

Content Security
Policy Version 3 S. 56

REST-Services unter
.NET mit Refit S. 66

LLMs mit Fine-Tuning
und RAG anpassen S. 84

dotnetpro

dotnetpro.de
dotnetpro
Das Magazin für Profi-Entwickler

2/25

Quantencomputing

Was steckt dahinter? Was
ist bereits heute möglich?
Welche Chancen eröffnen
sich in der Zukunft?
Ab Seite 10

DWX

'25

30. JUNI BIS 03. JULI 2025

MOVING TO MANNHEIM!

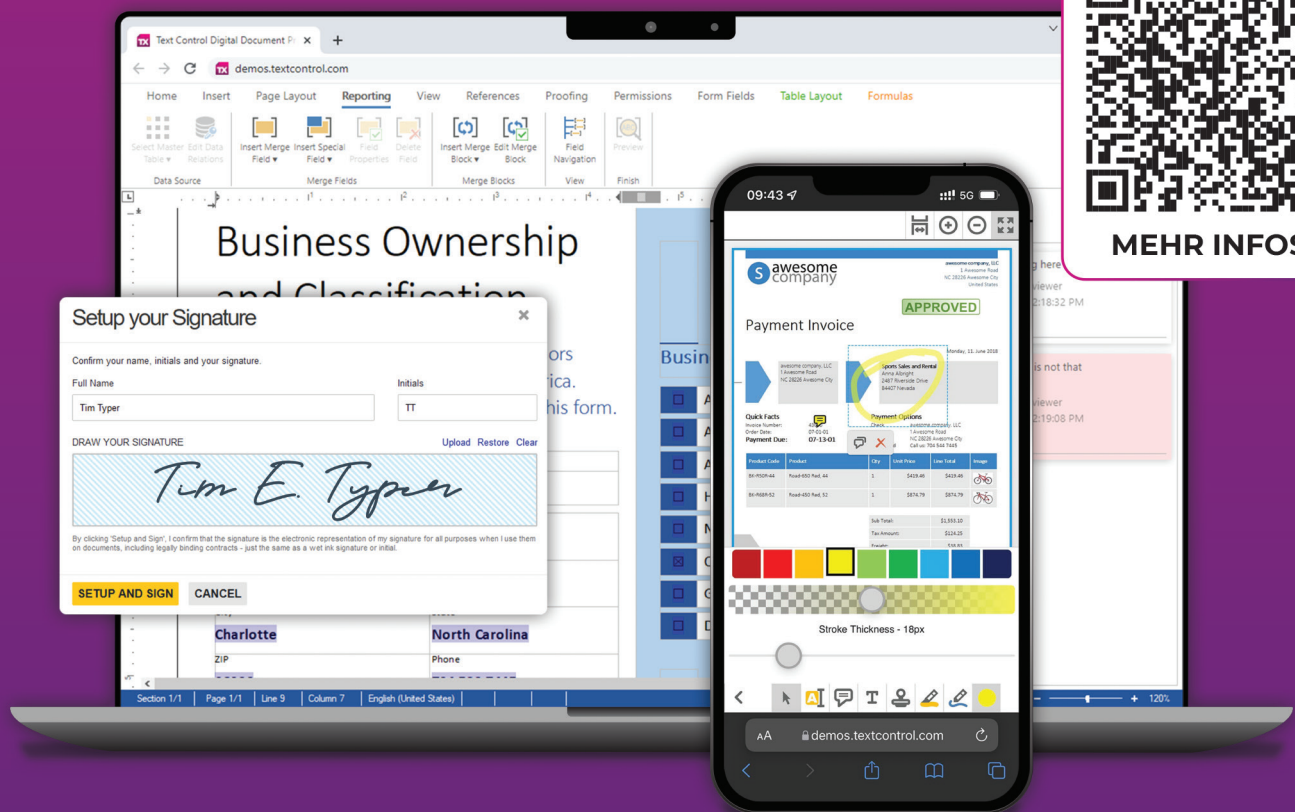
www.developer-week.de

Wir denken Dokumente neu!

Integriere die Bearbeitung, Anzeige, Erstellung und Automatisierung digitaler Dokumente in deine .NET-Anwendungen.



MEHR INFOS





Neue Welten

Wie Quantencomputing das Moore'sche Gesetz aushebeln könnte – und was das mit Künstlicher Intelligenz zu tun hat.

Erinnern Sie sich? Das Editorial der vorangegangenen dotnetpro schrieb ich unmittelbar vor der .NET Developer Conference. Damals beschäftigte mich die Rede von Satya Nadella auf der Ignite, in Erinnerung blieben mir davon vor allem die Passagen zur Künstlichen Intelligenz und der Begriff „Copilot“.

In den Gesprächen auf der DDC mit den Teilnehmerinnen und Teilnehmern, Sprecherinnen und Sprechern tauchte ein weiterer Begriff auf, den Nadella ebenfalls verwendete – Agenten. Natürlich geht es nicht um James Bond und andere Spione, sondern um Software-Agenten, die „zu eigenständigem und eigendynamischem Verhalten fähig sind“, wie es bei Wikipedia heißt.

1800 dieser KI-Agenten könne Microsoft seinen Kunden inzwischen anbieten, hieß es, und täglich würden es mehr. Es bedarf keiner besonderen Fantasie, um sich vorzustellen, mit welcher gigantischen Datenmengen LLMs fortlaufend im Hintergrund trainiert werden müssen, um die Heerschaaren leistungsfähiger Agents bereitzustellen. Und es stellt sich die Frage, welche Auswirkungen dies auf das Computing der Zukunft haben wird: Werden wir an Grenzen stoßen – beim Verarbeitungstempo, beim Energieverbrauch?

„Quantencomputing vermag komplexe Probleme zu lösen, vor denen klassisches Computing kapituliert.“

Einen spannenden Ansatz bietet hier das Quantencomputing: Indem es sich die Vorteile zunutze macht, die Quantenmechanik und Quantenalgorithmen bieten, hat es das Potenzial, die Verarbeitung riesiger Datenmengen erheblich zu beschleunigen. Um die Größenordnung zu verdeutlichen: „El Capitan“, der derzeit schnellste Supercomputer, bringt es auf 1742 Exaflops, also kaum vorstellbare 1742 Trillionen Rechenoperationen pro Sekunde. Quantencomputern wird eine Vervielfachung dieser Rechenleistung nachgesagt.

Im Schwerpunkt dieser dotnetpro-Ausgabe erfahren Sie deshalb mehr über die Möglichkeiten des Quantencomputing. Gleich mehrere Artikel behandeln wichtige Konzepte und Grundlagen und liefern Beispiele, wie sich Quantencomputing bereits heute über die Cloud auch in der .NET-Welt realisieren lässt.

Ich wünsche Ihnen eine schöne Zeit mit der neuen dotnetpro!

Herzlichst

Fernando Schneider
Chefredakteur dotnetpro



Bernhard Pichler

sucht nach Wegen, beim Setzen von Prioritäten heute schon die Bedürfnisse von morgen mitzudenken (S. 46)



David Tielke

befasst sich mit den unterschiedlichen Arten von Schulden in der Softwareentwicklung und deren Auswirkungen (S. 54)



INHALT

Update

Wissen und Motivation tanken

.NET Developer Conference 2024

Schwerpunkt

Neue Denkweisen

Quantum Computing: Ein Überblick

Wegweisende Symbiose

Quantum Computing in der Cloud

Marke Eigenbau

Software-Quantensimulatoren in C#

Q# für Einsteiger

Quantenalgorithmen mit Visual Studio Code

Planung

Time-out: Zukunft wirft keine Schatten

Prioritäten auch für die Zukunft setzen

Pragmatisch Richtung Microservices

Mit Modulithen alle Wege offenhalten

Dauids Deep Dive: Nicht nur in der Technik ...

Arten von Schulden in der Softwareentwicklung

Frontend

Dreimal hält besser

Web Security: Der Stand bei CSP 3

qo' vlvän!

Lokalisierung für ASP.NET Core

Core

NETirol: Fit mit Refit

HTTP-Client-Entwicklung

Vorsicht an der Bahnsteigkante

Risiko virtueller Funktionen

6

10

20

32

38

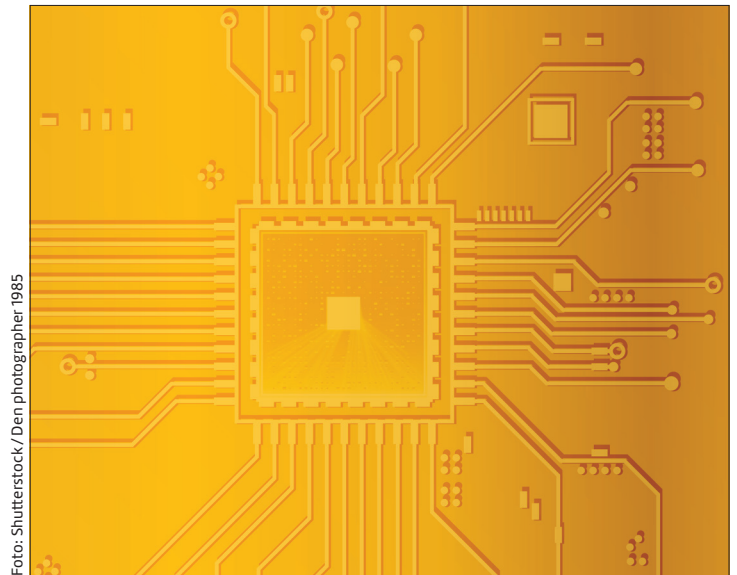


Foto: Shutterstock / Den photographer 1985

Quantum Computing: Wie Quantencomputing funktioniert, welche Rolle es spielt und was aktuell möglich ist.

10

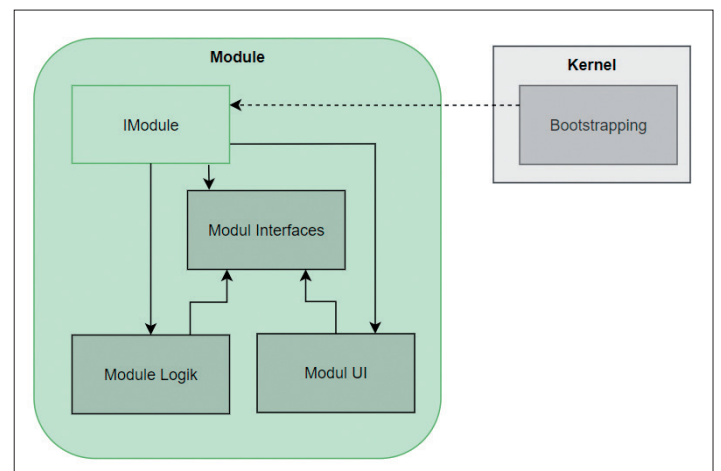
46

48

54

56

60



Mit Modulithen den Prozess so gestalten, dass Sie den Weg in Richtung Microservice oder Monolith offenhalten.

48

Experten in dieser Ausgabe



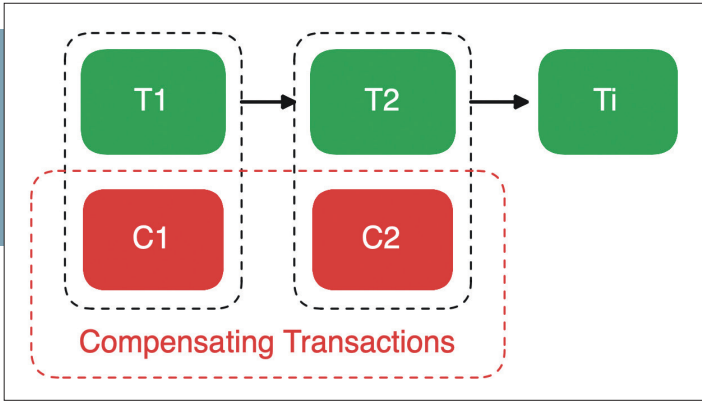
Christian Havel erläutert, wie Sie Probleme mit virtuellen Funktionen im Konstruktor umgehen.

69



Robert Mühsig beschäftigt sich mit der .NET-Anbindung von Metas Llama mithilfe von LlamaSharp.

110



Wolverine einsetzen: In Teil 8 beschäftigen wir uns mit der Mandantenfähigkeit und der ASP.NET-Core-Integration. **72**

demo-devsecops

Settings Policies Security Approvals and checks

On

Advanced Security

Advanced Security is billed based on t

[View billing](#) | [View alerts](#)

For setup tips, [view documentation](#) or

Block secrets on push

Scan all pushes to the repository

DevSecOps:

Wie Sie mit GitHub Advanced Security for Azure DevOps neuen Code pro-aktiv und auto-matisiert auf Schwachstellen überprüfen und diese effizient be-seitigen können.

98

src

pages

data_page.rs

index_page.rs

mod.rs

lib.rs

target

tests

integration_tests.rs

Cargo.lock

Cargo.toml

Programmieren mit Rust:

Im abschließen-den Teil seiner Rust-Serie zeigt Ihnen Klaus Aschenbrenner, wie Sie Fehler ab-fangen können und sich automati-sche Tests einbau-en lassen.

116

Quellcode gefällig?

Sie möchten den Quellcode eines Artikels ausprobieren? Oder den Artikel als PDF herunterladen?

Dann achten Sie auf den dnpCode im Autorenkasten am Ende des Artikels. Hängen Sie diesen Code an [www.dotnetpro.de/](#) an. Dieser Link führt Sie zur passenden Seite im dotnetpro-Archiv.

The Saga continues	
Messaging mit Wolverine und .NET, Teil 8	72
Frameworks: Kontakt zur KI	
Sprachmodelle per Ollama API ansprechen	79

KI	
Anpassen oder erweitern?	
KI und Data, Teil 4	84

Backend	
Data Access: Verschieden schnelle Wege nach Rom	
Performance unterschiedlicher Ladestrategien für verbundene Objekte bei EF Core	90
Sicherheit in jeder Zeile Code	
DevSecOps mit Azure DevOps	98

Tools	
.NET-Anbindung für die Llama-KI	
LLamaSharp	110

Workout	
Tests und Fehler	
Rust-Kurs, Teil 10	116
Heute wird gespielt!	
MonoGame für Einsteiger, Teil 8.1	128

Editorial	3
Lernstoff	108
Arbeitsmarkt	140
Impressum / Inserentenverzeichnis	144
Anbieterverzeichnis	145
Vorschau	146

.NET DEVELOPER CONFERENCE 2024

Wissen und Motivation tanken

Als letzte Konferenz des Jahres flößte die .NET Developer Conference den Teilnehmerinnen und Teilnehmern wieder jede Menge Energie ein.

Wenn die .NET Developer Conference (DDC) in Köln ihre Pforten öffnet, ist Weihnachten in greifbarer Nähe. Klar, dass zu dieser Zeit des Jahres der Akku leer ist, dass Ideen und Motivation fehlen und man sich nach den ruhigen Tagen mit Plätzchen sehnt. Softwareentwicklung? Nee, zu der Zeit nicht mehr.

Doch glücklicherweise gibt es die DDC. Deren Besucherinnen und Besucher erhalten zum Jahresende noch einmal einen richtigen Energieschub. Wissen tanken, Akku aufladen, Motivation stärken, und das Ganze im Dunstkreis von .NET: Das ist die Intention der .NET Developer Conference. Teilnehmer Kevin Faller drückt das auf LinkedIn [1] so aus: „Neben vielen neuen Eindrücken und Ideen nehme ich vor allem eines mit: Inspiration. Inspiration, die ich jetzt zurück in den Betrieb und hoffentlich auch direkt in meine Arbeit einfließen lassen kann.“

Genau so haben es die Macher der .NET Developer Conference geplant. Ein Treffen von Experten, Teilnehmenden und Partnern, um .NET zu frönen – vor allem aber, um sich auszutauschen und Anregungen zu holen.

Das ehrwürdige Hotel Pullman Cologne mit dem amerikanischen Einschlag und leckerster Verpflegung sorgte für die angenehme Wohlfühlatmosphäre der viertägigen Konferenz.

Workshops: Ab in die Tiefe

Los ging es am 25. November mit den eintägigen Workshops. Hier konnten die Teilnehmenden wählen zwischen Themen wie der perfekten .NET-Architektur, Webentwicklung mit ASP.NET, Cross-Plattform-Entwicklung mit .NET MAUI und Softwareentwicklung mithilfe von KI.

Dr. Holger Schwichtenberg führte die Teilnehmerinnen und Teilnehmer in seinem Workshop zunächst durch die Neuerungen von Blazor 9.0 und die verschiedenen Architekturmodelle des Frameworks. Ein ganzer Tag zu einem Thema bedeutet aber eben nicht nur Frontalunterricht. So sollten die Teilnehmenden auch selbst Hand anlegen und eine Blazor-9.0-Anwendung implementieren, die serverseitig vorgerendert wird, Blazor Server nutzt und dann im Hintergrund automatisch die Dateien für Blazor WebAssembly herunterlädt.

Die Entwicklung von Cloud-Native-Anwendungen war das Thema des Workshops von Loek Duys und Alex Thissen. Sie zeigten dies auf Basis von .NET Aspire. Die neue Plattform von Microsoft erleichtert das Programmieren von Cloud-basierten, verteilten Lösungen. Sie bietet dazu Funktionen wie Orchestrierung und Integration und unterstützt die Arbeit durch das passende Tooling. Auch in diesem Workshop galt: Selbst programmieren gehört dazu.

Die Zeichen standen auf Austausch: Wie geht es dir? Welche Technologie setzt du ein?





Anziehungspunkte
waren die Stände der
Partner wie TextControl,
Tom Wendels Toypad
Challenge und Xebia /
Xpirit Germany

Auch die anderen Trainer wie David Tielke, André Krämer, Christian Giesswein und Jörg Neumann sorgten mit ihrem Know-how für eine äußerst angenehme Wissensbetankung.

.NET der Kindheit und Happiness at Work

Tag zwei und drei sind auf der DDC traditionsgemäß als klassische Konferenz aufgebaut. Das heißt, dass jede Stunde ein neuer Talk stattfindet. Dieses Jahr standen vier parallele Themenstränge zur Auswahl, in denen rund 50 Sprecherinnen und Sprecher ihre Erfahrungen vermittelten. Beide Tage wurden von einer Keynote oder General Session eröffnet. Christian Giesswein nahm am ersten Tag einen Rückblick auf die Geschichte von .NET aus einem persönlichen Blickwinkel vor. Denn er ist mit der Plattform nicht nur aufgewachsen, sondern hat die Softwareentwicklung nach seinem Studium damit erst so richtig erlernt. Heute ist er immer noch begeis-

tert von den Möglichkeiten, die .NET bietet, konstatiert aber auch, dass sich .NET glücklicherweise wesentlich weiterentwickelt hat. „.NET 9 ist nicht mehr das .NET meiner Kindheit“, sagte er.

Um Begeisterung ging es im weitesten Sinne auch bei der Keynote von Kathleen Janz und Stefan Mintert am zweiten Tag. „Happiness at Work“ lautete der Titel, und es dauerte nur wenige Augenblicke, bis Stefan den Zuhörern mitteilte, dass es nicht um die Höhe des Gehalts ginge. Kathleens Beispiele ihrer persönlichen Erlebnisse machten dabei deutlich, wie sehr fehlende Zufriedenheit die Motivation der Mitarbeitenden dämpfen kann, bis hin zur Kündigung. Auf der anderen Seite steigt die Leistung von Mitarbeiterinnen und Mitarbeitern, wenn sie gerne zur Arbeit gehen. Die Faktoren, die diese Zufriedenheit beeinflussen, waren ebenso Teil der Keynote wie die Möglichkeiten, diese zu beeinflussen. ►



4



5

Bild 1 Voller Saal bei der Keynote von Christian Giesswein zum Stand von .NET.

Bild 2 Großes Angebot: Welche Session besuchen wir als Nächstes?

Bild 3 Fachsimpeln: In den Pausen kamen die Teilnehmer schnell ins Gespräch.

Bild 4 Die Toypad Challenge lockte viele Teilnehmerinnen und Teilnehmer an.

Bild 5 Was machen die Kopiloten mit mir? Diskussion auf offener Bühne mit Ina Einemann, Dr. Damir Dobric, Julia Kordick, Jörg Neumann und Tilman Börner (von rechts).

Einen gemeinsamen Schwerpunkt bei den Thementracks Software Architecture, Frontend Development, Artificial Intelligence und Data & Databases am ersten Tag und Cloud Development, Backend Technologies, Approaches und Fundamentals am zweiten Tag bildete der Blick auf die praktische Anwendung von Technologien, Tools und Vorgehensweisen. So konnten die Teilnehmenden die Entwicklung mit Entity Framework Core erlernen, sich über den Einsatz von Dapr informieren oder sich Wissen über das Refactoring aneignen.

KI allerorten

Ein wesentlicher Teil der Konferenz drehte sich selbstverständlich um die künstliche Intelligenz – sei es, um Anwendungen mit KI-Funktionen auszustatten oder die KI als Pair Programmer oder Programmierwerkzeug zu nutzen.

Jörg Neumann war der, der den KI-Reigen am Montag eröffnete. „AI-driven Software Development mit GPT und Co.“ lautete der Titel seines Workshops. Darin erklärte er den Teilnehmenden die Funktionsweise von Large Language Models (LLM) und wie sie diese in ihre eigenen Anwendungen integrieren können. Außerdem zeigte er, wie sie LLMs in Microsoft Teams einbauen können oder eine semantische Suche über Retrieval Augmented Generation (RAG) aufbauen.

Jörg diskutierte auch am zweiten Tag im Diskussionspanel mit, das im Track Artificial Intelligence stattfand. Welchen Einfluss haben Kopiloten auf das Wissen und das Lernen von Softwareentwicklern? Unter der Moderation von Tilman Börner diskutierten Jörg Neumann, Ina Einemann, Dr. Damir Dobric und Julia Kordick, ob die KI nicht zu einer Reduzierung des Wissens führt. Die klare Aussage aller Diskussionsgäste lautete: Die KI ist da. Sie wird nicht mehr weggehen. Nutze sie, sonst wirst du als altes Eisen aussortiert.

Ein Zuhörer brachte es am Ende der Diskussion auf den Punkt: „Endlich sind die Kopiloten da. Denn sonst würden wir die Masse an Klassen, APIs und Methoden irgendwann nicht mehr beherrschen können.“

DevSessions: Darf es auch ein halber Tag sein?

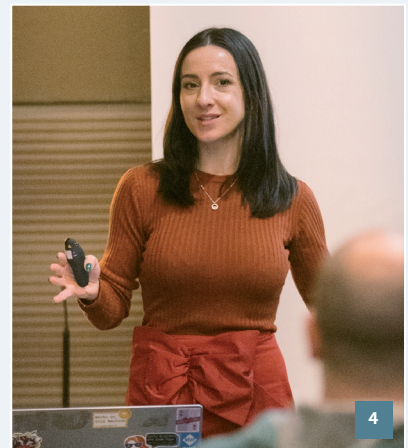
Für manches Thema genügt ein halber Tag, um es zu vermitteln. Die sogenannten DevSessions tragen dem Rechnung. So konnten die Teilnehmenden vormittags beispielsweise „Vom Monolith zum Modulith – Eine Fallstudie aus der Halbleiterindustrie“, „Mit .NET Aspire: Verteilte Anwendungen entwickeln“, „Master Class: Architecting and Development of Custom Copilots“ oder „Testautomatisierung für WebApps mit Playwright“ besuchen und nachmittags den DevSessions „Qualität von Anfang an: Der Testprozess als Erfolgsfaktor in der agilen Entwicklung“, „Part II – Deep dive into git“, „Azure Messaging – Streams, Queues und Pub Sub“ oder „Kleinteilig & großartig – Micro Frontends mit Blazor“ beiwohnen.

Aus familiären Gründen konnte die ursprünglich geplante DevSession über „Hochqualitative Softwareentwicklung“ von David Tielke nicht stattfinden. Als Ersatz sprang Miriam Hunger ein und erklärte den Teilnehmern, wie sie Qualität von Anfang an berücksichtigen können und wie risikobasiertes Testen dabei hilft, kritische Bereiche der Anwendung sicher zu machen. Verzichteten müssen die Teilnehmenden deshalb aber nicht auf die DevSession von David Tielke. Er hat sie im Nachgang auf Video aufgezeichnet. Für die DevSession Angemeldete erhalten das Video kostenfrei.

Spaß am Abend

Der Softwareentwickler lebt ja nicht nur von Wissen allein – auch der Spaß darf nicht zu kurz kommen. Mit Oliver Sturm und Jörg Neumann haben sich zwei gefunden, die man als tiefenentspannte Spaßmacher bezeichnen darf. In der Night-session „AI Ninja Moves: Wie kann ich AI in der Entwicklung konkret nutzen?“ hatten sie zuerst mit massiven technischen Problemen zu kämpfen – Spoiler: Das MacBook konnte sich mit dem Beamer nicht auf eine Kommunikation einigen –, was sie aber überhaupt nicht aus der Ruhe gebracht hat. Mit technischen Tricks konnten sie trotzdem zeigen, wie ein LLM dabei helfen kann, die Validierung für Frontend und Backend gleichermaßen zu implementieren. Der Clou dabei ist, dass

Die Experten teilen ihr Wissen





Die Faktoren für Happiness at Work vermittelten Kathleen Janz und Stefan Mintert in der zweiten Keynote (Bild 1)

der Domänenspezialist die Validierungskriterien in deutscher Sprache eingeben kann.

Wer Software entwickelt, macht Fehler. Dieser banale Fakt hat aber leider die Kraft, die Motivation in den Keller ziehen. Wie also umgehen mit den Fehlern – oder besser: wie den Fehlern etwas Gutes abringen? In der zweiten Nightsession unterfütterte Iris Classon diese Problematik mit ihren Erlebnissen in der Softwareentwicklung. Sie beantwortete die Fragen, wie man in der sich ständig ändernden Welt up to date bleibt oder eben, wie man Fehler in Erfolgsgeschichten umwandelt.

Das Salz in der Suppe

Neben hervorragenden Experten, spannenden Themen und guter leiblicher Versorgung gehört noch ein ganz wichtiger Faktor zum Erfolg einer Konferenz: die Partner, die ihre Produkte oder Dienstleistungen anbieten.

Zu den sehr treuen Partnern zählt TextControl [2], die man fast schon zum Inventar der DDC zählen könnte. Die UI-Control-Firma ist mit ihrem Stand bei den Teilnehmern nicht nur wegen ihres vielseitigen Produkts gern gesehen. Nahezu bei jedem Event verlost TextControl eine Xbox, und so freute sich auch dieses Jahr ein Teilnehmer über seinen Gewinn.

Auch Xebia/Xpirit Germany [3] war mit von der Partie. An ihrem Stand zeigte die Firma ihre Leistungspalette, von Soft-

waretechnologie über Datenlösungen bis hin zu Azure-Cloud-Anwendungen. Dabei hebt Xebia hervor, dass der DevOps-Gedanke immer mitschwingt.

An einem weiteren Stand erklärte Tom Wendel, was es mit seiner Toypad Challenge [4] auf sich hat. „Quatsch mit Software“ nennt er die Aufgabe, die er auf seiner Website stellt. Der Kernpunkt ist ein spezieller NFC-Reader von Lego, der einmal ein Computerspiel mit der realen Welt verbinden sollte. Nachdem die Nachfrage danach aber eher gering war, hat Tom die Hardware in großer Stückzahl erwerben können. Jetzt können Teams und Einzelpersonen Software für das Toypad schreiben und so mit großem Spaßfaktor die Programmierung erlernen oder neue Spielkonzepte erfinden.

Auch Developer Media Jobs (jobs.developer-media.de) war mit vor Ort und präsentierte seine Trainings und vielfältigste Stellenangebote für Softwareentwickler.

Fazit

Wer sich so kurz vor Weihnachten aufmachte, um noch einmal richtig Wissen zu tanken, hatte mit der .NET Developer Conference 2024 die richtige Wahl getroffen. Die DDC brachte die .NET-Begeisterten zusammen und sorgte für Happiness at Work für die Teilnehmenden. Durch Expertenwissen vom Feinsten, zwanglosen Austausch im Sinne der Community-Stärkung, gut gefüllte Säle und leckeres Essen gab es einen Push für Motivation und Inspiration. Das Team der DDC freut sich schon auf kommendes Jahr, wenn es wieder die Akkus der Teilnehmenden auffüllen darf. (tib) ■

[1] Kevin Faller auf LinkedIn,

www.dotnetpro.de/SL2502DDC24_1

[2] TextControl, www.textcontrol.com

[3] Xebia/Xpirit Germany, <https://xebia.com>

[4] Toypad Challenge,

www.wendelpunkt.de/toypad-challenge

dnppCode

A2502DDC24

Bild 1 André Krämer ließ Cross-Plattform-Apps mit .NET entstehen.

Bild 2 Wenn es um .NET-Architektur und Softwarequalität geht, ist David Tielke in seinem Element.

Bild 3 Keynoter Christian Giesswein verriet seine Geschichte mit .NET.

Bild 4 Iris Classon erläuterte, wie man Migrationen meistert.



5



6

Bild 5 Dr. Holger Schwichtenberg zeigte, was mit Entity Framework und Blazor möglich ist.

Bild 6 Mark Allibone verwandelte eine Website in eine hybride .NET-MAUI-Anwendung.

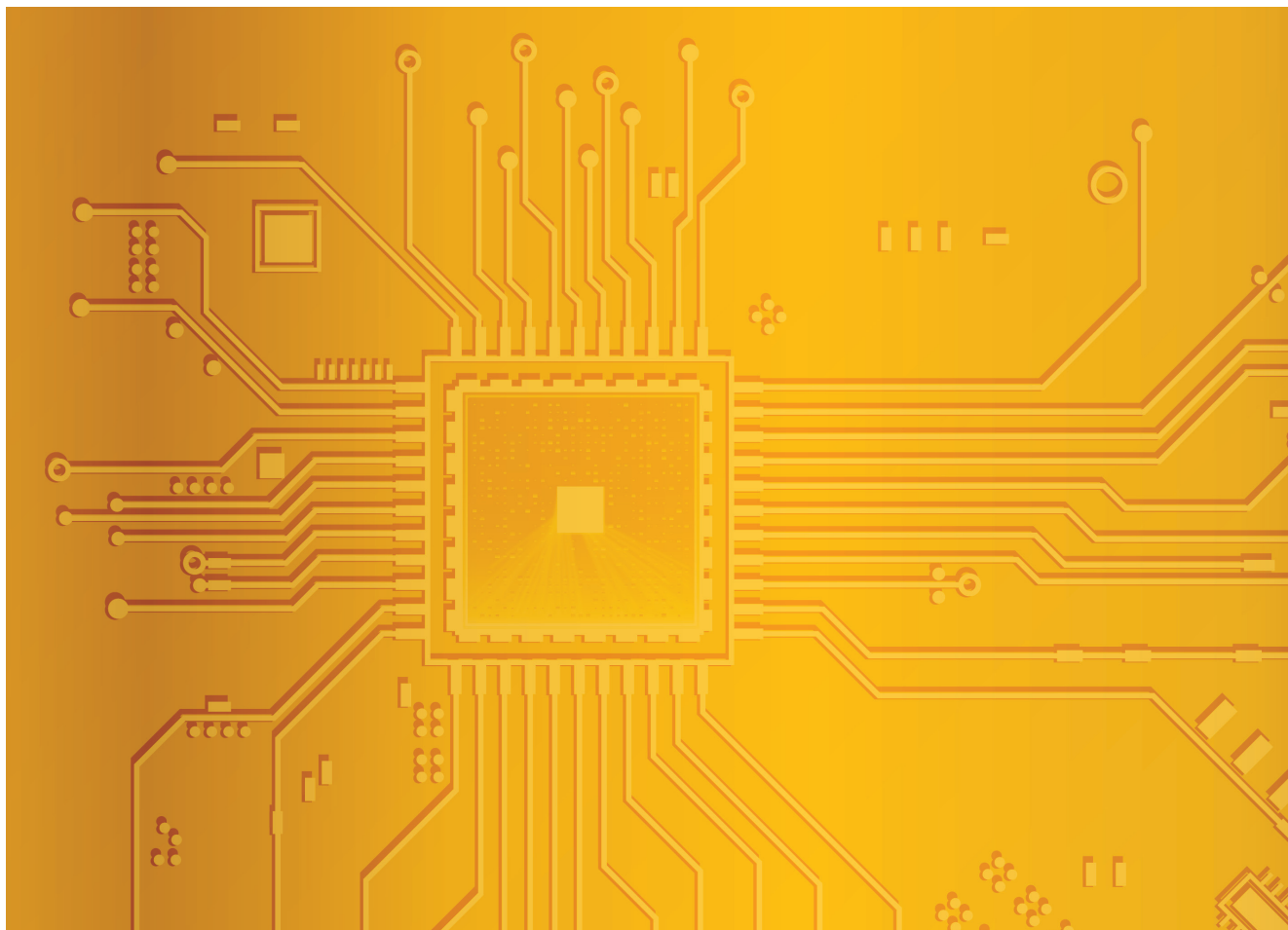


Foto: Shutterstock / Den photographer 1985

QUANTUM COMPUTING: EIN ÜBERBLICK

Neue Denkweisen

Die Landschaft der Quantenprogrammiersprachen und Quantencomputerarchitekturen wächst rapide. Doch wie vernetzt und steuert man Qubits, und welche Errors treten auf?

Quantum Computing sitzt inmitten der neuen Generation an 2G-Quantentechnologien, zu denen wir sechs Themenbereiche zählen: Quantencomputer, Quantenalgorithmen, Quantenkryptografie, Quantensensoren, Quantennetzwerke und Quantensimulation. Wie im traditionellen Computing kann der Begriff „Quantum Computing“ daher klar in zwei Forschungsbereiche zu Quantencomputern (Hardware) und Quantenalgorithmen (Software) aufgeteilt werden. Auf der Hardwareseite gibt es drei große Herausforderungen:

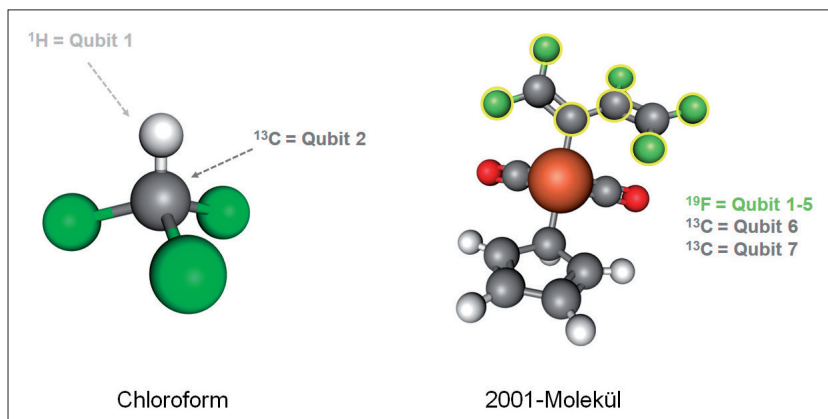
- Stabilität von Qubits und Verlässlichkeit von Gatteroperationen,
- die Einigung auf weltweite Standards und
- das Verknüpfen einzelner Hardware zu Netzwerken.

Um dies zu verstehen, gehen wir kurz in die Geschichte zurück. 1998 wurde der erste Quantencomputer von Chuang,

Gershenfeld und Kubinec in Berkeley realisiert [1]. Dieser besaß zwei Qubits und beruhte auf Chloroformmolekülen und dem Kernspintomografie-Prinzip. Wenn Sie schon einmal ei-

• Alternativen zu Chloroform

Vielleicht fragen Sie sich: Könnten die ^{35}Cl -Chloratome (mit ebenfalls ungerader Nukleonenzahl) nicht auch als Qubit genutzt werden? Ja, das hätte man tun können, diese wechselwirken ebenfalls mit dem Magnetfeld und haben einen halbzahligen Spin, allerdings mit $I = \frac{3}{2}$. Wenn Sie dies und die Unterschiede berechnen wollen ... herzlich willkommen in der Welt der Quanten- und Kernphysik!



Chloroform und Shor-Molekül: Kernmagnetresonanz-basierte Quantencomputer (NMR) waren die ersten Hardware-Realisationen (Bild 1)

nen MRT-Scan im Krankenhaus erleben durften, kennen Sie dieses Prinzip. Man gibt Moleküle (im Fall des ersten Quantencomputers Chloroform, im Fall Ihres MRT-Scans vor allem Wasser) in ein starkes Magnetfeld. Atome in diesen Molekülen besitzen einen Spin, der sich vorzugsweise in Richtung des starken Magnetfelds ausrichtet, das entspricht einem Niedrigenergiezustand. Diesen niedrigen Energiezustand bezeichnen wir mit $|0\rangle$. Dann kann man mit elektromagnetischen Pulsen (im Fall des MRT also Radiowellen) dem System eine gewisse Energie hinzufügen, und manche Atome werden nun im Spin gegen das Magnetfeld geflippt – sie wechseln in einen Hochenergiezustand. Diesen bezeichnen wir mit $|1\rangle$.

Nun ist der springende Punkt, dass die Energie zwischen $|0\rangle$ und $|1\rangle$ quantisiert ist. Und bei dieser Quantisierung kommt es auf das Umfeld des Atomspins an. In anderen Worten: Wie in einem Aufzug gibt es keinen Zwischenausstieg, und nur wenn die Frequenz der Radiowelle genau der Energie des Übergangs zwischen $|0\rangle$ und $|1\rangle$ entspricht, wird ein starkes Resonanzsignal gemessen. Im Fall Ihres MRT-Scans etwa könnte das Klinikpersonal an den Resonanzsignalen und speziell an deren Frequenzen erkennen, welcher Gewebetyp sich wo in Ihrem Körper befindet.

Im Fall des Quantencomputers wurden Resonanzfrequenzen genutzt, um die Atomspins der Chloroformmoleküle für Berechnungen zu steuern. Die Atomspins waren die Qubits (kurz für Quantenbits) in der Berechnung, die elektromagnetischen Inputs führten die Operationen (sprich die Quantengatter) durch.

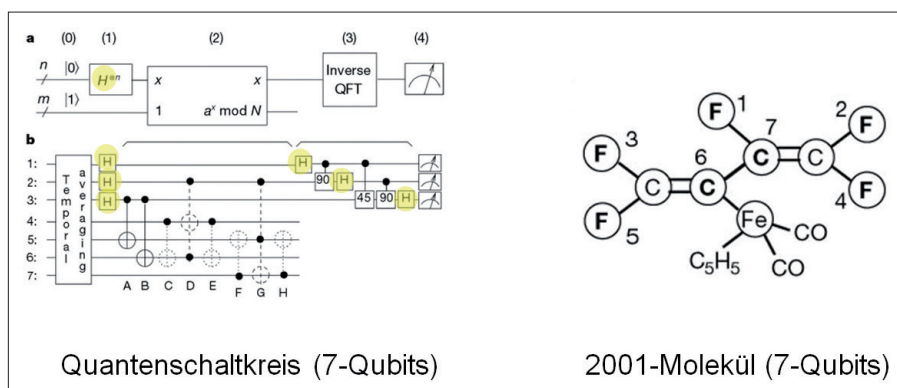
Jetzt wird klar, warum auf der Hardwareseite die Stabilität der Qubits und die Verlässlichkeit von Gatteroperationen wichtige technische Anforderungen darstellen. Es ist nicht einfach, Kernspins und genaue Radiowellenpulse exakt zu bauen und über längere Zeit zum Beispiel in einem hochenergetischen Zustand zu halten.

Betrachten wir das Chloroformmolekül in Bild 1 (links): Wie viele Qubits hatte der erste Quantencomputer? Die Antwort ist zwei, das Wasserstoffatom besitzt ein Proton (und damit eine ungerade Nukleonenzahl) und der Kohlenstoff ^{13}C besitzt sechs Protonen und sieben Neutronen (wieder eine ungerade Nukleonenzahl), wodurch ein Kernspin von $I = \frac{1}{2}$ entsteht, der mit dem Magnetfeld wechselwirken konnte (vergleiche Kasten Alternativen zu Chloroform). Auf diesen zwei Qubits wurde der Grover-Algorithmus erstmals demonstriert.

Zwei Jahre später, im Jahr 2001, wurde der Shor-Algorithmus ebenfalls von einer Gruppe um Isaac Chuang auf einem Quantencomputer über Kernspins demonstriert.

In anderen Worten: Ein Molekül löst die Rechenaufgabe $15 = 3 \cdot 5$. Beim Shor-Algorithmus geht es um das Aufbrechen von Zahlen in deren Primfaktoren. Wie viele Qubits wurden dafür benötigt? Die Antwort ist sieben mit fünf ^{19}F -Fluoratomen und zwei ^{13}C -Kohlenstoffatomen [2], die mit Frequenzpulsen von ~ 470 MHz und ~ 125 MHz angesprochen werden konnten (rechts in Bild 1).

Schickt man einen sogenannten $\pi/2$ -Puls (oder 90° -Rotationspuls) an ein Qubit, kann man eine Superposition einstellen. Damit entsteht ein Quantenzustand zwischen $|0\rangle$ und $|1\rangle$, den wir als $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$ bezeichnen. Diese Operation nennt sich algorithmisch Hadamard-Gatter. Wir kommen

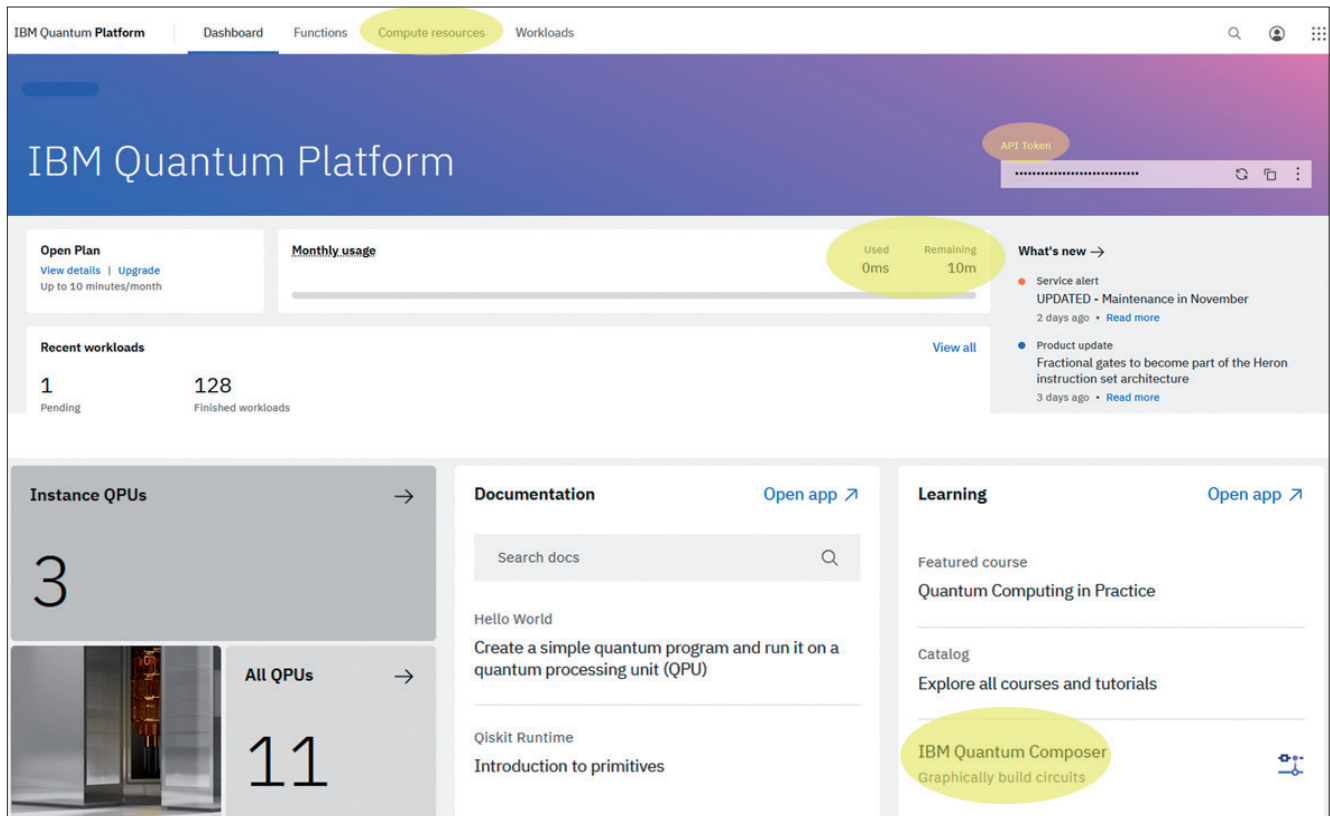


Schaltkreis des implementierten Shor-Algorithmus: Das 2001-Molekül mit 7 Qubits (vergleiche Bild 1) und der darauf implementierte Quantenschaltkreis (Bild 2)

später noch einmal darauf zurück, sehen dieses H-Quantengatter aber bereits sehr integral im Bild 2 stehen (gelb markiert), das den Schaltkreis des implementierten Shor-Algorithmus zeigt.

Weltweite Standardhardware

Seit 2001 hat sich die Quantencomputerhardware weiterentwickelt. Große Unternehmen wie IBM oder Google nutzen keine Moleküle mehr, sondern Superconducting Circuits. Stellen Sie sich diese als Schwingkreise vor, die zwischen ►



Übersicht der IBM-Quantenplattform, die Ihnen freien Zugang auf echte Quantencomputer ermöglicht (Bild 3)

zwei Energien $|0\rangle$ und $|1\rangle$ schwingen können und mit Mikrowellenfrequenzen angesprochen werden. Das Prinzip bleibt ähnlich.

Andere Hardwarearchitekturen sind:

- **Ion Traps:** Ionisierte Metallionen werden in einer Magnetfeldfalle mit Lasern angesprochen.
- **Neutral- und Rydberg-Atome:** Atome in optischen Fallen, die mit Lasern angesprochen werden.
- **Photonen:** Lichtteilchen, bei denen zum Beispiel Polarisation über Polarisationsfilter gesteuert wird.
- **Spin Qubits:** Elektronenspins [3] in leitenden Materialien, die über das Anlegen elektrischer Spannung einzeln isoliert und angesprochen werden.

• Quantensimulatoren

Quantensimulatoren in der Experimentalphysik sind keine klassischen Computer, die eine Qubitanzahl simulieren. Vielmehr setzen Physiker kontrollierbare Quantensysteme auf, um anhand derer bestimmte Parameter zu untersuchen und so verwandte Quantensysteme zu verstehen (Simulation eines Quantensystems durch ein verwandtes). Außerdem gibt es eine weitere Nutzung des Begriffs: Quantensimulation auf Quantencomputern. Algorithmen und Gatterabfolgen werden entwickelt, um chemische, materialtechnische oder andere komplexe Natursysteme zu berechnen.

Die Liste ist nicht vollständig, mehr dazu ist auch im Artikel „Wegweisende Symbiose“ [4] zum Thema Quantencomputing in der Cloud in dieser dotnetpro-Ausgabe zu finden. Was Sie aber bereits vermuten: Die unterschiedlichen Hardwarearchitekturen bringen unterschiedliche Stabilität der Qubits, unterschiedliche Frequenzspektren und unterschiedliche Eigenschaften mit. Superconducting Circuits etwa werden maschinell in Reinräumen erzeugt, und die Resonanzfrequenz jedes Qubits kann individuell eingestellt werden. Somit fabrizieren IBM oder Google Chips, auf denen die unterschiedlichen Qubits auf unterschiedliche Frequenzen anspringen, was Vorteile für eine genaue Ansteuerung liefert. Andererseits sind die Energieniveaus für Ionen und Atome weltweit immer stabil reproduzierbar, ein Vorteil von Ion Traps.

Während Superconducting Circuits (vor allem IBMs Transmon-Qubits) speziell in den Jahren 2018 bis 2020 die vorherrschende Architektur waren, stellte Intel unlängst ein Spin-Qubit-System vor [5], das extrem interessant werden könnte, weil der Herstellungsprozess klassischen Transistoren angeglichen ist und daher schnell gestreamlined werden könnte.

Photonen bergen großes Potenzial, um auf Raumtemperatur zu arbeiten oder an längere Distanzen für Signalübertragung gekoppelt zu werden. Beispielsweise wurden 2017 zwischen Österreich und China mit dem Micius-Satelliten verschränkte Photonen verschickt, um nach dem abhörsicheren BB84-Protokoll, einer Quantenverschlüsselungsmethode, eine Videokonferenz abzuhalten.

Leider gibt es heutzutage nicht den einen „Quantentransistor“. Zurzeit sind viele Hardwarearchitekturen auf dem

Mathematisch

Ein Quantenschaltkreis beginnt traditionell mit allen Qubits im niedrigen Energiezustand $|0\rangle$. Wir schreiben den Quantenzustand $|\Psi\rangle = 1 \cdot |0\rangle + 0 \cdot |1\rangle$, können dies allerdings auch in Vektorform bringen. Somit kann ein Qubit zwei (komplexe) Zahlen speichern:

$$|\Psi\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \begin{array}{l} \leftarrow \text{Koeffizient } \alpha \text{ für } |0\rangle \\ \leftarrow \text{Koeffizient } \beta \text{ für } |1\rangle \end{array}$$

Für zwei Qubits:

$$|\Psi\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} \quad \begin{array}{l} \leftarrow \text{Koeffizient } \alpha \text{ für } |00\rangle \\ \leftarrow \text{Koeffizient } \beta \text{ für } |01\rangle \\ \leftarrow \text{Koeffizient } \gamma \text{ für } |10\rangle \\ \leftarrow \text{Koeffizient } \delta \text{ für } |11\rangle \end{array}$$

In gleicher Weise können Sie die experimentellen Mikrowellenpulse in eine theoretische Sprache gießen. Quantengattern entsprechen quadratische Matrizen, deren Größe mit der Qubitanzahl korreliert. Ein 1-Qubit-Gatter ist eine 2×2 -Matrix, ein 2-Qubit-Gatter eine 4×4 -Matrix, und diese Matrizen werden mit den Zustandsvektoren multipliziert, um den theoretischen Ablauf der Operationen zu beschreiben. Quantencomputer sind somit viel lineare Algebra, und dasselbe gilt für maschinelles Lernen (vergleiche Bild 4)!

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \begin{array}{l} \leftarrow \text{Koeffizient } \alpha \text{ verschwunden} \\ \leftarrow \text{Koeffizient } \beta \text{ geflippt zu } 100\% \end{array}$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \begin{array}{l} \text{Koeffizient } \alpha \text{ zu } 50\% \\ \leftarrow (|\alpha|^2 = \text{Wahrscheinlichkeit}) \\ \leftarrow \text{Koeffizient } \beta \text{ zu } 50\% \\ \leftarrow (|\beta|^2 = \text{Wahrscheinlichkeit}) \end{array}$$

Markt, und weitere stecken in den Kinderschuhen der Laboratorien. Daher sind industrielle Standards noch nicht weltweit definiert, die rechtlichen Rahmenbedingungen werden eventuell in den nächsten Jahren folgen, und aktive Forschung wird betrieben, um einzelne Chips, einzelne Quantencomputer sowie Satelliten zu Informationsnetzwerken zusammenzufügen (vergleiche [6] und [7]).

Im Jahr 2023 haben nach McKinsey rund 367 000 Personen relevante Studien in dem Quantentechnologiebereich abgeschlossen, und es wurden rund 42 Milliarden US-Dollar weltweit von Regierungen in die Technologieentwicklung investiert [8]. Deutschland ist mit dem Munich Quantum Valley Vorreiter (5,2 Milliarden US-Dollar) und direkt an zweiter Stelle nach China (15,3 Milliarden US-Dollar). Die Quantencomputing-Initiative des DLR fördert aktuell 39 Projekte.

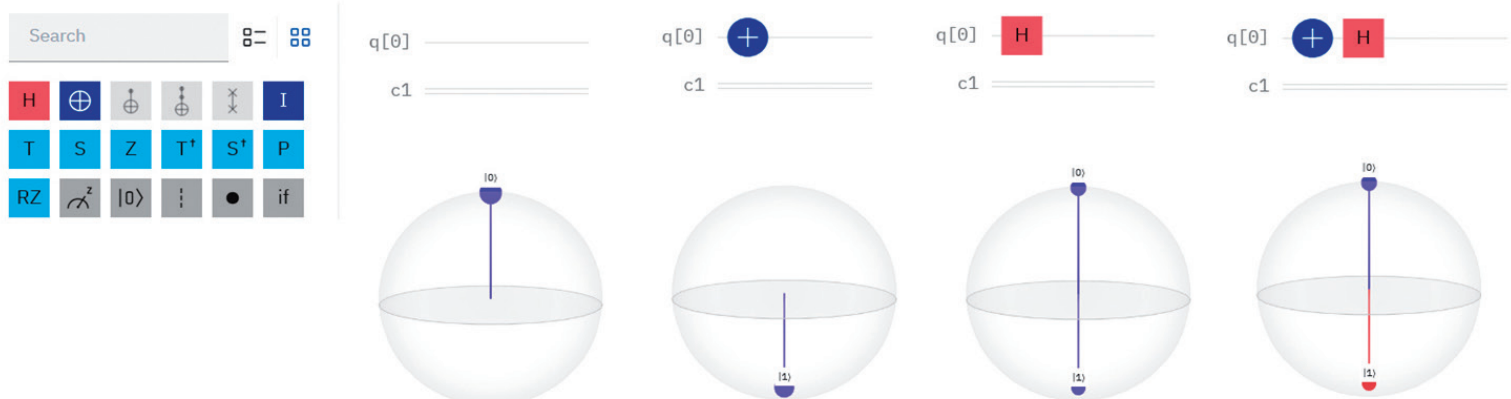
Jüngst wurde in Hamburg der erste industrietaugliche Quantencomputer basierend auf Ion Traps in Betrieb genommen [9] [10], in Jülich (Nordrhein-Westfalen) gibt es seit 2022 den Quantenannealer JUPSI [11]. In Ehningen (Baden-Württemberg) hat IBM im Oktober 2024 das erste Quantenrechenzentrum mit Superconducting Circuits außerhalb der USA eingeweiht [12]. Und über die Cloud haben Sie einen Quantencomputer gleich bei sich daheim!

Zugriff auf Quantencomputer

Geben Sie die Webadresse <https://quantum.ibm.com/> in Ihrem Webbrowser ein. Nach einer kostenlosen Registrierung öffnet sich die in Bild 3 gezeigte Plattform-Übersicht mit frei zugänglicher Hardware zu IBM-Quantencomputern über die Cloud, Ihrem persönlichen API-Key (rechts oben) und Softwaredokumentation. Zusätzlich ist eine Liste der durchgeführten Jobs, ein Barometer der genutzten Zugriffsdauer (pro Monat sind zehn Minuten kostenlos, für Premium-Nutzer gibt es erweiterte Funktionen) und unten eine Weiterleitung zu Jupyter-Notebooks, Tutorials und einem grafischen Circuit-Composer ersichtlich.

Schauen Sie sich kurz den Circuit-Composer an [13], und klicken Sie sich danach zurück zu der Übersicht aller verfügbaren Quantencomputer-QPUs [14].

Der Circuit-Composer zeigt die fundamentalste Programmieraufgabe im Quantencomputing-Bereich: Wir definieren Qubits, auf denen wir Quantengatteroperationen (Quantum Gate) ausführen. Mit jedem hinzugefügten Gatter ändern wir den Quantenzustand eines Qubits. Beispielsweise funktioniert ein X-Gatter wie ein klassisches NOT-Gatter. Fangen Sie also mit einem Qubit in Zustand $|0\rangle$ an, so flippt ein X-Gatter dieses Qubit in den Zustand $|1\rangle$. Im Unterschied zu ►



IBM Circuit Composer mit vier Minischaltkreisen. Verwendet werden X-Gatter und H-Gatter aus dem Gesamtrepertoire links, um Zustandsänderungen zu erzielen (Bild 4)

Quantum Computing mit Simulator

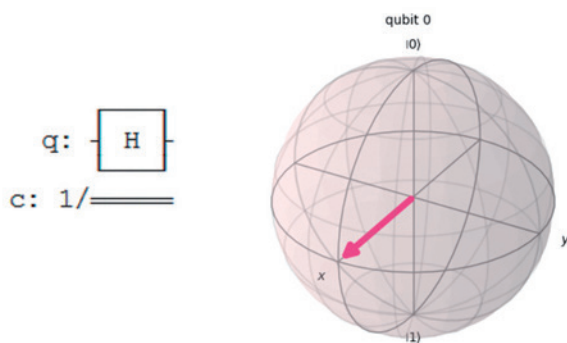
Quantenschaltkreise können in kleinem Format auf Statevektor-Simulatoren oder Hardwaresimulatoren mit Messungen getestet und untersucht werden.

Öffnen Sie eine Python-Entwicklungsumgebung. Konkret könnte das ein Jupyter-Notebook oder ein Colab-Notebook sein. Sie werden vier Standardschritte durchführen und die Ergebnisse in der Bloch-Sphäre und nach Messung auf einem Simulator plotten. Wir beginnen mit dem Import der benötigten Libraries. Qiskit besteht aus mehreren Modulen: Qiskit Aer stellt Simulatoren bereit, mit Qiskit Visualization können wir Bloch-Sphären visualisieren.

```
!pip3 install qiskit
!pip3 install qiskit_aer

from qiskit import QuantumCircuit
from qiskit.visualization import
    plot_bloch_multivector, plot_histogram
from qiskit.quantum_info import Statevector
from qiskit_aer import AerSimulator
import matplotlib.pyplot as plt

# 1. Definieren eines Quantenschaltkreises mit einem
#   Qubit und einem klassischen Bit
qc = QuantumCircuit(1, 1)
qc.h(0)
# Anwenden des X-Gatters auf das erste Qubit
qc.draw()
# funktioniert am besten am Ende einer
# Notebook-Zelle
```



```
Statevector([0.70710678+0.j, 0.70710678+0.j],
            dims=(2,))
```

Der Minischaltkreis mit einem Qubit, einem klassischen Bit und einem H-Gatter. Die Superposition wird auf der Bloch-Sphäre am „Äquator“ angezeigt. Der Zustandsvektor $[0.707, 0.707]$ gibt die Koeffizienten wie erwartet für den Quantenzustand $|\Psi\rangle = 1/\sqrt{2} |0\rangle + 1/\sqrt{2} |1\rangle$ an: $[\alpha, \beta] = [1/\sqrt{2}, 1/\sqrt{2}] = [0.707, 0.707]$ (Bild 7)

```
# 2. Ausgabe des Zustandsvektors
statevector = Statevector(qc)
print(statevector)      # sollte  $\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$ , also
                        # [0.707 0.707] ausgeben,
                        # wie vorher hergeleitet
```

```
# 3. Darstellung des Zustandsvektors auf der
#   Bloch-Sphäre (Bild 7)
```

```
# Challenge: Probieren Sie, andere Gatter zu
# verwenden, siehe [31]!
```

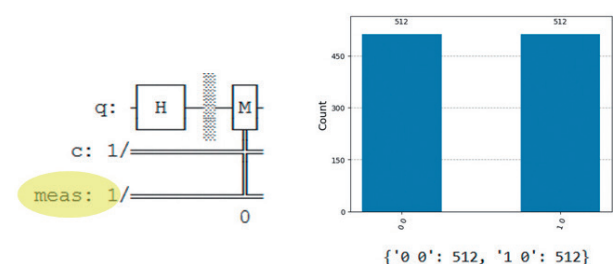
```
# Wo landen Sie auf der Bloch-Sphäre? Was ist der
# Unterschied zwischen X-, RX-, CX-Gattern?
```

```
plot_bloch_multivector(statevector)
```

```
# 4a. Kollabieren des Quantenzustands und Messung an
#   simulierter Hardware
```

```
qc.measure_all()
backend = AerSimulator()
results = backend.run(qc).result()
plot_histogram(results.get_counts())
```

```
# Eine 50:50-Verteilung entsteht (Bild 8).
```



QiskitError: 'Cannot apply instruction with classical bits: measure'

Auf einem realen Quantencomputer werden alle Zustände der Bloch-Sphäre für Berechnungen verwendet, gemessen werden kann allerdings nur „Nordpol“ mit $|0\rangle$ oder „Südpol“ mit $|1\rangle$. Die Koeffizienten α, β entscheiden die Wahrscheinlichkeit der jeweiligen Messergebnisse. Mit dem AerSimulator muss eine Messung hinzugefügt werden, der Zustandsvektor kann nicht mehr abgerufen werden – *QiskitError* (da quantenphysikalisches Zufallsereignis). Das Ergebnis im meas-Register zeigt nach 1024 Shots (Replikaten) eine exakte Gleichverteilung der Häufigkeiten mit je 512-mal $|0\rangle$ beziehungsweise $|1\rangle$ als Simulatorergebnis (Bild 8)

klassischen Computern lassen sich Qubits aber nun nicht nur zwischen 0 und 1 flippen. Qubits können jeden Punkt auf der Oberfläche einer Kugel mit Nordpol $|0\rangle$ und Südpol $|1\rangle$ einnehmen (Bild 4). Dies bezeichnen wir als Bloch-Sphäre. Ein Beispiel dafür kennen Sie bereits: das H-Gatter. Beginnen Sie mit einem Qubit im Zustand $|0\rangle$, so rotiert das H-Gatter den Quantenzustand an den Äquator, in eine 50:50-Superposition, siehe Kasten **Mathematisch**.

Quantensimulatoren und Quantenprogrammiersprachen

Wenn Sie dies auf dem Circuit-Composer ausführen, läuft der Schaltkreis nicht auf einem physischen Quantencomputer, sondern einem Simulator. Der Begriff „Simulator“ wird im Bereich der Quantentechnologie vielseitig und missverständlich eingesetzt. Womit Sie als Programmierer:in konfrontiert werden, wird das Nutzen eines klassischen Computers sein, um eine kleine Anzahl an Qubits rechnerisch zu simulieren, siehe Kasten **Quantensimulatoren**. Dies ist für kleine Schaltkreise möglich, allerdings wächst mit jedem hinzugefügten Qubit die Anzahl der komplexen Koeffizienten auf das Dop-



Die wichtigsten Quantenprogrammiersprachen, SDKs und Packages (Bild 5)

pelte, und auch die Matrizen der Quantengatter skalieren entsprechend. Beispielsweise kann man mit 4 GByte RAM ungefähr 27 Qubits simulieren [15], für 28 Qubits bräuchte man bereits rund 16 GByte.

Schauen wir uns an, wie wir Quantenschaltkreise in einem Programm definieren und ausführen können. Zur Verfügung steht Ihnen mittlerweile eine große Auswahl an Open-Source-Packages, Development-Kits und teilweise eigenen Quantenprogrammiersprachen [16]: Angefangen hat es unter an-

● Weitere Quantenprogrammiersprachen

QASM ist eine Mischung zwischen C und Assemblersprache, während Pulser und QuTIP eine starke Brücke zu Physik und Hardware bilden. Trauen Sie sich, Pulser einmal aufzumachen! Auch wenn Sie als Programmierer:in später kaum Pulssequenzen definieren werden, ist es lustig und gut für die Intuition, mit Amplitude, Zeit und Phase eines Laserpulses zu spielen.

derem in den Jahren 1998 bis 2000 an der TU Wien mit der Quantum Computation Language (QCL) von Bernhard Ömer. In ihrer Syntax ähnelte sie der Programmiersprache C. Neben klassischen Datentypen gibt es dort Quantendatentypen wie *qureg*, *quvoid*, *quconst* oder Built-in-Funktionen zu Quantengattern wie *H*, *Matrix2x2*, *X*, *Y* und *Z*.

In weiteren Artikeln dieser dotnetpro-Ausgabe lernen Sie die Quantenprogrammiersprache Q# aus dem Microsoft QDK (Quantum Development Kit) kennen [17] und erfahren, wie man einen Quantensimulator über C# aufsetzt [18].

Ein großer Gegenspieler zu Q# ist die an der ETH entwickelte abstrahierte Programmiersprache Silq. Ein Vergleich zwischen Q# und Silq ist unter [19] zu finden, siehe auch den Kasten **Weitere Quantenprogrammiersprachen**. Die meisten Programmierer:innen verwenden allerdings IBMs Qiskit oder Googles Cirq. Diese sind keine eigenen Programmiersprachen, sondern Python-basiert. Können Sie ein bisschen Python, sind Sie also mehr als geeignet, bereits ohne Circuit-Composer loszulegen, wie im Kasten **Quantum Computing mit Simulator** gezeigt!

Bild 5 zeigt die Quantenprogrammiersprachen-Landschaft im Überblick. Qiskit mit der IBM-Plattform hat eine besonders große Community mit freiem Zugriff auf echte Hardware. Ebenso können QuTIP und Pulser Python-affine Programmierer:innen begeistern, speziell Pulser ermöglicht ein gutes Verständnis zum Hardwareübergang. Cirq kommt mit TensorFlow-Quantum- (Machine Learning) und OpenFermion-Erweiterungen (Chemie). Qrisp bietet ähnlich zu Silq eine High-Level-Programmiersprache. Anstelle von Gattern braucht man hier oftmals nur Funktionen aufzurufen.

Während es anfangs – IBM stellte erstmals 2016 Qiskit und Quantencomputer über die Cloud der Welt zur Verfügung [20] – üblich war, ganz analog einen Quantenschaltkreis anstelle eines Simulators an einen Quantencomputer zu ►

● Optimierungen

Quantengatter können parallel zwischen verschiedenen Qubits ausgeführt werden. Beispielsweise könnten wir einen `qc = QuantumCircuit(2, 2)` mit zwei Qubits definieren und dann auf das erste Qubit ein X-Gatter und auf das zweite Qubit ein H-Gatter legen. Das X- und H-Gatter können – weil verschiedene Qubits – zum

gleichen Zeitpunkt stattfinden. Sie bilden gemeinsam eine „Schicht“ / Spalte / Layer im Schaltkreisdiagramm. Würde man zu dem X-Gatter ein H-Gatter auf dem ersten Qubit hinzufügen, muss dieses seriell ausgeführt werden. Die Depth erhöht sich: Layer = 1 auf Layer = 2.

● Quantum Computing mit echter Hardware

Öffnen Sie erneut Ihre Python-Entwicklungsumgebung. Praktisch wäre wieder ein Jupyter-Notebook oder ein Colab-Notebook. Zuerst installieren Sie den IBM Runtime Service, danach verbinden Sie sich mit dem API-Token von <https://quantum.ibm.com/> (rechts oben im Dashboard) mit dem IBM Quantum Runtime Service über die Cloud. Lassen Sie sich eine Liste aller verfügbaren Quantencomputer mit `service.backends()` ausgeben. Ist `ibm_brisbane` dabei?

4b. Echten Quantencomputer ansprechen mittels Sampler:

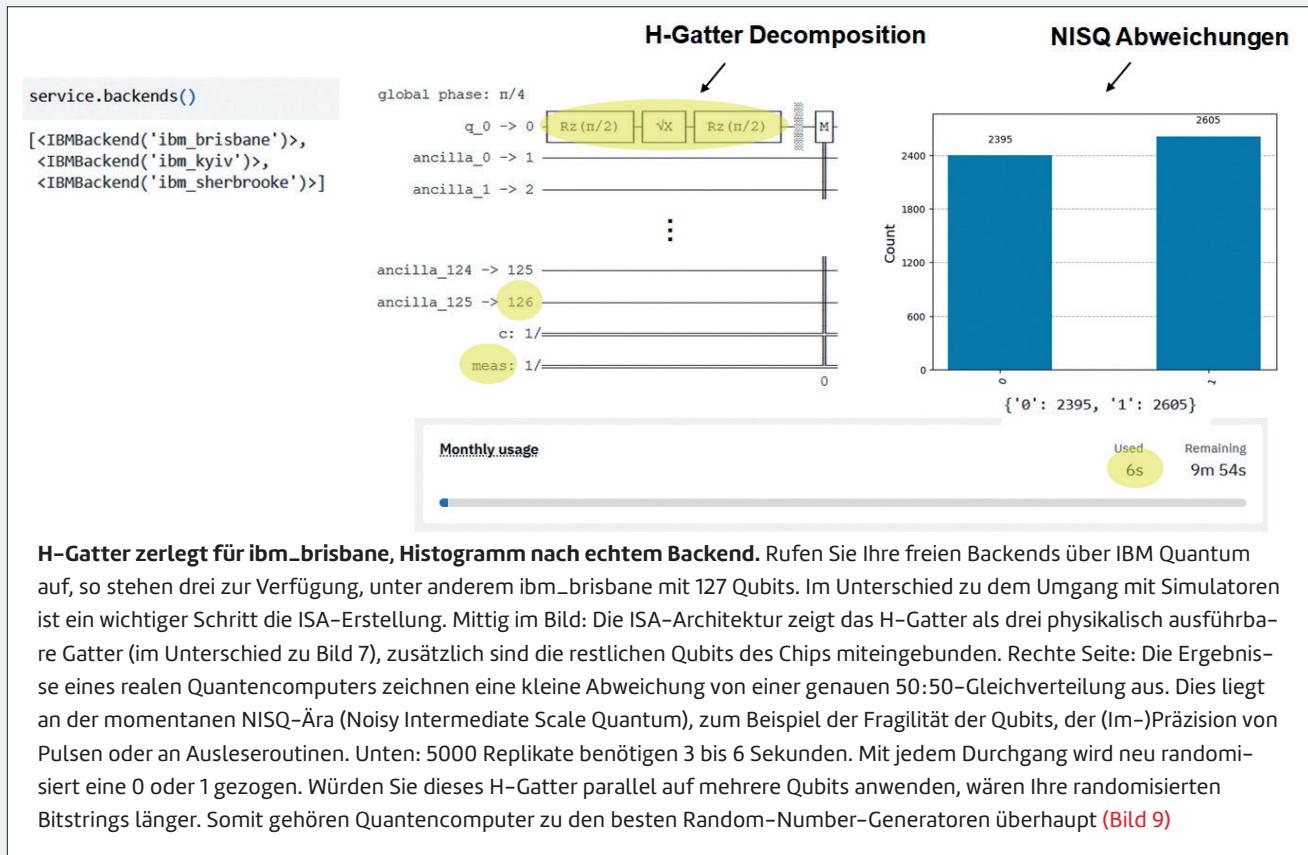
```
!pip3 install qiskit_ibm_runtime
from qiskit_ibm_runtime import QiskitRuntimeService
```

```
service = QiskitRuntimeService(channel="ibm_quantum",
                                token="s. Bild 3, Ihr Token")
```

```
print(service.backends())
backend = service.least_busy(simulator=False,
                              operational=True)
```

Anstatt mittels `backend.run()` den Quantenschaltkreis nun gleich ausführen zu können, kommen zwei Schritte #5 und #6 hinzu.

```
qc.measure_all()
# diese Zeile ist die letzte Zeile, welche wir aus
# 4a. mitkopieren
```



schicken, sind die Chip-Architekturen von damals drei bis fünf Qubits auf heutzutage bereits über 100 physikalische Qubits angewachsen, und je komplexer die Chips und Möglichkeiten zu Errormitigation, -korrektur und Schaltkreisoptimierung, desto notwendiger werden Transpiler [21], Pass Manager [22] und Transformer.

Transformer sind hier etwas anderes als die bekannten Large-Language-Model-Architekturen. In der offiziellen Transformer-Cirq-Dokumentation werden drei wichtige Aufgaben der Transformer beschrieben, um einen Schaltkreis für einen Quantenchip vorzubereiten:

- **Gate Decomposition:** Zerlegung der programmierten Quantenoperationen in Quantengatter, welche als Pulse wirklich ausführbar sind.
- **Qubit-Mapping und -Routing:** Zuweisen der logischen Qubits zu physikalischen Qubits auf dem Chip, Beachten der Verbindungen zwischen physikalischen Qubits (nicht alle sind verschränkbar).
- **Schaltkreisoptimierung:** Vereinigung vieler Quantengatter in möglichst effiziente Operationen, Reduktion der Circuit Depth (Anzahl der Operationen, siehe Kasten **Optimierungen**) und Reduktion der Circuit Width (Anzahl der Qubits).

```
# 5a. Definieren eines Pass-Managers

# 5b. Umführen des Quantenschaltkreises in hardware-
# konforme ISA (Instruction Set Architecture)
from qiskit.transpiler import
    generate_preset_pass_manager
prun = generate_preset_pass_manager(
    backend=backend, optimization_level=1)
isa = prun.run(qc)
isa.draw()
# je nach Backend eine Liste von ~127 Qubits mit
# Gattern

# 6. Ansprechen des Quantencomputers im SamplerV2-
# Modus (EstimatorV2-Modus ist die Alternative)
# Mittels SamplerV2- oder EstimatorV2-Runtime-Modus
# können Laufzeitparameter wie "Default Shots" oder
# "Resilience Level" in EstimatorV2 für Noise-Level
# definiert werden
from qiskit_ibm_runtime import SamplerV2 as Sampler

sampler = Sampler(mode=backend)
sampler.options.default_shots = 5000
real_job = sampler.run([isa])
# Listentyp, da eine Reihe an Schaltkreisen in
# eine Runtime passt.
results = real_job.result()
plot_histogram(results[0].data.meas.get_counts())
# Klassische Speicherbits mit "meas" abrufen
```

Wir haben uns wenig um den Unterschied zwischen Sampler oder Estimator gekümmert. Zu bemerken wäre, dass ein Sampler die technisch intuitiveren Ergebnisse liefert: unsere Histogramme der Qubitzustände, aus denen wir statistisch Wahrscheinlichkeiten und Ergebnisse erfahren. Estimator liefert einen Erwartungswert und wird in Optimierungsaufgaben verwendet (minimiert). Dies klingt sehr physikalisch, bedeutet aber nicht viel anderes, als dass wir für eine Matrix (Quantenoperator/Observable) die passenden Eigenvektoren und minimalen Eigenwerte finden können (Bild 9).

Die Dokumentation unter [23] ist lesenswert, ebenso wie die Qiskit-Dokumentation zu Transpilern [24].

Diese Schritte werden Ihnen sofort deutlich, wenn Sie in der IBM-Cloudplattform die Übersicht aller verfügbaren Quantencomputer [14] wieder aufrufen und detaillierter inspizieren. Schauen Sie sich den Chip *ibm_brisbane* an. Dieser besitzt 127 Qubits und kann 30 000 Layeroperationen pro Sekunde (CLOPS) ausführen. In der Chiptopologie – der grafischen Darstellung der Anordnung von Qubits auf einem Chip – können wir in Farbe die Fehlerraten (je heller, desto schlechter) oder die Frequenzen (je heller, desto höhere Frequenz) ein-

stellen (Bild 6). Hier sehen Sie einerseits, dass wie anfangs erwähnt Mikrowellen für Superconducting Circuits im Bereich von 4 bis 5 GHz verwendet werden und diese fabrizierten Superconducting-Qubits wirklich verschiedene Resonanzfrequenzen haben, im Gegensatz zu Atomen oder Ionen. Andererseits sehen wir, dass die 127 Qubits nicht mit jedem anderen verbunden sind, sondern in einem wabenförmigen Aufbau, wobei jedes Qubit zwei beziehungsweise vier Nachbarn besitzt. Diese Connectivity bestimmt Verschränkungsmöglichkeiten. Möchten wir beispielsweise Qubit 42 mit Qubit 69 verschränken, hätte das früher einen Error ausgegeben. Mithilfe von ISA (Instruction Set Architectures) und Transpilern wird eine korrekte Zuordnung Ihres Codes zu einem Quantenchip automatisiert möglich; der Code gerät dadurch leider etwas komplizierter.

Probieren Sie die Ansteuerung des Chips *ibm_brisbane* im Vergleich zum Simulator vorher, wie im Kasten **Quantum Computing mit echter Hardware** gezeigt!

QFT, VQE, Shor, Grover, QAOA, QUBO ... Was bringt die Zukunft?

Während ich diesen Artikel verfasste, war ich total beeindruckt von den Ergebnissen einer Forschungsstudie über die Verwendung von Quantenprogrammiersprachen [25], laut der 16,3 Prozent der befragten Quantumcomputing-Ingenieure bereits mehr als fünf Jahre Erfahrung mit Quantenprogrammiersprachen hatten. Plötzlich fiel mir auf, dass dies nun auf mich bereits ebenso zutrifft ... ich hatte knapp nach Eröffnung der IBM-Quantenplattform mit Fünf-Qubit-Computern und ohne nennenswerte Grundkenntnisse begonnen. Was wichtig ist: Sie haben eine Ausbildung als Programmierer:in, und 86,2 Prozent aller Quantenprogrammierer:innen haben einfach mehr als fünf Jahre Erfahrung mit klassischen Programmiersprachen!

Genauso wie Sie Ihre Programmierweise verändert haben, als Sie oben zwischen Simulator und echter Quantenhardware umstiegen, haben auch wir Quantenprogrammierer:innen ständig umzudenken. Die Hardware verändert sich, die Laufzeitparameter, die theoretischen Algorithmen. Monat für Monat wächst die „Werkzeugkiste“. Sie sind also nicht zu spät, und Sie sind nicht zu unphysikalisch.

Jene Algorithmen, welche Sie kennenlernen können, sind:

- **QFT:** Quantum Fourier Transform (und dessen inverse Transformation) [26]
- **VQE:** Variational Quantum Eigensolver (für Eigenwertoptimierungsprobleme) [27]
- **Shor-Algorithmus:** Aufbrechen von RSA-Verschlüsselung [28]
- **Grover-Algorithmus:** Datenbanksuche mit quadratischem Speed-up [29]
- **QUBO:** Quadratic Unconstraint Binary Problem (wertvolle Problemformulierungsmethode)
- **QAOA:** Quantum Approximate Optimization Algorithm (Lösungsmethode für QUBO) [30]

Nochmals hervorheben möchte ich: Quantencomputer funktionieren auf eine kreative Art und Weise. Die erfolgreichen Algorithmen haben sehr unterschiedliche Designs. ►

2G-Quantentechnologie erlebt momentan einen großen Hype. Ist dieser berechtigt? Wir wissen es nicht. Quantentechnologien haben jedoch bereit viele physikalische Chancen eröffnet, und nun mit 2G-Quantentechnologien haben wir als Programmierer:innen die Möglichkeit, nicht nur mitzureden, sondern auch mitzugestalten.

Wir haben eingangs den ersten Quantencomputer aus dem Jahr 1998 erwähnt. Bereits morgen könnte uns eine neue Implementierung in Hardware oder Software überraschen. Warum uns Quantumcomputing heutzutage so beschäftigt, fußt ebenfalls auf überraschenden Ideen aus den Jahren 1994 und 1996, siehe den Kasten **Nutzen der probabilistischen Eigenschaften**. Bereits davor wurden Quantenalgorithmen entworfen,

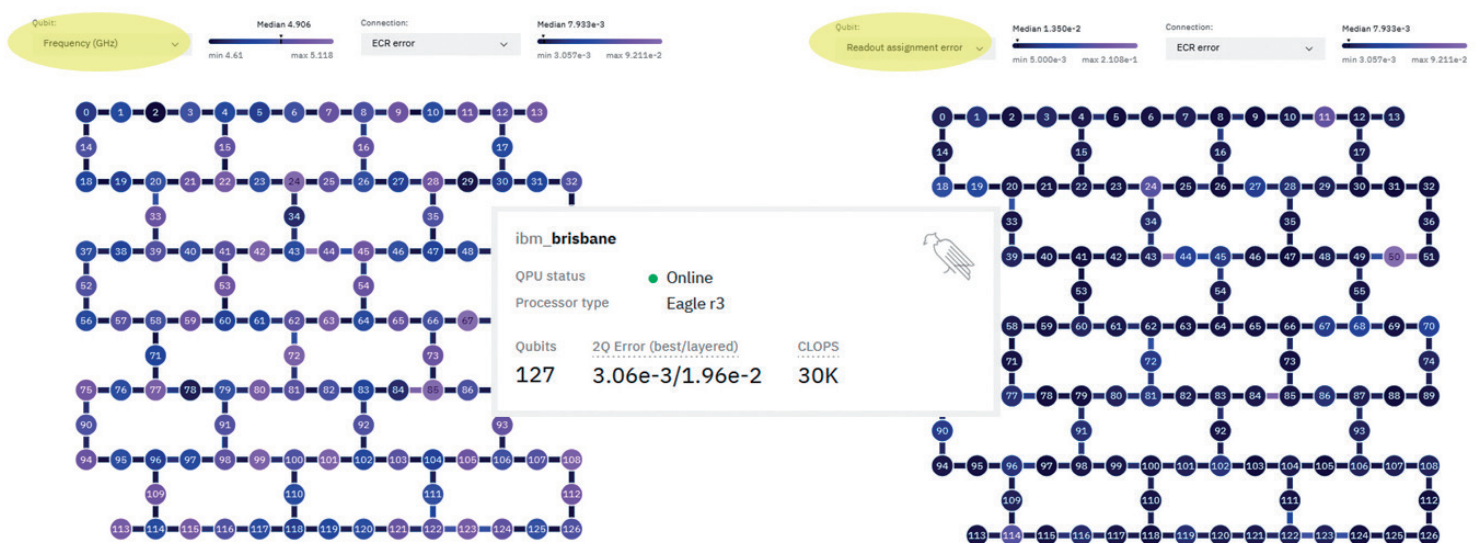
[2] Lieven M. K. Vandersypen et al., *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*, www.dotnetpro.de/SL2502Quantum2

[3] QuTech Academy, *Spin qubits (beginner level)*, www.dotnetpro.de/SL2502Quantum3

[4] Mykola Dobrochynskyy, *Wegweisende Symbiose*, *dotnetpro 2/2025*, Seite 20 ff., www.dotnetpro.de/A2502QuantumCloud

[5] Intel, *Intel Takes Next Step Toward Building Scalable Silicon-Based Quantum Processors*, www.dotnetpro.de/SL2502Quantum4

[6] The Quantum Flagship, <https://qt.eu>



Fehlerraten und Frequenzen für den `ibm_brisbane`-Quantencomputer. 127 physische Superconducting Circuits (Transmon-Qubits) sind in wabenförmiger Anordnung ansprechbar (Bild 6)

allerdings verband Peter Shor 1994 ein tiefgehendes Verständnis für Kryptografie (RSA-Verschlüsselung) mit einer neuen mathematischen Formulierung.

Plötzlich wusste die Welt: Mit ausreichend großen Quantencomputern können wir richtig wichtige Verschlüsselungsschlüssel aufbrechen ... und es begann ein Wettrennen, diese Quantencomputer wirklich zu bauen. 1998 war der erste fertiggestellt (wie wir eingangs mit Kernmagnetresonanz und Molekülen betrachtet haben), und 2001 wurden die ersten kleinen Zahlen in Primfaktoren zerlegt.

Heutzutage sind die hardwareseitigen Herausforderungen, um große Chips zu erreichen, bekannt und werden weltweit bearbeitet. Auf der Softwareseite gibt es sehr viel Spielraum, um neue Subroutinen für neue Problemlösungen auszutesten.

Bleiben Sie neugierig und kreativ, und lesen Sie Dokumentation und Papers! ■

[1] Isaac L. Chuang, Neil Gershenfeld, Mark Kubinec, *Experimental Implementation of Fast Quantum Searching*, www.dotnetpro.de/SL2502Quantum1

[7] Quantum Internet Alliance, <https://quantuminternetalliance.org>

[8] McKinsey, *Steady progress in approaching the quantum advantage*, www.dotnetpro.de/SL2502Quantum5

[9] DLR Quantencomputing-Initiative, <https://qci.dlr.de/projekte/>

[10] Universität Innsbruck, *Deutschland nimmt Quantencomputer in Betrieb*, www.dotnetpro.de/SL2502Quantum6

● Nutzen der probabilistischen Eigenschaften

Der Shor-Algorithmus bricht Zahlen nicht durch Division in Primfaktoren, sondern durch das Finden von Periodizität. Im Jahr 1996 erkannte Lov Grover dann die Nutzung der probabilistischen Eigenschaften von Quantenalgorithmen, um die lineare Datenbanksuche (~1 E-Mail in 100 E-Mails finden) quadratisch zu beschleunigen (~1 E-Mail in 10 E-Mails finden).

- [11] Jülich Supercomputing Centre (JSC), www.dotnetpro.de/SL2502Quantum7
- [12] Deutsche Bundesregierung, Deutschland kann Quanten-Computer, www.dotnetpro.de/SL2502Quantum8
- [13] IBM Quantum Learning, Composer, <https://quantum.ibm.com/composer/>
- [14] IBM Quantum Platform, Quantum processing units, www.dotnetpro.de/SL2502Quantum9
- [15] IBM Quantum Documentation, Hardware considerations, www.dotnetpro.de/SL2502Quantum10
- [16] The Quantum Insider, Top 5 Quantum Programming Languages 2024, www.dotnetpro.de/SL2502Quantum11
- [17] Daniel Basler, Q# für Einsteiger, dotnetpro 2/2025, Seite 38 ff., www.dotnetpro.de/A2502QSharp
- [18] Daniel Basler, Marke Eigenbau, dotnetpro 2/2025, Seite 32 ff., www.dotnetpro.de/A2502Quantensimulator
- [19] Silq, Comparison to Q#, <https://silq.ethz.ch/comparison>
- [20] IBM Cloud, FAQ zu IBM Cloud Qiskit Runtime, www.dotnetpro.de/SL2502Quantum12
- [21] IBM Quantum Documentation, Transpiler, www.dotnetpro.de/SL2502Quantum13
- [22] IBM Quantum Documentation, PassManager, www.dotnetpro.de/SL2502Quantum14
- [23] Google Quantum AI, Circuit Transformers, www.dotnetpro.de/SL2502Quantum15
- [24] IBM Quantum Documentation, Transpiler stages, www.dotnetpro.de/SL2502Quantum16
- [25] Felipe Ferreira, José Campos, An exploratory study on the usage of quantum programming languages, www.dotnetpro.de/SL2502Quantum17
- [26] Qiskit Textbook auf GitHub, Quantum Fourier Transform, www.dotnetpro.de/SL2502Quantum18
- [27] Google Quantum AI, Quantum variational algorithm, www.dotnetpro.de/SL2502Quantum19
- [28] Shor's Algorithm for Integer Factorization auf GitHub (Achtung: Alte Qiskit-Version!), www.dotnetpro.de/SL2502Quantum20
- [29] Microsoft Learn, Theorie des Grover-Suchalgorithmus, www.dotnetpro.de/SL2502Quantum21
- [30] Qrisp developers, QUBO as a QAOA Problem Instance, www.dotnetpro.de/SL2502Quantum22
- [31] IBM Quantum Documentation, Introduction to Qiskit, <https://docs.quantum.ibm.com/guides>

**Alexandra Waldherr**

forscht nach Studien an den Innsbrucker Universitäten an den Max-Planck-Instituten in Tübingen. Sie ist Fellow am Stanford Center for Responsible Quantum Technology für Quantum Hardware, Simulation & Education und Mitgründerin der Quantum Society Austria (QSA).

dnpCode

A2502Quantum

DEV ACADEMY

HANDS-ON-WORKSHOPS UND
WEITERBILDUNG FÜR SOFTWARE-
ENTWICKLER UND -ARCHITEKTEN

2 TAGE

AI-driven Software Development mit GPT und Co.

TRAINING!

Was wird behandelt

- GPT-Modelle und ihre Unterschiede
- AI-Tools für Software-Entwickler
- Grundlagen des Prompt Engineering
- Verwenden der OpenAI API
- AI Search und Vektordatenbanken
- Eigene LLMs bereitstellen mit Azure OpenAI
- Entwicklung eigener Assistenten
- Anbindung Backendsysteme



Jörg Neumann,
MVP OpenAI
Trainer

QUANTUM COMPUTING IN DER CLOUD

Wegweisende Symbiose

Leistungsstarke Quantum-Computing-Rechenressourcen in der Cloud werden die Art und Weise revolutionieren, wie wir Probleme angehen und Lösungen entwickeln.

Quantum Computing als eine neue und faszinierende Disziplin innerhalb der Informatik hat in den letzten Jahren zunehmend an Bedeutung gewonnen und die Aufmerksamkeit von Forschern, Unternehmen und der breiten Öffentlichkeit auf sich gezogen. Dieser Bereich repräsentiert nicht nur eine Evolution, sondern vielmehr eine Revolution in der Art und Weise, wie man klassische Probleme der Informatik angeht und löst. Die Entwicklungen im Quantum Computing könnten sogar einen Paradigmenwechsel für die gesamte IT-Branche bedeuten. In diesem Artikel soll zunächst ein kurzer Blick auf die Hintergründe des Quantum Computing geworfen werden, um die Grundlagen dieses aufstrebenden Feldes zu verstehen und die bahnbrechenden Möglichkeiten zu entdecken, die es bietet.

Grundbausteine von Quantum Computing

Zu den fundamentalen Einheiten im Bereich des Quantumcomputing gehören Quantenbits (kurz Qubits). Die Qubits unterscheiden sich grundlegend von Bits in einem klassischen Computer. Um die einzigartige Natur von Qubits zu verstehen, ist es entscheidend, einen Einblick in die Welt der Quantenmechanik zu gewinnen, da Qubits auf den Prinzipien dieser physikalischen Theorie beruhen.

Im Gegensatz zu herkömmlichen Bits, die in einem klassischen Computer entweder den Zustand 0 oder den Zustand 1 annehmen können, können Qubits aufgrund eines Phänomens namens Quantenüberlagerung (englisch: Quantum Su-

● Quantenüberlagerung

Quantenüberlagerung (englisch: Quantum Superposition) ist ein fundamentales Prinzip der Quantenmechanik, das besagt, dass Quantenobjekte wie zum Beispiel Photonen, Elektronen, Ionen und Atome gleichzeitig in mehreren Zuständen existieren können, bis sie gemessen werden. Dies steht im Kontrast zur klassischen Physik, in der ein System sich in einem eindeutigen Zustand befindet. Es bedeutet, dass Quantenobjekte wie Qubits (Quantenbits) in einem Quantencomputer nicht nur den Zustand 0 oder 1 aufweisen, sondern eine lineare Kombination beider Zustände.

Quantenüberlagerungsphänomene werden für parallele Berechnungen in Quantencomputern, für sichere Kommunikation in Quantennetzwerken und für komplexe Probleme in Quantenalgorithmen verwendet.

perposition, siehe auch Kasten **Quantenüberlagerung**) nicht nur 0 oder 1, sondern auch gleichzeitig eine Kombination dieser beiden Zustände annehmen. Dies bedeutet, dass ein Qubit nicht eine klare Identität hat, sondern sich in einem Zustand der Superposition befindet (siehe **Bild 1**).

Der Qubit-Status in Quantenüberlagerung wird mathematisch als normalisierter komplexer Zustandsvektor dargestellt:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \beta \end{bmatrix} = \alpha \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Dabei gilt: $|\alpha|^2 + |\beta|^2 = 1$

Die Ausdrücke α und β stellen hier als komplexe Zahlen sogenannte Wahrscheinlichkeitsamplituden für die jeweiligen Qubit-Zustände 0 und 1 dar. Werte von α und β geben an, wie „nahe“ das Qubit jeweils am Zustand 0 oder 1 ist.

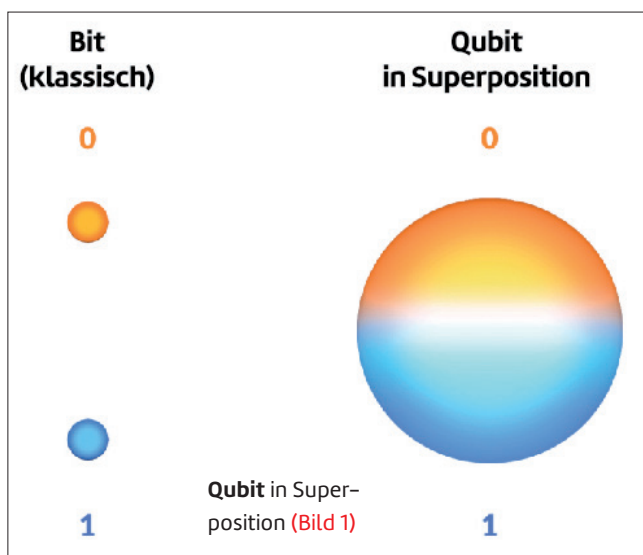
$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ ist ein Vektor, der ein Qubit im Zustand 0 repräsentiert, und $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ist ein Vektor, der für ein Qubit im Zustand 1 steht. Das sind sogenannte Basiszustände, die in der für die Quantenmechanik üblichen Dirak-Notation [1] wie folgt geschrieben werden:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Damit können alle möglichen Quantenzustände eines Qubits noch einfacher ausgedrückt werden:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha |0\rangle + \beta |1\rangle$$

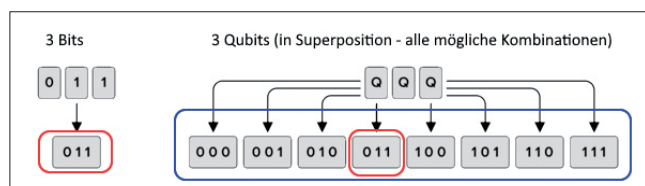
oder für einen beliebigen Ψ -Vektor:



$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

Ein Qubit kann somit eine Kombination aus 0 und 1 sein, und dies nicht als diskrete Werte, sondern als kontinuierliche Wahrscheinlichkeitsverteilung. Man kann sich das bildlich auch als Gedankenexperiment mit der berühmten Schrödinger-Katze [2] vorstellen, bei dem die Katze gleichzeitig „tot“ und „lebendig“ ist, oder anhand einer sich in der Luft drehenden Münze, bei der die beiden Seiten der rotierenden Münze „verschmelzen“, bis zu dem Zeitpunkt, an dem man die Kiste mit der Katze öffnet beziehungsweise die Münze in der Hand landet und aufgedeckt wird. Damit wird ein endgültiger Zustand „gemessen“ – und die Katze für „tot“ oder „lebendig“ erklärt beziehungsweise Kopf oder Zahl einer Münze festgelegt, was jeweils dem Zustand 0 oder 1 entsprechen kann. Eine verbreitete Fehlvorstellung liegt dabei in der Sichtweise, ein Qubit wäre immer im Zustand 0 oder im Zustand 1 (es sei bloß unbekannt), und man erfahre erst bei einer Messung, in welchem Zustand es sich befindet. Das ist aber nicht korrekt. Ein Qubit in der Superposition befindet sich in einer linearen Kombination der Zustände 0 und 1. Bei der Messung wird das Qubit dazu gezwungen, sich für eine der möglichen Varianten zu „entscheiden“. Anders ausgedrückt: Die Messung eines Qubits ist ein drastischer Prozess, der ursprünglichen Superposition-Zustand des gemessenen Qubits zerstört, und dann kollabiert es zu einem der möglichen Zustände.

Im Superposition-Zustand ist ein Qubit also in der Lage, alle möglichen Zustände parallel zu repräsentieren. Diese Pa-



Qubit-Daten-Repräsentation (Bild 2)

rallelität ermöglicht es Quantencomputern, eine enorme Anzahl von Berechnungen gleichzeitig durchzuführen, was bei bestimmten Problemstellungen zu erheblichen Geschwindigkeitsvorteilen führen kann. Dieser Effekt ist in Bild 2 ersichtlich – während bei „klassischen“ drei Bits für alle möglichen Kombinationen $3 \times 8 = 3 \times 2^3 = 24$ Bits nötig wären, kann man das Gleiche mit nur drei Qubits in Superposition abbilden. Weiterdenkend wären statt vier überlagerten Qubits schon $4 \times 2^4 = 64$ Bits nötig und so weiter. Wie man an diesen einfachen Beispielen sieht, ist ein Qubit-Rechner 2^N -fach effektiver als ein klassischer.

Während die Leistung eines klassischen Computers linear wächst, verläuft bei einem Quantencomputer die Leistungssteigerung exponentiell – das bedeutet, mit nur einem einzigen zusätzlichen Qubit verdoppelt man die Berechnungsmöglichkeit! Diese Potenzpower macht einige Aufgaben mit einem Quantencomputer lösbar, die auf einem klassischen Weg sonst weiterhin ungelöst blieben. Zum Beispiel kann es

Quantenverschränkung

Quantenverschränkung (englisch: Quantum Entanglement) ist ein fundamentales Phänomen der Quantenmechanik, bei dem ein System mit mehreren Elementarteilchen als Ganzes betrachtet einen wohldefinierten Zustand einnimmt, ohne dass man jedem der Teilsysteme einen eigenen wohldefinierten Zustand zuordnen kann. Dabei sind zwei oder mehr Teilchen so miteinander verbunden, dass der Zustand eines Teilchens sofort den Zustand des anderen beeinflusst, unabhängig von der Entfernung zwischen diesen Teilchen.

im Go-Spiel [3] bei einem 19×19 -Brett $2,08 \times 10^{170}$ oder circa 2^{565} verschiedene Stellungen geben. Das ist eine unvorstellbar große Zahl, wenn man bedenkt, dass es im beobachtbaren Universum mit geschätzt 2 Billionen Galaxien, von denen jede einzelne noch Hunderte Milliarden von Sternen enthält, „nur“ zwischen 10^{84} (2^{278}) und 10^{89} (2^{296}) Atome gibt! Also wäre es offensichtlich unmöglich, alle mögliche Go-Stellungen in klassischer binärer Form zu berechnen – auch wenn man pro Bit nur ein Atom reserviert, reicht das ganze (bekannte) Universum schon lange nicht! Mit circa 600 kohärenten verschränkten Quantenbits wäre die Berechnung aller Go-Stellungen lösbar, für eine binäre Abbildung aller Atome des Universums hätte sogar die Hälfte davon ausgereicht!

Ein einzelnes Quantenbit bedeutet keinen großen Unterschied gegenüber einem klassischen Bit und findet kaum praktische Anwendung. Mit jedem weiteren hinzugefügten Qubit verdoppeln sich jedoch Leistung und Möglichkeiten eines Quantencomputers. Man muss also einen Weg finden, mehrere Quantenbits „zusammenzukleben“. Dafür wird ein weiteres Phänomen der Quanten-Welt verwendet: die Quantenverschränkung (siehe Kasten [Quantenverschränkung](#)).

Die Verschränkung der Qubits spielt vor allem in Quantencomputer-Berechnungen eine zentrale Rolle – einerseits durch Verschränkung vieler Qubits im Verlauf der Rechnung mit Quantenalgorithmen, andererseits im Verfahren zur ►

Dekohärenz

Dekohärenz [4] ist ein Phänomen der Quantenphysik, das teilweise oder vollständig zu einem Verlust der Kohärenz oder der quantenmechanischen Superposition zwischen Teilchen führt. Durch Wechselwirkungen mit ihrer Umgebung, beispielsweise durch externe Einflüsse wie Wärme oder Strahlung oder durch Messungen, verlieren Quantenzustände ihre speziellen Eigenschaften und werden „klassischer“ Natur. Dieser Prozess führt dazu, dass die einzigartigen, nichtklassischen Merkmale von Quantensystemen allmählich verschwinden, was eine Herausforderung für die Aufrechterhaltung quantenmechanischer Eigenschaften in realen Umgebungen darstellt.

Fehlerkorrektur, die nötig ist, um die Quantenbits vor Dekohärenz (siehe Kasten **Dekohärenz** und [4]) zu schützen.

Außerdem bildet die Quantenverschränkung eine Grundlage für Quantenkommunikationsprotokolle wie Quantenschlüsselverteilung [5] (englisch: Quantum Key Distribution, QKD). Ein weiteres faszinierendes Anwendungsgebiet von Quantenverschränkung ist die Quantenteleportation [6], bei der der Zustand eines Teilchens von einem Ort zum anderen übertragen wird, ohne dass das Teilchen selbst den Raum dazwischen durchläuft.

Wenn die Quantenzustände nicht als individuelle Qubit-Zustände geschrieben werden können, spricht man von nicht separierbaren Quantensystemen, auch als verschränkte Systeme bezeichnet. Wenn ein Zustand als Produktzustand der einzelnen Teilsysteme geschrieben werden kann, ist dieser Zustand nicht verschränkt. Verschränkung ist eine quantenmechanische Korrelation, die sich deutlich von klassischen Korrelationen unterscheidet. Bei Verschränkung ist der Zustand der Teilsysteme nicht festgelegt, und es kann nur über die Wahrscheinlichkeiten der Ergebnisse gesprochen werden. Das globale System muss als Einheit betrachtet werden.

Die Quantenverschränkung ist ein entscheidender Baustein im Quantencomputer-Konzept. Sie ermöglicht es, die Qubits so zu verknüpfen, dass sie sich nicht wie Einzelteile, sondern wie ein großes, komplexeres System verhalten. In verschränkten Systemen ändert die Messung eines Qubits den Zustand der anderen Qubits und gibt uns Informationen über ihren Zustand. Man betrachte zum Beispiel zwei überlagerte Qubits A und B (in Superposition), sodass der Zustand des globalen Quantensystems folgendermaßen wäre:

$$|\psi\rangle_{AB} = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle$$

In einem solchen Zustand wären nur zwei Ergebnisse möglich, wenn man den Zustand beider Qubits in der Standardbasis misst: $|00\rangle$ und $|11\rangle$, wobei jedes Ergebnis die gleiche Wahrscheinlichkeit von $\frac{1}{2}$ hätte. Dabei wäre es unmöglich

● Quanteninterferenz

Quanteninterferenz ist ein Phänomen in der Quantenmechanik, das aus der wellenartigen Natur von quantenmechanischen Teilchen wie Elektronen oder Photonen resultiert. Wenn ein Teilchen in einer Überlagerung (Superposition) mehrerer Zustände ist, können diese Zustände miteinander interferieren und zu konstruktiver (Verstärkung) oder destruktiver Interferenz (Unterdrückung) führen. Diese Interferenz kann spezifische Muster von Ergebnissen bei der Messung des Teilchens erzeugen. In der Quanteninformatik wird Interferenz auf verschiedene Weisen genutzt, um Quantenzustände zu manipulieren, zu kontrollieren und somit Rechenaufgaben auszuführen. Durch Anwendung von Quantengattern [12], die Überlagerungen von Qubits erzeugen, und durch Kontrolle der relativen Phasen der Zustände kann Interferenz verwendet werden, um bestimmte Ergebnisse zu verstärken und andere zu unterdrücken.

● Logisches Qubit

Ein logisches Qubit ist eine abstrakte, fehlertolerante Einheit in der Quanteninformatik, die aus mehreren physischen Qubits konstruiert wird. Im Gegensatz zu physischen Qubits, die anfällig für Fehler und Dekohärenz sind, wird ein logisches Qubit durch Fehlerkorrekturtechniken geschützt, um zuverlässige Quanteninformationen zu speichern. Die Verwendung von Fehlerkorrekturcodes ermöglicht es logischen Qubits, stabilere und länger anhaltende Quantenzustände aufrechtzuerhalten, was für die Realisierung von zuverlässigen Quantencomputern von entscheidender Bedeutung ist.

(Wahrscheinlichkeit gleich 0), die Ergebnisse $|01\rangle$ und $|10\rangle$ zu bekommen. Wenn man nun das erste Qubit A misst und feststellt, dass es im Zustand $|0\rangle$ ist, kann man sicher sein, dass das zweite Qubit B sich ebenfalls im Zustand $|0\rangle$ befindet, ohne es zu messen zu müssen! Die Messergebnisse in diesem Fall korrelieren zueinander, und die Qubits A und B sind verschränkt. Diese Verschränkungseigenschaft wird sehr intensiv in vielen Quantenalgorithmen genutzt. Die Quantenverschränkung ermöglicht es also, dass die Qubits in einer gemeinsamen Informationseinheit arbeiten, wodurch sie kollektiv Informationen verarbeiten können.

Ein weiteres wichtiges Konzept der Quantenmechanik, das in Quantencomputern intensiv benutzt wird ist die Quanteninterferenz (siehe Kasten **Quanteninterferenz**). Interferenz bildet die Grundlage vieler Quantenalgorithmen und kann zu erheblichen Beschleunigungen in der Berechnung führen. Ein Beispiel für eine solche Anwendung ist der Grover-Algorithmus [7], der für die unstrukturierte Suche verwendet wird. Ein weiteres Beispiel für die Nutzung von Interferenz in einem Algorithmus ist die Quanten-Fourier-Transformation (QFT) [8], die eine effiziente Berechnung bestimmter mathematischer Funktionen ermöglicht. Quanteninterferenz wird auch in der Quantenphasenschätzung verwendet, einer Technik, die in verschiedenen Quantenalgorithmen zur Schätzung der Phase eines Quantenzustands eingesetzt wird. Die Quantenphasenschätzung umfasst die Interferenz zwischen verschiedenen Quantenzuständen, die durch Anwendung von Quantengattern gesteuert und durch Messung der resultierenden Wahrscheinlichkeitsverteilung gemessen werden kann. Darüber hinaus ist Interferenz neben dem logischen Qubit (siehe Kasten **Logisches Qubit**) ein Schlüsselkonzept in der Quantenfehlerkorrektur, die notwendig ist, um Quanteninformation vor den möglichen Fehlern zu schützen. Durch Messung des Interferenzmusters von Qubits können Quantencomputer Fehler erkennen und korrigieren, die während der Berechnung aufgetreten sein könnten.

In einigen Quantencomputern kann sich ein Elementarteilchen nicht in zwei, sondern in drei oder noch mehr überlagerten Zuständen befinden. Dabei spricht man jeweils von Qutrit [9] beziehungsweise Qudit [10] (siehe Kasten **Qutrit und Qudit**) als mehrdimensionalen Quanteninformations-Baustei-

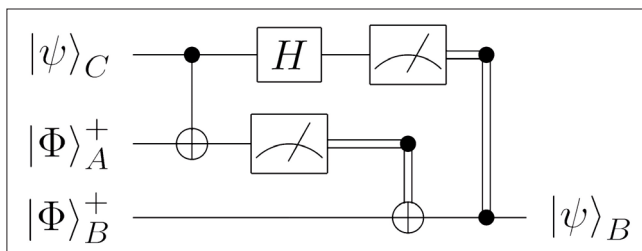
nen. Qutrits und Qudits bieten eine höhere Kapazität für Quantenparallelismus. Anwendungen von Qudits finden sich in verschiedenen Bereichen, darunter die Fourier-Transformation, Quantenverschlüsselung und spezifische Quantenalgorithmien. Die Implementierung von Qudits bringt jedoch technische Herausforderungen mit sich, insbesondere in Bezug auf die Kontrolle und Messung von Zuständen in höheren Dimensionen.

Rechenmodelle im Quantencomputing

Quantencomputing umfasst verschiedene Rechenmodelle, die auf unterschiedlichen physikalischen Prinzipien beruhen. Jedes Modell hat seine eigenen spezifischen Stärken und Schwächen sowie spezifische Anwendungsbereiche. In den nachfolgenden Abschnitten sollen die wichtigsten Quantencomputing-Modelle vorgestellt werden.

Schaltkreis-/Quantum-Gate-Modell

Ein maßgeblicher Akteur im Bereich des Quantencomputing ist das Schaltkreis-Modell [11] (englisch: Quantum Circuit), auch als Quantum-Gate Computing bekannt. Dieses Modell basiert auf der Anwendung von Quantengattern [12] (englisch: Quantum logic gate), um Qubits zu manipulieren und komplexe Berechnungen durchzuführen. In diesem Kontext werden Quantenalgorithmien als Schaltkreise von Quantengattern repräsentiert, wodurch fundamentale Quantenprinzi-



Schaltkreis für Quantenteleportation (Bild 3)

● Qutrit und Qudit

Qutrit [9] und Qudit sind Erweiterungen des Konzepts von Qubits in der Quanteninformatik [10]. Während Qubits binäre Einheiten von Quanteninformation sind, die die Zustände 0, 1 oder eine Superposition von beiden zweidimensional repräsentieren, können Qutrits und Qudits in einem höherdimensionalen Raum existieren. Anders ausgedrückt: Während Qubits zwei mögliche Zustände haben, können Qutrits drei und Qudits noch mehr Zustände in einem Quantensystem darstellen. Die höheren Dimensionen der Qutrits beziehungsweise Qudits eröffnen erweiterte Möglichkeiten für die Speicherung und Verarbeitung von Quanteninformationen. Die Quantenüberlegungen und Prinzipien, die für Qubits gelten, können auf Qutrits und Qudits erweitert werden, wodurch komplexere und leistungsfähigere Quantenalgorithmien und -protokolle möglich wären.

● Quantenteleportation

Die Quantenteleportation [6] ist ein Prozess, der den quantenmechanischen Zustand eines Quantensystems, wie beispielsweise eines Photons oder eines Qubits (Quelle), auf ein anderes Quantensystem (Ziel) überträgt. Während dieser Übertragung wird der Zustand der Quelle verändert. Das Verfahren bedient sich dabei zweier Kanäle: eines Quantenkanals und eines klassischen Kanals. Als Quantenkanal dient ein Paar von verschränkten Teilchen. Die Quantenteleportation nutzt die Eigenschaften der Quantenverschränkung sowie beide Kanäle, um den Zustand von der Quelle zum Ziel zu übertragen, ohne dabei eine Messung vorzunehmen.

Die Quantenteleportation spielt eine entscheidende Rolle beim Aufbau von Quantennetzwerken. Es ist wichtig zu betonen, dass der Begriff „Teleportation“ in diesem Kontext nicht mit der Teleportation aus der Science-Fiction gleichzusetzen ist. In der Realität setzt die Quantenteleportation voraus, dass am Zielort bereits ein vorbereitetes Zielsystem vorhanden ist, auf das der Zustand der Quelle übertragen wird.

pien wie Superposition und Verschränkung genutzt werden. Dieser Ansatz wird zum Beispiel in dem berühmten Shor-Algorithmus [13] für die Faktorisierung von Primzahlen (Entschlüsselung) und dem Grover-Suchalgorithmus [7] angewandt. Gate-basierte Quantencomputer können ähnlich wie aktuelle klassische digitale Computer betrachtet werden, bei denen eine Reihe von Transformationen (Logikgatter) durchgeführt werden, um ein bestimmtes Ergebnis zu erhalten.

Ein Quantenschaltkreis ist ein Modell für Quantenberechnungen, ähnlich den klassischen Schaltkreisen, bei dem eine Berechnung eine Abfolge von Quantengattern, Messungen, Initialisierungen von Qubits auf bekannte Werte und möglicherweise anderen Aktionen ist. Die Schaltkreise werden so geschrieben, dass die horizontale Achse die Zeit darstellt, beginnend auf der linken Seite und endend auf der rechten Seite. Horizontale Linien repräsentieren Qubits, doppelte Linien stehen für klassische Bits. Die Elemente, die durch diese Linien verbunden sind, sind Operationen, die an den Qubits durchgeführt werden, wie Messungen oder Gatter-Transformationen. Diese Linien definieren die Abfolge der Ereignisse und sind normalerweise keine „physischen“ Leitungen im eigentlichen Sinne.

Zur Veranschaulichung ist in Bild 3 ein Schaltkreis für Quantenteleportation (siehe Kasten **Quantenteleportation** und [6]) dargestellt. Zu den Vorteilen des Quanten-Schaltkreis-Rechenmodells gehören vor allem:

- **Klarheit:** Das Schaltkreis-Modell bietet eine klare und intuitive Darstellung von Quantenalgorithmien. Die Verwendung von Quantengattern ermöglicht eine präzise Repräsentation von Schritten in einem Algorithmus.
- **Verbreitung:** Das Schaltkreis-Modell ist vielseitig einsetzbar und eignet sich für verschiedene Quantenalgorithmien, zum Beispiel den Shor-Algorithmus [13] oder den Grover- ▶

Suchalgorithmus [7], und es deckt eine breite Palette von Anwendungen ab.

- **Quantenparallelismus:** Das Schaltkreis-Modell nutzt Quantenparallelismus effektiv aus, wodurch mehrere Berechnungen gleichzeitig durchgeführt werden können. Dies kann zu erheblichen Geschwindigkeitsvorteilen bei bestimmten Problemstellungen führen.
- **Programmierbarkeit:** Das Schaltkreis-Modell ermöglicht die präzise Programmierung von Quantenoperationen. Durch die Anwendung von Quantengattern kann der Algorithmus gezielt gesteuert und angepasst werden.

Wenn man die Nachteile von Schaltkreis-Rechenmodellen betrachtet, sind folgende Herausforderungen festzustellen:

- **Fehleranfälligkeit:** Schaltkreise mit vielen Qubits sind anfällig für Fehler, insbesondere durch Dekohärenz [4] und Rauschen in der Quantenhardware. Dies erfordert komplexe Fehlerkorrekturmechanismen.
- **Komplexe Hardware:** Die Umsetzung von Schaltkreis-Modellen erfordert hochentwickelte Quantenhardware. Die Realisierung von stabilen Quantengattern mit einer großen Anzahl von Qubits stellt technologische Herausforderungen dar (Qubit-Skalierung, Störeinflüsse et cetera).
- **Schaltkreis-Komplexität:** Bei längeren Schaltkreisen mit vielen Quantengattern kann die Komplexität und damit die Anzahl der benötigten Qubits schnell ansteigen. Dies erhöht die Anforderungen an die Hardware und erschwert die Fehlerkorrektur.
- **Fragilität:** Die Leistungsfähigkeit des Schaltkreis-Modells kann durch Umgebungsstörungen (zum Beispiel Strahlung, Wärme et cetera) und physikalische Einschränkungen (zum Beispiel die Erfordernis von Tieftkühlung für Supraleitung) beeinträchtigt werden, was seine Robustheit gegenüber externen Einflüssen begrenzt.

Technologisch führende Unternehmen wie Google, IBM, Intel, Honeywell und viele mehr investieren intensiv in diesen Ansatz, um die Grenzen der Leistungsfähigkeit von Quantum-Gate Computing weiter zu erforschen, und haben bereits einige Schaltkreis-basierte Quantenprozessoren (QPUs) hergestellt. Manche davon stehen via IBM, AWS, Azure und Google Cloud für benutzerspezifische Quanten-Schaltkreisalgorithmen bereits zur Verfügung.

Analoge Quantencomputing-Modelle

Im Unterschied zu diskreten Quanten-Rechenmodellen wie dem Schaltkreis-/Gate-basierten Modell gibt es analoge Quanten-Rechenmodelle, die sich auf die Art und Weise beziehen, wie die Informationen in einem Quantencomputer dargestellt und verarbeitet werden. Hier sind die wichtigsten Unterscheidungsmerkmale:

- **Kontinuierliche Zustände:** Im analogen Ansatz werden physikalische Quantenzustände kontinuierlich dargestellt und manipuliert. Die Veränderungen der Zustände erfolgen stufenlos und können als analog zu den kontinuierlichen Werten in klassischen analogen Systemen betrachtet werden.

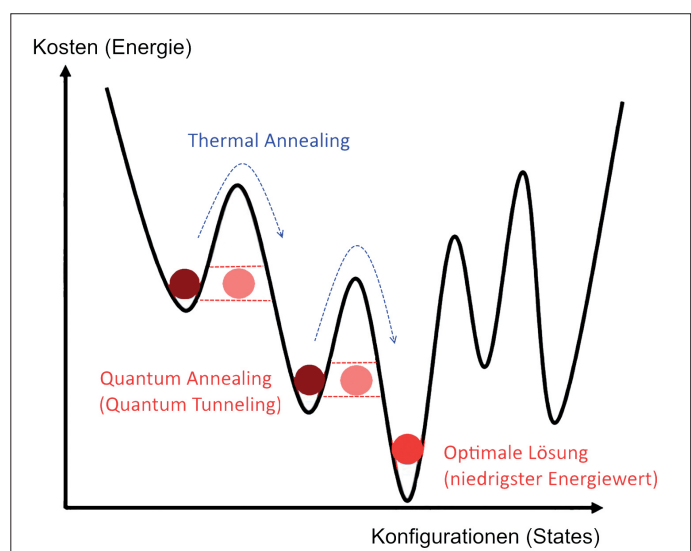
- **Kontinuierliche Transformationen:** Analoge Quantencomputer nutzen kontinuierliche physikalische Transformationen, um Informationen zu repräsentieren und zu verarbeiten. Dies geschieht oft über Prozesse wie adiabatisches Quantencomputing [14] oder Quantum Annealing [15].

Analoge Quanten-Rechenmodelle haben ihre eigenen spezifischen Merkmale. So sind für adiabatisches Quantencomputing die folgenden Punkte hervorhebenswert:

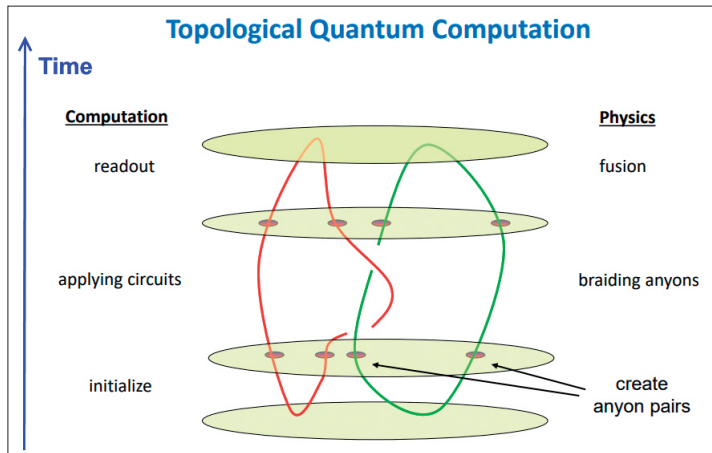
- **Prinzip:** Sie basieren auf dem adiabatischen Theorem, das den langsamen Übergang eines Systems zwischen Zuständen beschreibt.
- **Funktionsweise:** Durch adiabatische Evolution wird der Quantenzustand eines Systems kontinuierlich verändert, um eine Lösung für ein Optimierungsproblem zu finden.
- **Anwendungen:** Gut geeignet für spezifische Optimierungsprobleme wie das Handlungsreisenden-Problem [16] und das Maximum-Schnitt-Problem [17].
- **Vorteile:** Robust gegenüber bestimmten Fehlern und natürlich anwendbar für bestimmte physikalische Systeme.
- **Herausforderungen:** Begrenzte Vielseitigkeit für allgemeine Berechnungen und technologisch anspruchsvollere Skalierung.

Beim Annealing-Quantencomputing sind folgende Punkte erwähnenswert:

- **Prinzip:** Verwendung von Quanten-Annealing, einem Prozess, der darauf abzielt, das globale Energieminimum eines Systems zu finden.
- **Funktionsweise:** Das Quanten-Annealing-Verfahren [15] sucht nach der minimalen Energiekonfiguration eines Systems, was als Ergebnis für ein relevantes Optimierungsproblem zu betrachten ist. Quantum Annealing (rote Linie in Bild 4) durchläuft Energielandschaften effizient, indem es Quanten-Tunneling nutzt, um das globale Minimum zu finden. Quantum Annealing bietet einen erheblichen Leistungsvorteil gegenüber simuliertem Annealing (blaue Linie



Lösungsfindung (globales Minimum) mit Quantum Annealing (Bild 4)



Topologisches Quantencomputing-Modell (Bild 5)

in Bild 4) und ermöglicht die Lösung komplexer Optimierungsprobleme, die auf andere Weise nur schwer oder gar nicht gelöst werden konnten.

- **Anwendungen:** Effektiv für bestimmte Optimierungsprobleme, insbesondere in Bezug auf das Finden globaler Minima.
- **Vorteile:** Effektiv für bestimmte Optimierungsprobleme, kann in bestimmten Anwendungen Gate-basierte Ansätze ergänzen.
- **Herausforderungen:** Begrenzte Vielseitigkeit, hohe Anforderungen an das Design von Quantenhardware.

Technologisch ist die kanadische Firma D-Wave Systems [18] im Bereich Quantum Annealing am weitesten fortgeschritten. Im Mai 2011 wurde „D-Wave One“ als „der welterste kommerziell verfügbare Quantencomputer“ vorgestellt. Heute kann jeder via D-Waves Cloud-Platform Leap [19] eigene Quantum-Lösungen bauen.

Ein weiteres Beispiel für analoge Quantencomputing-Rechenmodelle ist Quantencomputing mit kontinuierlichen Variablen (englisch: Continuous-variable quantum computing [20], [21]). Quantencomputing mit kontinuierlichen Variablen basiert auf der Nutzung von Quantenzuständen, die nicht diskret, sondern kontinuierlich sind. Im Gegensatz zu herkömmlichen zweidimensionalen Qubits beziehungsweise drei- oder mehrdimensionalen Qutrits/Qudits, die diskrete Zustände aufweisen, verwenden diese Systeme kontinuierliche Größen wie Phasen- oder Amplitudenmodulationen. Die Vorteile dieses Ansatzes liegen unter anderem in der hohen Redundanz, die eine robuste Fehlerkorrektur ermöglicht, sowie in der effizienten Codierung von Informationen, die komplexe Rechenoperationen erleichtern kann. Zudem könnten Modelle mit kontinuierlichen Variablen in einigen Fällen weniger Hardwarekomponenten erfordern. Es gibt jedoch auch Herausforderungen in diesem Feld wie zum Beispiel präzise Kontrolle und Messung kontinuierlicher Variablen oder Empfindlichkeit gegenüber externen Störungen. Quantencomputing mit kontinuierlichen Variablen ist ein vielversprechendes Forschungsfeld, das bereits verschiedene praktische [22] sowie experimentelle Implementierungsansätze gefunden hat (siehe Kapitel 5 in [20]).

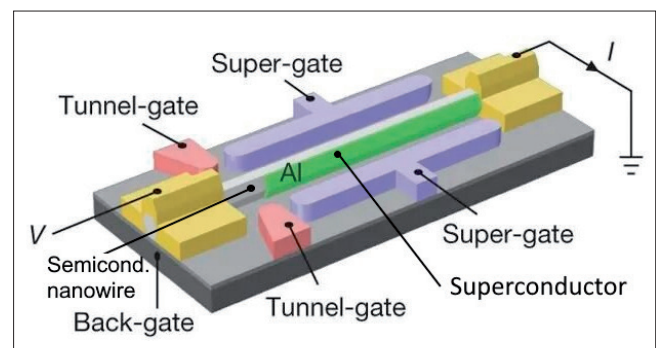
Analog-Quantencomputer sind auch leistungsstark bei der Quantensimulation, da sie flexible Anregungs- und Antriebsparameter bieten. Ihre Vielseitigkeit erstreckt sich über verschiedene komplexe Aufgaben in den Bereichen Optimierung, maschinelles Lernen und Simulation von Quantensystemen.

Und schließlich ist es noch an dieser Stelle zu bemerken, dass „digitale“ Quantencomputer (zum Beispiel Gatter-/Schaltkreis-basierte) im Unterschied zu analogen Modellen aufgrund ihrer diskreten Natur vielseitiger einsetzbar sind. Sie können eine breitere Palette von Quantenalgorithmen implementieren, darunter auch solche, die für klassische Computer schwer zu simulieren sind. Die Unterscheidung zwischen „digitalen“ (diskreten) und „analogen“ Ansätzen spiegelt die verschiedenen Strategien wider, die bei der Realisierung von Quantencomputern verfolgt werden und für die sich dann jeweils eigene Vor- und Nachteile in Bezug auf Anwendbarkeit und Implementierung ergeben.

Experimentelle Rechenmodelle im Quantencomputing

Quantencomputing hat sich zu einem faszinierenden Forschungsbereich entwickelt, der verschiedene experimentelle Rechenmodelle umfasst. Während gängige Gate-basierte und Annealing-Modelle bereits ihre technologische Realisierung und kommerzielle Anwendung gefunden haben, eröffnen experimentelle Ansätze weiterer Quanten-Rechenmodelle neue Perspektiven für die Realisierung von Quantencomputern. Im Folgenden werfen wir einen Blick auf einige dieser experimentellen Modelle, um ihre Besonderheiten sowie Vor- und Nachteile zu erkunden.

Der Ansatz des topologischen Quantencomputing (englisch: Topological Quantum Computing) basiert auf topologischen Quantenzuständen und -operationen, die in der Theorie widerstandsfähiger gegenüber Fehlerquellen sein könnten. Dabei verwendet ein topologischer Quantencomputer topologische Zustände von Qubits, um Quanteninformationen zu speichern und zu verarbeiten. Er nutzt Quasiteilchen in zweidimensionalen Systemen, auch als Anyons [23] bezeichnet. Die Weltlinien [24] (eine zeitartige Kurve in einer mehrdimensionalen, relativistischen Raumzeit) von Anyons verlaufen umeinander und bilden sogenannte Braids [25] (Flechten) ►



Struktur des topologischen Qubit-Geräts (Bild 6)

in einem dreidimensionalen Raum-Zeit-Gefüge (also einer zeitlichen plus zwei räumlicher Dimensionen, siehe [Bild 5](#)).

Kombinationen dieser Flechten fungieren als die Logikgatter, die einen Computer auf einer elementaren logischen Ebene ausmachen. Die grundlegende Struktur eines topologischen Qubits platziert einen geeigneten Halbleiter-Nanodraht in unmittelbarer Nähe zu einem Supraleiter (siehe [Bild 6](#)). Aufgrund dieser Nähe wird der Halbleiter-Nanodraht ebenfalls supraleitend, und unter den richtigen Bedingungen (einschließlich eines geeigneten Magnetfelds entlang des Drahts und angelegter Spannungen an der Vorrichtung) tritt der Halbleiter-Nanodraht in die topologische Phase ein.

Die topologische Phase zeichnet sich durch sogenannte Majorana Zero Modes (MZMs) an jedem Ende des Drahts und eine „Lücke“ (Gap) in der übrigen Länge des Drahts aus, wobei die Lücke ein Bereich von Energie oder Spannung ohne erlaubte quantenmechanische Zustände ist. Der Vorteil eines topologischen Quantencomputers liegt darin, dass er deutlich stabiler ist, da die Informationen in topologischen Qubits durch topologische Eigenschaften geschützt sind, die schwerer zu destabilisieren wären. Dadurch braucht man deutlich weniger physikalische Qubits, um ein logisches Quantenbit zu bilden. Dieser inhärente Schutzmechanismus macht topologische Qubits zu vielversprechenden Kandidaten für den Aufbau stabiler Quantencomputer für komplexe Berechnungen mit minimalen Fehlerquoten. Um komplexe wissenschaftliche und kommerzielle Probleme lösen zu können, werden mindestens eine Million stabile Qubits benötigt, die eine Trillion Operationen bei Fehlerquoten weit unter 0,1 Prozent durchführen können. Zu den Herausforderungen des Ansatzes mit topologischen Qubits gehören unter anderem die komplexe Implementierung und die Fehleranfälligkeit bei thermalen Fluktuationen. Microsoft packt diese Herausforderungen bereits an und arbeitet an der Entwicklung ei-

nes topologischen Quantencomputers. Weitere Informationen zum Thema findet man unter [26], [27] und [28].

Ein weiteres experimentelles Quantencomputing-Rechenmodell wird in Form des messbasierten oder Einweg-Quantencomputing [29] realisiert (englisch: Measurement-Based/One-way Quantum Computing). Im Gegensatz zu traditionellen Gate-basierten Modellen, die auf der schrittweisen Anwendung von Quantengattern beruhen, verwendet man eine interessante Herangehensweise: Anstatt die Zustände der Qubits durch Gatter zu manipulieren, werden diese in einem vorbereiteten Verschränkungszustand platziert. Durch selektives Messen einzelner Qubits wird der Quantenzustand dann schrittweise verändert, wodurch der Computer schließlich zu einem Zielzustand gelangt.

Dieses Rechenmodell nennt man „One-way“-Computing, da die Ursprungszustände durch die Messung verändert (zerstört) werden. „One-way“-Quantencomputing bietet interessante Perspektiven für bestimmte Quantenalgorithmen und hat Potenzial für die Bewältigung spezifischer Herausforderungen im Bereich der Fehlerkorrektur. Als Herausforderung dabei gelten der hohe Bedarf an Verschränkung und die anspruchsvollen Hardware-Anforderungen.

In der Reihe der experimentellen Quantencomputing-Modelle lohnt es sich noch, das holonomische Quantencomputing (HQC) [30] als eine vielversprechende Variante zu erwähnen. HQC hat das Potenzial, die traditionellen gatterbasierten Modelle zu ergänzen und verbessern. Es wird als experimentelles Modell erforscht, um seine Eignung für bestimmte Quantenaufgaben und -anwendungen zu verstehen. Der Ansatz von HQC unterscheidet sich grundlegend von gatterbasierten Quantencomputern, da er auf geometrischen Phasen und Transformationen basiert. Die Idee ist, Quantengatter durch die Gestaltung von Pfaden im Quantenzustandsraum zu realisieren, wodurch geometrische Phasen genutzt werden, um Quantenoperationen durchzuführen. Obwohl HQC noch in der Forschungs- und Entwicklungsphase ist, werden experimentelle Fortschritte in diese Richtung gemacht. Wichtig zu beachten ist jedoch, dass sich die Technologie und die Forschung ständig weiterentwickeln und bestimmte Umsetzungen oder Anwendungen von HQC möglicherweise noch nicht weit verbreitet sind.

Neue und experimentelle Quantencomputing-Rechenmodelle bringen einzigartige Prinzipien, Vor- und Nachteile sowie Herausforderungen mit sich. Während topologisches beziehungsweise holonomisches Quantencomputing die Stabilität von Qubits durch topologische beziehungsweise geometrische Phasen und Transformationen versprechen, setzt One-way Quantum Computing auf die Vereinfachung von Quantenberechnungen durch spezielle Zustandspräparation, und Continuous-Variable Quantum Computing wiederum nutzt kontinuierliche Variablen für Rechenoperationen, was deutlich höhere Skalierbarkeit im Vergleich zu den Rechenmodellen mit diskreten Qubits und Qudits bedeutet.

Die Vielfalt dieser experimentellen Modelle trägt dazu bei, das Verständnis von Quantencomputing zu vertiefen und neue Wege für die Entwicklung von robusten, fehlertoleranten Quantenrechnern zu erforschen.

● QPU – Quantum Processing Unit

Eine QPU (Quantum Processing Unit) [31] ist eine Hardwareeinheit, die speziell für die Durchführung von Quantenberechnungen entwickelt wurde. Im Gegensatz zu herkömmlichen Computern, die auf Bits basieren, die den Zustand 0 oder 1 annehmen können, arbeitet ein QPU mit Qubits, welche aufgrund von Quantenüberlagerung (Superposition) und Verschränkung (Entanglement) gleichzeitig mehrere Zustände repräsentieren. Auf einer QPU werden Quantenalgorithmen ausgeführt, die aus mehreren Quantenoperationen bestehen.

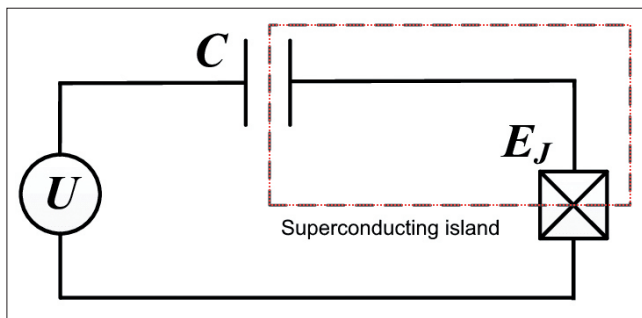
Es ist wichtig zu beachten, dass QPUs aufgrund ihrer besonderen Natur und der Herausforderungen der Quantenfehlerkorrektur noch in einem frühen beziehungsweise experimentellen Stadium stehen. Unternehmen und Forschungseinrichtungen arbeiten jedoch intensiv daran, QPUs zu verbessern und ihre Anwendungen in verschiedenen Bereichen wie Kryptografie, Optimierung, Simulation und maschinelles Lernen zu ermöglichen und zu verbessern.

Quantum Volume

Quantum Volume wurde ursprünglich 2018 von Nikolaj Moll als Metrik vorgeschlagen und im darauffolgenden Jahr von IBM erweitert. Quantum Volume misst als Metrik die Fähigkeiten und Fehlerquoten eines Quantencomputers. Sie gibt die maximale Größe von quadratischen Quantenschaltungen an, die erfolgreich von einem Computer implementiert werden können.

Die Form der Schaltungen ist unabhängig von der Architektur des Quantencomputers, aber der Compiler kann sie transformieren und optimieren, um die Funktionen des Computers optimal zu nutzen. Auf diese Weise können Quantum Volumes verschiedener Architekturen miteinander verglichen werden. Es handelt sich also um eine umfassende Metrik, die nicht nur die Anzahl der Qubits, sondern auch andere wichtige Faktoren wie Fehlerkorrektur, Gatterfehlerraten und die Qualität der Verbindungen zwischen den Qubits berücksichtigt. Dies ermöglicht eine ganzheitliche Bewertung und Vergleichbarkeit der Leistungsfähigkeit von Quantencomputern.

Ein höheres Quantum Volume deutet auf einen leistungsfähigeren und robusteren Quantencomputer hin.



Schaltkreis von Charge-Qubit (Bild 7)

Während noch viele Herausforderungen bestehen und weitere Forschung erforderlich ist, eröffnen diese experimentellen Ansätze vielversprechende Perspektiven für die Zukunft der Quanteninformatik.

Quantenprozessoren und ihre Archetypen

Das „Herz“ jedes Quantencomputers bildet eine Quantenverarbeitungseinheit – Quantenprozessor (englisch: Quantum Processing Unit, QPU, siehe Kasten **QPU – Quantum Processing Unit** und [31]). Im Folgenden werden zunächst verschiedene Typen und Ausführungen der Quantenprozessoren betrachtet, die bereits ihre Anwendung in zurzeit hergestellten Quantencomputern finden. Im Anschluss daran stellt ein weiterer Abschnitt übersichtsartig wichtige Archetypen von Quantenprozessoren vor.

Superconducting-QPU

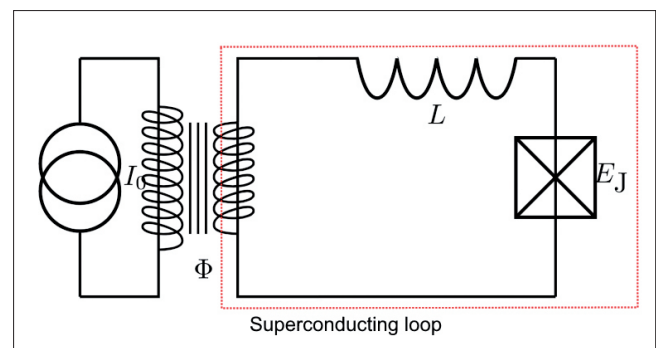
Ein Quantenprozessor mit supraleitenden Schaltkreisen [32] (superconducting circuits) repräsentiert eine vielversprechen-

de und weitverbreitete Technologie für die Realisierung von Quantencomputern. Die Herangehensweise basiert auf den Prinzipien der Supraleitung, bei denen bestimmte Materialien bei extrem niedrigen Temperaturen keinen elektrischen Widerstand aufweisen. Im Zentrum dieser Technologie stehen Josephson-Übergänge (siehe „ E_J “ in Bild 7, Bild 8 und Bild 9 sowie „Josephson junction“ in [32]): schwache Verbindungen zwischen zwei supraleitenden Materialien. Diese Josephson-Übergänge ermöglichen den ungehinderten Fluss von Cooper-Paaren, also Paaren von Elektronen, ohne elektrischen Widerstand. Die grundlegenden Bausteine sind hier supraleitende Qubits, die auf Josephson-Übergängen basieren. Diese Qubits können aufgrund von quantenmechanischer Überlagerung in mehreren Zuständen gleichzeitig existieren.

Die Operationen auf diesen Qubits werden durch die Anwendung von Mikrowellenimpulsen und Magnetfeldern durchgeführt, wodurch Quantengatter entstehen, die spezifische Operationen ausführen. Die Quantenphänomene der Verschränkung und Überlagerung werden genutzt, um komplexe Rechenoperationen durchzuführen. Die Leistungsfähigkeit von Quantenprozessoren mit supraleitenden Schaltkreisen wird durch Metriken wie das Quantum Volume (siehe Kasten **Quantum Volume**) bewertet, wobei Aspekte wie die Konnektivität der Qubits, Gattergenauigkeiten und Fehlerquoten berücksichtigt werden.

Zurzeit werden drei Archetypen von supraleitenden Qubits (siehe „Qubits Archetypes“ in [32]) umgesetzt:

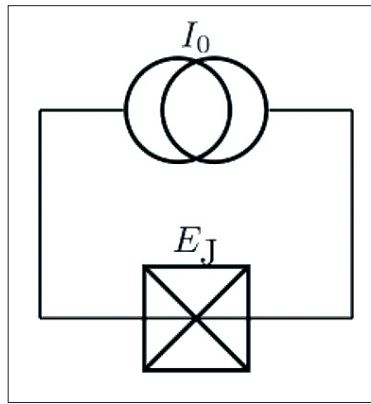
- **Charge-Qubits:** Charge-Qubits (siehe Bild 7) nutzen die elektrische Ladung als Basis für die Repräsentation von Quanteninformation. Die Information wird in diskreten Ladungszuständen eines elektrischen Systems gespeichert. Ein einzelnes Elektron kann beispielsweise in zwei verschiedenen Ladungszuständen existieren, die als 0 und 1 codiert werden. Die Quantenüberlagerung ermöglicht es Charge-Qubits, gleichzeitig in beiden Zuständen zu existieren, was zu parallelen Berechnungen führt.
- **Flux-Qubits:** Flux-Qubits (siehe Bild 8) verwenden magnetische Flüsse als Basis für die Codierung von Qubit-Informationen. Hierbei werden supraleitende Schleifen genutzt, durch die ein magnetischer Fluss fließen kann. Die Flussquantisierungseigenschaften in Supraleitern ermöglichen es, die Qubit-Information in den Flusszuständen zu speichern. Durch Superposition von Flusszuständen kann ►



Schaltkreis für das Flux-Qubit (Bild 8)

ein Flux-Qubit in mehreren Zuständen gleichzeitig existieren.

- **Phase-Qubits:** Phase-Qubits (siehe Bild 9) nutzen die Phasenunterschiede in supraleitenden Materialien, um Quanteninformation zu repräsentieren. Hierbei wird die Phase eines supraleitenden Rings verwendet, um zwischen den Zuständen 0 und 1 zu wechseln. Die Superposition von Phasenzuständen ermöglicht es Phase-Qubits, mehrere Berechnungen parallel durchzuführen. Sie sind bekannt für ihre hohe Kohärenzzeit, was bedeutet, dass sie länger in einem superpositionierten Zustand bleiben können.



Schaltkreis von Phase-Qubits (Bild 9)

Jeder Qubit-Typ hat seine spezifischen Vor- und Nachteile, die bei der Auswahl für bestimmte Anwendungen berücksichtigt werden müssen:

- **Charge-Qubits** sind durch ihre einfachere Implementierung in herkömmlichen Halbleitermaterialien gekennzeichnet, was ihre Verwendung erleichtert. Sie können jedoch anfällig für begrenzte Kohärenzzeiten und Umgebungstörungen sein, die durch elektrisches Rauschen verursacht werden.
- **Flux-Qubits** hingegen bieten eine geringere Anfälligkeit für elektrische Rauschquellen und nutzen die natürliche Quantisierung der magnetischen Flüsse in supraleitenden Schleifen. Dies ermöglicht eine präzise Codierung von Qubit-Informationen. Allerdings erfordert ihre Implementierung spezielle supraleitende Materialien und komplexe Schaltungen.
- **Phase-Qubits** zeichnen sich durch eine hohe Kohärenzzeit aus und sind weniger empfindlich gegenüber elektromagnetischen Störungen. Allerdings erfordert ihre Herstellung spezialisierte Techniken und Materialien, und sie können empfindlich gegenüber thermischen Effekten sein.

Fortschritte in der Forschung könnten dazu führen, dass bestimmte Einschränkungen in Zukunft überwunden werden und die Effizienz aller Qubit-Typen verbessert wird. Es bleibt spannend zu sehen, wie sich diese Technologien weiterent-

wickeln und welche Rolle sie in der Zukunft der Quanteninformatik spielen werden.

Namhafte Unternehmen wie IBM und Google sowie aufstrebende wie Rigetti Computing und einige andere sind aktiv an der Weiterentwicklung von supraleitenden Quantenprozessoren beteiligt. Google hat sogar im Oktober 2019 mit seinem Sycamore-Quantenprozessor eine Quantenüberlegenheit (Quantum Supremacy) deklariert – die 53-Qubit-Sycamore-QPU von Google habe ein Problem in 200 Sekunden gelöst, für das der zu diesem Zeitpunkt fortschrittlichste Supercomputer „Summit“ von IBM etwa 10 000 Jahre gebraucht hätte [33]. Auch wenn IBM widerspricht und „nur“ 2,5 Tage dafür veranschlagt, ist die Performance-Steigerung immer noch beträchtlich. Trotz bestehender Herausforderungen bieten diese Systeme vielversprechende Perspektiven für die Realisierung praktischer und skalierbarer Quantencomputer in der Zukunft und sind Vorreiter im Quantencomputing-Rennen.

„Trapped Ions“-QPU

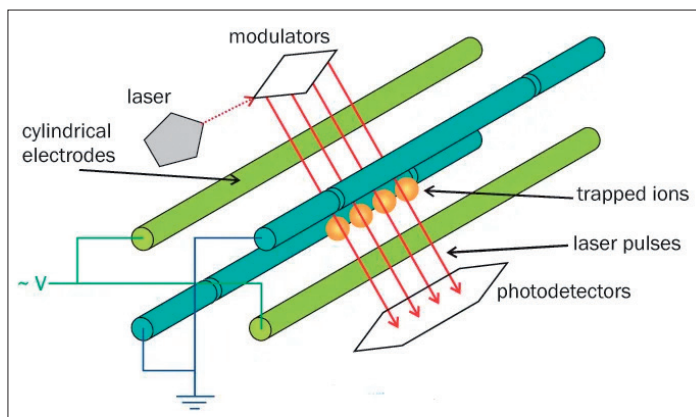
Der Ansatz des Quantencomputing mit „Trapped Ions“ („gefangenen Ionen“) basiert auf der Verwendung von einzelnen Ionen, die in elektromagnetischen Fallen eingefangen werden, um als Qubits zu fungieren. In [34] ist dieser Prozess schematisch dargestellt (siehe Bild 10).

In diesem Ansatz werden die Ionen in einer Ionenfalle mithilfe elektromagnetischer Felder eingefangen, wobei ihre internen Zustände als logische 0 und 1 dienen. Durch die präzise Manipulation dieser internen Zustände mithilfe von elektromagnetischen Pulsen und Laserstrahlen werden Quantengatter implementiert, um komplexe Berechnungen durchzuführen. Die gezielte Verschränkung der Ionen ist ein Schlüsselement, um Quantenalgorithmen effizient auszuführen. Am Ende der Berechnung wird der Quantenzustand der Ionen ausgelesen, um das Ergebnis zu erhalten.

Die Vorteile dieses Ansatzes liegen in den in der Regel längeren Kohärenzzeiten der Gefangene-Ionen-Systeme im Vergleich zu einigen anderen Qubit-Technologien. Die präzise Kontrolle über die Qubits durch elektromagnetische Felder erleichtert die Implementierung von Quantengattern. Forschungen zur Fehlerkorrektur bei Qubits in Fallen könnten dazu beitragen, Fehler in Quantenberechnungen zu reduzieren.

Allerdings steht dieser Ansatz vor Herausforderungen, insbesondere im Hinblick auf die Skalierbarkeit. Das Vernetzen und Skalieren von Gefangene-Ionen-Systemen bleiben technisch anspruchsvoll. Zudem erfordert die komplexe Apparatur, darunter elektromagnetische Fallen und Kontrollmechanismen, beträchtlichen technologischen Aufwand.

Unternehmen wie IonQ, Honeywell Quantum Solutions, Quantinuum und Alpine Quantum Technologies konzentrieren sich auf die Weiterentwicklung dieser



Qubits-Modell mit „Trapped Ions“ (Bild 10)

Technologie, um ihre Potenziale im Bereich des Quantencomputing zu erschließen.

Neutral-Atom-QPU

Die Neutral-Atom-QPU basiert auf der Verwendung neutraler Atome als perfekte Qubits. Diese Atome werden mit Laserstrahlen und elektromagnetischen Feldern manipuliert, um quantenmechanische Phänomene zu erzeugen.

Die Neutral-Atom-Technologie bietet den Vorteil, dass neutrale Atome im Vergleich zu geladenen Ionen weniger anfällig für Störungen sind. Allerdings erfordert die effektive Kontrolle der neutralen Atome technische Raffinesse und sehr niedrige Temperaturen.

Neutral-Atom-QPUs verwenden Rubidium-Atome als Qubits. Laser dienen als „optische Pinzetten“, um einzelne Atome einzufangen und bei nahezu absoluter Kälte zu stabilisieren. Optische „Gitter“, die in Bild 11 dargestellt sind, verwenden Laser, welche Rubidium-Atome (rot) voneinander trennen, um sie als Informationseinheiten in Neutral-Atom-Quantenprozessoren zu verwenden.

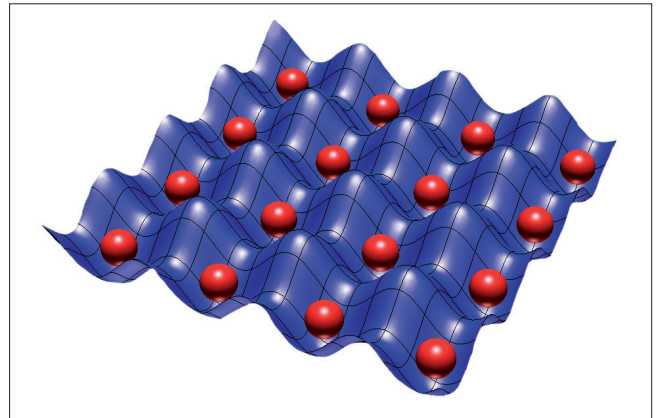
Die Rydberg-Blockade [35] ermöglicht die Implementierung von Multi-Qubit-Gates und verringert Fehler. Mit Rydberg-Wechselwirkungen können hochwertige Multi-Qubit-Gates realisiert werden. Die Technologie ermöglicht es, Qubits effizient zu manipulieren und in verschiedenen Modi zu arbeiten. Diese QPUs unterstützen analoge und digitale beziehungsweise hybride Berechnungsmodi. Der digitale Modus ermöglicht universelle Funktionalität, während der analoge Modus direkt zu Antworten führt und weniger anfällig für Fehler ist.

Die flexible Anordnung von Qubits erlaubt rekonfigurierbare Layouts und eine hohe Skalierbarkeit. Die QPUs mit neutralen Atomen erreichen eine nicht simulierbare Phase, die klassische Supercomputer übertrifft, wie zum Beispiel „Aquila“ – eine 256-Qubit-Neutral-Atom-QPU von Quera [22]. Diese Technologie zeichnet sich durch Robustheit gegenüber Fehlern und effiziente Kontrolle aus.

Die Modulbauweise des Neutral-Atom-Prozessors ermöglicht die Integration verschiedener Funktionalitäten wie digitale Quantengatter, Fehlerkorrektur, Speicher und Verarbeitungszonen. Dies bewirkt Flexibilität und Anpassungsfähigkeit an verschiedene Problemstellungen. Zudem erlaubt die hohe Skalierbarkeit die Erhöhung der Qubit-Zahl.

Durch das gezielte Anregen der Atome zu Rydberg-Zuständen [36] (hoch angeregte Zustände, bei denen ein oder mehrere Elektronen sich in energetisch weit entfernten Bahnen vom Atomkern bewegen) können die Elektronenwolken der Atome vergrößert werden. In diesem Zustand können verschiedene Atome über große Entfernungen miteinander interagieren, was den Quanteninformationstransfer ermöglicht. Dies führt zur Entstehung von Quantenverschränkung, einer Schlüsselkomponente für die Manipulation von Quanteninformation.

Die Anwendungsmöglichkeiten für Neutral-Atom-QPUs sind vielfältig. Dazu zählen die Simulation quantenmechanischer Systeme, Optimierungsalgorithmen, maschinelles Lernen und viele mehr. Die Flexibilität in der Anordnung der



Qubits als neutrale Atome im optischen „Gitter“ (Bild 11)

Qubits erlaubt es, die Konnektivität nach Bedarf anzupassen.

Die Kombination aus Robustheit, Skalierbarkeit, Flexibilität und verschiedenen Betriebsmodi positioniert die Neutral-Atom-QPU-Technologie als vielversprechende Plattform für zukünftige Quantenberechnungen. Firmen wie Quera und Pasqual haben schon Neutral-Atom-QPUs hergestellt – es stehen bereits erste Generationen von Queras 256-Qubit-Quantenprozessor Aquila [31] und von Pasquals 100-Qubit-Quantensystem (bald 200 Qubit) via Cloud zur Verfügung, und für die Zukunft wird die noch höhere Marke von 1000 Qubits angepeilt.

Aufstrebende QPU-Archetypen

Die in drei den vorigen Abschnitten vorgestellten QPU-Archetypen repräsentieren am meisten verbreitete und bereits via Cloud zur Verfügung stehende Quantenprozessoren – Superconducting-, „Trapped Ion“- und Neutral-Atom-QPUs. Es gibt jedoch noch weitere aufstrebende und vielversprechende Quantentechnologien, die einen kurzen Überblick verdienen, um den Gesamtblick über die ganze technologische Landschaft der Quantenprozessoren abzurunden:

- **Quantum Dot oder Spin Qubit QPU:** Ein Schlüsselement dabei sind Quantum-Dot-Qubits, kleine Halbleitervorrichtungen, die ein einzelnes Elektron einfangen können [37]. Der Spin des Elektrons wird als Qubit genutzt und durch ein Magnetfeld gesteuert. Die hohe Skalierbarkeit von Quantum Dots ermöglicht die Bildung von Qubit-Arrays, wobei jedes Dot unabhängig gesteuert werden kann. Trotz vielversprechender Aspekte müssen noch Herausforderungen wie die Aufrechterhaltung der Qubit-Kohärenz überwunden werden. Intel bietet bereits seine 12-Qubit-QPU „Tunnel Falls“ [38] zu Forschungszwecken für Universitäten und Forschungslabore an.
- **Photonics QPU:** Diese QPUs verwenden Photonen als Quantenbits (Qubits) und speichern Informationen in Form von Lichtquanten. Der Ansatz basiert auf der Quanteninformation von Photonen, wobei Quantenverschränkung und Superposition durch die Manipulation der Eigenschaften von Lichtteilchen erreicht werden. Die Vorteile umfassen schnelle Datenverarbeitung und geringe Anfälligkeit für Störeinflüsse. Allerdings können einige Herausforde- ►

rungen wie zum Beispiel die Photonen-Wechselwirkung und der Bedarf an hochentwickelten Photonenquellen und hochpräzisen Detektoren die Marktreife der brauchbaren photonischen Quantencomputer noch in die fernere Zukunft verschieben. Einige Firmen und Forschungseinrichtungen haben markante Erfolge mit photonischen QPUs erzielt – die kanadische Firma Xanadu etwa bietet bereits Lösungen mit ihren X-Series-Photonic-Quantencomputern an, das französische Unternehmen Quandella positioniert sich als erster europäischer Cloud-Anbieter für Quantencomputing [39], und schließlich hat die Chinesische Universität für Technologie und Wissenschaft (USTC) mit ihrem Photonic-Quantencomputer „Jiuzhang“ im Dezember 2020 die Quantenüberlegenheit und den Titel „mächtigster Quantencomputer der Welt“ für sich beansprucht [33].

Die bisher hier vorgestellten QPU-Archetypen – Superconducting, „Trapped Ions“, Neutral-Atom, Quantum-Dot und Photonic – spiegeln weitverbreitete und etablierte beziehungsweise vielversprechende Lösungen wider. Es gibt aber noch viele weitere aufstrebende Ansätze, die eines Tages zu einem Durchbruch führen könnten. Interessierte Leser finden in [40] entsprechende Verweise.

Fazit

Die Cloud ist die beste und sehr häufig die einzig mögliche beziehungsweise tragbare Option für den Zugriff auf einen Quantencomputer. Durch die Auslagerung von Quantum-Computing-Ressourcen in die Cloud können Unternehmen und Forschungseinrichtungen auf leistungsstarke Quantencomputer zugreifen, ohne in eigene teure Hardware und Software investieren zu müssen. Dies eröffnet eine neue Ära der Zusammenarbeit und Innovation, indem es selbst kleinen Unternehmen und Forschungsteams Zugang zur Quantenrechenleistung ermöglicht.

Die Cloud wird zu einem zentralen Ort für Quantum-Computing-Ressourcen, der es Nutzern ermöglicht, komplexe Quantenalgorithmen auszuführen, und innovative Anwendungen zu entwickeln. Die Integration von Quantum-Computer- und Cloud-Provider schafft eine flexiblere und skalierbare Umgebung für Quantum Computing, die es jedem ermöglicht, sich auf Anwendungen mit Quantum-Algorithmen zu konzentrieren, ohne sich um die Komplexität der zugrundeliegenden Hardware kümmern zu müssen.

Insgesamt wird Quantum Computing in der Cloud nicht nur den Zugang zu leistungsstarken Rechenressourcen revolutionieren, sondern auch die Art und Weise, wie man Probleme angeht und Lösungen entwickelt. Dieser symbiotische Ansatz zwischen Quantum Computing und Cloud-Technologien verspricht nicht nur Effizienzsteigerungen, sondern könnte auch zu Durchbrüchen in Bereichen wie künstlicher Intelligenz, Verschlüsselung und Simulation führen. Der Weg in die Zukunft der Informationstechnologie wird nun nicht zuletzt durch die Konvergenz von Quantum Computing und Cloud-Computing-Ressourcen geebnet.

Etablierte Cloud-Plattformen wie Azure Quantum [41], Amazon Braket [42], IBM Quantum [43] und Google Quantum

AI [44] sowie ganze Reihe weiterer noch aufstrebender Unternehmen bieten bereits Quantum-Computing-Zugänge über die Cloud – sowohl für Simulatoren als auch zu den „echten“ Quantencomputern. Damit bieten sie für Entwicklerinnen und Entwickler bereits heute Einstiegspunkte, an denen mit modernen Programmiersprachen wie Python und Q# und mit Frameworks beziehungsweise SDKs wie zum Beispiel Microsoft QDK und Amazon Braket SDK das Eintauchen in die faszinierende Welt der Quantenprogrammierung auf Haupt-Cloudplattformen möglich gemacht wird. ■

- [1] Dirac-Notation bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud1
- [2] „Schrödingers Katze“ bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud2
- [3] Go-Spiel bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud3
- [4] Dekohärenz bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud4
- [5] Quantenschlüsselaustausch bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud5
- [6] Quantenteleportation bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud6
- [7] Azure-Quantum-Dokumentation, Grover-Algorithmus, www.dotnetpro.de/SL2502QuantumCloud7
- [8] QFT – Quanten-Fouriertransformation bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud8
- [9] Qutrit bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud9
- [10] N. Lossau, Von Qubits zu Qudits, Welt.de, www.dotnetpro.de/SL2502QuantumCloud10
- [11] Quantum circuit bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud11
- [12] Quantengatter bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud12
- [13] Shor's algorithm bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud13
- [14] Adiabatic Quantum Computation bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud14
- [15] Quantum annealing bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud15
- [16] Problem des Handlungsreisenden bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud16
- [17] Maximaler-Schnitt-Problem bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud17
- [18] D-Wave Systems bei Wikipedia, www.dotnetpro.de/SL2502QuantumCloud18
- [19] Leap – D-Wave Quantum Cloud-Service, www.dotnetpro.de/SL2502QuantumCloud19
- [20] Olivier Pfister, Continuous-variable quantum computing in the quantum optical frequency comb, 2020 J. Phys. B: At. Mol. Opt. Phys. 53 012001, www.dotnetpro.de/SL2502QuantumCloud20
- [21] Quera.com, What is analog quantum computing?, www.dotnetpro.de/SL2502QuantumCloud21
- [22] Quera.com, 256-qubit quantum Computer Aquila, <https://www.quera.com/aquila>

- [23] Anyon bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud22
- [24] Weltlinie bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud23
- [25] Braid group bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud24
- [26] Azure Quantum, Topological qubits,
www.dotnetpro.de/SL2502QuantumCloud25
- [27] nLab.org, Topological Quantum Computation.
www.dotnetpro.de/SL2502QuantumCloud26
- [28] arXiv.org, E.C. Rowell, Z. Wang, Mathematics of Topological Quantum Computer,
<https://arxiv.org/pdf/1705.06206.pdf>
- [29] One-way Quantum-Computer bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud27
- [30] arXiv.org, Geometric and holonomic quantum computation. <https://arxiv.org/abs/2110.03602>
- [31] Nvidia.com, R. Merrit, What is a QPU,
www.dotnetpro.de/SL2502QuantumCloud28
- [32] Superconducting quantum computing bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud29
- [33] Quantenüberlegenheit,
www.dotnetpro.de/SL2502QuantumCloud30
- [34] J.Hui. How to build a Quantum Computer with Trapped Ions, www.dotnetpro.de/SL2502QuantumCloud31
- [35] Quera.com, Rydberg-Blockade,
www.dotnetpro.de/SL2502QuantumCloud32

- [36] Rydberg-Zustand bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud33
- [37] Spin qubit quantum computer bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud34
- [38] Intel.com, Intel's New Chip to Advance Silicon Spin Qubit Research for Quantum Computing,
www.dotnetpro.de/SL2502QuantumCloud35
- [39] Quandela Cloud, <https://cloud.quandela.com/>
- [40] List of proposed Quantum registers bei Wikipedia,
www.dotnetpro.de/SL2502QuantumCloud36
- [41] Microsoft Learn, Azure Quantum,
www.dotnetpro.de/SL2502QuantumCloud37
- [42] Amazon Braket,
www.dotnetpro.de/SL2502QuantumCloud38
- [43] IBM Quantum, <https://quantum.ibm.com/>
- [44] Google Quantum AI, <https://quantumai.google>



Mykola Dobrochynskyy

ist Senior Architekt bei der Porsche AG. Seine Schwerpunkte sind Softwarearchitektur, modellgetriebene Softwareentwicklung, generative Programmierung, KI sowie Cloud- und dienstorientierte Softwarearchitekturen.

www.xing.com/profile/Mykola_Dobrochynskyy

dnpCode

A2502QuantumCloud

DEV ACADEMY

HANDS-ON-WORKSHOPS UND
WEITERBILDUNG FÜR SOFTWARE-
ENTWICKLER UND -ARCHITEKTEN



**2 TAGE
REMOTE/INHOUSE**

Erste Schritte mit Docker für .NET-Entwickler

TRAINING!

Was wird behandelt

- Einführung in Docker
- Source-Code mit Container verwenden
- Images erstellen und bereitstellen
- Container-Kommunikation und Management
- Docker-Compose in Aktion
- Security
- Monitoring



Gregor
Biswanger
Trainer

SOFTWARE-QUANTENSIMULATOREN IN C#

Marke Eigenbau

Zurzeit sind Quantensimulatoren die am besten zugänglichen Werkzeuge für den Entwurf und das Testen von Quantenalgorithmen. Warum diese nicht mit C# selber bauen?

Echte Quantencomputer, also hardwaretechnische Systeme, die mit quantenmechanischen Zuständen arbeiten, sind extrem aufwendig, fehleranfällig und teuer. Daher wird das Computing mit Quantencomputern nach Möglichkeit im Vorfeld simuliert. Neuerdings möchte man allerdings Aufgaben, für deren Lösung in der Vergangenheit aufwendige Quantencomputer notwendig waren, direkt durch Simulatoren lösen. So gibt es in der Quantenchemie schon erste praktische Anwendungen in der Quantensimulation von Materialien.

Für Forschung und Lehre stellen Quantensimulatoren somit einen extrem wichtigen Lösungsansatz dar, um kostengünstig und effektiv variationelle Quantenalgorithmen zu entwickeln und zu testen. Diese Entwicklung kann man auch sehr gut im Quantencomputer-Infoportal Quantiki [1] beobachten. Auf der Webseite gibt es immer wieder neue Simulatoren für Quantencomputer für eine Vielzahl von Programmiersprachen. Zurzeit finden Sie dort rund 150 verschiedene Simulatoren, darunter viele von namhaften Anbietern wie Microsoft, IBM oder dem Fraunhofer-Institut.

Ein einfacher Simulator für das QDK

Wie man unter Einsatz des Microsoft Quantum Development Kits (QDK) einen eigenen reversiblen Simulator in der Programmiersprache C# erstellen kann, zeigt das nachfolgende Beispiel. Als Grundlage genutzt wird der von Microsoft vorgestellte Code aus dem Quellcode-Repository. Grundkenntnisse in der C#-Programmierung werden dabei vorausgesetzt.

Der im Beispielfall vorgestellte Simulator kann nur Quantenprogramme simulieren, die aus klassischen Operationen wie X, CNOT, CCNOT oder beliebig kontrollierten X-Operationen bestehen. Der Simulator stellt den Quantenzustand dar, indem er jedem Qubit einen booleschen Wert zuordnet. Damit ist das Testen von Quantenoperationen möglich, die boolesche Funktionen auswerten.

Download und Start des Simulators in C#

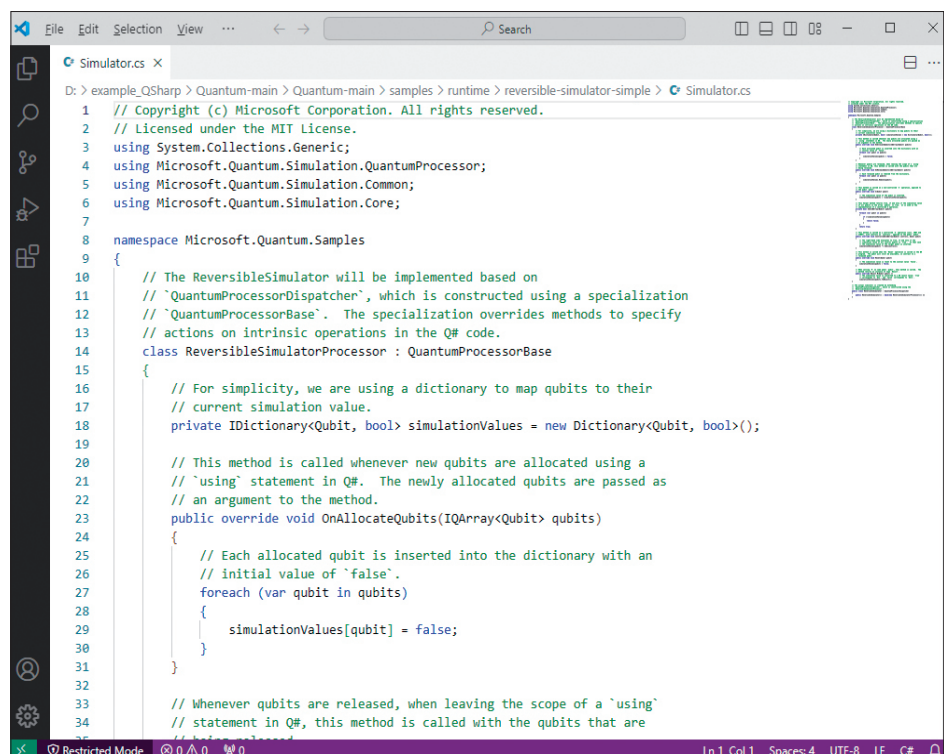
Als Entwicklungsumgebung für dieses Programmbeispiel wird Visual Studio Code in Verbindung mit dem C# Dev Kit und dem Quantum Development Kit (QDK) verwendet.

Um das hier vorgestellte Beispiel nachvollziehen zu können, benötigen Sie das aktuelle GitHub-Repository *microsoft/Quantum* als Download-ZIP-Datei [2].

Nachdem Sie das benötigte Repository eingerichtet beziehungsweise die ZIP-Datei entpackt haben, können Sie auch schon mit dem Simulator-Beispiel beginnen. Öffnen Sie hierfür die Datei *Simulator.cs* unter der Verzeichnisstruktur *Quantum-main/samples/runtime/reversible-simulator-simple* mit Visual Studio Code (Bild 1).

Die Klasse ReversibleSimulatorProcessor

Den Einstieg in die Erstellung eines eigenen Simulators ermöglicht Ihnen die Klasse *ReversibleSimulatorProcessor*, indem Sie die *QuantumProcessorBase*-Klasse erweitern. Das *IQuantumProcessor*-Interface macht es für Sie sehr einfach, eigene Simulatoren für klassische Operationen zu schreiben.



Die Datei Simulator.cs in Visual Studio Code (Bild 1)

```
class ReversibleSimulatorProcessor :
    QuantumProcessorBase {
    private IDictionary<Qubit, bool>
        simulationValues = new Dictionary<Qubit, bool>();
    // Aktion und Methoden zum Definieren von Operationen
}
```

Der Klasse wird im Beispielcode ein *Dictionary* hinzugefügt, das den aktuellen Simulationswert für jedes Qubit im Programm speichert. Die Quantenzustände $|0\rangle$ und $|1\rangle$ werden als boolesche Werte mit *True* und *False* dargestellt.

Die Basisklasse *QuantumProcessorBase* enthält eine Methode, die es ermöglicht, jede intrinsische Operation [3], die Sie überschreiben möchten, zu definieren, um ihr Verhalten im neuen Simulator festzulegen.

Achten Sie bei der Verwendung eines reversiblen Simulators darauf, keine Quantenprogramme zu simulieren, die nichtklassische Operationen wie Hadamard-Gatter [4] und Quanten-T-Gatter [5] enthalten.

Im nächsten Schritt beginnen Sie damit, neu zugewiesene Qubits mit *false* zu initialisieren, indem Sie die Methode *OnAllocateQubits* überschreiben:

```
public override void OnAllocateQubits(IQArray<Qubit>
    qubits) {
    foreach (var qubit in qubits) {
        simulationValues[qubit] = false;
    }
}
```

Über die Methode *OnReleaseQubits* werden Qubits aus dem *Dictionary* entfernt, wenn sie freigegeben werden.

```
public override void OnReleaseQubits(IQArray<Qubit>
    qubits) {
    foreach (var qubit in qubits) {
        simulationValues.Remove(qubit);
    }
}
```

Diese beiden Quantenoperationen gewährleisten, dass die Simulationswerte immer dann zur Verfügung stehen, wenn eine Operation auf ein Qubit angewendet wird, und dass keine Simulationswerte für Qubits im *Dictionary* vorhanden sind, zu denen es im aktuellen Kontext keinen Bezug mehr gibt.

Die beiden im weiteren Fortgang verwendeten Methoden *X()* und *ControlledX()* implementieren die Aktionen der klassischen Operationen:

```
public override void X(Qubit qubit) {
    simulationValues[qubit] = !simulationValues[qubit];
}

public override void ControlledX(IQArray<Qubit>
    controls, Qubit qubit) {
    simulationValues[qubit] ^= And(controls);
}
```

Listing 1: Q#-Mehrheitsoperation

```
operation ApplyMajority(a : Qubit, b : Qubit,
    c : Qubit, f : Qubit) : Unit {
    within {
        CNOT(b, a);
        CNOT(b, c);
    } apply {
        CCNOT(a, c, f);
        CNOT(b, f);
    }
}

operation RunMajority(a : Bool, b : Bool, c : Bool)
: Bool {
    using ((qa, qb, qc, f) =
        (Qubit(), Qubit(), Qubit(), Qubit())) {
        within {
            ApplyPauliFromBitString(
                PauliX, true, [a, b, c], [qa, qb, qc]);
        } apply {
            ApplyMajority(qa, qb, qc, f);
        }
        return MResetZ(f) == One;
    }
}
```

Die Methode *X()* wird aufgerufen, wenn eine einfache X-Operation, also eine direkte Aktion mit einem Qubit, simuliert wird. Eine willkürlich kontrollierte X-Operation kann hierbei auch die Simulation von CNOT- und CCNOT-Gattern umfassen. Die Wirkung einer X-Operation invertiert den Simulationswert eines Qubits, während im Fall einer willkürlich kontrollierten X-Operation das Ziel-Qubit nur dann invertiert wird, wenn alle Kontroll-Qubits mit *true* belegt sind.

Die Methode *Result* liefert die Messung eines Qubits in einem Q#-Programm mit dem Ergebnis *One* oder *Zero*. Die *Reset*-Methode entfernt in diesem Fall nicht das letzte Qubit aus dem aktuellen Kontext, sondern setzt den Simulationswert auf seinen Anfangswert *false* zurück.

```
public override Result M(Qubit qubit) {
    return simulationValues[qubit] ? Result.One :
        Result.Zero;
}

public override void Reset(Qubit qubit) {
    simulationValues[qubit] = false;
}
```

Die *ReversibleSimulatorProcessor*-Klasse sollte als Simulator verwendet werden, indem eine *QuantumProcessorDispatcher*-Instanz erstellt wird. Das Quantum-Entwicklerteam rät, immer eine eigene Klasse für den Simulator zu erstellen. ►

```
public class ReversibleSimulator :
    QuantumProcessorDispatcher {
    public ReversibleSimulator() :
        base(new ReversibleSimulatorProcessor()) {}
}
```

Hiermit haben Sie Ihren ersten eigenen Simulator in Verbindung mit dem Quantum Development Kit für Q# erstellt.

Verwendung des neuen Simulators

Ist die Implementierung des Simulators abgeschlossen, können Sie anschließend den neuen Simulator mit einer Q#-Operation ausführen, die eine Mehrheitsoperation durchführt, indem sie drei boolesche Eingabewerte bereitstellt. **Listing 1** zeigt die Implementierung dieser Q#-Operation – siehe auch den Artikel zur Programmierung mit Q# unter [6] in dieser dotnetpro-Ausgabe – in einer *.qs-Datei für Visual Studio Code.

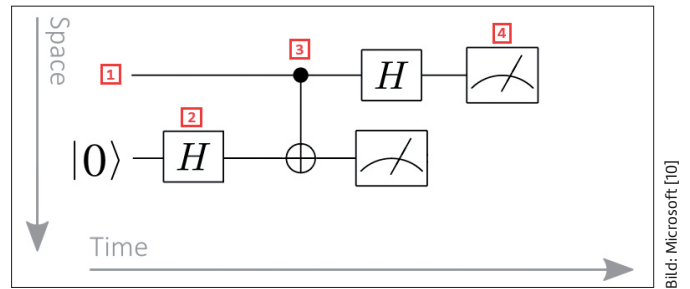
Zum Schluss können Sie alles zusammenfügen, indem Sie die Q#-Operation mit dem neu erstellten Simulator in einem C#-Host-Programm ausführen. **Listing 2** zeigt den Aufruf der Q#-Operation aus dem C#-Code heraus. Der Code wertet die Mehrheitsoperation für alle verschiedenen Eingaben aus und gibt alle Simulationsergebnisse zurück.

Dieses Beispiel sollte aufzeigen, wie Sie alternative Anwendungen für benutzerdefinierte Simulatoren im Umfeld des Quantum Development Kits implementieren können.

C#-Quantengatter-Simulator

Der oben vorgestellte Quantensimulator oder auch alternativ der QDK-Simulator (siehe [6]) unterstützen eine Vielzahl von Quantengattern und -schaltungen. Bei der Entwicklung im Quantum Computing handelt es sich in den meisten Fällen um eine sehr kleine Teilmenge von Quantengattern und Mechanismen, die tatsächlich genutzt werden.

Der Vorteil, wenn man etwas von Grund auf neu entwickelt: Man erhält eine Menge Wissen darüber, wie bestimmte Vorgänge und Verfahren konkret funktionieren. Im Quantum Computing werden viele Theorien in mathematischen Operationen ausgedrückt – siehe den Artikel zum Quantum Computing unter [7] in dieser dotnetpro-Ausgabe. Diese Operationen werden dann in den tatsächlichen Quantenschaltkreisen auf einer realen Hardware simuliert. Bei der Entwicklung des Codes fällt aber auch auf, dass dieser in erster Linie aus einer Reihe von mathematischen Schritten besteht.



Hadamard-Quantenschaltkreis (Bild 2)

Das nachfolgende C#-Programm soll als Simulator in der Lage sein, verschiedene Quantengatter-Operationen auf einem Zwei-Qubit-System zu simulieren. Mit diesem Programm können Sie Operationen wie Pauli-X [8], oft auch einfach nur als X-Gatter bezeichnet, Hadamard [4] und CNOT (Controlled-not-Gate) [9] simulieren.

Für Quantengatter, die in diesem Fall auf zwei Qubits arbeiten, ist eine Wechselwirkung zwischen den fraglichen Qubits erforderlich. Das X-Gatter verschränkt ein Qubit-Paar, das Hadamard-Gatter sorgt für Überlagerungszustände, und beim CNOT-Gatter wird abhängig vom Zustand des Kontroll-Qubits der Zustand des Ziel-Qubits entweder negiert oder beibehalten. **Bild 2** zeigt ein Beispiel anhand eines Hadamard-Quantenschaltungsdiagramms [10]. Hierbei wird das Qubit-Register (1) als horizontale Linien angezeigt. Die Quantenoperationen werden durch das verwendete Quantengatter (2) dargestellt. Das kontrollierte Gatter (3) wirkt auf zwei oder mehr Qubits. Im Beispiel stellt das Symbol ein CNOT-Gatter dar. Über Punkt 4 (4) erfolgt der Messvorgang. Hierbei ist zu beachten, dass in einer Quantenschaltung die Zeit immer von links nach rechts fließt.

Quantengesteuerte Gatter sind immer Zwei-Qubit-Gatter, die ein Ein-Qubit-Gatter auf ein Ziel-Qubit anwenden, wenn sich ein Kontroll-Qubit in einem bestimmten Zustand befindet. Beachten Sie, dass für Multi-Qubit-Vorgänge die gleichen Konventionen wie für Ein-Qubit-Vorgänge gelten.

Die Aufgabe

Das hier vorgestellte Simulationsprogramm soll programmtechnisch dem Quantenschaltungsdiagramm aus **Bild 3** folgen. Das heißt, beim Starten des Programms wird standardmäßig der Anfangszustand $|0\rangle$ verwendet.

Sobald der Start des Programms abgeschlossen ist, fordert der Simulator Sie auf, eine Sequenz von Quantengattern an-

● Listing 2: Ausführung des neuen Simulators mit Q#-Operation

```
public static void Main(string[] args) {
    var sim = new ReversibleSimulator();
    var bits = new[] {false, true};

    foreach (var a in bits) {
        foreach (var b in bits) {
            foreach (var c in bits) {
                var f = RunMajority.Run(sim, a, b, c).Result;
                Console.WriteLine(
                    $"Majority({a,5}, {b,5}, {c,5}) = {f,5}");
            }
        }
    }
}
```


zuwenden. Wenn Sie in die Konsole beispielsweise `h0,cnot` eingeben, wird der Simulator angewiesen, das Hadamard-Gatter auf Qubit $|0\rangle$ anzuwenden, gefolgt von einer CNOT-Gatteroperation. Im Anschluss an diese Zustandsmanipulation führt der Simulator eine definierte Anzahl von Iterationen durch (im Beispiel sind es 100) und simuliert die Entwicklung des Quantenzustands mit jedem Schritt.

Am Ende des Prozesses wird in der Konsole die Verteilung der Quantenzustände ausgegeben.

Implementierung in C#

Zur Erstellung des Simulators wird als Entwicklungsumgebung Visual Studio 2022 Version (in einer Version ab Community Edition) mit dem .NET Framework 8.0 verwendet. Alternativ können Sie auch .NET Framework 6.0 benutzen.

Legen Sie für das Beispiel ein neues Console-App-Projekt in Visual Studio an. Vergeben Sie als *Project name* den Begriff *QuantumSimulator* und wählen Sie .NET 8.0 als Framework aus. Nach dem Klick auf die Schaltfläche *Create* erstellt Visual Studio automatisch ein lauffähiges Konsolenprogramm mit der Ausgabe *Hello World!*

Im nächsten Schritt können Sie für das Beispiel die beiden Codezeilen in der Klasse *Program.cs* entfernen.

Der Programmcode in Listing 3 stellt den kompletten Beispiel-Simulator dar. Die definierten Variablen für die Simulationsdaten werden beim Start alle mit 0 initialisiert. Mit deren Hilfe werden die Qubits programmtechnisch nachgebildet. Im Beispiel wird gedanklich von zwei Bits ausgegangen. Diese beiden Bits können sich in verschiedenen Kombinationen von Zuständen befinden, denn wenn eines 0 ist, kann das andere entweder 0 oder 1 sein und umgekehrt. Die möglichen Zustände sind also *00*, *01*, *10* und *11*. Mathematisch gesehen kann es bei *n* Bits insgesamt 2^n Kombinationen geben. In dem Beispiel gehen wir von der Annahme aus, dass man ein einzelnes ursprüngliches Bit in zwei Listen darstellen kann. Bit 0 ist [1,0] und Bit 1 ist [0,1].

Ein Qubit stellt fast dasselbe dar. Die beiden Zustände eines einzelnen Qubits werden ja immer als $|0\rangle$ oder $|1\rangle$ dargestellt und sind damit in Form eines Spaltenvektors wieder gleich: Qubit $|0\rangle$ ist [1 0] und Qubit $|1\rangle$ ist [0 1]. Das heißt, ein Qubit kann sich entweder in einem Zustand $|0\rangle$ oder im Zustand $|1\rangle$ befinden, oder in einem beliebigen Zustand dazwischen. Das bezeichnet den Superpositionszustand des Qubits.

Aus dem allgemeinen Spaltenvektor eines Qubits können Sie ersehen, dass wenn $b=0$ ist, Sie $|0\rangle$ erhalten, und wenn $a=0$ ist, Sie $|1\rangle$ erhalten. Dies unterscheidet den Spaltenvektor eines klassischen Bits von einem Spaltenvektor eines Qubits. Bei klassischen Bits kann der Spaltenvektor in der Zeile nie etwas anderes als 1 enthalten, und es kann immer nur eine der Zeilen 1 sein. Bei Qubits können in jeder Zeile des Spaltenvektors beliebige komplexe Zahlen stehen, solange diese alle quadriert zu 1 summiert werden.

Quantengatter

Qubits oder besser gesagt die Simulation von Qubits reichen allein aber nicht aus. Sie müssen auch einige Operationen mit ihnen durchführen. Hier kommen die Quantengatter ins ►

Listing 3: Implementierung des Simulators (Teil 1)

```
using System;

namespace QuantumSimulator
{
    class Program
    {
        static double r0 = 1; static double r1 = 0;
        static double r2 = 0; static double r3 = 0;
        static double i0 = 0; static double i1 = 0;
        static double i2 = 0; static double i3 = 0;
        static double a0 = 0; static double a1 = 0;
        static double b0 = 0; static double b1 = 0;
        static int shots = 100;
        static void Main(string[] args)
        {
            Console.Clear();
            Console.WriteLine(
                "C# Quanten-Gatter-Simulator");
            Console.WriteLine(
                "Mögliche Gatter sind: x0, x1, h0, h1, cnot");

            string gate = Console.ReadLine();
            string[] gates = gate.Split(',');

            foreach (var g in gates)
            {
                switch (g)
                {
                    case ("x0"):
                        Console.WriteLine("Initialisierung " +
                            "Gatter Pauli-X Qubit 0");
                        InitGateX0();
                        break;
                    case ("x1"):
                        Console.WriteLine("Initialisierung " +
                            "Gatter Pauli-X Qubit 1");
                        InitGateX1();
                        break;
                    case ("h0"):
                        Console.WriteLine("Initialisierung " +
                            "Gatter Hadamard Qubit 0");
                        InitGateH0();
                        break;
                    case ("h1"):
                        Console.WriteLine("Initialisierung " +
                            "Gatter Hadamard Qubit 1");
                        InitGateH1();
                        break;
                    case ("cnot"):
                        Console.WriteLine("Initialisierung " +
                            "Gatter CNOT");
                        InitGateCNOT();
                        break;
                }
            }
        }
    }
}
```

● Listing 3: Implementierung des Simulators (Teil 2)

```

        default:
            Console.WriteLine(
                "Keine Auswahl getroffen!");
            break;
    }
}

Console.WriteLine("Simulation läuft... " +
    "Iterationen -> " + shots.ToString());
double sq = r0 * r0 + i0 * i0 + r1 * r1 + i1 *
    i1 + r2 * r2 + i2 * i2 + r3 * r3 + i3 * i3;
if (Math.Abs(sq - 1) > 0.00001)
    normalizationStateVector(sq);

Console.WriteLine($"Prozesslauf für " *
    "{shots.ToString()} Iterationen");
double z0 = 0; double z1 = 0; double z2 = 0;
double z3 = 0;
double p0 = (r0 * r0 + i0 * i0);
double p1 = (r1 * r1 + i1 * i1) + p0;
double p2 = (r2 * r2 + i2 * i2) + p1;
double p3 = (r3 * r3 + i3 * i3) + p2;

for (int i = 0; i <= shots; i++)
{
    Random rnd = new Random();
    double r = rnd.NextDouble();

    if (r < p0) z0 = z0 + 1;
    if (r >= p0 && r < p1) z1 = z1 + 1;
    if (r >= p1 && r < p2) z2 = z2 + 1;
    if (r <= p2 && r < p3) z3 = z3 + 1;
}

Console.WriteLine("Ergebnisse:");
Console.WriteLine($"00 -> {z0}");
Console.WriteLine($"01 -> {z1}");
Console.WriteLine($"10 -> {z2}");
Console.WriteLine($"11 -> {z3}");

Console.ReadLine();
}

private static void normalizationStateVector(
    double sq)
{
    double nf = Math.Sqrt(1/sq);
    r0 = r0 * nf;
    i0 = i0 * nf;
    r1 = r1 * nf;
    i1 = i1 * nf;
    r2 = r2 * nf;
    i2 = i2 * nf;
    r3 = r3 * nf;
}

```

Spiel. Deren Name stammt noch aus der klassischen Informatik von den wohl bekannten *AND*-, *OR*-, und *NOT*-Gattern. Im Unterschied zum klassischen booleschen Gatter haben Quantengatter jedoch einige Eigenschaften, die erfüllt sein müssen.

- Quantengatter können durch unitäre Matrizen [11] dargestellt werden.
- Kann die Operation eines Quantengatters nicht durch eine entsprechende unitäre Matrix ausgedrückt werden, so kann es nicht als gültiges Gatter betrachtet werden.
- Die Operationen von Quantengattern sind vollständig umkehrbar. Dies ist ein großer Unterschied zu booleschen Gattern.
- Bei einem *UND*-Gatter können Sie die Eingänge nicht bestimmen, wenn Sie die Ergebnisse betrachten. Sie können nur sagen: Wenn das Ergebnis 1 ist, dann sind beide Eingänge 1. Wenn das Ergebnis aber 0 ist, können Sie nicht bestimmen, ob die Eingänge (1,0), (0,1) oder (0,0) sind.
- Bei einem Quantengatter lässt sich der Eingang nur allein durch die Betrachtung des Ausgangs rekonstruieren.

Da Quantengatter nur Matrizen sind, können Sie diese wie aufgezeigt im Code leicht konstruieren, indem Sie in der Methode die entsprechende Matrix definieren. Die Anwendung

Verteilung der Quantenzustände (Bild 3)

eines Gatters ist dann einfach die Multiplikation des Gatters mit dem Qubit.

Normalisierter Zustand

Die Methode *normalizationStateVector(double sq)* sorgt dafür, dass die Summe der quadrierten Amplituden gleich eins ist, wodurch sichergestellt wird, dass die mit jedem Zustand verbundenen Wahrscheinlichkeiten konsistent sind. Bild 3 zeigt die Verteilung der Quantenzustände nach 100 Iterationen bei einem Hadamard-Gatter auf Qubit 1, gefolgt von einer *CNOT*-Gatteroperation.

```

        i3 = i3 * nf;
    }

    private static void InitGateX0()
    {
        a0 = r0; r0 = r1; r1 = a0;
        a0 = i0; i0 = i1; i1 = a0;
        a0 = r2; r2 = r3; r3 = a0;
        a0 = i2; i2 = i3; i3 = a0;
    }

    private static void InitGateX1()
    {
        a0 = r1; r1 = r3; r3 = a0;
        a0 = i1; i1 = i3; i3 = a0;
        a0 = r0; r0 = r2; r2 = a0;
        a0 = i0; i0 = i2; i2 = a0;
    }

    private static void InitGateH0()
    {
        a0 = (r0 + r1) / Math.Sqrt(2);
        a1 = (i0 + i1) / Math.Sqrt(2);
        b0 = (r0 - r1) / Math.Sqrt(2);
        b1 = (i0 - i1) / Math.Sqrt(2);
        r0 = a0; i0 = a1; r1 = b0; i1 = b1;
        a0 = (r2 + r3) / Math.Sqrt(2);
        a1 = (i2 + i3) / Math.Sqrt(2);
        b0 = (r2 - r3) / Math.Sqrt(2);
        b1 = (i2 - i3) / Math.Sqrt(2);
        r2 = a0; i2 = a1; r3 = b0; i3 = b1;
    }

    private static void InitGateH1()
    {
        a0 = (r0 + r2) / Math.Sqrt(2);
        a1 = (i0 + i2) / Math.Sqrt(2);
        b0 = (r0 - r2) / Math.Sqrt(2);
        b1 = (i0 - i2) / Math.Sqrt(2);
        r0 = a0; i0 = a1; r2 = b0; i2 = b1;
        a0 = (r1 + r3) / Math.Sqrt(2);
        a1 = (i1 + i3) / Math.Sqrt(2);
        b0 = (r1 - r3) / Math.Sqrt(2);
        b1 = (i1 - i3) / Math.Sqrt(2);
        r1 = a0; i1 = a1; r3 = b0; i3 = b1;
    }

    private static void InitGateCNOT()
    {
        a0 = r1; r1 = r3; r3 = a0;
        a0 = i1; i1 = i3; i3 = a0;
    }
}

```

Fazit

Dieser kleine Workshop sollte zeigen, welche Möglichkeiten für die Programmierung von Quantensimulationen eine Rolle spielen können. Die Mehrzahl der verfügbaren Bibliotheken, als Beispiel sei Qiskit genannt, sind schwergewichtig und benötigen eine Menge Abhängigkeiten, um zu funktionieren.

Das C#-Beispiel will aufzeigen, dass es sich bei Simulationen im Kern um eine Reihe von mathematischen Transformationen handelt, die man nicht nur auf fortschrittlicher Quanten-Hardware simulieren kann. Für kleine Modelle kann sich deshalb die Erstellung eines eigenen Simulators lohnen.

Das Thema Quantensimulatoren und -programmiersprachen bleibt ein spannendes Feld für Forschung und Entwicklung. Bleiben Sie auf jeden Fall am Ball. ■

[1] Quantum Information Portal, <https://www.quantiki.org>

[2] Microsoft QDK Samples, www.dotnetpro.de/SL2502Quantensimulator1

[3] Intrinsische Funktion bei Wikipedia, www.dotnetpro.de/SL2502Quantensimulator2

[4] openHPI, Hadamard-Gatter, www.dotnetpro.de/SL2502Quantensimulator3

[5] Quanten-T-Gatter bei Wikipedia, www.dotnetpro.de/SL2502Quantensimulator4

[6] Daniel Basler, Q# für Einsteiger, dotnetpro 2/2025, Seite 38 ff., www.dotnetpro.de/A2502QSharp

[7] Mykola Dobrochynskyy, Wegweisende Symbiose, dotnetpro 2/2025, Seite 20 ff., www.dotnetpro.de/A2502QuantumCloud

[8] openHPI, Pauli-X-Gatter, www.dotnetpro.de/SL2502Quantensimulator5

[9] openHPI, CNOT-Gatter, www.dotnetpro.de/SL2502Quantensimulator6

[10] Microsoft Learn, Quantum circuit diagram conventions, www.dotnetpro.de/SL2502Quantensimulator7

[11] Unitäre Matrizen bei Wikipedia, www.dotnetpro.de/SL2502Quantensimulator8



Daniel Basler

arbeitet als Lead Developer und Softwarearchitekt. Seine Schwerpunkte liegen auf Cross-Plattform-Apps, Android, JavaScript und Microsoft-Technologien.

dnpCode

A2502Quantensimulator

QUANTENALGORITHMEN MIT VISUAL STUDIO CODE

Q# für Einsteiger

Mit der Programmiersprache Q# von Microsoft, konzipiert für Quantencomputing, können Interessierte eigene Erfahrungen mit Quantenalgorithmen sammeln.

Nachdem der Beitrag „Wegweisende Symbiose“ [1] die Grundbausteine des Quantum Computing in der Cloud dargelegt hat, beschäftigt sich dieser Workshop mit der Frage, wie man einen Quantencomputer programmiert.

Daher das Beste zuerst: Ein Quantencomputer kann analog zu einem klassischen Computer programmiert werden. Es werden allerdings spezielle Algorithmen eingesetzt, um die besonderen Eigenschaften von Superposition und Verschränkung aus dem Bereich des Quantum Computing zu nutzen. Das heißt, zur Lösung eines Problems beziehungsweise einer Aufgabe werden diverse Rechenoperatoren durchgeführt. Im Unterschied zur klassischen Programmierung werden Quantengatter angewendet, um Qubits zu verschränken, ihren Zustand zu manipulieren oder sie auszulesen. Dazu muss ein Algorithmus geschrieben werden, der festlegt, nach welcher Logik die Gatter angewendet werden sollen.

Man könnte einen Quantencomputer auch nur mit klassischen Algorithmen verwenden. Das würde jedoch keinerlei Vorteile bringen, da die besonderen Eigenschaften wie Superposition und Verschränkung einfach ungenutzt blieben.

Spezielle Quantenalgorithmen wie der Faktorisierungs-Algorithmus von Shor [2] ermöglichen es, sehr komplexe Probleme zu lösen, für die althergebrachte Methoden viel zu viel Zeit benötigen würden. Der Shor-Algorithmus ist für die Kryptografie sehr bedeutend, weil er einen nichttrivialen Teiler sehr viel schneller findet als klassische Algorithmen, so zum Beispiel auch bei der Primfaktorzerlegung. Der Shor-Algorithmus zählt zur Klasse der Faktorisierungsverfahren und gehört zu den bekanntesten Quantenalgorithmen.

Ein weiteres wichtiges Beispiel ist der Suchalgorithmus von Grover [3]. Will man zum Beispiel in einem Telefonbuch mit N Einträgen eine bestimmte Nummer finden, so muss man einen Eintrag nach dem anderen durchsuchen, bis man den richtigen Wert gefunden hat. Um einen exakten Eintrag in einer unsortierten Liste zu finden, benötigt man im Allgemeinen $N/2$ Versuche. Bei einem Telefonbuch mit 100 000 Einträgen entspricht das 50 000 Aufrufen. Mit dem Quantensuchalgorithmus von Grover benötigt man im Durchschnitt nur $\sqrt{N}$ Schritte zum Ergebnis. Möglich wird dies durch die Grover-Iteration im Quantenregister. Sie erhöht die Wahrscheinlichkeit für das richtige Ergebnis. Hierzu später mehr – bei der Implementierung des Grover-Algorithmus.

Quantencomputer und Quantenalgorithmen ermöglichen eine Vielzahl von Dingen, die klassische Programmierung nur mit sehr großem Aufwand an Zeit lösen kann. Daher starten Sie doch jetzt einfach mit der programm-basierten Ausführung

● Listing 1: Quantenverschränkung in Q#

```
import Microsoft.Quantum.Diagnostics.*;

operation Main() : (Result, Result) {
    use (qubit_1, qubit_2) = (Qubit(), Qubit());
    H(qubit_1);
    CNOT(qubit_1, qubit_2);
    DumpMachine();
    let (result_1, result_2) = (M(qubit_1),
        M(qubit_2));
    Reset(qubit_1);
    Reset(qubit_2);
    return (result_1, result_2);
}
```

des Q#-Compilers und der Q#-Simulation aus Visual Studio Code heraus.

Das Quantum Development Kit

Das Quantum Development Kit, kurz QDK, bietet einen hervorragenden Einstieg mit niedrigen Hürden in die Q#-Entwicklung. Der programmgesteuerte Einstieg in den Q#-Compiler ist über Visual Studio Code und seine NuGet-Pakete problemlos möglich. Voraussetzungen hierfür sind die aktuelle Version von Visual Studio Code [4] und die entsprechende QDK-Erweiterung [5]. Nach der Installation der QDK-Erweiterung haben Sie alle technischen Voraussetzungen, um Programme für den integrierten Quantensimulator zu schreiben, den Q#-Code zu debuggen und ihn auszuführen.

Des Weiteren unterstützt das QDK die Ausführung für Python- und Jupyter-Notebooks. Wenn Sie über ein Azure-Konto verfügen, können Sie Ihre Q#-Algorithmen auch an eine Azure-Quantenhardware binden und den Q#-Code per Visual Studio Code übermitteln und auf echter Quantenhardware ausführen. Für den Einstieg ist der integrierte Quantensimulator aber auf jeden Fall ausreichend.

Schreiben von Q#-Programmen

Sind die Projektvorbereitungen abgeschlossen, sprich die entsprechenden Tools wie Visual Studio Code und das QDK installiert, können Sie auch schon mit der Programmierung in Q# beginnen.

Auch in Q#-Programmen stellen Operationen und Funktionen die Grundeinheiten für die Ausführung von Code dar. Sie entsprechen dabei in etwa einer Funktion in C++ oder Python beziehungsweise einer statischen Methode in C# oder Java.

In Q# stellt eine Operation immer eine Quanten-Unterroutine (Subroutine) dar. Das heißt, sie ist einfach eine aufrufbare Routine, die Quantenoperationen enthält.

Eine Q#-Funktion verkörpert eine klassische Subroutine, die innerhalb eines Quantenalgorithmus verwendet wird. Sie darf klassischen Code, aber keine Quantenoperationen enthalten. Insbesondere dürfen Funktionen weder Qubits zuweisen oder ausleihen, noch dürfen sie Operationen aufrufen. Es ist jedoch möglich, ihnen Operationen oder Qubits zur Verarbeitung zu übergeben.

Funktionen sind in Q# somit völlig deterministisch in dem Sinne, dass ihr Aufruf mit den gleichen Argumenten immer das gleiche Ergebnis liefert.

Das erste Q#-Programm

Jetzt soll es endlich losgehen! Der Start in Verbindung mit Visual Studio Code ist ganz einfach: Öffnen Sie Visual Studio Code und stellen Sie die QDK-Erweiterungen auf *Debug* und *Simulation* ein. Mithilfe dieser Einstellungen können Sie später den Q#-Code debuggen und die Codeausführung direkt in Visual Studio Code im integrierten Quantensimulator ausführen.

Grundsätzlich besteht ein Q#-Programm aus einer oder mehreren Programmdateien. Diese Programmdateien haben die Dateiendung *.qs* und enthalten den Q#-Quelltext.

Wählen Sie für das Beispiel den Menübefehl *File | New Text File* aus und speichern Sie das neue Textdokument mit *Save As...* als *Main.qs*-Programmdatei ab.

Als Einstieg in die Programmierung mit Q# erstellen Sie hier ein kleines Beispielprogramm. Das minimale Q#-Programm in **Listing 1** zeigt auf, wie sich Q#-Code mit Visual Studio Code implementieren und starten lässt.

Das Beispiel veranschaulicht die Quantenverschränkung, ein Schlüsselkonzept des Quantencomputing. Die Verschränkung ist ein grundlegendes Konzept der Quantenmechanik, das eine Quantenkorrelation zwischen Quantensystemen beschreibt [6]. Wenn zwei oder mehrere Qubits verschränkt sind, hängt der Zustand eines Qubits vom Zustand des anderen Qubits ab, auch wenn beide sehr weit auseinanderliegen. Diese Quantenkorrelation ist ein einzigartiges Merkmal von Quantensystemen, das kein Gegenstück in der klassischen Physik hat.

Im Beispiel aus **Listing 1** erstellen Sie einen bestimmten Zwei-Qubit-verschränkten Zustand, der als Bell-Paar (Bell-Zustand) [7] bezeichnet wird. Wenn Sie in einem Bell-Paar ein Qubit im Zustand $|0\rangle$ messen, so wissen Sie, dass sich das andere Qubit ebenfalls im Zustand $|0\rangle$ befindet, ohne es messen zu müssen. In dem Beispiel weisen Sie ein Paar Qubits zu und verschränken diese dann.

Im ersten Schritt importieren Sie die Q#-Standardbibliothek `Microsoft.Quantum.Diagnostics.*`. Diese liefert Ihnen eine Anzahl von vordefinierten Funktionen und Operationen

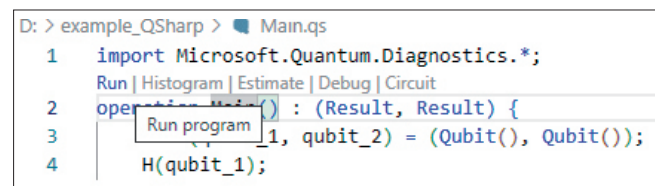
für die Entwicklung mit Q#. Mehr Informationen zu den Q#-Bibliotheken finden Sie unter [8].

Als Nächstes implementieren Sie die Operation *Main()*. Sie erhält keine Argumente und liefert zwei *Result*-Werte zurück. Beachten Sie, dass man Q#-Code immer in eine Operation einbetten muss, die den Code ausführt und mit *return* das Messergebnis zurückliefert, das *Zero* oder *One* sein kann.

Der Vorteil der Operation *Main()* in Visual Studio Code ist, dass man diese Operation direkt über *Run* über der *Main*-Operation ausführen kann (**Bild 1**). Sollten Sie eine anderen Operation-Namen benötigen, müssen Sie zwingend die Operation in einen .NET-Namespace einbetten.

Über das Schlüsselwort *use* weisen Sie zwei Qubits zu. In Q# werden Qubits immer im Zustand $|0\rangle$ zugewiesen. Somit befinden sich die Qubits *qubit_1* und *qubit_2* im Zustand $|0\rangle$.

Um die Qubits für die Verschränkung vorzubereiten, müssen Sie eines davon in eine gleichmäßige Oberposition *set*



```
D: > example_QSharp > Main.qs
1 import Microsoft.Quantum.Diagnostics.*;
2 operation Main() : (Result, Result) {
3     use qubit_1, qubit_2 = (Qubit(), Qubit());
4     H(qubit_1);
}
```

Ausführung direkt aus Visual Studio Code heraus (**Bild 1**)

zen, wo es eine 50-Prozent-Wahrscheinlichkeit hat, den Zustand $|0\rangle$ oder $|1\rangle$ anzunehmen.

Das Setzen des Qubits in die sogenannte Superposition (Überlagerung) erfolgt über die Hadamard-Operation [9] *H(qubit_1)*. Das Hadamard-Gatter ist ein Gatter in der Quanteninformatik, das einen Zustand von $|0\rangle$ oder $|1\rangle$ in eine gleiche Superposition von $|0\rangle$ und $|1\rangle$ verwandelt. In einem Quantengatter-Schaltplan wird ein Hadamard-Gatter durch ein Quadrat mit dem Buchstaben *H* dargestellt.

Die Verschränkung der Qubits geschieht über eine Verkopplung mit der *CNOT*-Operation. Das *CNOT*-Gatter [10] ist eine Zwei-Qubit-Operation, wobei das erste Qubit als Steuer- und das zweite Qubit als Ziel-Qubit bezeichnet wird. Im Beispiel stellt *qubit_2* das Steuer- und *qubit_1* das Ziel-Qubit dar.

Bevor die Qubits im Beispiel über die Operation *M* (Measure) gemessen werden, ist es wichtig zu überprüfen, ob Ihr vorheriger Code sie erfolgreich verschränkt hat. Sie können mithilfe des Befehls *DumpMachine* den aktuellen Status Ihres Q#-Programms ausgeben.

Nach der Überprüfung der Verschränkung können Sie über die Vorgabe *M* (Measure) die Qubits messen. Sie messen hierbei *qubit_1* und *qubit_2* und reduzieren ihre Quantenzustände mit gleich hoher Wahrscheinlichkeit in *Zero* oder *One*. Über das Schlüsselwort *let* deklarieren Sie zwei neue Variablen und speichern darin das Messergebnis.

Bevor Sie eine Operation in Q# verlassen dürfen, müssen die verwendeten Qubits zwingend in den Zustand $|0\rangle$ versetzt werden. Dazu verwenden Sie im Beispiel einfach den *Reset*-Befehl. Über *return(result_1,result_2)*; wird die *Main*-

Operation abgeschlossen, und es werden die *Result*-Werte *Zero* oder *One* zurückgegeben.

Führen Sie das Programm über *Run* aus oder drücken Sie einfach [Strg]+[F5]. **Bild 2** zeigt das Ergebnis der Qubit-Verschränkung des Bell-Paares in der Debug-Console.

Grundstruktur eines Q#-Programms

Um Ihnen ein Gefühl für die Sprache Q# zu vermitteln, ist es wichtig zu verstehen, wie Q# seinen Quellcode strukturiert.

```

1 import Microsoft.Quantum.Diagnostics.*;
2 operation Main() : (Result, Result) {
3     use (qubit_1, qubit_2) = (Qubit(), Qubit());
4     H(qubit_1);
5     CNOT(qubit_1, qubit_2);
6     DumpMachine();
7     let (result_1, result_2) = (M(qubit_1), M(qubit_2));
8     Reset(qubit_1);
9     Reset(qubit_2);
10    return (result_1, result_2);
11 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Basis	Amplitude	Probability	Phase
00>	0.7071+0.0000i	50.0000%	0.0000
11>	0.7071+0.0000i	50.0000%	0.0000

(One, One)
Finished shot 1 of 1
Q# simulation completed.

Ergebnisausgabe in der Debug Console (**Bild 2**)

Die Operation *Main()* steht in Q# für die direkte Ausführung der Funktionalität. Schauen Sie sich **Listing 2** genauer an. Das Q#-Programm erstellt einen Superpositionszustand. Im Code weist man *q* einem Qubit im Zustand $|0\rangle$ zu, wendet das Hadamard-Gatter an, um das Qubit in Superposition zu setzen, misst den Qubit-Zustand, setzt das Qubit zurück und gibt schließlich das Messergebnis als *Result* zurück.

Der Q#-Code beginnt mit einem benutzerdefinierten Namespace. Auch in Q# helfen Namespaces bei der Organisation verwandter Funktionen. Jedes Q#-Programm kann nur einen Namespace haben. Die Q#-Standardbibliothek verfügt, wie in anderen .NET-Libraries, über vordefinierte Namespaces, in denen Funktionen und Operationen enthalten sind, die Sie in Quantenprogrammen verwenden können, siehe [8].

Standardmäßig startet Q# die Ausführung eines Programms aus der *Main()*-Operation. Optional können Sie aber auch eigene Einstiegspunkte definieren, indem Sie das Attribut *@EntryPoint()* verwenden, um eine beliebige Operation im Programm als Ausführungspunkt anzugeben. In **Listing 2** stellt die Operation *MeasureOneQubit* den Einstiegspunkt dar.

Wie Sie in **Listing 2** erkennen können, ist Q# strukturell sehr ähnlich zu vertrauten Sprachen wie C# oder auch Java, etwa

in seiner Verwendung von Semikolons zum Beenden von Anweisungen, geschweiften Klammern zum Gruppieren von Anweisungen, den Funktionsaufrufen und dem Doubleslash zum Einleiten von Kommentaren.

Q# ist eine stark typisierte Sprache, sodass alle Variablen, Argumente und Rückgabewerte einen zugehörigen Typ besitzen müssen. Hierfür bietet Q# integrierte Typen wie *Int*, *Double*, *Bool* und *String* an. Des Weiteren gibt es spezielle Typen für das Quantencomputing, hierzu zählen *Qubit*, *Result* und *Pauli*. So stellt der Typ *Result* das Ergebnis einer Quantenmessung dar und kann einen von zwei Werten (*Zero* oder *One*) aufweisen.

Die Superposition in **Listing 2** gibt den Typ *Result* zurück, der dem Rückgabebetyp der Funktion *M* entspricht. Das Messergebnis wird auch hier in einer neuen Variablen gespeichert, die mithilfe der *let*-Anweisung definiert wurde. Q# bietet auch Typen, die Bereiche, Arrays und Tupel definieren. Über die Typdeklaration [11] können Sie zudem benutzerdefinierte Typen erstellen. Eine Übersicht über die primitiven Typen in Q# finden Sie unter [8].

Variablen, die über *let* definiert wurden, sind unveränderlich und können in interpolierten Strings (Strings, die mit einem Dollarzeichen \$ beginnen) verwendet werden, um auf der Konsole ausgegeben zu werden.

Q# fasst alle Operationen und Funktionen unter dem Begriff *Callables* zusammen. Hierbei gehören alle *Callables* zu einem Namespace.

In Q# können nur Operations Quantenoperationen durchführen, Funktionen sind ausschließlich für Berechnungen vorbestimmt und dadurch auf diese beschränkt.

Die wichtigsten Q#-Features für den Einstieg sind:

- *Mutable* wird verwendet, um Variablen zu initialisieren, die später im Code mit *set* geändert werden können.
- *Bool* entspricht einem Wahrheitswert mit *true* oder *false*.
- *Int* ist ein Datentyp für einen 64-Bit-Integerwert mit Vorzeichen.
- *BigInt* stellt einen Integerwert mit beliebiger Größe und Vorzeichen dar.
- *Double* ist der Datentyp für eine 64-Bit-Fließkommazahl.
- *String* nimmt eine Zeichenkette als Text auf. Der Text ist UTF-16-codiert.
- *Unit* stellt einen Datentyp für ein Singleton mit dem Wert *()* zur Verfügung.
- *Range* definiert eine geordnete Sequenz – entweder auf- oder absteigend – von Integerwerten mit gleichem Abstand.

● Listing 2: Superposition bestimmen

```
namespace Superposition {
    @EntryPoint()
    operation MeasureOneQubit() : Result {
        // Weisen Sie ein Qubit zu. Standard ist Zustand 0.
        use q = Qubit();
        H(q);
        let result = M(q);
        Reset(q);
        return result;
    }
}
```

- `new Result[n]` wird verwendet, um ein neues Result-(Ergebnis-)Array der Größe `n` zu initialisieren.
- `[ ]` wird verwendet, um auf das Element eines Array zuzugreifen.
- Neue Arrays können über Expressions [12] aus bestehenden Arrays erstellt werden. Eine Kopier- und Aktualisierungs-Expression ist ein Ausdruck der Form `arr w/ idx < value`, der ein neues Array konstruiert, bei dem alle Elemente auf das entsprechende Element in `arr` gesetzt werden, mit Ausnahme des Elements (der Elemente) an `idx`, die auf das/die Element(e) gesetzt werden. Das resultierende Array kann mit dem Operator `w/=` derselben Variablen zugewiesen werden.
- `ApplyToEach` ist eine Operation, die eine andere Operation als Parameter und ein Array erhält, um die gegebene Operation auf jedes Element in dem Array auszuführen.
- `IndexRange` stellt eine Funktion dar, die bei einem Array einen Bereich erstellt, um über die Indizes der Elemente im Array zu iterieren.

- `ResetAll` stellt sicher, dass alle Elemente eines Qubit-Arrays im Zustand `Result.Zero` sind, sodass sie sicher freigegeben werden können.

Des Weiteren lassen sich in Q# die Typen von Ein- und Ausgabeparametern sehr gut parametrieren; dieser Vorgang wird sehr intensiv von den Q#-Standardbibliotheken genutzt. Auch das mit einem `@EntryPoint` angegebene Callable muss konkrete Ein- und Ausgabetypen verwenden.

Qubits

Das Zuweisen eines Qubits geschieht mithilfe des Schlüsselworts `use`. Qubits werden immer im $|0\rangle$ -Zustand zugewiesen. Sie können auch mehrere Qubits zuordnen und dann über ihren Index auf die einzelne Zuweisung zugreifen.

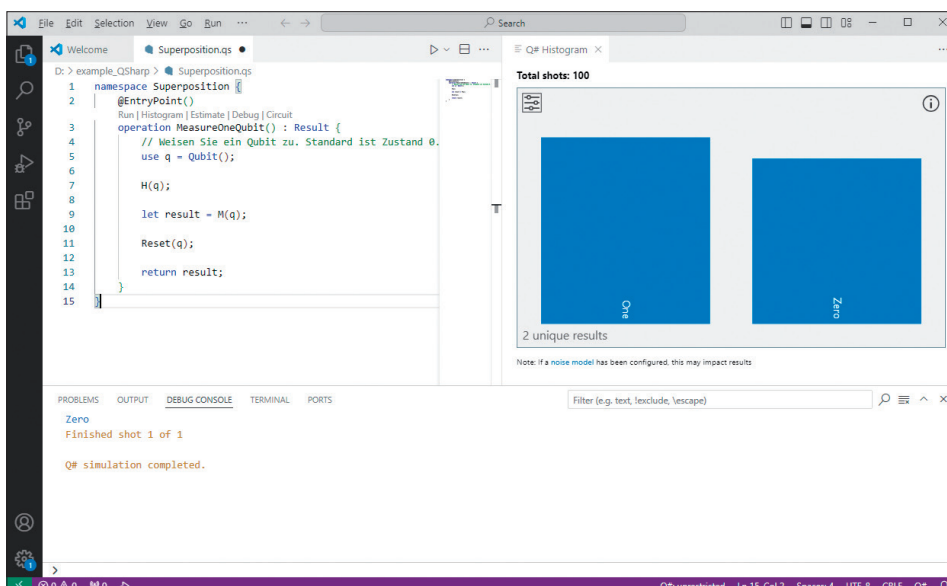
Nachdem Sie in Q# ein Qubit zugeordnet haben, können Sie es an Operationen und Funktionen übergeben, also an eine aufrufbare Routine, die Quantenvorgänge enthält, die den Zustand des Qubit-Registers ändern.

Q# konzentriert sich beim Messen von Qubits auf die sogenannten projektiven Messungen auf einzelne Qubits. Dieses Konzept wird als Pauli-Messung [13] bezeichnet. Das heißt, in Q# misst die `Measure`-Operation (`M`) eines oder mehrere Qubits (Multi-Qubit) in der angegebenen Pauli-Basis, die in der Regel durch Arrays vom Typ `PauliX`, `PauliY` und `PauliZ` dargestellt werden.

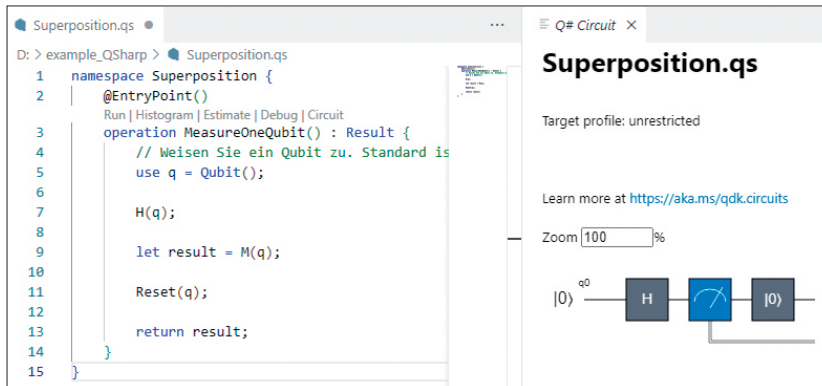
Beachten Sie unbedingt bei der Programmierung mit Q#, dass Qubits sich nach dem Durchlaufen einer Routine immer im Zustand $|0\rangle$ befinden müssen. Verwenden Sie hierfür den Aufruf über `Reset`, um jedes Qubit auf den $|0\rangle$ -Zustand zurückzusetzen, bevor es am Ende des Programms freigegeben wird. Das Nichtzurücksetzen eines Qubits führt in Q# zu einem Laufzeitfehler.

Sie können den in Q# vordefinierten Namespace `Std.Intrinsic` nutzen, indem Sie die Routine `MResetZ` verwenden, um zum Beispiel das Listing 2 zu optimieren. `MResetZ` (`MResetY`, `MResetZ`) kombiniert die Mess- und Rücksetzungsvorgänge in einem Schritt.

Der Hauptunterschied der beiden vorgestellten Q#-Programme zwischen Quantenverschränkung in Listing 1 und der Superposition in Listing 2 besteht darin, dass bei der Verschränkung der Austausch von Quanteninformationen zwischen zwei Teilchen erfolgt. Verschränkung ist die Fähigkeit der Quantenteilchen, ihre berechneten Ergebnisse miteinander zu verknüpfen. Im Unterschied dazu beinhaltet die Superposition die Unsicherheit, dass ein Teilchen mehrere Zustände annehmen kann. Das heißt, bei der Superposition hat das ►



Verteilung der Ergebnisse als Ansicht im Histogramm (Bild 3)



Superposition im Quantenschaltkreis (Bild 4)

Teilchen die Fähigkeit, eine Kombination aller möglichen Zustände zu sein.

Run, Debug, Histogram und Circuit

Durch die gute Integration des QDK in Visual Studio Code lassen sich Q#-Programme ganz schnell lokal auf dem integrierten Simulator testen. Klicken Sie hierfür einfach in der Befehlsliste unterhalb des Eintrags `@EntryPoint` oder `Main()` auf `Run` oder drücken Sie `[Strg]+[F5]`, und Ihre Ausgabe wird in der Debug-Console angezeigt.

Um Ihr Q#-Programm zu debuggen, verwenden Sie den Befehl `Debug` aus der Ausführungsliste, oder Sie drücken `[F5]`. Verwenden Sie dabei die Debug-Steuerelemente in der Menüleiste von Visual Studio Code, um den Code zu überspringen, in die Routine einzusteigen oder diese zu verlassen.

Über das Häufigkeitshistogramm, in der Befehlsliste als `Histogram` aufgeführt, können Sie eine Übersicht über die Verteilung der Ergebnisse durch mehrmaliges Ausführen eines Q#-Programms ermitteln. Jeder Balken im Histogramm (Bild 3) entspricht einem möglichen Ergebnis, und seine Höhe stellt die Häufigkeit dar, mit der dieses Ergebnis betrachtet wird. Beim Aufruf des Histogramms legen Sie die Anzahl der Aufnahmen fest, die das Programm ausführen soll. Die Standardeinstellung ist hier mit 100 Aufnahmen vorgelegt.

Über den Befehl `Circuit` können Sie das Visualisieren eines Quantenschaltkreises vornehmen. Bei Quantenschaltkreisdigrammen handelt es sich um eine visuelle Darstellung von Quantenoperationen. Sie zeigen den Fluss von Qubits durch das Quantenprogramm, einschließlich der auf sie angewendeten Gatter und Messungen. Die Schaltung in Bild 4 zeigt das Qubit-Register, das auf den Zustand $|0\rangle$ initialisiert wird. Dann wird ein Hadamard-Gatter H auf das Qubit angewendet, gefolgt von einer Messoperation, die durch ein Meter-Symbol dargestellt wird. Weitere Informationen zum Visualisieren von Quantenschaltkreisen finden Sie unter [14].

Zufallszahlen generieren

Nach den beiden bisherigen doch sehr theoretischen Beispielen in Q# erstellt das Q#-Programm in Listing 3 eine Zufallszahl zwischen 0 und 5000. Der QDK-Simulator erzeugt in diesem Fall Pseudozufallszahlen, wie Sie es vergleichbar aus C# oder Java kennen. Die gewünschte Zufallszahl wird von der

Operation `RandomBits()` erzeugt, die mit gleicher Wahrscheinlichkeit die Werte `Zero` oder `One` liefert. Die Operation `Main()` liefert dann einen Integer-Wert als Zufallszahl aus dem Maximalwert 5000 zurück.

Bild 5 zeigt das Ergebnis der Top 10 im Histogramm nach 100 Programmausführungen. Das Beispiel ist für den QDK-Simulator optimiert; da als Rückgabewert kein `Result` genutzt wird, ist eine Ausführung auf einer echten Quantenhardware nicht möglich. Das Beispiel macht deutlich, dass man in Q#-Programmen auch fast klassische Code-Routinen erstellen kann.

Lambda und Partial Application

Q# beherrscht wie andere Programmiersprachen auch die Verwendung von Lambdaausdrücken. Dabei wird der Pfeil-operator `->` für eine anonyme Funktion und der Operator `=>` für eine anonyme Operation verwendet.

In Q# gibt es auch die Besonderheit der Partial Application, dabei handelt es sich um eine Teilanwendung, bei der man aus einem Callable ein neues erstellt, in dem aber einige Argumente belegt und andere offengelassen sind, was durch einen Unterstrich markiert wird.

Konzeptionell ist die Partial Application gleichbedeutend mit einer Lambda-Expression, die die angewandten Argumen-

● Listing 3: Create Random in Q#

```
import Std.Convert.ResultArrayAsInt;
import Std.Math.BitSizeI;
import Microsoft.Quantum.Diagnostics.*;

operation RandomBits() : Result{
    use qubit = Qubit();
    H(qubit);
    return MResetZ(qubit);
}

operation Main() : Int{
    mutable sample = 0;
    let max = 5000;

    repeat{
        mutable bits = [];
        for i in 1..BitSizeI(max){
            set bits += [RandomBits()];
        }
        set sample = ResultArrayAsInt(bits);
    } until (sample <= max);

    return sample;
}
```


te erfasst und die offengelassenen Argumente als Parameter aufnimmt. So führt der Lambda-Ausdruck $a \rightarrow f(x, a)$ in einer Partial Application zu den neuen Aufruf $f(x, _)$. Hierbei ist f die Funktion, und die erfasste Variable x ist wandelbar. Mithilfe der Partial Application kann man sich als Entwickler sehr schnell eigene, individuelle Operationen erstellen.

Grover-Algorithmus

Das letzte Beispiel zeigt die Q#-Implementierung des Grover-Algorithmus. Der Algorithmus bezieht sich auf die Suche nach unstrukturierten Einträgen in einer Datenbasis. Wie zu Beginn des Artikels bereits erläutert, benötigt ein klassischer Algorithmus für die Suche $N/2$ Rechenschritte. Der Grover-Algorithmus liefert bei der Ausführung eine quadratische Beschleunigung in Form von $\sqrt{N}$ -Schritten. Eine ausführliche Erklärung der Theorie hinter dem Grover-Algorithmus finden Sie unter [15]. Bei dem Beispiel handelt es sich um einen randomisierten Algorithmus, der eine korrekte Antwort mit hoher Wahrscheinlichkeit wiedergibt. Die Wahrscheinlichkeit einer fehlerhaften Antwort kann durch wiederholte Ausführung des Algorithmus verkleinert werden. Um die Übersichtlichkeit in dem Beispiel zu gewährleisten, definiert der vorgestellte Algorithmus nur ein einzelnes Suchproblem.

In dem Beispiel wird davon ausgegangen, dass man jedes Suchproblem mit einer mathematisch abstrakten Funktion in Form von $f(x)$ formulieren kann. Die Lösung für das Suchproblem ist $f(x) = 1$. Wenn das Element x keine Lösung darstellt, ist $f(x) = 0$.

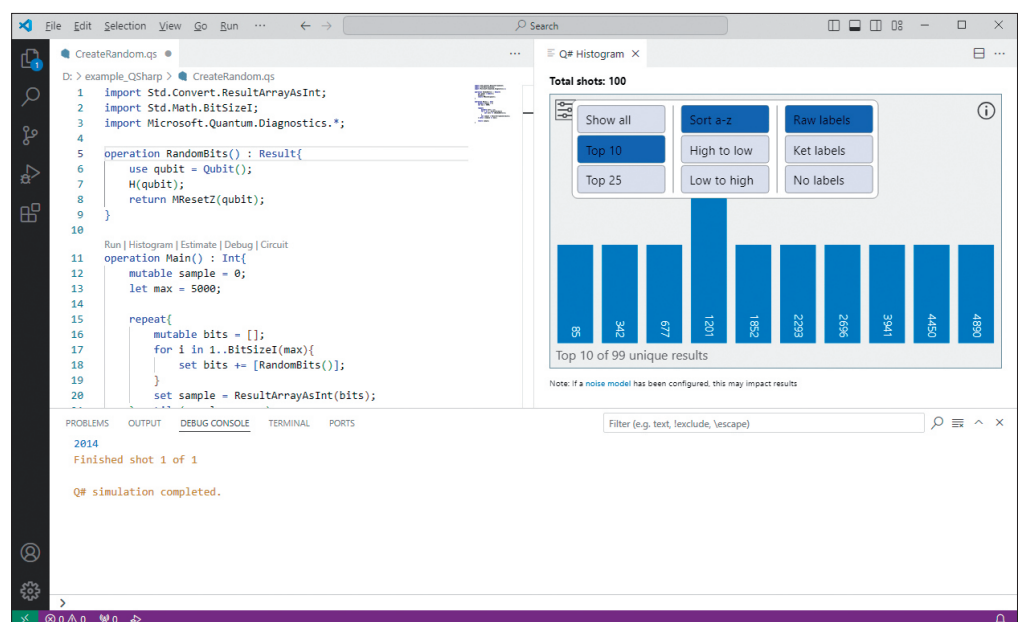
Listing 4 zeigt die Implementierung des Grover-Algorithmus in seiner einfachsten Form: Nach dem Import der benötigten Q#-Namespaces ruft die *Main*-Operation die Funktion *CalculateOptimalIterations* auf Grundlage der angegebenen Anzahl von Qubits auf. Die implementierte Q#-Funktion berechnet die optimale Anzahl an Iterationen für eine bestimmte Anzahl von Qubits in einem Register. Nach der Berechnung gibt die Funktion die Anzahl als gerundete nächste Ganzzahl zurück.

Beim Aufruf der Operation *GroverSearch* wird dieser als Parameter die Anzahl der Qubits im Qubit-Register übergeben. Dieses Register codiert die vorläufige Lösung zum Suchproblem und wird nach der Operation über *MResetEach2* gemessen, und die Qubits werden wieder in den Zustand $|0\rangle$ versetzt. Des Weiteren verwendet die Operation die drei Helferoperationen *PrepareUniform*, *ReflectAboutUniform* und

ReflectAboutAllOne. Im Register *All-Zero State* wird eine einheitliche Überlagerung (Superposition) über alle Basiszustände hergestellt.

Die Operation *ReflectAboutUniform* reflektiert über den Zustand der gleichmäßigen Überlagerung. Zunächst transformiert sie die Überlagerung in den Zustand $|0\rangle$. Dann transformiert sie den Zustand von $|0\rangle$ auf $|1\rangle$. Schließlich wird über den Zustand *All-One* reflektiert. Die Operation heißt *ReflectAboutUniform*, weil sie geometrisch als eine Spiegelung im Vektorraum am einheitlichen Überlagerungszustand interpretiert werden kann.

Die Operation *ReflectAboutMarked* reflektiert über den durch abwechselnde Nullen und Einsen markierten Basiszu-



Top 10 nach 100 Programmausführungen (Bild 5)

stand. Dies geschieht durch die Anwendung des Grover'schen Diffusionsoperators auf die Eingangsqubits. Die Operation verwendet das *outputQubit* als Hilfsqubit und initialisiert dieses in dem Zustand, indem die Gatter angewendet werden.

Die Operation wendet dann das Gatter auf jedes andere Qubit im Register an, wodurch der Zustand des jeweiligen Qubits umgedreht wird. Schließlich wird das kontrollierte Gatter auf das Hauptqubit und die Eingangsqubits angewendet. Diese Operation kippt das Hilfsqubit nur dann, wenn sich alle Eingangsqubits in dem Zustand befinden, der der markierte Zustand ist.

Der Befehl *Adjoint* führt eine Adjunktion [16] der jeweiligen Operation aus. *Adjoint* vor einer Operation führt folglich diese selbst aus, und ein einfaches *Adjoint* macht eine Operation wieder rückgängig. Ein vorangestelltes *Funktor Controlled* hingegen führt die Operation nur aus, wenn sich alle Qubits in einem Kontroll-Array im Zustand $|1\rangle$ befinden.

Somit ist der Grover-Algorithmus implementiert und kann in Visual Studio Code mit dem QDK-Simulator ausgeführt werden. Bild 6 zeigt die Ausgabe für den Algorithmus in ►

der Debug-Console. Über die Auswahl *Circuits* können Sie sich den Verlauf der visualisierten Gatter in der Quantenschaltung des Grover-Algorithmus ansehen (Bild 7).

Azure Quantum

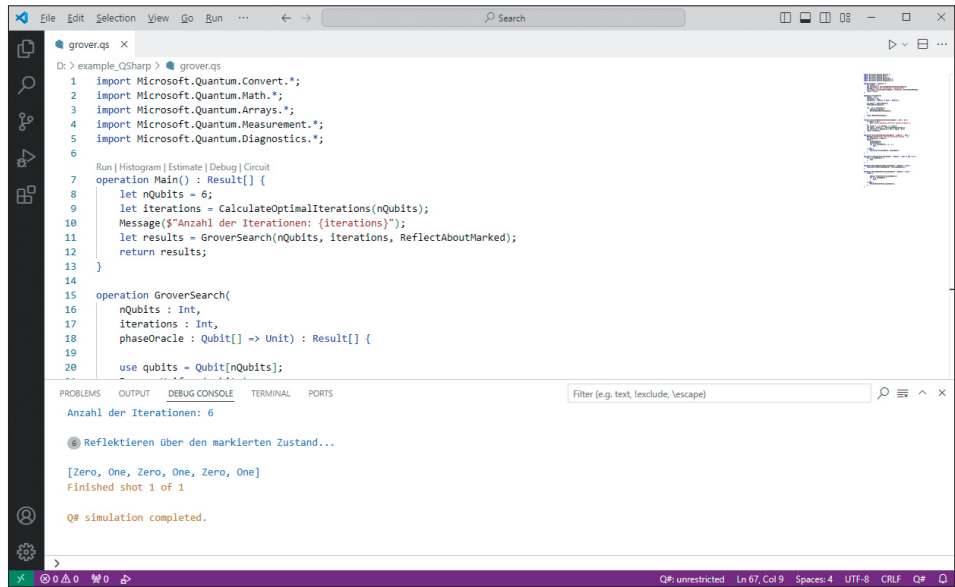
Verfügen Sie über ein entsprechendes Microsoft-Azure-Konto, so können Sie auch Azure Quantum nutzen, um Ihre Quantenprogramme auf einer realen Quantenhardware auszuführen. Diese Integration ermöglicht es Ihnen, das Potenzial von Quantencomputing für komplexe Probleme zu nutzen. Für den Einstieg und das Erkunden von Q# reicht jedoch in den meisten Fällen der integrierte Simulator im QDK im Zusammenspiel mit Visual Studio Code aus.

Fazit

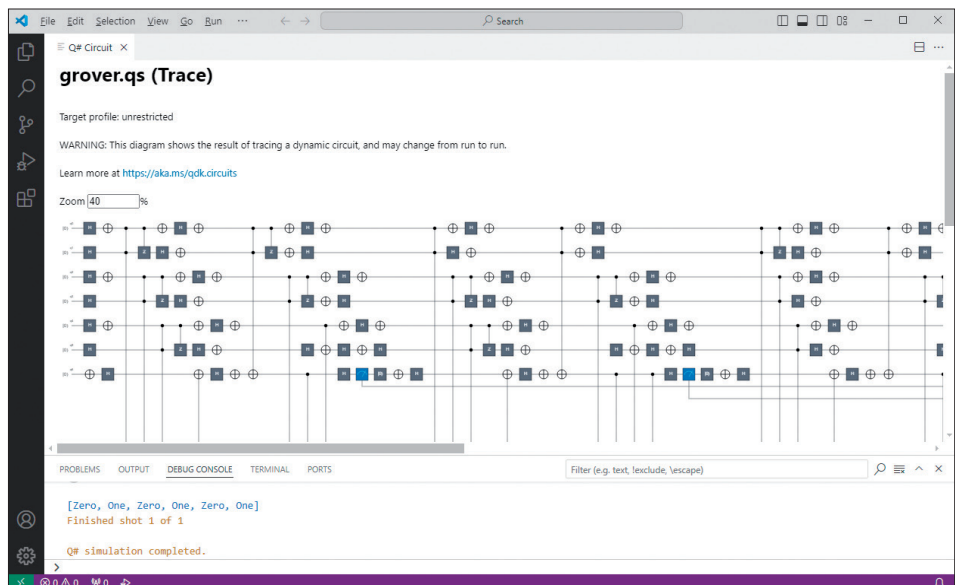
Dieser kleine Workshop wollte einen kurzen Überblick über die Programmiersprache Q# von Microsoft geben. Quantencomputing ist ein sich schnell entwickelndes Feld, das allerdings erst am Anfang steht und allmählich versucht, sein Potenzial nutzbar zu machen. Auch wenn Q# eine noch recht junge Programmiersprache ist, so kann sie durch ihre charakteristische Form überzeugen. Und durch die Nutzung von Quanteneigenschaften wie Superposition und Verschränkung besitzt das Quantencomputing das Potenzial, die Art und Weise zu revolutionieren, wie wir heute programmieren.

Q# umfasst noch viele weitere Bibliotheken, etwa für Machine Learning, Chemie und Mathematik. Werfen Sie doch einfach mal einen tieferen Blick auf die Entwicklung von Quantencomputing und der Programmiersprache Q#. Die Community freut sich über jedes neue Mitglied. ■

- [1] Mykola Dobrochynskyy, *Wegweisende Symbiose*, dotnetpro 2/2025, Seite 20 ff., www.dotnetpro.de/A2502QuantumCloud
- [2] *Shor-Algorithmus* bei Wikipedia, www.dotnetpro.de/SL2502QSharp1
- [3] *Grover-Algorithmus* bei Wikipedia, www.dotnetpro.de/SL2502QSharp2
- [4] *Visual Studio Code*, <https://code.visualstudio.com>



Das Ergebnis des Grover-Algorithmus (Bild 6)



Quantenschaltkreis von Grover (Bild 7)

- [5] QDK, www.dotnetpro.de/SL2502QSharp3
- [6] *Quantenverschränkung* bei Wikipedia, www.dotnetpro.de/SL2502QSharp4
- [7] *Bell-Zustand* bei Wikipedia, www.dotnetpro.de/SL2502QSharp5
- [8] Microsoft Learn, *Q#-API-Referenz*, www.dotnetpro.de/SL2502QSharp6
- [9] *Hadamard-Gatter*, www.dotnetpro.de/SL2502QSharp7
- [10] Microsoft Learn, *CNOT-Gatter*, www.dotnetpro.de/SL2502QSharp8
- [11] Microsoft Learn, *Type System*, www.dotnetpro.de/SL2502QSharp9
- [12] Microsoft Learn, *Expressions*, www.dotnetpro.de/SL2502QSharp10

● Listing 4: Lösen eines Suchproblems mit Q#

```

import Microsoft.Quantum.Convert.*;
import Microsoft.Quantum.Math.*;
import Microsoft.Quantum.Arrays.*;
import Microsoft.Quantum.Measurement.*;
import Microsoft.Quantum.Diagnostics.*;

operation Main() : Result[] {
    let nQubits = 6;
    let iterations =
        CalculateOptimalIterations(nQubits);
    Message($"Anzahl der Iterationen: {iterations}");
    let results = GroverSearch(nQubits, iterations,
        ReflectAboutMarked);
    return results;
}

operation GroverSearch(
    nQubits : Int,
    iterations : Int,
    phaseOracle : Qubit[] => Unit) : Result[] {

    use qubits = Qubit[nQubits];
    PrepareUniform(qubits);

    for _ in 1..iterations {
        phaseOracle(qubits);
        ReflectAboutUniform(qubits);
    }

    return MResetEachZ(qubits);
}

function CalculateOptimalIterations(nQubits : Int) :
    Int {
    if nQubits > 30 {
        fail "Dieses Beispiel unterstützt maximal 30
            Qubits.";
    }
    let nItems = 1 <<< nQubits; // 2^nQubits
    let angle = ArcSin(1. / Sqrt(IntAsDouble(nItems)));
    let iterations = Round(0.25 * PI() / angle - 0.5);
    return iterations;
}

}

operation ReflectAboutMarked(inputQubits : Qubit[]) :
    Unit {
    Message("Reflektieren über den markierten
        Zustand...");
    use outputQubit = Qubit();
    within {
        X(outputQubit);
        H(outputQubit);
        for q in inputQubits[...2...] {
            X(q);
        }
    } apply {
        Controlled X(inputQubits, outputQubit);
    }
}

operation PrepareUniform(inputQubits : Qubit[]) :
    Unit is Adj + Ctl {
    for q in inputQubits {
        H(q);
    }
}

operation ReflectAboutAllOnes(inputQubits : Qubit[]) :
    Unit {
    Controlled Z(Most(inputQubits), Tail(inputQubits));
}

operation ReflectAboutUniform(inputQubits : Qubit[]) :
    Unit {
    within {
        Adjoint PrepareUniform(inputQubits);
        for q in inputQubits {
            X(q);
        }
    } apply {
        ReflectAboutAllOnes(inputQubits);
    }
}

```

[13] Microsoft Learn, Einzel-Qubit- und Multi-Qubit-Pauli-Messungen, www.dotnetpro.de/SL2502QSharp11

[14] Microsoft Learn, Visualisieren von Quantenkreisdigrammen mit Q#, www.dotnetpro.de/SL2502QSharp12

[15] Microsoft Learn, Theory of Grover's search algorithm, www.dotnetpro.de/SL2502QSharp13

[16] Adjunktion bei Wikipedia, www.dotnetpro.de/SL2502QSharp14



Daniel Basler

arbeitet als Lead Developer und Softwarearchitekt. Seine Schwerpunkte liegen auf Cross-Plattform-Apps, Android, JavaScript und Microsoft-Technologien.

dnpCode

A2502QSharp

TIME-OUT

Zukunft wirft keine Schatten

Wenn wir unsere Prioritäten setzen, schauen wir vor allem auf das, was uns momentan bewegt. Wie es uns damit in der Zukunft geht, haben wir weniger gut im Blick ...



Wenn uns etwas besonders gut gefällt, dann sind wir außer uns vor Freude. Manche sagen auch: Ganz aus dem Häuschen. Neben sich stehen, außer sich sein, unsere Sprache kennt einige Begriffe dafür, wenn wir nicht „in unserer Haut stecken“, sondern die Außenperspektive einnehmen. Es fällt uns also anscheinend nicht sonderlich schwer, einen Blick von außen auf uns zu richten. Sie können dieses Experiment auch gerne einmal bei sich versuchen: Werfen Sie jetzt einen Blick auf sich, wie Sie die Zeitschrift in der Hand halten und sehr konzentriert lesen. Können Sie sich sehen? Voilà.

Wie wirke ich auf andere?

Wie sieht das aus, wenn ich tanze, Ski fahre oder lächle? Wie komme ich an? Wie wirke ich? Das scheint eine sehr wichtige Frage zu sein. Wir können uns nicht nur räumlich von außen betrachten, wir können uns auch erinnern, wie es war, als wir damals zum ersten Mal betrunken waren, an den ersten Kuss, das erste Mal im Auto alleine hinter dem Steuer. Auch das ist etwas, was wir Tieren und anderen Lebewesen vermutlich voraushaben. Diese Fähigkeit zur Abstraktion von der konkreten Situation hilft uns, über uns nachzudenken, das eigene Tun zu reflektieren und daraus die richtigen Schritte für die Zukunft abzuleiten.

Wie es uns jetzt geht, wie wir derzeit auf andere wirken, welche Erfahrungen wir in der Vergangenheit gemacht haben, all das bestimmt unsere Entscheidungen. So weit, so gut. Wenn es allerdings um die Zukunft geht, lässt uns unsere Fähigkeit, die Außenperspektive einzunehmen, im Stich. Und leider haben wir unsere Mängel nicht so stark im Fokus wie unsere Fähigkeiten. Einer davon ist, sich konkret und empathisch vorzustellen, wie es uns in Zukunft gehen wird. Wie werden wir in 20 Jahren unseren Körper spüren, wenn wir plötzlich nicht mehr so viel Kraft haben? Wie wird es uns in der Zukunft gehen? Was wird uns wichtig sein, und welche Bedürfnisse werden wir dann haben?

Es liegt in der Natur der Sache, dass wir diese Emotionen nicht so stark spüren wie den Kater von gestern. Und es ist offensichtlich, dass man nicht in die Zukunft schauen kann.

Gefühle wie Einsamkeit, die man vielleicht bisher noch nicht kannte, sind schwer vorstellbar, wenn man sie noch nie erlebt hat. Und all das hindert uns daran, diese Emotionen unsere derzeitigen Entscheidungen mitprägen zu lassen.

Wir laufen blind in die Katastrophe

Dabei sind viele Dinge absehbar, nur wollen wir sie nicht wahrhaben. Wir ärgern uns jetzt, dass der Handwerker nicht kommt. Was wird aber in zehn Jahren sein, wenn der Fachkräftemangel so weitergeht? Interessiert uns nicht! Wir ärgern uns, wenn wir beim Fußball nicht mehr so schnell sind wie die 20-Jährigen. Aber was wird mit uns in der Rente sein, wenn wir jetzt nicht regelmäßig Sport machen, sondern nur hinter dem PC sitzen und uns kaum mehr bewegen? Interessiert uns nicht! Wir erleben jetzt Stürme und Sintfluten, aber wie wird es uns gehen, wenn das erste Mal das Wasser in unserem Keller steht? Interessiert uns nicht!

Die zentrale Frage, wie es unserem zukünftigen Ich ergehen wird, ist uns völlig egal! Das zeigen die ein oder anderen Experimente, die Psychologen durchgeführt haben. Lassen Sie mich zwei davon zitieren: Bei einem ging es darum, ob die Probanden lieber jetzt zwei Löffel widerlich schmeckende Suppe zu sich nehmen oder in einer Woche eine ganze Schüssel. Raten Sie mal, wie das Ergebnis ausfiel ...

Bei einem anderen Versuch bot man 100 Euro sofort oder 110 Euro in einem Jahr an. Jeder Finanzmathematiker würde sich über eine zehnprozentige Verzinsung freuen – wohl wissend, dass die 110 Euro zwar wegen der Inflation nicht mehr 110 Euro wert wären, aber mehr als die 100 Euro heute. Wir hingegen denken vor allem an den Wunsch, den wir zurzeit haben und den wir uns mit den 100 Euro hier und jetzt erfüllen könnten. Zukünftige Bedürfnisse lassen wir hingegen gerne außen vor.

Stellen Sie sich einmal einen Wunsch vor, den Sie in Zukunft haben werden. Fällt Ihnen einer ein? Ich meine einen Wunsch, den Sie später haben könnten, nicht einen, den Sie jetzt im Hinblick auf die Zukunft haben. Und selbst wenn Ihnen einer einfällt, wird er ähnlich intensiv sein wie unsere

Sehnsucht im Hier und Jetzt? Wie der Wunsch nach der neuen Grafikkarte, der stylischen Espressomaschine oder dem wunderbaren Kurzurlaub? Sicher nicht.

Deshalb ist es auch kein Problem, jetzt zu fliegen, jetzt Schulden zu machen oder jetzt in die Kneipe zu gehen. Dass wir morgen peitschenden Wind im Gesicht haben werden, unter einem leeren Konto zu leiden haben oder eine wichtige Prüfung möglicherweise nicht bestehen werden – who cares?

Wir neigen nicht nur dazu, unangenehme Situationen in die Zukunft zu schieben, wir sehen oft nicht einmal, dass es unangenehm werden könnte.

Lieber den Spatz in der Hand ...

Das ständige Verschieben von Aufgaben in die Zukunft, die sogenannte Prokrastination, hat viele Ursachen: Versagensangst, unrealistische Ziele, zu viel Ablenkung. Eine davon ist unsere Unfähigkeit, sich Zukunft vorzustellen. Wie würde es uns gehen, wenn wir nicht gelernt haben und versagen? Wie würde es uns gehen, wenn wir uns auf uns verlassen können und brillieren können in der Prüfung? Könnten wir uns beides vorstellen, besser noch: könnten wir es fühlen, dann würden wir im Hier und Jetzt andere Schritte gehen.

Was machen wir aber stattdessen? Wir belohnen uns bereits vor der Herausforderung. Das hilft im Moment, macht aber nicht wirklich glücklich, befriedigt nicht, und so belohnen wir uns erneut. Es entsteht ein Teufelskreis.

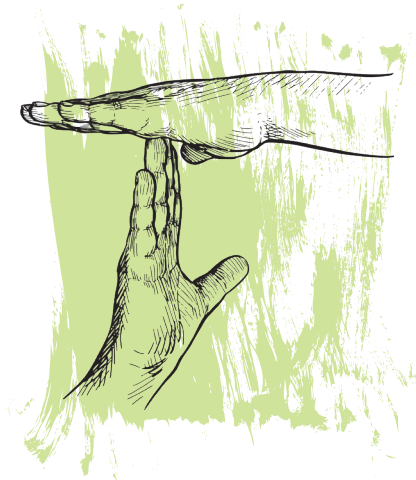
Weil wir die Zukunft nicht fühlen können, sprechen wir von Klimawandel statt von Klimakatastrophe, von Fachkräftemangel statt von Fachkräftedürre, vom Pflegemangel statt vom Pflegenotstand. Alles nicht so schlimm. Wird schon irgendwie werden. Wir richten es uns in der Gegenwart gemütlich ein, die Zukunft ist egal. Sie ist nicht dramatisch. Wir spüren sie nicht.

... oder die Innovation in der Zukunft

Sie werden einwenden, dass es in der Vergangenheit immer Umbrüche gegeben hat. Die Zukunft verläuft nicht linear, und so warten Sie lieber auf die eine Erfindung, die das CO₂ mit einem Schlag reduzieren kann. Theoretisch ist das möglich, keine Frage. Es mag auch sein, dass Sie in Zukunft statt eines Gebisses natürlich gewachsene Zähne implantiert bekommen, dass Sie mit einer Spritze schwache Muskeln aktivieren können oder mit einem implantierten (KI-)Chip fehlendes Wissen und Verständnis wettmachen.

Wir sprechen hier aber von ungewöhnlichen und disruptiven Prozessen mit einer gewissen Unwahrscheinlichkeit. Wir beginnen hier, uns anzulügen.

Unser kurzfristiges zukünftiges Ich ist ein gleichberechtigter Partner. Es ist einfach sehr unwahrscheinlich – nicht unmöglich –, dass man nächste Woche nicht mehr lebt. Aber ein Yolo sollte bis zum Ende des Lebens gelten. Es heißt ja nicht: You only live now!



Jetzt die Schritte gehen, um am Ende anzukommen

Verstehen Sie mich nicht falsch: Wir haben zu Recht eine Generation und Einstellung hinter uns gelassen, die ihr Leben lang gearbeitet hat, um dann kurz nach der Rente den Herzinfarkt zu bekommen. Wir wollen und sollen im Hier und Jetzt glücklich werden. Aber das morgige Hier und Jetzt sollte zumindest einen legitimen Platz bekommen. Denn wir wollen ja auch morgen glücklich sein. Oder nicht?

Unser Denken und Fühlen, was die Zukunft betrifft, ist jedoch limitiert. Was also können wir tun? Weil wir die Zukunft so schlecht einbeziehen können in unser Fühlen, sollten wir unserem Hirn hin und wieder auf die Sprünge helfen und uns vorstellen, wie sich eine Entscheidung für die Zukunft auswirken könnte. Bewusst nachspüren und fühlen, wie es uns dann gehen wird.

Ich denke, Sie können das in manchen Fällen schon. Wenn Sie ein Haus kaufen wollen, um eine Familie zu gründen, überlegen Sie sich auch, wie es sein wird, mit den Kindern im neuen Wohnzimmer zu spielen. Das gibt viel Kraft für manch stressige Phase während der Bauzeit oder wenn das Kind die ganze Nacht schreit. Und das Gleiche sollten wir auch für andere Lebensbereiche tun. Es hilft uns, mehr Sport zu machen, Geld anzulegen, Beziehungen zu pflegen oder zu lernen, einen sozialen Dienst zu übernehmen. Denn all das zahlt sich am Ende aus.

Wie hieß es bei einer Lebensversicherung einmal: „Früher schon an später denken.“ Ich würde das gerne erweitern: Stellen Sie sich vor, welche Bedürfnisse und Gefühle Sie in Zukunft haben werden, und seien Sie nett zu ihrem zukünftigen Ich. Gleichen Sie diesen natürlichen Mangel, den wir mitbekommen haben, aus. Dann haben Sie mehr von Ihrem Leben. Ein echtes Yolo.

In diesem Sinne: Happy Coding!



Time-out End



Dipl.-Theol. Bernhard Pichler

war 20 Jahre lang Geschäftsführer der Softwarefirma informare, bevor er 2020 zum Leiter Entwicklung und Produktmanagement der DPS BS ernannt wurde. Die DPS BS ist Deutschlands größter Softwarepartner für betriebswirtschaftliche Software für den Mittelstand.

dnpCode

A2502Timeout

MIT MODULITHEN ALLE WEGE OFFENHALTEN

Pragmatisch Richtung Microservices

Wie entwickelt man Monolithen, ohne sich die Option zu verbauen, auf Microservices zu wechseln, und wie entwickelt man Microservices, ohne dabei in Fallen zu tappen?

Bereits 2015 schrieb Martin Fowler in seinem Blog, dass die erfolgreichsten Microservice-Architekturen, die er kennengelernt habe, mit einem Monolithen starteten, während nahezu alle Teams, die direkt mit Microservices begannen, in große Schwierigkeiten gerieten [1].

In diesem Artikel wollen wir dieser Aussage auf den Grund gehen und aufzeigen, wie uns das Architekturmuster des Modulithen dabei helfen kann, auch schon bestehende Monolithen zu restrukturieren, damit sie von Vorteilen profitieren können, die sonst eher den Microservices zugesprochen werden. Hierbei gehen wir jedoch nicht allzu sehr in die Tiefe der Theorien der verschiedenen Architekturmuster, da diese in der *dotnetpro* bereits ausführlich beschrieben wurden [2]. Wir werden uns vielmehr damit befassen, den Prozess zu klären, der notwendig ist, um die beschriebenen Muster angemessen einzusetzen und umzusetzen.

Einige Grundlagen

Bevor wir nun aber in den besagten Prozess einsteigen, wollen wir dann doch noch einmal die wichtigsten Grundlagen wiederholen. Allerdings ist dies gar nicht so einfach, denn über die Jahre hinweg haben sich durchaus verschiedene Ausprägungen der gleichen Definitionen für die unterschiedlichen Architekturmuster etabliert. Gemein ist ihnen, dass es sich bei Monolithen um Systeme handelt, bei denen alle Bestandteile integral für das Gesamtsystem sind. Man kann also keinen Bestandteil weglassen, ohne das System als Ganzes zu gefährden.

Je nachdem, wo die entscheidenden Abhängigkeiten liegen, sprechen wir von:

- **Development Monolith** oder **Mono Repo** [3]: Dies bezeichnet eine Codebasis, in der der gesamte Code einer Anwendung in einem einzigen Repository oder Projekt verwaltet wird. Diese Struktur kann die Entwicklung und Zusammenarbeit in sehr großen Projekten erschweren, da Änderungen an einem Teil des Codes potenziell Auswirkungen auf andere Teile haben können. Zudem können Build- und Testzeiten mehr Zeit benötigen, was die Entwicklungsproduktivität beeinträchtigt.
- **Logical Monolith** [4]: Dies beschreibt eine Software, bei der die Geschäftslogik nicht modularisiert, aber stark miteinander verflochten ist. Obwohl die Architektur möglicherweise in verschiedene Module oder Schichten unterteilt ist,

sind diese nicht ausreichend entkoppelt, was zu Schwierigkeiten bei Wartung, Erweiterung und Testing führt. Solche Architekturen neigen dazu, über die Zeit an Komplexität zuzunehmen, und werden schwer verständlich und schwer änderbar.

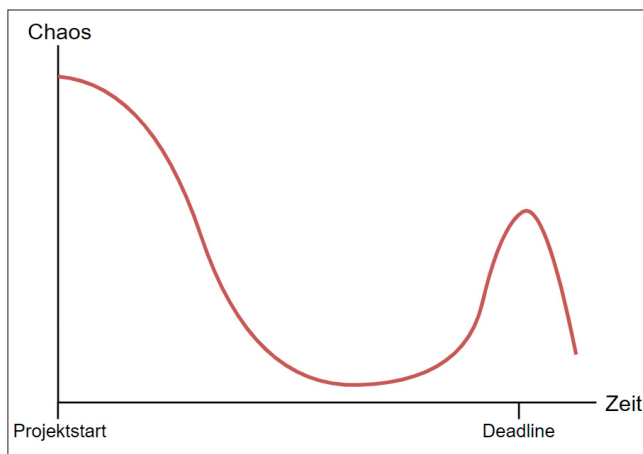
- **Architectural Monolith**: Dies bezeichnet ein System, bei dem die gesamte Architektur um einen einzigen Technologie-Stack oder ein einziges Framework herum aufgebaut ist. Das kann die Flexibilität einschränken, da die Einführung neuer Technologien oder Paradigmen schwierig wird. Zudem kann es zu Abhängigkeiten führen, die das System anfällig für Technologiealterung machen.
- **Runtime Monolith** [5]: Eine Anwendung, die zur Laufzeit als ein einziger Prozess oder eine einzige Einheit ausgeführt wird. Unabhängig davon, wie der Code strukturiert oder entwickelt wurde, wird die Anwendung während der Ausführung nicht in separate, unabhängige Dienste oder Prozesse aufgeteilt. Dies kann zu Skalierbarkeits- und Wartbarkeitsproblemen führen, da die gesamte Anwendung skaliert werden muss, selbst wenn nur ein Teil davon erhöhte Ressourcen benötigt.
- **Deployment Monolith**: Ein Softwaresystem, bei dem alle Komponenten als eine einzige Einheit bereitgestellt und veröffentlicht werden. Änderungen in einer einzelnen Komponente erfordern oft das erneute Deployment des gesamten Systems.

Monolith < Microservice?

Dies klingt jetzt sicher alles recht negativ, und lange Zeit wurden Monolithen auch so behandelt, als wären sie den Microservice-Architekturen weit unterlegen. Letztere stehen nicht zu Unrecht im Ruf, besonders flexibel, skalierbar, zuverlässig und wartbar zu sein. Immerhin sind sie so aufgebaut, dass sie statt aus einem großen Ganzen aus vielen kleineren, in sich geschlossenen Einheiten bestehen, die unabhängig voneinander deployt werden können und ihre Arbeit auch dann noch fortsetzen, wenn einmal der eine oder andere Dienst des Systems ausfällt. Und nicht nur das. Durch die Abgeschlossenheit der Dienste versprechen sie den Teams auch die Möglichkeit, immer auf den genau passenden Technologiestack zurückgreifen zu können, was die Produktivität steigert und eine bessere Ressourcen-Effizienz verspricht. Doch wenn das alles so toll ist, warum schrieb Fowler dann in seinem Artikel

von großen Herausforderungen, und warum beschwört er Monolithen als die besten Keimzellen für Microservices? Um dies zu verstehen, müssen wir uns von der Betrachtung einzelner Softwareapplikationen und Dienste hin zu einer Betrachtung des kompletten Softwaresystems sowie dessen Lebenszyklus bewegen. Was für eine kleine Anwendung und ein einzelnes Team stimmt, muss nicht auch gleich für die Summe vieler Applikationen und Teams zutreffen.

Bild 1 zeigt stark vereinfacht einen Zusammenhang, der sich in jedem neuen Projekt beobachten lässt. Gerade zu Beginn herrscht großes Chaos. Alle Beteiligten müssen sich gegenseitig, die Domäne und die einzusetzenden Werkzeuge kennenlernen. Eventuell muss sogar erst einmal geklärt werden, wer denn überhaupt am Projekt beteiligt werden soll und welche Technologien zum Einsatz kommen. Dieses Chaos verringert sich über die Zeit hinweg, wenn die Projektbeteiligten zunehmend eingespielt sind, die Rahmenbedingungen



Chaos im Projekt im zeitlichen Verlauf betrachtet (**Bild 1**)

klarer sind und es Dinge gibt, auf denen man aufbauen kann. Erst gegen Ende, zur Deadline hin, steigt dann das Chaoslevel wieder an, denn dann fällt all das auf, was man zuvor nicht bedacht hat, zumal natürlich auch Budget- und Zeitfragen viel drängender werden.

Um langfristig erfolgreich mit Microservices zu arbeiten und zu vermeiden, dass ein „verteilter Monolith“ entsteht – also eine Sammlung von Diensten, die so stark voneinander abhängen, dass sie nur als Ganzes funktionieren –, ist eine gründliche Planung und Analyse der Architektur unerlässlich. Dazu gehört, klare fachliche Grenzen zu definieren, die sich auch in der Teamstruktur widerspiegeln. Ebenso müssen Deployment-Strategien, Laufzeitumgebungen und Entwicklungsprozesse sorgfältig etabliert werden. Wichtige Detailentscheidungen erfordern ein tiefes Verständnis der jeweiligen Domäne, um sicherzustellen, dass die Microservices effektiv und unabhängig voneinander funktionieren. Natürlich sind das alles Entscheidungen, die man auch bei einem Monolithen treffen muss, und Teams, die in der Vergangenheit mit Microservices gearbeitet haben, werden sich hier wahrscheinlich leichter tun als Teams, die zuvor nur an Monolithen

gearbeitet haben. Durch ihre höheren Anforderungen an das Domänenverständnis, das Deployment und die eingesetzten Werkzeuge besitzen Microservices aber etwas, dass Fowler in einem anderen Artikel [6] als „Microservice Premium“ bezeichnet. Hierbei handelt es sich um eine Art Preisaufschlag oder Risikoaufschlag, den man einkalkulieren muss, wenn man dieses Architekturmuster einsetzt. Tut man es nicht, könnte die Spitze am Ende des Graphen aus **Bild 1** wesentlich größer ausfallen.

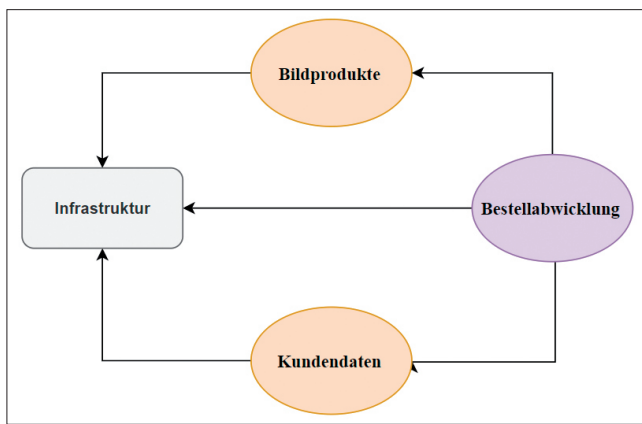
Der Aufschlag zeigt sich aber gegebenenfalls auch noch anders, wie beispielsweise Amazon vor einiger Zeit im Kontext des eigenen Dienstes Prime Video erfahren musste. So zeigte Mitte 2023 ein Amazon-Techniker in einem mittlerweile offline genommenen Blogpost [7], wie hoch die Betriebskosten einer stark verteilten Lösung werden können, da natürlich alle zwischen den Diensten übertragenen Daten auch monetär zu Buche schlagen. Die Lösung im Fall Amazons war dann auch, die vielen Dienste in einen Monolithen zusammenzufassen und damit den Datenstrom innerhalb eines einzelnen Prozesses zu bündeln.

Dies bedeutet nun aber wiederum nicht, dass Microservices grundsätzlich schlecht sind. Einen sehr großen Einfluss auf den Erfolg eines Projekts hat natürlich das Vorwissen der Projektbeteiligten. Je weniger sie neu lernen müssen, desto verlässlicher sind ihre Arbeitsergebnisse. Darüber hinaus muss sowohl bei Monolithen als auch Microservices der Einsatz gut geplant und den Aufgaben gegenüber angemessen sein. Wie Fowler schreibt, kann man einen Wechsel der Architekturen aber auch gut nutzen, um sich zunächst an die angemessene Komplexität des Zielsystems heranzutasten. Man beginnt also mit einem Monolithen, aber strukturiert diesen im Code und der Architektur so, dass seine Bestandteile später auch einfach als eigenständige Applikationen deployt werden könnten. Dazu nutzt man beispielsweise bestimmte Namensräume, achtet auf eindeutige und entkoppelte Schnittstellen, oder um es einfacher zu sagen: Man baut einen modularen Monolithen, wie er im Lehrbuch steht.

Der Modulith

Der Modulith ist im Grunde nichts anderes als der zuvor angesprochene modulare Monolith. Einige Autoren beschreiben ihn auch als die Art, Monolithen zu bauen, wie sie einmal gedacht waren und bevor sie ihren schlechten Ruf erhalten haben. Ein Modulith integriert die strukturelle Kohärenz eines Monolithen, während er gleichzeitig klar abgegrenzte Module nutzt. Diese Module kapseln ihre eigene Domänenlogik und können unabhängig entwickelt werden, was eine erhöhte Flexibilität bietet. Sie folgen dabei den gleichen Grundideen und Methodiken, die auch bei Microservices eingesetzt werden. Dabei vermeiden sie aber deren strukturelle Komplexität, die beispielsweise durch Konfiguration, Netzwerkmanagement und Orchestrierung entsteht. All dies hält die Herausforderungen für Teams in einem erträglichen Rahmen, damit sie die Domäne verstehen, Mehrwert für die Auftraggeber liefern können und erst an Verteilung denken müssen, wenn jene durch entsprechende nichtfunktionale Anforderung erzwungen wird. ►

Wie bereits erläutert besteht ein Modulith aus Modulen. Jedes Modul bildet eine unabhängige Komponente, die eine bestimmte Domäne kapselt und über klar definierte Schnittstellen mit anderen Modulen kommuniziert. Diese Schnittstellen minimieren die Abhängigkeiten zwischen den Modulen und ermöglichen weitgehende Unabhängigkeit, sodass Entwicklerteams an ihren Modulen arbeiten können, ohne die Funktionsweise anderer Module zu beeinflussen. Gleichzeitig wird eine hohe Kohäsion innerhalb der Module angestrebt. Dies bedeutet, dass die interne Logik eng miteinander verbunden ist und eine klare Verantwortung trägt. Dies steigert die Wartbarkeit und erleichtert es, neue Funktionen hinzuzufügen oder bestehende zu ändern. Klare Schnittstellen



Ein möglicher Modulschnitt als Context Map (Bild 2)

sind essenziell, um eine kohärente Kommunikation zwischen den Modulen zu gewährleisten. Diese sollten die internen Implementierungen verdecken und die fachlichen Zusammenhänge der jeweiligen Domäne widerspiegeln.

Bei der technischen Umsetzung eines Modulithen werden unterschiedliche Kommunikationsstrategien eingesetzt. Meistens nutzen Module Methodenaufrufe über die öffentlichen Schnittstellen. Da alle Module im selben Prozess laufen, vermeiden sie die sonst üblichen Latenzen und Sicherheitsbedenken, die im Rahmen von Netzwerkkommunikation bei serviceorientierten Architekturen entstehen. Alternativ können Module auch asynchrone Nachrichten über interne Messaging-Systeme austauschen. Dies ist beispielsweise dann hilfreich, wenn sie später als Teil von eigenen Diensten bereitgestellt werden sollen, die ebenfalls ein Messaging-System nutzen.

Dank ihres modulzentrischen Aufbaus erleichtern Modulithen zudem die Testbarkeit. Da Module isoliert getestet werden können, lassen sich Unit-Tests effizient durchführen. Aber auch Integrationstests werden einfacher, können diese doch durch die Modul-zu-Modul-Kommunikation als etwas umfangreichere Unit-Tests gestaltet sein und brauchen nicht zwangsläufig Interprozess- oder Netzwerkkommunikation. Dadurch können die Tests und Build-Pipelines wiederum schneller Ergebnisse liefern, und die allgemeine Entwicklerproduktivität steigt.

Theoretische Umsetzung

Modulithen scheinen in Summe also ein großartiges Konzept zu sein, aber wie setzt man sie nun konkret um? Dies hängt natürlich stark vom jeweiligen fachlichen Kontext und den damit verbundenen Anforderungen ab. Einen Modulithen für Desktop-Applikationen zu erstellen ist schon seit vielen Jahren dank Frameworks wie Prism recht einfach möglich [8]. Aber Achtung, hier haben sich in den vergangenen Jahren die Lizenzbedingungen deutlich verändert, und gerade kommerzielle Lizenzen kosten mehrere Hundert Euro pro Entwickler. Da es außerdem für Webapplikationen kein Framework mit vergleichbarem Funktionsumfang und Reichweite gibt, klären wir nachfolgend die notwendigen Grundlagen, um jede Form von .NET-Applikation möglichst einfach mit Bordmitteln als Modulith umzusetzen.

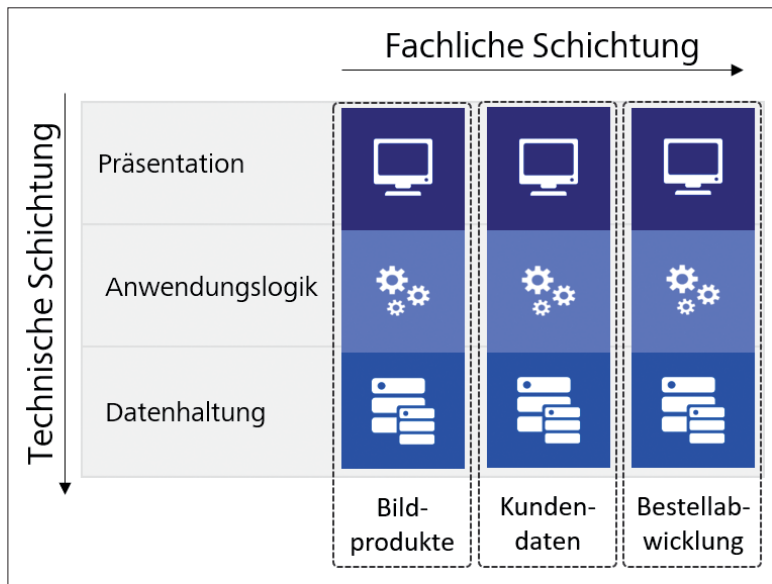
Bevor wir einen Modulithen bauen können, müssen wir, ähnlich wie im Fall einer Microservice-Architektur, ein grobes Verständnis für unsere fachlichen Zusammenhänge haben. Das erreichen wir in Form von fachlichen Kontexten. Diese beschreiben abgegrenzte und fachlich in sich schlüssige Anwendungsbereiche, in denen bestimmte Begriffe und Konzepte jeweils eine spezifische Bedeutung haben. Damit definieren sie klare Grenzen für logische Modelle und inhaltliche Verantwortlichkeiten.

Im Nachfolgenden wollen wir ein System modellieren, über das man Fotos bestellen kann. Wie in Bild 2 zu sehen ist, haben wir dafür drei fachliche Kontexte ausgemacht:

- **Bildprodukte:** Hierin wird alles abgedeckt, was wir für die Kunden als Produkt im Zusammenhang mit Bildern anbieten. Das können einzelne Fotos sein, aber auch bedruckte Tassen oder Kissen.
- **Kundendaten:** Dies sind die Stammdaten unserer Kunden, also Adressen, Zahlungsmittel und dergleichen.
- **Bestellabwicklung:** Hierin sind alle Prozesse, Dialoge und so weiter zentriert, die notwendig sind, um Bestellungen von Bildprodukten durch entsprechende Kunden zu realisieren.

Neben den Kontexten zeigt uns die Context Map aus Bild 2 aber noch mehr. Karten dieser Art werden genutzt, um auch die Abhängigkeiten zwischen den Fachlichkeiten sowie deren Priorisierung zu verdeutlichen. Darin sehen wir, dass die Bestellabwicklung später auf die Funktionen der Bildprodukte und Kundendaten zugreifen wird, um die eigentlichen Bestellprozesse umzusetzen, während jene Bereiche nur von der Infrastruktur abhängen – diese ist eckig gezeichnet, da es sich bei ihr um keine Fachlichkeit handelt.

Nachdem wir nun die Kontexte kennen, widmen wir uns als Nächstes der Infrastruktur. Diese enthält alle Querschnittsaspekte, die für alle Module gleich sein müssen. Dazu zählen Dinge wie Autorisierung und Authentifizierung, Ressourcenverwaltung, Logging und Monitoring, aber auch die Logik zur Aggregation der Module sowie die Bereitstellung eines Applikationsrahmens, in den die Module ihre eigenen Bestandteile einhängen können. Um es sich besser vorstellen zu können, kann man die Infrastruktur als den Kern der Applikation betrachten, in den die Module wie eine Art Plug-ins oder Add-ins beim Programmstart eingehängt werden. Da



Fachlicher Schnitt versus technischer Schnitt (Bild 3)

der Kern für alle Module gleich ist, muss er besonders stabil gestaltet sein und darf sich nicht unkontrolliert verändern. Daher sollte er in einem eigenen Repository entwickelt und nur über eine Paketverwaltung eingebunden werden, um Verschmutzungen mit Domain-Logik zu vermeiden.

Kernels dieser Art funktionieren langfristig am besten, wenn sie von Beginn an so gestaltet sind, dass man sie theoretisch auch außerhalb ihrer Ursprungsdomäne für jede Art von Applikation verwenden könnte. In jedem Fall muss vermieden werden, dass sie zu einer Art Ablage für alles werden, was man nicht in irgendeinem Modul unterbekommt. Tut man dies nicht, wird die Plattform selbst immer komplexer, was wiederum die Stabilität aller Applikationen beeinträchtigt, die auf sie aufsetzen.

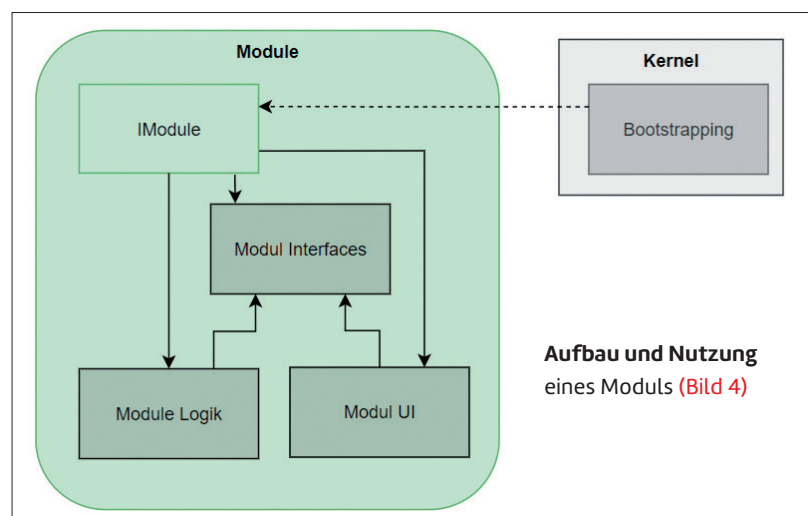
Das Modul

Die Definition eines Softwaremoduls ist schwieriger, als es zunächst den Anschein hat, denn es kann durchaus verschiedene Perspektiven auf sie geben. Zunächst ist ein Softwaremodul immer eine in sich geschlossene, wiederverwendbare Einheit von Quellcode oder Logik, die eine spezifische Funktionalität kapselt. In der Praxis können wir zwei verschiedene Arten von Modulen unterscheiden: fachliche Module und technische Module. Fachliche Module sind sehr stark durch die fachlichen Zusammenhänge eines fachlichen Kontexts getrieben und können durchaus Logik unterschiedlicher technischer Ebenen enthalten. Demnach also Teile des User-Interfaces, der Verarbeitungslogik und des Datenzugriffs. Technische Module bilden wiederum keine fachlichen Zusammenhänge ab und lassen sich nur einzelnen fachlichen Schichten zuordnen. Sie liefern dann beispielsweise nur bestimmte Dialoge, Algorithmen oder die Infrastruktur,

um überhaupt auf eine Datenbank zuzugreifen. Technische Module werden dann auch oft mit reinen Funktionssammlungen oder sogar DLLs beziehungsweise Assemblies gleichgesetzt. Microsoft selbst meint beispielsweise in der MSDN fast immer technische Module, wenn der Begriff dort verwendet wird. Da diese aber eher technische Bibliotheken oder Frameworks sind, liegen sie außerhalb dessen, was wir für unseren Modulithen bevorzugen. Dies heißt nicht, dass wir keine technischen Module brauchen. Vielmehr ist es so, dass wir vor allem fachlich denken müssen, um die wirklichen Vorteile des Modulithen nutzen zu können und ihm gegebenenfalls eine Aufteilung in mehrere Microservices zu erlauben.

Bild 3 stellt die Zusammenhänge noch einmal etwas klarer dar. Die fachlichen Module bestehen aus mehreren Bestandteilen über unterschiedliche technische Schichten hinweg. Dabei kann jede technische Schicht ein eigener Namensraum oder sogar eine eigene Assembly sein. Damit die

se nun korrekt in den Applikationsrahmen der Infrastruktur eingehängt werden können, benötigen wir noch eine Modulkonfiguration. Sie wird bei einem Programmstart ausgelesen, und für jedes einzelne Modul werden die passenden Methoden für die einzelnen Schritte im Startablauf aufgerufen. **Listing 1** zeigt hierbei ein weit verbreitetes Interface für Module dieser Art. Diese haben eine Methode, mit der sie ihren eigenen Typ bei einem IoC-Container registrieren können, und eine weitere, mit der sie auf den Container zugreifen, um ihre eigenen Funktionen zu erfüllen. Im Code wurden hierbei Platzhalter für die Registrierung und den Container verwendet, da beide von der jeweiligen Container-Implementierung abhängen. Die Idee des Ablaufs ist dabei, dass zunächst alle Module ihre Typen registrieren und im zweiten Schritt alle Module auf Basis der registrierten Typen mit ihrer Arbeit beginnen. Welche Module konkret zu laden sind, kann dann von der Infrastruktur entweder mithilfe von Reflection ermittelt werden, indem man beispielsweise alle Assemblies in ►



Aufbau und Nutzung eines Moduls (Bild 4)

einem bestimmten Verzeichnis durchsucht, oder indem die entsprechenden Konfigurationen in einer JSON-, XML- oder YAML-Datei eingetragen werden, die das Framework beim Programmstart ausliest.

All das mag im ersten Moment eventuell etwas komplex erscheinen, kann dank der entsprechenden Standardfunktionalität aus dem .NET Framework aber recht einfach und mit wenigen Zeilen Code umgesetzt werden.

So viel zur technischen Initialisierung. Aber was macht ein Modul nun anders als eine einfache Assembly oder DLL? Es trennt seine Schnittstellen deutlich von seiner Implementierung. Dies kann ebenfalls über einen Namensraum geschehen oder sogar physisch, indem die öffentlichen Schnittstellen als eigenständige DLLs abgelegt werden. **Bild 4** zeigt einen hierfür typischen Aufbau. Welchen Weg man geht, ist dabei davon abhängig davon, wie hoch die Wahrscheinlichkeit ist, dass es für die Schnittstellen eines Moduls mehrere Implementierungen gibt, und wie viele Module insgesamt zu erwarten sind. Denn es kann durchaus passieren, dass mehrere Dateien pro Modul zu entsprechend vielen DLLs und damit auch Projekten in der Visual-Studio-Solution führen. Dies kann man wiederum umgehen, indem man jedes Modul in einer eigenen Solution implementiert und sie dann in einer

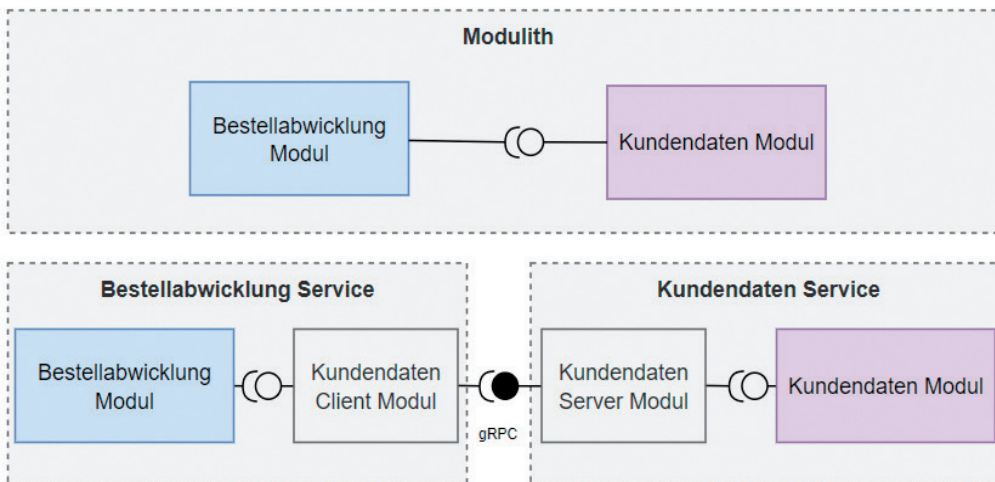
Listing 1: Ein Modulinterface

```
public interface IModule
{
    void RegisterTypes(ITypeRegistry typeRegistry);
    void Initialize.IContainerProvider
        containerProvider);
}
```

Schnittstellen, um beispielsweise Proto-Buf genannte Servicebeschreibungen für gRPC-Endpunkte oder dergleichen bereitzustellen. Dies ist entweder möglich, indem man die Schnittstellen per Hand in eine Proto-Beschreibung überträgt, oder indem man jene automatisiert aus den C#-Klassen und -Interfaces generiert. gRPC bietet sich hierbei besonders an, da es eine Form der Interprozesskommunikation darstellt, die von ihrer Natur her den direkten Zugriffen auf Methoden und Klassen sehr nahe kommt. Dazu werden aus den Proto-Beschreibungen Client- und Server-Klassen generiert, die dann wiederum im Client oder Server genutzt werden können,

um Funktionalität bereitzustellen oder abzufragen. Leider entsprechen die Schnittstellen dabei aber nicht wirklich eins zu eins den C#-Klassenhierarchien, wie man sie aus .NET kennt, weshalb zwischen die Aufrufe der gRPC-Logik auch noch eine Konvertierungsschicht gezogen werden muss.

Hat man diese Hindernisse aber genommen, können die Logikmodule wieder arbeiten wie zuvor. Nutzt also ein Modul in seiner Logik Features eines anderen Moduls, können nach wie vor die gleichen



Nutzung der gleichen Module im Modulithen und in Services (Bild 5)

Solution für das jeweilige Softwareprodukt zusammenführt. Dies ermöglicht es sogar, verschiedene Ausprägungen der gleichen Software bereitzustellen, was besonders bei Individualsoftware oder Software-Customization von großer Bedeutung ist.

Vom Modulithen zu Services

Das Versprechen des Artikels war aber nicht nur der Aufbau eines Modulithen, sondern auch die Migration zu Diensten. Wie kann man also das bewerkstelligen? Wenn die Applikation eine entsprechende Komplexität erreicht hat und eine Verteilung tatsächlich einen Mehrwert bietet, können Module mit überschaubarem Aufwand in eigenständige Dienste verwandelt werden. Hierzu nutzt man dann ihre öffentlichen

Interfaces angesprochen werden. Nur werden diese dann nicht auf die Logik des eigentlichen Zielmoduls übersetzt, sondern zunächst in eine Konverterschicht, die dann in die Client-Implementierung der gRPC-Schnittstelle greift, welche daraufhin die Anfrage über das Netzwerk an den Zielservice und zurück überträgt.

Dies wird in **Bild 5** verdeutlicht. Zunächst ist darin der Zugriff von Domänenmodulen in einem Modulithen gezeigt, um darunter das gleiche System, aber mit Diensten darzustellen. Wie man hierbei auch gut erkennen kann, bleiben die zentralen Logikmodule erhalten, während die Adapterlogik für die Interprozesskommunikation als rein technische Module über dieselbe Infrastruktur in die Prozesse geladen wird. Dies ist wiederum möglich, da sich alle Prozesse die gleiche Infra-

struktur teilen. Alternativ zu dem sehr konkreten Ansatz könnte man aber auch den Weg gehen, dass die Module sich innerhalb des gleichen Prozesses über eine Art prozessinternen Messagebus austauschen. Dessen Schnittstellen können dann bei der Prozessaufteilung auf ein anderes Messaging-System wie Azure Service Bus, RabbitMQ oder MassTransit umgestellt werden, was eine Event-Driven-Architektur ermöglicht.

Bleibt noch die Frage, wie man mit dem UI umgeht. Benutzerschnittstellen im gleichen Prozess in eine Art Rahmen einzuhängen ist ja noch einfach. Was passiert aber, wenn sie aus unterschiedlichen Prozessen stammen? In .NET ermöglichen Razor Pages und Minimal APIs die Implementierung von Web-Endpunkten, die als API-Gateways für bestimmte Module fungieren und damit die notwendigen Micro-UIs bereitstellen können. Diese werden dann in ein entsprechendes Host-UI eingebunden und somit vom Endanwender nicht mehr als gesonderte Webseiten wahrgenommen. Andere UI-Frameworks wie React, VueJs und Angular kennen die gleichen Prinzipien, wobei sie hier teils sogar noch stärker ausgereift sind.

Migration

Das Vorgehen scheint im ersten Moment etwas aufwendig zu sein, lässt sich aber leichter umsetzen, als man zunächst glauben mag. So kann man im ersten Schritt damit starten, die eigenen Projekte fachlich orientiert zu strukturieren und in eigenen Namensräumen abzulegen. Anschließend sollte man darauf achten, dass die Logik der Projekte nicht direkt, sondern nur über öffentliche Schnittstellen aufgerufen wird. Diesen Umstand kann man beispielsweise auch durch statische Codeanalyse erzwingen. Im nächsten Schritt kann man sich dann eine einfache Klasse für das Laden des Moduls als Infrastruktur bauen und von dort aus die Infrastruktur schrittweise gesondert behandeln. Damit wächst in kleinen Schritten die Infrastruktur unabhängig von der Modullogik. Erst beim Übertrag in die eigentlichen Dienste ist dann ein größerer Schritt notwendig, da die Module aus ihrem ursprünglichen Korsett heraus in eigene Laufzeitumgebungen transferiert werden müssen. Bis dies passiert, sollte das Softwaresystem aber schon so weit gewachsen sein, dass es den Kunden gegenüber einen gewissen Mehrwert stiftet und die Umbaumaßnahmen auch finanzieren kann.

Daran sieht man, dass das Verfahren auf eine inkrementelle Restrukturierung setzt, bei der man mit einem Monolithen startet, dessen Bestandteile in entkoppelten und unabhängigen Modulen abgelegt sind. Davon ausgehend kann man die Module dann noch weiter entkoppeln, indem sie beispielsweise in eigenen Solutions umgesetzt und erst später in eigenen Diensten bereitgestellt werden.

Tatsächlich lässt sich das Verfahren aber auch sehr gut anwenden, um bestehende Monolithen zu restrukturieren. Dabei hat es sich in der Praxis bewährt, zunächst das bestehende System mit einer Infrastruktur auszustatten, die es erlaubt, die Module strukturiert beim Programmstart zu initialisieren. Im nächsten Schritt wird die ursprüngliche Logik in Form eines einzelnen Moduls geladen und kann dann schrittweise in

weitere Module zerlegt werden. Dies hilft beispielsweise dabei, umfangreiche Legacy-Systeme nach und nach zu zerlegen, ohne dabei Gefahr zu laufen, dass die einzelnen Bestandteile nicht mehr zusammenpassen.

Fazit

Der Modulith ist ein hervorragendes Architekturmuster, mit dem man sich schrittweise an die tatsächlich notwendige Komplexität eines Softwaresystems herantasten kann. Es spart Entwicklungskosten, da man vergleichsweise schnell lauffähige Applikationen ohne umfangreiches Konfigurationsmanagement und Deployment bereitstellen kann. Zur Laufzeit kann es darüber hinaus auch Kosten einsparen, da die Kommunikation der Applikationsbestandteile internalisiert wird und somit keinen unnötigen Traffic verursacht. Damit eine Migration vom Modulithen zu Microservices möglich ist, müssen die Module aber insbesondere auch fachlich geschnitten sein und dürfen keine allzu großen Abhängigkeiten zueinander besitzen. Darüber hinaus ist das Architekturmuster sicher nicht immer und überall die beste Wahl, da es aufgrund seiner Geschlossenheit gerade im Cloud-Kontext nicht alle der dort üblichen Features und Optimierungsmöglichkeiten ausnutzen kann. Es ist somit sehr wichtig, im Vorfeld zu klären, welche langfristige Strategie und welche Qualitätsanforderungen an das Softwaresystem bestehen. ■

- [1] Martin Fowler, *Monolith First*,
www.dotnetpro.de/SL2502Modulithen1
- [2] Hendrik Lösch, *Mono? Micro? Macro? Modulo?*,
dotnetpro 5/2021, Seite 26 ff.,
www.dotnetpro.de/A2105Modulith
- [3] Git Blog, *Best Architecture for Dev Collaboration: Monorepo vs. Multi-Repo*,
www.dotnetpro.de/SL2502Modulithen2
- [4] Vaughn Vernon, *Domain-Driven Design Distilled*,
Addison-Wesley, 2016, ISBN 978-0-13-443442-1
- [5] Eberhard Wolff, *Microservices: Grundlagen flexibler Softwarearchitektur*, dpunkt.verlag, 2015,
ISBN 978-3-86490-313-7
- [6] Martin Fowler, *Microservice Premium*,
www.dotnetpro.de/SL2502Modulithen3
- [7] Joab Jackson, *Return of the Monolith: Amazon Dumps Microservices for Video Monitoring*,
www.dotnetpro.de/SL2502Modulithen4
- [8] Hendrik Lösch, *WPF modulo Prism*, *dotnetpro* 6/2021,
Seite 82 ff., www.dotnetpro.de/A2106ModulithPrism



Hendrik Lösch

ist Consultant und Architekt der ZEISS Digital Innovation. Sein Schwerpunkt liegt auf der Entwicklung und Bewertung von Software auf Basis von Microsoft-Technologien. Darüber hinaus ist er Sprecher und Autor.

hendrik.loesch@zeiss.com

dnpCode

A2502Modulithen

ARTEN VON SCHULDEN

Nicht nur in der Technik ...

Wenn wir in der Softwareentwicklung von Schulden reden, werden meist die technischen Schulden genannt. Aber das ist nur die Spitze des Eisbergs.

Warum der Begriff „Schulden“ in der Softwareentwicklung ungünstig gewählt ist und worin sich externe und interne Schulden unterscheiden, darüber haben wir im ersten Teil [1] dieser Reihe zu Schulden in der Softwareentwicklung gesprochen. Sehen wir uns nun genauer an, was alles unter die internen Schulden fällt. Würde man Entwickler danach fragen, wäre „technische Schulden“ wohl einer der am häufigsten genannten Begriffe. Diese zählen zweifellos meist zu den größten Problemen, doch gibt es noch eine Menge anderer Arten von Schulden, die Sie in Ihrer Softwareentwicklung im Blick behalten sollten. Im Folgenden schauen wir uns ein paar dieser internen Schuldenarten an, woraus sie entstehen und welche Konsequenzen sie auf lange Sicht haben.

Technische Schulden

Wie schon gesagt, sind die „technischen Schulden“ wohl die bekanntesten in der Softwareentwicklung und werden oft auch als Synonym für Schulden allgemein verwendet. Aber die technischen Schulden bezeichnen nur einen kleinen Bereich der Schulden in der Softwareentwicklung. Mit ihnen werden meist entstandene Probleme beschrieben, welche weitestgehend auf Fehler im Quellcode zurückzuführen sind. Beispielsweise führen die folgenden Fehler zu steigenden technischen Schulden:

- Suboptimale Implementierung (etwa durch Zeitdruck)
- Fehlende Unit-Tests
- Unklare oder inkonsistente Codestruktur
- Übermäßige Abhängigkeiten
- Schlecht lesbarer Quellcode

Solche technischen Schulden entstehen oft bei den Entwicklern – also während der Programmierung. Wichtig ist jedoch, dass die Entwickler nicht immer die Ursache sein müssen. So können technische Schulden natürlich durch einen schlecht ausgebildeten oder nicht sauber arbeitenden Entwickler entstehen, doch ist nach meiner Erfahrung die Ursache meist eine andere wie beispielsweise Zeitdruck, zu wenig Ressourcen für qualitätssteigernde Maßnahmen oder auch fehlende Vorgaben im Bereich des Softwaredesigns.

Architekturschulden

Architekturschulden sind ein häufig übersehener, aber äußerst bedeutsamer Bereich. Sie betreffen die grundlegende Struktur und Gestaltung des Softwaresystems. Diese Schulden entstehen, wenn die Architektur eines Systems nicht sauber, flexibel oder zukunftssicher gestaltet ist. Architektur-

schulden wirken sich langfristig auf die Skalierbarkeit, Wartbarkeit und Weiterentwicklung eines Systems aus. Typische Ursachen für Architekturschulden sind:

- Monolithische Strukturen
- Unzureichende Modularisierung
- Keine oder wenig Entkopplung
- Rolle des Architekten nicht besetzt
- Kein Softwaredesign-Wissen bei den Entwicklern
- Fehlende Architekturvorgaben

Oft ist Zeitdruck im Projekt das Problem, aber auch Unwissenheit: Vielen Entscheidern ist die Relevanz einer korrekten Softwarearchitektur nicht bewusst. So werden dann keine Zeiten für die Erstellung oder Implementierung der Architektur vorgesehen, und oft wird die Rolle des Softwarearchitekten erst gar nicht besetzt. Nach meiner Erfahrung aus vielen Projekten ist Letzteres ein kapitaler Anfängerfehler. Machen Sie ihn nicht: Besetzen Sie die Rolle und bauen Sie teamweites Wissen im Bereich Architektur und Design auf.

Die Auswirkungen von Architekturschulden sind oft schleichend, können aber auf Dauer zu erheblichen Kosten führen. Ein System mit gravierenden Architekturschulden lässt sich nur schwer modernisieren, erweitern und warten, was zu teuren und zeitaufwendigen Refactoring-Maßnahmen führt. Deshalb ist es wichtig, frühzeitig in die Architekturqualität zu investieren und diese regelmäßig zu überprüfen.

Dokumentationsschulden

Dokumentationsschulden sind der nächste oft unterschätzte, aber essenzielle Bereich der Softwareentwicklung. Sie entstehen, wenn die Informationen, die für das Verständnis der Funktionalität, die Wartung oder die Weiterentwicklung eines Systems notwendig sind, nicht vollständig, klar oder aktuell sind. Während sich technische und Architekturschulden eher auf die Struktur und Funktion des Codes konzentrieren, sind Dokumentationsschulden ein Problem der Kommunikation und Wissensweitergabe. Typische Beispiele sind:

- Fehlende Dokumentation
- Veraltete Dokumentation
- Unklare oder unstrukturierte Dokumentation
- Fehlende API- oder Schnittstellenbeschreibungen

Diese Schulden entstehen oft, weil die Dokumentation nicht denselben Stellenwert wie die eigentliche Entwicklung genießt. In der Praxis wird sie häufig als „nice to have“ betrachtet und aus Zeitgründen vernachlässigt.



Die Folgen von Dokumentationsschulden zeigen sich besonders dann, wenn neue Teammitglieder eingearbeitet werden müssen, ein Fehler in einem unbekannten Teil des Systems auftritt oder alte Module aktualisiert werden sollen.

Um sie zu vermeiden, ist es wichtig, Dokumentation als festen Bestandteil des Entwicklungsprozesses zu etablieren. Klare Richtlinien, regelmäßige Reviews und die Nutzung von automatisierten Tools zur

Generierung von Dokumentation können dabei helfen. Eine gut gepflegte Dokumentation ist nicht nur ein Zeichen von Professionalität, sondern auch eine wertvolle Investition in die Nachhaltigkeit und Effizienz der Softwareentwicklung.

Prozessschulden

Prozessschulden betreffen Arbeitsweisen und organisatorischen Abläufe im Entwicklungsprozess. Sie entstehen, wenn ineffiziente, unklare oder unzureichende Prozesse die Zusammenarbeit und Produktivität des Teams beeinträchtigen. Während technische, Dokumentations- und Architekturschulden meist greifbare technische oder konzeptionelle Probleme darstellen, sind Prozessschulden oft weniger greifbar und offensichtlich, aber nicht weniger schädlich. Typische Ursachen und Beispiele für Prozessschulden sind:

- Kein expliziter Prozess
- Ineffiziente Prozessschritte
- Keine kontinuierliche Verbesserung
- Überlastete Rollen
- Nicht besetzte Rollen
- Schlechte Ausbildung der Rollen

Da ein guter Prozess nicht direkt auf den Funktionsumfang und damit das Betriebsergebnis einzufließen wird, wird er oft vernachlässigt. Hinzu kommt, dass Softwareentwicklung auch (irgendwie) ohne expliziten Prozess funktioniert. Im Lauf der Zeit bildet sich immer ein impliziter Prozess, der jedoch oft alles andere als optimal und auch nicht skalierbar ist.

Die Auswirkungen von Prozessschulden zeigen sich in Form von längeren Entwicklungszyklen, unklaren Verantwortlichkeiten, fehlender Qualitätssicherung und frustrierten Teammitgliedern. Oft führen sie auch dazu, dass technische Schulden oder andere Probleme im Projekt unbemerkt bleiben.

Prozessschulden abzubauen ist meist ein langer Weg. Ein passender Ansatz muss gewählt, ein Prozess aufgesetzt, Rollen identifiziert, besetzt und geschult und schließlich auch im Alltag „gelebt“ werden. Auch wenn der Prozess meist eine der teuersten Baustellen ist, lohnt sich kaum eine Investition so sehr wie diese. Sind die Prozessschulden hoch, wirkt sich das in nahezu jedem Bereich stark aus, und eine geregelte Softwareentwicklung ist oft gar nicht mehr möglich.

Wissensschulden

Wissensschulden entstehen, wenn kritisches Wissen über das System, die Prozesse oder spezifische Technologien nicht vorhanden, nicht dokumentiert oder nicht ausreichend im Team

geteilt wird. Sie sind eine oft unterschätzte Form von Schulden, die besonders problematisch werden, wenn zentrale Teammitglieder ausscheiden oder kurzfristig nicht verfügbar sind. Analog zu Dokumentations- und Prozessschulden handelt es sich hier nicht um konkrete Defizite im Code, sondern um Lücken in der Verfügbarkeit und Weitergabe von Wissen. Typische Beispiele für Wissensschulden sind:

- Abhängigkeit von Schlüsselpersonen
- Fehlende Übergabedokumentation
- Unzureichende Teamkommunikation
- Fehlende Schulungen

Diese Schulden treten häufig in Phasen eines Projekts auf, in denen Dokumentation und Wissenstransfer als nachrangig angesehen werden. Ebenso tragen fehlende Standards für Wissensmanagement und eine mangelnde Feedbackkultur dazu bei, dass Wissensschulden weiter anwachsen.

Die Auswirkungen der Wissensschulden sind von allen bisher gezeigten Arten oft die fatalsten: Ein Team, das nicht richtig ausgebildet ist, erzeugt schlechte Prozesse, schlechte Architekturen, schlechte Dokumentation und natürlich schlechten Quellcode. Damit steigen von Projektbeginn an alle bisher gezeigten Schuldenarten kontinuierlich an und sorgen schnell für ein kaum noch zu steuerndes Softwareprojekt.

Abgebaut werden sollten solche Schulden durch regelmäßige und zielgerichtete Weiterbildung sowie die Förderung des Wissensaustauschs untereinander. Auch eine offene Fehler- und Feedbackkultur kann dazu beitragen, Lücken im vorhandenen Wissen oder Fehler beim Aufbau von Wissen schnellstmöglich zu erkennen und zu beseitigen.

Fazit

Die hier gezeigten fünf Arten von Schulden in der Softwareentwicklung sind erst der Anfang. Man könnte diese Liste weiter fortführen: Testschulden, Datenbankschulden, UI/UX-Schulden, organisatorische Schulden, Sicherheitsschulden, DevOps-Schulden, Kommunikationsschulden und viele weitere. Sie sollten sich jetzt bewusst sein, dass es eine ganze Menge Schuldenarten gibt, bei denen Sie vielleicht nicht so gut abschneiden. Das ist nach meiner Erfahrung eine der wichtigsten Erkenntnisse. Denn dann haben Sie ein erstes wichtiges Ziel erreicht, nämlich die Beantwortung der Frage „Wie schlimm ist es denn wirklich?“ – viel Spaß dabei! ■

[1] David Tielke, *Der Schuldensumpf*,
www.dotnetpro.de/A2501DDD



David Tielke

ist freiberuflicher Berater und von Microsoft zertifizierter Trainer für die Anwendungsentwicklung auf der .NET-Plattform. Darüber hinaus hat er sich auf die Bereiche Softwarearchitektur, Softwarequalität und ALM spezialisiert.
mail@david-tielke.de

dnpCode

A2502DDD

WEB-SECURITY

Dreimal hält besser

Wie ist der Stand bei Content Security Policy 3?

Content Security Policy (CSP) ist ein „Defense in depth“-Mechanismus, der per HTTP-Header oder `<meta>`-Tag einschränkt, welche Ressourcen Browser verwenden dürfen. Der Hauptzweck dieses W3C-Standards liegt in der Vermeidung von XSS-Angriffen: Selbst wenn eine Webanwendung eine Cross-Site-Scripting-Lücke besitzt, kann CSP deren Ausnutzung in vielen Fällen verhindern. Wir haben uns Content Security Policy in der dotnetpro bereits ausführlich angesehen [1].

Wirrwarr bei Versionen und Kompetenzen

Diesmal werfen wir einen Blick auf neue Entwicklungen in Content Security Policy 3. Es gibt nämlich mehrere Versionen davon, mit unterschiedlichem Funktionsumfang (verständlich) und Browser-Support (in Teilen unverständlich, dazu gleich mehr). Content Security Policy Version 1.0 stand bereits im Dezember 2012 [2] auf der Schwelle hin zum Standard. Der Status „W3C Candidate Recommendation“ war erreicht, die nächste Stufe wäre „W3C Recommendation“, also ein Standard gewesen. Gut zwei Jahre später, im Februar 2015, gab es das nächste Update [3] – allerdings in eine ganz andere Richtung. CSP 1.0 war jetzt eine „W3C Working Group Note“, was den Rang einer Besprechungsmitschrift hat. Der Grund: Wie schon früher in der Geschichte – man denke etwa an HTML5 – war die Praxis schneller als das W3C und die Arbeit am Nachfolger Content Security Policy 2 bereits so weit fortgeschritten, dass der Fokus auf die neue Version gelegt wurde. Doch es sollte noch bis Dezember 2016 [4] dauern, bis der Status „W3C

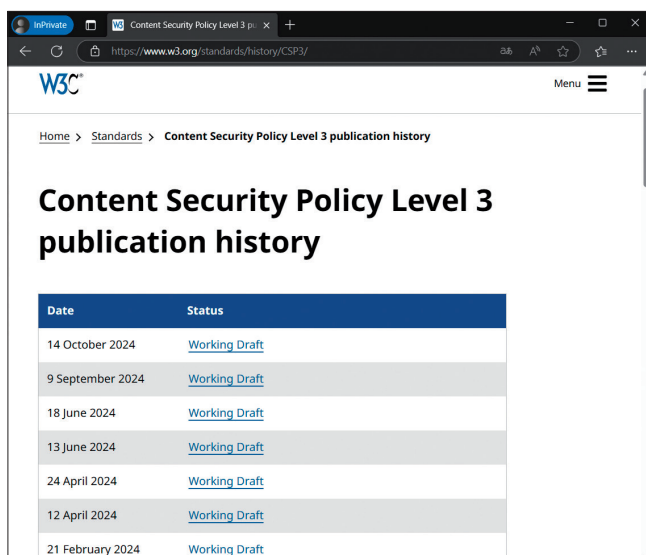
Recommendation“ erreicht war, mit hervorragendem Browser-Support in allen heute relevanten Browsern. Also Ende gut, alles gut? So einfach ist es leider nicht.

Interessant zu betrachten ist, wer denn für die jeweiligen Standarddokumente hauptverantwortlich war. Bei Content Security Policy 1 waren das Brandon Sterne, einst ein Mozilla-Mitarbeiter, sowie Adam Barth von Google. Diese browserübergreifende Kooperation sorgte dafür, dass CSP 1.0, das ja nie zum finalen Standard erkoren wurde, trotzdem von vielen Browsern bereits implementiert wurde, neben Firefox und Chrome unter anderem auch in Safari und später dem Microsoft-eigenen Edge. So drängend waren die Probleme durch XSS-Lücken allerorten!

Content Security Policy 2.0 war ebenfalls ein Gemeinschaftswerk. Mozilla-Mitarbeiter Dan Veditz übernahm den Posten von Sterne (der trotzdem im Standarddokument erwähnt wird), und Barth bekam Unterstützung von seinem Kollegen Mike West. Ergebnis: Wieder einhellige Browser-Unterstützung, alle ziehen an einem Strang.

Dennoch ist das Kapitel Content Security Policy nicht abgeschlossen, denn es gibt noch viele weitere Ideen für Features. Die Arbeiten gingen direkt nach Verabschiedung von CSP 2.0 weiter. Die nächste Version erhielt den Arbeitstitel Content Security Policy 3.0. Zum Redaktionsschluss Mitte November 2024 war deren Status „W3C Working Draft“, zum Standard hin sind also noch einige Stufen zu erklimmen. Letztmals aktualisiert wurde das Dokument Mitte Oktober 2024 [5], es wird also weiterhin daran gearbeitet. Die Liste der Aktualisierungen des kommenden Standards [6] in **Bild 1** zeigt, dass es fast im Monatsrhythmus Änderungen gibt, woraus wir die positive Schlussfolgerung ziehen können: Es geht weiter voran.

Ein Blick auf die Autorenliste zeigt ein interessantes Bild: Mike West ist noch mit dabei, sowie neu sein Google-Kollege Antonio Sartori. Mozilla schickt keine Vertreter mehr. Natürlich steht jetzt die Befürchtung im Raum, dass es ähnlich läuft wie bei vielen der JavaScript-APIs, die wir in den letzten Jahren in der dotnetpro behandelt haben: Google macht den Standard, Chrome (und verwandte Produkte wie Edge und andere) setzen ihn um, die anderen Browser-Teams warten eher ab, wann der Standard auch tatsächlich final ist – und bieten so weniger Features an. Und in der Tat haben insbesondere Firefox und Safari sehr zögerlich auf die Bemühungen rund um CSP 3 reagiert. Wohl weil immer wieder die Überlegung existiert, man könne Content Security Policy ähnlich wie HTML zu einem „Living Standard“ machen, also ein Dokument erstellen, das nie fertig ist, wurden doch sukzessive einige Features aus Content Security Policy 3 in andere Browser integriert. Auf der Standard-Anlaufstelle Ca-



Date	Status
14 October 2024	Working Draft
9 September 2024	Working Draft
18 June 2024	Working Draft
13 June 2024	Working Draft
24 April 2024	Working Draft
12 April 2024	Working Draft
21 February 2024	Working Draft

Workaholics: Die Arbeiten an Content Security Policy 3 gehen voran (**Bild 1**)

Bild: W3C [6]

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android
Content-Security-Policy	✓ 25 ...	✓ 14	✓ 23 ...	✓ 15	✓ 7 ...	✓ 25 ...	✓ 23
base-uri	✓ 40	✓ 79	✓ 35	✓ 27	✓ 10	✓ 40	✓ 35
block-all-mixed-content	✓ 44 *	✓ 79	✗ 48-121	✓ 31 *	✓ 10	✓ 44 *	✗ 48-121
child-src	✓ 40	✓ 15	✓ 45	✓ 27	✓ 10	✓ 40	✓ 45
connect-src	✓ 25	✓ 14	✓ 50 ...	✓ 15	✓ 7	✓ 25	✓ 23

Detailgetreu: Aktuelle Auflistung bei MDN zum CSP-3-Support (Bild 2)

niuse.com ist hiervon aktuell leider nichts zu sehen, aber das Mozilla Developer Network bietet eine gut gepflegte Übersicht [7] (Bild 2). Wichtig ist natürlich immer, das stets selbst zu verifizieren. Eine Content Security Policy kann Auswirkungen auf die Funktionsweise einer Anwendung haben, unabhängig davon, was irgendein Dokument im Internet sagt.

Dynamischer Code, mit Einschränkungen

Mit einer Content Security Policy ist es insbesondere möglich, das Laden von JavaScript-Code zu beschränken – schließlich geht es um die Verhinderung von Cross-Site Scripting. Wenn allerdings JavaScript-Code durch eine Policy freigeschaltet wird, besteht möglicherweise ein Vertrauensverhältnis zwischen Website(-Code) und JavaScript-Code. Was also passiert, wenn der eingebundene JavaScript-Code selbst auch JavaScript-Code nachlädt? Nachfolgend ein typischer Auszug aus Googles reCAPTCHA-JavaScript-Bibliothek [8], et was gekürzt und aus Lesbarkeitsgründen angepasst:

```
var po = document.createElement('script');
po.type='text/javascript';
po.src='https://www.gstatic.com/recaptcha/releases/
-ZG7BC9TxCVEbzIO2m429usb/';
var s=document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(po, s);
```

Der Code erzeugt also ein neues `<script>`-Element und integriert dieses in die aktuelle Seite. Der Code liegt in diesem Beispiel allerdings auf <https://www.gstatic.com>, während die JavaScript-Bibliothek selbst auf <https://www.google.com> gespeichert ist. Das kann beim Einsatz einer Content Security Policy zu Problemen führen, denn es müssen beide Hosts in der Policy freigeschaltet werden, oder noch besser: beide Dateien. Dies kann aber anderen Ärger auslösen: Google kann jederzeit den URL der nachgeladenen JavaScript-Datei än-

dern. Wenn wir tatsächlich ein Vertrauensverhältnis zu Googles JavaScript-Code haben (Thema für einen anderen Artikel), könnten wir das Nachladen von weiterem Code erlauben. Dies erledigt der Wert `'strict-dynamic'`. In unserem Fall hätte dann der Code, der von `gstatic.com` nachgeladen wird, dasselbe Vertrauenslevel wie der von `google.com`.

```
Content-Security-Policy: script-src 'https://www.google
.com/recaptcha/api.js' 'strict-dynamic'
```

Die Spezifikation stellt klar, dass `'strict-dynamic'` nur JavaScript-Code freischaltet, der „non parser inserted“ ist, also nicht durch den Parser läuft, sondern per DOM-API eingefügt wird. Der Einsatz etwa von `document.write('<script...>')` wäre „parser inserted“ und auch durch `'strict-dynamic'` nicht möglich, außer der URL des nachgeladenen JavaScript-Codes wäre in der Policy freigeschaltet.

Inline, aber nur ein bisschen

Eine der größten Herausforderungen für eine Content Security Policy, insbesondere für Legacy-Anwendungen, ist Inline-Code. Natürlich gibt es den Wert `'unsafe-inline'`, mit dem Inline-JavaScript-Code und Inline-CSS-Stile freigeschaltet werden können, aber der Name deutet bereits an, dass das nicht unbedingt eine gute Idee ist. Direkt ausgesprochen: `script-src 'unsafe-inline'` erlaubt die Ausnutzung etwaiger XSS-Lücken. Nun lassen sich Anwendungen mit gewissem Aufwand so umwandeln, dass damit eine gute CSP eingesetzt werden kann, beispielsweise durch den Einsatz eines Einmal-Tokens (Nonce) oder durch die Angabe des/der Hashes von Inline-JavaScript-Code oder -CSS-Stilen.

Was aber in vielen älteren Webanwendungen nur mit großem Aufwand zu beseitigen ist, sind JavaScript-Eventhandler, also im Wesentlichen HTML-Attribute, die mit `on` beginnen: `onload`, `onclick`, `onchange`, `oninput`, und viele mehr. Hier ein Beispiel:

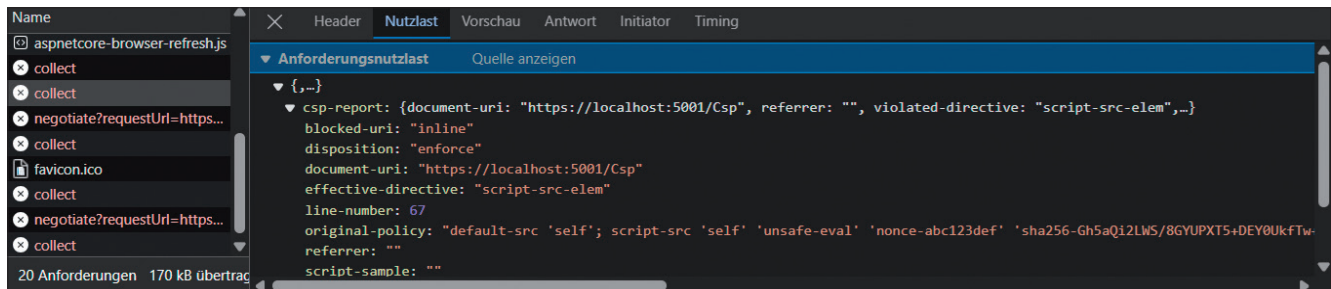
```
<input type="text" id="tb" onchange="doSomething()">
```

Jeder derartige Eventhandler musste bisher durch JavaScript-Code nach diesem Muster ersetzt werden:

```
document.querySelector('#tb').addEventListener(
  'change',
  (e) => { doSomething(); });
```

Anstatt das `onXXX`-Attribut durch derartigen Code zu ersetzen, wäre es ein alternativer Ansatz, auf den Hash des Codes zu setzen. Dies hat bei Content Security Policy 2 nur für Inline-JavaScript-Code funktioniert. Bei Content Security Policy 3 sorgt die Zusatzangabe `'unsafe-hashes'` dafür, dass auch die Werte solcher Eventhandler-HTML-Attribute miterfasst werden. Die Policy kann dann beispielsweise so aussehen:

```
Content-Security-Policy: script-src
  'unsafe-hashes' 'sha256-bnQkgwAfjTxnZS1FxZe1ogJadBHLnR
  uuL54WC+v+tMY='
```



Browserspion: Policy-Verletzungen werden auf Wunsch gemeldet (Bild 3)

Der in der Policy angegebene Hash ist der von `doSomething()`, also dem Wert des `onchange`-Attributs (und nicht das Attribut selbst!). Dieser ist nun freigeschaltet, und zwar in allen Browsern, die `'unsafe-hashes'` unterstützen. Und das sind heutzutage alle; Firefox ist Anfang 2023 mit Version 109 dazu gestoßen.

Doch warum steht da ein `unsafe` in dem Wert? Weil es einen potenziellen Angriffsvektor gibt: Durch die Angabe des Hashs wird der Inline-Code `doSomething()` freigeschaltet. Angenommen, der tut etwas Gefährliches. Findet ein Angreifer eine XSS-Lücke in unserer Anwendung, könnte genau dieser Code eingeschleust und zur Ausführung gebracht werden. Es ist also ein kleines Restrisiko, das bleibt.

Passend dazu: Ergänzend zu den Direktiven `style-src` und `script-src` gibt es jetzt spezifischere Varianten, die zwischen CSS- beziehungsweise JavaScript-Elementen (`style-src-elem` beziehungsweise `script-src-elem`) und -Attributen (`style-src-attr`, `script-src-attr`) unterscheiden.

Mit Hash, aber extern

Die Angabe von Hashes in der Content Security Policy gilt ab CSP 3 übrigens auch für externe JavaScript-Dateien. Es gibt jedoch eine Bedingung: Im `integrity`-Attribut des `<script>`-Tags muss derselbe Hashwert wie in der Policy angegeben sein. Dieses Attribut ist Teil der Spezifikation Subresource Integrity [9], seit 2016 W3C-Standard. Wenn die geladene externe Datei nicht zum Hash passt, wird der Code nicht ausgeführt, was beispielsweise in dem Szenario interessant sein kann, in dem ein CDN nach einem Angriff Schadcode ausliefern würde.

Hier ein Beispiel, wie Content Security Policy und Subresource Integrity Hand in Hand arbeiten. Im Markup einer Seite wird die aktuelle Version von Bootstrap geladen; das `integrity`-Attribut ist entsprechend gesetzt:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU90FeRpok6YctnYmDr5pN1yT2bRjXh0JMhY6hW+ALEwIH">
```

Die Content Security Policy schaltet das CDN nicht frei, aber per Hash die gewünschte CSS-Datei:

```
Content-Security-Policy: style-src 'sha384-QWTKZyjpPEjISv5WaRU90FeRpok6YctnYmDr5pN1yT2bRjXh0JMhY6hW+ALEwIH'
```

Reporting, aber anders

Eine weitere Änderung ergibt sich im Reporting. Zur Erinnerung: Die Direktive `report-uri` gibt einen Endpunkt an, der bei einer Policy-Verletzung per HTTP-POST-Request Informationen über das „Vergehen“ im JSON-Format erhält. Hier ein typisches Beispiel:

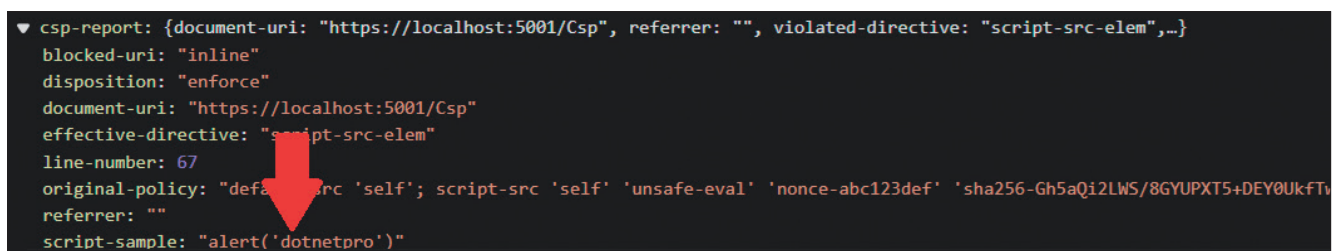
```
Content-Security-Policy: default-src 'self';
report-uri /collect
```

Bild 3 zeigt die Payload eines typischen Reporting-Requests.

Als neues Feature bietet CSP 3 den Ausdruck `'report-sample'`. Dieser wird nicht bei `report-uri` angegeben, sondern bei der jeweiligen einschränkenden Direktive, etwa `default-src`:

```
Content-Security-Policy: default-src 'self'
'report-sample'; report-uri /collect
```

Ergebnis: Die HTTP-Anfrage an den Reporting-Endpunkt enthält jetzt auch einen Auszug aus dem geblockten Markup. Verhindert die Content Security Policy etwa gerade einen XSS-Angriff, so wird der eingeschleuste JavaScript-Code mit übermittelt. **Bild 4** zeigt, wie das aussehen kann. In der Praxis



Code-Review: Der Grund für die Policy-Verletzung wird mitgeliefert (Bild 4)

hilft diese Option insbesondere bei der Fehlersuche während der Entwicklung der Content Security Policy, aber natürlich ist es bei einem Angriff immer interessant zu sehen, wie genau eine Lücke ausgenutzt wird. Alle modernen Browser unterstützen inzwischen dieses Feature.

Einschneidender ist jedoch eine andere Änderung in Content Security Policy 3 hinsichtlich des Reportings: Die Direktive *report-uri* gilt als veraltet (*deprecated*). Das bedeutet, dass sie in einer zukünftigen Version von Content Security Policy (oder, wenn es schlecht läuft, in einer zukünftigen Veröffentlichung von CSP 3) entfernt werden wird. Das Feature selbst wird natürlich nicht getilgt, denn eine andere Direktive steht als Ersatz zur Verfügung: *report-to*. Mit der Umbenennung ist es allerdings nicht getan, damit einher gehen auch eine neue Syntax sowie ein neuer HTTP-Header. Hier die Entsprechung zum vorherigen CSP-Header, diesmal mit Einsatz von *report-to*:

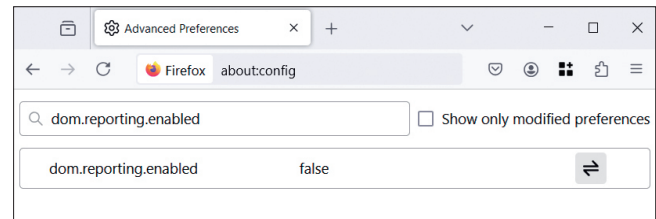
```
Reporting-Endpoints: csp-endpoint="/collect"
Content-Security-Policy: default-src 'self'
'report-sample'; report-to csp-endpoint
```

Der HTTP-Header *Reporting-Endpoints* definiert einen oder mehrere Endpunkte, an die dann ein Report geschickt werden kann. Der hier verwendete Name, *csp-endpoint*, ist beliebig gewählt. In der Content Security Policy selbst wird dann der Endpunkt per Name referenziert, der URL taucht also in der Content Security Policy nicht auf. Grundlage des Ganzen ist ein weiterer Standard, das Reporting API [10]. Oder, genauer gesagt, zukünftiger Standard, denn die aktuelle Fassung ist aus dem August 2024 und – wie CSP 3 – ein W3C Working Draft. Die Idee ist, für verschiedene Reporting-Optionen im Browser eine generische Sammelstelle für die Definition der Endpunkte zu haben. Beteiligt ist neben anderen Mike West, Hauptautor von Content Security Policy 3.

Diese Änderung ist allerdings (noch) kein Grund für großen Aktionismus, und das liegt am Firefox-Browser. Dieser hat sehr lange an *report-uri* festgehalten. Mit Firefox-Version 130 (Anfang September 2024) wurde ein experimenteller Support für das Reporting API eingeführt. Das dazugehörige Bug-Ticket stammt aus dem August 2017 ... [11] Das Feature war aber zum Redaktionsschluss noch standardmäßig deaktiviert. Um es zu verwenden, muss im „geheimen“ Administrationsmenü von Firefox (erreichbar über den URL *about:config*) der Wert *dom.reporting.enabled* auf *true* gesetzt werden (siehe Bild 5). Bis das überall ohne manuellen Eingriff aktiv ist, tut es weiterhin *report-uri*. Wer vollständig auf Nummer sicher gehen will, verwendet einfach *report-uri* und *report-to*. Browser, die beides können, nehmen Letzteres. Systeme ohne Unterstützung für *report-to* verwenden die Angaben von Ersterem.

Neuerungen, aber eher im Detail

Im Wesentlichen waren das die relevantesten Neuerungen in Content Security Policy 3. Darüber hinaus gibt es aber auch noch kleinere Anpassungen in Details, hier eine kurze Auflistung der wichtigsten:



Durch die Hintertür: Reporting API ist im Firefox nur per Feature-Schalter möglich (Bild 5)

- Die Direktive *frame-src*, in Content Security Policy 2 zugunsten von *child-src* entfernt, feiert ihr Comeback und beschränkt die URLs, die in (I)Frames geladen werden können. Neu ist die Direktive *worker-src* für *Worker*, *ServiceWorker* und *SharedWorker*.
- Weitere neue Direktiven sind *manifest-src* (für Manifestdateien) und *webrtc* für bestimmte Realtime-Inhalte.
- HTTP-URLs in der Policy (*http://...*) gelten jetzt auch für HTTPS.

Die Browser-Unterstützung ist da, insofern lohnt es sich, auf Content Security Policy 3 zu setzen, auch wenn das Erreichen des Status „W3C Recommendation“ noch in weiter Ferne zu liegen scheint. Die dotnetpro bleibt am Ball! ■

- [1] Christian Wenz, *Der Browser als Türsteher, dotnetpro 3/2021, Seite 54 ff., www.dotnetpro.de/A2103BrowserAPI*
- [2] *Content Security Policy 1.0 als Candidate Recommendation, www.dotnetpro.de/SL2502ContentSecurity1*
- [3] *Content Security Policy 1.0, herabgestuft als Working Group Note, <https://www.w3.org/TR/CSP1/>*
- [4] *Content Security Policy 2.0 als Standard, <https://www.w3.org/TR/CSP2/>*
- [5] *Content Security Policy 3.0 als Working Draft, <https://www.w3.org/TR/CSP3/>*
- [6] *Arbeitsverlauf bei Content Security Policy 3.0, www.dotnetpro.de/SL2502ContentSecurity2*
- [7] *Browser-Support für Content Security Policy, www.dotnetpro.de/SL2502ContentSecurity3*
- [8] *JavaScript-Bibliothek von reCAPTCHA, <https://www.google.com/recaptcha/api.js>*
- [9] *Subresource Integrity, <https://www.w3.org/TR/SRI/>*
- [10] *Reporting API, <https://www.w3.org/TR/reporting-1/>*
- [11] *Mozilla-Bug zur Einführung von report-to, www.dotnetpro.de/SL2502ContentSecurity4*



Christian Wenz

ist Spezialist für die Architektur von Webanwendungen und für Web Application Security. Seit 2004 ist er Microsoft MVP. Seine neuesten Bücher sind „ASP.NET Core Security“ (erschienen bei Manning) und „C# und .NET 8“ (Hanser). www.christianwenz.de

dnpCode

A2502ContentSecurity

LOKALISIERUNG FÜR ASP.NET CORE

qo' vlvan!

Übersetzungen in ASP.NET Core – ganz einfach.

Webanwendungen, die eine Internationalisierung erhalten haben, besitzen viele Vorzüge. Wer mit ihnen interagiert, bekommt nicht nur Inhalte in der gewünschten Sprache angeboten, sondern freut sich im Idealfall auch über die zugehörigen Währungs- und Datumsformate. In der vorangegangenen Ausgabe der dotnetpro [1] haben wir bereits die Grundlagen gelegt und nach einer Begriffsbestimmung die Herangehensweise von Microsoft an das Thema erarbeitet. Bei .resx-Dateien handelt es sich im Wesentlichen um XML-Dateien mit Name-Wert-Paaren. Für jede Sprache, oder genauer, für jede Culture können wir eine solche Datei anlegen.

Im Rahmen des Artikels verwenden wir aus Gründen der Übersichtlichkeit nur zwei Sprachen. Wenn Sie als dritte Klingonisch einfügen (der ISO-Code lautet *tlh*), könnten Sie möglicherweise auch den Titel dieses Artikels in die Ressourcen-datei einbauen, der bedeutet – angeblich – „Hallo Welt“.

Lokalisierung per Interface

Zuständig für die Ausgabe von lokalisierten Inhalten ist in den meisten Fällen eines von zwei Interfaces: *IStringLocalizer* oder *IHtmlLocalizer*. Beginnen wir zunächst mit dem ersten, das in der vorherigen Ausgabe schon kurz zum Einsatz



Name	Neutraler Wert	de	en
AppointmentDate	Termin: {0}. {1}	Termin: {0}. {1}	Appointment: {1}/{0}

Ressourcensparend:
Übersetzungen in
Visual Studio anlegen
(Bild 1)

Die Namen sind immer dieselben, die Werte dann Locale-spezifisch. Kommt Visual Studio zum Einsatz, gibt es sogar ein halbwegs brauchbares UI für die Verwaltung der Übersetzungen obendrauf.

Im Web-Stack, also ASP.NET Core, basiert Lokalisierung auf einer eigenen Middleware. Diese haben wir in zwei Schritten freigeschaltet, die so essenziell sind, dass sie hier kurz wiederholt werden sollen. Zunächst einmal wird die Middleware konfiguriert. Dabei geben wir den Namen des Verzeichnisses an, in dem die Ressourcendateien (*.resx) liegen:

```
builder.Services.AddLocalization(_ =>
{
    .ResourcesPath = "Resources";
});
```

Im nächsten Schritt fügen wir die Middleware in die HTTP-Request-Pipeline ein. Hierbei müssen wir zusätzliche Optionen angeben: Welche Cultures sollen unterstützt werden, und welche davon ist der Standard? Das kann beispielsweise wie folgt aussehen:

```
app.UseRequestLocalization(_ =>
{
    var cultures = new[] { "de", "en" };
    .AddSupportedCultures(cultures);
    .AddSupportedUICultures(cultures);
    .SetDefaultCulture("de");
});
```

kam, allerdings ohne ausführlichere Erläuterungen. Die Interface-Definition ist recht übersichtlich:

```
public interface IStringLocalizer
{
    LocalizedString this[string name] { get; }

    LocalizedString this[string name, params object[]
arguments] { get; }

    IEnumerable<LocalizedString> GetAllStrings(
        bool includeParentCultures);
}
```

Ein Ausdruck, der dieses Interface implementiert, kann also ähnlich wie ein Array behandelt werden – über einen Namen erhalten wir Zugriffe auf eine *LocalizedString*-Instanz, der Name ist Programm. Die einzige Methode, *GetAllStrings()*, liefert alle Übersetzungen auf einmal zurück, auf Wunsch sogar die von übergeordneten Cultures (also etwa auch alle Texte mit der Culture *de*, wenn wir gerade die spezifischere Culture *de-AT* einsetzen).

Der Einsatz dieses Interfaces geschieht ohne die Notwendigkeit weiterer Konfigurationen. Wir benötigen eine Razor Page (mit Modellklasse, sonst ist es etwas mühsamer) oder eine Razor-View im Fall eines MVC-Projekts. Dort injizieren wir das Interface *IStringLocalizer*; als generischer Typ kommt die zugehörige Modellklasse zum Einsatz. Der injizierte Variablenname ist in der Praxis meist entweder *Localizer* (weil

sprechend) oder *L* (weil kurz). Wir setzen der Lesbarkeit wegen in diesem Artikel auf Ersteres.

```
@inject IStringLocalizer<CulturesModel> Localizer
```

Schlüssel für den Indexer ist dann der in den *.resx*-Dateien verwendete Schlüssel. Im Beispiel aus der letzten Eingabe lautete dieser *CurrentTimeCaption*, dementsprechend kann eine Ausgabe wie folgt aussehen:

```
@Localizer["CurrentTimeCaption"]:
    @DateTime.Now.ToLongDateString()
    @DateTime.Now.ToLongTimeString()
```

Die Übersetzung für *CurrentTimeCaption* wird aus der zugehörigen Ressourcendatei verwendet. Die darauffolgende Datumsausgabe verwendet das Format, das von der aktuellen UI-Culture (*CurrentUICulture*-Eigenschaft) vorgegeben wird. Das war der einfache Fall, aber es gibt – wie oben in der Interface-Definition zu sehen – noch eine zweite Option: die Angabe von Parametern. Es ist nämlich möglich, in dem Übersetzungstext auch Platzhalter zu verwenden, die dann befüllt werden können. Hier ein etwas konstruiertes, aber dafür gut verständliches Beispiel. Ziel ist es, einen Text auszugeben, der in etwa so aussieht:

- **Deutsch:** Termin: 8.3.
- **Englisch:** Appointment: 3/8

Wir haben also sowohl einen lokalisierten Text als auch ein angepasstes Datumsformat – jeweils der 8. März. (Da wir auf das Jahr verzichten, bekommen wir das Format nicht ganz so trivial über .NET direkt.) Mit einem parametrisierten Ressourcenwert können wir die Aufgabe bewältigen. Dazu legen wir einen neuen Schlüssel an, etwa *AppointmentDate*, und dann folgende Definition in der deutschen *.resx*-Datei:

```
<data name="AppointmentDate" xml:space="preserve">
    <value>Termin: {0}.{1}</value>
</data>
```

In der englischen Version sieht es wiederum so aus:

```
<data name="AppointmentDate" xml:space="preserve">
    <value>Appointment: {1}/{0}</value>
</data>
```

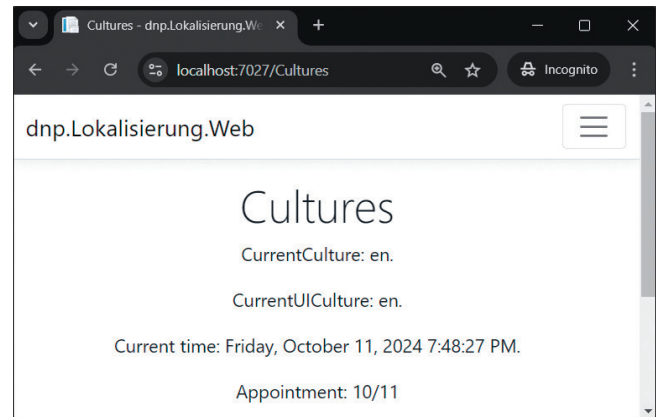
Etwas bequemer ist es natürlich im UI von Visual Studio erledigt (**Bild 1**).

Beachten Sie die Platzhalter: Sie ähneln denen, die von *String.Format()* erwartet werden. Als ersten Parameter müssen wir also den Tag übergeben, als zweiten den Monat. Dies sieht dann folgendermaßen aus:

```
@{
    var now = DateTime.Now;
}
@Localizer["AppointmentDate", now.Day, now.Month]
```

Bild 2 zeigt das Ergebnis, sofern eine englische Culture verwendet wird. In der Praxis findet sich vielerorts das Muster, im Bezeichner für die Übersetzung ebenfalls die Parameter zu erwähnen, etwa *AppointmentDate {0} {1}*. Das stellt dann explizit dar, dass die Übersetzung mit Platzhaltern arbeitet (und wie viele es sind). Zwingend notwendig ist es aber wie gezeigt nicht.

Neben *IStringLocalizer* gibt es mit *IHtmlLocalizer* noch eine zweite Schnittstelle, um lokalisierte Texte auszugeben. Der Unterschied zeigt sich, wenn wir eine Ressource mit



Formatwirrwarr: 10/11 repräsentiert tatsächlich den 11. Oktober (**Bild 2**)

HTML-Markup anlegen. Die Ausgabe mit dem Datum wird mit einem **-Element umgeben, in dem das *class*-Attribut (zur Angabe einer CSS-Klasse) mit einem Platzhalter gefüllt wird:

```
<span class="{2}">Termin: {0}.{1}</span>
```

So sieht es natürlich nur im UI von Visual Studio aus; in den *.resx*-Dateien wird die Ausgabe HTML-codiert:

```
<data name="HtmlAppointmentDate" xml:space="preserve">
    <value>&lt;span class="{2}"&gt;Termin:
        {0}.{1}&lt;/span&gt;</value>
</data>
```

Zur Erinnerung: *Localizer["..."]* liefert einen Wert vom Typ *LocalizedString*. Wird dieser per *@* im UI ausgegeben – egal ob in einer Razor Page oder innerhalb einer MVC-Anwendung –, so erfolgt erneut eine HTML-Codierung. Dieses Standardverhalten von ASP.NET Core dient vor allem der Erschwerung von Cross-Site Scripting, also dem Einschleusen von JavaScript. Bei Verwendung von *IHtmlLocalizer* ist der Rückgabewert hingegen vom Typ *LocalizedHtmlString*. Hier erfolgt keine Ausgabecodierung!

Um *IHtmlLocalizer* verwenden zu können, benötigen wir zunächst eine weitere Middleware. Nach dem Aufruf von *AddControllersWithViews()* oder *AddRazorPages()* oder Ähnlichem müssen wir *AddViewLocalization()* einsetzen: ►

```
builder.Services.AddRazorPages().AddViewLocalization();
```

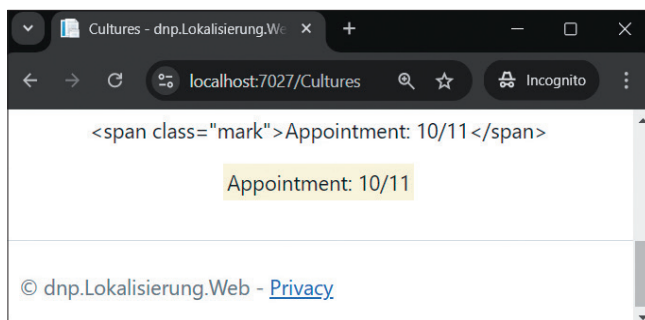
Dann ist es möglich, das Interface per Dependency Injection in der Razor Page oder View verfügbar zu machen:

```
@inject IHtmlLocalizer<CulturesModel> HtmlLocalizer
```

Die Syntax ist dann exakt die gleiche wie beim *IStringLocalizer*:

```
<p>@HtmlLocalizer["HtmlAppointmentDate", now.Day,
    now.Month, "mark"]</p>
```

Bild 3 zeigt das Ergebnis: Zuerst die Ausgabe, wenn *IStringLocalizer* zum Einsatz kam, und dann das (gewünschte) Re-



Moderner Eskapismus: Oben mit HTML-Escaping (mit *IStringLocalizer*), unten ohne (mit *IHtmlLocalizer*) (**Bild 3**)

sultat bei Verwendung von *IHtmlLocalizer*. Sie müssen natürlich darauf achten, dass die Werte für die Platzhalter entweder vertrauenswürdig sind oder vorher – beispielsweise mit *HttpUtility.HtmlEncode()* – ungefährlich gemacht worden sind.

Lokalisierung von Views

Beim Namen der Methode *AddViewLocalization()* kreisen unsere Gedanken sofort um das Thema MVC. Steht dafür nicht das „V“? Nun ist bei ASP.NET Core die Überlappung von Razor Pages und MVC sehr groß, verwenden doch beispielsweise die Views bei MVC die Razor-Syntax. Der Hauptunterschied zwischen beiden Frameworks ist mehr oder minder das Routing.

Allerdings gibt es noch eine Möglichkeit der Lokalisierung, die sich primär an MVC-Anwendungen richtet: das Interface *IViewLocalizer*. Sehen wir es uns erst einmal an und betrachten wir dann im Nachgang die Hintergründe.

Als Erstes müssen wir die Anwendung, die bisher auf Razor Pages basiert, fit für MVC machen, indem wir die entsprechenden Anweisungen in die Datei *Program.cs* einfügen. Wir benötigen unter anderem einen Aufruf von *builder.Services.AddMvc()*, an den wir die Methode *AddViewLocalization()* anhängen (wir brauchen sie also nicht mehr bei *AddRazorPages()*). Hier die relevanten Codestellen zum Aktivieren von sowohl Razor Pages als auch MVC:

```
builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();
builder.Services.AddMvc().AddViewLocalization();
```

```
// ...
```

```
app.MapRazorPages();
app.MapDefaultControllerRoute();
```

Als Nächstes legen wir die Datei *Controllers/LocalizationController.cs* mit minimalem Inhalt an:

```
using Microsoft.AspNetCore.Mvc;

namespace dnp.Lokalisierung.Web.Controllers;

public class LocalizationController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

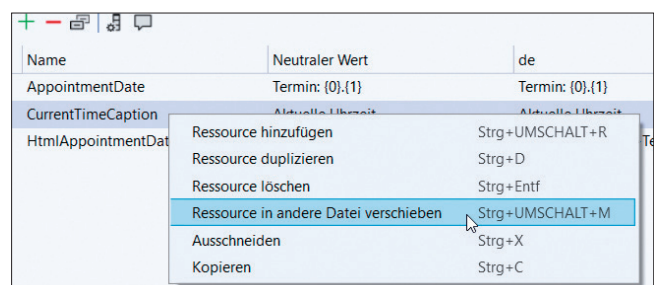
Die zugehörige View, *Views/Localization/Index.cshtml* holt sich per Dependency Injection eine Instanz von *IViewLocalizer*; der Einsatz funktioniert analog zum *IStringLocalizer*:

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
```

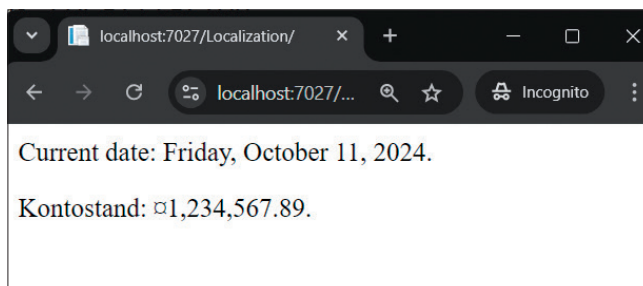
```
<p>
    @Localizer["CurrentDateCaption"]:
    @DateTime.Now.ToLongDateString()
    @DateTime.Now.ToLongTimeString().
</p>
```

Fehlen nur noch die zugehörigen Ressourcendateien. Das Namensschema orientiert sich auch hier an Namen und Ort der View, analog zu Razor Pages. Beispielsweise kann die Ressourcendatei für die Locale *de* einen der beiden folgenden Namen haben:

- *Views/Localization/Index.de.resx*
- *Views.Localization.Index.de.resx*



Umzugsservice: Übersetzungen in Visual Studio in eine andere Ressourcendatei verschieben (**Bild 4**)



You are welcome: Englisch es Datum, englisches Zahlenformat (Bild 5)

Beim Einsatz von Visual Studio ist es besonders bequem. Legen Sie eine neue Ressourcendatei an – gerne ohne Locale –, und fügen Sie dann die Übersetzungen direkt im UI ein. Im Beispiel verwenden wir einen neuen Schlüssel namens *CurrentDateCaption* mit den Werten *Aktuelles Datum* und *Current date*.

Wenn Sie übrigens, zum Beispiel im Rahmen eines Refactorings, eine Übersetzung von der einen in die andere Datei übertragen möchten, geht das komfortabel über das Kontextmenü (siehe Bild 4).

Was ist nun der Hauptunterschied zwischen *IStringLocalizer* und *IViewLocalizer*? Syntactisch gibt es keinen, es liegt eher am Anwendungsfall. *IViewLocalizer* ist jeweils an eine spezifische Razor-View gebunden. Bei *IStringLocalizer* handelt es sich um einen etwas allgemeineren Ansatz, bei dem Ressourcen zum Einsatz kommen können, die mehrfach innerhalb einer Anwendung Verwendung finden und nicht nur in einer einzigen View.

Lokalisierungen teilen

Um solche geteilten Ressourcen zu erstellen, benötigen wir zunächst einmal wieder entsprechende *.resx*-Dateien. In der Praxis hat sich der Dateiname *SharedResource.resx* oder *SharedResources.resx* (sowie die Locale-spezifischen Varianten) eingebürgert, aber es spricht natürlich nichts gegen andere Bezeichnungen – beispielsweise in großen Webanwendungen, bei denen es mehrere themenspezifische geteilte Ressourcen gibt.

Nächster, nicht intuitiver, aber notwendiger Schritt: Legen Sie im Haupt-Namespaces der Anwendung eine leere öffentliche Klasse an, die wie die Ressourcendatei heißt (in unserem Beispiel: *SharedResources*):

```
namespace dnp.Lokalisierung.Web;

public class SharedResources
{
}
```

Dann nämlich „kennt“ ASP.NET Core den Typ der Ressourcen, und Sie können damit einen *IStringLocalizer* oder *IHtmlLocalizer* oder *IViewLocalizer* typisieren und auf diese Weise die geteilten Ressourcen überall in der Anwendung einsetzen. Hier ein Beispiel:

```
@using Microsoft.Extensions.Localization
@using dnp.Lokalisierung.Web
@inject IStringLocalizer<SharedResources>
    SharedLocalizer
<p>
    @SharedLocalizer["AccountBalanceCaption"]:
    @(1_234_567.89.ToString("C")).
</p>
```

In Bild 5 sehen Sie das Ergebnis, zusammen mit der Ausgabe des *IViewLocalizer* aus dem vorherigen Beispiel.

Abschließend noch ein Wort der Warnung: Der Ressourcen-Explorer von Visual Studio hat sporadisch, aber nicht immer, Schwierigkeiten mit geteilten Ressourcen. Beispielsweise werden (vorhandene) Lokalisierungen als fehlend angezeigt (Bild 6), oder es kommt sogar zu einem Absturz der IDE. Speichern Sie also rechtzeitig ...

Annotations lokalisieren

Doch zurück zu MVC. Hier wird gern mit Data Annotations gearbeitet, und diese können auf im Wesentlichen dieselbe Art und Weise lokalisiert werden wie Texte in den Views selbst. Der Übersichtlichkeit halber befindet sich die folgende Implementierung in den Downloads zu diesem Artikel in einem separaten Projekt. Zunächst brauchen wir entsprechende Annotationen in den Modellklassen. Hier ein einfaches Beispiel:

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace dnp.Lokalisierung.MVC.Models;

public class PersonViewModel
{
    [Required]
    [DisplayName("FirstNameCaption")]
    [MaxLength(20, ErrorMessage =
        "FirstNameTooLongError")]
    public string FirstName { get; set; } =
        string.Empty;
    [DisplayName("LastNameCaption")]
    public string LastName { get; set; } =
        string.Empty;
}
```



Name	Neutraler Wert	de	en
AccountBalanceCaption	Kontostand		

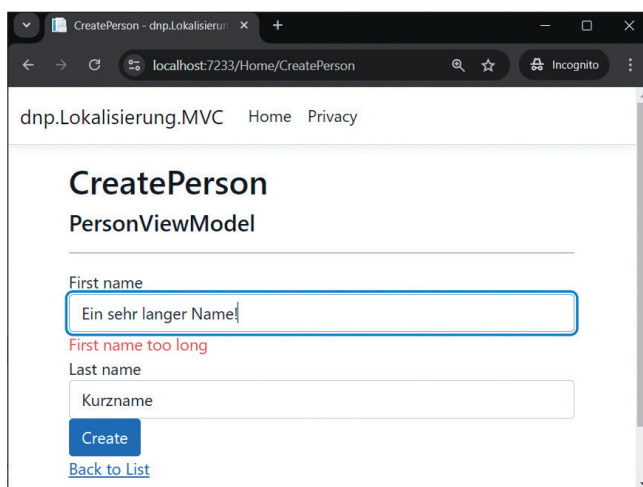
Fake News: Das UI von Visual Studio tut sich (manchmal) mit geteilten Ressourcen schwer (Bild 6)

Name	Neutraler Wert	de	en
FirstNameCaption	Vorname	Vorname	First name
FirstNameTooLongError	Vorname zu lang	Vorname zu lang	First name too long
LastNameCaption	Nachname	Nachname	Last name

Internationales Annotieren: Die Übersetzungen für die Data Annotations (Bild 7)

Die bei *DisplayName* und *ErrorMessage* angegebenen Werte sind dann natürlich die Schlüssel in den entsprechenden Ressourcendateien. Achtung, die Data-Annotations-Attribute, die *ErrorMessage* unterstützen, besitzen auch die Parameter *ErrorMessageResourceName* und *ErrorMessageResourceType*. Allzu verführerisch wäre es, diese nach dem folgenden Muster einzusetzen:

```
[MaxLength(20, ErrorMessageResourceName =
    "FirstNameTooLongError", ErrorMessageResourceType =
    typeof(SharedResources))]
```



Mehrsprachig geschimpft werden: Beschriftungen und Fehlermeldungen sind lokalisiert (Bild 8)

Das aber wird nicht funktionieren; Sie erhalten die Meldung, dass *SharedResources* keine statische Eigenschaft namens *FirstNameTooLongError* besitzt, was sogar korrekt ist.

Damit die Data Annotations auch tatsächlich auf die Ressourcen zugreifen können, ist eine weitere Konfiguration in der *Program.cs* vonnöten, nämlich der Aufruf von *AddDataAnnotationsLocalization()*:

```
builder.Services.AddControllersWithViews()
    .AddViewLocalization()
    .AddDataAnnotationsLocalization();
```

Der Name der zugehörigen Ressourcendatei muss sich am Namespace (ohne den Applikations-Namensraum) der jeweiligen Modelldatei orientieren. Im vorliegenden Beispiel gibt es folgende Möglichkeiten:

- *Models.PersonViewModel.resx*
- *Models/PersonViewModel.resx*

Bei Verwendung von geteilten Ressourcen ist ein Zusatzschritt erforderlich, nämlich eine Extra-Konfiguration innerhalb von *AddDataAnnotationsLocalization()*:

```
// ...
.AddDataAnnotationsLocalization(options =>
{
    options.DataAnnotationLocalizerProvider =
        (type, factory) =>
            factory.Create(typeof(SharedResources));
})
```

Falls Sie dies unterlassen, erscheint anstelle des lokalisierten Werts aus dem jeweiligen Attribut der reine Text (also etwa *FirstNameTooLongError*) in der Ausgabe.

Mit eingerichteten Data Annotations samt lokalisierten Ressourcen (siehe Bild 7) ist die Anwendung nun tatsächlich auch in den Fehlermeldungen übersetzt, selbst die clientseitigen. Bild 8 zeigt das UI; beachten Sie die übersetzten Beschriftungen der Felder sowie die spezifische Fehlermeldung, die ebenfalls in den Ressourcendateien hinterlegt ist.

Damit haben wir die wichtigsten Einsatzzwecke für Lokalisierung von ASP.NET-Core-Anwendungen implementiert. Darüber hinaus gibt es aber noch eine Vielzahl von Spezialthemen. Beispielsweise lässt sich in zahlreiche Aspekte der Lokalisierung – etwa die Bestimmung der Culture, oder die Übersetzung per Localizer, oder die Validierung – per Code eingreifen. Und außerdem gibt es da ja noch Blazor als Webframework, in dem ebenfalls Lokalisierungsoptionen vorhanden sind, nur eben ein paar weniger. All dies behandeln wir im dritten und letzten Artikel dieser Miniserie in der nächsten Ausgabe der dotnetpro. ■

[1] Christian Wenz, *Salamu, Dunia!*, dotnetpro 1/2025, Seite 6 ff., www.dotnetpro.de/A2501WebApps



Christian Wenz

ist Spezialist für die Architektur von Webanwendungen und für Web Application Security. Seit 2004 ist er Microsoft MVP. Seine neuesten Bücher sind „ASP.NET Core Security“ (erschienen bei Manning) und „C# und .NET 8“ (Hanser). www.christianwenz.de

dnpCode

A2502WebApps



HANDS-ON-WORKSHOPS UND
WEITERBILDUNG FÜR SOFTWARE-
ENTWICKLER UND -ARCHITEKTEN



Hands-on Workshops Webentwicklung

Webgrundlagen: Programmierung
mit HTML, CSS und JavaScript (2 Tage)

Next Level TypeScript (2 Tage)

Clean-Code-Entwicklung für JavaScript-
und TypeScript-Entwickler (3 Tage)

Einstieg in Angular (3 Tage)

Angular für .NET-Entwickler (3 Tage)

Angular im Enterprise-Projekt meistern:
Redux & NGRX (2 Tage)

Moderne React Patterns (3 Tage)

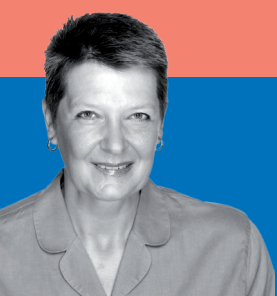
Einführung in Vue.js (2 Tage)

Vue.js 3 - What's new? (2 Tage)

Progressive Web App Bootcamp (3 Tage)

Einführung in GraphQL (2 Tage)

REMOTE
ODER
INHOUSE



Weitere Informationen auf developer-academy.de
Preise auf Anfrage!

Ihre Ansprechpartnerin: Susanne Herl • +49 731 88005-8835 • susanne.herl@ebnermedia.de

HTTP-CLIENT-ENTWICKLUNG

Fit mit Refit

Eine automatische, typsichere REST-Bibliothek für .NET.



Vielleicht empfinde nur ich das so, aber das Einbinden von REST-Services scheint im .NET-Bereich fast eine Wissenschaft zu sein. Aus der Sicht des Clients wirkt die Anbindung ziemlich komplex. Bei ASP.NET Core gestaltet sich der Weg hingegen recht unkompliziert: Man kann Controller-basierte APIs verwenden, Minimal-APIs einsetzen oder komplett ei-

diese Routen relativ statisch, sofern man eine Abwärtskompatibilität bereitstellen will.

Auf dieser Basis gilt es nun, Clients zu entwickeln. Will man dies mit JavaScript oder TypeScript tun, ist der Weg mittels OpenAPI-Spezifikation relativ komfortabel. Nutzt man jedoch .NET, ist der Aufbau eines Clients eher steinig.

Das NuGet-Paket Refit (Bild 1)

gene Middlewares schreiben. Doch wie entwickelt man nun den Client beziehungsweise die Client-APIs? Schreibt man sie alle manuell? Generiert man sie mit OpenAPI und Swagger/NSwag? Kürzlich entdeckte ich einen weiteren Ansatz, der es verdient, hier erwähnt zu werden: Refit.

Refit

Das Grundsatzproblem ist eigentlich schnell erklärt: Man hat verschiedene Endpunkte auf dem eigenen oder einem fremden API, die einerseits aus Routen bestehen und andererseits aus HTTP-Verben wie *GET*, *POST*, *PUT*, *DELETE* und vielleicht noch *PATCH*. Einmal entwickelt und ausgerollt sind

Mit Refit versucht ein mittlerweile recht populäres NuGet-Paket (knapp 100 Millionen Downloads) einen etwas anderen Ansatz (Bild 1).

Die Grundlagen

Der Grundgedanke von Refit ist, dass ein Kontrakt in Form eines Interfaces das API auf der Client-Seite beschreibt. Das bedeutet: Refit ist zunächst eine reine Client-Bibliothek – dennoch lassen sich damit theoretisch sogar APIs implementieren, doch dies ist eine andere Geschichte.

Das Interface gibt dabei einen oder mehrere Endpunkte über eine Methodendefinition an. Mittels Attributen werden dem Interface Meta-Informationen mitgegeben: einerseits die Route und andererseits auch das HTTP-Verb, welches für die Kommunikation mit dem entfernten REST-Service erforderlich ist.

Refit nutzt dafür einen sehr effizienten und modernen Ansatz. Gearbeitet wird nur mit dem Interface, das „irgendwie“ mit dem entfernten Service kommuniziert. Das „irgendwie“ findet unter der Haube statt, wo am Ende ein HTTP-Client und ein JSON-Serializer ihre Arbeit tun.

Refit nutzt den C#-Source-Generator, um bereits zur Kompilierungszeit einen fertigen Client zu generieren. Früher musste man dynamisch IL-Code zur Laufzeit erzeugen. Aber

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net9.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Refit" Version="8.0.0" />
</ItemGroup>
```

Installation von Refit (Bild 2)


```

1 {}
2
3 interface ISunriseSunsetApi
4 {
5     Task<SunriseSunsetModel> GetSunriseSunset(double latitude, double longitude);
6 }
7
8 public class Results
9 {
10     public string sunrise { get; set; }
11     public string sunset { get; set; }
12 }
13
14 public class SunriseSunsetModel
15 {
16     public Results results { get; set; }
17 }

```

Die Schnittstelle wird definiert (Bild 3)

Refit verschiebt diesen Prozess bis zum Kompilieren des Codes. Der große Vorteil: Die Performance leidet praktisch nicht darunter, und man erhält einen fertigen Client.

Ein Beispiel

So viel zur Theorie, doch wie sieht die Praxis aus, und was muss man tun, um mit Refit ein beliebiges API anzusprechen? Für das Beispiel wird der Einfachheit halber ein öffentliches API von Sunrise-sunset.org genutzt, das für die ihm übergebenen Längen- und Breitengrade die Uhrzeit für Sonnenauf- und -untergang liefert. Ein Beispielaufruf sieht so aus:

```

https://api.sunrise-sunset.org/
json?lat=47.449081532022255&
lng=12.088978455606618

```

Bei diesem Aufruf steht *lat* für Latitude (Breitengrad) und *lng* für Longitude (Längengrad). Für dieses sehr einfache Beispiel wäre es nicht

schwer, eine Anbindung mit einem HTTP-Client zu entwickeln; hier geht es aber darum, wie man das mit Refit tut.

Der erste Schritt ist die Installation des NuGet-Pakets (Bild 2). Danach muss nur noch die Schnittstelle definiert werden. Das Interface legt fest, welche Methoden es gibt. Die Namen können beliebig gewählt werden. Gebraucht wird zudem eine Ergebnisklasse, die man sich von Visual Studio generieren lassen kann oder einfach selbst codiert. Damit ergibt sich ein Aufbau wie in Bild 3.

Weil Refit als NuGet-Paket in die Anwendung integriert wurde, kann man damit das API spezifizieren (Bild 4).

Relevant ist hierbei, dass die Methode mit einem *Get*-Attribut versehen wird, dem der Pfad relativ zum Basispfad mit-

gegeben wird (hier: *Get(path: "json")*). Außerdem muss die Methode „wissen“, wie die Parameter gebunden werden sollen. Dafür erlaubt Refit, mit *AliasAs* für den Client einen anderen Namen zu wählen.

Sollten sich die Namen nicht unterscheiden, kann man sich dies natürlich sparen, jedoch ist es so gerade in der .NET-Welt etwas lesbarer.

Verwendung

Das Sunrise-Sunset-API einzusetzen ist dank dieser Definition nun denkbar einfach. Über *RestService.For<T>* ruft ►

```

interface ISunriseSunsetApi
{
    [Get(path: "/json")]
    Task<SunriseSunsetModel> GetSunriseSunset([AliasAs(name: "lat")]double latitude,
        [AliasAs(name: "lng")] double longitude);
}

```

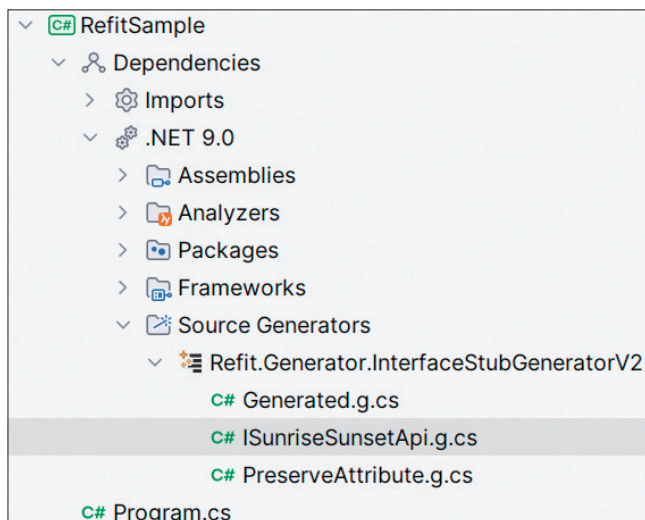
Routen einfügen
(Bild 4)

```

1 using Refit;
2
3 var api = Refit.RestService.For<ISunriseSunsetApi>(hostUrl: "https://api.sunrise-sunset.org");
4 var result :SunriseSunsetModel = await api.GetSunriseSunset(latitude: 47.449081532022255,
5     longitude: 12.088978455606618); // Task<SunriseSunsetModel>
6 Console.WriteLine(result.results.sunrise);
7 Console.WriteLine(result.results.sunset);
8
9 interface ISunriseSunsetApi
10 {
11     [Get(path: "/json")]
12     Task<SunriseSunsetModel> GetSunriseSunset([AliasAs(name: "lat")]double latitude,
13         [AliasAs(name: "lng")] double longitude);
14 }

```

Verwendung des
Clients (Bild 5)



Der generierte Code für den C#-Client (Bild 6)

man eine generische Methode auf, der das Interface mitgegeben wird (Bild 5). Darüber hinaus gibt es noch zwei weitere Wege, das API anzusprechen: Entweder wird dem Aufruf der

ansetzen und dabei herausfinden, was Refit unter der Haube tut (Bild 6). Wer genau hinsieht, erkennt, dass eine Implementierung unseres Interfaces erfolgt, diese mithilfe eines *IRequestBuilders* den Request zusammenbaut und diesen dann per HTTP-Client an die Zieladresse schickt.

Je tiefer man in die Code-Analyse einsteigt, desto deutlicher wird, dass immer noch Reflection im Einsatz ist, um alle Informationen zusammenzutragen. Es gibt jedoch Bestrebungen [3], auch diese Informationen mit einem Source Generator aufzubauen (Bild 7). Insbesondere für Native AOT wäre das sehr verlockend.

Weitere Funktionen

Refit ist kompatibel zu .NET Standard 2.0 und somit für alle .NET-Varianten (mehr oder weniger) verfügbar. Refit kann mehr, als hier vorgeführt wurde. Beispielsweise lässt sich Refit zur Polly-Integration nutzen (für Retry-Versuche von Anfragen) oder auch zur Authentifizierung. Weitere Informationen finden Sie in der Dokumentation auf GitHub [4].

Ebenfalls sehr dynamisch ist die Art und Weise, wie sich das Interface definieren lässt. Wie in Bild 8 ersichtlich, kann man dabei auch mit komplexen Typen arbeiten.

```
public async global::System.Threading.Tasks.Task<global::SunriseSunsetModel> GetSunriseSunset(double @latitude, double @longitude)
{
    var _____arguments = new object[] { @latitude, @longitude };
    var _____func = requestBuilder.BuildRestResultFuncForMethod("GetSunriseSunset", _____typeParameters );

    return await ((global::System.Threading.Tasks.Task<global::SunriseSunsetModel>)_____func(this.Client, _____arguments)).ConfigureAwait(
}

private static readonly global::System.Type[] _____typeParameters0 = new global::System.Type[] { typeof(double), typeof(double) };
```

Source-Generatoren (Bild 7)

```
[Get("/group/{request.groupId}/users/{request.userId}")]
Task<List<User>> GroupList(UserGroupRequest request);

class UserGroupRequest{
    int groupId { get;set; }
    int userId { get;set; }
}
```

Weitere Möglichkeiten der Routen-Definition (Bild 8)

URL des Service mitgegeben oder ein fertig konfigurierter HTTP-Client.

Wer möchte, kann über NuGet sogar eine HttpClient-Factory und den Dependency Injection Container von Microsoft nachrüsten. Für unser einfaches Beispiel bleibt es bei einem fixen URL als Basisadresse. Anschließend kann man das Interface verwenden und auch die dafür definierten Methoden nutzen.

Generierter Client

Wie bereits angesprochen generiert Refit beim Kompilieren einen C#-Client. Man kann sich diesen auch ohne Weiteres

Fazit

Wieder ein Weg mehr, um zu einem REST-Client zu kommen, aber – wie ich meine – ein äußerst angenehmer Ansatz, sofern man vorrangig in der .NET-Welt unterwegs ist, um Clients zu bauen. Vielleicht ist Refit auch für Sie eine willkommene Idee und das nächste Projekt. ■

[1] Refit auf NuGet, www.dotnetpro.de/SL2502NETirol1

[2] Sunrise-Sunset-API, <https://sunrise-sunset.org/api>

[3] Refit-Issue 1746, www.dotnetpro.de/SL2502NETirol2

[4] Dokumentation, www.dotnetpro.de/SL2502NETirol3



Christian Giesswein

studierte Wirtschaftsinformatik in Wien und entwickelt von klein auf Software mit .NET und C#. In Tirol hat er das Unternehmen Giesswein Software-Solutions gegründet, das sich auf Individualsoftware und Consulting spezialisiert.

christian@software.tirol

dnpCode

A2502NETirol

RISIKO VIRTUELLER FUNKTIONEN

Vorsicht an der Bahnsteigkante

Unerwartete Probleme mit virtuellen Funktionen im Konstruktor.

Virtuelle Funktionen sind ein fester Baustein der Polymorphie [1]. Es handelt sich hierbei um Methoden einer Klasse, deren Einsprungsadresse erst zur Laufzeit ermittelt wird. Die Bindung erfolgt somit dynamisch und ermöglicht es, Klassen von einer Basisklasse abzuleiten und im Zuge dessen Funktionen der Basisklasse zu überschreiben, um diese an spezifische Bedürfnisse anzupassen. Durch die Ableitung der Klassen ergibt sich die Polymorphie.

Ein konkretes Beispiel für Polymorphie ist eine Anwendung, welche Kontakte aus verschiedenen Systemen (M365, AD, CSV-Datei) einlesen und in ein einheitliches Format wandeln soll. Dafür könnten für die verschiedenen Protokolle spezifische Enumeratoren implementiert werden. Die Ermittlung der eindeutigen Kennung müsste individuelle Eingriffe ermöglichen, da diese gegebenenfalls aus mehreren Attributen des Kontakts berechnet werden muss (zum Beispiel Vorname, Separator, Nachname, Separator, Abteilung).

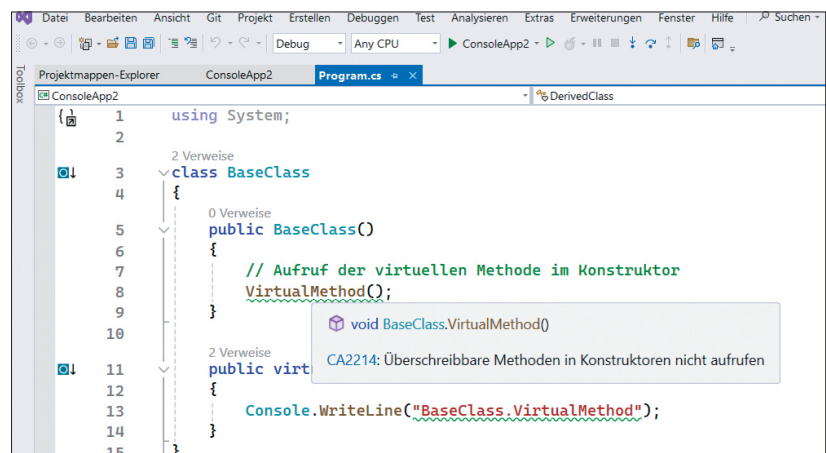
Im Klassen-Model würde die zentrale Funktionalität innerhalb einer Basisklasse implementiert. Spezifisches würde dagegen in die Ableitungen ausgelagert.

Hier könnte der Gedanke Einzug halten, die Berechnung der zuvor erwähnten eindeutigen Kennung bereits im Konstruktor von einem Repräsentanten eines Kontakts durch den Aufruf einer virtuellen Funktion durchzuführen.

Genau an dieser Stelle ist jedoch Vorsicht geboten. Womöglich ist Ihnen bei Ihrer Arbeit schon einmal die in **Bild 1** gezeigte Warnung von Visual Studio aufgefallen, die erscheint, wenn im Konstruktor eine virtuelle Funktion aufgerufen wird.

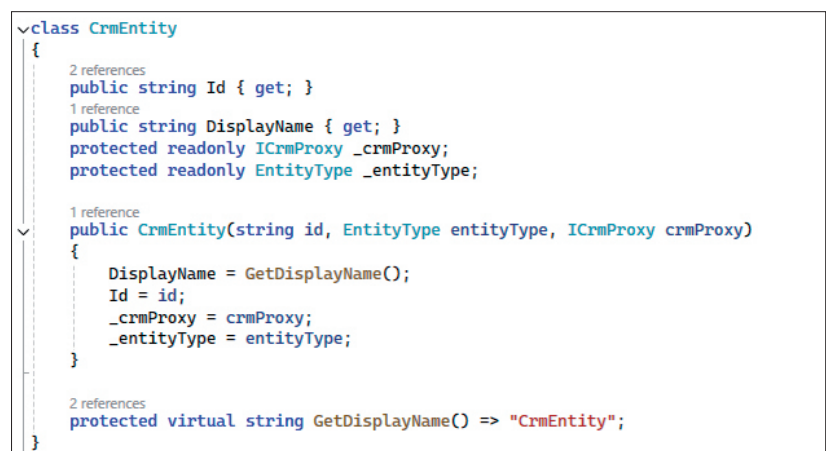
Konstruktive Lösung

Zu den Problemen, die ein solcher Aufruf verursachen kann, hat Zoran Horvat, Principal Consultant bei Coding Helmet sowie unabhängiger Trainer für den .NET-Technologie-

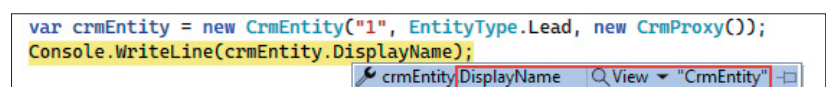


Visual Studio warnt: Überschreibbare Methoden sollte man nicht in Konstruktoren aufrufen (Bild 1)

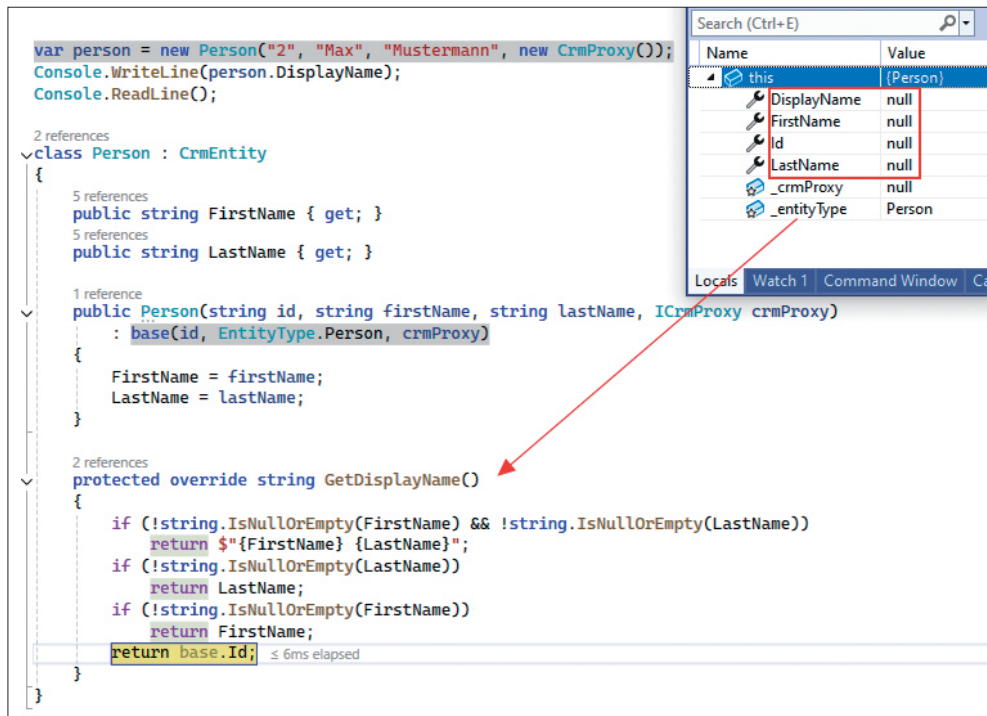
Stack, unter [2] einen interessanten Lösungsvorschlag gemacht, den ich im vorliegenden Beitrag vorstellen und anwenden möchte. Horvat löst das Problem des Zugriffs auf ►



Klasse CrmEntity: Einfache Vererbung (Bild 2)



Verwenden der Basisimplementierung von CrmEntity (Bild 3)



Die spezialisierte Klasse Person (Bild 4)

einen nicht initialisierten Zustand in seinem Artikel konstruktiv, sodass es keine virtuelle Funktion mehr gibt, die aufgerufen werden muss. Folgt man seinem Entwurf, vermeidet man die Notwendigkeit, eine virtuelle Funktion zu benutzen, um die Initialisierung des Zustands der Basisklasse im Konstruktor zu variieren. Auf diese Weise lassen sich auch alle Fallstricke vermeiden, die mit dem Zugriff auf den Zustand der nicht oder nur teilweise initialisierten Basisklasse in der Funktion der abgeleiteten Klasse verbunden sind.

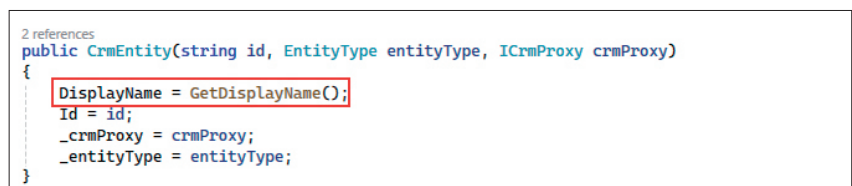
Zur Veranschaulichung der Vorgehensweise dient ein stark vereinfachtes Beispiel, bei dem Entitäten aus einem CRM-System im Code abgebildet werden. Diese werden im Programmcode durch die Klasse *CrmEntity* repräsentiert, die in Bild 2 zu sehen ist.

Der Konstruktor der Klasse erwartet eine eindeutige Kennung, nämlich den Typ der Entität und eine Referenz auf den CRM-Service. Abgeleitete Klassen sollen in der Lage sein, den Anzeigenamen selbst zu berechnen, daher ist die Funktion *GetDisplayName* als virtuelle Methode markiert. Es ist zu beachten, dass es sich hier nicht um Code aus der Praxis, sondern lediglich um ein Beispiel handelt, das bei der Erläuterung des Problems helfen soll.

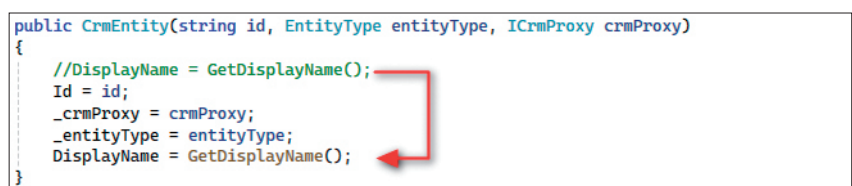
Wird die Basisimplementierung ausgeführt, wird der Anzeigename wie erwartet

geliefert und ist dabei unabhängig vom Status des Objekts, vergleiche Bild 3. Wie bereits erwähnt, kann eine abgeleitete Klasse bei Bedarf die virtuelle Funktion *GetDisplayName* überschreiben, um den Anzeigenamen entsprechend ihren Anforderungen zu berechnen. Hierfür greift die abgeleitete Klasse *Person* legitimerweise auf ihren Zustand zu, wie Sie in Bild 4 sehen. Leider funktioniert dies erst einmal nicht, weil der Zustand, also die Klassenvariablen, noch nicht gesetzt sind.

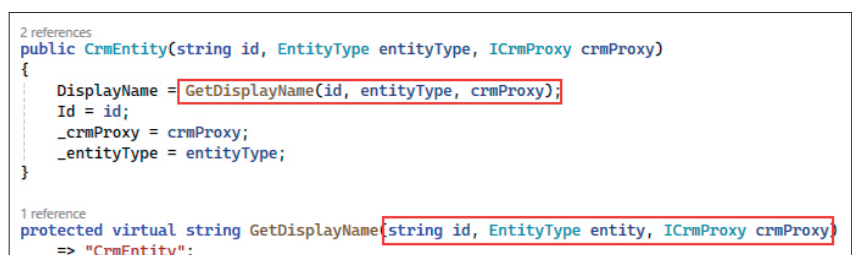
Aufmerksamen Leserinnen und Lesern ist die Ursache dafür unter Umständen schon zuvor aufgefallen. Den Fehler sehen Sie in Bild 5. Er resultiert schlicht aus der falschen Reihenfolge: Zunächst



Die Ursache des Problems (Bild 5)



Ändern der Aufrufreihenfolge liefert die Lösung (Bild 6)



Der Zustand wird hier per Parameter an die Funktion übergeben (Bild 7)

Die Lösung ist allerdings trivial: Das Ändern der Aufrufreihenfolge in der Basisimplementierung beseitigt das Problem, wie Sie in **Bild 6** sehen können.

Der Ansatz, den Zustand wie in **Bild 7** gezeigt innerhalb des Funktionsaufrufs zu übergeben, wird nur der Vollständigkeit halber erwähnt und umgehend wieder verworfen, denn es ist einfach sehr sonderbar, die Werte sowohl als Klassen-Member als auch über die Parameter zur Verfügung zu stellen

Am Kern des Problems angesetzt

Die eigentliche Ursache der Probleme mit diesem klassischen Ansatz scheint darin begründet zu sein, dass die Berechnungslogik des Anzeigenamens in der Vererbungshierarchie verteilt ist.

Und jetzt kommt der interessante Ansatz: Die Basisklasse wird so geändert, wie es **Bild 8** zeigt. Es gibt nun einen Konstruktor, der alle Werte erwartet. Die Berechnung des Anzeigenamens muss folglich außerhalb der Basisklasse erfolgen. Der zuvor dargestellte Fehler kann deshalb nicht mehr auftreten. Die schon bisher mögliche Instanziierung von *CrmEntity* klappt nun mithilfe der statischen Factory-Funktion *Create*.

Da die Basisklasse alle Werte als Eingabe erwartet, ist sie nun nicht mehr in der Lage, die erforderlichen Berechnungen durchzuführen. Diese Aufgabe müssen folglich die Ableitungen übernehmen. In **Bild 9** sehen Sie die entsprechend angepasste Version des Codes. Sie bietet eine statische Factory-Funktion an, welche die Parameter erwartet und die „komplexe“ Berechnung in einer statischen Funktion *GetDisplayName* ausführt. Deren Ergebnis wird zusammen mit den weiteren Parametern verwendet, um das gewünschte Resultat zu berechnen.

Fazit

Der von Zoran Horvat vorgestellte Ansatz ist sehr interessant: Die Verantwortlichkeiten sind klar getrennt, die abgeleitete Klasse muss sich selbst darum kümmern, alle benötigten Werte für die vollständige Instanziierung zu ermitteln. Eine virtuelle Funktion ist nun nicht mehr erforderlich und auch die in der Klassenhierarchie zuvor mehr oder minder verteilte Logik ist verschwunden – eine aus meiner Sicht recht einfache, aber gerade deshalb sehr effiziente Lösung. ■

```
class CrmEntity
{
    1 reference
    public string Id { get; }
    3 references
    public string DisplayName { get; }
    protected readonly ICrmProxy _crmProxy;

    0 references
    static CrmEntity Create(string id, ICrmProxy crmProxy)
    {
        return new CrmEntity(id, "CrmEntity", crmProxy);
    }

    2 references
    protected CrmEntity(string id, string displayName, ICrmProxy crmProxy)
    {
        Id = id;
        DisplayName = displayName;
        _crmProxy = crmProxy;
    }
}
```

Die Basisklasse ohne den Aufruf virtueller Funktionen (**Bild 8**)

```
class Person : CrmEntity
{
    0 references
    private Person Create (string id, string firstName,
        string lastName, ICrmProxy crmProxy)
    {
        var displayName = GetDisplayName(id, firstName, lastName, crmProxy);
        return new (id, displayName, crmProxy);
    }

    1 reference
    private static string GetDisplayName(string id, string firstName,
        string lastName, ICrmProxy crmProxy)
    {
        if (!string.IsNullOrEmpty(firstName) && !string.IsNullOrEmpty(lastName))
            return $"{firstName} {lastName} - {crmProxy.GetCustomerState()}";
        if (!string.IsNullOrEmpty(lastName))
            return $"{lastName} - {crmProxy.GetCustomerState()}";
        if (!string.IsNullOrEmpty(firstName))
            return $"{firstName} - {crmProxy.GetCustomerState()}";
        return $"{id} - {crmProxy.GetCustomerState()}";
    }

    1 reference
    protected Person(string id, string displayName, ICrmProxy crmProxy)
        :base(id, displayName, crmProxy)
    {
    }
}
```

Die Spezialisierung führt die Berechnungen aus (**Bild 9**)

[1] Polymorphie, www.dotnetpro.de/SL2502Konstruktor1

[2] Zoran Horvat bei Coding Helmet, www.dotnetpro.de/SL2502Konstruktor2



Christian Havel

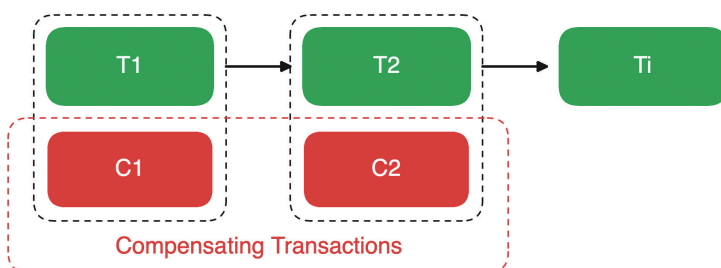
wohnt in einem Vorort von München und ist seit mehr als zwanzig Jahren in der Softwareentwicklung tätig. Er programmiert hauptsächlich in C#. Sie erreichen ihn unter christian.havel@gmail.com.

MESSAGING MIT WOLVERINE UND .NET, TEIL 8

The Saga continues

Wir lösen auf: Kreditlimit prüfen per Saga. Außerdem: Mandantenfähigkeit und ASP.NET-Core-Integration.

Zum Abschluss des vorangegangenen Teils 7 der Wolverine-Artikelserie [1] haben wir eine „Hausaufgabe“ aufgegeben: Basierend auf den gezeigten Beispielen galt es eine Saga zu implementieren, die bei Eingang einer neuen Bestellung durch einen Kunden prüft, ob dessen Kreditlimit es erlaubt, die Bestellung zu bearbeiten, oder ob das Kreditlimit durch diese Bestellung ausgeschöpft wäre und die Bestellung abgewiesen werden muss.



Transactions und Compensating Transactions einer Saga (Bild 1)

Vorwärts immer – und rückwärts?

Sagas bilden komplexe Geschäftsprozesse ab. Was wir bisher außer Acht gelassen haben: Wenn ein Schritt in einem Geschäftsprozess fehlschlägt, müssen alle vorhergehenden Schritte rückgängig gemacht werden. Bei Sagas spricht man hier von Compensating Transactions, also Transaktionen, die die vorhergehenden Schritte kompensieren.

Was sich zunächst einfach anhört, kann in der Praxis durchaus komplex werden: Was, wenn eine Compensating Transaction fehlschlägt? Konkrete Beispielgründe dafür: Das Konto für eine Rückerstattung ist erloschen, oder das Konto, von dem aus zurückerstattet werden soll, ist nicht ausreichend gedeckt.

Bild 1 zeigt, dass jede Transaktion T_i eine Compensating Transaction C_i haben muss. Das bedeutet, dass mit den Domänenexperten nicht nur über die Schritte gesprochen werden muss, die zum erfolgreichen Abschluss der Saga notwendig sind, sondern auch darüber, was im jeweiligen Fehlerfall als Compensating Transaction durchgeführt werden muss.

In Bild 2 wird verdeutlicht, dass die vorangegangene Transaktion T_1 durch die Compensating Transaction C_1 kompensiert werden muss. Gibt es weitere Schritte, die vor der Kompensation stattgefunden ha-

ben, müssen auch diese „zurückgerollt“ werden. Die Abfolge bei drei Transaktionen, bei denen die dritte fehlschlägt, ist somit: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow C_2 \rightarrow C_1$.

Rolle rückwärts

In unserem Kreditlimit-Beispiel ist der Fehlerfall, dass das Kreditlimit überschritten wurde. Die Compensating Transaction ist, dass wir die Bestellung ablehnen.

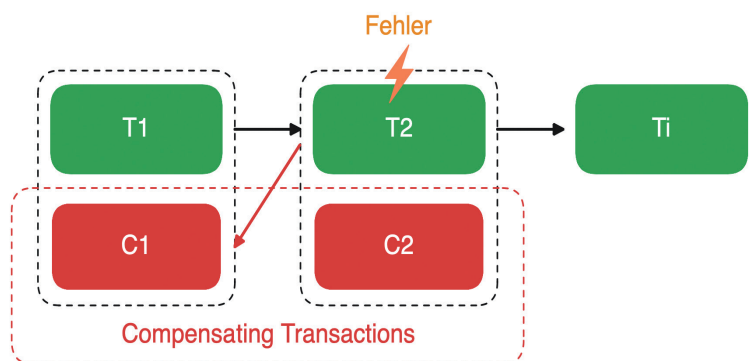
Unsere .NET-Solution besteht aus den vier Projekten, die in Bild 3 gezeigt sind. Unsere Saga liegt im Projekt *Orders*. Die neuen Bestellungen treffen per *PlaceOrder*-Kommando im *Orders*-Service ein. Die Validierung des Kreditlimits findet im *Customers*-Service statt. Die Saga besteht aus den folgenden Transaktionen T_i :

```
OrderPlaced -> ReserveCredit -> CreditReserved ->
ApproveOrder -> OrderApproved
```

Schlägt die *ReserveCredit* fehl, haben wir statt *CreditReserved* einen *CreditLimitExceeded*-Event und das Command lautet dann *RejectOrder*.

Aus Platzgründen ist nur der relevante Teil zum Roll-back der Saga in Listing 1 abgebildet. Den vollständigen Code finden Sie in den Downloads zum Artikel in der Solution *WolverineMartenCreditLimit*.

Folgender Hinweis sei zur Marten-Integration noch gegeben: In [1] haben wir gesehen, dass wir Sagas mit der Marten-Integration für Wolverine persistieren können. Dies geschieht über den Aufruf der *.IntegrateWithWolverine()*-Methode bei der Konfiguration von Marten. Der *PlaceOrderHandler* im *Order*-Service sieht wie folgt aus:



Compensating Transaction bei Fehler in T2 (Bild 2)

```

public class OrderPlacedHandler
{
    public async Task<object[]> Handle(
        PlaceOrder message,
        IDocumentSession session,
        ILogger logger
    )
    {
        var (orderId, customerId, amount) = message;
        logger.LogInformation("Received new Order: {
            OrderId}", message.OrderId);
        var orderPlaced = new OrderPlaced(
            orderId,
            customerId,
            amount
        );
        session.Events.Append(orderId, orderPlaced);
        await session.SaveChangesAsync();

        var order =
            await session.LoadAsync<PurchaseOrder>(orderId);

        return [order, orderPlaced];
    }
}

```

Die *PurchaseOrder*-Instanz wird hierbei über die Cascading Messages von Wolverine an den Client zurückgegeben, der das *PlaceOrder*-Kommando ausgelöst hat. Verwendet man die Integration mittels *.IntegrateWithWolverine()*, funktioniert dieser Ansatz allerdings nicht – dies liegt daran, wie die Integration intern arbeitet. Das bedeutet, dass man hier einen Workaround finden muss, wenn man die Integration nutzen möchte.

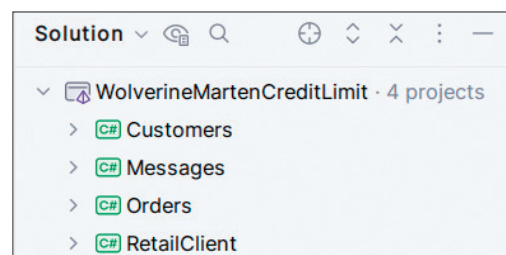
Ein Ansatz für einen solchen Workaround kann sein, dass der Client, statt die *PurchaseOrder*-Instanz als Rückgabewert des Kommando-Aufrufs zu erwarten, eine Subscription auf den *OrderPlaced*-Event einrichtet und nach Erhalt der Nachricht einen HTTP-Call auf einen Endpunkt ausführt, der die *PurchaseOrder*-Instanz über deren ID abrufbar macht.

Eine Lösung, die mit der Integration und Cascading Messages funktioniert, ist laut Maintainer nicht vor Version 4 von Wolverine zu erwarten.

HTTP und Messaging: gemeinsam besser

Die Anbindung über einen MessageBroker wie RabbitMQ, um ein Kommando auszuführen, ist in vielen Fällen nur internen Clients vorbehalten. Dennoch möchte man externen Nutzern die Möglichkeit geben, Kommandos auszuführen. Hierfür bietet sich HTTP – also ASP.NET Core – an.

Den Code für diese Lösung finden Sie in den Downloads zum Artikel in der .NET-Solution *WolverineMartenCreditLimitWeb*;



Projekte für die Kreditlimit-Saga (Bild 3)

Listing 1: Rollback der Saga

```

public class Order : Saga
{
    // weitere Methoden

    public object[] Handle(
        CreditLimitExceeded creditLimitExceeded,
        ILogger logger
    )
    {
        OrderStatus = OrderStatus.CreditLimitExceeded;
        logger.LogInformation("Saga: Credit limit
            exceeded for Order {OrderId}", Id);
        return [
            new RejectOrder(creditLimitExceeded.OrderId)];
    }

    public void Handle(
        OrderRejected orderRejected,
        ILogger logger
    )
    {
        OrderStatus = OrderStatus.Rejected;
        logger.LogWarning("Saga: Order {OrderId}
            rejected", Id);
        logger.LogInformation("Saga: Order {OrderId}
            completed", Id);
        MarkCompleted();
    }

    // weitere Methoden
}

```

er basiert auf oben gezeigtem Saga-Beispiel. Das HTTP-API, das die *PlaceOrder*-Kommandos entgegennimmt, lebt im *Order*-Service.

Unseren *HostBuilder*, der bereits Wolverine und Marten beheimatet, erweitern wir über einen Aufruf der *ConfigureWebHostDefaults*-Methode und fügen Routing und Endpoints für ASP.NET Core Minimal APIs hinzu:

```

app =>
{
    app.UseRouting();
    app.UseEndpoints(
        endpoints =>
        {
            endpoints.MapPost(
                "/api/orders",
                async (
                    HttpContext context,
                    IMessageBus bus
                ) =>
            {

```

```

    ) =>
    {
        var orderId = Guid.NewGuid().ToString();
        var order = await
            bus.InvokeAsync <PurchaseOrder>(
                new PlaceOrder(
                    orderId,
                    "janedoe@tempuri.org",
                    1000
                )
            );
        await context.Response.WriteAsync(
            $"Order {orderId} placed");
    }
};
}
};
}

```

Damit wir die Bestellungen unseren Kunden zuordnen können, implementieren wir die in [Listing 2](#) gezeigte einfache

API-Key-Middleware, die über den API-Key herausfindet, welcher Benutzer die Bestellung geschickt hat.

Im Dictionary `_apiKeys` sind unsere API-Keys den Kunden-IDs zugeordnet. In der `InvokeAsync`-Methode prüfen wir zunächst, ob der HTTP-Request-Header `X-API-KEY` vorhanden ist. Dies entspricht der Authentifizierung.

Danach folgt die Autorisierung, indem wir in `_apiKeys` einen dem übergebenen API-Key entsprechenden Eintrag suchen. Ist dieser vorhanden, wird der `HttpContext` mit einem `ClaimsPrincipal` erweitert und an den nächsten Schritt in der Middleware-Pipeline übergeben.

Die Kunden-IDs im Dictionary `_apiKeys` entsprechen den IDs der Kunden, die wir per `InitialData` – ein Marten-Feature – in den Projekten `Customers` und `RetailClient` seeden. Das Seeden der Benutzer sowie Authentifizierung und Autorisierung sind im Rahmen unseres Beispiels bewusst einfach gehalten.

Um eine Bestellung über einen HTTP-Endpunkt auslösen zu können, benötigen wir nun noch die Implementierung des Endpunkts selbst. Mit ASP.NET Core Minimal APIs sieht diese wie folgt aus:

● Listing 2: API-Key-Middleware

```

public class ApiKeyMiddleware
{
    private readonly RequestDelegate _next;
    private const string ApiKeyHeaderName = "X-API-KEY";

    private readonly IDictionary<string, string>
        _apiKeys = new Dictionary<string, string>()
    {
        { "jane-key", "janedoe@tempuri.org" },
        { "john-key", "johndoe@tempuri.org" },
        { "alice-key", "alice@tempuri.org" },
        { "bob-key", "bob@tempuri.org" }
    };

    public ApiKeyMiddleware(
        RequestDelegate next
    ) =>
    {
        _next = next;

        public async Task InvokeAsync(
            HttpContext context
        )
        {
            // Authentifizierung
            if (!context.Request.Headers.TryGetValue(
                ApiKeyHeaderName, out var extractedApiKey))
            {
                context.Response.StatusCode = 401;
                // Unauthorized
                await context.Response.WriteAsync(
                    "API Key fehlt.");
                return;
            }

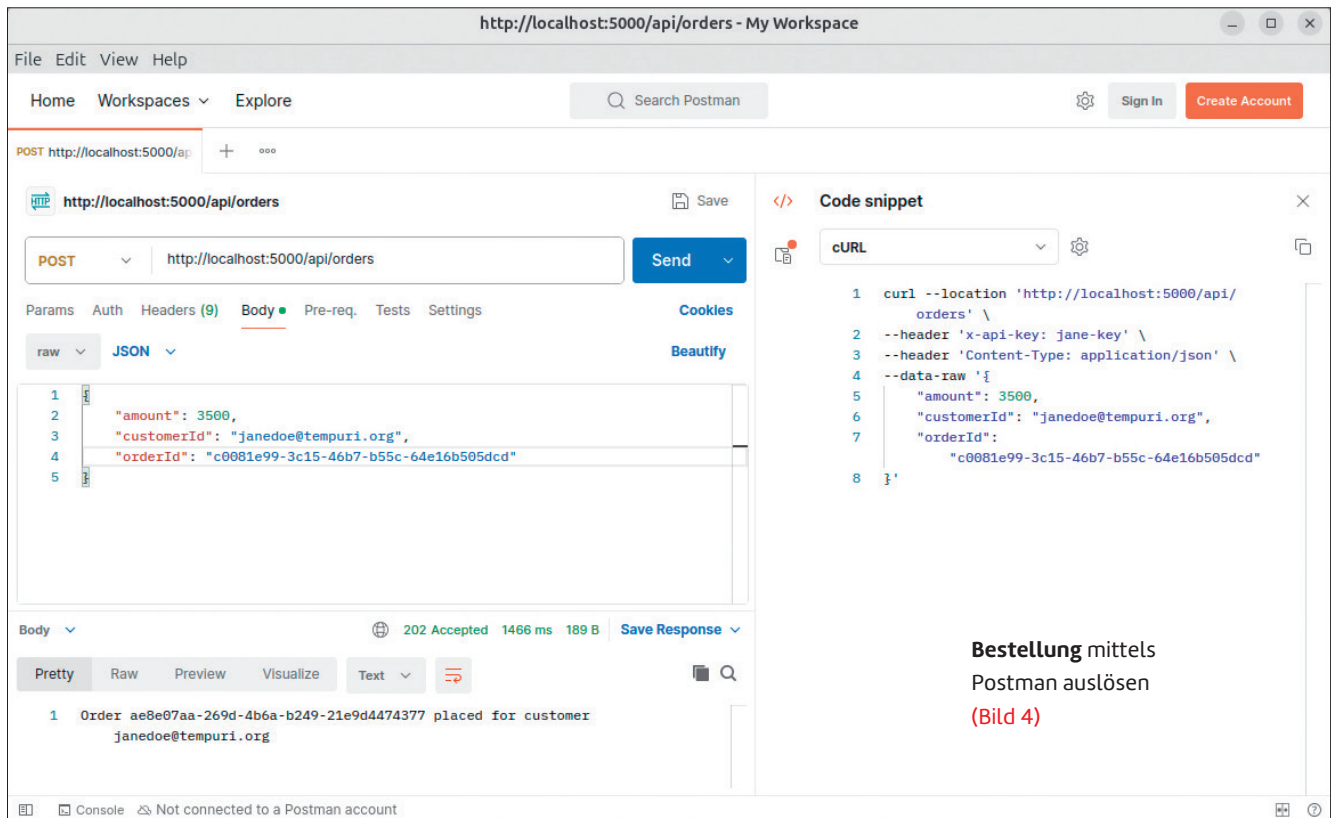
            // Autorisierung
            if (!_apiKeys.TryGetValue(extractedApiKey,
                out var username))
            {
                context.Response.StatusCode = 403;
                await context.Response.WriteAsync(
                    "Ungültiger API Key.");
                return;
            }

            var claims = new Claim[] {
                new(ClaimTypes.NameIdentifier, username) };
            context.User = new ClaimsPrincipal(
                new ClaimsIdentity(
                    claims,
                    authenticationType: "ApiKey"
                )
            );

            // Weiter zur nächsten Middleware, wenn API Key
            // gültig ist

            await _next(context);
        }
    }
}

```

**Bestellung mittels
Postman auslösen
(Bild 4)**

```
endpoints.MapPost(
    "/api/orders",
    async (
        HttpContext context,
        IMessageBus bus
    ) =>
    {
        var orderId = $"{Guid.NewGuid()}";
        var order = await bus.InvokeAsync<PurchaseOrder>(
            new PlaceOrder(
                orderId,
                context.User.GetUserId(),
                new Random().Next(1000, 10000)
            )
        );
        context.Response.StatusCode =
            StatusCodes.Status202Accepted;
        await context.Response.WriteAsync(
            $"Order {orderId} placed for customer {
                context.User.Claims.First(c => c.Type ==
                    ClaimTypes.NameIdentifier).Value}"
        );
    }
);
```

Die Implementierung entspricht weitestgehend der, die wir bisher im Worker im Projekt *RetailClient* verwendet haben: Wir erzeugen uns eine neue Order-ID, lesen den Benutzer, der zuvor über die *ApiKeyMiddleware* gesetzt wurde, aus

und erzeugen einen zufälligen Bestellwert zwischen 1000 und 10000. Verpackt als *PlaceOrder*-Nachricht senden wir dieses Kommando über unseren Wolverine-Messagebus. Dem Aufrufer signalisieren wir durch den Response-Status-Code *202 Accepted*, dass der Request angenommen wurde, die gestellte Aufgabe – das Anlegen der Bestellung – aber noch nicht erledigt wurde.

Dies ist bei Messaging-basierten-Systemen ein übliches Vorgehen. Analog sollte deshalb auch in einer grafischen Benutzeroberfläche kommuniziert werden, dass die Bestellung eingegangen ist und geprüft wird – nicht aber, dass die Bestellung erfolgreich war. Der Grund hierfür ist, dass unsere Prüfung des Kreditlimits noch fehlschlagen kann (neben weiteren möglichen Problemen, deren Behandlung nicht Gegenstand dieses Artikels ist).

Sofern der aufrufende Client den HTTP-Header *Prefer: respond-async* mit dem Request sendet, kann man zusätzlich zum Status-Code 202 einen *Location*-Header senden, der den URL zum aktuellen Status der Bestellung beinhaltet. Weitere Informationen zur sogenannten *respond-async*-Preference lassen sich in [2] nachlesen.

Nun können wir unseren Request mittels Postman – oder einem anderen Tool unserer Wahl – an den Endpunkt senden, was in **Bild 4** zu sehen ist.

Abkürzungen

Im Lauf der Artikelserie haben wir bereits festgestellt, dass Wolverine uns Hilfen an die Hand gibt, um weniger Code zu schreiben, beziehungsweise nur solchen Code, der das korrekte Vorgehen – nach Meinung des Wolverine-Teams – ►

einfacher macht („Pit of success“). So verhält es sich auch bei der Integration von HTTP, also ASP.NET Core, mit Wolverine. In Form des NuGet-Pakets `WolverineFx.Http` erhalten wir eine Bibliothek, von deren Features wir im Folgenden einige näher betrachten werden.

Wie bei anderen Wolverine-Erweiterungen beziehungsweise Integrationen muss auch diese zunächst im Host beziehungsweise mit Wolverine registriert werden. Die IoC-Integration von `WolverineFx.Http` erledigt der Aufruf von `services.AddWolverineHttp()`; an geeigneter Stelle. Der zweite Aufruf muss in der bereits angesprochenen Endpunkt-Konfiguration durchgeführt werden und sieht wie folgt aus:

```
endpoints.MapWolverineEndpoints();
```

Dieser Aufruf gibt bereits einen ersten Hinweis darauf, was die Bibliothek `WolverineFx.Http` bereitstellt. Analog zu Message-Handlern, die über Messaging adressiert werden, schafft die Bibliothek die Möglichkeit, „Endpoint-Handler“ für HTTP zu implementieren. Dieser Ansatz ist eigenständig, integriert sich jedoch mit bestehenden Ansätzen wie Minimal APIs oder MVC und kann auch Side-by-Side betrieben werden.

Die Implementierung unseres Endpunkts, den wir bereits mit Minimal APIs umgesetzt haben, sieht als Wolverine-HTTP-Endpoint wie folgt aus:

```
public class PlaceOrderEndpoint
{
    [WolverinePost("/api/orders")]
    public async Task<PurchaseOrder> PlacePurchaseOrder(
        PlaceOrderRequest placeOrderRequest,
        IMessageBus bus,
        ClaimsPrincipal principal
    )
    {
        var id = Guid.NewGuid().ToString();
        await bus.InvokeAsync<PlaceOrder>(
            new PlaceOrder(
                id,
                principal.GetUserId(),
                placeOrderRequest.Amount
            )
        );
        return new PurchaseOrder(id);
    }
}
```

Die Klasse `PlaceOrderEndpoint` besitzt eine Methode `PlacePurchaseOrder`, die mehrere Parameter definiert. `PlaceOrderRequest` ist der Typ, der den JSON-Body abbildet, welcher per Request gesendet wird. `IMessageBus` ist bereits bekannt. `IClaimsPrincipal` ist uns ebenfalls bekannt. Neu hingegen ist, wie dieser dem Endpunkt bekannt gemacht wird: nicht wie bei Minimal APIs über den `HttpContext` oder bei MVC als Eigenschaft der Basisklasse, sondern als Abhängigkeit per Method-Injection. Damit werden die Endpunkte einerseits sehr schlank und andererseits auch sehr explizit definiert.

Die Implementierung der eigentlichen Funktionalität erfolgt analog zum Ansatz mit Minimal APIs. Damit die Wolverine-Endpunkte registriert werden können und deren Route bekannt ist, werden Attribute in der Form `Wolverine<HttpVerb>(<RouteTemplate>)` definiert – hier über `[WolverinePost("/api/orders")]`. Neben `IClaimsPrincipal` können folgende weiteren Klassen von ASP.NET Core als Parameter definiert werden:

- `HttpContext`
- `HttpRequest`
- `HttpResponse`

Außerdem kann auch, wie bei den Message-Handlern, eine `CancellationToken`-Instanz angefordert werden. Ferner lassen sich, ähnlich wie bei den ASP.NET-Core-Ansätzen, die Route-Parameter als simple Typ-Abhängigkeiten wie `int`, `string` et cetera definieren.

HTTP-Header-Parameter können mittels `[FromHeader(Name = "header name")]`-Attribut gekennzeichnet werden. Als HTTP-Request-Body wird der erste Parameter erwartet, der kein simpler Typ ist wie `string`, `int` et cetera. Eine Signatur wie `(string CustomerId, PlaceOrderRequest placeOrderRequest)`, bei der `CustomerId` ein Route-Parameter wäre, ist also gültig. Abhängigkeiten, die nicht unter die genannten Regeln fallen, versucht die Bibliothek über den IoC-Container aufzulösen.

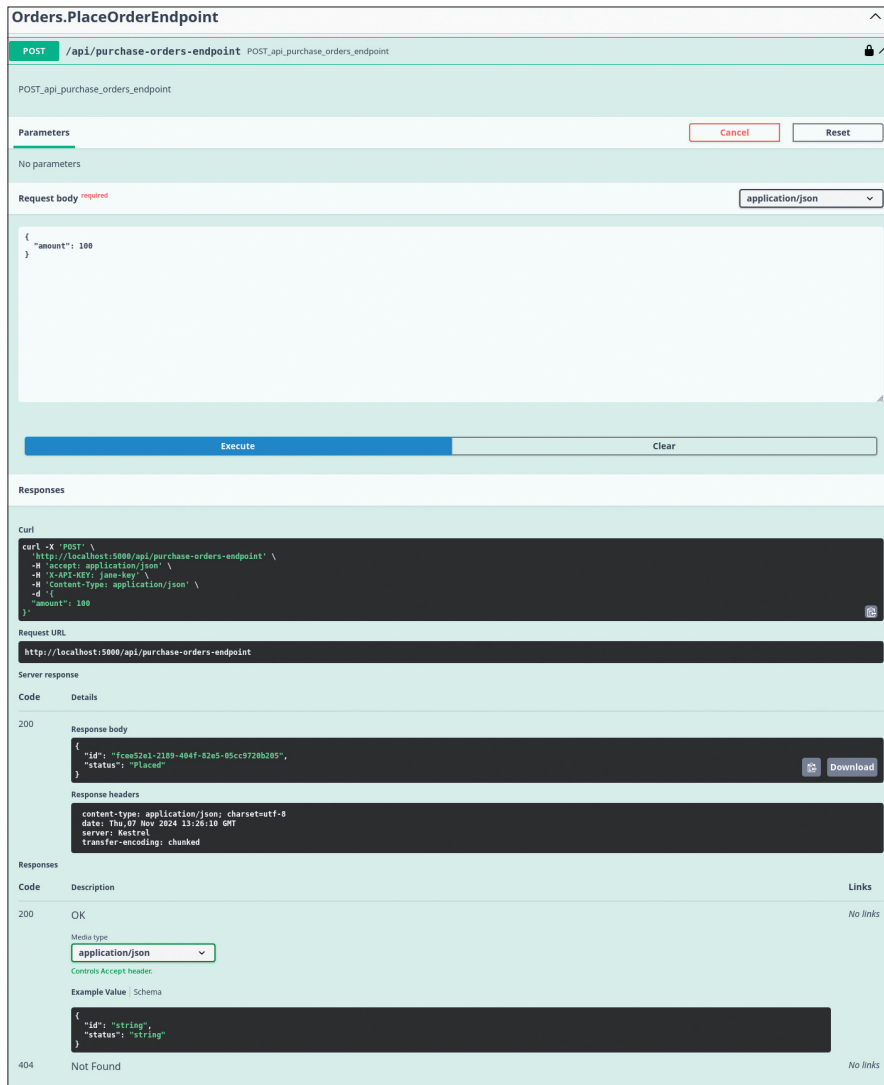
Selbstverständlich funktioniert `WolverineFx.Http` mit den Authentifizierungs- und Autorisierungsmechanismen von ASP.NET Core. Verwendet unser Wolverine-HTTP-Endpoint Marten, um Dokumente oder Events zu speichern und im Anschluss Nachrichten über den `IMessageBus` zu versenden, wird dies automatisch über die in [1] vorgestellte Outbox-Funktionalität mit transaktionalem Verhalten robust und zuverlässig abgebildet.

In unserem Szenario haben wir die Funktionalität des bereits vorhandenen `PlaceOrderHandler` als Wolverine-HTTP-Endpoint dupliziert und nochmals implementiert. Hierbei gibt es allerdings noch eine einfachere Lösung: Hat man bereits einen Message-Handler wie in unserem Szenario, kann man auf diesen zurückgreifen und mittels folgender Endpunkt-Konfiguration einen HTTP-Endpoint „über den Message-Handler stützen“:

```
endpoints.MapPostToWolverine<PlaceOrder,
    PurchaseOrder>("/api/orders");
```

Bei dieser Variante steckt im Methodennamen, was er tut: Ein HTTP-Request wird an einen Message-Handler weitergeleitet. Der erste generische Parameter ist hierbei der Typ der eingehenden Nachricht, der zweite ist der Rückgabety – sowohl des Message- als auch den Endpoint-Handlers. Schließlich wird noch die Route spezifiziert, unter der dieser Endpunkt erreichbar sein soll.

Wie bereits erwähnt, integriert sich `WolverineFx.Http` in bestehende ASP.NET-Core-Umgebungen. Dies geht sogar so weit, dass zum Beispiel OpenAPI-Integration („Swagger“) für Wolverine-HTTP-Endpunkte bereitgestellt wird.



Das Swagger-UI nach erfolgreichem Request (Bild 5)

Damit wir das Beispiel testen können, müssen wir unsere API-Key-Authentifizierung noch bei Open API bekannt machen:

```
services.AddSwaggerGen(
    options =>
    {
        options.AddSecurityDefinition(
            "ApiKey",
            new OpenApiSecurityScheme()
            {
                Name = "X-API-KEY",
                In = ParameterLocation.Header,
                Type = SecuritySchemeType.ApiKey,
                Description = "Authorization by x-api-key inside request's header",
                Scheme = "ApiKeyScheme"
            }
        );

        options.AddSecurityRequirement(
            new OpenApiSecurityRequirement
            {
```

```

        {
            new
            {
                OpenApiSecurityScheme()
            }
            {
                Reference = new
                {
                    OpenApiResponse
                }
                {
                    Type = ReferenceType.
                        SecurityScheme,
                    Id = "ApiKey"
                },
                In = ParameterLocation.
                    Header
            },
            new List<string>()
        }
    );
};
```

Bild 5 zeigt das Swagger-UI nach erfolgreichem Request inklusive API-Key für die Route des Wolverine-Endpoints.

Nun haben wir bereits etliche Merkmale von WolverineFx.Http gesehen. Darüber hinaus bietet es aber noch weitere Möglichkeiten zur Konfiguration, wie zum Beispiel Middleware oder auch die Verwendung von *IResult* für die Responses. Alle diese Optionen im Detail aufzuführen würde allerdings den Rahmen dieses Artikels

sprengen. Deshalb sei an dieser Stelle auf die Dokumentation zu WolverineFx.Http unter [3] verwiesen.

Eins, zwei, viele

Bisher funktioniert unsere Webapplikation nur für einen Mandanten. Dies wollen wir nun ändern, indem wir sowohl Marten als auch Wolverine und die ASP.NET-Core-Integration für Mandantenfähigkeit einrichten. Den Beispiel-Code finden Sie in der .NET-Solution *WolverineMartenCreditLimitSaaS* in den Downloads zum Artikel.

Die Herausforderung, die sich stellt, ist, vom eingehenden HTTP-Request bis zum Speichern der *PurchaseOrder*-Projektion durchgängig die ID des Mandanten (*TenantId*) zur Verfügung zu haben. Darüber hinaus muss die Tenant-ID auch serviceübergreifend transportiert und wiederverwendet werden können – in unserem Fall im *Customers*-Service. Glücklicherweise unterstützen uns die genannten Komponenten hierbei.

Beginnen wir mit dem Request. Hier greift unsere *ApiKey*-Middleware zunächst den API-Key ab und setzt den *IClaimsPrincipal*. Bisher hatten wir hier keine Tenant-ID. Diese setzen wir nun auf den gleichen Wert wie die Customer-ID. Hierzu erweitern wir unser *Claims*-Array um einen Eintrag: ►

```
var claims = new Claim[]
{
    new(ClaimTypes.NameIdentifier,
        username),
    new("tenant", username)
};
```

Im nächsten Schritt müssen wir sicherstellen, dass die Wolverine-HTTP-Integration im *Orders-Service* diesen Claim ausliest und weiterverwendet. Bisher haben wir dem Aufruf von *endpoints.MapWolverineEndpoints* noch keine Konfiguration übergeben und mit den Standardeinstellungen gearbeitet. Nun konfigurieren wir die Endpoints so, dass der *tenant*-Claim als Tenant-ID verwendet wird:

```
endpoints.MapWolverineEndpoints(
    options => options.TenantId.IsClaimTypeNamed("tenant")
);
```

Neben der Möglichkeit, die Tenant-ID aus einem Claim zu lesen, bietet WolverineFx.Http noch weitere Optionen, wie zum Beispiel HTTP-Header, Route-Parameter oder Subdomains. Mit diesen beiden Änderungen ist bereits sichergestellt, dass die Mandanten-ID über den Request bis hin zum Senden der *PlaceOrder*-Message verwendet und auch über den Message Broker, also RabbitMQ, mit der Nachricht versendet wird.

Im nächsten Schritt muss unser *PlaceOrderHandler* die Tenant-ID wieder auslesen können. Sehen wir uns hierzu die Signatur der *Handle*-Methode erneut an:

```
public async Task<Object[]> Handle(
    PlaceOrder message,
    IDocumentSession session,
    ILogger logger
)
```

Auf den ersten Blick wirkt es, als sei es nicht möglich, aus einem der bereits definierten Parameter die gesendete Tenant-ID zu erhalten. Tatsächlich aber existieren sogar zwei Möglichkeiten.

Die erste Variante sieht wie folgt aus: Über die Method-Injection können wir uns eine *Envelope*-Instanz übergeben lassen. Hierbei handelt es sich um den „Umschlag“, in dem die Nachricht verpackt ist. Dieser beinhaltet, ähnlich wie ein echter Briefumschlag, weitere Informationen – Metadaten – zur Nachricht. Unter anderem enthält die Instanz auch die Eigenschaft *TenantId*. Diese könnten wir dann für unsere *IDocumentSession* verwenden (zum Beispiel mit der Methode *session.ForTenant(envelope.TenantId)*).

Allerdings geht es noch wesentlich eleganter – mit der zweiten Variante: Indem wir bei der Marten-Konfiguration die bereits bekannte Methode *.IntegrateWithWolverine()* aufrufen (sofern nicht bereits geschehen), können wir nun einfach die *IDocumentSession* unverändert verwenden. Im Hintergrund sorgt die Marten-Integration dann dafür, dass

A-z type	timestamp	A-z tenant id
customer_credit_limit_set	2024-11-07 16:30:39.413 +0100	janedoe@tempuri.org
customer_credit_limit_set	2024-11-07 16:30:39.414 +0100	johndoe@tempuri.org
customer_credit_limit_set	2024-11-07 16:30:39.414 +0100	alice@tempuri.org
customer_credit_limit_set	2024-11-07 16:30:39.414 +0100	bob@tempuri.org
credit_reserved	2024-11-07 16:31:41.066 +0100	janedoe@tempuri.org
order_approved	2024-11-07 16:31:41.391 +0100	janedoe@tempuri.org

Unsere eigene Tenant-ID in Anwendung (Bild 6)

die Session bereits die korrekte Tenant-ID enthält. Gleiches gilt natürlich auch, falls wir in einem Handler über den IMessageBus weitere Nachrichten publizieren. Auch dieser verwendet dann bereits die entsprechende Tenant-ID. Im *Customers-Service* müssen wir nur die Methode *.IntegrateWithWolverine()* an der Marten-Konfiguration ergänzen, da es dort keine HTTP-Endpunkte gibt, die wir aufrufen könnten. Bild 6 zeigt, dass die Einträge in der Datenbank nun nicht mehr mit dem Mandanten *DEFAULT* gespeichert werden, sondern unsere eigene Tenant-ID verwenden.

Damit war der Umstieg von einer nicht mandantenfähigen hin zu einer mandantenfähigen Webapplikation mit nur wenigen Konfigurationsschritten und praktisch keinerlei Handler-Code-Anpassungen erledigt. Dies zeigt, wie durchdacht sowohl Wolverine und Marten als auch deren Ergänzungen wie die ASP.NET-Core-Integration sind.

Fazit

In diesem Artikel haben wir mehrere Themen behandelt: Wir haben das Verständnis für Sagas noch einmal vertieft und damit die im vorangegangenen Teil angestrebte Lösung zu einer verteilten und robusten Kreditprüfung fertiggestellt. Diese haben wir zunächst um ein HTTP-API mit Authentifizierung/Autorisierung und OpenAPI erweitert. Schließlich haben wir die Lösung dann noch mandantenfähig gemacht.

Damit biegt unsere Serie zu Wolverine auf die Zielgerade ein. Im letzten Teil werden wir uns unter anderem mit AWS SQS und Wolverine beschäftigen. ■

- [1] Alexander Zeitler, *Sagahafte Zusammenarbeit*, dotnetpro 1/2025, Seite 68 ff., www.dotnetpro.de/A2501Wolverine
- [2] *respond-async-Preference*, RFC7240, www.dotnetpro.de/SL2502Wolverine1
- [3] *Dokumentation zu WolverineFx.Http*, www.dotnetpro.de/SL2502Wolverine2



Alexander Zeitler

entwickelt als Cloud Solutions Architect B2B-Plattformen im industriellen Umfeld mit Schwerpunkt AWS, .NET und Node.js. Er unterstützt Unternehmen mit Architekturberatung und Entwicklung für Event-basierte Systeme.

alexander.zeitler@pdmlab.com

dnpCode

A2502Wolverine

SPRACHMODELLE PER OLLAMA API ANSPRECHEN

Kontakt zur KI

Binden Sie verschiedene große Sprachmodelle unter .NET mit OllamaSharp an.

Die Integration von leistungsfähigen KI-Modellen wird zunehmend zu einer Schlüsselanforderung in modernen Softwareprojekten. Die C#-Bibliothek OllamaSharp ermöglicht die einfache Anbindung von Modellen wie Llama 3.2, Mistral und Gemma 2 in eigene Projekte. Sie bietet eine nahtlose Schnittstelle, um Text- und Bildanalysen mit leistungsfähigen Modellen umzusetzen, ohne tiefgreifende KI-Kenntnisse vorauszusetzen.

Ein praktisches Beispiel für diese Integration ist Llama 3.2 Vision, ein KI-Modell, das speziell Bildanalysen unterstützt. Dieses Modell kombiniert visuelle Verarbeitung mit textuel-

ler Llama 3.2 Vision, wie OllamaSharp die Nutzung von Modellen im Entwicklungsprozess vereinfacht und welche Anwendungen sich damit realisieren lassen.

OllamaSharp: Grundlagen und Funktionen

Wie erwähnt dient die C#-Bibliothek als Schnittstelle zwischen C#-Anwendungen und den KI-Modellen, wodurch sich die Funktionalitäten der Modelle schnell und effizient in Projekte integrieren lassen. OllamaSharp macht die Nutzung von KI-Technologien nicht nur zugänglicher, sondern reduziert auch den Entwicklungsaufwand.

```

Windows PowerShell
run      Run a model
stop     Stop a running model
pull     Pull a model from a registry
push     Push a model to a registry
list     List models
ps       List running models
cp       Copy a model
rm       Remove a model
help     Help about any command

Flags:
-h, --help      help for ollama
-v, --version   Show version information

Use "ollama [command] --help" for more information about a command.
Fabia@BELLEROPHON > ollama list
NAME ID SIZE MODIFIED
Fabia@BELLEROPHON > ollama pull llama3.2-vision
pulling manifest
pulling 11f274007f09... 100% 6.0 GB
pulling ece5e659647a... 100% 1.9 GB
pulling 715415638c9c... 100% 269 B
pulling 0b4284c1f870... 100% 7.7 KB
pulling fefc914e46e6... 100% 32 B
pulling fbd313562bb7... 100% 572 B
verifying sha256 digest
writing manifest
success
  
```

Das erfolgreiche Laden eines Modells auf das lokale System (Bild 1)

ler Beschreibung, was es ideal für Szenarien wie Objekterkennung, Bildklassifikation und automatisierte Bildbeschreibungen macht. Mit Llama 3.2 Vision lässt sich eine Vielzahl von Anwendungen realisieren, von der automatisierten Inhaltsanalyse bis hin zu benutzerzentrierten Lösungen wie visueller Barrierefreiheit.

In dieser Episode der Kolumne Frameworks und mehr liegt der Fokus auf dem Zugriff auf das Ollama API mithilfe der Bibliothek OllamaSharp. Dieser Beitrag zeigt am Beispiel von

Die Bibliothek bietet vielseitige Funktionen, darunter die Verarbeitung natürlicher Sprache und die Analyse visueller Inhalte. Mit OllamaSharp lassen sich problemlos Text-Prompts an Modelle wie Llama senden, die Antworten empfangen und diese nahtlos in eigene Anwendungen einbetten.

Besonders interessant ist die Unterstützung der neuesten Llama-Modelle wie Llama 3.2 Vision, die speziell für Bildanalysen optimiert sind. Anwendungsfälle reichen von Chatbots über Textgenerierung bis hin zu komplexen Bildver- ►

arbeitungsaufgaben wie Objekterkennung und Bildbeschreibungen. OllamaSharp unterstützt eine breite Palette an Modellen der Llama-Familie, darunter textzentrierte Varianten sowie multimodale Modelle wie Llama 3.2 Vision.

Die Bibliothek bietet zudem eine flexible Konfiguration, um spezifische Modellversionen auszuwählen, die den Anforderungen des Projekts entsprechen. Darüber hinaus ist die Bibliothek einfach zu integrieren und weist eine hohe Usability auf. Das Projekt bietet intuitive Methoden und eine nützliche Dokumentation, um die Interaktion mit KI-Modellen zu vereinfachen.

Trotz ihrer Vorteile bringt die Nutzung von OllamaSharp auch Herausforderungen mit sich. Die Modelle der Llama-Reihe sind rechenintensiv und erfordern leistungsstarke Hardware, insbesondere bei großen Modellversionen wie Llama 3.2 Vision. Dies kann die Nutzung für Entwickler mit begrenzten Ressourcen einschränken.

Darüber hinaus sind die Modellgrößen und deren Speicherbedarf zu berücksichtigen, da diese eine erhebliche Menge an Arbeitsspeicher und weiteren Ressourcen beanspruchen können. Diese Einschränkungen gelten für die Nutzung von lokalen Modellen. Wenn über die Schnittstelle Remote-Modelle angesprochen werden, lassen sich diese Anforderungen zwar nicht plötzlich ausknipsen, aber sinnvoll auf andere Systeme verlagern.

Was ist das Ollama API?

Das Ollama API ist eine Programmierschnittstelle, die den Zugriff auf verschiedene große Sprachmodelle (LLMs) ermöglicht. Mit diesem API lassen sich Modelle wie Llama 3.2, Mistral, Gemma 2 und andere lokal auf dem System ausführen und in eigene Anwendungen integrieren. Es bietet Endpunkte für verschiedene Funktionen, darunter Textgenerierung, Chat-Komplettierungen, eine Modellverwaltung zum Auflisten, Anzeigen, Kopieren, Löschen, Herunterladen und Hochladen von Modellen, sowie Einbettungen für das Generieren von Vektorrepräsentationen für Texte.

Das API ist so konzipiert, dass es sowohl für lokale als auch für serverseitige Anwendungen geeignet ist. Es unterstützt Streaming-Antworten, was besonders nützlich für Anwendungen ist, die Echtzeit-Feedback erfordern. Daneben ist das API kompatibel mit verschiedenen Plattformen, einschließlich macOS, Linux und Windows. Für alle, die in Python arbeiten, steht die Ollama-Python-Bibliothek zur Verfügung, die eine einfache Integration des API in Python-Projekte ermöglicht.

OllamaSharp versus LLamaSharp

Sowohl OllamaSharp als auch LLamaSharp bieten leistungsstarke Schnittstellen, um KI-Modelle wie Llama (Large Language Model Meta AI) in C#-Projekten zu integrieren.

OllamaSharp ist für die Interaktion mit dem Ollama API entwickelt worden. Die Bibliothek dient als Wrapper, um die Anbindung von Llama-Modellen (wie Llama 3.2 Vision) in C# zu vereinfachen. OllamaSharp bietet eine nahtlose API-Integration, multimodale Unterstützung und einen Fokus auf das Ollama-Ökosystem.

● Tabelle 1: Vergleich von OllamaSharp und LLamaSharp

Aspekt	OllamaSharp	LLamaSharp
Abhängigkeit	Ollama API erforderlich	Direkte Nutzung von Modellen
Installation	Einfach über NuGet	Mehr manuelle Einrichtung
Unterstützte Modelle	Modelle des Ollama-Ökosystems	Flexible Unterstützung für viele Modelle
Multimodale Fähigkeiten	Optimiert für Bild- und Textverarbeitung	Textfokussiert, Anpassung für Vision-Modelle möglich
Performance	Abhängig von der Ollama-Serverkonfiguration	Optimierbar durch direkte Modellverwaltung

LLamaSharp ist eine generische Bibliothek, die für die direkte Integration von Llama-Modellen ohne ein spezifisches API entwickelt wurde. Sie bietet eine Modellunterstützung ohne API, eine hohe Flexibilität bei der Modellnutzung und mehr Kontrolle über Modelle. **Tabelle 1** vergleicht OllamaSharp mit LLamaSharp und gibt eine Übersicht zur Abgrenzung und Einschätzung der beiden Bibliotheken.

Bildmächtig: Llama 3.2 Vision

Llama 3.2 Vision markiert einen deutlichen Fortschritt in der Entwicklung multimodaler KI-Modelle. Im Vergleich zu früheren Versionen der Llama-Reihe bringt diese Version entscheidende Neuerungen, die ihre Leistungsfähigkeit speziell in der Bildverarbeitung und bei multimodalen Anwendungsfällen erheblich steigern.

Eine der wichtigsten Neuerungen von Llama 3.2 Vision ist die erweiterte Fähigkeit, Bild- und Textdaten gleichzeitig zu verarbeiten.

Während frühere Llama-Versionen vor allem auf Textverarbeitung spezialisiert waren, ermöglicht die 3.2-Vision-Version eine tiefere Integration von visuellen und textbasierten Informationen.

Das Modell kann Bilder nicht nur analysieren, sondern auch beschreiben, Muster erkennen und mit Text-Prompts interagieren, die visuelle Kontexte einbeziehen.

Das CLI von Ollama

Das Backend für die Tests in diesem Artikel ist ein laufender Ollama-Server. Das Projekt steht für verschiedene Plattformen zur Verfügung und lässt sich dort problemlos installieren. Mitinstalliert wird ein CLI, um zum Beispiel Modelle zu verwalten. Dazu gehört das Laden und Starten eines Modells. **Bild 1** zeigt einen Screenshot, wie das Modell llama3.2-vision erfolgreich über den Befehl

```
ollama pull llama3.2-vision
```

auf den lokalen Rechner geladen wird. Anschließend zeigt der Befehl `ollama list` alle nun vorhandenen Modelle an:

Listing 1: Das erste Beispiel zur Abfrage von Modellen in OllamaSharp

```
using OllamaSharp;

const string URL = "http://localhost:11434";

var ollama = new OllamaApiClient(URL);
// Verbindung zum lokalen Ollama-Server

var models = await ollama.ListLocalModelsAsync();

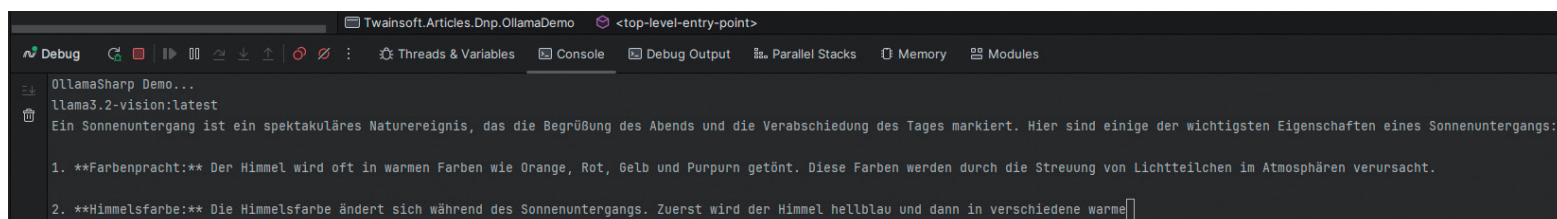
foreach (var model in models)
{
    Console.WriteLine(model.Name, model.Size,
        model.Details);
}

// ... weitere Verarbeitung ...
```

```
NAME      llama3.2-vision:latest
ID        38107a0cd119
SIZE      7.9 GB
MODIFIED   About a minute ago
```

Desktop-Projekten nutzen. Diese Vielseitigkeit macht die Bibliothek ideal für unterschiedliche Projekttypen.

Die Installation von OllamaSharp ist über das NuGet-Paket am einfachsten, beispielsweise über die Paket-Manager-



Screenshot einer gestreamten Antwort in der Konsole von JetBrains Rider (Bild 2)

Die Ausgabe zeigt die Größe des Modells und wann es heruntergeladen wurde. Der Befehl `ollama show llama3.2-vision` zeigt weitere Details an, wie der folgende Ausschnitt zeigt:

```
architecture    mllama
parameters      9.8B
context length   131072
embedding length 4096
quantization     Q4_K_M
```

Zu guter Letzt startet `ollama run <Modell>` ein Sprachmodell, sodass es über eine lokale Adresse zur Verfügung steht.

Technische Grundlagen und Installation

OllamaSharp unterstützt .NET 6.0 und höher und lässt sich sowohl in Konsolenanwendungen als auch in Web-API- oder

Konsole oder direkt im .NET-CLI, wie der folgende Befehl zeigt:

```
dotnet add package OllamaSharp
```

OllamaSharp benötigt außerdem eine funktionierende Installation des Backends. Die Abhängigkeiten umfassen ein REST-API, das entweder lokal oder in der Cloud betrieben werden kann.

Details zur Konfiguration des Ollama-Servers finden sich in der offiziellen Dokumentation [1]. Die Beispiele in diesem Artikel basieren auf dem Ollama-Projekt, mit dem sich große und kleine Sprachmodelle lokal ausführen lassen.

Welche Modelle zur Verfügung stehen, beschreibt die Projektwebseite [2] zu Ollama. Aktuell steht das Projekt in Version 4.0.8 von Ende November 2024 zur Verfügung. ►

Listing 2: Die Analyse eines Bilds über die Chat-Funktion

```
const string imagePath = "data/sunset.jpg";

var imageBytes = new[] {
    File.ReadAllBytes(imagePath) };

var chat = new Chat(ollama);

await foreach (var answerToken in chat.SendAsync(
    "Was ist auf dem Bild zu sehen?", imageBytes))
{
    Console.WriteLine(answerToken);
}
```

Es steht unter der MIT-Lizenz und bietet den Quelltext auf GitHub [3]. Nach der Installation reicht es für einen ersten Anwendungsfall aus, den Namensraum *OllamaSharp* einzubinden.

Listing 1 zeigt ein einfaches Beispiel, wie *OllamaSharp* in einem Projekt zum Einsatz kommt. Der Code verdeutlicht, wie die Verbindung zum Ollama-Server aufgebaut werden muss. Über die Schnittstelle lassen sich anschließend alle bereits heruntergeladenen Modelle auflisten, was in diesem Beispiel zur Ausgabe des Llama-3.2-Vision-Modells führt, das zuvor über das CLI geladen wurde.

Primäre Funktionen der Bibliothek

Eine der zentralen Funktionen von *OllamaSharp* ist die Bereitstellung eines benutzerfreundlichen API-Wrappers, der die Interaktion mit Llama-Modellen erheblich vereinfacht. Anstatt komplexe HTTP-Anfragen manuell zu implementieren, bietet die Bibliothek Schnittstellen in Form von Methoden und Eigenschaften an, um Text- und Bilddaten an das Modell zu senden und die Antworten zu verarbeiten.

Der API-Wrapper abstrahiert die zugrunde liegenden technischen Details und stellt eine intuitive Schnittstelle bereit, die sowohl asynchrone als auch synchrone Aufrufe unterstützt.

Das erste Beispiel im **Listing 1** hat bereits die Methode *AnalyzeImageAsync* gezeigt, mit der direkt eine Bildanalyse angestoßen werden kann. Andere Methoden ermöglichen es, allgemeinere Prompts an das verknüpfte Modell zu schicken, wie der folgende Codeausschnitt demonstriert:

```
ollama.SelectedModel = "llama3.2-vision";

await foreach (var stream in
    ollama.GenerateAsync("Beschreibe die Eigenschaften " +
        "eines Sonnenuntergangs."))
    Console.Write(stream?.Response);
```

Der vorherige Code-Ausschnitt zeigt, wie das Modell im Ollama-Backend selektiert wird, damit das Backend weiß, an

welches Modell die Anfragen geschickt werden sollen. Die Antworten werden anschließend gestreamt und auf der Konsole ausgegeben – in dieser Demo in JetBrains Rider, wie der Screenshot in **Bild 2** zeigt.

Neben Text- und Bilddaten kann Llama 3.2 Vision auch interaktive Chats implementieren, sodass mit recht wenig Aufwand interaktive Anwendungen implementierbar sind.

Ein Beispiel für die Nutzung der Chat-Funktionen zeigt der nachfolgende Code-Ausschnitt:

```
var answerToken in chat.SendAsync(message);
```

Use-Case: Implementierung einer Bildanalyse

Für die Implementierung einer Bildanalyse mit *OllamaSharp* und Llama 3.2 Vision ist der Download einer unterstützten Llama-3.2-Vision-Modellversion notwendig, die sich von der offiziellen Plattform herunterladen und im Ollama-Server einrichten lässt. Anschließend starten Sie den Server mit folgendem Befehl:

```
ollama run llama3.2-vision
```

Die zurückgegebene URL ist wichtig für die Verbindung aus dem Beispielcode heraus. Die Initialisierung des Ollama-Clients ist sehr ähnlich zum ersten Beispiel.

Nachdem diese Grundkonfiguration abgeschlossen ist, lässt sich die Funktionalität zur Analyse eines Bildes implementieren. Das verläuft recht ähnlich zur Verarbeitung aus unserem ersten Beispiel. Wir nutzen einen Chat, um die Anfragen aus diesem Chat heraus an das Modell zu schicken und das Bild anzuhängen. Den Code dazu zeigt **Listing 2**.

Diese Implementierung zeigt, wie einfach es ist, *OllamaSharp* und das Modell Llama 3.2 Vision in einem C#-Projekt einzusetzen, um Bilder zu analysieren und aussagekräftige Ergebnisse zu liefern.

Weiterführende Themen und Optimierungen

Die Nutzung von Llama 3.2 Vision mit *OllamaSharp* erfordert eine effiziente Ressourcennutzung, da KI-Modelle, vor allem

● Listing 3: Die Kombination von *OllamaSharp* und *Microsoft.Extensions.AI*

```
var builder = WebApplication.CreateBuilder(args);

// Hinzufügen von OllamaSharp zu
// Microsoft.Extensions.AI
builder.Services.AddAIService<OllamaService>(
    options =>
    {
        options.BaseUrl = "http://localhost:11434";
        // Ollama-Server-URL
        options.DefaultModel = "llama-3";
    });

var app = builder.Build();

app.MapGet("/chat", async (
    IOllamaService ollamaService, string input) =>
{
    var response =
        await ollamaService.QueryAsync(input);
    return response.Text;
});

app.Run();
```


● Listing 4: Integration mit Microsoft Semantic Kernel

```
var kernel = Kernel.Builder
    .ConfigureAIService<OllamaService>(config =>
    {
        config.BaseUrl = "http://localhost:11434";
        config.DefaultModel = "llama-3";
    })
    .Build();

string prompt =
    "Schreibe einen kurzen Blogpost über KI.";
var result = await kernel.RunAsync(prompt);
Console.WriteLine(result.GetResult());
```

solche für Bildverarbeitung, sehr rechenintensiv sind. Um die Performance möglichst hoch zu halten, lassen sich einige Optimierungspotenziale ausschöpfen.

Beispielsweise ist eine Batch-Verarbeitung denkbar, um mehrere Anfragen gleichzeitig zu senden. Auch Caching ist ein Thema.

Ergebnisse häufiger Analysen lokal zu speichern, um wiederholte Berechnungen zu vermeiden, ist in vielen Szenarien eine sehr gute Idee. Das betrifft insbesondere Szenarien, in denen externe Modelle bei Anbietern angesprochen werden müssen, die bei jeder Anfrage Kosten produzieren.

Zudem lassen sich die Modellparameter anpassen und optimieren. Parameter wie *MaxTokens* oder *Temperature* lassen sich einstellen, um die Berechnungszeit zu reduzieren, ohne die Ergebnisqualität erheblich zu beeinträchtigen.

Für die optimale Leistung empfiehlt sich zudem die Nutzung einer GPU. Um GPU-Beschleunigung zu aktivieren, ist die Installation der entsprechenden CUDA-Treiber und -Bibliotheken notwendig. Anschließend ist der Server neu zu starten, was die GPU-Unterstützung aktiviert.

Unterstützung für Microsoft.Extensions.AI und Microsoft Semantic Kernel

OllamaSharp bietet nahtlose Unterstützung für Microsoft.Extensions.AI [4], ein Framework, das die Integration von KI-Services in .NET-Anwendungen vereinfacht. Dies ermöglicht es, KI-Modelle in bestehende Pipelines zu integrieren, ohne viel Boilerplate-Code schreiben zu müssen.

Mit OllamaSharp lassen sich beispielsweise lokale oder remote bereitgestellte KI-Modelle mit wenigen Zeilen Code nutzen. Ein einfaches Beispiel für die Integration in eine Anwendung zeigt Listing 3. Hier wird der Ollama-Service als KI-Dienst konfiguriert und in einem Web-API verfügbar gemacht.

Microsoft Semantic Kernel [5] ist ein leistungsfähiges Framework für KI-gestützte Anwendungen, das auf Plug-and-Play-Funktionen setzt.

OllamaSharp erweitert das Semantic-Kernel-Framework, indem es als KI-Plug-in verwendet werden kann. Damit lassen sich KI-Funktionen wie Textgenerierung, Codierung oder Wissensabfragen effizient integrieren.

Die Beispielimplementierung in Listing 4 veranschaulicht schließlich die Nutzung von OllamaSharp innerhalb des Semantic Kernel. Dieses Beispiel zeigt, wie OllamaSharp direkt in ein Kernel-Setup integriert wird, um eine flexible und skalierbare KI-Architektur bereitzustellen.

Fazit

Die Integration von KI-Modellen ist in der modernen Softwareentwicklung essenziell. Das Ollama API und die dazugehörige C#-Bibliothek OllamaSharp ermöglichen die einfache Nutzung leistungsstarker Sprach- und Vision-Modelle wie Llama 3.2 Vision. Mit ihrer benutzerfreundlichen Schnittstelle, den multimodalen Fähigkeiten und der nahtlosen Integration in .NET-Projekte erleichtert OllamaSharp den Zugang zu KI-Technologien erheblich, selbst ohne tiefgehende KI-Kenntnisse.

Während Herausforderungen wie hoher Ressourcenbedarf bestehen, bietet die Bibliothek flexible Konfigurationsmöglichkeiten, um individuelle Anforderungen zu erfüllen.

OllamaSharp zeigt, wie leistungsfähige KI-Funktionen einfach in Projekte integriert und vielseitige Anwendungsfälle – von Text- und Bildanalysen bis hin zu interaktiven Chats – realisiert werden können. Wer Modelle lokal betreiben möchte, um mit diesen zu experimentieren oder sie in produktive Anwendungen zu integrieren, sollte einen Blick auf Ollama und OllamaSharp werfen. Die Bibliothek verdient sich ein „Gut +“ und eine Empfehlung. An der Dokumentation lässt sich sicherlich noch arbeiten. Zugleich empfohlen seien das Projekt LlamaSharp und das Llama-Backend. ■

- [1] Das GitHub-Repository zu Ollama, www.dotnetpro.de/SL2502Frameworks1
- [2] Übersicht der Modelle in Ollama auf der Projektwebseite, <https://ollama.com/library>
- [3] Das GitHub-Repository zu OllamaSharp, www.dotnetpro.de/SL2502Frameworks2
- [4] Informationen zu Microsoft.Extensions.AI, www.dotnetpro.de/SL2502Frameworks3
- [5] Informationen zu Microsoft Semantic Kernel, www.dotnetpro.de/SL2502Frameworks4



Dr. Fabian Deitelhoff

arbeitet als IT-Leiter, Head of Software Development und Head of Product bei der Education Partners GmbH. Darüber hinaus ist er als Autor, Dozent und Softwareentwickler im .NET- und Webumfeld tätig.

@FDeitelhoff

dnPCode

A2502Frameworks

Anpassen oder erweitern?

Um spezifischere Anforderungen zu erfüllen, brauchen Sprachmodelle etwas „Nachhilfe“. Zwei Methoden hierfür sind Fine-Tuning und Retrieval-Augmented Generation (RAG).

Die Entwicklung von künstlicher Intelligenz (KI) hat mit dem Aufkommen von Large Language Models (LLMs) einen neuen Meilenstein erreicht [1]. Das betrifft auch die Möglichkeiten, ein solches aufwendig vortrainiertes Modell auf sehr spezialisierte Aufgaben oder Bereiche abzustimmen: Mit Techniken wie Fine-Tuning [2] und Retrieval-Augmented Generation (RAG) [3] lassen sich bestehende Modelle schnell und gezielt anpassen beziehungsweise erweitern, ohne dass das Modell von Grund auf neu trainiert werden muss.

Das traditionelle Training nutzt einen festen Datensatz, um das Modell zu trainieren, was zeit- und ressourcenintensiv ist. Heutige Methoden erlauben es, vortrainierte Modelle auf spezifische Anforderungen zuzuschneiden (via Fine-Tuning) oder externe Wissensquellen dynamisch zu integrieren ((via Retrieval-Augmented Generation, RAG). Dies eröffnet eine größere Flexibilität, ohne die Vorteile des vorherigen Trainings zu verlieren.

Je nach Anwendungsfall ist die eine oder die andere Vorgehensart mehr oder weniger komplex und sinnvoll. Der Artikel stellt beide Varianten vor, versucht sich an einer praxistauglichen Abgrenzung und zeigt mittels Codebeispielen, wie ein RAG-System mit Semantic Kernel funktioniert. Das ist eine gute Fingerübung und eine passende Vorbereitung für eine der künftigen Ausgaben dieser Serie, wenn ein RAG-System mit Azure und Microsofts Copilot umgesetzt werden soll – zwei Varianten, die im Unternehmenskontext erfahrungsgemäß häufig angefragt und implementiert sind.

Fine-Tuning als Modellanpassung

Fine-Tuning ermöglicht es, die Leistung und Genauigkeit eines KI-Modells für spezifische Anwendungsfälle zu optimieren, ohne es komplett neu trainieren zu müssen. Large Language Models (LLMs) wie GPT sind bereits mit enormen Datenmengen trainiert, um eine breite Sprachkompetenz zu entwickeln. Durch Fine-Tuning können sie gezielt mit einem kleineren, domänenspezifischen Datensatz nachtrainiert werden. Ziel ist es, das Modell so anzupassen, dass es präzise und relevant für bestimmte Aufgaben oder Branchen bleibt, ohne die allgemeinen Sprachfähigkeiten zu verlieren.

Die Methodik des Fine-Tuning erfordert eine Kombination aus geeigneten Daten, Ressourcen und Tools. Für erfolgrei-

● Listing 1: Datenstruktur für den zu trainierenden Datensatz

```
support_data = [
    {"query": "Mein Laptop startet nicht mehr", "category":
      "Hardware-Probleme"},
    {"query": "Ich kann mich nicht in mein Konto einloggen", "category":
      "Kontoprobleme"},
    {"query": "Die Software stürzt ständig ab", "category":
      "Software-Probleme"},
    # ... weitere Einträge
]
```

ches Fine-Tuning sind qualitativ hochwertige und repräsentative Datensätze notwendig. Diese sollten gut kuratiert, ausreichend groß und auf den Anwendungsbereich zugeschnitten sein. Das Nachtrainieren von Modellen ist rechenintensiv und erfordert leistungsfähige Hardware, wie GPUs oder TPUs, um die Rechenzeit zu minimieren. Plattformen wie Microsoft Azure, OpenAI oder Hugging Face bieten spezialisierte APIs und Frameworks, die den Fine-Tuning-Prozess erleichtern. Python-Bibliotheken wie PyTorch oder TensorFlow unterstützen die technische Implementierung.

Der Fine-Tuning-Prozess umfasst in der Praxis mehrere Schritte:

- Laden eines vortrainierten Modells: Als Ausgangspunkt wird ein bereits auf großen Datensätzen trainiertes Modell verwendet, wie beispielsweise GPT-4 von OpenAI.
- Modellanpassung: Das vortrainierte Modell wird nach dem Laden mit einem spezifischen, kleineren Datensatz weiter trainiert. Dabei werden nur bestimmte Gewichte des Modells angepasst, um eine Überanpassung zu vermeiden.
- Hyperparameter-Optimierung: Wichtige Parameter wie Lernrate und Losgröße werden optimiert, um die besten Ergebnisse zu erzielen.

Die Lernrate bestimmt, wie groß die Schritte sind, die der Optimierungsalgorithmus (zum Beispiel Stochastic Gradient Descent, Adam) bei jedem Iterationsschritt macht, um die Modellparameter zu aktualisieren. Sie ist ein Skalierungsfaktor, der den Grad der Anpassung an den Fehler signalisiert. Oft kommt beim Fine-Tuning eine kleinere Lernrate zum Einsatz als beim vollständigen Training, um bereits gelernte Gewichte nicht drastisch zu ändern und Overfitting zu vermeiden.

Die Losgröße (Batch Size) gibt an, wie viele Datenproben gleichzeitig durch das Modell laufen, bevor die Parameter basierend auf dem aggregierten Fehler aktualisiert werden. Beim Fine-Tuning kommen oft mittlere Losgrößen zum Einsatz, da sehr kleine Losgrößen beim Update stärker mit Rauschen belastete Ergebnisse verursachen können und große Losgrößen viel Rechenleistung und Speicher erfordern.

Fine-Tuning eröffnet vielfältige Möglichkeiten in unterschiedlichen Bereichen. Unternehmen können Modelle für branchenspezifische Anforderungen wie juristische Texte, medizinische Berichte oder technische Dokumentationen anpassen. Modelle lassen sich so trainieren, dass sie den spezifischen Sprachstil, die Terminologie und die Prozesse eines Unternehmens verstehen. Das führt zu personalisierten und effizienteren Interaktionen mit Kunden. In der Forschung lassen sich LLMs durch Fine-Tuning zum Beispiel auf Fachliteratur anpassen, um gezielte Unterstützung bei der Analyse oder Erstellung wissenschaftlicher Texte zu bieten.

Fine-Tuning am Beispiel

Nehmen wir als konkreten Anwendungsfall ein Technologieunternehmen, das ein Kundensupport-System verbessern möchte. Das Ziel ist es, eingehende Support-Anfragen automatisch in verschiedene Kategorien einzuordnen, um sie effizienter an die zuständigen Abteilungen weiterzuleiten. Für dieses Beispiel dient Googles Gemini-Modell [4] als Ausgangspunkt. Der Datensatz besteht aus einer Sammlung von technischen Support-Anfragen, die in Kategorien wie „Hardware-Probleme“, „Software-Probleme“ und „Kontoprobleme“ eingeteilt sind. Für das Fine-Tuning ist zunächst der Datensatz vorzubereiten. Die Datenstruktur ist in [Listing 1](#) zu sehen. Die Daten sind zu bereinigen und in Token zu überführen, was [Listing 2](#) zeigt. Nun lässt sich das Modell initialisieren und das Fine-Tuning starten, was in [Listing 3](#) abgebildet ist. Ist der Fine-Tuning-Prozess abgeschlossen, kann das Modell für Vorhersagen genutzt werden (siehe [Listing 4](#)).

● Listing 2: Umwandlung der Daten zu Tokens

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(
    "google/gemini-base")

def preprocess_data(data):
    queries = [item["query"].lower() for item in data]
    categories = [item["category"] for item in data]

    tokenized_inputs = tokenizer(queries, padding=True,
                                  truncation=True, return_tensors="pt")
    return tokenized_inputs, categories

inputs, labels = preprocess_data(support_data)
```

Dieses Beispiel zeigt, wie sich ein vortrainiertes Gemini-Modell für die spezifische Aufgabe der Kategorisierung von Support-Anfragen feinabstimmen lässt. In der Praxis kommt verständlicherweise ein deutlich umfangreicherer Datensatz zum Einsatz, als an dieser Stelle in einem Artikel abbildbar ist. Und sehr wahrscheinlich kommen weitere Optimierungen zum Tragen, um die Leistung des Modells und der Kategorisierung zu verbessern.

RAG-Systeme als Modellerweiterung

Retrieval-Augmented Generation (RAG) ist ein Konzept, das die Leistungsfähigkeit von großen Sprachmodellen (LLMs) mit externen Datenquellen kombiniert. Ziel ist es, die Qualität und Relevanz von Antworten durch den Zugriff auf aktuelle und spezifische Informationen zu erhöhen. LLMs sind zwar auf umfangreichen Datensätzen trainiert, können ►

● Listing 3: Der Fine-Tuning-Prozess

```
from transformers import
    AutoModelForSequenceClassification, Trainer,
    TrainingArguments
import torch

num_labels = len(set([item["category"] for item in
    support_data]))
model = AutoModelForSequenceClassification.from_
    pretrained("google/gemini-base",
    num_labels=num_labels)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=8,

    per_device_eval_batch_size=8,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=inputs,
    eval_dataset=inputs, # In der Praxis würden Sie hier
    # einen separaten Evaluierungsdatensatz verwenden
)

trainer.train()
```

● Listing 4: Nutzung der Daten für eine Vorhersage

```
def predict_category(query):
    inputs = tokenizer(query, return_tensors="pt")
    outputs = model(**inputs)
    predicted_class = torch.argmax(
        outputs.logits, dim=1).item()
    return list(set([item["category"] for item in
        support_data]))[predicted_class]

# Beispielanfrage
test_query = "Meine E-Mails werden nicht
    synchronisiert"
predicted_category = predict_category(test_query)
print(
    f"Vorhergesagte Kategorie: {predicted_category}")
```

ohne zusätzliche Informationen aber veraltet oder unzureichend auf kontextspezifische Anfragen reagieren. RAG löst dieses Problem, indem es zunächst relevante Informationen aus externen Wissensdatenbanken, Dokumenten oder APIs abrufen, was der „Retrieval“-Teil ist, und diese dynamisch in die Antwortgenerierung, den „Generation“-Teil, integriert. Dies erfolgt durch eine zweistufige Pipeline: Im ersten Schritt filtert ein Retrieval-Modell relevante Inhalte heraus, die anschließend vom LLM verarbeitet werden. Auf diese Weise bleiben die Modelle flexibel und lassen sich ohne kostspieliges Nachtraining für spezifische Anwendungen nutzen.

Ein zentraler Vorteil von RAG besteht darin, dass keine permanente Modelländerung erforderlich ist. Statt ein Modell ständig mit neuen Daten nachzutrainieren, setzt RAG auf die dynamische Einbindung externer Quellen. Dies führt zu einer deutlichen Zeit- und Kosteneinsparung. Darüber hinaus bleibt das System durch die Einbindung aktueller Datenquellen stets auf dem neuesten Stand, was insbesondere in schnelllebigen Branchen von großer Bedeutung ist. Kon-

textspezifische Antworten können durch den direkten Zugriff auf spezifische Dokumente oder Datenbanken präziser und relevanter gestaltet werden. Zudem verbessert RAG die Skalierbarkeit, da bestehende Modelle für unterschiedliche Anwendungsfälle wiederverwendet werden können, ohne individuelle Anpassungen vornehmen zu müssen.

Die Anwendungsbereiche von RAG sind vielfältig. In Wissensdatenbanken, wie sie etwa in technischen Supportsystemen oder internen Unternehmenslösungen genutzt werden, erlaubt RAG den Zugriff auf umfassende und aktuelle Informationen. Unternehmen können so Mitarbeitenden oder Kunden präzise Antworten auf komplexe Fragen bieten.

Ein weiteres Einsatzgebiet liegt in der Echtzeit-Dokumentation, beispielsweise in der medizinischen Forschung oder juristischen Analyse, wo stets die neuesten Erkenntnisse berücksichtigt werden müssen.

Darüber hinaus wird RAG in Unternehmensanwendungen eingesetzt, etwa bei der Automatisierung von Kundenserviceleistungen oder bei der Erstellung von personalisierten Berichten und Analysen. Auch für die Entwicklung smarter Assistenzsysteme, die kontextspezifisches Wissen benötigen, eignet sich RAG hervorragend.

Fine-Tuning versus RAG-System

Häufig werden Fine-Tuning und RAG-Systeme „in einen Topf geworfen“ beziehungsweise verwechselt. Anhand einiger anschaulicher Vergleiche soll daher der Unterschied zwischen den Methoden aufgezeigt werden, um eine passende Auswahl im Projektkontext zu ermöglichen. Die Vergleiche verdeutlichen, dass Fine-Tuning eine tiefgreifende Anpassung des Modells selbst darstellt, während RAG eine flexiblere Methode ist, um das Modell mit externen Informationen anzureichern, ohne seine Grundstruktur zu verändern.

So lässt sich Fine-Tuning beispielsweise mit einem Auswendiglernen von Informationen vergleichen, bei dem das Modell spezifisches Wissen internalisiert. RAG dagegen ermöglicht dem Modell, bei Bedarf auf externe Informationen zuzugreifen, ähnlich dem Nachschlagen in einem Buch. Auch die intensive Schulung eines Mitarbeiters, um diesem neues

● Listing 5: Initialisieren des Kernels in C# für Semantic Kernel

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.Memory.Qdrant;
using Microsoft.SemanticKernel.Memory;

// Initialisierung des Kernels
IKernel kernel = new KernelBuilder()
    .WithLoggerFactory(ConsoleLogger.LoggerFactory)
    .WithOpenAIChatCompletionService(
        "gpt-3.5-turbo",
        "YOUR_OPENAI_API_KEY")
    .WithOpenAITextEmbeddingGenerationService(
        "text-embedding-ada-002",
        "YOUR_OPENAI_API_KEY")
    .Build();

// Initialisierung des Qdrant-Speichers
const string qdrantEndpoint = "http://localhost:6333";
const string vectorSize = "1536"; // Für OpenAI Ada-002
var qdrantMemoryStore =
    new QdrantMemoryStore(qdrantEndpoint, vectorSize);

// Hinzufügen des Speichers zum Kernel
kernel.UseMemory(qdrantMemoryStore);
```


● Listing 6: Importieren der Dokumente als Chunks in den Vektorstore

```
public async Task ImportDocumentAsync(
    string documentText, string documentId)
{
    const string memoryCollectionName =
        "DocumentCollection";
    const int maxChunkSize = 1000;

    var lines = documentText.Split('\n');
    var currentChunk = new StringBuilder();
    var chunkIndex = 0;

    foreach (var line in lines)
    {
        if (currentChunk.Length + line.Length >
            maxChunkSize)
        {
            await kernel.Memory.SaveInformationAsync(
                memoryCollectionName,
                currentChunk.ToString(),
                $"{documentId}-chunk-{chunkIndex}");
            currentChunk.Clear();
            chunkIndex++;
        }
        currentChunk.AppendLine(line);
    }

    if (currentChunk.Length > 0)
    {
        await kernel.Memory.SaveInformationAsync(
            memoryCollectionName,
            currentChunk.ToString(),
            $"{documentId}-chunk-{chunkIndex}");
    }
}
```

Wissen zu vermitteln, kann als Analogie zum Fine-Tuning dienen. RAG würde dem Mitarbeiter dagegen Arbeitsanweisungen und Nachschlagewerke zur Verfügung stellen, die er bei Bedarf konsultieren kann. Und noch ein Beispiel aus einem ganz anderen Bereich: Man kann sich Fine-Tuning auch wie das Maßschneidern eines Anzugs vorstellen, während RAG eher Accessoires zu einem Standardanzug hinzufügt, um ihn für bestimmte Anlässe zu optimieren.

Ein erstes RAG-System mit Semantic Kernel

Das nachfolgende Beispiel demonstriert die Implementierung eines RAG-Systems mit Semantic Kernel in C#. Dieses System lässt sich leicht mit neuen Informationen aktualisieren, ohne das gesamte Modell neu trainieren zu müssen. Das System kann zudem stets auf die neuesten Informationen zugreifen, die in der Vektordatenbank gespeichert sind. Die Implementierung ermöglicht es darüber hinaus, die Quellen der verwendeten Informationen nachzuvollziehen, was bei Fine-Tuning schwieriger ist. Die RAG-Implementierung erfordert weniger Rechenressourcen als das kontinuierliche Fine-Tuning großer Sprachmodelle. RAG kann leicht an spezifische Domänen oder Anwendungsfälle angepasst werden, indem relevante Dokumente in die Wissensbasis aufgenommen werden.

Das vorgestellte Beispiel zeigt, wie Dokumente in Chunks aufgeteilt, als Vektoren gespeichert und bei Abfragen effizient abgerufen werden können, um kontextbezogene und präzise Antworten zu generieren. Zunächst initialisieren wir den Semantic Kernel und die benötigten Dienste, was in [Listing 5](#) zu sehen ist. Dieser Code-Abschnitt erstellt den Kernel inklusive der notwendigen Konfiguration und den notwendigen Diensten. Zum Einsatz kommt OpenAI für die Textvollständigung und die Generierung der Einbettungen (Text Embeddings). Außerdem initialisiert der Code einen Qdrant-Speicher [5] als Vektordatenbank für das RAG-System.

Als Nächstes ist eine Funktion zum Import und Speichern der Dokumente notwendig. [Listing 6](#) zeigt den erforderlichen C#-Code. Diese Funktion teilt ein Dokument in Chunks auf und speichert diese im Qdrant-Speicher. Wir verwenden eine maximalen Chunk-Size von 1000 Zeichen, um die Effizienz der Einbettungen zu gewährleisten.

Chunks sind ein zentraler Bestandteil von RAG-Systemen, da sie Effizienz und Genauigkeit steigern. Große Dokumente lassen sich besser durchsuchen, wenn sie in kleinere Teile (Chunks) zerlegt werden, wodurch spezifische Informationen präziser gefunden werden können. Sprachmodelle haben begrenzte Kontextgrößen, und Chunks ermöglichen eine effiziente Verarbeitung innerhalb dieser Grenzen, was die Qualität des Abgleichs zwischen Anfrage und Dokument verbessert. Sie erhöhen die Relevanz, indem sie irrelevante Teile großer Dokumente ausblenden und sich auf die wesentlichen Abschnitte konzentrieren. Zudem reduzieren sie die Belastung der Systemressourcen, da kleinere Einheiten die Verarbeitung beschleunigen.

Die optimale Chunk-Größe in einem RAG-System hängt von einer Balance zwischen Informationsgehalt und Modellgrenzen ab. Die Kontextgröße des Modells bestimmt die maximale Länge eines Chunks, beispielsweise 2048 Token bei GPT-3. Chunks sollten etwa 50 bis 70 Prozent dieser Kapazität nutzen, um Platz für Anfragen und Überlappungen zu lassen. Überlappende Chunks, etwa durch ein Sliding-Window-Verfahren, helfen, wichtige Informationen an Chunk-Grenzen nicht zu verlieren. Dabei sollten Chunks thematisch oder inhaltlich zusammenhängend sein, um eigenständige Antworten zu ermöglichen. Die ideale Größe variiert je nach Anwendungsfall und erfordert Tests, bei denen Metriken wie Relevanz und Genauigkeit berücksichtigt werden. Als Heuristik gelten 200 bis 300 Wörter für allgemeine Texte oder 100 bis 150 Wörter für stark strukturierte Inhalte als sinnvoll. ►

● Listing 7: Die primäre RAG-Funktion für eine RAG-Abfrage

```
public async Task<string> QueryDocumentsAsync(
    string query)
{
    const string memoryCollectionName =
        "DocumentCollection";
    const int maxRelevantChunks = 5;

    var relevantMemories = await kernel.Memory.SearchAsync(
        memoryCollectionName,
        query,
        limit: maxRelevantChunks,
        minRelevanceScore: 0.7);

    var context = new StringBuilder();
    context.AppendLine("Relevant information:");
    foreach (var memory in relevantMemories)
    {
        context.AppendLine(memory.Metadata.Text);
    }

    context.AppendLine("Query: " + query);

    var promptTemplate = @"
    Based on the following information and query,
    provide a concise and accurate answer:

    {context}

    Answer:";

    var prompt = promptTemplate.Replace(
        "{context}", context.ToString());

    var result =
        await kernel.InvokeSemanticFunctionAsync(prompt);

    return result.ToString();
}
```

Damit sind die initialen Vorbereitungen abgeschlossen. Zeit, um die primäre Funktion für die RAG-Abfrage zu implementieren. Dazu werfen wir einen Blick auf den Code in [Listing 7](#). Diese Funktion führt verschiedene Schritte aus. Zunächst sucht der Code nach relevanten Dokumenten-Chunks basierend auf der Abfrage. Aus diesen relevanten Chunks lässt sich dann ein Kontext erstellen. Unter Verwendung dieses Kontexts und der ursprünglichen Abfrage lässt sich anschließend eine Antwort erzeugen, die das Sprachmodell berechnet.

[Listing 8](#) zeigt die Nutzung des RAG-Systems unter der Annahme, dass sich der zuvor gezeigte Code in der Klasse *RAG-System* befindet. Zunächst lassen sich die notwendigen Dokumente importieren, was in einem realen System nur einmal initial passiert und dann lediglich bei Änderungen. Anschließend ist eine Abfrage möglich, um Informationen aus dem RAG-System zu beziehen. Diese Informationen basieren auf der Anfrage und dem ermittelten Kontext der relevanten Dokumente.

Relevante Chunks finden

Das Finden relevanter Chunks in einem Vektorstore basiert auf der semantischen Ähnlichkeit zwischen der Benutzeranfrage (Query) und den gespeicherten Chunks. Dazu wird zunächst jeder Chunk eines Dokuments mit einem Embedding-Modell, wie OpenAI Embeddings [6] oder Sentence-BERT [7], in einen Vektor umgewandelt, der dessen semantische Bedeutung repräsentiert. Diese Vektoren werden zusammen mit Metadaten wie Chunk-ID oder Dokumentquelle im Vektorstore gespeichert.

Zur Laufzeit transformiert der Code die Benutzeranfrage ebenfalls in einen Vektor, wobei dasselbe Embedding-Modell zum Einsatz kommt, um die Anfrage in den gleichen semantischen Vektorraum zu projizieren. Anschließend vergleicht

der Vektorstore den Query-Vektor mit den gespeicherten Vektoren, typischerweise anhand von Ähnlichkeitsmaßen wie Kosinusabstand, Euklidischer Distanz oder Dot-Produkt. Um

● Listing 8: Nutzung des RAG-Systems in C#

```
class Program
{
    static async Task Main(string[] args)
    {
        var ragSystem = new RAGSystem(); // Angenommen,
        // die obigen Methoden sind in dieser Klasse

        // Dokumente importieren
        await ragSystem.ImportDocumentAsync(
            "Inhalt von Dokument 1...", "doc1");
        await ragSystem.ImportDocumentAsync(
            "Inhalt von Dokument 2...", "doc2");

        // RAG-Abfrage durchführen
        var query = "Was ist die Hauptaussage der importierten Dokumente?";
        var answer =
            await ragSystem.QueryDocumentsAsync(query);

        Console.WriteLine($"Frage: {query}");
        Console.WriteLine($"Antwort: {answer}");
    }
}
```

die Suche bei großen Datenmengen effizient zu gestalten, kommen Algorithmen für Approximate Nearest Neighbor (ANN) wie FAISS, Milvus oder Pinecone zum Einsatz.

Die gefundenen Chunks lassen sich anhand ihrer Ähnlichkeit zum Query-Vektor sortieren, wobei die höchstbewerteten als die relevantesten gelten. Optional sind Filter nützlich, um nur Chunks mit bestimmten Metadaten, etwa zu einer spezifischen Kategorie oder einem Zeitrahmen, einzubeziehen. Schließlich gibt der Vektorstore die relevantesten Chunks zurück, die entweder direkt als Antwort verwendet oder für weitere Verarbeitungsschritte wie Zusammenfassung oder Antwortgenerierung durch ein Sprachmodell genutzt werden.

Beispielsweise könnte eine Anfrage wie „Was ist die Funktion von Mitochondrien?“ in einen Vektor umgewandelt werden. Der Vektorstore durchsucht daraufhin seine gespeicherten Vektoren nach ähnlichen und gibt relevante Chunks mit Informationen über Mitochondrien zurück. Dieser Ansatz ermöglicht es RAG-Systemen, präzise und effizient relevante Informationen abzurufen.

Frameworks

Mit einer Vielzahl von verfügbaren Frameworks wie Semantic Kernel, LangChain oder PromptFlow stehen Entwicklern leistungsstarke Tools zur Verfügung, um die jeweiligen Stärken von Fine-Tuning und RAG optimal auszuschöpfen.

Semantic Kernel ist ein Open-Source-Framework von Microsoft, das modulare KI-Funktionalitäten wie Sprachmodelle oder APIs als „Skills“ integriert. Es zeichnet sich durch Gedächtnisfunktionen und Plug-in-basierte Architekturen aus, wodurch personalisierte und wiederverwendbare Anwendungen möglich sind. Das Framework ist ideal für die Orchestrierung verschiedener KI-Modelle in Chatbots, digitalen Assistenten und Workflow-Automatisierung.

PromptFlow optimiert Prompts für Sprachmodelle (LLMs) und bietet eine visuelle Benutzeroberfläche zur Workflow-Erstellung und -Testung. Dank der Integration in Azure-AI-Services unterstützt es Unternehmen mit Tools wie A/B-Tests zur systematischen Verbesserung von Modellausgaben. PromptFlow ist wie gemacht für Teams, die KI-generierte Flows optimieren möchten.

LangChain ist ein Open-Source-Framework zur Entwicklung kontextspezifischer KI-Anwendungen. Es kombiniert LLMs mit externen Datenquellen wie Datenbanken und APIs und ist bekannt für Retrieval-Augmented Generation (RAG), das dynamische und aktuelle Antworten ermöglicht. Das Framework ist besonders geeignet für datenintensive Anwendungen wie Wissensmanagement, Echtzeit-Analysen oder personalisierte Assistenten. **Tabelle 1** fasst die Eigenschaften und Unterschiede in den Ansätzen zusammen.

● **Tabelle 1: Vergleich der Ansätze von Semantic Kernel, PromptFlow und LangChain**

Kriterium	Semantic Kernel	PromptFlow	LangChain
Fokus	KI-Skills-Integration	Prompt-Optimierung	Datenintegration und RAG
Flexibilität	Modularer Aufbau	Speziell für LLM-Prompts	Vielseitige Datenanbindung
Nutzergruppe	Entwickler, Unternehmen	Prompt-Designer, Analysten	Entwickler, Datenwissenschaftler
Integration	Plug-ins, APIs	Azure AI	Verschiedene Datenquellen
Besonderheit	Gedächtnisfunktion	Visuelle Workflow-Tools	RAG und externe Datenquellen

Fazit und Ausblick

Fine-Tuning und Retrieval-Augmented Generation sind leistungsstarke Ansätze zur Anpassung und Erweiterung großer Sprachmodelle. Fine-Tuning erlaubt eine gezielte Spezialisierung von Modellen auf spezifische Anforderungen, während RAG durch die dynamische Integration externer Wissensquellen flexible und aktuelle Anwendungen ermöglicht.

Beide Ansätze haben klare Stärken: Fine-Tuning glänzt bei hochspezifischen und wiederkehrenden Aufgaben, bei denen maximale Präzision erforderlich ist, während RAG-Systeme eine effiziente und skalierbare Lösung für kontextabhängige und dynamische Anforderungen bieten.

Der Artikel zeigt nicht nur die technischen Grundlagen, sondern auch praxisnahe Umsetzungen und verdeutlicht die Einsatzmöglichkeiten in verschiedenen Branchen. Das dient als gute Vorbereitung für eine nächste Ausgabe, in der eine konkrete RAG-Umsetzung für ein Unternehmen im Fokus steht. Dieses Mal mit Microsofts Copilot als vergleichende Implementierung. ■

[1] Fabian Deitelhoff, *Über Sprache lernen*, dotnetpro 12/2024, Seite 74 ff., www.dotnetpro.de/A2412KIundData
 [2] Weitere Informationen zu Fine-Tuning, <https://www.it-p.de/lexikon/fine-tuning/>
 [3] Weitere Informationen zu RAG, www.dotnetpro.de/SL2502KIundData1
 [4] Googles Gemini-Modell, <https://gemini.google.com/>
 [5] Qdrant Vektordatenbank, <https://qdrant.tech>
 [6] OpenAI Embeddings, www.dotnetpro.de/SL2502KIundData2
 [7] SBERT, <https://sbert.net>



Dr. Fabian Deitelhoff
 arbeitet als IT-Leiter, Head of Software Development und Head of Product bei der Education Partners GmbH. Darüber hinaus ist er als Autor, Dozent und Softwareentwickler im .NET- und Webumfeld tätig.
@FDeitelhoff

dnpCode	A2502KIundData
---------	----------------



DATA ACCESS

Die Datenzugriffsberatung mit DR. HOLGER SCHWICHTENBERG

PERFORMANCE UNTERSCHIEDLICHER LADESTRATEGIEN FÜR VERBUNDENE OBJEKTE BEI EF CORE

Verschieden schnelle Wege nach Rom

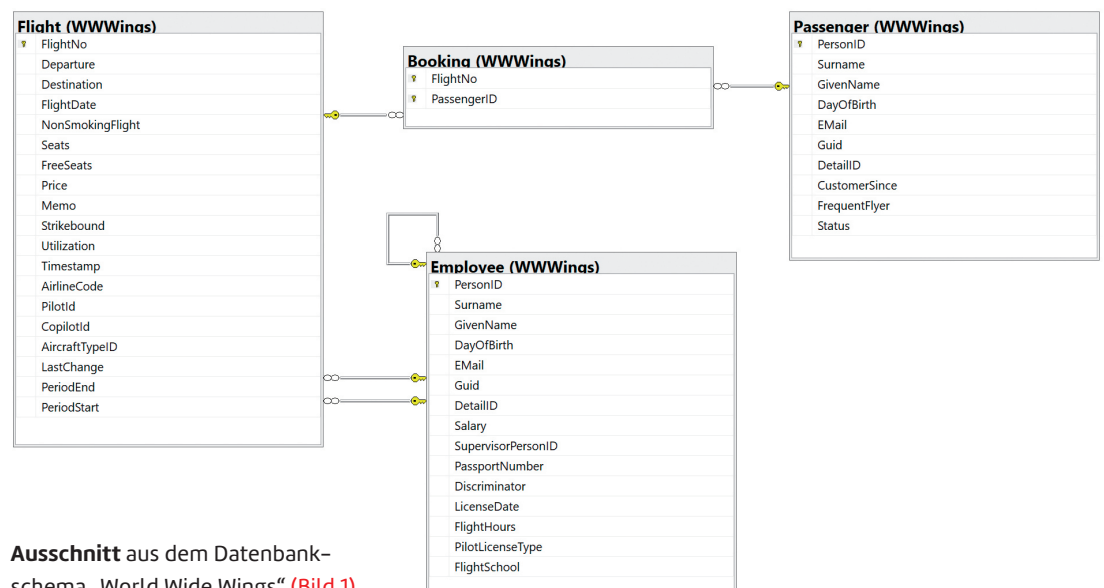
Entity Framework Core bietet einen bunten Strauß von Ladestrategien für verbundene Datensätze. Welche davon ist die schnellste?

Die Datenzugriffskolumne hat sich in der Vergangenheit natürlich bereits mit den verschiedenen Ladestrategien für verbundene Datensätze bei Entity Framework Core beschäftigt (siehe [1]). Und auch No-Tracking-Abfragen, bei denen die aus den Datensätzen materialisierten Objekte nicht im First-Level-Cache landen und nicht mit dem Change Tracker verbunden werden, waren schon Thema in dieser Kolumne (siehe [2]).

In dieser Ausgabe sollen No-Tracking-Abfragen mit den unterschiedlichen Ladestrategien verbunden werden. Seit Entity Framework Core 8.0 funktioniert No-Tracking auch in Verbindung mit dem automatischen Nachladen bei Bedarf („Lazy Loading“).

Das Szenario

Das Beispielszenario stammt wieder einmal von der fiktiven Fluggesellschaft „World Wide Wings“. Bild 1 zeigt den hier relevanten Ausschnitt aus dem Datenbankschema: Ein Datensatz in der Tabelle *Flight* besitzt eine N:M-Beziehung zur Tabelle *Passenger*, die durch die Zwischentabelle *Booking* im relationalen Datenbankschema ausgedrückt wird. Die Piloten stecken zusammen mit anderen Angestellten der Flugge-



Ausschnitt aus dem Datenbankschema „World Wide Wings“ (Bild 1)

sellschaft in der Tabelle *Employee* (per Diskriminator-Spalte werden hier verschiedene Objekttypen unterschieden). Zwischen den Tabellen *Employee* und *Flight* gibt es zwei 1:N-Beziehungen: eine für den Piloten, einen für den Copiloten (entsprechend sieht man zwei Fremdschlüssel *PilotId* und *CopilotId* in der Tabelle *Flight*).

Im zugehörigen Klassendiagramm (Bild 2) sieht man keine Repräsentation der Tabelle *Booking*: Entity Framework Core kann diese wegabstrahieren und besitzt stattdessen zwei Listen (*List<T>*) zwischen *Flight* und *Passenger*. *Passenger* und *Pilot* sind jeweils Teil einer Vererbungshierarchie mit der abstrakten Klasse *Person* an der Spitze. Der *PilotLicenseType* ist ein Enumerationstyp.



Die Aufgabe

Die hier für die Leistungsbetrachtung zu verwendende Programmieraufgabe lautet: Für einen Datenexport sollen Entwickler:innen eine bestimmte Teilmenge der Flüge (hier von zwei Abflugorten und mit vorhandenen Passagieren) zusammen mit den Namen des Piloten und des Copiloten sowie den Namen aller Passagiere exportieren. Das Szenario lässt sich auch auf einen Application Server übertragen, auf dem ein Webservice/Web-API eine Datenmenge serialisieren soll. Dafür gilt die gleiche Anforderung wie für den Datenexport.

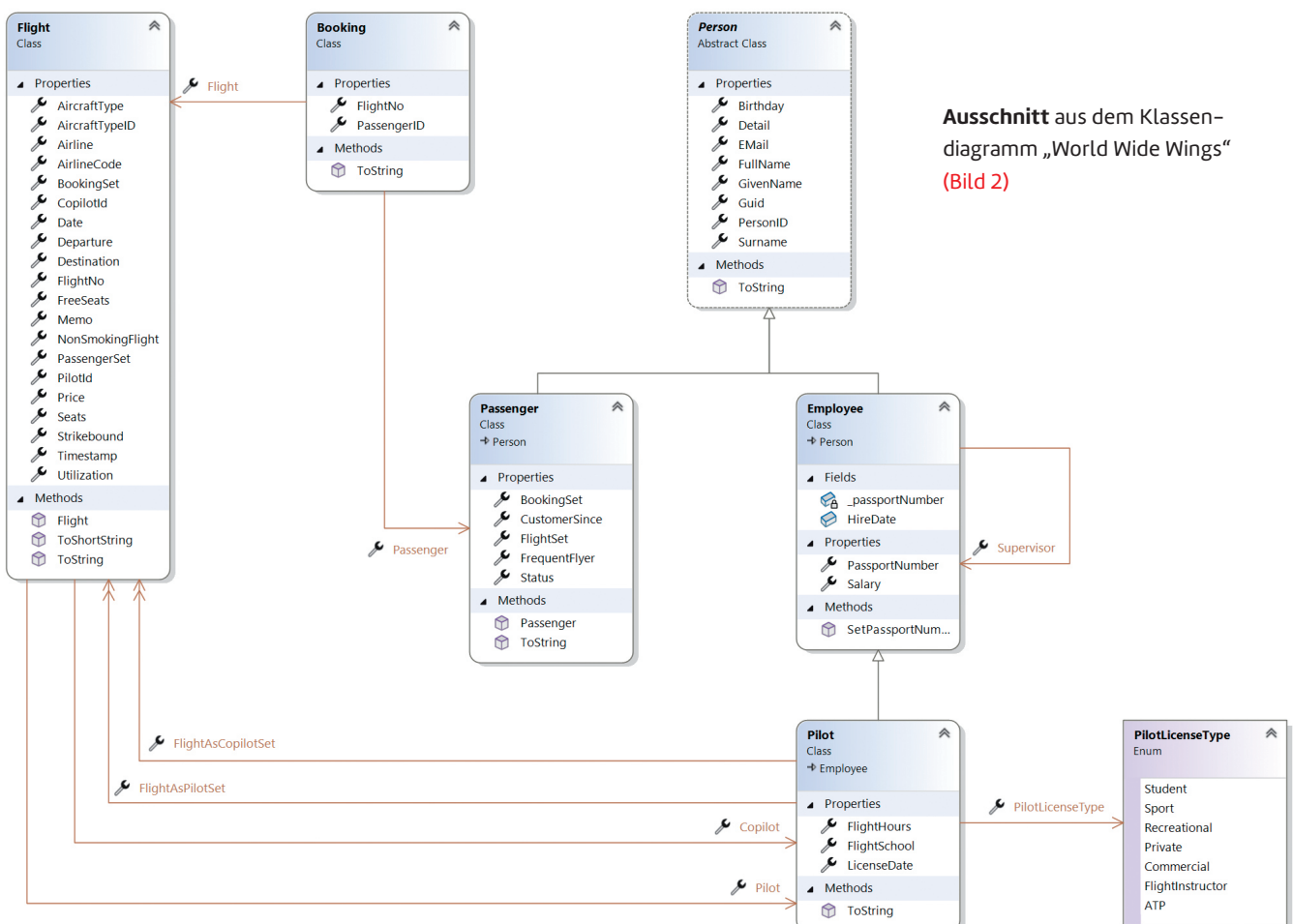
Die Ladestrategien

Lassen Sie uns kurz die Ladestrategien für verbundene Datensätze in Entity Framework Core rekapitulieren:

1. Entity Framework Core lädt im Standard (genau wie der Vorgänger ADO.NET Entity Framework) erst einmal nur

die Daten, die konkret per LINQ-Ausdruck oder SQL-Befehl festgelegt werden.

2. Entity Framework Core lädt (anders als der Vorgänger ADO.NET Entity Framework) im Standard keine verbundenen Datensätze nach, wenn in einem Entitätstyp ein Zugriff auf Navigationseigenschaften (1:1, 1:N, N:M) zu anderen Entitätstypen erfolgt. Dieses automatische Lazy Loading kann man aber in Entity Framework Core aktivieren: per *UseLazyLoadingProxies()* aus dem NuGet-Paket *Microsoft.EntityFrameworkCore.Proxies* oder *ILazyLoader* aus der Assembly *Microsoft.EntityFrameworkCore.Abstractions*.
3. Entwicklerinnen und Entwickler können verbundene Objekte jeweils explizit per Aufruf der Methode *Load()* nachladen, zum Beispiel mit *ctx.Entry(flug).Reference(x => x.Pilot).Load();*



● Listing 1: EF-Core-Ladestrategien in einem Aufzählungstyp für den automatischen Durchlauf durch alle Strategien

```
public enum LoadStrategy
{
    NoLoading, // Standard in EFCore !!!
    NoLoadingNoTracking,
    NoLoadingNoTrackingIdentity,
    LazyLoading, // Nachladen
    LazyLoadingNoTracking, // Nachladen
    LazyLoadingNoTrackingIdentity, // Nachladen
    LazyLoadingExplicit, // Explizites Nachladen
    LazyLoadingExplicitNoTracking, // Explizites
                                // Nachladen
    LazyLoadingExplicitNoTrackingIdentity, // Explizites
                                // Nachladen

    EagerLoading, // Server Join
    EagerLoadingNoTracking, // Server Join
    EagerLoadingNoTrackingIdentity, // Server Join
    EagerLoadingSplit, // Server Join
    EagerLoadingSplitNoTracking, // Server Join
    EagerLoadingSplitNoTrackingIdentity, // Server Join
    PreLoadingAll, // Client Join
    PreLoadingFilterFlight, // Client Join
    PreLoadingFilterFlightPassengers, // Client Join
    PreLoadingAllSplit, // Client Join
    PreLoadingFilterFlightSplit, // Client Join
    PreLoadingFilterFlightPassengerSplit // Client Join
}
```

4. Die grundsätzliche Alternative zum Nachladen (Lazy Loading) ist das Mitladen (Eager Loading): Durch *Include()* und *ThenInclude()* legt man in der LINQ-Abfrage bereits fest, welche verbundenen Datensätze mitgeladen werden sollen. In der Grundform entsteht immer ein einziger (oft großer) SQL-Befehl mit (oft vielen) Joins und einem großen kartesischen Produkt.
5. Seit Entity Framework Core Version 5.0 gibt es eine Variante des Eager Loading, bei dem man es dem OR-Mapper überlässt, den großen Befehl in mehrere zu zerlegen. Wenn die SQL-Befehle weniger Joins enthalten, wird das kartesische Produkt kleiner, das heißt, das Datenbankmanagementsystem sendet eine kleinere Datenmenge, und der Aufwand für Entity Framework Core, die Duplikate zu eliminieren, sinkt.
6. Alternativ kann man per Pre-Loading auch selbst die verbundenen Objekte laden: Entity Framework Core fügt diese per Relationship-Fixup im RAM mit bereits vorhandenen Objekten zusammen. Dies ist also ein Join im Client.
7. Beim Pre-Loading gibt es noch die Varianten, nur die benötigten Datensätze zu laden (dafür sind wieder Bedingungen über andere Tabellen, also Joins, notwendig) oder aus bestimmten Tabellen alle Datensätze zu holen (das lohnt sich, wenn es dort nur eine kleine bis mittlere Anzahl von Datensätzen gibt). Oft wird daher Preloading nicht für alle Entitäten verwendet, sondern nur für kleinere Mengen. Die übrigen Entitäten lädt man dann per Eager Loading. Pre-Loading ist ähnlich wie das Splitting beim Eager Loading, allerdings entscheiden die Entwickler:innen selbst, was getrennt geladen wird.

Verbindet man diese Varianten nun mit dem No-Tracking-Konzept, entstehen die Ladestrategien, die man in Listing 1 sieht. Nur Pre-Loading funktioniert ohne Tracking nicht, weil ohne Tracking kein Caching stattfindet und damit kein automatisches Relationship-Fixup im RAM möglich ist. Die Pre-Loading-Varianten lassen sich aber mit dem einfachen Eager Loading und dem Eager Loading mit Splitting kombinieren.

Die Rahmenbedingungen

Die Messungen der Geschwindigkeit der verschiedenen Ladestrategien erfolgt unter folgenden Rahmenbedingungen:

- Das Datenbankmanagementsystem ist ein virtualisierter Microsoft SQL Server 2019 im lokalen 1-Gigabit-Netzwerk.
- Die Datenbank von „World Wide Wings“ umfasst 100 000 Flüge, 500 Piloten, eine Million Passagiere und rund 6,1 Millionen Buchungen (Datenbankdateigröße rund ein Gigabyte).
- Der Client ist eine Konsolenanwendung mit .NET 8.0.8 und Entity Framework Core 8.0.8 auf Windows.
- Die Anzahl der Flüge im Resultset wurde mit *Take(100)* in LINQ (also *SELECT TOP 100...* in SQL) auf 100 begrenzt. Diese Begrenzung kann im Code leicht angepasst werden.
- Die Ausführungsdauer von Bildschirmausgaben wird nicht mitgemessen.
- Der Leistungstest findet nicht im Debugger statt.

Der Code für die Performance-Messungen

Listing 2 zeigt den Code für die Leistungstests. Kern ist die Methode *LoadWithStrategy()*, die die den Entwickler:innen gestellte Aufgabe mit jeweils einer der in Listing 1 definierten Strategien umsetzt. Dazu sind zahlreiche Fallunterscheidungen im Programmcode notwendig.

In der Praxis würde man in einer Datenzugriffsschichtmethode nicht alle diese Strategien implementieren, sondern in der Regel eine (für den jeweiligen Fall optimale) oder wenige davon (wenn man die Methode in verschiedenen Szenarien einsetzen will).

Den No-Tracking-Modus schaltet *LoadWithStrategy()* anstelle des Aufrufs von *AsNoTracking()* global ein mit:

```
ctx.ChangeTracker.QueryTrackingBehavior =
    QueryTrackingBehavior.NoTracking
```

Während der No-Tracking-Modus im Standard bei einem von der Datenbank gelieferten kartesischen Produkt stumpf jeweils ein eigenes Objekt im RAM anlegt (was zu mehr ►

● Listing 2: Programm für die Leistungsmessung der Ladestrategien (Teil 1)

```

using System;
using System.Diagnostics;
using System.Linq;
using B0;
using DA;
using Microsoft.EntityFrameworkCore;

namespace EFC_Console;

class EFCoreLoadingStrategyPerformanceTests
{
    // Konfiguration
    int limit = 100; // Maximalanzahl der Flüge, die
                    // geladen werden sollen
    int runs = 101; // Anzahl der Durchläufe beim
                    // Leistungstest
    string city1 = "Berlin";
    string city2 = "Rom";
    // Globale Variablen
    int sqlCommandCounter = 0; // Anzahl der ausgeführten
                              // SQL-Befehle

    /// <summary>
    /// In EFCore verfügbare Ladestrategien
    /// </summary>
    public enum LoadStrategy
    {
        NoLoading, // Standard in EFCore !!!
        NoLoadingNoTracking,
        NoLoadingNoTrackingIdentity,
        LazyLoading, // Nachladen
        LazyLoadingNoTracking, // Nachladen
        LazyLoadingNoTrackingIdentity, // Nachladen
        LazyLoadingExplicit, // Explizites Nachladen
        LazyLoadingExplicitNoTracking, // Explizites
                                   // Nachladen
        LazyLoadingExplicitNoTrackingIdentity, // Explizites
                                   // Nachladen

        EagerLoading, // Server Join
        EagerLoadingNoTracking, // Server Join
        EagerLoadingNoTrackingIdentity, // Server Join
        EagerLoadingSplit, // Server Join
        EagerLoadingSplitNoTracking, // Server Join
        EagerLoadingSplitNoTrackingIdentity, // Server Join
        PreLoadingAll, // Client Join
        PreLoadingFilterFlight, // Client Join
        PreLoadingFilterFlightPassengers, // Client Join
        PreLoadingAllSplit, // Client Join
        PreLoadingFilterFlightSplit, // Client Join
        PreLoadingFilterFlightPassengerSplit // Client Join
    }

    /// <summary>
    /// Schleife über alle Strategien mit Zeitmessung
    /// </summary>
    public void LoadingStrategiesPerformance()
    {
        CUI.Demo();
        //LoadWithStrategy(LoadStrategy.
            PreLoadingFilterFlightPassengerSplit);
        //return;

        for (int i = 1; i <= runs; i++)
        {
            // Schleife über alle Ladestrategien
            foreach (LoadStrategy ls in Enum.GetValues(
                typeof(LoadStrategy)))
            {
                Console.Title = ls + ": " + i + "/" + runs +
                    " " + ls;
                Console.Write("-> " + ls + "...\\r");
                ITVisions.Timer.Run($"[{ls}], () =>
                    LoadWithStrategy(ls), true, true,
                    "SQL-Befehle");
            }
        }
        CUI.Line("Ergebnisse");
        ITVisions.Timer.PrintResults();
    }

    /// <summary>
    /// Lädt die Daten mit einer bestimmten Strategie
    /// und gibt sie bei aktiviertem Debugger auch an
    /// der Konsole aus
    /// </summary>
    long LoadWithStrategy(LoadStrategy LoadStrategy)
    {
        if (System.Diagnostics.Debugger.IsAttached)
            CUI.Demo($"{nameof(LoadWithStrategy)} -->
                Strategie: {LoadStrategy}");

        sqlCommandCounter = 0;

        if (LoadStrategy.ToString().Contains("LazyLoading"))
            if (System.Diagnostics.Debugger.IsAttached)
                CUI.Warning("ACHTUNG: Lazy Loading ist aktiv!");
            else if (System.Diagnostics.Debugger.IsAttached)
                CUI.Success("Lazy Loading ist nicht aktiv!");

        using var ctx =
            new WWWingsContext(enableLazyLoading:
                LoadStrategy.ToString().Contains("LazyLoading"),
                logTo: LogCommands);

        #region NoTracking
        if (LoadStrategy.ToString().ContainsInsensitive(

```

● Listing 2: Programm für die Leistungsmessung der Ladestrategien (Teil 2)

```

        "NoTracking"))
    {
        if (LoadStrategy.ToString().ContainsInsensitive(
            "Identity"))
        {
            ctx.ChangeTracker.QueryTrackingBehavior =
                QueryTrackingBehavior.
                NoTrackingWithIdentityResolution;
        }
        else
        {
            ctx.ChangeTracker.QueryTrackingBehavior =
                QueryTrackingBehavior.NoTracking;
        }
    }
#endregion

#region Preloading
if (LoadStrategy.ToString().ContainsInsensitive(
    "PreLoading"))
{
    if (LoadStrategy.ToString().ContainsInsensitive(
        "All"))
    {
        ctx.PilotSet.TagWith("Preloading all Pilots").
            ToList(); // für Piloten und Copiloten :- )
        // Preloading ergibt aufgrund der Menge keinen
        // Sinn für Buchungen und Passagiere!
        //ctx.PassengerSet.TagWith("Preloading all
            Passengers").ToList();
        //ctx.BookingSet.TagWith(
            "Preloading all Bookings").ToList();
    }
    else if (LoadStrategy.ToString().
        ContainsInsensitive("FilterFlightPassengers"))
    {
        // Preloading ergibt aufgrund der Menge keinen
        // Sinn für Buchungen und Passagiere!
        ctx.PilotSet.TagWith("Preloading some Pilots").
            Where(x => x.FlightAsPilotSet.Any(f => (
                f.Departure == city1 && f.Destination ==
                city2) && f.PassengerSet.Count > 0)).ToList();
        // für Piloten und Copiloten :- )
        // Preloading ergibt aufgrund der Menge keinen
        // Sinn für Buchungen und Passagiere!
        //ctx.PassengerSet.TagWith("Preloading some
            Passengers").Where(x => x.BookingSet.Any(
                b => (b.Flight.Departure == city1 ||
                b.Flight.Destination == city2) &&
                b.Flight.PassengerSet.Count > 0)).ToList();
        //ctx.BookingSet.TagWith("Preloading some
            Bookings").Where(b => (b.Flight.Departure ==
            city1 && b.Flight.Destination == city2) &&
            b.Flight.PassengerSet.Count > 0)).ToList();
    }
    else if (LoadStrategy.ToString().
        ContainsInsensitive("FilterFlight"))
    {
        // Preloading ergibt aufgrund der Menge keinen
        // Sinn für Buchungen und Passagiere!
        ctx.PilotSet.TagWith("Preloading some Pilots").
            Where(x => x.FlightAsPilotSet.Any(f =>
                (f.Departure == city1 && f.Destination ==
                city2))).ToList();
        // für Piloten und Copiloten :- )
        // Preloading ergibt aufgrund der Menge keinen
        // Sinn für Buchungen und Passagiere!
        //ctx.PassengerSet.TagWith("Preloading some
            Passengers").Where(x => x.BookingSet.Any(
                b => (b.Flight.Departure == city1 &&

```

RAM-Bedarf führt und zu Änderungsanomalien aus Benutzersicht führen kann), macht sich Entity Framework Core im Modus *NoTrackingWithIdentityResolution* die Mühe, Duplikate zu erkennen (siehe auch [3]).

LoadingStrategiesPerformance() ruft *LoadWithStrategy()* nacheinander mit den verschiedenen Strategien auf und misst die Dauer. Damit die Ergebnisse aussagekräftig sind, wird jede Strategie 101-mal aufgerufen, wobei der erste Durchlauf als „Warm-up“ nicht mitgemessen wird.

In Listing 2 sieht man, dass das Preloading nur für die Piloten und Copiloten verwendet wird, denn dort ist die Menge mit 500 Datensätzen klein. Ein Preloading für die 6,1 Millionen Buchungen und eine Million Passagiere wäre nicht sinnvoll, denn hierbei würden auch viele Buchungen beziehungsweise die zugehörigen Passagiere geladen werden, die nicht zu den Flügen in der Ergebnismenge gehören.

Die Ergebnisse

Bild 3 und Bild 4 zeigen die Ergebnisse der Messungen, einmal für 100 Flüge mit zugehörigen Piloten und Passagieren und einmal für 1000 Flüge mit zugehörigen Piloten und Passagieren. Man sieht zu jeder gemessenen Ladestrategie in grüner Schrift die Gesamtdauer über alle Durchläufe (in Millisekunden) und die Durchschnittsdauer einer Ausführung (in Millisekunden) sowie am Ende in gelber Schrift die Anzahl der zum Datenbankmanagementsystem gesendeten SQL-Befehle.

Bevor es hier zu Missverständnissen kommt: Die ersten drei sehr schnellen Einträge in den Tabellen in Bild 3 und Bild 4 mit *NoLoading* im Namen haben die gesetzte Aufgabe nicht erfüllt: Sie laden jeweils nur die Flüge und ermöglichen nicht die Ausgabe von Pilotennamen und Passagierennamen.

Die schnellsten Ergebnisse mit Erfüllung der Aufgabe bei 100 Flügen (siehe Bild 3) liefert das Pre-Loading der Piloten ►


```

        b.Flight.Destination == city2) ).ToList();
//ctx.BookingSet.TagWith("Preloading some
Bookings").Where(b => (b.Flight.Departure ==
city1 && b.Flight.Destination ==
city2)).ToList();
    }
}
#endregion

// Abfrage für Flüge
IQueryable<Flight> fluege = (from f in
    ctx.FlightSet.TagWith("Loading Flights")
    where f.Departure == city1 && f.Destination ==
    city2 && f.PassengerSet.Count > 0
    select f).Take(limit);

#region Eager Loading
if (LoadStrategy.ToString().ContainsInsensitive(
    "EagerLoading"))
{
    if (LoadStrategy.ToString().ContainsInsensitive(
        "Split"))
    {
        fluege = fluege.AsSplitQuery().Include(x =>
            x.Pilot).Include(x => x.Copilot).Include(y =>
            y.BookingSet).ThenInclude(z => z.Passenger);
    }
    else
    {
        fluege = fluege.Include(x => x.Pilot).Include(
            x => x.Copilot).Include(y => y.BookingSet).
            ThenInclude(z => z.Passenger);
    }
}
}
#endregion

#region Preloading: Für Buchungen und Passagiere
ergibt Preloading aufgrund der Menge keinen Sinn
--> Daher Eager Loading für Buchungen und
Passagiere!
if (LoadStrategy.ToString().ContainsInsensitive(
    "PreLoading"))
{
    if (LoadStrategy.ToString().ContainsInsensitive(
        "Split"))
    {
        fluege = fluege.AsSplitQuery().Include(y =>
            y.BookingSet).ThenInclude(z => z.Passenger);
    }
    else
    {
        fluege = fluege.Include(y => y.BookingSet).
            ThenInclude(z => z.Passenger);
    }
}
}
#endregion

// Abfrage ausführen und Materialisieren
var fluegeList = fluege.ToList();

// Schleife über die Flüge zur Ausgabe von Flug-,
// Pilot- und Passagierdaten
foreach (var f in fluegeList)
{
    if (System.Diagnostics.Debugger.IsAttached)
        Console.WriteLine("Flug: " + f.FlightNo +
            " von " + f.Departure +
            " nach " + f.Destination + " hat " +
            f.FreeSeats + " freie Plätze");
}

#region --- Pilot ausgeben

```

Ergebnisse			
100x NoLoadingNoTracking	429 ms (Summe)	4 ms (Durchschnitt)	SQL-Befehle = 1
100x NoLoadingNoTrackingIdentity	504 ms (Summe)	5 ms (Durchschnitt)	SQL-Befehle = 1
100x NoLoading	611 ms (Summe)	6 ms (Durchschnitt)	SQL-Befehle = 1
100x PreLoadingFilterFlight	20.093 ms (Summe)	200 ms (Durchschnitt)	SQL-Befehle = 2
100x PreLoadingFilterFlightPassengers	20.189 ms (Summe)	201 ms (Durchschnitt)	SQL-Befehle = 2
100x PreLoadingAll	20.401 ms (Summe)	204 ms (Durchschnitt)	SQL-Befehle = 2
100x EagerLoadingNoTracking	26.319 ms (Summe)	263 ms (Durchschnitt)	SQL-Befehle = 1
100x EagerLoading	27.953 ms (Summe)	279 ms (Durchschnitt)	SQL-Befehle = 1
100x EagerLoadingNoTrackingIdentity	28.122 ms (Summe)	281 ms (Durchschnitt)	SQL-Befehle = 1
100x EagerLoadingSplitNoTracking	44.110 ms (Summe)	441 ms (Durchschnitt)	SQL-Befehle = 2
100x PreLoadingFilterFlightPassengerSplit	54.040 ms (Summe)	540 ms (Durchschnitt)	SQL-Befehle = 3
100x EagerLoadingSplitNoTrackingIdentity	55.579 ms (Summe)	555 ms (Durchschnitt)	SQL-Befehle = 2
100x PreLoadingAllSplit	56.279 ms (Summe)	562 ms (Durchschnitt)	SQL-Befehle = 3
100x PreLoadingFilterFlightSplit	57.169 ms (Summe)	571 ms (Durchschnitt)	SQL-Befehle = 3
100x EagerLoadingSplit	60.614 ms (Summe)	606 ms (Durchschnitt)	SQL-Befehle = 2
100x LazyLoadingExplicitNoTrackingIdentity	636.380 ms (Summe)	6.363 ms (Durchschnitt)	SQL-Befehle = 6500
100x LazyLoadingExplicitNoTracking	649.826 ms (Summe)	6.498 ms (Durchschnitt)	SQL-Befehle = 6500
100x LazyLoadingNoTracking	653.348 ms (Summe)	6.533 ms (Durchschnitt)	SQL-Befehle = 6500
100x LazyLoadingExplicit	750.858 ms (Summe)	7.508 ms (Durchschnitt)	SQL-Befehle = 6457
100x LazyLoading	755.710 ms (Summe)	7.557 ms (Durchschnitt)	SQL-Befehle = 6457
100x LazyLoadingNoTrackingIdentity	757.201 ms (Summe)	7.572 ms (Durchschnitt)	SQL-Befehle = 6500

Messergebnisse mit 100 Flügen (Durchschnitt von 100 Durchläufen) (Bild 3)

● Listing 2: Programm für die Leistungsmessung der Ladestrategien (Teil 3)

```

        if (LoadStrategy.ToString().ContainsInsensitive(
            "Explicit")) ctx.Entry(f).Reference(x =>
                x.Pilot).Load();
        var pilotName = f.Pilot?.FullName;
        if (!LoadStrategy.ToString().ContainsInsensitive(
            "NoLoading")) Debug.Assert(f.Pilot != null);
        if (System.Diagnostics.Debugger.IsAttached)
            Console.WriteLine("Pilot: " + pilotName);
        #endregion

        #region --- Copilot ausgeben
        if (LoadStrategy.ToString().ContainsInsensitive(
            "Explicit")) ctx.Entry(f).Reference(x =>
                x.Copilot).Load();
        var copilotName = f.Copilot?.FullName;
        if (!LoadStrategy.ToString().ContainsInsensitive(
            "NoLoading")) Debug.Assert(f.Copilot != null);
        if (System.Diagnostics.Debugger.IsAttached)
            Console.WriteLine("Copilot: " + pilotName);
        #endregion

        #region --- Passagiere ausgeben
        if (LoadStrategy.ToString().ContainsInsensitive(
            "Explicit")) ctx.Entry(f).Collection(x =>
                x.BookingSet).Load();
        foreach (var b in f?.BookingSet)
        {
            if (LoadStrategy.ToString().ContainsInsensitive(
                "Explicit")) ctx.Entry(b).Reference(x =>
                    x.Passenger).Load();
            var passengerName = b.Passenger?.FullName;
            if (System.Diagnostics.Debugger.IsAttached)
                Console.WriteLine(" Passenger: " +
                    passengerName);
        }
        if (!LoadStrategy.ToString().ContainsInsensitive(
            "NoLoading")) Debug.Assert(f.BookingSet !=
                null);
        #endregion

        if (System.Diagnostics.Debugger.IsAttached)
            CUI.Success("Anzahl SQL-Befehle: " +
                sqlCommandCounter);
        return sqlCommandCounter;
    }

    // Zählt die SQL-Befehle und gibt sie bei
    // aktiviertem Debugger auch an der Konsole aus
    public void LogCommands(string s)
    {
        sqlCommandCounter++;
        if (System.Diagnostics.Debugger.IsAttached)
            CUI.PrintAlternatingColor(s);
    }
}

```

(~200 Millisekunden für insgesamt zwei Abfragen). Das gefilterte Pre-Loading (173 Piloten) ist dabei marginal besser als das Laden aller Piloten (500 Piloten), da das Resultset etwas kleiner ist. Falls es mehrere verschiedene Abfragen über *Flight* in der gleichen Kontextklasse gibt, sollte man definitiv besser alle Piloten auf Vorrat halten, statt hier mit Filtern falsch zu optimieren.

Nach dem Pre-Loading gewinnt das Eager Loading die nächsten Plätze in **Bild 3**. Eager Loading ohne Tracking liegt knapp vor dem Eager Loading mit Tracking und dem Eager Loading ohne Tracking, aber mit Identity Resolution (wobei der Zeitunterschied zwischen letzteren beiden marginal ist). Alle diese Varianten kommen mit einer einzigen SQL-Abfrage aus, aber erzeugen ein großes kartesisches Produkt, dessen Auswertung für den objektrelationalen Mapper entsprechend aufwendig ist.

Dahinter kommen dann alle Ladestrategien, die sich auf *AsSplitQuery()* stützen. Beim Pre-Loading mit Splitting sind drei SQL-Abfragen notwendig, beim reinen Automatismus nur zwei Abfragen.

Sehr weit abgeschlagen landen alle Varianten des Nachladens bei Bedarf (Lazy Loading), denn hier sendet der Client schon bei nur 100 Flügen sagenhafte 6457 beziehungsweise

6500 SQL-*SELECT*-Abfragen zum Datenbankmanagementsystem, deren Resultsets dann zwar jeweils sehr klein (oft nur ein Datensatz) sind, aber einzeln materialisiert werden müssen. Beim Lazy Loading ohne Tracking verschlimmert sich das Problem noch, weil einige Passagiere, die auf mehreren Flügen gebucht sind, doppelt geladen werden müssen, da Entity Framework Core ohne Tracking ja eben nicht erkennen kann, dass diese im RAM schon vorhanden sind (daher hier jeweils 6500 SQL-Abfragen). Mit aktivem Tracking ist die Anzahl der Abfragen daher mit 6457 ein klein wenig niedriger.

Bei der Variante mit 1000 Flügen in **Bild 4** obsiegt hingegen das Pre-Loading mit dem Vorab-Laden aller Piloten inklusive Eager Loading mit Splitting beim Laden der Buchungen und Passagiere (825 ms im Durchschnitt). Danach kommt das Eager Loading mit Splitting und No-Tracking (864 ms); dann folgen die beiden Varianten mit Filtern beim Pre-Loading (873 beziehungsweise 937 ms).

Deutlich abgeschlagen mit rund zwei bis fast drei Sekunden sind hier

- die Pre-Loading-Varianten ohne Splitting,
- das Eager Loading mit Splitting und mit Tracking,
- das Eager Loading mit Splitting, ohne Tracking und mit Identity Resolution,

- das Eager Loading ohne Splitting und mit No-Tracking,
- das Eager Loading ohne Splitting und mit Tracking und
- das Eager Loading ohne Splitting, ohne Tracking und mit Identity Resolution.

Bei der Messvariante mit 1000 Flügen in **Bild 4** wurde auf das langwierige Ergebnis mit Lazy Loading verzichtet.

übrigens auch beim Eager Loading nicht immer aktiv, sondern eine Option, die man als Entwickler:in wählen muss.

Eine allgemeingültige Faustregel für die beste Ladestrategie gibt es nicht von Microsoft, und sie lässt sich auch aus der jahrelangen Praxiserfahrung des Autors mit Entity Framework Core in sehr vielen .NET-Projekten beim besten Willen nicht ableiten. Das heißt also: Wer seinen Datenzugriffscode richtig

Ergebnisse				
100x NoLoadingNoTracking	3.023 ms (Summe)	30 ms (Durchschnitt)	SQL-Befehle = 1	
100x NoLoadingNoTrackingIdentity	3.147 ms (Summe)	31 ms (Durchschnitt)	SQL-Befehle = 1	
100x NoLoading	3.744 ms (Summe)	37 ms (Durchschnitt)	SQL-Befehle = 1	
100x PreLoadingAllSplit	82.578 ms (Summe)	825 ms (Durchschnitt)	SQL-Befehle = 3	
100x EagerLoadingSplitNoTracking	86.494 ms (Summe)	864 ms (Durchschnitt)	SQL-Befehle = 2	
100x PreLoadingFilterFlightSplit	87.382 ms (Summe)	873 ms (Durchschnitt)	SQL-Befehle = 3	
100x PreLoadingFilterFlightPassengerSplit	93.701 ms (Summe)	937 ms (Durchschnitt)	SQL-Befehle = 3	
100x PreLoadingAll	196.870 ms (Summe)	1.968 ms (Durchschnitt)	SQL-Befehle = 2	
100x PreLoadingFilterFlight	199.914 ms (Summe)	1.999 ms (Durchschnitt)	SQL-Befehle = 2	
100x EagerLoadingSplit	202.321 ms (Summe)	2.023 ms (Durchschnitt)	SQL-Befehle = 2	
100x EagerLoadingSplitNoTrackingIdentity	207.244 ms (Summe)	2.072 ms (Durchschnitt)	SQL-Befehle = 2	
100x PreLoadingFilterFlightPassengers	208.812 ms (Summe)	2.088 ms (Durchschnitt)	SQL-Befehle = 2	
100x EagerLoadingNoTracking	249.093 ms (Summe)	2.490 ms (Durchschnitt)	SQL-Befehle = 1	
100x EagerLoading	279.698 ms (Summe)	2.796 ms (Durchschnitt)	SQL-Befehle = 1	
100x EagerLoadingNoTrackingIdentity	282.819 ms (Summe)	2.828 ms (Durchschnitt)	SQL-Befehle = 1	

Messergebnisse mit 1000 Flügen (Durchschnitt von 100 Durchläufen). Auf die Messungen mit Lazy Loading wurde verzichtet, da diese in Summe Stunden gedauert hätten und damit diese Ladestrategie für eine solche Datenmenge sowieso keinerlei praktische Relevanz hat (**Bild 4**)

Fazit

Sind diese Ergebnisse allgemeingültig? Teils ja, teils nein.

Lazy Loading ist allgemein keine gute Lösung für ein Szenario, bei dem alle Datensätze innerhalb kurzer Zeit gebraucht werden, also für Datenexporte oder Services in verteilten Systemen, welche die Objekte serialisieren und dann weitergeben wollen. Lazy Loading ergibt allenfalls Sinn in 2-Tier-Anwendungen (also ohne Middle Tier) mit Master-Detail-Darstellungen, wenn man es sich als Entwickler:in einfach machen will, indem verbundene Datensätze automatisch nachgeladen werden, sobald der Benutzer auf einen Datensatz klickt. In diesem Fall würde sich das Absenden weiterer SQL-Befehle nicht drastisch auf die Leistung auswirken, weil es immer nur wenige Befehle bei jedem Klick wären, die der Benutzer in vielen Fällen kaum merkt.

Den No-Tracking-Modus zu verwenden ist hingegen immer eine gute Idee. Das wussten wir schon vorher. Die Identity Resolution kostet immer Zeit. Aber es geht besser als No-Tracking, wenn nicht nur Daten aus einer Datenbanktabelle, sondern aus mehreren Datenbanktabellen geladen werden sollen: In jeder Messung hier wurde No-Tracking von einer Pre-Loading-Lösung geschlagen!

Darüber hinaus kann man leider nicht allgemeingültig sagen kann, ob Pre-Loading oder Eager Loading besser ist. Hier hat zwar zweimal Pre-Loading gewonnen, aber die Praxis zeigt, dass das nicht immer gilt.

Auch zeigt der Vergleich von **Bild 3** und **Bild 4**, dass die Aktivierung des automatischen Splitting beim Eager Loading sich lohnen kann (ganz vorne im Szenario mit 1000 Flügen), aber auch nicht lohnen kann (deutlich langsamer als der Sieger im Szenario mit 100 Flügen). Daher ist das automatische Splitting

optimieren will, muss also in jedem konkreten Fall immer selbst messen, ob Eager Loading, Eager Loading mit Splitting oder Pre-Loading (mit/ohne Splitting) schneller ist. ■

- [1] Holger Schwichtenberg, *Vorausschauend, Ladestrategien für Objektbeziehungen bei Entity Framework Core*, dotnetpro 3/2017, Seite 86 ff., www.dotnetpro.de/A1703DataAccess
- [2] Holger Schwichtenberg, *Datenzugriffs-Turbo, Leistungssteigerungen bei Entity Framework Core im No-Tracking-Modus*, dotnetpro 6/2017, Seite 98 ff., www.dotnetpro.de/A1706DataAccess
- [3] Holger Schwichtenberg, *Einmaligkeit, Identity Resolution in Entity Framework Core*, dotnetpro 11/2020, Seite 90 ff., www.dotnetpro.de/A2011DataAccess
- [4] Holger Schwichtenberg, *Schneller, als die Datenbank erlaubt, Client-Joins in Entity Framework Core*, dotnetpro 10/2020, Seite 100 ff., www.dotnetpro.de/A2010DataAccess

Dr. Holger Schwichtenberg

alias der „Dotnet-Doktor“ ist Leiter des Expertennetzwerks www.IT-Visions.de, das mit 53 renommierten Experten mittlere und große Unternehmen durch Beratungen und Schulungen sowie bei der Softwareentwicklung unterstützt. Durch seine Fachbücher und Vorträge auf Fachkonferenzen gehört er zu den bekanntesten Experten für .NET und Webtechniken in Deutschland.

www.dotnet-doktor.de

dnpCode A2502DataAccess

DEVSECOPS MIT AZURE DEVOPS

Sicherheit in jeder Zeile Code

Wie Sie mit GitHub Advanced Security for Azure DevOps neu geschriebenen Code proaktiv und automatisiert auf Schwachstellen überprüfen und diese effizient beseitigen.

Sicherheit und Künstliche Intelligenz sind aktuelle Kernthemen der heutigen Softwareentwicklung. Darum verwundert es niemanden, dass GitHub sich genau diese zwei Themen auf die Fahne geschrieben hat und die eigenen Produkte darauf ausrichtet. Natürlich kommen auch die anderen DevOps-Themen wie Code-Verwaltung, CI/CD-Automatisierung, Arbeitsplanung und Artefakte nicht zu kurz, doch ist dies mittlerweile die Basis, die jede DevOps-Umgebung heutzutage aus dem Effeff beherrschen sollte.

Doch warum sprechen wir zum Beginn des Artikels über GitHub, obwohl es um DevSecOps mit Azure DevOps geht? Die Antwort ist ganz einfach und zeigt auch die neue Strategie von Microsoft: Mit dem Kauf und der Weiterentwicklung von GitHub hat Microsoft zusammen mit Azure DevOps zwei mehr oder weniger konkurrierende DevOps-Plattformen im eigenen Portfolio. GitHub ist ganz klar das Zugpferd in Sachen Innovation, und mit dem Themenfokus Sicherheit und KI ist die DevSecOps-Funktionalität ebenfalls ganz klar bei GitHub angesiedelt. Azure DevOps dagegen ist bei vielen großen Unternehmen noch immer als die firmenweite DevOps-Umgebung gesetzt, und auch hier wird eine moderne, effiziente und automatisierte Sicherstellung der Sicherheit der zu entwickelnden Applikationen benötigt. Anstatt nun zwei Lösungen zu entwickeln, hat sich Microsoft dazu entschieden, die Advanced-Security-Funktionen von GitHub in Azure DevOps zu integrieren. Technisch laufen also in beiden Plattformen dieselben Security-Dienste, jedoch hat jede Plattform die eigene Integration in die Benutzeroberfläche sowie die Verwaltung von Alerts implementiert.

DevSecOps

Bevor wir die Funktionen von GitHub Advanced Security for Azure DevOps (kurz GHASDO) zusammen anschauen, sollten wir erst die Grundlagen klären, damit wir das Ganze richtig einordnen können. Mit GHASDO bekommen wir Funktionen und Services, um Sicherheitslücken in Codestellen, Abhängigkeiten oder gar versehentlich veröffentlichte Geheimnisse zu erkennen, zu eskalieren und deren Beseitigung zu überwachen. Kurzum: Wir erweitern unseren DevOps-Prozess zu einem DevSecOps-Prozess.

DevSecOps ist eine Erweiterung von DevOps, um Sicherheitspraktiken in den DevOps-Ansatz zu integrieren. Im Gegensatz zum traditionellen Modell mit einem zentralen Sicherheitsteam wird jedes Entwicklungsteam befähigt, die notwendigen Sicherheitskontrollen in den Entwicklungsprozess einzubinden. Diese Sicherheitspraktiken werden früh im

Entwicklungszyklus durchgeführt, was als „Shift Left“-Ansatz bezeichnet wird. Wie bei allem in einem optimalen DevOps-Prozess geht es also auch hier darum, möglichst frühzeitig Feedback an die Entwickler zu geben. Somit wird der frühere Roundtrip zu und vom Security-Team verkürzt oder gar weggelassen, und die Sicherheitslücken können effizient während der Entwicklung geschlossen werden.

Generell wird in einem DevSecOps-Prozess die Sicherheit in den folgenden drei Hauptbereichen getestet:

- **Statische Analyse:** Bei der statischen Analyse, auch als Static Application Security Testing (SAST) bekannt, wird der Sourcecode auf Fehler und Schwachstellen überprüft. Das Tool hat vollen Zugriff auf den Sourcecode.
- **Softwarezusammensetzung:** Mittels der Analyse der Zusammensetzung der Software, auch als Software Composition Analysis (SCA) bekannt, werden die Abhängigkeiten der Software auf Vulnerabilities überprüft. Gerade bei modernen Frameworks und Applikationen ist die Anzahl der Open-Source-Bibliotheken sehr groß und unübersichtlich, sodass es für Angreifer ein leichtes Spiel ist, über diesen Kanal etwas einzuschleusen. Umso wichtiger ist es, diese Abhängigkeiten automatisiert und permanent zu überprüfen.
- **Dynamische Analyse:** Im Unterschied zu den zwei vorherigen Bereichen wird hierbei die Software während der Laufzeit analysiert. Die Tools agieren als virtuelle Penetration-Tester und versuchen, die Software über bekannte Schwachstellen zu „hacken“. Bei dieser Form der Sicherheitsanalyse wird zwischen Dynamic Application Security Testing (DAST) und Interactive Application Security Testing (IAST) unterschieden. Bei der ersteren Analyseform wird die Applikation als Blackbox angesehen, und es wird lediglich von außen getestet. Die zweite Variante kombiniert den Blackbox- und Whitebox-Ansatz, hat Kenntnisse über das Innenleben und kann die Applikation daher während des Tests von innen überwachen (Hook Points, Debugger, Monitoring, Interceptors und anderes).

Programmierfehler sind nach wie vor der Attack-Vektor Nummer eins, wenn es darum geht, Systeme anzugreifen. Die obigen Mechanismen helfen dem Softwareentwicklungsteam, diese Fehler möglichst früh zu erkennen und zu beseitigen. Im Idealfall wird diese Überprüfung als zwingende Policy in den Pull-Request-Workflow integriert, sodass niemals ungetesteter Code integriert werden kann. Viele moderne Softwareumgebungen setzen stark auf das Einbinden von Dritthersteller- oder Open-Source-Bibliotheken. Diese wie-

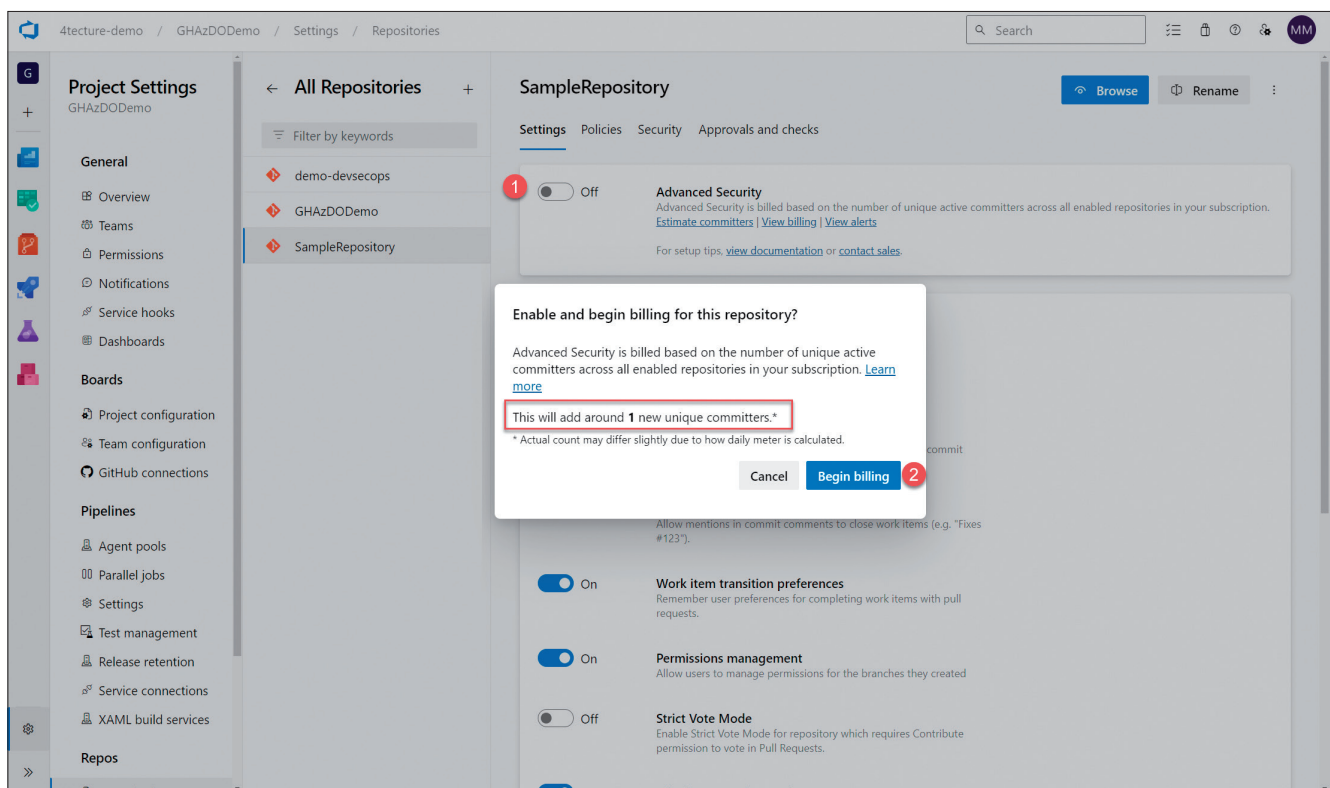
derum haben ihre eigenen Abhängigkeiten und so weiter. Kurzum, der Abhängigkeitsbaum einer moderat komplexen Applikation ist schon zu groß, als dass dieser manuell überprüft werden könnte. Zudem werden Schwachstellen in Abhängigkeiten oft erst Tage oder Monate nach der Veröffentlichung entdeckt. Somit ist eine automatische und periodische Überprüfung aller Abhängigkeiten Pflicht. Die technische Überprüfung an sich ist nicht allzu komplex, jedoch ist der Unterhalt einer stets aktualisierten Vulnerability-Datenbank das A und O für den Erfolg eines solchen Dienstes.

GitHub Advanced Security hat sich auf den statischen Teil der Sicherheitsüberprüfungen spezialisiert. Mittels Code Scanning wird der Sourcecode auf Schwachstellen gescannt,

GitHub Advanced Security for Azure DevOps

Die Nutzung von GitHub Advanced Security for Azure DevOps ist an zwei Bedingungen geknüpft: das Vorhandensein von Azure DevOps Services – es steht also nur Kunden der Cloud-Lösung zur Verfügung – sowie einer dedizierten Lizenz pro Benutzer.

Die Lizenzierung läuft vollautomatisiert über die Abrechnung von Azure DevOps über eine Azure Subscription ab. Der Lizenzpreis ist identisch mit den Preisen von GitHub für Advanced Security. Tatsächlich wird dasselbe Feature einfach mit der Integration in Azure DevOps oder GitHub bezahlt und genutzt. Die Aktivierung erfolgt über das Repository, und die Lizenzen werden pro aktivem Committer über die



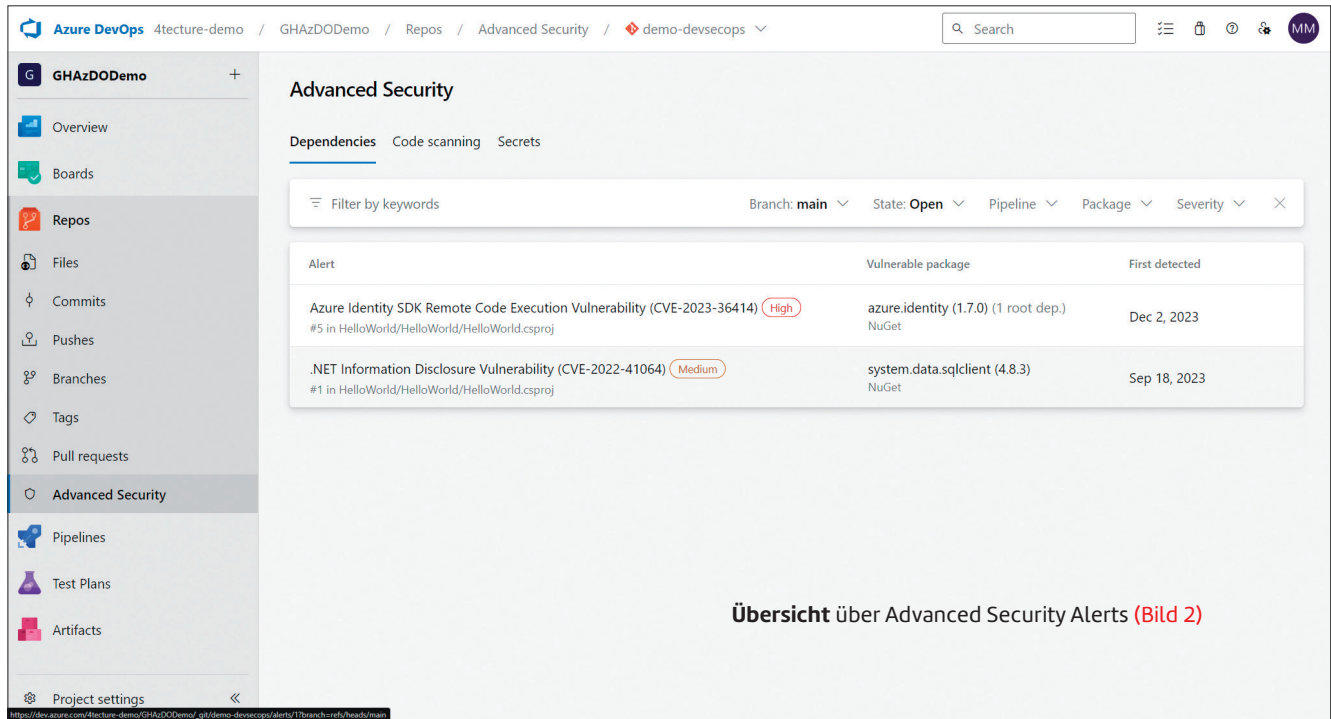
Aktivierung von Advanced Security auf einem Repository (Bild 1)

und mittels Dependency Scanning werden die Abhängigkeiten wie zum Beispiel NuGet-Pakete oder NPM-Pakete anhand einer Vulnerability-Datenbank überprüft. Mittels Secret Scanning kann der Code nach Passwörtern oder Schlüsseln durchsucht werden oder gar der Git-Push verweigert werden, sollten sich Secrets in den Code-Fragmenten befinden. Im Bereich der dynamischen Analyse ist derzeit keine Standardfunktionalität vorhanden, da diese Tools oft spezifisch auf die Applikation hin evaluiert und konfiguriert werden müssen. Es lässt sich jedoch problemlos mittels Pipelines ein DAST/IAST-Tool integrieren, und Sicherheitslücken können ebenso in das zentrale Alert-Management integriert werden.

Bevor wir nun in die Details der drei Bereiche eintauchen, müssen wir erst klären, wie GHASDO aktiviert und konfiguriert wird.

letzten 90 Tage berechnet. Natürlich wird innerhalb derselben Organisation die Lizenz pro Benutzer nur einmal verrechnet. Ist man also aktiver Committer in fünf Repositories in der gleichen Organisation, bezahlt man die Lizenz für einen Benutzer ein Mal. Der Lizenzpreis schlägt mit 49 US-Dollar pro Monat und aktiven Committer, verglichen mit der Basic-Lizenz von 6 US-Dollar pro Monat, eher etwas teuer zu Buche. Vergleicht man ihn aber mit anderen Anbietern in diesem Bereich, liegt er durchaus im üblichen Rahmen.

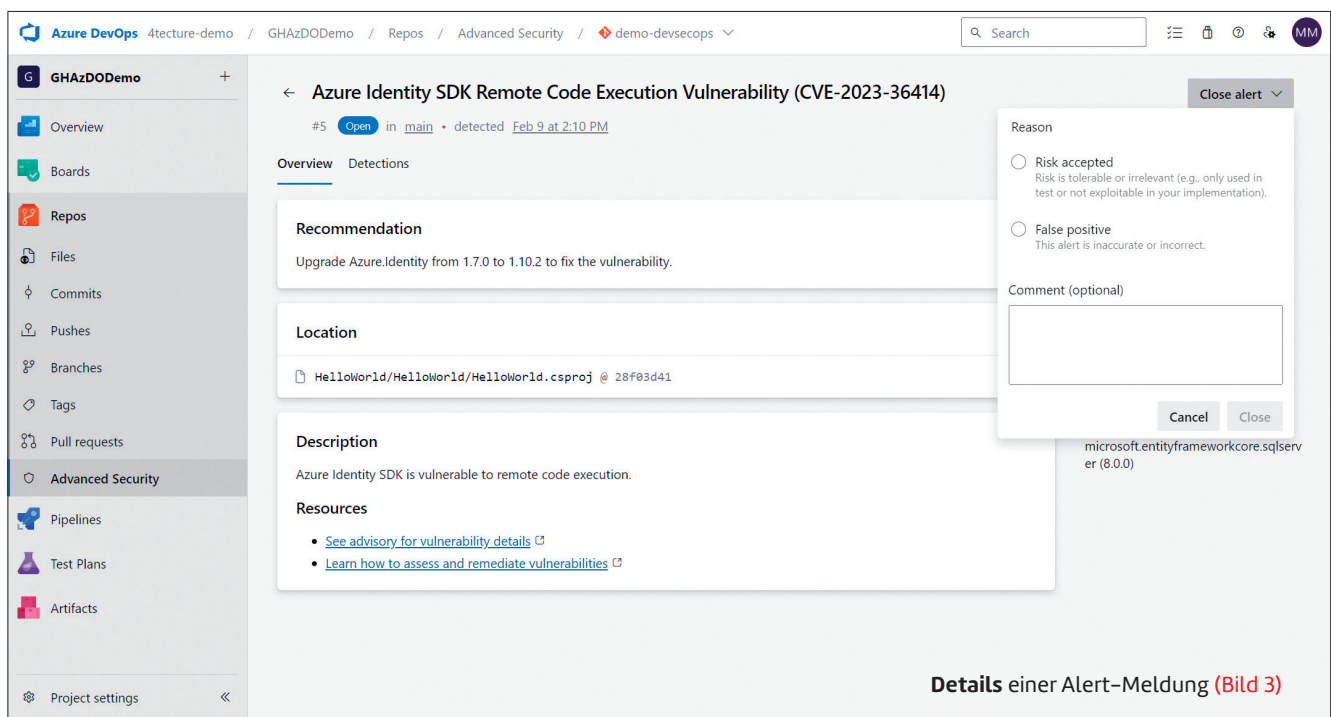
Advanced Security kann pro Repository respektive für alle Repositories in einem Projekt oder sogar in der gesamten Organisation aktiviert werden. Das Aktivieren des On/Off-Schalters wird durch einen Dialog unterbrochen, der die Anzahl zu zahlender Lizenzen berechnet und eine erneute Bestätigung erfordert. Danach sind die Lizenzen im Unter- ►



menü *Billing* in den Einstellungen ersichtlich. Bild 1 zeigt den Dialog bei der Aktivierung auf einem Repository.

Mit der Aktivierung von Advanced Security bekommen wir unter dem Menüeintrag *Repos* ein neues Untermenü *Advanced Security* (siehe Bild 2). Hier werden zentral alle Alerts dargestellt und können entsprechend gesichtet und verarbeitet werden. Die Funktionalität integriert sich in das Permission-System von Azure DevOps, und es lässt sich feingranular auf Gruppen- und Benutzerebene einstellen, wer Alerts sehen

oder diese auch manuell schließen darf. Die Alert-Übersicht zeigt die Alerts der jeweiligen Security Scanner: Dependency Scanning, Code Scanning und Secret Scanning. Für jeden Alert kann eine Detailansicht geöffnet werden, die Zusatzinformationen zur eigentlichen Vulnerability anzeigt sowie die Details, wo genau die Sicherheitslücke aufgetreten ist beziehungsweise welcher Code-Abschnitt dafür verantwortlich ist. Grundsätzlich werden diese Alerts automatisch geschlossen, wenn ein Folge-Build aufzeigt, dass die Abhängigkeit



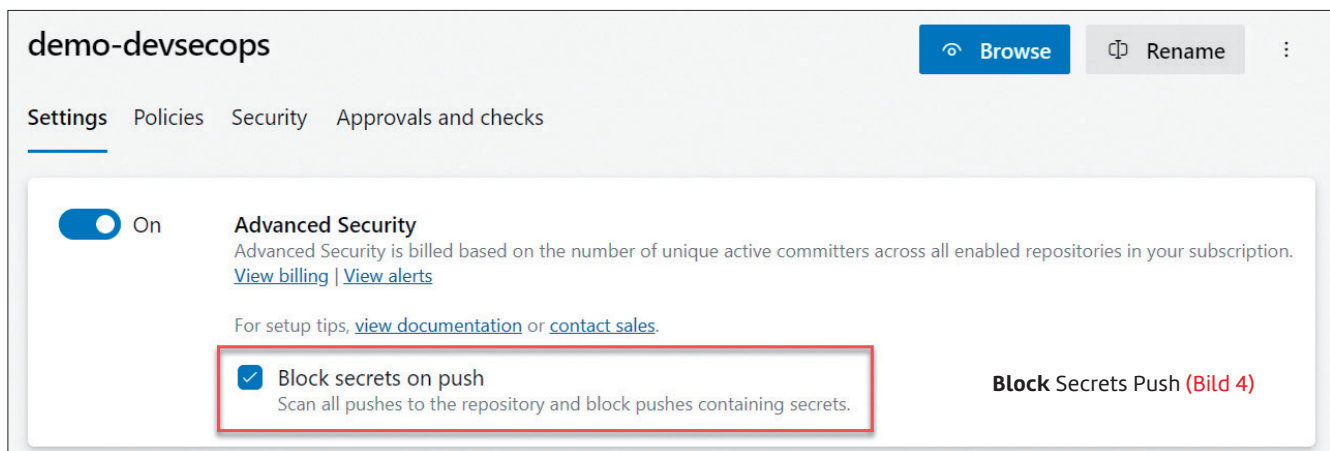
aktualisiert wurde respektive der Code korrigiert wurde. Für den Fall, dass der Alert nicht behandelt werden soll, kann dieser auch manuell geschlossen werden. Hierbei muss eine Begründung angegeben werden. Die Detailansicht sowie das manuelle Schließen des Alerts sind in **Bild 3** dargestellt.

Advanced Security verwendet das SARIF-Format, um die Sicherheitslücken zu melden. Da dies ein standardisiertes Format ist, können sich auch Drittanbieter-Tools integrieren und zusätzliche Alerts darstellen. Wichtig zu erwähnen ist hierbei noch, dass nur auf die Alerts zugegriffen werden kann, solange auch eine Lizenz vorhanden ist. Wird die Lizenzierung für Advanced Security beendet oder unterbrochen, erlischt auch der Zugriff auf die Alerts.

Nun sind wir fast so weit, um mit der Konfiguration von Advanced Security zu starten. Da das Code- und Dependency-Scanning auf Azure Pipelines basiert, müssen wir sicherstellen, dass unsere Agents dafür gerüstet sind. Im Fall von Hosted Agents muss nichts unternommen werden; diese haben

Unfreiwillig veröffentlichte Secrets, also Passwörter, Access Tokens und Keys, sind eine große Bedrohung, da sich ein potenzieller Angreifer durch sie mit echten Authentisierungsschlüsseln ausweisen kann. Schnell ist es passiert und ein Entwickler hat unbeabsichtigt, nach dem raschen Ausprobieren mit in den Sourcecode kopierten Secrets, den Commit gepusht, ohne diese zuvor wieder entfernt zu haben. Das Secret ist nun in der Git-Historie und auch über die Volltextsuche auffindbar. Hier kommt das Secret Scanning ins Spiel. Sollte ein Secret im Sourcecode gefunden werden, so wird direkt ein Alert erstellt, und der Entwickler bekommt eine detaillierte Anleitung, was zu tun ist und wie dies aus der Git-Historie zu entfernen ist.

Die Secrets in der Git-Historie zu entfernen ist mühsam, und auch der Revocation-Prozess ist mit Aufwand verbunden. Besser ist es also, wenn das Secret erst gar nicht erst den Entwicklerrechner verlässt. Dies wird mittels der Push Protection verhindert. Die Push Protection hängt sich serverseitig in den



bereits alle notwendigen Konfigurationen und Komponenten installiert. Im Fall von Self-Hosted Agents müssen folgende Punkte sichergestellt werden: Während der Ausführung der Scans muss der Agent auf die Vulnerability-Advisory-Datenbank zugreifen können. Hierzu sind einige Domains respektive IP-Adressen auf den Firewall-Einstellungen der Agents respektive des Agent-Netzwerks zu erlauben. Zusätzlich benötigt der Agent eine .NET-6.0.x-Runtime, und das CodeQL-Bundle muss installiert sein beziehungsweise installiert werden können. Haben die Agents keinen Zugriff auf die Download-Seiten respektive auf das Internet, so müssen die Agents zuvor mit den entsprechenden Runtimes und Tools ausgestattet werden. Detaillierte Informationen zum Setup der Agents stellt Microsoft unter [1] bereit.

Secret Scanning

Das Secret Scanning wird automatisch aktiviert, wenn auf einem Repository (respektive über die projekt- oder organisationsweiten Einstellungen) Advanced Security eingeschaltet ist. Die Push Protection lässt sich jedoch per Einstellung aktivieren oder deaktivieren (**Bild 4**).

Git-Push-Event und analysiert den Dateinhalt der gepushten Objekte. Wird ein Secret erkannt, so wird der Push nicht akzeptiert und der Vorgang abgebrochen. Daraus resultiert, dass die Secrets nicht serverseitig gespeichert werden und somit nicht für andere zugänglich sind. Das Problem kann also lokal gelöst werden und danach – ohne Secrets – nochmals gepusht werden. Die Push Protection funktioniert jedoch nur mit Secrets, bei denen mit einer hohen Wahrscheinlichkeit identifiziert wurde, dass es sich tatsächlich um ein Secret handelt. Das bedeutet: Die Push Protection findet weniger Secrets als das Scanning im Repository.

Das Secret Scanning unterscheidet bei den Secrets zwischen Provider Patterns und Non-Provider Patterns. Microsoft listet unter [2] die einzelnen Provider Patterns und Non-Provider Patterns detailliert auf, inklusive der Angabe, welche Patterns in der Push Protection unterstützt sind und welche als User Alerts gefunden werden können.

Was ist nun zu unternehmen, wenn ein Secret erkannt wurde? Im Fall eines User Alerts ist das Secret bereits auf dem Server gespeichert und somit für andere zugänglich. Hier muss erstens das Secret widerrufen werden. Je nach Secret ►

● Listing 1: Azure-Pipelines mit GHaZDO

```

# Trigger the pipeline on changes to the main branch
trigger:
- main

# Define the build environment
pool:
  vmImage: 'ubuntu-latest'

# Define variables used in the pipeline
variables:
  buildConfiguration: 'Release'
  workingDirectory:
    '$(Build.SourcesDirectory)/HelloWorld'
  # DependencyScanning.Timeout: 600 # default is 300,
  # anything under 300 has no effect
  # DependencyScanning.Skip: true # break-glass
  # scenario to skip dependency scanning from blocking
  # the build

# Pipeline steps
steps:
# Initialize Advanced Security CodeQL
- task: AdvancedSecurity-Codeql-Init@1
  inputs:
    languages: 'csharp'
    queriesuite: 'security-extended'
    displayName: 'Initialize CodeQL'

# Install .NET Core SDK
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '8.x'
    installationPath: '$(Agent.ToolsDirectory)/dotnet
    displayName: 'Install .NET Core SDK'

# Restore dependencies using dotnet command
- script: dotnet restore $(workingDirectory)/
  HelloWorld.sln
  displayName: 'Restore dependencies'
  workingDirectory: $(workingDirectory)

# Build the solution using dotnet command
- script: dotnet build $(workingDirectory)/
  HelloWorld.sln --configuration
  $(buildConfiguration) --no-restore
  displayName: 'Build solution'
  workingDirectory: $(workingDirectory)

# Run unit tests using dotnet command
- script: dotnet test $(workingDirectory)/
  HelloWorld.sln
  --configuration $(buildConfiguration) --no-build
  --logger trx
  displayName: 'Run unit tests'
  workingDirectory: $(workingDirectory)

# Publish the web application to the artifact staging
directory
- script: dotnet publish $(workingDirectory)/
  HelloWorld/HelloWorld.csproj
  --configuration$(buildConfiguration)
  --output $(Build.ArtifactStagingDirectory)
  displayName: 'Publish web app'
  workingDirectory: $(workingDirectory)

# Create an SBOM and add it to the pipeline artifact
- script: |

```

und Anbieter kann dies über ein API durchgeführt werden respektive das Passwort oder dergleichen sofort muss aktualisiert werden. Nachdem das kompromittierte Secret nicht mehr gültig ist, kann der entsprechende Alert geschlossen werden. Falls gewünscht und möglich, kann das Secret noch aus der serverseitigen Historie gelöscht werden. Bei Feature-Bran­ches ist dies relativ einfach; bei gemeinsam verwendeten Integrations-Bran­ches ist es dagegen aufwendiger, und der Nutzen dieser Aktion muss beurteilt werden.

Bei einem Secret, das durch die Push Protection blockiert wird, muss dies lokal korrigiert werden. Das Secret ist somit nur auf dem Entwicklerrechner vorhanden und nicht für andere ersichtlich. Somit müssen wir das Secret aus dem Sourcecode und der lokalen Git-Historie entfernen. Einfach einen neuen Commit mit dem Entfernen des Secrets zu erstellen würde nichts bringen, da das Secret dennoch in der Com­mit-Historie vorhanden wäre.

Wurde das Secret auf dem letzten Commit des Pushs hin­zugefügt, genügt es, mittels *git commit -amend* den letzten Commit zu ersetzen. Liegt das Einfügen des Secrets mehrere Commits zurück, so muss diese Historie umgeschrieben wer­den. Mittels *git rebase -i [commit ID before credential intro­duction]~1* kann ein interaktiver Rebase durchgeführt wer­den. Mittels *pick* und *edit* können die Commits editiert und das Secret entfernt werden. Der Commit mit dem Secret kann mittels *git commit --amend* ersetzt werden, und der Rebase kann dann mit *git rebase --continue* fortgesetzt werden.

Nun gibt es noch die letzte Kategorie, bei denen ein Alert eintreten kann: False-Positives oder Beispielwerte in den Kon­figurationsdateien. Bei Beispielwerten oder Platzhaltern soll­te der Wert immer mit dem Präfix *Placeholder* definiert wer­den. Somit sollte dies klar als Platzhalter erkannt und nicht als Secret identifiziert werden. Für die anderen Fälle ist es mög­lich, den Push zu forcieren. Hierzu muss die Commit-Message


```

curl -Lo $(Agent.TempDirectory)/sbom-tool
https://github.com/microsoft/sbom-tool/releases/
latest/download/sbom-tool-linux-x64
chmod +x $(Agent.TempDirectory)/sbom-tool
mkdir $(Build.ArtifactStagingDirectory)/sbom
$(Agent.TempDirectory)/sbom-tool generate
-b $(Build.ArtifactStagingDirectory)/sbom
-bc $(Build.SourcesDirectory) -pn HelloWorld
-pv $(Build.BuildNumber) -ps 4tecture
-nsb https://sbom.4tecture.ch -V Information
displayName: Generate SBOM
- task: PublishPipelineArtifact@1
inputs:
  targetPath:
    '$(Build.ArtifactStagingDirectory)/sbom'
  artifact: 'sbom'
  publishLocation: 'pipeline'
displayName: Publish SBOM artifact

# Publish the web application as a pipeline artifact
- task: PublishPipelineArtifact@1
inputs:
  targetPath: '$(Build.ArtifactStagingDirectory)'
  artifact: 'webapp'
  publishLocation: 'pipeline'
displayName: 'Publish Pipeline Artifact'

# Run Advanced Security Dependency Scanning
- task: AdvancedSecurity-Dependency-Scanning@1
# inputs:
#   directoryExclusionList: 'sampledir/
#   ignoreddirectory1;sampledir/ignoreddirectory'
displayName: 'Dependency Scanning'

# Run CodeQL Analysis
- task: AdvancedSecurity-Codeql-Analyze@1
displayName: 'CodeQL Analysis'

# Publish CodeQL Analysis Results
- task: AdvancedSecurity-Publish@1
displayName: 'Publish CodeQL Results'

# Add a CSV report for Advanced Security alerts
- task: PowerShell@2
displayName: 'Generate a csv report based on
Advanced Security alerts'
inputs:
  targetType: filePath
  filePath: $(Build.SourcesDirectory)/
    .azure-pipelines/ghazdo-csv-report.ps1
  pwsh: true
env:
  MAPPED_ADO_PAT: $(GHAzDO_PAT)

# Conditional step for Pull Requests: Run a PowerShell
# script for additional PR validation
- ${{ if eq(variables['Build.Reason'],
  'PullRequest') }}:
- task: PowerShell@2
displayName:
  'PR Gating - Additional Checks for Pull Requests'
inputs:
  targetType: filePath
  filePath: $(Build.SourcesDirectory)/
    .azure-pipelines/CIGate.ps1
  pwsh: true
env:
  MAPPED_ADO_PAT: $(GHAzDO_PAT)

```

den String `skip-secret-scanning:true` beinhalten. Hiermit wird die Push-Protection für diesen Commit ausgeschaltet.

Dependency Scanning

Das Dependency Scanning ist ein Pipeline-basiertes Scanning-Tool. Es funktioniert mittels YAML-Pipelines und Classic Builds. Im Artikel wird nur auf die YAML-Pipeline-Variante eingegangen, die Funktionsweise für die Classic Builds ist jedoch identisch. Einer Pipeline muss lediglich der Task `AdvancedSecurity-Dependency-Scanning@1` hinzugefügt werden. Das klingt erst einmal einfach, und es ist auch so.

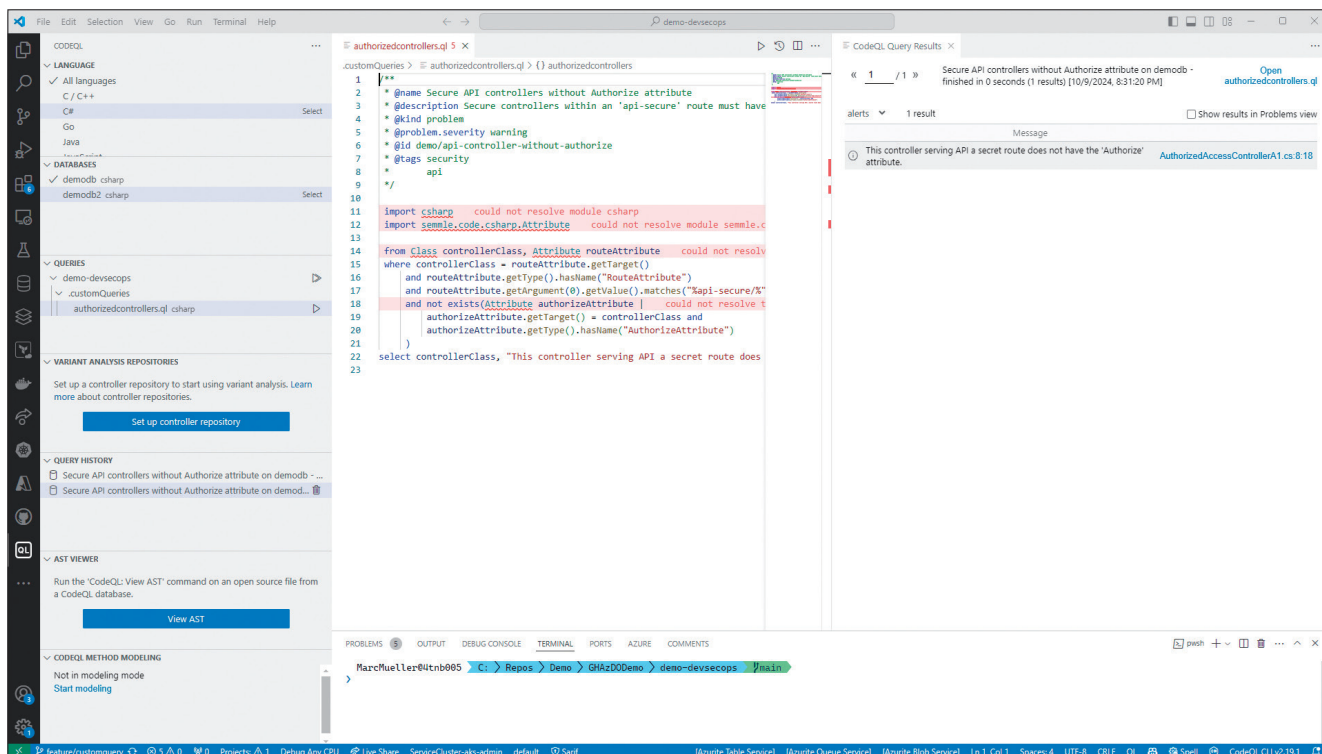
Beim Dependency Scanning werden alle Abhängigkeiten, direkt und transient, mit einer Vulnerability-Datenbank überprüft. Sollte eine Vulnerability gefunden werden, so wird ein entsprechender Alert erstellt. Advanced Security Dependency Scanning verwendet die GitHub Advisory Database, die unter [3] öffentlich verfügbar ist. Die Datenbank unterstützt

die gängigsten Package-Manager wie zum Beispiel NuGet, NPM und viele mehr.

Der zu Anfang erwähnte Task geht also für die unterstützten Package-Manager den Dependency Tree der Applikation durch und prüft jede Dependency mit der effektiven Version gegenüber der Datenbank.

Für die Pipeline, die den Dependency Scan ausführt, gibt es einige Punkte zu beachten. Einerseits sollten die Abhängigkeiten vor der Ausführung dieses Tasks wiederhergestellt worden sein. Bei .NET sollte also ein `dotnet restore` oder bei JavaScript-Projekten ein `npm install` ausgeführt worden sein. Zudem ergibt es Sinn, diese Pipeline nicht nur mittels eines Push-Triggers, sondern auch periodisch (zum Beispiel täglich) auszuführen. Auch ohne Code-Änderungen wollen wir über neu bekannt gewordene Vulnerabilities informiert werden.

Ist nun ein Alert vorhanden, gilt es diesen zu behandeln. Im einfachsten Fall gibt es bereits eine neuere Version der ►



CodeQL in VS Code (Bild 5)

Abhängigkeit, die die Sicherheitslücke schließt. Somit muss die Version aktualisiert werden, und der Alert wird automatisch geschlossen. Falls es keine Aktualisierung gibt, muss das Vorgehen projektspezifisch analysiert und entschieden werden. Entweder wird das Risiko getragen, oder es wird eine Alternative evaluiert. Im ersten Fall kann der Alert manuell geschlossen werden.

Einige GitHub-Nutzer:innen werden sich nun fragen, wie-so die Version des Pakets manuell aktualisiert werden muss, denn GitHub bietet mit Dependabot einen Automatismus, der automatisch einen Pull-Request mit dem Dependency-Update erstellt. Im Fall einer kompletten CI/CD-Automatisierung wird das Ganze im Pull-Request direkt noch automatisiert getestet, und der Aufwand für den Entwickler reduziert sich auf den Review und das Approval. Diese Feature ist leider (noch) nicht in Azure DevOps verfügbar.

Code Scanning

Das Code Scanning ist wie das Dependency Scanning ein Pipeline-basiertes Scanning-Tool. Entsprechend muss eine Pipeline mit den jeweiligen Tasks erweitert werden. Das Code Scanning beinhaltet drei Schritte:

- **Initialisierung** mittels *AdvancedSecurity-Codeql-Init@1*
- **Erstellen der Applikation** mittels Auto-Build oder eigenen CI-Build-Tasks (zum Beispiel *dotnet build*)
- **Analyse** mittels *AdvancedSecurity-Codeql-Analyse@1*

Die Code-Scanning-Pipeline kann problemlos mit dem Dependency Scanning kombiniert werden und ist oftmals der empfohlene Weg, um beide Scans in derselben Pipeline durchzuführen. Gehen wir zunächst aber nochmals auf das

Code Scanning ein. Advanced Security verwendet CodeQL als Abfragesprache, um den Code auf Schwachstellen zu untersuchen. Bei CodeQL handelt es sich um eine neue – oder wie es von Microsoft angekündigt wurde: revolutionäre – semantische Code Engine. CodeQL wurde während 13 Jahren von 30 Personen an der Oxford University entwickelt und hat schlussendlich über Akquisen den Weg zu GitHub gefunden. Der Sourcecode wird von der Engine analysiert und als Datenbank exportiert. Diese Datenbank kann nun mit der Abfragesprache CodeQL durchsucht werden. Eine Vulnerability wird also gefunden, wenn eine entsprechende Query ein oder mehrere Resultate zurückgibt. Die CodeQL-Engine muss dazu auf dem Agent vorhanden sein. Wie zu Beginn des Artikels erläutert, kann diese automatisch über den Task oder manuell auf dem Agent installiert werden. Der *Init*-Task dient zur Konfiguration der Engine und zum Aufbau der Datenbanken. Bei kompilierten Sprachen hängt sich die Engine an den Kompilationsprozess und baut so diese Datenbank auf. Deshalb auch das Dreierkonstrukt mit *Init*, *Build* und *Analyse* in der Pipeline. Microsoft hat die Konfiguration dieser Tasks einfach gehalten, und der *Init*-Task ist mehr oder weniger der einzige Task, der entscheidende Konfigurationswerte erlaubt. Mitunter wird hier auch die Query Suite definiert, also die Queries, mit denen nach Vulnerabilities gesucht wird. **Listing 1** zeigt den Aufbau einer solchen Pipeline anhand einer einfachen ASP.NET-Core-Applikation. Advanced Security bietet auch einen *Auto Build*-Task an, bei dem versucht wird, die Technologie zu erkennen und einen Build durchzuführen. In der Praxis klappt dies jedoch oft nicht, da die meisten Applikationen spezifische Anforderungen an den Build mit sich bringen. Darum werden die eigentlichen Build-Steps der Ap-

plikation von *Init* und *Analyze* umschlossen. Im Code von [Listing 1](#) sind noch weitere Tasks vorhanden, auf die wir gleich näher eingehen werden. Mit unserer Pipeline und den drei Advanced-Security-Tasks *AdvancedSecurity-Codeql-Init@1*, *AdvancedSecurity-Dependency-Scanning@1* und *AdvancedSecurity-Codeql-Analyze@1* haben wir eine funktionierende Pipeline, die unseren Code und die Dependencies untersucht und bei Sicherheitslücken entsprechende Alerts erstellt.

CodeQL-Queries

In unserer Beispiel-Pipeline verwenden wir im Initialisierungs-Task eine Query Suite mit den Namen *security-extended*. Microsoft stellt also verschiedene Sammlungen für die Überprüfung des Sourcecodes bereit. Aktuell sind eine Default Suite sowie die Security Extended Suite verfügbar. Eine Übersicht, welche Queries in der Suite vorhanden sind, ist bei GitHub [4] verfügbar. Die Queries selbst sind in einem GitHub-Repository [5] einsehbar.

GitHub möchte bewusst andere mit in die Entwicklung der Queries mit einbeziehen. Im Kern stehen die sogenannten GitHub Powered Queries. Hierbei handelt es sich um die wichtigsten Checks in Sachen Security und Vulnerabilities wie OWASP Top 10, SANS 25 sowie andere Best Practices. Zusätzlich werden sogenannte Community Powered Queries mit in die Suites aufgenommen. Unter Community Queries sind zum Beispiel Queries von Herstellern zu verstehen, welche diese nach dem Bekanntwerden einer Sicherheitslücke veröffentlichen. Microsoft hatte im Jahr 2021 die eigenen Queries als Open-Source-Code veröffentlicht, um Solorigate-Angriffen zu identifizieren [6]. Natürlich kann jeder respektive jede Firma noch zusätzliche Custom Queries erstellen, die dann in die Code-Scans mit aufgenommen werden. Diese Queries müssen nicht öffentlich sein und können sich im Code-Repository befinden.

Den letzteren Fall möchte ich anhand eines einfachen Beispiels veranschaulichen. Unser Szenario lautet wie folgt: In

der Vergangenheit ist es immer wieder vorgekommen, dass Entwickler vergessen hatten, kritische API-Controller mit einem *Authorize*-Attribut zu versehen. Kritische Controller sind in diesem Kontext alle Controller, die unter einer Route mit Präfix */api-secure* laufen. Sollte ein Controller mit diesem Pfad konfiguriert sein und kein *Authorize*-Attribut aufweisen, so wird ein Alert erstellt.

Hierzu müssen wir nun eine eigene CodeQL-Query schreiben. GitHub stellt einige Beispiele sowie eine Dokumentation zur Verfügung. Damit wir das nicht im Trial-and-Error-Verfahren mithilfe der Pipelines durchführen müssen, können wir die Queries natürlich lokal entwickeln und ausprobieren. Hierzu benötigen wir das CodeQL-CLI, mit dem wir eine entsprechende Datenbank von unserem Sourcecode erstellen und dann die Queries gegen diese Datenbank ausführen können. Am einfachsten konstruieren wir also den Fehlerfall in unserem Sourcecode und kompilieren die Solution zusammen mit dem CodeQL-CLI, die sich dann an den Compiler hängt und die Datenbank für die semantische Code-Analyse erstellt. Die Query können wir bequem mittels VS Code entwickeln und auch ausführen. Eine entsprechende VS-Code-Extension ist verfügbar, und in [Bild 5](#) sehen wir einen erfolgreichen Query Run mit Resultaten, der gegen eine lokale Datenbank ausgeführt wurde. Die Anleitungen für das CLI sowie die VS-Code-Extension sind unter [7] und [8] verfügbar.

Kümmern wir uns zunächst aber um unsere Query, die in [Listing 2](#) aufgeführt ist. Als Erstes importieren wir die Module für C# und die Attribute. Danach schreiben wir unsere Query, was ein wenig an SQL erinnert. Mit der *from*-Anweisung durchsuchen wir Klassen und Attribute. In unserem Beispiel ist ja die Kombination von zwei bestimmten Attributen, die nicht zusammen auf der gleichen Klasse zu finden sind, der Ansatz, um diesen Alert auszulösen. Konkret also, wenn ein Controller unter einem bestimmten Pfad läuft und nicht über ein *Authorize*-Attribut eingeschränkt ist. Wir filtern daher zunächst alle Attribute, die auf der Klasse gesetzt sind (mit ►

● Listing 2: Custom-CodeQL-Query

```
/**
 * @name Secure API controllers without Authorize
 * attribute
 * @description Secure controllers within an
 * 'api-secure' route must have an Authorize
 * attribute.
 * @kind problem
 * @problem.severity warning
 * @id demo/api-controller-without-authorize
 * @tags security
 * api
 */

import csharp
import semmle.code.csharp.Attribute

from Class controllerClass, Attribute routeAttribute
where controllerClass = routeAttribute.getTarget()
  and routeAttribute.getType().hasName(
    "RouteAttribute")
  and routeAttribute.getArgument(0).getValue().
    matches("%api-secure/%")
  and not exists(Attribute authorizeAttribute |
    authorizeAttribute.getTarget() =
      controllerClass and
      authorizeAttribute.getType().hasName(
        "AuthorizeAttribute")
  )
select controllerClass, "This controller serving API
a secret route does not have the 'Authorize'
attribute."
```

`getTarget`) sowie vom Typ `RouteAttribute`. Im Fall eines `Route`-Attributs muss dieses noch als Argument den Wert `api-secure` aufweisen. Der nächste Schritt ist eine Sub-Query. Wir filtern erneut die Attribute derselben Klasse, dieses Mal jedoch vom Typ `AuthorizeAttribute`. Diese Sub-Query verwenden wir mit einem `not exists`-Filter. Somit bekommen wir alle Klassen, die das `Route`-Attribut mit dem bestimmten Wert gesetzt haben, aber kein `Authorize`-Attribut aufweisen. Für alle gefundenen Klassen wird dann ein entsprechender Alert erstellt. Was in diesem Alert genau steht, wird über den Kommentar dieser Queries definiert. Hier definieren wir Name, Description sowie die Kategorisierung.

Nun müssen wir diese Query noch in den Code-Scan mit aufnehmen. Wir können die Query zusätzlich zur Standard-Suite ausführen oder ein eigenes Query-Pack definieren. In [Listing 3](#) lagern wir die Konfiguration der CodeQL-Queries in eine YAML-Datei aus, und in [Listing 4](#) verändern wir den `AdvancedSecurity-CodeQL-Init-Task` so, dass er diese Konfiguration anzieht.

Wenn wir nun die Probe aufs Exempel machen und einen neuen Controller für den geschützten Pfad ohne Autorisierungs-Attribut in unserem Feature-Branch einfügen, wird über die PR-Policy unser CI-Build mit Code Scanning gestartet und wir erhalten den Alert, der in [Bild 6](#) dargestellt ist.

● Listing 3: CodeQL-Konfiguration

```
name: "Run custom queries"

disable-default-queries: false

queries:
  - name: Authorized controllers
    uses: ../customQueries/authorizedcontrollers.q1
```

Mit der Custom-Query-Option können wir uns also neben den Standard-Checks mit relativ wenig Aufwand auf die Suche nach applikationsspezifischen Sicherheitslücken oder gängigen Programmierfehlern machen. Die firmeneigene Query-Suite kann so zentral für alle Teams gepflegt und stetig erweitert werden.

Ein letzter Hinweis zu den Custom-Query-Möglichkeiten: CodeQL ist außerhalb des Forschungsbereichs lizenzpflichtig. Mit der Aktivierung von GitHub Advanced Security / GitHub Advanced Security for Azure DevOps ist eine entsprechende Lizenz vorhanden. Ohne diese Subscription darf das CodeQL-CLI nur für Forschungszwecke verwendet werden.

Pipeline-Erweiterungen

Grundsätzlich haben wir nun die Funktionalität von Advanced Security implementiert und die drei verschiedenen Scans – Code, Dependency und Secrets – über Konfiguration und Pipelines angewendet, und wir erhalten entsprechende

● Listing 4: Custom-CodeQL-Query

```
steps:
  # Initialize Advanced Security CodeQL
  - task: AdvancedSecurity-Codeql-Init@1
    inputs:
      languages: 'csharp'
      queriesuite: 'security-extended'
      configfilepath: '$(build.sourcesDirectory)/
        .azure-pipelines/codeqlconfiguration.yml'
      displayName: 'Initialize CodeQL'
```

Alerts in der Advanced-Security-Ansicht. Jedoch fühlt es sich noch nicht ganz rund an. Wenn wir die Azure-DevOps-Integration von Advanced Security mit der Integration von GitHub vergleichen, fällt einem ziemlich schnell auf, dass bei GitHub nebst dem Dependabot auch eine Pull-Request-Integration vorhanden ist. Dies sind beides Funktionen, welche die Effizienz für die Erkennung und Abarbeitung der Alerts drastisch erhöhen. Viele Teams arbeiten mit einem Pull-Request-Workflow, der über Policies und Pipelines sicherstellt, dass alle Qualitätsanforderungen vor der Integration erfüllt sind. Bei GitHub werden die Alerts, die in einem Pull-Request-Build festgestellt werden, gleich automatisch in den PR hineinkommentiert, und der Pull-Request kann ohne deren Abarbeitung nicht geschlossen werden. Dies hätten wir auch gerne bei Azure DevOps. Leider müssen wir uns damit noch ein wenig gedulden.

In der Zwischenzeit können wir uns mit eigenen Skripten in den Pipelines behelfen. Advanced Security ist über das API verfügbar und hat auch dedizierte Permissions, die in den Repository-Einstellungen verwaltet werden können. Somit können wir relativ einfach über das API abfragen, ob es Alerts für den aktuellen PR gibt, und anhand der Alert-Informationen, die Branch, Datei und Zeilennummern enthalten, einen automatisierten PR-Kommentar erstellen. Microsoft hat unter [9] sogar ein GitHub-Repository erstellt, in dem alle diese Skripte als Beispiele vorhanden sind. Über das Einbinden des PowerShell-Skripts `CIGate.ps1` lassen sich automatisch PR-Kommentare erstellen, und der Pull-Request kann blockiert werden, sollten nicht gelöste Alerts vorhanden sein. Das [Listing 1](#) zeigt das Einbinden dieser Skripte in der Pipeline in den letzten zwei Tasks. Böse Zungen würden nun fragen, wieso Microsoft die PR-Integration über Skriptbeispiele zur Verfügung stellt, diese aber nicht wie bei GitHub als Standardfunktion in die Pull-Requests mit aufnimmt und der Benutzer diese Kommentare ohne ein Customizing einfach angezeigt bekommt. Wie immer wird es an den Prioritäten im Backlog gelegen haben, und wir können hoffen, dass Azure DevOps hier bald nachzieht und einen ähnlichen Funktionsumfang im PR bieten wird, wie ihn Advanced Security bei GitHub aufweist.

Die Pipeline in [Listing 1](#) beinhaltet noch einen weiteren Task. Auf Zeile 53 ff. wird ein SBOM unserer Software erstellt, der dann im Folgeschritt als Pipeline-Artefakt publiziert wird.

Secure API controllers without Authorize attribute (demo/api-controller-without-authorize)

#9 **Open** in pull/283 • detected Feb 14 at 2:20 PM

Overview Detections

Location

HelloWorld/HelloWorld/Controllers/AuthorizedAccessControllerA1.cs:8 @ 359df21d

```

4 namespace HelloWorld.Controllers
5 {
6     [Route("api-secure/[controller]")]
7     [ApiController]
8     public class AuthorizedAccessControllerA1 : ControllerBase
9     {
10         [HttpGet("GetSecretInfo")]
11         public IActionResult GetSecret()
12         {
13             return Ok("Something secret");
14         }
15     }
16 }

```

Description

Severity
Warning

First detected
Feb 14 at 2:20 PM

Finding details

Tool
CodeQL

Rule ID
demo/api-controller-without-authorize

Alert von einer Custom Query (Bild 6)

Somit können wir unsere Anwender respektive Kunden über die Zusammensetzung, also den gesamten Abhängigkeitsbaum unserer Software, informieren. Der Kunde kann dann anhand dieser Informationen ebenfalls eine Vulnerability-Datenbank durchsuchen. Ein Software Bill of Material, kurz SBOM, ist eine Stückliste aller Komponenten der Software, die sich in einem standardisierten Format exportieren lässt. Microsoft bietet dafür derzeit ein eigenes Tool als CLI an, das auf GitHub verfügbar ist [10] und ohne großen Aufwand in eine Pipeline integriert werden kann.

Fazit

Mit GitHub for Advanced Security bekommen wir ein DevSecOps-Tool, das sich ohne großen Konfigurationsaufwand in unsere bestehende Azure-DevOps-Umgebung integriert. Das Softwareentwicklungsteam erhält quasi auf Knopfdruck alle wichtigen Sicherheits-Scans, welche die statische Code-Analyse zu bieten hat. Die Alerts beinhalten die Traceability respektive Verlinkung zu den Code-Artefakten. Über das bestehende Berechtigungssystem können die Rechte für das Lesen und Verwalten der Alerts administriert werden. GitHub stellt eine eigene, kuratierte Vulnerability-Datenbank zur Verfügung. Diese kann als vertrauenswürdig betrachtet werden, zumal GitHub als größte Open-Source-Hosting-Plattform an der Quelle sitzt, schnell über Sicherheitslücken informiert zu werden, respektive diese über die eigenen Tools erkennen kann. Ebenfalls wichtig zu erwähnen ist hier, dass Advanced Security für Open-Source-Projekte kostenfrei ist und so die Abhängigkeiten unserer Applikationen immer sicherer und stabiler werden. Die Möglichkeit, eigene CodeQL-Queries in die Scans zu integrieren, rundet das Angebot ab. Die nicht vorhandene PR-Integration in Azure DevOps ist dagegen ein klarer Wermutstropfen, zumal Advanced Security in GitHub für den gleichen Preis deutlich mehr Funktionalität und Usability bietet. Hier warten wir gespannt, bis Microsoft diese Funktionen auch bei Azure DevOps integriert.

Auch wenn man aufgrund der zusätzlichen Lizenzkosten erstmal ein wenig zurückhaltend ist, lohnt sich diese Investi-

tion schnell. Zum einen ist ein möglicher Schadensfall weit aus teurer, und auch der manuelle Überprüfungsaufwand, der mit dieser Automatisierung eingespart werden kann, wird diese Investition in Kürze amortisiert haben. ■

[1] Microsoft Learn, *Configure GitHub Advanced Security for Azure DevOps*, www.dotnetpro.de/SL2502DevSecOps1

[2] Microsoft Learn, *Secret scanning*, www.dotnetpro.de/SL2502DevSecOps2

[3] GitHub Advisory Database, <https://github.com/advisories>

[4] GitHub Docs, *C# queries for CodeQL analysis*, www.dotnetpro.de/SL2502DevSecOps3

[5] *codeql-queries* auf GitHub, www.dotnetpro.de/SL2502DevSecOps4

[6] Microsoft Security Blog, *Microsoft open sources CodeQL queries used to hunt for Solorigate activity*, www.dotnetpro.de/SL2502DevSecOps5

[7] GitHub Docs, *About the CodeQL CLI*, www.dotnetpro.de/SL2502DevSecOps6

[8] GitHub Docs, *About CodeQL for VS Code*, www.dotnetpro.de/SL2502DevSecOps7

[9] *GHAzDO-Resources* auf GitHub, www.dotnetpro.de/SL2502DevSecOps8

[10] *sbom-tool* auf GitHub, www.dotnetpro.de/SL2502DevSecOps9



Marc Müller

arbeitet als Principal Consultant für DevOps und .NET-/Azure-Lösungen bei 4tecture und ist Microsoft MVP für DevOps. Zu seinen Schwerpunkten zählen Ausbildung und Coaching von DevOps- und .NET-Projektteams.

www.4tecture.ch

dnpCode

A2502DevSecOps



LERNSTOFF

dnpCode A2502Lernstoff

PRAXISBUCH

KI-Coding

Den Autoren ist ein besonnener Titel gelungen, der gut in die Zone zwischen Hype und Ernüchterung passt. Es geht darum, pragmatisch die Möglichkeiten zu nutzen, die schon die bisher verfügbaren KI-Tools bereitstellen. Sie verkürzen die Zeit für öde Routinearbeiten. Gleichzeitig kann die Arbeit von Vorschlägen profitieren, die durch geschicktes Prompting gewonnen werden. Der Umgang mit ChatGPT oder Copilot erinnert an ein Gespräch, in dem intelligente Antworten meist auch nur durch intelligente Fragen gewonnen werden können. Dafür werden zahlreiche Beispiele angeboten.

Im ersten Teil werden kurz das Coding per Chat, die Kunst des Promptings und die Rolle der Large Language Models (LLMs) eingeführt. Dann wird der KI-Einsatz in den verschiedenen Bereichen der Softwareentwicklung wie Programmierung, Debugging, Refactoring, Testen, Dokumentieren, Einbindung von Datenbanken, Scripting und Administration behandelt. Dies wird unterstützt durch zahlreiche Codebeispiele



Michael Kofler, Bernd Öggl, Sebastian Springer

Coding mit KI. Das Praxisbuch für die Softwareentwicklung

Rheinwerk, 2024, 412 Seiten, Deutsch, ISBN 978-3-367-10344-7, 39,90 Euro, E-Book 27,93 Euro

Zielgruppe: Softwareentwicklung

Voraussetzungen: Programmiererfahrung

Leseprobe:

www.dotnetpro.de/SL2502Lernstoff1

le und Abwägungen, wo mit guten und wo mit weniger verlässlichen Ergebnissen zu rechnen ist.

Der zweite Teil geht auf Versuche ein, die Sprachmodelle lokal, also unabhängig von Cloud-Diensten, zu nutzen. Untersucht werden Werkzeuge wie GPT4All, Ollama, Continue oder Tabby. Vorgestellt werden zudem Möglichkeiten, vorhandene Sprachmodelle durch aktuelle Daten zu ergänzen. Tools wie OpenHands und Aider versprechen ein höheres Level in der Automatisierung der Codegenerierung. (vh) ■

SOFTWARE VERBESSERN MIT KI

ChatGPT-Praxis

Schon im Vorwort weist der Autor darauf hin, was das vorliegende Buch nicht ist: Eine Anleitung, um neuen Code mit ChatGPT zu generieren. Vielmehr geht es darum, wiederkehrende Aufgaben im Prozess der Softwareentwicklung mit ChatGPT zu optimieren, und das anhand vieler praktischer Beispiele von Prompts und der KI-Antworten darauf.

Die ersten beiden Abschnitte erklären grundlegende Zusammenhänge, vom Machine Learning bis zum Optimieren von Prompts, sowie allgemeine Techniken und Aspekte in Bezug auf die Qualität von Software. Dabei werden auch Schwächen von KI-Antworten betrachtet. Erst danach geht es darum, wie man die Codequalität mithilfe von LLMs lesbarer gestalten kann. Das wird anhand von Beispiel-Prompts und den Antworten darauf erläutert. Ein eigenes Kapitel fasst den Nutzen besserer Codequalität durch Anforderungen und Dokumentation theoretisch zusammen, gefolgt von einer Reihe



Patrick Schnell

ChatGPT in Softwareprojekten

O'Reilly, 2024, 206 Seiten, Deutsch, ISBN 978-3-96009-237-7, 26,90 Euro

Zielgruppe:

Entwickler, Projektleiter, Product Owner

Voraussetzungen: keine

Leseprobe:

www.dotnetpro.de/SL2502Lernstoff2

von Prompts, mit denen man ChatGPT dafür nutzt, Anforderungen und Dokumentation zu verbessern. Auch für das Beschreiben von Algorithmen wird die KI eingesetzt.

Die Prompt-Beispiele sind recht umfangreich. Es kommt vor, dass Prompt plus Antwort zwei oder mehr Buchseiten belegen. Man findet also nicht nur triviale Mini-Prompts, sondern Beispiele, die recht nah am Entwickleralltag sind. Abschließend skizziert der Autor noch seine Gedanken zur Zukunft von KI in Softwareprojekten. (bl) ■



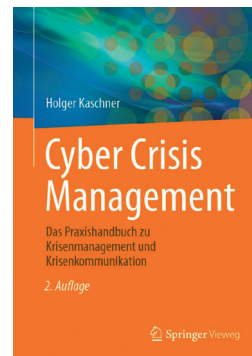
HANDBUCH KRISENMANAGEMENT

Cyber Crisis

Ausgehend von dem schweren Cyberangriff auf Sony 2014 als lehrreichem Beispiel stellt Holger Kaschner zunächst den Faktor Mensch in den Mittelpunkt des Umgangs mit Cyberkrisen. Typische Stressreaktionen und Verhaltensmuster stellen bestimmte Anforderungen an Personen, welche die Krisenbewältigung organisieren. Die Antwort auf eine bedrohliche Krise erfordert zunächst einen effektiven Informationsaustausch über die Lage und Handlungsalternativen, Regeln für die Krisenkommunikation sowie die Wahl der Reaktionen auf der taktisch-operativen Ebene.

Wie sich eine Organisation auf solche Krisen vorbereiten kann, wie ein Krisenstab am besten zusammengesetzt werden sollte und welche Hilfsmittel oder vorbereitende Trainings sinnvoll sind, wird ausführlich behandelt.

Ein weiterer Schwerpunkt ist die Krisenprävention. Dazu gehören Elemente der IT-Sicherheitsarchitektur, Frühwarnsysteme, vorausschauende Risikoanalysen sowie Audits für Cyber Resilience.



Holger Kaschner

Cyber Crisis Management

Springer, 2. Auflage 2024, 271 Seiten, Deutsch, ISBN 978-3-658-43464-9, 44,99 Euro, E-Book 34,99 Euro

Zielgruppe:

Verantwortliche für IT-Sicherheit und Führungskräfte

Voraussetzungen: Keine

Leseprobe (Anmeldung erforderlich): www.dotnetpro.de/SL2502Lernstoff3

Anschließend wird zusammengestellt, worauf es besonders ankommt, um die Ordnung in der Krisensituation zurückzugewinnen. Dazu gehört Klarheit über den tatsächlichen Schutzbedarf und den Business Impact. Auch die Nachbereitung einer überstandenen Krise auf der strategischen und der taktischen Ebene wird sorgfältig behandelt. Zur Abschreckung folgt am Ende noch eine Liste der größten Fehler, die im Krisenmanagement vorkommen. Zahlreiche Abbildungen und Checklisten unterstützen den Text. (vh) ■

Kurz vorgestellt

Ashirwad Satapathi

Building Intelligent Apps with .NET and Azure AI Services

Das Buch erklärt, wie man Azure AI Services bereitstellt und nahtlos in .NET-Anwendungen integriert. Als Beispiel dient eine Textübersetzer-App mit .NET MAUI. Außerdem werden KI-Dienste und Lösungen vorgestellt.

Apress, 2024, 214 Seiten, Englisch, ISBN 979-8-8688-0434-2, 64,19 Euro, E-Book 46,99 Euro

Allen B. Downey

Python lernen mit KI-Tools

Praxiskurs Python mit einer Schritt-für-Schritt-Einführung. Für die Übungen nutzt der Autor LLMs wie ChatGPT, mit denen der Lernprozess beschleunigt werden kann, und zeigt den Nutzen von KI-Tools beim Testen von Code und der Fehlersuche.

O'Reilly, 3. Auflage 2024, 360 Seiten, Deutsch, ISBN 978-3-96009-259-9, 29,90 Euro

André Krämer

Cross-Plattform-Apps mit .NET MAUI entwickeln

Die Neuauflage des Leitfadens für den Einstieg in .NET MAUI wurde erweitert um XAML-Grundlagen, die Entwicklung eigener Steuerelemente, das Lokalisieren von Apps sowie das Deployment von Apps in die App Stores. Mit durchgängigem Beispiel.

Hanser, 2. Auflage 2024, 724 Seiten, Deutsch, ISBN 978-3-446-47981-4, 49,99 Euro

Eric Amberg, Daniel Schmid

Hacking

Praxisorientierter Leitfaden für Penetration-Tester und IT-Systembeauftragte mit Fokus auf Angriffstechniken, die jeder Penetration-Tester kennen muss. Vorgestellt werden Gegenmaßnahmen, die effektiv vor Angriffen schützen. Mit Testfragen zur Vorbereitung auf Zertifizierungen.

mitp, 3. Auflage 2024, 1232 Seiten, Deutsch, ISBN 978-3-7475-0871-8, 49,99 Euro

LLAMASHARP

.NET-Anbindung für die Llama-KI

Open-Source-KI auf dem lokalen Gerät nutzen – mit handhabbarem Speicherbedarf und optionalem Zugriff auf externe Datenquellen.

Wer die Tech-News von heute verfolgt, stellt schnell fest: Fast jedes Produkt braucht inzwischen einen CoPilot oder ein anderes KI-Gimmick, um im Gespräch zu bleiben. OpenAI und deren KI-Dienst sind weithin bekannt, aber gibt es auch eine Möglichkeit, eine KI „lokal“ – also ohne riesige Rechenzentren im Hintergrund – zu nutzen? Genau hier kommt LLamaSharp ins Spiel.

Was ist Llama?

Llama ist ein Large Language Model (LLM) von Meta, das ähnliche Funktionen wie das bekannte ChatGPT von OpenAI bietet: Es kann menschliche Sprache verstehen und wiedergeben. Der wesentliche Unterschied zu OpenAI liegt darin, dass Meta sein Modell als Open Source bereitstellt. Dadurch hat sich ein breites Ökosystem rund um Llama entwickelt.

Ein zentraler Bestandteil dieses Ökosystems ist die Bibliothek llama.cpp [1], die es ermöglicht, Llama auch auf Geräten mit begrenzten Ressourcen lokal auszuführen. Diese Geräte können zum Beispiel ein Server im heimischen Rack, ein Laptop oder sogar ein Smartphone sein. Für den Hauptentwickler Georgi Gerganov war es von Beginn an wichtig, dass man Llama auch ohne leistungsstarke Grafikkarten direkt auf der CPU nutzen kann – was zur enormen Beliebtheit dieser Bibliothek beigetragen hat.

Das Ökosystem rund um Llama und GGUF

Durch das Open-Source-Modell von Llama und die Vorteile der kostengünstigen Nutzung von llama.cpp sind auch andere Sprachmodelle von verschiedenen Firmen entstanden, die mit llama.cpp kompatibel sind. Inzwischen hat sich ein Sprachformat namens GGUF (Grok's General Unified Format) etabliert. Dieses Format wird von Sprachmodellentwicklern und Tools genutzt und ist mit llama.cpp kompatibel.

LLamaSharp – .NET trifft auf Llama

LLamaSharp [2] basiert auf llama.cpp und stellt dessen Funktionen und APIs als NuGet-Paket für die .NET-Welt bereit. Damit eröffnet LLamaSharp .NET-Entwickler:innen den Zugang zu leistungsfähiger Sprachverarbeitung und KI-Funktionen direkt auf ihren Geräten.

Ein wichtiger Hinweis vorweg: Dieser Artikel basiert auf der zum Zeitpunkt der Abfassung aktuellen Version 0.18 von LLamaSharp vom Oktober 2024. Das API ist seit einigen Ver-

sionen relativ stabil, jedoch handelt es sich bei KI-Themen generell um ein dynamisches und schnelllebiges Umfeld.

Erste Schritte mit LLamaSharp

Die ersten Schritte mit LLamaSharp sind recht schlicht. Es beginnt mit dem NuGet-Package LLamaSharp. Dieses Package bringt Infrastrukturcode und Definitionen mit sich.

Um die Modelle auszuführen, verwendet LLamaSharp sogenannte Backends, die so eingesetzt werden sollten, dass sie zur eigenen Software und vor allem Hardware passen. Die Backends nutzen das Kompilat aus dem llama.cpp-Projekt und laufen daher mit der nahezu gleichen Geschwindigkeit wie das native llama.cpp. Folgende Backends sind derzeit für LLamaSharp verfügbar:

- **LLamaSharp.Backend.Cpu:** In diesem Backend läuft die Berechnung für Windows und Linux auf der CPU, und für Mac gibt es damit auch direkt Metal-(GPU-)Unterstützung.
- **LLamaSharp.Backend.Cuda11:** Für alle GPUs von Nvidia, die CUDA 11 auf Windows und Linux unterstützen
- **LLamaSharp.Backend.Cuda12:** Für CUDA-12-GPUs unter Windows und Linux.
- **LLamaSharp.Backend.Vulkan:** Für alle Vulkan-fähigen GPUs unter Windows und Linux.

Alle diese Backends stehen als NuGet-Packages bereit und müssen lediglich zum Projekt hinzugefügt werden. Grundsätzlich können auch mehrere Backends hinzugefügt werden (zum Beispiel CPU und CUDA12), sodass LLamaSharp je nach Hardware entweder auf der CPU oder der GPU arbeitet. Mit der Version 0.18 scheint es hierbei jedoch gelegentlich Probleme zu geben, sodass die GPU ignoriert wird, siehe dazu das GitHub-Issue #456 [3].

LLamaSharp-Quickstart

Gehen wir davon aus, dass wir nur mit der CPU arbeiten wollen, so reichen folgende zwei NuGet-Pakete für einen rudimentären Chatbot aus:

```
dotnet add package LLamaSharp
dotnet add package LLamaSharp.Backend.Cpu
```

Der Code in Listing 1 lässt einen Chatbot auf der Kommandozeile entstehen. Hinweis: Der Code entspricht der aktuellen

Projekt-*ReadMe*, allerdings mit einer Fehlerbereinigung. Die *InferenceParams* müssen unbedingt mit der *DefaultSamplingPipeline* initialisiert werden, ansonsten kommt es zu einer Null-Reference-Exception.

Quickstart – Erklärung

Der erste Teil des Codes in [Listing 1](#) bis zur *ChatHistory* betrifft nur das Laden des Modells. Wie man konkret zu einem Modell kommt und ein passendes auswählt, wird im nächsten Abschnitt beschrieben. Als Einstieg kann man mit den vorgegebenen Werten loslegen, ohne irgendwelche Befürchtungen haben zu müssen, dass das System gleich in die Brüche geht. In einer Produktionsumgebung ist jedoch wahrscheinlich eine detailliertere Anpassung erforderlich.

Anschließend wird die *ChatHistory* instanziiert. Wer bereits Erfahrungen mit ChatGPT-Einstellungen hat, wird das Mus-

ter erkennen: Ein genereller Kontext wird als *System*-Nachricht festgelegt und definiert die Grundregeln für den Chatbot. Der *Assistant* repräsentiert den eigentlichen Chatbot, und der Benutzer wird als *User* bezeichnet. Die *ChatHistory* dient zur Speicherung des Gesprächsverlaufs und wird der *Session* übergeben. Die Interaktion mit dem Benutzer erfolgt dann innerhalb der *while*-Schleife.

So viel zum Code, jedoch bleibt die Wahl des Sprachmodells der wichtigste Faktor.

Modellsuche mit LM Studio

Der einfachste Weg, an Modelle zu kommen, führt über LM Studio. Dieses kostenlose Programm läuft sowohl unter Linux und Windows als auch unter macOS und ermöglicht es, lokale LLMs (Large Language Models) herunterzuladen und direkt auf dem eigenen System zu verwenden. ►

● Listing 1: Ein Chatbot auf der Kommandozeile

```
using LLama.Common;
using LLama;

string modelPath = @"<Your Model Path>";
// change it to your own model path.

var parameters = new ModelParams(modelPath)
{
    ContextSize = 1024,
    // The longest length of chat as memory.
    GpuLayerCount = 5
    // How many layers to offload to GPU. Please
    // adjust it according to your GPU memory.
};
using var model = LLamaWeights.LoadFromFile(parameters);
using var context = model.CreateContext(parameters);
var executor = new InteractiveExecutor(context);

// Add chat histories as prompt to tell AI how to act.
var chatHistory = new ChatHistory();
chatHistory.AddMessage(AuthorRole.System,
    "Transcript of a dialog, where the User interacts
    with an Assistant named Bob. Bob is helpful,
    kind, honest, good at writing, and never fails to
    answer the User's requests immediately and with
    precision.");
chatHistory.AddMessage(
    AuthorRole.User, "Hello, Bob.");
chatHistory.AddMessage(AuthorRole.Assistant,
    "Hello. How may I help you today?");

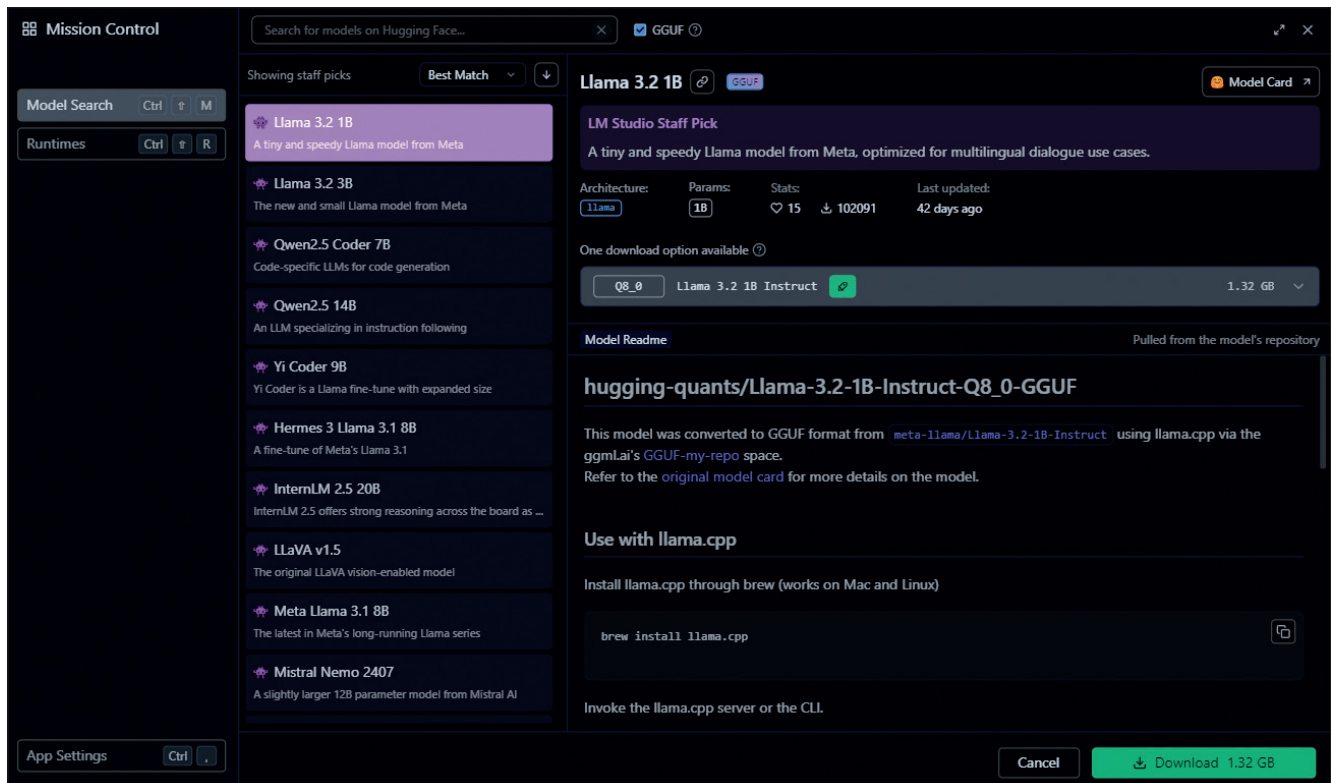
ChatSession session = new(executor, chatHistory);

InferenceParams inferenceParams =
    new InferenceParams()
{
    SamplingPipeline = new DefaultSamplingPipeline(),
    // Use default sampling pipeline
    MaxTokens = 256,
    // No more than 256 tokens should appear in
    // answer. Remove it if antiprompt is enough for
    // control.
    AntiPrompts = new List<string> { "User:" }
    // Stop generation once antiprompts appear.

    SamplingPipeline = new DefaultSamplingPipeline(),
};

Console.ForegroundColor = ConsoleColor.Yellow;
Console.Write(
    "The chat session has started.\nUser: ");
Console.ForegroundColor = ConsoleColor.Green;
string userInput = Console.ReadLine() ?? "";

while (userInput != "exit")
{
    await foreach (
        // Generate the response streamingly.
        var text
        in session.ChatAsync(
            new ChatHistory.Message(AuthorRole.User,
                userInput),
            inferenceParams))
    {
        Console.ForegroundColor = ConsoleColor.White;
        Console.Write(text);
    }
    Console.ForegroundColor = ConsoleColor.Green;
    userInput = Console.ReadLine() ?? "";
}
```



Auswahl eines KI-Modells über die „Discover“-Funktion in LM Studio (Bild 1)

Neben den Basisfunktionen bietet LM Studio auch einen integrierten Chatbot, sodass Modelle direkt getestet werden können. Außerdem gibt es ein lokales Web-API im Stil des OpenAI-API, über das Anwendungen nahtlos auf die Modelle zugreifen können.

Wichtig: LM Studio hostet selbst keine Modelle, sondern greift auf die Daten von Huggingface.co zu. Man könnte also sagen: Was GitHub für Code ist, ist Huggingface für KI-

Modelle. Dort finden sich nicht nur Modelle zur Erstellung von Chatbots, sondern auch solche für Aufgaben wie Bilderkennung oder -erzeugung, Audioverarbeitung und andere mehr. In LM Studio beschränkt man sich allerdings aktuell auf die Chatbot-Modelle.

Um unser LLamaSharp-Beispiel auszuführen, suchen wir ein Modell über die *Discover*-Funktion in LM Studio aus, siehe Bild 1.



Heruntergeladene Modelle finden sich unter „My Models“ (Bild 2)

Modellwahl

Wie in **Bild 1** zu sehen ist, stehen uns weit mehr als nur ein Modell zur Verfügung. Aktuell gilt: „Große“ Modelle bewältigen deutlich komplexere Aufgaben als kleinere. Die LLM-Community bietet auf der Plattform LMArena.ai [4] eine Art Ranking für diese Modelle an, in dem verschiedene Leistungsaspekte verglichen werden.

Der derzeitige Spitzenreiter ist das Modell ChatGPT-4o. Allerdings ist es proprietär und aufgrund seiner enormen Größe für lokale Anwendungen ausgeschlossen.

Ein guter Kompromiss – besonders für experimentelle Zwecke – ist das Modell „Phi 3.1 Mini 128k“ von Microsoft. Es ist mit einer Größe von 3,1 GByte relativ kompakt und kann daher problemlos in den Arbeitsspeicher der meisten aktuellen Systeme geladen werden.

Das heruntergeladene Modell findet sich später unter dem Menüpunkt *My Models* wieder, siehe **Bild 2**. Von dort können wir den Dateipfad zum Modell herauskopieren und nun in unsere Demoanwendung einfügen.

Demo mit Phi 3.1

Nachdem wir die Demo mit dem Phi-3.1-Modell gestartet haben, erhalten wir zunächst viele Informationen rund um das Modell. Anschließend kommt der „Assistent“ ins Spiel, und es ist möglich, mit ihm wie in **Bild 3** gezeigt Nachrichten auszutauschen.

Mehr Wissen mit RAG

Sprachmodelle verfügen über ein grundlegendes Verständnis der Welt, jedoch fehlt ihnen oft Detailwissen. Besonders kleinere Modelle stoßen hier schnell an ihre Grenzen, aber auch größere Modelle wie ChatGPT benötigen zusätzliches „Domänenwissen“, um spezifische Themen abzudecken.

Eine einfache Möglichkeit hierfür besteht wie in **Bild 3** gezeigt darin, relevante Informationen als Chat-Nachricht in das Gespräch einzubringen. Das funktioniert gut für kleinere Textabschnitte, ist jedoch bei längeren Dokumenten kaum praktikabel. Hier kommt RAG (Retrieval-Augmented Generation) ins Spiel. Mit RAG kann ein Sprachmodell auf externe Datenquellen zugreifen und so Informationen dynamisch abrufen. LLamaSharp unterstützt RAG ebenfalls: mithilfe des Kernel-Memory-Projekts von Microsoft [5].

Überblick und Fazit

Wie bereits im Intro erwähnt, ist das Thema KI allgegenwärtig – und dabei äußerst umfangreich. LLamaSharp bietet hier eine elegante Lösung als Wrapper für das llama.cpp-Projekt und macht den Einstieg in die Arbeit mit KI für .NET-Entwicklerinnen und -Entwickler deutlich einfacher. Dank der Integration des Kernel-Memory-Projekts von Microsoft steht auch die Option zur Verfügung, Retrieval-Augmented Generation (RAG) [6] zu nutzen und somit dynamisch auf externe Datenquellen zuzugreifen. ►

DEV ACADEMY

HANDS-ON-WORKSHOPS UND
WEITERBILDUNG FÜR SOFTWARE-
ENTWICKLER UND -ARCHITEKTEN

3 TAGE
REMOTE

C# für Fortgeschrittene

- Klassenbibliotheken, Vererbung
- Schnittstellen
- Events
- Linq
- UnitTests mit MSTest
- Debugging
- Dependency Injection
- Entwurfsmuster, Kodierungsrichtlinie

TRAINING!



David Tielke
Trainer

Für alle, die sich mit LLMs beschäftigen oder dies planen, ist LM Studio eine klare Empfehlung. Besonders für Anfänger, die noch nie von „GGUF“ oder „Huggingface“ gehört haben, stellt das Programm eine wertvolle Hilfe dar: Es ver-

dung vollständig lokal ausgeführt werden soll, ist ein Umweg über ein WebAPI eher hinderlich. In anderen Konstellationen sollte jedoch geprüft werden, ob eine API-basierte Lösung sinnvoll ist. ■

```

C:\Users\muehsig\source\rep x + v
llm_load_print_meta: PAD token      = 32000 '<|endoftext|>'
llm_load_print_meta: LF token       = 13 '<0x0A>'
llm_load_print_meta: EOT token      = 32007 '<|end|>'
llm_load_print_meta: max token length = 48
llm_load_tensors: ggml ctx size = 0.10 MiB
llm_load_tensors: offloading 5 repeating layers to GPU
llm_load_tensors: offloaded 5/33 layers to GPU
llm_load_tensors: CPU buffer size = 2281.66 MiB
.....
llama_new_context_with_model: n_ctx      = 1024
llama_new_context_with_model: n_batch    = 512
llama_new_context_with_model: n_ubatch   = 512
llama_new_context_with_model: flash_attn = 0
llama_new_context_with_model: freq_base  = 10000.0
llama_new_context_with_model: freq_scale = 1
llama_kv_cache_init: CPU KV buffer size = 384.00 MiB
llama_new_context_with_model: KV self size = 384.00 MiB, K (f16): 192.00 MiB, V (f16): 192.00 MiB
llama_new_context_with_model: CPU output buffer size = 0.12 MiB
llama_new_context_with_model: CPU compute buffer size = 102.01 MiB
llama_new_context_with_model: graph nodes = 1286
llama_new_context_with_model: graph splits = 1
The chat session has started.
User: Say "Hello DotNetPro" in German
Assistant: "Hallo DotNetPro" auf Deutsch.
User: Der Autor dieses Textes ist Robert Mühsig.
Assistant: Der Autor des Textes ist Robert Mühsig.
User: Wer ist der Autor dieses Textes?
Assistant: Der Autor dieses Textes ist Robert Mühsig.
User: Wie ist der Vorname?
Assistant: Der Vorname des Autors ist Robert.

```

Nachrichtenaustausch mit dem Assistenten des Phi-3.1-Modells (Bild 3)

einfacht den Download und das Testen von Sprachmodellen und überwindet so technische Hürden auf elegante Weise.

Alternativen zu LLamaSharp

Ein aktueller Trend in der Chatbot-Entwicklung ist die Verwendung des OpenAI-Web-API als De-facto-Standard. Das zeigt sich auch in den verfügbaren NuGet-Paketen: Mit Azure.AI.OpenAI und OpenAI lassen sich Anwendungen sowohl für den ChatGPT-Endpunkt als auch für den Azure-Dienst entwickeln, da beide denselben API-Standard nutzen. Interessanterweise bieten auch andere KI-Dienste, wie etwa xAI, ein OpenAI-kompatibles API an. Ein weiteres Beispiel ist LM Studio, das ein lokales, OpenAI-kompatibles Web-API bereitstellt und somit einfache Tests ermöglicht. Und es gibt Ollama, ein Open-Source-Projekt, das die Funktionen von llama.cpp über ein OpenAI-kompatibles API zugänglich macht.

Der Vorteil dieser Kompatibilität liegt darin, dass ein OpenAI-API-Client flexibel zwischen verschiedenen Implementierungen wechseln kann. LLamaSharp selbst bietet aktuell jedoch noch kein eigenes Web-API. Ein Nachteil der API-Nutzung ist der zusätzliche Bedarf an einem Server, der die Anfragen verarbeitet – dies kann jedoch auch zur Lastverteilung beitragen. In der LLamaSharp-Community gibt es Überlegungen für ein OpenAI-kompatibles Web-API (siehe Issue #269, [7]), wobei die Umsetzung aktuell noch offen ist.

Zusammengefasst lässt sich sagen, dass LLamaSharp besonders gut in eine .NET-Umgebung passt. Wenn die Anwen-

- [1] llama.cpp bei GitHub,
www.dotnetpro.de/SL2502LLama1
- [2] LLamaSharp bei GitHub,
www.dotnetpro.de/SL2502LLama2
- [3] LLamaSharp, Using CUDA when both CPU and Cuda12 back-ends are present, #456,
www.dotnetpro.de/SL2502LLama3
- [4] Chatbot Arena LLM Leaderboard: Community-driven Evaluation for Best LLM and AI chatbots,
<https://lmarena.ai/?leaderboard>
- [5] kernel-memory bei GitHub,
www.dotnetpro.de/SL2502LLama4
- [6] Retrieval-augmented generation bei Wikipedia,
www.dotnetpro.de/SL2502LLama5
- [7] LLamaSharp, Create HTTP API server and provide API like OAI, #269, www.dotnetpro.de/SL2502LLama6



Robert Mühsig

ist Softwareentwickler und arbeitet vornehmlich mit dem Microsoft-Technologie-Stack. Durch sein Community-Engagement – von User Groups bis hin zu Blogs – ist er seit mehr als 15 Jahren Microsoft MVP.

robert.muehsig@codeinside.eu

dnpCode

A2502LLama



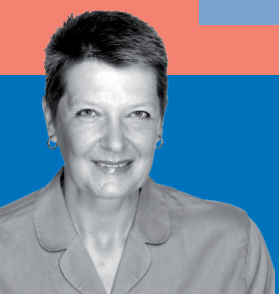
HANDS-ON-WORKSHOPS UND
WEITERBILDUNG FÜR SOFTWARE-
ENTWICKLER UND -ARCHITEKTEN



Hands-on Workshops .NET / C#

C#-Grundlagen	(3 Tage)
C# für Fortgeschrittene	(3 Tage)
ASP.NET Core und Web API	(3 Tage)
Datenbankentwicklung mit dem Entity Framework Core	(3 Tage)
Clean Code und Software Design	(3 Tage)
Async & Await in der Praxis	(2 Tage)
C#, .NET & SQL-Server-Update	(1 Tag)
Fix von .NET Framework zu .NET (CORE)	(1 Tag)
Einstieg in test- und behaviour-driven development (TDD/BDD) mit .NET Core	(2 Tage)
Testen von .NET-Anwendungen	(3 Tage)
Einstieg in MVVM mit WPF	(2 Tage)
Modulare WPF-Anwendungen mit PRISM	(3 Tage)
Einstieg in NoSQL mit MongoDB für .NET-Entwickler	(3 Tage)
ASP.NET Blazor - SPA-Anwendungen mit C# und .NET	(3 Tage)

REMOTE
ODER
INHOUSE



Weitere Informationen auf developer-academy.de
Preise auf Anfrage!

Ihre Ansprechpartnerin: Susanne Herl • +49 731 88005-8835 • susanne.herl@ebnermedia.de

RUST-KURS, TEIL 10

Tests und Fehler



Wie man Fehler abfängt und automatische Tests einbaut.

Die Folge 9 des Rust-Kurses [1] legte den Fokus auf Reference Counting, atomare Datentypen und Memory Ordering. Diese Konzepte ermöglichen die Implementierung von Anwendungen, die mehrere Code-Pfade synchronisieren. So lassen sich Data Races und Race Conditions verhindern. In dieser Folge werden neue Konzepte eingeführt, die die Entwicklung leistungsfähiger Anwendungen unterstützen.

Fehlerbehandlung

Auch bei der Fehlerbehandlung geht Rust einen Weg, der sich von anderen Programmiersprachen unterscheidet. So bietet Rust keine klassischen Exceptions, sondern stellt stattdessen ein Typsystem zur Verfügung, das Entwickler dazu zwingt, Fehler explizit zu behandeln. Dieser Ansatz führt zu sicherem und vorhersehbarem Code. Rust bietet für die Fehlerbehandlung zwei grundlegende Datentypen:

- `Option<T>` wird verwendet, wenn ein Wert vorhanden ist oder aber auch nicht vorhanden ist.
- `Result<T, E>` wird verwendet, wenn eine Operation entweder erfolgreich ist (`Ok(T)`) oder fehlschlägt (`Err(E)`).

Den Datentyp `Option<T>` haben Sie bereits in der zweiten Folge des Rust-Kurses kennengelernt [2]. Er ergänzt Werte einer Enumeration um einen generischen Parameter. Diese Enumeration kann die beiden folgenden Ausprägungen aufweisen (Bild 1):

- `None` wird verwendet, um anzugeben, dass kein Wert vorhanden ist.
- `Some(T)` liefert ein Ergebnis vom Datentyp `<T>` sofern ein Wert vorhanden ist.

Das folgende Beispiel verwendet den Datentyp `Option<T>` als Funktionsergebnis, um bei einer Division durch null keinen Fehler auszulösen.

```
fn div_with_option_datatype(
    x: i32, y: i32) ->
    Option<i32>
{
    if y == 0 {
        None
    } else {
        Some(x / y)
    }
}
```

Wird hier der Wert null als Divisor übergeben, liefert die Funktion `Option<i32>.None` zurück, ansonsten einen Wert vom Typ `Option<i32>.Some`. Mithilfe des `if-let`-Statements können Sie ermitteln, welche Ausprägung der Enumeration zurückgegeben wurde.

Alternativ können Sie `Result<T, E>` verwenden, wenn eine Operation fehlschlagen kann und der Fehler explizit behandelt werden muss. Diese Enumeration ist zum Beispiel für I/O-Operationen nützlich, bei denen Fehler häufig auftreten und teilweise behebbar sind. Bild 2 zeigt die Definition dieser Enumeration in der Standard-Bibliothek von Rust. Listing 1 zeigt, wie Sie `Result<T, E>` beim Zugriff auf das Dateisystem einsetzen können. Darin wird mit `File::open()` eine Datei geöffnet und deren Inhalt mittels `File::read_to_string()` gelesen und in einer String-Variablen gespeichert.

Da beide Funktionen den Datentyp `Result<T, E>` zurückliefern, müssen Sie bei jeder Funktion das `match`-Statement einsetzen, um festzustellen, ob der Funktionsaufruf erfolgreich durchgeführt wurde (`Ok(T)`), oder ob ein Fehler aufgetreten ist (`Err(E)`).

Leider beeinträchtigt dieser restriktive Ansatz die Lesbarkeit des Codes. Daher bietet Rust auch den sogenannten Error-Propagation-Operator, der als Fragezeichen in den Code eingefügt wird. Er lässt sich bei jeder Funktion anwenden, die `Result<T, E>` zurückgibt.

Tritt beim Aufruf einer Funktion ein Fehler auf, leitet der Fragezeichen-Operator den Fehler `Err(E)` an die aufrufende Funktion weiter. Im Erfolgsfall wird dagegen der Wert `Ok(T)`

```
/// The `Option` type. See [the module level documentation](self) for more.
#[derive(Copy, PartialOrd, Eq, Ord, Debug, Hash)]
#[rustc_diagnostic_item = "Option"]
#[lang = "Option"]
#[stable(feature = "rust1", since = "1.0.0")]
75 implementations
pub enum Option<T> {
    /// No value.
    #[lang = "None"]
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    /// Some value of type `T`.
    #[lang = "Some"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

Der Datentyp `Option<T>` (Bild 1)

Listing 1: Result<T, E>

```
fn work_with_errors()
{
    match read_file("file.txt")
    {
        Ok(content) => println!(
            "The content of the file is: {}", content),
        Err(e) => println!("An error occurred: {}", e)
    }
}

fn read_file(file_name: &str)
-> Result<String, io::Error>
{
    match File::open(file_name)
    {
        Ok(mut file) =>
        {
            let mut content = String::new();
            match file.read_to_string(&mut content)
            {
                Ok(_) => Ok(content),
                Err(e) => Err(e)
            }
        },
        Err(e) => Err(e)
    }
}
```

zurückgegeben. Listing 2 zeigt noch einmal den Code aus Listing 1, der per Fragezeichen-Operator vereinfacht wurde. Wie Sie sehen, hat sich die Komplexität des Codes erheblich reduziert, da er keine *match*-Statements mehr enthält. Tritt bei einem Funktionsaufruf ein Fehler auf (*File::open()* oder *File::read_to_string()*), wird dieser vom Fragezeichen-Operator an die aufrufende Funktion weitergeleitet. Diese benötigt jedoch nach wie vor ein *match* Statement, da in dieser Funktion der Fragezeichen-Operator nicht eingesetzt werden kann, da sie kein Ergebnis produziert und somit auch kein *Result<T, E>* zurückliefert.

Dasselbe würde auch auf die Funktion *main()* zutreffen, die ebenfalls kein Ergebnis liefert. Sie können hier jedoch die Funktion *.expect()* aufrufen, die im Fehlerfall zu einer sogenannten Panic führt:

```
fn work_with_errors_panic()
{
    let content = read_file("file.txt")
        .expect("An error occurred");

    println!("The content of the file is: {}",
        content);
}
```

Kommt es in dieser Funktion zu einem Fehler, erscheint die die Fehlermeldung aus Bild 3 auf der Konsole. Es handelt sich dabei um einen Laufzeitfehler, der vom Programm nicht abgefangen wurde. Vorstellen kann man sich eine Rust-Panic wie eine unbehandelte Exception in C#. Wie die Abbildung zeigt, bietet Rust auch das Setzen der Umgebungsvariablen

RUST_BACKTRACE an. Ein Backtrace ist eine detailliertere Variante eines Stack Trace, der die Reihenfolge der Funktionsaufrufe ausgibt (*RUST_BACKTRACE = 1*) beziehungsweise auch weiterführende Informationen über den Zustand des Stacks liefert (*RUST_BACKTRACE = FULL*).

Alternativ können Sie den Backtrace über folgende Kommandozeilenbefehle erzeugen:

- *RUST_BACKTRACE=1*
cargo run
- *RUST_BACKTRACE=FULL*
cargo run

In Bild 4 sehen Sie die Ausgabe des Backtraces, der ►

```
/// `Result` is a type that represents either success (['Ok']) or failure (['Err']).
///
/// See the [module documentation](self) for details.
#[derive(Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
#[must_use = "this `Result` may be an `Err` variant, which should be handled"]
#[rustc_diagnostic_item = "Result"]
#[stable(feature = "rust1", since = "1.0.0")]
34 implementations
pub enum Result<T, E> {
    /// Contains the success value
    #[lang = "Ok"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Ok(#[stable(feature = "rust1", since = "1.0.0")] T),

    /// Contains the error value
    #[lang = "Err"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Err(#[stable(feature = "rust1", since = "1.0.0")] E),
}
```

Result<T, E> mit success- und error-value (Bild 2)

```

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
Running `target/debug/HelloWorld`
Hello World from the Rust programming language!
thread 'main' panicked at src/main.rs:2378:41:
An error occurred: 0s { code: 2, kind: NotFound, message: "No such file or directory" }
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
on  main !2

```

Die Panic-Meldung in einer Funktion ohne Rückgabewert (Bild 3)

durch die gerade gezeigte Funktion `work_with_errors_panic` generiert wird.

Listing 2: Der Fragezeichen-Operator

```

fn work_with_errors_propagation()
{
    match read_file_error_propagation("file.txt")
    {
        Ok(content) => println!(
            "The content of the file is: {}", content),
        Err(e) => println!(
            ("An error occurred: {}", e)
        )
    }
}

fn read_file_error_propagation(file_name: &str)
-> Result<String, io::Error>
{
    let mut file = File::open(file_name)?;
    let mut content = String::new();
    file.read_to_string(&mut content)?;

    Ok(content)
}

```

Panic können Sie auch selbst in Ihrem Code auslösen, indem Sie das Makro `panic!()` verwenden. Das klappt so:

```

fn work_with_panic_macro(
    dividend: i32, divisor: i32) -> i32
{
    if divisor == 0
    {
        panic!("Divide by zero!");
    }
    dividend / divisor
}

```

Über die Funktion `std::panic::catch_unwind()` haben Sie zudem die Möglichkeit, Panic-Meldungen abzufangen, zu behandeln und so verhindern, dass ein Programm vollständig abstürzt. Die folgenden Zeilen zeigen den Einsatz dieser Funktion.

```

fn work_with_panic_catch_unwind()
{
    let result = panic::catch_unwind(||
        work_with_panic_macro(10, 0));

    match result
    {
        Ok(r) => println!("Result: {}", r),
        Err(_) => println!("We have caught a panic!")
    }
}

```

```

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.06s
Running `target/debug/HelloWorld`
Hello World from the Rust programming language!
thread 'main' panicked at src/main.rs:2378:41:
An error occurred: 0s { code: 2, kind: NotFound, message: "No such file or directory" }
stack backtrace:
0: rust_begin_unwind
   at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/std/src/panicking.rs:665:5
1: core::panicking::panic_fmt
   at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs:74:14
2: core::result::unwrap_failed
   at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/result.rs:1679:5
3: core::result::Result<T,E>::expect
   at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/result.rs:1059:23
4: HelloWorld::work_with_errors_panic
   at ./src/main.rs:2378:19
5: HelloWorld::main
   at ./src/main.rs:132:5
6: core::ops::function::FnOnce::call_once
   at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
on  main !2

```

Backtrace des Fehlers in der Funktion `work_with_errors_panic` (Bild 4)


```

}
println!("This code is always executed,
because the panic was caught.");
}

```

Im Code wird der Funktion `catch_unwind()` einfach eine Closure übergeben und ausgeführt. Als Ergebnis wird wieder der Datentyp `Result<T, E>` zurückgeliefert, auf dessen konkrete Ausprägung man über das `match`-Statement zugreifen kann.

Rust erlaubt es zudem, eigene Fehlertypen zu definieren, die man in Verbindung mit `Result<T, E>` verwenden kann. Dies ist immer dann nützlich, wenn eine Anwendung spezielle Fehlertypen abbilden soll. Listing 3 zeigt ein Beispiel dafür.

Wie Sie dem Listing entnehmen können, wird eine Struktur `DivideByZeroError` definiert, die drei öffentliche Variablen enthält. Innerhalb der Funktion `divide()` wird bei einer Division durch null nun eine neue Instanz dieser Struktur zurückgeliefert, die anschließend per `match`-Statement auf der Konsole ausgegeben wird.

Die Fehlerbehandlung von Rust ist durchaus leistungsfähig. Nicht enthalten ist in Rust ein Structured Exception Handling (SEH), das Programmiersprachen wie C++ und C# in Form von `try/catch/finally`-Blöcken anbieten. Der Grund dafür liegt darin, dass Rust aus der Ecke der funktionalen Programmierung stammt, bei denen SEH nicht zu den grundlegenden Designprinzipien gehört.

In Rust sind Fehler einfach Teil der Funktionsergebnisse (und somit Teil der Funktionssignatur) und werden durch den Datentyp `Result<T, E>` explizit ausgewiesen. Das zwingt Sie als Entwickler dazu, alle potenziellen Fehler bereits zur Kompilierzeit zu berücksichtigen und zu behandeln. Dieser Ansatz reduziert das Risiko eines fehlerhaften Verhaltens der Anwendung.

Automatisierte Tests

Das automatisierte Testen von Code ist wesentlicher Bestandteil der modernen Softwareentwicklung, um die Zuverlässigkeit und Stabilität von Anwendungen gewährleisten zu können. Rust bietet folgende Tests an:

- Unit-Tests: Isolierte Tests einzelner Funktionen oder Module.
- Integrationstests: Überprüfung des Zusammenspiels mehrerer Komponenten oder Module.

Für das Schreiben von Unit-Tests bietet Rust die folgenden Erweiterungen an:

- `#[test]`: Dieses Attribut markiert eine Funktion als Testfunktion.
- `#[cfg(test)]`: Dieses Attribut kompiliert ein markiertes Modul nur dann, wenn eine Testausführung über den Befehl `cargo test` durchgeführt wird.
- `assert_eq!()`: Mit diesem Makro prüfen Sie in einer Testfunktion einen erwarteten Wert.
- `cargo test`: Dieser Befehl führt alle implementierten Testfunktionen auf der Kommandozeile aus.

Listing 4 zeigt zwei einfache Unit-Tests, die den Primzahlen-generator aus einem früheren Teil des Rust-Kurses testen. Für

die Unit-Tests wird ein neues Modul namens `tests` definiert. Mit dem Attribut `#[cfg(test)]` wird festgelegt, dass dieses Modul nur dann kompiliert wird, wenn die Tests via `cargo test` ausgeführt werden.

Damit Sie auf den Primzahlen-Generator zugreifen können (der außerhalb des Test-Moduls implementiert wurde), müssen Sie diesen über den Befehl `use super::*` importieren. Der Befehl `cargo test` führt die mit `#[test]` markierten Testfunktionen automatisch aus.

Erst danach werden die Primzahlen generiert, und das Makro `assert_eq!()` prüft, ob die zurückgelieferten Primzahlen den im Test hinterlegten Werten entsprechen. Bild 5 zeigt die Ausgabe der beiden Unit-Tests: Der erste wird erfolgreich ausgeführt, da alle aufgestellten Behauptungen des Makros `assert_eq!()` zutreffen. Der zweite Unit-Test schlägt jedoch ►

Listing 3: Benutzerdefinierte Fehlertypen

```

#[derive(Debug)]
pub struct DivideByZeroError
{
    pub error_message: String,
    pub dividend: i32,
    pub divisor: i32
}

fn divide(dividend: i32, divisor: i32) ->
    Result<i32, DivideByZeroError>
{
    if divisor == 0
    {
        return Err(DivideByZeroError{
            error_message: "Divide by zero!",
            .to_string(),
            dividend: dividend,
            divisor: divisor });
    }

    Ok(dividend / divisor)
}

fn work_with_custom_error_types()
{
    match divide(10, 0)
    {
        Ok(result) => println!("Result: {}", result),
        Err(e) =>
        {
            println!("An error occurred: {}, {} / {}",
                e.error_message, e.dividend, e.divisor);
            println!("{}", e);
        }
    }
}

```


● Listing 4: Zwei einfache Unit-Tests

```
#[cfg(test)]
mod tests
{
    use super::*;

    #[test]
    fn test_prime_number_generator1()
    {
        // Generate the prime numbers from 0 to 10
        let prime_numbers =
            PrimeNumberGenerator::new(10);
        let result: Vec<i32> =
            prime_numbers.into_iter().collect();

        // Test if we got back 4 prime numbers in
        // the range from 0 to 10
        assert_eq!(result.len(), 4);

        // Test the individual prime numbers
        assert_eq!(result[0], 2);
        assert_eq!(result[1], 3);

        assert_eq!(result[2], 5);
        assert_eq!(result[3], 7);
    }

    #[test]
    fn test_prime_number_generator2()
    {
        // Generate the prime numbers from
        // 0 to 100
        let prime_numbers =
            PrimeNumberGenerator::new(100);
        let result: Vec<i32> =
            prime_numbers.into_iter().collect();

        // Test if we got back 4 prime numbers in
        // the range from 0 to 100
        // This test will FAIL!
        assert_eq!(result.len(), 4);
    }
}
```

fehl, da die vom Test vorgegebene Annahme (`assert_eq!(result.len(), 4)`) nicht den Tatsachen entspricht. Im untersuchten Bereich von null bis 100 gibt es nicht nur vier, sondern 25 Primzahlen.

Neben dem Makro `assert_eq!()` bietet Rust zudem noch das Makro `assert_ne!()` an, um zu überprüfen ob zwei Werte ungleich sind.

Mit Integrationstests überprüfen Sie das Zusammenspiel mehrerer Komponenten. Als zu testendes Beispiel dient die

in der vierten Folge dieses Kurses [3] implementierte Bibliothek *storage_engine_library*. Die Bibliothek kann fiktive Data Pages und Index Pages lesen und als Ergebnis zurückgeben. Um Integrationstests zu implementieren, müssen Sie auf Ebene des Projektstammverzeichnisses den Unterordner *tests* anlegen, siehe [Bild 6](#).

Dieser Unterordner nimmt die Code-Dateien auf, welche die Integrationstests implementieren. [Listing 5](#) zeigt einen einfachen Integrationstest für die Bibliothek. Wie Sie anhand des

```
running 2 tests
test tests::test_prime_number_generator1 ... ok
test tests::test_prime_number_generator2 ... FAILED

failures:

---- tests::test_prime_number_generator2 stdout ----
thread 'tests::test_prime_number_generator2' panicked at src/main.rs:2472:9:
assertion `left == right` failed
  left: 25
 right: 4
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::test_prime_number_generator2

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

error: test failed, to rerun pass `--bin HelloWorld`
~ /d/G/KlausAschenbrenner/rust-tutorials/HelloWorld on main !2
```

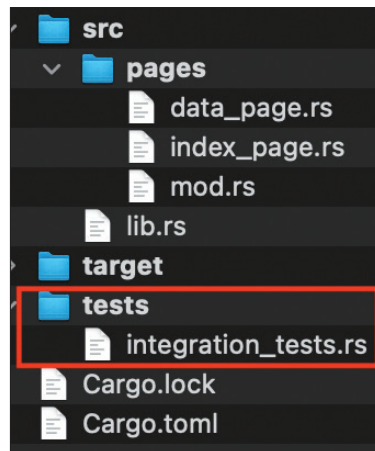
Automatische Tests: Der erste ist grün, der zweite Test scheitert ([Bild 5](#))

Codes erkennen, sind Integrationstests ähnlich wie Unit-Tests aufgebaut. Der Unterschied: Die Bibliothek wird mithilfe von realistischem Code getestet. Auch Integrationstests werden mit `cargo test` ausgeführt. **Bild 7** zeigt das Ergebnis.

Unsafe Code

Ein zentrales Prinzip von Rust ist die integrierte Sicherstellung von Speicher- und Thread-Sicherheit. Der Borrow-Checker sorgt dafür, dass Sie nicht gleichzeitig lesend und schreibend auf denselben Speicherbereich zugreifen. Das ist besonders in Multi-Threading-Anwendungen wichtig.

Das Ownership-Konzept garantiert, dass jeder auf dem Heap allokierte Speicherbereich immer nur einen Besitzer hat. Dadurch vermeidet man unter anderem Double-Free-Operationen. Diese treten auf, wenn Speicherbereiche auf dem Heap doppelt freigegeben wer-



Integrationstests vorbereiten (Bild 6)

den, woraus ein fehlerhaftes Verhalten der Anwendung resultieren kann.

In der vorherigen Folge [1] haben Sie gesehen, dass der Rust-Compiler oft Data Races und Race Conditions verhindert. Das erreicht er, indem er das Kompilieren des problematischen Codes verweigert.

Manchmal muss man die strengen Sicherheitsgarantien des Rust-Compilers bewusst umgehen. Dies geschieht beispielsweise, um systemnahe Operationen auszuführen. Auch der Umgang mit weniger sicherem externem Code, wie C/C++-Bibliotheken, erfordert dies. Dafür bietet Rust spezielle *unsafe*-Code-Blöcke an.

Das Schlüsselwort *unsafe* ermöglicht in Rust Operationen, die der Compiler nicht kontrolliert. Der Compiler kann daher in *unsafe*-Blöcken keine Speicher- und Thread-Sicherheit garantieren. Diese Verantwortung liegt dann beim Entwickler. Das folgende Beispiel zeigt die Verwendung von *unsafe*:

```
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running tests/integration_tests.rs (target/debug/deps/integration_tests-339d5306ef165c06)

running 1 test
test test_storage_engine ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests storage_engine_library
running 0 tests
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

~ /d/G/K/rust-tutorials/L/storage_engine_library on main !2 ?1
```

Ergebnis des Integrationstests (Bild 7)

Listing 5: Integrationstest

```
#[test]
fn test_storage_engine()
{
    // Read some pages
    let data_page =
        storage_engine_library::read_data_page();

    let index_page =
        storage_engine_library::read_index_page();

    // Test the data page values
    assert_eq!(data_page.page_id, 80);
    assert_eq!(data_page.file_id, 2);
    assert_eq!(data_page.record_count, 19);

    // Test the index page values
    assert_eq!(index_page.page_id, 80);
    assert_eq!(index_page.file_id, 2);
    assert_eq!(index_page.record_count, 19);
    assert_eq!(index_page.index_level, 3);
}
```

```

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.23s
Running `target/debug/HelloWorld`
Hello World from the Rust programming language!
[1] 28687 segmentation fault cargo run
~/.d/G/KlausAschenbrenner/rust-tutorials/HelloWorld on main !2 ?1

```

Absturz aufgrund
von fehlerhaftem
Code in einem
unsafe-Block
(Bild 8)

```

// Null-Pointer mit unsafe
fn work_with_unsafe_code_crash()
{
    // This code dereferences a null pointer and
    // crashes the program
    unsafe
    {
        *(0 as *mut i32) = 42;
    }
}

```

Hier wird in einem *unsafe*-Block per Stern-Operator ein Null-Pointer dereferenziert und diesem ungültigen Speicherbereich der Wert 42 zugewiesen. Wird diese Funktion ausgeführt, stürzt das Programm mit einem *Segmentation Fault* ab, weil das Betriebssystem den Zugriff auf die Hauptspeicheradresse null verhindert (Bild 8).

In diesem Beispiel wurde ein Raw Pointer erzeugt. Dieser ermöglicht den direkten Lese- und Schreibzugriff auf eine Hauptspeicheradresse. Sie dürfen einen Raw Pointer in Rust nur innerhalb eines *unsafe*-Blocks erzeugen. Er kann daher auf eine ungültige Speicheradresse zeigen.

Es ist wichtig zu beachten, dass ein *unsafe*-Block die Geschwindigkeit oder Effizienz des Codes nicht beeinflusst. Er bedeutet lediglich, dass Sie als Entwickler vorübergehend die Verantwortung für die Speicher- und Thread-Sicherheit übernehmen.

Ein sinnvoller Einsatz von *unsafe*-Blöcken in Rust ist das Implementieren eines Spinlocks [4]. Ein Spinlock ist – ähnlich wie ein Mutex – ein Objekt, das das Synchronisieren mehrerer Threads erlaubt.

Ein Mutex sorgt dafür, dass immer nur ein Thread eine gemeinsam benutzte Ressource verwenden kann. Versucht ein Programmteil, einen bereits gesperrten Mutex zu benutzen, muss er warten. Das Betriebssystem hält diesen Programmteil so lange an (Status *Suspended*), bis der Mutex wieder frei ist.

Ein Mutex ist nur dann sinnvoll, wenn der kritische Abschnitt, der synchronisiert werden muss, länger dauert, da sich ansonsten der Overhead durch das Kontext-Switching durch das Betriebssystem nicht bezahlt macht.

Bei einem Spinlock wartet ein Thread in einer Schleife so lange, bis die Sperre freigegeben wird, ohne vom Betriebssystem in einen *Suspended*-Status versetzt zu werden. Ein Spinlock ist daher dann sinnvoll, wenn der kritische Abschnitt nur sehr kurze Wartezeiten verursacht.

Bei diesem Code-Beispiel möchte ich mich an einem Blogpost von Mara Bos [5] orientieren, das ebenfalls Schritt für Schritt zeigt, wie ein Spinlock in Rust mithilfe von *unsafe*-Blöcken implementiert werden kann. Auf Implementierungsebene ist ein Spinlock nichts anderes als eine Struktur mit einer atomaren Boolean-Variablen, die anzeigt, ob der Spinlock gesperrt ist oder nicht. Der implementierte Spinlock enthält die Daten, die er schützt (wie ein Mutex). Listing 6 zeigt die Definition der Struktur und deren Implementierungsblock.

Listing 6: Spinlock-Implementierung

```

pub struct SpinlockWithDataAndGuard<T>
{
    locked: AtomicBool,
    data: UnsafeCell<T>
}

impl <T> SpinlockWithDataAndGuard<T>
{
    // Create a new Spinlock
    pub fn new(data: T) ->
        SpinlockWithDataAndGuard<T>
    {
        // Return the new spinlock
        SpinlockWithDataAndGuard
        {
            locked: AtomicBool::new(false),
            data: UnsafeCell::new(data)
        }
    }

    // Lock the Spinlock
    pub fn lock(&self) -> SpinlockGuard<T>
    {
        // Try to acquire the spinlock
        while self.locked.swap(
            true, Ordering::Acquire)
        {
            std::hint::spin_loop();
        }

        // Return the Spinlock Guard
        SpinlockGuard { spinlock: self }
    }
}

```

Wie Sie im Listing sehen, wird in der Struktur *SpinlockWithDataAndGuard*<T> über das Flag *locked* gespeichert, ob der Spinlock aktuell gesperrt ist oder nicht. Die Variable *data* (Typ *UnsafeCell*<T>) sichert die durch den Spinlock geschützten Daten. Der Datentyp *UnsafeCell*<T> stellt einen Raw Pointer (Typ **mut T*) auf die eingebetteten Daten zur Verfügung, der nicht den Regeln des Borrow-Checkers folgt. Diesen Raw Pointer können Sie daher zum direkten Lesen und Schreiben der eingebetteten Daten verwenden.

Aufgabe der Funktion *lock()* ist es, den Spinlock zu sperren. Dazu wird in einer *while*-Schleife versucht, die interne atomare Boolean-Variable auf *true* zu setzen. Zusätzlich wird dem Prozessor über einen *Spinloop Hint* mitgeteilt, dass sich der aktuelle Thread in einer Schleife befindet, der sogenannten Spinlock Wait Loop. Dadurch kann der Prozessor Optimierungen durchführen.

Sobald der Spinlock erfolgreich gesperrt wurde, liefert die Funktion *lock()* eine Instanz der Struktur *SpinlockGuard* zurück. Dieses Guard-Objekt verhält sich wie ein *MutexGuard*, das heißt, dass Sie darüber auf die geschützten Daten des Spinlocks zugreifen können. Verlässt der Spinlock-Guard seinen Gültigkeitsbereich, wird auch der Spinlock freigegeben, damit ein anderer Thread diesen erfolgreich sperren kann. Listing 7 zeigt die Implementierung der Struktur *SpinlockGuard*, welche die beiden Traits *Deref* und *DerefMut* implementiert.

Die Funktionen *deref()* und *deref_mut()* ruft man auf, wenn der Spinlock-Guard zum Lesen oder Schreiben mit dem Stern-Operator dereferenziert wird. In der Implementierung beider Funktionen liefert ein *unsafe*-Block einen Raw Pointer durch den Aufruf von *UnsafeCell::get()*. In der Funktion *deref_mut()*

wird dieser Raw Pointer zusätzlich zu einer veränderlichen Referenz konvertiert (*&mut*).

Ein Konstruktor fehlt beim Spinlock-Guard, da eine Instanz dieser Struktur nur beim erfolgreichen Sperren des Spinlocks instanziiert werden kann. Dadurch ermöglicht das Dereferenzieren des *Guard*-Objekts einen Thread-sicheren Zugriff auf die geschützten Daten des Spinlocks. Kein weiteres *Guard*-Objekt kann gleichzeitig existieren.

Außerdem in Listing 7 zu sehen: Der Spinlock nutzt zwar die Funktion *lock()*, hat jedoch keine Funktion *unlock()*. Das Entsperren des Spinlocks erfolgt automatisch, wenn der Spinlock-Guard seinen Gültigkeitsbereich verlässt. Dazu wurde auf dieser Struktur die Trait *Drop* implementiert, siehe Listing 8.

Nachdem Spinlock und Guard implementiert sind, nutzen Sie den Spinlock wie einen Mutex. Der Unterschied besteht darin, dass Sie aktiv auf der CPU warten, bis der Spinlock gesperrt werden kann.

Listing 9 zeigt die Anwendung des Spinlocks zur Thread-Synchronisierung. Im ersten Schritt erzeugen Sie einen neuen Spinlock. Dieser schützt einen Integer-Wert, der mit null initialisiert ist. Anschließend starten Sie in einem Thread-Scope zehn Worker-Threads. Diese erhöhen den geschützten Integer-Wert jeweils um 1. Bild 9 zeigt die Ausgabe des Programms.

Um auf den Integer-Wert zugreifen zu können, wird der Spinlock zunächst gesperrt (*lock()*). Diese Funktion wird so lange aktiv auf der CPU blockiert, bis die Sperre des Spinlocks erfolgreich war. Als Ergebnis wird das *Guard*-Objekt zurückgeliefert. Danach kann das *Guard*-Objekt mithilfe des Stern-Operators dereferenziert und auf die geschützten Daten des Spinlocks zugegriffen werden. ►

Listing 7: SpinlockGuard-Implementierung

```
pub struct SpinlockGuard<'a, T>
{
    spinlock: &'a SpinlockWithDataAndGuard<T>
}

// Implement the Deref Trait on the
// Spinlock Guard
impl<T> Deref for SpinlockGuard<'_, T>
{
    type Target = T;

    fn deref(&self) -> &T
    {
        unsafe
        {
            // Return a raw pointer to the protected
            // data of the spinlock
            println!("Read access to the spinlock: {:?}",
                std::thread::current().id());
            &*self.spinlock.data.get()
        }
    }
}

// Implement the DerefMut Mutable Trait on
// the Spinlock Guard
impl<T> DerefMut for SpinlockGuard<'_, T>
{
    fn deref_mut(&mut self) -> &mut T
    {
        unsafe
        {
            // Return a mutable raw pointer to the
            // protected data of the spinlock
            println!("Write access to the spinlock: {:?}",
                std::thread::current().id());
            &mut *self.spinlock.data.get()
        }
    }
}
```

Da hier ein schreibender Zugriff erfolgt (`*guard += 1`) wird intern die Funktion `deref_mut()` aufgerufen, die über die Trait `DerefMut` implementiert wurde.

Diese Funktion liefert in einem `unsafe`-Block einen Raw Pointer auf den Hauptspeicherbereich zurück (Datentyp `UnsafeCell<T>`). Da der aktuelle Thread zu diesem Zeitpunkt der einzige mögliche Besitzer des `Guard`-Objekts ist, ist der Zugriff über diesen Raw Pointer Thread-sicher, da kein anderer Thread zum gleichen Zeitpunkt den Spinlock gesperrt haben kann.

Verlässt das `Guard`-Objekt seinen Gültigkeitsbereich, ruft Rust automatisch die Funktion `drop()` auf. Diese Funktion entspermt den Spinlock wieder, indem in der atomaren Boolean-Variablen `locked` der Wert `false` gespeichert wird.

Das Ergebnis: Nun kann ein anderer Thread den Spinlock nutzen und bekommt exklusiven Zugriff auf die geschützten Daten.

Foreign Function Interface

Das Foreign Function Interface (FFI) ist eine Schnittstelle, die es erlaubt, auf Funktionen zuzugreifen, welche in anderen Programmiersprachen geschrieben wurden.

Das Beispiel in [Listing 10](#) zeigt, wie Sie in C verfasste Funktionen aufrufen und wie das Speicher-Management von Objekten funktioniert, die durch den C-Code auf dem Heap allokiert wurden. Darin werden im C-Code zwei Funktionen definiert: `add()` addiert zwei Integer-Werte und gibt das Ergebnis zurück. Die Funktion `say_hello_world()` erwartet einen nullterminierten String als Parameter und liefert einen auf dem Heap allokierten nullterminierten String als Ergebnis zurück.

Listing 8: Die Trait Drop

```
// Implements the Drop Trait on the Spinlock-
// Guard. It unlocks the spinlock when the
// Spinlock-Guard goes out of scope.
impl<T> Drop for SpinlockGuard<'_, T>
{
    fn drop(&mut self)
    {
        // Unlock the spinlock
        println!("The spinlock is unlocked: {:?}",
            std::thread::current().id());
        println!("");
        self.spinlock.locked.store(
            false, Ordering::Release);
    }
}
```

Damit Sie diese beiden Funktionen in Ihrem Rust Code verwenden können, müssen Sie deren Code zunächst mit einem C-Compiler kompilieren. Dank der von Rust angebotenen Build-Skripte lässt sich dieser Schritt im Zuge des Befehls `cargo build` durchführen. Build-Skripte werden vor dem Kompilieren des eigentlichen Rust-Codes ausgeführt. Den vorab zu kompilierenden Code legen Sie in einer Datei namens `build.rs` in dem Ordner ab, in dem auch die Datei `Cargo.toml` liegt. In den folgenden Code-Zeilen finden Sie ein einfaches Build-Skript, das den C-Code in der Datei `src/interop.c` mithilfe der

Listing 9: Spinlock verwenden

```
fn work_with_spinlock_guard()
{
    let spinlock =
        SpinlockWithDataAndGuard::new(0);

    thread::scope(|s|
    {
        let mut handles = vec![];

        // Create 10 new threads
        for _ in 0..10
        {
            let handle = s.spawn(||
            {
                let mut guard = spinlock.lock();
                println!("The spinlock is locked: {:?}",
                    thread::current().id());

                thread::sleep(Duration::from_secs(1));

                // Change the protected data
                *guard += 1; // Calls "deref_mut"
                println!("Current value of the protected
                    integer: {:?}", *guard); // Calls "deref"
            });
            // The guard is automatically released here,
            // and the spinlock is unlocked

            handles.push(handle);
        }

        // Wait for all threads until they are finished
        for handle in handles
        {
            handle.join().unwrap();
        }
    });

    println!("Finished!");
}
```



```

The spinlock is locked: ThreadId(3)
Write access to the spinlock: ThreadId(3)
Read access to the spinlock: ThreadId(3)
Current value of the protected integer: 8
The spinlock is unlocked: ThreadId(3)

The spinlock is locked: ThreadId(7)
Write access to the spinlock: ThreadId(7)
Read access to the spinlock: ThreadId(7)
Current value of the protected integer: 9
The spinlock is unlocked: ThreadId(7)

The spinlock is locked: ThreadId(4)
Write access to the spinlock: ThreadId(4)
Read access to the spinlock: ThreadId(4)
Current value of the protected integer: 10
The spinlock is unlocked: ThreadId(4)

Finished!

```

Arbeiten mit Spinlock (Bild 9)

Crate `cc` [6] kompiliert. Der C-Compiler muss dafür separat installiert worden sein.

```

// Build-Skript für C-Code
fn main()
{
    cc::Build::new()
        .file("src/interop.c")
        .compile("interop");

    println!("cargo::rerun-if-changed=src/interop.c");
}

```

Der C-Code in `src/interop.c` wird kompiliert und als statische Bibliothek abgelegt, die mit der ausführbaren Datei des Rust-Programms verlinkt wird. Um C-Funktionen aus Rust heraus aufzurufen, müssen Sie die Signaturen der C-Funktionen mithilfe von `extern "C"` bekannt machen. Damit wird sichergestellt, dass der Rust-Compiler beim Aufruf dieser Funktionen die C Calling Convention [7] verwendet.

```

use std::ffi::{CStr, CString};
use std::os::raw::{c_char, c_void};

```

```

extern "C"
{
    fn add(a: i32, b: i32) -> i32;
    fn say_hello_world(str: *const c_char) -> *mut c_char;
    fn free(ptr: *mut c_void);
}

```

Hier werden die Signaturen der Funktionen `add()` und `say_hello_world()` definiert. Zusätzlich wurde die Signatur der Funktion `free()` aufgenommen, die Bestandteil der C-Standard-Bibliothek ist, die ebenfalls mit jedem Rust-Programm verlinkt wird. Wie Sie im Code sehen, wurden bei den Signaturen von `say_hello_world()` und `free()` Datentypen verwendet, die Rust so nicht kennt (`c_char` und `c_void`). Im Modul `std::os::raw` stellt Rust spezifische Datentypen zur Verfügung, die auf die zugehörigen C-Datentypen abgebildet werden, um beide Programmiersprachen miteinander verbinden zu können.

Nun zum Aufruf der C-Funktionen innerhalb des Rust-Codes. Da `add()` lediglich mit Integer-Werten arbeitet, ergeben sich dabei keine Probleme. Interessanter ist der Aufruf von `say_hello_world()`, da hier Strings ausgetauscht werden und die C-Funktion Speicher auf dem Heap (über die Funktion `malloc()`) allokiert. Diesen Speicherbereich muss dann die C-Funktion `free()` wieder freigeben, damit kein Memory-Leak entsteht.

Listing 11 zeigt, wie dieser Funktionsaufruf implementiert wird. Der Austausch von Strings zwischen C und Rust gestaltet sich schwierig. In C sind Strings immer nullterminiert, während Strings in Rust eine feste Länge als Teil ihrer Datenstruktur haben. Deshalb muss man Strings konvertieren, wenn man sie austauscht.

Zuerst legt man in Rust einen nullterminierten String mithilfe des Datentyps `CString` an. Dieser wird intern über ein `Box`-Objekt auf dem Heap gespeichert. Da `CString` ein Rust-Datentyp ist, gelten die Regeln des Borrow-Checkers sowie des Ownership-Systems. Das bedeutet, dass `CString` auf dem Heap freigegeben wird, sobald die Variable ihren Gültigkeitsbereich verlässt.

Im nächsten Schritt greift man innerhalb eines `unsafe`-Blocks auf die passende C-Funktion zu. Die Funktion `as_ptr()` liefert dabei einen Raw Pointer auf die Startadresse des Strings, den man der C-Funktion als Parameter übergibt. ►

Listing 10: C-Code einbinden

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// A simple add function
int add (int x, int y)
{
    return (x + y);
}

// Function that returns a heap-allocated string
char* say_hello_world(const char *name)
{
    const char *str = "Hello World, ";
    char *result = (char *)malloc(
        strlen(str) + strlen(name) + 1);
    strcpy(result, str);
    strcat(result, name);
    return result;
}

```

```

Compiling c-interop v0.1.0 (/Users/klaus/dev/GitHub/KlausAschenbrenner/rust-tutorials/c-interop)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.23s
Running `target/debug/c-interop`
The sum of 5 and 7 is 12
Hello World, Klaus Aschenbrenner
c-interop(12143,0x1f6847240) malloc: Double free of object 0x153e05e20
c-interop(12143,0x1f6847240) malloc: *** set a breakpoint in malloc_error_break to debug
[1] 12143 abort      cargo run
~/.d/G/K/rust-tutorials/c-interop on main ?2

```

Eine **Double-Free-Operation** führte zu diesem Laufzeitfehler (Bild 10)

Schließlich gibt die C-Funktion einen Raw Pointer vom Typ `*mut c_char` zurück, der auf den Beginn des Strings zeigt.

Der zurückgegebene Raw Pointer wird mit `CStr::from_ptr()` in einen „Borrowed C String“ umgewandelt. Dies ist eine ausgeborgte, lesende Referenz (`&CStr`) auf den nullterminierten String. Anschließend wandelt `to_str()` diese Referenz in einen Rust-String und gibt ihn auf der Konsole aus.

Damit kein Memory-Leak entsteht, wird am Ende des `unsafe`-Blocks der Speicherblock auf dem Heap (der in der C-Funktion mit `malloc()` allokiert wurde) mit der C-Standard-Funktion `free()` wieder freigegeben. Würde man `free()` mehrere Male aufrufen, wäre das eine Double-Free-Operation, die zu einem Laufzeitfehler führt (Bild 10).

Listing 11: Strings zwischen C und Rust austauschen

```

fn work_with_strings()
{
    let name = CString::new(
        "Klaus Aschenbrenner").expect(
        "Error: embedded null characters!");

    unsafe
    {
        // Call the C function with a raw pointer to
        // the start of the string (as_ptr())
        let char_ptr = say_hello_world(
            name.as_ptr());

        // Convert the raw pointer to a
        // representation of a borrowed C string
        let c_str = CStr::from_ptr(char_ptr);

        // Convert the borrowed C string to a
        // Rust string
        let rust_str = c_str.to_str().expect("Error");
        println!("{}", rust_str);

        // Free the memory allocated by the
        // previously called C function
        free(char_ptr as *mut c_void);
    }
}

```

Wie Sie sehen, ist die Zusammenarbeit von Rust mit C-Code komplex. Man muss genau wissen, welche C-Funktionen Speicher allokieren und welche Code-Pfade für ihre Freigabe verantwortlich sind. Dieses Beispiel zeigt die Vorteile einer sicheren Programmiersprache wie Rust. Rust erkennt und verhindert speicherrelevante Fehler bereits beim Kompilieren.

Fazit

In diesem Artikel haben Sie fortgeschrittene Programmier-techniken in Rust entdeckt. Jetzt wissen Sie, wie Sie Fehler behandeln und die Korrektheit Ihrer Programme automatisch mit Unit-Tests überprüfen.

Der Abschnitt über `Unsafe`-Codeblöcke zeigte, wie man die strengen Sicherheitsvorgaben von Rust für kurze Abschnitte durchbrechen kann. Dies ist nützlich, um zum Beispiel einen Spinlock zur Thread-Synchronisation zu implementieren.

Zum Schluss haben Sie gelernt, wie man C-Code über das Foreign Function Interface aufruft. Dabei haben Sie auch erfahren, welche Probleme der Aufruf von C-Bibliotheken aus Rust heraus verursachen kann. ■

- [1] Klaus Aschenbrenner, *Threadsync, Rust-Kurs, Teil 9*, dotnetpro 1/2025, Seite 128 ff., www.dotnetpro.de/A2501Rust
- [2] Klaus Aschenbrenner, *Kontrollstrukturen, Rust-Kurs, Teil 2*, dotnetpro 4/2024, Seite 110 ff., www.dotnetpro.de/A2404Rust
- [3] Klaus Aschenbrenner, *Structs, Module, Crates*, Rust-Kurs, Teil 4, dotnetpro 6/2024, Seite 125 ff., www.dotnetpro.de/A2406Rust
- [4] Spinlocks, <https://en.wikipedia.org/wiki/Spinlock>
- [5] Building Spinlocks, www.dotnetpro.de/SL2502Rust1
- [6] Die Crate cc, <https://crates.io/crates/cc>
- [7] C Calling Convention, www.dotnetpro.de/SL2502Rust2



Klaus Aschenbrenner

berät Unternehmen in Europa beim Einsatz des Microsoft SQL Server und beschäftigt sich mit Windows-Programmierung und .NET. Zweimal zeichnete Microsoft ihn für sein Engagement als Microsoft MVP aus. Er ist zu erreichen über www.SQLpassion.at.

dnpCode

A2502Rust



HANDS-ON-WORKSHOPS UND
WEITERBILDUNG FÜR SOFTWARE-
ENTWICKLER UND -ARCHITEKTEN



Hands-on Workshops C++ und Rust

NEU! Rust für Einsteiger (3 Tage)

C++ für Fortgeschrittene (3 Tage)

C++-Templates für
das Daily Business (Basics) (1 Tag)

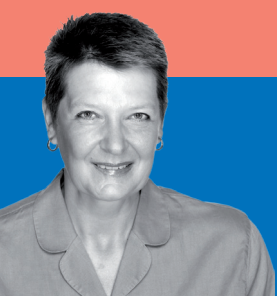
C++-Templates für
Fortgeschrittene (1 Tag)

C++-Templates -
Von A bis Z (2 Tage)

Design Patterns in C++ (2 Tage)

Interface Design mit C++ (2 Tage)

REMOTE
ODER
INHOUSE



Weitere Informationen auf developer-academy.de
Preise auf Anfrage!

Ihre Ansprechpartnerin: Susanne Herl • +49 731 88005-8835 • susanne.herl@ebnermedia.de

MONOGAME FÜR EINSTEIGER, TEIL 8.1

Heute wird gespielt!

Ursprünglich sollte diese Serie mit dem achten Teil enden. Doch manchmal ändern sich Pläne, das kennen Sie ja nun schon. Nennen wir diesen Teil daher 8.1 ...

Anders als in der Anwendungsentwicklung gibt es in der Spieleentwicklung irgendwann den Zeitpunkt, ab dem es einfach Spaß macht, auf [F5] zu drücken. Für einen relativ langen Zeitraum arbeiten Sie allerdings erst einmal darauf hin, tatsächlich ein Spiel zu entwickeln. Doch irgendwann stellen Sie sich auch die Frage, die ich im letzten Artikel [1] bereits gestellt habe: „Ab wann ist (m)ein Spiel eigentlich ein Spiel?“ – Und? Haben Sie darauf schon eine Antwort gefunden? In diesem Artikel wird es unter anderem um genau diese eine Frage gehen.

Doch zunächst einmal wieder hallo und herzlich willkommen zum bereits achten Teil dieser kleinen Serie, die Sie mitnimmt auf eine Reise, auf der Sie lernen, wie Sie Spiele „from scratch“ entwickeln können. Und der Titel hat es bereits geteasert: Heute wird ... gespielt!

Zeit für einen Teaser

Nach inzwischen sieben Artikeln haben Sie es sich verdient, ausnahmsweise direkt zum Start des neuen Artikels anzuschauen, wo wir demnächst landen werden (Bild 1). Und würden Sie sagen, dass es sich dabei auch um ein Spiel von itch.io handeln könnte? Wäre ja schön, wenn!

Dazu noch ein Hinweis: Entsprechend den Anforderungen im Spiel selbst kommt es auch zu Änderungen in der Map. Die erläutere ich aber nicht im Einzelnen, da Sie zum einen wissen, wie bestimmte Ideen darin umgesetzt werden können, und zum anderen ja auch frei darin sind, Ihrer eigenen Kreativität Vorschub zu leisten. Im Zweifel gilt aber auch wie immer: Schauen Sie ins Repository [2] (das entwickelt sich

• ;tldr

Es wird richtig viel passieren in diesem Artikel, und mit 8.2 nähern wir uns einem fulminanten (vorübergehenden) Abschluss dieser Serie – bleiben Sie dran! Dieses Mal bevölkern Sie die Lounge. Dazu bauen Sie die einfache Waypoint-Navigation um in eine Finite State Machine mit Potenzial auf „Sprungmarken“, visualisieren den sprichwörtlichen Geduldsfaden der NPC, bauen erste UI-Elemente ein und lassen Ihren Player endlich (!) Getränke vom Regal nehmen und den NPCs überreichen. Das Highlight dieses Artikels ist dabei sicher die FSM, und in der Fortsetzung schreiben Sie unter anderem Ihren ersten Pixel-Shader und ergänzen dynamische Lichteffekte!

spätestens heute zu einem geflügelten Wort!). Nähern wir uns dem Stand also Schritt für Schritt an und beginnen damit, einen NPC eine Bestellung aufgeben zu lassen.

Die Bestellung

Unter Kenneys Asset Packs finden Sie eines namens *Emote Pack*. In dessen Unterverzeichnis *Tilesheets* suchen Sie sich einfach eines der Pixel Style Packs (Bild 2) aus und übernehmen es in Ihr Spiel auf bekannte Art. Die Idee hinter diesem Pack ist es eigentlich, Emotionen darzustellen. Fröhlich, traurig, gelangweilt, voller Ideen, belustigt et cetera. Wir übernehmen sie und gestalten sie für unsere Zwecke um: Bauen

Sie einfach eine Kaffeetasse und eine Energydrink-Flasche ein wie im Screenshot zu sehen. Für den Moment können Sie das TileSheet aber auch schon mal as-is nutzen, einfach, um erst einmal damit zu testen.

Bisher haben Sie Grafiken eher selten direkt aus TileSheets heraus geladen, sondern das Framework den Job machen lassen. Falls Sie übrigens den Commits aus dem Repository folgen, haben Sie aber schon gesehen, wie Sie direkt Texturen aus TileSheets heraus extrahieren können: Für das visuelle Debugging gibt es inzwischen einen Shortkey, um es ein- und auszuschalten, und dabei – wenn eingeschaltet – ändert sich der Mauscursor (in eine Textur, die direkt aus einem TileSheet extrahiert wurde).



Bunter, interessanter, schöner: die Lounge (Bild 1)



Emotionen, wo man hinschaut (Bild 2)

Aber zurück zum Thema. Für die Emotes binden Sie das TileSheet Ihrer Wahl in das Projekt wie gewohnt über das Content-Verzeichnis und MGCB ein. In *NonPlayer.cs* benötigen Sie dann einen *Texture2DAtlas*, ein *Sprite* und für einen kurzen Moment eine *Texture2D*. Fügen Sie nun vor *LoadContent* folgende Variablen in Ihren Quellcode ein:

```
Texture2DAtlas emoteAtlas;
Sprite emoteSprite;
```

In *LoadContent* laden Sie das Sheet und erzeugen dann ein *Sprite*:

```
var emoteTexture =
    Game.Content.Load<Texture2D>("Player/EmoteSheet");
emoteAtlas = Texture2DAtlas.Create("emoteAtlas",
    emoteTexture, 16, 16);
var emoteSheet = new SpriteSheet(
    "emoteSheet", emoteAtlas);
emoteSprite =
    emoteSheet.CreateSprite(0);
emoteSprite.IsVisible = false;
emoteSprite.Depth = 0.2f;
```

Damit wird für den Moment eine Sprechblase als *Sprite* bereitgestellt, deren Inhalt uns nicht wirklich interessiert. Aber dazu kommen wir gleich. Sichtbar ist sie eh (noch) nicht. Außerdem finden Sie hier bereits einen Hinweis auf das Thema „DrawOrder“. Sie erinnern sich? Da war doch was ...

Nun ist es so, dass Sie das *Sprite* immer dann anzeigen wollen, wenn Ihr NPC am Tresen steht und eine Bestellung aufgeben möchte. Zwar gibt es schon eine *Waypoint*-Beschreibung für den Tresen – aktuell nutzen Sie dafür die *IdleTime* und die *FaceDirection* –, aber noch nichts, das auf eine Bestellung hinweisen würde. Ergänzen Sie daher die *WayPoint*-Klasse um eine Property *ShowEmote* vom Typ *bool* und setzen Sie sie in *GameMain* bei der Initialisierung der beiden NPCs auf *true*.

Außerdem ergänzen Sie *NonPlayer.cs* noch um die private Variable *wayPointReached* vom Typ *bool* – am besten direkt oberhalb der *Update*-Methode. Setzen Sie sie dabei direkt auf *false*.

Und nun kann es schon losgehen: In *NonPlayer.Update*, innerhalb des Blocks, der positiv geprüft hat, ob der *Waypoint* erreicht wurde, und bevor die *remainingIdleTime* auf 0 geprüft wird, fügen Sie folgenden Block ein:

```
wayPointReached = true;

if (waypoints[currentWaypointIndex].ShowEmote &&
    emoteSprite is not null && !emoteSprite.IsVisible)
{
    emoteSprite.TextureRegion = emoteAtlas.GetRegion(
        random.Next(0, 29));
    emoteSprite.IsVisible = true;
}
```

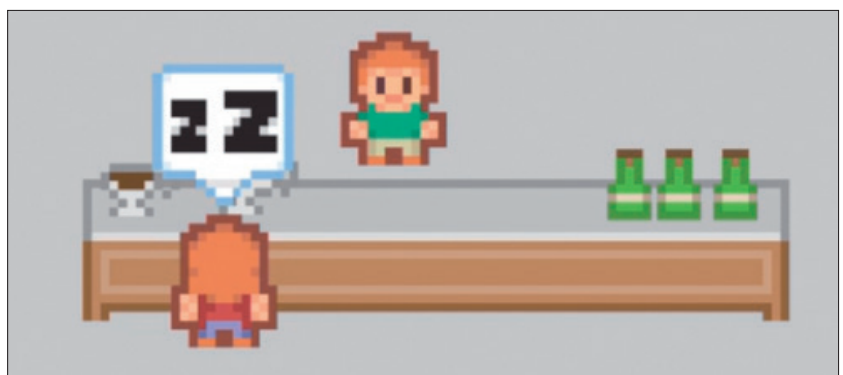
Hier wird die tatsächlich angezeigte Sprechblase für den Moment zufällig ausgewählt. Und damit das nicht bei jedem *Update*-Aufruf passiert, geschieht das nur, wenn sie zunächst noch unsichtbar ist.

Ein paar Zeilen später, innerhalb von *if(!isIdling)*, setzen Sie das *Sprite* wieder zurück:

```
wayPointReached = false;

if (waypoints[currentWaypointIndex].ShowEmote &&
    emoteSprite is not null && emoteSprite.IsVisible)
    emoteSprite.IsVisible = false;
```

Abschließend müssen Sie es nur noch rendern. Dazu ergänzen Sie die *Draw*-Methode zu Beginn um folgende Zeilen: ►



Was bitte sehr ist das für eine Bestellung? (Bild 3)


```
if (waypoints[currentWaypointIndex].ShowEmote
    && wayPointReached)
{
    emoteSprite.Draw(spriteBatch, new Vector2
        (position.X, position.Y - 16), 0f,
        new Vector2(1, 1));
}
```

Spielen Sie auch gerne mit den Werten. Die Idee dabei ist, dass das Sprite etwas oberhalb des NPC angezeigt wird.

Damit sollte Ihr NPC nun zumindest schon mal vor sich hinbrabbeln (vergleiche **Bild 3**). Allerdings wird Ihnen noch immer auffallen, dass es Probleme mit der Reihenfolge gibt, in der NPCs, Sprechblasen, Objekte und auch Ihr eigener Player gezeichnet werden. Laufen Sie einfach etwas herum im Spiel; Sie sehen dann schon, wovon die Rede ist.

DrawOrder

Um das Problem zu lösen, müssen Sie MonoGame erklären, in welcher Reihenfolge die Inhalte, die an dieser Stelle relevant sind, zu zeichnen sind. Da wir uns im Moment auf den Main-Screen konzentrieren, schauen Sie einmal in dessen *Draw*-Methode. Der *SpriteBatch*, der den Player, die NPCs und die Objekte umgibt, ist der Schlüssel. Ergänzen Sie den Aufruf um eine Sortierreihenfolge:

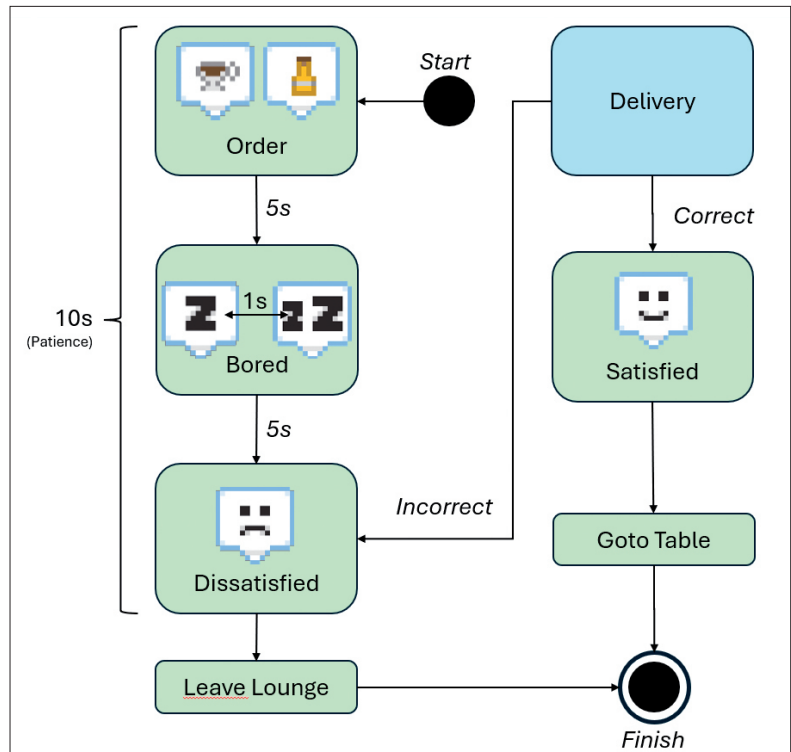
```
spriteBatch.Begin(
    transformMatrix: Game.Camera.GetViewMatrix(),
    samplerState: SamplerState.PointClamp,
    sortMode: SpriteSortMode.BackToFront );
```

Das allein genügt aber nicht, da MonoGame nicht weiß, was genau weiter vorne und was weiter hinten ist. Diese Information müssen Sie selbst hinterlegen. Und da sich NPCs auch bewegen, muss das in Teilen in jedem Update passieren.

Woran also machen Sie fest, was weiter vorne und was weiter hinten ist? Ganz einfach: In Ihrem Spiel ist alles das weiter vorne, was eine größere *y*-Koordinate hat. Denken Sie kurz darüber nach. Ergibt das Sinn? Zu einfach eigentlich, oder?

● Und wenn alles auf gleicher Höhe ist?

Es gibt einen Trick, um dafür zu sorgen, dass sich bewegende Sprites, die auf der gleichen *y*-Koordinate unterwegs sind, sich nicht bei jedem *Draw*-Aufruf abwechselnd überdecken: In dem Fall könnten Sie zum Beispiel einen kleinen Teil der *x*-Koordinate bei der Berechnung der Tiefe mit einbeziehen. Im Ergebnis würde dann beispielsweise das Sprite, das etwas weiter rechts auf dem Bildschirm zu sehen ist, immer zuvorderst gerendert.



Der Order-Workflow (Bild 4)

Die Auflösung, mit der Sie aktuell arbeiten, geht von 900 Pixeln in der Vertikalen aus. Nehmen Sie daher die Mitte davon und ermitteln Sie, auf was für einen Wert zwischen 0 und 1 sich die aktuelle Position eines Objekts projizieren lässt.

In *GameCharacter* ersetzen Sie dazu die abstrakte *Update*-Methode durch etwas Konkretes:

```
public virtual void Update(GameTime gameTime)
{
    sprite.Depth = (450 - position.Y) / 450;
}
```

Damit das auch mit den Objekten aus dem *ObjectsLayer* funktioniert, passen Sie deren *Draw*-Methode in den *Tiled-MapObjectLayer*-Extensions an. Ersetzen Sie den bisherigen *Draw*-Aufruf durch folgende zwei Zeilen:

```
var depth = (450 - position.Y) / 450;

spriteBatch.Draw(tileset.Texture, position,
    sourceRectangle, Color.White, 0f, new Vector2(0, 0),
    1.0f, SpriteEffects.None, depth);
```

Das müsste es im Grunde genommen schon gewesen sein. Probieren Sie es aus, um sicherzugehen, dass nun alles in der Reihenfolge gezeichnet wird wie gedacht.

Demnächst gelangen wir übrigens an einen wirklich lästigen Punkt. Tatsächlich wirft er bereits seine Schatten voraus und mich würde wirklich interessieren, ob Sie es schon gemerkt haben! Spätestens mit dem nächsten Thema sollte es Ihnen auffallen.

Der OrderState und gelangweilte NPC

Da ist es. Der Begriff kündigt es an. „OrderState“. Aber genug der Andeutungen, lassen Sie uns einmal einen Blick auf den gewünschten Workflow des NPC bei einer Bestellung werfen (Bild 4).

Ob die Zeiten darin stimmen oder die Sprechblasen so auch im Spiel aussehen werden, sei erst einmal dahingestellt. Jetzt gerade geht es um die Idee, die hinter dem Ablauf einer Bestellung steckt. Und die geht so: Der NPC sagt, ob er einen Kaffee oder einen Energydrink haben möchte. Dann wartet er geduldig fünf Sekunden lang. Dann ist er zunächst einmal gelangweilt und demonstriert das, indem die beiden Emotes Z und ZZ im Sekundentakt abwechselnd angezeigt werden (Fies: Falls der Spieler abgelenkt war, sieht er nun nicht mehr, was der NPC bestellt hat!). Nach aber spätestens weiteren fünf Sekunden verlässt der NPC unzufrieden die Lounge. Sollten Sie irgendwann innerhalb der zehn Sekunden ein falsches Getränk ausgeliefert haben: Weg isser. Haben Sie es aber geschafft, dann geht der NPC zunächst einmal an einen Tisch. Entsprechend den sonstigen Waypoints läuft er dann irgendwann von dort aus wieder raus aus der Lounge.

Wie bekommen Sie dieses ultrakomplexe Verhalten in die *NonPlayer.Update*-Methode implementiert? Indem Sie (Schritt 1) alles überspringen, was Sie in den entsprechenden Commits von mir zu dem Thema vorfinden und es dann (Schritt 2) gleich richtig machen.

Und auch auf die Gefahr hin, dass er dabei die Hände über dem Kopf zusammenschlägt, überlege ich ernsthaft, mit Stefan Lieser über das Thema Flow Design zu sprechen [3].

Die Kurzzusammenfassung lautet: Die *WayPoint*-Klasse darf dran glauben. Denn die hat safe mehr Probleme geschaffen, als sie gelöst hätte. Glauben Sie nicht? Ist aber so.

Die *WayPoint*-Klasse hat keine Sicht darauf, dass sich der NPC in unterschiedlichen Status befinden kann: Er kann sich von A nach B bewegen, in der Gegend herumstehen, am Tresen etwas bestellen. Und da die *WayPoint*-Klasse das nicht weiß, muss sie alle Informationen in sich beherbergen, und über Toggle muss von außen entschieden werden, wie damit umzugehen ist. Hinzu kommt, dass dadurch alle möglichen Zustände in einer einzigen *Update*-Klasse verarbeitet werden, die ansonsten nur von einem Waypoint zum nächsten läuft.

Und das ändern Sie nun, indem Sie Ihren ersten Rohentwurf einer sogenannten Finite State Machine implementieren. Kurz: FSM. Zu Deutsch würde man dazu Zustandsautomat oder endlicher Automat sagen. Tut aber niemand.

FSM I

Beginnen Sie mit etwas Einfachem. Ergänzen Sie ein leeres Interface als Basis für die künftigen States:

```
public interface INpcState
{
}
```

Nun ändern Sie den Primary Constructor der Klasse *NonPlayer*, sodass keine Liste von Waypoints, sondern eine Liste von *INpcStates* entgegengenommen wird:

```
public class NonPlayer(Game game, Character character,
    List<INpcState> states) : GameCharacter(game, character)
```

Statt eines *currentWaypointIndex* brauchen Sie jetzt einen *currentStateIndex*. Dass Visual Studio Ihnen nun jede Menge Fehler meldet, ignorieren Sie für den Moment. Darum kümmern wir uns gleich.

Ergänzen Sie die *NonPlayer*-Klasse dafür um die folgende Methode:

```
public INpcState GetNextState()
{
    currentStateIndex = (currentStateIndex + 1) %
        states.Count;
    return states[currentStateIndex];
}
```

Und um Ihnen schon einmal einen Eindruck zu vermitteln, wie Sie nun Zustände eines NPC beschreiben können, beginnen Sie mit dem einfachsten und fügen dem Projekt eine neue Datei hinzu: *NpcStateIdling*.

```
public class NpcStateIdling(TimeSpan idleTime,
    Direction faceDirection = Direction.None) : INpcState
{
    public TimeSpan IdleTime { get => idleTime; }

    public Direction FaceDirection { get =>
        faceDirection; }
}
```

Da es im Moment übrigens erst einmal um die Struktur der *Update*-Methode der *NonPlayer*-Klasse geht, wird sich zunächst noch nicht viel „Verhalten“ in den State-Klassen befinden. Das kommt noch. Versprochen.

Ergänzen Sie eine weitere neue Datei: *NpcStateWalking*:

```
public class NpcStateWalking(Vector2 position,
    bool reset = false) : INpcState
{
    public Vector2 Position { get => position; }

    public bool Reset { get => reset; }
}
```

Last, but not least folgt noch *NpcStateOrdering*:

```
public class NpcStateOrdering(TimeSpan patienceTime,
    TimeSpan unpatienceTime, Direction faceDirection) :
    INpcState
{
    public TimeSpan PatienceTime { get => patienceTime; }

    public TimeSpan UnpatienceTime { get => unpatienceTime; }

    public Direction FaceDirection { get => faceDirection; }
}
```

In der Annahme, dass der erste State eines NPC immer *NpcStateWalking* sein wird (für den Moment), könnten Sie nun die Start-Position des NPC in *NonPlayer.LoadContent* entsprechend anpassen:

```
SetX(((NpcStateWalking)states[0]).Position.X);
SetY(((NpcStateWalking)states[0]).Position.Y);
```

Der eigentliche Knackpunkt ist nun allerdings, dass Sie in der *Update*-Methode einen *switch* über *states[currentStateIndex]* machen können. Probieren Sie es aus und schauen Sie einmal, ob Sie Ihren Code dahingehend angepasst bekommen. Alternativ finden Sie die vollständige *Update*-Methode ob des recht großen Umfangs wieder im Repository [2].

Als Nächstes ergänzen Sie in *GameMain* die Liste der Enums um den *NpcState*:

```
public enum NpcState
{
    Walking,
    Ordering,
    Idling
}
```

Und wenn alles so weit geklappt hat, können Sie ebenfalls in *GameMain* die Initialisierung des ersten NPC durch den Code aus [Listing 1](#) ersetzen.

Was Sie damit erreicht haben, ist schon einmal ein riesiger Schritt raus aus der unübersichtlichen *Update*-Methode.

Listing 1: NPC

```
NPCs.Add(new NonPlayer(this, Character.Luna, states:
[
    new NpcStateWalking(new Vector2(28, 170), reset:
        true),
    new NpcStateWalking(new Vector2(170, 170)),
    new NpcStateWalking(new Vector2(170, 230)),
    new NpcStateWalking(new Vector2(220, 230)),
    new NpcStateOrdering(
        patienceTime: TimeSpan.FromSeconds(5),
        unpatienceTime: TimeSpan.FromSeconds(5),
        faceDirection: Direction.Up),
    new NpcStateWalking(new Vector2(220, 260)),
    new NpcStateWalking(new Vector2(190, 260)),
    new NpcStateWalking(new Vector2(190, 295)),
    new NpcStateWalking(new Vector2(175, 295)),
    new NpcStateIdling(idleTime: TimeSpan.FromSeconds(5),
        faceDirection: Direction.Up),
    new NpcStateWalking(new Vector2(285, 295)),
    new NpcStateWalking(new Vector2(285, 474)),
    new NpcStateIdling(
        idleTime: TimeSpan.FromSeconds(7)),
]));
```

Denn nun steht der Weg offen für in sich gekapselte Zustände, die nach Belieben ergänzt oder angepasst werden können, ohne dass Ihnen gleich alles zusammenbricht.

Nach diesem Zwischenschritt wird es Zeit, den Code aus der *NonPlayer.Update*-Methode gänzlich auf die States zu verteilen und so die Möglichkeiten einer Finite State Machine noch besser zu nutzen.

FSM II

Erweitern Sie dafür das *INpcState*-Interface:

```
public interface INpcState
{
    void Update(NonPlayer npc, GameTime gameTime);
}
```

Den Code für die Implementierung der einzelnen States finden Sie in seiner vollständigen Pracht... Sie wissen schon.

Da die States bei Abschluss übrigens direkt zum nächsten wechseln, passen Sie die entsprechende Methode in *NonPlayer* entsprechend an:

```
public void GoToNextState()
{
    currentStateIndex = (currentStateIndex + 1) %
        states.Count;
}
```

Falls Sie sich selbst an das kleine Refactoring wagen, holen Sie einfach den jeweiligen Code aus der *Update*-Methode der *NonPlayer*-Klasse in die entsprechenden neuen State-Klassen. Das Ziel ist es dabei, die *Update*-Methode selbst auf folgende Zeilen zu reduzieren:

```
public override void Update(GameTime gameTime)
{
    base.Update(gameTime);

    states[currentStateIndex].Update(this, gameTime);

    position = Vector2.Round(position * 5) / 5.0f;

    sprite.Update(gameTime);
}
```

Damit ist es gelungen, das Verhalten des NPC in den verschiedenen Zuständen sauber in eigenen Klassen zu definieren und so einiges an Abhängigkeiten aufzulösen!

Und raten Sie mal: Auch Verzweigungen werden dadurch möglich. Dazu später mehr. Für den Moment bringen Sie nun den zweiten NPC zurück (in *GameMain*) und freuen sich darüber, dass im Spiel alles so aussieht, wie... nun ja, wie vorher.

```
NPCs.Add(new NonPlayer(this, Character.Tommy, states:
[
    new NpcStateWalking(new Vector2(300, 430), reset:
        true),
```

● PlayOnce

An dieser Stelle auch mal ein kleines Gimmick! Sie erinnern sich daran, dass ein NPC beim Aufgeben einer Bestellung eine kleine Fanfare abspielt. Das Problem: Das darf nur einmal passieren. Daher gab es bisher einen kleinen Schalter, den wir umgelegt haben, sobald der Sound abgespielt wurde, sodass das kein zweites Mal passiert.

Im Repository [2] finden Sie weitere nützliche Extension-Methods für die Klasse *SoundEffectInstance*, mit denen sich das vereinfachen lässt. In der *Update*-Methode aus der *NonPlayer*-Klasse wird sie bereits genutzt. Prüfen Sie einmal für sich selbst, ob Sie das so für sinnvoll halten.

```
new npcStateIdling(idleTime: TimeSpan.FromSeconds(3)),
new npcStateWalking(new Vector2(300, 230)),
new npcStateWalking(new Vector2(255, 230)),
new npcStateOrdering(
    patienceTime: TimeSpan.FromSeconds(4),
    unpatienceTime: TimeSpan.FromSeconds(4),
    faceDirection: Direction.Up),
new npcStateWalking(new Vector2(255, 295)),
new npcStateWalking(new Vector2(240, 295)),
new npcStateIdling(idleTime:
    TimeSpan.FromSeconds(4),
    faceDirection: Direction.Up),
new npcStateWalking(
    new Vector2(190, 295)),
new npcStateWalking(
    new Vector2(190, 185)),
new npcStateWalking(
    new Vector2(15, 185)),
new npcStateIdling(idleTime:
    TimeSpan.FromSeconds(5)),
));
```

OrderState-Visualisierung

Was ist damit gemeint? Nun, so wie Sie die NPCs angelegt haben, haben diese eine gewisse Geduldsspanne. Das nutzen Sie zum einen, um nach einigen Sekunden das Bild der Bestellung zu entfernen. Das macht es interessanter für den Spieler. Sie können das auch dazu nutzen, um dem Spieler Feedback darüber zu geben, wie lange er noch Zeit hat (Bild 5). Bei der Gelegenheit wäre es auch zusätzlich empfehlenswert, den Code in *NpcStateOrdering* etwas aufzuräumen. Ihr Ziel ist folgende *Update*-Methode:

```
public override void Update(Npc npc, GameTime gameTime)
{
    UpdateProgressBar(gameTime);

    // Zeige oder aktualisiere Emote-Sprite
    UpdateEmoteSprite(npc, gameTime);
```

```
// Aktualisiere Idle-Status und Zeit
UpdateIdleTime(npc, gameTime);
```

```
// Zeige Idle-Animation
UpdateIdleAnimation(npc);
```

```
// Prüfe, ob Idle-Zeit abgelaufen ist
if (!IsIdling())
{
    TransitionToNextState(npc);
}
}
```

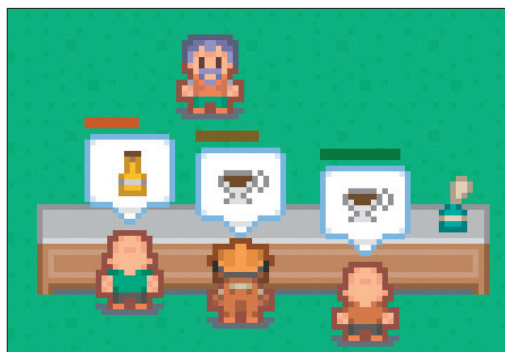
Und wie immer gilt: Schauen Sie gerne ins Repository. Sie finden zum Thema *OrderState* Visualization ein entsprechendes Commit.

Doch eins nach dem anderen. Auf dem Screenshot haben Sie bereits erkannt, dass die Farbe der Leiste von Grün (noch geduldig) bis Rot (ziemlich ungeduldig) wechselt (Idee meiner Tochter). Was Sie auf dem Screenshot nicht sehen können: Kurz vor Ablauf der Zeit blinkt der Balken auch noch (meine Idee).

Für das Colorfading ist ein Feature aus MonoGame.Extended verantwortlich, das sich Tweening nennt. Das kommt von „In Between“ und bedeutet so viel, als dass damit automatisch Zwischenschritte berechnet werden.

Meine Empfehlung lautet: Orientieren Sie sich an der offiziellen Dokumentation [4] und implementieren Sie diesen Effekt selbstständig direkt in *NpcStateOrdering*.

Vergleichen Sie im Anschluss Ihren Stand mit dem aktuell vollständigen Code für eine Methode *UpdateProgressBar* wie gewohnt im Repository.



OrderState-Visualisierung (Bild 5)

Collision Refactoring

Bisher – das ist Ihnen schon mehrfach aufgefallen – wurde die Kollisionserkennung nur im *MainScreen* aufgerufen und das Ergebnis entsprechend verarbeitet.

Dem *StorageScreen* haben wir bis dato kaum Beachtung geschenkt. Und was erschwerend hinzukommt: Später wollen Sie von dort Getränkenachschub organisieren. Dazu wird es auf der Map irgendeine Art von Treisen oder Regal geben müssen, und Sie kommen nicht um eine Kollisionserkennung herum.

Dass der Code so hart mit dem *MainScreen* verdrahtet ist, wie er es ist, ergibt daher mittelfristig keinen Sinn.

Mein Vorschlag: Überlegen Sie sich eine neue Klasse *CollisionSystem*, die die notwendigen Schritte in sich vereint, und entwerfen Sie vielleicht eine zusätzliche Hilfsklasse *CollisionResult*, die Ergebnisse aggregiert. Dazu kann der bisherige *CollisionImpact* gehören sowie etwaige Türen und auch eine Möglichkeit, von draußen zu erfragen, ob eine Kollision stattgefunden hat. Auf diese Weise können Sie das neue ►

● Listing 2: NinePatch

```
private Texture2DRegion[] GetNinePatchRegions(
    Texture2DAtlas atlas, int startRow,
    int startColumn, int tilesPerRow)
{
    Texture2DRegion[] patches =
    [
        atlas.GetRegion(tilesPerRow * (startRow + 0) +
            startColumn + 0),
        atlas.GetRegion(tilesPerRow * (startRow + 0) +
            startColumn + 1),
        atlas.GetRegion(tilesPerRow * (startRow + 0) +
            startColumn + 2),
        atlas.GetRegion(tilesPerRow * (startRow + 1) +
            startColumn + 0),
        atlas.GetRegion(tilesPerRow * (startRow + 1) +
            startColumn + 1),
        atlas.GetRegion(tilesPerRow * (startRow + 1) +
            startColumn + 2),
        atlas.GetRegion(tilesPerRow * (startRow + 2) +
            startColumn + 0),
        atlas.GetRegion(tilesPerRow * (startRow + 2) +
            startColumn + 1),
        atlas.GetRegion(tilesPerRow * (startRow + 2) +
            startColumn + 2),
    ];

    return[.. patches];
}
```

Collision System sowohl im MainScreen wie auch im StorageScreen einfach nutzen. Später werden Sie es zudem um Kollisionsabfragen zwischen Player und NPC erweitern.

Meine Idee davon, wie das aussehen kann, finden Sie – wie üblich – im Repository [2] im dazugehörigen Commit.

NinePatch und ein weiteres Update

Hier mal wieder ein neuer Begriff, dem Sie in der Spieleentwicklung häufiger über den Weg laufen werden. Insbesondere, sobald es um das Thema UI geht. Im ersten Screenshot dieses Artikels haben Sie bereits gesehen, dass es erste UI-Elemente geben wird. In Windows Forms würden Sie vielleicht von Textboxen sprechen, oder Labels. Doch schauen wir uns zunächst einmal das Problem an – danach die Lösung.

Ein Beispiel dafür, was Sie erreichen wollen, zeigt **Bild 6**. Und auf **Bild 7** sehen Sie, was Sie beispielsweise im UI Pack von Kenney und anderen vorfinden.

Möglicherweise erkennen Sie die Herausforderung. Angefangen damit, dass Sie auf bekannte Weise das TileSheet laden können und über einen Atlas auf die einzelnen Tiles darin zugreifen können ... aber wie erzeugen Sie daraus nun eine Fläche, wie sie in **Bild 6** dargestellt ist?

Was Sie wollen, ist, dass Sie einfach beim *Draw*-Aufruf auf einem *SpriteBatch* ein Rechteck angeben können mit vier Werten: X, Y, Width, Height. Fertig. Doch wie soll das gehen?

Technisch sicherlich, indem Sie dann ausrechnen, wo die Tiles mit den vier Ecken genau hinmüssen, um dann die vier Tiles mit den Seiten dazwischen vertikal und horizontal zu strecken, um abschließend das Tile aus der Mitte zu verwenden,

den, um die Mitte des Rechtecks aufzufüllen. Klingt flexibel. Dynamisch. Einfach. Viel Spaß dabei.

Oder aber Sie aktualisieren MonoGame.Extended auf die neueste Version (mindestens 3.8.2.1105) und verwenden einen sogenannten *NinePatch*. Den gibt es zwar auch schon in Ihrer Version, aber sollten Sie nicht zwischendurch noch einmal wieder aktualisiert haben, ist die Wahrscheinlichkeit groß, dass Ihre Implementierung vom *NinePatch* noch nicht ganz so arbeitet, wie sie soll.

Was ist nun also überhaupt ein *NinePatch*? Einfach gesagt: Ein Flickenteppich aus neun Texturen, aus denen sich mittels der gerade beschriebenen Idee eine Fläche dynamisch erzeugen lässt. Und das funktioniert so: Beginnen Sie damit, dass Sie in *GameMain* eine neue Variable einfügen:

```
NinePatch uiBubblePatch;
```

Dann ergänzen Sie eine kleine Hilfsmethode, die so aussieht wie in **Listing 2**. Zwar ist diese Methode nicht maximal flexibel, sie hilft aber enorm im Umgang mit den TileSheets, die wir aus den Asset Packs von Kenney nutzen.

Denn nun können Sie Ihren *NinePatch* direkt in *GameMain.LoadContent* erzeugen:

```
uiBubblePatch = new NinePatch(
    GetNinePatchRegions(uiAtlas,
        startRow: 13,
        startColumn: 6,
        tilesPerRow: 30));
```

Falls Sie den *uiAtlas* noch nicht aus einem der letzten Commits haben, in dem auch der Mauscursor aus dem gleichen TileSheet heraus erstellt wurde, benötigen Sie vor dem Aufruf noch folgende Zeilen:

```
var uiTexture = Content.Load<Texture2D>(
    "UI/UIpackSheet_transparent");
```

Zeit für die Levelzeit
(Bild 6)




```
var uiAtlas = Texture2DAtlas.Create(
    "uiAtlas", uiTexture, 16, 16, spacing: 2);
```

Mittels *tilesPerRow*, *startRow* und *startColumn* geben Sie die Daten an, die die Methode *GetNinePatchRegions* braucht, um die korrekten Tiles aus der Gesamttextur zu extrahieren und die entsprechenden Regions der neuen Instanz von *NinePatch* zuzuordnen. Unter bestimmten Voraussetzungen geht das auch einfacher, funktioniert aber nicht mit den Texturen von Kenney (Stichwort: „Spacing“).

Würden Sie nun die *Draw*-Überladung am *SpriteBatch* einfach so nutzen, die der *NinePatch* für Sie mitbringt, dann würde das aber leider nicht ganz funktionieren.

Der Grund: Bei allen Ihren *SpriteBatches* geben Sie bisher die *ViewMatrix* Ihrer Kamera als Transformationsmatrix an. Das tun Sie, damit Sie sich mit dem Spieler durch die Map bewegen können. Die UI-Elemente aber sollen an Ort und Stelle fixiert sein.

Natürlich könnten Sie dazu die Koordinaten in jedem Update neu berechnen – in Abhängigkeit zur Map. Aber das allein wäre noch nicht genug. Denn die Kamera-*ViewMatrix* enthält zudem noch eine Skalierung vom virtuellen *Viewport* mit 320 x 180 Pixeln auf den fünfmal so großen *BackBuffer* mit 1600 x 900 Pixeln (und merken Sie sich das; später wollen wir das noch konfigurierbar machen).

Um das zu lösen, fügen Sie folgende Variablen in *GameMain* ein:

```
const int uniformScale = 5;

const int virtualWidth = 320;
const int virtualHeight = 180;

int backBufferWidth = virtualWidth * uniformScale;
int backBufferHeight = virtualHeight * uniformScale;
```

Nutzen Sie sie direkt in der *Initialize*-Methode:

```
var viewportAdapter = new BoxingViewportAdapter(
    Window, GraphicsDevice, virtualWidth, virtualHeight);
...
graphics.PreferredBackBufferWidth = backBufferWidth;
graphics.PreferredBackBufferHeight = backBufferHeight;
```

Und nun können Sie in *GameMain-Draw* Richtung Ende der Methode einfach Ihren *NinePatch* wie folgt zeichnen:

```
var uniformScaleMatrix = Matrix.CreateScale(
    (float)uniformScale);

spriteBatch.Begin(transformMatrix: uniformScaleMatrix,
    samplerState: SamplerState.PointClamp);
{
    spriteBatch.Draw(uiBubblePatch, new Rectangle(
        5, 5, 75, 25), Color.White);
```

```
}
spriteBatch.End();
```

Damit haben Sie Ihr erstes richtiges UI-Element auf dem Bildschirm. Herzlichen Glückwunsch! Im nächsten Abschnitt werden Sie darin auch direkt die Levelzeit anzeigen. Ach stimmt ... bisher gibt es noch gar keine Levelzeit. Kein Wunder, dass sich das Spiel noch nicht wie ein Spiel anfühlt. Ist es das vielleicht? Zeit, die abläuft?

Levelzeit und Barkeeper in (Half) Action

Ich wette, an dieser Stelle sind Sie absolut selbstständig in der Lage, so etwas wie eine Levelzeit zu implementieren. Hier die Anforderung: Im Code können Sie festlegen, wie lange eine Runde grundsätzlich dauert. Die Zeit läuft während des Spiels ab, und sobald sie auf null runter ist, wird das Spiel zunächst einmal beendet. Probieren Sie es aus.

Und hier kommt mein Vorschlag: Beginnen Sie damit, in *GameMain* direkt einen *SpriteFont* und ein paar Variablen zu definieren, mit denen Sie das Ganze steuern können.

```
SpriteFont font;
const int levelTime = 30;
int timeToEnd;
```

Den Font laden Sie wie gewohnt in der *LoadContent*-Methode. Den Font selbst bekommen Sie wie üblich bei Kenney [5].

```
font = Content.Load<SpriteFont>(
    "Fonts/KenneyPixel");
```

Am Ende der *Update*-Methode ermitteln Sie dann, wie viel Zeit noch übrig ist, und beenden bei Bedarf das Spiel.

```
timeToEnd = (int)(levelTime -
    gameTime.TotalGameTime.TotalSeconds);
```

```
if (timeToEnd == 0)
    Exit();
```

Und abschließend zeichnen Sie die Levelzeit in der *Draw*-Methode im *SpriteBatch*, das bereits das UI zeichnet (unter Umständen verkleinern Sie dabei den Kasten auf eine Breite von 50 Pixeln).

```
string time = timeToEnd.ToString();
Vector2 textMiddlePoint = font.MeasureString(time) / 2;
Vector2 position = new(31,16);

spriteBatch.DrawString(font, time, position, Color.Beige,
    Of, textMiddlePoint, 1f, SpriteEffects.None, 0f);
```

Für ein kleines Spiel wie dieses mag es übrigens convenient sein, mit Magic Numbers (31, 16) um sich zu werfen. Den- ►

Listing 3: TryGetTileTextureOfType

```

public static bool TryGetTileTextureOfType(
    this TiledMapObjectLayer layer, string typeName,
    out Texture2DRegion textureRegion)
{
    foreach (var obj in
        layer.Objects.OfType<TiledMapTileObject>())
    {
        if (obj.Tile.Properties.TryGetValue("Type",
            out TiledMapPropertyValue value) &&
            value.ToString() == typeName)
        {
            var tileset = obj.Tileset;
            textureRegion = new Texture2DRegion(
                tileset.Texture, tileset.GetTileRegion(
                    obj.Tile.LocalTileIdentifier));
            return true;
        }
    }

    textureRegion = null;
    return false;
}

KeyboardState newState = Keyboard.GetState();

if (newState.IsKeyDown(Keys.Space) &&
    previousState.IsKeyUp(Keys.Space))
{
    var coffeeBoxCollision =
        collisionSystem.CheckObjectCollision(coffeeBox,
            Game.Player);
    var energyBoxCollision =
        collisionSystem.CheckObjectCollision(energyBox,
            Game.Player);

    if (coffeeBoxCollision)
    {
        GameMain.Log.AddLog("CoffeeBox Collision Detected");
        energySprite.IsVisible = false;
        coffeeSprite.IsVisible = true;
    }

    if (energyBoxCollision)
    {
        GameMain.Log.AddLog("EnergyBox Collision
            Detected");
        energySprite.IsVisible = true;
        coffeeSprite.IsVisible = false;
    }
}

```

ken Sie alternativ aber auch über eine sinnvolle Lösung nach wie beispielsweise einen UI-Manager, mit dem Sie das UI vernünftig verwalten können.

Wenn Sie das Spiel nun starten, sollten Sie dabei zuschauen können, wie Ihnen die Zeit durch die Finger rinnt wie Sand. Das erzeugt schon mal ein wenig Druck, oder? Oder auch nicht. Denn Sie können rein gar nichts dagegen tun. Kommen wir also direkt zum nächsten Schritt und lassen Ihren Spieler (endlich!) Getränke aufsammeln.

„Ober! Einen Kaffee bitte!“

An dieser Stelle führen viele Wege nach Rom. Einige sind komplexer als andere. Wir wählen einen eher einfachen Weg und denken insbesondere nicht so, wie wir es in der Realität täten, sondern mehr so, wie es in einem Computerprogramm möglich ist. Das bedeutet: Wir organisieren uns einfach die beiden Texturen für Kaffee und Energydrink und verarbeiten beide, als würden sie immer getragen werden ... zeichnen aber immer nur dann eine Textur, wenn der Player das entsprechende Getränk genommen hat. Dazu muss das Spiel zunächst einmal verstehen, dass der Player am Regal steht – und wo genau –, und dann auf eine Taste – zum Beispiel [Space] – reagieren.

Beginnen Sie erst einmal damit, den Code für das Tragen eines Getränks zu schreiben, und ergänzen Sie die *TiledMapObjectLayer*-Extensions um eine weitere Hilfs-Erweiterung *TryGetTileTextureOfType* (Listing 3). Die Idee ist, dass Sie in

Tiled Objekte hinzufügen, die das Regal in Form von Rechtecken umgibt, und diesen Bezeichnungen geben („Coffee“, „Energy“). Auf Bild 8 sehen Sie ein Beispiel.

Dann ergänzen Sie den *MainScreen* um zwei Variablen:

```

Sprite energySprite;
Sprite coffeeSprite;

```

In *LoadContent* können Sie die Texturen nun extrahieren:

```

if( objectsLayer.TryGetTileTextureOfType(
    "Energy", out Texture2DRegion energyTexture))
{
    energySprite = new Sprite(energyTexture);
    energySprite.IsVisible = false;
}

if (objectsLayer.TryGetTileTextureOfType(
    "Coffee", out Texture2DRegion coffeeTexture))
{
    coffeeSprite = new Sprite(coffeeTexture);
    coffeeSprite.IsVisible = false;
}

```

Im Grunde genommen könnten Sie nun auch einfach in der *Draw*-Methode das Sprite über den Player zeichnen und da-

bei ein wenig auf der x- und der y-Achse verschieben (ausgehend von der Player-Position), um es so aussehen zu lassen, als trüge der Player zum Beispiel die Kaffeetasse. Aber lassen Sie uns doch direkt mal ein wenig Spice in die Sache bringen und folgende Ideen umsetzen:

- Wie wäre es, wenn der Player das Getränk immer in der gleichen Hand halten würde – egal, in welche Richtung Sie gerade gehen?
- Wie wäre es, wenn sich das Getränk auf der y-Achse ebenfalls immer ein wenig hoch und runter bewegen würde – genau wie der Player, der „atmet“.

Meine Idee davon, wie sich so etwas implementieren ließe, können Sie sich in dem entsprechenden Commit anschauen.

In jedem Fall sollte Ihr Player ab diesem Moment eine Kaffeetasse mit sich herumtragen. Das ist doch schon mal was! Und haben Sie bereits die Map um Boxen rund um die Bereiche des Regals ergänzt wie auf **Bild 8** zu sehen? Eine Box für die Kaffeetassen, eine für die Energydrinks? Falls ja, schauen wir uns als Nächstes die Kollision zwischen Player und den Boxen an.

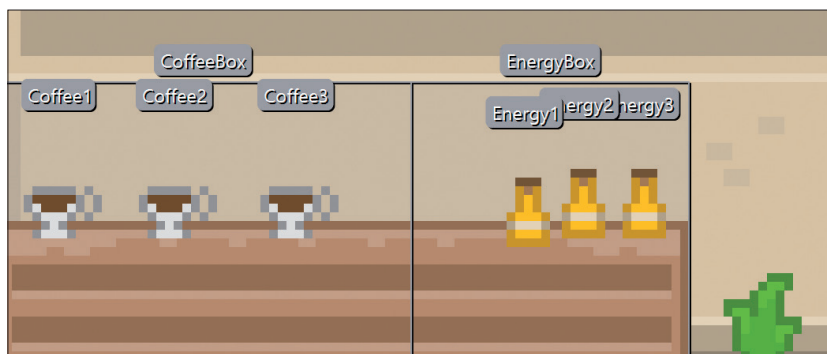
Zunächst benötigen Sie eine weitere *TiledMapObjectLayer*-Extension:

```
public static TiledMapObject GetObjectByName(
    this TiledMapObjectLayer layer, string name)
{
    foreach (
        var obj in layer.Objects.OfType<TiledMapObject>())
    {
        if (obj.Name == name)
            return obj;
    }
    return null;
}
```

Um nun auf Kollisionen mit Objekten, die Sie so ermitteln können, prüfen zu können, ergänzen Sie Ihr Collision System (Sie haben doch eines, oder?) um folgende Methode:

```
public bool CheckObjectCollision(
    TiledMapObject mapObject, Player player)
{

```



Getränkeregale mit Kaffee und Energydrinks (**Bild 8**)

```
var globalRect = new RectangleF(
    mapObject.Position.X, mapObject.Position.Y,
    mapObject.Size.Width, mapObject.Size.Height);

var playerRect = new RectangleF(
    player.Position.X, player.Position.Y, 16, 16);

var collision = globalRect.Intersects(playerRect);

return collision;
}
```

Nun holen Sie sich Referenzen auf die Boxen, die Sie in Tiled erstellt haben, im *MainScreen*:

```
TiledMapObject coffeeBox;
TiledMapObject energyBox;
```

Und in *LoadContent*:

```
coffeeBox = objectsLayer.GetObjectByName("CoffeeBox");
energyBox = objectsLayer.GetObjectByName("EnergyBox");
```

Vor der *Update*-Methode legen Sie nun eine Variable für den *KeyboardState* an:

```
KeyboardState previousState = Keyboard.GetState();
```

In *Update* nutzen Sie sie direkt nach der Kollisionsprüfung (siehe **Listing 4**). Und wie es der Zufall so will ... war es das schon. Sobald Sie während des Spiels in der Lounge die Leerschritt-Taste drücken, prüft der Code, ob Sie zufällig mit der *CoffeeBox* oder der *EnergyBox* kollidieren. Ist das so, wird das jeweilige Sprite sichtbar geschaltet; das andere unsichtbar. Den Rest übernimmt die *Draw*-Methode. Probieren Sie es aus!

An dieser Stelle gibt es übrigens einige Kleinigkeiten zu beachten, an die Sie eventuell noch nicht gedacht haben: Was passiert eigentlich genau, wenn der Player die Lounge verlässt und in den Vorratsraum verschwindet? Ich sage es Ihnen: Die Lounge wird vom *ScreenManager* vereinfacht gesagt zurückgesetzt. Eigentlich aber sollten die NPCs ja weiterhin dort auf Sie warten, oder?

Long story short: Nach ein paar Experimenten, die Sie bei einer Tüte Popcorn in den Commits verfolgen können, habe ich das Aktualisieren der NPCs einfach vom *MainScreen* nach *GameMain* verschoben. Das genügt für unseren Fall bereits. Tun Sie das auch der Einfachheit halber. Nun aber geht es um die Player-NPC-Kollision.

Player trifft auf NPC

Die Übergabe eines Getränks an einen NPC passiert über mehrere Schritte. Wichtig an dieser Stelle zu erwähnen ist übrigens, dass der Prozess in diesem Spiel extrem verein- ►

facht umgesetzt ist – im Vergleich zu anderen Spielen. Denn es ist nicht wichtig, nachvollziehen zu können, wer wann was bekommt und oder mit sich herumträgt. Oder anders ausgedrückt: Es gibt keine Form von Inventar. Alles, was dieses Spiel wissen muss, ist: Welches Getränk hat ein NPC geordert und hat der Player es innerhalb der verfügbaren Zeit übergeben.

Damit das funktioniert, muss auch hier das Spiel auf das Drücken der [Space]-Taste reagieren und verstehen, ob das am Tresen passiert ist. Ist das der Fall, muss es prüfen, ob der Player gerade in der Nähe eines NPC ist. Und ist das wiederum der Fall, muss geprüft werden, ob das Getränk, das gerade in der Hand des Players sichtbar ist (!), zu der Bestellung passt. Im Anschluss wird das Getränk dann einfach wieder unsichtbar geschaltet – und fertig ist die Laube.

Beginnen Sie mit der Erweiterung der Kollisionserkennung. Für die Kollision zwischen Player und NPC halte ich es für sinnvoll, das Rechteck der LegCollision vom Player mit einem Kreis rund um einen NPC kollidieren zu lassen. Dafür nutzen Sie folgende Methode, deren Quelle ein Thread auf StackOverflow ist [6], und ergänzen Sie im CollisionDetector:

```
public GameCharacter CollisionCheck(
    GameCharacter character, Vector2 position)
{
    if (RectangleCircleOverlap(
        new CircleF(
            character.Position + new Vector2(8, 8), 20),
        new RectangleF(
            position.X + 4, position.Y + 12, 8, 4)))
        return character;

    return null;
}
```

Die Methode *RectangleCircleOverlap* leiten Sie einfach aus dem StackOverflow-Artikel ab oder entleihen sie sich aus dem Repository [2]. Nun müssen Sie noch die *CollisionResult*-Klasse um eine Property *CollisionCharacter* vom Typ *GameCharacter* ergänzen und sind all set to go.

Dabei ist es der erste Schritt, genau wie mit dem Getränke-regal zunächst auf Kollision mit dem Tresen zu prüfen und dann im Anschluss herauszufinden, ob Sie dort auf einen NPC getroffen sind:

```
var barBoxCollision =
    collisionSystem.CheckObjectCollision(barBox,
        Game.Player);
...
if (barBoxCollision)
{
    GameMain.Log.AddLog("BarBox Collision Detected");
    energySprite.IsVisible = false;
    coffeeSprite.IsVisible = false;

    if (collisionResult.CollisionCharacter != null) {
        var npc = (NPC)collisionResult.CollisionCharacter;
        if (npc.GetState() is NPCStateOrdering orderingState)
```

● Listing 4: Coffee-/Energybox-Kollision

```
KeyboardState newState = Keyboard.GetState();

if (newState.IsKeyDown(Keys.Space) &&
    previousState.IsKeyUp(Keys.Space))
{
    var coffeeBoxCollision =
        collisionSystem.CheckObjectCollision(
            coffeeBox, Game.Player);
    var energyBoxCollision =
        collisionSystem.CheckObjectCollision(
            energyBox, Game.Player);

    if (coffeeBoxCollision)
    {
        GameMain.Log.AddLog(
            "CoffeeBox Collision Detected");
        energySprite.IsVisible = false;
        coffeeSprite.IsVisible = true;
    }

    if (energyBoxCollision)
    {
        GameMain.Log.AddLog(
            "EnergyBox Collision Detected");
        energySprite.IsVisible = true;
        coffeeSprite.IsVisible = false;
    }
}
```

```
{
    if (orderingState.CurrentOrder ==
        Game.Player.BeverageHolding)
    {
        orderingState.CurrentOrder = Order.Satisfied;
    }
    else
    {
        orderingState.CurrentOrder = Order.Dissatisfied;
    }
}

OnBeverageDropped();
}
```

CurrentOrder ist eine neue Property an *NpcStateOrdering*:

```
Order currentOrder = Order.None;

public Order CurrentOrder
{
    get { return currentOrder; }
```

```
set {
    currentOrder = value;
    remainingIdleTime = TimeSpan.Zero;
}
}
```

Und statt dass eine Bestellung wie bisher einfach nur zufällig generiert und angezeigt wird, darf sich der NPC diese nun auch in *NpcStateOrdering.UpdateEmoteSprite* merken:

```
if (!emoteSprite.IsVisible)
{
    currentOrder = random.Next(0, 2) ==
        0 ? Order.EnergyDrink : Order.Coffee;

    GameMain.Log.AddLog($"{npc.Character} orders
        {currentOrder}");

    emoteSprite.TextureRegion = emoteAtlas.GetRegion(
        (int)currentOrder);
    emoteSprite.IsVisible = true;
}
```

Eine Möglichkeit übrigens, um *GameMain* etwas über den Bestellvorgang aus *NpcStateOrdering* mitzuteilen, wäre ein sogenannter Callback-Handler. So kann *GameMain* dann nämlich zum Beispiel auch Punkte für die Auslieferung von Getränken anzeigen. Übergeben Sie dazu einfach eine Methode aus *GameMain* an den State und rufen Sie diese direkt aus dem State heraus auf. In der Definition der States kann das in *GameMain* zum Beispiel so aussehen:

```
new NpcStateOrdering(
    patienceTime: TimeSpan.FromSeconds(5),
    unpatienceTime: TimeSpan.FromSeconds(5),
    faceDirection: Direction.Up,
    onDrinkDelivered: OnDrinkDelivered),
...
private void OnDrinkDelivered(bool drinkDelivered)
{
    ...
}
```

Und in *NpcStateOrdering* so:

```
public class NpcStateOrdering(
    TimeSpan patienceTime, TimeSpan unpatienceTime,
    Direction faceDirection, Action<bool>
    onDrinkDelivered) : NpcStateBase
{
    ...

    private void TransitionToNextState(Npc npc)
    {
        ...
        onDrinkDelivered?.Invoke(
            currentOrder == Order.Satisfied ? true : false);
    }
}
```

```
currentOrder = Order.None;

// Zum nächsten State wechseln
npc.GoToNextState();
}
}
```

Eine Idee für eine vollständige Implementierung finden Sie wie immer in den Commits. Der Code dort enthält darüber hinaus noch viel mehr. Und deshalb lohnt sich ein Blick darauf!

So, liebe Leser: Wir haben soeben die 12-Seiten-Schallgrenze erreicht und blenden den Artikel hier für Sie aus.

Fazit und Ausblick

Der Titel dieses Artikels lautete: „Jetzt wird gespielt“. Und ... haben Sie gespielt? Falls nicht, probieren Sie bis zum nächsten Artikel doch mal, die Punktevergabe selbstständig zu implementieren. Die Frage danach, ab wann sich das Spiel auch wie ein solches anfühlt, kann ich für meinen Teil zumindest inzwischen beantworten. Erstens: Die Spielzeit muss limitiert sein. Zweitens: Es muss eine Aufgabe geben und der Spieler muss in der Lage sein, sie zu erfüllen. Drittens: Ohne Belohnung kein Spaß. Sie denken schon darüber nach, oder?

Neben dem dafür erforderlichen Balancing nutzen wir im kommenden Artikel die neue State Machine, um auch Sprungmarken einzuführen. Sprich: Je nach Ausgang einer Bestellung verhalten sich die NPCs unterschiedlich.

Dann bauen wir noch einen PixelShader ... Sie kommen aber nie drauf, wofür. Und last, but not least spielen wir noch mit Lichteffekten und versuchen das Spiel mit der letzten noch offenen Baustelle grundsätzlich abzuschließen: dem Restocking-Mechanismus.

Und dann? Dann ändern wir das Format. Ganz einfach. Denn Sie glauben doch nicht allen Ernstes, dass mir die Ideen ausgehen, oder? ■

[1] Christian Jacob, *Einfach um die Ecke gebracht*, dotnetpro 1/2025, Seite 114 ff., www.dotnetpro.de/A2501MonoGame

[2] GitHub-Repository zur Artikelserie, www.dotnetpro.de/SL2502MonoGame1

[3] Flow Design, <https://flow-design.info>

[4] Tweening, www.dotnetpro.de/SL2502MonoGame2

[5] Kenney Fonts, <https://kenney.nl/assets/kenney-fonts>

[6] RectangleCircleOverlap auf StackOverflow, <https://stackoverflow.com/a/43546279>



Christian Jacob

ist Entwickler und Co-Founder der Blue Sky Innovations GmbH. Er unterstützt Unternehmen bei der Einführung moderner Strategien sowie bei Planung und Umsetzung individueller Softwarelösungen auf Basis von Microsoft .NET.

cjacob@blueskyinnovations.de

dnpCode

A2502MonoGame

ARBEITSMARKT

TRENDS UND JOBS FÜR ENTWICKLER

Gisma

Bewerbungstrends

Die Bewerbungsanforderungen werden immer schlanker: In den meisten Unternehmen ist inzwischen nur noch der Lebenslauf zwingend erforderlich. Nur vier Prozent der Unternehmen fordern ein klassisches Anschreiben. Diese Entwicklung zeigt eine aktuelle Analyse der Gisma University of Applied Sciences. Die Studie vergleicht die Bewerbungsanforderungen von DAX-40-Konzernen mit denen deutscher Unicorns. Sie verdeutlicht den Bedeutungsverlust traditioneller Bewerbungsdokumente. Die zentralen Ergebnisse sind in **Bild 1** dargestellt. Etwa acht von zehn Unternehmen betrachten einen Lebenslauf als unerlässlich, während nur zwölf Prozent diesen als optional einstufen. Auch Zeugnisse haben an Bedeutung verloren. Lediglich bei neun Prozent der befragten Unternehmen sind sie verpflichtend. Rund jedes vierte Unternehmen

ermöglicht eine One-Click-Bewerbung über Plattformen wie LinkedIn (16 Prozent), Xing (9 Prozent), Indeed (6 Prozent) oder Facebook (1 Prozent). Nutzer können zum Beispiel ihr LinkedIn-Profil mit einem Jobportal verknüpfen. Alle relevanten Daten werden dabei automatisch übertragen, um sofort eine Bewerbung abzusenden.

Ein klarer Trend zeigt sich in der Wahl der Bewerbungssprache. 44 der 68 befragten Unternehmen bieten ihre Bewerbungsprozesse hauptsächlich auf Englisch an. Deutsch nutzen insgesamt 29 Unternehmen. Sechs Unternehmen erlauben sowohl Englisch als auch Deutsch. Zwei weitere Unternehmen stellen noch andere Sprachoptionen zur Verfügung.

Freelancer-Markt

Sinkende Auslastung

Immer mehr Freiberufler sind aktuell offen für neue Projektaufträge. Das meldet die Platt-

form für die Vermittlung selbstständiger Experten im deutschsprachigen Raum Freelance.de. Erstmals haben mehr als 100 000 Freelancer ihr Profil online geschaltet, so der Dienstleister. Robin Gollbach, der CEO von Freelance.de: „Der Rekordwert spiegelt das stark wachsende Angebot hochqualifizierter Freelance-Leistungen wider. Noch nie standen so viele spezialisierte Fachkräfte bereit, projektweise bei Unternehmen etwas voranzubringen.“

Dem wachsenden Angebot selbstständiger Expertinnen und Experten steht eine gleichzeitig fallende Nachfrage nach ihren Diensten gegenüber (**Bild 2**). So haben Unternehmen bis Ende Oktober 2024 57 000 neue Projekte auf Freelance.de ausgeschrieben. Im gleichen Zeitraum des Vorjahres waren es noch 68 300 Projekte (-16,5 Prozent).

Gollbach: „Trotz der Rückgänge im Auftragsvolumen sind viele hochqualifizierte Freelancer derzeit noch gut ausgelas-

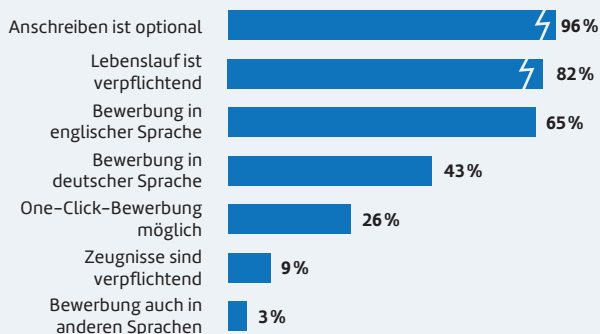
tet. Seit dem Corona-bedingten Peak der Freelancer-Nachfrage im Jahr 2022 normalisiert sich die Lage langsam wieder. Viele freiberufliche Expertinnen und Experten vor allem aus der IT und dem Interims-Management waren bis zum Anschlag ausgelastet. Nun scheint sich der Projektstau in diesen Bereichen abzubauen. Spezialisten auf Zeit sind jetzt eher verfügbar als noch vor einem Jahr.“

Bitkom

KI-Fortbildung

Viele Arbeitnehmer werden in Fragen Künstlicher Intelligenz allein gelassen. Nur fünf Prozent der Unternehmen schulen alle Mitarbeiter im Umgang mit ChatGPT und Co., weitere 16 Prozent bilden zumindest einen Großteil weiter. Und 28 Prozent beschränken ihre KI-Fortbildungen auf ausgewählte Beschäftigte. Zugleich geben 48 Prozent der 602 befragten Unternehmen an, dass sie gar keine Weiterbildungen zu KI

● Bewerbungstrends

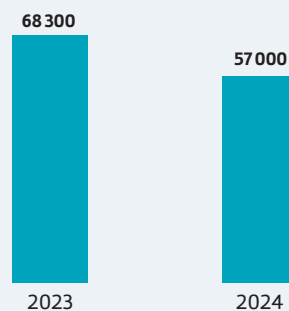


Für eine Bewerbung bei deutschen DAX-Konzernen und Unicorns ist in der Regel ein Lebenslauf verpflichtend, Anschreiben und Zeugnisse sind meist optional (**Bild 1**)

dotnetpro 2/2025

Quelle: Gisma University of Applied Sciences, n=68

● Projekte für Freelancer



Die Zahl der von Unternehmen auf Freelance.de ausgeschrieben Projekte für Freiberufler ist im Jahresvergleich um 16,5 Prozent zurückgegangen (**Bild 2**)

dotnetpro 2/2025

Quelle: Freelance.de

Zahl des Monats

Laut Entwicklerplattform WeAreDevelopers sind **70 Prozent** der Entwickler aktuell nicht aktiv auf der Suche nach einem Job, sind aber durchaus offen für neue Karrieremöglichkeiten. Aktiv auf der Suche sind lediglich **6 Prozent** der über 200 000 Mitglieder der Plattform. Die übrigen 24 Prozent sind nicht zu einem Wechsel bereit oder machten keine Angaben. Quelle: WeAreDevelopers

ermöglichen (Bild 3). Dabei gibt es ein großes Interesse aufseiten der Beschäftigten – 61 Prozent aller Erwerbstätigen würden sich gerne zu KI fortbilden.

Bitkom-Präsident Dr. Ralf Wintergerst: „Generative KI kann schon heute die Produktivität am Arbeitsplatz erhöhen. Unternehmen sollten ihre Beschäftigten frühzeitig rund um KI weiterbilden und dabei auch auf rechtliche Vorgaben, Datenschutz und Datensicherheit hinweisen.“

Obwohl nur knapp die Hälfte der Unternehmen aktuell zumindest Teile der Belegschaft zu KI fortbildet, wird der Bedarf an KI-Know-how in der Wirtschaft in den kommenden Jahren stark wachsen. Mehr als zwei Drittel der Unternehmen sind überzeugt, dass generative KI einen zusätzlichen Bedarf an Fachkräften schafft. Wintergerst: „Wir werden in den kommenden Jahren mehr KI-Expertise in Wirtschaft und Verwaltung brauchen. Unternehmen sind daher gut beraten, schon

heute durch Weiterbildungsangebote auf den Bedarf an mehr KI-Kompetenz zu reagieren.“

GitHub

Bessere Code-Qualität

Schon frühere Studien haben ergeben, dass GitHub Copilot Entwicklern hilft, schneller zu programmieren, und überdies haben viele Entwickler mehr Vertrauen in mit KI-Unterstützung geschriebenen Code.

Jetzt hat das GitHub-Forschungsteam untersucht, ob die Qualität von mithilfe von GitHub Copilot geschriebenem Code objektiv besser oder schlechter ist als der ohne KI-Hilfestellung. Um diese Frage zu beantworten, wurde eine randomisierte kontrollierte Studie durchgeführt, für die 202 Entwickler mit mindestens fünf Jahren Erfahrung rekrutiert wurden. Die Hälfte der Teilnehmer erhielt nach dem Zufallsprinzip Zugang zu GitHub Copilot, während die andere Hälfte

angewiesen wurde, keine KI-Tools zu verwenden. Alle Teilnehmer wurden gebeten, eine Programmieraufgabe zu lösen und API-Endpunkte für einen Webserver zu schreiben. Die Bewertung des Codes erfolgte mit Unit-Tests und einer von Entwicklern durchgeführten Expertenprüfung.

Die Ergebnisse zeigen insgesamt, dass der mit GitHub Copilot erstellte Code eine höhere Funktionalität und bessere Lesbarkeit aufweist, von besserer Qualität ist und höhere Genehmigungsraten erhält. Die wichtigsten Ergebnisse:

- **Erhöhte Funktionalität:** Entwickler mit GitHub-Copilot-Zugang hatten eine um 56 Prozent höhere Wahrscheinlichkeit, alle zehn Unit-Tests in der Studie zu bestehen, was darauf hindeutet, dass GitHub Copilot Entwicklern hilft, funktionelleren Code zu schreiben.
- **Bessere Lesbarkeit:** Bei Blindprüfungen wies der mit GitHub Copilot geschriebene

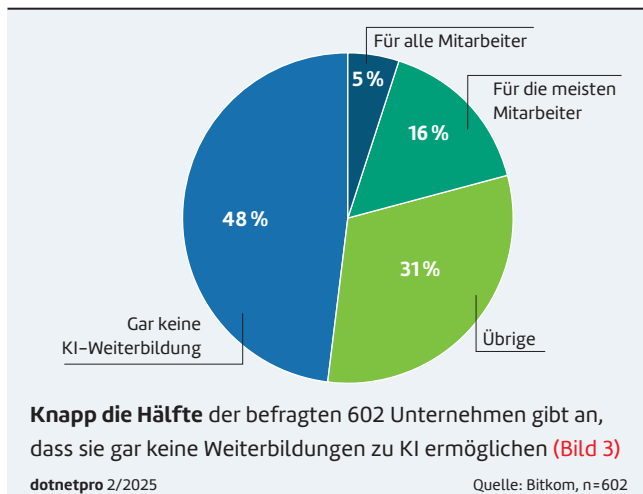
Code weniger Lesbarkeitsfehler auf, sodass die Entwickler im Durchschnitt 13,6 Prozent mehr Codezeilen schreiben konnten, ohne auf Lesbarkeitsprobleme zu stoßen.

- **Bessere Codequalität:** Die Lesbarkeit verbesserte sich um 3,62 Prozent, die Zuverlässigkeit um 2,94 Prozent, die Wartbarkeit um 2,47 Prozent und die Prägnanz um 4,16 Prozent. Laut GitHub waren alle Zahlen statistisch signifikant.

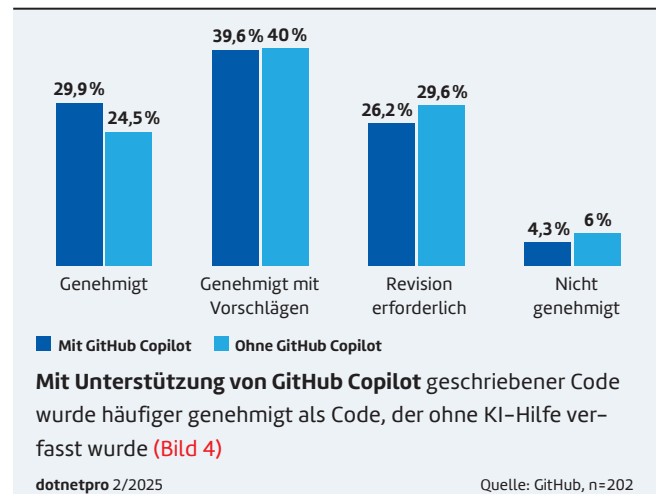
- **Höhere Genehmigungsraten:** Entwickler genehmigten Code, der mit GitHub Copilot geschrieben wurde, mit einer höheren Wahrscheinlichkeit (Bild 4). Das bedeutet, dass solcher Code früher zusammengeführt werden kann.

Weitere Informationen zu den Ergebnissen, zum Aufbau der Studie sowie dazu, wie die Studienautoren einen Code Error definiert haben, finden Sie im GitHub-Blogpost unter <https://tinyurl.com/GHCodeQy>.

• KI-Fortbildungen



• Code-Qualität





developer media jobs

Der Stellenmarkt für Entwickler

LANGE SUCHEN ODER GLEICH FINDEN. WAS DARF'S SEIN?

**JETZT
ANZEIGE
SCHALTEN!**



jobs.developer-media.de

Reinhold Fritsch, Tel.: +49 731 88005-8285 | Sabine Vockrodt, Tel.: +49 731 88005-8222
oder per E-Mail an: jobs@developer-media.de

JOBS für Entwickler

Auf jobs.developer-media.de finden
Software-Entwickler und -Architekten
ständig aktuelle Job-Angebote

Ihr Job-Angebot fehlt?

Sie suchen Entwickler in Festanstellung oder projektbezogen?
Dann sollten auch Sie unser Online-Jobportal für Ihre Mitarbeitersuche nutzen.
Ihr Stellenmarkt-Team berät Sie gern zu den attraktiven Promotionmöglichkeiten.

**Gerne können Sie Ihre Stellenausschreibung zu jederzeit online
selbst einbuchen unter jobs.developer-media.de**



Diese und viele weitere aktuelle Job-Angebote finden Sie auf jobs.developer-media.de

Landesbank Hessen-Thüringen	Datenbank Experte (w/m/d)	Offenbach am Main, Erfurt
Landesbank Hessen-Thüringen	IT-Risikomanager (w/m/d)	Düsseldorf
SWAGLab GmbH	Mitgestalter (m/w/d)	Hamburg
Landesbank Hessen-Thüringen	IT Portfoliomanager (w/m/d)	Offenbach am Main
Landesbank Hessen-Thüringen	Application Manager SAP (w/m/d)	Wiesbaden, Offenbach
J.P. Sauer & Sohn Maschinenbau GmbH	ERP-Anwendungsentwickler (m/w/d) Infor LN	Kiel
Hochschule Schmalkalden	Leitung des Hochschulrechenzentrums (m/w/d)	Schmalkalden
Landesbank Hessen-Thüringen	Senior Application Manager (w/m/d)	Düsseldorf
J.P. Sauer & Sohn Maschinenbau GmbH	Data Manager (m/w/d) Engineering Data	Kiel
Bergische Universität Wuppertal	Wissenschaftliche*r Mitarbeiter*in IT-Sicherheit	Wuppertal
Justus-Liebig-Universität Gießen	IT-Mitarbeiter/in (m/w/d) im Bereich Service und IT-Support	Gießen
Klinikum Garmisch-Partenkirchen GmbH	IT-Spezialist klinische Systeme (m/w/d)	Garmisch-Partenkirchen
Landesbank Hessen-Thüringen	Application Manager (w/m/d) SAP Bilanzerstellung	Erfurt, Frankfurt am Main, Offenbach am Main
Hochschule Worms	Informationssicherheitsbeauftragte/r (m/w/d)	Worms
Justus-Liebig Universität Giessen	IT-Administrator/in (m/w/d) für (BOS-)Funksysteme und Gefahrenmeldesysteme	Gießen
Katholisches Klinikum Bochum gGmbH	IT-Leitung (m/w/d)	Bochum
PMG AG	Fachinformatiker / Systemtechniker (Voice Solutions) (m/w/d)	Inning am Ammersee

Ihre Anzeige ist noch nicht dabei? Rufen Sie uns an:

Reinhold Fritsch, Tel.: +49 731 88005-8285, E-Mail an: jobs@developer-media.de

● Impressum

**Ebner Media Group GmbH & Co. KG****Büro München**

Postfach 20 15 52, 80015 München
Brienner Straße 45 a-d, 80333 München

Sitz von Redaktion, Anzeigen, Vertrieb

Brienner Straße 45 a-d, 80333 München
Telefon +49 731 88005-8000
info@dotnetpro.de
www.dotnetpro.de

Sitz des Verlags

Ebner Media Group GmbH & Co. KG
Karlstraße 3, 89073 Ulm

Geschäftsführer

Marco Parrillo

Chefredaktion

Fernando Schneider (ViSdP)

Redaktion

Tilman Börner

Freie Mitarbeiter dieser Ausgabe**Redaktion**

Ernst Altmannshofer, Sabine Block,
Bernhard Lauer

Autoren

Klaus Aschenbrenner, Daniel Basler, Tilman Börner,
Dr. Fabian Deitelhoff, Mykola Dobrochynskyy,
Christian Giesswein, Christian Havel, Christian Jacob,
Bernhard Lauer, Hendrik Lösch, Robert Mühsig,
Marc Müller, Bernhard Pichler,
Dr. Holger Schwichtenberg, David Tielke,
Helmut Vonhoegen, Alexandra Waldherr,
Christian Wenz, Alexander Zeitler

Art Directorin

Maria-Luise Steinkühler

Gestaltung**EMG Design Unit**

Dagmar Breitenbach, Simone Köhnke,
Petra Reichenspurner

Kundenservice / Einzelheftbestellung

EBNER MEDIA GROUP,
Brienner Straße 45 a-d, 80333 München
Telefon +49 731 88005-8205
kundenservice@ebnermedia.de
(Mo.-Do. 9.00 Uhr bis 17.00 Uhr,
Fr. 9.00 Uhr bis 15.00 Uhr)

Erscheinungsweise

dotnetpro erscheint monatlich.
ISSN 1610-1553. Postvertriebskennzeichen B30751

Nachdruck

© Nachdruck nur mit schriftlicher Genehmigung

Developer Events

Charlotte Frieß, Director Live Communication Developer
charlotte.friess@ebnermedia.de

Mediaberatung und Content-Marketing-Lösungen

Thomas Deck
Telefon +49 731 88005-8282
thomas.deck@ebnermedia.de

Beratung Stellenmarkt

Reinhold Fritsch
Telefon +49 731 88005-8285
reinhold.fritsch@ebnermedia.de

Client Success (Disposition)

clientsuccess@ebnermedia.de

Leitung Herstellung / Vertrieb / Kundenservice

Thomas Heydn

Anzeigentarife

Anzeigenpreisliste gültig ab 1.1.2019
digital.dotnetpro.de/mediadaten/

Druck

Silber Druck GmbH & Co. KG
Otto-Hahn-Straße 25, 34253 Lohfelden

Sonderdruck

Von allen Beiträgen dieser Ausgabe können für
Werbezwecke Sonderdrucke hergestellt werden.
Anfragen bitte an fernando.schneider@ebnermedia.de

Bezugspreise

dotnetpro ist das Profi-Magazin für Entwickler
und erscheint zwölfmal im Jahr.
Das Einzelheft kostet im Inland 19,90 €.
Der Preis für das Premium-Plus-Abonnement
(12 Ausgaben Print + Digital)
beträgt im Inland 182,40 €
(Digital 152,40 € + Print 30,00 €),
Schweiz 304,00 CHF
(Digital 167,70 CHF + Print 136,30 CHF),
sonstiges Ausland 206,40 €
(Digital 152,40 € + Print 54,00 €) pro Jahr.

Abonnement-Bedingungen

Das Abonnement gilt für ein Jahr, anschließend bis
zur Kündigung zum jeweils geltenden Jahresbezugspreis.
Dein Jahresabo ist ab dem Folgejahr auch monatlich
kündbar. Bei vorzeitiger Kündigung deines bereits
abgerechneten Bezugszeitraums erhältst du den
Betrag der Restlaufzeit zurückerstattet.

Veröffentlichung gemäß Art. 8 Abs. 3 BayPrG:

Alleinige Gesellschafterin der Ebner Media Group
GmbH & Co. KG ist die Ebner Ulm MGv GmbH, Ulm.

● Inserentenverzeichnis

Text Control GmbH

www.textcontrol.com

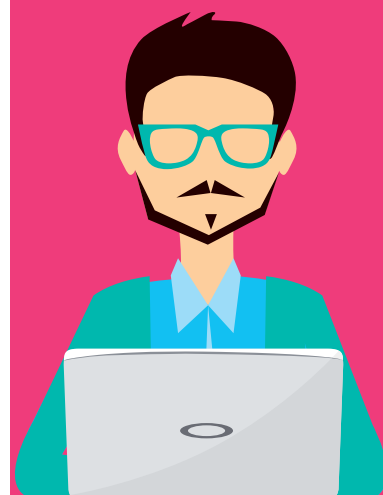
2

Ebner Media Group

www.ebnermedia.de 19, 31, 65,
113, 115, 127, 142-143, 147, 148

Autor:in werden!

Helfen Sie uns,
Softwareentwickler
weiterzubilden.



Schreiben Sie
Artikel für die
dotnetpro



info@dotnetpro.de

Anbieterverzeichnis

Persönliche Beratung bei Thomas Deck | +49 731 88005-8282 | thomas.deck@ebnermedia.de

	COUNT IT Group Softwarepark 49 4232 Hagenberg/AUSTRIA T: +43 (0)7236 – 20077 – 0 F: +43 (0)7236 – 20077 – 6717 m.szoeky@countit.at www.countit.at Markus Szöky	Die COUNT IT Group ist österreichweit einer der führenden Unternehmensdienstleister und mit über 120 MitarbeiterInnen das größte Unternehmen im Softwarepark Hagenberg. Unser agiles Softwarehouse realisiert neben maßgeschneiderten ERP- und DMS-Lösungen vor allem Individualsoftware-Projekte auf Basis modernster .NET-Technologien. Die Bandbreite reicht hier von Web-Anwendungen bis Softwarelösungen für die Industrie 4.0, wobei für uns als Qualitätsdienstleister Testautomatisierung wesentlich ist. Auf uns zählen (inter) nationale Kunden aus Handel, Baubranche und Industrie. Rechnen auch Sie mit uns!
	DATEV eG Paumgartnerstr. 6 – 14 90429 Nürnberg Deutschland Telefon: +49 911 319-0 https://www.datev.de	Die DATEV eG ist der drittgrößte Anbieter für Business-Software in Deutschland und einer der großen europäischen IT-Dienstleister. Das Leistungsspektrum der DATEV reicht von mehr als 200 PC-Programmen über Cloud-Lösungen wie Online-Anwendungen, Datenverarbeitung und -archivierung im Rechenzentrum bis hin zu Outsourcing-Leistungen sowie Sicherheitsdienstleistungen, mit denen die DATEV Unternehmen, Selbstständige, Kommunen, Vereine und Institutionen unterstützt. Beratungsleistungen und Angebote zur Wissensvermittlung in Deutschland und einigen anderen europäischen Ländern runden das Angebot ab.
	Developer Express, Inc 505 N. Brand Blvd. Suite 1450 CA 91203 Glendale USA ClientServices@devexpress.com www.devexpress.com	DevExpress engineers feature-complete UI Controls, enterprise-ready reporting systems, IDE productivity tools and business application frameworks for Visual Studio. Our technologies help you build your best, see complex software with greater clarity, increase your productivity and create stunning touch-enabled applications for Windows, the web and mobile devices without limits or compromise. To learn more and see how you can create business solutions customers expect today and leverage existing skillsets to build next generation applications for tomorrow, visit us online at www.devexpress.com .
	Kutura GmbH Oberaltenallee 20a 22081 Hamburg Telefon: +49 177 216 70 80 Mail: contact@kutura.digital kutura.digital/contact	Unternehmen, in denen sich Menschen entwickeln und wachsen können, gehört die Zukunft. Das gilt besonders in Zeiten, in denen Künstliche Intelligenz, Krisen und Komplexität unsere Aufmerksamkeit erfordern und unsere mentalen Ressourcen beanspruchen. Kutura entwickelt für und mit ihren Kunden eine zukunftsfähige Führungs- und Unternehmenskultur, die drei Zielen dient: Gemeinsam Ergebnisse schaffen, mehr Motivation und weniger Konflikte, zukünftige Stärke sicherstellen. Reimagining Leadership, Inspiring Happiness.
	prodot GmbH Schifferstraße 196 47059 Duisburg T: +49 (0)203 – 396508 – 0 info@prodot.de https://prodot.de	prodot – Software für Marktführer Intelligente Software für internationale Konzerne und mittelständische Unternehmen: prodot stärkt seit über 20 Jahren namhafte Kunden im weltweiten Wettbewerb mit effizienten, stabilen und kostensenkenden Lösungen. Mit Sorgfalt und Enthusiasmus entwickeln wir hochwertige Software, die sich durch intuitives Design auszeichnet. Digitale Prozesse und innovative Technologien wie IoT, AI, Mobile, Cloud sind unser Antrieb – Fortschritt, Qualität und Kontinuität unser Anspruch. Mit rund 80 Mitarbeitern setzen wir die Anforderungen unserer Kunden in hochwertige Softwareprodukte um. ALDI SÜD, Microsoft, Siemens, TK Elevator und weitere vertrauen uns seit vielen Jahren. Gerne zeigen wir Ihnen warum! Sprechen Sie uns an.
	Space Blocks Permissions as a Service Ein Service der wemogy GmbH Graf-Bernadotte-Str. 23 47906 Kempen Deutschland https://www.spaceblocks.cloud hello@spaceblocks.cloud	Die Developer-Plattform für Rollen- und Rechtemanagement Einfach Rollen und Berechtigungen im grafischen Low-Code-Editor anlegen und Zugriffsabfragen mit wenigen Zeilen Quellcode in die eigenen Anwendungen integrieren. Inklusive RBAC, Multi-Tenancy, hierarchischer Berechtigungsvererbung und vielem mehr. Geschrieben in .NET und für .NET-Entwickler konzipiert. Entweder von uns in der Space Blocks Cloud bereitgestellt oder im Selbstbetrieb. Jetzt ausprobieren mit der kostenlosen Entwickler-Version oder eine unverbindliche, kostenfreie Demo buchen!
	Text Control GmbH Überseeter 18 28217 Bremen Deutschland Telefon: +49 421 42706710 https://www.textcontrol.com/	Text Control wurde 1991 gegründet und ist ein preisgekrönter Microsoft Visual Studio Industry Partner und führender Anbieter von Reporting- und Textverarbeitungskomponenten für Microsoft-Entwickler. Die Produkte helfen tausenden von Entwicklern, Text- und Dokumentenverarbeitung in ihre Windows, Web und Mobil-Anwendungen zu integrieren. Text Control ist zu 100% inhabergeführt und damit komplett unabhängig von Investorenentscheidungen. Kunden profitieren von jahrelanger Erfahrung, einer sehr großen Nutzerbasis und den ausgereiften und zuverlässigen Produkten. Kernprodukt ist das Reporting-Framework, welches die Stärken eines Reporting-Tools mit einer, einfach zu handhabenen, Textverarbeitung kombiniert – voll programmierbar und integrierbar in einer .NET Anwendung.
	Carl Zeiss Digital Innovation GmbH Fritz-Foerster-Platz 2 01069 Dresden Telefon: +49 351 49701 – 500 info.digitalinnovation.de@zeiss.com www.zeiss.de/digital-innovation	ZEISS Digital Innovation – Als Teil der ZEISS Gruppe entwickeln wir individuelle und hochwertige digitale Lösungen, die unter anderem auf modernen Web- und Cloud-Technologien basieren. Mit agilen Methoden unterstützen wir unsere Kunden entlang ihrer Digitalisierungsprojekte – und darüber hinaus. Dabei entstehen innovative Plattformen auf einer skalierbaren Gesamtarchitektur. So steigern wir die Innovationskraft unserer Zielgruppe in ihren Märkten. www.zeiss.de/digital-innovation



Während Christian Sender sich mit dem Zusammenspiel von **Entity Framework Core mit mehreren DBA** beschäftigt, nehmen Elena Bochkor und Veikko Kryczyk die **Frontend-Backend-Kommunikation mit .NET MAUI, Azure und Firebase** unter die Lupe. Und nachdem Datenzugriffe eine leistungsfähige Datenbank erfordern, stellt Klaus Aschenbrenner das Open-Source-ORDBMS **PostgreSQL** vor.

Bild: EMConie

```

    usecaseDiagram
        actor Besucher
        actor Kontrolleur
        participant System
        participant Tickets
        participant QRCodeGenerator as QR Code Generator
        participant Kassensystem

        Besucher --> System
        Kontrolleur --> System
        System --> Tickets
        System --> QRCodeGenerator
        System --> Kassensystem
    
```

The diagram illustrates the interactions in a ticketing system. It features a central system boundary (a large circle with a smaller inner circle) and three actors: 'Besucher' (Visitor) and 'Kontrolleur' (Controller), represented by stick figures, and three external systems represented by cylinders: 'Tickets', 'QR Code Generator', and 'Kassensystem' (Cashier system). Arrows indicate the flow of interactions: 'Besucher' and 'Kontrolleur' interact with the central system. The central system interacts with 'Tickets', 'QR Code Generator', and 'Kassensystem'.



X: die News im Stream
https://twitter.com/dotnetpro_mag



SAVE
THE DATE

.NET DEVELOPER CONFERENCE '25

DIE KONFERENZ FÜR .NET ENTWICKLER:INNEN
24. - 27. NOVEMBER 2025 | KÖLN

Die .NET Developer Conference ist eine der führenden Plattformen für Weiterbildung in .NET-Technologien. Mit einem umfangreichen Programm aus Vorträgen und Workshops bietet die Konferenz fundiertes Wissen und praxisnahe Einblicke in zentrale Themen des .NET-Bereichs im Zeitalter von KI. Ein Muss für alle, die ihr Know-how erweitern und aktuelle Trends verfolgen möchten.

dotnet-developer-conference.de | #DDC | Find us on [f](#) [@](#) [X](#) [in](#)

Präsentiert von:



DEVELOPER WEEK '25

30. JUNI - 3. JULI 2025

M:CON CONGRESS CENTER ROSENGARTEN
MANNHEIM

**JETZT
TICKET
SICHERN!**

DAS EVENT FÜR AI, CLOUD, WEB UND .NET

Software Design Principles | Software Architecture | Software Quality | Software Testing | Software Security | Doing Agile/Agile in Practice | DevOps | CI / CD
Git, Github | .NET Fundamentals | .NET Backend (Development)
.NET Frontend (Development) | .NET Doing | Web Frontend | Web Backend
Web Hybrid | Web Design | Web Languages | Cloud Development
Container | Data & Analytics | Databases | GenAI Fundamentals
GenAI Tools & Solutions | Personal Growth | From Me to We | Sustainability
Robotics | IoT & Industry 4.0

developer-week.de | #DWX | Find us on

