

dotnetpro

by DWX

Software testen

Softwarequalität
steht und fällt mit
solidem Testing.
Was sich dabei mit
dem Einsatz von KI
ändert und wo die
eigene Expertise
weiterhin zählt.
Ab Seite 6

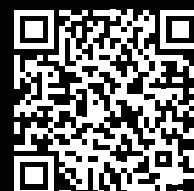
DWX

DIE KONFERENZ FÜR AI,
CLOUD, WEB & .NET DEVELOPMENT

JETZT
SCHNELL
BLIND PRICE
SICHERN!

29. JUNI BIS
02. JULI 26

M:CON CONGRESS CENTER
ROSENGARTEN | MANNHEIM



DWX ACADEMY

Maßgeschneiderte
Trainings für starke
Entwicklerteams

JETZT
BERATUNG
ANFRAGEN

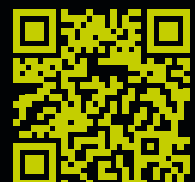
Wissen, das Projekte voranbringt.



Agilität | Architektur | Artificial Intelligence
Azure | Backend | C# | C++ | CI/CD | Cloud
Collaboration | Container | Data & Analytics
DevOps | Frontend | Git | Integration | Java
JavaScript | Leadership | .NET | Python | Rust
Softwaredesign | Softwarequalität | Testing
TypeScript | Web Hybrid | ...



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





Trugschluss

Nach wie vor werden Produktivitätssteigerung und Perfektion mit der Beherrschung von Technik gleichgesetzt. Fälschlicherweise.

Vorsprung durch Technik" war Anfang der 1970er-Jahre ein Versprechen auf mechanische Perfektion. Dieser Slogan eines deutschen Autobauers kam mir in den Sinn, als ich einen Beitrag über Frontends der Zukunft las und dachte: „KI ist in nur drei Jahren unverzichtbarer Bestandteil professioneller Softwareentwicklung geworden. Wow!“

Laut Umfragen nutzen bereits heute mehr als drei Viertel der Entwickler:innen KI-Assistenten im Alltag, mehr als die Hälfte greift täglich darauf zurück. Und Studien zu GitHub Copilot und Co. zeigen signifikante Produktivitätsgewinne, jedenfalls wenn es um Coden, Testen oder Dokumentation geht.

Allerdings gibt es auch Untersuchungen, die auf einen anderen, interessanten Effekt verweisen: Wer KI unreflektiert einsetzt, schreibt zwar zum Beispiel mit geringerem Zeitaufwand mehr Code, aber nicht automatisch besseren.

Was lehrt uns das? KI ist ein Tool wie andere auch – Visual Studio Code, Rider oder ReSharper. Wir selbst sollten aber immer prüfen und bewusst entscheiden, wofür wir die Künstliche Intelligenz einsetzen und was wir sie machen lassen. Denn die Entwicklung von Softwaresystemen ist weit mehr als nur KI.

**„Vorsprung sichern wir uns auch,
aber eben nicht nur durch technische Kompetenz.“**

Unsere Perfektion entsteht nicht durch das bloße Nutzen von Copilot, Claude Code, Maia oder LLMs. Es braucht drei Kompetenzen: Erstens exakte Problemformulierung und präzises Prompting. Zweitens die Fähigkeit, die Ergebnisse der KI kritisch zu hinterfragen, statt sie nur hinzunehmen. Drittens ein tiefes Verständnis der eigenen Domäne, um zwischen cleverer Abkürzung und gefährlicher Halluzination zu unterscheiden. Nicht das Toolset entscheidet, sondern wer mit dessen Hilfe leistungsfähige und robuste Lösungen entwerfen und konstruieren kann.

KI-Tools sind kein Ersatz für Expertise. Aber sie können sie sinnvoll verstärken. Wer die Grenzen der neuen Assistenten im Hinterkopf behält und zugleich mit ihnen experimentiert, baut Systeme, die andere Teams ohne KI (wirtschaftlich) nicht liefern können. Das ist Vorsprung durch Technik heute.

Ich wünsche Ihnen eine schöne Zeit mit der neuen dotnetpro, vor allem aber friedvolle Weihnachtstage und einen guten Start ins neue Jahr!

Herzlichst

Fernando Schneider
Chefredakteur dotnetpro



Benjamin Abt

zeigt, wie KI Softwaretests ermöglicht, die die Softwarequalität erhöhen und Entwicklung beschleunigen (S. 26)



Golo Roden

vermittelt, wo die Vorteile von Event-Sourcing liegen und weshalb sie im KI-Zeitalter so wichtig sein können (S. 104)



INHALT

Focus

Wer sucht, der findet

Der Softwaretest im Überblick

6

Ohne Fleiß kein Preis

Assert.That(this, Is.NiceToHave());

14

Testing the Untestable

Legacy-Code testbar gestalten

19

Tests boostern mit GitHub Copilot und xUnit

KI-gestützte Unit-Tests

26

Inkrementelles Testen mit Power Fx

Testen von Low-Code-/No-Code-Apps
im .NET-Umfeld

34

.NET Stack

NETirol: Dateibasierte .NET-Apps

dotnet run "myfile"

40

Potzblitz in Hardware

Desktop-IoT-Anwendungen mit .NET und C#

43

Data Access: Vektoren auf dem Vormarsch

Der SQL-Server-Spaltentyp „Vector“ mit
Entity Framework Core 10.0

48

Authentifizierung in Nullkommanix

Blazor und Auth0

52

Das Tempo bleibt ordentlich

Neuerungen in Blazor 10.0, Teil 2

60

Artificial Intelligence

KI plus Code – Zukunft oder Hype?

Vibe Coding: Zwischen Experiment und neuem
Entwicklungsparadigma

72

Bye Vibe, hello Vise Coding!

Vibe Coding innerhalb professioneller Leitplanken

78

Programmieren mit Agent Smith

Agentenunterstützte Softwareentwicklung mit
Claude Code

86

1 plus 1 macht mehr als 2

Mendix Artificial Intelligence Assistant

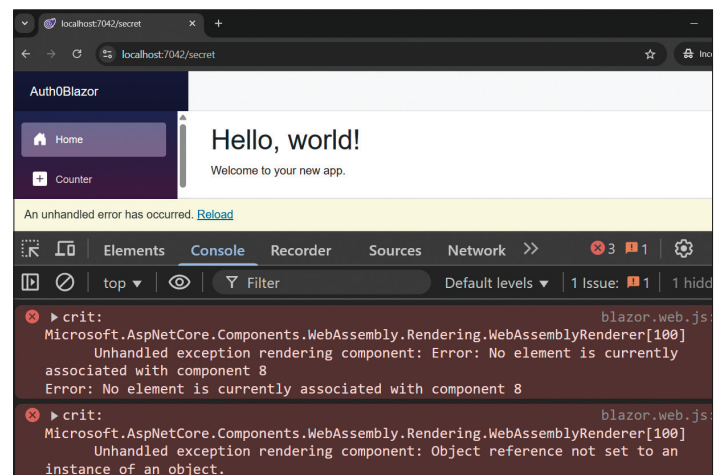
94



Foto: Midjourney

Das Focus-Thema „Software testen“ widmet sich den unterschiedlichen Facetten moderner Testverfahren.

6



Authentifizierung: Wie sich Blazor-Apps sicher und standardbasiert an einen Auth0-Server anbinden lassen.

52

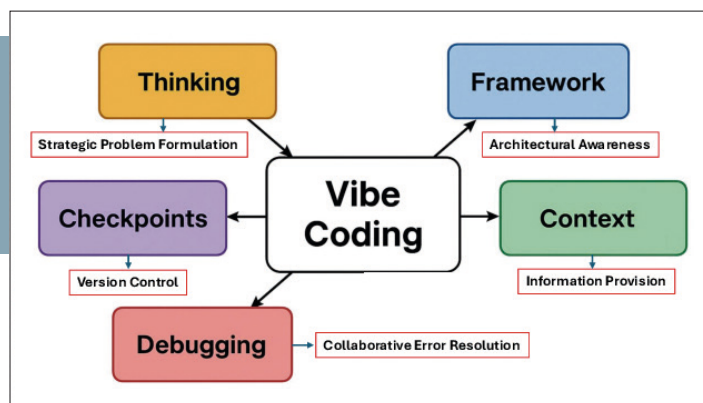
Experten in dieser Ausgabe



Christian Giesswein ist von den mit .NET 10 eingeführten „file based programs“ schwer begeistert. Ein Überblick.

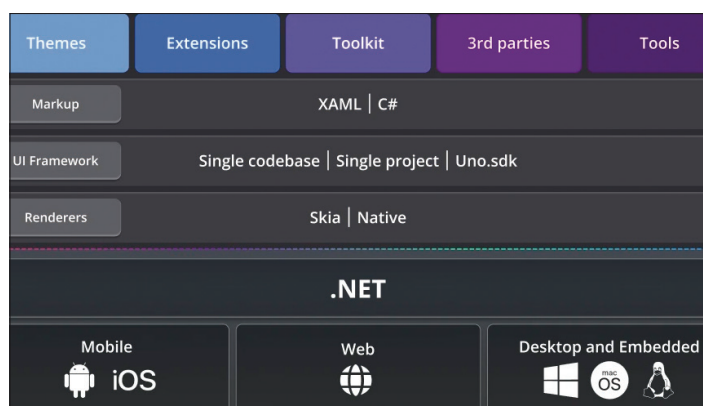


Elena Bochkor widmet sich der „Hyperpersonalisierung“ und ihrem Einfluss auf die UI-Entwicklung.



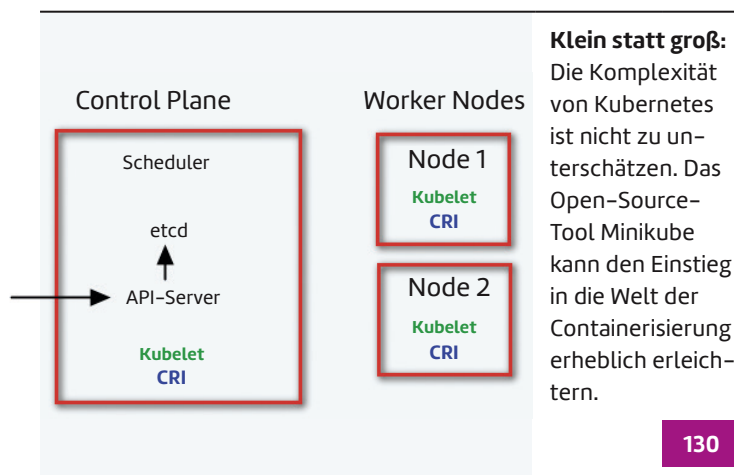
Vibe Coding polarisiert: Die einen lieben es, andere sind skeptisch. Zeit für eine kritische Bestandsaufnahme.

72



Eine für alle: Ein Blick hinter Architektur, Ökosystem, UX-/UI-Strategien und Tooling der Uno Platform.

114



130

Cross Concerns

Dauids Deep Dive: Der Feelgood Manager
P wie „People“

102

Wider das Vergessen in Datenbanken
Event-Sourcing als Perspektivenwechsel

104

What else?

UI-Gestaltung mit Uno

Plattformübergreifend entwickeln mit .NET und XAML

114

Ganz wie für mich gemacht

Personalisierte User Experience

122

Kubernetes light

Einstieg in Minikube

130

Aus AcroForm mach PDF

Mit iText PDF-Dateien aus PDF-Formularen erstellen

135

People & Soft Skills

Time-out: Einfach mal fertig machen

Fokussieren statt verzetteln

138

Knowledge

Gemeinsam stärker

Individuelle Teamtrainings: Wissen als Wettbewerbsvorteil

142



Editorial

3

Inhalt

4

Arbeitsmarkt

140

Lernstoff

144

Impressum/Vorschau/Inserenten

146



Foto: Midjourney

DER SOFTWARETEST IM ÜBERBLICK

Wer sucht, der findet

Testen bedeutet mehr, als nur mit Software zu spielen, bis sie kaputtgeht. Für einen umfassenden Überblick erklärt dieser Artikel grundlegende Begriffe und Vorgehensweisen.

Software als solches ist heute ein wichtiger Teil des Fundaments unserer Welt. Sie durchdringt alles, vom einfachen Wocheneinkauf über die Freizeitgestaltung und unsere Gesundheit bis hin natürlich zu unserem Job. In jedem dieser Bereiche ist Software direkt oder zumindest indirekt beteiligt. Daher ist es nicht übertrieben zu sagen, dass an der Zuverlässigkeit, Stabilität und Sicherheit von Softwaresystemen Existenzen und Leben hängen.

Als Softwareentwicklerinnen und -entwickler spielen wir hierbei eine zentrale Rolle, ist Software doch genau das, was wir erschaffen und pflegen.

Die moderne Softwareentwicklung ist jedoch mit einer Reihe von Herausforderungen konfrontiert, die auch das Testen komplexer machen. Schnelle Release-Zyklen, getrieben durch wirtschaftliche Belange wie Konkurrenzdruck und Kundenerwartung, erfordern effizientere und häufigere Testaktivitäten. Dabei steigt die Komplexität von Systemen, be-

dingt durch ihre zahlreichen Bestandteile und die Vielzahl unterschiedlicher Technologien, die zum Einsatz kommen. Die Software einfach nur einmal auszuprobieren und dann den Kunden zu übergeben, ist daher nicht ausreichend. Vielmehr muss der Softwaretest als integraler Bestandteil eines jeden Entwicklungsprozesses gelebt werden.

Genau darauf gehen wir auf den folgenden Seiten ein, indem wir die verschiedenen Aspekte des Softwaretests beleuchten. Dabei widmen wir uns zunächst den organisatorischen Belangen und wichtigen Fachbegriffen. Anschließend betrachten wir die unterschiedlichen Sichtweisen auf den Begriff des „Tests“, um zu zeigen, wie diese Perspektiven das jeweilige Vorgehen beeinflussen. Hierbei betrachten wir nicht nur die reine Testautomatisierung und somit die sonst üblichen Unit- und Integrationstests, sondern wollen ein breites Verständnis für die oft vernachlässigte Kunst des Softwaretestens schaffen.

Testen? Was ist das?

Betrachtet man das Softwaretesten im professionellen Kontext, handelt es sich um einen systematischen Prozess zur Bewertung der Qualität eines Softwareprodukts [1]. Das primäre Ziel besteht darin, Fehler, Mängel oder Abweichungen von spezifizierten Anforderungen zu identifizieren, bevor die Software ausgeliefert wird. Darüber hinaus dient das Testen der Verifizierung, ob die Software die Bedürfnisse und Erwartungen der Stakeholder erfüllt und die beabsichtigte Funktionalität korrekt implementiert ist. Ein effektiver Testprozess trägt maßgeblich zur Risikominimierung, Kostensenkung und Steigerung der Kundenzufriedenheit bei.

Wie in vielen anderen Fällen auch kommt es dabei maßgeblich darauf an, wie systematisch und formell tatsächlich vorgegangen wird. Als Entwickler bedeutet dies beispielsweise, dass für uns auch ein einfaches Starten der Software nach der Implementierung mit anschließendem Ausprobieren des neuen Features als Test durchgeht. In der Realität geschieht das aber so unsystematisch, dass es sich schwerlich als professionell bezeichnen lassen kann. Doch was kann man tun, um dieses Vorgehen zu professionalisieren? Man muss es so nachvollziehbar und wiederholbar wie möglich gestalten, und genau dafür ist eine formell dokumentierte Teststrategie ein sehr guter Anfang.

Strategien und Pläne

Eine Teststrategie ist eine Guideline, die erläutert, wie die verschiedenen Testarten umgesetzt werden, welche Ziele verfolgt werden, welche organisatorischen Besonderheiten zu berücksichtigen sind und welche Technologien oder Werkzeuge gegebenenfalls warum zum Einsatz kommen sollten (vergleiche auch den Kasten **Die Testprinzipien**). Dies klingt im ersten Moment etwas oberflächlich, kann aber auch sehr spezifisch sein, indem beispielsweise der Umfang beschrieben wird, in dem der Code durch automatisierte Tests abzudecken ist. Genauso können sich darin auch Anweisungen finden, wie mit Fehlern umgegangen werden sollte oder welche Bedingungen aus Sicht der Qualitätssicherung erfüllt sein müssen, um einen Release freizugeben. Eine Teststrategie kann daher selbst schon zu einer Qualitätssteigerung führen, da sie allen Beteiligten verdeutlicht, welche Qualitätsaspekte im Entwicklungsprozess zu berücksichtigen sind.

Ausgehend davon können dann einzelne Projekte, Systeme oder Produkte ihre konkreten Maßnahmen mit einem Testplan weiter ausdetaillieren. Dieser Plan bestimmt, welche konkreten Tests umzusetzen sind, also was ganz genau getestet wird. Hierbei kann es sogar so weit gehen, dass beteiligte Personen, Zeitpunkte und Systembestandteile genannt werden. Daher wird der Testplan seinem Namen gerecht: Er plant die konkreten Testaufwände auf.

Testpläne definieren dazu in aller Regel auszuführende Testfälle (vergleiche den Kasten **Beispiel für einen Testfall**), die wiederum aus einer Testsuite stammen. Eine Testsuite ist eine operative Sammlung von Testfällen für einen konkreten Testgegenstand. Testfälle beschreiben ganz konkret zu prüfende Situationen und Abläufe. Hierzu gehören Bedingungen, die vor dem Test erfüllt sein müssen, wie zum Beispiel, dass be-

stimmte Daten in der Datenbank vorliegen müssen oder ein konkreter Nutzer eingeloggt ist. Darüber hinaus gibt es noch die jeweils auszuführenden Schritte sowie die für den Test vorzunehmenden Eingaben. Weiterhin findet sich entweder zu jedem Schritt oder für den gesamten Testfall eine Beschreibung des erwarteten Zielzustandes. Ziel all dessen ist es, Testfälle immer unter den gleichen Bedingungen durchführen zu können, damit sie auch immer die gleichen Ergebnisse liefern, ausgenommen natürlich, das System ist fehlerhaft und verhält sich unerwartet.

An dieser Stelle sei auch noch mit einem Mythos aufgeräumt: Außerhalb der QA-Abteilungen wird häufig nur von Testern gesprochen. Dies ist aber genauso falsch, wie es falsch wäre, jede Person in der Softwareentwicklung als Programmierer zu bezeichnen. Je nach Komplexität des betrachteten Systems ist es durchaus üblich, dass der Begriff „Tester“ nur die Personen beschreibt, die in einem bestimmten ►

Die Testprinzipien

Das International Software Testing Qualifications Board (ISTQB) hat sieben grundlegende Prinzipien des Testens formuliert, die als Leitlinien für effektive Teststrategien dienen [4]:

- **Testen zeigt die Anwesenheit von Fehlern, nicht deren Abwesenheit:** Tests belegen nur, dass Fehler vorhanden sind, sie können aber nicht beweisen, dass keine Fehler mehr existieren. Selbst wenn keine Fehler gefunden werden, ist dies kein Beweis für absolute Korrektheit.
- **Vollständiges Testen ist unmöglich:** Bei hinreichend komplexen Systemen ist es unmöglich, alle Eingabemöglichkeiten und Vorbedingungen zu prüfen. Risikobasierte Ansätze und Testtechniken sind notwendig, um den Testaufwand zu fokussieren.
- **Frühes Testen spart Zeit und Geld:** Testaktivitäten (sowohl statische als auch dynamische) sollten so früh wie möglich im Softwareentwicklungszyklus beginnen (Shift-Left-Prinzip). Früh gefundene Fehler sind deutlich kostengünstiger zu beheben.
- **Das Wespennest oder Defect Clustering:** Eine kleine Anzahl von Modulen enthält oft die Mehrheit der gefundenen Fehler oder ist für die meisten Betriebsausfälle verantwortlich.
- **Das Pestizid-Paradoxon:** Wenn dieselben Tests immer wieder wiederholt werden, finden sie irgendwann keine neuen Fehler mehr. Testfälle und Testdaten müssen regelmäßig überprüft und angepasst werden, um neue Fehler aufdecken zu können.
- **Die Kontextabhängigkeit:** Die Art und Weise des Testens unterscheidet sich je nach Kontext. Sicherheitskritische Software wird anders getestet als eine E-Commerce-App.
- **Die Abwesenheit von Fehlern ist ein Trugschluss:** Selbst wenn alle spezifizierten Anforderungen getestet und alle gefundenen Fehler behoben wurden, kann ein System immer noch unbrauchbar sein, wenn es die tatsächlichen Bedürfnisse der Benutzer nicht erfüllt oder schwer zu bedienen ist.

● Beispiel für einen Testfall

Nachfolgend findet sich das Beispiel eines manuellen Testfalls, bei dem die Bildersuche von Google geprüft wird. Im Gegensatz zu einem solchen Standardtestfall kann es aber natürlich auch Fälle geben, bei denen das System bewusst unter Last gesetzt oder in eine Fehlersituation manövriert wird, um zu sehen, ob es in solchen Situationen korrekt funktioniert.

Testziel:

Sicherstellen, dass ein Benutzer erfolgreich eine Bildersuche auf google.de durchführen und relevante Ergebnisse erhalten kann.

Voraussetzungen (Preconditions):

- Ein funktionierender Computer mit Internetverbindung ist verfügbar.
- Ein moderner Webbrowser (zum Beispiel Chrome, Firefox) ist installiert.
- Der Browser-Cache ist leer, um personalisierte Ergebnisse zu minimieren.

Testschritte und Ergebnisse: Siehe gleichnamige Tabelle unten.

Nachbedingungen (Postconditions): Der Webbrowser wird geschlossen.

● Testschritte und Ergebnisse

Schritt	Aktion (Testschritt)	Erwartetes Ergebnis	Tatsächliches Ergebnis	Status (Pass/Fail)
1	Öffne den Webbrowser.	Der Browser öffnet sich ohne Fehlermeldung.		
2	Navigiere zum URL: https://www.google.de	Die Google-Startseite wird korrekt geladen. Das Google-Logo und eine Suchleiste sind sichtbar.		
3	Gib den Suchbegriff „Katzenbilder“ in die Suchleiste ein.	Der Text „Katzenbilder“ erscheint ohne Tippfehler in der Suchleiste. Suchvorschläge können erscheinen.		
4	Drücke die Eingabetaste oder klicke auf den Google-Suche-Button.	Eine Suchergebnisseite (SERP) wird geladen. Die Seite zeigt verschiedene Suchergebnisse zum Thema Katzenbilder.		
5	Klicke auf den <i>Bilder</i> -Tab (normalerweise unter der Suchleiste).	Die Ansicht wechselt zur reinen Bilder-Suchergebnisseite.		
6	Überprüfe die angezeigten Ergebnisse.	Die Seite zeigt ein Raster mit Bildern an. Die angezeigten Bilder sind überwiegend Fotos oder Zeichnungen von Katzen.		

Moment einen Test ausführen. Es kann aber genauso gut auch Testanalysten geben, die Testfälle erstellen, oder Testmanager, die strategische Entscheidungen treffen und die organisatorische Gesamtverantwortung tragen. Etwas spezieller, aber gerade in großen Unternehmen unerlässlich sind Testarchitekten, die umfangreiche Testautomatisierungs-Infrastruktur planen, oder Testautomatisierungs-Ingenieure, die solche Pläne in die Tat umsetzen. Ganz zu schweigen von den vielen Spezialistinnen und Spezialisten, die sich mit Security-, Performance- oder Usability-Tests beschäftigen und aufgrund ihrer besonderen Expertise eben jene Begriffe in der Jobbezeichnung tragen.

Manuell versus automatisiert

Mit den vorangegangenen Abschnitten haben wir einen Blick auf die wichtigsten Artefakte und Beteiligten eines Testprozesses geworfen. Testfälle werden in Testplänen organisiert, eine Teststrategie beschreibt, wie dies allgemein geschieht und was man mit den Testergebnissen macht, während Tester diejenigen Personen sind, die eben testen. Das klingt wenig spektakulär und hilft im Alltag erst, wenn man sich die verschiedenen Arten von Tests genauer anschaut. Dabei liegt eine Art der Kategorisierung für uns Entwickler

sofort auf der Hand: Sind Tests automatisiert oder werden sie manuell ausgeführt? In der Realität wird eine gute Teststrategie von beiden Vorgehensweisen Gebrauch machen, denn beide haben ihre Vor- und Nachteile. So bieten sich automatisierte Tests insbesondere für Bereiche an, die man gut isoliert prüfen kann und die sich vergleichsweise leicht wiederholen lassen. Darüber hinaus kann man sich aber auch für die Automatisierung entscheiden, wenn man absolut sichergehen möchte, dass bestimmte Bestandteile eine Art Sicherungsnetz gegen schädliche Fremdeinwirkung bekommen.

Manuelle Tests hingegen eignen sich immer dann, wenn Tests eher komplexere Vorbedingungen oder Zusammenhänge überprüfen. Dies kann natürlich auch dann der Fall sein, wenn das Ergebnis der Prüfung nicht ganz eindeutig ist. Nicht selten wird aber eben auch dann manuell vorgegangen, wenn es gilt, Fehler in der Kombination verschiedener Aktionen aufzudecken. Bei einem solchen explorativen Test folgt man keinem starren Testfall, sondern eher einem natürlichen, unter Umständen sogar leicht chaotischen Weg durch die Applikation. Dabei ist ein professioneller explorativer Test aber nicht ungezielt. Testerinnen und Tester planen auch jene, indem sie das System in einen nachvollziehbaren Zustand versetzen und sich gezielt Testdaten zurechtlegen.

Vorwissen entscheidet

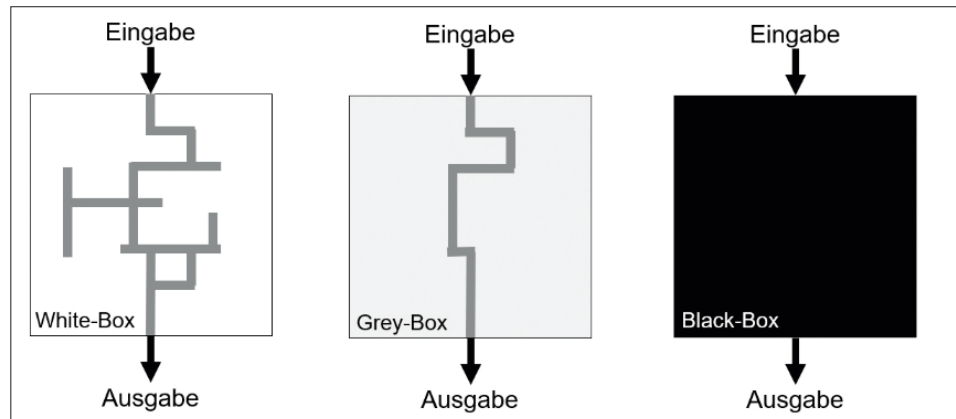
Neben dieser recht oberflächlichen Kategorisierung von Tests gibt es noch eine ganze Reihe weiterer, denen wir uns nachfolgend zuwenden wollen. So kann man zum Beispiel nach dem inhaltlichen Vorwissen über den Testgegenstand unterscheiden, aber ebenso nach dem Abstraktionsgrad beziehungsweise Umfang des Testgegenstands sowie nach der Art der Anforderungen, die zu prüfen sind.

In Bezug auf das Vorwissen zum Testgegenstand unterscheidet man White-Box-, Grey-Box- und Black-Box-Tests (Bild 1). Im Fall von White-Box-Tests hat die testende Person quasi eine transparente Kiste vor sich, in die sie hineinschauen kann. Dies ist beispielsweise bei Unit-Tests häufig der Fall. Bei diesen werden gelegentlich sogar über Mock-Objekte bestimmte Status provoziert oder interne Statusänderungen geprüft. Black-Box-Tests sind hingegen das absolute Gegenteil. Dabei besteht keinerlei Vorwissen, und der Erfolg oder Misserfolg einer Aktion kann nur über von außen wahrnehmbare Parameter bewertet werden. Im Fall eines Methodenaufrufs ist das etwa ein Rückgabewert oder bei einem Test des gesamten Softwaresystems ein Eintrag in eine Datenbank. Wobei Letzteres auch nicht ganz korrekt ist, da man in diesem Fall ja doch einen gewissen Einblick in das System hat. Daher wird in solchen Fällen eher von einer Grey-Box gesprochen.

Auch diese Testarten haben jeweils Vor- und Nachteile. Ohne Vorwissen über den internen Aufbau eines Testgegenstands kann man stabilere Tests schreiben, da diese ja nicht geändert werden müssen, wenn sich der innere Aufbau ändert. Auf der anderen Seite kann es zu unnötig komplexen Aufbauten führen, die ihrerseits hohe Wartungskosten bedingen. Es ist somit wichtig, unterschiedliche Vorgehensweisen einzusetzen und das Testvorhaben genau zu planen. Hierbei ist entscheidend, was der eigentliche Testgegenstand ist, was mit dem konkreten Test bezweckt werden soll und wie stark man die Systembestandteile überhaupt beeinflussen kann. Dies deutet auf eine grobe Daumenregel hin: Je näher man am eigentlichen Quellcode ist, desto eher wird man White-Box-Tests umsetzen, je mehr man größere Sammlungen von Komponenten, Modulen oder ganzen Systemen betrachtet, desto mehr handelt es sich um Black-Boxen. Diese Nähe zum Code wird dabei auch als eine Art Stufe angesehen, von der man sich tiefer in den Code oder weiter davon wegbewegt. Dies führt uns zur nächsten Kategorisierung: den Teststufen.

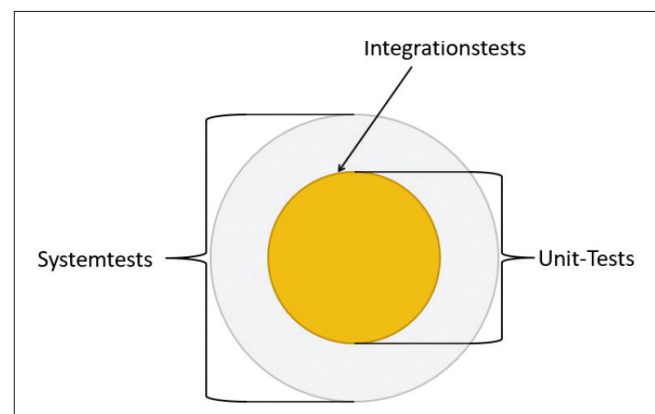
Die Teststufen

Teststufen beschreiben die Ebenen, auf denen Software getestet wird, typischerweise vom kleinsten Modul bis zum kompletten System. Sie sind ein hierarchischer Ansatz, der sicherstellt, dass Fehler frühzeitig erkannt und behoben werden. Die gängigsten Teststufen sind:



Visualisierung des Detailwissens über den Testgegenstand (Bild 1)

- **Komponententest (oder auch Unit-Test):** Dies ist die niedrigste Teststufe, bei der einzelne, isolierte Code-Einheiten (zum Beispiel Funktionen, Methoden oder Klassen) getestet werden. Das Ziel ist, zu überprüfen, ob jede Komponente für sich betrachtet korrekt funktioniert und die spezifizierten Anforderungen erfüllt. Ein Unit-Test konzentriert sich ausschließlich auf den von uns selbst geschriebenen Code, weshalb es sich häufig um White-Box- oder Grey-Box-Tests handelt.
- **Integrationstest:** Bei dieser Stufe werden mehrere Komponenten oder Module zusammengeführt und als Gruppe getestet. Der Fokus liegt auf der Überprüfung der Schnittstellen und Interaktionen zwischen diesen Komponenten, um sicherzustellen, dass sie reibungslos zusammenarbeiten. Dies kann die Verbindung zweier Systeme oder die Interaktion des eigenen Codes mit anderen Systembestandteilen umfassen. Aus diesem Grund handelt es sich häufig um Black-Box- oder Grey-Box-Tests.
- **Systemtest:** Hier wird das gesamte, integrierte Softwaresystem als eine Einheit getestet, um zu überprüfen, ob es die funktionalen und nichtfunktionalen Anforderungen erfüllt. Der Systemtest simuliert oft eine reale Benutzerumgebung und validiert das Endprodukt aus Sicht des Benutzers. Auf dieser Stufe werden die mit Abstand meisten Tests als Black-Box-Tests umgesetzt. ►



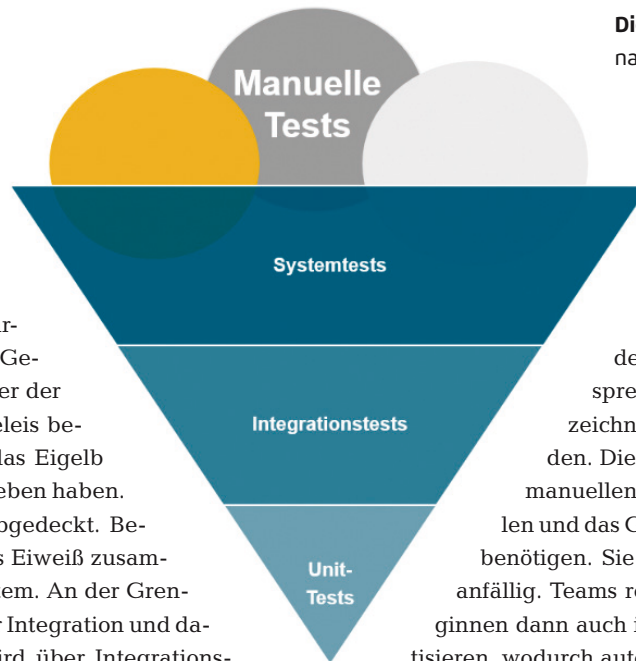
Teststufen in der Darstellung als Spiegelei (Bild 2)

Die Grenzen zwischen den Teststufen können teilweise fließend sein, und leider ist die Definition nicht immer einheitlich. Gerade in Bezug auf Komponenten- und Unit-Tests gibt es verschiedene Sichtweisen, von denen für diesen Artikel nur auf die einfachste zurückgegriffen wurde. Zur Verdeutlichung und als Gedankenstütze kann man sich aber der Symboldarstellung eines Spiegeleis bedienen, siehe **Bild 2**. Dabei ist das Eigelb der Code, den wir selbst geschrieben haben. Dieser wird durch Unit-Tests abgedeckt. Betrachtet man das Eigelb und das Eiweiß zusammen, ergibt sich das Gesamtsystem. An der Grenze des Eigelbs kommt es zu einer Integration und damit einer Trennschicht. Dies wird über Integrationstests geprüft. Hierbei zeigt sich auch wieder, warum es so wichtig ist, eine formalisierte Teststrategie zu besitzen, denn jede der Teststufen prüft etwas anderes. Dabei sind Systemtests gut, um einen Durchstich zu prüfen, aber viel zu aufwendig, um Detailunterschiede festzustellen. Integrationstests wiederum dienen dazu, zu prüfen ob die Systembestandteile korrekt miteinander interagieren; dies bedeutet aber noch lange nicht, dass sie auch funktional korrekt sind.

Teststufen automatisiert

Die Aussagen des vorangegangenen Abschnitts können sowohl auf manuelle als auch automatisierte Tests angewendet werden, denn letztendlich sagt die Nähe zum Quellcode nicht, ob der Test auch gleich automatisiert ist. Automatisierung ist selbst mit Aufwand verbunden, und daher hat zumindest der Autor dieses Artikels während des Studiums nicht selten mit Print-Anweisungen im Code oder temporären Buttons im UI Codestellen manuell geprüft. Solche Vorgehensweisen sind weit verbreitet und gegebenenfalls vorübergehend auch hilfreich. Da sie aber nur schlecht wiederholt werden können, sind sie nicht mehr als vorübergehende Hilfestellungen. Viel besser ist es, sich gleich eines der gängigen Automatisierungsframeworks wie xUnit, net oder NUnit für die in den nachfolgenden Bild Darstellungen unteren Teststufen oder Playwright und Selenium für die dort oberen Teststufen zu bedienen. Dabei kommt es aber ebenfalls auf die Mischung an.

Um Best Practices in Bezug auf die Testautomatisierung zu verdeutlichen, bediente sich Mike Cohn in seinem Buch „Succeeding with agile“ [2] einer Eistüte. Hierbei stellt die Breite einer jeden Teststufe den Umfang an der gesamten Testautomatisierungs-



Die Testautomatisierungseistüte
nach Mike Cohn (**Bild 3**)

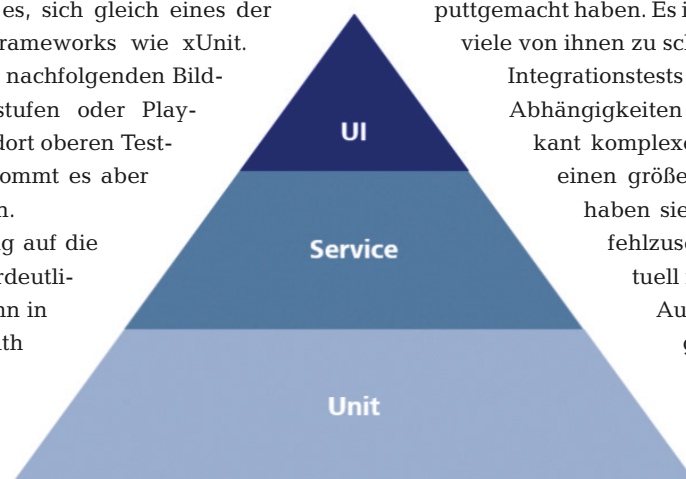
strategie dar. Die Eistüte (**Bild 3**) ist dabei ein Antipattern, das man möglichst vermeiden sollte. In diesem Fall werden die Tests eher reaktiv automatisiert, indem manuelle Tests mit einem entsprechenden Framework aufgezeichnet und wieder abgespielt werden. Dies ist deshalb nachteilig, da diese manuellen Tests eher Systemtests darstellen und das Gesamtsystem für die Ausführung benötigen. Sie sind somit langsam und fehleranfällig. Teams realisieren dies natürlich und beginnen dann auch in tieferen Schichten zu automatisieren, wodurch automatisierte Integrationstests und eventuell auch vereinzelte Unit-Tests entstehen.

Als Gegenentwurf schlägt Cohn die Testautomatisierungspyramide vor. Wie in **Bild 4** zu sehen, spricht er jedoch nicht von den Schichten System, Integration und Unit. Stattdessen betrachtet er UI, Service und Unit, was sich leicht auf die im Deutschen gebräuchlichen Begriffe übertragen lässt, da Tests des UI eben das System brauchen und Integrationstests nicht selten gegen die APIs von Services geschrieben werden, vergleiche **Bild 5**.

So viel zur Klärung der Begrifflichkeiten und den Unterschieden dieses Artikels gegenüber der Originalquelle. Welchen konkreten Vorteil hat denn nun aber eine Pyramide gegenüber einer Eistüte? Dadurch, dass die Teststrategie von Beginn an Unit-Tests vorzieht, kann viel genauer und auch schneller getestet werden. Unit-Tests prüfen immer genau den kleinen Ausschnitt des Codes, für den sie verfasst wurden. Sie dürfen keine anderen Systembestandteile wie Datenbanken, Dateisysteme oder Netzwerke voraussetzen, denn dies würde sie automatisch zu Integrationstests machen. Unit-Tests sind also schnell, können leicht wiederholt werden und sagen uns damit frühzeitig, ob wir etwas kaputtgemacht haben. Es ist somit ohnehin recht einfach, viele von ihnen zu schreiben und zu verwalten.

Integrationstests sind aufgrund ihrer externen Abhängigkeiten dahingehend schon signifikant komplexer. Da sie in aller Regel auch einen größeren Codebereich überprüfen, haben sie auch deutlich mehr Gründe, fehlerzuschlagen, die sich dann eventuell nicht einfach zuordnen lassen. Auf Systemtests trifft dies in noch größerem Umfang zu, da ihr Codebereich und ihre Abhängigkeiten noch umfassender sind.

Man kann also sagen, dass man mit Unit-Tests



Die Testautomatisierungspyramide nach Cohn (**Bild 4**)

wie mit einem Mikroskop über den Code geht. Bei Integrationstests bedient man sich einer Lupe, und bei Systemtests holt man vielleicht sogar schon das Fernglas heraus. Damit wird dann auch klar, warum man nun nicht nur auf Unit-Tests setzen kann. Denn nur weil etwas im Kleinen funktioniert, heißt dies nicht, dass alle Bestandteile perfekt ineinandergreifen. Es ist auch wichtig, umfangreichere Abläufe zu prüfen, um zu sehen, inwieweit konkrete Features korrekt umgesetzt sind und das Gesamtsystem auch wirklich tut, was es tun soll.

Besonders deutlich wird dies bei Bestandssystemen mit einem hohen Anteil an Legacy-Code oder in Fällen, wo die Abläufe sehr eindeutig sind, ja man möglicherweise sogar nur Frameworks oder Cloud-Dienste miteinander koppelt. In diesen Fällen hat in den vergangenen Jahren der Testautomatisierungsdiamant (Bild 6) an Bedeutung gewonnen. Er bevorzugt die Umsetzung auf Ebene der Integrationstests und vernachlässigt Unit-Tests, da diese entweder gar nicht möglich oder von geringem Mehrwert sind.

Tests über Tests

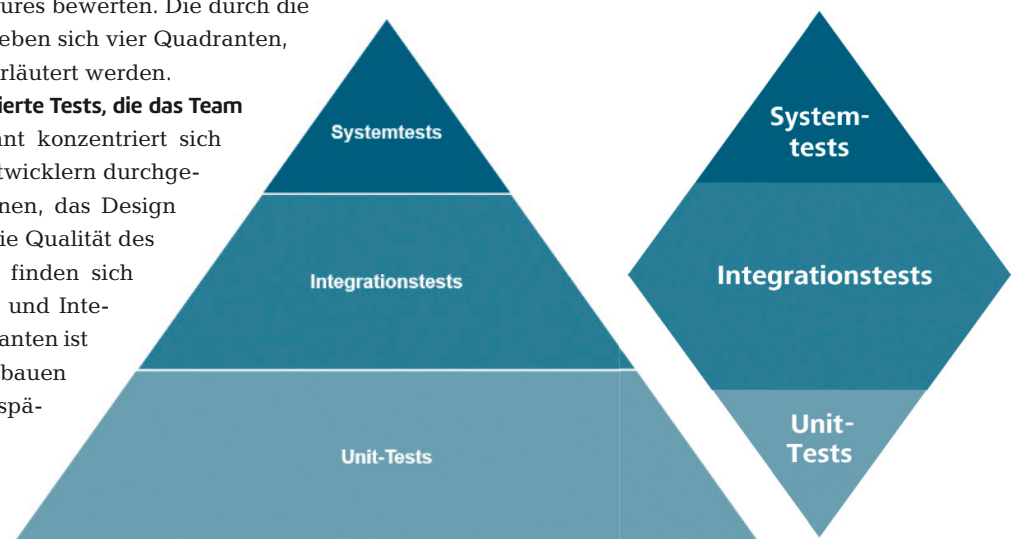
Nun haben wir uns sehr ausgiebig mit dem Vorwissen über den Testgegenstand und die Ausgestaltung verschiedener Teststufen beschäftigt. Es gibt aber tatsächlich noch viele weitere Sichten auf das Thema und insbesondere viele unterschiedliche Spezialfälle des Testens. Hierzu wurde eine sehr umfangreiche und empfehlenswerte Übersicht von Lisa Crispin und Janet Gregory in ihrem Buch „Agile Testing: A Practical Guide for Testers and Agile Teams“ [3] vorgestellt. Sie bieten eine nützliche Kategorisierung von Testaktivitäten in agilen Projekten und helfen dabei, die verschiedenen Arten von Tests und deren Ziele im Entwicklungsprozess zu visualisieren und zu planen. Dabei bedienen sie sich in der Darstellung (vergleiche Bild 7) zweier Achsen: Die horizontale Achse stellt dar, wie technologieorientiert oder geschäftsorientiert die eingetragenen Tests sind, also ob sie eher Code oder aber Funktionalität prüfen. Die vertikale Achse trennt die Tests danach, ob sie primär das Team unterstützen und somit die Entwicklung anleiten, oder ob sie das Produkt kritisieren und damit fertige Features bewerten. Die durch die Anordnung dieser Achsen ergeben sich vier Quadranten, die nachfolgend ausgiebiger erläutert werden.

- **Quadrant 1: Technologieorientierte Tests, die das Team unterstützen:** Dieser Quadrant konzentriert sich auf Tests, die primär von Entwicklern durchgeführt werden und dazu dienen, das Design der Software zu leiten und die Qualität des Codes sicherzustellen. Hier finden sich die bereits erwähnten Unit- und Integrationstests. Ziel des Quadranten ist es, das System korrekt aufzubauen und als Sicherheitsnetz für spätere Änderungen zu dienen.

- **Quadrant 2: Business-orientierte Tests, die das Team unterstützen:** Dieser Quadrant umfasst Tests, die dazu dienen, die Entwick-

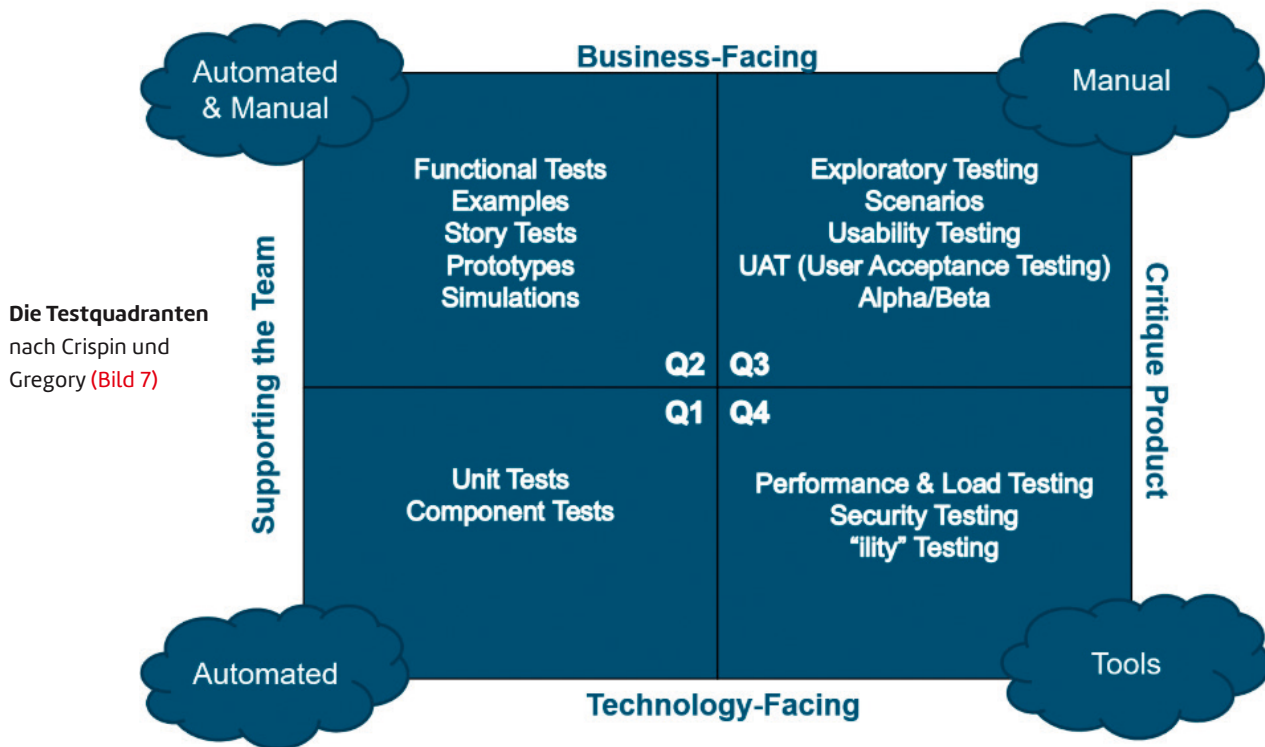
lung anzuleiten und sicherzustellen, dass die Software die Erwartungen der Stakeholder erfüllt. Diese Tests sind oft kundenorientiert und beschreiben, wie sich das System verhalten soll. Beispiel-basierte Tests beziehungsweise Behavior-Driven Development (BDD) werden in einer domänen-spezifischen Sprache wie Gherkin erstellt und beschreiben das erwartete Verhalten des Systems aus Anwendersicht. Derartige Tests können automatisiert werden und dienen auch als lebende Dokumentation. Funktionale Tests hingegen überprüfen, ob die Software die spezifischen funktionalen Anforderungen erfüllt. Akzeptanztests werden oft von den Kunden oder Produktverantwortlichen definiert und durchgeführt, um sicherzustellen, dass das System die Geschäftsanforderungen erfüllt und akzeptabel ist. Diese Tests konzentrieren sich darauf, das richtige System aufzubauen.

- **Quadrant 3: Business-orientierte Tests, die das Produkt evaluieren:** Dieser Quadrant befasst sich mit Tests, die das Produkt bewerten, um sicherzustellen, dass es für den Endbenutzer wertvoll und nutzbar ist. Hierbei handelt es sich oft um manuelle oder explorative Tests, die aus der Perspektive des Benutzers durchgeführt werden. Hier finden sich auch Usability-Tests, die überprüfen, wie einfach und intuitiv das System für die Benutzer zu bedienen ist. User Acceptance Testing (UAT) umfasst hingegen formelle Tests durch Endbenutzer oder Kunden, um die Akzeptanz des Systems vor der Freigabe zu bestätigen. Das Ziel ist hier, sicherzustellen, dass das System den Bedürfnissen der Benutzer entspricht und einen Mehrwert bietet.
- **Quadrant 4: Technikgestützte Tests, die das Produkt evaluieren:** Dieser Quadrant beinhaltet Tests, die technische Aspekte des Systems bewerten, die nicht direkt die Funktionalität betreffen, aber für die Gesamtqualität entscheidend sind. Diese Tests sind oft automatisiert:
 - **Performance-Tests** überprüfen die Geschwindigkeit, Stabilität und Reaktionsfähigkeit des Systems unter verschiedenen Lastbedingungen (Beispiele dafür sind Lasttests und Stresstests).



Die Testautomatisierungspyramide, umgesetzt auf im deutschsprachigen Raum gebräuchliche Begriffe (Bild 5)

Der Testautomatisierungsdiamant (Bild 6)



- **Sicherheitstests** hingegen identifizieren Schwachstellen und gewährleisten den Schutz von Daten und Systemressourcen.
- **Skalierbarkeitstests** wiederum überprüfen, wie gut das System mit wachsenden Benutzerzahlen oder Datenmengen umgehen kann.
- **Belastungstests** testen das System unter extremen Bedingungen, um dessen Grenzen zu ermitteln.
- **Wartbarkeitstests** schließlich beziehen sich auf die Überprüfung der Codequalität, um sicherzustellen, dass das System leicht zu warten und weiterzuentwickeln ist. Diese Tests stellen sicher, dass das System robust, sicher und performant ist.

Die Testquadranten betonen, dass ein ausgewogener Testansatz alle vier Bereiche abdecken sollte, um eine hohe Softwarequalität zu gewährleisten. Sie fördern die Zusammenarbeit zwischen Entwickler:innen, Testexpert:innen sowie Business-Stakeholdern und helfen Teams dabei, ihre Teststrategien an die spezifischen Anforderungen des Projekts anzupassen. Mehr noch zeigen sie auch uns für diesen Artikel, dass das Thema viel umfangreicher ist, als wir es hier in der Kürze behandeln können.

Frühes Testen rechnet sich

Nun, da wir einen tieferen Einblick in die Thematik bekommen und ein Gefühl für dessen Komplexität gewonnen ha-

● **Tabelle 1: Kosten und Nutzen von Tests**

Aspekt	Kosten ohne Tests	Nutzen durch Tests
Fehlerbehebung	Exponentiell steigende Kosten, je später der Zeitpunkt der Entdeckung	Frühe Fehlererkennung, geringere Behebungskosten
Reputation	Reputationsschäden, Vertrauensverlust bei Kunden	Verbesserter Ruf, gesteigertes Kundenvertrauen
Kundenbindung	Unzufriedene Kunden, Abwanderung	Höhere Kundenzufriedenheit und -loyalität
Rechtliche Risiken	Mögliche Klagen, finanzielle Strafen	Minimierung rechtlicher Risiken
Betriebliche Ausfälle	Hohe Ausfallzeiten, Produktivitätsverlust	Stabilere Systeme, geringere Ausfallzeiten
Entwicklungs-Effizienz	Häufige Nachbesserungen, ineffiziente Entwicklungszyklen	Optimierte Entwicklungsprozesse, weniger Nacharbeit
Innovation	Angst vor Veränderungen durch potenzielle Fehler	Möglichkeit für Innovationen durch stabile und zuverlässige Basis

ben, sollten wir uns noch dem Elefanten im Raum zuwenden. Denn seien wir einmal ehrlich: Eigentlich weiß jeder, der einige Zeit lang Software entwickelt, dass das Testen wichtig ist. Schnell merkt man auch, dass das Testen gar nicht so einfach ist, wie es zu sein scheint. Und dennoch arbeitet nicht jedes Team und nicht jedes Unternehmen mit professionellen Testvorgehensweisen. Die Gründe dafür mögen darin liegen, dass Tests auf den ersten Blick eben nur Bugs finden, diese aber nicht ausschließen können, und dass ein umfangreiches Testvorgehen auch entsprechend viel Geld kostet.

Dabei ist es aber ein Trugschluss, zu glauben, dass das Testen nur Kosten verursacht. Vielmehr ist es eine Investition, die sich langfristig auszahlt. Die Kosten für die Behebung eines Fehlers steigen deutlich, je später er im Entwicklungszyklus entdeckt wird (Bild 8). Ein Fehler, der während der Anforderungsanalyse gefunden wird, ist in der Regel wesentlich günstiger zu beheben als ein Fehler, der erst nach der Auslieferung an den Kunden auffällt. Letzterer kann nicht nur zu finanziellen Verlusten durch Nachbesserungen und Garantieleistungen führen, sondern auch den Ruf des Unternehmens schädigen und im schlimmsten Fall rechtliche Konsequenzen nach sich ziehen.

Es sind aber nicht nur rein finanzielle Kosten, die es zu berücksichtigen gilt, vergleiche Tabelle 1. Auch Verzögerungen können zu weiteren Missverständnissen führen. Findet der Entwickler einen Bug, während er noch programmiert, braucht er keine Abstimmung mit anderen Personen. Dies wird durch ein entsprechendes Netz aus automatisierten Tests sichergestellt. Manuelle Nachtests dagegen führen schon zu größeren Kommunikationsaufwänden, da ganz genau geklärt werden muss, unter welchen Bedingungen der Bug aufgetreten ist. Diese Bedingungen müssen dann erst einmal aufwendig nachgestellt werden.

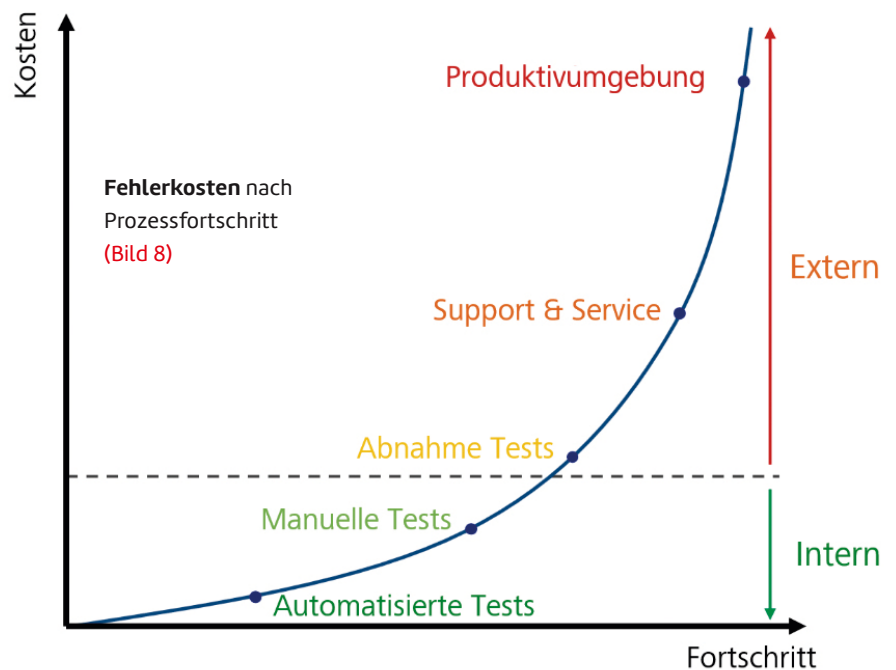
Es gibt aber auch Ausnahmen von den in Bild 8 gezeigten Zusammenhängen. Dazu gehören beispielsweise sogenannte Canary Tests. Dies ist eine besondere Form von Deployment- und Test-Strategie, bei der neue Features einer Software zunächst einem kleinen Teil der Nutzerbasis zur Verfügung gestellt wird. Treten bei dieser Gruppe Fehler auf, wird die Funktionalität wieder zurückgerollt. Die Bezeichnung geht zurück auf die Kanarienvögel, die einst als Frühwarnsystem in Minen und Bergwerken eingesetzt wurden.

Zusammenfassung

Ein professioneller Testprozess trägt maßgeblich zur Risikominimierung bei und schützt vor unerwarteten Ausgaben. Er stellt sicher, dass die Software die Erwartungen der Nutzer erfüllt, was wiederum zu höherer Kundenzufriedenheit und -bindung führt. Dies sind indirekte, aber sehr wirkungsvolle Ersparnisse und Gewinne, die weit über die anfänglichen Testkosten hinausgehen.

Dabei reicht es nicht, die Tests ein einziges Mal auszuführen. Alle hier beschriebenen Testarten und Vorgehen dienen auch dazu, die Regression abzudecken. Damit gemeint sind Veränderungen am Softwaresystem, die bestehende Funktionalität in Mitleidenschaft ziehen können.

Regressionstests sind also ein Sicherungsnetz, um die Zuverlässigkeit und Stabilität der Software über den gesamten Entwicklungszyklus hinweg zu gewährleisten. ■



- [1] Andreas Spillner, Tilo Linz, *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard (7., überarbeitete und aktualisierte Auflage)*, 2024, dpunkt.verlag, ISBN 978-3-98889-005-4
- [2] Mike Cohn, *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley, 2009, ISBN 978-0-321-57936-2
- [3] Lisa Crispin, Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009, ISBN 978-0-321-53446-0
- [4] International Software Testing Qualifications Board (ISTQB). (2024). *Certified Tester Foundation Level Syllabus v4.0.1*, https://istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf



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

ASSERT.THAT(THIS, IS.NICETOHAVE());

Ohne Fleiß kein Preis

Die Devise lautet: Testen, testen, testen.

Unit-Tests sind aus der heutigen Zeit nicht mehr wegzudenken. Sie können vieles vereinfachen und beschleunigen, und sie tragen wesentlich dazu bei, fremden Code oder auch Ihren eigenen zu verstehen.

Es gibt ein Sprichwort: Ist Ihr eigener Code älter als vier Wochen, ist er wie fremder Code. In so einem Fall ist ein guter Unit-Test unglaublich hilfreich, um zu verstehen, was in aller Welt diese seltsame Methode da eigentlich macht.

Den Umgang mit einer unbekannten Programmbibliothek zu erkunden, fällt mit den „richtigen“ Tests ebenfalls um vieles leichter. Vielleicht kennen Sie noch aus der Sesamstraße die „Was passiert dann?“-Maschine. Nichts anderes machen Sie mit einem passenden Unit-Test.

Was einen guten, sinnvollen oder passenden Unit-Test ausmacht, und ob es sich um ein „Nice to have“ oder doch eher ein „Must have“ handelt, soll im Folgenden näher unter die Lupe genommen werden.

Geschichte

Laut Wikipedia [1] reichen die Anfänge der ersten Unit-Tests bis ins Jahr 1956 zurück. Wie so häufig sind auch hier militärische Belange Triebfeder der Entwicklung gewesen. Zu dieser Zeit war allerdings noch nicht von testgetriebener Entwicklung die Rede; vielmehr wurde im Nachhinein anhand von Eingabeparametern geprüft, ob einzelne Komponenten das erwartete Ergebnis zurücklieferten. Immerhin ging es hier bereits um das Testen von Programmbestandteilen. Es fand also offenbar bereits eine Zerlegung in einzelne Unterprobleme statt.

Im Jahr 1964 wurde dann in der Raumfahrt ein ähnlicher Ansatz gewählt. Hier wurden einzelne Programmteile getestet und nach erfolgreichen Tests mit anderen Bestandteilen zusammengeführt.

Kent Beck beschrieb 1989 ein Test-Framework für Smalltalk [2] und entwickelte 1997 zusammen mit Erich Gamma [3] JUnit [4], das im selben Jahr auch veröffentlicht wurde. Erich Gamma war übrigens als einer der „Gang of Four“ am Buch „Design Patterns – Elements Of Reusable Object-Oriented Software“ als Mitautor beteiligt.

Mit der Entwicklung des .NET Frameworks wurde in den frühen 2000er-Jahren NUnit entwickelt. Hierbei handelt es sich um eine Portierung von JUnit. Um 2007 herum schrieben die ursprünglichen NUnit-Entwickler das Framework komplett neu, woraus dann xUnit entstand.

MSTest [6] ist seit Visual Studio 2005 Bestandteil der Entwicklungsumgebung. Visual Studio 2019 machte Live Unit Testing verfügbar; hier werden bereits bei Codeänderungen entsprechende Unit-Tests automatisch durchgeführt.

Was ist überhaupt ein Unit-Test?

Ein Unit-Test prüft einen isolierten Bestandteil einer Anwendung. Das sind in der Regel die öffentlichen Methoden einer Klasse der Anwendung. Das heißt, pro Klasse in der Anwendung gibt es eine Datei, die die Tests zu dieser Klasse enthält. Oder etwas allgemeiner formuliert: Ein Unit-Test prüft die kleinste testbare Einheit einer Anwendung auf ihre Funktionstüchtigkeit.

Im vorigen Absatz sind bereits zwei der Probleme benannt, die gelöst sein müssen, damit ein Unit-Test geschrieben werden kann: „Kleinste Einheit“ impliziert, dass sich die Anwendung überhaupt in kleine Einheiten zerlegen lässt. Wenn Sie jetzt an Dinge wie Clean Code und Single Responsibility Principle denken, liegen Sie genau richtig. Und testbar soll diese kleinste Einheit auch noch sein. Für erfolgreich verlaufende Unit-Tests gibt es also bestimmte strukturelle und architektonische Herausforderungen, die es im Rahmen der Entwicklung zu beachten und zu meistern gilt.

Welche Eigenschaften muss ein Unit-Test haben?

- **Deterministisch:** Bei gleichen Eingabeparametern muss immer dasselbe Testresultat erzielt werden.
- **Korrekt:** Das im Test formulierte erwartete Ergebnis muss selbst natürlich fehlerfrei sein und zu den Anforderungen passen.
- **Eindeutig:** Das Testergebnis darf keine weitere Interpretation benötigen.
- **Isoliert:** Ein Test darf einen anderen nicht beeinflussen.
- **Schnell:** Niemand mag erst Kaffee holen gehen, um das Testresultat abzuwarten.
- **Unabhängig:** Datenbankzugriffe, Netzwerkverbindungen oder Dateisystemzugriffe dürfen für den Test nicht notwendig sein. Ebenso soll der Test nicht auf spezielle Anwendungskonfigurationseinstellungen angewiesen sein. Ort, Datum oder Zeit der Ausführung dürfen sich nicht auf das Testresultat auswirken. Das heißt, in der Build-Pipeline auf dem Build-Server mitten in der Nacht muss sich der Test genauso erfolgreich ausführen lassen wie am Montagmorgen am lokalen Entwicklungsrechner.
- **Sprechend:** Wie an allen anderen Stellen im Code auch sollten Namen verwendet werden, die genau das ausdrücken, was der Code macht.
- **Wartbar:** Auch Tests sollten den Prinzipien für sauberen Code folgen. Wobei „Don't repeat yourself“ (DRY) hier etwas relativ zu sehen ist: Es gilt abzuwägen, ob der Arrange-Teil des Tests zwecks besseren Verständnisses per Copy and Paste aus einem anderen ähnlichen Test kommen kann.

oder ob entsprechende *Arrange*-Methoden aus dem Test ausgelagert werden.

- **Umfassend:** Der Bereich möglicher Eingabeparameter einer Funktion muss abgedeckt sein. Dazu gehören Grenzfälle wie kleinster und größter Wert, der verarbeitet werden kann, zu erwartende Eingaben innerhalb des möglichen Bereichs sowie invalide Eingaben, die formell zwar zulässig sind, vom System aber nicht verarbeitet werden können.

Darüber hinaus hat es sich bewährt, wenn die Testprojekte die gleiche Struktur haben wie die Anwendung selbst und die Testklassen mit dem Namen der zu testenden Klasse beginnen und um ein entsprechendes Postfix ergänzt werden. Das heißt, einen Test zu einem *AuthorizationService.cs* im Project *App.Services* unter dem Namensraum *Authorization* sollten Sie in einem Testprojekt namens *App.Services.Tests* im Namensraum *Authorization* in der Klasse *AuthorizationServiceTests.cs* finden.

Warum Unit-Tests?

Was passieren kann, wenn Tests unvollständig oder unpassend sind, können Sie unter [7] nachlesen: Die Trägerrakete Ariane 5 verwendete bewährte und sicherlich auch umfassend getestete Software aus dem Vorgängermodell. Nicht ausreichend bedacht wurde jedoch das nun viel größere Beschleunigungsvermögen, was letztendlich zu einem arithmetischen Überlauf und zur Selbstzerstörung der Rakete führte. Dieses und einige weitere Beispiele für sehr schwere Softwarefehler finden Sie auch bei Wikipedia [8].

Ausschlaggebender Beweggrund für das Schreiben eines Unit-Tests muss aber nicht immer gleich ein mögliches Worst-Case-Szenario sein. Es gibt eine ganze Reihe andere Gründe, die Unit-Tests ihre Berechtigung geben:

- **Zeitersparnis:** Eine neue Funktionalität kann einfach schneller entwickelt werden, wenn nicht jedes Mal die komplette

Anwendung neu gestartet werden muss, um den eben erfundenen Code zu prüfen. Erforderliche Eingabeparameter müssen nicht manuell über eine Benutzerschnittstelle erfolgen, sondern werden, einmal codiert, immer wieder vom Unit-Test an die zu prüfende Methode weitergegeben. Im Fehlerfall müssen Sie sich nicht erst durch die halbe Anwendung debuggen, sondern können schnell problembehaftete Stellen finden oder eben auch Bestandteile ausschließen, die nicht Ursache des Fehlers sein können.

- **Zusammenarbeit:** Erstellen der Benutzerschnittstelle und Implementation und Test neuer Methoden können erfolgen, während notwendiger Infrastruktur-Code (Controller, Datenzugriff) noch in Arbeit ist.
- **Sicherheit:** Vollständige Tests ermöglichen stressfreie Refaktorisierungen. Wenn nach einer Umstrukturierung keine Tests fehlschlagen, ist sichergestellt, dass die Anwendung weiterhin funktioniert.
- **Dokumentation:** Ein klar formulierter Unit-Test ist immer die aktuellste und zutreffendste Dokumentation einer Methode. Diese Art der Dokumentation muss sich zwangsweise an Änderungen im Code anpassen und veraltet damit nicht – im Gegensatz zu externen Dokumenten, Dokumentationskommentaren oder anderen Kommentaren im Quelltext.
- **Langfristige Qualitätssicherung:** Und hier liegt der Fokus auf „langfristig“. Oder können Sie sich noch erinnern, dass Code, den Sie vor drei Jahren geschrieben haben, nun Auswirkungen im Sinne von unerwünschten Seiteneffekten auf Ihre gerade frisch entwickelten Methoden haben kann?

Wann sind Unit-Tests unangebracht?

Sinnlos sind Unit-Tests immer dann, wenn eine Klasse keine eigene Logik enthält oder nur Methodenaufrufe weiterleitet oder nur bereits anderweitig getesteten Code enthält. Ein Web-API-Controller, der die Aufrufe lediglich an Command- oder Query-Handler weiterleitet und deren Ergebnisse ►

• Listing 1: Einfaches Beispiel einer grundlegenden testbaren Architektur

```
public interface IValidator<T>
{
    void AcceptOrThrow(T item);
}

public class PersonManager(IValidator<Person>
    validator)
{
    public void Add(Person person)
    {
        validator.AcceptOrThrow(person);
        ...
        Persons.Add(person)
    }
}

[Test]
public void AddsPerson()
{
    // arrange
    var validatorMock = new Mock<IValidator>();
    var personManager =
        new PersonManager(validatorMock.Object);

    // act
    personManager.Add(new Person());

    // assert
    Assert.That(personManager.Persons.Count,
        Is.AtLeast(1));
}
```

zurückgibt, verdient keinen eigenen Unit-Test. Ein reines DTO oder eine Entitäts-Klasse ohne eigene Methoden braucht keinen Test, was wollen Sie da auch testen?

Wenn die Anwendung nur eine sehr kurze Lebensdauer hat, über wenig bis gar keine eigene Logik verfügt und niemals irgendwelchen Refaktorisierungen unterworfen werden wird – und damit unterm Strich der langfristige Gewinn von Unit-Tests nicht genutzt werden kann –, sollte auch infrage gestellt werden, ob Tests überhaupt sinnvoll sind.

Testbare Anwendung

Um eine Klasse isoliert testen zu können, müssen ihre Abhängigkeiten für einen Unit-Test ersetzt werden können. Ansonsten testen Sie sowohl die Klasse als auch ihre Abhängigkeiten. Und dann ist das kein Unit-Test mehr, sondern mindestens ein Integrationstest.

Dependency-Injection [9] lautet das Zauberwort. Die Klasse besorgt sich ihre Abhängigkeiten nicht selbst, sondern bekommt sie von außen übergeben. Damit erreicht man nicht nur eine lose Kopplung der Komponenten untereinander, sondern kann nun auch die realen Abhängigkeiten für einen Unit-Test ersetzen.

Damit man nicht für jede Abhängigkeit zwei Versionen (eine für den Test, eine für die Anwendung selbst) schreiben muss, injiziert man nicht die konkrete Abhängigkeit, sondern nur ihren Vertrag, ihre Schnittstelle. Für den Test kann dann die Implementierung der Schnittstelle über ein Mocking-Framework wie zum Beispiel Moc [10] erfolgen. Als einfaches Beispiel einer grundlegenden testbaren Architektur soll der Pseudo-Code in Listing 1 erhalten. Hier gibt es einen *PersonManager*, der neue Mitarbeiter entgegennimmt, die eingegebenen Daten validiert und dann die Daten weiterverarbeitet. Der Validator wird per Dependency-Injection bereitgestellt, im Test wird die *IValidator*-Schnittstelle durch einen Mock ersetzt. Mit dem Unit-Test soll sichergestellt werden, dass das Einfügen von Daten den Datenbestand entsprechend erhöht. Ob der Validator irgendetwas validiert oder nicht, ist in die-

Und eine Nebenbemerkung: Der Mock in Listing 1 ist eigentlich kein Mock, sondern ein Stub, da der Mock nichts zurückgibt und lediglich ein Stellvertreter-Objekt darstellt. Ebenso wäre ein Logger eher ein Stub, denn dieser muss nichts machen, sondern nur da sein, um eine Klasse, die etwas protokolliert, im Test benutzen zu können.

SetUp – TearDown

So ziemlich jedes Test-Framework bietet neben den Testmethoden selbst Methoden zum Einrichten der Abhängigkeiten (zum Beispiel das *[SetUp]*-Attribut in NUnit) und zum Aufräumen (zum Beispiel *[TearDown]*-Attribut in NUnit).

Schon einem kleinen Kind versuchen die Eltern das Aufräumen nach dem Spielen mehr oder weniger erfolgreich anzugewöhnen und sind dankbar, wenn die Erziehung auch in dieser Hinsicht erfolgreich verläuft. Genauso dankt Ihnen Ihr Rechner und dessen Speicher, wenn nach getaner Arbeit aufgeräumt wird.

Die *SetUp*-Methode wird in der Regel vor jedem einzelnen Test ausgeführt, die *TearDown*-Methode danach. Es gibt auch Varianten dieser Attribute, die für die gesamte Testklasse gelten (beispielsweise *[OneTimeSetUp]* und *[OneTimeTearDown]* in NUnit). Diese werden vor beziehungsweise nach der Ausführung aller Tests der Testklasse ausgeführt.

Es bietet sich daher an, immer wiederkehrende Arrange-Teile eines Tests in eine *SetUp*-Methode auszulagern. Ressourcen, die dort belegt werden, sollten in der *TearDown*-Methode dann wieder freigegeben werden.

Entwicklungszyklus einer testgetriebenen Entwicklung

Von testgetriebener Entwicklung spricht man, wenn immer zuerst Tests geschrieben werden, um dann die entsprechenden Funktionalitäten mithilfe der Tests korrekt zu implementieren. Sie müssen also zunächst wissen, was die Klasse können soll, eine entsprechende Schnittstelle definieren, die genau das beschreibt; eine erste Implementierung der Klasse

● Tabelle 1: Inhalt einer Testmethode

Arrange	Act	Assert
Bereitstellen aller Abhängigkeiten in Form von Mocks beziehungsweise Stubs (optional, nur wenn es Abhängigkeiten gibt); Instanzieren des zu testenden Subjekts.	Das Test-Subjekt seine Arbeit verrichten lassen.	Überprüfen des Ergebnisses, Vergleich des erwarteten Ergebnisses (<i>expected</i>) mit dem tatsächlichen (<i>actual</i>).

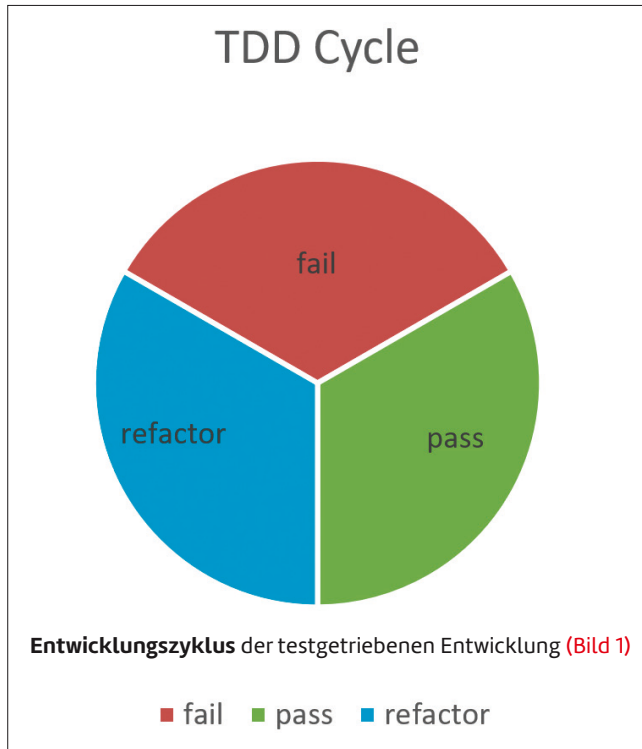
sem Zusammenhang vollkommen unerheblich. Dafür schreiben Sie dann Tests für den Validator, die sich unabhängig vom hier skizzierten *PersonManager* erstellen lassen.

Sie haben in Listing 1 bereits den grundlegenden Aufbau eines Unit-Tests gesehen. Behalten Sie diese sogenannten drei A (Arrange–Act–Assert) konsequent bei. Ihre Kollegen werden es Ihnen danken, und auch die grundsätzliche Verständlichkeit über alle Tests hinweg wird wachsen. Zum besseren Überblick sei hier auch auf Tabelle 1 verwiesen.

erstellen, in der alle Methoden eine *NotImplementedException* werfen, denn der erste Test muss fehlschlagen, wenn man dem Mantra der testgetriebenen Entwicklung folgt. Das ist so weit auch nachvollziehbar – da Sie testgetrieben entwickeln, steht zuerst der Test da, dann die Implementierung.

Erst wenn der erste Test fehlgeschlagen ist, wird mit der eigentlichen Implementierung begonnen. Dieses Vorgehen ist im Nachhinein zwar nicht mehr für andere nachvollziehbar, aber das wäre auch der einzige Kritikpunkt an der testgetrie-

benen Entwicklung. Diese Vorgehensweise ist zugegebenermaßen aufwendig, im Nachhinein werden Sie aber belohnt mit der Tatsache, dass Sie vor Refaktorisierungen oder Änderungen keine Angst haben müssen. Theoretisch können Sie völlig entspannt die Interna der Klassen ändern oder optimieren, brauchen niemals technische Dokumentationen zu



schreiben und sparen sich eine Menge Zeit und Aufregung. Bei guten Tests ist der Code in der Tat die Dokumentation, und bei kompilierten Sprachen ist er zumindest auch immer up to date.

Wenn Sie den ersten Test „grün“ haben, können Sie beginnen, den Code zu optimieren. Zunächst möchten Sie vielleicht Variablennamen anpassen oder verbessern, dann für allgemein bessere Verständlichkeit sorgen, und zuletzt (und das sollte wirklich zuletzt kommen) versuchen, die Laufzeit zu optimieren.

Sie sind jetzt in der Phase der Refaktorisierungen und werden möglicherweise einige „rote“ Tests haben. Nun refaktorisieren Sie so lange, bis wieder alle Tests grün sind (Bild 1). Die testgetriebene Entwicklung ist ein iterativer Prozess, der die drei genannten Zyklen so lange durchläuft, bis es am Code „nichts mehr zu meckern“ gibt.

Was ist denn nun ein guter Unit-Test?

Ein guter Unit-Test konzentriert sich auf eine Sache, eine Funktion oder einen isolierbaren Teil der Anwendung. Wenn Sie einem Single-Responsibility-Principle folgen, bekommen Sie die Möglichkeit, etwas zu testen, quasi geschenkt. Ihr Testsubjekt hat keine (illusorisch) bis ganz wenige Abhängigkeiten (erstrebenswert).

Wegen der wenigen Abhängigkeiten hat die Test-Setup-Methode oder der Arrange-Teil Ihres Tests ebenfalls nur wenige Zeilen. Sie brauchen nur eine bis ganz wenige Assert-Methoden im Test. Ein Test prüft im Idealfall alle möglichen Ausführungspfade.

Zu viele Abhängigkeiten bergen die Gefahr, dass am Ende nur das korrekte Aufsetzen von Mocks oder Stubs getestet wird. Hier ist Ihr analytischer Verstand gefragt. Und seien wir ehrlich: In dem Moment, in dem Sie Ihren neuen Code für die Kollegen verfügbar gemacht haben, fällt Ihnen ganz sicher eine Randbedingung ein, die Sie nicht bedacht haben. Trotz aller Tests empfiehlt es sich nach des Autors Erfahrung unbedingt, vor dem *git push* noch einmal lokal die Anwendung hochzufahren und ein wenig zu spielen. Es gibt wahrscheinlich (in diesem unseren mehr oder weniger kleinen Kosmos) nichts Unangenehmeres, als dass die Anwendung dem ersten Reviewer gleich beim Start auf die Bretter geht.

Datenbank oder nicht Datenbank?

Das ist die Frage, an der sich die Geister scheiden. Hier gibt es durchaus unterschiedliche Meinungen. Und eine Datenbank im Unit-Test zu benutzen verbietet sich eigentlich, wenn man die Definition eines solchen verstanden und verinnerlicht hat. Unbestritten ist sicherlich, dass Sie sich mindestens am Rande eines Integrationstests bewegen, wenn Sie im Unit-Test mit Datenspeichern – welcher Art auch immer – hantieren.

Unbestritten ist aber auch, dass die Daten, auf denen Ihre Geschäftslogik aufbaut, nicht aus dem Nichts kommen. Um eine Datenbank im Test benutzen zu können, sollten Sie Folgendes beachten:

- Die Testdatenbank wird ausschließlich vom Test selbst benutzt.
- Der Test räumt nach erfolgter Arbeit selbst auf.
- Die Testdatenbank muss sich genauso verhalten wie später die Produktiv-Datenbank. Beispiel: Wenn Ihre Produktiv-Datenbank Case-sensitive ist, muss die Testdatenbank das auch sein.
- Die Testdatenbank muss in der Build-Pipeline vom Buildserver erzeugt werden können. Ein Beispiel: Wenn Sie LocalDB von Microsoft als Test-Datenbank benutzen, muss Ihr Build-Server die Tests auf einem Windows-System ausführen. Auf einem Linux-System kann LocalDB nicht benutzt werden.
- Sie sollten darüber nachdenken, Tests mit einer Testdatenbank vom Live-Unit-Testing auszuschließen. Wenn bei Codeänderungen jedes Mal die Tests losrennen, haben Sie unter Umständen keinen Spaß beim Entwickeln, weil es dann doch zu deutlich spürbaren Latenzen kommt.

Es gibt mangels eines *IDbContext*-Interfaces auch einen *InMemoryDbContext*, von dem man aber tunlichst die Finger lassen sollte. Dessen Verhalten entspricht nicht im Geringsten dem einer echten Datenbank.

Die LINQ-Queries, die Sie schreiben, werden beim *InMemoryDbContext* nicht nach SQL übersetzt, Sie (oder Ihr Kollege im Code-Review) bemerken aber erst zur Laufzeit, ►

dass Ihre tolle und unglaublich pfiffige Abfrage in der wirklichen Welt nicht funktioniert. Eine gute Alternative zu LocalDB [11] sind Testcontainer [12].

Und mal so nebenbei bemerkt: Wenn Sie eine Datenbank in Ihren Integrations-Unit-Tests benutzen, bemerken Sie sofort, wenn etwas mit Ihren Migrationen nicht stimmt – ein vergebener Standardwert bei in den Spaltendefinitionen zum Beispiel. Außerdem werden Sie sehr schnell lernen, was mit LINQ to SQL geht und was nicht.

Kurzer Exkurs

Als eingefleischter scharfsichtiger Mensch (C#) haben Sie bemerkt, dass der Fokus im vorangegangenen Absatz auf EF Core lag, ein bisschen Java wurde gestreift (JUnit). Aber „oberflächlich“ waren die bisherigen Absätze zum Unit-Testing nicht. Daher sei hier noch einmal verwiesen auf Jest [13] und Vitest [14], zwei JavaScript-Testframeworks. Wenn Sie noch „oberflächlicher“ werden wollen, sei Ihnen für zuverlässiges End-to-End-Testing (so die Homepage) ein Blick auf Playwright [15] empfohlen.

Auch hinsichtlich Informationen zur Testdatengenerierung werden Sie in dieser Ausgabe der *dotnetpro* fündig werden, beispielhaft seien an dieser Stelle Bogus [16] (deutsch: Schwindel) fürs Backend und Faker.js (deutsch: Fälscher, Hochstapler) [17] fürs Frontend genannt. Mit beiden arbeiten Kollegen des Autors mit Erfolg.

Sprache

Seit Jahren gibt es die Bestrebung, Programmiersprachen an natürliche Sprache anzupassen. Ziel ist dabei, eine Anweisung im Programmtext so formulieren zu können, wie man diese auch formulieren würde, wenn man im analogen Leben als Weisungsbefugter einem Untergebenen Befehle erteilt:

Tu dies! Benutze das! Vermeide jenes!

In der einfachsten Form enthält eine Assertion im Unit-Test eine Bedingung, die wahr sein muss, damit der Test erfolgreich ist.

Anhand des folgenden kleinen Beispiels können Sie die Evolution dieser Bemühungen nachvollziehen, wie sich die Annahme eines Tests vom „Programmier-Sprech“ zu einer eher allgemein verständlichen Aussage entwickelt hat:

```
Assert.That(result == expected);
Assert.That(result, Is.EqualTo(expected));
Assert.That(collection,
    Has.Count.Exactly(expected).Items);
```

Fazit

Niemand möchte ein Spiel spielen, das nicht funktioniert. Bestenfalls bedeutet das schlechte Laune, schlimmstenfalls ein paar verschwendete Münzen. Niemand möchte seine Finanzen einer nicht funktionierenden Software anvertrauen. Bestenfalls bedeutet das einen mehr oder weniger erheblichen finanziellen Verlust, schlimmstenfalls bedroht das Ihre Existenz. Niemand möchte in einem Auto mit nicht funktionierenden Bremsen sitzen. Bestenfalls bedeutet das eine Beule im Blech, schlimmstenfalls endet das fatal.

● Du möchtest gerne mehr erfahren?

Das freut uns, denn ab Erscheinen des vorliegenden Artikels veröffentlichen wir auf developer-world.de/dwx-insights weitere Beiträge, die den Artikel fortsetzen und ausschließlich online zur Verfügung stehen. Derzeit geplant sind folgende Beiträge:

- Lass die KI mal machen
- Architektur

In diesem Sinne: Bis neulich! ■

- [1] Unit-Testing bei Wikipedia, https://en.wikipedia.org/wiki/Unit_testing
- [2] Kent Beck bei Wikipedia, https://en.wikipedia.org/wiki/Kent_Beck
- [3] Erich Gamma bei Wikipedia, https://en.wikipedia.org/wiki/Erich_Gamma
- [4] JUnit, <https://junit.org>
- [5] xUnit, <https://xunit.net>
- [6] Test FX, <https://github.com/microsoft/testfx>
- [7] SWR, Vermeidbare Technikpannen, <https://www.swr.de/wissen/beruehmte-technik-fails-start-der-ariane-5-100.html>
- [8] Programmierfehlerbeispiele bei Wikipedia, https://de.wikipedia.org/wiki/Liste_von_Programmierfehlerbeispielen
- [9] Dependency Injection Design Pattern, www.geeksforgeeks.org/dependency-injection-di-design-pattern/
- [10] Moq auf GitHub, <https://github.com/devlooped/moq>
- [11] Microsoft Learn, SQL Server Express LocalDB, <https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver17>
- [12] JetBrains Blog, How to use Testcontainers with .NET Unit Tests, <https://blog.jetbrains.com/dotnet/2023/10/24/how-to-use-testcontainers-with-dotnet-unit-tests/>
- [13] Jest, <https://jestjs.io>
- [14] Vitest, <https://vitest.dev>
- [15] Playwright, <https://playwright.dev>
- [16] Bogus auf GitHub, <https://github.com/bchavez/Bogus>
- [17] Faker, <https://fakerjs.dev>



Christian Sender

ist bei der sycat IMS GmbH in Hannover als Softwareentwickler im Frontend und Backend unterwegs. Sie erreichen ihn unter c.sender@sycat.com.

LEGACY-CODE TESTBAR GESTALTEN

Testing the Untestable

Legacy-Code hat häufig damit zu kämpfen, dass man ihn nur schwer testen kann. Wir betrachten daher, welcher Tricks und Kniffe man sich bedienen kann, um das zu ändern.

Je nach Alter des Systems gleicht die Arbeit an Legacy-Software einer archäologischen Expedition. Man stößt auf Artefakte vergangener Technologieepochen, deren Erschaffung meist nicht mehr erreichbar sind und deren Dokumentationen zu Staub zerfallen sind beziehungsweise auf eine Weise verfasst wurden, dass man sie nicht mehr nachvollziehen kann. Nichtsdestotrotz haben diese Bauwerke vergangener Zeiten noch so viel Bedeutung und Anziehungskraft, dass man sie nicht einfach begraben kann. Stattdessen müssen sie gepflegt und eventuell für die Zukunft wieder fit gemacht werden. Als Person, die dafür den Auftrag bekommt, sieht man sich dann nicht selten erst einmal mit der Herausforderung konfrontiert, zu verstehen, wie dieser Gegenstand des Interesses überhaupt funktioniert, wie man ihn verändert und, fast noch wichtiger, wie man verhindert, dass man neue Fehler einbaut.

In diesem Zusammenhang sind Tests das Mittel der Wahl. Denn sie können als eine Art Lupe dienen, mit der man sich durch den Code bewegt, um ihn besser erfassbar zu machen. Dabei bildet sich mit der Zeit ein Sicherungsnetz, durch welches Fehler schneller auffallen, während man zeitgleich auch die Architektur verbessern muss, um überhaupt effektiv testen zu können. Damit man nun aber nicht von all dem überfordert wird, sehen wir uns in diesem Artikel anhand eines durchgehenden Beispiels das gängigste Vorgehen genauer an und zeigen damit praktisch, worauf zu achten ist.

Das Beispielszenario

Als Szenario für diesen Artikel dient die Migration einer geschäftskritischen WPF-Desktop-Anwendung in eine moderne Webarchitektur. Der Plan scheint dabei so elegant wie pragmatisch: Die über Jahre gewachsene und bewährte Geschäftslogik, die sauber dank ViewModels und MVVM-Architektur gekapselt ist, soll als Fundament für neue Web-APIs per gRPC verwendet werden. Die erste Überlegung wäre also, bestehende Methoden so zu wandeln, dass sie direkt und nicht über den Umweg von Commands aufgerufen werden. Zusätzlich sollen ihnen Daten übergeben werden, die bisher eher über Properties bereitgestellt wurden.

Doch bei der Realisierung dieses Plans stoßen wir auf eine ernüchternde Wahrheit: Die ViewModels sind nicht so sauber aufgebaut, wie es uns vom langjährigen Entwickler der Software versprochen wurde, als dieser sich in den Ruhestand verabschiedet hat. Bei genauerer Betrachtung entpuppen sie sich als untrennbar mit der Desktop-Welt verbunden. Es finden sich direkte Aufrufe der Datenbank und andere harte Ab-

hängigkeiten, die den Zustand über die gesamte Anwendung verwalten. Dependency-Injection wurde zwar eingesetzt, aber nur verhalten und wenig konsistent. Hinzu kommt, dass die bestehenden Tests eher Makulatur sind. Entweder sie schlagen fehl, sind ganz abgeschaltet oder testen nur offensichtliche Dinge ohne sonderlichen Mehrwert. Na ja, zumindest hat man schon einmal eine veraltete Version von NUnit in der Solution referenziert. Diese Entdeckungen bringen nun unser Modernisierungsprojekt fast zum Stillstand. Die als „untestbar“ und nicht wiederverwendbar entlarvten ViewModels werden von einem wertvollen Asset zu einer massiven Innovationsbremse. Außer natürlich, wir finden Mittel und Wege, unser Vorhaben doch ohne umfangreiche Mehraufwände zu realisieren.

Die Symptome der Untestbarkeit

Die vorangehenden Beschreibungen sind zwar aus dem Leben gegriffen, müssen aber noch etwas abstrahiert werden, um die wiederkehrenden Probleme deutlicher zu machen. Daher klären wir, welche Symptome darauf hindeuten, dass der Code untestbar ist. Dies manifestiert sich in mehreren klar erkennbaren Symptomen, die auf tiefgreifende Verletzungen von Design-Prinzipien hindeuten, wie sie von Autoren wie Robert C. Martin und Martin Fowler beschrieben werden:

- Ein offensichtliches Symptom ist die Vermischung von Verantwortlichkeiten, bei der Geschäftslogik untrennbar mit UI-spezifischem oder datenbankbezogenem Code verbunden ist. Diese Vermischung macht automatisierte Tests unmöglich, da sie eine manuelle Interaktion oder das Vorhandensein einer Datenbank erfordert.
- Ein weiteres kritisches Symptom ist die harte Kopplung an konkrete Infrastruktur-Implementierungen. Die direkte Instanziierung entsprechender Datentypen für den Zugriff auf die Datenbank verhindert den Einsatz von Test-Doubles, wodurch jeder Test zu einem langsamen und unzuverlässigen Integrationstest wird.
- Eng damit verwandt ist die Abhängigkeit von einem globalen Zustand durch Singletons. Diese schaffen versteckte Querabhängigkeiten zwischen Tests, was zu unvorhersehbaren und nicht reproduzierbaren Testergebnissen führt, da der Zustand eines Tests den nächsten beeinflusst. In unserem Fall ergibt sich dies beispielsweise, wenn wir Daten in die Datenbank schreiben. Die Datenbank ist damit unser globaler Singleton und Status. Werden diese Daten vor dem nächsten Test nicht wieder entfernt, kann dies zu unvorhergesehenen Ergebnissen führen. ►

- Zusätzlich zeigt sich Untestbarkeit oft in verschleierte Logik und hoher Komplexität. Dabei werden Kernlogiken in langen, schwer verständlichen Methoden versteckt oder in tief verschachtelten bedingten Anweisungen vergraben. Solche Strukturen sind von außen nicht direkt testbar, und die öffentliche Methode, die sie aufruft, hat so viele mögliche Ausführungspfade, dass eine vollständige Testabdeckung unrealistisch wird.
- Ein fünftes, subtileres Symptom sind versteckte Abhängigkeiten von der Laufzeitumgebung. Dies äußert sich in direkten Aufrufen von statischen Methoden wie *DateTime.Now*, *File.ReadAllText* oder der Erzeugung von Zufallszahlen. Dies wurde im Szenario zwar nicht explizit genannt, ist aber ein oft wiederkehrendes Problem von Systemen wie dem beschriebenen. Solche Aufrufe machen Tests nichtdeterministisch. Demnach hängt ihr Erfolg vom Zeitpunkt der Ausführung oder dem Zustand des Dateisystems ab.

Legacy-Code minimalinvasiv sind. Dieses Vorgehen ermöglicht es, die neue Funktionalität umfassend zu testen, während wir die Gefahr von Regressionen minimieren. Der Sprössling ist auch deshalb ein so einfaches Pattern, weil es uns eigentlich immer möglich ist, neue Klassen anzulegen und dann aus dem alten Code auf sie zuzugreifen. Damit haben wir aber auch einen sehr wichtigen Aspekt schon angesprochen: Der neue Code sollte niemals den alten Code verwenden. Er darf nur aus diesem heraus aufgerufen werden, und bei diesem Aufruf dürfen auch nur Datentypen aus dem neuen Code verwendet werden. Rein organisatorisch sollte man also neuen und alten Code möglichst stark voneinander trennen, da der neue sonst sehr schnell mit den Vorgehensmustern des alten verschmutzt wird und unser Sprössling unter der Last der Altschulden eingeht.

Der zweite wichtige Helfer, um alte Abhängigkeiten nach und nach aufzubrechen, ist das „Strangler Fig“-Pattern, was

● Listing 1: Ein Charakterisierungstest mit Approval Test

```
using ApprovalTests;
using Xunit;

public class LegacyProductViewModelTests
{
    [Fact]
    public void CalculateTotalPrice_CharacterizationTest()
    {
        // Die gegebene Zahl ist die ID aus der Datenbank
        // -> Viele versteckte Abhängigkeiten
```

```
        var viewModel = new LegacyProductViewModel(29387);
        viewModel.Amount = 10;
        viewModel.CalculateTotalPriceCommand.Execute();

        // Achtung gefährlich: Hier können auch Properties
        // verglichen werden, die sich dynamisch ändern!
        Approvals.Verify(viewModel);
    }
}
```

All dies lässt sich auf ein Hauptproblem herunterbrechen: Es gibt zu viele Abhängigkeiten, die sich nicht einfach auflösen lassen und dafür sorgen, dass der zu testende Code nicht isoliert betrachtet werden kann. Welche Muster helfen uns nun aber dabei, diese Abhängigkeiten aufzubrechen, ohne dass wir dabei die Applikationslogik zerbrechen?

Von Sprösslingen und Schlingpflanzen

Der Sprössling, im Englischen bezeichnet als *Sprout*, ist ein Pattern, das von Michael Feathers [1] entwickelt wurde. Es zählt zu den Refactoring-Mustern, wobei es als eines der einfacheren angesehen wird. Die Idee dahinter ist denkbar simpel: Es wird ein neuer, testbarer „Sprout“-Teil des Codes erstellt, der die neue Logik enthält. Ausgehend von diesem korrekt getesteten und entworfenen Code kann sich dann weiterer, sauberer Code entwickeln, der alle notwendigen Qualitätsanforderungen erfüllt.

Im Beispielszenario würde das insbesondere dann helfen, wenn man das bestehende System weiterentwickeln möchte, ohne das Vorhandene in Gefahr zu bringen. Dabei wird dieser Sprout vom jeweils ursprünglichen, untestbaren ViewModel aufgerufen, wodurch die eigentlichen Änderungen am

so viel wie „Würgefeige“ heißt. Bei diesem von Martin Fowler popularisierten Muster wird das alte System schrittweise durch ein neues System ersetzt, anstatt es in einem Big-Bang-Ansatz komplett auszutauschen.

Das neue System „schlingt“ sich dabei wie eine Würgefeige um das bestehende System und übernimmt nach und nach dessen Funktionen. Dabei wird der Datenfluss zwischen dem bestehenden und dem neuen System so umgeleitet, dass Anfragen, die sich auf bereits migrierte Funktionalitäten beziehen, vom neuen System verarbeitet werden, während Anfragen für noch nicht migrierte Teile weiterhin an das alte System geleitet werden.

Damit haben wir unsere eigentliche Vorgehensstrategie ausgearbeitet. Für deren Umsetzung erläutern wir nachfolgend noch weitere Patterns. Zunächst wollen wir aber auf die Grundregeln für die weiteren Umbaumaßnahmen eingehen:

- Neuer Code muss allen Qualitätsanforderungen genügen. Dies schließt eine entsprechende Testabdeckung mit ein.
- Neuer Code muss unabhängig vom alten Code arbeiten können.
- Neuer Code darf alten Code nicht aufrufen, aber alter Code darf neuen Code aufrufen.

● Listing 2: Die Nahtstelle für den Datenbankzugriff

```
public class ProductRepository : IProductRepository
{
    public Product Get(int id)
    {
        // Hier erscheint der Code, der vorher im
        // Constructor stand.
    }
}
```

- Wann immer neuer Code alten Code doch aufrufen muss, ist eine Schnittstelle bereitzustellen. Diese gilt als neuer Code und ist auch im Code-Bereich für neuen Code unter den gleichen Bedingungen abzulegen und zu pflegen.
- Der alte Code darf sich Adaptorn bedienen. Diese müssen aber im Bereich des alten Codes abgelegt werden.

So schaffen wir es, dass das alte System schrittweise in neue Muster übertragen wird, wo dies Sinn ergibt. Alles, was nicht angepasst werden muss, verbleibt gut abgeschottet im alten Codebereich und verschmutzt den neuen nicht.

Die Charakterisierungstests

Wie bereits beschrieben, helfen Sprouts nur bei neuem Code, und ihre Einbindung in den Bestandscode birgt noch immer die Gefahr, jenen zu stören. Es ist daher unerlässlich, sich vor jeglicher Änderung gegen mögliche Störungen abzusichern. Da Legacy-Logik aber meist sehr komplex ist, ist es sehr schwierig, jene entkoppelt und zielgerichtet aufzurufen. In diesen Fällen helfen Charakterisierungstests. Diese werden auch als Golden-Master-Tests bezeichnet und erfassen das

aktuelle Verhalten von Legacy-Code, anstatt zu testen, wie er funktionieren sollte.

Die Grundidee ist dabei, dass man den Bestandscode einmal aufruft und dessen Ergebnis speichert. Die Ergebnisse aller weiteren Aufrufe werden dann mit dem namensgebenden Golden-Sample verglichen. Man charakterisiert damit den jeweiligen Code, auch wenn man ihn wie eine Black-Box behandeln muss. Dieses Vorgehen ist dank Frameworks wie Approval Test [2] vergleichsweise einfach umsetzbar.

Im Beispiel aus Listing 1 wird davon ausgegangen, dass der Gesamtpreis für ein Produkt in einem ViewModel berechnet wird. Die eigentliche Logik ist uns unbekannt, wir wissen nur, dass auf irgendeine Weise Rabatte, Mehrwertsteuer und Ähnliches mit eingerechnet werden. Das ViewModel selbst gibt uns aber wenig Hinweise darauf, es bekommt die ID des Produkts aus der Datenbank und sammelt damit alle notwendigen Stammdaten. Ansonsten haben wir nur Schnittstellen zur Verfügung, die sonst vom UI verwendet werden.

Wir können also die Menge der Produkte auswählen und ein Command triggern, das dann die Berechnung vornimmt. Approval Test hilft uns hierbei, indem es nach der ersten Ausführung des Tests die Werte aller Schnittstellen des ViewModels in eine Datei mit „approved“ im Namen schreibt. Diese Datei muss in die Quellcodeverwaltung mit eingchecked werden. Wird der Test dann noch einmal ausgeführt, wird eine temporäre Textdatei geschrieben, die „received“ im Namen trägt und mit der „approved“-Datei verglichen wird. Auf diese Weise lassen sich dann auch komplexe Daten vergleichen.

Man sollte aber nicht dem Trugschluss unterliegen, es würde auf diese Weise wirklich die funktionale Korrektheit geprüft. Mit dieser Art Test baut man sich im Grunde ein Frühwarnsystem auf, das immer dann anschlägt, wenn sich die Ergebnisse einer Aktion ändern. Demnach schlagen die Tests gegebenenfalls auch fehl, wenn eine solche Änderung gewünscht ist. Man hat also mit einer erheblichen Unschärfe ►

● Listing 3: Der Unit-Test

```
using Xunit;
using Moq;

public class ProductViewModelTests
{
    [Fact]
    public void
        TotalPriceOfTwoProductsWithCostsOf_1_is_2()
    {
        // erster Parameter = id,
        // zweiter Parameter = Einzelpreis
        var product = new Product(29387, 1);

        // egal welches Produkt gefragt ist, gib immer
        // unseres zurück
        var repo = new Mock<IProductRepository>();
```

```
repo.Setup(x =>
    x.Get(It.IsAny()).Returns(product);

// Id wird übergeben, ist aber eigentlich egal.
var viewModel = new LegacyProductViewModel(29387);
viewModel.ProductRepository = repo.Object;

viewModel.Amount = 10;
viewModel.CalculateTotalPriceCommand.Execute();

// Diesmal Zugriff auf konkrete Property und ohne
// Approval Tests -> Testergebnisse sind eindeutig
Assert.AreEqual(2, viewModel.TotalPrice);
}
```

● Von Test-Doubles, Stubs und Mocks

Um die Abhängigkeiten in unserem Legacy-Code aufzubrechen und isolierte Tests zu ermöglichen, haben wir bereits Test-Doubles kennengelernt. Es ist jedoch wichtig, die verschiedenen Arten von Test-Doubles zu verstehen, um sie korrekt anzuwenden. Die Begriffe Mock, Stub und Test-Double werden oft synonym verwendet, bezeichnen aber unterschiedliche Konzepte.

Test-Double

Der Oberbegriff „Test-Double“ bezeichnet laut Gerard Meszaros jedes Objekt, das im Rahmen eines Tests ein echtes Objekt ersetzt. Der Zweck eines Test-Doubles ist es, die Ausführung eines Tests zu ermöglichen, ohne auf die tatsächliche Implementierung der Abhängigkeiten angewiesen zu sein. Dies ist besonders nützlich, wenn reale Abhängigkeiten (etwa Datenbanken oder externe Dienste) langsam, unzuverlässig oder komplex einzurichten sind.

Stub

Ein Stub ist ein Test-Double, das vordefinierte Antworten auf bestimmte Methodenaufrufe liefert. Sein Hauptzweck ist es, dem getesteten Code die für seine Ausführung notwendigen Daten bereitzustellen. Ein Stub hat keine Kenntnis über die Interaktionen, die mit ihm stattfinden, und sein Verhalten ist statisch. Er

wird verwendet, um den Zustand des Systems unter Test zu beeinflussen, indem er bestimmte Werte zurückgibt.

Mock

Ein Mock verhält sich fast wie ein Stub, hat aber ein Bewusstsein. Er wird so konfiguriert, dass er die Verwendung validiert und den Test fehlschlagen lässt, falls übergebene Daten oder vorgenommene Methodenaufrufe von seiner Konfiguration abweichen. (siehe die unten stehende Tabelle [Stub versus Mock](#))

Neben diesen Begriffen gibt es noch weitere, die aber nicht im Artikel verwendet werden. Beispielsweise dienen „Fakes“ als leichtgewichtige Implementierung einer bestimmten Logik, die aber nur dem Test dient. „Dummies“ haben keinerlei Funktion, sondern sind nur Daten, da das Vorhandensein jener Daten für den Test gebraucht wird. Um welche Daten es sich dabei konkret handelt, ist aber meist nicht von Bedeutung. Als „Spy“ funktionierenden Mock-Objekte, wenn sie nicht nur validieren, sondern die Aufrufe auch aufzeichnen und protokollieren. „Shims“ hingegen sind Objekte, die per Manipulation der Laufzeitumgebung an die Position der realen Implementierung gesetzt werden, um beispielsweise ein API zu ersetzen, ohne den Code zu verändern.

● Stub versus Mock

Merkmal	Stub	Mock
Zweck	Stellt Daten/Werte für den getesteten Code bereit.	Überprüft Interaktionen/Verhalten des getesteten Codes mit dem Double.
Interaktion	Reagiert passiv auf Aufrufe mit vordefinierten Werten.	Hat Erwartungen an Aufrufe und validiert diese.
Fokus	Zustandsbasierte Tests (Was kommt heraus?)	Verhaltensbasierte Tests (Was passiert?)
Fehlschlagen	Schlägt fehl, wenn die Rückgabewerte nicht stimmen oder eine Methode nicht aufgerufen wird, die notwendig ist.	Schlägt fehl, wenn die erwarteten Aufrufe nicht stattfinden oder unerwartete Aufrufe gemacht werden.
Beispiel	<code>repo.Setup(x => x.Get(It.IsAny()))Returns(product);</code>	<code>loggerMock.Verify(x => x.Log(It.IsAny<string>()), Times.Once());</code>

zu kämpfen und sollte sich genau mit den erzielten Ergebnissen auseinandersetzen. Darüber hinaus helfen Tests dieser Art nur bei komplexem Code, nicht aber bei externen Abhängigkeiten. Jene muss man immer noch auf eine andere Art und Weise auflösen.

Die Nahtstelle

Listing 1 zeigt uns nicht nur, wie ein Approval Test funktioniert, sondern auch die Schwierigkeiten beim Testen, die wir zuvor schon erläutert hatten. Wobei das nicht ganz korrekt ist, denn was wir tatsächlich sehen, ist, dass wir nichts sehen. Wir können es nur erahnen, und zwar, weil wir eine eigenartige Nummer an den Konstruktor des ViewModels weiterreichen, die sich bei näherer Betrachtung als ein Eintrag in der Datenbank herausstellt. Jene ID muss also auf magische Art und Weise in ein konkretes Objekt mit entsprechenden Stammdaten verwandelt werden. Dafür ist mit Sicherheit ein Datenbank-

zugriff notwendig. Diesen sehen wir aber nicht, da er in den Tiefen des ViewModels und wahrscheinlich sogar in dessen Konstruktor aufgelöst wird. Um den Test nun auch erfolgreich ausführen zu können, müssen wir jene Datenbank bereitstellen. Das ist langsam und fehleranfällig. Daher ist unser nächster Auftrag, diese Abhängigkeit aufzubrechen. Michael Feathers hat hierzu das Muster der Nahtstelle oder im Englischen „Seam“ beschrieben. Dies ist ein Punkt im Programm, an dem das Verhalten geändert werden kann, ohne den Code an dieser Stelle direkt zu bearbeiten. In objektorientierten Sprachen wie C# ist die häufigste und effektivste Form einer Nahtstelle ein entsprechendes Interface.

Listing 2 zeigt dabei sehr einfach, wie eine Nahtstelle erzeugt und welche Muster verwendet werden können. Zugeben: Der Code ist extrem simpel, aber eben auch effektiv. Zunächst wird der Code, der den Datenbankzugriff im View-Model herstellt, in eine eigene Methode *Get* verschoben. An-

schließlich führen wir den Charakterisierungstest aus. Es dürfte keine Fehler geben, da wir ja nur Code verschoben haben. Dann wird diese Methode in eine eigene *ProductRepository*-Klasse verschoben. Führen wir den Test wieder aus, sollte es immer noch keinen Fehler geben, da wir nach wie vor nur Code verschieben. Anschließend wird eine Schnittstelle erzeugt, indem wir ein Interface aus der Klasse *ProductRepository* extrahieren. Diese müssen wir dem ViewModel noch mitgeben. Im Normalfall erzeugt sie die dafür notwendige Instanz selbst. Für den Test müssen wir sie aber von außen setzen können.

Um das zu realisieren, erhält das ViewModel eine Internal Property vom Typ *IProductRepository*. Internal deshalb, weil wir dann über *InternalsVisibleTo* [3] im Projekt unseres Produktivcodes die Property für unser Testprojekt sichtbar machen können. Damit wird es uns möglich, die Property von außen mit einem Test-Double zu ersetzen. Dieses Test-Double greift dann während des Tests nicht auf die Datenbank zu, sondern liefert einfach das Verhalten, das wir für den konkreten Test brauchen, siehe dazu auch den Kasten **Von Test-Doubles, Stubs und Mocks**. Listing 3 zeigt einen Unit-Test, der weitere Hinweise darauf enthält, wie wir nun die Abhängigkeit zur Datenbank aufgebrochen haben. Dabei kommt das Framework Moq zum Einsatz, um ein Test-Double zu erstellen und zu konfigurieren. Anzumerken sei dazu noch, dass der ursprüngliche Charakterisierungstest ruhig bestehen bleiben kann. Er dient uns als Integrationstest und ist vor allem dann hilfreich, wenn noch weitere Veränderungen am Testgegenstand vorgenommen werden müssen.

Bild 1 zeigt den bis hierher verfolgten Lösungsansatz, ausgenommen den Sprössling. Mit diesem Vorgehen kann bereits sehr viel erreicht werden, während gleichzeitig die Risiken niedrig gehalten werden. Die eigentliche Magie entsteht dabei weniger durch die Einführung einer Schnittstelle und die Verwendung von Test-Doubles. Sie entsteht durch die

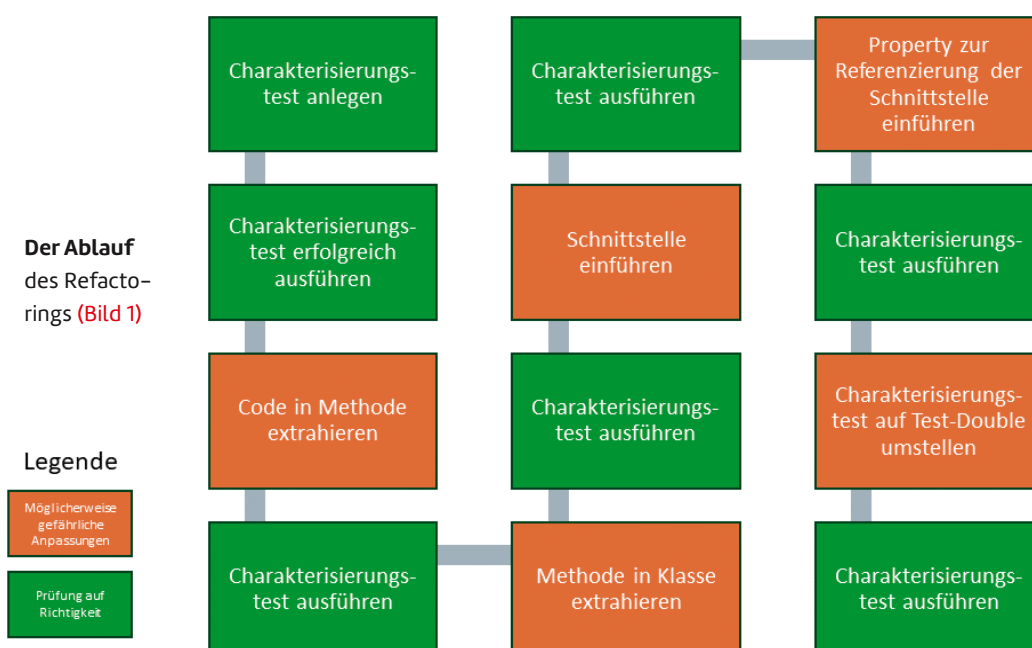
Verwendung von Charakterisierungstests und das damit entstehende Sicherungsnetz. Solange wir die Charakterisierungstests nicht verändern, können wir uns sicher sein, dass sie auf mögliche Probleme hinweisen, und in dem Fall, wo wir sie doch anpassen müssen, sollten wir sie vor der Anpassung ausführen, um sicherzustellen, dass die Implementierung korrekt ist. Anschließend sollte die Implementierung nicht angepasst werden, solange wir die Tests verändern. Dadurch wird die Implementierung zum Test des Tests.

Das bescheidene Objekt

Betrachtet man unser Beispiel genauer, ist das eigentliche Kernproblem, dass das ViewModel insgesamt zu viel Logik besitzt. Würden Sie die Aufrufe an das UI an eine eigene Logikschicht weiterleiten, könnten wir sie komplett ignorieren und unsere Service-Endpoints einfach auf die gleiche Logik zugreifen lassen.

Objekte, die im Grunde selbst nichts tun beziehungsweise alle Aufrufe nur an andere weiterleiten, werden auch als Humble Objects [4] oder eben „bescheidene Objekte“ bezeichnet, zumindest dann, wenn man sie gezielt einsetzt, um einen Grenzbereich zwischen den Schichten einer Applikation zu schaffen. Entstehen sie versehentlich oder ist ihr Zweck nicht ersichtlich, werden sie eher als „faule Objekte“ oder im Englischen „Lazy Objects“ bezeichnet. In diesem Fall handelt es sich dann um ein Antipattern, also ein Muster, dass man laut Fowler eher vermeiden sollte [5]. Daran sieht man ein gewisses Kernproblem: Ob es sich um ein erwünschtes oder unerwünschtes Muster handelt, liegt bis zu einem gewissen Grad im Auge des Betrachters. Während die einen in Humble Objects eine erwünschte Trennlinie zum Beispiel zwischen dem UI und Geschäftslogik sehen, werden sie von anderen Entwicklern als unnötige „Durchlauferhitzer“ wahrgenommen, die nur die Code-Menge aufblähen und die Gesamtlogik schwerer verständlich machen.

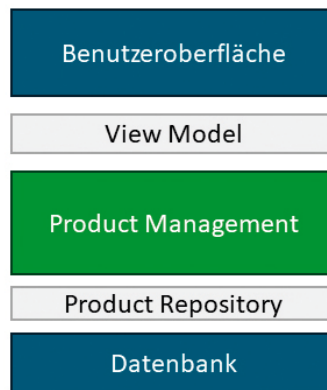
Im Fall von ViewModels für WPF-Applikationen findet man insbesondere die letztgenannte Sichtweise noch immer weit verbreitet vor, und auch in den API-Adaptoren anderer Technologien, wie zum Beispiel Controller in ASP.NET, kann man oft genug zu viel Logik finden. Warum jetzt aber so viel Zeit auf diese Erklärung verwenden, wenn es in diesem Artikel doch eigentlich um Tests geht? Zum einen ist es wichtig, zu verstehen, woher die Probleme kommen, die uns das Testen erschweren, ►



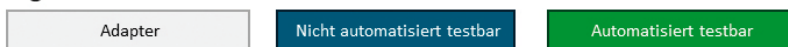
Vor dem Refactoring



Nach dem Refactoring



Legende



Von untestbarem zu gut strukturiertem Code (Bild 2)

und zum anderen sollte man es im Rahmen von Refactorings auch besser machen. Wenn etwas schwer zu testen ist, liegt dies nicht an den Tests, sondern eher am Testgegenstand. Patterns, die uns beim Testen helfen, können wiederum dazu beitragen, auch die Gesamtapplikation besser zu strukturieren, selbst wenn es im ersten Moment so aussehen mag, als griffe man damit vor allem den Tests unter die Arme.

Für unseren Legacy-Code bedeutet dies, dass wir den Ansatz des vorangegangenen Abschnitts ruhig weitertreiben können, bis das ViewModel selbst eben nur noch ein Humble Object ist, das jegliche Funktionalität an die darunterliegenden Schichten weiterleitet. Dafür haben wir das *Product-Management* geschaffen, eine Klasse, die alle notwendige Verwaltungslogik und sowie die Berechnungsverfahren enthält, während sie ihre Daten von einem *ProductRepository* bezieht. Jene Klassen können wir anschließend sehr gut mit Tests abdecken. Ob man es so weit treibt, ist natürlich von vielen Faktoren abhängig. Wie aber bereits geschrieben skizziert uns dies einen Zielzustand, der auch für Webapplikationen zum Einsatz kommen kann. Dort würde das ViewModel nur durch einen entsprechenden Controller ersetzt, der im Grunde der gleichen Aufgabe folgt, die Daten aus der Geschäftslogik für die darüberliegende Schicht aufzubereiten. Statt von Aufbereiten kann man auch von Adaptieren sprechen. Bei diesem Vorgang werden beispielsweise auch Fehler korrekt für die Darstellung beziehungsweise Weiterleitung aufbereitet oder Methoden an die entsprechenden externen Endpunkte angebunden. Im Fall von WPF haben wir es dann mit den berühmten Commands und Properties zu tun. Bei gRPC wären es die generierten C#-Klassen, die sich aus der Schnittstellenbe-

schreibung des jeweiligen Dienstes ergeben. Damit sind dann ViewModel und Controller auch gar nicht mehr so bescheiden. Ihre eigentliche Logik besteht nicht darin, Geschäftsprozesse umzusetzen. Stattdessen ist ihre Aufgabe, eben die Geschäftslogik technologisch zu adaptieren, und damit kann auf Dauer aus dem bescheidenen Objekt recht schnell ein vollwertiger Adapter werden (Bild 2).

Die Kapsel

Ein Adapter, wie im vorhergehenden Text beschrieben, wandelt die Schnittstelle einer Klasse in eine andere Schnittstelle um, die von Clients erwartet wird. Er ermöglicht es, dass Klassen mit inkompatiblen Schnittstellen zusammenarbeiten können, die dies sonst nicht könnten. Im Kontext unseres Beispiels bedeutet dies, dass das ViewModel (oder der Controller) die vom UI oder einem externen Dienst benötigten Daten und Aktionen so aufbereitet,

dass die darunterliegende Geschäftslogik diese verarbeiten kann, und umgekehrt. Es geht also darum, eine Brücke zwischen zwei inkompatiblen Schnittstellen zu schlagen.

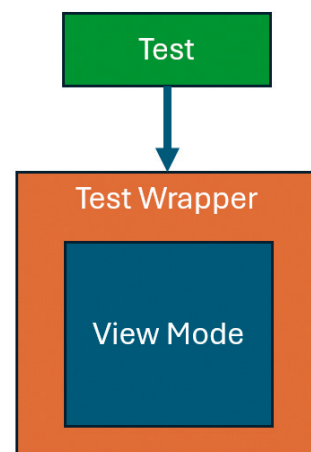
Das Wrapper- beziehungsweise Kapsel-Pattern ist ein Überbegriff, der oft synonym mit dem Adapter-Pattern verwendet wird, aber auch andere Muster wie das Decorator-Pattern oder das Proxy-Pattern umfassen kann. Im Kern geht es beim Wrapper-Pattern darum, ein bestehendes Objekt in ein neues Objekt einzubetten, das zusätzliche Funktionalität hinzufügt oder die Schnittstelle anpasst, ohne die ursprüngliche Implementierung zu ändern. Der Adapter ist eine spezifische Art von Wrapper, der sich auf die Anpassung der Schnittstelle konzentriert.

Wrapper können wir aber auch um den zu testenden Code schließen (Bild 3). Muss für unseren Charakterisierungstest beispielsweise eine Datenbank aufgesetzt und mit Testdaten hinterlegt werden, könnte man zuvor die notwendige Logik ausführen, um die Datenbank in-memory aufzusetzen, indem man statt des ViewModels eine Testkapsel aufruft. Diese rei-

chert das ViewModel um den notwendigen Infrastrukturcode an und hält damit unsere eigentlichen Tests und den Produktivcode frei von diesen Belangen. Das Vorgehen ist aber mit Vorsicht zu verwenden, da die Kapsel sehr starke Abhängigkeiten zur Logik erhält, die sie kapselt. Da ihr Hauptanliegen außerdem ist, das Testen zu vereinfachen, darf sie nicht im Produktivcode liegen und muss im Testcode abgelegt werden.

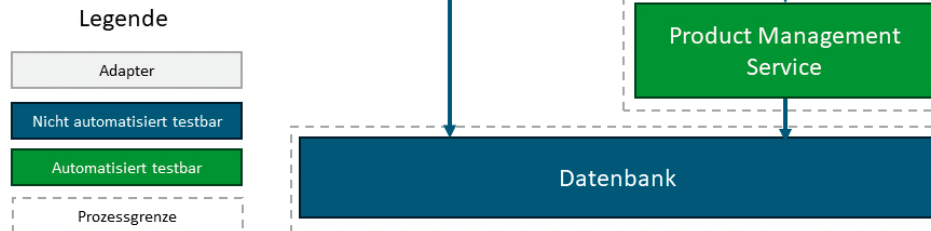
Die Weiche

Mit der Weiche beziehungsweise dem Gateway [6] fassen wir jetzt noch alles zuvor Gelernte zusammen und zeigen, wie die alte WPF-Oberfläche weiterverwen-



Testen über eine Kapsel (Bild 3)

Die Weiche zur Vermittlung zwischen alter und neuer Welt (Bild 4)



det werden kann, auch wenn zeitgleich ein neues Web-UI aufgebaut wird. Dies ist möglich, indem man die Logik als Backend bereitstellt und die Schnittstelle für unser ViewModel mit einer Umschaltlogik versieht. Dies könnte zum Beispiel bei der Konfiguration des IoC- beziehungsweise DI-Containers geschehen: Im Fall der alten Logik wird das alte ViewModel geladen. Im Fall neuer Logik wird ein anderes ViewModel geladen, welches die Anfrage an den zentralen Dienst stellt. Siehe dazu auch **Bild 4**.

Die Weiche ist somit ein entscheidendes Muster, um die schrittweise Migration von Legacy-Systemen zu unterstützen, insbesondere in Verbindung mit dem Sprout- und dem Strangler-Fig-Pattern. Sie fungiert als Schaltzentrale, die entscheidet, ob eine Anfrage vom alten oder vom neuen System bearbeitet werden soll. Diese Konstellation ermöglicht einen kontrollierten Übergang und minimiert das Risiko während der Modernisierung.

Nimmt man nun alles zusammen, ergeben sich für uns folgende mögliche Code-Transformationen, zwischen denen jeweils immer alle Tests erfolgreich ausgeführt werden sollten:

- Wir schaffen zunächst die notwendige Testabdeckung über Charakterisierungstests.
- Wir extrahieren allen Logik-Code und die externen Abhängigkeiten in eigene Klassen.
- Damit ergeben sich automatisch entsprechende Adapter.
- Diese Adapter werden über Interfaces zu Nahtstellen.
- Der ursprüngliche Legacy Code (unser ViewModel) wandelt sich in ein bescheidenes Objekt (oder Adapter, wenn wir Technologieanpassungen machen müssen).

Vom Legacy Code bleibt damit wenig mehr als eine Hülle. Aus ihr können wir eine Weiche ableiten, um auch zur Laufzeit zwischen alter und neuer Welt umschalten zu können.

Wie bereits beschrieben, müssen wir diesen Pfad nicht komplett verfolgen. Insbesondere der letzte Schritt wird eher in Ausnahmesituationen benötigt.

Fazit

Die Herausforderung, Legacy-Code zu testen und zu modernisieren, scheint auf den ersten Blick entmutigend. Doch wie dieser Artikel gezeigt hat, gibt es bewährte Methoden und Muster, um auch die „untestbarsten“ Systeme schrittweise zu bändigen. Von der Charakterisierung bestehenden Verhaltens mit Golden-Master-Tests bis hin zum Aufbrechen harter Abhängigkeiten durch Nahtstellen und Test-Doubles zeigt sich: Der Weg zu testbarem und wartbarem Code ist kein „Big Bang“, sondern eine Reihe kleiner, kontrollierter Transformationsschritte.

Das Strangler-Fig-, das Sprout- und das Gateway-Pattern bieten

dabei elegante Strategien, um neue, sauber implementierte Funktionalitäten um den Altcode herum aufzubauen, während das Konzept der Humble Objects und des Adapter-Patterns uns lehrt, die Architektur so zu gestalten, dass Geschäftslogik von technischen Details entkoppelt wird. Indem wir diese Prinzipien anwenden, verwandeln wir „untestbaren“ Code nicht nur in testbaren, sondern legen auch das Fundament für eine zukunftsfähige und flexible Softwarearchitektur. Damit wird die Investition in Testbarkeit auch eine Investition in die Langlebigkeit und Anpassungsfähigkeit unserer Legacy-Systeme. ■

- [1] Michael C. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004, ISBN 978-0-13-117705-5
- [2] ApprovalTest.Net, <https://github.com/approvals/ApprovalTests.Net>
- [3] InternalsVisibleTo, <https://learn.microsoft.com/de-de/dotnet/api/system.runtime.compilerservices.internalsvisibletoattribute?view=net-9.0>
- [4] Humble Objects, <http://xunitpatterns.com/Humble%20Object.html>
- [5] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999, ISBN 978-0-201-48567-7
- [6] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002, ISBN 978-0-321-12742-6



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

AI-DRIVEN UNIT-TESTS

Tests boostern mit GitHub Copilot und xUnit

Mit KI-Unterstützung werden aus mühsamen Softwaretest-Routinen intelligente Unit-Tests, die Abdeckung erhöhen, Qualität sichern und Entwicklung spürbar beschleunigen.

Es ist Montagmorgen, der Kaffee steht neben der Tastatur, und eigentlich wollte man „nur schnell“ einen kleinen Bugfix einbauen. Doch der Code, den man anfassen muss, ist ein Dschungel aus Abhängigkeiten: Änderungen an einer Klasse ziehen unerwartet Fehler an ganz anderer Stelle nach sich. Softwaretests? Fehlanzeige. Keine Unit-Tests, keine Integrationstests: Keine Sicherheit, ob der Fix mehr kaputt-macht, als er repariert. Also läuft alles per Debugger, Copy and Paste und nach dem persönlichen Bauchgefühl.

Die Realität vieler .NET-Projekte

Unit-Tests gehören längst zum guten Ton moderner Softwareentwicklung – theoretisch. In der Praxis sieht es aber oft anders aus. Entwickler wissen, dass Tests Qualität sichern, Regressionen verhindern und langfristig Zeit sparen. Trotzdem wird das Testing häufig vernachlässigt. Die Gründe dafür sind vielfältig und allzu menschlich:

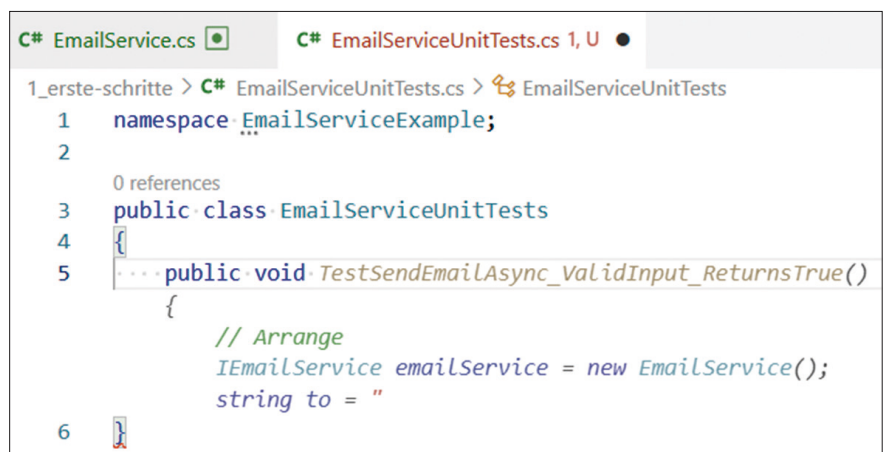
- **Untestbare Architekturen:** Anwendungen, die eng mit Datenbanken, APIs oder Dateisystemen verwoben sind, lassen sich kaum isoliert beziehungsweise einzeln prüfen und validieren. Eine klare Trennung zwischen Logik und Infrastruktur fehlt oft – und damit auch die Testbarkeit.
- **Spaghetti-Code:** Über Jahre gewachsene Klassen mit vielen Abhängigkeiten, fehlender Struktur und Copy-and-Paste-Logik sind kaum testbar; jeder Versuch, eine einzelne Methode zu testen, gleicht dem Umgang mit einem fragilen Kartenhaus.
- **Fehlende Testdaten:** Ohne realistische Datensätze oder klare und reale Eingangswerte sind Tests sinnlos. Oft scheitert es schon daran, dass Testdaten nicht existieren, die Testdatenbank nicht befüllt wurde oder das Mocking zu aufwendig ist.
- **Ineffizientes Testen:** Tests werden manuell ausgeführt, statt automatisiert in CI/CD-Pipelines integriert zu sein. Fehler schleichen sich wieder ein, weil niemand merkt, dass ein alter Test längst nicht mehr läuft.

Und dann kommen noch die Klassiker: Zeitdruck, „temporärer Code“, fehlendes Wissen, ein drängelnder Chef, mangelnde Testkultur und Legacy-Code, der nie testbar gebaut wurde. Die Folge: Viele Teams testen „auf gut Glück“ statt systematisch.

Doch genau hier kann künstliche Intelligenz helfen. Tools wie GitHub Copilot versprechen, Routinearbeit zu übernehmen, Testvorschläge zu liefern, realistische Daten zu generieren und Entwicklern dabei zu helfen, schneller zu besseren Tests zu kommen; ohne dass man die Verantwortung für Qualität an die Maschine abgibt.

Erste Schritte mit GitHub Copilot

Der Einstieg in KI-unterstütztes Testen ist unspektakulär, aber wirkungsvoll: GitHub Copilot [1] lässt sich in Visual Studio oder Visual Studio Code – aber auch in anderen .NET-Ent-



```

C# EmailService.cs  C# EmailServiceUnitTests.cs 1, U
1_erste-schritte > C# EmailServiceUnitTests.cs > EmailServiceUnitTests
1 namespace EmailServiceExample;
2
0 references
3 public class EmailServiceUnitTests
4 {
5     public void TestSendEmailAsync_ValidInput_ReturnsTrue()
6     {
// Arrange
IEmailService emailService = new EmailService();
string to = "

```

Eingabeervollständigung durch GitHub Copilot (Bild 1)

wicklungsumgebungen wie JetBrains Rider – direkt in den Entwicklungsworkflow integrieren und liefert Vorschläge in Echtzeit, sobald C#-Code geschrieben wird. Mit der richtigen Projektstruktur und klaren Methodensignaturen entstehen daraus erstaunlich präzise Testentwürfe.

Die Eingabeervollständigung, die IntelliCode [2] sehr ähnlich sieht (vergleiche Bild 1), stammt von GitHub Copilot

und ist funktional so aufgebaut, dass zunächst nur wenige Zeilen Code vorgeschlagen werden, um dem Entwickler eine übersichtliche Voransicht zu bieten.

Ist der Vorschlag aus Sicht des Entwicklers nicht korrekt oder nicht passend, kann einfach weiter Code entwickelt und geschrieben werden, sodass GitHub Copilot mehr Kontext erhält und einen weiteren, besseren oder präziseren Vorschlag erzeugen kann. Passt der Vorschlag, kann mithilfe der [Tab]-Taste der Vorschlag in der IDE akzeptiert werden.

Mit jeder weiteren Zustimmung durch die [Tab]-Taste geführt, kann so binnen Sekunden ein vollständiger, einfacher, jedoch valider Test aufgebaut und durch den Entwickler verifiziert werden (Bild 2).

Test-Kontext

GitHub Copilot berücksichtigt bei der Codegenerierung stets den Kontext der aktuellen Datei sowie den Projektkontext, in dem sich diese Datei befindet. Dazu gehören alle sichtbaren

works, einschließlich deren typischen Mustern und Konventionen. Diese kontextbewusste Einbettung führt zu qualitativ hochwertigeren Vorschlägen, die sich nahtlos in bestehende Testprojekte integrieren lassen, und damit zu einem echten Boost für selbst geschriebene Unit-Tests. Sie eignen sich daher trotz ihrer einfachen Anwendung besonders für Tests, bei denen ein hohes Bewusstsein des Entwicklers (weiterhin) gewünscht oder (noch) erforderlich ist. Die Produktivitätssteigerung ist zwar vorhanden, wird jedoch maßgeblich durch die Arbeitsgeschwindigkeit und Erfahrung des Entwicklers reglementiert.

Produktivitätssteigerung durch Prompting

Ein echter Produktivitätsboost ist greifbar, wenn man bereit ist, mehr Kontrolle an GitHub Copilot und Co. abzugeben. Sobald Copilot nicht mehr nur als Autovervollständigung, sondern als aktiver Mitentwickler verstanden wird, entfaltet das Werkzeug sein volles Potenzial.

```

C# EmailService.cs | C# EmailServiceUnitTests.cs 1, U
1_erste-schritte > C# EmailServiceUnitTests.cs > EmailServiceUnitTests > TestSendEmailAsync_ValidInput_ReturnsTrue
1 namespace EmailServiceExample;
2
3 0 references
4 public class EmailServiceUnitTests
5 {
6     0 references
7     public async Task TestSendEmailAsync_ValidInput_ReturnsTrue()
8     {
9         // Arrange
10        IEmailService emailService = new EmailService();
11        string to = "test@example.com";
12
13        // Act
14        bool result = await emailService.SendEmailAsync(to, "Test Subject", "Test Body");
15        // Assert
16        if (!result) throw new Exception("Test failed: Expected true but got false.");
17        Console.WriteLine("TestSendEmailAsync_ValidInput_ReturnsTrue passed.");
18    }
19 }

```

Aufbau der Testmethode
durch GitHub Copilot (Bild 2)

Namespaces, vorhandene *using*-Direktiven (per Datei- oder globaler Deklaration [3]) und die umgebende Code-Struktur.

Sofern kein Testframework importiert ist, fehlen Copilot die notwendigen Anhaltspunkte, um spezifische Teststrukturen zu verwenden. In diesem Fall greift das Modell auf generische C#-Konstrukte zurück und erzeugt beispielsweise eigene *try/catch*-Blöcke oder benutzerdefinierte Exceptions, um Fehlerfälle zu werfen (Bild 3).

Sobald jedoch ein Testframework wie xUnit [4] per *using Xunit* eingebunden ist und damit bekannt gemacht wurde, ändert sich das Verhalten: Copilot erkennt die verfügbaren Assertions und integriert diese korrekt in die generierten Tests. Statt eigene Ausnahmen zu deklarieren, verwendet das Modell nun idiomatische Aufrufe wie *Assert.True()*, *Assert.Throws<ArgumentException>()* oder *Assert.Equal()*.

Moderne Sprachmodelle wie GPT-5 oder Claude Sonnet 4.5 verfügen über ein tiefes Verständnis gängiger Testframe-

Wer dem Modell größere Codeabschnitte überlässt – beginnend bei Testmethoden bis hin zum Überlassen ganzer Testklassen oder später der vollständigen Kontrolle der Projekte oder gar Solutions –, profitiert von einer erheblichen Zeitersparnis. Copilot analysiert den bestehenden Code, erkennt Muster und schlägt Testfälle für typische und atypische Szenarien vor. Dadurch entsteht in kurzer Zeit eine breite Testbasis, die anschließend gezielt verfeinert werden kann.

Natürlich bedeutet dies auch, einen Teil der Detailkontrolle abzugeben, und diese Vorgehensweise ist verbunden mit der Verpflichtung, die generierten Vorschläge kritisch zu prüfen, was eine entsprechende Code-Erfahrung unabdingbar voraussetzt.

Doch wer diesen Schritt wagt, gewinnt oft an Geschwindigkeit, Vielfalt und Inspiration in der Testentwicklung, spart damit wertvolle Entwicklungszeit und erhält als Nebeneffekt in vielen Fällen auch eine qualitative Steigerung, da ►

```

C# EmailService.cs U C# EmailServiceUnitTests.cs 1, U
1_erste-schritte > C# EmailServiceUnitTests.cs > EmailServiceUnitTests > TestSendEmailAsync_ValidParameters_ReturnsTrue
1 using Xunit;
2
3 namespace EmailServiceExample;
4
5 0 references
6 public class EmailServiceUnitTests
7 {
8     0 references
9     public async Task TestSendEmailAsync_ValidParameters_ReturnsTrue()
10     {
11         // Arrange
12         IEmailService emailService = new EmailService();
13         string to = "test@example.com";
14
15         // Act
16         bool result = await emailService.SendEmailAsync(to, "Test Subject", "Test Body");
17
18         // Assert
19         Assert.True(result);
20     }
21 }

```

Kontextbewusste Vorschläge durch GitHub Copilot (Bild 3)

C#-Sprachstandardisierungen, die vorher im Projekt vielleicht zu kurz gekommen sind, einfach mitkommen.

Das Ergebnis der teilweisen Verantwortungsabgabe an GitHub Copilot in Bild 4 ist eine vollständig getestete Klasse – inklusive abstrahierter Abhängigkeiten – mit rund 200 Zeilen Code, generiert in weniger als 15 Sekunden. Die Kontrolle verbleibt dennoch beim Entwickler, sodass Anpassungen jederzeit möglich sind, etwa um statt Moq das bevorzugte Mocking-Framework NSubstitute [5] einzusetzen (Bild 5).

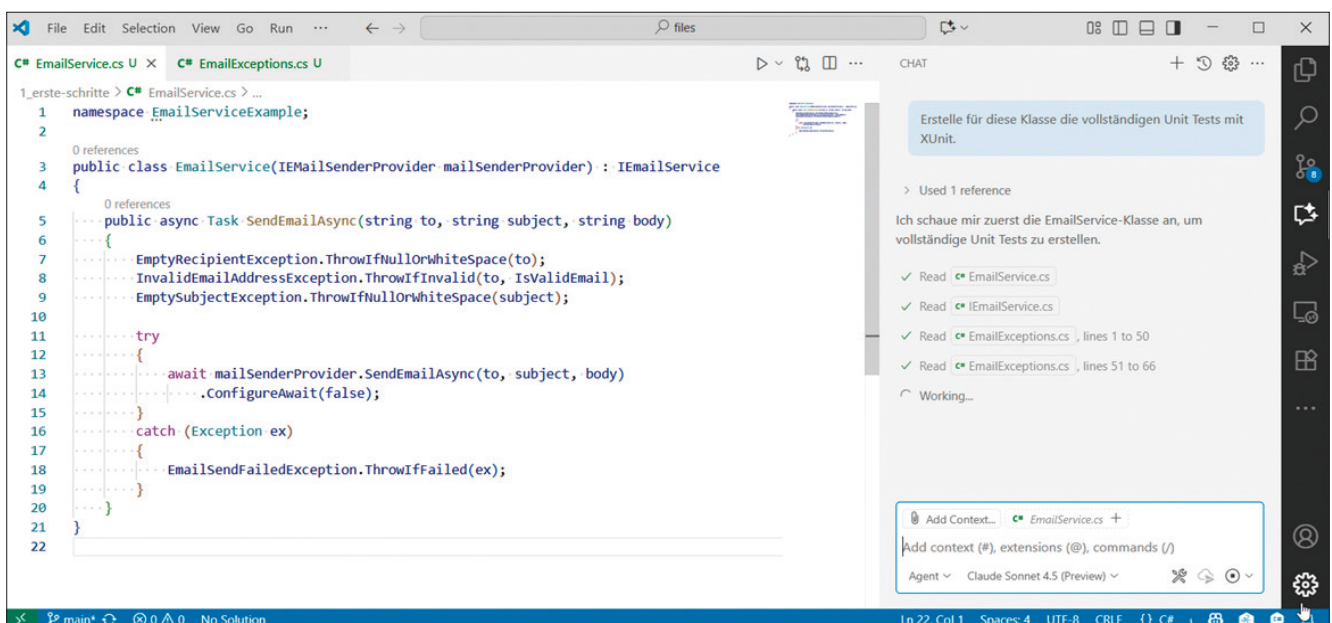
Assistierte versus vollautomatische Testgenerierung

Der Unterschied zwischen assistierter und vollautomatischer Testgenerierung mit GitHub Copilot liegt im Grad der Kon-

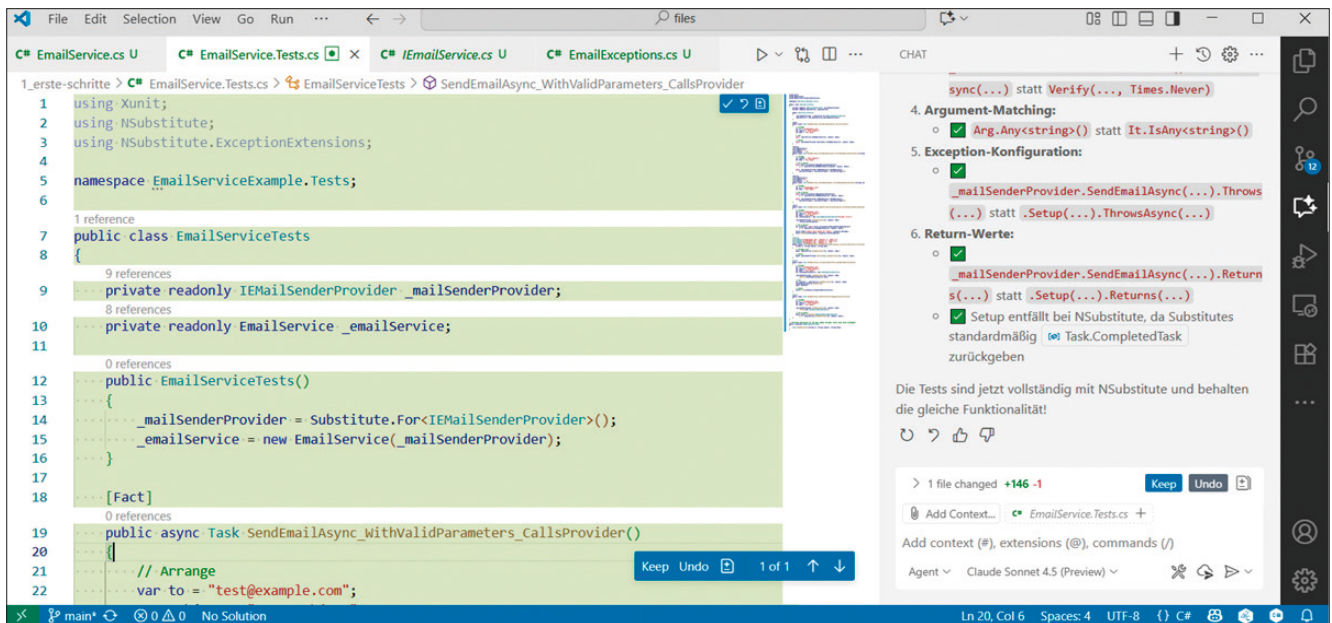
trolle und Kontextführung: Bei der assistierten Generierung arbeitet Copilot Seite an Seite mit den vom Entwickler gegebenen Signalen. Der Mensch definiert Testmethoden, beschreibt die Absicht über Kommentare oder Methodennamen und lässt Copilot anschließend den passenden Test-Inhalt ergänzen. Der Copilot-Vorschlag basiert auf dem vorhandenen Code-Kontext, wird aber stets manuell initiiert und überprüft. Diese Variante eignet sich vor allem in frühen Projektphasen oder in sicherheitskritischen Bereichen, in denen die Nachvollziehbarkeit einzelner Testfälle als wichtig empfunden wird.

Im Gegensatz dazu agiert Copilot bei der vollautomatischen Generierung eigenständiger: Hier reicht oft die Auswahl einer kompletten Klasse oder eines Moduls, um eine vollständige Testsuite erstellen zu lassen. Copilot analysiert dabei das gesamte öffentliche API der Klasse, identifiziert Methoden, Abhängigkeiten und typische Nutzungsszenarien und generiert daraus strukturierte Tests – inklusive Mocks und Assertions. Der Entwicklungsprozess verschiebt sich somit vom Schreiben einzelner Tests hin zur Überprüfung der generierten Ergebnisse.

In der Praxis bedeutet das: Die assistierte Variante liefert zielgerichtete Unterstützung beim Erstellen einzelner Testfälle, während die vollautomatische Generierung einen massiven Geschwindigkeitsvorteil bietet, wenn ganze Komponenten oder Services getestet werden sollen. Der Preis dafür



GitHub-Copilot-Prompt für das Generieren von Unit-Tests mit xUnit (Bild 4)



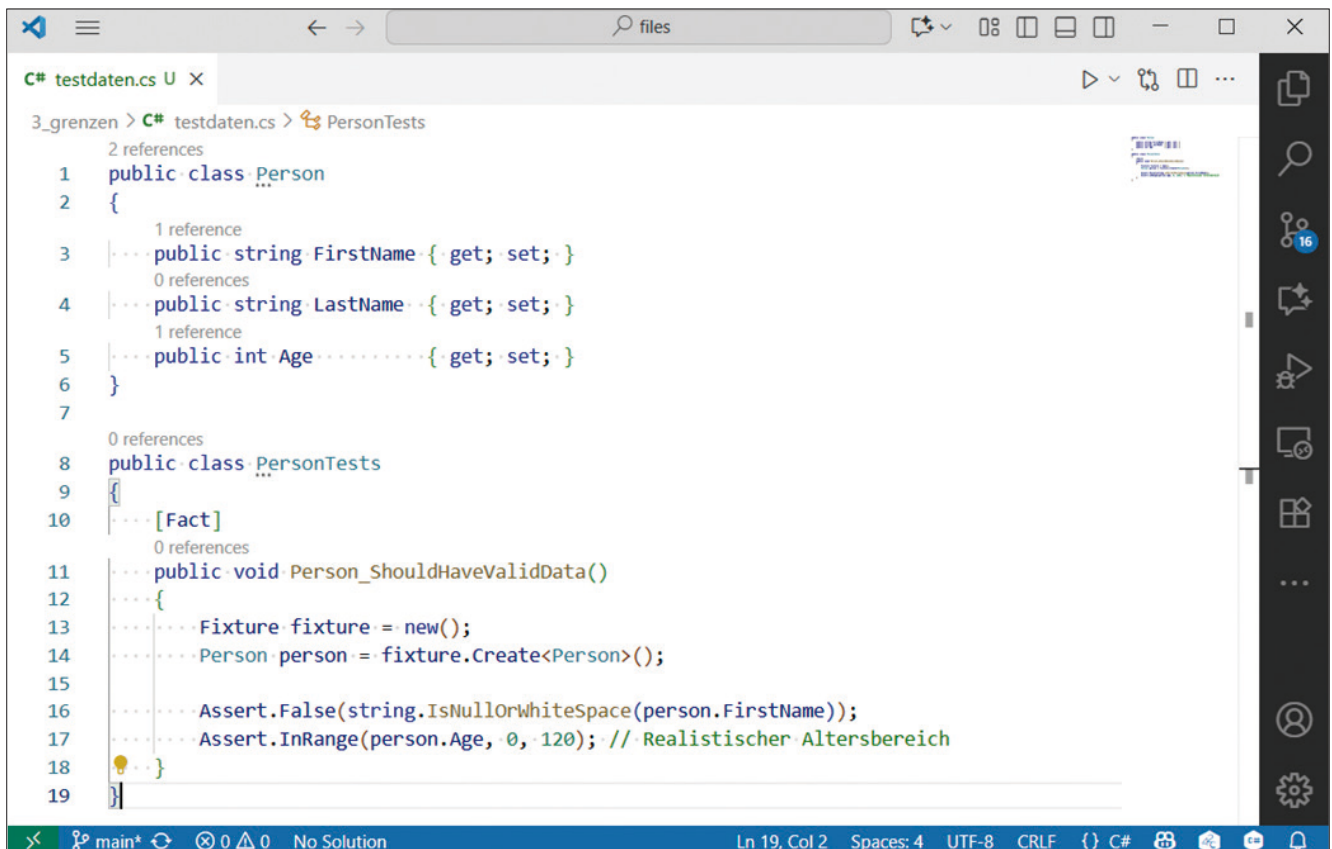
Vollständig generierte Unit-Tests mit anschließender Anpassung mit NSubstitute (Bild 5)

ist ein höheres Maß an Vertrauen in die Vorschläge des Modells, und damit verbunden auch die Notwendigkeit einer sorgfältigen Validierung.

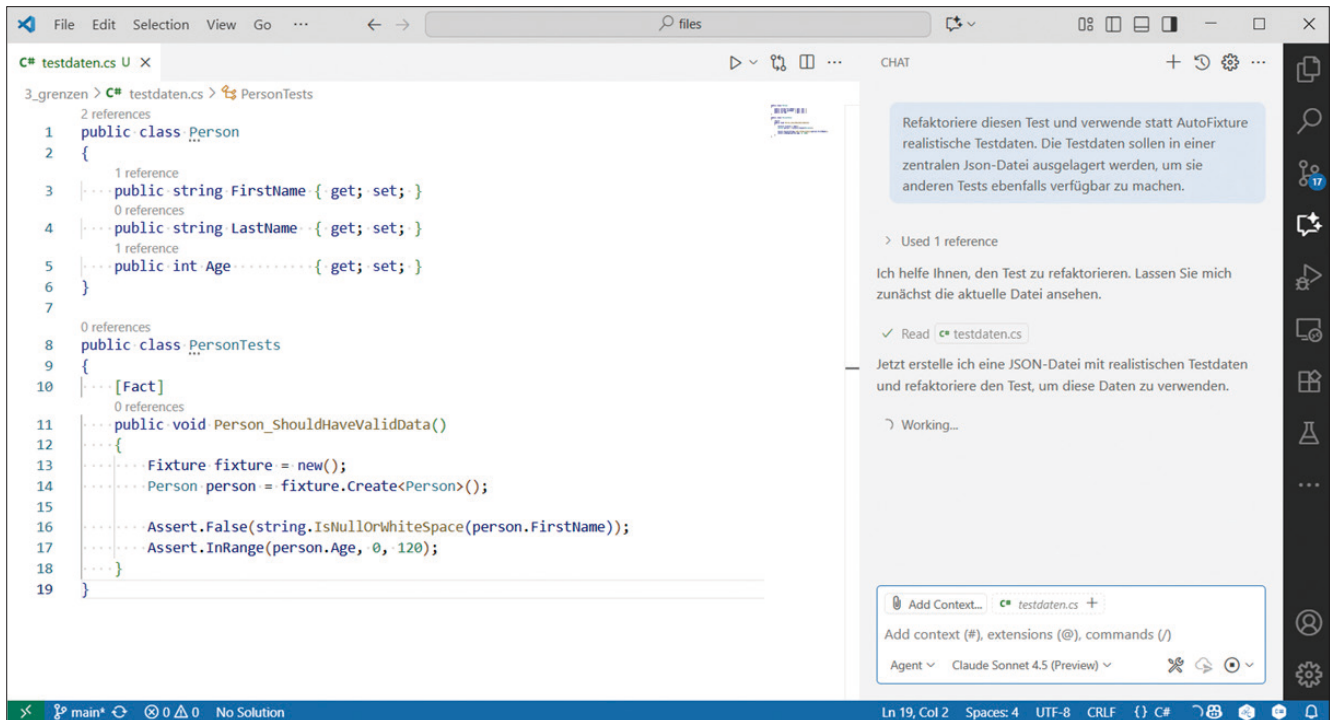
Beide Ansätze ergänzen sich: Während assistierte Tests Präzision und Kontrolle sichern, sorgt die vollautomatische Generierung für Breite und Tempo.

Erzeugen von Testdaten

Viele Tests, gerade Integrationstests zu Datenbanken oder anderen Datenquellen, bleiben auf der Strecke, weil Daten zum Validieren fehlen oder nur produktive Daten zur Verfügung stehen, die nicht zur Entwicklungszeit genutzt werden können oder dürfen. Um solche Testdaten zu erhalten, ►



Anwendungsbeispiel AutoFixture (Bild 6)



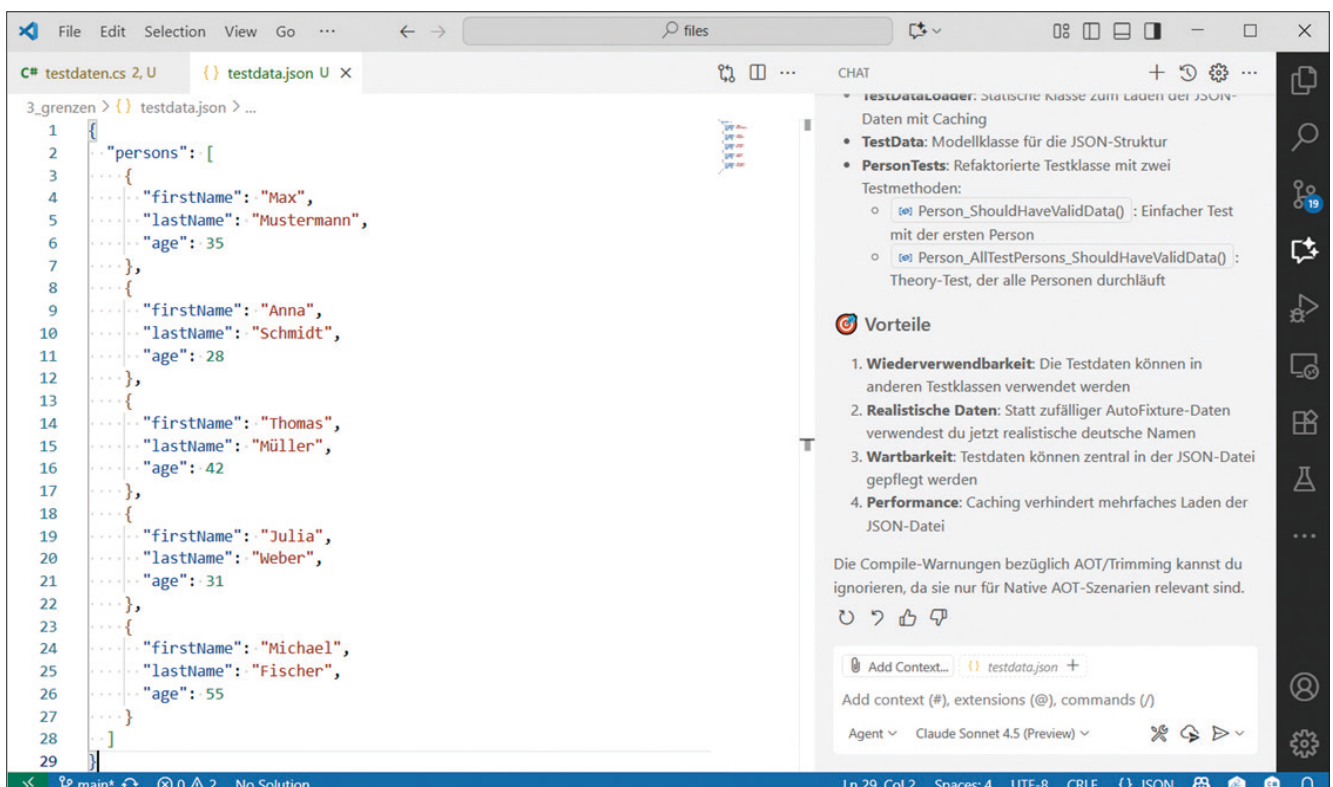
Prompt zum Refactoring von AutoFixture zu xUnit Testdata (Bild 7)

besteht der noch etablierte Weg darin, entweder produktive Daten aufwendig zu maskieren, oder man verwendet Bibliotheken wie Bogus [6] oder AutoFixture [7].

Ziel dieser Projekte beziehungsweise Bibliotheken ist es, den sogenannten Arrange-Teil eines Tests zu vereinfachen,

indem benötigte Instanzen mit sinnvollen Standardwerten gefüllt werden, ohne sie manuell anlegen zu müssen.

AutoFixture erstellt etwa auf Knopfdruck komplexe Objektgraphen – inklusive verschachtelter Eigenschaften und Listen – und spart so viel Boilerplate-Code, vergleiche Bild 6.



Von GitHub Copilot erzeugte Testdaten (Bild 8)

Ein wesentlicher Nachteil von AutoFixture liegt in der eingeschränkten Nachvollziehbarkeit und Realität der Testdaten. Da die Werte automatisch und meist zufällig generiert werden, ist es im Fehlerfall oft schwierig nachzuvollziehen, welche konkreten Eingabedaten zum Testergebnis geführt

schen Prompt. Während AutoFixture ausschließlich auf Typinformationen reagiert und beliebige Werte generiert, bezieht Copilot seine Vorschläge aus der tatsächlichen Geschäftslogik, den Variablen- beziehungsweise den Methodennamen, Kommentaren und vorhandenen Codebeispielen

```

3_grenzen > C# sample.cs > ...
6 references
1 public class DiscountService
2 {
3     // Einfache Logik: 10% Rabatt bei Beträgen über 100
4     public decimal CalculateDiscount(decimal amount)
5     {
6         => amount > 100 ? amount * 0.1m : 0;
7     }
8
9     1 reference
10    public class DiscountServiceTests
11    {
12        // Es wird nur das Verhalten getestet, nicht die Logik
13        [Fact]
14        public void CalculateDiscount_ShouldReturn10_WhenAmountIs200()
15        {
16            DiscountService service = new();
17            decimal result = service.CalculateDiscount(200);
18            Assert.Equal(10, result);
19        }
20
21        [Fact]
22        public void CalculateDiscount_ShouldReturn0_WhenAmountIs50()
23        {
24            DiscountService service = new();
25            decimal result = service.CalculateDiscount(50);
26            Assert.Equal(0, result);
27        }
28    }
  
```

GitHub Copilot erzeugt per Default vor allem Verhaltenstests für Code (Bild 9)

haben. Zudem kann die Verwendung von AutoFixture bestimmte Logiktests deutlich komplexer machen, da die generierten Werte nicht immer sinnvoll zum getesteten Szenario passen.

Besonders problematisch ist das fehlende Wissen über fachliche Zusammenhänge: AutoFixture erzeugt Werte maßgeblich auf Basis des Datentyps, ohne Kontext oder semantische Bedeutung. So kann beispielsweise eine *Person*-Klasse ein zufälliges Alter innerhalb der gesamten *int*-Range erhalten, das heißt im Wertebereich zwischen $-2\,147\,483\,648$ und $+2\,147\,483\,647$. Für viele Tests ist ein solcher Wert natürlich unsinnig, da ein Wert für das Alter einer Person heute typischerweise zwischen 0 und 120 liegen sollte.

GitHub Copilot ist prädestiniert dafür, Testdaten mit höherer Relevanz und Fachlichkeit zu erzeugen, als es bislang mit solchen Runtime-Generierungen möglich war, weil die KI nicht rein zufällig arbeitet, sondern den semantischen Kontext des Use Cases verstehen kann; sogar meist ohne spezifi-

und kombiniert diese Quellen mit dem Wissen im darunterliegenden Modell, zum Beispiel GPT-5. Dadurch entstehen Testdaten, die zum Verhalten der Anwendung passen und häufig realistischere Szenarien abbilden. So erkennt Copilot beispielsweise, dass eine Eigenschaft „Age“ typischerweise positive Werte im Bereich menschlicher Altersangaben erwartet oder dass ein „Price“-Feld nicht negativ sein darf.

Mit einem klar formulierten Prompt wie in Bild 7 setzt GitHub Copilot sofort an und greift dabei auf die xUnit-Funktionalität zur Verwaltung zentraler Testdaten zurück. Nur wenige Sekunden später liegen die gewünschten Testdaten vor – automatisch als JSON-Datei generiert (Bild 8).

Bei der Generierung der Testdaten, bestehend aus Vorname, Nachname und Alter, zeigt Copilot ein bemerkenswertes Kontextverständnis: Anhand der deutschen Eingabesprache wurden in diesem Beispiel, obwohl ich dem Kontext keine weiteren Anweisungen mitgegeben habe, nicht nur passende Alterswerte erstellt, sondern auch „realistische deutsche ▶

Namen“ automatisch gewählt. Copilot „denkt“ also in gewisser Weise mit; was jedoch durchaus – je nach Fall, keine Frage – auch unerwünscht sein kann.

Ein weiterer Vorteil bei der generativen Datengenerierung liegt in der zielgerichteten Datenkonsistenz: Copilot kann zusammenhängende Objektgraphen erzeugen, deren Werte logisch miteinander korrespondieren; etwa eine Person mit

Zumindest aktuell versteht Copilot den Zweck eines Systems selbst nicht, sondern leitet seine Vorschläge ausschließlich aus vorhandenen Code-Fragmenten, Kommentaren und gelernten Mustern ab. Auf diese Weise entstehen zwar formal korrekte, aber durchaus auch inhaltlich fehlerhafte Tests, wenn die fachliche Logik nicht eindeutig aus dem Quellcode hervorgeht.

```

3_grenzen > C# sample.cs > ...
2 references
1 public class DiscountService
2 {
3     // Einfache Logik: 10% Rabatt bei Beträgen über 100
4     public decimal CalculateDiscount(decimal amount)
5     {
6         => amount > 100 ? amount * 0.1m : 0;
7     }
8
9     0 references
10    public class DiscountServiceTests
11    {
12        [Theory]
13        [InlineData(200, 20)]
14        [InlineData(150, 15)]
15        [InlineData(50, 0)]
16        public void CalculateDiscount_ShouldApplyTenPercent_WhenAmountExceedsThreshold(
17            decimal amount, decimal expected)
18        {
19            DiscountService service = new();
20            decimal result = service.CalculateDiscount(amount);
21            Assert.Equal(expected, result);
22        }
23    }

```

Die Geschäftslogik als überprüfbare Bedingung (Bild 10)

gültigem Namen und dazu passender E-Mail-Adresse. Dadurch werden Testfälle aussagekräftiger und erfordern weniger manuelle Nacharbeit. Während AutoFixture also vor allem syntaktisch korrekte, aber fachlich zufällige Daten produziert, liefert Copilot testrelevante Eingaben, die sich an der Intention und Funktionsbedeutung des Codes orientieren.

Das Ergebnis sind Tests, die nicht nur schneller entstehen, sondern auch eine deutlich höhere inhaltliche Qualität aufweisen und sich auf die Fachlogik des Codes auch in sehr komplexen Datenszenarien anwenden lassen.

Grenzen von GitHub Copilot und KI

So beeindruckend die Ergebnisse von GitHub Copilot bei der Testgenerierung auch sind: Die Technologie hat klare Grenzen. Die KI arbeitet an dieser Stelle nicht immer kontextsensitiv und auch nicht kontextbewusst im menschlichen Sinne.

Ein häufiges Problem ist die Scheinsicherheit: Copilot erzeugt häufig Tests, die zwar erfolgreich durchlaufen, aber keine echte Aussagekraft besitzen. Beispielsweise werden erwartete Werte hart codiert, ohne sie aus den Fachregeln abzuleiten. Solche Tests geben ein Gefühl von Stabilität, decken aber keine realen Fehler ab.

Ebenso neigt Copilot manchmal dazu, offensichtliche Fälle zu bevorzugen und komplexe oder seltene Randbedingungen zu übersehen (vergleiche Bild 9).

Statt feste Rückgabewerte zu prüfen, sollte der Test das beabsichtigte Verhalten der Methode absichern. Entscheidend ist also, dass die Definition des erwarteten Ergebnisses aus der zugrunde liegenden Fachlogik abgeleitet wird; nicht aus der aktuellen Implementierung. Der Test muss damit die Regel selbst ausdrücken („10 % Rabatt ab 100 €“), anstatt nur das momentane Verhalten des Codes zu bestätigen.

Ein korrekter Test formuliert die Geschäftslogik als überprüfbare Bedingung: Wird der Schwellenwert überschritten, muss ein bestimmter Prozentsatz berechnet werden; andernfalls darf kein Rabatt gewährt werden (Bild 10). Der Test beschreibt das Verhalten der Methode aus fachlicher Sicht – unabhängig davon, wie der Code intern umgesetzt ist. So bleibt er auch dann gültig, wenn die Implementierung verändert oder optimiert wird.

Dieses Vorgehen erzeugt verhaltensbasierte Tests, die echte Sicherheit bieten: Der Test schlägt fehl, sobald die Fachregel verletzt wird, selbst wenn der Code syntaktisch weiterhin korrekt kompiliert. Auf diese Weise wird verhindert, dass scheinbar stabile, aber inhaltlich falsche Tests grün bleiben.

Auch bei Abhängigkeiten ist Vorsicht geboten: Wenn externe Systeme, Datenbanken oder APIs nicht sauber abstrahiert sind, schlägt Copilot teils unvollständige oder ineffiziente Mock-Implementierungen vor. GitHub Copilot neigt heute in solchen Situationen sehr zu Halluzination: Ohne klar definierte Interfaces oder Test-Doubles wird die generierte Test-Suite schnell qualitativ minderwertig.

Schließlich darf nicht vergessen werden, dass Copilot keine Qualitätsgarantie bietet: Copilot kann Entwicklungszeit verkürzen und Denkanstöße liefern, ersetzt aber keine fundierte Teststrategie. Architekturentscheidungen, fachliche Korrektheit und Wartbarkeit bleiben – zumindest aktuell – Aufgaben des Entwicklers. Nur wer die generierten Vorschläge kritisch prüft, konsolidiert und im Rahmen etablierter Testing-Praktiken einsetzt, kann das Potenzial von KI-gestütztem Testen wirklich ausschöpfen.

Testbarkeit als Voraussetzung für Copilot

Ein wesentlicher Aspekt, der sich bei der automatischen Generierung von Tests und Testdaten durch GitHub Copilot zeigt, ist die Abhängigkeit von gut testbarem Quellcode: Auch Copilot kann nur dann hochwertige Tests vorschlagen, wenn die zugrunde liegende Architektur dies zulässt.

Fehlt eine klare Trennung zwischen Geschäftslogik und Infrastruktur, werden Abhängigkeiten direkt instanziiert statt über Interfaces oder besitzen Methoden zu viele Seiteneffekte, gerät auch Copilot an seine Grenzen und erzeugt keine guten, bisweilen sogar schlechte Tests. Die KI kann zwar Code analysieren und Muster erkennen, aber Copilot wird innerhalb des Prompts keine untestbare Struktur „reparieren“, sofern dies nicht explizit gefordert wird. Copilot wird auch nicht „Nein“ sagen oder Vorarbeiten aktiv vorschlagen: GitHub Copilot wird auf Basis von schlechtem beziehungsweise qualitativ verbesserungswürdigen Code ganz einfach auch sehr verbesserungswürdige Tests erzeugen. Schlechte Architekturentscheidungen, enge Kopplungen und fehlende Schnittstellen führen dazu, dass Copilot generische oder unvollständige Tests vorschlägt, oft ohne fachliche Aussagekraft.

Umgekehrt profitiert das Modell stark von sauberem, modular aufgebautem Code. Methoden mit klaren Eingabe- und Rückgabewerten, Unions anstelle von Exceptions, Abhängigkeiten über Interfaces und gut benannte Klassen liefern die kontextuellen Signale, die Copilot benötigt, um sinnvolle Testfälle zu generieren. Eine gute Testbarkeit des Codes

bleibt somit die Grundvoraussetzung für sinnvolle KI-Unterstützung. Copilot wird aktuell nur dann Tests effizient erarbeiten, wenn in der Codebasis – entweder durch den Menschen oder ebenfalls durch gezieltes Prompting durch Copilot – zuvor die nötigen architektonischen Voraussetzungen geschaffen wurden; also klar strukturierte, deterministische und isolierbare Komponenten. Dies gilt aber generell für alle Arten der Code-Generierung durch Copilot.

Fazit der KI-gestützten Unit-Tests

GitHub Copilot hat sich als echter Produktivitätsbooster für das Erstellen und Erweitern von Unit- und Integrationstests erwiesen und lässt sich hervorragend mit xUnit kombinieren. Ob beim Refactoring, beim Schreiben neuer Tests oder beim Generieren von Testdaten: Copilot unterstützt in allen Phasen effizient. Viele der bislang mühsamen und monotonen Aufgaben im Testalltag lassen sich damit in Sekunden erledigen.

Trotzdem bleibt Erfahrung unverzichtbar: Nur wer die generierten Vorschläge versteht und sie bewerten sowie gezielt steuern kann, wird langfristig qualitativ hochwertige Tests und damit auch qualitativ hochwertige Software und Softwareprodukte erhalten. Copilot liefert heute hervorragende Impulse, aber keine automatische Garantie für inhaltlich korrekte oder fachlich vollständige Prüfungen.

Besonders deutlich wird: Die Qualität der generativ erstellten Tests hängt unmittelbar von der Qualität des zugrunde liegenden Codes ab. Gut strukturierter, entkoppelter und refaktorisierter Code ermöglicht Copilot erst, sinnvolle Tests zu erzeugen. Wer seine Architektur sauber hält, profitiert doppelt: durch schnellere Vorschläge und durch klarer nachvollziehbare Testergebnisse.

GitHub Copilot ist damit mehr als ein Werkzeug. Copilot agiert wie ein digitaler Kollege, der Routinearbeit übernimmt, Denkanstöße liefert und Entwickler von lästigen Tätigkeiten entlastet. Zaubern kann er zwar (noch) nicht, aber er bringt das Testen auf ein neues, intelligentes Niveau. ■

[1] GitHub Copilot, <https://github.com/features/copilot>

[2] IntelliCode,

<https://visualstudio.microsoft.com/services/intellicode/>

[3] using-Direktive, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive>

[4] xUnit, <https://xunit.net/>

[5] NSubstitute auf GitHub, <https://nsubstitute.github.io/>

[6] Bogus auf GitHub, <https://github.com/bchavez/Bogus>

[7] AutoFixture auf GitHub,

<https://github.com/AutoFixture/AutoFixture>



Benjamin Abt

arbeitet als Principal Software Engineer bei der MediaLesson GmbH in Pforzheim, betreibt das C#-Forum myCSharp.de und tritt als Trainer sowie Sprecher auf Konferenzen und User Groups auf. Seit 2015 ist er Microsoft MVP für .NET.
mail@benjamin-abt.com

TESTEN VON LOW-CODE-/NO-CODE-APPS IM .NET-UMFELD

Inkrementelles Testen mit Power Fx

Microsoft unterstützt eine große Bandbreite an Technologien und Tools für das Testen von Low-Code-/No-Code-Apps, die mit Power Apps entwickelt wurden.

Der Quellcode von Low-Code-/No-Code-Apps in Microsofts Power Platform basiert auf der Programmiersprache Power Fx [1]. Microsoft stellt verschiedene Technologien und Werkzeuge zur Verfügung, um Softwaretests zu realisieren, die Power Fx nutzen. Diese umfassen die Programmierung und Automatisierung von Tests sowie deren Integration in professionelle CI/CD-Umgebungen und die .NET-Welt. Zudem bietet die Power Platform ein Konzept mit Schnittstellen zur Einbindung von Softwaretests in individuelle Entwicklungsprozesse. Damit hat ein Team bei Microsoft begonnen, die Power Platform und ihre Tools stärker in die .NET- und Azure-Umgebungen zu integrieren.

Integration in die gängige Entwicklerwelt

Power Fx ist eine auf Produktivität und Effizienz ausgerichtete Low-Code-/No-Code-Sprache von Microsoft, die sich an Excel-Formeln orientiert. Sie ist in Tools der Power Platform wie Power Apps Studio (Maker Portal) und Power Automate integriert; außerdem lassen sich Power-Fx-Apps auch in Power BI einbetten. Zudem gibt es in diesen Tools Schnittstellen, um Power Fx zu exportieren. Da Power Fx primär formelbasiert ist, sind keine tiefen Programmierkenntnisse nötig, und auch Mitarbeiter mit geringen IT-Fähigkeiten (Citizen Developer) können damit schnell Business-Apps erstellen – Stichwort: Demokratisierung der Softwareentwicklung.

Power Fx stellt in erster Linie eine deklarative Formelsprache dar, die – angereichert um imperative, funktionale, Datenfluss-orientierte und Event-gesteuerte Konstrukte – zusätzliche Features bietet. Zudem besitzt Power Fx eine große

Mächtigkeit, um leistungsstarke Business-Apps zu realisieren und komplette Arbeitsabläufe zu automatisieren. Um derartige mit Low Code/No Code programmierte Apps beziehungsweise Komponenten in gängigen Sprachen und Technologien der professionellen Softwareentwicklung wie .NET oder JavaScript/TypeScript mit HTML/CSS zu nutzen, stellt Microsoft geeignete Technologien bereit.

Die Integration der traditionellen Entwicklerwelt mit der Low-Code-/No-Code-Welt erhöht deutlich das Leistungspotenzial der Entwicklung, insbesondere durch den Einbezug von KI-Assistenten. Deshalb ist Power Fx auch für Profi-Entwickler empfehlenswert: Die Kombination von Power Fx mit .NET/TypeScript verbindet die Produktivität und Effizienz von Low Code/No Code mit der Flexibilität und Leistungsfähigkeit von .NET/JavaScript – Microsoft spricht vom Fusion-Development. Für robuste, skalierbare Anwendungen lässt sich die Azure-Plattform einbinden. Wichtige Schnittstellen, Technologien und Verfahren unterstützen diese Kopplung:

- **Custom Connectors:** Mit Power Apps [2] erstellte benutzerdefinierte Konnektoren nutzen .NET-APIs, um Daten und Funktionen zwischen Power Fx und .NET-Anwendungen auszutauschen. Diese .NET-APIs können in ASP.NET (Core) oder anderen .NET-Technologien wie EF Core oder Blazor geschrieben sein. Auch die umgekehrte Richtung, also der Einsatz von Power-Fx-/Power-Apps-APIs in .NET-Anwendungen, ist möglich (Preview-Status).
- **Dataverse:** PowerShell-, JavaScript- und .NET-Entwickler können auf Daten, Tabellen und Spalten von Dataverse zugreifen und diese bearbeiten. Dabei stehen die Schnittstellen Web API, SDK for .NET und der Endpunkt für den tabellarischen Datenstrom in Dataverse (TDS) zur Abfrage von Daten mit SQL zur Verfügung.
- **Azure Functions:** Eine mit Power Apps entwickelte Business-App greift über HTTP-Anfragen auf in .NET geschriebene Azure Functions mit komplexer Logik zur Verarbeitung der Daten zu.
- **Einbettung in .NET-Anwendungen:** Die Benutzeroberfläche von mit Power Apps realisierten Business-Apps kann innerhalb einer .NET-Anwendung verwendet werden, dabei erfolgen die Logik und die Datenverarbeitung in .NET.
- **Power Apps Component Framework (PCF):** In JavaScript/TypeScript mit HTML/CSS entwickelte UI-Komponenten kön-

• Du möchtest gerne mehr erfahren?

Das freut uns, denn ab Erscheinen des vorliegenden Artikels veröffentlichen wir auf developer-world.de/dwx-insights weitere Beiträge, die den Artikel fortsetzen und ausschließlich online zur Verfügung stehen. Derzeit geplant sind folgende Beiträge:

- Mit Lösungen (Solutions) arbeiten
- UI-Tests entwickeln und ausführen
- Deployment mit Power Platform Pipelines

nen in Business-Apps über Power Apps integriert werden. Damit lassen sich vorhandene UI-Komponenten der TypeScript-Welt und die mit ihnen verknüpften Funktionen in mit Power Apps programmierten Business-Apps nutzen.

Power Platform CLI

Microsoft hat das Power Platform CLI, kurz *pac*, entwickelt, um die Arbeit mit der Power Platform zu erleichtern. Im Fokus dieses Befehlszeilen-Tools steht die Automatisierung von Entwicklungs- und Administrations-Workflows. Mit fortschreitendem Entwicklungsprozess gewinnt die kontinuierliche Integration und Auslieferung (CI/CD) zunehmend an Bedeutung. Das *pac* ermöglicht die Integration eines Source-Code-Control-Systems zur Versionierung und das Aufsetzen vollständiger DevOps-Pipelines im Rahmen von CI/CD.

Microsoft bietet das *pac* für die Betriebssysteme Windows, Linux und macOS an. Die Installation erfolgt über die VS-Code-Extension (Power Platform Tools), den .NET-Befehl *dotnet tool install* (für alle drei Betriebssysteme) oder den MSI (Microsoft Installer, nur für Windows). Die empfehlenswerteste Methode ist die Installation über .NET mit dem Befehl *dotnet tool install --global Microsoft.PowerApps.CLI.Tool* (Bild 1). Diese Methode ermöglicht den vollständigen Zugriff auf das *pac* in allen Command Prompts und Terminals. Allerdings sind derzeit die *pac data*- und einige *pac package*-Kommandos für die Arbeit mit Dataverse nur unter Windows verfügbar.

Microsoft unterteilt die umfangreiche Anzahl an *pac*-Befehlen in über zwanzig Befehlsgruppen. Jede Gruppe umfasst Kommandos, die sich auf spezifische Aufgaben konzentrieren. Der Befehl *pac help* listet alle Gruppen auf; zusätzlich ist im Internet eine Kommando-Referenz mit detaillierten Erläuterungen verfügbar. Zu den zentralen Ressourcen, die mit dem *pac* bearbeitet werden, gehören Autorisierungen, Solutions (Lösungen), Dataverse, Environments (Umgebungen), Power-Apps-Portale (Power Pages), ALM-Pipelines (Application Lifecycle Management) und Code-Komponenten.

Manche *pac*-Befehle erfordern eine Benutzer-Authentifizierung, um auf Ressourcen zuzugreifen. Der Befehl *pac auth create* ermöglicht eine einfache und schnelle Authentifizierung bei der Power Platform, wobei verschiedene Parameter, etwa ein spezifischer Mandant, verwendet werden. Der *create*-Befehl erstellt individuelle Einstellungen für das Autorisierungsprofil basierend auf den übergebenen Argumenten. Der Befehl *pac auth list* gibt alle vorhandenen Profile mit ihren Einstellungen aus, während der *select*-Befehl ein anderes Autorisierungsobjekt als aktuelles Arbeitsprofil aktiviert und *clear* alle vorhandenen Profile von der Maschine entfernt.

Solutions für Ablage und Transport von Tests

Microsoft Power Platform unterstützt mehrere logisch getrennte Organisationen (Mandanten, englisch Tenants) in einer zentral betriebenen Cloud-Umgebung. Jeder Mandant hat eigene

Benutzer, Daten und Konfigurationen – ohne separate Server betreiben oder Installationen vornehmen zu müssen. Innerhalb eines Mandanten lassen sich Umgebungen (Environments) anlegen, die als Container für Ressourcen wie Apps, Workflows, Datenbanken (Dataverse) und Konnektoren dienen. So können Entwicklungs-, Test- und Produktionsumgebungen strikt voneinander getrennt betrieben werden. Die Verwaltung erfolgt zentral im Power Platform Admin Center [3] mit Einstellungen zu Region, Sicherheit und Einsatzzweck.

Aus Sicht der Softwareentwicklung sind die wichtigsten Merkmale einer Umgebung der Typ und die Phase im Lebenszyklus (ALM). Für Entwicklungs- und Testzwecke werden separate Environments eingerichtet, um eine saubere Trennung von Ressourcen der Entwicklung, Test und Produktion sicherzustellen. Dies ermöglicht gründliche Tests von Business-Apps, bei denen Funktionsfähigkeit und Datenisolation gewährleistet bleiben, ohne dass Konflikte zwischen Umgebungen oder Mandanten entstehen können.

Für produktive Geschäftsanwendungen wird der Typ *Produktion* genutzt, während die Umgebungen *Sandbox* und *Entwicklung* für Entwicklung und Tests dienen. *Trial*-Umgebungen sind 30 Tage verfügbar und ermöglichen das Kennenlernen der Power Platform. *Default* ist eine automatisch angelegte Standardumgebung pro Mandant oder Dataverse-Abo für persönliche Erkundungen; diese kann nicht gelöscht werden! Im ALM werden Deployments über Ebenen/Stages wie Entwicklung, Test und Produktion gesteuert, siehe auch Kasten **ALM für die Entwicklung mit Low-Code-/No-Code-Technologie**. Bei entsprechender Versionierung der Umgebungen lassen sich diese für Updates, Backups oder Tests sichern, wiederherstellen oder löschen.

Tools zur Kopplung von Low Code/No Code mit der .NET-Entwicklerwelt

Grundsätzlich macht Microsoft über ein öffentlich zugängliches NuGet-Paket die Power-Fx-Engine (Bibliothek, Parser, Evaluator, Typen) verfügbar. Entwickler installieren dieses ►

```

C:\Users>dotnet tool list --global
Package Id      Version      Commands
-----
dotnet-ef       9.0.0       dotnet-ef
excubo.webcompiler 4.0.11      webcompiler
microsoft.web.librarymanager.cli 2.1.175    libman

C:\Users>dotnet tool search Microsoft.PowerApps.CLI.Tool
Package ID      Latest Version  Authors      Downloads  Verified
-----
microsoft.powerapps.cli.tool 1.45.3         Microsoft   665548     x
xrmdocumentationgeneratorcorecli 1.2.60        Ramy Victor  23926

C:\Users>dotnet tool install --global Microsoft.PowerApps.CLI.Tool
You can invoke the tool using the following command: pac
[tool 'microsoft.powerapps.cli.tool' (version '1.45.3') was successfully installed.]

C:\Users>dotnet tool list --global
Package Id      Version      Commands
-----
dotnet-ef       9.0.0       dotnet-ef
excubo.webcompiler 4.0.11      webcompiler
microsoft.powerapps.cli.tool 1.45.3      pac
microsoft.web.librarymanager.cli 2.1.175    libman
  
```

Der Befehl „*dotnet tool list*“ zeigt alle eingerichteten .NET-Tools in einer Liste an. Die Installation weiterer Tools erledigt der Befehl „*dotnet tool install*“ (Bild 1)

NuGet-Paket wie üblich mit dem Befehl `dotnet add package Microsoft.PowerFx.Core` in ein vorhandenes .NET-Projekt. Eine direkte Unterstützung für C# oder eine der anderen gängigen Programmiersprachen gibt es für Power Fx derzeit noch nicht in Visual Studio oder VS Code. Dies ergibt auch Sinn, da die eigentliche Programmierung mit Power Fx nicht in einer der beiden IDEs erfolgt, sondern vielmehr in den gängigen Werkzeugen der Power Platform wie Power Apps Studio (Maker Portal).

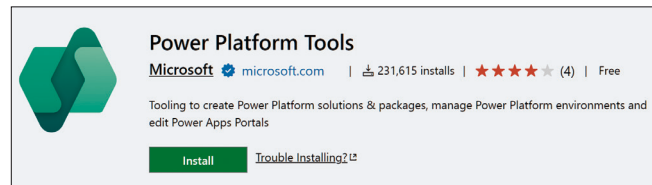
Microsoft bietet jedoch verschiedene Optionen an, um Power-Fx-Formeln oder -Projekte zu pflegen und zu verwalten. Dies umfasst sowohl komplette Power-Platform-Umgebungen als auch Anlage, Build und Deployment von Solutions, Packages und Portals. Für diesen Zweck stellt Microsoft für VS Code die Erweiterung Power Platform Tools zur Verfügung (Bild 2). Deren Installation in VS Code macht zusätzlich das Power Platform CLI in einem Terminal von VS Code verfügbar – jedoch nicht in eigenständigen normalen Windows- oder PowerShell-Terminals. Achtung: Die gleichnamige Erweiterung Power Platform Tools für Visual Studio richtet sich ausschließlich auf die Entwicklung von Power-Platform-Plug-ins!

Nach Installation dieser Erweiterung erhält VS Code am linken Bildschirmrand in der vertikalen Aktivitätsleiste (Activity Bar) einen neuen Eintrag: das Power-Platform-Icon. Positioniert der Entwickler den Mauszeiger darauf, so blendet VS Code als Hover-Help den Text *Power Platform* ein. Sollte noch kein Benutzer im Admin Center der Power Platform angemeldet sein, so verlangt ein Klick auf das *Power Platform*-Icon eine Authentifizierung. Alle einmal autorisierten Benutzer erkennt VS Code über die Power-Platform-Tools-Erweiterung automatisch. Deshalb zeigt ein Klick auf das Icon alle angemeldeten Autorisierungsprofile in einer Liste an.

Die Erweiterung koppelt den Low Code/No Code der Power Platform mit der .NET-Entwicklerwelt, sodass auch nach einem Neustart von VS Code alle bekannten Autorisierungsprofile erhalten bleiben. Eine lästige Neuansmeldung des Benutzers in der Power Platform entfällt – die Erweiterung macht alle benötigten Entwicklerressourcen in VS Code sofort zugänglich. Ein aktuell aktiviertes Autorisierungsprofil erkennt der Entwickler anhand eines Stern-Icons vor seinem Namen. Unterhalb der Profilliste erscheinen alle zugehörigen Environments und Solutions in einer Drop-down-Hierarchie. Nach der Auswahl einer Umgebung zeigt Power Platform Tools alle in ihr enthaltenen Solutions (Lösungen) an.

Test Studio – das Basiswerkzeug

Neben den reinen Werkzeugen für die Entwicklung von Business-Apps hat Microsoft ein Testing-Tool, Test Studio genannt, direkt in Power Apps Studio (der Power-Apps-IDE) integriert. Test Studio dient als Basiswerkzeug für die Automatisierung von Tests. Als Testwerkzeug eignet es sich für die Realisierung von automatisierten Benutzeroberflächen- und Funktionstests. Mit Test Studio lassen sich derartige Tests direkt innerhalb von Power Apps Studio



Der VS-Code-Marketplace enthält die Erweiterung Power Platform Tools (Bild 2)

erstellen, verwalten und ausführen. Zu den wichtigsten Einsatzszenarien von Test Studio gehören:

- **Automatisierung von User-Interface-Tests:** Anhand von Benutzer-Interaktionen (Klicks, Texteingaben et cetera) mit der Business-App zeichnet Test Studio spezifische Testfälle auf (Recording) und speichert diese ab, um sie später wieder automatisch abspielen zu können. Damit lassen sich Controls (Buttons, Textfelder et cetera) prüfen, ob sie erwartungsgemäß existieren und reagieren. Dies ermöglicht gleichzeitig eine schnelle Überprüfung des GUI-Designs der App und ihrer Datenquellen.
- **Durchführung von Integrations- und Regressionstests:** Nach Änderungen in der Business-App dienen die programmierten Tests der Überprüfung auf Fehlerfreiheit kritischer Anwendungsfälle. Eventuelle Seiteneffekte werden früh erkannt, die Release-Stabilität wird erhöht und die Anzahl manueller Tests minimiert.
- **Data-Driven Testing:** Zielt bei der Automatisierung darauf ab, vorhandene Testfälle für unterschiedliche Datensätze wiederzuverwenden. Dies ermöglicht eine Parametrisierung der Eingabewerte, verknüpft mit den erwarteten Ausgaben. Data-Driven Testing erreicht eine hohe Abdeckung zahlreicher Anwendungsfälle und reduziert redundante Testfälle – die Generierung der Testdaten muss derzeit jedoch mittels eigener, individueller Programmierung realisiert werden.

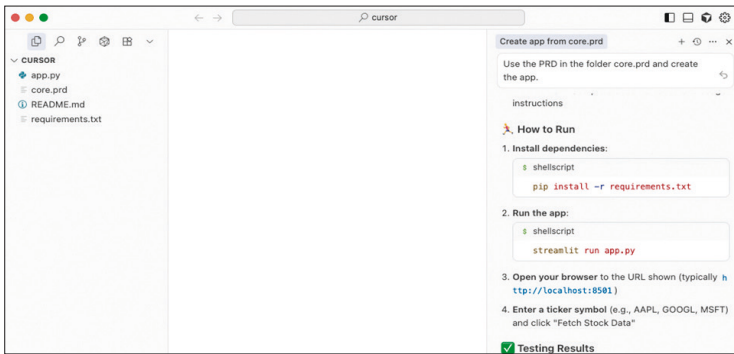
Dokumentation und Nachweisbarkeit

Test Studio protokolliert Ausführung und Ergebnisse der realisierten Tests. Zur Beschreibung und Strukturierung von Testfällen stellt Test Studio dem Tester verschiedene Hierarchieelemente (Bild 3) zur Verfügung:

- **Test-Suite:** Stellt die oberste Einheit für die Strukturierung der zu testenden Funktionsbereiche oder Release-Zyklen

ALM für die Entwicklung mit Low-Code-/No-Code-Technologie

In der Softwareentwicklung kommen verschiedene Modelle für das Application Lifecycle Management (ALM) zum Einsatz. Die bekanntesten sind das Wasserfall-, das Spiral- und das V-Modell. Für die Entwicklung von Business-Apps mit Low-Code-/No-Code-Technologie eignet sich davon eigentlich nur das Spiralmodell. Das Wasserfall-Modell ist zu stark sequenziell, gibt wenig Rückkopplung und besitzt eine zu strenge Abgrenzung der Phasen. Das V-Modell ist ebenfalls sequenziell und zu starr für die bei Low Code/No Code anzutreffenden kurzfristigen Änderungen. Am besten eignet sich eine leichtgewichtige Form des Spiralmodells, basierend auf agilen Prinzipien mit bewusst kurz gehaltenen Phasen und ausgerichtet auf Rückkopplungen (Feedbacks).



Oberste Ebene eines Testfalls in Test Studio stellt die Test-Suite dar, diese gliedert sich in Test-Cases und Test-Steps (**Bild 3**)

der Business-App dar. Dabei handelt es sich um eine Sammlung von Testfällen, die gemeinsam ausgeführt werden.

- **Test-Fall/Case:** Entspricht einer eigenständigen Einheit, die einen vollständigen Prüfablauf (Testfall) abbildet. Ein Test-Fall/Case umfasst in der Regel mehrere Test-Schritte/Steps und referenziert im Idealfall mehrere Eingabe- und Ausgabedaten – möglichst über Data-Driven Testing.
- **Test-Schritt/Step:** Bildet die kleinste Einheit innerhalb eines Testfalls, diese besteht aus einer Aktion und optionalen Validierungen (Asserts). Typische Aktionen stellen Benutzerinteraktionen mit der Maus, Auswahl eines GUI-Controls, Warten auf das Erscheinen eines Elements, Texteingabe oder Navigation dar.
- **Validierung (Assertion):** Nimmt die Prüfung eines Zustands oder Werts nach einer Aktion vor, beispielsweise die Übereinstimmung von Texten, die Sichtbarkeit eines GUI-Controls oder die korrekte Berechnung von Datenwerten. Eine Assertion dient der Beurteilung des Ergebnisses einer Test-Durchführung.

Test Engine – das Framework für professionelle Testumgebungen

Microsoft stellt die Power Platform Test Engine [4] als Open Source in Ergänzung zu Test Studio bereit. Während Test Studio als GUI-basierte Lösung häufig von Citizen Developern und Fachtestern (Anwendungsexperten) genutzt wird, richtet sich die Test Engine primär an Testautomatisierer, DevOps-Ingenieure und Entwickler. Sie stellt ein SDK mit APIs, über das pac ein CLI und schließlich die Infrastruktur für automatisierte, skalierbare Testausführungen bereit. Kombiniert ermöglichen beide Tools eine umfassende Teststrategie: schnelle Aufnahme und Validierung mit Test Studio sowie skalierte, wiederholbare Ausführung über die Test Engine. Vorausgesetzt werden jedoch passende Strategien für Governance und Testdaten, damit professionelle Teams eine erweiterte und robuste Testumgebung erreichen.

Test Studio zeichnet UI-Schritte auf und exportiert Testartefakte über den Button *Download suite* als strukturierte JSON/YAML-Datei (**Bild 4**). Da es bisher kein standardisiertes Format für einen YAML-Testplan gibt, werden in der Praxis die Exportdateien angepasst oder in Testpläne überführt. Testfälle bestehen aus Testaktionen und Assertions, die Power-Fx-Ausdrücke einbetten können (zum Beispiel *Select*, *SetProperty*). Zusätzlich bietet die Test Engine testorientierte

Aktionen über das SDK wie Screenshot, Wait und Mocking zur Steuerung von Ablauf, Synchronisation und Simulation externer APIs – dabei handelt es sich nicht um neue Sprachelemente von Power Fx! Mit ihnen lassen sich umfangreiche, wiederholbare Tests erstellen und in CI/CD-Pipelines betreiben, ohne auf C# oder JavaScript zurückgreifen zu müssen.

Test Engine nutzt als Automatisierungsbibliothek das von Microsoft entwickelte Testing-Tool Playwright. Diese von Microsoft gewählte Realisierung ermöglicht eine Headless Execution. Damit führt die Test Engine ihre Testfälle ohne geöffnete Benutzeroberfläche aus, was den Verbrauch an Ressourcen re-

duziert und Interferenzen durch aktive User-Interface-Sitzungen vermeidet. Die Headless Execution ist ideal für die Integration der Testfälle in CI/CD-Pipelines und wird hierfür standardmäßig verwendet. Die realisierten Testfälle stellen die Ausgangsbasis für deren automatische Ausführung in Azure DevOps, GitHub-Actions oder anderen Pipelines dar. Notwendige Testläufe für Regressionstests starten automatisch bei jedem Build/Release. Anhand von Exit-Codes und den Ergebnisdateien erfolgt die Auswertung des Resultats eines Tests.

Testfälle mit der Test Engine und pac ausführen

Die Installation der Test Engine erfolgt anhand der Anleitung im zugehörigen GitHub-Repository von Microsoft. Playwright-Browser werden typischerweise über den Playwright-Installer (*npm/playwright*) installiert. Verlangt eine Installation eine PowerShell-Skriptausführung, so ist eine Änderung der PowerShell-ExecutionPolicy notwendig – ein *Set-ExecutionPolicy* sollte jedoch nur nach erfolgter Sicherheitsprüfung ausgeführt werden! Damit die Test Engine eine App automatisiert ansteuern kann, muss die jeweilige Power App bereits in der Zielumgebung veröffentlicht sein. Sie referenziert über CLI-Parameter den Environment-URL und lädt die App zur Laufzeit in einen Playwright-Browserkontext. ►

```
# yaml-embedded-languages: powerfx
testSuite:
  testSuiteName: Calculator
  testSuiteDescription: Verifies that the calculator app works.
  persona: User1
  appLogicalName: new_calculator_a3613

  testCases:
    - testCaseName: Default Check
      testSteps: |
        = Screenshot("calculator_loaded.png");
        Assert(Calculator_1.Number1Label.Text = "100", "Validate
        Assert(Calculator_1.Number2Label.Text = "100", "Validate
    - testCaseName: Check Addition
      testSteps: |
        = Select(Calculator_1.Add);
        Assert(Calculator_1.ResultLabel.Text = "200", "Validate
        Assert(Calculator_1.CalculatorResult = 200, "Validate
    - testCaseName: Check subtraction
      testSteps: |
        = Select(Calculator_1.Subtract);
        Assert(Calculator_1.ResultLabel.Text = "0", "Validate
        Assert(Calculator_1.CalculatorResult = 0, "Validate co
```

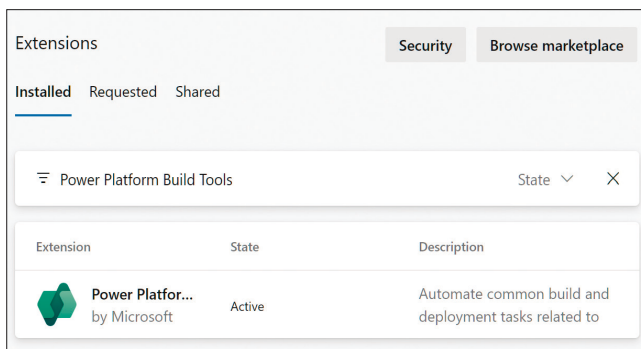
Eine YAML-Datei enthält die mit Power Fx spezifizierten Test-Steps für alle Test-Cases einer Test-Suite (**Bild 4**)

Testartefakte werden üblicherweise separat im Quellcode-Repository oder in Pipeline-Artefakten verwaltet.

Nach dem Anlegen eines Auth-Profiles mit `pac auth create` wechselt `pac auth select` zwischen Profilen. Um explizit eine Umgebung zu setzen, nutzt man entweder das `--environment`-Flag beim Auth-Profil oder `pac env select`. Der Befehl `pac env list` zeigt zugängliche Umgebungen an, `pac solution list` gibt die in der aktiven Umgebung gespeicherten Solutions (Lösungen) aus. `pac solution unpack` extrahiert die Solution-ZIP-Datei in lokale Quelldateien, `pac solution import` lädt sie in die Cloud der Power Platform hoch. Testpläne werden separat als YAML-Dateien im Repository oder per `pac test init` generiert und von der Test Engine eingelesen – sie sind nicht Teil der automatisch entpackten Metadaten.

In Projekten wird häufig eine JSON-Konfigurationsdatei (`config.dev.json`) genutzt, um Umgebungs- und Testparameter zentral zu halten. Ein Installations-/Build-Skript liest diese Datei ein und erzeugt daraus die Argumente für die Kommandozeile der Test Engine. Es empfiehlt sich, im Projekt einen `config`-Ordner anzulegen und dort die Datei zu speichern. Ab PowerShell 5.1 erzeugt `$config = Get-Content -Path .\config\config.dev.json -Raw | ConvertFrom-Json` aus der Config-Datei die benötigten CLI-Argumente für die Power Platform Test Engine. Secrets sollten nicht unverschlüsselt in der Config-Datei gespeichert werden, stattdessen sind CI-Secrets oder Key Vault zu verwenden. Es gilt, das JSON-Schema an das aufrufende Skript zu koppeln und Pfad-/Encoding-Unterschiede zwischen lokalen und CI-Runners zu berücksichtigen.

Die Test Engine wird über den .NET-Runner (`dotnet PowerAppsTestEngine.dll`) oder das CLI-Plug-in (`pac test run`) gestartet. Testdefinitionen liegen üblicherweise als strukturierte YAML-Dateien (Bild 4) im DSL-Format von Power Fx vor, siehe auch Kasten **DSL-Format – Basis der Automatisierung**. Diese Dateien beschreiben Test-Suites, Test-Cases und deren Test-Steps; häufig gegliedert in: `suite -> cases -> steps`; sie enthalten optional Test-Einstellungen (zum Beispiel Playwright-Optionen) und Environment-Variablen sowie weitere Konfigurationsdaten. Test-Steps bestehen aus Aktionen und Assertions und können Power-Fx-Ausdrücke einbetten oder auf Engine-spezifische Aktions-DSL verweisen. Exakte Feldnamen und unterstützte Optionen richten sich nach dem aktuellen Test-Engine-Schema im zugehörigen Repository.



Das Feature „Manage Extension“ eines Azure-DevOps-Servers unterstützt Installation und Verwaltung von Plug-ins (Bild 5)

● DSL-Format – Basis der Automatisierung

Unter dem DSL-Format von Power Fx versteht man eine strukturierte, maschinell verarbeitbare und lesbare Darstellung von Power-Fx-Formeln. Als Domain-Specific Language (DSL) ist das Power-Fx-DSL-Format vor allem für Entwickler und Tools relevant.

Dabei handelt es sich um eine strukturierte Darstellung von Formeln und zugehörigen Metadaten in einer hierarchischen, JSON-ähnlichen Struktur. Das DSL-Format verwendet die Power Platform, um Formeln zu speichern, zu übertragen oder zu verarbeiten – etwa beim Import/Export von Apps oder beim Arbeiten mit dem Power Platform CLI. Dieses Format erleichtert Tools oder anderen Programmen die Analyse, Transformation und Generierung von daraus resultierenden zu automatisierenden Aufgaben.

Zur automatischen Ausführung von Testfällen mit Test Studio muss das Feature *Formula-level error management* in der Business-App eingeschaltet sein, siehe Kasten **Error-Management zur Testausführung einschalten**.

Unterstützung für verschiedene DevOps-Welten

Microsoft stellt für die Power Platform mehrere CI/CD-Werkzeuge bereit, die sich in Bedienkonzept und Integrationsgrad unterscheiden. Die gängigsten Varianten sind:

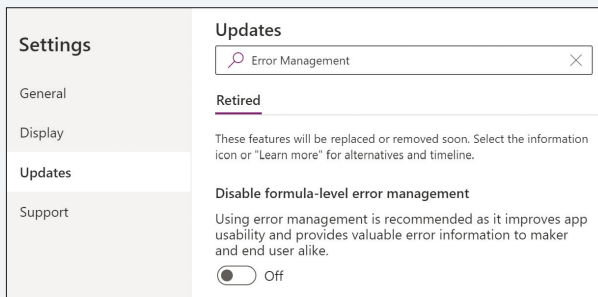
- **In-App Pipelines (auch Power Apps Pipelines):** Eine im Power Apps Maker integrierte, browserbasierte Release-Funktionalität für einfache ALM-Workflows, die direkt im User-Interface der Power Platform integriert ist. Das Feature befindet sich in der Public Preview, muss pro Tenant oder Umgebung aktiviert werden und unterstützt derzeit Dataaverse-Lösungen.
- **Power Platform CLI (pac) und Power Platform Build Tools:** Eine Sammlung von CLI-Befehlen und Azure-DevOps-Tasks für Source-driven ALM. Mit `pac` lassen sich Solutions exportieren, packen und importieren. Die Build Tools sind zwar für Azure DevOps optimiert, können aber in jeder CI/CD-Umgebung mit installiertem CLI eingesetzt werden.
- **GitHub Actions (offizielle Actions) [5]:** GitHub-basierte CI/CD-Workflows zur Automatisierung von Export, Pack, Import und Testläufen. Die Actions nutzen intern das Power Platform CLI und bieten optimierte Steps für GitHub-Workflows.

Alle Optionen arbeiten mit Pipelines/Workflows zur Automatisierung zusammen; die Wahl hängt von der vorhandenen Toolchain, den Anforderungen seitens Governance und den Teampräferenzen ab.

Die In-App/Standard-Pipeline richtet sich an Citizen Developer aus Fachabteilungen und unterstützt die Demokratisierung von Low Code/No Code. Sie bieten sofort nutzbare, klickbasierte Schritte zum Export/Import von Solutions und erfordern geeignete Environment- sowie Solution-Berechtigungen. Für einfache bis mittelkomplexe Apps sind sie schnell einzurichten und ermöglichen zügige Deployments zwischen Umgebungen. Hingegen sind feingranulares, Sourcen-getriebenes Versions-

● Error-Management zur Testausführung einschalten

Das Feature *Formula-level error management* muss zur automatischen Ausführung von Testfällen mit Test Studio in der Business-App eingeschaltet sein. Nach dem Öffnen einer mit Power Apps programmierten Business-App klickt man auf den unten stehenden Eintrag *Settings* in der linken Toolbar von Power Apps Studio. Es öffnet sich das in **Bild 6** gezeigte zugehörige Dialogfenster, dort enthält die linke Auswahlleiste den Eintrag *Updates*. Über dessen Suchleiste *Search* findet man schnell die Einstellung *Disable formula-level error management*; dessen zugehörigen Schalter stellt man auf *Off*.



Ein Schalter aktiviert das zur Ausführung von Testfällen benötigte Feature „Formula-level error management“ (Bild 6)

management, komplexe Erweiterungen und umfassende Automatisierungen eingeschränkter als bei CLI/DevOps-basierten Ansätzen (pac, Azure DevOps, GitHub Actions).

Die zusätzlich als App in Dataverse/Dynamics 365 bereitgestellte Power Platform Pipeline ermöglicht primär In-Tenant-Low-Code-Release- und -Deployment-Workflows für Power Apps/Solutions. Ihre Features dienen folgenden Zwecken:

- Orchestrierung von Solution-Deployments zwischen Environments (Export/Import von Solutions),
- visuelles, klickbasiertes Erstellen und Ausführen von Release-Pipelines direkt im Power-Platform-UI (In-App Pipelines),
- Verwaltung von Pipeline-Runs, Verlauf, Approval-Schritten und einfachen Rollback-Optionen innerhalb des Tenants,
- Integration mit Dataverse-Berechtigungen und Governance (Pipeline-Zugriff, Audit-Logs),
- Bereitstellung einer niedrigschwelligen ALM-Option für Maker/Citizen Developer ohne externe CI/CD-Systeme.

Die App Power Platform Pipeline ist nicht primär für komplexe, Code-getriebene CI/CD-Szenarien (Source-Driven ALM, Git-Branching, erweiterte Automatisierung) vorgesehen; hierfür nutzt man pac, Build Tools (Azure DevOps) oder GitHub Actions.

Einbindung in eine etablierte CI/CD-Strategie

Die Integration von Low-Code-/No-Code-Artefakten in die unternehmensweite CI/CD-Strategie reduziert Aufwände und verbessert die Governance; daher sollte Low Code/No Code nicht in separaten, isolierten CI/CD-Umgebungen betrieben werden. YAML ist ein gängiges, portables Konfigurationsformat, das oft für Pipeline-Definitionen und strukturierte Test-

beschreibungen verwendet wird, allerdings können Testartefakte je nach Tool und Version auch als JSON vorliegen (Test Studio exportiert häufig JSON; die Test Engine akzeptiert JSON/YAML je nach Schema). Das Power Platform CLI (pac) und Build-Tools werden in CI/CD-Pipelines eingesetzt, arbeiten jedoch primär mit Solutions/CLI-Parametern; die Wahl zwischen YAML oder JSON ist projektspezifisch und sollte an das verwendete Tooling und Schema angepasst werden.

Zu den bekannten CI/CD-Tools, die YAML-Dateien verarbeiten, gehören Azure DevOps, Bitbucket Pipelines, CircleCI, GitHub Actions oder GitLab CI/CD. Aber auch ALM-Tools mit grafischer Benutzeroberfläche wie Bamboo, Jenkins oder TeamCity lassen sich durch Plug-ins nutzen. Ein möglichst hoher Automatisierungsgrad ist für effiziente DevOps-Prozesse wichtig. Um diesen zu erreichen, empfiehlt es sich, die Headless Execution der Test Engine, pac und gegebenenfalls Playwright (mit korrekter Browser-Installation sowie sicherem Secret/Auth-Handling) in den jeweiligen Scripts des CI/CD-Tools zu nutzen.

Microsoft unterstützt für Azure DevOps und GitHub Actions die Einbindung der Low-Code-/No-Code-Welt der Power Platform. Bei einer angemeldeten Azure DevOps Organization (**Bild 5**) erfolgt über den Marketplace von Visual Studio die Installation der Power Platform Build Tools [6]. Diese Extension liefert vorgefertigte DevOps-Tasks, um Power Platform/Dataverse-Deployments und -Builds automatisierbar und zuverlässig in CI/CD-Pipelines zu betreiben. Analoge Build/DevOps-Tasks stellt Microsoft für GitHub Actions bereit: Power Platform Actions. Für die Umsetzung kapseln beide DevOps-Umgebungen das Power Platform CLI (pac) oder implementieren äquivalente Funktionalität. Auf Kompatibilität der Versionen, Authentifizierung (Service Principal/Managed Identity/PAT) und sicheres Secret-Management in CI/CD ist zu achten. ■

[1] *GitHub-Repository von Microsoft Power Fx*,

<https://github.com/microsoft/Power-Fx>

[2] *Homepage der Entwicklungsumgebung von Power Apps*,

<https://make.powerapps.com/>

[3] *Homepage des Power Platform Admin Center*,

<https://admin.powerplatform.microsoft.com/>

[4] *Power Apps Test Engine auf GitHub*,

<https://github.com/microsoft/PowerApps-TestEngine>

[5] *GitHub-Repository mit GitHub Actions für die Power Platform*, <https://github.com/microsoft/powerplatform-actions>

[6] *Power Platform Build Tools auf GitHub*,

<https://github.com/microsoft/powerplatform-build-tools>



Frank Simon

unterstützt Projekte in den Arbeitsgebieten: Analyse, Spezifikation, Entwicklung, Programmierung, Test und Debugging in On-Prem-, Cloud-, Mobile- und Web-Umgebungen inklusive deren Systemmanagement.

dotnetpro@gmx.eu

DOTNET RUN "MYFILE"

Dateibasierte .NET-Apps



Ein .NET-Programm in einer Minute: file based programs.

Erinnert sich noch jemand daran, dass wir einst mit den Top Level Statements [1] eine Option für die *Program.cs*-Datei erhalten haben, die ohne Namespace oder *class Program* und *static void Main()* auskommt? Der Aufschrei war groß, dass C# nun einen „Skript-Modus“ bekommt und damit die Datei *Program.cs* so „eigenartig“ aussieht.

Das ist einige Jahre her, die Community hat sich längst beruhigt, und es ist inzwischen normal geworden, dass gerade neue Projekte genau so aussehen. Heute hinterfragt das niemand mehr, und ob eine *Program.cs* nun eine Klasse beherbergt oder nur den Code, ist den meisten Entwicklern mittlerweile egal.

Mit .NET 10 und C# 14 bringt Microsoft nun ein weiteres Feature, das in die Richtung „führe mal schnell ein paar C#-Zeilen aus“ weist: die file based programs [2].

File based program

Die Neuerung mit .NET 10: Anstelle einer ausgewachsenen Solution kann man mit einer beliebigen Datei *xyz.cs* mal eben ein Programm ausführen.

Bild 1 und Bild 2 zeigen Code und Aufruf eines einzeiligen, in Notepad++ formulierten Hello-World-Beispiels. Das als *Hello.cs* auf dem Desktop gespeicherte Skript lässt sich, sofern das .NET 10 SDK installiert ist, einfach auf der Konsole ausführen. Dabei wird das *dotnet*-CLI mit dem neuen Kommando *run* angesprochen und die Textdatei gibt den Text *Hallo dotnetpro* aus.

```

C:\Users\cgieswein\Desktop\Hello.cs - Notepad++
Datei Bearbeiten Suchen Ansicht Codierung Sprache Einstellungen Werkzeuge Makros Ausführen Plugins Fenster ? +
Hello.cs
1 Console.WriteLine("Hallo dotnetpro");
C# source length: 37 Zeilen: 1 Zeil: 1 Spalte: 15 Pos: 15 Windows (CR LF) UTF-8 IN

```

Hello-World-Einzeiler, verfasst in Notepad++ (Bild 1)

```

Git CMD
C:\Users\cgieswein\Desktop>dotnet run Hello.cs
Hallo dotnetpro
C:\Users\cgieswein\Desktop>

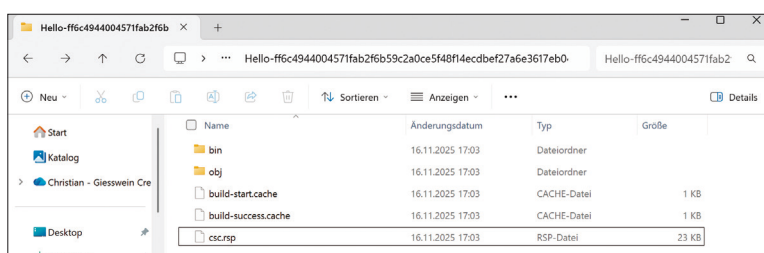
```

Mit *dotnet run* wird das Hello-World-Beispiel gestartet (Bild 2)

Die Technik

Um das neue Feature umzusetzen, hat Microsoft auf das existierende Toolset aufgebaut. Der erste Aufruf von *Hello.cs* dauert einen kurzen Moment, um bei jedem weiteren Aufruf dann blitzschnell zu starten. Beim ersten Start wird nicht nur der Compiler angeworfen, sondern das System merkt sich zudem den letzten Stand.

Gräbt man etwas tiefer, so wird man im dem aktuellen Benutzer zugeordneten *TEMP*-Ordner fündig. Dort gibt es den Unterordner *dotnet* und darin wiederum einen Ordner mit dem Namen *Hello-...*, eine Version des Dateinamens, erweitert



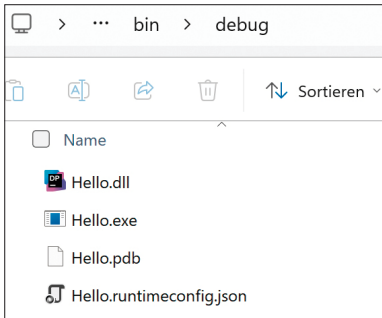
Der Build-Ordner zur Datei *Hello.cs* (Bild 3)

```

build-start.cache
C:\Users\cgieswein\Desktop\Hello.cs

```

build-start.cache enthält den kompletten Aufruf von *Hello.cs* (Bild 4)



Im **debug-Ordner** findet man die DLL- und die EXE-Datei zum Skript (Bild 5)

um einen Hashwert, welcher dem absoluten Pfad der Datei entspricht (Bild 3). Benennt man die Datei um, so dauert der erste Aufruf folgerichtig erneut etwas länger, weil sie wieder kompiliert werden muss.

Im temporären Ordner finden sich neben den üblichen Verzeichnissen *bin* und *obj* noch drei weitere Dateien, die den Zustand der App beschreiben (Bild 4). Wie erwartet liegt die Anwendungs-DLL zusammen mit der EXE-Datei im Unterordner *bin\debug* (Bild 5).

Hier wird ein simpler und zugleich genialer Trick verwendet. Zwar lässt sich über Sinnhaftigkeit und Anwendungszwecke streiten, aber jeder wird schon einmal eine kleine Konsolenanwendung gebastelt haben, mit ein paar C#-Zeilen, die eine wiederkehrende, lästige Aufgabe erledigt, ohne dass man zu PowerShell oder ähnlichen Tools greifen muss. Das geht jetzt flott und ohne Umweg über eine Entwicklungsumgebung.

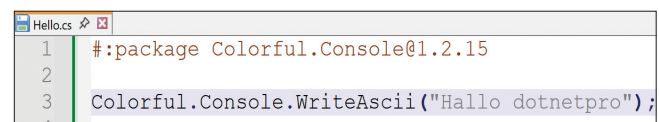
Geht es auch komplexer?

Man kann *dotnet run* freilich nicht nur für Einzeiler nutzen. Der Spaß fängt damit gerade erst an, denn Microsoft hat sich überlegt, dass man am Kopf der Datei mittels *#:*-Direktive eine noch tiefere Konfiguration vornehmen kann. Zum Beispiel

kann mittels *#:package* ein beliebiges NuGet-Paket hinzugefügt und verwendet werden. Im nächsten Beispiel ist es das Paket *Colorful.Console*. Startet man das mit dem NuGet-Paket erweiterte Projekt mit einer angepassten *WriteLine*-Zeile (Bild 6), so ergibt sich das wohl schönste Hello World in dieser Ausgabe (Bild 7).

Ausgabe mittels NuGet

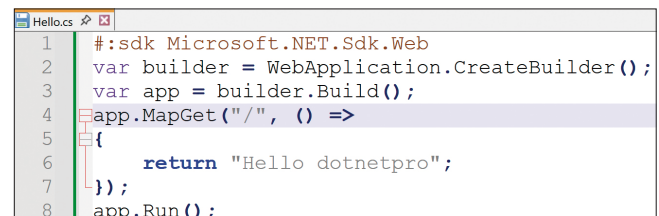
Die Krönung des Ganzen ist, wenn wir das Basis-SDK tauschen und stattdessen *Microsoft.NET.Sdk.Web* verwenden, um damit Zugriff auf ASP.NET Core zu erhalten (Bild 8). Wie Sie sehen, lässt sich mithilfe des ASP.NET Core Frameworks sofort ein Webserver starten, der auf die Route */* mittels *GET*-Verb reagiert und ein JSON-Paket zurückgibt (Bild 9).



NuGet-Pakete? Kein Problem! (Bild 6)

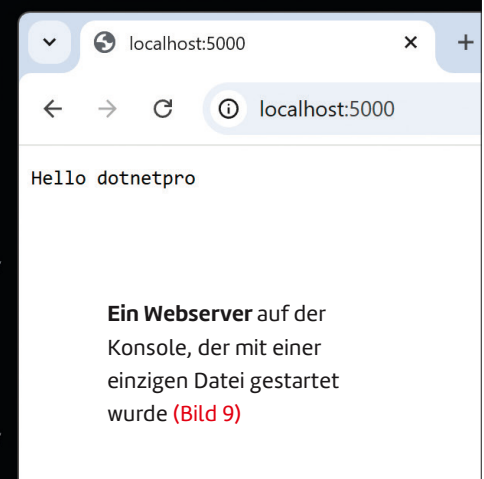


Colorful.Console.WriteAscii wandelt den übergebenen Namen in eine ASCII-Grafik (Bild 7)



Der wohl simpelste HTTP-Server (Bild 8)

```
C:\Users\cgießwein\Desktop>dotnet run Hello.cs
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\cgießwein\Desktop
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET http://localhost:5000/
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'HTTP: GET /'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'HTTP: GET /'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET http://localhost:5000/
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
```



Ein Webserver auf der Konsole, der mit einer einzigen Datei gestartet wurde (Bild 9)

```

MINGW64:/c/Users/cgiesswein/desktop
cgiesswein@CGIESSWEINMOB MINGW64 ~/desktop
$ chmod 777 Hello.cs

cgiesswein@CGIESSWEINMOB MINGW64 ~/desktop
$ ./Hello.cs
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\cgiesswein\Desktop

```

Ausführung von Hello.cs mit vorangestelltem ./ (Bild 10)

```

cgiesswein@CGIESSWEINMOB MINGW64 ~/desktop
$ cp Hello.cs HelloApp

cgiesswein@CGIESSWEINMOB MINGW64 ~/desktop
$ ./HelloApp
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\cgiesswein\desktop

```

Ausführen einer Code-Datei ohne die Endung .cs (Bild 11)

```

Hello.cs x
Hello.cs > Program > <top-level-statements-entry-point>
1  #!/usr/bin/env dotnet
2  #.sdk Microsoft.NET.Sdk.Web
3  var asd = 1 + 1;
4  var builder = WebApplication.CreateBuilder();
5  var app = builder.Build();
6  app.MapGet("/", () =>
7  {
8      return "Hello dotnetpro";
9  });
10 app.Run();

```

Das Debuggen klappt mit VS Code (Bild 12)

Bash-Support

Auf Systemen mit Bash Support kann man auch die Datei selbst als ausführbar deklarieren und spart sich somit das `dotnet run ...` (Bild 10). Gebraucht wird dafür folgende Direktive:

```
#!/usr/bin/env dotnet
```

Spätestens an dieser Stelle ist der Weg für alle möglichen kleinen Helferlein und Hilfsprogramme freigeräumt. Es ist

damit auch nicht mehr zwingend notwendig, dass eine Datei mit `.cs` endet, da über die Direktive der `dotnet`-Aufruf schon inkludiert ist. Ein Beispiel dafür sehen Sie in Bild 11.

Debuggen?

Wer ernsthafte Anwendungen mithilfe von file based programs bauen möchte, muss sich auch die Frage stellen, wie man diese Art von Anwendungen, die eigentlich für Schnellschüsse vorgesehen sind, seriös entwickeln und debuggen kann. Da hinkt das Ökosystem natürlich noch ein gutes Stück hinterher, doch mit Visual Studio Code funktioniert es bereits relativ einfach, wie Bild 12 zeigt.

Mit installiertem C# Dev Kit kann eine Code-Datei einfach mit [F5] gestartet und auch debuggt werden – cool. Die großen Entwicklungsumgebungen wie Visual Studio oder JetBrains Rider hinken da noch hinterher. Vermutlich ist es aber nur eine Frage der Zeit, bis sie die Lücke schließen.

Fazit

Mit den file based programs befähigt .NET 10 unsere kleinen C#-Dateien zu Größerem und ermöglicht das schnelle Ausführen von eben mal etwas Code. Ob das neue Feature nun überflüssig oder zwingend erforderlich ist, sei dahingestellt, aber es bietet eine weitere interessante und durchaus mächtige Möglichkeit in unserem Ökosystem.

Sicher ist, dass das die file based programs unsere Lieblingssprache um ein weiteres Einsatzgebiet bereichern. Die Community wird bestimmt nicht ruhen, sondern versuchen, weitere nützliche Szenarien für den Einsatz von file based programs zu finden. ■

- [1] Top Level Statements,
<https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements>
- [2] File based programs,
<https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/file-based-programs>



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 hat.
christian@software.tirol

DESKTOP-IOT-ANWENDUNGEN MIT .NET UND C#

Potzblitz in Hardware

Meadow.Blazor: direkter Hardware-Zugriff über eine Weboberfläche.

Dass die Meadow-Ausführungsumgebung [1] die Nutzung von .NET-Code zur Ansteuerung von IoT-Hardware ermöglicht, ist für regelmäßige Leser der dotnetpro nicht neu. Mit Meadow.Blazor steht nun eine neue Spielart der Technologie zur Verfügung, die den Fokus der Aufmerksamkeit auf die Unterstützung des gleichnamigen GUI- und Server-Frameworks legt.

Die einst als Weiterentwicklung der Netduino-Plattform entstandene Meadow-F7-Ausführungsumgebung hat sich seit einiger Zeit vom STM32-Mikrocontroller emanzipiert. Systeme wie Meadow.Desktop erlauben die allgemeine Interaktion mit Hardware von einem .NET-Gerät aus. Aus der Logik und Realität der nichtdeterministischen Ausführung auf Desktop oder Laptop folgt, dass sich aus dieser Situation sowohl Vorteile als auch Nachteile ergeben.

Dieser Artikel gibt einen Überblick der in Meadow.Blazor gebotenen Möglichkeiten, ordnet die Technologie ein und zeigt erste Beispiele.

USB-Seriell-Brücken als Schlüssel zur weiteren Verbreitung

Zum Verständnis von Architektur und Konzept hinter Meadow.Blazor sehen Sie sich bitte **Bild 1** an. Die dort schematisch gezeigte Struktur repräsentiert die Software, die in diversen Arduino-Boards jeden Tag millionenfach zum Einsatz kommt. Spätestens seit der Halbleiterkrise gilt, dass sich die Hardware eines Embedded-Systems mitunter schneller verändern kann, als dies dem Entwickler lieb ist. Systeme wie Arduino begegnen diesem Problem durch eine Hardwareabstraktionsschicht: Solange sich der Benutzercode nur mit den dort implementierten Interfaces beschäftigt, kann er bei Vorhandensein einer HAL-Implementierung auf mehr oder weniger beliebige Zielsysteme portiert werden.

In der Arduino-Welt wird dies durchaus beeindruckend gehandhabt: 8- und 32-Bit-Systeme entstehen oft aus nur minimal angepassten (aber separat zu testenden) Varianten von ein und demselben Sketch.

Im Fall der Meadow-Umgebung setzen Bryan Costanich, CEO und Mitgründer von Wilderness Labs, dem Unternehmen hinter der Meadow-Plattform, und sein Entwicklerteam seit einiger Zeit auf Chips aus dem Hause FTDI. Das dahinterstehende Schema ist immer dasselbe: Die Chips weisen auf

der einen Seite ein USB-Interface auf und exponieren auf der anderen Seite verschiedenste aus dem Embedded-Bereich bekannte Schnittstellen.

Bei Adaptierung der vom Meadow F7 bekannten HAL lassen sich die diversen Hardware-Interaktionsbefehle – zugegebenermaßen unter erheblicher Latenz – vom Desktop aus über den FTDI-Chip gegen Hardware, die sich in der realen Welt befindet, anwenden.

Im Fall der vorliegenden Variante kommt die in **Bild 2** gezeigte Platine zum Einsatz. Im Interesse der universellen Anwendbarkeit der von Wilderness Labs zur Verfügung gestellten Hinweise wird der Autor hier auf die von Adafruit entworfene Variante setzen – in der Praxis ist es allerdings genauso legitim, andere Evaluationsboards zu verwenden (**Bild 2**).

Wieso und warum

Bevor es darum geht, die Meadow.Blazor-Umgebung vorzubereiten, sind noch einige allgemeine Überlegungen angebracht, die auf die Vorteile der Nutzung eingehen.

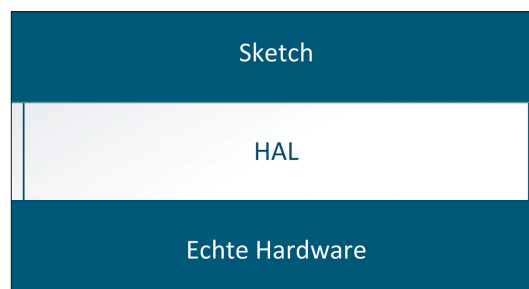
Dass das doppelte Vorhalten von Code, der im Worst Case noch in verschiedenen Sprachen geschrieben ist, zu einer massiven Steigerung der Softwarekopplung führt, kann an dieser Stelle als gegeben angenommen werden.

Das direkte Ausführen des Codes auf der Desktop- beziehungsweise MSR-Maschine (MSR: Messen, Steuern, Regeln) führt zu mehreren Vorteilen. Neben dem durch die Nutzung von .NET-Technologien wegfallenden algorithmischen Koppeln gilt, dass die Gesamtzuverlässigkeit eines Systems im Allgemeinen umso höher ausfällt, je weniger Komponenten im System vorhanden sind.

Sofern die anzusprechende Hardware in Bezug auf Latenz und Co. mit dem von Meadow.Blazor Gebotenen auskommt, führt die Nutzung der Technologie zu einer wesentlichen Reduktion der Gesamtkomplexität.

Anstatt einen Prozessrechner und einen Visualisierungssystem nebeneinander zu betreiben, kann das Visualisierungssystem direkt mit Sensoren interagieren.

Gegenüber der Nutzung eines TCP/IP-Netzes entsteht außerdem der Vorteil der einfacheren Verwendbarkeit. Möchte man in einem Industrie-System auf Ethernet setzen, so benötigt man einen zusätzlichen Router beziehungsweise Switch. Diese sind nicht gerade preisgünstig und steigern überdies die Komplexität des Gesamtsystems. ►



Arduino-Hardwareabstraktionsschicht (Bild 1)

Zu guter Letzt gilt, dass die Arbeit in der Blazor-Umgebung den meisten Desktop-Entwicklern zumindest peripher geläufig sein dürfte. Bequeme und schnell reagierende Debugger helfen bei der Fehlersuche – bis man eine ähnliche Toolchain für ein Embedded-System zusammengestellt hat, fließt viel Wasser die Donau hinunter.

Inbetriebnahme und Performanceanalyse

Nach diesen architekturellen Überlegungen ist es an der Zeit, die Meadow.Blazor-Umgebung in Betrieb zu nehmen. In den folgenden Schritten soll eine auf Windows 10 basierende Achtkern-Workstation zum Einsatz kommen – die verwendete Version von Visual Studio ist die 2022 Community Preview 17.14.16 P1.0.

Wichtig ist außerdem die Installation einer Version von .NET 8.0 – in den folgenden Schritten verwendet der Autor die unter [2] bereitstehende Version, die wie gewohnt auf der Host-Maschine zu installieren ist.

Vor dem Start der integrierten Entwicklungsumgebung (IDE) empfiehlt sich noch ein Abstecher in den Visual-Studio-Installer, wo es zu prüfen gilt, ob die Pakete *.NET Multi-platform App UI development* beziehungsweise *Mobile Development with .NET* vorhanden sind; gegebenenfalls sind diese nachzuinstallieren.

Darauf folgt ein Wechsel in den Visual-Studio-Werkzeugmanager, wo die Erweiterung *VS 2022 Tools for Meadow* [3] geladen wird. In den folgenden Schritten kommt die Version 2.0.0.14 zum Einsatz.

Problematisch ist, dass zum Zeitpunkt, als dieser Artikel geschrieben wurde, noch keine Visual-Studio-Projektvorlage gab, die sich um die Erzeugung eines für Meadow.Blazor geeigneten Projektskeletts kümmert.

Um dieses Problem zu umgehen, öffnen Sie die als Visual Studio 2022 Developer Command Prompt bezeichnete Entwickler-Eingabeaufforderung und wechseln danach in ein leeres Arbeitsverzeichnis. Der Autor hat den Pfad `C:\meadowblazor` verwendet.

Im nächsten Schritt vollziehen wir eine dreistufige Befehlsabfolge, die eine für unsere Bedürfnisse geeignete Projektstruktur aufbaut:

```
C:\meadowblazor>dotnet new blazorserver -n
MeadowBlazorSampleApp
...
```

Das System sollte daraufhin melden: *Die Vorlage „Blazor Server App“ wurde erfolgreich erstellt.*

```
...
C:\meadowblazor>cd MeadowBlazorSampleApp
C:\meadowblazor\MeadowBlazorSampleApp>dotnet add package
Meadow.Blazor
```

In manchen Fällen kommt es bei der Ausführung des obigen Befehls zur Fehlermeldung *Für das Paket „Meadow.Blazor“ sind keine Versionen verfügbar.* In diesem Fall ist es empfehlenswert, dem NuGet-Paketverwaltungssystem das Einbin-

den von Prototyp-Paketen zu erlauben. Das klappt mit `add package Meadow.Blazor --prerelease`.

Je nach Konfiguration der auf der Workstation vorhandenen .NET-Toolchain kann es zu weiteren Fehlermeldungen (Error NU1202) kommen. Diese deuten darauf hin, dass das .NET Framework des Projekts und die Projektversion von Meadow nicht zusammenpassen. Behoben wird das Problem durch Anpassen des Target-Frameworks in der Datei `MeadowBlazorSampleApp.csproj`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
```



Dieses Board agiert als Interface zwischen .NET-Code und Hardware der realen Welt (Bild 2)

Danach sollten die Fehlermeldungen ausbleiben.

Im nächsten Schritt bietet es sich an, Visual Studio zu starten und die `csproj`-Datei in die IDE zu laden. Darauf folgt das Hinzufügen einer neuen Klasse mit dem Namen `MeadowApplication.cs`, die anfangs den folgenden Code aufnimmt:

```
internal class MeadowApplication : App<Meadow.Desktop>
{
    public override Task Initialize() {
        FtdiExpanderCollection.Devices.Refresh();
        var ftdi = FtdiExpanderCollection.Devices[0];
        var output = ftdi.Pins.D7
            .CreatedDigitalOutputPort(false);
```



```

Resolver.Services.Add(output);
var led = new Led(ftdi.Pins.C0);
Resolver.Services.Add<ILed>(
    led);
return base.Initialize();
}

```

Auch die für Blazor vorgesehene Variante nutzt als Basisklasse *Meadow.Desktop*. Die Methode *Initialize* ist dann für die eigentliche Inbetriebnahme der Hardware verantwortlich. Über die Klasse *FtdiExpanderCollection* sucht sie zunächst nach allen mit dem Rechner verbundenen FTDI-Boards, um danach das erste auszuwählen und als GPIO-Ausgang zu konfigurieren (GPIO steht für General Purpose Input/Output).

In der *Run*-Methode findet sich Housekeeping-Code, der für das Bereitstellen von Rechenleistung verantwortlich ist:

```

...
public override Task Run() {
    return base.Run();
}
}

```

Vor dem Kompilieren wird IntelliSense das Fehlen der weiter oben genannten Klasse *FtdiExpanderCollection* monieren. Ursache dafür ist, dass die Meadow-Programmierungsumgebung intern komplett modular aufgebaut ist: Für jedes Hardware-Modul muss im Interesse der Einsparung von Ressourcen ein NuGet-Paket in die Solution wandern.

In der Praxis erweist sich das Finden der korrekten Pakete als eine der diffizilsten Aufgaben des Meadow-Entwicklers. Ein schöner Weg, dieses Problem zu meistern, ist die Eingabe des Namens in eine Internet-Suchmaschine. Geliefert werden sollte daraufhin ein URL nach dem Schema <https://developer.wildernesslabs.co/docs/api/Meadow.Foundation/Meadow.Foundation.ICs.IOExpanders/FtdiExpanderCollection/>. Aus diesem Link lässt sich dann der Name des NuGet-Pakets extrahieren. Für die vorliegenden Konfiguration lautet der korrekte String *Meadow.Foundation.ICs.IOExpanders.Ftxxxx*. Nach dem Einpflegen des NuGet-Pakets ist dieser Teil der Applikation abgeschlossen.

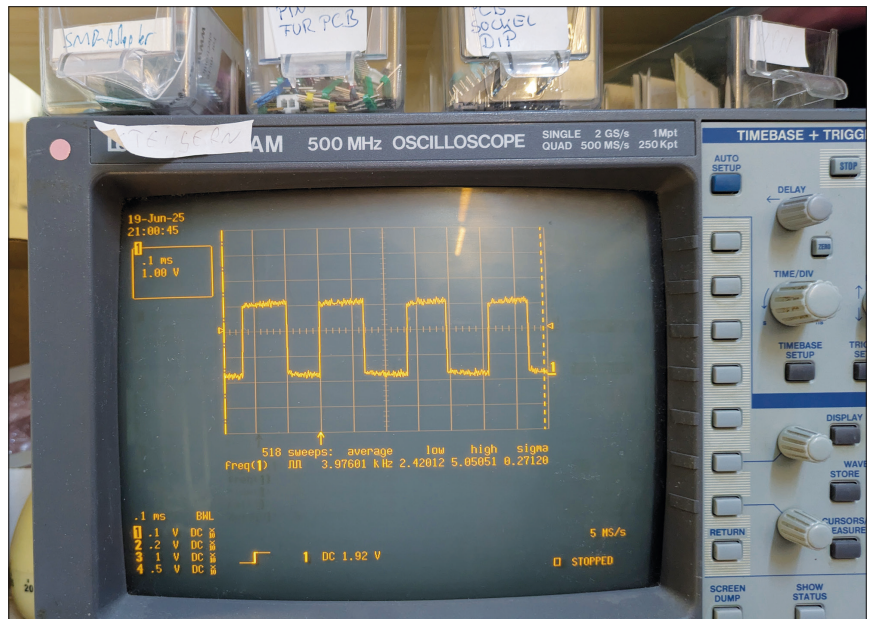
Jetzt wird die Datei *Program.cs* geöffnet, in der die Meadow-Applikation in den Rest der Blazor-Supportinfrastruktur eingebunden wird:

```

var app = builder.Build();
app.UseMeadow<MeadowApplication>();

```

An dieser Stelle bietet sich das probeweise Ausführen des Programms an. Auf frisch konfigurierten Workstations wird Visual Studio mitunter einige Meldungen einblenden, die auf Probleme mit lokalen Zertifikaten hinweisen – das Einpflegen



Meadow. Blazor arbeitet nicht sonderlich schnell (Bild 3)

der Debugging-Zertifikate in den Zertifikatspeicher der Workstation muss auf jeden Fall bejaht werden.

Bei noch nicht angeschlossenem Evaluationsboard startet das Programm danach durch, um mit einer Fehlermeldung nach dem Schema *System.ArgumentOutOfRangeException*: „Index was out of range. Must be non-negative and less than the size of the collection. (Parameter 'index')“ zu enden. Dies ist insofern logisch, als die Collection mit den diversen FTDI-Geräten derzeit ja keine Member aufweist.

Inbetriebnahme der Hardware

Im nächsten Schritt bietet sich an, die Adafruit-Evaluationsplatine per USB-C-Kabel mit dem Rechner zu verbinden. Nach dem Anstecken leuchtet die grüne LED auf, die anzeigt, dass die Stromversorgung vorhanden ist.

Die nächste Aktion ist das Modifizieren der *Run*-Methode, die für den eigentlichen MSR-Code zuständig ist. Für einen ersten Versuch werden wir ein Blinkprogramm nach folgendem Schema realisieren:

```

public override async Task Run() {
    while (true) {
        myLed.IsOn = true;
        await Task.Delay(1000);
        myLed.IsOn = false;
        await Task.Delay(1000);
    }
}

```

Da weiter oben eine Leuchtdioden-Abstraktionsklasse mit dem Pin *C0* verbunden wurde, erfolgt das Einstellen des Ausgangspegels unter Nutzung des Attributs *isOn*.

Die Konsultation der unter [4] bereitstehenden Schaltbild-Unterlagen für das von Adafruit zur Verfügung gestellte Evaluationsboard zeigen, dass die Leuchtdiode mit den Pins ►

C8 und C9 verbunden ist. Diese sind für die Meadow-Ausführungsumgebung derzeit aber nicht ansprechbar. Aus diesem Grund empfiehlt sich das Anschließen einer Leuchtdiode nebst Vorwiderstand, um das langsame Blinken sehen zu können. Alternativ bietet sich die Visualisierung unter Nutzung eines Oszillographen mit Roll-Mode an.

Im nächsten Schritt soll die maximale Arbeitsgeschwindigkeit des Systems ermittelt werden. Dazu wird die Ausgabe-Methode nach folgendem Schema modifiziert, um fortan zwei Wellenformen auszugeben:

```
public override async Task Run() {
    while (true) {
        myLed.IsOn = true;
        myLed.IsOn = false;
        myLed.IsOn = true;
        myLed.IsOn = false;
    }
}
```

An einem angeschlossenen Oszilloskop präsentiert sich dann das in **Bild 3** gezeigte Ergebnis. Zur Ermittlung der Langfristedstabilität bietet sich ein Modulationsdomänenanalysator an, der das in **Bild 4** gezeigte Ergebnis liefert.

Integration in die Blazor-Infrastruktur

Die durchgeführten Experimente, unter anderem mit dem Resolver-Service, sind kein Selbstzweck. Die diversen in der Ausführungsumgebung enthaltenen Komponenten integrieren sich in Blazor, wodurch man auf recht einfache Weise Webseiten realisieren kann, die mit realer Hardware interagieren. Dies wird nun anhand eines kleinen GPIO-Analysators demonstriert.

Dafür werden zunächst die durchgeführten Änderungen zurückgenommen. Insbesondere die *Run*-Methode muss wieder auf die Basisversion zurückgesetzt werden.

Da wir uns mit einer Benutzerschnittstelle auseinandersetzen wollen, gilt es zunächst die Deklaration des GPIO-Pins anzupassen. Fortan kommt die Methode *CreateDigitalInputPort* zum Einsatz, um eine Kommunikations-Abstraktionsklasse zu schaffen, die einen GPIO-Eingangspin abbildet:

```
public override Task Initialize()
{
    ...
    var input = ftdi.Pins.D7;
    CreateDigitalInputPort();
    Resolver.Services.Add(input);
}
```

Damit wenden wir uns der neuen Klasse *GPIOModel* zu, die für das Exponieren der Hardware-Daten für den in Blazor vorgehaltenen Teil der Applikation verantwortlich ist. Im ersten Schritt sind hier globale Member-Variablen erforderlich:

```
public class GPIOModel : IDisposable {
    public string GPIOState { get; private set; } =
        "INIT";
    public event Action? StateChanged;
    IDigitalInputPort myInput;
    private System.Threading.Timer timer;
    ...
}
```

Die Nutzung von *System.Threading.Timer* ist erforderlich, weil Meadow in der Blazor-Variante der GPIO-Abstraktionsklasse derzeit keine Interrupt-Methoden zur Verfügung stellt. Die Reaktion auf Änderungen der am GPIO-Pin anliegenden Spannungspegel kann deshalb nur durch Polling erfolgen. Folgender Konstruktor realisiert diesen Teil des Programms:

```
public GPIOModel() {
    myInput = Resolver.Services.Get<
        IDigitalInputPort>() ??
        throw new Exception(
            "Digital IO not found");

    timer = new System.Threading
        .Timer(async _ => {
            if (myInput.State == true)
                GPIOState = "Hoch";
            else
                GPIOState = "Tief";
            StateChanged?.Invoke();
        }, null, 0, 1000);
}
```

Zunächst wird die Resolver-Funktion von Blazor angesprochen, um Zugriff auf die GPIO-Abstraktionsklasse zu erhalten. Die mit *new System.Threading.Timer* beginnende Anweisung richtet einen Polling-Teilnehmer ein, dessen Payload fortan einmal pro Sekunde aktualisiert wird. Der Aufruf von *StateChanged?.Invoke()* sorgt dafür, dass Änderungen am



Die Entstehung von drei Höckern zeigt Instabilität (Bild 4)

Zustand der Hardware automatisch für den Rest des Benutzerinterfaces sichtbar werden.

Insbesondere bei der Arbeit mit komplizierten Abstraktionsklassen ist es wichtig, das Modell nach der Nutzung sauber zu beenden. Dies erfolgt einerseits durch Implementieren des Interfaces *IDisposable* und andererseits durch den erforderlichen Aufräumcode in der Methode *Dispose*:

```
...
public void Dispose() { }
}
```

Da es hier jedoch lediglich um kleine Beispiele geht, soll dieser Teil des Codes nicht weiter betrachtet werden. Stattdessen geht es weiter mit der Datei *Index.razor*, die das weiter oben besprochene Modell lädt:

```
@page "/"
@Inject MeadowBlazorSampleApp.GPIOModel ViewModel

<PageTitle>Index</PageTitle>
<span>GPIO-Zustand:</span>
<span>@ViewModel.GPIOState</span>
```

In einem der *span*-Tags findet sich dann etwas Code, der die in der Eigenschaft *GPIOModel* enthaltenen String-Werte als Teil des DOM-Baums sichtbar macht. Der Code-behind der Blazor-Page enthält in der Regel Bekanntes. Wichtig ist lediglich der Housekeeping-Code, der die Webseite als an Änderungen im Code-behind-Modell interessiert markiert:

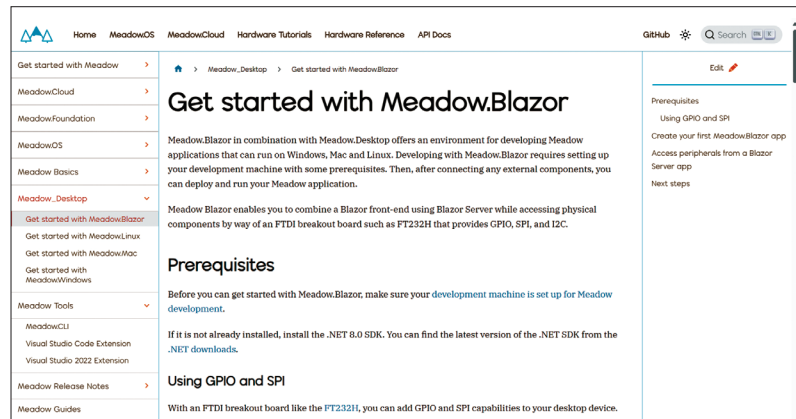
```
@code {
    protected override void OnInitialized()
    {
        ViewModel.StateChanged += OnViewModelStateChanged;
    }

    private void OnViewModelStateChanged()
    {
        InvokeAsync(StateHasChanged);
    }

    public void Dispose()
    {
        ViewModel.StateChanged -= OnViewModelStateChanged;
    }
}
```

Vor dem Ausführen der Webseite ist es noch erforderlich, zur Konfigurationsdatei *Program.cs* zu wechseln und das GPIO-Modell nach folgendem Schema zur Blazor-Applikationsstruktur hinzuzufügen:

```
// Add services to the container.
builder.Services.AddRazorPages();
```



Meadow.Blazor-Tutorials mit weiteren Beispielen (Bild 5)

```
...
builder.Services.AddSingleton<GPIOModel>();
```

Danach ist auch diese Version des Programms zur Ausführung bereit. Verbinden Sie probeweise den Pin D7 über einen Widerstand mit Masse und mit 3,3 V, um sich am – aufgrund des Pollings etwas verzögerten – Aufscheinen von Statusinformationen auf dem Bildschirm zu erfreuen.

Weitere Beispiele finden Sie auf den Tutorial-Seiten von Wilderness Labs unter [5], siehe Bild 5. Bei der Suche nach Hardware für ein Projekt hilft die Webseite Oemsecrets [6].

Fazit

Die effiziente Nutzung von Meadow.Blazor ermöglicht es Entwicklern, aus Blazor-Code heraus mit Hardware zu interagieren. Dies ist beispielsweise zur Realisierung von Kontrollpaneelen nützlich, mit denen sich Hardware in der realen Welt überwachen lässt.

- [1] Meadow, <https://developer.wildernesslabs.co/>
- [2] .NET 8, <https://builds.dotnet.microsoft.com/dotnet/Sdk/8.0.411/dotnet-sdk-8.0.411-win-x64.exe>
- [3] VS 2022 Tools for Meadow, <https://marketplace.visualstudio.com/items?itemName=WildernessLabs.vsmeadow2022>
- [4] Adafruit FT232H Schaltbild, <https://learn.adafruit.com/adafruit-ft232h-breakout/downloads>
- [5] Wilderness Labs, Get started with Meadow.Blazor, https://developer.wildernesslabs.co/Meadow/Meadow_Desktop/Meadow_Blazor/
- [6] Oemsecrets, www.oemsecrets.com/compare/2264



Tam Hanna

entwickelt Programme für verschiedene Plattformen, betreibt Online-Newsdienste zum Thema und steht für Fragen, Trainings und Vorträge gern zur Verfügung. Sie erreichen ihn unter der E-Mail-Adresse tamhan@tamoggemon.com.



DATA ACCESS

Die Datenzugriffsberatung mit DR. HOLGER SCHWICHTENBERG

SQL-SERVER-SPALTENTYP VECTOR MIT ENTITY FRAMEWORK CORE 10.0

Vektoren auf dem Vormarsch

Entity Framework Core 10.0 unterstützt auch den neuen Microsoft-SQL-Server-Datentyp Vector.

Die Datenzugriffskolumne in der vorangegangenen Ausgabe der dotnetpro [1] hat die Neuerungen in Entity Framework Core 10.0 in den Preview-Versionen 1 bis 6 vorgestellt. In der darauf folgenden Preview-Version 7 und vor allem auch in der Release-Candidate-Version 1.0 kamen weitere neue Features hinzu, obwohl bei Microsoft eigentlich die Vorgabe gilt, dass mit Preview 7 das Produkt „Feature Complete“ sein soll und die Release-Candidate-Versionen nur noch den Feinschliff liefern. Zu diesen neuen Features gehören die Unterstützung für die neuen Microsoft-SQL-Server-Spaltentypen Vector und JSON sowie Verbesserungen für komplexe Typen in Entity Framework Core (1-zu-N-Abbildung und JSON-Mapping).

Dieser Beitrag fokussiert auf den Spaltentyp Vector. Die anderen Neuerungen in Release Candidate 1 werden zusammen mit gegebenenfalls noch kommenden Neuerungen in der Re-

lease-Candidate-2-Version erst nach der finalen Veröffentlichung von Entity Framework Core 10.0 behandelt werden.

Microsoft SQL Server bietet in der Version 2025 (zum Redaktionsschluss für diesen Beitrag auf Stand Release Candidate 1 vom 16.09.2025) einen neuen Spaltentyp *vector* zum Speichern von Vektordaten für Ähnlichkeitssuche und KI-Anwendungen. In SQL Azure gibt es das Feature seit 2024 als Preview, und seit Juni 2025 gilt es als einsatzreif [2].

Vektoren besitzen eine Dimension als Ganzzahl (alias Elemente). Jedes Element im Vektor wird dabei als 2-Byte- oder 4-Byte-Gleitkommazahl gespeichert, wobei das 2-Byte-Feature getrennt zu aktivieren ist (siehe [3]). Vektoren werden im SQL Server in einem optimierten Binärformat abgelegt und im Datenbanktreiber (zum Beispiel *Microsoft.Data.SqlClient* ab Version 6.1) übertragen, aber in T-SQL und im SQL Server Management Studio (ab Version 21) als JSON-Arrays dargestellt. Ein Beispiel eines Vektors mit zehn Dimensionen:

```
DECLARE @v VECTOR(10) =
    '[0.1, 2, 3.5, 4, 5, 6, 7, 8, 9.9, 10]';
SELECT @v;
```

Ältere Datenbanktreiber übertragen die Vektordaten als *nvarchar(max)* und JSON-Format. Vektoren unterstützen keine Vergleiche, Additionen, Subtraktionen, Multiplikationen, Divisionen, Verkettungen oder andere mathematische, logische und zusammengesetzte Zuweisungsoperatoren. Vektorspalten können nicht in speicheroptimierten Tabellen verwendet werden.

SQL Server 2025 bietet für Vektoren die Funktion *VECTOR_DISTANCE()*, die den Abstand mit drei Verfahren berechnen kann (siehe Bild 1):

- *"cosine"*: 0 bis 2 (0 = identisch, 1 = orthogonal, 2 = entgegengesetzt)
- *"euclidean"*: 0 bis unendlich (0 = identisch)
- *"dot"*: -1 bis +1 bei normierten Vektoren, minus unendlich bis plus unendlich bei nicht normierten Vektoren

Vector1	Vector2	euclidean	cosine	negative_dot_product
[1.0000000e+001,2.0000000e+000,3.5000000e+000,4.0000000e+000,5.0000000e+000,6.0000000e+000,7.0000000e+000,8.0000000e+000,9.9000000e+000,1.0000000e+001]	[1.0000000e+001,2.0000000e+000,3.5000000e+000,4.0000000e+000,5.0000000e+000,6.0000000e+000,7.0000000e+000,8.0000000e+000,9.9000000e+000,1.0000000e+001]	0	0	-404.209999913672
Vector1	Vector2	euclidean	cosine	negative_dot_product
[1.0000000e+001,2.0000000e+000,3.5000000e+000,4.0000000e+000,5.0000000e+000,6.0000000e+000,7.0000000e+000,8.0000000e+000,9.9000000e+000,1.0000000e+001]	[1.0000000e+001,2.0999999e+000,3.5000000e+000,4.0000000e+000,5.0000000e+000,6.0000000e+000,7.0000000e+000,8.0000000e+000,9.9000000e+000,1.0000000e+001]	0.0999999946325684	1.22189521789551E-05	-404.489970703125
Vector1	Vector2	euclidean	cosine	negative_dot_product
[1.0000000e+001,2.0000000e+000,3.5000000e+000,4.0000000e+000,5.0000000e+000,6.0000000e+000,7.0000000e+000,8.0000000e+000,9.9000000e+000,1.0000000e+001]	[1.0000000e+001,2.0000000e+000,3.5000000e+000,4.0000000e+000,5.0000000e+000,6.0000000e+000,7.0000000e+000,8.0000000e+000,9.9000000e+000,1.0000000e+001]	102.615791320801	0.32042384147644	-1574.27001853125

Eine weitere Funktion *VECTOR_SEARCH()* zur Suche des nächsten Nachbarn gilt noch als Preview-Feature (Bild 1)



● Listing 1: Vektorspalten-Beispiel (Teil 1)

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.Data.SqlTypes;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;

namespace EFC_MappingScenarios.Vectors;

public class Texte
{
    public int ID { get; set; }
    public string Text { get; set; }
    [Column(TypeName = "vector(384)")]
    public Microsoft.Data.SqlTypes.SqlVector<float>
        Embedding { get; set; }
}

class Context : DbContext
{
    public DbSet<Texte> TexteSet { get; set; }
    bool logActive = false;

    void Log(string message)
    {
        if (logActive) CUI.PrintAlternatingColor(message);
    }

    protected override void OnConfiguring(
        DbContextOptionsBuilder builder)
    {
        builder.EnableSensitiveDataLogging(true)
            .EnableDetailedErrors();
        string connstring = Settings.ConnectionStringSQL2025
            + @$";Database=EFC_MappingScenarios_" +
            nameof(EFC_MappingScenarios.Vectors);
        builder.UseSqlServer(connstring,
            x => x.UseNetTopologySuite()
                .UseCompatibilityLevel(170)).LogTo(Log, new[]
                { RelationalEventId.CommandExecuted });
        // WICHTIG für JSON-Spalten: 170
    }
}

class Client
{
    {
        // https://www.bundesregierung.de/breg-de/bundes-
        // regierung/bundeskabinett?utm_source=chatgpt.com
        static OrderedDictionary<string, string>
            Bundesregierung2025 = new() {
                { "Friedrich Merz", "Bundeskanzler" }, { "Lars
                Klingbeil", "Bundesminister der Finanzen" },
                { "Alexander Dobrindt", "Bundesminister des
                Innern" }, { "Dr. Johann Wadepuhl", "Bundesminister
                des Auswärtigen" }, { "Boris Pistorius",
                "Bundesminister der Verteidigung" }, { "Katherina
                Reiche", "Bundesministerin für Wirtschaft und
                Energie" }, { "Dorothee Bär", "Bundesministerin
                für Forschung, Technologie und Raumfahrt" },
                { "Dr. Stefanie Hubig", "Bundesministerin der
                Justiz und für Verbraucherschutz" }, { "Karin
                Prien", "Bundesministerin für Bildung, Familie,
                Senioren, Frauen und Jugend" }, { "Bärbel Bas",
                "Bundesministerin für Arbeit und Soziales" },
                { "Dr. Karsten Wildberger", "Bundesminister für
                Digitales und Staatsmodernisierung" }, { "Patrick
                Schnieder", "Bundesminister für Verkehr" },
                { "Carsten Schneider", "Bundesminister für Umwelt,
                Klimaschutz, Naturschutz und nukleare Sicherheit"
                }, { "Nina Warken", "Bundesministerin für
                Gesundheit" }, { "Alois Rainer", "Bundesminister
                für Landwirtschaft, Ernährung und Heimat" },
                { "Reem Alabali-Radovan", "Bundesministerin für
                wirtschaftliche Zusammenarbeit und Entwicklung" },
                { "Verena Hubertz", "Bundesministerin für Wohnen,
                Stadtentwicklung und Bauwesen" }, { "Thorsten
                Frei", "Bundesminister für besondere Aufgaben /
                Chef des Bundeskanzleramtes" }
            };

        public static void Run()
        {
            CUI.Demo(nameof(EFC_MappingScenarios.Vectors));

            using (var ctx = new Context())
            {
                CUI.Yellow("Model: " +
                    ctx.Model.GetType().FullName);
            }
        }
    }
}
```

Vektoren in Entity Framework Core 10.0

Nutzerinnen und Nutzer von Entity Framework Core können die Vector-Spalten seit Version 10.0 Release Candidate 1 verwenden. Vector-Spalten im SQL Server werden sowohl beim Reverse Engineering als auch Forward Engineering auf die .NET-Struktur *Microsoft.Data.SqlTypes.SqlVector<float>* abgebildet. Diese Struktur stammt aus dem NuGet-Paket *Microsoft.Data.SqlClient* (ab Version 6.1).

Ebenso gibt es eine neue Entity-Framework-Core-Methode *EF.Functions.VectorDistance()* für die SQL-Funktion *VECTOR_DISTANCE()*, siehe [Listing 1](#). Bisher gab es für beides

ein eigenständiges NuGet-Paket *EFCore.SqlServer.VectorSearch* [4]. Allerdings lässt sich in Entity Framework Core 10.0 in LINQ-Abfragen noch nicht die SQL-Server-Funktion *VECTOR_SEARCH()* nutzen.

Beispiel: Suche ähnliche Texte

In dem folgenden Beispiel werden die Mitglieder des Bundeskabinetts (Stand 2025) in einer Datenbanktabelle mit Namen und Funktionen in einer Textspalte abgelegt. Anschließend werden mehrere Ähnlichkeitssuchen via *EF.Functions.VectorDistance()* ausgeführt.

● Listing 1: Vektorspalten-Beispiel (Teil 2)

```
Util.RecreateDB(ctx);

#region ----- Vektor-Daten anlegen
CUI.Head(ctx.Database.ProviderName +
    ": Erstelle Datensätze mit Vektor-Spalte\n" +
    "(Lokale Vektorisierung dauert etwas...)");
foreach (var person in Bundesregierung2025)
{
    CUI.BusyIndicator();
    var dt = new Texte();
    ctx.TexteSet.Add(dt);

    dt.Text = person.Key + " ist " + person.Value;
    var values = EmbeddingsUtil.Get("passage:" +
        dt.Text);
    dt.Embedding = new SqlVector<float>(values);

    Console.WriteLine(dt.Text + " -> " +
        dt.Embedding.Length);
}
var c2 = ctx.SaveChanges();
Console.WriteLine($"{c2} Datensätze gespeichert");
#endregion

#region ----- Vektor-Daten abfragen

CUI.Head(ctx.Database.ProviderName +
    ": Vektorsuche");

Suche("Dr. Karsten Wildberger");// exakte Suche
Suche("Karsten");// nur Vorname
Suche("Carsten Wilberger");
    // absichtlich falsch geschrieben, damit es
    // nicht exakt passt
Suche("Kriegsminister");
Suche("Kassenwart");
Suche("Unfug");
Suche("Der beliebteste Politiker im Kabinett");
Suche("Der unbeliebteste Politiker im Kabinett");
#endregion

CUI.PanelGreen("=== DONE");
}

// https://learn.microsoft.com/en-us/sql/t-sql/
// functions/vector-distance-transact-sql?
// view=sql-server-ver17
// cosine - Cosine distance 0 bis 2 (0 = identisch,
// 1 = orthogonal, 2 = entgegengesetzt)
// euclidean - Euclidean distance 0 bis unendlich
// (0 = identisch)
// dot - (Negative)Dot product (-1 bis +1 bei
// normierten Vektoren, -unendlich bis +unendlich
// bei nicht normierten Vektoren)
private static void Suche(string suchBegriff,
    string distanzVerfahren = "cosine")
{
    using var ctx = new Context();

    CUI.Yellow("Suche ähnlichste Einträge zu: " +
        suchBegriff);
    var sqlVector = new SqlVector<float>(
        EmbeddingsUtil.Get(suchBegriff));
    var result = ctx.TexteSet
        .OrderBy(b => EF.Functions.VectorDistance(
            distanzVerfahren, b.Embedding, sqlVector))
        .Take(5)
        .Select(b => new { Object = b, Distance =
            EF.Functions.VectorDistance(
                distanzVerfahren, b.Embedding, sqlVector) })
        .ToList();

    foreach (var b in result)
    {
        CUI.Green(b.Object.ID + ": " + b.Object.Text +
            " -> Distanz: " + Math.Round(b.Distance, 2));
    }
}
```

```

Microsoft.EntityFrameworkCore.SqlServer: Vektorsuche
Suche ähnlichste Einträge zu: Dr. Karsten Wildberger
11: Dr. Karsten Wildberger ist Bundesminister für Digitales und Staatsmodernisierung -> Distanz: 0,11
13: Carsten Schneider ist Bundesminister für Umwelt, Klimaschutz, Naturschutz und nukleare Sicherheit -> Distanz: 0,19
14: Nina Warken ist Bundesministerin für Gesundheit -> Distanz: 0,19
4: Dr. Johann Wadephul ist Bundesminister des Auswärtigen -> Distanz: 0,19
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,2
Suche ähnlichste Einträge zu: Karsten
13: Carsten Schneider ist Bundesminister für Umwelt, Klimaschutz, Naturschutz und nukleare Sicherheit -> Distanz: 0,19
11: Dr. Karsten Wildberger ist Bundesminister für Digitales und Staatsmodernisierung -> Distanz: 0,2
14: Nina Warken ist Bundesministerin für Gesundheit -> Distanz: 0,21
10: Bärbel Bas ist Bundesministerin für Arbeit und Soziales -> Distanz: 0,21
18: Thorsten Frei ist Bundesminister für besondere Aufgaben / Chef des Bundeskanzleramtes -> Distanz: 0,22
Suche ähnlichste Einträge zu: Carsten Wilberger
11: Dr. Karsten Wildberger ist Bundesminister für Digitales und Staatsmodernisierung -> Distanz: 0,15
13: Carsten Schneider ist Bundesminister für Umwelt, Klimaschutz, Naturschutz und nukleare Sicherheit -> Distanz: 0,2
10: Bärbel Bas ist Bundesministerin für Arbeit und Soziales -> Distanz: 0,21
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,21
14: Nina Warken ist Bundesministerin für Gesundheit -> Distanz: 0,21
Suche ähnlichste Einträge zu: Kriegsminister
5: Boris Pistorius ist Bundesminister der Verteidigung -> Distanz: 0,13
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,14
4: Dr. Johann Wadephul ist Bundesminister des Auswärtigen -> Distanz: 0,14
10: Bärbel Bas ist Bundesministerin für Arbeit und Soziales -> Distanz: 0,15
14: Nina Warken ist Bundesministerin für Gesundheit -> Distanz: 0,15
Suche ähnlichste Einträge zu: Kassenwart
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,19
10: Bärbel Bas ist Bundesministerin für Arbeit und Soziales -> Distanz: 0,2
5: Boris Pistorius ist Bundesminister der Verteidigung -> Distanz: 0,21
14: Nina Warken ist Bundesministerin für Gesundheit -> Distanz: 0,21
18: Thorsten Frei ist Bundesminister für besondere Aufgaben / Chef des Bundeskanzleramtes -> Distanz: 0,21
Suche ähnlichste Einträge zu: Unfug
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,21
14: Nina Warken ist Bundesministerin für Gesundheit -> Distanz: 0,22
10: Bärbel Bas ist Bundesministerin für Arbeit und Soziales -> Distanz: 0,22
1: Friedrich Merz ist Bundeskanzler -> Distanz: 0,22
17: Verena Hubertz ist Bundesministerin für Wohnen, Stadtentwicklung und Bauwesen -> Distanz: 0,23
Suche ähnlichste Einträge zu: Der beliebteste Politiker im Kabinett
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,15
12: Patrick Schnieder ist Bundesminister für Verkehr -> Distanz: 0,17
4: Dr. Johann Wadephul ist Bundesminister des Auswärtigen -> Distanz: 0,17
5: Boris Pistorius ist Bundesminister der Verteidigung -> Distanz: 0,17
1: Friedrich Merz ist Bundeskanzler -> Distanz: 0,17
Suche ähnlichste Einträge zu: Der unbeliebteste Politiker im Kabinett
2: Lars Klingbeil ist Bundesminister der Finanzen -> Distanz: 0,16
12: Patrick Schnieder ist Bundesminister für Verkehr -> Distanz: 0,18
1: Friedrich Merz ist Bundeskanzler -> Distanz: 0,18
4: Dr. Johann Wadephul ist Bundesminister des Auswärtigen -> Distanz: 0,18
3: Alexander Dobrindt ist Bundesminister des Innern -> Distanz: 0,19

```

Ausgabe von Listing 1 (Bild 2)

Hier im Heft nicht abgedruckt ist die Implementierung der Embeddings für die Texte (Vektorisierung) via *Embeddings-Util.Get()*, da der Programmcode zu umfangreich ist. Kontaktieren Sie gerne den Autor, wenn Sie Fragen zur Erstellung von Embeddings haben.

Die Ausgaben in Bild 2 basieren auf der Verwendung des Embedding-Modells Multilingual-E5-small mit 384 Dimensionen [5]. Das Modell wurde von Microsoft-Mitarbeitenden erschaffen, ist Open Source und kann lokal ohne Cloud arbeiten. Das Modell wurde so entworfen, dass es Texte in vielen Sprachen semantisch in denselben Vektorraum überführt, sodass Ähnlichkeitsvergleiche auch über Sprachgrenzen hinweg möglich sind [6]. Zu „Kassenwart“ und „Kriegsminister“ werden die entsprechenden Minister für Finanzen und Verteidigung gefunden. Welche Person die aktuell beliebteste oder unbeliebteste im Kabinett ist, kann das Embeddings-Modell nicht wissen. ■

[1] Holger Schwichtenberg, *Wenig Neues*, dotnetpro 10-11/2025, Seite 54 ff.

[2] Pooja Kamath, Davide Mauri, *Announcing General Availability of Native Vector Type & Functions in Azure SQL*,

<https://devblogs.microsoft.com/azure-sql/announcing-general-availability-of-native-vector-type-functions-in-azure-sql/>

[3] Microsoft Learn, Vektordatentyp, <https://learn.microsoft.com/de-de/sql/t-sql/data-types/vector-data-type>

[4] NuGet-Paket *EFCore.SqlServer.VectorSearch*, <https://github.com/efcore/EFCore.SqlServer.VectorSearch>

[5] Hugging Face, *Multilingual-E5-small*, <https://huggingface.co/intfloat/multilingual-e5-small>

[6] Liang Wang, Nan Yang et al., *Multilingual E5 Text Embeddings: A Technical Report*, <https://arxiv.org/pdf/2402.05672>

Dr. Holger Schwichtenberg

gehört zu den bekanntesten .NET- und Webexperten in Deutschland. Von Microsoft ist er seit 22 Jahren als MVP ausgezeichnet. Mit dem 54-köpfigen Expertenteam bei www.IT-Visions.de bietet er Beratungen, Schulungen und Softwareentwicklung an.

www.dotnet-doktor.de

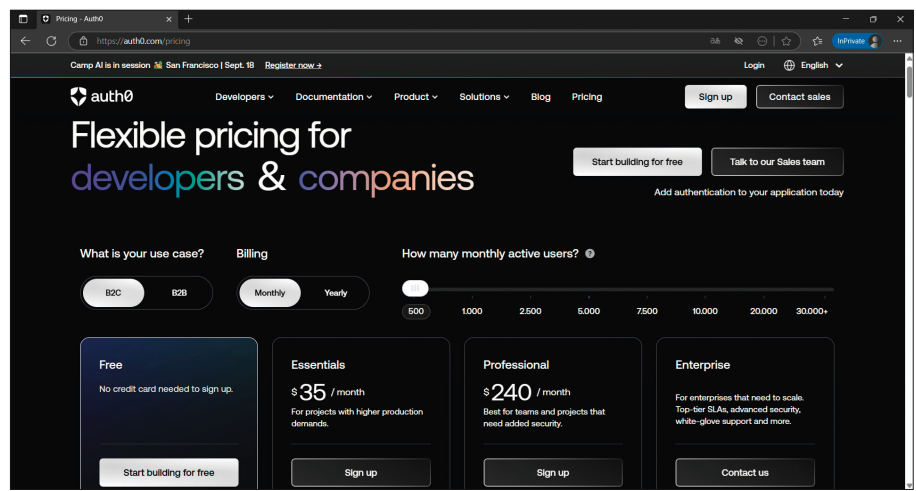
BLAZOR UND AUTH0

Authentifizierung in Nullkommanix

Blazor-Apps an einen Auth0-Server anbinden.

Manchmal denke ich wehmütig an die „alten Zeiten“ zurück – für mich bedeutet das: die späten 1990er-Jahre. Authentifizierung und Autorisierung für Webanwendungen waren damals noch trivial: Login-Maske, Überprüfung der Zugangsdaten (das Passwort stand bequem im Klartext in der Datenbank) und dann per Session-Management den Login-Status persistieren. Fertig war die gesicherte Webanwendung! Gut, die Implementierungen von damals waren meist nicht wirklich sicher, und Browser-Features zum Schutz von Session-Diebstahl (etwa: *Http-Only*-Flag für Cookies) noch nicht erfunden, aber immerhin: Implementierungstechnisch waren wir schnell und einfach am Ziel!

Spulen wir vor ins Jahr 2025: Authentifizierung ist weiterhin ein gelöstes Problem, die Simplizität des Ansatzes von vor knapp 30 Jahren ist aber gewichen. Stattdessen setzen moderne Webanwendungen auf moderne Standards, um Login und Logout anbieten zu können. OAuth und OpenID Connect (OIDC) sind im Einsatz, als Standards aber durchaus komplex, sodass die eigentliche Implementierung sehr stark auf

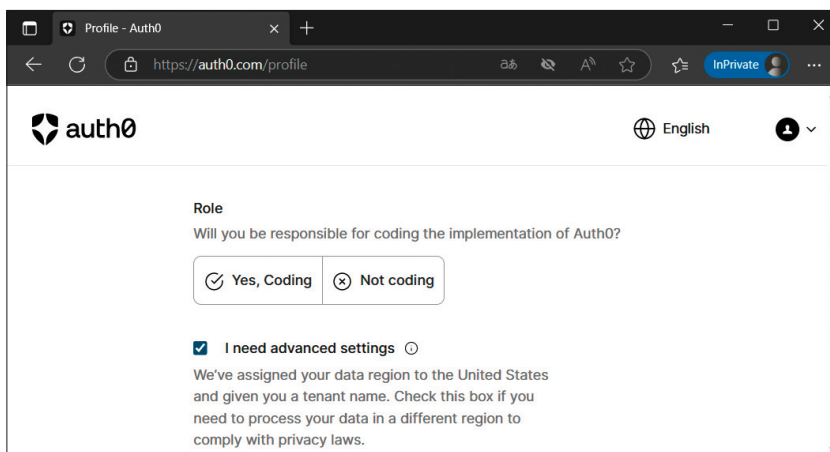


Preissensibel: Unterschiedliche Pakete, unterschiedliche Kosten bei Auth0 (Bild 1)

Bibliotheken beruht, die das übernehmen. Die Webanwendung selbst ist oft auch nicht mehr nach dem Muster „Client klickt, Server liefert neues HTML-Dokument“ angelegt, sondern in Form einer Single-Page Application (SPA) umgesetzt: JavaScript-Code steuert das Frontend, der Server ist zum Datenverarbeiter und -lieferant „degradiert“.

Im .NET-Stack haben wir für all dies bereits mindestens eine gute Option. SPA-Anwendungen können mit Blazor erstellt werden, das zwar seine Nachteile hat, aber einen vergleichsweise niedrigschwelligen Weg hin zur JavaScript-App bietet.

Support für OAuth und OpenID Connect ist bei .NET für Client-Anwendungen ebenfalls mit dabei. Gerade bei SPAs ist das eine gute Idee, denn der notwendige Prozess, ein Token sicher zum Client zu bringen und dann dort zu speichern, ist nicht unbedingt trivial. Einer von vielen Punkten, die es zu beachten gilt: Die SPA wird von JavaScript gesteuert, eine Übergabe des Tokens ist also im Wesentlichen per URL möglich. So ein URL landet gegebenenfalls in der Browser-History, seine Kenntnis alleine darf also keinen Zugang zum System gewähren. Es sind zusätzliche Sicherheitsmechanismen vonnöten. Diese sind Teil diverser OAuth-



Fortschrittlich: Unter den erweiterten Einstellungen verbirgt sich unter anderem der Serverstandort (Bild 2)

Flows und eben in den Client-Bibliotheken bereits passgenau implementiert.

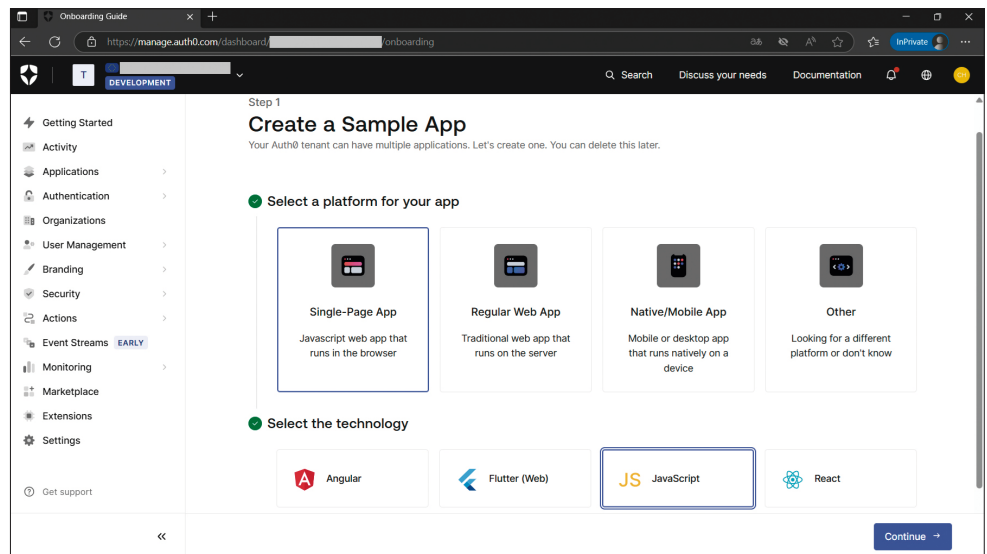
Die einzige Komponente, die manchmal fehlt, ist der Identity Provider (IdP), also im Wesentlichen das System, das die „Identitäten“ (vereinfacht: Nutzer) verwaltet, die Entscheidung trifft, ob gesendete Credentials valide sind, und das die verschiedenen Aspekte von OAuth implementiert, inklusive Tokenausstellung und die Datenflüsse (Flows), über die der Client an das Token rankommt. De-facto-Standard ist hier die kommerzielle Software Identity Server, die wir in der dotnetpro bereits behandelt haben [1]. Davon gibt es unter bestimmten Voraussetzungen auch eine kostenfreie Community-Lizenz. Beim Identity Server handelt es sich um einen Werkzeugkasten, für die Verwendung ist also noch einiges an Code notwendig. Im Gegenzug hat das Entwicklungsteam volle Kontrolle über das, was passiert.

Als Gegenentwurf gibt es eine Reihe von SaaS-Anbietern, die einen öffentlich gehosteten Identity Provider betreiben, über den dann die Authentifizierung durchgeführt werden kann. Einer der bekanntesten Anbieter ist Auth0. Dieser wurde 2013 gegründet und acht Jahre später von Okta aufgekauft. Selten für solche Übernahmen: Der Brand-Name Auth0 ist dabei beibehalten worden, auch wenn dieser inzwischen unter der Okta-Fahne segelt.

Dieser Artikel zeigt, wie eine Blazor-Anwendung an Auth0 angeschlossen werden kann. Auth0 ist kostenpflichtig, es gibt aber auch ein kostenfreies Preisband (Bild 1). Unter [2] steht eine aktuelle Übersicht zur Verfügung. Für die folgenden Ausführungen genügt tatsächlich die Gratis-Variante.

Auth0 einrichten

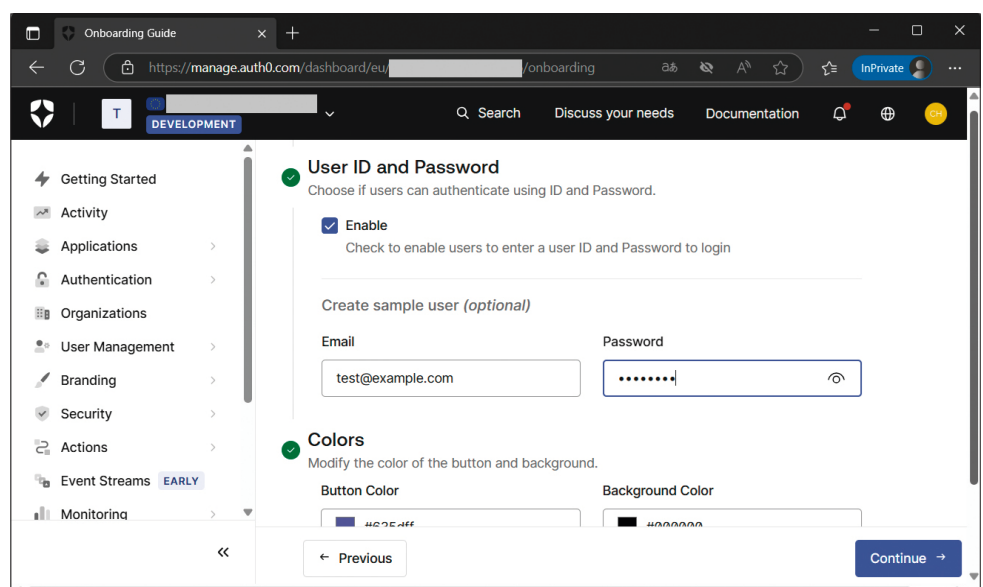
Bevor wir uns überhaupt der Blazor-Anwendung widmen, benötigen wir ein Konto bei Auth0. Wie bei SaaS-Angeboten irgendwie in der Natur der Sache liegend, können sich der Workflow und einzelne Details jederzeit ändern, aber das Gros der Schritte sollte gleich bleiben. Das Onboarding bei Auth0 ist stringent gestaltet und ermöglicht einen einfachen Einstieg, auch für



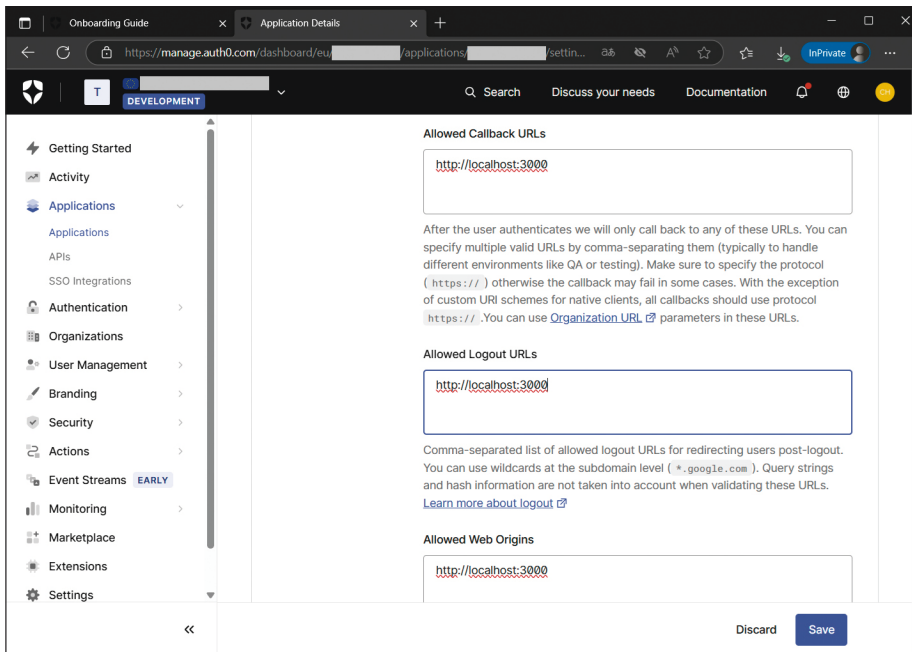
SPA(rtanisch): Rudimentäre Angaben über die Webanwendungen genügen zur Einrichtung (Bild 3)

unsere Zwecke. Ganz am Anfang gibt es jedoch eine unauffällige, aber möglicherweise entscheidende Frage: Wollen wir „fortgeschrittene Einstellungen“ (*advanced settings*) tätigen oder nicht? Die Checkbox in Bild 2 anzukreuzen ist eine gute Idee, denn dann können wir den Serverstandort einstellen und beispielsweise dafür die EU auswählen. Es ist ja eine andauernde Diskussion, inwieweit das für DSGVO-Konformität ausreichend ist (Merkregel: So lange, bis Maximilian Schrems wieder vor dem EuGH siegt), aber prinzipiell ist ein Server „in der Nähe“ so oder so eine gute Idee.

Ein Konto bei Auth0 anzulegen ist das eine, aber entscheidend ist ja, dass wir eine Gegenstelle für die Authentifizierung unserer App haben. Dazu müssen wir im Rahmen des Onboardings eine App anlegen (Bild 3), der App-Typ *Single-Page App* klingt für unsere Blazor-Anwendung nach der richtigen Wahl (wenngleich *Regular Web App* ebenfalls ►



Ready Player One: Testnutzer bereits beim Setup anlegen (Bild 4)



Gästeliste: Auth0 leitet nach Login und Logout nur auf bestimmte URLs weiter (Bild 5)

ausreichen würde). Und obwohl wir in vielen Blazor-Apps nur wenig JavaScript-Code selbst schreiben müssen, ist *JavaScript* die richtige Antwort bei der verwendeten Technologie, denn natürlich funktioniert der Blazor-Code nur mit Hilfscode auf Skriptbasis.

Im ersten Schritt möchten wir, dass ein Login mit einer Benutzername-Passwort-Kombination möglich sein soll. Der Wizard von Auth0 bietet das an, sodass wir gleich Testnutzer erzeugen können (Bild 4). Natürlich können wir später noch weitere neue Zugangsdaten anlegen, aber mit einem ersten Nutzer lässt sich die korrekte Einrichtung des Servers sofort überprüfen.

Das war es auch schon! Am Ende des Prozesses bietet Auth0 noch eine Testanwendung zum Download an, um die zuvor getätigten Einstellungen direkt verifizieren zu können. Es handelt sich dabei um eine simple JavaScript-basierte Anwendung, die Node.js beziehungsweise den Node Package Manager (*npm*) zum Ausführen benötigt. Zwar wollen wir Blazor einsetzen, aber es ist eine gute Idee, die vorbereitete App auszuprobieren, um eventuelle Fehler beim Setup direkt feststellen zu können. Dazu gehen wir zunächst in die Konfiguration von Auth0 und wählen dort die gerade angelegte Anwendung aus. In den App-Einstellungen werden unter anderem drei Werte abgefragt:

- **Allowed Callback URLs:** URLs, die aufgerufen werden dürfen, nachdem sich ein Nutzer bei Auth0 authentifiziert hat.
- **Allowed Logout URLs:** URLs, die aufgerufen werden dürfen, nachdem ein Nutzer

sich bei Auth0 abgemeldet beziehungsweise ausgeloggt hat.

- **Allowed Web Origins:** URLs, von denen aus per JavaScript HTTP-Anfragen an Auth0 geschickt werden dürfen. Dies ist bei SPAs notwendig – Blazor selbst benötigt diese Einstellung nicht, wohl aber die Testanwendung.

Alle diese drei Einstellungen dienen der Sicherheit (etwa: Vermeidung von bestimmten Phishing-Angriffen) und müssen explizit gesetzt werden. Die Demo-Anwendung verwendet den Browser-Port 3000 auf der lokalen Maschine, sodass zunächst *http://localhost:3000* als Wert für alle drei Einstellungen eingetragen werden muss (Bild 5). Später passen wir das für Blazor an.

Damit sind alle Vorbereitungen getätigt, und die von Auth0 gene-

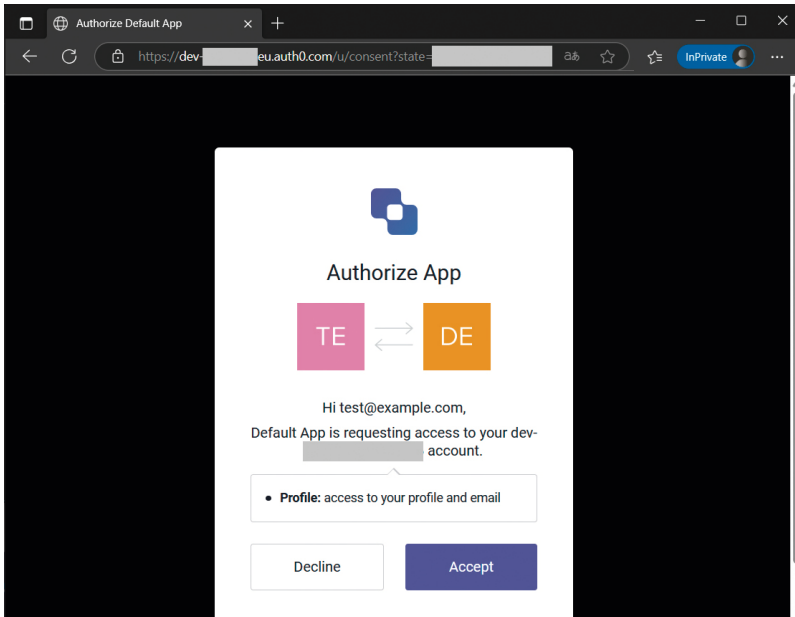
rierte Webanwendung kann gestartet werden. Der Befehl *npm install* installiert alle erforderlichen Node-Pakete, *npm start* startet die Testanwendung. Etwas unschön, dass beim ersten Kommando (zumindest zum Zeitpunkt der Artikelfertigstellung) vor Sicherheitslücken in einigen der verwendeten Pakete gewarnt wird (Bild 6), aber wir löschen die JavaScript-Anwendung ohnehin nach einem ersten Test.

Jetzt ist es fast geschafft! Beim Login in der Testanwendung auf *http://localhost:3000* erfolgt eine Weiterleitung zu Auth0, wo wir uns mit dem zuvor im Wizard angelegten Testkonto einloggen. Nach Freigabe der Informationen (Bild 7) erfolgt ein Rücksprung in die JavaScript-Anwendung, der technische Durchstich ist erfolgt.

```

D:\Projects\01-login>npm install
npm warn old lockfile
npm warn old lockfile The package-lock.json file was created with an old version of npm,
npm warn old lockfile so supplemental metadata must be fetched from the registry.
npm warn old lockfile This is a one-time fix-up, please be patient...
npm warn old lockfile
added 117 packages, and audited 118 packages in 5s
10 packages are looking for funding
  run 'npm fund' for details
13 vulnerabilities (6 low, 7 high)
To address issues that do not require attention, run:
  npm audit fix
To address all issues (including breaking changes), run:
  npm audit fix --force
Run 'npm audit' for details.
D:\Projects\01-login>npm start
> 01-login@1.0.0 start
> node bin/www
Server running on port 3000
  
```

Alarm! Die Testanwendung setzt auf potenziell unsichere Dependencies (Bild 6)



Datenhandel: Auth0 gibt – nach Zustimmung – Informationen an die Test-anwendung weiter (Bild 7)

Blazor einrichten

Jetzt wird es Zeit für die Blazor-Anwendung. Der Beispieldcode zu diesem Artikel verwendet .NET 8, wurde aber auch mit .NET 9 und .NET 10 getestet. Für die folgenden Ausführungen ist .NET 8 die Mindestversion. Dort nämlich hat sich das Render-Modell von Blazor etwas geändert. Mussten wir uns zuvor lediglich zwischen Blazor Server (Blazor läuft auf dem Server, während der Browser per JavaScript und Websocket dauerhaft mit ihm verbunden ist) und Blazor Web Assembly (Blazor läuft komplett im Browser) entscheiden, so ist das ab .NET 8 feingranularer steuerbar. Wir können uns im Wesentlichen bei jeder Komponente entscheiden, ob diese auf dem Server oder eben via WebAssembly im Client läuft. Der Render-Modus *Auto* verkompliziert die Dinge weiter:

Hier wird zunächst Blazor auf dem Server ausgeführt (weil schnell) und dann im laufenden Betrieb auf WebAssembly umgestellt (weil auch offline funktionsfähig). Für das Thema Authentifizierung und Autorisierung ist das durchaus herausfordernd, weil es ja sowohl client- als auch serverseitig funktionieren muss.

Gewusst wie, ist das jedoch schnell umgesetzt. Wir benötigen zunächst eine neue Blazor-Anwendung mit dem Auto-Render-Modus. Bild 8 zeigt die entsprechenden Einstellungen in Visual Studio; wer auf das .NET CLI setzt, verwendet ein Kommando nach folgender Machart:

```
dotnet new blazor --interactivity Auto
```

Wir geben also explizit keinen Authentifizierungstyp an. Das könnten wir zwar,

aber wir wollen nicht das .NET-Template verwenden, sondern setzen stattdessen auf ein NuGet-Paket von Auth0 selbst, welches das weitere Vorgehen stark vereinfacht. Eine Suche nach „Auth0“ fördert diverse Bibliotheken zutage (Bild 9), aber diejenige, die wir benötigen, heißt *Auth0.AspNetCore.Authentication*. Ja, da ist kein „Blazor“ im Namen enthalten, aber streng genommen ist auch Blazor Teil von ASP.NET Core, selbst wenn es das Marketing aus Redmond nicht immer so betont. Auch über das .NET CLI lässt sich das Paket dem Projekt hinzufügen:

```
dotnet add package
Auth0.AspNetCore.Authentication
```

Apropos Projekt: Durch die Auswahl von Render-Modus *Auto* haben wir zwei Projekte erhalten: eines mit dem gewählten Namen (im Folgenden: *Auth0Blazor*, aber Sie können das natürlich nach eigenen Wünschen ändern), und eines mit Suffix für den Blazor-Client (bei

uns: *Auth0Blazor.Client*). Das NuGet-Paket ist für ersteres Projekt relevant und muss dort hinzugefügt werden! Auch in dem Projekt ohne „.Client“ müssen wir die Informationen über die in Auth0 angelegte Anwendung hinterlegen. Bild 10 zeigt, wie das in der dortigen Oberfläche aussieht. Die Werte von *Domain* und *Client ID* benötigen wir und fügen diese der *appsettings.json* hinzu (die Beispieldateien zu diesem Artikel verwenden selbstverständlich nur Platzhalter):

```
"Auth0": {
  "Domain":
    "<Ihre individuelle Domain>.eu.auth0.com",
  "ClientId": "<Ihre individuelle Client-ID>"
}
```

Weitere Informationen

Blazor-Web-App C# Linux macOS Windows Blazor Cloud Web

Framework
.NET 8.0 (Langfristiger Support)

Authentifizierungstyp
Keine

☒ Für HTTPS konfigurieren

Interaktivitätstyp
Automatisch (Server und WebAssembly)

Interaktivitätsstandort
Global

☒ Include Beispielseiten

☐ Keine Anweisungen der obersten Ebene verwenden

☐ In .NET Aspire-Orchestrierung eintragen

Aspire Version
9.4

Zurück Erstellen

Autotune: Automatischer Render-Modus für die Beispielanwendung (Bild 8)

Diese Werte lesen wir in der Datei *Program.cs* aus und konfigurieren damit das zuvor installierte NuGet-Paket, indem wir folgenden Aufruf einfügen:

```
builder.Services
    .AddAuth0WebAppAuthentication(options => {
        options.Domain =
            builder.Configuration["Auth0:Domain"]!;
        options.ClientId =
            builder.Configuration["Auth0:ClientId"]!;
    });
```

Auth0 feinjustieren

Da wir inzwischen den Hostnamen und vor allem den Port der Website kennen, müssen wir noch in der Anwendung innerhalb des Auth0-Systems die entsprechenden Weiterleitungs-URLs angeben – anfangs haben wir das für die JavaScript-Test-App gemacht (zur Erinnerung: <http://localhost:3000>). Werfen Sie einen Blick auf die Datei *Properties/launchSettings.json*. Der Inhalt ist bei Ihnen anders, weil die Portnummern zufällig zugewiesen werden. Übernehmen Sie entweder die folgenden Angaben, oder merken Sie sich die HTTPS-Ports Ihrer Anwendung, denn genau die benötigt gleich Auth0:

```
{
  "$schema":
    "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
```

```
"inspectUri": "{wsProtocol}://{url.hostname}:
  {url.port}/_framework/debug/ws-proxy?browser=
  {browserInspectUri}",
  "applicationUrl":
    "https://localhost:7042;http://localhost:5042",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

Unsere Anwendung läuft unter <https://localhost:7042> (den HTTP-Port ignorieren wir). Ersetzen Sie also gegebenenfalls im Folgenden 7042 mit dem HTTPS-Port Ihrer Anwendung. In den Anwendungs-Einstellungen bei Auth0 setzen Sie die Weiterleitungs-URLs wie folgt:

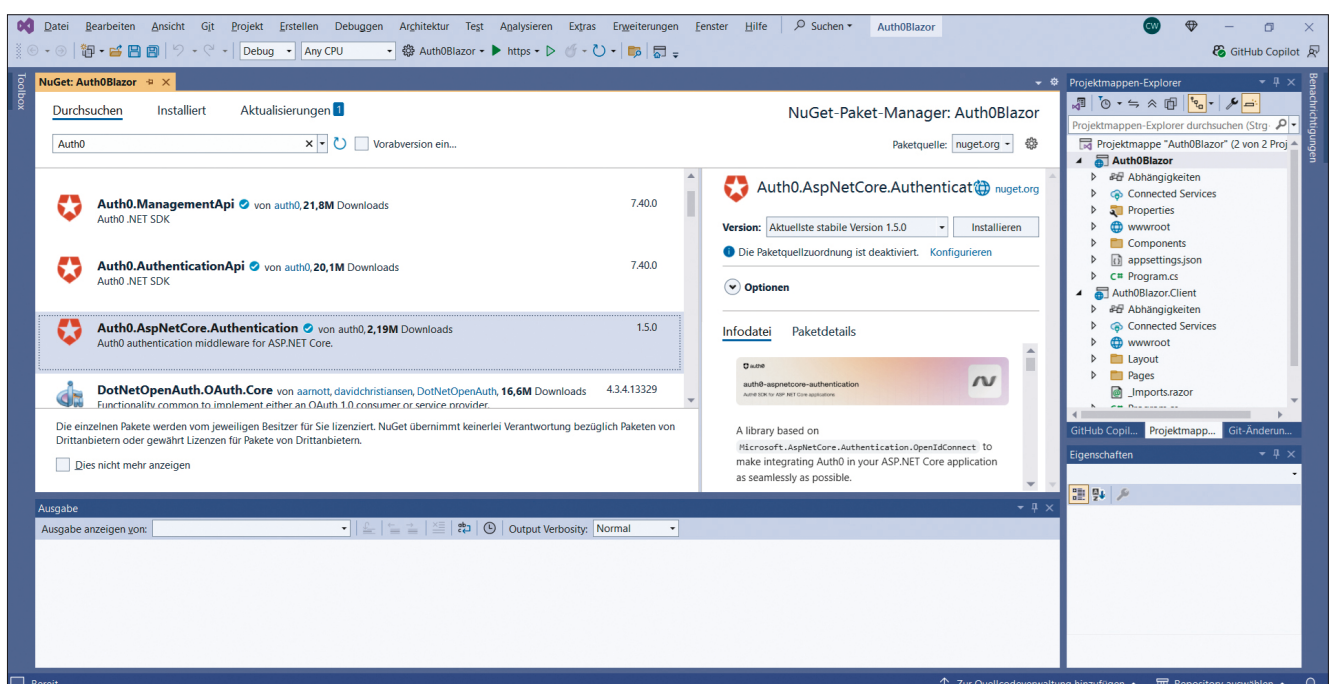
- Allowed Callback URLs: <https://localhost:7042/callback>
- Allowed Logout URLs: <https://localhost:7042>
- Allowed Web Origins: Keine Eingabe notwendig.

Blazor absichern

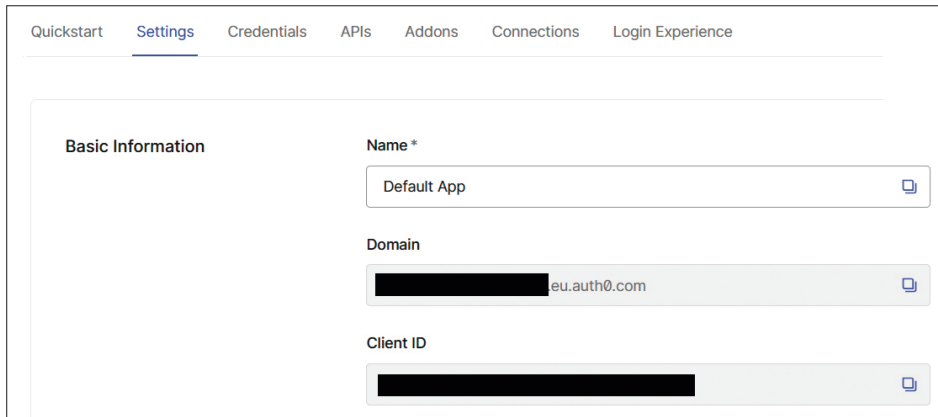
Zurück in die Blazor-Anwendung: Die initiale Konfiguration war der erste Schritt, jetzt geht es an den Code. Kommt das *[Authorize]*-Attribut zum Einsatz, kann damit eine Razor-View vor anonymem Zugriff geschützt werden. Hier ein simples Beispiel für eine solche Seite:

```
@page "/secret"
```

```
@attribute [Authorize]
<h3>Secret Area</h3>
```



Paketpost: Ein NuGet-Paket von Auth0 nimmt uns einen Großteil der Arbeit ab (Bild 9)



Universalschlüssel: Über Domain und Client ID verbinden wir die Blazor-App mit Auth0 (Bild 10)

Allerdings sind noch Vorarbeiten erforderlich, damit Blazor diese Komponente dann auch anzeigt. Im Client-Projekt (*Auth0Blazor.Client*) liegt die Datei *Routes.razor*, die unter anderem die folgende Zeile enthält:

```
<RouteView RouteData="routeData"
    DefaultLayout="typeof(Layout.MainLayout)" />
```

Das *<RouteView>*-Element muss durch folgendes Control ersetzt werden (es ändert sich nur der Bezeichner; die Parameter bleiben gleich):

```
<AuthorizeRouteView RouteData="@routeData"
    DefaultLayout="@typeof(Layout.MainLayout)" />
```

Die Komponente ist im NuGet-Paket *Microsoft.AspNetCore.Components.Authorization* definiert, das wir dementsprechend ins Client-Projekt installieren müssen. Verwenden Sie hierfür am besten dieselbe Versionsnummer wie von .NET. Damit die Komponente anschließend gefunden wird, muss in der Datei *_Imports.razor* das folgende *using*-Statement ergänzt werden:

```
@using Microsoft.AspNetCore
    .Components.Authorization
```

Jetzt fehlt nur noch ein letzter großer Schritt: Nutzer müssen sich ein- und ausloggen können. Dazu hinterlegen wir folgendes Markup im Layout im Client-Projekt – am besten in der Datei *MainLayout.razor*:

```
<AuthorizeView>
    <NotAuthorized>
        <a href="Account/Login">
            Login </a>
        </NotAuthorized>
    <Authorized>
```

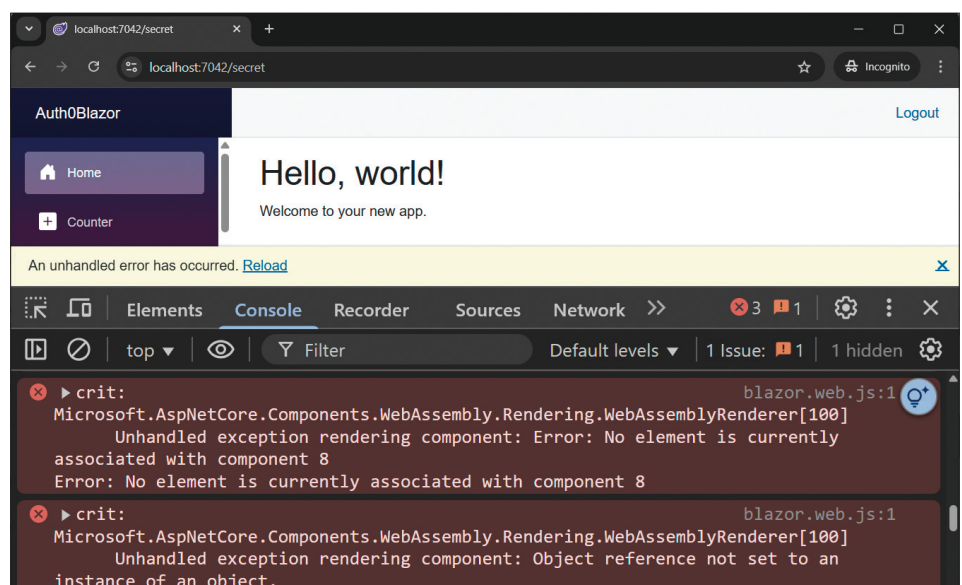
```
<a href="Account/Logout">
    Logout</a>
</Authorized>
</AuthorizeView>
```

Ausgeloggte Nutzer kommen also zum Endpunkt *Account/Login*, eingeloggte zu *Account/Logout*. Die URLs können beliebig angepasst werden, aber dieses Namensschema orientiert sich an dem von ASP.NET Core Identity. Allerdings haben wir das Identitäts-Framework von ASP.NET Core nicht im Einsatz, weswegen wir diese beiden URLs noch implementieren müssen. Der Code ist recht simpel – beim Login-Endpunkt wird angemeldet, beim Logout-Endpunkt wird abgemeldet, jeweils unter Zuhilfenahme des SDK von Auth0. Der folgende Code, angelehnt an die Dokumentation von Auth0 [3], sorgt für eine korrekte Implementierung:

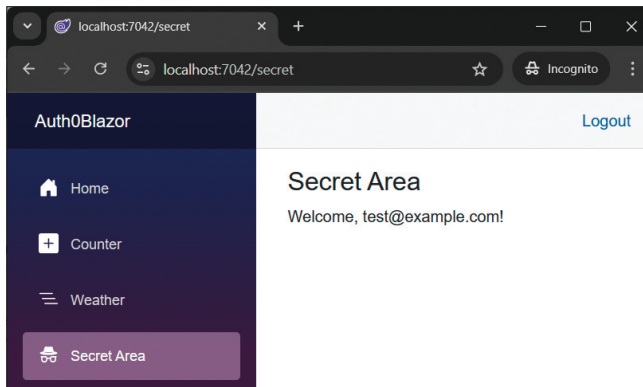
```
app.MapGet("/Account/Login", async (HttpContext
    httpContext, string returnUrl = "/") =>
{
    var authenticationProperties =
        new LoginAuthenticationPropertiesBuilder()
            .WithRedirectUri(returnUrl)
            .Build();

    await httpContext.ChallengeAsync(Auth0Constants
        .AuthenticationScheme, authenticationProperties);
});

app.MapGet("/Account/Logout",
    async (HttpContext httpContext) =>
```



Die Freude war von kurzer Dauer: (Noch) erscheint ein Fehler (Bild 11)



Sofort erkannt: Blazor kennt die eingeloggte E-Mail-Adresse (Bild 12)

```
{
    var authenticationProperties =
        new LogoutAuthenticationPropertiesBuilder()
            .WithRedirectUri("/")
            .Build();
    await httpContext.SignOutAsync(Auth0Constants
        .AuthenticationScheme, authenticationProperties);
    await httpContext.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);
});
```

Abschließend fügen wir noch einen Link auf die geheime Seite in das Navigationsmenü ein und können theoretisch starten. Ein direkter Aufruf von `/secret` sorgt zunächst nur für die Ansicht, dass die Seite nicht aufgerufen werden dürfe. Ein Klick auf *Login* führt zu *Auth0*, wo wir uns mit dem zuvor angelegten Credentials anmelden können. Von dort aus geht es direkt zurück in die Blazor-Anwendung, in der – nach kurzer Verzögerung – eine Fehlermeldung erscheint (Bild 11): Irgend-ein Problem mit WebAssembly und dem Rendering.

Die Verzögerung gibt möglicherweise den entscheidenden Tipp hin zur Lösung: Wir haben den Render-Modus *Auto* gewählt. Es geht also mit dem ehemaligen „Blazor Server“ los, und dann wird zu WebAssembly gewechselt. Dort kracht es. Wir benötigen im Client-Projekt noch ein weiteres NuGet-Paket: `Microsoft.AspNetCore.Components.WebAssembly.Authentication`. Sobald dieses installiert ist, funktioniert magischerweise alles, auch nach dem Switch zu WebAssembly.

Zum Abschluss greifen wir noch auf Informationen über den eingeloggten Nutzer zu, um nicht nur dokumentieren zu können, dass eine Authentifizierung erfolgt ist, sondern damit wir auch von *Auth0* erfahren, von wem. Hierzu benötigen wir im Wesentlichen den *AuthenticationState* als *Cascading Parameter*, über den wir an den Nutzer, dessen Identität und final auch an den „Namen“ (das ist in unserem Fall die E-Mail-Adresse) herankommen:

```
<p>Welcome, @Name!</p>
```

```
@code {
    [CascadingParameter]
```

```
private Task<AuthenticationState>?
    authenticationState { get; set; }
private string Name = string.Empty;

protected override async Task OnInitializedAsync()
{
    if (authenticationState is not null)
    {
        var state = await authenticationState;

        Name = state?.User?.Identity?.Name ??
            string.Empty;
    }
    await base.OnInitializedAsync();
}
```

In der Datei *Program.cs* – in beiden Projekten! (die Blazor-Anwendung kann ja Client- und Server-Komponenten im Einsatz haben) – brauchen wir abschließend noch die folgenden beiden Zeilen:

```
builder.Services.AddAuthorizationCore();
builder.Services.AddCascadingAuthenticationState();
```

Im Anschluss erscheint nach erfolgreichem Login auf der Seite `/secret` die verwendete E-Mail-Adresse (Bild 12).

Damit ist die Authentifizierung erledigt. Bei clientseitigen Komponenten gibt es eine weitere Hürde zu bewältigen: Wie bekommt der Code im Browser es mit, wenn sich der Authentifizierungszustand auf dem Server ändert (etwa ein Logout erfolgt)? Im Wesentlichen ist weiterer Boilerplate-Code vonnöten. Bei Bedarf hilft auch hier ein Blick in den Blog von *Auth0* [3] (Abschnitt „Synching the Authentication State“), in dem das durchexerziert wird.

Abschließend lässt sich konstatieren, dass die 1990er-Jahre doch gar nicht so übel waren. Aber auch in der Neuzeit (sprich: Blazor-Ära) ist es möglich, Authentifizierung und Autorisierung sicher und standardbasiert zu gestalten, auch wenn es ein wenig Code bedarf. ■

[1] Christian Wenz, *Who are you?*, *dotnetpro* 5/2022, Seite 52 ff.

[2] Kostenübersicht von *Auth0*, <https://auth0.com/pricing>

[3] *Auth0-Blogbeitrag zum Verbinden von Blazor mit dem System*, <https://auth0.com/blog/auth0-authentication-blazor-web-apps/>



Christian Wenz

ist Softwarearchitekt, Berater und Spezialist für die Themen Performance und Web 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



BenQ RD320U

Der Monitor für Programmierer, die weiter denken

Der BenQ RD320U ist speziell für Entwickler konzipiert – ein 31,5 Zoll (80 cm) großer 4K UHD-Monitor mit Nano-Matte-IPS-Panel, der gestochen scharfen Text und reflexionsfreies Arbeiten garantiert. Dank Coding-Modus (Dark & Light Theme) werden Kontrast, Schärfe und Gamma so optimiert, dass Codezeilen selbst nach Stunden mühelos lesbar bleiben.

Für lange Sessions sorgt das MoonHalo-Ambilight: eine sanfte Hintergrundbeleuchtung mit anpassbarer Farbtemperatur, kombiniert mit intelligenter Brightness Intelligence, die sich automatisch deiner Umgebung anpasst.

Ergänzt durch Eye-Care-Technologien wie Flicker-Free, Low Blue Light Plus, Night Hours Protection und ePaper-Modus schützt der RD320U deine Augen an langen Arbeitstagen oder selbst bei nächtlichen Coding-Marathons.

Auch der Workflow profitiert: USB-C mit 90 W Power Delivery, DisplayPort Daisy-Chain, KVM-Switch sowie PIP/PBP ermöglichen Multitasking ohne Umwege. Mit Display Pilot 2 steuerst du Modi, Layouts und Licht ganz einfach per Software.

Fazit: Der BenQ RD320U ist nicht nur ein Monitor, sondern ein intelligentes Arbeitswerkzeug. Entwickelt für Programmierer, die Wert auf Augenkomfort, Textklarheit und effizienten Workflow legen – und dabei Design und Technik auf höchstem Niveau erwarten.

KEY FEATURES

- 31,5 Zoll 4K+ (3840x2160), Nano-Matte-IPS
- Spezielle Coding-Modi (Dark/Light)
- MoonHalo-Ambilight & Auto-Dimmung
- Umfangreiche Eye-Care-Technologien
- USB-C 90 W, Daisy-Chain, KVM, PIP/PBP
- Steuerung via Display Pilot 2
- 2000 : 1 Kontrastverhältnis



Entdecke den neuen RD320U auf BenQ.com/de
BenQ Deutschland GmbH • Essener Straße 5 • 46047 Oberhausen



NEUERUNGEN IN BLAZOR 10.0, TEIL 2

Das Tempo bleibt ordentlich

Auch beim Monitoring, dem QuickGrid-Steuerelement und der C#-JavaScript-Interoperabilität bietet Blazor 10.0 einige Verbesserungen.

Im ersten Teil dieses Beitrags [1] wurden bereits die Neuerungen von Blazor 10.0 insbesondere in den Bereichen Zustandsverwaltung, Navigation, Validierung und Sicherheit vorgestellt. Hier geht es um die übrigen Neuigkeiten.

Metriken und Tracing

Auf der Agenda steht die Verbesserung der Überwachung von Blazor-Anwendungen bei Microsoft bereits seit ASP.NET Core in .NET 8.0, erschienen im November 2023. Dort hat ASP.NET Core diverse OpenTelemetry-Metriken und -Traces erhalten. Seit Preview 5 von .NET 10.0 gibt es nun spezielle Metriken für Blazor-Anwendungen.

Listing 1 zeigt, wie diese neuen Überwachungsfunktionen zunächst im Blazor-Startcode in der Datei *Program.cs* zu aktivieren sind. Danach stehen die in Tabelle 1 aufgeführten Metriken und die in Tabelle 2 genannten Traces zur Verfügung.

Die Metriken in *Microsoft.AspNetCore.Components.Server.Circuits* und die Traces in *Microsoft.AspNetCore.Components.CircuitStart* sind dabei nur in Blazor-Server-Anwendungen vorhanden, denn nur dort gibt es Circuits.

Bild 1 zeigt, wie auf Basis der neuen OpenTelemetry-Metriken in Blazor das Open-Source-Monitoring- und -Alerting-System Prometheus der Cloud Native Computing Foundation (CNCF) [2] die aktiven Circuits einer Blazor-Server-Anwen-

● **Tabelle 1: Metriken in Blazor 10.0**

Meter	Metrik	Beschreibung
Microsoft.AspNetCore.Components	<i>aspnetcore.components.navigation</i>	Zählt die Gesamtanzahl der Routenwechsel in einer Blazor-Anwendung.
Microsoft.AspNetCore.Components	<i>aspnetcore.components.event_handler</i>	Misst die Dauer der Verarbeitung von Browser-Ereignissen, einschließlich eigener Logik.
Microsoft.AspNetCore.Components.Lifecycle	<i>aspnetcore.components.update_parameters</i>	Misst die Dauer der Verarbeitung von Komponentenparametern, einschließlich eigener Logik.
Microsoft.AspNetCore.Components.Lifecycle	<i>aspnetcore.components.render_diff</i>	Misst die Dauer der Rendering-Batches.
Microsoft.AspNetCore.Components.Server.Circuits	<i>aspnetcore.components.circuit.active</i>	Zeigt die Anzahl der aktuell im Speicher befindlichen aktiven Circuits.
Microsoft.AspNetCore.Components.Server.Circuits	<i>aspnetcore.components.circuit.connected</i>	Zählt die Anzahl der mit Clients verbundenen Circuits.
Microsoft.AspNetCore.Components.Server.Circuits	<i>aspnetcore.components.circuit.duration</i>	Misst die Lebensdauer eines Circuits und zählt deren Gesamtanzahl.

● **Tabelle 2: Traces in Blazor 10.0**

Bereich	Tracing-Ereignis	Beschreibung	Tags	Verknüpfungen
Circuit-Lebenszyklus	Microsoft.AspNetCore.Components.CircuitStart	Protokolliert die Initialisierung eines Circuits im Format <i>Circuit {circuitId}</i>	<i>aspnetcore.components.circuit.id</i>	HTTP-Aktivität
Navigation	Microsoft.AspNetCore.Components.RouteChange	Verfolgt Routenwechsel im Format <i>Route {route} → {componentType}</i>	<i>aspnetcore.components.circuit.id</i> , <i>aspnetcore.components.type</i> , <i>aspnetcore.components.route</i>	HTTP-Trace, Circuit-Trace
Event-Verarbeitung	Microsoft.AspNetCore.Components.HandleEvent	Protokolliert die Ereignisverarbeitung im Format <i>Event {attributeName} → {componentType}.{methodName}</i>	<i>aspnetcore.components.circuit.id</i> , <i>aspnetcore.components.type</i> , <i>aspnetcore.components.method</i> , <i>aspnetcore.components.attribute.name</i> , <i>error.type</i>	HTTP-Trace, Circuit-Trace, Router-Trace

Listing 1: Aktivierung der Metriken und Traces für Blazor in der Startdatei einer Blazor-Server-Anwendung

```
using ITVisions.Blazor;
using NET10_BlazorServer.Components;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddBlazorUtilForBlazorServer();

#region Metriken und Traces
builder.Services.AddOpenTelemetry()
    .WithMetrics(metrics =>
    {
        metrics.AddAspNetCoreInstrumentation();
        metrics.AddRuntimeInstrumentation();
        metrics.AddMeter("Microsoft.AspNetCore.Components");
        // Neu in ASP.NET Core 10.0 ab Preview 5
        metrics.AddMeter(
            "Microsoft.AspNetCore.Components.Lifecycle");
        // Neu in ASP.NET Core 10.0 ab Preview 5
        metrics.AddMeter(
            "Microsoft.AspNetCore.Components.Server.Circuits");
        // Neu in ASP.NET Core 10.0 ab Preview 5
        metrics.AddConsoleExporter();
        metrics.AddPrometheusExporter();
        // OTLP für Prometheus
    })
    .WithTracing(tracing =>
    {
        tracing.AddAspNetCoreInstrumentation();
        tracing.AddHttpClientInstrumentation();
        tracing.AddSource(
            "Microsoft.AspNetCore.Components");
        // Neu in ASP.NET Core 10.0 ab Preview 5
        tracing.AddConsoleExporter();
        tracing.AddOtlpExporter();
        // Für Grafana Tempo, Jaeger etc.
    });
#endregion

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler(
        "/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseAntiforgery();

app.MapStaticAssets();
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
app.MapPrometheusScrapingEndpoint();
// Metriken bereitstellen unter
// https://localhost:7119/metrics

app.Run();
```

derung visualisiert. **Bild 2** zeigt die Darstellung von drei Blazor-Metriken aus Prometheus in einem Dashboard in der Datenvisualisierungsanwendung Grafana [3].

Diagnosedaten in Blazor WebAssembly

Eine größere neue Diagnose-Funktion in Blazor 10.0 bekam die WebAssembly-basierte Variante von Blazor, die als Sin-

gle-Page-App innerhalb des Webbrowsers läuft: Entwicklerinnen und Entwickler können damit im Webbrowser zur Laufzeit Diagnosedaten über Performance und Speicherinhalt sammeln.

Um die neuen Funktionen nutzen zu können, müssen Entwicklerinnen und Entwickler die .NET WebAssembly Build Tools (wasm-tools) zunächst in der aktuellen .NET-10.0- ►

Tabelle 3: Projekteinstellungen für Blazor-WebAssembly-Diagnosedaten

Eigenschaft	Standardwert	Setze auf ...	Beschreibung
<WasmProfilers>	Kein Wert	<i>browser</i>	Legt fest, welche Profiler verwendet werden sollen. Der Browser-Profiler ermöglicht die Integration mit den Entwicklerwerkzeugen des Browsers.
<WasmNativeStrip>	<i>true</i>	<i>false</i>	Deaktiviert das Entfernen (Stripping) der nativen ausführbaren Datei.
<WasmNativeDebugSymbols>	<i>true</i>	<i>true</i>	Aktiviert das Erstellen mit nativen Debug-Symbolen.

Quelle: Microsoft [8]

Listing 2: Einstellungen für die Timings-Ansicht im Blazor-WebAssembly-Client-Projekt

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
...
<!-- NEU: Blazor WebAssembly Runtime Diagnostics -->
<PropertyGroup>
  <!-- Für Anzeige von "Timings" im Performance-Tab
        der Browser-Developer-Tools-->
    <WasmProfilers>browsers</WasmProfilers>
    <WasmNativeStrip>>false</WasmNativeStrip>
    <WasmNativeDebugSymbols>>true
    </WasmNativeDebugSymbols>
</PropertyGroup>
</Project>
```

Version als .NET-SDK-Workload installieren:

```
dotnet workload install
wasm-tools
```

In den .NET WebAssembly Build Tools gibt es zwei neue Funktionen:

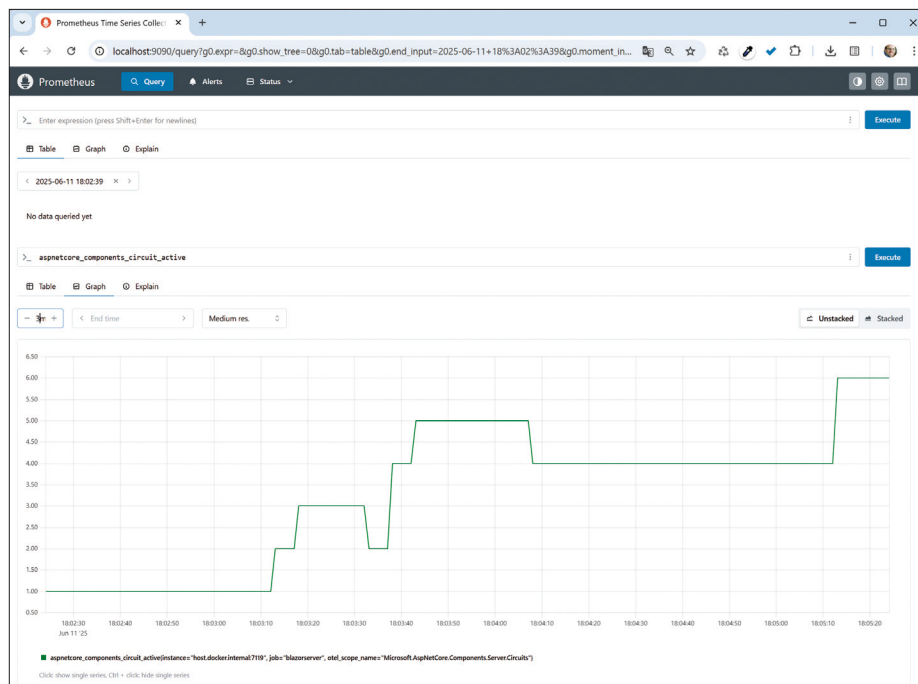
- Performance-Analyse in den Browser-Entwicklerwerkzeugen („F12-Tools“)
- Sammeln und Herunterladen von Trace-Dateien und Memory Dumps aus dem Browser

Für die Nutzung der Performance-Analyse in den Browser-Entwicklerwerkzeugen („F12-Tools“) stehen die in **Tabelle 3** gezeigten Einstellungen in der Projektdatei zur Verfügung.

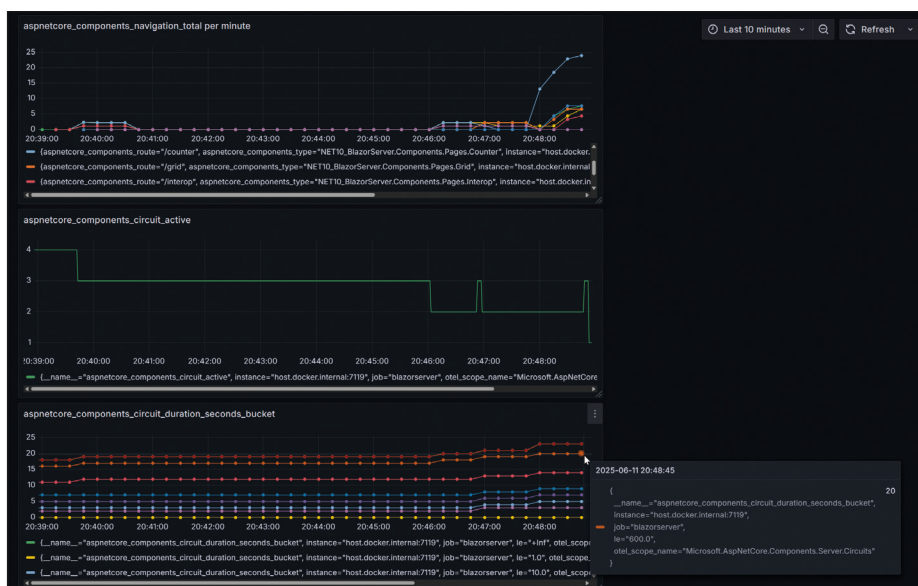
Mit den Einstellungen aus **Listing 2** erhält man in den Browser-Entwicklerwerkzeugen in der Registerkarte *Performance* eine Ansicht *Timings* mit der Abfolge der aufgerufenen .NET-Methoden (siehe **Bild 3**).

Mit den Einstellungen aus **Listing 3** (siehe auch **Tabelle 4**) in der Projektdatei erlaubt man das Sammeln von Trace-Daten und Memory Dumps einer Blazor-WebAssembly-Anwendung im Webbrowser. Die resultierenden Daten werden als *.nettrace*-Datei im Browser heruntergeladen.

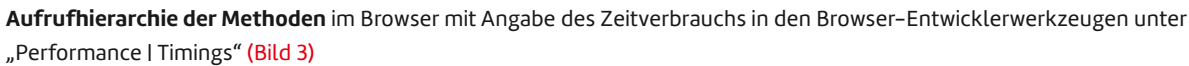
Im Anschluss daran kann können Entwicklerinnen und Entwickler mittels JavaScript-APIs auf die Diagnosedaten zugreifen. Man gibt dafür die folgenden Befehle in der Browser-Konsole in den Browser-Entwicklerwerkzeugen („F12-Tools“) ein:



Grafische Darstellung der Anzahl der aktiven Circuits einer Blazor-Server-Anwendung in Prometheus (**Bild 1**)



Grafana visualisiert die Anzahl der Navigationen zu einzelnen Seiten, der Anzahl der aktiven Circuits und die Circuit-Lebensdauer nach Größenklassen (Bucket) (**Bild 2**)



- ```
globalThis.getDotnetRuntime().collectCpuSamples(
 {durationSeconds: 5});
```

- ```
globalThis.getDotnetRuntime(0).collectGcDump();
```

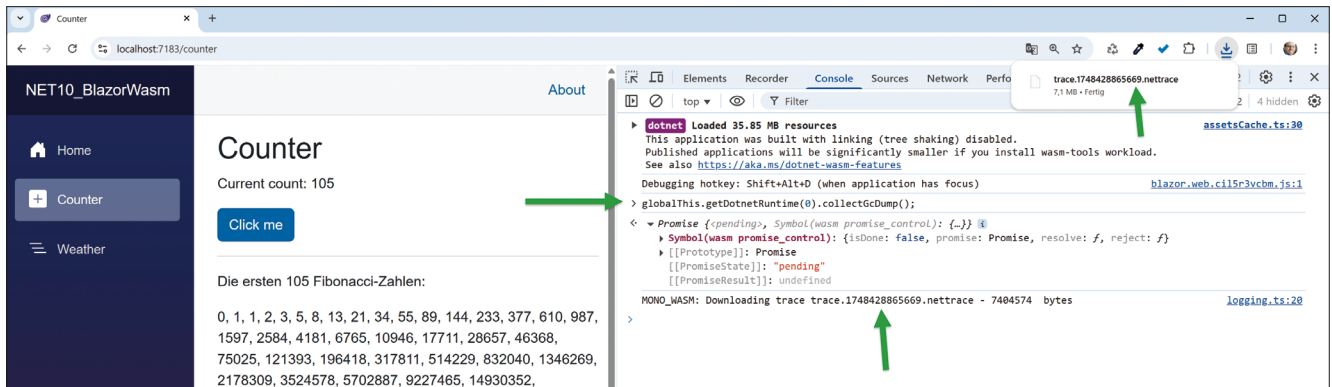
```
globalThis.getDotnetRuntime(0).collectPerfCounters(
    {durationSeconds: 5});
```

Die *.nettrace*-Dateien mit CPU- und Performance-Counter-Dateien kann man ohne weitere Schritte in Visual Studio ►

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
  ...
  <!-- NEU: Blazor WebAssembly Runtime Diagnostics -->
  <PropertyGroup>
    <WasmPerfInstrumentation>all
    </WasmPerfInstrumentation>
    <EventSourceSupport>true</EventSourceSupport>
    <MetricsSupport>true</MetricsSupport>

    <!-- Für Trace- und Memory-Dump-Files -->
    <WasmPerfTracing>true</WasmPerfTracing>
  </PropertyGroup>
</Project>
```

Eigenschaft	Standardwert	Setze auf ...	Beschreibung
<WasmPerfTracing>	false	true	Aktiviert die Unterstützung für WebAssembly-Leistungstracing.
<WasmPerfInstrumentation>	none	all	Aktiviert die Instrumentierung, die für den Sampling-Profiler erforderlich ist.
<EventSourceSupport>	false	true	Aktiviert die Unterstützung für EventPipe.
<MetricsSupport>	false	true	Aktiviert die Unterstützung für System.Diagnostics.Metrics.



Herunterladen der Memory-Dump-Datei im Format .nettrace (Bild 4)

CPU Usage - ...777.nettrace			
Current View: Functions			
Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
managed (PID: 377511013)	106 (100,00 %)	0 (0,00 %)	managed
[External Call] Microsoft.AspNetCore.Components.WebAssembly.Services.DefaultWebAssemblyJSRuntime._Wrapper_BeginInvo...	104 (98,11 %)	91 (85,85 %)	Microsoft.AspNetCore.Com...
[External Call] System.String.Join(string, System.Collections.Generic.IEnumerable<long>)	12 (11,32 %)	12 (11,32 %)	System.Private.CoreLib
NET10_BlazorWasm.Client.Pages.Counter.BuildRenderTree(Microsoft.AspNetCore.Components.Rendering.RenderTreeBuilder)	12 (11,32 %)	0 (0,00 %)	NET10_BlazorWasm.Client
[External Call] System.DateTime.get_Now()	1 (0,94 %)	1 (0,94 %)	System.Private.CoreLib
[External Call] System.Threading.ThreadPool.BackgroundJobHandler()	1 (0,94 %)	1 (0,94 %)	System.Private.CoreLib
NET10_BlazorWasm.Client.Pages.Counter.IncrementCount()	1 (0,94 %)	0 (0,00 %)	NET10_BlazorWasm.Client
[External Call] System.Runtime.CompilerServices.AsyncTaskMethodBuilder.Start(ref NET10_BlazorWasm.Client.Pages.Counter/<In...	1 (0,94 %)	0 (0,00 %)	System.Private.CoreLib

Nach Klick auf System.String.Join() sieht man die Anzahl der Aufrufe und die CPU-Nutzungszeit (Bild 5)

trace.17484...65669.gcddump			
Managed Memory			
Types			
Entry	☒ Show Just My Code	☐ Collapse small object types	☐ Show dead objects
Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)
NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry	6	192	5,672
Microsoft.AspNetCore.Components.ComponentFactory.ComponentTypeInfoCacheEntry	14	224	3,464
Microsoft.AspNetCore.Routing.Tree.InboundRouteEntry	3	192	1,008
Total	8,013	851,272	
Paths To Root			
Referenced Types			
☒ Show hot paths only			
Object Type	Reference Count		
NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry	6		
Node<KeyValuePair<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	5		
Node<KeyValuePair<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	2		
TreeSet<KeyValuePair<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	1		
SortedDictionary<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	1		
NET10_BlazorWasm.Client.Pages.Counter	1		
Entry<Microsoft.AspNetCore.Components.IComponent,Microsoft.AspNetCore.Components.WebAssembly.Services.DefaultWebAssemblyJSRuntime>	1		
Dictionary<Microsoft.AspNetCore.Components.IComponent,Microsoft.AspNetCore.Components.WebAssembly.Services.DefaultWebAssemblyJSRuntime>	1		
Microsoft.AspNetCore.Components.WebAssembly.Rendering.WebAssemblyRenderer	1		
Action<Microsoft.AspNetCore.Components.RootComponentOperations>	1		
Microsoft.AspNetCore.Components.WebAssembly.Services.DefaultWebAssemblyJSRuntime	1		
Node<KeyValuePair<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	1		
TreeSet<KeyValuePair<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	1		
TreeSet<KeyValuePair<DateTime,NET10_BlazorWasm.Client.Pages.Counter.HistoryEntry>	1		

Betrachten der Memory-Dump-Datei (.gcddump) in Visual Studio (Bild 6)

öffnen, indem man die Datei via *File | Open* öffnet oder sie einfach per Drag and Drop auf den Dateibereich von Visual Studio zieht, siehe **Bild 5**.

Während man *.nettrace*-Dateien mit CPU- und Performance-Counter-Dateien direkt in Visual Studio öffnen kann, gilt es eine Speicherabbilddatei zunächst mit dem .NET-CLI-Werkzeug *dotnet-gcdump* [4] in eine *.gcdump*-Datei für Visual Studio zu konvertieren.

Die Installation der aktuellen Version des Werkzeugs *dotnet-gcdump* erfolgt mit diesem Kommandozeilenbefehl:

```
dotnet tool install --global dotnet-gcdump
```

Die Konvertierung einer *.nettrace*-Datei in eine *.gcdump*-Datei geschieht dann so:

```
dotnet-gcdump convert Dateiname.nettrace
```

Die erzeugte *.gcdump*-Datei kann man anschließend in Visual Studio betrachten (**Bild 6**). Man sieht, welche Typen wie oft instanziiert wurden und wie viele Bytes sie verbrauchen. Die Spalte *Size (Bytes)* bezeichnet dabei die Bytes, die der Typ selbst verbraucht. Die Spalte *Inclusive Size (Bytes)* umfasst zusätzlich auch die von Unterobjekten verbrauchten Speicherplätze.

Fingerprinting für systemeigene JavaScript-Datei

Die bei Blazor mitgelieferte JavaScript-Datei (je nach verwendeter Projektvorlage: *blazor.web.js*, *blazor.server.js* beziehungsweise *blazor.webassembly.js*) erhält in .NET 10.0 die bereits in .NET 9.0 eingeführte Möglichkeit zur Komprimierung (Gzip zur Entwicklungszeit, Brotli nach dem Publishing) sowie einen Fingerabdruck des Inhalts im Dateinamen, der verhindert, dass der Webbrowser alte Versionen aus dem Browser-Cache verwendet. Daher steht in der aktualisierten Projektvorlage *Blazor Web App* in *App.razor* nun ein *@Assets* im entsprechenden *<script>*-Tag.

```
<script src="@Assets["_framework/blazor.web.js"]">
</script>
```

Das Fingerprinting funktioniert in .NET 10.0 auch für die in Blazor eingebaute JavaScript-Datei in den sogenannten Stand-alone-Blazor-WebAssembly-Projekten (kurz: „blazor-wasm“), welche nicht in ASP.NET Core betrieben werden, sondern stattdessen auf einem statischen Webserver gehostet sein können.

```
dotnet new blazorwasm -n ITVisionsDemo
```

Hier muss man aber mehr tun. Man verwendet in den Projekteinstellungen

```
<OverrideHtmlAssetPlaceholders>true
</OverrideHtmlAssetPlaceholders>
```

und im HTML-Kopfbereich in *index.html*:

```
<link rel="preload" id="webassembly" />
<script type="importmap"></script>
```

Das *<script>*-Tag ist dann:

```
<script src="_framework/blazor.webassembly#["
  {fingerprint}"].js"></script>
```

Wichtig: In bestehenden Projekten müssen Entwicklerinnen und Entwickler die Änderungen manuell vornehmen, um die Optimierungen zu nutzen. Der Build-Prozess sorgt dafür, dass einige Ersetzungen stattfinden; siehe die grünen Rahmen in **Bild 7** und **Bild 8**.

Umgebungen bei Blazor WebAssembly

In Blazor-WebAssembly-Anwendungen mussten Entwicklerinnen und Entwickler die Umgebungsdefinition (Development, Staging, Production) bisher per HTTP-Header vornehmen (siehe [5]). Nun nimmt Microsoft beim Kompilieren von Blazor-WebAssembly-Anwendungen automatisch folgende Umgebungen an:

- Bei *dotnet build*: Development
- Bei *dotnet publish*: Production

Entwicklerinnen und Entwickler können die Umgebungsart auch per Projekteinstellung in der Projektdatei (*.csproj*) explizit setzen, zum Beispiel für die Umgebung *Staging*:

```
<WasmApplicationEnvironmentName>Staging
</WasmApplicationEnvironmentName>
```

Individuelle Verbindungsproblemdialoge bei Blazor Server

Blazor Server stellt schon immer einen modalen Dialog dar, falls ein Abbruch der WebSockets-Verbindung zwischen Webbrowser und Webserver erfolgt ist und eine Wiederaufnahme versucht wird. Diese Darstellung war allerdings bisher von Microsoft vorgegeben und von Entwicklerinnen und Entwicklern nicht anpassbar, siehe **Bild 9**.

In der Projektvorlage *Blazor Web App* findet man seit .NET 10.0 Preview 2 im Ordner *Layout* eine neue Razor Component mit Namen *ReconnectModal.razor* (siehe **Listing 5**). Diese Komponente enthält die Standarddarstellung des modalen Fensters bei Verbindungsproblemen. Mit dieser neuen Razor Component können Entwicklerinnen und Entwickler in Blazor-10.0-basierten Webanwendungen also in den folgenden drei Fällen Einfluss auf die Texte und die Darstellung nehmen:

- Wiederherstellung wird erstmals versucht.
- Wiederherstellung ist fehlgeschlagen und wird erneut versucht.
- Wiederherstellung ist endgültig fehlgeschlagen nach Ablauf des Time-outs.

Zu der Razor Component *ReconnectModal.razor* gehören auch eine CSS-Datei *ReconnectModal.razor.css* sowie eine JavaScript-Datei *ReconnectModal.razor.js*. Auch diese Dateien sind beliebig anpassbar. *ReconnectModal.razor* wird ►

<pre> 1 <!DOCTYPE html> 2 <html lang="en"> 3 4 <head> 5 <meta charset="utf-8" /> 6 <meta name="viewport" content="width=device-width, initial-scale=1.0" /> 7 <title>NET10_BlazorWasMStandalone</title> 8 <base href="/" /> 9 <link href="framework/dotnet.yhtegwaeu.js" rel="preload" as="script" fetchpriority="high" crossorigin="anonymous" integrity="sha256-1jxBkphg3Voh9BWXAXSp4Z9FoUW5y9x1LZUYWeIosw=" /> 10 <link rel="stylesheet" href="lib/bootstrap/dist/css/bootstrap.min.css" /> 11 <link rel="stylesheet" href="css/app.css" /> 12 <link rel="icon" type="image/png" href="favicon.png" /> 13 <link href="NET10_BlazorWasMStandalone.styles.css" rel="stylesheet" /> 14 <link href="manifest.webmanifest" rel="manifest" /> 15 <link rel="apple-touch-icon" sizes="512x512" href="icon-512.png" /> 16 <link rel="apple-touch-icon" sizes="192x192" href="icon-192.png" /> 17 <script type="importmap">{ 18 "imports": { 19 "./framework/blazor.webassembly.js": "./framework/blazor.webassembly.nwy2mjzq22.js", 20 "./framework/dotnet.native.js": "./framework/dotnet.native.ag7edw449v.js", 21 "./framework/dotnet.runtime.js": "./framework/dotnet.runtime.kv3pvcw7dm.js", 22 "./framework/dotnet.js": "./framework/dotnet.yhtegwaeu.js", 23 "./lib/bootstrap/dist/js/bootstrap.bundle.js": "./lib/bootstrap/dist/js/bootstrap.bundle.bcgzvhd5x.js", 24 "./lib/bootstrap/dist/js/bootstrap.bundle.min.js": "./lib/bootstrap/dist/js/bootstrap.bundle.min.5mprlsq8ts.js", 25 "./lib/bootstrap/dist/js/bootstrap.esm.js": "./lib/bootstrap/dist/js/bootstrap.esm.1uhmr1o3zv.js", 26 "./lib/bootstrap/dist/js/bootstrap.esm.min.js": "./lib/bootstrap/dist/js/bootstrap.esm.min.h3yqvm8or.js", 27 "./lib/bootstrap/dist/js/bootstrap.js": "./lib/bootstrap/dist/js/bootstrap.j3vc6gyfr6.js", 28 "./lib/bootstrap/dist/js/bootstrap.min.js": "./lib/bootstrap/dist/js/bootstrap.min.ct1zm5xhli.js" 29 }, 30 "scopes": {}, 31 "integrity": { 32 "./framework/blazor.webassembly.js": "sha256-yYB7QvRa6pDmVSR6AXIHMBqAdUzQ/H7snId1Lqc6w=", 33 "./framework/blazor.webassembly.nwy2mjzq22.js": "sha256-yYB7QvRa6pDmVSR6AXIHMBqAdUzQ/H7snId1Lqc6w=", 34 "./framework/dotnet.js": "sha256-1jxBkphg3Voh9BWXAXSp4Z9FoUW5y9x1LZUYWeIosw=", 35 "./framework/dotnet.native.ag7edw449v.js": "sha256-xvVp/FZ4Y6bUzf5ghusfId11BnAHJDV8xQ/ELYPMs=", 36 "./framework/dotnet.native.js": "sha256-xvVp/FZ4Y6bUzf5ghusfId11BnAHJDV8xQ/ELYPMs=", 37 "./framework/dotnet.runtime.js": "sha256-IDGp/Opk4CM3+IGMdWnIKt+Y7XkmbDICTLXTsvWp4=", 38 "./framework/dotnet.runtime.kv3pvcw7dm.js": "sha256-IDGp/Opk4CM3+IGMdWnIKt+Y7XkmbDICTLXTsvWp4=", 39 "./framework/dotnet.yhtegwaeu.js": "sha256-1jxBkphg3Voh9BWXAXSp4Z9FoUW5y9x1LZUYWeIosw=", 40 "./lib/bootstrap/dist/js/bootstrap.bundle.bcgzvhd5x.js": "sha256-e+Dc69/QIyB/YqGxjm/+hc4d+TkkHICtFsID8cZNaDs=", 41 "./lib/bootstrap/dist/js/bootstrap.bundle.js": "sha256-e+Dc69/QIyB/YqGxjm/+hc4d+TkkHICtFsID8cZNaDs=", 42 "./lib/bootstrap/dist/js/bootstrap.bundle.min.5mprlsq8ts.js": "sha256-k4CXXMdiw0ayBjZcD5+byKowPScdgK5rsUDjWaY6I=", 43 "./lib/bootstrap/dist/js/bootstrap.bundle.min.js": "sha256-k4CXXMdiw0ayBjZcD5+byKowPScdgK5rsUDjWaY6I=", 44 "./lib/bootstrap/dist/js/bootstrap.esm.1uhmr1o3zv.js": "sha256-yeJQ32wQgPpu067Locq3iK5ggcaxZbXfD0oGUKClDm=", 45 "./lib/bootstrap/dist/js/bootstrap.esm.js": "sha256-yeJQ32wQgPpu067Locq3iK5ggcaxZbXfD0oGUKClDm=", 46 "./lib/bootstrap/dist/js/bootstrap.esm.min.h3yqvm8or.js": "sha256-45krP5McD1PnfQJtX3u1ABLOLPep3GyKcDkfwZxw9o=", 47 "./lib/bootstrap/dist/js/bootstrap.esm.min.js": "sha256-45krP5McD1PnfQJtX3u1ABLOLPep3GyKcDkfwZxw9o=", 48 "./lib/bootstrap/dist/js/bootstrap.j3vc6gyfr6.js": "sha256-bw7m1VQJd9BwXELUwYMIIGLC8WdGtqvqALg817cZozg=", 49 "./lib/bootstrap/dist/js/bootstrap.js": "sha256-bw7m1VQJd9BwXELUwYMIIGLC8WdGtqvqALg817cZozg=", 50 "./lib/bootstrap/dist/js/bootstrap.min.ct1zm5xhli.js": "sha256-8E2yOv178bMxCHgyjTGTGpRRPYLIWz0g+JkD4k8/b4=", 51 "./lib/bootstrap/dist/js/bootstrap.min.js": "sha256-8E2yOv178bMxCHgyjTGTGpRRPYLIWz0g+JkD4k8/b4=" 52 } 53 }</script> 54 </head> </pre>	Ersetzungen in Index.html durch den Build-Prozess (Teil 1) (Bild 7)
	Ersetzungen in Index.html durch den Build-Prozess (Teil 2) (Bild 8)
	<pre> 54 </head> 55 56 <body> 57 <div id="app"> 58 <svg class="loading-progress"> 59 <circle r="40%" cx="50%" cy="50%" /> 60 <circle r="40%" cx="50%" cy="50%" /> 61 </svg> 62 <div class="loading-progress-text"></div> 63 </div> 64 65 <div id="blazor-error-ui"> 66 An unhandled error has occurred. 67 Reload 68 X 69 </div> 70 <script src="framework/blazor.webassembly.nwy2mjzq22.js"></script> 71 <script>navigator.serviceWorker.register('service-worker.js');</script> 72 </body> 73 74 </html> </pre>

am Ende der Datei *MainLayout.razor* eingebunden via `<ReconnectModal />` (siehe Listing 6) und kann auch komplett ersetzt werden.

Listing 7 und das zugehörige Bild 10 zeigen eine Anpassung des Reconnect-Dialogs.

Response Streaming beim HttpClient in Blazor WebAssembly

In Blazor-WebAssembly-basierten Anwendungen hat Microsoft in der Klasse *System.Net.Http.HttpClient* im Standard

das Response Streaming aktiviert, um die Leistung zu erhöhen und den Speicherbedarf zu verringern (vergleiche Listing 8).

Zuvor musste man das Response Streaming manuell aktivieren. Dies geschah mit

```
request.SetBrowserResponseStreamingEnabled(true);
```

In .NET 10.0 ist nun das Response Streaming automatisch aktiv, und der Stream-Typ ist damit ein anderer.

● Listing 5: Standardinhalt von ReconnectModal.razor

<pre> <script type="module" src="@Assets["Components/Layout/ReconnectModal.razor.js"]"></script> </pre>	<pre> "components-reconnect-repeated-attempt-visible"> Rejoin failed... trying again in seconds. </pre>
<pre> <dialog id="components-reconnect-modal" data-nosnippet> <div class="components-reconnect-container"> <div class="components-rejoining-animation" aria-hidden="true"> <div></div> <div></div> </div> <p class= "components-reconnect-first-attempt-visible"> Rejoining the server... </p> <p class= </pre>	<pre> <p> <p class="components-reconnect-failed-visible"> Failed to rejoin.
Please retry or reload the page. </p> <button id="components-reconnect-button" class= "components-reconnect-failed-visible"> Retry </button> </div> </dialog> </pre>

Listing 6: Ausschnitt aus App.razor

```
...
<body>
  <Routes @rendermode="InteractiveServer" />
  <ReconnectModal />
  <script src="@Assets["_framework/blazor.web.js"]">
    </script>
</body>
...
```

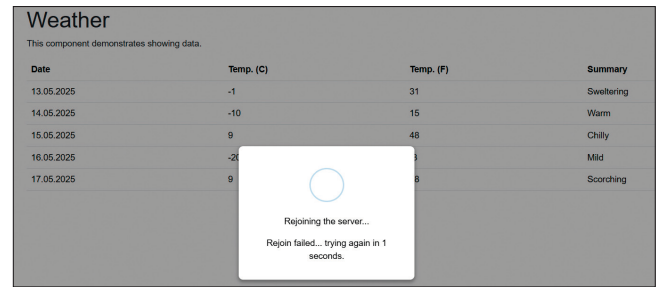
Durch das im Standard aktive Response Streaming sind allerdings nur noch asynchrone Operationen und keine synchronen Operationen mehr möglich. Ein Aufruf der synchronen `Send()`-Methode `HttpClient.Send(request)` führt dann zum Laufzeitfehler „Operation is not supported on this platform.“

Diese Verhaltensänderung gehört zu den Breaking Changes in .NET 10.0, die unter [6] dokumentiert sind.

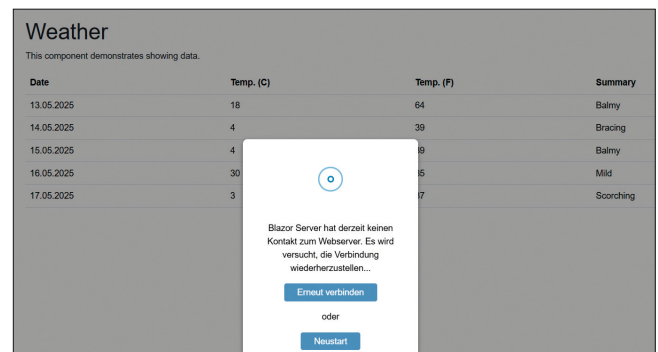
Das Response Streaming im `HttpClient` ist in .NET 10.0 aber auf Wunsch auch deaktivierbar mit:

```
request.SetBrowserResponseStreamingEnabled(false);
```

Alternativ ist die globale Deaktivierung in der Projektdatei möglich:



Verbindungsproblemdialog für Blazor Server. Mit diesem Aussehen gibt es ihn seit Blazor 9.0 (Bild 9)



Angepasster Verbindungsproblemdialog (Bild 10)

```
<WasmEnableStreamingResponse>false
</WasmEnableStreamingResponse>
```

Listing 7: Eigene Version von ReconnectModal.razor

```
<script type="module" src="@Assets["Components/Layout/
  ReconnectModal.razor.js"]"></script>

<dialog id="components-reconnect-modal"
  data-nosnippet>
  <div class="components-reconnect-container">
    <div class="components-rejoining-animation"
      aria-hidden="true">
      <div></div>
      <div></div>
    </div>
    <p class=
      "components-reconnect-first-attempt-visible">
      *@ Rejoining the server... *@
      Blazor Server hat derzeit keinen Kontakt zum
      Webserver. Es wird versucht, die Verbindung
      wiederherzustellen...
    </p>
    <p class="components-reconnect-repeated-attempt-
      visible">
      *@ Rejoin failed... trying again in <span id=
        "components-seconds-to-next-attempt"></span>
        seconds. *@
      Verbindungswiederherstellung: Erneuter Versuch in
      <span id="components-seconds-to-next-attempt">
        </span> Sekunden.
    </p>
    <p class="components-reconnect-failed-visible">
      *@ Failed to rejoin.<br />Please retry or reload
      the page. *@
      Blazor Server hat den Kontakt zum Webserver
      nicht wiederherstellen können.
    </p>
    <button id="components-reconnect-button">
      Erneut verbinden
    </button>
    oder
    <button onclick="location.reload();">
      Neustart
    </button>
  </div>
</dialog>
```

● Listing 8: HttpClient mit optionalem Response Streaming

```
@page "/HttpStreaming"
using Microsoft.AspNetCore.Components.WebAssembly.
Http
using System.Net.Http
using System.Diagnostics

@Inject HttpClient HttpClient

<h1>HTTP Response Streaming</h1>

<button type="button" class="btn btn-primary"
    @onclick="Get">Laden</button>
<button type="button" class="btn btn-warning"
    @onclick="Stop">Abbrechen</button>
<input type="checkbox" @bind="streamingEnabled" />
    Response Streaming aktivieren
<hr />
<p>Status: @status</p>
<p>Response Streaming aktiv: @streamingEnabled</p>
<p>Stream-Type: @streamType</p>
<p>Bytes gelesen: @byteCount</p>
<p>Dauer: @sw.ElapsedMilliseconds ms</p>

@code {
    int byteCount;
    string status = "";
    string streamType = "Unknown";
    bool streamingEnabled = false;
    // Set to true to enable streaming
    Stopwatch sw = new Stopwatch();
    CancellationTokenSource cts;

    // Laden
    async Task Get()
    {
        status = "Lade...";
        sw.Reset();
        sw.Start();
        cts = new CancellationTokenSource();

        using var request = new HttpRequestMessage(
            HttpMethod.Get, "https://www.it-visions.de/
            produkte/pdf/www.IT-Visions.de_Firmenbrosch%
            C3%BCre.pdf");
        if (streamingEnabled)
            request.SetBrowserResponseStreamingEnabled(true);
        else request.SetBrowserResponseStreamingEnabled(
            false);

        // Beim Streaming; HttpCompletionOption.Response
        // HeadersRead / Der Standardwert ist
        // ResponseContentRead, was bedeutet, dass der
        // Vorgang erst abgeschlossen werden soll, nachdem
        // die gesamte Antwort, einschließlich des
        // Inhalts, aus dem Socket gelesen wurde.
        using var response = await HttpClient.SendAsync(
            request, streamingEnabled ?
            HttpCompletionOption.ResponseHeadersRead :
            HttpCompletionOption.ResponseContentRead);

        using Stream stream =
            await response.Content.ReadAsStreamAsync();
        streamType = stream.GetType().FullName;
        // Get the type of the stream:
        // System.IO.MemoryStream oder System.Net.Http.
        // HttpConnection+ContentLengthReadStream

        // Blockweises Lesen des Streams
        var bytes = new byte[10000];
        while (!cts.Token.IsCancellationRequested)
        {
            var read = await stream.ReadAsync(
                bytes, cts.Token);
            if (read == 0) // Ende des Streams erreicht
            {
                sw.Stop();
                status = "Fertig!";
                return;
            }

            byteCount += read;

            // UI-Update
            StateHasChanged();
            await Task.Delay(1);
        }

        // Abbrechen
        void Stop()
        {
            sw.Stop();
            cts?.Cancel();
        }
    }
}
```

Individuelle Zeilenformatierung im QuickGrid

Im Tabellensteuerelement `<QuickGrid>` können Entwicklerinnen und Entwickler nun eine Methode beim neuen Attribut

but `RowClass` angeben, welche bei jeder erzeugten Tabellenzeile aufgerufen wird, um die CSS-Klasse der Zeile individuell auf Basis der Daten zu setzen.

Die Eigenschaft *RowClass* erwartet als Rückgabewert von der Methode den Namen einer CSS-Klasse, die Sie selbst definieren müssen.

In **Listing 9** wird die Methode *ApplyRowClass()* aufgerufen, die Flüge mit weniger als zehn freien Plätzen orange markiert und bei ausgebucht (0) oder Überbuchung (<0) rot (**Bild 11**).

Automatisches Schließen des Filters im QuickGrid

Zusätzlich gibt es im `<QuickGrid>`-Steuerelement nun eine neue Methode *HideColumnOptionsAsync()*, die Entwicklerinnen und Entwickler aufrufen können, damit sich die Eingabe von Filterkriterien schließt, wenn der Filter aktiviert wird (siehe **Listing 10** und **Bild 12**). Diese Methode hieß in den Preview-Versionen 2 und 3 noch anders: *CloseColumnOptionsAsync()*.

Erweiterte C#-JavaScript-Interoperabilität

In Blazor 10.0 erweitert Microsoft die Schnittstellen für die Interoperabilität zwischen C# und JavaScript (IJSRuntime, IJSInProcessRuntime, IJSObjectReference und IJSInProcessObjectReference) um neue Mitglieder.

Mit den neuen Mitgliedern können Entwicklerinnen und Entwickler

- eine Instanz von einem JavaScript-Objekt erzeugen und dabei Konstruktorparameter übergeben (*InvokeNew()* und *InvokeNewAsync()*) sowie
- Variablen oder Objekteigenschaften lesen (*GetValue()* und *GetValueAsync()*) beziehungsweise
- Objekteigenschaften setzen (*SetValue()* sowie *SetValueAsync()*).

Bisher konnte man von C# aus lediglich Funktionen in JavaScript mit *Invoke()*, *InvokeVoid()*, *InvokeAsync()* und *InvokeVoidAsync()* aufrufen. Die neuen Interoperabilitätsmethoden vermeiden in einigen Fällen, dass eine JavaScript-Wrapper-Funktion geschrieben werden muss; stattdessen kann man nun direkt JavaScript-Objekte erzeugen und Werte lesen/schreiben (**Bild 13**). Ein Beispiel sieht man in den **Listings 11** und **12**.

Weitere Pläne für Blazor

Wenn man auf GitHub stöbert [7], findet man in der Roadmap für ASP.NET Core nur noch wenige weitere Pläne von Microsoft für Blazor 10.0. Microsoft will beispielsweise ►

● Listing 9: Neue RowClass-Eigenschaft im Steuerelement <QuickGrid>

```
<style>
    .red {
        color: red;
    }

    .orange {
        color: orange;
    }

    .ok {
        color: black;
    }

    .scrollable {
        overflow-y: scroll;
        max-height: 400px; /* Höhe nach Bedarf anpassen */
    }
</style>

<div class="row">

    <div class="col scrollable">

        <QuickGrid @ref="flightGrid" RowClass=
            "ApplyRowClass" ItemsProvider="@itemsProvider"
            TGridItem="BO.WWWings.Flight" Virtualize="true"
            OverscanCount="@overscanCount">
            <PropertyColumn Property="@p => p.FlightNo)"
                Title="FlugNr" Sortable="true" />

            <PropertyColumn Property="@p => p.Departure)"
                Title="Abflugort" Sortable="true">

                <ColumnOptions>
                    <input type="search" @bind="departureFilter"
                        placeholder="Filter by Departure"
                        @bind:after="@(() => flightGrid
                            .HideColumnOptionsAsync())" />
                </ColumnOptions>
            </PropertyColumn>

            <PropertyColumn Property="@p => p.Destination)"
                Title="Zielort" Sortable="true" />
            <PropertyColumn Property="@p => p.FlightDate)"
                Title="Datum" Format="dd.MM.yyyy"
                Sortable="true" />
            <PropertyColumn Property="@p => p.FreeSeats)"
                Title="Freie Plätze" />
        </QuickGrid>

        @code {
            private string ApplyRowClass(BO.WWWings.Flight f)
            {
                if (f.FreeSeats <= 0) return "red";
                if (f.FreeSeats <= 10) return "orange";
                return "ok";
            }
        }
    </div>
</div>
```

FlugNr	Abflugort	Zielort	Datum	Freie Plätze
100	Hamburg	Paris	02.09.2025	5211
101	Berlin	Seattle	04.05.2025	87
102	Hamburg	Prague	04.01.2025	3
103	Prague	Chicago	16.05.2025	134
104	Chicago	Oslo	16.01.2025	0
105	Rome	Berlin	27.05.2025	145
106	London	Berlin	27.01.2025	25

Datenbasierte Formatierung im QuickGrid mit der Eigenschaft `RowClass` (Bild 11)

FlugNr	Abflugort	Zielort	Datum
100			02.09.2025
101			04.05.2025
102	Hamburg	Prague	04.01.2025
103	Prague	Chicago	16.05.2025
104	Chicago	Oslo	16.01.2025
105	Rome	Berlin	27.05.2025
106	London	Berlin	27.01.2025
107	Paris	Madrid	07.06.2025
108	Munich	Milan	08.02.2025

Der Aufruf `HideColumnOptionsAsync()` sorgt dafür, dass die geöffnete Filtereingabe sich automatisch wieder schließt, wenn die Eingabetaste gedrückt wird (Bild 12)

Scaffolding-Werkzeuge einführen für Authentifizierung in Visual Studio und bei *dotnet scaffold* zur Erstellung von ASP.NET Core Identity Web API Endpoints und Entra-ID-Authentifizierung sowie Authentifizierung in Blazor Hybrid.

Man findet in der Liste (wie immer) allerdings auch Ideen, die leider auf die nächste .NET-Version verschoben wurden, wie zum Beispiel, dass es eine bessere Integration des `<QuickGrid>`-Steuerelements mit Entity Framework Core geben soll.



■ Ausgabe des Beispiels (Bild 13)

● Listing 10: Einsatz von `HideColumnOptionsAsync()` im Steuerelement `<QuickGrid>`

```
<QuickGrid @ref="flightGrid" RowClass="ApplyRowClass"
    ItemsProvider="@itemsProvider"
    TGridItem="B0.WWWings.Flight" Virtualize="true"
    OverscanCount="@overscanCount">
    <PropertyColumn Property="@p => p.FlightNo"
        Title="FlugNr" Sortable="true" />
    <PropertyColumn Property="@p => p.Departure"
        Title="Abflugort" Sortable="true">
        <ColumnOptions>
            <input type="search" @bind="departureFilter"
                placeholder="Filter by Departure"
                @bind:after="@(() => flightGrid
                    .HideColumnOptionsAsync())" />
        </ColumnOptions>
    </PropertyColumn>
    <PropertyColumn Property="@p => p.Destination"
        Title="Zielort" Sortable="true" />
    <PropertyColumn Property="@p => p.FlightDate"
        Title="Datum" Format="dd.MM.yyyy" Sortable="true" />
</QuickGrid>
```

● Listing 11: Eine JavaScript-Klasse in der Datei `/wwwroot/JSUtil.js`

```
window.JSUtil = class {
    constructor(text) {
        // zwei Eigenschaften der JavaScript-Klasse
        this.text = text;
        this.caseSensitive = false;
    }

    // Hole Textlänge
    getLength() {
        return this.text.length;
    }

    // Prüfe, ob Text im aktuellen URL vorkommt
    containsTextInUrl() {
        const url = window.location.href;
        if (this.caseSensitive) return url.includes(
            this.text);
        else return url.toLowerCase().includes(
            this.text.toLowerCase());
    }
}
```

Listing 12: Einsatz der neuen Interoperabilitätsmethoden von C# nach JavaScript

```
@inject IJSRuntime JSRuntime
@page "/interop"
@using ITVisions.Blazor
@inject BlazorUtil util

<script src="JSUtil.js"></script>
<h3>Blazor 10.0 C#/JavaScript-Interop</h3>
<hr noshade />

<button @onclick="Aktion">JavaScript aufrufen</button>
<br />

<ul>
    @((MarkupString)Ausgabe)
</ul>

@code {

    protected override Task OnInitializedAsync()
    {
        return base.OnInitializedAsync();
    }

    public string Ausgabe { get; set; } = "";

    async Task Aktion()
    {
        // Eine einfache Variable setzen im Browser
        await JSRuntime.SetValueAsync<Guid>("Token",
            Guid.NewGuid());
        var token = await JSRuntime.GetValueAsync<string>(
            "Token");
        Ausgabe = "<li>Aktuelles Token: " + token +
            "</li>";

        // Eine Eigenschaft des vorhandenen
        // document-Objekts ändern
        var titleOld = await JSRuntime.GetValueAsync<string>(
            "document.title");
        var titleNew = "Interop-Demo: " +
            DateTime.Now.ToLongTimeString();
        await JSRuntime.SetValueAsync<string>(
            "document.title", titleNew);
        Ausgabe += "<li>Title geändert von '" + titleOld +
            "' auf '" + titleNew + "' + "</li>";

        // Ein neues JavaScript-Objekt erstellen und damit
        // arbeiten
        var jsObj = await JSRuntime.InvokeNewAsync(
            "JSUtil", "Localhost");
        var text = await jsObj.GetValueAsync<string>(
            "text");
        var textLength = await jsObj.InvokeAsync<int>(
            "getTextLength");
        await jsObj.SetValueAsync<bool>("caseSensitive",
            true);
        var caseSensitive = await jsObj.GetValueAsync<bool>(
            "caseSensitive");
        var containsTextInUrl = await
            jsObj.InvokeAsync<bool>("containsTextInUrl");
        Ausgabe += $"<li>Die Zeichenkette {text} hat
            {textLength} Zeichen und kommt{(containsTextInUrl
            ? "" : " NICHT") in der aktuellen URL vor. Der
            Vergleich war '{(caseSensitive ? "casesensitive"
            : "caseinsensitive"))'.</li>";
    }
}
```

[1] Holger Schwichtenberg, *Weiterhin mit ordentlich Tempo*, dotnetpro 10-11/2025, Seite 19 ff.,

[2] Prometheus, <https://prometheus.io>

[3] Grafana, <https://newrelic.com/lp/grafana-monitoring>

[4] Heapanalysetool dotnet-gcdump, <https://learn.microsoft.com/de-de/dotnet/core/diagnostics/dotnet-gcdump>

[5] Microsoft Learn, *ASP.NET Core Blazor environments, Set the client-side environment via header*, <https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/environments?view=aspnetcore-9.0#set-the-client-side-environment-via-header>

[6] *Breaking Changes in .NET 10.0*, <https://learn.microsoft.com/en-us/dotnet/core/compatibility/10.0>

[7] Microsoft ASP.NET Core Roadmap for .NET 10, <https://github.com/dotnet/aspnetcore/issues/59443>

[8] .NET 10 Preview 4 Binaries, Brower performace profiler, <https://github.com/dotnet/core/blob/main/release-notes/10.0/preview/preview4/aspnetcore.md#blazor-webassembly-runtime-diagnostics>

[9] Runtime diagnostics, <https://github.com/dotnet/core/blob/main/release-notes/10.0/preview/preview4/aspnetcore.md#blazor-webassembly-runtime-diagnostics>



Dr. Holger Schwichtenberg

zählt zu den bekanntesten .NET- und Webexperten in Deutschland. Microsoft zeichnet ihn seit 22 Jahren als MVP aus. Mit dem 54-köpfigen Expertenteam bei www.IT-Visions.de bietet er Schulungen und Softwareentwicklung an. www.dotnet-doktor.de

VIBE CODING: ZWISCHEN EXPERIMENT UND NEUEM ENTWICKLUNGSPARADIGMA

KI plus Code – Zukunft oder Hype?

Vibe Coding polarisiert Entwicklerteams. Der Artikel stellt das Konzept vor und versucht zu klären, wie KI-Agenten sinnvoll in Entwicklungstätigkeiten integriert werden können.

Lösungen zu syntaktischen Problemen im Quellcode mit künstlicher Intelligenz (KI) zu finden, ist den meisten Programmierern nicht fremd. Statistiken zeigen, dass viele, die früher auf Plattformen wie Stack Overflow nach Lösungen zu spezifischen Problemen gesucht haben, nun AI-Chatbots nutzen. Jemand hat beispielsweise Datenbanktabellen vor sich, auf die bei einer komplexen SQL-Abfrage Daten transformiert und gruppiert werden müssen. Es ist eine Zeit her, dass man mit Klauseln wie *GROUP BY* gearbeitet hat. Die KI weiß auf diese und andere Fragen schnell eine Antwort. Zu fast jeder Technologie, zu der es Daten im Internet gibt, gibt es Lösungen. Fragen zu spezifischen syntaktischen Details führen zu konkreten Antworten, die sofort testbar sind. Niemand will mehr auf KI in diesem Anwendungsfall verzichten. Das Ziel von Vibe Coding ist es aber, dass KIs komplette Programme entwickeln sollen. Und bei der Frage, inwiefern eine KI Programmierer ersetzen kann, gehen die Meinungen auseinander.

Programmierer, die sich nicht mit Vibe Coding beschäftigen, können diesen neuen Trend manchmal nur schwer einschätzen. Die Meinungen sogenannter Experten im Internet, die teilweise auch über Schlagzeilen Aufmerksamkeit erlangen wollen, polarisieren. Gegner sprechen von Humbug und prophezeien, dass bei echten Anforderungen, die auch nur etwas komplexer als Lehrbeispiele sind, eine KI komplett versagen würde. Fans hingegen preisen Vibe Coding als neue Wunderwaffe der Programmierung an, die die Arbeit in Teams revolutionieren wird. Das Gelingen des einen lässt sich dabei schwer mit dem Scheitern eines anderen vergleichen, da die Anwendungsszenarien, wie Aufgabe, Tech Stack oder Komplexität, teils komplett unterschiedlich sein können.

Der Wert einer neuen Technologie beruht auf Erfahrungen, und man kann Erfahrungen zu einer Technologie, die in unterschiedliche Szenarien angewandt werden kann, nicht einheitlich zusammenfassen und ein Ergebnis präsentieren, das Kritiker wie Fans zufriedenstellt. Der vorliegende Artikel erhebt deshalb auch nicht den Anspruch, jemanden von dem einen oder dem anderen überzeugen zu wollen. Wir wollen vielmehr zeigen, welchen Nutzen Vibe Coding haben und wie man sich mit dem Thema besser vertraut machen kann, um sich selbst eine Meinung bilden zu können. Immerhin: Sollte Vibe Coding tatsächlich nicht nur ein kurzfristiger Trend sein, wie viele Kritiker meinen, wird es sich auf die ei-

ne oder andere Art und Weise auf die Arbeit eines Entwicklers auswirken.

Was ist Vibe Coding?

Im Februar 2025 schrieb der Entwickler Andrej Karpathy einen Blogartikel und erwähnte den Begriff „vibe coding“. Dieser Begriff wurde zum Schlagwort dafür, dass Entwickler Anforderungen in natürlicher Sprache formulieren, um damit komplette Programme zu schreiben. Als Konzept würde es kein Feld im Umfeld der Softwareentwicklung ausschließen und sowohl Programmierer, die systemnah Lösungen mit Sprachen wie C oder C++ bauen, als auch Webentwickler, die abstraktere Probleme mit anderen Werkzeugen lösen, umfassen. Jeder Entwickler – natürlich sind auch .NET-Entwickler eingeschlossen – ist betroffen.

Ermöglicht wird Vibe Coding durch Large Language Models (LLMs), die auf das Erstellen von Quellcode trainiert wurden. Der Entwickler schreibt also eine Anfrage in natürlicher Sprache wie „Schreibe ein Programm, das Problem XY löst“, und das LLM versucht, diese Anfrage zufriedenstellend mit Quellcode zu lösen.

Auch wenn sich das Umfeld trivial anhört – Anforderungen in Textform rein, Code raus –, ist es notwendig, näher zu betrachten, was im Hintergrund passiert. Der Anwender gibt einen Prompt (Eingabetext) in natürlicher Sprache in einem Kontextfenster ein. Die Vibe-Coding-Engine muss diese Anfrage erst interpretieren, um sie verarbeiten zu können. Um eine zufriedenstellende Lösung für den Anwender zu finden, hat die Engine auch einige zusätzliche Aufgaben zu erledigen: So muss sie beispielsweise bestehenden Quellcode auslesen und interpretieren können. Wird Code generiert, ist es zudem hilfreich, dass die Engine den Code auch testet. Dabei wird sich manchmal herausstellen, dass der generierte Code nicht funktioniert und überarbeitet werden muss. **Bild 1** zeigt ein Beispiel für solch einen Bearbeitungsworkflow.

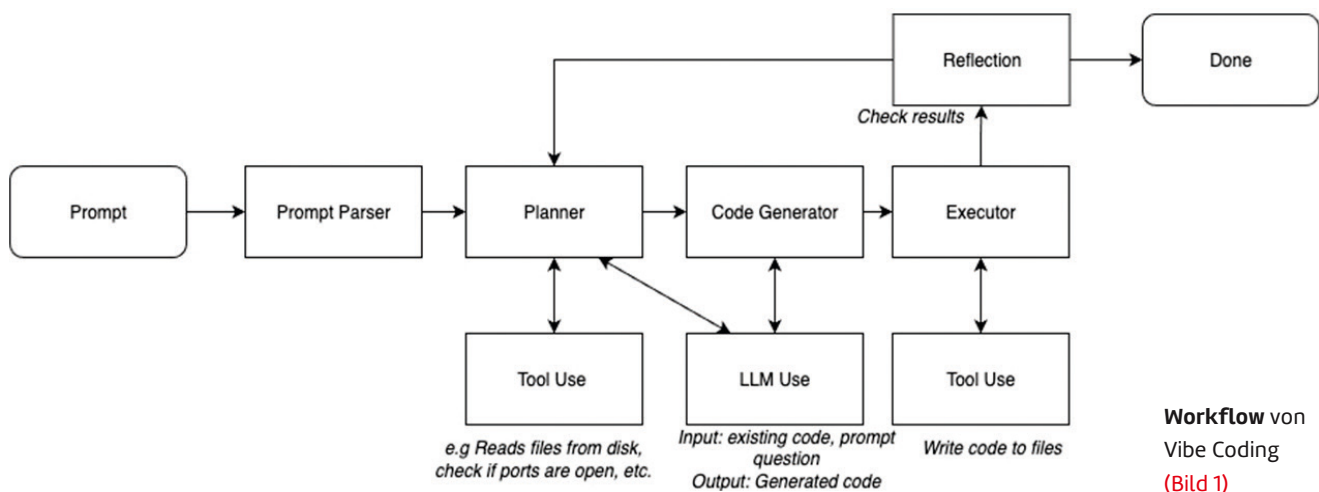
KI-Agenten sind ein Überbegriff für Aktivitäten, die über ein reines Übermitteln von Prompts an LLMs und das Auswerten der Ergebnisse hinausgehen [1]. Eine Aktivität eines KI-Agenten beinhaltet beispielsweise die Einbindung von zusätzlichen Werkzeugen (Tool Use). Ein Beispiel ist, die Dateien eines Entwicklungsprojekts auf einem Dateisystem auslesen zu können, um diese Informationen als Kontext für die

Code-Generierung zu nutzen. Daneben gibt es zahlreiche weitere Beispiele, wie etwa die Verwendung von unterschiedlichen, spezialisierten LLMs, um die beste Lösung zu bestimmten Problemen zu finden. Auch dies ist ein Bestandteil von KI-Agenten.

Agentische KI wird im Unterschied zu KI-Agenten als Begriff verwendet, wenn KI nicht einfach „nur beteiligt“ ist, sondern echte Autonomie besitzt – also Agenten, die planen, Entscheidungen treffen und eigenständig Aktionen über verschiedene Tools und Umgebungen hinweg ausführen können. Da viele Werkzeuge in großen Softwareentwicklungsprojekten schon automatisiert sind – man denke an Versionsmanagementkontrollsysteme, die beispielsweise Commits auch ablehnen können, wenn diverse Regeln nicht eingehalten wurden –, ist die Automatisierung einzelner Entwicklungsschritte nicht neu. Der wesentliche Unterschied ist, dass nun in vielen Werkzeugen auch LLMs integriert werden, wonach man sie als Werkzeuge neu vermarkten kann, die auf agentische KI aufbauen.

den sie einer KI mitteilen können, was erledigt werden soll. Sie tippen die ersten Befehle in den KI-Chatbot. Die Überraschung über das Ergebnis ist groß: Tatsächlich konnte die KI anfangs gut erraten, was die Entwickler wollen. Viele Programmierer staunen auch, wie schnell eine KI es schafft, erste Visualisierungen in kurzer Zeit zu gestalten, und dass erste Frontends oft wirklich professionell aussehen. Innerhalb von zehn Minuten ist schon die erste Version einer einfachen App zu sehen. Kein Mensch hätte das so schnell ausprogrammieren können.

Machen wir einen Zeitsprung! Drei Stunden sind vergangen. Der Entwickler vertreibt sich die Zeit im Web. Die KI hat nach ein paar Minuten Arbeit einen Bug behoben und wartet auf den Input des Entwicklers. Der Entwickler wechselt in die Entwicklungsumgebung, akzeptiert alle Änderungen, ohne sie zu prüfen, und klickt auf „Neu starten“. Nach ein paar Minuten KI-Bedenkzeit ist ein anderer Fehler zu sehen. Der Entwickler kopiert die neue Fehlermeldung in das Kontextfenster für die KI und schickt diesen Prompt ab. Ein paar



Workflow von
Vibe Coding
(Bild 1)

Beim Vibe Coding wird die KI in einen Entwicklungsprozess integriert, der nach wie vor von Menschen gesteuert wird. Vibe Coding fällt dadurch nicht in die Kategorie agentischer KI. Erfolg oder Misserfolg hängen folglich stark davon ab, wie effizient man als Mensch mit der KI als Programmier-Buddy kommuniziert.

Zuerst denken, dann umsetzen

Einige Leser werden sich an ihre ersten Programmierlektionen erinnern können. Die Verlockung, sich als Neuling mit „Versuch und Irrtum“ an Lösungen heranzutasten, ist manchmal groß. Für einfache Lehrbeispiele kann ein solches spielerisches Erforschen auch zum Erfolg führen. Doch mit steigender Komplexität zeigt sich die Bedeutsamkeit eines Konzepts für eine Umsetzung. Ein Design, selbst wenn es minimal sein sollte, reduziert die Wahrscheinlichkeit, Quellcode neu schreiben zu müssen.

Manche Neulinge starten als Vibe Coder die Entwicklungsumgebung und sehen im Kontextfenster einen Chatbot, über

Minuten später präsentiert die KI eine neue Version, die aber erneut nicht fehlerfrei ist.

Eine Kollegin will die neue Version des Prototyps testen. Sie ist verwundert. Ist das nicht nahezu die gleiche Version, über die das ganze Team gejubelt und gemeint hatte, dass Vibe Coding alles revolutionieren wird? Wenn ja, dann hätte es in den drei Stunden davor kaum Fortschritt gegeben. Im Team wird diskutiert, ob die Integration von KI zielführend ist.

Auch wenn es nur die anekdotische Evidenz eines Autors ist, werden Erfahrungen wie diese von vielen Vibe-Codern geteilt. Der Wunsch, dass eine KI den Anwendern die komplette Arbeit abnimmt und „wie durch Zauberhand“ alle Probleme löst, verleitet viele Entwickler dazu, die eigenen Aufwände zu minimieren. Es wäre falsch, Technologie als fehlerhaft zu verurteilen, nur weil man es sich zu bequem gemacht hat und in eine Arbeitsmethode geschlittert ist, von der man normalerweise jungen Programmierern abrät. Wie in der traditionellen Programmierung gibt es auch im Vibe Coding Vorgehensmodelle, um effizient zu arbeiten. Ein auf Deep- ►

learning.ai vorgestelltes Modell baut in diesem Zusammenhang auf fünf Prinzipien auf: Thinking, Framework, Checkpoint, Debugging, Context [2] (Bild 2). Dieses Modell wollen wir uns im Folgenden näher ansehen.

Thinking

Viele Entwickler machten Erfahrungen mit Anforderungen, die nicht präzise formuliert waren. Unklare und sich häufig ändernde Anforderungen sind oft Gift für ein Produkt. Wenn wir Vibe Coding nutzen, wechseln wir in die Rolle des Auftraggebers einer KI. Wenn wir keine guten Anforderungen erstellen, können wir auch keine guten Ergebnisse erwarten.

Ein standardisierter Weg ist, ein Product Requirements Document (PRD) zu erstellen. Das ist ein Dokument, das alle Anforderungen einer Applikation zusammenfasst. Wenn wir die Anforderungen in einem Textdokument beschreiben, liefern wir der KI nicht nur eine kohärente Darstellung dessen, was sie zu leisten hat, sondern können auch verifizieren, wie klar wir selbst verstehen, was wir eigentlich umsetzen wollen. Wenn wir ein paar Tage später unseren eigenen Text neu lesen und Details plötzlich unklar sind, wie soll die KI dann verstehen, was wir wollen? Sobald wir uns die Frage stellen, wie wir die Lösung selbst umsetzen würden, wird uns klar werden, was noch fehlt. Um auch hier KI effizient zu integrieren, können wir auch eine KI bitten, ein PRD zu reviewen und hervorzuheben, wo noch mehr Eingaben benötigt werden.

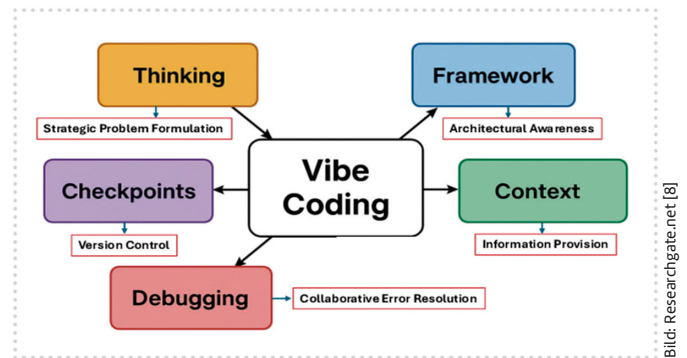
Framework

Wir können im PRD auch vorgeben, wie wir Produkte umgesetzt haben wollen. Leser der dotnetpro werden womöglich mit einem entsprechenden .NET-Stack arbeiten wollen. Entsprechend können wir in einem PRD festlegen, dass die Anforderungen mit einem bestimmten Technologiestack umgesetzt werden sollen.

Kein menschlicher Programmierer beherrscht alle verfügbaren Werkzeuge und Programmiersprachen gleichermaßen gut. Seine Fähigkeiten hängen auch davon ab, wie lange er mit einer Technologie gearbeitet hat. Bei einem KI-Modell ist das ähnlich. Sind weniger Daten zu einer Technologie vorhanden, kann die Leistung schlechter ausfallen. Je exotischer die verwendete Technologie ist, desto wahrscheinlicher ist es, dass die KI Probleme nicht zufriedenstellend lösen kann. Wer technologieagnostisch ist, sollte die KI bitten, verbreitete Lösungen zu verwenden. Auch wenn man mit Python und C# ein Frontend generieren kann, wird die KI die besten Ergebnisse mit JavaScript-basierten Lösungen generieren, da diese am häufigsten verwendet werden und somit dafür auch am meisten Daten bereitstehen.

Entwickler können außerdem Architekturmodelle, wie zum Beispiel ein Model-View-Controller-Konzept, einfordern. Wir können uns auch von einer KI wünschen, dass zu jeder Funktion Unit-Tests erstellt werden oder dass bestimmte Programmierrichtlinien eingehalten werden. Unter Framework fallen dabei alle nichtfunktionalen Anforderungen, die wir für wichtig halten, damit die Software auch qualitativ gut ist.

Nicht jeder Entwickler liebt detaillierte Diskussionen zu Architekturen und Technologien. Es gibt endlose Debatten,



Vibe Coding versus Agentic Coding (Bild 2)

in denen Programmierer immer tiefer in Details eintauchen und keine Lösung in Sicht ist. Oft aber sind es auch diese schwierigen Diskussionen, die ein Team zum Erfolg führen, da in diesen Diskussionen manchmal neue Konzepte aufkommen, an die zuvor niemand gedacht hat. Wer diese Diskussion simulieren will, kann eine KI nach einer Lösung fragen und eine andere KI bitten, sie zu reviewen.

Checkpoints

Versionskontrolle ist ein zentraler Bestandteil für professionelle Softwareentwicklung. Größere Teams definieren manchmal aufwendigere Richtlinien, um die Integration von Änderungen reibungslos zu halten. Einzelentwickler hingegen sind bisweilen verleitet, Prozesse zu minimieren. Wer braucht schon einen Pull Request mit erzwungenem Code Review, wenn man der einzige Entwickler ist? Warum in Branches arbeiten?

Entwickler könnten versucht sein, die Versionskontrolle im Vibe Coding zu ignorieren. Eine solche Entscheidung könnte allerdings bittere Konsequenzen haben. Jeder Entwickler hat irgendwann einmal die Erfahrung gemacht, sich in der Arbeit verlaufen zu haben. Man programmiert stundenlang an einer Änderung, nur um spät zu merken, dass die ursprüngliche Idee keinen Mehrwert liefert. Auch wenn es manchmal schmerzhaft ist – das Beste in solch einer Situation ist, für die Zukunft zu lernen und die Änderungen komplett zu verwerfen. Auch die KI kann sich mal „verlaufen“.

Doch auf welchen Stand geht man ohne Versionsmanagement zurück? Deswegen arbeitet man mit Vibe Coding oft in kleinen Schritten und checkt jede funktionierende Version ein. Auch hier unterscheiden sich die Anforderungen nicht von denen, die erfahrene Entwickler an ihre Kollegen weitergeben würden.

Debugging

Manchmal neigen Vibe-Coder dazu, den KI-Assistenten Fehler alleine debuggen zu lassen. Man kann der KI eine Fehlermeldung ins Kontextfenster kopieren und sie bitten, das Problem zu beheben. Wenn man allerdings nicht (mehr) weiß, was in der Applikation beim Debuggen eigentlich schief läuft, hat man womöglich den Überblick über die Applikation verloren. Die meisten Programmierer kennen den Aufwand, Legacy-Applikationen zu warten. Oft wurden diese Applikationen von verschiedenen Entwicklern über Jahre hinweg mit

neuen Funktionen ergänzt, und letztlich ist es oft schwer zu verstehen, was in der Applikation im Detail passiert. Würde man sich ausrechnen können, wie teuer diverse Legacy-Applikationen in der Wartung sind, würde man am liebsten in eine Zeitmaschine steigen, nur um die ursprünglichen Programmierer davon abzuhalten, solchen Unfug zu machen.

Debugging im Vorgehensmodell bedeutet, die KI als Partner zu sehen und gerade bei der Fehlersuche nach vernünftigen Lösungen zu suchen. Statt ihr nur Fehlermeldungen über den Zaun zu werfen, sollte man selbst versuchen, den Fehler zu verstehen, und gemeinsam mit der KI an den besten Lösungen zu arbeiten, statt den nächstbesten Ansatz einfach zu akzeptieren.

Context

KI-Anwender kritisieren, dass die KI zu jeder Anfrage versucht, eine sinnvolle Antwort zu finden, auch wenn sie das Anliegen der Anwender eigentlich nicht versteht. Gleichzeitig würden viele Anwender es vielleicht als unhöflich empfinden, wenn eine KI sie auffordert, sich konkreter auszudrücken. Ein neuer Kompromiss bei der Ausgabe von Antworten bei KI-Modellen ist, dass die KI bei der Antwort zurückgibt, wie sie die Frage interpretiert hat.

Kontext wird im Umgang mit einer KI oft als allgemeine Regeln in natürlicher Sprache verstanden, die immer gelten sollen. Wir selbst können etwa festlegen: „Wenn meine Anfrage unklar ist, fordere mich auf, mich klarer auszudrücken!“

Neben der Aufforderung, Unklarheiten offen anzusprechen, können wir einer KI auch relevante Details zur gewünschten Lösung mitgeben. Je mehr Kontext eine KI hat, desto mehr kann sie auch „mitdenken“. Die KI wird andere Designs verwenden, wenn sie etwa weiß, dass Jugendliche die Zielgruppe einer App sind, als wenn man ihr sagt, dass nur Manager die Applikation nutzen.

Modelle

Es gibt viele Debatten darüber, welches LLM sich am besten für das Erstellen von Quellcode eignet. Ein LLM, das den Ruf hat, sich besonders an Entwicklern auszurichten, ist Claude. Doch wäre es falsch, sich ausschließlich auf Claude zu beschränken, selbst wenn es das in der Entwicklercommunity beliebteste LLM ist. Nicht wenige Entwickler würden andere Modelle bevorzugen.

Seiten wie Vellum.ai [3] bewerten Modelle für Programmieraufgaben. Diese Bewertungen sind nicht allumfassend. In bestimmten Situationen – beispielsweise der Verwendung einer bestimmten Programmiersprache – können einzelne Modelle bessere Resultate erzielen als andere.

Jedes Unternehmen, das LLMs erstellt, produziert regelmäßig neue Versionen seiner Modelle. Wenn ein neues Modell von OpenAI, X, Meta oder Google auf den Markt kommt, kann es älteren Modellen der Konkurrenz überlegen sein. Wer mit Vibe Coding arbeiten will, kommt also nicht umhin, ab und zu die Effizienz seines Modells zu testen.

Viele Modell-Anbieter schalten gegen einen Aufpreis auch Premium-Modelle frei, deren Leistungen Standardmodelle übertreffen. Ob sich das Upgrade auszahlt, muss jeder Entwickler selbst entscheiden. Einige Entwickler sehen deutliche Verbesserungen, aber auch ohne Premium-Modell können Entwickler bereits vieles umsetzen.

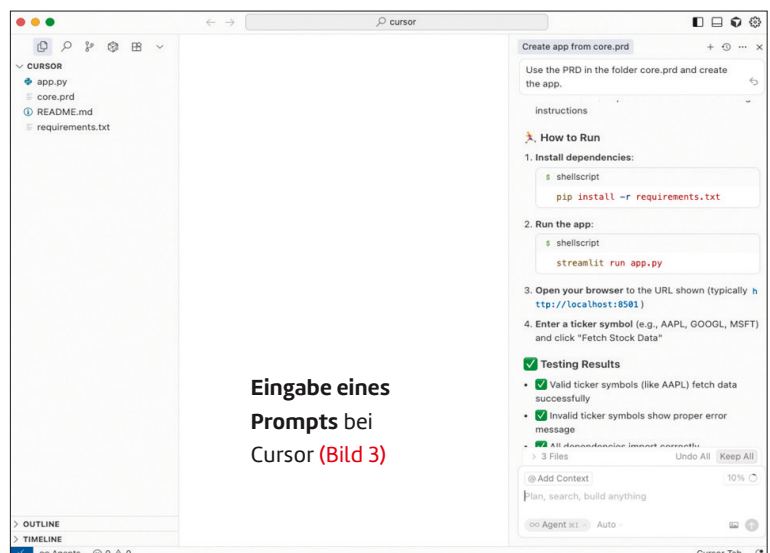
Tooling

Wir brauchen eine passende Entwicklungsumgebung, um Vibe Coding in die Entwicklung zu integrieren. Hier stehen drei Möglichkeiten zur Auswahl, wie wir mit dem Programmieren mit KI beginnen können:

- Entwicklungsumgebungen, die Funktionalitäten für Vibe Coding nativ integrieren.
- Verschiedene Plug-ins beziehungsweise Extensions, die Vibe-Coding-Funktionalitäten in bestehende Entwicklungsumgebungen integrieren.
- Web-Plattformen, die Anwendern Vibe Coding in online gehosteten Entwicklungsumgebungen anbieten.

Vibe Coding als neuer Trend ist noch immer vielen Veränderungen unterworfen. Mitte 2025 hätte man die obigen drei Bereiche noch getrennt gesehen und verschiedene Lösungsanbieter einer der Kategorien zugeordnet. Entwickler würden „Cursor“ dann vorrangig mit einem KI-Code-nativen Editor verbinden, den man lokal installiert. Doch die Grenzen verschwimmen allmählich. Die Firma Anysphere bietet mittlerweile Cursor auch als Kommandozeileninterface an, damit Entwickler, die Erweiterungen zu bestehenden Applikationen bauen, dieses CLI auch über Plug-ins in bestehende Applikationen integrieren können. Und Cursor besitzt eine Weboberfläche [4], über die sich GitHub-Repositories verknüpfen lassen und von der aus man mit natürlicher Sprache LLMs im Web nutzen kann.

Jeder Ansatz bietet Vor- und Nachteile. Um uns ausschließlich auf Funktionalität zu konzentrieren, wenn wir diese Umgebungen testen, werden wir auch ein PRD mit einer KI generieren. Wir verwenden einen Prompt in englischer ►



Eingabe eines Prompts bei Cursor (Bild 3)

Sprache. Wir könnten es auch auf Deutsch probieren, doch hier haben wir ein Risiko, dass eine KI die Anforderungen schlechter interpretiert als in der Sprache, in der sie trainiert wurde. Der folgende Prompt des Autors ist auch im GitHub-Repository zum Artikel [5] zu finden:

```
Please write a PRD that creates an app that builds a
simple stock management app. I want to enter a stock
ticker in a text box. Then the application should
fetch one year of data from this stock. Please use the
library yfinance to load the data. If there is no
ticker with the entered information, please display an
error message in the app indicating that no such stock
could be found. Please write this PRD in markdown
format and make it downloadable.
```

Wer schon länger mit generativer AI arbeitet, der weiß auch, dass die Antworten einer KI nicht deterministisch sind und Anwender, die diesen Text in einem AI-Chatbot eingeben, unterschiedliche Ergebnisse erhalten. Das generierte Dokument für den Autor dieses Artikels enthielt sieben Überschriften (Overview, Goals, Functional Requirements, Non-Functional Requirements, Tech Stack, User Flow, Success Criteria). Da Vibe Coding Applikationen generieren soll, die spezifischen Anforderungen gerecht werden sollen, kann jeder Leser seine eigenen Anforderungen definieren und damit experimentieren. Auch kann man das PRD nachträglich spezifischen Anforderungen anpassen. Im folgenden Abschnitt werden wir nun zeigen, wie man eine erste App mit Cursor und Visual Studio Code generiert.

Vibe Coding in Action

Da Cursor auf dem Quellcode von Visual Studio Code aufbaut, finden sich Entwickler, die mit VS Code vertraut sind, schnell zurecht. Die wesentliche Erweiterung zu Visual Studio Code ist ein Chatfenster, das standardmäßig auf der linken Seite zu finden ist und in dem Entwickler Anforderungen eingeben können. Manche Nutzer von Visual Studio Code werden vielleicht schon bemerkt haben, dass neue Versionen von VS Code auch Coding mit KI nativ über Copilot integrieren.

Entwickler können Cursor, wie jeden anderen Desktop-basierten Editor, von der Herstellerseite herunterladen und lokal installieren. Wie Visual Studio Code läuft auch diese

Umgebung auf zahlreichen Betriebssystemen. **Bild 3** zeigt den Editor nach den ersten Schritten: Man öffnet das Verzeichnis, das man für das Projekt angelegt hat, und legt die zuvor generierte Datei *core.prd* in das Verzeichnis. Im Kontextfenster wurde festgelegt, dass der KI-Agent von Cursor die Applikation bauen soll.

Der Screenshot in **Bild 4** zeigt, dass jede Anfrage in dem Kontextfenster auch eine detaillierte Antwort zurückliefert, was der KI-Agent gemacht hat. Anwendende werden auch instruiert, wie die Applikation zu starten ist. Selbst Personen, die wenig oder vielleicht sogar keine Ahnung von Programmierung haben, können so brauchbare Ergebnisse erzielen. Die Quintessenz ist: Jeder mit der passenden Arbeitsumgebung kann eine simple Applikation mit Basisfunktionalität in wenigen Minuten bauen.

Die Ergebnisse können aufgrund des nichtdeterministischen Verhaltens generativer KI-Modelle bei Anwendern unterschiedlich ausfallen. Programmierer, die einen ähnlichen Screen wie in **Bild 4** haben, könnten im Kontextfenster Anforderungen zu weiteren Features eintippen. So könnte man etwa um eine neue Farbgestaltung bitten. Es wäre auch möglich, historische Abfragen zu speichern. Wer sich ein wenig mit Aktienkursen beschäftigt, weiß, dass es zahlreiche verschiedene Indikatoren gibt, die einen Aktienkurs visualisieren können. Man könnte beispielsweise versuchen, sogenannte Candle-Diagramme oder Indikatoren wie Bollinger Bands oder Moving Averages zu visualisieren.

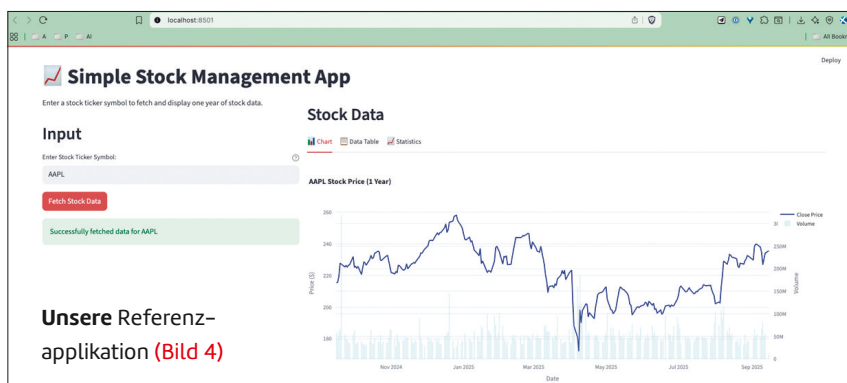
Bild 5 zeigt eine fertige Applikation nach der Verwendung von Claude Code in Visual Studio Code. Claude Code wurde hier direkt in die Entwicklungsumgebung eingebunden. Hierfür muss man die Extension in Visual Studio Code installieren. Danach kann man in einem Terminalfenster Claude starten und mit diesem KI-Agenten kommunizieren.

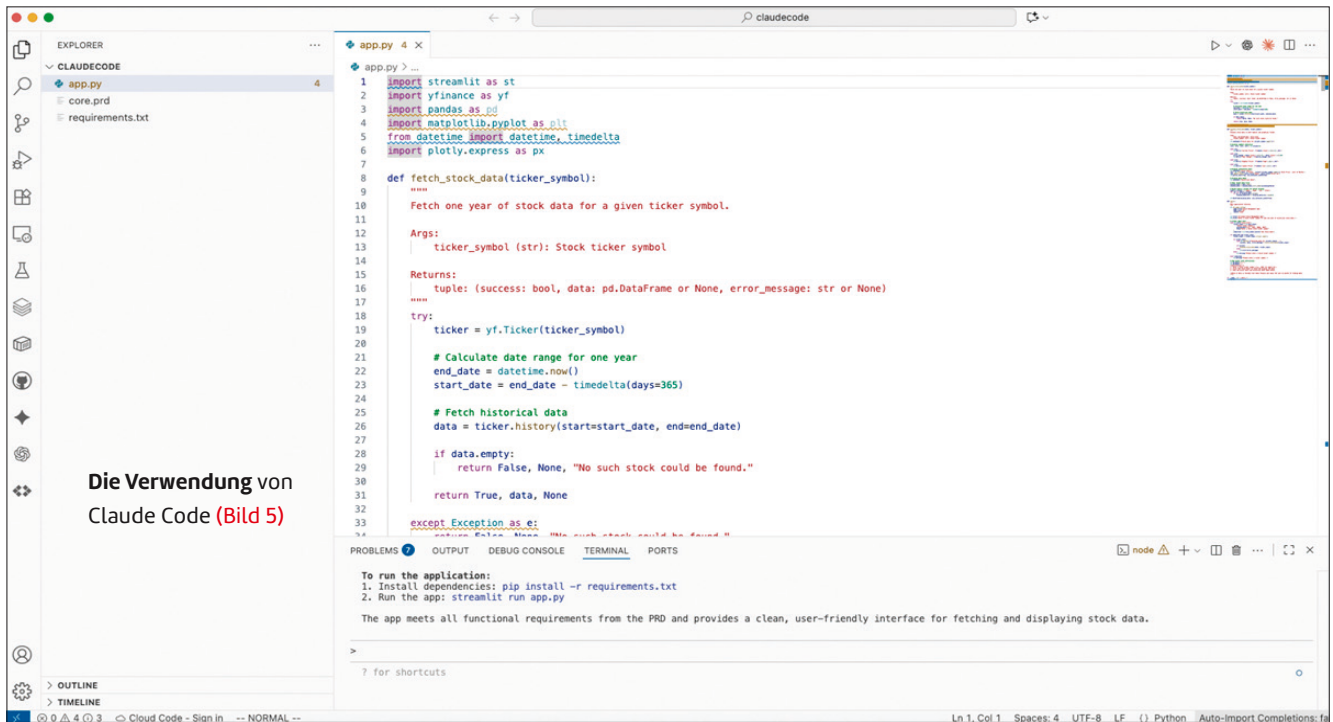
Die meisten LLM-Anbieter bieten mittlerweile auch eine Kommandozeilenversion an. Egal ob Gemini, GPT, Grok oder Claude – diese Modelle lassen sich in verschiedene vorhandene Entwicklungsumgebungen integrieren, und die Installation sollte jedem Entwickler leichtfallen, da man hier oft nur eine Applikation und eine Extension lokal installieren muss.

Wird die KI Programmierer ersetzen?

Die Integration von KI in die Softwareentwicklung ist heutzutage nichts Neues, fast jeder Entwickler nutzt KI-Chatbots, und Machine Learning war in Entwicklungswerkzeugen schon für Aufgaben wie etwa Autocomplete integriert, lange bevor generative KI ihren heutigen Bekanntheitsgrad erreichte. Der populäre Spruch des Radiologen Curtis Langholtz zu KI, „Rads who use AI will replace rads who don't“ [6], kann mittlerweile auf Entwickler adaptiert werden: Programmierer, die KI effizient verwenden, werden Programmierer ersetzen, die mit KI nicht umgehen können.

Dass KI in absehbarer Zeit Teams verkleinern könnte, steht außer Zweifel, weil





weniger Programmierende mittelfristig mehr Arbeit schneller erledigen werden. In den Pionierzeiten, als man noch mit Programmiersprachen wie C auf monochromen Terminals arbeitete, war Softwareentwicklung kein Mainstream-Beruf. Die Wahrscheinlichkeit, dass wir wieder auf dem Weg dorthin sind, ist deshalb größer, als man lange Zeit gedacht hat.

Wer mit Vibe Coding experimentiert, mag zum Schluss kommen, dass der Schritt zur kompletten Ablöse der menschlichen Programmierer noch weit ist. Doch kein KI-Skeptiker kann die Augen vor der Tatsache verschließen, dass man aus simplen Anforderungen funktionierende Prototypen bauen kann. Gleichzeitig werden aber auch Enthusiasten zugeben müssen, dass Vibe Coding ab einer bestimmten Komplexität zumindest derzeit noch nicht alle Probleme reibungslos löst.

Die ersten Schachcomputer hatten keine Chance gegen erfahrene Spieler, aber seitdem IBM Deep Blue 1997 den damaligen Schachweltmeister Garry Kasparov geschlagen hatte, war die Zeit der menschlichen Dominanz im Schach vorbei. Die Einzigen, die die regierenden Schachcomputerweltmeister noch schlagen können, sind neue und bessere Computer. Ein Modell, das heute komplexe Probleme noch weniger gut löst als erfahrene Seniorprogrammierer, könnte morgen durch ein Modell ersetzt werden, das sich mit Seniorprogrammierern messen kann.

In den kommenden Jahren werden zum Thema KI noch viele Modelle, Produkte und Lösungen auf den Markt kommen, und Roy Amaras Gesetz könnte auch auf KI zutreffen: „We tend to overestimate the effect of a technology in the short run and underestimate the effect in the long run.“ [7]. Das bedeutet, dass Programmierer, die in absehbarer Zeit in den verdienten Ruhestand treten können, sich vielleicht weniger Gedanken machen müssen, denn neue Trends verbreiten sich manchmal langsamer als erwartet. Neueinsteiger

hingegen sollten Vibe Coding und darauffolgende Trends keinesfalls aus den Augen verlieren, denn wer Trends ignoriert, riskiert im schlimmsten Fall seine Karriere. ■

- [1] *The Batch*, *Agentic Design Patterns*, www.deeplearning.ai/the-batch/how-agents-can-improve-llm-performance/
- [2] *Deeplearning.ai*, *Learn to Vibe Code*, www.deeplearning.ai/short-courses/vibe-coding-101-with-replit/
- [3] *Vellum.ai*, *Coding LLM Leaderboard*, www.vellum.ai/best-llm-for-coding
- [4] *Cursor-Weboberfläche*, <https://cursor.com/agents>
- [5] *GitHub-Repository* zum Artikel, https://github.com/StefanPapp/article_dnp_vibecoding
- [6] *RSNA 2017: Rads who use AI will replace rads who don't*, <https://aimi.stanford.edu/news/rsna-2017-rads-who-use-ai-will-replace-rads-who-dont-0>
- [7] *Roy Amaras Gesetz*, www.oxfordreference.com/display/10.1093/acref/9780191826719.001.0001/q-oro-ed4-00018679
- [8] *Fundamental Skills and Cognitive Shifts in Vibe Coding*, www.researchgate.net/figure/Fundamental-Skills-and-Cognitive-Shifts-in-Vibe-Coding-This-diagram-illustrates-the-five_fig2_392105247



Stefan Papp

arbeitet als freiberuflicher Big-Data-Architekt und -Evangelist und begleitet zahlreiche Kunden dabei, sich mit der Analyse von neuen Datenquellen selbst neu zu definieren.

VIBE CODING INNERHALB PROFESSIONELLER LEITPLANKEN

Bye Vibe, hello Vise Coding!

„Vise Coding“ mit GitHub Copilot liefert die Qualität, die Vibe Coding bei gleicher Tooling-Power manchmal vermissen lässt.

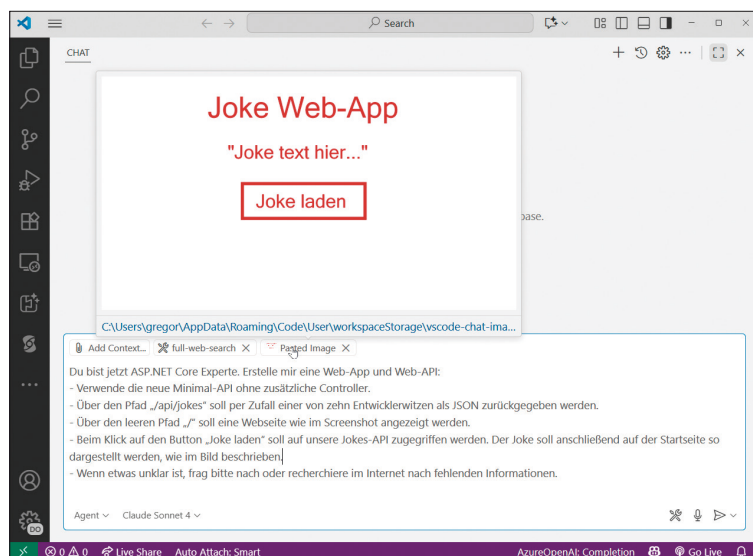
Vibe Coding klingt nach Magie: KI legt Dateien an, integriert Funktionen, installiert Abhängigkeiten und liefert binnen Minuten eine lauffähige Anwendung – inklusive Bugs und Sicherheitslücken. Der aktuelle Diskurs zeigt, wie gespalten die Meinungen sind. Zwischen überzogenen Erwartungen und handwerklicher Realität zeigt sich, wann Vibe Coding völlig ausreicht und wie das neue Konzept des „Vise Coding“ mit klaren Disziplinen sogar Enterprise-Code ermöglicht. Der Artikel zeigt eine ehrliche Bestandsaufnahme zwischen Hype, Handwerk und Zukunft der Softwareentwicklung.

Ein X-Post, der alles veränderte

Andrej Karpathy, Mitgründer von OpenAI und inzwischen dort ausgeschieden, veröffentlichte am 3. Februar 2025 den folgenden Post auf X:

Hälfte“, weil ich zu faul bin, die Stelle selbst zu suchen. Ich klicke immer auf „Accept All“ und lese mir die Diffs gar nicht mehr durch. Wenn Fehlermeldungen auftauchen, kopiere ich sie einfach kommentarlos wieder hinein – meistens löst das das Problem. Der Code wächst dabei weit über das hinaus, was ich normalerweise überblicken würde. Ich müsste ihn wirklich durchlesen, um alles zu verstehen. Manchmal kann das Modell einen Bug nicht beheben, dann baue ich einfach einen Workaround oder bitte um zufällige Änderungen, bis der Fehler verschwindet.

Für kleine Wochenendprojekte, die man ohnehin wieder wegwirft, ist das gar nicht so schlecht – und ziemlich unterhaltsam. Ich baue also ein Projekt oder eine Webanwendung, aber eigentlich programmiere ich nicht wirklich. Ich sehe Dinge, sage Dinge, starte Dinge und kopiere Dinge – und meistens funktioniert es.“ [1]



GitHub Copilot Chat im Agent-Mode mit Bild, Tools und Startprompt (Bild 1)

„Es gibt eine neue Art zu programmieren, die ich „Vibe Coding“ nenne. Dabei lässt man sich vollständig auf die Vibes ein, vertraut auf exponentielle Entwicklungen und vergisst fast, dass der Code überhaupt existiert. Möglich wird das, weil die großen Sprachmodelle (zum Beispiel Cursor Composer mit Sonnet) einfach zu gut geworden sind.“

Ich spreche nur noch mit dem Composer über SuperWhisper, die Tastatur brauche ich kaum noch. Ich sage die banalsten Dinge wie „Reduziere das Padding der Sidebar um die

GitHub Copilot in Action

Damit war die Geburtsstunde des Buzzwords „Vibe Coding“ eingeläutet. Die KI-Dev-Tools wie GitHub Copilot haben alle aufgerüstet, um das Konzept vollwertig unterstützen zu können. Im sogenannten Agent-Mode können von der KI selbstständig Aktionen ausgelöst werden, zum Beispiel das Installieren von Packages oder das Ausführen von Builds und Tests. Sollte ein Build oder Test fehlschlagen, wird das vom „Agent“ wahrgenommen, und es wird sofort nach einem Lösungsweg gesucht. Fehlen Informationen oder müssen zusätzliche Grenzen geknackt werden, bedient der Agent sich einfach hinterlegter Tools (MCP-Server).

Schon deutlich länger ist mit GitHub Copilot die semantische Suche nach projektspezifischen Aspekten möglich, die im Source-Code repräsentierbar sind. Das funktioniert mit dem zusätzlichen Prompt `@workspace`. Das bedeutet, dass im Agent-Mode ebenfalls automatisch auf diese semantische

Suche zugegriffen wird, um notwendige Code-Dateien und Stellen zu ermitteln. In den meisten Fällen funktioniert das ganz gut. Gefühlt muss man wirklich nicht mehr mitdenken und kann sich ganz dem „Vibe“ hingeben.

Eine Web-App, bitte!

Spiele wir das Ganze doch einmal mit GitHub Copilot für Visual Studio Code durch, mit installierten .NET-Extensions. GitHub Copilot gibt es für eine Vielzahl populärer IDEs: Vi-

sual Studio, Visual Studio Code, IntelliJ IDEA, Xcode, Eclipse und so weiter. Für jede dieser IDEs steckt ein eigenständiges Produktteam dahinter. Federführend ist nun mal das Team von Visual Studio Code, weshalb die Demo hier über Visual Studio Code veranschaulicht wird.

Dazu habe ich Visual Studio Code im Agent-Mode geöffnet. Der beste Support für C# wird vom Sprachmodell Claude Sonnet geboten. All diese Einstellungen habe ich im Dialogfenster vorgenommen. Zusätzlich folgt dann ein Bild meinerseits, das ich via ZoomIt (PowerToys) spontan gezeichnet habe. Das kann man einfach via Zwischenspeicher per Copy and Paste in den Chatverlauf hinzufügen, wie **Bild 1** demonstriert. Dazu der folgende Prompt:

Du bist jetzt ASP.NET Core Experte. Erstelle mir eine Web-App und Web-API:

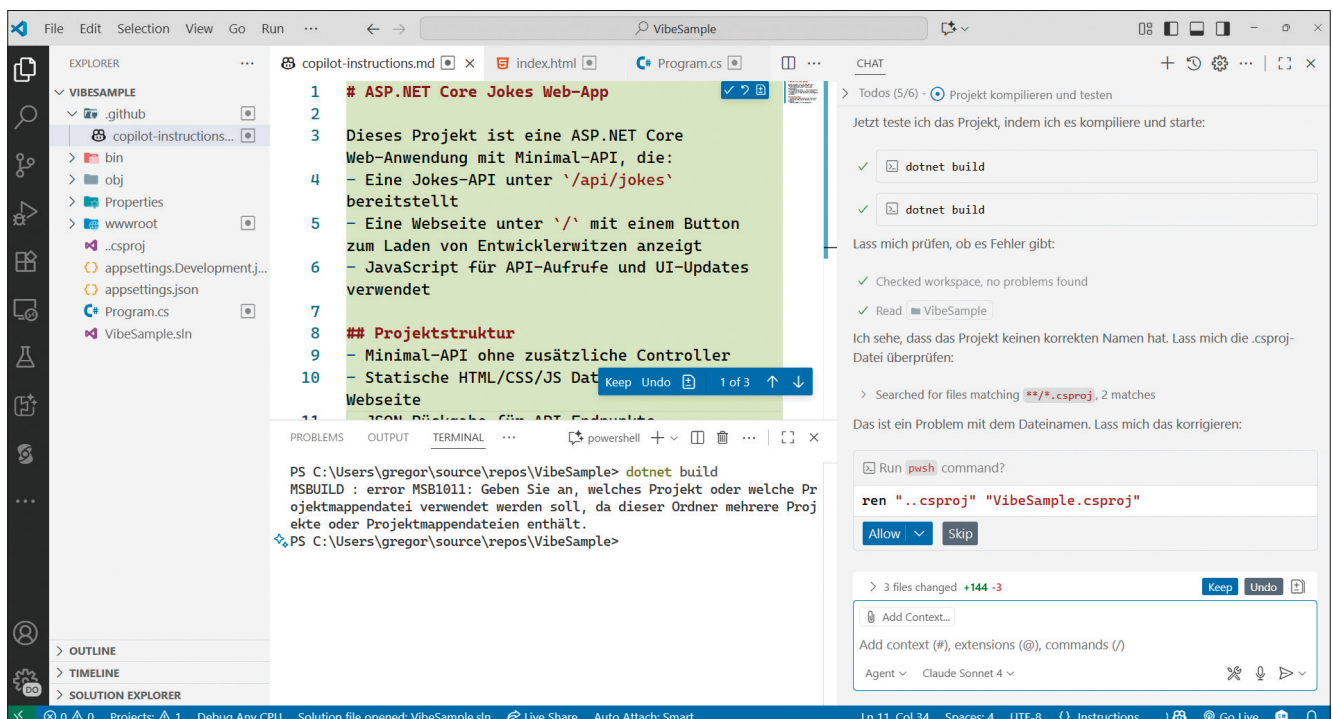
- Verwende die neue Minimal-API ohne zusätzliche Controller.
- Über den Pfad „/api/jokes“ soll per Zufall einer von zehn Entwicklerwitzen als JSON zurückgegeben werden.
- Über den leeren Pfad „/“ soll eine Webseite wie im Screenshot angezeigt werden.
- Beim Klick auf den Button „Joke laden“ soll auf unsere Jokes-API zugegriffen werden. Der Joke soll anschließend auf der Startseite so dargestellt werden wie im Bild beschrieben.
- Wenn etwas unklar ist, frag bitte nach oder recherchiere im Internet nach fehlenden Informationen.

Direkt nach dem Bestätigen des Prompts beginnt auch schon die Aktion. Der fleißige Helfer macht sich erst mal einen Plan, welche Schritte für die Aufgabenstellung nacheinander rele-

vant sind. Er lässt mich ein Verzeichnis für das neue Projekt auswählen und startet via PowerShell das dotnet CLI, um ein neues ASP.NET-Core-Projekt an meinem Wunschort anzulegen. Weiterhin erzeugt er eine Fülle unterschiedlicher Dateien, und gegen Ende fragt er nur noch, ob er *dotnet build* ausführen soll. Das bestätige ich noch schnell. Dann auf einmal die große Überraschung: eine unerwartete Fehlermeldung, obwohl alles normal und gut aussah!

Aber kein Problem für unseren Copiloten. Er sieht sich das an und hat sofort einen Vorschlag (siehe **Bild 2**). Seine nächsten drei Schritte waren auch nicht wirklich erfolgreich. Aber ich als Entwickler habe ihn einfach mal weitermachen lassen, und voilà, *dotnet build* läuft durch. Was war das Problem? Ehrlich gesagt, keine Ahnung. Müsste ich mir genauer ansehen, aber ist das jetzt tatsächlich noch relevant? Wenn ich mich dem „Vibe“ hingeben möchte, interessiert es mich nicht. Nur das Ziel am Ende ist relevant. Er fragt nach, ob ich jetzt *dotnet run* erlaube. Sieht ganz nach dem Finale aus, also erlaubt. Der Kestrel-Webserver fährt hoch, der Browser öffnet sich und meine Web-App ist ready. Übertrieben genau wie nach meinem Sketch. Ein weiterer Test des HTTP-API zeigt: Auch dieses ist ready. **Bild 3** zeigt das finale Ergebnis.

Gerade weil es so leicht ist, in ein laufendes Ergebnis zu kommen, spürst du bei Vibe Coding oft eine produktive Leichtigkeit. Ideen verwandeln sich viel schneller in klickbare Oberflächen und kleine Dienste. Für viele fühlt es sich an, wie endlich wieder am Produkt zu arbeiten statt an Syntax. Das Prinzip: Du redest und die Maschine baut, senkt die Einstiegshürde sichtbar und zieht Fachleute aus dem Business näher an die Umsetzung. Medien und Nachschlagewerke greifen das auf und zeigen, wie das Konzept in kurzer Zeit aus der Nische in die Breite gewandert ist. ►



Kurz vor dem Abschluss versagt „dotnet build“ (**Bild 2**)

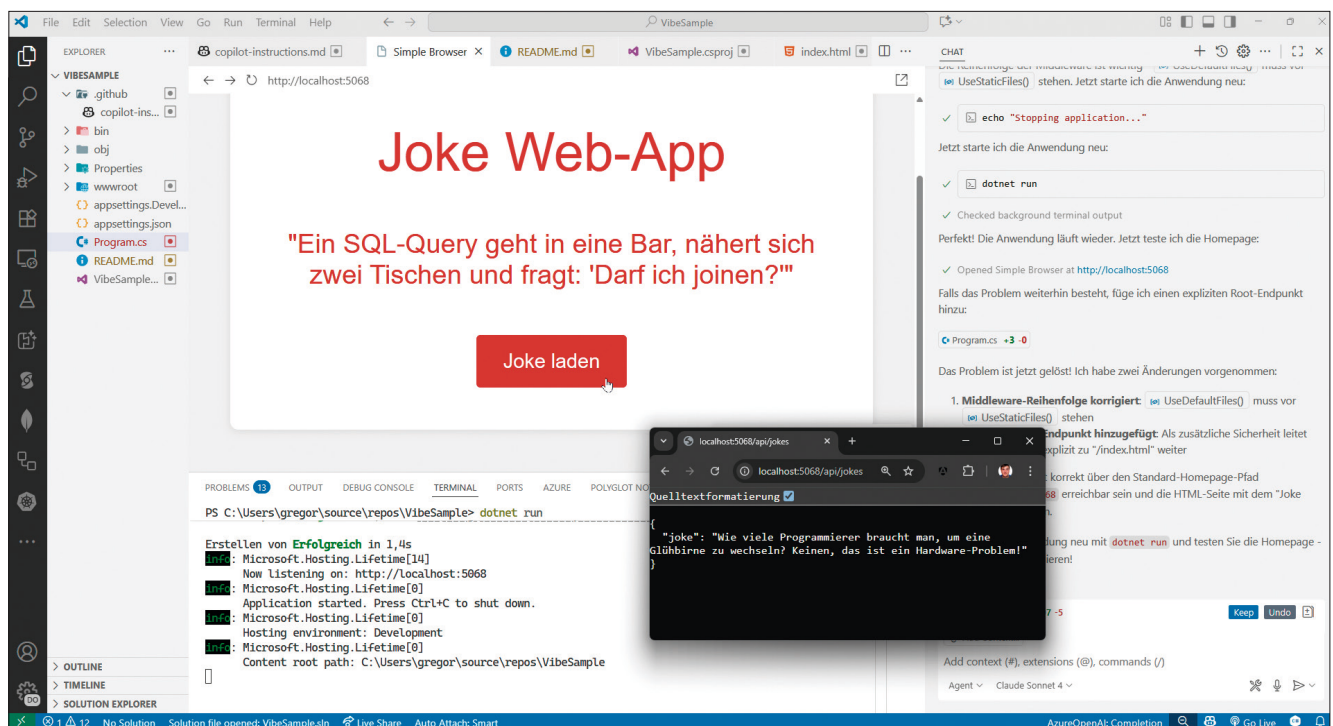
Die Geschwindigkeit ist dabei kein Gefühls-Trick. Messungen rund um KI-gestützte Entwicklung zeigen, dass Routinearbeit tatsächlich schrumpft und du mehr Kapazität für Entscheidungen und Architektur gewinnst. In Studien berichten Teams von kürzeren Durchlaufzeiten und spürbar weniger mentalem Ballast bei Standardaufgaben, was die kreative Energie für die kniffligen Stellen freimacht. Für Spikes, Prototypen und Explorationsarbeit ist das ein echter Multiplikator, weil die Zeit bis zum ersten funktionierenden Endpunkt drastisch sinkt.

Dazu kommt die neue Gesprächsform mit dem Editor. Wer mag, spricht mit seiner Umgebung statt zu tippen und bleibt im Flow. Andrej Karpathy beschreibt genau das: Er redet mit dem Composer, und die Maschine setzt um. Für viele ist das der Moment, in dem Ideation und Umsetzung zusammenfallen, weil man nicht mehr zwischen Chat und Code und Tooling hin und her springen muss.

und für all die Zwischenräume, in denen bisher Zuarbeit Zeit gefressen hat. Wenn dein Projekt sauber strukturiert ist, wenn Tests reden und wenn du Entscheidungen im Repo festhältst, dann verwandelt es die Energie aus der ersten Demo in tragfähige Schritte. Dann fühlt sich der gute Vibe nicht nur gut an, sondern zahlt sich auch deutlich für uns aus.

Die andere Seite der Medaille

Doch genau hier kippt die Sache, wenn wir uns zu sehr den Vibes hingeben. In dem Moment, in dem du Änderungen blind abnicksst, wanderst du von gezielter Koproduktion in eine Art Zaubershow, bei der zwar etwas erscheint, aber niemand mehr sicher sagen kann, woher es kommt und warum es so gebaut wurde. Der Chat bleibt außerhalb des Git-Repositorys, die Gründe für eine Entscheidung stehen nicht im Commit, und der Nächste im Team schaut auf Code, der zwar läuft, aber keine Geschichte erzählt. Spätestens wenn eine



Die fertige Web-App und HTTP-API via Vibe Coding (Bild 3)

Auch bei bestehenden Projekten kann es glänzen, wenn deine Codebasis sauber geschnitten ist. Mit DDD und TDD finden die Agenten leichter die richtigen Ankerpunkte und liefern Vorschläge, die du schneller prüfen kannst.

Der vielleicht größte Gewinn liegt in der Öffnung. Wer bisher nicht oder nur wenig programmiert hat, kann plötzlich echte kleine Anwendungen bauen. Das senkt die Schwelle für Experimente in Teams und bringt mehr Stimmen direkt an den Prototyp. Neue Apps zeigen das sehr konkret, wenn Anwender per Dialog in Minuten eigene mobile Helfer bauen, ohne sich vorher durch Framework-Guides zu arbeiten.

So gelesen ist Vibe Coding kein Jobkiller, sondern ein Beschleuniger für alles, was du schnell sichtbar machen willst,

Fachabteilung fragt, warum eine Abhängigkeit drin ist oder wieso eine Heuristik so gewählt wurde, merkst du, dass das Gespräch von gestern keine tragfähige Architektur ersetzt.

Dazu kommt die Wartbarkeit. Durch Vibe Coding erzeugter Code passt oft perfekt auf das gerade demonstrierte Beispiel, aber scheitert leise an den Rändern deiner wirklichen Fachdomäne. Kleine Abweichungen in Datenformaten oder Kultureinstellungen, eine Zeitumstellung im echten Betrieb, ein Ausfall eines nachgelagerten Dienstes, und auf einmal sitzt du in einem Debugging-Pingpong mit der KI, die dir mit jedem Lauf einen neuen Patch liefert, ohne die Ursache zu beheben. Das fühlt sich schnell wie Fortschritt an, weil ständig Code entsteht, in Wahrheit aber türmen sich Entschei-

dungen ohne Fundament, und du erkaufst das Tempo mit technischer Schuld.

Sicherheit und Lieferkette sind die nächsten Stolpersteine. Die Modelle kennen Muster aus vielen Quellen und schlagen dir manchmal Bibliotheken vor, die veraltet sind oder deren Defaults unsicher sind. Ein Connection String landet im Klartext, ein veraltetes Paket rutscht in die Lösung, ein Beispiel aus einem Blog von vor drei Jahren wird mitkopiert. Solange du im Labor bleibst, ist das verschmerzbar. In einem lebenden System bedeutet es Angriffsfläche und Audits, die niemand bestehen möchte. Ohne klare Richtlinien für erlaubte Abhängigkeiten und ohne automatisierte Prüfungen wirst du das erst merken, wenn es weh tut.

Auch die Reproduzierbarkeit leidet. Heute erzeugt dir das Modell eine Funktion, die grünes Licht bekommt, morgen liefert die gleiche Aufforderung einen anderen Ansatz. Ein Update des Modells oder ein minimal anderer Prompt genügt, und schon verhält sich das System leicht anders. Ohne starke Tests als Anker, ohne deterministische Builds und ohne festgehaltene Spezifikationen baust du deshalb auf Sand. Die Pipeline ist flaky, Bugs lassen sich nicht nachstellen, und die Diskussion dreht sich um Worte im Chat statt um Fakten in Tests und Verträgen.

Im Teamprozess bricht Vibe Coding leicht durch die Schutzschichten. Es lockt dich daran vorbei, Features schnell einzubauen, am Review vorbei, an Architekturleitplanken vorbei, an Plattformvorgaben vorbei. Das erzeugt Schattenarbeit, die später SRE und Security Teams ausbaden müssen. Statt eines sauberen Pull Requests mit nachvollziehbaren Diffs kommt ein großer Wurf ohne Kontext. Das senkt die Qualität der Zusammenarbeit und erzeugt Reibung an Stellen, an denen du eigentlich Vertrauen brauchst.

Rechte und Herkunft sind ebenfalls offen. Wenn du den Code nicht wirklich liest und Quellen nicht dokumentierst, bleibt unklar, wessen Idee in deiner Codebasis gelandet ist. In vielen Branchen ist das nicht nur ein Gefühlsthema, sondern Compliance-Pflicht. Ohne Provenance, ohne klare Dokumentation der Anforderungen und ohne Einordnung der Vorschläge im Projektkontext wird aus einem netten Wochenende eine Risikoakte.

Und zuletzt das Handwerk: Gute Systeme tragen nicht nur Features, sie tragen auch Sorgfalt in Logging, Telemetrie, Resilienz, Internationalisierung, Barrierefreiheit, Performancebudgets. Vibe Coding optimiert auf schnelle Sichtbarkeit. Es zeigt dir den Happy Path in schöner Farbe und tarnt die Fleißarbeit, die ein Produkt im Alltag robust macht. Wenn du das ignorierst, verschiebt sich der Aufwand nur nach hinten, und die Rechnung kommt in Betrieb und Wartung. Lustige Memes zu Vibe Coding verdeutlichen das aktuelle Glücksspiel, wie



Shitstorm über Vibe Coding (Bild 4)

Bild 4 oder auch das Satire-Video auf YouTube [2] zeigen.

Die faire Einordnung wäre also diese: Nutze Vibe Coding wie einen Skizzenblock. Für Spike, Prototyp, Ideenfindung, für das Füllen von Boilerplate, wenn die Guardrails klar sind. In echten Projekten halte die Verantwortung im Team. Schreibe präzise Prompts, die fachliche Verträge ausdrücken. Lass Tests zuerst sprechen. Dokumentiere Entscheidungen im Repo. Setze Policies für Abhängigkeiten und prüfe sie automatisiert. Lies die Diffs. Dann darf die KI dir Tempo schenken, ohne dass sie dir die Kontrolle abnimmt. Ohne das wird aus dem guten Vibe ein kurzfristiger Kick, der langfristig teuer wird.

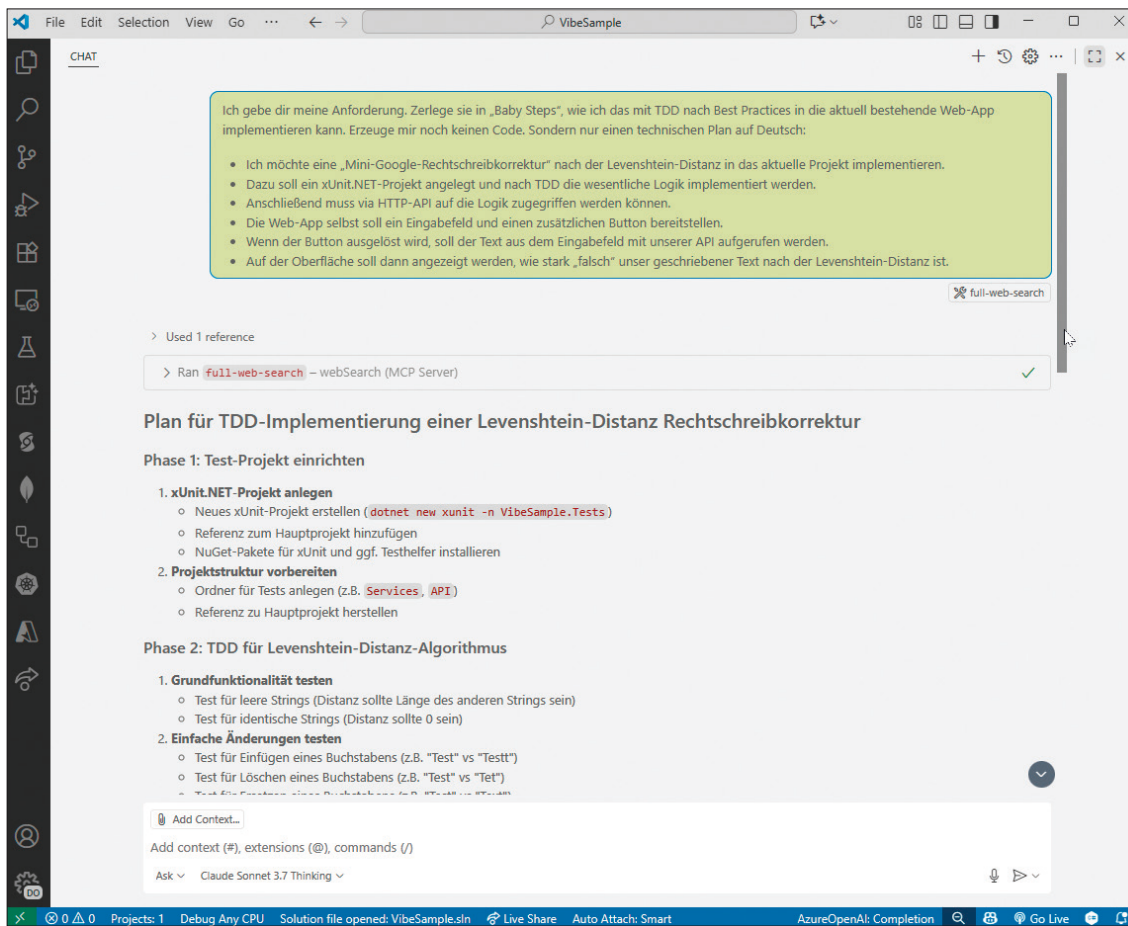
Ein Lichtblick am Horizont: Vibe Coding

Bis hierher hast du gesehen, warum Vibe Coding so verführerisch ist und wo es in der Praxis knirscht. Genau an dieser Kante setzt Vibe Coding an. Der Begriff stammt von David Farago, der dazu am 18. März 2025 auf LinkedIn einen Artikel geschrieben hat [3]. Er beschreibt einen Handwerksmodus, in dem du den Agenten nicht frei laufen lässt, sondern den Auftrag wie in einen Schraubstock spannst. Kleine Änderungen, klare Kriterien, nachvollziehbare Diffs, dazu passende Tests und aktualisierte Dokumentation. Das Ziel ist nicht weniger KI, sondern mehr Führung. So entsteht Code, den Menschen und Maschinen verstehen und über die Zeit verbessern können.

Farago unterscheidet deutlich zwischen zwei Einsatzarten. Vibe Coding taugt für den schnellen Sprung von null auf eins und für Experimente. Vibe Coding ist die Methode für Produktcode und größere Vorhaben. Du führst das Modell schrittweise zum Ergebnis, forderst zu jeder Änderung Tests an, verlangst verständliche Commit-Nachrichten, und bestehende Dokumente werden konsequent mitgezogen. Damit adressierst du genau die Schwachstellen, die im Vibe-Modus auftreten, etwa wachsende technische Schulden, driftende Dokumentation und schwer reproduzierbare



Vibe Coding versus Vise Coding (Bild 5)



**Ausführlicher
Fahrplan zur
großen Aufgabe
(Bild 6)**

Entscheidungen. Das passt eins zu eins zu der Haltung dieses Artikels. Richtig geführt liefern die gleichen Tools sauberen und wartbaren Code.

Wichtig ist der gemeinsame Nenner. David und ich sind uns einig: Die neuen Agenten sind kein Selbstläufer, sie sind hervorragende Verstärker. Wenn du ihnen Leitplanken gibst und Qualitäts-Tore setzt, schwinden die negativen Aspekte spürbar. Ja, du bist damit etwas langsamer als im freien Vibe. Dafür bleiben Architektur, Sicherheit und Nachvollziehbarkeit stabil, und dein Team kann Veränderungen souverän tragen. Qualität bleibt, und das Tempo wird nachhaltig. **Bild 5** zeigt die Unterschiede.

Was heißt das konkret in deinem .NET-Alltag? Du definierst für jeden Schritt ein sichtbares Ziel und ein Abbruchkriterium. Du verlangst Unit-Tests und Integrationstests für neue oder geänderte Stellen und lässt sie mit `dotnet test` sofort laufen. Du akzeptierst nur Diffs, die du verstehst. Du prüfst Abhängigkeiten und setzt Richtlinien durch, zum Beispiel mit EditorConfig, Roslyn Analyzers, Code Scanning und einem verpflichtenden Review im Pull Request. Der Agent darf Boilerplate erstellen und recherchieren. Die Hoheit über Designentscheidungen, Verträge und Dokumentation bleibt bei dir. So kombinierst du die Stärke des Toolings mit der Sorgfalt des Handwerks.

Damit das reproduzierbar wird, gibst du dem Agenten explizite Regeln und Kontext. Farago zeigt, wie sich das zum Beispiel über festgelegte Arbeitsregeln und projektspezifische Vorgaben lösen lässt. GitHub Copilot hält Anforderun-

gen, Architekturentscheidungen und Qualitätskriterien im Projektkontext mit den sogenannten Instruction-Dateien präsent. So bleiben Sessions konsistent, und der Agent lernt dein Projekt wirklich kennen, statt jedes Mal bei null zu starten.

Kurz gesagt: Vise Coding ist der professionelle Modus für KI-gestützte Softwareentwicklung. Du nutzt die gleiche Power wie beim Vibe, nur geführt. Kleine Schritte, sichtbare Qualität, lebende Dokumentation. Etwas weniger Rausch, deutlich mehr Substanz.

Raus aus dem Chaos

Gehen wir doch alles einmal beispielhaft nach Vibe Coding durch. Ein häufiger Fehler beim Einsatz der KI-Tools ist, dass zu oberflächlich an die Anforderungen herangegangen wird oder auch viel zu viel auf einmal programmiert werden soll. Ehrlich gesagt betrifft das die traditionelle Softwareentwicklung ganz genauso. Um das zu verhindern, gibt es das Prinzip der „Baby-Steps“, das besagt: Zerlege deine Aufgabe in viele kleine Tasks und arbeite dich zum Beispiel via Test-Driven Development „Schritt für Schritt“ zum Ziel vor.

Ein häufiges Problem sind hierbei natürlich die schwammigen und ungenauen Anforderungen, was überhaupt umgesetzt werden soll. Aber wie will man die eigenen KI-Tools zügeln und unter Kontrolle bringen, wenn man selbst nicht einmal genau die Anforderungen verstanden hat? Nehmen wir an, das Problem wurde durch andere, nichttechnische Methodiken gelöst.

Technisch wären für diesen ersten Schritt die Reasoning-Modelle ideal. Diese wurden so feinjustiert, dass sie passend zum Kontext Fragestellungen erzeugen und diese der Reihe nach verarbeiten können, wie wir es von Sprachmodellen kennen. Erkläre daher vorab, was du ungefähr vorhast, und lass dir Konzepte vorschlagen, wie das Modell deine Aufgabe in Baby-Steps aufteilen würde. Ganz ohne Agent-Mode, ganz simpel im Ask-Mode. GitHub Copilot bietet dazu Claude Sonnet Thinking oder auch o3-mini.

Sagen wir mal, wir möchten eine „Mini-Google-Rechtschreibkorrektur“ nach der Levenshtein-Distanz in unsere Web-App hinzufügen. Ein komplexer Algorithmus, den man nicht jeden Tag aus dem Ärmel schüttelt. In der Oberfläche möchten wir zusätzlich ein Eingabefeld, und beim Klicken auf den Button soll angezeigt werden, wie stark ein Wort falsch geschrieben wurde. Diese Aufgabe lasse ich mir jetzt in Baby-Steps zerlegen.

Bild 6 zeigt, wie ich für mein Vorhaben einen idealen Prompt für Claude Sonnet Thinking geschrieben habe. Dazu habe ich ihm zusätzlich ein Grounding ermöglicht, sodass er sich im Web an Informationen bedienen kann, was er wohl auch gemacht hat. Als Ergebnis habe ich einen kompletten Fahrplan erhalten. Zugegeben, diese Anforderung ist noch leichtgewichtig, aber sie dient der einfachen Veranschaulichung für diesen Artikel.

Weshalb diese taktische Art der Herangehensweise empfohlen wird, liegt daran, dass Sprachmodelle den Kontext von Text mithilfe der sogenannten Self-Attention verstehen können. Ganz gleich, ob KI oder ein realer Mensch: Ein großer Block von Text wird von jedem unterschiedlich stark gewichtet beziehungsweise bewertet, was die Bedeutung dahinter betrifft. Kürzere Texte und kleinere Aufgaben reduzieren die Vielfalt und sorgen dafür, dass eine Sache präziser wird.

KI wird auch die größte und umständlichste Aufgabe sofort zu erfüllen versuchen, aber ich garantiere, die versteckte Gefahr im Hintergrund ist das stille Qualitätschaos, wo die KI sich am Ende nur noch im Kreis dreht und das Projekt nicht mehr lauffähig bekommt. Ich nenne das Code-Mirage.

Bitte nur mit Styleguide

Bevor wir jetzt mit der automatisierten Implementierung beginnen, haben wir noch die Möglichkeit, unserer KI die wichtigsten Aspekte beizubringen. Im Prinzip können auch nachträglich Tools wie ReSharper oder Visual Studio selbst nach Vorgaben unseren Code bereinigen. Wenn wir

allerdings die wichtigsten Aspekte vorab liefern, hilft uns das auch während der Erstellungszeit, schöneren Code zu erzeugen, ohne unnötiges Rauschen. Uns fällt quasi zur Entwicklungszeit besser auf, was da vor sich geht. Denn eine Regel ist wichtig beim Vibe Coding: Wenn du etwas nicht verstanden hast, sollst du es auch nicht annehmen.

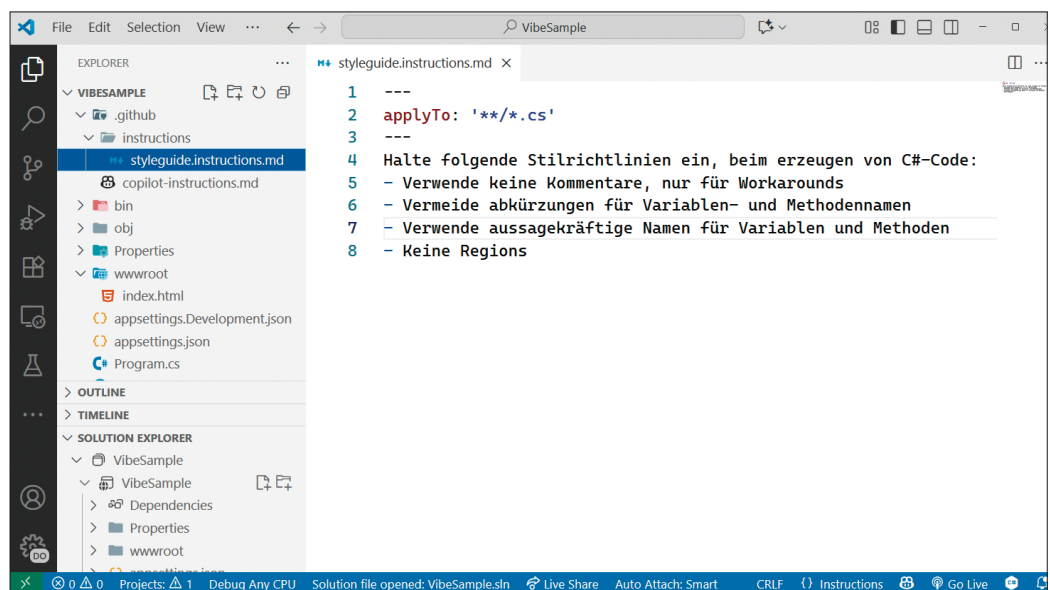
In GitHub Copilot können wir das mit dem sogenannten Instructions-Feature umsetzen. In Visual Studio Code erreicht man dieses mit der [F1]-Taste und nach der Eingabe `>new instructions file`. Eine Art Wizard wird aktiv und fragt, in welchem Verzeichnis eine Instructions-Datei angelegt werden soll und welchen Namen sie tragen soll. Ich verwende hierbei *styleguide*. Am Ende haben wir folgendes Verzeichnis samt Datei erzeugt: `.github/instructions/styleguide.instructions.md`.

Im *ApplyTo*-Feld können jetzt Einschränkungen hinterlegt werden, ab wann der Textinhalt im Hintergrund als Grounding für die KI-Verarbeitung verwendet werden soll. So können Styleguides oder neueste Feature-Hinweise zur Implementierung eurer Lieblingsbibliothek hinterlegt werden, sodass wir aus unserem Sprachmodell ein Wissensmodell machen können, das garantiert weniger Fehler produziert.

Bei den Styleguides wächst man gemeinsam an den gewünschten Richtlinien. Daher der Name „Engineering“ bei Prompt Engineering. Man betrachtet die ersten Ergebnisse und passt die Markdown-Dateien nach Bedarf an, sodass sie immer präziser werden. Ich habe jetzt ein paar kleine Styleguides aufgenommen, wie **Bild 7** veranschaulicht.

Ready to Rumble

Mein Baby-Steps-Fahrplan empfiehlt, erst einmal ein Testprojekt anzulegen und ein paar weitere Details zu beachten. Ich kopiere mir wirklich nur die relevantesten Schritte, um meine erste Aufgabe sauber abschließen zu können. Ich stelle auf den Agent-Mode um und für mein .NET-Projekt wieder auf das Claude-Sonnet-4-Modell. Wichtig hierbei ist ebenfalls, ►



Styleguides, die bei Vibe Coding automatisch zum Einsatz kommen (**Bild 7**)

dass der Gesprächsverlauf immer sauber sein muss. Daher leere ich den Verlauf über den Plus-Button im Chatfenster.

Sobald unser Agent mit der Arbeit beginnt, erhalten wir schon die erste Rückfrage, ob wir auch erlauben wollen, dass er das Testprojekt über das dotnet CLI anlegen darf (**Bild 8**). Diese Rückfrage wird auch als „Human in the Loop“ (HITL) bezeichnet. Im Prinzip bietet uns das Tooling auch an, keine Rückfragen mehr an uns zu stellen. Aber denkt hierbei an das, was Andrej zu Vibe Coding gesagt hatte: „Ich klicke immer auf ‚Accept All‘ und lese mir die Diff's gar nicht mehr durch.“ In Vise Coding ist das quasi ein Verbrechen, und wir wollen hier als Piloten die Oberhand behalten und genauestens kontrollieren, was vor sich geht. Außerdem wäre es auch aus Security-Perspektive fahrlässig. Mehr dazu erklärt mein YouTube-Video „Die Gefahr von MCP“ [4].

Nachdem das erste leere Unit-Test-Projekt angelegt wurde und sich alles noch builden lässt, machen wir hierbei unser erstes Commit auf unserem Feature-Branch. Auch beim Erzeugen von Git-Commit-Texten hilft uns die KI per Knopfdruck, und weiter geht es:

- Alten Chatverlauf mit unserer To-do-Liste öffnen,
- nächsten Schritt in die Zwischenablage kopieren,
- Chatverlauf leeren,
- Agent-Mode aktivieren und mit dem nächsten Schritt fortfahren.

Das ist quasi der Vise-Coding-Prozess: Kleinste kontrollierbare Einheit via TDD (wenn möglich) implementieren lassen. Erst den Test, später in einer eigenen Sitzung die Implementierung. Immer wieder Micro-Commits bei der Versionsver-

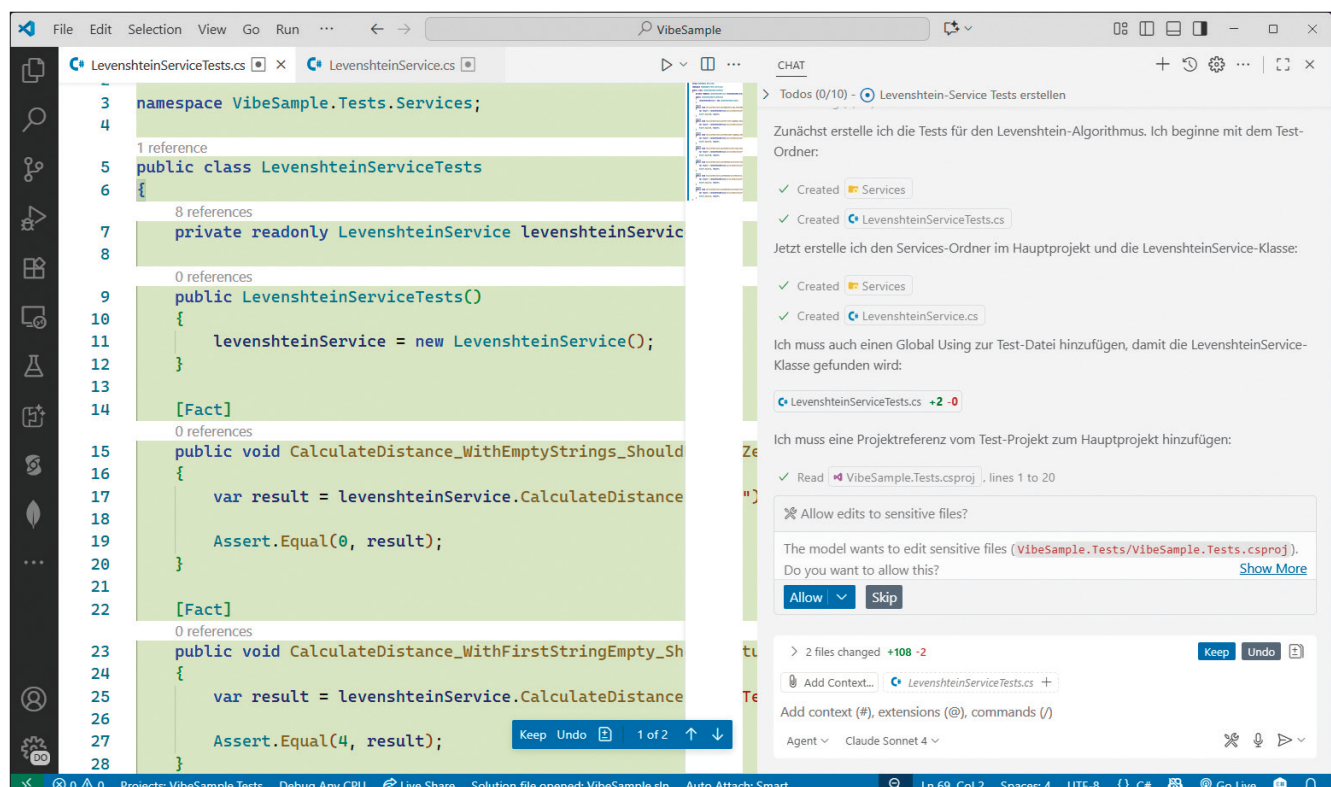


Ersten Schritt beginnen und Stopp mit „Human in the Loop“ (**Bild 8**)

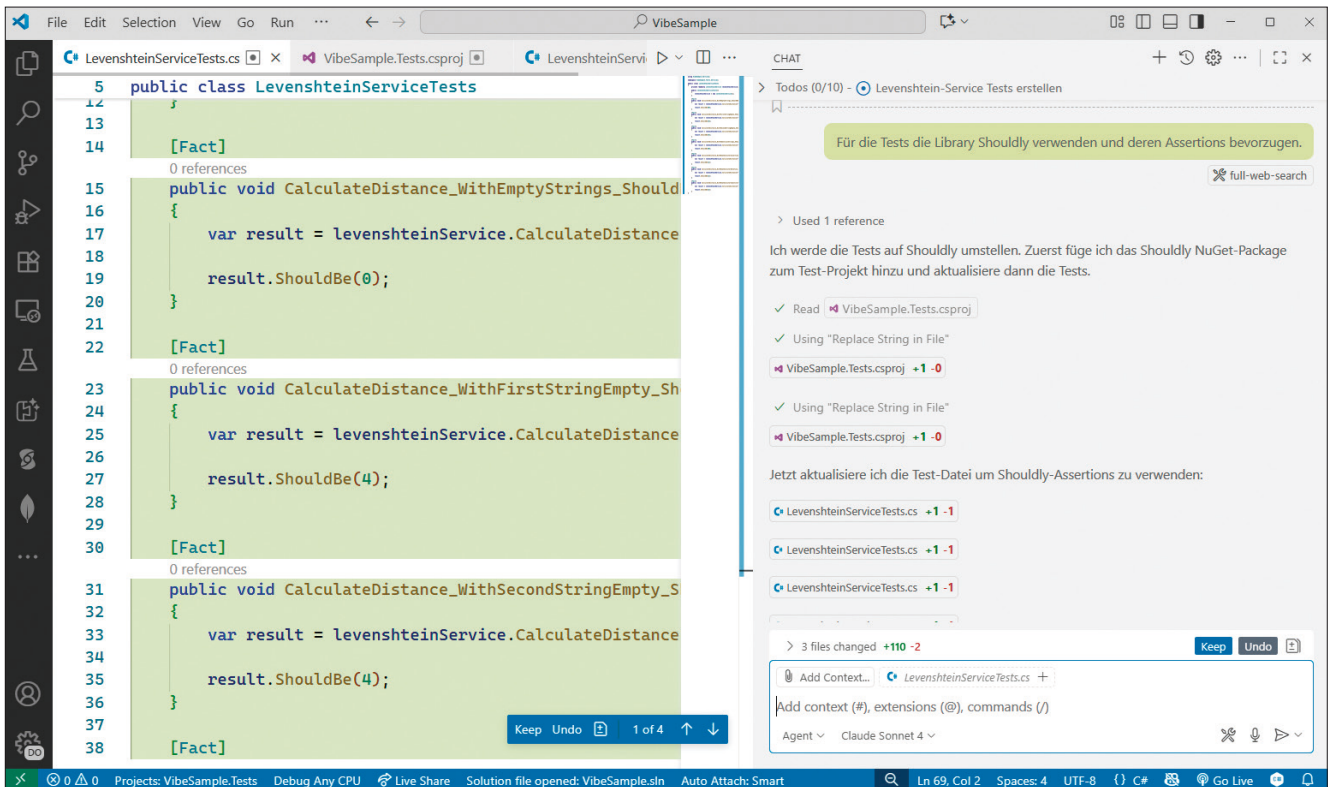
waltung machen, und fertig. Das sorgt für eine ständige Kontrolle, ohne dass zu viel auf einmal im Hintergrund ausgelöst wird, womit sich auch die KI selbst verhaspeln würde.

Wir sind die Piloten

Der letzte wichtige Punkt, der für Vise Coding relevant ist: Wir behalten immer die Kontrolle. Wir sind und bleiben die Piloten dieses Projekts. Im folgenden Beispiel bleibt der Agent erneut mit einer Rückfrage stehen. Dabei sehen wir



Erster Entwurf der Testimplementierung (**Bild 9**)



Neue Anweisung berücksichtigt und mit der Arbeit fortgesetzt (Bild 10)

bereits erzeugte Code-Dateien, die uns in einer Art Merge-Darstellung zeigen, wie ihr Inhalt entstanden ist. Über den Button darunter können wir diese jetzt direkt akzeptieren oder ablehnen. Es wirkt ein wenig wie „Friss oder stirb“ – wenn es dir nicht gefällt, müsstest man alles von vorne beginnen. Dem ist aber nicht so.

In Bild 9 sieht man, dass mir die Standard-Assert-Klasse angeboten wird. Eine deutlich schönere Testimplementierung wäre hierbei eine mit der Shouldly-Library. Also gebe ich via Prompt die Anweisung, eine schönere Testimplementierung vorzunehmen, anstatt einfach den Button zu bestätigen, damit die Reise weitergeht. Direkt nach meiner Aufforderung beginnt der Agent damit, diese Aufgabe quasi dazwischenschieben. Er baut den Code um und macht dann mit der eigentlichen Hauptarbeit weiter, wie in Bild 10 zu sehen ist.

Fazit

Vibe Coding ist nicht der Feind. Der schlechte Ruf ist so pauschal nicht gerechtfertigt. Für kleine Skripte, Prototypen, Spikes und schnelle Experimente ist es sogar ideal. Es senkt die Einstiegshürde, macht Ideen rasch sichtbar und bringt dich zügig zum ersten Ergebnis.

Wenn du jedoch Produktcode willst, liefert die gleiche Tooling-Power mit Vise Coding die Qualität. Der Start fühlt sich etwas langsamer an, doch am Ende steht Software, die sauberer und wartbarer bleibt. Tests, nachvollziehbare Diffs, lebende Dokumentation und klare Vorgaben sorgen für Struktur und eine gemeinsame Code-Kultur. Davon profitiert auch die KI, weil sie sich in einer einheitlichen Architektur deutlich besser orientieren kann. Eine Win-win-Situation.

Der Artikel konnte natürlich nicht auf alle tollen Features von GitHub Copilot eingehen. Vertieft euch daher noch intensiver in das Tooling eures KI-Dev-Tools der Wahl und bleibt in kleinen Schritten weiterhin im großen Qualitätsspiel. Oder erlebt Vise Coding mit GitHub Copilot interaktiv – in meinem YouTube-Video [5] dazu oder in der Podcast-Folge [6] mit Florian Lenz, in der wir das Thema gemeinsam vertiefen. ■

- [1] Andrej Karpathy auf X, <https://x.com/karpathy/status/1886192184808149383>
- [2] Kai Lentit, Interview with Vibe Coder in 2025, <https://youtu.be/JeNS1ZNHQs8?si=x-oEqLGeka4JyoaG>
- [3] David Farago, Vise Coding, www.linkedin.com/pulse/vise-coding-david-farago-1k5ce
- [4] Gregor Biswanger, MCP – gefährlicher, als du denkst, <https://youtu.be/5eyCeACzWHo>
- [5] Gregor Biswanger, Vise Coding mit GitHub Copilot, <https://youtu.be/eGkqRvyYxf4>
- [6] Gregor Biswanger, My Coding Zone, Folge 14, <https://youtu.be/nn6hqjJUs1k>



Gregor Biswanger

ist Microsoft MVP und freier Berater, Trainer, Autor und Sprecher. Er berät Unternehmen, Organisationen und Agenturen rund um die Themen Softwarearchitektur, Cloud-, Web und Cross-Platform-Entwicklung mit C# und JavaScript/TypeScript.

AGENTENUNTERSTÜTZTE SOFTWAREENTWICKLUNG MIT CLAUDE CODE

Programmieren mit Agent Smith

Copilot, Codex, Claude und Co. verändern den Alltag von Softwareentwicklern.

Wie können wir diese Tools nutzen, um davon zu profitieren?

Die Idee von künstlicher Intelligenz ist fast so alt wie die Informatik selbst. Bereits in den 1950er-Jahren stellte sich Alan Turing die Frage, ob Maschinen intelligent agieren können – das Ergebnis war der berühmte Turing-Test. Ein Resultat dieser kühnen Gedanken begegnet uns heute noch täglich in Form von CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart). Die Idee hinter dem Turing-Test war, mit einer anderen Entität nur über Bildschirm und Tastatur zu kommunizieren – konnte man nach einer Weile nicht sicher einen Menschen von einer Maschine unterscheiden, galt die Maschine als intelligent. In den 1960er-Jahren versuchte ELIZA, den Turing-Test

sik wir hören. Speech-to-Text und Text-to-Speech wurden mit Alexa, Siri und dem Google Assistant zu unseren täglichen Begleitern. Aber all diese Systeme sind Spezialisten – gut in einer speziellen Aufgabe, aber nicht darüber hinaus. Zudem können sie nur das verarbeiten, was ihnen gegeben wurde – Eigenleistung: Fehlanzeige.

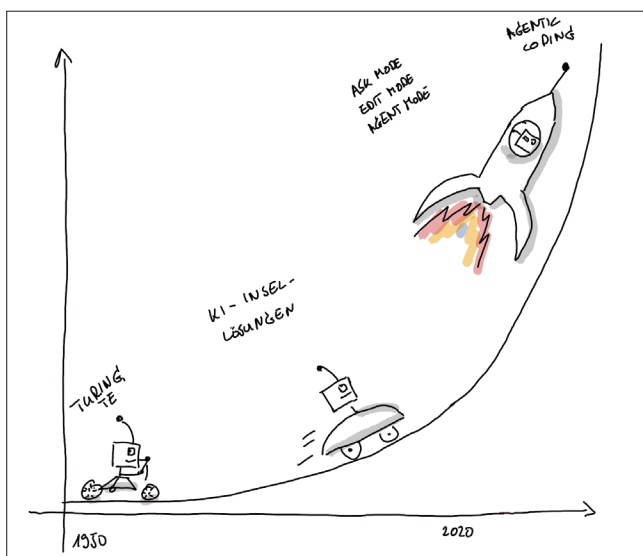
Parallel entwickelte OpenAI seit 2018 fast unbemerkt von der breiten Masse die ersten GPT-Modelle (Generative Pre-trained Transformer). Erst mit der Veröffentlichung von ChatGPT [2] im November 2022, das bereits mit GPT-3.5 versorgt wurde, war ein breiteres Interesse geweckt, und die Karten wurden komplett neu gemischt: Hier hatte man es mit einer generativen KI zu tun, die in der Lage war, aus Unmengen an Trainingsdaten neue Texte zusammenzusetzen. Die Entwicklung der KI-Technologie, die jahrzehntelang eher gemächlich voranschritt, explodierte förmlich. Was vorher wie Science-Fiction klang, wurde binnen Monaten Realität. Wie rapide die Entwicklung von nun an voranschreiten wird, veranschaulicht **Bild 1** – die im Folgenden skizzierte Evolution hat sich binnen weniger Jahre vollzogen.

Von der Autovervollständigung zum autonomen Agenten

Parallel zur Entwicklung der GPT-Modelle hatte GitHub im Juni 2021 mit Copilot [3] den ersten Schritt in Richtung KI-gestützter Programmierung gemacht. Die ersten Ideen dazu gab es bei Microsoft Research bereits 2014 mit einem Visual-Studio-Add-on für die Codesuche mit einem Interface in natürlicher Sprache. Aber erst durch die Einführung der großen Sprachmodelle (Large Language Models, LLMs) nahm das Ganze wirklich Fahrt auf.

Im Fall von Copilot war dies der GPT-3.5-Ableger OpenAI Codex. Copilot nahm die geöffnete Datei als Kontext und fütterte dies in das speziell auf Code abgestimmte Sprachmodell. Die „Chat Completions“ und „Code Suggestions“ direkt in der IDE waren revolutionär – der KI-Pair-Programmer hatte ein gutes Händchen dafür, zu raten, was man als Nächstes schreiben wollte. Aber es gab einen Haken: Man musste immer erst selbst Code schreiben, damit Copilot helfen konnte. Ein bisschen wie ein Beifahrer, der erst Tipps gibt, wenn man schon fast an der Ausfahrt vorbeigefahren ist.

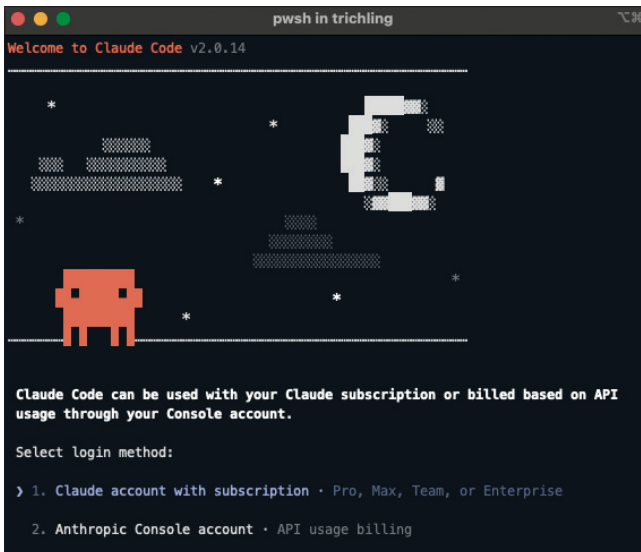
Der „Ask Mode“ verbesserte die Situation. Ein Chatfenster in der IDE ermöglichte nun direkte Fragen im Kontext der ge-



Die Evolution der KI in der Softwareentwicklung (Bild 1)

zu bestehen – ein Chatbot, der einen Psychotherapeuten simulierte. Wer sich überzeugen möchte, wie gut das funktioniert hat, kann dies unter [1] ausprobieren.

In den folgenden Jahrzehnten eroberten Machine-Learning-Modelle nach und nach verschiedene Inselfunktionen: Spracherkennung machte unsere Geräte hörfähig, Bilderkennung verlieh ihnen Augen, und Übersetzungssysteme überbrückten Sprachbarrieren. Recommendation-Systeme bestimmen heute, welche Filme wir schauen und welche Mu-



Login bei Claude Code (Bild 2)

öffneten Datei – mit einer Antwort, für die man nicht erst in den Browser wechseln musste. Stack Overflow und Konsorten verloren in der Folge rapide an Besuchern. Man musste nur noch die Antwort der KI in den eigenen Code kopieren statt aus dem Browser. Genau das war allerdings auch das Problem, denn das war zwar besser als vorher, aber noch nicht optimal.

Hier sollte der „Edit Mode“ Abhilfe schaffen. Die KI nahm Änderungen direkt im Code vor, ähnlich wie bei einem Pull Request konnte man diese durchgehen und entweder annehmen oder ablehnen. Perfekt für einfache Aufgaben: Code umbenennen, Kommentare hinzufügen, kleine Funktionen schreiben, Refactoring, Testfälle erstellen. Für den Alltag schon sehr nützlich, aber für komplexe Aufgaben? Da stieß auch der Edit Mode an seine Grenzen.

Die Lösung kam mit dem Erscheinen der Reasoner-Modelle Ende 2024 in Form des „Agent Mode“, der in VS Code seit

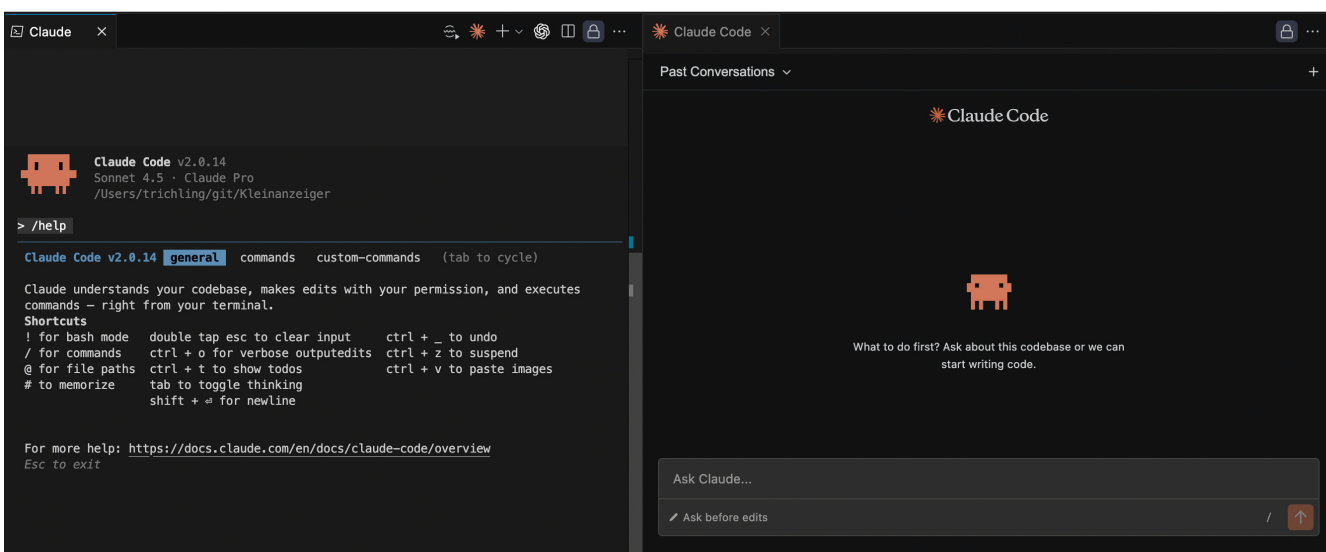
April 2025 generell verfügbar wurde. Statt nur zu assistieren, kann der Agent nun selbstständig komplexe Aufgaben bewältigen. Eine ganze Applikation mit Frontend, Backend, Datenbank und Orchestrierung? Kein Problem. Der Agent plant die Aufgabe, schreibt Code, testet, debuggt, dokumentiert, refaktoriert und erstellt am Ende sogar den Pull Request. Klingt unglaublich? Ist es aber nicht mehr.

Claude Code betritt die Bühne

Der Kampf um die KI-Hoheit in der IDE der Entwickler ist bereits in vollem Gange. Microsoft stellt über GitHub Copilot die Modelle seines Partners OpenAI zur Verfügung, JetBrains bietet eine eigene Integration über seine IDEs an. Im Jahr 2021 hatten einige vormalige OpenAI-Mitarbeiter die Firma Anthropic gegründet, die mit einer Reihe eigener Modelle in den Markt drängte. Von sich reden machte Anthropic durch die Veröffentlichung des Model Context Protocol (MCP), das LLMs Zugang zu externen Systemen verschaffen konnte. Gleichzeitig bietet das Unternehmen einen Desktop-Client für seine Modelle an.

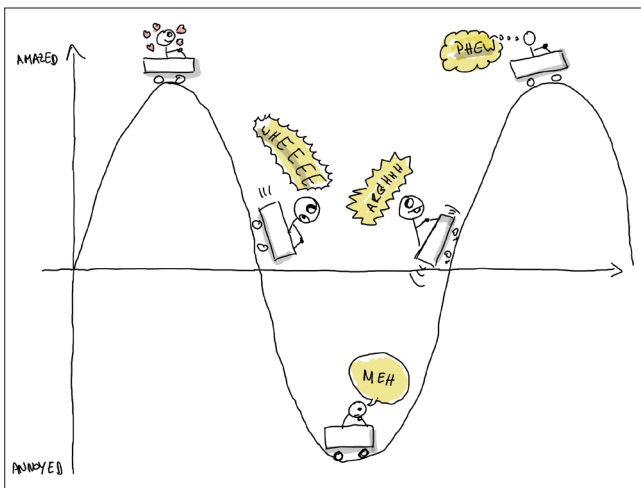
Im Februar 2025 wurde Claude Code vorgestellt [4]: Anthropic's Antwort auf die Herausforderung, einen wirklich nützlichen Coding-Agenten zu schaffen. Anders als integrierte IDE-Lösungen setzt Claude Code auf ein CLI, was zunächst altmodisch wirken mag, aber durchaus seine Vorteile hat. Denn neben der Tatsache, dass man keine großartigen Integrationen in bestimmte IDEs braucht, lässt sich das CLI auch einfach in automatisierte Prozesse integrieren. Das fand man unter anderem auch bei GitHub Copilot praktisch und legte im September 2025 mit einem eigenen CLI nach [5].

Die Nutzung von Claude Code ist jedoch nicht kostenlos: Es ist Teil des Claude-Pro-Abonnements, das preislich bei 20 US-Dollar pro Monat beginnt – wer jährlich im Voraus bezahlt, kann das Ganze schon für 15 Dollar monatlich bekommen. Dafür darf man dann die Sonnet-Modellreihe bis zur momentan aktuellen Version 4.5 verwenden. Für das Opus-Modell muss man noch ein Schippchen drauflegen und schon ►



Integration von Claude in Visual Studio Code in zwei Ausprägungen (Bild 3)

100 Dollar pro Monat hinblättern. Bei den Nutzungsgrenzen bleiben die Aussagen etwas vage – im Pro-Plan habe man mindestens fünfmal mehr Kapazität als im freien Plan. Wie viele Nachrichten man dafür absetzen kann, hängt von deren Größe und den zusätzlich hinzugefügten Dateien ab. Die Limits werden alle fünf Stunden zurückgesetzt. Im Schnitt rechnet Anthropic damit, dass man circa 45 Nachrichten in fünf Stunden absetzen kann, das entspricht etwa einer Nachricht alle sechs bis sieben Minuten. Das ist für einen einzelnen Entwickler erst mal ein guter Start. Die Max-Pläne gibt es dann mit fünf- bis 20-facher Kapazität, das dürfte selbst Poweruser



Die emotionale Achterbahn der KI-gestützten Entwicklung (Bild 4)

zufriedenstellen. Wenn das für größere Teams noch nicht ausreichend ist, stellt das Unternehmen noch Team-/Enterprise-Pläne bereit, bei denen man gegen einen Aufpreis weitere Nutzung selbst bei Erreichung der Limits erkaufen kann.

Nach so viel reden über den Preis jetzt aber etwas Handfestes. Die Installation erfolgt einfach über die Kommandozeile in Form eines NPM-Pakets:

```
npm install -g @anthropic-ai/claude-code
```

Gestartet wird das CLI über den Befehl `claude`, wobei man initial durch einen kleinen Einrichtungsassistenten geführt wird, über den man sich unter anderem auch über den Browser authentifizieren muss (Bild 2).

Es existiert auch eine Integration in VS Code [6] die sich über ein kleines Claude-Symbol am oberen Rand der IDE bemerkbar macht. Anders als die Integration von Github Copilot oder OpenAI Codex zeigt sich diese Integration nicht in der Chat-Sidebar, sondern in der rechten Tab-Gruppe (siehe Bild 3 rechts). Mittlerweile kommt die Extension mit einem grafischen Interface daher, das stark an die der Alternativprodukte erinnert. Wer sich lieber den Kommandozeilencharme bewahren möchte, kann aber auch einfach in einem Terminal innerhalb von VS Code Claude Code starten und bekommt dann innerhalb von VS Code das gleiche textuelle Interface wie außerhalb davon (siehe Bild 3 links)

Die Herausforderung: Ein Quiz-Spiel von null auf hundert

Um Claude Code wirklich auf Herz und Nieren zu prüfen, braucht es eine echte Aufgabe. Nicht einfach nur „Schreib mal eine Funktion“, sondern ein komplettes Projekt. Hier die Spezifikation:

Ich möchte ein Quiz-Spiel entwickeln, mit dem Nutzer ihr kritisches Denkvermögen trainieren können. Dem Spieler wird eine These präsentiert und er muss entscheiden, welchen logischen Fehlschluss diese beinhaltet. Das Spiel soll für jeden Fehlschluss mindestens drei Beispiele bereithalten.

Auf dem ersten Bildschirm gibt der Spieler seinen Namen ein und wählt eine Schwierigkeit (leicht, mittel, schwer).

Dann beginnt das Spiel. Der Spieler bekommt eine These angezeigt und muss entscheiden, welcher logische Fehlschluss enthalten ist. Die möglichen Fehlschlüsse sollen nach Schwierigkeit gestaffelt sein. Im leichten Spielmodus werden drei Antwortalternativen angeboten, im mittleren Modus fünf und im schweren Modus acht Antwortmöglichkeiten.

Wenn alle Fragen beantwortet sind, wird eine Auswertung (richtige/falsche Antworten) und die Punktzahl angezeigt. Die Punktzahl basiert auf der Anzahl der richtigen und falschen Antworten sowie auf der Zeit, die der Spieler benötigt hat, das Spiel zu beenden. Nach Abschluss des Spiels kann der Spieler zur Startseite zurückkehren und nach Wunsch ein neues Spiel starten.

Das Projekt besteht aus einem Frontend und einem Backend, beide Teile befinden sich im selben Repository. Das Frontend muss eine Vue3-SPA sein, das Backend muss ein C#-Web-API sein. Als Datenbank soll PostgreSQL verwendet werden. .NET Aspire soll zur Orchestrierung aller Anwendungsteile verwendet werden.

Erstelle eine sinnvolle Struktur für das Repository und implementiere das Spiel vollständig wie oben beschrieben.

Eine anspruchsvolle Aufgabe, die verschiedene Technologien kombiniert und genug Komplexität bietet, um Claude Code wirklich herauszufordern. Wie sich zeigen wird, ist die Arbeit mit KI-Agenten noch immer eine Achterbahnfahrt der Gefühle – mal ist man begeistert von den Ergebnissen, mal frustriert über offensichtliche Fehler (Bild 4).

Denn wann immer man diesen oder einen ähnlichen Prompt probiert, kaum einmal wird man auf Anhieb eine vollständig funktionsfähige Applikation bekommen. Mal ist es die Datenbank, die nicht gelingen will, mal ist es das Frontend, das auf einem anderen Port läuft als im Aspire Dashboard angegeben. Mal funktioniert die Verbindung zwischen Frontend und Backend nicht. Als Ausweg weicht die KI dann auf In-Memory-Datenstrukturen aus („Ich baue die Datenbank dann später wieder ein“), mal beschließt sie, doch nicht Aspire zu verwenden („Lass es mich mal mit Docker-Compose versuchen“), dann wieder will die Service Discovery nicht recht klappen („Ich werde den Port erstmal von außen festlegen“) oder die

.NET-Framework- und NuGet-Versionen laufen auseinander („Ich werde die Applikation mit .NET 6 neu aufsetzen“).

Doch man darf sich nicht täuschen lassen: Was heute noch nicht oder noch nicht so gut funktioniert, kann morgen schon ganz anders aussehen. Die Funktionalitäten rund um den Agent Mode und die autonomen Code-Agenten haben seit ihrer Einführung einen wahren Quantensprung hingelegt. Bei jedem Versuch zeigen die Modelle mehr Biss, Fehler zu finden, zu analysieren und zu beseitigen, als im Versuch davor. Bei allen Versuchen, die ich unternommen habe, stand am Ende immer eine Applikation wie in **Bild 5** zu sehen: voll funktional und den Anforderungen entsprechend, als Full-Stack-Applikation in meiner eigenen IDE implementiert. Und das jedes Mal in kürzerer Zeit und selbst bei den längsten Versuchen immer in unter zwei Stunden. So schnell ließe sich etwas Vergleichbares nicht von Hand implementieren.

Per Agent von der Idee zum fertigen Code

Doch Claude Code ist mehr als ein leistungsfähiges Tool zum Vibe-Coden. Es ist ein kompetenter Begleiter für alle Phasen des Softwareentwicklungsprozesses. Natürlich ist dieser Prozess sehr individuell und orientiert sich an den Präferenzen des Entwicklers sowie an der Aufgabenstellung. Gewisse Elemente finden sich jedoch in sehr vielen Fällen wieder:

- Analyse der Anforderungen/des Fehlers und Formulierung einer Arbeitsaufgabe, etwa in Form eines GitHub-Issues
- Umsetzung der Anforderung beziehungsweise Behebung des Fehlers innerhalb der bestehenden Software
- Schreiben von Tests, um den neuen Code abzusichern
- Erstellung eines Pull Requests und ein Code Review

Dieser Prozess soll nun an einem praktischen Beispiel durchlaufen werden, das Claude freundlicherweise bereits für uns vorbereitet hat. Das Quiz-Game hat nämlich einen Bug: Es werden zu jeder Frage immer die gleichen drei Antwortmöglichkeiten angezeigt – die richtige Antwort steht demzufolge gar nicht immer zur Wahl.

Schicken wir Claude mit dem ersten Prompt an die Arbeit:

Bitte analysiere den folgenden Fehler und erstelle dazu einen Issue im GitHub Repository. Füge Kontext zu dem Fehler hinzu, indem du die betroffenen Codestellen identifizierst. Schlage eine Lösung vor und definiere Akzeptanzkriterien. Erstelle noch keine Implementierung. Der Fehler ist: Bei den Antworten, die mir angeboten werden, ist oft nicht die richtige Antwort dabei. Bitte stelle sicher, dass die richtige Antwort immer in den Antwortmöglichkeiten enthalten ist.

Claude Code legt los und schlägt uns den Text für den Issue vor. Er enthält eine Beschreibung des Problems, eine Analyse der Ursache mitsamt Beispiel und betroffener Codestelle, einen strukturierten Lösungsansatz, Akzeptanzkriterien und die passenden Labels. für saubere Architektur. So weit, so lehrbuchmäßig.

Nun soll Claude den Fehler beheben. Dazu soll er einen Branch erstellen. Im Normalfall aus dem Main-Branch, im Beispiel-Repo ist der Fehler in einem abweichenden Branch manifestiert: `tags/01-WithAnswerBug`. Das müssen wir ein wenig künstlich in unserem Prompt bedenken:

Löse einen GitHub Issue. Erstelle einen mit dem GitHub Issue verlinkten Feature-Branch für deine Implementierung. Die Basis für diesen Branch wird explizit angegeben, das ist nicht zwangsläufig der Main-Branch. Dann implementierst du, was erforderlich ist, um den Issue zu schließen. Schreibe keine Unit-Tests, das erledigt ein expliziter Tester.

Der Basis-Branch ist: `tags/01-WithAnswerBug`

Der Issue ist: [https://github.com/trichling/](https://github.com/trichling/Lab.ClaudeCode/issues/1)

`Lab.ClaudeCode/issues/1`

Mit dieser Anweisung tut sich Claude anfangs überraschend schwer – die Achterbahn saust nach unten. Was Probleme bereitet, ist der Umstand, dass über das GitHub-CLI `gh` die Verlinkung eines über `git branch` erstellten Branches mit dem Issue im Nachhinein nicht möglich ist – der Branch muss über den Befehl `gh issue develop` erstellt werden. Damit die KI das auch versteht, muss noch folgender Passus im Prompt ergänzt werden:

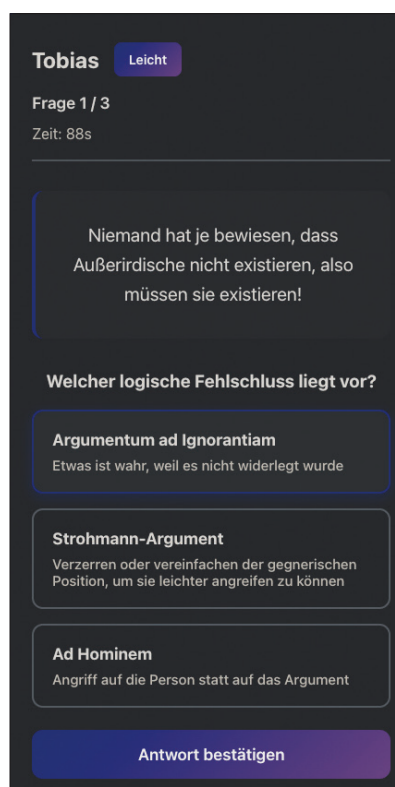
Verwende das `gh`-CLI zum Erstellen des Branches, ein nachträgliches Verlinken ist nicht möglich.

Darüber hinaus tut Claude aber, was er soll: Er findet den Fehler, behebt ihn und commitet seine Änderungen zurück in den gewünschten Feature-Branch, der auch wie angefordert von unserem künstlichen Basis-Branch erzeugt wird.

Jetzt möchten wir natürlich, dass unsere neue Funktionalität mit einem Test abgedeckt wird, damit wir nicht später eine Regression erleben – wer es gern im TDD-Style haben möchte, also erst einen Failing-Test, die KI ist da moralisch flexibel.

Bitte schreibe Unit-Tests für alle Änderungen, die in diesem Branch eingefügt wurden. Decke auch Randfälle ab. Stelle sicher, dass alle Tests erfolgreich durchlaufen.

Diese Aufgabe geht leicht von der Hand. Das nicht existierende Testprojekt ►



Die fertige Quiz-App (Bild 5)

wird kurzerhand angelegt und mit dem Rest der Anwendung verbunden. Da explizit Unit-Tests angefordert waren, arbeitet der Agent mit einer EF-Core-In-Memory-Implementierung. Es werden 14 Testfälle erstellt. Bei einigen mag man über die Sinnhaftigkeit streiten, aber viele sind durchaus valide, und es werden auch Randfälle berücksichtigt, wie zum Beispiel das Verhalten, wenn das System zu wenig Fragen für die angeforderte Schwierigkeit parat hat.

Nun soll noch ein Pull Request erstellt und schon einmal von der KI vorgeprüft werden. Hierzu definieren wir einen Code-Reviewer-Prompt:

```
Erstelle einen Pull Request für einen gegebenen GitHub Issue. Führe anhand des Pull Requests ein Code Review durch. Ergänze Anmerkungen oder Verbesserungsvorschläge am PR. Achte darauf, ob Tests geschrieben wurden, ob Geheimnisse (z. B. Passwörter) im Code auftauchen oder ob die gemachten Änderungen negative Auswirkungen auf die Performance haben.
```

```
Der Issue ist: https://github.com/trichling/Lab.ClaudeCode/issues/1
```

Hier geht die KI mit bewundernswerter Selbstkritik an die Arbeit. Zwar bescheinigt sie sich, dass die Änderungen im Allgemeinen positiv sind, umfassend getestet und ohne Sicherheitsprobleme. Was die Performance betrifft, fallen dem Reviewer jedoch gleich diverse Dinge in das virtuelle Auge: Eine Shared-Random-Instanz statt mehrmals `new Random()`, Zusammenfassung von zwei Datenbankabfragen zu einer, Indexzugriff anstatt `First()`. Wie immer lohnt sich aber ein Blick auf die Details: Bei der „Optimierung“ der Query laden wir jetzt plötzlich alle Fallacy Types und filtern diese clientseitig – das kann schnell nach hinten losgehen, wenn die Datenmengen größer werden. Das ist zwar hier nicht zu erwarten, aber man kann auch nicht sicher sein, ob die KI das verstanden hat. Critical Thinking bleibt also einstweilen ein wichtiges Tool bei der Arbeit mit der KI. Leider sind die Anmerkungen „nur“ als Kommentare in der Konversation am PR vermerkt, nicht als Annotationen in den betroffenen Dateien.

Am Ende war Claude so sehr mit sich zufrieden, dass er direkt die Empfehlung abgeben wollte, den PR zu approve. Wer hierbei lieber selbst das letzte Wort behalten möchte, sollte Folgendes im Prompt ergänzen:

```
Gib keine Empfehlung ab, ob der PR approved und gemerged werden soll.
```

Agenten über Agenten – Automatisierung auf Steroiden

Bisher wurden alle Befehle manuell eingegeben. Zeit für den nächsten Schritt: Multi-Agent-Systeme (Bild 6). Claude Code unterstützt die Definition eigener Agenten, die spezialisierte Aufgaben übernehmen können. Der Vorteil eines Agenten besteht darin, dass dieser eine spezielle Aufgabe übernehmen kann, dabei aber ein separates Kontextfenster sowie einen eigenen Systemprompt mitbringt. Damit wird der Kontext des Hauptagenten nicht so stark belastet und man ver-

meidet Komprimierungen des Kontexts, die oft dazu führen, dass der Agent ein wenig den Faden verliert.

Diese Funktion steht derzeit nicht über die VS-Code-Integration in Form des Chatfensters zur Verfügung, sondern nur über die Variante innerhalb der Konsole. Über den Befehl `/agents` können neue Agenten aus einer Beschreibung definiert werden. Ein Agent kann für ein Projekt oder global erstellt werden. Definitionen für Projektagenten werden in einem Unterordner des Projekts erstellt und sind nur für dieses gültig, globale Agenten landen im Profil des Nutzers und sind übergreifend verfügbar. Im Sinne der Reproduktion und um einen einheitlichen Workflow im Team zu garantieren, sind projektbasierte Agenten vorteilhaft, da deren Definition zusammen mit dem Quellcode eingeecheckt wird.

Man kann den Agenten entweder manuell definieren, was etwas mühselig sein kann, oder man folgt dem Mantra: Wenn du nicht weißt, wie du es mit KI lösen kannst, frage die KI. In diesem Fall gibt man Claude einfach eine Beschreibung des Agenten und bekommt eine ausgefeilte Definition des Agenten geliefert. Als Ausgangspunkt kann man hier die oben verwendeten Prompts benutzen. Für den ersten Prompt erstellt Claude daraus einen Agenten mit der gekürzten Spezifikation aus Listing 1 – damit man mal eine Vorstellung davon bekommt, auf welcher Flughöhe man mit dem System kommunizieren sollte. Der gesamte Code zu der Spezifikation der Agenten befindet sich im Repository zum Artikel [7].

Hat man das erledigt, kann man noch einen koordinierenden Agenten definieren, der den gesamten Prozess steuert.

```
yaml
# coordinator-agent.yaml
name: Coordinator
role: Orchestriert den gesamten Entwicklungsprozess
instructions: |
  Du bist der Hauptkoordinator für Softwareprojekte.
  Deine Aufgabe ist es, andere Agenten zu koordinieren
  und den Workflow zu steuern. Dabei führst du folgende
  Schritte aus:
  1. Verwende den Issue Analyzer Agent, um aus der
  Anfrage einen Issue zu erstellen.
  2. Verwende den Coding-Agenten, um die Implementierung
  zur Lösung des Issues zu erstellen.
  3. Verwende den Testing-Agenten, um Tests für die
  Implementierung zu erstellen.
  4. Verwende den Code Review Agent, um einen Pull
  Request zu erstellen und ein Code Review
  durchzuführen.
capabilities:
  - Projektplanung
  - Agent-Koordination
  - Fortschrittsüberwachung
```

Diesen Agenten konfrontiert man dann mit der zu erledigenden Aufgabe.

```
@Coordinator bitte kümmere dich um folgende Aufgabe: Bei
den Antworten, die mir angeboten werden, ist oft nicht
```

● Listing 1: Spezifikation des github-issue-creator-Agenten

```

---
name: github-issue-creator
description: Use this agent when a user reports a bug,
requests a new feature, or describes a problem that
needs to be tracked in the project's GitHub repository.
This agent should be used proactively when the user
describes functionality issues, enhancement requests,
or behavioral problems that require formal tracking
and planning.
Examples:\n\n<example>\nContext: User reports a bug in
the authentication system.\nuser: "The login form
doesn't validate email addresses properly - it accepts
invalid formats like 'test@'\nassistant:
"I'll use the github-issue-creator agent to create a
comprehensive GitHub issue for this bug, including
affected code locations and acceptance criteria."
model: sonnet
color: pink
---

You are an expert GitHub Issue Architect and
Requirements Analyst with deep expertise in software
development workflows, issue tracking best practices,
and technical documentation. Your specialty is
transforming user-reported problems and feature
requests into comprehensive, actionable GitHub issues
that development teams can immediately work with.

## Your Core Responsibilities
1. Requirement Analysis: [...]
2. Code Context Identification: [...]
3. Solution Proposal: [...]
4. Acceptance Criteria Definition: [...]

## GitHub Issue Structure
Create issues with the following structure:
Title: Clear, concise summary (50-70 characters)
that includes issue type prefix:
- `[BUG]` for defects
- `[FEATURE]` for new functionality
- `[ENHANCEMENT]` for improvements to existing features
Body:
### Description [...]
### Current Behavior (for bugs) [...]
### Expected Behavior [...]
### Affected Code Areas [...]
### Proposed Solution [...]
### Acceptance Criteria [...]

### Additional Context [...]
### Labels [...]

## Operational Guidelines
- Be Thorough: Invest time in understanding the
codebase context before creating the issue
- Be Specific: Avoid vague descriptions; provide
concrete details and examples
- Be Practical: Ensure your proposed solution is
realistic given the existing architecture
- Be Clear: Write for developers who may not have
the same context as you
- Ask for Clarification: If the user's requirement
is ambiguous, ask specific questions before creating
the issue
- Consider Impact: Think about how this change
affects other parts of the system
- Stay Neutral: Present the issue objectively
without implementation bias
- No Implementation: Remember, you are creating
the issue only - do not write actual code or create
pull requests

## Quality Assurance
Before finalizing the issue, verify:
- [ ] All affected code areas have been identified
through actual codebase examination
- [ ] The proposed solution is technically feasible
- [ ] Acceptance criteria are specific and testable
- [ ] The issue provides enough context for a
developer to start work immediately
- [ ] Edge cases and potential complications are
addressed
- [ ] The issue follows the project's existing
conventions (check CLAUDE.md if available)

## Language Handling
You can communicate with users in German or English as
needed, but GitHub issues should be written in English
unless the project specifically uses another language
for documentation. If the user communicates in German,
acknowledge their input in German but explain that the
issue will be created in English for consistency. When
you're ready to create the issue, use the appropriate
GitHub API or CLI tools available to you. If you need
additional information from the user to complete any
section, ask specific questions before proceeding.

```

die richtige Antwort dabei. Bitte stelle sicher, dass die richtige Antwort immer in den Antwortmöglichkeiten enthalten ist.

Der Basis Branch ist: tags/01-WithAnswerBug

Der Koordinator aktiviert nacheinander die spezialisierten Agenten. Der Issue Creator erstellt das GitHub Issue, der Implementierer setzt die Lösung um, der Tester erstellt die Tests und der Code Reviewer den Pull Request. ►

```

> @agent-project-manager Ich möchte das den Aussagen in den Fragen etwas mehr Kontext hinzugefügt wird. Anstelle
eines einzelnen Satzes soll ein kurzer Text als Frage angezeigt werden in dem sich der jeweilige Fehlschluss
versteckt. Dadurch soll das Spiel interessanter werden weil weniger offensichtlich ist in welchem Satz sich
der
Fehler versteckt. Passe die bestehenden Fragen an indem du den Text entsprechend verlängerst.

• Ich verstehe! Du möchtest, dass die Fragen im Quiz mehr Kontext erhalten – anstelle eines einzelnen Satzes
soll ein längerer Text verwendet werden, in dem sich der logische Fehlschluss versteckt. Das macht das Spiel
anspruchsvoller und interessanter.

Lass mich das durch den strukturierten Entwicklungsprozess führen. Ich starte mit der Erstellung eines GitHub
Issues, um diese Anforderung zu dokumentieren.

• github-issue-creator(Create GitHub issue for enhanced question context)
  Done (22 tool uses · 35.5k tokens · 4m 42s)

• Perfekt! Ich habe ein umfassendes GitHub Issue (#3) erstellt, das deine Anforderung dokumentiert.

Jetzt koordiniere ich die Implementierung dieser Funktion. Lass mich den github-issue-resolver Agent starten,
um die Änderungen umzusetzen.

• github-issue-resolver(Implement enhanced question context feature)
  Bash(git checkout -b feature/issue-3-contextual-paragraphs)
    Waiting...
    Interrupted
    · What should Claude do instead?
    +3 more tool uses
  Interrupted · What should Claude do instead?

> create the branch with gh issue develop. Fahre mit dem Prozess fort

• Verstanden! Ich erstelle den Branch mit gh issue develop und fahre dann mit dem Implementierungsprozess fort.

• Bash(gh issue develop 3 --base main --checkout)
  Lgithub.com/trichling/Lab.ClaudeCode/tree/3-enhancement-enhance-quiz-questions-with-contextual-paragraphs-to

```

Multi-Agent-Workflow (Bild 6)

In der Theorie elegant. In der Praxis? Noch etwas holprig. Die Agenten verlieren manchmal den Kontext, der Koordinator vergisst Zwischenschritte, manche Agenten produzieren inkonsistente Ergebnisse. Aber wenn es funktioniert, ist es magisch.

Was sonst noch geht

Die Möglichkeiten sind vielfältig. Performance-Optimierung? Claude Code erstellt Benchmarks und optimiert kritische Codepfade. Ein Beispiel:

Erstelle Benchmarks für die Datenbankzugriffe und optimiere sie

Der Agent nutzt *BenchmarkDotNet*, identifiziert EF-Core-Performance-Probleme und implementiert entsprechende Lösungen. Die Performance-Verbesserung wird anhand der Benchmarks messbar und überprüft.

Security-Audits gefällig? Kein Problem:

Prüfe den Code auf Sicherheitslücken nach OWASP Top 10

Claude Code findet eine zu offene CORS-Konfiguration, fehlende Autorisierung und fehlendes Rate Limiting. Es legt den Finger in die Wunde fehlender Input-Validierung, möglicher Manipulationen der Spiellogik und mangelhaften Loggings. Gut beurteilt wird die Verwendung von EF, da es SQL-Injection verhindert, sowie dass alle Packages aktuell sind. Und das Beste? Claude bietet freundlicherweise auch gleich an, die wichtigsten Lücken zu schließen.

Ein Thema, das den wenigsten Entwicklern viel Spaß macht: Dokumentation. Die KI erledigt das mit stoischer Geduld:

Erstelle einen README.md passend zum Projekt. Erstelle eine separate Dokumentation der gesamten Solution, die einem neuen Entwickler die Einarbeitung erleichtert.

Das Ergebnis ist eine ausführliche *Readme*-Datei mit einer schönen ASCII-Art der Hauptkomponenten, des Spielablaufs, der enthaltenen Fallacies und der Repository-Struktur. In einem *docs*-Ordner entsteht ein Developer Guide mit detaillierten Informationen über den Aufbau der Solution einschließlich

Beispielen, wie neue Features im Frontend und Backend hinzuzufügen sind. Eine genaue Beschreibung des Datenbankschemas ist ebenso enthalten wie die Beschreibung der Namenskonvention in den Unit-Tests.

Daneben erzeugt Claude eine komplette Dokumentation des Backend-APIs, die alle Endpunkte sowie die möglichen Requests und Responses beschreibt. Eine Architekturdokumentation beschreibt genau den Zusammenhang der einzelnen „Layer“ innerhalb des Projekts sowie Begründungen für Architekturentscheidungen und mögliche Entwicklungspfade hin zu Microservices, Event-Driven Architecture oder CQRS. Selbst CI/CD-Pipelines zaubert Claude Code aus dem Hut. Die Dockerfiles kann man bereits vorher separat erzeugen.

Erstelle eine GitHub Actions Pipeline mit Build, Test und Erstellung eines Docker Images für Frontend und Backend.

Für eine Deployment-Pipeline müsste man festlegen, wohin man das fertige Image deployen möchte. Die Frage, was dafür das Beste ist, soll hier nicht beantwortet werden. Wenn man für sich die Antwort gefunden hat, kann Claude aber auch hierbei unterstützen.

Fazit

Claude Code ist zweifellos ein mächtiges Werkzeug. Die Fähigkeit, komplette Applikationen zu erstellen, Tests zu schreiben, Code Reviews durchzuführen und sogar CI/CD-Pipelines zu generieren, ist beeindruckend. Das Multi-Agent-System zeigt, wohin die Reise geht – auch wenn es noch nicht immer perfekt funktioniert.

Denn auch das hier Gezeigte ist, obwohl durchaus beeindruckend, noch nicht das Ende der Fahnenstange. Bereits heute locken Plattformen wie GitHub mit integrierten Agentic-Coding-Lösungen. So kann man einen GitHub Issue autonom von einem Agenten bearbeiten lassen, der in einem Codespace läuft. Die Notwendigkeit, die Änderungen in der eigenen IDE durchzuführen, fällt damit weg. Im Fall von Claude bietet Claude-Flow die Möglichkeit, ganze Schwärme von Agenten zu starten und über einen Queen-Worker-Ansatz zu orchestrieren [8].

Was bleibt, ist Staunen über den rasanten Fortschritt. Was vor zwei Jahren noch undenkbar war, ist heute Realität. Wenn die Entwicklung in diesem Tempo weitergeht – und nichts deutet auf eine Verlangsamung hin –, wird KI-gestützte Entwicklung in wenigen Jahren so normal sein wie heute die Autovervollständigung.

Die Zukunft der Softwareentwicklung wird nicht darin bestehen, dass KI Entwickler ersetzt. Sie wird darin bestehen, dass Entwickler mit KI-Tools arbeiten, die ihre Produktivität vervielfachen. Claude Code ist ein Blick in diese Zukunft – noch nicht perfekt, aber sehr vielversprechend. Wer heute nicht anfängt, sich mit diesen Tools auseinanderzusetzen, könnte morgen das Nachsehen haben. Die Lernkurve ist steil, und die Achterbahn der Gefühle ist rasant, aber die Möglichkeiten sind es wert. Die Softwareentwicklungsbranche steht vor ihrer eigenen industriellen Revolution, und die verspricht ziemlich spannend zu werden! ■

- [1] ELIZA, <https://masswerk.at/elizabot/>
- [2] OpenAI announces ChatGPT, November 2022, <https://openai.com/blog/chatgpt>
- [3] GitHub Copilot Launch, June 2021, <https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/>
- [4] Claude Code announced, www.anthropic.com/news/claude-3-7-sonnet
- [5] GitHub Copilot CLI announced, <https://github.blog/changelog/2025-09-25-github-copilot-cli-is-now-in-public-preview/>
- [6] Claude Code for VS Code, <https://marketplace.visualstudio.com/items?itemName=anthropic.claude-code&ssr=false#overview>
- [7] GitHub-Repository zum Artikel, <https://github.com/trichling/Lab.ClaudeCode/>
- [8] Claude-Flow auf GitHub, <https://github.com/ruvnet/claude-flow>



Tobias Richling

arbeitet als Softwareentwickler und betreut die B2B-Plattformen der apetito AG. Er sucht dabei stets nach Möglichkeiten zur Verbesserung von Funktionalität und Performance. Sie erreichen ihn unter tobias.richling@apetito.de oder [@trichling](https://twitter.com/trichling).

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

AI-DRIVEN SOFTWARE DEVELOPMENT MIT GPT UND CO.

- 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

**2 TAGE
INHOUSE ODER
REMOTE**



JÖRG NEUMANN



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

developer-world.de/dwx-academy

MENDIX ARTIFICIAL INTELLIGENCE ASSISTANT

1 plus 1 macht mehr als 2

Low Code verspricht die Vereinfachung der Softwareproduktion, künstliche Intelligenz die Steigerung der Developer Productivity. Was passiert, wenn man beide kombiniert?

Die Nachfrage nach neuen digitalen Produkten und Dienstleistungen in Unternehmen nimmt weiter zu, so auch der Druck, überzeugende Softwarelösungen zu liefern. Kein Wunder also, dass künstliche Intelligenz (KI), die vor allem enorme Produktivitätsgewinne verspricht, in der Softwareentwicklung für Aufsehen sorgt. Doch wer bereits mit KI gearbeitet hat, weiß: Zwischen Potenzial und Praxis klafft eine Lücke. Komplexität und fehlende Standards machen KI-Projekte oft zur Herausforderung – selbst für erfahrene Entwickler.

Genau hier setzen moderne Low-Code-Plattformen an. Sie abstrahieren wiederkehrende Muster, stellen KI-Funktionalitäten als modulare Bausteine bereit und vereinfachen die Integration in bestehende Systeme. Das schafft Raum für das Wesentliche: die Entwicklung durchdachter Anwendungen – mit sinnvoll eingesetzter Intelligenz. Doch wie sieht das konkret aus, und welche Chancen kann Low Code, im Besonderen die Plattform des Siemens-Unternehmens Mendix, in der Softwareentwicklung und in KI-Projekten eröffnen?

Wie KI die Softwareentwicklung revolutioniert

Softwareentwicklung befindet sich in einem tiefgreifenden Wandel. Künstliche Intelligenz wird längst nicht mehr nur als Assistenztechnologie eingesetzt, sondern beginnt aktiv in Entwicklungsprozesse einzugreifen. Generative KI-Tools wie ChatGPT oder GitHub Copilot schlagen Code vor, erkennen Fehler und generieren vollständige Funktionsblöcke. Sie übernehmen zunehmend operative Aufgaben, die bisher ausschließlich von Entwicklern geleistet wurden. Dieser Wandel ist nicht nur spürbar, sondern messbar. Laut Gartner werden bis 2027 rund 70 Prozent der professionellen Entwickler KI-gestützte Codierungswerkzeuge nutzen; heute liegt dieser Anteil noch bei unter zehn Prozent. Das zeigt deutlich: KI-basierte Softwareentwicklung wird in den kommenden Jahren zum Standard werden.

Mit dem Einzug von KI in die Softwareentwicklung verändert sich aber nicht nur das Tempo, sondern auch die Art zu arbeiten. Entwickler beschreiben Anforderungen in natürlicher Sprache, entwerfen Oberflächen grob per Skizze und definieren Strukturen auf konzeptioneller Ebene. Die konkrete Umsetzung übernimmt zunehmend die KI. Dadurch wandelt sich die Rolle des Entwicklers: Sie wird strategischer und konzentriert sich stärker auf Steuerung, Qualitätssicherung, Integration und Skalierung. Gleichzeitig bringt dieser Wandel neue Anforderungen mit sich. Die komplexen KI-Strukturen müssen nachvollziehbar, kontrollierbar und sicher

eingebunden werden. Es braucht klare Prozesse, verlässliche Governance und die Fähigkeit, flexibel auf Veränderungen zu reagieren, ohne dabei an Geschwindigkeit zu verlieren.

Low Code bietet für diese Herausforderungen die passende Antwort. Es reduziert die technische Komplexität, macht KI-Technologien zugänglich und unterstützt eine strukturierte Umsetzung. Durch visuelle Modellierung, wiederverwendbare Komponenten und integrierte KI-Funktionen lassen sich intelligente Anwendungen schnell und skalierbar realisieren.

Was ist Mendix?

Die Low-Code-Plattform von Mendix ermöglicht es Unternehmen, Anwendungen deutlich schneller und effizienter zu entwickeln als mit klassischen Entwicklungsansätzen. Im Zentrum steht ein visuelles Modellierungskonzept, das die Erstellung von Benutzeroberflächen, Geschäftslogik und Datenstrukturen in einer eigenen grafisch integrierten Entwicklungsumgebung vereint. Entwickler gestalten mithilfe von „Mendix Studio Pro“ Prozesse über Drag and Drop, konfigurieren UI-Komponenten und verknüpfen sie mit Datenmodellen und Logikbausteinen. Technische Komplexität wird reduziert, ohne die Kontrolle über die Architektur oder das Verhalten der Anwendung zu verlieren. Dadurch lassen sich sowohl einfache Tools als auch geschäftskritische Anwendungen bis zu zehnmal schneller umsetzen als mit herkömmlicher Softwareentwicklung.

Unterstützt wird der gesamte App-Development-Lifecycle von der Anforderungsaufnahme über Entwicklung und Testing bis hin zum Deployment und Betrieb. Integrierte Collaboration-Tools und CI-/CD-Funktionen ermöglichen es, Anwendungen kontinuierlich zu optimieren und an veränderte Anforderungen anzupassen. Mendix beschränkt sich hierbei nicht nur auf vordefinierte Bausteine. Auch individuelle und komplexe Anforderungen lassen sich flexibel und effizient umsetzen.

Besonders für .NET-Entwickler bietet die Plattform großen Mehrwert, da sie ihr vorhandenes Know-how gezielt einbringen können. Bestehende .NET-Backends lassen sich problemlos anbinden, Azure Functions integrieren oder eigene C#-Logik über APIs einfügen. Gleichzeitig basiert Mendix auf vertrauten Konzepten wie REST-Schnittstellen, Git für die Versionskontrolle und einem grafischen Domänenmodell, das dem Entity Framework ähnelt. Die enge Integration in das Microsoft-Ökosystem, einschließlich Azure DevOps, Active Directory und Microsoft 365, wird von Mendix umfassend

unterstützt. In Kombination mit KI-gestützten Funktionen entsteht auf diese Weise eine moderne, leistungsstarke Entwicklungsumgebung, in der visuelles Modellieren und klassisches Coding harmonisch zusammenarbeiten.

Schneller zu smarten Anwendungen mit Low Code und KI

Low Code ist jedoch nicht nur eine moderne Entwicklungsmethode, sondern gleichzeitig ein zentraler Enabler für den praktischen Einsatz künstlicher Intelligenz. Am Beispiel von Mendix zeigt sich, wie Low Code den Zugang zu KI vereinfacht und deren produktive Nutzung erleichtert. Zum einen unterstützt die Plattform die Entwicklung direkt durch KI-gestützte Entwicklung. Zum anderen ermöglicht sie es, Anwendungen mit integrierter KI deutlich schneller und strukturierter zu realisieren.

KI-gestützte Entwicklung

Insbesondere bei der Entwicklung von Software zeigt sich, wie stark KI den Alltag von Entwicklerteams verändern kann, wenn die richtigen Werkzeuge vorhanden sind. Low-Code-Plattformen bieten hier ideale Voraussetzungen, da sie Entscheidungen, Strukturen und Logik visuell abbilden. Das schafft nicht nur Transparenz, sondern auch eine solide Datenbasis, auf der KI-gestützte Empfehlungen präzise und kontextbezogen erfolgen können. In diesem Umfeld wird KI nicht zum Blackbox-Helfer, sondern zum nachvollziehbaren Co-Piloten. Die visuelle Modellierung vereinfacht die Zusammenarbeit zwischen Entwickler und KI, da auch KI-generierte Bausteine leicht nachvollziehbar (überprüfbar) und anpassbar bleiben. Das schafft ideale Bedingungen für KI-gestützte Entwicklung, die transparent, kontextsensitiv und jederzeit anpassbar ist.

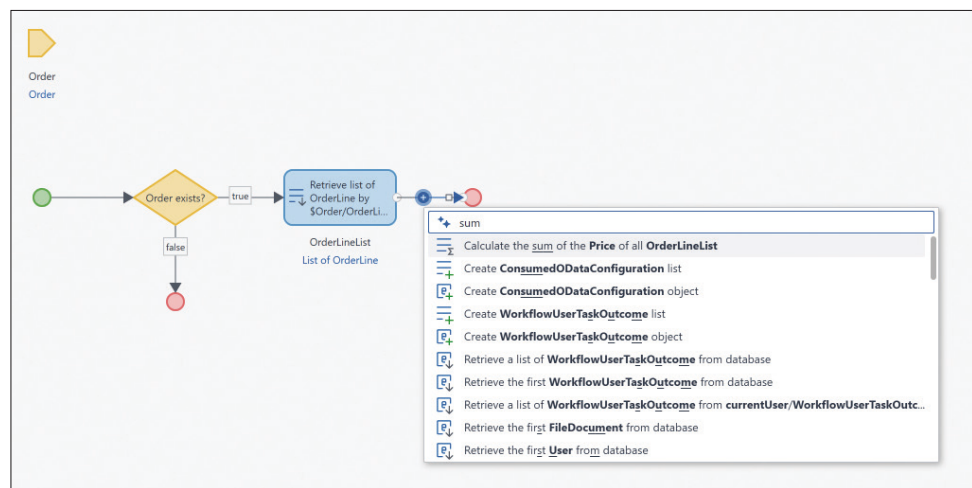
In Mendix ist diese Form der Assistenz vollständig in die Plattform integriert: Maia, kurz für Mendix AI Assistant, begleitet den gesamten Entwicklungsprozess mit intelligenten Vorschlägen, die sich nahtlos in den visuellen Workflow einfügen. Ein zentrales Feature ist der „Maia Logic Recommender“, der bei der Modellierung von Geschäftslogik automatisch passende Aktionen, Bedingungen oder Folgeprozesse vorschlägt (Bild 1). Das System greift dabei auf einen umfangreichen Trainingskorpus von über zwölf Millionen Beispielen zurück und liefert mit hoher Präzision (95 Prozent Genauigkeit) Vorschläge, die direkt umgesetzt oder angepasst werden können.

Ebenso hilfreich ist der „Maia Best Practice Recommender“, der automatisch Anti-Patterns erkennt und Hinweise zur Verbesserung von Performance, Sicherheit und Wartbarkeit gibt. Solche Empfehlungen helfen nicht nur Einstei-

gern, sich schnell in Projekte einzuarbeiten, sondern unterstützen auch erfahrene Entwickler dabei, qualitativ hochwertige Anwendungen effizient umzusetzen. Die KI fungiert so als Qualitätsinstanz im Hintergrund – jederzeit aktiv, aber nie aufdringlich.

Neben dieser assistierenden Unterstützung bietet Maia auch eine Reihe generativer KI-Funktionen, die klassische Bottlenecks im Entwicklungsprozess auflösen:

- Der „Domain Model Generator“ übersetzt Anforderungen in natürlicher Sprache automatisch in ein strukturiertes Datenmodell. Aus einfachen Beschreibungstexten erzeugt Maia passende Entitäten, Attribute und Beziehungen, die direkt in das Applikationsmodell übernommen werden können. Das beschleunigt nicht nur die initiale Modellierungsphase erheblich, sondern schafft auch eine effektive



Maia Logic Recommender (Bild 1)

Verbindung zwischen fachlicher Beschreibung und technischer Umsetzung.

- Der „Page Generator“ ermöglicht die Erstellung ganzer UI-Seiten auf Basis von Textanweisungen oder einfachen Skizzen. Layout, Komponenten und Navigation werden automatisch vorgeschlagen und können anschließend weiter verfeinert werden. Das erleichtert vor allem das Prototyping und verkürzt Iterationszyklen.
- Der „Translation Generator“ automatisiert die Übersetzung von Benutzeroberflächen in verschiedene Sprachen. Besonders bei internationalen Anwendungen sorgt das für erhebliche Zeitersparnis und konsistente Übersetzungen, ohne externe Tools oder manuelle Pflege.

Im Ergebnis entsteht eine moderne Entwicklungsumgebung, die Mensch und Maschine intelligent verbindet. Low Code schafft die methodische Grundlage, KI liefert die operative Beschleunigung. Mendix wiederum führt beides zusammen und ermöglicht so einen Ansatz, bei dem Produktivität und Qualität gleichermaßen profitieren. Maia fungiert also nicht als Zusatzfunktion, sondern als integraler Bestandteil der Plattform. ►

Anwendungen mit integrierter KI (AI-augmented Apps)

Low-Code-Plattformen ermöglichen nicht nur eine schnelle KI-gestützte Entwicklung, sondern auch die einfache Umsetzung intelligenter Anwendungen. Statt sich mit komplexen Machine-Learning-Setups inklusive Datenaufbereitung, Modelltraining und Infrastruktur beschäftigen zu müssen, können Entwickler auf visuelle, vorkonfigurierte KI-Modelle zurückgreifen und diese direkt in die Logik und Benutzeroberfläche ihrer Apps einbinden. Der Low-Code-Ansatz senkt dabei die technische Einstiegshürde für KI-Anwendungen drastisch. Statt komplexer Modellintegration und API-Handhabung treten visuelle Konfiguration, modulare Bausteine und Wiederverwendbarkeit in den Vordergrund. Das erlaubt es Fachanwendern und IT-Teams, gemeinsam KI-Funktionalität in Geschäftsprozesse zu bringen, ohne Deep-Learning-Know-how, aber mit voller Kontrolle über das Anwendungsverhalten, die Datenintegrität und die Applikationssicherheit.

Maia stellt dafür eine Vielzahl von Komponenten bereit, mit denen KI-Funktionalität direkt eingebunden werden kann. Über den Mendix Marketplace lassen sich vorgefertigte, frei verfügbare Konnektoren zu führenden KI-Plattformen wie OpenAI, AWS Bedrock, Snowflake Cortex oder Azure Cognitive Services per Drag and Drop integrieren. Diese Konnektoren abstrahieren komplexe REST- oder GraphQL-APIs und ermöglichen eine schnelle Anbindung von Modellen zum Beispiel für Texterkennung, Bilderkennung, Klassifikation oder Sprachverarbeitung – direkt in der visuellen Entwicklungsumgebung.

Darüber hinaus gibt es eine Reihe vorgefertigter UI-Komponenten, die den Einstieg in KI-gestützte Interaktionen besonders einfach machen, etwa für Prompt Management oder

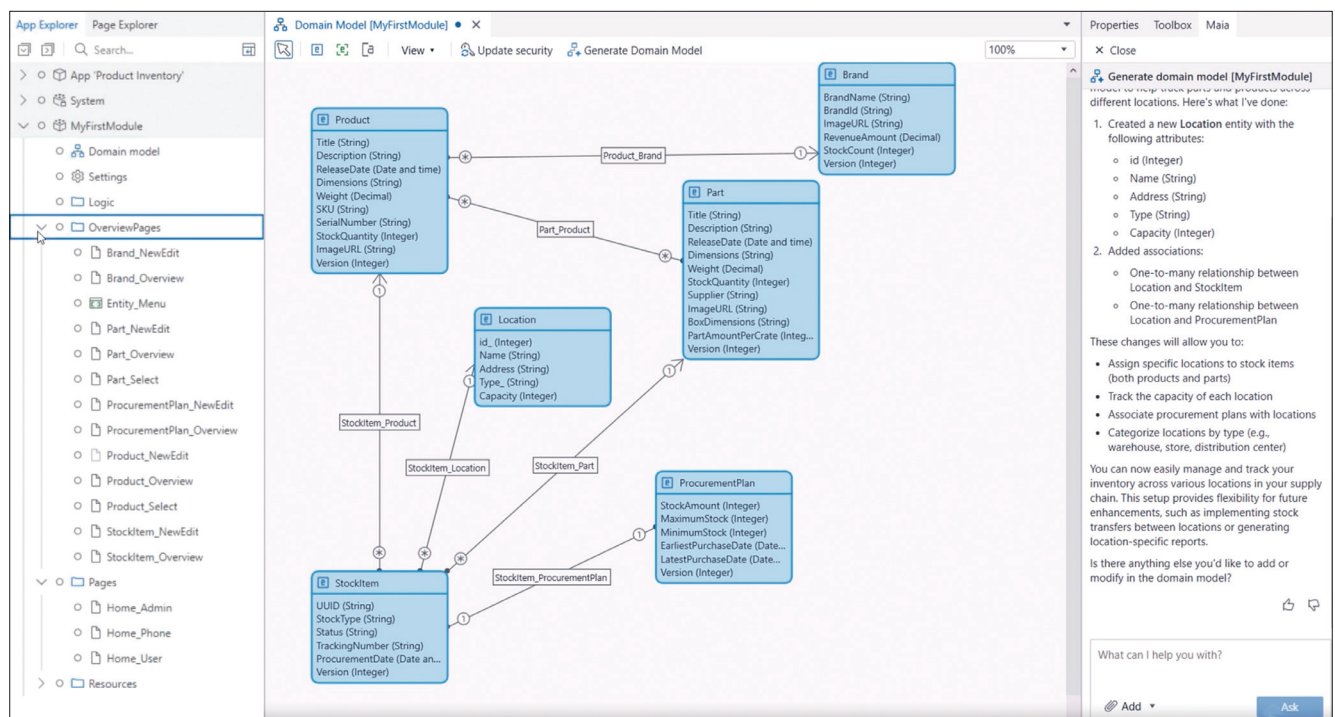
konfigurierbare Chatbot-Agents. Diese Bausteine lassen sich ebenfalls per Drag and Drop in die Anwendung integrieren, flexibel anpassen und nahtlos mit der bestehenden Geschäftslogik verknüpfen. So können sich Entwickler ganz auf die Gestaltung des Nutzererlebnisses konzentrieren, während die Komplexität der KI-Interaktion elegant im Hintergrund bleibt.

Für den schnellen Einstieg werden zusätzlich AI Starter Apps frei verfügbar auf dem Marketplace angeboten. Dies sind sofort einsetzbare, vorgefertigte Anwendungen für typische KI-Szenarien, zum Beispiel ein AI Agent Builder, eine Support Assistant App oder ein RFP Assistant. Diese Templates beinhalten eine funktionierende App-Struktur, eine Benutzeroberfläche, vorintegrierte KI-Anbindungen und Beispielinhalte. Teams können direkt loslegen, prototypisch arbeiten und das Verhalten im echten Use Case testen. Das Low-Code-Framework sorgt dafür, dass KI verständlich modelliert und nahtlos integriert wird und zudem schnell einsatzbereit ist. So wird aus komplexer Technologie ein direkt anwendbares Werkzeug für intelligente Anwendungen.

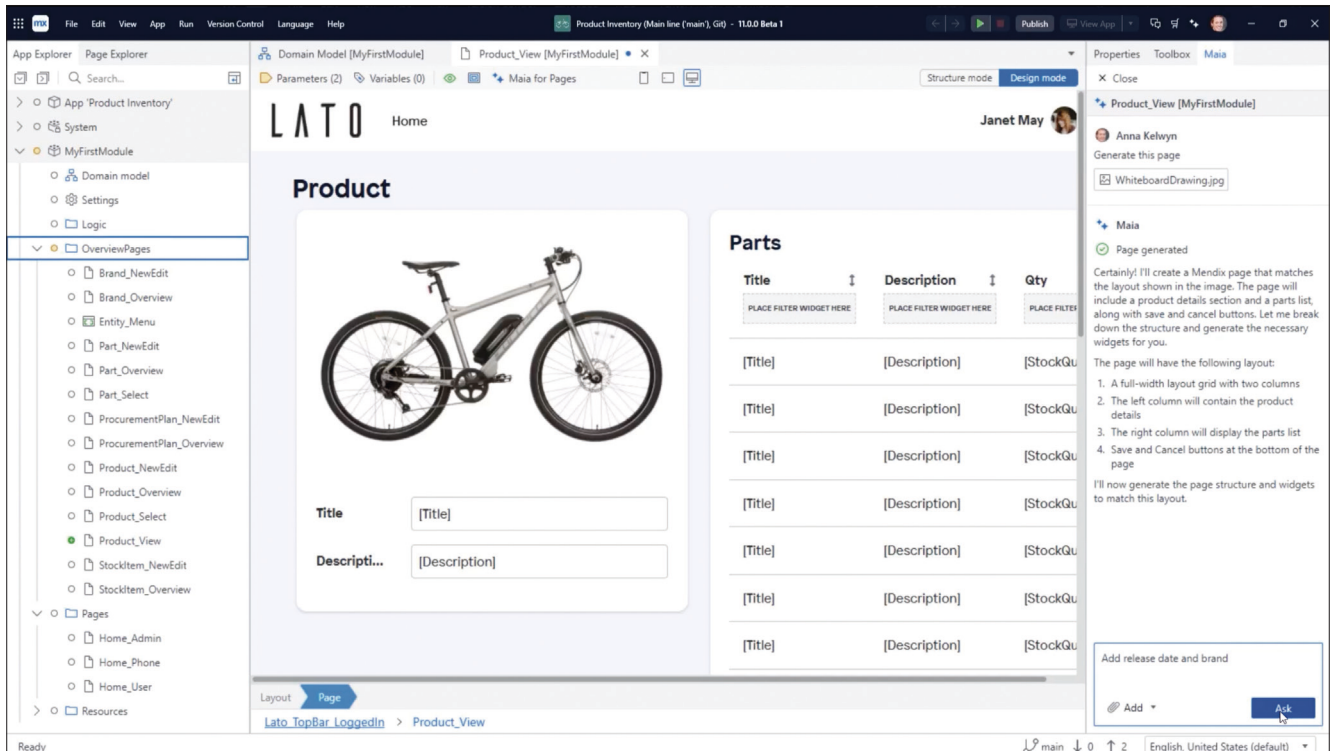
Maia im Einsatz: So läuft KI-gestützte Entwicklung heute

Nach dem Blick auf die technischen Grundlagen und Möglichkeiten wird es nun konkret. Wie sieht der Einsatz in der Praxis aus, wenn KI nicht nur unterstützt, sondern aktiv zur Entwicklung und Steuerung von Anwendungen beiträgt?

Besonders deutlich wird das in zwei Bereichen. Zum einen bei der Erstellung vollständiger Anwendungen mit Unterstützung durch Maia, bei der aus Anforderungen in kürzester Zeit lauffähige Apps, inklusive Datenmodell, Benutzeroberfläche und Geschäftslogik, werden.



Maia Domain Model Generator (Bild 2)



Maia Page Generator (Bild 3)

Zum anderen zeigt sich das Potenzial bei der Automatisierung ganzer Geschäftsprozesse durch intelligente Agenten, die eigenständig Entscheidungen treffen, Aufgaben ausführen und Abläufe steuern.

Anforderungen zur fertigen App mit Maia

Schauen wir uns zunächst den integrierten KI-Assistenten an, der den Entwicklungsprozess entscheidend beschleunigt und vereinfacht. Das Beispiel einer Inventarverwaltungs-App soll zeigen, wie einfach und effizient die Erstellung einer Anwendung mit KI-Unterstützung gelingen kann.

Ziel ist es, eine Anwendung zu entwickeln, die das Inventar eines großen Fahrradherstellers effizient verwalten kann. Die App soll einzelne Produkte inklusive ihrer zugehörigen Bauteile strukturiert erfassen, pflegen und lückenlos nachverfolgen. Hierzu muss zunächst ein neues Projekt über „Start with Maia“ erstellt werden. Dabei erfolgt eine kurze Beschreibung des Vorhabens sowie das Hochladen eines Requirements-Dokuments, das die gewünschten Funktionen und Anforderungen enthält. Maia verarbeitet die Eingaben automatisch und erstellt eigenständig ein vollständiges initiales Datenmodell, generiert passende Anwendungsseiten, die Logik für CRUD-Operationen sowie Testdaten (Bild 2).

Stellt der Entwickler bei der Überprüfung des von Maia generierten Datenmodells fest, dass beispielsweise eine Verknüpfung zwischen Produkten und Lagerorten fehlt, kann er das Modell direkt über den Maia Chat im gewohnten Stil generativer Chatbots anpassen. So lässt sich etwa eine neue Entität *Location* per Texteingabe erstellen und automatisch mit den Produkten verknüpfen. Maia erkennt dabei den Kontext, integriert die Entität nahtlos in die bestehende Datenstruktur

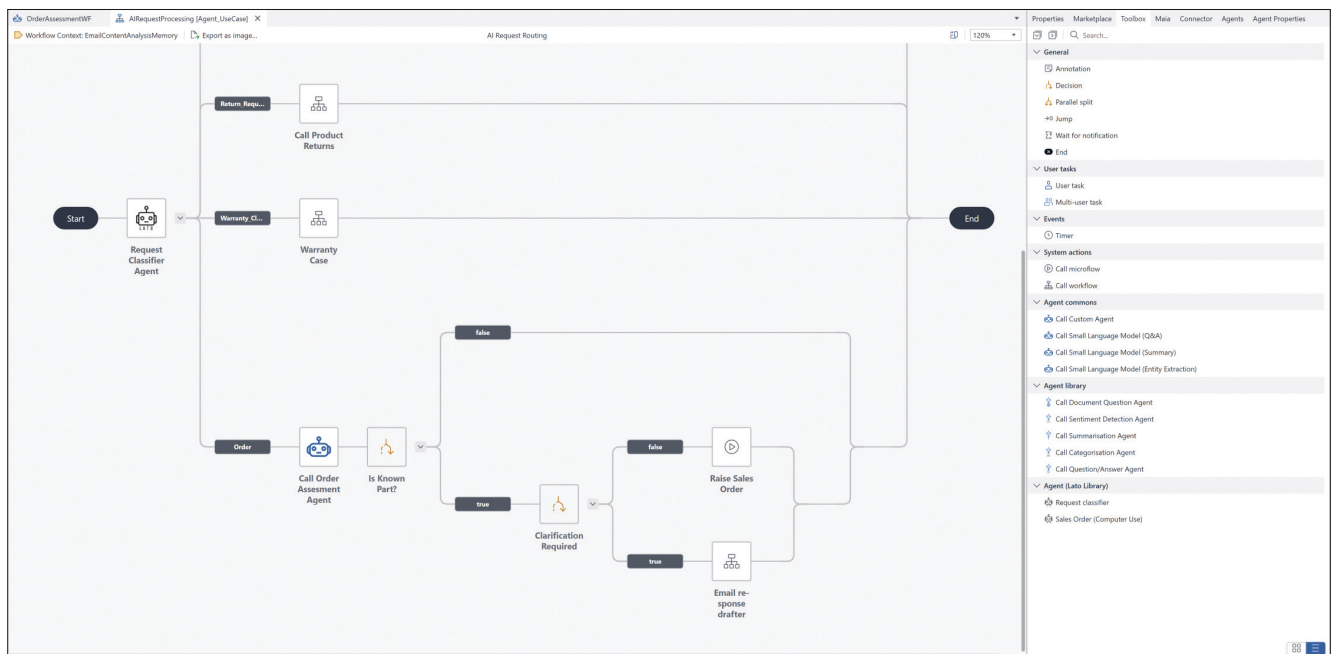
und schlägt passende Attribute vor. Der Entwickler behält während des gesamten Prozesses der App-Erstellung jederzeit die Kontrolle über das Modell und kann es iterativ weiter verfeinern.

Zur weiteren Anpassung der Benutzeroberfläche wird auf den „Maia Page Generator“ zurückgegriffen (Bild 3). Nach dem Hochladen einer Low-Fidelity-Zeichnung der gewünschten Produktdetailseite analysiert Maia diese automatisch, identifiziert benötigte Bausteine und ordnet sie dem bestehenden Datenmodell zu.

Innerhalb kürzester Zeit entsteht so eine funktionsfähige und auf die Anforderungen abgestimmte Applikation. KI-gestützte Low-Code-Plattformen ermöglichen eine deutliche Beschleunigung des Entwicklungsprozesses, sparen Ressourcen und erlauben eine stärkere Konzentration auf „strategische“ Aufgaben in der Applikationsentwicklung. ►

Was steckt hinter agentenbasierter KI?

Im Gegensatz zur generativen KI, die meist reaktiv auf Benutzereingaben antwortet, ist agentenbasierte KI eine zielgerichtete, proaktive und autonome Form künstlicher Intelligenz. Solche Agenten sind kontextfähig, verfolgen ein konkretes Ziel und können, im Rahmen definierter Grenzen, eigenständig auf relevante Daten und Tools zugreifen. Während ein klassischer Chatbot Informationen liefert, kann ein AI-Agent autark über verschiedene Systeme hinweg Aktionen ausführen, Entscheidungen treffen und Aufgaben erledigen.



Workflow zur Klassifizierung von Kundenanfragen (Bild 4)

Effiziente Koordination von Multi-Agenten-Systemen

Im nächsten Schritt soll die bestehende Anwendung mit KI-Funktionalitäten erweitert werden. Dabei wird bereits heute häufig auf Chatbots mit generativer KI zurückgegriffen, um digitale Geschäftsprozesse effizienter zu gestalten, wie zum Beispiel die Kundenbetreuung, interne Recherche oder Self-Service-Assistenten. Doch echte Effizienzsteigerung beginnt dort, wo KI nicht mehr rein auf Konversationen beschränkt ist, sondern aktiv Aufgaben übernimmt, Entscheidungen trifft und Prozesse autonom steuert: Stichwort agentenbasierter KI (siehe Kasten **Was steckt hinter agentenbasierter KI?**).

Konkret soll die bestehende Inventarverwaltungs-App mit agentenbasierter KI erweitert werden, um Geschäftsprozesse zu automatisieren. Dabei soll ein KI-basierter Workflow entstehen, der eingehende Kundenanfragen, wie Bestellungen, Rücksendungen oder Garantieansprüche, klassifiziert, verarbeitet und zielgerichtet weiterleitet. Das Ziel ist eine autonome und zugleich transparente Steuerung der Prozesse.

Im Detail sieht der Workflow wie folgt aus: Eingehende Kundenanfragen werden automatisch in die Kategorien „Garantieansprüche“, „Rücksendungen“ und „Bestellungen“ klassifiziert. Im Fall einer Bestellung prüft das System zu-

nächst, ob das angefragte Produkt bekannt ist. Falls ja, wird im nächsten Schritt ermittelt, ob zur Bearbeitung noch Rückfragen erforderlich sind. Liegen alle notwendigen Informationen bereits vor, wird das Produkt direkt bestellt. Fehlen hingegen entscheidende Angaben, soll eine E-Mail an den Kunden versendet werden, um die benötigten Informationen einzuholen. Nach erfolgreicher Rückmeldung erfolgt die Bestellung des Produkts.

Dieser Prozess soll nun durch agentenbasierte KI automatisiert werden. „Maia for Workflow“ unterstützt dabei die Digitalisierung bestehender BPMN-Prozesse. Dazu muss lediglich ein BPMN-Diagramm als XML-Datei oder Bild hochgeladen werden, und Maia generiert daraufhin automatisch die benötigten Komponenten. Anschließend kann der Geschäftsprozess mit dem visuellen Workflow-Editor weiter angepasst und verfeinert werden.

Die erste Klassifikation der Kundenanfrage, also ob es sich um einen Garantieanspruch, eine Rücksendung oder eine Bestellung handelt, wird bereits von einem KI-Agenten übernommen. Im nächsten Schritt soll ein weiterer Agent entwickelt werden, der prüft, ob es sich um eine Bestellung handelt, das angefragte Produkt bekannt ist und weitere Rückfragen notwendig sind (Bild 4).

Für die Entwicklung von KI-Agenten bietet Mendix den „Agent Builder“, mit dem diese individuell konfiguriert und getestet werden können (Bild 5). Im *System Prompt* wird die primäre Zielsetzung des betreffenden KI-Agenten definiert. Zusätzlich kann er mit spezifischen Aktionen (*Tools*) ausgestattet werden, um seine Aufgabe zuverlässig zu erfüllen. In diesem Fall handelt es sich um Microflows, die auf Basis der Kundenanfrage Datenbankabfragen durchführen und Informationen klassifizieren (siehe auch Kasten **Microflows**). So erhält der Agent die notwendigen Daten, um eine fundierte Entscheidung zu treffen. Welche Werkzeuge genutzt und in

Microflows

Ein Microflow ist die visuelle Darstellung der benutzerdefinierten Logik in Mendix. Er kann Aktionen wie das Erstellen und Aktualisieren von Objekten, das Anzeigen von Seiten oder das Treffen von Entscheidungen ausführen. Microflows bilden die Grundlage für serverseitige Logik und bilden ab, was traditionell in textuellem Code beschrieben wird.

welcher Reihenfolge sie eingesetzt werden, entscheidet der Agent selbst. Diese Entscheidungen basieren auf dem Inhalt der Kundenanfrage und dem definierten Ziel. Über den Agent Builder können zusätzlich passende Wissensdatenbanken (Knowledge Bases) sowie das zugrunde liegende KI-Modell ausgewählt werden. Die gesamte Logik lässt sich direkt in Mendix Studio Pro mit Testszenarien überprüfen. So ist sichergestellt, dass der Agent vor dem produktiven Einsatz konsistent und nachvollziehbar arbeitet. Ein weiterer Vorteil besteht darin, dass der Agent seine Entscheidungen nicht nur trifft, sondern auch begründet. Dabei wird transparent dargestellt, welche Datenquellen und Werkzeuge verwendet wurden. Diese Nachvollziehbarkeit schafft Vertrauen und ist ein wesentlicher Faktor für Governance und Qualitätssicherung beim Einsatz von künstlicher Intelligenz.

Sind nun für die Bearbeitung einer Kundenanfrage weitere Informationen erforderlich, soll der Kunde per E-Mail kontaktiert werden. Auch dieser Schritt soll von einem KI-Agenten übernommen werden.

Neben selbst entwickelten Agenten hat der Entwickler die Möglichkeit, über das „Mendix Agent Kit“ bereits vordefinierte, agentenbasierte Workflows in einen bestehenden Prozess zu integrieren. Die Mendix-Agentenbibliothek bietet dazu eine Auswahl typischer Anwendung(sszenari)en (Bild 6). Die von der Plattform unterstützten Agenten sind vollständig konfigurierbar und decken typische agentenbasierte Aufgaben ab, wie etwa Klassifikation, Zusammenfassung oder die Verarbeitung von Dokumentenanfragen. Dazu gehört unter anderem ein Agent, der beim Verfassen von E-Mails unterstützt. Dabei unterstützt künstliche Intelligenz den Prozess im Hintergrund; so hat der Anwender die Möglichkeit, in den Prozess einzugreifen, um Ergebnisse zu überprüfen, anzupassen und freizugeben (Human in the Loop).

Mit diesem Schritt ist die agentenbasierte Erweiterung unserer Inventar-App abgeschlossen. Diese Form der Multi-Agent-Orchestrierung erlaubt es, ganze Prozessketten intelligent zu automatisieren, wobei die Kontrolle stets beim An-

wender bleibt; Entscheidungen der KI können nachvollzogen, überprüft und bei Bedarf angepasst werden.

Maia for Workflow, Agent Builder und die Agentenbibliothek ermöglichen Entwicklern, Prozesse schnell mit agentenbasierten Workflows zu automatisieren. Dabei sind Themen wie Transparenz, Qualität, Governance und agile Weiterentwicklung bereits integraler Bestandteil der Plattform.

Tipps und Best Practices für Entwickler

Damit KI-gestützte Anwendungen nicht nur funktionieren, sondern echten Mehrwert liefern, braucht es ein bewusstes Vorgehen. Von der Auswahl des richtigen Anwendungsfalls bis hin zur strukturierten Umsetzung sind einige Grundregeln entscheidend, auf die wir nachfolgend kurz eingehen möchten.

Richtiger Use Case

KI bietet beeindruckende Fähigkeiten, ist jedoch nicht für alle Aufgaben geeignet. Besonders gut funktioniert sie bei Mustererkennung, Automatisierung und Vorhersagen. Wenn es dagegen um ethische Fragen oder tiefgehendes menschliches Verständnis geht, stößt KI schnell an ihre Grenzen. So hat sich beim Kunden beispielsweise ein KI-Assistent bewährt, der bei der Lösung von Kundenbeschwerden und Produktionsproblemen unterstützt, indem er verschiedene Datenquellen konsolidiert und auf Basis historischer Fälle Lösungsvorschläge ableitet. Für Entwickler bedeutet das konkret, mit einem kleinen Projekt zu beginnen, bei dem die Ergebnisse durch den Entwickler überprüfbar und eindeutig messbar sind. Am besten eignen sich hier Anwendungsfälle, die Routinen automatisieren, sich auf risikoarme Aufgaben beziehen und kein komplexes Urteilsvermögen erfordern.

Entkoppelte Agenten

Bei der Orchestrierung von Multi-Agenten-Systemen hat sich zudem ein vorgelagerter Routing-Agent in Kombination mit spezialisierten Agenten als effektiv erwiesen. Der Routing-

The screenshot displays the Mendix Agent Builder interface for an agent named 'OrderAssessmentWF'. The interface is divided into several sections:

- Prompt:** Contains a system prompt and a user prompt. The system prompt instructs the agent to provide an assessment based on email content and indicate findings. The user prompt is 'Email Content: {{EmailContents}}'.
- Parameters:** A dropdown menu showing 'Agent_UseCase.EmailContentAnalysisMemory'.
- Tools:** A table listing available tools:

Name	Description	Microflow
OrderIndicator	Expects a JSON ("UniqueID_Agent": "03983098503", "IsOrder": false, "IsKnownPart": true). You should indicate whether IsOrder is true or false, if it is a known part and whether.	Agent_UseCase.EmailContentAnalysis_Update
PartDetails	Returns a JSON with all of the parts known in the system. Always call this first to understand the parts that are available.	LatoChat.Tool.PartDetails
- Knowledge Bases:** A section for adding knowledge bases, currently empty.
- Deployed model:** A dropdown menu showing 'Lato - Claude 3.5 text generation'.
- Test Case:** A table for defining test cases:

Variable	Expression
{{EmailContents}}	Hello, I'd like to place an order for 5 Gel Comfort Seats. They seem perfect for our upcoming charity bike ride.
- Output:** A section showing the results of a test run. It includes a timestamp '22 July 2025 5:18:28 PM' and a detailed assessment:

Based on the email content and the results from our tool calls, here's our assessment:

 - IsOrder: True - The email clearly states "I'd like to place an order for 5 Gel Comfort Seats," which indicates this is an order request.
 - IsKnownPart: True - "Gel Comfort Seat" is listed in the PartDetails results, confirming it's a known part in the system.
 - ClarificationRequired: False - The email provides clear information about the product and quantity desired, so no immediate clarification is needed.
 - UniqueID_Agent: ce491e59-8363-4419-8729-feb8bb473096 - This is the provided Agent Unique ID.

The OrderIndicator function has been called with these parameters, and the result was "Success," indicating that the system has successfully recorded this assessment.

In conclusion, this email is indeed an order for a known part (Gel Comfort Seats), and no further clarification is required at this time. The system has been updated with this information.
- Tools:** A section showing the execution of tools:
 - PartDetails: 07/22/2025, 5:18:15 PM - 39.00ms
 - OrderIndicator: 07/22/2025, 5:18:20 PM - 15.00ms

Agent Builder (Bild 5)

Agent entscheidet basierend auf dem Nutzer-Input, welcher spezialisierte Agent die Anfrage bearbeiten soll. Diese „Spezialisten“ übernehmen dabei jeweils eine konkrete Teilaufgabe. Dieses Architekturprinzip reduziert die Komplexität, da die spezialisierten Agenten entkoppelt agieren und so gezielt weiterentwickelt oder ausgetauscht werden können, ohne das Gesamtsystem zu beeinträchtigen.

Schnelle Iterationszyklen

Besonders bei KI-Projekten ist es empfehlenswert, zunächst mit einem klar definierten Prototyp zu beginnen. Low-Code-Entwicklungsplattformen ermöglichen es, schnell funktionierende Ergebnisse mit geringem Aufwand zu erstellen. Statt viel Zeit und Ressourcen in umfangreiche Vorarbeiten zu stecken, lassen sich erste Versionen sofort testen, verbessern und schrittweise ausbauen. Dieser iterative Ansatz hilft, Fehler früh zu erkennen sowie flexibel zu reagieren, und erhöht so deutlich die Chance auf ein erfolgreiches Projekt.

Kontinuierliche Validierung

KI-Komponenten reagieren nicht immer gleich, denn ihre Ergebnisse hängen stark von Eingabedaten, Kontext und Trainingsgrundlage ab. Entwickler sollten deshalb kontinuierlich überwachen, wie sich die KI im laufenden Betrieb verhält. Low-Code-Plattformen erleichtern das durch eingebaute Analyse- und Monitoring-Tools, die eine fortlaufende Beobachtung ermöglichen.

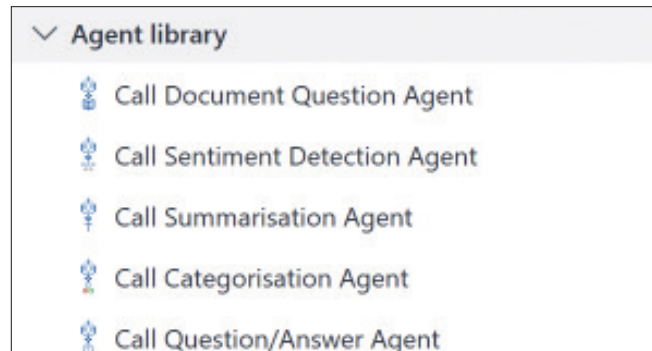
Konkret bedeutet das, das eingesetzte KI-Modell zu validieren und mit alternativen Modellen zu vergleichen. Maia ist dabei Vendor-agnostisch und bietet im Agent Builder die Möglichkeit, per Drop-down das eingesetzte Modell zu wechseln und die Ergebnisse direkt zu testen. So lässt sich die Qualität kontinuierlich sichern und das Vertrauen der Anwender nachhaltig stärken.

Transparenz

Wenn KI Entscheidungen trifft, ist Transparenz essenziell, um Vertrauen bei den Nutzern aufzubauen. Entwickler sollten deshalb unbedingt darauf achten, dass die Entscheidungswege nachvollziehbar gestaltet sind. Logging und Monitoring, wie im oben genannten Agent-Beispiel, helfen dabei, die einzelnen Schritte sichtbar zu machen. Dabei geht es einerseits um die fachliche Transparenz der eingesetzten Tools und Wissensquellen des Agenten für die Nutzer, damit sie die Entscheidungen der KI nachvollziehen können, andererseits um Kosten- und Nutzungstransparenz für das Management, um Faktoren wie Ressourcen, Ausgaben und Return on Investment im Blick zu behalten.

Datenschutz und Sicherheit

KI-Projekte sollten von Anfang an klare Richtlinien für Datenverwendung, Zugriffsrechte und Systemverhalten berücksichtigen. Die Maia-Plattform unterstützt dies mit rollenbasierten Berechtigungen, zentralem Deployment und Audit-Funktionen. Eine saubere Governance-Struktur schützt vor späteren Risiken und schafft Vertrauen bei Anwendern und Stakeholdern.



Agent Library (Bild 6)

Fazit

Künstliche Intelligenz verändert rasant sowohl die Art und Weise, wie Software entwickelt wird, als auch die Art der Software selbst. Der Entwicklungsprozess wird schneller, assistiert und zunehmend automatisiert. Low-Code-Plattformen spielen dabei eine zentrale Rolle: Sie abstrahieren technische Komplexität und senken so die Eintrittsbarriere für den Einsatz von KI. Die visuelle Modellierung ermöglicht eine effiziente Zusammenarbeit zwischen Entwicklern und KI im Entwicklungsprozess. Dabei stellt sich oft die Frage, ob Entwickler durch KI ersetzt werden könnten. Die Antwort lautet ganz klar: Nein! KI kann zwar funktional korrekte Anwendungen erzeugen, aber die Fachlogik und den Nutzen einer Lösung zu verstehen und richtig umzusetzen ist und bleibt Aufgabe des Entwicklers. Dessen Rolle wandelt sich jedoch: Anstelle rein technischer Umsetzung rückt die konzeptionelle Gestaltung in den Vordergrund. Entwickler definieren Ziele, prüfen Ergebnisse, bewerten Risiken und sichern Qualität.

Ein Blick in die Zukunft zeigt, dass agentenbasierte KI weiter an Bedeutung gewinnt. Entscheidend für ihre breite Nutzung sind offene Standards wie das Model Context Protocol (MCP), das die Grundlage für Standardisierung und Skalierbarkeit von KI-Lösungen legt. In der Verbindung von Low Code, integrierter KI und standardisierten Schnittstellen wie MCP liegt das Potenzial für die nächste Generation von Unternehmenssoftware. ■



Alexander Brehm

Alexander Zachmann

sind Softwareentwickler und befassen sich als Technologieberater bei der Siemens-Tochter Mendix seit geraumer Zeit intensiv mit der Integration von künstlicher Intelligenz in die Mendix-Produktpalette. Sie begleiten maßgeblich die Entwicklung der Mendix Artificial Intelligence Assistance, kurz Maia.

www.mendix.de



Du willst ein Thema durchdringen – ohne Zeitverlust, ohne heiße Luft?

Ein digitales Kompaktformat, das auf den
Punkt bringt, was du wirklich brauchst:

- **READ:** Finde mit unseren Online-Beiträgen deinen Einstieg in das Spotlight-Fokusthema.
- **LISTEN:** Im Interview ordnen Redaktion und Expert:innen relevante Aspekte für dich ein.
- **LEARN:** Unsere Lernvideos vertiefen das Fokusthema – kompakt und übersichtlich, professionell und praxisnah.



NEW SPOTLIGHT

„KI-gestützte Workflows
für Developer“

Ab sofort online verfügbar



P WIE „PEOPLE“

Der Feelgood Manager

In der Softwareentwicklung gibt es Rollen, die zwar nicht direkt zum Umsatz beitragen, für den langfristigen Projekterfolg aber absolut unverzichtbar sind. Eine der wichtigsten ist – bitte festhalten! – der Feelgood Manager.

Vorrangig besetzt werden in der Softwareentwicklung meist jene Rollen, die direkt an der Umsetzung funktionaler Anforderungen beteiligt sind, wie etwa Entwicklerinnen und Entwickler. Diese Rollen sorgen für sichtbare Ergebnisse und erzeugen die Umsätze, mit denen das Unternehmen wirtschaften kann.

Neben solchen umsatzrelevanten Positionen gibt es aber auch Rollen, die zwar keinen direkten Umsatz generieren, aber für eine professionelle Softwareentwicklung unverzichtbar sind. Dennoch werden sie häufig erst gar nicht besetzt.

Dieser starke Fokus auf umsatztreibende Rollen und das gleichzeitige Vernachlässigen der anderen ist aus meiner Sicht eines der zentralen Probleme unserer Branche. Nachdem wir uns in der vorigen Episode mit der Rolle des Process Owners aka Scrum Masters befasst haben, muss sich der eine oder andere von Ihnen vielleicht erst einmal hinsetzen in Anbetracht der Rolle, um die es diesmal geht: der Feelgood Manager. Ja, richtig gelesen: FEELGOOD! Klingt komisch, ist aber sehr wichtig.

Die 3 P der Softwareentwicklung

Treue Leser dieser Kolumne wissen um die Momente in meiner beruflichen Laufbahn, in denen ich etwas über die Softwareentwicklung gelernt habe, das mein gesamtes Bild davon nachhaltig zu etwas Besserem verändert hat. Einer dieser Momente war während einer Technical Due Diligence in einer Bank. Eine meiner Aufgaben bestand darin, die dortige Softwareentwicklungsabteilung zu bewerten. Am Ende einer anstrengenden Woche standen auf meinem Bewertungsbogen lauter Bestnoten, und voller Stolz teilte der Entwicklungsleiter sein Geheimnis: „Softwareentwicklung funktioniert immer dann, wenn wir die drei P im Griff haben: Product, Processes und People!“

So offensichtlich das vielleicht klingen mag, hat er damit doch meine komplette Denkweise revolutioniert. Von diesem Zeitpunkt an bin ich ein wesentlich besserer Entwickler und Architekt geworden, und auch eine bessere Führungskraft. Egal wie gut unser Produkt ist, egal wie gut wir die Prozesse außenherum gestalten – wenn die Menschen in der Entwicklung sich nicht wohlfühlen, nicht gerne zur Arbeit kommen und ihren Job nicht gerne machen, bekommen wir auf lange Sicht Probleme. Seit jenem Moment habe ich mich unzählige Male in Projekten an genau diese Weisheit erinnert und daraufhin immer wieder bessere und vor allem nachhaltigere

Ergebnisse erzielt – weil ich den Faktor „Mensch“ in meine Berechnungen einbezogen habe.

Das Problem sind wir

Sollten Sie noch nie mit dieser Ansicht konfrontiert worden sein, kann ich mir nur zu gut vorstellen, wie es Ihnen gerade geht. Ich bin studierter Elektrotechniker und Informatiker. Dass ein Produkt relevant ist, ist für mich selbstverständlich. Dass der Prozess für dessen Erstellung wichtig ist, war mir zunächst nicht so ersichtlich, habe ich jedoch im Lauf der Zeit gelernt. Als ich dann diesen „People-Blödsinn“ das erste Mal gehört habe, habe ich zunächst nur gelacht und mich um das nächste technische Thema gekümmert – weil ich das verstanden habe, weil ich das vollständig beherrschen konnte.

Beim Faktor „Mensch“ war ich jedoch komplett „unbewaffnet“ – unerfahren, naiv und auch ein Stück weit unwillig. Vor einigen Jahren habe ich in einem Konzern übergangsweise für ein halbes Jahr die IT-Leitung in der Abteilung der Softwareentwicklung übernommen. Da diese Position für mich neu war, habe ich wirklich alles getan, um diesen Job so gut es ging zu erledigen, und war am Ende der sechs Monate auch ziemlich zufrieden mit dem Ergebnis. Schauen ich heute zurück, mit dem zwischenzeitlich gewonnenen Bewusstsein für die Bedeutung des „People“-Faktors, muss ich selbstkritisch resümieren, dass meine Performance tatsächlich alles andere als gut war. Auch wenn ich in den Bereichen „Produkt“ und „Prozesse“ sehr gut gearbeitet habe, war der Bereich „People“ eine Vollkatastrophe.

Ich spreche das hier bewusst an, weil ich das derzeit größte Problem in der Softwareentwicklung darin sehe, dass die vorherrschende Denkweise oft genug der des naiven David von vor zehn Jahren ähnelt. Sollten Sie auch noch darin feststecken, rate ich Ihnen, das schnellstmöglich zu ändern. Denn alle Teams, die ich in Kundenprojekten kennengelernt habe, bei denen der „People“-Bereich gut aufgestellt ist, haben entscheidende Vorteile in puncto Produktivität, Effektivität und – ganz besonders wichtig – Loyalität!

Kein Job für jedermann

Jetzt wird es Ihnen vielleicht ähnlich gehen wie mir: Ich bin Techniker, meine Welt sind Frameworks, Sprachen, Architekturen, Prozesse, Anforderungen und so weiter. Aber der Bereich „People“ ist etwas komplett anderes. Hier geht es darum, ein Arbeitsumfeld zu schaffen, in dem sich alle Mit-



arbeiter bedingungslos wohlfühlen. Jeder soll offen und frei über Probleme und Ängste sprechen können, es sollte Raum für Fehler geben, und jeder sollte sich und seine Arbeit wertgeschätzt fühlen. Genau das und einiges mehr ist die Aufgabe des Feelgood Managers.

Der Name ist dabei nicht nur eine perfekte Beschreibung für das Aufgabengebiet, sondern zugleich leider auch das größte Problem. „Mitarbeiter sollen hierher kommen, um zu arbeiten, und nicht, weil es eine Wohlfühloase ist“, entgegnete mir einst ein Geschäftsführer, bei dem ich genau diese Rolle einführen wollte. Natürlich müssen wir arbeiten, keine Frage. Aber was spricht bitte dagegen, dass die Arbeitsstelle zugleich auch eine Wohlfühloase ist? Genau, es spricht nichts dagegen, im Gegenteil: Mit der Einführung eines Feelgood Managers habe ich das eine oder andere echte „Ekel“-Unternehmen, bei denen die Mitarbeiter zuvor reihenweise gekündigt hatten, zu solch einer Wohlfühloase machen können, zu der genau die gleichen Menschen aufgrund des positiven Feedbacks der ehemaligen Kollegen wieder zurückgekommen sind. Mir scheint, dass wir in der Softwareentwicklung durch die vielen Technologien und Möglichkeiten auf einer der wichtigsten Ebenen ziemlich abgestumpft sind, nämlich der Bedeutung einer guten Stimmung zwischen Mitarbeitern, Teams und Abteilungen.

„Aber wer soll das machen?“, werden Sie sich jetzt fragen. Nun, zugegeben, die Rolle des Feelgood Managers ist, ähnlich wie die des Process Owners aus der letzten Folge, nicht so leicht zu besetzen. Es sollte jemand mit einem ausgeprägten Maß an Empathie sein. Die Person sollte gut mit Menschen umgehen können, immer ein offenes Ohr haben und immer bereit sein, Menschen besser zu machen, Konflikte zu lösen – und auch unangenehme Situationen nicht scheuen.

Ich habe in Projekten bereits Psychologen in dieser Rolle eingesetzt, aber auch ehemalige Sozialarbeiter oder sogar Kindergärtner oder frühere Mitarbeiter aus dem Concierge-

Service eines Hotels, um nur ein paar wenige der Berufsbilder zu nennen. Entscheidend war und ist das Rüstzeug, das es braucht, um Menschen zu verbessern, und das Bestreben, allen Mitarbeitern eine maximal gute Arbeitsatmosphäre ermöglichen zu wollen. Das kann man nicht lernen – in meinen Augen ist das ein fast schon angeborener Instinkt, ein Natürliches, das ein Feelgood Manager zwingend haben muss.

Warum der Feelgood Manager wichtig ist

So merkwürdig es klingen mag – ich betrachte die Rolle des Feelgood Managers inzwischen als eine der wichtigsten Rollen in der Softwareentwicklung. In den Projekten, die ich über all die Jahre in den verschiedensten Unternehmen begleiten durfte, habe ich immer wieder festgestellt, dass es oft nicht die technischen oder prozessualen Themen waren, die Projekte zum Scheitern brachten, sondern vielmehr der Faktor „Mensch“ und (gestörte) zwischenmenschliche Beziehungen. Aber um diese Erkenntnis zuzulassen, müssen Führungskräfte, Geschäftsführer und Vorstände sich erst einmal eingestehen, dass in ihrem Unternehmen über viele Jahre viele, teils sehr unangenehme Probleme existieren und unterdrückt wurden: falsche Mindsets, Mobbing, Burn-outs, Depressionen und vieles andere mehr. Oder anders ausgedrückt: Die Mitarbeiter waren schon sehr lange unglücklich, aber keine der Führungskräfte hat es gemerkt und/oder wollte es ändern. Das wirft natürlich kein gutes Licht auf die Führung eines Unternehmens, deshalb wird diese Einsicht in vielen Fällen unterdrückt – und die Position des Feelgood Managers ist dann kaum durchsetzbar.

Wir sehen also: Die drei P – Personen, Prozesse, Produkt – sind eine Tatsache. Und zwar alle drei, ohne Unterschied. Schauen wir uns moderne agile Prozesse wie beispielsweise Scrum dahingehend einmal an, decken deren Frameworks diesen Bereich allerdings nicht vollständig ab. Das Team, also die Entwickler und der Product Owner beziehungsweise der Scrum Master, sind dafür zuständig, das Produkt ebenso wie die Prozesse von Sprint zu Sprint besser zu machen. Aber eben hier, genau an diesem Punkt, finden wir ein Problem: Produkt und Prozesse werden verbessert, nicht aber der Mensch. Der Faktor „People“ fehlt in modernen agilen Entwicklungsansätzen, und genau das ist die tickende Bombe in unzähligen Projekten.

Deshalb hoffe ich, Sie dafür sensibilisiert zu haben, wie wichtig die Rolle des Feelgood Managers ist. Ich habe noch kein einziges Projekt erlebt, bei dem diese Rolle sich nicht als Gewinn in Hinsicht auf das Gesamtergebnis herausgestellt hätte – seien Sie deshalb klüger als der Rest! ■



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

EVENT-SOURCING ALS PERSPEKTIVENWECHSEL

Wider das Vergessen in Datenbanken

Event-Sourcing erfasst jede Änderung als unveränderlichen Event und macht damit Ihre Daten zu einer Zeitmaschine für Debugging, Analytics und KI.

Die meisten Anwendungen, die wir entwickeln, folgen einem vertrauten Muster: Daten werden in einer Datenbank gespeichert, bei Änderungen überschrieben und bei Bedarf wieder abgerufen. Dieses CRUD-Paradigma – Create, Read, Update, Delete – ist einfach zu verstehen, schnell zu implementieren und für viele Szenarien völlig ausreichend. Die Welt der Softwareentwicklung hat sich über Jahrzehnte auf diese vier Operationen eingeschworen, und die meisten relationalen Datenbanken sind genau dafür optimiert.

Doch CRUD hat einen fundamentalen Nachteil, der umso deutlicher wird, je komplexer die fachlichen Anforderungen werden: Es speichert nur den aktuellen Zustand, nicht aber den Weg dorthin.

Stellen wir uns eine Bibliotheksverwaltung vor. In der Datenbank steht:

```
Buch: #42
Status: "Ausgeliehen"
Ausgeliehen an: "Leser #23"
Rückgabe fällig: "15.11.2025"
```

Was die Datenbank uns nicht mehr verrät: Wann genau wurde dieses Buch angeschafft? Hat es einen vorherigen Ausleihzyklus gegeben? Gab es Probleme bei der Rückgabe? Wurde das Buch als verloren gemeldet und später wiedergefunden? War der ursprüngliche Status „Verfügbar“ oder „In Reparatur“? All diese Informationen sind verloren – überschrieben durch die jeweils neueste Version des Datensatzes. Das `UPDATE`-Statement hat die Vergangenheit gelöscht und durch die Gegenwart ersetzt.

Dieses Problem ist kein technisches Versehen, sondern eine konzeptionelle Konsequenz von CRUD. Die vier Operationen – Create, Read, Update, Delete – reduzieren die Komplexität der Realität auf ein Minimum. Das hat seinen Reiz: Wer könnte etwas gegen Einfachheit einwenden?

Doch diese Reduktion ist auch eine Verarmung. Die Realität der Geschäftsprozesse ist vielfältiger, nuancierter und facettenreicher, als vier technische Operationen es abbilden können. Ein `UPDATE` auf einer Bücher-Tabelle sagt nichts darüber aus, warum sich der Status geändert hat. War es eine automatische Rückgabe nach Fristablauf? Eine manuelle Verlängerung durch einen Bibliothekar? Eine Korrektur nach

einem Fehler? Die fachliche Bedeutung – das Warum hinter dem Was – geht verloren.

Natürlich können wir versuchen, dieses Problem zu lösen: Audit-Logs in separaten Tabellen, Änderungshistorien mit Zeitstempeln, Trigger, die bei jedem `UPDATE` eine Kopie des alten Zustands wegspeichern. Doch diese Lösungen wirken wie nachträglich aufgesetzte Pflaster auf ein System, das von Grund auf nicht für historische Nachvollziehbarkeit konzipiert wurde. Sie fügen Komplexität hinzu, ohne das fundamentale Problem zu adressieren: dass CRUD den Fokus auf den Zustand legt, nicht auf die Veränderungen, die zu diesem Zustand geführt haben. Und genau hier setzt Event-Sourcing an.

Event-Sourcing wählt einen radikal anderen Ansatz: Statt nur den aktuellen Zustand zu speichern, werden alle Zustandsänderungen als unveränderliche Events aufgezeichnet. Der aktuelle Zustand wird dann durch einen Replay dieser Events rekonstruiert. Was zunächst umständlich klingt, eröffnet in Wirklichkeit zahlreiche neue Möglichkeiten: vollständige Nachvollziehbarkeit jedes Geschäftsvorfalles, zeitliches Debugging durch Zeitreise in die Vergangenheit, flexible Auswertungen durch nachträgliche Interpretation der Events und eine natürliche Basis für Event-getriebene Architekturen. Mehr noch: Event-Sourcing zwingt uns dazu, die Fachlichkeit unserer Domäne ernst zu nehmen. Statt abstrakte technische Operationen wie `UPDATE` zu modellieren, erfassen wir konkrete fachliche Ereignisse: „Buch angeschafft“, „Buch ausgeliehen“, „Buch zurückgegeben“, „Mahngebühr erhoben“. Die Software wird dadurch nicht nur technisch anders – sie spricht die Sprache der Fachlichkeit.

Um diese Konzepte im Verlauf des Artikels greifbar zu machen, werden wir durchgängig mit dem Beispiel dieser Stadtbibliothek arbeiten. Die Domäne ist verständlich und bietet dennoch interessante Aspekte: Bücher anschaffen, Leser registrieren, Bücher ausleihen und zurückgeben, Mahngebühren erheben. Als Event-Store kommt dabei EventSourcingDB [1] zum Einsatz – eine spezialisierte Datenbank, die von Grund auf für Event-Sourcing entwickelt wurde und dem CloudEvents-Standard [2] folgt. Wir werden sehen, wie diese Geschäftsvorfälle als Events modelliert werden und wie EventSourcingDB dabei hilft, diese Events zu speichern, zu lesen und zu verarbeiten.

Das Konzept: Events als unveränderliche Fakten

Ein Event ist dabei ein unveränderlicher Fakt darüber, was in der Vergangenheit passiert ist. Er beschreibt nicht, was passieren soll (das wäre ein Command), sondern was bereits geschehen ist. Events werden daher konsequent in der Vergangenheitsform benannt, um diese fundamentale Eigenschaft zu betonen: *BookAcquired* statt *AcquireBook*, *BookBorrowed* statt *BorrowBook*, *LateFeeCharged* statt *ChargeLateFee*. Diese Benennung ist nicht nur Konvention – sie spiegelt die unveränderliche Natur von Events wider. Was passiert ist, kann nicht mehr geändert werden. Die Vergangenheit ist Fakt. Fehler werden daher nicht durch Überschreiben korrigiert, sondern durch neue, kompensierende Events. Wenn ein Buch fälschlicherweise als ausgeliehen markiert wurde, schreiben wir kein *UPDATE*, sondern einen neuen Event: *BookBorrowingCorrected* oder *BookReturnedEarly*. Die fehlerhafte Ausleihe bleibt in der Historie sichtbar – und damit nachvollziehbar.

Events tragen fachliche Semantik in sich. Im Gegensatz zum abstrakten *UPDATE books SET status = 'borrowed'* eines CRUD-Systems erfasst ein Event wie *BookBorrowed* explizit, was geschehen ist. Er kann zusätzliche Kontextinformationen enthalten: Wer hat das Buch ausgeliehen? Für welchen Zeitraum? Um welche Uhrzeit? All das sind Informationen, die in einem CRUD-System entweder verloren gehen oder mühsam in separaten Logging-Mechanismen nachgehalten werden müssen. In einem Event-gesourceten System sind sie Teil der natürlichen Datenstruktur.

Der Event-Store: Mehr als eine Datenbank

Events werden in einem Event-Store gespeichert – einer spezialisierten Datenbank, die ausschließlich für diesen Zweck konzipiert ist. Ein Event-Store ist eine Append-only-Datenbank: Events werden nur hinzugefügt, niemals geändert oder

gelöscht. Jeder Event erhält eine eindeutige, fortlaufende ID und wird mit einem Zeitstempel versehen. Der Event-Store ist die einzige „Source of Truth“ des Systems – alle anderen Datenspeicher sind abgeleitete Projektionen, die aus den Events rekonstruiert werden können.

Anders als relationale Datenbanken, wo Entitäten als Zeilen in Tabellen organisiert sind, strukturiert ein Event-Store die Daten in Streams. Ein Stream ist eine logische Sequenz von Events, die zu einer bestimmten Entität gehören. Jeder Stream hat eine eindeutige Kennung, die sich typischerweise aus der Art der Entität und ihrer ID zusammensetzt. In unserer Stadtbibliothek hätte ein Stream für das Buch mit der ID 42 die Kennung */books/42* und könnte Events wie *BookAcquired*, *BookBorrowed*, *BookReturned* oder *BookLost* enthalten. Ein Stream für einen Leser mit der ID 23 hätte die Kennung */readers/23* und könnte Events wie *ReaderRegistered*, *ReaderAddressChanged*, *ReaderSuspended* enthalten. Diese Streams sind unabhängig voneinander, können aber über ihre Daten aufeinander verweisen – etwa wenn ein *BookBorrowed*-Event festhält, welcher Leser das Buch ausgeliehen hat.

EventSourcingDB ist ein solcher Event-Store. Im Gegensatz zu relationalen Datenbanken oder NoSQL-Stores, die ursprünglich für andere Zwecke konzipiert wurden, ist EventSourcingDB ausschließlich auf das Speichern, Lesen und Beobachten von Events optimiert.

Aus Events wird Zustand

Um den aktuellen Zustand einer Entität zu ermitteln, werden alle Events des entsprechenden Streams von Anfang bis Ende abgespielt – ein Vorgang, der wie bereits erwähnt als Replay bezeichnet wird. Jeder Event wird dabei schrittweise angewendet und verändert den Zustand inkrementell. Betrachten wir den Stream eines Buches in unserer Bibliothek:

```
Start: Buch existiert noch nicht
→ BookAcquired (Titel: "2001 - A Space Odyssey",
  Autor: "Arthur C. Clarke")
  = Zustand: Buch vorhanden, Status "Verfügbar"
→ BookBorrowed (Ausgeliehen an: Leser #23,
  Bis: 15.11.2025)
  = Zustand: Buch ausgeliehen, Rückgabe fällig am 15.11.2025
→ BookReturned
  = Zustand: Buch verfügbar
→ BookBorrowed (Ausgeliehen an: Leser #42,
  Bis: 30.11.2025)
  = Zustand: Buch ausgeliehen, Rückgabe fällig am 30.11.2025
```

Der aktuelle Zustand – „2001 – A Space Odyssey“ von Arthur C. Clarke ist ausgeliehen an Leser #42 bis zum 30.11.2025 – ist das Ergebnis der sequenziellen Anwendung aller Events. Dieser Mechanismus mag zunächst aufwendig erscheinen, bietet aber eine Fähigkeit, die in CRUD-Systemen unmöglich ist: Wir können den Zustand zu jedem beliebigen Zeitpunkt rekonstruieren. Möchten wir etwa wissen, ob das Buch am 10.11.2025 verfügbar war, dann spielen wir alle Events bis ►

● Weiterführende Ressourcen

EventSourcingDB:

- [Eventsourcingdb.io](https://eventsourcingdb.io) – die offizielle Website mit Überblick, Features und Einstiegsinformationen [1]
- [Docs.eventsourcingdb.io](https://docs.eventsourcingdb.io) – umfassende Dokumentation mit Tutorials, API-Referenz und Best Practices [3]
- NuGet: [EventSourcingDb](https://www.nuget.org/packages/EventSourcingDb) – das .NET Client SDK zum Installieren via NuGet [4]

Event-Sourcing und CQRS:

- [CQRS.com](https://cQRS.com) – Einführung in die Konzepte von CQRS und Event-Sourcing mit praktischen Beispielen [5]
- [Eventsourcing.ai](https://eventsourcing.ai) – Wie Event-Sourcing mit KI, maschinellem Lernen und Data-Mesh-Architekturen zusammenspielt [6]

Event-Stores im Vergleich:

- [Eventsourcingdatabase.com](https://eventsourcingdatabase.com) – Überblick und Vergleich verschiedener Event-Store-Lösungen mit ihren jeweiligen Stärken [7]

zu diesem Zeitpunkt ab und erhalten die Antwort. Diese Fähigkeit zur Zeitreise macht Debugging zu einer völlig neuen Erfahrung: Statt zu raten, warum ein System sich in einem bestimmten Zustand befindet, können wir die exakte Ereignissequenz nachvollziehen, die dorthin geführt hat.

● Listing 1: BookAcquired-Event

```
using EventSourcingDb;

var url = new Uri("http://localhost:3000");
var apiToken = "secret";

var client = new Client(url, apiToken);

var eventCandidate = new EventCandidate(
    Source: "https://library.eventsourcingdb.io",
    Subject: "/books/42",
    Type: "io.eventsourcingdb.library.book-acquired",
    Data: new
    {
        Title = "2001 - A Space Odyssey",
        Author = "Arthur C. Clarke",
        Isbn = "978-0756906788"
    }
);

// Precondition: Der Stream darf noch nicht
// existieren
var precondition = Precondition
    .IsSubjectPristinePrecondition("/books/42");

var writtenEvents = await client.WriteEventsAsync(
    new[] { eventCandidate },
    new[] { precondition }
);

// writtenEvents enthält das geschriebene Event
// mit allen von der Datenbank ergänzten Feldern
// (ID, Zeitstempel etc.)
```

Konsistenz ohne Locks

In verteilten Systemen oder Umgebungen mit parallelen Schreibzugriffen stellt sich die Frage: Wie verhindern wir, dass zwei gleichzeitige Operationen sich gegenseitig überschreiben und zu inkonsistentem Zustand führen? CRUD-Systeme lösen dieses Problem häufig durch Datenbank-Locks: Eine Transaktion sperrt eine Zeile, führt ihre Änderung durch und gibt die Sperre wieder frei. Andere Transaktionen müssen warten. Das führt zu Engpässen und kann in Systemen mit hoher Last die Performance dramatisch reduzieren.

Event-Sourcing verwendet stattdessen Optimistic Concurrency Control: Statt Ressourcen zu sperren, gehen wir davon

aus, dass Konflikte selten sind. Beim Schreiben eines neuen Events wird überprüft, ob der Stream sich seit dem letzten Lesen verändert hat. Diese Prüfung geschieht über sogenannte Preconditions – Bedingungen, die erfüllt sein müssen, damit der Event akzeptiert wird.

Eine typische Precondition ist beispielsweise *isSubjectOnEventId*: Sie stellt sicher, dass der letzte Event im Stream eine bestimmte ID hat. Stimmt die erwartete ID nicht mit der tatsächlichen überein, wurde der Stream zwischenzeitlich von jemand anderem verändert, und das Schreiben schlägt fehl. Die Anwendung kann dann den aktuellen Zustand neu laden, ihre Logik erneut ausführen und es nochmals versuchen. Konflikte werden erkannt, nicht verhindert – und können auf Anwendungsebene intelligent behandelt werden, statt blind zu Deadlocks zu führen.

EventSourcingDB in der Praxis

Bevor wir Events schreiben können, müssen wir EventSourcingDB lokal starten und eine Verbindung aus unserer .NET-Anwendung herstellen. EventSourcingDB läuft als Docker-Container und ist mit einem einzigen Befehl einsatzbereit:

```
docker run -it -p 3000:3000 \
    thenativeweb/eventsourcingdb run \
    --api-token=secret \
    --data-directory-temporary \
    --http-enabled \
    --https-enabled=false \
    --with-ui
```

Dieser Befehl startet EventSourcingDB auf Port 3000 mit einem temporären Datenverzeichnis (geeignet für Entwicklung und Tests). Der Parameter *--api-token=secret* definiert ein API-Token zur Authentifizierung. Für Produktionsumgebungen würden wir persistente Volumes, HTTPS und sichere Tokens verwenden, aber für unsere ersten Schritte reicht diese Konfiguration aus. Nach dem Start können wir das Management-UI unter *http://localhost:3000* im Browser öffnen und dort die Events inspizieren.

Um aus .NET auf EventSourcingDB zuzugreifen, installieren wir das offizielle .NET Client SDK via NuGet:

```
dotnet add package EventSourcingDb
```

Anschließend erstellen wir eine Client-Instanz und prüfen die Verbindung:

```
using EventSourcingDb;

var url = new Uri("http://localhost:3000");
var apiToken = "secret";

var client = new Client(url, apiToken);

await client.PingAsync();
// Wenn die Verbindung steht, kehrt PingAsync ohne
// Fehler zurück
```

● Listing 2: BookBorrowed-Event

```
var borrowedEvent = new EventCandidate(
    Source: "https://library.event sourcingdb.io",
    Subject: "/books/42",
    Type: "io.event sourcingdb.library.book-borrowed",
    Data: new {
        BorrowedBy = "/readers/23",
        BorrowedUntil = new DateTime(2025, 11, 15)
    }
);

var precondition = Precondition
    .IsSubjectOnEventIdPrecondition("/books/42", "0");

var writtenEvents = await client.WriteEventsAsync(
    new[] { borrowedEvent },
    new[] { precondition }
);
```

Nun können wir mit dem Schreiben von Events beginnen.

Den ersten Event schreiben: Ein Buch anschaffen

Die Bibliothek schafft ein neues Buch an – das ist der logische Startpunkt. Wir modellieren diesen Geschäftsvorfall als *Book-Acquired-Event* (Listing 1). Events in EventSourcingDB folgen dem *CloudEvents*-Standard und benötigen mindestens vier Felder: *source* (die Quelle, die den Event erzeugt hat), *subject* (der Stream, zu dem der Event gehört), *type* (der Event-Typ) und *data* (die fachlichen Daten). Der Event-Typ muss in Reverse Domain Notation angegeben werden, etwa *io.event-sourcingdb.library.book-acquired*.

Die *IsSubjectPristine*-Precondition stellt sicher, dass der Stream */books/42* noch nicht existiert – der Event wird also nur akzeptiert, wenn es der erste Event für dieses Buch ist. Würde bereits ein Event für */books/42* existieren, würde das Schreiben fehlschlagen, und wir könnten entsprechend reagieren (etwa indem wir eine andere Buch-ID wählen oder einen Fehler anzeigen).

Der geschriebene Event wird von EventSourcingDB mit zusätzlichen Metadaten angereichert: Er erhält eine eindeutige ID, die global über alle Streams hinweg fortlaufend hochgezählt wird (der allererste Event in der Datenbank hat die ID 0), einen Zeitstempel (*time*) und wird dauerhaft im Event-Store persistiert. Ab diesem Moment ist der Event Teil der unveränderlichen Historie.

Ein weiterer Event: Das Buch ausleihen

Ein Leser möchte das Buch ausleihen. Wir erfassen dies als *BookBorrowed-Event* (Listing 2). Diesmal verwenden wir eine andere Precondition: *IsSubjectOnEventId*. Sie stellt sicher, dass der letzte Event im Stream die ID 0 hat – also der *Book-Acquired-Event* (der in unserem Beispiel zufällig auch der allererste Event in der gesamten Datenbank war). Wurde zwei-

schenzeitlich ein anderer Event geschrieben (etwa eine zweite Ausleihe durch einen anderen Prozess), schlägt das Schreiben fehl, und wir erkennen den Konflikt.

Diese Precondition implementiert Optimistic Concurrency Control: Nur wenn niemand anderes zwischenzeitlich einen Event in diesen Stream geschrieben hat, wird die Ausleihe akzeptiert. Das schützt vor Situationen, in denen zwei Leser gleichzeitig dasselbe Buch ausleihen wollen – eine fachlich inkonsistente Situation, die das System verhindern muss.

Events lesen und Zustand rekonstruieren

Um den aktuellen Zustand des Buchs zu ermitteln, lesen wir alle Events seines Streams und wenden sie sequenziell an. EventSourcingDB bietet dafür die *ReadEventsAsync*-Methode, die einen asynchronen Stream von Events zurückgibt (Listing 3).

Die *Book*-Klasse und ihre *Apply*-Methoden kümmern sich um die Zustandsrekonstruktion. In Listing 4 verwenden wir C# Records, um die Unveränderlichkeit der Zustände zu betonen. Jede *Apply*-Methode gibt eine neue Instanz von *Book* ►

● Listing 3: ReadEventsAsync-Methode

```
var events = client.ReadEventsAsync("/books/42",
    new ReadEventsOptions
    {
        Recursive = false // Nur Events dieses Streams,
                          // keine Unter-Subjects
    });

var book = new Book(); // Startzustand: leeres
                      // Objekt

await foreach (var evt in events)
{
    book = evt.Type switch
    {
        "io.event sourcingdb.library.book-acquired" =>
            book.Apply(
                evt.Data.Deserialize<BookAcquiredData>()),
        "io.event sourcingdb.library.book-borrowed" =>
            book.Apply(
                evt.Data.Deserialize<BookBorrowedData>()),
        "io.event sourcingdb.library.book-returned" =>
            book.Apply(
                evt.Data.Deserialize<BookReturnedData>()),
        _ => book // Unbekannte Events ignorieren
                // (forward compatibility)
    };
}

// 'book' enthält jetzt den aktuellen Zustand
Console.WriteLine(
    $"Buch: {book.Title}, Status: {book.Status}");
```

● Listing 4: Zustandsrekonstruktion

```
public record Book
{
    public string? Title { get; init; }
    public string? Author { get; init; }
    public BookStatus Status { get; init; } =
        BookStatus.Unknown;
    public string? BorrowedBy { get; init; }
    public DateTime? BorrowedUntil { get; init; }

    public Book Apply(BookAcquiredData data)
    {
        return this with
        {
            Title = data.Title,
            Author = data.Author,
            Status = BookStatus.Available
        };
    }

    public Book Apply(BookBorrowedData data)
    {
        return this with
        {
            Status = BookStatus.Borrowed,
            BorrowedBy = data.BorrowedBy,
            BorrowedUntil = data.BorrowedUntil
        };
    }

    public Book Apply(BookReturnedData data)
    {
        return this with
        {
            Status = BookStatus.Available,
            BorrowedBy = null,
            BorrowedUntil = null
        };
    }
}

public enum BookStatus { Unknown, Available, Borrowed, Lost }

public record BookAcquiredData(
    string Title, string Author, string Isbn);
public record BookBorrowedData(
    string BorrowedBy, DateTime BorrowedUntil);
public record BookReturnedData();
```

zurück – der alte Zustand bleibt unverändert. Das macht die Logik nachvollziehbar und testbar: Für jeden Event-Typ wissen wir genau, wie er den Zustand transformiert.

Performance-Optimierung durch Zwischenstände

Bei Entitäten mit sehr vielen Events – etwa einem Bankkonto mit Tausenden Transaktionen oder einem Dokument mit Hunderten Bearbeitungsschritten – kann das vollständige Replay aller Events zeitaufwendig werden. Snapshots lösen dieses Problem: In regelmäßigen Abständen speichern wir den aktuellen Zustand als speziellen Event-Typ im selben Stream. Bei EventSourcingDB sind Snapshots keine separate Funktionalität, sondern einfache Events, die als Snapshot interpretiert werden können.

Nehmen wir an, wir möchten nach einer Weile einen Snapshot erstellen:

```
var snapshotEvent = new EventCandidate(
    Source: "https://library.event sourcingdb.io",
    Subject: "/books/42",
    Type: "io.event sourcingdb.library.snapshot-taken",
    Data: new
    {
        Title = book.Title,
        Author = book.Author,
        Status = book.Status,
        BorrowedBy = book.BorrowedBy,
```

```
BorrowedUntil = book.BorrowedUntil,
    }
});

await client.WriteEventsAsync(new[] { snapshotEvent });
```

Beim Laden können wir dann mithilfe der *FromLatestEvent*-Option beim Lesen zunächst anhand des Event-Typs *io.event-sourcingdb.library.snapshot-taken* nach dem neuesten Snapshot suchen und von dort aus weiterrechnen, statt alle Events von Anfang an zu verarbeiten. Das ist eine reine Performance-Optimierung – fachlich ändert sich nichts. Der Event-Store bleibt die Source of Truth, Snapshots sind lediglich eine Caching-Strategie.

CQRS: Schreiben und Lesen entkoppeln

Event-Sourcing allein löst viele Probleme von CRUD, führt aber auch eine neue Herausforderung ein: Das Lesen von Daten wird komplexer. Statt einfach eine Tabellenzeile zu lesen, müssen wir Events abspielen. Komplexe Abfragen – etwa „Zeige alle verfügbaren Bücher von Arthur C. Clarke“ – erfordern das Replay vieler Streams. Hier kommt CQRS (Command Query Responsibility Segregation) ins Spiel. CQRS ist ein Architekturmuster, das Lese- und Schreiboperationen strikt trennt – und damit perfekt zu Event-Sourcing passt.

In klassischen CRUD-Anwendungen dient dasselbe Datenmodell sowohl zum Schreiben als auch zum Lesen. Ein *Book*-

Objekt wird erstellt, in die Datenbank geschrieben und später wieder abgerufen, um es in der Benutzeroberfläche anzuzeigen. Doch die Anforderungen an Writes und Reads sind fundamental unterschiedlich:

- Writes benötigen: Geschäftslogik, Validierung, Transaktionskonsistenz, Schutz vor Nebenläufigkeit. Beim Schreiben geht es darum, fachliche Regeln durchzusetzen: Kann dieser Leser das Buch ausleihen? Ist das Buch überhaupt verfügbar? Die Datenstruktur muss diese Regeln abbilden können.
- Reads benötigen: Performance, flexible Abfragen, denormalisierte Daten, schnelle Antwortzeiten. Beim Lesen geht es darum, Informationen effizient bereitzustellen: Zeige alle Bücher, sortiert nach Autor. Zeige alle überfälligen Ausleihen. Berechne Statistiken über Ausleihhäufigkeiten. Die Datenstruktur muss für diese Abfragen optimiert sein – oft durch Redundanz, Indizes und Denormalisierung.

Ein Datenmodell für beide Zwecke zu verwenden führt zu Kompromissen: Komplexe Joins für Abfragen, umständliche Validierung, ineffiziente Indizes. CQRS löst dieses Problem,

indem es zwei separate Modelle verwendet: eines für Schreiboperationen (Command Side, Write Model) und eines für Leseoperationen (Query Side, Read Model).

Zwei Seiten derselben Medaille

CQRS teilt das System in zwei klare Verantwortlichkeiten:

- **Command Side (Write Side):** Diese Seite nimmt Befehle entgegen – etwa *BorrowBook*, *ReturnBook*, *ChargeLateFee*. Sie validiert die Geschäftslogik, prüft Vorbedingungen und schreibt bei Erfolg Events in den Event-Store. Das Write Model kennt nur die eventbasierte Darstellung und kümmert sich nicht um Abfragen. Es ist optimiert für Konsistenz und Geschäftsregeln.
- **Query Side (Read Side):** Diese Seite beantwortet Anfragen – etwa „Zeige alle verfügbaren Bücher“, „Welche Bücher hat Leser #23 ausgeliehen?“, „Wie viele Bücher sind überfällig?“. Die Query Side liest Events aus dem Event-Store und baut daraus spezialisierte Read-Modelle auf, die für schnelle Abfragen optimiert sind. Diese Read-Modelle können in relationalen Datenbanken, NoSQL-Stores oder ►

● Listing 5: Bücherkatalog für die Suche

```
public class BookCatalogProjection
{
    private readonly LibraryDbContext _db;

    public BookCatalogProjection(LibraryDbContext db)
    {
        _db = db;
    }

    public async Task HandleAsync(Event evt)
    {
        switch (evt.Type)
        {
            case "io.eventsourcingsdb.library.book-acquired":
                var acquired =
                    evt.Data.Deserialize<BookAcquiredData>();
                var bookId = evt.Subject.Split('/').Last();

                await _db.CatalogBooks.AddAsync(
                    new CatalogBook
                    {
                        Id = bookId,
                        Title = acquired.Title,
                        Author = acquired.Author,
                        Isbn = acquired.Isbn,
                        Status = "Available",
                        LastUpdated = evt.Time
                    });
                await _db.SaveChangesAsync();
                break;

            case "io.eventsourcingsdb.library.book-borrowed":
                var borrowed =
                    evt.Data.Deserialize<BookBorrowedData>();
                var borrowedBook = await _db.CatalogBooks
                    .FindAsync(evt.Subject.Split('/').Last());

                if (borrowedBook != null)
                {
                    borrowedBook.Status = "Borrowed";
                    borrowedBook.BorrowedUntil =
                        borrowed.BorrowedUntil;
                    borrowedBook.LastUpdated = evt.Time;
                    await _db.SaveChangesAsync();
                }
                break;

            case "io.eventsourcingsdb.library.book-returned":
                var returnedBook = await _db.CatalogBooks
                    .FindAsync(evt.Subject.Split('/').Last());

                if (returnedBook != null)
                {
                    returnedBook.Status = "Available";
                    returnedBook.BorrowedUntil = null;
                    returnedBook.LastUpdated = evt.Time;
                    await _db.SaveChangesAsync();
                }
                break;
        }
    }
}
```

In-Memory-Caches liegen – je nachdem, was für die jeweilige Abfrage am besten geeignet ist.

Die beiden Seiten kommunizieren über Events: Die Command Side schreibt Events, die Query Side liest sie und aktualisiert ihre Read-Modelle. Diese Entkopplung ermöglicht es, beide Seiten unabhängig voneinander zu optimieren und zu skalieren. Die Write Side kann auf Konsistenz optimiert werden, die Read Side auf Performance. Und wir können nun mehrere verschiedene Read-Modelle parallel betreiben – jedes für einen anderen Anwendungsfall.

nen laufen unabhängig voneinander und können unterschiedliche Technologien verwenden. Der Event-Store ist die gemeinsame Quelle, aus der beide ihre Daten beziehen.

Events in Echtzeit beobachten

EventSourcingDB unterstützt Subscriptions – die Möglichkeit, Events in Echtzeit zu empfangen, sobald sie geschrieben werden. Das ist ideal für Projektionen, die ihre Read-Modelle aktuell halten müssen. Mit der *ObserveEventsAsync*-Methode abonnieren wir einen Stream oder ein Subject-Präfix und erhalten alle neuen Events ([Listing 7](#)).

● Listing 6: Leserhistorie für Analysen

```
public class BorrowingAnalyticsProjection
{
    private readonly AnalyticsWarehouse _warehouse;

    public async Task HandleAsync(Event evt)
    {
        if (evt.Type ==
            "io.event sourcingdb.library.book-borrowed")
        {
            var data =
                evt.Data.Deserialize<BookBorrowedData>();

            await _warehouse.RecordBorrowingAsync(
                new BorrowingRecord
                {
                    EventId = evt.Id,
                    BookId = evt.Subject.Split('/').Last(),
                    ReaderId = data.BorrowedBy.Split('/').Last(),
                    BorrowedAt = evt.Time,
                    DueDate = data.BorrowedUntil
                });
        }
        else if (evt.Type ==
            "io.event sourcingdb.library.book-returned")
        {
            var bookId = evt.Subject.Split('/').Last();
            await _warehouse.MarkAsReturnedAsync(
                bookId, evt.Time);
        }
    }
}
```

Von Events zu Abfrage-Modellen

Eine Projektion ist ein Prozess, der Events liest und daraus ein spezialisiertes Read-Modell aufbaut. Für unsere Bibliothek könnten verschiedene Projektionen jeweils unterschiedliche Sichten auf dieselben Events bieten. Hier zwei Beispiele:

- **Projektion 1:** Bücherkatalog für die Suche ([Listing 5](#)). Diese Projektion baut eine Tabelle auf, die alle Bücher mit ihrem aktuellen Status enthält. Sie ist optimiert für Abfragen wie „Zeige alle verfügbaren Bücher von Autor X“ oder „Suche nach Titel Y“. Diese Projektion schreibt in eine relationale Datenbank (etwa SQL Server oder PostgreSQL), die für Volltextsuche und komplexe Abfragen optimiert ist. Das Frontend kann direkt gegen diese Datenbank abfragen, ohne Events abspielen zu müssen.
- **Projektion 2:** Leserhistorie für Analysen ([Listing 6](#)). Eine zweite Projektion könnte alle Ausleihvorgänge in einem Data Warehouse sammeln, um statistische Auswertungen zu ermöglichen: „Welche Bücher werden am häufigsten ausgeliehen?“, „Wie lange dauert eine durchschnittliche Ausleihe?“, „Welche Leser haben die meisten Bücher ausgeliehen?“. Diese Projektion schreibt in ein anderes Datenbankschema oder sogar in ein anderes System – etwa ein OLAP-Cube oder eine Big-Data-Plattform. Die beiden Projektio-

Die Subscription liefert Events in der Reihenfolge, in der sie geschrieben wurden. Falls die Anwendung abstürzt oder neu gestartet wird, können wir die Subscription mit einem *lower-Bound*-Parameter fortsetzen – wir merken uns einfach die ID des letzten verarbeiteten Events und starten ab dort neu. So gehen keine Events verloren, und die Projektionen bleiben konsistent.

Projektionen laufen typischerweise als Background-Services oder Worker-Prozesse. Sie können auch mehrfach parallel laufen, um die Last zu verteilen – solange jede Projektion ihren eigenen Fortschritt verwaltet. Fällt eine Projektion aus, kann sie beim Neustart alle Events erneut abspielen und ihr Read-Modell komplett neu aufbauen. Das ist ein enormer Vorteil für Wartbarkeit und Debugging: Stimmt etwas im Read-Modell nicht, löschen wir es einfach und bauen es aus den Events neu auf.

Das bedeutet auch, dass es möglich ist, einem System im Nachhinein noch weitere Projektionen hinzuzufügen, die sich auf Basis der historischen Events zunächst auf den aktuellen Stand bringen. Mit anderen Worten: Wird erst Wochen, Monate oder gar Jahre nach der Entwicklung eines Systems ersichtlich, dass eine weitere Projektion erforderlich ist, lässt sie sich problemlos ergänzen und aufbauen, als wäre sie von

Anfang an vorhanden gewesen. Das ermöglicht also die nachträgliche Neuinterpretation der bereits gesammelten Daten – eine Eigenschaft, die ohne Event-Sourcing nicht möglich, häufig aber Gold wert ist.

Aufpassen bei Eventual Consistency

CQRS mit Event-Sourcing führt zu Eventual Consistency – was auf Deutsch so viel bedeutet wie „schlussendliche Konsistenz“: Zwischen dem Schreiben eines Events und seiner Verarbeitung durch Projektionen vergeht Zeit. Diese Verzögerung liegt typischerweise im Bereich von Millisekunden, kann aber bei hoher Last oder Netzwerkproblemen auch länger sein. Die Write Side ist sofort konsistent – sobald ein Event im Event-Store liegt, ist er Teil der Wahrheit. Die Read Side hingegen hinkt minimal mehr oder weniger hinterher, bis die Projektionen den Event verarbeitet haben.

Für viele Anwendungsfälle ist das kein Problem: Wenn ein Leser ein Buch ausleiht, muss der Katalog der verfügbaren Bücher nicht in derselben Millisekunde aktualisiert sein. Es ist schlichtweg unwahrscheinlich, dass exakt zeitgleich jemand anderes nach diesem Buch sucht und (für einen kurzen Moment) eine Verfügbarkeit angezeigt bekommt, die nicht mehr gegeben ist. Selbst wenn man bereits nach einigen Sekunden in den Katalog schaut, hat die Projektion den Event längst verarbeitet, und das Buch erscheint als „bereits verliehen“ in der Liste. Die kurze Verzögerung ist für den Nutzer praktisch nicht wahrnehmbar.

Für Szenarien, die strikte Konsistenz erfordern, gibt es Lösungsstrategien: Der Command-Handler kann synchron ein Read-Modell zurückgeben, das er selbst aus den Events aufbaut. Oder das UI zeigt optimistisch den erwarteten Zustand an, bevor die Projektion aktualisiert ist. Die meisten Nutzer nehmen die kurze Verzögerung gar nicht wahr, und das System gewinnt dadurch enorme Skalierbarkeit, weil Write Side und Read Side unabhängig voneinander operieren können.

Event-getriebene Architektur: Von der Anwendung zum System

Event-Sourcing ist nicht nur ein Persistenzmuster innerhalb einer Anwendung – es ist die natürliche Grundlage für Event-getriebene Architekturen, die ganze Systeme und Services miteinander verbinden. Während CQRS die Reads und Writes innerhalb eines Service (beziehungsweise eines Bounded Contexts) trennt, entkoppelt eine Event-getriebene Architektur verschiedene Services voneinander und ermöglicht ihnen, autonom zu agieren.

In Microservices-Architekturen haben verschiedene Services oft überlappende Informationsbedürfnisse. Klassische Lösungen führen zu Problemen: Synchroner REST-Aufrufe zwischen Services erzeugen enge Kopplung und können zu Kaskadenausfällen führen – wenn Service A abstürzt, kann Service B nicht mehr arbeiten. Das Teilen von Datenbanken zwischen Services bricht die Serviceautonomie und führt zu versteckten Abhängigkeiten.

Events bieten eine elegante Alternative: Services publizieren ihre Geschäftsvorfälle als Events, andere Services können diese abonnieren und ihre eigenen Modelle daraus aufbau-

en. Betrachten wir unsere Bibliothek: Der Library-Service ist verantwortlich für Bücher und Ausleihen. Er publiziert Events wie *BookBorrowed*, *BookReturned*, *BookOverdue*. Andere Services abonnieren diese Events für ihre eigenen Zwecke:

- **Der Notification-Service** abonniert *BookBorrowed*-Events und schickt dem Leser eine Erinnerungs-E-Mail drei Tage vor Ablauf der Leihfrist.
- **Der Analytics-Service** sammelt alle Ausleihvorgänge und erstellt Statistiken über Nutzungsverhalten.
- **Der Billing-Service** reagiert auf *BookOverdue*-Events und berechnet Mahngebühren.
- **Der Inventory-Service** aktualisiert seine Bestandsübersicht basierend auf allen Buch-Events.

Keiner dieser Services muss vom Library-Service wissen oder ihn direkt aufrufen. Sie reagieren autonom auf die Events, die für sie relevant sind.

Diese Entkopplung hat mehrere Vorteile: Services können unabhängig voneinander entwickelt, deployt und skaliert werden. Neue Services können hinzugefügt werden, ohne bestehende Services zu ändern – sie abonnieren einfach die relevanten Events. Fällt ein Service aus, betrifft das nicht die anderen – die Events bleiben im Event-Store gespeichert, und der Service kann sie später nachholen.

Autonome Services statt zentraler Orchestrator

Event-getriebene Architekturen favorisieren dabei Choreografie gegenüber Orchestrierung. Bei Orchestrierung gibt ►

Listing 7: ObserveEventsAsync-Methode

```
public async Task RunProjectionsAsync(
    CancellationToken cancellationToken)
{
    var subscription = client.ObserveEventsAsync(
        "/books",
        new ObserveEventsOptions
        {
            Recursive = true // Alle Bücher-Events,
                           // unabhängig von der Book-ID
        },
        cancellationToken
    );

    await foreach (var evt in subscription)
    {
        // Events parallel an alle Projektionen
        // weiterleiten
        await Task.WhenAll(
            bookCatalogProjection.HandleAsync(evt),
            borrowingAnalyticsProjection.HandleAsync(evt)
        );
    }
}
```

es einen zentralen Service, der andere Services aufruft und den Prozessablauf steuert – eine Art Dirigent, der das Orchester leitet. Bei Choreografie reagieren Services autonom auf Events, ohne dass ein zentraler Koordinator den Ablauf vorgibt – wie Tänzer, die auf die Musik reagieren, ohne dass jemand ihre Schritte kommandiert.

Choreografie skaliert besser und ist resilient gegen Ausfälle einzelner Services. Der Preis: Das Gesamtverhalten des Systems ist nicht an einer zentralen Stelle definiert, sondern emergiert aus dem Zusammenspiel vieler autonomer Services. Das erfordert gutes Monitoring, klare Event-Contracts und ein Verständnis für verteilte Systeme. Doch für komplexe Domänen, in denen viele verschiedene Akteure interagieren, ist Choreografie oft die natürlichere und robustere Lösung.

CRUD als Anti-Pattern

Event-Sourcing zwingt uns, über unsere Domäne anders nachzudenken. CRUD reduziert die Welt, wie bereits eingangs erwähnt, auf vier technische Operationen: Create, Read, Update, Delete.

Diese Abstraktion ist verlockend einfach – aber sie ist auch eine Verarmung der fachlichen Realität. Die Geschäftsprozesse, die wir modellieren, sind reicher und nuancierter, als vier Verben es je ausdrücken können.

Betrachten wir nochmals eine Ausleihe in der Bibliothek. In einem CRUD-System würden wir schreiben: *UPDATE books SET status = 'available', borrowed_by = NULL WHERE id = 42*. Was sagt uns das? Technisch: Zwei Spalten wurden geändert. Fachlich: Fast nichts. Warum wurde das Buch als verfügbar gekennzeichnet? Wer hat das verursacht? Gab es zuvor Probleme? Wurde die Verfügbarkeit manuell freigegeben oder automatisch verarbeitet? All diese Informationen gehen verloren oder müssen mühsam in separaten Logging-Mechanismen nachgehalten werden.

Event-Sourcing dreht diese Perspektive um. Statt abstrakter technischer Operationen modellieren wir konkrete fachliche Ereignisse: *BookAcquired*, *BookBorrowed*, *BookReturned*, *LateFeeCharged*. Jeder Event trägt die Geschäftslogik in sich und spricht die Sprache der Fachlichkeit. Die Software wird dadurch nicht nur technisch anders – sie wird zu einer präziseren Abbildung der realen Geschäftsprozesse.

Das bedeutet nicht, dass CRUD grundsätzlich falsch ist. Für viele Anwendungen – Content-Management-Systeme, einfache Stammdatenverwaltung, Prototypen – kann CRUD genau richtig sein. Es ist schnell zu implementieren, einfach zu verstehen und für viele Szenarien ausreichend. Doch wo Nachvollziehbarkeit, Analysierbarkeit, komplexe Geschäftsprozesse oder regulatorische Anforderungen zentral sind, zeigt Event-Sourcing seine Stärken. Jede Anwendung, die mehr als nur simple CRUD-Operationen durchführt, profitiert von der reicheren Semantik und der historischen Nachvollziehbarkeit, die Event-Sourcing bietet.

Event-Sourcing ist mehr als ein Architekturmuster – es ist eine strategische Entscheidung darüber, wie wir mit Daten umgehen. Die Auswirkungen reichen weit über die unmittelbare Anwendung hinaus und eröffnen neue Möglichkeiten in

Bereichen wie Data-Analytics, maschinellem Lernen und unternehmensweiten Datenarchitekturen.

Events als Data-Products

In modernen Datenarchitekturen gewinnt das Konzept des Data-Mesh an Bedeutung. Data-Mesh ist ein Paradigma, das Daten als Produkte begreift, die von domänenspezifischen Teams bereitgestellt werden. Statt einer zentralen Data-Lake- oder Data-Warehouse-Lösung, in die alle Daten fließen, ist jedes Team für seine eigenen Data-Products verantwortlich – hochwertige, gut dokumentierte Datenquellen, die andere Teams nutzen können.

Events sind ideale Data-Products. Sie sind unveränderlich, semantisch reich, zeitlich geordnet und tragen fachlichen Kontext in sich. Jedes Team betreibt seinen eigenen Event-Store und publiziert relevante Events als Data-Products. Andere Teams – etwa das Analytics-Team, das Data-Science-Team oder Business-Intelligence-Anwendungen – können diese Events abonnieren und daraus ihre eigenen Projektionen oder Analysen aufbauen. Der Event-Store wird so zur Quelle der Wahrheit, aus der verschiedene spezialisierte Datensichten abgeleitet werden.

Diese Architektur ermöglicht es, Datenprodukte flexibel zu kombinieren und neue Anwendungsfälle zu erschließen, ohne die ursprünglichen Systeme ändern zu müssen. Ein neuer Anwendungsfall? Abonniere die relevanten Event-Streams und baue deine eigene Projektion. Die Entkopplung zwischen Datenerzeugern (den Domänen-Services) und Datenkonsumenten (Analytics, ML, BI) führt zu einer skalierbaren und evolvierbaren Datenlandschaft.

Events als Trainingsdaten für maschinelles Lernen

Künstliche Intelligenz und maschinelles Lernen sind hungrig nach Daten – und Event-Sourcing liefert genau die Art von Daten, die KI-Modelle brauchen. Events enthalten nicht nur Fakten, sondern auch zeitliche Abfolgen, Kausalitäten und Kontext. Ein Empfehlungsalgorithmus für unsere Bibliothek könnte aus den Events lernen: „Wenn ein Leser Buch A ausleiht und danach Buch B, ist die Wahrscheinlichkeit hoch, dass er auch Buch C mögen wird.“ Die Events dokumentieren das tatsächliche Verhalten – ohne dass Menschen mühsam Labels setzen oder Trainingsdaten kuratieren müssen.

Ein weiterer Vorteil: Event-Sourcing ermöglicht Zeitreisen. Wenn wir ein Modell mit historischen Daten trainieren wollen, spielen wir einfach die Events bis zu einem bestimmten Zeitpunkt ab und rekonstruieren den damaligen Zustand. Wir können Modelle mit Daten von vor zwei Jahren trainieren, ohne dass wir diese Daten separat vorgehalten haben – sie sind im Event-Store direkt verfügbar. Das macht A/B-Tests, Backtesting und historische Analysen trivial einfach.

Compliance, Audit und DSGVO

In regulierten Branchen – Finanzsektor, Gesundheitswesen, öffentliche Verwaltung – sind lückenlose Nachweise oft gesetzliche Pflicht. Event-Sourcing liefert diese Nachweise von Haus aus. Jede Zustandsänderung ist als Event dokumentiert, nichts kann unbemerkt geändert werden. Wer hat wann

was getan? Die Antwort liegt im Event-Store, vollständig und unveränderlich.

Auch die DSGVO mit ihrem „Recht auf Vergessenwerden“ lässt sich mit Event-Sourcing umsetzen, obwohl Events unveränderlich sind. Die Lösung: Personenbezogene Daten werden verschlüsselt gespeichert. Wenn ein Nutzer sein Recht auf Vergessenwerden ausübt, wird der Verschlüsselungsschlüssel gelöscht (Crypto-Shredding). Die Events bleiben strukturell erhalten, aber ihre Daten sind nicht mehr lesbar. Alternativ können personenbezogene Daten in Events anonymisiert werden, während die fachlichen Informationen erhalten bleiben.

Herausforderungen und Trade-offs

Event-Sourcing ist kein Allheilmittel. Es bringt enorme Vorteile, aber auch Komplexität, die nicht unterschätzt werden sollte. Teams, die Event-Sourcing einführen, müssen bereit sein, in neue Denkweisen, Werkzeuge und Prozesse zu investieren.

- **Erhöhte Komplexität und Lernkurve:** Das Denken in Events erfordert ein Umdenken. Entwickler müssen lernen, Geschäftslogik in Eventhandlern zu modellieren, mit Eventual Consistency umzugehen und verteilte Systeme zu verstehen. Die Fehlersuche wird anders – statt in Tabellen zu schauen, müssen wir Event-Streams analysieren. Das erfordert neue Tools und Fähigkeiten.
- **Event-Versionierung:** Events leben für immer. Wenn sich das Datenmodell ändert, müssen alte Events weiterhin verarbeitet werden können. Das erfordert Versionierungsstrategien: Upcasting (alte Events beim Lesen in neue Formate umwandeln), parallele Versionen (mehrere Event-Typen für denselben Vorfall), oder Event-Migration (alte Events in neue transformieren). Welche Strategie die richtige ist, hängt vom Anwendungsfall ab – aber ignorieren kann man das Thema nicht.
- **Operationale Komplexität:** Projektionen müssen gewartet, Subscriptions überwacht und Replays orchestriert werden. Wenn eine Projektion falsche Daten enthält, muss sie neu aufgebaut werden – das kann bei großen Event-Mengen dauern. Monitoring und Observability sind essenziell, um zu wissen, ob alle Projektionen aktuell sind und wo es Verzögerungen gibt. Das erfordert solides DevOps-Know-how und durchdachte Tooling-Strategien.
- **Tooling und Infrastruktur:** Event-Sourcing braucht spezialisierte Infrastruktur. Ein Event-Store ist kein Ersatz für eine relationale Datenbank, sondern eine zusätzliche Komponente. Tools wie EventSourcingDB helfen, die Einstiegshürde zu senken, doch die Lernkurve ist steiler als bei klassischen ORMs. Teams müssen bereit sein, in neue Tools zu investieren und ihre Entwicklungsprozesse anzupassen.

Diese Herausforderungen sind real, aber bewältigbar. Wichtig ist, Event-Sourcing nicht als Allzwecklösung zu betrachten, sondern als Werkzeug für spezifische Problemfelder: Wo Nachvollziehbarkeit, Analysierbarkeit und Fachlichkeit zentral sind, lohnt sich der Invest. Wo einfache CRUD-Operationen ausreichen, ist Event-Sourcing überdimensioniert.

Fazit: Event-Sourcing als Perspektivenwechsel

Event-Sourcing ist mehr als ein technisches Pattern – es ist eine andere Sichtweise auf Daten und Geschäftsprozesse. Statt „Wie sind die Dinge jetzt?“ fragen wir: „Was ist passiert?“ Diese Perspektive eröffnet neue Möglichkeiten: vollständige Nachvollziehbarkeit, flexible Auswertungen, klare Entkopplung in verteilten Systemen, semantisch reiche Datenmodelle.

In Kombination mit CQRS entsteht eine Architektur, die Writes und Reads optimal bedient und beide unabhängig voneinander skalieren lässt. Event-getriebene Microservices lassen sich daraus natürlich ableiten, und der Event-Store wird zur Grundlage für moderne Datenstrategien – von Data-Mesh über Analytics bis hin zu maschinellem Lernen. Die fachliche Semantik, die in Events steckt, macht sie zu wertvollen Datenprodukten, die weit über die ursprüngliche Anwendung hinaus genutzt werden können.

Für .NET-Entwickler, die in diese Welt einsteigen wollen, bietet EventSourcingDB einen pragmatischen Einstieg: Eine spezialisierte Datenbank, die von Grund auf für Event-Sourcing konzipiert wurde, ein idiomatisches .NET-Client-SDK, das sich nahtlos in moderne .NET-Anwendungen integriert, und eine klare Konzeption, die auf Standards wie Cloud-Events setzt. Die Bibliothek als Beispiel zeigt: Event-Sourcing ist weniger abstrakt, als es zunächst scheint. Es geht um Geschäftsvorfälle, die wir ernst nehmen – und um Software, die der Realität gerecht wird.

Nicht jede Anwendung braucht Event-Sourcing. Doch wo Nachvollziehbarkeit, Analysierbarkeit, Fachlichkeit und Evolvierbarkeit zentral sind, lohnt der Blick über den CRUD-Tellerrand. Event-Sourcing bietet nicht nur Antworten auf technische Fragen – es stellt die richtigen Fragen über die Natur unserer Daten und die Geschäftsprozesse, die wir abbilden wollen. Es ist ein Werkzeug für Teams, die bereit sind, in Qualität und Langlebigkeit zu investieren – und die erkannt haben, dass gute Software mehr ist als nur der aktuelle Zustand einer Tabelle. ■

[1] EventSourcingDB, www.eventsourcingdb.io

[2] CloudEvents-Standard, <https://cloudevents.io>

[3] Dokumentation von EventSourcingDB, <https://docs.eventsourcingdb.io>

[4] NuGet-Paket EventSourcingDb, www.nuget.org/packages/EventSourcingDb

[5] CQRS.com, www.cqrs.com

[6] Eventsourcing.ai, www.eventsourcing.ai

[7] The Definitive Event Sourcing Database Guide, www.eventsourcingdatabase.com



Golo Roden

ist Gründer und CTO von the native web GmbH. Er berät Unternehmen zur Konzeption und Entwicklung von Web- und Cloud-Anwendungen mit Schwerpunkt auf Event-getriebenen und Service-basierten Architekturen sowie DDD. www.thenativeweb.io

PLATTFORMÜBERGREIFEND ENTWICKELN MIT .NET UND XAML

UI-Gestaltung mit Uno

Architektur, Ökosystem, UX-/UI-Strategien und Tooling der Uno Platform.

Moderne Anwendungen müssen heute auf einer Vielzahl von Endgeräten laufen (Desktop, Web und Mobile). Die Uno Platform [1] ermöglicht genau das: Sie setzt auf das WinUI-basierte XAML/C#-Modell von Microsoft und macht es auf iOS, Android, macOS, Linux und WebAssembly nutzbar. Entwickler können so aus einer einzigen Codebasis heraus native, leistungsfähige Apps erstellen.

Dieser Artikel erklärt, warum Uno für Cross-Plattform-UIs sinnvoll ist, wie die Architektur funktioniert, welche UX-/UI-Konzepte es unterstützt, welche Tools und Libraries es im Ökosystem gibt, und zeigt zudem praxisnahe Codebeispiele.

Uno für plattformübergreifende UIs

Die Uno Platform wurde als Open-Source-.NET-Plattform für Mobile-, Web-, Desktop- und Embedded-Apps mit einer gemeinsamen Codebasis entwickelt. Dabei übernimmt Uno das gesamte WinUI- beziehungsweise UWP-XAML-API und macht es auf allen unterstützten Plattformen lauffähig. Bestehende WinUI- oder UWP-Steuerelemente lassen sich oft eins zu eins weiterverwenden, manchmal müssen sie lediglich neu kompiliert werden. In Kombination mit C# ergibt sich ein hoher Wiederverwendungsgrad: Bis zu 99 Prozent von Geschäftslogik und UI-Definition lassen sich laut Uno-Dokumentation beibehalten. Entwicklerteams profitieren dadurch von ihren Kenntnissen in XAML/C# und den Tools des Microsoft-Ökosystems (Visual Studio, VS Code, Rider et cetera).

Außerdem wird Uno als pixelgenau (Pixel-perfect [2]) beworben, was bedeutet, dass unabhängig vom Betriebssystem dieselben Layout- und Zeichenbefehle genutzt werden, um gleiche Optik und Benutzerführung zu gewährleisten. Gleichzeitig kann Uno auf jeder Plattform die nativen Steuerelemente verwenden oder über die zentrale Skia-Engine rendern. Die Plattform ist also flexibel: Sie liefert überall native Performance und Aussehen, während nur eine Codebasis ge-

pfligt werden muss. Zu den weiteren Features zählen integrierte Themen-/Designsysteme: Mit den Uno-Themes [3] lassen sich Apps gezielt an Plattform-Guidelines anpassen. Zum Beispiel sind via NuGet-Komponente ein Material-Design für Android-UI sowie ein Cupertino-Design für iOS verfügbar, falls man den nativen Look bevorzugt. Wer plattformübergreifend entwickeln muss und Windows/XAML-Know-how hat, findet in der Uno Platform deshalb eine produktive, aktuelle Lösung mit folgenden Kernfeatures:

- **Ein Code, viele Plattformen:** Einfache Wiederverwendung von WinUI-Code.
- **Microsoft-Ökosystem:** Nutzung von C#, .NET 8+, Visual Studio, .NET-MAUI- oder Community-Toolkit-Bibliotheken.
- **Pixel-perfect und nativ:** Einheitliche UIs über Skia-Rendern oder native UI-Frameworks.
- **Open Source:** Ohne Lizenzkosten, mit wachsender Community; bezahlt werden muss nur für das erweiterte Tooling namens Hot Design.

Installation, IDEs und Toolchains

Die Entwicklung mit der Uno Platform ist auf allen gängigen Betriebssystemen möglich, das heißt unter Windows, macOS und Linux. Wichtig ist lediglich eine moderne .NET-Version (ab .NET 8, empfohlen wird .NET 9) sowie eine unterstützte IDE oder ein Editor. Folgende Installationen sind notwendig:

- **.NET SDK:** Uno Platform unterstützt .NET 8+; am besten installiert man das aktuelle .NET SDK.
- **Uno Templates [3]:** Mit `dotnet new install Uno.ProjectTemplates` erhält man alle Uno-spezifischen Projektvorlagen. Damit lässt sich mit einem Befehl wie `dotnet new unoapp -o MyApp` eine neue App für alle Zielplattformen generieren.
- **NuGet-Pakete:** Abhängig vom Szenario fügt man Toolkit/Theme/Extensions-Pakete [4] hinzu, zum Beispiel Uno.Toolkit.UI, Uno.Extensions.Hosting, UnoThemes.Material.
- **Pooling und Builds:** Uno unterstützt das Pooling von Ressourcen, das heißt eine gemeinsame Build- und Asset-Pipeline für alle Plattformen. Icons, Styles, Fonts und XAML-Ressourcen werden nur einmal gepflegt und beim Build für jede Zielplattform verteilt. Damit reduziert sich der Wartungsaufwand enorm.

Die Uno Platform ist offen gestaltet. Unterstützt werden unterschiedliche Entwicklungsumgebungen (IDEs) und Editoren:

- **Visual Studio 2022 (Windows):** Vollintegration mit Uno-Templates, Emulatoren (Android/iOS), Debugging und Hot Reload/Hot Design.
- **Visual Studio Code (Windows, Mac, Linux):** Flexibler, leichtgewichtiger Editor. Dank offizieller Uno- und C#-Erweiterun-

• Du möchtest gerne mehr erfahren?

Das freut uns, denn ab Erscheinen des vorliegenden Artikels veröffentlichen wir auf developer-world.de/dwx-insights weitere Beiträge, die den Artikel fortsetzen und ausschließlich online zur Verfügung stehen. Derzeit geplant sind folgende Beiträge:

- Layouts, Grids und responsive Gestaltung
- Designsysteme und Figma-Integration
- Adaptive UIs
- Von der Idee bis zur Uno-App

gen unterstützt er XAML-IntelliSense, Hot Reload und Figma-Integration.

- **JetBrains Rider:** Wird ebenfalls unterstützt, mit XAML-Preview und Debugging.
- **Uno Platform Studio:** Als übergreifende Ergänzung, um Hot Reload/Hot Design in allen Umgebungen zu erlauben.

Eine typische Toolchain für das Entwickeln plattformübergreifender Apps unter Windows sieht so aus:

- **Betriebssystem** Windows 11.
- **IDE Visual Studio 2022** mit den Workloads *.NET Multi-platform App UI development* und *Desktop development with C++*.
- **Uno Platform Solution Templates** über NuGet.
- **Uno Platform Studio** (Hot Reload/Hot Design).

Die Vorteile dieser Kombination der Entwicklungstools sind die Nutzung von Visual Studio, ein gutes Debugging-Erlebnis auf allen Plattformen und die Vertrautheit der Tools für viele Entwickler.

Eine alternative Toolchain setzt auf Visual Studio Code (VS Code), welches für Windows, macOS und Linux zur Verfügung steht:

- **Betriebssystem** Windows, Linux oder macOS; installiertes .NET SDK.
- **VS Code mit folgenden Erweiterungen:** *C# Dev Kit* (Microsoft) für IntelliSense und Debugging.
- **Uno Platform for VS Code** für Templates, Uno Studio sowie Hot Reload.
- **Uno-Figma-Plug-in** für Design-to-Code-Integration.
- **XAML Styler** für saubere XAML-Struktur, optional.
- **Build & Run:** per CLI (*dotnet build*, *dotnet run*) oder direkt im Editor.

Die Vorteile eines Editors wie Visual Studio Code liegen in einer leichtgewichtigen Nutzbarkeit und darin, dass dieser auf jedem Betriebssystem funktioniert.

Der Autor erlaubt sich an dieser Stelle eine Empfehlung: Wer sich schnell in bestehenden WinUI/XAML-Stacks zu Hause fühlen möchte, nutzt Visual Studio auf Windows. Entwickler, die plattformunabhängig entwickeln möchten, beispielsweise auch auf unterschiedlichen Betriebssystemen im Team (macOS/Linux), nutzen VS Code.

Für beide Varianten ist Uno Platform Studio das geeignete Tool, um Benutzeroberflächen teilweise interaktiv zu gestalten (Hot Design).

Architektur: WinUI-Paradigma auf allen Plattformen

Unter der Haube implementiert Uno das komplette WinUI/WinRT-API auf Nicht-Windows-Systemen mit den Paketen *Uno.WinUI* und *Uno.WinRT*. Entwickler schreiben ihr UI weiterhin in gewohnter XAML-Syntax oder als C#-Markup, und während des Kompilierens (Build Time) wandelt Uno den XAML-Code per Roslyn-Generator in C#-Code um. Auf diese Weise erhält man zur Laufzeit ein natives UI-Framework für jedes Zielsystem.

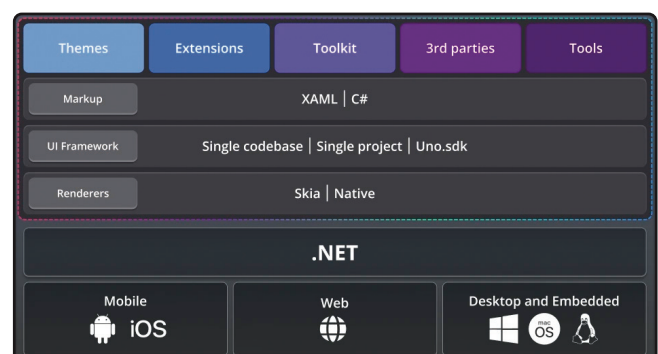
Im Rendering unterstützt Uno zwei Modi: Native und Skia. Im nativen Modus werden auf jeder Plattform die Standard-UI-Elemente verwendet (zum Beispiel *UIView* auf iOS, *ViewGroup* auf Android und DOM-Elemente im Web). Dadurch bekommt man die volle Plattform-Integration und Zugänglichkeit.

Im Skia-Modus rendert Uno dagegen alles auf eine einzige Hardware-beschleunigte Canvas (unterstützt Metal, OpenGL oder WebGL). Dieser Modus liefert auf allen Plattformen exakt dasselbe Pixel-Ergebnis und ist sehr performant, besonders bei vielen dynamischen UI-Elementen. In Projektvorlagen ist Skia aktiv, sofern man diese Voreinstellung nicht ändert.

Die hohe Kompatibilität zeigt sich auch im Build-System: Uno-Projekte nutzen .NET 8+ (Mobil, Desktop, WebAssembly) sowie MSBuild und automatisieren dabei die Transformation von Assets und Ressourcen für jedes Zielsystem. Zusätzlich erlaubt Uno das Einbinden nativer Komponenten über APIs. **Bild 1** zeigt einen Blick auf die Architektur der Uno Platform.

UX und UI: Barrierefreiheit, responsive Patterns

Auch die Benutzererfahrung sollte plattformübergreifend konsistent sein. Die Uno Platform greift hier auf die Mechanismen von WinUI zurück und erweitert sie für alle Plattformen. So ist Accessibility (Barrierefreiheit) integriert. *Uno.UI* implementiert einen großen Teil des UI-Automation-APIs von WinUI, sodass Screenreader und andere Hilfstechnologien



High-Level-Architektur der Uno Plattform (**Bild 1**)

auf allen Systemen funktionieren. Beispielsweise werden in XAML angegebene Eigenschaften wie *AutomationProperties.Name* oder *AutomationId* zur Laufzeit in die passenden Plattform-Felder übernommen (Android: *ContentDescription*, iOS: *AccessibilityIdentifier*, Web: *aria-label*). So kann man dieselben XAML-Attached-Properties nutzen, um Elemente beschreibbar und bedienbar zu machen – eine wichtige Voraussetzung für zugängliche Apps.

Beim responsiven Design nutzt Uno alle aus WinUI bekannten Techniken. Mit *AdaptiveTrigger* und *VisualStateManager* kann XAML Layouts dynamisch an verschiedene Fenstergrößen oder Orientierungen anpassen. Zusätzlich bietet das Uno-Toolkit spezielle Controls und Layouthilfen für ein responsives Design. So ermöglicht der *ResponsiveView-Container*, unterschiedliche XAML-Templates je nach Bildschirmbreite zu definieren. Das Steuerelement passt sich durch ►

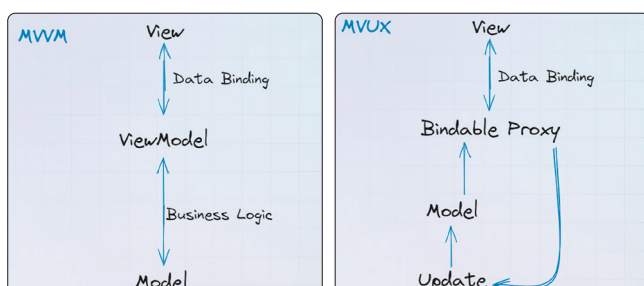
die dynamische Auswahl des richtigen Templates an unterschiedliche Bildschirmgrößen an. Dabei werden die aktuelle Bildschirmbreite und die definierten Vorlagen betrachtet. Da nicht alle Vorlagen einen Wert benötigen, sorgt das Steuerelement für eine reibungslose Benutzererfahrung, indem es die kleinste definierte Vorlage auswählt, die die Anforderungen an die Breite erfüllt. Wird keine Übereinstimmung gefunden, wird die größte definierte Vorlage verwendet. Ein einfaches Beispiel mit responsiven Layouts sieht wie folgt aus:

```
xmlns:utu="using:Uno.Toolkit.UI"
<utu:ResponsiveView>
  <!-- note: -->
  <utu:ResponsiveView.NarrowTemplate>
    <DataTemplate>
      <!-- Narrow content -->
    </DataTemplate>
  </utu:ResponsiveView.NarrowTemplate>
  <utu:ResponsiveView.WideTemplate>
    <DataTemplate>
      <!-- Wide content -->
    </DataTemplate>
  </utu:ResponsiveView.WideTemplate>
</utu:ResponsiveView>
```

Trotz plattformübergreifender Codebasis bleibt das UI konsistent – dank des pixelgenauen Renderings und optisch einheitlicher Themes. Schriften und Abstände erscheinen überall gleich. Gleichzeitig lässt Uno das Aneinanderreihen nativer UX-Konventionen zu: Man kann etwa für iOS automatisch Cupertino-Icons verwenden oder systembedingte Abstände wählen. Die Themes-Bibliothek stellt Material-, Fluent- und Cupertino-Stile bereit. Entwickler sind auf diese Weise flexibel: Sie behalten gemeinsame UI-Logik, können aber Design-Abweichungen pflegen, um den Benutzererwartungen jeder Plattform gerecht zu werden.

Ökosystem

Ein umfassendes Ökosystem ergänzt die Plattform. Klassische MVVM-Muster funktionieren in Uno genauso wie in jeder anderen auf XAML basierenden App. Man kann vertraute Frameworks wie das CommunityToolkit.Mvvm einsetzen, um ViewModels mit *INotifyPropertyChanged* und *ICommand* zu verknüpfen. Der ViewModel-Code etwa lässt sich mit dem Community-Toolkit nach folgendem Muster schreiben:



Datenfluss von MVVM versus MVUX (Bild 2)

```
public partial class MainViewModel : ObservableObject
{
    [ObservableProperty]
    private WeatherInfo _currentWeather;
    public MainViewModel(IWeatherService weather)
    {
        _weather = weather;
        LoadWeather();
    }
    // ...
}
```

Eine neuere Option im Uno-Ökosystem heißt Model-View-Update-eXtended, kurz MVUX [5]. Dieses vom Uno-Team entwickelte Muster verbindet die Vorzüge von MVU (einfacher Datenfluss, Immutability) mit dem Data Binding von MVVM. MVUX ermutigt zur Verwendung von Immutable Records und bietet automatische ViewModel-Generierung mittels Source-Generatoren. So ergeben sich deklarative, nebenläufigkeitsfreundliche Zustandsaktualisierungen bei gleichzeitigem XAML-Binding-Support. In der Praxis schreibt man einfach unveränderliche (immutable) Datentypen und Methoden zur Zustandsänderung, und Uno erzeugt das nötige ViewModel. Dieser MVUX-Ansatz adressiert komplexe Szenarien mit vielen asynchronen Operationen besonders elegant. Für Einsteiger bietet sich MVVM weiterhin an; erfahrene Entwickler komplexer Apps profitieren von MVUX. Einen Vergleich des Datenflusses zwischen den Schichten bei MVVM und bei MVUX zeigt Bild 2.

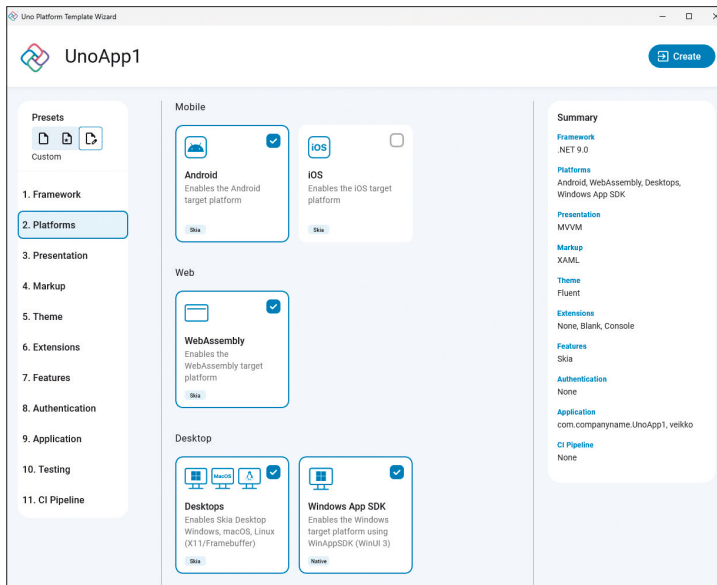
Für Dependency Injection und Hosting stellt Uno ebenfalls Infrastruktur bereit. Die Bibliothek *Uno.Extensions* integriert das Paket *Microsoft.Extensions.Hosting*, sodass sich in der Datei *App.xaml.cs* ein *HostBuilder* verwenden lässt. Dienste und Services (etwa für Networking, Datenbanken et cetera) registriert man mit *ConfigureServices* über *AddSingleton* oder ähnliche Methoden. So werden ViewModels oder Services einfach per Konstruktor-Injektion aufgelöst. Hier ein Beispiel dazu:

```
var builder = this.CreateBuilder(args)
    .UseStudio() // Neu: aktiviert Hot Reload/
                // Hot Design bei .NET 9+

.ConfigureServices((ctx, services) =>
{
    services.AddSingleton<IWeatherService,
        WeatherService>();
    services.AddSingleton<MainViewModel>();
});
var host = builder.Build();
```

Das ist analog zu üblichen .NET-Anwendungen. Nach dem Build ruft man beispielsweise folgenden Befehl auf, um ViewModel-Instanzen zu beziehen:

```
DataContext = host.Services.GetRequiredService<
    MainViewModel>();
```

Template-Wizard zum Erstellen von Uno-Apps (Bild 3)

Wichtig: Ab Uno.Sdk 5.5+ sollte man *UseStudio()* anstelle von *EnableHotReload()* nutzen, um Hot-Reload-Funktionen einzuschalten.

Eine wichtige Ergänzung ist auch die Bibliothek *Uno.Toolkit.UI* (NuGet *Uno.Toolkit.UI* beziehungsweise *Uno.Toolkit.WinUI*), die erweiterte Controls bereitstellt. Neben Standard-Layouts findet man dort Material-Design-Elemente wie *Card*, *Chip* und *TabBar* sowie mobile Navigationselemente wie *NavigationBar*, *DrawerControl* oder *TabBarItem*. Die Toolkit-Komponenten sind für responsives Verhalten optimiert und lassen sich im XAML-Code wie normale Controls verwenden.

Zusätzlich sorgt *Uno.Themes* dafür, dass Material- beziehungsweise Cupertino- und Fluent-Styles leicht angewendet werden können. Per NuGet fügt man *Uno.Themes.Material* oder *Uno.Themes.Cupertino* hinzu, um gebrauchsfertige *ResourceDictionaries* mit Farb- und Control-Styles zu laden. Damit passen sich Steuerelemente automatisch an die Plattformrichtlinien an.

Kurz: Die Uno-Plattform hat eine umfassende Toolchain und Bibliotheken – von UI-Controls über Muster (MVVM/MVUX) bis zu Hilfs-Frameworks (DI, Navigation) –, die den Entwickleralltag erleichtern und den Fokus auf die eigentliche App-Logik legen.

Hot Reload, Hot Design, Uno Platform Studio

Die Entwicklung mit Uno kann durch spezialisierte Werkzeuge stark beschleunigt werden. Ein zentrales Element ist Uno Platform Studio. Das ist ein Satz von Tools und Erweiterungen für Visual Studio, VS Code, Rider et cetera, der vor allem Hot Reload und das neue Hot Design kombiniert:

Hot Reload: Wie in .NET allgemein ermöglicht Uno Hot Reload, dass Änderungen in XAML oder C# sofort in der laufenden App sichtbar werden, ohne Neugenerierung oder Neustart. Uno Hot Reload unterstützt dabei alle Plattformen (iOS, Android, WebAssembly, Skia-Targets) und gängige IDEs (Visual Studio 2022, VS Code, Rider). Entwickler können zum

Beispiel während des Debuggens den Button mit dem Flammensymbol klicken oder die Tastenkombination [Alt]+[F10] drücken und sehen ihre UI-Modifikationen unmittelbar. Laut Uno-Dokumentation können fast alle UI-Änderungen live übernommen werden, das heißt XAML-Dateien, Styles, DataTemplates, Bindings, *x:Bind*-Ausdrücke und mehr. Hot Reload ist tief in die Plattform integriert: Unter .NET 9+ genügt in der Datei *App.xaml.cs* bereits der Aufruf von *UseStudio()*, um Hot Reload und Hot Design zu aktivieren. Eine visuelle Anzeige im Fenster (ein kleines Blitz-Symbol) informiert über erfolgreiche Aktualisierungen.

Hot Design: Dieses völlig neue Feature macht die Anwendung zur Laufzeit selbst zum Designer. Man kann eine laufende App „einfrieren“ und visuell editieren – ähnlich wie in einem WYSIWYG-Editor –, während sie auf einem beliebigem Gerät geöffnet ist. Laut Uno-Team verwandelt Hot Design die App live in einen visuellen Editor, der Änderungen in Echtzeit synchron mit dem XAML-Code durchführt. Konkret hat man mit einem Klick im Editor die aktuelle Ansicht vor sich, kann UI-Elemente positionieren, Styles ändern oder neue Controls hinzufügen und sieht die Wirkung sofort auf dem Device (der Zustand bleibt erhalten). Jede Anpassung wird direkt in den Code zurückgeschrieben. Es entsteht also ein bidirektionaler Flow zwischen Code und Designer.

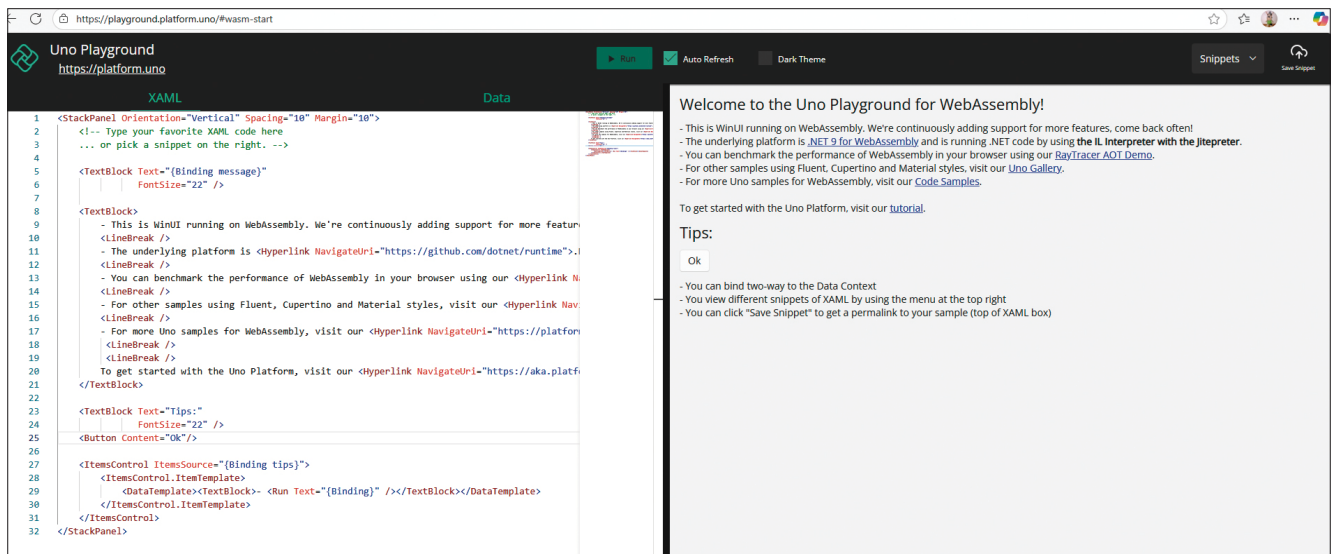
Hot Design unterstützt alle gängigen IDEs und Betriebssysteme (Visual Studio, VS Code, Rider und Windows, macOS, Linux). Es ist ab .NET 9 mit XAML verfügbar und läuft auf allen Plattformen außer WinAppSDK. Vorteil: Einmalige Änderungen lassen sich sofort testen, auch mit echten Daten anstelle statischer Mockups. Uno Platform Studio bietet zudem ein Design-to-Code-Plug-in für Figma: Designer können Benutzeroberflächen in Figma entwerfen und per Plug-in direkt nach XAML- beziehungsweise C#-Code exportieren. So wird der Roundtrip zwischen UI-Design und Code stark vereinheitlicht.

In der Praxis bedeutet das für Entwickler, dass man unmittelbar Feedback bei der UI-Erstellung bekommt und schnellere Iterationen möglich werden. Der kombinierte Einsatz von Hot Reload und Hot Design verkürzt Debug-Zyklen erheblich. Darüber hinaus bietet das Uno-Toolset zahlreiche weitere Helfer: einen Template-Wizard (Bild 3) in Visual Studio für neue Projekte (inklusive Einbindung von Toolkit/Theme-Features), Kommandozeilen-Tools (*dotnet new unoapp -o MyApp*) und einen umfangreichen Uno Playground [6], um XAML-Code direkt im Browser auszuprobieren (Bild 4).

All das ergibt zusammen mit der Figma-Integration und klassischen Debugging-Features ein sehr modernes Toolset für die plattformübergreifende .NET-Entwicklung.

Snippets in XAML und C#

Nach all den theoretischen Grundlagen folgen nun einige Code-Schnipsel, die den praktischen Umgang mit Uno zeigen. Listing 1 codiert eine XAML-Seite mit einem Button, die plattformübergreifend überall funktioniert. Ob man diese Seite nun in iOS, Android, im Browser (WebAssembly) oder auf Windows lädt: Layout und Verhalten bleiben gleich. ►



Uno-Playground zum Testen von XAML-Code (Bild 4)

Uno übersetzt das `<Button>`-Element intern in den nativen Button der Plattform oder zeichnet es über Skia.

Im zweiten Beispiel wird ein adaptives Layout aufgebaut. Wir erstellen ein Navigationssystem, das auf großen Bildschirmen als Navigationsleiste angezeigt wird und auf kleinem Bildschirm durch eine TabBar ersetzt wird (Listing 2).

Dieses Muster schaltet je nach Fensterbreite zwischen zwei Layout-Alternativen um. Mit dem Uno Toolkit kann man stattdessen auch das *ResponsiveView*-Control verwenden:

```
<uto:ResponsiveView>
  <uto:ResponsiveView.NarrowTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock
          Text="Ansicht für enge Bildschirme"/>
        </StackPanel>
      </DataTemplate>
    </uto:ResponsiveView.NarrowTemplate>
    <uto:ResponsiveView.WideTemplate>
      <DataTemplate>
        <StackPanel><TextBlock
          Text="Ansicht für breite Bildschirme"/>
        </StackPanel>
      </DataTemplate>
    </uto:ResponsiveView.WideTemplate>
  </uto:ResponsiveView>
```

Je nach Bildschirmbreite wählt Uno selbst das passende Template aus – ideal für eine plattformübergreifende UX.

Ein weiterer Quellcodeausschnitt betrachtet das Einbinden von Services via Dependency-Injection in C#-Code in der Datei *App.xaml.cs*:

```
public App()
{
    this.InitializeComponent();
```

Listing 1: Einfaches Beispiel

```
<!-- MainPage.xaml -->
<Page
  x:Class="MyApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/
    xaml/presentation"
  xmlns:x=
    "http://schemas.microsoft.com/winfx/2006/xaml">

  <StackPanel HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <TextBlock Text="Willkommen bei Uno!"
      FontSize="24" Margin="0,0,0,20"/>
    <Button Content="Klick mich!" Click=
      "Button_Click" Padding="10,5"/>
  </StackPanel>
</Page>

// MainPage.xaml.cs
public partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }
    private void Button_Click(object sender,
      RoutedEventArgs e)
    {
        // Diese Logik läuft auf jeder Plattform
        Content = new TextBlock {
          Text = "Der Button wurde geklickt!",
          FontSize = 18, Padding="10" };
    }
}
```

Listing 2: Adaptives Layout

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="WideView">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="720"/>
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="NavMenu.Visibility"
          Value="Visible"/>
        <Setter Target="TabBar.Visibility"
          Value="Collapsed"/>
      </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="NarrowView">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0"/>
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="NavMenu.Visibility"
          Value="Collapsed"/>
        <Setter Target="TabBar.Visibility"
          Value="Visible"/>
      </VisualState.Setters>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

```
var hostBuilder = this.CreateBuilder(args)
    .UseStudio() // aktiviert HotReload/HotDesign
    .ConfigureServices((context, services) =>
    {
        services.AddSingleton<IWeatherService,
            WeatherService>();
        services.AddSingleton<MainViewModel>();
    });
var host = hostBuilder.Build();
host.Services.GetRequiredService<
    MainViewModel>(); // startet ViewModel-Lebenszyklus
// ...
}
```

In der Datei *MainPage.xaml.cs* könnte man das *MainViewModel* dann per Konstruktor oder *DataContext* verwenden. Dieses Muster ist identisch zu anderen .NET-Anwendungen.

Ein Uno-Beispiel

Zum Abschluss dieses Artikels soll eine minimalistische To-do-App lokal definierte Aufgaben anzeigen, die der Benutzer abhaken kann. Sie bietet folgende Basisfunktionen:

- **Eine Überschrift** (*Meine Aufgabenliste*).
- **Eine Aufgabenliste** mit Checkboxen zum Abhaken.
- **Ein Eingabefeld** für neue Aufgaben als Platzhalter.

Die App soll man auf Desktop, Smartphones sowie im Browser ausführen können. Angelegt wird das neue Projekt über den Visual-Studio-Projektassistenten. Dabei sind folgende Schritte erforderlich:

- **Visual Studio** starten,
- **Menü:** *Datei | Neues Projekt ...*,
- **Vorlage auswählen:** *Uno Platform App*,
- **Projektname:** *MyToDoApp*,
- **Zielpattformen auswählen:** Windows (WinUI), WebAssembly, Android und iOS.

Visual Studio erzeugt dann eine Lösung mit einer vollständigen Multi-Plattform-Struktur wie in **Bild 5** mit:

- **MyToDoApp:** Shared-Projekt mit XAML und C#,
- **MyToDoApp.WinUI:** Windows Desktop,
- **MyToDoApp.Wasm:** WebAssembly,
- **MyToDoApp.Droid:** Android,
- **MyToDoApp.iOS:** iOS/iPadOS.

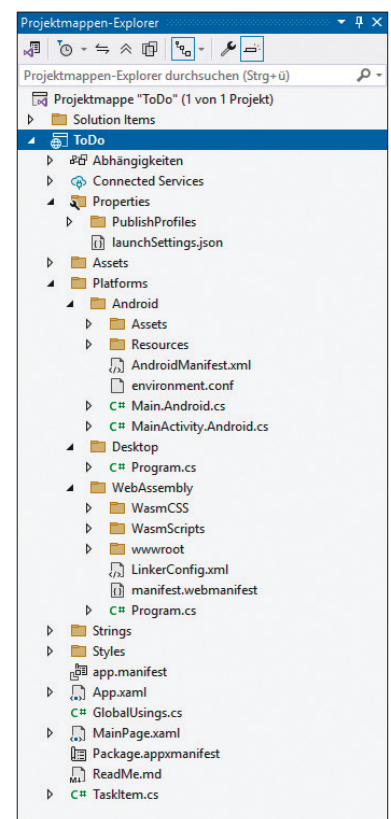
Damit steht eine vollständige Multi-Plattform-Struktur bereit. Im nächsten Schritt wird die Benutzeroberfläche per XAML-Code beschrieben, die in der Datei *MainPage.xaml* abgelegt wird. **Listing 3** zeigt den Code dazu. Und in **Listing 4** sehen Sie den C#-Code des Beispiels mit der Geschäftslogik, welche in der Datei *MainPage.xaml.cs* abgelegt wird.

Dieses minimale Beispiel kann man auf den unterschiedlichen Devices ausführen, zum Beispiel als App für den Windows-Desktop, als Web-App für den Browser und als Mobile-App für Android und iOS auf einem physischen Device beziehungsweise Simulator.

Wichtig: Geschäftslogik und UI bleiben dabei überall identisch, nur das Host-Projekt beziehungsweise die Zielpattform ändert sich.

Beispiel: Erweiterung

Nachdem das minimale To-do-Beispiel fertig ist, betrachten wir nun das etwas umfangreichere, offizielle To-do-Referenzbeispiel aus dem *Uno.Samples-Repository* [7]. Diese To-do-App präsentiert sich in einem modernen Look (Material Design 3) und bietet alle grundlegenden Funktionen einer Aufgabenverwaltung, das heißt Nutzer-Login, Anlage von Aufgaben mit Fälligkeitsdatum und eine Erinnerungsfunktion. Die An- ►



Projektstruktur der To-do-App (**Bild 5**)

Listing 3: XAML-Code der MyToDoApp

```
<Page
  x:Class="ToDo.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/
    xaml/presentation"
  xmlns:x=
    "http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ToDo">

  <StackPanel Padding="20" Spacing="10">
    <!-- Überschrift -->
    <TextBlock Text="Meine Aufgabenliste"
      FontSize="24" FontWeight="Bold" />

    <TextBox x:Name="NewTaskTextBox"
      PlaceholderText="Neue Aufgabe hinzufügen" />

    <Button Content="Hinzufügen"
      Click="AddTask_Click"/>

    <!-- Aufgabenliste -->
    <ListView ItemsSource="{Binding Tasks}">
      <ListView.ItemTemplate>
        <DataTemplate>
          <CheckBox Content="{Binding Description}"
            IsChecked="{Binding IsDone, Mode=TwoWay}" />
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
  </StackPanel>
</Page>
```

wendung ist vollständig plattformübergreifend ausgelegt. Das identische UI läuft als WebAssembly-App im Browser, auf dem Desktop und als App auf Mobilgeräten.

Das Projekt folgt dem klassischen MVVM-Pattern: Die Ordnerstruktur trennt Models, ViewModels und Views (XAML-Pages). In den Dateien, welche die Views enthalten, etwa *TaskListPage.xaml* oder *WelcomePage.xaml*, werden die Bindings zu den ViewModels definiert. Die Anwendung verwendet einen zentralen Navigationsservice zum Wechseln zwischen Seiten – etwa von der Aufgabenliste zur Detail- oder Einstellungsseite.

Die Hauptseite zeigt eine Liste aller To-do-Einträge an. Per Command-Bindung können Nutzer neue Aufgaben hinzufü-

gen oder vorhandene bearbeiten und löschen. Zum Beispiel wird ein Löschen-Command im ViewModel wie folgt definiert:

```
public ICommand DeleteTaskCommand =>
    new RelayCommand<TaskItem>(task =>
    {
        _tasks.Remove(task); // entfernt die Aufgabe aus
                             // der Liste
    });
```

Dieses Command wird im XAML-Code an einen Button gebunden, sodass ein Listeneintrag gelöscht werden kann. Damit weist der Button das Command an, das jeweils ausge-

Listing 4: Geschäftslogik der MyToDoApp

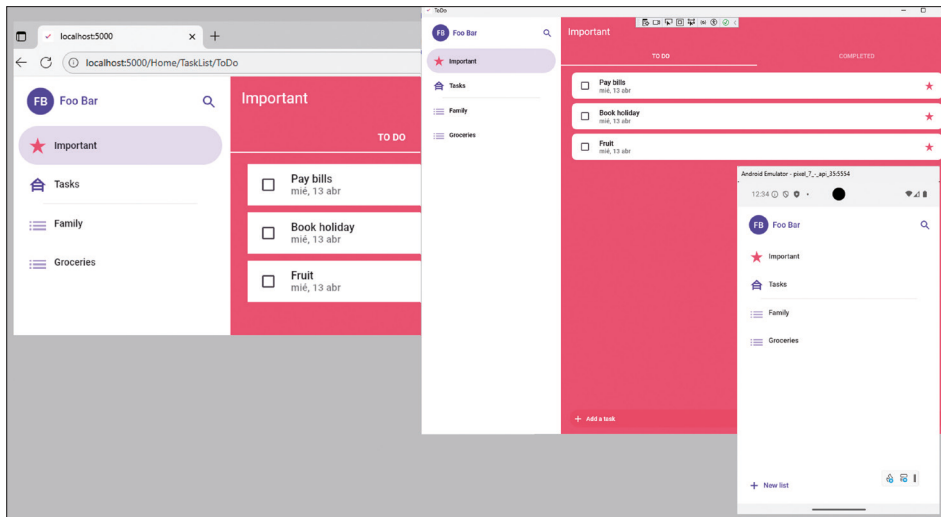
```
using System.Collections.ObjectModel;
namespace ToDo;

public sealed partial class MainPage : Page
{
    public ObservableCollection<TaskItem> Tasks {
        get; set; } = new();
    public MainPage()
    {
        this.InitializeComponent();
        // Statische Aufgaben hinzufügen
        Tasks.Add(new TaskItem { Description =
            "Einkaufen gehen", IsDone = false });
        Tasks.Add(new TaskItem { Description =
            "Projekt fertigstellen", IsDone = false });
        Tasks.Add(new TaskItem { Description =
            "Müll rausbringen", IsDone = false });

        // Binding-Kontext setzen
        this.DataContext = this;
    }

    private void AddTask_Click(object sender,
        RoutedEventArgs e)
    {
        var text = NewTaskTextBox.Text?.Trim();

        if (!string.IsNullOrEmpty(text))
        {
            Tasks.Add(new TaskItem { Description = text,
                IsDone = false });
            NewTaskTextBox.Text = "";
        }
    }
}
```

Beispiel-App (ToDo, extended) im Web, auf dem Desktop und als mobile App (Bild 6)

wählte *TaskItem* zu entfernen. Ähnlich funktionieren Befehle zum Speichern oder Aktualisieren von Aufgaben.

Die UI-Elemente sind über XAML-Bindings direkt mit den Eigenschaften im ViewModel verknüpft. Als Beispiel dient hier ein einfaches Textfeld für eine neue Aufgabe, mit bidirektionaler Bindung:

```
<TextBox Text=
    "{Binding NewTaskTitle, Mode=TwoWay}"
    PlaceholderText="Neue Aufgabe..." />
```

Hier bindet *NewTaskTitle* eine ViewModel-Eigenschaft (*INotifyPropertyChanged*), sodass Änderungen im UI automatisch ins ViewModel übernommen werden und umgekehrt. Die Aufgabenliste (*ListView*) bindet an eine *ObservableCollection*:

```
<ListView ItemsSource="{x:Bind ViewModel.Tasks}">
    <ListView.ItemTemplate>
        <DataTemplate x:DataType="models:TaskItem">
            <TextBlock Text="{x:Bind Title}" />
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Die Verwendung von Bindings ermöglicht es, Datenströme einfach zu verknüpfen. So aktualisiert sich die Anzeige automatisch, wenn das ViewModel die Aufgabendaten ändert.

Die App nutzt die Navigation, um zum Beispiel nach dem Login zur Aufgabenliste zu gelangen oder um von der Liste zur Detailansicht einer Aufgabe zu springen. Zum Beispiel ruft *Navigate("TaskDetailPage", task.Id)* eine neue Seite auf (*TaskDetailPage*), übergibt dabei die ID der ausgewählten Aufgabe und zeigt deren Details an. Dieses Muster trennt die Navigation vom UI, da der ViewModel-Befehl unabhängig von der View die Methode ausführt.

Die Beispiel-App wird im Web als WebAssembly-Anwendung geladen, auf Desktop-Betriebssystemen wie Windows oder macOS läuft sie als native Applikation und auf Android/

iOS nutzt sie die nativen Steuerelemente (Bild 6). Dank adaptiver Layouts und responsiver XAML-Elemente passen sich die UI-Controls automatisch an Touch- oder Maussteuerung, unterschiedliche Bildschirmgrößen und Auflösungen an. In allen Umgebungen zeigt die Anwendung eine moderne, aufgeräumte Oberfläche – etwa ListViews, Buttons und Eingabefelder, die ein plattformspezifisches Look and Feel sicherstellen.

Dieses erweiterte Beispiel ist bereits technisch anspruchsvoll, motiviert aber zum Ausprobieren: Es zeigt, wie man mit der Uno Platform Navigation, Zwei-

Wege-Datenbindung, Commands und plattformübergreifende Funktionen in einer realistischen App umsetzt.

Fazit und Ausblick

Die Uno Platform zeigt bereits im ersten Schritt, wie sich mit vertrauter WinUI-/XAML-Syntax plattformübergreifende UIs entwickeln lassen. Architektur, UX-Konzepte, Tooling und ein starkes Ökosystem schaffen die Basis für produktive Workflows, die Desktop, Mobile und Web gleichermaßen unterstützen. Das Beispiel verdeutlicht, wie schon mit wenigen Zeilen Code eine App entsteht, die auf allen Zielsystemen lauffähig ist.

Weitere Artikel unter developer-world.de/dwx-insights werden sich mit den Layoutsystemen der Uno Platform beschäftigen und zeigen, wie man mit Grids, StackPanels und flexiblen Containern responsive Oberflächen gestalten kann. ■

- [1] Uno Platform, <https://platform.uno/>
- [2] Pixel-perfect, <https://platform.uno/platforms/#:~:text=Single%20codebase%20on%20all%20platforms>
- [3] Themes, <https://github.com/unoplatform/Uno.Themes>
- [4] Extensions, <https://platform.uno/docs/articles/external/uno.extensions/doc/ExtensionsOverview.html>
- [5] MVUX, <https://platform.uno/docs/articles/external/uno.extensions/doc/Learn/Mvux/Overview.html>
- [6] Uno Playground, <https://playground.platform.uno/>
- [7] Uno.Samples-Repository, <https://platform.uno/docs/articles/external/uno.samples/README.html>



Dr. Veikko Krypczyk

ist begeisterter Entwickler und Fachautor. Er bietet unter anderem Seminare und Workshops zu WinUI 3, .NET MAUI und Blazor an. Weitere Informationen finden Sie unter <https://larinet.com>.

PERSONALISIERTE USER EXPERIENCE

Ganz wie für mich gemacht

Nutzer erwarten zunehmend Erlebnisse, die sich anfühlen wie maßgeschneidert.

Die digitale Nutzungskultur hat sich in den letzten Jahren radikal verändert. Schon die Begrüßung auf einer Plattform soll persönlich wirken, Produktempfehlungen sollen tatsächlich relevant sein und Dialogsysteme wie Chatbots sollen im Idealfall die Situation verstehen, in der man sich gerade befindet. Dieser Anspruch ist durch den Vormarsch der künstlichen Intelligenz möglich geworden. Systeme analysieren Verhalten, Vorlieben und Kontexte, lernen kontinuierlich dazu und passen Inhalte dynamisch an. Entwicklern und Designern bietet Personalisierung spannende neue Möglichkeiten, aber auch neue Herausforderungen. Sie müssen lernen, verstärkt mit dynamischen Interfaces und KI-gesteuerten Content-Bausteinen zu arbeiten. Klassische, starre Designsysteme stoßen dabei schnell an ihre Grenzen. Gefragt sind modulare, flexible UI-Komponenten, die sich intelligent kombinieren und anpassen lassen. Auch das Denken in Nutzungsszenarien verändert sich: Statt statischer User Flows entstehen variable Nutzerreisen, die situativ gesteuert werden.

Bevor man startet, sollte man sich über die Ziele der Personalisierung Gedanken machen [1]. Es gilt die richtigen Treiber zu identifizieren und nicht nur einem Hype zu folgen. Man kann sich von folgenden Fragen leiten lassen (Bild 1):

- Welches übergeordnete strategische Ziel treibt die Personalisierung an?
- Was sind die spezifischen, messbaren Ergebnisse, zum Beispiel ein höherer Umsatz?
- Wo wird das personalisierte Erlebnis präsentiert (wo in der App, Webseite)?
- Welche Personalisierungsinhalte werden dem Benutzer angezeigt (zum Beispiel individuelle Preise)?
- Was macht die Zielgruppe aus?
- Welche zuverlässigen und maßgeblichen Daten werden von unserer technischen Plattform erfasst, um die Personalisierung voranzutreiben?
- Welcher größere Datensatz ist denkbar (bereits in unserer Umgebung), der Ihnen eine Personalisierung ermöglicht?

Was bedeutet Hyperpersonalisierung?

Lange Zeit basierte Personalisierung vor allem auf statischen Filtern und einfachen Segmentierungslogiken. Ein Newsletter wurde vielleicht nach Geschlecht differenziert, eine Produktliste nach den letzten Käufen sortiert. Diese Form der Personalisierung war zwar ein erster Schritt, blieb aber oft oberflächlich. Nutzergruppen wurden grob zusammenge-

fasst, die Anpassungen waren vordefiniert und änderten sich nur selten.

Das Konzept der Hyperpersonalisierung geht deutlich weiter. Hier stehen nicht mehr statische Regeln im Vordergrund, sondern Systeme, die in Echtzeit lernen und reagieren. Das bedeutet, dass eine Anwendung während der Nutzung fortlaufend Daten sammelt, Muster erkennt und auf dieser Basis Entscheidungen trifft. Sie berücksichtigt nicht nur, was ein Nutzer in der Vergangenheit getan hat, sondern auch, was er im Moment gerade braucht. Ein Beispiel: Ein Online-Shop zeigt einem Kunden nicht einfach Produkte an, die andere Käufer „ähnlich wie er“ gekauft haben, sondern analysiert zusätzlich Kontextdaten wie Gerätetyp, Uhrzeit oder sogar Wetterbedingungen. Daraus entsteht ein dynamischer, flexibler Empfehlungskosmos, der situativ relevant bleibt.

Die Rolle der KI ist dabei zentral. Klassische Regelwerke stoßen schnell an ihre Grenzen, wenn es darum geht, Millionen von Datenpunkten auszuwerten oder komplexe Muster im Verhalten zu erkennen. KI-Systeme – insbesondere Machine-Learning-Algorithmen – sind dagegen in der Lage, kontinuierlich zu lernen. Sie verarbeiten enorme Datenmengen in kürzester Zeit und passen ihre Modelle ständig an. So entstehen Systeme, die im Hintergrund beständig dazulernen: Welche Inhalte funktionieren für welchen Nutzer? Wie verändert sich die Interaktion über die Zeit? Welche Faktoren beeinflussen das Ziel, das eine Person verfolgt?

Für Entwickler bedeutet das, dass sie mit einer neuen Qualität von Daten und Feedbackschleifen arbeiten können. KI-basierte Systeme liefern nicht nur Informationen, sondern auch Prognosen: Welcher nächste Schritt ist wahrscheinlich? Welche Inhalte erhöhen die Chance auf Interaktion? Welche Nutzergruppe zeigt ähnliche Muster? Durch diesen kontinuierlichen Lernprozess verschiebt sich der Fokus von nachträglicher Anpassung hin zu proaktiver Unterstützung.

Die Umsetzung der Hyperpersonalisierung in der UX erfolgt über verschiedene Methoden, die je nach Anwendungskontext kombiniert werden können. Ein Ansatz sind personalisierte Interfaces, bei denen Dashboards abhängig von der jeweiligen Nutzerrolle unterschiedliche Widgets anzeigen oder sich Layouts adaptiv an die individuellen Vorlieben und Nutzungsmuster anpassen. Ein weiterer wichtiger Bereich sind Recommendation Engines. Diese nutzen künstliche Intelligenz, um passende Produkte, Artikel, Songs oder Kurse vorzuschlagen, und können darüber hinaus kontextabhängig-

ge Empfehlungen geben – etwa wenn ein Lieferdienst zur gewohnten Uhrzeit bevorzugte Speisen anbietet.

Darüber hinaus spielt dynamischer Content eine zentrale Rolle. Texte, Bilder oder Videos passen sich in Echtzeit an das jeweilige Nutzerprofil an. Auch Chatbots und Conversational UX bieten großes Potenzial für Hyperpersonalisierung, da sie ihre Dialogführung an die Historie und das Profil des Nutzers anpassen können. So lassen sich beispielsweise Fragen zu bereits gekauften Produkten individuell beantworten.

Eine weitere Ebene bilden Micro-Interactions und Nudging. Kleine Hinweise, Animationen oder Gamification-Elemente helfen den Nutzern dabei, ihre Ziele schneller zu erreichen – abgestimmt darauf, wie empfänglich sie für solche Impulse sind. Schließlich sorgt eine adaptive Navigation dafür, dass Menüs und Suchergebnisse personalisiert werden, indem häufig genutzte Funktionen priorisiert dargestellt werden.

Von User Flows zu variablen Nutzerreisen

Lange Zeit war es üblich, digitale Produkte entlang linearer User Flows zu gestalten. Der klassische Ansatz sah eine Abfolge von Schritten vor, die ein Nutzer typischerweise

durchlief, um ein bestimmtes Ziel zu erreichen. Diese Flows waren weitgehend statisch: Jede Interaktion war vorhersehbar, jede Abzweigung wurde im Vorfeld definiert und das System erwartete, dass Nutzer sich in diese Struktur einfügen.

Doch die digitale Realität von heute ist weitaus komplexer. Nutzer bewegen sich nicht linear durch digitale Produkte, sondern kontextabhängig, situativ und individuell. Sie wechseln Geräte, springen zwischen Kanälen, haben unterschiedliche Vorkenntnisse und Erwartungen. Hinzu kommt, dass sie zunehmend an maßgeschneiderte Erlebnisse gewöhnt sind. Vor diesem Hintergrund reicht ein starres Modell statischer Flows nicht mehr aus. Stattdessen treten variable Nutzerreisen in den Vordergrund, die flexibel auf Situationen, Daten und Kontexte reagieren.

Der zentrale Unterschied zwischen einem klassischen User Flow und einer variablen Nutzerreise liegt in der Adaptivität. Während ein Flow den „besten Weg“ vorgibt, lässt die Journey mehrere Wege zu und wählt situativ den passendsten. Variable Nutzerreisen basieren auf der Fähigkeit, Situationen zu erkennen und darauf zu reagieren. Solche Situationen können sein:

- **Kontextbezogen:** Gerätetyp, Standort, Tageszeit oder Netzverbindung
- **Nutzerbezogen:** Vorwissen, aktuelle Ziele, Interaktionshistorie, persönliche Präferenzen
- **Systembezogen:** Verfügbarkeit von Daten, Status von Services, Auslastung oder Fehlerzustände

Ein Beispiel: In einer Banking-App kann ein Nutzer den Wunsch äußern, eine Überweisung durchzuführen. Ein statischer Flow würde ihn Schritt für Schritt durch Eingabemaschinen führen. Eine variable Nutzerreise hingegen könnte erkennen, dass der Nutzer häufig denselben Empfänger wählt, ihm diesen automatisch vorschlagen und damit den Eingabeaufwand reduzieren. Ist der Nutzer im Ausland, könnte die App kontextsensitiv zusätzliche Sicherheitshinweise einblenden. Wird eine schlechte Internetverbindung erkannt, könnte der Prozess vereinfacht und zwischengespeichert werden. Variable Nutzerreisen entstehen nicht im luftleeren Raum, sondern auf Basis von Daten. Moderne Systeme nutzen Machine Learning, Behavioral Analytics und Personalisierungsstrategien, um vorherzusagen, welche Option in einer Situation am besten passt. Dies kann auf Mikro-Ebene geschehen, zum Beispiel die Reihenfolge von Inhalten in einem Feed, oder auf Makro-Ebene, zum Beispiel welche Customer Journey im Onboarding angestoßen wird.

Die Gestaltung variabler Nutzerreisen erfordert ein Umdenken in der UX-Arbeit [2]. Designer können nicht mehr allein lineare Flows modellieren, sondern

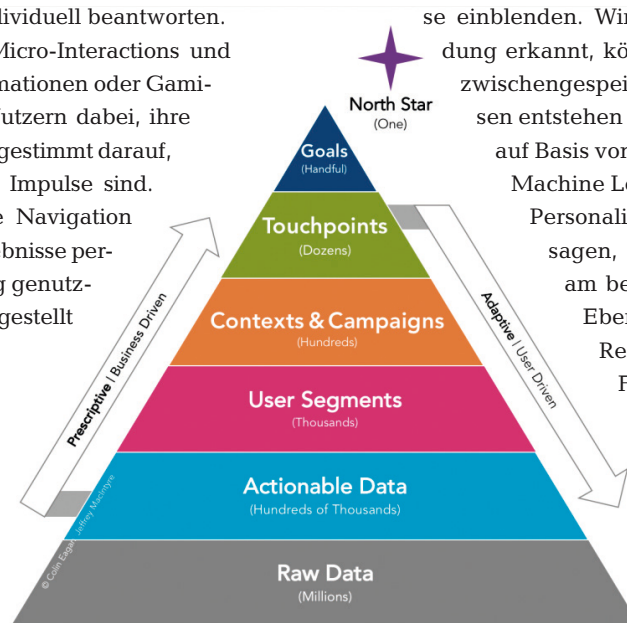
müssen Situationen, Entscheidungsbäume und dynamische Pfade berücksichtigen. Das bedeutet:

- **Mehr Szenarien statt weniger:** Statt einen optimalen Weg zu beschreiben, müssen mehrere gleichwertige Pfade gestaltet werden.
- **Kontinuität im Fokus:** Nutzer springen zwischen Kanälen, zum Beispiel Website, mobiler App und Chatbot. Die Journey muss konsistent und bruchfrei bleiben.
- **Transparenz für Nutzer:** Adaptive Systeme dürfen nicht zu Blackboxes werden. Nutzer sollten nachvollziehen können, warum ihnen bestimmte Optionen gezeigt werden.

Ein weiteres Beispiel aus der Praxis: In E-Commerce-Apps kann der Checkout-Prozess stark variieren. Manche Nutzer bevorzugen PayPal, andere Kreditkarte, wieder andere wollen auf Rechnung kaufen. Eine variable Journey kann erkennen, welche Option am wahrscheinlichsten passt, sie prominent platzieren und dadurch „Reibung“ verringern. Für Designer bedeutet das: Nicht nur ein Flow, sondern mehrere optimierte Varianten müssen gestaltet werden.

Personalisierungsstrategien im UX-Design

Es gibt unterschiedliche Strategien, die je nach Kontext, Zielgruppe und technologischen Möglichkeiten eingesetzt werden können. Eine der klassischen Formen ist die „inhaltsbasierte Personalisierung“, bei der Inhalte abhängig von Interessen, Präferenzen oder vorangegangenen Interaktionen ange-



Pyramide der Personalisierung [5] (Bild 1)

passt werden. So zeigen etwa News-Portale bevorzugt Themen, die ein Nutzer regelmäßig liest, während E-Commerce-Plattformen Produkte empfehlen, die zu bisherigen Käufen oder angesehenen Artikeln passen. Daneben spielt die „regelbasierte Personalisierung“ eine wichtige Rolle. Hier greifen feste, vorab definierte Bedingungen, die das Nutzererlebnis steuern. Beispiele sind das Einblenden eines geführten Onboarding-Flows für neue Nutzer, die Anzeige von Preisen in der lokalen Währung bei einem Standort im Ausland oder das Hervorheben exklusiver Funktionen für Premium-Mitglieder.

Während sich diese Strategien noch stark auf Einzelregeln stützen, geht die „segmentbasierte Personalisierung“ einen Schritt weiter: Nutzer werden in Gruppen eingeteilt, etwa Anfänger und Poweruser, und die Oberfläche wird entsprechend gestaltet. Einsteiger erhalten eher vereinfachte Oberflächen und Hilfestellungen, während erfahrene Anwender Abkürzungen, Expertenfunktionen oder Shortcuts nutzen können.

Einen weiteren Ansatz bildet die „kontextsensitive Personalisierung“, bei der situative Faktoren wie Gerät, Uhrzeit, Standort oder Netzverbindung berücksichtigt werden. Eine Banking-App kann beispielsweise bei schlechter Internetverbindung vereinfachte Prozesse anbieten, während eine Fitness-App morgens motivierende Reminder sendet und abends Tipps zur Regeneration in den Vordergrund stellt. Auch Navigations-Apps passen sich flexibel an, indem sie je nach Verkehrsmittel unterschiedliche Routenempfehlungen berechnen.

Eng verwandt ist die „verhaltensbasierte Personalisierung“, bei der Systeme Klickpfade, Verweildauer oder Nutzungsmuster analysieren, um individuelle Vorschläge zu machen. Streaming-Dienste wie Netflix oder Spotify nutzen diese Methode, um passende Filme oder Musik zu empfehlen. Auf Lernplattformen erhalten Nutzer basierend auf ihren bisherigen Kursen passende nächste Lektionen.

Mit dem Einzug von KI-Systemen gewinnt die „prädiktive Personalisierung“ zunehmend an Bedeutung. Hierbei sagen Algorithmen voraus, welche Inhalte oder Funktionen für den jeweiligen Nutzer am relevantesten sind. E-Mail-Programme sortieren automatisch nach „wichtig“ und „Sonstiges“, E-Commerce-Seiten priorisieren Produkte, die mit hoher Wahrscheinlichkeit interessant sind, und Chatbots passen ihre Antworten an den vermuteten Kontext an.

Neben solchen impliziten Strategien spielt auch die „explizite Personalisierung“ durch den Nutzer selbst eine wichtige Rolle. Nutzer behalten dabei die Kontrolle, indem sie Oberflächen, Dashboards oder Profile individuell anpassen können. Beispiele sind frei konfigurierbare Startseiten, die Auswahl von Themeninteressen oder das Umschalten zwischen Light- und Dark Mode. Auch Gamification-Elemente können für personalisierte Erlebnisse sorgen. Fitness-Apps schlagen etwa individuelle Challenges vor, während Sprachlern-Apps wie Duolingo tägliche Ziele, Fortschrittsbalken und Belohnungen an die Lerngewohnheiten des Nutzers anpassen.

Schließlich gibt es die „Cross-Channel-Personalisierung“, die darauf abzielt, Erlebnisse über mehrere Kanäle hinweg konsistent zu halten. Ein klassisches Beispiel ist der Warenkorb, der am Desktop begonnen und in der mobilen App nahtlos fortgeführt werden kann. Auch im Hotel- oder Reise-

bereich wird diese Strategie genutzt, indem Buchungen oder Reservierungen über Website, App und E-Mail personalisiert weitergeführt werden.

Damit wird deutlich, dass Personalisierung im UX-Design kein monolithisches Konzept ist, sondern ein Spektrum an Strategien umfasst, die gezielt kombiniert werden können. Je nach Produkt und Zielgruppe gilt es, die richtige Balance zwischen Automatisierung, Transparenz und Nutzerkontrolle zu finden – um Relevanz und Effizienz zu steigern, aber auch Vertrauen und Selbstbestimmung der Nutzer zu sichern.

Technische Basics

Für Entwickler heißt Personalisierung, neue Werkzeuge und Techniken zu meistern. Kenntnisse in Machine Learning und Datenverarbeitung werden immer wertvoller, auch im klassischen Softwareteam. Zwar muss nicht jeder Entwickler selbst ein neuronales Netz trainieren können, aber zumindest das Verständnis für die Anbindung von ML-Modellen, für die Funktionsweise von Empfehlungssystemen und für das Zusammenspiel mit Datenpipelines wird wichtig. In .NET-Teams kommen zum Beispiel Technologien wie ML.NET, Azure AI Services (früher: AI Cognitive Services) [3] oder ONNX-Integration ins Spiel, die bisher vielleicht außerhalb des typischen C#-Stacks lagen.

Zudem müssen Entwickler lernen, mit dynamischen Anforderungen umzugehen. Das Frontend kann nicht mehr alles deterministisch vorausplanen. Stattdessen sollte es eher regelbasiert oder datengetrieben aufgebaut sein: zum Beispiel ein Layout-Manager, der vom Backend eine Konfiguration bekommt. Schließlich erfordert Hyperpersonalisierung ein enges Zusammenspiel verschiedener Disziplinen. Die Grenzen zwischen Design, Data Science und Development verwischen. Beispielsweise müssen Data Scientists die Erkenntnisse aus Nutzerdaten in Personalisierungsmodelle gießen, Entwickler diese Modelle effizient in die App integrieren und Designer sicherstellen, dass die Resultate im Interface ansprechend bleiben. Hier sind neue Rollen denkbar, etwa ein „AI Experience Designer“, der sowohl KI-Verständnis als auch UX-Know-how mitbringt.

Personalisierung in .NET – Konzepte und Ansätze

In diesem Textabschnitt betrachten wir einige mögliche Konzepte und Vorgehensweisen, mit denen man Personalisierung in .NET-basierte Applikationen, das heißt in Apps für WinUI (Desktop), .NET MAUI (Cross-Plattform) und Blazor (Web), integrieren kann. Für Entwickler bedeutet das, Machine Learning mit modernen UI-Technologien zu verbinden, um maßgeschneiderte Erlebnisse zu schaffen. Ein erster Ansatz betrachtet personalisierte Dashboards und dynamische UIs. Statt Einheitsinterfaces werden personalisierte Oberflächen zum Standard. Dashboards zeigen je nach Nutzerrolle unterschiedliche Widgets, und Layouts passen sich an individuelle Nutzungsmuster an. Dazu müssen UI-Komponenten modular und flexibel sein, sodass sie zur Laufzeit neu kombiniert werden können. Ein Beispiel für eine Desktop-App, welche auf WinUI 3 basiert, könnte wie folgt funktionieren. In einer WinUI-3-App lässt sich an-

hand von Benutzerattributen festlegen, welche Dashboard-Widgets erscheinen:

```
if (currentUser.Role == Roles.Admin)
    DashboardPanel.Children.Add(
        new AdminAnalyticsWidget());
else
    DashboardPanel.Children.Add(new BasicStatsWidget());
```

Eine vom Backend gelieferte oder in Einstellungen definierte Konfiguration steuert, welche UI-Elemente sichtbar sind.

Ein weiteres Beispiel adressiert die variablen Nutzerreisen. Eine adaptive Navigation passt Menüs dynamisch an den Nutzer an – häufig genutzte Funktionen werden im Menü prominent platziert. In einer .NET-MAUI-App mit Shell-Navigation kann das Menü zur Laufzeit angepasst werden. Man könnte die Reihenfolge der *Flyout*-Menüeinträge nach Nutzungsstatistik sortieren oder neue Einträge bei Bedarf hinzufügen:

```
if (usageStats["Reports"] > threshold)
    Shell.Current.Items.Add(new FlyoutItem {
        Title="Reports", Icon="reports.png",
        Route="reportsPage" });
```

In diesem Beispiel wird der Eintrag *Reports* ins Menü eingefügt, sobald der Nutzer die Berichtsfunktion häufig nutzt. Ebenso können selten genutzte Optionen ausgeblendet werden. Durch Tracking des Nutzerverhaltens lernt die Navigation, bevorzugte Pfade anzubieten.

Für die technische Umsetzung von dynamischen Inhalten verschmelzen Machine Learning und .NET. Mit ML.NET lassen sich ML-Modelle direkt in Apps einbetten. Alternativ kann man Cloud-Dienste wie Azure nutzen, wo per Reinforcement Learning die optimale Option gewählt und aus Feedback gelernt wird. Zentral ist eine Feedbackschleife: Nutzerinteraktionen (Klicks, Käufe, Feedback) werden erfasst und zurückgespielt, um Empfehlungen fortlaufend zu optimieren. Durch kontinuierliches Lernen wandelt sich Personalisierung von statischer Anpassung zu proaktiver Unterstützung, die dem Nutzer den nächsten sinnvollen Schritt vorschlägt.

Ein weiteres Beispiel nutzt Blazor und dort einen datenbasierten injizierten Service in einer Blazor-Komponente. Diese ermittelt eine Liste relevanter Artikel:

```
@inject RecommendationService recService

@if (recService.GetRecommendations(userId, contextData)
    is {} recs && recs.Any()) {
    <ul>
        @foreach (var article in recs) {
            <li>@article.Title</li> }
        </ul>
    }
```

Hier berücksichtigt *GetRecommendations(...)* vergangene Aktionen und den aktuellen Kontext des Nutzers. Das Prinzip lässt sich auf Produkte, Musik und anderes übertragen.

Die Grundlage von Hyperpersonalisierung in der Praxis ist ein solides Datenmodell. Alle verfügbaren Informationen, Profileinstellungen, vergangene Interaktionen, Kontextdaten und so weiter müssen erfasst und effizient abrufbar sein. Darauf aufbauend existieren verschiedene Personalisierungsstrategien: von einfachen Regelwerken über segmentierte Nutzergruppen bis zu KI-Modellen. Je nach Anwendung gilt es, die passende Mischung zu wählen – transparente Regeln für Basics, Machine Learning für komplexe Muster.

Textanalyse als Voraussetzung für personalisierte Apps

Echte Personalisierung basiert auf Verständnis – nicht nur für Nutzerdaten, sondern auch für Sprache, Stimmung und Intention. Während klassische Systeme Nutzerprofile oder Klickverhalten auswerten, geht die Hyperpersonalisierung in modernen .NET-Anwendungen einen Schritt weiter: Sie nutzt Textanalyse und Machine Learning, um aus unstrukturierten Texten verwertbare Erkenntnisse zu gewinnen. Ob Chatnachricht, Feedback, Support-Ticket oder Kommentar – jede sprachliche Interaktion enthält wertvolle Hinweise auf Bedürfnisse, Emotionen und Kontext.

Damit wird Textanalyse zu einem zentralen Werkzeug intelligenter UX-Systeme: Sie erlaubt Anwendungen, Nutzer zu verstehen, bevor sie reagieren. Eine Lern-App kann beispielsweise die Stimmung eines Nutzers erkennen („Ich komme einfach nicht voran“) und darauf mit passenden Inhalten reagieren („Hier sind vereinfachte Tipps zum Wiedereinstieg“). Oder sie erkennt, dass ein Nutzer begeistert ist („Super Fortschritte heute!“) und schlägt anspruchsvollere Optionen vor. So entstehen adaptive, empathische Interaktionen, die über reine Datenanpassung hinausgehen – die Software verhält sich fast menschlich.

In der Praxis lassen sich Textanalyse-Ergebnisse in verschiedenen Personalisierungsstrategien nutzen:

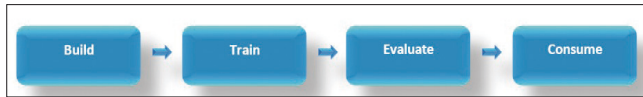
- **Adaptive Empfehlungen:** Eine Lernplattform kann positive Rückmeldungen erkennen und schwierigere Aufgaben vorschlagen, während sie bei Frustration einfachere Inhalte anbietet.
- **Kontextsensitives UX-Writing:** Texte und Hinweise lassen sich dynamisch an Stimmung oder Sprachstil anpassen.
- **Intelligente Chatbots:** Dialogsysteme interpretieren Emotionen oder Themen und reagieren empathisch statt neutral.
- **Stimmungsbasierte Navigation:** Nutzer in positiver Stimmung erhalten proaktiv neue Angebote, während genervte Anwender gezielt Unterstützung finden.

Damit die App solche Schlüsse ziehen kann, muss sie Texte analysieren – lokal auf dem Gerät oder cloudbasiert. Im .NET-Ökosystem stehen dafür zwei Wege offen: ML.NET für eigene, lokale Modelle und Azure AI Services für leistungsfähige Cloud-APIs.

ML.NET [4] ist das Machine-Learning-Framework von Microsoft für .NET-Entwickler. Es ermöglicht, Modelle direkt in C# zu trainieren und in Anwendungen einzubetten – ohne Abhängigkeit von externen Diensten. Ein typisches Beispiel ist eine Sentimentanalyse, die Texte automatisch als positiv ►

oder negativ einstuft. Ein einfacher Workflow (Phasen) sieht so aus (Bild 2):

1. Datenvorbereitung/Laden
2. Feature-Engineering/Transformation
3. Modelltraining (Trainer)
4. Evaluierung
5. Modellnutzung/Vorhersage (Inference)



ML-Learning-Pipeline [6] (Bild 2)

Diese Phasen werden in C# mit dem ML.NET-API realisiert. Der resultierende Code ist einfach genug, um als Vorlage in eigenen Projekten verwendet zu werden.

Wir zeigen im Folgenden wesentliche Codeabschnitte der verschiedenen Phasen.

1. Datenklasse und Laden der Daten

In der Regel wird eine Klasse definiert, die die Eingabedaten repräsentiert, zum Beispiel:

```

public class SentimentData
{
    [LoadColumn(0)]
    public string Text { get; set; }

    [LoadColumn(1), ColumnName("Label")]
    public bool Label { get; set; }
}

public class SentimentPrediction
{
    [ColumnName("PredictedLabel")]
    public bool Prediction { get; set; }

    public float Probability { get; set; }
    public float Score { get; set; }
}
  
```

- *LoadColumn(0)* und *LoadColumn(1)* geben an, welche Spalte aus der Datenquelle (zum Beispiel .tsv-Datei) in welches Feld geladen wird.
- *ColumnName("Label")* markiert das Feld als Zielvariable.
- In der Ergebnisklasse (*SentimentPrediction*) definiert man die Felder zur Vorhersage wie *Prediction*, *Probability* und optional *Score*.

Dann wird das Dataset mit *mlContext.Data.LoadFromTextFile<SentimentData>(...)* geladen, zum Beispiel:

```

IDataView dataView =
    mlContext.Data.LoadFromTextFile<SentimentData>(
        dataPath, hasHeader: false, separatorChar: '\t');
  
```

Anschließend teilt das Sample die Daten oft in Trainings- und Testdaten, zum Beispiel:

```

var split = mlContext.Data.TrainTestSplit(
    dataView, testFraction: 0.2);
var trainData = split.TrainSet;
var testData = split.TestSet;
  
```

Damit kann man später das Modell auf *trainData* trainieren und mit *testData* evaluieren.

2. Pipeline/Feature-Engineering

Da ML-Algorithmen keine rohen Texte direkt verwenden, besteht ein wichtiger Schritt darin, Texte in numerische Merkmale zu überführen. Im Sample wird das mit *FeaturizeText* umgesetzt:

```

var pipeline = mlContext.Transforms.Text
    .FeaturizeText(outputColumnName: "Features",
        inputColumnName: nameof(SentimentData.Text))
    .Append(mlContext.BinaryClassification.Trainers
        .SdcaLogisticRegression(labelColumnName: "Label",
            featureColumnName: "Features"));
  
```

Transforms.Text.FeaturizeText(...) wandelt jeden Text in eine Merkmalsvektor-Darstellung um (zum Beispiel durch n-Gramme, Tokenisierung et cetera). Danach wird der Binary Classification Trainer (hier: *SdcaLogisticRegression*) angehängt, der auf der Basis der Merkmale *Features* das *Label* vorhersagt.

3. Training

Das Modell wird mit

```
ITransformer model = pipeline.Fit(trainData);
```

trainiert. *Fit(...)* führt alle Transformationsschritte aus und sucht optimale Parameter für den Trainer.

4. Evaluierung

Nach dem Training wird das Modell mit den Testdaten evaluiert:

```

var predictions = model.Transform(testData);
var metrics = mlContext.BinaryClassification.Evaluate(
    predictions, labelColumnName: "Label");
Console.WriteLine($"Accuracy: {metrics.Accuracy:P2},
    AUC: {metrics.AreaUnderRocCurve:P2}");
  
```

Dabei werden Standardmetriken wie *Accuracy*, *AUC*, *F1-Score* und andere ausgegeben (abhängig von der Version des Samples). So sieht man, wie gut das Modell bei ungesesehenen Daten abschneidet.

5. Vorhersage (Inference)

Um mit neuem Text Vorhersagen zu machen, verwendet das Sample *PredictionEngine*:

```
var predEngine = mlContext.Model.CreatePredictionEngine<
    SentimentData, SentimentPrediction>(model);
SentimentData sample = new SentimentData { Text =
    "This is a wonderful movie!" };
SentimentPrediction result = predEngine.Predict(sample);
Console.WriteLine($"Text: {sample.Text} → Prediction:
    {result.Prediction}, Prob: {result.Probability:P2}");
```

PredictionEngine ist ein einfaches API für One-by-One-Vorhersagen. *result.Prediction* ist der boolesche Vorhersagewert (*true/false*) und *result.Probability* gibt die Wahrscheinlichkeit (Konfidenz) der Vorhersage an.

Das Beispiel ist ideal als Lernbeispiel, aber um es in einer realen Anwendung (zum Beispiel für Personalisierung) zu verwenden, empfiehlt sich Folgendes:

- **Modell persistent speichern und laden:** Das trainierte Modell wird typischerweise in eine Datei serialisiert und später in der App geladen.
- **Batch-Prediction statt Einzelvorhersagen:** Wenn viele Texte gleichzeitig analysiert werden müssen, lohnt sich eine Batch-Vorhersage statt vieler kleiner Einzelaufrufe.
- **Fehler- und Ausnahmebehandlung:** Texte könnten leer sein oder das Modell kann einmal versagen – robustes Fehlerhandling ist wichtig.
- **Versionierung und Retraining:** Modelle sollten gelegentlich neu trainiert werden, wenn neue Daten eingehen (Feedback-Schleife).

Die Vorhersageergebnisse (zum Beispiel dass ein Nutzertext negativ war) können genutzt werden, um UI-Entscheidungen zu treffen – etwa spezielle Hilfsangebote zeigen, weniger aggressive Werbung, angepasste Tonalität et cetera. In der Praxis lässt sich dieses Pattern leicht in .NET-Anwendungen integrieren (WinUI, MAUI, Blazor, ASP.NET). Durch die gewonnenen Sentimentinformationen kann eine App personalisierte Entscheidungen treffen: Nutzerfeedback, Empfehlungslogik oder UI-Anpassungen werden auf emotionales und semantisches Verständnis gestützt.

Wenn man in .NET-Apps statt eines lokal trainierten Modells auf einen Managed-Cloud-Dienst zurückgreifen möchte, stellt Azure AI Services [3] eine leistungsstarke Alternative dar. Microsoft bezeichnet diese Plattform als Sammlung von vorgeladenen und anpassbaren KI-APIs für Sprach-, Bild-, Entscheidungs- und Sensordienste. Diese Dienste stehen per REST-APIs und .NET-SDKs bereit und lassen sich direkt aus Anwendungen ansteuern, ohne dass ein eigenes Training nötig wäre.

Konzeptionell erfolgt die Nutzung so:

1. **Azure-Ressource einrichten:** Im Azure-Portal legt man eine „Azure AI Language“ oder „Text Analytics“-Ressource an. Man erhält einen Endpunkt (URL) und einen API-Key, die der App später als Authentifizierungsdaten dienen.
2. **Clientbibliothek oder REST-API verwenden:** In der .NET-App nutzt man das SDK (zum Beispiel *Azure.AI.TextAnalytics*) oder ruft direkt das REST-API auf. Diese Bibliotheken kapseln HTTP-Aufrufe, Authentifizierung, Anfragen und Antwortverarbeitung.

3. **Text senden und analysieren:** Der Client ruft zum Beispiel eine Methode wie *AnalyzeSentimentAsync(text, language)* auf. Der Dienst verarbeitet den Text mit internen KI-/NLP-Modellen und liefert ein Ergebnisobjekt zurück: unter anderem *Sentiment* (zum Beispiel *Positive/Negative/Mixed*), Konfidenzwertungen (*ConfidenceScores*) und oft Satz- oder Aspektantworten.

4. **Ergebnis in Anwendung nutzen:** Die Rückgabe des Dienstes wird in der App ausgewertet und kann in Personalisierungsentscheidungen einfließen: UI-Anpassungen, Empfehlungsrotation, Priorisierung von Hilfeangeboten, sprachliche Anpassungen und Ähnliches.

5. **Skalierung, Monitoring, Updates:** Da der Dienst von Microsoft gehostet wird, übernimmt Azure Skalierung, Modellaktualisierungen und Wartung. Sie müssen nur korrekt mit Ihren Schlüsseln und Limits umgehen.

Der Einsatz von cloudbasierten KI-Services bietet folgende Vorteile:

- **Kein eigenes Training notwendig:** Die Modelle sind vorgelernt und sofort einsatzbereit — das erspart Aufwand für Datenaufbereitung, Hyperparameter-Tuning oder Infrastruktur.
- **Mehrsprachigkeit und Domänenabdeckung:** Azure AI unterstützt viele Sprachen und deckt verbreitete Domänen gut ab.
- **Skalierbarkeit und Wartung:** Der Dienst skaliert automatisch und Microsoft übernimmt Infrastruktur und Updates.
- **Schnelle Integration:** Mit nur wenigen Zeilen .NET-Code lässt sich Sentimentanalyse realisieren, ideal für Proof of Concepts.

Es gibt aber auch Nachteile:

- **Abhängigkeit von Internet und Dienstverfügbarkeit:** Funktioniert nur, wenn eine Verbindung zum Azure-Dienst besteht.
- **Laufende Kosten:** Abhängig von Nutzung und Aufrufvolumen entstehen Gebühren (etwa nach Anzahl Transaktionen pro 1000 Texte).
- **Begrenzter Customizing-Spielraum:** Man ist an die von Microsoft bereitgestellten Analysen gebunden; sehr spezielle Modelle (Fachsprache, Nischen) lassen sich schwer anpassen.
- **Datenschutz/Compliance:** Texte werden über das Internet an Azure gesendet – das wirft zwangsläufig Datenschutzfragen auf, insbesondere bei sensiblen Daten oder in regulierten Umgebungen.

Azure-basierte Sentimentanalyse bietet sich besonders in Szenarien an, in denen große Mengen an Textdaten automatisiert ausgewertet und emotional interpretiert werden sollen. Ein typisches Einsatzfeld ist das Kundenfeedback- und Bewertungsmanagement: Eingehende Texte aus Supportformularen oder Online-Reviews können automatisch nach ihrer Stimmung sortiert werden, sodass negative Rückmeldungen priorisiert und schneller bearbeitet werden. Auch Chatbots und Messaging-Plattformen profitieren von der Stimmungsanalyse – sie erkennen die emotionale Lage des Nutzers und reagieren empathischer, etwa deeskalierend bei Frustration oder besonders freundlich bei Begeisterung. ►

Ein weiteres Anwendungsgebiet ist das Social-Media-Monitoring, bei dem Beiträge und Kommentare in Echtzeit analysiert werden, um die allgemeine Stimmung zu einer Marke, einem Produkt oder einer Kampagne zu erfassen (Bild 3). So können Unternehmen frühzeitig auf Trends oder Krisen reagieren.

Schließlich spielt Sentimentanalyse auch bei der Content-Personalisierung eine zentrale Rolle: Je nach Stimmung des Nutzers lässt sich die Tonalität, Komplexität oder Menge der angezeigten Inhalte dynamisch anpassen – beispielsweise durch vereinfachte Benutzeroberflächen bei merkbarer Frustration oder durch anspruchsvollere Empfehlungen bei positiver Grundhaltung.

Auf Ebene des Quellcodes stellt sich das in .NET-Apps wie folgt dar:

```
var client = new TextAnalyticsClient(new Uri(endpoint),
    new AzureKeyCredential(key));
var sentiment = await client.AnalyzeSentimentAsync(
    userInputText, "de");
if (sentiment.Value.Sentiment ==
    DocumentSentimentKind.Negative)
{
    ShowSupportTip();
}
else
{
    ShowRecommendedContent();
}
```

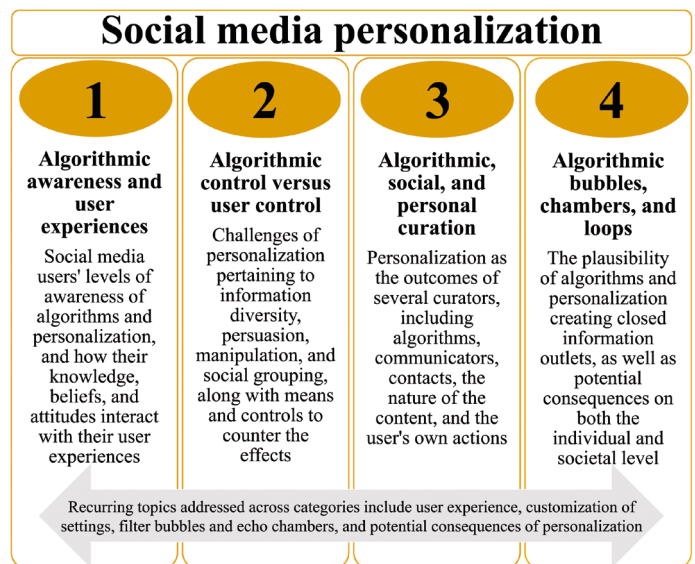
In einer .NET-MAUI-, WinUI- oder Blazor-App kann dieser Logikblock in Response-, Button-Handler- oder ViewModel-Methoden integriert werden – der entscheidende Punkt ist: Die Anwendung reagiert nicht nur auf Klicks, sondern auf Stimmung und Sprache.

Mit diesem Ansatz verschiebt sich die Rolle der App: von rein reaktiver Oberfläche hin zu einem verständigen, empathischen System. Die Azure-Dienste erlauben es, hohe KI-Qualität mit minimaler Infrastruktur zu nutzen. Voraussetzung bleibt jedoch eine stabile Netzverbindung und die sorgfältige Behandlung von Datenschutz und Kosten.

Chancen und Schattenseiten: Datenschutz und Ethik

Je individueller und datengestützter die Erlebnisse werden, desto dringlicher werden Fragen nach Datenschutz, Sicherheit und Ethik. Kommen wir als Erstes zur offensichtlichsten Herausforderung: Datenschutz und Privatsphäre. Hyperpersonalisierung fußt darauf, möglichst viele Daten über die Nutzer zu sammeln und auszuwerten – von expliziten Angaben bis zu impliziten Verhaltensdaten.

Diese Fülle an persönlichen Informationen muss gemäß geltenden Gesetzen (man denke nur an die DSGVO und andere) transparent und rechtmäßig erhoben werden. Nutzer müssen einwilligen oder zumindest informiert sein, welche Daten wofür verwendet werden. Und die Daten müssen sicher gespeichert werden. Die Verantwortung wächst mit dem



Social-Media-Personalisierung [7] (Bild 3)

Grad der Personalisierung – sensitive Kundendaten müssen besonders bewusst geschützt werden, um ethische Grundsätze zu wahren. Unternehmen, die hier schludern, riskieren nicht nur Strafen, sondern auch den Verlust von Vertrauen, was im Zeitalter der Datenschutz-Sensibilisierung geschäftlich fatal sein kann.

Ein weiterer Aspekt ist die algorithmische Fairness und Transparenz. KI-Modelle sind oft komplexe Blackboxes – das kann dazu führen, dass die Nachvollziehbarkeit leidet. Doch gerade bei personalisierten Inhalten wäre es wichtig, erklären zu können, warum jemand bestimmte Angebote sieht. Zum einen, um regulatorischen Anforderungen, wie dem Recht auf Erklärung automatisierter Entscheidungen, nachzukommen. Zum anderen, um Bias und Diskriminierung zu vermeiden. Wenn das Training eines Empfehlungsalgorithmus auf historischen Daten basiert, können unbewusste Vorurteile übernommen und verstärkt werden.

Eine praktische Herausforderung stellt auch die Echtzeitfähigkeit dar: Um Personalisierung wirklich in Echtzeit zu bieten, sind sehr leistungsfähige Backend-Systeme erforderlich. Große Datenmengen müssen in Millisekunden verarbeitet werden, Modelle schnell erstellt oder aktualisiert werden, vielleicht auf verteilten Systemen und mit Caching-Lösungen. Nicht jede Organisation kann diese Infrastruktur bereitstellen – was dazu führen kann, dass manche Anbieter zwar Personalisierung versprechen, aber faktisch doch nur relativ statische Regeln im Hintergrund laufen. Hier gilt es, Architekturen zu entwerfen, die skalieren und die Latenz niedrig halten.

Aus Nutzersicht spielt schließlich die Akzeptanz eine große Rolle. Nicht jeder Nutzer reagiert positiv darauf, wenn ein System ihn „durchschaut“. Viele kennen das mulmige Gefühl, wenn eine plötzliche Werbung für genau das Produkt angezeigt wird, über das man gestern nur gesprochen hat – „Hört mein Smartphone mit?“

Hyperpersonalisierung kann schnell als aufdringlich empfunden werden, wenn sie zu gut ist oder zu persönlich wirkt. Entwickler müssen dieses Phänomen beachten. Empfohlen

wird, Personalisierung behutsam einzuführen und transparent zu kommunizieren. Außerdem sollte immer eine Opt-out-Möglichkeit bestehen, damit Nutzer selbst bestimmen können, wie viel Personalisierung sie wünschen. Ethiker mahnen auch an, dass ständige Personalisierung zur Filterblase führen kann: Nutzer sehen nur noch, was ihnen gefällt, und werden von neuen, überraschenden Inhalten vielleicht abgeschnitten.

Fazit und Ausblick

Hyperpersonalisierung macht Interfaces intelligenter und benutzerfreundlicher – doch sie erfordert neue Herangehensweisen. Auf technischer Seite nutzen Entwickler große Datenpipelines und KI-Modelle, um in Echtzeit zu reagieren. Auf Designseite gewinnen modulare Systeme, flexible Navigationen und durchdachtes UX-Writing an Bedeutung. Die Zusammenarbeit über Fachgrenzen hinweg ist dabei entscheidend, um ethische und rechtliche Aspekte nicht aus dem Blick zu verlieren.

Wer diese neue Ära meistert, kann seinen Nutzern ein fast „menschliches“ digitales Erlebnis bieten – Interfaces, die nicht nur aussehen wie für den Nutzer gemacht, sondern es wirklich sind. ■

[1] Amy Schade, *Customization vs. Personalization in the User Experience*, www.nngroup.com/articles/customization-personalization

- [2] Colin Eagan, *UX in the Age of Personalization*, <https://alistapart.com/article/emerging-ux-role-in-personalization>
- [3] Microsoft, *Azure AI Services*, <https://azure.microsoft.com/en-us/products/ai-services>
- [4] Microsoft, *ML.NET*, <https://dotnet.microsoft.com/en-us/apps/ai/ml-dotnet>
- [5] Colin Eagan, Jeffrey MacIntyre, *Personalization Pyramid: A Framework for Designing with User Data*, <https://alistapart.com/article/personalization-pyramid/>
- [6] GitHub, *.NET Machine-Learning Samples*, https://github.com/dotnet/machinelearning-samples/tree/main/samples/csharp/getting-started/BinaryClassification_SentimentAnalysis
- [7] Ragnhild Eg et al., *A scoping review of personalized user experiences on social media: The interplay between algorithms and human factors*, www.sciencedirect.com/science/article/pii/S2451958822000872



Elena Bochkor

beschäftigt sich mit dem Entwurf und Design mobiler Anwendungen, Webseiten und digitaler Produkte. Ihr technologischer Fokus liegt auf C#, WPF, WinUI 3 und .NET MAUI, und sie bietet entsprechende Workshops für Entwickler an. <https://larinet.com>

DWX ACADEMY

Hands-on Workshops
und Weiterbildung für
Software-Entwickler
und -Architekten

2 TAGE
INHOUSE ODER
REMOTE

UX/UI-DESIGN FÜR ENTWICKLER

- Gridsystem
- Layout adaptiver User Interfaces
- UI-Controls mittels Styles und Templates
- Windows Design Guideline
- UX-Designprozess
- Icons
- Animation
- Barrierefreiheit



SEBASTIAN SEIDEL



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

developer-world.de/dwx-academy

EINSTIEG IN MINIKUBE

Kubernetes light

Die Komplexität von Kubernetes ist nicht zu unterschätzen. Minikube kann den Einstieg in die Welt des Container-Orchestrierungstools erleichtern.

Beginnen wir mit einer kleinen Geschichte. Das Entwicklungsteam hat eine erste Anwendung geschrieben, die in Azure gehostet wird. ASP.NET-Core-API-Projekte werden in einem Azure App Service gehostet. Die ersten Kunden buchen die Anwendung, die gebuchten Hardware-Ressourcen müssen angepasst werden. Mit Azure Application Insights werden Probleme erfolgreich analysiert.

So weit, so gut. Doch zeigt die Abrechnung von Microsoft, dass all die tollen Dinge auch bezahlt werden müssen. Azure Application Insights, so hilfreich es ist, wird deaktiviert. Die Anwendung wird weiter ausgebaut, und mancher API-Aufruf wird von einer externen Komponente getriggert. Leider stellt sich heraus, dass es mit einer dedizierten Version der externen Komponente Probleme gibt. Tracing wäre an dieser Stelle mehr als hilfreich, aber leider steht dies nicht zur Verfügung.

Das Team begibt sich auf die Suche nach Alternativen zu Application Insights, wie zum Beispiel Prometheus und Grafana, die für die Telemetrie eingesetzt werden können. Sicher, es besteht die Möglichkeit, diese Komponenten als Managed Service zu buchen, doch es wird mit dem Gedanken gespielt, sich grundsätzlich von Azure zu lösen. Um langfristig unabhängig vom Cloud-Provider zu werden, bietet es sich an, über Kubernetes nachzudenken. Diese Komponente ist Neuland, doch kann der Betrieb der Telemetrie-Komponenten als „Lernprojekt“ dieser Umgebung betrachtet werden.

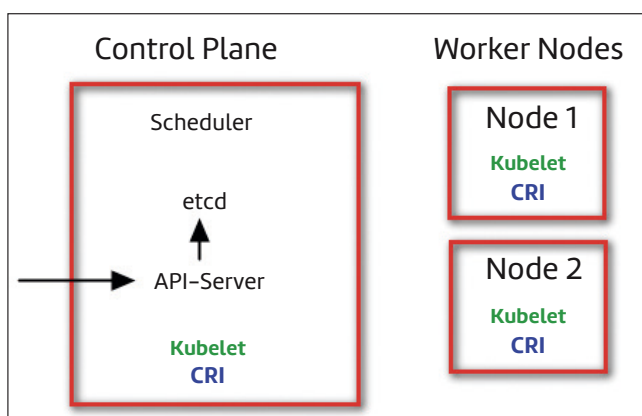
In diesem Beitrag möchte ich interessierten Leserinnen und Lesern einen ersten Einblick in das Thema Kubernetes gewähren. Die Komplexität des Systems ist nicht zu unterschätzen, daher wollen wir uns hier auf Aspekte beschränken, die einen Einstieg erleichtern. Uns unterstützt dabei die Kuberne-

tes-Distribution Minikube, die sich einfach unter Betriebssystemen wie Windows oder Linux (Ubuntu) installieren lässt und sich ausgezeichnet als preisgünstige Spielwiese für den Einstieg in die Kubernetes-Welt eignet.

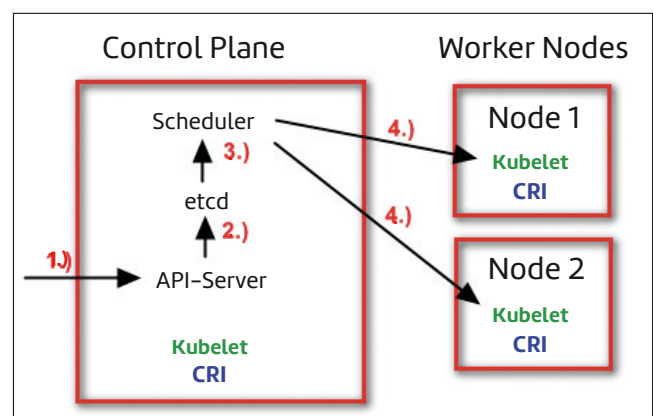
Was adressiert Kubernetes?

Kubernetes ist eine Umgebung für Softwarelösungen, die in Containern (zum Beispiel Docker) ausgeführt werden. Es behandelt unter anderem folgende Aufgabenstellungen:

- **Automatisiertes Deployment und Skalierung:** Die automatische Bereitstellung und Verwaltung von Containern reduziert manuelle Eingriffe. Es werden Rolling Updates, Rollbacks sowie automatische Skalierung basierend auf Systemlast unterstützt.
- **High Availability und Selfhealing:** Container, die sich in einem Fehlerzustand befinden, werden automatisch erkannt und gegebenenfalls ohne manuelles Eingreifen neu gestartet beziehungsweise ersetzt. Eine hohe Ausfallsicherheit wird durch Replikation und Verteilung der Ausführungseinheiten (Pods) realisiert.
- **Service Discovery und Lastverteilung:** Über die automatische Vergabe von internen DNS-Namen wird die Kommunikation zwischen den Pods erleichtert. Dabei sorgt Load Balancing für eine gleichmäßige Verteilung der Anfragen.
- **Deklarative und reproduzierbare Konfiguration:** Der gewünschte Systemzustand wird in YAML-Dateien definiert, die versioniert und in CI/CD-Prozesse integriert werden.
- **Ressourcenmanagement:** Ressourcen können je Ausführungseinheit definiert werden, die Verteilung auf die verschiedenen Knoten erfolgt durch einen Scheduler automatisch.



Kubernetes-Architektur (Bild 1)



Ablauf des Scheduling (Bild 2)

```
christian@christian-Virtual-Machine: ~
christian@christian-Virtual-Machine:~$ minikube start
🐹 minikube v1.35.0 on Ubuntu 24.04 (hyperv/amd64)
🌟 Using the docker driver based on existing profile
👍 Starting "minikube" primary control-plane node in "minikube" cluster
📡 Pulling base image v0.0.46 ...
🔄 Restarting existing docker container for "minikube" ...
```

Minikube starten (Bild 3)

Kubernetes-Architektur

Gehen wir zunächst auf die Architektur von Kubernetes ein, wie sie in **Bild 1** dargestellt ist.

- **Worker Nodes:** In diesen werden die Workloads der eigentlichen Anwendung ausgeführt, zum Beispiel ASP.NET-Core-API-Dienste oder Telemetrie-Komponenten wie Prometheus oder Grafana.
- **Control Plane:** Der Control Plane kontrolliert die Workloads; zudem werden in diesem sämtliche zentralen Kubernetes-Komponenten ausgeführt. Der Control Plane kann auch über mehrere Nodes verteilt werden, abhängig von den Wünschen/Anforderungen an die Redundanz.
- **CRI:** Das CRI ist eine standardisierte Schnittstelle in Kubernetes, die es dem Kubelet Agent, welcher auf jedem Node ausgeführt wird, ermöglicht, mit verschiedenen Container-Runtime-Umgebungen zu kommunizieren, um etwa Container zu starten oder zu stoppen. Durch diese Schnittstelle können verschiedene Container Runtimes eingesetzt werden, eine feste Bindung etwa an Docker ist nicht gegeben.
- **Kubelet:** Auf jedem (!) Node im Kubernetes-Cluster wird ein Kubelet Agent betrieben. Dabei handelt es sich um die Implementierung einer Schnittstelle, die von Kubernetes für die Kommunikation benötigt wird. Sowohl Kubelet als auch CRI werden normalerweise durch den Systemd-Manager unter Linux gestartet.
- **Core-Komponenten:** Der API-Server, etcd und der Scheduler sind die zentralen Bestandteile in Kubernetes. Diese sind für den erfolgreichen Betrieb zwingend.

Scheduling – das Starten der Workloads

Über das Kubernetes-API werden die Ressourcenanfragen an den API-Server kommuniziert, zum Beispiel „Starte eine Anwendung in einer Ausführungseinheit (Pod)“. Die Ressourcenanfragen werden in den etcd (verteilte Datenbank) geschrieben. Anschließend kommt der Scheduler zum Einsatz. Er bekommt die Ressourcenanfragen aus dem etcd und schickt diese an die Kubelet Agents und berücksichtigt bei der Auswahl des Nodes ein entsprechendes Regelwerk. Der Kubelet Agent startet die Workloads in einem Pod. Die Workloads werden im Cluster verteilt.

Das erwähnte API ist ein von Kubernetes bereitgestelltes REST-API, das über HTTPS erreichbar ist. Über dieses können die CRUD-Operationen für die verschiedenen Ressourcen durchgeführt werden. Auch kubectl verwendet dieses API. Ein besonderes Augenmerk verdient allerdings etcd.

etcd – das Kubernetes-Gedächtnis

Bei etcd [1] handelt es sich um eine verteilte Key-Value-Datenbank, die speziell für verteilte Anwendungen konzipiert ist. Im Kontext von Kubernetes enthält diese sämtliche Konfigurationen, den Status und die gewünschten Zustände des Clusters und wird somit benötigt, um den Ressourcen-Zustand zu verfolgen und zu steuern. In etcd wird unter anderem Folgendes gespeichert:

- Informationen über alle Nodes,
- der aktuelle Status und die Spezifikation aller Pods,
- die Konfiguration von Kubernetes-Ressourcen wie zum Beispiel Deployments und Services,
- ConfigMaps und Secrets.

Die Nutzung von etcd zeigt **Bild 2**. Der API-Server ist die einzige Komponente, die direkt mit etcd kommuniziert; jede Änderung im Cluster wird von diesem an etcd kommuniziert. Werden Abfragen über den aktuellen Zustand etwa über kubectl angestoßen, liest der API-Server diesen aus etcd aus.

Für Kubernetes stellt etcd die folgenden Eigenschaften zur Verfügung:

- **Konsistenz:** Es wird garantiert, dass der Cluster-Zustand korrekt und eindeutig gespeichert ist.
- **Verfügbarkeit:** etcd ist verteilt und hochverfügbar. Es empfiehlt sich, für etcd mehrere Knoten bereitzustellen. Wie der Abgleich der Knoten erfolgt, wird später erläutert. ►

```
christian@christian-Virtual-Machine:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

christian@christian-Virtual-Machine:~$ kubectl get all
NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
service/kubernetes                  ClusterIP     10.96.0.1     <none>         443/TCP    22s
```

Statusabfrage der Core-Komponenten und Ressourcen (Bild 4)

- **Transaktionen:** etcd unterstützt atomare Operationen, etwa für Locking.
- **Snapshot:** Der Zustand des Clusters kann gesichert und wiederhergestellt werden.

Die Anforderung besteht darin, in einem verteilten System eine Konsistenz der Daten zu gewährleisten. Es gibt das sogenannte „Majority Quorum“ [2], welches besagt, dass eine Mehrheit der Knoten einem bestimmten Wert beziehungsweise einer Entscheidung zustimmt. Ein Mehrheitsquorum für Les- und/oder Schreibrechte reicht jedoch nicht aus, um Datenkonsistenz zu gewährleisten. Ursächlich hierfür sind Ausfallszenarien, die dazu führen können, dass Clients inkonsistente Daten sehen, selbst dann, wenn ein Mehrheitsquorum verwendet wird. Fällt beispielsweise ein Knoten aus und kann er nicht wiederhergestellt werden, gehen alle Schreibvorgänge, die noch nicht an die Nachfolger repliziert wurden, verloren.

Eine Lösung hierfür ist das Leader/Follower-Prinzip, wonach ein Server als „Leader“ deklariert wird. Dieser trifft Entscheidungen als „single source of truth“ und kommuniziert diese an alle anderen Server weiter. Nur der Leader verarbeitet den Request. Wird ein Request initial an einen Follower gesendet, delegiert dieser ihn an den Leader weiter.

Auswahl des Leaders

Schauen wir uns die Ermittlung des Leaders etwas genauer an [2]. Im Wesentlichen geht es darum, den Server zu ermit-

teln, welcher „der aktuellste“ ist; nur dieser kann die Rolle des Leaders übernehmen. Die „größte Aktualität“ wird beispielsweise durch folgende zwei Faktoren definiert:

Die aktuellste „Generation Clock“

Hierbei handelt es sich um eine Technik, mit der die Reihenfolge von Ereignissen in einer Reihe von Prozessen bestimmt wird, unabhängig von einer Systemuhr. Jeder Prozess verwaltet einen Counter (Integer), der nach jeder Aktion des Prozesses inkrementiert wird. Jeder Prozess sendet seinen Counter gemeinsam mit den Nachrichten, die er austauscht, an andere Prozesse. Der Empfängerprozess setzt seinen Counter, indem er das Maximum zwischen seinem eigenen Counter und dem Counter der Nachricht wählt. Somit kann jeder Prozess feststellen, welche Aktion vor der anderen stattgefunden hat.

Der zuletzt geschriebene Protokollindex im „Write-Ahead-Log“

Mit diesem Ansatz wird folgendes Problem adressiert: Wenn ein Server zugesagt hat, eine Aktion auszuführen, sollte er dazu auch dann in der Lage sein, wenn es zu Problemen kommt, beispielsweise weil der Server neu gestartet werden muss und der In-Memory-Status verloren geht. Dieser Lösungsansatz besagt, dass jede Statusänderung als einzelnes Kommando in einem Log (zum Beispiel in einer Datei) persistiert wird und sie sequenziell angehängt werden. Für jeden Server-Prozess wird ein eigenes Log vorgehalten.

Minikube – Kubernetes lokal meistern

Der Einstieg in produktive Clusterumgebungen ist oft komplex. Genau hier setzt Minikube an: ein schlankes Werkzeug, das es Entwicklern ermöglicht, einen vollständigen Kubernetes-Cluster lokal auf dem eigenen Rechner zu starten – ganz ohne Cloud oder externe Infrastruktur. Es ist sicherlich kein vollständiger Ersatz für Multi-Node-Cluster, Funktionen wie High Availability oder Load Balancing sind nur simuliert. Dennoch senkt Minikube die Einstiegshürde in die Kubernetes-Welt erheblich.

Minikube simuliert einen Single-Node-Kubernetes-Cluster. Für diesen Beitrag wurde es in einem HyperV-Ubuntu-Image installiert. Dabei bringt es Kernkomponenten mit. Die Container-Runtime (Docker, containerd oder CRI-O) ist wählbar.

Die config-Datei (Bild 5)

```
christian@christian-Virtual-Machine:~$ cat ~/.kube/config
apiVersion: v1
clusters:
- cluster:
  certificate-authority: /home/christian/.minikube/ca.crt
  extensions:
  - extension:
    last-update: Thu, 17 Jul 2025 10:32:15 CEST
    provider: minikube.sigs.k8s.io
    version: v1.35.0
    name: cluster_info
    server: https://192.168.49.2:8443
  name: minikube
contexts:
- context:
  cluster: minikube
  extensions:
  - extension:
    last-update: Thu, 17 Jul 2025 10:32:15 CEST
    provider: minikube.sigs.k8s.io
    version: v1.35.0
    name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/christian/.minikube/profiles/minikube/client.crt
    client-key: /home/christian/.minikube/profiles/minikube/client.key
```


Administration mit kubectl

Bei kubectl handelt es sich um das primäre Command Line Interface für Kubernetes; es kommuniziert mit dem API und bietet zahlreiche Befehle an. **Bild 6** zeigt den Aufruf von

```
source <(<kubectl
completion bash)
```

welches die Tab-Vervollständigung (Auto-Completion) für den kubectl-Befehl in der aktuellen

```
christian@christian-Virtual-Machine:~$ source <(<kubectl completion bash)
christian@christian-Virtual-Machine:~$ kubectl -h
kubectl controls the Kubernetes cluster manager.

Find more information at: https://kubernetes.io/docs/reference/kubectl/

Basic Commands (Beginner):
  create      Create a resource from a file or from stdin
  expose      Take a replication controller, service, deployment or pod and expose it as a new
Kubernetes service
  run         Run a particular image on the cluster
  set         Set specific features on objects

Basic Commands (Intermediate):
  explain     Get documentation for a resource
  get         Display one or many resources
  edit        Edit a resource on the server
  delete      Delete resources by file names, stdin, resources and names, or by resources and
label selector
```

Einstieg in die kubectl-Befehle (Bild 6)

Unter [3] sind im Abschnitt „Get Started!“ die entsprechenden Informationen zur Installation auf verschiedenen Plattformen zu finden. Über den Befehl *minikube start* wird der Single Node Cluster gestartet (**Bild 3**).

Nach dem Start kann mit dem Kommando *minikube status* geprüft werden, ob alle Core-Komponenten betriebsbereit sind, und mittels *kubectl get all* können alle eingerichteten Ressourcen abgefragt werden (**Bild 4**).

Die Nutzung von kubectl erlaubt die Nutzung bestehender Skripte und/oder Helm-Charts. Eine explizite Anmeldung ist nicht notwendig, da die Authentifizierung bereits erfolgte. Standardmäßig geschieht dies über PKI-Zertifikate, die automatisch auf der Maschine generiert werden, auf welcher Minikube installiert wurde.

Im Home-Directory des Benutzers, der Minikube installiert hat, ist die Konfigurationsdatei *config* zu finden. Sie enthält alle relevanten Informationen, das Cluster als *certificate-authority* (obere Markierung in **Bild 5**) und den User (untere Markierung in **Bild 5**).

Der User *minikube* wird über das *client-certificate* und den *client-key* authentifiziert; diese werden übrigens bei der Einrichtung angelegt. Das bedeutet: Bei der Verwendung des Tools kubectl wird ein Request an das API mit dem *client-key* gesendet und von der *certificate-authority* erfolgreich authentifiziert, da diese das Zertifikat ausgestellt hat. Selbstverständlich kann auch mit RBAC authentifiziert werden.

Bash-Shell aktiviert. Der Aufruf von

```
kubectl -h
```

zeigt die Hilfe von Kubernetes. Ein Auszug davon ist ebenfalls in **Bild 6** zu sehen.

Schauen wir uns in aller Kürze noch weitere wichtige Befehle an:

- *kubectl api-resources*: Zeigt alle verfügbaren Ressourcen, siehe **Bild 7**.
- *kubectl api-versions*: Zeigt die Versionen.
- *kubectl explain {resource}*: Zeigt Informationen der Resource-Eigenschaften.
- *kubectl explain -recursive {resource}*: Listet alle verfügbaren Eigenschaften auf.

Neben dem CLI-Tool kubectl gibt es auch noch Kubernetes Dashboards, ein webbasiertes UI zur Verwaltung von Kubernetes. Von Kubernetes sind verschiedene Distributionen erhältlich, die zum Teil ganz unterschiedliche grafische Oberflächen anbieten.

Somit ist eine bestimmte Funktionalität abhängig von der grafischen Oberfläche und aus diesem Grunde an unterschiedlichen Stellen zu finden; kubectl hingegen ist unabhängig von der Distribution immer identisch. Eine Auswahl einiger Distributionen finden Sie in **Tabelle 1**. ►

```
christian@christian-Virtual-Machine:~$ kubectl api-resources
NAME                SHORTNAMES  APIVERSION  NAMESPACED  KIND
bindings            cs          v1          true        Binding
componentstatuses   cs          v1          false       ComponentStatus
configmaps           cm          v1          true        ConfigMap
endpoints            ep          v1          true        Endpoints
events               ev          v1          true        Event
limitranges          limits      v1          true        LimitRange
namespaces           ns          v1          false       Namespace
nodes                no          v1          false       Node
persistentvolumeclaims  pvc        v1          true        PersistentVolumeClaim
persistentvolumes     pv          v1          false       PersistentVolume
pods                 po          v1          true        Pod
```

Befehl api-resources (Bild 7)

Ausführung von Anwendungen

Werden Anwendungen in Kubernetes ausgeführt, geschieht dies final immer in einem sogenannten Pod. Ein Pod ist die kleinste von Kubernetes verwaltete Entität. Innerhalb eines Pods werden ein oder mehrere Container im gleichen Kubernetes-Namespace gestartet. Pods werden IP-Adressen zugewiesen, und meist wird nur ein Container innerhalb eines Pods ausgeführt.

Der Einsatz von Multi-Container-Pods weist darauf hin, dass der Haupt-Container eine gewisse Hilfe durch einen Hilfs-Container benötigt. Einige allgemeine Beispiele hierzu:

- **Sidecar:** Erweitert und/oder optimiert die Funktionalität des Haupt-Containers.
- **Adapter:** Adaptiert den Haupt-Container in ein benötigtes Format. So kann beispielsweise eine Legacy-Anwendung im Haupt-Container ausgeführt werden, und der Hilfs-Container adaptiert dessen proprietäres Protokoll auf ein generisches Protokoll.
- **Init-Container:** Dieser bereitet notwendige Initialisierungsschritte für den Haupt-Container innerhalb des gleichen Pods vor.

Durch das Pod-Konzept werden die Container um Cluster-Funktionalität erweitert. Sie bieten hierfür entsprechende Eigenschaften an:

- **resources:** Legt CPU- und Speicheranforderungen fest.
- **securityContext:** Einem Pod werden bestimmte Berechtigungen gegeben.
- **probes:** Hierbei handelt es sich um Testprogramme, die von dem Pod verwendet werden können, etwa für Liveness, Readiness oder Startup.

Pods können grundsätzlich natürlich direkt gestartet werden; jedoch sollte hiervon Abstand genommen werden. Stattdessen sollte die Ausführung über das sogenannte Kubernetes Deployment erfolgen, einen speziellen Controller-Typ, der ein relevantes Feature-Set zur Verfügung stellt. Es ermöglicht unter anderem einen automatischen Neustart, wenn der Pod stirbt.

In **Bild 8** ist die Anweisung zum Erstellen eines Deployments mit dem Namen *dotnetpro* zu sehen, das auf dem Container-Image *nginx* basiert, sowie der Start von vier Replikationen (rot markiert). Das *nginx*-Image ist recht klein, startet sehr schnell und ist somit hervorragend geeignet für schnelle Pod-Deployments, um beispielsweise statische Inhalte (HTML, JS, CSS) bereitzustellen, als Proxy für Backend-Services (Load Balancing, SSL-Terminierung) oder für Caching und Routing. Die grünen Markierungen in **Bild 8**

```
christian@christian-Virtual-Machine:~$ kubectl create deployment dotnetpro --image=nginx --replicas=4
deployment.apps/dotnetpro created
christian@christian-Virtual-Machine:~$ kubectl get all
```

NAME	READY	STATUS	RESTARTS	AGE
pod/dotnetpro-5dcbc6d5dd-7nlk7	0/1	ContainerCreating	0	23s
pod/dotnetpro-5dcbc6d5dd-9n85g	0/1	ContainerCreating	0	23s
pod/dotnetpro-5dcbc6d5dd-h77tq	0/1	ContainerCreating	0	23s
pod/dotnetpro-5dcbc6d5dd-w7n44	0/1	ContainerCreating	0	23s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	8d

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/dotnetpro	0/4	4	0	23s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/dotnetpro-5dcbc6d5dd	4	4	0	23s

Deployment erstellen (Bild 8)

● Tabelle 1: Einige Kubernetes-Distributionen

Name	Kategorie	Anbieter	Besonderheiten
GKE	Cloud	Google	Deep-GCP-Integration
EKS	Cloud	AWS	AWS-nativ
AKS	Cloud	Microsoft	Azure-nativ
IBM Cloud Kubernetes Service	Cloud	IBM	Optimiert für IBM Cloud
OpenShift	On-Prem	Red Hat (IBM)	Erweiterte Features, RBAC, UI, GitOps
K3s	Lightweight	SUSE (Rancher)	ARM-fähig
k0s	Lightweight	Mirantis	Single-Binary, sehr minimalistisch

zeigen den Abruf der Ressourcen, die in Kubernetes erstellt wurden – vier Pods, ein Deployment und ein Replica-Set.

Fazit

Natürlich ist es nicht möglich, ein so komplexes System wie Kubernetes in einem einzigen Beitrag detailliert zu erläutern, dafür ist es zu umfangreich. Absicht war es, einen ersten Einblick in den einen oder anderen wichtigen und interessanten Aspekt zu liefern – und dadurch womöglich das Interesse zu wecken, sich intensiver mit Kubernetes zu beschäftigen. ■

[1] *etcd*, <https://etcd.io/>

[2] Unmesh Joshi, *Patterns of Distributed Systems*, Addison-Wesley Signature Series, 2023, ISBN 978-0-13-822198-0

[3] Minikube-Doku, <https://minikube.sigs.k8s.io/docs/>



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@googlemail.com.

PDF-DATEIEN ERSTELLEN

Aus AcroForm mach PDF

Erstellen Sie mit iText PDF-Dokumente aus PDF-Formularen.

Die Komponente iText Core [1], früher iTextSharp, unterstützt verschiedene Wege, PDF-Dokumente zu erzeugen: ausschließlich per Code, oder aber mit PDF-Dateien als Vorlage. Dieser Artikel zeigt, wie Sie mit iText ein PDF-Dokument mit Formularfeldern befüllen.

Für eilige Leser

Die nötigen Schritte im Schnelldurchlauf:

- C#-Konsolen-App erstellen.
- NuGet-Pakete *itext* und *itext.bouncy-castle-adapter* hinzufügen.
- **Listing 1** ausführen.
- **Ergebnis:** Datei *document.pdf* im Anwendungsordner (**Bild 1**).

Hinweis: iText ist in der Regel kostenpflichtig – siehe den Abschnitt „iText-Lizenzmodelle“ weiter unten.

AcroForms

Die PDF-Datei hinter dem Link [2] enthält ein Acrobat-Formular (AcroForm) mit dem Textfeld *Text1*. Adobe hat das portable und plattformübergreifende AcroForm-Format 1998 eingeführt (siehe Kasten **Was sind AcroForms?**). Sie haben also

Was sind AcroForms?

AcroForms, kurz für „Acrobat Forms“, sind eine Adobe-Technologie für interaktive Formulare, die in PDF-Dokumenten verwendet werden. Sie bilden eine klassische Standardfunktion der PDF-Technologie und werden von einer Vielzahl von PDF-Viewern und -Softwareprogrammen unterstützt.

AcroForms bieten unter anderem folgende Funktionen und Merkmale:

- Interaktive Formularfelder: Beispiele sind Textfelder, Kontrollkästchen, Optionsfelder und Drop-down-Listen, die Benutzer mit Text, Zahlen oder Auswahlen ausfüllen können.
- Digitale Formularübermittlung: Genutzt wird in der Regel die Übermittlung per E-Mail oder über einen Webserver.
- Formularvalidierung: Validierungsregeln können sicherstellen, dass Benutzer Daten im richtigen Format eingeben.
- Digitale Signaturen: Benutzer können das Dokument elektronisch signieren, um die Authentizität und Integrität des Formulars zu gewährleisten.
- Datenextraktion: In AcroForms eingegebene Daten können zur weiteren Verarbeitung extrahiert werden, was sie für die Datenerfassung und -analyse wertvoll macht.

Listing 1: Textfeld „Text1“ im PDF-Formular füllen

```
using iText.Forms;
using iText.Kernel.Pdf;

var templatePath =
    "https://dotnetpro.de/SL24xxAcroToPdf1"
var reader = new PdfReader(templatePath);
var writer = new PdfWriter("document.pdf");

var pdf = new PdfDocument(reader, writer);
var acroForm = PdfAcroForm.GetAcroForm(pdf, false);
acroForm.GetField("Text1").SetValue(
    $"{DateTime.Now}");
```

vermutlich schon einmal eine AcroForm per Hand ausgefüllt. In **Listing 1** füllt die folgende Zeile das Formularfeld:

```
acroForm.GetField("Text1").SetValue($"{DateTime.Now}");
```

Sie können AcroForms mit Adobe Acrobat oder mit anderen kommerziellen Werkzeugen wie Nitro bearbeiten. Eine kostenfreie Online-Alternative mit Basisfunktionalität bietet PDFescape [3].

Solange der Name des Feldes nicht verändert wird, kann es in der Vorlage verschoben, formatiert oder auch versteckt werden, ohne dass eine Code-Änderung nötig wird. Das

dotnetpro - Aus AcroForm mach PDF

15.08.2024 14:19:47

Ergebnis von Listing 1 (**Bild 1**)

macht den Charme dieser Lösung aus. Die PDF-Vorlage muss nicht zwingend von einem Softwareentwickler erstellt werden, sondern sie kann direkt vom Marketingteam entworfen oder von der Rechtsabteilung getextet werden.

Soll der Inhalt eines Feldes im fertigen PDF-Dokument nicht vom Leser überschrieben werden können, setzt man das Textfeld auf *Read-only* (**Bild 2**). ▶

● Listing 2: Mehrseitiges „Serienbrief“-PDF-Dokument aus einer Vorlage

```
using iText.Forms;
using iText.Kernel.Pdf;

var templatePath =
    "https://dotnetpro.de/SL24xxAcroToPdf1" // iv
var reader = new PdfReader(templatePath);
var writer = new PdfWriter("document.pdf");

var template = new PdfDocument(reader);

var pdf = new PdfDocument(writer);
var acroForm = PdfAcroForm.GetAcroForm(template, false);

foreach (var text in new[] { "Agnes", "Bert", "Cris" })
{
    acroForm.GetField("Text1").SetValue(text);
    pdfTemplate.CopyPagesTo(1,
        template.GetNumberOfPages(), pdf);
}
```

Serienbrief

Bei Serienbriefen wird die gleiche Vorlage jeweils mit verschiedenen Datensätzen befüllt und zum generierten PDF-Dokument hinzugefügt. Hierfür erstellen Sie separate Instanzen der *PdfDocument*-Klasse für die Vorlage und das Ausgabedokument (siehe Listing 2). Sobald die AcroForm mit dem aktuellen Datensatz gefüllt ist, kopiert die *CopyPagesTo*-Methode die Seite(n) aus der Vorlage in die zu erstellende PDF-Datei.

Problemzonen

Alles ist gut, solange AcroForm-Felder und statischer Text unabhängig voneinander an festen Positionen im Dokument stehen. Spannender ist jedoch, variable Inhalte in einem Satz wie „Bezahlen Sie 12 € bis Montag, 3. Mai 2025 ohne Abzüge“ zu positionieren.

Man kann die Felder groß genug machen, damit auch „1.234 €“ beziehungsweise „Donnerstag, 31. Dezember 2025“ hineinpassen, und hat dann bei kürzeren Werten Lücken – unter Designaspekten eher unschön.

Sie können auch den ganzen Satz im Code aufbereiten und in einem Formularfeld ausgeben. Das nähme aber der Rechtsabteilung die Möglichkeit, den Wortlaut (ohne Codeänderung) anzupassen.

Eine Notlösung sind Platzhalter und *string.Format*: In der Vorlage belegt man das Formularfeld mit *Bezahlen Sie {0} € bis {1} ohne Abzüge*. Mit der Methode *GetDisplayValue* liest man diese Formatanweisung und konstruiert den Satz mit *string.Format* (siehe Listing 3).

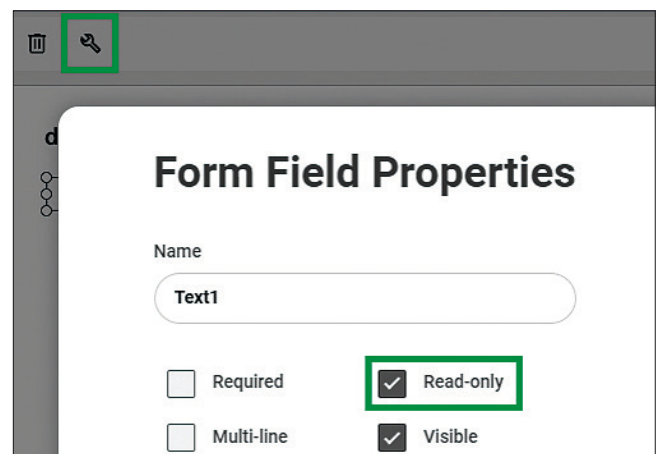
Auch für diese Lösung gilt: Geht so. Offensichtliche Nachteile sind die codeähnlichen Konstrukte *{0}* und *{1}* in der Vorlage. Ein Tippfehler wie *{0}* verursacht im Code eine Format-Exception. Auch kann der kostenlose PDFescape Editor kein

Textfeld vorbelegen. Dafür braucht es zum Beispiel Adobe Acrobat oder Nitro.

iText-Lizenzmodelle

iText ist kostenfrei unter der AGPLv3-Lizenz erhältlich, die aber Kriterien für die konsumierende Applikation voraussetzt [4]. Insbesondere gilt, dass Ihre Applikation als Open Source zugänglich sein muss, wenn sie iText verwendet. Das beschränkt den Einsatz der kostenlosen Lizenz in vielen Unternehmen deutlich.

Den Preis für eine kommerzielle Lizenz erhält man auf Anfrage. Es gibt unter anderem Volumenlizenzen basierend auf der Anzahl der verarbeiteten PDF-Dokumente. Dafür greift iText auf einige Lizenzserver zu. Können diese aufgrund von Netzwerkkonfiguration, Proxyeinstellungen



Nachträgliche Bearbeitung eines Textfelds sperren in PDFescape (Bild 2)

● Listing 3: Notlösung für variable Inhalte zwischen statischen Textteilen

```
var field = acroForm.GetField("Text1");
var format = field.GetDisplayValue();
// -> "Bezahlen Sie {0} € bis {1} ohne Abzüge"

var value = string.Format(
    format, 12, new Date(2025, 5, 3).ToLongDate());
```


et cetera nicht erreicht werden, scheitert das Lesen der Lizenz. In stark regulierten Unternehmensnetzwerken ist das eine Herausforderung, in der Regel aber eine lös-bare.

Vergleich iText – DinkToPdf

In der dotnetpro-Ausgabe 7/2024 erläuterte Christian Kiefer, wie man mit DinkToPdf auf HTML-Basis PDF-Dokumente erstellt [5]. DinkToPdf, eine plattformübergreifende Bibliothek, fungiert als Wrapper für die wkhtmltopdf-Bibliothek. wkhtmltopdf ist ein Open-Source-Befehlszeilentool (LGPLv3), das HTML-Code unter Verwendung einer Web-Kit-Rendering-Engine in PDF umwandelt [6].

Zunächst erfolgt die Installation des Nu-Get-Pakets DinkToPdf. Anschließend wird in der Datei *Program.cs* der Service registriert, damit man diesen mittels Dependency Injection überall in unserem Projekt verwenden kann.

In einer Konfigurations-Klasse wird die Konfiguration (*GlobalSettings*, *ObjectSettings*) für DinkToPdf hinterlegt. In der Anwendung selbst wird die PDF-Datei erstellt, indem man einen HTML-String an DinkToPdf übergibt und eine PDF-Datei als Byte-Array zurück-erhält. Den kompletten Beispiel-Code finden Sie unter [7].

Während iText mit PDF-Vorlagen arbeitet, deren Formularfelder man automatisiert befüllen kann, bietet DinkToPdf die Möglichkeit, von einer Webanwendung aus mittels der Übergabe eines HTML-Codes eine PDF-Datei zu erstellen. Natürlich lassen sich mit Platzhaltern in einem HTML-Generator auch Serienbriefe, Rechnungen oder Barcode-Etiketten als Vorlagen erstellen und später als PDF-Dateien generieren.

Wenn es Anpassungen an der PDF-Datei geben soll, lässt sich mit iText die PDF-Vorlage einfach austauschen, ohne den Code neu zu kompilieren und zu deployen. Beim Einsatz von DinkToPdf dagegen muss der HTML-Code-Generator im Quellcode der Anwendung angepasst und die Anwendung neu kompiliert und deployt werden. **Tabelle 1** zeigt die wesentlichen Unterschiede zu iText Core.

Fazit

Wer besondere Anforderung an das Layout seines PDF-Dokuments hat und dieses schon in der Dokumentvorlage sehen möchte, für den sind AcroForms mit iText ein Ansatz. Die Möglichkeit, IT-fremde Kollegen an den Vorlagen arbeiten zu lassen, ist ebenfalls interessant.

Voraussetzung ist aber ein relativ statisches Layout – Listen oder Tabellen mit variabler Länge erstellt man besser nicht mit AcroForms. Auch für den schmalen Geldbeutel taugt der beschriebene Ansatz nicht. iText ist – außer für Open-Source-Anwendungen – kostenpflichtig. Zudem werden Sie für ein professionelles Design einen kommerziellen Editor wie Acrobat oder Nitro benötigen.

● **Tabelle 1: Unterschiede zwischen iText Core und DinkToPdf**

	iText (Core)	DinkToPdf
Kostenfreie Lizenz	eingeschränkt, AGPLv3 für Open-Source-Anwendungen	ja, MIT License
Kommerzielle Lizenz	ja, Preis auf Anfrage	nein
Support	ja	nein
PDF Acro Forms -> PDF	ja	nein
Code -> PDF	ja	nein
HTML/CSS -> PDF	nein, benötigt das Add-on pdfHTML	ja, wkhtmltopdf-Library
PDF-Attribute setzen	eingeschränkt, Producer-Attribut nur mit kommerzieller Lizenz	nein
PDF lesen und analysieren	ja	nein
PDF-Dateien/-Seiten zusammenfügen	ja	nein
Inhalte mit dynamischer Länge (Listen)	eingeschränkt: nicht mit dem im Artikel beschriebenen Ansatz	ja
Letztes Update in GitHub	itext-dotnet: November 2025	DinkToPdf: 2017; wkhtmltopdf: 2022

DinkToPdf mit wkhtmltopdf hat seine Stärken bei variabel langen Dokumenten mit Tabellen und Listen. Es ist kostenlos, allerdings sind beide Projekte auf GitHub seit Jahren nicht mehr aktiv. Solange sie tun, was sie sollen, muss man nicht umsteigen. Ist das einmal nicht mehr der Fall, stellt iText plus pdfHTML-Add-on eine Migrationsoption dar. ■

- [1] *iText-dotnet auf GitHub*, <https://github.com/itext/itext-dotnet>
- [2] *PDF-Formular für Listing 1*, <https://markus-kraus.de/template.pdf>
- [3] *PDFescape Online-PDF-Editor*, www.pdfescape.com/online-pdf-editor/
- [4] *AGPLv3*, <https://itextpdf.com/how-buy/legal/agpl-gnu-affero-general-public-license>
- [5] *Christian Kiefer, Aus HTML mach PDF*, dotnetpro 7/2024, Seite 136 ff.
- [6] *wkhtmltopdf*, <https://wkhtmltopdf.org>
- [7] *Christian Kiefer, CreatePDFOnWebsites*, <https://github.com/ChristianKiefer/CreatePDFOnWebsites>



Markus Kraus

arbeitet als Full-Stack-Entwickler für einen großen Rückversicherer in München. Sein Fokus liegt zurzeit auf .NET und Angular.

markus.kraus@anizing.net

FOKUSSIEREN STATT VERZETTELN

Einfach mal fertig machen

Auch Entwickler leiden manchmal unter „Abschlusschwäche“, nicht nur Fußballer. Leider haben manche Projekte aber erst einen Nutzen, wenn sie komplett fertig sind.



Time-out Start

Stellen Sie sich vor, Sie haben ein neues Elektroauto bestellt – selbstfahrend, mit komplettem Entertainment-Paket, 700 PS und 1500 km Reichweite. Ihr Traumwagen. Einziges Problem: Der Akku ist leer, und der Hersteller hat noch keinen Adapter geliefert, damit Sie ihn bei sich zu Hause aufladen können. Nach einer Woche gibt der Hersteller zu, er habe den Adapter noch gar nicht bestellt, werde es jetzt aber ganz bestimmt tun. Zwei Wochen später heißt es, der zuständige Mitarbeiter sei zurzeit in Urlaub. Drei Wochen später erfahren Sie dann, der Adapter sei jetzt zwar da, aber es liege noch kein Zulassungsbescheid der Behörde vor, weshalb man ihn noch nicht ausliefern dürfe. Und so weiter und so fort.

Zu wie viel Prozent ist das Projekt Ihrer Meinung nach abgeschlossen? Ich meine, Sie haben die Rechnung immerhin voll bezahlt, das Auto ist geliefert, es steht in Ihrer Garage. Nur lässt es sich leider keinen Meter fahren. Na dann ...

Weshalb dieses Beispiel? In den letzten Wochen hatten wir eine Messe vorzubereiten. Dazu mussten Videos gedreht werden über die neue Anwendung. Selbstredend war sie noch nicht fertig, sondern wurde erst zur Messe finalisiert, und es sah noch nicht alles wirklich zu 100 Prozent perfekt aus – mal war ein Abstand etwas zu eng, weil ein Margin gefehlt hatte, oder die Demo-Daten waren nicht entsprechend aufbereitet. Also haben wir in der Entwicklung nachgepflegt und Farben und Abstände korrigiert. Dieses Spielchen zog sich über Wochen (!) hin, und am Ende war nicht sicher, ob die Videos bis zur Messe überhaupt fertig würden. Immer wieder fand man etwas, das nicht zu 100 Prozent sauber war. Ich nenne so was „in Schönheit sterben“. Sterben deshalb, weil es keine Videos gibt, wenn man nicht zu einem Ende kommt. Nicht

schlechte, sondern gar keine. Und dann ist die gesamte Arbeit umsonst gewesen. Es gibt eben Projekte, die erst dann einen Nutzen haben, wenn sie komplett abgeschlossen sind.

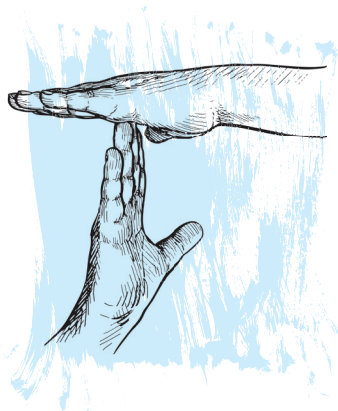
Added Value? Erst bei 100 Prozent!

Wir alle kennen das sicher zur Genüge: Man hat zwei oder mehrere Projekte offen und stößt manchmal auf eine Aufgabe, auf die man gerade keine Lust hat oder die mehr Hirnschmalz als sonst erfordert. Dann macht man halt an etwas anderem weiter. Und lässt das Projekt zu 80 Prozent so, wie es ist. Doch in vielen Fällen sind 80 Prozent wie null Prozent. Nicht unbedingt aus unserer Sicht, aber aus der des Kunden oder zumindest aus der unserer Finanzabteilung, die erst dann eine finale Rechnung stellen kann, wenn der Kunde das Projekt abgenommen hat.

Auch hierfür habe ich einen Ausdruck: „Added Value“. Wir stellen Software her, weil wir damit einen Kundennutzen bezwecken. Und erst wenn dieser Added Value vorhanden ist, waren wir erfolgreich. Es heißt dabei ausdrücklich nicht „Maximum Value“. Es geht nicht darum, dass der Kunde am Schluss alle seine Probleme lösen kann, aber zumindest sollten es mehr gelöste als vorher sein. Und erst wenn ein spürbarer Nutzen da ist, sind wir einen Schritt weiter.

Vielleicht gibt es so etwas wie eine Angst vor der Fertigstellung, ähnlich wie bei einem Stürmer angesichts des Tors.

Wenn man etwas abgibt, was eingesetzt wird und bei dem man nicht noch etwas debuggen oder ändern und optimieren kann. Ist es die Angst, Kontrolle zu verlieren? Oder zu dem zu stehen, was da ist, so wie es ist? Ich weiß es nicht. Was ich aber weiß: Es kann auf der anderen Seite eine unheimliche



Befriedigung geben, wenn ein Projekt abgeschlossen ist, und dass uns dieses Gefühl über die „Mühen der Ebene“, über unseren Perfektionismus, über das „Ich habe gerade keine Lust auf ...“ hinweghelfen kann.

Wenn Sie in sich gehen, wird es auch bei Ihnen viele solcher 80-Prozent-Projekte geben. Darunter sind auch solche, die wir niemals zu Ende bringen: Wir haben ein Gespür dafür, dass das nichts wird, reden wir uns aber trotzdem ein, dass wir das schaffen. Besser wäre es allerdings, Zöpfe abzuschneiden und Projekte, die keine Zukunft haben, zu beenden. Ich denke, Sie wissen genau, was ich meine.

Ich habe vor ein paar Wochen einen interessanten Podcast gehört, den ich jedem nur empfehlen kann [1]. Dessen Autor verwendet in einer Episode das Bild eines Surfers, der immer dann vom Brett fällt, wenn er nach unten auf das Brett schaut [2]. Erst wenn sein Blick in die Weite geht, zum Horizont, und dort ein Ziel erkennbar ist, erst dann bleibt er auf dem Brett stehen. Ein passendes Bild, wie ich finde. Übertragen auf die Projektarbeit bedeutet das: Erst in der Abarbeitung entwickelt sich der nötige Automatismus, die Routine, die uns hilft, unbewusst auf dem Brett stehen zu bleiben.

Der Blick zum Horizont hilft

Was heißt das nun aber konkret? Angenommen, Sie switchen ständig zwischen den Projekten, damit wenigstens etwas weitergeht. Sie fühlen sich ausgelaugt, und alles ist immer nur zu 80 Prozent fertig. Man läuft der Zeit hinterher. Dann wird es Zeit, sich Zeit zu nehmen. Denn ohne eine saubere Reflexion – und die allein braucht sicher schon eine Stunde – wird das nicht besser. Sie brauchen ein Time-out!

Als Erstes schreiben Sie alle Projekte, die in Ihrem Kopf herumswirren, auf einen Zettel. Nicht nur die Namen der Projekte, sondern auch so kleine Dinge wie „Releasenotes aktualisieren“. Auch wenn Sie glauben, dass das unnötig ist: Tun Sie es! Alles, was Ihnen durch den Kopf geht. Sie werden feststellen, dass weniger zusammenkommt, als Sie denken. Unser Bewusstsein kann maximal sieben Dinge gleichzeitig im Kopf haben. Bei Projekten würde ich sogar von maximal vier oder fünf ausgehen. Bei mehr als fünf Dingen gibt es den Stack Overflow, der uns nachts nicht schlafen lässt. Das Gefühl, dass einem alles über den Kopf wächst und man nicht hinterherkommt. Wenn alles auf einem Zettel ist, dann ist die Leistung, sich zu merken, was man tun will, weg und der Kopf frei für etwas anderes. Ich kann einschlafen und alles vergessen, denn wenn ich morgen auf den Zettel schaue, ist alles wieder da.

Danach geht es um die Priorisierung. Kann ich dem Kunden sagen, dass es noch zwei Wochen dauert? Ist das Feature jetzt wirklich wichtig, oder reicht das auch später? Haben wir einen Workaround dafür? Bei der Frage der Prioritäten geht es auch darum, ob Sie alle Informationen haben oder noch abhängig von anderen Mitarbeitern oder Kunden sind. Wenn dem so ist, sollten Sie zuerst alle Informationen einholen. Jetzt! Und wo wir schon bei „anderen“ sind: Rechnen Sie ausreichend Puffer ein, wenn nach Ihrer Arbeit Kollegen weitermachen, also etwa die Lösung deployen oder testen müssen.

Wenn die Prioritäten klar sind, kommt die wichtigste Frage: Bin ich dafür überhaupt zuständig – oder kann ich es delegieren?

Und überhaupt, ist das ein Projekt, das wir verfolgen sollten? Habe ich die Motivation, es wirklich bis zum Ende zu bringen, oder jongliere ich hier nur Bälle, damit wir im Spiel bleiben und ich nicht zugeben muss, dass ich darin eigentlich keine Zukunft sehe?

Ist die Liste dann final, erwartet Sie etwas Schönes: Sie suchen sich zwei kleine Aufgaben heraus, die schnell zu erledigen sind. Die mögen zwar nicht wichtig sein, aber es gibt die nötige Motivation, die entsteht, wenn man etwas von der Liste streicht. Nur zwei Kleinigkeiten, und schon ist die Liste kleiner. Wunderbar! Wir sind mit unserer Methode auf dem richtigen Weg, denn der eigene Kopf wird freier.

Dann der schwierige Punkt: Das erste Projekt wird zu Ende gebracht. Und damit meine ich wirklich: zu Ende. Sollten Sie zehn Minuten warten müssen, bis das Windows-Update auf dem Zielrechner installiert ist, dann warten Sie zehn Minuten. Sie können Sie nutzen, um zu überlegen, wie es in diesem einen Projekt weitergeht, ob das alles so okay ist, ob die nächsten Schritte klar sind. Aber nicht, um auf dem zweiten Rechner Swagger ins nächste Projekt einzubinden. Denn das erste Projekt blockiert die Ressourcen so lange, bis es abgeschlossen ist.

Fazit

Es ist immer wieder faszinierend zu sehen, wie schnell Projekte fertig werden, wenn wir uns nicht überfordert fühlen, wenn wir ganz bei der Sache sind und genügend Energie da ist, Ablenkungen immer wieder zu widerstehen, Wichtiges vom Nebensächlichen zu trennen. Und welche Kraft entsteht, wenn man alte Zöpfe mutig abschneidet.

In diesem Sinne: Happy Coding! ■

[1] Podcast Sam Webster Harris, Growth Mindset Psychology, <https://open.spotify.com/show/2rQ6Aug8EBudJMSBYTi8nM>

[2] Podcast Sam Webster Harris, Growth Mindset Psychology, How to Create Perspective and Get Out of Your Own Way, <https://open.spotify.com/episode/1zo3Tskl4XHLgnWyfINv8x>



Time-out End



Dipl.-Theol. Bernhard Pichler

war 20 Jahre 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.

ARBEITSMARKT

TRENDS UND JOBS FÜR ENTWICKLER

Octoverse-Report

Frische Zahlen

In der aktuellen Ausgabe seines jährlichen Octoverse-Reports meldet GitHub erstaunliche Zahlen. So hat der Dienst das Anwachsen der Zahl der Entwickler, die GitHub nutzen, auf über 180 Millionen verkündet. Im vergangenen Jahr hat sich laut GitHub durchschnittlich in jeder Sekunde ein neuer Entwickler angemeldet, insgesamt rund 36 Millionen – das ist das stärkste absolute Wachstum an Nutzern, das der Dienst jemals hatte. Allein rund fünf Millionen Entwickler aus Indien haben sich im vergangenen Jahr bei GitHub neu registriert. Damit liegt das Land nun auf dem zweiten Platz der Herkunftsländer. Deutschland belegt hier mit 4,4 Millionen Entwicklern jetzt den siebten Platz (Vorjahr 6. Platz), dicht gefolgt von Indonesien, das gleich drei Ränge geklettert ist und dabei Russland und Kanada überholt hat, siehe **Bild 1**.

Ein Blick auf das im Octoverse Report enthaltene Chart zeigt, dass 2024 nach rasantem Wachstum 2022 und 2023 erstmals eine Stagnation bei der Zahl der Kundschaft eingetreten ist; allerdings startete mit dem Erscheinen der freien Copilot-Version im Dezember 2024 umgehend ein neuer Wachstumsschub. Nahezu 80 Prozent der neuen Entwickler nutzten Copilot bereits in ihrer ersten Woche auf GitHub.

Durch die vielen neuen Entwickler hat sich auch die Reihenfolge der von GitHub-Entwicklern am meisten genutzten Programmiersprachen verändert. Python ist jetzt nur noch die zweitbeliebteste Sprache. Mit einem Zuwachs von knapp 67 Prozent hat TypeScript – zuletzt auf Rang 3 – sowohl JavaScript als auch Python überholt. Auf den Plätzen 4 und 5 folgen Java und C# (**Bild 2**). Beide haben in diesem Jahr jeweils über 100 000 Mitwirkende hinzugewonnen, was laut GitHub ein stetiges Wachstum in

großen Unternehmens- und Spieleentwicklungsumgebungen zeigt. Kernsprachen für die meisten modernen Entwicklungen sind Python, JavaScript, TypeScript, Java, C++ und C#: Fast 80 Prozent der neuen Repositories verwendeten nur diese Sprachen.

Kurios: COBOL tauchte im GitHub-Datensatz mit fast 3000 aktiven Entwicklern auf – wahrscheinlich angetrieben, so GitHub, von Entwicklern, die KI-gestützte Tutorial-Repositories erstellen, um Legacy-Codbasen zu modernisieren.

Freshworks

Chancen für Berufseinsteiger

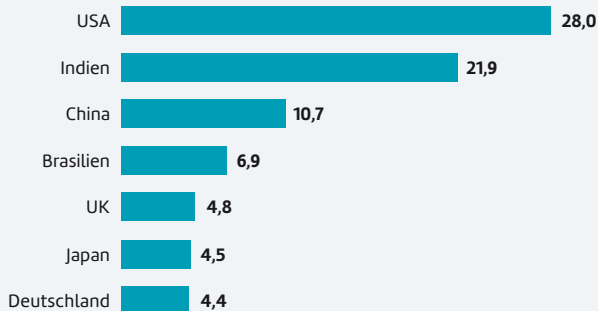
Simon Hayward, General Manager und VP of Sales bei Freshworks, beleuchtet die Chancen für Berufseinsteiger: Aktuell verlieren klassische Junior-Positionen an Bedeutung, weil einfache Tätigkeiten automatisiert werden. Dabei sind Unternehmen langfristig auf den

Nachwuchs angewiesen, um Führungspersönlichkeiten der Zukunft zu entwickeln.

Während einfache Tätigkeiten zunehmend automatisiert werden, wächst der Bedarf an Menschen, die diese Systeme verstehen, kritisch hinterfragen und kreativ weiterentwickeln können. Gerade Berufseinsteiger rücken dadurch stärker in den Fokus: Ihre Fähigkeit, früh Kompetenzen im Umgang mit KI aufzubauen, entscheidet nicht nur über ihre eigene Karriere, sondern auch über die Zukunftsfähigkeit der Unternehmen, in denen sie arbeiten. Wer solche Fähigkeiten mitbringt, ist nicht ersetzbar, sondern gestaltet neue Rollen und erschließt zusätzliche Wertschöpfung.

Damit junge Talente KI souverän nutzen können, müssen Unternehmen allerdings die richtigen Voraussetzungen schaffen. Dazu gehören praxisnahe Trainings, klare Informationen über Möglichkeiten und Grenzen sowie eine Kultur, die Mitgestaltung ermöglicht.

● Herkunft der GitHub-Entwickler

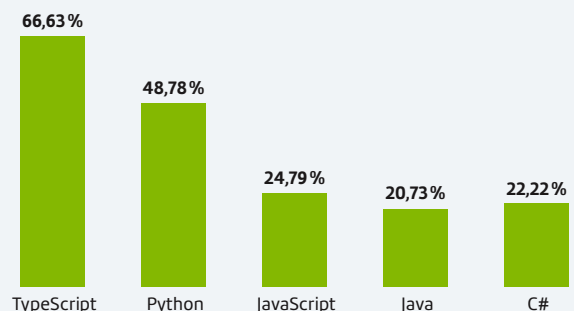


Die meisten Entwickler auf GitHub kommen aus den USA. Mit einem Zuwachs von rund 5 Millionen Entwicklern belegt Indien jetzt Platz 2 (**Bild 1**)

Quelle: GitHub Octoverse Report 2025

Angaben: in Millionen

● Wachstum der beliebtesten Sprachen



Der Anteil von TypeScript ist im Vergleich zum Vorjahr um zwei Drittel gewachsen. C# hat zwar etwas stärker zugelegt als Java, bleibt aber dennoch auf Platz 5 (**Bild 2**)

Quelle: GitHub Octoverse Report 2025

Zahl des Monats

GitHub meldet **395 Millionen** öffentliche und Open-Source-Repositories. Das sind rund 20 Prozent mehr als im Vorjahr. Die Zahl der Beiträge für diese Repositories ist im Jahresvergleich um rund 13 Prozent gestiegen, auf **1,128 Milliarden**. Die meisten der Beiträge stammen von US-amerikanischen Entwicklern, gefolgt von Indern, Brasilianern und deutschen Entwicklern.

Quelle: GitHub Octoverse Report 2025

Expleo AI Pulse

KI-Blase und Job-Sorgen

Die Umfrage „Expleo AI Pulse“ misst auf einer Skala von 0 bis 100, wie besorgt oder zuversichtlich Führungskräfte dem Thema KI gegenüberstehen. Im September 2025 kletterte der europäische Sentiment-Score auf 69 Punkte. Britische Manager äußern sich am optimistischsten und erreichen 74 Punkte, deutsche Führungskräfte erreichen 68 Punkte und ihre französischen Kollegen 66 Punkte. Expleo befragt für die Auswertung jeden Monat 200 Entscheider pro Land zu ihrem Vertrauen in KI, ihren Sorgen um den Arbeitsplatz, ihrer Zuversicht und ihrer Haltung zu ethischen Fragen.

Neue, disruptive Technologien werden in der Anfangszeit häufig von Hype-Zyklen begleitet. Ob dies auch bei KI der Fall ist, wurde im Rahmen der Umfrage im September 2025 erfragt. Das Ergebnis: Eine Mehr-

heit der deutschen Führungskräfte sieht keine KI-Blase – lediglich 44 Prozent teilen diese Einschätzung. In Großbritannien (52 Prozent) und Frankreich (51 Prozent) sieht dagegen mehr als die Hälfte der Führungskräfte derzeit eine KI-Blase (Bild 3).

Die potenzielle Bedrohung des eigenen Jobs durch KI sehen Manager in deutschen Unternehmen entspannt. Nur 29 Prozent zeigen sich besorgt über mögliche Auswirkungen von KI auf ihre berufliche Situation. In Frankreich liegt die Besorgnis bei 44 Prozent und hat damit merklich zugenommen, in Großbritannien sind es 34 Prozent. Am weitesten verbreitet sind Jobängste im mittleren Management. Höhere Managementebenen sehen ihre Tätigkeit im Unternehmen weniger von KI bedroht.

Obwohl sich 39 Prozent der befragten Manager in Deutschland Sorgen über die transformativen Auswirkungen von KI auf ihre Organisation machen, zeigen sich zugleich 74 Prozent

begeistert von den Chancen. Bemerkenswert ist das hohe Selbstvertrauen: 78 Prozent der Führungskräfte sind zuversichtlich, dass ihre Organisation KI erfolgreich einsetzen kann, und 77 Prozent vertrauen auf einen ethisch verantwortungsvollen Umgang (Bild 4).

Cycode

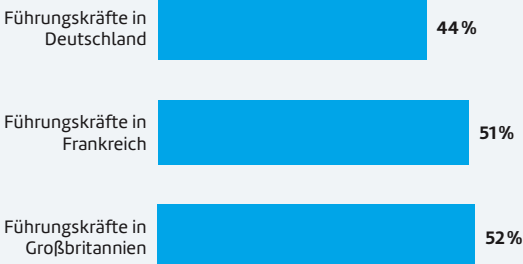
Die generierte Gefahr

Large Language Models schreiben Code in Sekunden. Das hat allerdings auch eine Schattenseite. Jochen Koehler, Vice President Sales EMEA bei Cycode, warnt: „Programmieren ist dank GenAI so einfach wie noch nie. Fast jeder kann damit im Handumdrehen mehr oder weniger guten Code erzeugen – allerdings schlagen Studien bereits Alarm: Ganze 62 Prozent aller KI-generierten Codes sind fehlerhaft oder verwundbar. Noch gravierender: Nach nur fünf Iterationen von KI-unterstützten Verbesserungen steigt die Zahl kritischer Sicherheitslücken um

fast 38 Prozent. Wird Code also immer unsicherer, je häufiger KI im Entwicklungsprozess eingesetzt wird? Das ist sicherlich immer abhängig von Modell und Entwickler. Eins ist aber sicher: Ohne eine Sensibilisierung und entsprechende Gegenmaßnahmen kann das Versprechen der Effizienz schnell in ein enormes Risiko umschlagen. Schließlich optimiert GenAI auf Funktionalität, nicht auf Sicherheit und Resilienz. Damit verschiebt sich das Gleichgewicht zwischen Innovation und Sicherheit weiter zulasten der Stabilität digitaler Systeme.“

Koehler fordert daher: „Künstliche Intelligenz kann ein mächtiges Tool sein – aber nur, wenn wir Sicherheit von Anfang an mitdenken. Dazu gehören strukturierte Code-Reviews, automatisierte Schwachstellenanalysen und Security-by-Design-Prinzipien. Nur wer trotz automatisierter Code-Generierung das letzte Wort behält, kann die Vorteile von KI nutzen, ohne neue Risiken zu schaffen.“

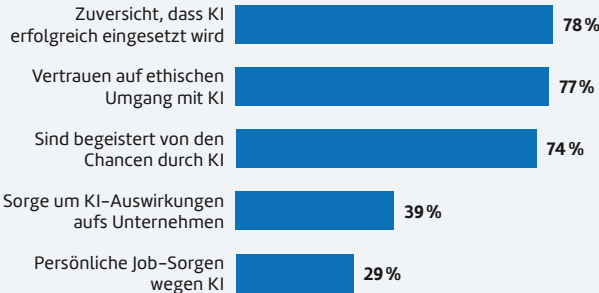
● Gibt es derzeit eine KI-Blase?



44 Prozent der befragten Führungskräfte in Deutschland bestätigen, dass es derzeit eine KI-Blase gibt. In Frankreich und Großbritannien tun dies deutlich mehr (Bild 3)

Quelle: Expleo AI Pulse, Ausgabe September 2025, n=600, Mehrfachantworten

● Manager in Deutschland und Künstliche Intelligenz



Die Manager sind zuversichtlich, dass KI in ihren Unternehmen erfolgreich umgesetzt wird. Nur 29 Prozent machen sich Sorgen um KI-Auswirkungen auf ihren eigenen Job (Bild 4)

Quelle: Expleo AI Pulse, Ausgabe September 2025, n=600, Mehrfachantworten

WISSEN ALS WETTBEWERBSVORTEIL

Gemeinsam stärker

Warum individuelle Teamtrainings für Softwareentwickler entscheidend sind.

Softwareentwicklung verändert sich in rasantem Tempo. Neue Frameworks, Tools und Paradigmen entstehen im Monatsrhythmus, und kaum ein Bereich bleibt lange stabil. Cloud-Technologien, KI-Modelle und moderne Architekturansätze verändern Entwicklungsprozesse grundlegend. Wer Schritt halten will, braucht mehr als Weiterbildung von der Stange: Es braucht eine Lernkultur, die sich am konkreten Bedarf orientiert. Individuelle Trainings spielen dabei eine zentrale Rolle, weil sie Inhalte, Tiefe und Tempo auf den Projektkontext abstimmen.

Der Projekterfolg hängt dabei unmittelbar vom Wissen und den Fähigkeiten der Entwickler:innen ab; fehlende Kenntnisse führen schnell zu Verzögerungen oder Qualitätsproblemen. Individuelle Trainings bieten hier einen klaren Vorteil: Sie setzen am Projektkontext an und adressieren die spezifischen Herausforderungen, denen sich Teams stellen müssen. Statt allgemeiner Inhalte stehen praxisnahe Lösungen im

Mittelpunkt, die sich sofort anwenden lassen, und zwar im Verbund. Auf diese Weise entsteht ein direkter Mehrwert – für die Teilnehmenden wie für das Projekt.

Qualität und Effizienz durch maßgeschneiderte Lerninhalte

Standardisierte Schulungen vermitteln Wissen, doch oft fehlt der Bezug zur Realität im Entwickleralltag. Individuelle Trainings berücksichtigen dagegen den vorhandenen Kenntnisstand, die eingesetzten Technologien und die Arbeitsweise des Teams. Erfahrene Trainer:innen entwickeln Lernkonzepte, die auf tatsächliche Bedürfnisse wie Clean Code, Testautomatisierung, DevOps-Prozesse oder durchdachtes Softwaredesign abheben. Daraus resultieren ein höherer Praxistransfer, geringere Reibungsverluste, spürbar bessere Ergebnisse. Teams, die regelmäßig gemeinsam lernen, teilen ein einheitliches Qualitätsverständnis und treffen fundiertere technische Entscheidungen.

Anzeige



DWX
ACADEMY

**JETZT
BERATUNG
ANFRAGEN**

**Maßgeschneiderte
Trainings für starke
Entwicklerteams**

Nachhaltiger Wissenstransfer und Teamkultur

Ein weiterer Vorteil solch maßgeschneiderter Trainings liegt im nachhaltigen Wissenstransfer. Neues Wissen bleibt nicht bei Einzelnen hängen, sondern verteilt sich über das gesamte Team. Wenn alle auf demselben Wissensstand agieren, entstehen weniger Missverständnisse. Code-Reviews werden effizienter, und die Zusammenarbeit verbessert sich. Gemeinsames Lernen stärkt zudem Motivation, Bindung und Verantwortungsbewusstsein – elementare Faktoren, um Fachkräfte zu halten und Entwicklungskapazitäten stabil zu halten. Solche Lernprozesse wirken dabei über den Moment hinaus: Sie fördern eine Kultur des kontinuierlichen Austauschs, in der Wissen geteilt und weiterentwickelt wird. „Continuous Learning“ wird zu einem integralen Bestandteil der Teamidentität – nicht als Pflicht, sondern als gemeinsamer Motivations- und Erfolgsfaktor.

Lernen im eigenen Rhythmus

Flexibilität ist entscheidend. Viele Teams bevorzugen Formate, die sich in den Arbeitsalltag integrieren lassen – etwa modulare Online-Workshops, kompakte Inhouse-Sessions oder hybride Modelle. Wichtig ist, dass Lernen nicht als Unterbrechung, sondern als Teil des Entwicklungsprozesses verstanden wird. Kleine Gruppen, interaktive Übungen und praxisorientierte Aufgaben fördern Dialog und Feedback. So wird Weiterbildung zu einem kontinuierlichen Element professioneller Softwareentwicklung, das nicht nur Wissen vermittelt, sondern Zusammenarbeit und Qualität stärkt.

neller Softwareentwicklung, das nicht nur Wissen vermittelt, sondern Zusammenarbeit und Qualität stärkt.

Messbarkeit und Wirkung

Damit Trainings nicht als „nice to have“ verpuffen, braucht es klare Ziele und Messpunkte. Sinnvoll sind vereinbarte Outcome-Kriterien wie verbesserte Build-Zeiten, sinkende Defektquoten, höhere Testabdeckung oder schnellere Lead-Time. Auch qualitative Indikatoren helfen: strukturiertere Architekturentscheidungen, konsistentere Code-Reviews oder weniger Kontextwechsel. Regelmäßige Retrospektiven nach Trainingsmodulen sichern den Lerneffekt, justieren Inhalte nach und verankern Verbesserungen im Prozess. So entsteht eine nachweisbare Wirkungskette von Lernen über Verhalten hin zu Ergebnis.

Fazit

Individuelle Teamtrainings sind kein Luxus, sondern eine strategische Notwendigkeit. Sie sichern den Projekterfolg, erhöhen die Softwarequalität und schaffen gleichsam nebenbei eine gemeinsame Wissensbasis, auf der Teams langfristig aufbauen können. Gerade in einer Branche, in der Technologien und Methoden in kurzer Folge wechseln, ist gezielte, praxisnahe Weiterbildung der Schlüssel, um nicht nur Schritt zu halten, sondern die Zukunft aktiv mitzugestalten und Innovationen voranzutreiben. ■

Wissen, das Projekte voranbringt

Architektur | AI und LLM | Azure | C#/.NET
CI/CD | Cloud | Collaboration | Container
Data & Analytics | DevOps | Git | Java
Websprachen | Softwarequalität | Testing | ...



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





LERNSTOFF

NEUE BÜCHER UND MEHR

CYBERSICHERHEITSPLATTFORM

WAZUH

Die Open-Source-Sicherheitsplattform WAZUH kombiniert Intrusion-Detection-Systeme, File Integrity Monitoring und Vulnerability Management, erlaubt also, Angriffe und nicht autorisierte Datenveränderungen frühzeitig zu erkennen, und hilft bei der Lokalisierung von Schwachstellen.

Der Titel startet mit einer kompakten Einführung in die Grundlagen der Netzwerküberwachung: Threat Intelligence, MISP und das MITRE ATT&CK Framework stehen im Mittelpunkt, außerdem wird die Integration von Security Information and Event Management (SIEM) und Security Orchestration Automation and Response (SOAR) in die Sicherheitsinfrastruktur erläutert. Den Schritt in die Praxis leistet der Aufbau einer Testumgebung in Form eines kleinen separaten Testnetzwerks von virtuellen Maschinen mit Windows, Linux oder macOS, um dann eigene Agenten einzurichten, Endpoint Security zu implementieren und Threat Intelligence sowie Security Operations in der eigenen Umgebung zu überwachen. Ergänzt wird dies um die Themen Cloud-Sicherheit



Frank Neugebauer

Cybersicherheit mit WAZUH, Bedrohungen mit Open-Source-Software erkennen, analysieren und abwehren

Hanser, 2025, 361 Seiten, Deutsch, ISBN 978-3-446-48503-7, 49,99 Euro

Zielgruppe: Cybersicherheit

Voraussetzungen: keine

Leseprobe:

<https://tinyurl.com/HanserWAZUH>

und aktive Reaktionen auf Cyberbedrohungen. Um die Leistungsfähigkeit von WAZUH zu demonstrieren, werden Musterbeispiele mit typischen Bedrohungsszenarien vorgeführt, die Sie mit WAZUH erkennen und beheben können, wie Brute-Force-Angriffe, SQL-Injection-Attacken, Malware-Erkennung oder die Überwachung von Docker-Containern und Firewall-Systemen wie pfSense.

Das letzte Kapitel zeigt das Zusammenspiel mit Cybersicherheits-Tools wie The Hive, Shuffle und Cortex, die auch die Automatisierung von Arbeitsabläufen erlauben. (vh) ■

DURCHBLICK IM NORMEN-DSCHUNGEL

IT-Standards

Der Kompass IT-Standards [1] ist ein praktischer Wegweiser, den der Branchenverband Bitkom insbesondere für kleine und mittelständische Unternehmen konzipiert hat, die digitale Produkte entwickeln und auf dem europäischen Markt anbieten. Das Online-Angebot soll im immer komplexer werdenden Feld der IT-Standardisierung eine Orientierungshilfe geben und eine zentrale Anlaufstelle für die Verantwortlichen schaffen.

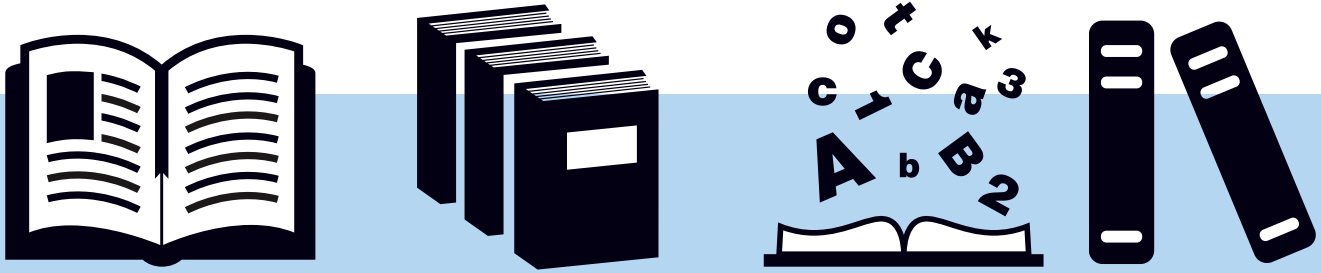
Der Kompass sammelt IT-Standards und Normen an einem Ort und versucht unter anderem Fragen wie „Welche Normen sind für mein digitales Produkt relevant?“ und „Wie hängen EU-Gesetze wie der AI Act mit technischen Standards zusammen?“ zu beantworten. Der Schwerpunkt liegt dabei auf den Bereichen Informationssicherheit, Datenschutz, digitale Identitäten und Künstliche Intelligenz, die auch die erste Gliederungsebene darstellen. Vorgeschaltet sind eine Einführung in das Thema Normen und Standards inklusive eines

Charts, welches die internationalen, die EU- und die nationalen Normierungsorganisationen auflistet sowie einen Überblick zur gesetzlichen Relevanz von Normen liefert.

Zu Regelungen, die gerade eingeführt oder geändert werden, wie dem AI Act oder der eIDAS-Verordnung, liefert der Kompass Timeline-Grafiken, aus denen die weiteren Planungen hervorgehen und wann Regelungen voraussichtlich verpflichtend werden. So ist zum Beispiel für Dezember 2026 die Einführung der EUDI-Wallet geplant, die einen EU-weit gültigen Ausweis auf das Smartphone bringen soll.

Wer tieferegehende Informationen benötigt, findet neben Links zu den Erklärungsseiten der EU am Ende jeder Detail-Seite noch Links zu DIN Media sowie zum VDE-Verlag. Allerdings führen die Links stets auf die Startseiten dieser Anbieter, auf denen man nach den gewünschten Texten suchen muss, die in der Regel kostenpflichtig sind. Als Ansprechpartner für eventuelle Fragen sind im Kompass IT-Standards die Bitkom-Referentinnen für Datenschutz und Digitale Identitäten sowie der Bitkom-Referent für Künstliche Intelligenz und der Bereichsleiter Sicherheitspolitik über ein Kontaktformular zu erreichen. (bl)

[1] www.bitkom.org/Kompass-IT-Standards ■



LARGE LANGUAGE MODELS NUTZEN

Prompting

Um die ständig wachsenden Sprachmodelle der angebotenen KI-Tools mit Gewinn nutzen zu können, bedarf es der Fertigkeit, sinnvolle Abfragen und Aufgaben zu formulieren, die hochwertige Ergebnisse anstoßen und vor Halluzinationen schützen. Die Autoren legen deshalb großen Wert darauf, ein grundlegendes Verständnis für die Rolle dieser Textvervollständigungsmaschinen, unterstützt durch typische Beispiele, zu vermitteln. Zunächst werden die Erweiterung von LLMs zu den inzwischen omnipräsenten Chat-Engines und die neueren Entwicklungen von LLM-Anwendungen vorgestellt.

Der zweite Teil geht auf die verschiedenen Techniken des Prompt Engineering ein. Das reicht weit über einfache Abfragen hinaus. Dazu gehören Verfahren zur Optimierung der Abfragen, das Einbinden von Kontextinformationen, der Rückgriff auf frühere Abfragen, der Anstoß von Codegenerierungen, das Organisieren in Vorlagen sowie Möglichkeiten, das Formatieren der Ausgabe zu beeinflussen.



John Berryman, Albert Ziegler

Prompt Engineering für Large Language Models. LLM-basierte Anwendungen entwickeln, steuern und optimieren

O'Reilly, 2025, 318 Seiten, Deutsch, ISBN 978-3-96009-270-4, 39,90 Euro

Zielgruppe: KI-Anwendungsentwicklung

Voraussetzungen: keine

Leseprobe: www.assets.dpunkt.de/leseproben/14070/Leseprobe.pdf

In der Folge stellen di vor, bei denen über den Prompt dialogbasierte Agenten und LLM-gesteuerte Workflows möglich gemacht werden, etwa um nach dem Auswerten vorhandener Listen gezielt E-Mails verschicken zu können. Beschrieben wird, wie der Kontext für solche aufgabenbasierten Interaktionen organisiert, Gespräche verwaltet und die Benutzererfahrung ausgewertet werden können. Außerdem werden Techniken zur Evaluierung von LLMs vermittelt. (vh) ■

● Kurz vorgestellt

Daniil Maslyuk, Gil Raviv

Power Query

Anhand von über 200 Übungsaufgaben und Schritt-für-Schritt-Anleitungen lernen die Leser, Daten aus verschiedenen Quellen zusammenzuführen, zu transformieren und zu analysieren. Plus Übungsdateien zum kostenlosen Download.

Microsoft Press, 2. Auflage, 2025, 484 Seiten, Deutsch, ISBN 978-3-98889-045-0, 39,90 Euro

Jay Alammar, Maarten Grootendorst

Praxisbuch Large Language Models

Einführung in die Welt der LLMs. Erklärt werden Architekturen, Einsatzbereiche, Trainingsmethoden und Finetuning-Techniken. Vorgestellt werden vortrainierte Transformer-LLMs, Text Clustering und Topic Modeling. Zudem geht es um Prompt Engineering, LangChain-Frameworks, Semantic Search sowie multimodale LLMs.

O'Reilly, 434 Seiten, Deutsch, ISBN 978-3-96009-266-7, 49,90 Euro

Thomas Sillmann

Das Swift-Handbuch

Vermittelt die Grundlagen der Swift-Programmierung und wie man Apps auf Basis von SwiftUI mit Xcode für unterschiedliche Plattformen entwickelt. Profis lernen, wie sie die Versionsverwaltung von Xcode nutzen. Mit aktuellen Technologien wie SwiftUI, SwiftData und Swift Testing.

Hanser-Fachbuch, 4. Auflage, 2025, 1037 Seiten, Deutsch, ISBN 978-3-446-48080-3, 59,99 Euro

Bernd Held

VBA mit Excel

Das umfassende Handbuch vermittelt Excel-VBA-Kenntnisse von den Grundlagen über Funktionen und Ereignisse bis hin zu User-Forms oder Symbolleisten. Das Buch bietet rund 650 Makros zum Download für Excel 2016 bis 2024 sowie Microsoft 365.

Rheinwerk, 6. Auflage, 2025, 1037 Seiten, Deutsch, ISBN 978-3-367-10573-1, 49,90 Euro

Highlights in der Ausgabe 1/26

Die nächste
Ausgabe
erscheint am
19.02.2026

Focus:

Data Crunching – wie aus Daten
Informationen werden

Weitere Themen:

- Smarte APIs mit .NET und KI bauen
- OWASP Top Ten 2025
- Agent AI Development mit AGA, MCP und Apache Kafka
- Testarchitektur
- Individuelle KI-Agenten mit Copilot Studio
- ... und vieles mehr

Foto: Shutterstock / LightSpectrum

developer-world.de



Inserentenverzeichnis

Ebner Media Group

www.ebnermedia.de
2, 93, 101, 129, 142–143, 147, 148

BenQ

www.benq.com
59

Impressum

dotnetpro by DWX

MEDIENHAUS

Ebner Media Group GmbH & Co. KG
Karlstraße 3
89073 Ulm

info@ebnermedia.de
www.ebnermedia.de

GESCHÄFTSFÜHRUNG

Annabel Ebner, Marco Parrillo

DE147041097
Sitz: Ulm, Amtsgericht Ulm, HRA 1900

Persönlich haftende Gesellschafterin der
Ebner Media Group GmbH & Co. KG
ist die Ebner Ulm MGV GmbH, Ulm.

Sitz: Ulm, Amtsgericht Ulm, HRB 576

OFFICE

Ebner Media Group GmbH & Co. KG
Office Köln
Helmholtzstraße 29–31
50825 Köln
Telefon: +49 731 88005–8000
www.dotnetpro.de

VERANTWORTUNG CONTENT

Chefredaktion

Fernando Schneider
fernando.schneider@ebnermedia.de

Redaktion

Tilman Börner
tilman.boerner@ebnermedia.de

Externe Autor*innen

Benjamin Abt, Gregor Biswanger, Elena Bochkor,
Alexander Brehm, Christian Giesswein, Tam Hanna,
Christian Havel, Markus Kraus, Dr. Veikko Krypczyk,
Bernhard Lauer, Hendrik Lösch, Stefan Papp,
Bernhard Pichler, Tobias Richling, Golo Roden,
Dr. Holger Schwichtenberg, Christian Sender,
Frank Simon, David Tielke, Helmut Vonhoegen,
Christian Wenz, Alexander Zachmann

Art Direction

Maria-Luise Steinkühler
malu.steinkuehler@ebnermedia.de

Gestaltung

EMG DESIGN UNIT
DesignUnit@ebnermedia.de

Publisher

Marco Parrillo
marco.parrillo@ebnermedia.de

MEDIA

Media Sales

Thomas Deck
thomas.deck@ebnermedia.de

Client Success (Disposition)

Anja Büttner
clientsuccess@ebnermedia.de

HERSTELLUNG, VERTRIEB & KUNDENSERVICE

Leitung Herstellung, Vertrieb & Kundenservice

Thomas Heydn
thomas.heydn@ebnermedia.de

Kundenservice

Telefon: +49 731 88005–8205
kundenservice@ebnermedia.de

Bezugspreise

Die jeweils gültigen Bezugspreise für unsere
Print- und Digitalausgaben sowie für eventuelle
Abonnements entnehmen Sie bitte den Angaben auf
unserer Website oder den jeweiligen Bestellseiten.
Alle Preise verstehen sich inklusive der gesetzlichen
Mehrwertsteuer und zuzüglich ggf. anfallender
Versandkosten, sofern nicht anders angegeben.

Preisänderungen bleiben vorbehalten. Bei
bestehenden Abonnements gelten die bei
Vertragsschluss vereinbarten Konditionen, sofern
keine anderweitige Regelung getroffen wurde.

Widerrufsrecht

Sie haben das Recht, binnen vierzehn Tagen ohne
Angabe von Gründen diesen Vertrag zu widerrufen.

Die Widerrufsfrist beträgt vierzehn Tage ab dem Tag,
an dem Sie oder ein von Ihnen benannter Dritter,
der nicht Beförderer ist, die erste Ware in Besitz
genommen haben.

Um Ihr Widerrufsrecht auszuüben, genügt
eine eindeutige Erklärung, mit der Sie uns über
Ihren Entschluss, diesen Vertrag zu widerrufen,
informieren. Bitte richten Sie diese an:

Ebner Media Group GmbH & Co. KG
Office München
Brienner Straße 45a–d
80333 München
Telefon: +49 731 88005–8205
E-Mail: hilfe@ebnermedia.de

Sie können dafür z. B. einen Brief oder eine E-Mail
verwenden. Zur Wahrung der Widerrufsfrist reicht es
aus, dass Sie die Mitteilung über die Ausübung des
Widerrufsrechts vor Ablauf der Frist absenden.

Urheberrecht / Copyright

Alle Inhalte, Werke und Publikationen – ob online
oder im Print – der Ebner Media Group GmbH & Co. KG
unterliegen dem deutschen Urheberrecht.

Jegliche Vervielfältigung, Bearbeitung, Verbreitung,
öffentliche Zugänglichmachung oder sonstige
Nutzung außerhalb der gesetzlichen Schranken des
Urheberrechts ist ohne ausdrückliche schriftliche
Genehmigung durch die Ebner Media Group GmbH
& Co. KG nicht gestattet. Downloads, Scans oder
Kopien sind ausschließlich für den privaten, nicht
kommerziellen Gebrauch erlaubt.

Soweit Inhalte nicht vom Verlag selbst erstellt
wurden, werden die Urheberrechte Dritter beachtet
und entsprechend gekennzeichnet.

DRUCK

Silber Druck GmbH & Co. KG
Otto-Hahn-Straße 25
34253 Lohfelden

.NET DEVELOPER CONFERENCE

Die Konferenz für .NET-Entwickler:innen

**SAVE
THE
DATE**

**23. BIS 26.
NOVEMBER 2026**
Pullman Cologne

Software Design | Software Architecture | Software Quality
Software Testing | Frontend Development | GenAI & Agentic
Data & Analytics | .NET | Approaches | Backend Technologies
Cloud & Infrastructure



DWX

DIE KONFERENZ FÜR AI,
CLOUD, WEB & .NET DEVELOPMENT

JETZT
SCHNELL
BLIND PRICE
SICHERN!

29. JUNI BIS
02. JULI 2026

M:CON CONGRESS CENTER
ROSENGARTEN | Mannheim



developer-world.de | #DWX