BASTA!

Experten-Dossier 2019

Über 80 Seiten mit praxisorientiertem Wissen für .NET-Entwickler rund um .NET Core, Azure DevOps, TypeScript, Cosmos DB, ML.NET, Git und Azure!

bastacon www.basta.net

Inhalt

Agile & DevOps

Microservices & APIs

.NET Framework & C#

Architektur

Inhalt

DevOps-Fallstricke und wie man ihnen entkommen kann

Die DevOps-Challenge

Noch vor einigen Jahren wurde die IT in Unternehmen als notwendiges Übel wahrgenommen. Doch in Zeiten von digitalen Geschäftsmodellen und Produkten haben Unternehmen den wahren Wert dieses Bereichs erkannt. Das birgt jedoch eine Menge neuer Herausforderungen, denn der Wettbewerbsdruck steigt stetig und die Entwicklungszyklen werden immer kürzer. DevOps scheint das Allheilmittel zu sein, der Weg dorthin ist aber steinig.

von Kevin Gerndt

Das „Digital Mindset“ hat unlängst Einzug in große deutsche Konzerne gehalten. Das zeigen diverse Studien und dass die Notwendigkeit erkannt wurde, einen neuen Platz in der Führungsetage für den Chief Digital Officer (CDO) zu schaffen. Der CDO soll Unternehmen erfolgreich durch die digitale Transformation führen. Denn der Wettbewerbsdruck wächst zunehmend und immer mehr Firmen setzen auf die Entwicklung digitaler Produkte und Services. Wer nicht Opfer der Disruption werden möchte, muss sich diesem Trend anschließen und der Konkurrenz immer einen Schritt voraus sein. Interessanterweise beschränkt sich diese Entwicklung nicht nur auf einzelne Branchen, sondern ist nahezu in allen Marktsegmenten erkennbar. Wer hätte vor fünfzehn Jahren schon gedacht, dass Unternehmen aus dem Bereich Automotive einmal Unsummen in die Entwicklung und Bereitstellung digitaler Services investieren würden?

Ein gutes Beispiel hierfür ist der Car-Sharing-Dienst DriveNow des Automobilherstellers BMW. Durch das Car-Sharing-Angebot verlängert sich einerseits die Wertschöpfungskette hin zum Kunden, auf der anderen Seite lassen sich wertvolle Benutzerdaten sammeln. Wie lukrativ Servicemodelle sein können, zeigen Unternehmen wie myTaxi, HRS oder Uber. Diese Firmen sind weder im produzierenden Gewerbe tätig, noch besitzen sie physische Güter. Sie fungieren lediglich als Serviceprovider und bieten die Plattform zur Vermittlung von Dienstleistungen. Doch sind es oftmals diese Unternehmen, die durch hohe zweistellige Wachstumszahlen brillieren und schnell einen Milliardenwert auf die Börsenwaage bringen. Um ein solches produktgetriebenes Wachstum zu erreichen, ist es erforderlich, extrem schnell und agil auf Veränderungen und Anforderungen reagieren zu können. In dieser Disziplin sind die oft „tankerähnlichen“, in Silos organisierten Großunternehmen ganz klar unterlegen.

Letztendlich ist jedoch alles eine Frage der Strategie und des Willens. Im Jahr 2018 hat Klaus Straub, CIO bei BMW, angekündigt, das Unternehmen im Rahmen einer agilen Transformation ganz klar in Richtung Produktorientierung bzw. Agile Operating Model auszurichten. Unter dem Motto „100 % agil“ ist das erklärte Ziel, bis Ende 2019 nicht nur sämtliche laufende IT-Projekte des Unternehmens auf agil umzustellen, sondern auch die IT-Organisation vollumfänglich nach agilen Prinzipien auszurichten. Ein zwingendes Erfordernis stellt die enge Zusammenarbeit zwischen Fachbereichen und der IT-Abteilung dar.

Aber auch hinter den Mauern der IT ist ein enger Schulterschluss gefragt. Gemeint ist damit natürlich DevOps – die Verschmelzung von Entwicklung und Betrieb. Das Schaffen einer DevOps-Kultur ist jedoch viel mehr als nur das Vorhaben, Abteilungen und Bereiche zusammenzuführen. Es handelt sich vor allem um eine Revolution im Kopf aller Beteiligten, einen Umdenkprozess, der von höchster Ebene angeregt werden muss. Vor

allem ist es wichtig, die so häufig im Kopf existierenden Silos abzureißen. Doch was genau bedeutet das?

Stellt man sich einmal den klassischen Entwicklungsprozess vor, läuft es meist so, dass die Fachabteilung mit einem Anforderungskatalog an den Entwicklungsbereich herantritt. Dieser entwickelt die Software, um sie anschließend an den Betrieb zu übergeben. Das Betriebsteam wiederum stellt dann fest, dass die Software so nicht den Betriebsanforderungen entspricht oder vielleicht auch einfach nur essenzielle Artefakte wie etwa das Betriebshandbuch fehlen. Indes ärgert sich der Fachbereich über die mangelnde Kompetenz der IT und die Verzögerung bei der Inbetriebnahme. Ist die Software an den Betrieb übergeben, lehnt sich die Entwicklungsabteilung oftmals entspannt zurück, denn ihre Bringschuld ist erfüllt. Kommt es im Betrieb einer Applikation zu Problemen, geht das Fingerzeigen erneut los. Der Betrieb beschuldigt die Entwickler, die Entwickler den Betrieb. In großen Unternehmen können diese Extrarunden schon einmal Wochen oder gar Monate an Verzug bedeuten; ein Zeithorizont, der sich nicht mit einer agilen Strategie und der Verkürzung von Go-to-Market-Zyklen vereinbaren lässt.

In einem DevOps-Szenario übernimmt das Team, bestehend aus Produktmanagern, Entwicklern und Betriebsleuten, gemeinsam die Verantwortung für einen erfolgreichen Ablauf und Betrieb. Alle verfolgen dieselben Ziele und ziehen am gleichen Strang. Welche kritischen Erfolgsfaktoren neben der Kultur noch existieren und welche Tools und Plattformen bei der Einführung von DevOps hilfreich sein können, werden wir im Folgenden betrachten.

Die Säulen des Erfolgs

Auch wenn einige Unternehmen DevOps nur nutzen, um kurzfristige Effizienzvorteile zu erhalten, wird diese Methodik doch mehr und mehr zu einem unverzichtbaren Bestandteil für sämtliche Wirtschaftsbereiche. Zögerliche Unternehmen riskieren mittel- und langfristig ihre Wettbewerbsfähigkeit. Im schlimmsten Fall werden sie Opfer der Disruption und verschwinden vom Markt. Doch auch Unternehmen, die sich unlängst für eine DevOps-Strategie entschieden haben, kämpfen mit der Umsetzung. Unternehmen buhlen mit allen Mitteln um die Gunst der wertvollen Entwicklerressourcen. Keine Entwickler zu finden, ist für moderne Unternehmen existenzgefährdender als der fehlende Zugang zu frischem Kapital an den Finanzmärkten. Umso wichtiger ist es, eine solide und nachhaltige Basis für einen langfristigen Erfolg zu schaffen. Nachfolgend erläutere ich einige der Grundprinzipien von DevOps.

Kultur der Zusammenarbeit: Das A und O einer erfolgreichen DevOps-Strategie ist das Erlangen einer Kultur der Zusammenarbeit. In einer traditionell ausgerichteten IT verfolgen Entwicklung und Betrieb unterschiedliche Ziele mit unterschiedlichen Prioritäten. Die Entwicklung ist meist agiler und benötigt die Freiheit, Änderungen vornehmen zu können, der Betrieb hingegen schwört auf Stabilität. Eine enge Zusammenarbeit dieser Bereiche ist jedoch einer der wichtigsten Aspekte, wenn es um DevOps geht. Das Schaffen eines Informationsaustauschs in Echtzeit ermöglicht es den Teams, einerseits schnelle Änderungen an einer Applikation vornehmen zu können und andererseits eine stabile und robuste Betriebsumgebung zu erhalten. Tools wie etwa Slack, Yammer und Microsoft Teams können dazu verhelfen, die Hürden einer klassisch formalen E-Mail-Kommunikation abzubauen.

Einhalten von Architekturrichtlinien: Um die Betreibbarkeit, Stabilität und Kompatibilität einer Software zu sichern, ist es wichtig, Leitlinien zu definieren und diesen zu folgen. Enthält eine Applikation beispielsweise viele direkte Abhängigkeiten zu anderen Applikationen, Datenbanken oder anderen Systemen, kann das schnell zu einem hohen Maß an Komplexität führen. Das kann sich wiederum negativ auf Testaufwände, aber auch Key Performance Indicators (KPIs) wie die Deployment Success Rate auswirken. Probleme im Applikationsbetrieb müssen schnell identifiziert und behoben werden. Hierfür ist eine standardisierte Integration von Logging und Monitoring in die Applikation ebenfalls unabdingbar. Oftmals wird im Kontext dieser Design-for-DevOps-Best-Practices von zwölf Faktoren gesprochen, auf die im späteren Verlauf des Artikels eingegangen wird.

Infrastruktur und Plattform: Die Infrastruktur trägt einen maßgeblichen Teil zu einem stabilen Applikationsbetrieb bei und ist eine essenzielle Komponente in DevOps-Szenarien. Häufig kommen Plattformen wie OpenShift, Clound Foundry oder Azure zum Einsatz, die einen sehr hohen Grad der Automation erlauben und als gemeinsame Basis für das DevOps-Team fungieren. Die Entwickler können sich dadurch voll und ganz auf ihre Aufgabe – das Entwickeln – fokussieren. Die Betriebseinheit des Teams kann zum Beispiel die

Entwickler durch das Bereitstellen von Infrastrukturtemplates unterstützen.

Dass jede Applikation auf einer einheitlichen Plattform läuft, trägt zu einer allmählichen Effizienzsteigerung bei, da nicht bei jeder Inbetriebnahme einer Applikation erneut darüber diskutiert werden muss, wo sich die Logs befinden, wie Loglevel zu konfigurieren sind oder wie ein Rollback auf eine frühere Applikationsversion durchzuführen ist.

CI/CD Pipeline: Das Verwenden eines zentralen Code-Repositorys ist für Entwickler ein alter Hut und ein wesentlicher Bestandteil von DevOps. Alle Codeänderungen werden regelmäßig in ein zentrales Repository wie etwa Git zusammengeführt. Nach dem Check-in werden diese Änderungen automatisiert erstellt und getestet. Die Hauptziele von Continuous Integration bestehen darin, Bugs möglichst schnell zu entdecken und zu beheben, die Qualität der Software zu optimieren und den Zeitraum bis zum Go Live auf ein Minimum zu reduzieren. Das übergeordnete Ziel ist, die Entwicklerproduktivität zu steigern und „Stupid Work Tasks" zu eliminieren bzw. zu automatisieren. Das gelingt zum Beispiel durch Testautomatisierung; nach einem erfolgreichen Build-Prozess lassen sich beispielsweise Unit-, Komponenten-, UI-, Last- oder API-Tests durchführen. Die Ergebnisse fließen wiederum in Reports ein, mit denen sich der Erfolg des Teams messen lässt. Anhand der Resultate kann ebenfalls eine Entscheidungsgrundlage für z. B. ein automatisiertes Deployment geschaffen werden.

Eine hundertprozentige Testautomatisierung ist jedoch nicht realistisch. Die Kunst liegt darin, unter den Testtasks diejenigen zu identifizieren, die sich leicht automatisieren lassen und einen hohen Mehrwert für das Team schaffen. Erfahrungsgemäß lassen sich ca. 40–60 Prozent aller Testaufgaben automatisieren. Die Integration von Artefakten wie etwa Testautomatisierung bildet den Continuous-Delivery-Teil einer CI/CD Pipeline ab. Gelegentlich wird in diesem Zusammenhang auch von Continuous Deployment gesprochen. Diese zwei Varianten unterscheiden sich lediglich durch das Vorhandensein einer manuellen Genehmigung. Bei Continuous Delivery erfolgt das Deployment zunächst auf eine nicht produktive Test- oder Integrationsumgebung. Nach erfolgreichem Test und einer Freigabe wird das Release dann in die Produktionsumgebung gestaget. Bei Continuous Deployment wird auf diesen zusätzlichen Prüfschritt verzichtet und die Produktionsumgebung unmittelbar aktualisiert.

Erzeugen von Messbarkeit: Damit ein DevOps-Team maximal erfolgreich wird und sich die Erfolge auch belegen lassen, spielt das Erzeugen von Messbarkeit eine große Rolle. Dadurch, dass viele Prozesse zentral und automatisiert ablaufen, lassen sich jede Menge Daten erzeugen, die sich dann wiederum auswerten lassen und aus denen Optimierungspotenziale abgeleitet werden können. Da eine Vielzahl von KPIs existiert, ist eine projektindividuelle Festlegung durchaus sinnvoll, denn nicht jede KPI hat für jedes Projekt die gleiche Relevanz. Gute Beispiele für Metriken im Kontext von DevOps sind die Bereitstellungsfrequenz sowie die Bereitstellungszeit. Ein häufig definiertes Ziel ist es, möglichst viele, kleine Deployments durchzuführen. Dadurch sinken die Testaufwände für ein Release, und potenzielle Fehler lassen sich leicht eingrenzen.

Damit ein DevOps-Team maximal erfolgreich wird, und sich dieser Erfolg auch belegen lässt, spielt das Erzeugen von Messbarkeit eine große Rolle.

In diesem Zusammenhang spielt natürlich auch die für eine Bereitstellung benötigte Zeit eine große Rolle: Je kürzer die für ein Release benötigte Zeit, desto schneller lassen sich im Fehlerfall Korrekturen auf dem System einspielen. Die allgemeine Servicequalität kann zum Beispiel durch die Verfügbarkeit der Systeme und die Anzahl an Benutzertickets gemessen werden. Im Idealfall werden Störungen und Probleme behoben, bevor sie bis zum Benutzer vordringen, zum Beispiel durch ein intelligentes Application-Monitoring. Hierzu eignen sich Tools wie Retrace, Grafana oder App Dynamics, die wiederum eine Vielzahl von Parametern wie etwa Application-Performance, Bandbreiten oder Applikationsfehler überwachen. Neben den hier beispielhaft aufgeführten Metriken existiert natürlich noch eine ganze Reihe weiterer KPIs.

Um eine kontinuierliche Verbesserung der Werte zu erreichen, ist es empfehlenswert, einen regelmäßigen Austausch innerhalb des DevOps-Teams anzustreben. Das kann zum Beispiel in Form von Meetings erfolgen, bei denen die Beteiligten darüber sprechen, welche Faktoren sich ihrer Meinung nach besonders negativ auf einzelne KPIs oder den gesamten Service auswirken. Diese Punkte lassen sich dann wiederum gezielt verbessern oder gar eliminieren.

DevOps-kompatible Softwarearchitektur

Da es sich bei DevOps in erster Linie um eine Philosophie handelt und nicht um eine Technologie, ergeben sich vielleicht zunächst Fragen, inwieweit DevOps die zugrunde liegende Technologie beeinflusst oder sie gar voraussetzt. Das im Mittelpunkt stehende Ziel von DevOps ist meist die Erreichung maximaler Effizienz und

größtmöglicher Ökonomie. Um diese Ziele zu erreichen, bedarf es neben dem richtigen Mindset und einer DevOps-kompatiblen Organisation auch der Wahl eines fortschrittlichen Technologiestacks. Die Basis hierfür bildet in der Regel eine DevOps-Plattform wie etwa OpenShift, Cloud Foundry oder Azure DevOps. Diese Plattformen bieten integrierte Funktionen wie ein Containermanagement, die notwendige Containerlaufzeitumgebung, Lastenausgleichsmodule, Integration von Source-Code-Management und Build-Tools, CI/CD Pipelines sowie weitere nützliche Features.

Mit der Festlegung auf eine DevOps-Plattform wird also in gewisser Weise auch die Entscheidung für einen Basistechnologiestack gefällt, der in der Regel auch ein Containerkonzept mit sich bringt. Die aktuell wohl bekannteste Implementierung der Containertechnologie ist Docker, die aufgrund ihrer Einfachheit und Benutzerfreundlichkeit den Begriff überhaupt erst populär gemacht hat. Die Idee hinter Containern ist eigentlich recht einfach: Ein Container fasst eine einzelne Anwendung mitsamt allen notwendigen Abhängigkeiten wie etwa Bibliotheken oder statischen Inhalten in einer Image-Datei ohne Overhead zusammen. Im Gegensatz zu einer virtuellen Maschine enthält ein Container aber kein komplettes Betriebssystem und benötigt daher auch weniger Ressourcen. Ein bedeutender Vorteil der Docker-Container ist die gute Skalierbarkeit: Werden aufgrund des Lastverhaltens zusätzliche Instanzen einer Anwendung benötigt, lassen sich nach Belieben neue Container auf Basis eines Images erzeugen. Nach dem Stoppen sind die Container wieder vollständig aus dem System verschwunden und alle Ressourcen werden freigegeben. Die Konfiguration ist so weit wie möglich bereits im Container-Image eingerichtet. Zusätzlich notwendige Anpassungen sollten skriptbasiert erfolgen.

Um das volle Potenzial der Containertechnologie auszunutzen, müssen die darin ausgeführten Applikationen verschiedene Architekturrichtlinien berücksichtigen. In diesem Kontext wird häufig von 12-Faktor-Applikationen gesprochen [1]. Hinter diesem Konzept verbergen sich zwölf Punkte, die Entwicklern dabei helfen sollen, Applikationen so zu designen und zu entwickeln, dass sich diese als SaaS (Software as a Service) und letztendlich auch problemlos in Containern betreiben lassen. Ein wichtiges und zugleich selbstverständliches Kriterium der zwölf Faktoren ist, dass der Quellcode einer Applikation stets in einer Versionsverwaltung wie zum Beispiel Git verwaltet wird. Dabei besteht immer eine eindeutige Korrelation zwischen einer Codebasis und einer App. Zwei Applikationen können sich nach diesem Konzept also einen Quellcode teilen. Das Vorhalten des Quellcodes ist ebenfalls ein wesentliches Erfordernis für den Aufbau einer CI/CD Pipeline. Applikationen sollten keine direkten Abhängigkeiten untereinander aufweisen. Durch das Verwenden eines Package Managers lässt sich sicherstellen, dass Abhängigkeiten zentral aufgelöst werden. Auch Problemen, die durch eine Diskrepanz in den Versionen entstehen, lässt sich so vorbeugen. Das gleiche Paradigma gilt für Services oder Ressourcen, die von einer Applikation verwendet werden. Dazu zählen zum Beispiel Datenbank- oder SMTP-Dienste sowie APIs. Eine App muss stets so entwickelt werden, dass es für die Einbindung und Nutzung der Services und Ressourcen unerheblich ist, von welchem Anbieter sie zur Verfügung gestellt werden. Im Rahmen eines Bereitstellungsprozesses ist es üblich, dass pro Umgebung eine eindeutige Konfiguration festgelegt wird. Diese Konfigurationsparameter müssen sich zwingend von außerhalb der Applikation ändern lassen. Ein Hinterlegen dieser Konfiguration im Code durch Konstanten, zum Beispiel für Systemzugangsdaten oder Datenbankverbindungsfolgen, stellt einen klaren Verstoß in einer 12-Faktor-Architektur dar.

In einer 12-Faktor-App wird strikt zwischen Build-, Release- und Run-Phase unterschieden. Checkt der Entwickler neuen Code ins Repository ein, wird mit der Build-Phase die Transformation des Codes in ein ausführbares Paket angestoßen. Im nächsten Schritt, der Releasephase, wird das Build-Ergebnis mit der zum Deployment passenden Konfiguration kombiniert. In der Run-Phase, der sogenannten Laufzeit, wird die Applikation dann ausgeführt. Aufgrund dieser strikten Trennung sind Codeänderungen zur Laufzeit nicht möglich, weil es keinen Weg gibt, die Änderungen zurück in die Build-Phase zu übernehmen. Es empfiehlt sich, jedes Release separat mit einem eindeutigen Identifier und Erstelldatum und -uhrzeit abzulegen. Das ermöglicht im Fehlerfall ein schnelles Rollback der Lösung. Eine 12-Faktor-App sollte immer zustandslos sein. Alle Daten werden in unterstützenden Diensten, in der Regel einer Datenbank, gespeichert. Der RAM oder das lokale Dateisystem können als kurzzeitiger Cache für eine Transaktion verwendet werden. Es darf aber niemals davon ausgegangen werden, dass Daten für einen längeren Zeitraum vorgehalten werden oder durch eine

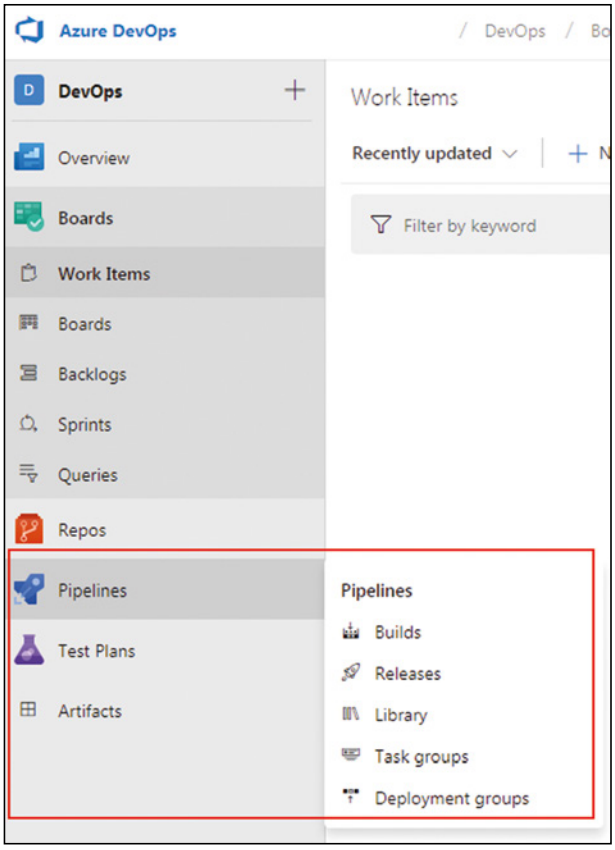


Abb. 1: Azure DevOps Pipelines

andere Transaktion nutzbar sind. In einer skalierbaren Umgebung ist es ebenso denkbar, dass ein künftiger Request von einer anderen Instanz als der ursprünglich angefragten und somit in einem anderen Transaktions-Scope bearbeitet wird. Da diese Flexibilität ein wesentlicher Teil der Grundidee ist, müssen Prozesse so ausgelegt werden, dass sie eine möglichst geringe Startzeit haben. Im Idealfall ist ein Prozess innerhalb weniger Sekunden hochgefahren und einsatzfähig. Eine kurze Start-up Time verleiht dem System noch mehr Agilität und Robustheit. Gleiches gilt natürlich für das Herunterfahren eines Prozesses.

Um das Verhalten einer laufenden Applikation sichtbar zu machen, ist Logging unabdingbar. Häufig werden Logs gezielt an einem bestimmten Ort, wie dem Dateisystem oder einer Datenbank, abgelegt. Dieses Verhalten wird oftmals durch die Applikation bestimmt. Im Fall einer 12-Faktor-App ist es nicht mehr Aufgabe der Applikation, sich um einen Ablageort zu kümmern. Stattdessen werden sämtliche Ereignisse ungepuffert in den *stdout*-Stream geschrieben. Die weitere Verarbeitung dieser Ereignisinformationen obliegt dem Hostsystem. Es entscheidet alleinig, ob und wohin die Logs persistiert werden. Das Einhalten der hier aufgezeigten Leitlinien trägt zu einer sauberen und skalierbaren Applikationsarchitektur bei. Auch wenn dies prinzipiell für jede Applikation gilt, sollte dem Thema im Kontext von DevOps eine besondere Bedeutung zugemessen werden.

Azure DevOps

Die Zeichen, die Microsoft aussendet, sind deutlich: Schon länger versucht der Konzern, ein Open-Source-freundlicheres Image aufzubauen. Ein großer Schritt in diese Richtung war der Beitritt Microsofts zum Open Innovation Network (OIN) und der damit verbundenen Bereitstellung von 60 000 eigenen Patenten. Ein weiterer bedeutender Schritt ist das Aufgeben der Marke „Visual Studio“. Die Cloud-Dienste für Entwickler, ehemals Visual Studio Team Services (VSTS), laufen ab sofort unter dem Brand „Azure DevOps“. Auch das lokal installierbare Gegenstück zu den VSTS, das seit 2005 unter dem Namen „Team Foundation Server“ (TFS) auf dem Markt ist, erhält ab der Version 2019 einen Brand: „Azure DevOps Server 2019“. Dieses Rebranding soll vor allem dafür sorgen, die Assoziation in den Köpfen vieler Entwickler zwischen Visual Studio und den klassischerweise damit verbundenen Programmiersprachen C++, C#, F# etc. zu lösen. Neben dem neuen Markennamen spendiert Microsoft natürlich auch noch ein paar neue Features wie etwa die „Azure Pipelines“. Hierbei handelt es sich um eine leistungsstarke CI/CD Pipeline Engine, die mit nahezu jeder gängigen Programmiersprache und einer Vielzahl von Projekttypen arbeitet. Natürlich gab es auch in der altbekannten Version bereits eine CI/CD Pipeline, doch vor dem Hintergrund immer größer werdenden Leistungsdrucks stellt die CI/CD Pipeline eine Komponente dar, die sicherlich noch viel Potenzial für Optimierung bietet. Azure Pipelines bieten die Möglichkeit, verschiedenste Zielsysteme für Bereitstellungsszenarien zu adressieren. Dazu gehören neben den Azure-eigenen Services, wie „Azure App Services“ oder „Azure Kubernetes Service“ (AKS), auch virtuelle Maschinen, klassische Container Registries oder Deployments zu Amazon Web Services. Eine neue Pipeline kann einfach über den Menüpunkt Pipelines auf der linken Seite der Azure-DevOps-Weboberfläche erstellt werden (**Abb. 1**).

Für das Erstellen einer Build Pipeline ist es zunächst erforderlich, eine Quelle auszuwählen, von der aus der Source Code abgerufen werden soll. Kompatible Quellcodeverwaltungssysteme sind zum Beispiel GitHub, Azure Repos Git, Subversion etc. Anschließend erfolgt die eigentliche Build-Konfiguration, die aber nicht jedes Mal aufs Neue manuell durchgeführt werden muss. Stattdessen kann der Benutzer aus einem Templatekatalog auswählen. So ist es mittels weniger Klicks beispielsweise möglich, eine Build Pipeline für eine Azure-Web-App for ASP.NET zu konfigurieren. Sind Container das präferierte Zielmodell, ist eine direkte Containerisierung mit anschließender Bereitstellung in einer Container-Registry ebenfalls mit einem überschaubaren Aufwand realisierbar. Wer sich in der Welt von YAML, einem JSON-ähnlichen deklarativen Format, wohlfühlt, ist eingeladen, die Build-Konfiguration auf Basis dieses Formats festzulegen. Das Erstellen einer Release-Pipeline fühlt sich ähnlich komfortabel an. End to

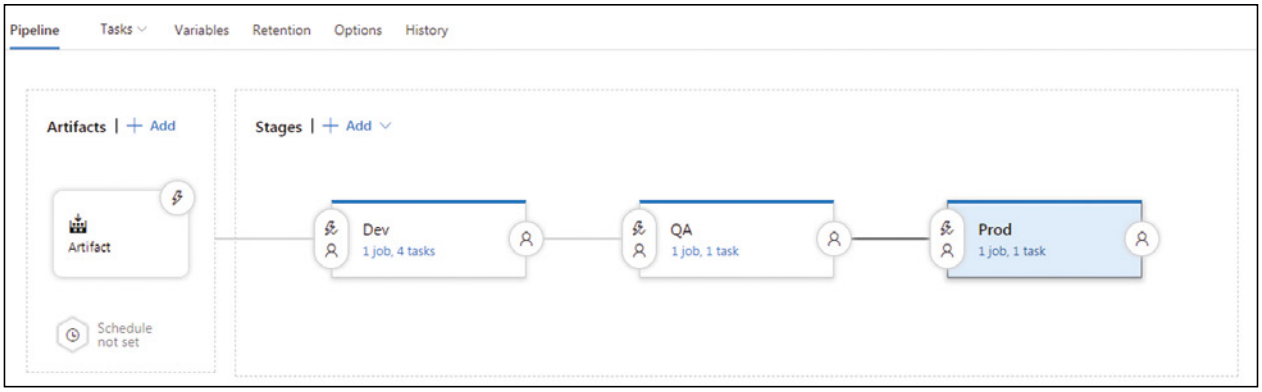


Abb. 2: Konfiguration einer Release-Pipeline

end gedacht, dient die Release-Pipeline dazu, den Code für die Benutzer auf einem System verfügbar zu machen. Ihre Konfiguration erfolgt mittels eines grafischen Editors (**Abb. 2**).

Als Ausgangsbasis dient ein sogenanntes „Artifact". Hierbei handelt es sich vereinfacht gesagt um den bereitstellbaren Teil einer Applikation, der in der Regel durch die Build Pipeline erzeugt wird. Am Artefakt selbst lässt sich wiederum das Verhalten über sogenannte Trigger festlegen. Der Bereitstellungsprozess wird somit zum Beispiel automatisch gestartet, sobald ein neuer Build zur Verfügung steht. Auch ein manueller Eingriff mittels Pull Request lässt sich einstellen. Über die Stages wird definiert, welche Umgebung zu welchem Zeitpunkt mit welchen Voraussetzungen deployt wird. Um den Konfigurationsprozess so einfach wie möglich zu gestalten, existieren auch hierfür zahlreiche vorgefertigte Templates für eine Bereitstellung auf Azure, Kubernetes, IIS etc. Das Deployment einer Stage kann an verschiedene Pre- und Post-Conditions geknüpft werden. Hierbei kann es sich beispielsweise um ein automatisch ausgelöstes Event oder eine benutzergetriebene Aktion handeln. Gates ermöglichen eine Integration von REST APIs, Azure Monitor Alerts oder Azure Functions. Sollte beim Deployment doch mal etwas schieflaufen, erlaubt die History eine transparente Einsicht in Änderungen sowie ein kontrolliertes Zurückrollen des Release.

Fazit

In Zeiten zunehmenden Wettbewerbsdrucks ist es wichtiger denn je, der Konkurrenz immer einen Schritt voraus zu sein. Dies trifft auf die Entwicklung physischer Güter, aber vor allem immer stärker auf digitale Services und Produkte zu. Mit der klassischen Unternehmensorganisation fällt es jedoch vor allem großen Konzernen mit ihrer Silostruktur schwer, mit dem vorherrschenden Tempo mitzuhalten. Doch immer mehr Unternehmen passen sich an und vollziehen einen Strukturwandel oder gründen Start-ups aus, um der heutzutage geforderten Agilität gerecht zu werden. Ein gutes Beispiel ist der Automobilkonzern BMW, dessen Vorstand einen radikalen Wandel in Richtung „Agile Operating Model" bis Ende 2019 angekündigt hat. Das Zauberwort für das Erreichen all dieser Ziele lautet DevOps. Hierzu gehört jedoch viel mehr als nur das Einführen neuer Technologien und Plattformen. Es gehört ebenfalls mehr dazu als nur das Vorhaben, Abteilungen und Bereiche zusammenzuführen. Es handelt sich erst einmal um eine Revolution im Kopf aller Beteiligten, einen Umdenkprozess, der von höchster Ebene angeregt werden muss. Dabei ist es vor allem wichtig, die häufig im Kopf existierenden Silos abzureißen. Auch wenn bei DevOps die Technik nicht im Vordergrund steht, bedarf es natürlich entsprechender Konzepte und einheitlicher Plattformen, um die größtmögliche Effizienz bei der Entwicklung zu erzielen. Finden all diese Punkte Berücksichtigung und gelingt der Schulterschluss zwischen Entwicklung, Betrieb und Fachbereichen, steht einer erfolgreichen DevOps-Organisation nichts mehr im Weg.

Kevin Gerndt arbeitet als Lead Consultant im Bereich Microsoft .NET Client und Web Technologies und hat während seiner beruflichen Laufbahn schon eine Vielzahl an Projekten begleitet. Er gilt als Experte auf dem Gebiet der modernen Web- und Softwarearchitektur und ist darüber hinaus als freiberuflicher Autor tätig, sowie regelmäßig auf namhaften Konferenzen vertreten. Schwerpunkte seiner Tätigkeit bilden die Analyse und Implementation von Geschäftsprozessen sowie das Konzipieren und Entwickeln moderner Softwarelösungen.

Links & Literatur

[1] https://12factor.net

Entwicklung von Microservices mit Microsoft .NET

Warum einfach? Es geht auch komplex!

Wie zu erwarten zieht das heißdiskutierte Thema Microservices auch an Microsoft nicht vorüber. Grund genug, einmal näher zu betrachten, wie man als .NET-Entwickler bei der Entwicklung von Microservices mit Docker von technischer Seite unterstützt wird.

von Dr. Felix Nendzig

Mit „.NET Microservices: Architecture for Containerized .NET Applications“ hat Microsoft nun ein mehr als dreihundertseitiges E-Book veröffentlicht, in dem das Unternehmen seine Sicht auf das Thema umfassend vorstellt und insbesondere die Benutzung von Docker bei der Entwicklung von Anwendungen mit Microservices-Architektur empfiehlt [1].

In diesem Beitrag soll Microsofts Sicht auf das Thema Microservices näher beleuchtet und mit Blick auf unsere eigene Erfahrung kommentiert werden. Weiterhin werden wir darauf eingehen, wie Docker bei der Entwicklung von Microservices genutzt werden kann.

Was ist eine Microservices-Architektur?

Das zentrale Erkennungsmerkmal einer Microservices-Architektur ist, dass sich die gesamte Anwendung aus einzelnen, voneinander unabhängigen Services zusammensetzt. Durch Dezentralisierung und Autonomiemaximierung soll auch bei komplexen Anwendungen eine gute Skalierbarkeit in der Entwicklung – gegenüber einer monolithischen Architektur – erreicht werden. Die Entwicklung kann in kleinen, unabhängigen Teams mit geringem Kommunikationsbedarf untereinander stattfinden. Um einen Vorteil gegenüber einer monolithischen Anwendung zu erreichen, sollten die Microservices jeweils folgende Bedingungen erfüllen:

- Ein Service erfüllt genau einen fachlichen Zweck und diesen vollständig
- Jeder Service umfasst sämtliche benötigte Schichten, Teams arbeiten unabhängig voneinander (vertikaler Schnitt)
- Asynchrone Kommunikation zwischen Services (keine zentral steuernde Einheit)
- Keine gemeinsamen Daten oder Code (shared Nothing)
- Jeder Service ist unabhängig deploybar
- Unabhängige Datenmodelle und minimale APIs zur Gewährleistung eigenständiger Entwicklungszyklen

Der Name „Microservices“ impliziert, dass die einzelnen Services klein sein sollen. Diese Charakterisierung bezieht sich aber nicht auf die Lines of Code oder die Anzahl der Klassen je Service. Stattdessen

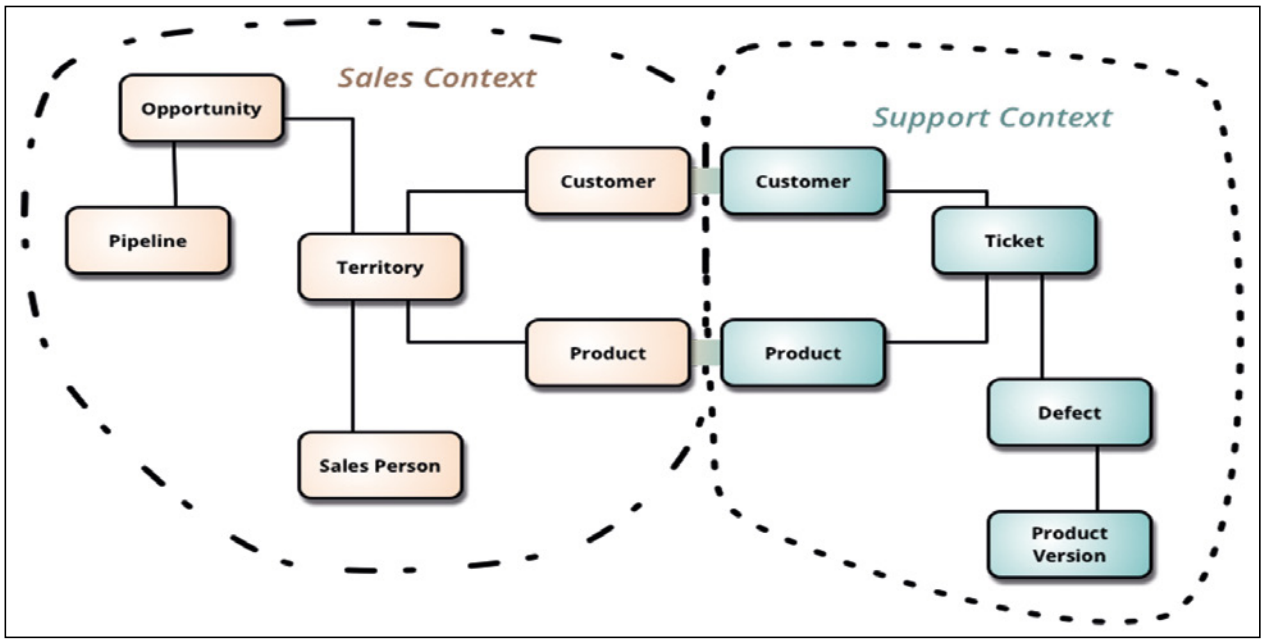


Abb. 1: Illustration zweier Bounded Contexts (Quelle: [2])

bedeutet es, dass ein Service nur genau eine sinnvolle Funktion im Sinne der fachlichen Logik erfüllen soll. Diese Idee folgt dem Konzept des Bounded Context aus dem Domain-driven Design [2]. Danach wird ein großer fachlicher Themenbereich in kleinere Bereiche, die Bounded Contexts (**Abb. 1**), aufgeteilt, die jeweils ihre eigenen, eindeutigen Fachbegriffe benutzen (Ubiquitous Language). Ein Microservice überschreitet keinesfalls die Grenzen eines Bounded Contexts. Er implementiert maximal den gesamten Bounded Context, typischerweise aber nur einen Teilbereich.

Man sollte die kleinsten Microservices implementieren, die die oben genannten Bedingung erfüllen. Wird die optimale Größe unterschritten, wachsen die Abhängigkeiten der Services untereinander. Das verschlechtert die Performance der Anwendung zur Laufzeit und während der Entwicklung durch erhöhten Kommunikationsbedarf über Service- und Teamgrenzen hinweg. Der Schnitt der Microservices sollte deren innere Kohäsion maximieren und die Abhängigkeiten nach außen minimieren. Gemäß dem Gesetz von Conway [3] sollten sich die Service-Grenzen hierbei an der Organisationsstruktur des Unternehmens orientieren. Es kann sich auch lohnen, dieses Prinzip umzukehren, und die Organisation so anzupassen, dass man den aus Softwaresicht sinnvollsten Service-Schnitt wählen kann.

Bei richtiger Umsetzung erhält man ein skalierbares System lose gekoppelter Services, die unabhängig voneinander entwickelt und ausgeliefert werden können. Releasezyklus, Entwicklungs- und Betriebsumgebung können je Service passend gewählt werden. Der Testaufwand je Service sinkt, da er nicht übermäßig komplex ist und nur über Schnittstellen kommuniziert. Die Entwicklung in kleinen Teams erzeugt nur geringen organisatorischen Aufwand. Ausgehend von Amazons „2-Pizza-Regel“ liegt die maximale Teamgröße bei fünf bis neun Personen [4].

Eine verteilte Anwendung bringt allerdings auch zusätzliche Schwierigkeiten mit sich, die bei falscher Vorgehensweise zu einer Verschlechterung gegenüber einem Monolith führen können: Es gibt keine zentrale Steuerung, sodass querschnittliche Aufgaben wie Service-übergreifendes Monitoring oder Logging zusätzlichen Entwicklungsaufwand mit sich bringen. Externe Tools wie der ELK-Stack [5], Zipkin [6], Jaeger [7] und AppDynamics [8] können hier Abhilfe schaffen. Die Durchführung von Service-übergreifenden Integrationstests oder Refactorings wird schwieriger. Transaktionen sind auf einzelne Microservices begrenzt. Das Prinzip des „shared Nothing“ erzwingt, dass jeder Service seine benötigten Daten replizieren oder über Schnittstellen von anderen Services abfragen muss. Jedes Entwicklerteam hat zusätzlichen Aufwand durch erschwertes Debugging, eine eigene Fehlerbehandlung je Microservice, die Bereitstellung einer Delivery Pipeline etc. Ein ungünstiger Service-Schnitt führt zu ständigen Schnittstellenanpassungen, gemeinsamen Releases mehrerer Services oder der Auslastung mehrerer Services durch eine einzige Anfrage.

Stateful oder stateless Microservices?

In einer verteilten Anwendung kann eine variierende Anzahl von Service-Instanzen simultan laufen. Der Orchestrator (mehr dazu etwas später) kann jederzeit Instanzen auf einen anderen Knoten schieben oder die Anzahl laufender Instanzen an die momentane Auslastung anpassen. Man kann sich nicht darauf verlassen, dass eine spezielle Service-Instanz dauerhaft existiert. Das spricht gegen die Implementierung von stateful Services, deren Zustand im Arbeitsspeicher gehalten und jederzeit repliziert werden können muss. Für stateless Services stellt das kein Problem

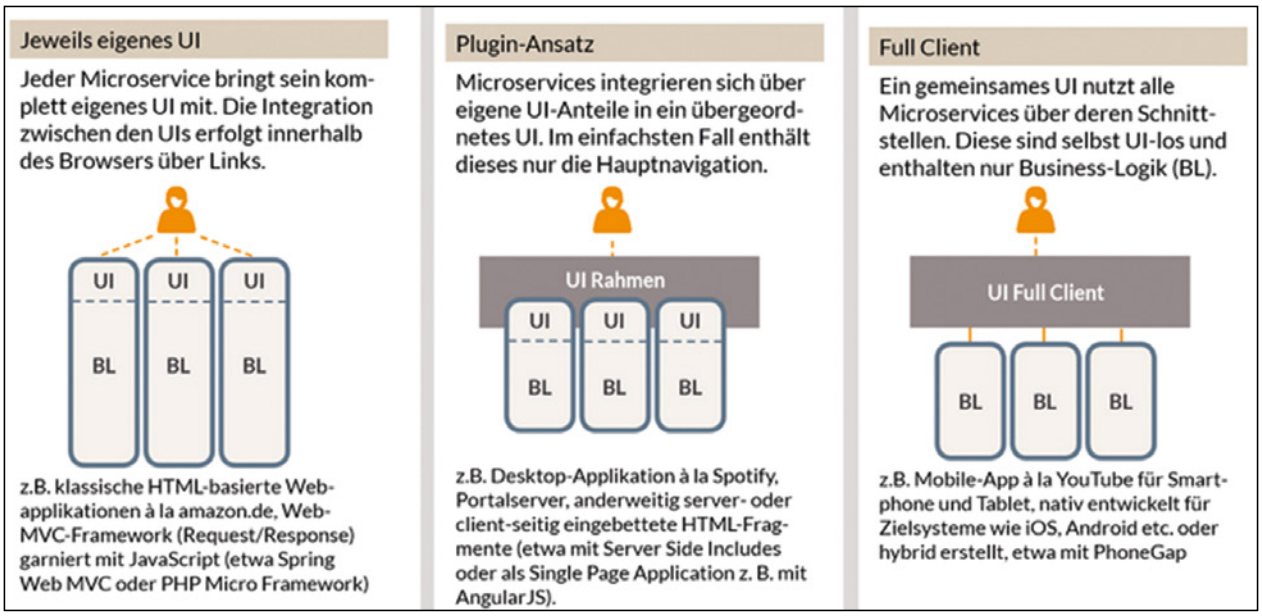


Abb. 2: Möglichkeiten zur Strukturierung des GUI in einer Microservices-Architektur (Quelle: [10])

dar, da sie die Daten in eine externe Datenbank schreiben oder im Client oder einem Cachetool wie bspw. Redis cachen. Die externe Datenquelle wird je Service passend zu dessen Anforderungen gewählt.

Was ist aber mit Anwendungsfällen, in denen man Daten von verschiedenen Microservices sammeln oder anzeigen muss? Tritt ein solcher Fall auf, sollte man sich gut überlegen, ob der Service-Schnitt passend gewählt wurde und ob nicht eine Verschmelzung von Microservices die beste Lösung darstellt. Lässt es sich jedoch nicht vermeiden, muss man bei der Implementierung besonders darauf achten, dass die Autonomie der betroffen Microservices dadurch nicht eingeschränkt wird. Zudem kann sich die Aggregation von Daten verschiedener Services aufgrund erhöhter Kommunikation negativ auf die Performance des Systems auswirken.

Microsoft nennt als Alternative noch eine auf den Anwendungsfall zugeschnittene, denormalisierte Tabelle in einer separaten Datenbank, in die die Microservices jeweils ihre Daten schreiben. Aus Konsistenzgründen dürfen die abfragenden Services auf diese nur lesend zugreifen. Mit Blick auf die geforderte Autonomie ist diese Lösung allerdings kritisch zu sehen.

Inter-Service-Kommunikation

Wie jede verteilte Anwendung muss auch eine Microservices-Anwendung mit Teilausfällen des Systems umgehen können. Wie gut die Gesamtanwendung solche Ausfälle verkraftet, hängt maßgeblich von der Inter-Service-Kommunikation ab. Synchrone Kommunikation, bei der der Aufrufer abwarten muss, bis eine Antwort zurückgeliefert wird, kann bei lesendem Zugriff gegebenenfalls eine Lösung für die Kommunikation zwischen Front- und Backend sein. Innerhalb des Backends ist sie allerdings zu vermeiden, da sich Verzögerungen und Ausfälle des angefragten Service direkt auf aufrufende Services auswirken. Auf diese Weise wird schnell die Performance der Gesamtanwendung beeinträchtigt. Das Problem potenziert sich noch, wenn mehrere synchrone Aufrufe in Reihe stattfinden. Bei asynchroner Kommunikation wirkt sich der Ausfall eines einzelnen Microservice dagegen nicht so verheerend auf seine Nachbarn aus. Sie fördert zudem die Autonomie der einzelnen Microservices.

Da an der Verarbeitung eines Geschäftsprozesses im Allgemeinen mehrere asynchron kommunizierende, unabhängige Microservices beteiligt sind, ist es nicht möglich, stets Konsistenz im Sinne einer ACID-Transaktion sicherzustellen. Latenzen, Teilausfälle und unterschiedliche Verarbeitungsdauern können zwischenzeitlich Service-übergreifende, inkonsistente Zustände erzeugen (Eventual Consistency), deren Behandlung zusätzlichen Aufwand, z. B. in Form von Timeouts, Circuit Breakers und Bulkheads, erfordert [9].

Das GUI

Sollten Clients direkt mit den Microservices kommunizieren oder den Umweg über ein vorgeschaltetes API-Gateway nehmen? In der ersten Variante kann der Client über einen URL direkt den betreffenden Service (bzw. einen vorgeschalteten Load Balancer) anfragen. Jeder Microservice besitzt sein eigenes GUI oder ist in einem Plug-in-Ansatz für einen Teil des GUI innerhalb eines allgemeinen Rahmens verantwortlich.

Für kleine Anwendungen mit geringem Aufwand in der GUI-Entwicklung kann diese Variante ausreichen. Man bekommt jedoch schnell Probleme, wenn die Entwicklung der Benutzeroberfläche in mehreren Microservices gleichzeitig stattfindet und zudem noch für verschiedene Endgeräte entwickelt werden muss. Sind zum Aufbau einer Oberfläche mehrere unabhängige Abfragen nötig, erhöht das die Latenz. Ein besseres Ergebnis erhält man, wenn die Daten zuvor serverseitig

Ein Orchestrator ermöglicht eine einfache Steuerung der Anwendung inklusive Scheduling, Load Balancing und Gewährleistung von Verfügbarkeit und Robustheit.

aggregiert werden. Auch querschnittliche Anforderungen wie Sicherheit, Autorisierung, Logging und Caching möchte man nicht individuell für jeden Microservice, sondern gerne zentral implementieren.

Schaltet man ein API-Gateway zwischen die Clients und die Microservices-Landschaft, kann es das Routing sowie weitere querschnittliche Aufgaben übernehmen. Der Aufbau einer Oberfläche erfordert keine mehrfachen Roundtrips und Sicherheitsthemen können zentral implementiert und verwaltet werden. Die Weiterentwicklung der Anwendung wird erleichtert, da die Clients nicht mehr über die Existenz bzw. Aufteilung der Microservices Bescheid wissen müssen.

Man sollte jedoch nicht einfach ein einzelnes „monolithisches" Gateway implementieren, sondern muss auch hier sowohl die Separation nach Geschäftsprozessen als auch nach Client-Apps einhalten (**Abb. 2**). Anderenfalls erzeugt man ungewünschte Kopplungen zwischen den Microservices.

Im Falle eines Web-UI mit dem Browser als Integrationskomponente, dürfte der direkte Ansatz am einfachsten umzusetzen sein. Die Entwicklung nativer Apps für diverse Endgeräte erzwingt dagegen den API-Gateway-Ansatz, bei der die Microservices im Backend einer übergeordneten Präsentationsschicht arbeiten (Backend for Frontend). Dieses Muster ist auch dann zu bevorzugen, wenn man besonders viel ausgeklügelte Logik in der Oberfläche unterbringen will. Zwar sind die Microservices ohne eigenes GUI nicht mehr unbedingt als vollständige Anwendungen zu betrachten, dennoch stellt das API-Gateway-Pattern keine Verletzung der Microservices-Philosophie dar. Die Unabhängigkeit der Services untereinander, einschließlich ihrer Datenhoheit, bleibt bei richtiger Umsetzung erhalten [11].

Entwicklung mit Visual Studio und Docker

Microsoft empfiehlt, die Vorteile von Docker bei Entwicklung, Deployment und Betrieb von Anwendungen mit Microservices-Architektur zu nutzen. Für jeden Microservice wird ein eigenes Docker Image erstellt, sodass zur Laufzeit schnell beliebig viele Instanzen in Form von Docker-Containern erstellt werden können. Für diesen Ansatz mit allen damit verbundenen Fragestellungen existiert eine umfangreiche Dokumentation [1], [12]-[15]. Docker-Container passen gut in das Konzept der Microservices, da sie unabhängiges Deployment und Skalieren unterstützen.

Microsoft bietet in Visual Studio und Visual Studio Code integrierte Unterstützung für die Entwicklung von Anwendungen, die in Docker-Containern laufen sollen. Unterstützt wird die Entwicklung mit .NET Framework, .NET Core und Mono in den Sprachen C#, F# und VB. Mit .NET Core entwickelte Microservices sind auf unterschiedlichen Plattformen lauffähig. Das modulare .NET Core ist zudem sehr leichtgewichtig und damit besser geeignet, containerbasierte Anwendungen mit möglichst kleinen Microservices zu entwickeln. Das klassische .NET Framework ist vorzuziehen, wenn die bereits bestehende Anwendung oder andere Abhängigkeiten das erfordern. Darunter würden zum Beispiel dringend benötigte Pakete oder Technologien wie ASP.NET Web Forms oder WCF fallen, die nicht in .NET Core verfügbar sind. Ab Visual Studio 2017 ist die Unterstützung für Docker bereits standardmäßig enthalten. Für ältere Versionen können die Docker Tools nachinstalliert werden [16].

Beim Anlegen eines neuen Projekts mit Docker Support wird ein passendes Dockerfile erzeugt. Zusätzlich muss man noch „Container Orchestrator Support" aktivieren, um für das Projekt ein Docker Compose anzulegen bzw. zu aktualisieren. Es enthält mit der Datei *docker-compose.yml* die Konfigurationsdaten der Container und Deployment-Umgebung. So kann später für jeden Service ein separates Docker Image erstellt werden. Der Entwicklungsworkflow in Visual Studio bleibt sonst größtenteils unverändert. Beim

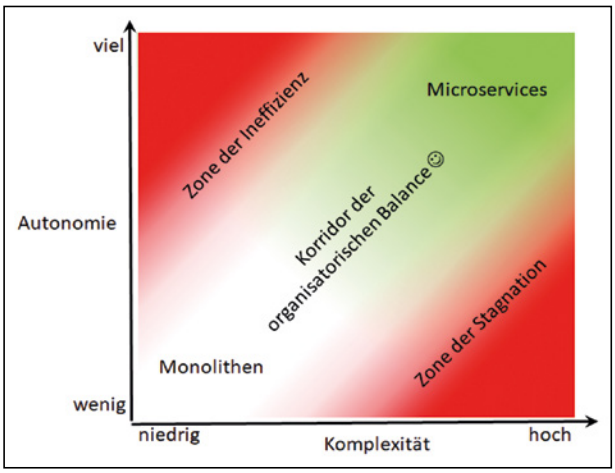


Abb. 3: Auswirkung von Autonomie und Komplexität bei der Entwicklung von Monolithen und Microservices (Quelle: [26])

Starten der Anwendung werden die Docker Images automatisch gebaut und direkt in Docker gestartet. Auch eine Multicontaineranwendung kann in Visual Studio debuggt werden. In Microsofts Visual-Studio-Dokumentation [17] wird erläutert, wie man eine Containeranwendung direkt in die Azure Container Registry deployen kann. Das Deployen und Testen sollte so früh wie möglich in Docker geschehen, da so die Produktivumgebung exakt reproduziert werden kann.

Um das System skalierbar zu machen, muss man eine Vielzahl Container gebündelt als Cluster managen und je nach Last die Anzahl der Service-Instanzen automatisch anpassen. Das wird erst durch einen Orchestrator ermöglicht, der die Komplexität eines Containerclusters abstrahiert und so das Management einer Multicontaineranwendung ermöglicht. Er erlaubt eine einfache Steuerung der Anwendung inklusive Scheduling, Load Balancing und Gewährleistung von Verfügbarkeit und Robustheit. Plattformen, die als Orchestrator agieren können, sind beispielsweise Azure Service Fabric, Kubernetes, Docker Swarm und Mesosphere DC/OS [18]-[22].

Wann verwendet man Microservices?

So lange die Prozesse eines Unternehmens oder eines Geschäftsbereichs durch eine Anwendung mit monolithischer Architektur abgebildet werden können, besteht kein Handlungsbedarf. Auch wenn die Anwendung schnell auf schwankende Last reagieren können muss, ist es nicht unbedingt nötig, den Flaschenhals im System als eigenen Service auszulagern. Nutzt man Docker, kann man einfach den gesamten Monolith in einem Container deployen und bei Bedarf zusätzliche Instanzen hochfahren. Wird das System aber zu komplex und steigt der Organisationsaufwand, dann wird es zunehmend schwerer, die Weiterentwickelung voranzutreiben, dabei das System wartbar zu halten und den Qualitätsanforderungen gerecht zu werden. Dann lohnt sich der Übergang zu einer Microservices-Architektur, um der Komplexität Herr zu werden.

Während komplizierte – aber nicht komplexe – Fachlichkeit gut durch monolithische Systeme abgebildet werden kann, lassen sich komplexe Systeme aufgrund einer Vielzahl wechselwirkender Teilsysteme nicht vollständig analysieren und nur teilweise steuern. Es ist daher nicht möglich, die optimale Architektur eines komplexen Systems vorab zu bestimmen. Diese muss viel mehr iterativ entwickelt werden, wobei praktische Erfahrungswerte einfließen können. Damit das möglichst unkompliziert vonstattengehen kann, sind z. B. kurze Releasezyklen, eine effektive Delivery Pipeline und geringer Kommunikationsbedarf durch kleine Teams nötig, wie es eine Microservices-Architektur ermöglicht [23].Daraus folgt allerdings auch, dass es gefährlich ist, Microservices auf der grünen Wiese zu entwickeln. Wählt man an kritischen Stellen den falschen Service-Schnitt, kann das folgenschwere Konsequenzen nach sich ziehen. Je besser man die Geschäftsprozesse und das Verhalten einer funktionierenden (aber schwer erweiterbaren) monolithischen Anwendung studieren kann, desto besser kann man einschätzen, wo der optimale Schnitt zu setzen ist. Es ist allerdings anzumerken, dass diese Frage in der Literatur [24], [25] kontrovers diskutiert wird (**Abb. 3**) .

Fazit

Der Einsatz von Microservices zielt darauf ab, komplexe Anwendungen, die als Monolith nur schwer zu managen wären, durch Dezentralisierung und Autonomiemaximierung skalierbar, wartbar und leicht erweiterbar zu machen. Jeder Microservice erfüllt dabei

genau eine Funktion im Sinne der fachlichen Logik. Die wichtigsten Treiber in Richtung Microservices sind hohe Komplexität, hoher Innovationsbedarf sowie Leidensdruck bei der Skalierung. Microservices erben allerdings alle Probleme, die aus verteilten Anwendungen bekannt sind. Service-übergreifendes Monitoring und Logging aber auch Refactoring und Testen stellen eine Herausforderung dar. Zudem muss das System mit Eventual Consistency und Teilausfällen zurechtkommen.

Auf der technischen Seite bringt Docker bei Entwicklung und Betrieb von Microservices viele Vorteile wie etwa schnelles Deployment und einfache Skalierbarkeit mit sich. Die nahtlose Integration in den Entwicklungsworkflow stellt besonders für den .NET-Entwickler einen großen Vorteil dar.

Dr. Felix Nendzig studierte Physik am KIT und der Universität Heidelberg, wo er auch promovierte. Der Wechsel in die IT-Branche führte ihn zur Accso – Accelerated Solutions GmbH, wo er als erfahrener Senior Software Engineer tätig ist.

Links & Literatur

[1] https://docs.microsoft.com/de-de/dotnet/standard/microservices-architecture/

[2] https://martinfowler.com/bliki/BoundedContext.html

[3] https://de.wikipedia.org/wiki/Gesetz_von_Conway

[4] https://www.informatik-aktuell.de/entwicklung/methoden/wie-gross-darf-ein-microservice-sein-und-ist-das-ueberhaupt-wichtig.html

[5] https://www.elastic.co/de/elk-stack

[6] https://zipkin.io

[7] https://www.jaegertracing.io

[8] https://www.appdynamics.com

[9] https://www.informatik-aktuell.de/entwicklung/methoden/microservices-sind-verteilte-systeme-asynchronitaet-und-eventual-consistency.html

[10] https://www.embarc.de/wp-content/uploads/2016/06/Architektur-Spicker3-Microservices.pdf

[11] https://www.informatik-aktuell.de/entwicklung/methoden/der-fachliche-schnitt-von-microservices-was-ist-das-was.html

[12] https://www.docker.com/community-edition

[13] https://docs.docker.com/docker-for-windows/

[14] de la Torre, Cesar; Microsoft: „Containerized Docker Application Lifecycle with Microsoft Platform and Tools“: https://azure.microsoft.com/de-de/resources/containerized-docker-application-lifecycle-with-microsoft-platform-and-tools/

[15] Lasker, Steve: „.NET Docker Development with Visual Studio 2017“: https://www.youtube.com/watch?v=hFCwfHo_Kbc

[16] https://docs.microsoft.com/aspnet/core/publishing/visual-studio-tools-for-docker/

[17] https://docs.microsoft.com/en-us/azure/vs-azure-tools-docker-hosting-web-apps-in-docker/

[18] https://azure.microsoft.com/documentation/articles/container-service-intro/

[19] https://docs.docker.com/swarm/overview/

[20] https://docs.docker.com/engine/swarm/

[21] https://docs.mesosphere.com/1.10/overview/

[22] http://kubernetes.io

[23] https://www.informatik-aktuell.de/entwicklung/methoden/microservices-nur-bei-ausreichender-komplexitaet.html

[24] https://martinfowler.com/bliki/MonolithFirst.html

[25] https://martinfowler.com/articles/dont-start-monolith.html

[26] https://www.informatik-aktuell.de/entwicklung/methoden/wann-und-fuer-wen-eignen-sich-microservices.html

Git erobert die Entwicklerwelt

Go Git!

Git ist „hip“ und gilt als Star unter den Versionskontrollsystemen. Der Grund dafür ist einfach: Git ist schnell, leistungsfähig und plattformunabhängig. Kein Wunder, dass immer mehr Teams, vor allem auch aus der Microsoft-Welt, den Umstieg planen. Doch um Git optimal zu nutzen, ist fundiertes Hintergrundwissen nötig – denn Git arbeitet von Grund auf anders als die bekannten „klassischen“ Systeme. Richtig eingesetzt macht Git aber ordentlich Spaß – durch mehr Performance, Produktivität und Flexibilität im Entwicklungsalltag.

von Uwe Baumann

Eigentlich – so schien es – war das Thema Versionskontrolle in der Microsoft-Welt abgehakt: Team Foundation Version Control (TFVC) als ausgereiftes, stabiles System war gesetzt – als Open-Source-Alternative stand Apache Subversion bereit –, um den Rest des Kuchens stritten sich Systeme wie Perforce und Rational ClearCase. Doch dann kam Git und eroberte die Welt im Sturm: Nach der aktuellen Umfrage des Entwicklerportals Stack Overflow nutzen rund 87 Prozent der Entwickler Git; Subversion und TFVC landen mit 16,1 und 10,9 Prozent abgeschlagen auf den Plätzen zwei und drei [1].

Was ist der Grund für den kometenhaften Aufstieg von Git? Was macht Git anders als die etablierten Platzhirsche? Und was gibt es zu beachten, wenn man das eigene Team auf Git migrieren will? Die wichtigsten Antworten auf diese Fragen möchte ich Ihnen im Folgenden liefern.

Zentral versus dezentral

Wer sich ein Git Cheat Sheet (wie es diesem Heft beiliegt) besorgt, könnte den Eindruck bekommen, dass Git eigentlich nur eine Variante des bereits bekannten Musters darstellt: Befehle zum Sichern des Versionsstands, zum Abrufen früherer Versionen sowie zum Verwalten von Varianten mittels Branching und Merging.

Aber der Schein trügt: Git funktioniert intern komplett anders, denn es ist als dezentrales System ausgelegt. Das bedeutet: Während TFVC und Subversion auf eine zentral verwaltete Datenbank setzen (**Abb. 1**), kommt Git auch gut ohne zentrale Datenbank aus. Jeder Nutzer hat vielmehr seine eigene Datenbank, unter Git „Repository“ oder kurz „Repo“ genannt (**Abb. 2**).

Unter Git arbeitet man die meiste Zeit mit dieser lokalen Datenbank und kann dann die erzeugten Änderungen (Commits) mit den anderen Projektbeteiligten austauschen. Obwohl Git dies nicht erfordert, wird hier dann doch wieder auf die bewährte Architektur mit einem Zentralserver (Blessed Repository) zurückgegriffen.

Schnell, sicher, flexibel

Das lokale Arbeiten bringt einige Vorteile mit sich: Git reagiert generell extrem schnell auf Benutzereingaben – es müssen ja nicht für jede Aktion langsame Netzwerkzugriffe durchgeführt werden. Diese gute Performance ist in der täglichen Praxis deutlich spürbar.

„Völlig losgelöst“ vom Server arbeiten zu können, erhöht auch die Flexibilität bei der Arbeit mit

der Versionskontrolle. Kein Netzwerk? Kein Problem, auch wenn man an einem Ort am Sourcecode arbeitet, an dem keine Verbindung zum Server besteht. So ist es beispielsweise im Zug, in abgeschotteten Sicherheitsbereichen, bei Ausfall der DSL-Leitung und in ähnlichen Szenarien trotzdem möglich, den vollen Funktionsumfang der Versionsverwaltung zu nutzen – ein Plus für Service-Ingenieure, Consultants und alle anderen örtlich mobilen Entwickler.

Ein weiterer Bonus der dezentralen Auslegung von Git: Änderungen am Repository sind zunächst einmal privat, bis sie auf den Server übertragen werden. So ist es unter Git üblich, bei der Arbeit eigene, private Branches anzulegen, die vor der Übernahme in das zentrale Repository oft auch wieder gelöscht werden. So lassen sich im Alltag leicht Experimente am Code vornehmen, die dann sehr einfach wieder zurückgerollt werden können – sozusagen ein „Undo auf Steroiden". Auch diese erhöhte Flexibilität bei der Codeerstellung ist ein Feature, das die meisten Git-Nutzer nicht mehr missen wollen.

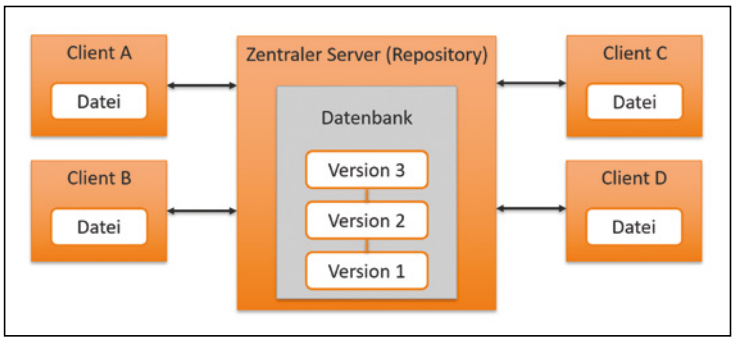


Abb. 1: Zentrale Versionskontrolle

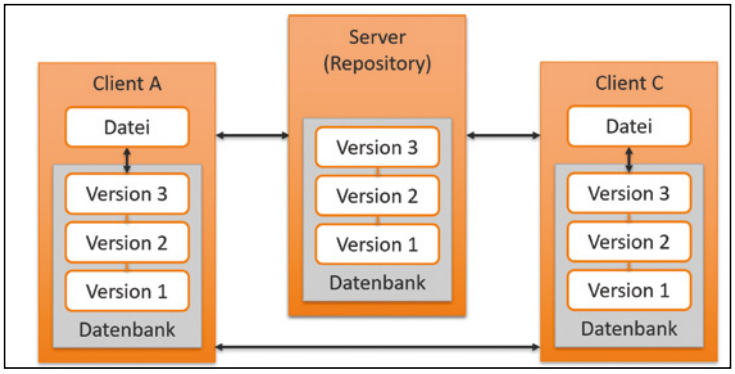


Abb. 2: Verteilte Versionskontrolle

Nicht zuletzt bietet die dezentrale Datenspeicherung auch noch einen weiteren Vorteil: Da die einzelnen lokalen Repos in der Praxis oft synchronisiert werden, hat jeder Entwickler sozusagen eine Sicherheitskopie, was zu einer extrem hohen physischen Datensicherheit führt – und das ganz ohne explizite Back-ups auf einem Server.

Clevere Datenspeicherung

Apropos Flexibilität: Um die dezentrale Datenspeicherung möglich zu machen, setzt Git auf ein flexibles grafenbasiertes Speichermodell. Die Versionsgeschichte wird durch eine einfach verkette Liste abgebildet, die einzelnen Nodes repräsentieren die Commits (Änderungsdatensätze) mitsamt der Metadaten wie Name, Datum und Eincheckkommentar. Branches und Label werden durch leichtgewichtige Pointer realisiert. Die Integrität der Commits wird durch Hash-Werte sichergesellt, die über jedes Commit berechnet werden und diese gleichzeitig eindeutig identifizieren (**Abb. 3**).

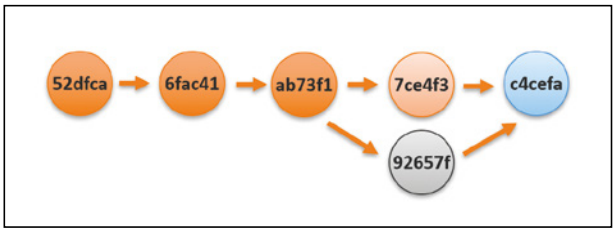


Abb. 3: Versionsgeschichte als Abfolge von Commits, identifiziert durch Hash-Werte

Dieses Speichermodell macht Git in jeder Hinsicht noch flexibler. Das Erstellen von Branches geht extrem schnell und einfach, da intern nur ein weiterer Listen-Pointer eingefügt werden muss. Die interne Datenstruktur von Git ermöglicht die Manipulation der Versionsgeschichte auf jede erdenkliche Weise. Commits können zusammengefasst, entfernt und verändert werden, durch spezielle Manipulationen am Datenmodell können „technische" Commits, wie sie bei anderen Systemen beispielsweise beim Mergen zwangsläufig entstehen, komplett vermieden werden. Diese Commits sind deshalb oft unerwünscht, weil sie keine inhaltliche Weiterentwicklung des Codes darstellen, sondern einfach nur markieren, dass eine technische Aktion wie ein Merge (Zusammenführen von Branches) oder ein Revert (Rückgängigmachen von Änderungen) auf dem Repo durchgeführt wurde.

Dadurch ist es unter Git möglich, eine sehr saubere und aufgeräumte Versionsgeschichte zu erstellen. Diese ist dann sehr gut les- und wartbar – im Gegensatz zur oft chaotischen Historie konventioneller Systeme. Viele Git-Nutzer sehen die Versionsgeschichte deshalb als organisch entstandenes Changelog an, das die logisch-semantische Projektgeschichte erzählt, und investieren entsprechend viel Aufwand in dessen Pflege.

Power und Komplexität

Doch um es mit den Worten von Spidermans Onkel zu sagen: „With Great Power Comes Great Responsibility" [8]. Das clevere Datenmodell bedeutet eben auch, dass ein erhöhter Lernaufwand nötig ist, um mit Git sicher und produktiv arbeiten zu können. Die vielen Möglichkeiten möchten beherrscht werden, und das geht nicht ohne ein intensives Studium der internen Funktionsweise von Git.

Viele Einsteiger stellen fest, dass der Einstig zwar sehr leichtfällt, es aber eine unsichtbare Barriere gibt, auf die man dann trifft, wenn man fortgeschrittene Techniken anwenden will oder in ungewohnte Situationen gerät. Auch ist es bei Git durchaus möglich, Daten zu „verlieren". Meistens sind diese dann nicht wirklich gelöscht, aber bedingt durch den aktuellen Zustand des Datenmodells gerade einfach nicht sichtbar. Git überrascht Anfänger oft mit nicht gerade intuitivem Ausgang von Operationen – etwa, wenn nach Anzeige eines historischen Commits die neueren Commits verschwunden zu sein scheinen. Ich habe einmal in einem Forum gelesen: „Git can smell your Fear!" [9]. Zudem ermöglicht Git es, bestimmte Operation „mit Gewalt" durchzuführen, die bei den Mitbewerbern schlicht und ergreifend unmöglich sind – der Parameter *–force* macht's möglich. Wer hier nicht genau versteht, was er oder sie tut, und einfach blind einem Rezept aus dem Internet folgt, läuft unter Umständen geradewegs ins Verderben. Also ist eine kurze, intensive Lernphase für jeden Ein- und Umsteiger Pflicht (Kasten: „Der 3-Schritte-Plan zum Umstieg").

Plattformübergreifend

Git hat seinen Ursprung in der Linux-Welt. Es wurde als Ersatz für den bis dahin vom Linux-Kernel-Team genutzten BitKeeper (einem kommerziellen Pionier der dezentralen Versionsverwaltung) entwickelt. Die Linux-Gemeinde hatte sich mit dem Hersteller von BitKeeper überworfen und brauchte eine Alternative. Der Legende nach schloss sich Linus Thorwalds im stillen Kämmerchen ein und codierte die erste Version von Git in wenigen Tagen kurzerhand selbst [10].

Aufgrund der offensichtlichen Vorteile verbreitete sich das System sehr schnell. Überall auf der Welt fanden sich Mitarbeiter für die Weiterentwicklung – und innerhalb weniger Jahr hatte sich Git zum De-facto-Standard entwickelt. Heute sind Git-Implementierungen und Clients für praktisch alle wichtigen Betriebssysteme und integrierten Entwicklungsumgebungen verfügbar (Kasten: „Gute Git-Clients"). Git ist schnell, kompakt, einfach zu installieren und sofort einsetzbar. Sämtliche Operationen lassen sich über die Kommandozeile ausführen und es gibt zahlreiche freie und kommerzielle grafische Git-Clients für Windows, Mac, Linux und Co. Egal ob Visual Studio, Visual Studio Code, Eclipse, Android Studio oder XCode – überall ist die integrierte Git-Unterstützung längst Standard.

All das macht Git auch in der aktuellen Cross-Plattform-Entwicklungswelt zu einem idealen Tool. Unter Windows mit Xamarin entwickeln, auf einem Mac bauen und im Apple Store veröffentlichen? Kein Problem. Ein zentrales Repo für alle Skripte im Unternehmen, egal ob Bash oder PowerShell? Kein Problem! Git standardisiert die Welt der Versionsverwaltung.

Der 3-Schritte-Plan zum Umstieg

Informieren und Lernen. Das beste Buch zu Git („Pro Git") findet sich auf der offiziellen Git-Website [2] – auch als E-Book für alle gängigen Reader. Ein sehr gutes Tutorial bietet auch die Firma Atlassian auf ihrer Website [3] – und es gibt zahlreiche Videotutorials und Quickstarts. Wichtig: Nicht nur Befehle lernen, sondern sich intensiv mit der Funktionsweise von Git vertraut machen. Git ist anders!

Üben. Es empfiehlt sich, zunächst mit der Kommandozeile und einem einfachen Projekt zu üben, bis man die grundlegenden Operationen sicher beherrscht. Eine große Hilfe hierbei ist das kleine Tools GitViz [4], das die internen Vorgänge in einem Git-Repo visualisiert und so beim Verständnis hilft.

Migrieren. Es ist möglich, Datenbanken von bestehenden Systemen wie Subversion und TFS nach Git zu migrieren – Git bietet direkten Support für den Import von SVN-Repos [5] und auch Team Foundation Server hat eine eingebaute Importfunktion (allerdings nur für maximal 30 Tage Versionsgeschichte) [6]. Open-Source-Projekte wie beispielsweise git-tf helfen weiter [7]. Viele Teams checken den Sourcecode aber einfach zu einem Stichtag aus dem bestehenden System aus und in Git wieder ein – so gibt es zwar einen Bruch in der Historie, aber dafür ist der Aufwand minimal. Das alte System kann dann zur Referenz in einen Read-only-Modus versetzt werden.

Grenzen und Fallstricke

Wo Licht ist, ist auch Schatten. Erwähnt habe ich schon, dass Git eine steile Lernkurve hat. Weitere Limitationen liegen genau in dem oben gelobten dezentralen Datenmodell. Was, wenn mein Repository sehr groß ist und ich es nicht in kleinere, modularere Teile aufspalten kann? Schließlich muss das Repo auf jedem Entwicklerrechner repliziert werden und dort der entsprechende Speicherplatz zur Verfügung stehen. Was das bedeutet, zeigt sich an einem extremen Beispiel: Das Windows-Team bei Microsoft wollte auf Git umsteigen – und musste feststellen, dass dies aufgrund der Größe des monolithischen Windows-Sourcecodes mit 3,5 Millionen Files und 300 GB Repo-Größe praktisch unmöglich war. Operationen dauerten teils Stunden – ein unhaltbarer Zustand. Microsoft reagierte auf die Probleme mit der Entwicklung von Git Virtual File System – einer Erweiterung von Git, die das System zumindest teilweise wieder in eine Client-Server-Software zurückverwandelt [11].

GitHub empfiehlt, ein einzelnes Repo unter 1 GB und einzelne Files unter 100 MB zu halten – klassische Versionskontrollsysteme ermöglichen hingegen riesige Peta-

Gute Git-Clients

Atlassian Sourcetree: Kostenlos nach Registrierung, verfügbar für Windows und Mac; der Klassiker unter den grafischen Git-Clients. Bietet alle Grundoperationen in einer relativ modernen Oberfläche, enthält aber keinen Merge-Editor (**Abb. 4**).
fournova Tower: 59–79 € pro User und Jahr; ursprünglich nur für Mac erhältlich, jetzt auch mit einer Windows-Version vertreten; sehr intuitiv mit guter Übersicht bei anspruchsvollen Operationen wie Merges (**Abb. 5**).
Axosoft GitKraken: 49 $ pro User und Jahr; frei verfügbar für nichtkommerzielle Arbeit und Open Source; erhältlich für Windows, Mac und Linux (**Abb. 6**).

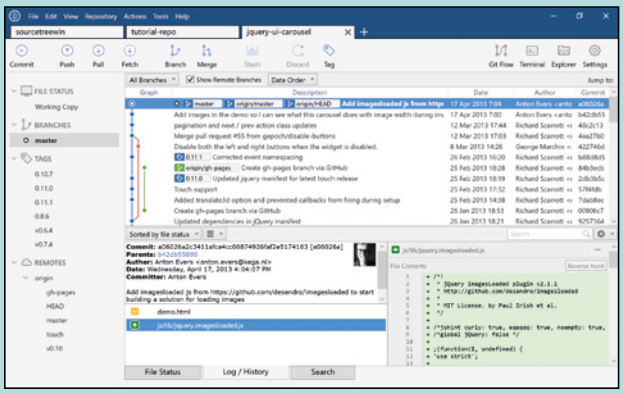


Abb. 4: Atlassian Sourcetree

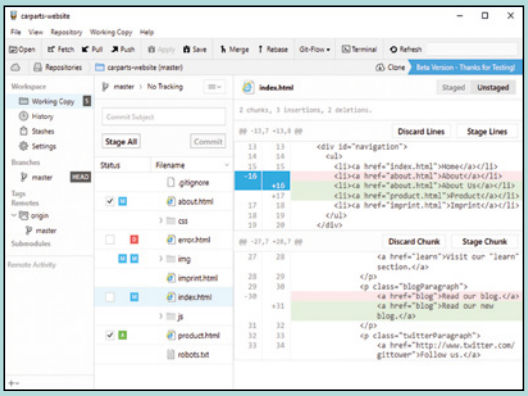


Abb. 5: fournova Tower

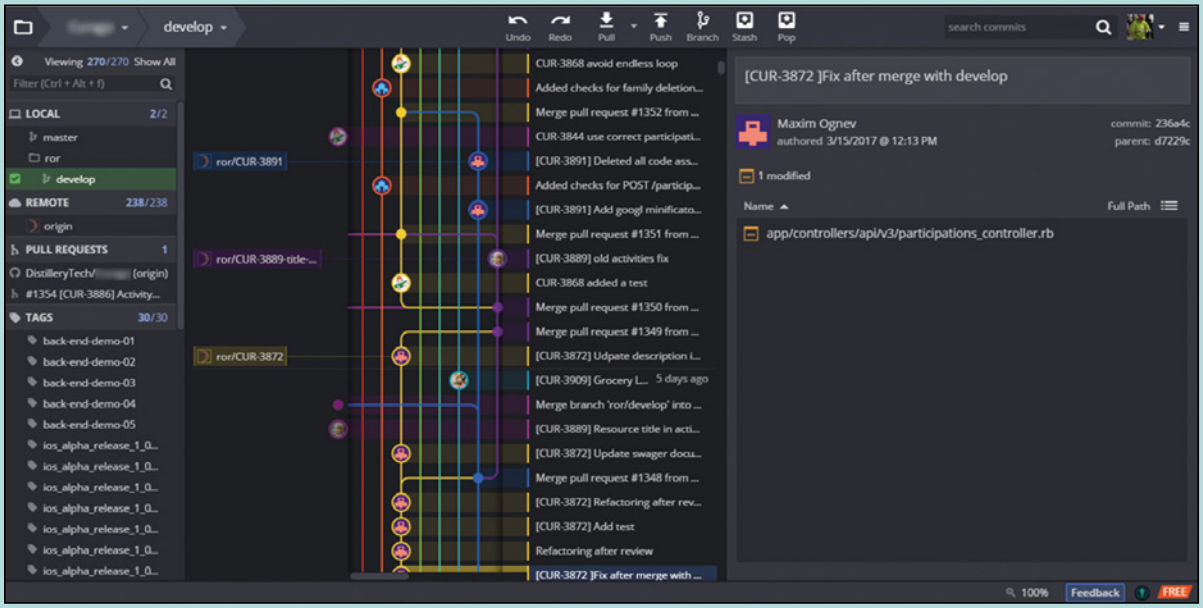


Abb. 6: Axosoft GitKraken

byte-Versionsdatenbanken – schließlich können Nutzer hier auch nur wenige Files aus dem Zentralserver auschecken, ohne gleich die ganze Datenbank kopieren zu müssen.

Zugegeben, auch unter Git existieren teilweise Lösungen für dieses Problem – Stichworte sind hier Grafting und Shallow Checkouts, Git Submodules und Subtrees. Diese Workarounds sind aber nur bis zu einem gewissen Grad praktikabel und performant. Wer also sehr große Repos hat, sollte sich den Umstieg auf Git gut überlegen und zunächst einmal Performancetests durchführen.

Ein weiteres Feature, das Git prinzipbedingt nicht unterstützen kann, ist das exklusive Auschecken von Dateien (Check-out Locks), sodass diese nur von einer Person zu einem Zeitpunkt geändert werden können. Gerade CAD- und Engineering-Systeme verlassen sich oft noch auf diese Funktionalität, die nur bei den klassischen Versionskontrollsystemen zu haben ist.

Neue Workflkows

Die neuen technischen Möglichkeiten von Git ebneten den Weg für neue Workflows in der Teamzusammenarbeit. Kollaborationsplattformen wie GitHub und Azure DevOps (vormals Team Foundation Server) bieten neue Wege, die Qualität und Transparenz von gemeinsam erstelltem Code zu steigern. Der Prozess der Pull Requests, ebenfalls aus der Open-Source-Szene übernommen, ist zum Quasistandard in modernen Softwareteams geworden. Dabei wird zunächst dezentral an Features und Bugfixes in eigenen, sogenannten Topic Branches gearbeitet – die Überführung in geschützte, qualitätsgesicherte Codelinien geschieht dann über einen kollaborativen Prozess, in dem automatisierte Tests und Builds ausgeführt werden können und das Team den Code in gemeinsamen Reviews bewerten und bearbeiten kann. Pull Requests sind kein Teil von Git, werden aber durch die Flexibilität der Git-Architektur erst praktikabel.

Fazit: Widerstand ist zwecklos

Git ist ein schnelles, leistungsfähiges und revolutionäres System zur Versionskontrolle, das sich in der Softwareindustrie durchgesetzt hat. In Zukunft dürfte Git das faktische Monopol für Versionsverwaltung besitzen. Wer sich noch nicht mit Git beschäftigt hat, sollte das schnell nachholen – es lohnt sich. Mit dem nötigen Respekt vor der Mächtigkeit des Systems, ein bis zwei Tagen Studium oder Seminar und ein bis zwei Wochen Zeit für die Umgewöhnung gelingt der Umstieg sicher und schnell. Go Git!

Uwe Baumann ist ausgewiesener Experte für Microsoft Visual Studio, ALM und .NET. Seit 1999 begleitet er die verschiedenen Technologien zur Softwareentwicklung in der Microsoft-Welt. Als ALM Consultant und strategischer Technologieberater bei der artiso solutions GmbH untertützt er mit seiner umfassenden Erfahrung Kunden dabei, den Herausforderungen der modernen Softwareentwicklung zu begegnen.

Links & Literatur

[1] https://insights.stackoverflow.com/survey/2018/
[2] https://git-scm.com/book/en/v2
[3] https://de.atlassian.com/git/tutorials
[4] https://github.com/Readify/GitViz
[5] https://git-scm.com/docs/git-svn
[6] https://docs.microsoft.com/en-us/azure/devops/repos/git/import-from-tfvc?view=vsts&tabs=new-nav
[7] https://blog.cellenza.com/devops/migrating-from-tfvc-to-git/
[8] https://quoteinvestigator.com/2015/07/23/great-power/
[9] https://stevebennett.me/2012/02/24/10-things-i-hate-about-git/
[10] https://www.linuxjournal.com/content/git-origin-story/
[11] https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/

Aus VSTS wird Azure DevOps – mehr als nur ein neuer Name?

Des Kaisers neue Kleider

Das Jahr 2018 hat für Nutzer der Microsoft-ALM-/DevOps-Plattformen TFS und VSTS durch zwei große Ankündigungen mehrere gedankliche Neuausrichtungen bedeutet: Zum einen war da im Frühjahr die Absichtserklärung von Microsoft, GitHub für 7,5 Mio. US-Dollar zu kaufen, zum anderen die Umbenennung von VSTS und TFS zu Azure DevOps bzw. Azure DevOps Server in der Sommerphase.

von Nico Orschel und Thomas Rümmler

Im Folgenden wollen wir beide Themen aufgreifen und zu Beginn das „Warum?“ etwas genauer anschauen; anschließend wird es um das „Wie manifestiert sich die Umbenennung im Produkt?“ bzw. „Was gibt es für neue Features?“ gehen. Einen weiteren inhaltlichen Schwerpunkt wird die Betrachtung des Zusammentreffens von GitHub und Azure DevOps darstellen, bzw. die Erkenntnis, dass Open Source mit Azure DevOps sich nicht ausschließt, sondern Geld spart und die Produktivität fördert.

Raider heißt jetzt Twix aber sonst ändert sich nix?

Das Umbenennen (oder neudeutsch Rebranding) von Microsoft-Produktnamen scheint für so manchen Nutzer schon eine Tradition zu sein. Im TFS-Cloud-Umfeld haben wir mittlerweile auch schon die eine oder andere Namensänderung durchlaufen: 2011 Team Foundation Service, 2013 Visual Studio Online (VSO) und 2015 Visual Studio Team Services (VSTS). Warum jetzt aber schon wieder ein neuer Name? Im Microsoft-Universum gibt es eine überschaubare Anzahl an „starken“ Markennamen. Exemplarisch seien Visual Studio, Azure, Office 365 oder Xbox genannt. An diese Marken gliedern sich wiederum die eigentlichen Produkte an. TFS heißt beispielsweise nicht nur Team Foundation Server, sondern formal gesehen Visual Studio Team Foundation Server. Und genau hier liegt aus Sicht von Microsoft das Problem. Mit dem Markennamen ist auch eine gewisse Erwartungshaltung des Nutzers verbunden. Bezogen auf den TFS suggeriert das Visual Studio im Namen gerade für viele, die nicht Visual Studio als Entwicklungsumgebung nutzen, dass der TFS nur für .NET-Entwickler funktionieren würde. Betrachtet man die Entwicklung des TFS, erkennt man, dass die ersten Versionen 2005 und 2008 tatsächlich sehr stark an die .NET-Plattform gekoppelt waren. Mittlerweile hat sich das gesamte Umfeld allerdings massiv verändert und zu einer offenen Technologieplattform entwickelt. 2005 war Microsoft im Allgemeinen noch massiv gegen Open Source und zehn Jahre später ist das Unternehmen aus der Open-Source-Community nicht mehr wegzudenken. Auch das TFS-Entwicklungsteam hat hier eine gewisse Vorreiterrolle eingenommen. Beispiele sind die Integration von Git als zweites Versionskontrollsystem mit der Version TFS 2015 oder nativer Build-Agent-Support für Linux und Mac OS X.

Unabhängig von diesen Bemühungen ist aber leider selbst im Jahr 2018 die Wahrnehmung von vielen Entwicklern außerhalb der Microsoft-Technologiewelt, dass TFS nur mit .NET sinnvoll zu verwenden ist, und das ist schlichtweg falsch. Mit dem neuen Namen versucht Microsoft jetzt eine Art Befreiungsschlag. Die Marke Azure ist für Microsoft eine Art Leuchtturm. Azure steht für Offenheit und Technologievielfalt. Das gleiche Gedankengut teilen auch TFS und VSTS, sodass jetzt die neuen Namen Azure DevOps für den Cloud-Service und Azure DevOps Server für den On-Premises-Server sind. Auf DevOps selbst muss an dieser Stelle nicht mehr genauer eingegangen werden, da man an diesem Thema gefühlt überhaupt nicht mehr vorbeikommt. DevOps als Philosophie wird dabei als der nächste Schritt nach Agile betrachtet.

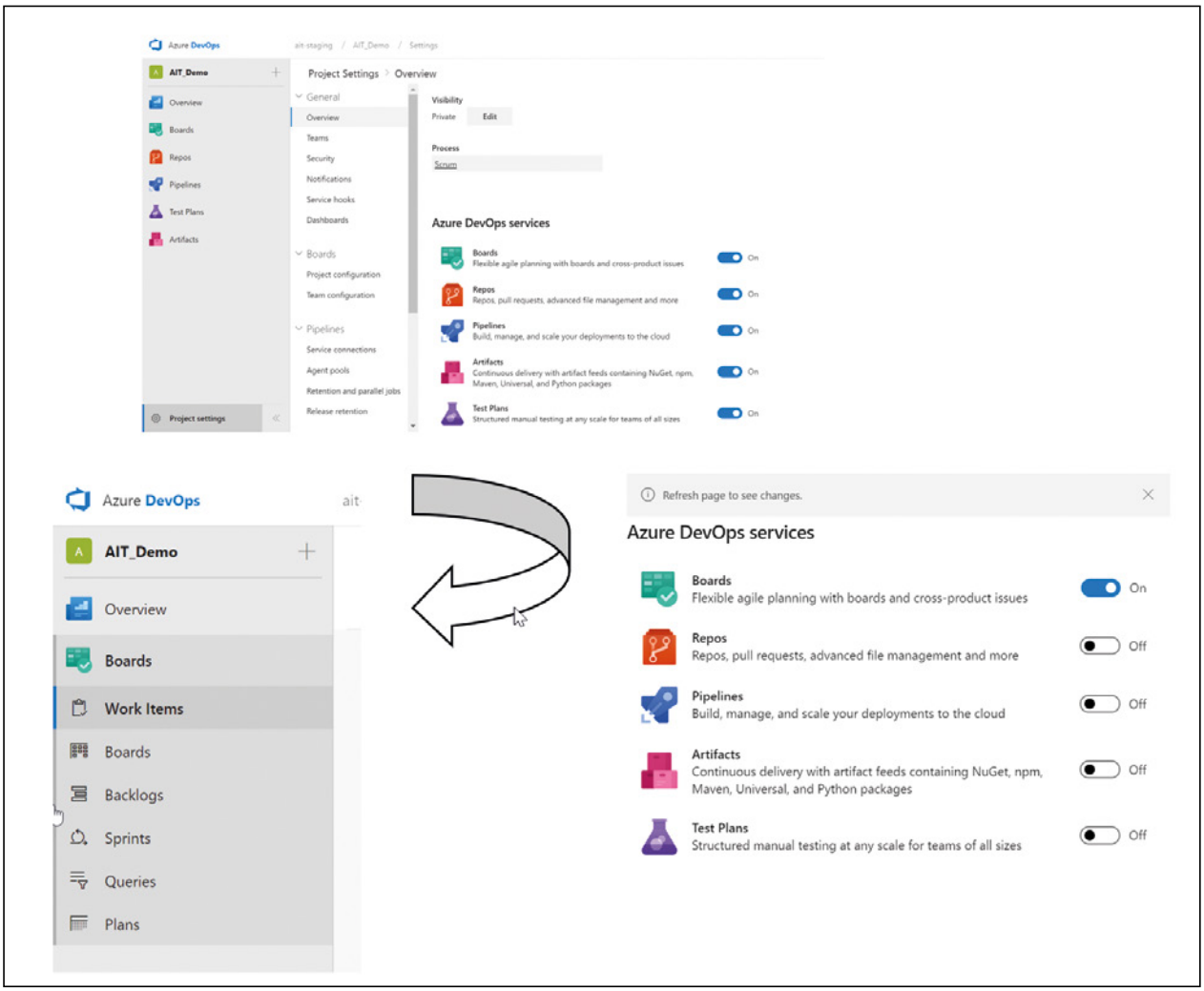


Abb. 1: Aktivierung/Deaktivierung einzelner Azure DevOps Services

Hat jetzt Microsoft etwa nur die Beschilderung des Produkts geändert oder habe ich als Nutzer auch etwas davon? Der erste große Punkt, der bei diesem Release auffällt, ist, dass jetzt die Services der Plattform als Gesamteinheit oder als getrennte einzelne Services nutzbar sind. Die gesamte Plattform trägt dabei den Namen Azure DevOps bzw. Azure DevOps Server. Im Gegensatz dazu tragen die einzelnen Services die Namen Azure Boards (Work-Item-Tracking, Backlogs, Enterprise-Portfolio-Management), Azure Repos (Git Repositories, Team Foundation Server Version Control), Azure Pipelines (alles rund um CI/CD als Weiterentwicklung des TFS/VSTS-Build- und Release-Managements), Azure Artifacts (Package Management) und Azure Test Plans (Testmanagement und Ausführung). In Unternehmen, die eine komplett neue Entwicklungslandschaft definieren, ist der Einsatz all dieser Teile die erste Wahl. Es gibt jedoch viele Fälle, in denen nur ein Teil benötigt wird. Das kann sein, weil es für andere Teile bereits gute Lösungen gibt, beispielsweise wenn ein Unternehmen eine gut ausgestattete Testmanagementumgebung hat, aber für die agile Planung und die CI/CD Pipeline eine durchgängige Lösung sucht. Des Weiteren gibt es die Fälle, in denen beispielsweise eine Fachabteilung eine Möglichkeit für agiles Anforderungsmanagement sucht oder schlichtweg ihre Arbeit mit den Azure Boards organisieren möchte. In beiden Szenarien ist es nun möglich, nur die benötigten Teile zu verwenden (**Abb. 1**) und die anderen, nicht benötigten Services einfach zu deaktivieren. Aus Sicht von Microsoft wird dadurch eine größere und breitere Zielgruppe besser ansprechbar.

Das neue UI

Neben der Umbenennung von VSTS in Azure DevOps wurde auch ebenfalls ein neues, durchgängigeres UI-Konzept für alle Benutzer etabliert. Mit dem neuen Layout wurden dabei mehrere Ziele verfolgt: Unterstützung des neuen Service-Konzepts und Verbesserung der Nutzungsmöglichkeiten der Services für den Nutzer durch ein konsistentes Design. Auch TFS/VSTS sind über die Jahre gewachsene Systeme, an denen viel verbessert und erweitert wurde; dadurch finden sich ggf. Optionen nicht an den Stellen, an denen ein Nutzer sie erwarten würde oder Navigationspfade sind unnötig lang.

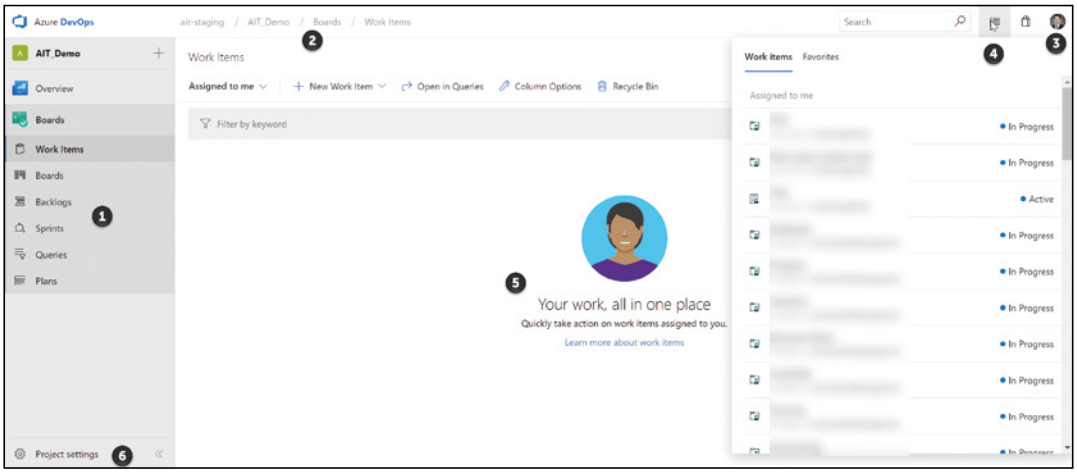


Abb. 2: Azure-DevOps-UI-Konzept

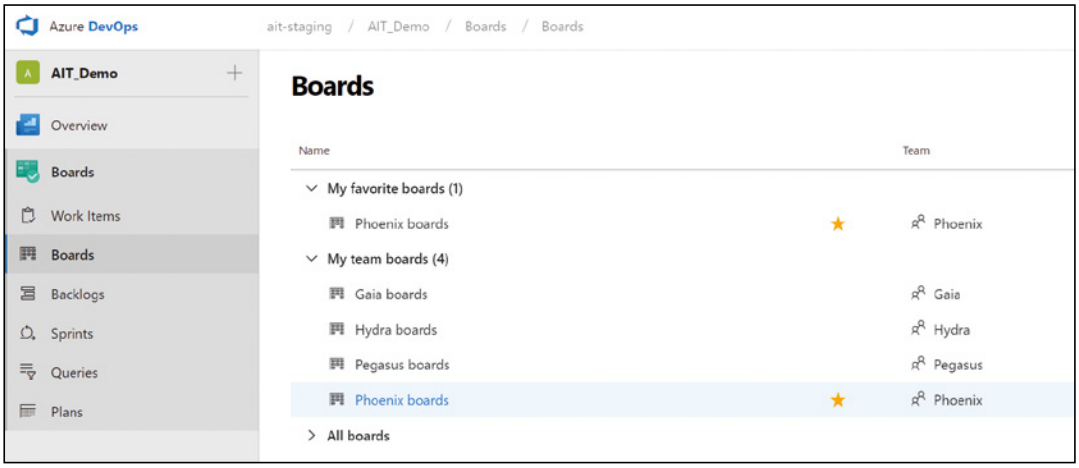


Abb. 3: Teams in Azure Boards

Die neue UI aus **Abbildung 2** folgt grob folgendem Schema: In der linken Leiste findet sich ein allgemeiner Überblicksbereich mit allgemeinen, Service-übergreifenden Funktionen wie Wiki, Dashboards und Reporting und natürlich auch die aktivierten Services (1). Links unten (6) finden sich immer kontextspezifische Einstellungen, wie z. B. der Link zu den Projekteinstellungen.

Rechts oben finden sich das Nutzerprofil (3) mit seinen Einstellungen und die Preview-Flags zur Aktivierung von Previewfunktionen. Daneben ist dort immer der Link zum Azure DevOps Marketplace zu finden. Ebenfalls befindet sich neben dem Profillink (4) auch eine Art Bereich mit den wichtigsten kontextsensitiven Funktionen bzw. eine Art Schnellstartbefehlsbereich.

Der flächenmäßig größte Bereich in der Mitte (5) ist immer zur Darstellung der Service-spezifischen Informationen gedacht. In diesem Bereich stellen alle Services bzw. die Adminseiten ihre Informationen dar. Es wird hier konsequent versucht, auf das Öffnen von zusätzlichen Browsertabs – wie in der Vergangenheit oft üblich – zu verzichten. Mittlerweile wird eher der Ansatz des kontextsensitiven Umschaltens präferiert. (2) zeigt in Form eines Bread-Crumbs-Links immer den aktuell aktiven Bereich bzw. Service an.

Azure Boards

Der erste „neue alte“ Service sind die Azure Boards. Unter ihm wurden alle Funktionalitäten bezüglich Work-Item-Tracking und der agilen Managementtools zusammengefasst. Beispiele für diesen Bereich sind das gesamte Thema Backlog-Portfolio-Management, Sprint-

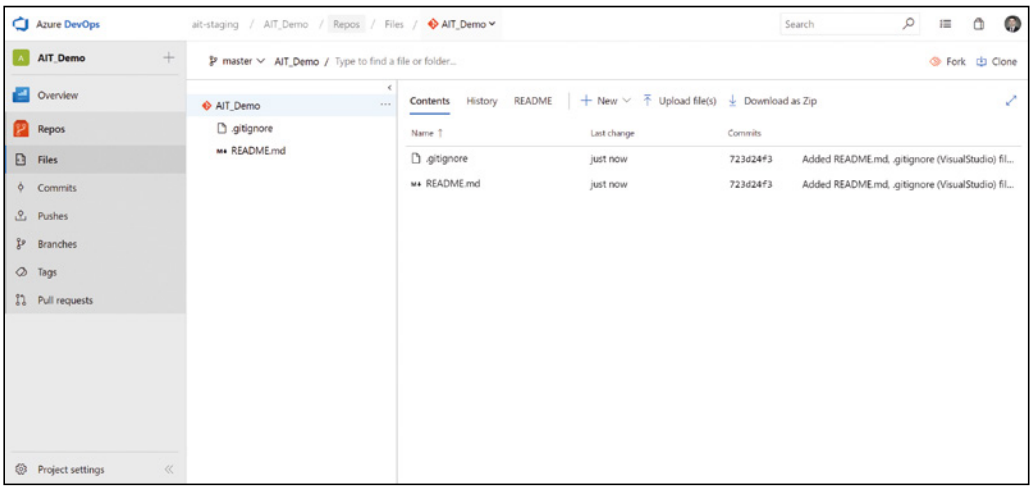


Abb. 4: Git in Azure Repos

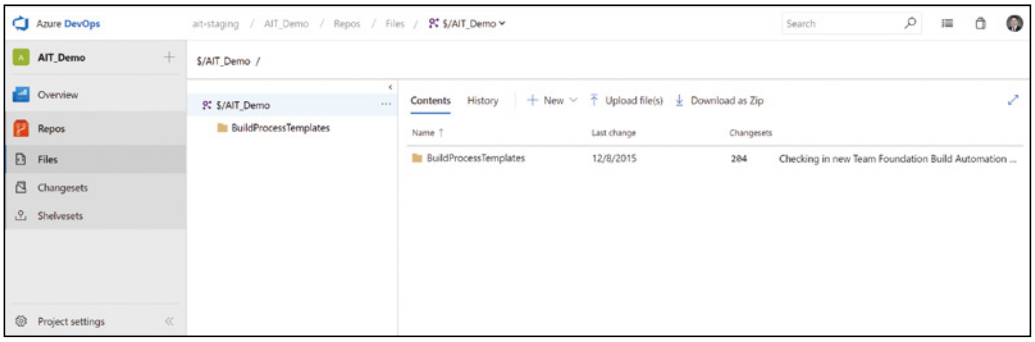


Abb. 5: TFVC in Azure Repos

Backlogs, Kanban Boards sowie die Verwaltung der Sprintzyklen.

In diesen Bereich fällt auch eine weitere Änderung durch das neue UI-Konzept auf (**Abb. 3**). In der Vergangenheit waren Teams immer direkt unter dem Teamprojekt angeordnet, erst dann kamen die eigentlichen Services. Das ist jetzt nicht mehr der Fall, stattdessen sind die Teams jetzt unterhalb der Services zu finden. Am Beispiel der Backlogs sieht man das recht anschaulich. Klickt man beispielsweise auf Backlogs, werden alle Team-Backlogs angezeigt. Früher war das genau andersrum, d. h., es wurde erst das Team ausgewählt und alle Services haben teambezogenen Informationen dargestellt. Die Änderung war wohl notwendig, um zum einen flexibler kontextbezogen navigieren zu können und zum anderen das neue Service-Modul bezüglich An- und Abwahl einzelner Services besser zu unterstützen.

Azure Repos

Den zweiten „neuen alten" Bereich bilden die Azure Repos. Dieser Bereich umfasst die ursprünglichen Funktionalitäten rund um das Thema Version Control (**Abb. 4** und **5**). Beispiele hierfür sind die Themen Git, Pull Requests, Branch Policies aber auch das gute alte zentrale Versionskontrollsystem TFVC (Team Foundation Server Version Control). Der Name von TFVC wurde wider Erwarten nicht in Azure DevOps Version Control umbenannt und wird es laut Microsoft auch nicht werden.

Im Gegensatz zu GitHub adressiert die Git-Funktionalität in Azure DevOps primär nichtöffentliche Git Repositories. Mit einer Lizenz habe ich hier als Nutzer theoretisch die Möglichkeit, unlimitiert private Git und TFVC Repositories anzulegen. Das gilt faktisch sowohl für die Anzahl als auch die Größe der jeweiligen Repositories. Dass Azure DevOps praktisch auch Riesen-Repositories handhaben kann, zeigt recht eindrucksvoll die Office- und Windows-Entwicklung mit Repository-Größen von mehr als 200 GB und über 20 Mio. Dateien.

Azure Pipelines

Der Bereich mit den in der persönlichen Wahrnehmung meisten Änderungen in den letzten Monaten ist Azure Pipelines. Unter Azure Pipelines wurden die ursprünglichen Build- und Releasemanagementbereiche zusammengefasst. Neben den fast schon üblichen Verbesserungen mit jedem Sprint, wie z. B. codebasierende YAML-Build- und -Release-Definitionen, bringt Microsoft mit der Umbenennung zwei weitere, sehr nützliche Verbesserungen mit: Das Kontingent der freien Minuten auf den Build und Release-Agents der CI/CD Pipeline für private Projekte wurde von 240 auf 1600 Minuten erhöht. Für Open-Source-Projekte stehen für Aufträge auf diesen Agenten sogar unlimitierte Minuten bereit. Die Open-Source-Projekte können dabei entweder unter Azure DevOps als Public Project oder auf einem externen System wie z. B. GitHub, GitLab, SVN, Sourceforge, usw. liegen. Microsoft hat dabei für Open-Source-Projekte die sonst übliche Limitierung von einer kostenlosen Pipeline auf zehn Pipelines

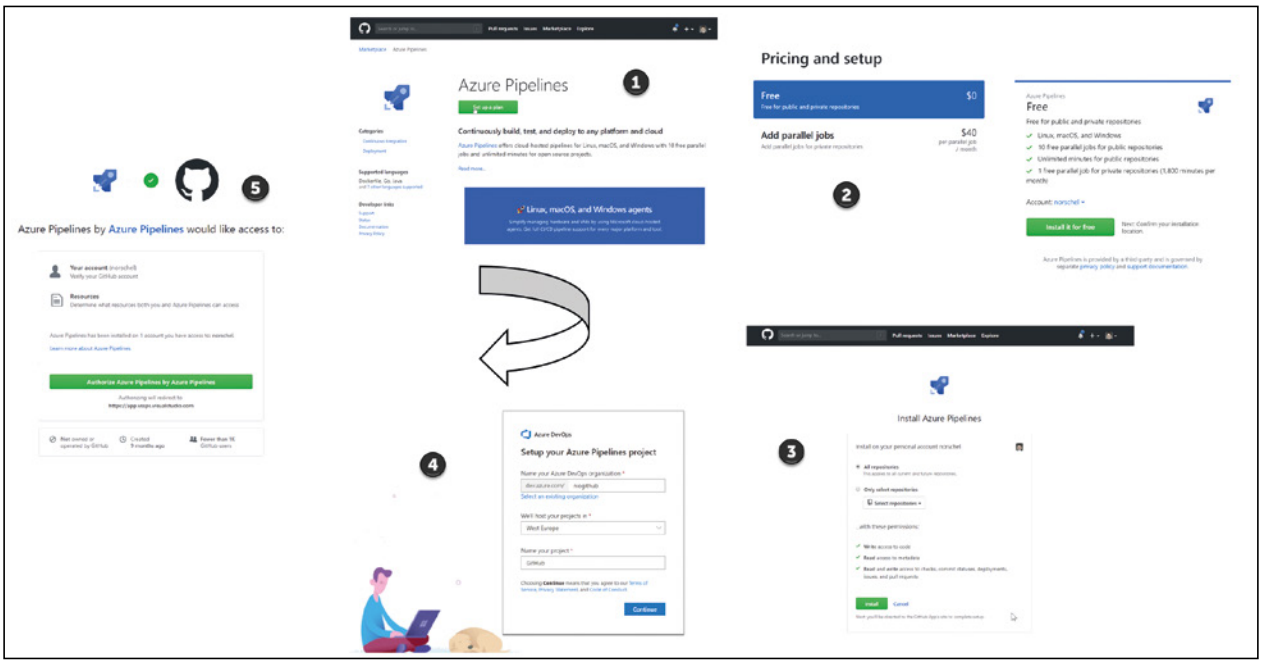


Abb. 6: Azure Pipelines mit GitHub verbinden

erhöht. Zehn Pipelines ermöglichen exemplarisch das parallele Kompilieren auf zehn verschiedenen Build-Agenten. Wenn aber Windows nicht genug ist, dann können die zehn parallelen Builds natürlich auch mit von Microsoft vorkonfektionierten Hosted-Agent-Pools auf Windows, Linux und Mac OS X genutzt werden. Das Betriebssystem spielt dabei für Microsoft bezüglich des Sponsorings keine Rolle. Für ein Open-Source-Projekt bedeutet das, dass Microsoft für sie faktisch die kompletten CI/CD Pipelines auf Windows, Linux und Mac OS X kostenlos bereitstellt. Die zehn parallelen Builds können dabei beliebig über alle Plattformen verteilt werden. Aus Sichtweise der Autoren ist das ein sehr spannendes Angebot für Open-Source-Projekte. Dieses Angebot an die Open-Source-Community sendet ebenfalls ein starkes Signal, dass Microsoft Open-Source-Projekte sehr wichtig sind und diese nicht als Spielerei ansieht, sondern auch finanziell nicht nur für GitHub tief in die Tasche greift.

Nachfolgend möchten wir kurz auf den Workflow zur Einbindung eines Open-Source-Projekts in Azure Pipelines eingehen. GitHub und Azure DevOps sind – obwohl beides mittlerweile zu Microsoft gehört – un-

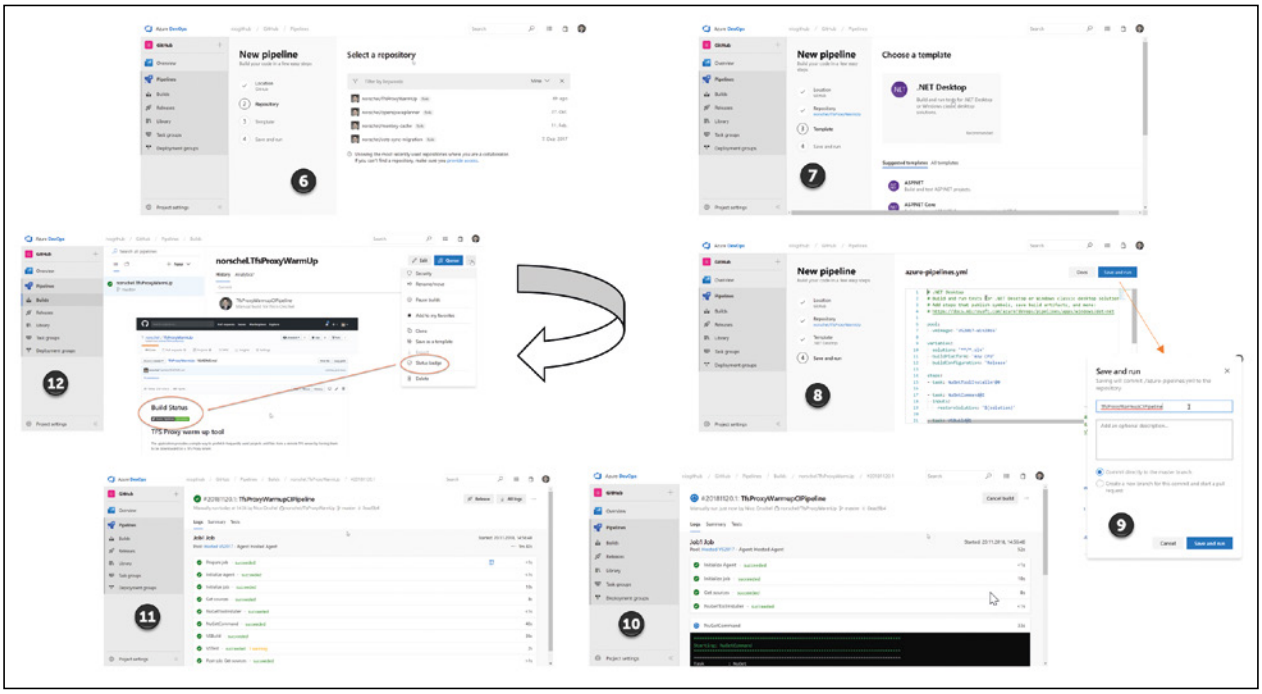


Abb. 7: YAML-basierte Azure-Pipeline-Definition anlegen

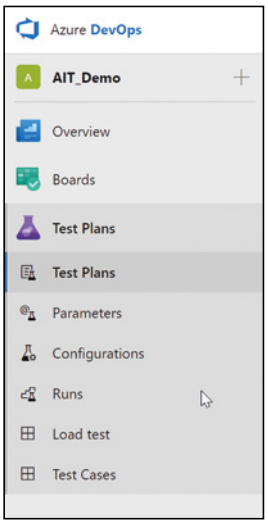


Abb. 8: Beispiel für Azure Test Plan

abhängige Plattformen. Die Integration von Azure DevOps ist deshalb auf GitHubs Seite auch nicht einfach in die eigentliche Plattform integriert, sondern verwendet die gleichen externen Erweiterungsmöglichkeiten wie alle anderen externen Partner auch: den GitHub Marketplace. Will ich auf GitHub Azure Pipelines nutzen, muss ich nicht erst ein Azure-Konto mit hinterlegter Kreditkarte anlegen, um den Service zu nutzen, sondern kann Azure Pipelines direkt und kostenlos über den GitHub Marketplace beziehen (**Abb. 6**). Im Hintergrund wird dann die Anbindung provisioniert. Als Ergebnis entstehen dann automatisch diverse GitHub Webhooks, die Azure DevOps über neue Aktivitäten auf GitHub informieren und eine abgespeckte Azure-DevOps-Organisation mit nur einem aktivierten Service, nämlich den Azure Pipelines, einrichten.

Nach der Anbindung von Azure Pipelines legt der Nutzer eine oder mehre YAML-basierte Build- und Release-Pipelines an. In **Abbildung 7** ist der Prozess dargestellt, der sich an **Abbildung 6** anschließt. Als Nutzer kann man natürlich den Prozess auch für weitere Build- und Releasedefinitionen nach einem ähnlichen Schema mehrfach durchlaufen. Nach der Verbindung von GitHub mit Azure Pipelines wählt der Nutzer das passende CI-/CD-Template und passt es ggf. an seine Bedürfnisse an. Anschließend muss er mit SAVE AND RUN nur noch seine Pipeline anstoßen. Will man auf seiner GitHub-Repo-Seite ebenfalls die Build-Statusinformationen sehen, kann man das Build-Status-Badge in seine Repo-Markdown-Files integrieren.

Azure Test Plans

Formale wie explorative Tests werden in Zukunft über Azure Test Plans verwaltet und ausgeführt. Hierbei hat sich strukturell im Vergleich zur vorherigen Version von TFS bzw. VSTS nichts verändert. Für das formale Testmanagement stehen immer noch sogenannte Testpläne und Testsuits zur Verfügung, die man ineinander verschachteln kann. Die eigentliche Testdefinition findet dann in den Testfall-Work-Items statt.

In der Navigation (**Abb. 8**) findet der Anwender zunächst die Verwaltung der Testpläne. Die nächsten Menüpunkte lassen die Definition von Parametern, also Eingabewerten, die wiederverwendet werden, sowie Testkonfigurationen zu. Über den Menüpunkt RUNS gelangt man in eine Ansicht der letzten Testdurchläufe, unabhängig davon, ob sie vom einem Build-/Releaseprozess oder Testplan erzeugt wurden.

LOAD TEST führt den Anwender zu dem Bereich der Cloud-basierten Lasttests. Hier besteht die Möglichkeit, die Anwendung durch die wiederholte und parallelisierte Ausführung bestimmter Testschritte einer definierten (ggf. hohen) Belastung auszusetzen. Interessant hierbei ist, dass ganz verschiedene Arten der Testdefinition für die Lasttests unterstützt werden, wie in **Abbildung 9** zu sehen ist.

Microsoft befindet sich im Bereich des Testmanagements seit längerer Zeit auf dem Weg in Richtung eines vollständigen Testmanagements im Web. Dabei gibt es

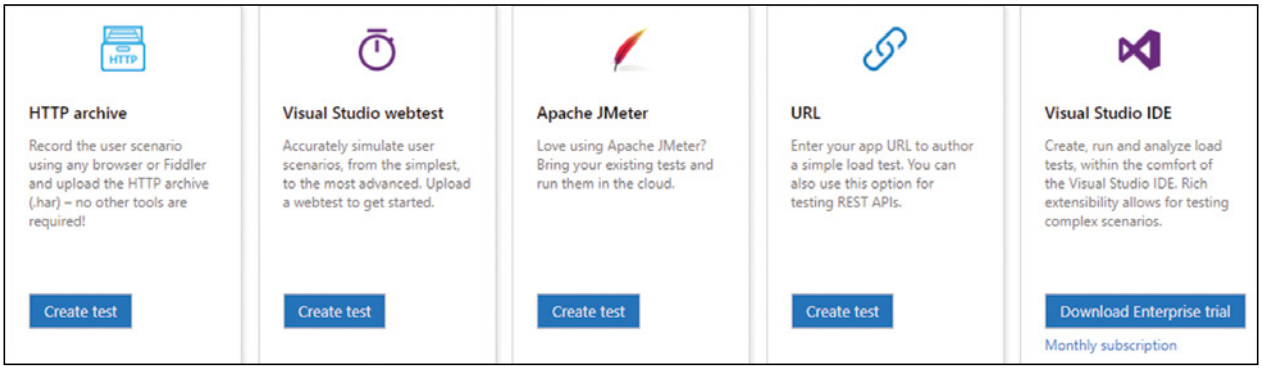


Abb. 9: Unterstützte Lasttesttypen in Azure DevOps

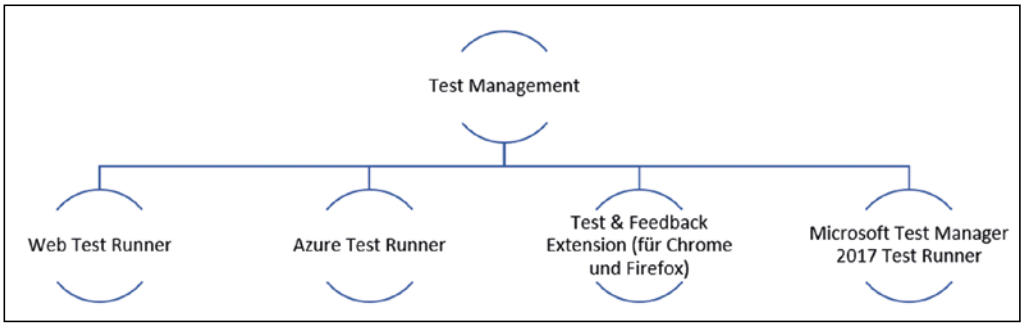


Abb. 10: Test-Runners für die manuelle Testausführung in Azure DevOps

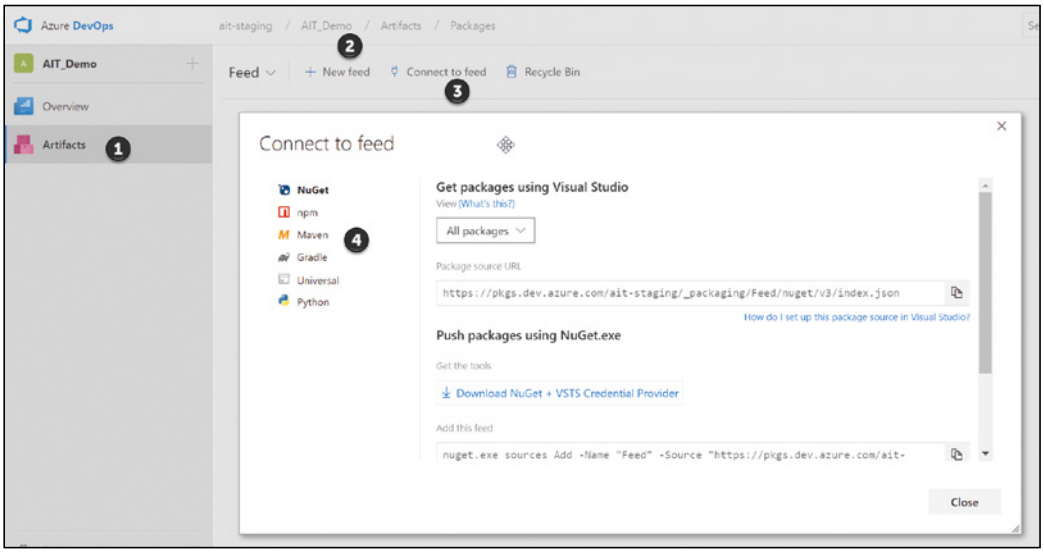


Abb. 11: Anlegen eines neuen Package Management Feeds in Azure DevOps

zwei wesentliche Knackpunkte, die im Web nicht so einfach zu lösen sind: Das Verbinden auf virtuelle Umgebungen im Bereich der automatisierten Tests (früher bekannt als Lab Management) sowie das Sammeln von systemnahen Diagnosedaten (z. B. Windows Eventlog, geöffnete Prozesse einer Desktopanwendung, usw.) beim Testen einer lokal installierten Anwendung (Desktopapplikation). Bei letzterem hat Microsoft nun einen Vorstoß gewagt und einen neuen Test-Runner bereitgestellt. Die zugrunde liegende Problematik ist, dass der Browser eine Webanwendung aus Sicherheitsgründen nicht so einfach auf systemnahe Ressourcen zugreifen lässt.

Durch den neuen Azure Test Runnes ergibt sich das auch in **Abbildung 10** dargestellte Bild an Test-Runnern, das in der Zukunft sicherlich wieder etwas aufgeräumt wird:

- Web Test Runner: für formale Tests von Webanwendungen
- Azure Test Runner: für formale Tests von Desktopanwendungen [1]
- Test- und Feedback-Extension (für Chrome und Firefox): für explorativen Testen von Webanwendungen und das Testen auf Mobilgeräten mittels des Diensts Perfecto Mobile [2]
- Microsoft Test Manager 2017 Test Runner: für formales Testen von Desktopapplikationen mit allen Diagnoseadaptern

Da der neue Azure Test Runner auf Basis des plattformübergreifenden Frameworks Electron implementiert ist und bereits beim Testen von Desktopapplikationen ein Aktivitätslog erstellt wird, kann man davon ausgehen, dass es sich hier um die potenzielle Ablösung des Microsoft Test Manager 2017 Test Runners (MTM) handelt. Noch immer steckt viel Funktionalität im MTM, die noch an keiner anderen Stelle abgebildet ist, aber mit dem Azure-Test-Runner geht es einen weiteren großen Schritt in Richtung der Ablösung dieser alten WPF-basierten Anwendung aus Visual-Studio-2010-Zeiten.

Azure Artifacts

Den Abschluss der Vorstellung der Azure DevOps Services bildet Azure Artifacts. Dieser neue Bereich fasst alle Funktionalitäten rund um das Thema Package Management zusammen (**Abb. 11**). Azure Artifacts unterstützt dabei die Platzhirsche in diesem Bereich wie

NuGet für den .NET-Bereich aber auch npm für den gesamten JavaScript-Bereich. Bei diesen Typen bleibt es aber nicht, sondern Maven, Gradle, Python sowie die Universal Packages sind ebenfalls mit an Board.

In Anhängigkeit vom Package Type unterstützt das Package Management zum Zeitpunkt der Drucklegung nicht nur die einfache Ablage und Versionierung der Artefakte in Form von Feeds, sondern bringt auch erweiterte Funktionalitäten wie z. B. Upstream-Support für NuGet und npm mit. Upstream-Support ermöglicht das Zwischenspeichern von Packages in Azure DevOps, wenn der Entwickler das Paket das erste Mal abruft. Steht das Paket aus diversen Gründen nicht mehr zur Verfügung, kann der Entwickler aber über Build Agents beispielsweise auf die zwischengespeicherte Version zurückgreifen. Auch dies ist ein Bereich, in dem Microsoft sicherlich noch das ein oder andere Feature liefern wird.

Zuvor relativ beiläufig erwähnt, ist auch das Universal Package für viele Entwickler sehr spannend. Das Feature selbst ist eine Weiterentwicklung einer zuvor nur Microsoft-intern entwickelten Technologie. Universal Packages ermöglichen das generische, versionierte Speichern von Dateien im TFS [3]. Das Besondere dabei ist die eingebaute Datenduplizierung bei Up- und Download, sodass gleiche Daten nicht mehrfach gespeichert oder abgerufen werden müssen. Durch diesen Ansatz sinkt das benötigte Speichervolumen für Build-Artefakte drastisch, während gleichzeitig die Performance steigt, da Teile der Daten oft schon auf lokalen PCs vorliegen.

Die Datenduplizierung selbst funktioniert momentan noch nur auf bestimmten Windows-Versionen; aber wenn sie unterstützt wird, dann werden die Inhalte bis auf die Blockebene genau mit dem jeweiligen Server abgeglichen. Hier kann man sicherlich auch davon ausgehen, dass es noch einige Neuerungen geben wird, weil wir uns noch eher am Anfang des Featurelebenszyklus befinden.

Fazit

Für die alten Hasen unter den Lesern – gemeint sind die Entwickler mit einer langen TFS- bzw. VSTS-Historie – mag sich die neue Namensgebung komisch anfühlen. Insbesondere wenn Microsoft Azure in der eigenen Entwicklung nicht die Zielplattform ist. Dennoch ist nachvollziehbar, wie sich Azure DevOps als starke Marke etablieren kann, die für Offenheit und eine Vielfalt an unterstützten Technologien steht. Diese Offenheit hat Microsoft durch den Zukauf von GitHub und die beibehaltene strikte Trennung zwischen den kommerziellen Produkten und dem Support der Open-Source-Community noch untermauert. Das kostenfreie Bereitstellen einer kompletten CI/CD Pipeline ist hier nur ein Beispiel.

Unterm Strich ist die aktuelle Weiterentwicklung von Microsofts Entwicklungsplattform also eine Win-win-Situation. Microsoft als Hersteller eröffnet sich Zugang zu einer größeren Community und arbeitet weiter am eigenen Image. Die Bestandskunden erhalten neue Features bzw. Vergünstigungen, wie bei den inkludierten Ausführungszeiten der CI/CD Pipeline zu sehen ist. Zu guter Letzt sei da noch die Open-Source-Community zu nennen, die von einem Weltkonzern mit vielen Möglichkeiten unterstützt wird.

In unseren industriellen Softwareentwicklungsprojekten nehmen wir diesen Wandel ebenfalls wahr. So ist der Einsatz von Azure-Cloud-Diensten eben auch bei einem mittelständischen Maschinen- oder Anlagenbauer kein Tabuthema mehr. Im Gegenteil, immer mehr unserer Projekte wickeln wir nicht mehr auf „alten" TFS-Servern ab, sondern verwenden Azure DevOps mit den hier aufgezeigten Möglichkeiten und betreiben Test- und häufig auch Produktionsumgebungen auf der Microsoft-Azure-Plattform.

Microsofts Plan scheint also – zumindest in unserer Wahrnehmung – aufzugehen, und wir als Teil der Entwicklercommunity freuen uns über die vielen neuen Möglichkeiten.

Nico Orschel ist Principal Consultant, Autor und Referent im Bereich DevOps und ALM bei der AIT GmbH & Co. KG Stuttgart und wurde von Microsoft als Most Valuable Professional (MVP) für Development Technologies ausgezeichnet. Er hilft Unternehmen, auf Basis von Microsoft Visual Studio Team Foundation Server/Microsoft Visual Studio Team Services effizienter Software zu entwickeln und zu testen und so ein höheres Qualitätsniveau bei kürzeren Releasezyklen zu erreichen

nico.orschel@aitgmbh.de

Thomas Rümmler arbeitet als Managing Consultant und Projektleiter bei AIT und ist von Microsoft als Most Valuable Professional (MVP) für Development Technologies ausgezeichnet worden. Sein Arbeitsschwerpunkt liegt auf Application-Lifecycle-Management und DevOps. Er hilft Unternehmen, ihren Entwicklungsprozess ganzheitlich zu verbessern. Seine Erfahrung gibt er als Autor des TFS-Blogs und Sprecher im Microsoft-DevOps-Umfeld weiter.

http://www.tfsblog.de/ thomas.ruemmler@aitgmbh.de

Links & Literatur

[1] https://aka.ms/ATRDownload

[2] https://marketplace.visualstudio.com/items?itemName=ms.vss-exploratorytesting-web

[3] https://blogs.msdn.microsoft.com/devops/2018/07/09/universal-packages-bring-large-generic-artifact-management-to-vsts/

.NET Framework, .NET Core und Mono sind tot – lang lebe .NET 5.0!

R.I.P .NET „Core“

Vom 6. – 8. Mai 2019 hielt Microsoft seine jährliche Entwicklerkonferenz Build in Seattle ab und bot einen Ausblick auf die Pläne für die kommenden Jahre – vor allem im Bereich .NET wird sich mächtig was tun.

von Dr. Holger Schwichtenberg

Für .NET-Entwickler gab es auf der Microsoft Build 2019 wieder einmal einen ordentlichen Paukenschlag: .NET 5.0 (kurz .NET 5) wird der gemeinsame Nachfolger der drei bisherigen .NET-Varianten (.NET Core, .NET Framework und Mono) sein. Alle .NET-Anwendungsarten von Desktop (WPF und Windows Forms) über Webserver (ASP.NET) und Webbrowser (Blazor/WebAssembly) bis zu Apps (UWP, Xamarin) und Spielen (Unity) sollen damit zukünftig eine gemeinsame .NET-Basis haben.

.NET 5 soll im November 2020 erscheinen und im Wesentlichen (aber nicht komplett) plattformneutral sein (**Abb. 2**). Einige Anwendungsarten, die auf .NET 5 basieren, werden nur auf bestimmten Plattformen laufen, z. B. UWP-Apps nur auf Windows 10, WPF- und Windows-Forms-Desktopanwendungen nur auf Windows ab Version 7. .NET 5 darf nicht verwechselt werden mit .NET Core 5 – dem Namen, den Microsoft für .NET Core 1.0 geplant hatte, bevor man sich entschloss, die Versionszählung wieder von vorne zu beginnen.

Nicht alle Features kommen in .NET 5

Technisch gesehen ist .NET 5 die Weiterführung von .NET Core (mit den zugehörigen Werkzeugen und Deployment-Optionen), in das große Teile (aber nicht alles) von .NET Framework und Mono einfließen (**Abb. 2**). Nach derzeitigem Stand werden in .NET 5 zum Beispiel Windows Workflow Foundation (WF) und Windows-Communication-Foundation-(WCF-)Server sowie Windows Forms aus Mono fehlen. Im Rahmen von .NET 5 wird aus Mono die statische Ahead-of-Time-(AOT-) Kompilierung als Alternative zur bisherigen Just-in-Time-(JIT-)Kompilierung in .NET einfließen, sodass .NET-Entwickler zukünftig zwischen JIT und AOT wählen können. AOT bietet .NET-Entwicklern kleinere Installationspakete und schnellere Anwendungsstarts, aber wegen der Vorkompilierung in Maschinencode eben keine plattformunabhängigen Installationspakete.

.NET 5 wird auch neue Funktionen beinhalten. Dazu gehört eine Integration mit anderen Frameworks und Programmiersprachen (Java, Objective-C und Swift) für die App-Entwicklung auf iOS und Android.

Der Begriff „Core“ verschwindet wieder

„.NET Core“ hatte Microsoft als Namen am 12.11.2014 für das schon am 13. Mai 2014 verkündete „cloud-optimized .NET Framework“ alias „Project K“ eingeführt. Version 1.0 erschien dann am 26. Juli 2016. Danach gab es die Versionen 1.1, 2.0, 2.1, 2.2. Version 3.0 von .NET Core soll im September 2019 erscheinen; auf der Build gab es die Preview-5-Version [1]; die Veröffentlichung des Release Candidate ist für Juli dieses Jahres geplant, dann soll es noch ein .NET Core 3.1 im November 2019 geben.

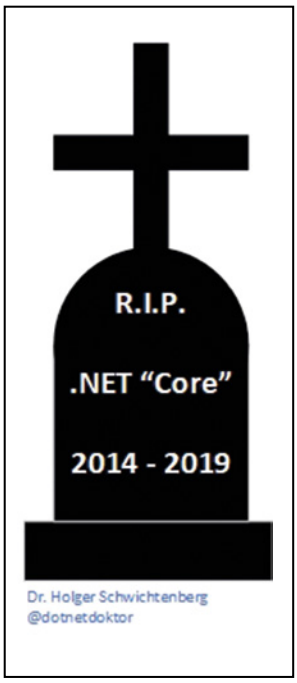


Abb. 1: Abschied von .NET Core

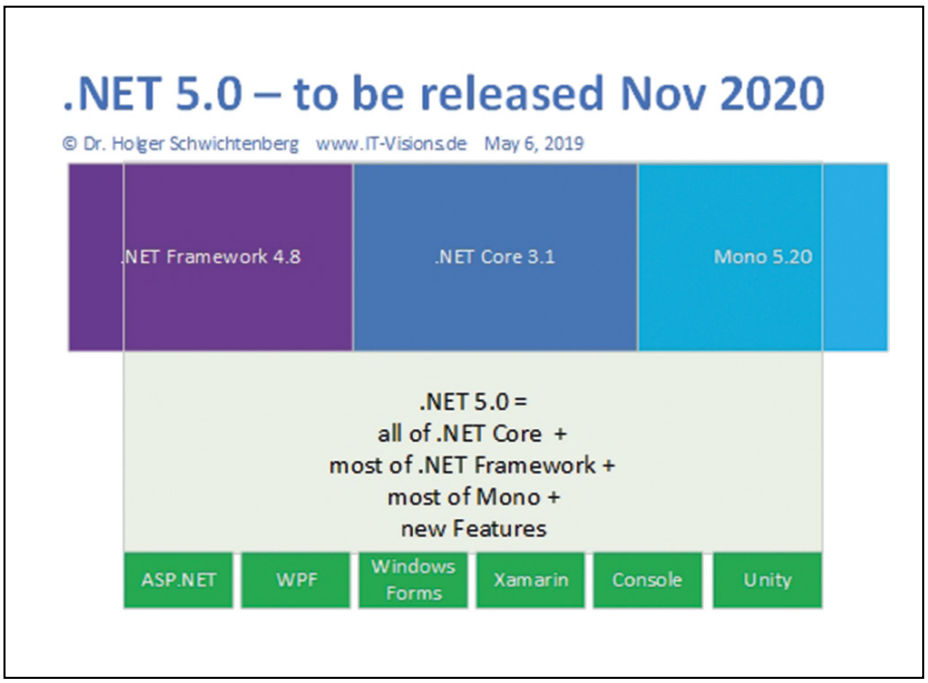


Abb. 2: .NET 5 im Verhältnis zu .NET Core, .NET Framework und Mono

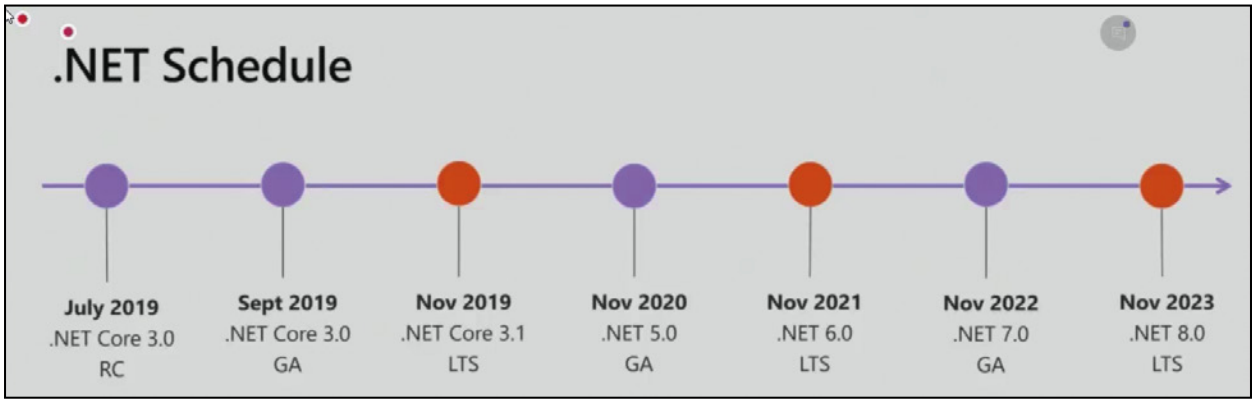


Abb. 3: Zeitplan für die kommenden .NET-Versionen (Quelle: Microsoft)

Damit soll dann die .NET-Core-Ära auch schon wieder enden (**Abb. 3**). Es wird keine weiteren Versionen mehr mit „Core" im Namen geben. Als Nächstes kommt .NET 5 (**Abb. 3**), denn Microsoft findet den Begriff inzwischen nicht mehr geeignet in der Kommunikation, gerade für neu in die .NET-Welt eintretende Softwareentwickler.

Mit .NET Core Version 2.0 hatte Microsoft begonnen, massiv Klassen aus dem .NET Framework nach .NET Core zu übernehmen. Man kehrte damit von dem .NET-Core-1.x-Gedanken ab, das neue .NET Core klein und aufgeräumt zu halten. Stattdessen stand nun auf der Agenda, möglichst kompatibel zum .NET Framework zu werden, damit bestehende klassische .NET-Framework-Anwendungen auf .NET Core umziehen können. In Version 2.1 führte Microsoft diese Trendwende mit dem Windows Compatibility Pack (WCP) fort: Erstmals gab es in .NET Core nur Klassen, die ausschließlich auf Windows laufen (z. B. für die Registry-Programmierung). Dies wird verstärkt auch in .NET Core 3.0 gelten: Dort gibt es zwar dann die GUI-Frameworks WPF und Windows Forms, aber sie sind weiterhin nicht plattformunabhängig, sondern an Windows gebunden. Der Sinn von WPF und Windows Forms liegt darin, dass Softwareentwickler bestehende .NET-Framework-basierte WPF- und Windows-Forms-Anwendungen auf .NET Core umziehen können. Auch in .NET 5 wird sich an dieser Plattformeinschränkung erst einmal nichts ändern.

.NET Core 3.0 liefert darüber hinaus .NET Standard 2.1, Unterstützung für gRPC und das neue ASP.NET-SignalR-basierte Webanwendungsmodell Server-side Blazor – nicht zu verwechseln mit dem WebAssembly-basierten Client-side Blazor, das zwar nun eine offizielle Vorschau ist, für das es aber weiterhin keinen Veröffentlichungstermin gibt.

.NET Framework 4.8 auf dem Abstellgleis

Softwareherstellern und Softwareentwicklern, die heute bestehende WPF- und Windows-Forms-Anwendungen besitzen oder neu planen, machte Microsoft auf der Build-Konferenz klar, was sie schon am 4. 10. 2018 in einem Blogeintrag [2] angedeutet hatten: Die am 18. April 2019 erschienene Version 4.8 ist die letzte Version des .NET Frameworks, die signifikante neue Funktionen bieten wird. Alles, was danach noch an Updates kommen wird, wird lediglich Sicherheitslücken und kritische Softwarefehler betreffen. Vielleicht wird das ein oder andere Netzwerkprotokoll noch auf den neuesten Stand gebracht, das wars dann aber.

Deterministische Erscheinungstermine

Microsoft will bei .NET 5 bei der in .NET Core etablierten, agilen und transparenten Entwicklungsweise eines Open-Source-Projekts auf GitHub bleiben. Einen Unterschied soll es aber zu .NET Core geben: Anstelle der bisherigen unregelmäßigen Erscheinungstermine soll es jedes Jahr im November eine neue Hauptversion mit Breaking Changes geben, d. h. nach .NET 5 im November 2020 gibt es dann .NET 6 im November 2021, .NET 7 im darauffolgenden Jahr usw. (**Abb. 3**).

Zwischendurch sind Unterversionen mit neuen Features, aber ohne Breaking Changes bei Bedarf angedacht. Jede zweite Version soll eine Long-Term-Support-(LTS-) Version mit drei Jahren Unterstützung durch Microsoft sein [3]. .NET Core 3.0 wird genau wie .NET Core 2.0 keine LTS-Version sein; die auf .NET Core 2.1 folgende LTS-Version wird .NET Core 3.1 sein. Dann geht es in der LTS-Linie erst 2021 weiter. Weitere Neuigkeiten von der Microsoft Build:

Windows

- Ein verbessertes Linux-Subsystem in Windows (WSL) bietet schnellere Dateisystemoperationen sowie die Unterstützung für Linux-basierte Docker-Container.
- Microsoft wird seinem Windows-Betriebssystem endlich ein neues Kommandoeingabefenster (Windows Terminal) mit zeitgemäßer Zeichensatzunterstützung inklusive Emoticons, Registerkarten, Layoutthemen und einem Erweiterungsmodell mit eigenem Marktplatz spendieren. Das Windows Terminal wird für die klassische CMD, die PowerShell und das Shell für Windows for Linux (WSL) zur Verfügung gestellt werden. Die erste Version soll es aber erst im Juni 2019 geben.
- Der neue Chromium-basierte Microsoft-Edge-Webbrowser erhält einen Kompatibilitätsmodus zum Internet Explorer 11 (IE Mode). Unternehmen, die auf Funktionen des Internet Explorers für alte Intranetanwendungen angewiesen sind (z. B. ActiveX), sollen damit auf Edge umsteigen können.

Werkzeuge

- Von Visual Studio 2019 gibt es eine dritte Vorschauveröffentlichung auf Version 16.1 [4]. Die bisher eigenständige IntelliCode-Erweiterung ist nun für die Sprachen C# und XAML enthalten. Auch die GitHub-Erweiterungen gehören nun zum Standard. Ebenso soll die Performance für C++- und .NET-Entwickler in einigen Punkten verbessert sein. In C# gibt es nun IntelliSense auch für Typen, die noch nicht importiert sind.
- Die Visual-Studio-Abonnementoptionen werden erweitert um „Visual Studio Professional with GitHub Enterprise“ und „Visual Studio Enterprise with GitHub Enterprise“ im Rahmen von Enterprise Agreements (EA) [5].
- Entwickler können mit Visual Studio Online (VSO) im Browser coden. Für diesen Onlineeditor verwen-

Build 2019

Die Build-Konferenz fand vom 6. bis 8. Mai 2019 in Seattle statt. Während die Veranstaltung früher immer sehr schnell ausgebucht war, konnte Microsoft sie nun zum zweiten Mal in Folge nicht ganz füllen. Einige Teilnehmer nennen als Grund dafür, dass Microsoft nicht mehr wie früher Hardwaregeschenke verteile. Die Aufzeichnungen der Vorträge kann jedermann kostenfrei unter https://www.microsoft.com/en-us/build ansehen.

det Microsoft wieder den Namen VSO, den man zwischen dem 14. 9. 2011 und dem 13. 12. 2013 als Name für die heutigen Azure DevOps Services verwendete [6].

.NET

- Mit .NET Core 3.0 Preview 3 liefert Microsoft eine erste Version des „Single-File Bundlers" aus [7], die alle Dateien einer .NET-Core-Anwendungen in eine selbstentpackende Executable verpackt. Eine echte *.exe* mit AOT-Compiler soll es erst in .NET 5.0 geben.
- Die auf der Build 2018 angekündigten „XAML-Inseln" werden in Windows 10 mit dem Update im Mai 2019 verfügbar sein. Damit können Steuerelemente der Universal Windows Platform (UWP) in WPF und Windows Forms eingebunden werden (natürlich setzt dies Windows 10 voraus!) [8].
- Mit MSIX Core können Entwickler das MSIX-Installationsformat auch auf Windows-Versionen vor Windows 10 verwenden.
- Ergänzend zur Microsoft Authentication Library (MSAL) for JavaScript [9] sowie Objective-C für iOS [10] und Android [11] gibt es nun diese Hilfsbibliothek für Azure Active Directory auch für .NET.
- ML.NET, die auf der Build 2018 angekündigte Machine-Learning-Bibliothek für .NET, erreicht Version 1.0 [12].
- .NET-Programmierung ist nun auch dem Cluster-Computing-Framework Apache Spark möglich mit „.NET for Apache Spark".
- Xamarin Forms 4.0 gibt es als Public Preview, u. a. mit dem neuen *CollectionView*-Steuerelement, das *ListView* ersetzen soll, indem es schneller ist.

SQL Server

- Mit Azure SQL Database Edge [13] liefert Microsoft nun eine sowohl auf x64- als auch auf ARM-Systemen installierbare SQL-Server-basierte Datenbank mit niedrigen Systemanforderungen. Das API der Azure SQL Database Edge soll kompatibel zu SQL Server und SQL Azure sein.

DevOps

- Für Azure DevOps gibt es ein neues Preismodell. Es gibt keine Preiskategorien mehr: Wer mehr als fünf Benutzer hat, zahlt 6 Dollar pro Benutzer. Für in der Cloud gespeicherte Artefakte zahlt man zukünftig zwischen 2 und 0.25 Dollar pro Gigabyte, wobei die ersten zwei Gigabyte frei sind.
- Durch die Azure-Active-Directory-(AAD-)Unterstützung in GitHub können Entwickler nun Benutzergruppen zwischen dem AAD und GitHub synchronisieren [14].
- Auf der anderen Seite unterstützten die Webportale für Azure und Azure DevOps nun auch die Benutzeranmeldung mit GitHub-Benutzerkonten.
- Bisher konnten Azure-DevOps-Nutzer die neuen YAML-basierten Pipeline-Definitionen nur für Build Pipelines nutzen. Nun gibt es dieses Feature auch für Release-Pipelines. Ein einziges YAML-Dokument kann dabei sowohl die Build- als auch die Release-Pipeline enthalten. YAML-Pipeline-Definitionsdokumente können im Quellcodeverwaltungssystem eingecheckt sein. Das gibt den Entwicklern die Möglichkeit, Pipeline-Definitionen auf einfache Weise spezifisch für einzelne Branches zu erstellen.
- Für das Deployment auf Azure Kubernetes Services und Red Hat OpenShift gibt es nun Vorlagen in Azure DevOps, sowohl für Cloud als auch On-Premises.

Dr. Holger Schwichtenberg (MVP) alias „der Dotnet-Doktor", ist technischer Leiter des auf .NET und Webtechniken spezialisierten Beratungs- und Schulungsunternehmens www.IT-Visions.de. Er gehört durch zahlreiche Fachbücher zu den bekanntesten Experten für .NET und Visual Studio in Deutschland. Dr. Holger Schwichtenberg unterrichtet in Lehraufträgen an den Fachhochschulen Münster und Graz. Seit 1999 ist er Sprecher auf jeder BASTA!.

hs@IT-Visions.de www.dotnet-doktor.de

Links & Literatur

[1] https://dotnet.microsoft.com/download/dotnet-core/3.0

[2] https://msdn.microsoft.com/en-us/magazine/mt848631

[3] https://dotnet.microsoft.com/platform/support/policy/dotnet-core

[4] https://devblogs.microsoft.com/visualstudio/visual-studio-2019-version-16-1-preview-3/

[5] https://visualstudio.microsoft.com/subscriptions/visual-studio-github/

[6] https://devblogs.microsoft.com/visualstudio/intelligent-productivity-and-collaboration-from-anywhere/

[7] https://github.com/dotnet/core-setup/pull/5286

[8] https://docs.microsoft.com/de-de/windows/uwp/xaml-platform/xaml-host-controls

[9] https://github.com/AzureAD/microsoft-authentication-library-for-js

[10] https://github.com/AzureAD/microsoft-authentication-library-for-objc

[11] https://github.com/AzureAD/microsoft-authentication-library-for-Android

[12] https://github.com/dotnet/machinelearning/releases

[13] https://azure.microsoft.com/en-us/services/sql-database-edge/

[14] https://github.blog/2019-05-06-team-synchronization-across-github-and-azure-active-directory/

Hintergrund und Einstieg in ML mit .NET

Machine Learning für die Zukunft

Das Thema Machine Learning ist so präsent wie noch nie, obwohl es eigentlich alles andere als neu ist. Algorithmen haben in den letzten Jahren in immer mehr Bereiche unseres Lebens Einzug gehalten und sind wohl dabei, die Gesellschaft nachhaltig zu verändern. Doch was verbirgt sich hinter den Buzzwords KI und Machine Learning und wie können Entwickler sich diese Technologien zu eigen machen? Mit diesen Fragen beschäftigt sich der nachfolgende Artikel und gibt anhand praxisnaher Beispiele Antworten.

von Kevin Gerndt

Die Machine-Learning-Technologie hat mittlerweile große Teile unseres Alltags erobert und ihr Siegeszug scheint nicht mehr aufzuhalten zu sein. Ob Spracherkennung im Auto, persönlicher Assistent im Smartphone oder Fraud Detection beim Bezahlvorgang – nahezu überall verbirgt sich Machine Learning beziehungsweise künstliche Intelligenz. Glaubt man jedoch führenden Wissenschaftlern und Visionären, handelt es sich hierbei erst um den Anfang. Unumstritten ist, dass Machine Learning unser Leben und die Arbeitswelt, wie wir sie heute kennen, verändern wird. Laut einer Studie des Instituts für Arbeitsmarkt- und Berufsforschung (IAB) [1] sind Computer in der Lage, mindestens 70 Prozent der Tätigkeit von acht Millionen Beschäftigten zu erledigen. Dadurch könnten bis zum Jahr 2025 bis zu 1,5 Millionen Arbeitsplätze in Deutschland wegfallen. Doch diese Entwicklung macht den Menschen nicht komplett überflüssig, denn ohne Menschen bringt Machine Learning keinen Mehrwert. Nicht außeracht zu lassen ist, dass die KI-Revolution auch jede Menge hochwertige Jobs und neue Berufsbilder, wie etwa Data Engineers, Data Scientists oder Data Strategen erschaffen wird. Berufe mit sich ständig wiederholenden Arbeitsabläufen und Prozessen werden allerdings unweigerlich und zunehmend durch intelligente Algorithmen ersetzt. Anders als bei der ersten und zweiten Industriellen Revolution, bei der vorwiegend Jobs der Arbeiterklasse und in der Landwirtschaft ersetzt wurden, wird es diesmal auch Berufe wie etwa Buchhalter, Steuerprüfer und klassische Bürokräfte treffen.

Warum jetzt?

Doch was hat eigentlich diesen regelrechten KI-Boom der letzten Jahre ausgelöst, obwohl das Thema an sich doch schon über fünfzig Jahre alt ist? Im Wesentlichen stützt sich die überproportionale Entwicklung auf drei wichtige Säulen: Daten, Algorithmen und Rechenleistung. Aktuell erzeugen über drei Milliarden Webnutzer sowie die unzähligen, vernetzten Smart- und IOT-Geräte jeden Tag etwa 2,5 Trillionen Bytes an Daten. Konkret bedeutet dies, dass 90 Prozent aller verfügbaren Daten in den letzten zwei Jahren entstanden sind. Ein autonom fahrendes Fahrzeug erzeugt pro Stunde eine gigantische Datenmenge von vier Terabytes. Ein selbstlernendes Computersystem benötigt eine enorm große Menge an Daten, die vor zehn Jahren einfach noch nicht zur Verfügung stand, zum Beispiel, wenn es um das Erkennen von Objekten wie etwa Personen, Autos oder Verkehrszeichen geht. Für die schnelle und zuverlässige Identifizierung von Objekten bedarf es wiederum effizienter Algorithmen. Auch in dieser Hinsicht hat die Wissenschaft in den letzten Jahren enorme Fortschritte vermelden können. Ein Schlagwort in diesem Zusammenhang ist Deep Learning. Neben großen Datenmengen und effizienten Algorithmen wird noch eine dritte Komponente benötigt: Rechenleistung. Das immense Cloud-Wachstum der letzten Jahre hat dazu geführt,

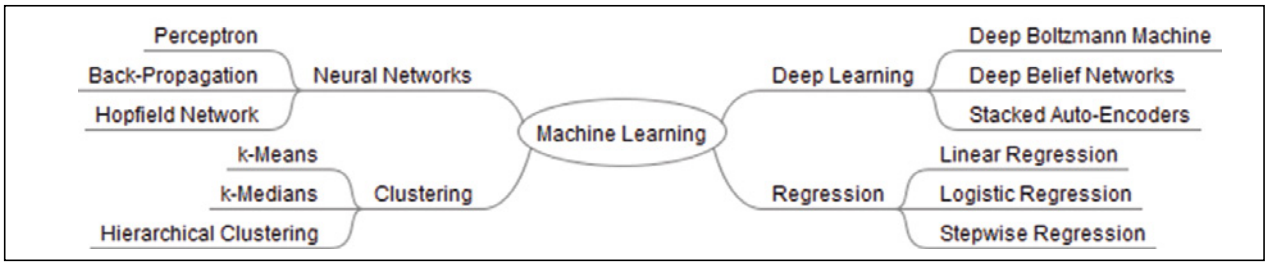


Abb. 1: Maschine-Learning-Übersicht

dass schier unendliche Rechenpower durch Cloud-Dienste zur Verfügung gestellt und abgerufen werden kann. Cloud-Anbieter wie Google, Microsoft und Amazon ermöglichen es Firmen oder Privatleuten, gigantische Rechenleistung für einen benötigten Zeitraum abzurufen, ohne dass erst in das eigene Rechenzentrum investiert werden muss. Aber auch Hardwarehersteller wie etwa Nvidia oder Intel haben den Trend unlängst erkannt und spezielle Hardware für die Verarbeitung von KI-Operationen entwickelt und optimiert. Die Basis für eine Welt voller Algorithmen und Automation scheint somit perfekt. Um jedoch moderne Applikationen und künstliche Intelligenz miteinander zu verknüpfen, bedarf es einer weiteren wichtigen Komponente: des Entwicklers. Im Folgenden gibt es deshalb einen Überblick über das Thema Machine Learning und es wird gezeigt, mit Hilfe welcher Werkzeuge sich künstliche Intelligenz in Softwarelösungen integrieren lässt.

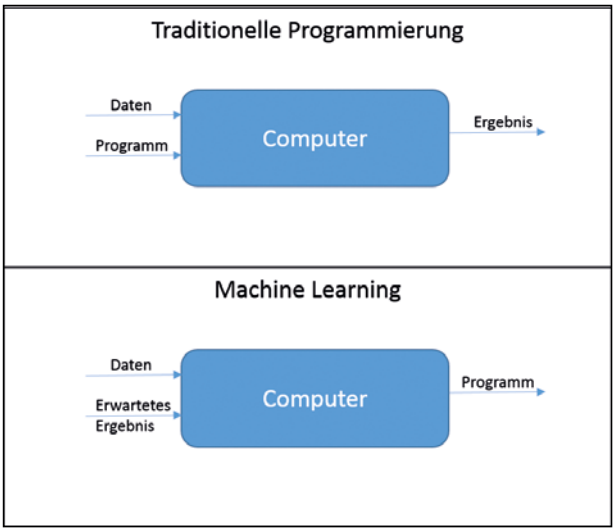


Abb. 2: Klassische Programmierung vs. Maschine Learning

Machine Learning, Deep Learning oder künstliche Intelligenz?

Künstliche Intelligenz ist wohl jedem ein Begriff, schließlich ist es ein beliebter Schwerpunkt unzähliger Kinofilme wie etwa „Matrix“, „Terminator“ oder „I, Robot“. Immer häufiger kann man aber auch in der Presse von künstlicher Intelligenz, neuronalen Netzen, Deep Learning oder Machine Learning lesen. Wer sich noch nicht intensiver mit der Materie beschäftigt hat, mag die Bezeichnungen womöglich für Synonyme halten. Um das Thema zu durchdringen, ist es zunächst erforderlich, die Unterschiede und Zusammenhänge zu verstehen. Hierbei soll die in **Abbildung 1** gezeigte Mindmap helfen.

Machine Learning ist der Oberbegriff und letztendlich nichts anderes als eine Sammlung mathematischer Methoden. Darunter angeordnet sind verschiedene Disziplinen, die wiederum dazu dienen, spezifische Probleme zu lösen. Die hier dargestellte Mindmap zeigt nur einen kleinen Ausschnitt des großen Ganzen. Zu jeder Problemstellung gibt es eine Vielzahl verschiedener Algorithmen, die im Laufe der Jahre entwickelt wurden. Diese Algorithmen sind der Schlüssel zu Machine Learning und künstlicher Intelligenz. Soll zum Beispiel ein einfaches Vorhersagemodell erstellt werden, eignet sich am besten ein Regressionsverfahren, für das Erkennen von Objekten hingegen ein Deep-Learning-Algorithmus. Doch wie ist ein Computer dazu in der Lage zu lernen? Damit eine Maschine in die Lage versetzt werden kann zu lernen, sind in erster Linie große Datenmengen erforderlich. Am besten lässt sich das anhand des Beispiels der Objekterkennung erklären. In der sogenannten Trainingsphase erhält eine Software eine gigantische Menge von Bildern mit zum Beispiel Verkehrssituationen. Die Software versucht nun, anhand eines Algorithmus ähnliche Objekte auf den Bildern zu identifizieren. Ein Stoppschild verfügt über charakteristische Merkmale wie etwa die Form, die Aufschrift und die Farbe. Der Computer weiß natürlich nicht um die Bedeutung des Stoppschilds, aber er erkennt das Muster. Im nächsten Schritt der Klassifizierung wird das Objekt mit einem sogenannten Label versehen. Somit ist nun bekannt, dass es sich um ein Stoppschild handelt. Mittels einer Programmierung könnte nun beispielsweise im Szenario des autonomen Fahrens das Fahrzeug angewiesen werden zu stoppen, wenn ein Stoppschild erkannt wird. An diesem Beispiel lässt sich der Unterschied zwischen klassischen Programmiermodellen und Machine Learning sehr gut veranschaulichen. Denn genauso gut wäre es natürlich denkbar und möglich, eine Routine zu entwickeln, die auf einem Bild speziell nach den charakteristischen Merkmalen eines Stoppschilds sucht. Jedoch wäre der Aufwand viel höher und der Algorithmus nicht in der Lage, durch eine größere Menge an Daten die Erkennung zu verbessern (**Abb. 2**).

Supervised und Unsupervised Learning

Im Bereich des Machine Learning werden im Wesentlichen zwei Kategorien des Lernens unterschieden:

Supervised Learning (überwachtes Lernen) und Unsupervised Learning (unüberwachtes Lernen). Beim Supervised Learning, der gebräuchlicheren Variante, wird der Mensch dazu benötigt, dem Algorithmus Schlussfolgerungen beizubringen. Diese Art des Lernens kommt der menschlichen Art des Lernens sehr nahe. Soll ein Programm zum Beispiel das Erkennen von Katzen erlernen, wird in der Trainingsphase eine große Menge mit Labels versehener Katzenbilder eingelesen. Nach dem Lernprozess erfolgt die Prüfung auf Korrektheit in der Erkennung durch eine Testphase, bei der wiederum Bilder ohne Label verarbeitet werden. Die Erwartungshaltung ist, dass das Programm Katzen zu einem möglichst hohen Prozentsatz identifiziert. Auch wenn die Unterscheidung zwischen Hund und Katze für Menschen schon im frühen Kindesalter keine große Herausforderung darstellt, bedarf es beim Anlernen eines Modells etwas mehr Geduld und mehrfacher Optimierungsvorgänge. Google schafft mittlerweile mit einem Identifizierungsalgorithmus immerhin eine Trefferquote von 75 Prozent. Ein weiteres Anwendungsgebiet des überwachten Lernens sind Regressionsprobleme, auf die im weiteren Verlauf des Artikels noch eingegangen wird. Mit Hilfe dieser statistischen Verfahren lässt sich zum Beispiel das Gehalt oder Gewicht einer Person anhand von Abhängigkeitsdaten vorhersagen.

Der häufigste Anwendungsfall für unbeaufsichtigtes Lernen ist wohl das sogenannte Clustering. Es dient, wie der Begriff schon vermuten lässt, dem Bilden von Gruppierungen, um große Datenmengen zu visualisieren. Am einfachsten kann Clustering mit einem Beispiel erklärt werden. Ein Algorithmus analysiert

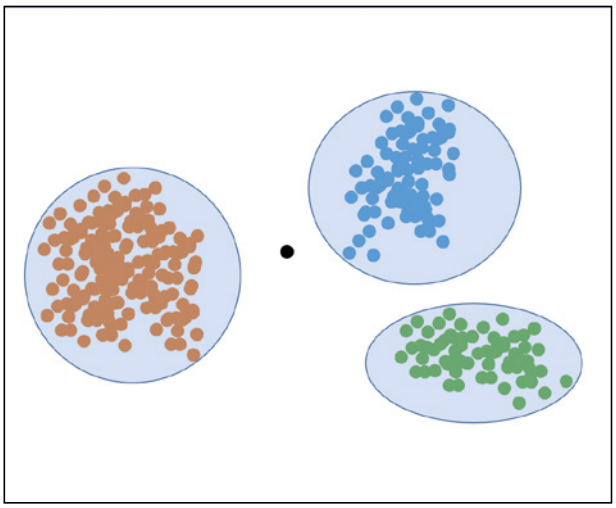


Abb. 3: Clustering

mehrere Millionen Wählerdaten mit einer Reihe verschiedener Parameter wie etwa Einkommen, Alter, Geschlecht und gewählter Partei. Ziel ist es, das wahrscheinliche Wahlverhalten eines Wählers nun anhand von Einkommen, Alter und Geschlecht zu ermitteln. Um dieses Ziel zu erreichen, werden mittels eines Algorithmus zunächst Cluster oder Gruppierungen gebildet (**Abb. 3**).

Um die Darstellung zu vereinfachen, ist diese auf zwei Dimensionen reduziert und gleicht einem Idealszenario. Die Herausforderung des Clusterings wird jedoch an dem schwarzen Ausreißer erkennbar. Um diese Problemstellung zu lösen, existiert eine Vielzahl an Algorithmen. Eines der einfachsten mathematischen Verfahren ist der K-Means-Algorithmus. Bei diesem werden zunächst *K* Zufallspunkte generiert, um die wiederum die Daten gruppiert werden. Die Anzahl der Zufallspunkte *K* entspricht in diesem Fall der Anzahl von Parteien, was die Clusterbildung enorm vereinfacht. Daraus wiederum ergibt sich auch direkt der erste Nachteil dieses Vorgehens, denn es wird immer zwingend davon ausgegangen, dass ein unbekannter Wähler einer der existierenden Parteien (Cluster) zugeordnet werden kann. Um eine Zuordnung zu erreichen, wird für jeden Datenpunkt die euklidische Distanz zum sogenannten Centroid errechnet. Der Vorgang zur Berechnung der Clusterzentren und der Zuordnung der Datenpunkte wiederholt sich so lange, bis sich die Zentren nicht mehr oder nur noch minimal ändern. Anhand der eingegebenen Daten konnten in diesem Szenario drei Cluster ermittelt werden. Die Aufgabe besteht nun darin, den schwarzen Punkt ohne bekanntes Parteienmerkmal einem Cluster zuzuordnen. Auf den ersten Blick kann ausgeschlossen werden, dass der schwarze Punkt zum grünen Cluster gehört. Die eindeutige Zuordnung zwischen dem orangen und blauen Cluster gestaltet sich für das menschliche Auge schon etwas schwieriger. Für den Algorithmus hingegen ist es nur die logische Zuordnung zum nächsten Clusterzentrum.

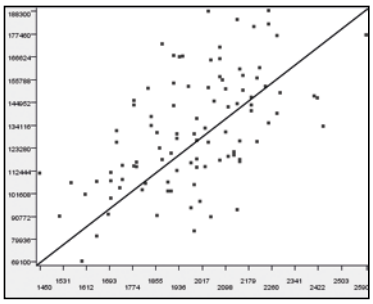


Abb. 4: Visualisierung der Eingangsdaten

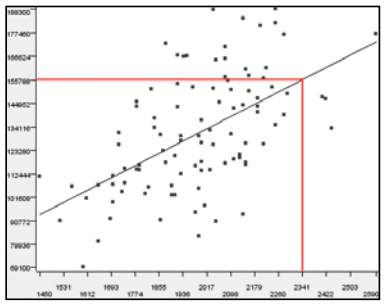


Abb. 5: Line of Best Fit

Regression mit ML.NET

Eine gute Einstiegsmöglichkeit, um mit der Entwicklung eigener Modelle oder der Integration von Maschine Learning in Software zu starten, bietet das Open-Source-Machine-Learning-Framework ML.NET von Microsoft. Ein großer Vorteil ist die einfache Handhabung. Damit ist eine Verwendung der Bibliothek ohne fortgeschrittene Kenntnisse im Bereich des Maschine Learnings möglich. Die Bibliothek wurde ursprünglich von dem Microsoft-Forschungsbereich Microsoft Research entwickelt und kommt in vielen Microsoft-Produkten wie etwa Bing oder der Office-Produktpalette zum Einsatz. Mit der ML.NET-Bibliothek lassen sich Aufgaben wie etwa Klassifizierung, Sentimentanalysen oder auch das Entwickeln von Vorhersagemodellen (Regression) einfach lösen. Ein weiterer Vorteil besteht in der lokalen Verwendung. Es ist also nicht erforderlich, einen Account bei einem Cloud-Dienst wie Azure zu erstellen und für die benötigten Ressourcen und Rechenleistung zu bezahlen. Auf der anderen Seite kann sich eine Trainingsphase aufgrund der mangelnden Rechenleistung dadurch natürlich stark ausdehnen. Im Nachfolgenden wird anhand eines Beispiels erklärt, wie sich ein Vorhersagemodell zur Bestimmung von Wohnungspreisen anhand verschiedener Kriterien wie etwa Größe, Anzahl der Bäder und Zimmer sowie das Vorhandensein eines Balkons mittels ML.NET entwickeln lässt. Um eine Lösung für dieses Problem erarbeiten zu können, ist es zunächst wichtig zu verstehen, um was für eine Art Problem es sich handelt. Da in diesem Fall anhand verschiedener Eingangsparameter (Fläche, Anzahl der Zimmer etc.) ein Preis ermittelt werden soll, handelt es sich um ein sogenanntes Regressionsproblem. Für dessen Lösung wird auf die Regressionsanalyse, also ein statistisches Verfahren zurückgegriffen, die das Ziel verfolgt, die Beziehung zwischen einer abhängigen und einer oder mehreren unabhängigen Variablen zu modellieren. Die Vorgehensweise lässt sich zunächst am besten anhand eines vereinfachten Models mit zwei Variablen erklären. Die Eingangsparameter werden auch als Features bezeichnet. Wird ein Modell durch das Entfernen von Features vereinfacht, wird häufig von Featurereduktion gesprochen. Der Preis einer Wohnung soll aufgrund der Größe bestimmt werden. Zwischen diesen zwei Größen besteht eine Korrelation: je größer die Wohnung, desto höher der Preis. Jedoch ist eine Wohnung mit 200 Quadratmetern nicht genau doppelt so teuer wie eine Wohnung mit 100 Quadratmetern. Um nun ein Vorhersagemodell basierend auf einer linearen Regression entwickeln zu können, wird zunächst ein Datensatz bestehend aus den Informationen Größe und Preis benötigt. Um die Beziehung der zwei Variablen zueinander besser zu verstehen, bietet es sich an, die Daten zunächst zu visualisieren. Das kann zum Beispiel mittels eines Datenanalysetools wie KNIME erfolgen. Aus dieser Darstellung (**Abb. 4**) geht nun ganz klar hervor, dass die Daten zwar linear sind, aber nicht der Funktion $f(x)=x$ entsprechen.

Um ein aussagekräftiges Modell entwickeln zu können, muss die Linie gefunden werden, die die Beziehung zwischen Wohnfläche und Preis am ehesten widerspiegelt. Hierfür kommen die beiden Verfahren lineare Regression und Gradient Descent zum Einsatz. Vereinfacht gesagt wird zunächst ein Fehlerwert berechnet, der sich aus der Summe der Abstände zwischen den einzelnen Datenpunkten und der festgelegten Linie errechnet. Um den kleinsten Fehlerwert zu finden, kommt Gradient Descent, ein Verfahren zur Optimierung einer Funktion, zum Einsatz. Bei dieser Vorgehensweise wird bildlich gesprochen die Linie solange verschoben, bis der kleinstmögliche Fehlerwert erreicht ist. Bei jedem dieser Durchläufe wird mittels der Learning Rate bestimmt, wie schnell sich der Algorithmus dem Ziel nähert. Genau hier verbirgt sich auch die Krux, denn je niedriger die Learning Rate, desto mehr Durchläufe werden benötigt und somit verlängert sich auch die Trainingsphase. Bei einer zu hoch angesetzten Learning Rate kann es wiederum passieren, dass die optimale Funktion nicht gefunden wird. Die Line of Best Fit für das hier skizzierte Beispiel ist in **Abbildung 5** dargestellt.

Mit Hilfe des trainierten Models lässt sich nun herausfinden, dass eine Wohnung mit 2 341 Square Feet ca. 156 000 Pfund kostet. Allerdings ist der Wert nur

Price	SqFt	Bedrooms	Bathrooms	Offers	Brick
114300	1790	2	2	2	False
114200	2030	4	2	3	False
114800	1740	3	2	1	False
…	…	…	…	…	…

Tabelle 1: Analyse der Trainingsdaten

wenig verlässlich, da einige zuvor entfernte Einflussfaktoren, wie etwa ob die Wohnung einen Balkon hat, die Anzahl der Zimmer usw., entfernt wurden. Ein wichtiges Kriterium für die Verlässlichkeit eines Modells ist somit die Auswahl der richtigen Features. Zum Beispiel bestimmt das Vorhandensein eines Balkons den Preis mit hoher Wahrscheinlichkeit. Die Wandfarbe in der Wohnung hingegen spielt eine sehr geringe bis gar keine Rolle bei der Preisfindung. Wird diese jedoch in die Liste der Features mit aufgenommen, kann es passieren, dass der Algorithmus ungewünschte Korrelationen in den Daten findet, die aber eigentlich gar keine Relevanz haben. Bei diesem Effekt wird häufig von Overfitting gesprochen. Mit zunehmender Anzahl von Features wird allerdings auch die Darstellung des Models immer komplizierter, weshalb sich ein solches Szenario am besten unter Zuhilfenahme einer Bibliothek wie ML.NET realisieren lässt. Um mit der Entwicklung starten zu können, muss zunächst ein neues Projekt auf Basis des Visual-Studio-Templates Console App (.NET Core) angelegt werden. Anschließend ist es erforderlich, mittels des NuGet Package Managers das Paket Microsoft.ML in der letzten vorliegenden Version zu installieren (*Install-Package Microsoft.ML*). Die Bibliothek unterstützt ausschließlich Programme auf 64-Bit-Architektur-Basis, weshalb es notwendig ist, die Zielplattform in den Projekteigenschaften auf x64 festzulegen. Um ein strukturiertes Einlesen der Trainingsdaten, die in Tabelle 1 exemplarisch dargestellt sind, zu ermöglichen, muss eine Modellklasse auf Basis der vorliegenden Daten entwickelt werden.

Diese beinhaltet Eigenschaften für alle im Trainingsdatenset enthaltenen Spalten und bildet diese in der vorliegenden Reihenfolge mittels des *Column*-Attributs ab (Listing 1). Mit der zusätzlichen *name*-Eigenschaft und der Bezeichnung *Label* wird das Feature markiert, das die standardmäßig korrekten Werte für die Vorhersage enthält.

Eine genaue Spezifikation dessen, was bestimmt werden soll, findet sich in der Klasse *HousePricePrediction*. Für die Regressionsaufgabe enthält die Spalte *Score* die vorhergesagten Bezeichnungswerte. Nachdem nun Klarheit über die Datenstruktur herrscht, ist es an der Zeit, das Model zu trainieren. Die Basis hierfür stellt die *LearningPipeline*-Klasse der ML.NET-Bibliothek dar, die sich im Namespace *Microsoft.ML.Legacy* befindet (Listing 2). Mittels der Learning-Pipeline wird zunächst die Quelldatei für die Trainingsdaten festgelegt. Da der Algorithmus, der das Model trainiert, numerische Fea-

Listing 1

```
public class House
  {
    [Column("0", name: "Label")]
    public float Price;

    [Column("1")]
    public float SqFt;

    [Column("2")]
    public float Bedrooms;

    [Column("3")]
    public float Bathrooms;

    [Column("4")]
    public float Offers;

    [Column("5")]
    public bool Brick;
  }

public class HousePricePrediction
  {
    [ColumnName("Score")]
    public float Price;
  }
```

Listing 2

```
public static async Task<PredictionModel<House, HousePricePrediction>>
Train()
  {
    string _datapath = Path.Combine(
      Environment.CurrentDirectory, "Data", "house-prices-train.csv");

    var pipeline = new LearningPipeline
    {
      new TextLoader(_datapath).CreateFrom<House>
      (useHeader: true, separator: ','),
      new CategoricalOneHotVectorizer(
        "Brick"),
      new ColumnConcatenator(
        "Features",
        "SqFt",
        "Bedrooms",
        "Bathrooms",
        "Offers",
        "Brick"),
      new Microsoft.ML.Legacy.Trainers.FastTreeTweedieRegressor()
    };

    PredictionModel<House, HousePricePrediction> model =
    pipeline.Train<House, HousePricePrediction>();

    string _modelpath = Path.Combine(
    Environment.CurrentDirectory, "Data", "Model.zip");

    await model.WriteAsync(_modelpath);

    return model;
  }
```

tures erfordert, muss das Boolean-Feld *Brick* mittels des *CategoricalOneHotVectorizer* zunächst transformiert werden. Die Transformationsklasse *ColumnConcatenator* dient zur Vorbereitung der Daten und überführt die Werte aller angegebenen Spalten in das Feld *Features*. Für das Training des Models kommt ein sehr effizienter, entscheidungsbaumbasierter Algorithmus zum Einsatz, dessen Implementierung sich in der Klasse *FastTreeTweedieRegressor* befindet. Alternativ ließe sich auch die Implementierung der Klasse *FastTreeRegressor* verwenden. Während der Tests hat sich jedoch der *FastTreeTweedieRegressor* als genauer in der Vorhersage erwiesen. Die Daten des trainierten Models werden abschließend durch einen asynchronen Aufruf der Methode *WriteAsync* in das Archiv *Model.zip* gespeichert.

Nachdem jetzt ein trainiertes Model vorliegt, ist es natürlich interessant zu wissen, wie exakt die Vorhersage nun funktioniert. Um das herauszufinden, wird ein Gegentest mit Testdaten durchgeführt. Für das Ermitteln dieser Daten existieren verschiedene Herangehensweisen. Im einfachsten Fall wird der vorliegende Datensatz nach dem 80/20-Prinzip aufgeteilt – 80 Prozent Trainingsdaten und 20 Prozent Testdaten. Bei der Extraktion der Evaluierungsdaten sollte darauf geachtet werden, sie nach dem Zufallsprinzip auszuwählen, um das Bilden falscher Korrelationen durch aufeinanderfolgende Datenreihen zu vermeiden. Um das Modell zu verifizieren, wird zunächst die Testdatendatei mittels der *TextLoader*-Klasse eingelesen (Listing 3). Den Rest übernimmt die *Evaluate*-Methode des *RegressionEvaluator*-Objekts. Aus dem Ergebnis lassen sich verschiedene Metriken abrufen, mit der sich die Exaktheit bestimmen lässt. Ein Indikator für die Exaktheit der Vorhersage ist der sogenannte Root Mean Squared Error (RMS).

Listing 3

```
private static void Evaluate(PredictionModel<House,
HousePricePrediction> model)
 {
  string _testdatapath = Path.Combine(
  Environment.CurrentDirectory, "Data", "house-prices-test.csv");

  var testData = new TextLoader(_testdatapath)
  .CreateFrom<House>(useHeader: true, separator: ',');

  var evaluator = new RegressionEvaluator();
  RegressionMetrics metrics = evaluator.Evaluate(model, testData);

  Console.WriteLine($"Rms = {metrics.Rms}");
 }
```

Listing 4

```
public static async Task Main(string[] args)
 {
  PredictionModel<House, HousePricePrediction> model = await Train();

  Evaluate(model);

  HousePricePrediction prediction = model.Predict(new House
  {
   SqFt = 1790,
   Bedrooms = 2,
   Bathrooms = 2,
   Offers = 2,
   Brick = false
  });

  Console.WriteLine("Predicted price: {0}.", prediction.Price);
  Console.Read();
 }
```

Listing 5

```
class ApiKeyServiceClientCredentials : ServiceClientCredentials
 {
  public override Task ProcessHttpRequestAsync
                                        (HttpRequestMessage request,
  CancellationToken cancellationToken)
  {
   request.Headers.Add("Ocp-Apim-Subscription-Key", "Your key");
   return base.ProcessHttpRequestAsync(request, cancellationToken);
  }
 }

static void Main(string[] args)
 {
  ITextAnalyticsClient client = new TextAnalyticsClient(new
  ApiKeyServiceClientCredentials())
  {
   Endpoint = https://westus.api.cognitive.microsoft.com
  };

  SentimentBatchResult sentimentBatchResult = client.
                                                SentimentAsync(
   new MultiLanguageBatchInput(
    new List<MultiLanguageInput>()
     {
      new MultiLanguageInput("en", "0", "I had a wonderful trip to
                    Seattle and enjoyed seeing the Space Needle!."),
  })).Result;

  foreach (var document in sentimentBatchResult.Documents)
  {
   Console.WriteLine($"Sentiment Score: {document.Score}");
  }

  Console.ReadLine();
 }
```

Dieser beziffert die Differenz zwischen den Werten des Modes und den Werten, die auf Basis der Testdaten ermittelt wurden. Je näher sich der RMS am Wert 0 befindet, desto besser ist die Leistung des Modells. Ein Wert von 0 dürfte jedoch in einem Real-World-Szenario nicht zu erreichen sein. Mit einer Handvoll (optimierter) Testdaten ergibt sich im hier gezeigten Beispiel eine Abweichung von 343,64 – ein fast zu guter Wert. Mit einer größeren Datenmenge, ausgewählt durch das oben beschriebene Vorgehen, fällt der Wert jedoch auch bedeutend höher aus.

Nachdem das Model nun trainiert und geprüft wurde, kommt der letzte Teil: die Datenvorhersage (Prediction). Der dafür notwendige Code ist in Listing 4 dargestellt. Da innerhalb der *Main*-Methode asynchrone Methodenaufrufe erfolgen, ist es notwendig, diese als *async Task* zu deklarieren. Damit das wiederum funktioniert, ist es erforderlich, in den erweiterten Build-Einstellungen des Projekts die Sprachversion auf C# 7.1 oder höher zu setzen. Der Vorhersagecode ist relativ einfach zu verstehen. Der *Predict*-Methode wird ein Objekt vom Typ *House* übergeben, welches mit Informationen zum Wohnobjekt gefüllt ist, jedoch nicht den Preis enthält, da dieser über den Algorithmus ermittelt wird. Die Methode wiederum liefert ein *HousePricePrediction*-Objekt zurück, über dessen *Price*-Eigenschaft der ermittelte Preis abgerufen werden kann.

Azure Cognitive Services

Mit den Azure Cognitive Services bietet Microsoft eine Reihe von APIs und SDKs an, die Entwicklern beim Erstellen intelligenter Anwendungen helfen, ohne dass diese spezielle KI- oder Data-Science-Kenntnisse besitzen müssen. Dadurch lassen sich Anwendungen leicht mit kognitiven Fähigkeiten wie etwa Emotions- und Videoerkennung, Gesichtserkennung, Spracherkennung und Sprachverständnis ausstatten. Ein einfach zu verstehendes Beispiel stellt die Sentimentanalyse dar, bei der das Stimmungsbild eines angegebenen Textes bestimmt wird. Mittels der Azure Cognitive Services lässt sich eine solche Anforderung mittels weniger Codezeilen und ohne spezielle Vorkenntnisse realisieren. Wie auch beim vorangegangenen ML.NET-Beispiel dient eine .NET-Core-Konsolenapplikation als Projektbasis. Die Cognitive Services lassen sich entweder direkt über das HTTP REST API ansprechen oder komfortabler mit Hilfe des SDK. Hierzu ist es erforderlich, mittels des NuGet-Package-Manager-Kommandos *Install-Package Microsoft.Azure.CognitiveServices.Language.TextAnalytics -Version 2.8.0-preview* die aktuelle Preview zu installieren. Der für die Sentimentanalyse notwendige Code ist in Listing 5 dargestellt. Zunächst wird ein neuer Client-Credential-Provider implementiert, der von der Klasse *ServiceClientCredentials* abzuleiten ist. Dieser dient lediglich dazu, den API Request durch Übermitteln des API Keys zu authentifizieren. Der notwendige API Key lässt sich einfach über den Cognitive-Services-Bereich des Azure-Portals generieren. Das setzt natürlich ein Azure-Konto voraus, das aber ebenfalls kostenlos eingerichtet werden kann. Der TextAnalyticsClient bietet eine Vielzahl von Möglichkeiten zur Textanalyse, wie etwa die Bestimmung der Sprache, das Extrahieren von Schlüsselbegriffen oder eben die Bewertung der Textstimmung. Beim Instanzieren muss diesem neben dem Client-Credential-Provider auch der Service-Endpunkt übergeben werden, der ebenfalls dem Azure Portal zu entnehmen ist. Danach ist der Client einsatzfähig und mittels des Methodenaufrufs *client.SentimentAsync(…)* lässt sich eine erste Sentimentation mit einem oder mehreren Texten ausführen. Abschließend kann über das Ergebnis iteriert und das Stimmungsbild in der Konsole ausgegeben werden.

Fazit

Maschine Learning ist schon jetzt nicht mehr aus unserem Alltag wegzudenken und wird immer schneller immer mehr Teile unseres Lebens erobern. Auch wenn das Thema stark gehypt wird, steckt es in vielen Bereichen immer noch in den Kinderschuhen. Ein Algorithmus, der Katzen auf einem Bild mit einer Wahrscheinlichkeit von 75 Prozent erkennt, ist beeindruckend, hat aber dennoch Optimierungspotential. Es ist ein bisschen wie mit den ersten Handys, diese stellten zwar eine Revolution dar, waren aber unhandlich und nicht wirklich praktikabel. Drei Jahrzehnte später trägt jeder einen leistungsstarken Minicomputer mit sich herum. Mit KI-Systemen ist die Wissenschaft jedoch schon jetzt an einem Punkt angekommen, an dem die Entwicklung überproportional voranschreiten wird. Auch die Integration von KI in neue oder bestehende Softwarelösungen ist eine immer häufiger aufkommende Anforderung. Als Entwickler muss man jedoch kein studierter Mathematiker sein, um solche Lösungen zu schaffen. Vielmehr kommt es, wie so häufig, darauf an, die richtigen Werkzeuge zu kennen und diese optimal einzusetzen. Dank einfach zu verwendender Frameworks wie etwa ML.NET oder der Cloud-Lösung Microsoft Azure Cognitive Services steht der Integration nichts mehr im Wege.

Kevin Gerndt arbeitet als Lead Consultant im Bereich Microsoft .NET Client und Web Technologies und hat während seiner beruflichen Laufbahn schon eine Vielzahl an Projekten begleitet. Er gilt als Experte auf dem Gebiet der modernen Web- und Softwarearchitektur und ist darüber hinaus als freiberuflicher Autor tätig, sowie regelmäßig auf namhaften Konferenzen vertreten. Schwerpunkte seiner Tätigkeit bilden die Analyse und Implementation von Geschäftsprozessen sowie das Konzipieren und Entwickeln moderner Softwarelösungen.

Links & Literatur

[1] http://doku.iab.de/kurzber/2018/kb0418.pdf

Zehn Hausaufgaben für die Cloud-Architektur – Eine gute Softwarearchitektur setzt klare Ziele voraus

Kolumne: Stropek as a Service

von Rainer Stropek

Softwarearchitekturen für Cloud-Lösungen zu entwickeln, ist seit Jahren fester Bestandteil meiner Arbeit. Ich werde von Firmen eingeladen, mit ihnen Architekturen für Cloud-basierende SaaS-Produkte zu entwickeln. Dabei stelle ich oft fest, dass Teams wichtige Hausaufgaben vernachlässigen. Die Projektziele und -rahmenbedingungen sind nicht klar formuliert. Man erwartet, dass Cloud-Anbieter wie Microsoft oder Berater wie ich eine allgemeingültige Cloud-Architektur aus dem Ärmel zaubern, die unabhängig vom Geschäftsmodell und der fachlichen Domäne funktionieren. Diese Vorgehensweise ist gefährlich. Sie führt zu unnötig komplexen oder unpassenden Architekturen. Es wäre, als würde man sein Haus planen, ohne vorher festgelegt zu haben, ob man einen Keller will oder wie viele Autos in der Garage Platz haben sollen. In dieser Ausgabe meiner Kolumne möchte ich eine Liste von zehn Hausaufgaben zusammenfassen, die ich Teams *vor* einem Cloud-Architekturworkshop mitgebe. Die Antworten sind die Basis, auf der eine solide Architekturentwicklung aufbauen kann.

1. Vision

Beginnen Sie mit einer Zusammenfassung der zentralen Vision, die hinter der Cloud-basierenden SaaS-Lösung steckt. Halten Sie sich dabei bewusst kurz. Idealerweise gibt es für das Projekt bereits ein Visionsdokument [1], auf das Sie verweisen können.

Eine Vision, die nur in den Köpfen der Projektbeteiligten existiert, ist zu wenig. Schreiben Sie sie nieder. Die Vision in prägnante Worte oder Schaubilder zu fassen, hilft, Inkonsistenzen und Unklarheiten aufzudecken.

Beschreiben Sie die Einordnung des Projekts in die Gesamtstrategie Ihrer Organisation. Ein Werkzeug, das ich in diesem Bereich gerne verwende, ist die Strategy Map [2]. Sie erklärt die primären strategischen Ziele und ihre Zusammenhänge. Stellen Sie für das SaaS-Projekt dar, wie es die Zielerreichung unterstützt.

2. Scope

Welchen Scope hat das Projekt, für das die Softwarearchitektur entwickelt werden soll? Was ist Teil der Aufgabenstellung für das vorliegende Projekt und was wird im Moment bewusst ausgeklammert? Wie ist das aktuelle Projekt in eine längerfristige Produkt-Roadmap eingegliedert?

3. Innovation

Damit der Fokus im Projekt richtig gesetzt werden kann, muss klar sein, was das Innovative am zu entwickelnden SaaS-Produkt ist. Ich empfehle, die Innovation aus zwei Perspektiven heraus zu beschreiben:

Externe Perspektive: Was sind aus Kundensicht die wichtigsten funktionalen und nichtfunktionalen Anforderungen, durch deren Lösung sich das SaaS-Angebot von existierenden Produkten abheben wird? Ein Werkzeug, das ich persönlich in diesem Bereich gerne einsetze, ist die Blue-Ocean-Strategie [3].

Interne Perspektive: Welche internen Aspekte des Unternehmens werden sich durch das SaaS-Angebot fundamental verändern? Betroffen sein könnten beispielsweise interne Geschäftsprozesse, die Aufbauorganisation oder die Unternehmensfinanzierung. Wie verändert das neue SaaS-Angebot die Wertschöpfungskette der Firma? Ich nutze bei dieser Frage gerne den Business Model Canvas als Hilfsmittel [4].

Hilfreich für die Qualität der Antwort auf diese Frage ist meiner Erfahrung nach eine Beschränkung auf jeweils maximal fünf Schlüsselinnovationen. Dadurch ist man gezwungen, sich auf das Wesentliche zu beschränken und zu priorisieren.

4. Zielumgebung

Ich habe in meinen Kolumnen schon oft darüber geschrieben, dass ein Produkt, das nur in der Cloud läuft, vielleicht sogar nur in einer bestimmten, per se nichts Schlechtes ist. Man verliert vielleicht einige Kunden, die auf jeden Fall eine On-Premises-Lösung suchen. Man

begibt sich auch in eine gewisse Abhängigkeit zum jeweiligen Cloud-Anbieter. Auf der anderen Seite profitiert man aber von qualitativ hochwertigen, günstigen PaaS- und Serverless-Diensten.

Die Architektur einer Cloud-Lösung wird massiv davon beeinflusst, ob man eine Cloud-native-Lösung entwickeln will und auf welche Cloud-Zielumgebung man sich konzentriert. Daher müssen vor der Architekturentwicklung insbesondere folgende Fragen geklärt werden:

- Muss es möglich sein, die Software in Kunden- oder Partnerrechenzentren (On Premises) zu installieren? Wenn ja, welche Anforderungen an das Zielrechenzentrum sind realistisch? Vermeiden Sie bei der Beantwortung dieser Frage technische Details (z. B. „ein Kubernetes-Cluster wird vorausgesetzt"), da dies Architekturentscheidungen schon vorwegnehmen könnte. Beschreiben Sie stattdessen den beim Kunden vorausgesetzten Reifegrad seiner IT-Infrastruktur (z. B. tiefgehende DevOps-Kenntnisse vs. kein eigenes Team mit fundiertem IT-Wissen).
- Falls es eine Cloud-native-Lösung wird: Ist ein Fokus auf einen gewissen Cloud-Anbieter gewünscht? Erklären Sie die Gründe für die Fokussierung (z. B. existierende strategische Partnerschaft). Beschreiben Sie allgemein den Unternehmensstandpunkt hinsichtlich Cloud Vendor Lock-in [5].
- Gibt es Einschränkungen (z. B. Region, ISO-Zertifizierungen, garantierte Verfügbarkeiten) für die Cloud-Zielumgebung, die sich aus internen oder externen Richtlinien (z. B. Konzernvorgaben, rechtliche Bestimmungen) ergeben?

5. Zielgruppe

Beschreiben Sie die Kundenzielgruppe für die SaaS-Lösung. Gehen Sie dabei insbesondere auf folgende Dinge ein:

- Beschreiben Sie die Kunden, die sie ansprechen wollen, möglichst konkret. Meiner Erfahrung nach hilft es, wenn man konkrete Personas [6] für Kunden (bei B2B-Projekten wären das Stakeholder, die den SaaS-Dienst nutzen oder über seine Nutzung entscheiden) entwickelt und beschreibt. Bei B2B-Lösungen gehen Sie auf Merkmale typischer Kundengruppen ein (z. B. Branche, Größe, Region) und bringen konkrete Beispiele für potenzielle Kunden.
- Beschreiben Sie die wichtigsten Anwendungsfälle je Persona. Ich empfehle, dass Sie sich ein Limit von fünf Fällen setzen, damit Sie gezwungen sind, zu priorisieren.
- Fügen Sie quantitative Kundenmerkmale hinzu (z. B. Anzahl Benutzer pro Kunde, Anzahl Transaktionen pro Kunde pro Zeiteinheit, Dauer der Nutzung der SaaS-Lösung je Benutzer pro Monat etc.).
- Erwähnen Sie eventuell wichtige technische Einschränkungen bei den Kunden (z. B. schlechte/keine Internetverbindung).

6. Skalierung

Legen Sie das Mengengerüst fest, an dem sich die Softwarearchitektur orientieren muss. Diese Frage ist speziell für Start-ups schwierig zu beantworten, da sich der Markterfolg nur schwer prognostizieren lässt. Dieser Aspekt beeinflusst die Architektur aber stark und daher ist eine Festlegung wichtig. Hier einige Aspekte, die bei der Beantwortung der Frage helfen können:

- Geben Sie Bandbreiten für die Anzahl an Kunden und Benutzer an. Es spricht nichts gegen mehrere Szenarien (z. B. optimistisch, realistisch, pessimistisch). Bei der Entwicklung der Architektur können Varianten erarbeitet werden, die unterschiedlich gut skalieren. Die endgültige Entscheidung kann dann durch Gegenüberstellung der erwarteten Kosten je Architekturvariante und Eintrittswahrscheinlichkeit des jeweiligen Szenarios getroffen werden.
- Strukturieren Sie das Mengengerüst nach Preisplänen (z. B. Kundenanzahl im Freemium-, Standard- und Premiumplan).
- Schätzen Sie den zeitlichen Verlauf des Mengengerüsts durch Schätzungen für Neukunden- und Abwanderungsquoten (Churn Rate) ab.
- Gehen Sie auf Besonderheiten im Nutzungsverhalten der Kunden ein (z. B. besonders hohe Last am Monatsende, fast keine Nutzung an Wochenenden, relative stabile Nutzung tagsüber).
- Schätzen Sie die Transaktionszahlen für die wichtigsten Geschäftsprozesse ab. Konzentrieren Sie sich auf die wichtigsten fünf, um sich nicht von Randthemen ablenken zu lassen.
- Wenn Sie das Mengengerüst nicht wirklich abschätzen können, nennen Sie zumindest Ober- und Untergrenzen (z. B. sicher nicht mehr als … Transaktionen pro Benutzer und Tag).

Beachten Sie, dass keine Aussage über das Mengengerüst keine Option ist. Irgendjemand muss eine Annahme treffen. Eine bewusste unternehmerische Entscheidung ist besser als eine implizite Festlegung durch Annahmen, die das Umsetzungsteam trifft.

7. Daten

Datenhaltung ist ein wichtiger Bestandteil einer Softwarearchitektur. Während es vor einigen Jahren fast immer auf eine relationale Datenbank hinauslief, gibt es heute in der Cloud eine Vielzahl an Speicherdiensten mit spezifischen Eigenschaften. Als Basis für die Wahl der richtigen Speichertechnologie sollten Sie insbesondere die folgenden Aspekte der Datenhaltung beschreiben:

- Mengengerüst für die wichtigsten Entitäten, gegliedert nach Kundentyp.
- Logisches Datenmodell aus fachlicher Sicht für die wichtigsten Entitäten. Vermeiden Sie dabei technische Details, da diese Architekturentscheidungen vorwegnehmen würden.
- Gehen Sie auf besondere nichtfunktionale Anforderungen ein (z. B. garantierte maximale Zugriffszeit bei gewissen Abfragen, Verfügbarkeitsgarantien, Datenwiederherstellung).

8. Geschäftsmodell

Ich bin in früheren Ausgaben dieser Kolumne schon darauf eingegangen, wie wichtig der Design-to-Cost-Ansatz für Cloud-basierende SaaS-Lösungen ist. Ausgangsbasis dafür sind Preis- und Kostenvorgaben, die für das jeweilige Projekt gelten. Wichtig sind insbesondere folgende Punkte:

- Beschreiben Sie das Preismodell (z. B. Preispläne, Mengenrabatte, Free- und Premium-Angebote).
- Beschreiben Sie Zusatzprodukte und Dienstleistungen rund um das SaaS-Angebot sowie deren Umsätze und Kosten (z. B. kostenpflichtiger Support, Einführungsprojekte, Self-Service-Portale).
- Legen Sie Kostengrenzen für Entwicklung und Betrieb des SaaS-Produkts fest (TCO, nicht nur Infrastrukturkosten), die sich aus dem Projektbudget oder der Start-up-Finanzierung ergeben.
- Wie beim Mengengerüst sollten Sie zumindest Ober- und Untergrenzen festlegen (z. B.: Wir werden nicht mehr verrechnen als …, der Standard-Preisplan wird mindestens … kosten).

Ich habe in diesem Bereich gute Erfahrung damit gemacht, Customer-Lifetime-Value-(CLV-)Kalkulationen [7] für verschiedene Szenarien durchzuführen. Sie machen meiner Erfahrung nach schnell klar, ob das Geschäftsmodell im Hinblick auf Umsatz- und Kostenstruktur tragfähig ist.

9. Vorhandene Ressourcen

In der heutigen Zeit sind die vorhandenen personellen Ressourcen genauso wichtig wie die finanziellen. Manchmal ist es sogar leichter, Geld aufzutreiben als Personal. Darauf muss in der Architekturentwicklung Rücksicht genommen werden. Sind die Ressourcen knapp, wird die Fertigungstiefe gering sein müssen und man greift mehr auf vorhandene Services und Komponenten zurück. Hat man ausreichend Leute, kann man es sich erlauben, mehr selbst herzustellen und dadurch auf individuelle Besonderheiten einzugehen oder Alleinstellungsmerkmale zu entwickeln.

Beschreiben Sie quantitative (Anzahl) und qualitative (vorhandenes Wissen) Aspekte des Personals, das für das Projekt zur Verfügung stehen wird.

10. Bestehende Komponenten

Beschreiben Sie, welche bestehenden Komponenten (z. B. existierende Codebasis, vorhandene Services) in der SaaS-Lösung weiterverwendet werden müssen. Gehen Sie auch auf Schnittstellen ein, die die SaaS-Lösung haben muss (z. B. System Context Diagram [8]). Vermeiden Sie, wie auch bei den zuvor genannten Punkten, technische Details. Es geht darum, die großen Zusammenhänge aufzuzeigen. Verweise auf externe Dokumente, in denen Details zu den vorhandenen Komponenten und notwendigen Schnittstellen zu finden sind, sind aber keineswegs von Nachteil.

Fazit

Eine allgemeine Softwarearchitektur, die nicht für ein spezifisches Problem entwickelt wird, ist selten besonders nützlich. Je klarer die Vorstellungen über die zu erreichenden Ziele sind, desto zielführender kann in der Architekturentwicklung gearbeitet werden. Ich hoffe, die oben genannten zehn Punkte helfen, Ihr nächstes SaaS-Projekt erfolgreich zu starten.

Rainer Stropek ist IT-Unternehmer, Softwareentwickler, Trainer, Autor und Vortragender im Microsoft-Umfeld. Er ist seit 2010 MVP für Microsoft Azure und entwickelt mit seinem Team die Zeiterfassung für die Dienstleistungsprofis time cockpit.

www.timecockpit.com

Links & Literatur

[1] https://en.wikipedia.org/wiki/Vision_document

[2] https://en.wikipedia.org/wiki/Strategy_map

[3] https://en.wikipedia.org/wiki/Blue_Ocean_Strategy#Concept

[4] https://en.wikipedia.org/wiki/Business_Model_Canvas

[5] https://en.wikipedia.org/wiki/Vendor_lock-in

[6] https://de.wikipedia.org/wiki/Persona_(Mensch-Computer-Interaktion)

[7] https://en.wikipedia.org/wiki/Customer_lifetime_value

[8] https://en.wikipedia.org/wiki/System_context_diagram

Nachhaltige Angular-Architekturen mit Nx und Strategic Design

Große Business-Apps mit Angular meistern

Strategic Design bietet eine etablierte Methodik zum Schneiden großer Anwendungen in möglichst autarke Domänen. Nx erlaubt die Umsetzung dieser Domänen in einem Angular Monorepo und stellt sicher, dass dabei möglichst wenig Abhängigkeiten entstehen.

von Manfred Steyer

Angular bietet sich aufgrund seiner Struktur und des gebotenen Leistungsumfangs für die Umsetzung von Frontends großer Softwaresysteme an. Um solche Systeme beherrschbar zu gestalten, gilt es, sie in kleine und wenig komplexe Module zu untergliedern; das ist soweit bekannt. Die Frage, die sich hierbei jedoch immer wieder aufdrängt, ist, nach welchen Kriterien der Modulschnitt erfolgen soll. Außerdem gilt es festzulegen, wie diese Module zu implementieren sind, aber auch, wie sie untereinander kommunizieren können.

Dieser Artikel gibt eine Antwort auf beide Fragen: Um eine Vorgehensweise für den Modulschnitt aufzuzeigen, beleuchtet er zunächst Strategic Domain Design. Dabei handelt es sich um eine Disziplin aus dem Umfeld von Domain-driven Design (DDD). Danach werden wir betrachten, wie sich eine damit gestaltete Architektur mit Nx [1] umsetzen lässt, einer populären und freien Erweiterung für das Angular CLI.

Domain-driven Design auf Metaebene

DDD beschreibt einen Ansatz, der eine Brücke zwischen Anforderungen an komplexe Softwaresysteme auf der einen und einem dazu passenden Anwendungsdesign auf der anderen Seite schlägt. Es lässt sich in die Teildisziplinen Tactical Design und Strategic Design untergliedern. Ersteres schlägt konkrete Denkweisen, Konzepte und Muster für ein objektorientiertes Design vor. Alternativ dazu existieren auch Ansätze, die die dahinterstehenden Ideen in die Welt der funktionalen Programmierung übertragen [2].

Strategic Design beschäftigt sich hingegen mit der Untergliederung eines großen Systems in einzelne Domänen und deren Ausgestaltung. Egal, ob man die meinungsstarke (engl. opinionated) Denkwelt von DDD

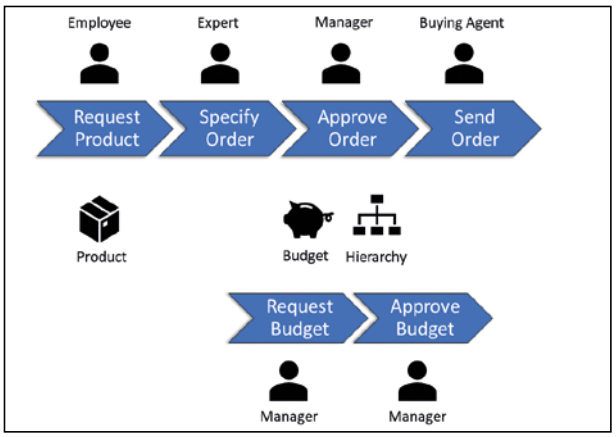


Abb. 1: Analyse von Abläufen zum Identifizieren von Domänen

gut findet oder nicht: Einige Ideen von Strategic Design haben sich bewährt, um ein System in kleinere, möglichst autarke Teile zu untergliedern. Es sind genau diese Ideen, die dieser Artikel aufgreift und im Kontext von Angular darstellt. Ob die restlichen Aspekte von DDD auch Berücksichtigung finden, ist hingegen für den Artikel unerheblich.

Modularisierung mit Strategic Domain Design

Ein Ziel von Strategic Design ist es, möglichst autarke (Sub-)Domänen zu identifizieren. Sie sind an einem spezifischen Vokabular zu erkennen. Sowohl Domänenexperten als auch Entwickler müssen dieses Vokabular rigoros verwenden. Das beugt Missverständnissen vor und führt dazu, dass die Anwendung den jeweiligen Fachbereich widerspiegelt. In diesem Sinn ist auch von einer allgegenwärtigen Sprache (engl. Ubiquitous Language) die Rede. Ein weiteres Merkmal von Domänen ist, dass häufig eine oder wenige Gruppen von Domänenexperten primär damit interagieren.

Um Domänen zu erkennen, lohnt sich ein Blick auf die Abläufe im System. Ein E-Procurement-System, das sich um die Beschaffung von Büromaterial kümmert, könnte bspw. die beiden in **Abbildung 1** gezeigten Prozesse unterstützen.

Dabei fällt auf, dass sich die Prozessschritte *Approve Order*, *Request Budget* und *Approve Budget* vorrangig um Organisationshierarchien und das verfügbare Budget drehen. Außerdem sind hier primär Manager eingebunden. Beim Prozessschritt geht es hingegen vor allem um Mitarbeiter und Produkte.

Natürlich lässt sich argumentieren, dass Produkte bei einem E-Procurement-System omnipräsent sind. Bei einer genaueren Betrachtung stellt sich jedoch heraus, dass das Wort Produkt in einigen der hier gezeigten Prozessschritte ganz unterschiedliche Dinge bezeichnet. Während ein Produkt beim Auswählen im Katalog bspw. sehr detailreich ist, muss sich der Freigabeprozess lediglich ein paar wenige Eckdaten merken (**Abb. 2**).

Im Sinne der allgegenwärtigen Sprache, die innerhalb jeder Domäne vorherrscht, ist zwischen diesen beiden Ausprägungen eines Produkts zu unterscheiden. Das

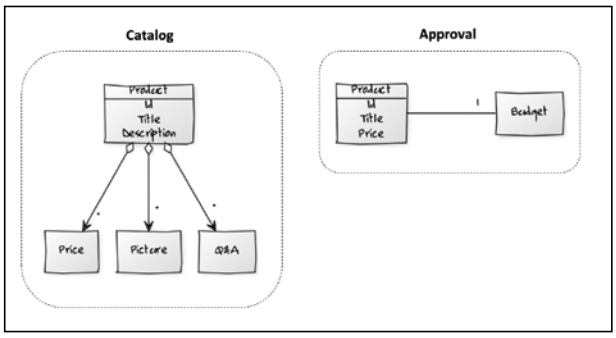


Abb. 2: Produkteigenschaften im Freigabeprozess

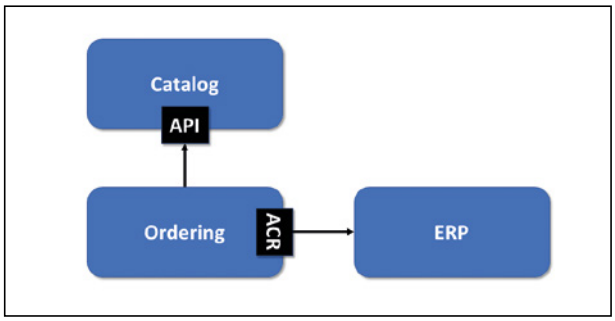


Abb. 3: Context Mapping

führt dazu, dass möglichst konkrete und somit auch aussagekräftige Modelle entstehen. Gleichzeitig unterbindet dieser Ansatz, dass ein einziges Modell entsteht, das versucht, die gesamte Welt zu beschreiben. Solche Modelle sind häufig unübersichtlich und mehrdeutig. Außerdem weisen sie zu viele wechselseitige Abhängigkeiten auf, die eine Entkopplung sowie Untergliederung unmöglich machen.

Auf logischer Ebene können die einzelnen Sichten auf das Produkt trotzdem miteinander in Verbindung stehen. Wird das durch dieselbe ID auf beiden Seiten ausgedrückt, kommt man hier ohne technische Abhängigkeiten aus.

Entsprechend ist jedes Modell nur innerhalb eines bestimmten Anwendungsbereichs gültig. Diesen Anwendungsbereich und den damit einhergehenden Rahmen bezeichnet DDD auch als den Bounded Context. Idealerweise hat jede Domäne ihren eigenen Bounded Context. Wie der nächste Abschnitt jedoch zeigt, lässt sich dieses Ziel gerade beim Einbinden von Drittsystemen nicht immer erreichen.

Context Mapping

Obwohl die einzelnen Domänen möglichst autark sind, müssen sie dennoch ab und an miteinander interagieren. Im hier betrachteten Beispiel könnte eine weitere Domäne *Ordering* zum Versenden von Bestellungen sowohl auf die Domäne *Catalog* als auch auf ein angebundenes ERP-System zugreifen (**Abb. 3**).

Die Art und Weise, wie diese Domänen miteinander interagieren, wird über eine Context Map festgelegt. Prinzipiell könnten sich *ordering* und *booking* die gemeinsam benötigten Modellelemente teilen. In diesem Fall ist jedoch darauf zu achten, dass eine Modifikation keine In-

Abb. 4: Ordner der Umsetzung

konsistenzen auf der anderen Seite mit sich bringt. Eine Domäne könnte auch die andere einfach verwenden. In diesem Fall stellt sich jedoch die Frage, wem in diesem Zusammenspiel wie viel Macht zukommt. Kann der Konsument dem Anbieter bestimmte Änderungen aufzwingen und auf Abwärtskompatibilität pochen? Oder muss sich der Konsument mit dem zufriedengeben, was er vom Anbieter bekommt?

Im betrachteten Fall bietet *catalog* ein API an, um zu verhindern, dass sich Änderungen in der Domäne zwangsweise auf die Konsumenten auswirken. Da *ordering* wenig Einfluss auf das ERP-System hat, nutzt es einen Anti-Corruption-Layer (ACR) zum Zugriff. Ändert sich etwas im ERP-System, muss es somit nur genau diesen aktualisieren. Daneben definiert Strategic Design noch weitere Strategien für das Verhältnis zwischen Konsumenten und Anbietern.

Umsetzung mit Nx

Für die Umsetzung der definierten Architektur kommt hier ein Workspace auf der Basis von Nx [1] zum Einsatz. Dabei handelt es sich um eine Erweiterung für das Angular CLI, das unter anderem dabei unterstützt, eine Lösung in verschiedene Anwendungen und Libraries zu untergliedern. Das ist natürlich nur einer von mehreren möglichen Ansätzen. Als Alternative könnte man bspw. jede Domäne als komplett eigene Lösung umsetzen. Dies entspräche dem Micro-App-Ansatz oder käme ihm zumindest sehr nahe.

Die Lösung untergliedert alle Anwendungen in einem Ordner *apps* und alle wiederverwendbaren Bibliotheken finden sich nach Domänen geordnet im Ordner *libs* (**Abb. 4**).

Da solch ein Workspace aus mehreren Anwendungen und Bibliotheken besteht, die in einem gemeinsamen Quellcode-Repository verwaltet werden, ist auch von einem Monorepo die Rede. Dieses Muster kommt unter anderem bei Google und Facebook sehr exzessiv zum Einsatz und ist auch seit rund zwanzig Jahren der Standardfall bei der Entwicklung von .NET-Lösungen in der Microsoft-Welt.

Es erlaubt auf besonders einfache Weise das Teilen von Quellcode zwischen den Projektbeteiligten und beugt in diesem Fall auch Versionskonflikten vor, indem es lediglich einen zentralen *node_modules*-Ordner mit abhängigen Bibliotheken gibt. Somit ist sichergestellt, dass z. B. jede Bibliothek dieselbe Angular-Version verwendet.

Die nötigen Befehle zum Erzeugen solch eines Monorepos und zum Hinzufügen von Anwendungen und Bibliotheken finden sich in der Dokumentation von Nx [1].

Listing 1

```
"nx-enforce-module-boundaries": [
  true,
  {
    "allow": [],
    "depConstraints": [
      { "sourceTag": "scope:app", "onlyDependOnLibsWithTags":
                                                    ["type:shell"] },
      { "sourceTag": "scope:catalog", "onlyDependOnLibsWithTags":
                                    ["scope:catalog", "scope:shared"] },
      { "sourceTag": "scope:shared", "onlyDependOnLibsWithTags":
                                                    ["scope:shared"] },
      { "sourceTag": "scope:booking", "onlyDependOnLibsWithTags":
                ["scope:booking", "scope:shared", "name:catalog-api"] },

      { "sourceTag": "type:shell", "onlyDependOnLibsWithTags":
                                            ["type:feature", "type:util"] },
      { "sourceTag": "type:feature", "onlyDependOnLibsWithTags":
                                          ["type:data-access", "type:util"] },
      { "sourceTag": "type:api", "onlyDependOnLibsWithTags":
                                          ["type:data-access", "type:util"] },
      { "sourceTag": "type:util", "onlyDependOnLibsWithTags":
                                                    ["type:util"] }
    ]
  }
]
```

Kategorisierung von Bibliotheken

Den Vorschlägen der hinter diesem quelloffenen Projekt stehenden Firma Nrwl folgend [3], unterteilt die betrachtete Lösung die einzelnen Bibliotheken in die folgenden Kategorien:

- *feature*: implementiert einen Use Case
- *data-access*: implementiert Datenzugriffe, z. B. via

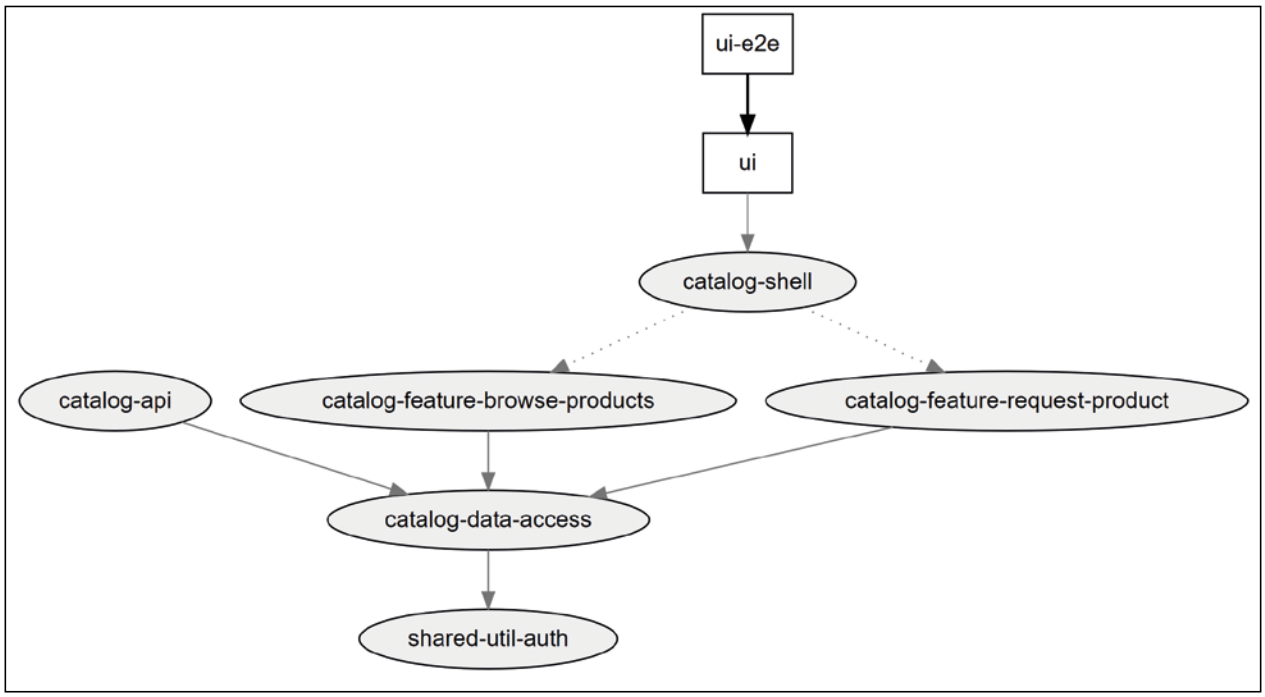


Abb. 5: Abhängigkeiten zwischen Bibliotheken

HTTP oder WebSockets
- *ui*: bietet Use-Case-agnostische und somit wiederverwendbare Komponenten
- *util*: bietet Hilfsfunktionen

Der Autor erweitert diese Kategorien um die folgenden:

- *shell*: bietet bei einer Anwendung, die mehrere Domänen umsetzt, den Einsprungpunkt für eine Domäne
- *api*: bietet Funktionalitäten für andere Domänen
- *domain*: Domänenlogik (hier nicht verwendet)

Falls die Geschäftslogik auch teilweise am Client stattfindet, würden Domainbibliotheken z. B. im Sinne von Functional Domain Modelling [2] neben Entitäten und Repositories zum Datenzugriff auch Services für Berechnungen sowie weitere Building-Blocks aus der Welt von DDD beinhalten.

Zur Wahrung der Übersicht werden die Kategorien als Präfix für die einzelnen Bibliotheksordner verwendet. Somit werden in einer sortierten Übersicht auch Bibliotheken derselben Kategorie nebeneinander präsentiert.

Jede Bibliothek hat außerdem ein öffentliches API (Listing 1), über das es einzelne Bestandteile veröffentlicht:

```
export * from './lib/catalog-data-access.module';
export * from './lib/catalog-repository.service';
```

Alle anderen Bestandteile verbergen sie hingegen; diese lassen sich somit beliebig verändern.

Zugriffe auf Bibliotheken kontrollieren

Zur Verbesserung der Wartbarkeit gilt es, die Abhän-

Listing 2

```
"projects": {
  "ui": {
    "tags": ["scope:app"]
  },
  "ui-e2e": {
    "tags": ["scope:e2e"]
  },
  "catalog-shell": {
    "tags": ["scope:catalog", "type:shell"]
  },
  "catalog-feature-request-product": {
    "tags": ["scope:catalog", "type:feature"]
  },
  "catalog-feature-browse-products": {
    "tags": ["scope:catalog", "type:feature"]
  },
  "catalog-api": {
    "tags": ["scope:catalog", "type:api", "name:catalog-api"]
  },
  "catalog-data-access": {
    "tags": ["scope:catalog", "type:data-access"]
  },
  "shared-util-auth": {
    "tags": ["scope:shared", "type:util"]
  }
}
```

gigkeiten zwischen den einzelnen Bibliotheken zu minimieren. Das Erreichen dieses Ziels lässt sich mit Nx zunächst mal auf grafische Weise prüfen (**Abb. 5**).

Im hier betrachteten Fall wurden ein paar Regeln bei der Kommunikation zwischen Bibliotheken eingehalten und diese führen zu einer konsequenten Schichtentrennung. Beispielweise darf jede Bibliothek nur auf Bibliotheken derselben Domäne oder auf Bibliotheken aus dem Bereich *shared* zugreifen. Der Zugriff auf APIs wie das *catalog-api* muss einzelnen Domänen explizit gewährt werden.

Auch aus der Kategorisierung der Bibliotheken ergeben sich Einschränkungen: Eine Shell greift nur auf Features zu und ein Feature wiederum nur auf Data-Access-Bibliotheken. Daneben darf aber jeder auf Utils zugreifen. Um solche Einschränkungen zu erzwingen, kommt Nx mit eigenen Linting Rules (Listing 1).

Entsprechend einem Vorschlag aus der Veröffentlichung „Enterprise Angular Monorepo Patterns“ von Nrwl [3] werden die Domänen hier mit dem Präfix *scope* und die Bibliotheksarten mit dem Präfix *kind* versehen. Präfixe dieser Art sollen lediglich die Lesbarkeit erhöhen und lassen sich frei vergeben. Außerdem zeigt dieses Beispiel auch die Domäne *Booking*, die entsprechend des Context Mappings Zugriff auf das *CatalogApi* erhält.

Die Zuordnung zwischen den Projekten und den hier gezeigten Bibliotheksarten sowie Domänen findet in der Datei *nx.json* statt (Listing 2).

Alternativ dazu lassen sich diese Tags auch beim Einrichten der Applikationen und Bibliotheken angeben.

Um gegen diese Regeln zu prüfen, reicht ein Aufruf von *ng lint* auf der Kommandozeile. Entwicklungsumgebungen wie WebStorm/IntelliJ oder Visual Studio Code zeigen solche Regelverletzungen auch schon während des Tippens an. In letzterem Fall ist ein entsprechendes Plug-in zu installieren.

Fazit

Strategic Design bietet eine bewährte Vorgehensweise, um eine Anwendung in möglichst autarke Domänen zu unterteilen. Diese Domänen zeichnen sich durch ein eigenes Fachvokabular aus, das rigoros von allen Beteiligten zu nutzen ist.

Die CLI-Erweiterung Nx bietet eine sehr charmante Möglichkeit zur Umsetzung dieser Domänen mit unterschiedlichen, nach Domänen gruppierten Bibliotheken. Um den Zugriff durch andere Domänen zu beschränken und zur Verminderung der Abhängigkeiten im Sinne einer Schichtentrennung, erlaubt es das Festlegen von Einschränkungen für den Zugriff auf einzelne Bibliotheken.

Es lässt sich natürlich argumentieren, dass ein Angular-Client nicht zwangsweise Domänenlogik beinhaltet. Fakt ist aber, dass gerade bei Single-Page-Applikationen immer mehr Logik auch am Client zu finden ist. Unabhängig davon haben sich gerade die Ideen von Strategic Design in diesem Umfeld als äußerst nützlich erwiesen, um einen guten Schnitt zu erreichen.

Manfred Steyer unterstützt als Trainer und Berater Softwareteams im gesamten deutschen Sprachraum bei der Entwicklung mit Angular. Er wurde als Google Developer Expert ausgezeichnet und schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. In seinem aktuellen Buch zu Angular behandelt er die vielen Seiten des populären JavaScript Frameworks aus der Feder von Google.

www.softwarearchitekt.at

Links & Literatur

[1] https://nx.dev

[2] https://pragprog.com/book/swdddf/domain-modeling-made-functional

[3] https://go.nrwl.io/angular-enterprise-monorepo-patterns-new-book

API Authorization in ASP.NET Core 3.0 mit IdentityServer

Du kommst hier nicht rein

Manche Dinge sind, zumindest gefühlt, unnötig aufwändig, kompliziert oder beides. Aktuell ist die Absicherung eines ASP.NET Core Web API leider auch noch eins dieser Themen. Noch – denn es ist Besserung in Sicht.

von Sebastian Gingter

Als Softwareentwickler, die die Plattform „Web" bedienen, haben wir es in aller Regel mit einem von zwei Fällen zu tun: Wir bauen entweder eine Webanwendung, die für die gesamte Öffentlichkeit uneingeschränkt zur Verfügung steht, oder wir arbeiten an einer Anwendung, die irgendeine Art von Zugangskontrolle in Form einer Anmeldung eines Benutzers benötigt. Sei es ein einfacher Blog mit Kommentarfunktion, ein Webshop, bei dem wir den Kunden kennen wollen, oder auch eine interne Businessanwendung, bei der wir den Benutzer zur Berechtigungssteuerung identifizieren müssen.

Von der Anmeldung an Webanwendungen …

Für serverseitig gerenderte Webapplikationen mit ASP.NET (Core) MVC bringt uns das ASP.NET Framework alles dafür Notwendige schon von Haus aus mit, und zwar schon seit der allerersten Version: Bereits mit der einfachen Forms Authentication in ASP.NET 1.x und etwas später dann mit dem Membership Provider in ASP.NET 2.x waren die Mittel, um Benutzer zu verwalten, sie zu authentifizieren und dann zu autorisieren, direkt an Bord. Mit der Einführung von OWIN und damit Middlewares mit .NET 4.5.1 im Jahr 2013 wurde auch das Benutzersystem revolutioniert und heißt seitdem ASP.NET Identity. Seit damals hat sich im .NET-Umfeld viel getan: Wir bewegen uns auf .NET 4.8 zu und haben parallel das neue ASP.NET Core an der Hand, das uns ein viel moderneres Entwickeln für alle erdenklichen Plattformen ermöglicht.

Umso erstaunlicher erscheint es, dass wir zwar mit klassischen WebForms, ASP.NET MVC mit Razor Templates und auch den neuen Razor Pages das Identity-System verwenden können, wir aber bis heute keine eingebaute Möglichkeit haben, es auch aus ASP.NET MVC Web API oder den API-Controllern in ASP.NET MVC Core zu verwenden. Doch warum ist das so?

Serverseitig gerenderte Webanwendungen basieren darauf, dass ein Webbrowser eine Webseite anfragt und nach einer Benutzeraktion entweder eine weitere, andere Seite angefragt oder für die aktuelle Seite Formulardaten an den Server per POST gesendet werden. All das passiert immer aus dem Browser heraus. Wir können hierfür also problemlos Cookies für die Anmeldung und die Sessionverwaltung verwenden. ASP.NET Identity setzt daher bei MVC (Core) und Razor Pages genau hierauf auf und realisiert die Authentication mit Cookies, die ASP.NET Identity für uns ausstellt, und die der Browser für uns handhabt.

… zur Authentifizierung bei API-Endpunkten

Bei API-Endpunkten sieht das anders aus: Diese müssen nicht unbedingt von einem Browser angesprochen werden, der sich vorher einen solchen Cookie geholt hat. Es können natürlich moderne, rein browserseitige Webanwendungen wie Single Page Applications (SPAs) sein, aber es können genauso gut mobile iOS- oder Android-Anwendungen, Windows-Dienste bzw. Linux Daemons oder gar klassische Desktopanwendungen sein, die mit unserem API arbeiten wollen und dazu einen authentifizierten Zugriff benötigen. Alle diese unterschiedlichen Szenarien haben dabei andere Anforderungen. Cookies würden hier nur bedingt helfen. Und da wir Authentifizierung ja auch nicht einfach nur machen, damit wir so in etwa wissen, wer da gerade mit unserem API spricht, sondern hier auch, je nach Anwendungsfall, Berechtigungen daran hängen und wir uns auf diese Information verlassen können müssen, bewegen wir uns damit automatisch auch im Bereich Security, und damit haben diese unterschiedlichen Szenarien leider auch unterschiedliche und vielfältige Angriffsmöglichkeiten.

Dankenswerterweise haben sich deswegen schon vor Jahren einige Leute bei der IETF (Internet Engineering Task Force) intensive Gedanken darüber gemacht und uns OAuth als Standard vorgeschlagen, der 2006 veröffentlicht wurde. Dieser Standard hat sich dann auch durchgesetzt und wurde 2012 mit der neuen Version OAuth 2.0 überarbeitet. Zwei Jahre später hat die OpenID Foundation dann auf OAuth 2.0 aufsetzend noch OpenID Connect als Protokoll bereitgestellt, das es erlaubt, mehr Informationen rund um die Identität des Benutzers zu erhalten. Also genau das, was wir als Entwickler für unser API benötigen.

OAuth 2.0 und OpenID Connect

Zwischen Cookie- und OAuth-basierter Authentifizierung gibt es einige Unterschiede. Uns interessieren hierbei aber erst einmal nur die beiden auffälligsten: Erstens geschieht die Anmeldung bei OAuth meist nicht auf dem Webserver, auf dem auch das API liegt. Stattdessen meldet sich der User bei einem sogenannten Authentication Server an, der der Clientanwendung dann einen Token als Anmeldeinformation ausstellt. Diese Tokens können einfache, zufällige Strings sein, oder auch sogenannte JSON Web Tokens (JWT), die Informationen in einer definierten JSON-Datenstruktur beinhalten. Unser API vertraut dem Authentication-Server und akzeptiert die von ihm ausgestellten Tokens. Der zweite auffällige Unterschied besteht darin, dass diese Tokens über einen HTTP-Header von der Clientanwendung an unser API übertragen werden, und eben nicht mehr in einem Cookie.

Die eigenen Endpunkte mit OAuth 2.0 bzw. OpenID Connect abzusichern, ist heutzutage der bevorzugte Weg für API-Entwickler, und auch in ASP.NET gibt es schon seit gefühlten Ewigkeiten die Clientbibliotheken, um unsere APIs über einen externen Authorization-Server mittels Tokens abzusichern. Als externe Dienste dienen hier gerne die bekannteren sozialen Netzwerke, und so kann man die Benutzer seines API problemlos gegen die Authorization-Server von Facebook oder Twitter authentifizieren lassen. Auch Google und Microsoft bieten OAuth-2.0-Authorization-Server an, sodass man sich mit seiner Gmail-Adresse oder seinem Microsoft-Account anmelden kann. Daneben gibt es auch eher technische Anbieter wie GitHub, an die man sein API hängen kann.

Doch der Clientteil ist natürlich nur die halbe Miete: Bis heute muss man sich nach externen Lösungen umschauen, wenn man einen Authorization-Server mit eigener Benutzerhaltung bereitstellen will oder muss. Auch hier kann man sich natürlich einen Authorization-Server als Software as a Service anmieten, wie zum Beispiel von Auth0 (Auth Zero) [1], aber wenn man eine anpassbarere Lösung benötigt oder seine Benutzerdaten nicht einem fremden Unternehmen überlassen möchte, muss man aktuell leider noch selbst Hand anlegen. Im .NET-Umfeld hat sich hierfür das Open-Source-Projekt IdentityServer4 [2] von Dominick Baier und Brock Allen bewährt: Er wird vielfach für die Authentifizierung von eigenen Benutzern gegenüber einem API eingesetzt.

Eine unerfreulich große Menge an Arbeit – oder?

Doch auch wenn wir hier nun eine Open-Source-Lösung für einen Authorization-Server an der Hand haben, die das OpenID-Connect-Protokoll und damit auch OAuth 2.0 beherrscht, so ist der IdentityServer dennoch leider keine Komplettlösung, die man einfach installieren und sofort nutzen kann. Selbst wenn man „nur mal schnell“ einen Authorization-Server zu Testzwecken bereitstellen will, muss man erst einmal eine Konfiguration für seine Clientanwendung und das abzusichernde API bereitstellen. Dafür muss man sich dann allerdings schon intensiver mit der Funktionsweise des IdentityServers auseinandersetzen, damit man auch eine korrekte Konfiguration für seinen Anwendungszweck erstellen kann. Wenn man diese dann nicht hartcodieren möchte, schreibt man extra Code, um die Konfiguration entweder aus Dateien oder aus einer Datenbank zu laden. Für den zweiten Fall gibt es zwar schon eine Bibliothek auf NuGet mit dem passenden Entity-Framework-(EF-) Datenmodell (das Paket IdentityServer4.EntityFramework), aber selbst dann müssen noch die dazu passenden Migrationen erstellt und ausgeführt werden. Das Gleiche gilt auch für die Testbenutzer: Entweder man gibt sich mit den festen Testbenutzerkonten „Alice“ und „Bob“ aus den Beispielen zufrieden, oder man muss wieder Hand anlegen und das, was man schon für die Konfigurationsdaten gemacht hat, auch noch einmal für die Benutzerdaten machen. Aber auch hier muss man sich selbst um die notwendigen Migrationen und die Konfiguration kümmern, wenn man den existieren Adapter für ASP.NET Identity nutzt (das Paket IdentityServer4.AspNetIdentity). Damit die Benutzer dann auch irgendwo ihre Anmeldedaten eingeben können, benötigen wir noch die nötigen Eingabeformulare. Zwar bringt der IdentityServer in seinen Beispielprojekten auch ein Boot-

strap-basiertes UI mit, aber auch das muss erst einmal in das eigene Projekt installiert und eingerichtet werden.

Alles in allem ist das leider eine unerfreulich große Menge an Arbeit, die da vor uns liegt. Dazu kommt noch der Lernaufwand über die OAuth-Konzepte und das Konfigurations-API des IdentityServers. Das ist unangemessen viel, insbesondere, wenn man doch eigentlich „nur mal schnell“ einen Authentication-Server für Testzwecke benötigt. Wenn man dann bedenkt, wie einfach es im Gegenzug ist, eine Authentifizierung für eine MVC- oder Razor-Pages-Anwendung zu realisieren, erscheint das Thema API-Authentifizierung ungelenk, sehr kompliziert und über die Maßen aufwändig.

Das muss es aber nicht sein, und zum Glück ist das ASP.NET-Identity-Team bei Microsoft auch zu dieser Ansicht gekommen: Es hat bereits damit begonnen, die Situation zu verbessern. So hat Barry Dorrans, der bei Microsoft im (ASP).NET-Team im Securitybereich arbeitet, angekündigt, dass es mit ASP.NET Core 3.0 ein eingebautes API-Authorization-Feature geben wird [3]. Microsoft baut dafür keinen eigenen Authorization-Server, sondern setzt auch auf den IdentityServer4 auf. Wie bereits erwähnt, muss dieser jedoch konfiguriert werden, und wenn man diese Konfiguration selbst erstellt, muss man die Konzepte von OAuth bzw. OpenID Connect kennen und dazu wissen, wie sie auf die Einstellungen des IdentityServers zu übertragen sind. Damit wir uns nicht damit aufhalten müssen, versucht das ASP.NET-Identity-Team, uns möglichst viel Arbeit und (Lern-)Aufwand abzunehmen.

API Authorization kommt in ASP.NET Core 3.0

Da ASP.NET Core 3.0 erst für die zweite Jahreshälfte 2019 angekündigt ist [4], müssen wir uns noch etwas gedulden, bis das Feature wirklich bei uns ankommt. Jedoch ist eine erste, beispielhafte Implementierung schon heute im ASP.NET Core Repository auf GitHub verfügbar [5]. An dieser Stelle ist anzumerken, dass es sich derzeit um einen Stand handelt, an dem noch entwickelt wird und der aktuell nur als Preview zur Verfügung steht. Wir dürfen bis zur Veröffentlichung also noch mit Änderungen rechnen. Dennoch lohnt es sich, schon jetzt einen genaueren Blick auf das zu werfen, was uns Microsoft hier an Automatismen liefern möchte.

Die Integration verbindet ASP.NET Identity mit dem IdentityServer ausschließlich mit .NET-(Core-)Hausmitteln, daher setzt das Identity-Team hier auf das Entity Framework, um notwendige Daten in der Datenbank abzulegen. Im Code müssen wir daher zuerst in der Startup.cs unseres Projekts in der Methode *ConfigureServices()* den entsprechenden Datenbankkontext konfigurieren und in die verfügbaren Services unseres Projekts eintragen:

```
services.AddDbContext<ApplicationDbContext>(options =>
  options.UseSqlite(Configuration.GetConnectionString("DefaultConnecti
on")));
```

Der *ApplicationDbContext*, den uns Microsoft hier mitliefert, leitet von dem neuen *ApiAuthorizationDbContext* ab, der selbst wiederum vom *IdentityDbContext* ableitet, und uns damit auch sofort Zugriff auf die bereits bekannte ASP.NET-Identity-Datenstruktur gibt. Zusätzlich implementiert der *ApiAuthorizationDbContext* auch ein Interface des IdentityServers, über das dieser seine Bewegungsdaten persistiert. Wir kommen also mit einem einzigen Datenbankkontext aus und müssen auch nur für diesen einen Kontext Migrationen verwalten. Im zweiten Schritt müssen wir das normale ASP.NET-Identity-System konfigurieren:

```
services.AddDefaultIdentity<ApplicationUser>()
  .AddEntityFrameworkStores<ApplicationDbContext>();
```

Hiermit tragen wir die Identity Services in die Dependency Injection (DI) unseres Projekts ein und sagen ihnen, dass sie den oben bereitgestellten *ApplicationDbContext* für die Persistenz unserer Benutzer nutzen sollen. Bis auf die Verwendung des erweiterten Identity-Kontexts sind dies nur die Schritte, die man auch für die normale Konfiguration von ASP.NET Identity vornehmen müsste. Im dritten Schritt fügen wir unserem Projekt den eigentlichen IdentityServer hinzu:

```
services.AddIdentityServer()
  .AddApiAuthorization<ApplicationUser, ApplicationDbContext>();
```

Dies führt zuerst die normale Eintragung aller Typen des IdentityServers in die DI durch. In der Methode

AddApiAuthorization() geschieht nun die eigentliche Integrationsmagie des ASP.NET-Identity-Teams. Anstatt hier, wie es bei der normalen Nutzung des IdentityServers der Fall wäre, die Konfiguration zu übergeben, wird sie im Hintergrund für uns und unser API erstellt und vollautomatisch bereitgestellt. Wir gehen hier später noch auf die Details ein, aber für jetzt soll es genügen, dass durch die Angabe der Typen für das Nutzerobjekt und den Datenbankkontext auch der IdentityServer so konfiguriert wird, dass er unseren Kontext sowohl für die Authentifizierung der Nutzer als auch für die Ablage seiner eigenen Daten verwenden soll. Hiermit wurde der IdentityServer konfiguriert, und nun muss er noch als Middleware registriert werden. Dazu müssen wir noch vor dem Aufruf von *UseMvc()* in der *Configure()*-Methode unserer *Startup*-Klasse eintragen:

```
app.UseIdentityServer();
```

Bis hierhin haben wir nun den IdentityServer konfiguriert, und unser API-Projekt ist nun gleichzeitig auch ein Authorization-Server im Sinne von OAuth 2.0. Unsere API-Endpunkte, also unsere Action-Methoden auf unseren Controller-Klassen, können mit den Tokens, die unser Server ausstellt, aber noch nichts anfangen. Dafür fehlt noch der letzte Schritt, in dem wir unserem API sagen, dass es überhaupt Authentication beherrschen soll und unsere Tokens akzeptiert. Hierfür fügen wir noch folgende Zeilen in unsere *ConfigureServices()*-Methode ein:

```
services.AddAuthentication()
  .AddIdentityServerJwt();
```

Danach kann ASP.NET MVC die Tokens, die unser frisch konfigurierter IdentityServer, der nun auch in unserem Projekt verfügbar ist, akzeptieren. Normalerweise müssten wir die Authentication explizit auf die passende Ressource eines spezifischen Authorization-Servers konfigurieren, aber auch das wird uns netterweise vom Identity-Team in der *AddIdentityServerJwt()*-Methode abgenommen.

Hinter den Kulissen

Das ASP.NET-Identity-Team hat sich wirklich viel Mühe gegeben, die Komplexität der IdentityServer-Konfiguration hinter diesen zwei einfachen Extension-Methoden zu verstecken. Die erste für den IdentityServer relevante, also *AddApiAuthorization()*, nutzt zuerst die schon vom IdentityServer bereitgestellten Funktionalitäten, die Benutzerhaltung von ASP.NET Identity zu verwenden und die eigenen Daten, zum Beispiel für ausgestellte Tokens, aus dem gleichen Datenbankkontext zu verwenden. Danach werden ein paar der vom IdentityServer in der DI registrierten Klassen durch eigene Varianten ersetzt. Der Hintergrund ist, dass der IdentityServer normalerweise auf seinen eigenen URL konfiguriert wird, dieser hier aber nicht angegeben wird. Daher werden hier Klassen, die für das URL-Handling im IdentityServer verwendet werden, ausgetauscht. Und zwar gegen Implementierungen, die die absoluten und relativen URLs zu dem API und damit auch zum IdentityServer im gleichen Projekt, selbst automatisch ermitteln können.

Als Nächstes werden einige Standard-Identity-Ressourcen registriert und danach das eigene API als sogenannte API-Ressource. Hier wird für das lokale API in diesem Projekt eine Ressource angelegt, die einen Zugriff von allen Clients erlaubt, die später noch registriert werden. Diese werden dann im nächsten Schritt generiert, wobei aktuell in der *appsettings.json* der Name der Clients und ein sogenanntes Profil konfiguriert werden müssen. In dem offiziellen Beispielprojekt [6] wird hier folgende Konfiguration verwendet:

```
"IdentityServer": {
  "Clients": {
    "ApiAuthSampleSPA": {
      "Profile": "IdentityServerSPA"
    }
  }
}
```

Hiermit wird lediglich gesagt, dass es einen Client namens *ApiAuthSampleSPA* gibt, und dieser mit dem Profil *IdentityServerSPA* generiert werden soll. Derzeit existiert noch keine Dokumentation, welche Profile es final geben wird und wie diese sich im Detail unterscheiden. Für das *IdentityServerSPA*-Profil wird derzeit ein Client generiert, der auf dem URL des aktuellen Servers ansprechbar ist. Der Name des Clients in der Konfiguration ist wichtig, denn die Anwendung, die das API verwenden will, muss ihre Autorisierungsanfragen an den IdentityServer mit diesem Namen als ClientId stellen. Damit weiß der IdentityServer, welche Clientkonfiguration er zu verwenden hat. Aktuell sieht das Feature Profile für eine externe SPA vor, die nicht vom URL des API-Servers kommt, eben die verwendete *Identity-*

Der Name des Clients in der Konfiguration ist wichtig, denn die Anwendung, die das API verwenden will, muss ihre Autorisierungsanfragen an den IdentityServer mit diesem Namen als ClientId stellen.

ServerSPA, die vom gleichen URL kommt, und *NativeApp* für eine Mobile- oder eine Desktopanwendung. Offenbar auch vorgesehen, aber aktuell (noch?) auskommentiert, ist ein Profil für eine Server-rendered MVC-Webanwendung mit dem Namen WebApplication.

Zuletzt werden noch die sogenannten Signing Credentials konfiguriert, also das kryptografische öffentliche und private Schlüsselpaar, mit dem der IdentityServer die Tokens, die er ausstellt, auch signiert. Zur Entwicklungszeit wird beim ersten Start ein neues Paar automatisch generiert und dann verwendet, zur Laufzeit in der Releasekonfiguration muss hier in der *appsettings.json* entweder ein x509-Zertifikat aus einer Datei oder aus dem betriebssystemeigenen Zertifikatsspeicher angegeben werden.

Dort, wo die IdentityServer-JWT-Authentifizierung hinzugefügt wird, wird auch wieder auf die automatisch generierten URL-Handler zurückgegriffen und der soeben konfigurierte lokale IdentityServer als sogenannte Authority, also zulässige Quelle für Tokens, eingetragen. Ab diesem Zeitpunkt kann der Standard-Authentication-Handler für JW-Tokens, den es auch schon seit der ersten ASP.NET-Core-Version gibt, seine Aufgabe ganz regulär erledigen. Damit ist die Konfiguration des IdentityServers und unserer API-Authentifizierung abgeschlossen. Was noch fehlen würde, wäre die Clientseite, also die Single Page Application, die nun gegen den IdentityServer authentifiziert und dann mit dem für sie ausgestellten Token unser API aufruft. In dem Beispielprojekt des ASP.NET-Teams ist dieser allerdings auch enthalten.

Fazit

Wer schon einmal intensiver mit dem IdentityServer gearbeitet hat, wird zu schätzen wissen, welchen initialen Konfigurationsaufwand man sich mit diesen wenigen Zeilen sparen kann, wenn diese ApiAuthorization-Integration mit ASP.NET Core 3.0 veröffentlich wird. Dieses neue Feature macht es erstaunlich einfach, einen IdentityServer für Entwicklungs- und Testzwecke aufzusetzen, und zu einem wirklich schnellen Ergebnis zu kommen. Und genau hier hört die Genialität der Einfachheit auch leider schon auf, denn diese Integration ist konzeptionell auf genau dieses Szenario beschränkt: Das API hostet und akzeptiert seinen eigenen IdentityServer, die Clients entsprechen den vorgegebenen Profilen, und als Datenablage wird Entity Framework mit ASP.NET Identity für die Nutzerhaltung verwendet. Weichen die eigenen Anforderungen von diesem Szenario ab, wenn auch nur leicht, hilft uns diese neue, direkt in ASP.NET Identity eingebaute Integration schon nicht mehr weiter. In diesem Fall ist es leider notwendig, den IdentityServer in seiner normalen Form zu verwenden und all die genannten Schritte doch selbst zu machen. Trotz allem: Es ist wunderbar, dass man bald in der Lage sein wird, so einfach zu starten, und die Art, wie das Microsoft-Team die Konfiguration gelöst hat, kann einem dank ihrer Open-Source-Natur als Inspiration dienen, wie man es dann später selbst elegant lösen könnte.

Moderne Softwarearchitektur sowie leichtgewichtige und performante Anwendungen, primär mit .NET Core auf dem Server sind die Themen, die **Sebastian Gingter** als Software Architect und Consultant bei der Thinktecture AG vorantreibt. Sein Wissen gibt er auch gerne in Artikeln und auf Konferenzen weiter.

https://gingter.org @PhoenixHawk

Links & Literatur

[1] https://auth0.com

[2] http://bit.ly/IdentitySrv

[3] http://bit.ly/ApiAuthAnnouncement

[4] http://bit.ly/NetCore3Announcement

[5] http://bit.ly/ApiAuthRepo

[6] http://bit.ly/ApiAuthSample

Neue Besen kehren gut, sagt man. Aber sind sie auch sicher?

Wasm – Ist das sicher oder kann das weg?

Schon wieder eine neue Technologie für aktive Inhalte – muss das denn wirklich sein? Als hätten wir mit Flash und Java nicht schon genug Ärger gehabt. Beides wurde wahrscheinlich öfter für Angriffe als für wirklich nützliche Anwendungen verwendet.

von Carsten Eilers

Jedenfalls wenn man mal von Spielen und in der Websteinzeit dem Wiedergeben von Videos durch Flash absieht. HTML5 und die dazugehörenden JavaScript APIs decken doch schon alle möglichen und unmöglichen Anwendungsfälle ab. Wieso also noch was Neues erfinden? Vor allem, wo doch jede neue Funktionalität immer auch die Angriffsfläche erhöht, die man doch eigentlich möglichst klein halten möchte. Die Antwort ist ganz einfach: Wieso nicht? Vielleicht ist das Neue ja besser als alles Alte.

Wichtig ist natürlich vor allem, dass es sicher ist. Nicht nur sicherer als Flash oder Java – das ist keine Kunst, so chronisch unsicher wie die sind. Sondern wirklich sicher, von Anfang an und ohne Kompromisse. Wobei Java in der Theorie auch sicher sein sollte, nur bei der Umsetzung, vor allem im Browser, hat es dann gewaltig gehakt. So sehr, dass Java im Browser inzwischen im Grunde tot ist.

Wie sieht es also mit der Sicherheit von WebAssembly (auch kurz „Wasm“ genannt) aus? Dazu schauen wir uns erst einmal an, was das überhaupt ist. Und zwar nur im Kontext von Webanwendungen und der Ausführung im Webbrowser, denn das ist der primäre Anwendungszweck. Dass WebAssembly auch in Nicht-Webumgebungen ausgeführt werden kann, ist laut den WebAssembly-Entwicklern zwar wünschenswert, aber kein zentrales Ziel der Entwicklung.

WebAssembly ganz einfach ...

Sehr vereinfacht gesagt ist WebAssembly ein Bytecode, der in einer Sandbox ausgeführt wird. Im Browser werden WebAssembly-Module in der JavaScript VM ausgeführt. Für den WebAssembly-Code gelten daher die gleichen Sicherheitsbeschränkungen wie für JavaScript-Code, insbesondere also die Same-Origin Policy. Hinzu kommt die zusätzliche Einschränkung, dass WebAssembly-Code nicht direkt auf das DOM der Seite zugreifen kann. Für Änderungen an der dargestellten Seite oder das Abfragen von Informationen daraus müssen JavaScript-Funktionen verwendet werden. Auch können die WebAssembly-Module nur dann auf die Hardware oder das Dateisystem zugreifen, wenn der Benutzer das in der Browserkonfiguration explizit erlaubt.

... und etwas formaler

Formal betrachtet ist WebAssembly die Spezifikation einer Instruction Set Architecture und eines Dateiformats. Die Instruction Set Architecture beschreibt einen (virtuellen) Rechner, und das Dateiformat legt den Aufbau von WebAssembly-Modulen fest. Der Browser lädt den WebAssembly-Binärcode in Form von Modulen und führt ihn aus.

Das ähnelt den bekannten Java-Applets, die zu Java-Bytecode kompiliert und in der Java Virtual Machine (VM) ausgeführt werden. Der Unterschied besteht darin, dass Java Applets eine Erweiterung des Browsers sind, die in einem Plug-in ausgeführt werden, während

WebAssembly-Bytecode, wie schon erwähnt, im JavaScript-Interpreter der Browser ausgeführt wird.

Entwickelt werden können die WebAssembly-Programme in etlichen Sprachen, angefangen bei C/C++ über Rust und Go bis hin zu C#. Und falls sich irgendwer hinsetzt und einen Compiler dafür schreibt, könnte man auch z. B. Basic C64 verwenden.

Aus Sicherheitssicht ist es aber egal, in welcher Sprache entwickelt wurde. Interessant sind dafür nur der Bytecode und dessen Ausführung. Und eigentlich sogar nur die Ausführung, denn der Bytecode stammt ja aus einer i. Allg. nicht vertrauenswürdigen Quelle, sodass man nicht weiß, ob er harmlos ist oder bösartig, oder vielleicht auch harmlos aber kompromittiert. Sämtliche Schutzmaßnahmen müssen daher im Browser ergriffen werden.

Die Schutzmaßnahmen

Einige der Sprachen, die nach WebAssembly kompiliert werden können, erlauben den Zugriff auf beliebige Speicheradressen. Ein Angreifer könnte das nutzen, um im Browser gespeicherte sicherheitsrelevante Informationen wie z. B. Passwörter oder Authentifizierungstokens auszuspähen, sofern er deren Speicherort kennt. WebAssembly-Modulen werden daher separate Speicherbereiche zugewiesen.

Schutz des Speichers

Speicherzugriffe sind nur auf eine dedizierte Untermenge des Heaps der JavaScript VM möglich. Dieser spezielle Heap wird durch einen JavaScript ArrayBuffer realisiert. Dadurch gelten für den gesamten Speicher eines WebAssembly-Moduls die gleichen Beschränkungen wie für normale JavaScript-Objekte. Außerdem überwacht dadurch die Garbage Collection die Speichernutzung des Moduls und gibt den Speicher ggf. frei. Speicherlecks sind dadurch nur in den gleichen Grenzen wie bei normalen JavaScript-Programmen möglich.

Da der komplette Heap des WebAssembly-Moduls im ArrayBuffer steckt, weiß die JavaScript-Umgebung, wie groß der Heap des Moduls ist und wo genau er im JavaScript Heap liegt. Dadurch kann sie bei jedem Speicherzugriff des WebAssembly-Codes zum einen exakt nachprüfen, ob sich der Zugriff innerhalb des ArrayBuffers bewegt, und zum anderen unerlaubte Zugriffe unterbinden.

Unerlaubte Speicherzugriffe in die JavaScript VM oder gar in den Adressraum des Browserprozesses werden dadurch effektiv verhindert. Zwar verlangsamen diese Prüfungen die Ausführung des Programms etwas, aber ohne sie wäre die Ausführung von WebAssembly-Code viel zu gefährlich. Denn darin kann potenziell beliebiger Schadcode stecken.

Schutz des Execution-Stacks

Genauso gefährlich wie beliebige Zugriffe auf den Speicher sind Zugriffe auf den Execution-Stack. Der enthält außer den lokalen Variablen auch die Rücksprungadresse an die aufrufende Funktion. Gelingt es bösartigem Code, die Rücksprungadresse zu manipulieren, kann der Angreifer sie nutzen, um beliebigen Code auszuführen. WebAssembly verhindert einen Schreibzugriff auf den Execution-Stack, indem er außerhalb des Speichers des WebAssembly-Moduls gespeichert wird.

Vorsicht bei Sprüngen

Auch Befehle, die an andere Stellen im Code springen, sind generell gefährlich. WebAssembly springt nur bei Funktionsaufrufen Speicheradressen an. Die werden teilweise erst dynamisch zur Laufzeit berechnet. Um Manipulationen zu verhindern, kommen dabei Tabellen zum Einsatz. Statt die Zieladresse direkt im *call*-Befehl zu speichern, verwenden *call*-Befehle zwei Parameter: einen Index in einer Tabelle und eine Funktionssignatur. Der Index zeigt in eine Tabelle mit Funktions-Pointern. Die wird genau wie der Execution-Stack außerhalb des WebAssembly-Moduls gespeichert, sodass ein Überschreiben durch das WebAssembly-Modul nicht möglich ist.

Bevor die WebAssembly-Laufzeitumgebung einen *call*-Befehl ausführt, prüft sie, ob die an den *call*-Befehl übergebene Funktionssignatur mit dem in der Tabelle am angegebenen Index gespeicherten Wert übereinstimmt. Nur wenn die beiden Signaturen übereinstimmen, wird die Funktion an der in der Tabelle gespeicherten Adresse aufgerufen. Gibt es unter dem Index keinen Eintrag in der Tabelle oder stimmen die beiden Signaturen nicht überein, wird die Ausführung des WebAssembly-Moduls sofort gestoppt.

Die beiden Parameter müssen nicht zwingend statisch gespeichert sein, sie können auch erst zur Laufzeit dynamisch generiert werden. Dadurch können z. B. Aufrufe an virtuelle Methoden in C++ oder Traits-Implementierungen in Rust realisiert werden.

WebAssembly ist fast immer deterministisch

Abgesehen von wenigen Ausnahmen [1] erfolgt die Ausführung der WebAssembly-Programme generell deterministisch. Eine nichtdeterministische Ausführung kann nur in wenigen, wohldefinierten Fällen auftreten, in denen die Implementierung eine aus einer limitierten Menge möglicher Verhaltensweisen wählt. Außerdem sind die Effekte einer nichtdeterministischen Ausführung immer nur lokal.

Ein nichtdeterministisches Verhalten tritt z. B. auf, wenn WebAssembly um neue Features erweitert wird und die verschiedenen Implementierungen die Features unterschiedlich unterstützen. Zwar kann über *has_feature* die Unterstützung abgefragt werden, trotzdem unterscheiden sich die Ausführungen auf den verschiedenen Implementierungen voneinander. Auch wenn die umgebungsabhängigen Ressourcenlimits erreicht werden, kann es zu einem nichtdeterministischen Verhalten kommen, z. B. wenn Speicherreservierungen fehlschlagen oder der Stack des Programms voll ist.

Sichere Entwicklung (fast) garantiert

Das Design von WebAssembly unterstützt die Entwicklung sicherer Programme [2], indem auf gefährliche Features verzichtet wurde, während gleichzeitig die Kompatibilität zu in C/C++ geschriebenen Programmen erhalten blieb.

Funktionen

Die WebAssembly-Module müssen alle zugänglichen Funktionen und die zugehörigen Datentypen beim Laden deklarieren. Und das auch dann, wenn dynamisch gelinkt wird. Dadurch ist es möglich, durch einen strukturierten Kontrollfluss die Integrität des Kontrollflusses (Control-Flow Integrity, CFI) zu erzwingen. Da kompilierter Code unveränderlich und während der Laufzeit nicht beobachtbar ist, kann der Kontrollfluss der WebAssembly-Programme nicht manipuliert werden, Hijacking-Angriffe sind also nicht möglich.

Wie zuvor bereits erwähnt, müssen Funktionsaufrufe den gültigen Index der gewünschten Funktion in der Funktionstabelle (Function Index Space) oder Tabellentabelle (Table Index Space) enthalten. Bei indirekten Funktionsaufrufen wird vor der Ausführung die Signatur der Funktion geprüft. Der außerhalb des Modulspeichers abgelegte Stack ist vor Pufferüberlaufangriffen sicher. Branches (*br*, *br_if*, *br_table*) müssen zu gültigen Zielen in der aufrufenden Funktion führen.

Variablen

C/C++-Variablen können in WebAssembly in Abhängigkeit von ihrem Gültigkeitsbereich (Scope) auf zwei unterschiedliche Primitive reduziert werden. Lokale Variablen mit festem Gültigkeitsbereich und globale Variablen werden als Werte mit festem Typ repräsentiert, auf die über einen Index zugegriffen wird. Die lokalen Variablen werden mit null initialisiert und im geschützten Call-Stack gespeichert, die globalen Variablen befinden sich im globalen Index Space und können von externen Modulen importiert werden.

Lokale Variablen mit unklarem statischem Gültigkeitsbereich werden beim Kompilieren in einem separaten Stack im dem Programm zugänglichen linearen Speicher gespeichert. Dazu gehören z. B. Variablen, die vom *address-of*-Operator verwendet werden oder die den Type *struct* haben und als „Returned by Value“ zurückgegeben werden. Dieser Stack ist ein isolierter Speicherbereich mit fester Maximalgröße, der per Default mit null initialisiert wird.

Traps

Traps werden verwendet, um ggf. die Ausführung sofort beenden zu können und unnormales Verhalten an die Laufzeitumgebung zu melden. Im Browser werden dazu JavaScript Exceptions verwendet.

Traps können z. B. ausgelöst werden, wenn in irgendeinem Index Space ein ungültiger Index spezifiziert, ein indirekter Funktionsaufruf mit falscher Signatur durchgeführt oder im linearen Speicher auf verbotene Adressen zugegriffen wird.

Speicherschutz

Im Vergleich zu normalen C/C++-Programmen verhindert WebAssembly verschiedene Klassen von Speicherfehlern. Z. B. sind die im Index-Space gespeicherten lokalen und globalen Variablen vor Pufferüberläufen sicher, da sie eine feste Größe haben und über einen Index adressiert werden.

Im linearen Speicher abgelegte Daten können benachbarte Objekte überschreiben, da die Prüfung des Gültigkeitsbereichs mit der Genauigkeit des linearen Speicherbereichs erfolgt und nicht kontextsensitiv ist. Das Vorhandensein von Kontrollflussintegrität und geschützten Aufrufstacks verhindert jedoch das direkte Einschleusen von Code. Dadurch sind die sonst üblichen Mitigations wie Data Execution Prevention (DEP) und Stack Smashing Protection (SSP) in WebAssembly-Programmen nicht nötig.

Eine weitere übliche Klasse von Speicherfehlern entsteht durch unsichere Pointer-Nutzung und undefiniertes Verhalten. Ein klassisches Beispiel dafür ist die Dereferenzierung von Pointern in nicht reserviertem Speicher oder die Freigabe noch genutzten Speichers. In WebAssembly enthalten Funktionsaufrufe und Variablen mit festem statischem Gültigkeitsbereich keine Pointer. Verweise auf ungültige Indexe in irgendeinem Index Space lösen i. Allg. schon beim Laden einen Fehler aus, im schlimmsten Fall eine Trap zur Laufzeit. Bei Zugriffen auf den linearen Speicher werden die Grenzen auf Regionsebene geprüft, ggf. wird zur Laufzeit eine Trap ausgelöst. Diese Speicherbereiche sind vom internen Speicher isoliert (s. o.) und werden per Default auf null gesetzt, sofern sie nicht anderweitig initialisiert werden.

Einige andere Klassen von Fehlern werden von WebAssembly allerdings nicht verhindert. So kann

zwar nicht direkt Code eingeschleust werden, aber das Hijacking des Kontrollflusses eines Moduls über Code Reuse Attacks gegen indirekte Funktionsaufrufe ist möglich. Konventionelle Angriffe über Return-oriented Programming (ROP) zum Missbrauch kurzer Codesequenzen (Gadgets) sind allerdings nicht möglich, da die Kontrollflussintegrität dafür sorgt, dass die Aufrufziele zur Ladezeit deklarierte, gültige Funktionen sind.

Auch Race Conditions wie z. B. „Time of Check to Time of Use"-(TOCTOU-)Schwachstellen und Seitenkanalangriffe sind möglich.

Kontrollflussintegrität

Die Wirksamkeit der Kontrollflussintegrität kann an ihrem Umfang gemessen werden. Generell muss der Kontrollfluss an drei Stellen geschützt werden:

1. bei direkten Funktionsaufrufen,
2. bei indirekten Funktionsaufrufen und
3. bei Rücksprüngen.

Bei direkten Funktionsaufrufen wird der Kontrollfluss in WebAssembly durch die Verwendung des Indexwerts geschützt, bei Rücksprüngen durch den geschützten Call-Stack.

Die Prüfung der Funktionssignaturen bei indirekten Funktionsaufrufen zur Laufzeit garantiert eine grobkörnige, typbasierte Integritätssicherung. Der einzige noch mögliche Angriff bei indirekten Aufrufen ist, wie schon erwähnt, die Wiederverwendung von Code auf Funktionsebene.

Clang/LLVM enthält eine integrierte Implementierung einer feineren Kontrollflussintegrität. Diese wurde in Version 3.9+ mit dem neuen WebAssembly Backend auf WebAssembly portiert. Damit ist u. a. ein besserer Schutz vor Code-Reuse-Angriffen auf indirekte Funktionsaufrufe möglich [3].

„Early Adopters" – die Cyberkriminellen

Das sieht doch alles ganz gut bzw. sicher aus. Aber Sie ahnen es sicher schon: Es gibt da doch noch ein paar Haken – irgendwas ist ja immer.

Im Fall von WebAssembly sind das z. B. Crypto Miner. Die Cyberkriminellen scheinen mit die ersten gewesen zu sein, die die Vorteile von WebAssembly erkannt haben: Damit lassen sich Kryptowährungen im Browser deutlich schneller minen als mit JavaScript-Code [4].

Dementsprechend gibt es schon etliche entsprechende Schädlinge, die über infizierte Websites oder bösartige Werbung verbreitet werden. Das kann man natürlich nicht WebAssembly vorwerfen, der Vollständigkeit halber soll es aber nicht unerwähnt bleiben. Crypto Miner im Browser gab es auch schon vor der Einführung von WebAssembly, nur waren die dann eben deutlich langsamer, da sie „nur" in JavaScript implementiert waren.

Generell kann fast jede JavaScript-basierte Schadsoftware nach WebAssembly portiert werden [5], aber das ist keine zusätzliche Gefahr. Ob der Schädling nun in JavaScript, WebAssembly oder „the next big Thing" implementiert ist, ist egal. Die einzigen, die darüber jammern, sind die Antivirushersteller, denn die müssen nun in einem weiteren Datenformat nach Schadsoftware suchen. Und, so eine Unverschämtheit: Das ist auch noch ein Binärformat, in dem sie nicht einfach nach Strings suchen können.

Ausbruch aus der Sandbox möglich

Alex Plaskett, Fabian Beterke und Georgi Geshev von den MWR Labs wollten mit einem Exploit für WebAssembly in Safari 11.0.3 unter macOS 10.13.3 am Pwn2Own-Wettbewerb 2018 teilnehmen [6]. Daraus wurde leider nichts, da die von ihnen entdeckte und für den Angriff ausgenutzte Schwachstelle (CVE-2018-4121) unabhängig von ihnen auch von Natalie Silvanovich von Googles Project Zero entdeckt [7] und an die Entwickler gemeldet wurde. Da Apple die Schwachstelle vor dem Wettbewerb kannte und dann auch geschlossen hatte, konnten Alex Plaskett, Fabian Beterke und Georgi Geshev ihren Exploit nicht mehr verwenden.

Er ist hier aber relevant, denn er beweist, dass trotz aller Schutzmaßnahmen ein Ausbruch aus der WebAssembly-Sandbox und die Kompromittierung des Systems möglich sind. Wie genau, haben die drei in einem Paper ausführlich beschrieben [6]. Und das ist nicht die einzige Schwachstelle in den WebAssembly-Implementierungen [8].

WebAssembly als Einfallstor für Spectre-Angriffe

Die Spectre-Varianten 1 und 2 lassen sich über den Browser ausnutzen [9]; Auch über WebAssembly. Dazu müssen die C/C++ Exploits nur nach WebAssembly kompiliert werden [10]. Das ist unschön, da gerade bekannt wurde, dass die ganzen softwarebasierten Schutzmaßnahmen gegen Spectre-Angriffe weitgehend nutzlos sind [11]. Von solchen Angriffen sind besonders Programme betroffen, die Daten aus nicht vertrauenswürdigen Quellen verarbeiten. Und bei mir und wahrscheinlich auch bei den meisten von Ihnen dürfte das am

stärksten gefährdete Programm der Webbrowser sein – keine gute Kombination.

Was sagen die Sicherheitsforscher?

WebAssembly ist noch ziemlich neu, aber die Sicherheitsforscher haben schon damit begonnen, es auf Schwachstellen abzuklopfen. Siehe die gefundenen Schwachstellen und den für den Pwn2Own-Wettbewerb entwickelten Exploit. Auch auf den Sicherheitskonferenzen gab es erste Vorträge, z. B. gleich zwei auf der Black Hat USA 2018.

Vorsicht Gefahrenstelle!

Natalie Silvanovich von Googles Project Zero (die ja auch schon Schwachstellen in WebAssembly gefunden hat) hat „The Problems and Promise of WebAssembly" vorgestellt [12]. Dabei gibt es einen sowohl negativen als auch positiven Aspekt: Für die meisten Problemfelder konnte sie bereits tatsächliche Schwachstellen präsentieren, sie sind also keine theoretische, sondern eine konkrete Gefahr. Das ist natürlich erst mal schlecht. Andererseits aber zumindest bisher auch gut: Die Schwachstellen betreffen „nur" Implementierungen von WebAssembly, nicht die Spezifikation. Die ist also (noch) sicher.

Konkret hat Natalie Silvanovich folgende Probleme angesprochen:

- Das Parsen des Binärformats sowie dessen Laden in das Modul (3 Schwachstellen)
- Die Speicherverwaltung (2 Schwachstellen)
- Die Tabellen mit den Funktionen (2 Schwachstellen)
- Die Initialisierung von Speicher und Elementen (1 Fehler, bisher keine Schwachstelle)
- Der Export der WebAssembly-Funktionen (bisher keine Schwachstellen)
- Probleme zur Laufzeit (bisher keine Schwachstellen)

Dass es Schwachstellen gibt, ist natürlich immer schlecht, aber ich denke, es hätte auch noch schlimmer kommen können. Die Anzahl der Schwachstellen ist doch sehr überschaubar. Außerdem sorgen die vorhandenen Sicherheitsfeatures dafür, dass die Wahrscheinlichkeit für Schwachstellen sinkt; jedenfalls in den WebAssembly-Programmen. Bei den Laufzeitumgebungen kommt es natürlich darauf an, wie sicher die Browserhersteller etc. sie implementieren.

Wie sich die Sicherheit von WebAssembly entwickelt, hängt nach Ansicht von Natalie Silvanovich von den zukünftigen Features ab. Und da sehe ich optimistisch in die Zukunft. Zum einen sind bereits weitere Sicherheitsfeatures angekündigt, zum anderen haben die WebAssembly-Entwickler schon bewiesen, dass sie Wert auf Sicherheit legen. Daher gehe ich davon aus, dass die nicht plötzlich anfangen, unsichere Features in die Spezifikation aufzunehmen. Vor allem, da die Entwicklung von WebAssembly von den Browserherstellern ausgeht, und die haben ein großes Interesse daran, dass die Browser sicher sind. Die werden kaum anfangen und die vorhandene Sicherheit durch unsichere WebAssembly-Features unterminieren.

WebAssembly = neue native Exploits im Browser

Kritischer sehen das Justin Engler und Tyler Lukasiewicz von der NCC Group WebAssembly, wie schon der Titel ihres Vortrags zeigt: „WebAssembly: A New World of Native Exploits on the Browser" [13]. Die neuen „Nativen Exploits" bestehen zum einen aus alten Exploits, die weiterhin funktionieren, und zum anderen aus neuen Exploits, die sich aus den Features von WebAssembly ergeben. Als Beispiele für alte Exploits nennen sie Angriffe auf Formatstringschwachstellen, Integer- und Pufferüberläufe. Einen Pufferüberlauf demonstrieren sie mit einem kleinen Beispiel. Weitere alte Exploits, die ihrer Ansicht nach wahrscheinlich funktionieren, sind Angriffe auf TOCTOU-Schwachstellen und Race Conditions sowie Timing- und Seitenkanalangriffe. Und auch Heap-basierte Schreibzugriffe können für Angriffe missbraucht werden.

Zum anderen gibt es neue Exploits. Da WebAssembly auf dem Umweg über JavaScript auf das DOM zugreifen kann, ist es möglich, über eine Pufferüberlaufschwachstelle einen DOM-basierten XSS-Angriff durchzuführen. Was auf den ersten Blick widersinnig erscheint, da sich über einen Pufferüberlauf sicher interessantere Sachen anstellen lassen, sieht auf den zweiten Blick doch schon etwas anders aus: Zum einen suchen die Funktionen zur Abwehr von XSS-Angriffen kaum im WebAssembly-Code danach, zum anderen kann über einen XSS-Angriff JavaScript und damit WebAssembly kontrolliert werden. Da lässt sich also durchaus Schaden mit anrichten, vor allem wenn man bedenkt, dass WebAssembly meist für aufwendige Webanwendungen verwendet wird.

Ein weiterer neuer Exploit sind Angriffe auf „Funktions-Pointer-Überläufe". Die sind möglich, da die „Funktions-Pointer" keine Pointer, sondern Indexe für die Funktionstabelle sind. Als Beispiele haben die beiden wieder einen XSS-Angriff durchgeführt und außerdem gezeigt, wie eine andere als die vorgesehene Funktion aufgerufen werden kann. Was je nach Funktion genauso gefährlich ist wie die Ausführung beliebigen eingeschleusten Codes.

Verschlimmert wird das Ganze dadurch, dass die Angriffe auch mit Node funktionieren – und das führt zu einer Remote Code Execution auf dem Server.

Da sind noch einige Anpassungen und Erweiterungen bei den Schutzfunktionen nötig, aber die wurden zum Teil auch schon angekündigt. Entwicklern empfehlen die beiden Forscher Folgendes:

- emscripten_run_script etc. zu vermeiden,
- den Optimizer zu verwenden, wodurch nicht genutzte, automatisch hinzugefügte Funktionen entfernt und die Angriffsfläche für Angriffe auf den Kontrollfluss reduziert werden,

- die Kontrollflussintegrität zu nutzen und vor allem
- ihre C-Fehler zu beheben. Wobei ich in so einem Fall ja vorschlagen würde, die gar nicht erst zu machen.

Sicherheitstools

Es gibt auch schon erste Sicherheitstools für WebAssembly: Wasabi [14] (kurz für „WebAssembly analysis using binary instrumentation“) ist ein am Software Lab der TU Darmstadt entwickeltes Framework zur dynamischen Analyse von WebAssembly-Programmen. Aron Szanto, Timothy Tamm und Artidoro Pagnoni von der Harvard University haben eine JavaScript-basierte VM für WebAssembly geschrieben, die Taint Tracking implementiert [15]. Damit ist es möglich, den Fluss sensitiver Informationen zwischen den lokalen Variablen zu verfolgen und problematische Parameter zu erkennen.

Fazit

Insgesamt sieht es mit der Sicherheit von WebAssembly gut aus. Darum kann das Fazit auch kurz ausfallen: Es spricht nichts dagegen, WebAssembly zu nutzen. Als Entwickler ebenso wie als Benutzer.

Natürlich gibt es da noch einiges, was sich verbessern lässt, aber das gilt wohl für jede Software. Zumindest bisher wurden keine Schwachstellen in der Spezifikation gefunden (evtl. wurde auch nur nicht danach gesucht?), und die Anzahl der in den Implementierungen gefundenen Schwachstellen ist überschaubar. Und was Schwachstellen in den WebAssembly-Programmen betrifft: Interessiert sich wirklich irgendwer für die? Ich könnte mir vorstellen, dass die Cyberkriminellen sich darauf stürzen werden. Aber erst, wenn sie keine anderen Schwachstellen mehr finden, die sie ausnutzen können. Und bis dahin wird noch einige Zeit vergehen.

Und selbst wenn es irgendwann Angriffe auf WebAssembly-Programme geben wird: Wie viel Schaden können die Angreifer dann schon anrichten? Ist es nicht eher wahrscheinlich, dass sie den Clientcode der einen betroffenen Webanwendung zwar kompromittieren können, dann aber in der Sandbox festsitzen? Und Angriffe auf den Clientcode der Webanwendung sind auch heute schon möglich, so extrem sicher sind die JavaScript-Clients nun auch nicht. Aber haben Sie schon von großflächigen Angriffen gehört? Oder auch nur einzelnen? „In the Wild“ ist es da doch ziemlich ruhig. Und ich glaube kaum, dass sich das mit WebAssembly ändert.

Außerdem ist das Zukunftsmusik und bis es soweit ist, werden die Sicherheitsfunktionen bestimmt sehr viel ausgereifter sein. Und die Entwickler der WebAssembly-Programme können ja ihren Teil zur Sicherheit beitragen, die Schwachstellen in ihren Programmen korrigieren und keine neuen mehr entstehen lassen.

Dipl.-Inform. **Carsten Eilers** ist freier Berater und Coach für IT-Sicherheit und technischen Datenschutz sowie Autor einer Vielzahl von Artikeln für verschiedene Magazine und Onlinemedien.

www.ceilers-it.de, www.ceilers-news.de

Links & Literatur

[1] WebAssembly Documentation: Nondeterminism in WebAssembly: https://webassembly.org/docs/nondeterminism/

[2] WebAssembly Documentation: Security: https://webassembly.org/docs/security/

[3] Foote, Jonathan; fastly: „Hijacking the control flow of a WebAssembly program“: https://www.fastly.com/blog/hijacking-control-flow-webassembly

[4] Neumann, Robert; Toro, Abel; Forcepoint: „In-browser mining: Coinhive and WebAssembly“: https://www.forcepoint.com/blog/security-labs/browser-mining-coinhive-and-webassembly

[5] Lonkar, Aishwarya; Chandrayan, Siddhesh; Virus Bulletin: „The dark side of WebAssembly“: https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly/

[6] Plaskett, Alex; Beterke, Fabian; Geshev, Georgi; MWR Labs: „Apple Safari - Wasm Section Exploit“: https://labs.mwrinfosecurity.com/publications/apple-safari-wasm-section-exploit/

[7] Silvanovich, Natalie; Google Project Zero: „Issue 1522: WebKit: WebAssembly parsing does not correctly check section order“: https://bugs.chromium.org/p/project-zero/issues/detail?id=1522&can=1&q=&start=1300

[8] Silvanovich, Natalie; Google Project Zero: „The Problems and Promise of WebAssembly“: https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html

[9] Eilers, Carsten; Windows Developer 3.19: „CPU-Schwachstellen im Überblick - Sind Spectre und Meltdown nur die Spitze des Eisbergs?“

[10] Jorgé; React, etc. Tech Stack: „Exploits in C/C++ to compiled JavaScript / WebAssembly“: https://react-etc.net/entry/web-security-exploits-c-to-javascript-webassembly

[11] Mcilroy, Ross; Sevcik, Jaroslav; Tebbi, Tobias; Titzer, Ben L.; Verwaest, Toon: „Spectre is here to stay: An analysis of side-channels and speculative execution“: https://arxiv.org/abs/1902.05178

[12] Silvanovich, Natalie; Black Hat USA 2018: „The Problems and Promise of WebAssembly“: https://www.blackhat.com/us-18/briefings/schedule/index.html#the-problems-and-promise-of-webassembly-11219 ; Video: https://www.youtube.com/watch?v=BHwqORo_83E

[13] Engler, Justin; Lukasiewicz, Tyler; Black Hat USA 2018: „WebAssembly: A New World of Native Exploits on the Browser“: https://www.blackhat.com/us-18/briefings/schedule/index.html#webassembly-a-new-world-of-native-exploits-on-the-browser-10043

Video: https://www.youtube.com/watch?v=DFPD9yl-C70

[14] Wasabi: http://wasabi.software-lab.org

[15] Szanto, Aron; Tamm, Timothy; Pagnoni, Artidoro: „Taint Tracking for WebAssembly“: https://arxiv.org/abs/1807.08349 ; GitHub: https://github.com/aronszanto/wasm-taint-tracking

Mit ASP.NET Core und Angular eine Webanwendung erstellen

Das Beste aus zwei Welten

Moderne verteilte Systeme im Web bestehen heutzutage aus vielen verschiedenen Teilen, Systemen und Technologien. Frontend und Backend sind zwei sehr wichtige Elemente einer aktuellen Webapplikation. Für maximale Flexibilität kann man diese Teile vollkommen trennen und eine im Browser laufende, eigene Applikation als Frontend mit einem REST-Service im Backend kommunizieren lassen.

von Fabian Gosebrink

Die Auswahl eines Frontend-Frameworks ist nicht leicht, jedoch hat sich Angular in der letzten Zeit nicht nur durch ein gutes „Separation of Concerns"-Konzept und gute Synchronisierungsmechanismen von Model, View, Architektur und hoher Performance hervorgehoben. Auch die Regelmäßigkeit, mit der neue Versionen erscheinen, das Angular CLI und nicht zuletzt der Internetgigant Google mit Long Term Support tragen dazu bei, dass mehr und mehr Businessanwendungen im Web auf die Angular-Plattform setzen.

Im Backend hat Microsoft mit ASP.NET Core spätestens seit Version 2.x den alten Mantel abgeworfen und kommt neu, schlank und vor allem schnell daher. Live-Reload, Middleware, die Geschwindigkeit und Cross-Plattform-Fähigkeit sind nur einige Gründe, warum man ASP.NET Core mehr als nur einen Blick widmen sollte.

In diesem Artikel möchte ich die Bestandteile und auch die Vorteile eines Frontends mit Angular sowie eines REST Backends mit ASP.NET Core darstellen und zeigen, wie man diese Teile einer Webapplikation programmieren kann. Als Beispiel programmieren wir einen Booktracker, der eine Merkliste für Bücher enthält, die man als gelesen markieren kann. Außerdem kann man neue Bücher hinzufügen und bestehende bearbeiten. Den kompletten Quellcode gibt es natürlich wieder auf GitHub [1].

Das ASP.NET Core Backend

ASP.NET Core bietet ein Command Line Interface (CLI) an, mit dem wir uns über Templates das Grundgerüst unseres Web-APIs erstellen lassen können [2]. Eine Kommandozeile im gewünschten Ordner und der Befehl *dotnet new webapi* in der Konsole erzeugen hierbei ein neues Web-API für uns, das wir mit *dotnet watch run* laufen lassen können. Beim Erstellen des Web-API wurde gleich der *dotnet restore*-Command ausgeführt, mit dem alle unsere NuGet-Pakete heruntergeladen und für unsere Anwendung bereitgestellt wurden. Hierbei wird ein neuer Webserver „Kestrel" in einer Konsole gehostet, der unsere Applikation hochfährt und für Requests bereitstellt. Dabei stehen uns in der neuesten Version ein HTTP- und ein HTTPS-Endpunkt zur Verfügung. *dotnet watch run* bietet zudem einen Live-Reload-Server, sodass jedes Mal, wenn wir eine Datei in unserem Projekt ändern, das Backend neu gestartet wird und wir es nicht manuell unterbrechen und wieder hochfahren müssen.

Konfiguration und Start des Web-API

ASP.NET-Core-Applikationen sind grundsätzlich Konsolenprogramme, die wir beispielsweise mit dem Befehl *dotnet run* von der Kommandozeile aus starten können. Somit ist der Startpunkt unseres Web-API eine einfache Konsolenapplikation, die uns einen Webserver bereitstellt, statt beispielsweise ein „Hello World" auf der Konsole auszugeben (Listing 1).

Falls etwaige Konfigurationen für unsere Applikation vorgenommen werden sollen, können diese Konfigurationsfiles hier eingelesen werden. Standardmäßig werden die mitgenerierten *appsettings.json* und *appsettings.*.json* zur Konfiguration herbeigezogen und appliziert. Aber es werden auch weitere Konfigurationsformate wie *.xml* oder sogar *.ini* unterstützt. ASP.NET Core erstellt also neben einem Webserver auch ein Konfigurationsobjekt, das wir in der *Startup.cs* zur Verfügung gestellt bekommen und nutzen können.

Start-up und Dependency Injecton

Basierend auf der fertigen Konfiguration, die uns mitgegeben wird, können wir nun unser Web-API konfi-

gurieren. ASP.NET Core macht dabei Gebrauch vom internen Dependency-Injection-System. In der Datei *Startup.cs* bekommen wir im Konstruktor der Klasse die Konfiguration übergeben (Listing 2). ASP.NET Core kommt also mit einem eigenen Dependency-Injection-System, auf das wir im weiteren Verlauf dieses Artikels noch eingehen werden.

Die *Startup.cs*-Datei stellt zwei weitere Methoden bereit: *ConfigureServices* und *Configure*. Erstere füllt den Dependency-Injection-Container, den wir von ASP.NET Core übergeben bekommen:

```
public void ConfigureServices(IServiceCollection services) { /*...*/ }
```

Auf diesem Container können wir unsere abhängigen Services eintragen und später in unseren Klassen via Dependency Injection injiziert bekommen und somit nutzen. Auch MVC selbst wird hier mitsamt seinen Services in den Container gelegt.

Die Methode *Configure* erstellt eine Pipeline für alle unsere Requests, bevor sie von unseren Controllern bearbeitet werden. Hierbei ist die Reihenfolge der hinzugefügten Middleware wichtig. Das heißt, jeder eingehende Request durchläuft die Middleware, die wir in dieser Methode angeben können. Ebenso die ausgehende Response, diesmal wird die Middleware jedoch in umgekehrter Reihenfolge abgearbeitet. Also können Features wie Authentication etc. hier in die Pipeline für unsere Requests „eingehängt" werden. Auch MVC selbst wird als Middleware hinzugefügt. Die Services haben wir schon in der *ConfigureServices*-Methode angegeben, die zugehörige Middleware nutzen wir mit *app.UseMvc()*. Somit kann unser Web-API Requests in Controllern empfangen, Routing nutzen etc.

Listing 1

```
public class Program
{
  public static void Main(string[] args)
  {
    CreateWebHostBuilder(args).Build().Run();
  }

  public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>();
}
```

Listing 2

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
    Configuration = configuration;
  }

  public IConfiguration Configuration { get; }
   // ...
}
```

Listing 3

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
  //
}
```

Verwenden des Dependency-Injection-Containers

In unserer Beispielapplikation brauchen wir ein Repository, mit dem wir unsere Entities in der Datenbank abspeichern können:

```
public interface IBookRepository { ... }
public class BookRepository : IBookRepository
```

Dieses Repository können wir nun in unserem Dependency-Injection-Container vor dem Interface registrieren:

```
services.AddScoped<IBookRepository, BookRepository>();
```

AddScoped sorgt hierbei dafür, dass die Instanz so lange gehalten wird, wie der Request bearbeitet wird, und dass mit jedem Request eine neue Instanz erstellt wird. Die weitere Methode *AddSingleton()* legt eine Instanz des Service beim Hochfahren der Applikation an, *AddTransient ()* würde eine Instanz pro Konstruktorinjection erstellen.

Definieren eines REST-Endpunkts mit Controllern

Der eigentliche REST-Endpunkt wird in ASP.NET Core in Controllern abgebildet. Die HTTP-Verben wie GET, POST, PUT, PATCH, DELETE etc. können in Controllern implementiert werden (Listing 3).

Das *Route*-Attribut über der Controllerklasse legt die Adresse des Endpunkts fest. Hierbei ist der Präfix herkömmlicherweise *api/*, gefolgt von einem generischen *[controller]*. Das wird ersetzt durch den Namen der Klasse ohne das Suffix „Controller". In diesem Fall also *api/values*.

Mit dem *ApiController*-Attribut legen wir fest, dass es sich um einen HTTP-Endpunkt handelt, der HTTP-Antworten verschickt. Mit dem Ableiten von *ControllerBase* verzichten wir auf alle View-Funktionalitäten, die wir im Fall einer kompletten MVC-Applikation bräuchten, da wir als Antwort keine komplett gerenderten Seiten, sondern JSON verschicken wollen.

Innerhalb dieses Controllers können wir nun den REST-Endpunkt definieren, mit dem sich unsere Bücher abspeichern und aktualisieren lassen. In unserem Beispiel erstellen wir uns einen *BooksController*, für den wir den Endpunkt *api/books* definieren. Da das Repository schon im Container registriert wurde, können wir es nun im Controller einfach injecten (Listing 4).

Um nun alle Bücher abrufen zu können, müssen wir auf das HTTP-GET-Verb reagieren (Tabelle 1) und als Antwort alle Bücher als JSON serialisiert zurückgeben (Kasten: „AddMvc()“). ASP.NET Core bietet hierbei als Rückgabetyp einer Methode unter anderem das *IActionResult*-Interface oder eine gar generische Klasse *ActionResult<T>* an, um HTTP-Antworten zu definieren. Gleichzeitig bietet das Ableiten der Controllerklasse von *ControllerBase* einen weiteren Vorteil: Man kann kleine Helfermethoden wie zum Beispiel *Ok(...)* oder *BadRequest(...)* benutzen, die dem Entwickler helfen, die richtige HTTP-Antwort mit dem korrekten HTTP-Statuscode zurück zum Client zu schicken. Das ist bei entkoppelten Systemen und einer solchen Service-Architektur von elementarer Wichtigkeit.

Eine Methode, die auf ein HTTP-Verb reagieren soll, kann mit dem jeweiligen HTTP-Verb als Attribut gekennzeichnet werden (Listing 5).

Diese Methode reagiert auf einen HTTP-GET-Aufruf und gibt einen 200er-Statuscode zurück, der für *Ok* steht. Die Helfermethode *Ok()* mit den Daten, die als Body mit der Response geschickt werden sollen, macht uns diese HTTP-Konvention sehr einfach.

Falls wir ein einzelnes Buch abfragen wollen, können wir auch Parameter im Routing angeben und auslesen. Hierfür geben wir das Routing-Attribut ebenfalls auf der Methode an und reichen es als Parameter in die Funktion selbst (Listing 6).

Falls das Buch nicht gefunden wird, geben wir einen 404-Statuscode zurück, ansonsten unseren bekannten Ok-Statuscode 200. Alle weiteren Endpunkte implementieren wir entsprechend. Ein Beispiel für einen kompletten Endpunkt kann in Listing 7 gefunden werden.

Listing 4

```
[Route("api/[controller]")]
[ApiController]
public class BooksController : ControllerBase
{
  private readonly IBookRepository _bookRepository;

  public BooksController(IBookRepository repository)
  {
    _bookRepository = repository;
  }
}
```

Listing 5

```
[HttpGet]
public IActionResult GetAll()
{
  List<Book> items = _bookRepository.GetAll().ToList();
  IEnumerable<BookDto> toReturn = items.Select(x =>
                                        Mapper.Map<BookDto>(x));
  return Ok(toReturn);
}
```

Listing 6

```
[HttpGet]
[Route("{id:int}")]
public IActionResult
GetSingle(int id)
{
  Book item = _bookRepository.
GetSingle(id);
  if (item == null)
  {
    return NotFound();
  }

  return Ok(Mapper.
Map<BookDto>(item));
}
```

Methode	Link
GET	api/books/
GET	api/books/{id}
POST	api/books/
PUT	api/books/{id}
DELETE	api/books/{id}

Tabelle 1: REST-Endpunkt

Hinzufügen von Cross-Origin Resource Sharing (CORS)

Damit unser API fähig ist, Aufrufe von anderen Domains statt nur der eigenen zu empfangen, können wir CORS konfigurieren. Da die Konfiguration des APIs über die *Startup.cs*-Datei passiert, können wir auch hier die CORS-Policy eintragen, die für uns passend ist.

Zuerst müssen wir CORS als Service hinzufügen und können es gleichzeitig konfigurieren (Listing 8).

Hierbei erstellen wir eine Regel, bei der wir nur Anfragen von unserem Development-Client der Angular Application zulassen. Diese Regel nennen wir *AllowAngularDevClient* und müssen diese nun noch in unserer Pipeline – also in der *Configure*-Methode – verwenden (Listing 9).

„AddMvc()“

Mit dem Aufruf *AddMvc()* in der *Startup*-Klasse haben wir bereits einen JSON-Serialisierer hinzugefügt, der uns die Daten automatisch von Objekten zu JSON und von JSON zu Objekten parst. ASP.NET Core liefert diese Funktionalität also out of the box.

Listing 7

```
[Route("api/[controller]")]
  [ApiController]
  public class BooksController : ControllerBase
  {
    private readonly IBookRepository _
                                    bookRepository;

    public BooksController(IBookRepository
                                    repository)
    {
      _bookRepository = repository;
    }

    [HttpGet(Name = nameof(GetAll))]
    public IActionResult GetAll()
    {
      List<Book> items = _bookRepository.
                                GetAll().ToList();
      IEnumerable<BookDto> toReturn = items.
          Select(x => Mapper.Map<BookDto>(x));
      return Ok(toReturn);
    }

    [HttpGet]
    [Route("{id:int}", Name = nameof(GetSingle))]
    public IActionResult GetSingle(int id)
    {
      Book item = _bookRepository.GetSingle(id);

      if (item == null)
      {
        return NotFound();
      }

      return Ok(Mapper.Map<BookDto>(item));
    }

    [HttpPost(Name = nameof(Add))]
    public ActionResult<BookDto> Add([
        FromBody] BookCreateDto bookCreateDto)
    {
      if (bookCreateDto == null)
      {
        return BadRequest();
      }

      Book toAdd = Mapper.
                    Map<Book>(bookCreateDto);

      _bookRepository.Add(toAdd);

      if (!_bookRepository.Save())
      {
        throw new Exception("Creating an item
                                    failed on save.");
      }

      Book newItem = _bookRepository.
                                GetSingle(toAdd.Id);

      return CreatedAtRoute(nameof(GetSingle),
                        new { id = newItem.Id },
        Mapper.Map<BookDto>(newItem));
    }

    [HttpPatch("{id:int}", Name =
                        nameof(PartiallyUpdate))]
    public ActionResult<BookDto> PartiallyUpdate(
          int id, [FromBody] JsonPatchDocument
                    <BookUpdateDto> patchDoc)
    {
      if (patchDoc == null)
      {
        return BadRequest();
      }

      Book existingEntity = _bookRepository.
                                    GetSingle(id);

      if (existingEntity == null)
      {
        return NotFound();
      }

      BookUpdateDto bookUpdateDto = Mapper.
          Map<BookUpdateDto>(existingEntity);
      patchDoc.ApplyTo(bookUpdateDto,
                                    ModelState);

      TryValidateModel(bookUpdateDto);

      Mapper.Map(bookUpdateDto, existingEntity);
      Book updated = _bookRepository.
                    Update(id, existingEntity);

      if (!_bookRepository.Save())
      {
        throw new Exception("Updating an item
                                    failed on save.");
      }

      return Ok(Mapper.Map<BookDto>(updated));
    }

    [HttpDelete]
    [Route("{id:int}", Name = nameof(Remove))]
    public IActionResult Remove(int id)
    {
      Book item = _bookRepository.GetSingle(id);

      if (item == null)
      {
        return NotFound();
      }

      _bookRepository.Delete(id);

      if (!_bookRepository.Save())
      {
        throw new Exception("Deleting an item
                                    failed on save.");
      }

      return NoContent();
    }

    [HttpPut]
    [Route("{id:int}", Name = nameof(Update))]
    public ActionResult<BookDto> Update(int id,
          [FromBody] BookUpdateDto updateDto)
    {
      if (updateDto == null)
      {
        return BadRequest();
      }

      var item = _bookRepository.GetSingle(id);

      if (item == null)
      {
        return NotFound();
      }

      Mapper.Map(updateDto, item);

      _bookRepository.Update(id, item);

      if (!_bookRepository.Save())
      {
        throw new Exception("Updating an item
                                    failed on save.");
      }

      return Ok(Mapper.Map<BookDto>(item));
    }
  }
```

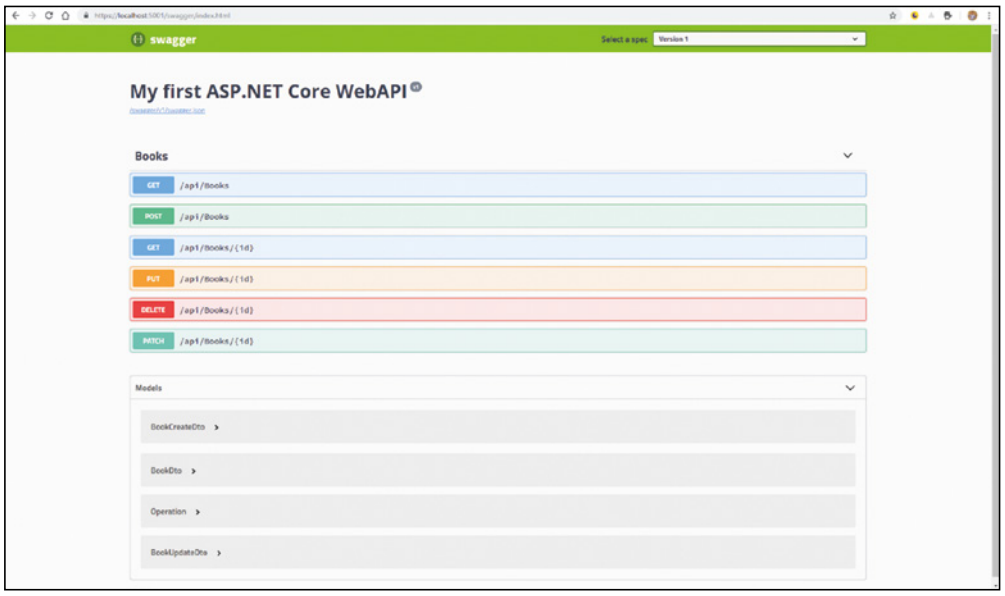


Abb. 1: Von Swagger generiertes UI

Dokumentation mit Swagger

Nicht nur wenn ein API von anderen, eventuell projektfernen Entwicklern oder Teams verwendet werden soll, bietet sich eine Dokumentation des APIs an. Auch für die bessere eigene Übersicht ist eine gute Dokumentation sehr hilfreich. Damit wir keine endlosen Word-Dokumente ausfüllen, pflegen und aushändigen müssen, gibt es die Lösung, unser API elektronisch dokumentieren zu lassen und die Dokumentation via Endpunkt zur Verfügung zu stellen. Swagger [3] bietet eine solche Lösung an, die wir mit wenigen Handgriffen in unserem Web-API nutzen können.

Mit *dotnet add Backend.csproj package Swashbuckle.AspNetCore* können wir das NuGet-Paket hinzufügen. Nachdem es in die **.csproj* eingefügt wurde, können wir es in der *Startup.cs*-Datei verwenden (Listing 10).

Swagger generiert ein JSON-File als Beschreibung unseres APIs, das wir mittels der *SwaggerUi* in ein leserlicheres User Interface gießen lassen können. Nachdem das API gestartet ist, können wir das UI via *https://localhost:5001/swagger* anzeigen lassen (**Abb. 1**).

Um das Suffix *swagger* nach der Adresse des Servers zu sparen, können wir auch den *RoutePrefix* auf einen leeren String setzen:

```
app.UseSwaggerUI(c =>
{
  c.SwaggerEndpoint("/swagger/v1/swagger.json", "Version 1");
  c.RoutePrefix = string.Empty;
});
```

Listing 8

```
services.AddCors(options =>
{
  options.AddPolicy("AllowAngularDevClient",
    builder =>
    {
      builder
        .WithOrigins("http://localhost:4200")
        .AllowAnyHeader()
        .AllowAnyMethod();
    });
});
```

Listing 9

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory,
IHostingEnvironment env)
{
  // ...
  app.UseCors("AllowAngularDevClient");
  // ...
}
```

Listing 10

```
public void ConfigureServices(IServiceCollection services)
{
  // ...
  services.AddSwaggerGen(c =>
  {
    c.SwaggerDoc("v1", new Info { Title = "My first ASP.NET Core WebAPI",
                                  Version = "v1" });
  });
  // ...
}

public void Configure(IApplicationBuilder app, ILoggerFactory
                                  loggerFactory, IHostingEnvironment env)
{
  // ...
  app.UseSwagger();
  app.UseSwaggerUI(c =>
  {
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Version 1");
  });
  // ...
}
```

Jetzt ist die Swagger-Dokumentationsseite ebenfalls unter der Serveradresse *https://localhost:5001/* sichtbar, was das Anzeigen noch einmal etwas vereinfacht.

Natürlich bietet ASP.NET Core noch viel mehr Möglichkeiten: WebSockets mit SignalR, Exception Handling, Configuration, Queryparameter und Arbeiten mit unterschiedlichen Environments sind einige der Möglichkeiten, die den Rahmen dieses Artikels sprengen würden.

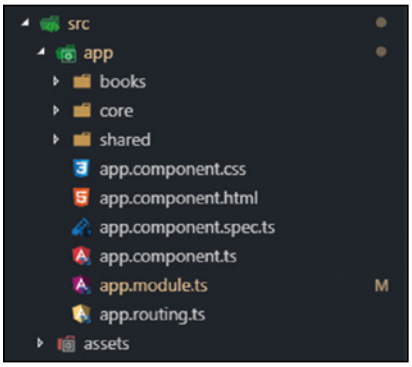
Widmen wir uns nun der Clientseite mit Angular. Im nächsten Abschnitt wollen wir das erstellte ASP.NET-Core-Web-API benutzen, Bücher eintragen, als gelesen markieren und auch wieder löschen.

Erstellen einer Angular-Applikation

Um eine Angular-Applikation zu erstellen, benutzen wir das Angular CLI, das Command Line Interface von Angular [4]. Mit *ng new angular-booktracker* können wir die Applikation erstmals erstellen lassen.

Nachdem die Applikation mit ihren Dateien erstellt wurde, können wir sie mit *npm start* starten. Ein Webserver zur Entwicklung steht uns nach ein paar Sekunden unter *http://localhost:4200* zur Verfügung. Angular baut auf dem Prinzip der Komponenten auf. Diese geben uns die Möglichkeit, unsere Applikation in viele kleine Teile zu unterteilen und diese einzeln zu implementieren.

Abb. 2: Aufteilung in Module



Aufteilung der Applikation in Module

Um eine Architektur auf dem Client zu erstellen, bietet Angular neben den Webkomponenten selbst eine weitere Abstrahierungsstufe: das Aufteilen der Applikation in verschiedene Angular-Module. Ein Modul separiert unsere Applikation in logische Bereiche und einzelne Features. Sie dienen als Container für Angular Components, Pipes, Directives etc., die von dem Modul benötigt werden. Durch die Aufteilung in Module wird die Applikation wartbarer, übersichtlicher und einfacher testbar (**Abb. 2**).

Für unseren Booktracker sehen wir drei Module vor: ein *BooksModule*, ein *CoreModule* und ein *SharedModule*. Das *AppModule*, das auch zum Starten unserer Applikation benutzt wird, ist natürlich ebenfalls dabei.

Das *BooksModule* ist für das Buchfeature zuständig. Das *CoreModule* bietet einen Container für alle unsere Services, das *SharedModule* wird in unserem Beispiel zur Abstrahierung von Model-Klassen und den Angular-Material-Modulen benutzt, und das *AppModule* brauchen wir, um unsere Applikation zu starten; zudem dient es als Einstiegsmodul.

Module bieten außerdem die Möglichkeit des Lazy Loadings. Wir können also das erforderliche Featuremodul erst dann laden, wenn der Benutzer es explizit

Listing 11

```
export const AppRoutes: Routes = [
  { path: '', redirectTo: 'books', pathMatch: 'full' },
  {
    // Falls dieser Pfad angewählt wird ...
    path: 'books',
    // ... lade dieses Modul nach
    loadChildren: './books/books.module#BooksModule'
  },
  {
    path: '**',
    redirectTo: 'books'
  }
];
```

anwählt, oder wir laden es automatisch, nachdem alle Module ohne Lazy Loading bereits geladen wurden.

Mithilfe der Routen können wir das Lazy Loading im *app.routing.ts* festlegen (Listing 11).

Verwenden von Angular Material

Um fertige UI-Controls wie eine Liste oder ein Menü benutzen zu können, greifen wir auf Angular Material zurück [5]. Alle benötigten Elemente werden bei Angular Material in einzelnen Modulen exportiert, die wir in einem *material.module.ts* zusammenfassen und unserer Applikation mithilfe des *SharedModule* zur Verfügung stellen (Listing 12).

Das *SharedModule* re-exportiert nun das *MaterialModule,* um anderen Modulen, die das *SharedModule* importieren, ebenfalls Zugriff auf die *Exports* des *MaterialModule* zu bieten (Listing 13).

Die Kommunikation mit dem API

Bevor wir Daten anzeigen oder darstellen können, müssen wir sie vom API abrufen, das wir soeben erstellt haben. Dazu erstellen wir einen Service, der uns die Kommunikation mit dem API abstrahiert und mittels Methoden zur Verfügung stellt. Der *HttpClient* aus dem *@angular/common/http*-Modul sowie der Import des *HttpClientModule* aus Angular stellt uns genau die HTTP-Verben zur Verfügung, die wir brauchen (Listing 14).

Die *@Injectable({ providedIn: ‚root' })*-Syntax fügt unseren Service in den Root Injector von Angular ein. Somit steht der Service in unserer Anwendung zur Verfügung. Durch die Syntax *constructor(private readonly httpBase: HttpClient) {}* benutzen wir die Dependency Injection von Angular auf der einen Seite, auf der anderen Seite registrieren wir eine private Property in der Klasse *BookService* namens „*http*", auf das wir von allen Methoden aus zugreifen können.

Um nun beispielsweise alle Bücher abzufragen, schicken wir einen GET-Request an den URL *https://localhost:5001/api/books*:

```
getAllBooks() {
    return this.http.get<Book[]>(this.url);
}
```

Wir rufen die GET-Methode auf, legen den generischen Rückgabetyp auf *Book[]* fest und verwenden als Ziel den URL, der im Service fix hinterlegt ist (Kasten „Konfiguration des URLs"). Der Rückgabetyp der Methode ist das Observable, also der Stream der Daten, die wir von dem REST Call erwarten. Von außen kann man sich auf die Antwort registrieren und, je nach Antwort, reagieren.

Die *Book*-Klasse ist eine reine DTO-Klasse, die wir gemäß der zu erwartenden JSON-Antwort festlegen können. Durch die Angabe des Typs wird die JSON-Antwort automatisch in diese Klasse serialisiert (Listing 15).

Die weiteren Methoden auf dem Service sind genau die, die wir am API implementiert haben und auch in unserer Applikation brauchen (Listing 16).

Anzeigen der Daten

Um die Daten anzuzeigen, erstellen wir Components, die ein HTML-Template haben. Wir geben den eben erstellten Service via Dependency Injection in die Components hinein und rufen die entsprechenden Methoden auf.

Man kann hierbei Komponenten in zwei verschiedene Arten aufteilen: Presentational Components und Container Components [6]. Erstere fokussieren sich darauf, wie Daten dargestellt werden, aber kümmern sich nicht darum, wie Daten geladen werden oder woher sie kommen. Container Components haben eine Abhängigkeit zu einem Repository oder Daten-Service und empfangen Events von Presentational Components. Presentational Components bekommen die Daten übergeben und

Listing 12

```
import { NgModule } from '@angular/core';
import { ... } from '@angular/material';

const materialModules: any[] = [
  // ... alle benötigten Material Modules
];

@NgModule({
  imports: materialModules,
  exports: materialModules
})
export class MaterialModule {}
```

Listing 13

```
import { NgModule } from '@angular/core';
import { MaterialModule } from './material.module';

@NgModule({
  imports: [MaterialModule],
  exports: [MaterialModule]
})
export class SharedModule {}
```

Listing 14

```
@Injectable({ providedIn: 'root' })
export class BookService {
  private url = `https://localhost:5001/api/books`;
  constructor(private readonly http: HttpClient) {}
  // alle Methoden
}
```

Listing 15

```
export class Book {
  id: number;
  read: boolean;
  title: sting;
  author: string;
  description: string;
  genre: string;
}
```

Konfiguration des URLs

Der URL kann durch verschiedene Möglichkeiten konfigurierbar gemacht werden: Environments, Config-Url-Requests etc.

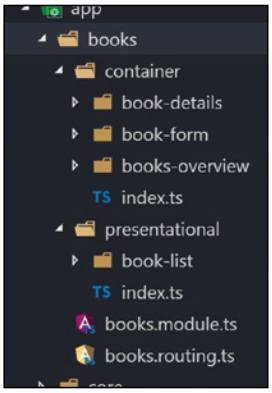


Abb. 3: Komponentenaufteilung im Projekt

kümmern sich um die Darstellung, während Container Components sich darum sorgen, wo die Daten herkommen, und über Services mit beispielsweise einer REST-Schnittstelle kommunizieren können. Das macht Presentational Components extrem wiederverwendbar und die Stellen im Code, die den Status einer Applikation manipulieren, übersichtlicher.

So können wir auch die Komponenten in unserem Projekt aufteilen (**Abb. 3**).

List Component

Die *books-list.component.ts* empfängt eine Collection von Büchern als Input und kümmert sich um die visuelle Darstellung dieser Auflistung mit einer Liste aus Angular Material. Sie sagt der sie verwendenden Component mittels eines Events Bescheid, falls jemand ein Buch als gelesen markiert hat.

Angular bietet uns hierfür die *@Input()*- und *@Output()*-Decorators an, mit denen wir eingehende Daten und ausgehende Events auf Properties der *Component*-Klasse markieren können (Listing 17).

Um diese Properties mit Daten zu füllen bzw. sich auf Events der Component zu registrieren, verknüpfen wir die Component im HTML mit der Parent Component: Sie verwendet die Child Component mittels ihres Selectors im HTML, bindet Daten an die Properties und registriert sich auf Events (Listing 18).

Wir binden also zwei Properties der Parent Component *readBooks* und *unreadBooks* an die Input-Property der Child Component *app-book-list*, die wir hier sogar wiederverwenden können.

Listing 16

```
getAllBooks() {
  return this.http.get<Book[]>(this.url);
}

getSingle(bookId: number) {
  return this.http.get<Book>(`${this.url}/${bookId}`);
}

update(updated: Book) {
  return this.http.put<Book>(`${this.url}/${updated.id}`, updated);
}

add(book: Book) {
  return this.http.post<Book>(this.url, book);
}

delete(bookId: number) {
  return this.http.delete(`${this.url}/${bookId}`);
}
```

Listing 17

```
@Component({
  /*...*/
})
export class BookListComponent implements OnInit {
  @Input()
  books: Book[] = [];

  @Output()
  bookReadChanged = new EventEmitter();
  // ...
}
```

Übersichts-Component

In der *Component*-Klasse der Parent Component füllen wir diese beiden Listen ab (Listing 19).

Via Dependency Injection bekommen wir den *BookService* in den Konstruktor übergeben und speichern ihn in einer internen Variablen *bookService*. Danach speichern wir das Observable für alle Bücher, ungeachtet des *read*-Status, in einer Observable und erstellen zwei weitere Observables: *unreadBooks$* und *readBooks$* (Kasten „$-Suffix“).

Async Pipe

Damit wir uns nicht manuell via *subscribe(...)* an die Auflösung der Observable hängen müssen, können wir sie an das Template binden und die Daten mit der Async Pipe auflösen. Das bietet den Vorteil, dass unser Code in der Component leserlicher wird und wir

Listing 18

```
<mat-tab-group dynamicHeight>
  <mat-tab *ngIf="unreadBooks$ | async as unreadBooks">
    <app-book-list [books]="unreadBooks" (bookReadChanged)=
                          "toggleBookRead($event)"></app-book-list>
  </mat-tab>
  <mat-tab *ngIf="readBooks$ | async as readBooks">
    <app-book-list [books]="readBooks" (bookReadChanged)=
                          "toggleBookRead($event)"></app-book-list>
  </mat-tab>
</mat-tab-group>
```

„$“-Suffix

Das *$*-Zeichen am Ende einer Variable (auch finnische Notation genannt) macht es uns Entwicklern einfacher, zu erkennen, hinter welcher Variablen ein bereits aufgelöster Wert liegt und welche Variable einen Stream darstellt, dessen Wert irgendwann in Zukunft aufgelöst wird.

uns nicht um das *unsubscribe* kümmern müssen, falls die Component von Angular wieder zerstört wird. Die Async Pipe macht das automatisch für uns [7].

Listing 19

```
export class BooksOverviewComponent implements OnInit {
  unreadBooks$: Observable<Book[]>;
  readBooks$: Observable<Book[]>;

  constructor(private readonly bookService: BookService) {}

  ngOnInit() {
    this.getAllBooks();
  }

  private getAllBooks() {
    const allBooks$ = this.bookService.getAllBooks().pipe(
      publishReplay(1),
      refCount()
    );

    this.unreadBooks$ = allBooks$.pipe(
      map(books => books.filter(book => !book.read))
    );

    this.readBooks$ = allBooks$.pipe(
      map(books => books.filter(book => book.read))
    );
  }
}
```

Listing 20

```
@NgModule({
  declarations: [/*...*/],
  imports: [
    // ...
    RouterModule.forRoot([
      { path: '', redirectTo: 'books', pathMatch: 'full' },
      {
        path: 'books',
        loadChildren: './books/books.module#BooksModule',
      },
       {
      path: '**',
      redirectTo: 'books',
    },
    ], { useHash: true }),
  ],
  providers: [],
  bootstrap: [...],
})
export class AppModule {}
```

Um die Daten der Observable im Template an eine Variable binden zu können, bietet Angular uns eine **ngIf-as*-Syntax an, die wir hier im Einsatz sehen:

```
<mat-tab *ngIf="unreadBooks$ | async as unreadBooks" label="to buy
({{unreadBooks.length}})">
  <!-- child component -->
</mat-tab>
```

**ngIf="unreadBooks$ | async as unreadBooks"* füllt uns den Inhalt der Observable an eine Variable *unreadBooks*, die wir im Scope des HTML-Elements verwenden.

Jede Verwendung einer Async Pipe im Template gleicht also der Verwendung eines *subscribe(...)* im TypeScript-Code. Dadurch, dass wir zwei Async Pipes verwenden, feuern wir die Observable zweimal, sodass eigentlich zwei Requests an unseren Server geschickt werden müssten.

Damit wir den Server und unser API nicht unnötig belasten, können wir einen „Cache" einbauen, den RxJs uns anbietet:

```
const allBooks$ = this.bookService.getAllBooks().pipe(
  publishReplay(1),
  refCount()
);
```

publishReplay(1), refCount() speichert uns den letzten Call auf der Observable, ohne nochmal einen neuen Aufruf zum API zu starten. Die Alternative wäre ein manuelles *subscribe(...)*, das Filtern des Ergebnisses mit der *filter(...)*-Funktion und die Zuweisung in zwei Arrays

auf der Component. Dieser Mehraufwand würde die Async Pipe obsolet machen.

Routing

Um zwischen verschiedenen Templates von Components zu wechseln, arbeiten wir mit Routing auf der Clientseite. Um das Routing zu aktivieren, importieren wir das *RoutingModule* im *AppModule* und konfigurieren es mit der *forRoot(...)*-Methode (Listing 20).

Somit können wir festlegen, bei welchem Begriff (*path*) zu welcher Komponente gesprungen werden soll. Mit der Property *loadChildren* und der bereits gezeigten Syntax können wir das Lazy Loading dieses Moduls aktivieren und die Routen des Child-Moduls verwenden. Ebendiese können wir mit der Methode *forChild(...)* konfigurieren, sie sollte zum Konfigureren in der Applikation nur einmal auf dem *AppModule* verwendet werden (Listing 21).

Somit definiert das *BooksModule* seine eigenen Routen und wird komplett – also mit seinen Routen – geladen. Der Verweis im App Module ist quasi der Einstiegspunkt zum Modul. Routen des Root-Moduls und des Child-Moduls werden einfach konkateniert.

Um die Templates der Komponenten darstellen zu können, benötigen wir einen auswechselbaren Teil, den Angular mit den Templates der Komponenten ersetzt.

Das *RouterModul* exportiert hierbei das *router-outlet*, das uns genau diese Funktionalität bereitstellt. Im Template der *AppComponent* können wir dies an die Stelle schreiben, die dynamisch ersetzt werden soll:

```
<mat-sidenav-container class="app-root" fullscreen>
  <!-- ... -->
  <router-outlet></router-outlet>
  <!-- ... -->
</mat-sidenav-container>
```

Basierend auf der Route wird an der Stelle von *router-outlet* jetzt die jeweilige Komponente angezeigt.

Forms

Natürlich kann man Informationen mit Angular-Daten nicht nur anzeigen, sondern sie auch an den Server senden. Wir haben die entsprechende POST-Methode am Server und in unserem Service schon implementiert.

Um dem Benutzer die Möglichkeit zu geben, neue Bücher zu erstellen oder zu bearbeiten, können wir Reactive Forms verwenden. Zuallererst importieren wir das *ReactiveFormsModule* und alles, was dieses Modul exportiert, in unser Modul (Listing 22).

Mit Reactive Forms [8] erstellen wir die Form im Code der Component, binden die erstellte *FormGroup* an das Template und stellen die *FormControls* dar (Listing 23).

Wir können der Property *form* eine neue *FormGroup* zuweisen, die ein Objekt mit Properties bekommt, die die einzelnen *FormControls id: new FormControl(,')* etc. darstellen. Als Parameter empfängt das *FormControl*-Objekt hier den Defaultwert. Man kann zusätzlich auch Validatoren oder Optionen mit angeben. In unserem Fall geben wir den Required Validator mit, der festlegt, dass die Form diesen Wert benötigt, um valide zu sein.

Ist dieses Objekt abgefüllt, können wir es im Template verwenden und unsere Form aufbauen (Listing 24).

Listing 21

```
@NgModule({
  imports: [
    RouterModule.forChild([
      { path: '', redirectTo: 'overview', pathMatch: 'full' },
      { path: 'overview', component: BooksOverviewComponent },
      { path: 'create', component: BookFormComponent },
      { path: 'edit/:id', component: BookFormComponent },
      { path: 'details/:id', component: BookDetailsComponent }
    ])
  ]
})
export class BooksModule {}
```

Listing 22

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // ...
    ReactiveFormsModule
  ]
})
export class BooksModule {}
```

Listing 23

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  /*...*/
})
export class BookFormComponent implements OnInit {
  form: FormGroup;

  ngOnInit() {
    this.form = new FormGroup({
      id: new FormControl(''),
      title: new FormControl('', Validators.required),
      author: new FormControl('', Validators.required),
      description: new FormControl('', Validators.required),
      genre: new FormControl('')
    });
  }
}
```

Wir erstellen eine Form mit dem HTML-*<form>*-Tag und binden sie an die Direktive *formGroup*, die vom *ReactiveFormsModule* exportiert wird. Innerhalb dieses HTML-Form-Tags stehen dann die Controls der Form zur Verfügung, die wir an die entsprechenden HTML Controls binden können.

Um die Form abzusenden, bietet uns Angular eine weitere Direktive *ngSubmit* an, die wir wie folgt verwenden können:

```
<form (ngSubmit)="addBook()" [formGroup]="form">
```

Die Methode *addBook()* wird also mittels Event Binding aufgerufen, wenn die Form abgeschickt wird. Der nächste Schritt ist ein Button, mit dem wir die Form abschicken können:

```
<form (ngSubmit)="addBook()" [formGroup]="form">
  <!-- more controls -->
  <button [disabled]="form.invalid || form.pristine">Add Book</button>
</form>
```

Angular bietet uns die Möglichkeit an, HTML Properties zu binden, wie im Codebeispiel die *disabled*-Property des Buttons. Wir können den Button inaktiv setzen, wenn die Form selbst *invalid* ist, also mindestens eine ihrer Controls nicht valide ist (*form.invalid*) oder die Form noch nicht verändert wurde (*form.pristine*).

Listing 24

```
<form [formGroup]="form">
  <div class="form-container">
    <input formControlName="title">
    <input formControlName="author">
    <textarea formControlName="description"></textarea>
   <!-- more controls -->
  </div>
</form>
```

Listing 25

```
export class BookFormComponent {
  constructor(
    private readonly bookService: BookService,
    private readonly notificationService: NotificationService
  ) {}

  addBook() {
    this.bookService
      .add(this.form.value)
      .subscribe(() => this.notificationService.show('Book added'));
  }
}
```

Die Methode *addGroup()* auf der Component benutzt erneut den zur Verfügung gestellten *BooksService*, um das Buch abzusenden. Der Property Value auf der Form stellt uns alle Form-Controls zur Verfügung, die wir so absenden, am Backend entgegennehmen und eintragen können (Listing 25).

Damit der Benutzer weiß, dass das Hinzufügen erfolgreich war, können wir noch eine Meldung anzeigen oder anderweitig entsprechend reagieren.

Zusammenfassung

Wir können mit einer serverseitigen Technologie wie ASP.NET Core moderne und flexible Backend-Applikationen bauen, die wir nicht nur von Angular aus konsumieren können. Tools wie das *dotnet* CLI, die Dependency Injection, das automatische Neustarten des Servers, die Geschwindigkeit, und dass man zum Entwickeln nicht nur Visual Studio, sondern jeden beliebigen Editor benutzen kann, machen ASP.NET Core extrem flexibel und das Erstellen von Webapplikationen auch jenseits des Web-API sehr einfach. Die Trennung von Front- und Backend setzt dazu auf eine sehr hohe Flexibilität und Entkopplung.

Angular als moderne Webplattform – mit Tooling wie etwa dem Angular CLI und Features wie Dependency Injection, das Aufteilen in Module, Lazy Loading etc. – ermöglicht es, auch große Applikationen im Web mit Struktur und Architektur in TypeScript umzusetzen. Wer hätte vor ein paar Jahren gedacht, dass man solche Architekturen – auf dem Client und letztendlich mit JavaScript – entwickeln kann?

Fabian Gosebrink ist Google Developer Expert für Angular und Webtechnologien, Microsoft MVP und Webentwickler im Bereich ASP.NET Core und Angular. Als Professional Software Engineer, Consultant und Trainer berät und unterstützt er Kunden bei der Umsetzung von Webapplikationen im Front- bzw. Backend bis hin zum mobilen Bereich.

Links & Literatur

[1] https://github.com/FabianGosebrink/Angular-Booktracker

[2] https://www.microsoft.com/net/download

[3] https://swagger.io/solutions/api-documentation/

[4] http://cli.angular.io

[5] https://material.angular.io/

[6] https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/

[7] https://angular.io/guide/pipes#the-impure-asyncpipe

[8] https://angular.io/guide/reactive-forms

Dependency Injection in unter 200 Zeilen Code – ohne Leerzeilen

Injections für echte TypeScript-Junkies

Dependency Injection hat sich bei den meisten Softwareprojekten, Frameworks und Apps durchgesetzt. Dennoch ist es erstaunlich, wie viele Fragen man in diversen Foren im Kontext von JavaScript Frameworks zu diesem Thema findet. Muss Dependency Injection kompliziert sein? Wir meinen nicht. Mit TypeScript MetaData und Decorators lässt sich eine sehr kompakte, aber wirkungsvolle Dependency Injection aufbauen.

von Thomas Mahringer

Spätestens seit dem Java Spring Framework hat sich Inversion of Control oder Dependency Injection (DI) als Designpattern etabliert. Die damals noch nicht so verbreitete Idee war es, Methoden, Klassen und Module so zu entwerfen, dass sie durch ihre Schnittstellen wirklich vollständig beschrieben sind. Das bedeutet, dass sie kein implizites oder globales Wissen mitbringen müssen, um korrekt zu funktionieren. Ein Gegenbeispiel dazu wären verschiedene Arten von Service-Locator-Patterns oder, noch schlimmer, mehrere globale Objekte, die Services bereitstellen. Diese Muster sind Antimuster, bei denen Methoden wissen, wo sie nachsehen müssen, um benötigte Services zu beziehen, z. B. für Datenzugriff, Authentifizierung usw. Der Zugriff auf die Services erfolgt global, beispielsweise über *static*-Methoden wie *ServiceLocator.getDataSource()*, *DataAccess.getDataSource()* oder *Security.getAuthenticationService()*. Je mehr von diesen globalen Abhängigkeiten umherschwirren, desto undurchschaubarer wird der Code, und er lässt sich vor allem nicht mehr modular Unit-testen, da sämtliche globalen Objekte in den richtigen Testzustand versetzt werden müssen. Bei der DI oder Inversion of Control läuft es eben umgekehrt: Nicht die einzelnen Codefragmente wissen, woher sie Services beziehen, sondern Letztere werden ihnen beim Aufruf einfach mitgegeben, z. B. als Konstruktorargumente. Damit sind jedes Codestück, jede Klasse und Methode wirklich eindeutig durch ihre Schnittstellen definiert.

Was hat das mit JavaScript zu tun?

In der guten alten Zeit, in der Web-Apps serverseitig, z. B. über MVC-Frameworks, gerendert wurden, beschränkte sich JavaScript auf die dynamische Manipulation des DOMs und evtl. auch noch auf Ajax-Datentransfer. jQuery, ähnliche Frameworks und ihr Programmiermodell saßen fest im Webstack-Sattel. Wie Sie wissen, hat sich die Situation mit voll ausgewachsenen JavaScript-Clientapplikationen stark verändert, denn sie sind eigentlich Rich Clients (Single-Page-Applikationen), die mit Backends kommunizieren. Statt jQuery und Co. finden wir MVVC-Patterns und reaktive Programmierung. Frameworks wie React.js, Angular und Vue.js sind wohlbekannt, und mit Progressive Web Apps wird die Lücke zwischen echter App und Web-App nochmal deutlich verkleinert.

Wenn wir heute von JavaScript-Clients sprechen, meinen wir daher in der Regel ausgewachsene Softwaresysteme, bei denen wir den gleichen Herausforderungen gegenüberstehen wie bei traditionellen Rich Clients. Daher wird es ganz selbstverständlich, dass wir zum Designen unserer Softwarebausteine Designpatterns wie Dependency Injection verwenden.

TypeScript Fanboy

Wie die regelmäßigen Leser vermutlich wissen, bin ich ein TypeScript-Fanboy. Bei den heutigen komplexen (mobilen) Web-Apps wird eine statische Typisierung, die die Flexibilität von JS nicht einschränkt, zum ALM Lifesaver. Wenn man dann noch Frameworks wie beispielsweise React.js verwendet, die UI-Komponenten

nach JavaScript/TypeScript transpilieren, hat man ein langersehntes Ziel der UI-Entwicklung erreicht: UI-Komponenten sind statisch typisiert und werden vom Compiler überprüft. Vorbei sind die Zeiten, in denen man erst während des Betriebs bemerkt, dass aufgrund von Typos im Data Binding Fehler zur Laufzeit auftreten. Hatten Sie auch schon mal das Property *fristName* als Data Binding in einer UI-Komponente?

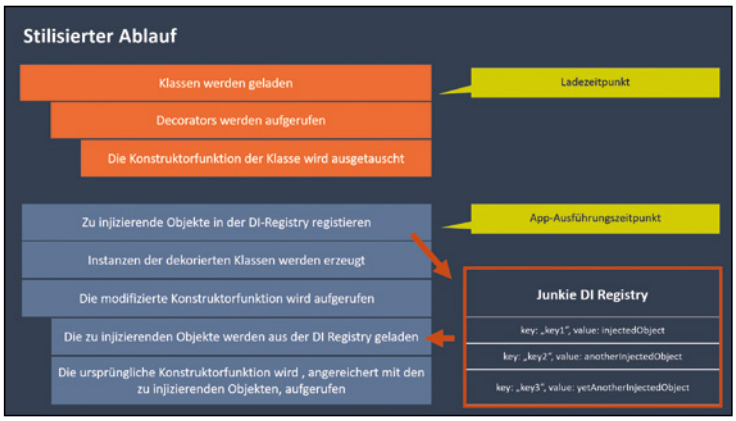


Abb. 1: Vereinfachter Ablauf: Laden des Codes und Ausführen der App

TypeScript Decorators und Metadata

Aber genug der Schwärmerei. TypeScript-Code lässt sich ähnlich wie in .NET mit Attributes oder wie in Java mit Annotations mit Metainformationen anreichern, die in TypeScript „Decorators" heißen. Einen ersten Decorator sehen wir hier:

```
@injectionTarget()
class TestClass {

}
```

Die Klasse *TestClass* wird mit dem Decorator *@injectionTarget()* dekoriert. Daher wird dieser Decorator Klassen-Decorator genannt. Wie wir sehen werden, gibt es u. a. auch noch Parameter-Decorators.

Unsere Testklasse erhält dadurch die Metainformation, dass offensichtlich irgendetwas in diese Klasse injiziert wird. Aber bei TypeScript bzw. JavaScript kommt ein entscheidender Vorteil einer interpretierten Skriptsprache zum Tragen: Wenn eine Skriptdatei geladen wird (über *<script>*-Tags, *import*, *require* oder Ähnliches), wird der Code sequenziell durchlaufen und ausgeführt und die entsprechenden JS-Objekte werden angelegt. Es wird in unserem Fall z. B. das Klassenobjekt (oder JS-Konstruktorfunktionsobjekt) *TestClass* angelegt. Diese dynamische Lade- und Ausführungssequenz bietet ein paar spannende Möglichkeiten wie das Modifizieren von Klassenobjekten beim Laden. Und genau das macht den Unterschied zwischen .NET Attributes und TypeScript Decorators aus: Decorators sind Funktionen, nicht nur Labels oder Tags.

„Ja, und?" kann man jetzt fragen. Und die Antwort lautet: „Man kann in Decorators alles tun – alles, was man in JS-Funktionen auch tun kann." Um es gleich konkreter zu machen, werfen wir einen Blick auf Listing 1, wo wir sehen, was der TS-Transpiler aus dem *@injectionTarget()* macht.

Es wird die Funktion *__decorate* aufgerufen, die als Parameter unseren Decorator erhält. Ohne auf die Details der *__decorate*-Funktion einzugehen, führt das dazu, dass zur Ladezeit unserer Testklasse der Decorator mit der Klasse als Parameter aufgerufen wird. Das ist auch schon die Hauptidee: Wir haben nun die Klasse in der Hand und können sie manipulieren. Wir werden später noch auf die Codedetails des Decorators eingehen.

Grundidee der Dependency Injection mittels Decorators

Ein Klassen-Decorator kann also eine Klasse manipulieren. Wir werden dadurch die Konstruktorfunktion so modifizieren, dass wir beim Erzeugen einer Instanz der Klasse die zu injizierenden Daten/Objekte hineinschmuggeln können. Der grundlegende Ablauf (**Abb. 1**) ist der folgende:

Listing 1: Der Transpiler erzeugt aus Decorators Funktionsaufrufe – „testClass.js"

```
let TestClass = class TestClass {

};
TestClass = __decorate([
  junkie_1.injectionTarget(),
  ...
  __metadata("design:paramtypes", [String, String,
Object, Object, InjectedTestClass, IDummyInterface])
], TestClass);
```

Listing 2: Parameter-Decorators

```
@injectionTarget()
class TestClass {
  constructor(public arg0: string, public arg1: string,
    // Inject a named value.
    @inject("key1") public arg2?: any,

    // Inject a singleton instance of a class.
    @inject(InjectedTestClass) public arg3?: any,

    // Inject the singleton instance of a class.
    // The class is taken from the parameter type.
    @inject() public arg4?: InjectedTestClass,

    // Inject the singleton instance of an interface.
    // The interface-class is taken from the parameter type.
    @inject() public arg5?: IDummyInterface,
  ) {
    console.log("TestClass.ctor: args == " + JSON.stringify(arguments));
  }
};
```

Listing 3: Die instrumentierte JS-Klasse inklusive Argument-Deocrators

```
let TestClass = class TestClass {
  constructor(arg0, arg1,
  // Inject a named value.
  arg2,
  // Inject a singleton instance of a class.
  arg3,
  // Inject the singleton instance of a class.
  // The class is taken from the parameter type.
  arg4,
  // Inject the singleton instance of an interface.
  // The interface-class is taken from the parameter type.
  arg5) {
    this.arg0 = arg0;
    this.arg1 = arg1;
    this.arg2 = arg2;
    this.arg3 = arg3;
    this.arg4 = arg4;
    this.arg5 = arg5;
    console.log("TestClass.ctor: args == " + JSON.stringify(arguments));
  }
};
TestClass = __decorate([
  junkie_1.injectionTarget(),
  __param(2, junkie_1.inject("key1")),
  __param(3, junkie_1.inject(InjectedTestClass)),
  __param(4, junkie_1.inject()),
  __param(5, junkie_1.inject()),
  __metadata("design:paramtypes", [String, String,
    Object, Object, InjectedTestClass, IDummyInterface])
], TestClass);
```

- Beim Laden der Skriptdateien werden die Klassenobjekte angelegt (Listing 1: *let TestClass = class TestClass…*)
  - Pro Klasse wird der Klassen-Decorator *@injectionTarget()* aufgerufen.
    - Der Decorator erzeugt eine neue Klasse, die von der originalen erbt (*extends*).

- Zur Ausführungszeit der Applikation:
  - Es werden die zu injizierenden Objekte/Daten in der DI Registry registriert.
  - Es werden Instanzen der dekorierten Klassen erzeugt, z. B. *testObject = new TestClass(…)*.
    - Es wird der *constructor* der neuen, vom Decorator erstellten Klasse aufgerufen.
      - Diese holt aus der DI Registry die zu injizierenden Objekte/Werte.
      - Er ruft den Original-Constructor auf und fügt dabei die zu injizierenden Objekte/Daten als Parameter ein.

Aber halt!

Woher wissen wir, welche Parameter durch injizierte ersetzt werden? Bisher haben wir nur darüber gesprochen, dass die Hauptmagie darin besteht, dass der Klassen-Decorator eine neue Klasse anlegt und deren *constructor* irgendwie die zu injizierenden Objekte hineinschmuggelt. Aber woher weiß der neue *constructor*, welche Parameter zu injizieren sind? An dieser Stelle kommen die sogenannten Parameter-Decorators zum Einsatz, wie wir sie in Listing 2 sehen.

Parameter-Decorators sind ebenfalls Funktionen, die zum Ladezeitpunkt der Klasse pro Parameter aufgerufen werden. In Listing 3 wird der komplette, transpilierte JS-Code gezeigt.

Dem Parameter-Decorator werden das Klassenobjekt sowie der Parameterindex übergeben. Letzterer ist

Listing 4: TypeScript-Trick: Interfaces als „class“ definieren

```
/**
 * Note: We deliberately use "class" because TS does not keep
 * interface info at runtime.
 * => This is a workaround for injecting interfaces.
 */
class IDummyInterface {
  propOfDummyInterface: string
}

/**
 * Use TypeScript "Trick": implement IDummyInterface
 * although it is defined as "class".
 */
class DummyInterfaceImpl implements IDummyInterface {
  propOfDummyInterface: string
};
```

Listing 5: Parameter-Decorator

```
/**
 * Class interface.
 * TypeScript way of declaring class parameter types,
 * e.g. in generic functions.
 */
export interface IClass<T> {
  new(...varargs: any[]): T;
  prototype: any;
  name: string;
}

/**
 * Decorator to inject a value either by key or by class.
 * Classes must be registered by {@link registerC} and ordinary
 * values/objects by {@link register}.
 * If key is a class, the key for looking up the value in the registry
 * is generated by the same mechanism as in {@link registerC}.
 * If key == undefined (so neither a value nor a class), the parameter
 * type must be a class. This class will be used as the key.
 * @param key
 */
export function inject<T>(key?: string | IClass<T> | Symbol | undefined) {
  let keyAny: any = key;
  let clazz: IClass<any> | undefined = undefined;

  // "Setup-Phase" of decorator:
  // Use the parameters passed by @inject().

  // Is the key a ES6 class?
  if (keyAny && MrReflect.isClass(keyAny)) {
    clazz = (key as IClass<T>);
    key = keyAny = Junkie.generateInjectorKey(keyAny);
  } else if (key instanceof Symbol) {
    keyAny = key.toString();
  } else if (typeof key === "string" || key instanceof String) {
    keyAny = key;
  }

  /**
   * Execution phase of the decorator: Handle the parameter.
   */
  return function (target: Object,
    propertyKey: string | symbol, paramInx: number): any {
    preventUnused(propertyKey);
    // If no key is given, we use the parameter position
    // and its type to generate the DI lookup key.
    if (!keyAny) {
      let ctorParamTypes = Reflect.getMetadata("design:paramtypes",
                                                          target);
      let currentParamType = ctorParamTypes[paramInx];
      // Helper function: Check if ES6 class. If not => Error.
      if (!MrReflect.isClass(currentParamType)) {
        throw new RumbleError(
          `Neither key nor class was passed to @inject.
           This is valid only if parameter has a class type
           because it will used as key for the DI registry.
           But parameter number ${paramInx} is of type
           ${currentParamType.name}`);
      }
      // Generate a hash, same as in {@link registerC}
      keyAny = Junkie.generateInjectorKey(currentParamType);
      clazz = currentParamType;
    }
    // Save the arg descriptor in the class object.
    Junkie.pushInjectedCtorArgDescriptor(
      (target as IClass<any>),
      { paramInx, key: keyAny }
    );
  }
}

/**
 * Descriptor for injected ctor arguments.
 */
export interface InjectCtorArgDescriptor {
  paramInx: number;
  key: string;
}

/**
 * Get the descriptors for injected ctor-arguments.
 * @param clazz
 */
static getInjectedCtorArgsDescriptors(clazz: IClass<any>):
  InjectCtorArgDescriptor[] {
  Ass.defined(clazz, "clazz");
  let injectedArgDescriptors =
    (clazz as any)[Junkie.classCtorInjectedArgsDescriptors] =
    (clazz as any)[Junkie.classCtorInjectedArgsDescriptors] || [];
  return injectedArgDescriptors;
}

/**
 * Save a descriptor for an injected ctor-argument.
 * @param clazz
 * @param argDescriptor
 */
static pushInjectedCtorArgDescriptor(clazz: IClass<any>,
  argDescriptor: InjectCtorArgDescriptor) {
  Ass.defined(clazz, "clazz");
  Ass.defined(argDescriptor, " argDescriptor ");
  let injectedArgs = Junkie.getInjectedCtorArgsDescriptors(clazz);
  injectedArgs.push(argDescriptor);
}
```

der Schlüssel für das Injizieren der Objekte: Im Decorator speichern wir den Parameterindex und den Key (z. B. *key1* aus Listing 3). Damit ist es im *constructor* der neuen Klasse möglich, den passenden Parameter mit dem passenden Objekt aus der DI Registry zu ersetzen. Die Details und der genaue Code dazu folgen noch.

Der Key zum Ablegen ist der Schlüssel

Die DI Registry speichert also die zu injizierenden Objekte mit dem zugehörigen Schlüssel ab (**Abb. 1**). Welche Arten von Schlüsseln wollen wir dabei unterstützen? Im einfachsten Fall verwenden wir einen String als Schlüssel, wie im Beispiel des *arg2* aus Listing 2.

Oft geht es aber um zentrale Services, z. B. Data Access Service, die als Singleton-Instanz der jeweiligen Service-Klasse leben. Dazu ist es hilfreich, nicht von Hand einen Schlüssel vergeben zu müssen, sondern einfach die Klasse selbst als Schlüssel zu verwenden, wie bei *arg3* aus Listing 2. Dazu wird ein Hash der Klasse als Schlüssel für die DI Registry verwendet.

Falls beim *@inject()* gar kein Key als Parameter übergeben wird, wird der Datentyp des Parameters zur Key-Generierung verwendet, wie bei *arg4* aus Listing 2. Der Decorator erkennt dabei den Datentyp *InjectedTestClass* und verfährt wie im vorherigen Fall.

Und last but not least: Interfaces. Interfaces können leider nicht nach JS transpiliert werden, da es in JS keine Interfaces gibt. Daher behelfen wir uns mit einem TypeScript-Trick: Wir definieren Interfaces mit dem *class*-Schlüsselwort, verwenden aber dennoch *implements* für Implementierungsklassen (Listing 4).

Und nun die schmutzigen Details: der Parameter-Decorator

Endlich kommen wir in Listing 6 zu den Details des *@inject()*-Decorators. Er überprüft in der Set-up-Phase die übergebenen Parameter.

Falls es sich um eine Klasse handelt, wird – ähnlich wie beim Registrieren in der DI Registry – ein Hash aus der Klasse erzeugt: *(generateInjectorKey())*. Im Fall eines Symbols wird dieses in einen String umgewandelt. Die Ausführungsphase – also der Aufruf der vom Decorator retournierten Funktion – verwendet drei Schlüsselelemente:

- Parameter *target*: die Klasse der Methode des Parameters, in unserem Fall *TestClass*
- Parameter *paramInx*: der Parameterindex
- Den Parametertyp: Er wird von der Metadatenfunktion *Reflect.getMetadata()* geliefert.

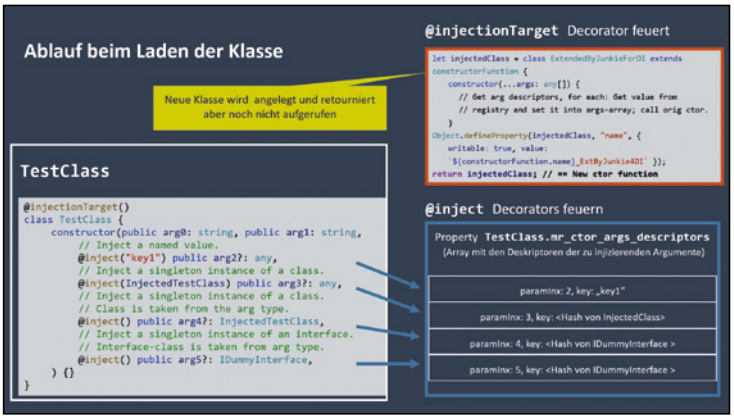


Abb. 2: Detaillierter Ablauf beim Laden der Klasse

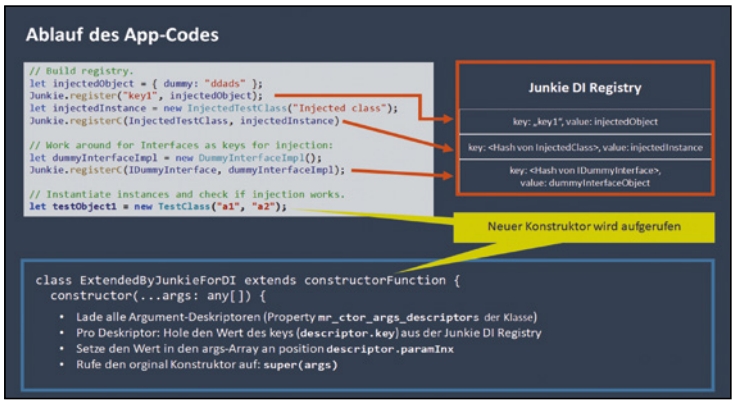


Abb. 3: Detaillierter Ablauf beim Instanziieren der Klasse

Falls beim *@inject()* ein Key mitangegeben wird, wird dieser verwendet. Wenn der Parametertyp eine ES6-Klasse ist, wird ein Hash Key generiert.

Schlussendlich werden die Informationen durch *pushInjectedCtorArgDescriptor()* in der Klasse gespeichert. Und Sie ahnen es: Genau diese Informationen ermöglichen es dem neuen *constructor*, die richtigen Parameter zu injizieren. (Hinweis: Der *readonly* Member *Junkie.classCtorInjectedArgsDescriptors* in Listing 5 enthält den String *mr_injector_ctor_args*).

Weitere Details: der Klassen-Decorator

Als letztem Codedetail widmen wir uns dem Klassen-Decorator aus Listing 6.

Hier gibt es wieder einen wichtigen TypeScript-Trick: Wenn der Klassen-Decorator eine Klasse zurückliefert, wird diese zum Konstruieren neuer Objekte verwendet. Das ist genau das, was wir brauchen. Aber schrittweise:

- Wir erzeugen eine neue Klasse, die von der ursprünglichen erbt.
- Der *constructor* dieser Klasse wird aufgerufen, wenn eine Instanz der dekorierten Klasse erzeugt wird, z. B. durch *new TestClass()*.
- Die Parameter werden als *var-args*-Array an den *constructor* übergeben.
- Der *constructor* holt zuerst die Deskriptoren der zu injizierenden Argumente.

- Pro Deskriptor wird der zu *descriptor.key* passende Wert aus der DI Registry gelesen.
- Dieser Wert wird an den im Deskriptor angegebenen Parameterindex des *varg-args*-Arrays gesetzt. Das ist die eigentliche Injection.
- Zu guter Letzt wird noch der Basis-/Super-Constructor aufgerufen.

Bevor die neue Klasse retourniert wird, setzen wir eine Property *name*, die den ursprünglichen Klassennamen mit dem Suffix *ExtByJunkie4DI* enthält. Damit wird sichergestellt, dass man beim Debuggen leicht erkennen kann, um welche Klasse es sich wirklich handelt.

All together now

In den **Abbildungen 2** und **3** sind noch einmal die detaillierten Abläufe beim Laden und beim Instanziieren der Klasse inklusive Code-Snippets zusammengefasst.

Fazit

TypeScript ist cool! Mit einer einfachen Registry und TypeScript Decorators lassen sich Klassen so manipulieren, dass man beim Instanziieren neue Werte in die Parameter ihrer Konstruktoren injizieren kann. Diese simple Dependency Injection genügt für viele Anwendungsfälle und lässt sich natürlich noch ausbauen.

Thomas Mahringer ist in der Lösungsberatung und im Lösungsvertrieb tätig. Als gelernter Informatiker hat er im Lauf seiner zweiundzwanzigjährigen Berufslaufbahn bei verschiedenen – auch internationalen – Projekten Software entwickelt, Kunden beraten, Projekte und Teams geleitet, Application Performance analysiert und optimiert, Presales- und Salesstrukturen aufgebaut sowie Partnermanagement- und Wissenstransferkonzepte erarbeitet. Aus dieser Erfahrung ergibt sich seine Leidenschaft, Kunden bei der effizienten Einführung von zeitgemäßen und wertschöpfenden Lösungen zu unterstützen.

Listing 6: Klassen-Decorator

```
/**
 * Class decorator for classes you need to inject into.
 * @returns a ctor function that will wrap the original ctor.
 *   The new ctor checks for all properties marked by {@link inject}
 *   and looks up their value from the registry.
 */
export function injectionTarget() {
  return function <T extends IClass<any>>(constructorFunction: T) {

    // Generate nuew a ctor function.
    let injectedClass = class ExtendedByJunkieForDI
      extends constructorFunction {

      constructor(...args: any[]) {
        // Get the arg descriptors from the
        // class object (== constructorFunction)
        let ctorArgDescriptors =
          Junkie.getInjectedCtorArgsDescriptors(constructorFunction);
        if (ctorArgDescriptors) {
          // Augment the arguments array if needed.
          for (let descriptor of ctorArgDescriptors) {
            if (args.length - 1 < descriptor.paramInx) {
              for (let i = 0;
                i < descriptor.paramInx - args.length; i++) {
                args.push(undefined);
              }
            }
            let value2Inject = Junkie.get(descriptor.key);
            // Only inject parameter if not yet set!
            // => We can unit test class as usual.
            if (args[descriptor.paramInx] === undefined) {
              args[descriptor.paramInx] = value2Inject;
            }
          }
        }
        // Call original ctor.
        super(...args);
      }
    };
    // Save the original ctor function for debugging purposes.
    (<any>injectedClass).origCtorFunction = constructorFunction;

    // NOTE: "function.name" is readonly property.
    // If we change it, the JS VM throws an error.
    //     => Use "Object.defineProperty"
    // If we do not change the name,
    // all classes will have the same name "ExtendedByJunkieForDI".
    //     => Debugging nightmare, especially in reactjs,
    // because all components have the same name!
    Object.defineProperty(injectedClass,
      "name",
      { writable: true,
        value: `${constructorFunction.name}_ExtByJunkie4DI` }
    );
    return injectedClass;
  }
}
```

Wenn's etwas weniger sein darf: React und Redux kombiniert

Olis bunte Welt der IT

von Oliver Sturm

Redux, Reduktion, reduzieren, kleiner, weniger ... aber was? Arbeit? Das wäre gut. Funktionalität? Nicht so gut. Weniger ist oft mehr, das ist klar! Mal sehen, was Redux so kann ...

Der Name Redux hat mit „weniger" und „kleiner" nur bedingt zu tun, denn er lehnt sich an die Funktion *reduce* an, die im Umfeld funktionaler Programmierung schon seit Jahrzehnten der Aggregierung von Daten dient. Dabei geht es durchaus um Reduktion, aber im Sinne des französischen „Jus", das Sie womöglich aus der Küche kennen: Flüssigkeiten werden konzentriert und im Volumen reduziert, während die leckeren Inhaltsstoffe erhalten bleiben. Redux arbeitet mit Reducern, also Funktionen, die der Reduktion dienen. Ach so. Na dann.

Im Kern geht es Zustandsverwaltung. Wir wissen, dass diese nicht einfach ist. Irgendwann war die mal einfach, als man sich für globale Variablen bei der Programmierung mit BASIC oder gar C noch nicht so sehr schämte wie heute. Aber die Erinnerung täuscht selbst hier gern: Letztlich war hauptsächlich deshalb alles einfacher, weil man im eingeschränkten Speicher der damaligen Computer sowieso nicht Programme derselben Komplexität bauen konnte, wie sie heute überall geschrieben werden. Sobald in umfangreichen Anwendungen *State*, also Zustandsinformation, verwaltet und zwischen verschiedenen Bestandteilen und Modulen koordiniert werden muss, wird die Sache ziemlich schwierig.

Zustand macht alles schwieriger

Seit vielen Jahren schreibe und rede ich selbst gern über die Vorzüge der funktionalen Idee beim Umgang mit *State*. Vielleicht haben Sie einmal eine meiner Präsentationen gesehen, in denen es um unveränderbare Daten ging. Solche Konzepte waren in C# umsetzbar, aber verursachten immer etwas zusätzlichen Aufwand beim Umgang mit Daten, weil in der statisch typisierten Umgebung von .NET nicht einfach ein vollwertiger Ersatz für Standardtypen erzeugt werden kann, der die Veränderung von Daten unmöglich macht.

In JavaScript ist der Umgang mit unveränderbaren Daten heute recht einfach. Ich benutze selbst gern das Paket seamless-immutable. Damit kann ich etwa ein unveränderbares Objekt so erzeugen:

```
const person = Immutable({ name: 'Oli', age: 23 });
```

Das geht auch mit Arrays und beliebig komplexen Daten, und seamless-immutable sorgt nun dafür, dass Properties und Array-Inhalte sich später nicht in-place ändern können.

Bei der praktischen Anwendung solcher Datentypen stellen sich nun einige Fragen. Eine der wichtigsten ist, wie mit der Tatsache umgegangen werden soll, dass sich in der wirklichen Welt manchmal Informationen ändern. Die Antwort ist im Prinzip einfach: Wenn es Änderungen gibt, werden neue Objekte erstellt, statt die vorhandenen zu ändern. Auch das kann seamless-immutable (Listing 1).

Wunderbar. Was bleibt? Es bleibt ein großes Problem. Wenn ich davon ausgehe, dass in einer durchschnittlichen Anwendung viele Objekte und Listen von Objekten gehalten werden, dann müssen alle Veränderungen, die irgendwann auftreten, koordiniert werden. Das ist wichtig, da offensichtlich jede Veränderung an einem Objekt, das selbst Bestandteil eines anderen Objekts (oder einer Liste) ist, zur Veränderung dieses übergeordneten Objekts führt.

Änderungen sind ansteckend

Dies ist das Problem, das Redux löst: die Verwaltung eines zentralen Zustandscontainers und die Verarbeitung von Änderungen an einzelnen Blöcken von *State*.

Die Beispiele, die ich im Folgenden für Redux zeigen werde, basieren auf der Verwendung mit einfachem JavaScript und React. Natürlich können Sie Redux auch mit anderen Plattformen einsetzen – ich habe selbst erfolgreich eine Konsolenanwendung damit gebaut, ganz ohne grafische Oberfläche. Wenn Sie gern, einfach so, viel mehr Code schreiben als nötig, können Sie auch Angular oder gar TypeScript verwenden. Die Grundlagen von Redux ändern sich durch diese Entscheidungen natürlich nicht.

Für Redux beginnt alles mit der Erzeugung von Actions. Das sind kleine Objekte, die Aktionen repräsentieren, mit all den Details, die für eine Aktion jeweils relevant sind. Meist beginnt man mit der Definition der Aktionstypen als Strings:

```
const ADD_TODO = 'ADD_TODO';
const CHANGE_DONE = 'CHANGE_DONE';
```

Aktionstypen müssen in der Anwendung eindeutig sein, eventuell sollten Sie also etwas komplexere Bezeichner verwenden. Im nächsten Schritt definieren Sie sich des Komforts halber ein paar Hilfsfunktionen, die zur Erzeugung der Action-Objekte dienen und sicherstellen, dass diese immer dieselbe Struktur haben.

```
const addTodo = text => ({ type: ADD_TODO, payload: { text } });
const changeDone = (id, done) => ({ type: CHANGE_DONE, payload: { id,
done } });
```

An dieser Stelle können TypeScript-Anhänger und andere Fans der objektorientierten Programmierung natürlich auch gern Klassen mit Konstruktoren verwenden. Die gezeigte Struktur ist übrigens eine Best Practice, kann aber beliebig an eigene Vorlieben angepasst werden.

An dieser Stelle kommt nun der Reducer ins Spiel. In der Anwendung werden letztlich Aktionen verschickt – der englische Begriff ist „dispatch" – also an Redux übergeben. Zum Beispiel könnte ein Klick des Benutzers auf einen Button die Aktion *ADD_TODO* auslösen, also die Erzeugung eines neuen Elements. Wenn Redux eine Aktion empfängt, von der UI oder aus einer anderen Quelle, ruft es alle bekannten Reducer auf, jeweils unter Übergabe des vorherigen bzw. aktuellen *State* und

Listing 1

```
const people = Immutable([{ name: 'Oli', age: 23}, { name: 'Anna',
                                                    age: 32}]);
// people ist nun:
// [ { name: 'Oli', age: 23 }, { name: 'Anna', age: 32 } ]

const newPeople = people.setIn([0, 'age'], 24);
// newPeople ist nun:
// [ { name: 'Oli', age: 24 }, { name: 'Anna', age: 32 } ]
```

Listing 2

```
const todoReducer = (state = Immutable([]), action) => {
  switch (action.type) {
    case ADD_TODO:
      return state.concat({ done: false, text: action.payload.text });
    case CHANGE_DONE:
      return state.setIn([action.payload.id, 'done'], action.payload.done);
    default:
      return state;
  }
};
```

der Action selbst. Ein Reducer für das Beispiel könnte etwa so aussehen wie in Listing 2.

Sie können der Funktionsdeklaration entnehmen, dass der *State* ein (unveränderbares) Array sein soll – dazu gleich noch mehr. Der Reducer hat die Aufgabe, anhand der Action und des übergebenen *State* einen neuen *State* zu berechnen. Wie er das macht, ist dem Programmierer überlassen. Im Beispiel wird die Funktion *setIn* verwendet, wie schon vorher demonstriert, bzw. *concat* für den Vorgang, ein neues Element an das Array anzuhängen.

Reducer erzeugen immer neue „State"-Objekte

Wichtig ist, dass analog zur erwähnten Logik der unveränderbaren Daten jeweils ein neues Objekt als Resultat berechnet wird. Es wird nichts am vorhandenen *State* geändert! Natürlich könnten Sie theoretisch denselben Code auch ohne *Immutable* schreiben, und dann ist eine Änderung des *State* technisch möglich. Damit würden Sie allerdings nur erreichen, dass später aufgerufene Reducer mit einer falschen Ausgangssituation konfrontiert würden – das gilt es zu vermeiden!

Behalten Sie ebenfalls im Kopf, dass für jede Aktion, die von Redux verarbeitet wird, alle Reducer aufgerufen werden. Wenn ein Reducer zu einer bestimmten Aktion nichts beizutragen hat, gibt er einfach den vorhandenen *State* unverändert zurück. Somit haben große Zahlen von Reducern im System nur wenig Overhead.

Gewöhnlich schreibe ich meine Reducer noch ein wenig anders als im Beispiel:

```
const createTodoReducer = initialState => (state = initialState, action) => {
  ...
}
```

Ich erzeuge anstelle des Reducers selbst zunächst eine Funktion, der ein Ausgangszustand übergeben werden kann, bevor sie den eigentlichen Reducer zurückliefert. Das finde ich praktisch, da ich somit die Ausgangszustände aller Reducer im System an zentraler Stelle definieren kann, statt sie in den Implementationen der Reducer zu verstecken.

Zur Initialisierung von Redux in einer React-Anwendung sind nun noch ein paar Schritte erforderlich, die direkt nach dem Start ausgeführt werden. Zunächst erzeugen Sie Ihre Reducer:

```
const todoReducer = createTodoReducer(
  Immutable([
    { done: true, text: "Read Oli's Redux article" },
    { done: false, text: 'Play with Redux yourself' }
  ])
);
```

Wenn Sie tatsächlich nur einen Reducer haben, ist der folgende Schritt eigentlich nicht notwendig. Ich empfehle ihn aber trotzdem, denn in einer echten Anwendung haben Sie natürlich schnell mehr als nur einen Reducer. Der Schritt besteht darin, aus den verschiedenen Reducern im System einen einzigen zu machen, bzw. die States aller Reducer in einem Objekt zu kombinieren.

```
const reducers = combineReducers({
  todos: todoReducer
});
```

Die Funktion *combineReducers* stammt aus dem Paket redux. In diesem Aufruf wird festgelegt, dass das Unterobjekt *todos* der Teil der Zustandsdaten ist, um den sich der *todoReducer* kümmert. Redux sorgt selbst dafür, dass beim Aufruf dieses Reducers nur der angegebene Teil als vorheriger *State* übergeben wird. Somit muss nicht jeder einzelne Reducer wissen, was sonst noch für Teile in der App existieren.

Soweit es Redux betrifft, gibt es nur noch einen weiteren Schritt: die Erzeugung des *Store*-Objekts. Oft beginnen Beschreibungen von Redux mit diesem Objekt, denn es ist zentral im System. Der Store hält die Zustandsdaten und ist für die Verarbeitung von Aktionen zuständig. Im Beispiel geht das ganz einfach mit der Funktion *createStore*, ebenfalls aus dem Paket redux:

```
const store = createStore(reducers);
```

Oft sieht man eine etwas kompliziertere Form des Aufrufs, etwa so:

```
const store = createStore(
  reducers,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
    window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Diese Variante sorgt dafür, dass Redux mit den Redux Dev Tools kommuniziert, die man beispielsweise

in Chrome installieren kann. Das ist sehr zu empfehlen, denn diese Tools sind sehr mächtig und können in der Aktionssequenz der laufenden Anwendung beliebig navigieren, etwa vergangene Zustände wiederherstellen und einzelne Aktionen im Detail anzeigen.

Nun fehlen noch ein paar kleine Schritte, die zur Anbindung von React notwendig sind. Aus dem Paket react-redux holen Sie sich den *Provider*, der so eingesetzt wird:

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Sie sehen, dass der *Provider* eine Art Wrapper für das Rendering der gesamten Anwendung darstellt. Dadurch können sich einzelne Komponenten an beliebiger Stelle der Hierarchie mit dem *Store* verbinden. Die Komponente in Listing 3 könnte etwa verwendet werden, um die Elemente der *Todo*-Liste in einer verkürzten Form auszugeben.

Die Komponente *TodoList* erhält zwei Parameter aus ihren *Props*: die Liste von *Todo*-Elementen und einen Event Handler namens *newItem*. Beide Parameter werden bei der Ausgabe verwendet. Aus der Liste der Elemente wird eine Sequenz von *<li>*-Tags generiert, und der Event Handler wird an einen Button angebunden. Aber woher stammen diese Parameter? Die Antwort ergibt sich aus dem Code in Listing 4, der auf die Deklaration der Komponente folgt.

Zur Verbindung der Komponente mit Redux werden zwei Funktionen benötigt. *stateToProps* bekommt das kombinierte *State*-Objekt übergeben und extrahiert den Teil, für den sich die Komponente interessiert. *dispatchToProps* empfängt eine Referenz auf die *dispatch*-Funktion des *Store*, und erzeugt den Event Handler *newItem*, in dem *dispatch* mit einer Aktion aufgerufen wird. Sie erinnern sich an *addTodo*? Diese Hilfsfunktion, mit der eine Instanz der Aktion erzeugt wird, habe ich oben beschrieben.

Listing 3

```
const TodoList = ({ todos, newItem }) => (
  <div>
    <ul>
      {todos.map((item, index) => (
        <li key={index}>{item.text}</li>
      ))}
    </ul>
    <button onClick={newItem}>New Item</button>
  </div>
);
```

Listing 4

```
const stateToProps = state => ({ todos: state.todos });
const dispatchToProps = dispatch => ({
  newItem: () => dispatch(addTodo('New Todo'))
});
const ConnectedTodoList = connect(
  stateToProps,
  dispatchToProps
)(React.memo(TodoList));
```

Fazit

Damit ist das Ende erreicht! Wir haben jetzt eine Anwendung, in der die React Views als Funktion des Zustands berechnet werden, so ganz im mathematischen Sinne. Redux sorgt dafür, dass dieser Zustand zentral gehalten werden kann, modifiziert einzig auf dem Weg über Aktionen und Reducer. Alle Daten im System sind unveränderbar, was funktional sauber ist und außerdem sehr effizient für Updates der UI, denn React braucht lediglich Objektreferenzen zu vergleichen, um festzustellen, ob sich etwas am Zustand geändert hat.

Wenn Sie Interesse an Redux gefunden haben, empfehle ich Ihnen, diesem Link zu folgen: https://osturm.me/oliswelt-redux. Dort können Sie direkt online mit einer Demoanwendung experimentieren, in der Sie die Elemente aus diesem Artikel wiederfinden.

In „Olis bunte Welt der IT“ kommentiert **Oliver Sturm** klar und direkt Entwicklungen, Behauptungen, Tatsachen und Trends der IT. Oliver Sturm ist Training Director bei DevExpress. In über fünfundzwanzig Jahren hat er weitreichende Erfahrungen als Softwareentwickler und -architekt, Berater, Trainer, Sprecher und Autor gesammelt.

http://oliversturm.com

Cosmos DB in eigenen Projekten einsetzen: Grundlagen und Einsatzszenarien

Ganz ir-relational

Dokumentendatenbanken wie Microsofts Cloud-basierte Azure Cosmos DB bieten neue Ansätze und Möglichkeiten, um mit Daten umzugehen – zumindest neu für die Microsoft-Entwicklerwelt, die von relationalen Datenbanken dominiert wurde.

von Thorsten Kansy

Dieser Artikel gibt Ihnen einen kleinen Überblick über die Grundlagen, Möglichkeiten und unterstützten APIs, die Azure Cosmo DB für Ihre Projekte bereitstellt, und bietet Überlegungen zu möglichen Einsatzszenarien an.

Einführung

Microsofts Azure Cosmos DB ist eine Dokumentendatenbank, ein NoSQL JSON Data Storage, das neben einer ganzen Reihe interessanter Features mehrere APIs bietet, um mit diesen zu arbeiten. „NoSQL“ steht dabei übrigens für „Not only SQL“, was einen Hinweis auf die unterschiedlichen APIs darstellt.

Dabei ist der Ansatz, Daten zu speichern, ein völlig anderer als z. B. bei relationalen Datenbanken. So werden Daten nicht in Tabellen abgelegt, sondern in Collections (auch Container genannt). Eine Dokumentendatenbank speichert Daten in Form von Dokumenten im JSON-Format. Dies erlaubt, diese Daten verschachtelt und – und das ist wichtig – heterogen abzuspeichern. Das heißt, Dokumente, die sich einen Container (Collection) teilen, müssen nicht über den gleichen Aufbau verfügen. Es können also mal mehr, mal weniger oder schlicht unterschiedliche Informationen in einem solchen Dokument untergebracht werden. Auch können von Dokument zu Dokument für gleichnamige Eigenschaften unterschiedliche Datentypen verwendet werden. Listing 1 zeigt ein kleines JSON-Dokument als Beispiel. Bei Abfragen wird dies (je nach API) entsprechend berücksichtigt.

Für den Zugriff auf diese Daten stehen aktuell fünf unterschiedliche APIs zur Verfügung (mehr dazu später). Existieren also schon Code für und Erfahrung mit z. B. MongoDB oder Apache Cassandra, steht einem Einsatz von Azure Comos DB eigentlich nichts im Weg. In **Abbildung 1** sehen Sie eine Übersicht aller APIs.

Die unterschiedlichen APIs können dabei prinzipiell alle Dokumente nutzen, die mit Hilfe eines anderen APIs erstellt wurden. In der Praxis gibt es jedoch deutliche Einschränkungen, da z. B. mit dem Gremlin API auch Graphdaten erzeugt werden, deren Strukturen von anderen APIs nicht verstanden werden.

Features

Welches sind nun die erwähnten interessanten Features? Als Cloud Service ist Azure Cosmos DB „fully

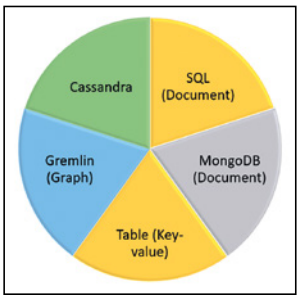


Abb. 1: Die unterstützten APIs

Abb. 2: Globale Verteilung

managed“, sodass keinerlei administrative Aufgaben anfallen. Mit einem Service Level Agreement (SLA) [4] garantiert Microsoft eine hohe Verfügbarkeit und schnelle Antwortzeiten mit der Option, bei Bedarf zu skalieren. Eine automatische und für eine Anwendung völlig transparente Verschlüsselung sorgt für die notwendige Sicherheit der Daten.

Globale Verteilung („globally distributed“) erlaubt die Festlegung, in welchem Rechenzentrum die Daten gespeichert werden. Dabei ist vorgesehen, dass dies mehr als nur ein Rechenzentrum ist. Die Infrastruktur im Hintergrund sorgt für die notwendige Replikation. Der Vorteil der globalen Verteilung liegt dabei auf der Hand: Beim Zugriff von Orten weltweit wird das nächstgelegene Replikat verwendet. Anwender z. B. aus Australien greifen damit auf eines der beiden Rechenzentren in Down Under zu (wenn dort ein Replikat konfiguriert wurde). Das betrifft übrigens nicht nur Lese- sondern auch Schreibzugriffe. **Abbildung 2** zeigt mit Hexagonen die möglichen Standorte weltweit.

ACID-Transaktionen (Atomicity, Consistency, Isolation, Durability) sorgen dafür, dass Änderungen an mehr als nur einem Dokument gemeinschaftlich nach dem Alles-oder-nichts-Prinzip durchgeführt werden. So arbeiten z. B. auch Prozeduren und Trigger im Hintergrund mit Transaktionen, um eine Datenkonsistenz zu gewährleisten.

Um Zugriffe auf Dokumente möglichst schnell auszuführen, verwendet Azure Comos DB Indizes, wie andere Datenbanken auch. Diese werden automatisch

Listing 1

```
{
  "BusinessEntityID": 1,
  "PersonType": "EM",
  "NameStyle": false,
  "Title": null,
  "FirstName": "Ken",
  "MiddleName": "J",
  "LastName": "Sánchez",
  "Suffix": null,
  "EmailPromotion": 0,
  "AdditionalContactInfo": null,
  "Demographics": "<IndividualSurvey xmlns=\"http://schemas.microsoft.
com/ sqlserver/2004/07/adventure-works/IndividualSurvey\"><TotalPurch
aseYTD>0</                    TotalPurchaseYTD></IndividualSurvey>",
  "rowguid": "92c4279f-1207-48a3-8448-4636514eb7e2",
  "ModifiedDate": "2009-01-07T00:00:00",
  "Password": {
    "Hash": "pbFwXWE99vobT6g+vPWFy93NtUU/orrIWafF01hccfM=",
    "ModifiedDate": "2009-01-07T00:00:00",
    "Salt": "bE3XiWw="
  }
}
```

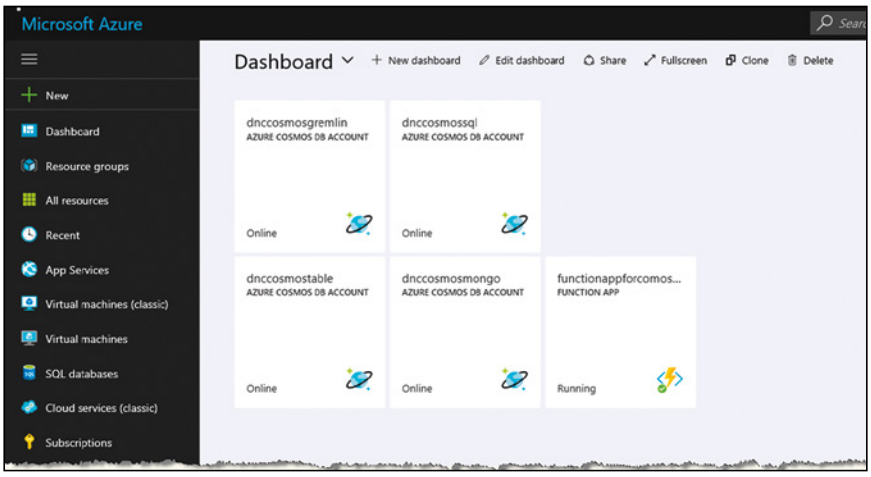


Abb. 3: Azure Portal

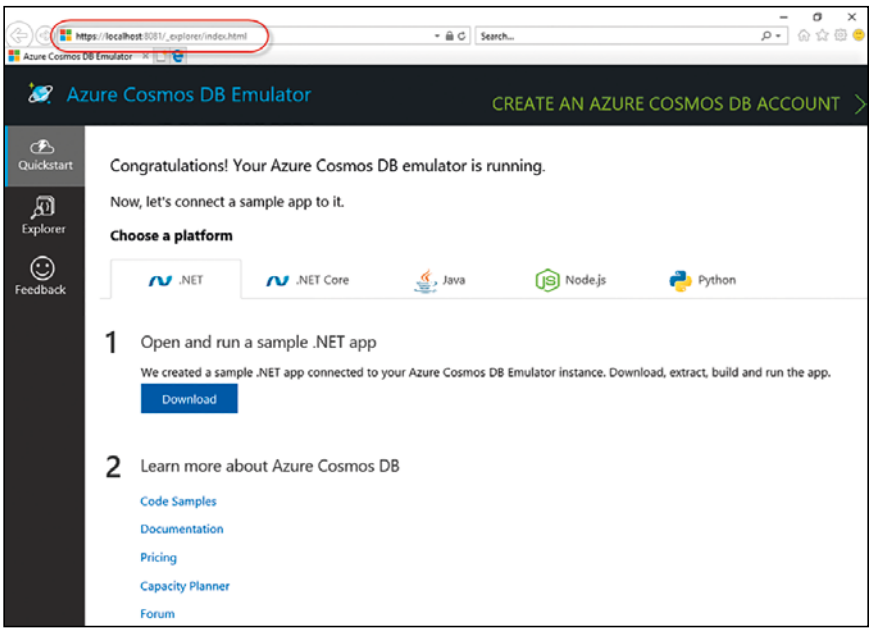


Abb. 4: Azure-Cosmos-DB-Emulator

erstellt und gepflegt. Dazu werden Abfragen analysiert, und es wird ermittelt, welche Eigenschaften wie oft zum Filtern verwendet werden. Diese automatische Indexierung kann durch Policies beeinflusst werden, funktioniert aber durchaus auch ohne weiteres Zutun.

Quelle	Ziel
JSON-Datei	JSON-Datei
DocumentDB - SQL API - Table API	DocumentDB - SQL API - Table API
SQL-Server	
CSV	
AzureTable	
DynamoDB	
HBase	

Tabelle 1: Quellen und Ziele des Azure Cosmos DB Migration Tools

Tools

Für die Entwicklung mit Cosmos DB existieren schon einige praktische Tools. Als Erstes wäre natürlich das schon erwähnte Azure Portal [1] zu nennen, über das die Datenbanken angelegt und verwaltet werden. Außerdem gibt es über das Portal die Möglichkeit, Daten abzufragen und zu modifizieren (**Abb. 3**).

Wer bei der Entwicklung auf keine (verlässliche) Verbindung zugreifen kann oder schlicht mögliche Kosten vermeiden möchte, dem steht der Azure-Cosmos-DB-Emulator [2] zur Verfügung (**Abb. 4**). Dieser Emulator, der natürlich ohne Azure Cloud auskommt, kann wahlweise installiert oder als Docker Image genutzt werden.

Zugriffsschlüssel und Verbindungszeichenfolgen sind (standardmäßig) für alle Installationen und Docker Images gleich, sodass bei einer Entwicklung im Team kein Sicherheitsproblem auftritt und Entwickler immer wieder Anpassungen vornehmen müssten, um auf die Daten im Emulator zuzugreifen.

Und als Letztes sei da noch das Azure Cosmos DB Migration Tool [3] genannt, mit dem Daten von Quellen in ein Ziel geschrieben werden können. Dabei steht eine Reihe von Formaten für die eine und auch die andere Seite zur Auswahl (Tabelle 1).

Da einige Quellen wie z. B. relationale Datenbanken wie SQL Server keine verschachtelten Daten unterstützen, gibt es hier die Option, in der Abfrage Joins zu verwenden und in den Namen der zurückgegebenen Spalten einen „Nesting Seperator“ (z. B. einen Punkt) einzubauen, der für die gewünschte Verschachtelung im Ziel sorgt.

Als Nebeneffekt lassen sich mit diesem Tool z. B. Daten aus einer SQL-Server-Abfrage in eine JSON-Datei schreiben, was sicherlich das eine oder andere Mal ganz nützlich sein kann.

Sicherheit

Selbstverständlich ist, gerade bei einer Cloud-basierten Technologie, das Thema Sicherheit wichtig. Neben der Sicherheit während des Transports über das Netzwerk mittels SSL/TLS und der Absicherung durch die Anbindung an ein virtuelles Netzwerk bietet Azure Cosmos DB eine Sicherheit, die zwischen nur lesenden und Lese-und-Schreib-Zugriffen unterscheidet (von administrativen Aufgaben einmal abgesehen). Gesteuert wird dies über entsprechende Schlüssel.

Eine differenzierte Kontrolle auf Ebene eines Containers (Collection) oder einzelner Elemente (Dokumente)

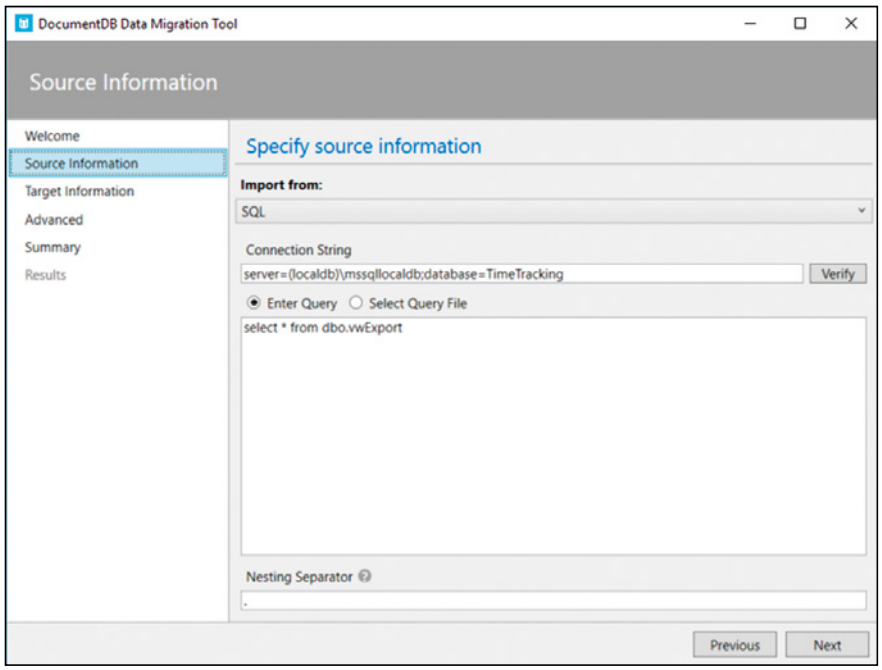


Abb. 5: Azure Cosmos DB Migration Tool

ist nicht vorgesehen. Dafür muss die Anwendung bei Bedarf selbst sorgen.

APIs

Für Anwendungen stehen fünf APIs zur Auswahl, um auf Azure Cosmos DB zuzugreifen. Ihr Vorteil: Wenn Sie bereits Erfahrung mit einem dieser APIs haben oder sogar bereits fertiger Code existiert, kann dieser mit wenigen (oder sogar ohne) Anpassungen verwendet werden. Beispielsweise verwendet der Zugriff via MongoDB API die gleichen Bibliotheken, die auch zum Einsatz kommen, um auf eine „gewöhnliche" MongoDB-Installation [6] zuzugreifen.

Angeboten wird also das MongoDB API und damit die Schnittstelle zu einer klassischen Open-Source-Dokumentendatenbank. Mit dem API für Gremlin steht eine Graph-Datenbank zur Verfügung. Eine Graph-Datenbank speichert Informationen in Knotenpunkten und deren Beziehungen untereinander. Damit lassen sich Beziehungsgeflechte in sozialen Netzwerken, Nutzungsprofile und Ähnliches abbilden und abfragen. Das Core (SQL) API bietet die Möglichkeit, Informationen mit einer (einfachen) SQL-Syntax abzurufen. Das Anlegen, Löschen und Verändern von Dokumenten geschieht hingegen via der üblichen HTTP-Verben POST, PUT und DELETE, während die Abfrage selbst ein GET verwendet. Mit dem Table API steht ein Schlüssel-Wert-Speicher (Key Value Store) à la Azure Table zur Verfügung. Bei dem Wert kann es sich um komplexe Objekte handeln, die als JSON serialisiert abgelegt werden. Ein Zugriff ist mittels der Schlüssel möglich. Und schließlich wird seit einiger Zeit als neuester Zuwachs das Cassandra API [7] verwendet, das von vielen großen Unternehmen wie GitHub, Ebay, Netflix und auch im CERN eingesetzt wird.

Der Einsatz von Entity Framework Core

Ab Version 2.2 bietet Entity Framework Core das erste Mal in der Geschichte von Microsofts O/R-Mapper die Option, auf nichtrelationale Datenbanken zuzugreifen. Im NuGet-Paket *Microsoft.EntityFrameworkCore.Cosmos* befindet sich der benötigte Code. Allerdings ist der aktuelle Stand der Entwicklung noch nicht so weit, ihn tatsächlich sinnvoll einzusetzen. Dafür gibt es noch zu viele Probleme, Ungereimtheiten und nicht umgesetzte Features. Konsequenterweise ist in der stabilen Version von .NET 2.2 (vom 4. Dezember 2018) die Untersetzung für Azure Cosmos DB entfallen. Diese taucht erst in der Preview für .NET 3.0 wieder auf. Dennoch ist es ein Schritt in die richtige Richtung, und es bleibt spannend zu sehen, was EF Core 3.0 in dieser Richtung bringen wird.

Funktionen, Prozeduren und Trigger

Azure Comos DB erlaubt die Verwendung von Funktionen, Stored Procedures und Trigger, die in Serverside JavaScript geschrieben werden können. Sämtliche Elemente werden für eine Collection geschrieben; auch Funktionen und Prozeduren gehören also, anders als bei SQL Server, zu der gesamten Datenbank.

Funktionen können z. B. mit dem Core (SQL) API verwendet werden, um weitere Möglichkeiten in die

Abfragen einzubauen. Werden für eine Abfrage reguläre Ausdrücke benötigt, reicht es, eine einfache Funktion wie im Folgenden zu schreiben:

```
function RexExpTest(s, p){
  return s.match(p) != null;
}
```

Für die Entwicklung bieten sich die üblichen Tools und Wege wie online JavaScript Playgrounds [5], Visual Studio Code mit Code Runner und viele andere Optionen an. Diese neu geschaffene Funktion kann einfach verwendet werden. Achten Sie dabei auf den Präfix *udf*, der zwingend notiert werden muss. Die folgende Zeile zeigt die Core-(SQL-)Abfrage:

```
SELECT * FROM c WHERE udf.RexExpTest(c.Name, 'Speaker [0-2]') = true
```

Das abschließende Listing zeigt eine kleine Demo für eine Prozedur, die einen kleinen Text an den Client zurücksendet. Der *body* legt dabei die Funktionalität fest:

```
var helloWorldDemoProc = {
  id: "helloWorldDemo",
  body: function () {
    var context = getContext();
    var response = context.getResponse();
    response.setBody("Hello World!");
  }
}
```

Gründe für den Einsatz

Die große Frage ist nun nur noch, ob sich der Einsatz einer Dokumentendatenbank, speziell Azure Cosmos DB, im eigenen Projekt lohnt. Diese ist pauschal weder einfach noch überhaupt zu beantworten; es sei denn, der Einsatz der Azure Cloud ist ein No-Go. Wenn Sie bis dato (wie der Autor) Ihre Daten in Tabellen und relationalen Datenbanken gespeichert haben, lohnt sich ein neugieriger Blick auf die anderen Möglichkeiten. Wenn die zu verarbeitenden Daten oft in flexiblen Varianten verarbeitet werden müssen oder wenn sie über Beziehungen verfügen, die in allen (oder den meisten) Use Cases aufgelöst werden müssen, dann bietet sich die Flexibilität von JSON-Dokumenten an. Müssen die Daten immer in der gleichen Form vorliegen, kann sich diese Flexibilität als eher nachteilig erweisen – schließlich muss die Anwendung für feste Vorgaben sorgen.

Auf der anderen Seite ist jedoch zu bedenken, dass die APIs aktuell keine partiellen Updates der Dokumente unterstützen. Wenn also ein Dokument auch noch so minimal verändert wird, muss das komplette Dokument übertragen werden. Häufige kleine Änderungen können sich also, gerade bei größeren JSON-Dokumenten (>1 Kilobyte), schlecht auf die Performance auswirken.

Bei nicht so großen, flexiblen Daten oder solchen, die kaum geändert werden müssen, kann NoSQL seine Stärke ausspielen. Das gilt auch für Azure Comos DB mit seinen unterschiedlichen APIs.

Fazit

Azure Cosmos DB im Speziellen und NoSQL-Datenbanken im Allgemeinen sind auf jeden Fall einen Blick wert und bieten Möglichkeiten, die mit relationalen Datenbanken nur schwer realisiert werden können. Praktische Tools bis hin zum Emulator und fertige Beispiele machen den Einstieg leicht.

Thorsten Kansy, seit knapp dreißig Jahren professionell tätig, ist mit Herz und Seele Entwickler, Softwarearchitekt und Mentor. Full Stack Developer meets Microsoft SQL Server. Er bietet individuelle Workshops zu Azure Comos DB, SQL Server und .NET Core unter www.dotnetconsulting.eu und www.dotnetcore.eu an.

Links & Literatur

[1] https://portal.azure.com

[2] https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator

[3] https://docs.microsoft.com/en-us/azure/cosmos-db/import-data

[4] https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/

[5] https://jscomplete.com/playground

[6] https://www.mongodb.com

[7] http://cassandra.apache.org