Gmail Andy Schulz <andy.schulz@gmail.com>

Bash

Schulz, Andreas <andreas.schulz@accenture.com> 31. August 2018 um 14:56
An: "andy.schulz@gmail.com" <andy.schulz@gmail.com>

```
checkFormat()
{
  local TARGET_FORMAT="$(echo "$1" | xargs | tr "[:lower:]" "[:upper:]")";
  local CURRENT_FORMAT="$(echo "$2" | xargs | tr "[:lower:]" "[:upper:]")";
--

check_diskspace()
{
  local TARGET_PATH="$1";
  local MINIMUM_DISKSPACE="$2";
  local FREE_DISKSPACE="";
  do_log "Checking free diskspace on '${TARGET_PATH}' ...";
  check_if_set "MINIMUM_DISKSPACE" "${DEFAULT_MINIMUM_DISKSPACE}";
  FREE_DISKSPACE="$(df -P ${TARGET_PATH} | tail -1 | awk '{print $4}')";
--

check_server()
{
  local SERVER="$1";
  local PORT=$2;
--

  do_log "Checking server: ${SERVER} port: ${PORT}";
  TARGET_SERVER="${TEMP}";  # Just for formatting reasons
  IP_ADDRESS="$(nslookup ${SERVER} | tail -4 | grep Address | awk '{print $2}')";
--
}

waitForListener(){
  local SERVER="${1}"
  local PORT="${2}"
--
}

waitForServerUntilTimeout(){
  local SERVER=${1}
  local PORT=${2}
  local TIMEOUT_SECONDS=${3}
--

removeStringIfContained()
{
  local VARIABLE_NAME="$1";
  local TO_BE_REMOVED="$2";
--

create_group()
```

```
{
  local GROUP="$1";
  local GROUPID="$2";
--

create_user()
{
  local USER="$1";
  local USERID="$2";
  local HOMEDIR="$3";
  local GROUP="$4";
--

remove_user()
{
  local USER="$1";
  local USERID="$2";
--

remove_group()
{
  local GROUP="$1";
--

copy_function()
{
  local SOURCE="$1";
  local TARGET="$2";
--

remove_content_from_file()
{
  local TARGET_FILE="$1";
  local INSERT_TAG_BEGIN="$2";
  local INSERT_TAG_END="$3";
--

searchAndReplaceInFile()
{
  local TOBE_REPLACED="$1";
  local REPLACEMENT="$2";
  local FILE="$3";
  local INSERT_TAG_BEGIN="$4";
  local INSERT_TAG_END="$5";
--

appendFileContentToFile()
{
  local SOURCE_FILE="$1";
  local DESTINATION_FILE="$2";
  local INSERT_TAG_BEGIN="$3";
  local INSERT_TAG_END="$4";
--

appendScriptOuputToFile()
{
  local SOURCE_FILE="$1";
  local DESTINATION_FILE="$2";
  local INSERT_TAG_BEGIN="$3";
  local INSERT_TAG_END="$4";
--
#INSERT_TAG_END="###${PACKAGE_NAME}_INSERTION - END";
#append_to_file "${SOURCE_FILE}" "${DESTINATION_FILE}" "${INSERT_TAG_BEGIN}"
"${INSERT_TAG_END}";
```

```
  local TEXT="$1";
  local DESTINATION_FILE="$2";
  local INSERT_TAG_BEGIN="$3";
  local INSERT_TAG_END="$4";
--

recreate_folder()
{
  local TARGET="${1%\/}";
--

recreate_link()
{
  local LINKFILE="${1%\/}";
  local TARGET="${2%\/}";
--

make_splunk_owner()
{
  local DIRECTORY="$1";
--

check_if_empty()
{
  local DIRECTORY="$1";
--

ask()
{
  local QUESTION="$1";
  local DEFAULT_ANSWER="$2";
  local QUESTION_TIMEOUT="$3";
--

askForInfo()
{
  local QUESTION="$1";
  local DEFAULT_ANSWER="$2";
  local QUESTION_TIMEOUT="$3";
--

askForPassword()
{
  local QUESTION="$1";
--
- If the check is positive the return code has to be 0, if not the return code has to be something else.
- It is recommended to suppress the output of this function.";

  local QUESTION="$1";
  local CHECK_FUNCTION="$2";
  local DEFAULT_ANSWER="$3";
  local QUESTION_TIMEOUT="$4";
```

- init()
  - This is an example for an initialization function that can be called before calling anything else
- setParameters()
  - INPUT

        - $* has to be handed over to the function

    - This function reads input parameters (like $1, $2, …) and creates corresponding variables. For example, setParameters hey ho --> leads to variables hey=true and ho=true

    - This comes in very handy when using scripts with Jenkins and in the shell

        - For example when there is a checkbox for "hey" and it is checked, Jenkins will create a variable hey=true

- checkIfUpgrade()

    - INPUT

        - SCRIPT_NAME has to be set

    - A very handy function for installation scripts in RPMs. It checks whether the RPM is currently upgraded. When it is upgraded it sets UPGRADE=yes which can be used in the rest of the script. When the current script is an uninstallation script checkIfUpgrade will exit the script to avoid unwanted uninstallation effects during the upgrade (see Reihenfolge der Skripte beim Ausführen eines rpm -U). To determine whether it is an upgrade the function needs the parameters that the script is called with (see Übergabewerte an die Skripte eines RPM-Pakets)

- cleanup()

    - INPUT (optional)

        - local ERRORCODE="${1:-0}";

    - This function is supposed to clean up after the script terminated. Thus it is called in the function exit_script. The function exit_script however is called either at the end of the script or by the function fail which indicates an error (see below). This makes sure that it is always clean after script execution independent of the success of the script.

    - For special cleanups it would be obvious to override this function. It still would be called the exit_script function

- exit_script()

    - INPUT

        - local ERRORCODE="$1";

    - This function is called either at the end of the script or by the function fail which indicates an error (see below).

    - It prints a short summary if available, shows all warnings that have occurred and calls the cleanup function

- add()

    - INPUT

        - local VARIABLE_NAME="$1";

        - local SUMMAND="$2";

- subtract()

    - INPUT

        - local MINUEND="$1";

        - local SUBTRAHEND="$2";

    - Those two functions are called with a variable name (not the variable with $) and a number. They add/subtract the number to the value stored in the variable. So the result it stored in the variable

again.

- These functions are created just because the usability of the math functions in bash is not very good
- If the variable does not contain a number it is set to zero

- fail()
  - INPUT
    - local RETURNCODE="$1";
    - local LINENUMBER="$2";
    - local ERRORMESSAGE="$3";
    - local ERRORCODE="$4"; (OPTIONAL)
    - local FUNCTION_OUTPUT="$5"; (OPTIONAL)
  - This function is called with those parameters $? $LINENO <Error message> <Individual error code> <Additional error information (optional)>
  - The function prints the corresponding information (Error codes/messages and the linenumber) and exits the script
- check_fail()
  - INPUT
    - local RETURNCODE="$1";
    - local LINENUMBER="$2";
    - local ERRORMESSAGE="$3";
    - local ERRORCODE="$4"; (OPTIONAL)
    - local FUNCTION_OUTPUT="$5"; (OPTIONAL)
  - This function Is called like the function fail with the parameters $? $LINENO <Error message> <Individual error code> <Additional error information (optional)>
  - This function checks the error code in $?. If it is not zero the function fail is called and the scripts exits (see above)
- do_log()
  - INPUT
    - local MESSAGE="$1";
    - add INDENTATION "$2";
  - This function prints the string given to it in a formatted way (with timestamp and server information).
  - It supports an additional number. The output text will be printed intended according to the number
    - Example: INFO 12:12:29 [REF-ODSTA-A9001]: |-- Cleaning up
    - "--" is the indentation
- warn()
  - INPUT

        - local RETURNCODE="$1";
        - local LINENUMBER="$2";
        - local ERRORMESSAGE="$3";
    - This function is called similar to the functions fail and check_fail with the parameters: $? $LINENO <Error message>
    - It prints the corresponding information (Error codes/messages and the linenumber), but does not exit (unlike the function fail)
- check_warn()
    - INPUT
        - local RETURNCODE="$1";
        - local LINENUMBER="$2";
        - local ERRORMESSAGE="$3";
    - This function is called like the function warn with the parameters: $? $LINENO <Error message>
    - This function checks the error code in $?. If it is not zero the function warn is called (see above)
- addToSummary()
    - INPUT
        - local RETURN_VALUE="$1";
    - This function adds entries to the summary that will be printed by the exit_script function
    - This function is very handy when generating or installing several objects that are very similar
    - It his recommended to adjust or override this function since every use case is unique
- check_directory()
    - INPUT
        - local DIRECTORIES="$*";
    - This function accepts a list of directories/paths
    - It checks if those directories exist and exits the script if one of them does not exist
- check_file()
    - INPUT
        - local FILES="$*";
    - This function accepts a list of files/paths
    - It checks if those files exist and exits the script if one of them does not exist
- check_if_set()
    - INPUT
        - local VARIABLE_NAME="$1";
        - local DEFAULT_VALUE="$2";
    - This function accepts the name of a variable and optionally a default value for this variable

- The function checks if the variable is set and if not it writes the default value into the variable and if there is no default value specified the function exits the script

- check_if_command_available()
- INPUT
  - local COMMAND="$1";
  - This function checks whether a bash function is available and if not it exits the script
- enableLogging()
  - INPUT
    - local LOGFILE="$1"; (Optional, but SCRIPT_NAME has to be set)
  - This function enables logging into a file. It accepts a logfile name, but is also able to generate a name from system variables
  - It just works with the do_log function
    - enableLogging sets LOGGING to true and the function do_log checks this variable during the print operation
  - Errors will occur on screen and in the log. All other output is written just to the file
- check_user()
  - INPUT
    - local USER="$1";
    - local USERID="$2";
    - local HOMEDIR="$3";
  - Checks if a user exists and exits the script if not
- checkFormat()
  - INPUT
    - local TARGET_FORMAT="$(echo "$1" | xargs | tr "[:lower:]" "[:upper:]")";
    - local CURRENT_FORMAT="$(echo "$2" | xargs | tr "[:lower:]" "[:upper:]")";
  - Checks if the format of one string matches the target format
- check_diskspace()
  - Accepts a path and the minimum required diskspace
  - It checks whether there is enough diskspace at this path
- check_server()
  - Checks whether a server is available
- waitForListener()
  - Tries to connect to a port on a server
  - It is used by waitForServerUntilTimeout. It is recommended to use rather waitForServerUntilTimeout
- waitForServerUntilTimeout()

- Waits for a server until it's connected or timed out. If times out the script exits.

- printParameters()
  - This function accepts a list of variable names
  - It checks whether the variables are set and prints them
  - If they are not set the function exits the script
- removeStringIfContained()
  - This function removes a string from another string if it is contained
- create_group()
  - This function creates a linux group
- create_user()
  - This function creates a linux user
  - It accepts a lot of user related parameter, but all except the user name are optional
- remove_user()
  - Removes a linux user (including home directory and mail file)
- remove_group()
  - Removes a linux group
- copy_function()
  - Copies files or directories and creates the target folder if it does not exist
  - When something goes wrong during this process the script will exit
- delete_function()
  - Removes files or directories
  - Prints a warning if the target does not exist
- remove_content_from_file()
  - Removes contents from a file that are tagged by tags like:
    - ### START - INSTALL SCRIPT
      ### END - INSTALL SCRIPT
  - Tags are specified in call of the function
- searchAndReplaceInFile()
  - Search and Replace for strings in files
- appendFileContentToFile()
  - Appending content from one file to another file
  - Tagging (see above) of the appended content is possible
- appendScriptOuputToFile()
  - Appending the output of a script to a file
  - Tagging (see above) of the appended content is possible

- appendTextToFile()
  - Appending text to a file
  - Tagging (see above) of the appended content is possible
- recreate_folder()
  - Creates a new folder and its superior folders (if necessary)
- recreate_link()
  - Creates a symbolic to a file or folder
  - Recreates the link if there is an existing one that is broken
- check_if_empty()
  - Checks if a folder is empty
  - Return value depends on the state of the folder
- ask()
  - Asks for a value specified by the question
  - This function requires a default answer
  - Depending on the default value and the given answer the return value is chosen
  - Handy for yes or no questions
- askForInfo()
  - Asks for information and stores it into the variable INPUT
  - INPUT can be used afterwards
- askForPassword()
  - Asks for a password and stores it into the variable PASSWORD
  - The password is not shown on the screen while typing
- askForInfoAndCheckAnswer()

Erfahrungen in AWS:

1) Trainings:

- Architecting on Amazon Web Services
- Migrating to Amazon Web Services
- DevOps Engineering on Amazon Web Services
- Systems Operations on Amazon Web Services

2) Projekterfahung mit AWS:

- Ich konnte meine Kenntnisse in AWS auf meinem Projekt in einem großen deutschen Logistikunternehmen gewinnen.

Durch die Cloud-Innovation der letzten Jahre kamen immer mehr Unternehmen auf die Idee ihre on-premises Umgebungen durch die Cloud

zu ersetzen.

In diesem zusammenhang konnten wir unsere Testumgebungen in AWS aufsetzen und dem Kunden die Vorteile des Cloud-Computing nachweisen.

Als erstes war es wichtig ein Konzept der gegebenen Architekur von einem Ist-Zustand in ein Soll-Zustand zu entwerfen.

Dies hatte zur Folge, dass wir eine gute Kommunikation mit den Verantworlichen herstellten, um einen Überblick des Ist-Zustandes zu bekommen.

Das Ziel war es eine Testumgebung in die Cloud zu immigrieren, auf die schnell und zuverlässig zugegirffen werden kann.

AWS bietet viele verschiedene Services auf ihrer Plattform an, die viele Herausforderungen, wie Loadbalancing, queuing, emails senden und

speichern von wichtigen Daten, auf einer einfache Weise lösen kann. Durch die Bereitstellung einer eigenen API ist es möglich alles zu automatisiern.

Man kann ein Code entwickeln, der ein gesamtes Netzwerk erstellt, ein virtuellen Cluster oder relationale Datenbanken automatisch generiert.

Automatisierung erhöt die Zuverlässigkeit und verbessert die Effizienz.

Ein sehr großer Vorteil, den wir auch für unsere Testumgebungen genutzt haben, ist die Möglichkeit flexibel Kapazitäten zu skalieren.

Wir konnten uns dadurch vom intensiven Planen und Kostenüberschreitungen befreien. Wir konnten schnell von zwei oder drei Servern zu mehrern 100

Skalieren oder umgekehrt. Wenn ein Speicher voll gelaufen war, mussten wir uns keine Sorgen machen, dass Applikationen nicht mehr laufen würden, denn

der Storage konnte auch automatisch skaliert werden.

Die Herausforderung bei einem Logistikunternehmen ist es, dass man ein saisonales Verkehrsgebilde hat. Bedenkt man, Tag vs. Nacht, Wochentage vs. Wochenenden

oder Feiertage(Ferien). So konnten wir Kapazitäten hinzufügen, wenn der Traffic stieg und Kapazitäten entfernen, wenn der Traffic wieder sinkte.

Für den Anfang entschieden wir uns für folgende Infrastrukutr:

- Elastic Load Balancing (ELB) - AWS bietet einen Load Balancer als Service an. Der ELB verteilt den Traffic auf mehrere Servern dahinter. Dies bietet schon

für den Anfang eine Hochverfügbarkeit.

- Elastic Compute Cloud (EC2) - Virteulle Server werden als EC2 Services bereitgestellt. Wir nutzen Linux-Server mit Red Hat, da diese auch auf den on-premises Umgebungen installiert war.

Virteulle Server können fehlschlagen oder zumindest kann es zu einem ausfall kommen, daher haben wir uns entschieden zwei Server pro Umgebung zu nehmen.

- Relational Database Service (RDS) für MySQL - Um die generellen Datenbanken Services bei AWS zu testen, haben wir uns entschlossen auch eine MySQL Datenbank zu testen, um Backups und updates zu speichern.

Die Hochverfügbarkeit stellten wir mit einer Replikation der Datenbank her. So war es möglich bei Fehlern oder Ausfall der Datenbank auf die Backups zu greifen zu können.

- Security Groups - Security Groups ist ein fundamentaler Service von AWS um Netzwerk Traffic wie eine Firewall zu kontrollieren. Wir konnten die Security Gruppen mit unseren anderen

Services wie ELB, EC2 oder RDS verknüpfen. Unseren Elasitc Load Balancer haben wir so konfiguriert, dass dieser nur Anfragen aus dem Internet mit Port 80 akzeptiert.

Unser Web-Server akzeptierte eine Verbindung nur auf Port 80 vom Load-Balancer und unsere MySQL Datenbank akzeptierte nur Verbindungen auf Port 3306 vom Web-Server.

Die Gesamte Infrastruktur konnten wir in AWS in eine Resourcen Gruppe hinzufügen, um bei der Konfiguration immer drauf zugreifen zu können.

Um die Sicherheit unserer Testumgebung zu gewährleisten,lief diese in einer VPC (Virtual Private Cloud). Hier hatten wir die Möglichkeit

Routings, Subnetze, ACLs und Gateways zum Internet zu kontrollieren und zu konfigurieren.So kamen wir auch zu dem Entschluss zwei Subnetze zu konfigurieren.

Ein öffentliches (public) und ein privates (private) Netzwerk.

Das öffentliche Subnetz mit den Web-Servern sollte ein Weg zum Internet haben und das private mit unserer Datenbank nicht. Wir haben die Subnetze von einander getrennt, um möglichen Schaden

zu vermeiden und Hacker-Angriffe entgegen zu treten.

Letztendlich konnten wir unsere Testumgebung aufsetzen und so konfigurieren, dass wir unsere Cloud-Umgebung für eine bestimmte Zeit am Tag nutzen und wieder ausschalten. Am Wochende wurde

kein Service genutzt, somit wurden auch kosten gespart.

Erfahrungen in Azure:

1) Trainings:

- Architecting Microsoft Azure Solution

2) Projekterfahrung mit Azure:

- Verantwortlich für die Kernarchitektur eines internen Blockchain Projekts. Erstellung von Konzepten und Bereitstellung der Softwarekomponenten auf Basis von

Hyperledger Fabric. Unterstützung der Entwickler und Tester bei Ihren täglichen Aufgaben.

Erfahrungen in Docker:

1) Trainings:

- Docker Fundamentals

- Docker Advanced

2) Projekterfahrung mit Docker:

Andreas Schulz
Accenture Technology Solutions GmbH

Kaistraße 20
D-40221 Düsseldorf

Solutions
Telefon +49 151 17156133
andreas.schulz@accenture.com
Sitz: Kronberg. Registergericht: Königstein im Taunus, HRB 5968.
Geschäftsführer: Marcus Huth, Frank Mang, Andreas Schuler
