JSON fragment extensions in Windows Terminal

10/06/2022

JSON fragment extensions are snippets of JSON that application developers can write to add new profiles to users' settings, or even modify certain existing profiles. They can also be used to add new color schemes to users' settings.

Structure of the JSON files

The JSON file should be split up into 2 lists, one for profiles and one for schemes. Here is an example of a json file that adds a new profile, modifies an existing profile, and creates a new color scheme:

JSON

```
{
  "profiles": [
    {
      // update a profile by using its GUID
      "updates": "{2ece5bfe-50ed-5f3a-ab87-5cd4baafed2b}",
      "fontSize": 16,
      "fontWeight": "thin"
    },
    {
      // create a new profile
      "name": "Cool Profile",
      "commandline": "powershell.exe",
      "antialiasingMode": "aliased",
      "fontWeight": "bold",
      "colorScheme": "Postmodern Tango Light"
    }
  ],
  "schemes": [
    {
      // create a new color scheme
      "name": "Postmodern Tango Light",
      "black": "#0C0C0C",
      "red": "#C50F1F",
      "green": "#13A10E",
      "yellow": "#C19C00",
      "blue": "#0037DA",
      "purple": "#881798",
      "cyan": "#3A96DD",
      "white": "#CCCCCC",
      "brightBlack": "#767676",
      "brightRed": "#E74856",
```

```
        "brightGreen": "#16C60C",
        "brightYellow": "#F9F1A5",
        "brightBlue": "#3B78FF",
        "brightPurple": "#B4009E",
        "brightCyan": "#61D6D6",
        "brightWhite": "#F2F2F2"
      }
    ]
}
```

The first item in the `"profiles"` list updates an existing profile, identifying the profile it wishes to update via the GUID provided to the `"updates"` field (details on how to obtain the GUID are in the next section). The second item in that list creates a new profile called "Cool Profile".

In the `"schemes"` list, a new color scheme called "Postmodern Tango Light" is defined, and can be subsequently referenced by the user in their settings file or in this JSON file itself (notice that "Cool Profile" uses this newly defined color scheme).

Of course, if the developer only wishes to add/modify profiles without adding color schemes (and vice-versa), only the relevant list needs to be present and the other list can be omitted.

> **Note**
>
> If you plan to use PowerShell to generate fragments, you must use `-Encoding Utf8`:
>
> PowerShell
>
> ```
> # BAD: PowerShell uses UTF16LE by default
> Write-Output $fragmentJson > $fragmentPath
> ```
>
> PowerShell
>
> ```
> # GOOD: Uses UTF8, so Terminal will read this
> Write-Output $fragmentJson | Out-File $fragmentPath -Encoding Utf8
> ```
>
> If you are using VS Code to edit the JSON, UTF8 is the default, but you can confirm in the bottom status bar.

Profile GUIDs

As previously stated profile GUIDs are a way to reference profiles and let users update and extend them without worrying about location or name changes. The only profiles that can be

modified through fragments are the default profiles, Command Prompt and PowerShell, as well as dynamic profiles. Providing a GUID is optional, however, strongly encouraged.

The GUIDs are generated using a Version 5 UUID generator which supports BOM-less UTF-16LE encoding.

The namespace GUID for Windows Terminal in case of profiles created by plugins and fragments is `{f65ddb7e-706b-4499-8a50-40313caf510a}`. Profiles created by the Windows Terminal Team use a separate GUID (`{2bde4a90-d05f-401c-9492-e40884ead1d8}`). This is done to disambiguate profiles created by the Windows Terminal Team from profiles created by plugins or fragments so they can never accidentally collide.

How to determine the GUID of an existing profile

To determine the GUID of a profile to be updated it depends on what kind of profile it is:

A profile shipped by a third party stored in a standard Windows Terminal Fragment location requires the profile & fragment namespace GUID `{f65ddb7e-706b-4499-8a50-40313caf510a}`, the application namespace GUID, and the profile name. For a profile fragment named 'Git Bash' supplied by the application 'Git' the generated GUID is: `{2ece5bfe-50ed-5f3a-ab87-5cd4baafed2b}`.

A profile automatically generated by Windows Terminal requires the Windows Terminal internal GUID `{2bde4a90-d05f-401c-9492-e40884ead1d8}` and the profile name. For a profile named 'Ubuntu' automatically generated during WSL installation, the resulting GUID is: `{2c4de342-38b7-51cf-b940-2309a097f518}`. In contrast to the previous fragment example there is no 'application name' involved.

Generating a new profile GUID

To generate a GUID for a completely new profile prior to distributing it you can use the following Python 3 example. It generates a GUID based on the profile & fragment namespace GUID for a profile called 'Git Bash' stored in a standard Windows Terminal Fragments folder under the 'Git' application name, conveniently matching the sanity check.

Python

```python
import uuid

# The Windows Terminal namespace GUID for custom profiles & fragments
terminalNamespaceGUID = uuid.UUID("{f65ddb7e-706b-4499-8a50-40313caf510a}")

# The Application Namespace GUID
```

```
appNamespaceGUID = uuid.uuid5(terminalNamespaceGUID, "Git".encode("UTF-
16LE").decode("ASCII"))

# Calculate the example GUID for the 'Git Bash' profile
profileGUID = uuid.uuid5(appNamespaceGUID, "Git Bash".encode("UTF-
16LE").decode("ASCII"))

# Output the GUID as Windows Terminal expects it (enclosed in curly brackets)
print(f"{{{profileGUID}}}")
```

Calculating a GUID for a built in profile

To calculate a GUID for a built in profile, such as the automatically generated WSL profiles, you can use the following Python 3 example. It calculates a GUID based on the Windows Terminal namespace GUID for a profile called 'Ubuntu' that was automatically generated for the WSL distribution, conveniently matching the sanity check.

Python

```
import uuid

# The Windows Terminal namespace GUID automatically generated profiles
terminalNamespaceGUID = uuid.UUID("{2bde4a90-d05f-401c-9492-e40884ead1d8}")

# Calculate the example GUID for the 'Git Bash' profile
profileGUID = uuid.uuid5(terminalNamespaceGUID, "Ubuntu".encode("UTF-
16LE").decode("ASCII"))

# Output the GUID as Windows Terminal expects it (enclosed in curly brackets)
print(f"{{{profileGUID}}}")
```

Minimum requirements for settings added with fragments

There are some minimal restrictions on what can be added to user settings using JSON fragments:

- For new profiles added via fragments, the new profile must, at a minimum, define a name for itself.
- For new color schemes added via fragments, the new color scheme must define a name for itself, as well as define every color in the color table (i.e. the colors "black" through "brightWhite" in the example image above).

Where to place the JSON fragment files

The location to place the JSON fragment files varies depending on the installation method of the application that wishes to place them.

Microsoft Store applications

For applications installed through the Microsoft Store (or similar), the application must declare itself to be an app extension. Learn more about how to Create and host an app extension. The necessary section is replicated here. The appxmanifest file of the package must include:

XML

```
<Package
  ...
  xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
  IgnorableNamespaces="uap uap3 mp">
  ...
    <Applications>
      <Application Id="App" ... >
        ...
        <Extensions>
          ...
          <uap3:Extension Category="windows.appExtension">
            <uap3:AppExtension Name="com.microsoft.windows.terminal.settings"
                               Id="<id>"
                               PublicFolder="Public">
            </uap3:AppExtension>
          </uap3:Extension>
        </Extensions>
      </Application>
    </Applications>
    ...
</Package>
```

Key things to note:

- The `"Name"` field must be `com.microsoft.windows.terminal.settings` for Windows Terminal to be able to detect the extension.
- The `"Id"` field can be filled out as the developer wishes.
- The `"PublicFolder"` field should have the name of the folder, relative to the package root, where the JSON files are stored (this folder is typically called "Public" but can be named something else if the developer wishes).
- Inside the public folder, a subdirectory called "Fragments" should be created, and the JSON files should be stored in that subdirectory.

Applications installed from the web

For applications installed from the web, there are 2 cases.

The first is that the installation is for all the users on the system. In this case, the JSON files should be added to the folder:

```
C:\ProgramData\Microsoft\Windows Terminal\Fragments\{app-name}\{file-name}.json
```

In the second case, the installation is only for the current user. In this case, the JSON files should be added to the folder:

```
C:\Users\<user>\AppData\Local\Microsoft\Windows Terminal\Fragments\{app-name}\{file-name}.json
```

Note that both the `ProgramData` and `LocalAppData` folders are known folders that the installer should be able to access. If in either case, if the `Windows Terminal\Fragments` directory does not exist, the installer should create it. The `{app-name}` should be unique to your application and the `{file-name}.json` can be anything - the terminal will read all .json files in that directory.