Adding Settings to Windows Terminal

Adding a setting to Windows Terminal is fairly straightforward. This guide serves as a reference on how to add a setting.

1. Terminal Settings Model

The Terminal Settings Model (`Microsoft.Terminal.Settings.Model`) is responsible for (de)serializing and exposing settings.

`INHERITABLE_SETTING` macro

The `INHERITABLE_SETTING` macro can be used to implement inheritance for your new setting and store the setting in the settings model. It takes three parameters:

- `type` : the type that the setting will be stored as
- `name` : the name of the variable for storage
- `defaultValue` : the value to use if the user does not define the setting anywhere

Adding a Profile setting

This tutorial will add `CloseOnExitMode CloseOnExit` as a profile setting.

1. In `Profile.h` , declare/define the setting:

```
INHERITABLE_SETTING(CloseOnExitMode, CloseOnExit, CloseOnExitMode::Graceful)
```

2. In `Profile.idl` , expose the setting via WinRT:

```
Boolean HasCloseOnExit();
void ClearCloseOnExit();
CloseOnExitMode CloseOnExit;
```

3. In `Profile.cpp` , add (de)serialization and copy logic:

```
// Top of file:
// - Add the serialization key
static constexpr std::string_view CloseOnExitKey{ "closeOnExit" };

// CopySettings() or Copy():
// - The setting is exposed in the Settings UI
profile->_CloseOnExit = source->_CloseOnExit;

// LayerJson():
// - get the value from the JSON
JsonUtils::GetValueForKey(json, CloseOnExitKey, _CloseOnExit);

// ToJson():
// - write the value to the JSON
JsonUtils::SetValueForKey(json, CloseOnExitKey, _CloseOnExit);
```

- If the setting is not a primitive type, in `TerminalSettingsSerializationHelpers.h` add (de)serialization logic for the accepted values:

```
// For enum values...
JSON_ENUM_MAPPER(::winrt::Microsoft::Terminal::Settings::Model::CloseOnExitMode)
{
    JSON_MAPPINGS(3) = {
        pair_type{ "always", ValueType::Always },
        pair_type{ "graceful", ValueType::Graceful },
        pair_type{ "never", ValueType::Never },
    };
};

// For enum flag values...
JSON_FLAG_MAPPER(::winrt::Microsoft::Terminal::TerminalControl::CopyFormat)
{
    JSON_MAPPINGS(5) = {
        pair_type{ "none", AllClear },
        pair_type{ "html", ValueType::HTML },
        pair_type{ "rtf", ValueType::RTF },
        pair_type{ "all", AllSet },
    };
};

// NOTE: This is also where you can add functionality for...
// - overloaded type support (i.e. accept a bool and an enum)
// - custom (de)serialization logic (i.e. coordinates)
```

Adding a Global setting

Follow the "adding a Profile setting" instructions above, but do it on the `GlobalAppSettings` files.

Adding an Action

This tutorial will add the `openSettings` action.

1. In `KeyMapping.idl`, declare the action:

```
// Add the action to ShortcutAction
enum ShortcutAction
{
    OpenSettings
}
```

2. In `ActionAndArgs.cpp`, add serialization logic:

```
// Top of file:
// - Add the serialization key
static constexpr std::string_view OpenSettingsKey{ "openSettings" };
```

```
// ActionKeyNamesMap:
// - map the new enum to the json key
{ OpenSettingsKey, ShortcutAction::OpenSettings },
```

3. If the action should automatically generate a name when it appears in the Command Palette...

```
// In ActionAndArgs.cpp GenerateName() --> GeneratedActionNames
{ ShortcutAction::OpenSettings, RS_(L"OpenSettingsCommandKey") },

// In Resources.resw for Microsoft.Terminal.Settings.Model.Lib,
// add the generated name
// NOTE: Visual Studio presents the resw file as a table.
//       If you choose to edit the file with a text editor,
//       the code should look something like this...
<data name="OpenSettingsCommandKey" xml:space="preserve">
    <value>Open settings file</value>
</data>
```

4. If the action supports arguments...

- In `ActionArgs.idl`, declare the arguments

```
[default_interface] runtimeclass OpenSettingsArgs : IActionArgs
{
    // this declares the "target" arg
    SettingsTarget Target { get; };
};
```

- In `ActionArgs.h`, define the new runtime class

```
struct OpenSettingsArgs : public OpenSettingsArgsT<OpenSettingsArgs>
{
    OpenSettingsArgs() = default;

    // adds a getter/setter for your argument, and defines the json key
    WINRT_PROPERTY(SettingsTarget, Target, SettingsTarget::SettingsFile);
    static constexpr std::string_view TargetKey{ "target" };

    public:
        hstring GenerateName() const;

        bool Equals(const IActionArgs& other)
        {
            auto otherAsUs = other.try_as<OpenSettingsArgs>();
            if (otherAsUs)
            {
                return otherAsUs->_Target == _Target;
            }
            return false;
```

```
        };

        static FromJsonResult FromJson(const Json::Value& json)
        {
            // LOAD BEARING: Not using make_self here _will_ break you in the
future!
            auto args = winrt::make_self<OpenSettingsArgs>();
            JsonUtils::GetValueForKey(json, TargetKey, args->_Target);
            return { *args, {} };
        }

        IActionArgs Copy() const
        {
            auto copy{ winrt::make_self<OpenSettingsArgs>() };
            copy->_Target = _Target;
            return *copy;
        }
};
```

- In `ActionArgs.cpp`, define `GenerateName()`. This is used to automatically generate a name when it appears in the Command Palette.
- In `ActionAndArgs.cpp`, add serialization logic:

```
// ActionKeyNamesMap --> argParsers
{ ShortcutAction::OpenSettings, OpenSettingsArgs::FromJson },
```

Adding an Action Argument

Follow step 3 from the "adding an Action" instructions above, but modify the relevant `ActionArgs` files.

2. Setting Functionality

Now that the Terminal Settings Model is updated, Windows Terminal can read and write to the settings file. This section covers how to add functionality to your newly created setting.

App-level settings

App-level settings are settings that affect the frame of Windows Terminal. Generally, these tend to be global settings. The `TerminalApp` project is responsible for presenting the frame of Windows Terminal. A few files of interest include:

- `TerminalPage`: XAML control responsible for the look and feel of Windows Terminal
- `AppLogic`: WinRT class responsible for window-related issues (i.e. the titlebar, focus mode, etc...)

Both have access to a `CascadiaSettings` object, for you to read the loaded setting and update Windows Terminal appropriately.

Terminal-level settings

Terminal-level settings are settings that affect a shell session. Generally, these tend to be profile settings. The `TerminalApp` project is responsible for packaging this settings from the Terminal Settings Model to the terminal

instance. There are two kinds of settings here:

- `IControlSettings`:
  - These are settings that affect the `TerminalControl` (a XAML control that hosts a shell session).
  - Examples include background image customization, interactivity behavior (i.e. selection), acrylic and font customization.
  - The `TerminalControl` project has access to these settings via a saved `IControlSettings` member.
- `ICoreSettings`:
  - These are settings that affect the `TerminalCore` (a lower level object that interacts with the text buffer).
  - Examples include initial size, history size, and cursor customization.
  - The `TerminalCore` project has access to these settings via a saved `ICoreSettings` member.

`TerminalApp` packages these settings into a `TerminalSettings : IControlSettings, ICoreSettings` object upon creating a new terminal instance. To do so, you must submit the following changes:

- Declare the setting in `IControlSettings.idl` or `ICoreSettings.idl` (whichever is relevant to your setting). If your setting is an enum setting, declare the enum here instead of in the `TerminalSettingsModel` project.
- In `TerminalSettings.h`, declare/define the setting...

```
// The WINRT_PROPERTY macro declares/defines a getter setter for the setting.
// Like INHERITABLE_SETTING, it takes in a type, name, and defaultValue.
WINRT_PROPERTY(bool, UseAcrylic, false);
```

- In `TerminalSettings.cpp`...
  - update `_ApplyProfileSettings` for profile settings
  - update `_ApplyGlobalSettings` for global settings
  - If additional processing is necessary, that would happen here. For example, `backgroundImageAlignment` is stored as a `ConvergedAlignment` in the Terminal Settings Model, but converted into XAML's separate horizontal and vertical alignment enums for packaging.

Actions

Actions are packaged as an `ActionAndArgs` object, then handled in `TerminalApp`. To add functionality for actions...

- In the `ShortcutActionDispatch` files, dispatch an event when the action occurs...

```
// ShortcutActionDispatch.idl
event Windows.Foundation.TypedEventHandler<ShortcutActionDispatch,
Microsoft.Terminal.Settings.Model.ActionEventArgs> OpenSettings;

// ShortcutActionDispatch.h
TYPED_EVENT(OpenSettings, TerminalApp::ShortcutActionDispatch,
Microsoft::Terminal::Settings::Model::ActionEventArgs);

// ShortcutActionDispatch.cpp --> DoAction()
```

```
// - dispatch the appropriate event
case ShortcutAction::OpenSettings:
{
    _OpenSettingsHandlers(*this, eventArgs);
    break;
}
```

- In `TerminalPage` files, handle the event...

```
// TerminalPage.h
// - declare the handler
void _HandleOpenSettings(const IInspectable& sender, const
Microsoft::Terminal::Settings::Model::ActionEventArgs& args);

// TerminalPage.cpp --> _RegisterActionCallbacks()
// - register the handler
_actionDispatch->OpenSettings({ this, &TerminalPage::_HandleOpenSettings });

// AppActionHandlers.cpp
// - direct the function to the right place and call a helper function
void TerminalPage::_HandleOpenSettings(const IInspectable& /*sender*/,
                                       const ActionEventArgs& args)
{
    // NOTE: this if-statement can be omitted if the action does not support
arguments
    if (const auto& realArgs = args.ActionArgs().try_as<OpenSettingsArgs>())
    {
        _LaunchSettings(realArgs.Target());
        args.Handled(true);
    }
}
```

`AppActionHandlers` vary based on the action you want to perform. A few useful helper functions include:

- `_GetFocusedTab()` : retrieves the focused tab
- `_GetActiveControl()` : retrieves the active terminal control
- `_GetTerminalTabImpl()` : tries to cast the given tab as a `TerminalTab` (a tab that hosts a terminal instance)

3. Settings UI

Exposing Enum Settings

If the new setting supports enums, you need to expose a map of the enum and the respective value in the Terminal Settings Model's `EnumMappings` :

```
// EnumMappings.idl
static Windows.Foundation.Collections.IMap<String,
Microsoft.Terminal.Settings.Model.CloseOnExitMode> CloseOnExitMode { get; };

// EnumMappings.h
```

```
static winrt::Windows::Foundation::Collections::IMap<winrt::hstring,
CloseOnExitMode> CloseOnExitMode();

// EnumMappings.cpp
// - this macro leverages the json enum mapper in
TerminalSettingsSerializationHelper to expose
//   the mapped values across project boundaries
DEFINE_ENUM_MAP(Model::CloseOnExitMode, CloseOnExitMode);
```

Binding and Localizing the Enum Setting

Find the page in the Settings UI that the new setting fits best in. In this example, we are adding `LaunchMode`.

1. In `Launch.idl`, expose the bindable setting...

```
// Expose the current value for the setting
IInspectable CurrentLaunchMode;

// Expose the list of possible values
Windows.Foundation.Collections.IObservableVector<Microsoft.Terminal.Settings.Editor.Enu
 LaunchModeList { get; };
```

2. In `Launch.h`, declare the bindable enum setting...

```
// the GETSET_BINDABLE_ENUM_SETTING macro accepts...
// - name: the name of the setting
// - enumType: the type of the setting
// - settingsModelName: how to retrieve the setting (use State() to get access to
the settings model)
// - settingNameInModel: the name of the setting in the terminal settings model
GETSET_BINDABLE_ENUM_SETTING(LaunchMode, Model::LaunchMode,
State().Settings().GlobalSettings, LaunchMode);
```

3. In `Launch.cpp`, populate these functions...

```
// Constructor (after InitializeComponent())
// the INITIALIZE_BINDABLE_ENUM_SETTING macro accepts...
// - name: the name of the setting
// - enumMappingsName: the name from the TerminalSettingsModel's EnumMappings
// - enumType: the type for the enum
// - resourceSectionAndType: prefix for the localization
// - resourceProperty: postfix for the localization
INITIALIZE_BINDABLE_ENUM_SETTING(LaunchMode, LaunchMode, LaunchMode,
L"Globals_LaunchMode", L"Content");
```

4. In `Resources.resw` for Microsoft.Terminal.Settings.Editor, add the localized text to expose each enum value. Use the following format: `<SettingGroup>_<SettingName><EnumValue>.Content`

- `SettingGroup`:
  - `Globals` for global settings

    - `Profile` for profile settings
- `SettingName`:
    - the Pascal-case format for the setting type (i.e. `LaunchMode` for `"launchMode"`)
- `EnumValue`:
    - the json key for the setting value, but with the first letter capitalized (i.e. `Focus` for `"focus"`)
- The resulting resw key should look something like this `Globals_LaunchModeFocus.Content`
- This is the text that will be used in your control

Updating the UI

When adding a setting to the UI, make sure you follow the UWP design guidance.

Enum Settings

Now, create a XAML control in the relevant XAML file. Use the following tips and tricks to style everything appropriately:

- Wrap the control in a `ContentPresenter` adhering to the `SettingContainerStyle` style
- Bind `SelectedItem` to the relevant `Current<Setting>` (i.e. `CurrentLaunchMode`). Ensure it's a TwoWay binding
- Bind `ItemsSource` to `<Setting>List` (i.e. `LaunchModeList`)
- Set the ItemTemplate to the `Enum<ControlType>Template` (i.e. `EnumRadioButtonTemplate` for radio buttons)
- Set the style to the appropriate one in `CommonResources.xaml`

```
<!--Launch Mode-->
<ContentPresenter Style="{StaticResource SettingContainerStyle}">
    <muxc:RadioButtons x:Uid="Globals_LaunchMode"
                       SelectedItem="{x:Bind CurrentLaunchMode, Mode="TwoWay"}"
                       ItemsSource="{x:Bind LaunchModeList}"
                       ItemTemplate="{StaticResource EnumRadioButtonTemplate}"
                       Style="{StaticResource RadioButtonsSettingStyle}"/>
</ContentPresenter>
```

To add any localized text, add a `x:Uid`, and access the relevant property via the Resources.resw file. For example, `Globals_LaunchMode.Header` sets the header for this control. You can also set the tooltip text like this: `Globals_DefaultProfile.[using:Windows.UI.Xaml.Controls]ToolTipService.ToolTip`.

Non-Enum Settings

Continue to reference `CommonResources.xaml` for appropriate styling and wrap the control with a similar `ContentPresenter`. However, instead of binding to the `Current<Setting>` and `<Setting>List`, bind directly to the setting via the state. Binding a setting like `altGrAliasing` should look something like this:

```
<!--AltGr Aliasing-->
<ContentPresenter Style="{StaticResource SettingContainerStyle}">
    <CheckBox x:Uid="Profile_AltGrAliasing"
              IsChecked="{x:Bind State.Profile.AltGrAliasing, Mode=TwoWay}"
              Style="{StaticResource CheckBoxSettingStyle}"/>
</ContentPresenter>
```

Profile Settings

If you are specifically adding a Profile setting, in addition to the steps above, you need to make the setting observable by modifying the `Profiles` files...

```
// Profiles.idl --> ProfileViewModel
// - this declares the setting as observable using the type and the name of the
setting
OBSERVABLE_PROJECTED_SETTING(Microsoft.Terminal.Settings.Model.CloseOnExitMode,
CloseOnExit);

// Profiles.h --> ProfileViewModel
// - this defines the setting as observable off of the _profile object
OBSERVABLE_PROJECTED_SETTING(_profile, CloseOnExit);

// Profiles.h --> ProfileViewModel
// - if the setting cannot be inherited by another profile (aka missing the Clear()
function), use the following macro instead:
PERMANENT_OBSERVABLE_PROJECTED_SETTING(_profile, Guid);
```

The `ProfilePageNavigationState` holds a `ProfileViewModel`, which wraps the `Profile` object from the Terminal Settings Model. The `ProfileViewModel` makes all of the profile settings observable.

Actions

Actions are not yet supported in the Settings UI.