Terminal Settings

- author: Mike Griese **migrie**
- created on: 2018-Oct-23

Abstract

This spec will outline how various terminal frontends will be able to interact with the settings for the terminal.

Terminology

- **Frontend** or **Application Layer**: This is the end-user experience. This could be a Terminal Application (ex. Project Cascadia) or something that's embedding a terminal window inside of it (ex. Visual Studio). These frontends consume the terminal component as an atomic unit.
- **Component Layer**: This is the UI framework-dependent implementation of the Terminal component. As planned currently, this is either the UWP or WPF Terminal component.
- **Terminal Layer**: This is the shared core implementation of the terminal. This is the Terminal Connection, Parser/Adapter, Buffer, and Renderer (but not the UX-dependant RenderEngine).

User Stories

1. "Project Cascadia" should be able to have both global settings (such as scrollbar styling) and settings that are stored per-profile (such as commandline, color scheme, etc.)
2. "Project Cascadia" should be able to load these settings at boot, use them to create terminal instances, be able to edit them, and be able to save them back.
3. "Project Cascadia" should be able to have terminal instances reflect the changes to the settings when the settings are changed.
4. "Project Cascadia" should be able to host panes/tabs with different profiles set at the same time.
5. Visual Studio should be able to persist and edit settings globally, without the need for a globals/profiles structure.
6. The Terminal should be able to read information from a settings structure that's independent of how it's persisted / implemented by the Application
7. The Component should be able to have its own settings independent of the application that's embedding it, such as font size and face, scrollbar visibility, etc. These should be settings that are specific to the component, and the Terminal should logically be unaffected by these settings.

Details

Some settings will need to be Application-specific, some will need to be component-specific, and some are terminal-specific. For example:

Terminal	Component	Frontend
Color Table	Font Face, size	Status Line Visibility, contents
Cursor Color	Scrollbar Visibility	~~Window Size~~[1]
History Size		
Buffer Size [1]		

- [1] I believe only the "Default" or "Initial" buffer size should be the one we truly store in the settings. When the app first boots up, it can use that value to with the font size to figure out how big its window should be. When additional tabs/panes are created, they should inherit the size of the existing window. Similarly, VS could first calculate how much space it has available, then override that value when creating the terminal.

Project Cascadia needs to be able to persist settings as a bipartite globals-profiles structure. VS needs to be able to persist settings just as a simple set of global settings.

When the application needs to retrieve these settings, they need to use them as a tripartite structure: frontend-component-terminal settings.

Each frontend will have its own set of settings. Each component implementation will also need to have some settings that control it. The terminal also will have some settings specific to the terminal.

Globals and Profiles

With *nix-like terminals, settings are typically structured as two parts: Globals and Profiles.

Globals are settings that affect the entirety of the terminal application. They wouldn't be different from one pane to the next. An example is the Terminal KeyBindings - these should be the same for all tabs/panes that are running as a part of the terminal application.

Profiles are what you might consider per-application settings. These are settings that can be different from one terminal instance to the next. One of the primary differentiators between profiles is the commandline used to start the terminal instance - this enables the user to have both a `cmd` profile and a `powershell` profile, for example. Things like the color table/scheme, font size, history length, these all change per-profile.

Per-Profile	Globals
Color Table	Keybindings
Cursor Color	Scrollbar Visibility
History Size	Status Line Visibility, contents
Font Face, size	Window Size
Shell Commandline	

Simple Settings

An application like VS might not even care about settings profiles. They should be able to persist the settings as just a singular entity, and change those as needed, without the additional overhead. Profiles will be something that's more specific to Project Cascadia.

Interface Descriptions

```
public class TerminalSettings
{
    Color DefaultForeground;
    Color DefaultBackground;
    Color[] ColorTable;
    Coord? Dimensions;
    int HistorySize;
    Color CursorColor;
    CursorShape CursorShape;
}

public interface IComponentSettings
{
    TerminalSettings TerminalSettings { get; }
}

public interface IApplicationSettings
{
    IComponentSettings ComponentSettings { get; }
}
```

The Application can store whatever settings it wants in its implementation of `IApplicationSettings`. When it instantiates a Terminal Component, it will pass its `IComponentSettings` to it.

The component will retrieve whatever settings it wants from that object, and then pass the `TerminalSettings` to the Terminal it creates.

The frontend will be able to get/set its settings from the `IApplicationSettings` implementation. The frontend will be able to create components using the

`IComponentSettings` in its `IApplicationSettings`. The Component will then create the Terminal using the `TerminalSettings`.

Project Cascadia Settings Details The `CascadiaSettings` will store the settings as two parts: * A set of global data & settings * A list of Profiles, that each have more data

When Cascadia starts up, it'll load all the settings, including the Globals and profiles. The Globals will also tell us which profile is the "Default" profile we should use to instantiate the terminal. Using the globals and the Profile, it'll convert those to a `ApplicationSettings : IApplicationSettings`. It'll read data from that `ApplicationSettings` to initialize things it needs to know. * It'll determine whether or not to display the status line * It'll query the Component settings for the default size of the component, so it knows how big of a space it needs to reserve for it

It'll then instantiate a `UWPTerminalComponent` and pass it the `UWPComponentSettings`.

This is a rough draft of what these members might all be like.

```
class CascadiaSettings
{
    void LoadAll();
    void SaveAll();
    GlobalAppSettings Globals;
    List<Profile> Profiles;
    ApplicationSettings ToSettings(GlobalAppSettings globals, Profile profile);
    void Update(ApplicationSettings appSettings, GUID profileID);
}
class Profile
{
    GUID ProfileGuid;
    string Name;
    string Commandline;
    TerminalSettings TerminalSettings;
    string FontFace;
    int FontSize;
    float acrylicTransparency;
    bool useAcrylic;
}
class GlobalAppSettings
{
    GUID defaultProfile;
    Keybindings keybindings;
    bool showScrollbars;
    bool showStatusline;
}
```

```
class ApplicationSettings : IApplicationSettings
{
    UWPComponentSettings ComponentSettings;
    Keybindings keybindings;
    bool showStatusline;
}
class UWPComponentSettings : IComponentSettings
{
    Point GetDefaultComponentSize();
    TerminalSettings TerminalSettings;
    string FontFace;
    int FontSize;
    bool showScrollbars;
    float acrylicTransparency;
    bool useAcrylic;
}
```

Updating Settings

What happens when the user changes the application's settings? The App, Component, and Terminal might all need to update their settings. The component will expose a `UpdateSettings()` method that will cause the Component and Terminal to reload the settings from their settings objects.

```
interface ITerminalComponent
{
    void UpdateSettings(IComponentSettings componentSettings);
}
partial class UWPTerminalComponent : ITerminalComponent
{
    void UpdateSettings(IComponentSettings componentSettings)
    {
        // Recalculate GlyphTypeFace
        // Recalculate rows/cols using current geometry and typeface
        // Update our terminal instance:
        terminal.UpdateSettings(componentSettings.TerminalSettings);
    }
}
```

Updating settings in Project Cascadia However, when Cascadia's settings change, we're going to possibly change some global settings and possibly some profile settings. The profile's that are changed may or may not be currently active.

> Say we have two different panes open with different profiles (A and B). What happens if we change the settings for one profile's font and

not the other's? ~~We resize the height of the terminal to account for the change in height of the win~~

We should never change the window size in response to a settings change if there is more than one tab/pane open. > never?

Cascadia would have to maintain a mapping of which components have which profiles:

```
class CascadiaTerminalInstance
{
    GUID ProfileGuid;
    UWPTerminalComponent component;
}
```

Then, when the settings are closed, it'll enumerate all of the components it has loaded, and apply the updated settings to them. It'll do this by looking up the profile GUID of the component, then getting the `ApplicationSettings` for the profile, then calling `UpdateSettings` on the component.

~~We need to have a way so that only the currently foreground component can change the window size.~~ I don't like that - if we change the font size, we should just recalculate how many characters can fit in the current window size.

Questions / TODO

- How does this interplay with setting properties of the terminal component in XAML?
  - I would think that the component would load the XAML properties first, and if the controlling application calls `UpdateSettings` on the component, then those in-XAML properties would likely get overwritten.
  - It's not necessary to create the component with a `IComponentSettings`, nor is it necessary to call `UpdateSettings`. If you wanted to create a trivial settings-less terminal component entirely in XAML, go right ahead.
  - Any settings that *are* exposed through XAML properties *should* also be exposed in the component's settings implementation as well.
    - Can that be enforced any way? I doubt it.