Sonderdruck aus JavaSPEKTRUM

Serverless all over the world?

Moderne Architekturmuster in der serverlosen Welt

Michael Kayser, Lino Janke

Serverless all the world? Und das alles ohne eigene beziehungsweise ohne dedizierte Infrastruktur – geht das überhaupt? Wir zeigen, was bei der Verwendung von Serverless Functions berücksichtigt werden muss, um vollständige Anwendungen in der Cloud zu implementieren. Neben skalierbaren Quellcodebausteinen entsteht eine anfangs ungewohnte Architekturform mit zahlreichen zusätzlichen Diensten und ganz neuen Herausforderungen.

Serverless – Was ist das? Ob AWS Lambda, Google Cloud Functions oder Microsoft Azure Functions – Serverless ist in aller Munde. Aber was steckt dahinter? Der Function-as-a-Service-Ansatz (FaaS) beschreibt die Verlagerung von Infrastruktur und Skalierungslogik hin zum Dienstanbieter. Der Entwickler soll sich nur noch auf die Geschäftslogik konzentrieren und die passenden Bausteine für die Applikation vorher aus dem bereitgestellten Portfolio auswählen. So stehen neben den klassischen FaaS-Diensten weitere verwaltete Dienste in allen Cloud-Umgebungen zur Verfügung.

Das Versprechen: Fernab von Netzwerken, virtuellen Maschinen, Docker und Kubernetes ermöglicht es Serverless, komplexe verteilte Anwendungen zu bauen, die dynamisch skalieren und kosteneffizienter eingesetzt werden können. Inwieweit das Versprechen gehalten werden kann, wird in der folgenden Fallstudie herausgearbeitet.

Projektanforderungen an die Applikation

Das Szenario: Ein großes Unternehmen (ca. 7.000 Mitarbeiter weltweit, mehr als 1 Mrd. Euro Jahresumsatz) besitzt zahlreiche Tochtergesellschaften, die jeweils eigene heterogene Systemlandschaften betreiben. Deshalb ist es für die Muttergesellschaft nur mit hohem Aufwand möglich, einen gesamtheitlichen Überblick über Unternehmenskennzahlen für das ganze Unternehmen zu erhalten. Dies beinhaltet Fragestellungen wie „Welches unserer Produkte fragen die Kunden am stärksten nach?", „Produzieren wir an verschiedenen Standorten das gleiche Produkt?" oder „Welche Produkte sind global am ertragreichsten?".

Aus diesen klassischen Managementfragen ist der Bedarf nach einem zentralen System entstanden, an das die verschiedenen IT-Systeme angeschlossen werden können, das erforderliche Daten harmonisiert, mit zusätzlichen Informationen anreichert und sie mit verbesserter Datenqualität wieder zur Verfügung stellt. Die Tatsache, dass es sich vorrangig um Stammdaten und Bewegungsdaten handelt, macht die gewachsene Anwendungslandschaft nicht weniger komplex. Zusätzlich bestehen Anforderungen zur Nachverfolgbarkeit aller Daten auf deren konkreten Ursprungsort sowie Anforderungen zur Erweiterbarkeit des Systems um weitere Quellsysteme, manuelle Anpassungsschritte, Ausweitung der Anwendungsfälle und künftige Erhöhung der Datenlast.

Um diesen Anforderungen gerecht zu werden, wurde das Architekturmuster des Event Sourcing gewählt.

Was ist Event Sourcing?

Eine Welt voller Ereignisse ist schon häufig thematisiert worden. Wer dazu mehr erfahren möchte, dem seien Veröffentlichungen von Greg Young und Udi Dahan ans Herz gelegt. Im vorliegenden Anwendungsfall gilt es, den vom Kunden auf Datei- und Tabellenbasis beschriebenen Prozess für einen ersten Prototyp in Aufgaben und Ereignisse aufzuteilen:

- Ereignisse sind dabei Statusänderungen einzelner Entitäten oder Prozessschritte, die ihrerseits die Ausführung von weiteren Aufgaben nach sich ziehen können.
- Aufgaben sind kleine Handlungsschritte, welche in der Regel Geschäftslogiken abbilden und neue Ereignisse produzieren können.

Entsprechend den Grundsätzen von Event Sourcing wird jedes einzelne Ereignis persistent mit Zeitstempel gespeichert und kann (später) als Aggregation wieder zu einem aktuellen Systemzustand zusammengefügt werden. Diese Aggregationen wiederum sind jedem Zielsystem als *Projektion* passgenau zugeschnitten. Dabei können unterschiedliche Regeln und Selektionen zum Aggregieren der Ereignisse angewandt werden. Die zeitlich geordnete Menge an Änderungen an einem Objekt wird als *Event Stream* bezeichnet.

Erste Überlegungen

Der Kunde nutzt bereits in anderen Projekten die Azure-Plattform von Microsoft. Anhand der Anforderungen wurden aus der Vielzahl der verwalteten Dienste von Azure diejenigen ausgewählt, die auf den ersten und auch auf den zweiten Blick am geeignetsten für den Anwendungsfall erschienen. Bevor aber damit begonnen werden kann, ein verteiltes Cloud-System zu designen, ist es ratsam, sich über die Möglichkeiten zur Analyse Gedanken zu machen. In Microsoft Azure gibt es das Bordmittel Application Insights. Diese Insights bieten einen tiefen Blick in die automatischen und manuellen Logausgaben mit vollständiger Durchsuchbarkeit und Strukturierung von Logdaten mittels *Azure Log Monitor*. Außerdem bietet das Tool Dashboards für die Analyse von aufgetretenen Fehlern mit Auswertung von Häufigkeit und Auslöseort. Daneben lassen sich bei Verwendung von verwalteten unterstützten Diensten sämtliche Abhängigkeitsaufrufe und deren Dauer und Häufigkeit auswerten.

Damit hat man insbesondere für Performance-Betrachtungen ein mächtiges Mittel von Hause aus an der Hand. Müssen die Analysen technischer in die Tiefe gehen und werden konkrete Logdateien benötigt, stellt Azure Functions selbst mit Apache Kudu ein Analysetool aus der Hadoop-Welt zur Verfügung, welches neben Dateisystemzugriff in der skalierbaren Serverless-Welt auch Fehler- und Laufzeitanalysen ermöglicht. Auch das gute alte Debugging stellt eine Möglichkeit zur Fehleranalyse bereit. Dies funktioniert mithilfe der Standardmechanismen der jeweiligen IDE und nach vorheriger Aktivierung des Remote Debuggings über das Azure-Portal für die jeweilige Azure Function App. Zu beachten ist, dass die Ge-

Michael Kayser ist Senior Developer bei EXXETA in Leipzig. Seine Aufgaben umfassen die Implementierung von Business-Logik im Energy-Sektor mit C# oder Java und die Realisierung von Cloud-Komponenten im Azure-Stack. Zusätzlich unterstützt er als Architekt im Bereich Serverless-Technologien und Managed Services in der Azure Cloud.
E-Mail: michael.kayser@exxeta.com

Lino Janke ist Senior Developer bei EXXETA in Leipzig. Sein Tätigkeitsfeld erstreckt sich von der Implementierung komplexer Business-Monolithen im Energy-Sektor mit C# unter .NET4.6 bis hin zur Verwendung neuester Azure-Cloud-Technologien unter .NET Core 3.1. Daneben ist er als Architekt neuer Cloud-Anwendungen verantwortlich für die Verwendung von Serverless-Technologien und Managed Services. E-Mail: lino.janke@exxeta.com

schwindigkeit des Remote Debuggings mit einer lokalen Anwendung nicht vergleichbar ist.

Ausgerüstet mit diesem Toolset kann die Auswahl der Dienste und deren Auswirkung auf das System schnell beurteilt werden. Die FaaS-Lösung *Azure Functions* wurde ausgewählt, weil das Aktualisierungsintervall der Daten zeitlich fest definiert ist und bei Nicht-Nutzung kaum Kosten verursacht. Für die Verteilung der Events wurde das *Azure Event Grid* gewählt und für die Persistenz der semistrukturierten Daten die verteilte, dokumentenbasierte Datenbank *Azure Cosmos DB*. Dieses Setup versprach eine hohe Skalierbarkeit bei gleichzeitig überschaubaren Infrastrukturkosten. Die Prozesssteuerung sollte, wie in zahlreichen Empfehlungen von Microsoft gezeigt, *Azure Logic Apps* übernehmen. In der Auswahl inklusive ist ein *Storage Account* für die Functions sowie die Loganalyse- und Monitoring-Komponente *Application Insights*.

Erster Architekturansatz

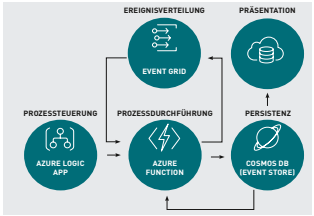
Abbildung 1 zeigt, wie der Fachanwendungsfall mittels *Azure Logic Apps* abgebildet wurde und dabei noch kein vollständiges Event Sourcing zum Einsatz kommt. Es gibt noch keine aktuelle Repräsentation des Status des Domänenmodells und jeder Zugriff auf eine Aggregation erfordert das Aggregieren aller zum Stream gehörenden Events. Die Probleme, die dabei entstanden, sind:

- Automatisierte Auslieferungen und Updates von Azure Logic Apps sind kompliziert.
- On-the-fly-Aggregationen der Events zu einem aktuellen Zustand erfordern erhöhte Lesezugriffe auf der Cosmos DB, welche die Kosten unnötig nach oben treiben.
- Der Trigger der Cosmos DB, welcher die persistierten Events zur Verarbeitung an den Event-Prozessor weitergeben sollte, funktioniert nicht wie erwartet, da man anhand des geworfenen Events nicht unterscheiden kann, ob ein Eintrag erzeugt oder aktualisiert wurde. Außerdem arbeitet der Trigger mit einem Polling-Mechanismus, der bis zu 10 Sekunden Abstände haben kann.

Zweiter Architekturansatz

Nach dem Review des ersten Architekturansatzes wurden erste Veränderungen vorgenommen. Die wesentlichsten Änderungen zeigen sich wie folgt (vgl. Abb. 2): Die Azure Logic Apps wurden vollständig durch *Durable Functions* ersetzt. Der Vorteil dabei ist, dass die Prozesse mittels Quellcode modelliert werden können und sie gemeinsam mit der Fachlogik auslieferbar sind (da Durable Functions in der gleichen App wie Azure Functions existieren können).

Abb. 1: Architekturansatz mittels Azure Logic Apps



Bei der Verwendung von Durable Functions empfiehlt es sich auf jeden Fall, die Best Practices von Microsoft zu konsultieren und zu verinnerlichen [Mic-a]. Das Problem zu vieler Datenbankzugriffe wurde durch verschiedene Änderungen gemildert:

- Einführung eines ReadModel, welches den aktuellen Stand eines jeden Event Stream (die Aggregation) flüchtig beibehält.
- Ablage dieses ReadModel, welches von jeder Function benutzt wird, im Redis Cache for Azure, um den Datenzugriff zu beschleunigen und den Durchsatz der Cosmos DB zu verringern.
- Auslagerung von großen Daten (z. B. Dateiinhalt und einzelne unstrukturierte CSV-Zeilen) in einen Azure Blob Storage mit eindeutiger Referenz auf einen zugehörigen Event Stream.

Mit diesen Änderungen konnten vor allem Konsistenz in der Datenverarbeitung und eine deutliche Verbesserung der Laufzeiten erreicht werden. Da jedoch weiterhin Engpässe in der Skalierung sichtbar waren und Nachrichten vom Event Grid verloren gingen, wurden die eingesetzten Komponenten mit den bestehenden Erfahrungen erneut einem Review unterzogen und alle technischen Entscheidungen hinterfragt, um die Qualität weiter zu verbessern.

Dritter Architekturansatz

Der zweite Architekturansatz stellte bereits eine wesentliche Verbesserung dar, jedoch war noch viel Potenzial für Optimierungen vorhanden. Das unzureichende Maß an Skalierung konnte auf die Menge an HTTP-Einzelanfragen zurückgeführt werden. Trotz Verarbeitungsraten von 100 HTTP-Anfragen pro Sekunde arbeitete das System langsam. Bei der Trennung fachlicher Komponentenbausteine von einem zentralen Kernsystem (so wie vom Kunden gefordert) ergaben sich neue Herausforderungen. Insbesondere die dynamische Erweiterbarkeit um neue Module ohne zentrales Dienstmanagement (Service Registry, Service Discovery) stellt sich dank loser Kopplung als schwierig heraus.

Die wichtigste Erkenntnis nach einer erneuten Evaluierung: Fast jeder Aufruf einer Funktion führt am Ende zu einer HTTP-Anfrage an einen anderen Dienst. In der „alten" Welt passiert das alles im Arbeitsspeicher, für eine Durchsatzverbesserung im Cloud-System kann nur das Reduzieren der Anfragemenge an weitere Dienste und

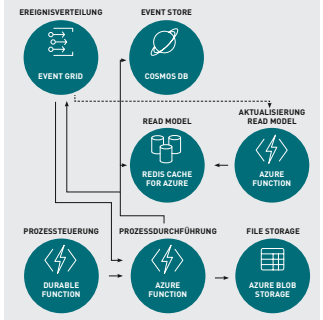


Abb. 2: Architekturansatz mittels Durable Functions

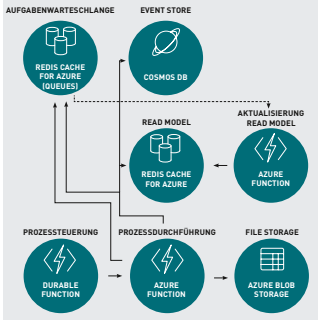


Abb. 3: Architekturansatz mithilfe von Redis

das Einführen von Stapelverarbeitungsschritten zu besseren Ladezeiten führen. Das bedeutet, in einer Anfrage sollen so viele Verarbeitungsschritte wie möglich gleichzeitig beauftragt werden. So stellte sich heraus, dass das Event Grid zwar sehr gut geeignet ist, schnell andere Komponenten wie Azure Functions oder Externe zu benachrichtigen, jedoch kaum Möglichkeiten für Failover Handling oder Stapelverarbeitung bietet. Einmal zugestellte Nachrichten gelten als erledigt, was zu ungeplanten Seiteneffekten führte.

Entsprechend der Zielstellung, die Aufgabenwarteschlange in einem anderen System mit Stapelverarbeitungsfunktionalität abzulegen, wurden der Aufbau und die Abarbeitung von Aufgabenqueues mithilfe von Redis implementiert. Zusätzlich wurde jeder offenkundig „langsame" Zugriff, der mehr als einmal im Prozessdurchlauf erforderlich ist, im Redis Cache flüchtig zwischengespeichert. Mit diesen beiden Maßnahmen und einem leicht veränderten Architekturbild (vgl. Abb. 3) wurde der Datendurchsatz weiter erhöht.

Vierter Architekturansatz

In der vorerst letzten Evolutionsstufe (vgl. Abb. 4) konnte auf reichhaltige Erfahrungen der letzten Iterationen zurückgegriffen werden und es wurde basierend auf den vorhandenen Erkenntnissen die agile Architektur weiterentwickelt. Queues in Redis haben den Nachteil, dass sie weder gefiltert noch geroutet werden können. Es kann also immer nur einen Empfänger für eine Nachricht geben. Ein Message-System nachzubauen, lohnt sich bei der Anzahl verfügbarer Systeme nicht. Deshalb wurden erneut die getroffenen Entscheidungen überdacht und geprüft, wie und mit welchen verwalteten Diensten von Azure filterbares Broadcast-Messaging aufgebaut werden kann.

Der anfangs für zu umfangreich und mächtig befundene *Azure Service Bus* wurde erneut analysiert. Die Nachteile der Queues im Redis Cache, bezogen auf den konkreten Anwendungsfall, konnten laut Beschreibung mit dem Azure Service Bus ausgeräumt werden. Das Einliefern und Empfangen von Nachrichten ist in Batches möglich. Nachrichten, welche nicht erfolgreich verarbeitet wurden, können zurück auf die Queues beziehungsweise Topics transferiert werden, ohne bei anderen Subscribern eine Doppelverarbeitung zu verursachen. Eine reichhaltige SQL-basierte Programmierschnittstelle ermöglicht es per REST oder Client-API, Routen und Verarbeitungsendpunkte einzurichten. Spannend: Das Feature zum Senden und Empfangen von Batches wurde zum Zeitpunkt der Entwicklung erst seit zwei Wochen offiziell unterstützt. Dazu wurde für erste Tests vorläufig auf Preview-Versionen der Client-Bibliotheken zurückgegriffen, um später etwas gereiftere Versionen zum Einsatz bringen zu können.

Durch Topics, Routing und Subscriptions ist es möglich, aus erzeugten Events zeitgleich mehrere neue Aufgaben abzuleiten. So erreicht eine Nachricht in diesem Zustand parallel drei verschiede Ziele:

- Speichern des Events im Event Store (Cosmos DB).
- Verarbeitung des Events im Event-Prozessor und damit Anschluss einer Folgeaufgabe.
- Aktualisieren des ReadModel, welches den aktuellen Zustand im Stream vorhält.

Mit der Parallelisierung einher geht auch die Herausforderung der Eventual Consistency (vgl. CAP-Theorem, [Wiki-b]). Das bedeutet, dass der „aktuelle Zustand" beziehungsweise das ReadModel beim Zugriff nicht zwingend konsistent und nicht zwingend aktuell ist, da noch unverarbeitete Ereignisse im Prozessor liegen könnten. Um dem zu begegnen und gleichzeitig die Datenkonsistenz zu wahren, mussten für jeden Prozessschritt Vorbedingungen festgelegt werden, an denen die aktuelle Datenkonsistenz identifiziert werden kann. So können Nachrichten, die auf noch nicht aktuellen

Pitfalls & Best Practices

Aus den Erfahrungen des Projekts ergeben sich folgende Learnings:

- Die Cloud-Umgebung ist stark veränderlich. Neue Funktionen, Dienste und Bibliotheken werden erstellt und aktualisiert oder ältere werden abgekündigt. Es lohnt sich immer, die Release Notes und einschlägigen Blogs zu lesen [Mic-c, Mic-d].
- Dokumentation ist vorhanden, aber für Spezialfälle teilweise sehr schwer auffindbar. Für neue Funktionalität versteckt sie sich häufig in Pull Request, GitHub Issues und GitHub-Dokumentationsrepositories. Manchmal wird Funktionalität genutzt, die erst seit wenigen Wochen existiert und noch keine offizielle Dokumentation hat.
- *Durbale Functions* mögen kein await-Pattern außer auf dem `DurableOrchestationContext` [Mic-a].
- Frühes Monitoring ist extrem wichtig. Application Insights liefert detaillierte Information über den aktuellen Zustand des Gesamtsystems. Dies funktioniert auch über mehrere Azure Function Apps hinweg out-of-the-box.
- Gegen Abstraktionen Interfaces implementieren, insbesondere dort, wo genutzte Azure-Komponenten(-Versionen) schnell austauschbar sein sollten. Teilweise auch zum Herstellen von testbarem Code. Ein Beispiel: Die Klasse `StorageAccount` aus dem Azure Storage SDK prüft direkt beim Anlegen mittels Konstruktor die intakte Verbindung. Im Produktionsbetrieb praktisch, für Testszenarien eher ungeeignet.
- Der Stapelverarbeitungsansatz (Batching) ist für einen guten Datendurchsatz bei moderaten Kostensteigerungen die beste Variante, auch wenn manche Services dies nur als Input oder Output unterstützen.
- Man muss bereit sein, Code oft zu aktualisieren. Azure Functions v1 sind bereits abgekündigt. Version v2 sollte bei Neuentwicklungen nicht mehr verwendet werden, sondern nur noch die aktuelle Version v3.
- Entwicklung an der Speerspitze der Frameworks: Oft bieten bereits releaste Features noch nicht einmal offizielle Dokumentation an und benötigte Funktionalität findet sich als offener Feature-Request für die jeweilige Komponente.
- Alte Architekturmuster loslassen. FaaS-Entwicklung ist stateless, serverless und erinnert eher an eine Kombination aus Webentwicklung und Microservice.
- Die sachlich und fachlich korrekte Auswahl geeigneter Cloud-Komponenten ist schwierig. Oft hilft es, etwas mehr Zeit als gewohnt in die Evaluierung von Diensten zu investieren, um sich vor Überraschungen zu schützen.
- Beispiele für die Verwendung von Cloud-Diensten sind häufig trivial und ohne Abhängigkeiten. Das erschwert die Adaption der Verwendung im komplexen Businessumfeld.
- Agile Methoden als neues Design-Muster für technische Entscheidungen anwenden: überprüfen, verbessern, verändern.
- Die Softwarearchitektur in der Cloud ist, ähnlich zum Vorgehensmodell, höchst agil und unterliegt stetigen Änderungen und Anpassungen. Und genau das ist gut so, denn damit gewinnt man die notwendige Flexibilität.

Daten basieren, auf die Queue zurückgelegt werden, um in einer weiteren Iteration erneut eine mögliche Abarbeitung zu prüfen.

Auslieferung von CI/CD/Infrastruktur

Was aber war jetzt mit der Infrastruktur? Bisher wurde darüber kein Wort verloren. Der Grund ist einfach: Sie muss auf jeden Fall auch in der Serverless-Welt bedacht werden, beansprucht aber bei Weitem nicht so viel Spezial-Know-how und Zeitressourcen wie herkömmliche Serverlandschaften.

Nach den ersten konkreten Entscheidungen für die zur Umsetzung zu verwendenden Komponenten stellt sich schnell die Frage, wie ein in der vorgesehenen Art verteiltes System bereitgestellt werden kann. Unabhängig von Quellcodeänderungen sollten Auslieferungen der Peripherie und Anwendungshüllen wiederholbar, nachvollziehbar, quellcodeverwaltet und transparent sein. Das kann man mit den Bordmitteln von Microsoft Azure mittels Azure Ressource Manager-Templates [ARM] erreichen. Diese setzten jedoch auf sehr komplexe JSON-Repräsentationen jeder Komponente ohne Standardkonfigurationen. Ein Vorteil ist dagegen, dass im Azure Portal manuell konfigurierte Komponenten als ARM-Template exportiert werden können.

Ein anderer Weg ist das Cloud-übergreifende Tool Terraform der Firma Hashicorp, welches letztlich aus folgenden Gründen auch benutzt wird:

- einfachere Handhabung und schnelle Ergebnisse,
- bereits vorhandenes Wissen bei EXXETA,
- mögliche Erweiterbarkeit durch OSS Contribution und Kombination mit ARM,
- potenzielle Ergänzung durch Dienste auf anderen Cloud-Plattformen.

Terraform nutzt eine eigene Notation und ein Plug-in-System für sehr viele unterschiedliche Provider. Für den bestehenden Anwendungsfall reicht die Verwendung des Providers *azurerm* aus. Terraform übernimmt die Parametrisierung von Namen und Einstellungen, sodass für jede Zielumgebung ein separates Setup geführt werden kann. Ein weiteres praktisches Feature ist die Unterstützung von Remote States. Terraform persistiert den letzten ausgelieferten Zustand in eine *tfstate*-Datei, um bei der nächsten Ausführung extern im Portal durchgeführte Änderungen zu erkennen und den Nutzer darüber zu informieren. Diese Information ist besonders nützlich, wenn Einstellungen im Portal manuell geändert wurden und diese später in die reguläre Auslieferung übernommen werden sollen. Standardmäßig wird die tfstate-Datei lokal abgelegt und mit Remote States können diese Informationen umgebungsspezifisch in einem Azure Blob Storage, Amazon S3 und vielen anderen Speichern abgelegt und verwaltet werden.

Nachdem die Infrastruktur bereitgestellt ist, erfolgt die Auslieferung der Azure Functions über automatisierte YAML-Pipelines in die verschiedenen Stages. Für den Betrieb und die Skalierbarkeit von Azure Functions ist ein *Azure App Service Plan* nötig. Dieser bietet gleichzeitig den Einstiegspunkt für die Konfiguration der Leistungsfähigkeit der Azure Function App. Im Plan kann je nach Auswahl zwischen unterschiedlich starken Maschinen entschieden und für diese eine automatische Skalierung (ab S1-Plan) konfiguriert werden. So werden bei Überschreiten einer definierten Metrik (z. B. CPU-Auslastung, Speicherverbrauch, Anfragemenge) weitere Instanzen hinzugeschaltet. Wichtig dabei ist, dass die Abschaltung nicht ausgelasteter Systeme ebenfalls konfiguriert werden muss. Für die Abrechnung in Azure ist jeweils die Arbeitszeit der Maschinen relevant, das heißt die Dauer der Auslastung der CPUs. Für Azure Functions, die über keine zeitlichen Trigger verfügen (z. B. nur http-Trigger), können sogar alle Server im Plan abgeschaltet werden und Azure führt selbstständig einen Kaltstart durch, sobald Anfragen eingeliefert werden.

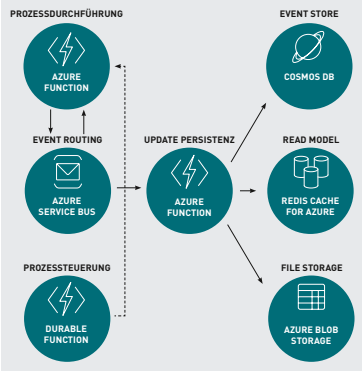


Abb. 4: Architekturansatz mittels Azure Service Bus

Neben der Wartung der Abhängigkeiten der verwalteten Dienste untereinander mittels Terraform, den Skalierungseinstellungen und einigen Sicherheitsmaßnahmen wie virtuelle Netze für IP-gesteuerte Zugriffe und Ähnliches erfordert die Infrastruktur keine größere Aufmerksamkeit.

Abschluss

Mit dem in Abbildung 4 dargestellten Architekturansatz geht der entwickelte Use-Case in Produktion und konsolidiert wie gefordert die hochgeladenen Stamm- und Bewegungsdaten. Weitere Anwendungsfälle sind vom Kunden bereits fest eingeplant und werden konzipiert. Es gibt eine klare Trennung zwischen Prozessdefinition (Durable Function) und Geschäftslogik (Azure Function) sowie der Datenhaltung (Cosmos DB, Blob Storage), was zuletzt durch modulare Architektur in Zukunft dazu beitragen wird, dass das System wachsen kann und diese Anwendungsfälle unkompliziert integriert werden können. Dabei werden zukünftig mithilfe von Azure Functions Schnittstellen entstehen, welche die Benutzung des Kernsystems vereinfachen und noch vorhandene Abhängigkeiten entkoppeln. Themen wie Eventregistrierung, konfigurierbare Projektionsbildung und die Anbindung weiterer Quellsysteme sind als kommende Schritte bereits geplant.

Sich in einem hochveränderlichen, providerzentrierten Cloud-System auf die Entwicklung komplexer Anwendungen einzulassen, bedeutet gleichzeitig, die Bereitschaft zu haben, sich häufigen externen Änderungen zu unterwerfen und mit diesen adäquat umzugehen. Bekannte Denkmuster müssen teilweise durchbrochen werden, um die Versprechen der Cloud auch umzusetzen. Ist man dazu bereit, so kann ein wirklich nutzenbringendes, zufriedenstellendes Wunderwerk erschaffen werden.

Links

[ARM] https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/
[Mic-a] Orchestration Functions Code Constraints, Microsoft, 2019, https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-code-constraints
[Mic-b] Azure Functions Best Practices, Microsoft, 2019, https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices
[Mic-c] Microsoft Azure Blog, https://azure.microsoft.com/en-us/blog/
[Mic-d] .NET Docs, https://github.com/dotnet/docs
[Wiki-a] https://de.wikipedia.org/wiki/Event_Sourcing
[Wiki-b] https://de.wikipedia.org/wiki/CAP-Theorem