S&S www.entwickler.de Deutschland € 9,80 Österreich € 10,80 Schweiz sFr 19,50

entwickler magazin

entwickler magazin

März/April 2.2021

Trends 2021

Java, .NET, JavaScript: Was man kennen sollte

Neue Kolumne: Per Anhalter durch den JavaScript-Dschungel

Machine Learning

Wie intelligente Neuronen dendritisch lernen

Go für Einsteiger

Einführung in das io-Paket

Legacy-Code

Tipps für die Modernisierung alter Anwendungen

4 194156 609807 02

Hype oder Trend, das ist hier die Frage

Liebe Leserinnen und Leser,

Tech-Hypes sind so eine Sache: Bitcoins sollten zum großen Trendsetter der IT-Branche werden, Blockchains für quasi jeden Anwendungszweck zum Einsatz kommen. Das sind nur zwei der großen Trends der vergangenen Jahre, die in den meisten Branchen doch nicht zum Disruptor, sondern eher zu starken Nischentechnologien geworden sind. Das ist nichts Schlechtes, immerhin hat jede Technologie ihren Kernanwendungsbereich. Dennoch zeigen diese Beispiele, dass die Themen, über die alle eine Zeitlang reden, nicht immer auch die sind, die sich wirklich in der breiten Masse durchsetzen. Oft sind es kleinere Trends und langsame Veränderungen, die wirklich einen Unterschied machen, wie neue Sprachfeatures und APIs auf den jeweiligen Plattformen. Manches Trendthema braucht einfach auch mehr Zeit als ursprünglich gedacht, um sich auf dem Markt zu etablieren. Chatbots gelten beispielsweise schon seit einigen Jahren als Trend und haben doch erst in den letzten ein, zwei Jahren wirklich auch in der Praxis an Beliebtheit gewonnen. Die Technologien dahinter brauchten Zeit, um auszureifen, bevor die breite Adaption starten konnte.

Spannend bleibt die Frage nach den kommenden Trends trotzdem. Immerhin bieten neue Technologien einen Raum zum Lernen und Experimentieren; Bereiche, in denen es noch keine gesetzten Standards gibt, sind aufregend und lassen Spielräume, die anderswo durch Best Practices besetzt sind. Es macht Spaß, Neues zu entdecken! Das gilt gerade für die IT-Welt, in der Trends häufig mit ganz neuen Möglichkeiten einhergehen. Wer hätte vor einigen Jahren gedacht, dass man mit WebAssembly heute so leicht Sprachen wie C++ in den Browser bringen kann? Hypes sind also mit Vorsicht zu genießen; Trends und Neuerungen hingegen sind ein zentraler Bestandteil der IT-Welt.

Die Vielfalt der neu aufkommenden Themen ist jedoch so groß, dass es manchmal durchaus schwerfällt, den Unterschied zwischen kurzlebigem Hype und wichtigem Trend einzuschätzen. Hat der Marktanalyst Gartner Recht damit, dass die nächste Evolutionsstufe des Internets ein Internet of Behavior sein wird, das das Verhalten der Nutzer auf Datenbasis beeinflusst? Oder ist jetzt eher das Jahr des Smart Home gekommen, dieses Mal aber wirklich? Womit sollte man sich beschäftigen?

Um Ihnen den Überblick über die wichtigsten Entwicklungen und Trends in der Tech-Welt zu erleichtern, haben wir unsere Autor*innen um ihre persönlichen Einschätzungen gebeten: Welche Neuerung aus dem Jahr 2020 muss man kennen? Womit wird es 2021 weitergehen? Was tut sich in den Tech-Welten? Das große Entwickler-Nähkästchen in dieser Ausgabe des Entwickler Magazins gibt einen Überblick über die ganz persönlichen Perspektiven der Profis auf die großen Trends der Branche. Daneben gehört zu unserem Titelthema „Trends 2021“ auch der JAMStack, der bereits seit Jahren immer wieder heiß diskutiert wird – eventuell nimmt das Architekturkonzept einen ähnlichen Weg wie der Chatbot-Trend und entwickelt sich langsam und stetig zum Standard. Oder schafft Go es dieses Jahr, JavaScript ein wenig mehr in den Schatten zu stellen? So oder so lohnt sich der Blick auf die Sprache, die nach dem ersten Hype zum beständigen Mitglied der Programmiersprachfamilie geworden ist. Angular 11 rundet das Titelthema ab – am Ball zu bleiben, Anwendungen auf dem neuesten Stand zu halten und somit von neuen Entwicklungen zu profitieren, liegt nämlich immer im Trend.

Und jetzt wünsche ich Ihnen viel Spaß bei der Lektüre!

Ann-Cathrin Klose
Redakteurin Entwickler Magazin
@entwickler

Was ist der JAMstack?

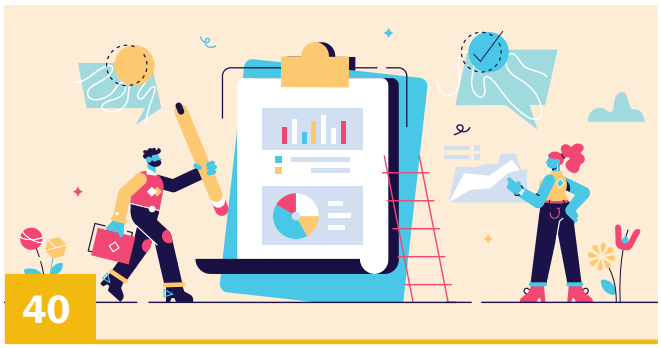
Gerade bei contentlastigen Seiten und Anwendungen, bei denen Skalierung und Sicherheit eine große Rolle spielen, bietet der JAMstack einen Ansatz, der die Entwicklung einfacher und günstiger machen kann.

15

10

Aus dem Entwickler-Nähkästchen

Welche Technologien haben 2020 an Bedeutung gewonnen, was wird 2021 besonders wichtig werden? Auch in diesem Jahr haben wir unsere Autoren zu diesen Themen befragt. Hier sind die Antworten aus den Bereichen .NET, JavaScript und Java.



40

Codequalität mit Node.js

Wichtig bei der Entwicklung mit Node.js ist nicht nur das Schreiben des eigentlichen Codes, sondern auch die Qualitätssicherung. Dazu zählen neben der gewissenhaften Strukturierung des Codes auch das Anwenden von Coderichtlinien und das Ausführen von automatisierten Tests.

59

Gut getestet läuft besser

Auf der SaaS-Plattform ServiceNow können Applikationen mit dem Automated Test Framework (ATF) getestet werden. Wie geht man dazu vor, was muss man beim Aufsetzen des ersten Tests beachten? Darauf geht dieser Artikel ein und gibt zudem einige allgemeine Tipps zur Arbeit mit der Plattform.

101

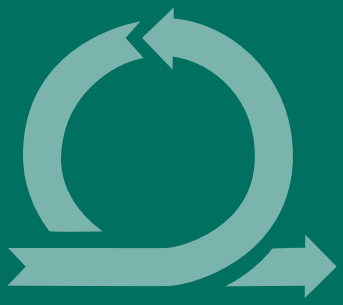
Der Faktor Mensch

Ein Softwareprojekt ist mitunter ein komplizierter Mikrokosmos. Diesen gilt es an die Veränderungen der Zeit anzupassen, zu warten und zu modernisieren. Wie kann das möglichst ohne Komplikationen gelingen? In dieser Artikelserie werden wir dieser Frage auf den Grund gehen.

INHALT 2.2021

Kolumne: A² – alles Agile

von René Schröder



Open Space Agility

Einladungen und kein Zwang sind der Schlüssel zu Vorhaben, die langfristig und nachhaltig gelingen können. Oft lässt sich jedoch genau das Gegenteil beobachten: In tayloristischen Unternehmensstrukturen wird in vielen Fällen weiterhin in sämtlichen Belangen Top-down entschieden. Auch Transformationen und die Wege durch diese werden so bestimmt. Beobachtungen legen nahe, dass diese Art, eine Transformation anzugehen, nicht funktioniert. Das hat mehrere Gründe. Ein wichtiger Punkt ist, auch in Erinnerung an das Cynefin Framework [1], dass eine Transformation in einem komplexen Umfeld stattfindet. Komplex bedeutet, dass nicht wie im Simple-Umfeld auf Best Practices zurückgegriffen werden kann, sondern der Weg mittels Inspect and Adapt beschritten werden muss, um für den aktuellen Kontext die richtigen Maßnahmen zu identifizieren. Sich das wieder ins Gedächtnis zu rufen, ist besonders wichtig. Eine Transformation kann nicht eingekauft werden, indem sich eine Methode ausgesucht wird, hierfür Berater engagiert werden, die diese Methode (z. B. ein agiles Framework) dann im Unternehmen implementieren. Der Wunsch nach Sicherheit, wenn auch nur einer trügerischen, lässt viele Unternehmen nach vermeintlichen Best-Practices-Ansätzen greifen, die schon in anderen Unternehmen funktioniert haben oder zumindest den Anschein hierfür wiedergeben. Komplex bedeutet, die für jedes Unternehmen richtige Methode muss auch im Unternehmen durch alle oder viele erarbeitet werden. Und dieses Erarbeitete funktioniert dann am besten, wenn es auf Freiwilligkeit beruht. Durch diese Herangehensweise kann jeder im Unternehmen selbst bestimmen, was er probieren möchte, um die Transformation erfolgreich zu gestalten. Dafür braucht es aber natürlich einen Rahmen, ansonsten wird aus Inspect and Adapt schnell ein unkoordiniertes Herumprobieren ohne Vorbereitung und Reflexion.

René Schröder ist seit dem Jahr 2001 in verschiedenen Branchen (u. a. Automotive, Consumer und Reisen) als Agile Coach, Business Coach und Trainer erfolgreich im Markt unterwegs. Er ist Experte für die erfolgreiche Implementierung eines ganzheitlichen Methodenmixes, bestehend aus agiler Produktentwicklung, Softwarearchitektur und Testing in komplexen Entwicklungskontexten, um langfristig den Projekt- und Produkterfolg sicherzustellen.

www.regsus.de r.schroeder@regsus.de @Rene__Schroeder

Überblick

Der Open-Space-Agility-Ansatz (OSA) [2] beschreibt diese Möglichkeit der gemeinsamen Transformation (**Abb. 1**). Eine Transformation besteht bei OSA aus drei Phasen: der Vorbereitungsphase, der Durchführungsphase und der Abschlussphase. In der Vorbereitungsphase, die 60 Tage dauert, findet unter anderem die Versendung der Einladung an die Teilnehmer statt. Die Einladung (zum ersten Open-Space-Tag, OST-1) dient dazu, einem möglichst großen Kreis von Personen im Unternehmen die Möglichkeit zu geben, die Transfor-

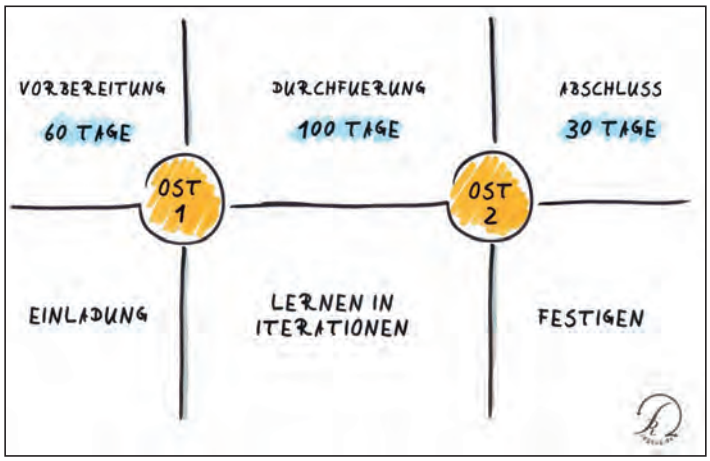


Abb. 1: OSA-Überblick

mation mitzugestalten. Das ist freiwillig. Um all jenen, die mitgestalten wollen, genug Zeit zu geben, empfiehlt Open Space Agility, hierfür 45 Tage einzuräumen.

Der Open Space bekommt ein Thema, das den Zweck der Transformation beschreibt. Im OST-1 erarbeiten die Teilnehmer zusammen, welche Ansätze und Methoden sie in den nächsten 100 Tagen, der Durchführungsphase ausprobieren möchten, um Erfahrungen sammeln zu können, ob diese Methoden im gegebenen Kontext geeignet sind, für mehr Agilität im Unternehmen zu sorgen. Durch diesen Modus wird keine Methode dem Unternehmen, den Mitarbeitern oder Teams aufgezwungen, sondern alle entscheiden selbst, was das Richtige sein könnte. Da in der Einladung zum OST-1 auch explizit darauf hingewiesen wurde, dass es eine Einladung zur Transformation und keine Pflicht ist, steht es jedem frei, diese Einladung auch abzulehnen und nicht am OST-1 und der Transformation teilzunehmen. Während dieser 100 Tage sammeln alle Willigen Erfahrungen und probieren aus, was geht und was nicht geht. Diese Erfahrungen werden im zweiten Open Space, dem OST-2 zusammen ausgewertet, um voreinander zu lernen und die richtigen Methoden langfristig im Unternehmen zu etablieren.

Die 30 Tage nach dem OST-2 festigen das in besonderem Maße. OST-1 und OST-2 sind die Open-Space-Meetings im Open-Space-Agility-Ansatz, die nach Open Space Technology durchgeführt werden. Open Space Technology wird kurz auch oft nur Open Space genannt, also als „wir führen ein Open Space Event durch“ verstanden. So ist es auch bei Open Space Agility: OST-1 und OST-2 sind Open Space Events.

Open Space Technology

Open Space Agility trägt bereits im Namen den Ansatz von Open Space Technology. Genau wie bei Open Space Agility wird auch hier zu einem Open Space in einem Unternehmen eingeladen. Das kennen manche vielleicht schon aus der Meet-up-Szene. Der Einladung kommt hier eine besondere Bedeutung zu, der wir uns später widmen. Sowohl die Teilnahme am gesamten Open Space (OST-1) wie auch die Beteiligung in den jeweiligen Diskussionen im Open Space ist komplett freiwillig. Open Space Technology geht davon aus, dass jedes Individuum selbst entscheiden kann, wo es wie den meisten Wert für sich oder das Unternehmen stiften kann. Das bedeutet, dass auch die Agenda für den Open Space durch die Teilnehmenden entwickelt wird, begrenzt durch das Thema, das in der Einladung durch den Sponsor vermittelt wurde. Open Space Events teilen sich auf in drei Phasen: Eröffnungsphase, Diskussionsphase und Schlussphase.

In der Eröffnungsphase, die der Sponsor eröffnet und die durch den Moderator unterstützt wird, wird die Agenda für das Open Space Event erstellt. Bevor das geschehen kann, erklärt der Moderator die Prinzipien von Open Space:

Das „Gesetz der zwei Füße“ besagt, dass Teilnehmer jede Diskussion zu jeder Zeit verlassen bzw. wechseln können.

- Wer auch immer kommt, es sind die richtigen Leute.
- Was auch immer geschieht, es ist das Einzige, was geschehen konnte.
- Es beginnt, wenn die Zeit reif ist.
- Vorbei ist vorbei. Nicht vorbei ist nicht vorbei.

Um nochmals die Idee der Freiwilligkeit und Einladung zu unterstreichen, wird das „Gesetz der zwei Füße“ übermittelt, das besagt, dass Teilnehmer jede Diskussion zu jeder Zeit verlassen und/oder wechseln können, wenn sie das Gefühl haben, nichts mehr beitragen zu können. Nach dieser Erläuterung erarbeiten die Teilnehmer die Agenda für die folgende Diskussionsrunde. Hierbei halten sich der Moderator und der Sponsor komplett zurück. Eine Möglichkeit, eine Agenda zu gestalten, besteht darin, dass die Teilnehmer in einer Brainstorming-Session beliebig viele Themen sammeln. Diese Themen werden dann darauffolgend mit allen Teilnehmern geteilt und entsprechend in einen Raum/Zeitslot verortet. In diesen Räumen und Zeitslots wird in der folgenden Runde diskutiert, gearbeitet, erarbeitet und vieles mehr. All jene Herangehensweisen, die richtig erscheinen, sind es auch und werden herangezogen. Im Anschluss an eine Session wird das Ergebnis dokumentiert. Dieser Punkt ist gerade für die abschließende Runde wichtig, in welcher der Tag reflektiert wird und die Ergebnisse dem Sponsor übergeben werden.

Einladung zum Open Space

Bevor ein Open Space oder im Kontext von Open Space Agility OST-1 starten kann, bedarf es einer Einladung, die vom Sponsor verschickt wird. Einladungen, wie der Name schon sagt, bedingen Freiwilligkeit, sie sind keine Vorladungen. Leider wird das in vielen Unternehmen nicht so gelebt, daher ist dieser Punkt nochmals besonders hervorzuheben und darauf hinzuweisen. Einladungen sind dann gut gestaltet, wenn sie diese vier Elemente besitzen:

- Ein Thema/Ziel
- Die Regeln
- Fortschrittsmessung
- Hinweis auf Freiwilligkeit

Das Thema soll die Eingeladenen dazu motivieren, am Open Space Event teilzunehmen, weil sie glauben, dass sie einen wertvollen Beitrag leisten können, sowohl im OST-1 wie auch in der darauffolgenden Phase. Entsprechend intensiv muss der Sponsor sich mit diesem auseinandersetzen. Das Verfassen der Einladung, auch wenn es im ersten Moment so einfach klingt, ist meist die erste große Hürde, die es zu bewältigen gilt. Und auch diese Tätigkeit kann vom Sponsor nicht ausgelagert oder eingekauft werden. Die Phase der Vorbereitung umfasst deshalb 60 Tage. Innerhalb der ersten 15 Tage wird die Einladung verfasst, sodass die potenziellen Teilnehmer genug Zeit haben, über diese zu reflektieren und diesen Tag möglich zu machen, so sie sich dafür entscheiden. Während der 60 Tage geht es ebenfalls darum, unter anderem mit dem Sponsor zu arbeiten und ihm zu ermöglichen, einen geeigneten Rahmen für den OST-1 und den gesamten Zyklus bereitzustellen. Dazu gehört, ein Verständnis dafür zu schaffen, was eine Einladung im Gegensatz zu einer Vorladung bedeutet. Ebenso muss man das Wissen vermitteln, was ein Open Space ist und wie er dazu dient, die Transformation erfolgreich zu gestalten.

Fazit

Der Rahmen für Open Space Agility ist gefasst, wenn innerhalb der ersten 15 Tage die Einladung für den OST-1 versendet wird und die Teilnehmer sich die Zeit nehmen können, eine Entscheidung darüber zu treffen, in welcher Art sie am besten am OST-1 und dem Zyklus teilnehmen wollen. Gelingt es mittels dieser Einladung, die nötige Transformation zu adressieren, ist die Chance, diese erfolgreich zu gestalten, um Einiges gestiegen. Die Macht der Einladung und der Freiwilligkeit sind die entscheidenden Schlüssel. Neben der Rolle des Sponsors existieren bei Open Space und Open Space Agility noch weitere Rollen, die zum Erfolg der Transformation beitragen. Diese Rollen und der weitere Verlauf von OSA werden in der folgenden Kolumne beleuchtet.

Links & Literatur

[1] Brougham, Greg: „The Cynefin MiniBook“, lulu.com, 2015

[2] Mezick, Daniel: „The Open Space Agility Handbook“, 2015

Content-zentrische Anwendungen für das Informationszeitalter

Innovationsblocker Lastenheft

Bei der Auswahl passender Content-zentrischer Lösungen für das Enterprise Content- und Dokumenten-Management gehen viele Unternehmen den klassischen Weg und erstellen ein Lastenheft. Sie beschreiben damit zwar die Anforderungen des Projekts nach aktuellem Kenntnisstand, berauben sich damit aber jeglicher Innovationsfähigkeit.

Weg mit der Vollkasko-Mentalität bei ECM-Projekten

Je mehr Funktionen, desto besser? Ein Trugschluss. Nicht erst die Corona-Krise hat eines klar gezeigt: In einem sich verändernden Markt behaupten sich vor allem agile Unternehmen, die auf Veränderungen flexibel reagieren können. Im Standard vieler ECM-Lösungen ist dieser Punkt aber nicht vorgesehen. OPTIMAL SYSTEMS bietet Unternehmen mit der yuuvis RAD-Plattform einen anderen Ansatz: Damit lassen sich Content-zentrische Anwendungen so einfach und schnell wie nie zuvor entwickeln sowie anpassen.

No-Code-/Low-Code: Programmieren, ohne zu coden

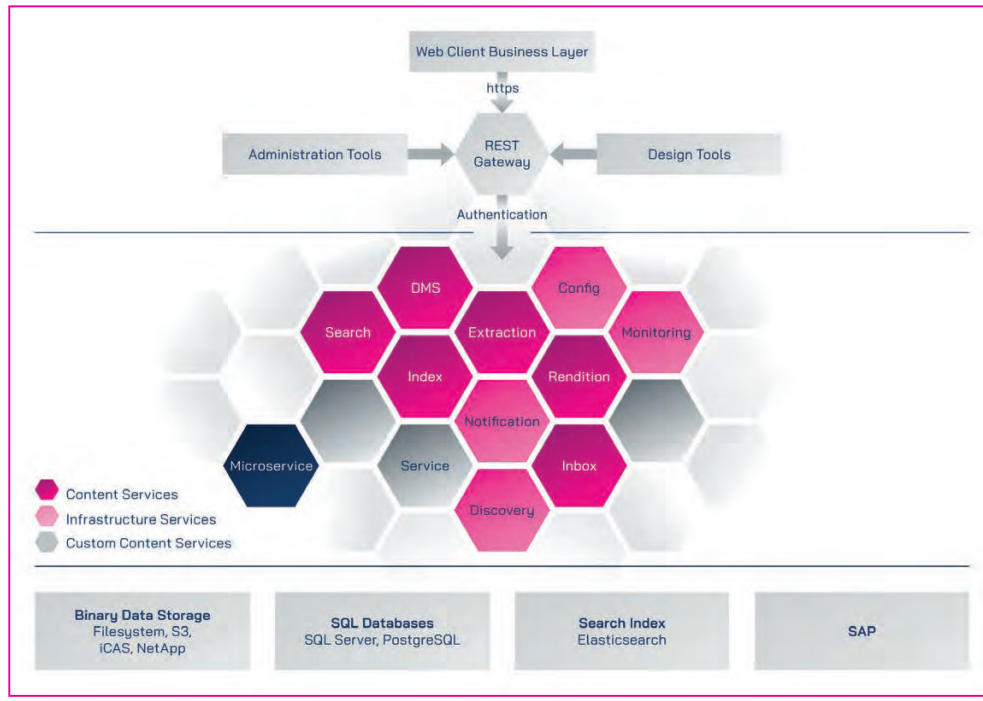
Ein konsequenter No-Code-/Low-Code-Ansatz zeichnet yuuvis RAD aus. Out-of-the-box stehen eine Vielzahl klassischer ECM-Funktionalitäten zur Verfügung, wie OCR/Capture, Langzeitarchivierung/-speicherung, Compliance und Auditierung, Zugriffsmanagement, Versionierung und viele mehr. Über ein modellbasiertes Design, Scripting, Plugins und projektspezifische Sichten (Custom States) entstehen mit geringem Aufwand hocheffiziente Anwendungen. Eigene Custom Microservices erlauben darüber hinaus eine flexible Anpassung sowie den weiteren Ausbau. Bereits bestehende Anwendungen können beispielsweise in kurzer Zeit um Archivfunktionalitäten erweitert werden – unabhängig vom enthaltenen Content. yuuvis RAD reduziert mit weniger Code das Potenzial an Fehlern und vereinfacht die Wartung.

Open-Source-Technologien reduzieren die Betriebskosten

Auch für komplexere Anwendungsfälle ist yuuvis RAD die geeignete Anwendung. Die Lösung baut auf offenen Standardtechnologien und Best Practices in der Softwareentwicklung auf. Alle serverseitigen Komponenten sind Java-basiert und implementieren eine Microservices-Architektur, die von Spring Cloud Netflix orchestriert wird. Die Kommunikation läuft über HTTP(S). Der Webclient ist eine 100-prozentige HTML5 Angular (TypeScript) Anwendung und das serverseitige API ist rein REST-basiert. Die strikte Anwendung dieser neuesten Standards ermöglicht yuuvis-Kunden den sofortigen Zugriff auf das größte Netzwerk von Softwareexperten weltweit. Mit der hochmodernen und freien Open-Source-Datenbank PostgreSQL wird neben Microsoft SQL Server zudem eine weitere relationale Datenbank-Engine unterstützt. Da PostgreSQL im Rahmen der Open-Source-Lizenz kostenfrei installiert und genutzt werden darf, lassen sich die Betriebskosten für yuuvis RAD deutlich reduzieren – ohne dass es zu Einbußen bei der Performance kommt.

Flexibel by design

Unternehmen können mit yuuvis RAD klein starten, schnell wachsen und flexibel auf Veränderungen reagieren. Im Vergleich zu Standard-ECM-Anwendungen mit ihrem festgeschriebenen Leistungsumfang und geringen Customizing-Möglichkeiten stellen sie mit yuuvis RAD sicher, weiterhin innovationsfähig für neue Geschäftsideen zu bleiben.



OPTIMAL SYSTEMS

Die OPTIMAL SYSTEMS Unternehmensgruppe ist ein Pionier für Informationsmanagement-Technologien und verfügt über 30 Jahre Erfahrung in der Geschäfts- und Standardsoftwareentwicklung im Bereich Dokumentenmanagement.
Seit Anfang 2020 ist das eigenständige Unternehmen Teil des weltweit agierenden Kyocera Konzerns.

https://www.optimal-systems.de
https://yuuvis.com

Technologietrends 2020 und 2021

Aus dem Entwickler-Nähkästchen

Welche Technologien haben 2020 an Bedeutung gewonnen, was wird 2021 besonders wichtig werden? Auch in diesem Jahr haben wir unsere Autoren zu diesen Themen befragt. Hier sind die Antworten aus den Bereichen .NET, JavaScript und Java.

Entwickler Magazin: Welche technologische Entwicklung im Jahr 2020 ist bei dir persönlich am meisten hängen geblieben?

Sandra Parsick: Am meisten hängen geblieben ist die Bewegung in der Containerwelt. Gefühlt waren hier jeden Tag ein neues Tool, neue Vorgehensweisen etc. zu lernen. Das erinnert mich ein wenig an die Bewegung vor zehn Jahren in der JavaScript-Welt.

Michael Vitz: Wie vermutlich bei vielen anderen auch hat sich dieses Jahr durch die Pandemie vor allem in Bezug auf das Thema Remotearbeit einiges bei mir geändert. Neben organisatorischen Themen hat sich hier auch technisch eine Menge getan. Plötzlich ist es für mich möglich, unseren Kunden per Chat und Video/Audio von zu Hause aus zu helfen, ohne längere Reisen. Abseits davon hat sich an der Technik, die ich in Projekten einsetze, dieses Jahr nichts Revolutionäres getan. Java funktioniert nach wie vor, hatte aber kein Major-Release, das wir einsetzen und auch im Spring-Umfeld gab es kein aufregendes neues Release dieses Jahr.

Holger Schwichtenberg: Die erste Version von Blazor WebAssembly, die im Mai 2020 erschienen ist! Sie ist sicherlich noch nicht perfekt, aber es ist der Anfang einer Revolution der Browserprogrammierung. .NET-Entwickler müssen nicht mehr JavaScript oder TypeScript lernen und bestehende Codebasen auf eine andere Programmiersprache umschreiben, um echte Single Page Web Applications bzw. Progressive Web Apps zu programmieren.

Daniel Mies: Im Frontend sind in diesem Jahr zwei Frameworks besonders aufgefallen: zunächst Vue.js, das

mit der Version 3.0 nochmal einen großen Sprung und vor allem mit TypeScript-Support und Composition API auf sich aufmerksam gemacht hat. Etwas kleiner, aber dafür umso spannender war das Thema Svelte, das auf einen eigenen Compiler setzt und damit sehr performante und kleine Anwendungen ermöglicht. Das Thema JAMStack ist ebenfalls nochmal aufgekocht und zeigt wahrscheinlich, wie wir in den nächsten Jahren Webseiten und Anwendungen bauen können.

Sebastian Springer: Der Satz, der bei mir 2020 hängen geblieben ist und der für mich auch so ein bisschen 2020 zusammenfasst: „No new Features". Das war die Releasezusammenfassung von React 17. Wir befinden uns im Moment an einem Punkt, an dem unsere Tools einen Grad der Komplexität erreicht haben, an dem sich die Maintainer echt schwertun, größere Änderungen zu integrieren. Und das betrifft nicht nur React mit seinem Concurrent Mode, auch Angular hat sich mit Ivy schwergetan. Aber auch Vue mit seiner Version 3 ist da keine Ausnahme. Das ist jetzt aber weniger ein Vorwurf in Richtung der Maintainer, sondern eher ein Aufruf an uns alle, dass wir unsere Erwartungshaltung an die (Open-Source-)Werkzeuge, die wir tagtäglich nutzen, neu justieren sollten. Ansonsten war 2020 für mich eher eine Evolution als eine Revolution. Es gab mehr Bibliotheken und Werkzeuge, die die Entwicklung vereinfachen. Eine nette Abwechslung war hier vielleicht noch Svelte, wobei das Framework auch nicht wirklich neu ist, allerdings einen interessanten und leichtgewichtigen Ansatz verfolgt.

EM: Dein Blick voraus: Welcher Trend wird aus deiner Sicht im Jahr 2021 besonders relevant werden?

Michael Hofmann: Ich glaube weiterhin daran, dass sich die Service-Mesh-Technologie weiter in der Cloud etablieren wird. Fast alle Cloud-Provider bieten ein Service-Mesh-Tool als managed Service an oder ermöglichen die eigenhändige Installation eines eigenen Mesh-Werkzeugs. Darüber hinaus sehe ich die Integration von Cloud-native API-Gateways mit der Service-Mesh-Technologie weiter fortschreiten.

Sebastian Springer: Das Wetten auf JavaScript-Trends fühlt sich für mich etwas nach Glücksspiel an: Gewinnchance 1:100 000. Ihr wisst ja: Es gibt Milchprodukte, die länger halten als so manche JavaScript-Bibliothek. Dieser Satz, den ich zum ersten Mal in einem Vortrag von Manfred Steyer gehört habe, fasst die JavaScript-Welt recht gut für mich zusammen. Aber zurück zu den Trends: Ich sehe für mich zwei relevante Baustellen: Wie schaffen wir es, möglichst wenig Code, und dabei möglichst nur relevanten Code, zum Client zu übertragen und dabei gleichzeitig die maximale Performance und Usability rauszuholen? Hier hat sich mit Server-side Rendering, Lazy Loading und sehr viel Toolunterstützung in letzter Zeit viel getan. Die zweite Baustelle: unabhängige Komponenten, also im weitesten Sinne Web Components, und mit ihnen die Möglichkeit, Microfrontends zu bauen. Dafür müssen wir jedoch noch einige Probleme lösen, wie beispielsweise den Footprint der Frameworks und Bibliotheken. Und auch hier kommt wieder Baustelle Nummer 1 zum Tragen: Ich möchte nur relevanten Code ausliefern müssen. In einer kleinen perfekten Welt können in einer solchen Microfrontend-Architektur Komponenten aus Angular, Vue und React friedlich mit nativen Web Components parallel zueinander existieren.

Holger Schwichtenberg: Ich freue mich auf „One .NET", die Zusammenführung von Mono/Xamarin mit .NET, die „wegen der Pandemie" verschoben wurde und nun im November 2021 erscheinen soll.

EM: Dein guter Vorsatz: Womit möchtest du dich im Jahr 2021 genauer auseinandersetzen?

Christian Kaltepoth: Für mich stand, wie vermutlich für viele, das Jahr 2020 ganz im Zeichen der Coronapandemie. Im Vergleich zu vielen anderen Branchen ist die IT-Industrie hier deutlich weniger stark getroffen worden. Trotzdem hat die Pandemie natürlich auch großen Einfluss auf unseren Berufsalltag. Während in vielen Unternehmen Homeoffice noch die Ausnahme war, hat sich das schlagartig geändert. Plötzlich waren alle gezwungen, von Zuhause zu arbeiten, was viele Vorteile, aber auch einige Nachteile mit sich bringt. Die Gespräche an der Kaffeemaschine fallen plötzlich weg und es ist teilweise nicht mehr so leicht, sich mal kurz mit den Kollegen auszutauschen. Ich denke, dass viele Teams hier noch dazu lernen müssen, um auch remote genauso effizient arbeiten zu können wie zuvor. Das ist aus meiner Sicht besonders deshalb wichtig, weil die Pandemie uns sicherlich noch einige Zeit begleiten wird. Besonders im Bereich der Tools gibt es hier Einiges zu entdecken. So hat JetBrains mit Code With Me vor kurzem eine erste Vorabversion eines Plug-ins für IntelliJ veröffentlicht, das mehreren Entwicklern erlaubt, zusammen in einer IDE zu arbeiten. Besonders, wenn sich die Entwickler nicht im selben Raum befinden, eröffnet diese neue Art des Pair Programmings mittels Collaborative Editing ganz neue Möglichkeiten. Für Visual Studio Code gab es hier schon länger entsprechende Möglichkeiten. Aber besonders bei den klassischen Java IDEs gibt es noch einiges aufzuholen. Ich denke, dass es im kommenden Jahr im Bereich des Toolings für verteilte Teams noch einiges Neues zu entdecken geben wird, um die Arbeit von verteilten Teams zu erleichtern.

Daniel Mies: Neugierig bin ich auf Snowpack da die in Entwicklung befindliche Version 3 sehr spannend aussieht (bye bye *npm install*). Auffrischen werde ich auch meine Kenntnisse rund um das Thema Accessibility: Hier werden die Maßstäbe in den nächsten Jahren sicher höhergesteckt. Eng damit verbunden ist für mich auch das Thema Progressive Enhancement.

Marc Teufel: Eigentlich bin ich ein Verfechter des Mottos „Weniger ist mehr" und habe mich bis jetzt erfolgreich vor TypeScript gedrückt. Warum auch, bisher konnte ich ja auch alles mit JavaScript machen. Und es funktionierte sogar! Meine Mitarbeiter haben es jetzt aber geschafft: In einer gemeinsamen Aktion haben sie

mich ganz behutsam in die Welt von TypeScript eingeführt. Vermutlich nicht ganz uneigennützig, aber in jedem Fall vorrausschauend und vernünftig. So habe ich mir jetzt für das nächste Jahr fest vorgenommen, mich mehr mit TypeScript zu beschäftigen – und die Entscheidung meines Teams zu akzeptieren. Und sie haben ja Recht! Ich bin gespannt, wie sich funktionale Programmierung mit TypeScript anfühlt und natürlich freue ich mich darauf, die Decorator-Implementierung zu analysieren und sie mit dem TC39-Proposal zu vergleichen. Und wer weiß, vielleicht berichte im Entwickler Magazin über meine neuen Erkenntnisse.

Holger Schwichtenberg: Ich will mich intensiver mit Uno Platform beschäftigen, zur Entwicklung von Cross-Platform-Apps. Uno unterstützt seit August 2020 neben Windows 7 und 10, Android, macOS, iOS und Browser (via WebAssembly) nun auch Linux [1].

.NET

EM: .NET 5 ist im November 2020 erschienen – was war daran dein persönliches Highlight?

Holger Schwichtenberg: Über 80 neue Features für meinen bevorzugten OR-Mapper, Entity Framework Core. Durch Table-per-Type-Vererbungsunterstützung und N:M-Abstraktion per Skip Navigations ist nun die Migration bestehender klassischer Entity-Framework-basierter Anwendungen nach Entity Framework Core viel einfacher geworden!

EM: Was wünschst du dir für die Zukunft von .NET?

Holger Schwichtenberg: Ich wünsche mir neben den angekündigten jährlichen Hauptversionen im November (.NET 5, .NET 6, .NET 7 usw.) noch ein bis zwei Featurereleases (5.1, 5.2, 6.1, 6.2 usw.) während des Jahres. Nur eine Version pro Jahr ist für mich etwas zu „unagil“. Mit Previews darf ich bei einigen unserer Kunden leider nicht arbeiten.

JavaScript

EM: Wie schätzt du die Entwicklung des JavaScript-Ökosystems ein: Erleben wir eine Konsolidierung oder stehen wir vor der nächsten Disruption?

Sebastian Springer: Momentan stehen für mich die Zeichen eher auf Stabilisierung als auf einem radikalen Umbruch. Ein Blick auf die Entwicklung im letzten Jahr sagt da schon viel aus. Alle Hersteller achten stark auf einen evolutionären Ansatz. Ein Vorgehen, wie es Google beim Umstieg von AngularJS auf Angular praktiziert hat, kann sich in der aktuellen Konkurrenzsituation kaum ein größerer Hersteller leisten. Zu groß ist das Risiko, dass die Community negativ reagiert und die Entwickler abwandern. Diese Situation kommt uns als Entwicklern natürlich zugute, da wir uns nur wenige Sorgen machen müssen, dass wir unsere großen Applikationen radikal umschreiben müssen.

Daniel Mies: Auf Sprachebene eher eine Konsolidierung. Hier sollten Entwickler einfach die für sie passende Sprache wählen und, wo es weiterhilft, auch gerne im Projekt mischen. Die Entwicklung bei den Build-Tools verspricht, dass die Entwicklungserfahrung sich weiter verbessert.

EM: Was würdest du dir für die JavaScript-Welt wünschen?

Marc Teufel: Weniger ist mehr! Die JavaScript-Welt ist so fragil, Frameworks und Bibliotheken schießen wie Pilze aus dem Boden. Wer nicht aufpasst, läuft gerade in JavaScript-Projekten Gefahr, in der Komplexitätshölle zu enden. Ich wünsche jedem einzelnen JavaScript-Entwickler von Herzen, dass er sich selbst vor diesem Höllenritt bewahrt. Und das geht ganz einfach: Nicht auf jeden Zug aufspringen, nicht jede coole Bibliothek sofort ins eigene Projekt aufnehmen; nachdenken, überlegen und auf das eigene Können vertrauen und vielleicht auch mal die eine oder andere Funktion selbst schreiben, statt nach einer passenden npm-Abhängigkeit zu suchen. Auf Standards setzen, sich mit den reichhaltigen Funktionen von JavaScript und TypeScript auseinandersetzen und auch die ECMAScript-Proposals des TC39 im Blick zu behalten. Das wäre mein Wunsch an die JavaScript-Welt.

Sebastian Springer: Ein bisschen weniger Geschwindigkeit und mehr Verlässlichkeit wären schön, aber das Chaos und die Schnelllebigkeit charakterisieren die

Die Experten

Sandra Parsick
freiberufliche Softwareentwicklerin und Consultant

Wolfgang Weigend
Oracle Global Services Germany GmbH

Stephan Rauh
OPITZ CONSULTING

Michael Vitz
INNOQ

Falk Sippach
embarc Software Consulting GmbH

Christian Kaltepoth
ingenit GmbH & Co. KG

JavaScript-Welt nun einmal. Aber gerade, wenn es um Applikationen mit einer erwarteten Lebenszeit von fünf bis zehn Jahren geht, ist es nicht unbedingt erstrebenswert, dass sich die Frameworks und das Ökosystem gefühlt jedes Jahr ändern. Andererseits können wir uns als JavaScript-Entwickler glücklich schätzen, dass wir mit einer so lebendigen und aktiven Sprache arbeiten können. So wird man von der Sprache und den darauf aufsetzenden Lösungen schon fast gezwungen immer am Ball zu bleiben und sich stetig weiter zu entwickeln.

Daniel Mies: Ich warte weiter darauf, dass Web Components richtig durchstarten. Jetzt wo die Browser den passenden Support liefern, fehlt vielleicht nur noch eine Library oder ein Framework, das sich gegen die Platzhirsche React und Angular durchsetzen kann.

Java

EM: Die GraalVM bietet mittlerweile gerade für Java-Entwickler nahezu revolutionäre Möglichkeiten: Setzt du die GraalVM bereits ein und wenn ja, wofür? Falls nein, warum nicht?

Stephan Rauh: Leider nein. Mich fasziniert die GraalVM vor allem aus technischer Sicht: Nach Jahrzehnten gibt es erstmals einen neuen Compiler. Und der ist richtig gut geworden! Polyglotte Entwicklung und der AOT-Compiler lassen mein Herz höherschlagen. Aus Managementsicht sieht die Sache anders aus. Hier sind viele Fragen zu klären. Wie sieht es mit dem Lizenzierungsmodell aus? Welche Kosten rollen auf uns zu? Laufen unsere Programme wirklich mit GraalVM? Und lohnen sich die Vorteile finanziell? In großen Firmen stellen sich diese Fragen als erstaunlich komplex heraus und ziehen oft ein monate-, wenn nicht gar jahrelanges Projekt nach sich. Wir könnten aber GraalVM für den Showcase von BootsFaces nutzen: Dort bereitet uns die Nashorn-Engine jede Menge Kummer. GraalVM könnte das Problem lösen.

Thorben Janssen: Mit meinem Blog [2] und in meinen Beratungsprojekten habe ich mich auf Hibernate und JPA spezialisiert. Dadurch bin ich meist im Umfeld größerer Enterprise Anwendungen tätig. Dort ist die GraalVM selbst bisher noch kein Thema, aber es besteht großes Interesse an Quarkus. Quarkus setzt auf GraalVM auf und unterstützt viele der bereits im Enterprise-Umfeld verbreiteten Libraries. Es bietet dadurch eine interessante Plattform, auf der neue Konzepte und bestehendes Know-how genutzt werden können. Bisher haben wir Quarkus nur in einigen Projekten evaluiert, aber ich denke, dass wir 2021 die ersten auf Quarkus basierenden Services in Betrieb nehmen können.

Wolfgang Weigend: Die Fähigkeit vom GraalVM Native Image setze ich für Projekte mit Helidon und Micronaut ein. Mit den GraalVM-20.3-Enterprise-Tools sind der Micronaut-Projekt-Wizard, der GraalVM-Native-Image-Support zur Generierung von Micronaut Executables sowie der Maven- und Gradle-Support für Micronaut eingeflossen. Dazu enthalten die Tools der GraalVM 20.3 Enterprise auch die Visual Studio Code Extensions mit Syntax Highlighting, Java Code Completion, integriertem Java-Debugger und Polyglott-Debugger.

Michael Vitz: Ich persönlich setze die GraalVM bisher noch nicht ein, beobachte ihre Entwicklung aber aus der Entfernung. Besonders spannend finde ich in dem Kontext den verringerten Speicherverbrauch. Die verkürzte Startzeit und der polyglotte Support können relevant sein, sind es in meinen Projekten bisher aber nicht. Aktuell überwiegt bei mir jedoch noch der zusätzliche Aufwand, um zum Beispiel eine Spring-Boot Anwendung als Natives Image lauffähig zu machen, gegenüber dem Speicherverbrauch.

Falk Sippach: Wirklich einsetzen tue ich sie nicht, da habe ich in meinem Beratungsumfeld aktuell keine Berührungspunkte. Ich habe aber für kleinere Demos mal das GraalVM Native Image ausprobiert. Der Compile-Vorgang zieht sich ziemlich in die Länge, dafür startet das erzeugte Binärartefakt in wenigen Millisekunden, verbraucht sehr wenig Arbeitsspeicher und ist trotz eingepackter JavaVM von überschaubarer Größe. Das ist schon sehr beeindruckend.

Christian Kaltepoth: GraalVM ist definitiv eine sehr interessante Technologie, die sicherlich in den nächsten Jahren noch mehr an Bedeutung gewinnen wird. Ich selbst habe mit GraalVM natürlich auch schon experi-


Thorben Janssen
selbstständig


Michael Hofmann
Berater, Coach, Referent und Autor


Tim Zöller
ilum:e informatik ag


Dr. Holger Schwichtenberg
MAXIMAGO, IT-Visions.net


Daniel Mies
codecentric AG


Marc Teufel
Hama GmbH & Co. KG


Sebastian Springer
MaibornWolff

mentiert und sehr positive Erfahrungen gemacht. In echten Kundenprojekten ist GraalVM bei uns jedoch noch nicht zum Einsatz gekommen. Das liegt vor allem daran, dass unsere Kunden bezüglich der Technologieentscheidungen meist noch etwas konservativer sind und daher lieber auf eine klassische JVM setzen. Ich denke, dass GraalVM auf jeden Fall ein Schritt in die richtige Richtung ist und zukünftig sehr viel mehr Verbreitung finden wird. Allerdings glaube ich auch, dass es noch einige Zeit brauchen wird, bis GraalVM im Mainstream ankommt.

Michael Hofmann: Der aktuelle Fokus liegt zurzeit darauf, die Projekte in die Cloud zu bringen. Das allein bringt schon genügend Komplexität mit sich. Der Vorteil, den die GraalVM bietet, spielt hier noch keine Rolle. Im Gegenteil: Die Komplexität wird sogar noch erhöht. Solange nur wenige Services in der Cloud betrieben werden, sind die Kosten dafür auch noch nicht so gravierend. Der Hauptkostenfaktor in diesen Projekten liegt im Aufwand für den Shift in die Cloud. Das wird sich in absehbarer Zeit ändern. Dann wird es spannend sein zu sehen, ob die GraalVM mit all den Vorteilen und Kompromissen der richtige Ansatz sein wird, oder ob andere Ansätze den besseren Kompromiss darstellen.

EM: Was würdest du dir für die kommenden Java-Versionen in Sachen Features wünschen?

Wolfgang Weigend: Durch eine gut gefüllte Java-Feature-Pipeline, unter Einhaltung der Abwärtskompatibilität, wird die innovative Weiterentwicklung ständig vorangetrieben, mit dem Wunsch ein attraktives JDK-17-LTS-Release zu erstellen. Nach meiner Einschätzung halte ich es für angebracht, zusätzliche Betriebsmerkmale für Enterprise-Anwender anzubieten, unabhängig davon, ob die Anwendungen in On-Premise- oder Cloud-Native-Umgebungen betrieben werden. Auch eine engere Verzahnung bei der Kombination von Java und GraalVM wäre wünschenswert.

Falk Sippach: Eigentlich bin ich relativ zufrieden mit dem jetzigen Funktionsumfang von Java. Und es tut sich ja bereits eine Menge. Aus den Inkubatorprojekten (Amber, Valhalla, Panama, ...) schwappen immer mal kleinere Features in die aktuellen halbjährlichen Releases herein, zuletzt die Records, die Sealed Classes und Pattern Matching für *instanceof*. Das sind tatsächlich alles Vorarbeiten für das Pattern Matching, das in Java Einzug halten soll. Ich bin gespannt, wann das so weit sein wird.

Sandra Parsick: Mich würde es freuen, wenn Pattern Matching und Records 2021 den Previewstatus verlassen.

Stephan Rauh: So verkehrt finde ich die Sprache gar nicht. Zugegeben, TypeScript gefällt mir besser. Die Syntax ist deutlich entspannter und kompakter. Ich würde mir mehr syntaktischen Zucker für Java und seine Bibliotheken wünschen: Kurzschreibweisen für Arrays und Hashtables, funktionale Programmierung ohne den Umweg über die Streams, noch mehr Typinferenz.

Michael Vitz: Ich bin im Großen und Ganzen sehr zufrieden mit dem aktuellen Featureset und mir fehlt die Fantasie für noch weitere große neue Features. Die Menge an größeren Features, die sich bereits jetzt seit dem JDK 11 angesammelt hat und dann mit JDK 17 ab September genutzt werden kann, bringt Java jedoch einen guten Schritt weiter. Neben den ganzen Sprachfeatures bin ich vor allem gespannt, ob und wenn ja, was genau sich durch die mit Projekt Loom gebauten Fibers, im Grunde leichtgewichtige Threads, ändern wird.

Christian Kaltepoth: Ich muss zugeben, dass viele meiner Wünsche bereits in den vergangenen Versionen erfüllt wurden. Mit Switch Expressions, Text Blocks und natürlich Records sind inzwischen einige sehr interessante und für die Praxis relevante Features in aktuellen Java-Versionen verfügbar. Ich persönlich bin auch ein großer Fan der Local Variable Type Inference und somit des *var*-Schlüsselwortes, auch wenn sich viele Java-Entwickler daran nicht so recht gewöhnen können. Dadurch, dass ich auch häufig mit TypeScript zu tun habe, habe ich aber auch einige andere Sprachfeatures schnell zu schätzen gelernt, die sicherlich auch Mehrwert für Java-Entwickler bringen würden. Dazu zählt zum Beispiel der Safe Navigation Operator, der auch unter dem Begriff Optional Chaining bekannt ist. Ähnliches kann man zwar in Java mit der Verwendung von *java.util.Optional* auch erreichen, allerdings ist es schon deutlich angenehmer, wenn die Sprache einen defensiveren Umgang mit potenziellen Nullwerten unterstützt. In diese Kategorie fällt auch der Elvis-Operator, den Groovy schon seit vielen Jahren unterstützt.

Thorben Jansen: Aktuell gibt es mit R2DBC und Vert.x zwei konkurrierende Ansätze zum reaktiven Zugriff auf relationale Datenbanken. Ich würde mir wünschen, dass sich ein standardisiertes API, ähnlich dem bekannten JDBC-Standard, etabliert. Dabei ist es aus meiner Sicht nicht so entscheidend, ob dies als Teil des JDK, als offizielle Spezifikation oder als unabhängiges Projekt entwickelt wird.

Tim Zöller: Ich würde mir wünschen, dass die Verschmelzung der funktionalen Features, also Pattern Matching, Sealed Classes, Records, finalisiert wird. Das eröffnet die Möglichkeit für ganz neue Paradigmen. Auf Project Loom freue ich mich natürlich auch.

Die Fragen stellten Hartmut Schlosser, Maika Möbus, Dominik Mohilo und Ann-Cathrin Klose. Aus Platzgründen konnten nicht alle Antworten aller Autoren abgedruckt werden. Weitere spannende Einblicke aus dem Entwickler-Nähkästchen finden Sie auf entwickler.de.

Links & Literatur

[1] https://platform.uno/blog/announncing-uno-platform-3-0-linux-support-fluent-material-and-more/

[2] https://thorben-janssen.com

Eine Einführung für performantere und sicherere Frontends

Was ist der JAMstack?

Gerade bei contentlastigen Seiten und Anwendungen, bei denen Skalierung und Sicherheit eine große Rolle spielen, bietet der JAMstack einen Ansatz, der die Entwicklung einfacher und günstiger machen kann.

von Daniel Mies

Wie schon 2020 können wir auch in diesem Jahr davon ausgehen, dass das Thema JAMstack [1] weiter Fahrt aufnimmt und für zahlreiche Firmen interessant wird. Auf den folgenden Seiten sehen wir uns an, worum es sich beim JAMstack handelt und wie er unsere Seiten performanter und sicherer macht und gleichzeitig die Entwicklungserfahrung im Frontend verbessert.

Hierzu betrachten wir zunächst, was JAMstack eigentlich bedeutet und wo die Unterschiede zu klassischen Ansätzen liegen. Im Anschluss geht es dann um mögliche Einsatzgebiete, in denen der JAMstack seine Stärken ausspielen kann.

Für alle, die auch gleich Hand anlegen wollen, stelle ich dann noch einige Frameworks vor, mit denen sich Anwendungen entwickeln lassen. Und keine Angst: Fans von Angular, React und Vue müssen nicht direkt ein neues Framework lernen.

Was bedeutet JAM?

JAM ist ein Akronym und steht für JavaScript, APIs und Markup. Die Architektur sieht vor, vorab gerenderte, statische Seiten über ein Content Delivery Network (CDN) auszuliefern. Die Anwendungen werden über die Integration von APIs und JavaScript dynamisch gemacht. Zu den Technologien, die hier ihre Stärken ausspielen können, zählen die gängigen JavaScript Frameworks, Static-Site-Generatoren und spezialisierte APIs von Drittanbietern (z. B. für Authentifizierung [2], Abwicklung von Zahlungsdienstleistungen [3] oder headless Contentmanagementsysteme (CMS) [4]). Damit bekommen wir eine sehr gute Performance, eine höhere Security, einfache und vor allem günstige Skalierung und eine sehr gute Entwicklungserfahrung.

Viel von dem, was der JAMstack einführt, ist nicht neu und wird sicher im einen oder anderen Projekt schon erfolgreich genutzt. Der JAMstack ist daher erst mal eine klare Definition einer modernen Architektur für Webseiten und Apps. Im Folgenden sehen wir uns an, wie der JAMstack sich hier von klassischen Anwendungen abhebt. Die hier gemachten Bewertungen werden natürlich nicht pauschal jeder Anwendung gerecht.

Server-side Rendering

Fangen wir mit einem klassischen Ansatz an: Unser Server erhält eine Anfrage und führt dazu passende Operationen auf der Datenbank aus. Das Ergebnis der Datenbankoperation wird verarbeitet und dann passend für den Client gerendert. Der Client zeigt diese Seite dem Nutzer dann im Browser an. Beim Thema Performance hängen wir hier zunächst stark von der Leistung des Servers und den Datenbankabfragen im Hintergrund ab. Dies lässt sich über Caching natürlich noch verbessern. Da die erzeugte Seite statisches HTML ist, sollte zumindest dieser Part für den Nutzer äußerst performant sein (**Abb. 1**).

Skalierung bedeutet in diesem Fall oft, dass mehrere Server (und/oder Datenbanken) benötigt werden, um zu skalieren, was natürlich mit erhöhten Kosten verbunden ist. Gerade, wenn hier nur einzelne Teilsysteme unter Last sind, kann das unwirtschaftlich werden. Aufseiten des Backends könnte man nun zu einer Serverless- oder Microservice-Architektur greifen. Im Frontend passen diese Ansätze nicht immer.

Aus einer Securityperspektive müssen Server und Datenbank betrachtet werden, allerdings kann man hier natürlich auch auf eine der zahlreichen standardisierten Lösungen zurückgreifen. Aus Sicht der Entwickler kann dieser Ansatz zur Folge haben, dass Anwendungslogik und Darstellung miteinander vermischt werden. Bei anspruchsvollen Frontends ist die Entwicklungserfahrung oft besser, wenn das Frontend klar vom Backend getrennt ist.

Single Page Applications

Bei der Single Page Application verlagern wir mehr Logik in den Client. Dieser stellt die Seite dar und kommuniziert über REST APIs mit dem Server. Der Server verarbeitet die Anfragen, führt Operationen auf der Datenbank durch und liefert das Ergebnis in der Regel über JSON oder XML an den Client zurück, der die Seite entsprechend aktualisiert (**Abb. 2**).

Aus Performancesicht wandert ein Teil der Verantwortung vom Server in den Client. Der Server muss hier „nur noch“ das ermittelte Ergebnis in Form von JSON oder XML an den Client zurückgeben. Caching ist auch

Vorteile von CDNs

Der Einsatz von CDNs, um ein Frontend auszuliefern, wurde mit der Einführung des JAMstack populär. Aber was ist ein CDN und warum erhalten wir dadurch eine optimierte Performance und höhere Sicherheit?
Ein CDN ist eine Gruppe von Servern, die geografisch verteilt sind. Die Aufgabe des CDN ist es, statische Inhalte (wie zum Beispiel HTML, CSS, JavaScript, Bilder, ...) schnell zur Verfügung zu stellen. CDNs erfreuen sich hierbei großer Beliebtheit und sind für einen Großteil des weltweiten Traffics verantwortlich.
Die geografische Trennung heißt für den Nutzer, dass die Seiten schneller laden, da eine Seite nicht von einem Ursprungsserver abgerufen wird, sondern vom geografisch nächsten Server. Im Falle eines Serverausfalls durch einen Hardwarefehler oder falls Server durch einen Anstieg der Anfragen an ihre Grenzen stoßen, können andere Server aus dem Netzwerk einspringen.
Die Dateigröße statischer Daten kann über Minifizierung und Komprimierung optimiert werden, wodurch auch hier Traffic (und damit Kosten) und Geschwindigkeit verbessert werden können. Für Unternehmen werden hier die Betriebskosten reduziert und gleichzeitig das Nutzererlebnis verbessert.
Ein CDN muss man natürlich nicht selbst betreiben: Die großen Cloud-Anbieter haben hier Lösungen parat und Firmen wie Netlify und Vercel bieten diese an.

DVD INHALT

Das Entwickler Magazin im PDF-Format:
Die aktuelle Ausgabe sowie ein komplettes Jahresarchiv, Quellcodes und Beispiele zu den Artikeln im Heft.

Wenn Sie hier keine DVD finden, haben Sie die Profi-DVD nicht abonniert und verpassen eine Menge Extras!

Auf dieser DVD: Videos zu aktuellen Themen des Hefts, die Quellcodes aller Artikel und die spielbare Version des Spieleklassikers von Alexander Rudolph.

BASTA! Spring 2020: Event Storming
von Henning Schwentner

Alle reden über Event Storming, aber was steckt wirklich dahinter? Wenn die Methode richtig eingesetzt wird, macht das Event Storming nicht nur eine Menge Spaß, sondern liefert auch gleich wichtige Hinweise zur Implementierung. Henning Schwentner erklärt in diesem Talk von der BASTA!, wie das gelingt.

BASTA! Spring 2020: Introduction to Windows Containers and Docker
von Marcel De Vries

Docker ist eines der großen Trendthemen der letzten Jahre. Viele Entwickler erhoffen sich wahre Wunder davon! Aber lohnt es sich wirklich, auf Container umzusteigen? Dieser Frage geht Marcel De Vries in diesem Talk von der BASTA! nach. Er erläutert die wichtigsten Begriffe aus dem Container-Kontext und zeigt auf, wie sich die Technologie gewinnbringend einsetzen lässt.

iJS 2020: Enrich your Angular App with Angular Forms
von Fabian Gosebrink

In fast jeder Angular-Anwendung kommt man an den Punkt, an dem man Informationen vom Benutzer benötigt. Angular Forms ist eine großartige Lösung für diesen Fall. Angular Forms ermöglicht es, reichhaltige Informationen über die Personen zu erhalten, die mit der Seite interagieren, und bietet viele Möglichkeiten mit einer großen Vielfalt an Formularen. Aber Benutzereingaben, selbst in ihrer offensichtlich einfachsten Form, können sehr komplex werden: Felder müssen validiert werden, können komplexe Abhängigkeiten zueinander haben und sollten testbar sein. In diesem Talk von der iJS wirft Fabian Gosebrink einen Blick auf die Komplexität von Angular Forms und stellt verschiedene Lösungswege für typische Probleme vor.

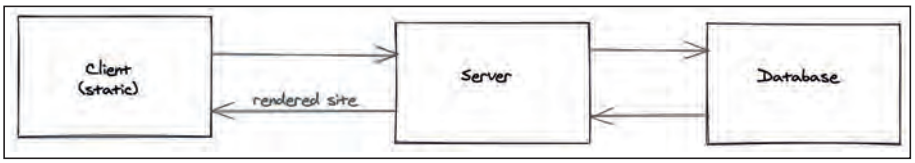


Abb. 1: Der Server interagiert mit einer Datenbank und rendert, passend zu den Anfragen, statische Webseiten aus

Abb. 2: Das Frontend ist eine Single Page Application und kommuniziert via REST mit dem Server

hier ein etabliertes Mittel, um die Performance zu verbessern. Das geht allerdings zulasten des Clients, der die im Einsatz befindlichen JavaScript Frameworks und Anwendungen laden und rendern muss. Das kann mit einer statischen Seite natürlich nicht mithalten und wird die Nutzererfahrung im Vergleich verschlechtern.

Bei den Themen Skalierung und Security bewegen wir uns in ähnlichen Dimensionen wie bei der zuerst beschriebenen Lösung. Durch Caching der REST APIs ist hier wahrscheinlich bei den meisten Anwendungen noch etwas mehr Spielraum gegeben, bevor zusätzliche Kosten entstehen. Nicht unterschätzt werden darf natürlich, dass ein Teil der Anwendungslogik in Frontend und Backend implementiert sein muss, was eine zusätzliche Quelle für Sicherheitslücken sein kann.

Die Entwicklungserfahrung ist im Frontend natürlich wesentlich besser. Es gibt zwar auch hier noch Projekte, in denen Backend und Frontend gemeinsam entwickelt werden, das Frontend ist allerdings klar(er) vom Backend getrennt. Gleichzeitig gibt es aber den bereits oben erwähnten Teil der Businesslogik, der in diesem Fall doppelt gepflegt werden muss.

JAMstack

JAMstack-Webseiten und -Anwendungen rendern alle Seiten vorab als statische HTML-Seiten und nicht für jeden Request einzeln, wie bei Server-side Rendering. Diese Seiten werden dann über ein CDN ausgeliefert. Wo notwendig werden APIs von Drittanbietern oder eigene APIs angebunden (**Abb. 3**). Natürlich haben wir hier zwei Dimensionen, die wir getrennt betrachten müssen: Frontend und Backend.

Im Frontend ist dieser Ansatz hochperformant: CDNs stellen sicher, dass die Seiten schnell ausgeliefert werden (Kasten: „Vorteile von CDNs“). Statische Seiten rendern naturgemäß schnell beim Nutzer. Die Performance ist, verglichen mit den klassischen Ansätzen, herausragend und auch das Thema Skalierung ist hier sehr interessant: CDNs bieten diese quasi von Haus aus und verglichen mit klassischem Hosting auch wesentlich günstiger. Wenn mehr Ressourcen benötigt werden, muss hier kein zusätzlicher (komplizierter) Server aufgesetzt werden, sondern es müssen lediglich statische Dateien verfügbar gemacht werden. Aus Securitysicht ist dies natürlich ebenfalls eine Verbesserung: Die Auslieferung von statischen Seiten ist einfacher abzusichern als der Betrieb eines Anwendungsservers.

Bei der Entwicklungserfahrung haben wir eine Trennung vom Backend erreicht. Die Anwendung kann mit dem gewünschten Toolset entwickelt werden. Das Backend wird nur über APIs angebunden. Es fällt direkt auf, dass der JAMstack in zwei Bereichen punkten kann:

- bei Seiten, die kein Backend brauchen, wo wir also den Content vorab rendern können und
- bei Microservices und Serverless-Anwendungen.

In Kombination mit diesen Lösungen ist der JAMstack besonders attraktiv, gerade wenn wir die Themen Performance und Skalierung betrachten. Für klassische Anwendungen wäre die Empfehlung, die eigene Anwendung auf Microservices bzw. Serverless zu migrieren, wahrscheinlich ein zu großer Schritt. Die Trennung von Backend und Frontend bringt dennoch auch so schon zahlreiche Vorteile bei der Entwicklung und vereinfacht die eigene Anwendung. Der Einsatz des JAMstack kann so der erste Schritt hin zu einer moderneren Architektur sein. Teile der eigenen Logik könnten in die Cloud ausgelagert werden (sei es als Microservices oder in Form von Serverless Functions). Ein weiterer Weg wäre die Nutzung der Domänenexpertise von Drittanbietern, um Bereiche wie Authentifizierung oder Zahlungen über diese abzuwickeln und dann auch direkt über das passende API aus dem Frontend heraus aufzurufen.

Natürlich gibt es aber nicht die eine Architektur, die alle Anforderungen abdeckt. Sehen wir uns im Folgenden an, wo der JAMstack seine Stärken ausspielt.

Anwendungsszenarien

Es gibt bereits zahlreiche Beispiele [5] für Seiten, die mit dem JAMstack umgesetzt wurden. Offensichtlich ist der Einsatz bei Seiten, die primär statische Inhalte darstellen. Angefangen beim Blog bis zur Nachrichtenseite, die hochverfügbar sein soll. Bei diesen Seiten führt die Nutzung von statischen Seiten selbstverständlich auch zu einer besseren Suchmaschinenoptimierung.

Der JAMstack nicht nur für statische Seiten geeignet. Dynamische Inhalte wie Kommentare können mit Hilfe von JavaScript und APIs eingebunden werden. Im Bereich der Contentmanagementsysteme gibt es zahlreiche Anbieter für headless CMS, wie zum Beispiel Contentful. Selbst für komplexe Vorgänge wie Zahlungen gibt es Lösungen auf dem Markt und auch Beispiele, wie diese mit Hilfe des JAMstack entwickelt werden können. Es bleibt dem Team überlassen, wie viele dynamische Inhalte über JavaScript und die passenden APIs in

eine JAMstack-Seite eingebunden werden. Die eigene Geschäftslogik kann dann initial weiterhin über den eigenen Server als API integriert oder in die Cloud migriert werden, um auch hier jederzeit einfach skalieren zu können.

Wenn nun die Anwendungsfälle passen und die Vorteile überzeugen, sollten wir uns mit dem Thema Entwicklung genauer auseinandersetzen. Für den JAMstack gibt es einige Best Practices, die ich im Folgenden beschreiben möchte.

JAMstack Best Pracices

JAMstack-Seiten benötigen keinen klassischen Server, um lauffähig zu sein. Auch wenn wir Seiten, die mit den passenden Frameworks erstellt wurden, auf einem klassischen Server deployen könnten, würde ich hier nicht mehr von JAMstack sprechen, da dieser eng mit dem Thema CDN verknüpft ist. Wie schon beschrieben, spielt der Einsatz eines CDNs eine erhebliche Rolle, um vom JAMstack zu profitieren. Aus Sicht der Entwicklung sollten die Vorteile einer separaten Frontend-Anwendung genutzt werden: Im JavaScript-Ökosystem gibt es zahlreiche Frameworks und Build-Tools, die die Erstellung von Webseiten vereinfachen und die Entwicklungserfahrung verbessern. Natürlich profitiert auch die Entwicklung im Backend, da hier der Fokus rein auf der Anwendungslogik und Bereitstellung von APIs liegt.

Gerade bei großen Seiten sollte das Thema „Atomic Deployments“ betrachtet werden. Bei umfangreichen Seiten, bei denen jeder Build mehrere hundert Seiten umfasst, kann eine Änderung zu Inkonsistenzen auf der Seite führen und wird wahrscheinlich auch verhältnismäßig lange dauern. Im Idealfall werden nur Änderungen deployt und nicht komplette Seiten.

Wie in der Einleitung versprochen wurde, müssen wir jetzt nicht alles neu lernen, um Anwendungen für den JAMstack zu entwickeln. Wir können sogar auf die von Single Page Applications bekannten Frameworks zurückgreifen.

Frameworks für den JAMstack

Natürlich ist die Verlockung groß, einfach eine SPA in einem CDN zu deployen, und wahrscheinlich ist das für bestehende Anwendungen auch ein guter erster Schritt. Für viele Anwendungen ist es aber nicht notwendig, ein entsprechendes Framework einzusetzen. Natürlich gibt es auch hier Abhilfe, um weiterhin das SPA Framework der Wahl einzusetzen, dem Nutzer aber statische Seiten zur Verfügung zu stellen.

Sehen wir uns zunächst die Lösungen für weitestgehend statische Seiten an. Hier gibt es zahlreiche Static-Site-Generatoren auf dem Markt (z. B. Jekyll [6], Hugo [7] oder Eleventy [8]), die statische Seiten erzeugen. Das kann über Markdown geschehen oder auch mit Hilfe eines headless CMS wie Contentful. Bei Bedarf kann hier auch JavaScript genutzt werden, um die jeweilige Seite dynamischer zu gestalten.

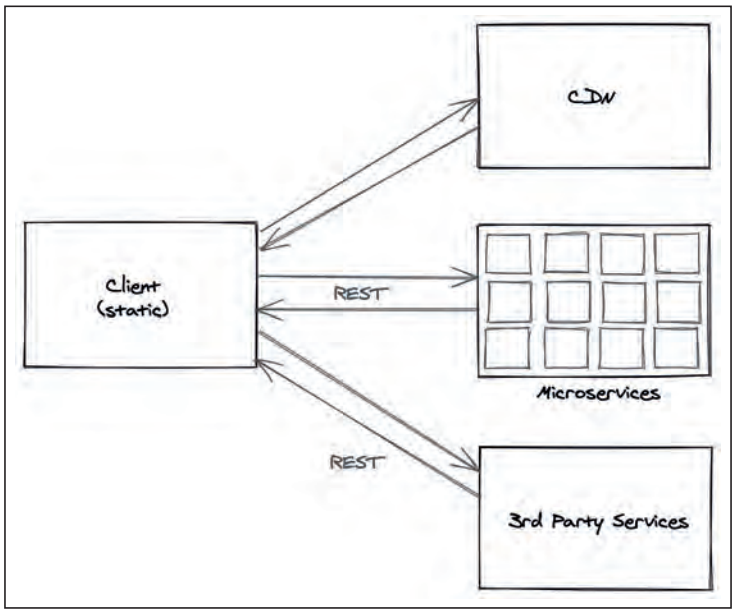


Abb. 3: Beim JAMstack wird das Frontend von einem CDN ausgeliefert

Für Anwendungen, die SPA-Funktionalität benötigen, können ebenso Frameworks genutzt werden, die auf React, Angular oder Vue basieren. In dem Fall bieten Frameworks wie Next.js (React) [9], Scully (Angular) [10] oder Nuxt (Vue) [11] die Möglichkeit, statische Seiten aus den SPAs zu erzeugen. Wenn diese Seiten im Browser geladen werden, wird das zugehörige JavaScript (z. B. die Runtime des Frameworks) nachgeladen und die Seite nachträglich um die Funktionalität aus der SPA erweitert (man spricht hier auch von Hydration). Das kombiniert die Vorteile der statischen Seite mit denen des jeweiligen Frameworks und bietet auf diese Weise eine hervorragende Nutzererfahrung.

Eine Anwendung mit Next.js erstellen

Wer jetzt eine JAMstack-Anwendung anlegen und deployen möchte, kann beispielsweise zu Next.js greifen. Next.js wird von Vercel [12] angeboten, einem Provider, der ebenfalls CDNs anbietet. Bei diesem Beispiel zeige ich aber, wie man das Ganze auf Netlify [13] deployen kann.

Die Einrichtung ist vergleichbar mit den Ansätzen von SPAs: Es gibt ein passendes npm-Paket, um die Anwendung zu erstellen. In diesem Fall reicht im Terminal

```
npm init next-app
```

Ein Dialog führt durch die Einrichtung. Danach muss in der *package.json* nur noch der *build*-Befehl angepasst werden, damit Next.js mit *next export* statische Seiten erzeugt.

```
"scripts": {
  // ...
  "build": "next build && next export",
},
```

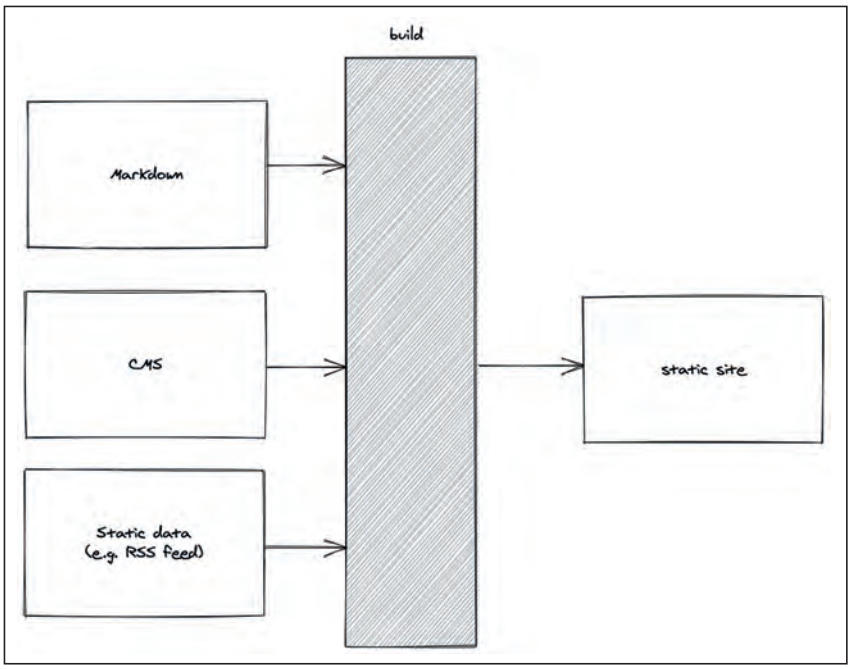


Abb. 4: Alle Seiten werden hier vorab gerendert

Wenn wir die Anwendung nun mit *npm run build* bauen, werden im *out*-Ordner die statischen Dateien angelegt, die wir später deployen werden.

Deployment

Während beim Server-side Rendering die Seiten zur Laufzeit erzeugt werden, werden sie in einer JAMstack-Seite vorab gerendert (**Abb. 4**). Hierzu können Quellen wie Markdown, CMS oder ein RSS-Feed während des Builds genutzt werden, um daraus die passenden statischen Seiten zu erzeugen. Diese Seiten werden im Anschluss im CDN veröffentlicht.

Die wenigsten Firmen werden ein eigenes CDN betreiben wollen, weshalb sich hier die Nutzung von passenden Anbietern empfiehlt. Unternehmen, die schon auf Cloud-Anbieter setzen, können auch hier den JAMstack nutzen. So lässt sich auf AWS [14] beispielsweise ein S3-Bucket mit Hilfe von CloudFront über ein CDN verteilen.

Mit Netlify und Vercel gibt es aber auch noch spezialisierte Anbieter, die die benötigte Funktionalität zur Verfügung stellen. So können Seiten bei Netlify deployt und über deren CDN verteilt werden. Netlify Functions bietet Serverless-Funktionen, um die eigene Anwendung zu erweitern.

Jeder größere Anbieter bietet hier inzwischen das notwendige Tooling, um die eigene Anwendung zu deployen. Wir stellen sicher, dass unsere Next.js-Anwendung bei GitHub verfügbar ist (Netlify ist natürlich nicht auf GitHub beschränkt) und können diese dann aus dem Terminal heraus mit Netlify verknüpfen.

Zunächst müssen wir das Netlify CLI via *npm install -g netlify-cli* installieren. Mit *ntl login* loggen wir uns bei Netlify ein (hier muss dann ggf. einmalig eine Registrierung im sich öffnenden Browser erfolgen). Im Verzeichnis unserer Seite reicht nun schon ein *ntl init*, um unsere Seite mit Netlify zu verknüpfen. Der Dialog, durch den man geführt wird, verknüpft das GitHub Repository mit Netlify und stellt sicher, dass nach jedem *push* auf den Main Branch die Seite bei Netlify aktualisiert wird. Für private und kleinere Projekte ist das kostenlose Paket bei Netlify übrigens vollkommen ausreichend.

Mehr ist nicht notwendig und schon kann die Seite weiterentwickelt werden. Es gibt zahlreiche Anleitungen im Netz die zeigen, wie statische Ressourcen in eine Next.js-Seite integriert werden können. Dieser Artikel soll nur als erste Einführung dienen.

Fazit

Mit dem JAMstack haben wir einen modernen Architekturansatz, der die Entwicklung nicht nur beschleunigt, sondern Entwicklern auch eine moderne Umgebung bietet. Nutzer profitieren von performanten Seiten, die hochverfügbar sind. Unternehmen können Kosten im Betrieb sparen und müssen sich weniger Gedanken um Security machen.

Es gibt zahlreiche Beispiele für den Einsatz des JAMstack. Auch wenn der Gedanke, alle Seiten vorab statisch zu generieren, anfangs ungewohnt erscheint, ist hier dennoch einiges möglich. Nicht nur contentlastige Seiten können davon profitieren. Mit modernen Frameworks rund um Next.js, Scully und Nuxt lassen sich die Vorteile der SPA einfach mit denen der statischen Seite verknüpfen.

Daniel Mies arbeitet als Senior IT Consultant bei der codecentric AG. Seine Leidenschaft gilt moderner Frontend Entwicklung und Webstandards. In seiner Freizeit podcastet er.

https://mies.me, https://ready-for-review.dev

Links & Literatur

[1] https://jamstack.org

[2] https://auth0.com

[3] https://stripe.com

[4] https://www.contentful.com

[5] https://jamstack.org/examples/

[6] https://jekyllrb.com

[7] https://gohugo.io

[8] https://www.11ty.dev

[9] https://nextjs.org

[10] https://scully.io

[11] https://nuxtjs.org

[12] https://vercel.com

[13] https://www.netlify.com

[14] https://aws.amazon.com

Ein- und Ausgabe über den Datenstrom

io.Reader und io.Writer in Go erklärt

Go-Anfänger sollten verstehen, wie das io-Paket der Sprache funktioniert. Welche Möglichkeiten bietet es und wie können diese im eigenen Code genutzt werden? Darauf geht dieser Artikel ein.

von Andreas Schröpfer

Nahezu jedes Programm verarbeitet während der Laufzeit Daten. Diese können dabei aus unterschiedlichsten Quellen stammen. Ein Server liest beispielsweise Einträge aus Datenbanken, ein Texteditor liest eine Textdatei vom Filesystem oder ein Browser lädt Webseiten per HTTP. Das Einlesen und Verarbeiten erfolgen dabei in der Regel sequenziell. Das bedeutet, dass Daten nacheinander eingelesen werden.

Dieser Artikel beschreibt, wie Datenströme in Go abgebildet werden und wie sie im Code verwendet werden können. Er baut auf die Grundlagen der Syntax auf und gibt einen kurzen Überblick über das io-Paket. Besonders Go-Anfänger sollten sich einmal mit diesem Paket beschäftigen, da dieses die wichtigsten Abstraktionen der Standardbibliothek beinhaltet. Denn über die Interfaces *io.Reader* und *io.Writer* lassen sich Daten einlesen und ausgeben. Idiomatischer Go-Code sollte diese Interfaces immer dann verwenden, wenn Daten verarbeitet werden. Das ist am Anfang ungewohnt, führt aber auf Dauer zu besseren Schnittstellen und flexiblerem Code. Die Verwendung von *io.Reader* und *io.Writer* zieht sich auch stringent durch die Standardbibliothek.

io.Reader: die Basics

Bevor wir uns näher mit den weiteren Features des io-Pakets befassen, wird hier zuerst die grundsätzliche Funktionsweise dieser Abstraktion betrachtet. Der *io.Reader* besitzt nur die *Read*-Methode. Sie hat als Input ein *[] byte* und als Output die Werte *n* und *err*.

```
type Reader interface {
  Read(p []byte) (n int, err error)
}
```

An dieser Stelle ist wichtig zu wissen, wie in Go ein Slice intern abgbildet wird. Dieses ist dabei ein Pointer auf ein Array. Die Variable *p* zeigt auf einen Bereich im Arbeitsspeicher, der innerhalb der *Read*-Methode beschrieben wird. Der Reader schreibt somit in das übergebene Slice hinein. Der Rückgabewert *n* beinhaltet dabei die Anzahl der Bytes, die geschrieben wurden, und *err* wird für das Melden von Fehlern verwendet.

Dieser Aufbau mag anfangs vielleicht ein wenig kompliziert anmuten, jedoch hat er einen entscheidenden Vorteil. Denn dadurch ist nicht der Reader dafür verantwortlich, Speicher für den Datenstrom bereitzustellen. Durch das Erzeugen von *p* definiert unser Code, wie groß die einzelnen Stücke des Datenstroms sind.

Sobald ein Datenstrom zu Ende ist, wird über *err* der Wert *io.EOF* (*end of file*) gemeldet. Gemäß Dokumentation dürfen danach keine weiteren Daten bei weiteren *Read*-Aufrufen geschrieben werden. Das bedeutet, dass ein Datenstrom aus einem *io.Reader* nur einmal ausgelesen werden kann.

Um zu verstehen, wie der *io.Reader* funktioniert, lesen wir einfach mal alle Daten aus einem Reader aus. Hierfür rufen wir die *Read*-Methode in einer *for*-Schleife auf. Diese Schleife wird beendet, sobald *io.EOF* als Fehler gemeldet wird. Die Daten werden in *buf* eingelesen. Da diese Variable ein Slice ist, zeigt es auf ein Array. Die Größe des Arrays wird durch die *make*-Funktion definiert. Sie legt im Hintergrund das Array an und reserviert damit entsprechenden Platz im Arbeitsspeicher. Zum Schluss wird die Anzahl der gelesenen Bytes und deren Inhalt als String ausgegeben. Mit *buf[:n]* wird die Ausgabe immer auf die gelesenen Bytes beschränkt (Listing 1).

Da diese Abstraktion so tief in der Standardbibliothek verwurzelt ist, muss in der Praxis die *Read*-Funktion so gut wie nie direkt aufgerufen werden. Wenn zum Beispiel alle Daten eines Readers gelesen werden müssen, kann *ioutil.ReadAll()* aus dem *ioutil*-Paket verwendet werden. Aber auch bei anderen Anforderungen wird der Reader einfach an andere Funktionen übergeben. Wie diese Aktionen aussehen, wird später hier im Artikel anhand der Funktionen des io-Pakets dargestellt.

Abb. 1: Dieser Gopher ist kein TeeReader, sondern ein TeaReader

io.Writer

Das Gegenstück zum Reader ist der Writer. Wie der Name bereits vermuten lässt beinhaltet dieses Interface nur eine *Write*-Methode:

```
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

Der Aufbau ähnelt hier dem des Readers. Die Methode liefert die Anzahl der geschriebenen Bytes und eine Fehlerwert zurück. Mit *p* werden die zu schreibenden Daten direkt übergeben. Dabei kann ein Writer beliebig oft aufgerufen werden. Die Daten werden einfach hintereinander gehängt. So ist es möglich, auch größere Datenmengen in kleinere Häppchen aufzuteilen.

io.Copy

Eine sehr häufig verwendete Funktion ist *io.Copy*, über die Daten aus einem Reader in einen Writer kopiert werden. So ist es z. B. möglich, die Daten aus einem File über den Standardoutput auszugeben, wie Listing 2 zeigt.

Der Standardoutput wird über *os.Stdout* abgebildet und implementiert das Writer-Interface. Eine Datei wird über den Typ *os.File* abgebildet und implementiert das Reader-Interface. Somit ist eine einfache Verwendung von *io.Copy* möglich. Das Copy liefert zusätzlich einen Fehlerwert und die Anzahl der geschriebenen Bytes. Wenn wir nun *n* und die Größe der Datei vergleichen würden, ließe sich feststellen, ob alle Bytes kopiert wurden. Da sowohl beim Lesen als auch beim Schreiben etwas schiefgehen kann, sollten wir den Fehler bei Copy nicht ignorieren.

TeeReader

Mit *TeeReader* ist kein Gopher wie in **Abbildung 1** gemeint, der bei einer Tasse Tee ein Buch liest, sondern eine Funktion. Diese hat als Eingangswerte einen Writer und einen Reader. Als Rückgabewert wird ein neuer Reader zurück geliefert. Sobald nun von diesem Reader gelesen wird, schreibt der *TeeReader* die Daten aus dem Input-Reader in den Writer.

Das Beispiel in Listing 3 verwendet einen *TeeReader*. Das Programm erzeugt für die Daten aus *r* einen md5-Hash. Dafür müssen die Daten nach *h* geschrieben werden. Diese Variable hat den Typ *hash.Hash*. Dieser Typ implementiert auch das Writer-Interface, wodurch *h* auch als *io.Writer* verwendet werden kann.

Zusätzlich wird der Inhalt aus *r* über *Stdout* ausgegeben. Ohne das Konstrukt *TeeReader* müsste hier zweimal aus *r* gelesen werden. Da das, wie bereits unter *io.Reader*-Basics beschrieben, nicht möglich ist, benötigt es eine andere Lösung. Hier kommt der *TeeReader* ins Spiel, denn genau für diesen Anwendungsfall wurde er erstellt. Der Aufruf *io.TeeReader(r, h)* kopiert die Daten von *r* nach *h*, sobald von dem neuen Reader *tee* gelesen wird. Das passiert eine Zeile später mit *io.Copy(os.Stdout, tee)*.

Eine Besonderheit des TeeReader ist, dass er keine Daten puffert. Die Daten werden direkt beim Lesen von *tee* in den Writer geschrieben, was sich insbesondere bei größeren Datenmengen höchst effizient gestalten lässt.

MultiReader

Grundsätzlich fügt ein *MultiReader* beliebig viele Reader zusammen. Dabei werden diese in der gleichen Reihenfolge aufgerufen, wie sie in die Funktion geschrieben wurden. Diese Funktion kann beliebig viele Reader aufnehmen, wie Listing 4 zeigt.

Eine praktische Anwendung wäre beispielsweise, Header, Body und Footer in einzelne Dateien zu gliedern. Über die Funktion des *MultiReader* können wir diese sehr komfortabel zu einem Reader zusammenführen.

LimitReader

Die Funktion *LimitReader()* ermöglicht es, dass nicht alle Bytes aus einem anderen Reader gelesen werden. Die Funktion benötigt als Input einen *io.Reader* und die zu limitierende Bytezahl. Als Rückgabe erhalten wir wiederum einen Reader:

```
func LimitReader(r Reader, n int64) Reader
```

Listing 1: Daten aus einem Reader über eine for-Schleife auslesen

```
func main() {
  r := strings.NewReader("Hallo ich lese hier Text aus")
  buf := make([]byte, 5)
  for {
    n, err := r.Read(buf)
    if err == io.EOF {
      break
    }
    fmt.Println(n, string(buf[:n]))
  }
}
```

Listing 2: Verwendung von io.Copy

```
func main() {
  fd, _ := os.Open("text.txt")
  defer fd.Close()
  n, err := io.Copy(os.Stdout, fd)
  fmt.Printf("%d Bytes kopiert\n", n)
  if err != nil {
    fmt.Println("io.Copy error: ", err)
  }
}
```

Listing 3: Gleichzeitige Erzeugung eines Hashwertes und Ausgabe der Daten mit dem TeeReader

```
func main() {
  r := strings.NewReader("Hallo, TeeReader!")
  h := md5.New()
  tee := io.TeeReader(r, h)
  io.Copy(os.Stdout, tee)
  fmt.Printf("\nmd5: %x", h.Sum(nil))
}
```

Der so eingeschränkte Reader hört nach der maximalen Größe mit einem *io.EOF* auf. Das ist wichtig, da ansonsten der Programmfluss solange unterbrochen sein würde, bis alle Daten gelesen wurden. Wenn wir beispielsweise einen Fileupload umsetzen und der Nutzer versucht, 10 GB hochzuladen, bricht dieser Reader den Lesevorgang nach der zuvor festgesetzten Größe ab.

Pipe

Mit *Pipe()* können Daten synchron verarbeitet werden. Bei der Verwendung muss bedacht werden, dass sowohl Auslesen als auch Schreiben in der Pipe gleichzeitig stattfinden. Durch diese Logik wird sichergestellt, dass alle gesendeten Daten auch beim Empfänger ankommen. Das bedeutet in der Praxis, dass der *PipeWriter* solange blockiert, bis alle geschriebenen Daten durch den *PipeReader* ausgelesen wurden. Dieses Prinzip kommt übrigens auch bei Channels zum Einsatz. Deshalb müssen wir einen Teil der Pipe in einer eigenen Goroutine aufrufen. Ansonsten kommt es zu einem Deadlock in unserem Programm.

Die Funktion *io.Pipe()* liefert beim Aufruf einen PipeReader und einen PipeWriter zurück (Listing 5). Nun muss entweder das Lesen oder das Schreiben in eine Goroutine gepackt werden. Der große Vorteil der Pipe ist die Methode *CloseWithError*, die sowohl *PipeWriter* als auch *PipeReader* besitzen. Sollte es nun innerhalb der Goroutine zu einem Fehler beim Lesen oder Schreiben kommen, können wir diesen einfach an *CloseWithError* übergeben. Damit wird dieser sofort an das andere Ende der Pipe übermittelt und an dieser Stelle ausgegeben.

Die Pipe ist sozusagen ein Ersatz für *io.Copy()*, jedoch mit dem kleinen Unterschied, dass Lesen und Schreiben synchron ablaufen und in unterschiedlichen Goroutinen stattfinden können. Die Pipe kümmert sich um die Synchronisierung zwischen der Goroutine des Readers und der Goroutine des Writers. Die *io.Pipe* ist somit, wie hier im Beispiel, eine Art Channel für Datenströme. Denn die *main*-Funktion läuft ja auch in einer eigenen Goroutine. Falls jetzt auf der Seite des Writers der Programmfluss blockiert wird, wartet die andere Routine. Sobald das Programm dort weiterläuft, werden die Daten auch wieder kopiert. Das passiert solange, bis mit *Close()* der *PipeWriter* geschlossen wird.

Listing 4: Zusammenführung mehrerer Reader über den MultiReader

```
func main() {
  r1 := strings.NewReader("Header\n")
  r2 := strings.NewReader("Body\n")
  r3 := strings.NewReader("Footer\n")

  r := io.MultiReader(r1, r2, r3)

  if _, err := io.Copy(os.Stdout, r); err != nil {
    fmt.Println(err)
  }
}
```

Seeker

Ein weiteres Interface im io-Paket ist der *Seeker*, der das Herumspringen innerhalb eines Objekts erlaubt. In unserem Kontext ist es das Herumspringen im Datenstrom. Mit *offset* kann gezielt ein Byte angesteuert werden.

```
type Seeker interface {
  Seek(offset int64, whence int) (int64, error)
}
```

Die zweite Variable *whence* erscheint Anfangs vielleicht etwas verwirrend. Denn diese Signatur definiert einen numerischen Wert. Welcher Wert nun welche Bedeutung hat, wird über die Konstanen des Pakets definiert (Listing 6).

Damit unser Code lesbar bleibt, sollten wir also nicht die Nummer, sondern die Konstante übergeben. Denn *foo.Seek(0, io.SeekCurrent)* ist verständlicher als *foo.Seek(0,1)*.

Listing 5: Einfache Verwendung der Pipe

```
func main() {
  r, w := io.Pipe()
  go func() {
    fmt.Fprint(w, "wir schreiben etwas\n")
    time.Sleep(2 * time.Second)
    fmt.Fprint(w, "weitere Daten\n")
    w.Close()
  }()
  if _, err := io.Copy(os.Stdout, r);
                                    err != nil {
    log.Fatal(err)
  }
}
```

Listing 6: Optionen des Seekers

```
const (
  SeekStart   = 0 // seek relative to the origin of the file
  SeekCurrent = 1 // seek relative to the current offset
  SeekEnd     = 2 // seek relative to the end
)
```

Listing 7: Mehrfaches Lesen einer Datei

```
func main() {
  fd, _ := os.Open("text.txt")
  defer fd.Close()
  io.Copy(os.Stdout, fd)
  fd.Seek(0, io.SeekStart)
  io.Copy(os.Stdout, fd)
}
```

Listing 8: Verwendung des SectionReader

```
func main() {
  r := strings.NewReader("some io.Reader stream to be read\n")
  s := io.NewSectionReader(r, 5, 17)

  if _, err := io.Copy(os.Stdout, s); err != nil {
    log.Fatal(err)
  }

}
```

ReadSeeker

Der *ReadSeeker* ist die konsequente Erweiterung eines Readers, der zusätzlich das *Seeker*-Interface implementiert. Das Lesen erfolgt dabei grundsätzlich wie bei jedem anderen Reader auch. Mit der *seek*-Methode gibt es jetzt allerdings zusätzlich die Möglichkeit, innerhalb des Datenstroms die Leseposition zu ändern. Dadurch ist es möglich, Daten mehrmals zu lesen.

Dateien werden dabei über den Typ *os.File* abgebildet. Dieser Typ implementiert auch die Seek-Methode. Somit implementieren diese *io.ReadSeeker*. In Listing 7 ist dargestellt, wie es nun möglich ist, Daten aus einer Datei mehrfach auszulesen.

Ein weiterer großer Vorteil dabei ist, dass flexibel in der Datei gesprungen werden kann, d. h., wir können auch problemlos einzelne Teile unabhängig auslesen. Insbesondere bei großen Dateien kann diese Funktionalität sehr hilfreich sein.

SectionReader

Mit der Funktion *NewSectionReader()* können wir einen Reader erstellen, der nur einen bestimmten Teil aus einem Reader liest. Wenn innerhalb einer Datei nur eine Sequenz benötigt wird, kann dieser Typ verwendet werden. Der *SectionReader* ändert dabei nicht die Leseposition des ursprünglichen Readers. Wenn also nur bestimmte Teile eines Datenstroms separat verarbeitet werden sollen, ist der *SectionReader* eine gute Lösung, wie Listing 8 zeigt.

WriteString()

Zuletzt soll auch noch eine hilfreiche Funktion zum Writer vorgestellt werden. Mit *WriteString()* ist es möglich, direkt Strings in einen Writer zu schreiben. Da diese Methode Bytes erwartet, wäre sie sonst nur mit einer Typkonvertierung möglich. Das würde zu folgender Lösung führen: *w.Write([]byte("Hallo Writer!"))*. Die Verwendung von *io.WriteString(w, "Hallo Writer")* stellt eine besser lesbare Lösung dar.

Zusammenfassung

Dieser Exkurs hat einen kurzen Überblick über die wichtigsten Elemente des io-Pakets gegeben. Die Funktionen und Typen erleichtern den Umgang mit dem Einlesen und Ausgeben von Daten. Bei Problemstellungen rund um dieses Thema lohnt es sich immer auch einen Blick in die Dokumentation [1] zu werfen. Denn das Paket beinhaltet für die gängigsten Anwendungsfälle eine elegante Lösung.

Andreas Schröpfer ist seit über zehn Jahren in der IT-Beratung tätig und seit 2015 begeisterter Gopher. Er ist Autor des Buches „Go – Das Praxisbuch“ und Contributor bei mehreren Open-Source-Projekten. Er gibt Workshops zu Go, ist Mentor bei Excercism und unterrichtet auch auf Udemy.

Links & Literatur

[1] https://golang.org/pkg/io/

Neues in der Major-Version

Angular 11: Entwickler im Fokus

Angular hat eine neue Major-Version erhalten, die vor allem den Entwickler von Angular-Projekten adressiert. Enthalten sind einige kleine Neuigkeiten, die aber große Wirkung haben. Auch Breaking Changes gibt es zu vermelden, allerdings hilft hier das CLI weiter.

von Karsten Sitterberg

Am 11.11. erschien Angular in Version 11. Die rund fünf Monate Arbeit seit dem letzten Major Release am 24. Juni und dem Minor Release 10.1 am 8. September hatten vor allem das Feedback der weltweiten Angular-Community im Fokus. So wurde unter dem Schlagwort „Operation Byelog" intensiv an den auf GitHub gemeldeten Issues gearbeitet, um diese zu kategorisieren und gezielt angehen zu können. Dieses Ziel wurde erreicht. Natürlich ist die Abarbeitung der Issues noch nicht abgeschlossen – ein Zeichen eines aktiven Open-Source-Projekts ist es ja gerade, dass die Community sich in Form von Issues einbringt. In Zukunft soll die initiale Bearbeitung neuer Pull Requests und Issues innerhalb eines Zeitfensters von maximal zwei Wochen erfolgen. Allein seit Mitte Oktober wurden bis zum Release von Angular 11 über 340 Issues als erledigt markiert (**Abb. 1**).

Auch das Thema Transparenz, ein häufig geäußerter Kritikpunkt, wird aktiv angegangen: So ist nun unter [1] die Roadmap des Angular-Projekts zu finden. Sie spiegelt die aktuelle Einschätzung und Planung zu den wichtigsten Themen aus Sicht der Angular-Core-Entwickler wider. Angular ist dank des reichhaltigen Ökosystems inzwischen mehr als nur das Framework im Zentrum. Umso wichtiger, dass auch externe Entwickler und Contributors die Möglichkeit bekommen, ihre Planung entsprechend auszurichten.

Das Major-Release Angular 11 bringt natürlich auch technische Neuerungen rund um die Plattform, das Angular CLI, Material Components und TypeScript mit. Alle Bereiche werden wir uns im Folgenden im Detail ansehen.

TypeScript

TypeScript entwickelt sich kontinuierlich weiter und Angular hält Schritt: Mit Angular 11 wird der Support für TypeScript 3.9 eingestellt, TypeScript 4.0 ist nun nötig. Das Update ist vor allem auch deshalb wichtig, da mit TypeScript 4.1 bald eine Optimierung ausgerollt werden soll, die den Angular-Compiler ngcc bis zu viermal schneller machen soll.

Darüber hinaus bietet TypeScript 4.0 einige weitere interessante Neuerungen. So wurde die Typinferenz bei TypeScript-Klassen verbessert: Auch wenn der Typ der Properties *area* und *sideLength* nicht explizit angegeben ist, kann TypeScript diese anhand der Wertzuweisungen im Konstruktor erkennen. So bekommt die Property *sideLength* den Typ *number* (Listing 1). Falls nicht alle Codepfade innerhalb des Konstruktors sicher zu einer Wertzuweisung führen, wird die Property als potenziell *undefined* markiert. Daher hat die Property *area* im Beispiel den Typ *number | undefined*.

Mit TypeScript 4.0 wurden auch neue (Zuweisungs-)Operatoren hinzugefügt. Wie aus vielen anderen Sprachen bekannt, gab es auch in TypeScript bisher schon Zuweisungsoperatoren wie *a+=b;* (entspricht *a=a+b;*) oder *a-=b;* (entspricht *a=a-b;*). Nun werden solche Operatoren entsprechend des ECMAScript-Standards auch für die Verknüpfungen &&, || und ?? hinzugefügt. Es gibt nun also die folgenden Zuweisungsoperatoren:

- *a &&= b* (entspricht *a && a = b*),
- *a ||= b* (entspricht *a || a = b*, kann auch geschrieben werden als *if(!a){ a = b; }*) und
- *a ??= b* (entspricht *a ?? a = b*).

Wichtig ist bei allen diesen Operatoren, dass die Zuweisung nur geschieht, wenn sie notwendig ist. Bei *a||=b* etwa geschieht die Zuweisung nur dann, wenn der Wert von *a* falsy (also etwa *0, null* oder *undefined*) ist.

Eine kleine Verbesserung wurde für den *catch(error)*-Block in *try-catch*-Statements eingeführt: Bisher war eine Typdeklaration des übergebenen *error*-Parameters nicht möglich, da er immer den Typen *any* hatte. Somit hatte TypeScript keine Möglichkeit, Checks auf Zugriffe durchzuführen. Nun ist es auch möglich, *unknown*

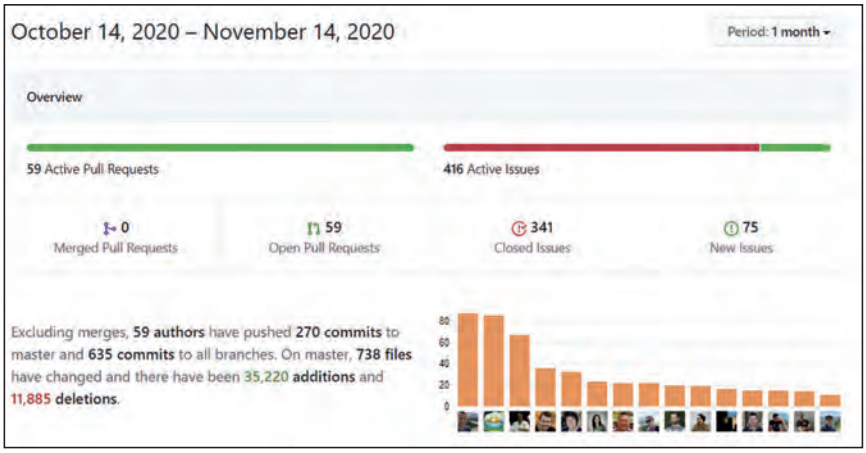


Abb. 1: Erledigte Issues

als Typ anzugeben. Das ist zwar kein definitiver Typ, allerdings erlaubt TypeScript Zugriffe auf *unkown*-Variablen nur dann, wenn sie durch einen Type Guard abgesichert werden. So kann an dieser Stelle Typsicherheit dazugewonnen werden, wie in Listing 2 zu sehen ist. In zukünftigen TypeScript-Versionen soll eventuell eine Compileroption hinzugefügt werden, mit der der *unknow*-Typ an dieser Stelle dann erzwungen würde.

Zudem wurde mit der neuen Type-Script-Version die Build-Geschwindigkeit und die Editorunterstützung durch den Language Service verbessert.

Mit der neuen Major-Version gab es auch einige Breaking Changes. So wurde *document.origin* entfernt, da dies ohnehin nur in alten IE und Safari-Versionen funktionierte. Stattdessen sollte *self.origin* verwendet werden. Zudem wird nun ein Fehler geworfen, wenn versucht wird, eine Objekt-Property mit einem *set*- bzw. *get*-Accessor zur überschreiben. Das gilt auch im umgekehrten Fall (Überschreiben eines Setters/Getters mit einer reinen Property).

Der *delete*-Operator konnte bisher auf alle Properties angewendet werden. In Zukunft kann *delete* nur noch angewendet werden, wenn die zu entfernende Property als optional gekennzeichnet ist.

Listing 1

```
class Square {

  // Bisher waren beide Properties vom
  // Typ 'any'
  sideLength; // Typ 'number'
  area; // Typ 'number | undefined'

  constructor(sideLength: number) {
    this.sideLength = sideLength;
    if (Math.random()) {
      this.area = sideLength**2;
    }

  }
}
```

Listing 2

```
try {
  // ...
} catch (e: unknown) {

  console.log(e.toUpperCase()); // Error: Object is of type 'unknown'.

  if (typeof e === "string") {
    // Type-Narrowing: 'e' ist vom Typ 'string'.
    console.log(e.toUpperCase());
  }
}
```

Update auf Angular 11

Wer das offizielle Angular CLI verwendet, ist beim Thema Update fein raus: Selbst wenn man nicht kontinuierlich auf die aktuelle Version migriert hat, wird man durch das Werkzeug bei Updates gut unterstützt. Bei einem kleinen Projekt ist das Update innerhalb weniger Minuten erledigt. Bei komplexen Projekten, die umfangreichen Gebrauch von den Angular APIs machen, ist typischerweise ein Aufwand von einer bis wenigen Stunden realistisch. Das Vorgehen ist gewohnt komfortabel und einfach. Das Angular CLI wird mit der *update*-Task aufgerufen:

```
ng update @angular/core @angular/cli
```

Danach sollten alle Tests ausgeführt werden, um potenzielle Regressionen aufzuzeigen. Ich empfehle, dass wirklich jeder in eine sinnvolle Testabdeckung investiert.

Wer sich über die Details eines Updates informieren möchte, findet dazu eine interaktive Anwendung unter [2]. Anhand der Ausgabe kann auch eine manuelle Migration auf Angular 11 unterstützt werden.

CLI

Nach dem Update auf Angular CLI 11 ist die erste und offensichtlichste Neuerung die aktualisierte Logausgabe der Befehle *ng serve* und *ng build*. Damit soll eine bessere Übersicht über die erzeugten Bundles und deren Größe geschaffen werden. Dabei werden die Grund-Bundles der Anwendung separat von den Lazy-Loading Bundles aufgeführt. Letztere sind nun nach Größe sortiert. In **Abbildung 2** und **3** sind die alte und die neue Ausgabe von *ng serve --prod* beispielhaft gegenübergestellt.

Werden innerhalb des Projekts Google-Fonts eingebunden, so werden diese nun als Teil des CLI Prod Builds als separater Link in die *index.html* eingebettet, wenn die Property *"optimization": true* in der *angular.json* gesetzt ist. Diese Optimierung erlaubt die Fonts parallel zum Anwendungsstart zu laden, und nicht erst als Teil der Anwendungs-Bundles. Damit reduziert sich praktischerweise auch der Umfang des Anwendungs-Bundles.

Wird eine Anwendung entwickelt, kann es vorkommen, dass das Set-up der Anwendung zur Entwicklungszeit ein anderes ist als das der Anwendung zur Produktionslaufzeit. Etwa wenn Backend und/oder Frontend zur Produktionslaufzeit in einer Umgebung laufen, die spezielle Sicherheitsfeatures erfordert zum Beispiel eine CSP oder die Types moderner Browser.

Beide Features werden vom Browser aber nur aktiviert, wenn der Server spezielle Header mitschickt. Da im Angular CLI Dev Server aber keine Custom Header gesetzt werden konnten, konnten diese Features bisher nicht leicht zur Entwicklungszeit simuliert werden. Mit Angular CLI 11 gibt es nun die Möglichkeit, Custom Header in der *angular.json* als *headers*-Property innerhalb der *architect.build.options* zu hinterlegen. Die *headers*-Property ist dabei eine Key-Value-Map, in der Key der Name des Headers ist und Value der entsprechende Wert. So lässt sich beispielsweise ein CSP-Header auf folgende Weise spezifizieren:

```
"headers": {
  "Content-Security-Policy": "default-src 'self'"
}
```

Die Möglichkeit, solche Header zu setzen, wurde v. a. in Hinblick auf die sogenannten Trusted Types eingebaut. Diese sind ein neues Sicherheitsfeature in Browsern und sollen Cross-Site-Scripting-Attacken noch effektiver verhindern. Auch Angular 11 nutzt sie, wenn möglich.

Ein Feature, das speziell Library-Autoren die Arbeit erleichtern soll, sind die sogenannten Declaration-Maps. Diese sollen es Editoren und IDEs ermöglichen, bei Nutzung von „Go to Definition“ direkt in den (lokalen) Library-Quellcode zu springen und nicht in die gebauten Library-Dateien innerhalb des */dist*-Ordners.

Falls mit der neuen Angular-CLI-Version zwei Angular-Anwendungen parallel gestartet werden, beschwert sich Angular nun nicht mehr wie früher über einen bereits belegten Port. Stattdessen wird der Entwickler nun direkt gefragt, ob die Anwendung auf einem anderen freien Port gestartet werden soll. Der freie Port wird dabei zufällig aus der Menge der freien Ports gewählt.

Wird mit dem Angular CLI ein neues Projekt angelegt, so wurde vom CLI schon bisher immer abgefragt, ob z. B. ein Style Preprocessor wie SCSS verwendet und Angular-Routing vorkonfiguriert werden soll. Mit Version 11 kommt nun ei neuer sogenannter Prompt hinzu, der fragt, ob die App im strikten Modus angelegt werden soll. Dadurch werden zum einen die strikten Type Checks von TypeScript (*"strict":true* in der tsconfig) aktiviert, etwa die Strict-Null-Checks oder die Strict Property Initialization. Zum anderen führt der Angular-Compiler diese strengen Checks auch im Template durch.

Das Angular CLI unterstützt sogenannte Schematics. Diese Schematics sind hilfreiche Tools, um beispielsweise neue Komponenten oder ganze Libraries per einfachem Kommandozeilenbefehl automatisch generieren zu lassen. Mit Angular 11 wurden diese Schematics so erweitert, dass per *ng generate resolver* ein neuer Router-Resolve erzeugt wird.

Außerdem setzt es zur Testausführung auf den Karma Test Runner. Um die für den Testing-Workflow wichtigen Code-Coverage-Reports generieren zu können, nutzt das

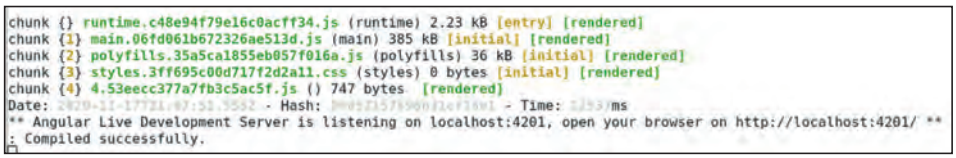


Abb. 2: Alte Ausgabe von ng serve –prod

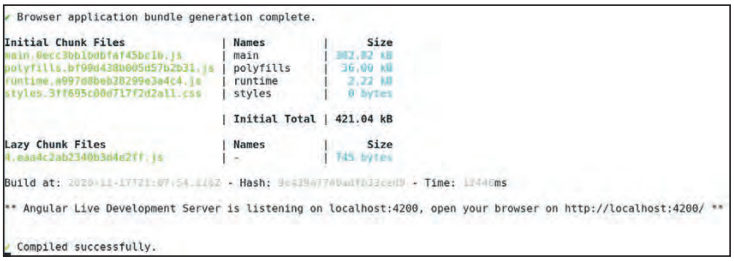


Abb. 3: Neue, geordnete und übersichtliche Ausgabe von ng serve –prod

Angular CLI bisher den karma-coverage-istanbul-reporter. Mit dem Angular CLI 11 werden neue Projekte nun allerdings mit dem besser unterstützten Paket karma-coverage generiert. Bestehende Projekte werden zwar (noch) nicht migriert, aber zumindest wird eine Deprecation-Nachricht bei Nutzung des alten Reporters ausgegeben.

Hot Module Replacement (HMR)

Während der Entwicklung sind kurze Zyklen zwischen Quellcodeanpassungen und Ergebnisbewertung wünschenswert. Schon jetzt kann das Angular CLI den Browser informieren, wenn Änderungen stattgefunden haben. Der Browser lädt dann autonom die Anwendung neu. Das spart einige Klicks oder Shortcuts, die Anwendung befindet sich danach jedoch wieder im Initialzustand. Um das zu verbessern, bietet webpack das Feature „Hot Module Replacement“ [3] an. Dabei können bestimmte Änderungen, wie zum Beispiel im globalen CSS-Styling, direkt live angewendet werden. Das spart natürlich viel Zeit und macht die Entwicklung angenehmer.

Nun bestehen die meisten Änderungen jedoch nicht aus Style-Anpassungen. Mit HMR ist das Ziel, dass der Anwendungszustand auch bei Änderungen der Komponenten weitestgehend erhalten bleibt und kein voller Reload erfolgt. Dazu gehören standardmäßig erst einmal die Scroll-Position im Fenster und Formulareingaben. Da Angular CLI unter der Haube auf webpack setzt, war auch in der Vergangenheit die Nutzung von HMR möglich. Dazu waren jedoch Anpassungen an der Konfiguration und dem Anwendungscode erforderlich. Mit Angular 11 ist das nun nicht mehr erforderlich, um eine grundsätzliche HMR-Unterstützung zu erhalten – beim Start ist lediglich der Schalter *--hrm* zu setzen. Damit sieht das vollständige Kommando beispielsweise so aus:

```
ng serve --hmr
```

Da sich dadurch das Verhalten der Anwendung verändert, wird auch ein entsprechender Hinweis ausgegeben:

```
NOTICE: Hot Module Replacement (HMR) is enabled for the dev server.
```

Aktuell bleibt der Anwendungszustand nicht in allen Konstellationen erhalten. HMR ist in jedem Fall ein vielversprechendes Feature und birgt großes Potenzial, sobald der volle Funktionsumfang im Kontext von Angular umgesetzt ist.

Abschied von TSLint

Um Codequalität besser analysierbar zu machen, gibt es sogenannte Linting-Tools, die den Quellcode etwa auf Einhaltung von Best Practices hin überprüfen. Ein solcher Linting-Vorgang kann im Angular CLI mit *ng lint* gestartet werden. Der bisher von Angular-CLI genutzte Linter war TSLint im Zusammenspiel mit dem Codelyzer-Plug-in. Mittlerweile wurde TSLint allerdings von den Projektverantwortlichen als deprecated markiert, die Migration auf ESLint wird offiziell empfohlen. Daher geht nun auch das Angular CLI diesen Schritt: TSLint wird als deprecated markiert und das Angular-Team arbeitet zusammen mit der Community an einer möglichst automatischen Migration der CLI-Projekte zu ESLint.

Listing 3: Build Angular-10-Projekt „Remote Training“

```
chunk {} runtime.43e2b8a007f700481.js (runtime) 2.53 kB [entry] [rendered]
chunk {1} 1.84010c5af42650386d40.js () 26 kB  [rendered]
chunk {2} 2.e9a861b71b633965b3b8.js () 16 kB  [rendered]
chunk {3} main.7fda6eefa7e683cc3ca1.js (main) 825 kB [initial] [rendered]
chunk {4} polyfills.8456b03b9e12a8e8.js (polyfills) 36 kB [initial] [rendered]
chunk {5} styles.3c95f0a9d119125db70f.css (styles) 68.1 kB [initial] [rendered]
chunk {6} 6.f3ee68e8823eec12d047.js () 881 kB  [rendered]
chunk {7} 7.b97f54a1e17c0375e958.js () 136 kB  [rendered]
chunk {8} 8.6fd325ff961e99e0808e.js () 4.78 kB  [rendered]
chunk {9} 9.26554fc7d61c8c809b39.js () 45.4 kB  [rendered]
chunk {10} 10.d6534ccfa9246c4adb20.js () 4.73 kB  [rendered]
chunk {11} 11.81cfc9377af48e428fbc.js () 2.21 kB  [rendered]
chunk {12} 12.bbb91bc2f292c14a7367.js () 11.8 kB  [rendered]
chunk {13} 13.3a5fb0fd7914837b0c8c.js () 1.16 kB  [rendered]
chunk {14} 14.a3b89a1c7ac7c0a5b08d.js () 2.79 kB  [rendered]
chunk {15} 15.b61a35f235d989c99a73.js () 93 bytes  [rendered]
chunk {16} 16.013f95e86cc15a0c515b.js () 3.22 kB  [rendered]
chunk {scripts} scripts.eaa39cabe4928.js (scripts) 1.57 MB [entry] [rendered]
Date: 2020-11-15T21:05:09.170Z - Hash: 77566713ce72558de04b - Time: 26732ms
```

Listing 4: Build Angular-11-Projekt „Remote Training“

```
Initial Chunk Files                | Names      |  Size
scripts.e97c506d3885f99c38a9.js    | scripts    |  1.57 MB
main.d09be28430557c5a3a88.js       | main       | 783.82 kB
styles.57d88b82d19b1de143f6.css    | styles     |  68.61 kB
polyfills.b798e2cd31320f43ed2f.js  | polyfills  |  36.00 kB
runtime.397a60f455e8ebb34f70.js    | runtime    |  2.42 kB

                                   | Initial Total |  2.44 MB

Lazy Chunk Files                   | Names      |  Size
5.044d6812ea6aa19f7c91.js          | -          | 135.79 kB
7.1e8a11aa598b7c849312.js          | -          |  46.65 kB
1.ca17a65df4afeb9bbdf4.js          | -          |  25.86 kB
10.8f6df508978a29dc7637.js         | -          |  11.82 kB
6.a9f19928649b8564995b.js          | -          |  4.77 kB
8.82788de94f5ae3bd26db.js          | -          |  4.73 kB
12.1d361f8ef7f203f1de3f.js         | -          |  2.79 kB
9.766b01f7f382aa133fba.js          | -          |  2.20 kB
11.eed7e9f8396eb884650a.js         | -          |  1.17 kB

Build at: 2020-11-15T21:23:35.729Z - Hash: f7c7ef39f09c24e - Time: 25253ms
```

Build-Zeit und Bundle Size

Jedes Update des Angular CLI begleitet der Wunsch nach einem schnelleren Build mit kleineren Artefakten im Ergebnis. Dabei besteht zwischen den beiden Anforderungen ein Spannungsfeld: Um kleinere Artefakte zu erzeugen, wird in der Regel der Build-Aufwand steigen, um intensiveres Tree Shaking und andere Optimierungen durchzuführen. Das wirkt sich negativ auf die Build-Zeit aus. Die Release Notes von Angular sprechen von deutlichen Verbesserungen durch ngcc (Faktor 2–4) und die aktuellere TypeScript-Version. ngcc wird verwendet, um npm-Pakete, die aus Kompatibilitätsgründen standardmäßig noch im alten View-Engine-Format publiziert werden, in das neue Ivy-Format zu übersetzen. Dieser Konvertierungsaufwand fällt typischerweise nur einmal an, da die Ergebnisse gecacht werden. Daher profitieren von diesen Verbesserungen in erster Linie CI Server, die so konfiguriert sind, dass sie jeden Build komplett mit einem leeren Projekt starten.

Im Folgenden wurde anhand von zwei aktuellen Kundenprojekten das Verhalten als *production*-Build gemessen. Für eines der Projekte konnte auch die Preview von webpack 5 getestet werden, da dort keine Web Worker verwendet wurden. Mehr zu webpack 5 folgt im nächsten Abschnitt. Betrachten wir zunächst die Ergebnisse in den Listings 3 bis 7.

Zu erkennen ist, dass sich die Build-Zeit zwischen Angular 10 und 11 nicht deutlich verändert. Kleinere Schwankungen bewegen sich im Rahmen der Messgenauigkeit. Auch die Bundlegrößen ändern sich nicht signifikant. Im Gegensatz dazu zeigt die Preview von webpack 5 deutlich den noch frühen Entwicklungsstand. Hier verändern sich sowohl die Build-Zeit als auch die Bundlegrößen negativ.

Vorschau auf webpack 5

Angular soll zukünftig auf webpack 5 umgestellt werden. Schon im Angular CLI 11 wird es die Möglichkeit geben, die neue Version zu verwenden. Dies dient dazu, erste Erfahrungen zu sammeln und auch mögliche Bugs frühzeitig melden zu können. Und in der Tat ist die aktuelle Umsetzung als Vorschau zu bezeichnen: Es wird zwingend Yarn als Paketmanager benötigt, und noch sind nicht alle Angular-Features kompatibel. So funktionieren zum Beispiel Web Worker noch nicht mit webpack 5. webpack 5 selbst, auch erst seit einigen Wochen freigegeben, bringt viele, zum Teil inkompatible, Änderungen mit sich.

Auf der anderen Seite verspricht webpack aber auch immense Verbesserungen: Durch einen persistenten Cache der Builds soll die Geschwindigkeit vor allem bei

wiederholten Builds deutlich verbessert werden. Die Bundles werden durch verbesserte Tree-Shaking-Algorithmen kleiner und die Codegenerierung wurde optimiert. Das Caching der Build-Ergebnisse wurde durch verbesserte Standardeinstellungen und geschickte Aufteilung der im Browser dauerhaft cachebaren Anteile verbessert.

Die Preview von webpack 5 kann im Angular CLI aktiviert werden, indem die folgende Einstellung in der *package.json* vorgenommen wird:

```
"resolutions": {
  "webpack": "5.4.0"
}
```

Module Federation

Ein besonders vielversprechendes Feature, das durch webpack 5 erschlossen wird, ist Module Federation. Damit wird es möglich, gemeinsamen Code zwischen voneinander unabhängigen Anwendungen zu teilen. Bisher musste dazu eine gemeinsame Library extrahiert werden, die sich dann in den Build-Ergebnissen beider Anwendungen wiederfand. Entsprechend wurde der gemeinsame Code auch mehrfach geladen. Mit webpack 5 soll es möglich werden, dass sich Module aus unterschiedlichen Builds und sogar unterschiedlichen Anwendungen wie Teile aus einem großen zusammenhängenden Graphen auffassen lassen. Damit können dann auch Abhängigkeiten aus anderen Quellen bezogen werden. Redundantes Laden derselben Abhängigkeit für verschiedene Anwendungen kann vermieden werden. Aufgrund der Komplexität kann das Thema im Rahmen dieses Beitrags lediglich angerissen werden und verdient in jedem Fall eine eigenständige Betrachtung.

Angular CLI Docker Image

Passend zum aktuellen Angular CLI stehen auch die von trion.de entwickelten Docker Images für den Build mit Angular 11 zur Verfügung. Die auf Docker Hub publizierten Images sind auf minimale Größe optimiert und erlauben neben dem eigentlichen Build mit dem Angular CLI auch die Ausführung von Karma-Unit-Tests und Protractor-E2E-Tests auf einem CI-Server. Folgende Images stehen zur Verfügung:

- trion/ng-cli [4] als minimales Build-Image, auch als ARM64-Image
- trion/ng-cli-karma [5] für Build und Angular-Unit-Tests
- trion/ng-cli-e2e [6] bringt Java für Protractor mit

Details zur Verwendung finden sich auch im Artikel auf JAXenter [7].

Angular Framework

In Angular 11 hat sich viel unter der Haube getan, um die Abarbeitung der Roadmapthemen vorzubereiten. So wurde dieses Release neben den bereits oben angesprochenen Build-Verbesserungen genutzt, um viele Deprecations der vergangenen Versionen durch Entfernung der entsprechenden Features aufzulösen. Einige nicht-breaking Features betreffen etwa den Compiler, der jetzt besser mit nichtvalidem Templatecode umgehen kann, etwa bei unvollständigen HTML-Tags. Für die Angular-PWA-Unterstützung gibt es das *@angular/service-worker*-Paket. Der Service Worker wird typischerweise genutzt, um die Offlinefähigkeit einer App zu erreichen. Dabei nimmt der Service Worker die Aufgaben des Cache wahr. Wenn bestimmte Assets einer Seite vom Browser aus dem Cache entfernt werden, aber z. B.

Listing 5: Build Angular-10-Projekt „Mobile Marketing“

```
chunk {} runtime.acf1f05ab39abc62fff7.js (runtime) 2.28 kB [entry] [rendered]
chunk {1} main.f5398d0a43ef90b15fa7.js (main) 443 kB [initial] [rendered]
chunk {2} polyfills.fa70ca150ae09dfd08a8.js (polyfills) 36.1 kB [initial] [rendered]
chunk {3} styles.7284b12cc0babc86e5d1.css (styles) 7.77 kB [initial] [rendered]
chunk {4} 4.ee4f58db44f19937f96b.js () 2.22 kB  [rendered]
chunk {5} 5.820f8ca212f4b8c30487.js () 105 kB  [rendered]
chunk {6} 6.4a76565d61df08843019.js () 1.6 kB  [rendered]
Date: 2020-11-19T20:08:37.614Z - Hash: 2fee8d19f9de74a4fd32 - Time: 16348ms
```

Listing 6: Build Angular-11-Projekt „Mobile Marketing“

```
Initial Chunk Files                | Names       |  Size
main.8107a14698d0bd82e78d.js       | main        | 438.35 kB
polyfills.63615701763a13869c01.js  | polyfills   |  36.00 kB
styles.7284b12cc0babc86e5d1.css    | styles      |   7.77 kB
runtime.682d939701041a13a436.js    | runtime     |   2.27 kB

                                   | Initial Total | 484.39 kB

Lazy Chunk Files                   | Names       |  Size
5.6038ec46ed64c4afe42c.js          | -           |  99.23 kB
4.528f682123022a5b6f59.js          | -           |   2.22 kB
6.9578413d7188f2995dbc.js          | -           |   1.60 kB

Build at: 2020-11-19T20:17:06.945Z - Hash: 3b898d74c9d72fee6b0b - Time: 15629ms
```

Listing 7: Build Angular-11-(webpack-5-)Projekt „Mobile Marketing“

```
Initial Chunk Files                | Names       |  Size
main.1dd1e520974bbbcca267.js       | main        | 480.49 kB
polyfills.6ccfce096debe4f0a331.js  | polyfills   |  38.79 kB
styles.ddd5330196eb11be102d.css    | styles      |   7.77 kB
runtime.6d8d1027a071439d0621.js    | runtime     |   3.44 kB

                                   | Initial Total | 530.47 kB

Lazy Chunk Files                   | Names       |  Size
552.7934c1452c58d7952579.js        | -           | 105.43 kB
297.7c3d900aa923a5dcb53c.js        | -           |   2.30 kB
437.4428e7dbed4aefe1459a.js        | -           |   1.67 kB

Build at: 2020-11-19T20:21:58.205Z - Hash: 2906ecca18ba441c0f17 - Time: 23270ms
```

wegen einer Serveraktualisierung nicht mehr vom Server zu laden sind, konnten früher Darstellungsfehler auf einer Seite entstehen, oder die Seite konnte nicht mehr komplett geladen werden. Ein solcher Fehler lässt sich nur durch Reload der Seite beheben. Um das an die App zu kommunizieren, wurde der *UnrecoverableStateError* als Fehlerzustand eingeführt.

Um Performance und Bundlegrößen zu optimieren, wurden im Router einige Codestellen angepasst, sodass bestimmte Fehlermeldungen nur noch zur Entwicklungszeit im Code bestehen. Der Fehler, der ausgegeben wird, wenn zweimal *Routermodule.forRoot()* aufgerufen wird, ist eher für Entwickler interessant, nicht aber in Produktion. Mit Angular 11 können diese Fehlermeldungen nun im Prod Build durch Tree Shaking entfernt werden.

Breaking Changes

Eine wichtige Breaking Change ist natürlich die Umstellung von TypeScript 3.9 auf TypeScript 4.0. Nach der Deprecation in Angular 10 ist eine andere, nicht unwesentliche Breaking Change das Ende der offiziellen Unterstützung von IE 9, IE 10 und IE Mobile durch Angular. Da diese Browser auch durch ihren Browserhersteller nicht mehr unterstützt werden, sollte das aber in der Regel kein Problem sein. Das Ende des Supports stellt allerdings Ressourcen frei, die nun für andere, wichtige Angular-Themen genutzt werden können.

Mit dem Paket @angular/platform-webworker hat das Angular-Team versucht, den Renderingprozess zwecks Performanceverbesserung in einen separaten Prozess, den sogenannten Web Worker, auszulagern. Da sich damit jedoch nicht die erhofften Verbesserungen erzielen ließen, wurde das Projekt mit Angular 8 deprecated und nun entfernt. Das bedeutet aber ausdrücklich nicht, dass man Web Worker nicht mehr mit Angular nutzen kann. Tatsächlich bietet das Angular CLI sogar ein spezifisches Schematic an, um eigene Web Worker zu generieren: *ng generate webWorker <name>*. Diese Worker können dann rechenintensive Aufgaben übernehmen, die ansonsten den JavaScript-Renderingthread blockieren würden.

Eine wichtige Renderingplattform, die Angular neben dem Browser unterstützt, ist das serverseitige Rendering mit @angular/platform-server. Dieses Paket kann auf unterschiedliche Arten genutzt werden. Beispielsweise lässt sich einstellen, dass der Renderingprozess mit absoluten oder relativen URLs durchgeführt wird. Wird die absolute Variante (*useAbsoluteUrl*) verwendet, ist es nun notwendig, auch die *baseUrl* anzugeben, mit der die absoluten URLs aufgebaut werden sollen. Das war vorher nicht der Fall und konnte zu undefiniertem, schwer zu debuggendem Verhalten auch in der Produktion führen. Im Folgenden wird noch auf einige wichtige Änderungen thematisch sortiert eingegangen. Die

wesentlichen Breaking Changes werden durch die *ng update*-Migrations automatisch behandelt.

Pipes und Internationalization

Pipes werden in Angular genutzt, um Transformationen wie Datums- oder Zahlenformatierungen im Template performant und auf Change Detection optimiert durchzuführen. Mit Angular 11 wurde die Typisierung einiger Pipes verbessert und insgesamt konsistenter gestaltet. So wird in allen Number Pipes und der Date Pipe jetzt explizit in der Signatur angegeben, welche Übergabetypen die Pipe akzeptiert. Vorher wurde bei falschen Übergabetypen in der Regel nur eine *RuntimeException* geworfen.

Die Date Pipe hat auch noch einen Breaking Change im Detail erfahren: Wurden der Pipe bisher Date-Time-Strings übergeben, die Bruchteile einer Millisekunde enthielten, wurden diese bisher auf die nächste Millisekunde gerundet. Das ist notwendig, da das Standard-Date-Objekt des Browsers nicht mit Bruchteilen einer Millisekunde umgehen kann. Das Runden zur nächsten Millisekunde konnte aber ungewollte Effekte an Tagesgrenzen haben (23:59:59,9999 Uhr wurde zu 0:00:00 am nächsten Tag). Von nun an werden Submillisekunden daher standardgemäß einfach abgerundet.

Die Uppercase/Lowercase Pipes erlauben nun keine falsy Werte wie *0*, *false*, *NaN* mehr, sondern werfen stattdessen eine Exception. Die Werte *null* und *undefined* werden zu *null*. Mit diesem Change wurden Signatur und Verhalten der Pipes aneinander angepasst. Ebenso wurde die Signatur der *async* Pipe angepasst: auch wenn der Pipe *undefined* übergeben wird, gibt sie *null* zurück. Bisher war in der Signatur angegeben, dass die Pipe in diesem Fall *undefined* zurückgab. Das widersprach aber dem tatsächlichen Verhalten. Die *slice* Pipe wiederum hatte bisher dieses Verhalten. Sie wurde nun so abgeändert, dass sie, wie auch die anderen Pipes, *null* zurückgibt, wenn ihr *undefined* übergeben wird.

Die Signatur der *KeyValue* Pipe wurde so angepasst, dass sie nun das eigentliche Verhalten widerspiegelt: Wurde die Pipe z. B. genutzt, um ein Objekt in ein Key-Value-Array zu verwandeln, wurde der Key dabei schon immer in einen String verwandelt, auch wenn der Key im ursprünglichen Objekt eine Number war. Das spiegelt sich nun auch in der Signatur der Pipe wider und gilt nur für normale Objekte. Wird die *KeyValue* Pipe etwa mit einer Map verwendet, so bleibt die *number* erhalten.

Die meisten Angular Pipes bieten die Möglichkeit, die Ausgabeformate zu lokalisieren, etwa die *number*, *date* und *currency* Pipe. Dazu werden *Locale*-Daten benötigt, die i. d. R. je Land unterschiedlich sind. Um Inkonsistenzen zu vermeiden, wurde dieses Locale Data API nun so angepasst, dass es die Daten nicht mehr in Form von veränderlichen (mutable) Arrays, sondern in Form von Read-only-Arrays zurückgibt. Ist eine Änderung der Lo-

cale-Daten nötig, müssen die Arrays mittels *slice()* kopiert und dann in der Anwendung selbst verwaltet werden.

Router

Der Router ist in Angular dafür zuständig, die Nutzer durch die App zu navigieren und diese Navigationsvorgänge durch URL-Wechsel sichtbar zu machen. Durch Eingabe des URL kann mit dem Router in dedizierte Anwendungsteile gesprungen werden. Im Router wurde vor allem fehlerhaftes Verhalten aufgeräumt, das bereits seit einigen Versionen als deprecated markiert wurde. So wurde die Property *preserveQueryParams*, die bei einem Routingvorgang dafür gesorgt hat, dass die Queryparameter im URL erhalten blieben, komplett entfernt. Stattdessen soll die eigentlich dafür gedachte Property *queryParamsHandling* genutzt werden, die zu diesem Zweck auf den Wert *preserve* eingestellt werden muss.

Bei der Konfiguration des Routers mittels *.forRoot()* kann das Verhalten des Routers auf unterschiedliche Arten konfiguriert werden. So kann mit der Property *initialNavigation* eingestellt werden, ob der Router beim Start der Anwendung eine initiale Navigation durchführen soll. Die möglichen Werte dieser Property wurden mit Angular 11 eingeschränkt, sodass jetzt noch die Werte *enabledNonBlocking*, *enabledBlocking* oder *disabled* möglich sind. Der Unterschied zwischen dem ersten und zweiten Wert ist, dass *enabledBlocking* zuerst die initiale Navigation ausführt und erst danach den App-Start zu Ende bringt (wird für Server-Side-Rendering benötigt). Bei *enabledNonBlocking* hingegen findet der initiale Routingvorgang statt, nachdem die Root-Komponente geladen wurde und blockiert somit nicht den App-Start. Die alten Optionen (*true, false, legacy_enabled, legacy_disabled*) wurden entfernt. Der Default ist jetzt *enabledNonBlocking* und entspricht dem bisher gewohnten Verhalten.

Eine weitere Einstellmöglichkeit, die angepasst wurde, ist die *relativeLinkResolution*. Mit der Einstellmöglichkeit *corrected* wurde ein Fehlverhalten des Routers gefixt, dass bei Navigation mit relativen Links innerhalb von Komponenten mit leerem Pfad (*path:''*), aber Kindrouten aufgetreten ist. Mit Angular 11 wird nun *corrected* zum Default. Das alte (fehlerhafte) Verhalten ist noch unter der Einstellmöglichkeit *legacy* vorhanden.

Eine wichtige Änderung ist auch für alle passiert, die ihre eigene *RouteReuseStrategy* entwickelt haben. Die *RouteReuseStrategy* ermöglicht das Wiederverwenden von Komponenten zwischen verschiedenen Navigationsvorgängen. Wenn also von Route */a* nach Route */b* und dann wieder nach */a* navigiert wird, kann durch eine geeignete *RouteReuseStrategy* erreicht werden, dass die Komponente *A* bei der letzten Navigation nicht neu erzeugt werden muss, sondern wiederverwendet wird (also in der Zwischenzeit nicht mit Aufruf von *ngOnDestoy* de-initialisiert wurde). Das Standardverhalten ist, dass Komponenten zwischen den Routingvorgängen zerstört und bei erneuter Navigation auf die Seite neu erzeugt werden. Die Änderung in Angular 11 bezieht sich nun auf die *shouldReuseRoute()*-Methode der *RouteReuseStrategy*: Bisher wurden der Methode bei Kindrouten die momentane und die zukünftige Route vertauscht übergeben. Dies wurde nun angepasst und sollte bei einem Update auf Angular 11 berücksichtigt werden.

Forms

Die eingeschränkte Typsicherheit bei Verwendung der Angular Forms ist schon länger ein Anliegen der Community. Mit Angular 11 wird in einem ersten Schritt die Typsicherheit in Bezug auf die Validatoren verbessert: Bisher wurden die Validatoren und Async-Validatoren in den *@angular/forms*-Direktiven als Typ *any[]* erzeugt. Jetzt werden die passenden Typen verwendet (*ValidatorFn[]*, bzw. *AsyncValidatorFn[]*).

Innerhalb eines verschachtelten Formulars ist es möglich, per *AbstractFormControl.parent* eine Referenz auf die umgebende *FormGroup* zu besorgen. Ist keine umgebende *FormGroup* vorhanden, konnte dieser Wert immer auch *null* sein, wie es nun auch der Typ von *AbstractFormControl.parent* abbildet. Zudem wurde das Verhalten von *FormControl/FormGroup/FormArray* mit asynchronen Validatoren korrigiert: Wurden diese schon bei der Initialisierung der Forms mit angegeben, hat das *statusChanges*-Observable nicht gefeuert, sobald die asynchrone Validierung das erste Mal abgeschlossen wurde. Das wurde nun umgestellt, sodass das Status-Change-Observable jetzt auch diesen initialen Status-Change emittiert.

Core

Angular hat sein Komponentenmodell schon von Beginn an eng an die Komponenten aus der Web-Components-Spezifikation angelehnt. Das drückt sich auch in der Möglichkeit aus, Style Encapsulation von Angular-Komponenten zu ermöglichen. In Angular ist das Standardverhalten dabei eine Emulation des Shadow-DOM über HTML-Attribute, um Kompatibilität mit Browsern zu gewährleisten, die das Shadow-DOM nicht unterstützen. Es gab bisher jedoch auch zwei Möglichkeiten, Komponenten so einzustellen, dass das native Shadow-DOM genutzt wird. Einmal die *ViewEncapsulation.Native*, dies hat die Anbindung einer alten Spezifikation des Shadow-DOM-Standards ermöglicht. Diese Option war schon länger deprecated und wurde nun entfernt. Stattdessen gibt es nun nur noch *ViewEncapsulation.ShadowDom*, mit der die neuere Version der Spezifikation für Angular-Komponenten aktiviert werden kann.

Wenn Webanwendungen entwickelt werden, sollte – wie bei allen anderen Anwendungen auch – ein Thema nicht vernachlässigt werden: Testing. Tests erhöhen die Resistenz des Codes gegenüber Bugs, vor allem wenn Änderungen oder Updates am Code durchgeführt werden. Da gerade Unit-Tests auch eine Form der Wiederverwendung des Codes darstellen, verbessern Tests darüber hinaus auch die Wartbarkeit des Codes. Aufgrund der asynchronen Eigenschaften des Browsers kommt es mitunter vor, dass man dies auch in den Unit-Tests berücksichtigen muss. Um dennoch einen korrekten Testablauf zu gewährleisten, stellt Angular Tools für solche

asynchronen Tests zur Verfügung. Hier ist zum Beispiel die *async()*-Funktion zu nennen, die sicherstellt, dass der nächste Testfall erst gestartet wird, wenn alle asynchronen Tasks im aktuellen Testfall abgeschlossen sind. Sowohl in IDEs als auch bei den Entwicklern hat der Name der Funktion aber häufig zu Verwechslungen mit dem ECMAScript *async*-Keyword geführt. Daher wurde die Funktion in der letzten Angular-Version umbenannt in *waitForAsync()* und die Verwendung als *async()* deprecated. Mit Angular 11 wurde der Name *async()* nun entfernt. Mit einer Migration werden alle noch vorhandenen *async()*-Aufrufe in *waitForAsync()*-Aufrufe umgewandelt.

Angular Material und CDK

Auch bei der Komponentenbibliothek Angular Material hat sich einiges getan. So existiert nun für jede Material-Komponente ein Test-Harness. Das verbessert die Qualität zwar nicht unmittelbar, erleichtert durch die höhere Abstraktion jedoch Tests. Davon abgesehen stehen auch viele kleinere Features parat, die zum Beispiel die Theming-Fähigkeiten der Material-Komponenten *mat-step*, *mat-vertical-stepper* und *mat-horizontal-stepper* durch einen *color*-Input verbessern. Die *MatSelect*- und *MatAutoComplete*-Komponenten bieten nun die Möglichkeit, das generierte Overlay mit Hilfe der Eigenschaft *overlayPanelClass* um eigene CSS-Klassen anzureichern.

Im CDK wurden u. a. die Fähigkeiten der Test-Harnesses ausgebaut, die mit der *parallel()*-Funktion nun die Möglichkeit unterstützen, mehrere Aktionen parallel gegen eine zu testende Komponente abzufeuern. Auch erlaubt die Funktion *manualChangeDetection()* eine präzisere Kontrolle des Testablaufs im Zusammenspiel mit Test-Harnesses.

Um die vielfältigen Einsatzmöglichkeiten des CDK noch besser zu unterstützen, wurden auch an diesen Komponenten Erweiterungen vorgenommen. Etwa kann das (Schließ-)Verhalten von *ConnectedOverlay*-Direktiven bei Nutzerinteraktionen konfiguriert werden (per *disableClose*-Input). Für Tabellen steht nun die Input-Property *fixedLayout* zur Verfügung, die aktiviert werden kann und dafür sorgt, dass die Tabellenspalten die gleiche feste Breite bekommen.

Eine hilfreiche Erweiterung im CDK sind die Coercion-Utilities, die auch im Anwendungscode hilfreich sein könnten. Diese nehmen einen bestimmten Wert entgegen und wandeln ihn in den geforderten Typ um. So gibt es etwa die Funktion *coerceBooleanProperty()*, die einen beliebigen Wert entgegennimmt und ihn in einen booleschen Wert umwandelt. Da die Coercion-Utilities zum Parsen von Komponenteninputs gedacht sind, wird z. B. der String *false* in den booleschen Wert *false* umgewandelt. Mit Angular 11 ist nun das Utility *coerceStringArray()* hinzugekommen, das einen Wert – oder auch ein Array von Werten – in ein Array von Strings verwandelt.

Die Zukunft von Angular

Angular ist eine inzwischen erwachsene Plattform. Das Versprechen, Entwickler bei der Umstellung von AngularJS zu unterstützen, wurde gehalten. Upgrades aktueller Versionen sind Dank des Angular CLI für Projekte jeder Größe gut zu bewerkstelligen. Die sehr gute Performance und das durchgängige, durchdachte Konzept machen Angular weiterhin zum Framework der Wahl für mittlere und große Anwendungen. Frameworks wie etwa Vue oder React liefern jedoch ein attraktives Modell für kleinste Anwendungen bzw. einzelne Komponenten. Dieser Bereich kann von Angular aktuell nicht optimal bedient werden. Das wirkt sich nicht nur auf den Einsatzzweck aus, sondern auch auf die Lernkurve: Ist ein „Hello World“ bei React oder Vue schnell erstellt, gilt es bei Angular viele Begriffe und Konzepte zu erlernen. Geht es darum, Entwicklerherzen zu gewinnen, spielen langfristige Wartbarkeit und Testbarkeit eine geringere Rolle als das erhebende Gefühl von schnellem, positivem Feedback.Zukünftig soll es möglich sein, Angular auch ohne das Modulsystem der *NgModules* einzusetzen. Dazu soll der Verzicht auf Zone.js optional möglich sein – wichtig, wenn Angular Elements als reiner Lieferant von Komponenten statt einer zusammenhängenden Anwendung eingesetzt werden sollen.

Aus eigener Projekterfahrung mit Angular, Vue und React kann ich sagen, dass Angular sich auch für kleinere Anwendungen gut eignet. Da Anwendungen oftmals dazu tendieren, zu wachsen, lohnt sich langfristig die Investition in ein sauberes Projekt-Set-up. Hier habe ich bei Angular dank des Opinionated-Ansatzes, Dependency-Injection und sehr guter Testbarkeit deutlich bessere Erfahrungen gemacht als mit Vue oder React. Wenn die Ziele der Roadmap erreicht werden, dann wird Angular nochmals flexibler einsetzbar sein.

Das nächste Angular-Release als Minor Increment ist für den 13. Januar 2021 geplant. Am 10. Februar 2021 folgt dann Angular 12, wenn alles wie geplant läuft. Man darf gespannt sein, was bis dahin mit den Themen webpack 5 und auch rund um Angular Elements passiert.

Karsten Sitterberg ist als freiberuflicher Entwickler, Trainer und Berater für Webtechnologien und Java tätig. Seine Schwerpunkte liegen im Bereich Frontend-Architektur und Microservices. Karsten ist Oracle-zertifizierter Java Developer. In Münster hat er die Frontend-Freunde als Meetup und die Java User Group mitgegründet.

https://sitterberg.com **@kakulty**

Links & Literatur

[1] https://angular.io/guide/roadmap

[2] https://update.angular.io

[3] https://webpack.js.org/guides/hot-module-replacement/

[4] https://hub.docker.com/r/trion/ng-cli/

[5] https://hub.docker.com/r/trion/ng-cli-karma

[6] https://hub.docker.com/r/trion/ng-cli-e2e

[7] https://jaxenter.com/build-and-test-angular-apps-using-docker-132371.html

iJS

International
JavaScript Conference

HYBRID EDITION

April 19 – 22, 2021
London or online

JavaScript is one of the core technologies and an essential part of web applications. The proliferation of its comprehensive frameworks and libraries helped JavaScript programming to improve its practices. Today, not only web developers, but virtually every business trusts in the power of high-level frameworks like Angular, React and Node.js to name a few.

At the International JavaScript Conference you will listen to inspiring talks, get insider tips and gain deep insights from our internationally known speakers and industry experts.

@JavaScriptCon #iJScon

Presented by: entwickler.de

Organizer: S&S Media Group

Until February 11

- ✓ PlayStation Classic Mini or C64 Mini for free
- ✓ Save up to £455
- ✓ Group discount

Conference Tracks

Angular · **React** · **Node.js** · **JavaScript Practices & Tools** · **Progressive Web Apps**

Exclusive Specials until February 11

Early Bird Special: Save up to £455 **until February 11**.

PlayStation Classic Mini or C64 Mini for free: Buy a 3-Day pass **until February 11** and get a PlayStation Classic Mini or C64 Mini for free.

Group Discount: Receive an additional **10% discount** when registering with 3+ colleagues.

Extra Specials: Freelancers and employees of scientific institutions benefit from individual offers – if you're interested, just send an email to **contact@javascript-conference.com**.

YouTube International JavaScript Conference

javascript-conference.com

Kolumne:
Per Anhalter durch den JavaScript-Dschungel

von Sebastian Springer

Der Weg zur Unverwundbarkeit

In JavaScript wird immer häufiger über Unveränderbarkeit oder englisch Immutability gesprochen beziehungsweise geschrieben. Natürlich sind wir alle davon überzeugt, dass diese Immutability eine sehr gute Idee ist – aber warum denn eigentlich? Und wie wird's gemacht? Worauf muss ich achten?

Bevor wir uns jetzt ins eigentliche Thema stürzen, sehen wir uns die Theorie dahinter an. Dabei geht es vor allem um das Typsystem von JavaScript und darum, wie die Engine mit den verschiedenen Typen umgeht. Und ja, JavaScript hat ein Typsystem, wenn auch nur ein schwaches. In JavaScript gibt es zwei unterschiedliche Arten von Typen: primitive und composite Types. Primitive Typen sind beispielsweise Strings, Zahlen oder Booleans. Composite Typen sind Arrays und Objekte. Der wichtigste Unterschied zwischen beiden Arten ist, dass sie primitive Typen by-value und composite Typen by-reference zuweisen. Am besten wird der Effekt, den die Zuweisungen haben, deutlich, wenn wir uns in Listing 1 ein konkretes Beispiel ansehen.

Sebastian Springer ist JavaScript-Entwickler bei MaibornWolff in München und beschäftigt sich vor allem mit der Architektur von client- und serverseitigem JavaScript. Sebastian ist Berater und Dozent für JavaScript und vermittelt sein Wissen regelmäßig auf nationalen und internationalen Konferenzen.

Der Code definiert zunächst zwei Variablen: eine Zeichenkette und ein Objekt. Die Funktion *mutate* weist zwei Parameter auf: eine Zeichenkette und ein Objekt. Die Funktion verändert beide Argumente. Die Zeichenkette, die ja by-value übergeben wird, ersetzt sie durch den Wert *Good morning* und beim Objekt ändert sie die *name*-Eigenschaft auf den Wert *Peter*. Zur Überprüfung geben wir beide Werte auf der Konsole aus. Im Beispiel rufen wir schließlich die Funktion auf und schreiben beide Werte auf die Konsole. Die Ausgabe des Scripts sehen Sie, egal ob Sie es im Browser oder über Node.js ausführen, in Listing 2.

Die erste Ausgabe ist die Zeichenkette *Good morning*, das erste Argument weist also einen neuen Wert auf, genauso wie die *name*-Eigenschaft des übergebenen Objekts. Die Ausgaben außerhalb der Funktion sehen etwas anders aus: Die primitive Variable bleibt unverändert auf dem ursprünglichen Wert und das Objekt hat die Modifikation aus der Funktion übernommen. Der Grund hierfür ist, dass Sie beim Aufruf der Funktion beim primitiven Wert eine Kopie des Wertes an die Funktion und beim composite Type lediglich eine Referenz auf den Speicherbereich des Objekts übergeben. Das bedeutet, dass eine Änderung in der Funktion das ursprüngliche Objekt modifiziert. Diese Eigenart von JavaScript könnten Sie sich theoretisch zunutze machen und auf Rückgabewerte von Funktionen verzichten. Das Problem an dieser Stelle ist, dass eine Funktion auf diese Weise ihre Umgebung verändert, ohne dass es aus ihrer Signatur direkt hervorgeht. Die Manipulation ist also ein Seiteneffekt der Funktion. In Einzelfällen können solche Seiteneffekte noch in Ordnung sein, wächst eine Applikation allerdings oder wird die Funktionalität in eine eigenständige Bibliothek ausgelagert, sind Seiteneffekte nicht mehr angebracht, da sie ein großes Fehlerpotenzial mit sich bringen.

An dieser Stelle können wir uns natürlich die Frage stellen: Kann man nicht etwas gegen die Mutability von Objekten unternehmen?

Objekte durch Einfrieren gegen Veränderungen schützen

Die kurze und einfache Antwort ist: Ja, es ist möglich, ein Objekt vor Veränderung zu schützen. Der JavaScript-Standard sieht für solche Fälle die Methode *Object.freeze* vor. Ihr übergeben Sie das Objekt, das Sie schützen möchten und anschließend sind keine Änderungen mehr möglich. Für unser Beispiel bedeutet das, dass Sie wie in Listing 3 die Zeile *Object.freeze(obj)* einfügen müssen.

Das Resultat dieser Änderung ist etwas gewöhnungsbedürftig. Der Versuch der Manipulation des Objekts wird einfach ignoriert, die JavaScript-Engine gibt keinerlei Rückmeldung in Form einer Exception. Jetzt könnte man das Ganze noch einen Schritt weitertreiben und prüfen, ob es sich bei dem Objekt um ein schreibgeschütztes Objekt handelt. Hierfür können Sie die Methode *Object.isFrozen()* verwenden. Dazu müssten Sie aber bei allen Änderungen davon ausgehen, dass das Objekt potenziell schreibgeschützt ist. Also ist das auch keine Option. Die wirkliche Alternative liegt in einer anderen Form von Immutability.

Immutable Datenstrukturen

Die Idee von immutable Datenstrukturen ist, dass Sie nicht, wie im Beispiel, auf der ursprünglichen Datenstruktur arbeiten, sondern eine Kopie erzeugen und diese verändern. Jetzt fragen Sie sich wahrscheinlich, ob es wirklich eine so gute Idee ist, Objekte wieder und wieder zu kopieren, statt einfach einen Wert zu ändern. Bei genauer Betrachtung ist dieser Lösungsansatz tatsächlich etwas speicherintensiv. Es gibt jedoch ein sehr gutes Argument dafür, Objektstrukturen trotzdem zu kopieren. Stellen Sie sich eine Objektstruktur mit mehreren Ebenen Tiefe vor und, dass Sie in einer dieser tiefen Ebenen eine Änderung vornehmen. Gehen wir jetzt von einem Algorithmus aus, der die grafische Oberfläche bei jeder Änderung der Objektstruktur aktualisiert. Damit dieser korrekt arbeiten kann, müsste er alle Äste des Objektbaums auf Änderungen überprüfen. Kopieren Sie jedoch die gesamte Objektstruktur, weiß der Algorithmus, dass eine Änderung stattgefunden hat. Und genau nach diesem Prinzip, wenn auch deutlich eleganter, arbeitet React mit seiner State-Verwaltung.

Doch zurück zu unserem Beispiel: Wenn Sie innerhalb der Funktion eine Kopie erzeugen, müssen Sie auch dafür sorgen, dass die Kopie an den aufrufenden Code zurückgegeben wird, damit dieser damit arbeiten kann (Listing 4).

Im Beispiel sehen Sie, wie Sie mit dem Spread-Operator, also den drei Punkten, eine Kopie eines Objekts anlegen können. Hier ist jedoch äußerste Vorsicht geboten, da diese Operation lediglich eine flache Kopie des Objekts erzeugt. Hat ein Objekt oder Array also wieder Unterstrukturen, werden diese lediglich by-reference kopiert und das Problem ist nur oberflächlich gelöst.

Eine weitere Möglichkeit, ein Objekt in JavaScript zu kopieren, besteht darin, die Methoden *JSON.stringify* und *JSON.parse* für etwas zu missbrauchen, wozu sie eigentlich nicht gedacht waren. So können Sie Ihr Objekt zunächst mit *JSON.stringify* in einen JSON-String umwandeln und anschließend mit *JSON.parse* wieder zurück in ein Objekt, und Sie erhalten eine tiefe Kopie mit dem kleinen Nachteil, dass sämtliche Methoden des Objekts verschwunden sind. Also auch keine vernünftige Lösung für unser Problem.

Die saubere Lösung besteht darin, eine Funktion zu schreiben, die eine tiefe Kopie einer Objekt- beziehungsweise Arraystruktur erzeugt. Glücklicherweise ist das nicht gerade ein exotisches Problem, sodass sich schon einige Leute an einer Lösung versucht haben. Jetzt wäre die JavaScript-Welt nicht das, was sie ist, wenn es nicht mittlerweile dutzende konkurrierende Lösungen gäbe und so zeige ich Ihnen jetzt drei Vertreter, die das Problem auf unterschiedliche Weisen angehen: Immutable, Immer und immutability-helper.

Listing 1: Veränderung von Datenstrukturen

```
let str = 'Hallo Welt!';
let obj = {
  name: 'Klaus',
};

function mutate(s, o) {
  s = 'Good morning';
  obj.name = 'Peter';
  console.log(s);
  console.log(obj);
}

mutate(str, obj);

console.log(str);
console.log(obj);
```

Listing 2: Ausgabe des Beispiels

```
Good morning
{ name: 'Peter' }
Hallo Welt!
{ name: 'Peter' }
```

Listing 3: Objekte vor Änderungen schützen

```
let obj = { name: 'Klaus' };
Object.freeze(obj);

function mutate(o) {
  o.name = 'Peter';
  console.log(o);
}

mutate(obj);
console.log(obj);
```

Listing 4: Immutable Datenstrukturen

```
let obj = { name: 'Klaus' };

function mutate(o) {
  let copy = { ...o };
  copy.name = 'Peter';

  console.log(copy);
  return copy;
}

mutate(obj);

console.log(obj);
```

Immutable – die große, allumfassende Lösung

Das Problem mit Immutability und JavaScript ist, dass die Sprache dieses Konstrukt nicht wirklich unterstützt. Die Bibliothek Immutable löst das Problem, indem sie eigene Datentypen, wie beispielsweise *Set*, *Map* und *Collection* definiert. Diese Datentypen stellen Methoden zur Verfügung, die dafür sorgen, dass die Operationen auf den Datentypen immutable sind. Unser Beispiel lässt sich mit dem Datentyp *Map* umsetzen und sieht dann wie in Listing 5 aus.

Damit das Beispiel funktioniert, müssen Sie Immutable mit dem Kommando *npm i immutable* installieren und den Quellcode des Beispiels über Node.js ausführen. Dem *Map*-Typen können Sie eine initiale Struktur übergeben und anschließend mit Methoden, wie beispielsweise *set* Werte setzen. Als Rückgabe erhalten Sie ein neues, modifiziertes Objekt vom Typ *Map*.

Der Nachteil von Immutable ist, dass Sie nicht mehr die JavaScript-Typen nutzen können, sondern sich auf Immutable einlassen müssen. Das bedeutet aber auch, dass eine Migration weg von Immutable in der Regel sehr viel Arbeit bedeutet.

Listing 5: Immutable im Beispiel

```
const { Map } = require('immutable');

let obj = Map({ name: 'Klaus' });

function mutate(o) {
  let copy = o.set('name', 'Peter');
  console.log(copy.toJSON());
  return copy;
}

mutate(obj);

console.log(obj.toJSON());
```

Listing 6: Immer für immutable Datenstrukturen

```
const produce = require('immer').
                                produce;

let obj = { name: 'Klaus' };

function mutate(o) {
  let copy = produce(o, (draft) => {
    draft.name = 'Peter';
  });

  console.log(copy);
  return copy;
}

mutate(obj);

console.log(obj);
```

Listing 7: Änderungen an Objekten mit immutability-helper

```
const update = require('immutability-
                                helper');

let obj = { name: 'Klaus' };

function mutate(o) {
  let copy = update(o, { name: { $set:
                            'Peter' } });

  console.log(copy);
  return copy;
}

mutate(obj);

console.log(obj);
```

Immer – die leichtgewichtigere Alternative

Immer geht einen anderen Weg als Immutable, indem es sich auf die nativen JavaScript-Datentypen stützt und stattdessen mit *produce* nur eine Funktion zur Verfügung stellt. Installiert wird Immer mit dem Kommando *npm install immer*. Listing 6 enthält den Quellcode unseres Beispiels, der auf Immer angepasst wurde.

Der *produce*-Funktion übergeben Sie als erstes Argument das Originalobjekt und als zweites eine Funktion, die die Operationen enthält, die Sie auf dem Objekt ausführen möchten. Immer sorgt dann dafür, dass diese Operationen nicht auf der eigentlichen Datenstruktur ausgeführt werden, sondern auf einer Kopie, die Sie als Rückgabewert der *produce*-Funktion erhalten.

Immutability-helper – Immutability mit Kommandos

In gewisser Hinsicht spielt das immutability-helper-Paket in einer ähnlichen Liga wie Immer. Der Unterschied zwischen beiden ist, dass Sie bei immutability-helper auf einen recht begrenzten Vorrat von Kommandos zur Manipulation von Strukturen zurückgreifen können. Nach der Installation des Pakets mit dem Kommando *npm install immutability-helper* können Sie den Code unseres Beispiels wie in Listing 7 anpassen.

Beim immutability-helper geben Sie mit Kommandos an, wie Sie das Quellobjekt modifizieren möchten. Dabei orientieren Sie sich an der ursprünglichen Struktur. In unserem Beispiel geht es um eine Änderung an der *name*-Eigenschaft. Anstelle des Wertes der Eigenschaft geben Sie hier ein Objekt mit dem Schlüssel *$set* zum Setzen des Wertes gefolgt vom neuen Wert an.

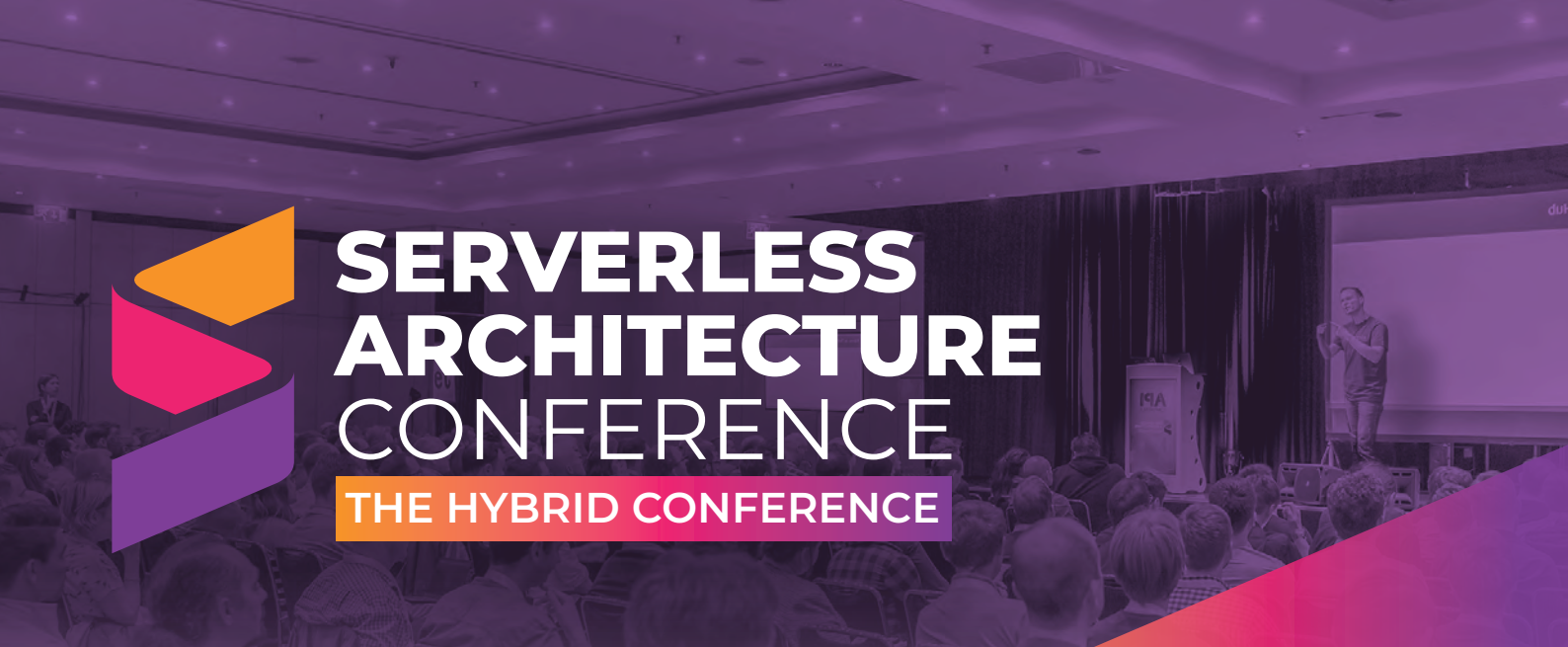
Fazit

Wie Sie gesehen haben, gibt es durchaus gute Gründe für Immutability in JavaScript, auch wenn es nicht nativ von der Sprache unterstützt wird. Das ist allerdings noch lange kein Grund, selbst eine Bibliothek zu schreiben, die Immutability für JavaScript bietet. Das haben leider schon etliche Entwickler vor Ihnen mit mehr oder weniger Erfolg getan. Aus der Fülle potenzieller Lösungen habe ich drei verschiedene und sehr populäre herausgepickt und sie Ihnen vorgestellt.

Mein Fazit: Immutable geht für meinen Geschmack einen Schritt zu weit, indem es eigene Datenstrukturen definiert, die Sie in einer Applikation durchgängig verwenden müssen, was einen Austausch der Bibliothek zu einem späteren Zeitpunkt so gut wie unmöglich macht.

Immer und immutability-helper hingegen stellen leichtgewichtige Lösungen dar und können im Notfall auch ausgetauscht werden. Hier stellt sich nur die Frage, ob Sie lieber den Immer-Weg gehen und die Änderungen an den Datenstrukturen in einer Funktion definieren oder ob Sie die Modifikationen lieber mit den Kommandos von immutability-helper beschreiben.

Ich würde dazu raten, die verschiedenen Bibliotheken auszuprobieren und zwar am besten nicht unbedingt mit einem so trivialen Beispiel, wie ich es Ihnen gezeigt habe, und sich dann für eine Lösung zu entscheiden.



SERVERLESS ARCHITECTURE CONFERENCE

THE HYBRID CONFERENCE

April 12 – 14, 2021

Expo: April 13 – 14, 2021

The Hague or online

Mastering Cloud Native Architectures, the Kubernetes Ecosystem and Functions

UNTIL FEBRUARY 11

- ✓ Raspberry Pi or C64 Mini for free
- ✓ Group discount
- ✓ Save up to € 404

Conference tracks

Serverless Architecture & Design

Serverless Development

Serverless Engineering & Operations

@ServerlessCon #SLA_con

ServerlessArchitectureCon

serverless-architecture.io

Presented by: e.academy entwickler.de

Organizer: S&S Media Group

Gestatten, Node.js – Teil 3

Codequalität mit Node.js

Wichtig bei der Entwicklung mit Node.js ist nicht nur das Schreiben des eigentlichen Codes, sondern auch die Qualitätssicherung. Dazu zählen neben der gewissenhaften Strukturierung des Codes auch das Anwenden von Coderichtlinien und das Ausführen von automatisierten Tests.

von Golo Roden

Im zweiten Teil dieser Serie ist eine in Node.js geschriebene Anwendung entstanden, die ein API zum Verwalten einer einfachen Aufgabenliste zur Verfügung stellt. Das API arbeitet dabei bewusst nicht nach REST, sondern trennt lediglich das Schreiben vom Lesen. Schreibende Vorgänge basieren auf POST, lesende auf GET.

Die eigentliche Semantik wurde dabei in den Pfad des URL verlagert, sodass die fachliche Intention erhalten bleibt und das API weitaus besser verständlich ist, als wenn es lediglich auf die vier in REST vorgesehenen technischen Verben setzen würde. Die aktuelle Ausbaustufe der Anwendung enthält drei Routen, auf der einen Seite zum Notieren und Abhaken von Aufgaben, andererseits zum Auflisten aller unerledigten Aufgaben:

Artikelserie

Teil 1: Node.js vorgestellt
Teil 2: Web-APIs mit Node entwickeln
Teil 3: Codequalität mit Node.js
Teil 4: Infrastruktur verwenden
Teil 5: Ab auf die Konsole

- *POST /note-todo*
- *POST /tick-off-todo*
- *GET /pending-todos*

Das Trennen von schreibenden und lesenden Vorgängen setzt sich im Code fort, außerdem finden sich hier auch die fachlichen Bezeichnungen. Unterschieden wird zwischen Funktionen, die den Zustand der Anwendung verändern, und Funktionen, die den aktuellen Zustand auslesen und zurückgeben. Damit implementiert die Anwendung eine einfache Form des CQRS-Entwurfsmusters [1], das genau diese Trennung empfiehlt.

Ebenfalls implementiert ist das Validieren der Eingaben mit Hilfe von JSON-Schemata und eine einfache RAM-basierte Datenhaltung, die sich aber problemlos gegen einen datenbankbasierten Ansatz austauschen ließe. Einen Client gibt es bislang nicht, allerdings lässt sich die Kommunikation mit dem HTTP API zum manuellen Testen leicht über Werkzeuge wie *curl* umsetzen.

Das größte Manko im Code ist die derzeit fehlende Struktur. Zwar sind der HTTP-Server (*./app.js*), das API (*./lib/getApp.js*) und die Datenhaltung (*./lib/Todos.js*) grundsätzlich getrennt, aber insbesondere im API werd n zahlreiche Aspekte vermischt. Das besser zu strukturieren, ist Bestandteil dieses Artikels. Das zweite Problem besteht in der bislang lediglich manuellen Qualitätssicherung, die durch die Anwendung von Coderichtlinien und das Ausführen von automatisierten Tests deutlich beschleunigt und verbessert werden könnte.

Coderichtlinien anwenden

Das Anwenden von Coderichtlinien ist der einfachste Teil. Unter einer Coderichtlinie versteht man eine Vorgabe, die festlegt, wie Code zu formatieren oder welches Sprachmerkmal in einer bestimmten Situation zu verwenden ist. Ein einfaches Beispiel wäre die Festlegung, Code generell mit zwei Leerzeichen einzurücken und Vergleiche stets mit dem typsicheren ===-Operator durchzuführen. In beiden Fällen kann man selbstverständlich versuchen, das von Hand regelmäßig zu überprüfen. Diese Arbeit ist allerdings zeitaufwendig und fehleranfällig. Daher empfiehlt es sich, diese Aufgabe einem Werkzeug zu übertragen, das die Überprüfung automatisiert durchführen kann. Ein solches Werkzeug wird als Linter bezeichnet. Für JavaScript gibt es verschiedene Linter, wobei ESLint [2] den De-facto-Standard darstellt.

Grundsätzlich ist es ein Leichtes, ESLint zu einem Projekt hinzuzufügen [3]. Dafür muss nur das Modul *eslint* mit Hilfe von npm installiert werden. Da ESLint in dem Fall nicht für die Ausführung der Anwendung relevant ist, sondern nur zur Entwicklungszeit benötigt wird, kann man das Modul als sogenannte *devDependency* installieren, also als eine Abhängigkeit, die nur für die Entwicklung relevant ist. Dazu muss npm beim Aufruf der Parameter *--save-dev* übergeben werden:

```
$ npm install eslint --save-dev
```

Nun gilt es, ESLint aufzurufen. Das erfolgt über das in der Node.js-Installation enthaltene Werkzeug npx, mit dem ausführbare Module aufgerufen werden können. Allerdings ist der Aufruf von *$ npx eslint* ernüchternd, er bewirkt nämlich nichts. Der Grund ist, dass ESLint als Parameter eine Liste der Dateien erwartet, die überprüft werden sollen. Da man in der Regel nicht sämtliche Dateien in allen Unterverzeichnissen einzeln angeben möchte, kann man ein sogenanntes Glob-Pattern verwenden, das die gewünschten Dateien beschreibt. Wichtig ist, darauf zu achten, dass man Anführungszeichen um das Pattern setzt, da ansonsten nicht ESLint die Pfade auflöst, sondern die Shell:

```
$ npx eslint './**/*.js'
```

Leider bewirkt auch dieser Aufruf nicht das gewünschte Ergebnis, aber immerhin weist ESLint nun darauf hin, dass es keine Konfiguration finden kann. ESLint ist nämlich in sehr hohem Maße konfigurierbar, was bedeutet, dass man für jedes Projekt individuell festlegen kann, welche Coderichtlinien aktiv sein sollen. Eine solche Konfiguration lässt sich als Ausgangsbasis erzeugen, indem man ESLint mit dem Parameter *--init* aufruft, es gibt jedoch noch einen anderen Weg. Es ist nämlich gar nicht ratsam, in jedem Projekt eine eigene ESLint-Konfiguration abzulegen, da man in der Regel eine einheitliche Konfiguration für alle Projekte haben möchte – immerhin soll beispielsweise die Einrückung nicht in dem einen Projekt so und in dem anderen Projekt anders sein, sondern stets auf die gleiche Art erfolgen. Deshalb bietet ESLint die Möglichkeit, eine Konfiguration in ein eigenes npm-Modul auszulagern, das dann in einem Projekt installiert werden kann. Die Konfigurationsdatei im Projekt muss dann nur noch auf dieses npm-Modul verweisen. Solche Konfigurationen können sich in ihrer Komplexität unterscheiden. Eine besonders strikte Variante ist *eslint-config-es* [4], die leicht installiert werden kann:

```
$ npm install eslint-config-es --save-dev
```

Listing 1

```
./app.js
  8:35  error  Invalid group length in numeric value  unicorn/numeric-separators-style

./lib/Todos.js
  10:20  error  Expected 'this' to be used by class async method 'initialize'
                                                                class-methods-use-this

./lib/getApp.js
  29:3  error  Expected blank line before this statement  padding-line-between-statements
  32:3  error  Expected blank line before this statement  padding-line-between-statements

✖ 4 problems (4 errors, 0 warnings)
  3 errors and 0 warnings potentially fixable with the `--fix` option.
```

Anschließend bedarf es nur noch einer Konfigurationsdatei namens *.eslintrc.json*, die im Hauptverzeichnis der Anwendung abgelegt werden muss. In ihr ist dann lediglich der Schlüssel *extends* anzugeben, der bestimmt, welches npm-Modul als Ausgangsbasis für die Regeln des Projekts gelten soll. Da das Modul *eslint-config-es* sowohl Regeln für Node.js als auch für den Webbrowser enthält, es sich in diesem Fall aber um eine Node.js-Anwendung handelt, ist der Eintrag wie folgt zu setzen:

```
{
  "extends": "eslint-config-es/node"
}
```

Ruft man nun die Codeanalyse mit dem bereits bekannten Befehl auf, wird tatsächlich eine Handvoll Fehler gemeldet (Listing 1), wobei es sich nur um Kleinigkeiten handelt.

Die beiden Fehlermeldungen bezüglich der Datei *./lib/getApp.js* lassen sich leicht beheben, hier fehlt lediglich jeweils eine Leerzeile im Code. Interessanter sind hingegen die beiden anderen Fälle. In der Datei *./app.js* wird die Zeile *const port = processenv('PORT', 3000);* bemängelt. Der Grund ist, dass der verwendete ESLint-Regelsatz erfordert, dass Zahlen zur besseren Lesbarkeit mit Tausendertrennzeichen versehen werden, was in JavaScript seit ECMAScript 2019 mit dem Feature Numeric Separator [5] möglich ist. Korrigiert lautet die Zeile daher:

```
const port = processenv('PORT', 3_000);
```

Die Meldung bezüglich der Datei *./lib/Todos.js* betrifft die Funktion *initialize*, die bislang außer einem Kommentar noch keinen Code enthält:

```
async initialize () {
  // Intentionally left blank.
}
```

Listing 2

```
'use strict';

const { Value } = require('validate-value');

const noteTodo = new Value({
  type: 'object',
  properties: {
    title: { type: 'string', minLength: 1 }
  },
  required: [ 'title' ],
  additionalProperties: false
});

module.exports = noteTodo;
```

Listing 3

```
'use strict';

const { Value } = require('validate-value');

const tickOffTodo = new Value({
  type: 'object',
  properties: {
    id: { type: 'string', format: 'uuid' }
  },
  required: [ 'id' ],
  additionalProperties: false
});

module.exports = tickOffTodo;
```

Hier stört sich ESLint daran, dass die Funktion als Instanzfunktion deklariert wurde, aber nicht auf die Instanz zugreift, da das Schlüsselwort *this* nicht in der Funktion aufgerufen wird. Aus technischer Sicht ist die Kritik durchaus berechtigt, allerdings ist es an der Stelle nicht sinnvoll, die Funktion mit *static* zu deklarieren. Deshalb ist der einfachste Ansatz, ESLint in diesem Fall anzuweisen, den Fehler lokal zu ignorieren:

```
// eslint-disable-next-line class-methods-use-this
async initialize () {
  // Intentionally left blank.
}
```

Mit dem zuletzt gezeigten Vorgehen sollte man allerdings vorsichtig sein, da zunächst davon auszugehen ist, dass jeder von ESLint bemängelte Fehler eine behebbare Ursache hat. ESLint stummzuschalten sollte stets nur der letzte Ausweg sein und dann mit einem Kommentar versehen werden, warum diese Regel an der Stelle deaktiviert wird, falls der Grund nicht offensichtlich ist.

Damit ist ESLint fast vollständig eingerichtet, es fehlt nur noch eine Möglichkeit, ESLint komfortabel aufrufen zu können. Am einfachsten gelingt das, indem man der Datei *./package.json* einen *scripts*-Abschnitt hinzufügt, in dem man ein *analyse*-Kommando hinterlegt, das den gewünschten Aufruf durchführt. Der Name des Kommandos lässt sich frei wählen, man könnte es also *lint* oder *eslint* nennen:

```
"scripts": {
  "analyse": "npx eslint './**/*.js'"
}
```

Um dieses Kommando aufzurufen, genügt es, npm mit dem Parameter *run* und dem Namen des Kommandos auszuführen. Damit steht die Codeanalyse automatisiert zur Verfügung: *$ npm run analyse*

Den Code strukturieren

Nachdem die statische Codeanalyse eingerichtet ist, kann man im nächsten Schritt den Code besser strukturieren [6]. Was gut strukturierten Code auszeichnet, lässt sich umfangreich erörtern – letztlich lassen sich aber sehr viele Regeln auf einige wenige Grundprinzipien zurückführen. Ein wesentliches Prinzip ist das Single-Responsibility-Prinzip (SRP) [7], das besagt, dass eine Funktion beziehungsweise eine Klasse nur eine fachliche Verantwortlichkeit haben sollte – alles, was auch in anderem Kontext wiederverwendet werden könnte, sollte isoliert werden. Ein gutes Beispiel dafür ist die Trennung der Dateien *./app.js* und *./lib/getApp.js*. Während sich die erste um den HTTP-Server kümmert, betrifft die zweite die Definition des API. Theoretisch hätte man beide Aspekte in einer gemeinsamen Datei unterbringen können, aber dann hätte die entstandene Datei mehr als eine Verantwortlichkeit.

webinale
HYBRID EDITION

7. – 11. JUNI 2021
EXPO: 8. – 9. JUNI 2021
BERLIN ODER ONLINE

Nur bis 25. Februar
✓ Workshop-Tag gratis
✓ Kollegenrabatt
✓ Bis zu 700 € sparen

.business
.technology
.design
.online marketing

webinale – the holistic web conference

Die webinale ist die Konferenz für digitale Professionals, Trendsetter und Macher im World Wide Web – sie ist die optimale Crossover-Plattform für Wissensvermittlung, Inspiration sowie Erfahrungsaustausch über alle Branchen hinweg.

webinale

www.webinale.de

Präsentiert von: e.academy webmagazin

Veranstalter: S&S Media Group

Listing 4

```
'use strict';

const noteTodoSchema = require('../../schemas/noteTodo');

const noteTodo = function ({ todos }) {
  if (!todos) {
    throw new Error('Todos is missing.');
  }

  return async (req, res) => {
    if (!noteTodoSchema.isValid(req.body)) {
      return res.status(400).end();
    }

    const { title } = req.body;

    await todos.noteTodo({ title });
    res.status(200).end();
  };
};

module.exports = noteTodo;
```

Listing 5

```
'use strict';

const bodyParser = require('body-parser');
const cors = require('cors');
const express = require('express');
const getPendingTodos = require('./routes/queries/getPendingTodos');
const noteTodo = require('./routes/commands/noteTodo');
const tickOffTodo = require('./routes/commands/tickOffTodo');
const Todos = require('./Todos');

const getApp = async function () {
  const todos = new Todos();

  await todos.initialize();

  const app = express();

  app.use(cors());
  app.use(bodyParser.json());

  // Commands
  app.post('/note-todo', noteTodo({ todos }));
  app.post('/tick-off-todo', tickOffTodo({ todos }));

  // Queries
  app.get('/pending-todos', getPendingTodos({ todos }));

  return app;
};

module.exports = getApp;
```

Es hätte gänzlich unterschiedliche Gründe gegeben, die Datei künftig anzupassen, beispielsweise die Umstellung von HTTP auf HTTPS oder das Hinzufügen einer neuen Route. Da aber die Anpassung des Protokolls unabhängig vom API und das Hinzufügen einer neuen Route unabhängig vom Server erfolgen kann, hätten sich diese Änderungen in die Quere kommen können. Insofern ist der bereits gewählte Weg der besser strukturierte.

Diese saubere Trennung findet sich allerdings nicht in der Datei *./lib/getApp.js* wieder, sie macht tatsächlich zu viel. Sie definiert nicht nur das API, sondern enthält auch die Definition jeder einzelnen Route und die JSON-Schemata. Es bietet sich an, diese Aspekte herauszutrennen. Am einfachsten ist das mit den JSON-Schemata, die man jeweils in eine eigene Datei extrahieren kann. Damit das *./lib*-Verzeichnis überschaubar bleibt, kann man zu dem Zweck auch ein neues Verzeichnis anlegen, beispielsweise *./lib/schemas*. So entsteht die Datei *./lib/schemas/noteTodo.js*, die den Inhalt aufweist, der in Listing 2 zu sehen ist. Analog dazu verhält es sich mit der Datei *./lib/schemas/tickOffTodo.js* (Listing 3).

Auf die gleiche Art kann man vorgehen, um die Routen zu extrahieren. Dabei ist allerdings zu beachten, dass sie Zugriff auf das *todos*-Objekt benötigen, das derzeit allen Routen gemeinsam zur Verfügung steht. Da es nicht möglich ist, einer Expressroute einen eigenen Parameter zu übergeben, bietet sich hier ein Trick an. Statt die eigentliche Routenfunktion zu extrahieren, extrahiert man eine Funktion, die diese Routenfunktion zurückgibt. Die äußere Funktion kann man dann nach Belieben parametrisieren, die Routenfunktion hingegen kann problemlos auf die Parameter zugreifen. Auch hier empfiehlt es sich, aus Gründen der Übersichtlichkeit ein eigenes Verzeichnis anzulegen, beispielsweise *./lib/routes*. Darunter kann man Unterverzeichnisse anlegen, *commands* und *queries*, um schreibende und lesende Zugriffe auch hier sauber voneinander zu trennen. Die Datei *./lib/route/commands/noteTodo.js* enthält dann die in Listing 4 dargestellte Funktion.

Analog werden die beiden anderen Routen definiert. Die Datei *./lib/getApp.js*, die zum Definieren des eigentlichen API dient, wird so weitaus kürzer und übersichtlicher. Ihr Sinn lässt sich nun auf einen Blick erfassen (Listing 5).

Den Code auf diese Art zu strukturieren, ermöglicht es, in kleineren Einheiten zu denken und stärker zu abstrahieren. Das macht nicht nur die Wartung des Codes und das Nachdenken über ihn einfacher, sondern ermöglicht auch ein effizienteres Testen.

Den Code testen

Um Code zu testen, benötigt man zunächst ein Test-Framework, das die Ausführung der Tests übernimmt. Wie bei der statischen Codeanalyse gibt es zahlreiche Optionen, wobei es auch hier einen De-facto-Standard gibt, nämlich Mocha [8]. Alternativ trifft man immer

häufiger auch auf Jest [9]. Die Unterschiede sind aber so marginal, dass man problemlos von einem auf das andere umsteigen kann, ohne Neues lernen zu müssen. Mocha hat allerdings die weitaus längere Historie und umgeht einige Probleme, in die man als Einsteiger in Jest gerne hineinläuft. Um Mocha verwenden zu können, muss man es zunächst installieren. Das erfolgt wiederum über npm, wobei auch Mocha eine *devDependency* darstellt, da die Testausführung ausschließlich zur Entwicklungszeit erfolgt:

```
$ npm install mocha --save-dev
```

Nun kann man Mocha auf dem gleichen Weg aufrufen wie ESLint, wobei das noch nicht allzu viel bringt, da es noch keinen einzigen Test gibt: *$ npx mocha*. Um einen Test zu schreiben, muss man zunächst ein Verzeichnis namens *./test* anlegen. Das ist der von Mocha vorgegebene Name, der sich zwar theoretisch auch ändern lässt, doch in der Praxis hat sich bewährt, beim vorgeschlagenen Standard zu bleiben. Ebenfalls bewährt hat es sich, innerhalb des *test*-Verzeichnisses das *lib*-Verzeichnis zu spiegeln. Das bedeutet, dass die Tests zu *./lib/schemas/noteTodo.js* in *./test/schemas/noteTodoTests.js* zu finden sind. Das angehängte Suffix *Tests* ist zwar nicht zwingend erforderlich, hilft aber, beim Lesen von Fehlermeldungen zügiger einzugrenzen, woher ein Fehler ursprünglich stammt.

Will man nun einen Test definieren, gilt es, die *test*-Funktion aufzurufen. Ihr sind der Name des Tests und ein Callback zu übergeben, der den eigentlichen Testcode enthält. Mehrere Tests lassen sich wiederum in einer *suite* zusammenfassen, die nach dem gleichen Prinzip arbeitet. Daher sieht das Grundgerüst für die Datei *./test/schemas/noteTodoTests.js* aus wie in Listing 6.

Die Formulierung der Testbeschreibung ist im Prinzip beliebig. Das System, dass Suite- und Testname gemeinsam einen Satz ergeben müssen, weshalb ein Test mit einem Verb in der dritten Person beginnt, skaliert aber sehr gut auch bei vielen Tests.

Mocha unterstützt neben den Begriffen *suite* und *test* auch die Varianten *describe* und *it*. Semantisch macht das keinen Unterschied, es handelt sich lediglich um eine andere Syntax. Da Mocha standardmäßig davon ausgeht, dass man mit *describe* und *it* arbeiten möchte, muss man beim Aufruf den Parameter *--ui* mit angeben und ihn auf *tdd* setzen, um die Syntax mit *suite* und *test* zu aktivieren. Außerdem gibt es einige weitere Parameter, die man generell verwenden sollte:

- *--async-only* erzwingt, dass jede Callback-Funktion eines Tests asynchron sein muss. Das verhindert unerwünschte Ergebnisse bei Code, der zwar asynchron arbeitet, aber ohne *await* aufgerufen wird.
- *--bail* bricht die Testausführung nach dem ersten fehlgeschlagenen Test ab. So findet man Fehler zügiger und muss nicht durch seitenweise Ausgabe im Terminal scrollen.
- *--recursive* bewirkt, dass Mocha nicht nur die Dateien im *test*-Verzeichnis berücksichtigt, sondern auch Dateien in Unterverzeichnissen dieses Verzeichnisses.

Damit sieht der Aufruf von Mocha wie folgt aus:

```
$ npx mocha --async-only --bail --recursive --ui tdd
```

Als Ergebnis erhält man die Meldung, dass ein Test erfolgreich durchgelaufen sei:

```
noteTodo
  ✓ returns true if a title is given.


1 passing (4ms)
```

Das liegt daran, dass der Test nicht in einen Fehler gelaufen ist: Jeder Test, der ohne Fehler ausgeführt werden kann, wird von Mocha als erfolgreich betrachtet. Bevor es nun daran geht, den Test mit Leben zu füllen, empfiehlt es sich, einen weiteren Eintrag in der *scripts*-Sektion in der Datei *package.json* vorzunehmen:

```
"scripts": {
  "analyse": "npx eslint './**/*.js'",
  "test": "npx mocha --async-only --bail --recursive --ui tdd"
}
```

Listing 6

```
'use strict';

suite('noteTodo', () => {
  test('returns true if a title is given.', async () => {
    // ...
  });
});
```

Listing 7

```
'use strict';

const noteTodo = require('../../lib/schemas/noteTodo');

suite('noteTodo', () => {
  test('returns true if a title is given.', async () => {
    const isValid = noteTodo.isValid({
      title: 'the native web'
    });

    // ???
  });
});
```

Es zeigt sich, dass gut testbarer Code auch stets Code ist, der gut strukturiert ist.

Praktisch ist außerdem, ein *qa*-Skript zu definieren, das beide Schritte vereint:

```
"scripts": {
  "analyse": "npx eslint './**/*.js'",
  "qa": "npm run analyse && npm run test",
  "test": "npx mocha --async-only --bail --recursive --ui tdd"
}
```

Der eigentliche Test muss nun überprüfen, ob das Schema funktioniert. Das heißt, es gilt, für eine beispielhafte korrekte Eingabe zu überprüfen, dass als Ergebnis wirklich *true* zurückgegeben wird. Dazu ist der Test zunächst wie in Listing 7 anzupassen.

Offen ist allerdings die Frage, wie der Wert *isValid* überprüft werden kann. Dazu benötigt man ein sogenanntes Assertion-Modul, beispielsweise assertthat [10], das sich ebenfalls wieder über npm als *devDependency* installieren lässt:

```
$ npm install assertthat --save-dev
```

In der Testdatei gilt es diese Abhängigkeit nun zu importieren:

```
const { assert } = require('assertthat');
```

Anschließend kann man den Kommentar im Test durch den Aufruf von *assert* ersetzen:

```
assert.that(isValid).is.true();
```

Führt man die Tests nun erneut aus, wird der Test wieder grün angezeigt. Verwendet man an Stelle von *is.true()* den Aufruf *is.false()*, lässt sich leicht nachweisen, dass der Test tatsächlich funktioniert. Die Fehlermeldung besagt, dass der Wert *true* zurückgegeben wurde, er aber *false* hätte sein sollen:

```
AssertionError [ERR_ASSERTION]: Expected true to be false.
```

Ein Test allein genügt natürlich bei Weitem nicht, aber weitere Tests zu schreiben ist nur noch Fleißarbeit. Je früher man beginnt, ein Projekt mit Tests zu versehen, desto besser ist von Anfang an die erreichte Qualität. Außerdem tragen Tests einen großen Teil dazu bei, eine gut wartbare Struktur aufzubauen. Es zeigt sich nämlich, dass gut testbarer Code auch stets Code ist, der gut strukturiert ist, wohingegen sich schlecht strukturierter Code in aller Regel auch nur umständlich testen lässt. Eine gute Struktur und einfache Tests bedingen sich also gegenseitig.

Ausblick

Damit endet der dritte Teil der Serie zu Node.js. Selbstverständlich gibt es insbesondere im Bereich des Testens noch viel mehr zu entdecken, spätestens, wenn der Anspruch über reine Unittests hinausgeht. Allein dem Thema Integrationstests ließe sich ein weiterer Artikel widmen.

Der nächste Schritt besteht darin, der Anwendung weitere Funktionalität hinzuzufügen und sie dann produktiv auszuführen. Das sollte wie bei jeder modernen Web- und Cloud-Anwendung mit Hilfe von Docker und Kubernetes erfolgen, wobei es einige Stolperfallen zu beachten gilt. Um diese Themen wird es im vierten Teil der Serie gehen, bevor im fünften und letzten Teil ein Client für die Anwendung entstehen wird.

Das Unternehmen des Autors, the native web GmbH, bietet einen kostenfreien Videokurs zu Node.js [11] mit annähernd 30 Stunden Spielzeit an. Die Folgen 11, 12 und 15 dieses Videokurses beschäftigen sich mit den in diesem Artikel behandelten Themen wie der statischen Codeanalyse und dem Schreiben von Unit- und Integrationstests. Daher sei dieser Kurs allen Interessierten für weitergehende Details empfohlen.

Golo Roden ist Gründer und CTO der the native web GmbH, einem auf native Webtechnologien spezialisierten Unternehmen. Für die Entwicklung moderner Webanwendungen bevorzugt er JavaScript und Node.js und hat mit „Node.js & Co." das erste deutschsprachige Buch zu diesem Thema geschrieben. Darüber hinaus ist er journalistisch für Fachmagazine und als Referent und Content-Manager für Konferenzen im In- und Ausland tätig. Für sein qualitativ hochwertiges Engagement in der Community wurde Golo dreifach als Microsoft MVP ausgezeichnet.

www.thenativeweb.io

Links & Literatur

[1] https://www.youtube.com/watch?v=k0f3eeiNwRA&t=1s

[2] https://eslint.org

[3] https://www.youtube.com/watch?v=7J6Yg21BOeQ

[4] https://www.npmjs.com/package/eslint-config-es

[5] https://github.com/tc39/proposal-numeric-separator

[6] https://www.youtube.com/watch?v=2tE_1RSmXkQ

[7] https://www.youtube.com/watch?v=3to0t7YQMnU

[8] https://mochajs.org

[9] https://jestjs.io

[10] https://www.npmjs.com/package/assertthat

[11] https://www.thenativeweb.io/learning/techlounge-nodejs

Bis 25. Februar anmelden und bis zu 200 € pro Ticket sparen!

JavaScript Days · Angular Days · React Days · HTML & CSS Days

22. – 25. März 2021
MÜNCHEN & ONLINE

Das große Hybrid-Trainingsevent für JavaScript, Angular, React, HTML & CSS

Mit diesem 4-in-1-Paket können Sie ein Event buchen und gleichzeitig vier unterschiedliche Tracks mit über 30 Workshops besuchen. Außerdem gibt es die Wahl zwischen einem Vor-Ort- oder Remote-Ticket, mit dem Sie ganz bequem digital teilnehmen. Suchen Sie sich schon jetzt Ihre Favoriten aus und sichern sich Ihren Platz!

UNSERE TRAINER*INNEN


Tobias Cabrera Cano


Hannah Ebert


Martina Kraus


Peter Kröner


Christian Liebel


David Müllerchen


Hans-Christian Otto


Sebastian Springer


Manfred Steyer

AUSZUG AUS DEM PROGRAMM

Type-Level-Programmierung in TypeScript
Peter Kröner (Webtechnologie-Erklärbär)

Angular meets Project Fugu: Produktivitäts-PWAs auf Desktopniveau (Hands-on)
Christian Liebel (Thinktecture AG)

Struktur für Ihre großen Angular-Anwendungen: Pakete, Monorepos und Microfrontends
Manfred Steyer (SOFTWAREarchitekt)

Sheep'n'Run – die Probleme des JavaScript-Alltags spielerisch lösen
Sebastian Springer (MaibornWolff)

Robuste Design Systems mit Storybook und Angular: vom Konzept zur lebendigen Anwendung
Andreas Wissel (innoQ Deutschland GmbH)

Präsentiert von: entwickler akademie

Powered by: e.academy

Veranstalter: S&S Media Group

javascript-days.de

Teil 13: Intelligente Neuronen und dendritisches Lernen (I)

Classic Games Reloaded

Im heutigen Artikel befassen wir uns mit der Funktionsweise und den Einsatzmöglichkeiten eines weiterentwickelten Neuronenmodells. Mit Hilfe dieser intelligenten Neuronen lassen sich unter anderem verschiedene Verhaltensweisen in einem Computerspiel simulieren, Gefahrenabschätzungen innerhalb der Spielewelt vornehmen oder Funktionsverläufe approximieren.

von Alexander Rudolph

Können Einzeller bzw. einzelne Neuronen intelligente Verhaltensweisen an den Tag legen? In Frank Schätzings Roman „Der Schwarm“ aus dem Jahr 2004 wird diese Frage eindeutig mit einem dicken, fetten **„Ja“** beantwortet. Intelligente Einzeller, die Yrr, die in den Tiefen des Meeres leben und die Menschen vom Angesicht der Erde tilgen wollen, gehören selbstredend in das Reich der Fantasie (hoffe ich zumindest). Nichtsdestotrotz handelt es sich ohne Frage um ein recht interessantes Gedankenspiel. Nun ja, zumindest der Versuch, das Verhalten dieser Einzeller mit Hilfe eines Computerprogramms simulieren zu wollen, scheint dann doch zunächst ein wenig an den Haaren herbeigezogen zu sein: Man nehme ein künstliches neuronales Netzwerk, kappe sämtliche neuronalen Verbindungen und gestatte es den einzelnen Neuronen, in Eigenregie miteinander zu verschmelzen – und schon soll die Post abgehen? Die aus den Verschmelzungen hervorgehenden Schwärme und Netzwerke sollen angeblich die Leistungsfähigkeit der modernsten Parallelrechner um Längen übertreffen und urplötzlich in der Lage dazu sein, selbst die kompliziertesten Probleme in Nullkommanix zu bewältigen. Ich muss an dieser Stelle wohl nicht extra betonen, dass sich mit Hilfe der künstlichen Neuronen, die in unseren Programmbeispielen bislang zum Einsatz gekommen sind, keine derartigen Computersimulationen realisieren lassen.

Die einzelnen Neuronen, die wir zum Aufbau der von uns verwendeten künstlichen neuronalen Netzwerke genutzt haben (siehe hierzu das in **Abbildung 1** illustrierte Neuronenmodell ohne Dendriten), repräsentieren lediglich sogenannte Aktivierungsfunktionen. Mit deren Hilfe lassen sich die jeweiligen Eingangssignale – hierbei kann es sich um Umgebungsreize handeln oder um Signale von vorgeschalteten Neuronen – in entsprechende Ausgangssignale umrechnen, bevor sie dann mit unterschiedlichen Gewichtungen an nachgeschaltete Neuronen weitergeleitet werden:

```
NeuronOutput = Activationfunction(SumOfNeuronInputs)
```

- **Beispiel 1** – Tanh-Aktivierung (**Tan**gens **h**yperbolicus):
  ```
  NeuronOutput = tanh(SumOfNeuronInputs);
  ```
- **Beispiel 2** – ReLU-Aktivierung (**Re**ctified **L**inear **U**nit):
  ```
  NeuronOutput = max(0.0f, SumOfNeuronInputs);
  ```
- **Beispiel 3** – binäre Aktivierung:
  ```
  NeuronOutput = 0.0f;
  if (SumOfNeuronInputs > Threshold)
    NeuronOutput = 1.0f;
  ```

Im heutigen Artikel werden wir uns unter anderem damit beschäftigen, auf welche Weise wir unsere künstlichen Neuronen zumindest mit ein wenig Intelligenz ausstatten können. Im Unterschied zu den Yrr, die in Schätzings Roman ihr selbsterlangtes und geerbtes Wissen in ihrer DNS kodieren, stellen in unserem Fall die sogenannten RBF-Neuronen den Ausgangspunkt für alle weiteren Betrachtungen dar. Die Abkürzung **RBF** verweist in diesem Zusammenhang auf einen speziellen Typus von Aktivierungsfunktionen, die bei der Simulation eines solchen Neurons zum Einsatz kommen: den sogenannten **r**adialen **B**asis-**F**unktionen. Ähnlich wie ein Yrr besitzt ein RBF-Neuron gewissermaßen ein eigenes Gedächtnis, in welchem sich beispielsweise Zahlenwerte, Zahlenreihen, Muster, Ziffern oder Buchstaben abspeichern lassen.

Das Funktionsprinzip eines RBF-Neurons

Wie Sie anhand der nachstehend vorgestellten RBF-Aktivierungsfunktion erkennen können, liegt die neuronale Aktivität (*neuronOutput*) eines RBF-Neurons in einem Bereich zwischen null und eins, sofern man als Vorfaktoren (*f1*, *f2* usw.) lediglich positive Zahlenwerte berücksichtigt. Der Aktivitätsverlauf selbst entspricht einer Glockenkurve, deren Breite von den einzelnen Vorfaktoren abhängt. Je kleiner die Faktoren sind, umso breiter ist die Kurve:

- RBF-Aktivierungsfunktion:

```
sumSquare = f1*(input1–c1)*(input1–c1) + f2*(input2–c2)*(input2–c2) + ...;
neuronOutput = exp(-sumSquare);
```

Das Gedächtnis eines RBF-Neurons wird nun gewissermaßen einerseits durch die einzelnen Centroid-Werte *c1*, *c2*, usw. (Centroid, zu Deutsch: Schwerpunkt) und andererseits durch die zugehörigen Vorfaktoren repräsentiert. Wenn nun, wie im ersten Rechenbeispiel gezeigt wird, die einzelnen Inputwerte mit den jeweils zugeordneten Centroid-Werten identisch sind (der Eingabevektor entspricht folglich dem Schwerpunktvektor), dann führt das zu der maximal möglichen neuronalen Aktivität von eins:

- **Beispiel 1** – maximale neuronale Aktivität, da die Eingabewerte mit den zugehörigen Centroid-Werten identisch sind:

```
input = (1, 0) sowie centroid = (1, 0) und f1 = f2 = 1

sumSquare = 1*(1–1)*(1–1) + 1*(0–0)*(0–0) = 0
neuronOutput = exp(-sumSquare) = exp(-0) = 1
```

Jede einzelne Abweichung zwischen einem Input- und dem jeweils zugeordneten Centroid-Wert führt zu einer Verringerung der neuronalen Aktivität. Mit Hilfe der einzelnen Vorfaktoren können wir nun die jeweiligen Abweichungen ganz individuell gewichten. Spielt eine Abweichung nur eine untergeordnete Rolle, muss der Wert des zugehörigen Vorfaktors entsprechend klein sein. Ein entsprechend großer Vorfaktor führt hingegen dazu, dass die neuronale Aktivität schon bei einer verhältnismäßig kleinen Abweichung gegen null geht:

- **Beispiel 2** – verringerte neuronale Aktivität aufgrund einer Abweichung zwischen dem zweiten Input- und dem zugehörigen Centroid-Wert:

```
input = (1, 0), centroid = (1, 2) und f1 = f2 = 1

sumSquare = 1*(1–1)*(1–1) + 1*(0–2)*(0–2) = 4
neuronOutput = exp(-sumSquare) = exp(-4) = 0.018
```

Intelligente Neuronen

Am Anfang dieses Artikels habe ich die Frage in den Raum gestellt, ob Einzeller oder einzelne Neuronen so

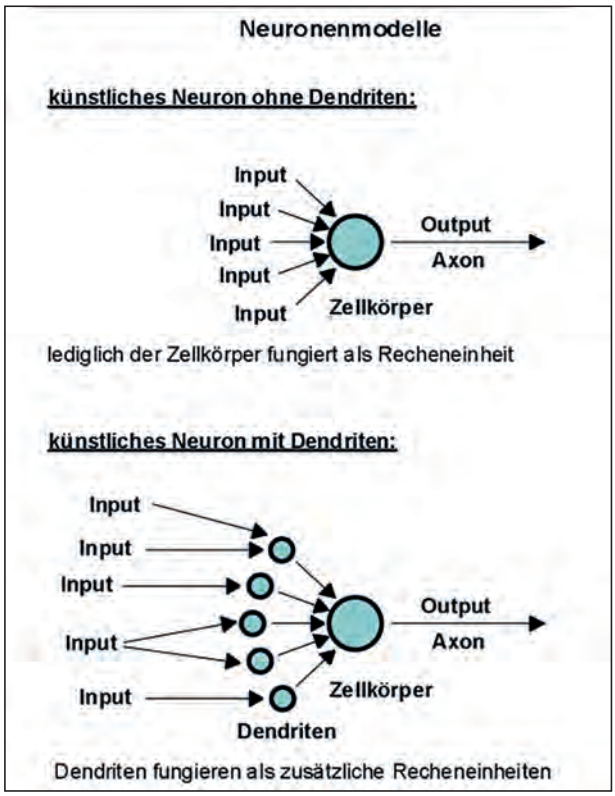


Abb. 1: Neuronenmodelle

etwas wie Intelligenz hervorbringen können. Anhand von mehreren einfachen Fallbeispielen möchte ich Ihnen an dieser Stelle demonstrieren, dass sich diese Frage für RBF-Neuronen durchaus mit einem „Ja“ beantworten lässt. In unserem ersten Beispiel beschäftigen wir uns zunächst einmal mit dem Thema der Risikoabschätzung. So ist die Voraussetzung dafür, dass sich eine von der KI kontrollierte Einheit oder Spielfigur möglichst gefahrlos durch die Spielewelt bewegen kann, eine Risikobewertung von sämtlichen in Frage kommenden Wegen. In diesem Zusammenhang könnte das für die Gefahrenabschätzung verantwortliche RBF-Neuron beispielsweise auf die nachfolgende Aktivierungsfunktion zurückgreifen:

```
risk = exp(-f*distSq);
```

Je größer der quadratische Abstand *distSq* zu einer potenziell gefährlichen Position innerhalb der Spielewelt ist (an dieser Stelle könnte sich beispielsweise ein möglicher Gegner oder ein todbringendes Objekt wie eine Bombe befinden), umso geringer ist die Gefahr für eine KI-Einheit oder -Spielfigur. Mit Hilfe des Faktors *f* können wir zudem festlegen, wie groß das Risiko an den jeweiligen Positionen tatsächlich ist. Befindet sich an einer Position zum Beispiel ein übermächtiger Gegner, dann muss der Wert des Faktors *f* entsprechend klein gewählt sein (der Funktionsverlauf entspricht dann einer breiten Glockenkurve). Geht von dem betreffenden Gegner hingegen nur eine mäßige Gefahr aus, dann müssen wir für den Faktor *f* einen entsprechend größeren Wert verwenden (Peak-förmiger Funktionsverlauf). So gesehen könnten wir den Faktor beispielsweise wie folgt berechnen:

```
f = 1.0f / (0.0001f + riskPotential);
```

Selbstverständlich können wir bei der Risikobewertung auch das Gefahrenpotenzial mehrerer Positionen gleichzeitig berücksichtigen:

```
risk = exp(-f1*distSq1) + exp(-f2*distSq2) + ...;
```

Kommen wir nun auf unser zweites Fallbeispiel zu sprechen. Wie Sie vielleicht wissen, ist die Aktivität vieler Lebewesen unter anderem von der Umgebungstemperatur T und der momentanen Helligkeit H abhängig. Sobald die besagten Werte von den optimalen Parametern (optimale Temperatur oT sowie optimale Helligkeit oH) abweichen, verringert sich die Aktivität der besagten Lebewesen zusehends. Mittels der nachfolgend gezeigten Aktivierungsfunktion können wir ein solches Verhalten unter Zuhilfenahme eines einzelnen RBF-Neurons simulieren:

```
activity = exp(-(f1*(H-oH)*(H-oH) + f2*(T-oT)*(T-oT)));
```

In Rahmen unseres dritten Fallbeispiels werden wir uns mit dem Thema der Verhaltenssimulation in einem Computerspiel auseinandersetzen. Es versteht sich von selbst, dass man auch in einem recht einfach gestrickten Spiel das Verhalten der einzelnen KI-Gegner nicht mal eben so nach dem Zufallsprinzip festlegen kann. In der Regel muss die KI zunächst einmal eine Reihe von Grundbedürfnissen (Gesundheit, Nahrung, Munition, usw.) im Blick behalten und wenn nötig durch Auswahl einer der nachfolgend aufgelisteten Verhaltensweisen sicherstellen, dass es hier zu keiner Schieflage kommt, die im schlimmsten Fall ein vorzeitiges Ableben zur Folge hätte:

- Gesundheitsfürsorge (Erste-Hilfe-Ausrüstung suchen, Lebenspunkte einsammeln usw.)
- Nahrungssuche bzw. Energiebeschaffung
- Munitionsbeschaffung

Geht beispielsweise der Munitionsvorrat zur Neige, müssen wir natürlich erst einmal der Munitionsbeschaffung den Vorrang geben, bevor der betreffende KI-Gegner wieder in den Angriffsmodus wechseln kann:

Codebeispiel und Listings zum Download

Achtung: Das Beispielprojekt und die Listings zu diesem Artikel finden Sie in unserem GitHub Repository: https://github.com/EntwicklerMagazin/Entwickler-Magazin. Dort stehen auch die Codes zu den vorherigen Teilen der Serie bereit.
Der Autor stellt die Programmbeispiele außerdem unter www.graphics-and-physics-framework.spieleprogrammierung.net bereit.

- Ermittlung des Prioritätswerts für die Munitionsbeschaffung (Variante 1):

```
value = max(Ammo, minTolerableAmmo) - minTolerableAmmo;
priority = exp(-f*value*value);
```

- Ermittlung des Prioritätswerts für die Munitionsbeschaffung (Variante 2):

```
value = min(Ammo, minTolerableAmmo) - minTolerableAmmo;
priority = 1 - exp(-f*value*value);
```

Sind die Prioritätswerte für die Gesundheitsfürsorge, Nahrungssuche und Munitionsbeschaffung allesamt deutlich kleiner als eins, kann sich die KI im zweiten Schritt auch mit komplexeren Fragestellungen auseinandersetzen. Zur Beantwortung der Frage, ob beispielsweise die Möglichkeiten für einen Angriff (*attackPossibility*) gegeben sind, muss die KI unter anderem den Gesundheitszustand sowie den Munitionsvorrat in ihre Bewertung mit einbeziehen:

```
value1 = min(Health, minTolerableHealth) - minTolerableHealth;
value2 = min(Ammo, minTolerableAmmo) - minTolerableAmmo;
attackPossibility = exp(-(f1*value1*value1 + f2*value2*value2));
```

Letzten Endes ist es von den Vorfaktoren und Centroid-Werten (*minTolerableHealth*, *minTolerableAmmo* usw.) der zuvor betrachteten Aktivierungsfunktionen abhängig, wie sinnvoll die Berechnungen unserer „intelligenten“ Neuronen tatsächlich sind. Die jeweiligen Werte für die einzelnen Parameter können wir natürlich von Hand vorgeben – sofern wir sie denn kennen. Da dies jedoch in der Regel nicht der Fall sein wird, müssen die einzelnen Neuronen die besagten Werte ähnlich wie ein neuronales Netzwerk normalerweise durch irgendeine Form des Trainings erlernen.

Neuronen-Ensembles und Schwarmintelligenzen

Um die Intelligenzleistung weiter zu erhöhen, bietet es sich zudem an, einzelne Neuronen zu einem Ensemble (zu einem „Schwarm“) zusammenzufassen, wobei wir auf die in Listing 1 deklarierte *CNeuronEnsembleV2*-Klasse zurückgreifen. Um verstehen zu können, welche Vorteile ein solches Neuronenensemble gegenüber einem einzelnen Neuron bietet, betrachten wir als konkretes Beispiel einmal ein RBF-Neuron, welches wir darauf trainiert haben, in diversen Bildern den Buchstaben B zu identifizieren. Aufgrund der Ähnlichkeiten zwischen dem Buchstaben B und der Ziffer 8 liegt es nun durchaus im Rahmen des Möglichen, dass auch das Bild einer 8 bei unserem Neuron fälschlicherweise eine gewisse neuronale Aktivität hervorruft. In einem Ensemble, in dem ein Neuron auf die Erkennung des Buchstaben B und ein anderes auf die Identifizierung der Ziffer 8 spezialisiert ist, spielt das hingegen keine Rolle, da wir mit Hilfe der in Listing 2 skizzierten *CNeuronEnsembleV2*-Methode *Get_ID_And_Output_Of_NeuronWithMaxOutput()* zielgenau den Outputwert sowie den Index des Neurons mit der maximalen Aktivität ermitteln können. Oder anders ausgedrückt, im Unterschied zu einem einzelnen

Neuron kann ein Neuronenensemble verschiedene Buchstaben und Ziffern eindeutig voneinander unterscheiden.

Die Ausgabewerte eines Neuronenensembles sind mitunter nicht ganz einfach zu interpretieren. Bevor wir beispielsweise den zweiten Outputwert in der nachstehend gezeigten Liste korrekt bewerten können, müssen wir diesen zunächst einmal in Relation zu den beiden anderen Outputwerten setzen:

```
output[0] = 8.0f;
output[1] = 7.0f;
output[2] = 0.5f;
```

Eine Möglichkeit, wie wir diese Aufgabe bewerkstelligen können, besteht nun darin, die besagten Outputwerte mit Hilfe der *CNeuronEnsembleV2*-Methode *Calculate_ProbabilityValues()* in Wahrscheinlichkeitsangaben umzurechnen und diese im Anschluss daran mittels der ebenfalls in Listing 2 gezeigten *Get_ModifiedNeuronOutputValues()*-Methode auszulesen:

```
f = 1.0f / (8.0f + 7.0f + 0.5f);  // f = 0.0645
modifiedOutput[0] = 0.516f;  // 8 * 0.0645    (51.6%)
modifiedOutput[1] = 0.452f;  // 7 * 0.0645    (45.2%)
modifiedOutput[2] = 0.032f;  // 0.5 * 0.0645  (3.2%)
```

Sehr viel häufiger greift man bei der Umrechnung der Ausgabewerte allerdings auf die sogenannte Softmax-Funktion (normalisierte Exponentialfunktion) zurück. So liefert uns ein Aufruf der in Listing 2 skizzierten *Calculate_SoftmaxValues()*-Methode beispielsweise die nachfolgend gezeigten Resultate:

```
modifiedOutput[0] = 0.73
modifiedOutput[1] = 0.27
modifiedOutput[2] = 0
```

Die händische Berechnung der modifizierten Ausgabewerte ist in unserem konkreten Beispiel nicht sonderlich kompliziert, obgleich die Implementierung einer numerisch stabilen Softmax-Funktion keine ganz triviale Angelegenheit ist:

```
f = 1.0f / ( exp(output[0]) + exp(output[1]) + exp(output[2]) );

modifiedOutput[0] = f * exp(output[0]);
modifiedOutput[1] = f * exp(output[1]);
modifiedOutput[2] = f * exp(output[2]);
```

Maßgeschneiderte Aktivierungsfunktionen

Bei den zuvor betrachteten Fallbeispielen haben die jeweils verwendeten Aktivierungsfunktionen unserer intelligenten Neuronen einen mehr oder weniger ähnlichen Aufbau gehabt. In Abhängigkeit von der zu lösenden Aufgabe kann es allerdings durchaus von Vorteil sein, die eingetretenen Pfade hier und da mal zu verlassen und bei der Konzeption einer passenden Aktivierungsfunktion ein wenig Kreativität an den Tag zu legen. In diesem Zusammenhang schauen wir uns an dieser Stelle einmal drei weitere Möglichkeiten an (die zugehörigen Programmbeispiele finden Sie in der Datei *mainDendritePolynomApproximation.cpp*), mit denen sich die Veränderung der Aktivität eines virtuellen Einzellers in Abhängigkeit von der Umgebungstemperatur modellieren lässt. In einer ersten Variante simulieren wir den Aktivitätsverlauf mit Hilfe einer horizontal planierten (eingeebneten) Glockenkurve, die wir einerseits mit Hilfe des Parameters *scale* in die Höhe ziehen und andererseits auf einen Maximalwert von eins beschränken:

- Temperaturschwellenwert (*t*), bei dem die Aktivität maximal ist:
  `float t = 6.0f; float f = 0.5f; float scale = 3.0f;`
- Berechnung der Aktivität:
  ```
  tempFloat = input - t;
  neuronOutput = min(scale*exp(-f*tempFloat*tempFloat), 1.0f);
  ```

Für einen Temperaturbereich von 0 bis 10 ergibt sich nun der nachstehend gezeigte Aktivitätsverlauf:

- Ruhezustand (Temperaturwerte *0*, *1*, *2*, *3*, *4* zu gering):
  `(0 => 0) (1 => 0) (2 => 0) (3 => 0.0) (4 => 0.4)`
- maximale Aktivität (optimaler Temperaturbereich)
  `(5 => 1) (6 => 1) (7 => 1)`
- Ruhezustand (Temperaturwerte *8*, *9*, *10* zu hoch):
  `(8 => 0.4) (9 => 0) (10 => 0)`

Einen Plateaubereich mit einer maximalen Aktivität können wir alternativ auch durch eine Überlagerung mehrerer unterschiedlich breiter Aktivitäts-Peaks erzeugen. Je kleiner die Werte der nachstehend definierten Vorfaktoren *f1*, *f2* sowie *f3* sind, umso breiter fallen die zugehörigen Peaks aus:

- unterer Temperaturschwellenwert (*t1*), bei dem die Aktivität abzufallen beginnt:
  `float t1 = 3.5f; float f1 = 1.5f;`
- Temperaturschwellenwert (*t2*), bei dem die Aktivität maximal ist:
  `float t2 = 5.0f; float f2 = 0.5f;`
- oberer Temperaturschwellenwert (*t3*), bei dem die Aktivität wieder abzufallen beginnt:
  `float t3 = 6.5f; float f3 = 1.5f;`

Die Berechnung des Aktivitätsverlaufs erfolgt nun gemäß den nachfolgend gezeigten Schritten:

- **Schritt1** – Aktivität für den Temperaturschwellenwert (*t1*) berechnen:
  ```
  tempFloat = input - t1;
  neuronOutput = exp(-f1*tempFloat*tempFloat);
  ```
- **Schritt 2** – Aktivität für den Temperaturschwellenwert (*t2*) berechnen und zum zuvor berechneten Aktivitätszustand hinzuaddieren:
  ```
  tempFloat = input - t2;
  neuronOutput += exp(-f2*tempFloat*tempFloat);
  ```

- **Schritt 3** – Aktivität für den Temperaturschwellenwert (*t3*) berechnen und zu den zuvor berechneten Aktivitätszuständen hinzuaddieren:
```
tempFloat = input - t3;
neuronOutput += exp(-f3*tempFloat*tempFloat);
```
- **Schritt 4** – Aktivität auf einen Maximalwert von eins begrenzen:
```
neuronOutput = min(neuronOutput, 1.0f);
```

Für einen Temperaturbereich von 0 bis 10 ergibt sich nun der nachstehend gezeigte Aktivitätsverlauf:

- Ruhezustand (Temperaturwerte *0*, *1*, *2* zu gering):
(0 => 0) (1 => 0.00042) (2 => 0.045)
- maximale Aktivität (optimaler Temperaturbereich)
(3 => 0.82) (4 => 1) (5 => 1) (6 => 1) (7 => 0.82)
- Ruhezustand (Temperaturwerte *8*, *9*, *10* zu hoch):
(8 => 0.045) (9 => 0.00042) (10 => 0)

Im Zuge von Variante 3 müssen wir uns im ersten Schritt zunächst die folgenden beiden Temperaturschwellenwerte definieren:

- unterer Temperaturschwellenwert (*t1*), ab dem die Aktivität zunehmend größer wird:
```
float t1 = 2.5f; float f1 = 1.5f;
```
- oberer Temperaturschwellenwert (*t2*), ab dem die Aktivität wieder abzunehmen beginnt:
```
float t2 = 7.5f; float f2 = 1.5f;
```

Solange die Umgebungstemperatur (*input*) unterhalb des Schwellenwerts von *t1* liegt, liefert uns die nachstehend gezeigte Gleichung einen Funktionswert (*tempOutput1*) von eins zurück. Sobald die Temperatur einen höheren Wert als *t1* annimmt, verringert sich der Funktionswert zunehmend und nähert sich in Abhängigkeit von *f1* mehr oder weniger schnell einem Wert von null an. Je größer der Wert von *f1*, umso schneller verringert sich der Funktionswert:

```
tempFloat = max(input, t1) - t1;
tempOutput1 = exp(-f1*tempFloat*tempFloat);
```

Kommen wir nun auf die zweite Gleichung zu sprechen. Solange die Umgebungstemperatur oberhalb des Schwellenwerts von *t2* liegt, bleibt der Funktionswert (*tempOutput2*) gleich eins. Sobald die Temperatur einen tieferen Wert als *t2* annimmt, verringert sich der Funktionswert zunehmend und nähert sich in Abhängigkeit von *f2* mehr oder weniger schnell einem Wert von null an:

```
tempFloat = min(input, t2) - t2;
tempOutput2 = exp(-f2*tempFloat*tempFloat);
```

Im letzten Schritt können wir schließlich die Aktivität unseres virtuellen Einzellers unter Berücksichtigung der beiden zuletzt berechneten Funktionswerte *tempOutput1* sowie *tempOutput2* ermitteln:

```
neuronOutput = 1.0f – (tempOutput1 + tempOutput2);
```

Für einen Temperaturbereich von 0 bis 10 ergibt sich nun der nachstehend gezeigte Aktivitätsverlauf:

- Ruhezustand (Temperaturwerte *0*, *1*, *2*, *3* zu gering):
(0 => 0) (1 => 0) (2 => 0) (3 => 0.31)
- maximale Aktivität (optimaler Temperaturbereich)
(4 => 0.97) (5 => 1) (6 => 0.97)
- Ruhezustand (Temperaturwerte *7*, *8*, *9*, *10* zu hoch):
(7 => 0.31) (8 => 0) (9 => 0) (10 => 0)

Vorstellung eines leistungsfähigeren Neuronenmodells (dendritic Learning)

In sämtlichen neuronalen Netzwerken, die in unseren Programmbeispielen bislang zum Einsatz gekommen sind, wurde im Zuge des Lernprozesses lediglich der Gewichtungsfaktor modifiziert. Mit diesem wurde dann das von einer Nervenzelle ausgehende Signal multipliziert, bevor es im nächsten Schritt von einem nachgeschalteten Neuron verarbeitet wird. Wenn man sich einmal Bilder von diversen biologischen Nervenzellen betrachtet, so stellt man fest, dass die Zellfortsätze, über die die Reizaufnahme erfolgt (die sogenannten Dendriten), ihrerseits mehr oder weniger komplexe Netzwerke und Baumstrukturen (dendrites, zu Deutsch: zum Baum gehörend) ausbilden. Die Dendriten sind mitnichten nur einfache Leitungsbahnen. Sie dienen gleichermaßen der Filterung und Weiterverarbeitung der eingehenden Signale und dürften, da es sich um keine starren Gebilde handelt, auch für den Lernprozess eine zentrale Rolle spielen. Die zusätzlichen Möglichkeiten, die sich uns an dieser Stelle bieten, könnten Sie unter anderem auch anhand des in **Abbildung 1** illustrierten Neuronenmodells mit Dendriten nachvollziehen. So übernimmt einer der dargestellten Dendriten (der obere) beispielsweise die Vorverarbeitung zweier Eingangssignale, während zwei der anderen Dendriten ein und dasselbe Eingangssignal auf unterschiedliche Weise verarbeiten. Mit Hilfe unserer neuronalen Netzwerke haben wir bislang lediglich das synaptische Lernverhalten simuliert (die einzelnen Gewichtungsfaktoren repräsentieren die sogenannte synaptische Plastizität). Die Bedeutung des dendritischen Lernens (dendritic Learning) hatten wir hingegen bislang noch überhaupt nicht auf dem Radar. Bringen wir es auf den Punkt: ohne Dendriten keine intelligenten Nervenzellen. Die bereits mehrfach angesprochenen RBF-Neuronen repräsentieren gewissermaßen den ersten Versuch, um die Arbeitsweise der Dendriten zumindest ansatzweise nachzubilden. Die Centroid-Werte und Vorfaktoren der zuvor betrachteten Aktivierungsfunktionen lassen sich demensprechend als mathematisches Äquivalent zu den Dendriten einer Nervenzelle auffassen. Unter Zuhilfenahme der *CNeuronV2*-Klasse, die in unseren neuesten Programmbeispielen zum Einsatz kommt, können wir allerdings nicht nur einfache RBF-Neuronen simulieren, sondern auch deutlich komplexere dendritische Prozesse. Hierfür ist es zunächst einmal erforder-

lich, dass wir uns maßgeschneiderte Aktivierungs- und Übertragungsfunktionen definieren können, die je nach Aufgabenstellung deutlich komplizierter aufgebaut sind, als die eingangs vorgestellten Aktivierungsfunktionen (Tanh, ReLU, binäre Aktivierung usw.). Der Aufruf der besagten Funktionen erfolgt im Rahmen eines Aufrufs der *CNeuronV2*-Methoden *Calculate_NeuronOutput()* sowie *Propagate_SynapticOutput()* unter Zuhilfenahme der nachstehend definierten Funktionszeiger:

```
typedef void(*pActivationFuncV2)(CNeuronV2 *pNeuron);
typedef void(*pSynapticFuncV2)(CNeuronV2 *pNeuron);
```

- Berechnung der neuronalen Aktivität (*NeuronOutput*)

```
void CNeuronV2::Calculate_NeuronOutput(void)
{ pActivationFunction(this); }
```

- Weiterleitung der in der Variablen *NeuronOutput* gespeicherten Aktivität an einzelne oder sämtliche nachgeschaltete Neuronen:

```
void CNeuronV2::Propagate_SynapticOutput(void)
{ pSynapticFunction(this); }
```

Für die praktische Umsetzung des dendritischen Lernprozesses können wir unter anderem auf die in den Listings 3 und 4 vorgestellten *CNeuronV2*-Methoden zurückgreifen. Im ersten Schritt muss zunächst ein Aufruf der in Listing 3 skizzierten *Calculate_Error()*-Methode erfolgen. Im Zuge dieses Methodenaufrufs wird erst einmal die Abweichung zwischen der tatsächlich berechneten neuronalen Aktivität (dem Istwert) und dem angestrebten Sollwert (*desiredNeuronOutput*) ermittelt:

```
variance = desiredNeuronOutput - NeuronOutput;
```

Unter Berücksichtigung der zuvor ermittelten Abweichung können wir im Anschluss daran einen Fehlerwert (*ErrorValue*) berechnen, mit dessen Hilfe sich die uns bereits hinlänglich bekannten Centroid-Werte und Vorfaktoren modifizieren lassen:

```
ErrorValue = variance * errorFactor1 *
  exp(-(errorFactor2 * NeuronOutput * NeuronOutput));
```

Sofern man im Verlauf des Trainings- und Lernprozesses den Centroid-Wert einzelner Dendriten wie nachstehend gezeigt modifizieren möchte, bietet sich ein Aufruf der in Listing 4 gezeigten *Adjust_Dendrite_CentroidValue_AfterErrorCalculations()*-Methode an:

```
pDendrite_CentroidValueArray[dendriteID] += learningRate*ErrorValue*
  pDendrite_InputValueArray[dendriteID];
```

Die Vorfaktoren einzelner Dendriten können hingegen zielgenau unter Zuhilfenahme der ebenfalls in Listing 4 illustrierten *Adjust_Dendrite_Factor_AfterErrorCalculations()*-Methode wie folgt modifiziert werden:

```
pDendrite_FactorArray[dendriteID] += learningRate*ErrorValue*
  pDendrite_InputValueArray[dendriteID];
```

Neuronale Funktionsapproximation

In den vorangegangenen Artikeln sind wir bereits mehrfach darauf zu sprechen gekommen, auf welche Weise sich Funktionsverläufe mit Hilfe eines neuronalen Netzwerks approximieren lassen. Im heutigen Artikel möchte ich Ihnen im Unterschied dazu demonstrieren, dass wir zur Bewältigung dieser Aufgabe genauso gut auf ein intelligentes Neuron zurückgreifen können, welches zum dendritischen Lernen befähigt ist. Die zugehörigen Programmbeispiele finden Sie in der Datei *mainDendritePolynomApproximation.cpp*. Das erste Programm demonstriert Ihnen die Funktionsapproximation mit Hilfe eines Backpropagation-Lernverfahrens, wohingegen das zweite Programmbeispiel auf dem Konzept der Neuroevolution basiert. Konkret ausgedrückt besteht die Aufgabe für beide Programme darin, ein zu einem vorgegebenen Funktionsverlauf passendes Polynom dritten Grades zu bestimmen.

- Polynom dritten Grades mit vier trainierbaren Parametern:
  `y = a*x*x*x + b*x*x + c*x + offset`

Während sich die Werte der vier Parameter *offset*, *a*, *b* und *c* im Verlauf der Neuroevolution völlig problemlos ermitteln lassen, funktioniert selbiges im Rahmen eines Backpropagation-Lernverfahrens mehr schlecht als recht. Um das Training in einem vertretbaren Zeitrahmen nach einer akzeptablen Anzahl von Trainingsdurchläufen abschließen zu können, ist es erforderlich, die Zahl der trainierbaren Parameter in der nachstehend gezeigten Weise zu vergrößern.

- Polynom dritten Grades mit sieben trainierbaren Parametern:
  `y = a1*x*x*x + a2*x*x*x + b1*x*x + b2*x*x + c1*x + c2*x + offset`

Funktionsapproximation mittels Backpropagation

Wir werden uns an dieser Stelle zunächst einmal damit befassen, wie sich eine solche Funktionsapproximation im Rahmen eines Backpropagation-Lernverfahrens (siehe hierzu Listing 7) umsetzen lässt. Als Aktivierungsfunktion nutzen wir die in Listing 5 skizzierte *PolynomApproximationOutputFunction1()*, welche für die Handhabung von sieben trainierbaren Funktionsparameter (*a1*, *a2*, *b1*, *b2*, *c1* sowie *c2*) verantwortlich ist:

```
CNeuronV2 Neuron;
Neuron.Set_ActivationFunction(PolynomApproximationOutputFunction1);
```

Da es sieben trainierbare Funktionsparameter gibt, müssen wir unser Neuron entsprechend mit sieben Dendriten ausstatten, deren Charakteristika wir schlicht und einfach nach dem Zufallsprinzip festlegen:

```
Neuron.Init_Dendrite_Arrays(7);
Neuron.Randomize_Dendrite_Factors(&RandomNumbers, -0.01f, 0.01f);
```

Um es nicht unnötig kompliziert zu machen, erfolgen sämtliche Trainingsdurchgänge innerhalb einer schlichten *for*-Schleife:

```
for (int32_t i = 0; i < maxCount; i++)
{
  // [Trainingsdurchgang]
}
```

Zu Beginn eines neuen Trainingsdurchgangs (einer sogenannten Epoche) setzen wir den Wert der Variablen *errorSum* (Summe aller Approximationsfehler) zunächst einmal zurück auf null und inkrementieren im Anschluss daran den Epochen-Zähler:

```
errorSum = 0.0f;
epoch++;
```

Im Zuge eines Trainingsdurchgangs steht unser Neuron vor der Aufgabe, alle Funktionswerte, die den im *InputData*-Array hinterlegten Zahlenwerten zugeordnet sind, so genau wie möglich zu reproduzieren:

```
// Trainingsdurchgang:
for (int32_t k = 0; k < NumInputOutputValues; k++)
{
// Trainingsschritt:
/* [Approximierung eines Funktionswerts, Fehlerberechnung, Lernprozeß] */
}
```

Jeder einzelne Trainingsschritt beginnt erst einmal mit der Approximierung eines weiteren Funktionswerts:

```
Neuron.Set_NeuronInput(InputData[k]);
Neuron.Calculate_NeuronOutput();
```

Im Anschluss daran berechnen wir unter Zuhilfenahme der *Calculate_Error()*-Methode einerseits den Approximationsfehler, also die quadratische Abweichung des approximierten Funktionswerts vom jeweiligen Sollwert (*DesiredOutputData[k]*) und andererseits einen Fehlerwert, den wir im Rahmen des Lernprozesses für die Modifizierung der trainierbaren Parameter benötigen:

```
errorSum += Neuron.Calculate_Error(DesiredOutputData[k],
  /*errorFactor1:*/ 1.0f, /*errorFactor2:*/ 0.001f);
```

- Modifizierung des *offset*-Parameters:
  ```
  Neuron.Adjust_Dendrite_Factor_AfterErrorCalculations(
    /*dendriteID:*/ 0, /*learningRate:*/ 0.01f);
  ```
- Modifizierung der übrigen Funktionsparameter (*a1*, *a2*, *b1*, *b2*, *c1* und *c2*):
  ```
  Neuron.Adjust_Dendrite_Factors_AfterErrorCalculations(
    /*learningRate:*/ 0.0001f, /*minDendriteID:*/ 1,
    /*maxDendriteID:*/ Neuron.Num_Of_Dendrite_Elements - 1);
  ```

Liegt die Summe aller Fehler, die unser Neuron bei der Approximierung der einzelnen Funktionswerte gemacht hat, unterhalb des maximal tolerierbaren Fehlers von *0.001*, können wir das Training schließlich abbrechen:

```
if (errorSum < 0.001f)
  break;
```

Funktionsapproximation mittels Neuroevolution

Schauen wir uns jetzt einmal an, wie wir das Problem der Funktionsapproximation mit Hilfe der Neuroevolution (Listing 8) lösen können. Erst einmal müssen wir uns genügend Speicherplatz für die zu trainierenden Neuronen samt den zugehörigen Fitnesswerten anfordern:

```
CNeuronV2 *pNeuronArray = new (std::nothrow) CNeuronV2
                                                [TrainingPopulationSize];
float *pFitnessScoreArray = new (std::nothrow) float[TrainingPopulationSize];
```

Im Rahmen unseres Programmbeispiels greifen wir für die Handhabung einer Population von *CNeuronV2*-Instanzen auf ein Objekt der *CDendriteNeuronPopulation*-Klasse zurück:

```
CDendriteNeuronPopulation NeuronPopulation;
NeuronPopulation.Initialize(TrainingPopulationSize);
```

Mit Hilfe der *Set_Neuron()*-Methode müssen wir die zu trainierenden *CNeuronV2*-Instanzen zunächst in die von einem *CDendriteNeuronPopulation*-Objekt verwaltete Population eingliedern:

```
for (int32_t i = 0; i < TrainingPopulationSize; i++)
  NeuronPopulation.Set_Neuron(&pNeuronArray[i], i);
```

Als Aktivierungsfunktion nutzen wir dieses Mal die in Listing 6 illustrierte *PolynomApproximationOutputFunction2()*. Im Unterschied zur Aktivierungsfunktion, die wir im Zuge des zuvor betrachteten Backpropagation-Lernverfahrens verwendet haben, ist die in Listing 6 gezeigte Funktion lediglich für die Handhabung von vier trainierbaren Funktionsparametern (*offset*, *a*, *b* sowie *c*) verantwortlich:

```
NeuronPopulation.Set_ActivationFunction(
  PolynomApproximationOutputFunction2);
```

Da es vier trainierbare Funktionsparameter gibt, müssen wir unsere Neuronen entsprechend mit vier Dendriten ausstatten, deren Charakteristika wir wie gehabt nach dem Zufallsprinzip festlegen:

```
NeuronPopulation.Init_Or_Reinitialize_Dendrite_Arrays(4);
NeuronPopulation.Randomize_Dendrite_Factors(-2.0f, 2.0f);
```

Für die Handhabung der einzelnen Evolutionsschritte greifen wir wiederum auf eine einfache *for*-Schleife zurück:

```
for (int32_t j = 0; j < NumTrainingGenerationsMax; j++)
{
  /* [Handhabung eines Evolutionsschritts] */
}
```

Im Verlauf eines Evolutionsschritts müssen sämtliche Individuen (Neuronen) der Populationen nacheinander einen Trainingsdurchgang absolvieren, an dessen Ende sich die Berechnung des von den Trainingsresultaten abhängigen Fitnesswerts anschließt:

```
// Evolutionsschritt:
NeuronPopulation.Reset_MinErrorSum_ActualGeneration();
for (int32_t i = 0; i < TrainingPopulationSize; i++)
{
  /* [Trainingsdurchgang mit sich anschließender
    Fitnesswert-Berechnung] */
}
```

Im Rahmen eines Trainingsdurchgangs steht das jeweils betrachtete Neuron vor der Aufgabe, sämtliche Funktionswerte, die den im *InputData*-Array hinterlegten Zahlenwerten zugeordnet sind, so genau wie möglich zu reproduzieren:

```
// Trainingsdurchgang:
errorSum = 0.0f;
for (int32_t k = 0; k < NumInputOutputValues; k++)
{
/* [Approximierung eines Funktionswerts, Fehlerberechnung] */
}
/* [Fitnesswertberechnung]*/
```

Jeder einzelne Trainingsschritt beginnt zunächst mit der Approximierung eines weiteren Funktionswerts:

```
pNeuronArray[i].Set_NeuronInput(InputData[k]);
pNeuronArray[i].Calculate_NeuronOutput();
```

Nun haben wir zwei Möglichkeiten, wie von uns fortzufahren ist. Im Rahmen von Variante 1 (Neuroevolution in Reinform) würden wir einfach nur die Approximationsfehler, mit anderen Worten, die quadratischen Abweichungen der approximierten Werte von den jeweiligen Sollwerten (*DesiredOutputData[k]*) ermitteln und für die spätere Fitnesswertberechnung aufsummieren:

```
errorSum += pNeuronArray[i].Calculate_VarianceSq(DesiredOutputData[k]);
```

In dem hier betrachteten Fallbeispiel bietet sich uns darüber hinaus auch die Möglichkeit, die Evolution der einzelnen Individuen zusätzlich noch durch einen zwischengeschalteten Backpropagation-Lernschritt ein wenig zu beschleunigen:

```
errorSum += pNeuronArray[i].Calculate_Error(DesiredOutputData[k],
  /*errorFactor1:*/ 1.0f, /*errorFactor2:*/ 0.001f);
```

- Modifizierung des *offset*-Parameters:

```
pNeuronArray[i].Adjust_Dendrite_Factor_AfterErrorCalculations(
  /*dendriteID:*/ 0, /*learningRate:*/ 0.01f);
```

- Modifizierung der Funktionsparameter *a*, *b* und *c*:

```
pNeuronArray[i].Adjust_Dendrite_Factors_AfterErrorCalculations(
  /*learningRate:*/ 0.0001f, /*minDentriteID:*/ 1,
  /*maxDentriteID:*/ pNeuronArray[i].Num_Of_Dendrite_Elements - 1);
```

Nach Abschluss eines Trainingsdurchgangs berechnen wir für jedes Individuum der Population den Fitnesswert. Je größer dieser Wert ist, umso weniger Fehler sind dem betreffenden Neuron bei der Approximation der einzelnen Funktionswerte unterlaufen:

```
pFitnessScoreArray[i] += 1.0f / (errorSum + 0.01f);
NeuronPopulation.Update_MinErrorSum_ActualGeneration(errorSum);
```

Unter Einbeziehung der zuvor berechneten Fitnesswerte steht als nächster Tagesordnungspunkt die Aktualisierung der Population mit Hilfe der *Update_Population()*-Methode auf dem Programm. Sobald die Summe der Fehler, die das am besten angepasste Neuron (größter Fitnesswert) bei der Approximation der einzelnen Funktionswerte gemacht hat, unterhalb des maximal tolerierbaren Fehlers von *0.01* liegt, können wir auf die Durchführung weiterer Evolutionsschritte verzichten:

```
NeuronPopulation.Update_Population(pFitnessScoreArray);
if (NeuronPopulation.MinErrorSum_ActualGeneration < 0.01f)
  break;
```

Kommen wir nun auf den Evolutionsprozess zu sprechen. Der nachfolgende Aufruf der *Update_BaseEvolution_Dendrite_Factors()*-Methode führt zu diversen Mutationen bei den noch nicht so gut angepassten Individuen, wohingegen die zehn Neuronen mit den bislang höchsten Fitnesswerten unangetastet bleiben. Die Wahrscheinlichkeit, dass im Zuge des Funktionsaufrufs ein mutiertes Individuum entsteht, das beim Training besser abschneidet als die übrigen Individuen der Population, nimmt im Verlauf des Evolutionsprozesses kontinuierlich ab:

```
NeuronPopulation.Update_BaseEvolution_Dendrite_Factors(
/*minVariance:*/ -0.1f, /*maxVariance:*/ 0.1f, /*mutationRate:*/ 0.75f);
```

Im Rahmen eines Aufrufs der *Update_Evolution_BestBrainOnly_Dendrite_Factors()*-Methode wird auf Grundlage des aktuell am besten angepassten Individuums temporär ein zusätzliches mutiertes Individuum kreiert. Das mutierte Individuum wird im Zuge des nächsten *Update_Population()*-Aufrufs allerdings nur dann in die Population eingegliedert, wenn es nach Abschluss eines erneuten Trainingsdurchgangs zu den drei am besten angepassten Individuen zählt und es darüber hinaus auch keine weiteren temporären Individuen mit höheren Fitnesswerten gibt:

```
NeuronPopulation.Update_Evolution_BestBrainOnly_Dendrite_Factors(
/*minVariance:*/ -0.1f, /*maxVariance:*/ 0.1f, /*mutationRate:*/ 0.75f);
```

Im Zuge eines Aufrufs der *Update_Evolution_SecondBestBrainOnly_Dendrite_Factors()*-Methode wird auf Grundlage des aktuell am zweitbesten angepassten Individuums temporär ein zusätzliches mutiertes Individuum kreiert. Das mutierte Individuum wird im Rahmen des nächsten *Update_Population()*-Aufrufs wiederum nur dann in die Population eingegliedert, wenn es nach Abschluss eines erneuten Trainingsdurchgangs zu den drei am besten angepassten Individuen zählt und darüber hinaus auch keine weiteren temporären Individuen mit höheren Fitnesswerten existieren:

```
NeuronPopulation.Update_Evolution_SecondBestBrainOnly_Dendrite_Factors(
/*minVariance:*/ -0.1f, /*maxVariance:*/ 0.1f, /*mutationRate:*/ 0.75f);
```

Während sich mit Hilfe der *Update_Evolution_Combine_BestTwoBrains()*-Methode die beiden am besten angepassten Individuen kreuzen lassen, erfolgt bei einem Aufruf der *Update_Evolution_Combine_TwoBrains()*-Methode die Auswahl der an einer Kreuzung beteiligten Individuen nach dem Zufallsprinzip. Das bei einer solchen Kreuzung entstehende temporäre Individuum wird im Verlauf des nächsten *Update_Population()*-Aufrufs wie gehabt nur dann in die Population eingegliedert, wenn es nach Abschluss eines erneuten Trainingsdurchgangs zu den drei am besten angepassten Individuen zählt und es darüber hinaus auch keine weiteren temporären Individuen mit größeren Fitnesswerten gibt:

```
NeuronPopulation.Update_Evolution_Combine_BestTwoBrains();
NeuronPopulation.Update_Evolution_Combine_TwoBrains();
```

Nach Abschluss des Evolutionsprozesses erzeugen wir uns im letzten Schritt schließlich noch eine Kopie desjenigen Neurons, welches den von uns vorgegebenen Funktionsverlauf im letzten Trainingsdurchgang am genauesten reproduzieren konnte:

```
CNeuronV2 BestNeuron;
NeuronPopulation.Get_Best_Evolved_Neuron(&BestNeuron);
```

Alexander Rudolph schrieb zwei Bücher über das Thema Spieleprogrammierung mit DirectX für den Markt+Technik Verlag, eine Artikelserie über die Spieleentwicklung mit OpenGL und OpenAL und war an der Entwicklung sowohl der Grafik- wie auch der Physik-Engine für das Spiel Söldner 2 beteiligt. In regelmäßigen Abständen veröffentlicht er auf seiner Webseite spieleprogrammierung.net Artikel und Tutorials über den Einsatz der modernen Vulkan-API sowie der OpenGL Spezifikationen 3.x und 4.x und ist als freier Mitarbeiter im Bereich 3-D-Programmierung tätig. Seine eBooks „Spieleentwicklung – Mathematik mit Fun-Faktor“ und „Spieleentwicklung – OpenGL mit Fun-Faktor“ vermitteln Einblicke in die Welt der Spieleprogrammierung.

Das ServiceNow Automated Test Framework

Gut getestet läuft besser

Im Entwickler Magazin 6.2020 haben wir uns die SaaS-Plattform ServiceNow ein wenig genauer angesehen. Heute zeigen wir, wie man Applikationen innerhalb von ServiceNow testen kann. Die Plattform stellt hierfür ein Test-Framework bereit. In diesem Artikel gehen wir darauf ein, wie das Automated Test Framework (ATF) von ServiceNow arbeitet. Wir veranschaulichen an einem Beispiel, wie schnell sich einfache Tests aufsetzen lassen. Darüber hinaus gehen wir auf einige Punkte ein, die man beachten sollte, wenn man mit dem ATF arbeitet.

von Martin Mohr

Warum das Testen von Applikationen so wichtig ist, hat vermutlich jeder Entwickler schon einmal leidvoll erfahren müssen. Man macht nur eine kleine Änderung kurz vor Feierabend, und erst in Produktion fällt auf, dass Basisfunktionalitäten nicht mehr richtig arbeiten. So etwas kann einem schon den Tag vermiesen. Da Testen selbst eine recht langweilige Angelegenheit ist, macht das kein Entwickler gerne. Prinzipiell müsste man bei jeder kleinen Codeänderung überprüfen, ob die komplette Applikation noch fehlerfrei arbeitet. Man muss auch bedenken, dass die Aufmerksamkeit eines Softwaretesters nachlässt, wenn er einen bestimmten Workflow zum x-ten Mal mit leicht unterschiedlichen Eingangsparametern testet. Das sind nur einige der Gründe, warum man so viel wie nur möglich automatisch testen sollte. Automatische Tests haben auch den ungemeinen Vorteil, dass sie immer in gleicher Qualität durchgeführt werden. Bei einem Softwaretester kann schon einmal das Koffeinlevel ein entscheidender Faktor für den Ausgang eines Tests sein.

Die meisten vorhandenen Test-Frameworks sind dazu gedacht, das Frontend einfacher Webapplikationen zu testen. Man ist immer gezwungen, die passenden Testdaten im System schon vorab zu erzeugen. Falls der Test

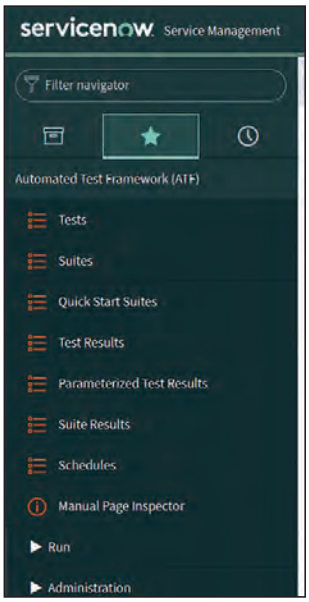


Abb. 1: Das Menü der ATF-Applikation

ungewollt Daten modifiziert, kann das bei Folgetests zu Problemen führen. Alles in allem ist das Testen von Applikationen keine Sache, die man so nebenher erledigen kann. Das ist wohl auch der Grund, warum Softwaretestexperten in jedem größeren Projekt mit dabei sind. In den nächsten Absätzen sehen wir uns an, wie sich ServiceNow der Thematik Applikations-Testing nähert.

ATF-Aufbau

Die drei Grundpfeiler des ATF sind der Test-Step, der Test und die Testsuite. Nach einer kurzen Übersicht, wie diese Teile zusammenpassen, sehen wir uns in den folgenden Abschnitten detailliert an, wie sie funktionieren. Das kleinste Zahnrad im Getriebe ist der sogenannte Test-Step. Die eigentlichen Tests sind aus beliebig vielen Test-Steps aufgebaut. Möchte man mehrere Tests kombinieren, kann dazu die Testsuite verwendet werden. Bevor wir uns intensiver mit dem Thema ATF beschäftigen, ist es wichtig zu sagen, dass ATF eine eigenständige Applikation innerhalb der ServiceNow-Plattform ist. Das ATF hat in der Navigation eigene Menüeinträge. Zunächst sehen wir uns kurz an, welche Funktionen sich hinter den einzelnen Menüeinträgen, die in **Abbildung 1** zu sehen sind, verbergen.

- Tests: Bearbeiten vorhandener Tests und Erstellen neuer Tests
- Suites: Bearbeiten vorhandener Test-Suites und Erstellen neuer Test-Suites
- Quick Start Suites: vordefinierter Filter, der nur Tests anzeigt, bei denen die Protection Policy = read only ist
- Test Results: Ergebnisse von Testläufen
- Parameterized Test Results: Ergebnisse von parametrisierten Testläufen
- Suite Results: Ergebnisse von Test-Suites
- Schedules: zeitgesteuertes Ausführen von Test-Suites
- Run: alles rund um die Ausführung von Tests; Statusinformationen laufender und wartender Test-Suites
- Administration: Eigenschaften des ATF verwalten und anzeigen; eigene Test-Steps erstellen

Nun, da wir uns einen groben Überblick über die Navigationseinträge verschafft haben, können wir etwas tiefer in die einzelnen Punkte einsteigen.

Test-Step

Der Test-Step ist die kleinste Komponente eines Tests. In der aktuellen Version von ServiceNow gibt es 96 unterschiedliche Test-Steps, mit denen man die verschiedenen Teile einer Applikation testen kann. Interessant hierbei ist, dass es auch Test-Steps gibt, die die Datenbasis ändern können. Um bei den zahlreichen Testmöglichkeiten den Überblick nicht zu verlieren, sind die einzelnen Test-Steps in Kategorien aufgeteilt. Mit den vorhandenen Test-Steps kann man so gut wie jede Funktion in einer ServiceNow-Anwendung testen. Sollte es allerdings durch eine Eigenentwicklung vorkommen, dass kein passender Test-Step im Fuhrpark ist, kann man auch leicht eigene Test-Steps entwickeln. Diese lassen sich dann in die vorhandenen Tests mit einbauen.

Test

Ein Test wird aus verschiedenen Test-Steps zusammengesetzt. So ist es möglich, eine bestimmte Funktionalität zu testen. Ein Test kann auf verschiedene Arten ausgeführt werden. Zum einen kann man ihn direkt ausführen und erhält gleich die Testergebnisse. Es ist aber auch möglich, den Test mit Hilfe einer Test-Suite ausführen zu lassen. Die Tests benötigen immer einen Browser für den Test-Runner. Falls noch kein Tab/Browser mit einem Test-Runner aktiv ist, wird ein neuer gestartet. Bitte beachten Sie, dass der Test-Runner bei den meisten Browsern nur ausgeführt wird, wenn sich der entsprechende Tab im Vordergrund befindet. Alle modernen Browser verwenden Algorithmen zur Verbesserung der Performance. Das bedeutet, dass nur Inhalte, die auch gesehen werden, Rechenleistung bekommen. Befindet sich der Test-Runner im Hintergrund, wird er nicht ausgeführt. Bei einigen Browsern kann man in den erweiterten Einstellungen vorgeben, wie mit Hintergrundtabs umgegangen werden soll. Sollten Sie dieses Problem haben, googeln Sie bitte nach den Einstellungen für Ihren konkreten Browser. In diesem Zusammenhang sollte man auch erwähnen, dass es mittlerweile die Möglichkeit gibt, Browser im Headless-Modus zu starten. Mit dieser Funktion kann man einen Browser simulieren, ohne dass eine Ausgabe erzeugt wird. Dieser Modus ist für automatische Tests im Allgemeinen sehr nützlich. Im Fall des ATF könnte man den Test-Runner auf einem Server laufen lassen. Wie wir im nächsten Abschnitt sehen werden, gibt es durchaus gute Gründe, dieses Set-up zu wählen.

Die einzelnen Test-Steps können mit Hilfe von sogenannten Data Pills Informationen von einem Schritt zum nächsten weitergeben. Damit ist es möglich, auch komplexere Testabläufe aufzubauen. Man kann sich zum Beispiel einige Datensätze erzeugen und sie dann auch in weiteren Tests immer wieder verwenden.

Test-Suite

Kommen wir nun zum letzten Baustein des ServiceNow-Test-Frameworks. Mit der Test-Suite können verschiedene Tests zu einem größeren Systemtest zusammengefügt werden. Jede Test-Suite kann weitere Test-Suites beinhalten, so ist es möglich, hochkomplexe Testszenarien für unterschiedliche Anwendungsfälle aufzubauen. Hierbei können Tests ebenso wie Test-Suites in unterschiedlichen Test-Suites verwendet werden. Damit ist es möglich, dass ein bestimmter Test nur

einmal entwickelt werden muss, um ihn dann mehrfach verwenden zu können.

Bei Tests verhält es sich wie bei Software, man sollte nie eine Funktion zweimal implementieren. „Copy and Paste"-Entwicklung ist auch bei Tests keine gute Idee. Der Hersteller empfiehlt tatsächlich, vorhandene Tests zu kopieren und sie dann an die neuen Gegebenheiten anzupassen. Das kann unter Umständen aufwendiger sein als schnell einen neuen Test zusammenzuklicken. Hier sollte man von Fall zu Fall entscheiden, was der bessere Weg ist.

Test-Suites können auch zeitgesteuert ausgeführt werden, so kann man die Tests zu Zeiten laufen lassen, in denen wenige oder im besten Fall gar keine User auf dem System arbeiten. Man muss immer im Hinterkopf behalten, dass Tests eine nicht zu verachtende Systemlast verursachen. Das ist der Grund, warum das ATF ab Werk erst einmal abgeschaltet ist. Der Hersteller rät davon ab, ATF auf produktiven Systemen einzusetzen. Wird eine Test-Suite zeitgesteuert ausgeführt, muss man dafür sorgen, dass man zum entsprechenden Zeitpunkt auch einen Test-Runner aktiv hat. Hier kommen die bereits erwähnten Headless-Browser wieder ins Spiel. Mit ihrer Hilfe lässt sich der nötige Test-Runner für eine zeitgesteuerte Test-Suite gut bereitstellen.

Wenn Sie in Ihrer Applikation mehrere ähnliche Testfälle haben, kann der Einsatz von Templates die Erstellung von Tests erheblich beschleunigen. Ein Template besteht aus mehreren Testschritten, die nach dem Einfügen nur noch konfiguriert werden müssen.

Testergebnisse

Nachdem wir uns intensiv mit dem Thema Testing beschäftigt haben, werfen wir einen kurzen Blick auf die Testergebnisse. Die einfachste und am Anfang auch interessanteste Methode ist, dem Test-Runner bei der Arbeit zuzusehen. Man kann nachverfolgen, wie nach und nach alle Test-Steps ausgeführt werden. Falls Fehler auftreten, kann man sie mit etwas Glück direkt erkennen. Leider bricht der Test-Runner im Fehlerfall recht schnell die Ausführung ab, ohne dass man sieht, was genau den Fehler verursacht hat. Das ist nicht gerade das Auswertungsverfahren, das man sich bei einem automatischen Test wünscht. Besser ist es, die Logfiles des Tests auszuwerten. Zu jedem einzelnen Test-Step existiert ein Logfile, das im Fehlerfall Auskunft darüber gibt, was bei dem Schritt versucht wurde und ob es funktioniert hat. Wenn ein Fehler im Test auftritt, wird er berichtet und der Test wird abgebrochen. Ist der Test Bestandteil einer Test-Suite, führt diese einfach den nächsten Test aus. Das ist möglich, weil das ATF nach jedem abgeschlossenen Test, egal ob erfolgreich oder nicht, alle Transaktionen im System wieder komplett rückgängig macht. Es ist also nicht nötig, die Datenbestände nach einem Test wieder auf einen definierten Stand zu bringen. Ein weiteres wirklich tolles Feature des ATF sind die Screenshots, die bei jedem Testschritt erzeugt werden. Mit ihrer Hilfe und den Logfiles ist es fast unmöglich, nicht herauszufinden, was den Abbruch verursacht hat.

Abb. 2: Neu erstellter Test

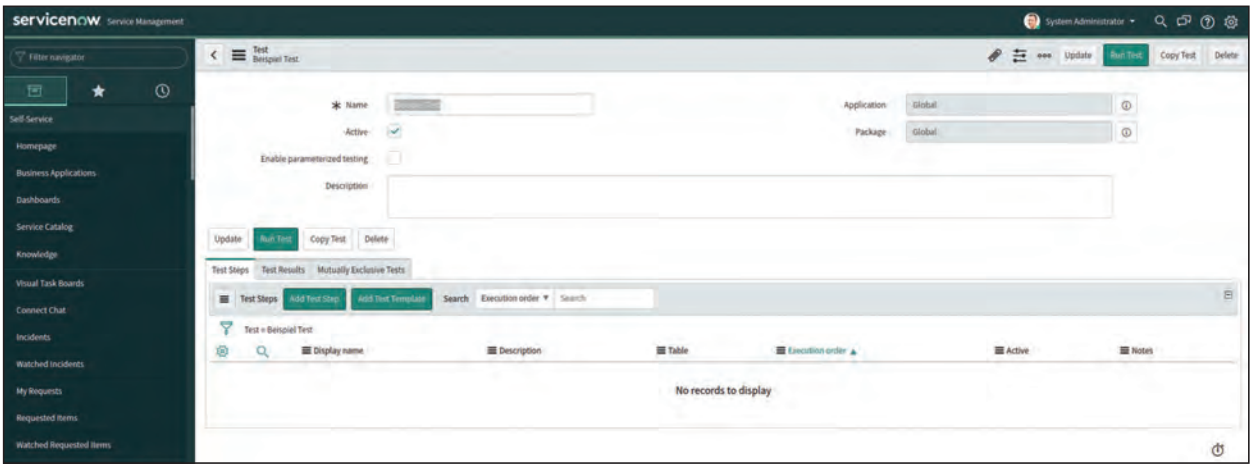
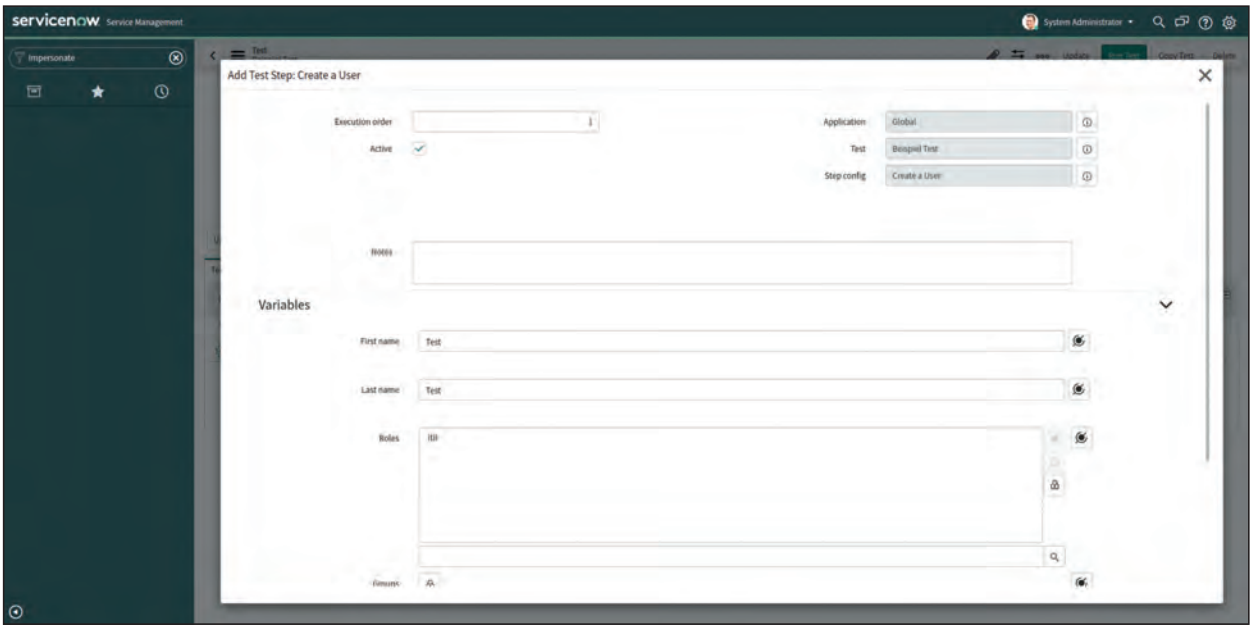


Abb. 3: Anlegen eines Users für den Test



Rollen

Innerhalb des ATF gibt es unterschiedliche, praxisnahe Rollen, die ein Tester haben kann. Sollten Sie das ATF in Ihrer persönlichen Entwicklerinstanz nur einmal ausprobieren wollen, können Sie auch einfach als Administrator arbeiten. Der Administrator hat alle ATF-Rollen, das sind die folgenden:

- *Testdesigner*: Der Testdesigner kann Tests und Test-Suites erstellen, bearbeiten und ausführen. Weiterhin kann er die Testergebnisse sehen und die ATF-Systemeigenschaften lesen. Üblicherweise reicht die Rolle des Testdesigners für einen Tester.
- *Testadministrator*: Der Testadministrator hat alle Berechtigungen, die auch der Testdesigner hat. Darüber hinaus kann er alle ATF-relevanten Systemeigenschaften verändern. Auch das Erstellen von eigenen Testschritt-konfigurationen ist dem Testadministrator vorbehalten.
- *Web-Service-Tester*: Der Web-Service-Tester hat die gleichen Rechte wie der Testdesigner. Der Web-Service-Tester kann aber zusätzlich Tests für Web Services erstellen.

Praxistest

Nach all der trockenen Theorie über das ATF sehen wir uns nun an einem praktischen Beispiel an, wie man einen Test erstellt und danach ausführt. Das Beispiel ist bewusst einfach gehalten, um den Rahmen des Artikels nicht zu sprengen. Falls Sie eine tiefergehende Einführung in das ATF wünschen, kann ich Ihnen den ServiceNow-Onlinekurs [1] zum Test-Framework empfehlen.

Für die folgenden Schritte benötigen wir eine ServiceNow-Entwicklerinstanz. Sie kann einfach auf der ServiceNow-Entwicklerhomepage [2] bestellt werden. Wie genau das geschieht, können Sie in Ausgabe 6.2020 im Artikel „Es wird Zeit für einen besseren Service“ [3] nachlesen. Um einen neuen Test zu erstellen, wechseln wir zunächst in die Liste der Tests, die schon im System vorhanden sind. Am einfachsten bewerkstelligen wir das, indem wir in den Filternavigator den Namen der Tabelle eingeben: *sys_atf_test.list*. Das *.list* sagt ServiceNow, dass wir die Listenansicht der Daten wünschen. Hier klicken wir nun oben auf den New-Button, um einen neuen Test zu erstellen. Auf der Seite, die sich nun öffnet, ist oben ein Hinweis zu finden, dass das Testing deaktiviert

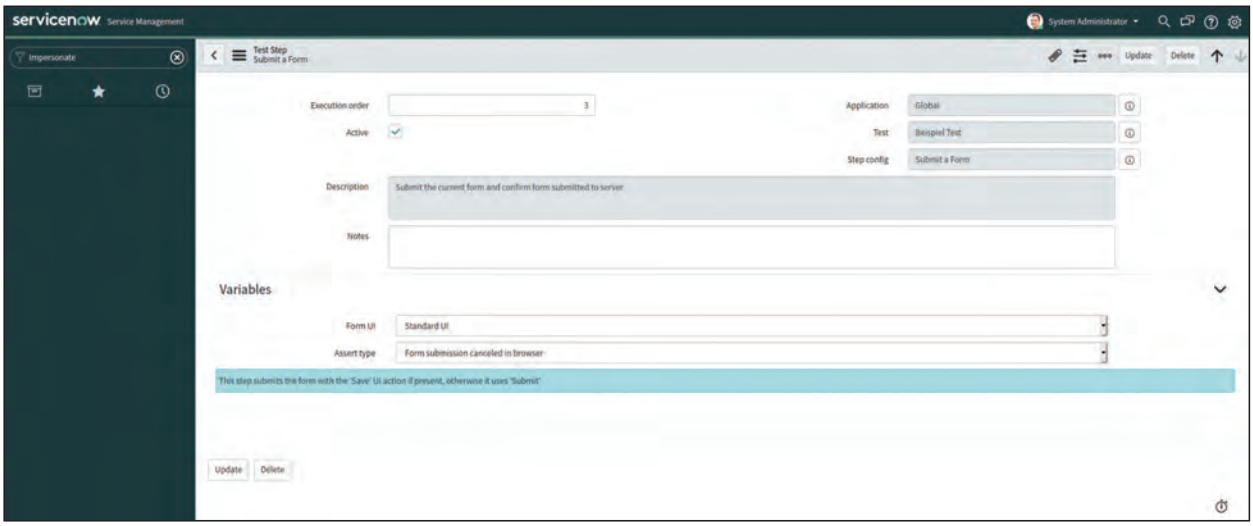


Abb. 4: Wir erwarten, dass der Submit fehlschlägt

ist. Klicken wir nun auf den Link in der Nachricht. Es öffnet sich die Seite mit den Einstellungen des ATF. Hier können wir nun die Ausführung von Tests direkt mit der ersten Checkbox aktivieren. Anschließend scrollen wir bis ans Ende der Seite und klicken auf den Save-Button. Mit Hilfe des Zurück-Buttons des Browsers können wir wieder auf die Seite zum Erstellen des Tests zurücknavigieren oder einfach noch einmal *sys_atf_test.list* in den Filternavigator eingeben und auf New klicken. Geben Sie bitte dem Test nun einen sprechenden Namen.

Wie eingangs erwähnt, ist der Test der Rahmen für die einzelnen Testschritte. Direkt nach dem Erstellen eines Tests ist es allerdings noch nicht möglich, Testschritte anzulegen. Dazu muss der neue Test zumindest einmal gespeichert werden. Das können wir mit dem Save-Button oder über den Save-Eintrag im „Hamburger-Menü" erledigen. In **Abbildung 2** ist nun unser neuer gespeicherter Test zu sehen. Hier ist es möglich, Testschritte hinzuzufügen.

Wir fügen nun einen Testschritt hinzu, der einen neuen User erzeugt, dem wir genau die Berechtigungen geben, die für den Test benötigt werden. Zum Hinzufügen eines Testschritts klicken wie auf den Add Test Step-Button. Es öffnet sich ein Wizard zum Erstellen von Testschritten. Der Create User-Testschritt ist der erste Eintrag in der Liste. Nachdem wir den Eintrag ausgewählt haben, klicken wir auf Next. Auf der folgenden Seite (**Abb. 3**) können wir dem User einen Namen geben und ihm Rollen und Gruppen zuweisen. Unser Test-User soll die Rolle *itil* bekommen. Wenn wir auf Submit klicken, wird der Testschritt angelegt und wir sind wieder in dem eigentlichen Test.

Nun soll der User ein Incident erzeugen. Dazu erstellen wir einen weiteren Testschritt, der uns ein neues Formular öffnet. Wir klicken wieder auf Add Test Step, dann wählen wir als Kategorie Form aus. In der Kategorie Form ist Open a New Form der erste Eintrag. Man kann sich über mehrere Wege zu den Test-Steps durchhangeln. Kennt man den genauen Namen des Step, kann man auch die Suchfunktion verwenden, um ihn zu finden. Nachdem wir Open a New Form ausgewählt haben, klicken wir auf Next. Danach wählen wir als Tabelle Incident aus und klicken auf Submit. Um das Formular, das wir eben erzeugt haben, zu speichern, benötigen wir einen Submit a Form-Testschritt. Wir fügen ihn dem Test hinzu. Um der Wahrheit die Ehre zu geben, glauben wir nicht wirklich daran, ein Incident erzeugen zu können, ohne auch nur einen Wert einzugeben. Wir könnten aber genau das einmal testen. Dazu setzen wir in den Testschritt Submit den Wert für den Asset type auf Form submission canceled in browser (**Abb. 4**).

Jetzt ist die Zeit gekommen, um unseren ersten kleinen Test einmal zu starten. Dazu klicken wir im Test auf den Button Run Test. Der sich öffnende Dialog informiert uns, dass ein neuer Test-Runner gestartet wird. Wir klicken hier abermals auf Run Test. Der Test-Runner startet den Test und wir können beobachten, was passiert. Die Erzeugung des Users geschieht im Hintergrund, sie macht sich aber mit einer kleinen Wartezeit bemerkbar. Danach sehen wir, wie ein neues Incident-Formular geöffnet wird. Beim genaueren Hinsehen fallen uns direkt zwei verpflichtend auszufüllende Felder auf. Beim Abschicken des Formulars erhalten wir eine Fehlermeldung, der Test an sich war aber erfolgreich (**Abb. 5**). Wir haben schließlich erwartet, dass wir kein leeres Formular abschicken können. Sollte es aber aufgrund eines Fehlers doch funktionieren, würde unser Test auf Rot gehen.

Dieser erste Test war wirklich einfach aufgebaut, aber er konnte gut zeigen, wie man vorgeht, wenn man mit dem ATF Tests erstellt. Sie können versuchen, den Test so zu erweitern, dass er ein Formular richtig befüllt und so ein Incident erzeugt. Sie müssen dazu nur einen Set Field Values-Schritt zwischen den Schritten 2 und 3 einfügen. In diesem Schritt müssen Sie dann die verpflichtend auszufüllenden Felder befüllen. Danach dürfen Sie allerdings nicht vergessen, den Assert in dem Submit-Schritt wieder umzustellen, denn sonst schlägt der Test fehl, obwohl im Grunde alles funktioniert hat.

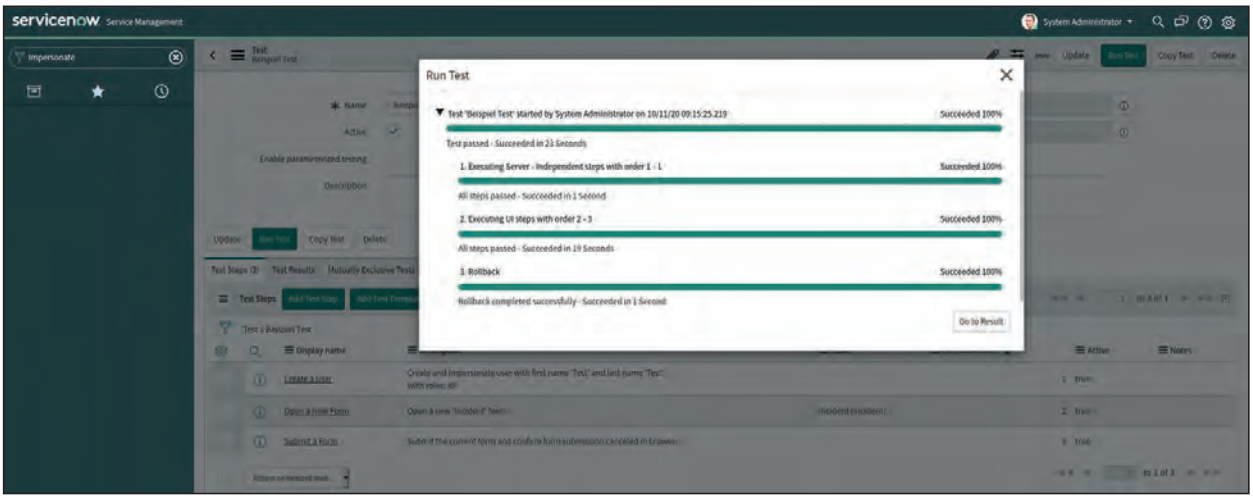


Abb. 5: Die Testergebnisse sehen gut aus

Falls Sie bei dem Versuch, den Test umzustellen, auf unerwartete Probleme stoßen, sollten Sie sich das Onlinetraining zum ATF ansehen [1]. Falls Sie lieber einfach nur ein Video zum Thema ansehen möchten, gibt es bei YouTube eine Menge über ATF zu finden. Dem Autor hat das Video unter [4] recht gut gefallen. Es wird zwar nicht die aktuelle Version von ServiceNow behandelt, doch das Video erklärt die Grundlagen sehr verständlich.

Tipps zum Schluss

Zu guter Letzt möchte ich Ihnen noch einige Tipps mit auf den Weg geben, die Ihnen beim alltäglichen Umgang mit dem ATF helfen können. Ein umfangreicher Test mit dem ATF kann Ihnen schnell Auskunft geben, ob das System als Ganzes noch so arbeitet, wie es sollte. Gerade vor und nach Deployments ist es wichtig, einen Test laufen lassen, um ein Vorher/Nachher-Bild zu erhalten. Falls Sie ein Upgrade auf der Plattform fahren, ist das ATF ein sehr wichtiges Werkzeug, um zu sehen, ob die eigene Applikation mit dem neuen Release der Plattform noch fehlerfrei arbeitet.

Wir sind hier nicht explizit darauf eingegangen, es sollte aber trotzdem erwähnt werden: Mit dem ATF kann man nicht nur ServiceNow-Applikationen testen, sondern auch die anderen Teile der Plattform. Dazu gehören zum Beispiel:

- Service Catalog
- Service Portal
- Rest Services
- serverseitige Scripts
- die User-Navigation der Plattform

Einige Test-Steps haben sogenannte Asserts. Mit ihnen kann man vorgeben, ob man erwartet, dass ein Test-Step fehlerfrei ausgeführt wird oder fehlschlägt. Somit sind also auch Negativtests möglich.

Die Tests erzeugen mit der Zeit recht viele Ergebnisse, weshalb es wichtig ist, ältere Testergebnisse hin und wieder zu löschen. Das kann automatisch oder auch manuell geschehen.

Fazit

Dieser Artikel hat eine kurze Einführung gegeben, was alles mit dem ATF möglich ist. Wir sind hier nur auf die wichtigsten Punkte eingegangen. Alle Aspekte des ATF zu beleuchten, würde den Rahmen sprengen. Zusammenfassend ist zu sagen, dass es mit dem ATF gut möglich ist, ServiceNow-Anwendungen zu testen. Dadurch, dass das Test-Framework Bestandteil der Plattform ist, sind Tests möglich, die ein externes Testtool nur schwer bis gar nicht durchführen kann. Das ATF befindet sich aktuell in der Aufbauphase. Mit jedem neuen Release kommen neue Funktionen hinzu und vorhandene werden verbessert. Wenn Sie die ServiceNow-Plattform in Ihrem Unternehmen einsetzen, sollten Sie sich auf jeden Fall auch mit dem ATF beschäftigen. Das ATF ist sehr leistungsfähig und erzeugt keine zusätzlichen Kosten, anders, als es bei einem externen Testtool der Fall wäre.

Martin Mohr erblickte im Zeitalter der Magnetringkernspeicher und Hebdrehwähler das Licht der Welt und hat somit die komplette Entwicklung der modernen Computertechnik live miterleben dürfen. Die Vorliebe für alles, was blinkt hat sich schon in seiner frühen Jugend entwickelt und wurde durch eine Ausbildung zum Elektroniker noch verstärkt. Nach dem Informatikstudium lag sein Beschäftigungsfeld überwiegend in der Entwicklung von Java-Applikationen. Mit dem Raspberry Pi ist die alte Liebe zur Elektronik wiedererwacht.

Links & Literatur

[1] Onlinekurs ATF: https://developer.servicenow.com/dev.do#!/learn/courses/paris/app_store_learnv2_atf_paris_automated_test_framework/app_store_learnv2_atf_paris_using_the_automated_test_framework/app_store_learnv2_atf_paris_automated_test_framework_objectives

[2] ServiceNow-Entwicklerhomepage: https://developer.servicenow.com

[3] Mohr, Martin: „Es wird Zeit für einen besseren Service“, Entwickler Magazin 6.2020

[4] YouTube-Video zum ATF: https://www.youtube.com/watch?v=LGj-ipn2ENY&t=30s

API CONFERENCE

HYBRID EDITION

UNTIL FEBRUARY 11

- ✓ Raspberry Pi or C64 Mini for free
- ✓ Group discount
- ✓ Save up to €404

April 12 – 14, 2021

Expo: April 13 – 14, 2021

The Hague or online

The Conference for Web APIs, API Design & Management

- API Management
- API Design
- API Development
- API Platforms & Business

API Conference | @api_conference #APICON | API Conference

apiconference.net

Presented by: e.academy jaxenter

Organizer: S&S Media Group

Tipps und Tricks für den Git-Alltag

Pimp my Git

In diesem Artikel werden Tipps und Tricks für Git anhand von Alltagssituationen vorgestellt. Die Autorin arbeitet mit Git am liebsten über einen Mix aus grafischen Werkzeugen und der Kommandozeile. Auch wenn im Artikel vieles mit der Nutzung der Kommandozeile vorgestellt wird, helfen einige Tipps auch, wenn die Entwicklerin ausschließlich mit einem grafischen Werkzeug arbeitet.

von Sandra Parsick

Im Beraterinnenalltag kommt es öfters vor, dass an verschiedenen Projekten gleichzeitig gearbeitet wird, die in verschiedenen Git-Management-Systemen liegen. Auch wenn die Entwicklerin keine Beraterin ist, kann sie mit dieser Situation konfrontiert werden, etwa wenn die Firma mitten in einer Migration der Git-Management-Systeme steckt und somit die Projekte auf verschiedenen solcher Systeme liegen, oder wenn die Entwicklerin Pull Requests auf GitHub erstellen möchte. Spätestens dann muss sie mit zwei Git-Identitäten und deren Anmeldedaten lokal umgehen können.

Als Erstes konfiguriert die Entwicklerin die Anmeldedaten für die Authentifizierung bei den verschiedenen Git-Management-Systemen. Natürlich kann sie die Authentifizierungen via Passwortmanager verwalten. Doch es bleibt umständlich, sich regelmäßig mit Benutzername und Passwort anzumelden. Alle gängigen Git-Management-Systeme bieten die Möglichkeit, sich über einen SSH-Schlüssel zu authentifizieren, und SSH kann so konfiguriert werden, dass es selbst erkennt, welchen Schlüssel es gerade benutzen soll.

Als Beispiel dienen zwei Git-Projekte, die auf zwei verschiedenen Git-Management-Systemen liegen. Die Git-Management-Systeme sind unter zwei verschiedenen Domänen erreichbar, *example1.com* und *example2.com*. Für jedes Git-Management-System generiert die Entwicklerin, wie in Listing 1 gezeigt, jeweils ein Public/Private-SSH-Schlüssel-Paar (*id_example1* und *id_example2*).

Der private Schlüssel liegt dann in der Datei *id_example1* und der öffentliche Schlüssel in der Datei *id_example1.pub*. Diese Prozedur wiederholt die Entwicklerin für *id_example2*. Diesen öffentlichen Schlüssel muss sie anschließend im jeweiligen Git-Management-System hinterlegen, damit die Authentifizierung via SSH funktionieren kann.

Als Nächstes trägt die Entwicklerin in der SSH-Konfiguration ein, bei welcher Domäne welcher Schlüssel benutzt werden soll. Dafür erstellt sie, wenn nicht vorhanden, eine Datei *config* unter *~/.ssh* und trägt darin ein, für welche Domäne welcher Schlüssel gilt (Listing 2).

Im Anschluss daran kann sie die Git-Repositorys klonen und pushen – ohne Eingabe der jeweiligen Anmeldedaten. Als Nächstes muss die Entwicklerin die Git-Identitäten konfigurieren. Mit Git-Identität sind dabei der Benutzername und die E-Mail-Adresse gemeint, die bei jedem Commit als Autor und Committer eingetragen werden. Der Benutzername und die E-Mail-Adresse können sich von Projekt zu Projekt unterscheiden. Eine herkömmliche Methode sieht folgendermaßen aus:

```
$ git clone git-repo
$ cd git-repo
```

Listing 1

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/sparsick/.ssh/id_rsa): /home/sparsick/.ssh/
                                                                  id_example1
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/sparsick/.ssh/id_example1.
Your public key has been saved in /home/sparsick/.ssh/id_example1.pub.
The key fingerprint is:
SHA256:9XVMIDlWX/6OvAzVmqKUBm8pmNNoiFkjJZ/+RERhp/
                                                  A sparsick@sparsick-ThinkPad-T460s
The keys randomart image is:
+---[RSA 2048]----+
| . +.. .oo.o|
| = o +. =.|
| . . E .. .. =|
| + o . . . o.|
| . = . S . . o|
| * + = o o o = |
| o o B o B o = .|
| + . = . + . |
| . . o |
+----[SHA256]-----+
```

```
$ git config user.name "Sandra Parsick"
$ git config user.email "sparsick@example1.com"
```

Diese Methode ist recht fehleranfällig, da die Anpassung der Userkonfiguration gerne vergessen wird. Dank des Git-Features Conditional Includes (verfügbar ab Git Version 2.13) lassen sich verschiedene Git-Identitäten pro Verzeichnispfad konfigurieren.

Die Idee von Conditional Includes ist, dass allgemein eine Default-Git-Konfiguration und seperate Git-Konfigurationen pro Verzeichnispfad definiert werden können. In diesen Git-Konfigurationen können der Autorname und die entsprechende E-Mail-Adresse hinterlegt werden. Das bedeutet, dass jedes Repository, das in einem der definierten Verzeichnispfade geklont wird, automatisch die Git-Identität erhält, die für dieses Verzeichnis definiert wurde. Dafür muss parallel zur Git-Konfigurationsdatei *.gitconfig* eine weitere Git-Konfigurationsdatei für jede Git-Identität angelegt werden.

```
$ touch ~/.gitconfig_example1
$ touch ~/.gitconfig_example2
```

Darin werden die jeweiligen Git-Konfigurationen eingetragen:

```
[user]
name = YourNameForExample1
email = name@example1.com
```

Danach muss in *.gitconfig* noch definiert werden, für welches Verzeichnis welche Git-Konfiguration angewandt werden soll (Listing 3).

Somit haben alle Repositorys, die im Verzeichnis *~/workspace_example1* geklont werden, automatisch die Git-Identität, die in der *.gitconfig_example1* konfiguriert wurde. Alle Repositorys, die im Verzeichnis *~/workspace_example2* geklont werden, haben entsprechend die Git-Identität aus *.gitconfig_example2*.

Git-Identität in mehreren Commits ändern

Dann ist es doch passiert: Die Entwicklerin hat mit einer nicht zum Projekt passenden Git-Identität gearbeitet und nun möchte sie nachträglich den Autor im Commit ändern. Wenn es nur den letzten Commit betrifft, ist sie mit dem Git-Befehl *commit --amend* schnell überschrieben:

```
$ git commit --amend --author "Sandra Parsick <sandra.parsick@example1.com>"
```

Wenn in mehreren Commits der Autorname überschrieben werden muss, hilft der Git-Befehl *rebase* weiter. Hier gibt es zwei Varianten, einmal einen interaktiven Modus, bei dem in jedem einzelnen Commit die Autorenänderungen bestätigt wird, oder einen Batchmodus, mit dem der Autor in mehreren Commits auf einmal geändert werden kann. Bei beiden Varianten muss die Entwicklerin die Commit-Nachricht aus der Commit-Historie heraussuchen (Listing 4).

In unserem Beispiel soll der Autor der letzten beiden Commits geändert werden. Daher muss die Entwicklerin den Hashcode des Commits auswählen, der vor den Commits liegt, die geändert werden sollen, und den Git-Befehl *rebase -i <Hashcode des Commits vor den Commits die geändert werden sollen>* übergeben (Listing 5).

Daraufhin öffnet sich der Editor mit einer Auswahl der Commits, die die Entwicklerin ändern kann. Alle Commits, die geändert werden sollen, markiert sie mit *edit*:

```
edit b2ade77 add unit test
edit 473f38a bug fix
# Rebase von 3d9866d..473f38a auf 3d9866d (2 Kommandos)
#
```

Dann ändert die Entwicklerin die Commits mit der Kombination der Git-Befehle *commit --amend* und *rebase --continue*. Der Befehl *commit --amend* ändert den Commit und *rebase --continue* geht zum nächsten zu ändernden Commit-Objekt. Dabei öffnet sich ein Editor, in dem sie die Commit Nachricht bestätigt:

```
$ git commit --amend --author "Sandra Parsick
                <sandra.parsick@example1.de>" && git rebase --continue
```

Listing 2

```
config
Host example1.com
HostName example1.com
User git
IdentityFile ~/.ssh/id_example1

Host example2.com
HostName example2.com
User git
IdentityFile ~/.ssh/id_example2
```

Listing 3

```
[user]
name = defaultName
email = default@email.com

[includeIf "gitdir:~/workspace_example1/"]
path = .gitconfig_example1

[includeIf "gitdir:~/workspace_example2/"]
path = .gitconfig_example2
```

Listing 4

```
$ git log
commit 473f38ada766e00704b811f24157479ac8570fc9 (HEAD -> master)
Author: Sandra Parsick <sparsick@example2.de>
Date:   Thu Nov 12 10:32:22 2020 +0100

    bug fix

commit b2ade77ab2d6af87a63885c7812eecc39f52e180
Author: Sandra Parsick <sparsick@example2.de>
Date:   Thu Nov 12 10:32:13 2020 +0100

    add unit test

commit 3d9866dc9266a68d9b1e3e8108328c0dac2960c7
Author:  Sandra Parsick <sparsick@example1.de>
Date:   Thu Nov 12 10:31:39 2020 +0100

    add integration tests
```

Abb. 1: Die Webseite gitignore.io

Das wiederholt die Entwicklerin so lange, bis alle markierten Commit-Objekte geändert wurden. Das Ganze kann sie aber mit Hilfe des Batchmodus auch in einem Rutsch durchführen. Dafür wählt sie wie im interaktiven Modus die letzte Commit-Nachricht im Log, die nicht geändert werden soll, und markiert die Commits, die geändert werden sollen, im Editor mit *edit*. Statt jetzt alle Commits einzeln bestätigen zu müssen, werden die jeweiligen Commit-Nachrichten in einem Batch geändert.

```
$ git commit --amend --author "Sandra Parsick
      <sandra.parsick@example1.de>" --no-edit && git rebase --continue
```

Listing 5

```
$ git rebase -i 3d9866dc9266a68d9b1e3e8108328c0dac2960c7
pick b2ade77 add unit test
pick 473f38a bug fix

# Rebase von 3d9866d..473f38a auf 3d9866d (2 Kommandos)
#
# Befehle:
# p, pick <Commit> = Commit verwenden
# r, reword <Commit> = Commit verwenden, aber Commit-Beschreibung bearbeiten
# e, edit <Commit> = Commit verwenden, aber zum Nachbessern anhalten
# s, squash <Commit> = Commit verwenden, aber mit vorherigem Commit vereinen
# f, fixup <Commit> = wie "squash", aber diese Commit-Beschreibung verwerfen
# x, exec <Commit> = Befehl (Rest der Zeile) mittels Shell ausführen
# b, break = hier anhalten (Rebase später mit 'git rebase --continue' fortsetzen)
# d, drop <Commit> = Commit entfernen
# l, label <Label> = aktuellen HEAD mit Label versehen
# t, reset <Label> = HEAD zu einem Label umsetzen
# m, merge [-C <Commit> | -c <Commit>] <Label> [# <eineZeile>]
# .      Merge-Commit mit der originalen Merge-Commit-Beschreibung erstellen
# .      (oder die eine Zeile, wenn keine originale Merge-Commit-Beschreibung
# .      spezifiziert ist). Benutzen Sie -c <Commit> zum Bearbeiten der
# .      Commit-Beschreibung.
#
# Diese Zeilen können umsortiert werden; Sie werden von oben nach unten
# ausgeführt.
#
# Wenn Sie hier eine Zeile entfernen, wird DIESER COMMIT VERLOREN GEHEN.
#
# Wenn Sie jedoch alles löschen, wird der Rebase abgebrochen.
#
# Leere Commits sind auskommentiert.
```

Wichtig bei dieser Technik ist, dass der Entwicklerin bewusst ist, dass sie mit dem Git-Befehl *rebase* die Git-Historie verändert. Das bedeutet, sie sollte diesen Befehl nur auf Commits ausführen, die nicht veröffentlicht (gepusht) und nur in ihrer lokalen Git-Historie vorhanden sind.

Schnelles Erzeugen von .gitignore-Dateien

Wenn die Entwicklerin ein neues Git-Repository erstellt, verbringt sie viel Zeit damit, eine vollständige *.gitignore*-Datei zu erzeugen. Glücklicherweise gibt es mehrere Werkzeuge, die ihr bei deren Erstellung helfen können. Zum Beispiel eine Webseite, die auch über die Kommandozeile benutzbar ist, sowie ein Plug-in für die IDE IntelliJ IDEA.

Die Webseite https://gitignore.io (**Abb. 1**) listet für verschiedene Programmiersprachen, Werkzeuge, IDEs etc. allgemeine *ignore*-Muster auf. Die Benutzung ist einfach gehalten: Die Entwicklerin füllt die Suchmaske mit dem Namen der Programmiersprache, Werkzeuge oder IDE aus, die sie im Git-Repository benutzen möchte, und die Webseite generiert den Inhalt für die *.gitignore*-Datei.

Wenn die Entwicklerin die Webseite auf der Kommandozeile benutzen will, dann erstellt sie sich eine Umgebungsfunktion für ihre Shell. Bei der Shell Zsh sieht das folgendermaßen aus:

```
echo "function gi() { curl -sLw "\n" https://www.toptal.com/developers/
      gitignore/api/\$@ ;}" >> ~/.zshrc && source ~/.zshrc
```

Wie diese Funktion in anderen Shells erstellt wird, ist in der gitignore.io-Dokumentation beschrieben [2]. Jetzt kann die Entwicklerin die Webseite auf der Kommandozeile benutzen (Listing 6).

- *$ gi java,maven*: generiert den Inhalt für Java und Maven
- *$ gi list*: listet alle vorhandene Templates für die *.gitignore*-Datei auf
- *$ gi java,maven >> .gitignore*: generierter Inhalt wird in eine .gitignore-Datei

Wenn die Entwicklerin nicht mit der Kommandozeile arbeiten will, sondern in ihrer IntelliJ IDEA bleiben möchte, dann kann sie die Inhalt der *.gitignore*-Datei mit dem Plug-in .ignore (**Abb. 2** und **3**) generieren. Dieses Plug-in generiert *ignore*-Dateien für verschiedene Werkzeuge wie Docker oder Mercurial.

Hilfe, meine .gitignore-Datei ist riesengroß!

Wenn die Entwicklerin das Glück hat, in einem Projekt zu arbeiten, bei dem Werkzeugfreiheit herrscht, kommt es vor, dass die *.gitignore*-Dateien in den Projekten unübersichtlich werden. Das liegt meist daran, dass alle Entwicklerinnen die Dateien ihrer Lieblingswerkzeuge in die *.gitignore* jedes Git-Repositorys des Projekts eintragen. Abhilfe kann geschaffen werden, wenn jede Entwicklerin bei sich lokal eine globale *.gitignore*-Datei anlegt, die für alle Git-Repositorys gilt, und dort alle Da-

teien einträgt, die ihre Lieblingswerkzeuge bei der Arbeit erzeugen, die nicht mit eingecheckt werden sollen:

```
$ cd ~
$ gi intellij+iml >> .gitignore_global
$ git config --global core.excludesfile ~/.gitignore_global
```

Wichtig ist hier, dass die Entwicklerin die globale *.gitignore*-Datei in die globale Git-Konfiguration einträgt.

Arbeiten auf mehreren Branches gleichzeitig

Muss die Entwicklerin an mehreren Branches arbeiten, klont sie gerne das Git-Repository mehrmals in verschiedene Verzeichnisse und checkt in jedem Verzeichnis den jeweiligen Branch aus. Nachteil dieser Methode ist, dass jeder Checkout eine eigene Kopie der Git-Historie hat. Dieser Nachteil kommt zum Tragen, wenn zwischen den ausgecheckten Branches lokal gemergt werden muss.

Eine umständliche Lösung wäre, dass die Entwicklerin einen Branch im Remote-Repository veröffentlicht und sich die Änderungen über das Remote-Repository in das lokale Repository holt. Diesen Nachteil kann sie mit dem Git-Befehl *worktree* umgehen. Mit diesem Befehl kann sie einen Branch in ein separates Verzeichnis auschecken und dennoch dieselbe Git-Historie benutzen:

```
$ git worktree add ../worktree-sample
$ git worktree add ../worktree-further-sample existingBranch
```

- Zeile 1: Ein neuer Branch mit dem Namen *worktree-sample* wird erzeugt und im Verzeichnis *../worktree-sample* ausgecheckt
- Zeile 2: Der existierende Branch mit dem Namen *existingBranch* wird im Verzeichnis *../worktree-further-sample* ausgecheckt

In dem so erzeugten Verzeichnis kann die Entwicklerin gewohnt und ganz wie in einem normalen Branch arbeiten. Der Git-Befehl *merge* zwischen den Branches funktioniert wie bei „normalen“ Branches, nur dass die Änderungen gleich im *worktree*-Verzeichnis zu sehen sind. Mit dem Git-Befehl *worktree remove* kann die Entwicklerin das vorhin erzeugte Verzeichnis löschen, ohne den Branch zu löschen:

```
$ git worktree remove ../worktree-sample
```

Hat die Entwicklerin die Übersicht über die vorhandenen Worktrees verloren, kann sie sich mit dem Git-Befehl *worktree list* wieder einen Überblick verschaffen:

Listing 6

```
$ gi java,maven

# Created by https://www.toptal.com/developers/gitignore/api/java,maven
# Edit at https://www.toptal.com/developers/gitignore?templates=java,maven

### Java ###
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.nar
*.ear
*.zip
*.tar.gz
*.rar

# virtual machine crash logs, see http://www.java.com/en/download/help/
# error_hotspot.xml
hs_err_pid*

### Maven ###
target/
pom.xml.tag
pom.xml.releaseBackup
pom.xml.versionsBackup
pom.xml.next
release.properties
dependency-reduced-pom.xml
buildNumber.properties
.mvn/timing.properties
# https://github.com/takari/maven-wrapper#usage-without-binary-jar
.mvn/wrapper/maven-wrapper.jar

# End of https://www.toptal.com/developers/gitignore/api/java,maven

$ gi list
1c,1c-bitrix,a-frame,actionscript,ada
adobe,advancedinstaller,adventuregamestudio,agda,al
alteraquartusii,altium,amplify,android,androidstudio
angular,anjuta,ansible,apachecordova,apachehadoop
appbuilder,appceleratortitanium,appcode,appcode+all,appcode+iml
appengine,aptanastudio,arcanist,archive,archives
archlinuxpackages,aspnetcore,assembler,ate,atmelstudio
ats,audio,automationstudio,autotools,autotools+strict
awr,backup,ballerina,basercms,basic
batch,bazaar,bazel,bitrise,bitrix
bittorrent,blackbox,bloop,bluej,bookdown
bower,bricxcc,buck,c,c++
....

$ gi java,maven >> .gitignore
```

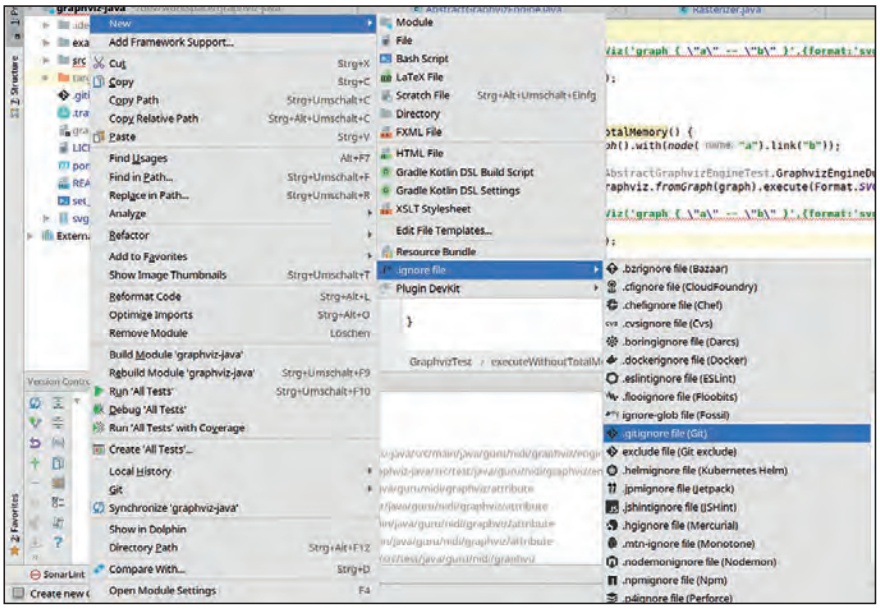


Abb. 2: Neues Dateimenü mit ignore-Plug-in

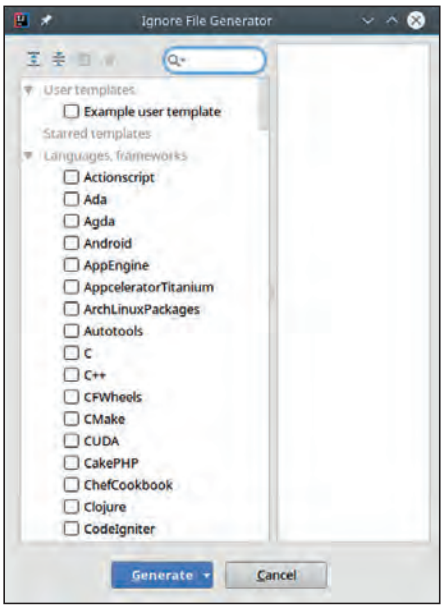


Abb. 3: Templateauswahl für jeweilige Programmiersprache, Werkzeug etc. im ignore-Plug-in

```
$ git worktree list
/home/sparsick/dev/workspace/git-sample  6d36f94 [master]
/home/sparsick/dev/workspace/git-sample1  6d36f94 [git-sample1]
```

Merge-Konflikte trotz Kommandozeile einfach lösen

Auch wenn das Arbeiten mit der Kommandozeile toll ist, gibt es Situationen, in denen die Entwicklerin mit

Listing 7

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        araxis
        kdiff3
        meld

The following tools are valid, but not currently available:
        bc
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        emerge
        gvimdiff
        gvimdiff2
        gvimdiff3
        opendiff
        p4merge
        tkdiff
        tortoisemerge
        vimdiff
        vimdiff2
        vimdiff3
        winmerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

Listing 8

```
$ git merge branch1

Auto-merging test
CONFLICT (content): Merge conflict in test
Automatic merge failed; fix conflicts and then commit the result.
$ git mergetool
Merging:
test

Normal merge conflict for 'test':
{local}: modified file
{remote}: modified file
```

Listing 9

```
$ git status

On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:

modified:   test

Untracked files:
(use "git add <file>..." to include in what will be committed)

test.orig
$ git commit
```

GUI-Unterstützung schneller ist. Das Lösen von Merge-Konflikten wäre so eine Situation. Git bietet von Haus aus den Befehl *mergetool*, mit dem die Entwicklerin aus der Kommandozeile heraus ein GUI-Werkzeug starten kann, das ihr hilft, die Merge-Konflikte zu lösen. Bevor das GUI-Werkzeug allerdings zum Einsatz kommen kann, muss die Entwicklerin in Git konfigurieren, welches Werkzeug benutzt werden soll, wenn der Befehl *mergetool* aufgerufen wird. Zuerst überprüft sie dazu, welches GUI-Werkzeug auf der Maschine verfügbar ist und zugleich von Git unterstützt wird (Listing 7).

In unserem Beispiel wählt die Entwicklerin das Werkzeug Meld [3] aus. Sie möchte es überdies für alle Git-Repositorys als *mergetool* konfigurieren:

```
$ git config --global merge.tool meld
```

Wenn das ausgewählte Werkzeug im *PATH* bereits konfiguriert ist, reicht diese Konfiguration. Ansonsten muss die Entwicklerin den Pfad zum Werkzeug in Git konfigurieren:

```
git config --global mergetool.meld.path /usr/bin/meld
```

Dabei muss sie beachten, dass der mittlere Teil des Konfigurationsschlüssels (im Beispiel *meld*) abhängig von dem gewählten Werkzeug ist. Wenn die Entwicklerin zum Beispiel *vimdiff* als Werkzeug gewählt hätte, dann würde der Konfigurationsschlüssel *mergetool.vimdiff.path* lauten. Möchte sie jetzt nach einem Merge einen Merge-Konflikt via GUI-Werkzeug lösen, ruft sie den Befehl *mergetool* auf (Listing 8).

Sie löst daraufhin die Konflikte im GUI-Werkzeug, schließt es und ruft schließlich den Befehl *commit* auf, um das Merging zu beenden (Listing 9).

In der Defaultkonfiguration von *mergetool* erzeugt Git immer ein Back-up der gemergten Datei. Die Entwicklerin kann dieses Verhalten wie folgt umstellen, sodass Git nach einem erfolgreichen Merge das Back-up automatisch löscht.

```
$ git config --global mergetool.keepBackup false
```

Fazit

Die in diesem Artikel gezeigten Tipps und Tricks kratzen nur an der Oberfläche der Kniffe, mit denen sich die Entwicklerin den Alltag mit Git vereinfachen kann. Die effektivste Methode für das Erlernen neuer Git-Kniffe ist es, den Kolleginnen zuzuschauen, wie sie Git benutzen.

Sandra Parsick ist als freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld tätig. Seit 2008 beschäftigt sie sich mit agiler Softwareentwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich Java-Enterprise-Anwendungen, agile Methoden, Software Craftsmanship und in der Automatisierung von Softwareentwicklungsprozessen. Darüber hinaus schreibt sie gerne Artikel und spricht auf Konferenzen. In ihrer Freizeit engagiert sich Sandra Parsick in der Softwerkskammer Ruhrgebiet, einer Regionalgruppe der Software Craftmanship Community im deutschsprachigen Raum. Seit 2019 ist sie Mitglied im Oracle-Groundbreaker-Ambassador-Programm.

mail@sandra-parsick.de **@SandraParsick**

www.sandra-parsick.de

Links & Literatur

[1] Webseite gitignore.io: https://gitignore.io

[2] Dokumentation der Installation für die Kommandozeile: https://docs.gitignore.io/install/command-line

[3] Meld: https://meldmerge.org

Block Cipher erklärt – Teil 1

Symmetrische Verschlüsselung in Ruby

Neben Stream Ciphern [1], [2] bietet die Standard-Library von Ruby auch Block Cipher, um symmetrische Verschlüsselung betreiben zu können. Allen voran wird natürlich auch der am weitesten verbreitete Block Cipher, AES [3], unterstützt. Zwar ist der Erfolg von AES nach knapp 20 Jahren ungebrochen und es gibt keinerlei Anzeichen, dass sich das in näherer Zukunft ändern könnte, doch bedeutet das Verschlüsseln mit AES nicht automatisch Sicherheit. Es kommt ganz darauf an, wie man mit AES verschlüsselt. Im Gegensatz zu Stream Ciphern bieten Block Cipher verschiedene Betriebsmodi an, die sich hinsichtlich ihrer Sicherheit teilweise gehörig unterscheiden.

von Martin Boßlet

So wie auch schon bei den Stream Ciphern möchte ich Ihnen anhand konkreter Beispiele zeigen, was genau ein Block Cipher ist, was sichere Verschlüsselung mit einem Block Cipher auszeichnet und auf was es zu achten gilt. Dazu werden wir uns zunächst unseren eigenen Block Cipher basteln, der uns als Anschauungsobjekt dienen wird, um mittels diesem die einzelnen Evolutionsstufen zu durchlaufen, bevor wir uns dann im nächsten Teil vor dem Hintergrund der gewonnenen Erkenntnisse konkret dem Angebot in der Standard-Library von Ruby widmen werden.

Was ist ein Block Cipher?

Am besten versteht man die Funktionsweise von Block Ciphern, wenn man sie den Stream Ciphern gegenüberstellt. Bei einem Stream Cipher hat man die Wahl, in welchen Portionen man den Plaintext verarbeiten möchte. Man kann alles auf einen Rutsch verarbeiten oder den Plaintext als einen konstanten Strom von Daten auffassen, den man Byte für Byte oder sogar Bit für Bit verarbeiten kann.

Artikelserie

Teil 1: Symmetrische Verschlüsselung in Ruby
Teil 2: Verschlüsselung in der Standard-Library

Bildlich betrachtet legt man über den Plaintext-Strom einen gleichlangen Strom aus scheinbar zufällig generierten Daten, die von der Generatorfunktion, dem Herzstück eines Stream Ciphers, generiert werden. Dabei werden die ursprünglichen Daten des Plaintexts mit denen der Generatorfunktion mittels *XOR* verknüpft (**Abb. 1**).

Man kann sich den Strom der Generatorfunktion als Schleier vorstellen, der sich mit dem Plaintext so verbindet, dass dem Außenstehenden im Nachhinein nicht mehr ersichtlich ist, was ursprünglich Schleier und was Plaintext war. Für den Uneingeweihten sieht das Resultat völlig willkürlich und nicht mehr von echten zufälligen Daten unterscheidbar aus. Um einen Ciphertext, der mit einem Stream Cipher verschlüsselt wurde, wieder zu entschlüsseln, muss man nun den exakt gleichen Strom aus scheinbar zufälligen Daten erneut generieren und ihn abermals mittels *XOR* mit dem Ciphertext verknüpfen. Auch hier kann man das auf einmal oder Byte für Byte machen. Wichtig ist, dass es exakt die gleichen Daten sind, die man mit der Generatorfunktion des Stream Ciphers generiert, und dass man sie an der exakt gleichen Position mit den Bytes des Ciphertexts mit *XOR* verknüpft. Dann kommen die Eigenschaften der *XOR*-Funktion zum Tragen und die scheinbar zufälligen Daten der Generatorfunktion heben sich auf. Was übrig bleibt, ist der ursprüngliche Plaintext. Der Schleier wird also sozusagen herausgekürzt und man hat am Ende wieder die unverschlüsselten Originaldaten.

Betrachten wir das Bild der Stream-Cipher-Konstruktion genauer, fällt auf, dass wir hier gar nicht wirklich von einem Ver- und Entschlüsseln reden können, da in beiden Fällen eine identische Operation durchgeführt wird: *XOR* mit dem Strom der Daten der Generatorfunktion. Wir können festhalten, dass Stream Cipher also im Grunde genommen nur aus einer einzigen Operation bestehen. Es macht in der Praxis keinen Unterschied, ob ich ver- oder entschlüssele, in beiden Fällen wird das Gleiche getan, nur die Daten, mit denen man das XOR durchführt, ändern sich. Bei Block Ciphern (**Abb. 2**) ist das anders. Sie bestehen aus zwei unterscheidbaren Funktionen, das Entschlüsseln unterscheidet sich vom Verschlüsseln. Dabei ist das Entschlüsseln die Umkehrfunktion des Verschlüsselns und ein valider Block Cipher muss unabhängig vom zugrunde liegenden Plaintext *m* folgende Invariante erfüllen:

$$D\_k(E\_k(m)) = m$$

wobei mit *E_k* die Verschlüsselung mit einem Schlüssel *k* und *D_k* die Entschlüsselungsfunktion mit demselben Schlüssel *k* bezeichnet sei. Es muss also unabhängig vom Plaintext immer gelten, dass die Verschlüsselung die Entschlüsselung aufhebt, sodass man nach dem Entschlüsseln des Ciphertexts stets den ursprünglichen Plaintext zurückerhält. Darüber hinaus arbeitet ein Block Cipher, wie es sein Name schon andeutet, nicht auf Daten beliebiger Länge, sondern man verarbeitet den Plaintext häppchenweise in Blöcken gleicher Länge. AES und viele weitere neuere Block Cipher arbeiten mit einer fixen Blockgröße von 16 Bytes, ältere Algorithmen wie etwa DES oder Blowfish, aber auch der von uns später zu implementierende TEA, arbeiteten mit einer Blockgröße von 8 Byte.

Der Plaintext wird also im Beispiel von AES zunächst in Häppchen à 16 Byte unterteilt und diese Blöcke werden dann individuell verarbeitet und am Ende wieder zusammengeführt. Je nachdem, ob man gerade ver- oder entschlüsselt, wird dabei entsprechend die Verschlüsselungs- oder die Entschlüsselungskomponente von AES auf die Blöcke angewandt. Im Unterschied zu Stream Ciphern wird der ursprüngliche Plaintext also nicht via *XOR* „verschleiert“, sondern die Plaintext-Blöcke werden von der Verschlüsselungsfunktion transformiert, entsprechend die Ciphertext-Blöcke von der Entschlüsselungsfunktion. Tatsächlich sind die Ent- und Verschlüsselungsfunktionen Permutationen, die nach außen hin völlig willkürlich erscheinen, sogenannte Pseudozufallspermutationen [4].

Um nun Plaintexte beliebiger Länge mit zum Beispiel AES zu verschlüsseln, wäre es also naheliegend, sie in Blöcke zu je 16 Byte aufzuteilen und die Blöcke jeweils mit der AES-Verschlüsselungsfunktion zu verschlüsseln. Dann werden die verschlüsselten Blöcke wie an einer Kette aufgereiht zusammenführt und dieses Ergebnis als finaler Ciphertext betrachtet. Zum Entschlüsseln würde man analog vorgehen, mit dem Unterschied, dass man statt der Verschlüsselungsfunktion die AES-

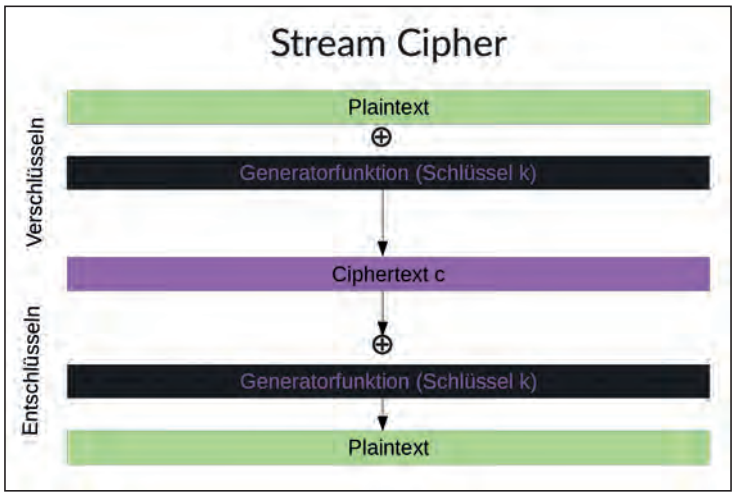


Abb. 1: Stream Cipher

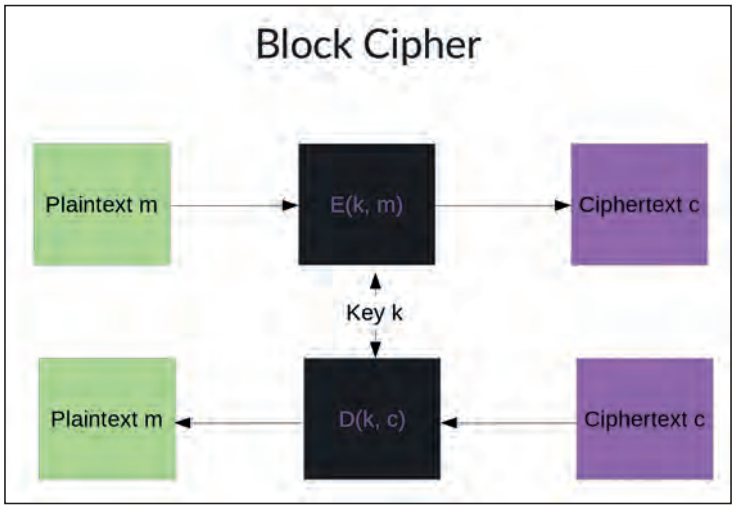


Abb. 2: Block Cipher

Entschlüsselungsfunktion verwendet, um die einzelnen Plaintext-Blöcke wieder zu erhalten. Diese würde man aneinanderreihen, um den ursprünglichen Plaintext zurück zu erhalten. Leider ist es nicht ganz so einfach, denn wie wir später sehen werden, ist dieser so pragmatische und naheliegende Ansatz alles andere als sicher.

Padding

Bevor wir uns Fragen nach der Sicherheit unserer Konstruktion stellen, müssen wir ein anderes Problem lösen, das unsere Konstruktion aufwirft: Was tun, wenn mein

Ausführen der Codebeispiele

Sollten Sie Ruby noch nicht installiert haben, können Sie dies über ruby-lang.org oder mit einem Tool wie rvm.io oder rbenv (https://github.com/rbenv/rbenv) vornehmen.

Damit Sie auch die neueren Algorithmen nutzen können, sollten Sie auf Ihrem System OpenSSL in einer Version >= 1.1.0 installiert haben.

Danach können Sie die Codebeispiele einfach ausführen, indem Sie das Repository unter https://github.com/emboss/block-ciphers klonen, in das Verzeichnis *code* wechseln und dort die Beispiele mit *ruby <beispiel.rb>* ausführen.

Plaintext eben nicht genau in Häppchen zu je 16 Byte teilbar ist? Das wird bei nämlich eher die Ausnahme sein. Zum Glück gibt es darauf eine einfache Antwort.

Zuerst einmal ist festzustellen, dass sich das Problem nur im allerletzten Block ergibt. Alle Vorgängerblocks kann man passgenau auf 16 Byte zurechtschneiden, nur zum Schluss bleibt potenziell ein Rest übrig. Diesen füllt man nun einfach so lange mit Bytes auf (englisch: to pad), bis es passt und 16 Bytes vollständig aufgefüllt sind. Das führt unmittelbar zur nächsten Frage: Wie erkennen die Empfänger einer solchen Nachricht, wo der eigentliche Plaintext aufhört und wo das Füllmaterial anfängt? Ganz so einfach, wie es auf den ersten Blick erscheinen mag, ist das Problem also doch nicht.

Um den Plaintext klar vom Padding unterscheiden zu können, wird ein Trick angewandt: Man kodiert im Padding selbst, aus wie vielen Bytes es besteht, sodass man nach dem Entschlüsseln genau weiß, wie viele Bytes man wieder abzuschneiden hat. Das De-facto-Standard-Verfahren für Padding ist das sogenannte PKCS#5- oder PKCS#7-Padding (zwei Namen für das gleiche Schema) [5]. Man fügt am Ende des Plaintexts gerade so viele Bytes ein, wie bis zu den vollen 16 Stück noch fehlen, und verwendet als deren Wert einfach die ursprüngliche Anzahl, die fehlte, wie hier im Beispiel gezeigt wird.

- 1 Byte fehlt: *<plaintext> 01*
- 2 Bytes fehlen: *<plaintext> 02 02*
- 3 Bytes fehlen: *<plaintext> 03 03 03*
- ...
- 10 Bytes fehlen: *<plaintext> 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A*

Listing 1

```
module Padding
  class PKCS5

    attr_reader :block_size

    def initialize(block_size)
      @block_size = block_size
    end

    def apply_to(plain_text)
      num_padding = block_size - (plain_text.bytesize % block_size)
      padding = num_padding.chr * num_padding
      "#{plain_text}#{padding}"
    end

    def strip_from(decrypted)
      num_padding = decrypted.byteslice(-1).ord
      decrypted.byteslice(0, decrypted.bytesize - num_padding)
    end

  end
end
```

- 14 Bytes fehlen: *<plaintext> 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E*
- 15 Bytes fehlen: *<plaintext> 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F*

Um das Padding nach dem Entschlüsseln wieder herauszulösen, schaut man auf das letzte Byte und schneidet gemäß der Anzahl, die man angezeigt bekommt, entsprechend viele Bytes vom Ende ausgehend wieder ab.

Ist das Problem also gelöst? Nur scheinbar. Was ist, falls man einer der Glücklichen war, und der ursprüngliche Plaintext bereits genau einem Vielfachen von 16 Bytes entsprach? Und also eigentlich gar kein Padding nötig war? Hat das letzte Byte des Plaintexts einen Wert > 0F, könnte man argumentieren, dass es folglich kein Padding sein kann und man als Entschlüsselnder kein Padding entfernen muss. Was aber, wenn das letzte Byte des Plaintexts einen Wert zwischen *01* und *0F* hat? Dann kann ein Empfänger nicht mehr klar unterscheiden, ob es sich in diesem Fall tatsächlich um Plaintext handelt oder vielleicht doch eher um Padding. Um also diese Doppeldeutigkeit zu vermeiden, muss man wohl oder übel in den sauren Apfel beißen. In den Fällen, in denen es gar keinen Rest gibt, muss man einen vollen Block, der nur aus Padding besteht, ans Ende hängen:

- 0 Bytes fehlen: *<plaintext> 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10*

Nur so kann die Doppeldeutigkeit vermieden werden. Wir können dieses Padding mit recht einfachen Mitteln in Ruby implementieren (Listing 1) [6].

Die Implementierung arbeitet mit *String#bytesize* statt *String#size* und *String#byteslice* statt *String#slice m*, um eventuellen Problemen mit dem String-Encoding aus dem Weg zu gehen. Bei der Verschlüsselung wird immer nur die rohe Byteform der Daten verwendet.

TEA: Tiny Encryption Algorithm

Als Beispiel für einen einfach zu implementierenden Block Cipher verwenden wir TEA [7]. Es gibt ein paar Schwächen im Design [8], was zu weiteren Versionen führte (XTEA, XXTEA), dennoch ist der Algorithmus trotz seiner Einfachheit erstaunlich robust [9]. Individualismus wird bei Kryptographie aber häufig bestraft, daher sollte man im produktiven Einsatz lieber mit dem Strom schwimmen und auf den Platzhirsch AES setzen. Nichtsdestotrotz ist der Algorithmus aber ideal für unsere Anschauungszwecke, da er sich in wenigen Zeilen Ruby-Code implementieren lässt und sich daher gut eignet, Block Cipher ganz allgemein zu untersuchen (Listing 2) [10].

Statt direkt Bytes zu verarbeiten, wurde TEA spezifiziert als Algorithmus für ganzzahlige 32-Bit-Integer (vier Byte) ohne Vorzeichen. Dabei wird der sechzehn Byte lange Schlüssel in vier Integer *k0* bis *k3* aufgeteilt und ein Plaintext-Block bestehend aus acht Byte wird in zwei 32-Bit-Integer *v0* und *v1* aufgeteilt. Normalerweise

SOFTWARE ARCHITECTURE SUMMIT

Bis zum 04.02.2021 bis zu 100 € sparen!

10. – 12. März 2021 | Remote

Das große Online-Trainingsevent für Softwarearchitektur

Der Summit ist das unverzichtbare Trainingsevent für alle Softwarearchitekt*innen, Senior-Entwickler*innen und IT-Projektleiter*innen. Sie erleben eine einzigartige Kombination topaktueller Themen mit einer Auswahl hochkarätiger Softwarearchitekturprofis, die in ihren Workshops tief ins Detail gehen. Erhalten Sie wertvolles Praxiswissen für die tägliche Arbeit!

- **Topaktuelle Power Workshops**
- **Hochkarätig besetzte Keynotes**
- **Hoher Praxisbezug durch Livedemos**
- **Interaktion und virtuelles Networking**
- **Maximaler Wissenstransfer in Kleingruppen**


Matthias Bohlen


Lisa Moritz


Stephan Pirnbaum


Michael Plöd


Golo Roden


Henning Schwentner


Johannes Seitz


Aminata Sidibe


Michael Sperber


Stefan Tilkov


Dr. Larysa Visengeriyeva


Eberhard Wolff

software-architecture-summit.de

Software Architecture Summit @SoftwArchSummit #SoftwareArchitectureSummit

Präsentiert von: entwickler akademie | Supported by: INNOQ | Powered by: e.academy | Veranstalter: S&S Media Group

ist für die Umwandlung roher Bytes zu Integer-Werten jede Menge Bitschubserei mit Bit-Shifts und binärem OR notwendig, aber Ruby bietet hierzu komfortabler Weise *String#unpack* [11] bzw. *Array#pack* [12] an, die einem die Arbeit abnehmen und vor allem viel effizienter arbeiten, da sie in nativem Code implementiert sind. Die für uns notwendigen Konversionen sind im Modul *UInt32* (angelehnt an den C-Integer-Typ *uint32_t*) festgehalten (Listing 3) [13].

Wer den TEA-Code mit dem Originalalgorithmus z. B. auf der Wikipedia-Seite vergleicht, wird feststellen, dass er beinahe eine 1:1-Kopie der C-Referenzimplementierung darstellt, es gibt nur einen kleinen, aber wichtigen Unterschied: Nach der Verarbeitung von *v0* bzw. *v1* folgt in unserer Implementierung jeweils

```
v0 = v0 & UInt32::MASK
```

und

```
v1 = v1 & UInt32::MASK
```

Der Grund hierfür ist die Tatsache, dass wir in Ruby keine ganzzahligen Integer-Typen mit fester Größe haben. Stattdessen erweitert Ruby dynamisch den Wertebereich ganzzahliger Typen, sodass es niemals zu einem Überlauf, wie er typisch für ganzzahlige Integer-Typen in C ist, kommen kann. Sie können in Ruby beliebig oft zu einer Ganzzahl etwas dazu addieren und Ruby wird das Ergebnis immer korrekt verwalten, ohne dass es jemals zu einem Überlauf käme. Im Gegensatz dazu würde in C, wenn Sie zu einem *uint8_t* mit dem Wert *255* noch *1* hinzuaddieren, die Zahl wieder zu *0* werden, da der Wertebereich fix ist und nur die Werte *0-255* annehmen kann. In den allermeisten Fällen führt dieses Verhalten zu Problemen, die dann oft nur sehr schwer zu debuggen sind. Bei Ruby hat man sich daher dafür entschieden, solche Fälle zu vermeiden, indem man einfach bei Bedarf den Speicherplatz, den eine Ganzzahl fassen kann, dynamisch erweitert. Allerdings setzt man gerade in der Kryptographie häufig auf diesen Effekt des Überlaufs bei Ganzzahlen, da er prädestiniert für die Verschlüsselung ist, weil er eine Prise Willkür und Unberechenbarkeit in die Berechnungen einfließen lässt, die der Verschlüsselung im Allgemeinen zuträglich sind.

Um diesen Effekt des Überlaufs in Ruby zu simulieren, muss man also nach jeder Addition oder Subtraktion für das Ergebnis noch zusätzlich etwas tun. Man lässt dabei die für die Berechnung relevanten Bits unberührt stehen und schneidet strikt die überflüssigen Bits ab, die außerhalb des relevanten Wertebereichs verändert wurden. Wieso bildet das einen Überlauf korrekt ab? Wenn wir betrachten, wie Addition und Subtraktion auf binärer Ebene funktionieren [14], bedeutet ein Überlauf, dass bei der Berechnung ein sogenanntes Carry-Bit entstanden ist, was außerhalb des darstellbaren Bereichs liegt. Das Carry-Bit kann man sich als „Übertrag“ vorstellen, ganz so, wie wir ihn auch von der schriftlichen Addition aus der Schule kennen. Angenommen, wir addieren 2-Bit-Ganzzahlen. Solange die addierten Zahlen klein

Listing 2

```
module TEA

  DELTA = 0x9e3779b9
  KEY_SIZE = 16 # bytes, 128 bit
  BLOCK_SIZE = 8 # bytes
  ROUNDS = 32

  module_function

  def encrypt(block, key)
    v0, v1 = UInt32.from_string(block)
    sum = 0
    k0, k1, k2, k3 = UInt32.from_string(key)

    ROUNDS.times do
      sum += DELTA

      v0 += ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5) + k1)
      v0 = v0 & UInt32::MASK

      v1 += ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5) + k3)
      v1 = v1 & UInt32::MASK
    end

    UInt32.to_string([v0, v1])
  end

  def decrypt(block, key)
    v0, v1 = UInt32.from_string(block)
    sum = DELTA << 5
    k0, k1, k2, k3 = UInt32.from_string(key)

    ROUNDS.times do
      v1 -= ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5) + k3)
      v1 = v1 & UInt32::MASK

      v0 -= ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5) + k1)
      v0 = v0 & UInt32::MASK

      sum -= DELTA
    end

    UInt32.to_string([v0, v1])
  end
end
```

genug sind und das Ergebnis wieder im Wertebereich von zwei Bit liegt, gibt es keinen Überlauf. Das wäre beispielsweise der Fall, wenn wir *1* (binär *01*) + *2* (binär *10*) = *3* (binär *11*) berechnen:

0 1
⊕ (Binäres *XOR*)
1 0
⊕
0 0 (Carry-Bit)
=
1 1

Es kommt zu keinem Übertrag und das Ergebnis unserer Addition ist einfach das binäre *XOR* beider Argumente. Sobald unser Ergebnis aber den Wertebereich von zwei Bit überschreitet, liegt ein Carry-Bit vor, was den Bereich jenseits der zwei verfügbaren Bits beeinflussen würde. Addieren wir nun also zum Beispiel *3* (binär *11*) + *2* (binär *10*) = *5* (binär *101*):

1 1
⊕
1 0
⊕
0 0 (Carry-Bit)
=
0 1

An der Stelle, wo die beiden Einsen mit *XOR* verknüpft werden, entsteht ein Carry-Bit, was nun zum nächsten Bit links davon wieder mit *XOR* hinzuaddiert werden müsste. Da links davon aber keine Bits mehr zur Verfügung stehen, fällt diese Information stillschweigend unter den Tisch. Das ist das Phänomen des Überlaufs und wir erhalten als Ergebnis *01*. Das heißt, bei 2-Bit-Zahlen ist das Ergebnis von *3* + *2* = *1*. Hätten wir die gleiche Rechnung mit 3-Bit-Zahlen durchgeführt, würde das Carry-Bit berücksichtigt werden und wir kämen zum richtigen Ergebnis:

0 1 1
⊕
0 1 0
⊕
1 0 0 (Carry-Bit)
=
1 0 1

Zusammenfassend bedeutet ein Überlauf also nur, dass wir alle Bits außerhalb des zulässigen Bereichs ignorieren. Die restlichen Bits innerhalb des Wertebereichs werden nach der Rechnung unverändert übernommen.

Im Fall von TEA möchten wir nun 32-Bit-Ganzzahlen ohne Vorzeichen simulieren, daher ist der für uns relevante Bereich genau 32 Bit groß, also 4 Byte. Um den Überlauf darzustellen, schneiden wir nach unseren Rechnungen also alles ab, was außerhalb dieser 4 Bytes liegt. Präziser: Wir setzen alle Bits außerhalb der für uns interessanten auf den Wert *0*. Das erreichen wir, indem wir unser Ergebnis mit binärem *AND* mit einer Bit-Maske verknüpfen. Die Bit-Maske ist dabei derart gestaltet, dass alle Bits, die wir behalten wollen, auf *1* gesetzt werden, die restlichen auf *0*. Bei einer 32 Bit-Zahl bedeutet das also, dass wir die ersten 32 Bit alle auf *1* setzen. Dies entspricht demnach der größtmöglich darstellbaren 32-Bit-Zahl. Hexadezimal kann man das ganz einfach darstellen, ohne dass man den genauen dezimalen Wert der Zahl im Kopf behalten müsste. Ein einzelnes Byte

Listing 3

```
module UInt32

  MASK = 0xFF_FF_FF_FF

  module_function

  def from_string(s)
    s.unpack('N*')
  end

  def to_string(uint32s)
    uint32s.pack('N*')
  end
end
```

Listing 4

```
module TEA

  BLOCK_SIZE = 8 # bytes

  class EncryptorAlgorithm

    attr_reader :key

    def initialize(key)
      @key = key
    end

    def block_size
      BLOCK_SIZE
    end

    def encrypt_block(block)
      TEA.encrypt(block, key)
    end

  end

  class DecryptorAlgorithm

    attr_reader :key

    def initialize(key)
      @key = key
    end

    def block_size
      BLOCK_SIZE
    end

    def decrypt_block(block)
      TEA.decrypt(block, key)
    end
  end
end
```

Listing 5

```
module Deterministic

  class Encryptor
    attr_reader :algorithm, :padding

    def initialize(algorithm:, padding:)
      @algorithm = algorithm
      @padding = padding
    end

    def encrypt(plain_text)
      padding.
        apply_to(plain_text).
        each_block(algorithm.block_size).
        map { |block|
          algorithm.encrypt_block(block)
        }.join
    end
  end

  class Decryptor
    attr_reader :algorithm, :padding

    def initialize(algorithm:, padding:)
      @algorithm = algorithm
      @padding = padding
    end

    def decrypt(encrypted)
      decrypted = encrypted.
          each_block(algorithm.block_size).
          map { |block|
            algorithm.decrypt_block(block)
          }.join

      padding.strip_from(decrypted)
    end
  end

end
```

kann hexadezimal gesehen genau den Wertebereich von 0x00 bis 0xFF, also dezimal 0 bis 255, annehmen. Dementsprechend können wir für 32 Bit, also nun vier statt einem Byte, die hexadezimale Zahl, die aus vier Mal FF besteht, nämlich 0xFFFFFFFF, verwenden. Das ist schließlich auch der Wert, den wir für *UInt32::MASK* verwenden. Wir simulieren also 32-Bit-Ganzzahlen, indem wir mit Rubys Ganzzahlen variabler Länge wie gewohnt rechnen, um dann das Ergebnis zum Schluss auf die relevanten 32 Bits zu begrenzen:

Listing 6

```
module StringRefinements
  refine String do

    def each_block(block_size, &blk)
      raise "Length is not a multiple of #{block_size}" unless bytesize % block_size == 0

      offset = 0
      enum = Enumerator.new do |yielder|
        while offset < bytesize
          sliced = byteslice(offset, block_size)
          yielder << sliced
          offset += block_size
        end
      end

      return enum unless block_given?
      enum.each(&blk)
    end

  end
end
```

Listing 7

```
key = "\x00" * TEA::KEY_SIZE
plaintext = "zickzack" * 3 # exakt drei Blöcke

encryptor_algorithm = TEA::EncryptorAlgorithm.new(key)
padding = Padding::PKCS5.new(encryptor_algorithm.block_size)
encryptor = Deterministic::Encryptor.new(algorithm: encryptor_algorithm, padding: padding)

encrypted = encryptor.encrypt(plaintext)

puts encrypted.to_hex.each_block(16).to_a.join(" ")

decryptor_algorithm = TEA::DecryptorAlgorithm.new(key)
padding = Padding::PKCS5.new(decryptor_algorithm.block_size)
decryptor = Deterministic::Decryptor.new(algorithm: decryptor_algorithm, padding: padding)

decrypted = decryptor.decrypt(encrypted)

puts decrypted
```

```
x = x & UInt32::MASK
```

Warum gerade binäres *AND*? Man nutzt hierbei aus, dass *x & 0...0 = 0...0* („*AND* mit Null gibt immer Null“) und *x & 1...1 = x* („*AND* mit *1* belässt den Wert, wie er vorher war“) ist. Unsere Maske besteht binär betrachtet aus genau 32 Einsen und der Rest sind lauter Nullen – daher bleiben die ersten 32 Bit nach dem binären *AND* unberührt und alle weiteren Bits werden auf *0* gesetzt.

Deterministische Verschlüsselung bei Block Ciphern

Genug der Theorie, wir wollen nun mit TEA etwas verschlüsseln und wieder entschlüsseln. Bisher erlaubt unsere Implementierung es aber nur, genau acht Byte zu verschlüsseln, da wir lediglich die Verschlüsselung eines einzelnen Blocks implementiert haben. Wie man beliebig lange Daten verschlüsselt, haben wir bereits geklärt: Wir müssen unsere Daten in Blöcke zu je acht Byte aufteilen, wobei wir den letzten Block mit Padding auffüllen müssen. Da unser Block Cipher an einen Schlüssel gebunden ist, führen wir zuvor noch das Konstrukt des *EncryptorAlgorithm* und des *DecryptorAlgorithm* ein, die jeweils ein uniformes Interface für die Ver- und Entschlüsselung mit einem Block Cipher unabhängig vom gewählten Algorithmus darstellen sollen (Listing 4) [10].

Anhand dieser abstrakten Basisklassen und der Padding-Klasse weiter oben kann man nun die Ver- und Entschlüsselung wie in Listing 5 definieren [15]. Das entspricht genau dem naheliegenden Ansatz einer Block-Cipher-Verschlüsselung, wie sie weiter oben eingeführt wurde. Wir füllen den ursprünglichen Plaintext mit Padding auf, zerteilen das Ergebnis in gleichlange Blöcke, verschlüsseln diese Blöcke einzeln und reihen sie wie an einer Kette hintereinander auf, um den finalen Ciphertext zu erlangen. Beim Entschlüsseln verfahren wir analog, nur dass wir hier aufpassen müssen, dass das Padding erst entfernt werden kann, nachdem der Ciphertext vollständig entschlüsselt ist. Wir werden im zweiten Teil des Artikels sehen, dass diese Art zu verschlüsseln genau dem sogenannten ECB-Modus von Block Ciphern entspricht. Dort werden wir auch ausnutzen, dass uns die Modularisierung des Algorithmus in *EncryptorAlgorithm* und *DecryptorAlgorithm* genügend Flexibilität verleiht, so dass wir dann auch AES statt TEA verwenden können, indem wir *Encryptor* und *Decryptor* beibehalten, einzig der *algorithm* muss ausgetauscht werden. Mit *Encryptor* und *Decryptor* können wir somit Verschlüsselung allgemein und unabhängig vom zugrundeliegenden Block Cipher definieren. Leider gibt es die Methode *String#each_block* nicht, wir können sie aber mittels Refinement einfach bereitstellen (Listing 6) [16]. Auf diese Art

Listing 8

```
module StringRefinements
  refine String do

    def to_hex
      unpack('H*').first
    end

  end
end
```

zu verschlüsseln funktioniert nun ohne Probleme, wie Listing 7 zeigt [15]. *String#to_hex* ist hier abermals ein Refinement, das an die bereits bekannte Methode *String#unpack* delegiert (Listing 8) [16].

Wenn wir das obige Beispiel ausführen, erhalten wir tatsächlich wieder den ursprünglichen Plaintext. Wenn wir es aber mehrmals ausführen, stellen wir fest, dass die Ausgabe des Ciphertexts immer absolut identisch ist. Zur besseren Abgrenzung der einzelnen Ciphertext-Blöcke wurden Leerzeichen eingefügt:

027cbf8db1fac8f9 027cbf8db1fac8f9 027cbf8db1fac8f9 3d020d08524ef0c8

Dank der Erkenntnisse, die wir über Stream Cipher gewonnen haben, wissen wir bereits, dass das kein gutes Zeichen ist. Eine Verschlüsselung, die rein deterministische Ergebnisse liefert, also bei gleicher Eingabe immer die gleiche Ausgabe liefert, kann nicht sicher sein. Das ist ausdrücklich nicht die Schuld unseres TEA-Algorithmus. Auch, wenn wir stattdessen AES verwenden, ist das Ergebnis das gleiche. Was noch schlimmer ist, nicht nur, dass das Ergebnis insgesamt identisch ist, nein, wir sehen sogar, dass sich die ersten drei Blöcke wiederholen. Nur der letzte ist anders. Dieses Phänomen lässt sich leicht erklären: Bei dem letzten Block handelt es sich um den verschlüsselten Padding-Block. Daher haben wir im Ciphertext auch vier Blöcke: die drei verschlüsselten Plaintext-Blöcke plus den verschlüsselten Padding-Block. Warum wiederholen sich die Plaintext-Blöcke?

Auch das wird unmittelbar klar, wenn man bedenkt, dass wir es hier trotz aller Komplexität mit einer deterministischen Funktion zu tun haben, die auf den einzelnen Blöcken agiert. Wenn ich bei gleichem Schlüssel zwei unterschiedliche, aber in ihrem Inhalt identische Blöcke verschlüssele, dann muss ich zwingend dasselbe Ergebnis erhalten. Alles andere wäre eine Überraschung, denn letztlich ist unser TEA auch nur ein Algorithmus mit klar vorgegebenen Rechenschritten. Mathematischer ausgedrückt bietet sich also bei Block Ciphern folgendes Bild: Wenn ich zwei Blöcke $x1$ und $x2$ habe, für die $x1 = x2$ gilt, dann muss auch $E(x1) = E(x2)$ gelten, wenn E meine Verschlüsselungsfunktion beschreibt – ganz unabhängig davon, wie „gut“ mein E ist. Selbst bei AES offenbart sich dieses Problem.

Letzten Endes ist die Problematik der deterministischen Verschlüsselung bei Block Ciphern auf den ersten Blick sogar noch schlimmer als bei den Stream Ciphern, weil nun nicht nur das Gesamtergebnis identisch bleibt, sondern zusätzlich auch noch Wiederholungen innerhalb der einzelnen Blöcke auftreten können. Sind Block Cipher also nutzlos? Oder zumindest weniger sicher als Stream Cipher? Die Antwort lautet erwartungsgemäß nein. Allerdings spielt es eine immense Rolle, wie genau wir mit ihnen verschlüsseln. Nur, weil etwas mit AES verschlüsselt wurde, heißt das noch lange nicht, dass es sicher verschlüsselt wurde. AES als Algorithmus an sich ist zwar sicher, wir als Anwender müssen aber zusätzlich noch sicherstellen, dass wir AES korrekt anwenden. Der naheliegendste Ansatz, Block für Block zu verschlüsseln, und die Ergebnisse aneinanderzureihen, hat sich leider als unsicher herausgestellt. Wie also wende ich meinen Block Cipher korrekt an?

Von den Stream Ciphern wissen wir bereits, dass man sich pseudozufälliger Elemente bedienen kann, um aus einer deterministischen Verschlüsselung eine nichtdeterministische zu machen. Wie diese Idee auf Block Ciphern angewandt werden kann und was es zudem noch braucht, um auch in der Welt der Block Ciphern eine zeitgemäße Verschlüsselung zu realisieren, die Authenticated Encryption liefert, werden wir uns im nächsten Teil anschauen. Wir werden die dazu notwendigen Maßnahmen erst für unsere Implementierung von TEA einführen, und so die Konzepte vollständig in Ruby-Code implementieren, um dann schließlich die äquivalenten Bestandteile von Rubys OpenSSL-Bibliothek einzuführen, mit denen eine zeitgemäße Verschlüsselung mit AES als zugrundeliegendem Block Cipher realisiert werden kann.

Martin Boßlet ist selbstständiger Berater mit den Schwerpunkten Sicherheit, Web- und Unternehmensanwendungen. Zudem vermittelt er regelmäßig sein Wissen in Seminaren als Trainer für Kryptografie, Programmiersprachen und verschiedene Themen aus dem Bereich der Webentwicklung. Seine besondere Leidenschaft gilt der Programmiersprache Ruby, wo er Mitglied im Core-Team ist und die Kryptografiebibliothek betreut.

Links & Literatur

[1] Boßlet, Martin: „Rubinrote Sicherheit“; in: Entwickler Magazin 4.20

[2] Boßlet, Martin: „Nichtdeterministische Verschlüsselung“; in: Entwickler Magazin 5.20

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

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

[5] https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS#5_and_PKCS#7

[6] https://github.com/emboss/block-ciphers/blob/main/code/padding.rb

[7] https://www.movable-type.co.uk/scripts/tea.pdf

[8] https://www.schneier.com/wp-content/uploads/2016/02/paper-relatedkey.pdf

[9] http://www.csshl.net/sites/default/files/downloadable/crypto/TEA_Cryptanalysis_-_VRAndem.pdf

[10] https://github.com/emboss/block-ciphers/blob/main/code/tea.rb

[11] https://ruby-doc.org/core-2.7.2/String.html#method-i-unpack

[12] https://ruby-doc.org/core-2.7.2/Array.html#method-i-pack

[13] https://github.com/emboss/block-ciphers/blob/main/code/uint32.rb

[14] https://www.electronics-tutorials.ws/de/kombinations/binaraddierer.html

[15] https://github.com/emboss/block-ciphers/blob/main/code/deterministic_tea.rb

[16] https://github.com/emboss/block-ciphers/blob/main/code/string_refinements.rb

Behördenrechnungen aus Papier werden durch XML ersetzt – zum Glück?

Zum Glück gezwungen?

Ab Ende November akzeptieren deutsche Bundesbehörden keine Papierrechnungen mehr, die XML-Datei wird für Rechnungen an Behörden Pflicht. Aber Geschichte hin und Rechtliches her, was steckt technisch dahinter, wie lässt es sich mit PHP umsetzen und ist das auch für Rechnungen zwischen Unternehmen anwendbar? Das soll dieser Artikel klären.

von Jochen Stärk

XML-Rechnungen an europäische Behörden (Business to Government, B2G) lassen sich gemäß den CEN-Vorgaben in EN16931-2 [1] als UN/CEFACT Cross-Industry-Invoice (CII) oder in Universal Business Language (UBL) ausdrücken (Kasten: „XRechnung“). Beide sind auch für Business-to-Business-(B2B-)Rechnungen geeignet, die CII ist aber auch die Basis für ZUGFeRD, das auch für Rechnungen an private Endkunden (Business to Consumer, B2C) eingesetzt werden kann. Einen Überblick über die Einsatzgebiete der einzelnen Formate finden sich zur besseren Übersicht in Tabelle 1.

XML-Beispieldateien gibt es beispielsweise unter den entsprechenden GitHub-Links [2], [3] und in einem XML-Editor (Kasten: „XML“).

Mit passenden Schemadateien [4], lässt sich eine Rechnung auch auf eigene Bedürfnisse anpassen. Weiterhin können Sie die Rechenregeln [5] oder die Liste möglicher Attributwerte [6] (= Codelisten) wie beispielsweise Mengeneinheiten einsehen oder eine Art Referenz beziehen, welche Elemente mit welchen Attributen in welcher Anzahl benutzt werden können. Diese findet sich beispielsweise im technischen Anhang von ZUGFeRD, der Teil des ZUGFeRD-Infopakets [7] ist, das neben Beispielen und Schemadateien auch die Codelisten enthält. Ebenfalls vorhanden ist eine Art offizielle Referenz für Behördenrechnungen: In EN16931-3-3 ist die Zuordnung zu den UN/CEFACT-Elementen aufgeführt, im Gegensatz zu den EN16931-1 bis -2 sind EN16931-3 bis -7 leider nicht kostenlos erhältlich. Mögliche Anlaufstellen für Communitysupport sind im Kasten „Hilfe zur Selbsthilfe“ zu finden.

	B2G	B2B	B2C
Papier	N**	J*	J
PDF	N**	J*	J
ZUGFeRD-PDF	J	J	J
OpenTrans	N	J	N
Financial Invoice	N	J	N
UN/CEFACT CII	J	J	N
UBL	J	J	N
EDIFACT	J	***	N

Tabelle 1: Formate und deren Einsatzgebiete; * = nicht maschinenlesbar **=bald nur noch an einige Länderbehörden *** kein XML-Format

Wenn Sie sicherstellen möchten, dass Sie nichts falsch gemacht haben, können Sie Ihre angepassten Dateien probeweise visualisieren, beispielsweise mit den XSL-Transformation-(XSLT-)Stylesheets von Kosit [8] oder Version 2 alpha von Mustang [9], die auch auf diesen Stylesheets basiert. So wird mit *java -jar mustang-cli-2.0.0.jar –action=visualize --source=xrechnung.xml* aus *xrechnung.xml* eine HTML-Datei *factur-x.html* erzeugt. Auch kann die Syntax der Dateien online oder offline automatisch validiert werden (Kasten: „Validierer“).

XML-typisch dient der Punkt als Dezimaltrenner (also *2.00 EUR*). Einzelpreise und Mengen haben vier, Zwischen- und Endsummen zwei Dezimalstellen. UN/CEFACT-Rechnungen bestehen in der Regel aus vier Namespaces. Es gibt eine Empfehlung, für diese bestimmte festgelegte Präfixe (*ram*, *rsm*, *qdt* und *udt*) zu verwenden und keinen Namespace zum Standard zu erklären, obwohl sich *ram* anbieten würde. Zeichensatztechnisch ist, zumindest bei ZUGFeRD, UTF-8 Pflicht.

Die erwähnten Rechenregeln im Standard EN16931-1 bergen, abgesehen von der etwas gewöhnungsbedürftigen Definition des Bruttopreises als „ohne Umsatzsteuer“ kaum Überraschungen: Die Idee dahinter ist, dass der Bruttopreis (gross price) eine Art Listenpreis ohne Umsatzsteuer ist. Der Preis in dieser Rechnung für diesen Kunden mit Nachlässen und Zuschlägen und ebenfalls ohne Umsatzsteuer ist der Nettopreis.

Generell wird bei der Berechnung zunächst pro Zeile Menge mit Preis multipliziert und die Summe der Zeilen mit demselben Umsatzsteuersatz addiert, auf zwei Stellen gerundet und der darauf anwendbare Umsatzsteuersatz angewendet. Man kann bei EN16931 nur eine (beliebige) Rechnungswährung verwenden, die im Element *InvoiceCurrencyCode* angegeben wird. Zu-

Hilfe zur Selbsthilfe

Communitysupport gibt es beispielsweise im Forum der ZUGFeRD Community [12] oder auf Englisch in einer vom Autor ins Leben gerufenen Google Group [13]. Issues zu @GPs Factur-X-Bibliothek beziehungsweise Mustang können wie üblich über GitHub [14] beziehungsweise [15] eingestellt werden, generell weiterführende Informationen zum Thema finden sich unter [16], [17], [18] sowie auf der vom Autor betreuten Seite [19].

XML

Namespaces

XML Namespaces sind die Möglichkeit, mehrere XML-Dokumente ineinander zu verarbeiten. Dafür werden sowohl der Namespace als auch ein grundsätzlich beliebiges Präfix definiert. Letzteres erhält einen Doppelpunkt vor dem Element, etwa *<xsl:value-of></xsl:value-of>*. Nur der Standard-Namespace bedarf keines Präfix. Der Vorteil dieser kontrollierten Dokumentdurchmischung besteht darin, dass weiterhin alle Namespaces anhand ihrer Schemadateien geprüft und alle Features aller Namespaces genutzt werden können.

XPath

XPath könnte man auch als die regulären Ausdrücke von XML bezeichnen. Es ist eine Methode innerhalb eines XML-Dokuments, mit der Elemente oder Attribute absolut oder relativ adressiert, gesucht oder aggregiert werden können – also beispielsweise eine Summe bilden. XPath-Ausdrücke werden beispielsweise in XSLT und Schematron verwendet. Der „kleine Bruder“ XPointer kann nicht aggregieren und der „große Bruder“ XQuery gilt vielmehr als eine Art SQL-ähnliche Abfragesprache für XML. Er unterstützt nicht nur die Abfrage ganzer Gruppen, sondern beispielsweise auch Schleifen.

	Offline	Online
XRechnung	https://github.com/itplr-kosit/validator	https://ref.xrechnung.bund.de/
PDF/A	https://verapdf.org/	https://www.pdf-tools.com/pdf20/en/products/pdf-converter-validation/pdf-validator/
ZUGFeRD	https://www.zugferd-community.net/de/open_community/validation	https://www.mustangproject.org/ ab Version 2 alpha

Tabelle 2: Validierer

Validierer

Bei *ref.xrechnung.bund.de* handelt es sich um die Testversion der „Zentralen Rechnungseingangsplattform des Bundes“ ZRE, die Produktivversion zum Upload von XRechnungen an Bundesbehörden findet sich auf *bund.xrechnung.de* (ohne „ref“). Sie setzt auf den Kosit-Validierer, allerdings wird zusätzlich die maximal 46-stellige Leitweg-ID (Spezifikation: [20]) geprüft. Für die Prüfregeln gibt es meist offizielle Regeldateien, beispielsweise als Schematron. Die meisten Validierer halten sich an die offiziellen Quellen, zum Beispiel Mustang (**Abb. 1**).

Die wenigsten Validierer prüfen innerhalb der Positionen: Während die Rechnungsgesamtsummen meist aus den addierten Positionssummen überprüft werden, wirft beispielsweise noch kein Validierer einen Fehler, wenn Sie den Bruttopreis fälschlicherweise mit Mehrwertsteuer ausgewiesen haben.

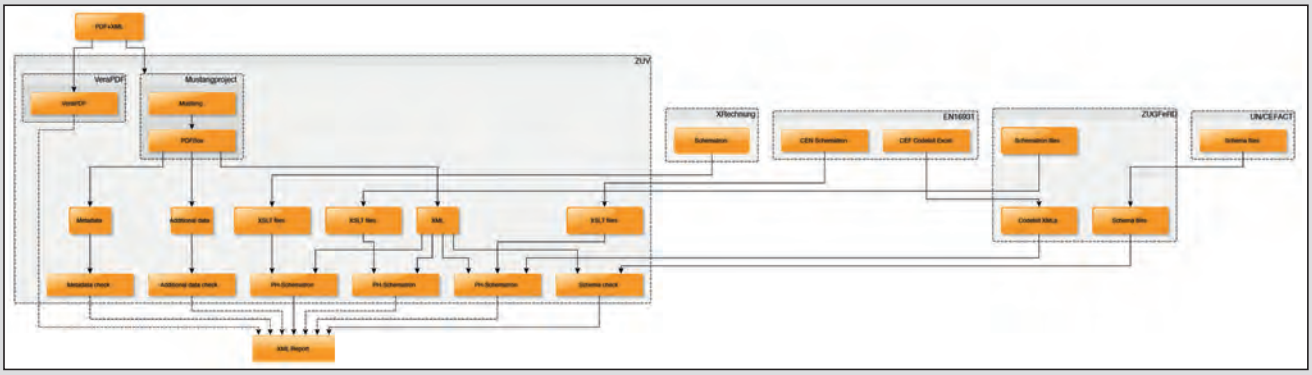


Abb. 1: Architektur der Validiererkomponente von Mustang

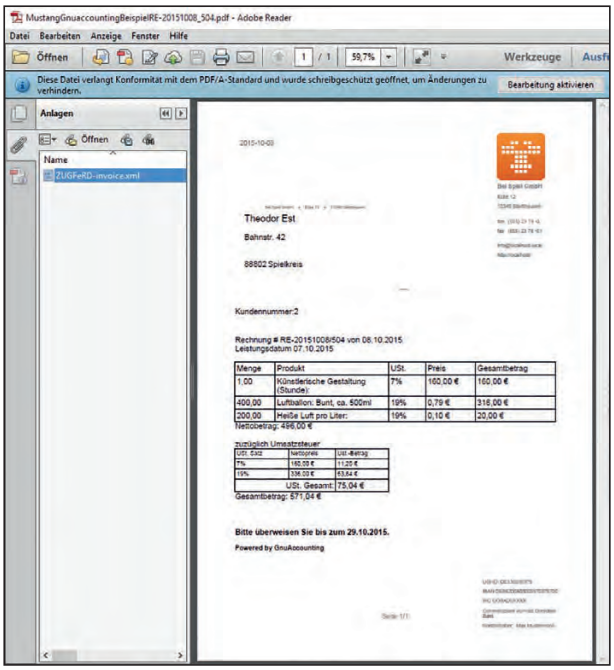


Abb. 2: Eine ZUGFeRD-Rechnung im Adobe Reader

sätzlich gibt es ein Attribut *currencyID*, dieses ist bei *TaxTotal* verpflichtend: Das hat den Hintergrund, dass in manchen Ländern die Angabe der Umsatzsteuerlast in Landeswährung verpflichtend ist. Entspricht dort der *InvoiceCurrencyCode* nicht der Landeswährung, gibt es zwei *TaxTotal*: eines mit dem Attribut *currencyID* der Rechnungs- und eines mit dem der Landeswährung. Da *TaxTotal* also theoretisch bis zu zweimal auftauchen kann, ist das *currencyID*-Attribut generell bei nur einem Element verpflichtend. Können Sie den XML-Teil Ihrer Rechnungen schreiben und validieren, ist ein wichtiger Meilenstein erreicht. Setzen Sie auch die *BuyerReference* und einen Ansprechpartner mit Namen, E-Mail-Adresse und Telefonnummer ein, haben Sie schon eine gültige XRechnung, die durch den Kosit-Validierer läuft.

XRechnungen lesen

Beim Lesen von XML spricht scheinbar erst einmal nichts gegen SimpleXML, zumal es XPath unterstützt.

```
$xml = new SimpleXMLElement(file_get_contents("factur-x.xml"));
$results = $xml->xpath('//ram:DuePayableAmount');
```

SimpleXML funktioniert wunderbar und liefert den zu zahlenden Betrag. Da *ram* aber nur das empfohlene und nicht das vorgeschriebenes Namespace Prefix ist, wäre streng genommen die Namespace-unabhängige Formulierung per *local-name()* korrekt:

```
$results = $xml->xpath('//*[local-name()="DuePayableAmount"]');
```

Auch Zugriff auf die Attribute per XPath ist möglich:

```
$results = $xml->xpath("//ram:BilledQuantity/@unitCode");
```

Listing 1: Lesen der Summe per XPath-sum() und DOMDocument

```
$doc = new DOMDocument();
$doc->load("factur-x.xml");

$xpath = new DOMXPath($doc);
$xpath->registerNamespace("ram", "urn:un:unece:uncefact:data:standard:
                    ReusableAggregateBusinessInformationEntity:100");
$xpath->registerNamespace("rsm",
        "urn:un:unece:uncefact:data:standard:CrossIndustryInvoice:100");
$xpath->registerNamespace("udt",
         "urn:un:unece:uncefact:data:standard:UnqualifiedDataType:100");
$xpath->registerNamespace("qdt",
           "urn:un:unece:uncefact:data:standard:QualifiedDataType:100");
$query = 'sum(//ram:SpecifiedTradeSettlementLineMonetarySummation/
                                                ram:LineTotalAmount)';
$xpathResult = $xpath->evaluate($query, $doc);
```

Listing 2: ZUGFeRD schreiben

```
require_once("vendor/autoload.php");
// Generating Factur-X PDF invoice from PDF and Factur-X XML
$facturx = new \Atgp\FacturX\Facturx();
$facturxPdf = $facturx->generateFacturxFromFiles("invoiceInPDFA.
                                                pdf","factur-x.xml");
$f=fopen("ZUGFeRD.pdf", "w");
fwrite($f, $facturxPdf);
fclose($f);
```

XRechnung

Die XRechnung ist die Anforderungsspezifikation (eine sogenannte CIUS) der deutschen Behörden an strukturierte elektronische Rechnungen in UBL oder UN/CEFACT. XRechnung Version 2 ist ab dem 01.01.2021 verpflichtend, bis dahin gilt Version 1.2.2. Bestimmte Elemente, die in EN16931 optional sind, wurden dabei als verpflichtend erklärt, beispielsweise kann eine Rechnung an Firma ABC in Frankreich gerichtet sein – in Deutschland wäre eine solche Rechnung ohne Postanschrift nach §19 UStG nicht vorsteuerabzugsfähig. Die Postanschrift ist daher Pflicht, daneben auf Behördenwunsch auch ein Ansprechpartner beim Absender und die Leitweg-ID im Element *BuyerReference*.

Leitweg-ID

Die Leitweg-ID ist eine durch die Verwaltung vergebene Empfänger-ID für Behörden und deren Abteilungen. Bundesbehörden (und einige Länder) haben einen zentralen elektronischen Rechnungseingang. Lädt man dort eine XRechnung hoch, kann sie durch die Leitweg-ID automatisch dem richtigen Empfänger zugeordnet werden.

Skonto

Da Skonto an Behörden wohl nur in Deutschland üblich ist, konnte sich schon EN16931 nicht dazu durchringen, die entsprechenden Felder von UBL bzw. UN/CEFACT in EN16931 einzublenden, die XRechnung definiert aber auf S. 36f in [10] ein Format, wie man Skonto maschinenlesbar in ein Freitextfeld codieren kann.

Beschränkungen mit eckigen Klammern haben wir schon gesehen, logische Operatoren wie „und“ und „oder“ sind in XPath ebenfalls kein Problem. Der Haken: SimpleXML unterstützt als Rückgabewert leider nur Nodesets:

```
$results = $xml->xpath('//ram:DuePayableAmount');
//funktioniert zwar
$results = $xml->xpath("count(//*[local-name()='IncludedSupplyChainTrade
                                                    LineItem'])");
//aber eben nicht
```

Mengeneinheiten

Die UN/ECE Recommendation 20 (und 21) sammelt seit Jahrzehnten auf Rechnungen gebräuchliche Einheiten und wird auch bei anderen Standards wie UBL oder BMECAT verwendet. In den erwähnten Codelisten [6] tauchen sie in der Tabelle *Units* auf. *H87* ist „Piece“, einige verwenden das vielleicht auch noch akzeptable *C62* („One“). Weiterhin gebräuchlich sind Pauschale (*LS*) sowie die Gewichts-, Zeit- und Raumeinheiten, beispielsweise die in Tabelle 3 aufgelisteten.

Einheit	Code
Stück	H87
Pauschale	LS
Gramm	GRM
Kilogramm	KGM
Metrische Tonne	TNE
Sekunde	SEC
Minute	MIN
Stunde	HUR
Tag	DAY
mm	MMT
cm	CMT
m	MTR
mm^2	MMK
cm^2	CMK
m^2	MKT
mm^3	MMQ
cm^3	CMQ
Liter	STL
m^3	MTQ

Tabelle 3: Einige Einheiten und ihre Unit-Codes

Bei mehr als 2100 Einheiten mit etwa *C63* („Parsec“) über *F57* („Milliampere per pound-force per square inch“) oder *B69* („Megacoulomb per cubic metre“) ist bestimmt für jeden was dabei. Nur nicht für die Gerüstbauer, denn der dort teilweise übliche „Wochenmeter“ befindet sich noch im Normierungsprozess. Die Komplettliste findet sich im Reiter Units der Attributcodes [6].

Will man XPath Funktionen wie *count()* oder *sum()* verwenden, kann man immer noch auf DOMDocument umsatteln. Dazu müssen die Namespaces erst einmal registriert werden, wie Listing 1 zeigt.

In diesem Beispiel wird der Nettobetrag nachgerechnet. Dieser steht ebenfalls im *LineTotalAmount*:

```
$query = '//ram:SpecifiedTradeSettlementHeaderMonetarySummation/
                                                    ram:LineTotalAmount';
$entries = $xpath->evaluate($query, $doc);
echo "Nettobetrag: {$entries[0]->textContent}";
```

Bevor Sie auf die Idee kommen, auf dieser Grundlage nachzurechnen: Im Prinzip ist die Prüfung mittels XPath die Funktionsweise der Schematron-Dateien und damit der Validierer. Wenn Sie keine Behörde sind, könnten Sie der Meinung sein, auch keine XRechnung lesen können zu müssen. Das stimmt, aber bei EN16931 wurde absichtlich darauf geachtet, auf gewisse Standards aufzubauen, die in der Wirtschaft verwendet werden. Und das UN/CEFACT CII XML, das Sie gerade parsen, ist auch die XML-Struktur, die bei ZUGFeRD in PDF-Dateien eingebettet wird.

ZUGFeRD/Factur-X

Die Idee hinter ZUGFeRD ist es, eine XML-Datei mit einer maschinenlesbaren Rechnung in eine PDF/A-3-

PDF/A und Ghostscript

PDF/A ist eine Teilmenge von PDF, das speziell für die Archivierung sinnvolle Features beinhaltet: Im normalen PDF möglich, in PDF/A jedoch absichtlich nicht erlaubt sind beispielsweise Schriftarten, die nicht im Dokument eingebettet sind. Das für ZUGFeRD verwendete PDF/A-3 bezieht sich auch nicht auf das Papierformat DIN A3, sondern auf die dritte Version des PDF/A-Standards. Das normale PDF gilt als so komplex, dass es syntaktisch praktisch nicht mehr prüfbar ist, für PDF/A hingegen gibt es Validierer wie VeraPDF. Eine Umwandlung von PDF/A-1 in PDF/A-3 ist mit vielen ZUGFeRD-Bibliotheken möglich, eine solide valide PDF/A-1-Datei erstellt man dazu beispielsweise mit der Open-Source-Software Ghostscript. Da auch Farbprofile eingebettet werden, lädt man sich zusätzlich die Datei *AdobeRGB1998.icc* herunter [11] und entpackt sie zusammen mit der zu konvertierenden PDF-Datei in ein lokales Verzeichnis. Dahin kopiert man auch die Datei *PDFA_def.ps* aus dem *lib*-Verzeichnis von Ghostscript. Dabei handelt es sich um eine Textdatei, die noch angepasst werden muss, *srgb.icc* ersetzt man darin durch *AdobeRGB1998.icc*. Die normale PDF-Datei *input.pdf* erledigt nach dem Aufruf von

```
gs -P -dPDFA=1 -sColorConversionStrategy=RGB -sDEVICE=pdfwrite -oOutput.pdf
                -dPDFACompatibilityPolicy=1 -dRenderIntent=3 PDFA_def.ps Input.pdf
```

im lokalen Verzeichnis die Konvertierung. Unter Windows heißt *gs* traditionell *gswin64c.exe*. Seit Ghostscript 9.5 ist der Zugriff auf Skripte (*PDFA_def.ps*) und Ressourcen (*AdobeRGB1998.icc*) sicherheitstechnisch stark eingeschränkt, der Parameter *-P* gibt aber immerhin das aktuelle Arbeitsverzeichnis frei.

Mittels Composer lässt sich @GPs Factur-X-Bibliothek einfach einbetten.

Datei mit einer menschenlesbaren Rechnung einzubetten. Das passiert so unauffällig, dass einige Nutzer vielleicht schon ZUGFeRD-Dateien für normale PDFs gehalten haben – die eingebettete XML-Datei ist nur am Heftklammer-Symbol zu erkennen (**Abb. 2**). Die Franzosen haben mit Factur-X einen ZUGFeRD-Fork geschaffen, der zu allem Überfluss auch noch Version 1 heißt, zu einem Zeitpunkt, als ZUGFeRD bereits Version 2 erhalten hatte. Der Wille ist allerdings vorhanden, die Zweige wieder zusammenzuführen, ZF2.1 entsprach genau Factur-X 1.0.50, daher gilt Factur-X auch als der internationale Name von ZUGFeRD. Eine PDF-A-Datei ist mit guter Software erhältlich oder man konvertiert normales PDF zu PDF/A, beispielsweise mit Ghostscript (Kasten: „PDF/A und Ghostscript"). Hat man ein gültiges PDF/A-1 seiner Rechnung, kann die Arbeit beginnen.

Per Composer lässt sich @GPs Factur-X-Bibliothek einfach einbetten:

```
composer require atgp/factur-x
```

Wenn man dann XML beispielsweise in *factur-x.xml* schreibt, kann man es wie in Listing 2 mit einer Rechnung in PDF/A-1 (im Beispiel *invoiceInPDFA.pdf*) zu einer Factur-X-(lies ZUGFeRD-)Datei (*ZUGFeRD.pdf*) konvertieren:

Wie aus der XML-Struktur einer ZUGFeRD-PDF-Datei gelesen werden kann, zeigt das folgende Beispiel:

```
<?php
  require_once("vendor/autoload.php");
  $facturx = new \Atgp\FacturX\Facturx();
  $facturxXml = $facturx->getFacturxXmlFromPdf($facturxPdf, true);
```

Profile

Profile definieren eine verbindliche (Unter-)menge von Elementen und Attributen, weil nicht jeder alles liefern kann oder braucht, ein Beispiel liefert **Abbildung 3**. Beim XRechnung-Feldtest hatte streng genommen schon keine der Papierrechungen alle Angaben [21]. ZUGFeRD Extended erlaubt beispielsweise Sammelrechnungen, die Profile Minimum und Basic-WL sind vor allem in anderen europäischen Ländern interessant, weil sie in Deutschland nicht genügend Attribute zum Vorsteuerabzug nach §14 UStG enthalten.
Welches Profil verwendet wird, steht in der *ram:ID* von *ram:GuidelineSpecifiedDocumentContextParameter* (Tabelle 4). Der Wert *urn:cen.eu:en16931:2017* ist für EN16931 Pflicht, aber durch das Anfügen von *#* können optionale Angaben ergänzt werden.

ZUGFeRD 2.1.1	XRechnung 2
Minimum (*urn:factur-x.eu:1p0:minimum*)	Standard (*urn:cen.eu:en16931:2017#compliant#urn:xoev-de:kosit:standard:xrechnung_2.0*)
Basic-WL (*urn:factur-x.eu:1p0:basicwl*)	Extension (*urn:cen.eu:en16931:2017#compliant#urn:xoevde:kosit:standard:xrechnung_2.0#conformant#urn:xoev-de:kosit:extension:xrechnung_2.0*)
Basic (*urn:cen.eu:en16931:2017#compliant#urn:factur-x.eu:1p0:basic*)	
EN16931 (*urn:cen.eu:en16931:2017*)	
Extended (*urn:cen.eu:en16931:2017#conformant#urn:factur-x.eu:1p0:extended*)	
„Referenzprofil" XRechnung (Guideline-IDs siehe XRechnung)	

Tabelle 4: Profile verschiedener Standards

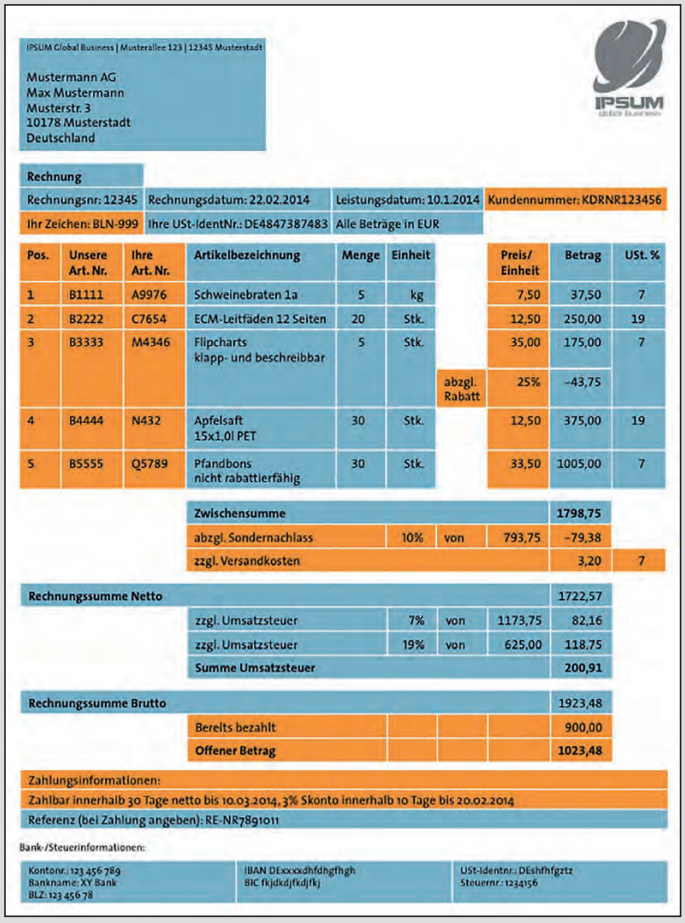


Abb. 3: Strukturiert abbildbare Informationen in zwei verschiedenen ZUGFeRD-Profilen; blau: Basic, orange: Comfort (entspricht EN16931) [22]

$facturxXml können Sie jetzt so parsen wie bereits besprochen.

Fazit

Die Behördenrechnung führt zu einer gewissen Konsolidierung sowohl der Formate als auch der Features. Sie hat zwar bislang noch nicht zu einem freien, plattübergreifenden Offline-Viewer geführt, aber in diese Bresche springt PDF, beziehungsweise ZUGFeRD. Die Behördenrechnung kann ebenfalls ein Anlass sein, auch normale Kunden mit E-Rechnungen zu beliefern: Schreibt man ohnehin schon PDF und kann jetzt auch XML, spricht wenig gegen ZUGFeRD.

Jochen Stärk ist PHP-Freiberufler in Frankfurt am Main und hat 2014 die Java-Bibliothek Mustangproject ins Leben gerufen. Schon Aufgrund der schieren Anzahl bereits verfügbarer E-Rechnungsexperten nennt er sich selbst gern Mustangproject.org Chief ZUGFeRD Amateur.

Links & Literatur

[1] https://www.beuth.de/de/technische-regel/din-cen-ts-16931-2/274991011

[2] https://github.com/itplr-kosit/xrechnung-testsuite/blob/master/src/test/business-cases/standard/01.01a-INVOICE_uncefact.xml

[3] https://github.com/ConnectingEurope/eInvoicing-EN16931/tree/master/cii/examples

[4] http://fnfe-mpe.org/wp-content/uploads/2020/06/Factur-X-1.0.05-EN-Complete.zip

[5] https://www.beuth.de/de/norm/din-en-16931-1/274990963

[6] https://ec.europa.eu/cefdigital/wiki/download/attachments/106234259/EN16931 code lists values used from 2020-02-14.xlsx

[7] https://www.ferd-net.de/standards/zugferd-2.1.1/index.html

[8] https://github.com/itplr-kosit/xrechnung-visualization

[9] https://www.mustangproject.org

[10] https://www.xoev.de/sixcms/media.php/13/200-XRechnung-2020-06-30.pdf

[11] https://www.adobe.com/digitalimag/adobergb.html

[12] https://www.zugferd-community.net/forum/

[13] https://groups.google.com/forum/?hl=de#!forum/zugferd

[14] https://github.com/atgp/factur-x

[15] https://github.com/ZUGFeRD/mustangproject/

[16] https://www.zugferd-tech.de

[17] https://invoice.fans

[18] https://rechnungsaustausch.org

[19] https://www.mustangproject.org/einvoices/

[20] https://www.xoev.de/xrechnung-16828#LID

[21] https://www.verband-e-rechnung.org/pdfs/Planspiel-eRechnung_Bericht.pdf

[22] https://www.bitkom.org/sites/default/files/file/import/140916-Broschuere-Zugferd.pdf

[23] https://www.bundesfinanzministerium.de/Content/DE/Downloads/BMF_Schreiben/Weitere_Steuerthemen/Abgabenordnung/2019-11-28-GoBD.html

[24] https://www.awv-net.de/themen/fachergebnisse/musterverfahrensdoku/index.html

[25] https://www.dstv.de/fuer-die-praxis/arbeitshilfen-praxistipps/musterverfahrensdokumentation-zum-ersetzenden-scannen

[26] https://www.ferd-net.de/ressourcen/bund-laender-uebersichten/stand-der-regelungen-bei-der-e-rechnung-beim-bund-und-in-den-bundeslaendern.html

Rechtliches

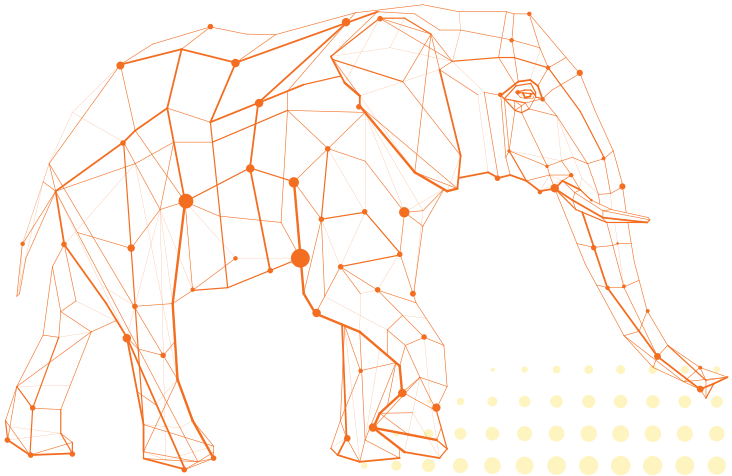
Zusätzlich zu den Grundsätzen Ordnungsgemäßer Buchführung (GOB) gelten in Deutschland für Firmen zum Versand und Empfang von elektronischen Rechnungen nicht nur die ohnehin vorgeschriebenen Prüf- und Archivierungspflichten, sondern zusätzlich auch die Grundsätze zur ordnungsmäßigen Führung und Aufbewahrung von Büchern, Aufzeichnungen und Unterlagen in elektronischer Form sowie zum Datenzugriff (GoBD).

GoBD

Die GoBD [23] greifen selbst dann, wenn Sie als Einzelgewerbetreibender nur eine per PDF-Datei in Rechnung gestellte Leistung bezahlen, und erst recht, wenn Sie selbst welche stellen. Sie beinhalten unter anderem, dass Sie eine revisionssichere digitale Archivierung gewährleisten und bei Bedarf eine entsprechende Verfahrensdokumentation vorweisen können. Eine Musterverfahrensdokumentation finden Sie auf [24]. Falls Sie zusätzlich Papierrechnungen abfotografieren und das Original vernichten möchten, fällt eine weitere Verfahrensdokumentation an, die Musterverfahrensdokumentationen für „ersetzendes Scannen“ [25].

Rechnungen an Behörden: 2014/55 EU und EN16931

Das Europäischen Parlament hat bereits vor sechs Jahren in Richtlinie 2014/55/EU entschieden, dass alle Mitgliedsstaaten E-Rechnungen entgegennehmen können müssen. Dafür wurde der Standard EN16931 geschaffen, und sowohl der Bund als auch die Länder haben entsprechende E-Rechnungsgesetze bzw. -verordnungen auf den Weg gebracht. E-Rechnungen an Bundesbehörden (und an Landesbehörden in Bremen) werden ab dem 27. November 2020 Pflicht und ersetzen ab 1000 Euro auch Papierrechnungen. Folgende Bundesländer schaffen Papierrechnungen in absehbarer Zeit ebenfalls ab: Bremen ab dem 27.11.2020, Baden-Württemberg und vermutlich das Saarland ab dem 01.01.2022 sowie Hessen ab dem 18.04.2024. [26] Für Behördenrechnungen gilt zusätzlich die Pflicht, die im EN16931-Profil von ZUGFeRD zu leistenden Angaben der XRechnung zu erfüllen.



IPC International PHP Conference

HYBRID EDITION

JUNE 7 – 11, 2021 | BERLIN OR ONLINE

EXPO: JUNE 8 – 9, 2021

LEARN FROM THE BEST

With over a decade's experience, the International PHP Conference is the must-attend event for web developers from around the globe. Attend inspiring sessions, unique in-depth workshops, and benefit from our experts' invaluable insights!

EXPAND YOUR KNOWLEDGE

At IPC, discover the exciting interaction of PHP with technologies such as Symphony, Docker and Kubernetes, as well as JavaScript and TypeScript. You'll also learn about the newest features in PHP, plenty of groundbreaking tools, test automation, cutting edge web security tactics, and state-of-the-art web architectures.

SHAPING THE FUTURE

Get ready to benefit from innovative approaches and valuable insights from IPC's outstanding speakers. Gain a real competitive advantage for your projects in a rapidly changing digital age.

phpconference | #IntPHPcon ipc.germany phpcon

Presented by: e.academy webmagazin Organizer: S&S Media Group

UNTIL FEBRUARY 25
✓ Workshop Day for free
✓ Group discount
✓ Save up to €700

TRACKS AND TOPICS

- PHP Core Technology
- Web Development
- Agile & Company Culture
- Security
- Software Architecture
- DevOps
- Testing & Quality
- #slideless (pure coding)

HIGHLIGHTS

- **70+** Workshops, sessions and keynotes
- **50+** International speakers and experts
- **Power Workshops:** June 7 and 11, 2021
- **Expo** of top-notch innovative businesses: June 8 – 9, 2021
- **2 in 1 Special:** An IPC ticket also grants access to the parallel International JavaScript Conference
- **Networking:** Get to know interesting people who share the same passion and enlarge your business network
- **All-inclusive:** Buffets, snacks and drinks
- Official **Certificate of Participation**
- **Goodies:** Bag, T-shirts, etc.

EXCLUSIVE SPECIALS UNTIL FEBRUARY 25

Early Bird Special: Save up to €700 until February 25.

Workshop Day for free: Register for a 3-Day Pass until February 25 and receive a Workshop Day for free.

Group Discount: Receive an additional 10% discount when registering with 3+ colleagues.

Extra Specials: Freelancers and employees of scientific institutions benefit from individual offers – if you're interested, just send an email to contact@phpconference.com.

phpconference.com

PHP End to End – Teil 1

Die richtige Technologie wählen

Bevor ein umfassendes PHP-Softwareprojekt gestartet wird, sind die Anforderungen zu erheben und zu dokumentieren. Bei Webapplikationen gehören neben den Kundenanforderungen auch wichtige Entscheidungen zur Technologie dazu. Eine gemeinsame Diskussionsbasis ist notwendig und muss offen geführt werden.

von Elena Bochkor und Dr. Veikko Krypczyk

In einer Vielzahl von Artikeln, Blogeinträgen und Diskussionen in Online-Boards werden die unterschiedlichsten Aspekte rund um ein PHP-Entwicklungsvorhaben diskutiert. Ein solches Format ist angebracht, um die notwendige Tiefe der Betrachtungen zu erreichen. Beispielsweise diskutieren wir die Vor- und Nachteile der zur Auswahl stehenden Frameworks oder die Art und Weise der Anbindung einer Datenbank. Dabei ist uns aufgefallen, dass man aufgrund dieser feingranularen Betrachtungen Gefahr läuft, das große Ganze aus dem Blick zu verlieren. Gerade am Anfang eines neuen Entwicklungsprojekts ist es an der Zeit, einen Blick aus der Vogelperspektive auf das geplante Vorhaben zu werfen. Gemeint sind die Vorgehensweise und die grundsätzlichen technologischen Entscheidungen. Dabei stehen folgende Fragen im Fokus:

1. *Technologieauswahl*: Die Breite der zur Auswahl stehenden Möglichkeiten ist heute so groß, dass es schwerfällt, im konkreten Fall eine passende Technologie auszuwählen. Haben wir tatsächlich mehrere Alternativen gegeneinander abgewogen oder sind wir (vereinfacht) so vorgegangen, wie wir es immer machen?
2. *Diskussion mit dem Kunden*: Der Auftraggeber hat primär seine fachlichen Anforderungen im Blick. Das ist auch gut so. Dennoch ist er bis zu einem bestimmten Grad mit in die (grundsätzlichen) Auswahlentscheidungen zur Technologie einzubeziehen. Haben wir hier offen und ehrlich beraten und gemeinsam eine passende Option ausgewählt?
3. *Vorgehensweise:* Als Entwickler sind uns die einzelnen Schritte bis zum ersten finalen Release vertraut und gewissermaßen in Fleisch und Blut übergegangen. Für viele Kunden ist ein Softwareentwicklungsprojekt jedoch eine einmalige oder seltene Angelegenheit. Der Abteilungsleiter im Vertrieb des Kunden hat nur eine vage oder gar keine Vorstellung davon, welche Schritt zu durchlaufen sind, an welchen Stellen seine Mitarbeit unabdingbar ist und wie lange üblicherweise die einzelnen Vorgänge dauern. Haben wir die Vorgehensweise mit dem Kunden erörtert?
4. *Interne Struktur*: Umfassende Softwareentwicklungsprojekte werden in unzählige Teilaufgaben zerlegt, an denen viele unterschiedliche Mitarbeiter beteiligt sind. Sie sind mit Detailfragen beschäftigt. Um jedoch für eine dauerhafte Motivation und damit eine gleichbleibend gute Qualität zu sorgen, ist es nötig, auch hier einen breiten Blick auf das Projekt zu haben. Sind alle Beteiligten umfassend – zum Beispiel im Rahmen einer Kick-off-Veranstaltung und regelmäßigen Meetings zu wichtigen Meilensteinen – informiert, damit alle genau das Ziel und den Weg dorthin kennen?

Im Rahmen einer mehrteiligen Artikelserie wollen wir anhand eines typischen Entwicklungszyklus einer Webapplikation auf der Basis von PHP ein solches Vorhaben betrachten.

Als Ansatzpunkt unserer Betrachtungen wollen wir die Idee der End-to-End-Prozessgestaltung aufgreifen. Hiermit meinen wir im Bereich der Softwareentwicklung, dass wir beim Kunden beginnen, d. h. mit der Erhebung der Anforderungen, und am Ende des Gesamtvorhabens

Artikelserie

Teil 1: Die richtige Technologie wählen
Teil 2: Architektur und Struktur
Teil 3: Anbindung der Datenwelt
Teil 4: Zum Kunden

auch wieder beim Kunden ankommen, indem wir ihm die Software zur Verfügung stellen und darüber umfassende weitere Leistungen, wie Wartung, Support usw. anbieten. In Bezug auf das Entwicklungsteam bedeutet dies, dass ein Team durchgängig von Beginn bis Ende am Projekt arbeitet. In diesem Teil der Artikelserie geht es primär um eine Diskussion der richtigen Technologie für das anstehende Webprojekt. Ausgangspunkt sind die spezifischen Kundenanforderungen. Wie kann es gelingen, von Anfang an eine gute Zusammenarbeit mit dem Kunden zu etablieren und sich nicht in technischen Details zu verlieren?

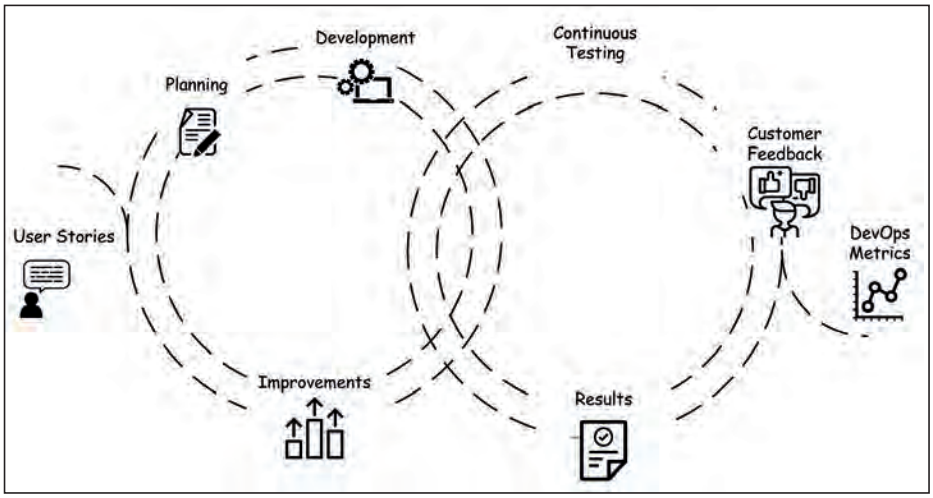


Abb. 1: DevOps-Schleife (nach einer Idee von [4])

The Big Picture

Sprechen wir von End to End in der Softwareentwicklung, meinen wir die Konzeption und Umsetzung von umfassenden Lösungen, die beim Kunden starten und auch wieder bei ihm enden. Die End-to-End-Softwareentwicklung umfasst mehrere Themen [1]: Software, Hardware und menschliche Fähigkeiten. Es bedeutet, dass nur ein Team durchgängig von Anfang bis Ende an der Umsetzung der Anwendung arbeitet. Diese durchgängige Arbeitsweise beschleunigt den Workflow. Notwendige Änderungen können schneller berücksichtigt und Ressourcen gespart werden. Jeder Entwickler, der am Entwicklungsprozess beteiligt ist, kennt vollständig die endgültigen Ziele, Meilensteine und die Art der Aufgaben, mit denen er sich befassen wird. Eine End-to-End-Vorgehensweise wirkt sich dann qualitätssteigernd auf alle Phasen aus:

1. *Planung*: Wir haben die Möglichkeit, die Wünsche der Nutzer direkt in der Planung umzusetzen. Nutzerforschung und Projektplanung erfolgen aus einer Hand.
2. *Codierung*: Von der Planung bis zur Umsetzung gibt es keine Brüche. Es ist nicht erforderlich, die Details der anstehenden Prozesse zu erläutern. Das Team ist in der Zusammenarbeit erprobt und man kennt die Knackpunkte.
3. *Aktion*: Man kommt schneller zum Handeln, d. h. zwischen der Planung und der Umsetzung klafft keine große zeitliche Lücke. Die Umsetzung erfolgt schneller und damit verkürzt sich die Time to Market.
4. *Auswertung*: Die letzten Schritte umfassen eine Bewertung des Plans und aller durchgeführten Arbeiten. Die Analyse gestaltet sich einfacher. Wenn es Vorschläge gibt, können diese in der Zukunft besser eingearbeitet werden.

Kommt Ihnen bei der Vorstellung dieser Schritte die Idee von DevOps in den Sinn, geht es Ihnen wie den Autoren. Mit dem Ansatz von DevOps wird versucht, die Lücke zwischen dem Betrieb und der Entwicklung zu schließen. Software soll innovativ sein, Probleme lösen und vor allem sollen aktuelle Versionen zeitnah bereitstehen und möglichst fehlerfrei funktionieren. Dafür ist die IT zuständig. Eine Trennung in Betrieb und Entwicklung ergibt für Außenstehende keinen Sinn. Diese Aufspaltung hat historische Gründe und führte zu unterschiedlichen Prioritäten der Bereiche. Demnach ist der Entwicklungsbereich stark daran interessiert, die implementierten Features möglichst kurzfristig an die Nutzer der Software auszuliefern. Dem IT-Betrieb ist dagegen vorrangig an einem kontinuierlichen und sicheren Betrieb der gesamten IT-Landschaft gelegen. Die Übernahme von Softwareupdates stellt in diesem Sinne einen Eingriff in das laufende System dar und kann dessen Stabilität bedrohen. Statt kurzfristiger und häufiger Releases werden lange Laufzeiten und umfassend geplante Updates bevorzugt. Dieses Silodenken ist nicht mehr zeitgemäß und führt nicht zu den erwünschten Zielen. Um den aktuellen Anforderungen gerecht zu werden, sind Agilität und Geschwindigkeit notwendig statt die Klärung von Zuständigkeiten. Der Ansatz von DevOps hebt diese Trennung der Bereiche auf und stellt sie unter eine gemeinsame Verantwortung. Das macht die Entwicklung und Auslieferung von Anwendungen flexibler und schneller.

Mit anderen Worten ist DevOps die Übertragung der umfassenden Kundenorientierung von der Analyse bis hin zur Bereitstellung und Wartung einer Software. Der Kern dieses Vorhabens – die eigentliche Entwicklung – ist damit Bestandteil dieses End-to-End-Prozesses [2] [3]. Wir finden, dass die DevOps-Schleife (**Abb. 1**) das ganz gut ausdrückt. Die einzelnen Vorgänge sind zusammengehörig und müssen auch als gemeinsamer Prozess verstanden werden. Ein Beispiel: Wir können nicht so tun, als ob mit der Entwicklung einer serverseitigen Webapplikation unser Job erledigt wäre. Wir müssen hier früher ansetzen und Kundenbedürfnisse erforschen, sie in die laufende Entwicklung einbeziehen (Feedback einholen, Prototypen evaluieren), die Software bereitstellen (das Deployment in seiner Arbeitsumgebung vornehmen) und für einen nachhaltige Verwendung des Produkts sorgen (Wartung, Anpassung).

Webtechnologien im Vorteil

Im Folgenden gehen wir davon aus, dass sich unser Kunde aus der Auswahl der zur Verfügung stehenden Typen von Applikationen, die üblicherweise im Businessumfeld anfallen, für eine Webapplikation entschieden hat. In vielen Fällen dürften die Argumente für eine Web-App gegenüber der Alternative einer Desktopanwendung überwiegen. Einige Argumente haben wir dazu im Kasten „Web-App meist im Vorteil“ als Entscheidungshilfe zusammengestellt.

Soweit die Diskussion zu den Vorteilen einer Webapplikation. Die nächste Ebene betrifft die Unterscheidung zwischen client- und serverseitigen Applikationen.

Client- vs. serverseitiges Scripting

Im Kundengespräch kommt auch die angewendete Technologie zur Sprache. Bei Webapplikationen ist es notwendig, gegenüber dem Kunden die technologischen Entscheidungen gut zu begründen. Gelegentlich hört man: „Eine PHP-Anwendung – ist das nicht überholt, gibt es nicht neuere Technologien?“ Dann werden Begriffe und Namen moderner, JavaScript-basierter Web-Frameworks aufgezählt. Aufklärung tut Not. Die Auswahl der passenden Technologie wird zwar immer ein Stück weit durch die eigenen Kenntnisse und Erfahrungen mitbestimmt, dennoch sollte der Blick zu Beginn des Projekts noch möglichst frei sein. Client- und serverseitige Technologien unterscheiden sich grundsätzlich und wir können sie nur bedingt miteinander vergleichen. Verdeutlichen wir uns (wiederholend) die Basics, die wir auch im Kundengespräch versuchen, zu erläutern. Bei der internen Diskussion befreit eine solche Vorgehensweise von Hypes und Diskussion der folgenden Form: „Wir programmieren in Angular oder React …. nicht mehr in PHP“.

Eine Webapplikation ist eine besondere Ausprägung der Client-Server-Architektur. Die Präsentation erfolgt durch den Client, in diesem Fall den Webbrowser. Der Browser selbst kann nur HTML (Struktur) und CSS (Layout) verarbeiten. Für die Logik im Browser ist die Sprache JavaScript zuständig. Weitere Technologien können bei Bedarf über Plug-ins nachgerüstet werden. Das ist jedoch nicht wünschenswert, damit die Applikation direkt und ohne Barrieren ausgeführt werden kann. Auf Seiten des Servers kann die dafür notwendige Infrastruktur schon umfassender aussehen und wird üblicherweise auch in weitere Teilkomponenten gegliedert. Die Kommunikation wird auf Serverseite durch den HTTP-Server abgewickelt. Dieser Teil der Serverstruktur stellt die Präsentationschicht serverseitig dar. Er liefert im Ergebnis den HTML-Code für den Webbrowser. Anwendungssysteme teilen die Verantwortung auf Serverseite neben den HTTP-Server noch auf einen Anwendungs- und einen Datenbankserver auf. Eine Basisarchitektur für eine Webapplikation ist in **Abbildung 2** dargestellt.

Sämtliche Kommunikation zwischen Client und Server erfolgt über das Web. Das Web basiert auf der Infrastruktur des Internets. Dabei sind zwei Bausteine wesentlich:

- *Protokoll*: Die Datenübertragung basiert auf der Anwendung des Transmission Control Protocol/Internet Protocol (TCP/IP).
- *Adressierung*: Die Adressen werden als Uniform Resource Locator (URL) angegeben.

Web-App meist im Vorteil

Bei der Auswahl der passenden Technologie für den Kunden sprechen folgende Argumente für eine Webapplikation:

- *Vereinfachte Installation:* Es ist nur eine Installation auf dem Server erforderlich. Der Client greift über das Netzwerk mit Hilfe eines Browsers auf die Anwendung zu. Lokal auf dem Client sind keine Installationen notwendig.
- *Updates:* Updates auf einer großen Anzahl von unabhängigen Clients (Desktopapplikationen) vorzunehmen, ist aufwendig und fehleranfällig. Diese Notwendigkeit entfällt jetzt. Webanwendungen sind aus Sicht des Anwenders stets aktuell. Darum muss sich der Benutzer nicht kümmern, sie werden direkt über den Server installiert.
- *Mehrere Nutzer:* Webapplikationen können auf mehreren Clients parallel genutzt werden. Über eine Nutzeranmeldung wird für eine korrekte Zuordnung der Datenverarbeitung gesorgt. Dadurch, dass die Daten stets auf dem Server gespeichert werden, ist das Problem der lokalen Speicherung der Daten hier nicht gegeben.
- *Datenspeicherung:* Eine lokale Speicherung der Daten ist meist weder gewünscht und sinnvoll (Datensicherheit, -schutz, -zugriff). Bei Desktopapplikationen wird oft dagegen verstoßen (unzählige Dokumente von Office werden auf der lokalen Festplatte gespeichert…). Webapplikationen speichern die Daten i. d. R. direkt auf dem Datenbankserver.
- *Systemanforderungen:* Die Applikation läuft auf dem Server, was bedeutet, dass die Daten- oder Internetverbindung für eine performante Ausführung wichtiger ist als die Ressourcen. Man kann Webanwendungen auch auf weniger leistungsfähiger Hardware ausführen.
- *Plattform- und Geräteunabhängigkeit:* Man kann i. d. R. von jedem System mit nahezu jedem beliebigen Browser arbeiten.
- *Kosten:* Webapplikationen sind bezüglich der langfristigen Wartungskosten meist günstiger als vergleichbare andere Applikationsarten. Auch die niedrigeren Anforderungen an Hardware können sich hier positiv auswirken.

Natürlich gibt es auch Einschränkungen von Web-Apps gegenüber dem Desktopapplikationen. Diese nutzen die Hardware besser aus; sind für grafikintensive Anwendungen, beispielsweise CAD, besser geeignet und erlauben einen weitgehenden Zugriff auf die interne und über Schnittstellen angeschlossene Hardware (Maschinensteuerung).

BASTA!

HYBRID-EDITION

BIS KONFERENZ-BEGINN
- ✓ 111-C64-Mini-Aktion
- ✓ 5-Tages-Special
- ✓ Kollegenrabatt

Die Konferenz für .NET, Windows & Web Development

15. – 19. Februar 2021
Expo: 16. – 18. Februar2021

Frankfurt Marriott Hotel oder Online

DIE ZUKUNFT ENTWICKELN

Starke Kompetenz in C#, .NET, Cloud- und Webtechnologien machen die BASTA! seit mehr als 20 Jahren zur führenden Konferenz für Microsoft-Technologien, Web und mehr.

OFFENE INNOVATION ERLEBEN

Neue Lösungen für zukünftige Herausforderungen – mit unseren praxisbezogenen Keynotes, Sessions und Power Workshops sind Sie immer am Puls der Microsoft-Zeit.

KERNKOMPETENZEN OPTIMIEREN

Die Vielfalt der Microsoft-Welt auf einer Konferenz: Ob Cloud oder Server, Agile oder DevOps – die BASTA! vermittelt das Wissen, das Sie als .NET-Entwickler weiterbringt!

HIGHLIGHTS

- **70+** Sessions, Workshops und Keynotes
- **50+** renommierte Topspeaker und Experten
- **7+** themenspezifische Special Days
- 2 Tage Hands-on **Power Workshops**
- Große **begleitende Expo** der Microsoft-Szene
- **Vollverpflegung** während der Konferenz
- Offizielles **Zertifikat** zur Konferenzteilnahme
- **Goodies:** Taschen, Freiabos, Gratismagazine u. v. m.

bastacon **basta.net**

Präsentiert von: windows .developer Veranstalter: S&S Media Group

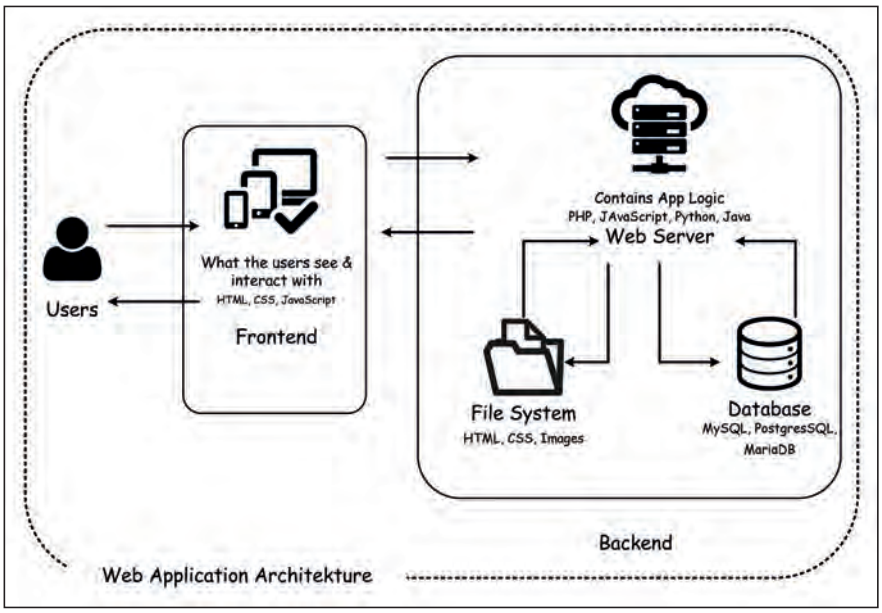


Abb. 2: Architektur einer Webapplikation (nach einer Idee von [5])

Es gibt unterschiedliche Technologien für die Umsetzung von webbasierten Anwendungssystemen. Insbesondere kann danach unterschieden werden, welche Aufgaben vom Client (Browser) bzw. vom Server wahrgenommen werden. Wir unterscheiden zwischen:

- *Serverseitiges Rendering (SSR)*: Wenn Sie eine Website besuchen, indem Sie auf einen URL klicken, wird eine Anforderung an den Server gesendet. Nachdem die Verarbeitung abgeschlossen ist, empfängt der Browser die Dateien (HTML, CSS und JavaScript) und damit den Inhalt der Seite und rendert sie. Wenn der Benutzer beschließt, zu einer anderen Seite zu wechseln bzw. Daten geändert werden, wird erneut eine Anfrage gestellt.
- *Clientseitiges Rendering (CSR)*: Der Hauptunterschied zwischen serverseitigem und clientseitigem Rendering besteht darin, dass nach dem Besuch einer Webseite, das clientseitige Rendering verwendet wird, d. h., dass nur eine einzige Anforderung an den Server gestellt wird, um das Hauptskelett der App zu laden. Der Inhalt wird dann dynamisch mit JavaScript (im Browser) generiert.

Eine Gegenüberstellung der Vor- und Nachteile findet sich in Tabelle 1.

	Server-side Rendering	Client-side Rendering
Vorteile	Webseiten sind einfach zu crawlen, was eine bessere Suchmaschinenoptimierung (SEO) bedeutet; das initiale Laden der Seite ist schneller; ideal für statische Websites, auf denen man keinen dynamischen Inhalt hat	Schnelle, direkte Interaktionen auf der Seite; nach dem ersten Laden reagiert die Seite sehr schnell
Nachteile	Häufige Serveranforderungen; Seite wird langsam gerendert; vollständige Seite wird bei der Anforderung neu geladen	Niedrige SEO, wenn nicht richtig implementiert; die anfängliche Performance ist möglicherweise zu langsam, da in den meisten Fällen externen Bibliotheken geladen werden

Tabelle 1: Server- vs. Client-side Rendering [5]

Eine serverseitige Verarbeitung eignet sich für die folgenden Anwendungsfälle [6]:

- *Datenbanktransaktionen*: das Erstellen, Lesen, Aktualisieren und Löschen von Einträgen
- *Geldtransaktionen*: für Zahlungs- und Kreditkartenprozesse
- *Erledigen von Benutzerabfragen*: Suchen nach bestimmten Informationen, zum Beispiel in Dokumenten oder Dateien auf dem Server
- *Serverübergreifende Dienstanforderungen*: Abruf von weiteren Informationen von einem anderen Server, typisch für eine E-Commerce-Webseite

Serverseitige Skripts (Backend-Prozesse) sind für den Nutzer nicht einsehbar und damit eine Voraussetzung, um Transaktionen sicher (Datenschutz) abzuwickeln. Vertrauliche Daten dürfen nicht auf Clientseite verarbeitet werden, da man als Nutzer hier einen direkten Zugriff auf diese Informationen hat (der Quellcode kann jederzeit eingesehen werden). Welche Programmiersprachen kommen auf dem Server zum Einsatz? Ruby, Python, Java und selbstverständlich PHP.

Wechseln wir auf die Seite des Clients, d. h., dass die Verarbeitung der Skripte im Browser erfolgt. Eine clientseitige Verarbeitung bietet sich daher für die folgenden Anwendungen an:

- Anzeige der Daten auf der Benutzeroberfläche und Präsentation der Informationen, zum Beispiel in Form von dynamischen Grafiken
- Akzeptieren und Verarbeiten von Benutzereingaben
- Kommunikation mit dem Server, Senden von Daten für die Backend-Verarbeitung

Im Allgemeinen werden clientseitige Skripts verwendet, um die Benutzereingaben, die visuellen Aspekte und nicht kritische Transaktionen zu behandeln. Die clientseitigen Skripts werden heruntergeladen und die Benutzer haben Zugriff auf alle Daten. Bei einer Interaktion erfolgt eine direkte Manipulation des DOM der Webseite. Zur Vereinfachung der clientseitigen Programmierung steht eine Reihe von Klassenbibliotheken und Frameworks zur Verfügung. Klassenbibliotheken bieten dabei Methoden und Funktionen zur universellen Nutzung in der eigenen Applikation an. Frameworks stellen dem Entwickler ein Anwendungsgerüst bereit, das von der eigenen App genutzt wird. Auf diese Weise ist es möglich, Webapplikationen zu erstellen, die eine ähnlich

hohe Performance und ein gutes Benutzerfeeling bieten, wie man es von Desktopapplikationen gewohnt ist. Wie gesagt, wir haben hier nur die Clientseite betrachtet und die o. g. Aufgaben des Servers ignoriert. Aus Sicht des Entwicklers wird mit JavaScript gearbeitet (etwas anderes versteht der Browser nicht). Alternativ kann auch mit TypeScript während der Entwicklung programmiert werden. Hier profitiert man dann zum Beispiel von den Vorteilen einer typsicheren Sprache und einer bessern Umsetzung der Objektorientierung. Letztendlich wird dennoch TypeScript wieder in JavaScript übersetzt. Diese Aufgabe erledigt i. d. R. ein Transpiler.

PHP-Webapplikation und SPA

Bei PHP-basierten serverseitigen Webapplikationen haben wir einen Aufbau, wie er in **Abbildung 3** dargestellt ist.

Der Ablauf ist wie folgt: Der Nutzer sendet eine Anfrage über den Browser an den Server. Diese Anfrage wird durch das PHP-Programm verarbeitet. Dazu kann es erforderlich sein, dass das PHP-Programm Daten aus einer Datenbank abruft. Die Datenbank ist ebenfalls auf dem Webserver installiert bzw. von ihm erreichbar. Der PHP-Interpreter erzeugt auf der Basis der Anfrage und seines spezifischen Algorithmus nunmehr die Antwort in Form einer HTML-Datei. Diese HTML-Datei wird an den Client übermittelt. Der Browser selbst bekommt also stets eine HTML-Datei als Ergebnis geliefert und ist nur für deren Anzeige verantwortlich.

Wir können hier bereits ein Zwischenfazit ziehen: Webapplikationen mit einem großen Bedarf an in sich abgeschlossenen Transaktionen mit dem Server abzuwickeln, zum Beispiel Abfrage von Kunden- oder Artikellisten usw., sind mit dem Szenario einer serverseitigen Webapplikation gut abgedeckt. Die Sicherheit der Daten steht hier an oberster Stelle, eine direkte Interaktion auf der Seite des Clients (interaktive Grafiken, Ad-hoc-Berechnungen, Drag-and-Drop-Operationen) ist in diesen Fällen auch nicht notwendig. Bestehen jedoch genau diese Anforderungen, muss man aus heutiger technischer Perspektive den Code für das User Interface (UI) vom Backend auf den Client verlagern – man kommt zum Ansatz der Single-Page-Applikation (SPA).

SPA sorgt für Reaktionsfähigkeit und Interaktivität im Client

Hier zeichnen wir den Weg von der serverseitigen Webapplikation zur SPA anhand von **Abbildung 4** nach [8]. Den linken Teil der Abbildung hatten wir bereits am Beispiel einer PHP-Applikation erläutert. Wichtig: Im Client erfolgt nur die Darstellung des User Interface, die dazu notwendigen Aufgaben der UI-Logik werden komplett vom Server erledigt. Dazu gehört zum Beispiel auch die Validierung der Daten.

Teilt man die Aufgaben der UI-Logik auf Client und Server auf (mittlerer Teil, **Abb. 4**), kann man einige Aufgaben direkt im Browser erledigen, ohne die Seite komplett neu zu laden. Das Problem an diesem Vorgehen ist, das es kaum gelingt, eine saubere Architektur aufzusetzen. Welche Aufgaben werden auf Clientseite, welche durch den Server erledigt? Wann werden nur einzelne Daten mittels JavaScript aktualisiert, wann wird die komplette Seite neu geladen? Der Schritt zur SPA besteht dann darin (rechter Teil, **Abb. 4**), dass alle Aufgaben im Zusammenhang mit dem UI (Darstellung und Logik) auf den Client verlagert werden. Die Art der Datenkommunikation wechselt nunmehr auch, d. h., dass der Server nun aufgrund der Anfrage einen JSON-Datenstrom liefert, der clientseitig mittels JavaScript durch den Parser des Browsers verarbeitet und zur Anzeige gebracht wird. Typische Bibliotheken und Frameworks auf Clientseite sind zum Beispiel Vue.js, React und Angular.

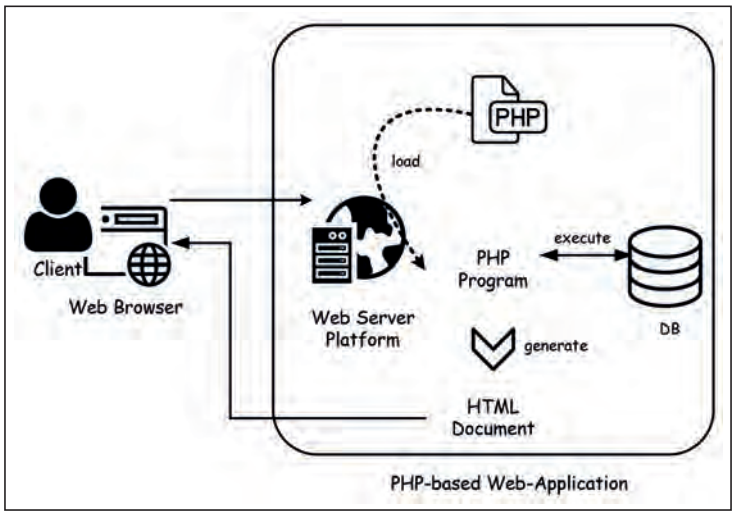


Abb. 3: Architektur einer PHP-basierten Webapplikation [7]

Technologieauswahl

Die Auswahl der passenden Technologie bei Webapplikationen ist daher weniger getrieben von Hypes und Kenntnissen der Sprachen (JavaScript, PHP), sondern orientiert sich eindeutig am Nutzerfokus. Mit diesem Wissen können wir unsere Kunden beraten und ihnen aus heutiger Sicht die passende Technologie empfehlen. Im Enterprise-Bereich laufen viele Anforderungen weiterhin auf klassische Datenbankapplikationen hinaus. Eine unmittelbare Bearbeitung der Daten auf Clientseite ist hier nicht immer notwendig. Hier ist eine PHP-Applikation nach „klassischem“ Architekturmuster weiterhin ein guter Ansatz. Wir folgen also dem Ansatz: „Keep it simple and stupid (KISS)“ gleichgültig, ob das „hip“ ist oder nicht. Andererseits gibt es eben auch eine Reihe von Anwendungszwecken, bei denen die Interaktion im Client notwendig ist. Ein Dashboard für einen Business Use Case mit Drag-and-Drop-Funktionen und direkter Nutzerbearbeitung kann man nicht über eine PHP-Anwendung realisieren. Hier ist der beschriebene SPA-Ansatz der richtige Weg. PHP und JavaScript sind daher zwei wichtige Sprachen des Webs, die sich ergänzen und einen unterschiedlichen Fokus aufweisen (**Abb. 5**).

Daraus können wir für die Praxis schlussfolgern: Die Entscheidung für eine Web-App ist nur der erste Schritt. Es folgt die Technologieauswahl, die erst dann seriös ge-

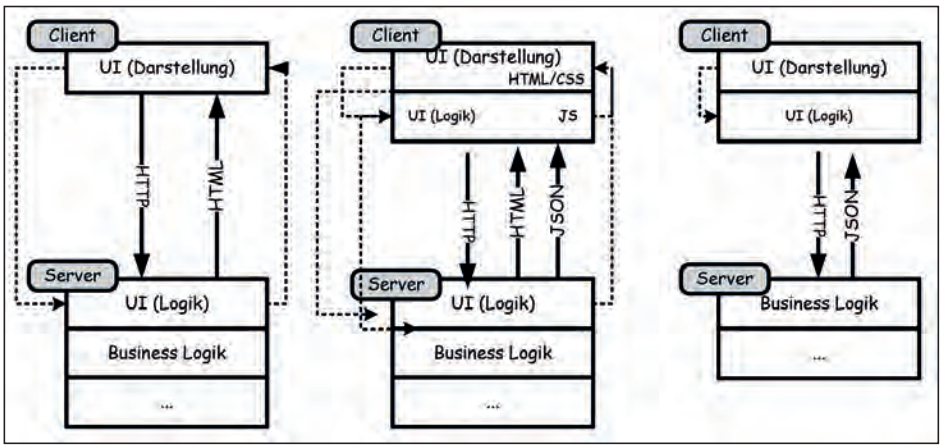


Abb. 4: Der Weg zur SPA (Darstellung nach [8])

lingt, wenn man die genauen Nutzeranforderungen kennt. Einige Szenarien lassen sich primär sowohl mit clientseitiger als auch serverseitiger Programmierung umsetzen. Dabei sind noch die folgenden Punkte zu berücksichtigen [5]:

- *Sicherheit*: Der Schutz von Daten rückt immer mehr in den Fokus. Die Wahl der Softwarearchitektur trägt dazu wesentlich bei.
- *Skalierbarkeit*: Die Leistungsfähigkeit der Anwendung muss einer umfassenderen Nutzung entsprechend einfach skalieren können.
- *Modularität*: Wartung und Weiterentwicklung funktionieren dauerhaft nur, wenn die Architektur möglichst in in sich geschlossene Komponenten aufgeteilt wird und die Softwarearchitektur einer klaren Logik folgt.

Abb. 5: Features von JavaScript und PHP [9]

Features	Java-Script	PHP
Developed by	Brendan Eich (1995)	Rasmus Lerdorf (1994)
Object-oriented	Yes	Yes
Easy to use existing code	Yes	Yes
Server side scripting language	No	Yes
Client side scripting language	Yes	No
Accepts both upper case and lower case boolean variable	No	Yes
Case sensitive to variables	Yes	Yes
Case sensitive in function	Yes	No
Objects & Arrays interchangeable	Yes	No
Requires HTTP to execute	Yes	Yes
Updates files on server	No	Yes
Execute with browser window	Yes	No
Supports framework	Yes	Yes
Platform Independent	Yes	Yes
Open Source	Yes	Yes
Support database	No	Yes
Memory Management (garbage collection)	Yes	Yes
Library	Yes	Yes
Exceptional Handling	Yes	Yes
Performance	Fast	Slow
Support of features	Less	More

Die eigenen Fähigkeiten spielen jedoch auch eine Rolle. Alle Facetten der modernen Webprogrammierung lassen sich unmöglich beherrschen. Die gezeigten Grundlagen zeigen uns aber schnell, welches Projekt man selbst bearbeiten kann und welches lieber dem Kollegen überlassen werden sollte. Wir sollten der Logik: „Wer als Werkzeug nur einen Hammer hat, sieht in jedem Problem einen Nagel“ nicht folgen.

Fazit und Ausblick

In diesem Artikel ging es um die grundlegenden technologischen Entscheidungen, die bei der Entwicklung einer neuen Webapplikation bei jedem Projekt anstehen. Ausgehend vom Kundennutzen müssen wir die heute passende und aktuelle Technologie auswählen. Bereitet man die technischen Aspekte sauber auf, kann man seine Entscheidungen auch gegenüber dem Management, den Produktverantwortlichen und Kunden argumentieren und vertreten. Im zweiten Teil der Serie steigen wir intensiver in die Welt der PHP-Applikationen ein. Im Fokus werden die Architektur und die Struktur stehen.

Dr. Veikko Krypczyk ist begeisterter Entwickler und Fachautor.

Elena Bochkor arbeitet primär am Entwurf und Design mobiler Anwendungen und Webseiten.

Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter http://larinet.com.

Links & Literatur

[1] https://codemotion.ninja/end2end-development/

[2] https://www.informatik-aktuell.de/entwicklung/methoden/das-dasa-devops-prinzip-3-end-to-end-responsibility.html

[3] https://www.eggs.de/de/blog/DevOps-best-practices.html

[4] https://www.qentelli.com/thought-leadership/insights/your-devops-loop-broken-solving-continuous-feedback-puzzle

[5] https://reinvently.com/blog/fundamentals-web-application-architecture/

[6] https://code-boxx.com/server-side-vs-client-side/

[7] Nakajima, Shin: „An Architecture of Dynamically Adaptive PHP-based Web Applications“, in: Computer Science, 2011, 18th Asia-Pacific Software Engineering Conference, https://www.semanticscholar.org/paper/An-Architecture-of-Dynamically-Adaptive-PHP-based-Nakajima/f6cfb2dcaf82cc228af32132ef840dbe2779e2c6

[8] https://nilshartmann.net/posts/microfrontends/

[9] https://www.guru99.com/php-vs-javascript.html

Produktdatenaustausch mit Hilfe der Pimcore-Schnittstelle PIM

Shopware 6 meets Pimcore 6

Heute gibt es unzählige Tools am Markt, die in den verschiedensten Bereichen Lösungen anbieten. Die Herausforderung besteht gegenwärtig darin, diese Tools miteinander interagieren zu lassen, um ein redundantes Pflegen von Daten oder eine manuelle Datenübertragung zu vermeiden.

von Timo Trautmann

Im E-Commerce-Bereich hat sich in den letzten Jahren insbesondere Shopware [1] einen Namen sowohl im Geschäft mit Privatkunden (B2C) als auch Geschäftskunden (B2B) gemacht. Pimcore [2] ist eine gute Lösung, wenn es um die Verwaltung von individuellen Produktdatenstrukturen geht. In Pimcore kann man sehr einfach Produktdatenstrukturen entwerfen, die man mittels eines View- und Rechtesystems individualisieren und steuern kann. Beide Systeme sind auf Basis von PHP 7 implementiert und nutzen das Framework Symfony als Basis. Somit findet jeder Entwickler mit Symfony-Kenntnissen einen schnellen Einstieg in beide Systeme. Die in Pimcore gepflegten Produktdaten möchte man zum Verkauf der Produkte nicht noch einmal in einem Shopsystem wie Shopware redundant pflegen. Daher gibt es eine von scope01 implementierte Schnittstelle [3] zur Übertragung von Produktdaten zwischen den beiden Systemen, die dieser Artikel näher beschreiben wird.

Produktdatenaustausch mit Hilfe von REST

Abbildung 1 zeigt vereinfacht die Kommunikation zwischen Shopware und Pimcore mit Hilfe des in Shopware 6 neu eingeführten Sync API [4]. Shopware hat dabei in der 6er-Version die bestehende REST-Schnittstelle komplett überarbeitet und verfolgt damit den „API first"-Ansatz [5]. Das Sync API ist Teil der neuen REST-Schnittstelle, mit der man Datenobjekte in Shopware aktualisieren kann. Ein Produkt, so einfach es sich erst einmal anhören mag, besteht nicht nur aus einem einfachen Datenobjekt, sondern aus vielen einzelnen Datenobjekten, wie zum Beispiel Medienobjekten oder Übersetzungen zum Produkt. Das Sync API ermöglicht es dabei, Objekte verschiedenster Art mit nur einem REST-API-Aufruf an Shopware zu übermitteln, um sie dort neu anzulegen oder zu aktualisieren.

Das PIM-Plug-in selbst bietet zur Realisierung dieser Funktion zwei eigenständige Plug-ins an. Das Plug-in ScopPimBundle wird dabei auf Pimcore-Seite installiert, während das Plug-in ScopPimPlugin auf Shopware-Seite installiert wird. ScopPimPlugin erweitert das Shopware REST API um weitere individuelle Funktionen,

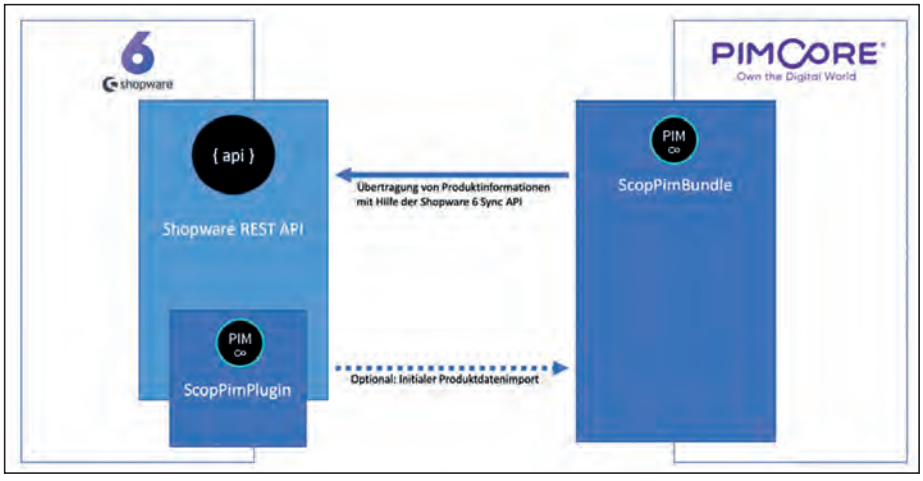


Abb. 1: Kommunikation zwischen Shopware und Pimcore via REST

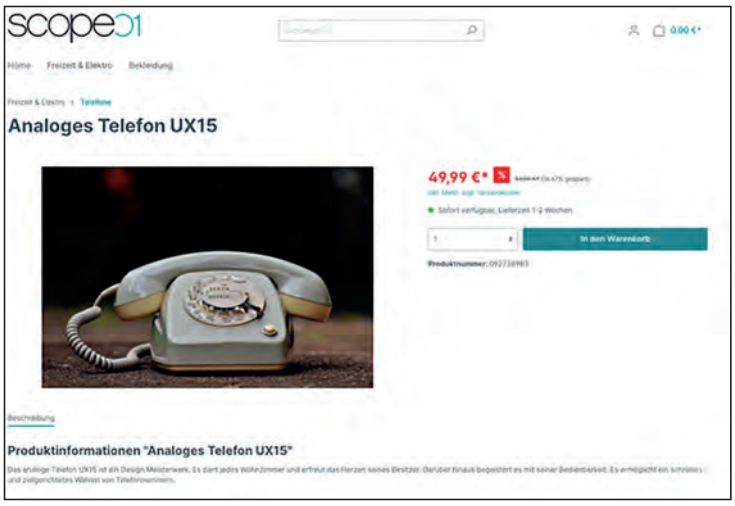


Abb. 2: Das Produkt im Shopware Frontend

die zum Datenaustausch benötigt werden, die Shopware im Standard nicht anbietet. Hierbei erhält das API eine Funktion, mit der seine Erreichbarkeit einfach getestet werden kann. Ein weiteres Beispiel für eine neue API-Funktion vereinfacht die Abfrage von vorhandenen Übersetzungen zu Datenobjekten.

Das deutlich umfangreichere Plug-in ist ScopPim-Bundle. Es stellt in Pimcore die im Standard von Shopware enthaltenen Datenstrukturen von Produkten und allen sie umgebenden Objekten wie Kategorien oder Medien bereit. Zudem enthält das Plug-in alle Commands und Cronjobs, die zum aktiven Datenaustausch notwendig sind.

Abbildung 1 zeigt den optionalen Weg zur Übertragung von Produktdaten nach Pimcore: Dieser Weg ist als initiale Einrichtung für alle Nutzer gedacht, die bereits ihre Produktdaten in Shopware pflegen und nun ein PIM-System einführen möchten. Mit Hilfe eines initialen Datenimports können so alle Standardproduktdaten aus Shopware in Pimcore importiert werden, um ab dann dort die Produkte weiter zu pflegen.

Fallstudie: Übertragung eines Produkts mit dem Sync API

Das Produkt „Analoges Telefon UX15“ in Shopware ist in **Abbildung 2** zu sehen. Es wurde zuvor in Pimcore gepflegt und anschließend in Shopware übertragen. **Abbildung 3** zeigt den Admin von Pimcore am Beispiel der Pflege des gleichen Produkts.

Die Darstellung der Datenobjekte in Pimcore kann mit unterschiedlichsten Eingabeobjekten und Formaten erfolgen. Da ein Produkt in diesem Beispiel sehr viele Datenfelder hat, werden sie zunächst wie im Bild gezeigt in verschiedenen Tabs angezeigt. Je nach Typ des jeweiligen Eingabefeldes wird es als Checkbox, Eingabefeld oder referenziertes Feld angezeigt. In unserem Beispiel sind Hersteller (Manufacturer) und Kategorien (Categories) referenzierte Felder, die auf andere Datenobjekte in Pimcore verweisen. In Pimcore werden übersetzbare und nicht übersetzbare Felder unterschieden. In **Abbildung 3** ist ersichtlich, dass die Felder Artikelname (Article name) und Verpackungseinheit (Packaging unit) übersetzbar sind, während die anderen Felder des Reiters Allgemein (General) nicht übersetzbar sind.

Auf der linken Seite des Pimcore Admin befindet sich die Navigation, unterteilt in verschiedene Tabs. Hierdurch kann man hierarchisch durch die Objekte in Pimcore navigieren und hat mit Hilfe des Tabsystems noch einmal eine weitere Ebene zur Untergliederung. Durch die Custom Views und Custom Layouts [6] kann die Oberfläche des Admin auf Basis der verwendeten Datenstrukturen individualisiert werden, um so jedem Admin-Benutzer die für ihn optimale Darstellung auf die gespeicherten Daten zu ermöglichen. Diese Individualisierbarkeit ist eine Stärke von Pimcore. Über die Rechteverwaltung kann man dann diese individuell konfigurierten Ansichten verschiedenen Rollen zuordnen, um so verschiedenen Abteilungen des Unternehmens die für sie jeweils optimale Darstellung auf die Daten zu ermöglichen. Über diesen Weg lassen sich ebenfalls die Zugriffe auf verschiedene Datenfelder steuern.

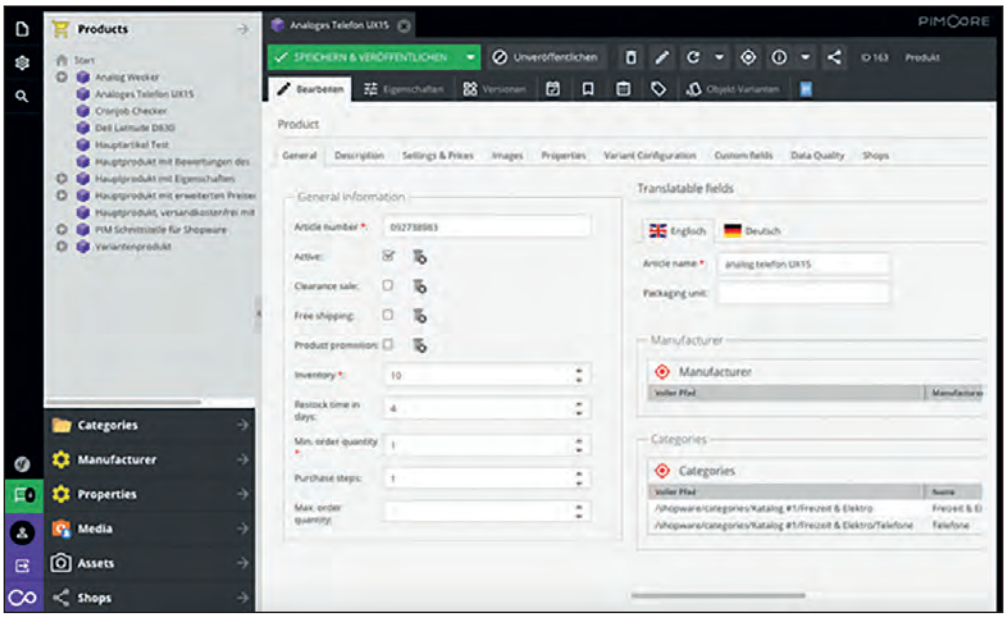


Abb. 3: Das Produkt in Pimcore

REST Call zur Übertragung des Produkts „Analoges Telefon UX15“

In **Abbildung 4** wird das Beispielprodukt via REST Call an das Sync API von Shopware übertragen. Die Darstellung des Aufrufs ist hierbei stark vereinfacht und zeigt nur ei-

```
POST /api/v1/_action/sync
{
  "write-product_translation": {
    "entity": "product_translation",
    "action": "upsert",
    "payload": [
      {
        "productId": "ee4e4e4101f54674804f6976cd1a5685",
        "languageId": "2fbb5fe2e29a4d70aa5854ce7ce3e20b",
        "name": "analog telefon UX15",
        "description": "<p>Das analoge Telefon UX15 ist ein Design...</p>"
      },
      {
        "productId": "ee4e4e4101f54674804f6976cd1a5685",
        "languageId": "0d484753ba524afaa17d54310b94b20e",
        "name": "nalog telefon UX15",
        "description": "<p>The UX15 analog phone is a design masterpiece...</p>"
      }
    ]
  },
  "write-product": {
    "entity": "product",
    "action": "upsert",
    "payload": [
      {
        "id": "ee4e4e4101f54674804f6976cd1a5685",
        "active": 1,
        "productNumber": "092738983",
        "ean": "19287321",
        "categories": [
          {
            "id": "251448b91bc742de85643f5fccd89051",
            "name": "Freizeit & Elektro"
          }
        ]
      }
    ]
  }
}
```

Abb. 4: Produktdatenübertragung mit dem Shopware Sync API

nen kleinen Bruchteil der Datenstrukturen, die im Rahmen des Produktupdates an Shopware übertragen werden. Auf erster Ebene des JSON-Datensatzes wird dabei die Operation definiert, die mit der übergebenen Payload geschehen soll. Unter der Payload versteht man hier einen strukturierten Datensatz, wie z. B. Produktdaten oder Übersetzungen, der zur Durchführung der Operation benötigt wird. In unserem Beispiel werden hier die zwei Objekte „Produkt“ und „Produktübersetzungen“ an Shopware übergeben. Mit Hilfe der Aktion *upsert* aktualisiert Shopware den betreffenden Datensatz oder legt ihn neu an, falls er noch nicht existiert. Um Datensätze zu löschen, kann die Aktion *delete* verwendet werden. Durch das Feld ENTITY wird Shopware mitgeteilt, um welchen Datensatz es sich konkret in der Datenbank handelt. Der Inhalt des Feldes PAYLOAD und dessen Struktur sind komplett individuell auf das übergebene Objekt zugeschnitten, das im Feld ENTITY definiert wurde.

In der Entität *product_translation* wird im in **Abbildung 4** gezeigten Beispiel die Übersetzung auf Deutsch und Englisch für den Produkttitel (*name*) und die Produktbeschreibung (*description*) übergeben. Als Identifier werden hierzu die ID des Produkts (*productId*) und die ID der Sprache (*languageId*) übergeben. In der Entität *product* werden denn die Produktstammdaten übergeben. Die zuvor übergebene *productId* findet sich zur Zuordnung hier im Feld *id* wieder. Zusätzlich werden noch ein Aktivkennzeichen (*active*), die Produktnummer, die das Unternehmen individuell vergibt (*productNumber*), die EAN und die Kategorien (*categories*) übergeben, denen das Produkt zugeordnet wird.

Datenmodellierung in Pimcore

Die Datenmodellierung, mit deren Hilfe man die Objekte zur späteren Speicherung der PIM-Daten modellieren kann, wird ebenfalls über den Pimcore Admin vorgenommen (**Abb. 5**). Hierzu kann man sich die einzelnen Elemen-

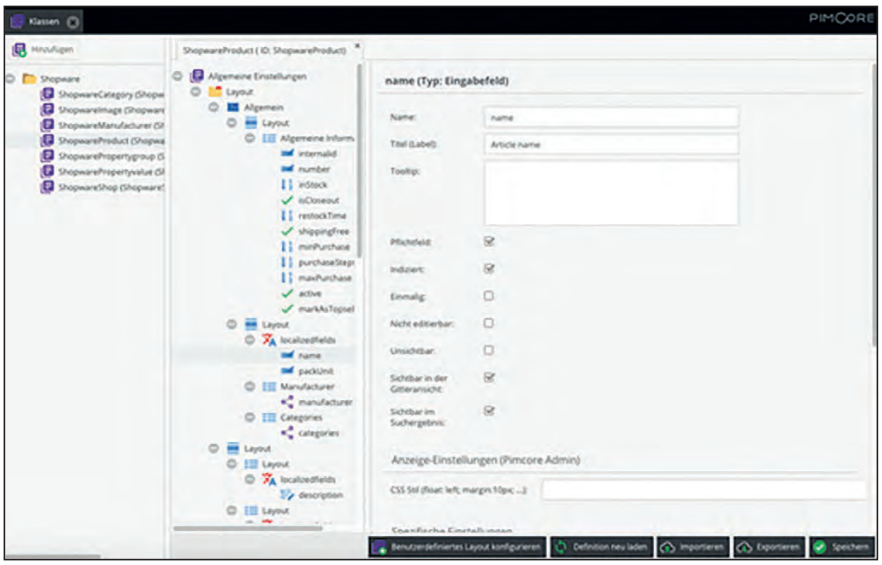


Abb. 5: Datenmodellierung im Pimcore Admin

te und Darstellungsoptionen, wie zum Beispiel Reiter oder Formulargruppen, dynamisch per Mausklick zusammenstellen.

Für jedes Eingabefeld wird anschließend ein Typ, wie zum Beispiel String, WYSIWYG oder Checkbox definiert. Jeder dieser Eingabetypen hat anschließend viele Optionen zur Konfiguration. So kann man definieren, ob das Feld ein Pflichtfeld sein soll, welchen Namen es haben soll oder in welchen Zusammenhängen es im Pimcore Admin angezeigt werden soll. Nicht zuletzt kann man mit weiterführenden Optionen auch noch das Aussehen anpassen oder komplexere Validierungsregeln integrieren. Diese Objektstrukturen werden dann auf dem Dateisystem und nicht in der Datenbank abgelegt. Das ist gerade für die Entwicklung immens wichtig, da so die Datenstrukturen in Git [7] versioniert werden können. Über den Konsolenbefehl *./bin/console pimcore:deployment:classes-rebuild* [8] kann man nach einem Deployment einer neuen Klassenstruktur

```php
<?php

return Pimcore\Model\DataObject\Fieldcollection\Definition::__set_state(array(
   'id' => 'ShopwareProduct',
   'name' => 'ShopwareProduct',
   'layoutDefinitions' =>
                  ClassDefinition\Layout\Panel::__set_state(array(
         'fieldtype' => 'panel',
         'labelWidth' => 100,
         'childs' =>
           array(
             0 =>
               ClassDefinition\Data\Input::__set_state(array(
                  'fieldtype' => 'input',
                  'width' => 190,
                  'defaultValue' => NULL,
                  'name' => 'number',
                  'title' => 'Article number'
               )),
             1 =>
               ClassDefinition\Data\Checkbox::__set_state(array(
                  'fieldtype' => 'checkbox',
                  'defaultValue' => 0,
                  'name' => 'active',
                  'title' => 'Active',
               )),
           ),
      )),
```

Abb. 6: Pimcore-Objektdefinition in PHP

dann einfach die aktuell in der Datenbank befindliche Version der Klassenstruktur aktualisieren. Hierbei können Felder ergänzt, entfernt oder auch komplett neue Objekte angelegt werden. Bei dem Update der Datenstrukturen auf Basis der vorhandenen Objektdefinitionen gehen dabei keine Daten verloren, es sei denn, es wurden manuell Felder entfernt. Das Datenmodell wird lediglich hinsichtlich seiner Struktur angepasst.

Abbildung 6 zeigt ein modelliertes Pimcore-Objekt in seiner Darstellung in PHP. Die Struktur wurde an dieser Stelle stark vereinfacht und viele Elemente und Konfigurationsdefinitionen wurden aus Gründen der Übersichtlichkeit weggelassen. Das Beispiel zeigt die Definition des Objekts *ShopwareProduct*, das die Basis für jedes Produkt darstellt. Auf der ersten Ebene wird ein Layout-Panel definiert, das zwei Datenfelder enthält: Das erste Datenfeld ist das Feld NUMBER. Hier kann im Admin die firmeninterne Artikelnummer des Produkts gepflegt werden. Das zweite Eingabeelement ist eine Checkbox ACTIVE. Diese Checkbox definiert, ob ein Produkt aktiv oder inaktiv ist. Der Defaultwert, der bei der Anlage eines neuen Produkts in Pimcore Anwendung findet, ist *0*, also inaktiv. Im gezeigten Beispiel werden den Elementen noch Layoutinformationen mitgegeben. So wird zum Beispiel die Breite der Felder festgelegt.

Erkennung der Datenobjekte zwischen Pimcore und Shopware

Ein Artikel hat in der Datenbank von Pimcore erst einmal eine andere ID als in der Datenbank von Shopware. Dennoch ist es essenziell, dass sich gleiche Datensätze zwischen beiden Systemen eindeutig identifizieren lassen, damit diese bei Bedarf bei Änderungen immer aktualisiert werden können. Hierzu hat jedes Shopware-Datenobjekt in Pimcore eine sogenannte Internal ID, in der eine eindeutige UUID gespeichert wird. Diese wird einmalig beim Erzeugen des Objekts in Pimcore generiert und ist auch nicht durch den Pimcore Admin editierbar. Diese ID wird stets zu Shopware übertragen und dort hinterlegt, damit die Datensätze eindeutig erkennbar sind. Seit Shopware 6 kommen UUIDs vom Typ Binary-16 zum Einsatz. Das ist für den Datenzugriff zwischen den beiden Systemen von Vorteil, da die Internal ID so direkt als Primärschlüssel auf Shopware-Seite fungieren kann. Hat beispielweise ein Produkt die Internal ID *28f7dec4be6349a6bafd1e10f863664a*, kann das Produkt auf Shopware-Seite einfach über den Admin URL */admin#/sw/product/detail/28f7dec4be6349a6bafd1e10f863664a/base* geöffnet werden. Im umgekehrten Fall kann man auf diese Weise einen Artikel in Shopware auch sehr schnell in Pimcore finden, indem man die UUID aus dem URL einfach schnell im Pimcore Admin sucht.

```
username@my-server.com:~/pimcore6.scope01.com$ php bin/console scop:export-sw-data

Start data export

Shop: Shopware 6(9) is reachable, start export...

Start export Category

Export 1 Category (0 - 1) from total 1. Done
[============================] 100% (1/1) Memory usage: 38.5 MiB

Start export Products

Export 2 Products (0 - 2) from total 2. Done

[============================] 100% (2/2) Memory usage: 40.5 MiB
```

Abb. 7: Ausführung des Symfony Export Commands

Datensynchronisierung mittels Symfony Command

Jedes für Shopware relevante Objekt in Pimcore hat einen Status, über den die Steuerung zur Übertragung zu Shopware vorgenommen werden kann. Das Feld heißt SYNC STATUS und kann über eine Select-Box verschiedene Werte annehmen. Soll das Objekt so beispielsweise nach Shopware synchronisiert werden, erhält es den Status *Ready for publishing*. Artikel, die dann durch den Symfony Command synchronisiert wurden, erhalten den Status *Published*. Vor der Synchronisierung erfolgt eine Prüfung der Felder in Pimcore. So wird beispielsweise geprüft, ob alle Felder eines Objekts gefüllt sind, die beispielsweise in Shopware ein Pflichtfeld für ein Objekt sind. Sind diese Felder nicht gefüllt, kann eine Übertragung nach Shopware nicht erfolgen, und das Feld SYNC STATUS wird bei der Übertragung auf den Wert *Error* gesetzt.

Der Symfony Command und dessen Ausgabe sind vereinfacht in **Abbildung 7** dargestellt. Dieser Befehl zum Export der Daten nach Shopware kann zum Test oder zu Debugging-Zwecken direkt von der Serverkonsole ausgeführt werden. Für die routinemäßige Verwendung empfiehlt sich die Einrichtung des Commands per Cronjob, der zum Beispiel minütlich ausgeführt wird. So lange keine Produkte im Pimcore Admin den Status *Ready for publishing* erhalten, führt der Command keine Aktion aus. Daher ist diese häufige Art der Ausführung unkritisch und hat den Vorteil, dass erfolgte Änderungen schnell nach Shopware synchronisiert werden.

Weitere Funktionen der Schnittstelle

In Shopware hat der Administrator die Möglichkeit, alle Datenbankobjekte, wie zum Beispiel Kategorien oder Produkte, um beliebige benutzerdefinierte Felder zu erweitern. Mittels dieser Custom Fields können die Shopobjekte an die Bedürfnisse des jeweiligen Unternehmens angepasst werden. So können zum Beispiel nur im Elektrohandel relevante Felder in den Produkten ergänzt werden. In Shopware gibt es hier analog zur Pimcore-Datenmodellierung die Möglichkeit, den Feldern einen Typ wie Textfeld oder Checkbox zuzuweisen. Diese Custom-Fields-Strukturen können analog auch in Pim-

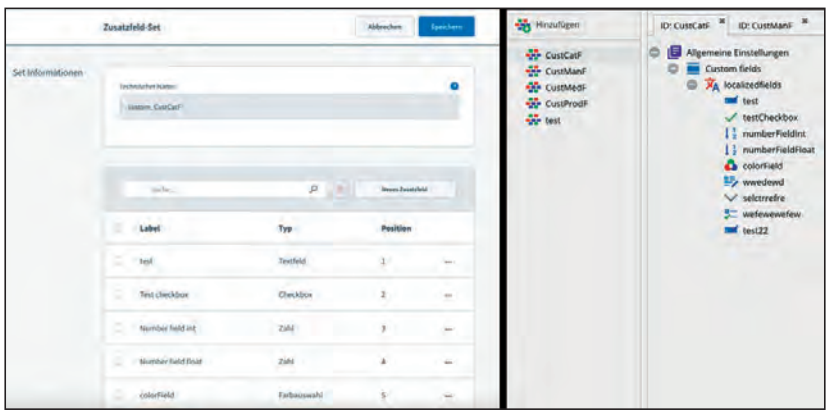


Abb. 8: Custom Fields zwischen Shopware und Pimcore

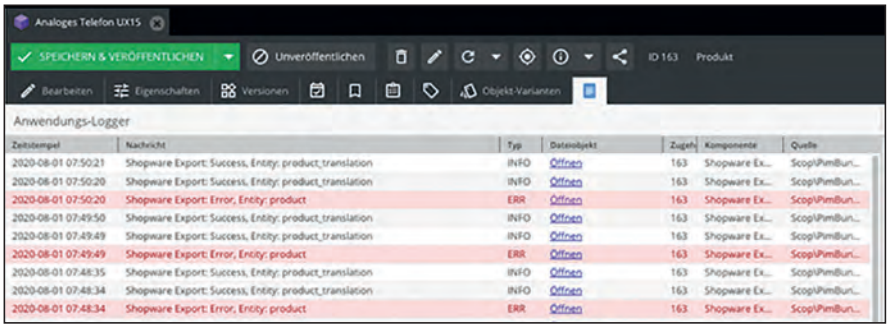


Abb. 9: Pimcore-Schnittstelle PIM Logs im Pimcore Application Logger

core modelliert werden. Dazu unterstützt die Pimcore-Schnittstelle PIM automatisiert die Verwaltung und Übertragung dieser Strukturen, ohne dass ein implementierungstechnischer Aufwand an der Stelle notwendig wäre. Sie werden in Pimcore mit Hilfe von Object Bricks [9] abgebildet. **Abbildung 8** zeigt auf der linken Seite ein Beispiel eines Custom-Fields-Blocks in Shopware, der zur Erweiterung des Kategorieobjekts angelegt wurde. Diese Modellierung ist auf der rechten Seite im Pimcore Admin zu sehen. Werden nun in Pimcore Daten in dieser Struktur abgelegt, werden sie automatisiert in die Custom-Fields-Strukturen von Shopware übertragen.

Der Application Logger [10] von Pimcore ist ein umfangreiches Entwicklertool zum Loggen von Vorgängen, Ereignissen und Fehlern in Pimcore. Die Pimcore-Schnittstelle PIM nutzt diese Funktion intensiv, um die Übertragung der Produktdaten von Pimcore nach Shopware zu loggen. So sieht man in den Logeinträgen beispielsweise auch, welche Felder bei der Übertragung nach Shopware in welcher Struktur übergeben wurden. Es ist somit im Fehlerfall ein praktisches Debugging-Tool für Entwickler, um die Fehlerursache schnell eingrenzen zu können. Den Application Logger gibt es sowohl auf Produktebene als auch im globalen Kontext im Pimcore Admin. So kann man gezielt nach einem Übertragungslog direkt am Objekt schauen oder alternativ sich auch das globale Log ansehen, um eventuelle globale Übertragungsprobleme zu erkennen. Nicht erfolgreiche Operationen werden dabei im Fehlerlog direkt rot hervorgehoben.

Die Pimcore-Schnittstelle PIM kann mit Hilfe von Events individualisiert werden. Hierdurch können kundenspezifische Anforderungen mit der Schnittstelle realisiert werden, ohne den Core Code anpassen zu müssen und die Updatefähigkeit zu gefährden. Durch diese Anpassungen kann man beispielsweise die Daten vor dem Export nach Shopware an weitere Bedürfnisse anpassen oder kundenspezifische Validierungsregeln implementieren.

Demosystem und Roadmap

Auf den Seiten https://shopware6.scope01.com und https://pimcore6.scope01.com findet sich eine aktuelle Version der Pimcore-Schnittstelle PIM [11], auf der alle im Artikel beschriebenen Funktionen live getestet und angesehen werden können. Hierdurch kann man einen ersten Einblick in die Pimcore-Schnittstelle PIM erhalten, ohne sich ein komplettes Set-up, bestehend aus Pimcore und Shopware, aufsetzen zu müssen.

In zukünftigen Versionen der Pimcore-Schnittstelle PIM wird insbesondere die Verknüpfung mit den Sales-Channel-Funktionen von Shopware weiter vertieft werden. So wird es eine Unterstützung für Sales-Channel-abhängige Canonical URLs auf Pimcore-Seite geben und die Artikel können den Sales Channels zugeordnet werden. Darüber hinaus ist auch die in Shopware bereitgestellte Funktion der Tags für Bilder und Kategorien zur Verwendung mit Pimcore geplant. Bei den Kategorien sollen zukünftig auch Kategorieinhalte (*Content*) über Pimcore verwaltet werden können.

Timo Trautmann verfügt über jahrelange Erfahrung in der Tech-Branche. Er hat erfolgreich in Agenturen größere Teams von PHP-Entwicklern geleitet und hunderte von Onlineprojekten umgesetzt. Spezialgebiet von Timo sind Schnittstellen aller Art. Hier sorgt er für die reibungslose Kommunikation von Systemwelten. Sein Fokus in den letzten Jahren lag insbesondere auf Schnittstellen von Drittsystemen wie Pimcore, SAP, Salesforce u. v. a. m. mit Shopware.

Links & Literatur

[1] https://www.shopware.com/de/

[2] https://pimcore.com/de

[3] https://scope01.com/pim-fuer-shopware/

[4] https://docs.shopware.com/en/shopware-platform-dev-en/admin-api-guide/sync-api

[5] https://www.shopware.com/de/news/api-first-entdecke-die-moeglichkeiten-mit-unserer-core-architektur/

[6] https://pimcore.com/docs/6.x/Development_Documentation/Objects/Object_Classes/Class_Settings/Custom_Views.html

[7] https://git-scm.com

[8] https://pimcore.com/docs/6.x/Development_Documentation/Deployment/Deployment_Tools.html

[9] https://pimcore.com/docs/6.x/Development_Documentation/Objects/Object_Classes/Data_Types/Object_Bricks.html

[10] https://pimcore.com/docs/6.x/Development_Documentation/Tools_and_Features/Application_Logger.html

[11] https://store.shopware.com/scope26257796298/pimcore-schnittstelle-pim-produktdaten-fuer-shopware-6.html

Erfolgreich Softwareprojekte modernisieren – Teil 1

Der Faktor Mensch

Ein Softwareprojekt ist mitunter ein komplizierter Mikrokosmos. Diesen gilt es an die Veränderungen der Zeit anzupassen, zu warten und zu modernisieren. Wie kann das möglichst ohne Komplikationen gelingen? In dieser Artikelserie werden wir dieser Frage auf den Grund gehen.

von Andreas Möller

Am Anfang ihrer Tätigkeit sorgen Softwareentwicklerinnen zumeist selbst für zukünftigen Modernisierungsbedarf. Bei der Mehrzahl der Projekte, an denen der Autor in den letzten zehn Jahren beteiligt war, wiesen diese Projekte bereits zu seinem Einstieg einen erheblichen Modernisierungsbedarf auf. Das ist nicht unbedingt verwunderlich, zumal der Auftrag in einigen dieser Fälle lautete, diese Projekte zu modernisieren. In anderen Fällen ging es eingangs darum, bei der Umsetzung dringend benötigter Funktionalitäten zu helfen. Hier waren sich die Auftraggeberinnen gar nicht im Klaren darüber, dass sich ihr Projekt in einem Zustand befand, der eine Weiterentwicklung nur schwerlich möglich machte. Nach den Einblicken, die der Autor im Laufe der Zeit gewonnen hat, geht er nun davon aus, dass die meisten Softwareprojekte einen massiven Modernisierungsbedarf haben. Dieser Bedarf bleibt meist so lange unentdeckt, bis sich etwas an den Umständen eines Unternehmens ändert.

Die Grundproblematik

Externe Schocks wie der Klimawandel, eine Wirtschaftskrise oder eine Pandemie können ein Unter-

Artikelserie

Teil1: Der Faktor Mensch
Teil 2: Einführung einer Build Pipeline
Teil 3: Einführung von Coding Standards
Teil 4: Einführung von Dependency Analysis
Teil 5: Einführung von Static Code Analysis
Teil 6: Einführung von Automated Tests
Teil 7: Einführung von Mutation Tests

Werden Modernisierungen alle Jahre wieder in Hau-Ruck-Aktionen angegegangen, gehen sie mit erheblichen Kosten einher.

nehmen in Zugzwang bringen. Aufträge und ganze Geschäftsfelder brechen weg, Kosten müssen eingespart werden. Konkurrenten, die sich besser aufgestellt haben, drängen in den Markt. Die Gesetzgebung erwartet unter Androhung von hohen Strafen, dass bestimmte Funktionalitäten eingeschränkt oder umgesetzt werden. Plattformen, Frameworks und Bibliotheken veralten, und Anbieterinnen zeigen an, dass Weiterentwicklung und Support eingestellt werden. Nun muss dringend etwas getan werden! Fehlen diese externen Faktoren, kann der Modernisierungsbedarf oft erst erkannt werden, wenn neue Mitarbeiterinnen eingestellt oder Freiberuflerinnen, Beraterinnen und Agenturen beauftragt werden. Neue Mitarbeiterinnen und Externe verfügen in der Regel über einen anderen Schatz an Erfahrungen und Kompetenzen, vor deren Hintergrund der konkrete Modernisierungsbedarf erst sichtbar werden kann. In einem Projekt werden Versionen von PHP verwendet, die offiziell nicht mehr unterstützt werden. Während neuere Versionen von PHP neue Funktionsumfänge und deutlich mehr Leistung bieten, sind sie allerdings nur bedingt abwärtskompatibel. In einem weiteren Projekt stellt sich das von den Gründern eines Unternehmens mit- und dann intern weiterentwickelte Framework als Sackgasse heraus. Es ist komplett ungetestet und wenig dokumentiert, und Entwicklerinnen, die ein Interesse haben, sich in dieses Framework einzuarbeiten und es weiterentwickeln wollen, sind schwer oder kaum zu finden. Angesichts einer Vielzahl an alternativen Bibliotheken und Frameworks und begrenzter Kapazitäten lohnt sich die Weiterentwicklung nicht.

In einem anderen Projekt wurden die verwendeten Bibliotheken und Frameworks in die Versionskontrolle aufgenommen. Hier ist zum Teil nicht mehr feststellbar, woher diese Bibliotheken und Frameworks kommen. Da im Laufe der Zeit leider auch Änderungen an diesem Code vorgenommen wurden, ist die Identifikation der konkreten Versionen zusätzlich erschwert. Mit Sicherheit kann man aber davon ausgehen, dass diese Bibliotheken und Frameworks komplett veraltet sind. Wiederum in einem anderen Projekt wird immerhin Composer verwendet, um Bibliotheken und Frameworks ins Projekt einzubinden. Leider sind auch hier die verwendeten Versionen komplett veraltet. Eine Aktualisierung ist aufwendig, da die Geschäftslogik viel zu eng mit diesen Bibliotheken und Frameworks verheiratet worden ist. Zum Teil ist eine Aktualisierung auch nicht mehr möglich, da die Weiterentwicklung kompatibler oder gänzlich neuer Versionen bereits vor Jahren eingestellt wurde. Die meisten Projekte leiden an einem geringen Automatisierungsgrad. Einen einheitlichen Coding-Standard gibt es nicht. Hier wird manuell getestet, da man keine Zeit hat, automatisierte Tests zu schreiben. In der Open-Source-Community weit verbreitete Tools sind nicht bekannt und werden hier auch nicht verwendet. Entwicklerinnen und Managerinnen haben sich an eine geringe Entwicklungsgeschwindigkeit gewöhnt und sind sich nicht im Klaren darüber, dass Softwareentwicklung insgesamt ganz anders aussehen und ablaufen kann.

Wie aber sollte die Modernisierung eines Softwareprojekts nun am besten angegangen werden? Die Modernisierung des Codes ist zwingend und zunächst auch naheliegend. Der Code bietet eine konkrete, greifbare Angriffsfläche, und die Ziele sind identifiziert. Die verwendeten PHP-Versionen sollen aktualisiert und die Kompatibilität soll überprüft werden. Das intern entwickelte Framework soll durch ein modernes Framework abgelöst, eine Verheiratung mit Geschäftslogik vermieden werden. Anstatt Fremdcode in die Versionskontrolle einzuchecken, soll dessen Herkunft festgestellt, und dieser mit Composer bereitgestellt werden. Anstatt diesen fremden Code zu modifizieren, sollen nötige Änderungen im eigenen Code vorgenommen oder sogar am Herkunftsort vorgeschlagen werden. Die heillos veralteten Bibliotheken und Frameworks sollen aktualisiert oder durch modernere Alternativen ersetzt werden. Es steht eine Vielzahl an Tools zur Verfügung, die dabei helfen können, Fehler aufzudecken und die Qualität des Codes in Kennzahlen zu erfassen. Zugleich können diese Tools dabei unterstützen, den Quellcode aufzuräumen und von Fehlern zu befreien. Diese Tools können Entwicklerinnen relativ schnell zur Seite gestellt werden, um sie einerseits bei der lokalen Entwicklung zu unterstützen, andererseits können dieselben Tools in einer gegebenenfalls vorhandenen oder noch einzurichtenden Build Pipeline verwendet werden. Dort können sie verhindern, dass Entwicklerinnen Änderungen am Quellcode vornehmen, die nicht den neuen Qualitätsansprüchen genügen. Wurden geeignete Kennzahlen ausgewählt, können hier schnelle Erfolge erzielt und Verbesserungen festgestellt werden. Aber sind diese Erfolge wirklich von Dauer, oder verbessern sich hier nur die Kennzahlen?

Lösungsansätze

Treten Entwicklerinnen und Managerinnen mit ihren Kenntnissen und Erfahrungen auf der Stelle, so werden sich kaum die Ergebnisse ihrer Arbeit verbessern. Werden Modernisierungen alle Jahre wieder in Hau-Ruck-Aktionen angegangen, gegebenenfalls bestimmt durch die Releasezyklen von verwendeten Frameworks, gehen diese Modernisierungen einher mit erheblichen Kosten

und Schmerzen. Wenn neu verwendete Tools diktieren, wie zu arbeiten ist, wird das hingenommen, aber oft nicht verstanden. Übernehmen Entwicklerinnen und Management neue Paradigmen und Qualitätsanforderungen, ohne deren Sinn zu verstehen oder zu erkennen, bleibt ihre persönliche Entwicklung auf der Strecke. Die persönlichen Erfahrungen der letzten Jahre haben dem Autor mehrfach gezeigt, dass alle Bemühungen, ein Softwareprojekt zu modernisieren, umsonst sind, wenn sie sich allein auf den Code beschränken. Der Code ist nur ein Resultat der Umstände, in denen er entstanden ist. Diese Umstände werden durch viele Faktoren bestimmt, von denen wiederum viele schwer und manche gar nicht beeinflusst werden können. Darunter fallen die Probleme, die durch das Projekt gelöst werden sollen. Ausschlaggebend sind aber letzten Endes immer die Fähigkeiten des Teams, diese Probleme zu verstehen und mit Hilfe von Software zu lösen, sowie die Fähigkeiten des Managements, den Bedürfnissen, Hinweisen und Anforderungen des Teams Rechnung zu tragen. Um hier nachhaltig Erfolge zu erzielen, ist es nach Erachten des Autors wichtig, dass Entwicklerinnen und Managerinnen, aber auch das von ihnen bestimmte Arbeitsumfeld in die Modernisierung einbezogen werden.

Grundsätzlich können Entwicklerinnen und Managerinnen auf eine Vielzahl von Angeboten zur Weiterbildung in der Community zurückgreifen. Der Quellcode von Bibliotheken und Frameworks ist öffentlich auf Plattformen wie GitHub und GitLab einsehbar. Jeder hat hier die Möglichkeit, den dort hinterlegten Code und die Interaktionen von Entwicklerinnen zu studieren. Ebenfalls ist hier jeder, sofern er sich an die gebotenen Verhaltensweisen hält, eingeladen, an der Wartung, Weiter- und Neuentwicklung von Bibliotheken und Frameworks teilzuhaben. Von behutsamen ersten Schritten, wie der Behebung von Rechtschreibfehlern in der Dokumentation, ist es oft nicht weit, bis man erste Fehler beheben und neue Funktionalitäten entwickeln kann. Wenn im eigenen Unternehmen keine Entwickler vorhanden sind, von denen man etwas lernen könnte, so kann man auf diesen Plattformen mit Sicherheit sehr gute Kenntnisse und Erfahrung gewinnen, die sich auch auf die Arbeitsweise im eigenen Unternehmen übertragen lassen.

Viele der dort umtriebigen Entwicklerinnen schreiben darüber hinaus Artikel auf persönlichen Webseiten, teilen diese in sozialen Medien, veröffentlichen Artikel in Magazinen oder schreiben ganze Bücher. Diese sind zum Teil kostenlos oder auch oft für relativ wenig Geld digital erhältlich. Seit COVID-19 sind zwar die Möglichkeiten für persönliche Treffen stark eingeschränkt, aber grundsätzlich gab es bis dahin auch sehr viele Möglichkeiten für einen direkten Austausch mit anderen Entwicklerinnen. Dazu gehören regelmäßig stattfindende, kostenlose User Groups, auf denen Entwicklerinnen Vorträge halten und man sich über Erfahrungen austauschen kann, oft auch über das offizielle Ende der Veranstaltung hinaus. Viele Open-Source-Projekte veranstalten regelmäßig Code-Sprints, auf denen sich an diesem Projekt interessierte Entwicklerinnen einfinden können, um dieses Projekt gemeinsam weiterzuentwickeln. Hier kann man tiefe Einsichten und Erkenntnisse gewinnen und Freundschaften schließen. Verschiedene Unternehmen veranstalten Hackathons, auf denen man mit anderen Menschen, die nicht immer nur Softwareentwicklerinnen sind, in kurzer Zeit Projekte umsetzen kann. Auch hier lassen sich viele neue Erkenntnisse gewinnen. Zu diesen Veranstaltungen gehören ebenfalls vielerorts stattfindende Konferenzen, auf denen (allerdings kostenpflichtig) Vorträge und Workshops besucht werden können. Bekannt sind hier auch die zwischen oder parallel zu Vorträgen und Workshops stattfindenden, sogenannten Hallway Tracks. Hier kann man leicht mit anderen Entwicklern und Experten in Kontakt treten und sich mit ihnen austauschen. Wenn auch wegen COVID-19 viele dieser Veranstaltungen ausfallen mussten, wurden einige von ihnen ins Internet verlagert. Hier ist die Teilnahme noch einmal erleichtert; die Reise in eine entfernte Stadt entfällt.

Nehmen Entwicklerinnen diese öffentlichen Angebote eigenverantwortlich in Anspruch, können viele positive Impulse auf andere Entwicklerinnen und die Arbeit im Team ausgehen. Mitunter fehlt hier nur ein Anstoß, um einmal über den eigenen Tellerrand hinauszublicken. Alternativ und zusätzlich zur eigenverantwortlichen Weiterbildung von Entwicklerinnen können unternehmensintern durchaus ähnliche Angebote geschaffen werden. Diese Angebote können neben einer individuellen Weiterbildung auch zu einem internen Wissensaustausch anregen. So könnten Entwicklerinnen gebeten werden, ein spannendes und eventuell auch für das Team wichtiges Thema zu erörtern. Das Thema könnte von ihnen dann so aufbereitet werden, dass sie ihr erlangtes Wissen in einem Vortrag oder einer Reihe von Vorträgen mit Kolleginnen teilen. Daraus könnte eine regelmäßige Veranstaltung erwachsen, die ein Sprungbrett darstellen könnte, zunächst auf einer User Group und später auch auf Konferenzen Vorträge zu halten. Eine solche interne Veranstaltung könnte auch für die Öffentlichkeit zugänglich gemacht werden. Die so entstandene User Group könnte ein Anziehungspunkt für Entwicklerinnen in der weiteren Umgebung werden und so den Wissensaustausch stärker fördern.

Ist ein Thema für einen Vortrag zu umfangreich oder verlangt nach praktischer Arbeit aller Beteiligten, dann könnte das Thema auch in einem intern veranstalteten Workshop erarbeitet werden. Fehlen intern dazu die Kompetenzen, kann man externe Entwicklerinnen oder Beraterinnen zu Rate ziehen, die nicht nur diese Workshops professionalisieren, sondern gegebenenfalls auch über einen längeren Zeitraum zur Verfügung stehen können, um konkrete Themen zu erarbeiten. Im Arbeitsalltag könnte der Wissensaustausch angeregt werden, wenn es dem Unternehmen gelingt, einen Raum zu schaffen, in dem Kritik offen ausgesprochen und angenommen werden kann. Betrachten Entwicklerinnen

Aus einem regelmäßigen Pair Programming mit rotierenden Entwicklerinnen könnte auch ein Mob Programming abgeleitet werden.

die Ergebnisse ihrer Arbeit als von sich selbst losgelöst, fühlen sie sich sicher, Zweifel auszusprechen, Fragen zu stellen und auf Fehler hinzuweisen, können oft kostspielige Fehlentwicklungen vermieden werden. Regelmäßige Code Reviews können Wunder bewirken, wenn sich Entwicklerinnen erst einmal daran gewöhnen, jeden Morgen zunächst die von anderen Entwicklerinnen geöffneten Pull Requests zu begutachten. Hier können nicht nur neue Erkenntnisse gewonnen, sondern auch Fehler früh erkannt werden. Anstatt diese Code Reviews asynchron zu vollziehen, könnte man Entwicklerinnen auch ermuntern, mit anderen Entwicklerinnen im Pair Programming an der Software zu arbeiten. Während eine Entwicklerin die Richtung vorgibt, schreibt die andere den Code. Code Reviews werden dann überflüssig, sie finden während der gemeinsamen Entwicklung statt. Aus einem regelmäßigen Pair Programming mit rotierenden Entwicklerinnen könnte auch ein gelegentliches Mob Programming abgeleitet werden, bei dem eine größere Gruppe von Entwicklerinnen die Richtung vorgibt, während im Wechsel eine Entwicklerin aus dieser Gruppe den Code schreibt. Hierbei können nicht nur Erkenntnisse zur Programmierung gewonnen, sondern auch neue Fertigkeiten im Umgang mit Tools und der IDE erlernt werden. Viele Arbeiten, die oft nicht interessant sind, können so schneller erledigt werden, damit man mehr Zeit hat, sich anspruchsvollen Aufgaben zu widmen.

Über längere Zeiträume könnten sich darüber hinaus erfahrenere den weniger erfahrene Entwicklerinnen als Mentorinnen zur Verfügung stellen. Hier kann oft mühsam erarbeitetes Wissen auf kurzen Wegen weitergegeben werden – häufig auch in beide Richtungen. Nicht zuletzt aber bewegen sich Entwicklerinnen und Managerinnen oft in einem Arbeitsumfeld, das eine vernünftige Arbeitsweise nur schwer ermöglicht. Gelingt es dem Unternehmen, hier auf die Bedürfnisse einzelner und des Teams einzugehen, können auch hier dauerhaft Erfolge erzielt werden. Dazu gehört zum Beispiel, dass es für Entwicklerinnen zur Lösung komplexer Probleme erforderlich ist, sich auch über längere Zeiträume hinweg auf diese Probleme konzentrieren zu können. Hier könnten die Anzahl und die Länge der Unterbrechungen deutlich reduziert werden. Störfaktoren könnten lautstarke Telefonate oder Gespräche in unmittelbarer Umgebung sein, aber auch eine Vielzahl von Meetings, bei denen oft nicht klar ist, warum sie einberufen worden sind. Hier könnten Rückzugsmöglichkeiten für ungestörtes Arbeiten geschaffen werden, oder Entwicklerinnen zeitweise oder dauerhaft ins Homeoffice entlassen werden. Nicht nur wegen COVID-19 ergibt es meiner Meinung nach wenig Sinn, Entwicklerinnen in ein Großraumbüro zu verfrachten. Warum sollen sie Zeit mit An- und Abreise verschwenden, wenn sie dort ihre Arbeit nicht vernünftig erledigen können? Wann und wo Arbeit erledigt wird, kann nicht wichtiger sein als die Ergebnisse der Arbeit selbst. Entwicklerinnen, die schon über längere Zeit remote arbeiten, schwärmen von den neu gewonnenen Freiheiten, dem ihnen entgegengebrachten Vertrauen und einem deutlichen Zuwachs an Produktivität. Hier sieht der Autor deutlichen Nachholbedarf und ein erhebliches Potenzial, die Lebens- und Arbeitsqualität dauerhaft zu erhöhen.

Fazit

Es sollte klar sein, dass Entwicklerinnen auch über Kompetenzen verfügen, die über die Programmierung hinausgehen. Viele Entwicklerinnen sind quer eingestiegen und haben bereits in anderen Berufen viele Erfahrungen sammeln können, und nicht zuletzt sind Entwicklerinnen auch immer Anwender von Software. Wenn diese Kompetenzen anerkannt werden und sich Entwicklerinnen im Arbeitsalltag auch an vielleicht unerwarteten Stellen einbringen, können ebenfalls deutliche Verbesserungen erzielt werden. Durch erweiterte Aufgaben, die Entwicklerinnen freiwillig annehmen, können sie auch motiviert werden, sich mehr mit der Problemwelt zu befassen. Das könnte zum Beispiel bedeuten, dass Entwicklerinnen früh in die Planung von Projekten mit einbezogen werden, und nicht erst, wenn es an die konkrete Umsetzung geht. Nicht jedes Problem muss mit Software gelöst werden, und nicht jede Software muss selbst entwickelt werden. Mitunter ist hier der Einsatz analoger oder fertiger Lösungen sinnvoller. Wenn ein Unternehmen Anreize schafft für eine kontinuierliche Weiterbildung und einen kontinuierlichen Wissensaustausch, wenn die Arbeitsweise und Arbeitsumgebung kontinuierlich verbessert werden, muss eine Modernisierung kein Projekt sein, das wiederkehrend geplant und angegangen werden muss. Sie findet kontinuierlich und von innen heraus statt.

Nach ersten Programmierversuchen Anfang der 1990er Jahre entdeckte **Andreas Möller** im Sommer 2001 PHP für sich. Seit 2007 ist er als freiberuflicher Softwareentwickler, und in den letzten Jahren auch als Berater tätig. Er entwickelt Open-Source-Software und veröffentlicht Artikel auf seinem Blog.

https://localheinz.com **@localheinz** **@localheinz**

Phalcon: Performant, robust, geschrieben in C

Der Falke unter den PHP Frameworks

Phalcon ist ein noch eher unbekanntes, inzwischen aber ausgereiftes PHP Framework, das von Version zu Version mächtiger wird. Höchste Zeit, den Falken etwas genauer unter die Lupe zu nehmen!

von Jérémy Pastouret

Wir befinden uns im Jahr 2011. Gerade ist die erste Version des Laravel-Frameworks erschienen. Im selben Jahr brütet ein gewisser Andrés Gutiérrez über einer Idee für eine neue Art von Webframework: Extrem performant soll es sein, praxisnah und voller nützlicher Features.

In einem Interview aus dem Jahr 2015 [1] erklärt Gutiérrez, er habe zunächst versucht, dieses Framework in Sprachen wie Go, Rust oder C zu schreiben – ohne Erfolg. Erst als er einen neuen Anlauf mit PHP unternimmt, erhält er die gewünschten Resultate. Zu diesem Zeitpunkt hat Andrés neun Jahre Erfahrung in Unternehmen bei der Entwicklung von personalisierten Web-Frameworks und beschäftigt sich ausgiebig mit den Konzepten, Design Patterns und Paradigmata von PHP. In einem späteren Interview von 2016 [2] rät er dazu, sich die Seite https://PHPTheRightWay.com [3] anzuschauen, um zu einem echten PHP-Guru zu werden.

Geburt des Frameworks

Um das schnellstmögliche PHP-Framework zu entwickeln, hatte Andrés die Idee, eine PHP-Erweiterung statt einer High-Level-Abstraktion zu schreiben. Auf diese Art und Weise sollte das Framework tief in der DNA von PHP verwurzelt werden. Da PHP-Erweiterungen genauso wie die Sprache PHP selbst in C geschrieben sind, mussten also Programmierkenntnisse in C vorhanden sein. Ausgehend von diesem Prinzip veröffentlichte Andrés eine erste Version namens Spark, die komplett in C verfasst war. Mit diesem Proof of Concept konnte er eine Gruppe von begeisterten Mitstreitern überzeugen, ein komplettes Framework zu entwickeln.

Die Gruppe entschied sich für den Namen Phalcon – ein Wortspiel aus PHP und dem englischen Falcon. Nach einer intensiven Arbeitsphase erschien im November 2012 eine erste Betaversion 0.4.5 des Frameworks.

Die Geburt von Zephir

Bei der Programmierung dieser ersten Version wurden sich die Entwickler eines Problems bewusst: Nur wenige Entwickler zeigten sich bereit, ein Framework in C zu schreiben. Um diesem Umstand zu begegnen, hatte die Gruppe eine weitere Idee: Wie wäre es, eine Sprache zu haben, die syntaktisch nahe an PHP angelehnt ist, aber darauf spezialisiert, Erweiterungen in C zu schreiben? Als Ergebnis dieses Gedankenspiels kam 2013 die Sprache Zephir zur Welt. Die Version 2.0 des Phalcon-Frameworks wurde daraufhin komplett in Zephir geschrieben, wie auch alle weiteren Versionen danach.

Zephir hat sehr zur Popularität von Phalcon beigetragen. Die Community vergrößerte sich und es wurde einfacher, Beiträge zu Phalcon zu leisten. Zephir wird mittlerweile auch von anderen Projekten genutzt: Viele performante Erweiterungen für Unternehmensanwendungen sind bereits entstanden.

Um einen ersten Eindruck der Syntax von Zephir zu erhalten, werfen wir einen Blick auf den Quellcode von Phalcon. Listing 1 zeigt eine Funktion, die die Zugriffsrechte auf verschiedene Seiten eines Webprojekts verwaltet.

Im Vergleich zu PHP sieht man einige kleine Unterschiede:

- Um den Rückgabewert einer Funktion zu erhalten, wird das Pfeilzeichen verwendet.

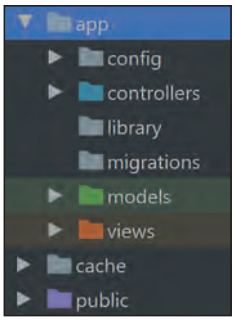


Abb. 1: MVC-Architektur von Phalcon

- Die Definition einer Variablen geschieht mit dem Schlüsselwort *var*.
- Das Schlüsselwort erlaubt es, definierten Variablen Werte zuzuweisen.
- Das Zeichen *$* nach *this* entfällt.

Abgesehen von diesen Abweichungen handelt es sich um traditionelles PHP.

Phalcon heute

Die neueste Version von Phalcon ist 4.0.6 vom 16. Mai 2020. Zephir ist am 13. Mai 2020 in der Version 0.12.19 erschienen. Am 11. November 2019 wurde Phalcon offiziell in den berühmten PECL-Katalog (PHP Extension Community Library) aufgenommen. Dadurch ist es jetzt noch einfacher, Phalcon zu installieren. Phalcon und Zephir haben sich beide seit ihren ersten Versionen sehr stark weiterentwickelt. Wir wollen im Folgenden die zentralen Funktionalitäten des aktuellen Phalcon Frameworks genauer beleuchten. Los geht's!

Listing 1

```
/**
 * Adds a role to the ACL list. Second parameter allows inheriting
 * access data from other existing role
 *
 * ```php
 * $acl->addRole(
 *     new Phalcon\Acl\Role("administrator"),
 *     "consultant"
 * );
 *
 * $acl->addRole("administrator", "consultant");
 * $acl->addRole("administrator", ["consultant", "consultant2"]);
 * ```
 */
public function addRole(role, accessInherits = null) -> bool
{
  var roleName, roleObject;

  if typeof role == "object" && role instanceof RoleInterface {
      let roleObject = role;
  } elseif is_string(role) {
      let roleObject = new Role(role);
  } else {
      throw new Exception(
        "Role must be either a string or implement RoleInterface"
    );
  }

  let roleName = roleObject->getName();

  if isset this->rolesNames[roleName] {
    return false;
  }
```

Die Funktionalitäten von Phalcon

Phalcon wird wie eine Extension installiert. Alle Ordner, aus denen Phalcon zusammengesetzt ist, werden in einem einzigen Ordner kompiliert. Allerdings muss Phalcon in einer bestimmten Version kompiliert werden, um mit dem jeweiligen System (32/64 Bit), der jeweiligen PHP-Version und dem verwendeten Betriebssystem zu funktionieren. Die Kompilierung bringt zahlreiche Vorteile mit sich:

- Der gesamte Code wird analysiert. Tritt ein Fehler auf, wird die Kompilierung angehalten. Das stellt im Vergleich zu interpretiertem Code eine erhöhte Stabilität sicher.
- Der Code wird komprimiert. Dadurch wird er schlanker und einfacher zu installieren. Anstatt eine Vielzahl von verschiedenen Ordnern einzusammeln, wie es andere Frameworks tun, genügt das Copy-Paste einer *.dll* oder eines *.so*-Ordners.
- Die Erweiterung ist für eine spezifische Architektur optimiert.
- Da Phalcon auf einem Webserver installiert ist, können alle Anwendungen es benutzen.
- Ab dem Zeitpunkt, da der Daemon des Webservers startet, sind die Klassen und Funktionen von Phalcon verfügbar.

Über die genannten Vorteile hinaus konsumiert Phalcon wenig Speicher und CPU. Wer sich genauer informieren möchte, dem sei der Benchmarktest unter [4] ans Herz gelegt.

MVC-Architektur

Abbildung 1 zeigt das Grundgerüst eines Phalcon-Projekts. Man erkennt dabei gut die gängige MVC-Architektur mit den Ordnern für Models, Views und Controllers. Der Ordner *config* erlaubt es, bestimmte Parameter, Services, Pfade und Namensräume zu setzen.

Listing 2

```
$app = new Micro();

$app->get('/', function () {
  return $this->response->setJsonContent(
    [
      'status' => 'Tout fonctionne',
    ]
  );
}
);

$app->notFound(function () use($app) {
  $app->response->setStatusCode(404, "Not Found")->sendHeaders();
});

$app->handle($_GET['_url'] ?? '/');
```

Über das Verzeichnis *migrations* lassen sich Datenmigrationen von SQL-Datenbanken durchführen. Der Ordner *public* enthält die statischen Ressourcen (JavaScript, Bilder, CSS etc.). Schließlich liegen im Cache-Verzeichnis die von der Volt/Phtml-Rendering-Engine erzeugten PHP-Dateien.

Das REST API

Es ist einfach, ein API mit Phalcon zu entwickeln. Es genügt, mit phalcon-devtools ein Projekt des Typs *Micro* zu erzeugen. Listing 2 zeigt ein einfaches Beispiel-API mit Phalcon.

Mit der Klasse *Micro* ist es möglich, Pfade mit ihren assoziierten Funktionen zu erzeugen. Das *GET* kann natürlich durch andere API-Methoden wie *POST* oder *PUT* ersetzt werden. Es ist auch möglich, unbekannte Pfade zu generieren (z. B. 404-Fehler).

Das Kommandozeilen-Interface: CLI

In den meisten Projekten müssen bestimmte Aufgaben im Hintergrund erledigt werden, und das aus mehreren Gründen:

- Lange andauernde Arbeiten, während derer der Entwickler etwas anderes tun kann und per E-Mail oder Notification informiert wird, wenn die Arbeit erledigt ist
- Wiederkehrende Aufgaben: typische Wartungsarbeiten wie das Bereinigen einer Datenbank
- Geplante Aufgaben, beispielsweise das automatische Versenden von E-Mails zu einem Zeitpunkt *T*

Um mit dieser Problematik umzugehen, gibt es zwei Lösungen: Entweder verwendet man dafür eine zusätzliche Technologie, oder man nutzt PHP in einer Konsole. Um zu vermeiden, zusätzliche Elemente zu installieren, bevorzuge ich Cronjobs und den Aufruf eines PHP-Skripts. Phalcon stellt einen Konsolenmodus zur Verfügung, um diesen Anforderungen – und noch vielen weiteren – zu begegnen. Um zu verstehen, wie das funktioniert, erzeugen wir ein Projekt des Typs CLI mit den Phalcon Devtools. Die folgende Anweisung erzeugt das in **Abbildung 2** gezeigte Projekt „MesTaches“:

```
phalcon project MesTaches cli
```

Ein Projekt des Typs CLI kann leicht in ein anderes Projekt integriert werden. Es genügt dafür, die folgenden Ordner hinzuzufügen:

- *run*
- *app/tasks/**
- *app/bootstrap.php*

Wie nicht schwer zu erraten ist, fügt man eine neue Aufgabe hinzu, indem man im Verzeichnis *tasks* neue Klassen erzeugt oder in bestehenden Klassen Funktionen hinzufügt. Mit der folgenden Anweisung kann eine Aufgabe gestartet werden:

```
php run version
```

Mit diesem Kommando startet Phalcon die Funktion *mainAction()* der Klasse *VersionTask.php*.

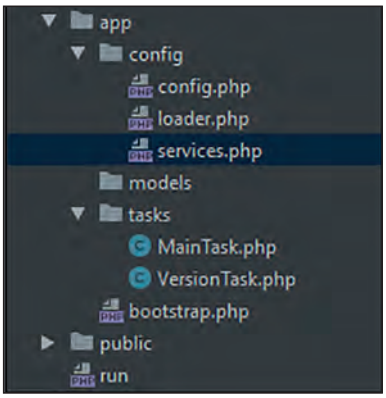


Abb. 2: Das via CLI erzeugte Projekt MesTaches

Abfrage der Datenbank

Phalcon bietet verschiedene Möglichkeiten, um mit einer Datenbank zu interagieren:

- ORM – Object Relational Mapping
- PHQL – Phalcon Query Language
- DAL – Database Abstraction Layer

Mit dem Database Abstraction Layer greift Phalcon mittels PDO auf die Datenbank zu. Das ist die klassische Methode, um mit einer Datenbank wie MySQL, PostgreSQL oder SQLite zu interagieren. Das folgende Beispiel nutzt DAL:

Listing 3

```
use Phalcon\Mvc\Model;

use Phalcon\Validation\Validator\Email as EmailValidator;

class Utilisateur extends Model
{
  public $email;

  public $password;

  public function initialize()
  {
    $this->hasMany('id', Cours::class, 'utilisateur_id', [ 'alias' => 'cours' ]);
  }

  public function validation()
  {
    $validator = new Phalcon\Validation();

    $validator->add('email',new EmailValidator(
      [
        'model'   => $this,
        'message' => 'Veuillez entrer une adresse mail correcte',
      ]
    )
    );

    return $this->validate($validator);
  }
}
```

```
$this->db->fetchAll('
  SELECT id, prenom, nom, email
  FROM utilisateurs
  WHERE prenom = :prenom',
  Db::FETCH_ASSOC,
  [
    'prenom' => 'Louise',
    'id'     => 2
  ]
);
```

Listing 4

```
$sPHQL = 'SELECT * FROM NovaMooc\Models\Utilisateurs
            WHERE prenom like :prenom:';

$aResultat = $this->modelsManager->executeQuery(
  $sPHQL,
  [
    'prenom' => '%a%'
  ]
);
```

Listing 5

```
$oConstructeur = $this->modelsManager
  ->createBuilder()
  ->from(Utilisateurs::class);

$oPaginateur = new QueryBuilder(
  [
    'builder' => $oConstructeur,
    'limit'   => 10,
    'page'    => $nPageCourante,
  ]
);

$oPagine = $oPaginateur->paginate();
```

Listing 6

```
use Phalcon\Acl\Role;
$aRoles = [
  'vendeurs' => new Role(
    'vendeurs',
    'Accès uniquement à la section de vente de produits.'
  ),
  'clients'  => new Role(
    'clients',
    'Accès uniquement à la section d\'achat de produits.'
  )
];

foreach ($aRoles as $oRole) {
  $oAcl->addRole($oRole);
}
```

Um eine Datentabelle mittels ORM abzufragen, muss eine Klasse erstellt werden. In dieser Klasse können die Attribute der Tabelle, die Beziehungen zu anderen Tabellen und Validatoren angegeben werden. Die Idee besteht darin, Objekte zu manipulieren, einfach von Tabelle zu Tabelle zu wechseln und die Gültigkeit der in die Datenbank eingegebenen Daten sicherzustellen. Listing 3 zeigt ein Beispiel für die Nutzung von ORM.

Kommen wir schließlich zur performantesten Methode: PHQL. PHQL ist als SQL-Abfrage-Parser in Phalcon integriert, jedoch in C geschrieben. Auf diese Weise werden Analyse und Ausführung von SQL-Abfragen beschleunigt. Darüber hinaus verbraucht PHQL sehr wenig Speicher, was es threadsicher macht. In Listing 4 wird die Nutzung von PHQL demonstriert.

Verwaltung eines Ergebnisstapels mit Paginierung

Phalcon bietet auch ein Paging-Modul, das effizient und schnell einzurichten ist. Tatsächlich kann Phalcon Ergebnisse direkt aus der Datenbank oder aus einer Datentabelle paginieren. Man kann beispielsweise eine CSV-Datei laden und sie von Phalcon paginieren lassen (Listing 5).

Zur Verfügung stehen eine Vielzahl an Properties, um auf Daten zuzugreifen:

- *getPrevious()*: die Seitenzahl der vorherigen Seite
- *getCurrent()*: die Seitenzahl der aktuellen Seite

Listing 7

```
use Phalcon\Acl\Component;
$aActionParRole = [
  'vendeurs' => [
    'vendeurs' => [ 'profil' ],
    'produits' => [ 'nouveau', 'édition' ]
  ],
  'clients'  => [
    'clients'  => [ 'profil' ],
      'produits' => [ 'acheter' ]
  ],
  '*' => [
    'index'   => [ 'index', 'connexion', 'déconnexion' ],
    'erreurs' => [ '*' ]
  ]
];

foreach ($aActionParRole as $sRole => $aRole) {
  foreach ($aRole as $sControleur => $aActions) {
    $oComponentControleur = new Component($sControleur);
    foreach($aActions as $sAction){
      $oAcl->addComponent($oComponentControleur, $sAction);
      $oAcl->allow($sRole, $sControleur, $sAction);
    }
  }
}
```

- *getItems()*: die Liste der aktuellen Elemente
- *getNext()*: die Seitenzahl der nächsten Seite
- *getLast()*: die Seitenzahl der letzten Seite
- *getTotalItems()*: die Anzahl der zu ladenden Elemente

Verwaltung von Zugriffsrechten

Mit Phalcon lassen sich einfach verschiedene Strategien einer Zugriffsverwaltung umsetzen. Zunächst wird dafür eine Defaultstrategie definiert: etwa „alles zurückweisen“ oder „alles erlauben“.

```
$oAcl = new Phalcon\Acl\Adapter\Memory();
$oAcl->setDefaultAction(Phalcon\Acl\Enum::DENY);
```

Danach werden Namen und Beschreibungen der Rollen definiert (Listing 6).

Schließlich teilt man Phalcon mit, welche Rechte man den verschiedenen Nutzertypen verleihen möchte (Listing 7).

Das Sternzeichen * steht für „alle“. In Listing 7 haben also alle Benutzer Zugriff auf die Index-, An- und Abmeldeseiten des Index-Controllers (*'index'*, *'connexion'*, *'déconnexion'*). Darüber hinaus haben alle Benutzer das Recht, auf sämtliche Aktionen des Controllers *erreurs* zuzugreifen. Mit der Funktion *addResource* müssen nun alle Controller dem Objekt ACL hinzugefügt werden. Es gibt dann zwei mögliche Vorgehensweisen: die erlaubten Aktionen festlegen oder die für einen bestimmten Benutzer verbotenen Aktionen definieren. Um einen Benutzer zu autorisieren, kommt die Funktion *allow* zum Einsatz. Um den Zugriff zu verweigern, existiert die Funktion *deny*. Um dreifach verschachtelte *foreach* zu vermeiden, schlägt Phalcon vor, das ACL-Objekt zu serialisieren, um es in einer Session oder in einer Datei zu speichern. Nachdem alle Einstellungen vorgenommen wurden, kann man einfach die Funktion *isAllowed* nutzen, um herauszufinden, ob ein Benutzer Zugriff auf eine Seite hat oder nicht. Bei der folgenden Abfrage gibt die Funktion *true* zurück, weil ein Benutzer das Recht hat, ein Produkt zu kaufen.

```
$oAcl->isAllowed('clients', 'produits', 'acheter');
```

Die folgende Abfrage resultiert in einem *false*, denn nur ein Verkäufer hat das Recht, ein neues Produkt zu erstellen.

```
$oAcl->isAllowed('clients', 'produits', 'nouveau');
```

Dependency Injection

Mittels Dependency Injection können Services erstellt werden, die überall in der Anwendung genutzt werden können. Es ist möglich, zwei Typen von Services zu erzeugen:

- einfach: Aufruf des kompletten Codes
- geteilt: funktioniert wie ein Singleton

Diese Services werden im Ordner *app/config/services.php* definiert. Hier ein Beispiel eines einfachen Service:

```
$di->set('fpdf', function () {
  $oPdf = new Fpdf();

   $oPdf->AddPage();
   $oPdf->Image('logo_entete.png',8,5,25);

  return $oPdf;
}
);
```

Dieser Code erzeugt bei jedem Aufruf des Service ein neues PDF. Dieses PDF ist mit einem Header-Logo vorkonfiguriert. Um den Dienst aufzurufen, z. B. von einem Controller aus, genügt die folgende Anweisung:

```
$oPDF = $this->di->get('fpdf');
```

Ein geteilter Service sieht folgendermaßen aus:

```
$di->setShared('config', function () {
  return include APP_PATH . "/config/config.php";
});
```

Um darauf zuzugreifen, schreibt man die folgende Anweisung:

```
$oConfig = $this->di->get('config');
```

Listing 8

```
{% block produit %}
  {% for produit in produits %}
    * Nom : {{ produit.nom|e }}
    {% if produit.status == 'actif' %}
      produit : {{ produit.prix + produit.taxes/100 }}
    {% endif %}
  {% endfor %}
{% endblock %}
```

Listing 9

```
class UtilisateurForm extends Form
{
  public function initialize()
  {
    $oEmail = new Email('email', [
      'placeholder' => 'Saisir un e-mail',
      'class'       => 'form-control',
    ]);
    $oEmail->setLabel('E-mail');
    $this->add($oEmail);

    $oMotDePasse = new Password('mot_de_passe', [
      'placeholder' => 'Saisir un mot de passe',
      'class'       => 'form-control',
    ]);
    $oMotDePasse->setLabel('Mot de passe');
    $this->add($oMotDePasse);
  }
}
```

Die Template-Engine Volt

Phalcon bietet eine eigene View-Rendering-Engine. Diese ist natürlich auch in Zephir/C geschrieben, was sie ausgesprochen schnell macht. Volt bettet viele sehr nützliche Funktionen ein und erlaubt zum Beispiel, die Daten zu formatieren (Leerzeichen entfernen, Zeichen vermeiden, die JS-Fehler verursachen etc.) oder sogar zu transformieren (die Werte eines Arrays sortieren, auf Großbuchstaben umstellen etc.). Diese Funktionen findet man in der Phalcon-Dokumentation, Beispiele zur Nutzung sind im Phalcon-Buch unter [5] beschrieben. Listing 8 ist in Volt verfasst:

Formularverwaltung

Mit Phalcon ist es möglich, in relativ kurzer Zeit verschiedene Arten von Formularen zu erstellen. In den phalcon-devtools können Formulare mit einem einzigen Befehl hinzugefügt, geändert oder gelöscht werden. Man muss lediglich angeben, welche Tabelle als Grundlage für die Erstellung der Formulare verwendet werden soll. Im Folgenden ist ein Beispiel für einen solchen Befehl:

```
phalcon scaffold --table-name=utilisateur --template-engine=volt
```

Formulare können auch ohne die phalcon-devtools erstellt werden. Grundsätzlich bietet Phalcon verschiedene Arten von Feldern mit vorkonfigurierten Validatoren an (E-Mail, Datei, Datum, Textbox, ...). Es können auch eigene Validatoren ergänzt werden, und mit einer einzigen Anweisung kontrolliert Phalcon die vom Benutzer eingegebenen Daten. Wenn es Fehler gibt, können die Meldungen abgerufen und angezeigt werden, wie in Listing 9 zu sehen.

Nachdem das zuvor erstellte Formular instanziiert wurde, muss das Objekt lediglich in die View weitergeleitet werden. In Volt verwendet man zu diesem Zweck die folgende Anweisung, um den Feldbezeichner zu erzeugen:

```
{{ form.label("email") }}
```

Das Formularobjekt generiert die folgende Zeile:

```
<label for="email">E-mail</label>
```

Um die Eingabe zu erhalten, wird die Render-Funktion verwendet:

```
{{ form.render('email') }}
```

Die Funktion erzeugt die folgende Zeile:

```
<input type="email" id="email" name="email" class="form-control"
placeholder="Saisir un e-mail">
```

Event-Manager

Phalcon verfügt über einen umfangreichen Katalog von Events, zu denen eine Verbindung hergestellt werden kann. Dieses System funktioniert genauso wie die Hooks von WordPress. Es ist auch möglich, eigene maßgeschneiderte Events zu entwickeln.

Diese Events sind in vielen Szenarien sehr nützlich: Beispielsweise kann damit der Zugang zu den verschiedenen Seiten einer Anwendung kontrolliert werden. Ein weiteres praktisches Beispiel besteht darin, alle Exceptions abzufangen, um sie dem Logging-System weiterzuleiten. Mit Phalcon gibt es verschiedene Gestaltungsmöglichkeiten, sodass sehr wenig Code dupliziert werden muss und Anwendungen einfacher zu warten sind. Listing 10 zeigt ein Beispiel.

In diesem Beispiel wird ein Plug-in (eine Klasse) mit dem Event *beforeExecuteRoute* verknüpft, um den Zugriff auf die Seiten zu steuern (verbundener/nicht verbundener Benutzer, Ebene der Rechte etc.). Das Ereignis *beforeException* erlaubt es, die Exception abzufangen, bevor sie den Nutzer der Webanwendung erreicht.

Cacheverwaltung

Wer weiter an der Performanceschraube drehen möchte, sollte sich das Caching genauer anschauen. Caching ist in verschiedenen Fällen nützlich:

- Wenn eine lange und komplexe Berechnung durchgeführt wird und sich das Ergebnis nur selten ändert.
- Wenn sich eine generierte HTML-Seite fast nie ändert (z. B. im Fall einer statischen Showcase-Seite).
- Wenn die aus einer SQL-Abfrage abgerufenen Daten sehr selten variieren.

Das Phalcon-Team empfiehlt jedoch in jedem Fall, die Performance vor und nach der Aktivierung des Caching miteinander zu vergleichen. Der Caching-Dienst hat Schnittstellen zu verschiedenen Engines:

- Apcu
- Libmemcached
- Speicher (in Memory)
- Redis
- Stream

Nachdem der Cachedienst mit einer der vorherigen Engines initialisiert wurde, können gecachte Daten von überall her hinzugefügt bzw. abgerufen werden. Listing 11 zeigt ein Beispiel.

In Listing 11 sollen Lotterieergebnisse im Cache zwischengespeichert werden. Es gibt zwei Gründe, warum der Cache *null* zurückgibt: Entweder wurde der Cache noch nicht initialisiert oder er ist abgelaufen. Wenn das Ergebnis *null* ist, wird eine Anfrage an einen Drittserver gestellt, um die neuesten Lottoergebnisse abzurufen. Schließlich werden diese neuen Ergebnisse im Cache für eine Dauer von 86 400 Sekunden (24 Stunden) gespeichert. Man muss die Ablaufzeit nicht unbedingt angeben, bei der Initialisierung des Cachedienstes wird ein Defaultwert für die Ablaufzeit festgelegt.

Der Cache für SQL-Abfragen

Phalcon geht noch einen Schritt weiter, wenn es um SQL-Abfragen geht. In einer PHQL-Abfrage kann di-

Listing 10

```
use Phalcon\Mvc\Dispatcher;
use Phalcon\Events\Manager;

$container->set(
 'dispatcher',
 function () {

   $oGestionnaireEvenement = new Manager();

   $oGestionnaireEvenement->attach(
    'dispatch:beforeExecuteRoute',
    new SecuritePlugin
   );

   $oGestionnaireEvenement->attach(
    'dispatch:beforeException',
    new ErreurPlugin
   );

   $oDispatcheur = new Dispatcher();

   $oDispatcheur
    ->setEventsManager($oGestionnaireEvenement);

   return $oDispatcheur;
 }
);
```

Listing 11

```
$oCache = $this->di->get('cache');
$sCleCache = 'loto.resultats';

$aResultats  = $oCache->get($sCleCache);

if ($aResultats  === null) {

  $aResultats  = Loto::getResultats();

  $oCache->set($sCleCache, $aResultats , 86400);
}
```

Listing 12

```
$oRequete = $this->modelsManager->createQuery('SELECT *
            FROM NovaMooc\Models\Utilisateurs
              WHERE prenom LIKE :prenom:');
$oRequete->cache(
 [
   'key'      => 'utilisateurs_phql',
   'lifetime' => 14400,
 ]
);
```

Listing 13

```
<?php

namespace Psr\Log;

/**
 * Describes a logger instance.
 *
 * The message MUST be a string or object implementing __toString().
 *
 * The message MAY contain placeholders in the form: {foo} where foo
 * will be replaced by the context data in key "foo".
 *
 * The context array can contain arbitrary data, the only assumption that
 * can be made by implementors is that if an Exception instance is given
 * to produce a stack trace, it MUST be in a key named "exception".
 *
 * See https://github.com/php-fig/fig-standards/blob/master/accepted/
                                                    PSR-3-logger-interface.md
 * for the full interface specification.
 */
interface LoggerInterface
{
  /**
   * System is unusable.
   *
   * @param string $message
   * @param array $context
   * @return void
   */
  public function emergency($message, array $context = array());

  /**
   * Action must be taken immediately.
   *
   * Example: Entire website down, database unavailable, etc. This should
   * trigger the SMS alerts and wake you up.
   *
   * @param string $message
   * @param array $context
   * @return void
   */
  public function alert($message, array $context = array());
```

Abb. 3: Logo der Website PHP-Fig

rekt angegeben werden, dass das Ergebnis der Abfrage in den Cache gehen soll. Listing 12 zeigt dafür ein Beispiel.

Dank der Cachefunktion ist es nicht nötig, zu prüfen, ob der Cache abgelaufen ist oder ob die Anfrage neu gestartet werden muss. Alles wird vom System verwaltet.

PSR – die PHP-Standard-Empfehlungen

Der Schöpfer von Phalcon, Andrés Gutiérrez, legte von Beginn an großen Wert darauf, die PHP Standard Recommendations (PSR) so weit wie möglich zu respektieren. Diese Empfehlungen sind auf der Website PHP-Fig [6] verfügbar und werden von einer Reihe anerkannter Framework-Experten herausgegeben. Die Community hat 2013 in einer Abstimmung beschlossen, Phalcon in den PSR-Katalog aufzunehmen. Seither ist Phalcon dort als Standard vertreten, zusammen mit anderen Frameworks wie PrestaShop, Yii, CakePHP, während andere Technologien wie Laravel oder Symfony sich wieder verabschiedet haben.

Der Begriff FIG steht für Framework Interoperability Group. Das Ziel der Gruppe besteht darin, darauf hinzuwirken, dass Frameworks gleiche Schnittstellen für ihre Komponenten verwenden. Zum Beispiel schlägt der „PSR 3 – Logger Interface“ spezifische Funktionsnamen und deren Verwendungen vor. Listing 13 zeigt einen Auszug aus dem PSR 3.

Frameworks, die die PSR-Empfehlungen berücksichtigen, sind für Entwickler besser zugänglich. Darüber hinaus haben Tools von Drittanbietern weniger Code zu schreiben, um sich mit diesen Frameworks zu verbinden, da sie nur die Standardnamen verwenden müssen. Die Phalcon-Entwicklergruppe hat sich vorgenommen, diese PSRs so weit wie möglich einzuhalten. Um das zu erreichen, erfordert Version 4 von Phalcon eine zusätzliche Erweiterung namens php-psr. Diese Erweiterung erlaubt es jedem Framework, die von PHP-Fig empfohlenen Schnittstellen zu verwenden. Gegenwärtig unterstützt Phalcon PSR-3, PSR-0, PSR-4, PSR-7, PSR-11, PSR-13 und PSR-17. Diese PSRs sind in der Onlinedokumentation von Phalcon ausführlich beschrieben [7].

Die Nachteile

Phalcon hat viele Vorteile, aber auch einige Schwächen. Das erste, was man wissen muss: Um Phalcon in der Produktion einzusetzen, muss man Hand an den Server legen – oder den eigenen Hostingprovider kontaktieren. Was Phalcon so mächtig macht, ist, dass es als Erweiterung geschrieben ist. Gleichzeitig ist das aber auch ein Schwachpunkt, denn Entwickler müssen in der Lage sein, Phalcon zu installieren und direkt zu PHP hinzuzufügen. Wenn der Produktionsserver gemeinsam genutzt wird, kann das kompliziert sein. Ist das der Fall, sollten Sie sich an Ihren Hostingprovider wenden und

fragen, ob es möglich ist, Phalcon zu unterstützen. Das Zünglein an der Waage könnte hier sein, dass Phalcon mittels PECL installiert werden kann. Das verleiht Phalcon eine gewisse Glaubwürdigkeit bei Hostingprovidern und Systemadministratoren. Auf diese Weise ist Phalcon nicht länger ein beliebiges, von der Community nicht validiertes Open-Source-Framework, sondern ein von PECL abgesegnetes Projekt.

Die Community

Die Phalcon-Community (**Abb. 4**) hat sich auf das konzentriert, was sie am besten kann, nämlich Technik und Engineering. Dabei ist der Aspekt der Kommunikation (Meetups, Veranstaltungen, Artikel etc.) bisher etwas zu kurz gekommen. Das hat sicherlich dazu beigetragen, dass es dem Projekt etwas an Sichtbarkeit bei Entwicklern fehlt. Die Folge: Nur wenige Menschen kennen Phalcon, nicht einmal dem Namen nach. Ich hoffe, dieser Artikel trägt dazu bei, dass sich das ändert und Phalcon einen festen Platz auf dem Markt der Frameworks einnimmt.

Seit ich dem Phalcon-Team beigetreten bin, ist mir Folgendes aufgefallen: Sobald sich ein neuer Entwickler für Phalcon zu interessieren beginnt, kommt er, um ein paar Fragen auf Discord oder im speziellen Forum zu stellen. Wenn sich der Entwickler ein paar Wochen später wieder meldet, ist er begeistert und hat Phalcon vollständig adoptiert.

Heute kann Phalcon auf fast 10 Jahre Erfahrung zurückblicken. Die Phalcon-Community ist klein, aber treu. In dieser Konstellation ist es schwierig, in großem Umfang neue Features für das Framework zu schreiben. Aber das hat auch einige positive Auswirkungen:

- Wir priorisieren die wichtigsten Aufgaben.
- Neue Features werden erst nach sorgfältiger Überlegung und Prüfung geschrieben.
- Wir können neue Beiträge problemlos berücksichtigen. Die Administratoren nehmen sich die Zeit, alle Vorschläge zu lesen und schnell darauf zu reagieren.

Wer schon einmal die Aufgabe hatte, ein trendiges Open-Source-Projekt zu verwalten, kennt das vielleicht: Administratoren sind oft mit etlichen Codevorschlägen überlastet, und es dauert einige Zeit, bis neuer Code gelesen und validiert werden kann. Mit Phalcon hatte ich dieses Problem nie. Die Administratoren sind sehr herzlich und helfen jedem, einen Beitrag zu leisten.

Da ich zum Phalcon-Team gehöre, kann ich auch Sie willkommen heißen und Sie bei Ihren ersten Schritten begleiten. Wenn Sie Angst haben, einen Fehler zu machen, kann ich Sie beruhigen: Das Projekt ist gut gesichert, und viele Aufgaben sind automatisiert. Dank GitHub Actions und der vielen Tests wird jeder neu gesendete Code ausgiebig geprüft. Der Code wird auf Linux, MacOS und Windows getestet, und wenn ein Fehler auftritt, muss er korrigiert werden, sonst kann

Abb. 4: Die Phalcon Community auf GitHub

der Code nicht in Phalcon integriert werden. Darüber hinaus ist eine Validierung durch zwei erfahrene Entwickler erforderlich. Also keine Angst!

Fazit

Ich hoffe, dieser Artikel hat Lust auf mehr gemacht. Wenn Sie sich einmal eine konkrete Phalcon-Anwendung anschauen wollen, empfehle ich das GitHub-Projekt unter [8]: Enthalten ist dort eine GitHub Action, die es ermöglicht, das Projekt mit Cypress zu testen und ein Testvideo zu generieren. Um das Video zu sehen, gehen Sie einfach auf die Registerkarte Action und klicken auf den letzten Test. In der Sektion Artefakte befindet sich eine herunterladbare Zip-Datei mit dem Video. Viel Spaß mit Phalcon!

Dieser Artikel erschien zuerst in dem französischen Magazin „Programmez“, Ausgabe 07-08.2020.

Jérémy Pastouret ist der Autor des Buchs: „Phalcon – Développez des applications web complexes et performantes en PHP“ und Redakteur der Webseite Les Enovateurs. Jérémy gehört zum Kernteam von Phalcon und trägt zu Open-Source-Projekten bei. Als Unternehmer gründete er Unlock My Data und vor kurzem die Plattform Garwen.

https://les-enovateurs.com

Links & Literatur

[1] https://blog.phalcon.io/post/interview-with-phalconphp-creator-andres-gutierrez

[2] https://link.medium.com/vfycm0Eff5

[3] https://phptherightway.com

[4] https://blog.phalcon.io/post/benchmarking-phalcon

[5] https://www.amazon.de/-/en/J%C3%A9r%C3%A9my-Pastouret/dp/240902274X

[6] https://www.php-fig.org

[7] https://docs.phalcon.io

[8] https://github.com/les-enovateurs/phalcon-nova-mooc

Vorschau auf die Ausgabe 3.2021

Das große TypeScript Cheat Sheet

TypeScript kann ganz schön kompliziert werden. Die Sprache bietet zahlreiche Optionen, die nicht alle Entwickler bereits kennen. Darum hat Peter Kröner die wichtigsten Sprachfeatures in einem Cheat Sheet zusammengefasst. Von den Basics bis zu fortgeschrittenen Features, die nur mit Vorsicht angewendet werden sollten, finden Sie hier alles auf einen Blick.

Redwood: Fullstack im JAMStack?

Es gibt Buzzwords, die einem wieder und wieder begegnen. Fullstack gehört dazu, JAMStack ebenso. RedwoodJS gelingt es jedoch, beide Konzepte miteinander zu vereinen. Das Framework bietet einige spannende Möglichkeiten für die Integration eines modernen Technologiestacks in die Webentwicklung. In der kommenden Ausgabe des Entwickler Magazins erwartet Sie eine Einführung in RedwoodJS.

Aus aktuellem Anlass kann es zur Verschiebung von Artikeln kommen.

Die nächste Ausgabe erscheint am 24. Februar 2021

Querschau

windows .developer — **Ausgabe 3.2021** | www.windowsdeveloper.de

Pattern Matching in C# 9.0

Anomalien aufdecken mit Azure Cognitive Services

Rechtssichere Einbindung von Open-Source-Code

Javamagazin — **Ausgabe 3.2021** | www.javamagazin.de

Auf pythonischen Spuren: Schließt Tribuo den Feature-Gap zwischen Python und Java?

Nützlich für den Alltag: Zehn Must-have-Java-Bibliotheken

Chaos Engineering bei der DB: Grundlagen, Prinzipien, Mehrwert und Praxis

PHPmagazin — **Ausgabe 2.2021** | www.phpmagazin.de

MoneyPHP: Internationale Transaktionen leicht gemacht

CaptainHook: Git Hooks lokal verwenden

PHP trifft Kubernetes: Aufbruch in die Container-Welt

Inserenten

Impressum

Verlag:
Software & Support Media GmbH

Anschrift der Redaktion:
Entwickler Magazin
Software & Support Media GmbH
Schwedlerstraße 8
D-60314 Frankfurt am Main
Tel. +49 (0) 69 630089-0
Fax. +49 (0) 69 630089-89
redaktion@entwickler-magazin.de
www.entwickler-magazin.de

Redaktion: Jan Bernecke, Ann-Cathrin Klose, Dominik Mohilo, Hartmut Schlosser
Redaktionsassistenz: Daniel Zuzek
Chefin vom Dienst/Leitung Schlussredaktion: Frauke Pesch
Schlussredaktion: Jonas Bergmeister, Anne Lorenz
Leitung Grafik und Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Dominique Kalbassi, Bianca Röder, Maria Rudi, Sibel Sarli, Sandra Schalk, Theresa Radig, Vincent Schlothauer, Michael Schütze

Autoren dieser Ausgabe:
Elena Bochkor, Martin Boßlet, Michael Hofmann, Thorben Janssen, Christian Kaltepoth, Dr. Veikko Krypczyk, Daniel Mies, Martin Mohr, Andreas Möller, Sandra Parsick, Jérémy Pastouret, Stephan Rauh, Golo Roden, Alexander Rudolph, René Schröder, Andreas Schröpfer, Dr. Holger Schwichtenberg, Falk Sippach, Karsten Sitterberg, Sebastian Springer, Jochen Stärk, Marc Teufel, Timo Trautmann, Michael Vitz, Wolfgang Weigend, Tim Zöller

Anzeigenverkauf:
Entwickler Magazin
Anika Stock
Software & Support Media GmbH
Tel.: +49 (0)69 630089-22
E-Mail: anika.stock@sandsmedia.com

Es gilt die Anzeigenpreisliste Mediadaten 2021

Pressevertrieb:
PressUp GmbH
Tel +49(0) 4041448-411
www.pressup.de

Druck:
Westdeutsche Verlags- und Druckerei GmbH
Kurhessenstraße 4 – 6
64546 Mörfelden-Walldorf

Abonnement und Betreuung:
Leserservice Entwickler Magazin
65431 Eltville
Tel.: +49 (0)6123 9238-239
Fax.: +49 (0)6123 9238-244
entwicklermagazin@vuservice.de

Abonnementpreise der Zeitschrift:

Inland:	10 Ausgaben	€ 89,00
Studentenpreis:	10 Ausgaben	€ 78,00
Ausland:	10 Ausgaben	€ 104,00
Stud. Ausland:	10 Ausgaben	€ 91,00

Preise zzgl. 7% MwSt.

Abonnementpreise der Zeitschrift:
(inkl. Profi-DVD):

Inland:	10 Ausgaben u. DVD	€ 130,20
Studentenpreis:	10 Ausgaben u. DVD	€ 111,20
Ausland:	10 Ausgaben u. DVD	€ 140,20
Stud. Ausland:	10 Ausgaben u. DVD	€ 121,20

ISSN: 1619-7941

Erscheinungsweise: Zehn Ausgaben pro Jahr

INTERNET OF THINGS CONFERENCE
HYBRID EDITION

JUNE 21 – 23, 2021 | MUNICH OR ONLINE
EXPO: JUNE 22 – 23, 2020

BUILD. CREATE. CONNECT. THINGS

ONLY UNTIL MARCH 11
- ✓ Workshop Day for free
- ✓ Save more than € 500
- ✓ 10% Team Discount

Internet of Things Conference brings together experts from all areas of the IoT product lifecycle. The conference offers a unique mix of topics including design, product development and IoT business models with a focus on software implementation.

CONFERENCE TRACKS

BUSINESS & STRATEGY

DEVELOPMENT & TOOLS

iotconf iotcon IoT Conference

iotcon.de

Presented by: e.academy entwickler magazin | Organizer: S&S Media Group

ML CONFERENCE

HYBRID EDITION

ONLY UNTIL MARCH 11
- ✓ Workshop Day for free
- ✓ Save more than € 500
- ✓ 10 % Team Discount

THE CONFERENCE FOR MACHINE LEARNING INNOVATION

JUNE 21 – 23, 2021 | MUNICH OR ONLINE

EXPO: JUNE 21 – 21, 2021

CONFERENCE TRACKS

- Machine Learning Business & Strategy
- Machine Learning Principles
- Advanced Machine Learning Development
- Tools, APIs & Frameworks

mlconference

mlconference.ai

Presented by: entwickler magazin Powered by: e.academy Organiser: S&S Media Group