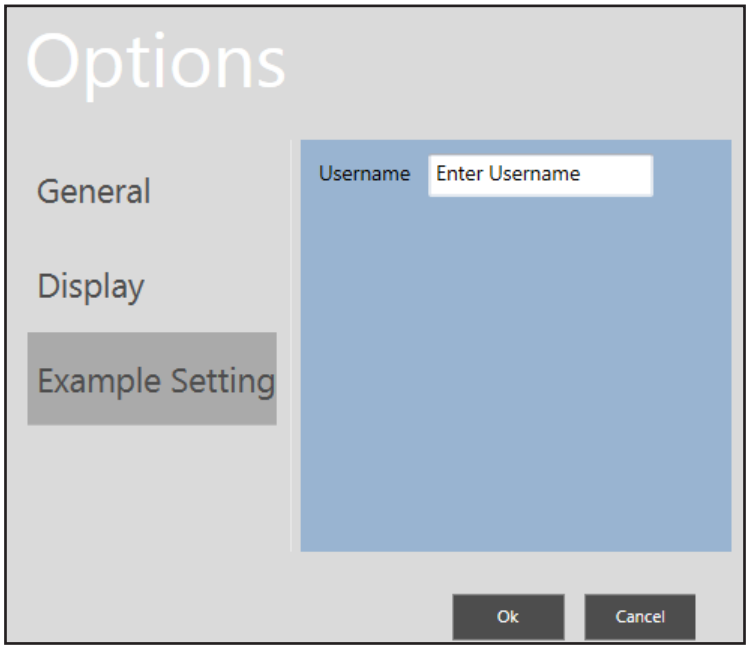
2. Add a **TextBlock** and **TextBox** to **SettingView**.

3. In the XAML editor, type the following code to modify your controls.

```
<UserControl x:Class="UX.CustomSettings.Views.SettingView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid Background="{DynamicResource {x:Static SystemColors.ActiveCaptionBrushKey}}">
        <TextBlock HorizontalAlignment="Left" Margin="10,10,0,0" TextWrapping="Wrap"
                   Text="Username" VerticalAlignment="Top"/>
        <TextBox HorizontalAlignment="Left" Height="23" Margin="68,7,0,0" TextWrapping="Wrap"
                 Text="Enter Username" VerticalAlignment="Top" Width="120"/>
    </Grid>
</UserControl>
```

4. Start debugging the application.

5. In **Essentials**, verify your section of settings has been added to the **Options** tab.



6. Stop debugging the application.

Creating configuration element

A configuration element needs to derive from the ConfigurationElement class and be customized to store properties that can be altered in Essentials.

1. In your **configuration** folder, add a **Class** item and name that class **MyCustomConfigElement**.

2. Type the following code to define your configuration element.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Configuration;
using VisualComponents.UX.Shared;

namespace UX.CustomSettings.UserSettings
{
    public sealed class MyCustomConfigElement: ConfigurationElement
    {
        public MyCustomConfigElement()
        {
        }

        [ConfigurationProperty("userName", DefaultValue = "", IsRequired = true)]
        public string UserName
        {
            get { return (string)this["userName"]; }
            set { this["userName"] = value; }
        }
    }
}
```

NOTE! In this example, the folder created to contain a configuration section and element is named UserSettings.

Creating configuration section

A configuration section needs to derive from the ConfigurationSection class and implement the ISection interface. To be discovered by the MEF and loaded at runtime or when requested, your configuration section must be exported as a type of ISection.

1. In your **configuration** folder, add a **Class** item and name that class **MyCustomConfigSection**.

The ISection interface has a SectionType property that allows you to define the type of your configuration section. For example, you can create an interface that contains properties for configuration elements.

2. Type the following code to create an **interface** that has a **property** with a value type of **MyCustomConfigElement**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel.Composition;
using System.Configuration;
using VisualComponents.Create3D;
using VisualComponents.UX.Shared;

namespace UX.CustomSettings.UserSettings
{
    public interface IExampleCustomSection : ISection
    {
        MyCustomConfigElement CustomElement { get; }
    }
}
```

3. Type the following code to define your configuration section.

```
...
    [Export(typeof(ISection))]
    public sealed class MyCustomConfigSection: ConfigurationSection, IExampleCustomSection
    {
        public MyCustomConfigSection()
        {
        }

        [ConfigurationProperty("customElement")]
        public MyCustomConfigElement CustomElement
        {
            get { return ((MyCustomConfigElement)this["customElement"]); }
            set { this["customElement"] = value; }
        }
    }
...
```

Requesting services

Your configuration section needs to implement the ISection interface in order to be requested and retrieved by the application setting service in Essentials.

1. In the **MyCustomConfigSection** class, implement the **ISection** interface and define the added members.

```
[Export(typeof(ISection))]
public sealed class MyCustomConfigSection: ConfigurationSection, IExampleCustomSection
{
    public MyCustomConfigSection()
    {
    }

    [ConfigurationProperty("customElement")]
    public MyCustomConfigElement CustomElement
    {
        get { return ((MyCustomConfigElement)this["customElement"]); }
        set { this["customElement"] = value; }
    }

    //check and update version of data
    private const string _versionPropertyName = "SectionVersion";
    [ConfigurationProperty(_versionPropertyName, DefaultValue = 1, IsRequired = true)]
    public int SectionVersion
    {
        get { return ((int)base[_versionPropertyName]); }
        protected set {base[_versionPropertyName] = value; }
    }

    public void CheckVersion()
    {
        const int currentVer = 1;
        if (currentVer > SectionVersion)
        {
            SectionVersion = currentVer;
        }
    }

    public ISection GetSection(Configuration userSettingConfiguration)
    {
        return ConfigurationHelper.GetConfigurationSection<MyCustomConfigSection>(SectionName, userSettingConfiguration);
    }

    public string SectionName
    { get { return "MyCustomSection"; } }

    public Type SectionType
    { get { return typeof(IExampleCustomSection); } }
}
```

Implementing configuration settings

You can implement your configuration settings in a ViewModel and handle configuration requests by using the IApplicationSettingService interface.

1. In the **SettingViewModel** class, add a **using** statement for your **configuration namespace** and **System.Configuration**.

```
...
using VisualComponents.Create3D;
using UX.CustomSettings.UserSettings;
using System.Configuration;
```

2. Type the following code to construct your ViewModel and to get handles for **IApplicationSettingService** and **MyCustomConfigSection**. Notice that a private read-only handle for IApplicationSettingService is used to get a configuration file which is then passed as an argument for retrieving your custom section.

```
namespace UX.CustomSettings.ViewModels
{
    [Export(typeof(ISettingViewModel))]
    [ExportMetadata("Order", 30)]
    public class SettingViewModel : Screen, ISettingViewModel
    {
        private IApplicationSettingService _appSettings;
        private readonly IApplicationSettingService _applicationSettingService;

        [ImportingConstructor]
        public SettingViewModel([Import]IApplicationSettingService appSettings, [Import]IApplicationSettingService appSettingService)
        {
            _appSettings = appSettings;
            _applicationSettingService = appSettingService;
        }
...
```

TECHNICAL! The MEF automatically supplies an IApplicationSettingService value when composing your part. That value has a reference to a user/client configuration file as well as a collection of custom configuration settings.

Updating property values

You need to further define your ViewModel to handle updates to configuration settings.

1. In the **SettingViewModel** class, add the following code.

```
//logic for updating config property UserName
private bool _isUserNameSettingChanged = false;
public bool IsSettingChanged
{
    get { return _isUserNameSettingChanged; }
}

private string _userName;
public string UserName
{
    get { return _userName; }
    set
    {
        if (_userName != value)
        {
            _userName = value;
            _isUserNameSettingChanged = true;
            NotifyOfPropertyChange(() => UserName);
        }
    }
}
```

2. Type the following code to modify the **CancelApplySetting()** and **SaveAndApplySetting()** methods.

```
//revert to stored value in config file
public void CancelApplyingSetting()
{
    if (_isUserNameSettingChanged)
    {
        Configuration config = _applicationSettingService.UserSettingConfigurationFile;
        IExampleCustomSection customSection = _appSettings.GetSection<IExampleCustomSection>(config);
        UserName = customSection.CustomElement.UserName;
    }
}

//store new value and update config file
public void SaveAndApplySetting()
{
    if (_isUserNameSettingChanged)
    {
        Configuration config = _applicationSettingService.UserSettingConfigurationFile;
        IExampleCustomSection customSection = _appSettings.GetSection<IExampleCustomSection>(config);
        customSection.CustomElement.UserName = UserName;

        _appSettings.RefreshSection(customSection, config);
        _isUserNameSettingChanged = false;
    }
}
```

Updating controls

1. In **SettingView**, access the **XAML** editor.

2. In **TextBox**, add an **x:Name** attribute to reference the **UserName** property.

```
<TextBox x:Name="UserName" HorizontalAlignment="Left" Height="23" Margin="68,7,0,0" TextWrapping="Wrap"
        VerticalAlignment="Top" Width="120"/>
```

3. In the **SettingViewModel** class, modify **OnInitialize()** to load the stored value of **UseName** in your configuration file.

```
//get stored value when first viewing screen
protected override void OnInitialize()
{
    Configuration config = _applicationSettingService.UserSettingConfigurationFile;
    IExampleCustomSection customSection = _appSettings.GetSection<IExampleCustomSection>(config);
    UserName = customSection.CustomElement.UserName;

    _isUserNameSettingChanged = false;
    base.OnInitialize();
}
```

4. Start debugging the application.

5. In **Essentials**, access the **Options** tab, and then in the **Example Setting** section, verify that **Username** has a preloaded value of **Enter your Username**.

6. In **Username** type **Example**, and then click **OK**.

7. Go back to **Example Setting** and verify your update was saved and stored in the UserName configuration setting.

This concludes the tutorial.