WINDOWS TERMINAL

Fast wie unter Linux

Einführung in die Funktionen des neuen Windows Terminals.

Einst galt Windows als Refugium der Maus-Freunde. Doch spätestens seit dem Erscheinen der PowerShell bietet Windows auch für Konsolenanhänger attraktive Lösungen. Es folgte das Windows-Subsystem für Linux (WSL), und jetzt kommt das Open-Source-Projekt Windows Terminal hinzu. Die Einführung von WSL in Windows 10 brachte Abhilfe für das in Bild 1 gezeigte Problem: Die in Linux zur Steuerung der Displayanzeige verwendeten ANSI/VT-Sequenzen waren bis dato in Windows weitgehend unbekannt, und auch sonst gab es eine Reihe von Unterschieden zwischen den Denkmodellen von Windows und unixoiden Systemen.

Wie so oft in der Welt von Windows gilt hier, dass keine allzu starken Änderungen möglich sind – spätestens mit dem Scheitern von Windows Phone war sich Microsoft der Wichtigkeit von Abwärtskompatibilität bewusst.

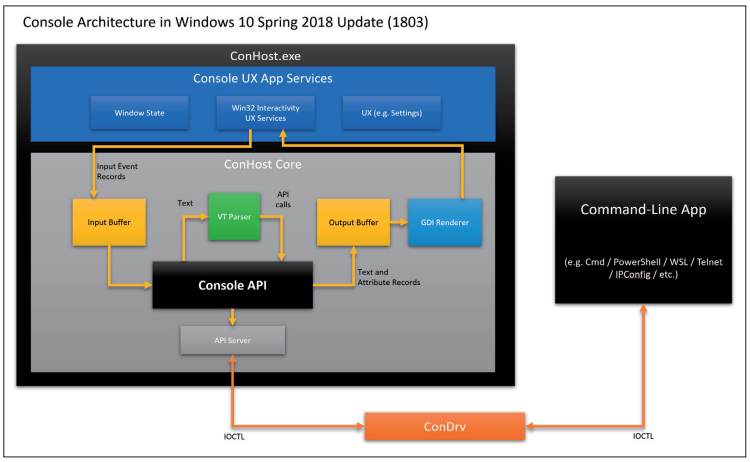


Bild: https://devblogs.microsoft.com

Bisherige Kommandozeilenapplikationen basierten auf einer zweiwertigen Architektur … (Bild 2)

Die Implementierung von ANSI/VT erwies sich als kleineres Problem – Microsoft hat sich entschieden, einen komplett neuartigen Kommandozeilen-Client zu realisieren. Dazu waren auch Änderungen an der Gesamtsystemarchitektur nötig, welche Bild 2 und Bild 3 zusammenfassen.

Für die Realisierung klassischer Kommandozeilen-Applikationen – CMD und PowerShell gehören hier übrigens dazu – kommt die in Bild 2 gezeigte Struktur zum Einsatz. Neben der im Standalone-Betrieb laufenden Kommandozeilenapplikation gibt es hier den Kerneltreiber *ConDrv*, der für die Kommunikation per *IOCTL* verantwortlich ist. Zudem gibt es die User-Mode-Applikation ConHost, die für das Bereitstellen der Benutzerinterface-Services verantwortlich ist.

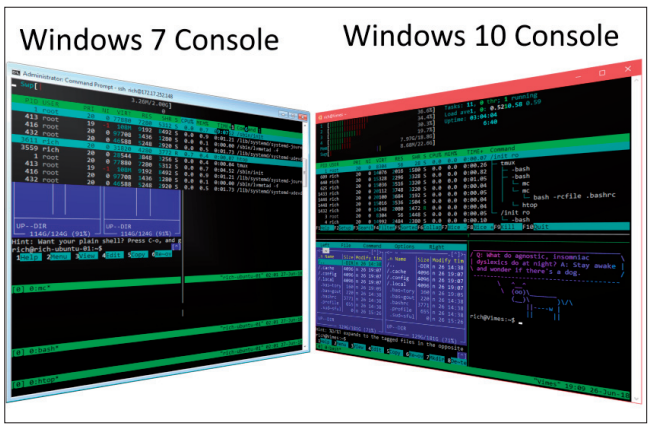


Bild: https://devblogs.microsoft.com

Die Rendering-Infrastruktur der „alten" Windows-Konsole ist mit vielen Linux-Applikationen überfordert (Bild 1)

Startet man beispielsweise PowerShell, so kümmert sich Windows im Hintergrund um die Bereitstellung einer Instanz von ConHost – dies lässt sich im Task-Manager unter *Details* durch das Auftauchen eines *ConHost*-Prozesses erkennen.

Dieser Unterschied ist von Bedeutung, wenn es an die Fernsteuerung von Kommandozeilenapplikationen geht. Die Kommunikation zwischen der eigentlichen Konsolenanwendung und ConHost erfolgt durch Messages, und nicht wie unter unixoiden Betriebssystemen durch den Austausch von Text-Datenströmen. Zudem erlaubt Microsoft nur die Verwendung von ConHost – andere Terminalemulatoren setzen, so weiß sogar das offizielle Microsoft-Blog [1], auf exotische Maßnahmen zur Umgehung dieses Problems: … *3rd party Consoles have to launch a Command-Line app off-screen at, for example, (-32000,-32000). They then have to send keystrokes to the off-screen Console, and screen-scrape the off-screen Console's text contents and re-draw them on their own UI!*

Die neue, in Bild 3 gezeigte Struktur führt ein zusätzliches Interface ein. Frei nach dem aus Unix bekannten Konzept des Pseudo-Terminals können die Konsolenanwendungen nun auch Textdatenströme eingeben und ausgeben. Microsoft realisiert dies durch eine Erweiterung des nach wie vor erforderlichen ConHost, der die Daten abgreift und über ein standardisiertes, von der Benutzerschnittstelle unabhängiges API zur Verfügung stellt. Windows Terminal ist dann ein Client dieser neuartigen Schnittstelle: Es ist in Theorie und Praxis vorstellbar und sogar explizit gewünscht, dass andere Entwickler ihre Programme an dieses neue Interface anhängen.

Das API ConPTY steht mittlerweile im Windows Insider Preview SDK bereit. In der unter [2] veröffentlichten Ankün-

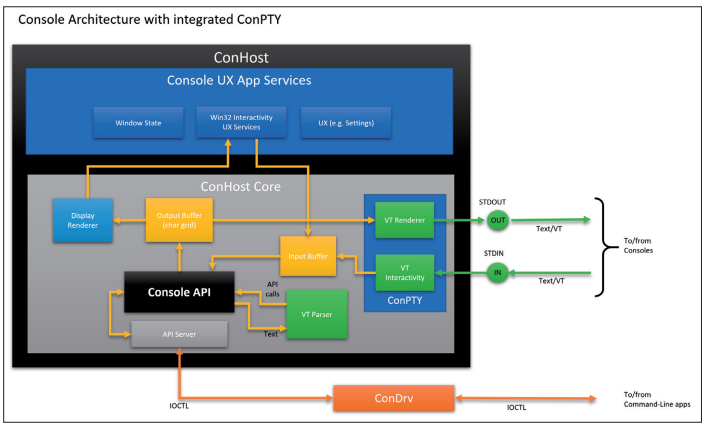


Bild: https://devblogs.microsoft.com

... während es dank der Pseudo-Console nun einen neuen Weg zur Kommunikation mit dem Terminal gibt (Bild 3)

digung des Features präsentiert Microsoft einen Überblick der Funktionen, für die derzeit allerdings noch kein .NET-Wrapper zur Verfügung steht.

Windows Terminal installieren

Microsoft stellt den Quellcode der Terminal-Anwendung unter [3] zur Verfügung. Das manuelle Kompilieren des Codes artet allerdings in Arbeit aus – zudem gibt es im Windows Store sowieso eine meist sehr aktuelle Binärversion, die sich mit geringem Aufwand installieren lässt. Aus der weiter oben besprochenen API-Erweiterung folgt, dass die Windows-Terminal-Anwendung nur mit Preview-Versionen von Windows 10 funktioniert. Der Autor nutzte in den folgenden Schritten den Insider-Build *Microsoft Windows [Version 10.0.19041.207]*.

Die Installation beginnt mit dem Aufruf des URL [4]. Im Windows Store autorisieren Sie den Download der mit 6 MByte nicht besonders großen Applikation. Nach Abschluss der Installation erfolgt der Start durch eine Suche nach *Windows Terminal*. Lohn der Mühen ist das in Bild 4 gezeigte Fenster, in dem sich einige Neuerungen verbergen.

Sofort fällt auf, dass das neue Terminal mehrere Tabs unterstützt. Das Plus-Symbol öffnet analog zu Firefox und Co. einen neuen Tab, der auf der Workstation des Autors von Haus aus eine PowerShell-Instanz belebt. Ein Klick auf das nach unten zeigende Pfeilsymbol öffnet ein Pop-up-Menü, das Ihnen die Auswahl verschiedener Konsolenarten ermöglicht. Auf der Workstation des Autors, die sowohl für Azure- als auch für WSL-Experimente eingerichtet ist, finden sich hier die Einträge *Ubuntu* und *Azure Cloud Shell*. Puristen finden auch eine Option, die einen klassischen CMD-Interpreter anwirft. Die Auswahl einer der Optionen sorgt dafür, dass ein neuer Tab stets mit der gewählten Konfiguration entsteht.

Der erste Test des Autors betraf die verschiedenen Versionen der WSL-Runtime. Ubuntu 18.04 ließ sich sowohl als WSL-1- als auch als WSL-2-Distribution starten. Interessanterweise erzeugen WSL-2-Tabs je eine Instanz der VM – ein in einem Tab losgetretener Ping-Run war im Neben-Tab für den Linux-Prozesslister *ps* nicht sichtbar.

Auf der Haben-Seite steht, dass Windows Terminal Mauseingaben für Linux-Applikationen unterstützt. Produkte wie *tmux* oder *mc* lassen sich nun ganz wie unter einem echten Unix per Mausklicks zum Navigieren durch Ordner und Verzeichnisstrukturen animieren.

Aufmerksame Beobachter bemerken eine weichere Textdarstellung im neuen Terminal. Dies ist keine optische Täuschung: Neben einem neuen Konsolen-Font setzt Microsoft in Windows Terminal auf Hardwarebeschleunigung, was die Verwendung fortgeschrittener Kantenglättungsverfahren ermöglicht.

Windows Terminal aus dem Kontextmenü

Der Wechsel zwischen grafischen und kommandozeilenbasierten Applikationen ist seit jeher aufwendig: Wer einige Male Schritt für Schritt durch das Dateisystem navigiert hat, um den Aufenthaltsort des Explorer-Fensters zu finden, schätzt in dieser Hinsicht jede Abhilfe.

Da Windows Terminal zum Zeitpunkt, als dieser Artikel entstand, nicht Teil des Betriebssystems war, bedarf es hierzu ein wenig Nachhilfe. Scott Hanselman bietet unter [5] eine *.inf*-Datei an, die sich wie jede andere Treiberdatei installieren lässt. Danach findet sich im Kontextmenü eine neue Option, die Windows Terminal im aktuellen Arbeitsverzeichnis des Windows Explorers aktiviert.

Konfiguration anpassen

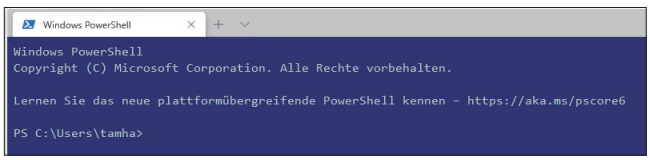
Microsoft erfreute die Entwicklerwelt in Windows 95 mit der Registry, einem interessanten Weg zum Speichern von Applikationseinstellungen. Im Lauf der Jahre nahm deren Umfang allerdings arg zu. Visual Studio Code diente als erste Testumgebung für eine neue Art der Konfigurationsverwaltung: Microsofts Mini-IDE lässt sich über eine JSON-Datei steuern.

In Windows Terminal findet sich eine ähnliche Funktion. Klicken Sie auf den nach unten zeigenden Pfeil, um sich danach für das Einstellungsmenü zu entscheiden. Windows Terminal reagiert darauf mit dem Start von Visual Studio 2017, die IDE lädt die Konfigurationsdatei automatisch. Die für Entwickler relevanteste Passage sieht so aus:

```
"defaultProfile": "{61c54bbd-c2c6-5271-96e7-
                   009a87ff44bf}",
```

Microsoft legt das Standard-Profil von Windows Terminal über eine GUID fest. Wer die ausreichend kommentierte Datei nach unten scrollt, findet die folgende Profil-Deklaration:

```
"profiles":
{
  "defaults":
  { // Put settings here that you want to apply
```



Windows Terminal meldet sich startklar (Bild 4)

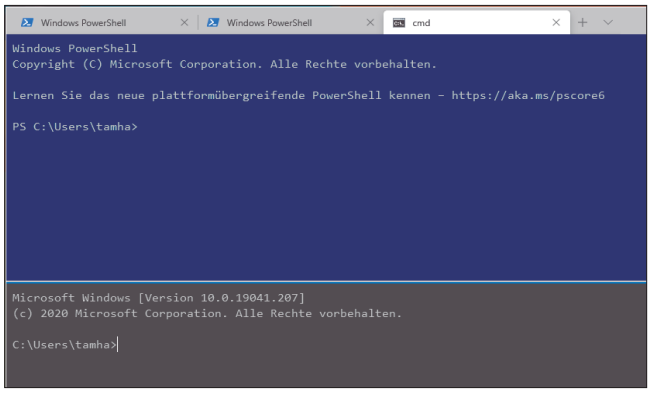
```
    // to all profiles
  },
  "list":
  [
   w {
      // Make changes here to the
      // powershell.exe profile
      "guid": "{61c54bbd-c2c6-5271-
                96e7-009a87ff44bf}",
      "name": "Windows PowerShell",
      "commandline": "powershell.exe",
      "hidden": false
    },
    {
      // Make changes here to the cmd.exe profile
      "guid": "{0caa0dad-35be-5f56-a8ff-afceeeaa6101}",
      "name": "cmd",
      "commandline": "cmd.exe",
      "hidden": false
    },
```

Besonders interessant ist hier die Verwendung des Parameters *commandline*. Er legt fest, welche EXE-Datei die Kommandozeilenapplikation aufruft.

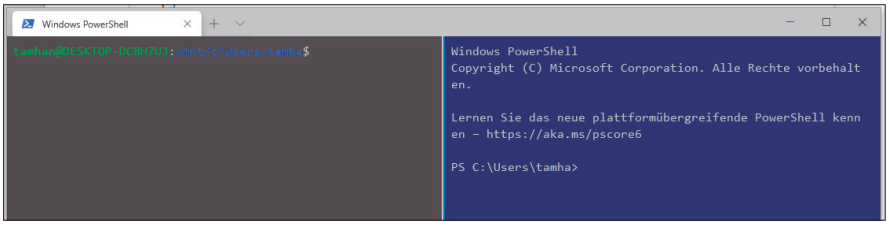
Dass PowerShell ein eigenes Profil hat, ist kein Selbstzweck. Haben Sie auf Ihrer Workstation mehrere PowerShell-Versionen und/oder eine Version von PowerShell Core, so bietet Windows Terminal im Menü alle PowerShell-Varianten als Zielsystem an. Für Gegenüberstellungen lassen sich so beispielsweise eine normale und eine Core-Version des Interpreters gleichzeitig laden.

Profile nehmen zudem plattformspezifische Attribute auf. Möchten Sie Ihre Ubuntu-Fenster beispielsweise immer im Verzeichnis eines bestimmten Users anwerfen, so hilft das *startingDirectory*-Attribut:

```
{
  "name": "Ubuntu-18.04",
  "commandline" : "wsl -d Ubuntu-18.04",
  "startingDirectory" : "//wsl$/Ubuntu-
   18.04/home/<Your Ubuntu Username>"
}
```



Die neue CMD-Pane ist einsatzbereit (Bild 6)



PowerShell und Linux Seite an Seite (Bild 5)

Fenster aufteilen

Gegenüberstellungen würden davon profitieren, wenn das Windows Terminal die einzelnen Bildschirmbereiche nebeneinander oder übereinander anzeigen könnte. Weiter oben in der Konfigurationsdatei gibt es einen *profiles*-Bereich, der so wirkt, als könne er dies leisten.

Als erste Aufgabe sollen mehrere Fenster nebeneinander angezeigt werden. Per Drag-and-drop funktioniert das in der im Windows Store erhältlichen Version noch nicht. Deshalb geht es erst einmal zurück zur Kommandozeile außerhalb von Windows Terminal. Dort wird folgender Befehl eingeben:

```
C:\Users\tamha>wt -p "Windows PowerShell" -d . ;
               split-pane
```

Das Kommando *wt* darf ob des Semikolons nur in eine CMD-Umgebung wandern: Wer es in die PowerShell-Kommandozeile eingibt, erhält nicht das erwartete Resultat. Die Übergabe von *split-pane* sorgt jedenfalls dafür, dass wir zwei Fenster nebeneinander bekommen. Laut der Dokumentation [6] kann eine schon laufende Windows-Terminal-Instanz weitere Aufrufe von *wt* abfangen, was das Erzeugen zusätzlicher Panes ermöglicht. Leider ist diese Vorgehensweise wenig bequem – schöner wäre es, wenn man diese Operation zur Laufzeit über ein Kommando direkt in Windows Terminal durchführen könnte.

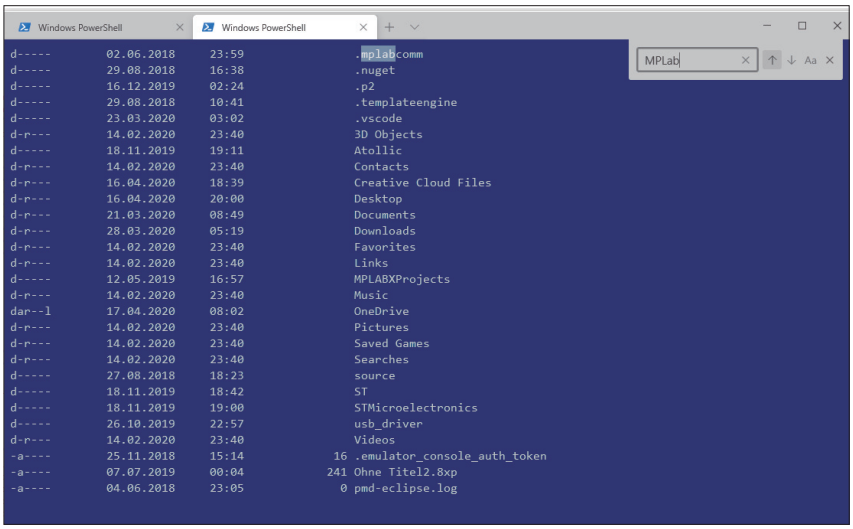
Zum Zeitpunkt, als dieser Artikel entstand, gab es nur vergleichsweise wenig Dokumentation zum Programm. Aus diesem Grund wird hier auf einen kleinen Trick zurückgegriffen: Jedes Windows Terminal bringt ein vollwertiges JSON-Abbild mit, das es im Rahmen des Programmstarts lädt. Die in der JSON-Konfigurationsdatei zu findenden Eigenschaften überschreiben dann Teile der Basiskonfiguration.

Angenehmer Nebeneffekt dieser Situation ist, dass das Basis-JSON-Dokument auch als Gesamtüberblick aller in Windows Terminal enthaltenen Funktionen herangezogen werden kann. Windows Terminal übergibt diesen Inhalt gerne an Visual Studio, wenn Sie beim Anklicken des weiter oben besprochenen *Settings*-Eintrags die [Alt]-Taste auf der Tastatur gedrückt halten. Der hier relevante Teil sind die *keybindings*, die folgendem Schema folgen:

```
"keybindings":
[
  { "command": "closePane", "keys": "ctrl+shift+w" },
  ...
  { "command": { "action": "splitPane",
```

```
    "split": "horizontal"}, "keys":
    "alt+shift+-" },
  { "command": { "action": "splitPane",
    "split": "vertical"}, "keys":
    "alt+shift+plus" },
```

Kommandos bestehen prinzipiell aus einem Command-String, der die zu erledigenden Aufgaben beschreibt. In manchen Fällen werden, wie beispielsweise hier bei *splitPane*, noch zusätzliche Parameter übergeben. Sie präzisieren die zu erledigende Aufgabe. Für die eigentliche Aktivierung ist dann eine Tastenkombination verantwortlich, die über das *keys*-Feld anzuliefern ist. Beachten Sie bitte, dass die Strings auch in deutschen Windows-Versionen in Englisch einzugeben sind.



Suchen mit Windows Terminal (Bild 7)

Für einen ersten Test können Sie die Kommandos eingeben – Tabs lassen sich öffnen und schließen. Ärgerlich ist in diesem Zusammenhang allerdings, dass die neu erzeugten Panes prinzipiell die Standard-Ausführungsumgebung verwenden: Auf dem Rechner des Autors ist dies immer PowerShell. Möchten Sie ein PowerShell- und ein anderes Fenster haben, so erzeugen sie im ersten Schritt das andere Fenster, um PowerShell danach hinzuzufügen, siehe Bild 5. Die Trennlinie zwischen den Fenstern lässt sich nicht mit der Maus verschieben. Stattdessen spendiert Microsoft die folgenden Tastenkombinationen, die sich immer auf das gerade aktive Fenster beziehen:

```
{ "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" },
```

Wer – wie der Autor – gerne sowohl CMD als auch PowerShell verwendet, kann sich eine neue Tastenkombination anlegen. Bei korrekter Parametrisierung kümmert sich diese darum, dass der gerade aktive Tab eine neue CMD-Instanz aufnimmt. Hierzu müssen Sie im ersten Schritt zur von weiter oben bekannten benutzerspezifischen JSON-Datei zurückkehren, in die Sie ein neues Kommando einpflegen:

```
// Add any keybinding overrides to this array.
// To unbind a default keybinding, set the command
// to "unbound"
"keybindings": [
  {
    "command": {
      "action": "splitPane",
      "split": "horizontal",
      "profile": "{0caa0dad-35be-5f56-a8ff-...  },
    "keys": "alt+shift+c"
  }
]
```

Das neue Key-Binding ist im Prinzip eine Ableitung, die Visual Studio aber bequemer formatiert. Wirklich neu ist das *profile*-Feld, das die GUID des Profils aufnimmt, die im neuen Tab zu erscheinen hat. Der Autor verwendet hier die GUID des CMD-Profils auf seiner Workstation: Es wäre theoretisch möglich, dass Sie auf Ihrem Rechner eine andere GUID vorfinden. Mit dem *split*-Attribut wird dann noch festgelegt, dass der neue Tab unter dem schon vorhandenen erscheinen soll.

Das Zusammenführen der Einstellungen erfolgt im Windows Terminal feingranular: Das Array entsteht also aus der Zusammenfassung der allgemeinen und der vom Benutzer festgelegten Attribute. Das führt dazu, dass die von Microsoft vorgegebenen Tastenkombinationen nach wie vor funktionieren, sofern sie nicht mit einer vom Benutzer angelegten kollidieren. Nach dem Speichern können Sie jedenfalls – wie in Bild 6 gezeigt – eine neue CMD-Pane erzeugen.

Microsoft listet unter [7] die vollständige Syntax der Einstellungsdatei auf und zeigt dort alle weiteren Kommandos, die sich mit Tastenkombinationen verdrahten lassen.

Inhalte suchen

Kommandozeilenapplikationen neigen dazu, den Benutzer mit Informationen zu überfluten – wer auf einer aktiv arbeitenden Linux-Workstation beispielsweise *dmesg* aufruft, bekommt mehr als nur einen Bildschirm an Informationen. Des Linux-Entwicklers klassisches Werkzeug ist dabei die Pipe: Werkzeuge wie *Grep* reduzieren den Umfang der am Bildschirm angezeigten Informationen.

Moderne Browser unterstützen ihre User mit flexiblen Suchfunktionen, die in Windows Terminal ebenfalls implementiert sind. Von Haus aus ist das dafür vorgesehene Kommando mit folgender Tastenkombination verdrahtet:

```
{ "command": "find", "keys": "ctrl+shift+f" }
```

Nach dem Drücken von [Ctrl]+[Shift]+[F] erscheint das in Bild 7 rechts oben eingeblendete Suchfenster, in das Sie den zu suchenden String eingeben dürfen. Beachten Sie allerdings, dass der Suchprozess nicht automatisch nach der Eingabe des Strings beginnt – Sie müssen die Eingabetaste drücken, um die erste Suche zu starten. Danach können Sie mit den Pfeilsymbolen wie gewohnt durch den gesamten Scrolling-Buffer navigieren.

Das Design anpassen

WinAmp verdankte seine immense Popularität seinen Themes: Nutzer durften das Aussehen des Mediaplayers an ihre Bedürfnisse anpassen. Microsoft stellt im Terminal selbst zwar keinen Theme-Manager bereit, unter [8] finden Sie aber eine von einem Drittentwickler verwaltete Themenliste.

Die Platzierung von Themenattributen ist insofern komplex, als Windows Terminal mehrere Andockstellen bereitstellt. Zur Demonstration soll der *retroTerminalEffect* dienen, eine von Microsoft eher als Gag eingeführte Rendering-Funktion, welche die GPU-Beschleunigung zur Simulation eines altmodischen Röhrenmonitor-Terminals einspannt.

Am Einfachsten geht die Inbetriebnahme von der Hand, wenn Sie das Attribut anfangs in die Rubrik *defaults* schreiben. Dort platzierte Einstellungen propagieren sich automatisch in alle neu geöffneten Fenster, was zur in Bild 8 gezeigten Optik des Terminals führt:

```
"profiles":
{
  "defaults": {
  "experimental.retroTerminalEffect": true
  // Put settings here that you want to apply
  // to all profiles
},
```

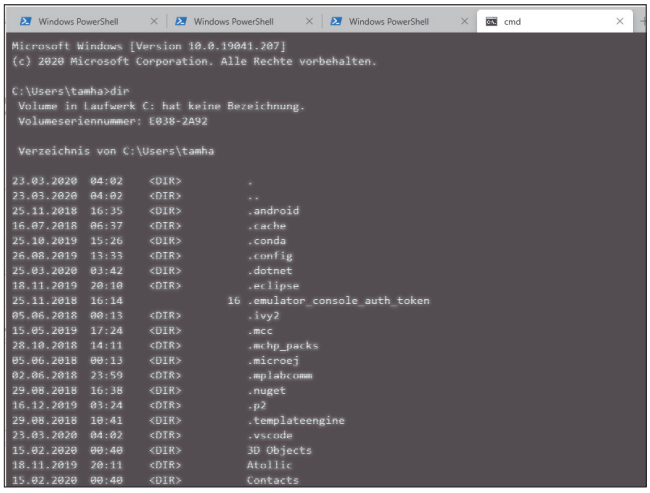
In der Praxis ist es empfehlenswert, auf das in Windows Terminal integrierte Theme-Verwaltungssystem zurückzugreifen. Microsoft hat den diesbezüglichen Speicherbereich in der Datei *profiles.json* vor einiger Zeit umbenannt – anstatt auf *profiles* hört der Bereich nun auf den Namen *schemes*:

```
// Add custom color schemes to this array
"schemes": [],
```

Das kann zu Problemen führen, da sich im Internet noch veraltete Vorlagen finden. Bedenken Sie die Umbenennung der Einstellungsvariable bei der manuellen Zusammenführung, um Probleme beim Start zu vermeiden.

Der Autor demonstriert die Vorgehensweise zur Theme-Integration in den folgenden Schritten anhand der Vorlage „Monokai Night for Windows Terminal“. Nach Anklicken des Code-Buttons erhalten Sie ein nach dem folgenden Schema aufgebautes JSON-Objekt:

```
{
  "name" : "Monokai Night",
  "background" : "#1f1f1f",
```



Die Retro-Effekte zeigen: Nicht alles war früher besser (Bild 8)

```
  "foreground" : "#f8f8f8",
  "black" : "#1f1f1f",
  ...
```

Themes beziehungsweise Profile sind in Windows Terminal eine Ansammlung von Eigenschaften, die im System vorhandene Voreinstellungen mit neuen Werten überschreiben. Zudem ist ein *name*-Attribut erforderlich, welches das Profil im System eindeutig benennt. Die erste Aufgabe besteht jedenfalls darin, den von der Webseite bereitgestellten Code in die *schemes*-Einstellung zu integrieren:

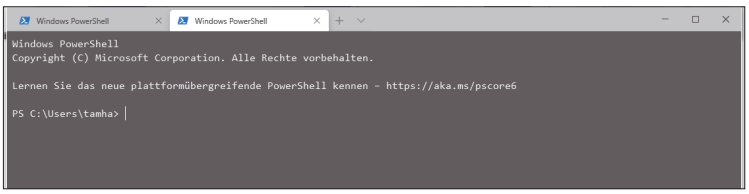
```
// Add custom color schemes to this array
"schemes": [
  {
    "name": "Monokai Night",
    "background": "#1f1f1f",
    "foreground": "#f8f8f8",
    ...
    "brightYellow": "#e6db74"
  }
],
```

Leider führt das Speichern der Änderungen noch nicht zum Erfolg. Windows Terminal verwendet von Haus aus stur das von der jeweiligen Kommandozeilenanwendung vorgegebene Farbschema.

Der weiter oben vergebene Name erlaubt die Anpassung. In Fällen, in denen Sie nur das Aussehen der PowerShell beeinflussen wollen, platzieren Sie das Attribut *colorScheme* im jeweiligen Profil:

```
"list": [
  ...
  { ...
    "commandline": "powershell.exe",
    "hidden": false,
    "colorScheme": "Monokai Night"
  },
```

Öffnen Sie nach diesen Änderungen ein PowerShell-Fenster, dann sehen Sie eine Veränderung der Hintergrundfarbe, vergleiche Bild 9. Die genaue Betrachtung der Ausgabe informiert auch darüber, wie Windows Terminal die Color-Attribute anwendet.



Der Hintergrund des PowerShell-Fensters ist nun grau (Bild 9)

Microsoft nimmt im ersten Schritt die von der jeweiligen Kommandozeilen-Applikation angelieferten Eingaben samt den in ihnen enthaltenen Farbcode-Informationen entgegen. Im nächsten Schritt erfolgt eine stufenweise Substitution, welche die vorgegebenen Farben durch die im Profil festgelegten Farbcodes ersetzt. Optionale Hintergrundbilder und ähnliche Attribute werden erst danach angewendet. Alternativ dazu dürfen Sie selbstverständlich auch auf das Feld *defaults* zurückgreifen.

Visual Studio zu Diensten

Microsoft bietet Visual-Studio-Nutzern seit einiger Zeit dedizierte Kommandozeilen an, deren Pfadvariablen verschiedene häufig benötigte Kommandozeilen-Werkzeuge direkt ansprechbar machen. Der Visual Studio Command Prompt ist dabei wohl der am häufigsten genutzte Vertreter des Genres.

Aus dem Autor unerfindlichen Gründen integriert das Windows Terminal von Haus aus kein zu dieser Umgebung passendes Profil. Für besonders häufig verwendete Systeme gibt es unter [9] vorgefertigte Profilvorlagen. Als kleines Beispiel soll der „Developer Command Prompt for Visual Studio“ dienen. Dazu navigieren Sie zunächst zur Seite www.guidgenerator.com, wo Sie eine neue GUID beantragen.

Die von Microsoft vorgegebenen Profile müssen von Hand eingegeben werden. Auf dem Rechner des Autors sieht das Resultat für Visual Studio 2019 wie folgt aus:

```
"list": [
  {
    "guid": "{5c9bb935-d558-4f0f-91d3-
              ef9dfa9c7bc6}",
    "name": "Developer Command Prompt for VS 2019",
    "commandline": "cmd.exe /k \"C:/Program Files
     (x86)/Microsoft Visual Studio/2019/
     Professional/Common7/Tools/
     VsDevCmd.bat\"",
    "startingDirectory": "%USERPROFILE%"
  },
```

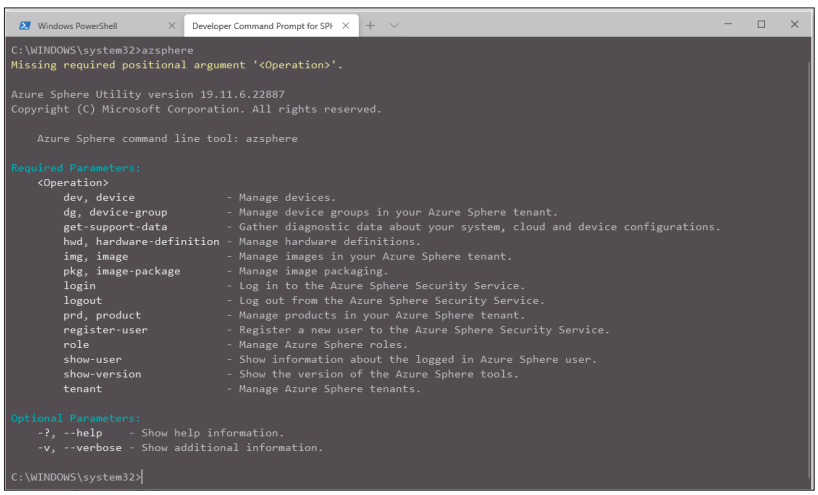
Zur Inbetriebnahme des Terminals müssen Sie dieses Snippet in die Einstellungs-JSON-Datei übertragen und diese speichern. Nach getaner Arbeit findet sich im Nach-unten-Menü ein neuer Eintrag, der die Visual-Studio-Eingabeaufforderung zur Verfügung stellt. Auf der Workstation des Autors öffnet seine Aktivierung zwar ein neues Fenster, wirft in diesem aber einen Fehler: *Der Befehl „C:/Program ...“ ist entweder falsch geschrieben oder konnte nicht gefunden werden*. Ursache des Problems ist, dass Vorschau- und Testversionen von Visual Studio 2019 in einem jeweils anderen Pfad liegen. Eine funktionsfähige Variante der Deklaration sieht wie folgt aus:

```
"list": [
  {
    ...
    "commandline": "cmd.exe /k \"C:/Program Files
     (x86)/Microsoft Visual Studio/2019/Preview/
     Common7/Tools/VsDevCmd.bat\"",
    "startingDirectory": "%USERPROFILE%"
  },
```

Wer in der Vergangenheit viel mit Azure gearbeitet hat, ärgert sich darüber, dass es auf der Webseite kein vorgefertigtes Profil für die „Azure Sphere Developer Command Prompt Preview“ gibt. Erfreulicherweise lässt sich dieses Problem einfach beheben. Im ersten Schritt klicken Sie die Verknüpfung im Startmenü rechts an, um den Ablageort im Windows Explorer zu öffnen. Die Datei *Azure Sphere Developer Command Prompt Preview* ist eine Verknüpfung, die nach einem weiteren Rechtsklick ihre Inhalte preisgibt.

Der enthaltene String *C:\Windows\System32\cmd.exe /k "C:\Program Files (x86)\Microsoft Azure Sphere SDK\\InitializeCommandPrompt.cmd"* erinnert stark an die weiter oben verwendete Deklaration.

Mit einer neuen GUID lässt sich der Developer-Prompt in die Einstellungsdatei einpflegen – als primäres Hindernis erweist sich Visual Studio 2017, das beim Einfügen eines Pfades mit Rückwärtsschrägstrichen (Backslash) unsinnige Anpassungsprozesse lostritt: ▶



Azure Sphere ist einsatzbereit (Bild 10)

```
"list": [
  {
    "guid": "{5c9bb935-d558-4f0f-91d3-
              ef9dfa9c7bc6}",
    "name": "Developer Command Prompt for SPHERE",
    "commandline": "cmd.exe /k \"C:/Program Files
    (x86)/Microsoft Azure Sphere SDK//
    InitializeCommandPrompt.cmd\""
  },
```

Nach dem Speichern steht auch das Azure-Sphere-Terminal – wie in Bild 10 gezeigt – im Windows Terminal zu Ihrer Disposition. Auf Wunsch können Sie das Attribut *startingDirectory* von weiter oben übernehmen, um im von Microsoft vorgegebenen Arbeitsverzeichnis zu starten.

Windows Terminal und die Zwischenablage

Visual Studio Code ist so lange lustig, bis man die angezeigten Codes in Word oder OpenOffice übernehmen möchte. Das Programm legt von Haus aus HTML-Markup in die Zwischenablage, was beim Einfügen zu allerlei Nervereien führt. Die gedankliche Verbundenheit zwischen Windows Terminal und Visual Studio Code zeigt sich daran, wie Windows Terminal mit Zwischenablagen-Transaktionen umgeht.

Am Wichtigsten ist die Feststellung, dass die aus Windows bekannten Tastenkombinationen in Windows Terminal nicht direkt funktionieren. Das liegt daran, dass beispielsweise [Strg]+[C] in der Linux-Welt als Beenden-Kommando beziehungsweise spezielles Steuerzeichen fungiert. Ein kurzer Blick in die Default-Konfiguration informiert über das Vorhandensein folgender alternativer Tastenkombinationen:

```
{ "command": "copy", "keys": "ctrl+shift+c" },
{ "command": "copy", "keys": "ctrl+insert" },
...
{ "command": "paste", "keys": "ctrl+shift+v" },
{ "command": "paste", "keys": "shift+insert" },
```

Interessanterweise scheint das Deklarieren eines zweiten Alias den ersten zu eliminieren. Auf der Workstation des Autors funktionierte nur [Ctrl]+[Insert] zum Kopieren eines zuvor mit der Maus gewählten Ausgabebereichs. Wer sich das Hantieren mit der Tastatur ersparen möchte, kann – analog zu Linux – auch das sofortige Aktualisieren der Zwischenablage beim Markieren von Text in der Ausgabe anweisen. Hierzu müssen Sie das Attribut *copyOnSelect* platzieren, das allerdings außerhalb des *profiles*-Arrays sitzen muss:

```
"defaultProfile": "{61c54bbd-c2c6-5271-96e7-
                   009a87ff44bf}",
"copyOnSelect": true,
"profiles": {
...
```

Problematisch ist in diesem Zusammenhang, dass Windows Terminal von Haus aus stets vollfarbige Ausgaben generiert. In der Theorie lässt sich dies durch das Attribut *copyFormatting* unterdrücken, welches die Zwischenablage mit formatfreiem Text füttert:

```
"defaultProfile": "{61c54bbd-c2c6-5271-96e7-
                   009a87ff44bf}",
"copyOnSelect": true,
"copyFormatting": false,
"profiles": {
  ...
```

Leider war *copyFormatting* zum Zeitpunkt der Drucklegung im Allgemeinen problematisch – unter [10] finden Sie eine Diskussion, die auf die eine oder andere Spitzfindigkeit eingeht. Um den Ärger zu umgehen, bietet es sich an, den String in einem einfachen Editor-Fenster abzulegen und von dort aus erneut zu kopieren.

Fazit

Wer viel Zeit mit der Arbeit auf der Konsole verbringt, erhält mit Windows Terminal eine ganze Gruppe von Funktionen, die der mit Linux arbeitende Kollege schon seit Längerem nutzen darf. Angesichts der nicht unerheblichen Produktivitätssteigerungen zahlt es sich auf jeden Fall aus, dafür einen kurzen Abstecher in den Windows Store zu unternehmen. Dass die Dokumentation im Moment noch „ruppig“ ist, vergisst man dank der Komfortfunktionen schnell. ■

[1] Inside the Windows Console,
www.dotnetpro.de/SL2008WindowsTerminal1
[2] Introducing the Windows Pseudo Console (ConPTY),
www.dotnetpro.de/SL2008WindowsTerminal2
[3] Windows Terminal auf GitHub,
https://github.com/microsoft/terminal
[4] Windows Terminal im Windows Store,
www.dotnetpro.de/SL2008WindowsTerminal3
[5] Scott Hanselman, WindowsTerminalHere,
www.dotnetpro.de/SL2008WindowsTerminal4
[6] Using the wt.exe Commandline,
www.dotnetpro.de/SL2008WindowsTerminal5
[7] Settings.json Documentation,
www.dotnetpro.de/SL2008WindowsTerminal6
[8] TerminalSplash, https://terminalsplash.com
[9] Adding profiles for third-party tools,
www.dotnetpro.de/SL2008WindowsTerminal7
[10] Ärger mit copyFormatting,
www.dotnetpro.de/SL2008WindowsTerminal8

Tam Hanna
entwickelt Programme für verschiedene Plattformen, betreibt Online-Newsdienste zum Thema und steht für Fragen, Trainings und Vorträge gern zur Verfügung. Sie erreichen ihn unter der E-Mail-Adresse **tamhan@tamoggemon.com**.

dnpCode	A2008WindowsTerminal