What is Windows Terminal?

Article • 02/02/2024

Windows Terminal is a modern host application for the command-line shells you already love, like Command Prompt, PowerShell, and bash (via Windows Subsystem for Linux (WSL)). Its main features include multiple tabs, panes, Unicode and UTF-8 character support, a GPU accelerated text rendering engine, and the ability to create your own themes and customize text, colors, backgrounds, and shortcuts.

Install Windows Terminal

https://www.microsoft.com/en-us/videoplayer/embed/RWHAdS?postJsllMsg=true

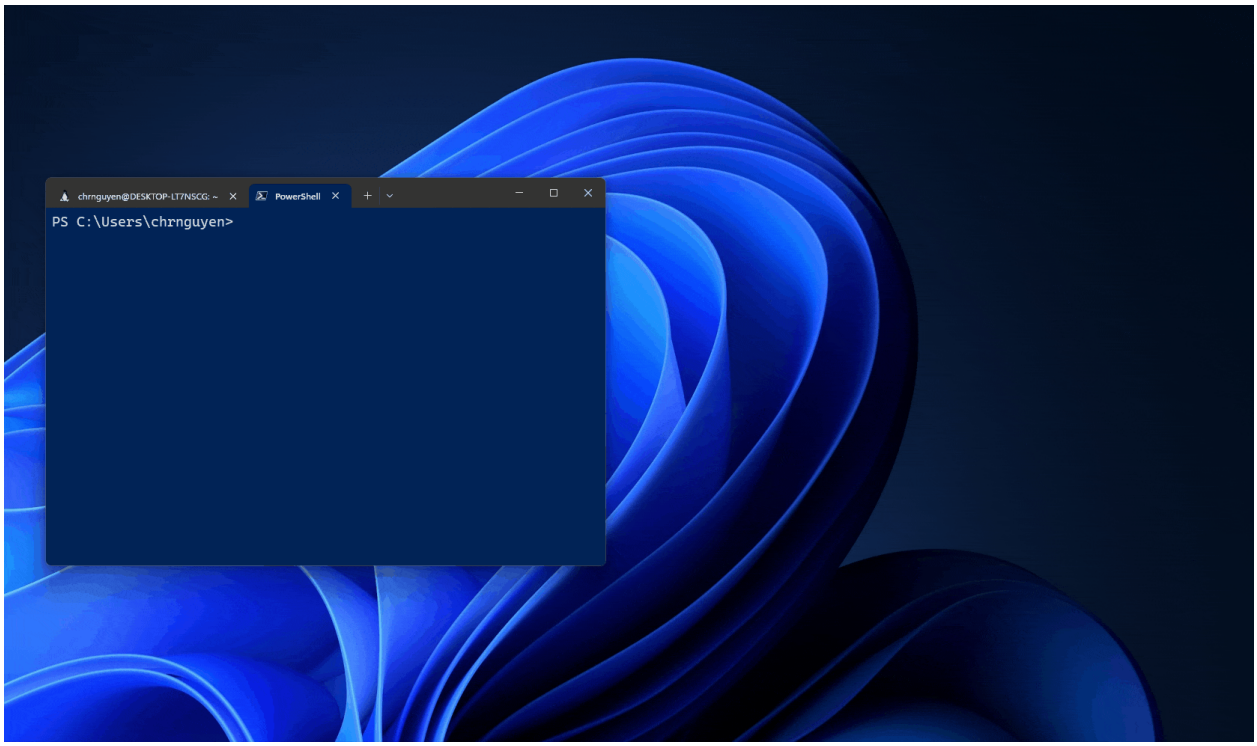
> **Note**
>
> For more general info, check out Scott Hanselman's article: **What's the difference between a console, a terminal, and a shell?** or Rich Turner's video **What is a command-line shell?**.

Multiple profiles supporting a variety of command line applications

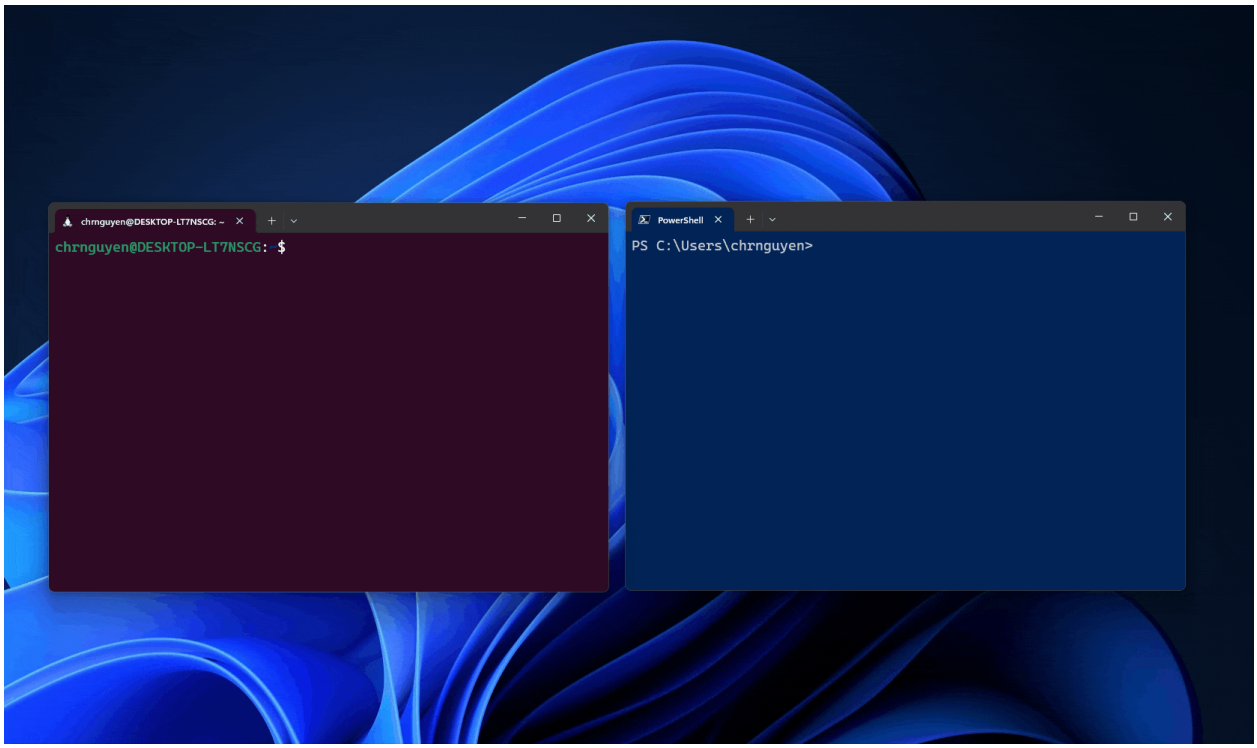
Any application that has a command line interface can be run inside Windows Terminal. This includes everything from PowerShell and Command Prompt to Azure Cloud Shell and any WSL distribution such as Ubuntu or Oh-My-Zsh.

Tab tearout (Preview)

You can tear out tabs in Windows Terminal and create new windows.



You can also drag and drop tabs into existing windows.



Customized schemes and configurations

You can configure your Windows Terminal to have a variety of color schemes and settings. To learn how to customize your prompt with cool themes, see Tutorial: Set up a custom prompt for PowerShell or WSL with Oh My Posh To learn how to make your own color scheme, visit the Color schemes page.

Custom actions

There are a variety of custom commands you can use in Windows Terminal to have it feel more natural to you. If you don't like a particular keyboard shortcut, you can change it to whatever you prefer.

For example, the default shortcut to copy text from the command line is `Ctrl+Shift+C`. You can change this to `Ctrl+1` or whatever you prefer. To open a new tab, the default shortcut is `Ctrl+Shift+T`, but maybe you want to change this to `Ctrl+2`. The default shortcut to flip between the tabs you have open is `Ctrl+Tab`, this could be changed to `Ctrl+-` and used to create a new tab instead.

You can learn about customizing shortcuts on the Actions page.

Unicode and UTF-8 character support

Windows Terminal can display Unicode and UTF-8 characters such as emoji and characters from a variety of languages.

GPU accelerated text rendering

Windows Terminal uses the GPU to render its text, thus providing improved performance over the default Windows command line experience.

Background image support

You can have background images and gifs inside your Windows Terminal window. Information on how to add background images to your profile can be found on the Profile - Appearance page.

Command line arguments

You can set Windows Terminal to launch in a specific configuration using command line arguments. You can specify which profile to open in a new tab, which folder directory should be selected, open the terminal with split window panes, and choose which tab should be in focus.

For example, to open Windows Terminal from PowerShell with three panes, with the left pane running a Command Prompt profile and the right pane split between your PowerShell and your default profile running WSL, enter:

PowerShell

```
wt -p "Command Prompt" `; split-pane -p "Windows PowerShell" `; split-pane -
H wsl.exe
```

Learn how to set up command-line arguments on the Command line arguments page.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Install and get started setting up Windows Terminal

Article • 10/06/2022

Install

Install Windows Terminal

To try the latest preview features, you may also want to install Windows Terminal Preview.

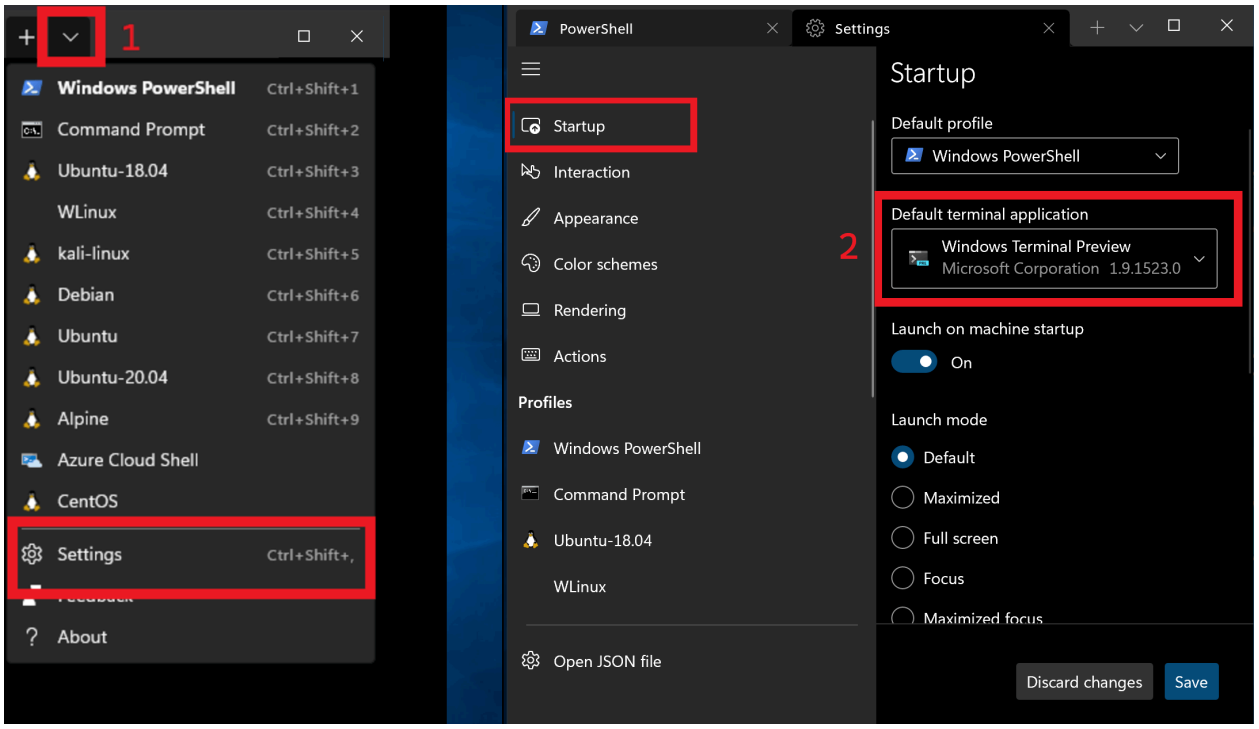
> **Note**
>
> If you don't have access to the Microsoft Store, the builds are published on the GitHub releases page. If you install from GitHub, Windows Terminal will not automatically update with new versions. For additional installation options using a package manager (winget, chocolatey, scoop), see the Windows Terminal product repo.

Set your default terminal application

> **Note**
>
> This feature is available in all versions of Windows 11 and versions of Windows 10 22H2 after the installation of the May 23, 2023 update, KB5026435.

To open any command line application with Windows Terminal, set it as your default terminal application.

1. Open Windows Terminal and go to the **Settings** UI window.
2. Select **Startup** and choose "Windows Terminal" as the **Default terminal application** setting.



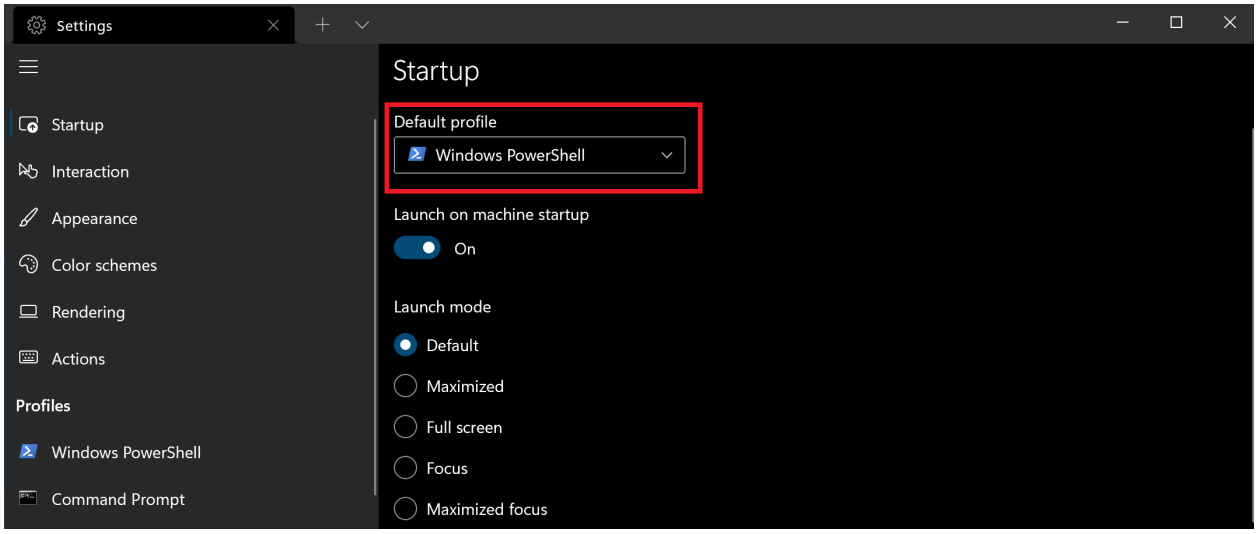
Set your default terminal profile

After installation, when you open Windows Terminal, it will start with the PowerShell command line as the default profile in the open tab.

To change the default profile:

1. Open Windows Terminal and go to the **Settings** UI window.
2. Select **Startup** and choose the **Default profile** that you prefer.

You can also set your default profile in the Settings.json file associated with Windows Terminal if you prefer.

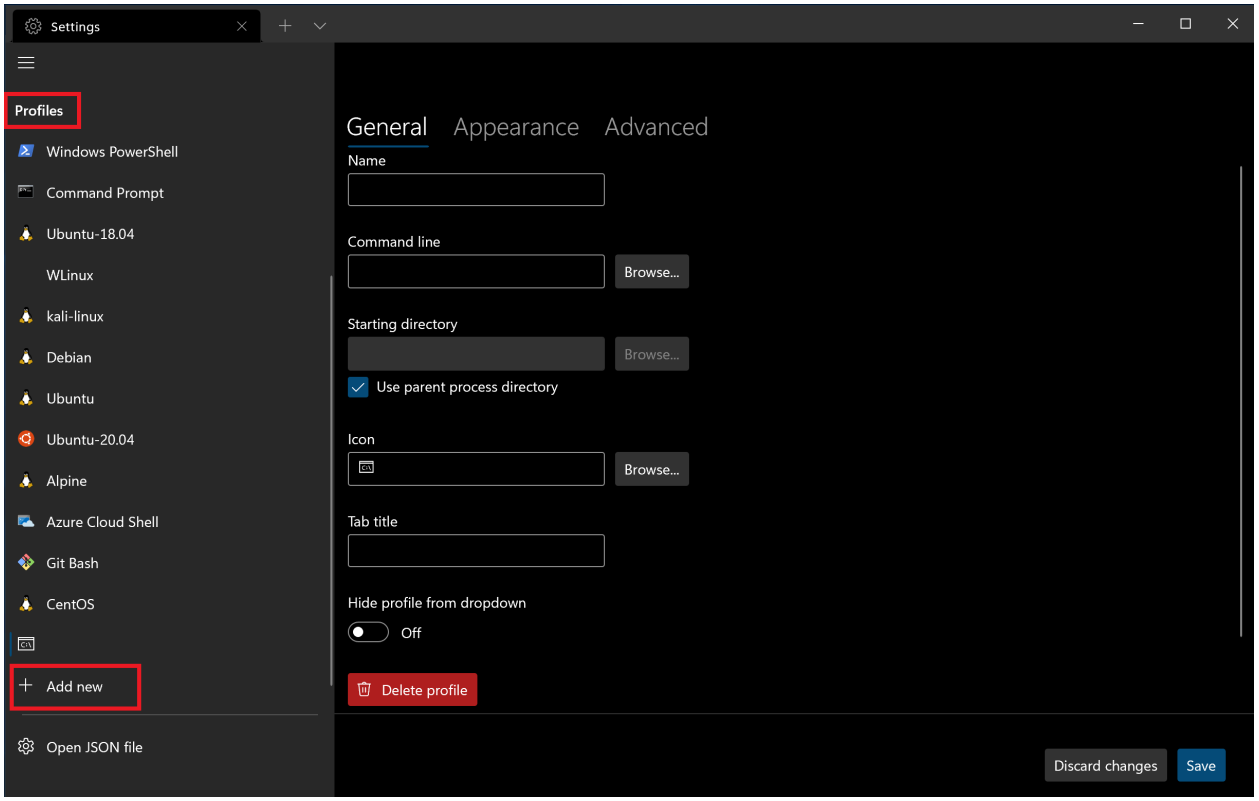


Add new profiles

Windows Terminal will automatically create profiles for you if you have WSL distributions or multiple versions of PowerShell installed.

Your command line profiles will be listed in the Settings UI, in addition to the option to + **Add new** profiles.

Learn more about dynamic profiles on the Dynamic profiles page.

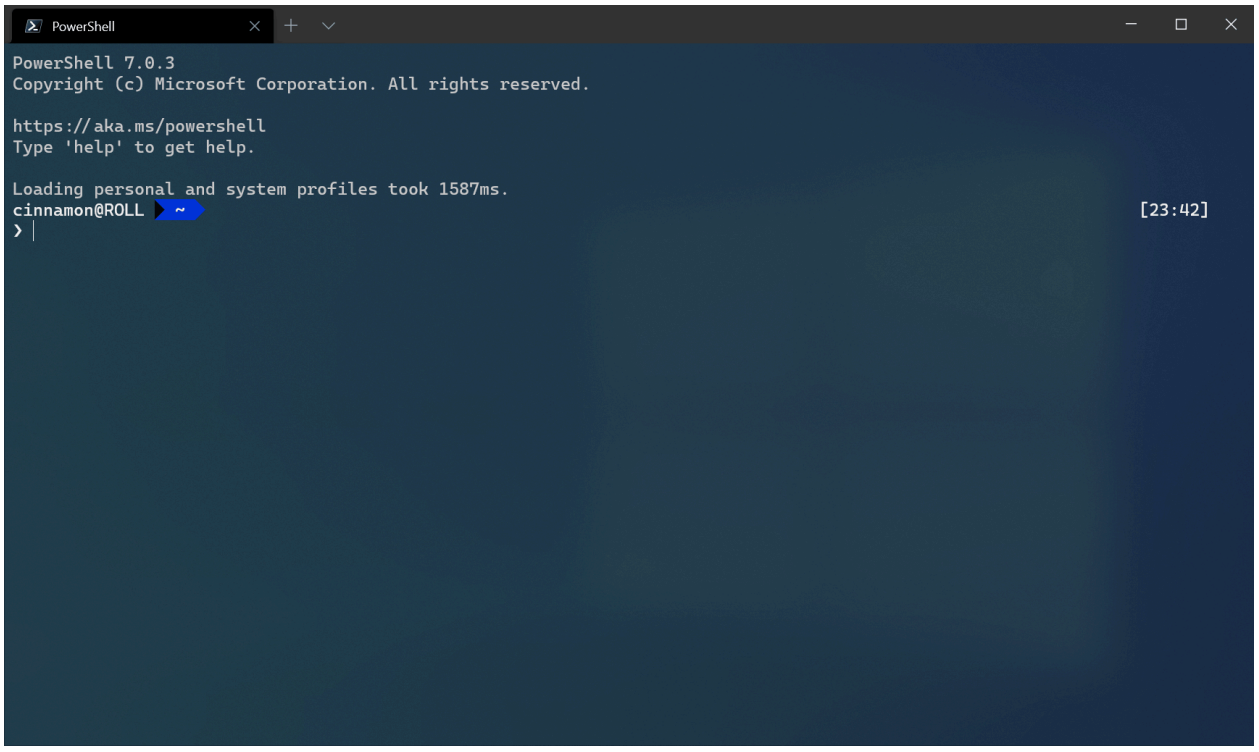


Open a new tab

You can open a new tab of the default profile by pressing `Ctrl+Shift+T` or by selecting the + (plus) button. To open a different profile, select the ∨ (arrow) next to the + button to open the dropdown menu. From there, you can select which profile to open.

Invoke the command palette

You can invoke most features of Windows Terminal through the command palette. The default key combination to invoke it is `Ctrl+Shift+P`. You can also open it using the **Command palette** button in the dropdown menu.



Open a new pane

You can run multiple shells side-by-side using panes. To open a pane, you can use `Alt+Shift++` for a vertical pane or `Alt+Shift+-` for a horizontal one. You can also use `Alt+Shift+D` to open a duplicate pane of your focused profile. Learn more about panes on the Panes page.

Configuration

To customize the settings of your Windows Terminal, select **Settings** in the dropdown menu. This will open the settings UI to configure your settings. You can learn how to open the settings UI with keyboard shortcuts on the Actions page.

Settings JSON file

If you prefer to configure your Windows Terminal settings using code, rather than the graphic user interface, you can edit the settings.json file.

Select **Settings** in the Windows Terminal dropdown menu while holding `Shift` to open the `settings.json` file in your default text editor. (The default text editor is defined in your Windows settings.)

The path for your Windows Terminal settings.json file may be found in one of the following directories:

- Terminal (stable / general release): `%LOCALAPPDATA%\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\LocalState\settings.json`
- Terminal (preview release): `%LOCALAPPDATA%\Packages\Microsoft.WindowsTerminalPreview_8wekyb3d8bbwe\LocalState\settings.json`
- Terminal (unpackaged: Scoop, Chocolatey, etc): `%LOCALAPPDATA%\Microsoft\Windows Terminal\settings.json`

> **Tip**
>
> 1. You can access the default settings for Windows Terminal by selecting **Settings** in the dropdown menu while holding `Alt` to open the `defaults.json` file in your default text editor. This file is auto-generated and any changes to it will be ignored.
> 2. It is possible to create a **JSON fragment extension** in order to store profile data and color schemes in a separate file, which can be useful to prevent excessively large configuration files.

Command line arguments

You can launch the terminal in a specific configuration using command line arguments. These arguments let you open the terminal with specific tabs and panes with custom profile settings. Learn more about command line arguments on the Command line arguments page.

Troubleshooting

If you encounter any difficulties using the terminal, reference the Troubleshooting page. If you find any bugs or have a feature request, you can select the feedback link in the **About** menu of the terminal to go to the GitHub page where you can file a new issue.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Windows Terminal Distribution Types

Article • 05/23/2023

Windows Terminal is distributed via GitHub releases in a variety of formats:

- Packaged, or "MSIX bundle"
  - This is the oldest and best-supported distribution of Windows Terminal.
  - The packaged distribution can be installed via the `.msixbundle` file provided on the GitHub releases page or through the Microsoft Store (Stable, Preview).
  - Installation via MSIX bundle may require network connectivity to download dependency packages from the Store.
  - When installed via MSIX bundle, Terminal will receive automatic updates through the Store.
- Preinstallation Kit
  - A preinstallation kit is available for system integrators and OEMs interested in preinstalling Windows Terminal on a Windows image.
  - More information is available in the DISM documentation on preinstallation. Users who do not intend to preinstall Windows Terminal should continue using the Packaged distribution.
  - When installed via preinstallation kit, Terminal will receive automatic updates through the Store.
- Unpackaged, or "ZIP" (new in 1.17 stable)
  - This distribution method was not officially supported until stable channel version 1.17.
  - The unpackaged distribution does not receive automatic updates, which puts you in control of exactly when new versions are installed.
- Portable
  - A variant of the unpackaged distribution, where Terminal stores its settings in a nearby directory.
  - Learn more about configuring Portable mode.

Distribution feature comparison

	Packaged	Preinstallation Kit	Unpackaged	Portable
Automatic updates	✅	✅	❌	❌

	Packaged	Preinstallation Kit	Unpackaged	Portable
Automatic architecture selection	✅	✅	❌	❌
Can be set as your default terminal	✅	✅	❌	❌
"Open in Terminal" context menu	✅	✅	❌	❌
Automatic start on login option	✅	✅	*manual*	*manual*
Double-click installation	✅	❌	❌	❌
Installation on non-networked machines	❌	✅	✅	✅
Preinstallation in a Windows image	❌	✅	*as plain files*	*as plain files*
User-controlled installation path	❌	❌	✅	✅
Double-click activatable	❌	❌	✅	✅
Settings storage location	User folder, per package	(same as packaged)	`%LOCALAPPDATA%`	Next to `WindowsTerminal.exe`

Windows Terminal Portable

As of stable channel version 1.17, Windows Terminal supports being deployed in "Portable mode". Portable mode ensures that all data created and maintained by Windows Terminal is saved next to the application so that it can be more easily moved across different environments.

Portable mode is supported by the unpackaged "ZIP" distribution.

This is an officially-supported mode of execution where Windows Terminal stores its settings in a `settings` folder next to `WindowsTerminal.exe`.

Portable mode is not supported in the packaged or preinstallation kit distributions of Windows Terminal.

Why use Portable mode?

The unpackaged and portable mode distributions of Windows Terminal allow you to use Terminal without installing it globally, e.g. on systems where you may not have permission to install MSIX packages or download software from the Microsoft Store.

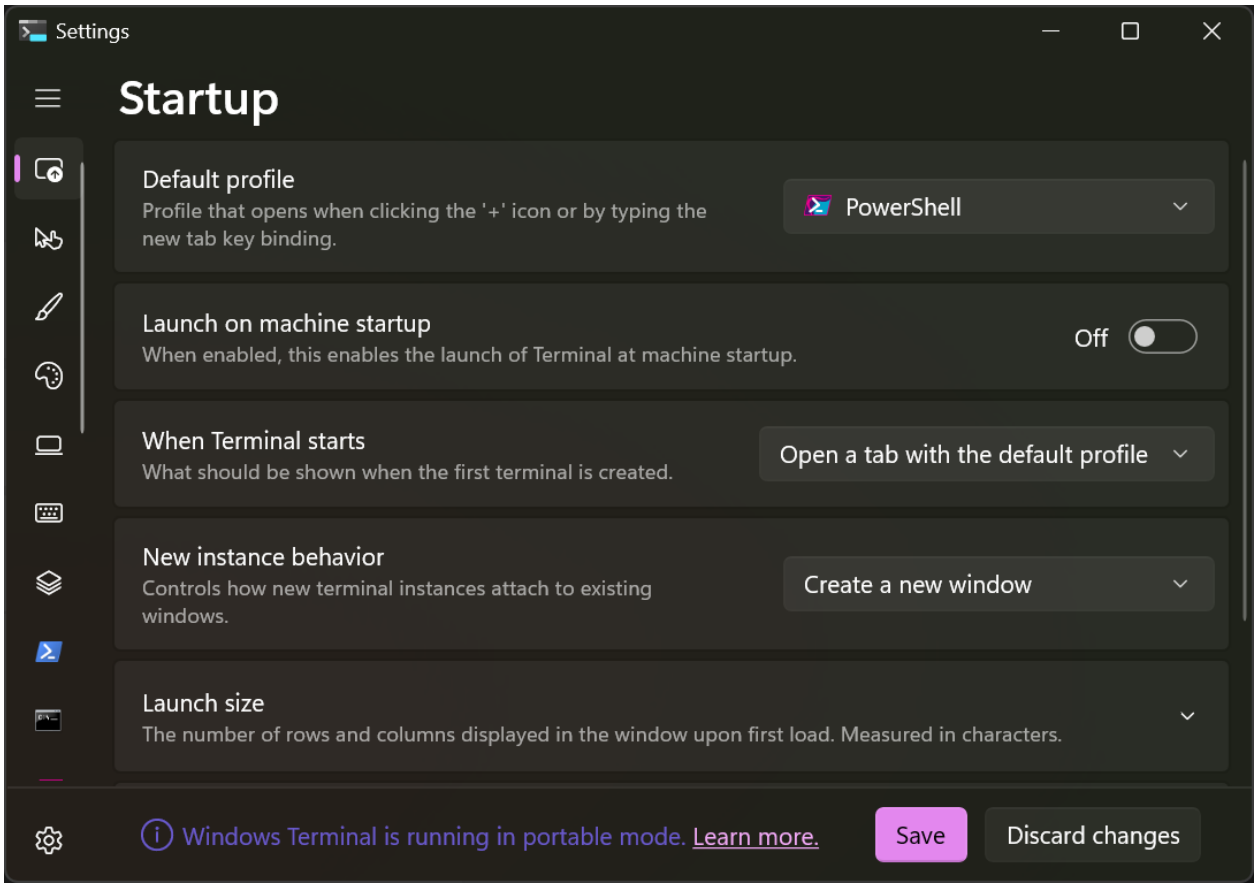
Portable mode allows you to carry around or archive a preconfigured installation of Windows Terminal and run it from a network share, cloud drive or USB flash drive. Any such installation is self-contained and will not interfere with other installed distributions of Windows Terminal.

Enabling Portable mode

Portable mode needs to be enabled manually. After unzipping the Windows Terminal download, create a file named `.portable` next to `WindowsTerminal.exe`.

> ⓘ **Note**
>
> Windows Terminal will not automatically reload its settings when you create the portable mode marker file. This change will only apply after you relaunch Terminal.

Windows Terminal will automatically create a directory named `settings` in which it will store both settings and runtime state such as window layouts.



Disabling Portable mode

You can restore Portable mode unpackaged installation to its original configuration, where settings are stored in `%LOCALAPPDATA%\Microsoft\Windows Terminal`, by removing the `.portable` marker file from the directory containing `WindowsTerminal.exe`.

If you wish to reenable portable mode, you can create a new `.portable` marker file next to `WindowsTerminal.exe`.

Upgrading a Portable mode Install

You can upgrade a portable mode installation of Windows Terminal by moving the `.portable` marker file and the `settings` directory to a newly-extracted unpackaged version of Windows Terminal.

Startup settings in Windows Terminal

Article • 11/03/2022

The properties listed below affect the entire terminal window, regardless of the profile settings. These should be placed at the root of your settings.json file.

Default profile

Set the default profile that opens by typing `Ctrl+Shift+T`, typing the key binding assigned to `newTab`, running `wt new-tab` without specifying a profile, or clicking the '+' icon.

Property name: `defaultProfile`

Necessity: Required

Accepts: GUID or profile name as a string

Default value: PowerShell's GUID

Default terminal application

Set the default terminal emulator in Windows for all command line applications to run inside of.

Property name: This modifies an OS setting and does not have a property name inside the settings.json file.

Necessity: Required

Accepts: Any terminal emulator that appears in the dropdown

Default value: Windows Console Host

> **ⓘ Important**
>
> This feature is only available when running Windows 10 Insider Program Dev Channel or Windows 11.

Launch on machine startup

When set to `true`, this enables the launch of Windows Terminal at startup. Setting this to `false` will disable the startup task entry.

Note: if the Windows Terminal startup task entry is disabled either by org policy or by user action this setting will have no effect.

Property name: `startOnUserLogin`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Behavior when starting a new terminal session

When set to `"defaultProfile"`, Windows Terminal will start a new session by opening a single tab with your default profile.

When set to `"persistedWindowLayout"`, this enables Windows Terminal to save the layout of open windows on close and restore all saved windows on starting a new session. Windows Terminal will save the layout of all open windows automatically to assist with restoration from crashes and will also save the layout when using the `quit` action. Additionally, closing the last open window by clicking the `X` button or using the `closeWindow` command will save the layout of that last window.

Note: Currently, Windows Terminal will save the following information:

- Each window's position, size, and name
- The tab layout of each window, including the layout and profile of each pane, but not any contents of those panes
- If your shell is configured to report the current working directory that will be saved as well

Property name: `firstWindowPreference`

Necessity: Optional

Accepts: `"defaultProfile"`, `"persistedWindowLayout"`

Default value: `"defaultProfile"`

Launch mode

This defines whether the terminal will launch as maximized, full screen, or in a window. Setting this to `focus` is equivalent to launching the terminal in the `default` mode, but with focus mode enabled. Similarly, setting this to `maximizedFocus` will result in launching the terminal in a maximized window with focus mode enabled.

Property name: `launchMode`

Necessity: Optional

Accepts: `"default"`, `"maximized"`, `"fullscreen"`, `"focus"`, `"maximizedFocus"`

Default value: `"default"`

New instance behavior

This setting controls how new terminal instances attach to existing windows. This property is only used if the `--window,-w window` command line argument is not provided. This setting accepts the following possible values:

- `useNew`: Create a new window, always. This is how the terminal always behaved prior to version 1.7.
- `useExisting`: Create new tabs in the most recently used window on this desktop. If there's not an existing window on this virtual desktop, then create a new terminal window.
- `useAnyExisting`: Create new tabs in the most recently used window, regardless of which virtual desktop the window is on.

Property name: `windowingBehavior`

Necessity: Optional

Accepts: `"useNew"`, `"useExisting"`, `"useAnyExisting"`

Default value: `"useNew"`

Launch size

Columns on first launch

This is the number of character columns displayed in the window upon first load. If `launchMode` is set to `"maximized"` or `"maximizedFocus"`, this property is ignored.

Property name: `initialCols`

Necessity: Optional

Accepts: Integer

Default value: `120`

Rows on first launch

This is the number of rows displayed in the window upon first load. If `launchMode` is set to `"maximized"` or `"maximizedFocus"`, this property is ignored.

Property name: `initialRows`

Necessity: Optional

Accepts: Integer

Default value: `30`

Launch position

This sets the pixel position of the top left corner of the window upon first load. On a system with multiple displays, these coordinates are relative to the top left of the primary display. If an X or Y coordinate is not provided, the terminal will use the system default for that value. If `launchMode` is set to `"maximized"` or `"maximizedFocus"`, the window will be maximized on the monitor specified by those coordinates.

Property name: `initialPosition`

Necessity: Optional

Accepts: Coordinates as a string in the following formats: `","`, `"#,#"`, `"#,"`, `",#"`

Default value: `","`

Center on launch

When set to `true`, the terminal window will auto-center itself on the display it opens on. The terminal will use the `"initialPosition"` to determine which display to open on.

This interacts with the other launch settings in the following ways:

- `"initialPosition": "x,y"`, `"centerOnLaunch": true`, `"launchMode": "default"`: center on the monitor that `x,y` is on.
- `"initialPosition": "x,y"`, `"centerOnLaunch": true`, `"launchMode": "maximized"`: maximized on the monitor that `x,y` is on (`centerOnLaunch` adds nothing).
- `"initialPosition": <omitted>`, `"centerOnLaunch": true`, `"launchMode": "default"`: center on the default monitor.
- `"initialPosition": <omitted>`, `"centerOnLaunch": true`, `"launchMode": "focus"`: center and enter focus mode on the default monitor.
- `"initialPosition": <omitted>`, `"centerOnLaunch": true`, `"launchMode": "maximized"`: maximized on the default monitor (`centerOnLaunch` adds nothing).

Property name: `centerOnLaunch`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Disable dynamic profiles

This sets which dynamic profile generators are disabled, preventing them from adding their profiles to the list of profiles on startup. For information on dynamic profiles, visit the Dynamic profiles page.

Property name: `disabledProfileSources`

Necessity: Optional

Accepts: `"Windows.Terminal.Wsl"`, `"Windows.Terminal.Azure"`, `"Windows.Terminal.PowershellCore"`, and/or `"Windows.Terminal.SSH"` inside an array

Default value: `[]`

Startup actions

This sets the list of actions to execute on startup, allowing the terminal to launch with a custom set of tabs and panes by default. These actions will be applied only if no command line arguments were supplied. The list of actions is represented by a string with the same format as commands in the command line arguments. For more information about the commands format, visit the Command line arguments page.

Property name: `startupActions`

Necessity: Optional

Accepts: String representing a list of commands to run

Default value: `""`

Continue running in the background (Preview)

When set to `true`, this enables the terminal to continue running in the background after the last window is closed. This allows globalSummon and quake mode to work even when no windows are open. This setting is only available in Preview builds of the Terminal.

Property name: `compatibility.allowHeadless`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Interaction settings in Windows Terminal

Article • 05/29/2024

The properties listed below affect the entire terminal window, regardless of the profile settings. These should be placed at the root of your settings.json file.

Enable unfocused acrylic

Controls if unfocused acrylic is possible. When this is set to `true`, unfocused windows can have acrylic instead of opaque. When set to `false` unfocused windows cannot have acrylic, when the focused window has acrylic the unfocused window will be Solid.

Property name: `compatibility.enableUnfocusedAcrylic`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Automatically copy selection to clipboard

When this is set to `true`, a selection is immediately copied to your clipboard upon creation. The right-click on your mouse will always paste in this case. When it's set to

`false`, the selection persists and awaits further action. Using your mouse to right-click will copy the selection.

Property name: `copyOnSelect`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Text format when copying

When this is set to `true`, the color and font formatting of the selected text is also copied to your clipboard. When it's set to `false`, only plain text is copied to your clipboard. You can also specify which formats you would like to copy.

Property name: `copyFormatting`

Necessity: Optional

Accepts: `true`, `false`, `"all"`, `"none"`, `"html"`, `"rtf"`

Default value: `false`

Remove trailing white-space in rectangular selection

When this is set to `true` and you copy text in a rectangular (block) selection to the clipboard, trailing white-spaces are removed from each line. When it's set to `false`, the white-spaces are preserved, ensuring that all lines have the same length. To copy text in a rectangular (block) selection, hold down the `Alt` key, click and drag your mouse over the text area you want to select. This can be useful for selecting text columns, etc.

Property name: `trimBlockSelection`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Trim trailing whitespace on paste

When enabled, the terminal will automatically trim trailing whitespace characters when pasting text to the terminal.

Property name: `trimPaste`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Word delimiters

This determines the word delimiters used in a double-click selection. Word delimiters are characters that specify where the boundary is between two words. The most common examples are spaces, semicolons, commas, and periods.

Property name: `wordDelimiters`

Necessity: Optional

Accepts: Characters as a string

Default value: `/\\()\"'-:,.;<>~!@#$%^&*|+=[]{}?|`
(`|` is `U+2502 BOX DRAWINGS LIGHT VERTICAL`)

> **ⓘ Important**
>
> The following characters must be escaped with a backslash : `\`, `"`

As an example, here are some sets of delimiters you can use to match the behavior of various other terminal emulators:

Expand table

Terminal	Delimiters
Xterm	`"$'()*;<>[\]^{|}`

Terminal	Delimiters
Gnome Terminal	`!"$'()*:;<>[]^{|}`
Konsole	`!"$'()*,;<>[\]^{|}`
Rxvt	`&();<>|`
Alacritty	`"'(),:<>[]{|}`
Kitty	`!"$'()*,:;<>[\]^{|}`

Snap window resizing to character grid

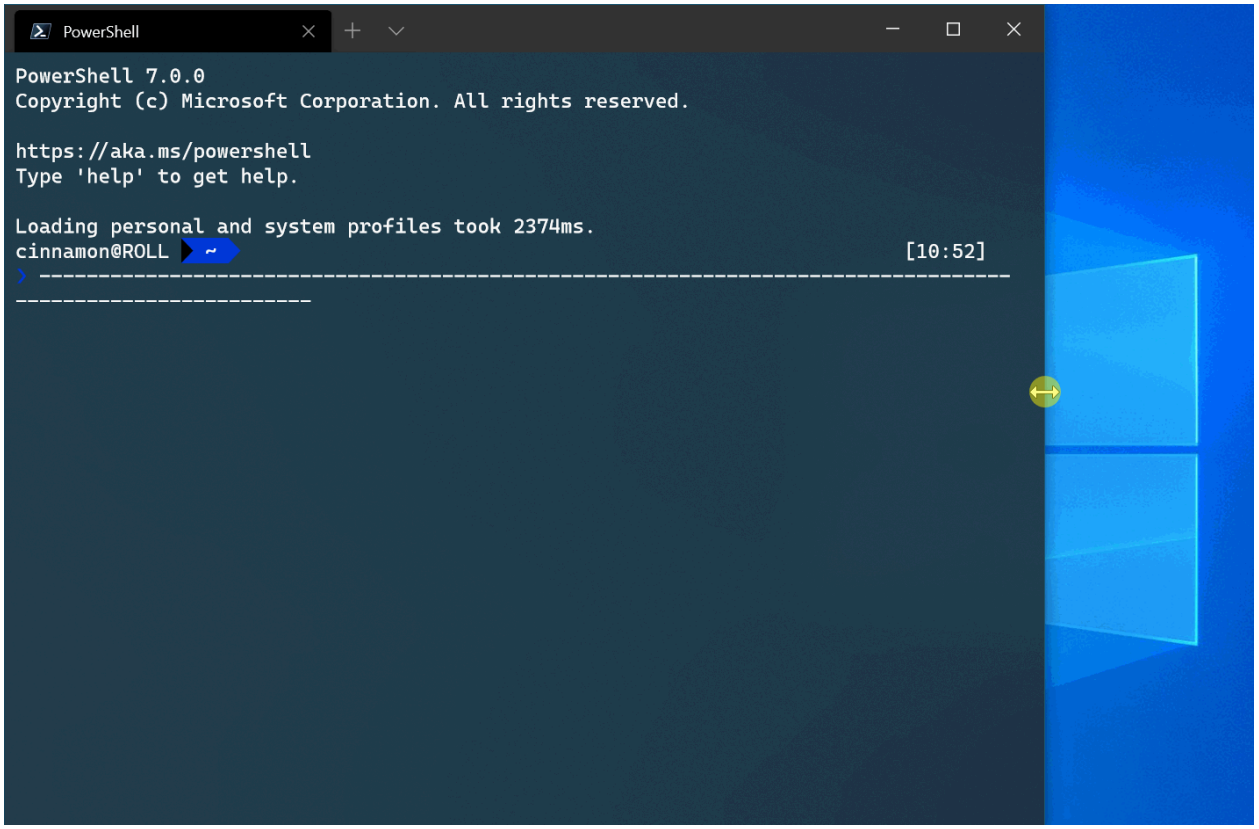
When this is set to `true`, the window will snap to the nearest character boundary on resize. When it's set to `false`, the window will resize "smoothly".

Property name: `snapToGridOnResize`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`



Minimize to notification area

When this is set to `true`, minimizing a window will hide it from the taskbar, making it inaccessible from that area. It will instead be accessible from terminal's notification area icon. If either this global setting or the `minimizeToNotificationArea` global setting is set to true, terminal will place an icon in the notification area.

Property name: `minimizeToNotificationArea`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

> ⓘ **Important**
>
> This setting was renamed from `"minimizeToTray"` to `"minimizeToNotificationArea"`.

Always show notification icon

When this is set to `true`, the terminal will place its icon in the notification area. If either this global setting or the `minimizeToNotificationArea` global setting is set to true, the terminal will place an icon in the notification area. The user will also be able to utilize the `minimizeToNotificationArea` *action*.

Property name: `alwaysShowNotificationIcon`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

> ⓘ **Important**
>
> This setting was renamed from `"alwaysShowTrayIcon"` to `"alwaysShowNotificationIcon"`.

Tab settings

Tab switcher interface style

When this is set to `true` or `"mru"`, the `nextTab` and `prevTab` commands will use the tab switcher UI, with most recently used ordering. When set to `"inOrder"`, these actions will switch tabs in their current order in the tab bar. The UI will show all the currently open tabs in a vertical list, navigable with the keyboard or mouse.

The tab switcher will open on the initial press of the actions for `nextTab` and `prevTab`, and will stay open as long as a modifier key is held down. When all modifier keys are released, the switcher will close and the highlighted tab will be focused. `Tab`/`Shift+Tab`, the `Up` and `Down` arrow keys, and the `nextTab`/`prevTab` actions can be used to cycle through the switcher UI.

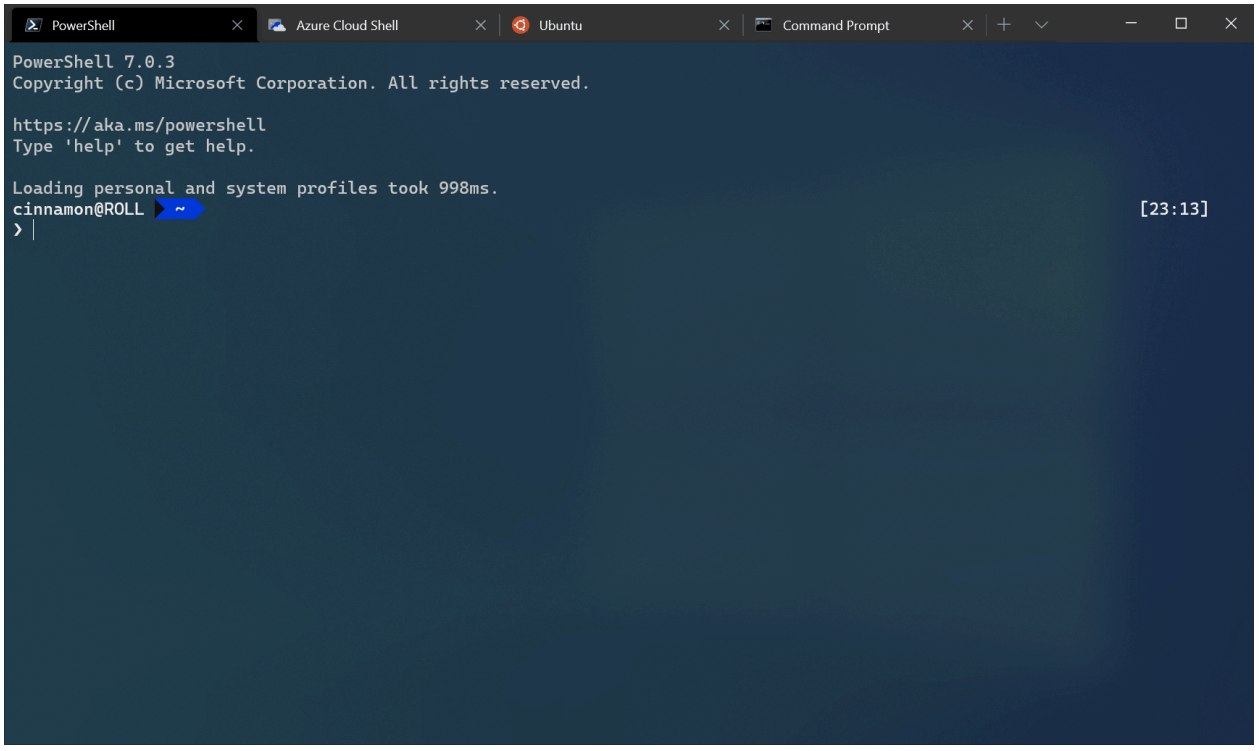
To disable the tab switcher, you can set this to `false` or `"disabled"`.

Property name: `tabSwitcherMode`

Necessity: Optional

Accepts: `true`, `false`, `"mru"`, `"inOrder"`, `"disabled"`

Default value: `"inOrder"`



Enable tab switcher

When this is set to `true`, the `nextTab` and `prevTab` commands will use the tab switcher UI. The UI will show all the currently open tabs in a vertical list, navigable with the keyboard or mouse.

The tab switcher will open on the initial press of the actions for `nextTab` and `prevTab`, and will stay open as long as a modifier key is held down. When all modifier keys are released, the switcher will close and the highlighted tab will be focused. `Tab`/`Shift+Tab`, the `Up` and `Down` arrow keys, and the `nextTab`/`prevTab` actions can be used to cycle through the switcher UI.

Property name: `useTabSwitcher`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

> **⊗ Caution**
>
> The `"useTabSwitcher"` setting is no longer available in versions 1.5 and later. It is recommended that you use the `"tabSwitcherMode"` setting instead.

Automatically hide on focus loss

When enabled, this allows the Terminal window to automatically be hidden as soon as the window loses focus.

Property name: `autoHideWindow`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

—

Automatically focus pane on mouse hover

When this is set to `true`, the terminal will move focus to the pane on mouse hover. When it's set to `false`, a click will be required to focus the pane using the mouse.

Property name: `focusFollowMouse`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Automatically detect URLs and make them clickable

When this is set to `true`, URLs will be detected by the terminal. This will cause URLs to underline on hover and be clickable by pressing `Ctrl`. This is an experimental feature and its continued existence is not guaranteed.

Property name: `experimental.detectURLs`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Paste warnings

Warn when the text to paste is very large

When this is set to `true`, trying to paste text with more than 5 KiB of characters will display a dialog asking you whether to continue or not with the paste. When it's set to `false`, the dialog is not shown and instead the text is pasted right away. If you often right-click on the terminal by accident after having selected a lot of text, this might be useful to prevent the terminal from becoming unresponsive while the program connected to the terminal receives the clipboard's content.

Property name: `largePasteWarning`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Warn when the text to paste contains multiple lines

When this is set to `true`, trying to paste text with multiple lines will display a dialog asking you whether to continue or not with the paste. When it's set to `false`, the dialog is not shown and instead the text is pasted right away. In most shells, one line corresponds to one command so if you paste text that contains the "new line" character into a shell, one or more command(s) might be executed automatically upon paste, without you having time to validate the commands. This can be useful if you often copy and paste commands from untrusted websites.

Property name: `multiLinePasteWarning`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Legacy input encoding

Force the terminal to use the legacy input encoding. Specific keys in some applications may stop working when enabling this setting, but it can be useful for advanced-level scenarios when debugging input issues, especially with the debug tap.

Property name: `experimental.input.forceVT`

Necessity: Optional

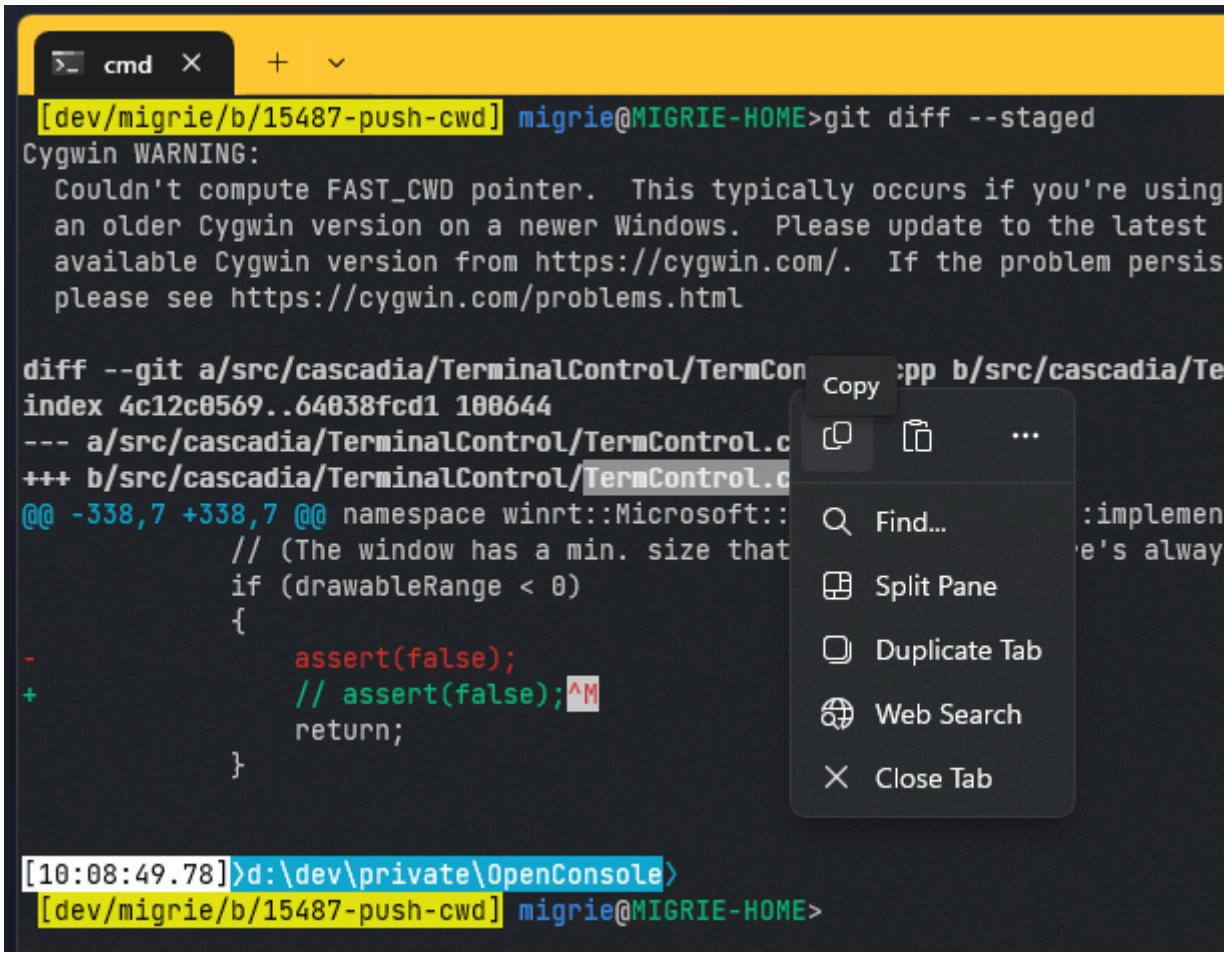
Accepts: `true`, `false`

Default value: `false`

Context Menu

The context menu in the Windows Terminal is an easy way to access common actions quickly. When this is set to `true`, a right-click in the Terminal will activate the context menu. When set to `false`, a right-click will paste.

The context menu can also be opened with the `showContextMenu` action, regardless if this setting is enabled or not.



Property name: `experimental.rightClickContextMenu`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Web searching

This is the default URL used when searching the web from the terminal with the `searchWeb` action (including the right-click context menu). The `%s` in this string is replaced with the selected text. The default value is `https://www.bing.com/search?q=%s`.

Property name: `searchWebDefaultQueryUrl`

Necessity: Optional

Accepts: URL as a string

Default value: `https://www.bing.com/search?q=%s`

> **Important**
>
> This feature is only available in Windows Terminal Preview.

Appearance settings in Windows Terminal

Article • 09/28/2023

The properties listed below affect the entire terminal window, regardless of the profile settings. These should be placed at the root of your settings.json file.

Language

This sets an override for the application's preferred language.

Property name: `language`

Necessity: Optional

Accepts: A BCP-47 language tag like `"en-US"`

Theme

This sets the theme (dark theme or light theme) of the application. `"system"` will use the same theme as Windows.

Property name: `theme`

Necessity: Optional

Accepts: `"system"`, `"dark"`, `"light"`, name of custom theme

Default value: `"system"`

Always show tabs

When this is set to `true`, tabs are always displayed. When it's set to `false` and `showTabsInTitlebar` is set to `false`, tabs are always displayed underneath the title bar. When this is set to `false` and `showTabsInTitlebar` is set to `false`, tabs only appear after more than one tab exists, by typing `Ctrl+Shift+T` or by typing the key binding assigned to `newTab`. Note that changing this setting will require starting a new terminal instance.

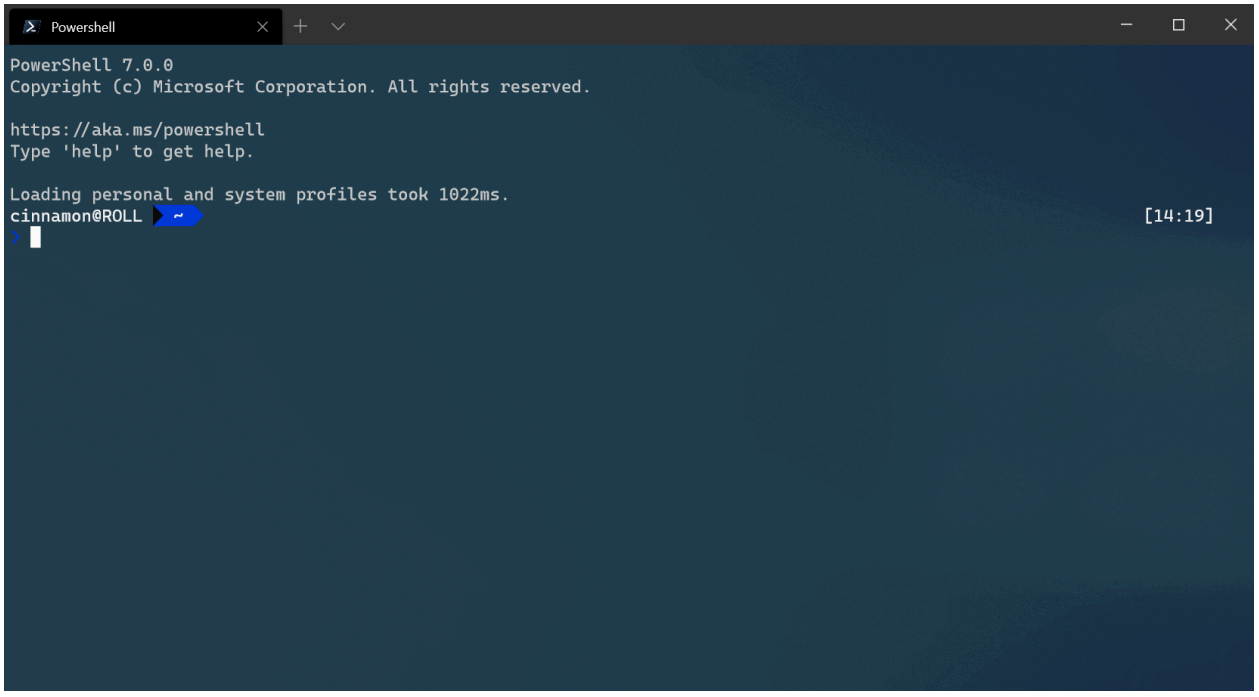
> **Note**
>
> This setting has no effect when `showTabsInTitlebar` is `true`.

Property name: `alwaysShowTabs`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`



Position of newly created tabs (Preview)

Specifies where new tabs appear in the tab row. When this is set to `"afterLastTab"`, new tabs appear at the end of the tab row. When it's set to `"afterCurrentTab"`, new tabs appear after the current tab.

Property name: `newTabPosition`

Necessity: Optional

Accepts: `"afterLastTab"`, `"afterCurrentTab"`

Default value: `"afterLastTab"`

Hide the title bar

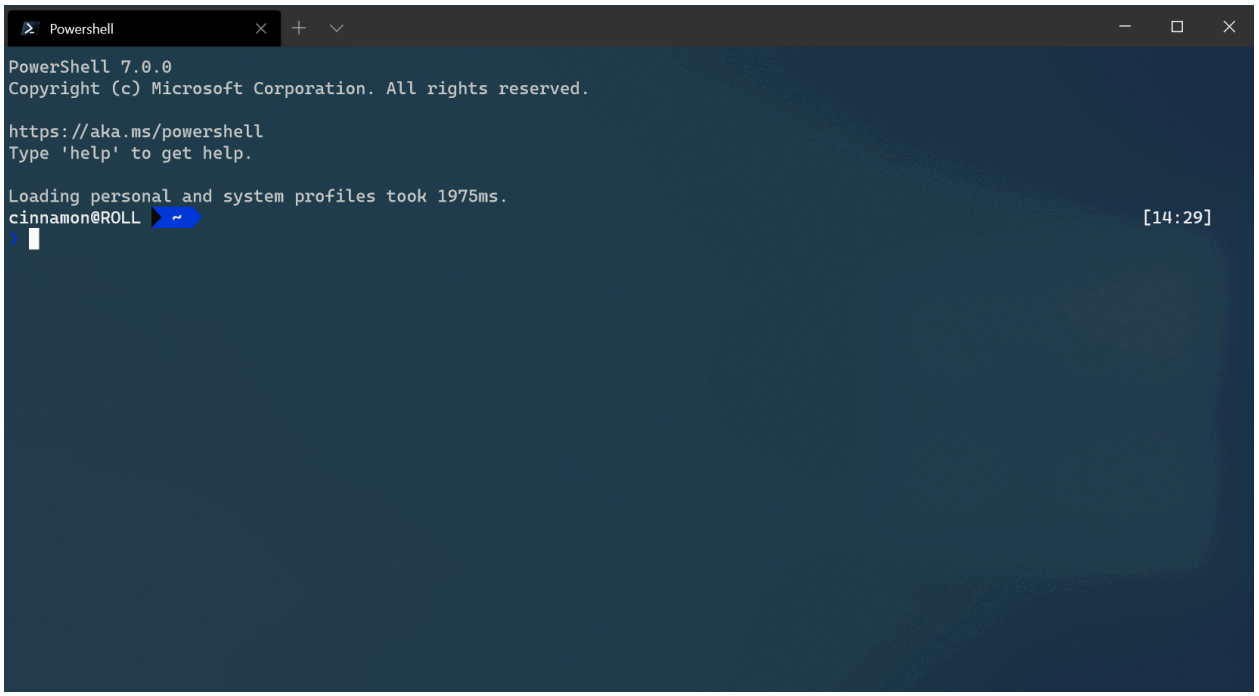
When this is set to `true`, the tabs are moved into the title bar and the title bar disappears. When it's set to `false`, the title bar sits above the tabs. Note that changing this setting will require starting a new terminal instance.

Property name: `showTabsInTitlebar`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`



Show acrylic in tab row

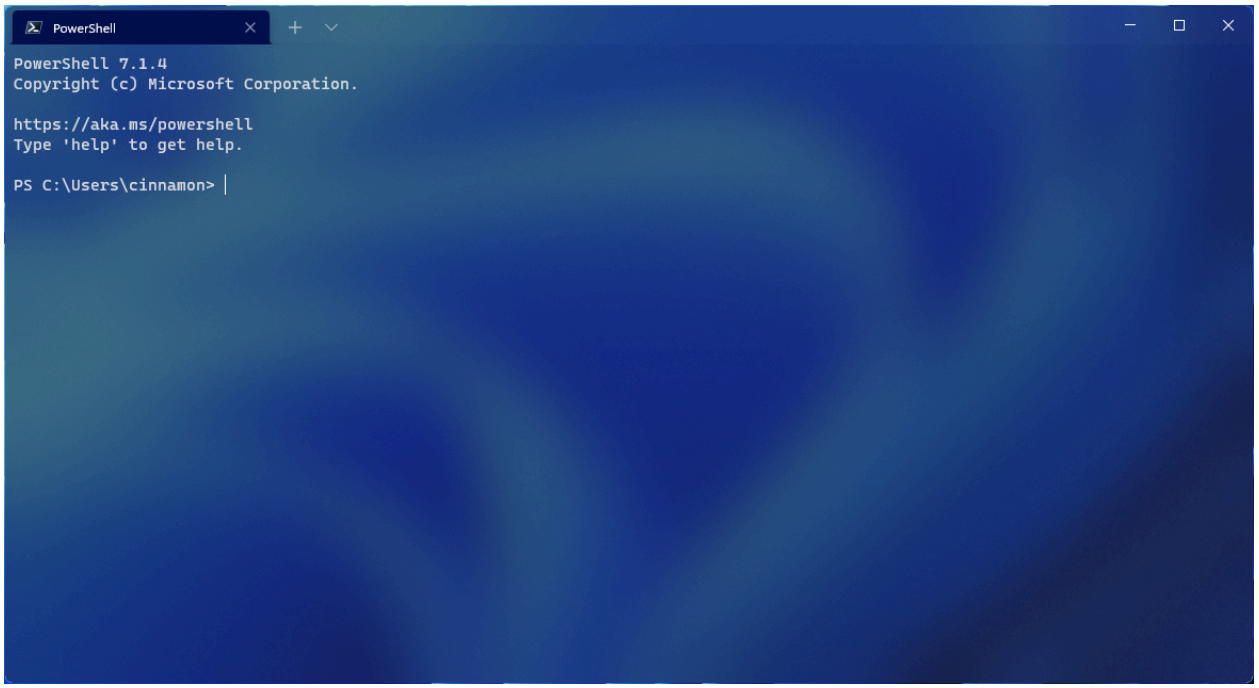
When this is set to `true`, the tab row is given an acrylic background at 50% opacity. When it's set to `false`, the tab row will be opaque. Note that changing this setting will require starting a new terminal instance.

Property name: `useAcrylicInTabRow`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`



Use active terminal title as application title

When this is set to `true`, the title bar displays the title of the selected tab. When it's set to `false`, title bar displays "Windows Terminal". Note that changing this setting will require starting a new terminal instance.

Property name: `showTerminalTitleInTitlebar`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Always on top mode

When set to true, Windows Terminal windows will launch on top of all other windows on the desktop. This state can also be toggled with the `toggleAlwaysOnTop` key binding.

Property name: `alwaysOnTop`

Necessity: Optional

Accepts: `true, false`

Default value: `false`

Tab width mode

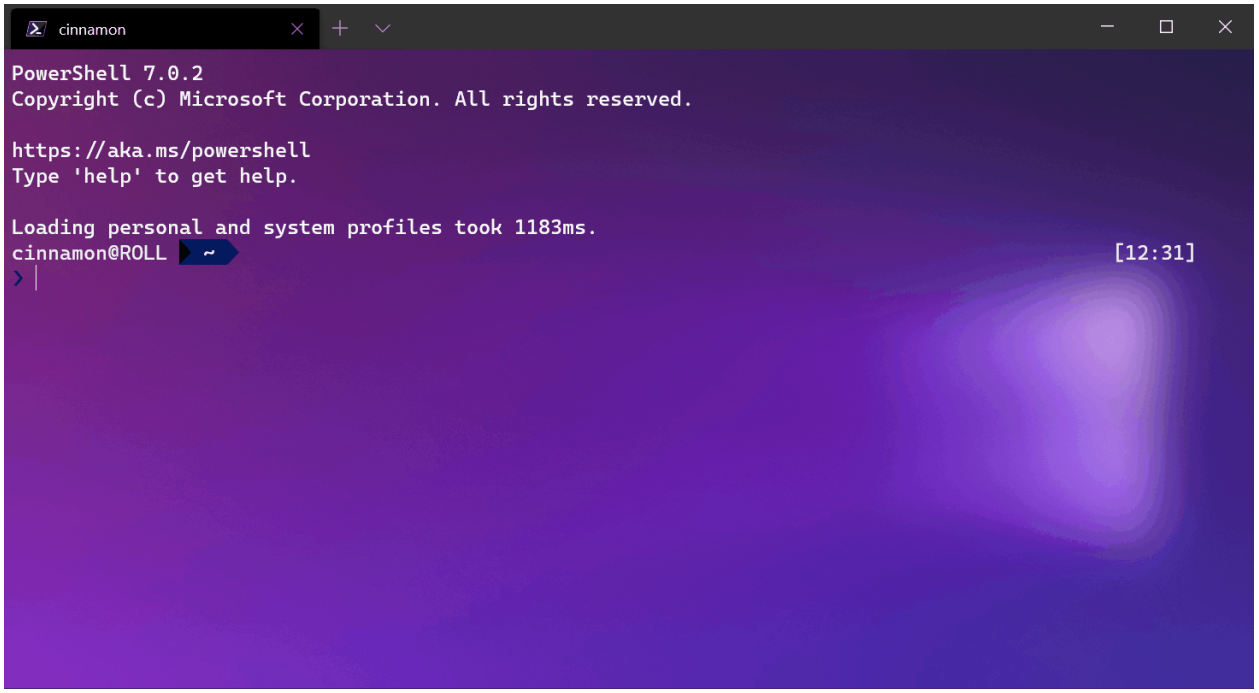
This sets the width of the tabs. `"equal"` makes each tab the same width. `"titleLength"` sizes each tab to the length of its title. `"compact"` will shrink every inactive tab to the width of the icon, leaving the active tab more space to display its full title.

Property name: `tabWidthMode`

Necessity: Optional

Accepts: `"equal"`, `"titleLength"`, `"compact"`

Default value: `"equal"`



Disable pane animations

This disables visual animations across the application when set to `true`.

Property name: `disableAnimations`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Show close all tabs popup

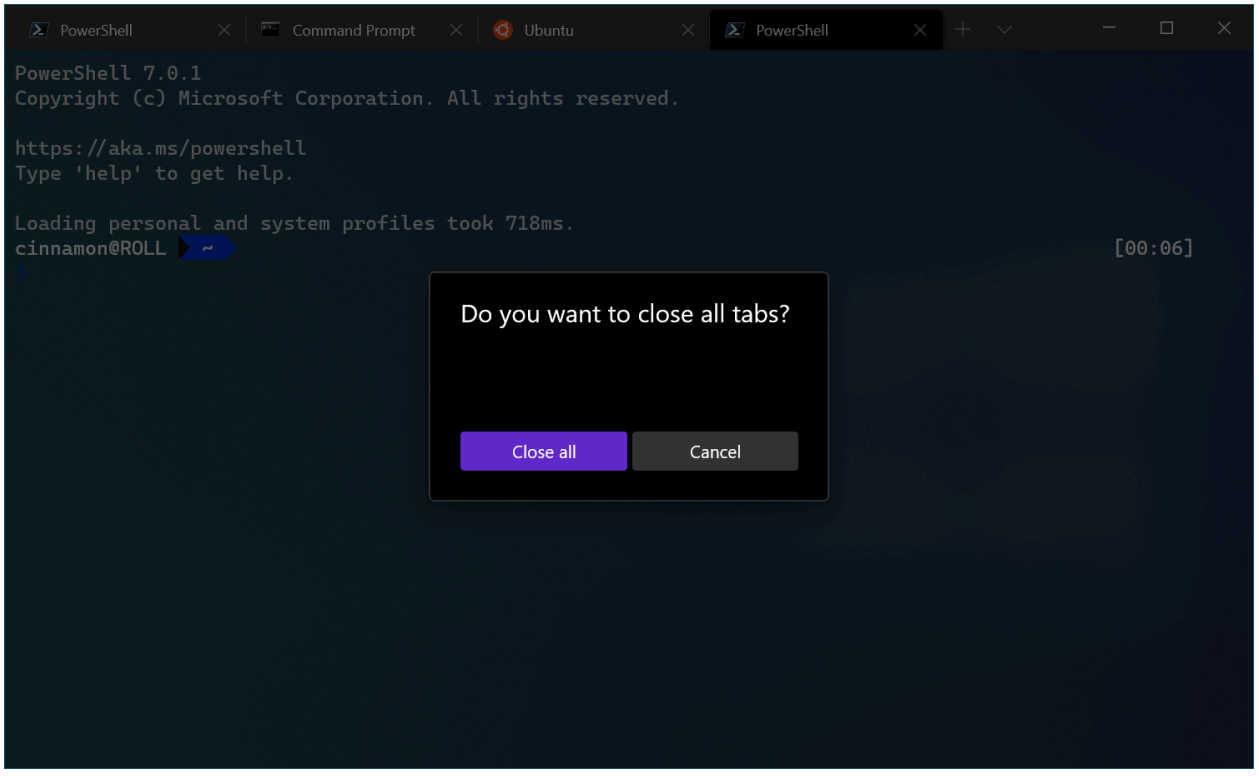
When this is set to `true`, closing a window with multiple tabs open *will* require confirmation. When it's set to `false`, closing a window with multiple tabs open *will not* require confirmation.

Property name: `confirmCloseAllTabs`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`



Use a background image for the entire window

When set to `true`, the background image for the currently focused profile is expanded to encompass the entire window, beneath other panes. This is an experimental feature, and its continued existence is not guaranteed.

Property name: `experimental.useBackgroundImageForWindow`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

New tab dropdown

This setting enables you to configure the list of profiles and the structure of the new tab dropdown menu. This lets you reorder profiles, nest profiles into sub-menus, hide profiles and more. The `newTabMenu` setting accepts a list of "New tab menu entries", which are described below.

An example of this setting might look like:

JSON

```
{
    "newTabMenu": [
        { "type":"profile", "profile": "Command Prompt" },
        { "type":"profile", "profile": "Windows PowerShell" },
        { "type":"separator" },
        {
            "type":"folder",
            "name": "ssh",
            "icon": "C:\\path\\to\\icon.png",
            "entries":
            [
                { "type":"profile", "profile": "Host 1" },
                { "type":"profile", "profile": "8.8.8.8" },
                { "type":"profile", "profile": "Host 2" }
            ]
        },
        {
            "type": "folder",
            "name": "WSL",
            "entries": [ { "type": "matchProfiles", "source":
"Microsoft.Terminal.Wsl" } ]
        },
        { "type": "remainingProfiles" }
    ]
}
```

Property name: `newTabMenu`

Necessity: Optional

Accepts: a list of new tab menu entries

Default value: `[ { "type":"remainingProfiles" } ]`

New tab menu entries

The following are different types of new tab menu entries that can be used in the `newTabMenu` setting. They are each in the form of a JSON object with a `type` property and other properties specific to that entry type. The values for the `type` property are listed below.

- profile
- folder
- separator
- matchProfiles
- remainingProfiles

Profile

This entry type represents a profile from your list of profiles. The profile can be specified by name or GUID.

JSON

```
{ "type":"profile", "profile": "Command Prompt" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`profile`	Required	Profile's name or GUID as a string	Profile that will open based on its GUID or name.

Folder

This entry type represents a nested folder in the new tab dropdown menu. Folders can be nested inside of other folders.

JSON

```
{
    "type":"folder",
```

```
    "name": "ssh",
    "icon": "C:\\path\\to\\icon.png",
    "entries":
    [
        { "type":"profile", "profile": "Host 1" },
        { "type":"profile", "profile": "Host 2" }
    ]
}
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`name`	Required	Folder name as a string	Name of the folder, displayed on the menu entry.
`icon`	Optional	Path to an icon as a string	Path to an icon that will be displayed next to the folder name.
`entries`	Required	List of new tab menu entries	List of new tab menu entries that will be displayed when the folder is clicked.
`allowEmpty`	Optional	Boolean (defaults to `true`)	If set to `true`, the folder will be displayed even if it has no entries. If set to `false`, the folder will not be displayed if it has no entries. This can be useful with `matchProfiles` entries.
`inline`	Optional	Boolean (defaults to `false`)	If set to `true`, and there's only a single entry in the folder, this folder won't create a nested menu. Instead, the entry in the menu will just be the single entry in the folder. This can be useful with `matchProfiles` entries.

Separator

This entry type represents a separator in the new tab dropdown menu.

JSON

```
{ "type":"separator" }
```

Remaining Profiles

This entry type represents all profiles that are not already represented in the new tab dropdown menu. This is useful if you want to have a set of profiles that are always displayed at the top of the new tab dropdown menu, and then have the rest of the profiles displayed in a folder at the bottom of the new tab dropdown menu.

This will return a list of the remaining profiles, in the order they appear in the `profiles` list.

JSON

```
{ "type": "remainingProfiles" }
```

Match Profiles

This entry type is similar to the remaining profiles entry. This entry will expand to a list of profiles that all match a given property. You can match based on the profiles by `name`, `commandline`, or `source`.

For example:

JSON

```
{ "type": "matchProfiles", "source": "Microsoft.Terminal.Wsl" }
```

Will create a set of entries that are all profiles with the `source` property set to `Microsoft.Terminal.Wsl`. Note that the `source` property is set from automatically generated profiles. If you are manually creating a profile and enter a custom source property, it will not be recognized by matchProfiles and the profile will not appear in the list.

A full string comparison is done on these properties - not a regex or partial string match.

Parameters

Expand table

Name	Necessity	Accepts	Description
`name`	Optional	Profile name as a string	A value to compare to the `name` of the profile.

Name	Necessity	Accepts	Description
`commandline`	Optional	Command line as a string	A value to compare to the `commandline` of the profile.
`source`	Optional	Profile source as a string	A value to compare to the `source` of the profile.

Color schemes in Windows Terminal

Article • 10/06/2022

Windows Terminal lets you define your own color schemes, either by using the built-in preset schemes, or by creating your own scheme from scratch. To change schemes, you'll need to edit the settings.json file in an editor such as Visual Studio Code.

Switching to a different color scheme

Launch Windows Terminal and then select the small downward-facing arrow in the title bar. This will open a pull-down menu that lists the available profiles on your system (for example, Windows PowerShell and Command Prompt) and some other options. Select **Settings**, and the settings.json file will open in your default text editor.

This file is where you can define various options per window or per profile. To demonstrate, let's change the color scheme for the Command Prompt profile.

Look down the JSON file until you find the section that includes:

JSON

```
"commandline": "cmd.exe",
"hidden": false
```

Change it to read:

JSON

```
"commandline": "cmd.exe",
"hidden": false,
"colorScheme": "Tango Light"
```

Notice the extra comma in the **hidden** line. Once you save this file, Windows Terminal will update any open window. Open a Command Prompt tab if you haven't already, and you'll immediately see that the colors have changed.

Creating your own color scheme

The "Tango Light" scheme is included as a default option, but you can create your own scheme from scratch or by copying an existing scheme.

Color schemes can be defined in the `schemes` array of your settings.json file. They are written in the following format:

JSON

```
{
    "name" : "Campbell",

    "cursorColor": "#FFFFFF",
    "selectionBackground": "#FFFFFF",

    "background" : "#0C0C0C",
    "foreground" : "#CCCCCC",

    "black" : "#0C0C0C",
    "blue" : "#0037DA",
    "cyan" : "#3A96DD",
    "green" : "#13A10E",
    "purple" : "#881798",
    "red" : "#C50F1F",
    "white" : "#CCCCCC",
    "yellow" : "#C19C00",
    "brightBlack" : "#767676",
    "brightBlue" : "#3B78FF",
    "brightCyan" : "#61D6D6",
    "brightGreen" : "#16C60C",
    "brightPurple" : "#B4009E",
    "brightRed" : "#E74856",
    "brightWhite" : "#F2F2F2",
    "brightYellow" : "#F9F1A5"
},
```

Every setting, aside from `name`, accepts a color as a string in hex format: `"#rgb"` or `"#rrggbb"`. The `cursorColor` and `selectionBackground` settings are optional.

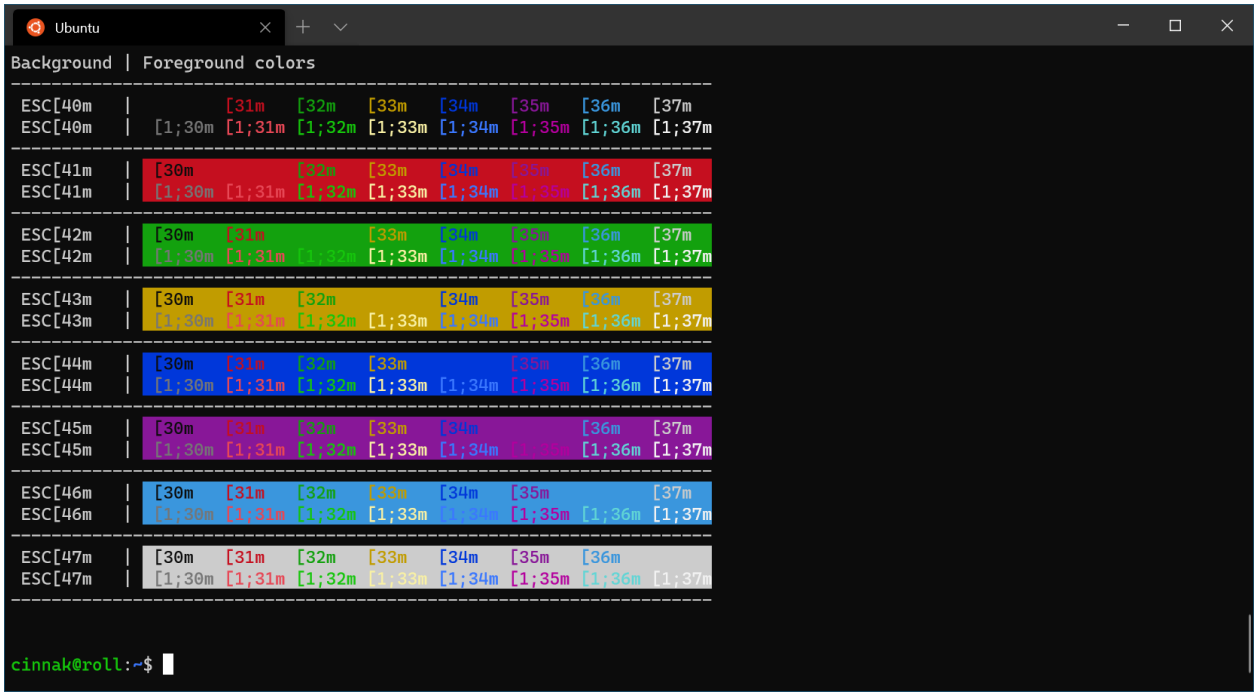
Included color schemes

Windows Terminal includes these color schemes inside the defaults.json file, which can be accessed by holding `Alt` and selecting the settings button. Color schemes can **not** be changed in the defaults.json file. For a color scheme to apply across all profiles, change it in the defaults section of your settings.json file.

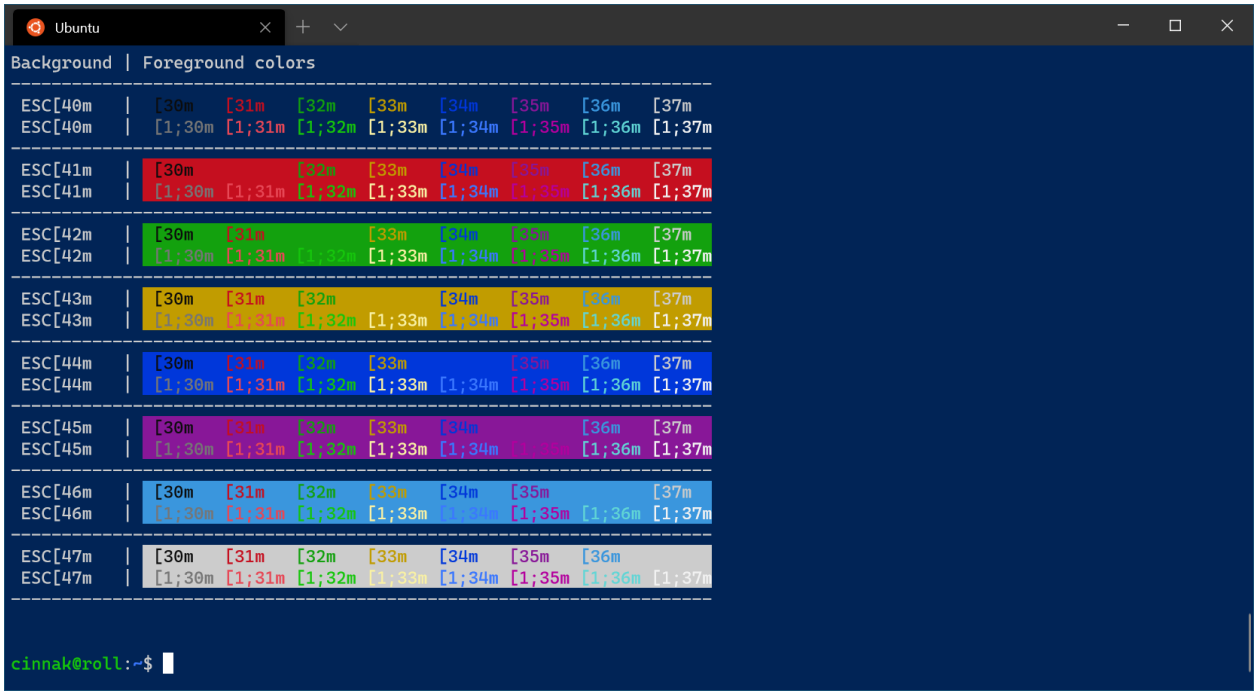
> **Note**

You can print the current color scheme to the Terminal using **colortool**, with the commandline `colortool -c`

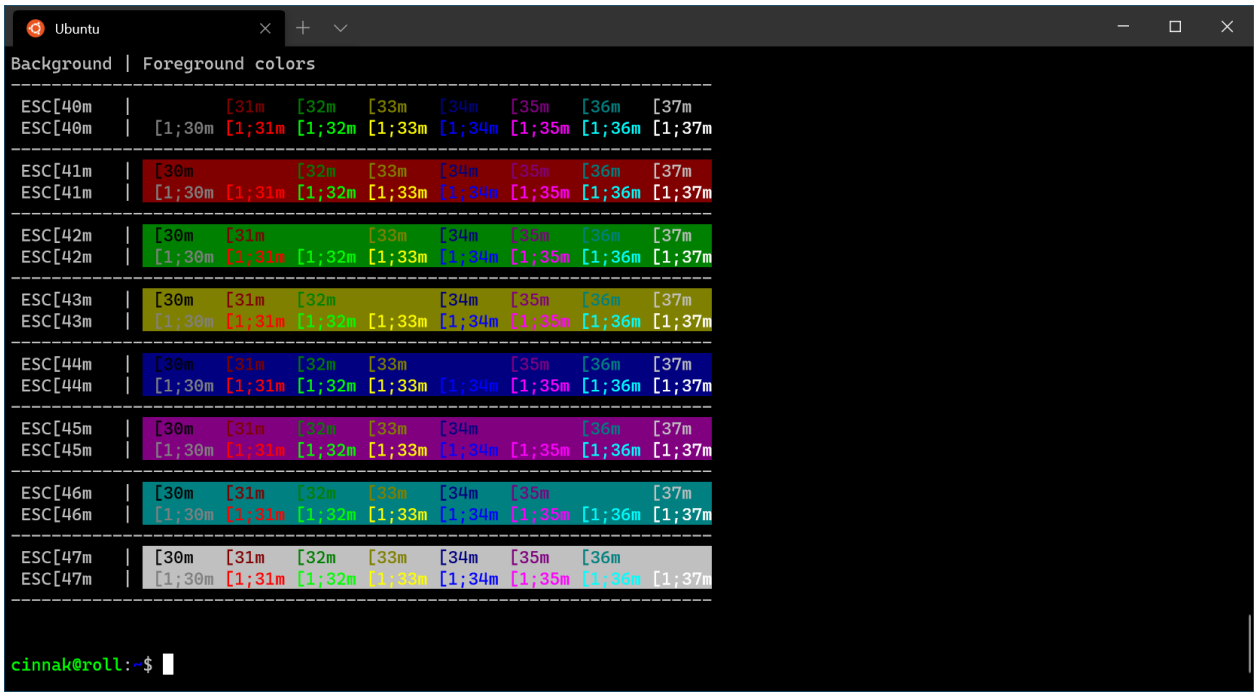
Campbell



Campbell Powershell



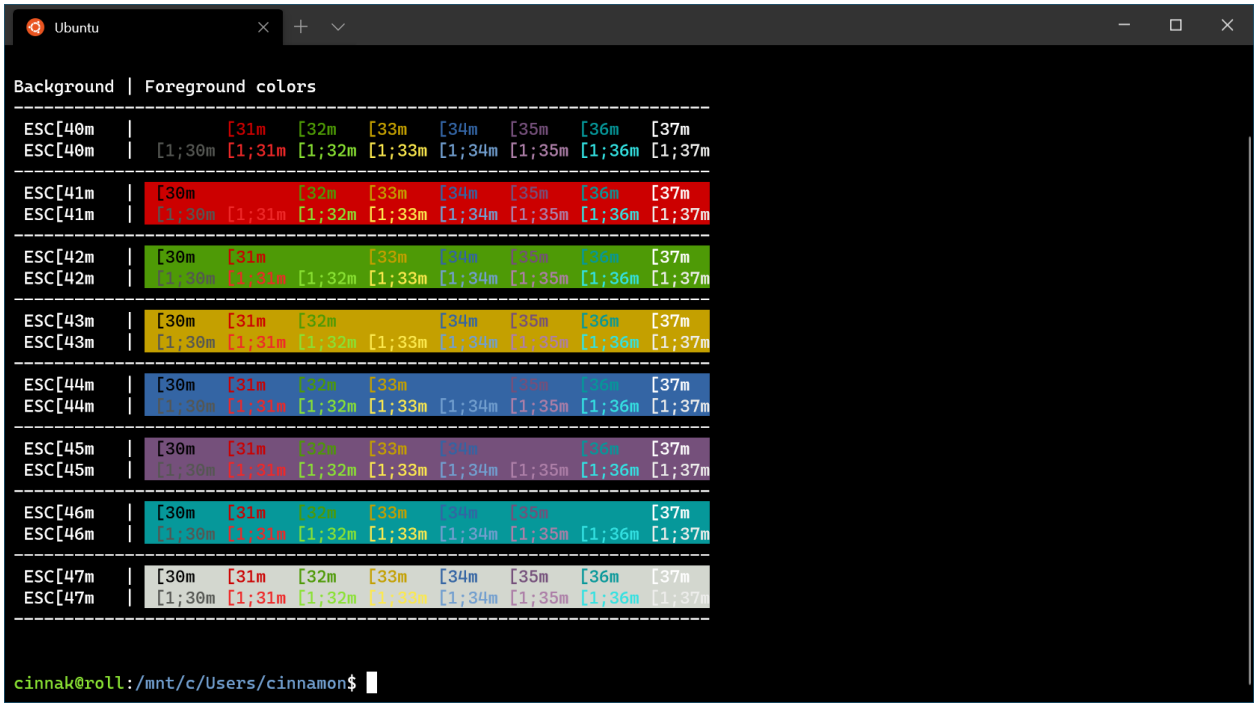
Vintage



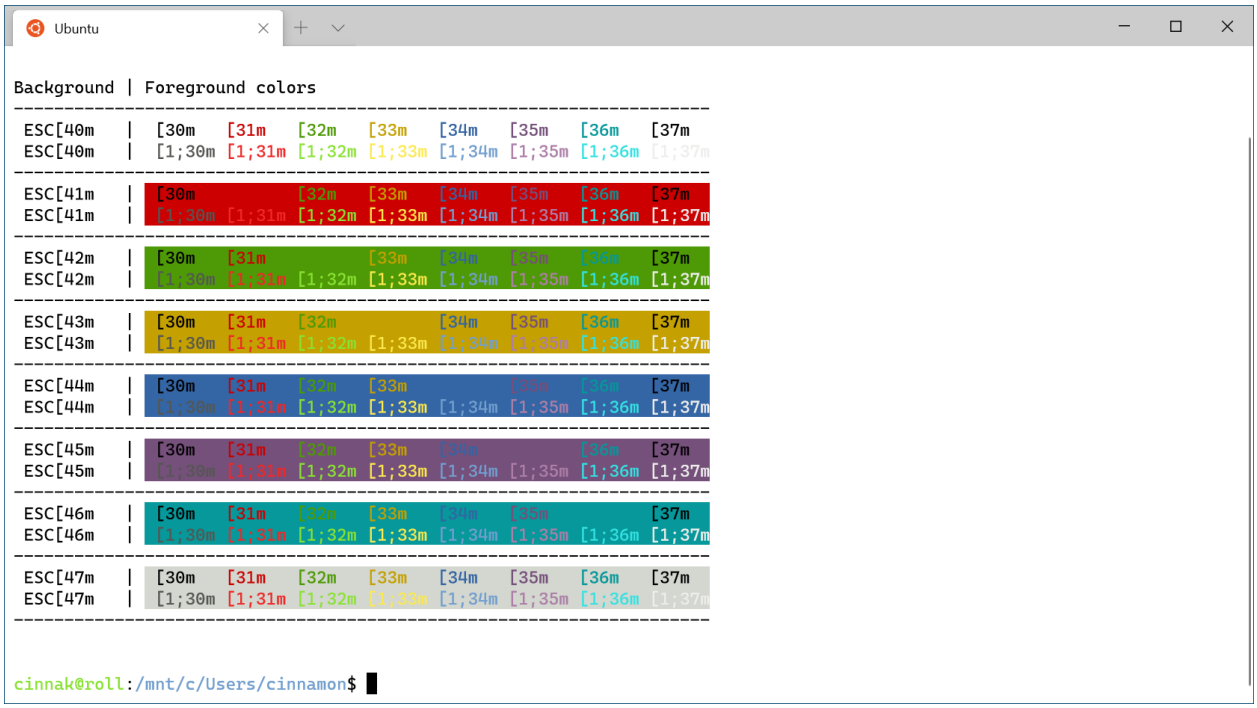
One Half Dark

One Half Light

Tango Dark



Tango Light



More schemes

For more schemes, see the Custom Terminal Gallery section.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Rendering settings in Windows Terminal

Article • 09/28/2023

The properties listed below affect the entire terminal window, regardless of the profile settings. These should be placed at the root of your settings.json file.

If you are thinking about changing the rendering settings, additional information is provided on the Troubleshooting page to help guide you.

Redraw entire screen when display updates

When this set to `true`, the terminal will redraw the entire screen each frame. When set to `false`, it will render only the updates to the screen between frames.

Property name: `experimental.rendering.forceFullRepaint`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Use software rendering

When this is set to `true`, the terminal will use the software renderer (a.k.a. WARP) instead of the hardware one.

Property name: `experimental.rendering.software`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Enable unfocused acrylic

When this is set to `true`, the terminal will attempt to use acrylic even when the window is unfocused. This will only work if acrylic is enabled in the OS, and in your profile. When

set to `false`, acrylic will only be used when the window is focused (even if `useAcrylic` is set to `true` in a profile's `unfocusedAppearance`).

Property name: `compatibility.enableUnfocusedAcrylic`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

> ⓘ **Important**
>
> This feature is only available in **Windows Terminal Preview**.

Custom actions and keybindings in Windows Terminal

Article • 10/29/2024

You can create custom actions inside Windows Terminal that give you control of how you interact with the terminal. These actions will automatically be added to the command palette.

Action formats

Actions can be structured in the following formats:

Commands without arguments

JSON

```
{ "command": "commandName", "id": "User.MyCommand" }
```

For example, this default setting uses the shortcut key `Alt+F4` to close the terminal window:

JSON

```
{ "command": "closeWindow", "id": "User.MyCloseWindow" }
```

Commands with arguments

JSON

```
{ "command": { "action": "commandName", "argument": "value" }, "id":
"User.MyCommand" }
```

For example, this default setting uses the shortcut key `Ctrl+Shift+1` to open a new tab in the terminal based on whichever profile is listed first in your dropdown menu (typically this will open the PowerShell profile):

JSON

```
{ "command": { "action": "newTab", "index": 0 }, "id": "User.MyNewTabAction"
```

```
}
```

Commands with command line arguments

JSON

```
{ "command": { "action": "wt", "commandline": "value" }, "keys":
"modifiers+key" }
```

For example, this default setting uses the shortcut key `Ctrl+Shift+O` to use wt to open a new PowerShell tab with additional panes for Command Prompt and Ubuntu:

JSON

```
{
  "command":
  {
    "action": "wt",
    "commandline": "new-tab pwsh.exe ; split-pane -p \"Command Prompt\" -d
C:\\ ; split-pane -p \"Ubuntu\" -H"
  },
  "keys": "ctrl+shift+o"
}
```

Action properties

Actions are stored in the `actions` array and can be constructed using the following properties.

Command

This is the command executed when the associated keys are pressed.

Property name: `command`

Necessity: Required

Accepts: String

Action

This adds additional functionality to certain commands.

Property name: `action`

Necessity: Optional

Accepts: String

Name

This sets the name that will appear in the command palette. If one isn't provided, the terminal will attempt to automatically generate a name.

Property name: `name`

Necessity: Optional

Accepts: String

Icon

This sets the icon that displays within the command palette.

Property name: `icon`

Necessity: Optional

Accepts: File location as a string, or an emoji

ID

This sets the id of this action. If one isn't provided, the terminal will generate an ID for this action. The ID is used to refer to this action when creating keybindings.

Property name: `id`

Necessity: Optional

Accepts: String

Keybindings

Actions can be assigned keybindings by referring to them with their unique ID. For example, here is a possible `keybindings` array that assigns `Alt+F4`, `Ctrl+Shift+1` and

`Ctrl+Shift+o` to the actions defined above. Multiple keybinding entries may be created for the same action.

JSON

```
"keybindings": [
  { "keys": "alt+f4", "id": "User.MyCloseWindow" },
  { "keys": "ctrl+shift+1", "id": "User.MyNewTabAction" },
  { "keys": "ctrl+shift+o", "id": "User.MyCoolSetup"}
]
```

Keybinding properties

Keybindings are stored in the `keybindings` array and are constructed using the following properties.

Keys

This defines the key combinations used to call the command. Keys can have any number of modifiers with one key. Accepted modifiers and keys are listed below.

If the action does not have keys, it will appear in the command palette but cannot be invoked with the keyboard.

Property name: `keys`

Necessity: Required

Accepts: String or array[string]

ID

This is the ID of the action to be invoked when this keybinding is pressed.

Property name: `id`

Necessity: Required

Accepts: String

Accepted Modifiers

`ctrl+`, `shift+`, `alt+`, `win+`

> **Note**
>
> While the `Windows` key is supported as a modifier, the system reserves most `Win+<key>` key bindings. If the OS has reserved that key binding, the terminal will never receive that binding.

Modifier keys

Expand table

Type	Keys
Function and alphanumeric keys	`f1-f24`, `a-z`, `0-9`
Symbols	`` ` ``, `plus`, `-`, `=`, `[`, `]`, `\`, `;`, `'`, `,`, `.`, `/`
Arrow keys	`down`, `left`, `right`, `up`, `pagedown`, `pageup`, `pgdn`, `pgup`, `end`, `home`
Action keys	`tab`, `enter`, `esc`, `escape`, `space`, `backspace`, `delete`, `insert`, `app`, `menu`
Numpad keys	`numpad_0-numpad_9`, `numpad0-numpad9`, `numpad_add`, `numpad_plus`, `numpad_decimal`, `numpad_period`, `numpad_divide`, `numpad_minus`, `numpad_subtract`, `numpad_multiply`
Browser keys	`browser_back`, `browser_forward`, `browser_refresh`, `browser_stop`, `browser_search`, `browser_favorites`, `browser_home`

Note: `=` and `plus` are equivalents. The latter must not be confused with `numpad_plus`.

Application-level commands

Quit

This closes all open terminal windows. A confirmation dialog will appear in the current window to ensure you'd like to close all windows.

Command name: `quit`

Default ID:

JSON

```
{ "command": "quit", "id": "Terminal.Quit" }
```

Close window

This closes the current window and all tabs within it. If `confirmCloseAllTabs` is set to `true`, a confirmation dialog will appear to ensure you'd like to close all your tabs. More information on this setting can be found on the Appearance page.

Command name: `closeWindow`

Default ID:

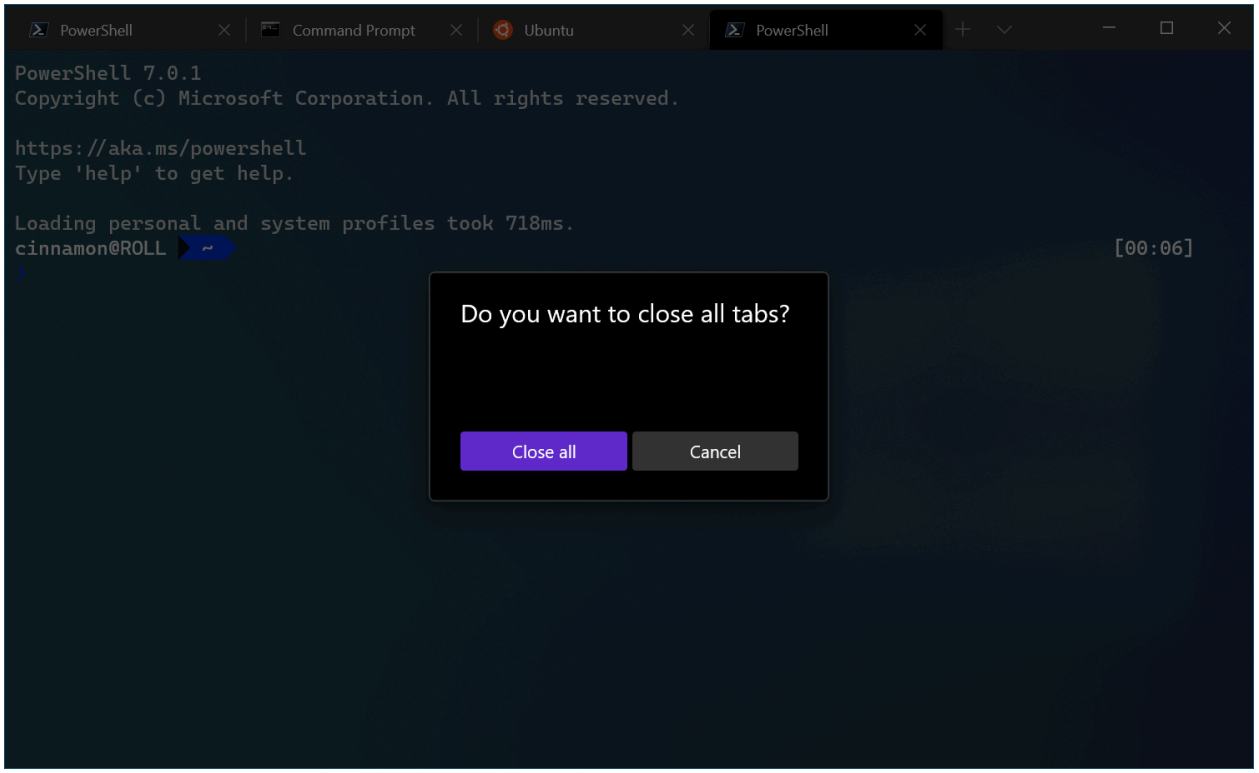
JSON

```
{ "command": "closeWindow", "id": "Terminal.CloseWindow" }
```

Default binding:

JSON

```
{ "keys": "alt+f4", "id": "Terminal.CloseWindow" }
```



Find

This opens the search dialog box. More information on search can be found on the Search page.

Command name: `find`

Default ID:

JSON

```
{ "command": "find", "id": "Terminal.FindText" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+f", "id": "Terminal.FindText" }
```

Find next/previous search match

This lets you navigate through your search matches.

Command name: `findMatch`

Default IDs:

JSON

```
{ "command": { "action": "findMatch", "direction": "next" }, "id":
"Terminal.FindNextMatch" },
{ "command": { "action": "findMatch", "direction": "prev" }, "id":
"Terminal.FindPrevMatch" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`direction`	Required	`"next"`, `"prev"`	The direction to navigate through search results.

Open the dropdown

This opens the dropdown menu.

Command name: `openNewTabDropdown`

Default ID:

```JSON
{ "command": "openNewTabDropdown", "id": "Terminal.OpenNewTabDropdown" }
```

Default binding:

```JSON
{ "keys": "ctrl+shift+space", "id": "Terminal.OpenNewTabDropdown" }
```

Open settings files

This opens either the settings UI, custom settings file (settings.json), or default settings file (`defaults.json`), depending on the `target` field. Without the `target` field, the custom settings file will be opened.

Command name: `openSettings`

Default IDs:

```JSON
{ "command": { "action": "openSettings", "target": "settingsUI" }, "id":
"Terminal.OpenSettingsUI" },
{ "command": { "action": "openSettings", "target": "settingsFile" }, "id":
"Terminal.OpenSettingsFile" },
{ "command": { "action": "openSettings", "target": "defaultsFile" }, "keys":
"Terminal.OpenDefaultSettingsFile" }
```

Default bindings:

```JSON
{ "keys": "ctrl+,", "id": "Terminal.OpenSettingsUI" },
{ "keys": "ctrl+shift+,", "id": "Terminal.OpenSettingsFile" },
{ "keys": "ctrl+alt+,", "id": "Terminal.OpenDefaultSettingsFile" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`target`	Optional	`"settingsFile"`, `"defaultsFile"`, `"settingsUI"`, `"allFiles"`	The settings file to open.

Open system menu

Opens the system menu at the top left corner of the window.

Command name: `openSystemMenu`

Default ID:

JSON

```
{ "command": "openSystemMenu", "id": "Terminal.OpenSystemMenu" }
```

Default binding:

JSON

```
{ "keys": "alt+space", "id": "Terminal.OpenSystemMenu" }
```

Toggle full screen

This allows you to switch between full screen and default window sizes.

Command name: `toggleFullscreen`

Default ID

JSON

```
{ "command": "toggleFullscreen", "id": "Terminal.ToggleFullscreen" }
```

Default bindings:

JSON

```
{ "keys": "alt+enter", "id": "Terminal.ToggleFullscreen" },
{ "keys": "f11", "id": "Terminal.ToggleFullscreen" }
```

Toggle focus mode

This allows you to enter "focus mode", which hides the tabs and title bar.

Command name: `toggleFocusMode`

Default ID:

JSON

```
{ "command": "toggleFocusMode", "id": "Terminal.ToggleFocusMode" }
```

Toggle always on top mode

This allows you toggle the "always on top" state of the window. When in "always on top" mode, the window will appear on top of all other non-topmost windows.

Command name: `toggleAlwaysOnTop`

Default ID:

JSON

```
{ "command": "toggleAlwaysOnTop", "id": "Terminal.ToggleAlwaysOnTop" }
```

Send input

Send arbitrary text input to the shell. As an example the input `"text\n"` will write "text" followed by a newline to the shell.

ANSI escape sequences may be used, but escape codes like `\x1b` must be written as `\u001b`. For instance `"\u001b[A"` will behave as if the up arrow button had been pressed.

Command name: `sendInput`

Default binding:

This command is not currently bound in the default settings.

JSON

```
{ "command": { "action": "sendInput", "input": "\u001b[A" } }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`input`	Required	String	The text input to feed into the shell.

Tab management commands

Close tab

This closes the tab at a given index. If no index is provided, use the focused tab's index.

Command name: `closeTab`

Parameters

Expand table

Name	Necessity	Accepts	Description
`index`	Optional	Integer	Position of the tab to close.

Close all other tabs

This closes all tabs except for the one at an index. If no index is provided, use the focused tab's index.

Command name: `closeOtherTabs`

Default ID:

JSON

```
{ "command": "closeOtherTabs", "id": "Terminal.CloseOtherTabs" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`index`	Optional	Integer	Position of the tab to be kept open.

Close tabs after index

This closes the tabs following the tab at an index. If no index is provided, use the focused tab's index.

Command name: `closeTabsAfter`

Default ID:

JSON

```
{ "command": "closeTabsAfter", "id": "Terminal.CloseTabsAfter" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`index`	Optional	Integer	Position of the last tab to be kept open.

Duplicate tab

This makes a copy of the current tab's profile and directory and opens it. This does not include modified/added ENV VARIABLES.

Command name: `duplicateTab`

Default ID:

JSON

```
{ "command": "duplicateTab", "id": "Terminal.DuplicateTab" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+d", "id": "Terminal.DuplicateTab" }
```

New tab

This creates a new tab. Without any arguments, this will open the default profile in a new tab. If an index is not specified, the default profile's equivalent setting will be used. If the index doesn't map to a profile, the keys are passed directly to the terminal (or ignored if no keys were used to invoke the action).

Command name: `newTab`

Default IDs:

JSON

```
{ "command": "newTab", "id": "Terminal.OpenNewTab" },
{ "command": { "action": "newTab", "index": 0 }, "id":
"Terminal.OpenNewTabProfile0" },
{ "command": { "action": "newTab", "index": 1 }, "id":
"Terminal.OpenNewTabProfile1" },
{ "command": { "action": "newTab", "index": 2 }, "id":
"Terminal.OpenNewTabProfile2" },
{ "command": { "action": "newTab", "index": 3 }, "id":
"Terminal.OpenNewTabProfile3" },
{ "command": { "action": "newTab", "index": 4 }, "id":
"Terminal.OpenNewTabProfile4" },
{ "command": { "action": "newTab", "index": 5 }, "id":
"Terminal.OpenNewTabProfile5" },
{ "command": { "action": "newTab", "index": 6 }, "id":
"Terminal.OpenNewTabProfile6" },
{ "command": { "action": "newTab", "index": 7 }, "id":
"Terminal.OpenNewTabProfile7" },
{ "command": { "action": "newTab", "index": 8 }, "id":
"Terminal.OpenNewTabProfile8" }
```

Default bindings:

JSON

```
{ "keys": "ctrl+shift+t", "id": "Terminal.OpenNewTab" },
{ "keys": "ctrl+shift+1", "id": "Terminal.OpenNewTabProfile0" },
{ "keys": "ctrl+shift+2", "id": "Terminal.OpenNewTabProfile1" },
{ "keys": "ctrl+shift+3", "id": "Terminal.OpenNewTabProfile2" },
{ "keys": "ctrl+shift+4", "id": "Terminal.OpenNewTabProfile3" },
{ "keys": "ctrl+shift+5", "id": "Terminal.OpenNewTabProfile4" },
{ "keys": "ctrl+shift+6", "id": "Terminal.OpenNewTabProfile5" },
{ "keys": "ctrl+shift+7", "id": "Terminal.OpenNewTabProfile6" },
```

```
{ "keys": "ctrl+shift+8", "id": "Terminal.OpenNewTabProfile7" },
{ "keys": "ctrl+shift+9", "id": "Terminal.OpenNewTabProfile8" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`commandline`	Optional	Executable file name as a string	Executable run within the tab.
`startingDirectory`	Optional	Folder location as a string	Directory in which the tab will open.
`elevate`	Optional	`true`, `false`, `null`	Overrides the elevate property of the profile. When omitted, this action will behave according to the profile's `elevate` setting. When set to `true` or `false`, this action will behave as though the profile was set with `"elevate": true` or `"elevate": false` (respectively).
`tabTitle`	Optional	String	Title of the new tab.
`index`	Optional	Integer	Profile that will open based on its position in the dropdown (starting at 0).
`profile`	Optional	Profile's name or GUID as a string	Profile that will open based on its GUID or name.
`colorScheme`	Optional	The name of a color scheme as a string	The scheme to use instead of the profile's set `colorScheme`
`suppressApplicationTitle`	Optional	`true`, `false`	When set to `false`, applications can change the tab title by sending title change messages. When set to `true`, these messages are suppressed. If not provided, the behavior is inherited from the profile's settings. In order to enter a new tab title and have that title persist, this must be set to true.

Open next tab

This opens the tab to the right of the current one.

Command name: `nextTab`

Default ID:

JSON

```
{ "command": "nextTab", "id": "Terminal.NextTab" }
```

Default binding:

JSON

```
{ "keys": "ctrl+tab", "id": "Terminal.NextTab" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`tabSwitcherMode`	Optional	`"mru"`, `"inOrder"`, `"disabled"`	Move to the next tab using `"tabSwitcherMode"`. If no mode is provided, use the globally defined one.

Open previous tab

This opens the tab to the left of the current one.

Command name: `prevTab`

Default ID:

JSON

```
{ "command": "prevTab", "id": "Terminal.PrevTab" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+tab", "id": "Terminal.PrevTab" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`tabSwitcherMode`	Optional	`"mru"`, `"inOrder"`, `"disabled"`	Move to the previous tab using `"tabSwitcherMode"`. If no mode is provided, use the globally defined one.

Tab search

This opens the tab search box.

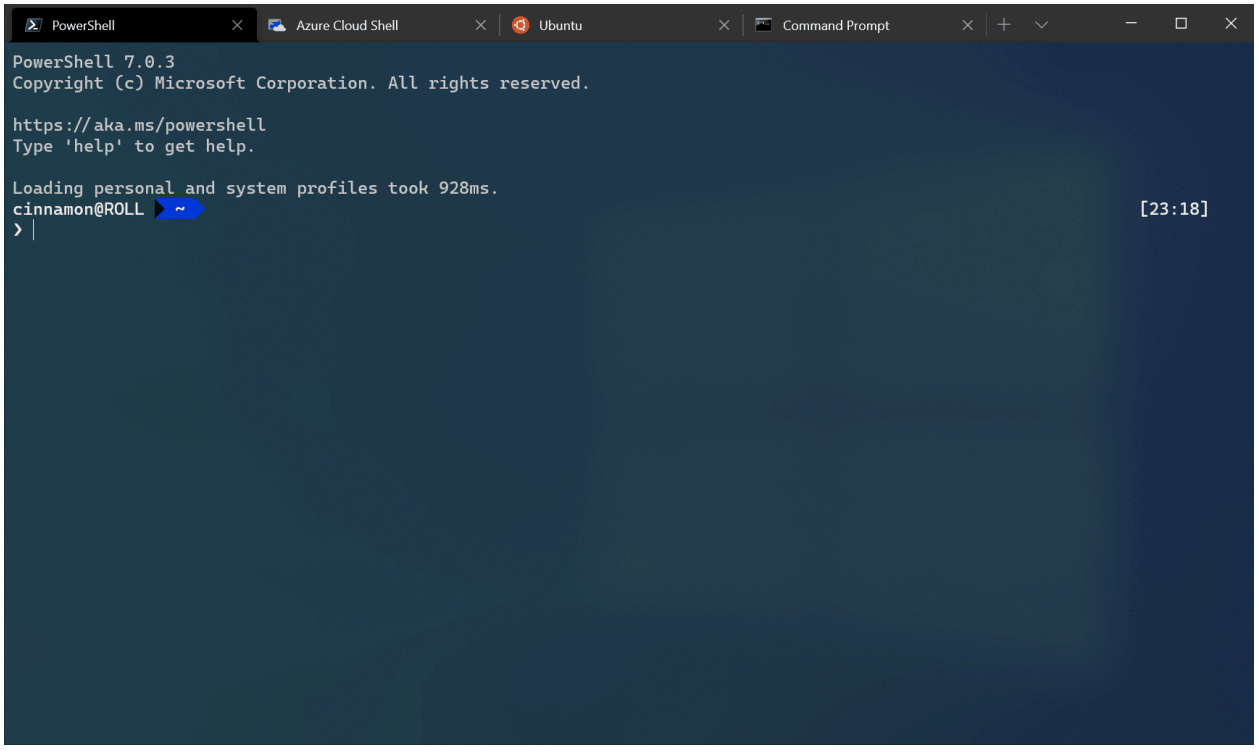
Command name: `tabSearch`

Default binding:

This command is not currently bound in the default settings.

JSON

```
{"command": "tabSearch"}
```



Open a specific tab

This opens a specific tab depending on the index.

Command name: `switchToTab`

Default IDs:

JSON

```
{ "command": { "action": "switchToTab", "index": 0 }, "id":
"Terminal.SwitchToTab0" },
{ "command": { "action": "switchToTab", "index": 1 }, "id":
"Terminal.SwitchToTab1" },
{ "command": { "action": "switchToTab", "index": 2 }, "id":
"Terminal.SwitchToTab2" },
{ "command": { "action": "switchToTab", "index": 3 }, "id":
"Terminal.SwitchToTab3" },
{ "command": { "action": "switchToTab", "index": 4 }, "id":
"Terminal.SwitchToTab4" },
{ "command": { "action": "switchToTab", "index": 5 }, "id":
"Terminal.SwitchToTab5" },
{ "command": { "action": "switchToTab", "index": 6 }, "id":
"Terminal.SwitchToTab6" },
{ "command": { "action": "switchToTab", "index": 7 }, "id":
"Terminal.SwitchToTab7" }
```

Default bindings:

JSON

```
{ "keys": "ctrl+alt+1", "id": "Terminal.SwitchToTab0" },
{ "keys": "ctrl+alt+2", "id": "Terminal.SwitchToTab1" },
{ "keys": "ctrl+alt+3", "id": "Terminal.SwitchToTab2" },
{ "keys": "ctrl+alt+4", "id": "Terminal.SwitchToTab3" },
{ "keys": "ctrl+alt+5", "id": "Terminal.SwitchToTab4" },
{ "keys": "ctrl+alt+6", "id": "Terminal.SwitchToTab5" },
{ "keys": "ctrl+alt+7", "id": "Terminal.SwitchToTab6" },
{ "keys": "ctrl+alt+8", "id": "Terminal.SwitchToTab7" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`index`	Required	Integer	Tab that will open based on its position in the tab bar (starting at 0).

Rename tab

This command can be used to rename a tab to a specific string.

Command name: `renameTab`

Default binding:

This command is not currently bound in the default settings.

JSON

```
// Rename a tab to "Foo"
{ "command": { "action": "renameTab", "title": "Foo" } }

// Reset the tab's name
{ "command": { "action": "renameTab", "title": null } }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`title`	Optional	String	The new title to use for this tab. If omitted, this command will revert the tab title back to its original value.

Open tab rename text box

This command changes the tab title into a text field that lets you edit the title for the current tab. Clearing the text field will reset the tab title back to the default for the current shell instance.

Command name: `openTabRenamer`

Default ID:

JSON

```
{ "command": "openTabRenamer", "id": "Terminal.OpenTabRenamer" }
```

Change tab color

This command can be used to change the color of a tab to a specific value.

Command name: `setTabColor`

Default binding:

This command is not currently bound in the default settings.

JSON

```
// Change the tab's color to a bright magenta
{ "command": { "action": "setTabColor", "color": "#ff00ff" } }

// Reset the tab's color
{ "command": { "action": "setTabColor", "color": null } }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`color`	Optional	String, in hex format: `"#rgb"` or `"#rrggbb"`	The new color to use for this tab. If omitted, this command will revert the tab's color back to its original value.

Open tab color picker

This command can be used to open the color picker for the active tab. The color picker can be used to set a color for the tab at runtime.

Command name: `openTabColorPicker`

Default ID:

JSON

```
{ "command": "openTabColorPicker", "id": "Terminal.OpenTabColorPicker" }
```

Move tab

This command moves the tab "backward" and "forward", which is equivalent to "left" and "right" in left-to-right UI.

Command name: `moveTab`

Default IDs:

```JSON
// Move tab backward (left in LTR)
{ "command": { "action": "moveTab", "direction": "backward" }, "id":
"Terminal.MoveTabBackward" }

// Move tab forward (right in LTR)
{ "command": { "action": "moveTab", "direction": "forward" }, "id":
"Terminal.MoveTabForward" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`direction`	Required	`"backward"`, `"forward"`	Direction in which the tab will move.
`window`	Optional	A window ID	See below

`window` is optional, and follows the same format as the `--window-id` argument to the `wt.exe` command line. If it's omitted, then this will move the tab within the current window. If provided, it may either be the integer ID of a window, or the name of a window. It also accepts the following reserved values:

- `"new"` or `-1`: Always run this command in a new window
- `"last"` or `0`: Always run this command in the most recently used window

If no window exists with the given `window` ID, then a new window will be created with that id/name.

Broadcast input

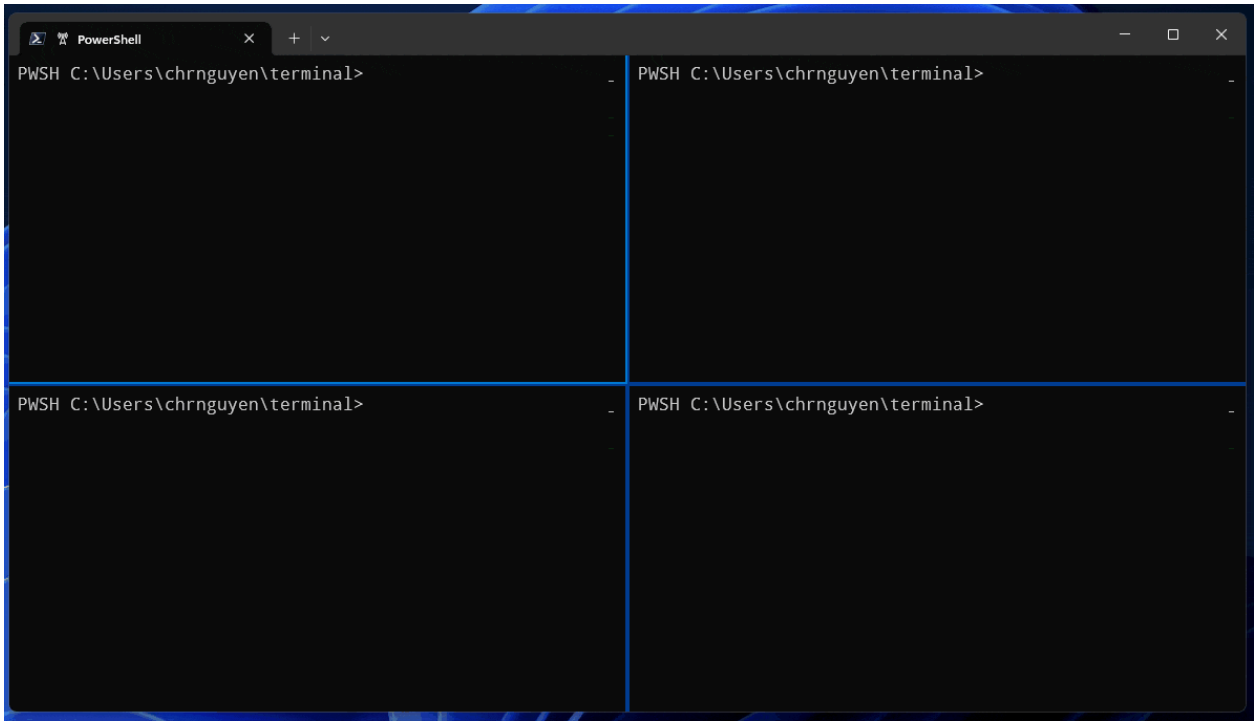
This command will toggle "broadcast mode" for a pane. When broadcast mode is enabled, all input sent to the pane will be sent to all panes in the same tab. This is useful for sending the same input to multiple panes at once.

As with any action, you can also invoke "broadcast mode" by search for "Toggle broadcast input to all panes" in the Command palette.

Command name: `toggleBroadcastInput`

Default ID:

```JSON
{ "command": "toggleBroadcastInput", "id": "Terminal.ToggleBroadcastInput" }
```



Open context menu

This command will open the "right-click" context menu for the active pane. This menu has context-relevant actions for managing panes, copying and pasting, and more. This action does not require the `experimental.rightClickContextMenu` setting to be enabled.

Command name: `showContextMenu`

Default ID:

```JSON
{ "command": "showContextMenu", "id": "Terminal.ShowContextMenu" }
```

Open about dialog

This command will open the about dialog for the terminal. This dialog contains information about the terminal, including the version number, the license, and more.

Command name: `openAbout`

Default ID:

```
JSON
{ "command": "openAbout", "id": "Terminal.OpenAboutDialog" }
```

> **Important**
>
> This feature is only available in Windows Terminal Preview.

Search the web

Attempts to open a browser window with a search for the selected text. This does nothing if there's no text selected. If the `queryUrl` parameter is not provided, the `searchWebDefaultQueryUrl` setting will be used instead. If the `queryUrl` parameter is provided, a `%s` in the string will be replaced by the selected text.

Command name: `searchWeb`

Default ID:

```
JSON
{ "command": { "action": "searchWeb" }, "id": "Terminal.SearchWeb" },
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`queryUrl`	Required	String	URL to use to search with. A `%s` in this string will be replaced by the selected text. If omitted, will default to the `searchWebDefaultQueryUrl` setting.

> **Important**
>
> This feature is only available in Windows Terminal Preview.

Window management commands

New window

This creates a new window. Without any arguments, this will open the default profile in a new window (regardless of the setting of `windowingBehavior`). If an action is not specified, the default profile's equivalent setting will be used.

Command name: `newWindow`

Default ID:

```JSON
{ "command": "newWindow", "id": "Terminal.OpenNewWindow" },
```

Default binding:

```JSON
{ "keys": "ctrl+shift+n", "id": "Terminal.OpenNewWindow" },
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`commandline`	Optional	Executable file name as a string	Executable run within the tab.
`startingDirectory`	Optional	Folder location as a string	Directory in which the window will open.
`tabTitle`	Optional	String	Title of the window tab.
`index`	Optional	Integer	Profile that will open based on its position in the dropdown (starting at 0).

Name	Necessity	Accepts	Description
`profile`	Optional	Profile's name or GUID as a string	Profile that will open based on its GUID or name.
`suppressApplicationTitle`	Optional	`true`, `false`	When set to `false` allows applications to change tab title by sending title change messages. When set to `true` suppresses these messages. If not provided, the behavior is inherited from profile settings.

Rename window

This command can be used to rename a window to a specific string.

Command name: `renameWindow`

Default binding:

This command is not currently bound in the default settings.

JSON

```
// Rename a window to "Foo"
{ "command": { "action": "renameWindow", "name": "Foo" } }

// Reset the window's name
{ "command": { "action": "renameWindow", "name": null } }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`name`	Optional	String	The new name to use for this window. If omitted, this command will revert the window name back to its original value.

Open window rename dialog

This command changes displays a popup window that lets you edit the name for the current window. Clearing the text field will reset the window name.

Command name: `openWindowRenamer`

Default ID:

```JSON
{ "command": "openWindowRenamer", "id": "Terminal.OpenWindowRenamer" }
```

Identify window

This pops up an overlay on the focused window that displays the window's name and index.

Command name: `identifyWindow`

Default ID:

```JSON
{"command": "identifyWindow", "id": "Terminal.IdentifyWindow" },
```

Identify windows

This pops up an overlay on all windows that displays each window's name and index.

Command name: `identifyWindows`

Default binding:

This command is not currently bound in the default settings.

```JSON
{ "command": "identifyWindows" },
```

Pane management commands

Split a pane

This halves the size of the active pane and opens another. Without any arguments, this will open the default profile in the new pane. If an action is not specified, the default profile's equivalent setting will be used.

Command name: `splitPane`

Default IDs:

JSON

```
{ "command": { "action": "splitPane", "splitMode": "duplicate", "split":
"auto" }, "id": "Terminal.DuplicatePaneAuto" },
{ "command": { "action": "splitPane", "split": "up" }, "id":
"Terminal.SplitPaneUp" },
{ "command": { "action": "splitPane", "split": "down" }, "id":
"Terminal.SplitPaneDown" },
{ "command": { "action": "splitPane", "split": "left" }, "id":
"Terminal.SplitPaneLeft" },
{ "command": { "action": "splitPane", "split": "right" }, "id":
"Terminal.SplitPaneRight" },
{ "command": { "action": "splitPane", "splitMode": "duplicate", "split":
"down" }, "id": "Terminal.DuplicatePaneDown" },
{ "command": { "action": "splitPane", "splitMode": "duplicate", "split":
"right" }, "id": "Terminal.DuplicatePaneRight" }
```

Default bindings:

JSON

```
{ "keys": "alt+shift+d", "id": "Terminal.DuplicatePaneAuto" },
{ "keys": "alt+shift+-", "id": "Terminal.DuplicatePaneDown" },
{ "keys": "alt+shift+plus", "id": "Terminal.DuplicatePaneRight" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`split`	Required	`"vertical"`, `"horizontal"`, `"auto"`, `"up"`, `"right"`, `"down"`, `"left"`	How the pane will split. `"auto"` will split in the direction that provides the most surface area.
`commandline`	Optional	Executable file name as a string	Executable run within the pane.

Name	Necessity	Accepts	Description
`startingDirectory`	Optional	Folder location as a string	Directory in which the pane will open.
`elevate`	Optional	`true`, `false`, `null`	Overrides the elevate property of the profile. When omitted, this action will behave according to the profile's `elevate` setting. When set to `true` or `false`, this action will behave as though the profile was set with `"elevate": true` or `"elevate": false` (respectively).
`tabTitle`	Optional	String	Title of the tab when the new pane is focused.
`index`	Optional	Integer	Profile that will open based on its position in the dropdown (starting at 0).
`profile`	Optional	Profile's name or GUID as a string	Profile that will open based on its GUID or name.
`colorScheme`	Optional	The name of a color scheme as a string	The scheme to use instead of the profile's set `colorScheme`
`suppressApplicationTitle`	Optional	`true`, `false`	When set to `false`, applications can change the tab title by sending title change messages. When set to `true`, these messages are suppressed. If not provided, the behavior is inherited from the profile's settings.
`splitMode`	Optional	`"duplicate"`	Controls how the pane splits. Only accepts `"duplicate"`, which will duplicate the focused pane's profile into a new pane.
`size`	Optional	Float	Specify how large the new pane should be, as a fraction of the current pane's size. `1.0` would be "all of the current pane", and `0.0` is "None of the parent". Defaults to `0.5`.

Close pane

This closes the active pane. If there aren't any split panes, this will close the current tab. If there is only one tab open, this will close the window.

Command name: `closePane`

Default ID:

JSON

```
{ "command": "closePane", "id": "Terminal.ClosePane" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+w", "id": "Terminal.ClosePane" }
```

Move pane focus

This changes focus to a different pane depending on the direction. Setting the `direction` to `"previous"` will move focus to the most recently used pane.

Command name: `moveFocus`

Default IDs:

JSON

```
{ "command": { "action": "moveFocus", "direction": "down" }, "id":
"Terminal.MoveFocusDown" },
{ "command": { "action": "moveFocus", "direction": "left" }, "id":
"Terminal.MoveFocusLeft" },
{ "command": { "action": "moveFocus", "direction": "right" }, "id":
"Terminal.MoveFocusRight" },
{ "command": { "action": "moveFocus", "direction": "up" }, "id":
"Terminal.MoveFocusUp" },
{ "command": { "action": "moveFocus", "direction": "previous" }, "id":
"Terminal.MoveFocusPrevious" }
```

Default bindings:

JSON

```
{ "keys": "alt+down", "id": "Terminal.MoveFocusDown" },
{ "keys": "alt+left", "id": "Terminal.MoveFocusLeft" },
{ "keys": "alt+right", "id": "Terminal.MoveFocusRight" },
```

```
{ "keys": "alt+up", "id": "Terminal.MoveFocusUp" },
{ "keys": "ctrl+alt+left", "id": "Terminal.MoveFocusPrevious" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`direction`	Required	`"left"`, `"right"`, `"up"`, `"down"`, `"previous"`, `"previousInOrder"`, `"nextInOrder"`, `"first"`, `"parent"`, `"child"`	Direction in which the focus will move.

Accepted `direction` values

- `up`, `down`, `left`, or `right` move focus in the given direction.
- `first` moves focus to the first leaf pane in the tree.
- `previous` moves the focus to the most recently used pane before the current pane.
- `nextInOrder`, `previousInOrder` moves the focus to the next or previous pane in order of creation.
- `parent` moves the focus to select the parent pane of the current pane. This enables the user to select multiple panes at once
- `child` moves the focus to the first child pane of this pane.

Move pane

Move the currently active pane to a different tab in the window.

Command name: `movePane`

Default IDs:

JSON

```
{ "command": { "action": "movePane", "index": 0 }, "id":
"Terminal.MovePaneToTab0" },
{ "command": { "action": "movePane", "index": 1 }, "id":
"Terminal.MovePaneToTab1" },
{ "command": { "action": "movePane", "index": 2 }, "id":
"Terminal.MovePaneToTab2" },
{ "command": { "action": "movePane", "index": 3 }, "id":
"Terminal.MovePaneToTab3" },
{ "command": { "action": "movePane", "index": 4 }, "id":
"Terminal.MovePaneToTab4" },
{ "command": { "action": "movePane", "index": 5 }, "id":
```

```
"Terminal.MovePaneToTab5" },
{ "command": { "action": "movePane", "index": 6 }, "id":
"Terminal.MovePaneToTab6" },
{ "command": { "action": "movePane", "index": 7 }, "id":
"Terminal.MovePaneToTab7" },
{ "command": { "action": "movePane", "index": 8 }, "id":
"Terminal.MovePaneToTab8" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`index`	Required	number	The zero-indexed index of the tab to move to

Swap panes

Swap the position of two panes in a tab. This operates on the active pane, and a target pane, as designated by the `direction` parameter.

Command name: `swapPane`

Default IDs:

JSON

```
{ "command": { "action": "swapPane", "direction": "down" }, "id":
"Terminal.SwapPaneDown" },
{ "command": { "action": "swapPane", "direction": "left" }, "id":
"Terminal.SwapPaneLeft" },
{ "command": { "action": "swapPane", "direction": "right" }, "id":
"Terminal.SwapPaneRight" },
{ "command": { "action": "swapPane", "direction": "up" }, "id":
"Terminal.SwapPaneUp" },
{ "command": { "action": "swapPane", "direction": "previous"}, "id":
"Terminal.SwapPanePrevious" },
{ "command": { "action": "swapPane", "direction": "previousInOrder"}, "id":
"Terminal.SwapPanePreviousInOrder" },
{ "command": { "action": "swapPane", "direction": "nextInOrder"}, "id":
"Terminal.SwapPaneNextInOrder" },
{ "command": { "action": "swapPane", "direction": "first" }, "id":
"Terminal.SwapPaneFirst" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`direction`	Required	`"left"`, `"right"`, `"up"`, `"down"`, `"previous"`, `"previousInOrder"`, `"nextInOrder"`, `"first"`, `"parent"`, `"child"`	Direction in which the focus will move.

Accepted `direction` values (these are the same values as the `moveFocus` command)

- `up`, `down`, `left`, or `right`: Swap the active pane with the one in the given direction.
- `first`: Swap the active pane with the first leaf pane in the tree.
- `previous`: Swap the active pane with the most recently used pane before the current pane.
- `nextInOrder`, `previousInOrder`: Swap the active pane with the next or previous pane in order of creation.
- `parent`: Does nothing.
- `child`: Does nothing.

Zoom a pane

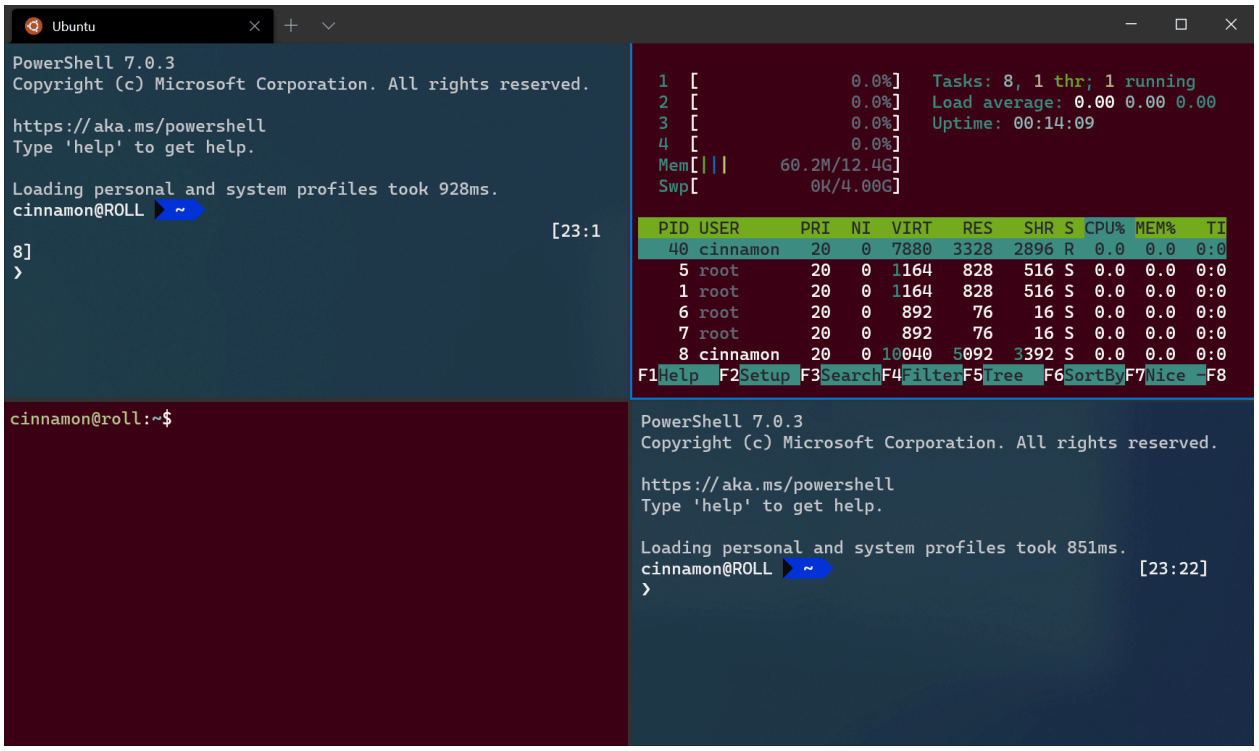
This expands the focused pane to fill the entire contents of the window.

Command name: `togglePaneZoom`

Default ID:

JSON

```
{ "command": "togglePaneZoom", "id": "Terminal.TogglePaneZoom" }
```



Resize a pane

This changes the size of the active pane.

Command name: `resizePane`

Default IDs:

JSON

```
{ "command": { "action": "resizePane", "direction": "down" }, "id":
"Terminal.ResizePaneDown" },
{ "command": { "action": "resizePane", "direction": "left" }, "id":
"Terminal.ResizePaneLeft" },
{ "command": { "action": "resizePane", "direction": "right" }, "id":
"Terminal.ResizePaneRight" },
{ "command": { "action": "resizePane", "direction": "up" }, "id":
"Terminal.ResizePaneUp" }
```

Default bindings:

JSON

```
{ "keys": "alt+shift+down", "id": "Terminal.ResizePaneDown" },
{ "keys": "alt+shift+left", "id": "Terminal.ResizePaneLeft" },
{ "keys": "alt+shift+right", "id": "Terminal.ResizePaneRight" },
{ "keys": "alt+shift+up", "id": "Terminal.ResizePaneUp" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`direction`	Required	`"left"`, `"right"`, `"up"`, `"down"`	Direction in which the pane will be resized.

Mark a pane as read-only

You can mark a pane as read-only, which will prevent input from going into the text buffer. If you attempt to close or input text into a read-only pane, the terminal will display a popup warning instead.

Command name: `toggleReadOnlyMode`

Default ID:

JSON

```
{ "command": "toggleReadOnlyMode", "id": "Terminal.ToggleReadOnlyMode" }
```

You can enable read-only mode on a pane. This works similarly to toggling, however, will not switch state if triggered again.

Command name: `enableReadOnlyMode`

Default ID:

JSON

```
{ "command": "enableReadOnlyMode", "id": "Terminal.EnableReadOnlyMode" }
```

You can disable read-only mode on a pane. This works similarly to toggling, however, will not switch state if triggered again.

Command name: `disableReadOnlyMode`

Default ID:

JSON

```
{ "command": "disableReadOnlyMode", "id": "Terminal.DisableReadOnlyMode" }
```

Restart a pane

This command will manually restart the `commandline` in the active pane. This is especially useful for scenarios like `ssh`, where you might want to restart a connection without closing the pane.

Note that this will terminate the process in the pane, if it is currently running.

Command name: `restartConnection`

Default ID:

JSON

```
{ "command": "restartConnection", "id": "Terminal.RestartConnection" }
```

Clipboard integration commands

Copy

This copies the selected terminal content to your clipboard. If no selection exists, the key chord is sent directly to the terminal.

Command name: `copy`

Default ID:

JSON

```
{ "command": { "action": "copy", "singleLine": false }, "id":
"Terminal.CopyToClipboard" }
```

Default bindings:

JSON

```
{ "keys": "ctrl+c", "id": "Terminal.CopyToClipboard" },
{ "keys": "ctrl+shift+c", "id": "Terminal.CopyToClipboard" },
{ "keys": "ctrl+insert", "id": "Terminal.CopyToClipboard" },
{ "keys": "enter", "id": "Terminal.CopyToClipboard" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`singleLine`	Optional	`true`, `false`	When `true`, the copied content will be copied as a single line. When `false`, newlines persist from the selected text.
`copyFormatting`	Optional	`true`, `false`, `"all"`, `"none"`, `"html"`, `"rtf"`	When `true`, the color and font formatting of the selected text is also copied to your clipboard. When `false`, only plain text is copied to your clipboard. You can also specify which formats you would like to copy. When `null`, the global `"copyFormatting"` behavior is inherited.

Paste

This inserts the content that was copied onto the clipboard.

Command name: `paste`

Default ID:

JSON

```
{ "command": "paste", "id": "Terminal.PasteFromClipboard" }
```

Default bindings:

JSON

```
{ "keys": "ctrl+v", "id": "Terminal.PasteFromClipboard" },
{ "keys": "ctrl+shift+v", "id": "Terminal.PasteFromClipboard" },
{ "keys": "shift+insert", "id": "Terminal.PasteFromClipboard" }
```

Expand selection to word

If a selection exists, this expands the selection to fully encompass any words partially selected.

Command name: `expandSelectionToWord`

Default ID:

JSON

```
{ "command": "expandSelectionToWord", "id": "Terminal.ExpandSelectionToWord"
}
```

Select all

This selects all of the content in the text buffer.

Command name: `selectAll`

Default ID:

JSON

```
{ "command": "selectAll", "id": "Terminal.SelectAll" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+a", "id": "Terminal.SelectAll" }
```

Mark mode

This toggles mark mode. Mark mode is a mode where you can use the keyboard to create a selection at the cursor's position in the terminal.

Command name: `markMode`

Default ID:

JSON

```
{ "command": "markMode", "id": "Terminal.ToggleMarkMode" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+m", "id": "Terminal.ToggleMarkMode" }
```

Switch selection marker

When modifying a selection using the keyboard, you are moving one end of the selection around. You can use this action to switch to the other selection marker.

Command name: `switchSelectionEndpoint`

Default ID:

JSON

```
{ "command": "switchSelectionEndpoint", "id":
"Terminal.SwitchSelectionEndpoint" },
```

Toggle block selection

Makes the existing selection a block selection, meaning that the selected area is a rectangle, as opposed to wrapping to the beginning and end of each line.

Command name: `toggleBlockSelection`

Default ID:

JSON

```
{ "command": "toggleBlockSelection", "id": "Terminal.ToggleBlockSelection"
},
```

Scrollback commands

Scroll up

This scrolls the screen up by the number of rows defined by `"rowsToScroll"`. If `"rowsToScroll"` is not provided, it will scroll up the amount defined by the system default, which is the same amount as mouse scrolling.

Command name: `scrollUp`

Default ID:

JSON

```
{ "command": "scrollUp", "id": "Terminal.ScrollUp" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+up", "id": "Terminal.ScrollUp" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`rowsToScroll`	Optional	Integer	The number of rows to scroll.

Scroll down

This scrolls the screen down by the number of rows defined by `"rowsToScroll"`. If `"rowsToScroll"` is not provided, it will scroll down the amount defined by the system default, which is the same amount as mouse scrolling.

Command name: `scrollDown`

Default ID:

JSON

```
{ "command": "scrollDown", "id": "Terminal.ScrollDown" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+down", "id": "Terminal.ScrollDown" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`rowsToScroll`	Optional	Integer	The number of rows to scroll.

Scroll up a whole page

This scrolls the screen up by a whole page, which is the height of the window.

Command name: `scrollUpPage`

Default ID:

JSON

```
{ "command": "scrollUpPage", "id": "Terminal.ScrollUpPage" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+pgup", "id": "Terminal.ScrollUpPage" }
```

Scroll down a whole page

This scrolls the screen down by a whole page, which is the height of the window.

Command name: `scrollDownPage`

Default ID:

JSON

```
{ "command": "scrollDownPage", "id": "Terminal.ScrollDownPage" }
```

Default binding:

JSON

```
{ "keys": "ctrl+shift+pgdn", "id": "Terminal.ScrollDownPage" }
```

Scroll to the earliest history

This scrolls the screen up to the top of the input buffer.

Command name: `scrollToTop`

Default ID:

```JSON
{ "command": "scrollToTop", "id": "Terminal.ScrollToTop" }
```

Default binding:

```JSON
{ "keys": "ctrl+shift+home", "id": "Terminal.ScrollToTop" }
```

Scroll to the latest history

This scrolls the screen down to the bottom of the input buffer.

Command name: `scrollToBottom`

Default ID:

```JSON
{ "command": "scrollToBottom", "id": "Terminal.ScrollToBottom" }
```

Default binding:

```JSON
{ "keys": "ctrl+shift+end", "id": "Terminal.ScrollToBottom" }
```

Clear buffer

This action can be used to manually clear the terminal buffer. This is useful for scenarios where you're not sitting at a command-line shell prompt and can't easily run `Clear-Host`/`cls`/`clear`.

Command name: `clearBuffer`

Default ID:

JSON

```
{ "command": { "action": "clearBuffer", "clear": "all" }, "id":
"Terminal.ClearBuffer" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`clear`	Optional	`"screen"`, `"scrollback"`, `"all"`	What part of the screen to clear. • `"screen"`: Clear the terminal viewport content. Leaves the scrollback untouched. Moves the cursor row to the top of the viewport (unmodified). • `"scrollback"`: Clear the scrollback. Leaves the viewport untouched. • `"all"` (*default*): Clear the scrollback and the visible viewport. Moves the cursor row to the top of the viewport.

Visual adjustment commands

Adjust font size

This changes the text size by a specified point amount.

Command name: `adjustFontSize`

Default IDs:

JSON

```
{ "command": { "action": "adjustFontSize", "delta": 1 }, "id":
"Terminal.IncreaseFontSize" },
{ "command": { "action": "adjustFontSize", "delta": -1 }, "id":
"Terminal.DecreaseFontSize" }
```

Default bindings:

```JSON
{ "keys": "ctrl+plus", "id": "Terminal.IncreaseFontSize" },
{ "keys": "ctrl+minus", "id": "Terminal.DecreaseFontSize" },
{ "keys": "ctrl+numpad_plus", "id": "Terminal.IncreaseFontSize" },
{ "keys": "ctrl+numpad_minus", "id": "Terminal.DecreaseFontSize" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`delta`	Required	Integer	Amount of size change per command invocation.

Reset font size

This resets the text size to the default value.

Command name: `resetFontSize`

Default ID:

```JSON
{ "command": "resetFontSize", "id": "Terminal.ResetFontSize" }
```

Default bindings:

```JSON
{ "keys": "ctrl+0", "id": "Terminal.ResetFontSize" },
{ "keys": "ctrl+numpad_0", "id": "Terminal.ResetFontSize" }
```

Adjust opacity

This changes the opacity of the window. If `relative` is set to true, it will adjust the opacity relative to the current opacity. Otherwise, it will set the opacity directly to the given `opacity`

Command name: `adjustOpacity`

Default bindings:

JSON

```
{ "command": { "action": "adjustOpacity", "relative": false, "opacity": 0 }
},
{ "command": { "action": "adjustOpacity", "relative": false, "opacity": 25 }
},
{ "command": { "action": "adjustOpacity", "relative": false, "opacity": 50 }
},
{ "command": { "action": "adjustOpacity", "relative": false, "opacity": 75 }
},
{ "command": { "action": "adjustOpacity", "relative": false, "opacity": 100
} }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`opacity`	Optional	Integer	How opaque the terminal should become or how much the opacity should be changed by, depending on the value of `relative`
`relative`	Optional	Boolean	If true, then adjust the current opacity by the given `opacity` parameter. If false, set the opacity to exactly that value.

Toggle pixel shader effects

This toggles any pixel shader effects enabled in the terminal. If the user specified a valid shader with `experimental.pixelShaderPath`, this action will toggle that shader on/off. This will also toggle the "retro terminal effect", which is enabled with the profile setting `experimental.retroTerminalEffect`.

Command name: `toggleShaderEffects`

Default ID:

JSON

```
{ "command": "toggleShaderEffects", "id": "Terminal.ToggleShaderEffects" }
```

⊗ **Caution**

The `toggleRetroEffect` action is no longer available in versions 1.6 and later. It is recommended that you use `toggleShaderEffects` instead.

Set the color scheme

Changes the active color scheme.

Command name: `setColorScheme`

Parameters

Expand table

Name	Necessity	Accepts	Description
`colorScheme`	Required	String	The `name` of the color scheme to apply.

Example declaration:

JSON

```
{ "command": { "action": "setColorScheme", "colorScheme": "Campbell" },
"id": "User.SetSchemeToCampbell" }
```

Add scroll mark

Adds a scroll mark to the text buffer. If there's a selection, the mark is placed at the selection, otherwise it's placed at the cursor row.

Command name: `addMark`

Parameters

Expand table

Name	Necessity	Accepts	Description
`color`	Optional	String, in hex format: `"#rgb"` or `"#rrggbb"`	The color of the mark.

Example declaration:

JSON

```
{ "command": { "action": "addMark", "color": "#ff00ff" }, "id":
"User.AddMark" }
```

> **Important**
>
> This action became stable in v1.21. Before that version, it was only available in Windows Terminal Preview

Scroll to mark

Scrolls to the scroll mark in the given direction. For more info, see Scroll marks and Shell Integration.

Command name: `scrollToMark`

Parameters

Expand table

Name	Necessity	Accepts	Description
`direction`	Required	`"first"`, `"previous"`, `"next"`, `"last"`	The direction in which to scroll.

Example declaration:

JSON

```
{ "command": { "action": "scrollToMark", "direction": "previous" }, "id":
"User.ScrollToMark" }
```

> **Important**
>
> This action became stable in v1.21. Before that version, it was only available in Windows Terminal Preview

Clear mark

Clears scroll mark at the current position, either at a selection if there is one or at the cursor position. This is an experimental feature, and its continued existence is not guaranteed.

Command name: `clearMark`

Example declaration:

JSON

```
{ "command": { "action": "clearMark" }, "id": "User.ClearMark" }
```

> **Important**
>
> This action became stable in v1.21. Before that version, it was only available in Windows Terminal Preview

Clear all marks

Clears all scroll marks in the text buffer. This is an experimental feature, and its continued existence is not guaranteed.

Command name: `clearAllMarks`

Example declaration:

JSON

```
{ "command": { "action": "clearAllMarks" }, "id": "User.ClearAllMarks" }
```

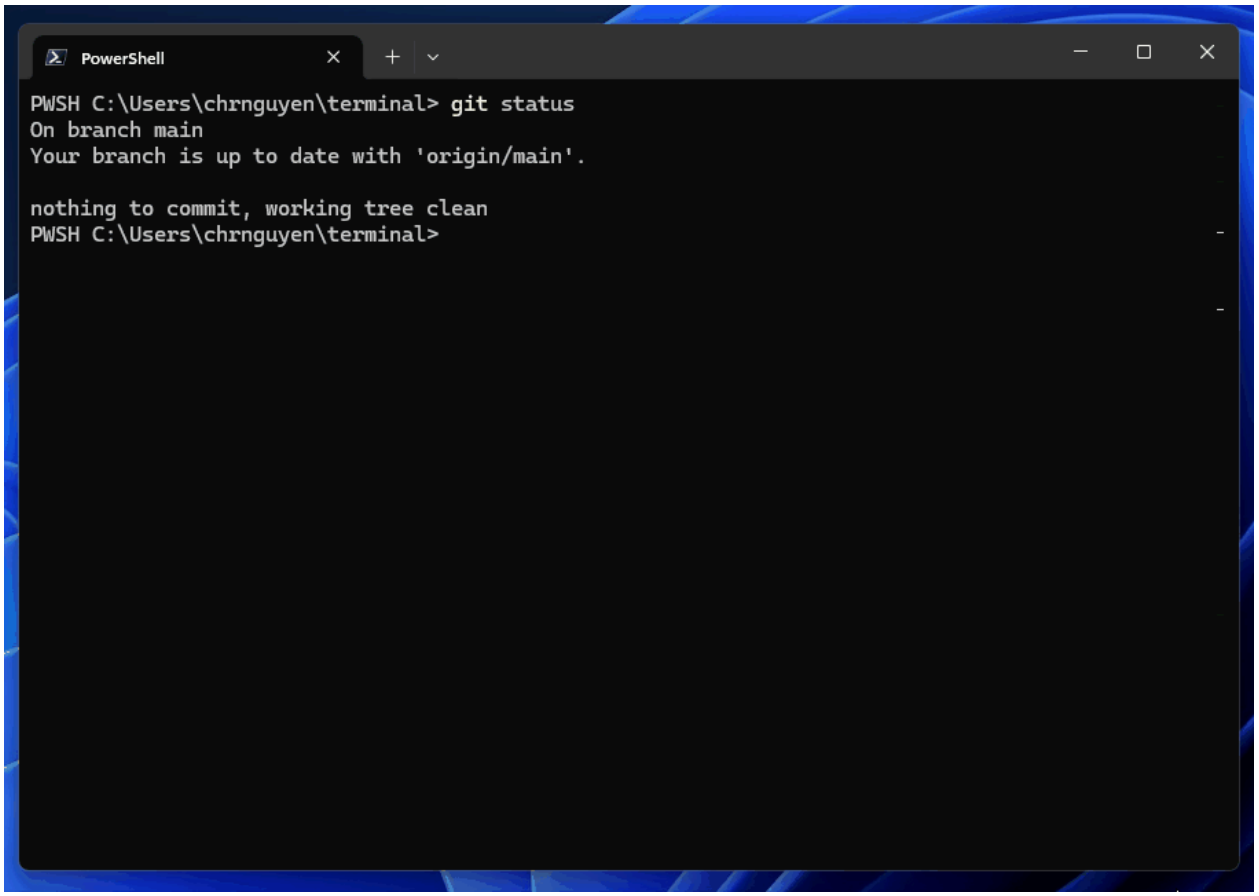
> **Important**
>
> This action became stable in v1.21. Before that version, it was only available in Windows Terminal Preview

---

Suggestions

Open suggestions menu

This allows the user to open the suggestions menu. The entries in the suggestions menu are controlled by the `source` property. The suggestions menu behaves much like the command palette. Typing in the text box will filter the results to only show entries that match the text. Pressing `enter` will execute the selected entry. Pressing `esc` will close the menu.



Command name: `showSuggestions`

Parameters

Expand table

Name	Necessity	Accepts	Description
`source`	Required	any number of `"recentCommands"`, `"tasks"`, or `"all"`	Which suggestion sources to use to populate this menu. See below for a description of each.
`useCommandline`	Optional	Boolean	If shell integration is enabled, and this is `true`, the suggestions menu will be pre-populated with the contents of the current commandline. Defaults to `true`

Suggestion sources

The following suggestion sources are supported:

- `"recentCommands"`: This will populate the suggestions menu with the most recently used commands. These are powered by shell integration, so they will only be available if you have your shell configured to support shell integration. See Shell Integration for more information.
- `"tasks"`: This will populate the suggestions menu with all of the `sendInput` actions from your settings.
- `"all"`: Use all suggestion sources.

These values can be used by themselves as a string parameter value, or combined as an array. For example:

JSON

```
{ "command": { "action": "showSuggestions", "source": ["recentCommands",
"tasks"] } },
{ "command": { "action": "showSuggestions", "source": "all" } },
{ "command": { "action": "showSuggestions", "source": "recentCommands" } },
```

In the above example, the first two commands will open the suggestions menu with both recent commands and tasks. The third command will open the suggestions menu with only recent commands.

> **ⓘ Important**
>
> This feature is only available in Windows Terminal Preview.

Buffer exporting

Export buffer

This allows the user to export the text of the buffer to a file. If the file doesn't exist, it will be created. If the file already exists, its contents will be replaced with the Terminal buffer text.

Command name: `exportBuffer`

Default ID:

```JSON
{ "command": { "action": "exportBuffer" }, "id": "Terminal.ExportBuffer" }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`path`	Optional	String	If provided, then the Terminal will export the buffer contents to the given file. Otherwise, the terminal will open a file picker to choose the file to export to.

Global commands

Global summon

This is a special action that works globally in the OS, rather than only in the context of the terminal window. When pressed, this action will summon the terminal window. Which window is summoned, where the window is summoned to, and how the window behaves when summoning it, is controlled by the properties on this action.

Notes

- Any keys bound to `globalSummon` actions in the terminal will not work in other applications while the terminal is running - they will always focus the terminal window.
- If another running application already registered for the given `keys` using the `RegisterHotKey` API, the terminal will be unable to listen for those key strokes.
- Elevated and unelevated instances of the terminal will not be able to both register for the same keys. The same applies to both Preview and Stable versions of the terminal - the first one to be launched will always win.
- These key strokes will only work when an instance of the terminal is already running. To launch the terminal automatically on login, see startOnUserLogin.

Command name: `globalSummon`

Default binding:

This command is not currently bound in the default settings.

JSON

```
{ "command": { "action": "globalSummon" } }
```

Parameters

Expand table

Name	Necessity	Accepts	Description
`desktop`	Optional	`any`, `toCurrent`, `onCurrent`	This controls how the terminal should interact with virtual desktops. • `"any"`: Leave the window on whichever desktop it's already on - will switch to that desktop as the window is activated. • `"toCurrent"` (*default*): Move the window to the current virtual desktop. • `"onCurrent"`: Only summon the window if it's already on the current virtual desktop.
`monitor`	Optional	`any`, `toCurrent`, `toMouse`	This controls the monitor that the window will be summoned from/to. • `"any"`: Summon the most recently used window, regardless of which monitor it's currently on. • `"toCurrent"`: Summon the most recently used window to the monitor with the current foreground window. • `"toMouse"` (*default*): Summon the most recently used window to the monitor where the mouse cursor is.
`name`	Optional	String	When omitted (*default*), use `monitor` and `desktop` to find the appropriate most-recently-used window to summon. When provided, summon the window whose name or ID matches the given `name` value. If no such window exists, then create a new window with that name.

Name	Necessity	Accepts	Description
`dropdownDuration`	Optional	Integer	Defaults to `0`. When provided with a positive number, "slide" the window in from the top of the screen using an animation that lasts `dropdownDuration` milliseconds. `200` is a reasonable value for this setting.
`toggleVisibility`	Optional	`true`, `false`	Defaults to `true`. When `true`, pressing the assigned keys for this action will dismiss (minimize) the window when the window is currently the foreground window. When `false`, pressing the assigned keys will only ever bring the window to the foreground.

When `name` is provided *with* `monitor` or `desktop`, `name` behaves in the following ways:

- `desktop`
  - `"any"`: Go to the desktop the given window is already on.
  - `"toCurrent"`: If the window is on another virtual desktop, then move it to the currently active one.
  - `"onCurrent"`: If the window is on another virtual desktop, then move it to the currently active one.
- `monitor`
  - `"any"`: Leave the window on the monitor it is already on.
  - `"toCurrent"`: If the window is on another monitor, then move it to the monitor with the current foreground window.
  - `"toMouse"`: If the window is on another monitor, then move it to the monitor with the mouse cursor on it.

The `desktop` and `monitor` properties can be combined in the following ways:

Expand table

Combinations	`"desktop": "any"`	`"desktop": "toCurrent"`	`"desktop": "onCurrent"`	Not included
`"monitor": "any"`	Go to the desktop the window is on (leave position alone)	Move the window to this desktop (leave position alone)	If there isn't one on this desktop: • Create a new one in the default position Else:	Summon the MRU window

Combinations	`"desktop": "any"`	`"desktop": "toCurrent"`	`"desktop": "onCurrent"`	Not included
			• Activate the one on this desktop (don't move it)	
`"monitor": "toCurrent"`	Go to the desktop the window is on, move to the monitor with the foreground window	Move the window to this desktop, move to the monitor with the foreground window	If there isn't one on this desktop: • Create a new one Else: • Activate the one on this desktop, move to the monitor with the foreground window	Summon the MRU window TO the monitor with the foreground window
`"monitor": "toMouse"`	Go to the desktop the window is on, move to the monitor with the mouse	Move the window to this desktop, move to the monitor with the mouse	If there isn't one on this desktop: • Create a new one Else: • Activate the one on this desktop, move to the monitor with the mouse	Summon the MRU window TO the monitor with the mouse
Not included	Leave where it is	Move to current desktop	On current desktop only	N/A

Examples

jsonc

```
// Summon the most recently used (MRU) window, to the current virtual
desktop,
// to the monitor the mouse cursor is on, without an animation. If the
window is
// already in the foreground, then minimize it.
{ "command": { "action": "globalSummon" }, "id": "User.MyGlobalSummon" },
```

```
// Summon the MRU window, by going to the virtual desktop the window is
// currently on. Move the window to the monitor the mouse is on.
{ "command": { "action": "globalSummon", "desktop": "any" }, "id":
"User.MyGlobalSummonAnyDesktop" },

// Summon the MRU window to the current desktop, leaving the position of the
window untouched.
{ "command": { "action": "globalSummon", "monitor": "any" }, "id":
"User.MyGlobalSummonAnyMonitor" },

// Summon the MRU window, by going to the virtual desktop the window is
// currently on, leaving the position of the window untouched.
{ "command": { "action": "globalSummon", "desktop": "any", "monitor": "any"
}, "id": "User.MyGlobalSummonAnywhere" },

// Summon the MRU window with a dropdown duration of 200ms.
{ "command": { "action": "globalSummon", "dropdownDuration": 200 }, "id":
"User.MyGlobalSummonDrop" },

// Summon the MRU window. If the window is already in the foreground, do
nothing.
{ "command": { "action": "globalSummon", "toggleVisibility": false }, "id":
"User.MyGlobalSummonIfNotVisible" },

// Summon the window named "_quake". If no window with that name exists,
then create a new window.
{ "command": { "action": "globalSummon", "name": "_quake" }, "id":
"User.MyGlobalSummonQuake" }
```

Open the quake mode window

This action is a special variation of the globalSummon action. It specifically summons the quake window. It is a shorthand for the following `globalSummon` action:

JSON

```
{
"id": "User.MySummonQuake",
"command": {
"action": "globalSummon",
"name": "_quake",
"dropdownDuration": 200,
"toggleVisibility": true,
"monitor": "toMouse",
"desktop": "toCurrent"
}
}
```

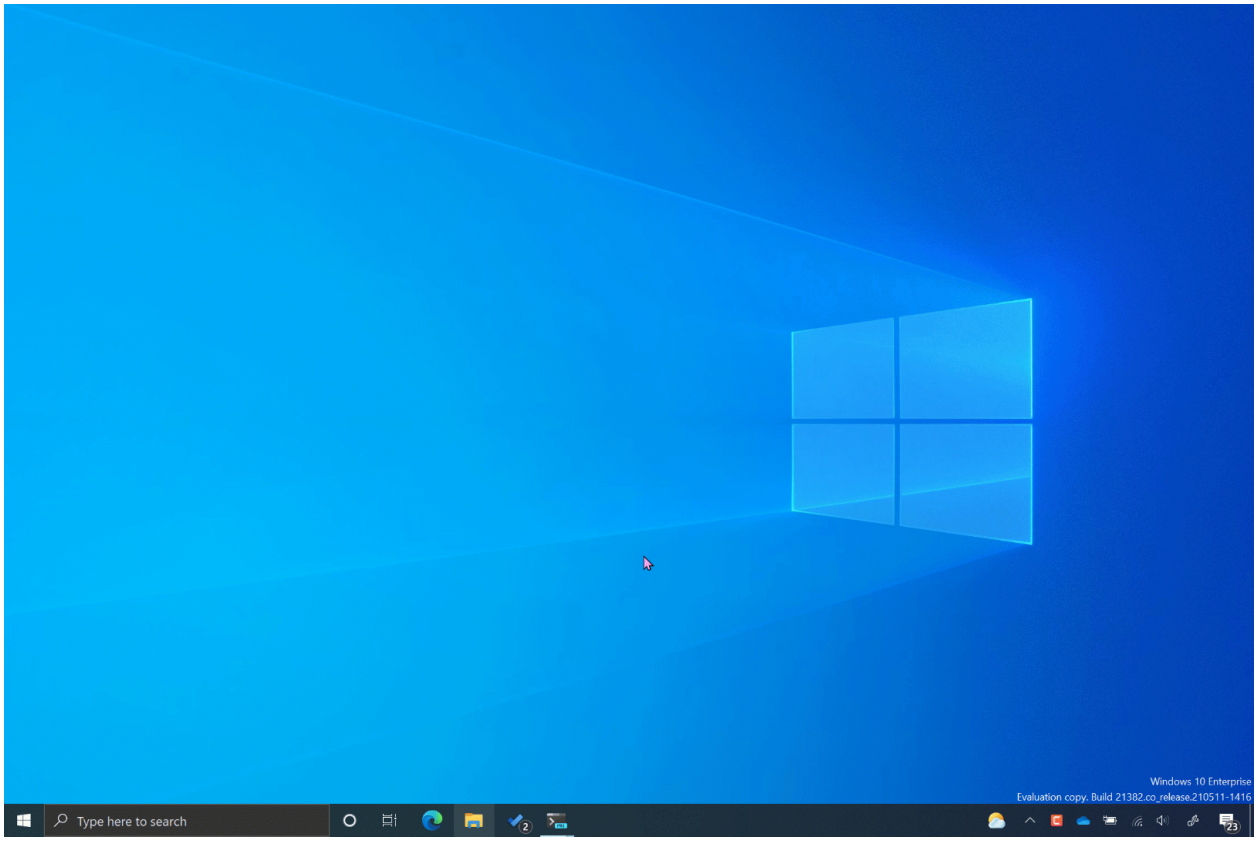
If you'd like to change the behavior of the `quakeMode` action, we recommended creating a new `globalSummon` entry in `actions` with the settings you prefer.

Command name: `quakeMode`

Default ID:

JSON

```
{ "command": "quakeMode", "id": "Terminal.QuakeMode" }
```



Run multiple actions

This action allows the user to bind multiple sequential actions to one command. These actions do not support IDs.

Command name: `multipleActions`

Parameters

Expand table

Name	Necessity	Accepts	Description
`actions`	Required	Array of Actions	The list of `action` to run.

Example

jsonc

```
{ "name": "Create My Layout", "command": {
    "action": "multipleActions",
    "actions": [
        // Create a new tab with 3 panes
        { "action": "newTab", "tabTitle": "Work", "colorScheme": "One Half
Dark" },
        { "action": "splitPane", "split": "vertical", "profile": "Windows
PowerShell", "tabTitle": "Work", "colorScheme": "Campbell Powershell", },
        { "action": "splitPane", "split": "horizontal", "profile": "Windows
PowerShell", "tabTitle": "Work", "colorScheme": "Campbell Powershell", },

        // Create a second tab
        { "action": "newTab", "tabTitle": "Misc"},

        // Go back to the first tab and zoom the first pane
        { "action": "prevTab", "tabSwitcherMode": "disabled" },
        { "action": "moveFocus", "direction": "first"},
        "togglePaneZoom"
        ]
}}
```

Unbind keys (disable keybindings)

You can disable keybindings or "unbind" the associated keys from any command. This may be necessary when using underlying terminal applications (such as VIM). The unbound key will pass to the underlying terminal.

Command name: `unbound`

Example using unbound:

For example, to unbind the shortcut keys `Alt+Shift+-`" and `Alt+Shift+=`", include these commands in the `actions` section of your settings.json file.

JSON

```
{
    "keybindings": [
        { "id": "unbound", "keys": "alt+shift+-" },
        { "id": "unbound", "keys": "alt+shift+=" }
    ]
}
```

Example using null:

You can also unbind a keystroke that is bound by default to an action by setting `"id"` to `null`. This will also allow the keystroke to associate with the command line application setting instead of performing the default action.

JSON

```
{
   "id" : null, "keys" : ["ctrl+v"]
}
```

Use-case scenario:

Windows Terminal uses the shortcut key binding `Ctrl+V` as the paste command. When working with a WSL command line, you may want to use a Linux application such as Vim to edit files. However, Vim relies on the `Ctrl+V` key binding to use blockwise Visual mode. This key binding will be blocked, with the Windows Terminal paste command taking priority, unless the `unbound` setting is adjusted in your settings.json file so that the key binding will associate with the Vim command line app, rather than with the Windows Terminal binding.

General profile settings in Windows Terminal

Article • 05/29/2024

The settings listed below are specific to each unique profile. If you'd like a setting to apply to all of your profiles, you can add it to the `defaults` section above the list of profiles in your settings.json file.

JSON

```
"defaults":
{
    // SETTINGS TO APPLY TO ALL PROFILES
},
"list":
[
    // PROFILE OBJECTS
]
```

Profile Ordering

The ordering of profiles in the `"list"` determines the profile index numbering. This is used to map to the launch key combo, such as `Ctrl+Shift+1`. To change the profile index number, simply cut/paste the profile objects above or below each other. The first in the `"list"` will map to index 1, hence, it will be assigned to the key combo, `Ctrl+Shift+1`.

Name

This is the name of the profile that will be displayed in the dropdown menu. This value is also used as the "title" to pass to the shell on startup. Some shells (like `bash`) may choose to ignore this initial value, while others (`Command Prompt`, `PowerShell`) may use this value over the lifetime of the application. This "title" behavior can be overridden by using `tabTitle`.

Property name: `name`

Necessity: Required

Accepts: String

Command line

This is the executable used in the profile.

Property name: `commandline`

Necessity: Optional

Accepts: Executable file name as a string

Default value: `"cmd.exe"`

Example: To run a batch file each time cmd.exe is run, set this value to "cmd.exe /k path\to\script.bat"

Starting directory

This is the directory the shell starts in when it is loaded.

Property name: `startingDirectory`

Necessity: Optional

Accepts: Folder location as a string

Default value: `"%USERPROFILE%"`

NOTE: When starting directory is not defined, the default value will be set to `"%USERPROFILE%"` (the path relative to your user settings, for example this may be `C:\Users\<your username>`). However, if the starting directory is explicitly set to `null`, then you will get different results depending on where you launch Terminal.

Example: Start the PowerShell profile in the *GitHubRepos* folder of your *Documents* directory by finding the powershell.exe profile and adding `"startingDirectory": "%USERPROFILE%/Documents/GitHubRepos",`

Example with WSL: When setting the starting directory for a Linux distribution installed via WSL, use the format: `"startingDirectory": "\\\\wsl$\\DISTRO NAME\\home\\USERNAME"`, replacing with the placeholders with the proper names of your distribution. For example, `"startingDirectory": "\\\\wsl$\\Ubuntu-20.04\\home\\user1"`. If you are using the Windows Terminal Settings UI, rather than the settings.json file, to declare this path, you can use the **Browse...** button to select your

starting directory or enter the WSL path as: `//wsl.localhost/DISTRO NAME/home/USERNAME`. For example, `//wsl.localhost/Ubuntu-20.04/home/user1`.

Default behavior: When the startingDirectory value is not specified, you will get different results depending on where you launch Terminal:

- If you run Windows Terminal from the Start menu: `%WINDIR%\System32`
- If you run wt.exe from the Start menu: `%WINDIR%\System32`
- If you run wt.exe from `Win+R`: `%USERPROFILE%`
- If you run wt.exe from the explorer address bar: whatever folder you were looking at.

> ⓘ **Note**
>
> Backslashes need to be escaped. For example, `C:\Users\USERNAME\Documents` should be entered as `C:\\Users\\USERNAME\\Documents`.

Icon

This sets the icon that displays within the tab, dropdown menu, jumplist, and tab switcher.

Property name: `icon`

Necessity: Optional

Accepts: File location as a string, or an emoji

Example: By placing the icon image `ubuntu.ico` in the folder located at `%LOCALAPPDATA%\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\RoamingState`, you can display the icon by adding this line to the profile in your settings.json: `"icon": "ms-appdata:///roaming/ubuntu.ico"`.

—

Tab title

If set, this will replace the `name` as the title to pass to the shell on startup. Some shells (like `bash`) may choose to ignore this initial value, while others (`Command Prompt`,

`PowerShell`) may use this value over the lifetime of the application. If you'd like to learn how to have the shell set your title, visit the tab title tutorial.

Property name: `tabTitle`

Necessity: Optional

Accepts: String

—

Automatically run as Administrator

If set, this profile will automatically open up in an "elevated" window (running as Administrator) by default. If you run this profile from an unelevated window, then a new elevated terminal window will be created to host this profile. If you launch this profile from an already elevated window, then it will open as a new tab.

When this property is set to `false`, opening this profile in an elevated window will not launch an *unelevated* window to host this profile. The profile will simply open in the elevated window, running as Administrator.

If you set this property in `profiles.defaults`, then *all* profiles will launch as Administrator by default, unless overridden by specifically setting this to false.

This property can be overridden in the newTab and splitPane actions, with the `elevate` property.

Elevated and unelevated tabs cannot exist in the same terminal window. For more details, please see the FAQ.

Property name: `elevate`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

—

Hide profile from dropdown

If `hidden` is set to `true`, the profile will not appear in the list of profiles. This can be used to hide default profiles and dynamically generated profiles, while leaving them in your settings file. To learn more about dynamic profiles, visit the Dynamic profiles page.

Property name: `hidden`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Appearance profile settings in Windows Terminal

Article • 03/11/2023

The settings listed below affect the visual settings of each profile separately. If you'd like a setting to apply to all of your profiles, you can add it to the `defaults` section above the list of profiles in your settings.json file.

JSON

```
"defaults":
{
    // SETTINGS TO APPLY TO ALL PROFILES
},
"list":
[
    // PROFILE OBJECTS
]
```

Text

Color scheme

This is the name of the color scheme used in the profile. Color schemes are defined in the `schemes` object. More detailed information can be found on the Color schemes page.

In addition to a single color scheme name, this property can accept a pair of color scheme names as follows:

JSON

```
"colorScheme":
{
    "light": "One Half Light",
    "dark": "One Half Dark",
},
```

When specified in this manner, the Terminal will automatically switch between the two given color schemes depending on the theme of the application. The Terminal will follow the theme.applicationTheme property of the Terminal's selected theme. If that `applicationTheme` is set to `system`, then this will instead use the color scheme matching the OS theme.

Property name: `colorScheme`

Necessity: Optional

Accepts: Name of color scheme as a string, or an object with a `light` and `dark` property

Default value: `"Campbell"`

Font

This is the structure within which the other font settings must be defined. An example of what this could look like in the JSON file is shown below.

Property name: `font`

Necessity: Optional

Font face

This is the name of the font face used in the profile. The terminal will try to fallback to Consolas if this can't be found or is invalid. To learn about the other variants of the default font, Cascadia Mono, visit the Cascadia Code page.

Property name: `face` (defined within the `font` object)

Necessity: Optional

Accepts: Font name as a string

Default value: `"Cascadia Mono"`

Font size

This sets the profile's font size in points.

Property name: `size` (defined within the `font` object)

Necessity: Optional

Accepts: Integer

Default value: `12`

Font weight

This sets the weight (lightness or heaviness of the strokes) for the profile's font.

Property name: `weight` (defined within the `font` object)

Necessity: Optional

Accepts: `"normal"`, `"thin"`, `"extra-light"`, `"light"`, `"semi-light"`, `"medium"`, `"semi-bold"`, `"bold"`, `"extra-bold"`, `"black"`, `"extra-black"`, or an integer corresponding to the numeric representation of the OpenType font weight

Default value: `"normal"`

Font example

JSON

```
"font": {
    "face": "Cascadia Mono",
    "size": 12,
    "weight": "normal"
}
```

> **Important**
>
> This `font` object is only available in Windows Terminal version 1.10+. Prior to that version, you should use the `fontFace`, `fontSize`, and `fontWeight` properties separately, like so:
>
> JSON
>
> ```
> "fontFace": "Cascadia Mono",
> "fontSize": 12,
> "fontWeight": "normal"
> ```

Font features

This sets the OpenType font features for the given font.

Property name: `features` (defined within the `font` object)

Necessity: Optional

Accepts: Feature properties in the format of: `"string": integer`

Example:

jsonc

```
// Enables ss01 and disables ligatures
"font": {
    "face": "Cascadia Code",
    "features": {
        "ss01": 1,
        "liga": 0
    }
}
```

Font axes

This sets the OpenType font axes for the given font.

Property name: `axes` (defined within the `font` object)

Necessity: Optional

Accepts: Axis properties in the format of: `"string": integer`

Example:

```jsonc
// Sets the font to italic
"font": {
    "face": "Cascadia Code",
    "axes": {
        "ital": 1
    }
}
```

Intense text formatting

This controls how "intense" text is formatted in the terminal. "Intense" text is text formatted with the escape sequence `\x1b[1m`.

Property name: `intenseTextStyle`

Necessity: Optional

Accepts: `"none"`, `"bold"`, `"bright"`, `"all"`

- `"all"`: render intense text as both **bold** and bright
- `"bold"`: render intense text as **bold**, but not bright
- `"bright"`: render intense text bright, but not bold
- `"none"`: the terminal won't do anything special for intense text

Default value: `"bright"`

Retro terminal effects

When this is set to `true`, the terminal will emulate a classic CRT display with scan lines and blurry text edges. This is an experimental feature and its continued existence is not guaranteed.

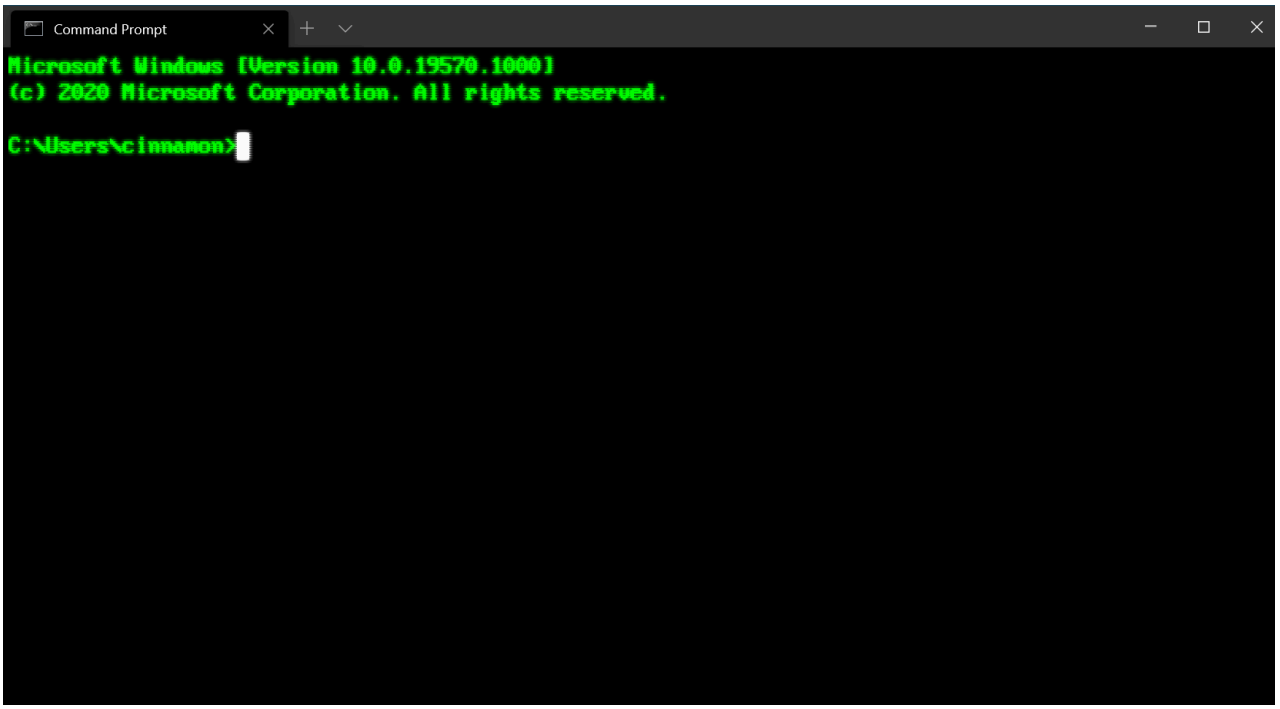
If `experimental.pixelShaderPath` is set, it will override this setting.

Property name: `experimental.retroTerminalEffect`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`



Configuration: Retro Command Prompt

Cursor

Cursor shape

This sets the cursor shape for the profile. The possible cursors are as follows: `"bar"` (▏), `"vintage"` (▃), `"underscore"` (_), `"filledBox"` (▉), `"emptyBox"` (▯), `"doubleUnderscore"` (‗)

Property name: `cursorShape`

Necessity: Optional

Accepts: `"bar"`, `"vintage"`, `"underscore"`, `"filledBox"`, `"emptyBox"`, `"doubleUnderscore"`

Default value: `"bar"`

Cursor height

This sets the percentage height of the cursor starting from the bottom. This will only work when `cursorShape` is set to `"vintage"`.

Property name: `cursorHeight`

Necessity: Optional

Accepts: Integer from 1-100

Background images and icons

Windows Terminal enables you to specify custom background images and icons using the settings UI menu or settings.json file for each of your command line profiles, allowing you to configure/brand/style each of your profiles independently from one another. To do so, specify your preferred `backgroundImage`, position it using `backgroundImageAlignment`, set its opacity with `backgroundImageOpacity`, and/or specify how your image will fill the available space using `backgroundImageStretchMode`.

For example:

JSON

```
    "backgroundImage": "C:\\Users\\username\\OneDrive\\WindowsTerminal\\bg-ubuntu-
256.png",
    "backgroundImageAlignment": "bottomRight",
    "backgroundImageOpacity": 0.1,
    "backgroundImageStretchMode": "none"
```

You can easily roam your collection of images and icons across all your machines by storing your icons and images in OneDrive (as shown above).

Background image path

This sets the file location of the image to draw over the window background. The background image can be a .jpg, .png, or .gif file. `"desktopWallpaper"` will set the background image to the desktop's wallpaper.

Property name: `backgroundImage`

Necessity: Optional

Accepts: File location as a string or `"desktopWallpaper"`

It is recommended that custom images and icons are stored in system-provided folders and referred to using the correct URI schemes. URI schemes provide a way to reference files independent of their physical paths (which may change in the future). The most useful URI schemes to remember when customizing background images and icons are:

Expand table

URI scheme	Corresponding physical path	Use / description
`ms-appdata:///Local/`	`%localappdata%\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\LocalState\`	Per-machine files
`ms-appdata:///Roaming/`	`%localappdata%\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\RoamingState\`	Common files

> ⚠ **Warning**
>
> Do not rely on file references using the ms-appx URI scheme (i.e. icons). These files are considered an internal implementation detail and may change name/location or may be omitted in the future.

Icons

Windows Terminal displays icons for each profile which the terminal generates for any built-in shells, for example: PowerShell Core, PowerShell, and any installed Linux/WSL distributions. Each profile refers to a stock icon via the ms-appx URI scheme. You can refer to you own custom icons by entering a path in your settings.json file:

JSON

```
"icon" : "C:\\Users\\username\\OneDrive\\WindowsTerminal\\icon-ubuntu-32.png",
```

Icons should be sized to 32x32px in an appropriate raster image format (e.g. .PNG, .GIF, or .ICO) to avoid having to scale your icons during runtime (causing a noticeable delay and loss of quality).

If no icon is specified for a command line you've installed, Windows Terminal will default to this glyph from the Segoe Fluent font:

Expand table

Glyph	Unicode point	Description
	e756	CommandPrompt

Background image stretch mode

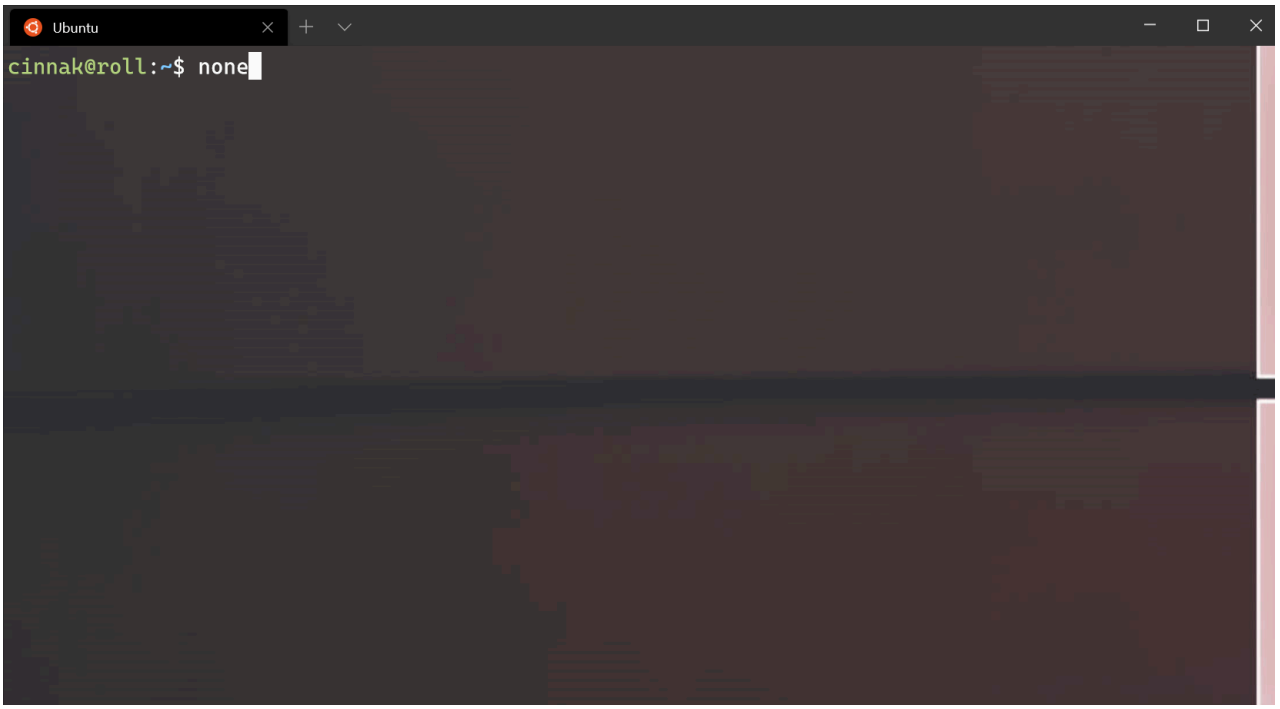
This sets how the background image is resized to fill the window.

Property name: `backgroundImageStretchMode`

Necessity: Optional

Accepts: `"none"`, `"fill"`, `"uniform"`, `"uniformToFill"`

Default value: `"uniformToFill"`



Background image source

Background image alignment

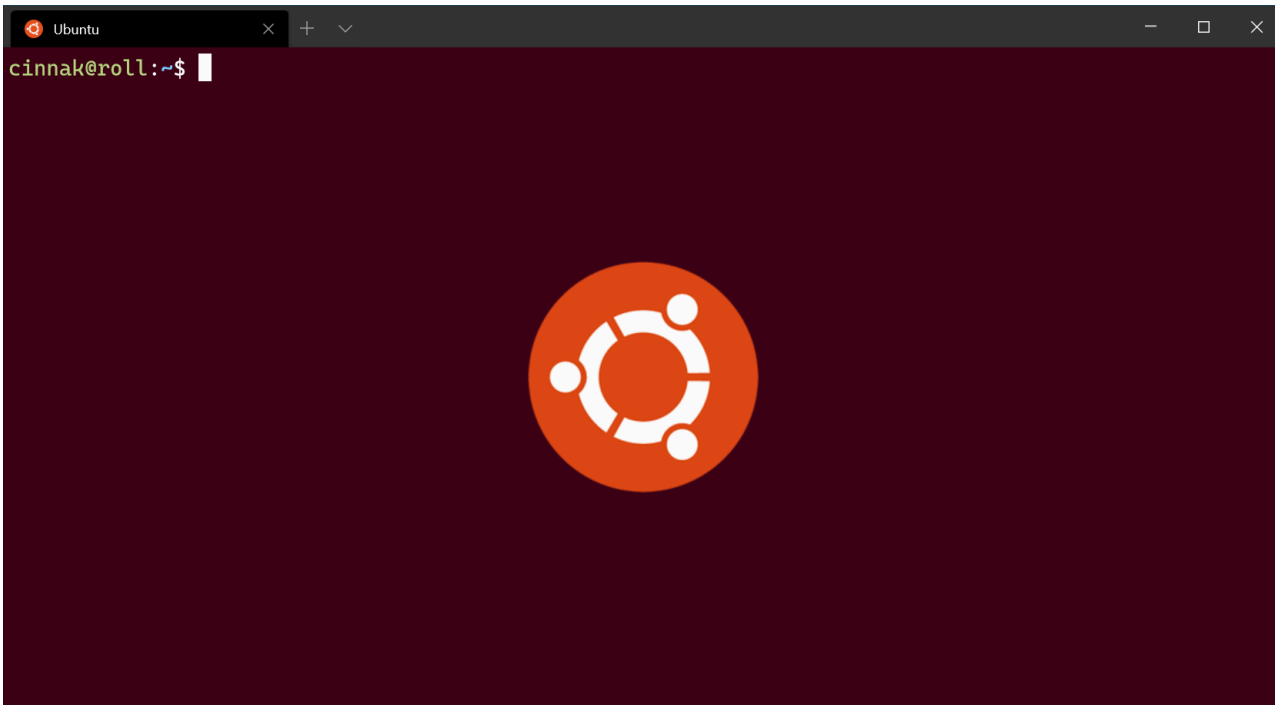
This sets how the background image aligns to the boundaries of the window.

Property name: `backgroundImageAlignment`

Necessity: Optional

Accepts: `"center"`, `"left"`, `"top"`, `"right"`, `"bottom"`, `"topLeft"`, `"topRight"`, `"bottomLeft"`, `"bottomRight"`

Default value: `"center"`



Background image source

Background image opacity

This sets the transparency of the background image.

Property name: `backgroundImageOpacity`

Necessity: Optional

Accepts: Number as a floating point value from 0-1

Default value: `1.0`

Transparency

Opacity

This sets the transparency of the window for the profile. This accepts an integer value from 0-100, representing a "percent opaque". `100` is "fully opaque", `50` is semi-transparent, and `0` is fully transparent.

When `useAcrylic` is set to `true`, the window will use the acrylic material to create a blurred background for the terminal. When `useAcrylic` is set to false, the terminal will use an unblurred opacity.

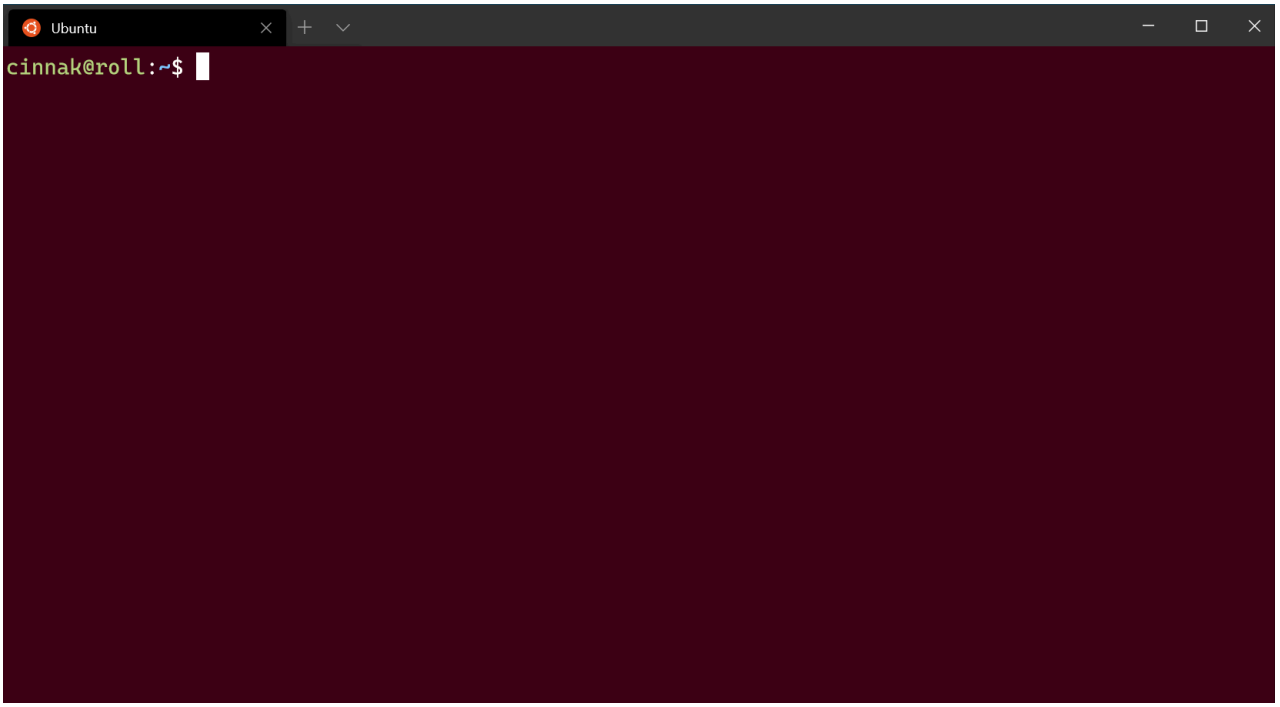
Users can choose different opacity values for focused and unfocused windows allowing for customization.

Property name: `opacity`

Necessity: Optional

Accepts: Number as an integer value from 0-100

Default value: `100`



> ⓘ **Important**
>
> Prior to Windows Terminal version 1.12, this setting was `acrylicOpacity`, was a float that accepted 0.0-1.0 which defaulted to 0.5, and the opacity would only apply if `useAcrylic` was set to true. On 1.12+, `acrylicOpacity` will gracefully continue to work as the equivalent `opacity` value.

> ⓘ **Important**
>
> Unblurred opacity (`"useAcrylic": false`) only works on Windows 11.

> ⓘ **Important**
>
> When Mica is enabled in the **theme settings**, Mica will appear underneath the Terminal contents when the `opacity` of the Terminal is set to a value <100.

Enable acrylic

When this is set to `true`, the window will have an acrylic background. When it's set to `false`, the window will have a plain, untextured background. Depending on the `Enable Unfocused Acrylic` global setting the transparency applies to unfocused windows aswell as focused windows when set to `true` or only applies to focused windows when set to `false`.

Property name: `useAcrylic`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Window

Padding

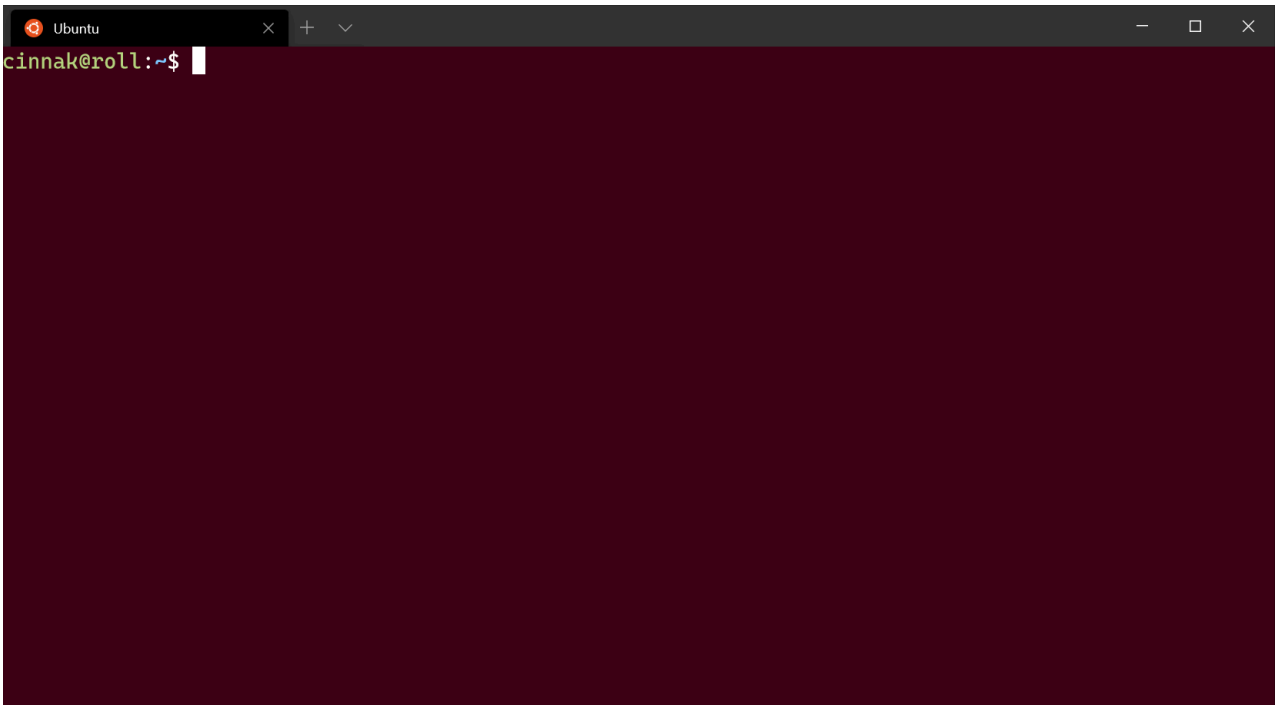
This sets the padding around the text within the window. This will accept three different formats: `"#"` and `#` set the same padding for all sides, `"#, #"` sets the same padding for left-right and top-bottom, and `"#, #, #, #"` sets the padding individually for left, top, right, and bottom.

Property name: `padding`

Necessity: Optional

Accepts: Values as a string in the following formats: `"#"`, `"#, #"`, `"#, #, #, #"` or value as an integer: `#`

Default value: `"8, 8, 8, 8"`



Scrollbar visibility

This sets the visibility of the scrollbar.

Property name: `scrollbarState`

Necessity: Optional

Accepts: `"visible"`, `"hidden"`, (Beginning in release 1.17, `"always"` will be included)

Color settings

Tab color

This sets the color of the profile's tab. Using the tab color picker will override this color.

Property name: `tabColor`

Necessity: Optional

Accepts: Color as a string in hex format: `"#rgb"` or `"#rrggbb"`

Foreground color

This changes the foreground color of the profile. This overrides `foreground` set in the color scheme if `colorScheme` is set.

Property name: `foreground`

Necessity: Optional

Accepts: Color as a string in hex format: `"#rgb"` or `"#rrggbb"`

Background color

This changes the background color of the profile with this setting. This overrides `background` set in the color scheme if `colorScheme` is set.

Property name: `background`

Necessity: Optional

Accepts: Color as a string in hex format: `"#rgb"` or `"#rrggbb"`

Selection background color

This sets the background color of a selection within the profile. This will override the `selectionBackground` set in the color scheme if `colorScheme` is set.

Property name: `selectionBackground`

Necessity: Optional

Accepts: Color as a string in hex format: `"#rgb"` or `"#rrggbb"`

Adjust indistinguishable colors

This setting adjusts the foreground color to make it more visible, based on the background color. When set to `always`, the colors will always be adjusted. When set to `indexed`, the colors will only be adjusted if those colors are part of the color scheme. When set to `never`, the colors will never be adjusted.

Property name: `adjustIndistinguishableColors`

Necessity: Optional

Accepts: `always`, `indexed`, `never`

Cursor color

This sets the cursor color of the profile. This will override the `cursorColor` set in the color scheme if `colorScheme` is set.

Property name: `cursorColor`

Necessity: Optional

Accepts: Color as a string in hex format: `"#rgb"` or `"#rrggbb"`

Unfocused appearance settings

An object you can add to a profile that applies settings to the profile when it is unfocused. This setting only accepts appearance settings.

Property name: `unfocusedAppearance`

Necessity: Optional

Accepts: `backgroundImage`, `backgroundImageAlignment`, `backgroundImageOpacity`, `backgroundImageStretchMode`, `cursorHeight`, `cursorShape`, `cursorColor`, `colorScheme`, `foreground`, `background`, `opacity`, `selectionBackground`, `useAcrylic`, `experimental.retroTerminalEffect`, `experimental.pixelShaderPath`

Example:

JSON

```
// Sets the profile's background image opacity to 0.3 when it is unfocused
"unfocusedAppearance":
{
    "backgroundImageOpacity": 0.3
},
```

Pixel shader effects

This setting allows a user to specify the path to a custom pixel shader to use with the terminal content. This is an experimental feature and its continued existence is not guaranteed. For more details on authoring custom pixel shaders for the terminal, see this documentation.

If set, this will override the `experimental.retroTerminalEffect` setting.

Property name: `experimental.pixelShaderPath`

Necessity: Optional

Accepts: A path to an `.hlsl` shader file, as a string

information, see our contributor guide.

Provide product feedback

Advanced profile settings in Windows Terminal

Article • 05/29/2024

The settings listed below are specific to each unique profile. If you'd like a setting to apply to all of your profiles, you can add it to the `defaults` section above the list of profiles in your settings.json file.

JSON

```
"defaults":
{
    // SETTINGS TO APPLY TO ALL PROFILES
},
"list":
[
    // PROFILE OBJECTS
]
```

Suppress title changes

When this is set to `true`, `tabTitle` overrides the default title of the tab and any title change messages from the application will be suppressed. If `tabTitle` isn't set, `name` will be used instead. When this is set to `false`, `tabTitle` behaves as normal.

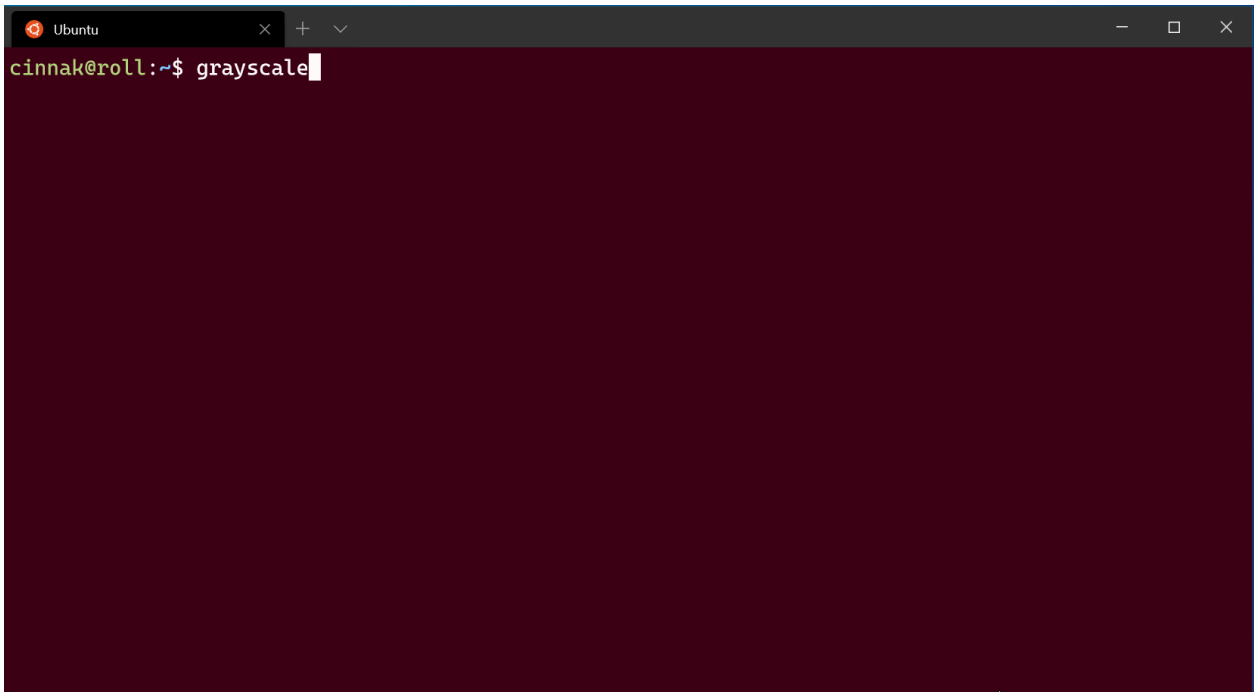
Property name: `suppressApplicationTitle`

Necessity: Optional

Accepts: `true`, `false`

Text antialiasing

This controls how text is antialiased in the renderer. Note that changing this setting will require starting a new terminal instance.



Property name: `antialiasingMode`

Necessity: Optional

Accepts: `"grayscale"`, `"cleartype"`, `"aliased"`

Default value: `"grayscale"`

AltGr aliasing

This allows you to control if Windows Terminal will treat `Ctrl+Alt` as an alias for `AltGr`.

Property name: `altGrAliasing`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

Scroll to input when typing

When this is set to `true`, the window will scroll to the command input line when typing. When it's set to `false`, the window will not scroll when you start typing.

Property name: `snapOnInput`

Necessity: Optional

Accepts: `true`, `false`

Default value: `true`

History size

This sets the number of lines above the ones displayed in the window you can scroll back to. The maximum history size is `32767`.

Property name: `historySize`

Necessity: Optional

Accepts: Integer

Default value: `9001`

Profile termination behavior

This sets how the profile reacts to termination or failure to launch. `"graceful"` will close the profile when `exit` is typed or when the process exits normally. `"always"` will always close the profile and `"never"` will never close the profile. `"automatic"` was added once Windows Terminal was allowed to be the default terminal application; for processes launched in Terminal directly, it behaves the same as `"graceful"` but for processes that were handed off to Terminal it behaves the same as `"always"`.

`true` and `false` are accepted as synonyms for `"graceful"` and `"never"`, respectively.

Property name: `closeOnExit`

Necessity: Optional

Accepts: `"automatic"`, `"graceful"`, `"always"`, `"never"`, `true`, `false`

Default value: `"automatic"`

> **Note**
>
> In Windows Command Prompt (cmd.exe), `exit` will return the return code of the previous command. If the command you typed before `exit` resulted in an error, then `"closeOnExit": "graceful"` will still show that error code, instead of closing the tab.

Bell notification style

Controls what happens when the application emits a BEL character. When set to `"all"`, the terminal will play a sound and flash the taskbar icon. When the terminal is not in focus, only the taskbar icon will flash.

Property name: `bellStyle`

Necessity: Optional

Accepts: `"all"`, `"audible"`, `"window"`, `"taskbar"`, `"none"`

Default value: `"audible"`

Bell sound

When `bellStyle` is set to `"all"` or `"audible"`, this allows you to choose the audio file for the bell. If you have an array of sounds set, the terminal will pick one at random.

Property name: `bellSound`

Necessity: Optional

Accepts: File location as a string or an array of file locations as strings

---

Scroll marks (Preview)

The following settings modify how scroll marks behave in Windows Terminal. For more info on marks and shell integration, visit the Shell integration page.

Automatically add scroll marks

When set to `true`, the Terminal will automatically mark the current like as a prompt when the user presses `Enter`. If the user has shell integration enabled, this will treat the current cursor position as a `FTCS;C`, ending the current command and starting the output.

Property name: `autoMarkPrompts`

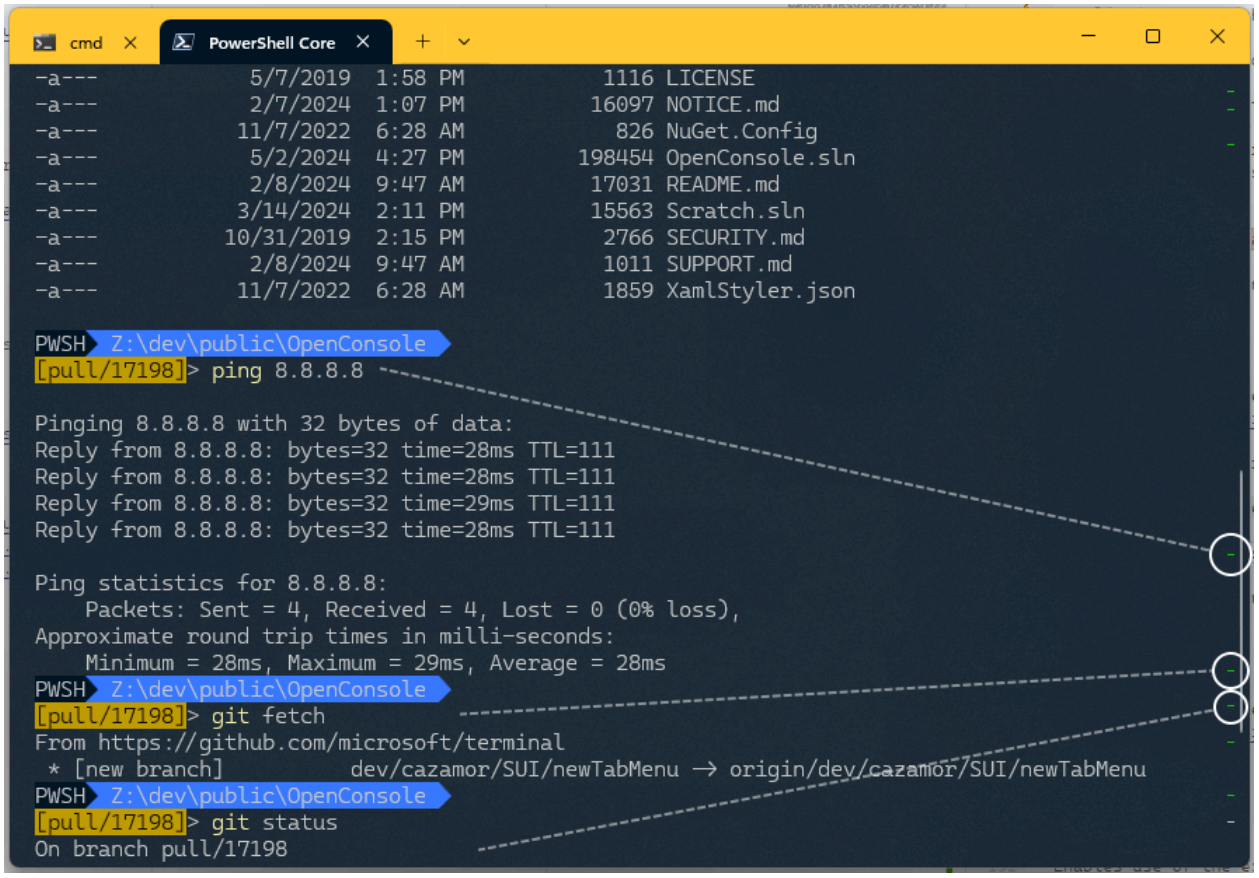
Necessity: Optional

Accepts: `true`, `false`

> **ⓘ Important**
>
> This feature became stable in v1.21. Before that version, it was only available in Windows Terminal Preview, and was named `experimental.autoMarkPrompts`.

Show marks on scrollbar

Displays marks on the scrollbar when set to `true`. When combined with `autoMarkPrompts` or shell integration, this will show the location of the prompts on the scrollbar.



Property name: `showMarksOnScrollbar`

Necessity: Optional

Accepts: `true`, `false`

> **Important**
>
> This feature became stable in v1.21. Before that version, it was only available in Windows Terminal Preview, and was named `experimental.showMarksOnScrollbar`.

Experimental text rendering engine

Enables use of the experimental text rendering engine for the profile. This is an experimental feature and its continued existence is not guaranteed. A new instance of the profile needs to be opened in order for this setting to take effect.

Property name: `experimental.useAtlasEngine`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Right click context menu (Preview)

When enabled, right-click will open a context menu with options to copy, paste, and more. When disabled, right-click will paste the contents of the clipboard into the terminal. With shell integration enabled, right-click will also allow you to select the current command or output. This is an experimental feature, and its continued existence is not guaranteed.

Property name: `experimental.rightClickContextMenu`

Necessity: Optional

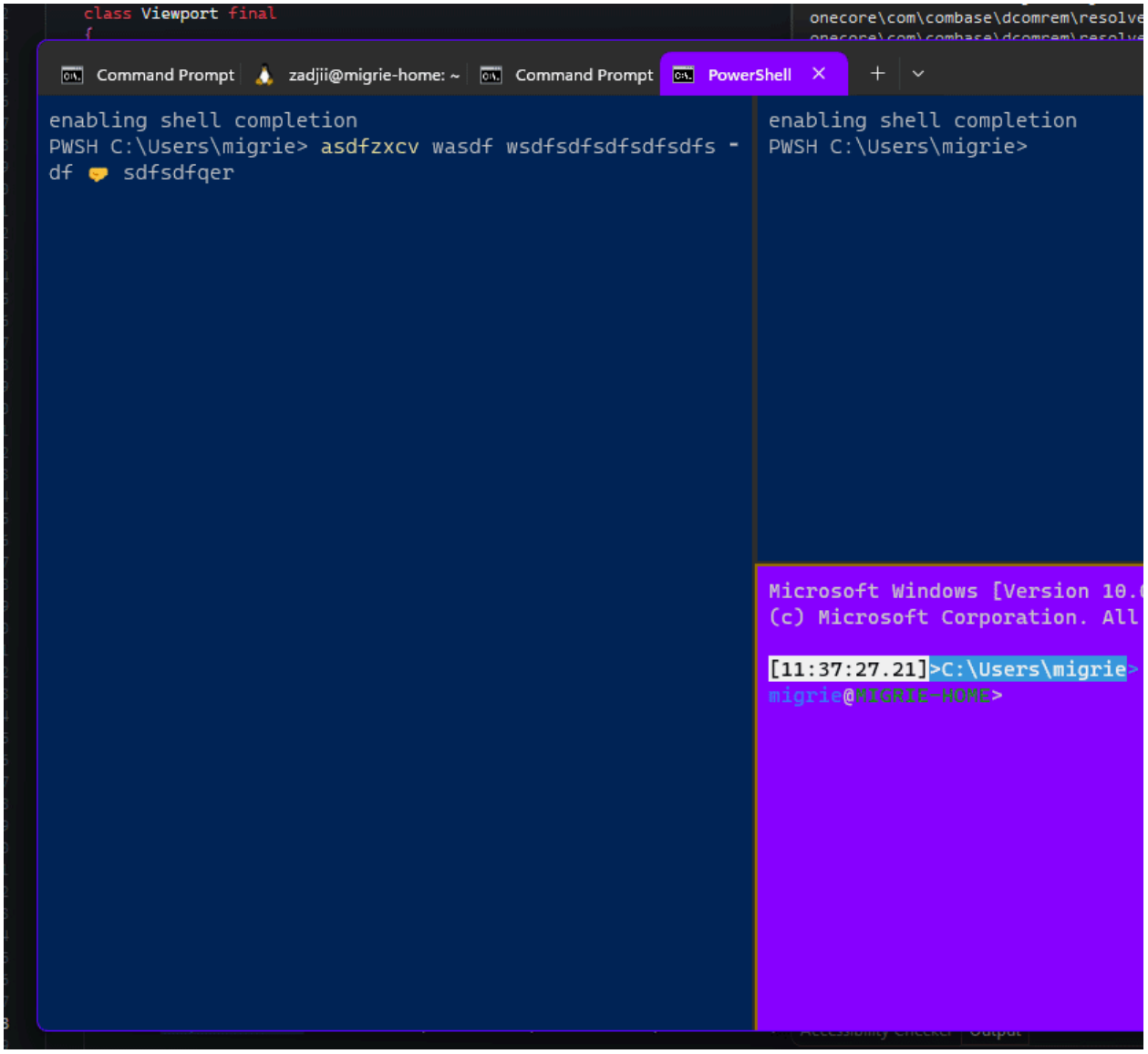
Accepts: `true`, `false`

Default value: `false`

> **Important**
>
> This feature is only available in Windows Terminal Preview.

Experimental: Move cursor with the mouse

This experimentally adds support for moving the text cursor by clicking with the mouse on the current commandline. This is an experimental feature - there are lots of edge cases where this will not work as expected. In order for this setting to work, you will need to enable shell integration in your shell.



Property name: `experimental.repositionCursorWithMouse`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

VT passthrough mode

When set to true, directs the PTY for this connection to use pass-through mode instead of the original Conhost PTY simulation engine. This is an experimental feature, and its continued existence is not guaranteed.

Property name: `experimental.connection.passthroughMode`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

Unique identifier

Profiles can use a GUID as a unique identifier. To make a profile your default profile, it needs a GUID for the `defaultProfile` global setting.

Property name: `guid`

Necessity: Required

Accepts: GUID as a string in registry format: `"{00000000-0000-0000-0000-000000000000}"`

> **Tip**
>
> You can run `[guid]::NewGuid()` in PowerShell to generate a GUID for your custom profile. You can also use the **online GUID generator** or for other command lines, use the **UUID generator**.

Source

This stores the name of the profile generator that originated the profile. *There are no discoverable values for this field.* For additional information on dynamic profiles, visit the Dynamic profiles page.

Property name: `source`

Necessity: Optional

Accepts: String

> **Note**
>
> This field should be omitted when declaring a custom profile. It is used by Terminal to connect automatically generated profiles to your settings file.

Theme settings in Windows Terminal (Preview)

Article • 09/28/2023

The settings listed below affect the visuals of the terminal window itself, rather than the appearance of an individual tab/pane. These settings are currently only editable directly in the settings.json file and are not configurable via the settings UI.

JSON

```
"theme": "dark"
"themes":
[
    // THEME OBJECTS
]
```

For some example themes, take a look at the Themes gallery.

Each theme in the `themes` list is comprised of a collection of property objects, that specify the properties of individual elements of the application. For example, the default `"dark"` theme is the following:

JSON

```
{
    "name": "dark",
    "window": {
        "applicationTheme": "dark"
    },
    "tab": {
        "background": "terminalBackground",
        "unfocusedBackground": "#00000000"
    },
    "tabRow": {
        "unfocusedBackground": "#333333FF"
    }
},
```

You may also configure the Terminal to use separate themes for light and dark mode in the OS and change automatically between those themes when the OS theme changes. To do this, specify the `theme` property as an object containing the keys `light` and `dark`:

JSON

```
"theme": { "dark": "<Dark Theme Name>", "light": "<Light Theme Name>" },
```

Theme name

This is the name of the theme. Names should be unique. The names `dark`, `light`, and `system` are reserved for the built-in default themes.

Property name: `name`

Necessity: Required

Accepts: Name of theme as a string

Window

These settings are used to configure the appearance of the whole of the window of the Terminal.

Property name: `window`

Application theme

This sets the UI theme of the application. This will stylize items such as buttons, the command palette, and other application UI elements. It can either be light or dark. `"system"` will use the same theme as Windows.

Property name: `applicationTheme`

Necessity: Optional

Accepts: `"system"`, `"dark"`, `"light"`

Default value: `"dark"`

Mica

This enables the Mica effect on this window, beneath all other UI layers. For Mica to be visible, the layers above it need to be transparent. As an example, to have a tab row with Mica in it you'll need to configure the alpha channel of the background to be `0` as follows:

JSON

```
{
    "name": "My Mica Theme",
    "tab":
    {
        "background": "terminalBackground"
    },
    "tabRow":
    {
        "background": "#00000000"
    },
    "window":
    {
        "applicationTheme": "system",
        "useMica": true
    }
},
```

Note that when Mica is enabled for the window it is enabled under the entirety of the window, including as a backdrop for the Terminal panes in the window. This means that profiles which are using `opacity` without `useAcrylic` enabled will show through to the new Mica background. It is not currently possible to have an unblurred transparent background for the Terminal and a Mica background for the tabs / tab row simultaneously.

Property name: `useMica`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

> ⓘ Note
>
> Mica is only available on Windows builds >= 22621.

Window border

This sets the color of the window border, when the window is active. When set to `null`, the border will use whatever the default color is for the OS theme.

Property name: `frame`

Necessity: Optional

Accepts: a theme color

Default value: `null`

> ⓘ **Note**
>
> Window border colors are only available on Windows 11.

> ⓘ **Important**
>
> This feature is only available in Windows Terminal Preview.

Inactive window border

This sets the color of the window border, when the window is inactive. When set to `null`, the border will use whatever the default color is for the OS theme.

Property name: `unfocusedFrame`

Necessity: Optional

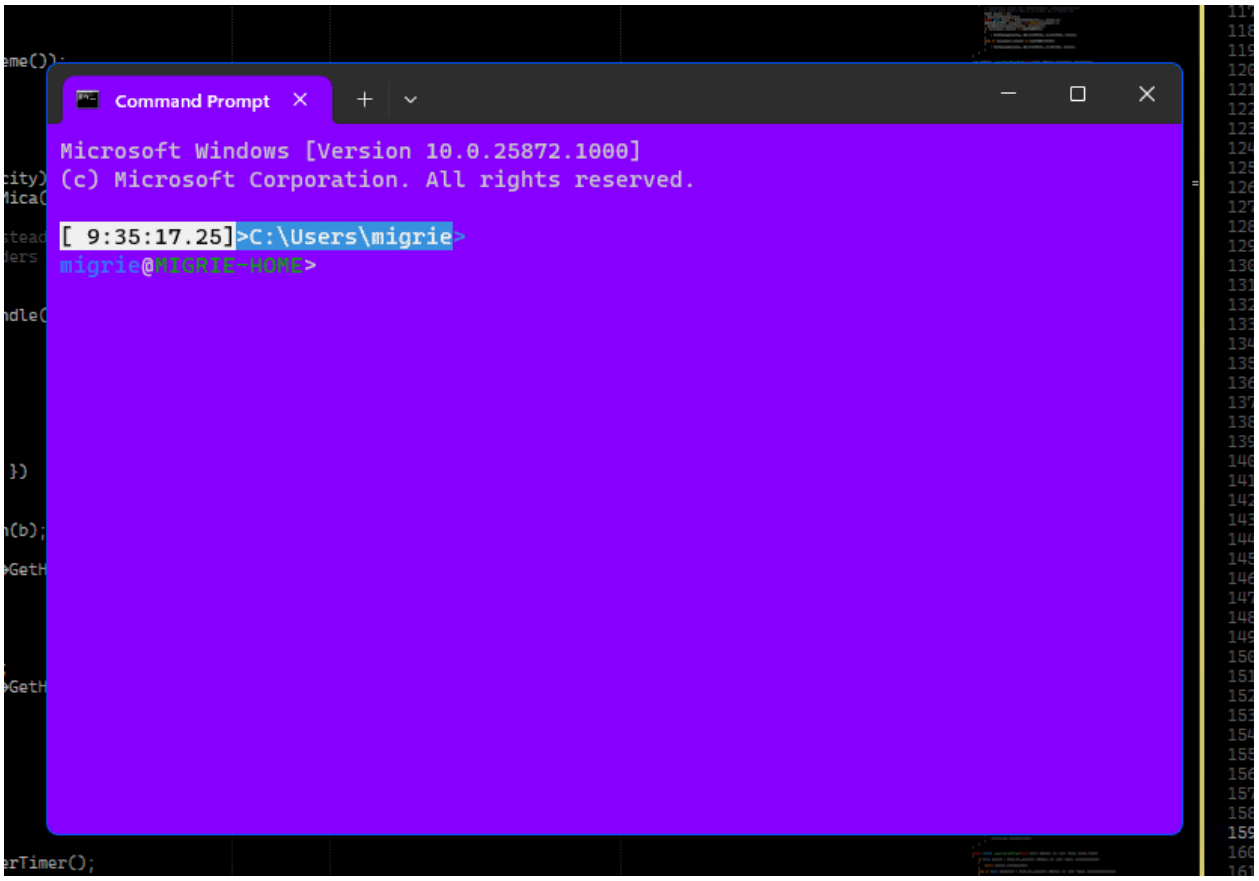
Accepts: a theme color

Default value: `null`

> ⓘ **Important**
>
> This feature is only available in Windows Terminal Preview.

Experimental: Rainbow Window border

When enabled, this setting will cause the window border to cycle through the colors of the rainbow. This is an experimental feature, and its continued existence is not guaranteed. When this setting is enabled, it will take precedence over both `frame` and `unfocusedFrame`.



Property name: `experimental.rainbowFrame`

Necessity: Optional

Accepts: `true`, `false`

Default value: `false`

> ⓘ **Important**
>
> This feature is only available in Windows Terminal Preview.

Tab row

These settings are used to configure the appearance of the tab row. When `showTabsInTitlebar` is `true` (the default), this configures the title bar.

Property name: `tabRow`

Background color

The color of the tab row when the window is in the foreground.

Property name: `background`

Necessity: Optional

Accepts: a theme color.

Inactive background color

The color of the tab row when the window is inactive.

Property name: `unfocusedBackground`

Necessity: Optional

Accepts: a theme color.

Tabs

These are settings that control the appearance of individual tabs in the Terminal.

Property name: `tab`

Background color

The color of the active tab. Setting a `tabColor` in a profile will override this value. Similarly, setting a color at runtime with the tab color picker will override this color.

This color is always treated as a solid color, even if set to a `terminalBackground` of a pane with an acrylic background.

Property name: `background`

Necessity: Optional

Accepts: a theme color.

Inactive background color

The color of inactive tabs. Setting a `tabColor` in a profile will override this value. Similarly, setting a color at runtime with the tab color picker will override this color.

This color is always treated as a solid color, even if set to a `terminalBackground` of a pane with an acrylic background.

When set to `terminalBackground` or `accent`, this will automatically use an alpha value of 30%, to be semi-transparent.

Property name: `unfocusedBackground`

Necessity: Optional

Accepts: a theme color.

Show close button

Configures how the "close" button on the tab should appear. This accepts the following values:

- `"always"`: Always show tab close buttons.
- `"hover"`: Show the tab close button on the active tab, and any tabs that are hovered with the mouse.
- `"never"`: Never show tab close buttons. This also disables the ability to close the tab with the middle mouse button.
- `"activeOnly"`: Show the tab close button on the active tab only.

Property name: `showCloseButton`

Necessity: Optional

Accepts: `"always"`, `"hover"`, `"never"`, `"activeOnly"`

Default value: `"always"`

Theme colors

The colors used in themes both accept RGBA color values, as well as a few special strings for custom values. The accepted values are as follows:

- `"#rgb"`, `"#rrggbb"`, `"#rrggbbaa"`: An RGB color value. When the alpha chanel is omitted, these colors default to a fully opaque alpha channel.
- `"accent"`: This is a special value that means "the accent color set in the system settings".

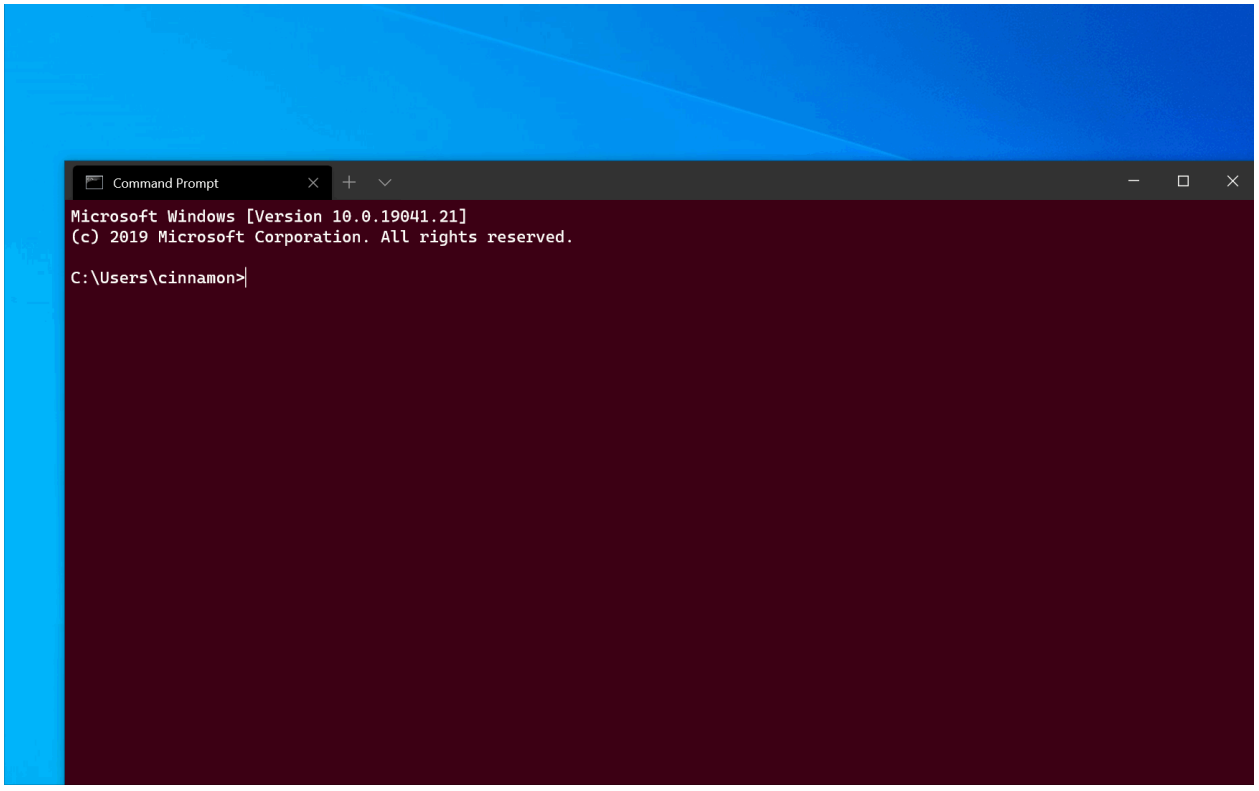
- `"terminalBackground"`: This is a special value evaluated to mean "the background color of the active terminal pane". If there are multiple panes in a tab, then this is the color of the active one. This always uses the `background` of the profile - it ignores anything from a `backgroundImage`, if set.

Using command line arguments for Windows Terminal

Article • 05/29/2024

You can use `wt.exe` to open a new instance of Windows Terminal from the command line. You can also use the execution alias `wt` instead.

> **Note**
>
> If you built Windows Terminal from the source code on GitHub, you can open that build using `wtd.exe` or `wtd`.



Command line syntax

The `wt` command line accepts two types of values: **options** and **commands**. **Options** are a list of flags and other parameters that can control the behavior of the `wt` command line as a whole. **Commands** provide the action, or list of actions separated by semicolons, that should be implemented. If no command is specified, then the command is assumed to be `new-tab` by default.

Windows Command Prompt

```
wt [options] [command ; ]
```

> **Note**
>
> The behavior of the `wt.exe` command may be affected by the windowingBehavior property. This setting can be adjusted to default between opening a new window or opening a new tab.

To display a help message listing the available command line arguments, enter: `wt -h`, `wt --help`, `wt -?`, or `wt /?`.

Options and commands

Below is the full list of supported commands and options for the `wt` command line.

Expand table

Option	Description
`--help, -h, -?, /?`	Displays the help message.
`--maximized, -M`	Launches the terminal maximized.
`--fullscreen, -F`	Launches the terminal as full screen.
`--focus, -f`	Launches the terminal in the focus mode. Can be combined with `maximized`.
`--pos x,y`	Launches the terminal at the given position. `x` or `y` can be omitted, to use the default value from the settings.
`--size c,r`	Launches the terminal with the specified number of columns (`c`) and rows (`r`).
`--window, -w window-id`	Runs the given command in a specific window.

The `--window` parameter can be used to send commands to existing terminal windows.

`window-id` may either be the integer ID of a window, or the name of a window. It also accepts the following reserved values:

- `new` or `-1`: Always run this command in a new window
- `last` or `0`: Always run this command in the most recently used window

If no window exists with the given `window-id`, then a new window will be created with that id/name.

For example, running `wt -w _quake` will open a new "quake window". Running that command again will open a new tab in the existing quake window.

New tab command

Used to create a new tab. See also the newTab action.

Expand table

Command	Parameter	Description	Values
`new-tab`, `nt`	`--profile, -p profile-name`	Creates a new tab based on the profile name assigned.	Profile name
`new-tab`, `nt`	`--startingDirectory, -d starting-directory`	Creates a new tab based on the starting directory path assigned.	Directory path
`new-tab`, `nt`	`commandline`	Creates a new tab based on the command line assigned.	Executable with optional commands
`new-tab`, `nt`	`--title`	Creates a new tab with the title assigned.	Text to use as the tab title
`new-tab`, `nt`	`--tabColor`	Creates a new tab with the tab color assigned.	Hex color as #RGB or #RRGGBB
`new-tab`, `nt`	`--suppressApplicationTitle`	Override the profile's `suppressApplicationTitle` setting, and set it to `true`	
`new-tab`, `nt`	`--useApplicationTitle`	Override the profile's `suppressApplicationTitle` setting, and set it to `false`	
`new-tab`, `nt`	`--colorScheme scheme-name`	Override the profile's `colorScheme` setting, and set it to the scheme from the settings with the name `scheme-name`	The name of a color scheme in the settings

Tip

If you change the title of a tab in Windows Terminal and want that title to persist, you must enable the **suppressApplicationTitle** option by setting it to `true`.

Split-pane command

Used to create a new split pane. See also the splitPane action.

Expand table

Command	Parameter	Description	Values
`split-pane`, `sp`	`-H, --horizontal`, `-V, --vertical`	Creates a new split window pane either horizontally or vertically.	N/A. No additional values to assign.
`split-pane`, `sp`	`--profile, -p profile-name`	Creates a new split window pane based on the assigned command line profile. If this parameter is not assigned, the default profile will be used.	Profile name
`split-pane`, `sp`	`--startingDirectory, -d starting-directory`	Creates a new split window pane based on the assigned starting directory path. If this parameter is not assigned, the default starting directory will be used.	Directory path
`split-pane`, `sp`	`--title`	Creates a new split window pane with the assigned title.	Text to use as the tab title
`split-pane`, `sp`	`--tabColor`	Creates a new split window pane with the assigned tab color.	Hex color as #RGB or #RRGGBB
`split-pane`, `sp`	`--size, -s size`	Creates a new split window pane with the assigned size.	Float that specifies the portion of the parent pane to use represented by a decimal. For example, `.4` to represent 40% of the parent pane.
`split-pane`, `sp`	`commandline`	Creates a new split window pane based on the assigned command line.	Executable with optional commands
`split-pane`, `sp`	`--duplicate, -D`	Creates a new split window pane that is a duplicate of the current pane.	N/A. No additional values to assign.

Command	Parameter	Description	Values
`split-pane`, `sp`	`--suppressApplicationTitle`	Override the profile's `suppressApplicationTitle` setting, and set it to `true`	
`split-pane`, `sp`	`--useApplicationTitle`	Override the profile's `suppressApplicationTitle` setting, and set it to `false`	
`split-pane`, `sp`	`--colorScheme scheme-name`	Override the profile's `colorScheme` setting, and set it to the scheme from the settings with the name `scheme-name`	The name of a color scheme in the settings

Focus-tab command

Used to focus a specific tab within the window. See also the switchToTab action.

Expand table

Command	Parameter	Description	Values
`focus-tab`, `ft`	`--target, -t tab-index`	Focuses on a specific tab according to it's tab index number.	Tab index as an integer

Move-focus command

Used to move focus within the window. See also the moveFocus action.

Expand table

Command	Parameter	Description	Values
`move-focus`, `mf`	`<direction>`	Move focus between panes.	See below for accepted `direction` values

Accepted `direction` values

- `up`, `down`, `left`, or `right` move focus in the given direction.
- `first` moves focus to the first leaf pane in the tree.
- `previous` moves the focus to the most recently used pane before the current pane.
- `nextInOrder`, `previousInOrder` moves the focus to the next or previous pane in order of creation.

Move-pane command

Used to move a pane within the window. See also the movePane action.

Expand table

Command	Parameter	Description	Values
`move-pane`, `mp`	`--tab,-t <index>`	Move the active pane to the given tab in the window	The zero-indexed index of the tab to move the pane to.

Swap-pane command

Used to swap the position of two panes within the window. See also the swapPane action.

Expand table

Command	Parameter	Description	Values
`swap-pane`	`<direction>`	Swap the pane with the pane in the given direction	See below for accepted `direction` values

Accepted `direction` values (these are the same values as the `move-focus` sub command)

- `up`, `down`, `left`, or `right`: Swap the active pane with the one in the given direction.
- `first`: Swap the active pane with the first leaf pane in the tree.
- `previous`: Swap the active pane with the most recently used pane before the current pane.
- `nextInOrder`, `previousInOrder`: Swap the active pane with the next or previous pane in order of creation.

Command line argument examples

Commands may vary slightly depending on which command line you're using.

Passing an argument to the default shell

To start an instance of Windows Terminal and have it execute a command, call `wt.exe` followed by your command.

Here's an example of calling Windows Terminal to pass a ping command argument to echo an IP address:

PowerShell

```
wt ping learn.microsoft.com
```

Here's an example of calling Windows Terminal to open a new tab with a PowerShell command line, confirming to call the Start-Service command, and opening another new tab with Windows Command Prompt open to the `/k` directory:

Windows Command Prompt

```
wt new-tab PowerShell -c Start-Service ; new-tab cmd /k dir
```

Target a specific window

Below are examples of how to target specific windows using the `--window,-w` option.

Command Prompt

Windows Command Prompt

```
// Open a new tab with the default profile in the current window
wt -w 0 nt

// Open a new tab in a new window with the default profile
wt -w -1 nt

// Open a new tab in the first-created terminal window with the default
profile
wt -w 1 nt

// Open a new tab in the terminal window named foo with the default
profile. If foo does not exist, create a new window named foo.
wt -w foo nt
```

Open a new profile instance

To open a new terminal instance, in this case the command will open the profile named "Ubuntu-18.04", enter:

Command Prompt

Windows Command Prompt

```
wt -p "Ubuntu-18.04"
```

The `-p` flag is used to specify the Windows Terminal profile that should be opened. Substitute "Ubuntu-18.04" with the name of any terminal profile that you have installed. This will always open a new window. Windows Terminal is not yet capable of opening new tabs or panes in an existing instance.

Target a directory

To specify the folder that should be used as the starting directory for the console, in this case the d:\ directory, enter:

Command Prompt

Windows Command Prompt

```
wt -d d:\
```

Multiple tabs

To open a new terminal instance with multiple tabs, enter:

Command Prompt

Windows Command Prompt

```
wt ; ;
```

To open a new terminal instance with multiple tabs, in this case a Command Prompt profile and a PowerShell profile, enter:

Command Prompt

Windows Command Prompt

```
wt -p "Command Prompt" ; new-tab -p "Windows PowerShell"
```

Multiple panes

To open a new terminal instance with one tab containing three panes running a Command Prompt profile, a PowerShell profile, and your default profile running a WSL command line, enter:

Command Prompt

Windows Command Prompt

```
wt -p "Command Prompt" ; split-pane -p "Windows PowerShell" ; split-pane
-H wsl.exe
```

The `-H` flag (or `--horizontal`) indicates that you would like the panes to be split horizontally. The `-V` flag (or `--vertical`) indicates that you would like the panes split vertically.

Multiple tabs and panes

The `new-tab` and `split-pane` commands can be sequenced to get multiple tabs, each with split panes. To open a new terminal instance with two tabs, each with two panes running a Command Prompt and a WSL command line, with each tab in a different directory, enter:

Command Prompt

Windows Command Prompt

```
wt -p "Command Prompt" ; split-pane -V wsl.exe ; new-tab -d c:\ ; split-
pane -H -d c:\ wsl.exe
```

Pane title

To open a new terminal instance with custom titles for each terminal pane, use the `--title` argument. To set the title of each pane when opening multiple tabs, enter:

Command Prompt

Windows Command Prompt

```
wt --title tabname1 ; new-tab -p "Ubuntu-18.04" --title tabname2
```

Panes in the same tab can have different titles, which will reflect on the tab title depending on which pane has focus. To name independent panes, you can set the title after splitting the panes by entering:

Command Prompt

Windows Command Prompt

```
wt --title pane1 ; split-pane -p "Command Prompt" --title pane2
```

Using application title

To open a new terminal instance allowing applications within it to set the tab title by sending title change messages, use the `--useApplicationTitle` flag. To suppress these messages, use the `--suppressApplicationTitle` flag. If none of these flags are provided, the behavior is inherited from the profile's settings. To open a tab with title `tabname` that will not be overridden by the application, enter:

Command Prompt

Windows Command Prompt

```
wt --title tabname --suppressApplicationTitle
```

Tab color

To open a new terminal instance with custom tab colors, use the `--tabColor` argument. This argument overrides the value defined in the profile, but can be overridden as well using the tab color picker. In the following example, a new terminal is created with two tabs of different colors:

Command Prompt

Windows Command Prompt

```
wt --tabColor #009999 ; new-tab --tabColor #f59218
```

When `--tabColor` is set for a tab, it is associated with the first pane of this tab. Hence in a tab with multiple panes, the color will be applied only if the first pane is in focus. To set the tab color for additional panes, you will need to add the `--tabColor` parameter to the `split-pane` subcommand as well. In the example below, a tab with two panes is created with tab colors specified for each pane:

PowerShell

```
wt new-tab --tabColor '#009999' `; split-pane --tabColor '#f59218'
```

Color scheme

To open a new terminal instance with a specific color scheme (instead of the `colorScheme` set in the profile), use the `--colorScheme` argument. This argument overrides the value defined in the profile.

Command Prompt

Windows Command Prompt

```
wt --colorScheme Vintage ; split-pane --colorScheme "Tango Light"
```

Tab focus

To open a new terminal instance with a specific tab in focus, use the `-t` flag (or `--target`), along with the tab-index number. To open your default profile in the first tab and the "Ubuntu-18.04" profile focused in the second tab (`-t 1`), enter:

Command Prompt

Windows Command Prompt

```
wt ; new-tab -p "Ubuntu-18.04" ; focus-tab -t 1
```

Examples of multiple commands from PowerShell

Windows Terminal uses the semicolon character `;` as a delimiter for separating commands in the `wt` command line. Unfortunately, PowerShell also uses `;` as a command separator. To work around this, you can use the following tricks to run multiple `wt` commands from PowerShell. In all the following examples, a new terminal window is created with three panes - one running Command Prompt, one with PowerShell, and the last one running WSL.

The following examples do not use `start` to run the command line. Instead, there are two other methods of escaping the command line:

- Only escaping the semicolons so that `PowerShell` will ignore them and pass them straight to `wt`.
- Using `--%`, so PowerShell will treat the rest of the command line as arguments to the application.

PowerShell

```
wt new-tab "cmd" `; split-pane -p "Windows PowerShell" `; split-pane -H
wsl.exe
```

PowerShell

```
wt --% new-tab cmd ; split-pane -p "Windows PowerShell" ; split-pane -H
wsl.exe
```

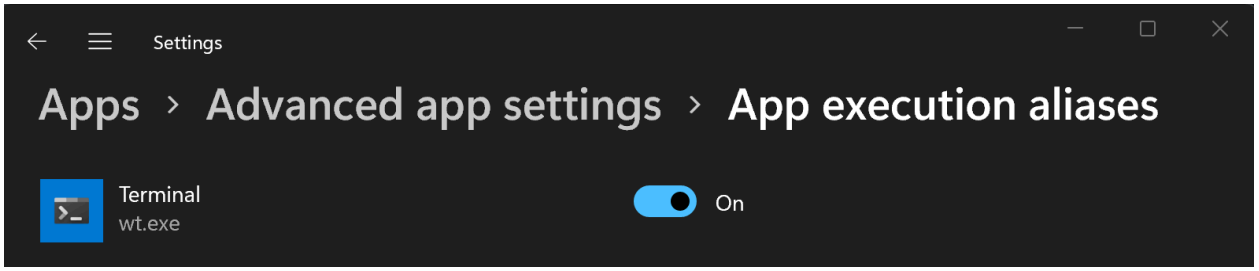
In both of these examples, the newly created Windows Terminal window will create the window by correctly parsing all the provided command-line arguments.

However, these methods are *not* recommended currently, as PowerShell will wait for the newly-created terminal window to be closed before returning control to PowerShell. By default, PowerShell will always wait for Windows Store applications (like Windows Terminal) to close before returning to the prompt. Note that this is different than the behavior of Command Prompt, which will return to the prompt immediately.

Add Windows Terminal executable to your PATH

To add the Windows Terminal executable file (wt.exe) to your PATH, enable its "app execution alias" in the **Manage app execution aliases** page of Windows Settings. The Windows Terminal alias is turned on by default, but may be worth confirming if you're having issues accessing it.

If you are still having trouble accessing app execution aliases, you might need to check whether your PATH contains: `%LOCALAPPDATA%\Microsoft\WindowsApps`. Do not attempt to make changes to `C:\Program Files\WindowsApps`.



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

How to use the command palette in Windows Terminal

Article • 10/06/2022

The command palette lets you see which actions you can run inside Windows Terminal. More information on how actions are defined can be found on the Actions page.

Invoking the command palette

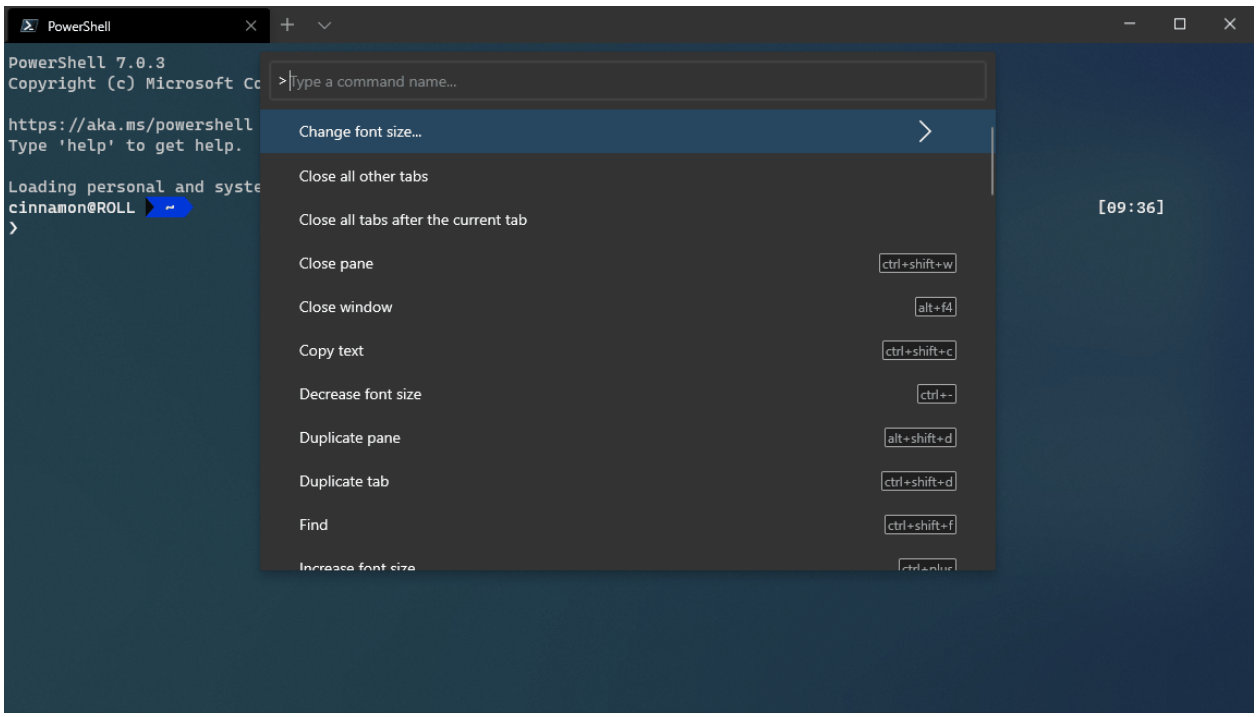
You can invoke the command palette by typing `Ctrl+Shift+P`. This can be customized by adding the `commandPalette` command to your key bindings.

JSON

```
{ "command": "commandPalette", "keys": "ctrl+shift+p" }
```

Command line mode

If you'd like to enter a `wt` command into the command palette, you can do so by deleting the `>` character in the text box. This will run the `wt` command in the current window. More information on `wt` commands can be found on the Command line arguments page.



You can add a custom key binding for invoking the command palette in the command line mode directly.

JSON

```
{ "command": "commandPalette", "launchMode": "commandLine", "keys": "" }
```

Adding an icon to a command

You can optionally add an icon to a command defined in your settings.json that appears in the command palette. This can be done by adding the `icon` property to the action. Icons can be a path to an image, a symbol from Segoe MDL2 Assets, or any character, including emojis.

JSON

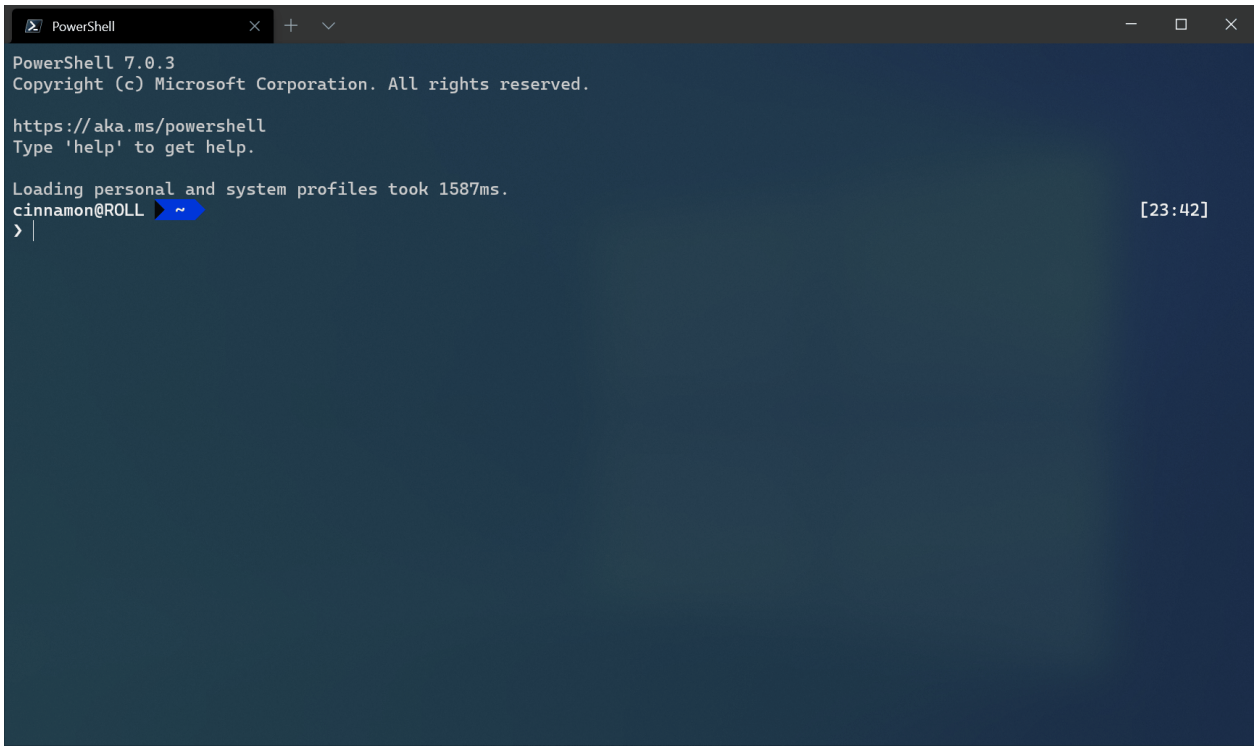
```
{ "icon": "C:\\Images\\my-icon.png", "name": "New tab", "command": "newTab",
"keys": "ctrl+shift+t" },
{ "icon": "\uE756", "name": "New tab", "command": "newTab", "keys":
"ctrl+shift+t" },
{ "icon": "⚡", "name": "New tab", "command": "newTab", "keys":
"ctrl+shift+t" }
```

Nested commands

Nested commands let you group multiple commands under one item in the command palette. The example below groups the font resize commands under one command palette item called "Change font size...".

JSON

```
{
    "name": "Change font size...",
    "commands": [
        { "command": { "action": "adjustFontSize", "delta": 1 } },
        { "command": { "action": "adjustFontSize", "delta": -1 } },
        { "command": "resetFontSize" },
    ]
}
```



Iterable commands

Iterable commands let you create multiple commands at the same time, generated from other objects defined in your settings. Currently, you can create iterable commands for your profiles and color schemes. At runtime, these commands will be expanded to one command for each of the objects of the given type.

You can currently iterate over the following properties:

Expand table

`iterateOn`	Property	Property syntax
`profiles`	`name`	`"name": "${profile.name}"`
`profiles`	`icon`	`"icon": "${profile.icon}"`
`schemes`	`name`	`"name": "${scheme.name}"`

Example

Create a new tab command for each profile.

JSON

```
{
    "iterateOn": "profiles",
```

```
    "icon": "${profile.icon}",
    "name": "${profile.name}",
    "command": { "action": "newTab", "profile": "${profile.name}" }
}
```

In the above example:

- `"iterateOn": "profiles"` will generate a command for each profile.
- At runtime, the terminal will replace `${profile.icon}` with each profile's icon and `${profile.name}` with each profile's name.

If you had three profiles:

JSON

```
"profiles": [
    { "name": "Command Prompt", "icon": null },
    { "name": "PowerShell", "icon": "C:\\path\\to\\icon.png" },
    { "name": "Ubuntu", "icon": null },
]
```

The above command would behave like the following three commands:

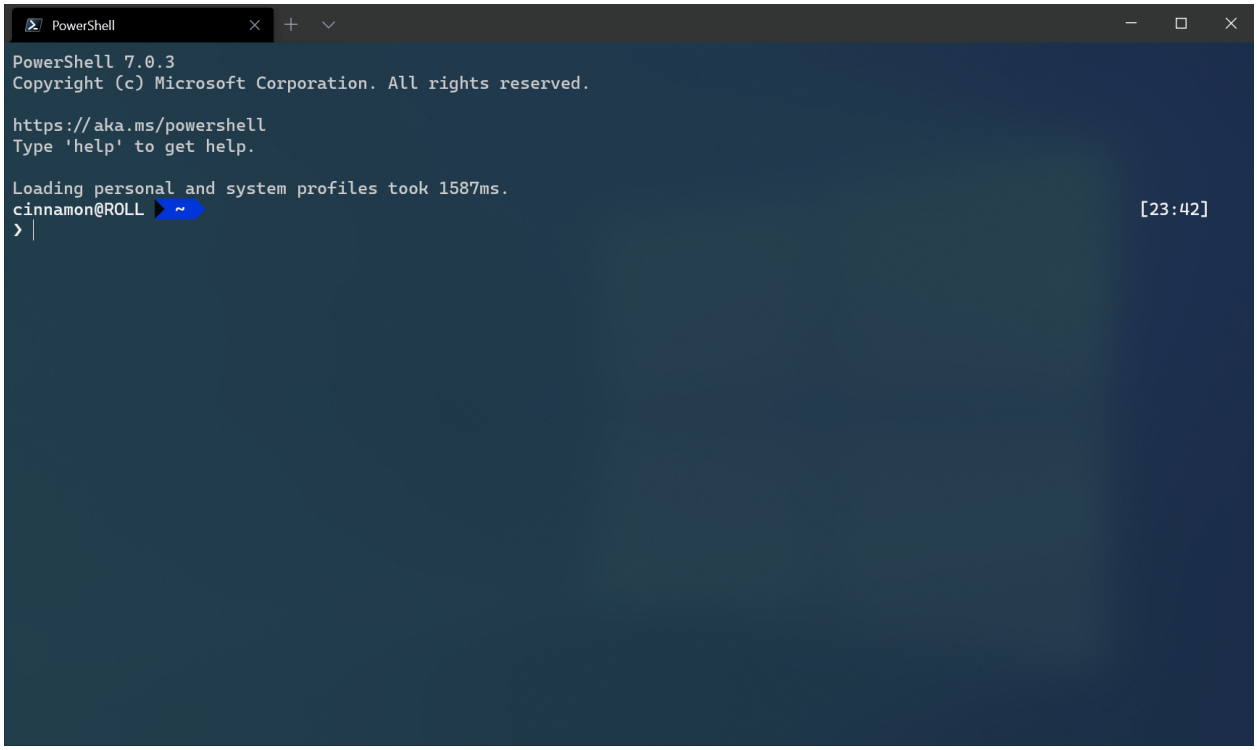
JSON

```
{
    "icon": null,
    "name": "Command Prompt",
    "command": { "action": "newTab", "profile": "Command Prompt" }
},
{
    "icon": "C:\\path\\to\\icon",
    "name": "PowerShell",
    "command": { "action": "newTab", "profile": "PowerShell" }
},
{
    "icon": null,
    "name": "Ubuntu",
    "command": { "action": "newTab", "profile": "Ubuntu" }
}
```

It's also possible to combine nested and iterable commands. For example, you can combine the three "new tab" commands above under a single "New tab" entry in the command palette, as shown in the image above, in the following way:

JSON

```
{
    "name": "New tab",
    "commands": [
        {
            "iterateOn": "profiles",
            "icon": "${profile.icon}",
            "name": "${profile.name}",
            "command": { "action": "newTab", "profile": "${profile.name}" }
        }
    ]
}
```



Hiding a command

If you would like to keep a command in your key bindings list but not have it appear in the command palette, you can hide it by setting its `name` to `null`. The example below hides the "New tab" action from the command palette.

JSON

```
{ "name": null, "command": "newTab", "keys": "ctrl+shift+t" }
```

Collaborate with us on GitHub

Windows Terminal feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

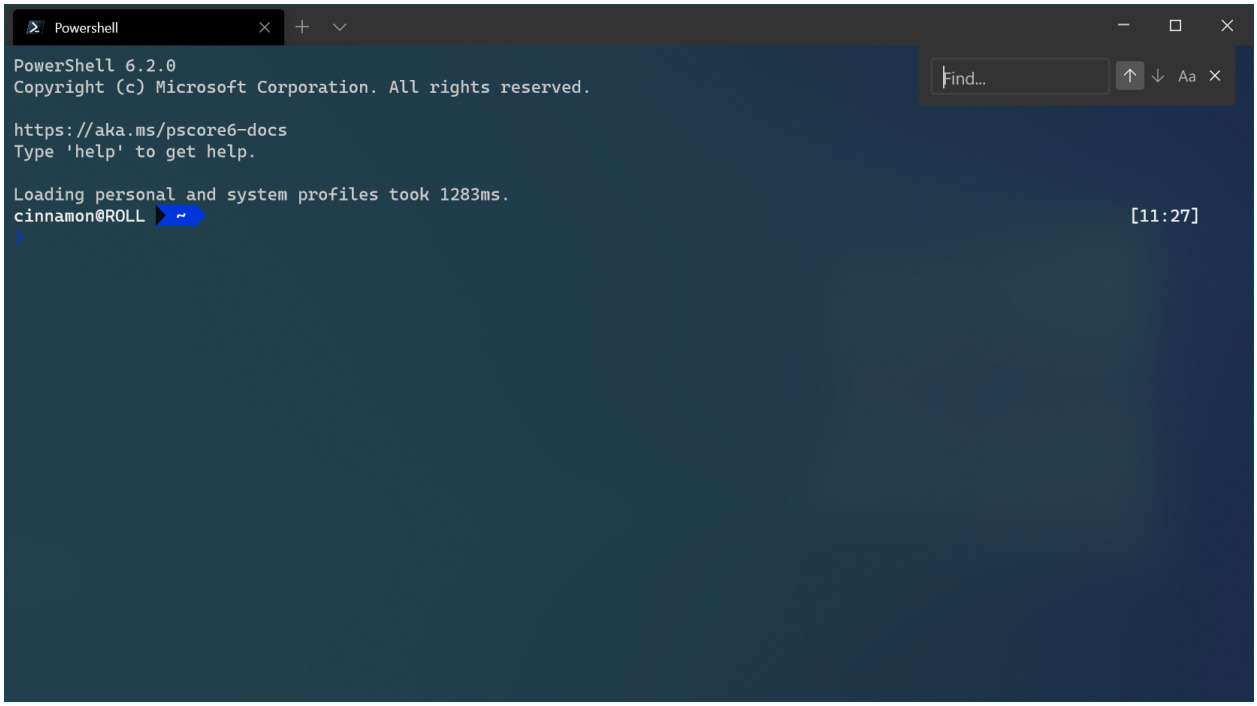
How to search in Windows Terminal

Article • 02/02/2024

Windows Terminal comes with a search feature that allows you to look through the text buffer for a specific keyword. This is useful when trying to find a command you had run before or for a specific file name.

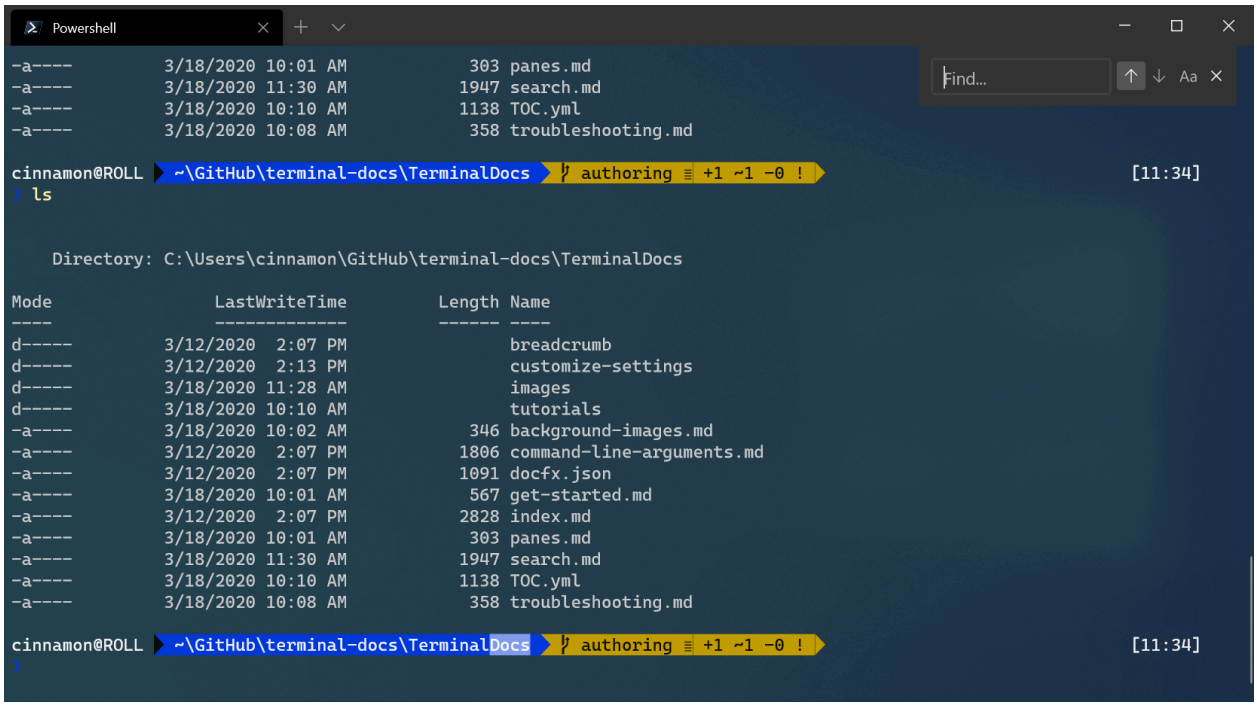
Using search

By default, you can open the search dialog by typing `Ctrl+Shift+F`. Once opened, you can type the keyword you're looking for into the text box and hit `Enter` to search.



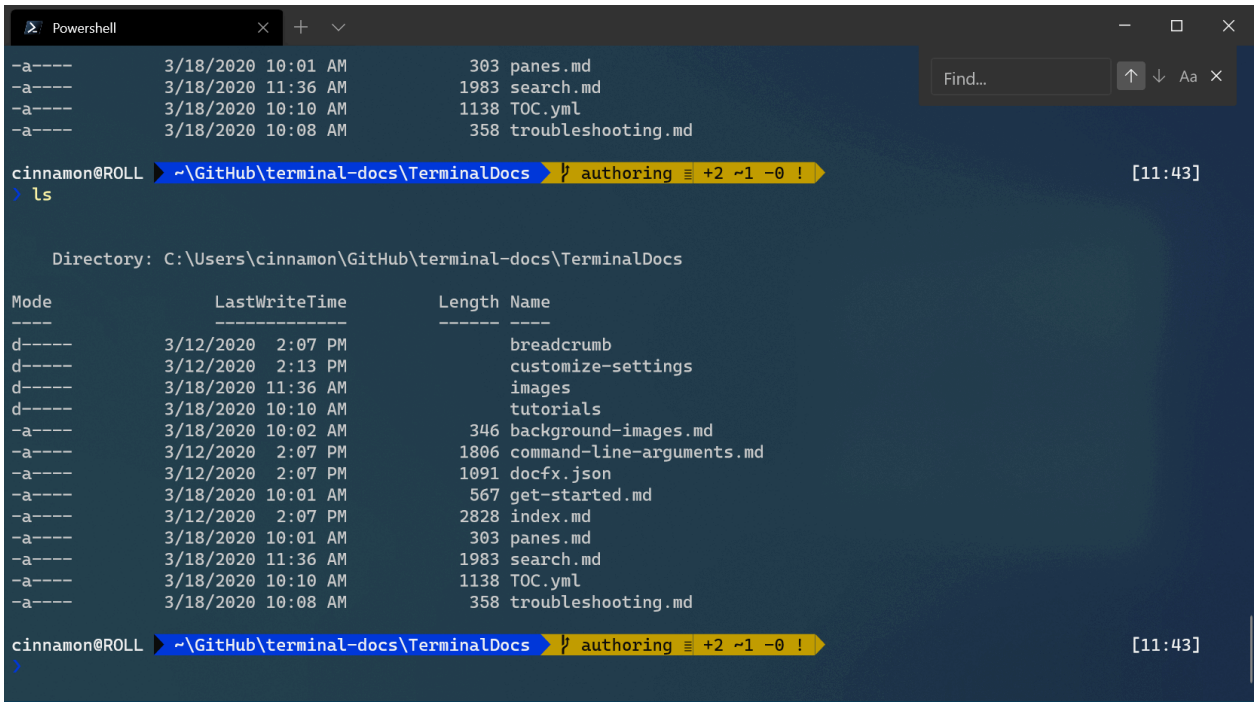
Directional search

The terminal will default to searching from the bottom to the top of the text buffer. You can change the search direction by selecting one of the arrows in the search dialog.



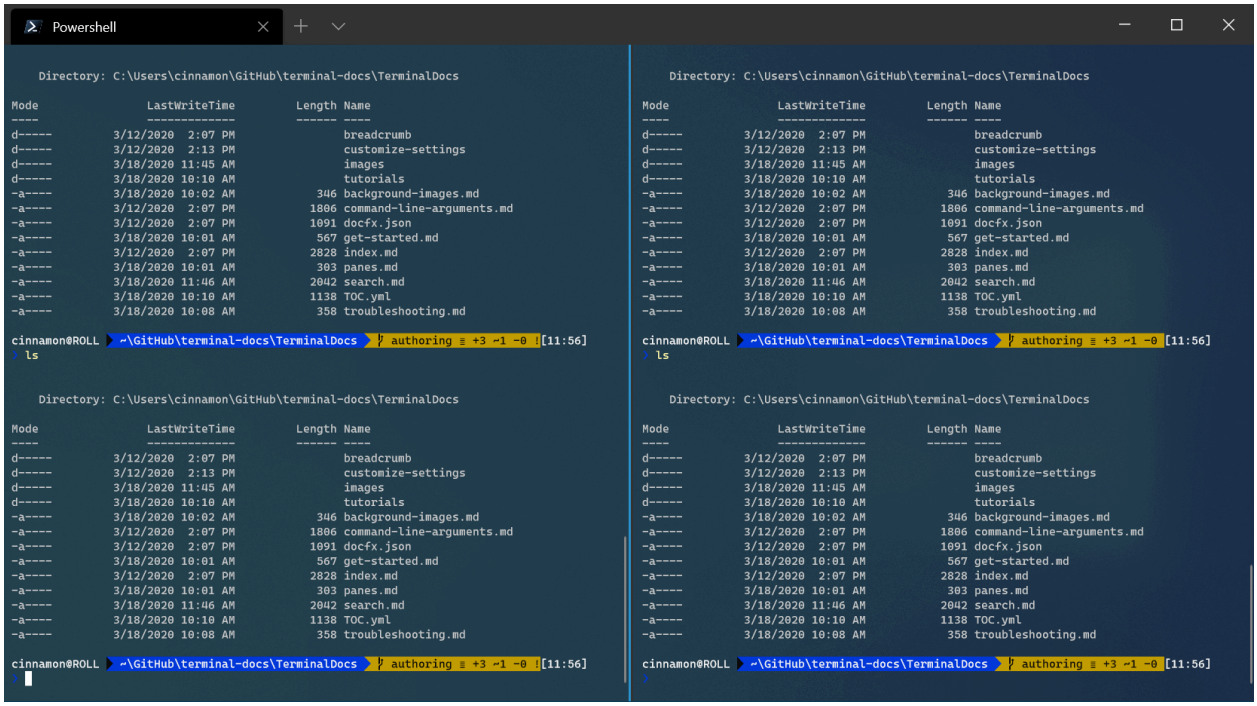
Case match search

If you'd like to narrow down your search results, you can add case matching as an option in your search. You can toggle case matching by selecting the case match button, and the results that appear will only match the keyword entered with its specific letter casing.



Searching within panes

The search dialog works with panes as well. When focused on a pane, you can open the search dialog and it will appear on the upper-right of that pane. Then any keyword you enter will only show results found within that pane.



Customize the search key binding

You can open the search dialog with any key binding (shortcut key combination) that you prefer. To change the search key binding, open your settings.json file and search for the `find` command. By default, this command is set to `Ctrl+Shift+F`.

JSON

```
// Press ctrl+shift+f to open the search box
        { "command": "find", "keys": "ctrl+shift+f" },
```

For example, you can change `"ctrl+shift+f"` to `"ctrl+f"`, so when typing `Ctrl+F`.

To learn more about key bindings, visit the Actions page.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Selecting text in Windows Terminal

Article • 02/02/2024

Selecting text is straightforward in Windows Terminal, but there are a lot of additional features in this space that make it even better.

Mouse support

Left-click and drag your mouse to create a selection. Double-click expands the selection by word, whereas triple-click expands by line.

If you are holding the `Alt` key, you will create a block selection (as opposed to a line selection). Block selections create a rectangular region that do not wrap to the end of the line.

If you are holding the `Shift` key, you can explicitly expand the selection to a specific point on the terminal without the need to click and drag.

Once you have a selection present, you have a few options. A single left-click will clear your selection. If you actually want to use it, you can right-click to copy the selected text to your clipboard and clear the selection. If you right-click again, the contents of your clipboard will then be pasted into the terminal.

> **Note**
>
> Windows Terminal supports mouse input in Windows Subsystem for Linux (WSL) applications as well as Windows applications that use virtual terminal (VT) input. This means applications such as **tmux** and **Midnight Commander** will recognize when you select items in the terminal window. If an application is in mouse mode, you can hold down `Shift` to make a selection instead of sending VT input.

Keyboard support

You can create a selection by using the `selectAll` or `markMode` actions. The `selectAll` action selects all the text in the buffer. The `markMode` action toggles a special mode where a selection is created at the cursor's position in the terminal. When in mark mode, you can use the following non-configurable key bindings to move the cursor around:

Expand table

Key binding	Result
Arrow keys	Move by character in the specified direction
`Ctrl+Left`	Move to the beginning of the previous or existing word
`Ctrl+Right`	Move to the end of the next or existing word
`Home`	Move to the beginning of the line
`End`	Move to the end of the line
`Pgup`	Move up by a page (viewport)
`Pgdn`	Move down by a page (viewport)
`Ctrl+Home`	Move to the beginning of the buffer
`Ctrl+End`	Move to the end of the buffer
`Enter`	Copy the active selection

When in mark mode, you can use `Tab` or `Shift+Tab` to navigate to the next or previous hyperlink in the buffer. Windows Terminal can automatically detect hyperlinks if experimental.detectUrls is enabled.

Regardless of being in mark mode, you can expand an existing selection using the following non-configurable key bindings:

Expand table

Key binding	Result
`Shift` + Arrow keys	Expand by character in the specified direction
`Ctrl+Shift+Left`	Expand to the beginning of the previous or existing word
`Ctrl+Shift+Right`	Expand to the end of the next or existing word
`Shift+Home`	Expand to the beginning of the line
`Shift+End`	Expand to the end of the line
`Shift+Pgup`	Expand up by a page (viewport)
`Shift+Pgdn`	Expand down by a page (viewport)
`Ctrl+Shift+Home`	Expand to the beginning of the buffer

Key binding	Result
`Ctrl+Shift+End`	Expand to the end of the buffer

Use the `toggleBlockSelection` action to transform the existing selection into a block selection.

Any selection created or modified by the keyboard also displays selection markers to indicate which end of the selection is actively being moved. You can use the `switchSelectionEndpoint` action to begin moving the other end of the selection.

Once you have a selection present, you have a few options. You can use the `ESC` key to clear the selection. Alternatively, most key input clears the selection and passes the key event directly to the underlying shell. If you actually want to use the selected text, you can use the `copy` action to copy it to your clipboard.

Copying selected text

As mentioned above, selected text can be copied with a right-click or the `copy` action. However, there are a number of settings regarding copying text that you can customize:

- Copying formatted text
  - You can use the `copyFormatting` global setting to also copy the formatting of the selected text itself to the clipboard. This allows you to copy the terminal's font information such as foreground color, background color, and font.
  - If you want to limit copying the formatting to certain key bindings (or commands), you can modify the `copyFormatting` parameter on a `copy` action.
- Copying without dismissing text selection
  - You can copy text without dismissing the text selection by setting the `dismissSelection` parameter in the `copy` action to `false`.
- Copying as a single line
  - You can copy text as a single line using the `singleLine` parameter in the `copy` action.
- Removing trailing whitespace from block selections
  - You can remove the trailing whitespace from a block selection using the `trimBlockSelection` global setting.

You can also use the `copyOnSelect` global setting to have newly selected text automatically copied to your clipboard. With this setting enabled, if a selection is present, right-clicking the terminal copies and pastes the selected text to your terminal.

> **Note**

If `copyOnSelect` is enabled, modifying the selection using the keyboard does not automatically copy the newly selected text. You will need to manually copy the text using the `copy` action or by right-clicking the terminal.

Customizing the appearance of selections

Color schemes let you customize the selection color using the `selectionBackground` property in a color scheme. Alternatively, you can override the selection color for a specific profile using the `selectionBackground` profile setting.

Customizing word delimiters

As mentioned above, double-clicking and using `Ctrl+Shift` + Arrow keys (or `Ctrl` + Arrow keys when in mark mode) allow you to navigate by word. However, words can be separated by more than just whitespace. You can customize these word boundaries using the `wordDelimiters` global setting.

Panes in Windows Terminal

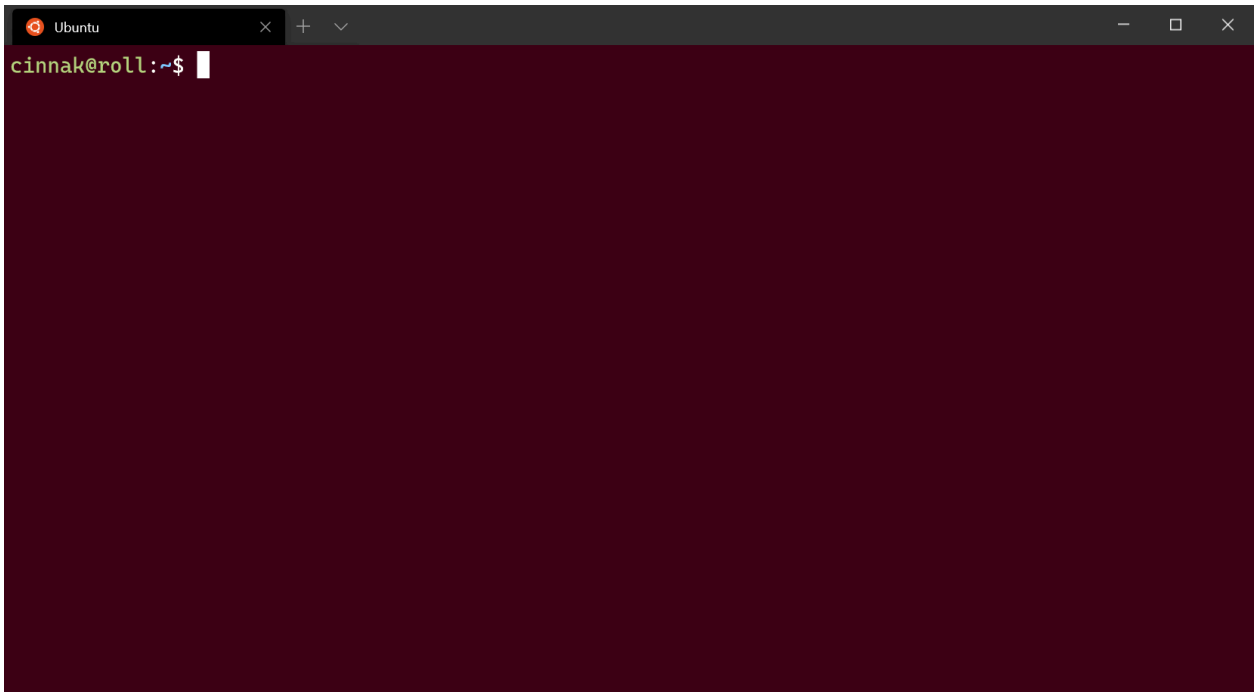
Article • 09/28/2023

Panes give you the ability to run multiple command-line applications next to each other within the same tab. This minimizes the need to switch between tabs and lets you see multiple prompts at once.

Creating a new pane

Using the keyboard

You can either create a new vertical or horizontal pane in Windows Terminal. Splitting vertically will open a new pane to the right of the focused pane and splitting horizontally will open a new pane below the focused pane. Using directional splits `up`, `right`, `down`, or `left` gives more options for where the new pane can go. `right` and `down` are equivalent to `vertical` and `horizontal`, whereas `up` and `left` allow you to put the new pane above and to the left of the focused pane respectively. To create a new vertical pane of your default profile, you can press the `Alt+Shift++` key combination. For a horizontal pane of your default profile, you can use `Alt+Shift+-`.



Configuration: Raspberry Ubuntu

If you would like to change these key bindings, you can create new ones using the `splitPane` action and `vertical`, `horizontal`, `up`, `right`, `down`, `left`, or `auto` values for the `split` property in your profiles.json file. The `auto` method will choose the direction

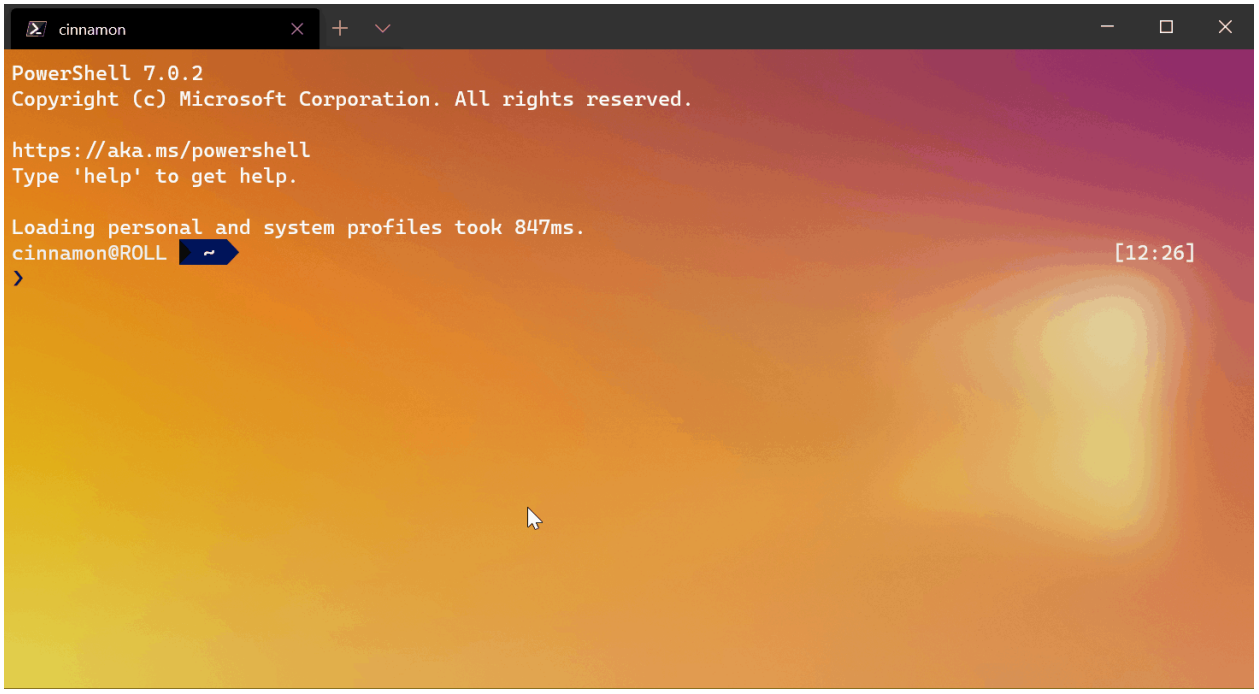
that gives you the squarest panes. To learn more about key bindings, visit the Actions page.

JSON

```
{ "command": { "action": "splitPane", "split": "vertical" }, "keys":
"alt+shift+plus" },
{ "command": { "action": "splitPane", "split": "horizontal" }, "keys":
"alt+shift+-" },
{ "command": { "action": "splitPane", "split": "auto" }, "keys":
"alt+shift+d" },
{ "command": { "action": "splitPane", "split": "up" } },
{ "command": { "action": "splitPane", "split": "right" } },
{ "command": { "action": "splitPane", "split": "down" } },
{ "command": { "action": "splitPane", "split": "left" } },
```

Using the new tab button and dropdown menu

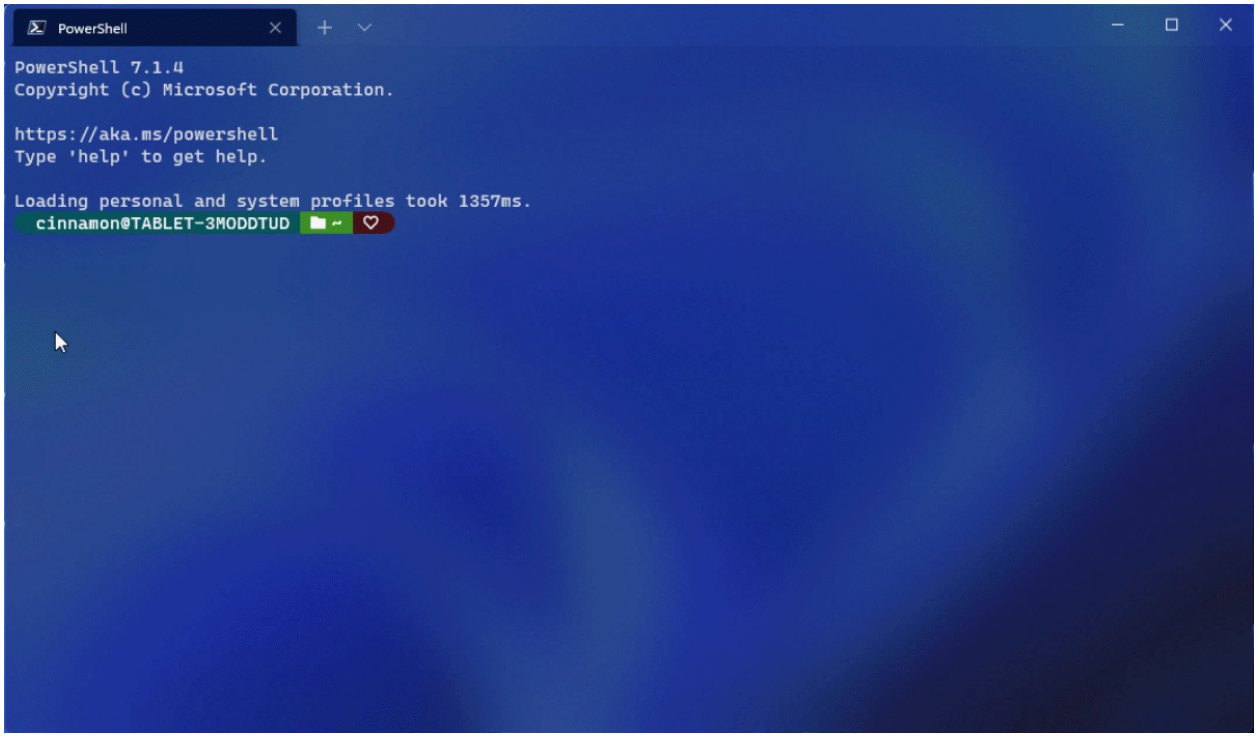
If you'd like to open a new pane of your default profile, you can hold the `Alt` key and click the new tab button. If you'd like to open a new pane through the dropdown menu, you can hold `Alt` and click on your desired profile. Both of these options will `auto` split the active window or pane into a new pane of the selected profile. The `auto` split mode splits in the direction that has the longest edge to create a pane.



Using the tab context menu

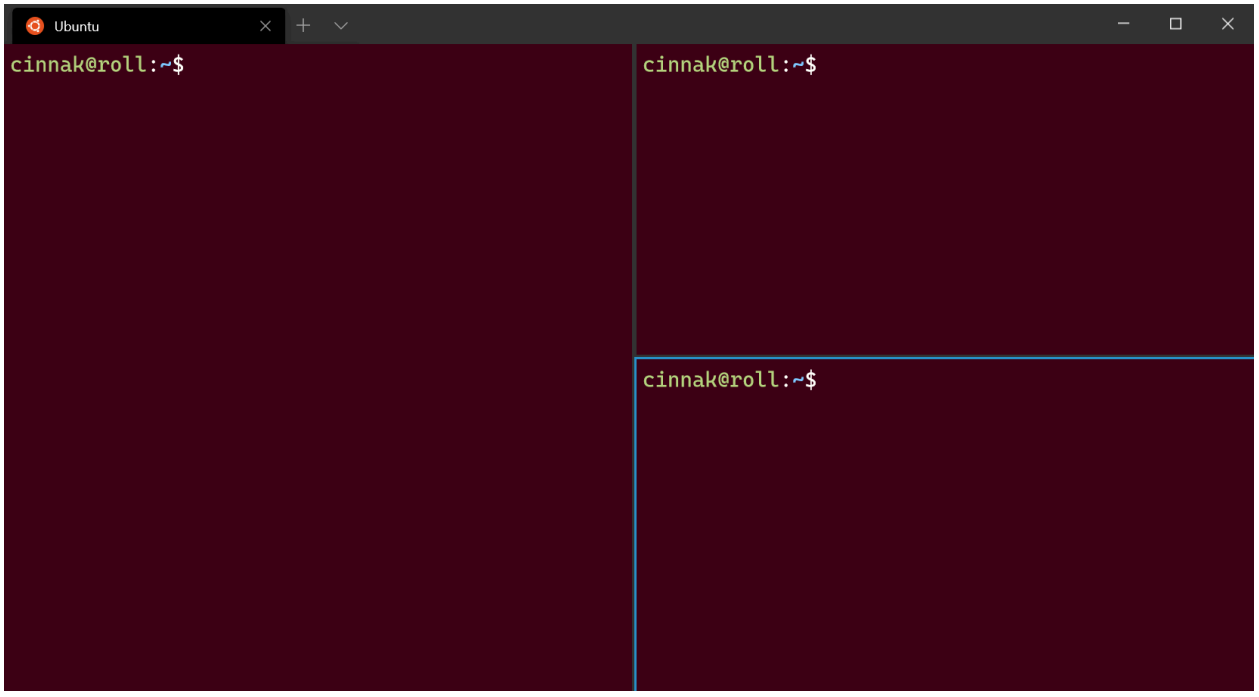
If you'd like to open a new pane of a profile that is already open in your terminal, you can right click on the tab and click Split Tab. This will duplicate the focused pane in the

current tab.



Switching between panes

The terminal allows you to navigate between panes by using the keyboard. If you hold the `Alt` key, you can use your arrow keys to move your focus between panes. You can identify which pane is in focus by the accent color border surrounding it. Note that this accent color is set in your Windows color settings.



You can customize this by adding key bindings for the `moveFocus` command and setting the `direction` to either `down`, `left`, `right`, or `up`. Additionally, `direction` can be

`previous` for the last used pane, or `previousInOrder` or `nextInOrder` for navigation by tree order, or `first` for the first pane. Lastly, you can navigate up the tree using the `parent` direction to select multiple panes, and then `child` to select fewer panes and move back down the tree. When multiple panes are selected you can perform actions like if you had a single pane focused.

JSON

```
{ "command": { "action": "moveFocus", "direction": "down" }, "keys":
"alt+down" },
{ "command": { "action": "moveFocus", "direction": "left" }, "keys":
"alt+left" },
{ "command": { "action": "moveFocus", "direction": "right" }, "keys":
"alt+right" },
{ "command": { "action": "moveFocus", "direction": "up" }, "keys": "alt+up"
},
{ "command": { "action": "moveFocus", "direction": "previous" } },
{ "command": { "action": "moveFocus", "direction": "previousInOrder" } },
{ "command": { "action": "moveFocus", "direction": "nextInOrder" } },
{ "command": { "action": "moveFocus", "direction": "first" } },
{ "command": { "action": "moveFocus", "direction": "parent" } },
{ "command": { "action": "moveFocus", "direction": "child" } }
```

Swapping panes

Once two panes have been created, you can swap their positions in the terminal.

The `swapPane` command can be customized using the same navigation `direction`s as `moveFocus`, except for `parent` and `child`. These commands will swap the positions of the currently focused pane and its neighbor according to `direction`.

JSON

```
{ "command": { "action": "swapPane", "direction": "down" } },
{ "command": { "action": "swapPane", "direction": "left" } },
{ "command": { "action": "swapPane", "direction": "right" } },
{ "command": { "action": "swapPane", "direction": "up" } },
{ "command": { "action": "swapPane", "direction": "previous" } },
{ "command": { "action": "swapPane", "direction": "previousInOrder" } },
{ "command": { "action": "swapPane", "direction": "nextInOrder" } },
{ "command": { "action": "swapPane", "direction": "first" } }
```

Moving panes

Panes can also be moved between tabs, creating a new tab if one with the target index does not exist.

The key bindings for the `movePane` command can be customized for moving panes to (zero-indexed) tabs according to their order.

JSON

```
{ "command": { "action": "movePane", "index": 0 } },
{ "command": { "action": "movePane", "index": 1 } },
{ "command": { "action": "movePane", "index": 2 } },
{ "command": { "action": "movePane", "index": 3 } },
{ "command": { "action": "movePane", "index": 4 } },
{ "command": { "action": "movePane", "index": 5 } },
{ "command": { "action": "movePane", "index": 6 } },
{ "command": { "action": "movePane", "index": 7 } },
{ "command": { "action": "movePane", "index": 8 } }
```

Changing split orientation

After two panes on a tab have been created, the split orientation of those panes can be switched between `vertical` and `horizontal` with the `toggleSplitOrientation` command.

JSON

```
{ "command": "toggleSplitOrientation" }
```

Swapping panes (Preview)

Once two panes have been created, you can swap their positions in the terminal.

The `swapPane` command can be customized using the same navigation `direction`s as `moveFocus`. These commands will swap the positions of the currently focused pane and its neighbor according to `direction`.

JSON

```
{ "command": { "action": "swapPane", "direction": "down" } },
{ "command": { "action": "swapPane", "direction": "left" } },
{ "command": { "action": "swapPane", "direction": "right" } },
{ "command": { "action": "swapPane", "direction": "up" } },
{ "command": { "action": "swapPane", "direction": "previous" } },
```

```
{ "command": { "action": "swapPane", "direction": "previousInOrder" } },
{ "command": { "action": "swapPane", "direction": "nextInOrder" } }
```

Moving panes (Preview)

Panes can also be moved between tabs, creating a new tab if one with the target index does not exist.

The key bindings for the `movePane` command can be customized for moving panes to (zero-indexed) tabs according to their order.

JSON

```
{ "command": { "action": "movePane", "index": 0 } },
{ "command": { "action": "movePane", "index": 1 } },
{ "command": { "action": "movePane", "index": 2 } },
{ "command": { "action": "movePane", "index": 3 } },
{ "command": { "action": "movePane", "index": 4 } },
{ "command": { "action": "movePane", "index": 5 } },
{ "command": { "action": "movePane", "index": 6 } },
{ "command": { "action": "movePane", "index": 7 } },
{ "command": { "action": "movePane", "index": 8 } }
```

Changing split orientation (Preview)

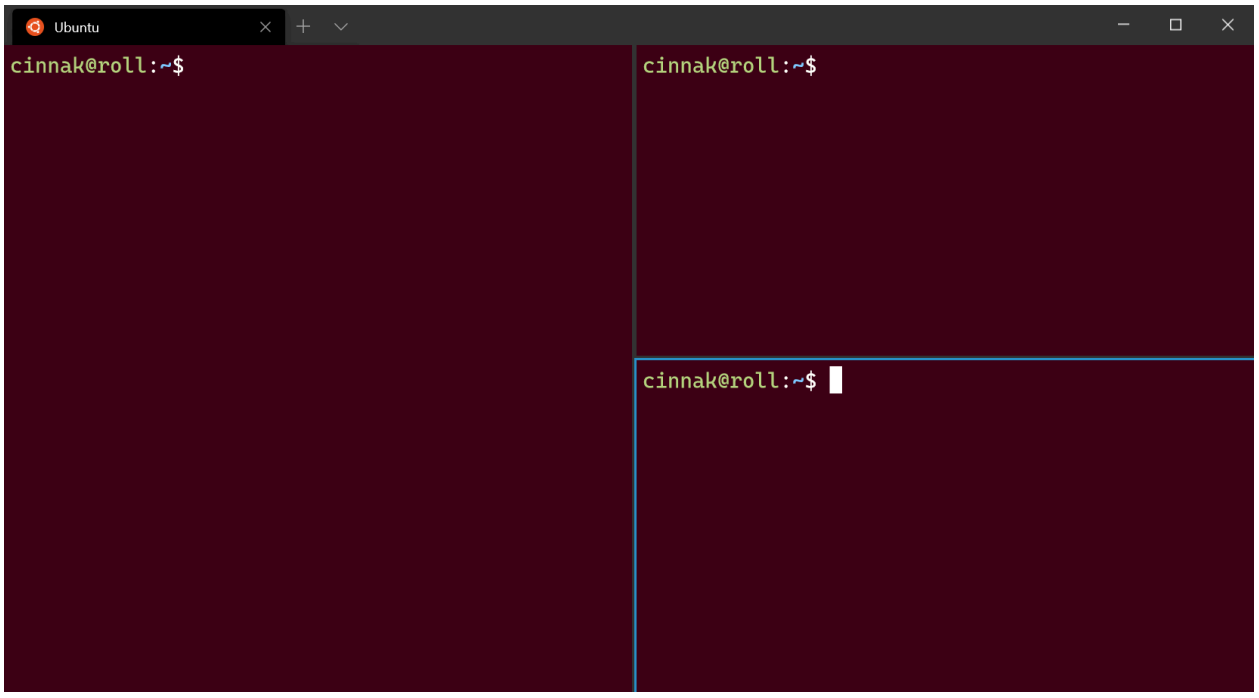
After two panes on a tab have been created, the split orientation of those panes can be switched between `vertical` and `horizontal` with the `toggleSplitOrientation` command.

JSON

```
{ "command": "toggleSplitOrientation" }
```

Resizing a pane

You can adjust the size of your panes by holding `Alt+Shift` and using your arrow keys to resize the focused pane.



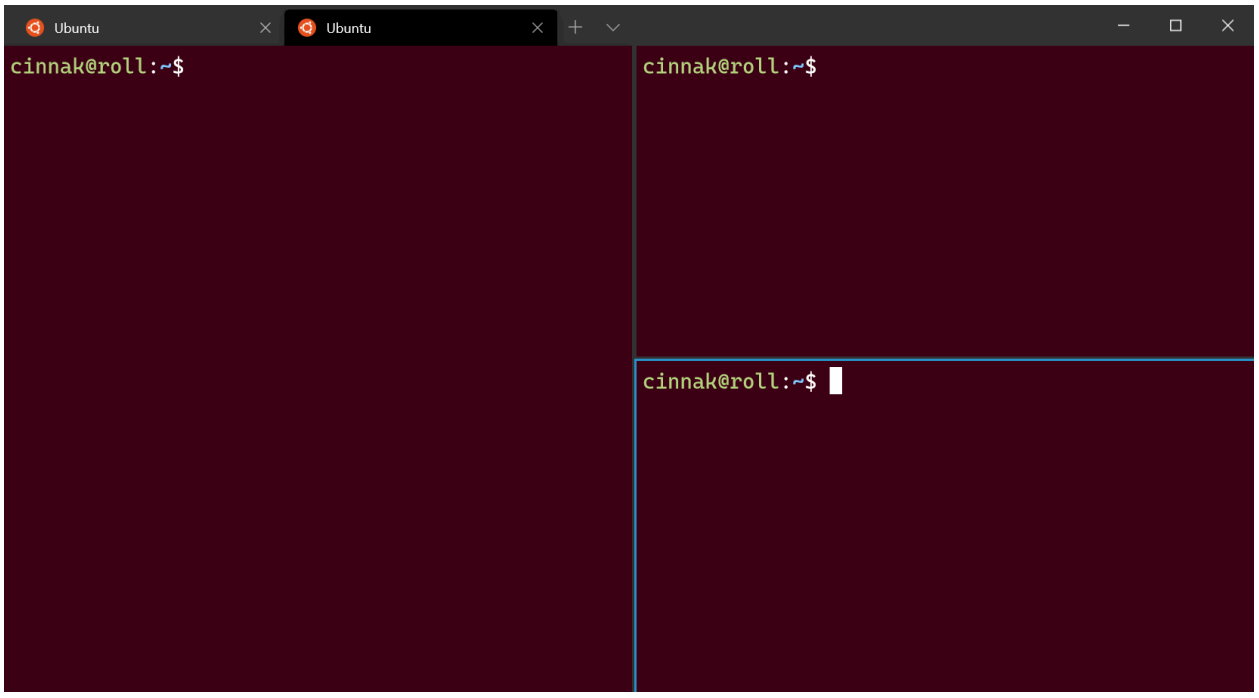
To customize this key binding, you can add new ones using the `resizePane` action and setting the `direction` to either `down`, `left`, `right`, or `up`.

JSON

```
{ "command": { "action": "resizePane", "direction": "down" }, "keys":
"alt+shift+down" },
{ "command": { "action": "resizePane", "direction": "left" }, "keys":
"alt+shift+left" },
{ "command": { "action": "resizePane", "direction": "right" }, "keys":
"alt+shift+right" },
{ "command": { "action": "resizePane", "direction": "up" }, "keys":
"alt+shift+up" }
```

Closing a pane

You can close the focused pane by typing `Ctrl+Shift+W`. If you only have one pane, `Ctrl+Shift+W` will close the tab. As always, closing the last tab will close the window.



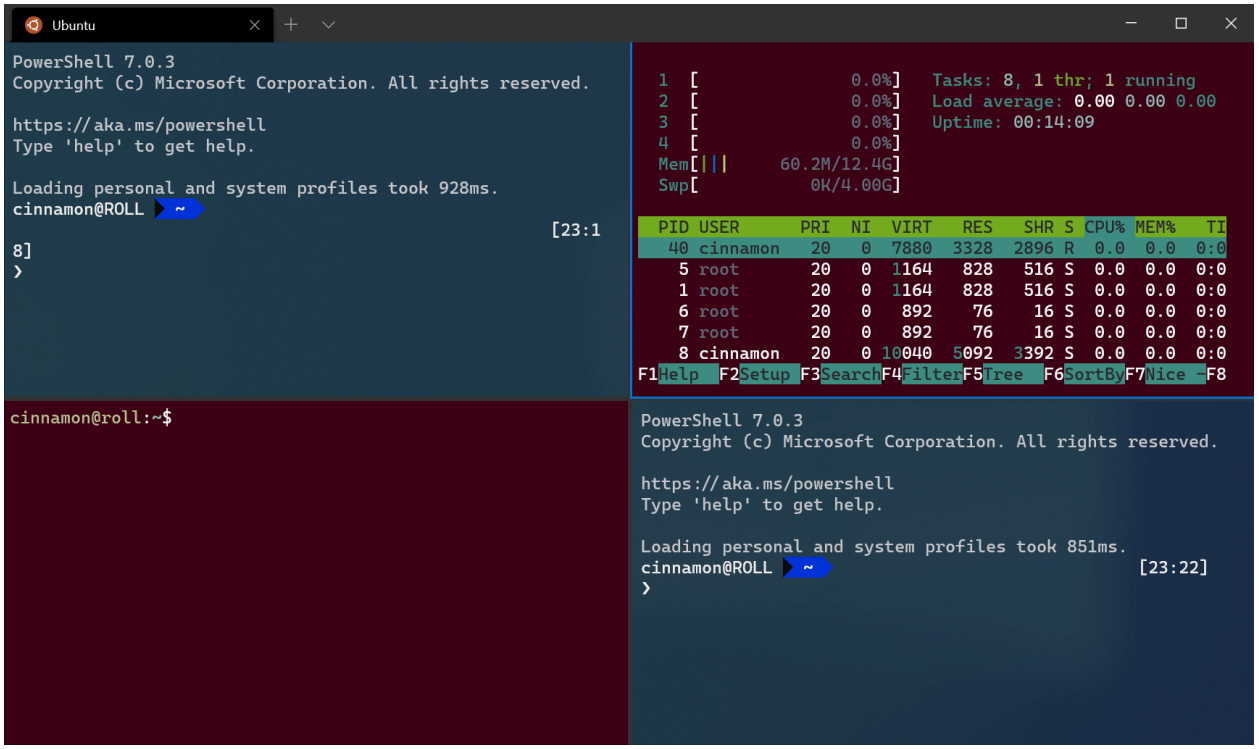
You can change which keys close the pane by adding a key binding that uses the `closePane` command.

JSON

```
{ "command": "closePane", "keys": "ctrl+shift+w" }
```

Zooming a pane

You can zoom the focused pane to fill the entire contents of the window.



> **Note**
>
> The menu in the above gif is the Command Palette, which can be opened with `Ctrl+Shift+P` by default.

This can be done by using the `togglePaneZoom` command.

```
JSON
{ "command": "togglePaneZoom" }
```

> **Note**
>
> The `togglePaneZoom` action is not bound to any keys by default, but it can be accessed through the command palette, which is bound to `Ctrl+Shift+P` by default.

Marking a pane as read-only

You can mark a pane as read-only, which will prevent input from going into the text buffer. If you attempt to close or input text into a read-only pane, the terminal will display a popup warning instead.

You can toggle read-only mode on a pane with the `toggleReadOnlyMode` command.

```
JSON
{ "command": "toggleReadOnlyMode" }
```

You can enable read-only mode on a pane. This works similarly to toggling, however, will not switch state if triggered again.

Command name: `enableReadOnlyMode`

Default bindings:

```
JSON
{ "command": "enableReadOnlyMode" }
```

You can disable read-only mode on a pane. This works similarly to toggling, however, will not switch state if triggered again.

Command name: `disableReadOnlyMode`

Default bindings:

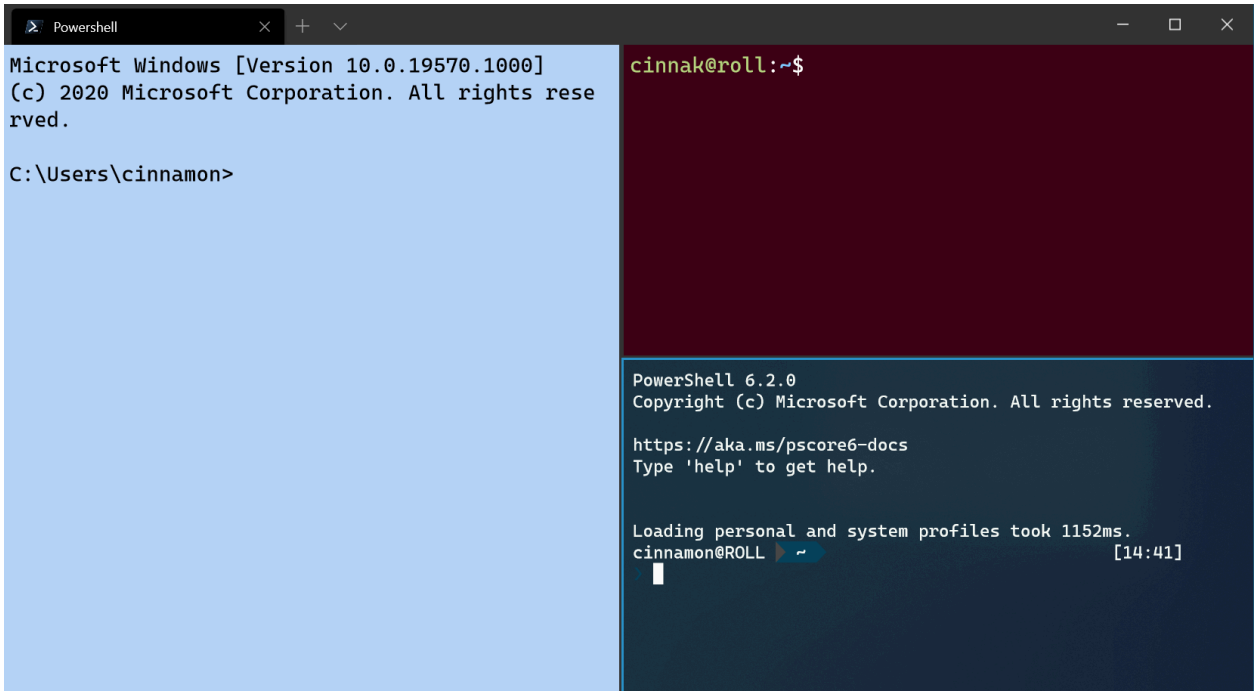
```JSON
{ "command": "disableReadOnlyMode" }
```

Customizing panes using key bindings

You can customize what opens inside a new pane depending on your custom key bindings.

Duplicating a pane

The terminal allows you to duplicate the focused pane's profile into another pane.



This can be done by adding the `splitMode` property with `duplicate` as the value to a `splitPane` key binding.

```JSON
{ "command": { "action": "splitPane", "split": "auto", "splitMode": "duplicate" }, "keys": "alt+shift+d" }
```

New terminal arguments

When opening a new pane or tab with a key binding, you can specify which profile is used by including the profile's name, guid, or index. If none are specified, the default profile is used. This can be done by adding `profile` or `index` as an argument to a `splitPane` or `newTab` key binding. Note that indexing starts at 0.

JSON

```
{ "command": { "action": "splitPane", "split": "vertical", "profile":
"profile1" }, "keys": "ctrl+a" },
{ "command": { "action": "splitPane", "split": "vertical", "profile": "
{00000000-0000-0000-0000-000000000000}" }, "keys": "ctrl+b" },
{ "command": { "action": "newTab", "index": 0 }, "keys": "ctrl+c" }
```

Additionally, you can override certain aspects of the profile such as the profile's command line executable, starting directory, or tab title. This can be accomplished by adding `commandline`, `startingDirectory`, and/or `tabTitle` to a `splitPane` or `newTab` key binding.

JSON

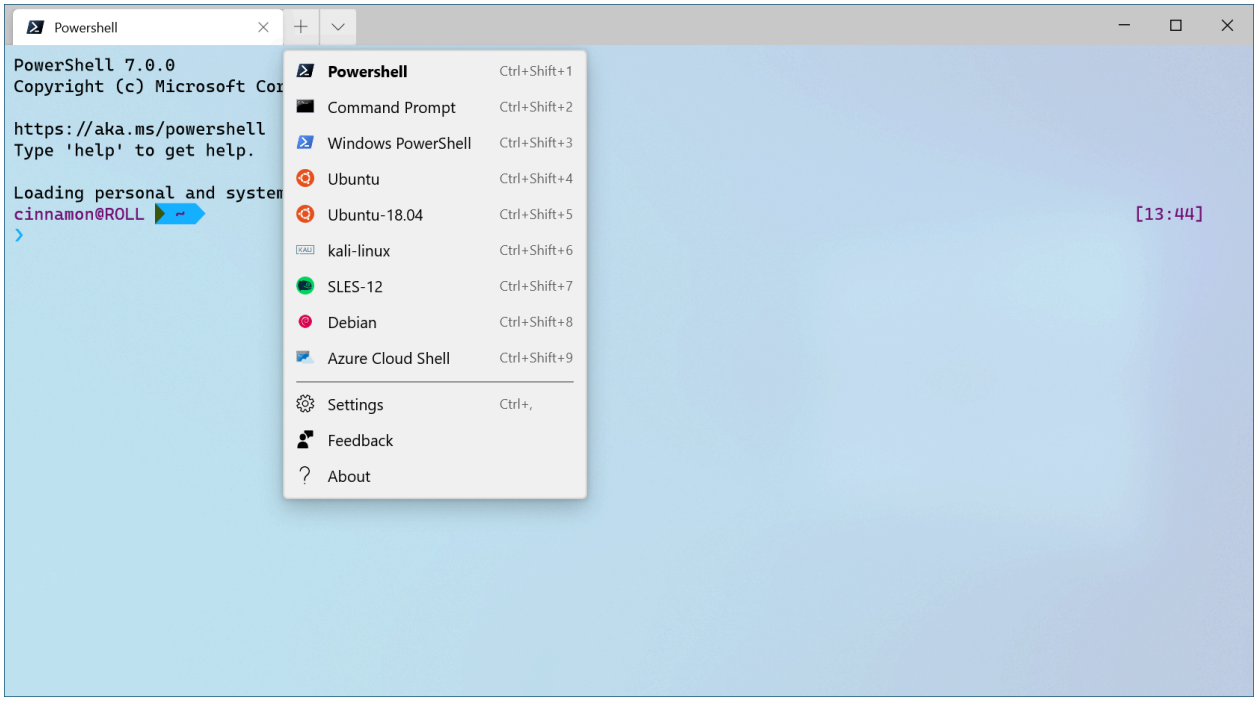
```
{ "command": { "action": "splitPane", "split": "auto", "profile":
"profile1", "commandline": "foo.exe" }, "keys": "ctrl+a" },
{ "command": { "action": "newTab", "profile": "{00000000-0000-0000-0000-
000000000000}", "startingDirectory": "C:\\foo" }, "keys": "ctrl+b" },
{ "command": { "action": "newTab", "index": 0, "tabTitle": "bar",
"startingDirectory": "C:\\foo", "commandline": "foo.exe" }, "keys": "ctrl+c"
}
```

Dynamic profiles in Windows Terminal

Article • 05/29/2024

Windows Terminal will automatically create Windows Subsystem for Linux (WSL) and PowerShell profiles for you if you have these shells installed on your machine. This makes it easier for you to have all of your shells included in the terminal without having to locate their executable files. These profiles are generated with the `source` property, which tells the terminal where to locate the proper executable.

Upon installing the terminal, it will set PowerShell as your default profile. To learn how to change your default profile, visit the Startup page.



Configuration: Light Theme

Installing a new shell after installing Windows Terminal

Regardless of whether a new shell is installed before or after your terminal installation, the terminal will create a new profile for the newly installed shell.

Hide a profile

To hide a profile from your terminal dropdown menu, add the `hidden` property to the profile object in your settings.json file and set it to `true`.

JSON

```
"hidden": true
```

Prevent a profile from being generated

To prevent a dynamic profile from being generated, you can add the profile generator to the `disabledProfileSources` array in your global settings. More information on this setting can be found on the Global settings page.

JSON

```
"disabledProfileSources": ["Windows.Terminal.Wsl", "Windows.Terminal.Azure",
"Windows.Terminal.PowershellCore", "Windows.Terminal.SSH"]
```

Add a third party profile

If a 3rd party command line tool does not have a profile auto-generated into your settings.json file, you can add it manually. Below are profiles for a few common 3rd party tools for your reference.

Anaconda

Assuming that you've installed Anaconda into `%USERPROFILE%\Anaconda3`:

JSON

```
{
    "commandline": "cmd.exe /k
\"%USERPROFILE%\\Anaconda3\\Scripts\\activate.bat
%USERPROFILE%\\Anaconda3\"",
    "icon": "%USERPROFILE%\\Anaconda3\\Menu\\anaconda-navigator.ico",
    "name": "Anaconda3",
    "startingDirectory": "%USERPROFILE%"
}
```

cmder

Assuming that you've installed cmder into `%CMDER_ROOT%`:

JSON

```
{
    "commandline": "cmd.exe /k %CMDER_ROOT%\\vendor\\init.bat",
    "name": "cmder",
    "icon": "%CMDER_ROOT%\\icons\\cmder.ico",
    "startingDirectory": "%USERPROFILE%"
}
```

Cygwin

Assuming that you've installed Cygwin into `C:\cygwin64`:

JSON

```
{
    "name": "Cygwin",
    "commandline": "C:\\cygwin64\\bin\\bash --login -i",
    "icon": "C:\\cygwin64\\Cygwin.ico",
    "startingDirectory": "C:\\cygwin64\\bin"
}
```

> ![NOTE] The starting directory of Cygwin is set in order for the path to work. The default directory opened when starting Cygwin will be `$HOME` because of the `--login` flag.

Far Manager

Assuming that you've installed Far into `c:\Program Files\Far Manager`:

JSON

```
{
    "name": "Far",
    "commandline": "\"c:\\program files\\far manager\\far.exe\"",
    "startingDirectory": "%USERPROFILE%",
    "useAcrylic": false
},
```

Git Bash

Assuming that you've installed Git Bash into `C:\\Program Files\\Git`:

JSON

```
{
    "name": "Git Bash",
    "commandline": "C:\\Program Files\\Git\\bin\\bash.exe -li",
    "icon": "C:\\Program Files\\Git\\mingw64\\share\\git\\git-for-
windows.ico",
    "startingDirectory": "%USERPROFILE%"
}
```

Git Bash (WOW64)

Assuming that you've installed Git Bash into `C:\\Program Files (x86)\\Git`:

JSON

```
{
    "name": "Git Bash",
    "commandline": "%ProgramFiles(x86)%\\Git\\bin\\bash.exe -li",
    "icon": "%ProgramFiles(x86)%\\Git\\mingw32\\share\\git\\git-for-
windows.ico",
    "startingDirectory": "%USERPROFILE%"
}
```

MSYS2

Assuming that you've installed MSYS2 into `C:\\msys64`:

JSON

```
{
    "name": "MSYS2",
    "commandline": "C:\\msys64\\msys2_shell.cmd -defterm -no-start -
mingw64",
    "icon": "C:\\msys64\\msys2.ico",
    "startingDirectory": "C:\\msys64\\home\\user"
}
```

For more details, see the Terminals section of the MSYS2 documentation.

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

JSON fragment extensions in Windows Terminal

Article • 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

```
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.

Shell Integration

Article • 05/29/2024

Starting in Terminal 1.15 Preview, the Windows Terminal has started experimentally supporting some "shell integration" features. These features make the command-line easier to use. In earlier releases, we enabled shell to tell the Terminal what the current working directory is. Now, we've added support for more sequences to allow your shell to semantically describe parts of the terminal output as a "prompt", a "command", or "output". The shell can also tell the terminal whether a command succeeded or failed.

This is a guide to some of the shell integration features we've rolled out as of Terminal v1.18. We're planning on building even more features on top of these in the future, so we'd love to get some additional feedback on how folks using them.

> **Note**: As of Terminal 1.21, marks are now a stable feature. Prior to 1.21, marks were **only enabled for Preview builds of the Terminal**. If you're using a version of Terminal before 1.21, the `showMarksOnScrollbar` setting was named `experimental.showMarksOnScrollbar`, and `autoMarkPrompts` was named `experimental.autoMarkPrompts`.

How does this work?

Shell integration works by having the shell (or any command line application) write special "escape sequences" to the Terminal. These escape sequences aren't printed to the Terminal - instead, they provide bits of metadata the terminal can use to know more about what's going on in the application. By sticking these sequences into your shell's

prompt, you can have the shell continuously provide info to the terminal that only the shell knows.

For the following sequences:

- `OSC` is the string `"\x1b]"` - an escape character, followed by `]`
- `ST` is the "string terminator", and can be either `\x1b\` (an ESC character, followed by `\`) or `\x7` (the BEL character)
- Spaces are merely illustrative.
- Strings in `<>` are parameters that should be replaced by some other value.

The relevant supported shell integration sequences as of Terminal v1.18 are:

- `OSC 133 ; A ST` ("*FTCS_PROMPT*") - The start of a prompt.
- `OSC 133 ; B ST` ("*FTCS_COMMAND_START*") - The start of a commandline (READ: the end of the prompt).
- `OSC 133 ; C ST` ("*FTCS_COMMAND_EXECUTED*") - The start of the command output / the end of the commandline.
- `OSC 133 ; D ; <ExitCode> ST` ("*FTCS_COMMAND_FINISHED*") - the end of a command. `ExitCode` If `ExitCode` is provided, then the Terminal will treat `0` as "success" and anything else as an error. If omitted, the terminal will just leave the mark the default color.

How to enable shell integration marks

Supporting these features requires cooperation between your shell and the Terminal. You'll need to both enable settings in the Terminal to use these new features, as well as modify your shell's prompt.

To enable these features in the Terminal, you'll want to add the following to your settings:

JSON

```
"profiles":
{
    "defaults":
    {
        // Enable marks on the scrollbar
        "showMarksOnScrollbar": true,

        // Needed for both pwsh, CMD and bash shell integration
        "autoMarkPrompts": true,

        // Add support for a right-click context menu
```

```
        // You can also just bind the `showContextMenu` action
        "experimental.rightClickContextMenu": true,
    },
}
"actions":
[
    // Scroll between prompts
    { "keys": "ctrl+up",   "command": { "action": "scrollToMark",
"direction": "previous" }, },
    { "keys": "ctrl+down", "command": { "action": "scrollToMark",
"direction": "next" }, },

    // Add the ability to select a whole command (or its output)
    { "command": { "action": "selectOutput", "direction": "prev" }, },
    { "command": { "action": "selectOutput", "direction": "next" }, },

    { "command": { "action": "selectCommand", "direction": "prev" }, },
    { "command": { "action": "selectCommand", "direction": "next" }, },
]
```

How you enable these marks in your shell varies from shell to shell. Below are tutorials for CMD, PowerShell and Zsh.

PowerShell (`pwsh.exe`)

If you've never changed your PowerShell prompt before, you should check out about_Prompts first.

We'll need to edit your `prompt` to make sure we tell the Terminal about the CWD, and mark up the prompt with the appropriate marks. PowerShell also lets us include the error code from the previous command in the `133;D` sequence, which will let the terminal automatically colorize the mark based if the command succeeeded or failed.

Add the following to your PowerShell profile:

PowerShell

```
$Global:__LastHistoryId = -1

function Global:__Terminal-Get-LastExitCode {
  if ($? -eq $True) {
    return 0
  }
  $LastHistoryEntry = $(Get-History -Count 1)
  $IsPowerShellError = $Error[0].InvocationInfo.HistoryId -eq
$LastHistoryEntry.Id
  if ($IsPowerShellError) {
    return -1
  }
```

```
  return $LastExitCode
}

function prompt {

  # First, emit a mark for the _end_ of the previous command.

  $gle = $(__Terminal-Get-LastExitCode);
  $LastHistoryEntry = $(Get-History -Count 1)
  # Skip finishing the command if the first command has not yet started
  if ($Global:__LastHistoryId -ne -1) {
    if ($LastHistoryEntry.Id -eq $Global:__LastHistoryId) {
      # Don't provide a command line or exit code if there was no history
entry (eg. ctrl+c, enter on no command)
      $out += "`e]133;D`a"
    } else {
      $out += "`e]133;D;$gle`a"
    }
  }


  $loc = $($executionContext.SessionState.Path.CurrentLocation);

  # Prompt started
  $out += "`e]133;A$([char]07)";

  # CWD
  $out += "`e]9;9;`"$loc`"$([char]07)";

  # (your prompt here)
  $out += "PWSH $loc$('>' * ($nestedPromptLevel + 1)) ";

  # Prompt ended, Command started
  $out += "`e]133;B$([char]07)";

  $Global:__LastHistoryId = $LastHistoryEntry.Id

  return $out
}
```

Oh My Posh setup

Using oh-my-posh? You'll want to slightly modify the above, to stash away the original prompt, then add it back in the middle of the shell integration escape sequences.

pwsh

```
# initialize oh-my-posh at the top of your profile.ps1
oh-my-posh init pwsh --config "$env:POSH_THEMES_PATH\gruvbox.omp.json" |
Invoke-Expression
# then stash away the prompt() that oh-my-posh sets
$Global:__OriginalPrompt = $function:Prompt
```

```
function Global:__Terminal-Get-LastExitCode {
  if ($? -eq $True) { return 0 }
  $LastHistoryEntry = $(Get-History -Count 1)
  $IsPowerShellError = $Error[0].InvocationInfo.HistoryId -eq
$LastHistoryEntry.Id
  if ($IsPowerShellError) { return -1 }
  return $LastExitCode
}

function prompt {
  $gle = $(__Terminal-Get-LastExitCode);
  $LastHistoryEntry = $(Get-History -Count 1)
  if ($Global:__LastHistoryId -ne -1) {
    if ($LastHistoryEntry.Id -eq $Global:__LastHistoryId) {
      $out += "`e]133;D`a"
    } else {
      $out += "`e]133;D;$gle`a"
    }
  }
  $loc = $($executionContext.SessionState.Path.CurrentLocation);
  $out += "`e]133;A$([char]07)";
  $out += "`e]9;9;`"$loc`"$([char]07)";

  $out += $Global:__OriginalPrompt.Invoke(); # <-- This line adds the
original prompt back

  $out += "`e]133;B$([char]07)";
  $Global:__LastHistoryId = $LastHistoryEntry.Id
  return $out
}
```

Command Prompt

Command Prompt sources it's prompt from the `PROMPT` environment variable. CMD.exe reads `$e` as the `ESC` character. Unfortunately, CMD.exe doesn't have a way to get the return code of the previous command in the prompt, so we're not able to provide success / error information in CMD prompts.

You can change the prompt for the current CMD.exe instance by running:

Windows Command Prompt

```
PROMPT $e]133;D$e\$e]133;A$e\$e]9;9;$P$e\$P$G$e]133;B$e\
```

Or, you can set the variable from the commandline for all future sessions:

Windows Command Prompt

```
setx PROMPT $e]133;D$e\$e]133;A$e\$e]9;9;$P$e\$P$G$e]133;B$e\
```

These examples assume your current `PROMPT` is just `$P$G`. You can instead choose to wrap your current prompt with something like:

Windows Command Prompt

```
PROMPT $e]133;D$e\$e]133;A$e\$e]9;9;$P$e\%PROMPT%$e]133;B$e\
```

Bash

You can add the following to the end of your `~/.bashrc` to enable shell integration in bash:

Bash

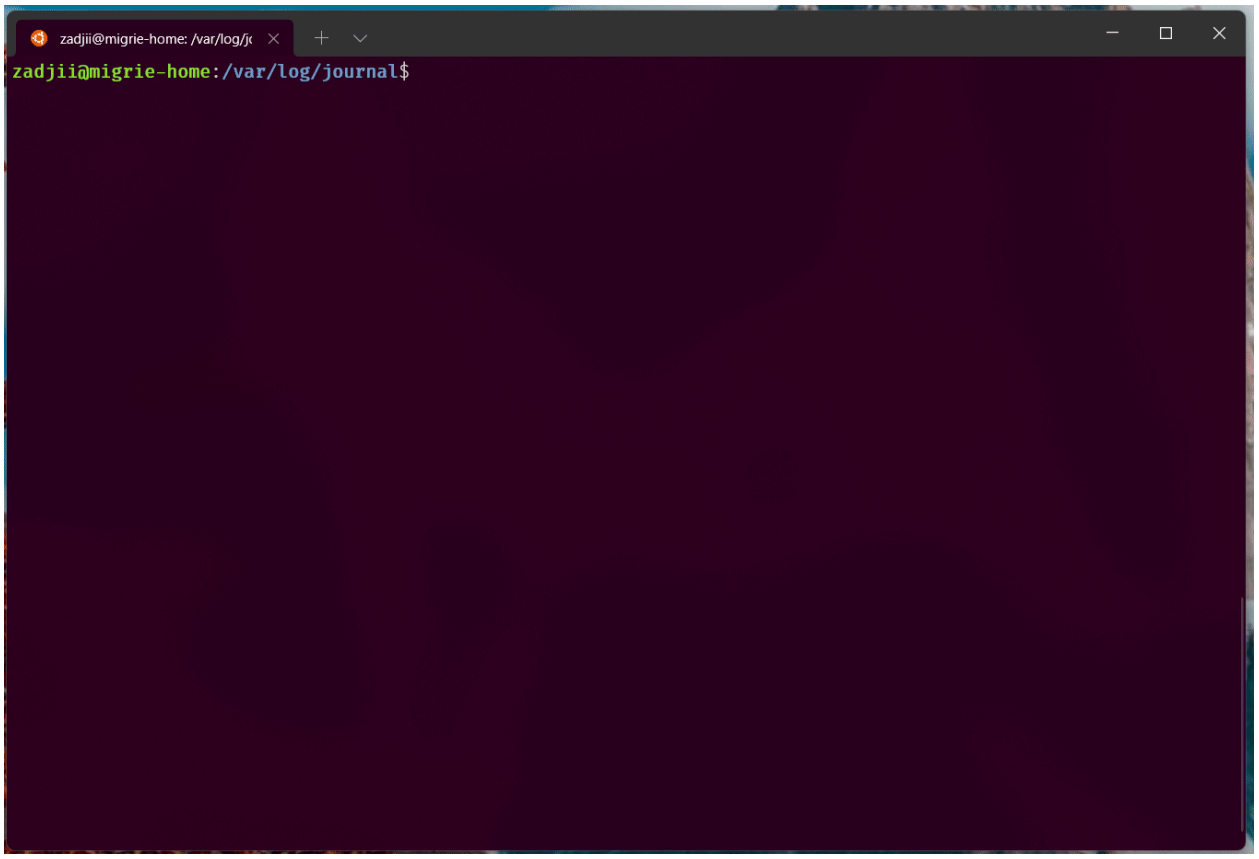
```
PS1="\[\033]133;D;\007\]\[\033]133;A;\007\]$PS1\[\033]133;B;\007\]"
```

This will wrap your existing `$PS1` with the necessary sequences to enable shell integration.

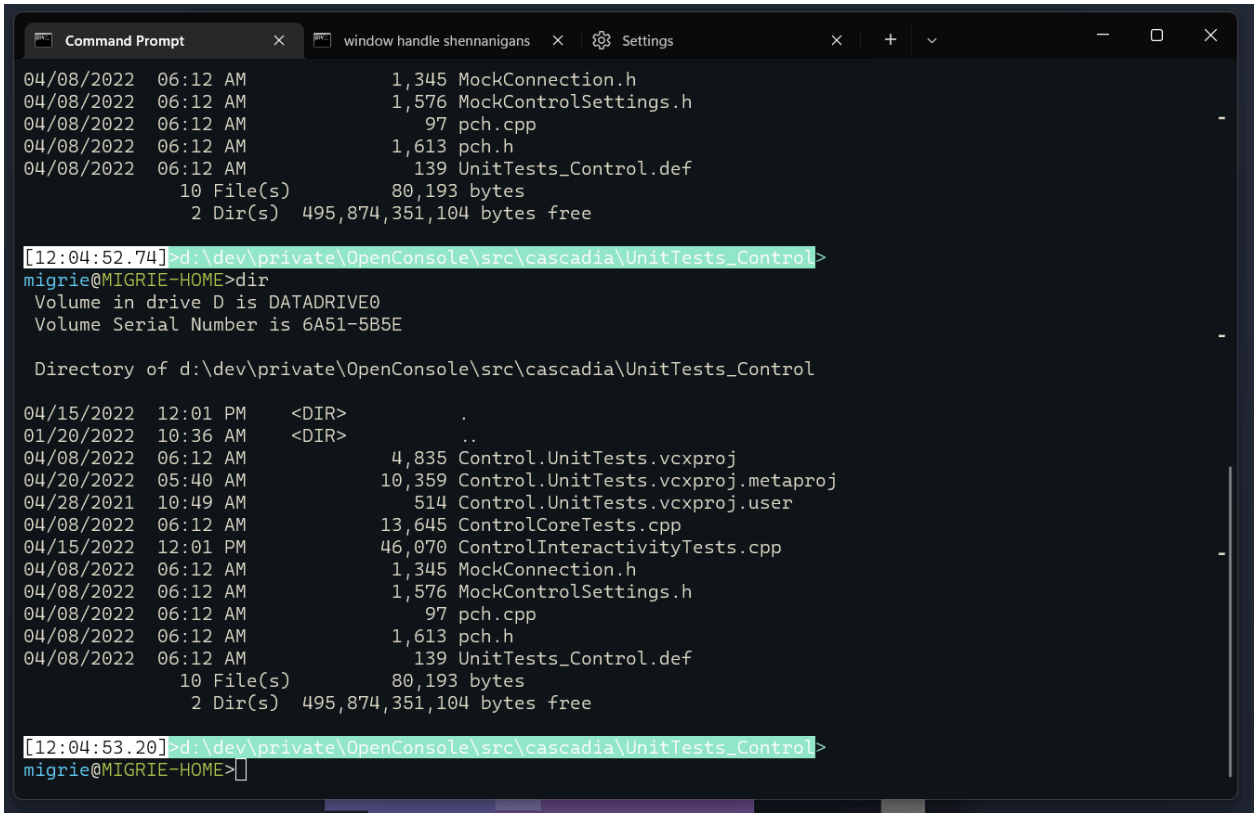
> **Note**: Don't see your favorite shell here? If you figure it out, feel free to contribute a solution for your preferred shell!

Shell integration features

Open new tabs in the same working directory

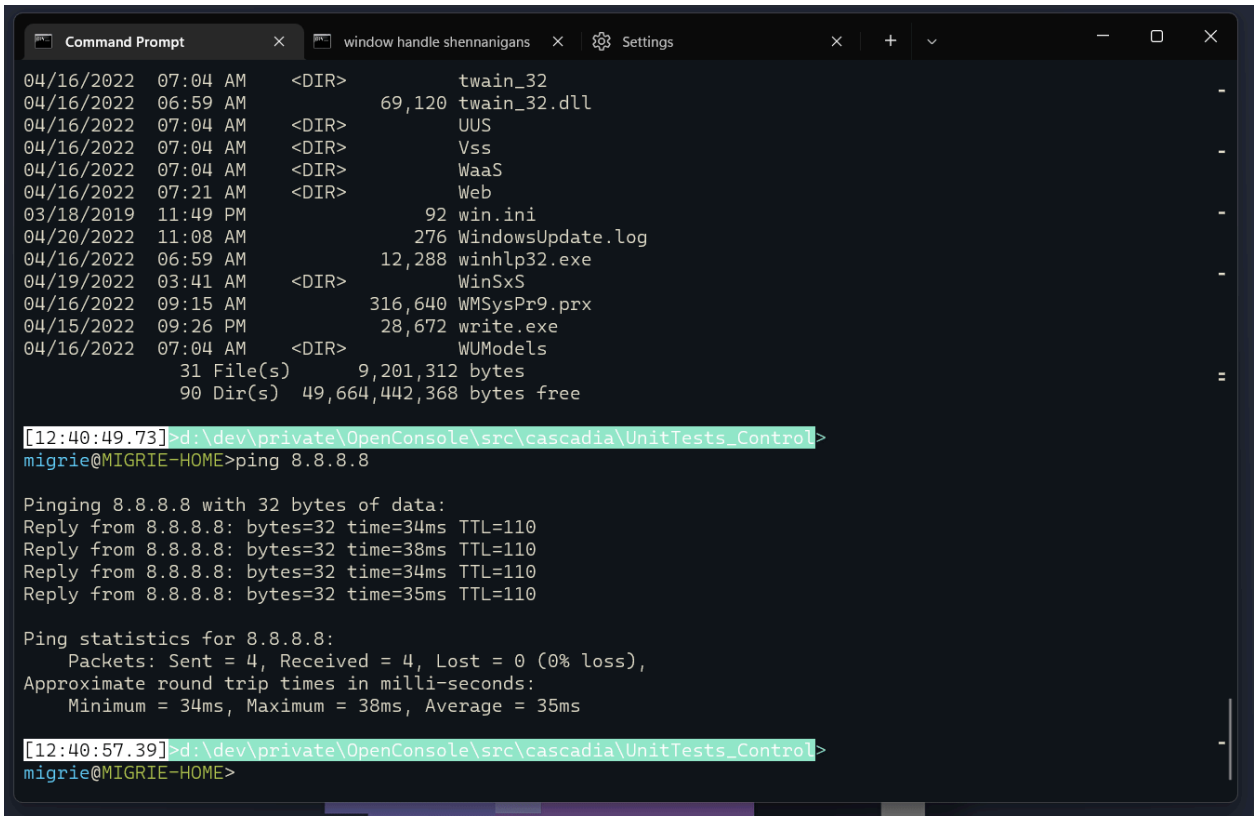


Show marks for each command in the scrollbar



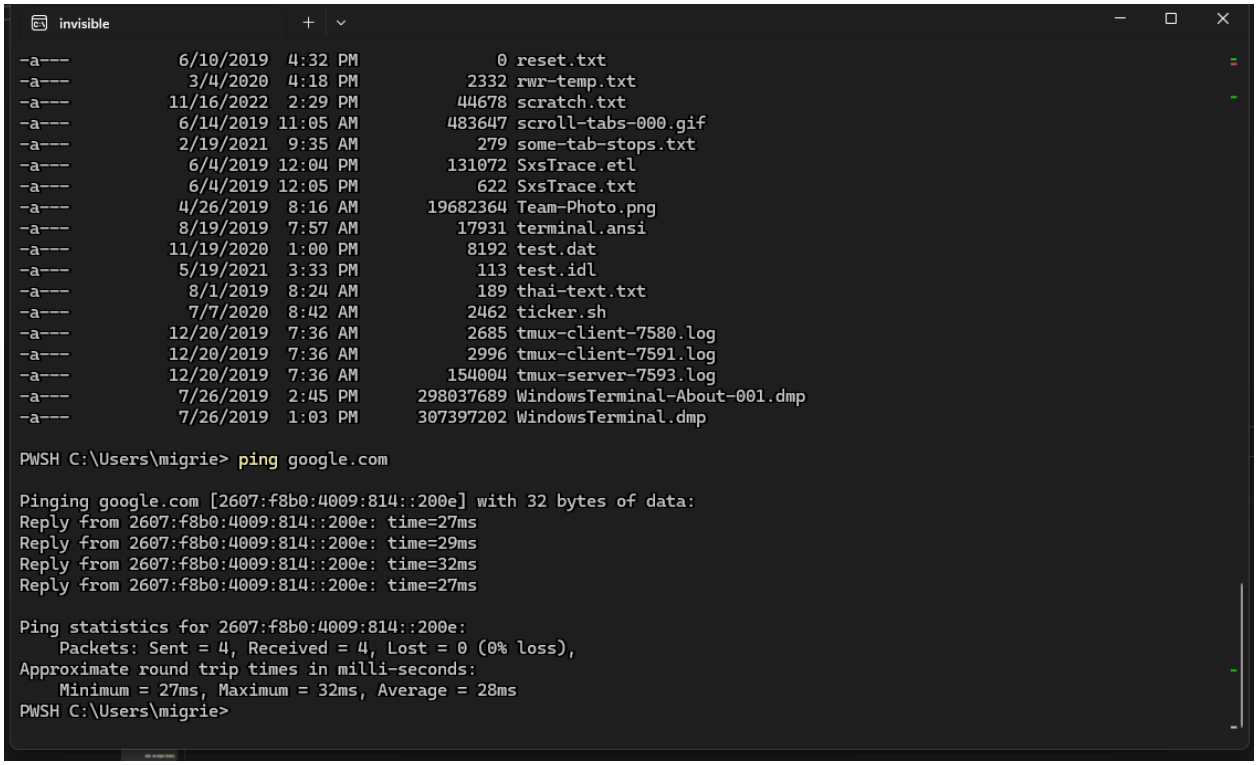
Automatically jump between commands

This uses the `scrollToMark` actions as we have them defined above.

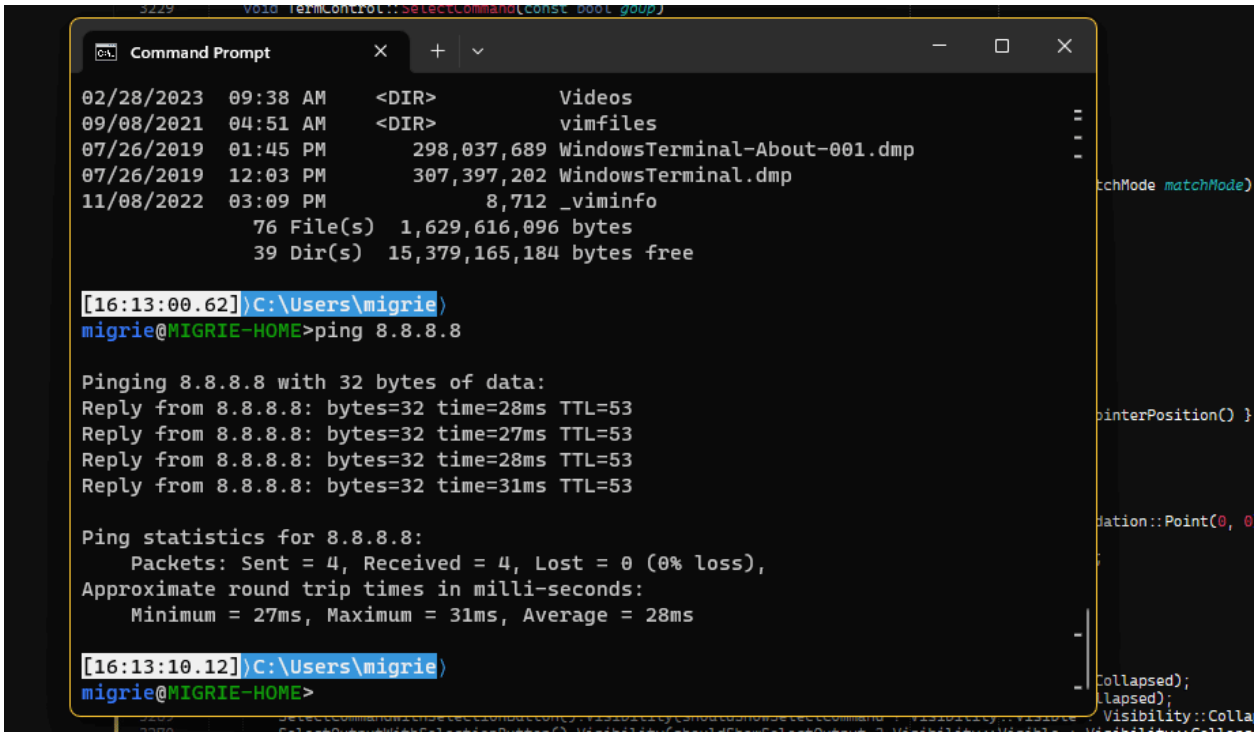


Select the entire output of a command

In this gif, we use the `selectOutput` action bound to `ctrl+g` to select the entire output of a command.

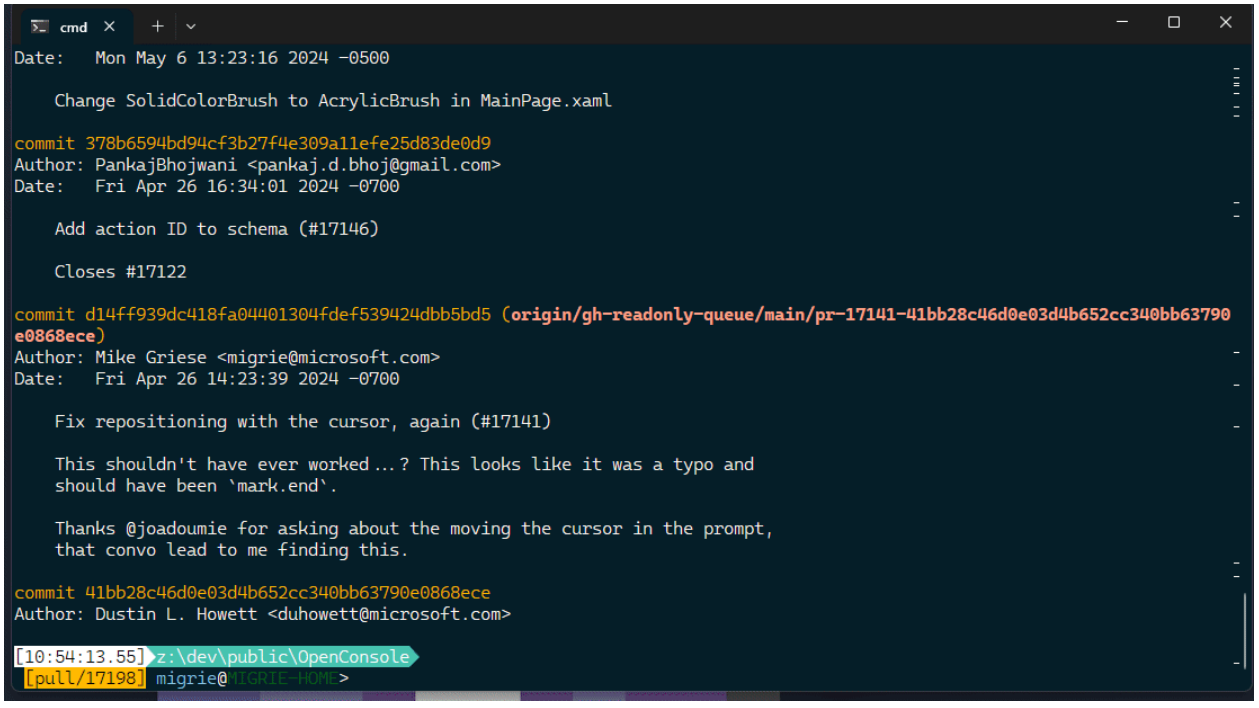


The following uses the `experimental.rightClickContextMenu` setting to enable a right-click context menu in the Terminal. With that and shell integration enabled, you can right-click on a command to select the entire command or its output.



Recent command suggestions

With shell integration enabled, the Suggestions UI can be configured to also show your recent commands.



You can open this menu with the following action:

JSON

```
{
    "command": { "action": "showSuggestions", "source": "recentCommands",
```

```
"useCommandline": true },
},
```

(For more info, see the Suggestions documentation)

Additional resources

- autoMarkPrompts
- showMarksOnScrollbar
- showSuggestions

Cascadia Code

Article • 12/10/2021

Cascadia Code is a new monospaced font from Microsoft that provides a fresh experience for command-line applications and text editors. Cascadia Code was developed alongside Windows Terminal. This font is most recommended to be used with terminal applications and text editors such as Visual Studio and Visual Studio Code.

Cascadia Code versions

There are multiple versions of Cascadia Code available that include ligatures and glyphs. All versions of Cascadia Code can be downloaded from the Cascadia Code GitHub releases page. Windows Terminal ships Cascadia Code and Cascadia Mono in its package and uses Cascadia Mono by default.

Font Name	Includes Ligatures	Includes Powerline Glyphs
Cascadia Code	Yes	No
Cascadia Mono	No	No
Cascadia Code PL	Yes	Yes
Cascadia Mono PL	No	Yes

Powerline and programming ligatures

Powerline is a common command-line plugin that allows you to display additional information in your prompt. It uses a few additional glyphs to display this information properly.

Programming ligatures are glyphs that are created by combining characters. They are most useful when writing code. The "Code" variants include ligatures, whereas the "Mono" variants exclude them.

Contributing to Cascadia Code

Cascadia Code is licensed under the SIL Open Font license on GitHub.

Terminal Chat (Experimental)

Article • 10/29/2024

Terminal Chat is a new experimental feature that enables you to integrate Windows Terminal Canary with your preferred AI service.

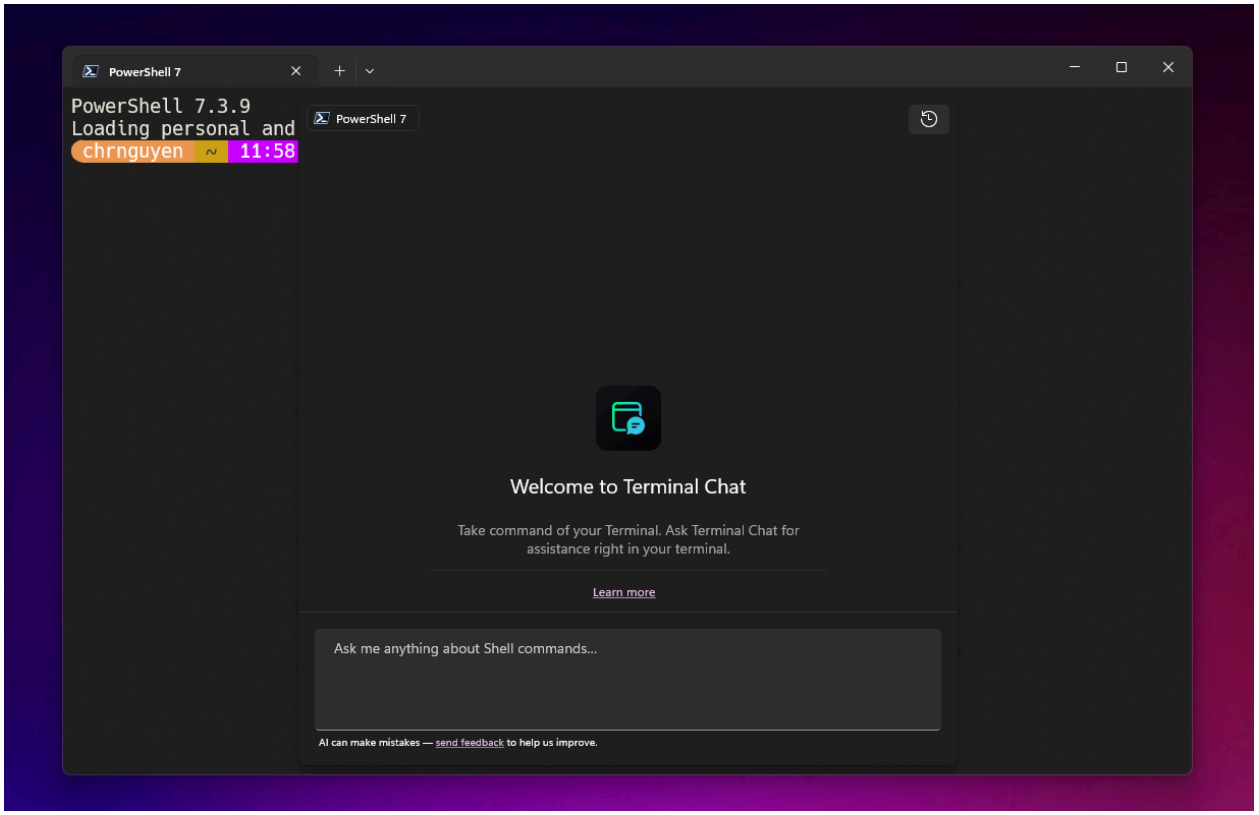
Once Terminal Chat is connected to your AI service provider (GitHub Copilot, Azure OpenAI, or OpenAI), you can ask questions specific to the shell you are using (PowerShell, CMD, WSL Ubuntu, Azure Cloud Shell, etc) while staying in the context of your terminal.

Terminal Chat can:

- Provide command syntax and descriptions
- Explain command line error messages
- Send code suggestions to command-line text editors

Terminal Chat does not ship with its own large-language model. For now, this experimental feature is only available in Windows Terminal Canary and only supports GitHub Copilot, Azure OpenAI Service, and OpenAI.

Terminal Chat only communicates with your selected AI service when you enter a message in the chat. The chat history and name of the user's active shell is also appended to the message that is sent to the AI service. The chat history is not saved by Windows Terminal after the terminal session is over.



Prerequisites

- This experimental feature is only available in Windows Terminal Canary.
- An AI service provider subscription is required. GitHub Copilot, Azure OpenAI, and OpenAI are currently supported.

Set up a Service Provider in Terminal Chat

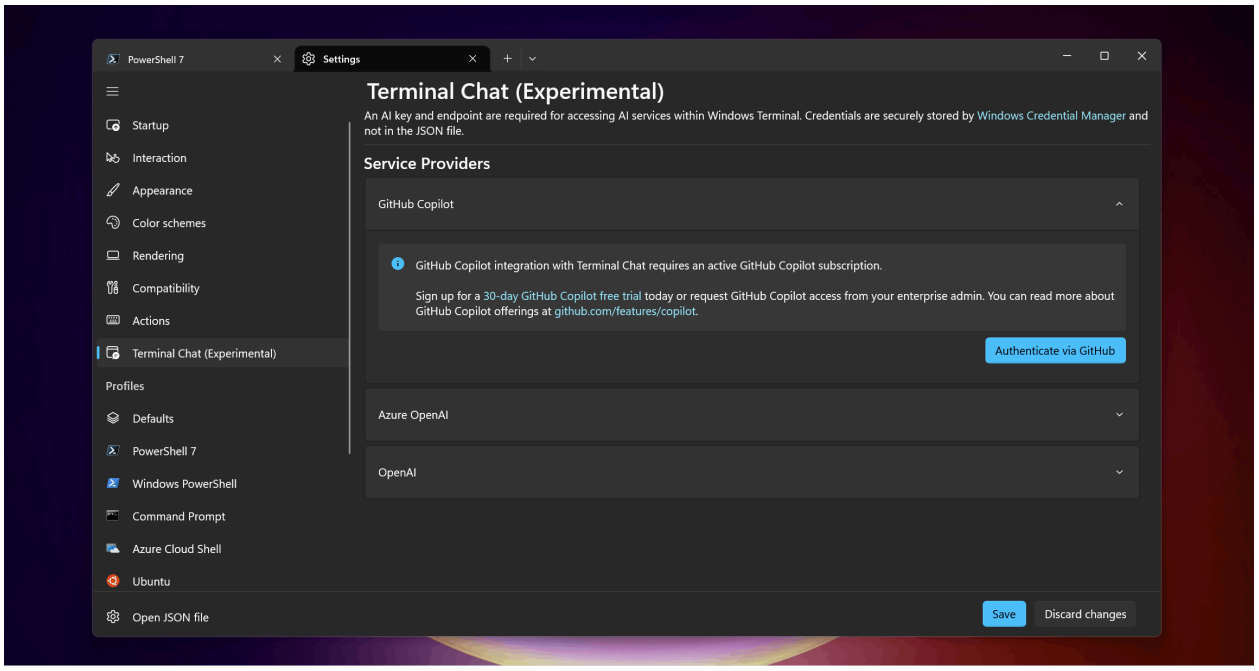
To use Terminal Chat, you will need to connect GitHub Copilot or add an Azure OpenAI or OpenAI endpoint to your Terminal Chat settings.

1. Open Windows Terminal and select **Settings** from the drop-down menu on the far-right of the top menu bar.
2. Select the **Terminal Chat (Experimental)** tab to display the service provider configuration settings.
3. Select a **Service Provider**. A subscription to one of the following AI service providers is required to use Terminal Chat. You will need to set up and authenticate the AI service in the service provider settings. See below for the steps to activate your preferred AI service provider.

AI Service Providers

GitHub Copilot: Under Service Providers, select **GitHub Copilot** and **Authenticate via GitHub** to sign in to GitHub. Check **Set as active provider** to set GitHub Copilot as your active Service Provider.

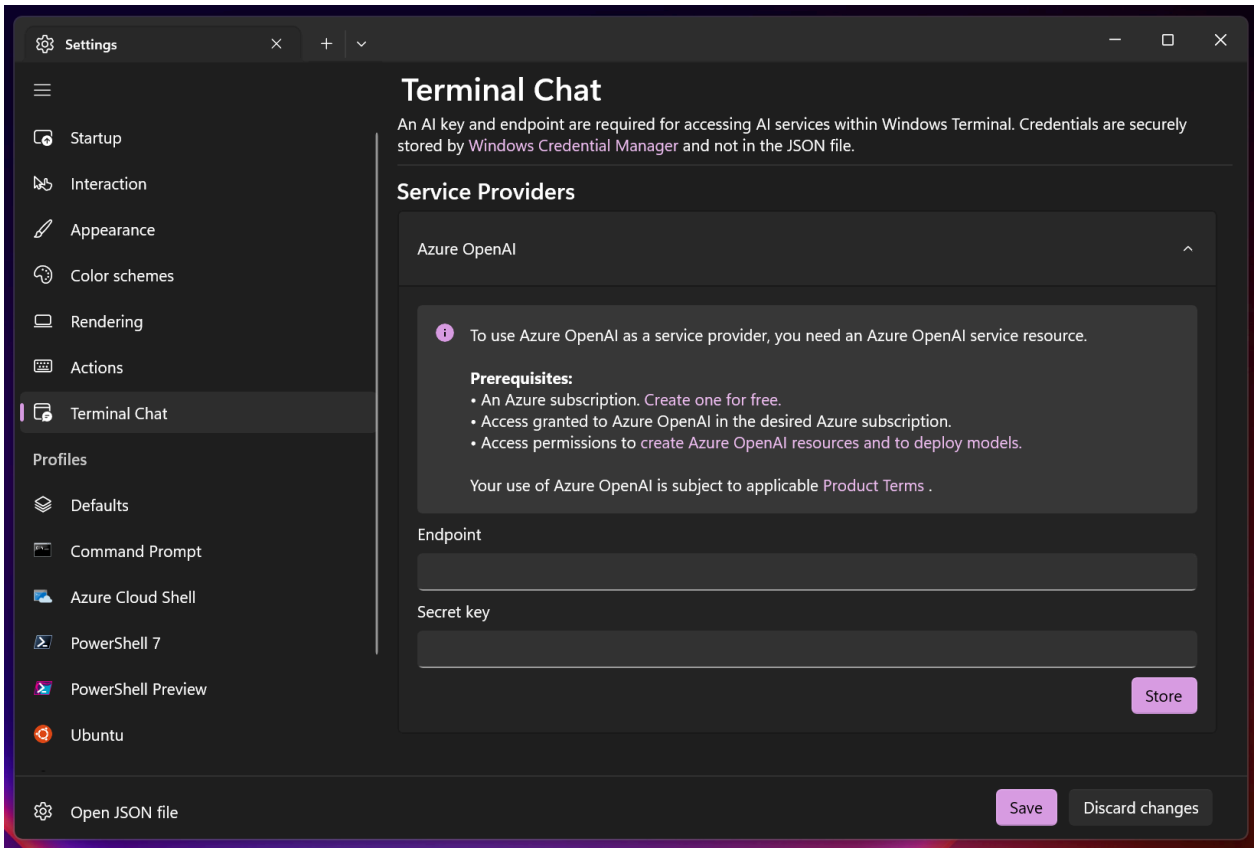
To connect GitHub Copilot with Terminal Chat, you must have an active subscription for GitHub Copilot in your personal account, or you need to be assigned a seat by your organization. You can sign up for a GitHub Copilot free trial in your personal account to evaluate GitHub Copilot.



> **Note**
>
> If you have access to GitHub Copilot via your organization, you won't be able to use GitHub Copilot if your organization owner has disabled GitHub Copilot in the CLI. See "Managing policies for Copilot in your organization."

For more information on how to use GitHub Copilot responsibly with Windows Terminal, see "Responsible use of GitHub Copilot in Windows Terminal".

Azure OpenAI: Under Service Providers, select **Azure OpenAI**, enter an endpoint URL and key, select **Store** and **Save**.



To get an Azure OpenAI Service endpoint and key, you will need to create and deploy an Azure OpenAI Service resource.

- Create and deploy an Azure OpenAI Service resource

You will need to use a `gpt-35-turbo` model and ensure that the jailbreak content filter is enabled for your deployment.

After creating a resource and deploying a model, you can find your Endpoint and API key by navigating to the **Chat** playground in Azure OpenAI Studio and selecting **View code** in the Chat session section. The pop-up dialog will provide an endpoint URL and key that you can use in the Terminal Chat Service Provider settings.

OpenAI : Under Service Providers, select **OpenAI**, enter an endpoint URL and key, select **Store** and **Save**.

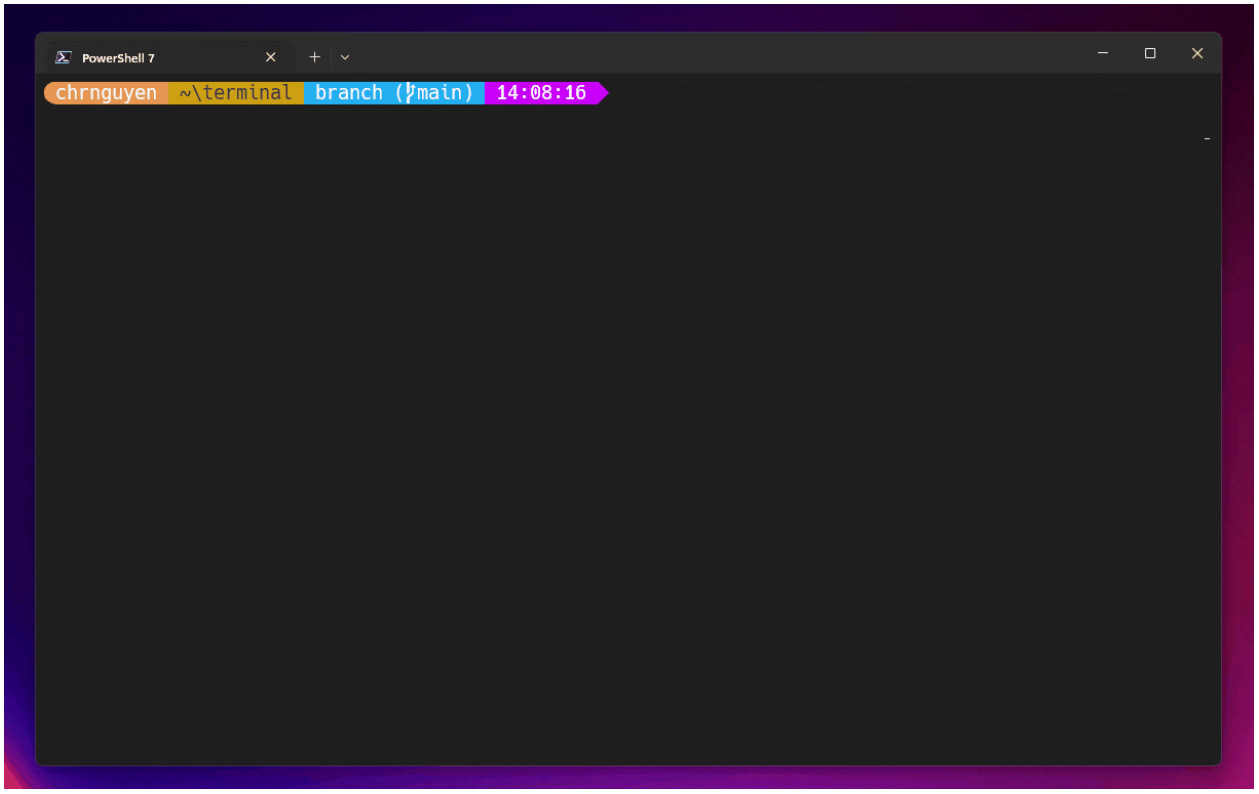
To get an OpenAI endpoint and key, you will need to refer to the OpenAI documentation.

- Developer quickstart: Create and export an API key

OpenAI is provided by a third-party and not Microsoft. When you send a message in Terminal Chat, your Terminal Chat history per session and the name of your active shell profile is sent to the third-party AI service for use by OpenAI. Your use of OpenAI is governed by the relevant third-party terms, conditions, and privacy statement.

Examples for using Terminal Chat

The following examples demonstrate a few ways that you might consider using Terminal Chat.



- **Command suggestions**: Ask for a command that you would like to use. Terminal Chat also adds the name of the active shell to the prompt after it is sent, so if you are using PowerShell and ask "How do I make a new directory?", the response may be `md`, but if you are using an Ubuntu (Linux) command line with WSL, the response may be `mkdir`. Clicking on the suggestion will copy it to the input line of the terminal. This will not run the suggestion for the user automatically.

- **Translate commands**: Terminal Chat can also be used to "translate" commands. For example, you can ask "What's `touch` in PowerShell?" or "How do I `touch` in PowerShell?" to get the suggestion of `New-Item`, a PowerShell command that is equivalent to the Linux / Unix-based `touch` command for creating a new file.

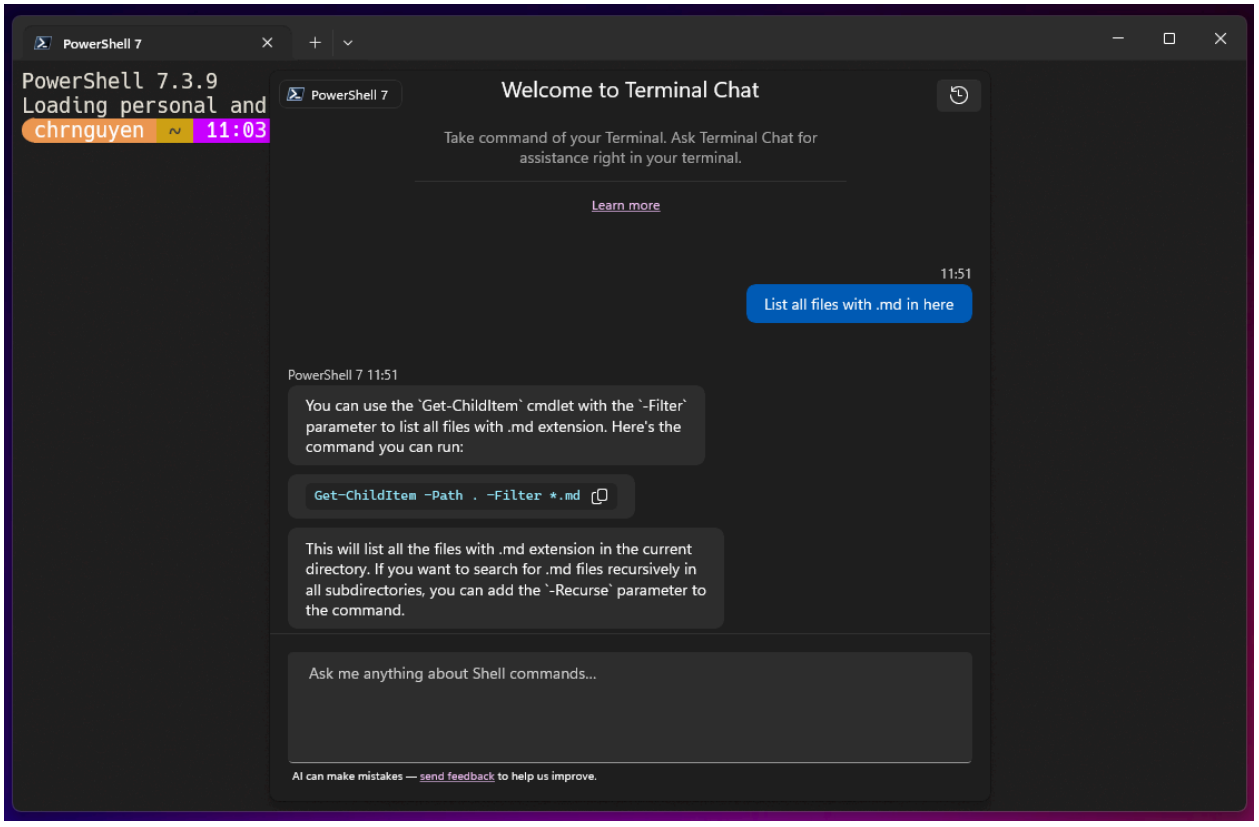
- **Explain an error**: If you've received an unfamiliar error response in your command line, copy and paste it into the Terminal Chat and ask for an explanation of the error code and how to fix it. For example, "How do I fix `Error: getaddrinfo ENOTFOUND`?"

- **Send code suggestions to command-line text editors**: If you're using a command-line text editor in WSL (like `nano` or `vi`), you can ask Terminal Chat to generate code and send the code suggestion to the editor by clicking the "Copy" button.

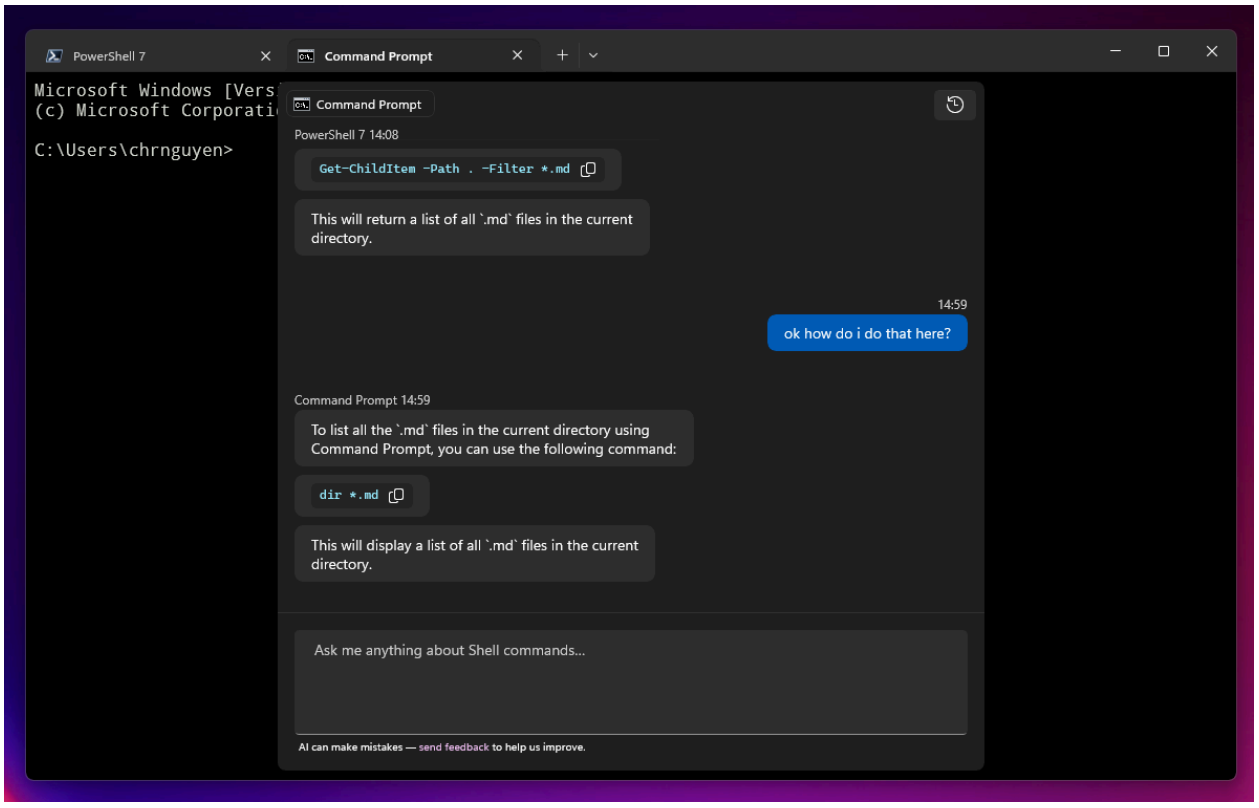
- **Find and describe PowerShell cmdlets**: A cmdlet (pronounced command-let) is a lightweight script command used to perform a specific function in PowerShell. Terminal Chat can help you to find cmdlets that may be useful and can explain what task they accomplish. For example, "Find a cmdlet to show a list of active processes" would result in `Get-Process`. Or "Explain the `Get-ChildItem` cmdlet" would describe that the cmdlet is used to retrieve a list of items within a specified location.

Terminal-specific context

Terminal Chat takes the name of the active shell and sends that name as additional context to the AI service to get suggestions that are more tailored towards the active shell.



This means that Terminal Chat can identify whether a user's active shell is Command Prompt or PowerShell for example.



Assigning a keybinding to Terminal Chat

Terminal Chat can be set as a keybinding in the **Actions** tab of Windows Terminal **Settings**. Add a new keybinding by selecting **+ Add new** and then selecting **Toggle Terminal Chat** from the dropdown to add a new keybinding Action for the Terminal Chat feature.

The new keybinding will also be reflected in the dropdown menu after these changes are saved.

Group Policy

Language Models and AI Services used by Terminal Chat can be disabled through the **Enabled Language Models/AI Providers** policy. The Terminal Chat feature can also be disabled with the same policy. To learn more, see the Group Policies page.

Group Policy for Windows Terminal

Article • 10/29/2024

Since Windows Terminal Preview 1.22, Windows Terminal is released on GitHub with Administrative Templates that help you to configure Windows Terminal using Group Policy.

Group Policy is a feature in Microsoft Windows that allows administrators in a business environment to manage and configure operating system settings, applications, and user environments centrally, including access available via Windows Terminal. Learn more in Group Policy overview for Windows Server.

How to set up Windows Terminal Group Policy

Download

You can find the latest administrative templates (ADMX and ADML files) in the assets section of our newest Windows Terminal release on GitHub. The file is named `GroupPolicyTemplates-<Version>.zip`

Add the administrative template to an individual computer

1. Unzip the `GroupPolicyTemplates-<Version>.zip` file to your **Policy Definition** template folder (`C:\Windows\PolicyDefinitions`).

> **Note**
>
> The Group Policy is available in English (United States region), or `en-US`, as the fallback language. If no language-location specific context is added, the `en-US` default will be used.

Add the administrative template to Active Directory

1. On a domain controller (a server that responds to security authentication requests within a Windows Server domain) or a workstation with RSAT, go to the **PolicyDefinition** folder (also known as the Central Store) on any domain controller for your domain. For older versions of Windows Server, you might need to create

the PolicyDefinition folder. For more information, see How to create an manage the Central Store for Group Policy Administrative Templates in Windows

2. Copy the `WindowsTerminal.admx` file to the **PolicyDefinition** folder. (`%systemroot%\sysvol\domain\policies\PolicyDefinitions`)
3. Copy the `WindowsTerminal.adml` file to the matching language folder in your language folder in your Policy Definition folder. Create the folder if it does not already exist. (`%systemroot%\sysvol\domain\policies\PolicyDefinitions\EN-US`)
4. If your domain has more than one domain controller, the new ADMX files will be replicated to them at the next domain replication interval.

Import the administrative template in Intune

You can find all instructions on how to import the administrative template in Intune on Import custom ADMX and ADML administrative templates into Microsoft Intune.

> ⓘ **Important**
>
> You will need to import `Windows.admx` since the Windows Terminal ADMX files contains references to that file.

Policies

Disabled Source Profiles

Supported on Windows Terminal 1.21 or later, this policy disables source profiles from being generated. Source names can be arbitrary strings. Potential candidates can be found as the "source" property on profile definitions in Windows Terminal's `settings.json` file.

Common sources are:

- Windows.Terminal.Azure
- Windows.Terminal.PowershellCore
- Windows.Terminal.Wsl

For instance, setting this policy to `Windows.Terminal.Wsl` will disable the built-in WSL integration of Windows Terminal. Existing profiles will disappear from Windows Terminal after adding their source to this policy.

Group Policy (ADMX) information

- GP unique name: DisabledProfileSources
- GP name: Disabled Profile Sources
- GP path: Administrative Templates/Windows Components/Windows Terminal/
- GP scope: Computer and user
- ADMX file name: WindowsTerminal.admx

Registry information

- Path: Software\Policies\Microsoft\Windows Terminal
- Name: DisabledProfileSources
- Type: MULTI_SZ
- Example value: `Windows.Terminal.Azure`

Enabled Language Models/AI Providers

Supported on Windows Terminal Canary 1.23 or later, this policy allows the listed Language Models / AI providers to be used with Terminal Chat.

Common providers are:

- OpenAI
- AzureOpenAI
- GitHubCopilot

For instance, setting this policy to `OpenAI` will allow the use of OpenAI in Terminal Chat.

Disabling Terminal Chat

Enabling this policy but leaving the list empty disallows all providers and disables the Terminal Chat feature.

Group Policy (ADMX) information

- GP unique name: EnabledLMProviders
- GP name: Enabled Language Models/AI Providers
- GP path: Administrative Templates/Windows Components/Windows Terminal/
- GP scope: Computer and user
- ADMX file name: WindowsTerminal.admx

Registry information

- Path: Software\Policies\Microsoft\Windows Terminal
- Name: EnabledLMProviders
- Type: MULTI_SZ
- Example value: `AzureOpenAI`

Windows Terminal tips and tricks

Article • 10/06/2022

On first launch

When you first install Windows Terminal, you will be greeted with a Windows PowerShell prompt. Windows Terminal ships with Windows PowerShell, Command Prompt, and Azure Cloud Shell profiles by default.

In addition to these profiles, if you have any Windows Subsystem for Linux (WSL) distributions installed, the terminal will automatically create profiles for those distributions as well. If you would like to install additional WSL distributions on your machine, you can do so after installing terminal and on your next terminal launch, the profiles for those distributions will automatically appear. These profiles will have the Linux Tux image as their icon.

> **Note**
>
> You can change the icon of each WSL distribution if desired. Specific distribution icons do not come shipped inside the terminal but can be downloaded and assigned using the terminal settings.

View default settings

Windows Terminal comes with a large set of default settings, including color schemes and keyboard shortcuts (now called "Custom actions"). If you'd like to view the default settings file, you can hold `Alt` and click on the Settings button inside the dropdown menu.

Default profile settings

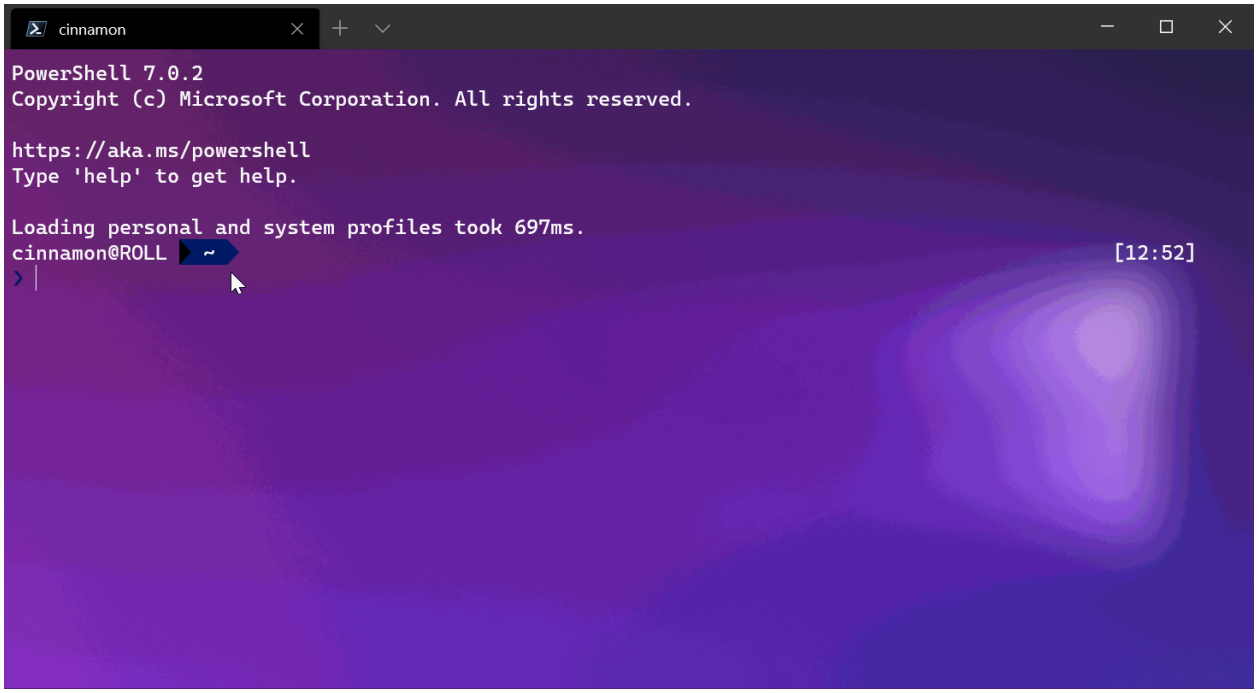
Windows Terminal enables you to apply a setting to every profile without having to duplicate the setting for each profile entry. This can be done by adding a setting inside the "defaults" array inside the profiles object. Learn more about General profile settings, Appearance profile settings, and Advanced profile settings.

JSON

```
"profiles":
    {
        "defaults":
        {
            // Put settings here that you want to apply to all profiles.
            "fontFace": "Cascadia Code"
        },
        "list":
        []
    }
```

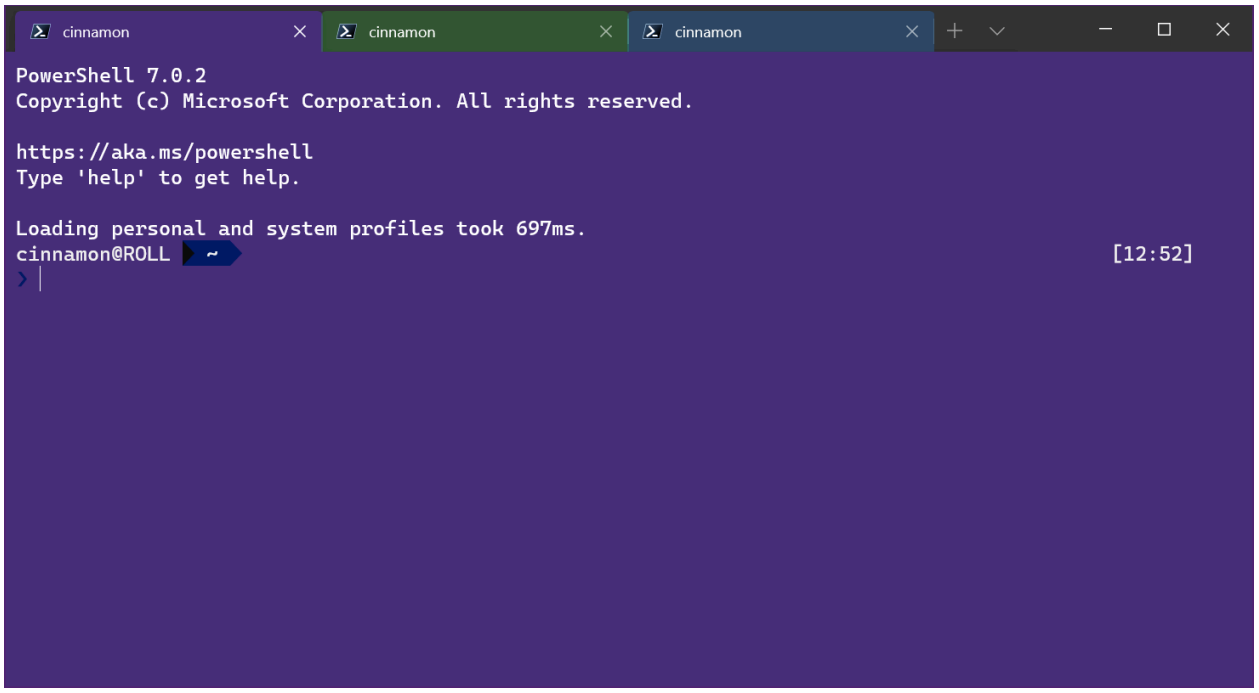
Rename a tab

You can right click on a tab and select Rename Tab to rename a tab for that terminal session. Clicking this option in the context menu will change your tab title into a text field, where you can then edit the title. If you'd like to set the tab title for that profile for every terminal instance, you can learn more in the Tab title tutorial.



Color a tab

You can right-click on a tab and select **Color**... to color the tab for that terminal session. You can select from a predefined list of colors or you can select **Custom**... to pick any color using the color picker or the RGB/HSV or hex fields.



> **Tip**
>
> Use the hex field to set your tab to the same color as your background color for a seamless look.

The `tabColor` can be set as part of a profile. See Profile - Appearance: Tab color. For example:

JSON

```
{
            "guid": "{1234abc-abcd-1234-12ab-1234abc}",
            "name": "Windows PowerShell",
            "background": "#012456",
            "tabColor": "#012456",
        },
```

The `tabColor` cannot be set as part of a color scheme. Additionally, while it is possible to set the tab title from the commandline with escape sequences, it currently isn't possible to set the tab color in this way.

Mouse interaction

There are several ways to interact with Windows Terminal using a mouse.

Zoom with the mouse

You can zoom the text window of Windows Terminal (making the text size larger or smaller) by holding `Ctrl` and scrolling. The zoom will persist for that terminal session. If you want to change your font size, you can learn more about the font size feature on the Profile - Appearance page.

Adjust background opacity with the mouse

You can adjust the opacity of the background by holding `Ctrl+Shift` and scrolling. The opacity will persist for that terminal session. If you want to change your acrylic opacity for a profile, you can learn more about acrylic background effects on the Profile - Appearance page.

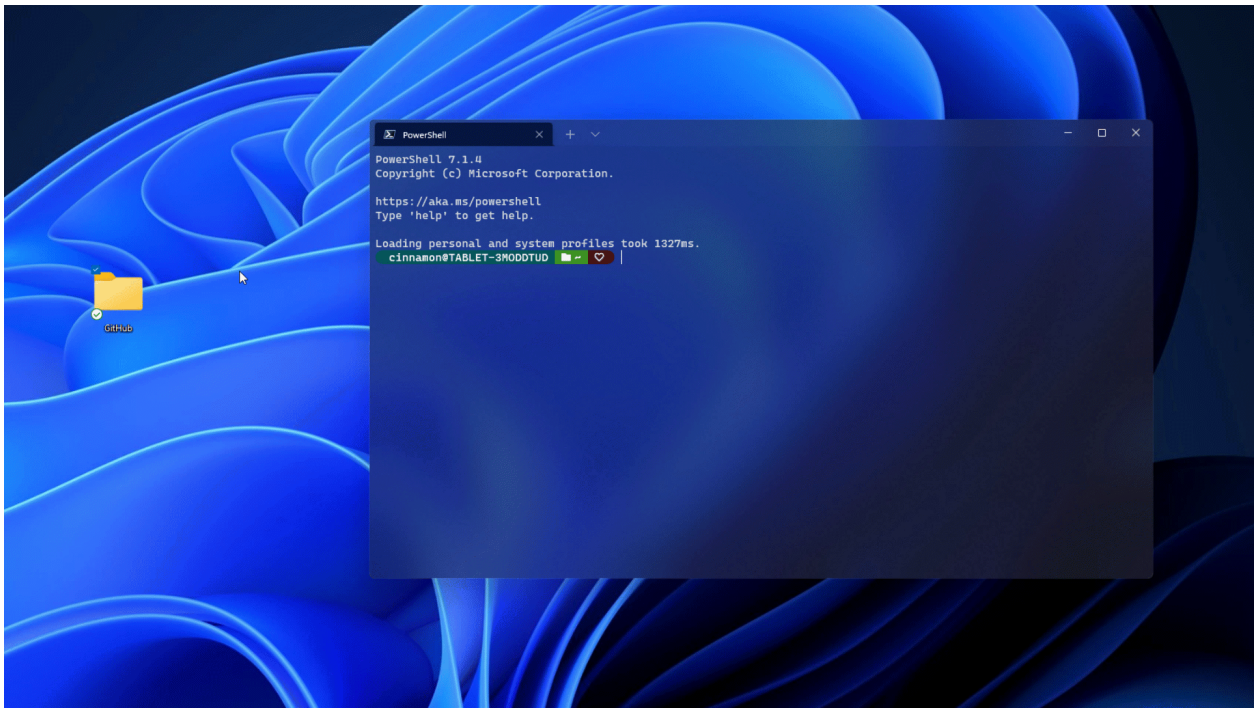
> **Note**
>
> In Windows Terminal version 1.12, changing the background opacity with the mouse wheel will use vintage-style opacity by default, unless `useAcrylic` is set to true in your settings. Prior to 1.12, the terminal would always use acrylic for transparency.

Open a hyperlink

You can open a hyperlink from inside Windows Terminal with your mouse using `Ctrl` + click.

Drag and drop file/folder to open

You can drag and drop a file or folder over the New Tab button to open your default profile with that given file or folder. By default, this will open a new tab. You can hold `Alt` to open a new pane in your current tab or hold `Shift` to open a new window.



Copy/paste

You can right-click with your mouse to copy and paste text within Windows Terminal using your clipboard storage.

Windows Terminal also includes a **copyOnSelect** setting that can be set to `true` in order for any text selected with your mouse to be immediately copied to your clipboard. The right-click on your mouse will always paste in this case.

Virtual Terminal and WSL mouse support

Windows Terminal supports mouse input in Windows Subsystem for Linux (WSL) applications as well as Windows applications that use virtual terminal (VT) input. This means applications such as tmux and Midnight Commander will recognize when you select items in the Terminal window. If an application is in mouse mode, you can hold down `Shift` to make a selection instead of sending VT input.

Send input commands with a key binding

Windows Terminal gives you the ability to send input to your shell with a key binding. This can be done with the following structure inside the `"actions"` array of your settings.json file.

JSON

```
{ "command": {"action": "sendInput", "input": ""}, "keys": "" }
```

You can also add a `"name": ""` value if desired.

Clear your screen

Sending input to the shell with a keyboard shortcut can be useful for commands you run often. One example would be clearing your screen:

JSON

```
{ "command": {"action": "sendInput", "input": "clear\r"}, "keys": "alt+k",
"name": "clear terminal" }
```

Navigate to parent directory

Navigating to the parent directory with a key binding may also be helpful.

JSON

```
{ "command": {"action": "sendInput", "input": "cd ..\r"}, "keys": "ctrl+up"
}
```

You can also use this functionality to run builds or test scripts.

Focus mode

"Focus mode" hides the title bar and tabs normally located at the top of Windows Terminal, letting you focus only on the terminal content. It is similar to "Zen mode" in Visual Studio Code.

To enter focus mode, open the command palette using `Ctrl` + `Shift` + `p`, enter "focus mode", and select "Toggle focus mode." To exit focus mode, repeat these same steps.

To set focus mode to launch every time you start Windows Terminal, open the **Settings** (`Ctrl` + `,`) and select the **Startup** tab. Under **Launch mode**, select **Focus** (or **Maximized focus**, which is focus mode with your terminal window maximized). Select **Save** before exiting. The next time you launch the Windows Terminal, it will open up in focus mode. To stop Windows Terminal from launching in focus mode, follow these same steps, but select **Default** from the list of **Launch mode** options.

To add a shortcut key (or keybinding) for entering focus mode, open the `settings.json` file (`Ctrl` + `Shift` + `,`). Inside your `settings.json` file, find the "actions": section and add the following command:

JSON

```
{ "command": "toggleFocusMode", "keys": "ctrl+f12" }
```

Replace "ctrl+f12" with the shortcut / keybinding of your choice, but be sure not to repeat any existing keybindings from the Actions list. You can also see a list of Actions with associated keybindings, and + **Add new** bindings, in the **Actions** tab of the Windows Terminal **Settings** dashboard. Remember to **Save** after making any changes. You can now toggle focus mode using the "action" shortcut key that you created. (In the case of our example, `Ctrl` + `F12`).

To learn more about this command, see toggleFocusMode.

Quake mode

"Quake mode" is the name for the special mode the terminal enters when naming a window `_quake`. When a window is in quake mode:

- The terminal is automatically snapped to the top half of the monitor.
- The window can no longer be resized horizontally or from the top. It can only be resized on the bottom.
- The window automatically enters focus mode (note that you may have multiple tabs in focus mode).
- When windowingBehavior is set to `"useExisting"` or `"useAnyExisting"`, they will ignore the existence of the `_quake` window.
- When minimized, the window will be hidden from the taskbar and from `Alt+Tab`.
- Only one window may be the quake mode window at a time.

The quake mode window can be created either by binding the `quakeMode` action, or by manually running the command line:

Console

```
wt -w _quake
```

> **Note**
>
> If you don't have a quakeMode action bound and minimize the quake window, you'll need to go into Task Manager to be able to exit that terminal window!

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Tutorial - Set up a custom prompt for PowerShell or WSL with Oh My Posh

Article • 10/29/2024

Oh My Posh provides theme capabilities for a fully customized command prompt experience providing Git status color-coding and prompts.

If you just want to customize the color schemes or appearance of your terminal, you can do so in the Windows Terminal settings (without the need to install Oh My Posh themes).

In this tutorial, you learn how to:

- ✓ Install a Nerd Font
- ✓ Customize your PowerShell prompt with Oh My Posh
- ✓ Customize your WSL prompt with Oh My Posh
- ✓ Use Terminal-Icons to add missing folder or file icons

Install a Nerd Font

Customized command prompts often use glyphs (a graphic symbol) to style the prompt. If your font does not include the appropriate glyphs, you may see several Unicode replacement characters '□' in your prompt.

To see all of the glyphs in your terminal, we recommend installing a Nerd Font like Cascadia Code NF, which can be downloaded from the Cascadia Code release page.

After downloading, you will need to unzip and install the font on your system. (How to add a new font to Windows ↗).

To set a Nerd Font for use with Oh My Posh and Terminal Icons, open the Windows Terminal settings UI by selecting **Settings** (Ctrl+,) from your Windows Terminal dropdown menu. Select the profile where you wish to apply the font (PowerShell for example) and then select **Appearance**. In the **Font face** drop-down menu, select *Cascadia Code NF* or whichever Nerd Font you want to use.

> ⓘ **Note**
>
> If you want to use a terminal font that does not support glyph icons, such as Cascadia Code PL ↗, consider using an Oh My Posh theme that contains the `minimal` function, indicating that additional icons aren't required.

Customize your PowerShell prompt with Oh My Posh

Oh My Posh enables you to use a full color set to define and render your terminal prompt, including the ability to use built-in themes or create your own custom theme.

Install Oh My Posh for PowerShell

To customize your PowerShell prompt, install Oh My Posh using winget, which will install:

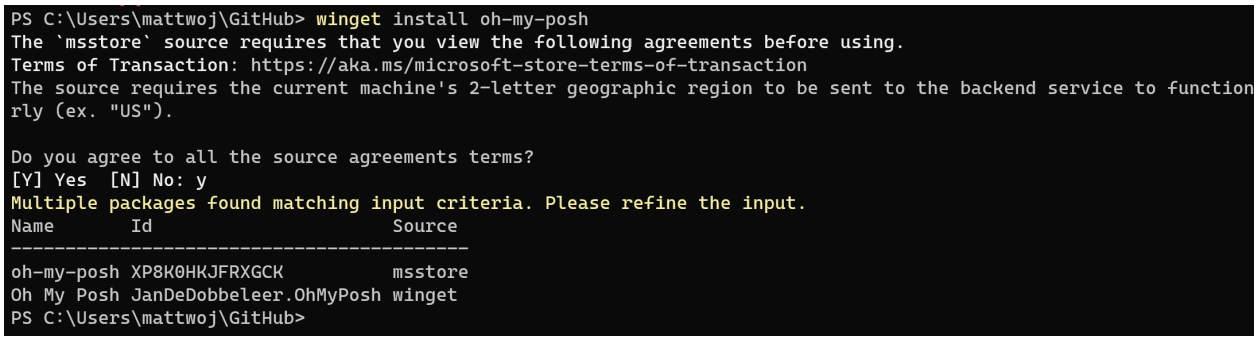
- `oh-my-posh.exe`: The Windows executable
- `themes`: The latest Oh My Posh themes ↗

To start the installation, enter the command:

PowerShell

```
winget install JanDeDobbeleer.OhMyPosh
```

You will need to agree to the source terms and may run into the instance that more than one package is available. In this case, select the package ID that you want to use and re-enter the command: `winget install <package ID>`.



To use the Microsoft Store version of Oh My Posh, which will automatically update when new versions are available, use the command:

PowerShell

```
winget install XP8K0HKJFRXGCK
```

Enter `oh-my-posh version` to confirm the version number of your Oh My Posh installation. To ensure you have the latest updates, you can use the following command: `winget upgrade oh-my-posh`.

> **Note**
>
> If you want to install the newest version of Oh My Posh in PowerShell, you may want to first remove the OMP module's cached files and uninstall the old module. There are instructions on how to do this in the Oh My Posh docs. If you are more familiar with the Scoop installer or a manual installation method that allows automation, these can also be used for installing on Windows, just follow the instructions in the Oh My Posh docs.

Choose and apply a PowerShell prompt theme

You may browse the full list of themes on the Oh My Posh themes page.

Choose a theme and update your PowerShell profile with this command. (You can replace `notepad` with the text editor of your choice.)

PowerShell

```
notepad $PROFILE
```

If you receive a path error, you may not yet have a profile for PowerShell. To create one, use the following PowerShell command to create a profile and then try opening it with a

text editor again.

```PowerShell
new-item -type file -path $profile -force
```

Add the following to the end of your PowerShell profile file to set the `paradox` theme. (Replace `paradox` with the theme of your choice.)

```PowerShell
oh-my-posh init pwsh --config "$env:POSH_THEMES_PATH\paradox.omp.json" |
Invoke-Expression
```

Now, each new PowerShell instance will start by importing Oh My Posh and setting your theme.

If you receive a script error when trying to open a new PowerShell instance, your Execution Policy for PowerShell may be restricted. To set your PowerShell Execution Policy to unrestricted, you will need to launch PowerShell as an administrator and then use the following command:

```PowerShell
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

> **Note**
>
> This is not your Windows Terminal profile. Your PowerShell profile is a script that runs every time PowerShell starts. Learn more about PowerShell profiles.

> **Tip**
>
> See the Oh My Posh FAQs for answers to common questions or issues. To learn more about the configuration and general settings, such as how to restore the current working directory, see the Oh My Posh docs.

Customize your WSL prompt with Oh My Posh

Oh My Posh also allows you to customize WSL prompts using built-in themes.

Install Oh My Posh for WSL

We recommend installing Oh My Posh for WSL, whether using Bash, Zsh, or something else, by following the Linux install guide in the Oh My Posh docs.

Customizing WSL prompts with Oh My Posh uses the Homebrew package manager for installation. When installing Homebrew for Linux, be sure to follow Next steps instructions to add Homebrew to your PATH and to your bash shell profile script.

Homebrew will install:

- `oh-my-posh` - Executable, added to /usr/local/bin
- `themes` - The latest Oh My Posh themes

Choose and apply a WSL prompt theme

The Oh My Posh themes will be found in the oh-my-posh directory as JSON files. You can find it by entering `cd $(brew --prefix oh-my-posh)`, then just `cd themes` and `ls` for the list. For Ubuntu-20.04 running via WSL, the path is likely to be something like: `\\wsl.localhost\Ubuntu-20.04\home\linuxbrew\.linuxbrew\Cellar\oh-my-posh\6.34.1\themes`. You can view what the themes look like in the Oh My Posh Themes docs.

To use a theme, copy it from the `themes` folder to your `$Home` folder, then add this line to the bottom of the `.profile` file found in your `$Home` folder:

Bash

```
eval "$(oh-my-posh init bash --config ~/jandedobbeleer.omp.json)"
```

You can replace `jandedobbeleer.omp.json` with the name of the theme you want to use (just make sure that it is copied in your `$Home` folder).

Alternatively, if you are using oh-my-posh in both Windows with PowerShell and with WSL, you can share your PowerShell theme with WSL by pointing to a theme in your Windows user's home folder. In your WSL distribution's `.profile` path, replace ~ with the path: `/mnt/c/Users/<WINDOWSUSERNAME>`. Replacing `<WINDOWSUSERNAME>` with your own Windows username.

You can customize the Oh My Posh themes if desired.

Use Terminal-Icons to add missing folder or file icons

Terminal-Icons is a PowerShell module that adds file and folder icons that may be missing when displaying files or folders in Windows Terminal, looking up their appropriate icon based on name or extension. It attempts to use icons for well-known files/folders, but falls back to a generic file or folder icon if one is not found.

To install Terminal-Icons with PowerShell, use the command:

```PowerShell
Install-Module -Name Terminal-Icons -Repository PSGallery
```

For more information, including usage and commands, see the Terminal-Icons repo on GitHub.

Additional resources

- Oh my Posh documentation
- Terminal-Icons Repository
- Posh-Git documentation: Posh-Git is a PowerShell module that integrates Git and PowerShell by providing Git status summary information that can be displayed in the PowerShell prompt.
- PowerLine documentation: Powerline is a statusline plugin for vim, and provides statuslines and prompts for several other applications, including zsh, bash, tmux, IPython, Awesome, i3 and Qtile.

Tutorial: SSH in Windows Terminal

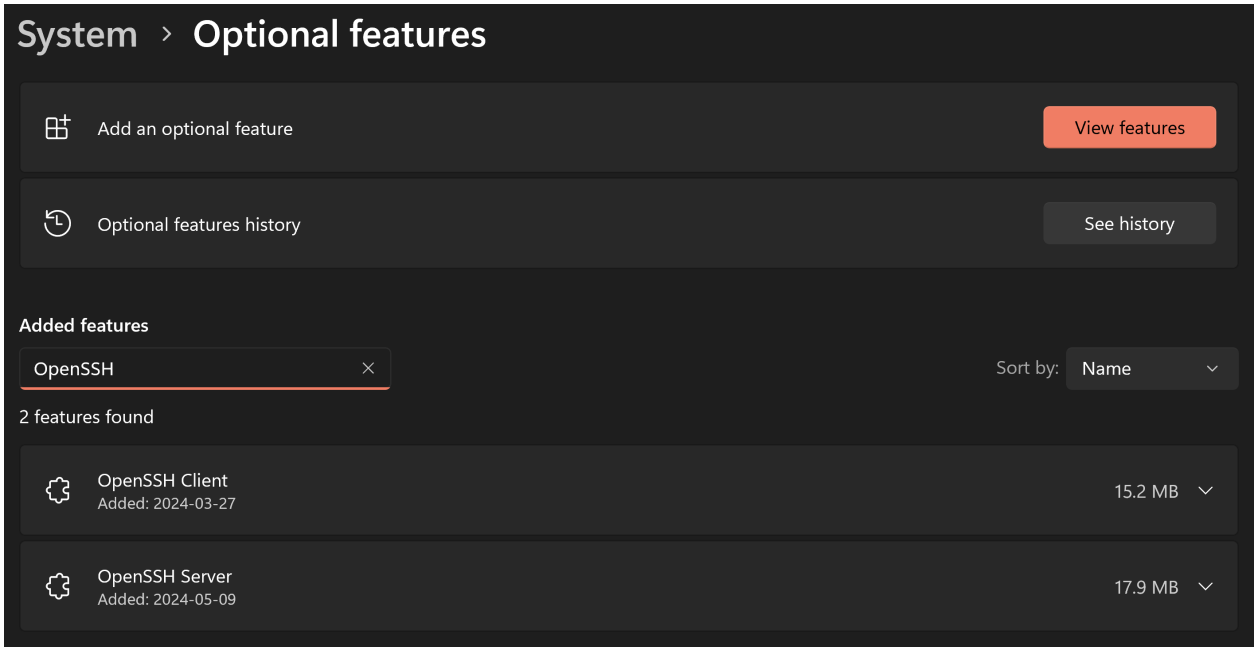
Article • 05/29/2024

Windows has a built-in SSH client and SSH server that you can use in Windows Terminal. In this tutorial, you'll learn how to set up a profile in Windows Terminal that uses SSH. Note that this feature is in preview.

Access Windows SSH Client and SSH Server

The latest builds of Windows 10 and Windows 11 include a built-in SSH server and client that are based on OpenSSH, a connectivity tool for remote sign-in that uses the SSH protocol. OpenSSH encrypts all traffic between client and server to eliminate eavesdropping, connection hijacking, and other attacks.

By default, the OpenSSH client and OpenSSH server are located in the directory: `C:\Windows\System32\OpenSSH`. You can also check that it is present in Windows Settings > System > Optional features, then search for "OpenSSH" in your added features.



For more information on configuring OpenSSH, see OpenSSH Server configuration for Windows.

Create a profile

You can start an SSH session in your command prompt by executing `ssh user@machine` and you will be prompted to enter your password. You can create a Windows Terminal

profile that does this on startup by adding the `commandline` setting to a profile in your settings.json file inside the `list` of profile objects.

JSON

```
{
  "name": "user@machine ssh profile",
  "commandline": "ssh user@machine"
}
```

For more information, see:

- Windows Terminal Profile - General settings

Specify starting directory

To specify the starting directory for a ssh session invoked by Windows Terminal, you can use this command:

JSON

```
{
  "commandline": "ssh -t bob@foo \"cd /data/bob && exec bash -l\""
}
```

The `-t` flag forces pseudo-terminal allocation. This can be used to execute arbitrary screen-based programs on a remote machine, e.g. when implementing menu services. You will need to use escaped double quotes as bourne shell derivatives don't do any additional parsing for a string in single quotes.

For more information, see:

- GH Issue: How to specify the starting directory for a ssh session?
- StackOverflow: How can I ssh directly to a particular directory?

Resources

- How to Enable and Use Windows 10's New Built-in SSH Commands

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Configure tab titles in Windows Terminal

Article • 08/24/2021

By default, the tab title is set to the shell's title. If a tab is composed of multiple panes, the tab's title is set to that of the currently focused pane. If you want to customize what is set as the tab title, follow this tutorial.

In this tutorial, you learn how to:

- ✓ Use the `tabTitle` setting
- ✓ Set the shell's title
- ✓ Using the `suppressApplicationTitle` setting

Use the `tabTitle` setting

The `tabTitle` setting allows you to define the starting title for a new instance of a shell. If it is not set, the profile `name` is used instead. Each shell responds to this setting differently.

Shell	Behavior
PowerShell	The title is set.
Command Prompt	The title is set. If a command is running, it is temporarily appended to the end of the title.
Ubuntu	The title is ignored, and instead set to `user@machine:path`
Debian	The title is set.

> ⓘ Note
>
> Though Ubuntu and Debian both run bash, they have different behaviors. This is to show that different distributions may have different behaviors.

Set the shell's title

A shell has full control over its own title. However, each shell sets its title differently.

Shell	Command
PowerShell	`$Host.UI.RawUI.WindowTitle = "New Title"`
Command Prompt	`TITLE New Title`
bash*	`echo -ne "\033]0;New Title\a"`

Note that some Linux distributions (e.g. Ubuntu) set their title automatically as you interact with the shell. If the above command doesn't work, run the following command:

Bash

```
export PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\
[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '
echo -ne '\033]0;New Title\a'
```

This will change the title to 'New Title'.

For easier access add this to the end of your `~/.bashrc`:

Bash

```
settitle () {
  export PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\
[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '
  echo -ne '\033]0;'"$1"'\a'
}
```

After you reopen your shell, you now can change the shell's title at any time using the following command:

Bash

```
settitle 'New Title'
```

Use the `suppressApplicationTitle` setting

Since a shell has control over its title, it may choose to overwrite the tab title at any time. For example, the `posh-git` module for PowerShell adds information about your Git repository to the title.

Windows Terminal allows you to suppress changes to the title by setting `suppressApplicationTitle` to `true` in your profile. This makes new instances of the

profile set your visible title to `tabTitle`. If `tabTitle` is not set, the visible title is set to the profile's `name`.

Note that this decouples the shell's title from the visible title presented on the tab. If you read the shell's variable where the title is set, it may differ from the tab's title.

Resources

- Setting the console title to be your current working directory
- Change the title of a terminal on Ubuntu 16.04

Tutorial: Opening a tab or pane in the same directory in Windows Terminal

Article • 12/21/2022

Typically, the "new tab" and "split pane" actions will always open a new tab/pane in whatever the `startingDirectory` is for that profile. However, on other platforms, it's common for new tabs to automatically use the working directory of the current tab as the starting directory for a new tab. This allows the user to quickly multitask in a single directory.

Unfortunately, on Windows, it's tricky to determine what the current working directory ("CWD") for a process is. Even if we were able to look it up, not all applications actually set their CWD as they navigate. Notably, Windows PowerShell doesn't change its CWD as you `cd` around the file system! Duplicating the CWD of PowerShell automatically would almost always be wrong.

Fortunately, there's a workaround. Applications can emit a special escape sequence (specifically the "OSC 9;9" format) to manually tell the Terminal what the CWD should be.

In this tutorial, you learn how to:

- ✓ Configure the shell to tell the Terminal about its current working directory
- ✓ Use the `duplicateTab` action to open a tab with the same CWD
- ✓ Use the `splitPane` action to open a pane with the same CWD
- ✓ Using the tab context menu to open tabs or panes with the same CWD

Configure your shell

To tell the Terminal what the CWD is, you'll need to modify your shell to emit an escape sequence as you navigate the OS. Fortunately, most shells have a mechanism for configuring the "prompt", which is run after every command. This is the perfect place to add such output.

Windows

Command Prompt: `cmd.exe`

`cmd` uses the `%PROMPT%` environment variable to configure the prompt. You can easily prepend the prompt with the command to set the CWD with the following command:

Windows Command Prompt

```
set PROMPT=$e]9;9;$P$e\%PROMPT%
```

This will append `$e]9;9;$P$e\` to your current prompt. When cmd evaluates this prompt, it'll replace

- the `$e` with the escape character
- the `$p` with the current working directory

Note that the above command will only work for the current `cmd.exe` session. To set the value permanently, AFTER running the above command, you'll want to run

Windows Command Prompt

```
setx PROMPT "%PROMPT%"
```

PowerShell: `powershell.exe` or `pwsh.exe`

If you've never changed your PowerShell prompt before, you should check out about_Prompts first.

Add the following to your PowerShell profile:

PowerShell

```
function prompt {
  $loc = $executionContext.SessionState.Path.CurrentLocation;

  $out = ""
  if ($loc.Provider.Name -eq "FileSystem") {
    $out += "$([char]27)]9;9;`"$($loc.ProviderPath)`"$([char]27)\"
  }
  $out += "PS $loc$('>' * ($nestedPromptLevel + 1)) ";
  return $out
}
```

PowerShell with posh-git

If you're using posh-git, then that will already modify your prompt. In that case, you'll want to only add the necessary output to the already modified prompt. The following

example is a lightly modified version of this example from the ConEmu docs:

PowerShell

```
function prompt
{
  $loc = Get-Location

  $prompt = & $GitPromptScriptBlock

  $prompt += "$([char]27)]9;12$([char]7)"
  if ($loc.Provider.Name -eq "FileSystem")
  {
    $prompt += "$([char]27)]9;9;`"$($loc.ProviderPath)`"$([char]27)\"
  }

  $prompt
}
```

PowerShell with Starship

If you're using Starship, then that will already modify your prompt. In that case, you'll want to only add the necessary output to the already modified prompt.

PowerShell

```
function Invoke-Starship-PreCommand {
  $loc = $executionContext.SessionState.Path.CurrentLocation;
  $prompt = "$([char]27)]9;12$([char]7)"
  if ($loc.Provider.Name -eq "FileSystem")
  {
    $prompt += "$([char]27)]9;9;`"$($loc.ProviderPath)`"$([char]27)\"
  }
  $host.ui.Write($prompt)
}
```

WSL

Windows Subsystem for Linux distributions primarily use BASH as the command line shell.

`bash`

Add the following line to the end of your `.bash_profile` config file:

Bash

```
PROMPT_COMMAND=${PROMPT_COMMAND:+"$PROMPT_COMMAND "}'printf "\e]9;9;%s\e\\"
"$(wslpath -w "$PWD")"'
```

The `PROMPT_COMMAND` variable in bash tells bash what command to run before displaying the prompt. The `printf` statement is what we're using to append the sequence for setting the working directory with the Terminal. The `$(wslpath -w "$PWD")` bit will invoke the `wslpath` executable to convert the current directory into its Windows-like path. The `${PROMPT_COMMAND:+"$PROMPT_COMMAND; "}` bit is some bash magic to make sure we append this command to any existing command (if you've already set `PROMPT_COMMAND` somewhere else.)

zsh

Add the following lines to the end of your `.zshrc` file:

zsh

```zsh
keep_current_path() {
  printf "\e]9;9;%s\e\\" "$(wslpath -w "$PWD")"
}
precmd_functions+=(keep_current_path)
```

The `precmd_functions` hook tells zsh what commands to run before displaying the prompt. The `printf` statement is what we're using to append the sequence for setting the working directory with the Terminal. The `$(wslpath -w "$PWD")` bit will invoke the `wslpath` executable to convert the current directory into its Windows-like path. Using `precmd_functions+=` make sure we append the `keep_current_path` function to any existing function already defined for this hook.

Fish

If you're using Fish shell, add the following lines to the end of your config file located at `~/.config/fish/config.fish`:

Bash

```
function storePathForWindowsTerminal --on-variable PWD
    if test -n "$WT_SESSION"
      printf "\e]9;9;%s\e\\" (wslpath -w "$PWD")
    end
end
```

This function will be called whenever the current path is changed to confirm the current session is opened by Windows Terminal (verifying $WT_SESSION) and sending Operating System Command (OSC 9;9;), with the Windows equivalent path (`wslpath -w`) of current path.

MINGW

For MINGW, Git Bash and Cygwin, you need to modify the `PROMPT_COMMAND` for WSL: replace `wslpath` with `cygpath`.

Add the following line to the end of your `.bashrc` file:

Bash

```
PROMPT_COMMAND=${PROMPT_COMMAND:+"$PROMPT_COMMAND; "}'printf "\e]9;9;%s\e\\" "`cygpath -w "$PWD" -C ANSI`"'
```

> **Note**
>
> Don't see your favorite shell here? If you figure it out, feel free to open a PR to contribute a solution for your preferred shell!

Using actions to duplicate the path

Once you've got the shell configured to tell the Terminal what the current directory is, opening a new tab or pane with that path is easy.

Open a new tab with `duplicateTab`

To open a new tab with the same path (and profile) as the currently active terminal, use the "Duplicate Tab" action. This is bound by default to `Ctrl+Shift+D`, as follows:

JSON

```
        { "command": "duplicateTab", "keys": "ctrl+shift+d" },
```

(see duplicateTab) for more details.

Open a new pane with `splitPane`

To open a new pane with the same path (and profile) as the currently active terminal, use the "Duplicate Pane" action. This is **NOT** bound by default. The simplest form of this action is:

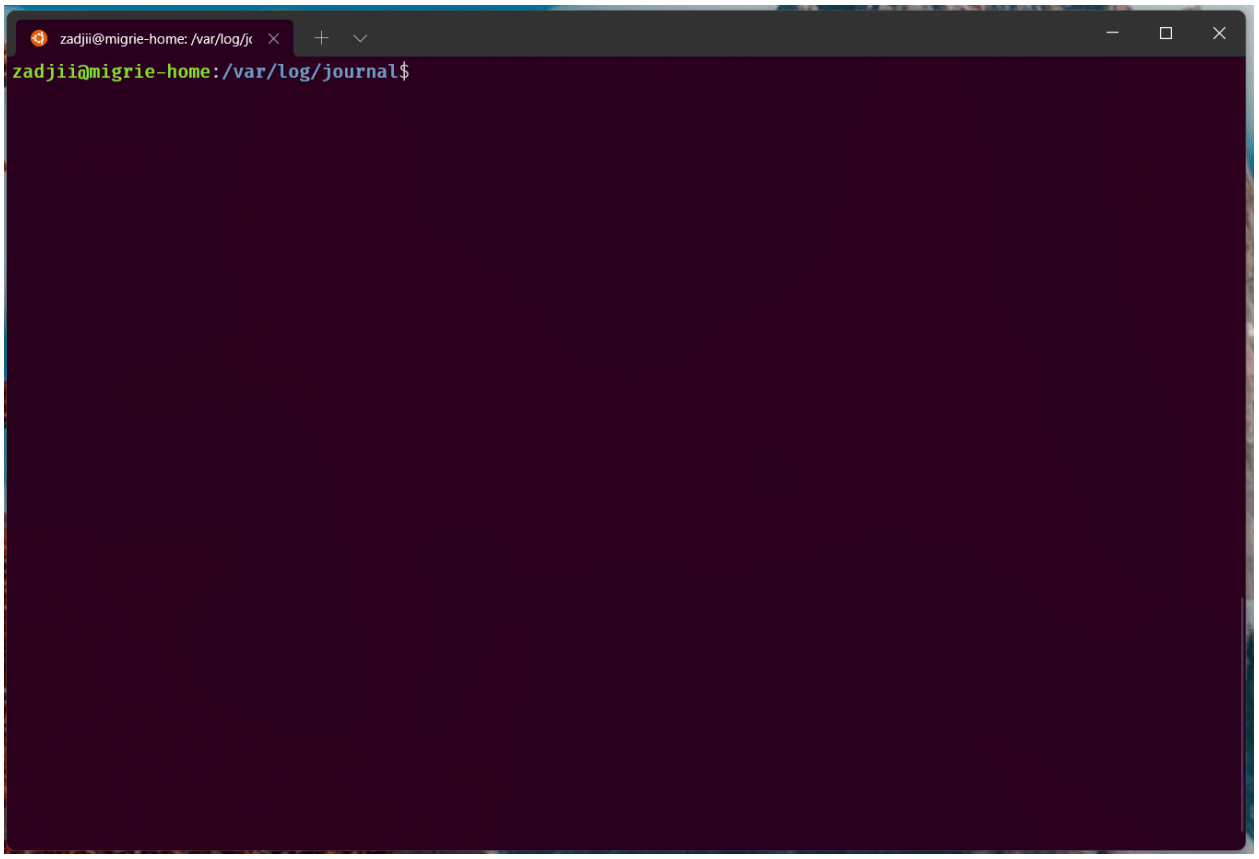
JSON

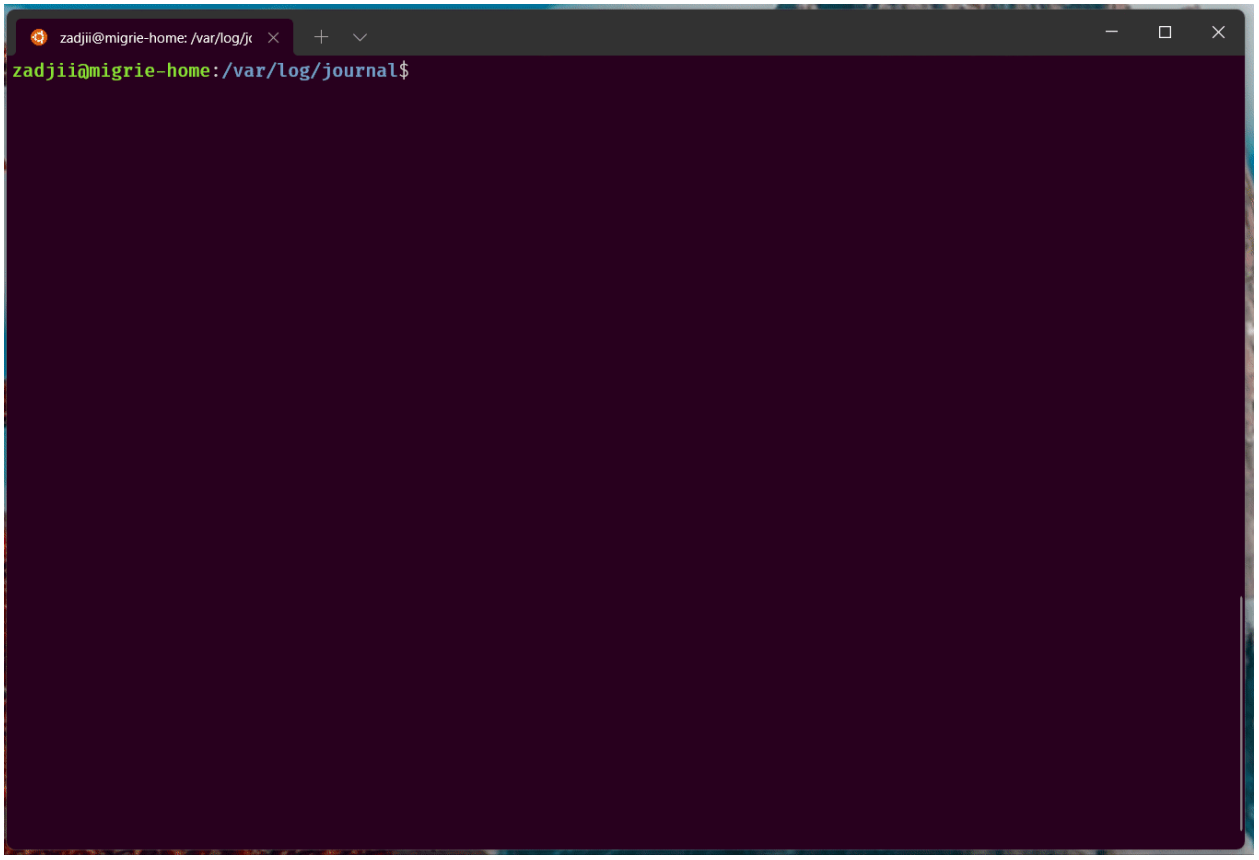
```
        { "command": { "action": "splitPane", "splitMode": "duplicate" } },
```

(see splitPane) for more details.

Using the menu to duplicate the path

the above actions are also available on the tab context menu, under the entries "Duplicate Tab" and "Split Pane".


zadjii@migrie-home: /var/log/jo
zadjii@migrie-home:/var/log/journal$


zadjii@migrie-home: /var/log/jo
zadjii@migrie-home:/var/log/journal$

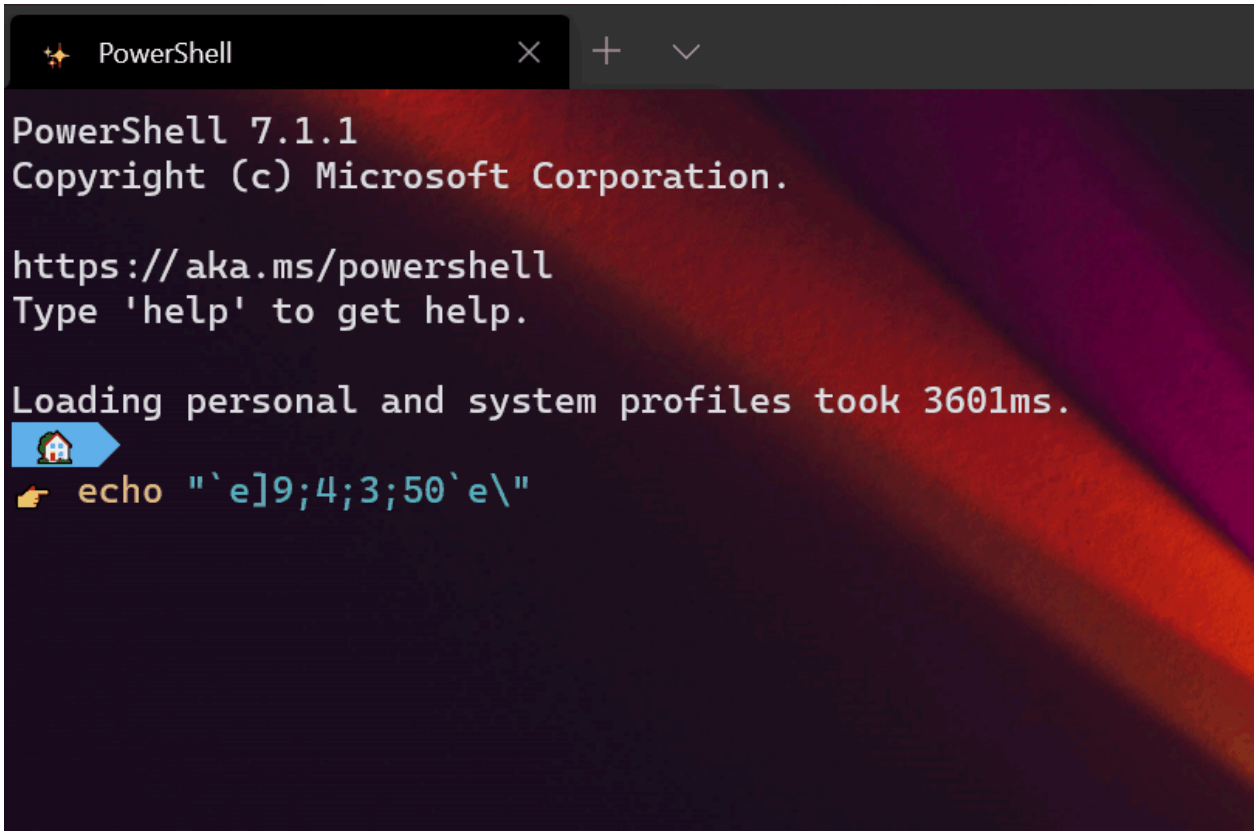
Tutorial: Set the progress bar in the Windows Terminal

Article • 09/28/2023

The Windows Terminal supports the ConEmu "Progress Bar" sequences, also known as "OSC 9;4". These sequences allow a command-line application to display a progress bar in the terminal window. This is useful for long-running commands, such as copying large files or deploying applications.

In the Windows Terminal, the progress bar is displayed in two places:

- In the tab header, as a progress ring
- In the Windows taskbar, in the same manner as a download progress bar.



Prerequisites

- Windows Terminal v1.6 or later.
- For the taskbar animation, you'll need to make sure "Show animations in Windows" is enabled in "Settings / Ease of Access / Display".

Progress bar sequence format

To set the progress bar, you need to send the OSC 9;4 sequence to the terminal. This sequence has the following format:

```text
ESC ] 9 ; 4 ; <state> ; <progress> BEL
```

- `ESC` is the escape character, ASCII 27.
- `BEL` is the bell character, ASCII 7.
- `<state>` is one of `0`, `1`, `2`, `3`, or `4`.
  - `0` is the default state, and indicates that the progress bar should be hidden. Use this state when the command is complete, to clear out any progress state.
  - `1`: set progress value to `<progress>`, in the "default" state.
  - `2`: set progress value to `<progress>`, in the "Error" state
  - `3`: set the taskbar to the "Indeterminate" state. This is useful for commands that don't have a progress value, but are still running. This state ignores the `<progress>` value.
  - `4`: set progress value to `<progress>`, in the "Warning" state
- `<progress>` is a number between 0 and 100, inclusive.

Examples

PowerShell

```PowerShell
# Set the progress bar to 50%
Write-Host -NoNewline ([char]27 + "]9;4;1;50" + [char]7)
```

Or, alternatively, in PowerShell 7:

```PowerShell
# Set the progress bar to 50%
Write-Host -NoNewline ("`e]9;4;1;50`a")
```

Bash

```
```

Bash

```bash
# Set the progress bar to 50%
echo -ne "\033]9;4;1;50\a"
```

C#

C#

```csharp
// Set the progress bar to 50%
Console.Write("\x1b]9;4;1;50\x07");
```

Command Prompt

Command Prompt is a little trickier, since it doesn't have great support for escape sequences. You can use the `echo` command to send the escape sequence, but you'll need to use literal ESC and BEL characters in the file. These might be rendered as boxes in the web browser, but they should work in the terminal.

bat

```bat
<NUL set /p =←]9;4;1;50
echo Started progress (normal, 50)
```

The above example uses the `NUL` device to write the escape sequence to the console without a newline.

> **Note**: Don't see your favorite shell here? If you figure it out, feel free to contribute a solution for your preferred shell!

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Custom Terminal guide

Article • 12/21/2022

Here are some color schemes for you to try or use as the basis of your own designs.

Installing schemes

Copy the JSON from the "**schemes**" section into the correct section in settings.json, for example:

Before:

JSON

```
"schemes": [],
```

After:

JSON

```
 "schemes": [
            {
                "name": "Retro",
                "background": "#000000",
                "black": "#00ff00",
                "blue": "#00ff00",
                "brightBlack": "#00ff00",
                "brightBlue": "#00ff00",
                "brightCyan": "#00ff00",
                "brightGreen": "#00ff00",
                "brightPurple": "#00ff00",
                "brightRed": "#00ff00",
                "brightWhite": "#00ff00",
                "brightYellow": "#00ff00",
                "cyan": "#00ff00",
                "foreground": "#00ff00",
                "green": "#00ff00",
                "purple": "#00ff00",
                "red": "#00ff00",
                "white": "#00ff00",
                "yellow": "#00ff00"
            }
        ]
```

Then add the profile-specific section, for example:

Before:

JSON

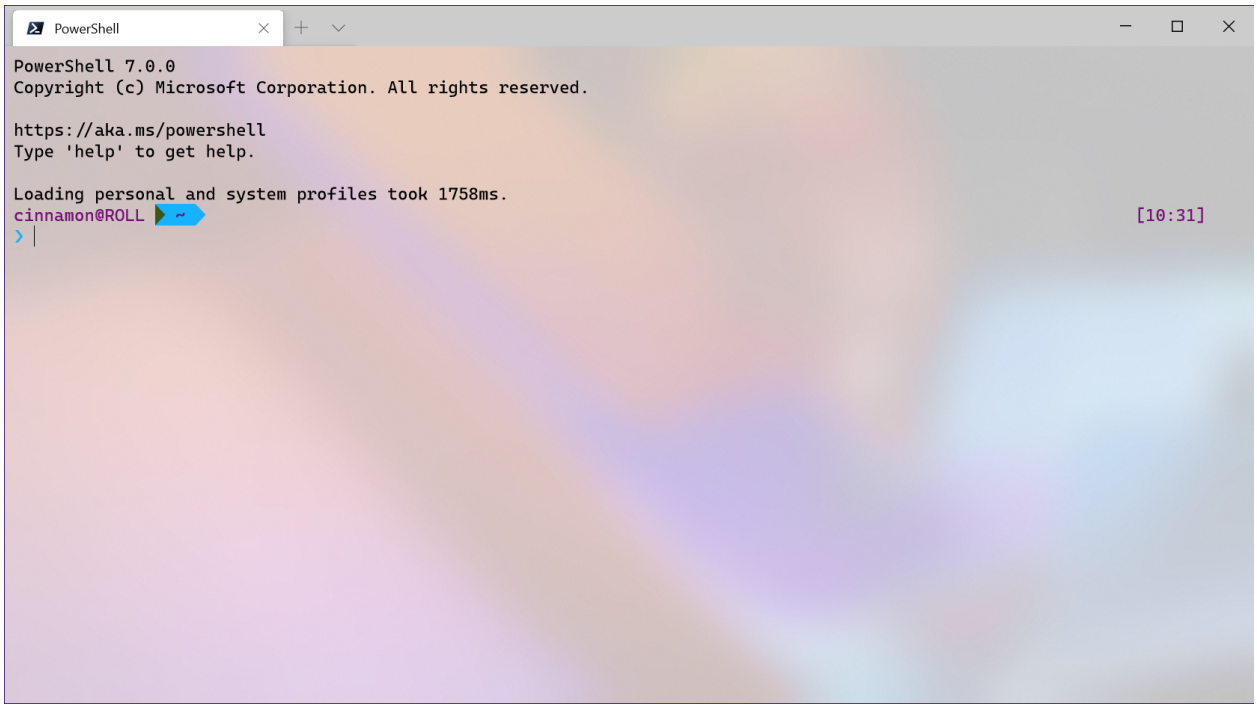
```
{
    "guid": "{234ab24f-34dd-ff3-ade434aad345}",
    "name": "Command Prompt",
    "commandline": "cmd.exe",
    "hidden": false
}
```

After:

JSON

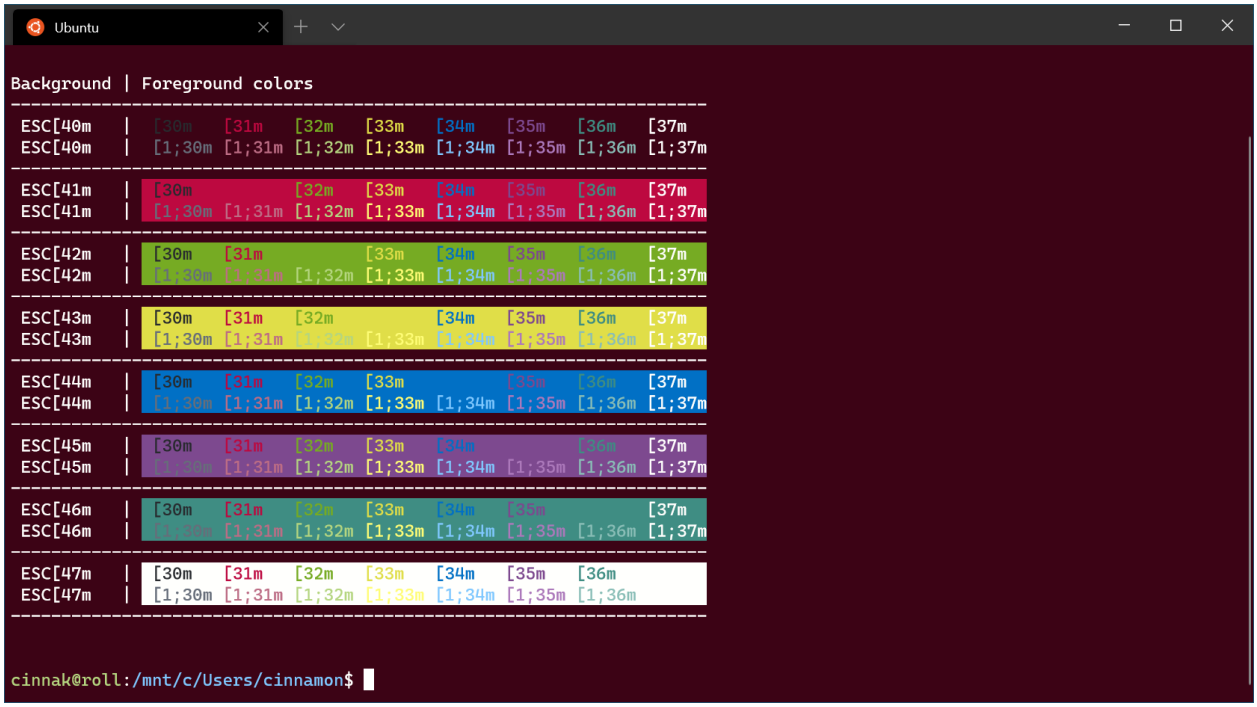
```
{
    "guid": "{234ab24f-34dd-ff3-ade434aad345}",
    "name": "Command Prompt",
    "commandline": "cmd.exe",
    "hidden": false,
    "colorScheme" : "Retro",
    "cursorColor" : "#FFFFFF",
    "cursorShape": "filledBox",
    "fontSize" : 16,
    "padding" : "5, 5, 5, 5",
    "tabTitle" : "Command Prompt",
    "fontFace": "PxPlus IBM VGA8",
    "experimental.retroTerminalEffect": true
}
```

Frosted Glass



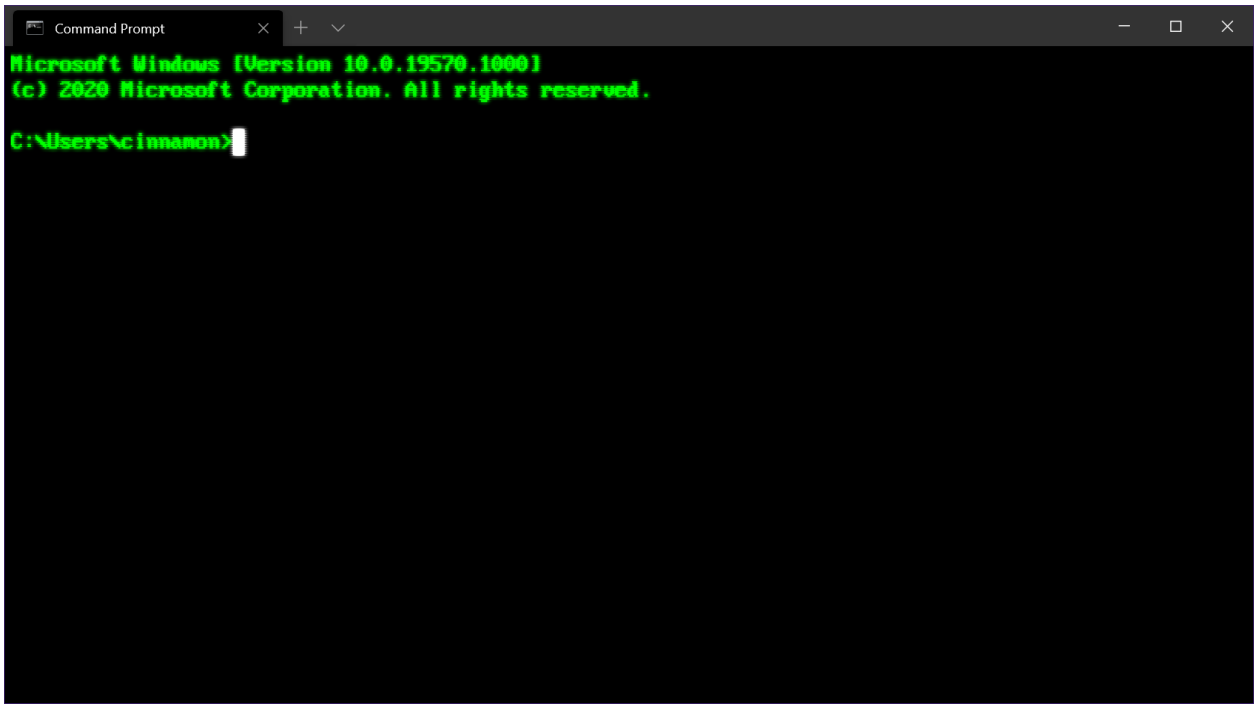
Details

Raspberry Ubuntu



Details

Retro Command



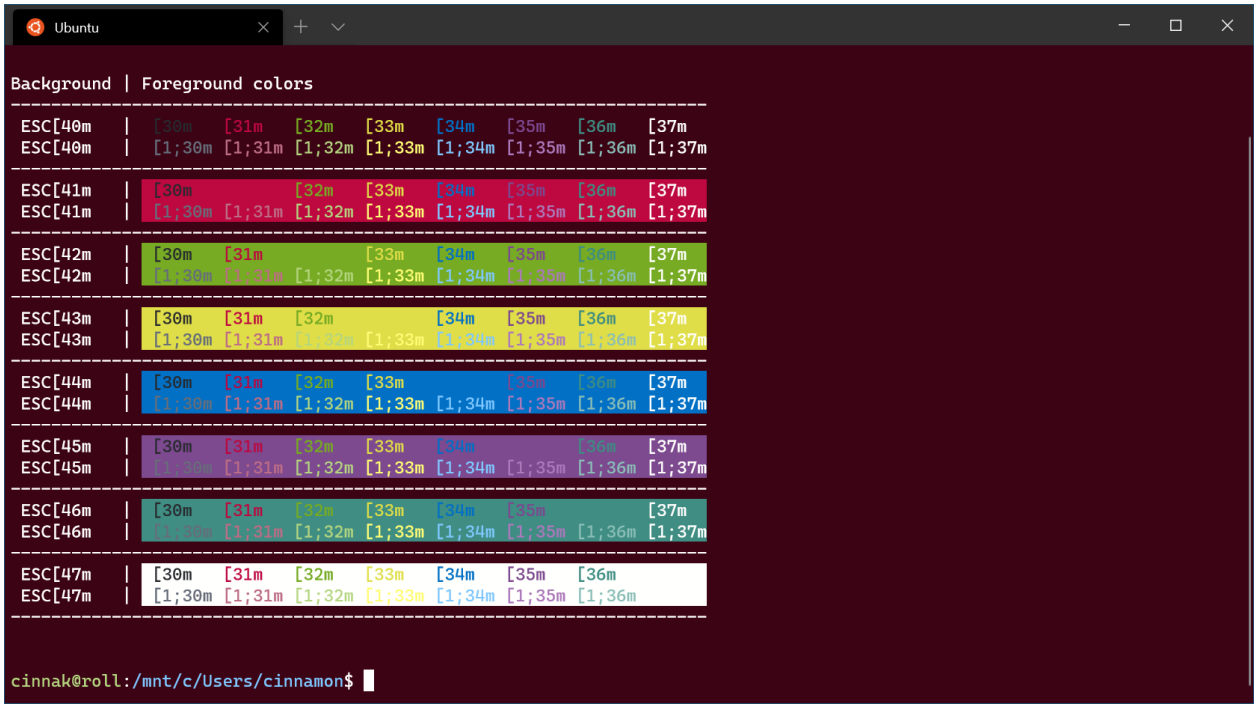
Details

Share!

Do you have a Windows Terminal scheme you would like to share? Show us on Twitter !

Raspberry Ubuntu in Windows Terminal

Article • 08/24/2021



JSON

```
{
    "theme": "dark",
    "profiles": [
        {
            "name" : "Ubuntu",
            "source" : "Windows.Terminal.Wsl",
            "colorScheme" : "Raspberry",
            "cursorColor" : "#FFFFFF",
            "fontFace" : "Cascadia Code",
            "padding" : "5, 5, 5, 5",
            "suppressApplicationTitle": true,
            "tabTitle": "Ubuntu"
        }
    ],
    "schemes": [
        {
            "name" : "Raspberry",
            "background" : "#3C0315",
            "black" : "#282A2E",
            "blue" : "#0170C5",
            "brightBlack" : "#676E7A",
            "brightBlue" : "#80c8ff",
            "brightCyan" : "#8ABEB7",
            "brightGreen" : "#B5D680",
            "brightPurple" : "#AC79BB",
            "brightRed" : "#BD6D85",
            "brightWhite" : "#FFFFFD",
```

```
                "brightYellow" : "#FFFD76",
                "cyan" : "#3F8D83",
                "foreground" : "#FFFFFD",
                "green" : "#76AB23",
                "purple" : "#7D498F",
                "red" : "#BD0940",
                "white" : "#FFFFFD",
                "yellow" : "#E0DE48"
            }
        ]
    }
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

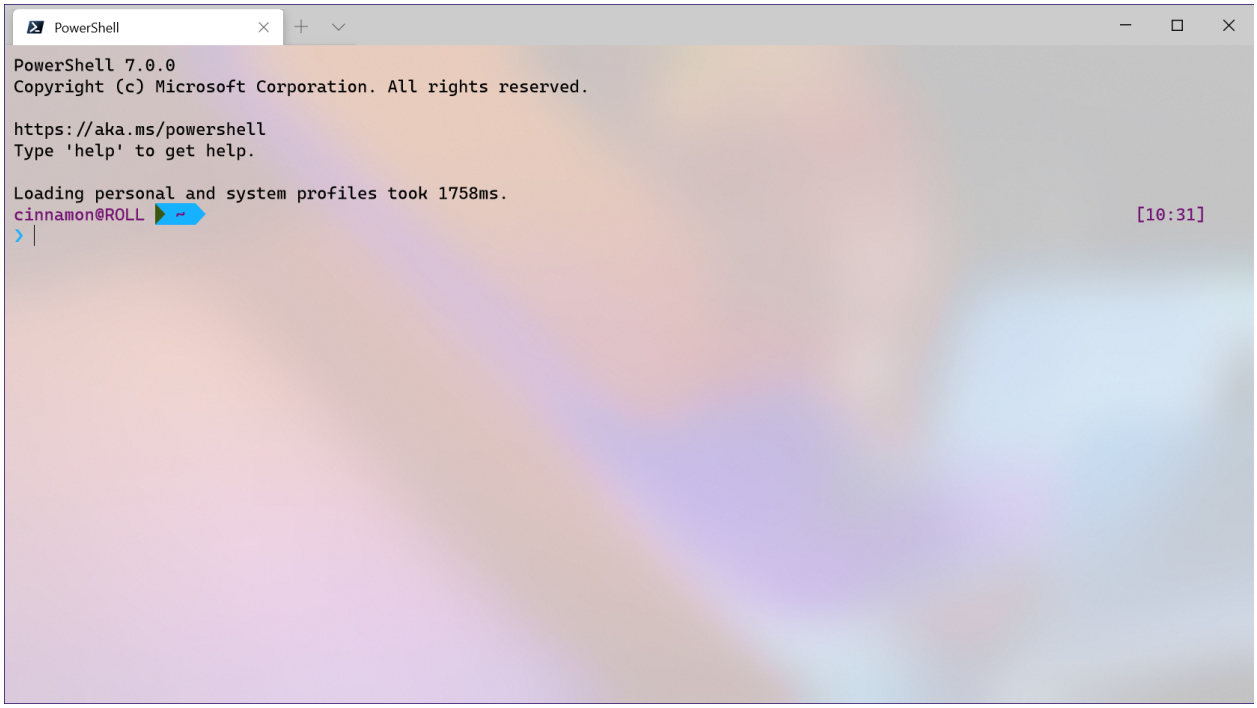
Windows Terminal is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Frosted Glass Theme in Windows Terminal

Article • 12/10/2021

The prompt is styled using Powerline and is using the `Cascadia Code PL` font, which can be downloaded from the Cascadia Code GitHub releases page.



JSON

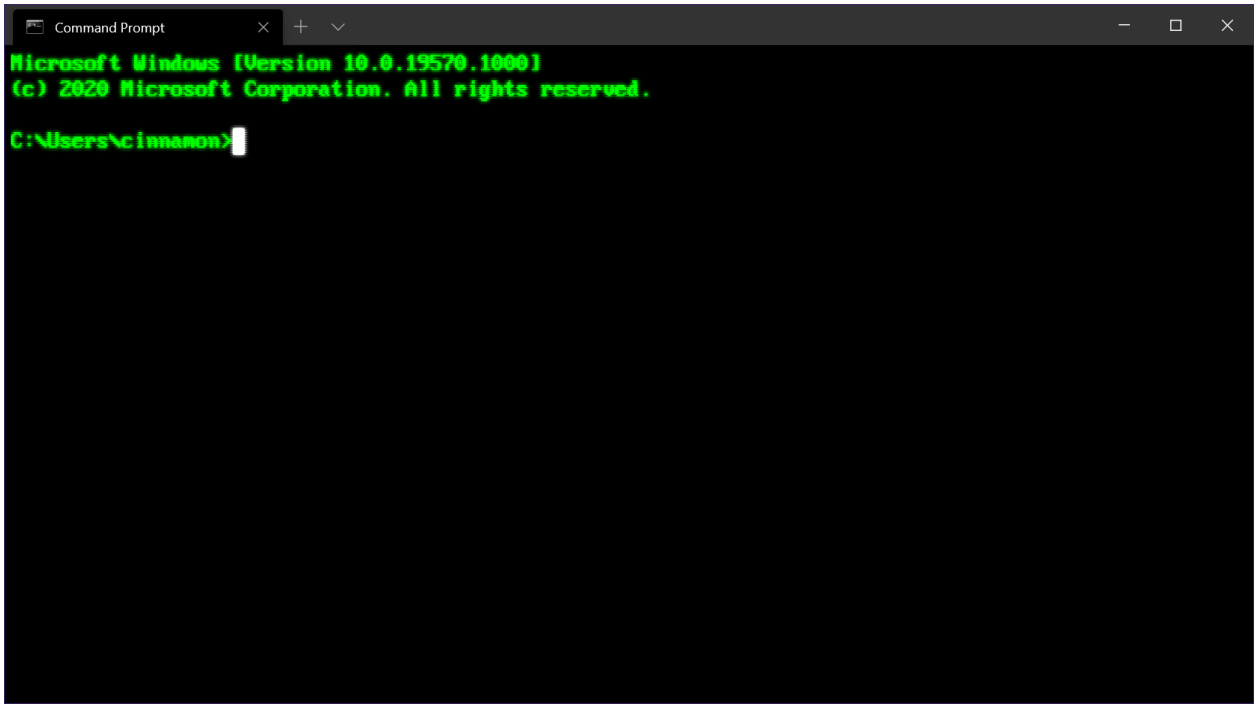
```
    {
        "theme": "light",
        "profiles": [
            {
                "name" : "PowerShell",
                "source" : "Windows.Terminal.PowershellCore",
                "acrylicOpacity": 0.7,
                "colorScheme" : "Frost",
                "cursorColor" : "#000000",
                "fontFace" : "Cascadia Code PL",
                "useAcrylic": true
            }
        ],
        "schemes": [
            {
                "name" : "Frost",
                "background" : "#FFFFFF",
                "black" : "#3C5712",
                "blue" : "#17b2ff",
                "brightBlack" : "#749B36",
                "brightBlue" : "#27B2F6",
```

```
                "brightCyan" : "#13A8C0",
                "brightGreen" : "#89AF50",
                "brightPurple" : "#F2A20A",
                "brightRed" : "#F49B36",
                "brightWhite" : "#741274",
                "brightYellow" : "#991070",
                "cyan" : "#3C96A6",
                "foreground" : "#000000",
                "green" : "#6AAE08",
                "purple" : "#991070",
                "red" : "#8D0C0C",
                "white" : "#6E386E",
                "yellow" : "#991070"
            }
        ]
    }
```

Retro Command Prompt in Windows Terminal

Article • 08/24/2021

The prompt is using the `PxPlus IBM VGA8` font, which is not included in Windows Terminal.



JSON

```
    {
        "theme": "dark",
        "profiles": [
            {
                "name": "Command Prompt",
                "commandline": "cmd.exe",
                "closeOnExit" : true,
                "colorScheme" : "Retro",
                "cursorColor" : "#FFFFFF",
                "cursorShape": "filledBox",
                "fontSize" : 16,
                "padding" : "5, 5, 5, 5",
                "tabTitle" : "Command Prompt",
                "fontFace": "PxPlus IBM VGA8",
                "experimental.retroTerminalEffect": true
            }
        ],
        "schemes": [
            {
                "name": "Retro",
                "background": "#000000",
```

```
            "black": "#00ff00",
            "blue": "#00ff00",
            "brightBlack": "#00ff00",
            "brightBlue": "#00ff00",
            "brightCyan": "#00ff00",
            "brightGreen": "#00ff00",
            "brightPurple": "#00ff00",
            "brightRed": "#00ff00",
            "brightWhite": "#00ff00",
            "brightYellow": "#00ff00",
            "cyan": "#00ff00",
            "foreground": "#00ff00",
            "green": "#00ff00",
            "purple": "#00ff00",
            "red": "#00ff00",
            "white": "#00ff00",
            "yellow": "#00ff00"
        }
    ]
}
```

Windows Terminal FAQ

FAQ

Find answers to some of the most frequently asked questions about Windows Terminal.

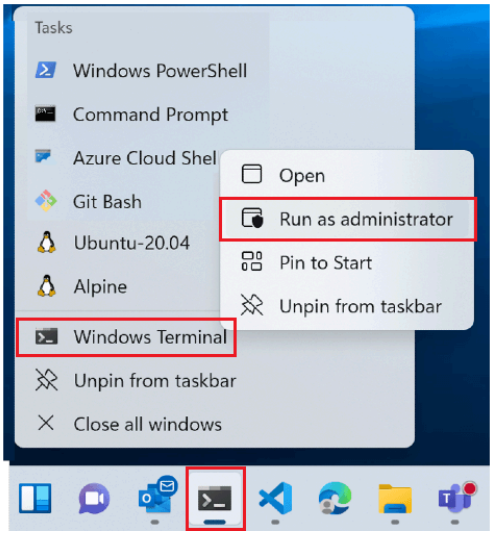
Where can I find the settings file?

The settings file can be found in the following location:

- Windows Terminal (Stable): `%localappdata%\Packages\Microsoft.WindowsTerminal_8wekyb3d8bbwe\LocalState\settings.json`
- Windows Terminal (Preview): `%localappdata%\Packages\Microsoft.WindowsTerminalPreview_8wekyb3d8bbwe\LocalState\settings.json`
- Windows Terminal (Canary): `%localappdata%\Packages\Microsoft.WindowsTerminalCanary_8wekyb3d8bbwe\LocalState\settings.json`
- Unpackaged distribution: `%localappdata%\Microsoft\Windows Terminal\settings.json`

How do I run a shell in Windows Terminal in administrator mode?

To run Windows Terminal with elevated administrator permission (admin mode), right-click on the Windows Terminal icon, then again on the Windows Terminal title that displays, then select "Run as administrator".



Alternatively, you can open the Windows Quick Access menu using the shortcut, ⊞ Windows key + X, and then selecting Windows Terminal (Admin).

Is it possible to mix admin and non-admin tabs in a Windows Terminal window?

No, mixing administrator-level permission tabbed shells with those that do not have elevated administrator permission is not supported due to security concerns.

Can I use Windows Terminal as the integrated terminal in VSCode?

No, Visual Studio Code is xtermjs and written in TypeScript while Windows Terminal is native code.

What shells does Windows Terminal support?

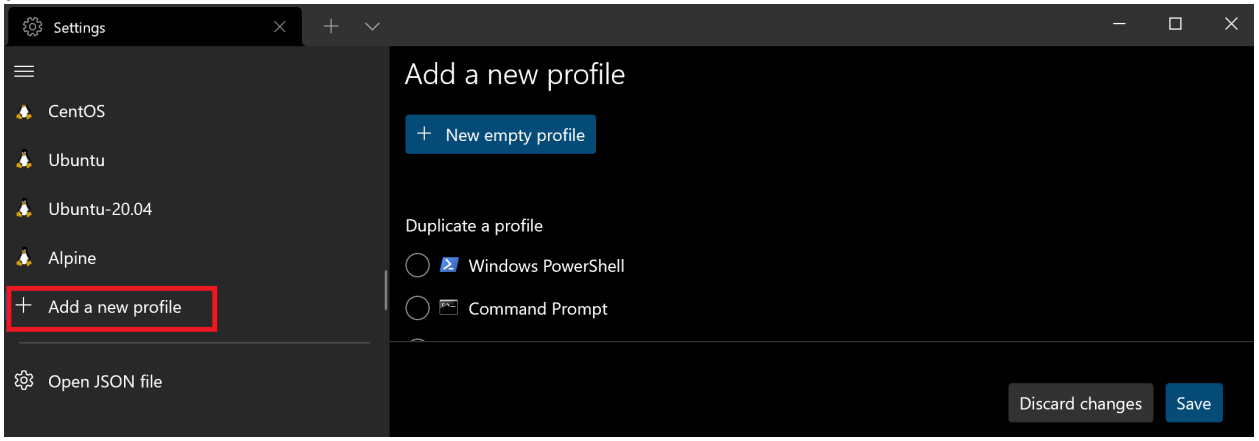
Windows Terminal will support any command line or shell that you have on your machine, including those that are included with Windows like PowerShell or Windows Command Prompt (cmd.exe), as well as any Linux distribution that can be installed with WSL, Azure Cloud Shell, Git Bash, etc. The terminal will automatically detect when you've installed a Linux distribution with WSL and create a profile for you. It can also save your Azure credentials, so you can quickly log in quickly to Azure Cloud Shell.

What is the difference between a shell and a terminal?

Windows Terminal is basically a host that enables you to run multiple command-line apps or shells side-by-side in customizable environment using tabs or window panes. Examples of "shell" applications include `cmd.exe` (the traditional Windows Command Prompt), `powershell`, or `zsh`. These are text-only applications that provide streams of characters and don't care about how they are rendered to the user. They are also sometimes referred to as "command-line client" applications. On the other hand, "terminal" applications, like Windows Terminal, gnome-terminal, xterm, iterm2, or hyper, are all graphical applications that can be used to render the output of command-line clients, customizing things like font, text size, colors, etc. On Windows, if you run `cmd.exe`, the operating system will create an instance of `conhost.exe` as the "terminal" for displaying the `cmd.exe` command-line client. The same thing happens for PowerShell, the system creates a new conhost window for any client not already connected to a terminal of some kind. Any terminal can run any command-line client application, so Windows Terminal can run any shell you prefer, such as Bash using Windows Subsystem for Linux (WSL).

How can I manually add a shell?

In your settings.json file, you can create or modify profiles that run any command-line executable. In the settings.json file, set "commandline" to whatever you want. For example, powershell --> "pwsh.exe". You can also add a profile using the terminal settings ui by scrolling to the bottom of your profiles list and selecting "+ Add a new profile".



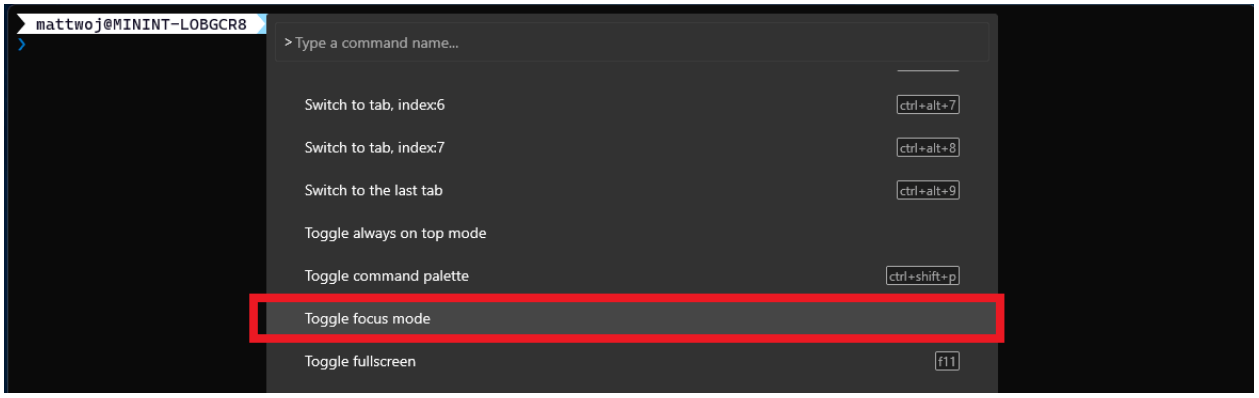
Help! The Terminal doesn't have a titlebar or tabs or any UI at all! What's

going on?

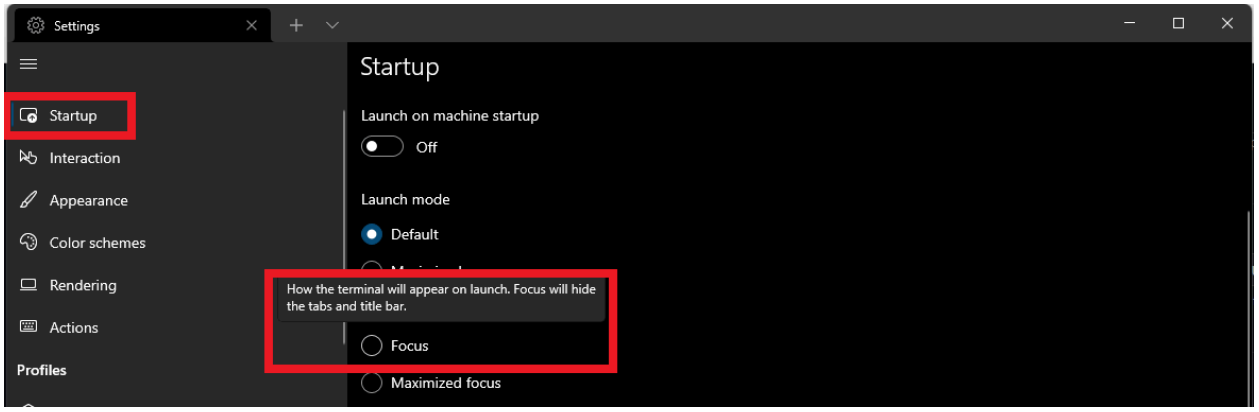
Sounds like you're in focus mode. Focus mode will hide the tabs and title bar.

How do I exit focus mode?

To exit focus mode, which hides the tabs and title bar on Windows Terminal, open the terminal command palette (Ctrl+Shift+P), find "toggle focus mode", then hit enter.



You may also want to ensure that your launch mode is not set to "focus" in the Startup settings.



What is the difference between Windows Terminal and Windows Terminal (Preview)?

Windows Terminal is the stable public release and receives regular updates that have been tested and debugged in the preview release. The recommended way to install is via the Microsoft Store, which will provide automatic updates whenever they are released. Windows Terminal Preview is a release for those interested in trying the latest features as they are being developed, tested for bugs, and becoming stable

enough to be added to the main terminal release. Features from this release are documented with the (Preview) tag.

What alternative ways are there to install Windows Terminal?

While we recommend installing Windows Terminal using the Microsoft Store, you can also install using Windows Package Manager, GitHub, Chocolatey, or Scoop.

Is it possible to initialize a Windows Terminal profile with a batch file?

Yes. You first need to go to the Profiles section of your settings.json file. Using the "commandline": property, you can specify any batch file, command, ssh connection, or executable that you want to run as a profile in Windows Terminal. You just need to enter the path to the file that you want to run.

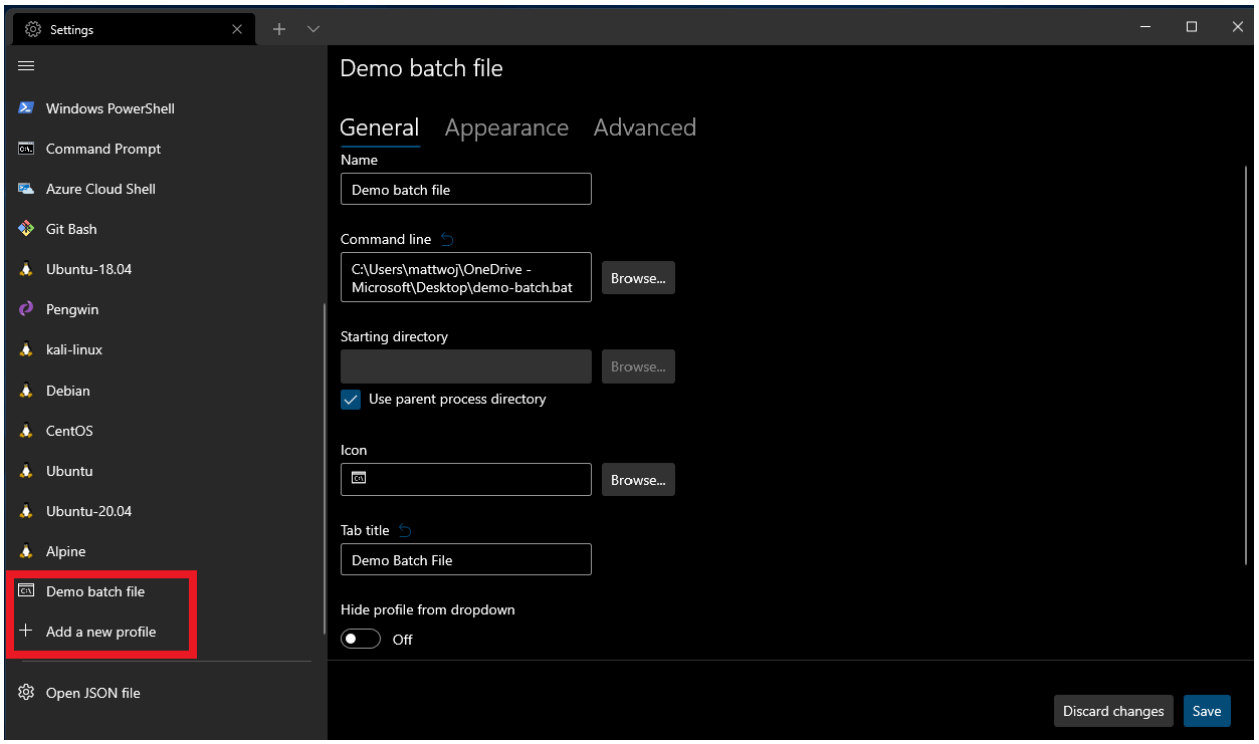
This example shows an example of a terminal profile set up based on a "demo" batch file.

JSON

```
{
"commandline": "%USERPROFILE%/OneDrive/demo.bat",
"name": "Batch Profile"
}
```

This can also be done in the Settings UI. Select "+ Add a new profile" > "+ New empty profile". Browse to the starting directory where your batch file (or SSH connection,

executable, command file, etc) is located. Give the profile a name and save.



What sort of features have open-source community contributors added to Windows Terminal?

There have been a wide variety of contributions to Windows Terminal, including bug fixes, identifying and discussing issues, contributing to this documentation, but a few of our favorite features that have come from community contributions have included support for background images and gifs, retro effects, and tab coloring, just to name a few. Learn more about how to contribute.

What is conhost.exe?

The Windows Console host, conhost.exe, is Windows' original command-line user experience. It also hosts Windows' command-line infrastructure and the Windows Console API server, input engine, rendering engine, user preferences, etc. A primary goal of Windows Console is to maintain backward compatibility, thus adding new features became prohibitive and led to the creation of the Windows Terminal. Learn more in the Windows Terminal open-source repo and in the WIndows Console docs.

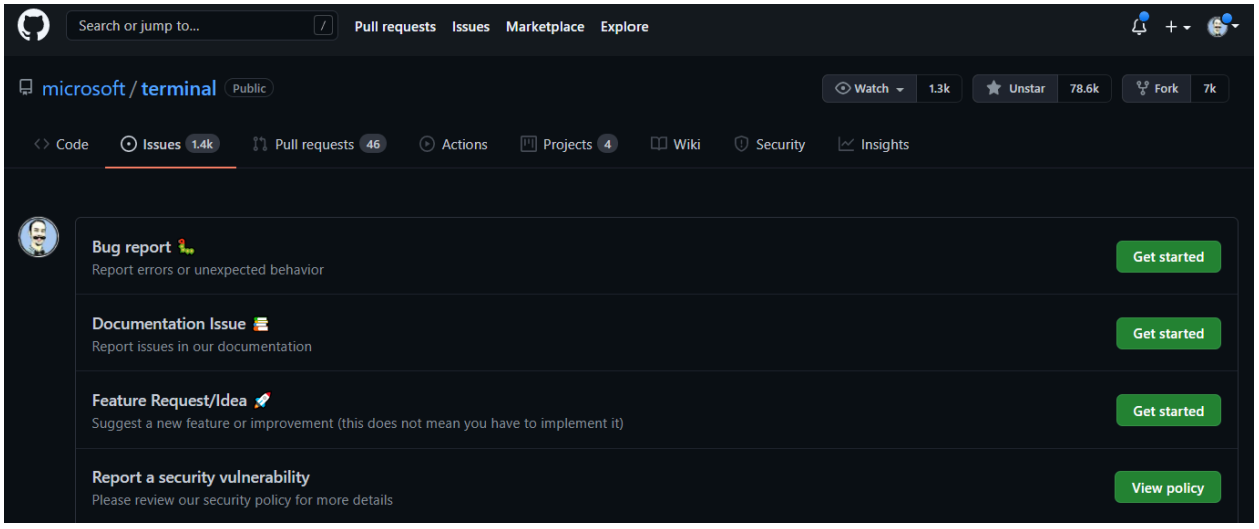
Can I save the layout of all of my open console windows when closing Windows Terminal and restore them when starting a new session?

Yes! As of Windows Terminal Preview v1.12.2922.0, you can now save window pane layouts upon closing a terminal session with the firstWindowPreference global setting.

What is being planned for Windows Terminal? Is there a development roadmap or feature request list that I can contribute to?

Windows Terminal is under very active development. You can view the team's plans on the Terminal 2.0 Roadmap document in the open-source terminal repo. New features go into Windows Terminal Preview first, then typically a month after they've been in Preview, those features move into Windows Terminal.

You can contribute feature Requests or ideas, as well as bug reports, security vulnerabilities, or documentation issues by filing an issue in the terminal repo.



How can I customize PowerShell prompt colors using PSReadLine?

The PSReadLine module is responsible for setting the colors and behavior of your PowerShell command line. You can set colors for specific attributes of your PowerShell prompt by using the Set-PSReadLineOption, see Example 4: Set multiple color options.

Can you have have tabs running as Administrator (elevated) in unelevated Terminal windows?

No. For more technical details, refer to this spec and this doc.

Why is the Windows Terminal default appearance theme set to Dark, when my Windows operating system settings are set to Light?

The system theme in Windows 11 defaults to the Light theme appearance unless changed by the user. Windows Terminal appearance color scheme, however, is Dark by default. Many users won't change the appearance themes and will just see the default themes. The goal of aligning Windows Terminal appearance with the colors of the Windows operating system (OS) titlebar leaves these options: 1) Do nothing (This is the Terminal default before v1.16). In the default setup, the result is a visually unappealing contrast between black terminal content with a light titlebar. 2) Default the color scheme of the terminal to match the OS theme and leave the app theme set to "system". In the default setup, this will result in the terminal appearing as black text on a white background. 3) Change the default theme of Windows Terminal to Dark, regardless of OS theme. In the default setup, this will result in the terminal appearing as white text on a black background, with a dark titlebar. Option 3 presented the optimum balance of the least surprise to users on the default settings, with the most aesthetically pleasing results. Windows Terminal v1.16 also introduced new toggles for customizing the appearance of the window, including: Customizing the title bar color, the tab color (including auto-matching the background), and enabling different colors for focused or unfocused windows. Windows Terminal 1.17 introduces additional flexibility with the ability to sync the Terminal theme to the OS theme and sync the color scheme to the OS theme. The Terminal is already 99% white text on a black background, these changes just line the title bar up with that.

How do I change Windows Terminal theme back to "system"?

Add `"theme": "system"` to your `settings.json`, or you can change the Theme on the "Appearance" page of the Windows Terminal Settings.

Why is the tab still black after I set Windows Terminal to Light theme?

This is a side effect of the Theme changes introduced in v1.16. The default themes in v1.16 and later will always use the Terminal background color as the default color for each tab. By default on a black terminal window, you will get a black tab. With a blue color scheme (like Campbell PowerShell), you'll get a blue tab. This is to give the Terminal a "seamless" feel. In Light mode, that creates a case where a black tab will appear on a white tab row. However, with v1.16 and later, you can customize the theme of the terminal. For example:

JSON

```
"theme": "White Tabs",
"themes":
[
    {
        "name": "White Tabs",
        "tab":
        {
            "background": "#ffffffff",
        },
        "window":
        {
            "applicationTheme": "light"
        }
    },
]
```

Alternatively, if you're using an OS Light theme and want to set the terminal color scheme to a white background, v1.17 enables setting different color schemes based on the theme of the window. For example, to change the background color based on the `window.applicationTheme` of Windows Terminal, you can do this:

JSON

```
"colorScheme":
{
    "light": "One Half Light",
    "dark": "One Half Dark",
},
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

Windows Terminal feedback

Windows Terminal is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Windows Terminal sample code

Article • 10/06/2022

Explore some of the sample code hosted on the Windows Terminal repo, including Pixel Shader .hlsl samples, an EchoCon ConPTY sample Win32 pseudo console, a GUIConsole sample WPF console targeting .NET, a MiniTerm sample using basic PTY API calls, and a ReadConsoleInputStream demo for monitoring of console events while streaming character input.

Pixel Shaders

Due to the sheer amount of computing power in GPUs, one can do awesome things with pixel shaders such as real-time fractal zoom, ray tracers and image processing.

Windows Terminal allows users to provide a pixel shader, applied to the terminal by adding the `experimental.pixelShaderPath` property to a profile in your settings.json file. Pixel shaders are written in a language called HLSL, a C-like language with some restrictions.

Try one of several Pixel Shader .hlsl samples provided in the Windows Terminal repo: Pixel Shaders.

EchoCon ConPTY sample app

This sample application illustrates how to use the Win32 Pseudo Console (ConPTY) by:

1. Creating an input and an output pipe
2. Calling CreatePseudoConsole() to create a ConPTY instance attached to the other end of the pipes
3. Spawning an instance of ping.exe connected to the ConPTY
4. Running a thread that listens for output from ping.exe, writing received text to the Console

Visit the Windows Terminal repo to find this sample: EchoCon ConPTY Sample App.

GUIConsole sample app

This sample application provides an example skeleton of a custom WPF console.

Within this sample, you will find:

- GUIConsole.WPF: a WPF application, targeting .NET 4.6.1, that creates a single WPF window which acts as the console and keeps the underlying console visible.
- GUIConsole.ConPTY: a .NET Standard 2.0 library that handles the creation of the console and enables pseudoconsole behavior. The Terminal.cs file contains the publicly visible pieces that the WPF application will interact with. Terminal.cs exposes two things that allow reading from, and writing to, the console:
  - `ConsoleOutStream`: a FileStream hooked up to the pseudoconsole's output pipe. This will output VT100.
  - `WriteToPseudoConsole`(string input): a method that will take the given string and write it to the pseudoconsole via its input pipe. This accepts VT100.

Visit the Windows Terminal repo to find this sample: GUIConsole.

MiniTerm sample app

This experimental terminal demonstrates basic API calls (not intended for "real-world" usage) using PTY APIs from Microsoft. Written in C# and heavily based on the native code examples.

Visit the Windows Terminal repo to find this sample: MiniTerm.

ReadConsoleInputStream Demo

Demonstration of asynchronous monitoring of console events (like mouse, menu, focus, buffer/viewport resize) while simultaneously streaming the character input view from the console. A particularly useful feature when working with VT100 streams and ConPTY.

Visit the Windows Terminal repo to find this demo: ReadConsoleInputStream Demo.

Troubleshooting in Windows Terminal

Article • 10/06/2022

This guide addresses some of the common errors and obstacles you may encounter when using Windows Terminal.

Opening the settings does nothing (or opens an unexpected application)

If you click on the "settings" button in the dropdown, the Terminal will attempt to open the settings file, `settings.json`. This will cause the OS to try and launch your configured `.json` file editor. This might be Visual Studio, or Notepad, or some other completely unexpected application. If there isn't a configured `.json` editor on your machine, then the OS will eventually show you the "How do you want to open this file" dialog.

> **Tip**
>
> You can also use the settings UI to configure your settings. You can learn how to open the settings UI on the **Actions page**.

Set your WSL distribution to start in the home `~` directory when launched in older versions of Windows Terminal

By default, the `startingDirectory` of a profile is `%USERPROFILE%` (`C:\Users\<YourUsername>`). This is a Windows path. For WSL distributions running a new version of Windows Terminal, the file systems can enter `~` to set this home path. In older versions of Windows Terminal, you can use `/home/<Your Ubuntu Username>` to directly refer to your home folder. For example, the following setting will launch the "Ubuntu-20.04" distribution in its home file path:

JSON

```
{
    "name": "Ubuntu-20.04",
    "commandline" : "wsl -d Ubuntu-20.04",
```

```
    "startingDirectory" : "/home/<Your Ubuntu Username>"
}
```

If you are using a very early version of Windows Terminal, WSL may require using the `\\wsl$\` prefix when referring to a distribution's home path for the `startingDirectory` setting. For example, the following setting will launch the "Ubuntu-18.04" distribution in its home file path:

JSON

```
{
    "name": "Ubuntu-18.04",
    "commandline" : "wsl -d Ubuntu-18.04",
    "startingDirectory" : "//wsl$/Ubuntu-18.04/home/<Your Ubuntu Username>"
}
```

> **ⓘ Important**
>
> On newer versions of Windows, `startingDirectory` can accept Linux-style paths.

Setting the tab title

To have the shell automatically set your tab title, visit the set the tab title tutorial. If you want to set your own tab title, open the settings.json file and follow these steps:

1. In the profile for the command line of your choice, add `"suppressApplicationTitle": true` to suppress any title change events that get sent from the shell. Adding *only* this setting to your profile will set the tab title to the name of your profile.

2. If you want a custom tab title that is not the name of your profile, add `"tabTitle": "TITLE"`. Replacing "TITLE" with your preferred tab title.

Command line arguments in PowerShell

Visit the Command line arguments page to learn how command-line arguments operate in PowerShell.

Command line arguments in WSL

Visit the Command line arguments page to learn how command-line arguments operate in WSL.

Problem setting `startingDirectory`

If the `startingDirectory` is being ignored in your profile, first check to make sure the syntax is correct in your settings.json file. To help you check this syntax, `"$schema": "https://aka.ms/terminal-profiles-schema"` is automatically injected. Some applications, like Visual Studio Code, can use that injected schema to validate your json file as you make edits.

If your settings are correct, you may be running a startup script that sets the starting directory of your terminal separately. For example, PowerShell has its own separate concept of profiles. If you are changing your starting directory there, it will take precedence over the setting defined in Windows Terminal.

Alternatively, if you are running a script using the `commandline` profile setting, it may be that you are setting the location there. Similar to PowerShell profiles, your commands there take precedence over the `startingDirectory` profile setting.

The purpose of `startingDirectory` is to launch a new Windows Terminal instance in the given directory. If the terminal runs any code that changes its directory, that may be a good place to take a look.

Ctrl+= does not increase the font size

If you are using a German keyboard layout, you may run into this problem. `Ctrl+=` gets deserialized as `Ctrl+Shift+0` if your main keyboard layout is set to German. This is the correct mapping for German keyboards.

More importantly, the app never receives the `Ctrl+Shift+0` keystroke. This is because `Ctrl+Shift+0` is reserved by Windows if you have multiple keyboard layouts active.

If you would like to disable this feature in order for `Ctrl+=` to work properly, follow the instructions for "Change Hotkeys to Switch Keyboard Layout in Windows 10" in this blog post.

Change the 'Switch Keyboard Layout' option to 'Not Assigned' (or off of `Ctrl+Shift`), then select **OK** and then **Apply**. `Ctrl+Shift+0` should now work as a key binding and is passed through to the terminal.

On the other hand, if you do use this hotkey feature for multiple input languages, you can configure your own custom key binding in your settings.json file.

The text is blurry

Some display drivers and hardware combinations do not handle scroll and/or dirty regions without blurring the data from the previous frame. To mitigate this problem, you can add a combination of these global rendering settings to reduce the strain placed on your hardware caused by the terminal text renderer.

My colors look strange! There are black bars on my screen!

> **Important**
>
> This applies only to version 1.2+ of Windows Terminal. If you are seeing color issues in Windows Terminal 1.0 or 1.1, or issues that are not captured here, please file a bug.

Windows Terminal 1.2 and beyond has an improved understanding of certain application color settings. Because of this improved understanding, we have been able to remove a number of compatibility blocks that resulted in a poor user experience. Unfortunately, there is a small number of applications that may experience issues.

We will keep this troubleshooting item up-to-date with the list of known issues and their workarounds.

Black lines in PowerShell (5.1, 6.x, 7.0)

Terminal, when coupled with PowerShell's line editing library PSReadline, may draw black lines across the screen. These miscolored regions will extend across the screen beyond your prompt wherever there are command parameters, strings or operators.

PSReadline version **2.0.3** has been released and contains a fix for this issue. If you are using the prerelease version of PSReadline, note that a fix is not yet available.

To update to the newest version of PSReadline, please run the following command:

```
PowerShell
```

```
Update-Module PSReadline
```

Why are my emojis not appearing as icons in the jumplist?

Only images linked from a file location can be rendered as profile icons in the jumplist. Emojis are not supported for jumplist icons.

Technical Notes

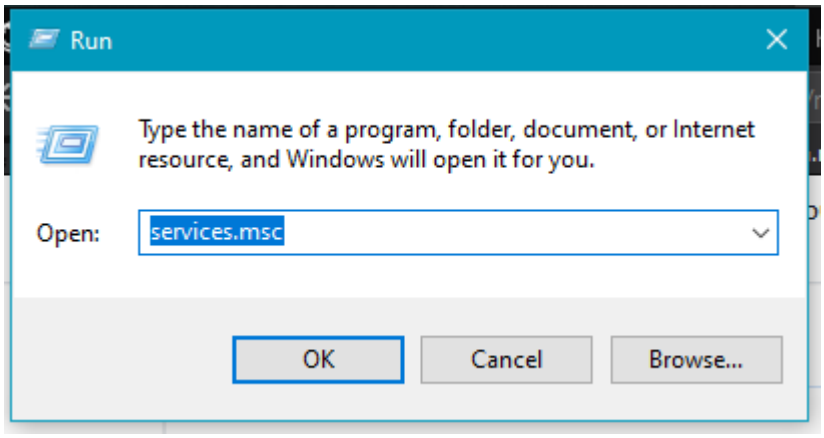
Applications that use the GetConsoleScreenBufferInfo family of APIs to retrieve the active console colors in Win32 format and then attempt to transform them into cross-platform VT sequences (for example, by transforming `BACKGROUND_RED` to `\x1b[41m`) may interfere with Terminal's ability to detect what background color the application is attempting to use.

Application developers are encouraged to choose either Windows API functions *or* VT sequences for adjusting colors and not attempt to mix them.

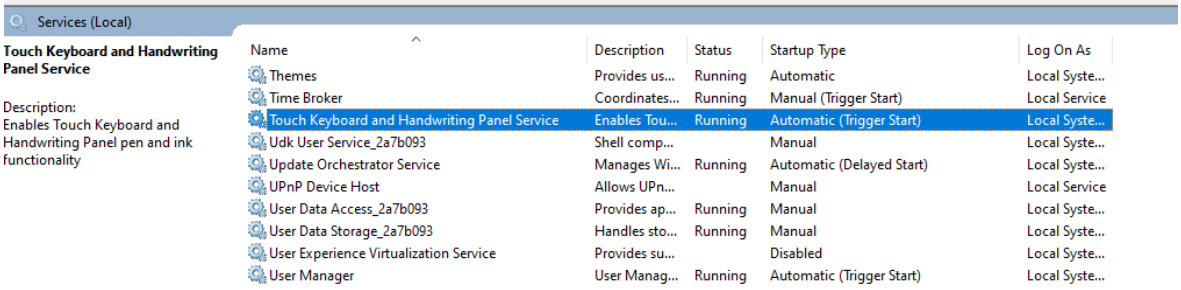
Keyboard service warning

Starting in Windows Terminal 1.5, the Terminal will display a warning if the "Touch Keyboard and Handwriting Panel Service" is disabled. This service is needed by the operating system to properly route input events to the Terminal application (as well as many other applications on Windows). If you see this warning, you can follow these steps to re-enable the service:

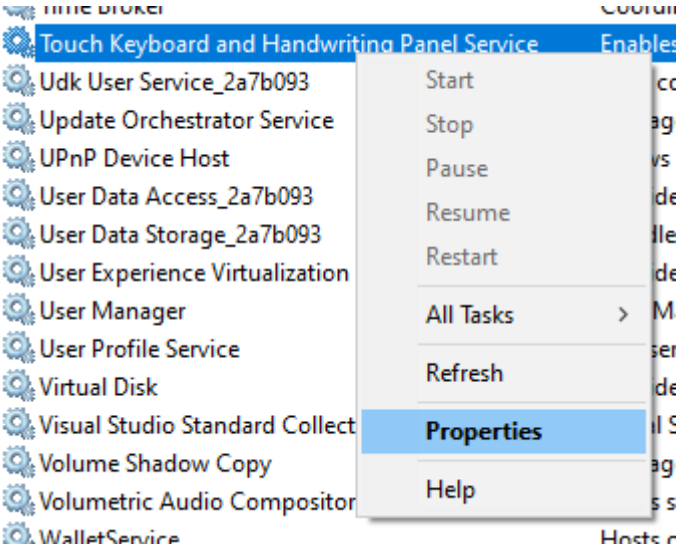
1. In the run dialog, run `services.msc`



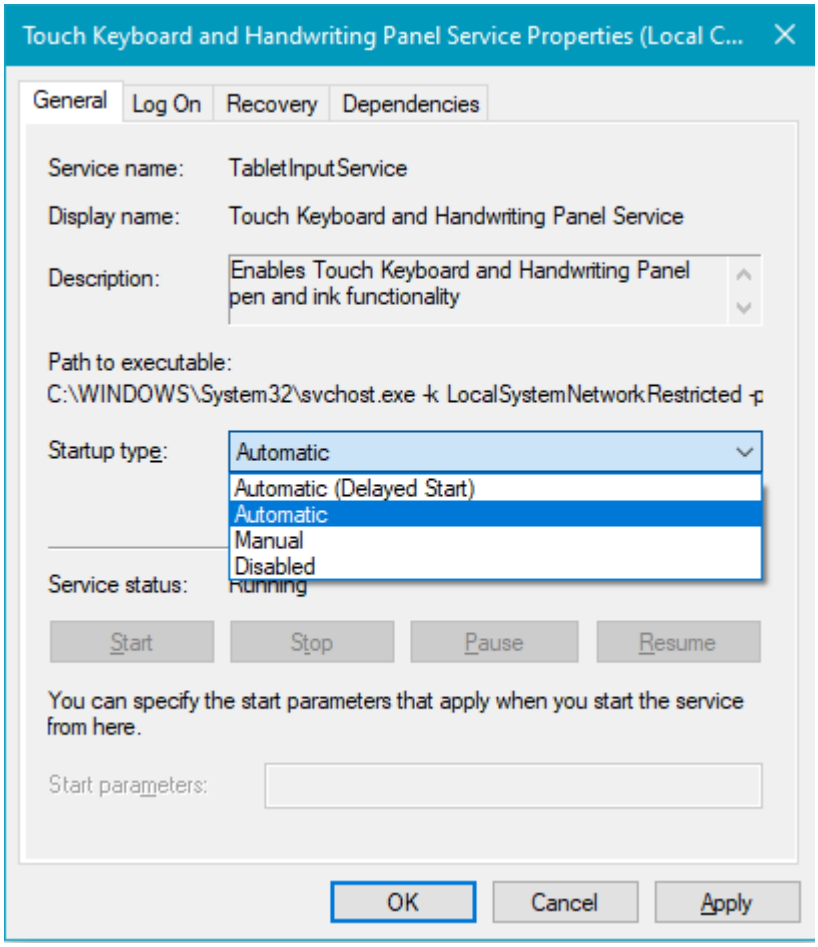
2. Find the "Touch Keyboard and Handwriting Panel Service"



3. Open the "Properties" for this service



4. Change the "startup type" to "Automatic"



5. Hit "Ok", and restart the PC.

After restarting the machine, the service should auto-start, and the dialog should no longer appear.

Why do I see blinking or flashing when using a git bash command line?

You may notice a blinking or flashing when using a git bash command line inside Windows Terminal. This behavior is actually by design. The Terminal is obeying what git bash is telling it to do (setting bell-style to visible, causing a flash to associate with the bell response), BUT we understand this may be distracting. To fix this, open the `.inputrc` file for your Git bash with a text editor. This file will likely be located in the path `C:\Program Files\Git\etc`. To open with the Nano text editor: `nano ~/.inputrc`

Change the default:

```Bash
# none, visible or audible
set bell-style visible
```

Set the bell-style to either `none` or `audible` to remove the visible flash:

```Bash
set bell-style none
```

Press Ctrl + O and Ctrl + X to Save and Exit.

How do I reset my settings in Windows Terminal back to the default settings?

To reset your settings back to the original default settings, delete your settings.json file. This will cause Windows Terminal to regenerate a settings.json file with the original default settings.

> **ⓘ Important**

As of Windows Terminal version 1.10 or greater, you'll also need to delete the `state.json` file in the same directory as the `settings.json` file to fully reset the settings to the defaults.

Why is Acrylic opacity not making my Windows Terminal background transparent?

You can set the transparency of a terminal window with the useAcrylic property. There are a few reasons why your opacity setting may not be working for Acrylic, including:

- As a system-wide policy, acrylic is only enabled for the foreground window. So if you activate any other window than the Terminal, the Terminal's acrylic will turn off.
- Acrylic doesn't work if your GPU hardware does not support it. If you're running an app in a Virtual Machine (VM) or over remote desktop, acrylic likely will not work.
- Acrylic can be disabled by the operating system for a number of reasons, like being in power saver (low-battery) mode or when accessing a machine using Remote Desktop.

Why does my mouse pointer disappear when hovering over a window and typing?

This cursor auto-hiding behavior is by design, but can be disabled in the by searching in Windows Settings for "Mouse settings" > "Additional Mouse Settings" > "Mouse Properties" > "Pointer Options" > Uncheck "Hide pointer while typing". You may need to restart your Windows Terminal in order for this change to take effect.