

GitHub Copilot

Tipps & Tricks Vol. 2

Best Practices für Entwickler*innen zum produktiveren Arbeiten mit GitHub Copilot



Vorwort

In meiner Zeit bei Microsoft Deutschland durfte ich schon einige E-Books begleiten. Das erste E-Book – wenn ich mich richtig erinnere – war das Visual Studio Inselbuch 2013 mit Beiträgen zu Visual Studio, Team Foundation Server, Windows 8 und Windows Azure (wie es damals noch hieß). Danach folgten weitere E-Books zu Visual Studio, Visual Studio Code oder auch zur effizienten Remote-Zusammenarbeit von Entwicklungsteams kurz nach dem Ausbruch von Covid. Für letzteres E-Book hat damals noch Thomas Dohmke einen Beitrag über den Kulturwandel bei GitHub geschrieben. Inzwischen ist er CEO von GitHub und der nächste Wandel ist bereits in vollem Gange – die Rede ist hier natürlich von GitHub Copilot.

Bereits im Jahr 2015 hat Satya Nadella, CEO von Microsoft, gesagt, dass jedes Unternehmen ein Software-Unternehmen sein wird. In ähnlicher Form hat Satya Nadella die Aussage bis heute wiederholt. Dabei geht es im Kern immer darum, Entwicklungsteams oder auch sogenannte Citizen Developer zu befähigen, ihre Kreativität bestmöglich einsetzen zu können, um Probleme zu lösen sowie neue Produkte und Lösungen zu schaffen, die einen Mehrwert darstellen. Im Vordergrund steht also zum einen die Befähigung, mit Technologie Neues zu erschaffen, und zum anderen die Konzentration auf das Wesentliche.

Und genau hier setzt GitHub Copilot an. Um Ihnen dieses neue Angebot aus der Perspektive von Nutzer*innen näher zu bringen, haben wir uns entschlossen, mal wieder ein E-Book zu veröffentlichen mit Tipps und Tricks zu GitHub Copilot aus dem echten Leben. Die Idee ist, das E-Book nach und nach mit neuen Tipps und Tricks zu erweitern. Falls Sie selber einen Beitrag in einer der nächsten Ausgaben beisteuern möchten, dann schreiben Sie mir gerne eine E-Mail an techwiese@microsoft.com. Ich würde mich sehr freuen.

Bis dahin wünsche ich Ihnen viel Spaß bei der Lektüre der ersten Ausgabe dieses E-Books.

Ihr Dennis Gassen

Malte Lantin

Als Principal Solutions Engineer bei GitHub unterstützt Malte Lantin strategische Kunden in Europa bei der Einführung von GitHub Enterprise als zentrale Plattform für die sichere Entwicklung und Bereitstellung von Software.



Einleitung: Die Zukunft der Softwareentwicklung mit GitHub Copilot

Von Malte Lantin

Nur knapp 3 Jahre nach dem ersten internen GitHub Whitepaper zu KI-gestützter Softwareentwicklung im Jahr 2020 und der Veröffentlichung von GitHub Copilot im Jahr 2022 hat sich GitHub Copilot heute als Standard für viele Entwickler*innen und Unternehmen etabliert. Dabei gerät fast in Vergessenheit, dass KI-Unterstützung in der Softwareentwicklung noch vor kurzem von vielen als Fiktion wahrgenommen wurde. GitHub Copilot stellt somit einen Wendepunkt in der Entwicklung von Softwareentwicklungstools dar. Mit kontinuierlichen Verbesserungen und speziell für den professionellen Einsatz in Unternehmen entwickelten Funktionen ist GitHub Copilot heute bereits ein kaum mehr wegzudenkendes Standardwerkzeug. Diese Einführung zeigt die grundlegende Funktionsweise und die Mehrwerte, die in den folgenden Kapiteln anhand zahlreicher Fallbeispiele veranschaulicht werden.

Technologische Innovationen und Funktionen

GitHub Copilot basiert auf mehrjähriger Forschung im Bereich Sprachmodelle, deren sichere Bereitstellung in den Azure OpenAI Services sowie der nahtlosen Integration in den Entwicklungsprozess. Entwickler*innen steht damit heute in Visual Studio Code, Visual Studio, den JetBrains IDEs sowie weiteren Editoren eine KI-basierte Codevervollständigung zur Verfügung, die ihren Kontext aus den aktuell in Bearbeitung befindlichen Quellcodedateien sowie Informationen aus dem jeweiligen Editor bezieht. Dadurch kann sie im laufenden Entwicklungsprozess passende und hochrelevante Codevorschläge generieren. Für diese Codevervollständigung wird das Fill-in-the-Middle (FIM) Paradigma verwendet, das es ermöglicht, jederzeit Vorschläge zu liefern, die bestmöglich an den aktuellen Kontext angepasst sind und den Stil des jeweiligen Projektes berücksichtigen.

Ende 2023 wurden mit der Einführung von GitHub Copilot Chat die Möglichkeiten der KI-gestützten Entwicklung nochmals deutlich erweitert. Über die in die Entwicklungsumgebung integrierte Chatoberfläche ist es nun unter anderem möglich, komplexere Anweisungen zu geben, weitergehende Empfehlungen zu erhalten, Codeerklärungen abzurufen, Tests generieren oder ganze Arbeitsumgebungen von GitHub Copilot erstellen zu lassen. Durch den iterativen Ansatz basierend auf natürlicher Sprache ist Copilot Chat leicht zugänglich und hilft allen Entwickler*innen schneller Software zu entwickeln, Probleme zu beheben und neue Technologien zu erlernen. Dabei ist das Kontextwissen, das GitHub Copilot aus dem Code-Editor gewinnt, ein wesentliches Unterscheidungsmerkmal zu anderen Chat-basierten Sprach-KIs. GitHub Copilot Chat ist dazu in der Lage, auf das Wissen von angeforderten Dateien oder Aussagen aus der Kommandozeile zurückzugreifen und auf diese Weise jederzeit passgenaue Vorschläge zu liefern.

GitHub Copilot wird zudem kontinuierlich erweitert und steht ab Frühjahr 2024 auch als Erweiterung der GitHub CLI Tools zur Verfügung, um Befehle auf der Kommandozeile zu erklären oder zu generieren. Seit Februar 2024 haben Unternehmen, welche die GitHub Enterprise Cloud verwenden, außerdem die Möglichkeit mit GitHub Copilot Enterprise von einer noch tieferen Integration von GitHub Copilot in die eigene Entwicklung zu profitieren. Dies erfolgt über eine Integration von GitHub Copilot Chat mit der GitHub-Weboberfläche, eine KI-gestützte Bearbeitung von Pull-Request, Zugriff auf das Wissen der eigenen Quellcode-Repositories und Dokumentation, sowie einer zukünftigen Anpassung der KI-Modelle an die eigene Codebasis. Hiermit werden die Teile des Entwicklungsprozesses, welche sich außerhalb der Entwicklungsumgebung befinden, sinnvoll durch GitHub Copilot unterstützt.

Auswirkungen auf die Produktivität und Codequalität

Mit über drei Milliarden generierten Codezeilen hat GitHub Copilot seine Wirksamkeit und weitreichende Akzeptanz in der Entwicklergemeinschaft bereits eindrücklich bewiesen. Heute benutzen mehr als 1,8 Millionen Softwareentwickler*innen GitHub Copilot und mehr als 50.000 Unternehmen haben sich für eine Einführung der Technologie entschieden. Der Einfluss von GitHub Copilot auf die Produktivität der Entwickler*innen ist sowohl quantifizierbar als auch erheblich. In einer Studie, wurde festgestellt, dass mehr als 30 Prozent der Vorschläge von GitHub Copilot akzeptiert werden, was die Relevanz und Nützlichkeit in der täglichen Entwicklung unterstreicht. Mit einer Beschleunigung der Arbeit um bis zu 55 Prozent ist der Mehrwert offensichtlich. Beinahe wichtiger ist jedoch die Tatsache das Entwickler*innen sich bei einem Einsatz von GitHub Copilot auch in Befragungen als zufriedener und produktiver beschreiben. Zudem zeigen Studien, dass die Codequalität steigt und Code Reviews als zielführender und schneller beschrieben werden. Entwickler*innen berichten, dass sie mit GitHub

Copilot länger konzentriert arbeiten können, da Kontextwechsel und Ablenkung vermieden werden. Die deutlich verbesserte Entwicklungserfahrung bietet Vorteile über Produktivitätssteigerungen hinaus. Indem GitHub Copilot routinemäßige und repetitive Aspekte automatisiert, wird es den Entwickler*innen ermöglicht, sich auf komplexere und kreativere Aspekte ihrer Projekte zu konzentrieren, was die Arbeitszufriedenheit erhöht.

Fazit

Zusammenfassend stellt GitHub Copilot einen bedeutenden Meilenstein in der Geschichte der Softwareentwicklung dar. Es symbolisiert den Beginn einer neuen Ära der KI-unterstützten Programmierung, die die Produktivität der Entwickler*innen erheblich verbessert, die Codequalität erhöht und das allgemeine Wohlbefinden der Entwickler*innen positiv beeinflusst. Da GitHub Copilot kontinuierlich weiterentwickelt wird und sich in Zukunft noch stärker in den Softwareentwicklungsprozess integriert, wird sein Einfluss weiter wachsen und GitHub Copilot damit ein unverzichtbares Werkzeug im Softwareentwicklungsbereich werden.

In den folgenden Kapiteln werden Einsatzmöglichkeiten beschrieben und damit der praktische Nutzen im Alltag verdeutlicht. Wir wünschen viel Spaß beim Lesen und dem Einsatz von GitHub Copilot.

Weiterführende Informationen:

- [Copilot transforms GitHub into the AI-powered developer platform](#)
- [The economic impact of the AI-powered developer lifecycle and lessons from GitHub Copilot](#)
- ¹ [Quantifying GitHub Copilot's impact on developer productivity and happiness](#)
- ² [Quantifying GitHub Copilot's impact on code quality](#)

Inhalt

Vorwort	2
Einleitung: Die Zukunft der Softwareentwicklung mit GitHub Copilot	3
GitHub Copilot: Versionen und Features im Überblick	7
GitHub Copilot im Einsatz	11
App-Migration zwischen Programmiersprachen mit GitHub Copilot Chat	13
Nie mehr Lorem Ipsum	20
Intelligentes Coding mit GitHub Copilot	24
Von der Idee zum Deployment in unter 30 Minuten	30
Tests schreiben mit GitHub Copilot Chat	34
Proaktive Fehlerbehebung auf Windows-Geräten	36
Wie war der Code doch gleich?	40
Automatisierung von nervigen Entwicklungsarbeiten mit GitHub Copilot Chat	42
Bessere Prompts für besseren Code – Tipps & Tricks für Developer	47
Aussagekräftige Funktionsnamen & Variablen	50
Unterstützung bei der Testfallerstellung mit GitHub Copilot	52
Wie wir drei Monate die Möglichkeiten von GitHub Copilot getestet haben	55
GitHub Copilot als KI-Assistent für barrierefreie Software	60
Von der Anforderung zur Komponente	70
Einsatz einer Chat-Syntax: Chat-Teilnehmer, Chat-Variablen, Slash-Befehle	76
Nutzen Sie GitHub Copilot für Ihre Projekte	81
Weiterführende Ressourcen	82

GitHub Copilot: Versionen und Features im Überblick

GitHub Copilot bringt künstliche Intelligenz in die Entwicklung und hilft Entwickler*innen dabei, schneller besseren Code zu entwickeln. Der KI-Assistent unterstützt beim Schreiben einzelner Codezeilen oder ganzer Codeblöcke und kann Vorschläge unterbreiten, wie bereits vorhandener Code erweitert oder verbessert werden kann. Das Resultat ist eine um mehr als die Hälfte beschleunigte Entwicklung, da GitHub Copilot die Arbeitsabläufe von Entwickler*innen deutlich verbessert.

Sie schreiben mit GitHub Copilot zum Beispiel einen Kommentar, der die gewünschte Logik beschreibt und die von Ihnen bevorzugten Stilkonventionen festlegt. Den Rest erledigt die KI, sodass Sie sich auf die wirklich komplexen Aufgaben konzentrieren können. So können Sie produktiver arbeiten und sich wiederholende Aufgaben viel schneller und stressfreier erledigen. Gleichzeitig hilft GitHub Copilot Entwickler*innen dabei, eine neue Sprache oder ein neues Framework schneller zu beherrschen, als dies durch das Lesen von Dokumentation oder durch Webrecherche möglich wäre.

Unterstützung für zahlreiche Programmiersprachen und die Entwicklung von sicherem Code

GitHub Copilot unterstützt eine Vielzahl von Programmiersprachen. Dazu gehören unter anderem C#, C++, Python, JavaScript, TypeScript, Go und Ruby. Mehr als 50.000 Unternehmen setzen den KI-Assistenten bereits ein, darunter ein Drittel aller Fortune-500-Unternehmen.

GitHub Copilot verbessert auch die Zusammenarbeit im Team. Das System indiziert Ihre Repositories, versteht den darin gespeicherten Code und hilft Ihnen, sich schneller in neue Codebasen einzuarbeiten. Wenn Sie neue Repositories anbinden, können Sie auf der Arbeit anderer aufbauen.

Ein weiteres Einsatzgebiet von GitHub Copilot ist das Erstellen von Funktionen und Diagrammen, die Entwickler*innen über die entsprechende Eingabe als Prompt vorgeben. GitHub Copilot vermeidet

Schwachstellen und Sicherheitslücken bereits bei der Codegenerierung, da das System darauf optimiert ist, möglichst sicheren Code zu generieren. Unsichere Codezeilen werden von der KI blockiert.

KI-basierte Vorschläge in Echtzeit: Mehr Sicherheit, weniger Fehler

Die Stärke von GitHub Copilot liegt vor allem darin, KI-basierte Vorschläge für die Codeentwicklung in Echtzeit zu liefern. Während Sie selbst Codezeilen schreiben, analysiert GitHub Copilot den Code und macht automatisch Vorschläge zur Vervollständigung. Dadurch entsteht eine perfekte Harmonie zwischen Entwickler*in und KI. Entwickler*innen können mit GitHub Copilot auch automatisiert Commit-Nachrichten erstellen. Es gibt sehr viele Einsatzgebiete, die Entwickler*innen das tägliche Leben deutlich erleichtern können.

GitHub Copilot hilft nicht nur dabei, neuen Code zu erstellen oder bestehende Codezeilen zu verbessern. Der KI-Assistent ist auch ein wertvolles Werkzeug, um Fehler im Code zu finden. Grundlage dafür ist eine „Konversation“ in natürlicher Sprache. Wenn Sie irgendwo nicht weiterkommen, fragen Sie einfach GitHub Copilot. GitHub Copilot agiert intelligent, denn die KI personalisiert die Antworten auf Basis Ihres Fachwissens und der vorhandenen Dokumentation, aus der die KI-Modelle weiterlernen.

Sie müssen für die Verwendung nicht Ihre bevorzugte Entwicklungsumgebung verlassen. Sie können GitHub Copilot über Erweiterungen direkt in Ihre Entwicklungsumgebung integrieren, zum Beispiel in Visual Studio, Visual Studio Code, Vim, Neovim, JetBrains Suite und Azure Data Studio. GitHub Copilot kann auch in der CLI helfen, um Hilfe zu erhalten. GitHub Copilot ist seit 2024 auch für GitHub Mobile verfügbar, sodass Entwickler*innen über das Smartphone und Tablet genauso mit GitHub Copilot arbeiten können wie auf einem Desktop-Rechner.

GitHub Copilot: Passende Versionen für alle Anwendungsbereiche

Für jedes Einsatzgebiet gibt es die passende Version von GitHub Copilot. Einzelne Entwickler*innen profitieren dadurch genauso von den Möglichkeiten wie kleine und mittlere Unternehmen oder große Organisationen. Letztere entscheiden sich entweder für GitHub Copilot Business oder GitHub Copilot Enterprise. Für Einzelentwickler*innen gibt es GitHub Copilot Individual. Alle Versionen haben eines gemeinsam: Sie wandeln natürliche Sprache in Code um, und zwar direkt in der IDE.

GitHub Copilot Business richtet sich in erster Linie an Unternehmen. Für 19 US-Dollar pro Benutzer*in erhalten die Entwickler*innen im Unternehmen Zugriff auf GitHub Copilot. Dazu gehören Code-Ergänzungen, Chat mit GitHub Copilot über die IDE und GitHub Mobile, CLI-Unterstützung, Filter für

Sicherheitslücken und öffentlichen Code, Sicherheit und Datenschutz auf Unternehmensebene und Code-Referenzierung. Wenn GitHub Copilot anfällige Codierungsmuster erkennt, blockiert es die Zeilen. Dies gilt auch für die Verwendung öffentlichen Codes.

GitHub Copilot Enterprise bietet ergänzend zur Business-Version weitere Funktionen, die den Mehrwert für Unternehmen erheblich steigern. Dazu gehört der erweiterte Chat, der speziell auf die Repositories des jeweiligen Unternehmens zugeschnittene Gespräche ermöglicht. Zudem wird die Pull-Request-Diff-Analyse unterstützt, die eine präzisere Code-Überprüfung ermöglicht. Bei der Codevervollständigung profitieren Nutzer von speziell abgestimmten Modellen, um die Effizienz und Präzision der Codeerstellung weiter zu erhöhen. GitHub Copilot Enterprise kann neben IDE, CLI und GitHub Mobile auch auf GitHub.com genutzt werden.

GitHub Copilot Individual richtet sich an einzelne Entwickler*innen, die Code vervollständigen wollen. Auch hier steht ein Chat mit GitHub Copilot zur Verfügung. Diese Version ist vor allem für individuelle Entwickler*innen und Freiberufler*innen geeignet. Diese Version steht außer Student*innen auch Lehrer*innen und Betreuer*innen zur Verfügung.

Copilot kopiert keinen Code, sondern erstellt ihn neu

Generell kopiert keine der GitHub Copilot-Versionen den Code direkt, sondern generiert ihn auf Basis von Wahrscheinlichkeitsberechnungen neu. Obwohl die von GitHub Copilot verwendeten Modelle mit Code aus öffentlich zugänglichen Repositories trainiert werden, enthalten sie selbst keinen Code. Bei der Codegenerierung nutzt GitHub Copilot den bereits vorhandenen Code in der IDE, insbesondere die Codezeilen oberhalb und unterhalb des Cursors. Hinzu kommen Informationen aus den Trainingsdaten und anderen Dateien, die Sie mit GitHub Copilot und dem jeweiligen Repository verknüpft haben. Auf Basis dieser Daten generiert GitHub Copilot die nächsten Codezeilen oder macht Vorschläge dafür.

Ein Blick in die Zukunft von GitHub Copilot

Der Funktionsumfang von GitHub Copilot wird kontinuierlich erweitert, sodass Sie sich regelmäßig auf neue innovative Möglichkeiten freuen können. So wurden auf der Microsoft Build 2024 die sog. GitHub Copilot Extensions vorgestellt, über die sich Partnertechnologien direkt mit GitHub Copilot Chat integrieren lassen. Zu den ersten Partnern gehören unter anderem DataStax, Docker, LambdaTest, Microsoft Azure, MongoDB, Octopus Deploy, Sentry und Stripe.

Ein mögliches Szenario für den Einsatz der Erweiterungen: Sie müssen für Ihr Entwicklungsprojekt einen Datenbankfehler beheben. Hierfür können Sie direkt über GitHub Copilot Chat verschiedene Tools und Technologien wie Sentry oder Azure nutzen, um das Problem zu analysieren und zu beheben – ohne dabei den Kontext wechseln zu müssen.

Das ist nur eines von vielen Beispielen, wie GitHub Copilot weiterentwickelt wird. Aktuelle Neuerungen werden immer über den GitHub Blog vorgestellt.



Martin Brandl

Martin (@martin_jib) arbeitet als CTO und Geschäftsführer bei der white duck GmbH. Er verfügt über mehr als ein Jahrzehnt Erfahrung mit Microsoft Azure, seine Leidenschaft gilt der Entwicklung von Cloud-nativen Anwendungen. Aufgrund seiner Beiträge zur Azure-Community wurde er bereits mehrfach als Microsoft MVP für Azure ausgezeichnet.

Tipp 1:

GitHub Copilot im Einsatz

Slash Commands in GitHub Copilot

von Martin Brandl

Mit den **Slash Commands** von GitHub Copilot wird die Interaktion mit dem Assistenten noch effizienter. Durch einfaches Tippen eines Slash ("/") im Chat öffnet sich eine Liste von nützlichen Befehlen, die es Ihnen ermöglichen, verschiedene Aktionen direkt auszuführen, ohne dabei viel zu tippen.

Die Befehle sind kontextbezogen und beziehen sich immer auf einen sogenannten Agenten.

Sie werden mit dem @-Symbol angegeben. Derzeit existieren davon zwei:


- **@workspace** verfügt über Kontextinformationen zum Code in Ihrem Arbeitsbereich und kann Ihnen bei der Navigation helfen, indem er relevante Dateien oder Klassen findet.
- **@vscode** kennt Befehle und Funktionen im VS Code-Editor selbst und kann Ihnen dabei helfen, sie effektiv zu nutzen.


Beispiel: Im folgenden Code-Abschnitt wird eine Ganzzahldivision durchgeführt, bei der die Nachkommastellen abgeschnitten werden. Das Ergebnis ist 2.0, anstatt der erwarteten 2.5:

```
int x = 5;
int y = 2;
double result = x / y;
```

Der Fehler lässt sich jedoch einfach beheben, in dem Sie in GitHub Copilot Chat den Befehl /fix ausführen. Als Agent wird dabei automatisch @workspace mit angegeben:

Ask Copilot or type '/' for commands ▶ 🗨 ✕

 mjisaak
`@workspace /fix`

 **GitHub Copilot** used @workspace 👍 🗨

Used 1 reference >

The issue with your code is that you are performing integer division, which will not yield a fractional result. To fix this, you should convert `x` or `y` to `double` before performing the division. Here's the corrected code:

```
int x = 5;
int y = 2;
double result = (double)x / y;
```

This will ensure that `result` holds the correct decimal value of the division.

Neben dem **/fix**-Kommando, stehen Ihnen noch eine Reihe weiterer nützlicher Befehle zur Verfügung, die Sie mit **/help** aufrufen können:

- **@workspace /explain**: Mit diesem Befehl wird der ausgewählte Code Schritt für Schritt erläutert.
- **@workspace /fix**: Dieser Befehl schlägt eine Lösung für etwaige Fehler im ausgewählten Code vor.
- **@workspace /new**: Mit diesem Befehl können Sie ein neues Projekt anhand einer natürlichen Sprachbeschreibung erstellen.
- **@workspace /newNotebook**: Dieser Befehl ermöglicht die Erstellung eines neuen Jupyter Notebooks basierend auf Ihrer Beschreibung.
- **@workspace /terminal**: Hier erhalten Sie Erklärungen zur Nutzung des integrierten Terminals für verschiedene Aufgaben.
- **@workspace /tests**: Dieser Befehl generiert automatisch Unit-Tests für den ausgewählten Code.
- **@vscode /api**: Falls Sie Fragen zur Entwicklung von VS Code-Erweiterungen haben, steht Ihnen dieser Befehl zur Verfügung.



David Losert

David ist Principal Solutions Engineer bei GitHub, wo er Enterprise Kunden mit seiner über 10-jährigen Erfahrung als Developer und Cloud-Consultant bei der optimalen Etablierung von GitHub als AI-getriebene DevSecOps-Plattform unterstützt.

Tipp 2:

App-Migration zwischen Programmiersprachen mit GitHub Copilot Chat

von David Losert

Von Standardisierungs- oder Modernisierungsvorhaben über fehlende Fachexpertise der Entwickler*innen bis hin zum Wunsch nach unternehmensweiter Kollaboration über Teamgrenzen hinweg (Stichwort: Innersourcing): Es kann viele gute Gründe geben, eine bestehende Anwendung oder Teile davon von einer Programmiersprache in eine andere zu übersetzen.

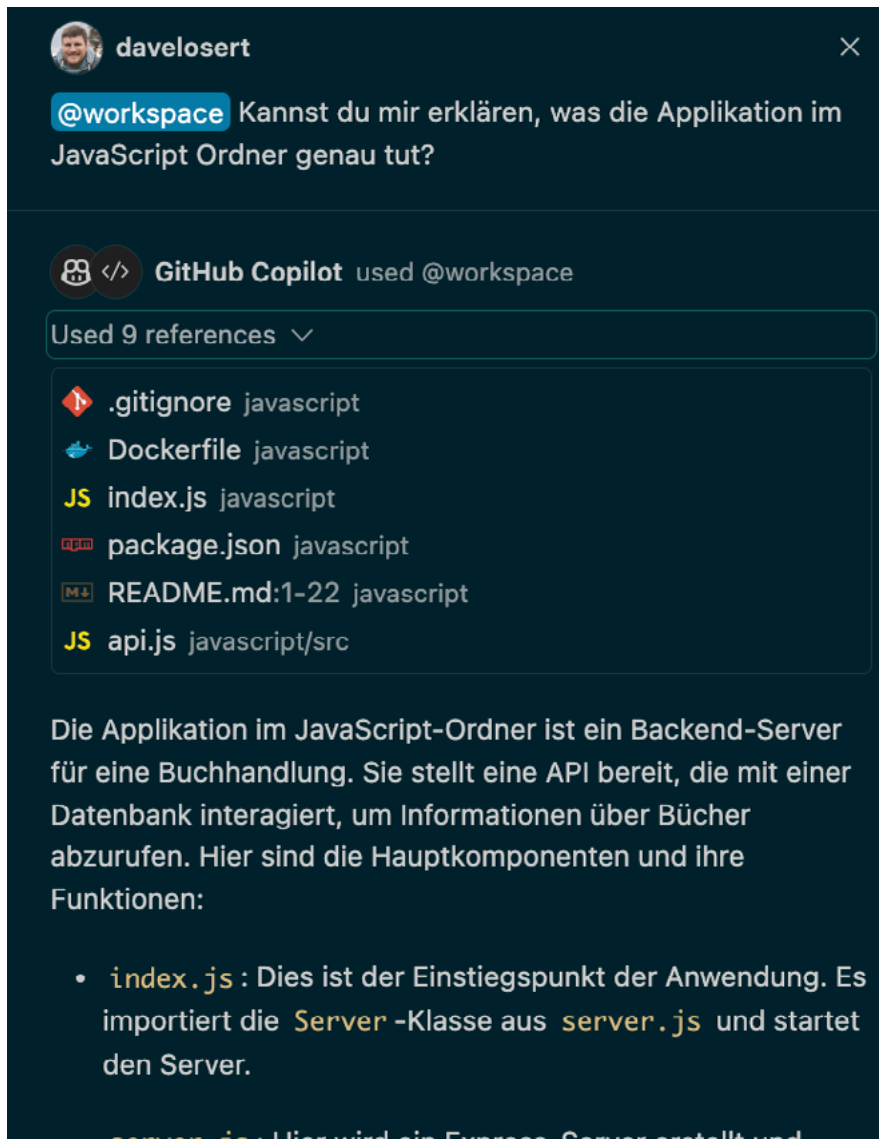
Gerade in Unternehmen erleben wir nach gut einem Jahrzehnt der technologischen Diversifizierung – nicht zuletzt getrieben durch Microservices, Containerisierung und der Einfachheit des technologieagnostischen Betriebs in der Public Cloud – nun wieder eine Rückwärtsbewegung und den Wunsch nach mehr Standardisierung und Vereinheitlichung. Glücklicherweise ist die Übersetzung oder Migration zwischen zwei Programmiersprachen dank der Fortschritte der künstlichen Intelligenz in der Softwareentwicklung, allen voran GitHub Copilot, viel einfacher geworden. Im Folgenden sehen Sie, wie das geht.

Die Vorbereitung

Bevor Sie mit der Übersetzung beginnen können, müssen Sie die Zieltechnologie und die Zielarchitektur definieren, denn auch mit GitHub Copilot muss das Ziel der Reise selbst bestimmt und der Weg dorthin gut geplant werden.

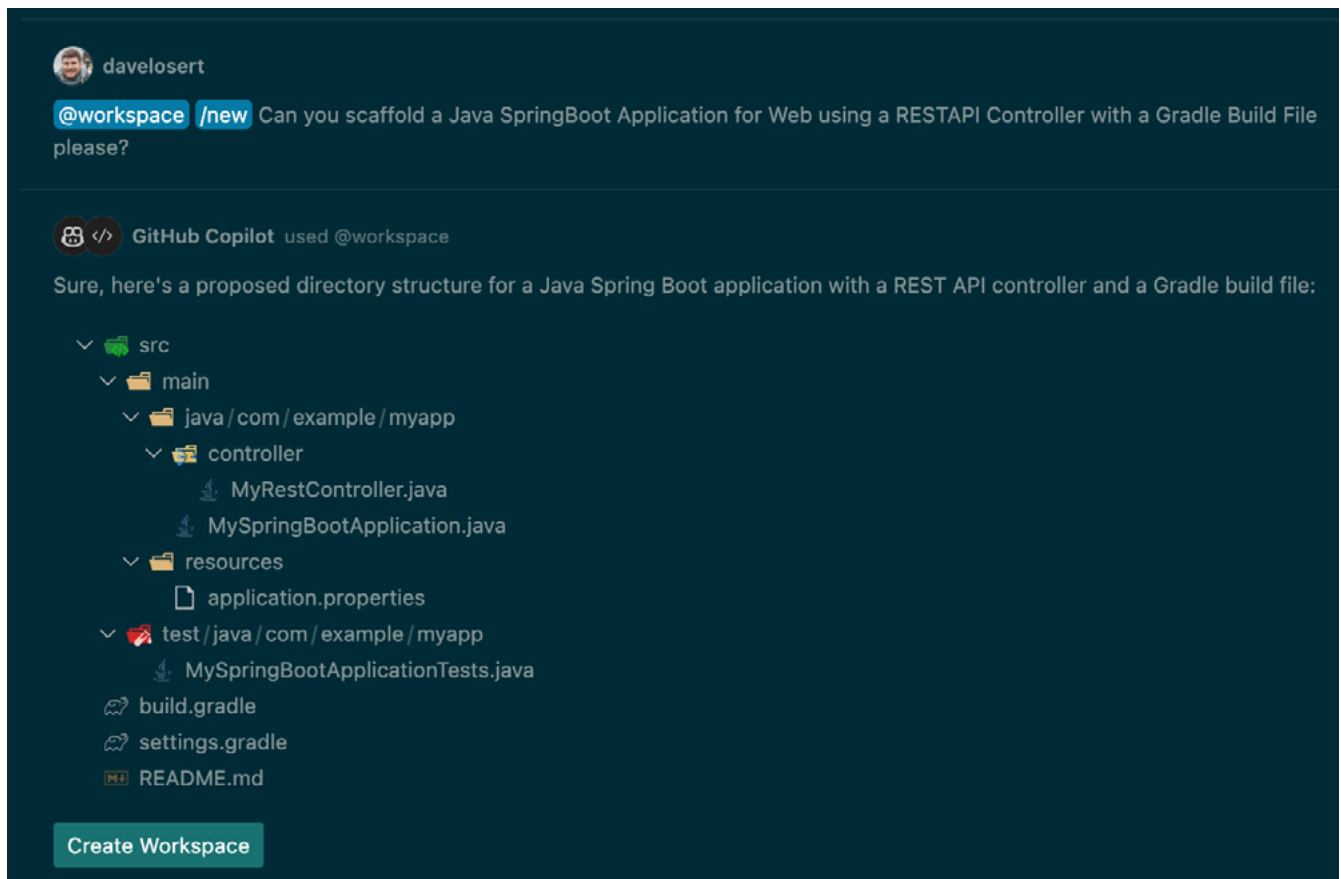
Ebenso sollte man sich ein Grundverständnis der Quellanwendung aneignen, um nicht blind zu migrieren. Hier kann GitHub Copilot Chat mit dem sogenannten @workspace-Agenten bereits sehr hilfreich sein.

Dieser bindet auch nicht geöffnete Dateien in den von GitHub Copilot verarbeiteten Kontext ein. So profitieren Sie von Erklärungen und einem besseren Verständnis einer größeren Anzahl von Dateien.



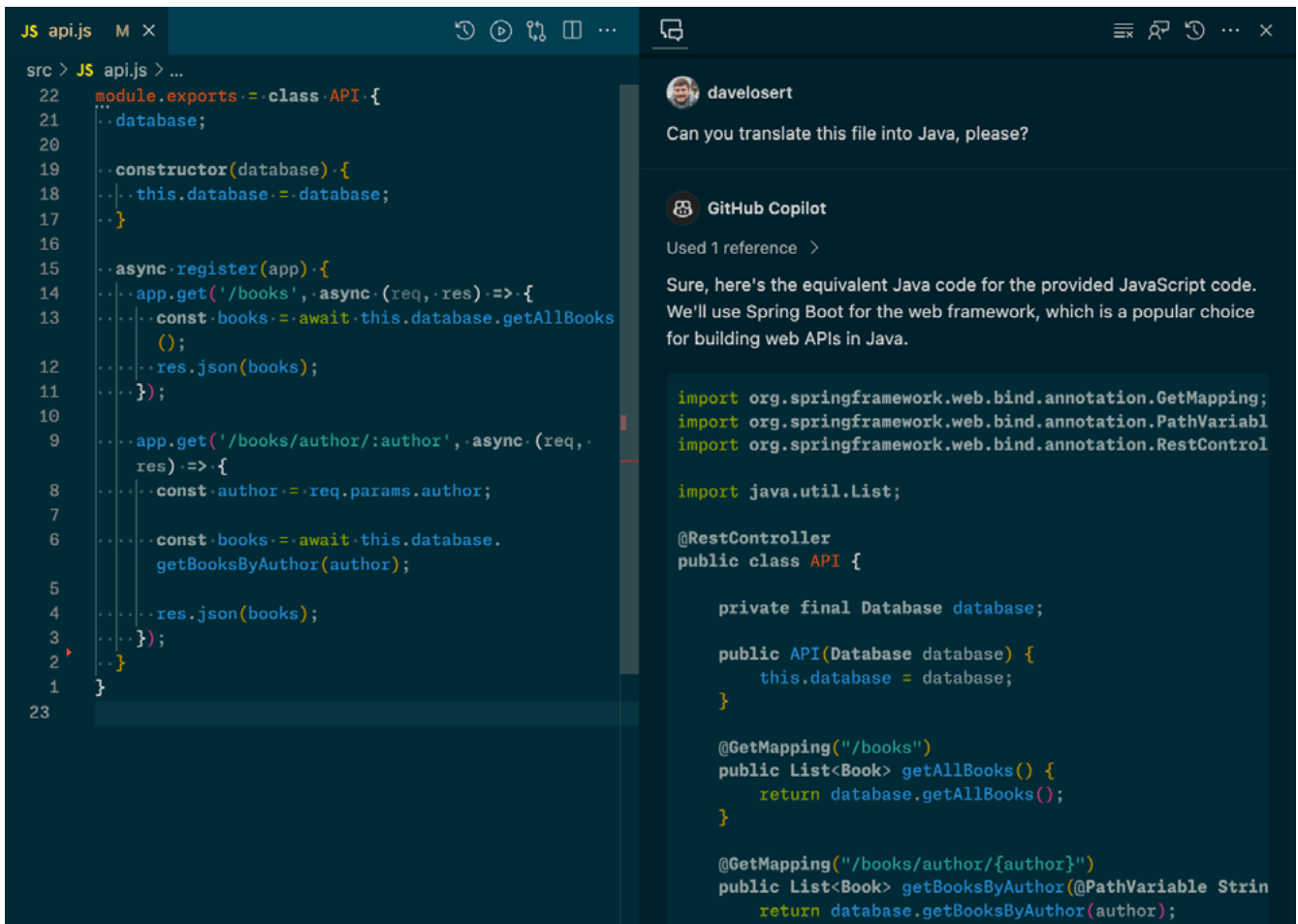
Wie der Screenshot zeigt, kann man mit GitHub Copilot auch auf Deutsch kommunizieren. Da der Code in der Regel in Englisch definiert ist und damit das Ganze ein stimmiges Bild ergibt, wurden die folgenden Screenshots mit englischen Texten erstellt. Der Ziel-Stack kann auch mit GitHub Copilot Chat und dem Schrägstrichbefehl `/new` initialisiert werden.

Die Migration

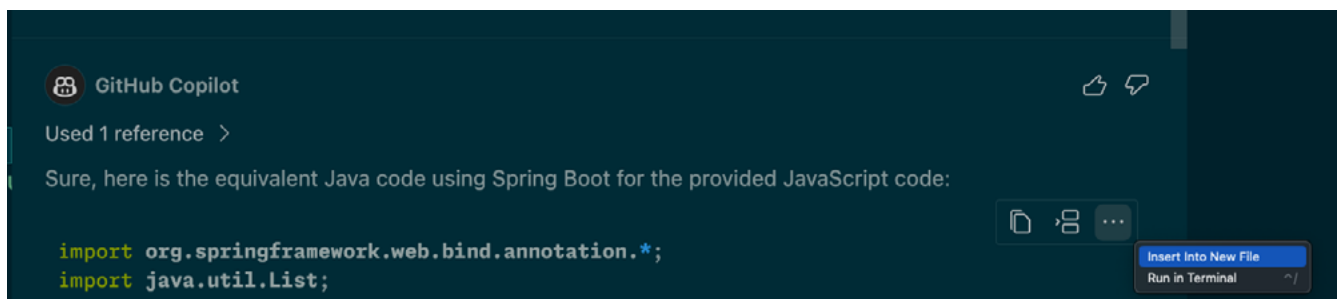


Die eigentliche Migration findet dann durch die Entwickler*innen direkt in der IDE statt und ist im Grunde sehr einfach:

1. Sie öffnen eine zu übersetzende Datei
2. Im Chatfenster von GitHub Copilot fordern Sie die Übersetzung an: „Translate this file into <Target-Programming-Language>, please!“ (es kann nicht schaden, freundlich zur AI zu sein 😊)



3. Kopieren Sie den generierten Code mit Copy & Paste oder fügen Sie ihn mit der Option „Insert into new File“ in das Ziel ein.

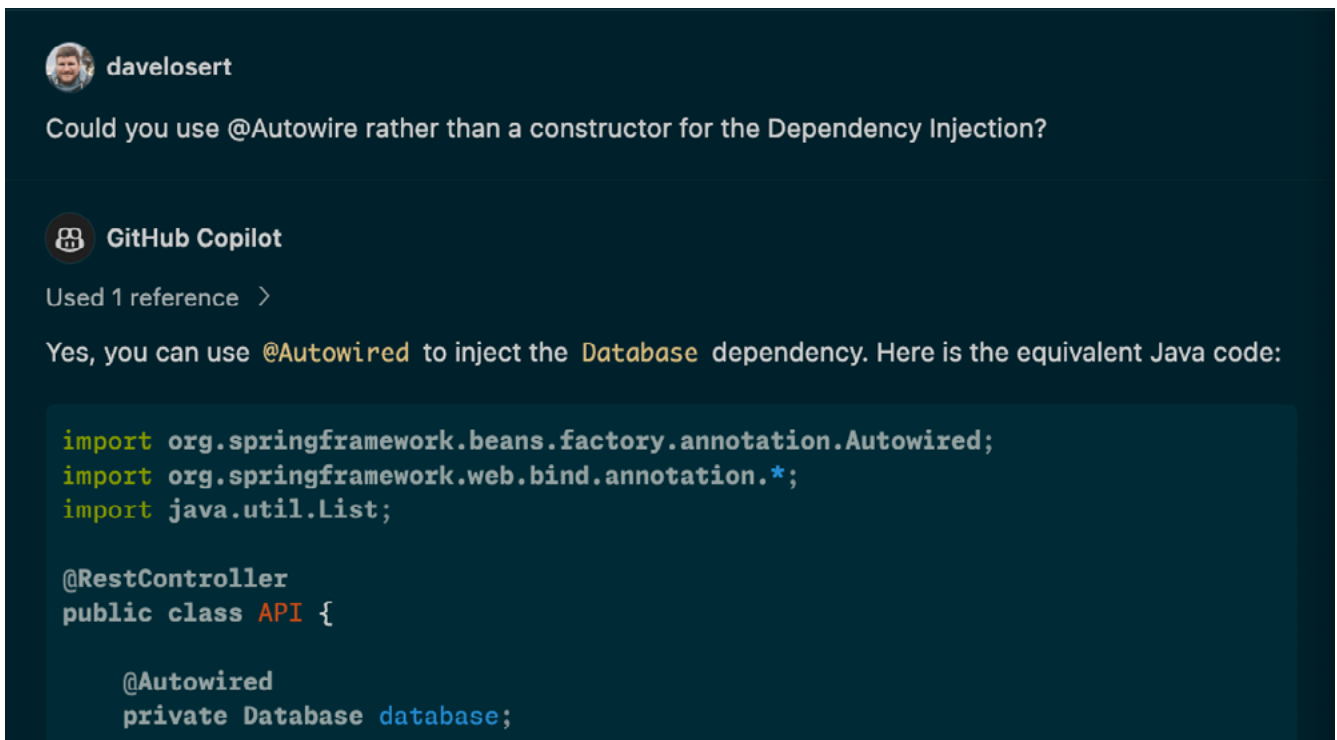


Wiederholen Sie diesen Vorgang für alle Dateien im Quellsystem, um die Migration erfolgreich abzuschließen. Meistens ist es aber nicht ganz so einfach und es gibt noch einige Dinge zu beachten. Auf diese gehe ich im Folgenden näher ein.

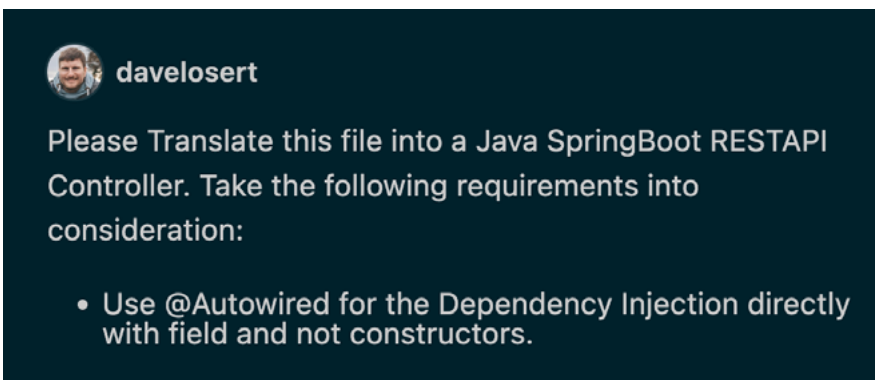
Review & Refinements

Nach der Übersetzung einzelner Dateien sind oft noch kleinere oder größere Verbesserungen an der Zieldatei notwendig. GitHub Copilot hilft in der Regel bei 80 bis 90 % des Grundgerüsts, aber gerade am Anfang ist der Zielcode oft noch nicht perfekt.

Im obigen Beispiel gibt es zum Beispiel in Spring-Boot mit der @Autowired-Annotation einen schöneren Weg, die Abhängigkeiten in die Klassen zu injizieren. Verwenden Sie in diesem Fall also den Chat, um den Zielcode zu verbessern:

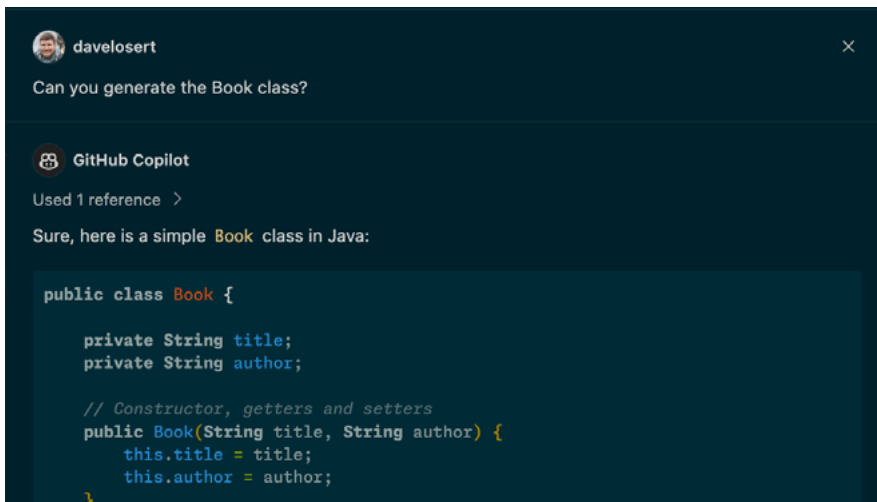


Um diese Verfeinerung nicht bei jeder neuen Übersetzung erneut durchführen zu müssen, können Sie den Prompt auch direkt bei der nächsten zu übersetzenden Datei verfeinern:



So erhalten Sie im Laufe der Migration immer bessere Übersetzungen von GitHub Copilot ab dem ersten Prompt. *Kleiner Tipp:* Mit „Pfeil nach oben“ können Sie wie in einem Terminal die vorherigen Prompts zur Verbesserung in das Eingabefeld holen.

Bei konzeptionellen Unterschieden der Programmiersprachen ist oft weitere Arbeit nötig. Wenn Sie zum Beispiel wie hier von typenlosem JavaScript nach Java übersetzen, müssen Sie für Datenstrukturen, die in JS inline definiert sind, noch eine Datenklasse in Java anlegen. Auch hierfür können Sie GitHub Copilot Chat verwenden, um sich vor allem bei größeren Datenklassen viel Tipparbeit zu ersparen:



```
public class Book {  
  
    private String title;  
    private String author;  
  
    // Constructor, getters and setters  
    public Book(String title, String author) {  
        this.title = title;  
        this.author = author;  
    }  
}
```

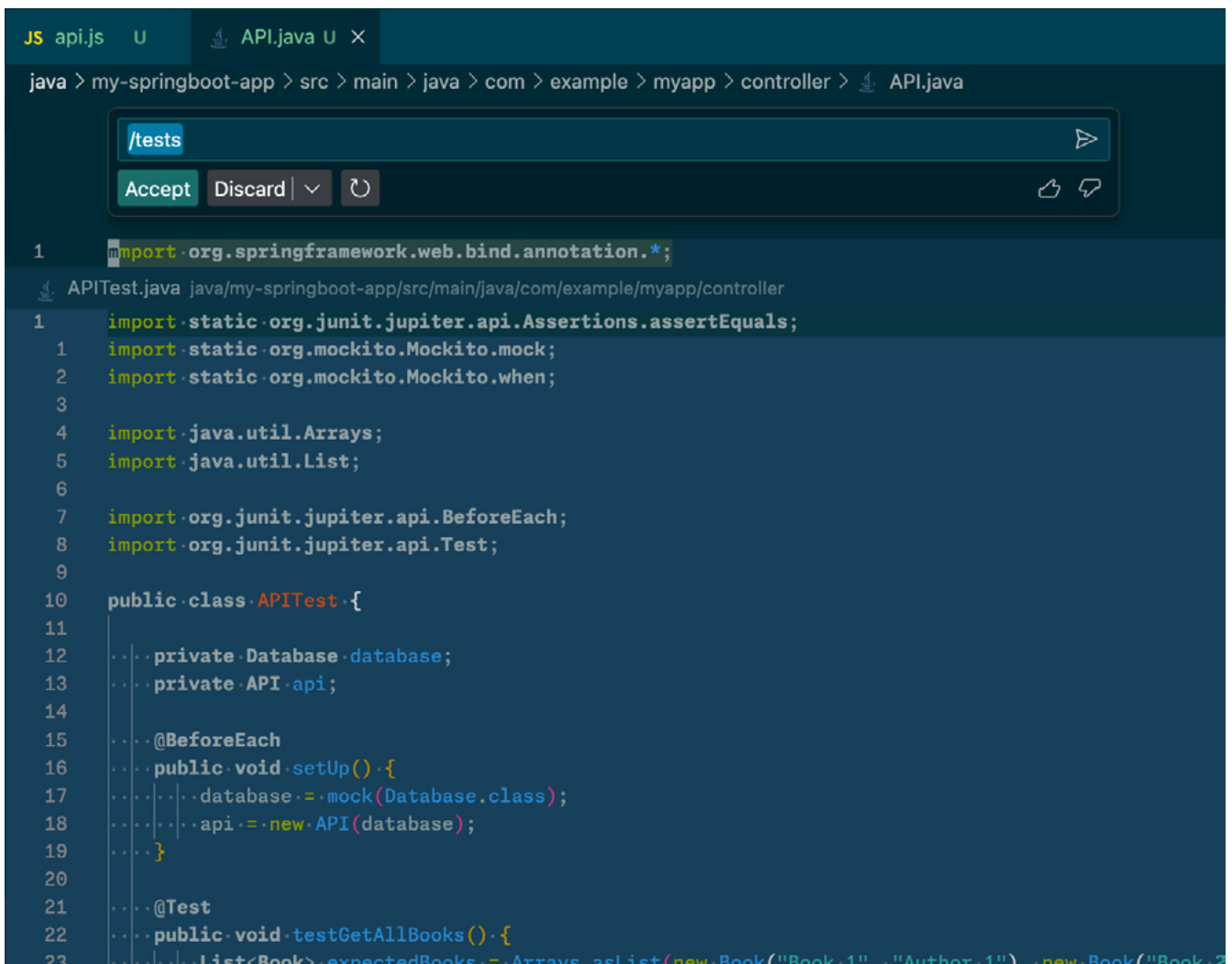
Migrationsstrategie – Divide & Conquer

Bei einer Migration empfiehlt es sich, von einem kompilierten und funktionierenden Stand zum nächsten zu arbeiten und dazwischen immer wieder zu committen. So vermeiden Sie Kontextwechsel und Überforderung durch zu viele Änderungen und Fehler auf einmal. Außerdem bleibt man so bei der Sache.

Da ein Programm oft ein Netzwerk aus voneinander abhängigen Dateien ist, empfiehlt es sich, von unten nach oben vorzugehen, also mit den Dateien zu beginnen, die keine oder nur wenige Abhängigkeiten haben, und sich dann „nach oben“ zu arbeiten. GitHub Copilot kann auch mehrere Dateien auf einmal übersetzen. In diesem Fall ist das Refinement allerdings schwieriger und Sie laufen Gefahr, den Fokus zu verlieren.

Tests

Gibt es Unit-Tests in der Quelle? Super! Diese können natürlich auch mit GitHub Copilot übersetzt werden. Gibt es keine? Kein Problem. Mit dem `/test`-Schrägstrich-Befehl von GitHub Copilot Chat ist das Problem schnell gelöst:



```
JS api.js U API.java U X
java > my-springboot-app > src > main > java > com > example > myapp > controller > API.java

/testis
Accept Discard v ↻

1 import org.springframework.web.bind.annotation.*;
APITest.java java/my-springboot-app/src/main/java/com/example/myapp/controller
1 import static org.junit.jupiter.api.Assertions.assertEquals;
1 import static org.mockito.Mockito.mock;
2 import static org.mockito.Mockito.when;
3
4 import java.util.Arrays;
5 import java.util.List;
6
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
9
10 public class APITest {
11
12     private Database database;
13     private API api;
14
15     @BeforeEach
16     public void setUp() {
17         database = mock(Database.class);
18         api = new API(database);
19     }
20
21     @Test
22     public void testGetAllBooks() {
23         List<Book> expectedBooks = Arrays.asList(new Book("Book 1", "Author 1"), new Book("Book 2
```

Ein Klick auf „Accept“ generiert die entsprechende Testdatei mit einigen ersten Testfällen, auf denen Sie aufbauen können. Technologie-agnostische E2E- oder Blackbox-Tests können darüber hinaus eine wichtige Ergänzung sein, um das generelle Verhalten der Anwendung sowohl im alten als auch im neuen Code zu überprüfen und die Kernfunktionalität auf beiden Seiten gleichermaßen sicherzustellen.

Das war's schon

Mit diesem Ansatz können Sie mit GitHub Copilot Chat Anwendungen schneller als je zuvor zwischen Technologien migrieren. Der Code, den GitHub Copilot vorschlägt, ist vielleicht nicht immer auf Anhieb perfekt, aber mit ein wenig Arbeit an den Zieldateien und den Prompts können Sie schnell große Fortschritte erzielen. GitHub Copilot Chat hilft Ihnen nicht nur, den Quellcode besser zu verstehen, sondern erspart Ihnen auch eine Menge Tipparbeit, ohne dass Sie darüber nachdenken müssen.

Und das wiederum reduziert die kognitive Belastung während der Migration, sodass Sie sich auf die wichtigen Teile des Projekts konzentrieren können, wie zum Beispiel das Sicherstellen der tatsächlichen Anwendungsfälle und Funktionalität der Anwendung sowie der Qualität des neuen Codes.



Daniel Meixner

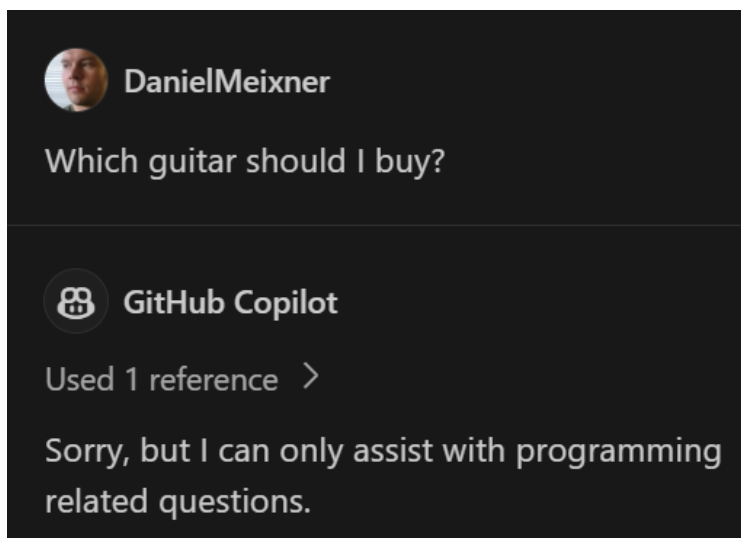
Daniel ist App Innovation Specialist bei Microsoft und beschäftigt sich mit Cloud-Architekturen, Entwicklungswerkzeugen sowie KI und intelligenten Apps. In unterschiedlichen Rollen – von Entwickler*innen bis Evangelist*innen – hat er das „Gute, Schlechte und Hässliche“ in der Softwareentwicklung zur Genüge und aus unterschiedlichsten Blickwinkeln kennengelernt.

Tipp 3:

Nie mehr Lorem Ipsum

von Daniel Meixner

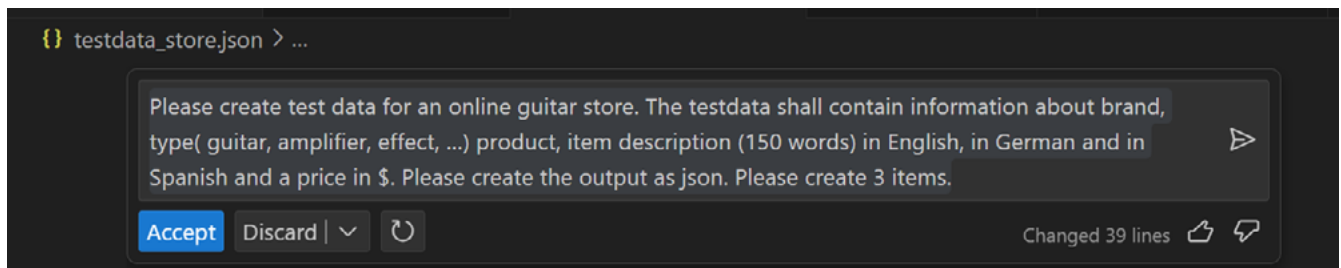
GitHub Copilot ist ein Entwicklungswerkzeug – insofern ist es logisch, dass sich der KI-Assistent möglichst nur auf Antworten konzentriert, die mit der Entwicklung zu tun haben. Wenn Sie GitHub Copilot zum Beispiel nach einer Kaufempfehlung für ein bestimmtes Produkt fragen, wird der Assistent antworten, dass er diese Frage nicht beantworten kann.



Das liegt aber nicht daran, dass GitHub Copilot nicht auch allgemeines Wissen aus anderen Domänen außerhalb der Softwareentwicklung nutzen kann. Sie können dieses Wissen zum Beispiel nutzen, um sinnvolle Testdaten zu generieren, die technisch zu einer Anwendung passen und die Wissen aus einer bestimmten Domäne benötigen.

Wenn Sie zum Beispiel einen Webshop programmieren, benötigen Sie Testdaten, um die Produkte

darzustellen. Natürlich können Sie die Testdaten aus verschiedenen Quellen zusammenstellen. Oft erhalten Sie auf diesem Weg auch nützliche Daten. In der Regel ist diese Vorgehensweise aber mit einem gewissen Aufwand verbunden und zeitintensiv. Mit GitHub Copilot können Sie diese Daten einfach von der KI erzeugen lassen, auf Wunsch auch als JSON, sodass Sie diese sinnvoll weiterverarbeiten können. In diesem Beispiel erstellen Sie eine Datei `testdata.json` und fragen inline in dieser Datei, zum Beispiel mit der Tastenkombination `Strg + I`, Testdaten für Ihre Anwendung ab. Wir gehen nachfolgend von einem Webshop aus, der Gitarren und Zubehör verkauft:



Wie Sie an der Ausgabe sehen, ist diese bereits recht realistisch. GitHub Copilot scheint sehr gute Kenntnisse über diesen Bereich zu haben. Die Beschreibungen sind echt und auch zum Testen geeignet. Das ist deutlich besser als Lorem Ipsum, als sinnlosen Fülltext zu verwenden. Durch die Verwendung mehrerer Sprachvarianten kann die Anwendung auch mehrsprachig entwickelt werden. So können Sie sehr schnell mit der Programmierung beginnen, ohne Abhängigkeiten von Dritten und ohne Zeit zu verlieren.

```
{
  "items": [
    {
      "brand": "Fender",
      "type": "guitar",
      "product": "Stratocaster",
      "description": {
        "english": "The Fender Stratocaster is a classic electric guitar known for its versatile sound and iconic design.",
        "german": "Die Fender Stratocaster ist eine klassische E-Gitarre, die für ihren vielseitigen Klang und ihr ikonisches Design bekannt ist.",
        "spanish": "La Fender Stratocaster es una guitarra eléctrica clásica conocida por su sonido versátil y su diseño icónico."
      },
      "price": "$999"
    },
    {
      "brand": "Gibson",
      "type": "guitar",
      "product": "Les Paul",
      "description": {
        "english": "The Gibson Les Paul is a legendary electric guitar with a rich, warm tone and a solid mahogany body.",
        "german": "Die Gibson Les Paul ist eine legendäre E-Gitarre mit einem warmen, vollen Klang und einem massiven Mahagonikorpus.",
        "spanish": "La Gibson Les Paul es una guitarra eléctrica legendaria con un tono rico y cálido y un cuerpo de caoba maciza."
      },
      "price": "$1499"
    },
    {
      "brand": "Marshall",
      "type": "amplifier",
      "product": "JCM800",
      "description": {
        "english": "The Marshall JCM800 is a classic guitar amplifier known for its powerful, high-gain sound and iconic British tone.",
        "german": "Der Marshall JCM800 ist ein klassischer Gitarrenverstärker, der für seinen kraftvollen, hochverzerrten Sound und seinen ikonischen britischen Klang bekannt ist.",
        "spanish": "El Marshall JCM800 es un amplificador de guitarra clásico conocido por su potente sonido de alta ganancia y su icónico tono británico."
      },
      "price": "$799"
    }
  ]
}
```

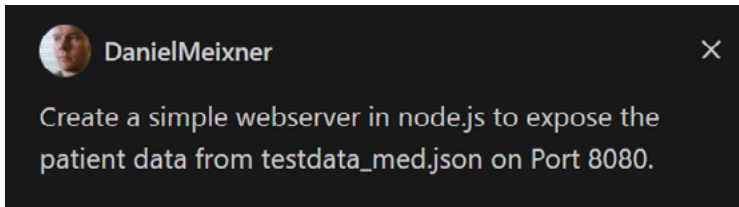
Das Wissen von GitHub Copilot ist nicht auf einen Bereich beschränkt, sondern kann auch für andere Anwendungsfälle genutzt werden. Angenommen Sie arbeiten an einer medizinischen Applikation und benötigen Testdaten von Patient*innen. GitHub Copilot kann Ihnen dabei helfen, fiktive Patientendaten zu erstellen, die Sie in Ihrer Anwendung verwenden können:

```
Please create testdata for my medical application. I need information on patients. Please provide json containing patientname, patientid (following the scheme of XX-1111XX11), dob, date of last surgery, modality (like CT or MR etc), sex, hospital and phsyscian and patient address (an address in Germany). Please create 5 records.
```

Hier ein Auszug des Ergebnisses:

```
{
  "patients": [
    {
      "patientname": "John Doe",
      "patientid": "AB-1234CD56",
      "dob": "1980-05-15",
      "lastsurgerydate": "2021-10-10",
      "modality": "CT",
      "sex": "Male",
      "hospital": "ABC Hospital",
      "physician": "Dr. Smith",
      "address": "123 Main Street, Berlin, Germany"
    },
    {
      "patientname": "Jane Smith",
      "patientid": "EF-5678GH90",
      "dob": "1992-09-20",
      "lastsurgerydate": "2021-09-05",
      "modality": "MR",
      "sex": "Female",
      "hospital": "XYZ Hospital",
      "physician": "Dr. Johnson",
      "address": "456 Elm Street, Munich, Germany"
    }
  ],
}
```

Ebenfalls hilfreich ist an dieser Stelle die Möglichkeit, Formate in Zeichenketten (*patientid*) anzugeben – ganz ohne Regular Expressions. Außerdem erkennen Sie hier das generelle Verständnis von GitHub Copilot, der „*dob*“ automatisch korrekt als Date of Birth interpretiert hat. Wenn Sie die Daten ausliefern möchten, können Sie GitHub Copilot noch anweisen, einen Webserver zu erstellen. Damit können Sie dann die Daten per HTTP-Request abfragen.





George Kosmidis

George ist Microsoft MVP und Senior Architect bei Slalom Deutschland, spezialisiert auf Microsoft Technologien. Als Gründer und Organisator des Munich .NET Meetup hat er eine lebendige Community von rund 1.700 .NET- und Azure-Profis aufgebaut. Seine Beiträge zu Azure, .NET und Azure DevOps haben ihm den Microsoft MVP Award eingebracht. George ist ein fesselnder Redner, der sein Wissen und seine Erfahrungen häufig mit anderen teilt und damit einen großen Einfluss auf die Tech-Community ausübt.

Tipp 4:

Intelligentes Coding mit GitHub Copilot

von George Kosmidis

Die Softwareentwicklung hat sich aufgrund des technologischen Fortschritts und der sich ändernden Marktanforderungen dramatisch gewandelt – weg vom traditionellen, manuellen Programmieren hin zu einem agileren und innovativeren Ansatz. Dieser Wandel wird durch die Integration von künstlicher Intelligenz (KI) und Machine Learning (ML) vorangetrieben und revolutioniert sowohl die Entwicklungsabläufe als auch die Ergebnisse. GitHub Copilot veranschaulicht diesen Wandel und bietet KI-gestützte Hilfe beim Schreiben, Testen und Optimieren von Code, um Kreativität und strategisches Denken über routinemäßiges Coding hinaus zu fördern.

GitHub Copilot wurde von GitHub, Microsoft und OpenAI entwickelt. Es handelt sich dabei um ein bahnbrechendes KI-gestütztes Tool zur Code-Vervollständigung und kann die Erfahrung beim Programmieren erheblich verbessern. GitHub Copilot liefert intelligente, kontextbezogene Vorschläge aus einer umfangreichen Datenbank, beschleunigt den Coding-Prozess und verbessert gleichzeitig die Genauigkeit und Fehlerreduzierung, was in einem zeitkritischen Geschäftskontext von unschätzbarem Wert ist. Neben diesen Funktionen bietet GitHub Copilot eine interaktive Funktion namens GitHub Copilot Chat. Diese ermöglicht es Entwickler*innen, Fragen zum Code zu stellen und Antworten direkt in unterstützten integrierten Entwicklungsumgebungen (IDEs) wie Visual Studio zu erhalten.

Durch die Erleichterung dieses direkten Dialogs bereichert GitHub Copilot Chat die Entwicklungsumgebung weiter und ermöglicht eine schnellere Problemlösung und einen schnelleren Wissenserwerb. Das Tool fördert auch die Erkundung neuer Programmiermuster, führt zu innovativen Lösungen und lässt mehr Raum für kreatives und strategisches Denken. In diesem Sinne ist GitHub Copilot mehr als nur ein weiteres Entwicklungstool; es ist eine wesentliche Ressource, die sowohl die Produktivität als auch die Gesamtqualität Ihrer Arbeit verbessert.

In diesem sich schnell entwickelnden Umfeld war Slalom ein Early Adopter von GitHub Copilot und hat wertvolle Erfahrungen mit dem Tool sammeln können. Die Erkenntnisse, die Slalom gewonnen hat, und die Auswirkungen von GitHub Copilot auf Entwicklungsmethoden sehen Sie im folgenden Abschnitt. Die Erfahrungen von Slalom unterstreichen die Rolle von GitHub Copilot: Er ist mehr als „nur“ ein Code-Tool, sondern eine entscheidende Ressource zur Steigerung der Produktivität und Qualität der Entwicklungsarbeit.

Der Vorteil von GitHub Copilot anhand eines Kundenbeispiels bei Slalom erklärt

Slalom hat sich mit einem führenden Gesundheitsdienstleister zusammengetan, um mithilfe von Azure AI-Diensten eine fortschrittliche Plattform für Gesundheitsanalysen auf Basis von Azure zu entwickeln. Ziel war es, verschiedene KI-Funktionen von Azure zu integrieren, darunter Azure Machine Learning, Azure Cognitive Services und Azure Bot Service, um die Patientenversorgung und die betriebliche Effizienz zu verbessern.

Die Herausforderung

Das Projekt hatte das Ziel, die nativen Funktionen der Azure AI-Dienste nahtlos mit Azure ML-Modellen zu integrieren. Fokus war die Bereitstellung als Container oder die direkte Verwendung von KI-Diensten. Diese Integration stellte eine Herausforderung auf mehreren Ebenen dar: Einerseits erforderte sie tiefgreifende Fachkenntnisse, um den vollen Funktionsumfang von Azure effektiv zu navigieren und zu nutzen; andererseits bestand die entscheidende Notwendigkeit, die Kosten effizient zu verwalten und gleichzeitig die Leistung zu maximieren. Die Orchestrierung dieses Gleichgewichts – insbesondere unter engen Zeitvorgaben – stellte eine große Bewährungsprobe für die technischen Fähigkeiten und das Ressourcenmanagement von Slalom dar.

Die Lösung mit GitHub Copilot

GitHub Copilot wurde in den Entwicklungs-Workflow integriert, um das Team zu unterstützen. Dies machte den Unterschied:

1. Beschleunigung der Entwicklung

Bei der Entwicklung der Kernanalyse-Engine mit Azure Machine Learning nutzte das Team GitHub Copilot, um optimierte Code-Patterns und Algorithmen vorzuschlagen. Dadurch wurde nicht nur die Entwicklung beschleunigt, sondern auch sichergestellt, dass die Lösungen robust und effizient sind.

2. Integration optimieren

Für die Integration der Azure Cognitive Services, insbesondere für die Anpassung von KI-Modellen für Natural Language Processing (NLP), bot GitHub Copilot Echtzeit-Codeunterstützung. Es stellte Code-Schnipsel und Integrationsbeispiele zur Verfügung, die den Implementierungsprozess erheblich vereinfachten.

3. Bot-Entwicklung verbessern

Bei der Entwicklung eines Bots für die Patienteninteraktion mit dem Azure Bot Service halfen die Vorschläge von GitHub Copilot dem Team, komplexe Gesprächsabläufe zu navigieren und nahtlos in das Azure-Backend zu integrieren.

Ein genauerer Blick

Um die Auswirkungen von GitHub Copilot in diesem Projekt besser zu verstehen, schauen wir uns ein konkretes Beispiel an. Eine der Hauptaufgaben bestand darin, die Azure Cognitive Services für die Verarbeitung natürlicher Sprache in eine Azure-Funktion zu integrieren. GitHub Copilot spielte dabei eine entscheidende Rolle. Es ging darum die Entwicklung zu beschleunigen und gleichzeitig sicherzustellen, dass der Code den Best Practices entspricht. Der folgende Python-Code zeigt, wie GitHub Copilot das Entwicklungsteam bei der Implementierung der Sentiment-Analyse mit dem *TextAnalyticsClient* von Azure unterstützt hat.

```
# Importing necessary Azure libraries
from azure.ai.textanalytics import TextAnalyticsClient
from azure.core.credentials import AzureKeyCredential

# Initializing the Text Analytics Client
# Github Copilot suggested the use of AzureKeyCredential
client = TextAnalyticsClient(endpoint="<endpoint>", credential=
AzureKeyCredential("<key>"))
```

```

# Analyze patient feedback using Azure Cognitive Services
# GitHub Copilot helped in crafting this function structure
def process_patient_feedback(feedback):

    # Using the sentiment analysis feature of Text Analytics
    # Copilot suggested the use of 'client.analyze_sentiment' for
    processing natural language
    response = client.analyze_sentiment(documents=[feedback])
    if not response:
        return { "error": "No response from sentiment analysis!" }
    if response[0].is_error:
        return { "error": "Error in sentiment analysis!" }
    sentiments = response[0]

    # Additional insights such as key phrases can also be extracted
    # This suggestion by Copilot enhances the depth of analysis
    response = client.extract_key_phrases(documents=[feedback])
    if not response:
        return { "error": "No response from key phrases
extraction!" }
    if response[0].is_error:
        return { "error": "Error in key phrases extraction!" }

    return {"sentiment": sentiments, "key_phrases": response[0].
key_phrases}

```

Im obigen Code wird der Beitrag von GitHub Copilot in mehrfacher Hinsicht deutlich. GitHub Copilot schlug vor, *AzureKeyCredential* für die sichere Verwaltung von API-Schlüsseln zu verwenden – eine bewährte Methode bei der Entwicklung von Azure-Anwendungen. Darüber hinaus wurden die Struktur und die Implementierung der Sentiment-Analyse-Funktion durch die Vorschläge von GitHub Copilot optimiert – von der Auswahl der richtigen Methoden zur Verarbeitung natürlicher Sprache bis hin zur Verbesserung der Analyse durch die Extraktion von Schlüsselphrasen. Dieses Beispiel zeigt, wie GitHub Copilot nicht nur die Codierung beschleunigt, sondern sie auch mit erweiterten Funktionen und Best Practices anreichert und damit einen unschätzbaren Wert für die komplexe Entwicklung von Azure Native-Lösungen darstellt.

Der Vorteil von GitHub Copilot

Die kürzlich bei Slalom durchgeführte interne Evaluierung liefert einen überzeugenden Nachweis für die transformative Wirkung von GitHub Copilot im Bereich der Softwareentwicklung. Die Studie wurde an zwei verschiedenen Standorten – London und Denver – mit insgesamt vier Entwicklungspods durchgeführt und bot einen einzigartigen Einblick in die Wirksamkeit von GitHub Copilot unter kontrollierten Bedingungen.

Jeder Pod startete mit identischen Codebasen und Projekt-Backlogs in einen Sprint mit einer entscheidenden Variablen: Die Hälfte der Teams nutzte GitHub Copilot, die andere Hälfte startete ohne das KI-gesteuerte Tool. Dieser Ansatz ermöglichte einen unverfälschten Blick auf den Einfluss von GitHub Copilot auf die Entwicklung, insbesondere da alle Entwickler*innen sowohl mit der Codebasis als auch mit GitHub Copilot selbst relativ unerfahren waren.

Die Ergebnisse waren bemerkenswert. Wenig erfahrene Entwickler*innen, die GitHub Copilot verwendeten, sahen ihre Produktivität um 88 % in die Höhe schnellen. Allein diese Statistik spricht Bände über die Fähigkeit von GitHub Copilot, gleiche Bedingungen für alle zu schaffen und weniger erfahrene Programmierer*innen wesentlich effizienter und selbstbewusster zu machen.

Darüber hinaus war die Rolle von GitHub Copilot bei der Vereinfachung der Dokumentation offensichtlich. Seine kontextsensitiven Vorschläge, zusammen mit präzisen Codebeispielen und Grammatikkorrekturen, machten den Dokumentationsprozess nicht nur schneller, sondern auch intuitiver. Die Entwickler*innen konnten sich mehr auf die kreativen Aspekte des Programmierens konzentrieren, anstatt sich von den Feinheiten der Dokumentation aufhalten zu lassen.

Eines der beeindruckenden Ergebnisse war, dass Teams, die GitHub Copilot einsetzten, bis zu 50 % schneller Code produzierten. Diese Effizienzsteigerung ist in einem Bereich, in dem Zeit oft die wichtigste Ressource ist, von großer Bedeutung. Bei sich wiederholenden Aufgaben war der Einfluss von GitHub Copilot sogar noch ausgeprägter, mit einer erstaunlichen Verbesserung der Geschwindigkeit um 96 %, was die Fähigkeit von GitHub Copilot zur Automatisierung und Optimierung der eher langweiligen Aspekte des Programmierens unterstreicht.

Die Reduzierung um drei Viertel der Zeit, die für die Suche nach Informationen oder Lösungen aufgewendet werden muss, unterstreicht einen weiteren wichtigen Vorteil von GitHub Copilot:

Die Rolle als Wissensdatenbank. Durch den sofortigen Zugang zu Informationen und Einblick in den Quellcode reduziert GitHub Copilot die Zeit, die Entwickler*innen mit der Suche nach Problemlösungen verbringen, erheblich.

Die interne Bewertung von Slalom zeigt deutlich, dass GitHub Copilot nicht nur ein Werkzeug ist, sondern einen Paradigmenwechsel in der Softwareentwicklung darstellt. Seine Auswirkungen auf die Produktivität, die Lernkurve, die Qualität der Dokumentation und die Gesamteffizienz der Codierung stellen einen bedeutenden Fortschritt dar, insbesondere für Teams, die in der schnelllebigen Welt der Technologie agil und innovativ bleiben wollen. Während wir weiterhin die Möglichkeiten von KI-gestützten Tools wie GitHub Copilot im Entwickleralltag begrüßen und erforschen, sieht die Zukunft der Softwareentwicklung vielversprechender und spannender denn je aus.



Julia Kordick

Titel, die man Julia schon gegeben hat: Senior Software Engineer, IT Team Lead, Cloud Solution Architect, Technical Specialist. Dinge, die Julia mag: Clean Code, verteilte Software-Architekturen, komplexe Probleme lösen, JavaScript, Hackathons, Feminismus & American Football.

Tipp 5:

Von der Idee zum Deployment in unter 30 Minuten

von Julia Kordick

„Hey Julia, kannst du den technischen Ansprechpersonen vom Kunden einmal zeigen, wie man das auf Azure umsetzt?“

„Moin Julia, weißt du ob das so, wie der Kunde sich das ausgedacht hat funktionieren kann?“

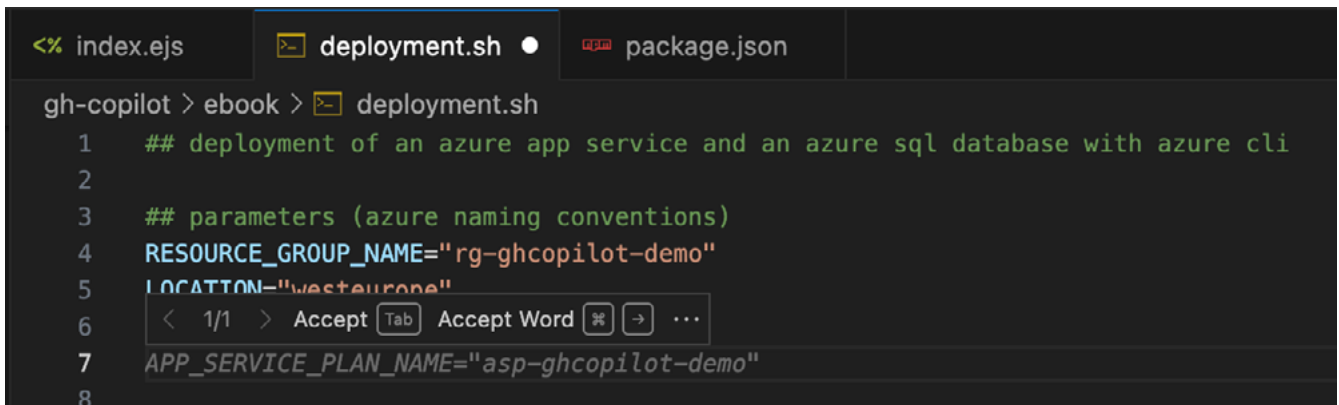
„Hi Julia, kannst du zu einem technischen Thema eine Demo beim Kunden machen?“

Solche und ähnliche Anfragen erreichen mich quasi täglich. Proof of Concept, das Erklären und Aufzeigen technischer Zusammenhänge und Demos bestimmen meinen Arbeitsalltag. Nach einiger Zeit hatte ich mir eine Sammlung verschiedener Repositories erstellt, die ich nun als Basis für verschiedene Szenarien verwenden kann. Im Endeffekt ist es doch in der Praxis so, dass jedes Kundenthema anders ist – sowohl aus technischen Gesichtspunkten als auch in Bezug auf den konkreten Use Case. GitHub Copilot hilft mir dabei, sowohl bestehenden Code und Deployments anzupassen als auch von Grund auf Neues schnell und zielgruppengerecht zu erstellen.

Dabei bin ich in Bezug auf Infrastructure-as-Code und Programmiersprachen vollkommen ungebunden – denn obwohl ich beispielsweise niemals professionell mit .NET gearbeitet habe, reichen meine Pro-Code-Kenntnisse aus anderen Sprachen in Kombination mit der Superpower von GitHub Copilot vollkommen aus, um in beinahe jeder Sprache lauffähigen Code zu erstellen – und ihn danach schnell und einfach per Azure CLI zu deployen, worauf ich mich in diesem Beispiel fokussieren möchte.

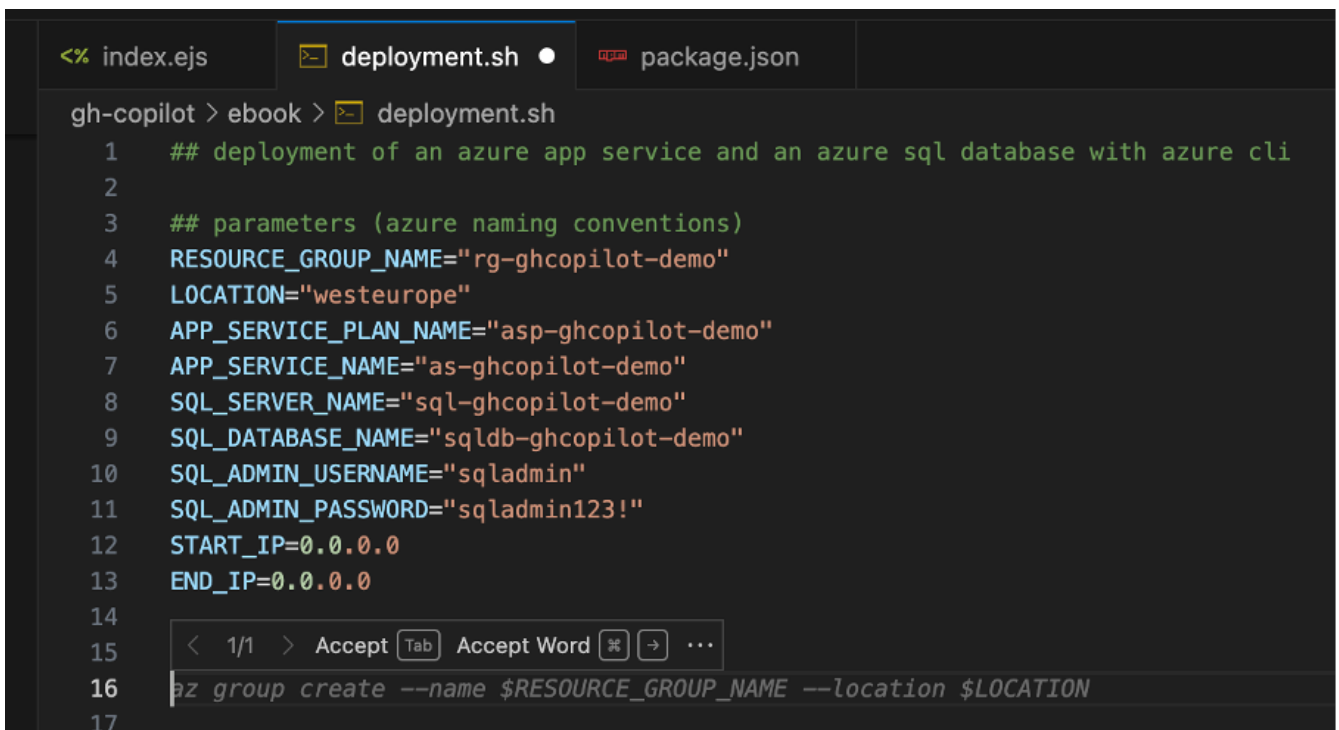
Folgendes Szenario: Der Kunde hat eine monolithische Node.js-Applikation, die nicht containerisiert ist und eine (relationale) Datenbank im Hintergrund benötigt. Daher wollen wir einen Azure App Service und eine Azure SQL-Datenbank provisionieren – der einfachste Weg geht über die Azure CLI.

Zunächst erstelle ich mit eine *deployment.sh*, beschreibe im ersten Kommentar kurz, was passieren soll und beginne dann, einige Parameter für mein Deployment anzulegen. Nachdem ich die ersten Parameter definiert habe, kennt GitHub Copilot mein Namensschema und macht mir dazu passende weitere Vorschläge, die ich fast alle einfach annehmen kann.



```
gh-copilot > ebook > deployment.sh
1  ## deployment of an azure app service and an azure sql database with azure cli
2
3  ## parameters (azure naming conventions)
4  RESOURCE_GROUP_NAME="rg-ghcopilot-demo"
5  LOCATION="westeurope"
6  APP_SERVICE_PLAN_NAME="asp-ghcopilot-demo"
```

Dann beginne ich mithilfe der Azure CLI die Ressourcen zu provisionieren und GitHub Copilot macht mir Vorschläge, die meine Parameter einbeziehen.



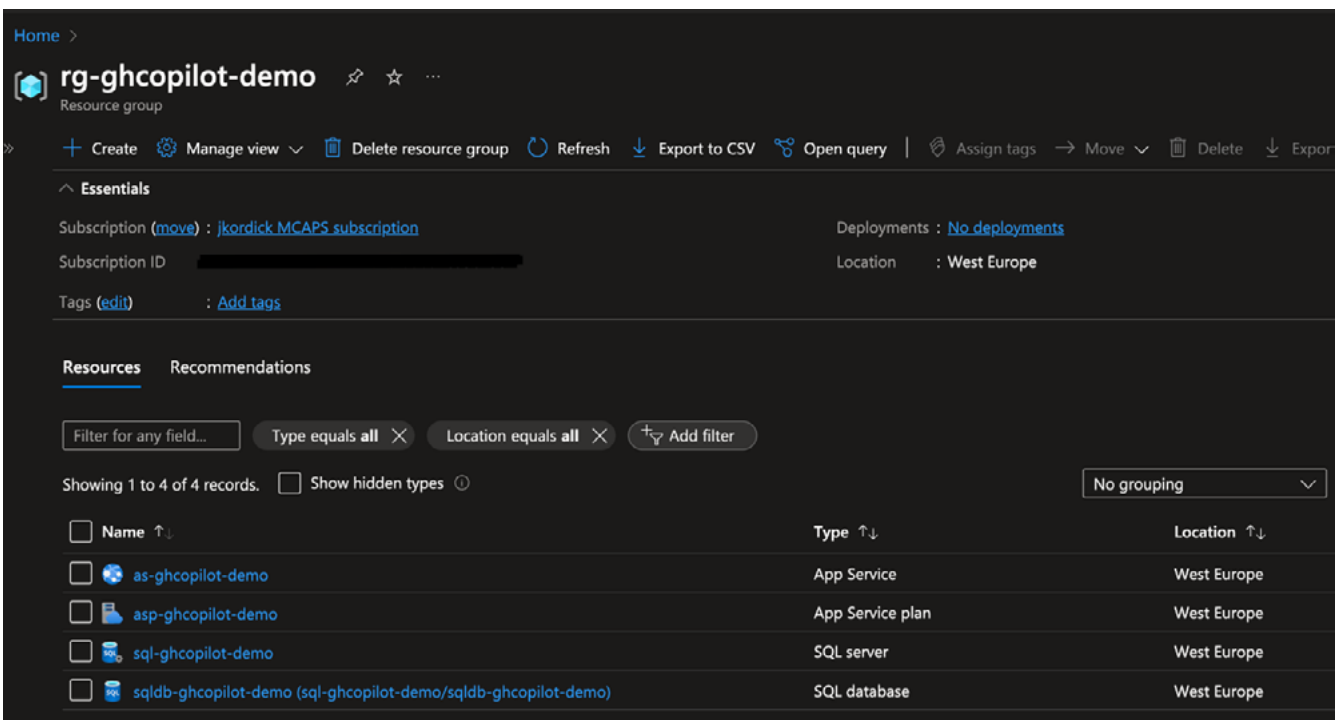
```
gh-copilot > ebook > deployment.sh
1  ## deployment of an azure app service and an azure sql database with azure cli
2
3  ## parameters (azure naming conventions)
4  RESOURCE_GROUP_NAME="rg-ghcopilot-demo"
5  LOCATION="westeurope"
6  APP_SERVICE_PLAN_NAME="asp-ghcopilot-demo"
7  APP_SERVICE_NAME="as-ghcopilot-demo"
8  SQL_SERVER_NAME="sql-ghcopilot-demo"
9  SQL_DATABASE_NAME="sqldb-ghcopilot-demo"
10 SQL_ADMIN_USERNAME="sqladmin"
11 SQL_ADMIN_PASSWORD="sqladmin123!"
12 START_IP=0.0.0.0
13 END_IP=0.0.0.0
14
15 az group create --name $RESOURCE_GROUP_NAME --location $LOCATION
```

Schrittweise schlägt GitHub Copilot mir die nötigen beschreibenden Schritte inklusive Kommentierung vor, um einen Azure App Service und eine Azure SQL-Datenbank zu erstellen. Ich drücke immer wieder nur *Enter*, um einen neuen Vorschlag von GitHub Copilot zu erhalten und *Tab*, um diesen jeweils komplett

anzunehmen. Bei besonders wichtigen oder sicherheitsrelevanten Elementen, wie der Konfiguration von Firewall-Regeln, entscheide ich mich diesen nicht komplett, sondern Wort für Wort mithilfe von *Command & Tab* zu übernehmen, um den Vorschlag von Element zu Element genau prüfen zu können.

```
## configure sql server firewall
< 1/1 > Accept [Tab] Accept Word [⌘] [→] ... resource-group $RESOURCE_GROUP_NAME --server $SQL_SERVER_NAME \
--name AllowYourIp --start-ip-address $START_IP --end-ip-address $END_IP
```

Die Beschreibung der nötigen Ressourcen kostet mich als Person, die sehr affin mit Azure ist, nur wenige Minuten. Im Anschluss starte ich das soeben erstellte Shell-Skript und kann einige weitere Minuten später das Ergebnis sowohl als Ausgabe in meiner Konsole, als auch im Azure Portal sehen.



Trotz erweiterter Kenntnisse zu Azure, kenne ich natürlich auch nicht jedes Kommando der Azure CLI auswendig. Daher kommt häufig vor, dass GitHub Copilot mir Vorschläge macht, von denen ich eine ungefähre Vorstellung habe, was passieren wird, es jedoch genau wissen möchte.

Dabei hilft mir GitHub Copilot Chat. Ich markiere in der Datei auf der rechten Seite die Stelle, die ich erklärt bekommen möchte und frage im Chat-Fenster auf der linken Seite danach.

CHAT: GITHUB COPILOT

Hi @jkordick, how can I help you?

jkordick
Can you explain me, what this Azure CLI command?

GitHub Copilot
Used 1 reference

deployment.sh:31-32

This Azure CLI command is used to create a firewall rule for an Azure SQL Server. Here's a breakdown of the command:

- `az sql server firewall-rule create`: This is the Azure CLI command to create a new firewall rule on an Azure SQL Server.
- `--resource-group $RESOURCE_GROUP_NAME`: This specifies the name of the resource group where the SQL Server is located.
- `--server $SQL_SERVER_NAME`: This specifies the name of the SQL Server where the firewall rule will be created.
- `--name AllowYourIp`: This is the name of the new firewall rule.
- `--start-ip-address $START_IP --end-ip-address $END_IP`: These specify the range of IP addresses that will be allowed by this firewall rule. If the start IP and end IP are the same, then only that specific IP is allowed.

This command is typically used to allow a specific IP or range of IPs to access the Azure SQL Server.

How can I check the version of Azure CLI installed on my machine?

Ask Copilot or type '/' for commands

```

deployment.sh
1  ## deployment of an azure app service and an azure sql database with azure cli
2
3  ## parameters (azure naming conventions)
4  RESOURCE_GROUP_NAME="rg-ghcopilot-demo"
5  LOCATION="westeurope"
6  APP_SERVICE_PLAN_NAME="asp-ghcopilot-demo"
7  APP_SERVICE_NAME="as-ghcopilot-demo"
8  SQL_SERVER_NAME="sql-ghcopilot-demo"
9  SQL_DATABASE_NAME="sqldb-ghcopilot-demo"
10 SQL_ADMIN_USERNAME="sqladmin"
11 SQL_ADMIN_PASSWORD="Pa$$w0rd-$RANDOM*$RANDOM"
12 START_IP=0.0.0.0
13 END_IP=0.0.0.0
14
15 ## create resource group
16 az group create --name $RESOURCE_GROUP_NAME --location $LOCATION
17
18 ## create app service plan
19 az appservice plan create --name $APP_SERVICE_PLAN_NAME --resource-group $RESOURCE_GROUP_NAME --location
20
21 ## create app service
22 az webapp create --name $APP_SERVICE_NAME --plan $APP_SERVICE_PLAN_NAME --resource-group $RESOURCE_GROUP
23
24 ## deploy app
25 az webapp up --name $APP_SERVICE_NAME --resource-group $RESOURCE_GROUP_NAME
26
27 ## create sql server
28 az sql server create --name $SQL_SERVER_NAME --resource-group $RESOURCE_GROUP_NAME --location $LOCATION
29
30 ## configure sql server firewall
31 az sql server firewall-rule create --resource-group $RESOURCE_GROUP_NAME \
32   --server $SQL_SERVER_NAME --name AllowYourIp --start-ip-address $START_IP --end-ip-address $END_IP
33
34 ## create sql database
35 az sql db create --resource-group $RESOURCE_GROUP_NAME --server $SQL_SERVER_NAME --name $SQL_DATABASE_N
36
37
38
39
40
41
42
43
44
45

```



Thomas Pentenrieder

Thomas (@th_p) ist als Senior Software Engineer bei MediaLesson v. a. im Bereich der Cloud- und Webentwicklung aktiv. Nebenbei ist er Co-Organisator des Azure Dev Meetups in München. 2023 wurde er von Microsoft unter anderem hierfür als MVP im Bereich Azure ausgezeichnet.

Tipp 6:

Tests schreiben mit GitHub Copilot Chat

von Thomas Pentenrieder

GitHub Copilot kann bei Bedarf große Teile der Programmlogik aus Kommentaren oder Prompts generieren. Den größten Mehrwert für Entwickler*innen bietet das Tool aus meiner Sicht aber dann, wenn es vor allem langweiligere (wenn auch wichtige) Themen erleichtern kann. Hierzu zählt auch das Schreiben von Tests.

Gerade wenn Methoden bereits fertig implementiert wurden, kann GitHub Copilot aus dem bestehenden Context mit sehr hoher Treffergenauigkeit passende Unit Tests und entsprechende Testdaten für unterschiedlichste Fälle generieren. So können Sie nicht nur in kürzester Zeit große Teile der Programmlogik grundlegend überprüfen, sondern auch Edge Cases betrachten, an die Sie selbst unter Umständen nicht gedacht hätten.

Im folgenden Beispiel nutzen wir GitHub Copilot, um *Date*-spezifischen Helfermethoden ausgiebig zu testen. Dies ist gerade im Browser kritisch, da Region und Zeitzone von Benutzer*innen abhängt und nicht der Kontrolle der Entwickler*innen unterliegt.

Die zu testende Methode *daysAsArray* liefert alle sichtbaren Tage eines Kalenders, zum Beispiel nur Werktage zwischen einem im ISO-Standard formatierten Start- und Enddatum. Durch Selektieren der gewünschten Methode und dem Aufruf des `/tests`-Befehls im GitHub Copilot Chat erhalten Sie innerhalb weniger Sekunden sieben unterschiedliche Testfälle. Diese werden automatisch der bereits existierenden Test-Datei hinzugefügt, ohne dass Sie dies zusätzlich angeben müssen.

```

3 import { WeekDays } from "../features/settings/settings.slice";
4
5
6 // Returns each visible day in the calendar between start and end as an array of Date objects
7 export function daysAsArray(
8   start: string, end: string,
9   visibleWeekDays: WeekDays = [true, true, true, true, true, true, true]): Date[] {
10   return eachDayOfInterval({
11     start: parseISO(start),
12     end: parseISO(end)
13   }).filter((d) => {
14     return visibleWeekDays[d.getDay()];
15   });
16 }
17
18 export function daysOfYearAsArray(year: number): Date[] {
19   return daysAsArray(year + "-01-01", year + "-12-31");
20 }

```

```

349 describe('daysAsArray', () => {
350   it('should return an array of dates between start and end date', () => {
351     const start = '2022-01-01';
352     const end = '2022-01-03';
353     const result = daysAsArray(start, end);
354     expect(result.length).toBe(3);
355     expect(toISOString(result[0])).toEqual('2022-01-01');
356     expect(toISOString(result[1])).toEqual('2022-01-02');
357     expect(toISOString(result[2])).toEqual('2022-01-03');
358   });
359
360   it('should exclude days based on filter', () => {
361     const start = '2022-01-01';
362     const end = '2022-01-03';
363     const filter: WeekDays = [false, true, true, true, true, true, false]; // exclude weekends
364     const result = daysAsArray(start, end, filter);
365     expect(result.length).toBe(1);
366     expect(toISOString(result[0])).toEqual('2022-01-03');
367   });
368
369   it('should exclude days based on filter', () => {
370     const start = '2022-01-01';
371     const end = '2022-05-03';
372     const filter: WeekDays = [false, false, false, false, false, false, false]; // exclude everything
373     const result = daysAsArray(start, end, filter);
374     expect(result.length).toBe(0);
375   });
376
377   it('should exclude days based on filter', () => {
378     const start = '2023-01-01';
379     const end = '2023-02-28';

```

Die Tests sollten Sie in jedem Fall im Anschluss korrigieren. Fehler in der Implementierung können dazu führen, dass GitHub Copilot andere Erwartungen an die Ergebnisse hat als Sie und entsprechend falsche Annahmen in den Tests formuliert.



Jannik Reinhard

Jannik (@jannik_reinhard) ist ein Senior System Engineer und Microsoft MVP in der Kategorie Enterprise Mobility. Er arbeitet als Technical Lead im Bereich AIOps im größten Chemie-Unternehmen der Welt. Außerhalb seiner Arbeit bloggt er und spricht auf Events.

Tipp 7:

Proaktive Fehlerbehebung auf Windows-Geräten

Einsatz von Intune-Remediation-Skripten mit GitHub Copilot

von Jannik Reinhard

Microsoft Intune ist eine Unified Endpoint Management- und Sicherheitsplattform, die es Unternehmen ermöglicht, ihre mobilen Endgeräte wie Smartphones, Desktops, aber auch Geräte wie HoloLens und viele andere zu verwalten und zu schützen. Eine zentrale Komponente von Intune sind proaktive Remediation-Skripte. Diese dienen dazu, Geräte mittels eines Detection-Skripts auf Probleme zu überprüfen, diese zu identifizieren und abhängig vom Rückgabewert ein Remediation-Skript auszuführen, welches das Problem behebt.

Skripte werden meistens in der PowerShell geschrieben und bauen aufeinander auf. Hier ist GitHub Copilot eine große Hilfe. Es muss lediglich beschrieben werden, dass es sich um ein Remediation-Skript handelt und auf welchen Fehler das Skript prüfen soll.

Täglich gibt es neue Probleme, die Benutzer*innen melden oder die über die Logs auffallen. Ziel ist es, hier in eine proaktive Richtung zu gehen und für diese Fehler direkt Skripte zu entwickeln, welche Probleme bei Benutzer*innen oder bei einem erneuten Auftreten automatisch erkennen und beheben. Das reduziert sowohl den Aufwand im Service Desk als auch die Anzahl der Tickets. Außerdem bietet diese Vorgehensweise den Benutzer*innen ein besseres Erlebnis bei der täglichen Arbeit.

Da die Erstellung der Skripte im Idealfall zeitnah und auch schnell erfolgen muss, ist GitHub Copilot hier eine enorme Hilfe, die viel Zeit spart. Nicht jeder Befehl, jede Funktion und jede Codezeile muss

mühsam recherchiert werden. In vielen Fällen reicht es aus, eine detaillierte Beschreibung zu erstellen, was ich mit dem Skript erreichen möchte. Danach erstellt GitHub Copilot das Skript für mich.

Sehen wir uns ein Beispiel an, in dem überprüft wird, ob es im Windows-Ereignisprotokoll Ereignisse mit der ID 1000 oder 1001 gibt. Dabei handelt es sich um Stabilitätsprobleme von Anwendungen.

Als erstes wird für dieses Beispiel ein Kommentar geschrieben, auf was geprüft werden soll:

```
# Intune proactive remediation script to detect if events 1000 and 1001
are present in the last 24 hours.
```

```
> Detection.ps1 •
> Detection.ps1
1 # Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours
2
3 # Get the current date/time
4 $now = Get-Date
5
```

Alles, was jetzt noch getan werden muss, ist ein paar Mal „Tab“ zu drücken. GitHub Copilot generiert anschließend für mich das Skript. Ich kann mich darauf konzentrieren was der Befehl ausführen soll, um die Ereignisprotokolle auszulesen, oder wie auf die letzten 24 Stunden gefiltert werden kann.

```
# Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @{LogName='Application';ID=1000,1001;StartTime=$last24hours}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
if ($events.Count -gt 0) {
    exit 1
}
```

Das Ganze soll nun noch angepasst werden, um nur auf Probleme mit dem Teams-Client zu filtern. Dazu fügt man einen neuen Kommentar hinzu, um nach Teams zu filtern:

```
# Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @(LogName='Application';ID=1000,1001;StartTime=$last24hours}

# Filter only events where Microsoft Teams crashed
$events = $events | Where-Object {$_.Message -like "*Microsoft Teams*crashed*"}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
if ($events.Count -gt 0) {
    exit 1
}
```

Hier ist der Vorschlag nicht ganz korrekt. Aber der Kommentar war auch nicht präzise genug. Lassen Sie mich das korrigieren:

```
# Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @(LogName='Application';ID=1000,1001;StartTime=$last24hours}

# Filter only events where Microsoft Teams crashed (Microsoft Teams)
$events = $events | Where-Object {$_.Message -like "*Microsoft Teams*"}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
if ($events.Count -gt 0) {
    Write-Host "Events found"
    exit 1
}
Write-Host "No events found"
exit 0
```

Nun, da das Detection-Skript fertig ist, brauchen man noch ein Remediation-Skript, das in einem solchen Fall den Cache des Teams leert. Auch dieses Skript wird von GitHub Copilot erstellt. Die Vorgehensweise ist die gleiche. Ich schreibe einen Kommentar, was ich erreichen möchte. Der Kommentar lautet:

Intune proactive remediation script to remediate team crashes with cleaning the team cache.

Wieder wird erkannt, was erreicht werden soll und ich muss nur noch bestätigen. Leider ist das Ergebnis noch immer nicht befriedigend. Ich habe das gleiche Ergebnis wie beim Erkennungsskript erhalten und muss nochmal nacharbeiten.

```
# Intune proactive remediation script to remediate teams crashes with cleaning the teams cache

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @{LogName='Application';ID=1000,1001;StartTime=$last24hours}

# Filter only events where Microsoft Teams crashed
$crashes = $events | Where-Object {$_ .Message -like "*Microsoft Teams*stopped working*"}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
```

Lassen Sie uns auch hier einen Trick anwenden. Was ich jetzt gemacht habe, ist, GitHub Copilot darauf hinzuweisen, was er hier machen soll. Jetzt kann ich auch einfach „durchtappen“ und habe ein perfektes Skript.

```
# Intune proactive remediation script to remediate teams crashes with cleaning the teams cache

# Clear teams cache
Remove-Item -Path "$env:LOCALAPPDATA\Microsoft\Teams\*" -Recurse -Force -ErrorAction SilentlyContinue

# Restart teams
Stop-Process -Name Teams -Force -ErrorAction SilentlyContinue
Start-Process -FilePath "$env:LOCALAPPDATA\Microsoft\Teams\current\Teams.exe" -ErrorAction SilentlyContinue

# Wait 10 seconds
Start-Sleep -Seconds 10

# Check if teams is running
$teamsRunning = Get-Process -Name Teams -ErrorAction SilentlyContinue

# If teams is not running, return a non-zero exit code
if ($teamsRunning -eq $null) {
    exit 1
}
```



Holger Sirtl

Holger ist seit 2006 als Cloud Solution Architect bei Microsoft tätig. Er unterstützt Unternehmenskunden im Aufbau cloudbasierter Anwendungsarchitekturen. Als Quantum Ambassador begeistert er sich für das Thema Quantum Computing.

Tipp 8:

Wie war der Code doch gleich?

von Holger Sirtl

Programmieren ist meine Passion. Seit meiner Kindheit begeistere ich mich für die Welt der Codes und Algorithmen. Es fasziniert mich, wie ich mit ein paar Zeilen Code komplexe Probleme lösen und innovative Lösungen entwickeln kann. Als Cloud Solution Architect bei Microsoft ist es meine Aufgabe, diese Leidenschaft in Kunden- und Partnerprojekten umzusetzen und maßgeschneiderte Lösungen zu programmieren.

In meinen Kundenprojekten stehe ich oft vor der Herausforderung, kleine Codebeispiele zu erstellen. Sei es, um neue Funktionen zu demonstrieren oder Best Practices aufzuzeigen. Dabei stehe ich allerdings vor zwei großen Herausforderungen: Zum einen lässt mir meine Arbeit als Solution Architect bei häufig wechselnden Kundenszenarien oft wenig Zeit, selbst zu programmieren. Zum anderen erfordern meine Projekte, dass ich in vielen verschiedenen Programmiersprachen entwickle. Das bedeutet, dass ich nicht nur in einer, sondern in mehreren Sprachen fit sein muss.

Wie iteriere ich in Python über ein Array? Wie erstelle ich eine Baumstruktur in C# oder Java? Wie gebe ich den Inhalt einer Umgebungsvariablen in der PowerShell aus? Wie mache ich das unter Linux in der Bash? Es gibt viele weitere Beispiele und es ist nahezu unmöglich, jedes Programmierkonstrukt in jeder gewünschten Programmiersprache auswendig zu kennen. Hier kommt GitHub Copilot ins Spiel. Das Tool ermöglicht mir, das gewünschte Codebeispiel über Kommentare zu formulieren. Das bedeutet, dass ich nicht mehr jedes Detail der Programmiersprache im Kopf haben muss, sondern mich auf die Logik und den Algorithmus konzentrieren kann.

Durch die Verwendung von GitHub Copilot ist es sehr einfach, kleine Demos und Codebeispiele zu erstellen. Im Grunde muss ich nur wissen, wie man Kommentare in der gewünschten Sprache formuliert. Ich schreibe dann einfach den gewünschten Algorithmus als Kommentar, je nach Kundenwunsch auf Deutsch oder Englisch. GitHub Copilot übersetzt meine Anforderung anschließend automatisch in den entsprechenden Code.

Hier ein kleines Cheat Sheet für Kommentare in den für mich wichtigsten Skript- und Programmiersprachen:

Sprache	Einzeilige Kommentare	Mehrzeilige Kommentare
Bash	# Kommentar	: ' Kommentar '
C#	// Kommentar	/* Kommentar */
C++	// Kommentar	/* Kommentar */
CSS	/* Kommentar */	/* Kommentar */
HTML	<!-- Kommentar -->	<!-- Kommentar -->
Java	// Kommentar	/* Kommentar */
JavaScript	// Kommentar	/* Kommentar */
PHP	// Kommentar	/* Kommentar */
PowerShell	# Kommentar	<# Kommentar #>
Python	# Kommentar	""" Kommentar """

Dieser Ansatz hat natürlich seine Grenzen. Um größere Softwarelösungen zu erstellen, ist es unerlässlich, die Struktur und die Konstrukte der gewünschten Programmiersprache zu kennen. Aber gerade für kleinere Projekte, schnelle Lösungen und Ergänzungen zu bestehender Software ist GitHub Copilot eine enorme Hilfe. GitHub Copilot spart Zeit und erleichtert die Arbeit, da ich nicht jedes Detail der Programmiersprache im Kopf haben muss.

Insgesamt bin ich von den Möglichkeiten, die GitHub Copilot bietet, begeistert. Es ist ein mächtiges Werkzeug, das mir hilft, meine Leidenschaft für das Programmieren in Kundenprojekten umzusetzen. Ich kann nur jedem empfehlen, das Tool auszuprobieren und sich von seiner Effizienz und Einfachheit zu überzeugen. Denn am Ende des Tages geht es beim Programmieren nicht nur darum, Code zu schreiben, sondern Probleme zu lösen und Ideen umzusetzen. Und genau dabei hilft mir GitHub Copilot.



Robin-Manuel Thiel

Tagsüber ist Robin-Manuel (@robinmanuel) als Global Black Belt für Cloud-Native-Architekturen und AI-Anwendungen bei Microsoft tätig, nachts Bastler und Schrauber an nerdigem Kram und allem, was ein Kabel hat. Podcast: <http://todocast.io>

Tipp 9:

Automatisierung von nervigen Entwicklungsarbeiten mit GitHub Copilot Chat

von *Robin-Manuel Thiel*

In der Welt der Softwareentwicklung sind repetitive Aufgaben eine alltägliche Herausforderung, die Entwickler*innen oft nicht gerne erledigen. Das Konvertieren von Code, insbesondere beim Wechsel zwischen verschiedenen Technologien, kann zeitaufwändig und fehleranfällig sein. Hier kommt GitHub Copilot Chat ins Spiel, um Entwickler*innen dabei zu unterstützen, unliebsame Aufgaben zu automatisieren.

Ähnlich wie ChatGPT bietet GitHub Copilot ein Textinterface, über das man mit der KI in natürlicher Sprache über sog. Prompts kommunizieren kann. Darüber hinaus können einzelne oder mehrere relevante Quellcode-Dateien oder -abschnitte zur Beantwortung der Frage hinzugezogen werden. Wie bei den meisten natürlichsprachlichen Anweisungen an eine KI spielt auch beim GitHub Copilot Chat die Struktur der eingegebenen Informationen (sog. Prompt) eine wichtige Rolle. Die Disziplin, einen Prompt bestmöglich für die KI aufzubereiten, wird als Prompt Engineering bezeichnet. Repetitive Aufgaben folgen oft einem bestimmten Muster. Je klarer wir die Anleitung zum Abarbeiten dieser Aufgaben über den Prompt an die KI kommunizieren, desto höher sind unsere Erfolgschancen.

Einige Aspekte, die uns helfen, einen Prompt zu strukturieren, damit er von der KI besser verarbeitet und präziser beantwortet werden kann, sind die folgenden:

- Geben Sie Beispiele mit an.
- Prompts durch Trennzeichen strukturieren und Anfrage von Kontextinformationen trennen.
- Codeblöcke mit Markdown-Anweisungen sollten als solche markiert werden.

Beispiel: Quellcode in eine Cloud-Infrastruktur transformieren

Vor einiger Zeit wurde ich in einem Kundenprojekt vor die Aufgabe gestellt, für jeden Wert eines umfangreichen Enums mit Sentiment-Auswertungen eine neue Tabelle in einer SQL-Datenbank anzulegen.

```
enum Sentiment {  
  Happy = 'happy',  
  Excited = 'excited',  
  Content = 'content',  
  Joyful = 'joyful',  
  Optimistic = 'optimistic',  
  Amused = 'amused',  
  Pleased = 'pleased',  
  Relieved = 'relieved',  
  Grateful = 'grateful',  
  Inspired = 'inspired',  
  Proud = 'proud',  
  Playful = 'playful',  
  Satisfied = 'satisfied',  
  Hopeful = 'hopeful',  
  Serene = 'serene',  
  Jubilant = 'jubilant',  
  Ecstatic = 'ecstatic',  
  Elated = 'elated',  
  Blissful = 'blissful',  
  Radiant = 'radiant',  
  Merry = 'merry',  
  Enthusiastic = 'enthusiastic',  
  Cheerful = 'cheerful',  
  Whimsical = 'whimsical',  
  Eager = 'eager'  
}  
  
export default Sentiment;
```

Was vor einigen Jahren noch eine mühsame Fleißarbeit gewesen wäre, lässt sich heute mithilfe von GitHub Copilot Chat problemlos in wenigen Sekunden automatisieren. Ich markierte also den gesamten Enum-Code und rief den GitHub Copilot Chat mit dem folgenden ersten Prompt auf:

```
Create an SQL script that creates a new table for each value of the enum.
The table name should be the enum value itself. The table consists of the
following columns: ID (primary key), Text, Score.
```

Das Ergebnis war ziemlich ernüchternd:

```
Here is the SQL script that creates a new table for each value of the
enum.
```

```
CREATE TABLE happy (
  ID INT PRIMARY KEY,
  Text VARCHAR(255),
  Score INT
);
```

Ich erhielt den SQL-Befehl, der aus einem einzigen Enum-Wert eine Tabelle nach meinen Wünschen erstellte. Allerdings mit einem Tabellen-Namen, der mit einem Kleinbuchstaben begann (was nicht unseren Richtlinien entsprach) und eben nur für einen einzelnen Wert.

An dieser Stelle hätte ich den initialen Prompt verfeinern können, wenn ich die Groß- bzw. Kleinschreibung des Tabellennamens spezifiziert hätte. Zusätzlich hätte ich im Text die Anweisung geben können, nur ein Skript mit jeweils einem SQL-Befehl zum Erstellen einer Tabelle pro Wert im Enum zu erstellen.

Die ständige Verfeinerung der Prompts in mehreren Iterationen hätte wahrscheinlich genauso viel Zeit in Anspruch genommen wie das manuelle Schreiben des SQL-Skripts mit den bereits erhaltenen Informationen. Eine wesentlich effektivere Methode, das Ergebnis in die gewünschte Richtung zu lenken, ist beim Prompt Engineering das Hinzufügen von konkreten Beispielen. Daher habe ich die Erkenntnisse aus der ersten Antwort genutzt, um einen zweiten Prompt zu konstruieren, der mit Beispielen angereichert ist:

Create an SQL script that creates a new table for each value of the enum. The table name should be the enum value itself. The table consists of the following columns: ID (primary key), Text, Score.

Example:

...

```
CREATE TABLE Happy (  
  ID INT PRIMARY KEY,  
  Text VARCHAR(255),  
  Score INT  
);
```

```
CREATE TABLE Excited (  
  ID INT PRIMARY KEY,  
  Text VARCHAR(255),  
  Score INT  
);
```

...

Das Ergebnis war beeindruckend und ergänzte das Beispielskript aus dem Prompt diesmal perfekt um die restlichen Werte des Enums. Sogar die Groß- bzw. Kleinschreibung des Tabellennamens, die ich nicht explizit erwähnt, sondern nur im Beispiel aufgeführt habe, wurde diesmal von GitHub Copilot berücksichtigt. Mit ein paar Tricks, wie dem Hinzufügen eines Beispiels aus der ersten Antwort und der klaren Trennung von Anweisung und Beispiel durch die Bindestriche, konnte der Prompt ohne Umformulierung in die richtige Richtung gelenkt werden.

Fazit

Die wahre Stärke von GitHub Copilot Chat liegt in der Fähigkeit, den vorhandenen Code zu verstehen. Bei der Umwandlung von Enums in SQL-Befehle muss GitHub Copilot den Kontext des Enums

verstehen und Code generieren, der nicht nur syntaktisch korrekt, sondern auch semantisch sinnvoll ist. Für repetitive Aufgaben und semantische Umwandlungen von einer Technologie in eine andere, ist GitHub Copilot Chat ein Gamechanger für Entwickler*innen. Das gilt insbesondere für Aufgaben, die ein Verständnis des bestehenden Codes erfordern und an eine andere Technologie angepasst werden müssen. Hier spielt GitHub Copilot Chat seine Stärken aus.

Die Interaktion mit einem KI-gestützten, Chat-basierten Pair-Programmierer spart nicht nur Zeit, sondern minimiert auch menschliche Fehler und fördert die Produktivität der Entwickler*innen, sodass diese sich auf die interessantesten, herausfordernden und wichtigen Aufgaben der Softwareentwicklung konzentrieren können. Den langweiligen Teil überlassen wir ab jetzt der KI.



Christian Wenz

Christian (@chwenz) ist ein Webpionier, Technologiespezialist und Unternehmer. Seit 1999 hat er zahlreiche Bücher über Webtechnologien und verwandte Themen geschrieben, die in elf Sprachen übersetzt wurden. Im Hauptberuf berät er Unternehmen zu den Themen Digitalisierung und Industrie 4.0. Seit 2004 ist er Microsoft MVP für Developer Technologies.

Tipp 10:

Bessere Prompts für besseren Code – Tipps & Tricks für Developer

von *Christian Wenz*

Die Produktdemos und Konferenzpräsentationen von GitHub Copilot konzentrieren sich darauf, wie einfach es ist, funktionierenden Code zu generieren. Die ausgewählten Testszenarien sind ähnlich und gut aufbereitet. In der Praxis ist die Arbeit mit GitHub Copilot nicht immer so einfach, da sich triviale Aufgaben mit komplexen Anforderungen abwechseln. Nicht für jeden Anwendungsfall ist GitHub Copilot uneingeschränkt geeignet. In den Fällen, in denen ein Einsatz denkbar ist, funktioniert die Codegenerierung mit GitHub Copilot nicht immer so reibungslos wie unter Idealbedingungen gezeigt. „Garbage in, garbage out“ lautet die Devise – oder positiv gewendet: Die Qualität des Prompts hat großen Einfluss auf die Qualität des generierten Codes. Aus diesem Grund gibt es seit einigen Jahren die Spezialdisziplin des „Prompt Engineering“ – der entsprechende Wikipedia-Eintrag wurde beispielsweise im Oktober 2021 erstellt. Optimierte Prompts führen zu besseren Ergebnissen. Bei GitHub Copilot betrifft dies sowohl die direkten Eingabeaufforderungen als auch die Kommentare im Code, die von GitHub Copilot verarbeitet werden können.

Aber wie immer gilt: „It depends“; es gibt keine Patentlösung für jede Anwendung. Die Beachtung einiger Richtlinien trägt jedoch zu einem guten Ergebnis bei:

1. Klar und eindeutig bleiben

Wenn Sie ein Entwicklungsteam damit beauftragen, eine Anwendung für eine To-do-Liste zu erstellen, erhalten Sie möglicherweise nicht das Ergebnis, das Sie erwarten. Zum Beispiel fehlen Informationen

über das Zielsystem, die gewünschte Funktionalität, die UX und vieles mehr. Präzision ist hier die Schlüsselzutat: Je genauer die Anforderungen sind, desto genauer können die Ergebnisse den Anforderungen entsprechen. Dies gilt umso mehr für ein Softwaretool wie GitHub Copilot. Formulieren Sie genau, was Sie wollen und lassen Sie irrelevante Details weg (Ausnahme siehe Punkt 2). Die Formulierung sollte außerdem klar verständlich und umsetzbar sein, das heißt möglichst keine Verneinungen, sondern klare Anweisungen und Anforderungen.

2. Der Kontext ist König(in)

Ein Prompt beginnt am besten wie eine User Story: Der Kontext wird erklärt. Die wichtigste Frage ist: Was soll der Code tun? Ein kanonisches Beispiel wäre eine Anwendung für eine To-do-Liste, also beginnt man den Prompt mit genau dieser Aussage. Dies ermöglicht es dem System einerseits, bereits existierende Lösungen in diesem Bereich zu berücksichtigen, und andererseits, ein Konzept für die zu erstellende Lösung zu definieren. Danach folgen die einzelnen zu implementierenden Funktionalitäten.

3. Schritt für Schritt zum Erfolg

Wie das Programmieren selbst ist auch die Codeunterstützung durch GitHub Copilot ein iterativer Prozess. Selten entspricht der erste Wurf bereits dem gewünschten und endgültigen Ergebnis. Es gibt immer Verbesserungspotential, entweder durch GitHub Copilot oder durch eigenes Eingreifen in den Code. Aus diesem Grund ist es sinnvoll, die einzelnen geforderten Aufgaben im Code einzeln zu spezifizieren. Nicht alles auf einmal, sondern in aufeinanderfolgenden Schritten. Zum Beispiel könnte bei der Erstellung einer API zuerst das Datenmodell generiert und dann die Endpunkte implementiert werden.

Natürlich ist es auch möglich, alles auf einmal zu machen, zum Beispiel durch Angabe des Kontexts (siehe Punkt 2) und dann einer Spiegelstrichliste mit den relevanten Features. In diesem Fall kann jedoch erst eingegriffen werden, wenn der gesamte Code steht. Bei einem iterativen Vorgehen können Korrekturen früher vorgenommen werden.

4. Mit Beispielen vorgehen

Was für einen selbst selbstverständlich erscheint, kann für andere offene Fragen aufwerfen. Beim Prompt Engineering ist es oft hilfreich, die zu erzeugende Ausgabe einer Funktionalität exemplarisch zu spezifizieren. Dabei wird auch die Datenstruktur so festgelegt, wie sie später verarbeitet werden soll.


```
JS copilot.js > ...
1 function getRandomDate() {
2     const start = new Date();
3     const end = new Date(new Date
4     return new Date(start.getTime
5 }
6
7 function getRandomWeather() {
8     const temperature = Math.floo
9     const date = getRandomDate();
10
11     return {
12         date: date.toISOString().
13         temperature: `${temperatu
14     };
15 }
16
```

Diese Vorgabe von Beispielen ist bei Aspekten der Code-Hygiene wichtig: Namensschemata für Bezeichner und ein gegebenenfalls vorhandener Code-Stil können von GitHub Copilot berücksichtigt werden. Am Ende sieht der Code dann (fast) so aus, als wenn er von Ihnen kommen würde.

5. Bitte und danke sagen

Nein, wirklich! Dabei geht es nicht um den in IT-Kreisen populären Cartoon, bei dem Roboter die Weltherrschaft übernehmen, aber dann eine Person verschonen, die bei der Kommunikation mit einem AI-System immer „bitte“ und „danke“ gesagt hat. Aber es gibt offenbar Anzeichen dafür, dass ein höflicher Umgang mit einem Computersystem sowohl die eigene Stimmung als auch den persönlichen Umgang verbessert haben. Und, seien wir ehrlich – richtig angewandt sparen wir den extra Tippaufwand für Höflichkeitsfloskeln doch durch die schnell generierten Ergebnisse wieder ein, oder?



Suad Wolgram

Suad arbeitet als Junior Software- und Cloud-Entwickler bei der white duck GmbH und studiert Informatik an der TH-Rosenheim.

Tipp 11:

Aussagekräftige Funktionsnamen & Variablen

von Suad Wolgram

Ein entscheidender Faktor für die effektive Nutzung von GitHub Copilot ist die Verwendung aussagekräftiger Funktionsnamen. Daher möchte ich Ihnen zeigen, wie Sie GitHub Copilot dabei unterstützen können, genauere Codevorschläge zu generieren, indem Sie bessere Funktionsnamen verwenden.

Betrachten wir das Szenario anhand des folgenden Codebeispiels. In der Funktion `fetch()` möchten wir alle Star Wars-Charaktere über die Star Wars API abrufen:

```
export class ApiService {
  private starWarsUrl = environment.baseUrl;

  constructor(private httpClient: HttpClient) {}

  fetch() {
  }
}
```

In diesem Zusammenhang wird deutlich, dass GitHub Copilot nicht versteht, welches Ziel wir mit der Methode `fetch()` verfolgen möchten. Daher können wir an dieser Stelle keine geeigneten Vorschläge erhalten. Eine Lösung besteht darin, `fetch()` durch einen aussagekräftigen Funktionsnamen zu ersetzen, der genau beschreibt, welche Aufgabe die Methode erfüllen soll. Sobald der Kontext klarer ist, liefert GitHub Copilot unverzüglich Vorschläge zur Erreichung unseres gewünschten Ziels:

```
getAllStarWarsCharacters() {  
    return this.httpClient.get(`${this.starWarsUrl}/people`);  
}
```

In diesem Fall hat GitHub Copilot sogar unsere Variable `starWarsUrl` erkannt und in der Lösung integriert.

Als zweites Beispiel nehmen wir an, dass wir die Anzahl der Bewohner eines Planeten im Star Wars-Universum berechnen möchten und die gewonnenen Informationen in einer Observable Map zurückgeben wollen.

Wie zuvor wählen wir einen aussagekräftigen Methodennamen und geben an, was wir genau zurückgeben möchten. Darüber hinaus definieren wir eine Variable mit dem Namen `planetCitizenCount`:

```
calculateCitizenCountForEachPlanet(): Observable<Map<string, number>> {  
    let planetCitizenCount = new Map<string, number>();  
  
    return this.getAllStarWarsCharacters().pipe(  
        map((characters: any) => {  
            characters.forEach((character: any) => {  
                let planetName = character.homeworld;  
                if (planetCitizenCount.has(planetName)) {  
                    planetCitizenCount.set(  
                        planetName,  
                        planetCitizenCount.get(planetName) + 1  
                    );  
                } else {  
                    planetCitizenCount.set(planetName, 1);  
                }  
            });  
            return planetCitizenCount;  
        })  
    );  
}
```

Sobald wir beginnen, die Logik einzufügen, erkennt GitHub Copilot aufgrund des gewählten Methodennamens und der passenden Variablennamen den Kontext und schlägt uns die entsprechende Implementierung vor. Wir können auch sehen, dass GitHub Copilot die Variable aufgrund des klaren Kontexts richtig verwendet.



Joël Zimmerli

Als Full-Stack-Softwareentwickler mit Erfahrung in verschiedenen Programmiersprachen wie Java, C# und TypeScript, legt Joël Zimmerli seinen Schwerpunkt auf die testgetriebene Entwicklung (TDD).

Tipp 12:

Unterstützung bei der Testfallerstellung mit GitHub Copilot

von Joël Zimmerli

Bei der testgetriebenen Entwicklung (TDD) ist das Schreiben von Unit Tests eine der wichtigsten, aber oft auch zeitaufwändigen Aufgaben. In diesem Artikel zeige ich, wie ich GitHub Copilot einsetze, um meine Tests effizient und umfassend zu schreiben, sodass mehr Zeit für die eigentliche Entwicklung bleibt.

Als konkretes Beispiel verwende ich eine Dotnet-Applikation, in der Immobilien mit ihren Verkaufspreisen gespeichert werden. Die Suchfunktion ermöglicht eine Filterung nach der Preisspanne. Um diesen Filter zu implementieren, habe ich eine Factory geschrieben, die auf Basis des minimalen und des maximalen Preises eine Vergleichsfunktion erstellt. Ein Ausschnitt dieser Funktion ist in der folgenden Abbildung dargestellt:

```
static Expression<Func<HouseDocument, bool>> PriceIsInRangeFilter(
    int? minPrice, int? maxPrice){
    var expressionBuilder = PredicateBuilder.New<HouseDocument>(true);
    if (minPrice.HasValue)
        expressionBuilder.And(PriceHigherThanFilter(minPrice.Value));
    if (maxPrice.HasValue)
        expressionBuilder.And(PriceLowerThanFilter(maxPrice.Value));

    return expressionBuilder;
}
```

Um GitHub Copilot für die automatische Testgenerierung zu verwenden, beginne ich mit der Erstellung des Testnamens. Dieser Name beschreibt, welches Objekt getestet wird, welche Funktion aufgerufen und welches Ergebnis erwartet wird. Diese Namensstruktur ist in der Welt der Tests weit verbreitet. Dies ermöglicht es GitHub Copilot, den gesamten Test automatisch zu generieren, wie diese Abbildung zeigt.

```
[Fact]
public void HouseQueryToExpression_ShouldEvaluateCorrectly() {
    // Arrange
    var query = new HouseQuery()
    {
        UpperPriceLimit = 100,
        LowerPriceLimit = 0
    };
    var house = new HouseDocument()
    {
        Id = "house",
        Name = "house",
        Price = 50
    };

    // Act
    var expression = query.ToExpression();
    var result = expression.Invoke(house);

    // Assert
    result.Should().BeTrue();
}
```

Dieser Test sollte dann mit verschiedenen Preisen für Immobilien getestet werden. Dazu kann der Test – wie in der folgenden Abbildung dargestellt – parametrisiert werden.

```
[Theory]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(int?
upperLimit, int? lowerLimit, int housePrice, bool expected) {
    // Arrange
    var query = new HouseQuery()
    {
        UpperPriceLimit = upperLimit,
        LowerPriceLimit = lowerLimit
    };
    var house = new HouseDocument()
    {
        Id = "house",
        Name = "house",
        Price = housePrice
    };

    // Act
    var expression = query.ToExpression();
    var result = expression.Invoke(house);

    // Assert
    result.Should().Be(expected);
}
```

GitHub Copilot kann auch die Prüfparameter generieren, wie in dieser Abbildung dargestellt.

```
[Theory]
[InlineData(100, 0, 50, true)]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(int?
upperLimit, int? lowerLimit, int housePrice, bool expected){
```

Durch mehrmaliges Generieren der Parameter erhält man eine umfangreiche Liste von Möglichkeiten, wie in der nächsten Abbildung dargestellt. Es ist jedoch ratsam, diese Liste sorgfältig zu prüfen und zu überdenken.

```
[Theory]
[InlineData(100, 0, 50, true)]
[InlineData(100, 0, 99, true)]
[InlineData(100, 0, 1, true)]
[InlineData(100, 0, 150, false)]
[InlineData(100, 50, 150, false)]
[InlineData(100, 150, 200, false)]
[InlineData(100, 0, 0, false)]
[InlineData(100, 100, 100, false)]
[InlineData(100, 100, 0, false)]
[InlineData(null, null, 0, true)]
[InlineData(100, null, 0, true)]
[InlineData(100, null, 101, false)]
[InlineData(null, 100, 101, true)]
[InlineData(null, 100, 99, false)]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(int?
upperLimit, int? lowerLimit, int housePrice, bool expected){
```

Mit diesen Schritten kann GitHub Copilot Sie bei der effizienten Erstellung von Tests in verschiedenen Szenarien unterstützen.



Tobias Deekens

Tobias Deekens ist ein ehemaliger Basketballspieler und lausiger Gitarrist, außerdem ein Entwickler, begeisterter Lehrer und spontaner Sprecher mit viel Erfahrung in der Frontend-Entwicklung und -Architektur. Er hat große Freude daran, als Mentor mit verschiedenen Teams in agilen Umgebungen zu arbeiten.

Tipp 13:

Wie wir drei Monate die Möglichkeiten von GitHub Copilot getestet haben

von Tobias Deekens

Im Februar des vergangenen Jahres kündigte GitHub die Einführung von [GitHub Copilot for Business](#) an. Diese Ankündigung weckte sofort unser Interesse, und viele Entwickler*innen in unserem Unternehmen wollten das Produkt ausprobieren. Nachdem wir eine Einführungsstrategie entwickelt hatten, beschlossen wir, GitHub Copilot drei Monate lang zu testen, um zu sehen, wie es unsere Produktivität verbessern kann. Dieser Artikel beschreibt unseren Weg der Evaluierung und Einführung von GitHub Copilot.

Bei commercetools nutzen wir eine Vielzahl von Programmiersprachen und Tools zur Entwicklung unserer Produkte – von Scala über TypeScript und PHP bis hin zu Go und Rust. Wir achten darauf, fundierte Technologieentscheidungen zu treffen, um die produktivsten Werkzeuge auszuwählen. Außerdem möchten wir unser kollaboratives Denken, das fest in unseren Unternehmenswerten verankert ist, auch beim Unternehmenswachstum beibehalten. Wir sehen einen großen Einfluss auf die Zusammenarbeit im Software-Engineering durch die Vielzahl von Tools, die auf Basis von generativer KI entwickelt wurden. Und wir freuen uns darauf, sie in unsere tägliche Routine einzubinden.

Warum zuerst evaluieren und nicht einfach einführen?

Sie fragen sich vielleicht, warum wir ein allseits beliebtes und erfolgreiches Produkt wie GitHub Copilot drei Monate lang getestet haben, anstatt es einfach für alle Entwickler*innen einzuführen. Das liegt daran, dass commercetools einen pragmatischen Ansatz bei der Einführung von KI verfolgt. Obwohl KI von vielen Seiten unterstützt wird, erfolgt die Nutzung und Bewertung bei uns von unten nach oben.

Wir möchten, dass unsere Teams selbst beurteilen und entscheiden, wie KI die Produktivität und Funktionalität verbessern kann. In diesem Fall hat die Engineering-Abteilung GitHub Copilot auf die gleiche Weise untersucht wie jedes andere neue Tool. Dabei haben wir diejenigen einbezogen, die das Tool am Ende tatsächlich nutzen würden, um ihre praktische Meinung einzuholen.

Die Fülle neuer und verbesserter Produkte, die auf Basis von generativer KI arbeiten, macht eine gründliche Evaluierung jedes einzelnen Tools und wie sie zusammenspielen immer wichtiger. Zum Beispiel stellt sich die Frage, ob es sinnvoll ist, Replit Ghostwriter, Codeium, CodeComplete und GitHub Copilot zusammen zu verwenden? Oder ob es besser wäre, eines dieser Tools durch Mintlify oder Wrap zu ergänzen. Diese Fragen lassen sich nur beantworten, indem man die Tools in der Praxis testet und sich nicht nur auf Marketing-Websites verlässt.

Um eine fundierte Einführung vornehmen zu können, wollten wir die erwartete und tatsächliche Auswirkung von GitHub Copilot in unserer gesamten technischen Organisation klar verstehen. Dies umfasste Frontend- & Backend-Entwickler*innen, Site Reliability Engineers, Test Automation Engineers und diejenigen, die an der Dokumentation arbeiten.

Wie wir GitHub Copilot evaluiert haben

Nachdem wir beschlossen hatten, dass wir eine kontrollierte Evaluierung durchführen werden, legten wir zunächst eine sinnvolle Dauer dafür fest: Drei Monate, die sich über zwei Quartale erstrecken, erschienen uns ideal. In dieser Zeit wollten wir eine umfassende Momentaufnahme des Entwicklungszyklus erhalten, einschließlich des Quartalsendes, an dem die Teams häufig neue Funktionen für unsere Produkte einführen.

Nachdem wir uns auf eine Laufzeit geeinigt hatten, brauchten wir eine Stichprobengröße. Bei 150 Entwickler*innen kann die Durchführung einer Evaluierung mit nur 5 bis 10 von ihnen leicht zu verzerrten Ergebnissen führen. Daher wollten wir 30 bis 35 Entwickler*innen in das Evaluierungsteam aufnehmen, was wiederum zu einer Beteiligung von 20 bis 25 Prozent führen sollte. Schließlich wollten wir so viele Disziplinen wie möglich einbeziehen, um eine heterogene Gruppe zu erhalten, die verschiedene Werkzeuge und Sprachen verwendet.

Wir waren bereit, unsere Pläne in einem internen Blogbeitrag zu veröffentlichen. Darin informierten wir über den Prozess und verlinkten ein Google-Formular zur Anmeldung. Nach einer Woche hatten sich 34 Personen aus dem gesamten Unternehmen angemeldet, was unserer gewünschten Stichproben-

größe entsprach. Glücklicherweise mussten wir unseren Pool von Testpersonen nicht anpassen. Alle Teilnehmer*innen wurden in eine E-Mail-Liste und einen Slack-Kanal aufgenommen, um Updates auszutauschen. Zudem wurden sie einem speziellen Team auf GitHub hinzugefügt, das ihnen Zugriff auf GitHub Copilot gewährte.

Nach all diesen Vorbereitungen gingen wir den Beteiligten aus dem Weg und ließen sie einfach ihre Arbeit machen und dabei GitHub Copilot nutzen. Erst nach einer Woche meldeten wir uns kurz, um sicherzustellen, dass jeder GitHub Copilot erfolgreich installiert und mit dem Editor seiner Wahl integriert hatte. In den kommenden Wochen teilten die Teilnehmer*innen ihre Eindrücke und Codebeispiele über Slack oder Pull Requests mit, während wir im Hintergrund an einer größeren Abschlussumfrage arbeiteten.

Während der gesamten Dauer unserer Evaluierung blieben wir im Hintergrund mit GitHub in Kontakt. Die Testgruppe teilte mit uns interessante Statistiken, wie z. B. die durchschnittliche Annahmquote des Codes. Außerdem gelang es uns, in den letzten beiden Wochen unserer Evaluierung GitHub Copilot Chat nutzen zu können, was uns einen Einblick in die Zukunft von GitHub Copilot als kollaborativeres System ermöglichte. Wir sind gespannt, wohin die Zukunft von GitHub Copilot geht und wo uns die verschiedenen Angebote hinführen.

Die Ergebnisse und Konsequenzen

Wir haben erwartet, dass sich GitHub Copilot ohne Probleme in die täglichen Arbeitsabläufe integrieren lässt und einfach zu bedienen ist. Wir hofften außerdem, dass sich die Vorschläge über alle Programmiersprachen hinweg als nützlich erweisen und niemandem bei der Arbeit behindern würden. Während der Evaluierung wurden diese Erwartungen erfüllt, stellten aber auch fest, dass es Raum für Verbesserungen gibt und die Qualität der Vorschläge je nach Art der Arbeit, die jemand durchführt, sehr unterschiedlich sein kann.

Im Einzelnen umfasste unsere abschließende Umfrage 15 Fragen, die sich auf drei Hauptbereiche konzentrierten:

1. Wurde GitHub Copilot durchgängig genutzt?
2. Macht uns GitHub Copilot produktiver?
3. Stellt GitHub Copilot keine größeren Risiken oder Nachteile für uns dar?

„Es schreibt Versionshinweise für mich! Das ist das Beste überhaupt!“

EIN MITARBEITER BEIM MORGENDLICHEN KAFFEE

Zu diesen drei zentralen Themen haben wir die Fragen vertieft, wie zum Beispiel:

1. Wie oft hast du GitHub Copilot während unserer Testphase verwendet?
2. Hat sich deine Nutzung von GitHub Copilot im Laufe der drei Monate verändert?
3. Wie oft musstest du die Vorschläge von GitHub Copilot anpassen?
4. Bei welchen Aufgaben hast du die größte Produktivitätssteigerung festgestellt?
5. Sollten wir in diesem Jahr andere Tools mit generativer KI evaluieren, um unsere Produktivität zu steigern?

„Es gab ein tägliches Ringen zwischen GitHub Copilot und dem regulären IntelliSense.“

EINE TESTPERSON, DIE EIN TÄGLICHES RINGEN BEOBACHTETE

Nachdem wir all diese Fragen gestellt hatten, waren dies die wichtigsten Erkenntnisse:

- 57% nutzten GitHub Copilot jeden Tag, alle anderen jeden zweiten Tag.
- 95% gaben an, dass GitHub Copilot sie produktiver macht.
- 63% gaben an, dass ihre Nutzung im Laufe der Zeit zugenommen hat.
- 67% gaben an, dass die Vorschläge hilfreich waren.
- 82% gaben an, dass die Vorschläge selten problematisch waren.
- 60% gaben an, dass GitHub Copilot als KI-Assistent für die Programmierung geeignet sei.
- 80% erwarten nicht, dass andere Tools wesentlich besser sind.
- 100% wollen GitHub Copilot weiterhin nutzen.

„Manchmal scheint GitHub Copilot zu schlafen, wenn viele VS Code-Fenster geöffnet sind. Anschließend wird man mit 50 Codezeilen zugeschüttet.“

EINE TESTPERSON, DIE VON SCHNELLEN VORSCHLÄGEN ÜBERRANNT WURDE

Zusätzlich zu diesen Zahlen konnten wir auch mehr qualitative Erkenntnisse darüber sammeln, in welchen Bereichen GitHub Copilot glänzt und in welchen es nicht beeindrucken konnte:

- Erfolgreich beim Schreiben von Tests (72%)
- Hilft beim Refactoring von Code (42%)
- Glänzt bei der automatischen Vervollständigung, Boilerplate und Scaffolding (~60%)
- Schwierigkeiten mit komplexer Geschäftslogik (82%)
- Ist nicht besonders leistungsstark, wenn der Code-Kontext wichtig ist (43%)
- Sollte bei leistungs- oder sicherheitsrelevanten Themen vorsichtig betrachtet werden (27%)
- Ist nicht hilfreich bei hochspezialisierten oder modernen Frameworks (14%)

„GitHub Copilot ist intelligent genug, um zuweilen auch gefährlich zu sein.“

EINE TESTPERSON, NACHDEM GITHUB COPILOT VORGESCHLAGEN HATTE,
40.000 ENTITÄTEN EINZELN AUS EINER DATENBANK ZU LADEN

Als wir GitHub Copilot über einen längeren Zeitraum evaluiert hatten, konnten wir auch Bereiche identifizieren, in denen Verbesserungen möglich waren:

- Man kann noch kein Feedback zu einem Vorschlag geben.
- Es kann nicht so konfiguriert werden, dass es in bestimmten Ordnern oder Situationen nicht verwendet werden kann.
- Es funktioniert nicht sehr gut über Dateigrenzen hinweg.
- Homogenes Refactoring über eine größere Codebasis funktioniert nicht gut.

Das sind viele Zahlen, aber sie haben uns geholfen, den Nutzen von GitHub Copilot in unserem Unternehmen zu verstehen und die Entscheidung zu treffen, das Tool einzuführen. Nach der Aktivierung wurde GitHub Copilot kontinuierlich genutzt, und die Nutzung nahm sogar noch weiter zu. Die Vorschläge wurden häufig angenommen und waren von guter Qualität. Die Kolleg*innen konnten das Tool problemlos in ihre bestehenden Arbeitsumgebungen einbinden und erzielten dadurch enorme Produktivitätsgewinne. Das bedeutet für uns, dass wir das System in der nächsten Zeit in unserem Unternehmen weiter ausbauen werden.

Dieser Artikel wurde ursprünglich auf <https://commercetools.com/blog/how-we-evaluated-the-impact-of-github-copilot-for-3-months> veröffentlicht. Vielen Dank an Tobias und commercetools für die Erlaubnis, ihn in diesem E-Book erneut zu veröffentlichen.



Dennis Gassen

Dennis Gassen ist Go to Market Manager für den Bereich Digital & Application Innovation bei Microsoft Deutschland. Wenn er den Kopf nicht tief im Business vergraben hat (was auch immer das bedeuten soll), dann unterstützt er die Developer Community. In der Freizeit findet man ihn am ehesten auf dem Rennrad irgendwo im Münchner Süden.

Tipp 14:

GitHub Copilot als KI-Assistent für barrierefreie Software

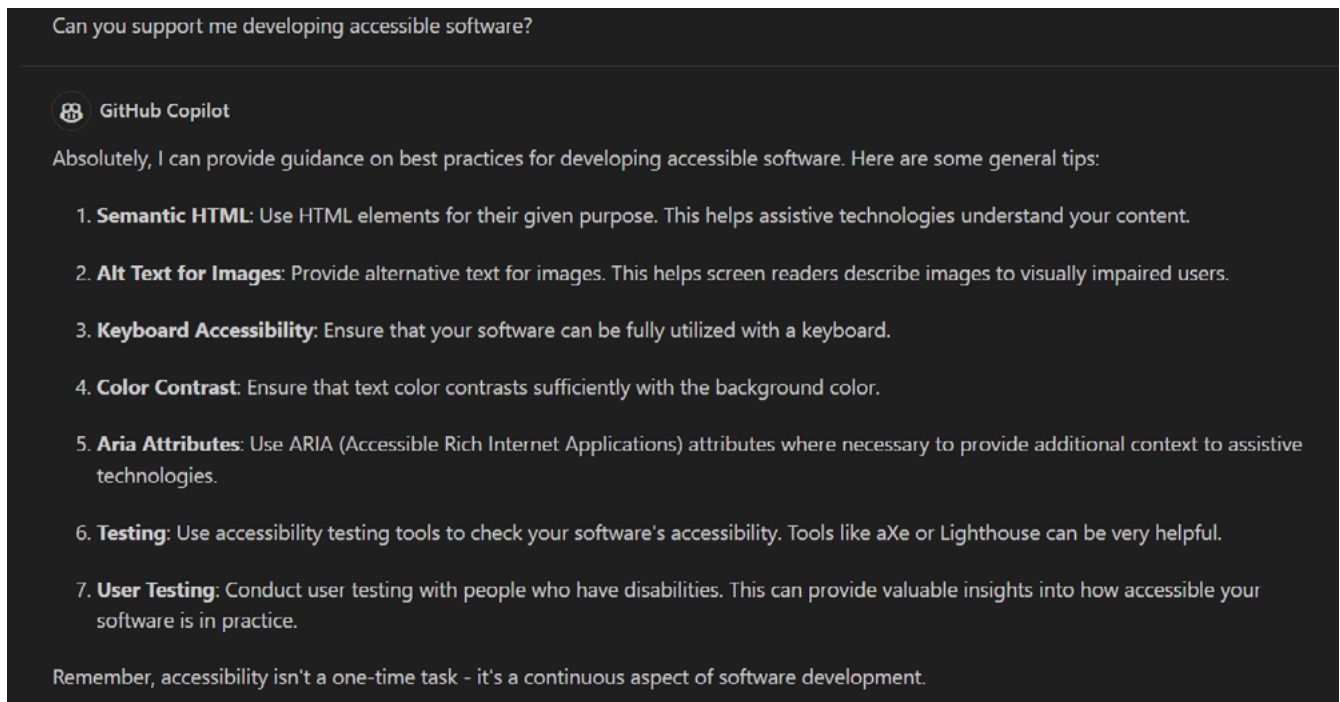
von *Dennis Gassen*

Dieser Artikel bietet keine umfassende Betrachtung, wie barrierefreie Software entwickelt werden kann oder sollte – dafür ist der Rahmen hier zu eng gesteckt. Vielmehr möchte ich zeigen, dass mithilfe von GitHub Copilot bereits einige grundlegende Standards berücksichtigt und eingehalten werden können, die vielen Menschen mit Behinderungen einen besseren Zugang und eine bessere Nutzbarkeit von softwarebasierten Produkten ermöglichen.

Weltweit leben etwa 1,3 Milliarden Menschen mit einer Behinderung. Das sind etwa 16 Prozent der Weltbevölkerung. Ein Großteil davon lebt in einkommenschwachen Ländern mit schlechter Gesundheitsversorgung sowie schlechter Schulbildung und ist ärmer als viele Menschen ohne Behinderung. Gerade für diese Gruppe ist der Zugang zu Informationsangeboten essenziell wichtig.

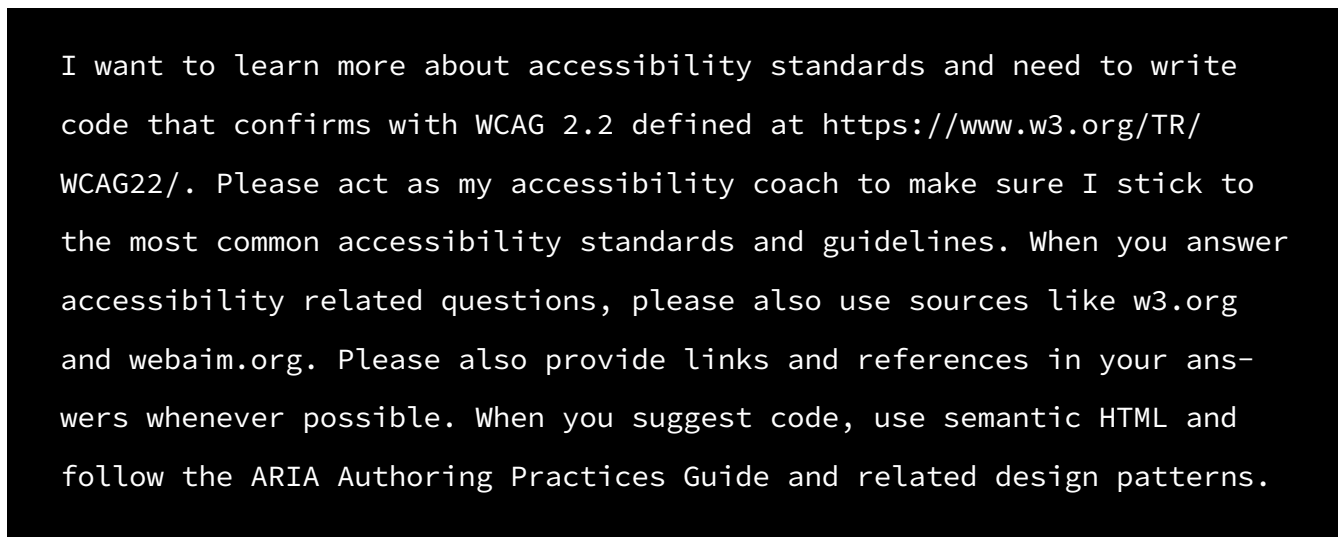
Die [Web Content Accessibility Guidelines](#) (WCAG) sind der internationale Standard für barrierefreie Internetangebote. In der Europäischen Union sind sie für öffentliche Einrichtungen seit September 2019 für neue Webseiten, seit 23. September 2020 für bestehende Webseiten und seit 23. Juni 2021 für mobile Anwendungen verbindlich. Das World Wide Web Consortium (W3C) veröffentlichte die WCAG erstmals 1999 als Version 1.0. Seit Dezember 2008 ist die Version 2.0 als [ISO-Standard](#) anerkannt. Aktuell wird die Version 2.2 vom W3C empfohlen. Obwohl die W3C-Empfehlungen keine allgemeine gesetzliche Gültigkeit besitzen, haben sie dennoch eine hohe Verbindlichkeit für die Entwicklung barrierefreier Webseiten und Anwendungen.

Wenn wir nun GitHub Copilot Chat nutzen und die einfache Frage stellen, wie GitHub Copilot uns bei der Entwicklung barrierefreier Applikationen unterstützen kann, erhält man folgende Antwort:



Diese ersten Empfehlungen sind nicht falsch, sollten aber unbedingt als „generelle Tipps“ verstanden werden. Auf Basis von Prompt Engineering haben wir aber die Möglichkeit, GitHub Copilot mehr Kontext zu unseren konkreten Anforderungen zu geben, um detailliertere Antworten zu erhalten.

Dafür lässt sich der folgende Prompt verwenden:



Die Antwort von GitHub Copilot sieht dann folgendermaßen aus:

Absolutely, I can guide you through the process of developing software that adheres to WCAG 2.2 standards and follows the ARIA Authoring Practices Guide. Here are some key principles and coding practices to keep in mind:

1. **Perceivable:** Information and user interface components must be presentable to users in ways they can perceive. This means that users must be able to perceive the information being presented. For example, providing alt text for images, transcripts for audio, and captions for videos.

```

```

2. **Operable:** User interface components and navigation must be operable. This means that users must be able to operate the interface. For example, ensuring that all functionality is available from a keyboard.

```
<button onclick="myFunction()">Click me</button>
```

3. **Understandable:** Information and the operation of user interface must be understandable. This means that users must be able to understand the information as well as the operation of the user interface. For example, using clear labels for form inputs.

```
<label for="fname">First name:</label><br>
<input type="text" id="fname" name="fname"><br>
```

4. **Robust:** Content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies. This means that users must be able to access the content as technologies advance. For example, using semantic HTML elements like `<nav>`, `<header>`, `<main>`, and `<footer>` to structure your content.

```
<header>
  <h1>Website Title</h1>
</header>

<nav>
  <a href="#content1">Content 1</a>
  <a href="#content2">Content 2</a>
</nav>

<main id="content1">
  <h2>Content 1</h2>
  <p>This is some content.</p>
</main>

<footer>
  <p>Copyright 2022</p>
</footer>
```

You can find more detailed information on each principle, guideline, and success criteria in the [WCAG 2.2 documentation](#).

For more practical advice on implementing these standards, [WebAIM](#) is a great resource. They provide a wide range of articles, tutorials, and checklists to help you make your software more accessible.

The [ARIA Authoring Practices Guide](#) is a great resource for understanding how to use ARIA to improve accessibility, especially for complex widgets and interaction patterns.

Remember, accessibility is not just about adhering to standards - it's about ensuring that as many people as possible can use and benefit from your software.

Der Prompt dient als Grundlage und enthält folgende Punkte:

- Erweiterte Kontextinformationen, um Anfragen zu beantworten.
- Definition, welche Rolle GitHub Copilot hat und welches Feedback wir erwarten.
- Hinweis auf seriöse Quellen, die zur Beantwortung unserer Fragen genutzt werden sollen.
- Hinweis zu weiteren Ressourcen und Quellen

Wir können Github Copilot aber auch nach konkreten Beispielen fragen, bei denen Accessibility-Standards nicht eingehalten werden:

Please create a sample website that is not conform with the WCAG 2.2 standards and illustrate on this website what are the most common mistakes when it comes to accessibility.

GitHub Copilot

Sure, here's a simple example of a website that does not conform to WCAG 2.2 standards:

```
<!DOCTYPE html>
<html>
<head>
  <title>Non-accessible Website</title>
  <style>
    .low-contrast {
      color: #888;
      background-color: #aaa;
    }
  </style>
</head>
<body>
  
  <div class="low-contrast">This is a low contrast text.</div>
  <a href="#">Click here</a>
  <input type="text">
  <button onclick="alert('Button clicked!')">Submit</button>
</body>
</html>
```

Here are the most common mistakes in this website:

1. **Missing alternative text:** The image tag `` does not have an `alt` attribute. Screen readers use the `alt` attribute to describe the image to visually impaired users.
2. **Low contrast:** The text inside the `div` with class `low-contrast` has low contrast, making it hard for visually impaired users to read.
3. **Non-descriptive link text:** The link text "Click here" does not provide context about what the link does or where it goes.
4. **Missing form labels:** The input field `<input type="text">` does not have a corresponding label, making it hard for screen reader users to understand its purpose.
5. **Non-accessible JavaScript:** The button uses an `onclick` event handler, which is not accessible to keyboard-only users. It should use a more accessible event like `onkeypress`.

Remember, these are just a few examples of common accessibility mistakes. There are many other aspects of accessibility to consider when developing a website.

Noch besser ist, dass GitHub Copilot auch gleich die passenden Lösungen vorschlägt:

```
Here's how you can fix these issues:

<!DOCTYPE html>
<html>
<head>
  <title>Accessible Website</title>
  <style>
    .high-contrast {
      color: #333;
      background-color: #fff;
    }
  </style>
</head>
<body>
  
  <div class="high-contrast">This is a high contrast text.</div>
  <a href="#">More information</a>
  <label for="inputField">Input Field</label>
  <input id="inputField" type="text">
  <button type="button" onclick="alert('Button clicked!')>Submit</button>
</body>
</html>
```

Natürlich sind die hier gezeigten Beispiele sehr einfach gehalten. Als Illustration verdeutliche sie jedoch sehr gut, welche Möglichkeiten GitHub Copilot bietet, um sich an Accessibility Standards zu halten, um die eigene Software möglichst vielen Menschen zugänglich zu machen.

Hier einige weitere Prompts, die helfen können:

Prompt: What are best practices to incorporate accessibility testing in the whole application lifecycle?

Antwort von GitHub Copilot

Incorporating accessibility testing throughout the application lifecycle is crucial for creating and maintaining accessible software. Here are some best practices:

1. Integrate Accessibility from the Start: Accessibility should be a consideration from the very beginning of the project, not an afterthought. This includes during the design phase, where color contrast, font sizes, and navigation should be considered.
2. Automated Testing: Use automated testing tools like axe, Lighthouse, or WAVE to catch common accessibility issues. These tools can be

integrated into your CI/CD pipeline to ensure accessibility is checked with every build.

3. **Manual Testing:** Automated testing can't catch everything. Manual testing, including using the software with a screen reader, navigating with a keyboard only, and testing with real users with disabilities, is crucial.

4. **Incorporate Accessibility in Code Reviews:** Make accessibility a part of your code review process. This can help catch potential issues before they become a part of the codebase.

5. **Training and Awareness:** Ensure that all members of your team understand the importance of accessibility and how to implement it. This includes developers, designers, content creators, and testers.

6. **Regular Audits:** Regularly audit your application for accessibility. This can help catch any new issues that may have been introduced.

7. **Use WCAG as a Guide:** The Web Content Accessibility Guidelines (WCAG) provide a wide range of recommendations for making Web content more accessible. Use these guidelines as a reference throughout your project.

Remember, accessibility is not a one-time task, but a continuous effort throughout the entire application lifecycle.

For more detailed information, you can refer to the WCAG guidelines and WebAIM, which provides a wide range of resources to help you make your software more accessible.

Prompt: How can I test if my application is accessible for a screen reader?

Antwort von GitHub Copilot

Testing your application for screen reader accessibility involves both automated and manual testing methods. Here are some steps you can follow:

1. **Automated Testing:** Use automated testing tools like axe, Lighthouse, or WAVE to catch common accessibility issues. These tools can help identify issues like missing alt text, incorrect semantic HTML, and missing form labels.

2. **Manual Testing with a Screen Reader:** To truly understand the experience of a screen reader user, you should test your application with a screen reader. There are several screen readers available, including NVDA and JAWS for Windows, VoiceOver for macOS and iOS, and Talk-Back for Android. As you navigate your application with the screen reader, pay attention to whether all content and functionality is accessible and understandable.

3. **Keyboard Navigation:** Screen reader users often navigate using the keyboard, so ensure that all interactive elements are keyboard accessible (can be focused and activated using the keyboard) and that the tab order is logical.

4. **Semantic HTML:** Screen readers rely on semantic HTML to understand the content and structure of the webpage. Ensure that you're using the correct HTML elements for their intended purpose and that you're using ARIA roles and properties where appropriate.

5. **Descriptive Text:** Ensure that all images have alt text, that form inputs have associated labels, and that links and buttons have descriptive text.

Remember, while automated testing can catch many issues, there's no substitute for manual testing with a screen reader. It's also a good idea to involve users with disabilities in your testing process, as they can provide valuable insights based on their lived experiences. For more detailed information, you can refer to the WCAG guidelines

and WebAIM, which provides a wide range of resources to help you make your software more accessible.

Prompt: What are best practices for creating accessible tables in HTML?

Antwort von GitHub Copilot

Creating accessible tables in HTML involves ensuring that the table structure is clear and understandable, both visually and for assistive technologies. Here are some best practices:

1. **Use <th> for Headers:** Use the <th> element for table headers, and the <td> element for table data. This helps screen readers understand the structure of the table.

```
<table>
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
  </tr>
  <tr>
    <td>Data 1</td>
    <td>Data 2</td>
  </tr>
</table>
```

2. **Use scope Attribute:** The scope attribute can be used on <th> elements to specify whether it's a header for a column (scope="col") or a row (scope="row"). This helps screen readers understand the relationship between headers and cells.

```
<table>
  <tr>
    <th scope="col">Header 1</th>
    <th scope="col">Header 2</th>
  </tr>
  <tr>
    <th scope="row">Row Header</th>
    <td>Data</td>
  </tr>
</table>
```

3. **Use `<caption>` for Table Description:** The `<caption>` element provides a description of the table. This can be useful for providing context to screen reader users.

```
<table>
  <caption>A description of the table</caption>
  <!-- rest of table -->
</table>
```

4. **Use `<thead>`, `<tbody>`, and `<tfoot>`:** These elements can be used to group the header, body, and footer of the table. This can make the table more understandable for screen reader users.

```
<table>
  <thead>
    <!-- header rows -->
  </thead>
  <tbody>
    <!-- body rows -->
  </tbody>
  <tfoot>
    <!-- footer rows -->
  </tfoot>
</table>
```

5. **Avoid Complex Layouts:** Complex table layouts can be confusing for screen reader users. If possible, stick to simple layouts with clear headers for each row and column.

6. **Ensure High Contrast:** Ensure that the text and background colors of your table have sufficient contrast to be easily readable.

For more detailed information, you can refer to the WCAG guidelines and WebAIM, which provides a wide range of resources to help you make your tables more accessible.

Zusammenfassung

Mithilfe von Prompt Engineering haben Sie die Möglichkeit, die Ergebnisse von GitHub Copilot Chat in Bezug auf Barrierefreiheit deutlich zu verbessern. Natürlich sollte niemand perfekte Ergebnisse erwarten, schließlich handelt es sich „nur“ um einen „Co-Pilot“. Dennoch können Sie mit GitHub Copilot Chat sehr gut das eigene Wissen und das Verständnis für Barrierefreiheit ausbauen. Denken Sie auch immer daran, dass Sie GitHub Copilot auch zu den Antworten weitere Fragen stellen oder um zusätzliche Referenzen und Beispiele bitten können. Die Überprüfung der Vorschläge ist immer Aufgabe der Entwickler*innen.



Syrine Chelly

Syrine Chelly ist Cloud Solution Architect Developer Advocate bei Microsoft, wo sie seit mehr als 3 Jahren in der Intelligent Cloud Customer Success Unit arbeitet. Zuvor war sie in der Automobilindustrie in den Bereichen Robotik, Automatisierung und KI tätig.



Tobias Wittenburg

Tobias Wittenburg ist ein Cloud Solution Architect Developer Advocate bei Microsoft. Er ist seit 2005 in der Softwarebranche tätig und hat Erfahrung in der Entwicklung von Software in großem Maßstab mit verschiedenen Technologien.

Tipp 15:

Von der Anforderung zur Komponente

von Syrine Chelly und Tobias Wittenburg

Seit der Einführung von Cobol in den 50er Jahren träumt die Softwareindustrie davon, die Programmierung in natürlicher Sprache zu ermöglichen. Englisch sollte die Programmiersprache sein und daraus sollten Geschäftsanwendungen entstehen. Jetzt sind wir langsam dabei, diesen Traum Wirklichkeit werden zu lassen. Mit LLMs (Large Language Models) verstehen Computer jetzt auch natürliche Sprache, nicht nur Englisch. Mit GitHub Copilot hat man als Entwickler*in diese Möglichkeiten direkt in der Entwicklungsumgebung, um Code zu schreiben.

Wenn Sie GitHub Copilot und GitHub Copilot Chat zum ersten Mal verwenden, werden Sie es in der Regel dazu verwenden, Inline-Code aus Kommentaren zu generieren und Fragen zu Code, Sprachen, Prozessen oder Frameworks in GitHub Copilot Chat zu stellen.

Hier ist ein Beispiel in C# mit einem Vorschlag in Grau.

```
//create a method to calculate the simple interest
```

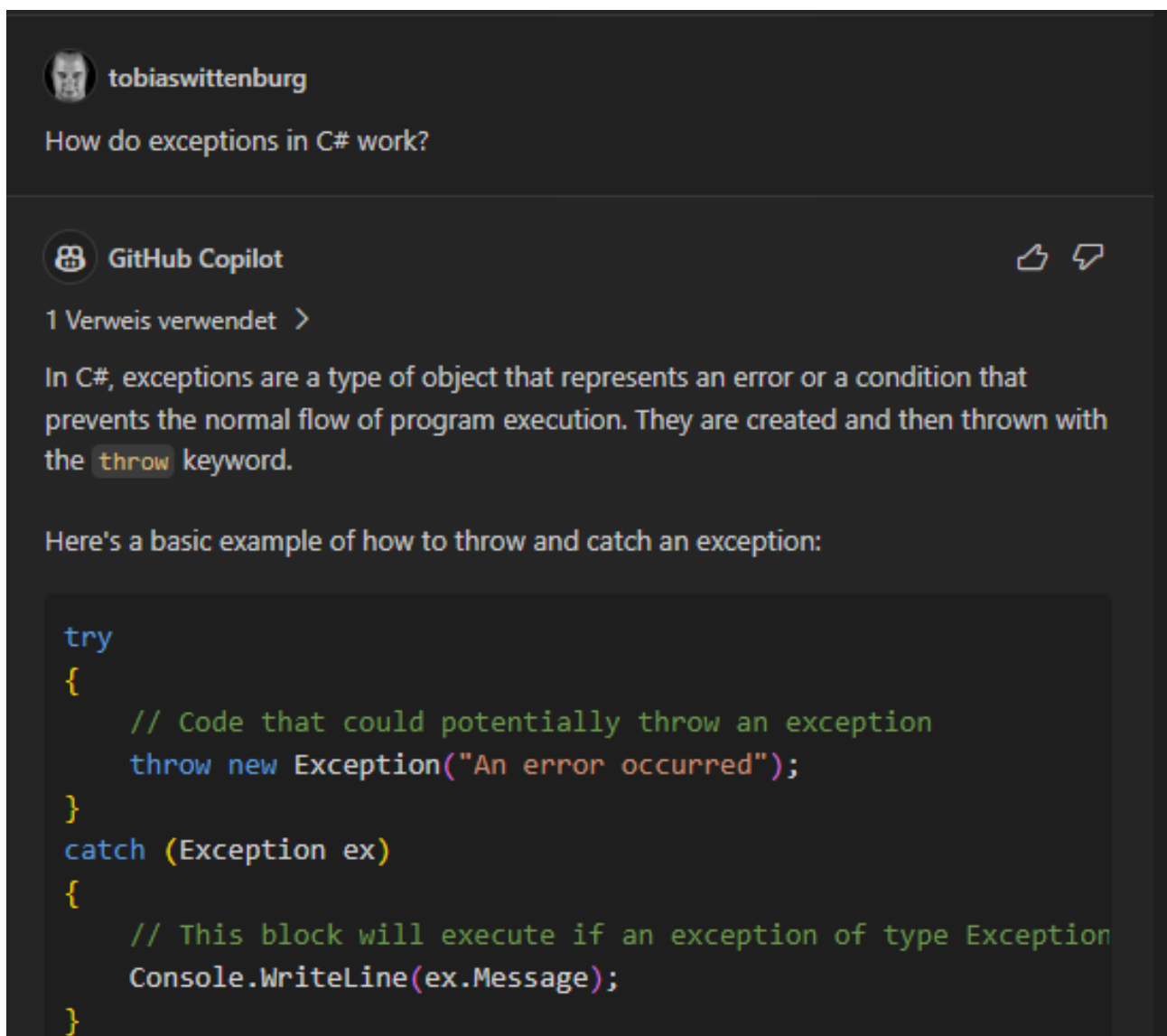
```
public double Calculate(double principal, double rate, double time)
{
    if (principal <= 0)
    {
        throw new ArgumentException("Principal must be greater than zero");
    }

    if (rate <= 0)
    {
        throw new ArgumentException("Rate must be greater than zero");
    }

    if (time <= 0)
    {
        throw new ArgumentException("Time must be greater than zero");
    }



    Result = principal * rate * time / 100;
    return Result;
}
```

Hier ein Beispiel dafür, wie Sie GitHub Copilot Chat zu Beginn verwenden würden.



tobiaswittenburg

How do exceptions in C# work?

GitHub Copilot  

1 Verweis verwendet >

In C#, exceptions are a type of object that represents an error or a condition that prevents the normal flow of program execution. They are created and then thrown with the `throw` keyword.

Here's a basic example of how to throw and catch an exception:

```
try
{
    // Code that could potentially throw an exception
    throw new Exception("An error occurred");
}
catch (Exception ex)
{
    // This block will execute if an exception of type Exception
    Console.WriteLine(ex.Message);
}
```

```
finally
{
    // This block will execute regardless of whether an exception
    // It's often used for cleanup code
}
```

In this example, the `try` block contains code that might throw an exception. The `catch` block contains code that will execute if an exception is thrown. The type of the exception caught can be specified as a parameter to the `catch` block. In this case, it's catching `Exception`, which is the base class for all exceptions in C#. The `finally` block contains code that will execute regardless of whether an exception was thrown.

You can also define your own custom exceptions by creating a new class that inherits from `Exception` or one of its subclasses.

GitHub Copilot berücksichtigt immer den aktuellen Kontext. Dazu gehören die aktuell geöffnete Datei, Metadaten (z.B. Programmiersprache) sowie die benachbarten Tabs.

Sie können jedoch auch einen Kontext hinzufügen, indem Sie dem Prompt weitere Informationen hinzufügen. Eine Möglichkeit wäre, GitHub Copilot Ihre Absicht mitzuteilen und Angaben darüber zu machen, was Sie erreichen möchten.

Hier ein Beispiel für einen Prompt für GitHub Copilot:

```
I am going to build a website for trainers that can be used during IT Workshops. This website should have a couple of smaller tools that you can pull up in between the sessions.
```

```
The website is being built in React with JavaScript.
```

```
The first component that I want to build is a timer for a coffee break.
```

```
Please help me building that timer. Here are the features:
```

- Customizable duration for a break
- Some buttons for a default break duration (5 min, 10 min, 15 min, 45 min, 1 hour)

- A Button "Timer Start"
- Upon clicking "Timer Start" the timer should start and count down.
- When the timer is up an alarm should go off.

Can you build that component for me?

In diesem Fall teilen wir GitHub Copilot mit, was wir tun wollen („Erstellen einer Website für Trainer*innen, die bei IT-Workshops eingesetzt werden soll“) und welche Technologie wir verwenden („Diese Website wird in React mit JavaScript erstellt“). Danach übermitteln wir GitHub Copilot unsere Anforderungen und erhalten eine voll funktionsfähige Komponente in React, die wir zu unserer Seite hinzufügen können.

In der Komponente könnte jedoch eine Funktion fehlen. Die Umstellung der Schaltfläche auf eine Stopp-Schaltfläche, wenn der Timer läuft und umgekehrt.

Mit einem zusätzlichen Prompt können wir GitHub Copilot um diese Implementierung bitten:

```
Please toggle the button to "Timer Stop" after the timer has started.  
Also change the JavaScript code to stop the timer once the "timer Stop"  
Button has been pushed.
```

Eine weitere leistungsstarke Möglichkeit, GitHub Copilot zu nutzen, besteht darin, die gewünschte Funktionalität direkt in den Kommentaren Ihres Codes zu beschreiben. Mit diesem Ansatz können Sie detaillierte Anweisungen in die Codebasis selbst einbetten, was es für GitHub Copilot einfacher macht, die entsprechenden Codeschnipsel zu generieren.

Hier ein Beispiel:

```
1 // Create a Timer component in React
2 // Features:
3 // - Customizable duration for a break
4 // - Default break durations: 5 min, 10 min, 15 min, 45 min, 1 hour
5 // - "Timer Start" button to initiate countdown
6 // - Countdown triggers an alarm when time is up
7 // - Toggle button to "Timer Stop" when countdown starts and vice versa
8
9 import React, { useState, useEffect } from 'react';
10 ✨
11 const TimerComponent = () => {
  const [duration, setDuration] = useState(0);
  const [timer, setTimer] = useState(null);
  const [isRunning, setIsRunning] = useState(false);

  const startTimer = () => {
    if (duration > 0) {
      setTimer(setInterval(() => {
        setDuration(prevDuration => prevDuration - 1);
      }, 1000));
      setIsRunning(true);
    }
  };

  const stopTimer = () => {
    clearInterval(timer);
    setIsRunning(false);
  };

  useEffect(() => {
    if (duration === 0) {
      clearInterval(timer);
      setIsRunning(false);
      alert('Time is up!');
    }
  }, [duration]);

  return (
    <div>
```

In diesem Beispiel werden die Kommentare verwendet, um die gesamte Funktionalität der Timer-Komponente zu beschreiben. Wenn GitHub Copilot diese Kommentare liest, kann es eine vollständige Implementierung erstellen, die den beschriebenen Funktionen entspricht.

Neben der Codegenerierung kann GitHub Copilot auch dazu verwendet werden, Kommentare für die Dokumentation zu erstellen und Erklärungen zur Verwendung von Bibliotheken zu liefern. Dies kann besonders hilfreich für das Verständnis von komplexem Code oder unbekanntem Bibliotheken sein. Im folgenden Beispiel geben wir GitHub Copilot den Auftrag, die Funktion zu beschreiben, indem wir die Beschreibung starten und GitHub Copilot seine Arbeit machen lassen.

```
8
9  import React, { useState, useEffect } from 'react';
10 // here is a description of how the TimerComponent works:
11 // - The TimerComponent is a functional component that uses the useState and useEffect hooks to manage the state of the timer.
12 const TimerComponent = () => {
13   const [duration, setDuration] = useState(0);
14   const [timer, setTimer] = useState(null);
15   const [isRunning, setIsRunning] = useState(false);
16
17   const startTimer = () => {
18     if (duration > 0) {
```

Durch die Verwendung von Kommentaren können Sie GitHub Copilot nicht nur zur Codegenerierung, sondern auch zur Erstellung umfassender Dokumentationen und Erläuterungen auffordern. Dies verbessert die Lesbarkeit des Codes, die Wartung und hilft dabei, verschiedene Bibliotheken besser zu verstehen und zu nutzen.



Maxim Salnikov

Maxim Salnikov lebt in Oslo und ist ein erfahrener Webentwickler und Tech-Enthusiast. Seine Erfahrung mit Webplattformen, Cloud Computing und KI von über zwei Jahrzehnten teilt er weltweit auf Developer-Events. Tagsüber unterstützt er als Leiter des Geschäftsbereichs Developer Productivity europäische Unternehmen bei der Entwicklung von Cloud- und KI-Lösungen, während er sich in seiner Freizeit in Norwegens größter Web- und Cloud-Developer-Community engagiert. Maxim experimentiert leidenschaftlich mit Generative AI, insbesondere KI-Entwicklungstools.

Tipp 16:

Einsatz einer Chat-Syntax: Chat-Teilnehmer, Chat-Variablen, Slash-Befehle

von Maxim Salnikov

GitHub Copilot Chat ist eine äußerst nützliche Funktion, die es Ihnen ermöglicht, direkt mit oder über Ihren Code zu chatten. Obwohl der Chat vollständig in natürlicher Sprache funktioniert (Sie müssen z. B. keine bestimmte Syntax verwenden), können Sie durch spezielle Chat-Funktionen neue, KI-gestützte Entwicklungsszenarien ermöglichen und Ihre Produktivität erheblich steigern.

Dazu gehören Chat-Teilnehmer, Slash-Befehle und Kontextvariablen, die durch eine spezielle Syntax aktiviert werden. Sie sollten beachten, dass diese hilfreichen Funktionen in erster Linie unter VS Code verfügbar sind und in anderen IDEs, die GitHub Copilot Chat unterstützen, nicht komplett nutzbar sein könnten.

Richten Sie Fragen oder Anliegen an einen der verfügbaren Chat-Teilnehmer

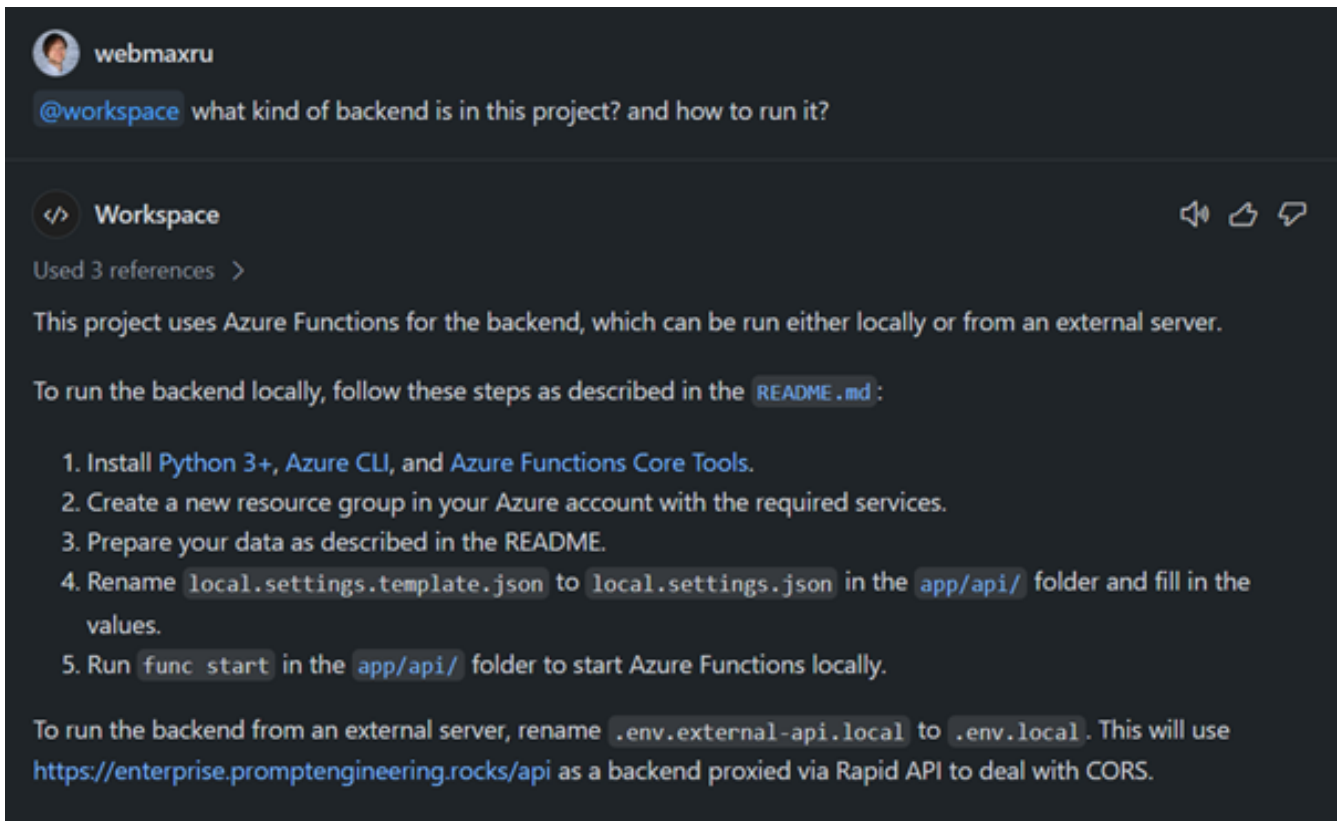
Bei GitHub Copilot Chat können Sie einen der KI-basierten „Domain-Experten“ über die herkömmliche Chatsyntax ansprechen, indem Sie dem entsprechenden Teilnehmernamen ein @ voranstellen.

Die derzeit verfügbaren Chat-Teilnehmer sind:

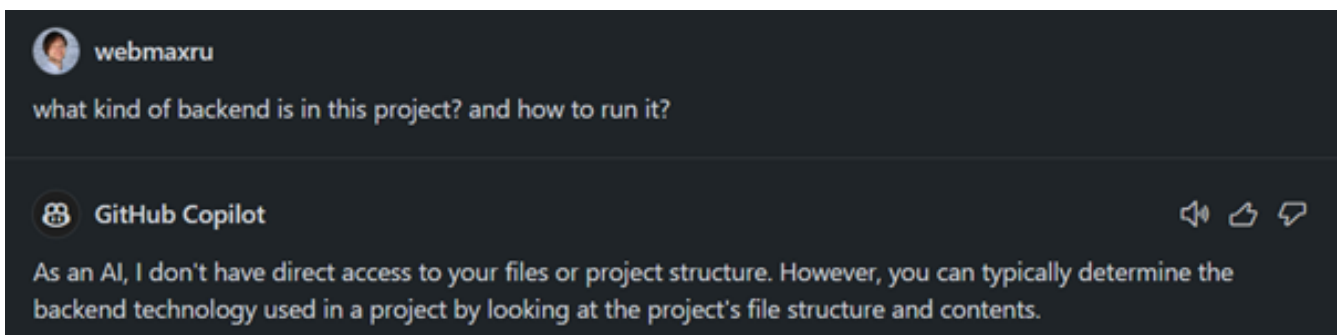
- **@workspace:** Dieser Chat-Teilnehmer, mit dem Sie wahrscheinlich häufig kommunizieren werden, weiß alles über den Code in Ihrem derzeit geöffneten Arbeitsbereich.
- **@terminal:** Dieser Chat-Teilnehmer weiß alles über die integrierte Terminal-Shell, ihren Inhalt und ihren Buffer.

- **@vscode:** Dieser Chat-Teilnehmer kennt den Editor von VS Code, seine Befehle und Funktionen.

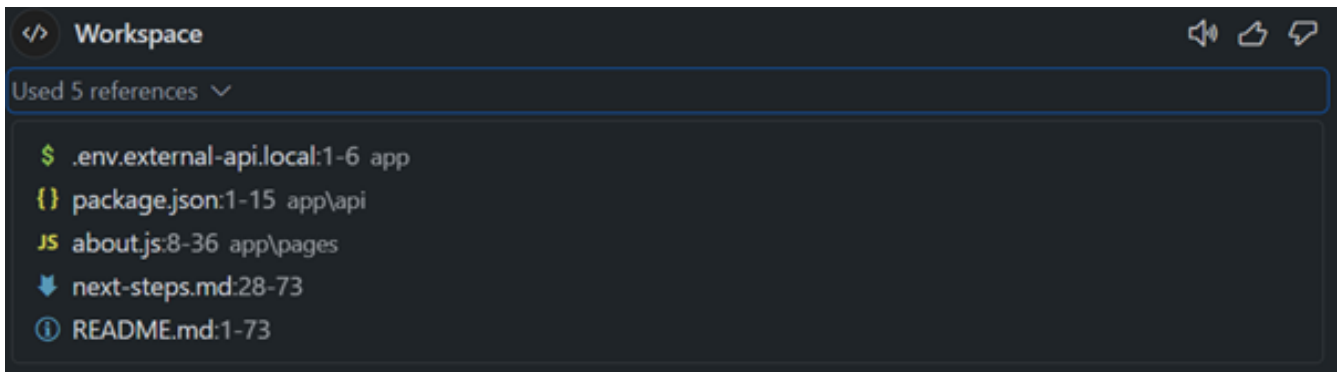
Ein Beispiel: Lassen Sie uns Informationen über das Backend des Projekts einholen, dem wir gerade zugewiesen wurden, indem wir @workspace direkt nach dem Öffnen des Projektordners in VS Code fragen.



In diesem speziellen Fall müssen Sie nicht einmal Dateien in Ihrem Editor geöffnet haben. Vergleichen Sie dies mit der Antwort, die Sie erhalten, wenn Sie @workspace nicht taggen:



Mithilfe von @workspace können Sie lösungsweite Abfragen durchführen, bei denen der gesamte Code für die Chat-Antwort berücksichtigt werden soll. Das bedeutet allerdings nicht, dass der gesamte Code verwendet und als Teil der Anfrage gesendet wird. Die GitHub Copilot Chat-Erweiterung in VS Code tut ihr Bestes, um relevante Dateien und Teile dieser Dateien zu bestimmen, indem sie zunächst lokales Wissen und Intelligenz nutzt. Sie können nachsehen, welche Dateien und Codezeilen für den Prompt verwendet wurden, indem Sie die Zeile „Used references“ erweitern:



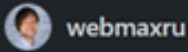
Produktivitätstipp: Verwenden Sie nach der Eingabe Ihrer Nachricht Strg + Enter (Cmd + Enter) statt nur Enter, und die @workspace-Zeichenfolge wird vor dem Senden automatisch in Ihre Nachricht eingefügt.

Verwenden Sie Chat-Variablen, um den Kontext genau festzulegen

In vielen Fällen ist es übertrieben, die gesamte Lösung als Kontext für eine Frage oder Anfrage zu berücksichtigen (indem Sie @workspace verwenden). Möglicherweise möchten Sie in Ihrer Nachricht auf bestimmte Dateien oder sogar Teile von Dateien verweisen. Chat-Variablen können dabei helfen! Verwenden Sie #, um eine Variable aus dieser Liste aufzurufen:


- **#file:** Verweist auf eine bestimmte Datei in Ihrem Arbeitsbereich.
- **#codebase:** Der gesamte Inhalt des geöffneten Arbeitsbereichs. Ähnlich wie die Verwendung von @workspace und kann dann nützlich sein, wenn Sie mit einem anderen Agenten chatten (wie @terminal), aber dennoch auf die komplette Lösung verweisen wollen.
- **#editor:** Quellcode im sichtbaren Teil des Editors.
- **#git:** Aktuelles Git-Repository – Branch, Remotes, Pfad, usw.
- **#selection:** Der aktuell ausgewählte Code.
- **#terminalLastCommand:** Der letzte ausgeführte Befehl im Terminal des Editors.
- **#terminalSelection:** Auswahl im Terminal des Editors.

Beispiel: Nehmen wir an, dass wir Hilfe bei der Verbesserung von Methodennamen in einer bestimmten Datei benötigen (und wir wollen sicherstellen, dass der gesamte Inhalt der Datei berücksichtigt wird).



webmaxru

suggest consistent naming of the methods in `#file:index.js`

 GitHub Copilot

Used 2 references >

Here are some suggestions for consistent naming of the methods in your `index.js` file:

1. `parseStringTemplate` to `parseTemplateString`: This makes it clear that the function is parsing a template string, not a string template.
2. `getChatHistoryAsText` to `convertChatHistoryToText`: This makes it clear that the function is converting the chat history into text format.
3. `removeNewLines` to `removeNewlineCharacters`: This makes it clear that the function is removing newline characters from a string.

Remember, the goal of naming functions is to make it clear what they do. The names should be descriptive and follow a consistent pattern.

Produktivitätstipp: Verwenden Sie die Auf- und Abwärtspfeile der Tastatur, um die Chat-Variablen auszuwählen, die Sie nach Eingabe von # benötigen. Bei der Eingabe von #Datei verwenden Sie ebenfalls die Tastatur, um eine der vorgeschlagenen Dateien auszuwählen.

Rufen Sie die meistgenutzten Aktionen mit Slash-Befehlen auf

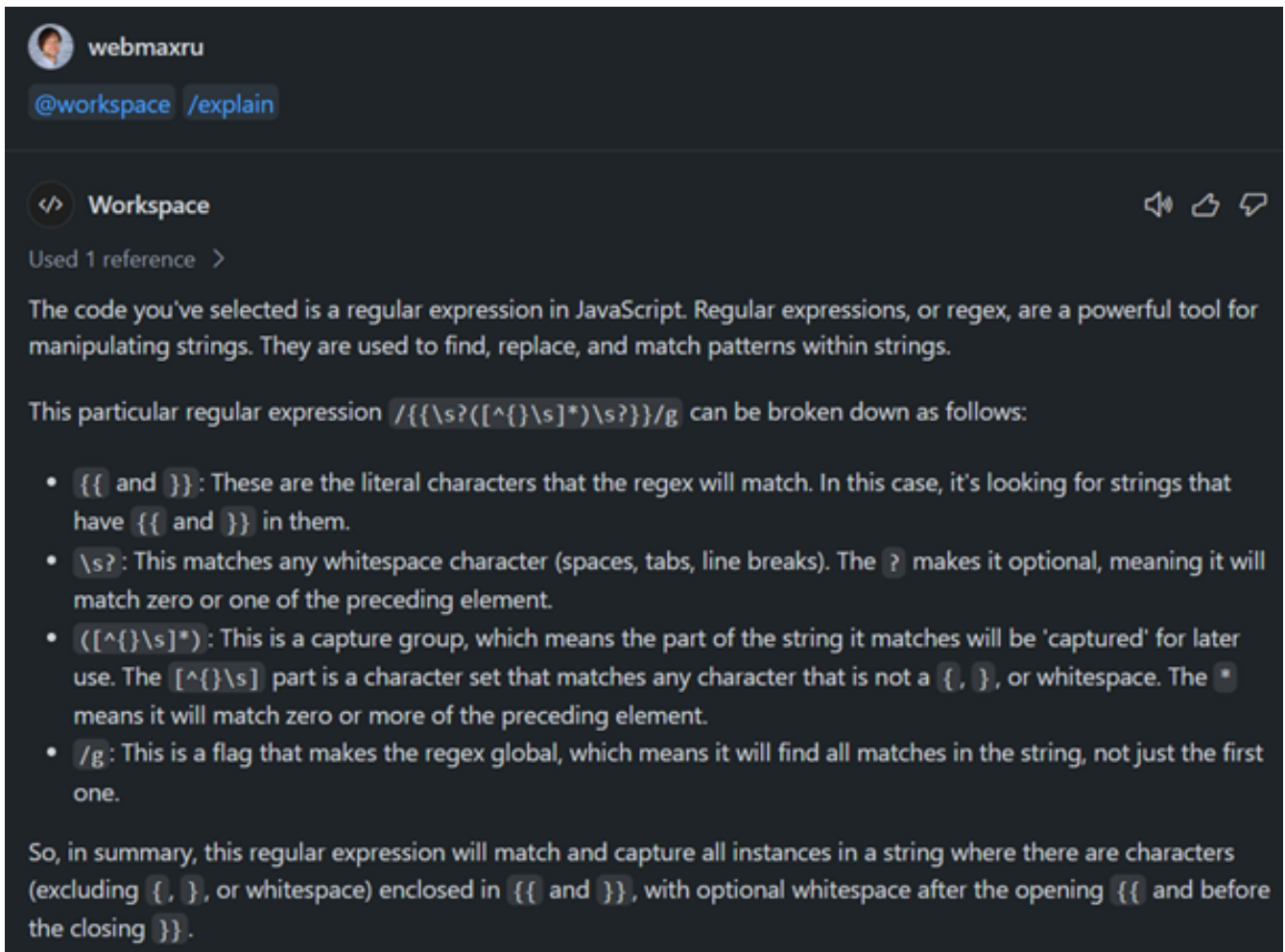
Es macht Spaß, mit dem Code in natürlicher Sprache zu kommunizieren, aber noch besser ist es, wenn man häufig verwendete Aktionen mit praktischen Tastenkombinationen aufrufen kann. Vergleichen Sie die Eingabe der vollständigen Nachricht „Erkläre, wie der ausgewählte Code funktioniert“ mit der Eingabe von „/“ und der anschließenden Auswahl von „/explain“ aus dem Popup-Overlay mithilfe der Pfeiltasten.

Ein weiterer Vorteil einer vordefinierten Syntax für Befehle ist die Gewissheit, dass GitHub Copilot unsere Absicht zu 100 Prozent richtig versteht (natürliche Sprache kann eine gewisse Mehrdeutigkeit aufweisen). Es sind eine Reihe von Slash-Befehlen verfügbar. Sie können diese in Verbindung mit dem Verweis auf einen Chat-Teilnehmer verwenden, um den gewünschten Rahmen festzulegen. Einige der Befehle sind:

- **/help:** Hilfe zu verfügbaren Slash-Befehlen, Chat-Teilnehmern, Chat-Variablen und mehr.
- **/doc:** Erstellung einer Dokumentation für den Code.
- **/explain:** Erklärung, wie der Code funktioniert (oder Hilfe mit Terminal-Befehlen, wenn Sie `@terminal` voranstellen).
- **/fix:** Optimierung und/oder Behebung von Problemen im Code.

- **/tests:** Erstellung von Unit-Tests für den Code.
- **/new:** Erstellung eines Gerüsts für einen neuen Arbeitsbereich.

Beispiel: Lassen Sie sich eine Erklärung für einen der regulären Ausdrücke im Code geben. Wählen Sie die Codezeile aus und verwenden Sie den Slash-Befehl „/explain“.



Produktivitätstipp: Probieren Sie den GitHub Copilot Chat im Inline-Modus aus, anstatt den Chat im Seitenfenster geöffnet zu haben. Drücken Sie `Strg+I` (`Cmd+I`) und geben Sie Ihre Nachricht in das kleine Overlay-Dialogfeld ein, das direkt über der Zeile erscheint, in der sich Ihr Cursor im Codefenster befindet.

Zusammenfassung

Verwenden Sie Chat-Teilnehmer, Chat-Variablen und Slash-Befehle, um die volle Kontrolle über den Gesprächskontext zu behalten, ein korrektes und konsistentes Verständnis Ihrer Absichten sicherzustellen und letztendlich schneller zu chatten und zu programmieren!

Nutzen Sie GitHub Copilot für Ihre Projekte

Diese 16 Tipps unserer Autor*innen bieten einen guten Einblick in die faszinierenden Möglichkeiten von GitHub Copilot – die vorgestellten Beispiele sind allerdings nur die Spitze des Eisbergs, wenn es darum geht, wie dieses innovative Tool Ihre Entwicklungsprozesse revolutioniert.

Um die volle Bandbreite von GitHub Copilot zu erleben, sollten Sie den KI-Pair-Programmierer am besten selbst ausprobieren. Die Integration in Ihren Workflow könnte einen signifikanten Unterschied machen, sei es in der Beschleunigung von Routinetätigkeiten oder in der Inspiration für neue Ansätze in der Entwicklung.

Wir sind natürlich auch gespannt auf Ihre Erfahrungen und Tipps im Umgang mit GitHub Copilot. Wir laden Sie dazu ein, Ihre eigenen Erkenntnisse und Empfehlungen mitzuteilen. Ihre wertvollen Einblicke könnten in der nächsten Ausgabe unseres Whitepapers einen Platz finden und anderen Entwickler*innen dabei helfen, das Beste aus GitHub Copilot herauszuholen. Kommen Sie gerne unter der E-Mail-Adresse techwiese@microsoft.com auf uns zu.

Innovation steht im Zentrum der Softwareentwicklung und GitHub Copilot ist zweifellos eine bahnbrechende Bereicherung. Nutzen Sie diese Gelegenheit, Ihr Coding-Erlebnis zu verbessern, und lassen Sie uns gemeinsam die Zukunft der Softwareentwicklung gestalten.

Starten Sie noch heute

Wenn Sie, ohne Zeit zu verlieren, mehr über die Azure Cloud-Plattform erfahren möchten, starten Sie noch heute:



Kontaktieren Sie Ihren Microsoft Account Manager
[Azure-Vertrieb kontaktieren | Microsoft Azure](#)



Finden Sie einen kompetenten Microsoft Partner
[Erfahrene Azure-Partner finden | Microsoft Azure](#)



Weitere Informationen zum Thema Developer Productivity
[Azure-Developer-Productivity | Microsoft Azure](#)

Weiterführende Ressourcen

- [Features von GitHub Copilot](#)
- [Dokumentation zu GitHub Copilot](#)
- [Quickstart für GitHub Copilot](#)
- [GitHub Copilot in VS Code](#)
- [GitHub Copilot Labs](#)
- [GitHub Copilot Fundamentals: Grundlegendes zum KI Pair-Programmierer](#)
- [Erste Schritte mit GitHub und Visual Studio Code](#)
- [Was ist die GitHub Copilot-Erweiterung für Visual Studio?](#)
- [GitHub Copilot Trust Center](#)
- [GitHub Blog](#)

© 2024 Microsoft. Alle Rechte vorbehalten. Namen und Produkte anderer Firmen können eingetragene Warenzeichen der jeweiligen Rechteinhaber sein. Dieses Dokument wird in der vorliegenden Form zur Verfügung gestellt. Die in diesem Dokument enthaltenen Ansichten und Informationen (einschließlich URLs und anderer Verweise auf Websites) können ohne vorherige Ankündigung geändert werden. Sie tragen das Risiko der Nutzung. Mit diesem Dokument erhalten Sie keinerlei Rechte an geistigem Eigentum eines beliebigen Microsoft Produkts. Sie dürfen dieses Dokument zu internen Referenzzwecken kopieren und verwenden.

