Dokumentation

# Präambel

Bei **NEat** handelt es sich um ein Tool, mit dem auf einem Windowsserver per Web-Frontend vorgefertigte Skripte ausgeführt werden können.

Das Hauptaugenmerk liegt auf der Administration von NetApp Servern, jedoch sind auch von NetApp unabhängige Skripte erstellbar.

Als Sprache für die Skripte wird PowerShell eingesetzt, weitere Sprachen wie z.B. Perl, Python und R sind in Arbeit.

Entwicklung und Debugging ist explizit vom System gelöst und kann mit der bevorzugten Toolchain durchgeführt werden.

# Installation

## Systemanforderungen

### Frontend

Server mit Node.js und React. Keine besonderen Anforderungen, kann alternativ auf anderen, schon vorhandenen, Webservern laufen.

### Backend

Windowsserver mit PowerShell 5.x und installierten Zusatzpaketen für die angedachten Einsatzgebiete. Als Beispiel ‚ NetApp PowerShell Toolkit‘ für Arbeiten mit NetApp Servern.

### Entwicklungsumgebung

Die Entwicklung kann auf dem Backend Server stattfinden, es wird jedoch empfohlen, dass jeder Entwickler auf seiner lokalen Workstation arbeitet. PowerShell 5.x und installierte Zusatzpakete für die angedachten Einsatzgebiete sind vorausgesetzt.

## Installation

### Frontend

Bitte installieren sie node.js (6.x)

Die node.js Kommandozeile (nicht node.js!) öffnen und npm install http-server –g eingeben um den Webserver zu installieren.

Die Frontend Dateien in den gewünschten Ordner kopieren.

In diesem Ordner http-server -–proxy [http://[backend host]:8000](http://localhost:8000) als Dienst starten.

Weitere Infos/Optionen für den Server findet man unter <https://www.npmjs.com/package/http-server>.

### Backend

Kopieren sie die Backend Dateien in den gewünschten Ordner.

Starten sie die Datei .\BackendListener\BackendListener.ps1 als Dienst und geben sie ihr Administrationsrechte.

Das Frontend muss Port 8080 freigeschaltet bekommen und kommuniziert über Port 8000 mit dem Backend, das Backend kommuniziert über die von WFA (bzw. anderen Servern) vorgegebenen Ports. Diese sind in der Firewall freizuschalten, so sich nicht alles auf dem gleichen System befindet.

Zur Ausführung von Powershell Skripten, die auf Netapp Filer zugreifen muss ‚NetApp PowerShell Toolkit‘ (bzw ‚ DataONTAP PowerShell Toolkit‘) installiert sein.

### Entwicklungsrechner

Die Entwicklung kann auf dem Backend Server stattfinden, es wird jedoch empfohlen, dass sich jeder Entwickler über eine Versionsverwaltung (z.B. git), die entsprechenden Dateien auf seinen Rechner zieht.

Es handelt sich dabei im Moment um die Ordner

* .\global
* .\BackendWorker\commands
* .\BackendWorker\helper
* .\BackendWorker\intern
* .\BackendWorker\json
* .\BackendWorker\logs

Es gibt keine Festlegung auf eine Entwicklungsumgebung oder Editor. Die vom Entwickler bevorzugte Toolchain kann verwendet werden. Testläufe/Debugging ist direkt mit Kopien des Produktionscodes möglich.

### Konfiguration

* Filerliste: in .\global\controller.json
* Allgemeine Einstellungen in .\global\global.json
* Frontend: URL zum Backend -> wird beim Dienstaufruf eingetragen
* [XXX?]

# Neat benutzen

## Weboberfläche

Die Entwicklungsserver sind mit <http://127.0.0.1:8080> aufrufbar.

Die Produktivserver sind mit <http://127.0.0.1:8080> aufrufbar. [xxx muss später angepasst werden]

## Manuelles starten von JSON Dateien

Im Hauptordner des Backends befindet sich das Skript ‚ ID-Starter.ps1‘, dieses holt eine Liste aller json Dateien aus dem ‚in‘ Folder und führt dann die entsprechenden Befehle aus.

Es ist möglich json Dateien zu laden, die nicht durch NEat erstellt wurden.

[xxx ACHTUNG momentan sind in dem Skript keinerlei Sicherheitsabfragen/Schutz vor Bedienfehlern vorhanden]

## BackendListener stoppen

Der BackendListener kann über <http://localhost:8000/end/> gestoppt werden und muss dann entweder manuell oder über die Dienstverwaltung von Windows neu gestartet werden. Der node.js http-server muss eventuell danach auch neu gestartet werden. (beendet sich wenn der BackendListener nicht existiert) [xxx http-server sollte sich auch Produktiv mit neuem Server ändern]

# *NEat* erweitern

## Neues Command erstellen

Ein Command besteht grundlegend aus:

* HTML Seite
* Javascript Code
* Backendfunktion
* In die Startseite einbinden

Manche Commands werden von mehreren HTML-Seiten aus aufgerufen

### Frontend

Neue Frontendseite:

* Passende Vorlage kopieren (siehe .\js\\_Vorlage\_Kommentiert.js)
* Anpassen
* Javascript-Code erstellen

Im oberen Bereich der JavaScript Dateien werden die Ractive Funktionen definiert und ausgeführt im unteren Bereich die Backend Aufrufe.

### Backend

Die Funktionen sind im Moment alle in PowerShell zu implementieren, andere Sprachen sind geplant.

Im Ordner .\BackendWorker\commands kopiert man den Ordner \_Vorlage und benennt ihn um. Es wird empfohlen sich der Übersichtlichkeit halber an die Namensregeln von Microsoft zu halten (‚Approved Verbs for Windows PowerShell Commands‘ <https://goo.gl/Ch3M6b>). In diesem Ordner sind alle notwendigen Dateien, sowie Beispielcode für ein neues Command enthalten.

Der eigentliche Code startet immer mit

param (

[string]$id = "148526049858982"

)

Die ID ist zur Übergabe, welches JSON aus dem .\BackendWorker\json\in die Parameter zur Programmausführung beinhaltet. Durch Eingabe einer ID die schon verwendet wird, kann das Skript direkt gestartet werden. Im Betrieb sollte die feste Definition auskommentiert werden.

Als Vorlage für die json Datei die im Frontend befüllt wird dient .\BackendWorker\commands\\_Vorlage\\_Vorlage.json. Diese wird vom Backendentwickler angepasst und vom Frontendentwickler in die .js Dateien eingebaut.

### Synchron / Asynchron

Der Entwickler des Commandos gibt in der zugehörigen config.json immer an, ob ein Command synchron oder asynchron ausgeführt werden soll. Asynchrone Commands sollten für länger laufende Aktionen verwendet werden, bei denen die Gefahr eines Timeouts vom Frontend (und der Nutzer) höher ist. Synchrone und asynchrone Commands werden parallelisiert ausgeführt und haben keine Bevorzugung.

Synchron: handler-Funktion im .js um das Ergebnis anzuzeigen  
Asynchron: handleCommand-Funktion im .js um das Ergebnis anzuzeigen

Beispiele findet man unter: .\js\demo-list-async.js und .\js\demo-list-sync.js

### Debugging

Zum Debuggen der Backendfunktionen braucht man nur PowerShell oder eine Entwicklungsumgebung, die PowerShell unterstützt und ein json mit den notwendigen Parametern. Man führt das Skript in dem Ordner aus, in dem es liegt, als Übergabewert nimmt man den Name des json (ein UNIX-Timestamp) oder trägt ihn in dem Skript in der Zeile [string]$id ein. Danach funktioniert alles wie man es von der Entwicklungsumgebung her gewohnt ist.

Alternativ kann das Skript ‚ ID-Starter.ps1‘ verwendet werden, welches oben schon erwähnt ist.

## Neue Backend-Funktionen für mehrere Commands

Wenn man PowerShell Funktionen hat die in mehreren Commands verwendet werden, legt man diese in das helper-Verzeichnis.

Es gibt zwei Ordner mit geteilten Funktionen, ‚helper‘ und ‚intern‘. In ‚helper‘ sind alle selbstgeschriebenen Funktionen, die in mehreren Commands verwendet werden können. Zum Beispiel wenn eine Funktion, um sich aus dem Firmennetzwerk tunneln. In ‚intern‘ stehen die geteilten Funktionen, die von der Firma V3Consulting zur Verfügung gestellt werden, in diesem Ordner sollte nichts bzw. nur nach Absprache verändert werden, da Änderungen beim nächsten Update von **NEat** überschrieben werden. Die mitgelieferten Funktionen eine Variation des ‚verb-noun‘ Schema mit der Ergänzung ‚V3‘, also ‚verb-V3noun‘.

Funktionen können je nach Bedürfnis per Dot-Sourcing oder mit Import-Module eingebunden werden.

### Debugging

Das Debugging funktioniert ähnlich wie bei den Commands, nur dass die Funktion in ein bestehendes Command eingebunden wird.

## Codebeispiele

Codebeispiele sind im html-Hauptordner sowie im .\js Unterordner für die Weboberfläche zu finden. Für die Skripte sind Codebeispiele unter .\BackendWorker\commands\\_Vorlage zu finden.

Wie neue Backend Funktionen für Commands erstellt und eingebunden werden, kann man z.B. unter dem Command .\BackendWorker\commands\get-ControllerList und der zugehörigen Funktion .\BackendWorker\intern\connect-V3controller sehen.

# Troubleshooting

Sollte ein Kommando gar nicht ausgeführt werden und der Browser bleibt einfach leer, dann ist die Wahrscheinlichkeit hoch, dass es sich um einen Javascript-Fehler handelt und man sollte in der Entwickler-Konsole des entsprechenden Browsers nachschauen, ob es dort eine Fehlermeldung gibt. Bei Chrome kann man diese mit F12 aktivieren.