MICROSOFT SERVICE FABRIC

Hochverfügbar und skalierbar

Verteilte Anwendungen dank Microservices und einer Laufzeitumgebung, die auf Microsoft Azure, Windows Server und Linux läuft.

Die Kombination aus Fabrik und Softwareentwicklung hat immer einen faden Beigeschmack. Schließlich werden Entwickler nicht müde zu betonen, dass das Schreiben von Software kein Prozess sei, der sich durch eine Fabrik gleichbedeutend einem Fließband abbilden ließe. Trotzdem hatte Microsoft auf der Build-Konferenz 2016 die Verfügbarkeit einer Service-Fabrik als Managed Service [1] gleich für drei Plattformen bekannt gegeben: Azure Service Fabric, Service Fabric für Linux und für Windows Server. Scott Guthrie war voll des Lobes: „… makes it easy to build and operate micro service based applications at tremendous scale."

Etwas ausführlicher beschrieben stellt Service Fabric eine Laufzeitumgebung für hochskalierende und ausfallsichere Applikationen zur Verfügung. Der Service ist für Cloud-Umgebungen verschiedener Anbieter wie auch für On-Premise-Umgebungen auf Windows und auf Linux [2] verfügbar.

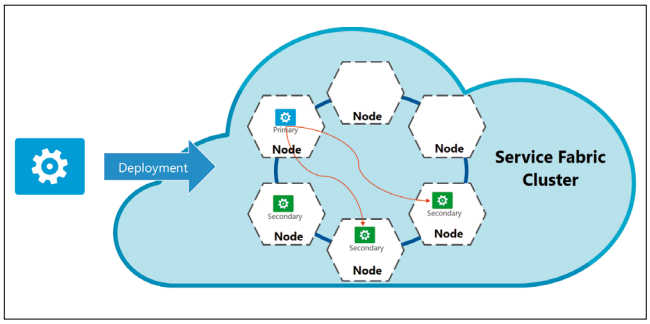
Dadurch ergibt sich erstmals die Möglichkeit, Applikationen für eine einheitliche Laufzeitumgebung zu entwickeln. Für bisherige Ansätze wie zum Beispiel Azure Web/Worker Roles existiert keine On-Premise-Laufzeitumgebung. Sie gibt es nur in Azure. Im Gegensatz dazu erlaubt der Ansatz mit Service Fabric den Betrieb auf den eigenen Servern, in der Azure Cloud und zusätzlich bei beliebigen Cloud-Anbietern. Somit steht Service Fabric on Premise, in der Cloud und über Betriebssystemgrenzen hinweg zur Verfügung.

Microsoft nutzt die Technologie intern bereits seit mehr als fünf Jahren und stellt die Laufzeitumgebung für eine breite Basis von Azure/Cloud-Diensten wie zum Beispiel Azure SQL Database, Azure Document DB, Intune, Cortana, Skype for Business et cetera zur Verfügung.

Microservices

Der Grundgedanke hinter Microservices besteht darin, komplexe Applikationen in kleine, lose miteinander verbundene Komponenten aufzuspalten [3]. Die jeweiligen Microservices sind dabei vollkommen unabhängig voneinander. Dies betrifft unter anderem die Auswahl der Entwicklungssprache und der Hostingumgebung wie auch die Wahl des Speichermediums für Status beziehungsweise Daten des Microservice.

Container-Umgebungen beziehungsweise Container-Orchestrierungstools wie Docker haben Microservices populär gemacht. Im Unterschied zu SOA-Architekturen ist bei Microservices auch der Datenbankteil beziehungsweise das Medium, auf dem der Status oder die Daten gespeichert werden, Bestandteil des Service.



Ein Service-Fabric-Cluster setzt sich aus einzelnen Knoten (Nodes) zusammen. Diese können alle auf einem oder auf mehreren Rechner verteilt laufen (Bild 1)

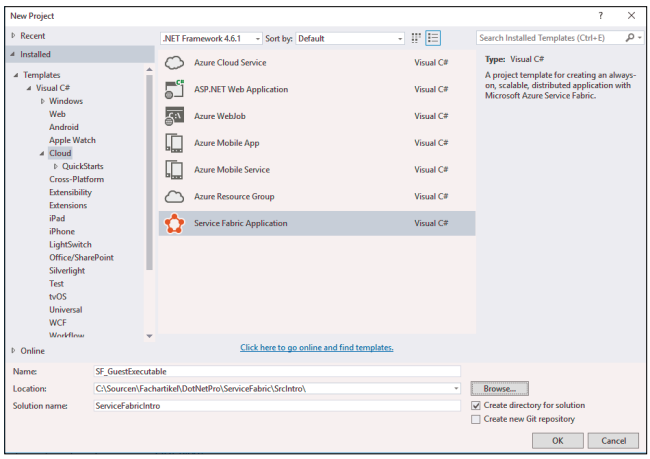
Unter einer Service-Fabric-Applikation kann man sich eine Zusammenfassung mehrerer Microservices vorstellen, die in ihrer Interaktion die gewünschte Applikationsfunktionalität zur Verfügung stellen. Jeder dieser Microservices entspricht hierbei einer autarken und funktionellen Einheit.

Allgemeiner Service-Fabric-Aufbau

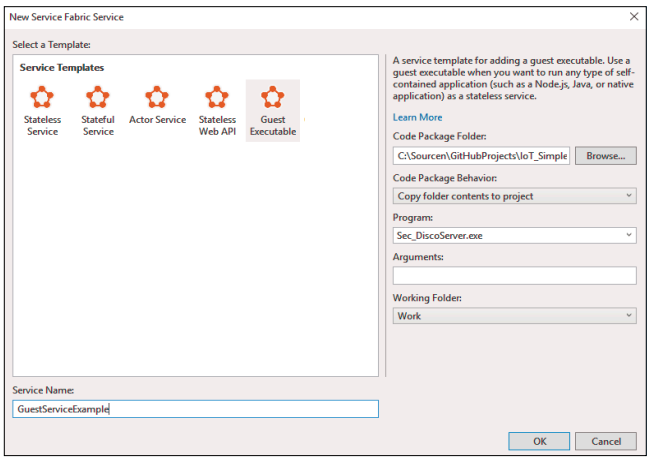
Die Service Fabric Runtime wird auf einer beliebigen Anzahl von physikalischen Servern installiert. Auf diesen physikalischen Servern laufen technisch betrachtet Service-Fabric-Runtime-Prozesse, die als Node bezeichnet werden.

In einer produktiven Laufzeitumgebung findet sich in der Regel ein Node-Prozess pro physikalischer Maschine. Bei der Entwicklung von Service-Fabric-Applikationen finden sich hingegen mehrere Node-Prozesse auf einem Entwicklungsrechner. Damit lässt sich die Verwendung von mehreren physikalischen Servern simulieren. Die Summe der physikalischen Server mit laufenden Node-Prozessen bildet den Service Fabric Cluster.

Wird nun eine Service-Fabric-Applikation auf dem Cluster verteilt, übernimmt der Cluster selbstständig die Installation der Applikation und damit die jeweiligen Microservices auf den Cluster-Nodes. Hierbei wird ein Node als Träger einer primären Installation gekennzeichnet. Dieser Instanz werden alle Aufrufe über den Service-Fabric-eigenen „DNS Dienst" weitergeleitet [4]. Zusätzlich zum primären Node werden für



Für ein neues Service-Fabric-Projekt steht ein Projekttemplate für Visual Studio bereit (Bild 2)



Dialogbox für die Auswahl des Executables und des Verzeichnisses (Bild 3)

jeden Microservice zusätzliche sekundäre Microservice-Installationen auf anderen Nodes deployt. Diese stehen je nach Konfiguration für Fail-over-Szenarien oder Read-only-Zugriffe zur Verfügung.

Sollte der zugrunde liegende Server oder der Node-Prozess ausfallen, erkennt das Service-Fabric-eigene Health System [5] diesen Fehler und kann eingehende Anfragen auf eine sekundäre Installation umleiten, die dann zur primären Installation erhoben wird. Parallel erzeugt der Service Fabric Cluster eine neue sekundäre Installation auf einem Node, um für weitere Ausfälle gewappnet zu sein.

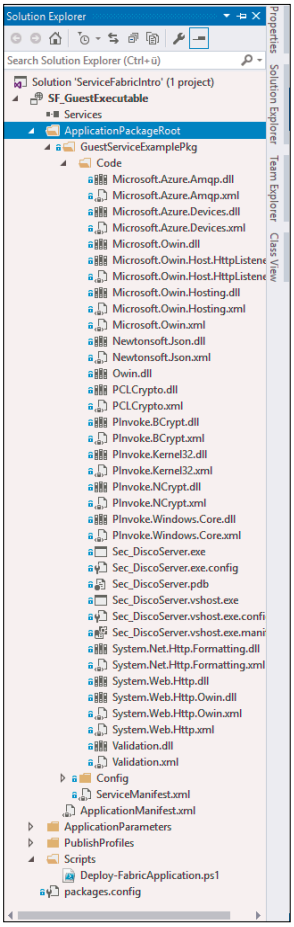
Die Programmiermodelle

Für die Entwicklung stellt Service Fabric drei Programmiermodelle zur Verfügung: Guest Executables beziehungsweise Guest Services, Reliable Services (Stateful und Stateless) sowie Reliable Actors.

Guest Executables / Guest Services

Bei den sogenannten Guest Executables beziehungsweise Guest Services handelt es sich um beliebige Executables/Assemblies, die auf dem Service Fabric zugrunde liegenden Betriebssystem ausgeführt werden können. Das heißt, ist Service Fabric auf einem Windows-System installiert, handelt es sich um Executables, welche auf der Windows-Plattform ausgeführt werden können, bei Linux als Betriebssystem um deren Äquivalente.

Das ist auch unabhängig davon, in welcher Programmiersprache, mit welchem Programmiermodell und so weiter das Executable entwickelt worden ist. Das Executable muss dabei keinerlei Service Fabric Libraries benutzen. Es weiß nichts davon, dass es in einem Service Fabric Cluster als Microservice ausgeführt wird. Service Fabric deployt das Executable auf den jeweiligen Nodes und startet es entsprechend beziehungsweise führt einen Fail-over aus, falls bei dem zugrunde liegenden Host ein Problem auftritt.



Die neue Solution ist angelegt (Bild 4)

Visual Studio macht es sehr einfach, ein entsprechendes Guest Executable beziehungsweise einen Guest Service als Bestandteil einer Service-Fabric-Applikation zu erzeugen. Im *New Project Wizard* unter *Templates* | *<Language of Choice>* | *Cloud* wählen Sie das Template *Service Fabric Application* aus. Sollte das Template nicht in der Auswahl verfügbar sein, kann mithilfe des Microsoft Web Platform Installer das Service Fabric SDK heruntergeladen und installiert werden [6].

Nach Auswahl des Templates im nachfolgenden Dialog *Guest Executable* wählen Sie sowohl das Verzeichnis mit dem auszuführenden Executable als auch das Executable aus. In dem Verzeichnis müssen alle zur Ausführung des Executable benötigten Dateien wie DLLs et cetera enthalten sein.

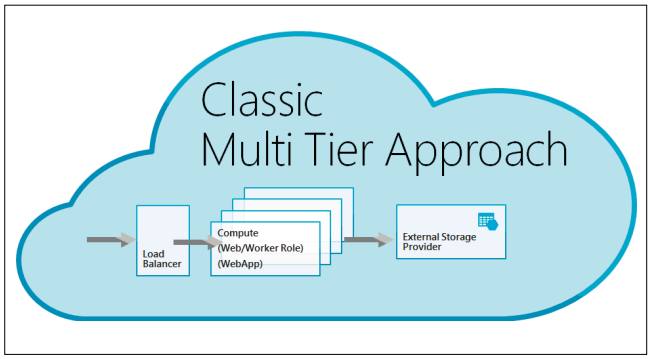
In der Textbox *Service Name* kann ein Name für den Microservice innerhalb der Service-Fabric-Applikation hinterlegt werden. Bestätigt man den Dialog mit *Okay,* wird eine neue Solution mit dem Guest Executable / Guest Service angelegt. Wählen Sie *Copy folder contents to project*, findet sich der Inhalt des *Code Package Folder* im Projekt unter *ApplicationPackageRoot\<ServiceName>Pkg\Code*. Der komplette Inhalt dieses Verzeichnisses wird beim Deployment zu einem Service Fabric Cluster auf die jeweiligen Nodes kopiert und das ausgewählte Executable gegebenenfalls gestartet.

Das sogenannte Application Manifest (siehe Listing 1) einer Service-Fabric-Applikation – siehe Datei *ApplicationManifest.xml* in der Visual Studio Solution – beschreibt den Aufbau der Service-Fabric-Applikation und damit, ▶

welche Microservices sich in der Applikation befinden, um welchen Typ es sich handelt, auf wie vielen Servern der Microservice installiert werden soll, sowie weitere Informationen, welche die Service-Fabric-Applikation beschreiben.

Reliable Services (Stateful und Stateless)

Reliable Services wissen im Gegensatz zu Guest Executables / Guest Services, dass sie in einem Service Fabric Cluster ausgeführt werden. Das heißt, sie können aktiv auf Ressourcen und Informationen des Service Fabric Cluster zugreifen. Eine dieser Ressourcen ist das Statemanagement. Dieses Statemanagement kann ein Microservice benutzen, um – einfach ausgedrückt – Daten abzulegen. Statt in einer Datenbank kann ein Service Daten und Informationen im Statemanagement des Service Fabric Cluster speichern. Dies ist insbesondere bei ausfallsicheren und skalierenden Applikationen von entscheidendem Vorteil.



Der klassische Ansatz für skalierbare Anwendungen (Bild 5)

Um den Vorteil zu verdeutlichen, betrachten Sie einen klassischen Ansatz für skalierende Applikationen (siehe Bild 5). Hierbei werden eingehende Anforderungen von einem Load Balancer an multiple Stateless-Compute-Instanzen weitergeleitet. Dabei kann es sich um virtuelle Maschinen, Web/Work Roles, WebApps, Container oder jedwede andere Form von Compute-Nodes handeln.

Kommen nun bei steigender Benutzung der Applikation mehr Anfragen am Load Balancer an und können diese nicht mehr von den existierenden Compute-Nodes verarbeitet werden, kommt es zu einem Stau am Load Balancer. Dieser Verarbeitungsstau kann durch Hinzufügen von weiteren Compute-Nodes aufgelöst werden. Nimmt die Benutzung der Applikation wieder ab, werden Compute-Nodes wieder aus dem System entfernt. Das System passt sich dadurch dynamisch dem jeweiligen Nutzeraufkommen an.

Als problematisch stellt sich jedoch die Skalierung der externen Datenbanksysteme dar. Da die Compute-Instanzen stateless ausgelegt sind, muss auch das nachgelagerte Datenbanksystem mit der Benutzeranzahl wachsen beziehungsweise wieder schrumpfen. Dies ist in der Praxis jedoch sehr schwer zu konfigurieren beziehungsweise teilweise unmöglich. Man stelle sich nur die Situation vor, einem SQL-Server-Cluster mehrmals am Tag zusätzliche Server hinzuzufügen beziehungsweise Server wieder aus dem Cluster zu entfernen, und das abhängig von der aktuellen aktiven Benutzeranzahl. In der Regel muss also ein Datenbanksystem herhalten, das ausreichend für die Spitzenlast konzipiert wurde und dementsprechend aber auch kostenintensiv und zeitweise nicht ausgelastet ist.

Die Abschaffung des Datenbanklayer

Azure Service Fabric adressiert dieses Problem nun auf sehr elegante und einfache Weise. Beim Deployment eines Reliable Stateful Service werden wie bei einem Guest Service / Guest Executable die Binaries auf multiplen Nodes installiert. Einer davon wird als Primary Node deklariert. Dies geschieht transparent für den Entwickler. Bei Reliable Services finden die gleichen Mechanismen bei einem eventuellen Ausfall der

Listing 1: Das Application Manifest

```
<?xml version="1.0" encoding="utf-8"?>
 <ApplicationManifest
  ApplicationTypeName="SF_GuestExecutableType"
  ApplicationTypeVersion="1.0.0"
  xmlns="http://schemas.microsoft.com/2011/01/fabric"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance">
  <Parameters>
   <Parameter Name="GuestServiceExample_InstanceCount"
    DefaultValue="-1" />
  </Parameters>
  <ServiceManifestImport>
   <ServiceManifestRef
    ServiceManifestName="GuestServiceExamplePkg"
     ServiceManifestVersion="1.0.0" />
   <ConfigOverrides />
  </ServiceManifestImport>
  <DefaultServices>
   <Service Name="GuestServiceExample">
    <StatelessService
     ServiceTypeName="GuestServiceExampleType"
     InstanceCount=
        "[GuestServiceExample_InstanceCount]">
     <SingletonPartition />
    </StatelessService>
   </Service>
 </DefaultServices>
</ApplicationManifest>
```

zugrunde liegenden Serverarchitektur Anwendung wie bei Guest Services / Guest Executables. Daher rührt auch der Name Reliable Service, da er gegen Ausfälle abgesichert ist.

Speichert der Reliable Service nun Daten, die im klassischen Fall in einer Datenbank abgelegt worden wären, geschieht dies über ein spezielles API des Service Fabric Cluster (Bild 6). Der Cluster wiederum speichert die Daten auf dem Primary Node auf der Festplatte beziehungsweise im Hauptspeicher und repliziert diese Daten zu den Secondary Nodes.

Dadurch ist bei einem Ausfall eines Nodes durch Umschalten auf einen Secondary Node nicht nur sichergestellt, dass sich das Executable dort befindet. Das Gleiche gilt auch für den Status, das heißt für die Daten des Microservice. Der Datenbanklayer wird durch dieses Verfahren Bestandteil des Compute-Layer! Der Microservice greift also für die Erfüllung seiner Aufgabe nicht mehr auf ein Datenbanksystem zu, sondern speichert alle notwendigen Daten über ein API der Service Fabric direkt auf dem Compute-Node. Dies klingt im ersten Schritt gewöhnungsbedürftig, zeigt seine Leistungsfähigkeit jedoch beim Praxiseinsatz.

Entwicklung eines Reliable Stateful Service

Visual Studio stellt für die Entwicklung eines Reliable Stateful Service auch ein Template zur Verfügung. Ähnlich wie bei Guest Executables / Guest Services kann im *New Project Wizard* unter *Templates | <Language of Choice> | Cloud* das Template *Service Fabric Application* sowie *Stateful Service* ausgewählt werden. Ein Reliable Service ist vom zugrunde liegende Projekttyp her eine Konsolen-Applikation, welche den Service in der Service Fabric Runtime registriert und von dieser gestartet und gemanagt werden kann. Details finden sich unter [7]. Entwickelt man einen Stateful Service, so stellt das Service Fabric SDK eine Basisklasse *StatefulService* zur Verfügung, von der abgeleitet werden muss.

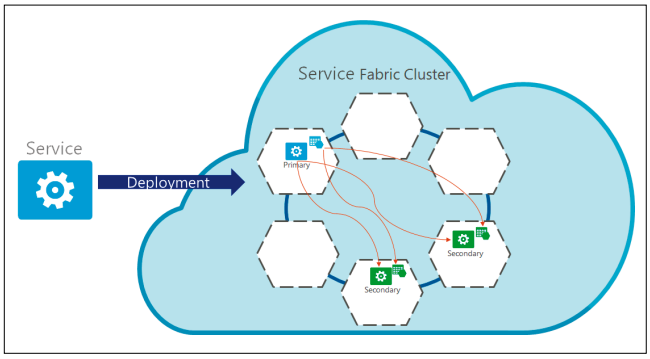
```
internal sealed class DeviceApiPartition :
        StatefulService {  }
```

Aktuell stellt die Basisklasse eine Read-only-Property *StateManager* vom Typ *IReliableStateManager* zur Verfügung. Diese Property stellt die Möglichkeit zur Speicherung von Status innerhalb eines Reliable Service zur Verfügung. Im Detail können über den *StateManager* ein sogenanntes *ReliableDictionary* und eine *ReliableQueue* [8] angefordert beziehungsweise erzeugt werden. Alle Werte, welche in dem Dictionary oder in der Queue eingestellt beziehungsweise entfernt werden, werden automatisch zu den sekundären Installationen des Microservice auf die unterschiedlichen Nodes repliziert. Beispiel Auslesen von Werten aus einem *ReliableDictionary* über den *StateManager*:

```
private async Task<string> StoreDataAsync()
{
 IReliableDictionary<String, String>
  reliableDictionary = await this.StateManager
   .GetOrAddAsync<IReliableDictionary<String, String>>
     ("<<SomeUniqueId>>");
    string position = "";
 using (ITransaction transaction =
  StateManager.CreateTransaction())
 {
  var Value = await reliableDictionary
   .TryGetValueAsync(transaction, "Position");
  if (conditionalValue.HasValue)
    position = conditionalValue.Value;
  }
  return position;
}
```

Zum Schreiben von Werten in eine *ReliableQueue* kann die Methode *AddOrUpdateAsync()* verwendet werden.

```
private async Task<bool> StoreDataAsync()
{
 IReliableDictionary<String, String> reliableDictionary
  = await this.StateManager.GetOrAddAsync<
   IReliableDictionary<String, String>>
     ("<<SomeUniqueId>>");
 using (ITransaction transaction = StateManager
  .CreateTransaction())
 {
  string position = "<<SomeValueToBeStored>>";
  var result = await reliableDictionary
```



Ablegen der Daten im Fall des Reliable Service (Bild 6)

```
   .AddOrUpdateAsync(transaction, "Position",
    position, (key, existingValue) => position);
  await transaction.CommitAsync();
 }
 return true;
}
```

Reliable Actors

Mit Reliable Actors stellt die Service Fabric ein zusätzliches Programmiermodell zur Verfügung. Dieses basiert auf dem Actor Pattern [9]. Ein Actor ist hierbei eine virtuelle, isolierte, unabhängige single-threaded Compute-Einheit.

Vereinfacht handelt es sich bei einem Actor um eine .NET-Klasse, wobei sich Service Fabric um die Instanzierung, ▶

Speicherung von Properties, Lebensdauer et cetera der Klasse kümmert. Das heißt im konkreten Fall: Wenn eine Business-Logik eine Instanz einer Klasse benötigt, so wird diese nicht mit dem Schlüsselwort *new* erstellt, sondern vielmehr wird das Service Fabric SDK verwendet:

```
Uri serviceUri = "<<ServiceFabricMicroServiceUri>>";
ActorId actorId = new ActorId("<<SomeUniqueId>>");
IActorInterface deviceActor = ActorProxy.Create<
  IActorInterface>(actorId, serviceUri);
```

Service Fabric stellt nun sicher, dass eine Instanz der Klasse auf einem Node innerhalb des Service Fabric Cluster erstellt wird. Sollte jemals in der Laufzeit des Clusters bereits eine Instanz der Klasse mit der angeforderten ID existiert haben, so wird diese mit dem letzten Status der Klasse wieder erstellt. Das heißt, Properties der Klasse weisen wieder den Wert der letzten Benutzung auf, unabhängig davon, ob die Klasse vor wenigen Sekunden oder vor vielen Jahren das letzte Mal benutzt wurde!

Service Fabric stellt sicher, dass der Status beziehungsweise der Wert der Properties im Cluster gespeichert und wie bei Reliable Stateful Services sicher im Cluster auf primären und sekundären Instanzen und Nodes gespeichert werden. Die Speicherung erfolgt hierbei in Memory oder auf Festplatte.

Sollte eine Klasseninstanz über einen bestimmten Zeitraum nicht verwendet werden, stellt Service Fabric sicher, dass die Instanz aus dem Hauptspeicher entfernt und auf Disk persisiert wird. Beim nächsten Zugriff oder bei der nächsten Instanzierung wird dieser Status von der Festplatte wiederhergestellt.

Um dieses Verhalten sicherzustellen, verwenden Actors genauso wie Reliable Stateful Services das Konzept eines *StateManager*:

```
[StatePersistence(StatePersistence.Volatile)]
internal class SomeActor : Actor, IActorInterface
{
 public Task<string> GetValueAsync()
 {
  return this.StateManager
   .GetStateAsync<string>("<<SomeKey>>");
  }

  public Task SetLocationAsync(string someValue)
  {
   return this.StateManager.AddOrUpdateStateAsync(
    "<<SomeKey>>", someValue, (key, value) =>
     someValue != value ? someValue : value);
  }

  protected override Task OnActivateAsync()
  {
   this.StateManager.AddStateAsync("<<SomeKey>>", "");
   return Task.FromResult<object>(null);
  }
}
```

Service Fabric bildet intern einen Actor mithilfe von Reliable Stateful Services ab. Das bedeuted, dass Aktoren eine Vereinfachung des Programmiermodells von Reliable Stateful Services darstellen. Wie bei jeder Vereinfachung erkauft man sich das einfache Modell mit dem Verzicht auf spezifische Funktionalitäten. Eine Übersicht mit Best Practices, wann Aktoren erfolgreich eingesetzt werden können und wann vom Einsatz abzuraten ist, findet sich unter [10].

Fazit

Aus Sicht des Autors stellt die Service Fabric eine Runtime zur Verfügung, um Applikationen zu entwickeln, die sowohl on Premise als auch in der Cloud bei beliebigen Anbietern ausgeführt werden können. Das bedeutet: Die Entwicklung muss nicht vorab entscheiden, bei welchen Anbietern beziehungsweise auf welcher Cloud-Runtime die Software ausgeführt werden kann. Die Verfügbarkeit von Service Fabric sowohl auf Linux als auch auf Windows Server stellt zusätzlich eine Funktionalität dar, die es gerade in größeren Unternehmen erlaubt, auf ein einheitliches Entwicklungsmodell unabhängig von bevorzugter Programmiersprache oder Betriebssystem hochskalierende und auch ausfallsichere Microservices zu erstellen. Ein guter Einstieg mit weiterführenden Informationen zu Service Fabric findet sich unter [11]. ■

[1] Azure Service Fabric, www.dotnetpro.de/SL1703ServiceFabric1
[2] Service Fabric on Linux, www.dotnetpro.de/SL1703ServiceFabric2
[3] Microservices, www.dotnetpro.de/SL1703ServiceFabric3
[4] Overview of Azure Service Fabric, www.dotnetpro.de/SL1703ServiceFabric4
[5] Connect and communicate with services in Service Fabric, www.dotnetpro.de/SL1703ServiceFabric5
[6] Introduction to Service Fabric health monitoring, www.dotnetpro.de/SL1703ServiceFabric6
[7] Reliable Services overview, www.dotnetpro.de/SL1703ServiceFabric7
[8] Reliable Collections, www.dotnetpro.de/SL1703ServiceFabric8
[9] Get started with Reliable Services, www.dotnetpro.de/SL1703ServiceFabric9
[10] Actor model, www.dotnetpro.de/SL1703ServiceFabric10
[11] Introduction to Service Fabric Reliable Actors, www.dotnetpro.de/SL1703ServiceFabric11

Robert Eichenseer
entwickelt seit langem Applikationen auf Basis der Microsoft-Technologien. Aufgrund seiner Arbeit in internationalen Projekten kann er Anforderungen heutiger Softwareentwicklungen umsetzen oder betriebswirtschaftliche Prozesse Entwicklungsthemen zuordnen.

dnpCode A1703ServiceFabric