.NET | Blazor | Azure | KI | Angular | IoT | DevOps | JavaScript | Bots | Testing

S&S

windows .developer

windows .developer

7.2018

Deutschland 9,80 €
Österreich 10,80 €
Schweiz 19,50 sFr

www.windowsdeveloper.de

.NET ohne Grenzen

Browser + Razor = Blazor

Das neue Single-Page-App-Framework bringt C# in den Browser

.NET Core 2.x konfigurieren

Das neue Konfigurationssystem in der Praxis

4 195809 809803 07

Was bringt Angular 6?

Die Neuerungen in Framework und CLI

▶ 20

Unterstützung bei der Fehlerbehebung

Stabilere Anwendungen mit dem Framework Polly

▶ 46

Mehr KI für bessere Apps

App – Cloud – KI: Die neue Dreifaltigkeit

▶ 56

Teil 2: Das eigene DevTest Lab im Detail

Die richtige Formel finden

Im ersten Teil der Serie wurde Azure DevTest Labs allgemein vorgestellt und vor allem der Nutzen für unterschiedliche Stakeholder und Teams in Unternehmen gezeigt. Nachdem also klar sein sollte, dass sich der Einsatz von Azure DevTest Labs sich lohnt und wie ein erstes Lab eingerichtet wird, zeigt dieser Teil, wie man tiefer in das Thema eintauchen kann. Dazu werden einzelne Funktionen im Detail betrachtet und Automatisierungswege gebahnt.

von Sebastian Schütze und Tyrone Guiamo

Azure DevTest Labs wurde entwickelt, um Entwicklern und Administratoren die Arbeit zu erleichtern. Im Zuge der sich entwickelnden DevOps-Philosophie ist die Nutzung dieses Diensts nur einer von vielen logischen Schritten. Und da dieses Thema auch nach DevOps-Manier behandelt werden soll, werden wir nicht einfach ein Lab mithilfe des Portals erstellen. Wir werden zeigen, wie ein Lab mit ARM-Templates erstellt wird, eigene Artefakte genutzt werden und benutzerdefinierte Formeln hinzugefügt werden können.

Man kann nicht oft genug wiederholen, dass Standardisierung und Automatisierung die Fehleranfälligkeit enorm verringern. Besonders, wenn es um Flexibilität der Umgebung geht, kommt man nicht daran vorbei, dafür zu sorgen, weniger Aufwand in die Voraussetzungen der eigentlichen Aufgabe zu stecken.

Wenn wir Tests einmal als Beispiel heranziehen, dann geht es vorrangig darum, Fehler frühzeitig zu finden und die Qualität und die Kundenzufriedenheit zu steigern – und damit auch den Ruf der Firma zu verbessern. Wenn jedoch erst einmal viel Geld und Zeit investiert werden müssen, um die Voraussetzungen dafür zu schaffen, dann wird oftmals an der nötigen Vorarbeit gespart oder schlichtweg die gewünschte Umgebung abgespeckt. Das führt wiederum zu ungenauen Tests, weil man auf Umgebungen testet, die sich von der produktiven Umgebung zu sehr unterscheiden. Das negiert dann den eigentlichen Sinn von Tests und der Aufwand verschiebt sich in Richtung eines mühsamen Bugfixings und Troubleshootings, bis das Problem dann doch gelöst ist.

Deshalb werden wir im Folgenden ARM-Templates, Artefakte und Formeln aus DevTest Labs zeigen, um dieses Problem besser angehen zu können. Dabei nehmen wir auch gleich die Methodik der programmierbaren Infrastruktur auf, auch gut bekannt als Infrastructure as Code (IaC).

Lab mit PowerShell und ARM-Templates erstellen

Wir können natürlich das Lab ganz normal über die Weboberfläche des Azure-Portals erstellen. Eine bessere und der Infrastructure as Code folgende Methode wäre es jedoch, ARM-Vorlagen zu verwenden. ARM steht in diesem Fall für Azure Resource Manager und ermöglicht die Erstellung von zahlreichen virtuellen Infrastrukturen

Artikelserie

Teil 1: Einführung und Nutzen von Azure DevTest Labs
Teil 2: Das eigene DevTest Lab im Detail

in Azure. Dabei bezieht es sich jedoch nur auf virtualisierte Hardware. Mit der sogenannten PowerShell DSC (Desired State Configuration) lässt sich aber auch die Softwarekonfiguration steuern, wenn notwendig. Dies würde den Rahmen dieses Artikels jedoch sprengen. In diesem Fall beziehen wir uns auf ein Beispiel [1] aus dem öffentlichen GitHub Repository von Azure. Das angegebene Beispiel besteht aus folgenden Dateien:

- *azuredeploy.json*: Diese Datei beinhaltet die gesamte Definition der Infrastruktur und Konfiguration, die Azure benötigt. Sie ist die zentrale Datei, um unser DevTest Lab zu verteilen.
- *azuredeploy.parameters.json*: Dies ist eine optionale Datei, die die Standardwerte der Hauptdatei zu jeder Zeit überschreiben kann. Wird mit einem Skript oder dem CLI gearbeitet, muss diese Datei nur angegeben werden, wenn man Parameter überschreiben möchte. In manchen Fällen gibt es auch nicht gesetzte Pflichtparameter, die mit dieser Datei übergeben werden müssen.
- *metadata.json*: Diese Datei ist nicht für das Deployment selbst wichtig, sondern stellt lediglich eine Vorlagenbeschreibung bereit, die angezeigt wird, falls das Template über [2] bereitgestellt wird.

Nun zur Hauptdatei *azuredeploy.json*, die sich aus drei Teilen zusammensetzt:

Parameter (Listing 1): In diesem Bereich werden die Parameter definiert, die von außen im Skript überschrieben werden können. In unserem Fall ist es die JSON-formatierte Parameterdatei. Außerdem können hier Standardwerte vorgegeben werden, falls diese zur Erstellungszeit optional sein sollen. Parameter ohne Standardwerte würden zu Fehlern führen, falls diese nicht gefüllt sind.

Variablen (Listing 2): Während Parameter nur einfache Strukturen besitzen, können Variablen wesentlich komplexer und dynamischer sein. Diese Variablen können aus dynamischen Werten und verschiedenen Parametern zusammengesetzt sein. Hierbei können sogenannte Vorlagenfunktionen [3] zum Einsatz kommen. Diese werden zur Vereinfachung verwendet, da voneinander abhängige Ressourcen mehrfach referenziert werden müssen. Variablen vereinfachen und verkürzen hierbei das Template.

Ressourcen (Listing 3): Dieser Bereich ist der komplexeste Teil. In dieser Sektion werden alle Azure-

Listing 1

```
"parameters": {
  "labName": {
    "type": "string"
  },
  "artifactRepositoryDisplayName": {
    "type": "string",
    "defaultValue": "Team Repository"
  },
  #...Weitere Parameter ...#
  "artifactRepoSecurityToken": {
    "type": "securestring"
  }
```

Listing 2

```
"variables": {
  "labVirtualNetworkName": "[concat('Dtl', parameters('labName'))]",
  "artifactRepositoryName": "[concat('Repo-',
                              uniqueString(subscription().subscriptionId))]"
}
```

Listing 3

```
"resources": [
  {
    "apiVersion": "2017-04-26-preview",
    "type": "Microsoft.DevTestLab/labs",
    "name": "[parameters('labName')]",
    "location": "[resourceGroup().location]",
    "resources": [
    {
      "apiVersion": "2017-04-26-preview",
      "name": "[variables('labVirtualNetworkName')]",
      "type": "virtualNetworks",
      "dependsOn": [
        "[resourceId('Microsoft.DevTestLab/labs', parameters('labName'))]"
      ]
    },
    {
      "apiVersion": "2017-04-26-preview",
      "name": "[variables('artifactRepositoryName')]",
      "type": "artifactSources",
      "dependsOn": [
        "[resourceId('Microsoft.DevTestLab/labs', parameters('labName'))]"
      ],
      "properties": {
        "uri": "[parameters('artifactRepoUri')]",
        "folderPath": "[parameters('artifactRepoFolder')]",
        "branchRef": "[parameters('artifactRepoBranch')]",
        "displayName": "[parameters('artifactRepositoryDisplayName')]",
        "securityToken": "[parameters('artifactRepoSecurityToken')]",
        "sourceType": "[parameters('artifactRepoType')]",
        "status": "Enabled"
      }
    }
  ]
}
],
"outputs": {
  "labId": {
    "type": "string",
    "value": "[resourceId('Microsoft.DevTestLab/labs', parameters('labName'))]"
  }
```

Ressourcen definiert, die mit dem Template verteilt werden sollen. In dem angegebenen Beispiel werden zwei Ressourcen verteilt. Die erste Ressource ist DevTest Lab selbst. Man beachte dabei, dass hier als Lab-Name die Variable *labVirtualNetworkName* verwendet wurde:

```
"name": "[variables('labVirtualNetworkName')]",
```

Die zweite Ressource definiert die Konfiguration eines Artefakt-Repositories, das im Lab verwendet wird. Dabei folgt die Konfiguration dem gleichen Schema, wie es im späteren Abschnitt zu Artefakten erklärt wird. Wir wollen jedoch auch sicherstellen, dass die Konfiguration erst verteilt wird, nachdem das Lab erstellt wurde. Dies wird im Prinzip durch die folgende Zeile in dem Konfigurationsteil in Listing 3 erreicht:

```
"dependsOn": [resourceId('Microsoft.DevTestLab/labs',
                                        parameters('labName'))]",
```

Auf diese Weise wird sichergestellt, dass das Lab bereits existiert. Hierbei wird die Vorlagenfunktion *resourceId* verwendet, bei der mithilfe des Ressourcentyps und der erstellten Anzeigenamen die interne ID der Ressource zurückgegeben wird. Diese ID gibt dann die eineindeutige Referenz zur Azure-Ressource an.

Um dann letztendlich das Template mit einem PowerShell Skript zu verteilen, wird das Azure Cmdlet *New-AzureRmResourceGroupDeployment* empfohlen, das in [4] dokumentiert ist.

Listing 4

```
{
  "$schema": "https://raw.githubusercontent.com/Azure/azure-devtestlab/master/
                                        schemas/2016-11-28/dtlArtifacts.json",
  "title": "ArtifactTitle",
  "description": "Artifakt Beschreibung",
  "tags": [
    "MeineEigenerTag",
    "NochEinTag"
  ],
 "parameters": {
  "InternerParameterName": {
    "type": "string|int|bool|array",
    "displayName": "AnzeigeParameterName",
    "description": "BeschreibungParameter"
  }
 }
  "iconUri": "",
  "targetOsType": "Windows|Unix",
  "runCommand": {
    "commandToExecute": "Kommando abhängig von Windows oder Unix"
  }
}
```

Für eine Einführung wird die Verwendung der Vorlage empfohlen, bevor man anfängt, Anpassungen vorzunehmen. ARM-Templates sind ein mächtiges Thema und nicht einfach, da es viele Optionen gibt, aber auch Fehlermöglichkeiten. Daher werden zur Einführung die Dokumentationen unter [5], [6] und [7] empfohlen.

Standardisierung – eigene Artefakte nutzen

Images für VMs zu nutzen, ist allgemein schon ein guter Anfang auf dem Weg zur Standardisierung. Man kann auch gerne eine Maschine mit einer VM aufsetzen, in der z. B. bereits Visual Studio 2017 installiert ist. Was aber nun, wenn der Entwickler die gleiche VM braucht, doch stattdessen Eclipse benötigt? Man kann sicher ein zweites Image erstellen, aber dann haben wir zwei vorbereitete „Templates“, die nicht sehr flexibel sind und ebenfalls extra gewartet werden müssen. Artefakte setzen genau an diesem Problem an. Sie ermöglichen die Modularisierung von Softwarekomponenten und Konfigurationen für eine VM, die allein die Person auswählen kann, die am besten weiß, was sie braucht: der Entwickler. Also nutzt man einfach das Basis-Image und erstellt die VM, danach können die Artefakte vom Anwender ausgewählt werden, die benötigt und installiert werden sollen. Wie funktioniert dies nun? Im Grunde werden nur vier Dinge benötigt:

- Ein eigenes Repository (in VSTS oder gerne auch GitHub), in dem die Artefakte gespeichert werden
- Eine Manifest-Datei, die die Artefakte beschreibt
- Das Skript, das ausgeführt werden soll
- Ein Logo, das bei der Artefaktliste angezeigt wird (optional)

Listing 4 zeigt ein Beispiel für solch eine Manifestdatei. Die meisten der Elemente sollten selbsterklärend sein, wobei die Dokumentation [8] selbst eine gute Anleitung gibt. Es sei jedoch gesagt, dass Artefakte multiplattformfähig sind. Somit kann mit dem Element *runCommand* entweder PowerShell mit einem Skript gestartet oder einfach ein Linux-Bash-Kommando aufgerufen werden. Eventuelle Skriptdateien müssen dann im gleichen Ordner wie die Artefaktdatei liegen. Sie werden dann über das *commandToExecute* ausgeführt. Die Tatsache, dass Linux-Kommandos auch funktionieren, liegt daran, dass die zu installierenden Artefaktskripte vor der Installation auf die VM kopiert und dort ausgeführt werden. Zwei Beispiele für die Ausführung in Linux und auch Windows können in Listing 5 nachgelesen werden.

Listing 5

```
#Windows
"commandToExecute": "powershell.exe -File MyScript.ps1"
#Unix
"commandToExecute": "bash MyShellScript.sh"
```

Nun besitzen wir ein Artefakt, aber wohin damit? Am einfachsten ist es, in GitHub oder in ein eigenes Repository in VSTS zu integrieren. Da GitHub und VSTS Zugriffstokens für Repositories verwenden, müssen sie für DevTest Lab verfügbar gemacht werden. Schauen wir uns dazu das Formular in **Abbildung 1** zum Hinzufügen eines Artefakt-Repositories an.

Der einzige Punkt, auf den geachtet werden muss, sind die Rechte des sogenannten PAT-Tokens (Personal Access Token), das von GitHub oder VSTS erstellt werden muss. Es muss mindestens Leserechte auf das Repository haben, um in den Artefakten angezeigt zu werden. Zudem sollte darauf geachtet werden, dass der angegebene relative Ordner vollständig vom Quellverzeichnis des Git Repositories angegeben werden muss. Wenn keine Artefakte angezeigt werden, dann stimmt sehr wahrscheinlich etwas mit der Manifestdatei nicht. Für einen ersten Test kann auf das öffentliche GitHub Repository der Tuleva AG zugegriffen werden [9]. Dieses ist lediglich als einfaches Beispiel gedacht, wie solch ein Artefakt-Repository aussehen kann und soll einen Eindruck davon geben.

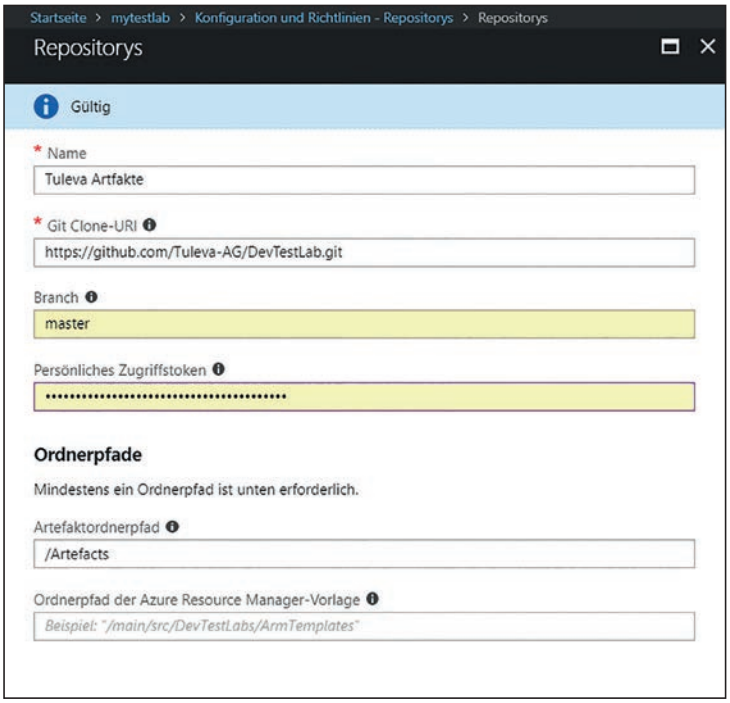

Abb. 1: Menü zum Hinzufügen eines neuen Artefakt-Repositorys

Standardisierung – Schritte zu eigenen Formeln

Nachdem wir einige Grundlagen, wie die Erstellung eines DevTest Labs bzw. von Artefakten, durchgegangen sind, können wir einen Schritt weiter gehen und eigene Formeln erstellen und deren Mächtigkeit zeigen. Eine Formel ist im Prinzip ein Template, das eine dynamische Möglichkeit bietet, VMs mit gewünschten Konfigurationen und Zuständen zu verteilen. Es gibt zwei Wege, Formeln zu erstellen: durch eine existierende VM im eigenen Lab oder von einem Basis-Image aus dem Marktplatz. Es gibt einige Unterschiede bei der Erstellung der beiden Varianten zu beachten. Weil deren Darstellung hier zu umfangreich wäre, verweisen wir auf einen ausführlichen Artikel dazu [10].

Abb. 2: Menü zur Erstellung einer neuen Formel auf Basis einer VM

Abb. 3: Menü zur Erstellung einer neuen Formel

Formeln aus einer VM erstellen

Eine Formel aus einer existierenden VM zu erstellen, kann dann empfohlen werden, wenn bereits eine VM mit der gesamten Konfiguration existiert und sich als wiederverwendbar herausgestellt hat. Somit könnte genau diese VM immer wieder einfach und ohne viel weitere Arbeit bereitgestellt werden. Möglich wäre damit eine Standard-VM für Entwickler oder eine Trainings-VM.

Um eine Formel zu erstellen, gehen wir zu der erstellten VM in DevTest Lab. Unter dem Punkt Operations (**Abb. 2**) wählen wir den Punkt Formel erstellen. Dabei müssen noch ein Titel und eine optionale Beschreibung angegeben werden. Die Formel ist dann nach kurzer Zeit verwendbar.

Um diese Formel dann zu verwenden, gehen wir einfach in das oberste Menü von DevTest Lab, wählen unter dem Bereich Eigenes Lab den Menüpunkt Formeln aus und finden dort die von uns erstellte Formel. Wenn eine Formel aus einer VM erstellt wurde, dann werden auch sämtliche Artefakteinstellungen aus der VM in die Formel übernommen. Einzelne Einstellungen wie Benutzerpasswort oder die Auswahl des Typs der Festplatte (HDD oder SSD) können weiterhin geändert werden.

Formeln aus einer Basis

Bei diesem Vorgehen wird eine Formel von Grund auf erstellt, was empfohlen wird, wenn man die volle Kontrolle über die neue VM und ihre Konfiguration haben möchte. Außerdem können eigene und von Microsoft bereitgestellte Artefakte leicht hinzugefügt werden. Dies kann unter dem Bereich Eigenes Lab unter dem Menüpunkt Formeln (wiederverwendbare Basis) (**Abb. 3**) im ausgewählten DevTest Lab durchgeführt werden.

Eine neue Formel wird durch Hinzufügen erstellt. Es kann sofort ein Basis-Image aus dem Marktplatz gewählt werden. Dort werden übrigens auch bereits selbst innerhalb des Labs erstellte Images oder Formeln als Basis angezeigt. Als Nächstes wird ein Formular angezeigt, in das die folgenden Daten eingetragen werden müssen:

Das Thema DevTest Labs ist sehr mächtig, wenn es darum geht, volle Kontrolle bei Trainings, Tests oder Entwicklungsszenarien zu haben; es lässt derzeit nur wenige Wünsche offen.

- Formelname,
- Beschreibung (optional),
- Benutzername und Passwort,
- Art der Festplatte (HDD oder SDD),
- Größe der Maschine,
- Liste der Artefakte, die mit installiert werden sollen,
- und erweiterte Einstellungen, wie das virtuelle Netzwerk, Subnet, IP-Adresseinstellungen

Außerdem kann eingestellt werden, ob die VM später von anderen zur Nutzung übernommen werden darf, und wie viele Instanzen standardmäßig sofort parallel erstellt werden sollen.

Achtung: Es können keine Formeln aus VMs erstellt werden, die zuvor mit ARM-Vorlagen erstellt wurden.

Nutzung von erstellten Formeln

Die eigenen Formeln lassen sich wiederum über verschiedene Wege erstellen. Ein selbsterklärender Weg ist die Erstellung über das Portal, die über das Menü der verfügbaren Formeln leicht gestartet werden kann.

Eine zweite Möglichkeit stellt die Erstellung von VMs mithilfe des Azure CLI (Command Line Interface) dar, das unter [11] heruntergeladen und installiert werden kann. Dieses Werkzeug ist sehr mächtig und funktioniert auch mit Linux. Es können viele Konfigurationen damit verwaltet werden, ohne jemals das Portal öffnen zu müssen. Der folgende Befehl soll die Erstellung einer VM aus einer Formel veranschaulichen:

```
az lab vm create --lab-name {LabName} -g {ResourceGroup}
                                    --name {VMName} --formula MyFormula
```

Natürlich muss sich vor der Verwendung erst ordentlich mit dem CLI gegenüber Azure authentifiziert werden, bevor man das Werkzeug nutzen kann.

DevTest Labs for the Win

Zusammenfassend kann gesagt werden, dass das Thema DevTest Labs sehr mächtig ist, wenn es darum geht, volle Kontrolle bei Trainings, Tests oder Entwicklungsszenarien zu haben. Egal, ob es darum geht, Entwicklungsumgebungen für neue Projekte sauber aufzusetzen, Trainings-VMs für Workshops vorzubereiten oder eine flexible Testinfrastruktur zu nutzen. Es lässt derzeit nur wenige Wünsche offen. Besonders die laufenden Kosten in Azure gehen dabei gegen Null, wenn keine anderen Ressourcen außer den Labs bereitgestellt werden.

Sebastian Schütze ist seit zehn Jahren Webentwickler, arbeitet seit fünf Jahren mit hybriden SharePoint-Plattformen und hat im Zuge der Entwicklung von Azure die Möglichkeiten der DevOps-Technologien für sich gefunden. Er arbeitet aktuell als Senior Azure/DevOps Consultant bei der Tuleva AG.

http://www.tuleva.de **sebastian.schuetze@tuleva.de**

@RazorSPoint

Tyrone Guiamo arbeitet als Microsoft Cloud Solution Architect bei Schindler und ist ebenfalls Koorganisator des monatlichen Azure Meetups in Berlin.

tyrone.guiamo@schindler.com

Links & Literatur

[1] Im Artikel verwendetes ARM-Template, um ein DevTest Lab zu erstellen: https://github.com/Azure/azure-devtestlab/tree/master/Samples/201-dtl-create-lab-with-artifact-repository

[2] Bereich in Azure, um Communitytemplates zu verteilen: https://portal.azure.com/#create/Microsoft.Template

[3] Vorlagenfunktionen, die bei ARM-Vorlagen verwendet werden können: https://docs.microsoft.com/de-de/azure/azure-resource-manager/resource-group-template-functions

[4] Dokumentation zur Verwendung des Azure Cdmlets „New-AzureRmResourceGroupDeployment“: https://docs.microsoft.com/en-us/powershell/module/azurerm.resources/new-azurermresourcegroupdeployment?view=azurermps-5.7.0

[5] Azure Resource Manager Overview: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview

[6] Create and deploy your first Azure Resource Manager template: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-manager-create-first-template

[7] Understand the structure and syntax of Azure Resource Manager templates: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-authoring-templates

[8] Erklärung der Manifestdatei für Artefakte: https://docs.microsoft.com/de-de/azure/devtest-lab/devtest-lab-artifact-author

[9] Beispiel eines einfachen Artefakt-Repositorys der Tuleva AG: https://github.com/Tuleva-AG/DevTestLab

[10] Vor- und Nachteile bei der Nutzung von Formeln und benutzerdefinierten Images: https://docs.microsoft.com/en-us/azure/devtest-lab/devtest-lab-comparing-vm-base-image-types

[11] Azure-Command-Line-Interface-Werkzeug von Microsoft: https://docs.microsoft.com/de-de/cli/azure/install-azure-cli?view=azure-cli-latest