Java aktuell

iJUG Verbund
www.ijug.eu

Progressive Web Apps
PWAs und ihre Vorteile in der Praxis

Flutter
Von der Entscheidung bis zur Entwicklung

Geteilte Hologramme
Shared Augmented Reality im Überblick

DIE ZUKUNFT IST MOBILE

Ausgabe 03/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 03

adesso business. people. technology.

Von Anfang an Teil des Java-Teams!

Entwickelt bereits kluge IT-Lösungen bei adesso:
Ihr neuer Kollege Kristof Hierath | Software Engineer

SOFTWARE DEVELOPMENT@adesso

Sie wollen dort einsteigen, wo Zukunft programmiert wird? Dann sind Sie mit einem Start in unserem Java-Team bei adesso genau richtig. Gemeinsam setzen wir herausfordernde Projekte für unsere Kunden um. Dafür brauchen wir Menschen, die Lust haben, ihr Wissen, ihre Talente und ihre Fähigkeiten einzubringen.

Planen und realisieren Sie in interdisziplinären Projektteams anspruchsvolle Anwendungen und Unternehmensportale auf Basis von Java/JavaScript-basierten Technologien als

→ **(Senior) Software Engineer (w/m/d) Java**
→ **Software Architekt (w/m/d) Java**
→ **(Technischer) Projektleiter Softwareentwicklung (w/m/d) Java**

CHANCENGEBER – WAS ADESSO AUSMACHT

Kontinuierlicher Austausch, Teamgeist und ein respektvoller, anerkennender Umgang sorgen für ein Arbeitsklima, das verbindet. So belegen wir nach 2016 auch 2018 den 1. Platz beim Wettbewerb „Deutschlands Beste Arbeitgeber in der ITK“!

Mehr als 650 Software Engineers Java bei adesso, über 120 Schulungen und Weiterbildungen – zum Beispiel in Angular2 oder Spring Boot – sowie ein Laptop und ein Smartphone ab dem ersten Tag warten auf Sie!

GREAT PLACE TO WORK® 2018 Beste Arbeitgeber™ ITK

IHRE BENEFITS – WIR HABEN EINE MENGE ZU BIETEN:

- Welcome Days
- Choose your own Device
- Weiterbildung
- Events: fachlich und mit Spaß
- Sportförderung
- Mitarbeiterprämien
- Auszeitprogramm

Es wird Ihnen bei uns gefallen! Mehr Informationen auf **www.karriere.adesso.de**.
Olivia Slotta aus dem Recruiting-Team freut sich auf Ihre Kontaktaufnahme:
adesso SE | Leona Demiri | T +49 231 7000-7100 | jobs@adesso.de

Liebe Leser der Java aktuell,

in dieser Ausgabe liegt der Fokus auf der Entwicklung von mobilen Anwendungen. Da es sich dabei um ein weites Feld handelt, schauen wir auch über den Tellerrand von reinem Java hinaus. Den Beginn macht der Artikel zu Progressive Web Apps (PWAs) von Yves Schubert ab Seite 16. Darin zeigt er uns, was die technischen Grundlagen von PWAs sind und wie damit beispielsweise Offline-Fähigkeit realisiert werden kann. Direkt im Anschluss berichtet Tobse Fritz, wie sich mit dem Vaadin-Framework ebensolche modernen Webanwendungen umsetzen lassen – entweder mit purem Java oder mit einem Mix aus TypeScript, HTML und Web Components.

Einen gänzlich anderen Weg zur eigenen Mobile-App beschreitet René Jahn im Artikel "Es Fluttert gewaltig" vor. Darin stellt er ab Seite 30 nicht nur das JVx-Framework vor, sondern berichtet vor allem vom Entscheidungsprozess bei der Technologieauswahl für die eigene App. Zum Zuge kam das relativ neue Framework "Flutter", welches auf der Programmiersprache Dart aufsetzt, die gewisse Ähnlichkeit mit Java besitzt. Damit lassen sich native Apps plattformübergreifend mit einer einzigen Codebasis realisieren. Ähnlich ist auch der Ansatz, den Markus Schlichting in "Cross-Plattform Apps mit NativeScript" beschreibt. Auch hier werden aus einer Codebasis mithilfe von NativeScript Apps für verschiedene Plattformen realisiert. Die Basis dafür bilden TypeScript und Angular. Damit richtet sich NativeScript an Webentwickler, die ihr Know-how auch für die mobile Welt nutzen wollen.

Auch abseits von Mobile bietet diese Ausgabe spannende Thematiken. Wie zum Beispiel im Praxisbericht zum Thema Teamentwicklung von Stephan Doerfel und Simon Krackrügge, in dem die beiden einen Blick auf die Team-Uhr werfen. Diesen finden Sie auf den Seiten 39 bis 43. Passend dazu auch der Artikel "Meetings sind giftig – Zeit für ein Detox!" von Kerstin Gronauer und Timothée Bourguignon ab Seite 55. Chris Papenfuß gibt uns einen Einblick in Hologramme und Shared Augmented Reality und Stefan López Romero zeigt uns, wie funktionale Programmierung in Kotlin gemacht wird. Zum Abschluss berichtet Dr. Carola Lilienthal im Interview, welche besondere Rolle Entwickler und Architekten in der technologischen Entwicklung der Welt einnehmen.

Am Anfang dieser Ausgabe finden wir wie gewohnt interessante Neuigkeiten aus der gesamten Java-Welt im Java-Tagebuch. Markus Karg gibt uns in der Eclipse Corner einen Überblick über die aktuellen Entwicklungen des Jakarta-EE-Projekts. Unsere unbekannte Kostbarkeit, die wir näher beleuchten ist diesmal die String-Klasse.

Selbst in Zeiten der Corona-Krise beweist die Java-Community viel Kreativität und Zusammenhalt: nur wenige Tage nach der Absage der JavaLand 2020 haben sie die "CyberLand" auf die Beine gestellt. Falk Sippach berichtet ab Seite 14 von der Online-Konferenz.

Viel Spaß beim Lesen!

Ihr

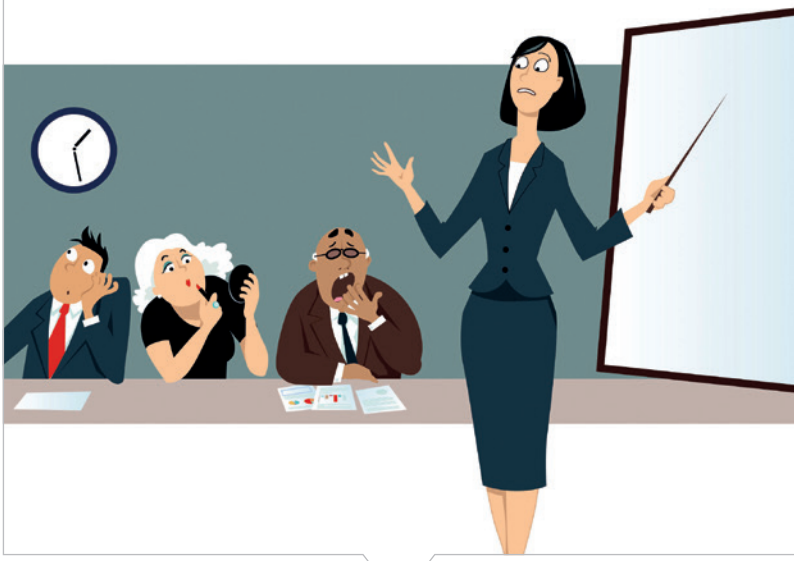
Manuel Mauky
Redaktionsbeirat Java aktuell

Inhalt

Kreativität trotz Krise: die CyberLand

So erstellen Sie eine Webanwendung mit dem Vaadin-Framework

Mit NativeScript zur Webanwendungen für unterschiedliche Plattformen



Schluss mit langweiligen, sinnlosen Meetings!

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

2. Dezember 2019

Graal, Wasm, was noch...

Wer noch nicht von Wasm gehört hat: Es geht um WebAssembly, seit Dezember ein W3C-Standard als „vierte Sprache für das Web". Zitat des W3C-Konsortiums: „The arrival of WebAssembly expands the range of applications that can be achieved by simply using Open Web Platform technologies."

Wasm-Code lässt sich aus anderen Sprachen wie Java generieren und dann performant (und sicher) in gängigen Browsern ausführen – oder auch außerhalb. WebStart, anyone? Nicht zufällig arbeitet Oracle nun an einer weiteren Facette für die GraalVM: GraalWasm. Das Ganze ist noch in einer frühen Phase, hat aber Potenzial. Der Code liegt zusammen mit dem Rest der GraalVM auf GitHub *[1]* und die Entwickler von Oracle bitten explizit um Contributions. Die GraalVM hat bereits jetzt viel in Bewegung gebracht. Auch das Thema „Native Images" beschäftigt nicht nur das Quarkus-Projekt. Pivotal arbeitet nach eigener Aussage auch eifrig an vollständiger Unterstützung. Auf GitHub zu sehen ist zumindest schon mal ein Entwurf: spring-projects-experimental/spring-graal-native.

6. Dezember 2019

Kotlin 1.4 und weiter

Kotlin 1.4, das für Frühling 2020 angekündigt ist, soll nur kleinere Änderungen an der Sprache mitbringen, stattdessen soll die „Overall Experience" verbessert werden. Damit sind zum Beispiel Performanceverbesserungen wie beim Import oder Kompilieren gemeint oder auch bei der Code Completion in diversen IDEs. Wesentlicher Bestandteil ist ein neues „Frontend" für den Compiler – der erste Schritt beim Kompilieren, der für das Parsen des Codes, Namensauflösung etc. zuständig ist und auch innerhalb der IDEs abläuft. Hier wird laut JetBrains aktuell die meiste Zeit benötigt und besteht daher das größte Einsparpotenzial. Außerdem wird das Thema Multiplattform weiter vorangetrieben, mit dem gemeinsamen Kotlin Library Format (KLib).

17. Dezember 2019

Lagom 1.6

Das Microservices-Framework Lagom, basierend auf dem Play-Framework und der Aktoren-basierten Laufzeitumgebung Akka, ist in Version 1.6 freigegeben worden. Unter anderem kann jetzt das neue typisierte Aktoren-API „Akka Typed" verwendet werden sowie „Akka Persistence Typed", das das Lagom-eigene Persistence-API ablösen soll. Unterstützte Sprachen sind Scala 2.12 und Java 11 (beziehungsweise auch Java 8).

20. Dezember 2019

Webbrowser-Programmierung in Java

Da ich vor ein paar Tagen über Wasm und Webentwicklung geschrieben hatte, will ich andere Ansätze, die auch im Java-Ökosystem verfolgt werden, nicht unberücksichtigt lassen: zum Beispiel die „Transpilierung" in JavaScript. Heise liefert einen Einstiegsartikel, der interessant für diejenigen ist, die sich noch mit dem GWT oder J2CL (Java to Closure Transpiler) auseinandersetzen wollen *[2]*.

2. Januar 2020

Neue Marke: Die Jakarta EE Ambassadors

Die Java EE Guardians, die selbsternannten unabhängigen Hüter des Java-Standards, haben ihr „Rebranding" abgeschlossen. Gegründet in „kämpferischen" Zeiten, als Oracle seine Java-Enterprise-Strategie komplett umgekrempelt hat, passen sie sich nun den neuen Gegebenheiten an und werden zu den Jakarta EE Ambassadors. Neue Website und neues Logo inklusive *[3]*.

10. Januar 2020

Microservices-Umfrage

Das JRebel-Team hat eine (nicht repräsentative) Umfrage unter Entwicklern zur Microservices-Nutzung durchgeführt. Bei 400 Rückmeldungen gaben etwas mehr als die Hälfte an, Microservices für ihre Hauptanwendungen zu nutzen, für etwas mehr als ein Viertel ist es ein Monolith. Spring Boot spielt prozentual in einer eigenen Liga, aber interessanter ist, dass fast 60 Prozent der Projekte immer noch bei Java 8 feststecken, während 23 Prozent Java 11 nutzen und sechs Prozent Java 12 oder später. Die Hürde wird als zu hoch angesehen beziehungsweise die Vorteile als zu gering – und Unternehmen haben es sich mit kostenpflichtigem Support von Oracle und anderen bequem gemacht. Mal sehen, wann die Masse der Entwickler anfängt, Bewerbungen zu schreiben, um nicht technologisch abgehängt zu werden...

Eine Sache gibt mir aber besonders zu denken: Von den Antwortenden mit Microservices-Fokus wurde das Aufsetzen der lokalen Entwicklungsumgebung als größte Einzel-Herausforderung angesehen (in Summe nur übertroffen von der Performance des Gesamtsystems plus der Einzel-Services). Gibt es irgendwo die Anforderung, dass das Gesamtsystem auch auf dem Laptop laufen muss? Oder was ist mit Docker, Testcontainers, Citrus und Co.? Testdaten sind sicher ein Thema, aber auch das lässt sich doch lösen. Das ist jedenfalls ein wichtiger Punkt für die nächsten Projekte. Früher haben wir uns über „runs on my machine" lustig gemacht. Jetzt sollte es wohl heißen: „If I can't run it on my machine, how can I possibly push it?"

11. Januar 2020

Dirty Code

Für alle Entwickler, die nicht gerne aufräumen: Ihr seid nicht allein! Hier eine kleine Spitze gegen das allgegenwärtige CleanCode-Mantra beziehungsweise dessen unreflektierte Auswüchse *[4]*. Der Autor arbeitet unter anderem bei React.js und Redux mit, die Beispiele sind daher in JavaScript, aber das ist offensichtlich auch nicht mehr das Paradies für Messies, das es mal war.

14. Januar 2020

Neues von Java 14

Für Java 14 (Release-Datum 17. März) wurde mehr als pünktlich die „Rampdown-Phase Zwei" eingeleitet, es geht nur noch um Bug Fixes der Prioritäten eins und zwei. Ein paar der Features, oder genauer JEPs, sind in der letzten Ausgabe schon erwähnt worden, insgesamt sind es 16 (wobei auch das Entfernen von Features als JEP verwaltet wird, ich komme gleich darauf zurück).

Neu hinzugekommen ist noch das „Foreign-Memory Access API", allerdings erst mal im Inkubator-Modul (sprich: es soll erstmal Feedback eingeholt werden, das API kann dann in späteren Releases auch „breaking changes" erhalten oder sogar entfernt werden). Mit JEP 349 kommt ein Event-Streaming für den Java Flight Recorder. „Helpful NullPointer Exceptions" sollen Entwickler in Zukunft zu noch mehr Faulheit beim Prüfen von Vorbedingungen verleiten (meine Interpretation, nicht das offizielle Ziel von JEP 358). Und mein geliebter „Concurrent Mark Sweep" Garbage Collector, der mir viele schöne Tuning-Stunden verschafft hat, wird aus Java 14 und zukünftigen Versionen verschwinden.

16. Januar 2020

Pläne für Jakarta EE 9

Die Pläne für Jakarta EE 9 sind verabschiedet, nachfolgend die wichtigsten Bestandteile. Die Top-Level Package-Namen aller APIs werden auf einen Schlag von javax auf jakarta umgestellt („Big Bang"-Ansatz), um Oracles Forderung direkt umzusetzen und nicht jahrelang über jedes API mit Änderungsbedarf einzeln zu diskutieren. Nicht mehr gewünschte Spezifikationen sollen entfernt werden (Release Jakarta XML Registries, Jakarta XML RPC, Jakarta Deployment und Jakarta Management sowie die „Distributed Interoperability" für EJB 3.2).

Wir ahnen es schon: Abwärtskompatibilität steht nicht im Fokus. Die APIs selbst müssen in Java 8 kompiliert werden, Implementierungen müssen allerdings Java-11-Kompatibilität nachweisen (Java 8 ist hier optional). Neue Features gibt es nicht, die werden dann erst mit Jakarta 10 kommen. Allerdings werden ein paar APIs übernommen, die aus Java SE herausgefallen sind (außer Jakarta Activation sind diese aber optional). Details sind online zu finden *[5]*.

21. Januar 2020

Quarkus und MicroProfile 3.2

Das direkt vor Weihnachten freigegebene Quarkus 1.1 ist kompatibel mit der ebenfalls neuesten MicroProfile-Version 3.2 von November. In den letzten Monaten ist laut Ken Finnigan, der als Engineer in beiden Projekten mitarbeitet, viel Arbeit in die Integration der MicroProfile TCKs in den Quarkus-Build geflossen, ebenso wie in die fälligen Bug Fixes – auch in den eingebundenen Bibliotheken wie RESTEasy oder SmallRye.

26. Januar 2020

jSpirit

Die jSpirit – eine kleine, aber feine Java Unconference in einer Destillerie am Schliersee – war wieder eine Reise wert. Nicht nur wegen der Leute und der vielfältigen Themen von Soft Skills über DevSecOps und „die Cloud" bis zu JVM-Details wie Project Loom. Sondern insbesondere, weil auf dieser Art von Konferenz ein Thema „Wie funktioniert X?" lauten kann und der Titel dann auch genau so gemeint ist. Kein „Ich erkläre euch das jetzt mal", sondern: „Ich weiß es auch nicht genau, lasst es uns gemeinsam herausfinden." Ein nach außen sichtbares Ergebnis ist der erste Entwurf eines „Open-Source-Manifesto" *[6]*, das ein Leitfaden für das Durchführen von und die Teilnahme an Open-Source-Projekten sein soll und logischerweise selbst Open Source ist.

29. Januar 2020

JDK 15 EA Build schon verfügbar

Der Release Train rollt immer weiter. Java 14 ist noch nicht fertig, aber Early Access Builds des JDK 15 sind bereits verfügbar *[7]*. Einige der anvisierten Features sind ein „Light-Weight JSON API", „Generics over Primitive Types" und „Enhanced Enums" (mit Generics und besserer Typprüfung).

29. Januar 2020

MicroProfile Starter "produktionsreif"

Der schon früher erwähnte MicroProfile Starter, nach Vorbild des Spring Starter, ist jetzt als „produktionsreif" deklariert worden. Für neue MicroProfile-Projekte also spätestens jetzt start.microprofile.io nutzen!

8. Februar 2020

Wieviel CDI darf's denn sein?

Tim Zöller von der JUG Mainz hat eine heiße Diskussion in der MicroProfile-Community losgetreten. Grund dafür ist sein Tweet über die unvollständige CDI-Implementierung in Quarkus. Wenn Quarkus MicroProfile implementiert und CDI zwar nicht innerhalb

des MicroProfile spezifiziert, aber ein „required API" ist – kann die CDI-Implementierung dann unvollständig sein (zum Beispiel keine „Implicit Bean Discovery" unterstützen)? Da scheint es (noch) keinen Konsens zu geben. Quarkus-Entwickler Greene möchte keinen unnötigen Ballast, sondern nur den für MicroProfile erforderlichen Funktionsumfang von CDI unterstützen. Die anderen pochen auf Zuverlässigkeit und Portabilität. Die Diskussion wird uns sicher auf die JavaLand begleiten. Vielleicht gibt es dann sogar eine Lösung, einige der Verantwortlichen sind ja vor Ort!

Referenzen:

[1] *github.com/oracle/graal*
[2] *https://bit.ly/2SdUkFb*
[3] *jakartaee-ambassadors.io*
[4] *overreacted.io/goodbye-clean-code*
[5] *https://bit.ly/2vVlSVI*
[6] *github.com/opensource-manifesto*
[7] *jdk.java.net/15*

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' eclipse-Corner

Es geht endlich voran! Nach einer weiteren längeren Atempause über die Weihnachtsfeiertage ist nun in einigen Repositories und in den entsprechenden Diskussionsforen Aktion zu verzeichnen. Und was sich da liest, klingt doch sehr vielversprechend: Die Eclipse Foundation (EF) hat den Plan für Jakarta EE 9 veröffentlicht *[1]*. Es wird definitiv im Sommer 2020 erscheinen und es bringt – äh… keinerlei neuen Features. Die kommen dann später und so ganz kennt man die noch gar nicht. Aber das ist (ausnahmsweise) kein Grund zur Trauer. Denn tatsächlich will man bereits kurz darauf Jakarta EE 10 veröffentlichen, und dieses ist dann (endlich) vollgepackt mit Neuem!

Tatsächlich steht sich derzeit die EF als Organisation selbst mehr im Weg als die üblicherweise von mir gescholtenen Hersteller. So hat man beispielsweise kürzlich dem in der Community sehr geschätzten Romain Manni-Bucau, seines Zeichens Kern des JSON-B-Spezifikationsteams, aus „formellen Gründen" seine Committer-Rechte im GitHub Repository entzogen, weil er zu einem Arbeitgeber, der keine 10.000 Euro Mitgliedsbeitrag zahlen wollte, gewechselt hat. Das war weder böser Wille noch Erpressung – es steht nur dummerweise so in der Satzung beziehungsweise den Jakarta-EE-Führungsgremien. Wie die EF unter anderem in einer Presseerklärung *[2]* schreibt, arbeite man bereits

unter Hochdruck mit der Rechtsabteilung an einer Lösung, damit Jakarta EE besser durch einzelne Menschen statt (zahlende) Unternehmen unterstützt werden kann. Aktuell ist noch keine Lösung gefunden worden.

Auch der gut gemeinte, kurzfristig angesetzte Umzug aller beteiligten Projekte auf eine eigene Infrastruktur von Sonatype, um deren Staging-Server *„oss.sontype.org“* zu entlasten, hat unerwartet Ressourcen gebunden und hätte nicht unbedingt zum aktuellen Zeitpunkt stattfinden müssen. Auch im Kern positive Entwicklungen können in der Summe nun mal zu etwas sehr Schlechtem – weiterem Zeitverzug – führen, wenn man sie zum falschen Zeitpunkt entscheidet oder deren Umsetzung nicht sinnvoll koordiniert. Doch wer kennt das nicht aus eigenen Projekten?

Doch zurück zu Jakarta EE 9. Es wird nun also definitiv im Sommer 2020 erscheinen und auf dem Inhalt von Jakarta EE 8 basieren. Der maßgebliche Unterschied ist also nicht, was man mit EE 9 an APIs erhält. Vielmehr bereitet EE 9 den entscheidenden technischen Unterbau, um zukunftssicher zu bleiben. EE 8 basierte nämlich auf Java SE 8 und dieses hatte gegenüber aktuellen JREs zwei entscheidende Unterschiede: Es kannte keine Module und es brachte ein paar APIs mehr mit als moderne JREs. Wo das Problem liegt? Moderne Anwendungen möchte man in Java 11 bis 14 schreiben, nicht in Java 8. Man möchte Module einsetzen, neue Sprachfeatures oder neue Garbage Collectors. All das ist zum Beispiel in Java 11 bis 14 vorhanden. Aber leider verspricht Jakarta EEs Abwärtskompatibilität einiges, das Java SE seit Version 11 nicht mehr kennt – beispielsweise das Activation API. Schlimmer noch, einige durchaus aktuelle Teile von Jakarta EE benötigen dieses sogar „unter der Haube“. Die Lösung: Jakarta EE 9 übernimmt diese APIs von Java SE, bringt sie also selbst mit. Das ist für die Hersteller ein ziemlicher Aufwand, denn bislang mussten sie sich nicht darum kümmern. Sie müssen sogar darüber hinaus diese lange nicht gepflegten Produkte in die moderne Post-SE-8-Welt überführen, unter anderem also modularisieren. Und sie müssen das dann auch noch pflegen. Aber das ist noch nicht alles. Dazu kommt ja noch der sogenannte „Big Bang“, also das zeitgleiche Umstellen aller beteiligten APIs auf das neue Package jakarta.* statt javax.*. Im Gegenzug hat man sich daher darauf verständigt, dass neue Features erst mal kein Thema sind und dass gegenüber EE 8 auch etwas fehlen darf: beispielsweise IIOP. Ja, Alarmglocken an, RMI over IIOP (also die Unterstützung für CORBA) entfällt! Wer das noch benutzt, zum Beispiel im Rahmen eines unter Long Term Support stehenden Bestandsprojektes, muss nun entweder hoffen, dass sein Serverhersteller der Wahl dies weiterhin auf eigene Kosten pflegt (und diese Kosten an seine Kunden weitergibt), oder muss die Altanwendung umbauen (beispielsweise von RMI auf JAX-WS und somit von IIOP auf HTTP). Aber mal ganz ehrlich: Damit war doch sowieso irgendwann mal zu rechnen!

Referenzen

[1] *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9ReleasePlan*

[2] *https://blogs.eclipse.org/post/tanja-obradovic/jakarta-ee-community-update-february-2020*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Unbekannte Kostbarkeiten des SDK Heute: Strings

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Strings sind integraler Bestandteil jeder Programmiersprache, da die meisten Daten eines jeden Programms häufig aus Strings bestehen. In Java sind Strings zum einen Instanzen einer ganz normalen Klasse, zum anderen nehmen sie eine besondere Stellung ein, werden explizit in der Sprachbeschreibung genannt und hocheffizient implementiert. Wir wollen uns in dieser Ausgabe unserer unbekannten Kostbarkeiten dieser Besonderheit von Strings widmen, aber auch die API-Erweiterungen der letzten Java-Versionen bekannt machen.

Ein Blick hinter die Kulissen: Strings und String-Literale

Die Klasse `String` ist eine gewöhnliche Klasse, nimmt aber dennoch eine Sonderstellung ein. Da sie sich im Package `java.lang` befindet, gehört sie mit zur Sprache, ist jedoch mit der Klasse `Object` die einzige, der in der Sprachbeschreibung *[1]* ein eigener Abschnitt gewidmet wurde. Im Abschnitt 4.3 werden Referenztypen beschrieben. Die Abschnitte 4.3.2 *The Class Object* und 4.3.3 *The Class String* sind die einzigen Abschnitte, die sich einzelnen Klassen widmen. Dies unterstreicht die zentrale Stellung von Strings.

Warum sind Strings etwas Besonderes? Scott Oaks, anerkannter Java-Fachmann und Autor mehrerer Bücher mit Java-Bezug, schreibt in seinem sehr empfehlenswerten Buch *Java Performance [2]*: „Strings are, far and away, the most common Java object; your application's heap is almost certainly filled with them." Kein Wunder also, dass Strings auf Bibliotheks-, Compiler-, JIT- und JVM-Ebene eine besondere Aufmerksamkeit genießen. Wenn Strings und String-Operationen effizient implementiert sind, kommt das der Laufzeit von Java-Anwendungen insgesamt zugute.

Beginnen wir mit unseren Betrachtungen ganz am Anfang. In der Version 1.0 von Java wurden Strings intern als Character-Array (`char[]`) implementiert. Um den Speicherbedarf zu minimieren, sind String-Literale in der virtuellen Maschine Unikate. Die Sprachbeschreibung *[1]* sagt in Abschnitt 3.10.5 *String Literals* dazu: *„String literals – or, more generally, strings that are the values of constant expressions – are 'interned' so as to share unique instances, using the method String.intern"*. Dazu gibt es einen String-Pool, der diese Unikate enthält. Wird eine neue Klasse in die VM geladen, so wird für jedes String-Literal im Kompilat dieser Klasse (.class-Datei) geprüft, ob es sich schon im String-Pool befindet. Falls ja, wird die Referenz auf diesen Pool-Eintrag verwendet, falls nein, wird das Literal eingetragen und diese neue Referenz verwendet. Der String-Pool ist in C innerhalb der VM als Hash-Map fester Größe implementiert. Die Größe kann beim VM-Start mit `--XX:StringTableSize=N` festgelegt werden. Die Default-Größe betrug bis Java 7u40 1009, danach 60013 Buckets. Bis Java 6 befand sich der String-Pool im PermGen. Da mit Java 8 der PermGen durch den Metaspace ersetzt werden sollte, wurde bereits mit Java 7 der String-Pool vom PermGen in den Heap verlagert. Der String-Pool wird *garbage-collected*. Da die String-Literale aber Referenzen innerhalb von Methoden darstellen, können nur String-Literale gelöscht werden, deren Methoden respektive Klassen entladen wurden. Pool-Einträge, die durch die noch vorzustellende `intern()`-Methode in den Pool aufgenommen wurden, können ganz normal *garbage-collected* werden.

Das explizite Internen von Strings

String-Literale sind Unikate im String-Pool, wie wir bereits gesehen haben. Andere Strings können programmatisch in den String-Pool aufgenommen und somit zu Unikaten gemacht werden. Die entsprechende Methode der Klasse `String` ist `intern()`. Diese prüft, ob der String bereits im Pool vorhanden ist. Falls ja, wird diese Referenz zurückgegeben, falls nein, wird der String eingetragen und diese neue Referenz zurückgegeben. Um dies zu verdeutlichen, werden in *Listing 1* die möglichen Fälle durchgespielt.

Der String „`Hello, World!`“ kommt insgesamt fünf Mal in *Listing 1* vor. Drei Mal bei der Initialisierung der Variablen, zwei Mal im Test 2. Zur Laufzeit in der VM existiert dieser String im String-Pool genau ein Mal und alle fünf Literalverwendungen verweisen auf diesen einen Pool-Eintrag. Der String existiert noch ein weiteres Mal auf dem Heap, da der String-Konstruktor bei der Initialisierung von `str3` ein Heap-Objekt als Kopie des Konstruktorparameters erzeugt. Die Tests 1 und 2 zeigen, dass alle Literalverwendungen sowie die Variablen `str1` und `str2` auf den Pool-Eintrag verweisen. Die Tests 3 und 4 belegen den oben wiedergegebenen Teil der Definition eines String-Literals mit „values of constant expressions“. Die bereits zur Compile-Zeit verbundenen Literale entsprechen dem diesbezüglichen Literal. Dass dies für andere Ausdrücke, zum Beispiel Ausdrücke mit Variablen, nicht zutrifft, zeigen die Tests 5 und 6. Die aneinandergehängten Variablen sind zwar „Equals“, aber nicht identisch. Dass solche Heap-Strings mit der `intern()`-Methode in den String-Pool überführt werden können, zeigt der Test 7. Die Tests 8 und 9 belegen schließlich die obige Aussage, dass sich `str3` im Heap und nicht im String-Pool befindet.

```
String str1 = "Hello, World!";
String str2 = "Hello, World!";
String str3 = new String("Hello, World!");
/* 1 */ assertSame(str1, str2);
/* 2 */ assertSame("Hello, World!", "Hello, World!");
/* 3 */ assertSame("Hel" + "lo", "Hel" + "lo");
/* 4 */ assertSame("Hel" + "lo", "Hello");
/* 5 */ assertEquals(str1 + str1, str2 + str2);
/* 6 */ assertNotSame(str1 + str1, str2 + str2);
/* 7 */ assertSame((str1 + str1).intern(),
                   (str2 + str2).intern());
/* 8 */ assertNotSame(str1, str3);
/* 9 */ assertEquals(str1, str3);
```

Listing 1

Schnittstelle ohne Implementierungsbezug

Dass das JDK von erfahrenen Entwicklern definiert wurde, weiterentwickelt wird und diese die Grundsätze des Software-Engineerings beherzigen, kann man an Strings recht gut belegen. Es gab zumindest zwei fundamentale Änderungen in der String-Implementierung, die praktisch keine Auswirkungen auf die Schnittstelle der Klasse `String` hatten. In der Version 1.0 sind Strings intern als Character Array (`char[]`) definiert. Ein char ist ein vorzeichenloser, ganzzahliger Datentyp mit Werten zwischen 0 und 65535 (\uffff), die Unicode-Zeichen repräsentieren und einen Speicherbedarf von 16 Bits haben. Mit Java 5 wurde Unicode 4.0 eingeführt, womit Zeichen-Codierungen größer als \uffff zu unterstützen waren. Diese sogenannten „supplementary“ Character wurde ohne eine Änderung des `char`-Datentyps und die damit unweigerlich verbundenen Inkompatibilitäten eingeführt, indem sie als ein Paar von Code-Point-Werten, sogenannte *Surrogate*, repräsentiert werden. Seit Java 5 kann also ein Unicode-Zeichen zwei `char`-Werte zur Darstellung benötigen. An uns als Entwickler mit typischerweise europäischen Zeichensätzen im Tagesgeschäft gingen die zusätzlichen Methoden in der Klasse String unbemerkt vorbei. In der Sprachbeschreibung *[3]* wurden lediglich die letzten Wörter bei der Definition von Strings von „Instances of class String represent sequences of Unicode characters“ in „... Unicode code points“ geändert.

Eine wesentlich tiefgreifende Änderung wurde mit Java 9 realisiert. Die String-Implementierung wurde intern auf ein Byte Array (`byte[]`) umgestellt. Wie wir gerade gesehen haben, wurden Zeichen in 16 Bits codiert. Einige chinesische und japanische Zeichen benötigen sogar 32 Bits. Für sehr viele Anwendungen würden jedoch auch 8 Bits ausreichen. Für ein einzelnes Zeichen ist das unerheblich, für Strings nicht. Der JEP 254: *Compact Strings [4]* hatte das Ziel, den Speicherbedarf der String-Implementierung zu verringern. Seit Java 9 werden Strings intern in einem Byte Array statt in einem Character Array (ein Field mit Name `value`) gehalten und das Flag `coder` identifiziert die Codierung des Byte Array. Das Flag kann die beiden Werte LATIN1 und UTF16 annehmen. Besteht ein String (Literal oder eingelesen) aus Zeichen, die eine 16- oder gar 32-Bit-Codierung benötigen, so wird UTF16 verwendet, falls nicht LATIN1.

String-Deduplication

Das Ziel des JEP 192: *String Deduplication [5]* ist: „Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of String are automatically and continuously deduplicated.“ Unter einem Duplikat sind zwei gleiche Strings (Methode `equals()`) zu verstehen. Die Deduplizierung findet jedoch nicht auf den String-Objekten, sondern auf den dahinter existierenden Arrays statt. Da JEP-Nummern in der Regel einfach hochgezählt werden, begannen die Arbeiten zum JEP 192 offensichtlich zu einem Zeitpunkt, als Stings noch als Character-Arrays implementiert waren; dies ist so auch im JEP nachlesbar. Mittlerweile, nach Umsetzung des JEP 254, werden Byte Arrays dedupliziert. Konzeptionell wird einfach eine der beiden Array-Referenzen auf das andere Array umgesetzt und das nicht mehr referenzierte Array kann speicherbereinigt werden. Auch dieses Verhalten kann man sehr anschaulich überprüfen, wie der Code in *Listing 2* zeigt

```
int random = new Random().nextInt();
String string1 = new String("some string " + random);
String string2 = new String("some string " + random);
Field field = String.class.getDeclaredField("value");
field.setAccessible(true);

Object object1 = field.get(string1);
Object object2 = field.get(string2);
/* 1 */ assertNotSame(object1, object2);

System.gc();
Thread.sleep(1000);

Object object3 = field.get(string1);
Object object4 = field.get(string2);
/* 2 */assertSame(object3, object4);
```

Listing 2

```
// Java 8

public static String join(CharSequence delimiter, CharSequence... elements)
// Returns a new String composed of copies of the CharSequence elements joined together with a copy of the
specified delimiter.

public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)
// Returns a new String composed of copies of the CharSequence elements joined together with a copy of the
specified delimiter.


// Java 9

public IntStream chars()
// Returns a stream of int zero-extending the char values from this sequence. Any char which maps to a surrogate
code point is passed through uninterpreted.

public IntStream codePoints()
// Returns a stream of code point values from this sequence. Any surrogate pairs encountered in the sequence are
combined as if by Character.toCodePoint and the result is passed to the stream.


// Java 11

public String strip()
// Returns a string whose value is this string, with all leading and trailing white space removed.

public String stripLeading()
// Returns a string whose value is this string, with all leading white space removed.

public String stripTrailing()
// Returns a string whose value is this string, with all trailing white space removed.

public boolean isBlank()
// Returns true if the string is empty or contains only white space codepoints, otherwise false.

public Stream<String> lines()
// Returns a stream of lines extracted from this string, separated by line terminators.

public String repeat(int count)
// Returns a string whose value is the concatenation of this string repeated count times.


// Java 12

public String indent(int n)
// Adjusts the indentation of each line of this string based on the value of n, and normalizes line termination
characters.

public <R> R transform(Function<? super String,? extends R> f)
// This method allows the application of a function to this string. The function should expect a single String
argument and produce an R result.

public Optional<String> describeConstable()
// Returns an Optional containing the nominal descriptor for this instance, which is the instance itself.

public String resolveConstantDesc(MethodHandles.Lookup lookup)
// Resolves this instance as a ConstantDesc, the result of which is the instance itself.


// Java 13

@Deprecated(forRemoval=true, since="13")
public String stripIndent()
// Returns a string whose value is this string, with incidental white space removed from the beginning and end of
every line.

@Deprecated(forRemoval=true, since="13")
public String translateEscapes()
// Returns a string whose value is this string, with escape sequences translated as if in a string literal.

@Deprecated(forRemoval=true, since="13")
public String formatted(Object... args)
// Formats using this string as the format string, and the supplied arguments.
```

Listing 3

Zunächst erzeugen wir mithilfe eines Zufallszahlengenerators zwei Strings, deren implementierende Arrays dieselben Inhalte enthalten, die aber verschieden in der Semantik von `assertNotSame()` sind, wie Test 1 zeigt. Danach rufen wir den Garbage Collector auf und müssen ihm, da er in einem anderen Thread läuft, auch genug Zeit geben, sein Werk – das Deduplizieren – zu beenden. Danach existiert nur noch eines der beiden Arrays und beide `value`-Fields verweisen darauf, wie Test 2 beweist. Falls Sie den Code selbst ausführen wollen, achten Sie darauf, dass G1 als Garbage Collector verwendet wird (`-XX:+UseG1GC`, Default seit Java 9) und String-Deduplication aktiviert ist (`-XX:+UseStringDeduplication`).

Die Idee unserer Kolumne ist, Ihnen unbekannte Kostbarkeiten des SDK vorzustellen, um zusätzliche Frameworks vermeiden und Aufwände minimieren zu können. Mit den bisherigen Ausführungen zu Strings verhält es sich anders. Sie müssen nichts tun, trotzdem verringert sich die Laufzeit und der Speicherbedarf Ihrer Anwendung reduziert sich, wenn Sie zu einer neueren Version migrieren, die die vorgestellten Optimierungen eingebaut hat. Es ist, wie wir meinen, eine beruhigende Vorstellung, dass dies von Zeit zu Zeit ganz ohne unser Zutun passiert. An dieser Stelle vielen Dank an alle Beteiligten, die Java voranbringen.

Neue Methoden der Klasse String

Um nicht nur automatisch stattfindende Verbesserungen zu beschreiben, wollen wir unseren Artikel mit API-Änderungen beschließen, die auch für die Klasse `String` regelmäßig stattfinden. In *Listing 3* sind diese beginnend mit Java 8 aufgeführt. Neben den Signaturen zeigt *Listing 3* auch die jeweils erste Zeile des JavaDoc der Methode. Da diese Beschreibungen sehr anschaulich sind, verzichten wir auf Beispiele. Ausnahmen sind lediglich die in Java 12 hinzugefügten Methoden `describeConstable()` und `resolveConstantDesc()`. Sie erlauben die Verwendung des Constant Pool der JVM und sind für die meisten Leser wahrscheinlich nicht interessant, wenn es um die Realisierung fachlicher Anforderungen geht. Wir verzichten auf ein Beispiel. Eine Besonderheit nehmen die mit Java 13 eingeführten Methoden ein, die zugleich als `@Deprecated` annotiert wurden. Dies hängt mit dem Preview Mode für Text-Blocks zusammen. Trotzdem können die Methoden verwendet werden, ohne beim Kompilieren oder Ausführen die Option `--enable-preview` verwenden zu müssen.

Zusammenfassung

Die Klasse `String` ist eine zentrale Klasse des SDK. Durch die häufige Verwendung von Strings in praktisch jeder Anwendung ist eine möglichst optimale Implementierung eine Grundvoraussetzung für hohe Performanz und geringe Speicheranforderungen. Die Ingenieure des SDK haben daher sowohl auf Sprach-, als auch auf Implementierungsebene in den letzten Jahren erhebliche Optimierungen vorgenommen, die wir vorgestellt haben. Sie gehören unzweifelhaft zu den unbekannten Kostbarkeiten des SDK und in das Wissen eines jeden Java-Entwicklers. Abschließend haben wir noch eine Reihe von Methoden der Klasse `String` aufgeführt, die seit Java 8 in die Klasse aufgenommen wurden *(siehe Listing 3)*.

Referenzen

[1] James Gosling, Bill Joy, Guy Steele. The Java Language Specification, Addision-Wesley, 1996.

[2] Scott Oaks. Java Performance – The Definitive Guide, O´Reilly, 2014.

[3] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith. The Java Language Specification, Java SE 13 Edition. 2019.

[4] JEP 254: Compact Strings. *https://openjdk.java.net/jeps/254.*

[5] JEP 192: String Deduplication in G1. *https://openjdk.java.net/jeps/192.*

Bernd Müller

Ostfalia

bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

CyberLand – virtuell, kreativ und informativ

Falk Sippach, JUG Darmstadt

Die Auswirkungen der Corona-Pandemie haben ab März 2020 das öffentliche Leben in Europa und etwas später auch weltweit nahezu lahmgelegt. Schulen und Kindertagesstätten blieben geschlossen und viele von uns mussten von zuhause aus arbeiten. Leider waren auch viele öffentliche Veranstaltungen, unter anderem Konferenzen, betroffen – darunter die JavaLand 2020. Aus den Reihen der Konferenzleitung und des Programmkomitees haben wir uns Gedanken über alternative Veranstaltungen im März gemacht. Aus einer dieser Ideen ist dann die CyberLand entstanden: eine Online-Konferenz mit den Referenten der JavaLand.

Die Teilnahme an der CyberLand war kostenfrei und jeder Interessierte durfte teilnehmen. Innerhalb von nur einer Woche wurde das Event mithilfe der CyberJUG und des iJUG e. V. auf die Beine gestellt. Es gab sogar mehr interessierte Sprecher, als am Ende angenommen werden konnten. Aufgrund der kurzen Vorlaufzeit hatten wir uns auf nur zwei parallele Vorträge beschränkt, um den organisatorischen Aufwand in Grenzen zu halten. Außerdem fehlte schlichtweg die Erfahrung mit einem großen Online-Event. Über die Planungen im Vorfeld hat Hendrik Ebbers einen Bericht verfasst *[1]*; Markus Harrer hat über seine Erfahrungen zu Remote-Events einen Blog-Post geschrieben *[2]*.

Als Veranstaltungstag hatten wir bewusst den ersten Konferenztag der abgesagten JavaLand gewählt, den 17. März. Bei der Plattform konnten wir auf die Erfahrungen der CyberJUG zurückgreifen. Pro Vortrag hätten bis zu 1.000 Teilnehmer live dabei sein können. Deutlich mehr als wir initial erwartet hatten – wir wären auch mit 100 Zuhörern sehr zufrieden gewesen. Das Interesse war jedoch sehr groß. Es gab letztlich mehr als 1.900 Registrierungen für die CyberLand. Bei der Keynote schalteten bereits knapp 800 Zuschauer ein, bei den nachfolgenden parallelen Vorträgen waren zusammen sogar etwa 900 Leute dabei.

Die Infrastruktur war erstaunlich stabil und es gab nahezu keine schwerwiegenden technischen Probleme. Über kleinere menschliche Pannen ließ sich leicht hinwegsehen. Und so wurde es ein spannender und sehr informativer Tag mit insgesamt 17 Vorträgen von hochkarätigen nationalen und internationalen Sprechern. Vielen Dank an alle, die mitgewirkt und teilgenommen haben. Bei den Vorträgen gab es übrigens auch eine ganze Menge Interaktion. Die Sprecher nutzten zum Beispiel Live-Umfragen, um das Publikum einzubeziehen. Außerdem hatten alle Teilnehmer die Möglichkeit, nebenbei zu chatten, und es durften auch explizit Fragen an die Vortragenden gestellt werden. Letztere wurden dann meist am Ende gebündelt von einem Moderator vorgelesen. Die Webinar-Räume waren zudem noch deutlich über das Vortragsende geöffnet. Einige Teilnehmer nutzten diese Gelegenheit, um mit dem Sprecher offene Fragen im Chat zu diskutieren. Leider wartete schon der nächste spannende Vortrag und so musste man sich wie auch bei einer normalen Konferenz entscheiden. Weiterer Diskussionsbedarf konnte dann aber noch im #CyberLand-Channel des deutschsprachigen JVM-German Slack *[3]* gestillt werden.

Die gemachten Erfahrungen waren sehr hilfreich, da an den darauffolgenden Tagen bei virtuellen lokalen Vortragsabenden zum Beispiel der Java User Groups in Darmstadt, Kassel (JUG Hessen) und Bremen/Oldenburg ebenfalls kleinere Remote-Events stattfanden. Alle Vorträge der CyberLand wurden übrigens aufgezeichnet und sind mittlerweile online verfügbar *[4]*. Wer Schwierigkeiten hatte, sich für den „Besuch" eines Vortrags zu entscheiden, hat nun die Möglichkeit, den anderen noch anzuschauen. Übrigens: Ein ganzer Tag vor dem Bildschirm ist ähnlich anstrengend wie bei einer echten Konferenz. Und leider fehlt die direkte, physische Interaktion mit echten Menschen. Aufgrund der problematischen Situation war dies jedoch ein guter Kompromiss und ein tolles Lebenszeichen der deutschsprachigen Java-Community!

Die CyberLand war eine Veranstaltung des iJUG e. V., organisiert wurde sie von Hendrik Ebbers (JUG Dortmund), Tobias Frech (JUG Stuttgart und Mitglied im Vorstand des iJUG), Markus Harrer (CyberJUG), Gerrit Meier (JUG Ostfalen) und Falk Sippach (JUG Darmstadt). Bei der Moderation ist zudem Gerd Aschemann kurzfristig eingesprungen und beim Schneiden der Videos hat Ralf D. Müller tatkräftig unterstützt. Weitere Infos und auch die Videos zu den Vorträgen gibt es auf der Konferenzseite *[4]*. Interessierte können der CyberLand auch bei Twitter *[5]* folgen, denn weitere virtuelle Events könnten folgen.

Referenzen:

[1] *https://guigarage.com/2020/03/12/cyberland.html*
[2] *https://www.feststelltaste.de/erfahrungen-remote-events/*
[3] *https://slackin-jvm-german.herokuapp.com/*
[4] *http://cyberland.ijug.eu/2020/*
[5] *https://twitter.com/CyberLandConf*

Falk Sippach
JUG Darmstadt
falk@jug-da.de

Falk Sippach hat 20 Jahre Erfahrung mit Java und ist bei der OIO - den Java-Experten der Trivadis - als Trainer, Software-Entwickler und -Architekt tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit Anderen die örtliche Java User Group. Falk twittert unter @sippsack.

Progressive Web Apps in der Praxis

Yves Schubert, iteratec GmbH

Oft macht das Aufrufen von Websites und die Benutzung von komplexeren Webanwendungen unterwegs oder auf mobilen Geräten wenig Spaß. Native Anwendungen sind hier klar im Vorteil, denn diese sind offline und vor allem bequemer aufzurufen als Websites. Der Begriff „Progressive Web App" steht für eine Sammlung von Konzepten und Techniken, die Websites und -anwendungen wieder attraktiv werden lassen. Sie helfen dabei, diese schneller und bei fehlender Internetverbindung auch offline laden zu lassen. Außerdem sollen sie imstande sein, auf unterschiedliche Endgeräte flexibel zu reagieren. In diesem Artikel wird die Philosophie der Progressive Web Apps erklärt und gezeigt, wie das Ganze technisch funktioniert.

Als Apple im Jahr 2007 das erste iPhone der Weltöffentlichkeit vorstellte [1], war es eigentlich angedacht, Anwendungen auf dem Gerät standardmäßig mit HTML, CSS, JavaScript und Ajax, damals auch unter Web 2.0 bekannt, zu erstellen [2]. Der Safari-Browser sollte hierfür Schnittstellen zu tieferen Schichten des Telefons anbieten. Diese Idee war für damalige Verhältnisse extrem innovativ und vorausschauend, da es die Hürde der Anwendungserstellung im Vergleich zu anderen Herstellern bedeutend niedriger machte. Apple erkannte schon sehr früh, dass ein Browser nicht nur für das Rendern von Webseiten geeignet ist, sondern vielmehr als Plattform für komplexe Anwendungen dienen kann. Leider waren die Möglichkeiten von HTML zu der Zeit begrenzt und HTML5 ließ noch eine Weile auf sich warten. Daher und auch aus marketingtechnischen Gründen änderte Apple diese Strategie zugunsten von nativ entwickelten Anwendungen, die über einen Extra-Marktplatz, den App Store, vertrieben wurden [3].

Die Jahre vergingen und die verschiedenen Browser wurden nicht zuletzt auch durch die mächtigen HTML5-Schnittstellen, die es erlauben, auf Hardware und Dateien zuzugreifen, tatsächlich immer mehr zu „App Engines". Diese sind in der Lage, komplexe Anwendungen unabhängig vom Betriebssystem oder Gerät auszuführen. CSS3 tat sein Übriges, denn aufwendige Animationen und grafische Spielereien sind damit relativ einfach umsetzbar. Responsive Layouts („Mobile First") werden zum Standard. Immer mehr sogenannte Cross-Plattform Frameworks erobern den Markt (zum Beispiel Xamarin, Electron.io, React Native und so weiter). Die Grenzen zwischen Web und Desktop, Geräten und Betriebssystemen verschwimmen zusehends.

Mit der Einführung der Web Worker und Service Worker war auch der Grundstein für Progressive Web Apps (PWAs) gelegt. Als treibende Kraft hinter den PWAs hat Google hier viel Pionierarbeit geleistet und gewisse Standards definiert. Hinter diesem Begriff steckt letztendlich eine Vielzahl von Ideen und Konzepten, die alle darauf abzielen, eine Webanwendung mehr wie eine native Anwendung erscheinen zu lassen. Eine der wichtigsten Überlegungen ist: Wie kann der Zugang zu einer Anwendung für möglichst viele Benutzer und damit also für unterschiedlichste Umgebungen (Browser, Betriebssystem, Gerät) so einfach wie möglich gestaltet werden? Die Anwendung selbst soll

- so schnell wie möglich erste sinnvolle Ergebnisse liefern
- auf Umgebungen, die wenig Features anbieten, sinnvolle Ergebnisse liefern
- auf Umgebungen, die vollen Funktionsumfang liefern, diesen auch nutzen
- offline sinnvolle Ergebnisse liefern
- so einfach wie möglich installierbar sein
- sicher sein
- aus Marketinggründen dafür sorgen, dass Benutzer nicht gleich wieder abspringen

```
<!DOCTYPE html>
<html>
<head>
  <title>My super PWA</title>
  <link rel="stylesheet" href="./style.css">
</head>
<body>
  <div>Hier kommt später der eigentliche Inhalt</div>
  <script src="./app.js"></script>
</body>
</html>
```

Listing 1: index.html – Grundgerüst der Webanwendung

Es gibt einige Beispiele von Websites, die nach der Einführung von PWAs signifikant positive Änderungen von KPIs (wie Umsatz oder Benutzerzahlen) vorweisen können. Es gibt sogar eine Website [4], die solche Erfolgsgeschichten sammelt – und dabei selbst eine PWA ist.

Die Bausteine für eine PWA

Die Bausteine, die eine gewöhnliche Website zu einer Progressive Web App machen, sind zunächst übersichtlich. Es steckt aber allerhand Überlegungsarbeit in der konzeptionellen Implementierung. Wir benötigen:

- HTML, CSS, etwas JavaScript
- eine Service-Worker-Datei
- eine Web-App-Manifest-Datei
- etwas Markup und JavaScript zum Zusammenstecken
- HTTPS: PWAs dürfen nur über HTTPS aufgerufen werden

Das in *Listing 1* dargestellte HTML Markup ist das zunächst notwendige Gerüst. Hier ist vorerst noch nichts Spannendes zu sehen, wir werden dieses Gerüst aber Schritt für Schritt erweitern.

Der eigentliche Trick im Hintergrund

Der Service Worker [5] ist ein kleiner Helfer im Hintergrund. Er ermöglicht es, gewisse Dinge unabhängig von der Seite selbst zu erledigen (beispielsweise eine PWA offline zu laden). Es handelt sich um eine JavaScript-Datei, die Zugriff auf Cache und Netzwerkfunktionen des Browsers hat. Sie wird nicht im Kontext der Website, sondern in einem separaten Thread ausgeführt. Für vom Browser oder programmatisch ausgelöste Web-Anfragen kann der Entwickler, anders als in dem veralteten AppCache-Mechanismus, feingranular entscheiden, bei welcher Anfrage wie verfahren wird. So können zum Beispiel bestimmte HTML-, JavaScript-Dateien oder REST-Anfragen in den Browser Cache geladen werden. Diese Daten liegen beim nächsten Aufruf der Anwendung sofort vor und es muss nicht erst auf eine Antwort des Servers gewartet werden. Dennoch kann aber, falls das Gerät online ist, die Anfrage zusätzlich an den Server gesendet werden. Wenn die Antwort angekommen ist, wird der Status der Anwendung entsprechend aktualisiert. Der Benutzer bekommt die Anwendung präsentiert und kann damit arbeiten, während im Hintergrund noch Inhalte nachgeladen werden. Der Service Worker kann auch Daten synchronisieren, ohne die Anwendung selbst zu blockieren und ohne dass die Anwendung überhaupt in einem Tab geöffnet ist.

In *Listing 2* ist der Inhalt der Service-Worker-Datei dargestellt, über die eine Webanwendung prinzipiell offlinefähig wird. Hier wird auch das Prinzip deutlich, wie sich ein Service Worker in die Anwendung einklinkt. So kann auf die verschiedenen Events des Lebenszyklus („install" und „activate") und auf Netzwerkanfragen („fetch") gehorcht werden. In unserem Beispiel wird beim „install"-Ereignis, das auftritt, wenn der Service Worker installiert wird, gesagt, dass die Antworten auf die Anfragen „/", „/index.html",

und „/images/logo.png" im Cache mit dem von uns ausgedachten Namen „v1" gespeichert werden sollen. Beim nächsten Start der Seite greift das „fetch"-Ereignis. Dieses ist so implementiert, dass bei einer Anfrage mit passendem Match im Cache dieser Match zurückgeliefert wird. Andernfalls wird die Anfrage einfach „durchgeschleift".

Um einen Service Worker zu verwenden, genügen wenige Zeilen Code in unserer Webanwendung, siehe hierzu *Listing 3*. Gemäß dem Motto, Funktionalität nur dann anzubieten, wenn der Browser das auch unterstützt (also Progressive Enhancement), wird der Service Worker nur bei Vorhandensein der entsprechenden Schnittstellen geladen.

Was sich zunächst als offensichtlich darstellt, ist hier ein sehr wichtiger Punkt: Während der Entwicklung der Anwendung muss stets an den „Progressive"-Ansatz gedacht werden. Die Webanwendung soll auf möglichst vielen Geräten zumindest die Grundfunktionalität bieten. Also sollte bei Verwendung jeder moderneren Schnittstelle geprüft werden, ob diese überhaupt vorhanden ist. Sinnvolle Alternativen sollten gut überlegt werden. Meistens ist es auch zielführend, mit sogenannten Polyfills zu arbeiten, denn diese können bestimmte Funktionalitäten nachbilden. Ziel ist es jedoch, möglichst viele Browser und Geräte zumindest für die Grundfunktionalitäten zu unterstützen.

Web App Manifest

Die nächste Zutat für eine PWA ist ein Web App Manifest *[6]*. Dies ist eine Datei im JSON-Format, die deklarativ einige Metainformationen der Anwendung beschreibt. In *Listing 4* ist ein Ausschnitt der Manifest-Datei für eine PWA dargestellt.

In dieser Datei wird ein Name für die Anwendung vergeben („name" und „short_name"), der dann später auch als Text unter dem App-Icon im Homescreen angezeigt wird. Es besteht die Möglichkeit, Farben und Icons anzugeben („theme_color", „background_color", „icons"), die unter anderem beim Laden der Webanwendung angezeigt werden. Und es kann angegeben werden, wie die App gestartet werden soll („display"), also zum Beispiel im Vollbildmodus oder mit einem Browserrahmen. Mit „scope" wird angegeben, für welchen Pfad der Webseite dieses Manifest gelten soll. Dies ist hilfreich, wenn sich die Inhalte der PWA nicht auf „/", sondern in einem Unterordner befinden. Eingebunden wird die Manifest-Datei über die index.html im Header-Bereich *(siehe Listing 5)*.

Wenn diese Datei vorhanden und eingebunden ist, Icons bereitgestellt werden, ein Service Worker registriert ist und die Seite mit HTTPS geladen wird, bietet der Browser (manchmal erst bei mehrfacher Verwendung der Anwendung) an, die Webseite zum „Homescreen" hinzuzufügen. Die Webseite erscheint dann als Link neben den anderen installierten Anwendungen und kann von dort direkt aufgerufen werden. Momentan wird das allerdings nur von Chrome auf mobilen Geräten und Desktop angeboten.

Der Schritt, dass die Anwendung „installiert" und direkt über das gewohnte App-Menü aufgerufen werden kann, ist ein kleiner Schritt mit großer Wirkung. Es erleichtert den Zugang zur Anwendung sowohl beim Herunterladen (Link genügt) als auch beim Aufrufen (One Click). Allein diese Tatsache beschert den Betreibern

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('v1').then((cache) => {
      return cache.addAll([
        '/',
        '/index.html',
        '/images/logo.png'
      ]);
    })
  );
});
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

Listing 2: service-worker.js – Service-Worker-Datei, die es ermöglicht, dass die index.html auch offline lädt

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/service-work-
er.js');
  });
}
```

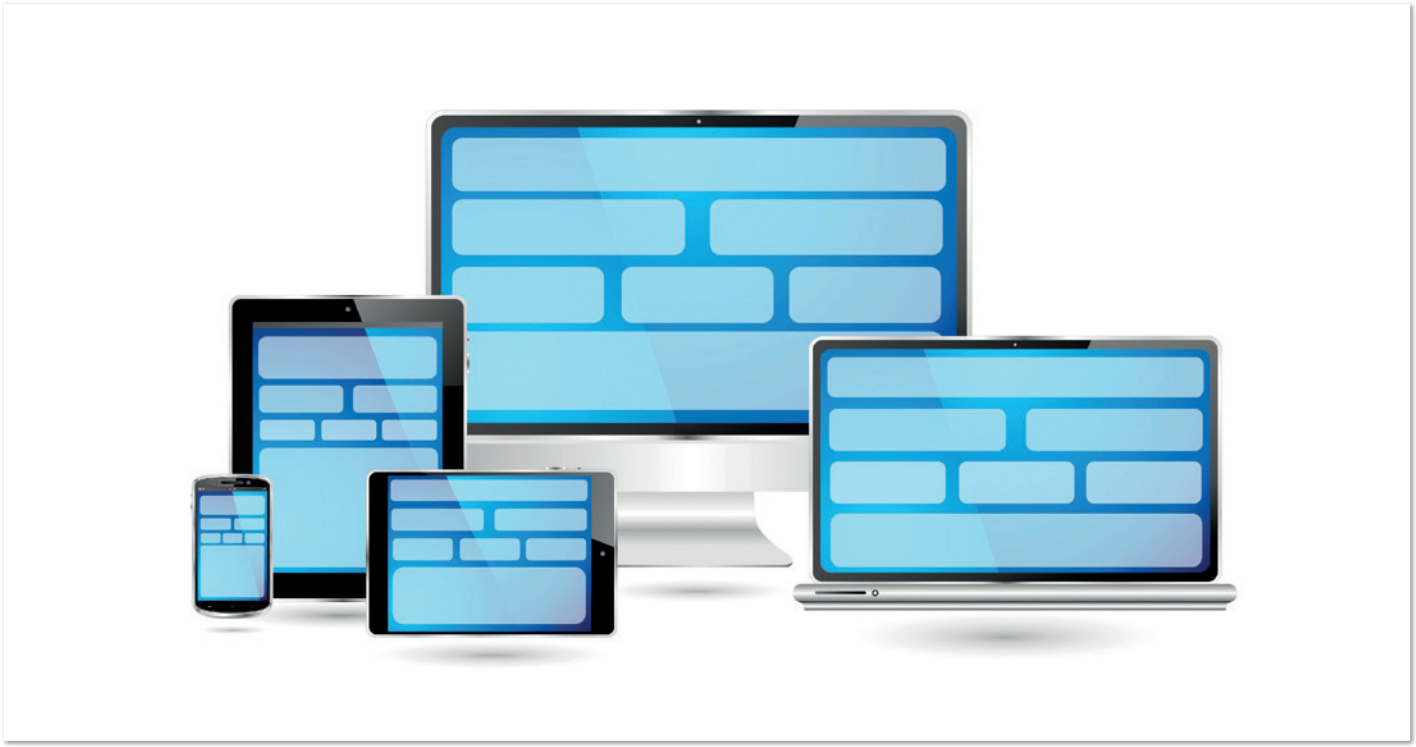
Listing 3: app.js – Einbinden des Service Worker in unsere Anwendung

```
{
  "name": "PWA Notes Example",
  "short_name": "PWA-Notes",
  "theme_color": "#2196f3",
  "background_color": "#2196f3",
  "display": "standalone",
  "scope": "/",
  "icons": [
    {
      "src": "/android-chrome-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/android-chrome-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Listing 4: manifest.json – Die Web-App-Manifest-Datei beschreibt Metainformationen der Webanwendung

```
<!DOCTYPE html>
<html>
<head>
  <title>PWA Notes Example</title>
  <link rel="stylesheet" href="./stylesheets/style.css">
  <link rel="manifest" href="/manifest.json">
</head>
<body>
  <div>Hier kommt der Inhalt</div>
  <script src="./javascripts/app.js"></script>
</body>
</html>
```

Listing 5: index.html – Grundgerüst der Webanwendung, um die Einbindung der Web-App-Manifest-Datei erweitert.



einer solchen Anwendung ungleich bessere Statistiken bei Zugriff und Verweildauer der Anwender *[4]*. Das spiegelt sich nicht zuletzt auch in besseren Umsatzzahlen wider.

Die richtige Balance finden

Mit den erwähnten Maßnahmen ist die Webanwendung schon ein gutes Stück näher an die PWA-Philosophie herangekommen. Sie sind auch die einfachsten, um eine bestehende Webanwendung in eine PWA umzuwandeln. Es fehlt aber ein wichtiger Kern: PWAs sollen schnell sein. Und zwar nicht nur schnell während der Benutzung, sondern auch schnell beim Laden. Nachgewiesenermaßen *[7]* verlieren Benutzer schon nach kurzer Zeit die Lust, wenn sie auf eine Anwendung einige Sekunden warten müssen, bevor sie mit ihr arbeiten können. Genau das soll bei echten PWAs nicht passieren. Abhilfe schafft hier ein Architektur-Muster, das sich „Application Shell" nennt *[8]*.

Der äußere Rahmen der Anwendung und, falls erforderlich, einige wenige Grundfunktionen werden unter Verwendung des Service Worker gecacht. Dies führt dazu, dass sich die Anwendung quasi instant anzeigt, wenn sie beim nächsten Mal geladen wird, selbst wenn das Gerät eine schlechte Verbindung hat oder gar offline ist. Wichtig ist, dass diese „Hülle" fast ausschließlich statisch ausgeliefert und so wenig wie möglich dynamisch per JavaScript aufgebaut wird. Denn das dynamische Aufbauen des Seiteninhalts kostet Zeit. Dynamischer Inhalt wird dann erst sukzessive nachgeladen.

```
// Import der Workbox Bibliothek
importScripts('https://storage.googleapis.com/workbox-cdn/releases/4.3.1/workbox-sw.js');

// Route registrieren, die js und css Dateien cached
workbox.routing.registerRoute(
  /\.(?:js|css)$/,
  new workbox.strategies.StaleWhileRevalidate({
    cacheName: 'static-resources',
  })
);

// Route registrieren, die Bilder cached (Mit Verfallsdatum)
workbox.routing.registerRoute(
  /\.(?:png|gif|jpg|jpeg|webp|svg)$/,
  new workbox.strategies.CacheFirst({
    cacheName: 'images',
    plugins: [
      new workbox.expiration.Plugin({
        maxEntries: 60, // Maximal 60 Bilder
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 Tage
      }),
    ],
  })
);
```

Listing 6: service-worker.js – Service Worker mit Workbox-Beispielen.

Zu entscheiden, was Teil der „Application Shell" ist und was wann in welcher Reihenfolge nachgeladen wird, muss gut überlegt sein und ist bei genauerem Hinsehen nicht trivial. Entscheidungen, die hier getroffen werden, haben enormen Einfluss auf die weitere Entwicklung. Was hier mit einfließt, ist die Frage, welcher Teil der Anwendung serverseitig („Server Side Rendering", SSR) und welcher Teil dynamisch im Browser erstellt wird („Client Side Rendering", CSR).

Im Gegensatz zu konventionellen Webanwendungen, bei denen eine Art zu rendern vorherrscht, findet man in PWAs meist eine Mischform. Die hohe Kunst ist es, die perfekte Balance zwischen SSR und CSR zu erhalten. Das gelingt realistisch eigentlich nur schrittweise mit Messen und Verbessern. Die Entwickler von Twitter haben ihre Bemühungen, die Balance zu finden, in einem Blog-Artikel festgehalten *[9]*.

Das Rad nicht neu erfinden

Genauer betrachtet sind die Aufgaben, die ein Service Worker zu erledigen hat, oft sehr ähnlich. Daher gibt es sowohl die Möglichkeit, über verschiedene Frameworks einen Service Worker generieren zu lassen, als auch diejenige, bestehende Bibliotheken für die Implementierung zu verwenden. Eine nützliche Bibliothek für das Cachen von Ressourcen ist Workbox *[10]*, siehe *Listing 6*.

Mit Workbox können verschiedene sogenannte Routen (das sind letztendlich reguläre Ausdrücke, die auf aufgerufene URLs matchen) registriert und einem Handler übergeben werden. Workbox bringt dabei schon einige eigene Handler mit, wie in unserem Beispiel StaleWhileRevalidate und CacheFirst.

StaleWhileRevalidate fragt zuerst beim Cache, ob eine passende Ressource enthalten ist, und liefert diese zurück. Gleichzeitig wird aber die Ressource im Hintergrund beim Server angefragt und der Cache aktualisiert. Beim nächsten Aufruf wird dann die aktualisierte Ressource verwendet.

CacheFirst fragt ebenfalls zunächst beim Cache an. Falls die Ressource nicht gefunden wird, wird sie über das Netzwerk geladen. Es findet hier keine Aktualisierung der Caches im Hintergrund statt. Diese Strategie bietet sich also nur an, wenn sich eine Ressource nicht mehr verändert oder, wie hier im Beispiel, ein Verfallsdatum angegeben wird.

Dieses Beispiel soll hier nur als Einstieg dienen, denn Workbox bietet noch einige Funktionen mehr an, die in der Dokumentation nachgelesen werden können.

Von Checklisten und Leuchttürmen

Weitere Hilfen für die Entwicklung von PWAs sind Checklisten *[11]* und Lighthouse *[12]*. Die Checklisten enthalten einige Punkte, die während der Entwicklung und Bereitstellung einer PWA zu beachten sind. Lighthouse ist eine in Google Chrome integrierte Anwendung, die eine Webanwendung auf unterschiedliche Kategorien wie Performance, Accessibility, Best Practices SEO und eben auch PWA testet. Lighthouse kann auch per Website oder Kommandozeile (zum Beispiel auch in einer Continuous Integration Pipeline) aufgerufen werden. Diese Listen sollten nicht dogmatisch sein, aber wo sinnvoll durchaus in die Entwicklung einfließen.

Fazit

Die Konzepte der Progressive Web Apps sind sicherlich äußerst sinnvolle Erweiterungen, die das Verwenden und Erleben von Web-Anwendungen für den Benutzer deutlich verbessern. Insbesondere unter Verwendung moderner Web-APIs, Push Notifications und der „Add-To-Homescreen"-Funktionalität verschmelzen die Grenzen zu einer nativen Anwendung immer mehr. Aber wie immer gilt hier, dass gut überlegt sein sollte, wann welches Werkzeug am sinnvollsten einzusetzen ist. So sind PWAs nicht als Ersatz für native Anwendungen gedacht, die nach wie vor ihre Daseinsberechtigung haben, sondern eben als nächster Evolutionsschritt von Webanwendungen. Die Basisschritte zur PWA sind relativ einfach und die Browserunterstützung wird immer breiter. Daher lohnt es sich, auf diesen Zug aufzuspringen, denn allein schon die Überlegungen, die für eine PWA getätigt werden müssen, helfen, jede Webanwendung zu verbessern.

Quellen

[1] https://de.wikipedia.org/wiki/IPhone_(erste_Generation)
[2] https://web.archive.org/web/20081215230338/http://www.apple.com/pr/library/2007/06/11iphone.html
[3] https://web.archive.org/web/20071018221832/http://www.apple.com:80/hotnews/
[4] https://www.pwastats.com/
[5] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
[6] https://developer.mozilla.org/en-US/docs/Web/Manifest
[7] https://ymedialabs.com/progressive-web-apps
[8] https://developers.google.com/web/fundamentals/architecture/app-shell
[9] https://blog.twitter.com/engineering/en_us/topics/open-source/2017/how-we-built-twitter-lite.html
[10] https://developers.google.com/web/tools/workbox/
[11] https://developers.google.com/web/progressive-web-apps/checklist
[12] https://developers.google.com/web/tools/lighthouse/

Yves Schubert
iteratec GmbH
Yves.schubert@iteratec.com

Schon über 15 Jahre ist Yves Schubert beruflich und auch privat in vielen Softwareprojekten unterschiedlichster Technologiebereiche maßgeblich beteiligt. Als sogenannter Full-Stack-Entwickler verfolgt er dabei immer das Ziel, möglichst viel von dieser schnelllebigen Softwarewelt mitzunehmen, aber auch die neuen Trends auf ihre Nachhaltigkeit hin abzuklopfen. Den Einsatz neuer Technologien wägt er dabei immer kritisch ab. In der Praxis wendet Yves die Prinzipien des Software Craftsmanship an.

Web Apps in Vaadin 14

Tobse Fritz, ITscope GmbH

Moderne Webanwendungen sollen sicher, schnell und responsive sein. Auch bei schlechter Internetverbindung, um dem Nutzer möglichst das Erlebnis einer lokal installierten Anwendung zu bieten – eine sogenannte Progressive Web App. Das stellt den Entwickler vor neue Herausforderungen. So reicht serverseitiges Rendern von HTML-Templates dazu nicht mehr aus. Webtechnologien wie WebSockets und Web Components bieten auf der Clientseite neue Möglichkeiten. Doch wie lassen sich diese in der Java-Welt nutzen? Das Vaadin-Framework stellt in der neusten Version eine komfortable und vielseitige Schnittstelle für diese neuen Webstandards zur Verfügung. Anhand einer Beispiel-Webanwendung wird gezeigt, wie sich Vaadin für die Client- und Backend-übergreifende Entwicklung einsetzen lässt und wo mögliche Fallstricke liegen.

Die Ursprünge des Vaadin-Frameworks gehen bis auf das Jahr 2002 zurück, als Erweiterung eines Web-Frameworks, das Java-Entwicklern mit Swing-Know-how die Web-Entwicklung erleichterte. Demzufolge erschien 2009 das Vaadin Framework unter neuem Namen mit Versionsnummer 6 – als Open-Source-Lösung mit kommerziellen Add-ons. Vaadin ist dabei seinem Ziel treu geblieben: die browserübergreifende Webentwicklung für Java-Entwickler möglichst einfach zu gestalten. Technisch setzte Vaadin hierzu auf Googles Web Toolkit (GWT), das clientseitige Entwicklung in Java erlaubte und die damals noch aufwendige Unterstützung unterschiedlicher Browser erleichterte. Die Entwicklung von Webanwendungen in Front- und Backend in einer Sprache ohne Kenntnisse von HTML und CSS war ein überzeugendes Feature. Seitdem hat sich Vaadin nicht einfach nur weiterentwickelt, sondern ab 2018 mit der Version 10 neu ausgerichtet: Mit der Abkehr von GWT hin zu „modern mobile-first web apps" *[1]* durch die Unterstützung von Webstandards und die Integration von Web Components. Diesmal nicht mit dem Ziel, Webtechnologien zu verstecken, sondern sie dem Java-Entwickler möglichst einfach zugänglich zu machen. Das aktuelle Vaadin-Framework bietet damit eine große Flexibilität und lässt den Entwicklern die Freiheit, sich auf reine Java-Entwicklung zu beschränken oder in die Feinheiten der clientseitigen Entwicklung mit JavaScript oder TypeScript einzusteigen.

Die Weboberfläche

Angenommen, für das nächste Webprojekt steht folgende Idee im Raum: Es soll eine Plattform entworfen werden, um die internen Entwickler zu guter Softwareentwicklung zu motivieren. Diese soll die Regeln für Clean Code *[2]* vermitteln, also als Nachschlagewerk

dienen. Freiwillig soll zudem jeder seine Fortschritte loggen können und durch Achievements für die Beachtung der Regeln belohnt werden. Einen UI-Entwurf gibt es auch schon *(siehe Abbildung 1)*. Mit Vaadin gibt es verschiedene Herangehensweisen, wie sich dieser Entwurf realisieren lässt:

1. **WYSIWYG („What You See Is What You Get") Editor**
   Der UI-Designer nutzt Standardkomponenten aus der *Flow*-UI-Bibliothek und entwirft die Oberfläche in Vaadins WYSIWYG Editor. Der Editor generiert eine XML-Repräsentation und passende Java-Adapter, um die Oberfläche einzubinden. Der Editor läuft als Plug-in in *Intellij* oder *Eclipse* und ist Teil der Vaadin Pro Subscription.
2. **Flow-Komponenten**
   Der Entwickler setzt die Oberfläche in Java-Code aus *Flow*- Komponenten zusammen. Flow ist seit Vaadin 10 die Standard UI Library und enthält 42 Standardkomponenten wie beispielsweise Textfeld, Button oder Link.
3. **Element-API**
   Standardkomponenten reichen nicht aus, um den Entwurf umzusetzen. Darum nutzt der Entwickler das Element-API im Java-Code, um objektorientiert HTML-Elemente anzulegen und zusammenzusetzen.
4. **Custom HTML**
   Das HTML kommt beispielsweise aus einer externen Quelle und soll direkt ausgegeben werden. Dazu nutzt der Entwickler die *Flow*-Komponente „HTML", die einen String oder Stream entgegennimmt. Somit lässt sich jedes valide HTML direkt ausgeben.
5. **Polymer Templates**
   Die benötigte Komponente ist aufwendiger und sie enthält auch clientseitige Logik – deren Entwicklung durch Java-Code stößt an seine Grenzen. Es wird ein Webentwickler hinzugezogen, der die Komponente in HTML, JavaScript und CSS umsetzt. Er schreibt dazu ein Polymer Template, das Slots enthält. Der Java-Entwickler kann dann einen Adapter anlegen, um das Template genau wie andere *Flow*-Komponenten nutzen zu können.
6. **Web Components**
   Die benötigte Komponente wurde bereits als Web Component von einem Web-Frontend-Entwickler veröffentlicht – diese selbst zu entwickeln, ist also nicht nötig. Der Java-Entwickler legt auf dieselbe Weise wie das Polymer Template einen Adapter an, um sie im Java-Code nutzen zu können.

Die Punkte 1 bis 6 verdeutlichen gut, dass es nicht nur einen richtigen Weg gibt, in Vaadin eine Weboberfläche zu entwickeln. Die Rollen „Web-Entwickler" und „Java-Entwickler" sind bewusst mit aufgeführt. Nicht, um klare Aufgabentrennungen vorzugeben, sondern – im Gegenteil – um aufzuzeigen, wie vielfältig eine Zusammenarbeit im Team aussehen kann. Wer bei Webentwicklung zwangsläufig an Trennung von Front- und Backend denkt, soll eines Besseren belehrt werden. Gerade in agilen, crossfunktionalen Teams ist es ein hohes Ziel, den Technologie-Stack kleinzuhalten, um ein Feature intern entwickeln zu können. Dies kann bedeuten, dass Java-Entwickler mit einem teaminternen Frontend-Webentwickler zusammenarbeiten, der wie in Punkt 5 Komponenten entwirft. Oder aber das Team entscheidet sich dazu, Entwicklungssprachbarrieren zu vermeiden und möglichst auch den gesamten Frontend-Code in Java zu schreiben. Oder aber diese Trennung ist gewünscht und ein UI-Designer entwirft die Oberfläche im Vaadin-Editor. Wo andere Frameworks feste Aufteilungen bedingen, ist Vaadin flexibel und lässt sich so in vielen Szenarien einsetzen.

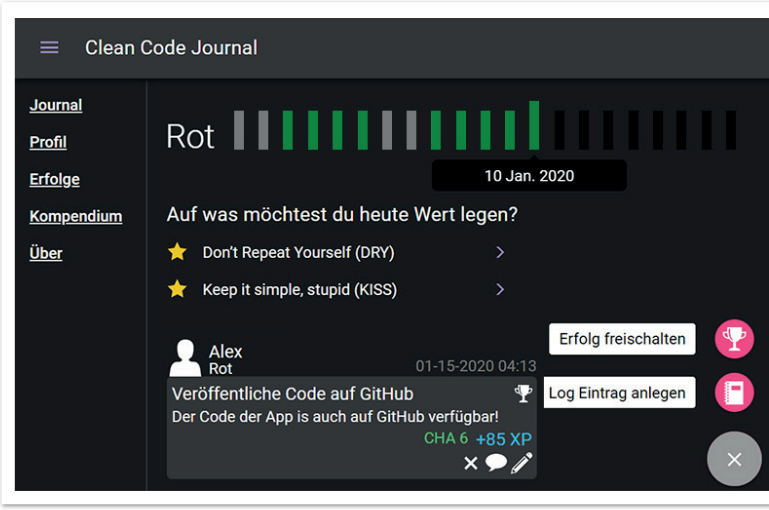


Abbildung 1: Beispielaufbau einer responsiven Vaadin 14 App: Ein Clean Code Logbuch. (© Tobse Fritz)

Eine Clean Code App

Das oben genannte Web-Projekt zum Clean Code soll als Beispiel für eine Vaadin-App herhalten. Die beiden Hauptfunktionen sind „Logeintrag erstellen" und „Achievement freischalten" und stehen in jeder Ansicht über ein globales Schnellmenü „+" zur Verfügung. Beides öffnet einen Log-Dialog, der nach Abschluss einen Logeintrag im Journal erzeugt. Im Kompendium können alle Clean-Code-Regeln tabellarisch aufgelistet und durchsucht werden. Jede Regel ist einem Grad zugeordnet, den ein Entwickler absolvieren kann und der, ähnlich wie bei Judo-Gürteln, einer Farbe zugeordnet ist. Die Demo dazu kann im Internet *[3]* ausprobiert werden.

Sie soll das Grundgerüst einer Vaadin-Applikation zeigen und dazu einladen, sich Details dazu im Quellcode nochmal genauer anzusehen, denn auf alle Klassen kann in diesem Artikel leider nicht eingegangen werden. Sie bietet aber den Vorteil, dass sich die vorgestellten Funktionen zu einem Ganzen fügen und auch etwaige Fallstricke aufdecken, die bei Einzelbeispielen nicht auffallen würden. Vor allem war es aber auch eine gute Übung aufs Exempel, eine Web-App in Vaadin von Grund auf neu zu entwickeln. Die Basis ist das Spring-Boot-Starter-Projekt *[4]*. Es enthält ein Projekt mit vorkonfigurierten Vaadin- und Spring-Bibliotheken, einer Startseite und dem Test-Setup. Das Setup mit Spring Boot wurde gewählt, um zu zeigen, wie gut Vaadin mit bekannten Java Frameworks zusammenspielt.

Einrichtung der Entwicklungsumgebung

Um die App zu entwickeln, muss ein Java Development Kit (JDK) 12 oder neuer sowie Node JS in der Version 10 oder neuer installiert sein. Wichtig dabei ist, dass der Paketmanager *npm* auf dem Systempfad verfügbar ist. *Maven* Goals lassen sich direkt über die IDE starten. Beim Import in die IDE ist darauf zu achten, beim Dependency Management „*Maven*" zu wählen. Nach dem Import lässt sich die Applikation über ein Maven Goal starten: `mvn spring-boot:run`. Oder über die IDE, indem die `main`-Methode der Klasse `org.cleancode.journal.Application` ausgeführt wird. Danach startet die App und ist unter *localhost:8080* aufrufbar. Um Änderungen schnell sehen zu können, ist die App mit den *Spring Boot DevTools* so konfiguriert, dass bei Codeänderungen der Server automatisch neu startet.

```
@Route(layout = MainView.class)
@PageTitle("Clean Code - About")
public class AboutView extends
Composite<VerticalLayout> {
[...]
```

Listing 1

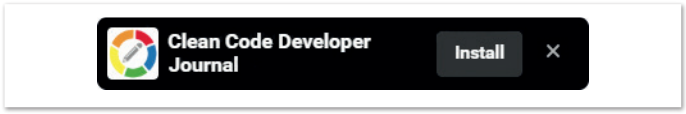


Abbildung 2: PWA-Popup, das dem Nutzer anbietet, die App zu installieren. (© Tobse Fritz)

Ein responsives App Layout

In Vaadin setzen sich Oberflächen aus Komponenten (`Component`) zusammen, so erben beispielsweise Button, Überschrift, Layout oder DIV von `Component`. Auch eine Seite ist in Vaadin eine Component, der mittels Annotation eine `@Route` zugeordnet wird. Die Route definiert den URL-Pfad, über den die Seite erreichbar ist. Entweder über die Angabe eines Anchor als String, oder durch den Verweis auf ein Parent-Layout. *Listing 1* zeigt die `AboutView`. Sie verweist auf die `MainView` und erbt von `Composite<VerticalLayout>`, das alle Komponenten darin untereinander anordnet. `@PageTitle` setzt den Titel der Seite im Header.

Die `AboutView` ist also als `Component` in die `MainView` eingehängt. Die `MainView` ist in *Listing 2* zu sehen. Sie erbt von `AppLayout`, einem vorgefertigten Layout, das bereits Plätze für Header, Menü und Content mitbringt und sich sowohl auf dem Desktop als auch mobil responsive verhält. So klappt sich beispielsweise das linke Menü automatisch ein, sobald der Platz nicht mehr ausreicht. Dann kann es bei Bedarf über einen Klick auf das Hamburger-Symbol als schwebendes Menü geöffnet werden. Das `AppLayout` ist somit eine ideale Basis für eine responsive App, die einiges an Styling und Layouting übernimmt. Der `@CssImport` legt den Pfad zu einem Stylesheet fest. Dieses wird nur dann im Browser angewandt, wenn die Component sichtbar ist. Da die `MainView` immer sichtbar ist, gilt der angegebene Stil in `shared-styles.css` immer. Das Vaadin Standard Theme ist *Lumo*. Durch die Annotation `@Theme` wurde das ebenfalls mitgelieferte *Material* Theme geändert und von „light" auf „dark" umgestellt. Schon erstrahlt die App in einem dunklen Design.

Als Letztes kommt die `@PWA`-Annotation, die die zur Konfiguration einer Progressive Web App nötigen Seiten-Header hinzufügt und automatisch eine `manifest.webmanifest` erzeugt. Außerdem wird eine Standardseite hinterlegt, für den Fall, dass die Webseite offline ist. Dem Browser wird somit mitgeteilt, dass sich die Seite gut dazu eignet, sich als App in das jeweilige Betriebssystem zu integrieren. Chrome zeigt dann sowohl unter Android als auch unter Windows einen Hinweis an, die App zu installieren *(siehe Abbildung 2)*. Beim Installieren wird dann eine Verknüpfung auf dem Startbildschirm mit dem passenden Icon hinterlegt. Dieser öffnet die App dann in einem Modus ohne Titelleiste und ohne Tabs, sodass sich die Seite mehr wie eine lokal installierte App anfühlt.

Im Konstruktor der `MainView` werden Header (Navbar) und das linke Seitenmenü (Drawer) initialisiert. Der Header bekommt ein Hamburger-Button zum Ein- und Aufklappen des Menüs und eine H4-Überschrift. Das Menü besteht aus horizontal angeordneten Links. Diese führen zu Seiten, die bei Aufruf im Content der `MainView` angezeigt werden – wie auch die eingangs erwähnte `AboutView`. *Listing 2* zeigt außerdem, wie sich mit Vaadin grundsätzlich Weboberflächen in Java zusammenbauen lassen. Komponenten werden konfiguriert und dem HTML-Baum entsprechend hierarchisch zusammengefügt. Das Framework kümmert sich dann darum, die entsprechenden HTML-Elemente zu rendern, diese passend zu stylen und das Eventmanagement zwischen Front- und Backend zu übernehmen.

Typsicheres Databinding

Die `CompendiumView` zeigt die Clean-Code-Regeln in tabellarischer Form mit den Spalten „Name" und „Grad" an. Ein Textfeld erlaubt die schnelle Suche nach einer Regel. Dabei wird nicht nur der Tabelleninhalt, sondern auch der Beschreibungstext der Regel durchsucht. Ein Klick auf eine Regel öffnet diese in einer Extra-Ansicht.

Die Tabelle wird über die Methode `createTable` angelegt *(siehe Listing 3)*. Das Modell wird per Parameter als Collection übergeben.

```
@CssImport("./styles/shared-styles.css")
@Theme(value = Material.class, variant = Lumo.DARK)
@PWA(name = "Clean Code Developer Journal")
public class MainView extends AppLayout {

    public MainView() {
        addToNavbar(new DrawerToggle());
        addToNavbar(new H4("Clean Code Journal"));
        addToDrawer(createMenuBar());
    }

    private VerticalLayout createMenuBar() {
        var journal = new RouterLink("Journal", JournalView.class);
        var profile = new RouterLink("Profil", ProfileView.class);
        var achievements = new RouterLink("Erfolge", AchievementsView.class);
        var compendium = new RouterLink("Kompendium", CompendiumView.class);
        var about = new RouterLink("Über", AboutView.class);
        return new VerticalLayout(journal, profile, achievements, compendium, about);
    }

}
```

Listing 2

```
public class GradeTopic implements Serializable {
    private String name;
    private String description;
    private Grade grade;
    // [... ] inkl. [getter, setter, equals, hashcode, toString]
}

enum Grade {Black, Red, Orange, Yellow, Green, Blue, White}


private Grid<GradeTopic> createTable(Collection<GradeTopic> gradeTopics) {
    Grid<GradeTopic> table = new Grid<>();
    table.setItems(gradeTopics);

    table.addColumn(GradeTopic::getName).setHeader("Name").setSortable(true);

    table.addColumn(this::formatTopic).setHeader("Grad").setSortable(true);

    table.addSelectionListener(event -> event.getFirstSelectedItem().
            ifPresent(this::openGradeTopic));
    return table;
}

private String formatTopic(GradeTopic topic) {
    return topic.getGrade().getGradeColor().name();
}
```

Listing 3

Eine Tabelle heißt in Vaadin „Grid". Das Grid ist per Generic auf ein Modell typisiert (`GradeTopic`). Dadurch sind das Anlegen von Spalten bis hin zum Event typsicher dem Modell zugewiesen. Ein Aufruf von `setItems`, um die Daten in das Grid zu laden, ist daher nur mit `List<GradeTopic>` möglich, `List<String>` würde der Compiler nicht zulassen. In `addColumn` wird eine Spalte hinzugefügt. Dabei wird eine Funktion angegeben, die den Inhalt der Zelle zurückgibt (`GradeTopic::getName`), den Spaltennamen festlegt (`"Name"`) und die Sortierung erlaubt. Um auf Klicks auf eine Spalte zu reagieren, wird der Tabelle noch ein `SelectionListener` hinzugefügt. Schon haben wir unser Modell an die Tabelle gebunden.

I18-N

Die App soll mehrsprachig sein und die vom Browser angefragte Sprache berücksichtigen. Dazu steht in jeder `Component` die Methode `getTranslation(String key, Object... params)` zur Verfügung. Darin wird mit einem Key bei einem `I18NProvider` eine Übersetzung angefordert. Dank der *Spring*-Integration lässt sich der `I18Nprovider` leicht als Service bereitstellen *(siehe Listing 4)*. Dieser lädt Übersetzungen mittels `ResourceBundle` aus den Property-Dateien `translation_de.properties` und `translation_en.properties`. Um die Leserlichkeit zu fördern, sind in den Listings aber alle `getTranslation(...)`-Aufrufe durch die tatsächliche deutsche Übersetzung ersetzt.

Das Element-API

Der farbige Fortschrittsbalken *(siehe Abbildung 3)* soll auf einen Blick anzeigen, an welchen Tagen ein Anwender einen Log-eintrag angelegt hat. Der Tag ist grün, wenn geloggt wurde, und grau, wenn nicht. Bei Mouseover wird das Datum des Tages als Beschreibungs-Popup angezeigt. Dazu reichen ein einfacher HTML-Aufbau *(siehe Listing 5)* und ein wenig CSS aus. Ein horizontales Flex-Layout richtet alle Elemente gleichverteilt aus. Jeder Tag ist ein extra `DIV`, der, wenn er grün sein soll, zusätzlich die Klasse `day-submitted` erhält. Der `SPAN` wird dank der Klasse `tooltiptext` ausgeblendet und erst beim Mouseover unterhalb angezeigt. (Das CSS der Klassen ist zur Übersicht nicht mit aufgeführt.)

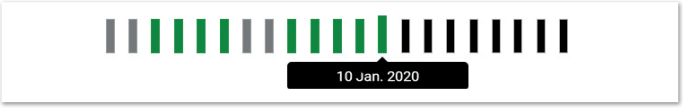


Abbildung 3: Fortschrittsanzeige, die mit Hilfe der Element-API implementiert ist. (© Tobse Fritz)

```
@Service
public class TranslationProvider implements I18NProvider {

    @Override
    public List<Locale> getProvidedLocales() {
        return List.of(Locale.ENGLISH, Locale.GERMAN);
    }

    @Override
    public String getTranslation(String key, Locale locale, Object... params) {
        var bundle = ResourceBundle.getBundle("translation", locale);
        return bundle.getString(key);
    }
}
```

Listing 4

```
<div class="grade-progress"
     style="display: flex;
     width: 100%;
     justify-content: space-evenly;
     align-items: flex-end;">

    <div class="grade-progress-day tooltip">
        <span class="tooltiptext">02 Jan. 2020</span>
    </div>
    <div class="grade-progress-day tooltip day-submitted">
        <span class="tooltiptext">03 Jan. 2020</span>
    </div>
    <div class="grade-progress-day tooltip day-submitted">
        <span class="tooltiptext">04 Jan. 2020</span>
    </div>
    [...]
</div>
```

Listing 5

Das Element-API bildet HTML-Elemente aus dem DOM im Java-Code ab. Mit ihr lassen sich Komponenten ändern oder auch neue Komponenten aufbauen. Ein solcher Aufbau ist in *Listing 6* zu sehen. Die Klasse `GradeProgressBar` ist die Basis mit dem `FlexLayout`, darin enthalten die Tage, die das Datum mit einem Tooltip anzeigen – `ProgressDay`. Das nötige CSS steht im globalen shared-styles.css (`@CssImport` Annotation an `MainView`). Um einen Tag als geloggt zu markieren, gibt es die Methode `setDaySubmitted()`, die an das `DIV` die CSS-Klasse `day-future` anfügt.

Bauklötzchen für das Web

Eine komponentenorientierte Anwendungsentwicklung in voneinander unabhängigen und wiederverwendbaren Bausteinen ist nicht erst seit Clean Code das hohe Ziel einer guter Softwarearchitektur. Im Web war dies jedoch lange Zeit schwierig zu erreichen. JavaScript lässt sich zwar modularisieren, hat aber trotzdem immer Zugriff auf alle anderen Funktionen von JS Imports. Bei CSS ist es noch schwieriger, denn eine einzelne zusätzliche Regel eines externen Moduls kann schnell das gesamte Look-and-Feel einer Seite zunichtemachen. UI-Komponenten aus unterschiedlichen Web-Frameworks lassen sich indes nicht ohne Weiteres miteinander kombinieren. So lassen sich beispielsweise Komponenten aus *vuetifyjs* oder *Vue Material* (wie die Namen andeuten) nur mit *Vue-JS* nutzen, nicht aber mit *Angular*.

Genau wegen dieses Dilemmas wurden die Web Components spezifiziert. Darunter ist eine Ansammlung von Webstandards gemeint, die es erlauben, Komponenten in JS, CSS und HTML-Templates zu entwickeln, die komplett isoliert von der restlichen Website agieren können. So besteht beispielsweise keine Gefahr mehr, dass durch den Import einer Web Component der Style der restlichen Seite beeinflusst wird. Andersherum ist auch der Inhalt einer Web Component geschützt und wird durch das CSS der Hauptseite nicht beeinflusst. Das funktioniert nur über CSS-Variablen, die eine Komponente als klare Schnittstelle definieren muss. Jede Web Component hat ihr eigenes HTML-Tag. So können sie im HTML, analog zu bekannten Tags wie `BUTTON` oder `INPUT`, verwendet werden. Google bietet mit Polymer eine Bibliothek für den Bau von Web Components an. Die folgenden zwei Kapitel zeigen, wie in Vaadin ein solches Polymer Template angelegt wird und wie sich eine externe Web Component anbinden lässt.

```
public class GradeProgressBar extends FlexLayout {

    public GradeProgressBar() {
        setWidthFull();
        setJustifyContentMode(JustifyContentMode.EVENLY);
        setAlignItems(Alignment.END);
        addClassName("grade-progress");
    }

    public static class ProgressDay extends Div {

        public ProgressDay(String tooltip) {
            addClassName("grade-progress-day");
            addToolTip(tooltip);
        }

        private void addToolTip(String tooltip) {
            addClassName("tooltip");
            Span span = new Span(tooltip);
            span.addClassName("tooltiptext");
            span.addClassName("tooltip-bottom");
            add(span);
        }

        public void setDaySubmitted() {
            addClassName("day-submitted");
        }

        public void setDayInFuture() {
            addClassName("day-future");
        }

    }

}
```

Listing 6

Polymer Templates

Unter dem Menüpunkt „*Erfolge*" sind in der App Achievements aufgelistet, die ein Entwickler für sich verbuchen kann. So gibt es beispielsweise eine Belohnung in Form von Erfahrungspunkten für „*Führe ein Code Review durch*" *(siehe Abbildung 4)*. Eine solche Zeile wird folgend als Polymer Template angelegt. Der Template Code wird dabei nicht in Java, sondern in einer JavaScript-Datei als `PolymerElement` abgelegt *(siehe Listing 7)*. Das Template enthält CSS und HTML und wird als String von der Funktion `get template()` zurückgegeben. Darin enthalten sind das Styling innerhalb eines `style`-Blocks, direkt gefolgt vom Template-Code, dem eigentlichen `<div id="achievement">`. Dabei werden Variablen, die aus Sicht des Clients „read-only" sind, in doppelten eckigen Klammern aufgeführt.

Zu dem `PolymerElement` in JavaScript gibt es den passenden Adapter im Java-Code *(siehe Listing 8)*. Dieser fällt übersichtlich aus. Die Annotation `@Tag("achievement-line")` gibt an, als welcher HTML-Tag die Komponente später im HTML angezeigt wird. Achtung bei der Namensgebung: Per Konvention muss der Name ein Minus enthalten. Fehlt nur noch der Link zur `achievement.js`-Datei mit `@JsModule("./src/achievement.js")`. Die dritte Zeile nennt eine

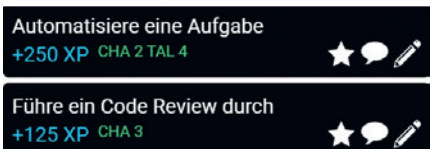


Abbildung 4: Achievement, das mit Hilfe eines Polymer Templates implementiert ist. (© Tobse Fritz)

```
import {html, PolymerElement} from '@polymer/polymer/polymer-element.js';
import '@polymer/iron-icon/iron-icon';

class Achievement extends PolymerElement {

    static get template() {
        return html`<style>
    #title {
        font-size: 18px;
    }

    #achievement {
        background-color: #101217;
        border-radius: 5px;
        padding: 8px;
    }

    [...]

</style>

<div id="achievement">
    <div id="title">[[title]]</div>
    <div id="footer">
        <div id="experience">
            <div id="xp">[[experience]]</div>
            <div id="skill-points">[[skills]]</div>
        </div>
        <div id="actions">
            <iron-icon class="action" icon="vaadin:star"></iron-icon>
            <iron-icon class="action" icon="vaadin:comment"></iron-icon>
            <iron-icon class="action" icon="vaadin:pencil"></iron-icon></div>
         </div>
    </div>
</div>`;
    }

    static get is() {
        return 'achievement-line';
    }
}

customElements.define(Achievement.is, Achievement);
```

Listing 7

```
@Tag("achievement-line")
@JsModule("./src/achievement.js")
@NpmPackage(value = "@polymer/iron-icons", version = "3.0.0")
public class AchievementComponent extends PolymerTemplate<AchievementModel> {

    @Override
    public AchievementModel getModel() {
        return super.getModel();
    }
}

public interface AchievementModel extends TemplateModel {
    String getExperience();
    void setExperience(String experience);
    String getSkills();
    void setSkills(String skills);
    String getTitle();
    void setTitle(String title);
}
```

Listing 8

Abhängigkeit für den *Node* Package Manager, nämlich „iron-icons“: `@NpmPackage(value="@polymer/iron-icons",version="3.0.0")`. Dadurch wird Vaadin mitgeteilt, dass bei Verwendung der Komponente auch das `iron-icons` JavaScript-Modul als Dependency von *npm* mit herunterladen werden muss.

Einbindung von Web Components

Das untere rechte Menü ist ein „SpeedDial“ *(siehe Abbildung 5)*, ein absolut positioniertes Kreis-Menü, das sich aufklappen lässt. Dieses gibt es online *[5]* schon als fertige *Polymer* Web Component. Wie lässt sich dieses nun in die App einbinden? Wie sich ein eigenes Polymer Template nutzen lässt, wurde schon gezeigt. Bei externen Komponenten ist das fast das Gleiche, nur wird kein JavaScript, sondern nur der Java Adapter gebraucht. Der ist aufgebaut wie beim Template *(siehe Listing 9)*: mit Tag (`@Tag`), Link zum Package (`@NpmPackage`) und der Referenz zum eigentlichen JavaScript-Modul (`@JsModule`). Das Menü hat jedoch auch noch Funktionen, etwa dafür, dass es auch wieder geschlossen werden kann. Dazu gibt es noch ein `close()`, das die Variable `opened` auf `false` setzt. Das Menü ergibt zudem nur Sinn, wenn es auch Menüeinträge enthalten kann. Die heißen in diesem Falle `SpeedDialAction` *(siehe Listing 10)*. Text und Icon des Eintrags sollten sich natürlich anpassen lassen. Da die Klasse von Label erbt, ist die Methode `setText()` schon vorhanden; fehlt nur noch die Methode `setIcon`. Diese übernimmt von einem Vaadin-Icon das `Icon`-Attribut. Mehr ist nicht nötig, um das Menü einzubinden.

Fallstricke

Bei der Umsetzung der App gab es auch die eine oder andere Überraschung. Die wichtigsten drei waren:

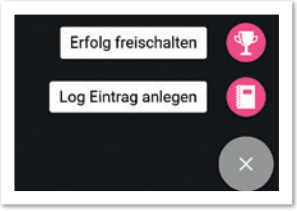


Abbildung 5: Speed-Dial-Menü, das als Web Component eingebunden ist. (© Tobse Fritz)

1. Eine `View` muss serialisierbar sein. Vaadin speichert den State einer `View` dadurch, dass diese komplett serialisiert wird. Darum muss sowohl jedes Feld einer View als auch ein View-Modell Javas `Serializable` Interface implementieren. Ansonsten erscheint beim erneuten Laden der Seite eine `UnwriteableSessionDataException`.
2. Der URL-Pfad einer View, der durch `@Path`-Annotation angeben wird, muss eindeutig sein. Werden, wie in diesem Beispiel, auf der Startseite `MainView` und `JournalView` gleichzeitig angezeigt, muss die `@RouteAlias`-Annotation zur Pfadangabe verwendet werden.
3. Views sind speicherhungrig. Die Erfolge-Seite zeigt 37 Zeilen mit der Achievements Web Component an. Alle Daten, die in dieses Template eingesetzt werden, sind gerade einmal 1,72 KByte groß. Trotzdem benötigt der Aufbau der Seite kurzzeitig bis zu 193 MB an Arbeitsspeicher (Used Heap). Vor dem Produktivbetrieb sollte daher immer anhand von Lasttests untersucht werden, ob die tatsächliche Speicherauslastung für das angestrebte System und die erwartete Nutzerzahl tragfähig ist.

```
@Tag("paper-fab-speed-dial")
@NpmPackage(value = "@cwmr/paper-fab-speed-dial", version = "3.0.0")
@JsModule("@cwmr/paper-fab-speed-dial/paper-fab-speed-dial.js")
public class SpeedDial extends Component {

    public SpeedDialAction addMenuItem(String item, Icon icon) {
        SpeedDialAction speedDialAction = new SpeedDialAction(item, icon);
        getElement().appendChild(speedDialAction.getElement());
        return speedDialAction;
    }

    public void close() {
        getElement().executeJs("this.opened=false");
    }

}
```

Listing 9

```
@Tag("paper-fab-speed-dial-action")
@NpmPackage(value = "@cwmr/paper-fab-speed-dial", version = "3.0.0")
@JsModule("@cwmr/paper-fab-speed-dial/paper-fab-speed-dial-action.js")
public class SpeedDialAction extends Label implements HasEnabled {

    public SpeedDialAction(String text, Icon icon) {
        super(text);
        setIcon(icon);
    }

     private Element setIcon(Icon icon) {
        String iconAttribute = icon.getElement().getAttribute("icon");
        return getElement().setAttribute("icon", iconAttribute);
    }


    public Registration addClickListener(ComponentEventListener<SpeedDialActionClickEvent> listener) {
        return addListener(SpeedDialActionClickEvent.class, listener);
    }

}
```

Listing 10

Fazit

Die Beispiel-App zeigt, wie vielseitig sich in Vaadin Oberflächen entwickeln lassen. Mit Bordmitteln benötigt es nur wenige Annotationen, um die App als Progressive Web App auszuweisen. Diese lässt sich in wenigen Schritten installieren und ist dank der Standard-Komponenten vom Anfang an responsive ausgelegt. Mit dem `AppLayout` steht ein Basis-Layout bereit, das sowohl mobil als auch auf dem Desktop eine gute Figur macht. Auch besteht keine Sorge, durch das Framework in der Frontend-Entwicklung eingeschränkt zu sein. Dank des Element-API steht auch auf Java-Seite eine direkte Schnittstelle für Operationen auf dem DOM bereit. Zudem bieten Templates und die Anbindung an Web Components auch die Freiheit, Komponenten direkt in HTML und JS zu entwickeln und dann komfortabel im Java-Code verfügbar zu machen. Da Vaadin dabei auf etablierte Standards wie Polymer setzt, steht eine große Anzahl fertiger Komponenten zur Verfügung. Frontend-Entwickler können zudem gut in den Workflow eingebunden werden, um noch fehlende Komponenten oder Templates unabhängig von Vaadin zu entwickeln. Java-Entwickler fühlen sich heimisch und können ihre bekannten Bibliotheken und Frameworks einsetzen. Vaadin 14 bietet dem ambitionierten Entwickler damit eine zukunftssichere Basis, um in Java moderne Webanwendungen zu entwickeln.

Referenzen

[1] *https://vaadin.com/components*

[2] Eine Initiative für mehr Professionalität in der Software-Entwicklung: *https://clean-code-developer.de*

[3] *http://clean-code.rocks*

[4] *https://vaadin.com/start/latest*

[5] *https://www.webcomponents.org/element/@cwmr/paper-fab-speed-dial/elements/paper-fab-speed-dial*

Tobse Fritz

ITscope GmbH

tfr@itscope.de

Tobse Fritz entwickelt seit fünf Jahren bei ITscope mit an einer B2B-Business-Plattform. Diese wird seit 2011 als Webapplikation mit Vaadin betrieben und hat aktuell mehr als 8.000 aktive Nutzer. Er ist Kotlin-Fanboy und verbreitet seine Begeisterung gerne in den lokalen User Groups rund um Karlsruhe. Besonderen Spaß an der Entwicklung hat er bei der Teilnahme an Game Jams und bei der Förderung von Kindern und Jugendlichen. In seiner Freizeit tobt er sich beim Bouldern aus, fotografiert leidenschaftlich, sitzt bei einer Brettspielrunde oder bastelt mit 3D-Drucken am ersten Perpetuum Mobile.

Es Fluttert gewaltig

René Jahn, SIB Visions GmbH

Wer eine App für mobile Geräte entwickeln möchte, steht zumindest vor einer wichtigen Frage: Mit welcher Technologie soll die App realisiert werden? Da es aktuell verschiedenste Möglichkeiten gibt, ist die Antwort nicht sofort klar. Mit diesem Problem war das Framework-Team von SIB Visions ebenfalls konfrontiert, als es darum ging, einen Client für das Open Source Application Framework JVx zu entwickeln. Wie es zu einer Entscheidung kam und vor allem was umgesetzt wurde, wird in diesem Artikel erklärt.

Das JVx Framework *[1]* ermöglicht die Erstellung von Datenbank-Applikationen für unterschiedlichste GUI-Technologien mit einer einzigen Source-Basis. Es ist sowohl eine GUI-Abstraktion als auch ein Full-Stack Application Framework. Die damit erstellten Applikationen laufen ohne Änderungen im Browser mittels JavaScript/CSS, am Desktop mit Swing oder JavaFX und auf mobilen Geräten mithilfe nativer Apps. Mit JVx können Applikationen unterschiedlichster Größe und Komplexität realisiert werden. Zwei Produkte, die mit JVx entwickelt wurden, sind beispielsweise SNOWsat Maintain *[3]* und VisionX *[4]*.

Wie soeben erwähnt, werden für die Ausführung auf mobilen Geräten native Apps eingesetzt. Es gibt eine App für iOS und eine andere für Android. Erfahrene App-Entwickler werden vermuten, dass die-

se Apps unabhängig voneinander entwickelt wurden. Das ist auch genauso. Die iOS-App wurde mit Objective-C und die Android-App mit Java entwickelt. Das war im Jahr 2014 auch ein gangbarer Weg, da es nur wenige Lösungen gab, die eine Entwicklung mit nur einer Source-Basis anboten. Noch dazu waren diese nicht sehr preiswert und auch nicht immer Open Source. Dem Framework-Team bei SIB Visions waren die beiden Apps immer ein Dorn im Auge, da sie doppelte Wartung und Weiterentwicklung bedeuteten. Man verfolgte über die Jahre hinweg immer wieder Ablösegedanken und verglich diverse Technologien. Im Jahr 2019 wurde die Ablöse beschlossen. Der Startschuss fiel im Herbst und das Team begann mit der Evaluierung der möglichen Technologien.

Die infrage kommenden Lösungen waren Xamarin *[5]*, React Native *[6]* und Flutter *[7]*. Im Vorfeld schloss man Lösungen wie Qt, Ionic, Phonegap, Sencha Touch, NativeScript und Gluon Mobile bereits aus. Andere Lösungen wie Jasonette, Weex, Framework7 oder CodeOne zog man ebenfalls nicht in Betracht.

Der Entscheidungsprozess für die geeignete Technologie war relativ unkompliziert und beruhte auf messbaren Kriterien, der Erfahrung des Framework-Teams und „Gefühl". Die Anforderungen:

- Die App performt wie eine native App
- Gerätefeatures wie Anruf, Geolocation und Kamerazugriff
- Ausreichend Komponenten, die für Datenbankapplikationen benötigt werden wie Tabelle, Datumauswahl, Auswahllisten
- Die Beispiel-Applikationen stürzen nicht ab und hinterlassen einen bleibenden Eindruck
- Open-Source-basiert oder frei für Open-Source-Projekte verwendbar
- Kurze Entwicklungszyklen
- Live-Voransicht ohne längere Wartezeiten
- Gutes Toolset mit IDE-Unterstützung
- Erweiterbar durch Add-ons, Plug-ins oder Ähnliche
- Einfache Versionsaktualisierungen und wenige Abhängigkeiten zu zusätzlichen Bibliotheken

Anhand dieser Anforderungen nahm man die einzelnen Technologien unter die Lupe. Bei einigen Kandidaten verhinderten einzelne Bedenken aus dem Framework-Team den Einsatz. Die meisten JavaScript- und Browser-basierten Technologien fühlten sich nicht nativ an und wurden deswegen ausgeschlossen. Andere verfügten nicht über die vollständigen Gerätefeatures oder der Source Code wirkte zu komplex und unverständlich. Andere wiederum erfüllten die kurzen Entwicklungszyklen nicht oder boten die gewünschte Live Preview nicht wie erwartet.

Letzten Endes blieben Xamarin, React Native und Flutter übrig. Obwohl Xamarin lange Favorit war, waren die langen Wartezeiten ein entscheidender Nachteil, wodurch sich die Entwickler nicht produktiv genug fühlten. Natürlich waren dies reine Empfindungen und damit schwer messbar, aber es sollte sich als richtig herausstellen. Der Grund, warum React Native zu den Favoriten zählte, war, dass bereits Erfahrung im Team existierte und eine produktive Applikation damit umgesetzt wurde. Doch letzten Endes führten die Erfahrungen bei der App-Entwicklung auch hier zum Ausscheiden. Es gab immer wieder interne Probleme, für die Workarounds gesucht werden mussten, das Fokus-Handling war nicht wie gewünscht und der Gesamteindruck war nicht zufriedenstellend. So blieb am Ende also Flutter übrig. Doch nicht, weil es der letzte Kandidat war, sondern weil man von Anfang an den Eindruck hatte, dass es sich um die richtige Technologie handelt.

Warum Flutter?

Das Framework-Team hatte keinerlei Erfahrungen mit Flutter. Auch die Programmiersprache Dart, mit der Flutter entwickelt wurde, war komplettes Neuland. Doch durch die Nähe zu Java stellte Dart kein Problem dar. Im Gegenteil, die Sprachfeatures machten Spaß und das Framework-Team kam gut damit zurecht. Als Entwicklungsumgebung kamen Android Studio und VSCode infrage. Egal, welche IDE eingesetzt wurde – die Produktivität war gegeben.

Einer der Wow-Effekte entstand durch die Live Preview und das Hot Reload Feature bei Änderungen am Code. Auch wenn es nicht immer problemlos funktionierte, so fühlte sich die Entwicklung rasend schnell an. Unabhängig von der Entwicklung beeindruckte die relativ junge Technologie mit einer Vielzahl von Erweiterungen und mit einem modernen, jedoch nicht neuen Ansatz der GUI-Entwicklung.

Doch Flutter erfüllte nicht nur die Anforderungen vollständig und schaffte ein gutes Gefühl, es passte auch von den internen Konzepten perfekt zu JVx. Natürlich war von Anfang an klar, dass es sich um eine relativ neue Technologie handelt und dass die Zukunftssicherheit offen ist. Doch das Entwicklungsteam hatte entschieden und schlussendlich muss es auch damit arbeiten wollen.

Der Ansatz, den Flutter verfolgt, ist ähnlich zu JVx, da es ebenfalls das GUI abstrahiert und unterschiedliche Implementierungen bietet. Davon bekommt der Entwickler jedoch im besten Falle nichts mit und wundert sich, warum das auf den unterschiedlichen Geräten so reibungslos funktioniert. Die Probleme mit den zugrunde liegenden Technologien werden vom Framework gelöst und der Entwickler kann sich auf die App-Entwicklung konzentrieren. Diese Philosophie wird auch von JVx verfolgt.

Bei Flutter handelt es sich vorranging nicht um eine Technologie, mit der grafisch anspruchsvolle Spiele entwickelt werden sollen. Auch wenn dies möglich ist und durch Low-Level APIs unterstützt wird, so liegt der Fokus dennoch auf der Entwicklung von Produktivitäts-Apps. Wer an dieser Stelle mehr über das Design von Flutter erfahren möchte, sollte sich ein Video von Ian Hickson *[8]* nicht entgehen lassen.

Es muss eine App sein

Mit Flutter sollte also eine App für JVx entwickelt werden. Diese musste am Ende des Tages eine JVx-Applikation auf mobilen Geräten darstellen. Hier könnte die Frage entstehen, warum die App nicht gleich ohne JVx entwickelt wurde. Das wäre möglich gewesen, allerdings bietet JVx die Möglichkeit, eine Applikation mit unterschiedlichen Technologien darzustellen, beispielsweise als Webanwendung, am Desktop oder auch Headless für automatisiertes Testen. Auch das sollte mit Flutter möglich sein, jedoch ist beispielsweise die Web-Implementierung noch nicht ausgereift. Daher ist die App ein guter Anfang, um in Zukunft eventuell vollständig auf Flutter zu setzen. Im Unterschied zu Flutter ist man mit JVx jedoch immer technologieunabhängig und wenn nötig, wird ganz einfach eine neue Technologie angebunden. Die damit entwickelte Applikation muss dann nicht neu entwickelt werden. Diesen entscheidenden Vorteil bietet eben nur JVx.

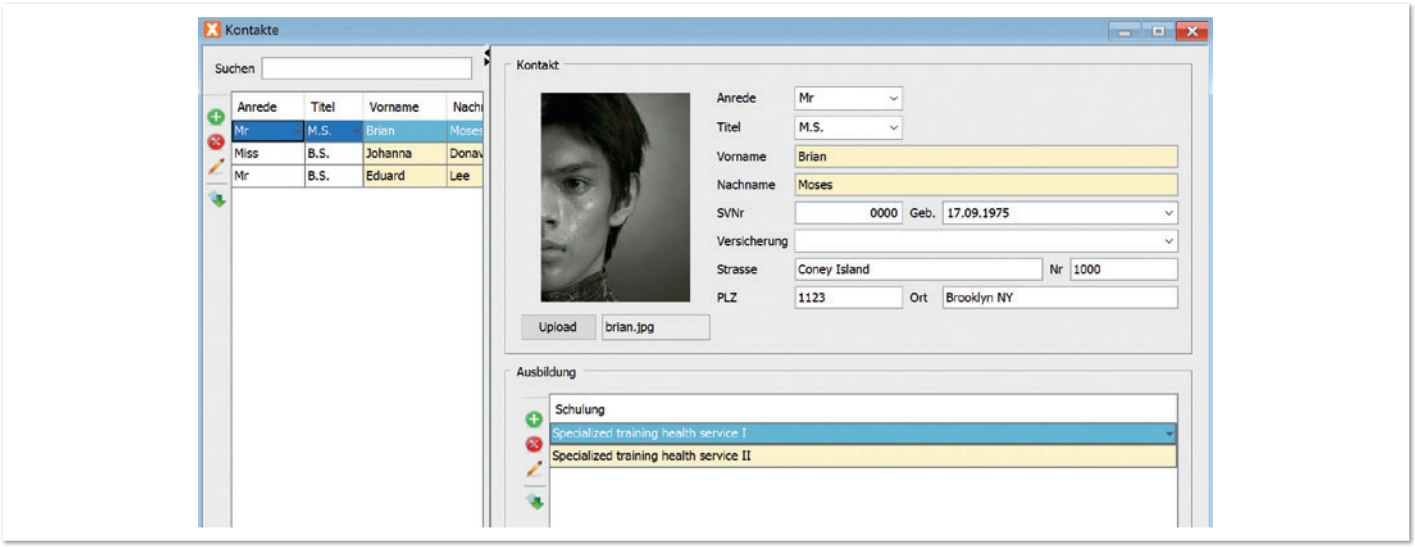


Abbildung 1: JVx-Applikation am Desktop (© René Jahn)

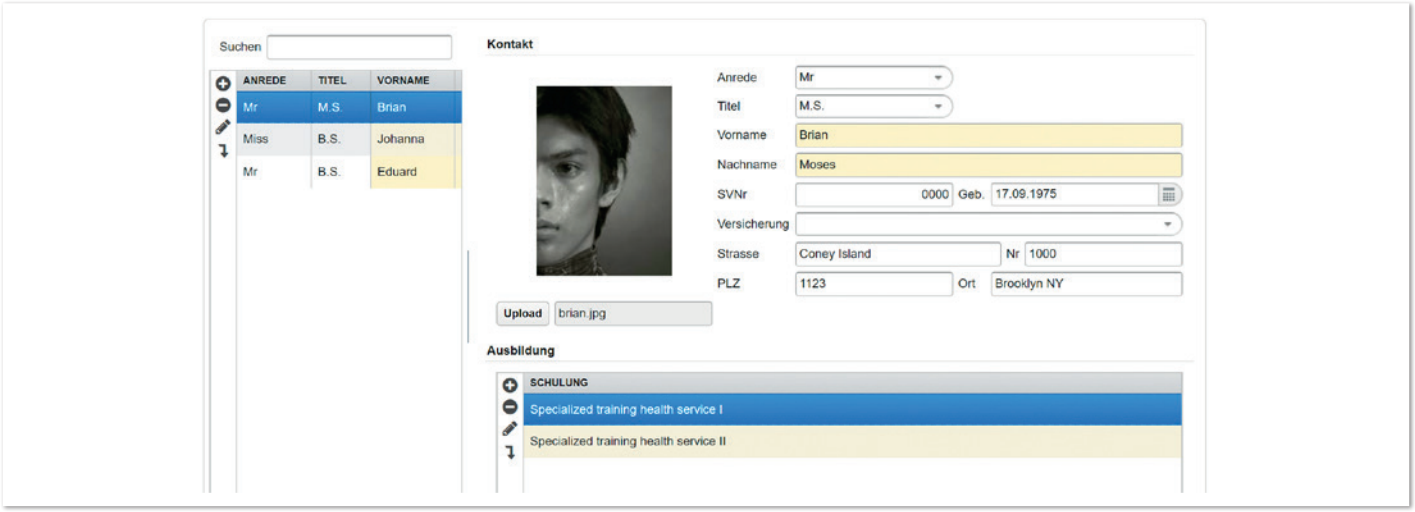


Abbildung 2: JVx-Applikation am Desktop, volle Größe (© René Jahn)

Abbildung 3: JVx-Applikation im Web, responsive Liste (© René Jahn)

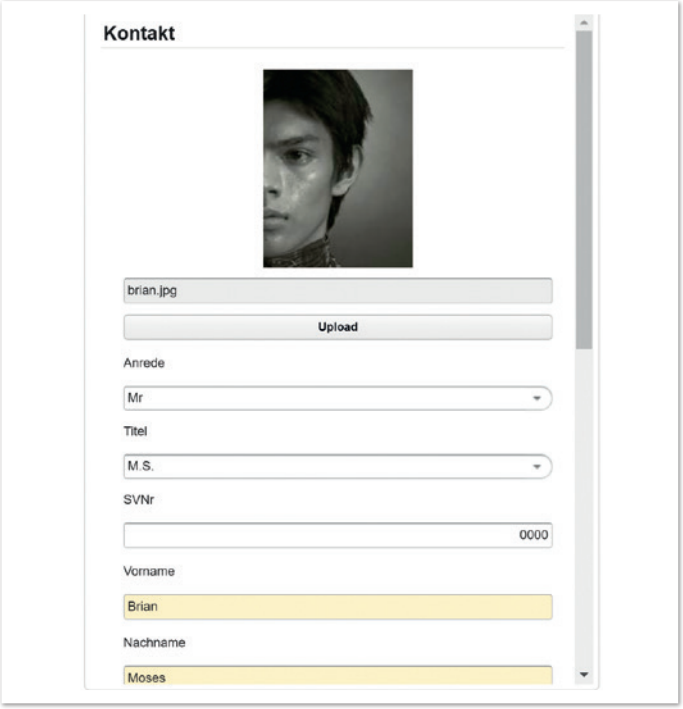


Abbildung 4: JVx-Applikation im Web, responsive Details (© René Jahn)

Um einen besseren Eindruck zu bekommen, sehen wir uns jetzt – anhand einer Beispielapplikation – eine JVx-Applikation für Desktop, Web und als Flutter-App an. Die Applikation verwaltet lediglich eine Liste von Kontakten, wie in der Desktop-Variante *(siehe Abbildung 1)* zu sehen ist.

Bei der Desktop-Variante ist im linken Bereich eine Liste von Kontakten zu sehen, im rechten die Kontaktdetails. Am Desktop ist ausreichend Platz für die gleichzeitige Darstellung aller Informationen. Die Web-Variante ist hier ein wenig ausgeklügelter, denn es gibt einerseits die volle Darstellung *(siehe Abbildung 2)*. Andererseits ist man es im Web gewohnt, dass Inhalte responsive sind. Die Kontakteverwaltung wird daher bei geringer Bildschirmgröße automatisch an die Platzverhältnisse angepasst (siehe Abbildungen 3 und 4). In *Abbildung 3* ist lediglich die Kontakteliste zu sehen. Erst, wenn ein Kontakt gewählt wird, erscheinen die Details wie in *Abbildung 4*. Die Beispielapplikation läuft natürlich auch auf mobilen Geräten im Browser ohne Probleme, allerdings bleibt es eine Web-Applikation und das fühlt sich nicht nativ an.

Wir werfen jetzt einen Blick auf die Flutter-App. In *Abbildung 5* ist ebenfalls nur die Kontakteliste zu sehen und bei Auswahl eines Kontaktes werden auch hier die Details angezeigt *(siehe Abbildung 6)*.

Der optische Unterschied ist aufgrund des Material-Designs deutlich zu sehen und auch die Navigationsmöglichkeit von mobilen Geräten ist im Header vorhanden. Aber im Großen und Ganzen ist die Darstellung ähnlich wie im Browser. Das war auch der Anspruch der Entwickler. Die Applikation soll zwar mit unterschiedlichen Technologien ausgeführt werden, aber optisch und funktional weitestgehend vergleichbar sein.

Aller Anfang ist schwer

Als mit der Umsetzung der App begonnen wurde, war die Euphorie im Team riesig, da auf das richtige Pferd gesetzt wurde und Flutter frisch und modern wirkte. Das mag auch stimmen, aber leider kam es bei der Entwicklung häufig zu unerwarteten Problemen. Um nur ein paar zu nennen:

- Die Daten von Auswahllisten können nicht dynamisch nachgeladen werden. Wenn man versucht, das zu umschiffen, kommt es vor, dass die Auswahlliste geschlossen wird.
- Das Widget für die Anzeige einer Auswahlliste [10] hat keinen Event, um zu notifizieren, dass die Auswahlliste geöffnet wird.
- Von Flutter werden Tabellen-Widgets [11][12] angeboten. Dort wurde dynamisches Nachladen von Daten aber nicht vorgesehen beziehungsweise funktionierte nicht richtig.
- Es gibt Textfeld-Formatierungsmöglichkeiten [13] um gegebenenfalls nur Nummern zu erlauben. Wenn man dann das Textfeld allerdings rechtsbündig ausrichtet, kommt es zu unschönen Effekten.

Um die Einschränkungen zu umgehen, setzte man eigene Widgets um. Das war aufgrund der umfangreichen Dokumentation von Flutter auch kein Problem. Eigene Konzepte konnte man durch Flutters offene Programmierung einfach umsetzen und es gibt keine Geheimnisse, auf die nicht zugegriffen werden könnte.

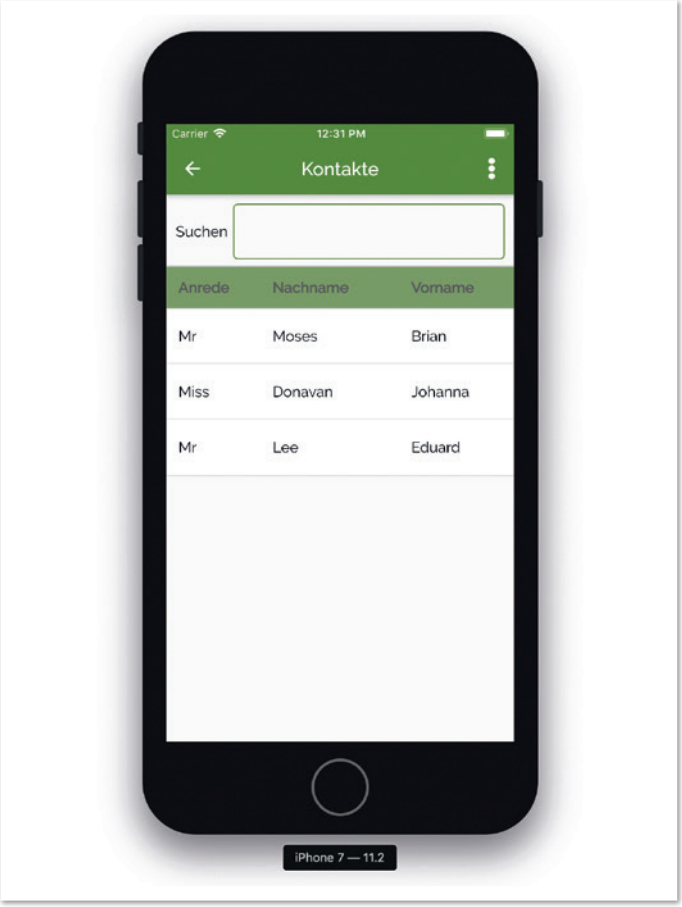


Abbildung 5: JVx-Applikation mit Flutter, Liste (© René Jahn)

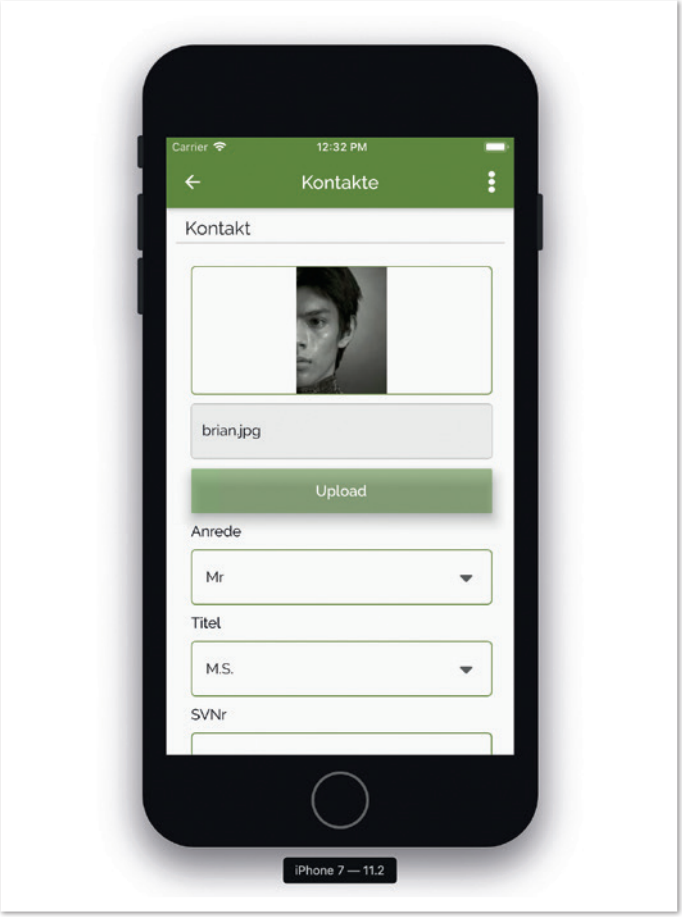


Abbildung 6: JVx-Applikation mit Flutter, Details (© René Jahn)

Dass einiges neu entwickelt werden musste, war dem Team von Anfang an klar, da Flutter diverse Layout Widgets *[14]* mitbringt, aber nicht die gewünschten. Es wäre zwar möglich, mit Schachtelungen das ein oder andere gewünschte Verhalten hinzubekommen, aber das macht die Layouts nur komplexer und erschwert die Wartung. Daher entwickelte man alle benötigten Layouts von Grund auf neu. In JVx gibt es nur wenige Layouts. Diese sind:

- BorderLayout
- FlowLayout
- FormLayout
- GridLayout

Das BorderLayout ist das einfachste von allen und entspricht dem Standard Java BorderLayout *[15]*. Das FlowLayout ist ebenfalls vergleichbar mit dem Standard Java FlowLayout *[16]* mit erweiterten Möglichkeiten zum Zeilenumbruch und Komponenten-Ausrichtung. Das FormLayout *[17]* ist ein Anker-basiertes Layout, das zeilen- und spaltenorientiert arbeitet, aber freie Platzeinteilung erlaubt. Das GridLayout ist zeilen- und spaltenorientiert, aber mit fixen Kachelgrößen.

Die Umsetzung der Layouts funktionierte wunderbar, da Flutter alles mitbringt, was für eigene Layouts benötigt wird. Das Framework-Team wunderte sich allerdings darüber, dass es nahezu keine Beispiele gab, wie eigene Layout Widgets umzusetzen sind. Im Web finden sich kaum bis keine Informationen oder sie sind einfach nur zu gut versteckt. Es blieb also nur der schwierige Weg über die manuelle Analyse des Flutter Source Codes und der darin enthaltenen Dokumentation. Im Nachhinein betrachtet, war das dann doch recht aufwendig und kostete Zeit. Wer nun wissen möchte, wie eigene Layout Widgets erstellt werden, der kann im Git Repository *[18]* stöbern. Hier wird es in Zukunft weitere Veröffentlichungen geben.

Nachdem man die meisten Widgets implementiert und die bekannten Einschränkungen gelöst hatte, konnte mit Volldampf an der Umsetzung der App gearbeitet werden. Das Geniale war, dass die Euphorie im Team anhielt und nicht gedämpft wurde. Das lag vor allem daran, dass Fortschritte schnell erzielt wurden und nicht mit hohem Aufwand verbunden waren. Die Entscheidung für Flutter wurde zu keinem Zeitpunkt angezweifelt.

Die Endausbaustufe

Wie zu Beginn des Artikels erwähnt, wurde VisionX mit JVx entwickelt. Dabei handelt es sich um eine Low-Code-Development-Plattform, um JVx-Applikationen zu erstellen. Die Applikationen basieren auf Open Source und können auch ohne VisionX weiterentwickelt werden. Dadurch, dass die Applikationen JVx verwenden, kann auch für dieses Produkt in Zukunft der Flutter Client eingesetzt werden. Die Vorteile sind gewaltig, da man mit VisionX eine Applikation erstellt, die dann im Browser, am Desktop und auf mobilen Geräten auf die gleiche Art und Weise funktioniert. Die Applikation kann jederzeit und mit jeder beliebigen Java-IDE (Eclipse, IntelliJ, NetBeans etc.) bearbeitet werden. Bei Bedarf ist es möglich, jede Technologie an die eigenen Bedürfnisse anzupassen. Sollte etwa der Funktionsumfang des Flutter Client nicht ausreichen, so kann man diesen ganz einfach an die eigenen Wünsche anpassen.

Fazit

Bei Flutter handelt es sich um eine neue Technologie, die rasend schnell die Entwicklergemeinde erobert. Die Möglichkeiten sind enorm und es fühlt sich gut an, damit zu entwickeln. Es ist eine Technologie, die Entwickler gerne einsetzen. Natürlich hat Flutter noch Luft nach oben und die Visionen sind groß. Auf die Web-Variante sollte besonders geachtet werden. Die Entscheidung für Flutter war, auch nach mehreren Monaten Bedenkzeit, die absolut richtige. Die Anforderungen konnten vollständig umgesetzt werden und es blieben keine Wünsche offen. Wir sind gespannt, was in Zukunft auf uns zufluttert.

Quellen

[1] *https://de.wikipedia.org/wiki/JVx_(Framework)*
[2] *https://www.informatik-aktuell.de/entwicklung/programmiersprachen/ein-gui-fuer-alle-faelle.html*
[3] *https://www.snowsat.com/aut/de/produkte/snowsat-maintain.html*
[4] *https://visionx.sibvisions.com/*
[5] *https://dotnet.microsoft.com/apps/xamarin*
[6] *http://www.reactnative.com/*
[7] *https://flutter.dev/*
[8] *https://www.youtube.com/watch?v=dkyY9WCGMi0*
[10] *https://api.flutter.dev/flutter/material/DropdownButton-class.html*
[11] *https://api.flutter.dev/flutter/material/DataTable-class.html*
[12] *https://api.flutter.dev/flutter/widgets/Table-class.html*
[13] *https://api.flutter.dev/flutter/services/TextInputFormatter-class.html*
[14] *https://flutter.dev/docs/development/ui/widgets/layout*
[15] *https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html*
[16] *https://docs.oracle.com/javase/tutorial/uiswing/layout/flow.html*
[17] *http://doc.sibvisions.com/jvx/reference#formlayout*
[18] *https://github.com/sibvisions/flutterclient/tree/master/lib/ui/layout*

René Jahn
SIB Visions GmbH
rene.jahn@sibvisions.com

René Jahn ist Mitbegründer der SIB Visions GmbH und Head of Research & Development. Er verfügt über langjährige Erfahrung im Bereich der Framework- und API-Entwicklung. Neben der Open-Source-Sparte bringt er seine Expertise auch in der Produktentwicklung ein. Seine Leidenschaft ist die Evaluierung neuer Technologien und Integration in klassische Business-Applikationen.

Cross-Plattform-Apps mit NativeScript

Markus Schlichting, Karakun AG

Mobile Anwendungen für verschiedene Plattformen aus einer Codebasis anzubieten, ist nicht nur aus Kostensicht reizvoll. Die Fokussierung auf eine zentrale Technologie kommt auch dem Entwicklungsteam entgegen und nützt damit der Qualität. Mit NativeScript steht ein Framework bereit, mit dem Entwickler ihr vorhandenes Angular-/TypeScript-Know-how nutzen können, um aus der gleichen Codebasis iOS-und Android-Apps sowie Web-Applikationen bereitstellen zu können.

Das 2010 federführend von Google ausgerufene Paradigma „Mobile First" hat seine Spuren hinterlassen. Bei einer großen Anzahl neu aufgelegter Projekte wird zumindest von einer Mobile-Kompatibilität des webbasierten UI ausgegangen – wenn nicht sogar die parallele Implementierung einer zusätzlichen App angestrebt wird. Dies gilt insbesondere für Anwendungsfälle, die nicht ausschließlich beim Sachbearbeiter auf dem Desktop, sondern zum Beispiel auf dem Werksgelände, beim Kunden im Wohnzimmer oder im Führerhaus eines LKW ablaufen.

Rund 200 Milliarden App-Downloads im Jahr 2018 *[1]* allein in den öffentlichen Stores Google Play und Apple AppStore sprechen für sich: Mobiles und ihre Applikationen sind mindestens so allgegenwärtig, wie es prophezeit wurde.

Cross-Plattform

Aus dieser Situation heraus müssen Entwickler also drei Plattformen gleichzeitig bedienen: die beiden dominanten Mobile-Plattformen (Android und iOS) und das Web – denn die Frage, ob eine Mobile-App komplett durch eine PWA (Progressive Web App) ersetzt werden kann oder ob man voll auf native Apps setzen muss, ist noch lange nicht beantwortet.

```
<ActionBar title="SampleApp" class="action-bar">
</ActionBar>

<GridLayout rows="*" height="1500px">
        <StackLayout top="0" left="0" width="100%" height="100%">
            <ScrollView class="page">
                <StackLayout>
                    <CardView *ngFor="let item of data" class="card">
                        <app-card [item]="item"></app-card>
                    </CardView>
                </StackLayout>
            </ScrollView>
        </StackLayout>

</GridLayout>
```

Listing 1: Beispiel für UI-Deklaration via XML-Template

Aber wie können Entwickler dieser Anforderung möglichst effizient gerecht werden? Dass man mit drei Teams ebenso viele Apps parallel baut und zu gleichwertigen, auf die jeweilige Plattform optimierten Ergebnissen kommt, ist sehr zweifelhaft und in der Praxis oft widerlegt worden. Gleich mehrere Frameworks versprechen, zumindest das Erstellen der beiden mobilen Applikationen zu vereinheitlichen. NativeScript geht darüber hinaus und stellt in Aussicht, mit nur einer Codebasis alle drei Plattformen bedienen zu können.

NativeScript

Das seit 2015 verfügbare Open Source Framework NativeScript erlaubt es, native Anwendungen für iOS und Android aus einer Codebasis zu erstellen. Da diese Codebasis auf Webtechnologien aufbaut, kann der vorhandene Code auch zumindest in Teilen für eine Webanwendung genutzt werden. Die Vorteile bezüglich Produktivität und Wartbarkeit sind deutlich.

Anfang 2020 steht NativeScript in der Version 6.3 bereit und folgt einem klaren Versionsrhythmus: eine Major-Version pro Jahr, dazwischen vier Minor Releases. Als plattformunabhängige Programmiersprache wird JavaScript beziehungsweise TypeScript eingesetzt und es besteht die Möglichkeit, zusätzlich Frameworks wie Angular, Vue.js, Svelte *[2]* oder sogar React zur Erstellung der Programmlogik zu verwenden. Letzteres sogar trotz der Konkurrenz von ReactNative.

Dieser Artikel fokussiert sich auf die Verwendung von Angular, da dies der Technologie-Stack ist, der im Hinblick auf Code- und Know-how-Wiederverwendung überwiegend zum Einsatz kommt und auch vom NativeScript-Projekt selbst in den Vordergrund gestellt wird.

Funktionsweise

NativeScript verwendet also keine WebViews, um das UI mit DOM-Elementen darzustellen. Stattdessen nutzt es einen Glue-Layer, um aus Angular-Code native Steuerelemente des jeweiligen OS anzusteuern und anzuzeigen. Dies ist möglich, da das Angular Framework seit Version 2 vom DOM entkoppelt und somit nicht mehr auf Browser als Renderer beschränkt ist.

So kann man also vorhandenes Know-how aus der Webentwicklung nutzen und die App in TypeScript erstellen – oder umgekehrt im Kontext von NativeScript aufgebautes Wissen für die Webentwicklung weiterverwenden.

Die UI-Elemente werden wie in *Listing 1* gezeigt mit XML-Templates beschrieben. Die verwendeten Widgets werden dabei von NativeScript auf native Steuerelemente gemappt *(siehe Abbildung 1)*. Das Styling erfolgt mit CSS, die Verwendung von SCSS ist in der Toolchain bereits vorbereitet. Eine vollständige Übersicht ist online zu finden *[3]*.

Der Einstieg

Den Einstieg in die Welt von NativeScript kann man auf zwei Wegen finden: dem NativeScript Playground *[4]* oder der Einrichtung der NativeScript CLI.

Beim NativeScript Playground handelt es sich um eine Web-IDE, mit der insbesondere kleine Beispiele und Prototypen schnell erstellt und ausprobiert werden können. Der springende Punkt des Playground ist, dass über eine dazugehörige App der im Playground erstellte Code direkt auf einem Android- oder iOS-Smartphone ausprobiert werden kann. Dafür erfolgt das Nachladen des Codes auf das Smartphone über einen im Playground angezeigten QR-Code.

Der für die Entwicklung vollständiger Apps nachhaltigere Weg ist die Einrichtung der NativeScript CLI auf dem eigenen Rechner. Ist diese einmal vorhanden, bietet das Kommando `tns create` einen Wizard zum Erzeugen eines neuen NativeScript-Projekts und die Möglichkeit, auf verschiedene Projektvorlagen aufzubauen. Diese Variante ist deshalb nachhaltiger, da man früher oder später im Zuge von Zertifikatverwaltung und Builds für Google Play und Apple AppStore auf den Funktionsumfang der CLI angewiesen ist.

Funktionale UIs

Für eine praktische App braucht es natürlich auch eine Verbindung zwischen Darstellung, Daten und Logik. Um dies zu implementieren, kommen die bekannten Angular-Paradigmen zum Einsatz. Angular-

Abbildung 1: Projektion generischer UI-Elemente auf native Widgets

Module, Components und Services können voll verwendet und Templates und Styles innerhalb der Komponenten genutzt werden. Für bekannte Basisbibliotheken, wie Router, Forms oder den HTTP-Client, gibt es für NativeScript angepasste Versionen, um der Funktionsweise außerhalb des Browsers gerecht zu werden. Diese verwenden die bekannten APIs, sodass dem direkten Einsatz, ohne ein abgewandeltes API adaptieren zu müssen, wenig im Wege steht.

Das Layout von UI-Komponenten ist nach wie vor eine anspruchsvolle Angelegenheit. Für NativeScript stehen eine Reihe von Layouts zur Verfügung, die diese Aufgabe für die beiden mobilen Plattformen vereinheitlichen. Diese reichen vom simplen StackLayout bis hin zu einer Implementierung von Flexbox, wie es aus dem Browser bekannt ist.

Zugriff auf native APIs

Die meisten Apps möchten von nativen APIs Gebrauch machen. NativeScript bietet für die am häufigsten verwendeten APIs wie beispielsweise UI-Komponenten, Layouts, FileSystem oder HTTP-Client Abstraktionen, sodass direkt im TypeScript Schnittstellen verfügbar sind, unter denen die OS-spezifischen APIs gekapselt werden. Für viele Schnittstellen, die nicht von NativeScript selbst abgedeckt werden, sind durch die Community gepflegte Plug-ins verfügbar (zum Beispiel „nativescript-fingerprint-auth" zur Verwendung biometrischer Merkmale für die Authentifizierung). Möchte man darüber hinaus direkt mit nativen Funktionen arbeiten, ist dies ebenfalls direkt aus TypeScript heraus möglich, da NativeScript Type Definitions für die APIs der Plattformen bereitstellt. Um den Code je Plattform entsprechend kapseln zu können, bietet NativeScript selbst Hilfsmittel, wie in *Listing 2* gezeigt. So kann man auch Features verwenden, die nur in Android, aber nicht auf iOS verfügbar sind.

Sehr nützlich ist es, dass auch bei den Steuerelementen, für die mit NativeScript eine Abstraktion für beide Plattformen bereitgestellt wird (zum Beispiel Button, Label, TextField etc.), jederzeit auf plattformspezifische Attribute und Funktionen zugegriffen werden kann.

Für einzelne Statements, die sich in solche if-Statements kapseln lassen, ist dieser Ansatz praktikabel, wartbar und damit akzeptabel. Für komplexeren Code (zum Beispiel für die Einbindung von umfangreicheren Komponenten), sollte man den Code in separate Dateien oder sogar ein Plug-in auslagern. In beiden Fällen erfolgt die Trennung der plattformspezifischen Implementierung durch Namenskonventionen für die Quellcodedateien, die Suffixe zur Trennung verwendet:

- EnhancedItemSwitch.ios.ts
- EnhancedItemSwitch.android.ts
- EnhancedItemSwitch.ts

Die Datei ohne Plattform-Suffix definiert dabei nur die Struktur (`declare class`), die Implementierung erfolgt in den Dateien mit spezifischem Suffix. Beim Erstellen der Artefakte für die jeweilige Plattform wird dann nur der Code mit dem passenden Suffix herangezogen.

Wirklich nativen Code, also Objective-C/Swift für iOS oder Java/Kotlin für Android, zu verwenden, ist natürlich auch möglich, indem man ihn als Library einbindet.

Testing

Automatisierte Tests sind aus der Softwareentwicklung nicht wegzudenken und werden natürlich auch bei der Entwicklung mit NativeScript voll unterstützt. Zum einen kann Logik und isolierte Teile des Codes mit den aus der Webentwicklung bekannten Tools (zum Beispiel Karma/Jasmine) getestet werden. Zur Ausführung wird ein Emulator gestartet (oder auf ein angeschlossenes Gerät zurückgegriffen), um den Code tatsächlich auf der JavaScript-VM der Zielplattform auszuführen und so während des Tests möglichst nahe an dem Zielsystem zu sein. Außerdem kann so auch Code, der auf plattformspezifische APIs zurückgreift, getestet werden.

Darüber hinaus möchte man gerade für Software mit einer so großen Zahl von Installationszielen wie mobilen Apps End-to-End-Tests (E2E) verwenden. Für NativeScript existiert mit nativescript-dev-appium *[5]* eine Integration für Appium, einem Open-Source-Testautomatisierungswerkzeug für Mobile Apps. Dieses bietet viele nützliche Features wie beispielsweise *findBy*-Methoden zum Lokalisieren von Elementen oder das einfache Ausführen von Actions (*tap, click, doubleTap, hold*) und Gesten (*scroll, swipe, drag*).

Um nicht selbst einen sehr großen Gerätepark pflegen zu müssen, kann man über die Vereinheitlichung der E2E-Tests mit Appium auf einschlägige Cloud Provider zurückgreifen, die sowohl virtualisierte als auch ferngesteuerte physikalische Hardware zum Testen bereitstellen.

Codesharing App und Web

Das vollmundige Versprechen von NativeScript ist es, aus einer Codebasis nicht nur die mobilen Anwendungen bereitstellen zu können, sondern auch eine Webapplikation. Dieses gilt vor allem für den Code, der nicht fest an den DOM gekoppelt ist, denn so gibt es keinen Grund, warum er nicht in beiden Kontexten funktionieren sollte.

Um von diesem Level der Code-Wiederverwendung profitieren zu können, steht mit `nativescript-schematics` *[6]* ein Schema bereit, das Entwickler beim Aufsetzen eines Projekts für alle drei Zielapplikationen optimal unterstützt. Zentrales Element dabei ist wiederum die Dateinamenskonvention, die ähnlich der für die Trennung von plattformspezifischem Code aufgebaut ist:

```
import * as platform from "@nativescript/core/platform";
if (platformModule.isAndroid) {
    base64String = android.util.Base64.encodeToString(data,android.util.Base64.NO_WRAP);
} else {
    base64String = data.base64EncodedStringWithOptions(0);
}
```

Listing 2: Einfache Verwendung plattformspezifischer APIs

- **name.component.ts** - web + NativeScript shared
- **name.component.html** - web UI
- **name.component.tns.html** - NativeScript UI
- **name.component.css** - web stylesheet
- **name.component.tns.css** - NativeScript stylesheet

Möchte man nun noch separaten Code für Android und iOS einbauen, kann man die .ts-Datei noch aufsplitten:

- **name.component.ts**
- **name.component.android.ts**
- **name.component.ios.ts**

Einen umfassenden Einblick und guten Einstieg bietet die Dokumentation online *[7]*.

Entwicklungsumgebung

Es stehen offiziell unterstützte Plug-ins für VSCode und IntelliJ IDEA/Webstorm zur Verfügung, mit denen die Entwicklung schon sehr gut unterstützt wird. Außerdem wird die Entwicklung durch standardmäßig aktives „Hot Reload" unterstützt, bei dem nach dem Start der App veränderte Dateien direkt auf dem Gerät (oder Emulator) neugeladen und getestet werden können.

Publishing

Apps werden natürlich dafür entwickelt, dass sie veröffentlicht und verwendet werden. Die Erfahrung aus dem Projektalltag hat gezeigt, dass gerade in diesem Prozess die Automatisierung möglichst vieler Schritte und deren Abbildung in einem CI-System (Jenkins, GitLab, TeamCity…) sehr hilfreich ist und sich schnell bezahlt macht. In diesem Zusammenhand ist *fastlane [8]* unbedingt zu empfehlen, da es viele Schritte vom Zertifikatshandling über die Signierung bis hin zum Upload in den App- oder Play Store sehr gut kapselt und handhabbar macht.

Fazit

NativeScript ist ein erwachsenes Framework, das durch die dahinterstehende Firma Telerik/Progress mit Unterstützung einer starken Community intensiv weiterentwickelt wird. Für Interaktion mit der Community gibt es einen belebten Slack-Channel *[9]* und auf Issues auf GitHub wird schnell reagiert.

Was man sich dennoch bewusst machen sollte: Der Technologie-Stack, auf den man sich einlässt, ist nicht zu unterschätzen *(siehe Abbildung 2)*. Im Zweifel sollte man bereit sein, tief in die jeweilige Technologie einzutauchen, um die gestellten Anforderungen in letzter Konsequenz erfüllen zu können.

Dennoch macht sich gerade durch die Wiederverwendung von Know-how, Tools und Komponenten aus der Webentwicklung die Arbeit mit NativeScript bezahlt. Sowohl für Entwickler, die mit einem attraktiven Werkzeug arbeiten, als auch für Anwender, die zeitgemäße Lösungen erhalten, und auch Teams, die für den Entscheid über einen zielführenden Technologie-Stack einen sehr guten Kandidaten bekommen, ist NativeScript empfehlenswert.

Quellen

[1] *https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/*

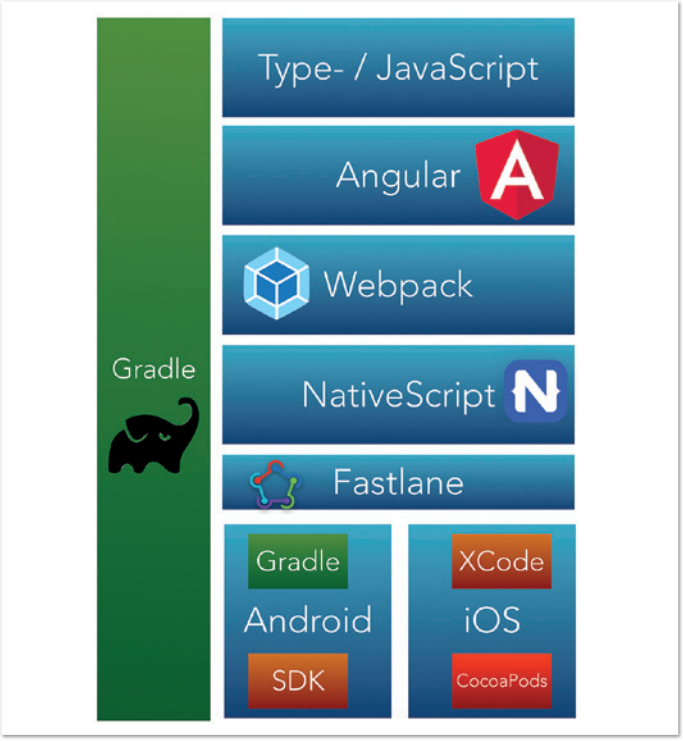


Abbildung 2: Technologie-Stack

[2] *https://svelte-native.technology/*
[3] *https://docs.nativescript.org/ui/styling*
[4] *https://play.nativescript.org/*
[5] *https://github.com/NativeScript/nativescript-dev-appium*
[6] *https://github.com/nativescript/nativescript-schematics*
[7] *https://docs.nativescript.org/angular/code-sharing/intro*
[8] *https://fastlane.tools/*
[9] *https://www.nativescript.org/slack-invitation-form*

Markus Schlichting
Karakun AG
markus.schlichting@karakun.com

Markus Schlichting ist Senior Software Engineer und Architekt bei Karakun AG. Software Engineering und -Architekturen, agile Methoden und Open-Source-Projekte zählen zu seinen Leidenschaften. Er liebt es, in den Bildschirmpausen Zeit mit seiner Familie zu verbringen oder beim Motorradfahren Frischluft zu tanken.

Wie spät ist es eigentlich? Ein Praxisblick auf die Team-Uhr

Stephan Doerfel und Simon Krackrügge, Micromata GmbH

Jeder, der in einem Team arbeitet, hat vermutlich schon einmal von Teamentwicklung, der Team-Uhr oder Ähnlichem gehört. Die Teamentwicklung erleben wir mal mehr, mal weniger bewusst. Ihr Ziel ist, dass aus einer Gruppe von Menschen ein performantes Team entsteht. Aber wie geht das eigentlich? In diesem Artikel reflektieren wir unsere eigenen praktischen Erfahrungen in einem Softwareentwicklungsteam. Dabei wird deutlich: Es gibt nicht den einen Uhrmacher, der die Uhr stellt. Vielmehr wird gemeinsam an der (Team-)Uhr gedreht; jedes Teammitglied trägt dazu bei.

In diesem Artikel teilen und diskutieren wir (reale) Begebenheiten, die wir in unserem Team erlebt haben. Unser Artikel ist also keine theoretische Abhandlung über Methoden und Werkzeuge der Teamentwicklung. Vielmehr ist das Anschauungsobjekt unser Team. Wir entwickeln und betreiben ein Softwaresystem im Kundenauftrag und sind crossfunktional aufgestellt: Zum Team gehören Backend- und Frontend-Spezialisten, ein Product Owner, ein Informationsarchitekt, ein Scrum Master und ein halber UX-Designer. In unserer Firma herrscht weitgehend Team-Autonomie und eine Kultur der gegenseitigen Wertschätzung. Gute Voraussetzungen also für erfolgreiche Teamarbeit – doch dafür bedarf es noch weiterer Zutaten!

Etwas Theorie

Wir stellen zwei Modelle für Teamarbeit vor, die uns geeignet erscheinen, die Dynamik in Teams zu verstehen.

Die Team-Uhr

Dieses Modell geht auf Beobachtungen des Psychologen Bruce Tuckman *[1]* zurück, der bemerkte, dass in vielen Beschreibungen von Teamentwicklungen vier Phasen (hier stark verknappt dargestellt) immer wieder auftreten:

- *Forming:* Das Team kommt zusammen, man sucht Orientierung bezüglich Zielen, Führung und Teamstruktur.
- *Storming:* Widerstand und Konflikte treten im Team auf.
- *Norming:* Das Team schafft neue Umgangsformen; Prozesse, Strukturen, Rollen und Befugnisse werden geklärt.
- *Performing*: Das Team arbeitet performant, jeder hat seine Rolle(n) gefunden.

Eine fünfte Phase, Adjourning, wurde in einer späteren Überarbeitung des Modells hinzugefügt und beschreibt die Auflösung eines Teams am Ende der gemeinsamen Arbeit *[2]*. Die Phasen laufen in der genannten Reihenfolge ab und gehen ineinander über. Jedoch ist die Entwicklung nicht mit dem Beginn der Performing-Phase abgeschlossen. Ereignisse wie Wechsel im Team, Veränderung der Aufgabe oder des Umfeldes – kurz, neue Situationen – starten die Phasen neu. Daher das Modell einer Uhr, auf der der Zeiger am Ende einer Runde schon die nächste beginnt. Tröstlich für alle, die schon stürmische Phasen im Team erlebt haben: Storming ist normal! Je-

der erlebt das. Und die Phase kommt immer wieder. Mit etwas Glück und Geschick wird sie jedoch mit der Zeit kürzer und lässt sich leichter in ein Norming überführen.

Fünf Dysfunktionen:

Das zweite Modell, das wir kurz anreißen, stammt von Patrick M. Lencioni [3] und beruht auf der Grundlage von Erfahrungen in Managerteams (in seinem Buch sehr spannend und deutlich genauer beschrieben). Das Modell nennt fünf typische Probleme (Dysfunktionen) von Teamarbeit, bei denen jeweils eines zum anderen führt:

- Das Grundproblem ist *fehlendes Vertrauen* (versteckte Agenden, unbekannte Mitarbeiter, Angst vor Schaden der eigenen Position).
- Dies führt zu *Scheu vor Konflikten* (keine Offenheit im Team, Gefühle nicht verletzen, keine Rache provozieren).
- Daraus folgt *fehlendes Engagement* (ohne Konfliktbereitschaft keine leidenschaftliche Debatte, Ziele und Absprachen werden nicht gemeinsam erkämpft oder wenigstens mitgestaltet, wichtige Punkte werden nicht geklärt).
- Ohne Engagement für die Ziele des Teams kommt es zu *Scheu vor Verantwortung* (man hält sich raus, übernimmt selbst keine Verantwortung und fordert auch keine Verantwortung ein).
- Die Kette endet schließlich bei *fehlender Ergebnisorientierung* (wo niemand für die Teamziele verantwortlich ist, folgt jeder seinen eigenen Zielen).

Ergebnisorientierung ist jedoch unerlässlich, wenn mehrere Menschen das gleiche Ziel erreichen sollen/wollen. Um diese zu erreichen, muss vorn in der Kette begonnen werden, die Weichen richtig zu stellen. Das bedeutet: Die Grundlage für erfolgreiches Teamwork ist Vertrauen im Team.

Im Rest des Artikels betrachten wir verschiedene Aspekte in der Teamentwicklung, exemplifizieren diese mit realen Erfahrungen aus unserem Team, ordnen sie Phasen der Team-Uhr zu und beschreiben unsere Sicht auf die Ereignisse.

Schlüsselmomente – Forming

Das Forming, die Findungsphase, setzt den Grundstein der Teamentwicklung. Sie bietet die Gelegenheit, Vertrauen zwischen den Teammitgliedern aufzubauen. Als wir mit dem gemeinsamen Projekt gestartet sind, waren einige Teammitglieder noch in alten Projekten eingebunden oder saßen entsprechend ihrer Spezialisierung in Büros mit Kollegen der gleichen Disziplin. Das Team begann also, über mehrere Räume und Etagen verteilt zu arbeiten. So entwickelten sich in dieser Zeit eigentlich zwei Teams, die parallel an Themen arbeiteten und sich nur zum Daily Scrum trafen. Die einzelnen Sub-Teams waren für sich erfolgreich, jedoch ging die gemeinsame Richtung verloren. Wir zogen daraus Konsequenzen und unser Umzug in ein gemeinsames Büro bekam oberste Priorität.

„Zusammenziehen" ist ein Schlüsselmoment, egal ob man in ein Teambüro zieht oder in eine Studenten-WG. Wer sitzt wo, wie erfolgt die Aufteilung – muss man losen oder einigt man sich? Wie wird ein neuer Mitbewohner willkommen geheißen? Gibt es das große Essen am Abend mit allen oder sitzt man mit Tütennudeln allein da, beziehungsweise allein am Einrichten der Entwicklungsumgebung? Räumliche Nähe hilft beim Kennenlernen – ja erzwingt es sogar. Beziehungen über die Arbeit hinaus entstehen in gemeinsamen Pausen, bei „Über-den-Tisch"-Gesprächen, bei gegenseitiger Hilfe mit kurzen Wegen – kurz, im gemeinsamen Alltag. Der gemeinsame Raum kann auch virtuell sein, dann sind die Herausforderungen beim Gestalten der Schlüsselmomente aber andere.

Wir erinnern uns: Die erste Phase der Team-Uhr ist durch die Suche nach Orientierung gekennzeichnet, also auch durch Unsicherheit und Verwirrung. Es geht zunächst darum, dass die Teammitglieder sich miteinander bekannt machen und ihre Zugehörigkeit zur Gruppe absichern. Gerade hier entscheiden Schlüsselmomente darüber, ob wir Vertrauen zu den Kollegen fassen, was unsere ersten Eindrücke über sie sind und welche Schlüsse wir über den zukünftigen Umgang miteinander ziehen.

Wir haben es selbst in der Hand, ob wir mit „First Date"-Stimmung ins Kennenlernen gehen oder ob wir gleich mit den Schwiegereltern starten wollen. Wie beim Beispiel mit der WG kommt es dabei, eine gemeinsame Basis zu schaffen, auf alle an. Gelungene Schlüsselmomente erleichtern den Einstieg in das Team und sie fördern Vertrauen. Weitere wichtige Momente sind zum Beispiel der erste Fehler und die ersten Meinungsverschiedenheiten (siehe Dysfunktion 2: Scheu vor Konflikten). In diesen Momenten wird der Weg für die Zukunft gesetzt. Wollen wir gemeinsam neue Wege erkunden oder nur keine Fehler machen? Suchen wir offen nach Lösungen, beharren wir auf unserer Meinung oder lassen wir ohne eigenes Engagement andere entscheiden?

Erwartungen – Storming

Ein Aspekt, der häufig zu persönlicher Unzufriedenheit und dann zu Konflikten führt, ist die enttäuschte Erwartung. Jedes Mitglied des Teams bringt seine eigenen Erwartungen, Vorstellungen und Vorhaben mit ins Projekt. Zur Laufzeit gleicht man dann (bewusst oder unbewusst) diese Erwartungen mit der Realität im Projekt ab und stellt gegebenenfalls Differenzen fest. Typische Themen, bezüglich derer viele Erwartungen im Vorhinein existieren, sind beispielsweise die Gruppenstruktur (wer hat welche Rolle? [5]), die Entwicklungsmethodik, Technologien, Code-Styles, Testing und so weiter. Auch profane Themen wie Pünktlichkeit oder generell das Verhalten miteinander können zu Konflikten führen.

In unserem Projekt gab es verschiedene Erwartungen zur Projekt-Methodik. Ganz klar war: Wir machen Scrum. Aber was erwartet jeder einzelne davon?

- Scrum ist eine Methode, der wir diszipliniert folgen, wie sie in der Literatur beschrieben ist.
- Scrum ist ein Methoden-Baukasten, aus dem wir uns bei Bedarf bedienen.
- Scrum heißt, dass wir uns alle zwei Wochen zum Planning treffen.
- Scrum heißt, dass wir Tasks zu Sprints zuordnen. Besonders eilige Tasks werden dem aktuellen Sprint zugeordnet.

Egal, welchem Statement man zustimmt, Enttäuschung ist vorprogrammiert, wenn die Realität eher einem der anderen Statements entspricht. Unklarheiten in der Projektmethode führen zu Unklarheiten über die zu erledigenden Aufgaben und schließlich zu Konflikten darüber, wer eigentlich wann was zu erledigen hat. In unserem

Team haben wir festgestellt, dass keiner unsere Umsetzung von Scrum als zielführend empfand. Um den Konflikt zu lösen, hatten wir zwei Möglichkeiten:

1. Wir einigen uns auf Scrum und passen unsere Arbeitsweise entsprechend an.
2. Wir suchen uns eine Methode, die besser zu unserer Arbeitsweise passt und den Projektgegebenheiten gerecht wird.

Lencioni *[3]* schreibt unter anderem, dass Teams, die nicht unter der dritten Dysfunktion leiden (also engagierte Teams), „Richtungswechsel ohne Zögern und Schuldgefühle" vornehmen. In diesem Sinne haben wir uns für Option zwei entschieden und gemeinsam mit unserem Kunden auf Kanban umgesattelt – eine Methode, die sehr gut zu unserem Projekt passt, da sie viel Flexibilität in der Task-Priorisierung erlaubt und vor allem auf die Minimierung von Bottlenecks im Workflow mit Tasks optimiert. Mit dem Wechsel von Scrum zu Kanban haben wir unsere tatsächliche Arbeit nur wenig angepasst (hauptsächlich die technischen Details der Umsetzung von Kanban). Das Verständnis davon, was möglich, gewünscht und gefordert ist, hat sich geändert und damit entstand eine stark verbesserte Zielorientierung.

Enttäuschte Erwartungen bergen Konfliktpotenzial und oft schwelen diese längere Zeit, bevor sie von den Beteiligten offengelegt werden. Damit diese frühzeitig bekannt werden, ist Vertrauen die Grundlage und räumliche Nähe eine gute Voraussetzung. Wenn im Halbscherz ein Satz wie „Na, Scrum hätte ich mir aber anders vorgestellt!" fällt, so kann man frühzeitig den Konflikt erkennen. Um möglichst von vornherein Enttäuschungen vorzubeugen, bietet es sich an, Erwartungen zeitig im Projekt gemeinsam offen zu besprechen und zu managen.

Wie spät ist es eigentlich auf der Team-Uhr? – Storming, Norming

Im Modell der Team-Uhr gehen Phasen ineinander über und die Uhr läuft jederzeit weiter. Wie erkennt also ein Team, wo es sich gerade befindet? Wir haben für unsere „erste Messung" einen Meilenstein im Projekt zu einem Project-Midway-Review genutzt. Der Fokus lag dabei weniger auf der entwickelten Software als auf dem Team selbst. Gemeinsam haben wir ein Teamprofile oder auch Team-Radar erstellt: Dabei werden Themen bestimmt, die auf einer Tafel, Brownpaper oder Ähnlichem auf je einer Achse wie Speichen eines Rades angeordnet werden. Wir haben Themen zum Projekt (zum Beispiel Zielklarheit), zum Team (zum Beispiel Wissensverteilung) und zur eigenen Person (zum Beispiel persönliche Entwicklung) gewählt. Jedes der Themen wird von jedem Teammitglied bewertet, indem ein Punkt (mit Stift oder Sticker) auf einer Stelle der Achse markiert wird. Die Achse dient dabei als Skala. Innen ist der schlechtmöglichste Wert, ganz außen der beste. Optional kann mit einem Klebezettel noch ein erklärendes Wort angebracht werden. Außerdem kann man seine Bewertung für die anderen erläutern.

Das entstandene Bild zeigte unsere Qualitäten, aber viel wichtiger auch unsere offenen Themen auf – Themen, bei denen die Bewertung niedrig war, oder Themen, bei denen die Bewertungen weit gestreut waren (Meinungsverschiedenheiten).

Um anhand des Team-Radars die richtige Uhrzeit ablesen zu können, bedarf es des Vertrauens, sich im Team offen äußern zu können, ohne selbst Schaden zu nehmen. Interessant ist, dass jeder bewerten darf, aber auch muss. So ist das Ergebnis tatsächlich ein Produkt des gesamten Teams.

Bei unserer ersten Anwendung der Methode haben wir uns viel Zeit und einen Moderator genommen und 14 Themen bearbeitet. Das Format hat für uns so gut funktioniert, dass wir es inzwischen einmal im Quartal in etwas kleinerem Umfang (acht Themen, kein Moderator) durchführen, um regelmäßig einen Blick auf die sich immer drehende Team-Uhr zu werfen (exemplarisch dargestellt in *Abbildung 1*).

Das bewusste Wahrnehmen der Team-Phase und der einzelnen Themen ist die Voraussetzung für die Steuerung des Teams. Themen mit Meinungsverschiedenheiten (Storming) können bewusst angegangen werden, um dafür gemeinsame Vereinbarungen (Norming) zu schaffen. Themen, die das gesamte Team positiv bewertet, kann man feiern!

Ziele – Norming, Performing

Aus unserem ersten Blick auf das Team-Radar ergab sich eine große Streuung beim Punkt Zielklarheit. Gerade als Product Owner benötigt man solche Informationen. Sie bedeuten, dass die Produktvision nicht beim gesamten Team komplett angekommen ist und nicht jeder sieht, wie die eigene Arbeit auf die Teamziele einzahlt. Wir haben daraufhin nach einer Methode gesucht, wie man gemeinsam Ziele entwirft und diese nachhält. Entschieden haben wir uns für Objectives and Key Results (OKRs), eine Methode, bei der man in regelmäßigem Turnus Ziele (Objectives) vereinbart, die mithilfe einiger weniger, messbarer Ergebnisse (Key Results) erreicht werden sollen *[4] (siehe Abbildung 2)*.

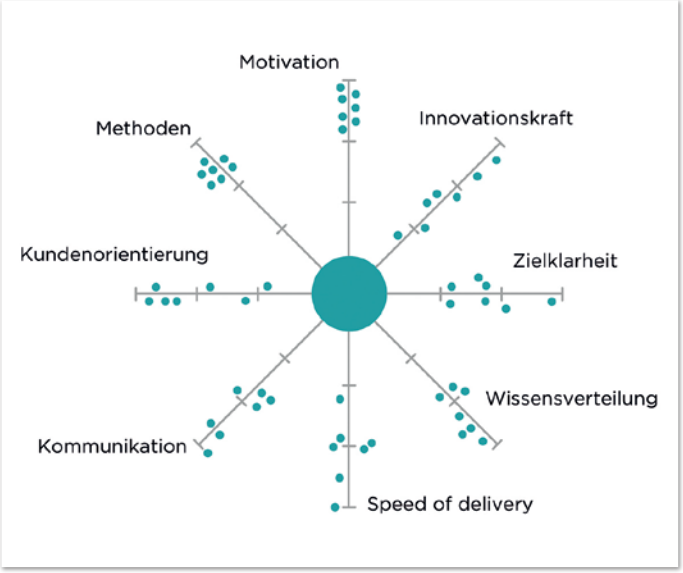


Abbildung 1: Ein Blick aufs Team-Radar. Speed of Delivery wurde niedrig und mit hoher Streuung bewertet. Bei Motivation herrscht eine einheitlich positive Sicht. (© Micromata GmbH)

Unsere OKRs hängen seitdem öffentlich im Teambüro und werden wöchentlich im Teammeeting bearbeitet. Es werden die aktuellen Top-Themen der Woche besprochen und geprüft, wie unser Fortschritt bei den Quartalszielen ist.

Unser erstes Team-Radar nach der Einführung von OKRs zeigte eine deutlich geringere Streuung auf der Achse „Zielklarheit" und eine positive Entwicklung. Wir haben entschieden, OKRs weiter zu nutzen und neben dem Team-Radar in unseren Alltagsgebrauch zu überführen.

Ein Schlüssel ist aus unserer Sicht das gemeinsame Erarbeiten der Ziele. Dies passt zu Lencioni *[3]*, denn nur, wer die Ziele mitgestaltet hat (durchaus in leidenschaftlicher Debatte), wird bereit sein, Verantwortung für diese zu übernehmen (Vermeidung von Dysfunktion 4).

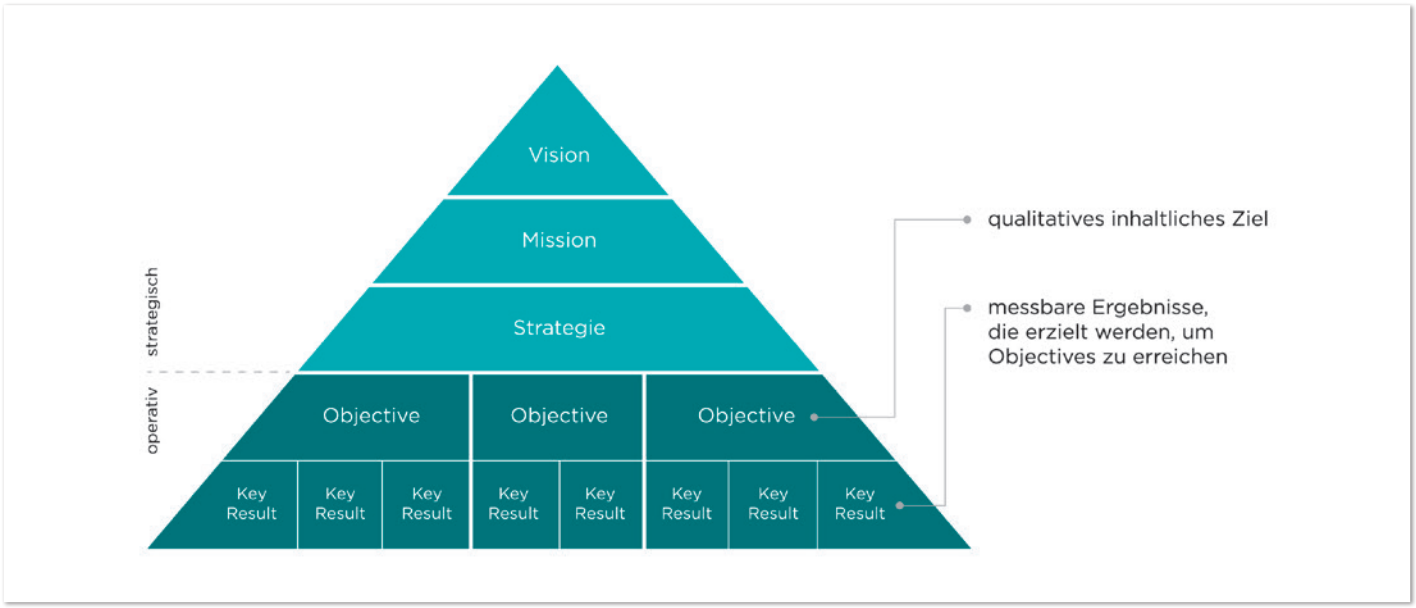


Abbildung 2: OKRs im Gesamtkontext des Projektleitbilds. (© Micromata GmbH)

Es sind die kleinen Dinge

Bisher haben wir in diesem Artikel die großen Themen betrachtet; Methoden, die ein Team in den ersten Phasen der Team-Uhr aufsetzen kann. Was aber braucht es noch, um ins Performing zu kommen und lange zu bleiben? Gerade die kleinen Dinge sind hier entscheidend! Um im Kontext des WG-Beispiels zu bleiben: Eine WG bleibt nicht stabil, weil der erste Abend schön war und die Schlüsselmomente sowie der Putzplan gestaltet wurden. Über einen langen Zeitraum funktioniert es gut, wenn der Alltag passt: viele kleine Dinge, die immer wieder stattfinden – der Einkauf, die Hilfe an der richtigen Stelle, das gemeinsame Abwaschen in der Küche. Die folgenden „kleinen Dinge" haben wir in der Projekt-Praxis als sehr wertvoll erlebt:

- Jeder hat eine Stimme! – Jeder darf und muss sich beteiligen; in der Retrospektive, im Team-Radar, bei den OKRs, etc.
- Räum den Müll weg! – Jeder im Team packt mit an und es gibt keine Arbeiten, um die man sich drückt. Jeder sollte die Spülmaschine ausräumen, wenn sie fertig ist, egal ob Geschäftsführer oder Praktikant
- Habt Spaß! – Gemeinsam lachen können ist ein Zeichen für Sicherheit, schafft Verbindung und Gemeinsamkeiten. Auch die NASA ist überzeugt, dass nur mit Clowns im Team die Reise zum Mars funktionieren wird [6].
- Widersteht dem Reflex, für andere eine Lösung zu suchen! – Kein Problem ohne Lösung; die Lösung sollte aber gemeinsam/selbst gefunden werden. Der Grad an Verantwortung, die ich für „meine" Lösung übernehmen werde, ist ungleich höher als der für eine fremde Lösung.
- Lasst das Team allein! – Läuft es auch ohne die Führungskraft, kann jeder im Team einfacher Verantwortung übernehmen und Vertrauen aufbauen. Nur als Eskalationsinstanz ist die Führungskraft gefragt.

Fazit

Es gibt keinen Uhrmacher, der allein die Team-Uhr auf „Performing" stellen kann – Teamentwicklung ist die Aufgabe des gesamten Teams. Dabei muss man sich immer wieder bewusst machen, wie es dem Team aktuell geht und in welcher Phase man sich vermutlich befindet. Die im Artikel beschriebenen Methoden und Werkzeuge sind keine Blaupause und kein Masterplan, sie haben uns aber geholfen und wir können empfehlen, sie im eigenen Team mal auszuprobieren. Die Erfahrungen, die wir gemacht haben, fassen wir zum Abschluss als Appell verkürzt zusammen:

- Gestaltet Schlüsselmomente gemeinsam!
- Erzeugt keine Erwartungen, die ihr nicht erfüllen könnt!
- Die Team-Uhr läuft immer weiter, nutzt die Chance, die „aktuelle Zeit" abzulesen!
- Übernehmt gemeinsam Verantwortung für eure Ziele!
- Achtet dauerhaft auf die kleinen Dinge!

Quellen

[1] Bruce W. Tuckman (1965): *Developmental sequence in small groups.* Psychological Bulletin (APA Publishing).

[2] Bruce W. Tuckman, Marry Ann C. Jensen (1977): *Stages of small-group development revisited.* Groups & Organization Studies, Volume 2, Issue 4 (Sage Journals).

[3] Patrick M. Lencioni (2014): *Die 5 Dysfunktionen eines Teams.* Wiley-VCH.

[4] John Doerr (2018): *Measure What Matters: OKRs: The Simple Idea that Drives 10x Growth.* Portfolio Penguin.

[5] J. Johnson, J. Boster, and L. Palinkas (2003): *Social roles and the evolution of networks in extreme and isolated environments.* Journal of Mathematical Sociology, Volume 27, Issue 2-3 (Routledge Taylor & Francis Group).

[6] J. Johnson (2019): *Human Exploration Crews: Informal Social Roles, Team Viability and Conflict.* AAAS Annual Meeting 2019, Washington DC.

Stephan Doerfel

Micromata GmbH

s.doerfel@micromata.de

Stephan Doerfel studierte zunächst an der TU Dresden Mathematik, wechselte dann an die Universität Kassel und wurde 2017 am Fachgebiet für Knowledge and Data Engineering promoviert. Beim Softwarehaus Micromata arbeitet er als Data Scientist und Softwareentwickler für verschiedene Projekte in unterschiedlichen Domänen. In seiner Funktion als Koordinator der Tech-Gilde berät er Micromata in den Bereichen Technisches Know-how und Technologische Innovation.

Simon Krackrügge

Micromata GmbH

s.krackruegge@micromata.de

Simon Krackrügge ist Product Owner/Requirements Engineer bei der Micromata GmbH. Seit 2010 begleitet er in verschiedenen Positionen die Entwicklung digitaler Produkte. Über das klassische Projektmanagement ging der Weg in die agile Entwicklung. Gemeinsam mit unterschiedlichen Teams entstanden in den letzten Jahren Produkte im Bereich Automotive, Logistik und Healthcare.

Ein Hologramm für alle (Magie?)

Chris Papenfuß, data experts GmbH

Dieser Artikel soll die Leser ermutigen, sich mehr mit der Thematik Shared Augmented Reality (AR) zu beschäftigen. Er zeigt, warum Hologramme großartig sind, geteilte jedoch noch viel besser. Zusätzlich möchte ich zeigen, warum aus meiner Sicht die Gaming-App „Minecraft Earth" technisch deutlich interessanter ist als „Pokémon GO". Hinweis: Im Text ist sämtlicher Quellcode in C# geschrieben.

Folgt man der Beschreibung für Hologramme auf Wikipedia *[1]*, ist die Darstellung von 3D-Modellen in AR-Apps wohl nicht als Hologramm zu bezeichnen. Trotzdem werden in diesem Artikel all diese 3D-Modelle so bezeichnet.

AR-Anwendungen

Die Definitionen von „Augmented Reality" gehen sehr weit auseinander. Liest man beispielsweise im Cambridge Dictionary *[2]*, so geht es „nur" um das gemeinsame Anzeigen von computergenerierten Bildern mit einer Ansicht der realen Welt. Bei Wikipedia gehen

die Autoren bereits etwas weiter *[3]*. So sind die erzeugten Bilder immer im Zusammenhang beziehungsweise als Erweiterung mit den Bildern der realen Welt zu verstehen.

Für meiner Meinung nach gute AR-Anwendungen sollten diese erzeugten Bilder nicht einfach nur eine Erweiterung der Realität, sondern auch in derselben „verankert" sein. Eine einfache und klassische Variante für solch eine Verankerung ist die Verwendung von Markern.

Damit das in *Abbildung 1* gezeigte Hologramm immer relativ zu dem QR-Code dargestellt werden kann, muss die Anwendung in jedem Frame aus dem Videostream der realen Welt wissen, wo exakt sich der Code befindet und wie er ausgerichtet ist. Dieser Bereich *[4]* der Computer Vision ist sehr spannend, kann hier aber nicht weiter ausgeführt werden. Dazu sei erwähnt, dass diese Art der Verankerung nicht nur mit QR-Codes einwandfrei funktioniert, sondern auch mit (fast) beliebigen anderen Bildern, wie zum Beispiel Fotos.

Damit man Hologramme an einer „beliebigen" Position in der realen Welt anzeigen kann, muss das Gerät mit der AR-App zu jedem Zeitpunkt möglichst genau wissen, wo es sich in der realen Welt befindet. Damit ist erst einmal nicht der genaue Längen- und Breitengrad gemeint, sondern eher, wie es sich relativ zu der Position bewegt hat, an der die Anwendung gestartet wurde. Dafür wird jedes Bild aus dem Video-Stream der Kamera analysiert und Features identifiziert *[5]*. Features aus dem vorherigen Frame werden wiedererkannt und über Triangulation wird die Änderung der Position im Raum bestimmt. Dieses Verfahren wird als „SLAM" (Simultaneous Localization and Mapping) bezeichnet und ist in der Praxis deutlich komplexer. In einer Implementierung sollte man zusätzlich die Daten von anderen Sensoren wie IMUs (inertiale Messeinheiten) *[6]* verwenden. Microsoft hat für seine HoloLens sogar extra einen Chip entwickelt [6a], der die Bewegung aufgrund der gesammelten Daten voraussagt und so eine Prognose für die Bewegung „schätzt", bevor diese dann in einer Anwendung verwendet werden kann.

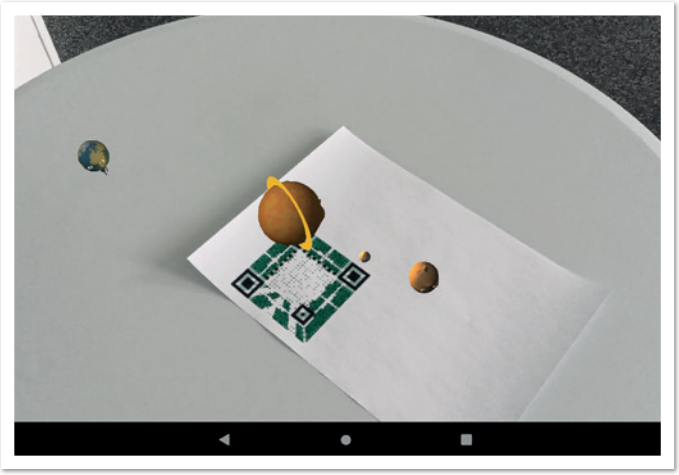


Abbildung 1: Hologramm vom Sonnensystem, verankert an QR-Code

Seit 2017 bietet Google die eigene Plattform ARCore *[7]*, die es ermöglicht, Hologramme mit entsprechendem Android-Smartphone beliebig im Raum zu platzieren. Auch Apple hat mit ARKit seit 2017 solch eine Plattform mit ähnlichem Feature-Set für entsprechende iPhones und iPads im Angebot *[8]*.

Mithilfe dieser Plattformen und der damit möglichen Verankerung verhält sich das Hologramm nun schon fast so, wie es sich für einen Anwender „natürlich" anfühlt. Das Hologramm wird an der gewünschten Position in der Realität dargestellt, ohne einen QR-Code oder anderen Bild-Marker zu verwenden *(siehe Abbildung 2)*. Das Hologramm verändert die Größe und Drehung, wenn sich der Anwender um den Anker herumbewegt *(siehe Abbildung 3 und 4)*.

Zum besseren Verständnis: Die „Verankerung" nennt sich bei Apples ARKit „ARAnchor" *[9]*, bei Googles ARCore heißt sie „Anchor" *[10]* und bei Microsofts HoloLens „World Anchor" *[11]*. In diesem Artikel wird der Begriff „Anchor" verwendet.

Abbildung 2: Hologramm einer Libelle, verankert auf einem Nachttisch

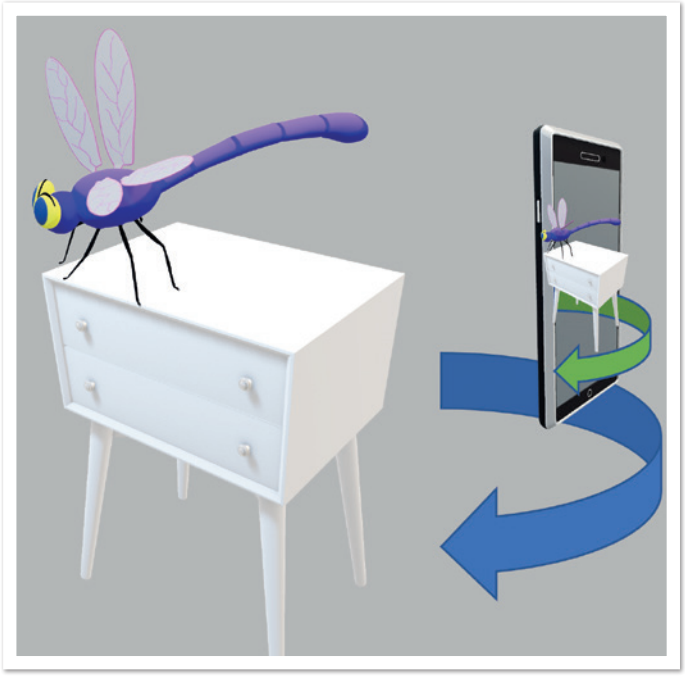


Abbildung 3: Verankertes Hologramm einer Libelle, verhalten bei Bewegung

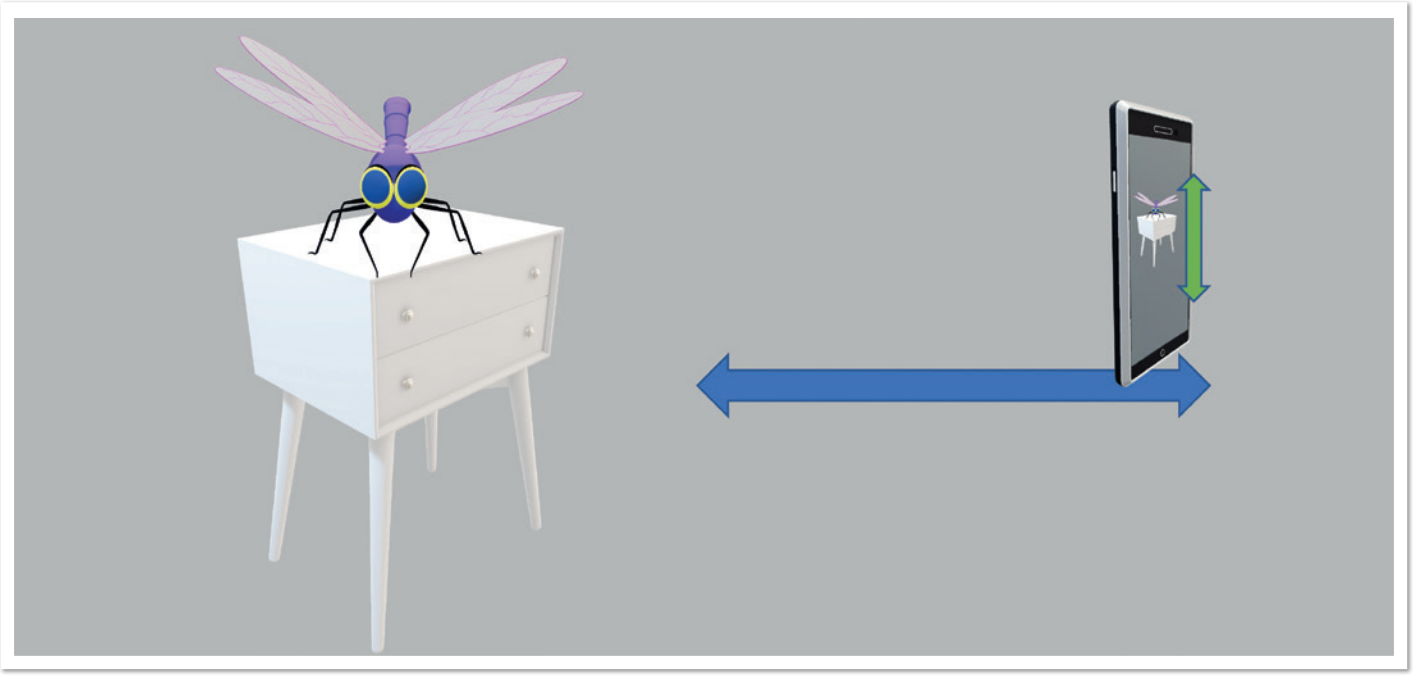


Abbildung 4: Verankertes Hologramm einer Libelle, verhalten bei Bewegung

„Multiplayer“

Damit die Interaktion mit den Hologrammen noch mehr „Spaß“ macht, sind neben der Verankerung noch ein bis zwei weitere Eigenschaften wünschenswert.

Zum einen möchte man seine Hologramme „teilen“. Dazu können entweder alle Beteiligten auf einen Smartphone-Bildschirm schauen oder noch besser: Jeder hat ein eigenes Gerät. Damit mehrere Geräte „dasselbe“ Hologramm sehen können, muss die entsprechende Verankerung gespeichert, übertragen und auf dem anderen Gerät wiedergefunden werden. Die Möglichkeit, diese Verankerung zu speichern und erneut zu laden, bieten alle erwähnten Plattformen. Will man diese allerdings übertragen und wiederfinden, gibt es aktuell die in *Tabelle 1* gezeigten Möglichkeiten/Services.

Aktuell (Beginn 2020) sind alle aufgeführten Services kostenlos, wobei Azure Spatial Anchors vermutlich in Zukunft kostenpflichtig sein wird *[15]*. Mit diesen Diensten kann man die Anchor von einem Gerät auf andere Geräte übertragen *(siehe Abbildung 5)*. Das ermöglicht mehreren Nutzern, das gleiche Hologramm an der gleichen Position in der Realität zu sehen *(siehe Abbildung 6)*.

Damit man nicht nur „statische“ Hologramme, also Statuen, sieht, benötigen die 3D-Modelle noch Interaktionsmöglichkeiten. Außerdem muss jede Interaktion möglichst synchron auf jedem Gerät ausgeführt werden. Das bedeutet, neben den geteilten Verankerungen benötigt man noch eine Verbindung zum Teilen von Interaktionen und Bewegungen. Auf der Suche nach der geeigneten IDE, um AR-Anwendungen zu entwickeln, landet man schnell bei Unity *[16]*. Diese IDE und 3D Engine hat ihre Wurzeln in der Entwicklung von Spielen. Somit ist es auch nicht weit hergeholt, das Teilen der Bewegung und Interaktionen als Multiplayer zu bezeichnen. Dementsprechend sind in diesem Bereich die Tools und Services zu finden, um unsere Anforderung zu erfüllen.

Aktuell verwende ich bei meinen Anwendungen für den Multiplayer-Anteil das SDK und den Service von Photon *[17]*. Die Integration in Unity ist gut gelungen und der Server kann je nach Anforderung

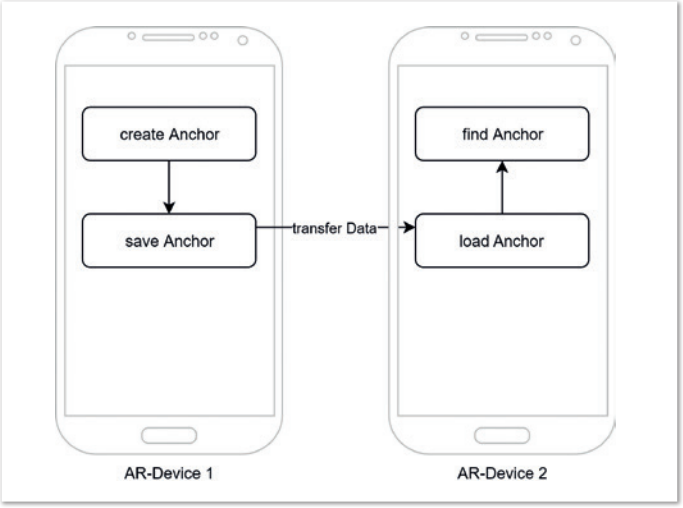


Abbildung 5: Ablauf Anchor-Sharing

	WorldAnchor [12]	Cloud Anchor [13]	Azure Spatial Anchor (ASA) [14]
iOS		x	x
Android		x	x
HoloLens	x		x

Tabelle 1

Abbildung 6: Anchor-Sharing in einer Anwendung mit der HoloLens

auch als Self-Hosted-Lösung verwendet werden. Die zusammengefasste Minimal-Architektur für eine Multiplayer-Anwendung ist in *Abbildung 7* dargestellt. Für die Verteilung des Anchors verwende ich Azure-Spatial-Anchors, da dort die meisten Plattformen bedient und zusätzliche Features, wie zum Beispiel Wayfinding, angeboten werden *[18]*.

„Simple" Anwendung

In diesem Kapitel erfahren Sie, wie alles zu einer funktionierenden Anwendung zusammengebaut wird.

Wie bereits erwähnt, setzte ich bei der Entwicklung auf Unity (Version 19.2.x) *[16]*. Nach der Erstellung eines neuen 3D-Projektes müssen das Azure-Spatial-Anchors SDK *[19]* und das Photon SDK *[20]* importiert werden. Ich verwende außerdem gerne das Open Source MixedRealityToolkit *[21]*, da viele Skripte, Designs und Tools für die Arbeit mit VR/AR-Anwendungen bereits enthalten sind. Um die Arbeit mit verschiedenen Zielplattformen (Android, iOS, HoloLens) weiter zu vereinfachen, ist es hilfreich, eine Abstraktionsebene auf AR-Core und AR-Kit aufzusetzen. Mit AR-Foundation *[22]* bietet Unity inzwischen genau solch eine Abstraktion an.

Zuerst sollte in der Anwendung die Verbindung zu Photon aufgebaut werden *(siehe Listing 1)*.

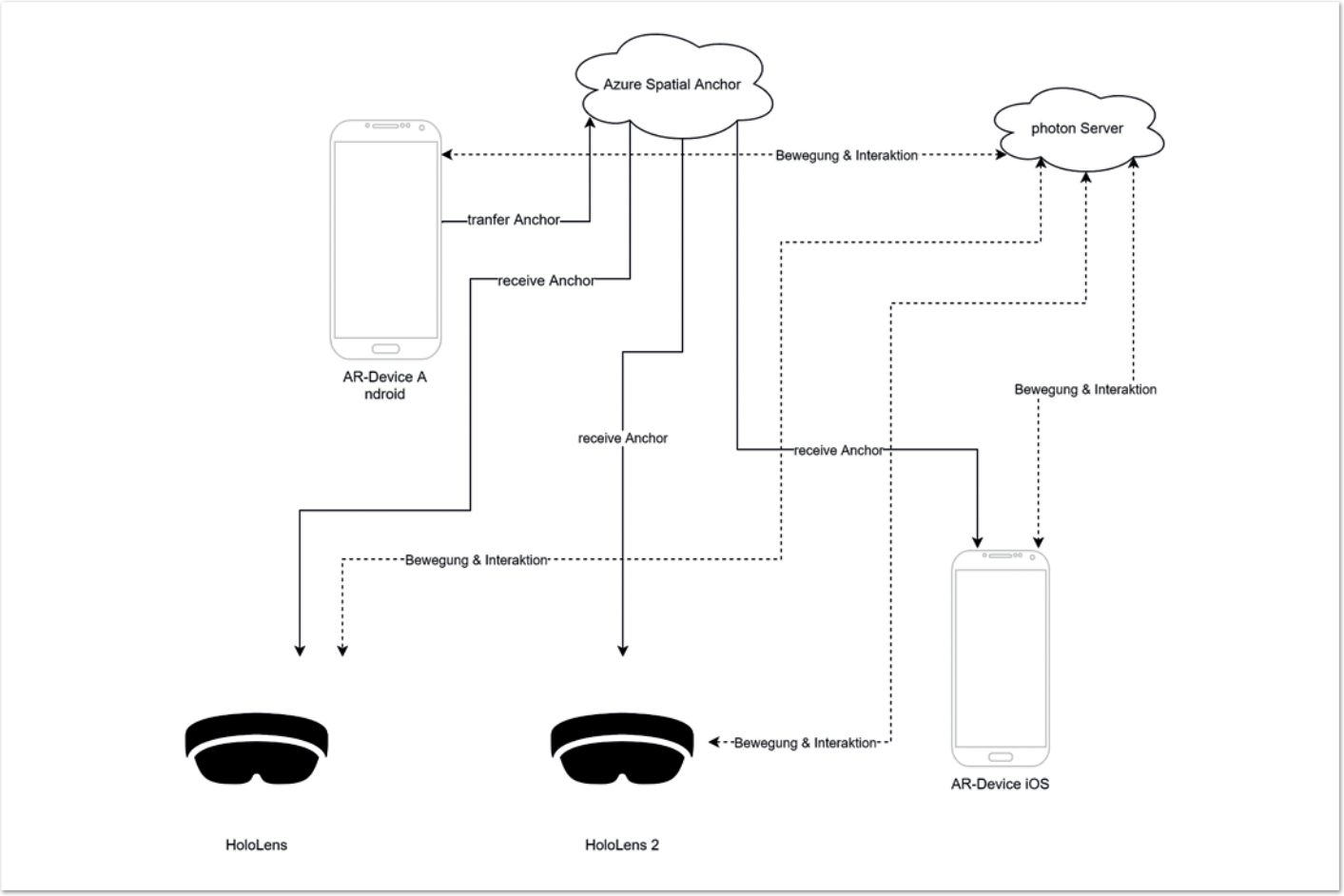


Abbildung 7: Minimalarchitektur „Multiplayer"-AR-Anwendung

```
// connect to photon server using settings from
// Assets\Photon\PhotonUnityNetworking\Resources\PhotonServerSettings.asset
PhotonNetwork.ConnectUsingSettings();
```

Listing 1: Verbindung zu Photon-Server

```
public override void OnConnectedToMaster()
{
    base.OnConnectedToMaster();
    var roomOptions = new RoomOptions
    {//60 seconds
        EmptyRoomTtl = 60000
    };
    PhotonNetwork.JoinOrCreateRoom("ROOM_NAME", roomOptions, null);
}
```

Listing 2: Verbindung zu einer Session „ROOM_NAME" auf einem Photon-Server

```
public SpatialAnchorManager ASAManager;
private async Task StartASAsync()
{
    try
    {
        //create session and wait for it
     //settings can be modified in file Assets\AzureSpatialAnchors.SDK\Resources\SpatialAnchorConfig.asset
        await ASAManager.CreateSessionAsync();
        //session creation was successfull now start session and wait for it
        await ASAManager.StartSessionAsync();
        //session is now successfully running
    }
    catch (Exception e)
    {
        Debug.LogError("Sessions SetUp failed");
        Debug.LogException(e);
    }
}
```

Listing 3: ASA (Azure Spatial Anchors)-Session initialisieren

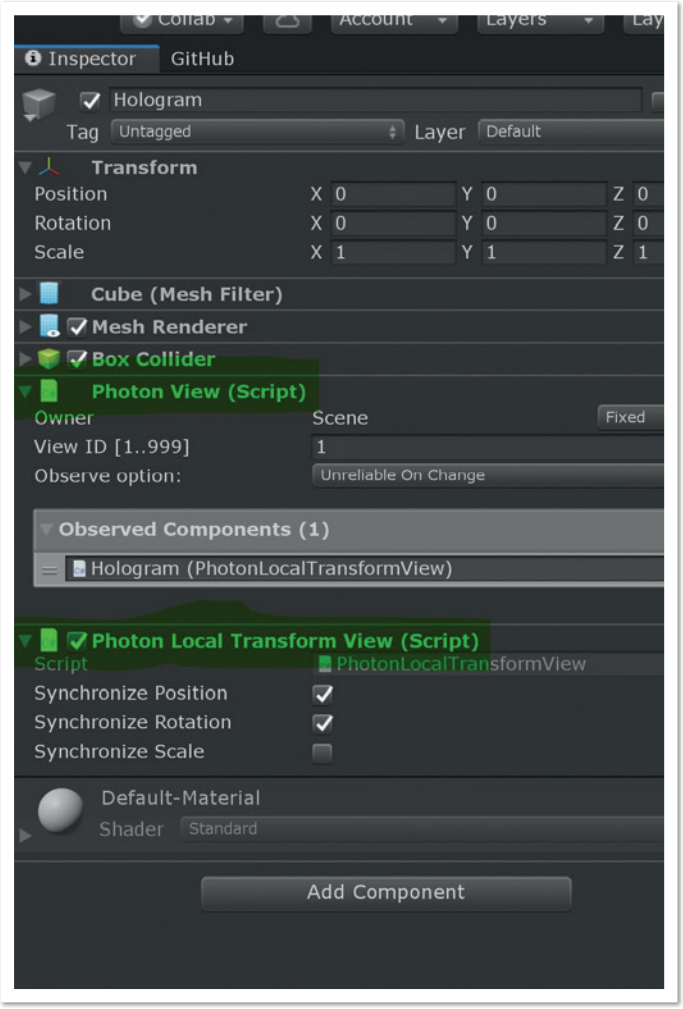


Abbildung 8: Unity-Editor: Gameobjekt mit Photon-Skripten

Da auf einem Server mehrere Sessions parallel laufen können, muss nach der Verbindung zum Server einem „Raum" beigetreten werden. In *Listing 2* wird gezeigt, wie man mit dem Callback der Photon-Klasse „MonoBehaviourPunCallbacks" einem definierten Raum beitritt.

Alle Teilnehmer eines Raumes können nun Daten, zum Beispiel Bewegung und Rotation von 3D-Objekten, miteinander teilen und synchronisieren. Das Photon-SDK bietet für verschiedene Aufgaben bereits vorbereitete Scripts. So gibt es PhotonView.cs *[23]* zur Markierung und Identifizierung eines Objektes, das Eigenschaften über das Netzwerk teilen soll. PhotonTransformView.cs *[24]* muss zusätzlich an Unity-Game-Objekte gehängt werden, um die Position und Rotation zu synchronisieren *(siehe Abbildung 8)*.

Über die Photon-Verbindung kann der Spatial-Anchor zwischen den Geräten geteilt werden. Bevor allerdings ein Anchor gefunden oder erstellt werden kann, muss eine Session aus dem Azure Spatial Anchors SDK initialisiert und gestartet werden *(siehe Listing 3)*.

Mit der gestarteten Session ist alles zum Erstellen, Speichern, Laden und Finden eines Anchor vorbereitet. In *Listing 4* werden genau diese beiden Optionen dargestellt. Dabei wird die ID des erstellten Anchor mit Photon an allen verbundenen Geräten übertragen.

Zu beachten ist noch, dass sämtlicher Quellcode hier ein Minimal-Beispiel darstellt. In einer „echten" Anwendung sollten an den nötigen Stellen Ausfall-Szenarien implementiert werden.

Bewegung von Hologrammen und Anchor

Anchor sind darauf ausgelegt, möglichst stabil an einem Ort in der Realität zu verharren, daher sollte in einer Anwendung davon ab-

```
async Task CreateAndSaveAnchor()
{
    // get or create cloud-native anchor
    var cna = GetComponent<CloudNativeAnchor>();
    if (!cna)
    {
        Debug.Log("Adding CloudNativeAnchor");
        cna = gameObject.AddComponent<CloudNativeAnchor>();
    }

    // If the cloud portion of the anchor hasn't been created yet, create it
    if (cna.CloudAnchor == null) { cna.NativeToCloud(); }

    // Get the cloud portion of the anchor
    var cloudAnchor = cna.CloudAnchor;

    // make anchor expire in 3 days
    cloudAnchor.Expiration = DateTimeOffset.Now.AddDays(3);

  //wait while device is capturing enough visual data to create anchor
    while (!ASAManager.IsReadyForCreate && ASAManager.SessionStatus.RecommendedForCreateProgress < 1F)
    {
        await Task.Delay(330);
        var createProgress = ASAManager.SessionStatus.RecommendedForCreateProgress;
        _debugText = $"Move your device to capture more environment data: {createProgress:0%}";
    }
    try
    {
        // save anchor to cloud
        await ASAManager.CreateAnchorAsync(cloudAnchor);

        // successfull?
        if (cloudAnchor != null)
        {
            _debugText = "Anchor saved";
            // save anchor-ID as property to photon room
            PhotonNetwork.CurrentRoom.SetCustomProperties(
                new Hashtable()
                {
                    { ANCHOR_ID_CUSTOM_PROPERTY, cloudAnchor.Identifier }
                }
            );
        }
        else
        {
            Debug.LogError(new Exception("Failed to save, but no exception was thrown."));
            _debugText = "Anchor saving failed";
        }
    }
    catch (Exception ex)
    {
        Debug.LogError(ex);
    }
}

async Task SearchAnchorAsync()
{
    // is there an anchor-ID in properties of photon room?
    if (PhotonNetwork.CurrentRoom.CustomProperties.TryGetValue(ANCHOR_ID_CUSTOM_PROPERTY, out var anchorID))
    {
        _debugText = "loading Anchor";
        Debug.Log("Achnor loading");
     // tell the watcher which anchor to look for
        var anchorLocateCriteria = new AnchorLocateCriteria
        {
            Identifiers = new[] { (string)anchorID }
        };
        _watcher = ASAManager.Session.CreateWatcher(anchorLocateCriteria);
        //don't do it this way in production code!
     //callback will be called, when anchor is visually found
        ASAManager.AnchorLocated += CloudManager_AnchorLocated;
    }
}
private void CloudManager_AnchorLocated(object sender, AnchorLocatedEventArgs args)
{
    if (args.Status == LocateAnchorStatus.Located)
    {
        var anchor = args.Anchor;
        UnityDispatcher.InvokeOnAppThread(() =>
        {//operation need to run in app thread
            var cna = GetComponent<CloudNativeAnchor>();
            if (!cna)
            {
                Debug.Log("Adding CloudNativeAnchor");
                cna = gameObject.AddComponent<CloudNativeAnchor>();
            }
        //adding found cloud anchor to cloud native anchor so the object will place itself at real world location
            cna.CloudToNative(anchor);
            _debugText = "Anchor Located";
        });
    }
    else if (args.Status == LocateAnchorStatus.NotLocatedAnchorDoesNotExist || args.Status == LocateAnchorStatus.NotLocated)
    {
        _debugText = "Anchor not Located";
    }
}
```

Listing 4: ASA Anchor Erstellungs- und Lokalisierungsprozess

gesehen werden, diese ständig zu bewegen. Da wir aber bewegte Hologramme wollen, bietet es sich an, die Anchor immer als eine Art „Basis" zu behandeln. Diese Basis wird in der Realität möglichst nah an realen Objekten verankert. Die Hologramme, die man bewegen möchte, werden dann relativ zu solch einem Anchor bewegt. Das bedeutet natürlich, dass man hier entweder wieder sein Mathebuch zum Thema Vektor- und Matrizenrechnung hervorholt oder sich auf die entsprechenden Methoden in der Unity-Engine verlässt. Meiner Meinung nach ist es durchaus hilfreich, wenn man versteht, was passiert. Dieser Anchor muss keine grafische Repräsentation in einer Anwendung besitzen. Außerdem sollte man drauf achten, dass die Hologramme sich nicht zu weit von einem Anchor wegbewegen, da bereits kleine Fehler in der Positionierung sich über größere Entfernung immer mehr bemerkbar machen. Daher sollte man bei größeren Entfernungen einen neuen Anchor erstellen und das Hologramm nun relativ zu diesem bewegen oder aber eine entsprechende Boundary in der Anwendung implementieren.

Pokémon Go versus Minecraft Earth

Eingangs hatte ich erwähnt, dass ich Minecraft Earth technisch interessanter im Vergleich zu Pokémon Go finde. In Minecraft Earth können nämlich mehrere Personen gemeinsam an ihren Bauwerken bauen und somit ihre Hologramme teilen. Auch wenn Pokémon Go einen großen Beitrag dazu geleistet hat, AR Mainstream zu machen, so waren zu Beginn die Pokémon-Hologramme nicht einmal in der Realität verankert.

Dieser Artikel erscheint begleitend zu meinem Vortrag in der JavaLand im März 2020 *[25]*.

Quellen

[1] *https://de.wikipedia.org/wiki/Holografie*
[2] *https://dictionary.cambridge.org/dictionary/english/augmented-reality*
[3] *https://en.wikipedia.org/wiki/Augmented_reality* (Englisch)
https://de.wikipedia.org/wiki/Erweiterte_Realit%C3%A4t (Deutsch)
[4] *https://docs.opencv.org/master/d9/d97/tutorial_table_of_content_features2d.html*
[5] *https://docs.opencv.org/master/d7/d66/tutorial_feature_detection.html*
[6] *https://de.wikipedia.org/wiki/Inertiale_Messeinheit*
[6a] *https://www.pcwelt.de/news/Hololens-Microsoft-lueftet-HPU-Geheimnis-10028103.html*
[7] *https://android-developers.googleblog.com/2017/08/arcore-augmented-reality-at-android.html*
[8] *https://9to5mac.com/2017/06/05/apple-announces-arkit-for-ios-11/*
[9] *https://developer.apple.com/documentation/arkit/aranchor*
[10] *https://developers.google.com/ar/develop/developer-guides/anchors*
[11] *https://docs.unity3d.com/ScriptReference/XR.WSA.WorldAnchor.html*
[12] *https://docs.microsoft.com/en-us/windows/mixed-reality/persistence-in-unity*
[13] *https://developers.google.com/ar/develop/java/cloud-anchors/overview-android*
[14] *https://docs.microsoft.com/en-gb/azure/spatial-anchors/overview*
[15] *https://azure.microsoft.com/en-us/pricing/details/spatial-anchors/*
[16] *https://unity.com/*
[17] *https://www.photonengine.com/*
[18] *https://docs.microsoft.com/en-us/azure/spatial-anchors/concepts/anchor-relationships-way-finding*
[19] *https://github.com/Azure/azure-spatial-anchors-samples/releases*
[20] *https://www.photonengine.com/en-US/sdks#realtime-unity-sdkrealtimeunity*
[21] *https://github.com/Microsoft/MixedRealityToolkit-Unity*
[22] *https://unity.com/unity/features/arfoundation*
[23] *https://doc-api.photonengine.com/en/pun/v2/class_photon_1_1_pun_1_1_photon_view.html*
[24] *https://doc-api.photonengine.com/en/pun/v2/class_photon_1_1_pun_1_1_photon_transform_view.html*
[25] *https://programm.javaland.eu/2020/#/scheduledEvent/590747*

Chris Papenfuß
data experts GmbH
Chris.Papenfuss@data-experts.de

Chris Papenfuß leitet seit 2017 das Team Holographic bei der Firma data experts GmbH. In den letzten drei Jahren hat sein Team diverse AR/VR-Projekte für Kunden wie zum Beispiel Rolls-Royce Deutschland, Deutsche Bahn und Homag realisieren und regelmäßig erfolgreich an Hackathons teilnehmen können. data experts gehört zu den ersten Microsoft-Mixed-Reality-Partnern.

Funktionale Programmierung in Kotlin

Stefan López Romero, MaibornWolff GmbH

Dieser Artikel gibt eine Einführung in die funktionale Programmierung mit Kotlin. Wir lernen die Grundbausteine und Mechanismen dieses Programmierparadigmas kennen. Unter anderem sehen wir, wie man Funktionen höherer Ordnung programmiert, diese kombiniert und „curryfiziert". Kotlin ist dabei unser Vehikel, um leichtgewichtig in die funktionale Programmierung einzusteigen. Die Sprache bietet im Vergleich zu Java echte Funktionstypen sowie viele syntaktische Verbesserungen und Vereinfachungen.

Was ist funktionale Programmierung?

Funktionale Programmierung ist ein deklaratives Programmierparadigma, das ausschließlich sogenannte pure Functions verwendet. Um zu verstehen, was eine *pure Function* ist, benötigen wir den Begriff der *referenziellen Transparenz*:

Ein Ausdruck ist referenziell transparent, wenn er durch seinen Rückgabewert ersetzt werden kann, ohne das Verhalten des Programms zu verändern [1].

Damit können wir nun formal definieren, was wir unter einer pure Function verstehen:

Eine Funktion f ist pur, wenn der Ausdruck f(x) für alle referenziell transparenten x referenziell transparent ist [1].

Diese etwas abstrakte Definition besagt, dass eine Funktion pur ist, wenn sowohl der Parameter x als auch die Funktionsauswertung f(x) durch ihr Ergebnis ersetzt werden können, ohne dass dies das Verhalten des Programms ändert. In *Listing 1* sehen wir ein Beispiel dafür.

Die Funktion `plus` ist pur, denn man kann `a.plus(2)` durch das Ergebnis `4` ersetzen, ohne dass dies etwas am weiteren Ablauf des Programmes ändert. In *Listing 2* sehen wir dagegen eine Funktion, die diese Eigenschaft nicht erfüllt.

Würden wir hier die zweite Zeile durch ihr Ergebnis „`Tic Tac`" ersetzten, wäre `c` nicht mehr „`Tic Tac Toe`", sondern „`Tic Toe`". Dies liegt daran, dass die `append`-Funktion einen Seiteneffekt hat. Neben der Rückgabe eines zusammengesetzten Strings wird zusätzlich der Zustand der Instanz verändert. Seiteneffekte sorgen dafür, dass Code schwerer zu verstehen und somit auch fehleranfälliger ist. Wir können nicht wie in *Listing 1* nur eine Zeile betrachten, um das Ergebnis einer Funktion zu kennen, sondern müssen alle Zustandsänderungen, die auf der Instanz passieren, im Auge behalten. Im Beispiel ist dies noch überschaubar. Liegen die besagten Zustandsänderungen jedoch weit auseinander oder sind sie auf verschiedene Funktionen verteilt, ist es oft nur noch schwer möglich, den Überblick zu behalten.

Hier punktet die funktionale Programmierung. Aufgrund der referenziellen Transparenz sind in pure Functions keine Seiteneffekte erlaubt. Funktionaler Code ist dadurch leichter zu verstehen, zu testen und zu debuggen. Zudem führt die Seiteneffektfreiheit zu aussagekräftigeren Funktionssignaturen: Eine Funktion nimmt einen oder mehrere Parameter entgegen, führt einen Algorithmus aus und gibt immer ein Ergebnis zurück. Daneben kann und darf nichts passieren. Aufgrund dieses Aspekts kann man funktionale Programmierung auch als Programmierung ohne Seiteneffekte bezeichnen.

```
val a = 2
val b = a.plus(2)  //=4
val c = a.plus(3)  //=5
```

Listing 1: Pure Function

```
val a = StringBuilder("Tic")
val b = a.append(" Tac").toString() //="Tic Tac"
val c = a.append(" Toe").toString() //="Tic Tac Toe"
```

Listing 2: Impure Function

Funktionen und Datentypen als Bausteine

Unsere Bausteine, um Programme zu erstellen, sind neben pure Functions (im weiteren Verlauf einfach als „Funktionen" bezeichnet) Datentypen. Ein Datentyp in der funktionalen Programmierung ist ein Name, der einer Menge möglicher Werte gegeben wird. Dieser Datentyp kann als Ein- und Ausgabe einer Funktion verwendet werden. Funktionen agieren dabei als Transformatoren zwischen den Datentypen. Sie wandeln Informationen von einem oder mehreren Eingabetypen in einen Ausgabetyp. Datentypen können in Kotlin am einfachsten mithilfe von Data Classes erstellt werden. In *Listing 3* sehen wir, wie eine Data Class `Name` definiert wird, die aus den beiden Strings `firstName` und `lastName` zusammengesetzt ist.

Funktionen als Werte

In funktionalen Sprachen sind Funktionen sogenannte *first-class citizens*. Das bedeutet, dass Funktionen (wie auch Werten) Variablen zugewiesen, als Argumente übergeben oder von einer anderen Funktion zurückgegeben werden können. Da Kotlin statisch typisiert ist, müssen auch Funktionen einen Typ haben. In *Listing 3* sehen wir ein Beispiel für die Deklaration eines Funktionstyps. Der Bezeichner `sayHello` ist vom Typ `(Name) -> String`. Dies bedeutet, dass sich hinter dem Namen `sayHello` eine Funktion verbirgt, die ein Argument des Datentyps `Name` entgegennimmt und einen `String` zurückgibt. Nach der Deklaration wird der Funktionstyp im Beispiel durch einen Lambda-Ausdruck instanziiert. Ein Lambda-Ausdruck ist eine Kurzschreibweise für eine anonyme Funktion *[2]* und immer von geschweiften Klammern umgeben. Innerhalb der Klammern folgen die Parameter-Deklarationen, die optional mit Typangaben versehen werden können. Der Body der Funktion steht nach dem Pfeil. Das Ergebnis des letzten Ausdrucks innerhalb des Lambda-Body ist zugleich der Rückgabewert. Eine so definierte Funktion nennen wir aufgrund ihrer Zuweisung zu einem Wert *value Function*.

Die Instanziierung durch einen Lambda-Ausdruck ist nur eine Möglichkeit der Funktionsdefinition. In *Listing 4* sehen wir, wie das auch durch eine anonyme Funktion sayHelloAno möglich ist.

```
data class Name(val firstName: String, val lastName: String)

val sayHello: (Name) -> String = {
    name -> "Hello, ${name.firstName}  ${name.lastName}"
}
```

Listing 3: Datentypen und Funktionen

Neben den bisher kennengelernten value Functions gibt es noch eine zweite Schreibweise für Funktionen in Kotlin. Wie wir anhand von `sayHelloFun` sehen, können Funktionen auch mit dem Schlüsselwort `fun` eingeleitet werden. Danach folgen der Bezeichner, die Argumente und am Ende der Rückgabewert. Der Body der Funktion steht nach dem Gleichheitszeichen. Diese Art von Funktionen bezeichnen wir als `fun` Functions. Hier stellt sich die Frage: Wieso gibt es überhaupt zwei Schreibweisen für Funktionen? `fun` Functions sind effizienter und können mit generischen Typen (Generics) umgehen. Deshalb sollte man für Funktionen, die nicht als Wert übergeben werden, diese Schreibweise verwenden. `fun` Functions können durch Funktionsreferenzen wie in `sayHelloRef` zu value Functions konvertiert werden.

Funktionskomposition

Funktionen und Datentypen sind also unsere Bausteine für funktionale Programmierung. Bleibt noch die Frage: Wie können wir diese Bausteine miteinander verbinden, sodass aus einzelnen kleinen, wiederverwendbaren Funktionen ein Programm wird, das eine bestimmte Aufgabe erfüllt? Durch das Fehlen von Seiteneffekten können Funktionen einzig und allein durch eine Verkettung von Funktionsaufrufen miteinander interagieren. In *Listing 5* sehen wir ein Beispiel dafür. Die Funktion `parseInt` liest einen String ein und wandelt diesen in einen Integer-Wert um. Die Funktion `multBy4` multipliziert einen gegebenen Integer-Wert mit vier. Wollen wir in der `main`-Funktion eine als String angegebene Zahl mit vier multiplizieren, so verketten wir beide Funktionen, sodass der Rückgabewert der einen Funktion als Eingabe für die nächste Funktion fungiert.

Wie bereits erwähnt können Funktionen auch andere Funktionen als deren Parameter oder Rückgabewert verwenden. Solche Funktionen werden Funktionen höherer Ordnung genannt. Machen wir uns diese Eigenschaft zunutze, so können wir die Verkettung von Funktionen abstrahieren und so wiederverwendbar machen. Der Mechanismus hierfür wird Funktionskomposition genannt und ist folgendermaßen definiert:

Seien A, B, C beliebige Mengen und g: A → B sowie f: B → C Funktionen, so heißt die Funktion ***f ° g: A → C , (f ° g)(x) = f(g(x))*** *Komposition von f und g.*

Auf unser Beispiel übertragen bedeutet dies: Wir können eine Funktion `compose` definieren, die zwei Funktionen `f` und `g` mit `f:(Int)-> Int` und `g: (String)-> Int` entgegennimmt und eine Funktion `(String)-> Int` zurückgibt. *Listing 6* zeigt die Implementierung dieser Funktion. Im Body der `compose`-Funktion sehen wir, wie die beiden gegebenen Funktionen per `f(g(x))` miteinander verkettet werden. Durch diese Abstraktion können wir jetzt ganz einfach eine neue Funktion `parseAndMult` durch Kombination der zwei vorhin definierten Funktionen erstellen *(siehe Listing 6)*.

Leider ist unsere `compose`-Funktion auf bestimmte Datentypen beschränkt. Aber auch dies können wir abstrahieren. In *Listing 7* sehen wir eine durch generische Typen noch allgemeinere Version dieser Funktion. Wie bereits erwähnt ist die Verwendung von Generics nur mit `fun` Functions möglich, deshalb wählen wir jetzt diese Schreibweise. Da wir keinerlei Angaben zu den generischen Typen `A`, `B` und `C` machen, sind sie durch jegliche Typen ersetzbar. Funktionen höherer Ordnung, die mit generischen Typen arbeiten, werden polymorphe Funktionen höherer Ordnung genannt. Am Aufruf der `compose`-Funktion ändert sich trotz der generischen Typen nichts.

```
val sayHelloAno: (Name) -> String = fun(name:Name) =
    "Hello ${name.firstName}  ${name.lastName}"

fun sayHelloFun(name: Name) : String =
    "Hello ${name.firstName}  ${name.lastName}"

val sayHelloRef: (Name) -> String = ::sayHelloFun
```

Listing 4: Weitere Möglichkeiten der Funktionsdefinition

```
val parseInt: (String) -> Int = {
    s -> s.toInt()
}
val multBy4: (Int) -> Int = {
        x -> x * 4
}
fun main() {
    val result = multBy4(parseInt("2"))
}
```

Listing 5: Funktionsverkettung

```
val compose: (f:(Int) -> Int, g:(String) -> Int) -> (String) -> Int = {
        f, g -> {
            x -> f(g(x))
    }
}
fun main() {
    val parseAndMult = compose(multBy4, parseInt)
    val result = parseAndMult("4")

}
```

Listing 6: Funktionskomposition

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C = {
        x -> f(g(x))
}
```

Listing 7: Polymorphe Funktion höherer Ordnung

```
fun <A, B, C> curry(f:(A, B)-> C) : (A) -> (B) -> C = {
    a -> {
        b -> f(a, b)
    }
}
```

Listing 8: Currying

```
fun calcTax(rate: Double, value: Double): Double {
    return value * rate
}

fun main() {
    val calcVat = curry(::calcTax)(0.19)
    val vat = calcVat(149.99)
}
```

Listing 9: Partielle Funktionsanwendung

Anhand der Signatur der zu komponierenden Funktionen kann Kotlins Typ-Inferenz den Funktionstyp `(String) -> Int` als Rückgabetyp selbstständig ermitteln.

Currying

In vielen funktionalen Sprachen kann eine Funktion nur ein Argument verarbeiten. Dies ist jedoch kein Nachteil, da man mithilfe eines Verfahrens namens Currying aus jeder Funktion mit mehreren Parametern eine gleichwertige Sequenz von Funktionen mit einem Parameter machen kann. Der Name dieses Verfahrens geht auf den Mathematiker Haskell Brooks Curry *[3]* zurück. Als Beispiel nehmen wir eine Funktion, die zwei Argumente annimmt – eines vom Typ A und eines vom Typ B – und ein Ergebnis vom Typ C erzeugt. Durch Currying wird diese in eine Funktion übersetzt, die ein einziges Argument vom Typ A annimmt und als Ergebnis eine Funktion von B nach C erzeugt. Wir konsumieren also immer nur ein Argument und geben Funktionen zurück, die die weiteren Argumente verarbeiten. Wie man dieses Verfahren in Kotlin-Code übersetzt, sehen wir in *Listing 8*. Die Funktion `curry` wandelt jede zweistellige Funktion in eine Funktion in Curry-Schreibweise um.

Diese Umformung einer Funktion bietet uns mehr Flexibilität. Es ist nun nicht mehr zwingend notwendig, dass wir beide Argumente der Funktion sofort angeben, sondern wir können erst mal ein Argument belegen und die Angabe des zweiten auf später verschieben. Dies können wir uns zum Beispiel in Anwendungsfällen zunutze machen, bei denen ein Parameter eine Konstante ist. In *Listing 9* sehen wir ein Beispiel dafür: Die Funktion `calcTax` berechnet den Steuerbetrag zu den in den Parametern angegebenen Steuersatz und Wert. Wir wollen diese Funktion zur Berechnung der Mehrwertsteuer verwenden, die in unserem Anwendungsfall konstant 19 Prozent ist. Mithilfe unserer `curry`-Funktion wandeln wir die `calcTax`-Funktion in die Curry-Schreibweise um. Hierdurch können wir den Steuersatz fest auf 0.19 setzen und erhalten eine neue Funktion, die nur noch den Wert als Parameter erwartet. Diesen Mechanismus, bei dem nicht alle Parameter einer Funktion sofort belegt werden, nennt man partielle Funktionsanwendung.

Zusammenfassung und Ausblick

Funktionen als *first-class citizens* bieten uns sehr viel Spielraum in der Software-Entwicklung. Wir können Abstraktionen für wiederkehrende Probleme erstellen, ohne hierfür komplexe Klassenhierarchien aufbauen zu müssen. Durch die Seiteneffektfreiheit wirkt eine Funktion nur lokal und kann keinen äußeren Zustand verändern, was das Testen und die Fehlersuche sehr vereinfacht. Jedoch ist jede Ein- oder Ausgabe, etwa das Lesen von der Konsole oder Schreiben in eine Datei, auch ein Seiteneffekt und somit nicht erlaubt. Hier fragt sich der eine oder die andere bestimmt, wie man mit dieser Einschränkung in der Praxis umgehen kann: In der funktionalen Programmierung geht es nicht darum, alle Seiteneffekte zu verbieten. Vielmehr will man bewusst mit diesen umgehen und funktionalen von nicht-funktionalem Code trennen. Inhaltlich zeige ich in diesem Artikel nur den grundlegendsten Teil der funktionalen Programmierung. Jedem, der tiefer in dieses Thema mit Kotlin einsteigen will, sei das Buch Functional Programming in Kotlin *[4]* sowie die Library Arrow *[5]* empfohlen.

Quellen

[1] Paul Chiusano, Rúnar Bjarnason (2015): *Functional Programming in Scala*. Mannning Publications Co, Shelter Island NY

[2] *https://en.wikipedia.org/wiki/Anonymous_function*

[3] *https://de.wikipedia.org/wiki/Haskell_Brooks_Curry*

[4] Marco Vermeulen, Rúnar Bjarnason, and Paul Chiusano (2020): *Functional Programming in Kotlin*. Manning Publications Co, Shelter Island NY

[5] *https://arrow-kt.io/*

Stefan López Romero

MaibornWolff GmbH

stefan.lopez@maibornwolff.de

Stefan López Romero ist IT-Architekt bei MaibornWolff. Der Diplom-Informatiker beschäftigt sich seit vielen Jahren mit funktionaler Programmierung in verschiedenen Sprachen und versucht diese, wo immer es geht, im Projektalltag anzuwenden.

Meetings sind giftig – Zeit für ein Detox!

Kerstin Gronauer und Timothée Bourguignon, MATHEMA Software GmbH

„Meetings sind giftig." Das haben David Heinemeier Hanson (DHH) und Jason Fried schwarz auf weiß in „Rework" [1] geschrieben. Zehn Jahre später sind wir kaum weiter. In dem einen oder anderen Punkt werden Sie sich bestimmt wiedererkennen: zu viele Teilnehmer, zu lang, nicht fokussiert, keine Agenda, keine Konsens-Kultur, keine Ergebnisse. Am Ende bleibt die Aufregung über verschwendete Zeit. Und trotzdem überleben diese verdammten Meetings immer weiter. Kaum haben wir es geschafft, ein Meeting abzuhaken, poppen zwei neue auf – wie eine moderne Hydra.

Aber Moment mal – Meetings sind doch eigentlich fantastisch! In einer Gruppe ist man nicht allein. Für Fehlentscheidungen kann kein Einzelner verantwortlich gemacht werden. Meetings stapeln sich wunderbar in Outlook und schwuppdiwupp ist der Arbeitstag rum! Wie wunderbar! Sarkasmus beiseite: Meetings sind eine notwendige Plage. Wir können sie aus Informationsmanagementsicht nicht vermeiden, müssen sie also so effizient und effektiv wie möglich gestalten. Best Practices für Meetings zu beleuchten ist Ziel dieses Artikels.

Synchronizität versus Asynchronität

Das effektivste Meeting ist ein Meeting, das nicht stattgefunden hat. DHH und Jason Fried haben es damals plakativ so ausformuliert: „Was, wenn ihr zu dem nächsten Termin nicht erscheinen würdet?" Als Teilnehmer sieht man so etwas positiv, aber als Veranstalter wohl eher nicht.

Kennen Sie den Unterschied zwischen einem Termin und einem Workshop? Naiverweise hätten wir gedacht, dass in einem Workshop etwas erarbeitet wird und Termine für Synchronisation oder kurze Entscheidungsfindung genutzt werden. Synchrones Erarbeiten von Themen, wie in einem Workshop, scheint sehr oft eine gute Lösung zu sein. Wir treffen uns, machen ein Brainstorming, tragen Input zusammen oder formulieren gemeinsam eine Botschaft. Synchrones Erarbeiten ist wichtig und nützlich, allerdings nur, wenn die gegenseitige Befruchtung notwendig ist. Wenn Kreativität, Reflektion etc. benötigt wird, kann Synchronizität kontraproduktiv sein. Überlegen Sie selbst, wie häufig Sie sich in Terminen fehl am Platz gefühlt haben, nichts beitragen konnten oder kein Interesse am Thema hatten, aber aufgrund von Politik teilgenommen haben.

Ihre erste Aufgabe als einladende Partei ist es, für eine Reduzierung von Meetings zu sorgen. Mit der richtigen Balance zwischen Synchronizität und Asynchronität fördern Sie eigene Arbeit an Themen, die am Ende nur kurz abgeglichen werden müssen.

Kurze Meetings, Schlüssel zum Erfolg

Wie lang sollte ein Termin überhaupt dauern? Gibt es da eine Faustregel? Und brauchen wir wirklich viel Zeit? Laut der Parkinson'schen Gesetze *[2]* blähen sich Meetings nur deswegen so auf, weil einfach die Zeit genutzt wird, die zur Verfügung steht. Knappheit schafft Fokus.

Wenn man sich an die hier beschriebenen Best Practices hält, sind Meetings von weniger als 30 Minuten meist ausreichend und man umgeht die Problematik, dass die Aufmerksamkeit leidet. Als Standardmaß empfehlen wir 22 Minuten, nach dem „22 Minuten Meeting"-Format von Nicole Steinbock *[3]*. Warum aber ausgerechnet 22 Minuten? Weil wir Menschen uns noch nicht beamen können und selbst Rechner nicht durchgehend arbeiten können.

Zeit für Pausen und „Meta"-Arbeit einkalkulieren

Werfen wir einen Blick in den Kalender von „Bob": Er leitet von 8:00 bis 9:00 Uhr einen Architekturreview-Termin, der meist rechtzeitig endet. Aber er müsste eigentlich danach noch das Protokoll nachbereiten, Mails dazu verschicken und den Folgetermin einstellen. Um 9:00 fängt der Jour fixe mit allen Team-Leads im Nachbarraum an. Selten schafft er all diese To-dos, bevor es losgeht. Und nie hat er Zeit für einen Kaffee. Bob verlässt heute den Raum kurz vor 10:00, rennt ins Nachbargebäude, wo sein Team arbeitet, und nimmt am Daily Scrum teil... und so geht es weiter. Abends, wenn der Termin-Marathon vorbei ist, kann er endlich sein(e) Protokoll(e) schreiben... sofern er sich noch an die Details erinnert. Zeit für Reise, Vor- und Nachbereitung, Aufräumen, Ein-Getränk-Holen, Frische-Luft-Schnappen etc. müssen miteinkalkuliert werden. All diese kleinen Tätigkeiten rund um ein Meeting tragen zum Erfolg eines Termins bei und genau deswegen sind dann unsere Meetings nur 11, 22 oder 44 Minuten lang.

Agenda rechtzeitig schicken

Ein Termin muss vorbereitet werden. Nicht nur vom Veranstalter/Moderator, sondern auch von den Teilnehmern. Nicht mal eine Überraschungsparty kommt ohne Vorbereitung aus. Schicken Sie als Veranstalter eine Agenda spätestens drei Tage vor dem Termin, um Ihren Kollegen Zeit für die Vorbereitung zu geben. In der Agenda beschreiben Sie Ziele des Meetings, Vorbereitungsschritte und was mit dem Output dieser Vorbereitung passieren wird.

Teilnehmer namentlich in der Einladung erwähnen

Haben Sie schon einmal über das Wort „Einladung" nachgedacht? Es impliziert, dass „Nein" eine mögliche Antwort ist. Oft sagen Kollegen nur zu, um keine wichtige Entscheidung zu verpassen, nicht erpressbar zu sein oder einfach reflexartig.
Viel besser ist es, in der Einladung den Teilnehmern den Termin schmackhaft zu machen. Wenn wir jemanden unbedingt dabeihaben wollen, sollten wir es explizit äußern.

Hier ein Beispiel: „Maria, deine Git-Kenntnisse sind für diesen Termin besonders wichtig. Zusammen mit Bobs Erfahrung zu unserer Serverlandschaft sollten wir schnell Lösungsansätze für unser CI-Problem finden können." Maria und Bob wissen jetzt ganz genau, warum ihre Teilnahme wichtig ist. Maria weiß nun, dass sie im Termin als Git-Expertin gebraucht wird und nicht als ehemaliger Projekt-Lead. Wir konnten auch ein Problem identifizieren: Mark wollten wir ursprünglich einladen, konnten aber keinen passenden Grund für seine Teilnahme formulieren. Er wäre nur aus politischen Gründen eingeladen worden und hätte wenig zur Lösungsfindung beitragen können.

Wenn Sie diesen ersten Schritten folgen, sollten Ihre Termine schon deutlich besser laufen. Die Termine sind dann kurz, vorbereitet, finden nur mit den richtigen Teilnehmern und unter besten Konditionen statt. Es kann also nichts mehr schiefgehen... oder doch? Die Durchführung selbst kann noch einige Probleme bereithalten. Das schauen wir uns als Nächstes an.

Moderators-Warm-up, der frühe Vogel fängt den Wurm

Für den Moderator fängt die Moderation des Termins bereits an, bevor der erste Teilnehmer den Raum überhaupt betreten hat. Er muss früh genug auftauchen, um potenzielle Probleme bis zum Terminstart gelöst zu haben. Was kann zum Beispiel alles schiefgehen? Frische Luft im Raum wäre gut. Es fehlt ein passender Adapter für den Beamer. Im Flipchart sind keine leeren Blätter. Ein Moderationskoffer fehlt. Das Whiteboard muss gewischt werden. Die Stifte funktionieren nicht mehr. Der Raum ist nicht richtig ausgestattet (richtige Anzahl an Stühlen). Es ist kein Wasser mehr für die Teilnehmer da. Und so weiter.

Der Moderator muss sich den Termin vorstellen, überlegen, wie der Raum genutzt werden könnte, und ihn entsprechend vorbereiten. Der Moderator gibt nicht nur den Rahmen vor, sondern auch Energie an den Raum ab. Wenn der Moderator Energie ausstrahlt, Spaß hat und sich wohlfühlt, färbt dies auf die Teilnehmer ab. Wir bereiten Workshops mit Musik vor und drehen die Musik erst dann ab, wenn die Hälfte der Teilnehmer schon angekommen ist.

Land in Sicht

Die Agenda muss für jeden Teilnehmer sichtbar im Raum platziert sein. Es ist besonders wichtig, dass Sie darauf als Moderator jederzeit referenzieren können. Wenn eine Diskussion aus der Bahn läuft, sollten Sie auf die Agenda verweisen und fragen können, was die Teilnehmer machen wollen.

Hier ein Beispiel: „Wir hatten für diesen Punkt zehn Minuten geplant und die Zeit ist abgelaufen. Wir können noch weitere fünf oder zehn Minuten spendieren, werden dann aber diese Agenda-Punkte voraussichtlich heute nicht mehr behandeln können. Wollt ihr dieses Thema parken und mit der Agenda weitermachen oder lieber diesen Punkt zu Ende diskutieren?"

Fire and forget

In seinem Buch „Getting Things Done" empfiehlt David Allen *[4]*, alle Tasks aufzuschreiben, die nicht sofort erledigt werden können, sie einzuordnen und zu vergessen. Es handelt sich hierbei um einen Trick, um die „Cognitive Load" (kognitive Belastung) zu reduzieren. Helfen Sie deswegen Ihren Teilnehmern mit einem Themen-Parkplatz. Man kann Flipchart oder Whiteboard nutzen, um Ideen zu sammeln, die nicht direkt zu der aktuellen Diskussion beitragen, aber wichtig sind.

Oh, unsere Teilnehmer sind jetzt da!

„Let's get it started!" – Deutsche Pünktlichkeit

Es ist genau 9:00 Uhr. Wir können starten. Aber was passiert, wenn Teilnehmer fehlen? Ja, es kann immer noch passieren, dass Teilnehmer zu spät oder gar nicht ankommen. Durch die vielen Änderungen, die wir bereits diskutiert haben, ist dieses Risiko aber drastisch gesunken:

- Aufgrund der 22-Minuten-Meeting-Kultur haben Ihre Kollegen keinen Grund mehr, zu spät anzukommen, und wissen, dass fünf Minuten Verspätung bereits 22 Prozent des Termins ausmachen.
- Weil Sie Ihre Kollegen persönlich einladen, wissen diese auch, wie wichtig ihre Teilnahme ist.
- Weil Sie die Agenda und die Vorbereitungsschritte vorab verschickt haben, sind Ihre Kollegen schon vorbereitet.
- Wenn Sie das Spiel ein paar Mal gemacht habt, wissen Ihre Kollegen, dass Ihre Meetings rund laufen!

Loben Sie rechtzeitiges Ankommen und starten Sie danach rechtzeitig. Und was, wenn eine wichtige Person doch nicht da ist? Dann starten Sie trotzdem rechtzeitig und entscheiden als Gruppe, was Sie tun wollen. Starten? Warten? Vertagen? Es ist nicht Ihre alleinige Entscheidung als Moderator, sondern die der anwesenden Gruppe.

Wo wollen wir hin?

Es ist genau 9:00 Uhr. Alle sind da. Lasst uns starten! Der erste Punkt eines jeden Termins ist es, die Zielsetzung des Meetings zurück ins Gedächtnis zu rufen, auch wenn Sie die Agenda in der Einladung ausführlich beschrieben haben. Die Agenda ist sichtbar für alle Teilnehmer aufgehängt. Stellen Sie in etwa 30 Sekunden die Zielsetzung vor und prüfen Sie, ob alle Kollegen das gleiche Verständnis davon haben. Wenn Sie eine Diskrepanz erkennen, müssen Sie reagieren und vielleicht sogar den Termin vertagen. Klären Sie danach die Prioritäten in folgender Reihenfolge:

1. Wichtige und dringende Themen
2. Dringende und weniger wichtige Themen
3. Wichtige und nicht dringende Themen

Und hoffentlich haben Sie die Themen, die weder wichtig noch dringend sind, von Ihrer Agenda schon längst entfernt... oder?

Zuletzt werfen Sie einen Blick auf die Beschlussfähigkeit. Sind die Teilnehmer nicht vorbereitet oder es fehlt eine notwendige Person, dann ist es in Ordnung, den Termin zu verschieben.

Der Moderator: Schulz und Schulz und Schulz

Es ist genau 9:02 Uhr. Alle sind da. Wir wissen, warum wir versammelt sind. Lasst uns starten! Oder doch nicht... Die Rolle des Moderators sollte bestimmt werden. Sie kann in drei unterschiedliche Rollen geteilt werden: Moderator, Timekeeper und Protokollant. Für kürzere Meetings mit wenigen Teilnehmern kann dies komplett überflüssig sein. Für größere, schwierigere Meetings, in denen mehr Moderation notwendig ist, kann es hilfreich sein, für Timekeeping und Protokollierung je eine eigene Person zu bestimmen und/oder sogar mehrere Moderatoren zu haben. Die Agenda sollte für die einzelnen Punkte bereits einen zeitlichen Rahmen aufweisen. Der Timekeeper muss diesen dann „nur noch" im Auge behalten.

Hierfür empfehlen wir die Nutzung einer Uhr, die sichtbar für alle Teilnehmer aufgestellt werden kann und eine Alarmfunktion aufweist. Diese hilft Ihnen dabei, Diskussionen abzubrechen: Wenn es Zeit ist, meldet sich die Uhr, auch wenn Ihr Kollege gerade einen seiner legendären Monologe hält. Dann ist es die Aufgabe des Moderators, die Reaktion zu kanalisieren: Abbrechen? Zum nächsten Punkt übergehen? Weitermachen?

Nun zum Protokoll: Wie protokolliert wird, ist meist zweitrangig. Wichtig ist, dass alle Teilnehmer sich auf den Termin konzentrieren können und nicht alles selbst notieren müssen. Ziel ist, das Protokoll während des Termins zu erstellen, sodass am Ende nur noch Bilder gemacht werden müssen. Nutzen Sie Klebezettel, um die Organisation laufend ändern und teilweise „neu schreiben" zu können. Mit etwas Übung funktioniert das und trägt zur Kooperation aller Teilnehmer bei.

Meeting Agreements

Es ist genau 9:03 Uhr. Alle sind da. Wir wissen, warum wir versammelt sind. Wir wissen, wer moderiert und wer protokolliert. Lasst uns starten! Rufen Sie sich noch kurz Ihre Grundregeln für Meetings ins Gedächtnis. Mittelfristig werden Ihre Kollegen alle diese Regeln kennen und sie sichtbar im Raum hängen sehen. Dann brauchen Sie sie nicht mehr allen Teilnehmern ausführlich zu erklären. Beispiele für Meeting Agreements: keine Laptops oder Handys, respektvoller Umgang miteinander, ein gemeinsames (Hand-)Zeichen bei thematischem Driften, Speaker-Tokens und so weiter. Achtung: Diese Regeln müssen einmal in der Gruppe erstellt worden und für alle verständlich sein.

Ein Ablaufbeispiel: „Liebe Kollegen, damit wir so schnell wie geplant durchkommen, halten wir uns bitte an unsere Regeln" mit dem Hinweis auf ein Regel-Flipchart, das für alle sichtbar aufgehängt ist. „Behaltet bitte eure Handys in der Tasche und wenn ihr doch rangehen müsst, geht bitte vor die Tür. Dann wissen auch alle Teilnehmer, dass ihr gerade nicht ansprechbar seid. Klappt bitte eure Laptops zu; wie gerade abgestimmt, schreibt Leo das Protokoll für uns. Wie immer gehen wir bitte respektvoll miteinander um, insbesondere bei den Themen X und Y, wo es Konfliktpotenzial gibt. Last, but not least, ihr erinnert euch an unser „Driften wir gerade ab"-Zeichen? Es hilft uns zu erkennen, wenn die Diskussion für euch vielleicht nicht mehr zielführend ist."

Fred, lass die anderen reden!

Es ist genau 9:04 Uhr. Alle sind da. Wir wissen, warum wir versammelt sind. Wir wissen, wer moderiert und wer protokolliert. Wir haben uns unsere Grundregeln ins Gedächtnis gerufen. Lasst uns starten! In manchen Gruppen ist es hilfreich, ein Speaker-Token einzuführen, um zu vermeiden, dass sich die Leute gegenseitig ins Wort fallen und sich nicht ausreden lassen. Das Token kann vom Moderator auch als Mittel genutzt werden, um alle Leute im Raum in die Diskussion einzubeziehen. Als Token kann alles Mögliche genutzt werden. Aber vielleicht keinen richtig schweren Gegenstand, wie Chet Rong aus „The Wong Way to do Agile" *[8]* es empfiehlt!

Check-in

Es ist genau 9:05 Uhr. Alle sind da. Wir wissen, warum wir versammelt sind. Wir wissen, wer moderiert und wer protokolliert. Wir haben uns unsere Grundregeln ins Gedächtnis gerufen. Wir haben ein Speaker-Token definiert. Wir können starten! Die Profis unter uns können auch noch einen „Check-in" (nach den McCarthy Core Protocols *[5]*) machen und ihre eigenen Gefühle kurz reflektieren. Wenn beispielsweise Ida zu Beginn erzählt, dass sie „ängstlich bei dem Thema Zukunft des Projekts ist, weil es möglicherweise massive Auswirkung auf ihr Team hat", wissen sofort alle anderen im Raum, warum sie auf einen schlechten Witz vielleicht nicht so gut reagieren wird.

Jetzt kann der Termin endlich wirklich starten. Alle Vorbereitungsübungen dauern am Anfang ein paar Minuten. Mit der Zeit werden sich diese auf einen Augenblick reduzieren. Die Meeting Agreements sind sichtbar und müssen nicht mehr angesprochen werden. Protokollant und Timekeeper melden sich freiwillig. Die Agenda ist klar. Zwei Minuten nach Start sind wir dann schon tief in der Materie. Was in dem Termin wirklich geschieht, ist Aufgabe des Moderators... und wäre Grund für einen eigenen Artikel. Es fehlt nun nur noch der Abschluss des Termins.

Time to say Goodbye

Der Abschluss selbst braucht gegebenenfalls Zeit, je nachdem, wie gut der Termin gelaufen ist. Fünf bis zehn Minuten sind nicht unüblich. Es muss vor allem klar sein, was die nächsten Schritte sind. Da empfehlen wir als Moderatoren: Reformulieren! Wenn Sie sich nicht sicher sind, ob etwas richtig verstanden wurde, einfach absichtlich falsch reformulieren und schauen, wie die Teilnehmer reagieren. Zum Abschluss gehört auch, das Protokoll zu verschicken. Verlassen Sie also nicht den Raum, ohne Bilder gemacht zu haben. Installieren Sie gegebenenfalls eine Scanning App auf Ihrem Smartphone. Diese Apps können zudem Bilder editieren und daraus gleich PDF-Dateien erzeugen.

Feedback einholen

Eine letzte Aktion gibt es noch, bevor der Raum verlassen werden sollte: eine kurze Feedback-Runde. Ob im „Return On Time Invested" (ROTI)-Format („Wie gut war die Zeit in dem Termin investiert?"), durch „Amazon-Review-Sterne", „Fist-of-five", visuelle Akkuladung auf dem Whiteboard/Flipchart (leer – rot , mittelvoll – orange oder voll – grün) oder mit einfachen Schlagwörtern... es ist einfach wichtig für den Organisator zu wissen, was an dem heutigen Termin gut oder schlecht gelaufen ist.

Fazit

Unsere Meeting-Kultur ist wahnsinnig schlecht. Mit unseren Tipps werden Sie diese wieder geradebiegen können. Üben und experimentieren Sie mit diesen Best Practices und erzeugen Sie Ihre eigenen. Lernen Sie diese Regeln, um sie dann besser für Sie passend

einfach zu brechen. Viel Spaß damit! Wenn Sie noch andere Best Practices teilen wollen, aber gerne doch, kontaktieren Sie uns!

Quellen

[1] Rework: *https://basecamp.com/books/rework*

[2] Parkinson'schen Gesetze: *https://de.wikipedia.org/wiki/Parkinsonsche_Gesetze*

[3] „22 Minuten Meeting": *http://22minutemeeting.info*

[4] Getting Things Done: *https://gettingthingsdone.com/*

[5] McCarthy Core Protocols: *https://liveingreatness.com/core-protocols/*

Kerstin Gronauer

MATHEMA Software GmbH

kerstin.gronauer@mathema.de

Kerstin Gronauer ist als Scrum Master bei der MATHEMA Software GmbH tätig. Sie ist in Teams unterwegs, um sie in ihrem eigenen Weg zur agilen Arbeitsweise zu bestärken und zu fördern. Neben Agilität interessiert sie sich sehr für Visual Thinking und Themen rund um New Work.

Timothée Bourguignon

MATHEMA Software GmbH

timothee.bourguignon@mathema.de

Timothée Bourguignon ist als Chief Learning Officer, Head of Agile und Agile Coach bei der MATHEMA Software GmbH tätig. Seine Themenschwerpunkte umfassen Agilität, Lean, Scrum, Kanban, Mentoring, Teambuilding und noch einiges mehr. Daneben hält er regelmäßig Vorträge auf Fachkonferenzen und schreibt Artikel für Fachmagazine. In seiner Freizeit kümmert er sich mit seinem Podcast „Developer's Journey" (devjourney.info) um die Verbreitung der Vielseitigkeit in unserer Entwicklerwelt.

Von 8 auf 11: Erfahrungen bei der Migration von JavaFX-Anwendungen

Manuel Mauky, Carl Zeiss Digital Innovation

Das UI-Toolkit JavaFX hat eine lange und wechselhafte Geschichte hinter sich. Zunächst als „JavaFX Script" mit eigener Programmiersprache entwickelt, fand mit Version 2 ein kompletter Neubau in purem Java statt und das Framework zog in das Oracle JDK ein. Seitdem wurde JavaFX zwar als Teil des JDK beziehungsweise JRE ausgeliefert, nahm jedoch immer eine gewisse Sonderrolle ein: Anders als Swing war JavaFX nie Teil der Java-Spezifikation und zum Beispiel unter Linux nicht Bestandteil des OpenJDK-Package. Auch hinsichtlich der Weiterentwicklung durch Oracle gab es in der Vergangenheit einige Bedenken und Unklarheiten. Im März 2018 hat Oracle schließlich bekannt gegeben, dass JavaFX ab JDK 11 nicht mehr als Teil des JDK mitgeliefert wird. Stattdessen muss JavaFX nun – genau wie andere Bibliotheken und Frameworks auch – separat zur Anwendung hinzugefügt und mit ausgeliefert werden. Die Weiterentwicklung wurde in Form des OpenJFX-Projekts als Teil des OpenJDK ebenfalls an die Community abgegeben. Gleichzeitig bietet Oracle aber bis mindestens 2022 weiterhin kommerziellen Support für JavaFX im Oracle JDK 8 an. Dieser Wechsel war spätestens mit der Einführung des Java Platform Module System mit Java SE 9 absehbar, denn ein wichtiges Ziel war schließlich die Modularisierung und Neustrukturierung des JDK selbst.

Für JavaFX-Entwickler hat dieser Umstand sowohl Vor- als auch Nachteile. Zum einen ist das Tempo der Weiterentwicklung von JavaFX nun nicht mehr an die des OpenJDK gebunden und auch die Möglichkeit, sich selbst bei der Weiterentwicklung einzubringen, wurde und wird stetig verbessert. Auf der anderen Seite dürfte der Wegfall beziehungsweise die Reduktion der Entwicklungskapazitäten von Oracle einige Lücken hinterlassen. Es wird sich zeigen, ob die Community diese Lücken schließen kann. Allerdings lassen die weiterhin hohe Aktivität auf den entsprechenden Mailinglisten *[1]* sowie die Features und Verbesserungen der neueren Versionen, beispielsweise die Unterstützung für natives Rendering in JavaFX 13, optimistisch in die Zukunft blicken.

Entwicklungsteams, die existierende JavaFX-Anwendungen betreuen und weiterentwickeln, stellt dieser Wechsel aber auch vor besondere Herausforderungen – zusätzlich zu den auch bei anderen Java-Anwendungen auftretenden Fragestellungen, wie der Wahl der richtigen OpenJDK-Distribution und dem Umgang mit dem neuen Module System. Dieser Artikel thematisiert deshalb einige Probleme, Schwierigkeiten und Erfahrungen bei der Migration von JavaFX-Anwendungen von Java 8 auf Java 11. Ist dieser Schritt erst einmal gemacht, sollten weitere Versionssprünge auf neuere Versionen keine allzu großen Schwierigkeiten mehr bereiten.

Wahl der OpenJDK-Distribution

Große Versionsupgrades der eingesetzten Basis-Technologien gehören sicherlich zu den weniger angenehmen Aufgaben in den meisten Entwicklungsabteilungen: Weil für Kunden und Anwender häufig keine sichtbaren Verbesserungen entstehen, werden fällige Updatearbeiten gerne nach hinten verschoben, obwohl damit natürlich technische Schulden angehäuft werden, deren Abtragung in der Zukunft nicht einfacher wird. Durch den Lizenzwechsel, den Oracle Mitte 2018 für das Oracle JDK angekündigt hat, wurden Firmen, die das JDK bisher kostenfrei einsetzten, nun aber zum Handeln gezwungen.

In der Theorie ist die Situation weniger dramatisch, denn es stehen mehrere Distributionen des OpenJDK zur Verfügung, aus denen nach eigenem Geschmack ausgewählt werden kann. Für JavaFX stellt sich die Angelegenheit jedoch etwas anders dar. Ist man bereit, den Sprung auf Java 11 oder höher mitzugehen, ist es einfach: Man wählt ein beliebiges OpenJDK aus und zieht JavaFX als Drittbibliothek hinzu. Möchte man hingegen erstmal bei Java 8 bleiben und nur den JDK-Anbieter wechseln, muss man genauer hinschauen, denn nicht jede Distribution liefert auch OpenJFX mit. Azul, Amazon Corretto und Bellsoft Liberica JDK haben beispielsweise JavaFX mit an Bord. Dagegen kommen Red Hat und AdoptJDK ohne JavaFX daher.

API-Kompatibilität

Hinsichtlich der API-Kompatibilität zwischen den einzelnen Versionen war JavaFX in der Vergangenheit relativ unkompliziert, denn auf größere Brüche wurde glücklicherweise verzichtet. Mit dem Sprung von JavaFX 8 auf JavaFX 9 war dies hingegen nicht mehr so einfach möglich, denn wie auch im restlichen Java gab es einige private APIs, die zwar anhand ihres Packagenames in „com.sun.javafx.*" oder Präfixes wie „impl_" eigentlich klar als nicht für die Benutzung vorgesehen erkenntlich waren, in der Praxis aber eben doch häufig eingesetzt wurden. Durch die Einführung des strikten Modulsystems standen diese APIs jedoch nicht mehr zur Verfügung. Im Fall von JavaFX war die Situation besonders knifflig, da für einige Zwecke bisher kein offizieller Ersatz existierte. Dazu zählen beispielsweise die sogenannten „Skins", mit denen das Aussehen und Verhalten von UI-Controls angepasst werden kann. Mit dem Java Enhancement Proposal (JEP) 253 wurden im Vorfeld solche und ähnliche Fälle gesammelt und Public-APIs als Ersatz entworfen.

Module System

Das neue Java Platform Module System (JPMS) dürfte wohl zu den kontroversesten Themen der letzten Jahre unter Java-Entwicklern gehören. Auf der einen Seite leuchtet der Zweck und Nutzen durchaus ein, auf der anderen Seite hält der Umstieg einige Fallstricke bereit, die nicht nur JavaFX, sondern ganz allgemein Java-Anwendungen betreffen.

Ob die Umstellung der eigenen Anwendung auf eine neuere Java-Version unter Nutzung des Modulsystems gelingt, hängt zu einem großen Teil von den eingesetzten Dritt-Bibliotheken ab. Ein häufiges Problem sind beispielsweise sogenannte „Split Packages", das bedeutet, dass zwei Bibliotheken das gleiche Package enthalten. Dies ist häufig bei Bibliotheken der Fall, die ihre Funktionalität auf mehrere separate Artefakte aufteilen. Letztlich sind Anwendungsentwickler hier auf die Mitwirkung der Library-Autoren angewiesen, die ihre Bibliotheken entsprechend anpassen müssen, wobei diese natürlich nicht selten ebenfalls von weiteren Bibliotheken abhängen. Während mittlerweile viele Bibliotheken nachgezogen sind, scheiterte eine Umstellung in der Anfangszeit nach dem Release von Java 9 noch häufiger an fehlender Unterstützung durch Drittbibliotheken.

Doch auch, wenn Bibliotheken kompatibel zum neuen Modulsystem sind, müssen auch die eingesetzten Werkzeuge wie Buildsystem und IDE mitspielen. Und hier gab und gibt es einige Probleme, die besonders JavaFX betreffen. In JavaFX lassen sich zwar, ähnlich wie bei Swing, alle UI-Controls im Java-Code zusammenstecken – deutlich angenehmer und weiter verbreitet ist allerdings die Verwendung von FXML, einem XML-Dialekt, mit dem die Struktur der Container und Controls festgelegt wird. FXML ist auch das Format, das der „JavaFX SceneBuilder" verwendet.

Um zur Laufzeit eine solche FXML-Datei zu laden, wird der Klasse FXMLLoader ein „java.net.URL" übergeben, das auf die entsprechende FXML-Datei verweist. Damit dies bei eingeschaltetem Module System funktioniert, müssen einige Dinge in der „module-info.java"-Datei berücksichtigt werden. Zunächst müssen alle verwendeten JavaFX-Module mittels `require` in die Liste der Abhängigkeiten aufgenommen werden. JavaFX stellt die Module „javafx.base", „javafx.controls", „javafx.fxml", „javafx.graphics", „javafx.media", „javafx.swing" sowie „javafx.web" bereit, wobei „javafx.base" von allen anderen automatisch mit adressiert wird. Die wichtigsten Module sind „javafx.controls" mit sämtlichen UI-Controls und „javafx.graphics", das unter anderem die Basisklassen des JavaFX Scene Graph enthält und damit für alle JavaFX-Anwendungen benötigt wird. Im nächsten Schritt muss das Package, das die Start-Klasse enthält (die Klasse, die von „javafx.application.Application" ableitet), mittels `exports com.example.myapp to javafx.graphics` für JavaFX verfügbar gemacht werden.

Im letzten Schritt bringt man nun noch den FXML-Mechanismus zum Laufen. Der „FXMLLoader" arbeitet intern mit Reflection, um die in der FXML-Datei definierten UI-Controls mit den dafür vorge-

sehenen Feldern in der zugehörigen Controller-Klasse zu verbinden. Reflection wird aber durch das Module System standardmäßig verhindert, weshalb wir es explizit aktivieren müssen. Dazu dient die Anweisung „opens com.example.myapp to javafx.fxml", die für jedes von FXML adressierte Package wiederholt werden muss. Hat man eigene UI-Komponenten entwickelt, die in den FXML-Dateien im Einsatz sind, ist noch ein weiterer Schritt notwendig: Wir müssen die Packages, in denen diese Komponenten liegen, mittels `exports com.example.myapp.components to javafx.fxml` für das „javafx.fxml"-Module freigeben. Damit sollte die Anwendung nun prinzipiell laufen. Es lauern aber noch einige Probleme, je nachdem, welche Werkzeuge im Einsatz sind.

Probleme mit „getResource()"

Ein Problem tritt bei der Verwendung von Gradle als Buildsystem auf: Wie bei Maven und Gradle üblich werden Java-Sourcecode-Dateien im Verzeichnis „src/main/java" und alle sonstigen Dateien – damit auch FXML-Files – unter „src/main/resources" abgelegt. Als Endergebnis des Build-Prozesses werden alle Dateien zusammen in eine JAR-Datei gepackt. Bei den Zwischenergebnissen unterscheiden sich Maven und Gradle allerdings. Bei Maven landen alle Dateien im gleichen Verzeichnis unter „target/classes", auch die Ressourcen-Dateien. Gradle dagegen behält die Trennung bei und legt kompilierte Java-Klassen unter „build/classes" und Ressourcen unter „build/resources" ab.

Dies ist dann wichtig, wenn man die Anwendung beispielsweise aus der IDE heraus oder mit dem Befehl `gradle run` direkt und ohne Umweg über die JAR-Datei starten möchte. Prinzipiell ist dies auch kein Problem, denn wichtig ist zur Laufzeit nur, dass die gewünschten FXML-Dateien mittels `getResource()` gefunden werden. Bei Anwendungen ohne Module System kommt der klassische `classpath` zum Einsatz, auf dem standardmäßig beide Verzeichnisse aufgeführt werden und damit Dateien ohne Probleme geladen werden können. Sobald jedoch dem Projekt eine „module-info.java"-Datei hinzugefügt und damit das Module System aktiviert wird, schlägt das Laden fehl. Aus der Perspektive von Java gehören die Ressourcen nicht zum eigenen Module und sind damit auch nicht verfügbar.

Zur Lösung existieren (mindestens) zwei Ansätze: Um zusätzliche Bestandteile zu einem Module hinzuzufügen, stellt Java den Kommandozeilenparameter `--patch-module` zur Verfügung. Damit können wir Java mitteilen, dass unsere Ressourcen-Dateien ebenfalls zu diesem Module gehören. Die zweite Möglichkeit wäre, Gradle so zu konfigurieren, dass Dateien unter „src/main/resources" ebenfalls in das Verzeichnis der Klassen kopiert werden. Beide Varianten sind in *Listing 1* zu sehen.

JavaFX-Gradle-Plug-in

Ein weiteres Problem lauert bei der Verwendung des offiziellen JavaFX-Plug-ins für Gradle *[2]*. Dieses verspricht in der GitHub-Beschreibung: „Simplifies working with JavaFX 11+ for Gradle projects". Tatsächlich vereinfacht sich das Basis-Setup durch das Plug-in um einiges. Der Grund dafür ist, dass JavaFX, anders als die meisten anderen Drittbibliotheken, auch auf nativen und damit betriebssystemspezifischen Code aufbaut. Deshalb liegen die Bibliotheken im Maven Central in mehreren Geschmacksrichtungen, jeweils für Mac OS, Linux und Windows vor. Bei der Deklaration der Abhängigkeit im Buildskript muss das Betriebssystem als Classifier mit angegeben werden, zum Beispiel „org.openjfx:javafx-controls:11.0.2:mac". Um plattformübergreifend zu arbeiten, müssen also entsprechende Weichen vorgesehen werden. Des Weiteren müssen die JavaFX-Module händisch zum Module-Pfad hinzugefügt werden, was mit dem Parameter `--add-modules=javafx.controls` geschieht, ähnlich wie weiter oben für Patch Modules beschrieben. Das JavaFX-Gradle-Plug-in erledigt diese Aufgaben automatisch. Leider gibt es im Plug-in einen seit längerer Zeit ungefixten Bug *[3]*, der die Ausführung von Unittests verhindert, sobald das Module System aktiviert ist.

Normalerweise laufen selbst bei modularisierten Projekten die Unittests ohne aktives Module System. Tests befinden sich in einem separaten Verzeichnis unter „src/test/java" ohne eigene „module-info". Dies ist auch sinnvoll, weil im Test-Kontext teils vollkommen andere Bibliotheken (beispielsweise JUnit) im Einsatz sind, die in der Hauptanwendung nichts verloren haben. Andernfalls müssten die Test-Abhängigkeiten im gleichen Scope wie normale Bibliotheken eingefügt und auch in der „module-info" eingebunden werden. Mit einem normalen Gradle-Setup funktioniert die Ausführung der Tests ohne Probleme. Bindet man aber das JavaFX-Plug-in ein, verhält sich das System so, als wäre das Module System auch bei der Testausführung aktiv. Entsprechend meckert der Compiler, dass beispielsweise JUnit nicht in der „module-info" als `required` markiert ist.

Bis das Problem im Plug-in gefixt ist, bleiben Entwicklern nur zwei Optionen. Entweder man entfernt das Plug-in und bildet die oben aufgezählten Konfigurationen selbst nach. Oder man bindet tatsächlich alle Test-Abhängigkeiten im gleichen Scope wie normale Bibliotheken ein und führt diese auch in der „module-info" auf. Dies kann jedoch insbesondere dann zur Komplikation werden, wenn im Test eine andere Implementierung für ein API genutzt werden soll als im Produktivsystem.

Extra-Starter-Klasse

Sollte der Umstieg auf Java 11 und aktiviertem Module System aufgrund von inkompatiblen Bibliotheken nicht gelingen, bleibt immer noch die Möglichkeit, das Module System im ersten Schritt ungenutzt zu lassen und nur das JDK zu aktualisieren. Dafür muss lediglich die „module-info"-Datei weggelassen werden.

Doch auch hier kann es zu Überraschungen kommen. Ein auf den ersten Blick kurioses Verhalten, das auftreten kann, ist die Fehlermeldung „Error: JavaFX runtime components are missing and are required to run this application". Diese Meldung weist eigentlich auf, dass bei einer modularisierten Anwendung die JavaFX-Module auf dem Module-Pfad fehlen. Entsprechend kann die Meldung hier in die Irre führen, denn wir haben gerade kein Module System im Einsatz. Der eigentliche Hintergrund ist: JavaFX-Anwendungen besitzen eine Haupt-Klasse, die von „javafx.application.Application" ableitet und die `start`-Methode implementiert. Diese ist der Einstiegspunkt in die Anwendung und bekommt die Haupt-Stage als Parameter übergeben, auf der die UI der Anwendung aufgebaut wird. Häufig besitzt diese Klasse auch die bei allen Java-Programmen übliche „public static void main(String...args)"-Methode, in der bei JavaFX der Aufruf der statischen Methode „Application.launch" erfolgt. Bei früheren Versionen von JavaFX funktioniert dies wunderbar. Mit Version 11 findet dagegen eine Prüfung statt, ob es sich bei der Klasse mit

```
plugins {
    id 'java'
    id 'application'
}

sourceCompatibility = JavaVersion.VERSION_11
mainClassName = "com.example.myapp.Main"

// Patch-Module
tasks {
    run {
        doFirst {
            jvmArgs = [
                "--module-path", classpath.asPath,
                "--patch-module", "mymodule=" + files(
                        sourceSets.main.resources.srcDirs
                    ).asPath,
                "--module", "myexamplemodule/" + mainClassName
            ]
            classpath = files()
        }
}

// Copy resources into class dir
sourceSets {
    main {
        output.resourcesDir = java.outputDir
    }
}
```

Listing 1

der Main-Methode um eine JavaFX-Application-Klasse handelt. In diesem Fall wird geprüft, ob „javafx.graphics" als benanntes Module auf dem Module-Pfad liegt. Das beschriebene Verhalten und die Begründung lassen sich auf der JavaFX-Mailingliste *[4]* nachlesen. Kurz gesagt möchte man erreichen, dass auch nicht-modulare Anwendungen stets das JavaFX-Module mit angeben. Auch wenn an eben dieser Stelle davon abgeraten wird, diesen Check zu umgehen, lässt sich anhand der beschriebenen Prüfungen leicht ableiten, wie die Anwendung auch ohne Setzen des Module-Pfads gestartet werden kann. Man muss lediglich die `main`-Methode in eine separate Starter-Klasse verschieben und dort die bisherige Haupt-Klasse als ersten Parameter der „Application.launch"-Methode übergeben.

Java-Packager

Wenn die Anwendung erst einmal mit Java 11 und Module System zum Laufen gebracht wurde, ist der nächste Schritt das Bundling. Bisher diente dazu das Werkzeug Java Packager (anfangs noch unter der Bezeichnung JavaFX Packager), das mit Version 8 Einzug ins JDK gehalten hat. Dieses Werkzeug ermöglicht es, aus der eigenen Anwendung eine ausführbare Datei (unter Windows zum Beispiel eine .exe-Datei) zu erzeugen. Dabei werden sowohl die benötigten Drittbibliotheken als auch das JDK mit gebündelt. Auf diese Weise lassen sich mögliche Probleme bei der Auslieferung aufgrund von fehlendem oder inkompatiblem JDK umgehen und die Benutzung entspricht eher den Gewohnheiten der meisten Nutzer. Auf der anderen Seite gibt es den Nachteil, dass die auszuliefernde Anwendung wesentlich größer ausfällt und Anwender keine (einfache) Möglichkeit mehr haben, unabhängig von Anwendungsbetreuern das JDK upzudaten. Es gilt also die Vor- und Nachteile im jeweiligen Fall abzuwägen. Entscheidet man sich für den Einsatz des Java Packager, ergibt sich zusätzlich die Möglichkeit, auch gleich Installer-Pakete für die eigene Anwendung zu erstellen, wobei unter Windows dafür noch zusätzliche Tools notwendig sind.

Leider wurde der Java Packager mit JDK 11 entfernt, ohne einen adäquaten Ersatz bereitzustellen. Stattdessen ist mit JEP 343 *[5]* ein neues Packaging-Tool angekündigt (zum Zeitpunkt der Entstehung des Artikels ist JDK 13 aktuell und das Packaging-Tool für JDK 14 geplant). Und obwohl Entwickler zunächst etwas allein gelassen werden, verspricht das neue Tool mit dem Namen „JPackager" tatsächlich ein mehr als würdiger Nachfolger zu werden. Denn die Modularisierung des JDK selbst, der eigenen Anwendung und Bibliotheken ermöglicht das Bundling von noch kompakteren Artefakten, wodurch auch das Problem der übergroßen Anwendungspakete minimiert wird. Für die Erzeugung von kompakten JDKs existiert bereits das Werkzeug „JLink", das in den Prozess des JPackager eingebunden sein wird.

Wer das neue Tool ausprobieren möchte, kann dies bereits mit Vorabversionen des OpenJDK 14 tun. Glücklicherweise ist die JavaFX-Community nach wie vor sehr aktiv und so wurden Backports des JPackager für JDK 11 bereitgestellt – allerdings ohne Gewährleistung *[6]*. Und tatsächlich funktioniert das Tool schon erstaunlich gut und erzeugt erwartungsgemäß deutlich kleinere Bundles. Bei einem Test konnte bei einer kleinen Anwendung eine Reduktion von etwa 200 MB auf 30 MB erzielt werden.

Allerdings existiert – anders als beim Java Packager von Java 8 – noch kein ausgereiftes Plug-in für Gradle oder Maven, das die Konfiguration und Bedienung des Werkzeugs abstrahiert und vereinfacht. Stattdessen müssen die einzelnen Schritte händisch durchgeführt oder in einem eigenen Batch-Skript automatisiert werden. In *Listing 2* ist ein solches Skript zu sehen, wobei an dieser Stelle aufgrund von jederzeit möglichen API-Änderungen und des Beta-Stadiums des JPackager keine Garantie für das Funktionieren übernommen werden kann. Der Leser möge es lieber als Inspirationsquelle für eigene Experimente sehen.

```
#!/bin/sh

module_path="$JAVA_HOME/jmods;./build/libs"

echo "calculcate dependencies"
deps=$(jdeps --module-path $module_path \\
--print-module-deps ./build/libs)

echo "create runtime"
jlink --no-header-files --no-man-pages --strip-debug \\
--compress=2 --verbose --module-path $module_path \\
--add-modules $deps --output build/runtime

echo "create executable"
jpackager create-image --runtime-image build/runtime \\
  --vendor "My Company" --strip-native-commands \\
  --icon src/main/resources/icon.ico \\
--module-path $module_path --input build/libs \\
--main-jar "myapp.jar" --output build/exe \\
--name MyApp

# or create an installer
echo "create installer"
jpackager create-installer --win-dir-chooser --win-shortcut \\
  --win-menu-group "My Company" --vendor "My Company" \\
  --strip-native-commands --icon src/main/resources/icon.ico \\
  --runtime-image build/runtime --module-path $module_path \\
  --input build/libs --main-jar "myapp.jar" \\
  --name MyApp --output build/installer
```

Listing 2

Fazit

Wir haben gesehen, dass auf dem Weg der Umstellung von JavaFX-Anwendungen von Java 8 auf Java 11 einige Stolpersteine liegen. Viele davon hängen direkt mit JavaFX selbst zusammen, beispielsweise Bugs in einigen Werkzeugen oder der Wegfall des Java-Packager-Tools. Auf der anderen Seite haben der Autor und seine Kollegen die Erfahrung gemacht, dass besonders bei größeren Anwendungen mehr und mehr allgemeine Probleme beim Umgang mit dem neuen Module System in den Vordergrund treten, die nichts mit JavaFX zu tun haben. Dies betrifft insbesondere inkompatible Dritt-Bibliotheken, die beispielsweise auf Reflection oder Bytecode-Manipulierung setzen und daher schwieriger mit der strengen Kapselung des Module System harmonieren. Eine weitere häufige Problemquelle sind auf mehrere Artefakte aufgeteilte Bibliotheken, die sich ein gemeinsames Package teilen und damit gegen das Split-Package-Verbot verstoßen.

Allerdings dürften diese Probleme nach und nach von selbst verschwinden, wenn Library-Autoren ihre Bibliotheken anpassen oder Alternativen entwickelt werden. So musste bei einem Projekt des Autors die Migration zu Zeiten, als das JDK 9 noch taufrisch war, erfolglos abgebrochen werden, wobei diese später mit JDK 11 doch noch erfolgreich abgeschlossen werden konnte.

In jedem Fall lohnt sich die Migration, denn nicht nur stehen damit moderne Sprachfeatures von Java wie Switch Expressions oder die Deklaration von Variablen mittels „var“ zur Verfügung. Auch der angekündigte JPackager ist ein gewichtiger Grund – lassen sich damit doch wesentlich kompaktere Anwendungsbundles erzeugen, als dies bisher der Fall war. Und auch neuere JavaFX-Versionen und Bibliotheken aus der JavaFX-Community versprechen viele neue Features und Verbesserungen, die ein Update lohnenswert machen.

Referenzen

[1] *https://mail.openjdk.java.net/pipermail/openjfx-dev/*
[2] *https://github.com/openjfx/javafx-gradle-plugin*
[3] *https://github.com/openjfx/javafx-gradle-plugin/issues/11*
[4] *https://mail.openjdk.java.net/pipermail/openjfx-dev/2018-June/021980.html*
[5] *https://openjdk.java.net/jeps/343*
[6] *https://mail.openjdk.java.net/pipermail/openjfx-dev/2018-September/022500.html*

Manuel Mauky
manuel.mauky@zeiss.com
Carl Zeiss Digital Innovation

Manuel Mauky ist Software-Entwickler bei der Carl Zeiss Digital Innovation und beschäftigt sich vor allem mit Frontend-Entwicklung. Dabei setzt er am liebsten TypeScript und React ein, hat aber auch mit JavaFX lange Jahre gearbeitet. Daneben interessiert er sich für funktionale Programmierung und neue Programmiersprachen. Er ist regelmäßiger Sprecher auf Konferenzen und in User Groups und leitet die Java User Group in Görlitz.

„Entwicklern und Architekten erwächst in der Rolle der technologischen Pioniere eine besondere Verantwortung."

Christian Luda, DOAG

Dr. Carola Lilienthal ist Geschäftsführerin der WPS (Workplace Solutions GmbH) und dort für den Bereich Softwarearchitektur zuständig. Sie analysiert deutschlandweit Architekturen in Java, C#, C++, ABAP sowie PHP und berät Entwicklungsteams, wie sie die Langlebigkeit ihrer Softwaresysteme optimieren können.

Frau Dr. Lilienthal, wie sehr haben Entwickler und Softwarearchitekten unsere Welt bereits verändert und wie groß sind aus Ihrer Sicht die Veränderungen, die noch bevorstehen?

Dr. Carola Lilienthal: In nur einem halben Jahrhundert haben wir – die Softwareentwickler*innen und Architekt*innen – die Welt auf den Kopf gestellt. Etwas bescheidener formuliert geht es darum, dass wir eben nicht nur Technologien, sondern dabei auch innovative Methoden und Verfahren erschaffen haben, die die Menschen in der Business-Welt wie Projektleiter*innen, Anwender*innen, Betriebs- und Qualitätsmanager*innen inspiriert und profitieren lässt. Wir werden unsere Welt weiter verändern: Software wird immer mehr die Funktionen übernehmen, die heute noch in Hardware eingebaut sind. Die Digitalisierung wird weiter voranschreiten, und ich glaube und hoffe, dass die Notwendigkeit für arbeitsbedingtes Reisen etwa zu Kunden durch die aktuelle Erfahrung des virtuellen Zusammenseins weiter abnimmt. Meine große Hoffnung ruht in diesem Punkt auf Hologrammen.

Inwieweit haben Entwickler und Architekten aufgrund des von Ihnen beschriebenen Einflusses eine besondere Verantwortung?

Dr. Carola Lilienthal: Man sollte als Softwareentwicklerin und Architektin den eigenen Einfluss zwar nicht überbewerten. Aber uns Entwickler*innen und Architekt*innen erwächst in der Rolle der technologischen Pionier*innen durchaus eine besondere Verantwortung. Wir schaffen täglich neue Software, die hoffentlich zu etwas Wichtigem gut ist und eine Vielzahl von Nutzer*innen Ihre Arbeit erleichtert. Wenn wir alle den Aspekt der Nutzerorientierung umfassend im Sinne der Gesamtgesellschaft begreifen, sind wir auf einem guten Weg, unsere Verantwortung zu übernehmen.

Welche sind die wichtigsten Voraussetzungen, um langlebige Softwarearchitekturen zu schaffen?

Dr. Carola Lilienthal: Dazu muss man „nur" einfache und schuldenfreie Architekturen entwickeln. Eine langlebige Architektur kann für lange Zeit von Generationen von Entwickler*innen und Architekt*innen verstanden und verändert werden, ohne dass der Aufwand für die Änderungen mit der Zeit signifikant zunimmt. Das ist aber nur möglich, wenn die Architektur einfach zu verstehen ist. Das gelingt, wenn Entwickler*innen und Architekt*innen den Sourcecode und die ihm zugrunde liegenden Strukturen schnell verstehen können. Voraussetzung dafür sind Modularität, Schichtung und Musterkonsistenz. Also ein Aufbau, bei dem man einzelne Teile – Module – lokal verstehen kann, es ein Oben und Unten gibt, und wiederkehrende Muster – Entwurfsmuster, Architekturmuster, Programmiermuster – immer wieder gleichartig eingesetzt werden. In meinem Buch „Langlebige Softwarearchitekturen" behandle ich dieses Thema ausführlich.

Was sind aus Ihrer Sicht als Architektur-Expertin die großen Vorzüge von Java? Wo sehen Sie Schwächen?

Dr. Carola Lilienthal: Ich finde Java eine großartige Sprache, mit der man sehr gut Objektorientierung lernen kann. Weiter möchte ich mich in den Krieg der Programmiersprachen, ehrlich gesagt, nicht begeben. Jede Sprache hat ihre Vor- und Nachteile. Je nach Anforderungen und Möglichkeiten sollte man die passende auswählen.

Welche Chancen birgt die aktuelle Corona-Krise, insbesondere für die IT-Branche? Gibt es bereits eine gestiegene Nachfrage an Digitalisierungslösungen zu beobachten?

Dr. Carola Lilienthal: Alle Arten von Softwarelösungen für virtuelle Konferenzen und Schulungen werden aktuell selbstverständlich vermehrt nachgefragt. Wir haben die Chance beim Schopfe gepackt und unsere Schulungen und Trainings auf Remote umgestellt. Meine Kolleg*innen aus dem Trainingsbereich haben sich einiges einfallen lassen, wie wir unser Schulungskonzept verändern und welche Software wir einsetzen müssen, damit die Schulungen für Architekten und zu DDD virtuell durchgeführt werden können. Und was soll ich sagen: Es funktioniert sehr gut!

Frau Dr. Lilienthal, vielen Dank für das Gespräch.

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen

iJUG Verbund

www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Stefan Kinnen. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Mylène Diacquenod
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: Bild © AndSus | *https://stock.adobe.com*
S. 10: Bild © Sean Gladwell | *https://stock.adobe.com*
S. 14: Bild © Tetiana Lazunova | *https://de.123rf.com*
S. 16: Bild © Freepik | *https://de.freepik.com*
S. 20: Bild © Welf Boris | *https://de.123rf.com*
S. 22: Bild © scyther5 | *https://de.123rf.com*
S. 30: Bild © Jess Sanz | *https://de.123rf.com*
S. 35: Bild © Odoroaga Monica | *https://de.123rf.com*
S. 39: Bild © yupiramos | *https://de.123rf.com*
S. 44: Bild © dolgachov | *https://de.123rf.com*
S. 51: Bild © iunewind | *https://de.123rf.com*
S. 55: Bild © Ioulia Bolchakova | *https://de.123rf.com*
S. 57: Bild © rawpixel | *https://de.freepik.com*
S. 60: Bild © Andrii Torianyk | *https://de.freepik.com*

Anzeigen:
Simone Fischer, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

Mediadaten und Preise unter:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH,
www.wir-machen-druck.de

Inserentenverzeichnis

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu

Eventpartner: AOUG SOUG swiss oracle user group iJUG Verbund ORACLE