# w-jax®

## 4. bis 8. November 2019

**Expo: 5. bis 7. November 2019**

München

# Java-Dossier für Software-Architekten 2019

**Mehr als 30 Seiten Wissen zu:**

Java, Microservices, Architektur und weiteren Themen

jaxcon  jax.konferenz  videosjaxenter  jax

**jax.de**

# Inhalt


## Web

**EnterpriseTales** 3
von Sven Kölpin

## Microservices

**Java Microservices: Ab in die Cloud** 8
von Arne Limburg und Lars Röwekamp

## Architektur

**„Die Effekte der Verteilung werden sehr oft stark unterschätzt“** 15
Interview mit Uwe Friedrichsen

**Req4Arc** 19
von Dr. Peter Hruschka und Dr. Gernot Starke

## Women in Tech

**„Was wir brauchen, sind Vorbilder“** 22
Interview mit Anna-Maria Schaum

## DevOps

**DevOps Stories** 24
von Konstantin Diener

## JavaScript

**Die Angular-Abenteuer** 27
von Manfred Steyer

## Java Core

**Warum Werttypen wichtig sind** 32
von Peter Verhas

Web Components

# EnterpriseTales

Im Web haben sich komponentenbasierte UI-Frameworks durchgesetzt. Alle heute relevanten Single Page Application Frameworks setzen vollständig oder mindestens zu großen Teilen auf dieses Paradigma. Mit Web Components existiert ein vielversprechender Standard, der die Erstellung Framework-unabhängiger Komponenten ermöglicht und so bereits heute für mehr Wiederverwendbarkeit im Web sorgen kann.

von Sven Kölpin

Mit Hilfe von Komponenten lassen sich Benutzungsoberflächen aus konfigurierbaren, wiederverwendbaren Bausteinen (auch Widgets) zusammensetzen. Den Grundstein für die Erstellung eigener Komponenten im Web legten einst Bibliotheken wie jQuery/jQuery UI [1]. Diese ermöglichen es, herkömmliche HTML-Elemente in Bedienelemente mit einem bestimmten Aussehen, Verhalten und Zustand zu verwandeln. Heute ist die Entwicklung von Webapplikationen ohne den Einsatz komponentenbasierter Frameworks wie Angular oder React schon gar nicht mehr denkbar.

Komponenten sind im Web traditionell Framework-spezifisch. Die APIs und sogar die Definition davon, was eine Komponente eigentlich im Konkreten ausmacht, unterscheiden sich zwischen den Frameworks. Deshalb ist es zum Beispiel nicht möglich, ein in Angular geschriebenes Bedienelement direkt in React zu verwenden (oder eben andersherum). Selbst grundlegende Widgets (z. B. Dialoge) müssen somit oftmals für verschiedene Webanwendungen neu entwickelt werden. Diese Herausforderung gewinnt im Zeitalter von Microservices und Microfrontends, in dem es durchaus vorkommen kann, dass verschiedene Frameworks parallel eingesetzt werden, zunehmend an Bedeutung. Genau hier können Web Components Abhilfe schaffen.

## Der Web-Components-Standard

Streng genommen gibt es den einen Web-Components-Standard gar nicht. Vielmehr verbergen sich hinter Web Components eine Menge APIs, deren Kombination die Erstellung von UI-Komponenten für die Webplattform ermöglicht. Die Spezifikation schafft eine einheitliche und somit Framework-unabhängige Definition von webbasierten Komponenten und legt den Grundstein für wiederverwendbare UI-Widgets im Web.

Web Components sind keineswegs eine neue Erfindung. Zum Beispiel gibt es bereits seit 2014 eine Implementierung im Chrome-Browser. Andere Browserhersteller haben frühe Versionen der Spezifikation aber größtenteils vernachlässigt, sodass eine plattformübergreifende Verwendung von Web Components lange ausschließlich mit Hilfe von Polyfills [2] und Frameworks wie Polymer [3] möglich war. Mittlerweile wurde der Standard, der jetzt den Versionsnamen V1 trägt, modernisiert und genießt eine breite Browserunterstützung (**Abb. 1**). Web Components können also heute in fast allen Browsern nativ verwendet werden. Aber auch Webbrowser ohne vollständige Unterstützung sind durch die bereits erwähnten Polyfills nicht ausgeschlossen. Allerdings leiden hier selbstverständlich die Performanz und Netzwerklast durch den zusätzlich benötigten JavaScript-Code.

## Die Bestandteile von Web Components

Die Web-Component-Spezifikation basiert auf insgesamt vier Standards: Custom Elements, HTML-Templates, Shadow DOM und ES Modules. Jedes dieser einzelnen APIs wird, wie für die Webplattform typisch, über JavaScript angesprochen. Zudem sind die Spezifikationen alleinstehend, lassen sich also theoretisch auch unabhängig voneinander verwenden.

Web Components können mit Hilfe aller oder aus der Kombination von bestimmten APIs erstellt werden. Es ist zum Beispiel nicht zwingend notwendig, dass eine Komponente die *Shadow DOM*- oder *HTML Template*-Spezifikation verwendet. Um das volle Potenzial auszuschöpfen, ist aber in den meisten Fällen der Einsatz aller APIs zu empfehlen.

Im Folgenden werden die einzelnen Bestandteile von Web Components genauer beleuchtet.

## Custom Elements

Der HTML-Standard liefert seit jeher Komponenten für die Erstellung von Benutzungsoberflächen. Diese häufig eher primitiven Elemente (z. B. Buttons, Eingabefelder oder Datumsfelder) reichen aber in den meisten Anwendungen nicht aus. Bislang musste man deshalb für aufwendige Bedienelemente auf nicht standardisierte Bibliotheken und Frameworks zurückgreifen. Das Custom Elements API hilft hier weiter. Mit diesem Standard lassen sich eigene HTML-Elemente (also Tags) definieren und mit Verhalten


Abb. 1: Browser Support Web Components [4]

versehen. Die Custom-Elements-Spezifikation bildet die Grundlage für die Erstellung von Web Components.

In der Regel erben Custom Elements vom *HTMLElement*, das die Basis für alle im HTML-Standard definierten Komponenten ist. So wird unter anderem der Zugriff auf grundlegende Eigenschaften und Methoden (z. B. *addEventListener*, *style*, ...) eines HTML-Elements ermöglicht. Aktuell kann nicht von konkreten nativen Komponenten abgeleitet werden (z. B. *extends HTMLButtonElement*). Hierzu gibt es einen gesonderten Spezifikationsvorschlag mit dem Namen *customized built-in elements*, der allerdings bislang nur in Chrome implementiert ist und von Safari abgelehnt wurde.

Das Registrieren von eigenen Tags erfolgt über das *customElements.define*-Interface. Der Name einer jeden erstellten Komponente muss mindestens einen Bindestrich beinhalten. Dadurch können Konflikte mit nativen HTML-Tags vermieden werden. Zudem lassen sich Web Components so stets am Namen des Tags erkennen. Das doppelte Registrieren eines Element-Namens innerhalb der gleichen Seite ist nicht möglich und führt zu einem Fehler. Listing 1 zeigt die Definition einer einfachen Komponente.

Ein Custom Element definiert verschiedene Lebenszyklusmethoden. Der *adoptedCallback* wird im Rahmen dieses Artikels nicht betrachtet. Neben dem *constructor*, der für jede Instanz eines Custom Elements (für jedes Tag) aufgerufen wird, gibt es noch die Methoden *connectedCallback*, *disconnectedCallback* und *attributeChangedCallback*.

Die *connected*- und *disconnected*-Methoden werden aufgerufen, wenn die jeweilige Komponente in den DOM-Baum eingefügt beziehungsweise wieder gelöscht wird. Sie können beispielsweise dazu verwendet werden, das Rendering zu initiieren, Daten vom Server abzufragen oder Event Listener zu registrieren oder freizugeben (z. B. WebSocket-Verbindungen).

Das *attributeChangedCallback* wird aufgerufen, wenn sich Attribute verändern, die vorher explizit als beobachtbar gekennzeichnet wurden. Dieses *callback* reagiert also auf Statusänderungen oder Änderungen der Konfiguration einer Komponente. Wie für HTML-Elemente üblich, können Attribute nur als String-basierte Werte übermittelt werden. Ein Transfer von komplexen JavaScript-Objekten oder Funktionen direkt aus dem HTML heraus ist also nicht möglich. Einzige Ausnahme ist hier die Definition von Event Listeners wie *onclick*, *onblur* etc.

Um solche Datenstrukturen trotzdem an Custom Elements zu übergeben, müssen (wie bei allen HTML-Elementen) Properties oder Methoden verwendet werden.

Die Custom-Element-Spezifikation ist bereits sehr mächtig und liefert noch viele weitere interessante APIs.

Für eine tiefgehende Einführung eignet sich der Google-Developer-Beitrag unter [5].

## HTML-Templates

Das HTML-Template-API erlaubt die Erstellung von wiederverwendbaren HTML-Fragmenten, die unter anderem zum Rendern von Custom Elements nutzbar sind. Anders als der Name es vermuten lässt, verbirgt sich hinter der Spezifikation leider keine browsereigene Templating Engine mit Konstrukten wie *if*-Bedingungen oder Schleifen – zu diesem Thema wurde aber bereits ein API-Vorschlag eingereicht [6]. Zum jetzigen Zeitpunkt sind HTML-Templates vor allem dabei hilfreich, ein effizientes Rendering von Custom Elements zu erreichen. Browser-Engines können die Templatefragmente nämlich sehr effizient verarbeiten, sodass deren Verwendung eine performante Alternative zum String-basierten Rendering bietet, wie es noch in Listing 1 dargestellt ist. Ein Template kann alles beinhalten, was über die *innerHTML*-Eigenschaft [7] gesetzt werden kann, dazu zählen zum Beispiel auch *<style/>*-Tags oder *<link/>*-Elemente. Listing 2 zeigt ein Beispiel für die Verwendung von HTML-Templates in Custom Elements.

### Listing 1: Einfache Web Component

```
<body>
<!-- usage of the web component -->
<hello-component name="r10"></hello-component>

<script>
  class HelloComponent extends HTMLElement {
    static get observedAttributes() {
      //observe the name attribute
      return ['name'];
    }

    constructor() {
      //called for every hello-component tag
      super();
    }

    connectedCallback() {
      //called when the component is attached to the dom
      this.render();
    }

    disconnectedCallback() {
      //called when component gets removed from the dom
    }

    attributeChangedCallback(name, oldVal, newVal) {
      //called initially and whenever the name attribute changes
      if (name === 'name' && oldVal && oldVal !== newVal) {
        this.render();
      }
    }

    render() {
      const name = this.getAttribute('name');
      this.innerHTML = `
        <h1>Hello ${name}.</h1>
      `;
    }
  }

  //register the hello-component
  customElements.define('hello-component', HelloComponent);

  //simulate attribute change after 1 second
  setTimeout(() => {
    document
      .querySelector('hello-component')
      .setAttribute('name', 'r20');
  }, 1000);
</script>
</body>
```

## Shadow DOM

Die Shadow-DOM-(SD-)Spezifikation liefert, neben den Custom Elements, das wohl wichtigste API für die Erstellung wiederverwendbarer Komponenten. Mit Hilfe dieses Standards können gesonderte DOM-Strukturen erstellt werden, die isoliert von der Außenwelt sind. Damit lassen sich zum Beispiel der externe Zugriff auf den Inhalt und das Aussehen einer Web Component unterbinden.

### Listing 2: HTML-Template

```
<body>
<hello-component name="r10"></hello-component>

<script>
  //definition of a reusable template
  const template = document.createElement('template');
  template.innerHTML = `
    <style>
      p {
        background: rebeccapurple;
        color: white;
      }
    </style>
    <p>Hello to <span></span></p>
  `;

  class HelloComponent extends HTMLElement {
    connectedCallback() {
      //clone the template's content
      this.appendChild(template.content.cloneNode(true));
      //set the name
      this.querySelector("p span").innerHTML = this.getAttribute('name');
    }
  }

  //register the hello-component
  customElements.define('hello-component', HelloComponent);
</script>
</body>
```

Vor allem beim Styling von Komponenten bietet das Shadow DOM viele Vorteile. Alle innerhalb des SD definierten CSS-Regeln gelten nämlich nur für die Komponente. Regeln von außen werden zudem nur dann angewendet, wenn sie nicht explizit im Shadow DOM überschrieben sind. Auch alle im Shadow Tree vergebenen IDs und CSS-Klassennamen haben nur dort ihre Gültigkeit. Das macht die Definition von CSS-Regeln und das Selektieren von DOM-Elementen sehr einfach und zudem effizient. Mit Hilfe von CSS-Properties ist es zusätzlich möglich, das Aussehen bestimmter Bereiche des SD konfigurierbar zu machen und so isolierte, aber trotzdem wiederverwendbare Web Components zu erstellen. Listing 3 zeigt ein Beispiel für die Benutzung des Shadow DOM API.

Ein weiterer wichtiger Bestandteil des Shadow DOM ist das *<slot/>*-API, mit dem ganze DOM-Strukturen oder Web Components von außen in das SD eingefügt werden können. Jeder *<slot/>* kann dabei mit einer bestimmten Rolle versehen werden, was die Komposition verschiedener Web Components und damit eine sehr hohe Wiederverwendbarkeit ermöglicht (Listing 4).

Es ist wichtig darauf hinzuweisen, dass die Shadow-DOM-Spezifikation keineswegs ein Sicherheitsfeature ist. Beispielsweise lässt sich mit Hilfe der browsereigenen Entwicklungswerkzeuge weiterhin der Inhalt des versteckten DOM untersuchen. Auch der programmatische Zugriff auf den Shadow DOM einer Komponente ist möglich, zumindest im *mode: open* [8].

## ES Modules

Der ES-(ECMAScript-)Modules-Standard ist, im Gegensatz zu den anderen APIs, nicht im Rahmen der Web-Component-Spezifikation entstanden. Das API ist Teil von ECMAScript 2015 und bietet unter anderem die Möglichkeit, JavaScript-Module zu definieren, die das (dynamische) Importieren und Exportieren von Funktionen und Datenstrukturen aus JavaScript-Dateien heraus erlauben. ES Modules werden mittlerweile von allen gängigen Browsern unterstützt.

Für den Web-Component-Standard stellen ECMAScript Modules die Grundlage für das Zusammensetzen von Benutzungsoberflächen aus unterschiedlichen Komponenten dar. Mit Hilfe der Spezifikation können Bedienelemente in verschiedenen JavaScript-Dateien definiert und Abhängigkeiten formuliert werden.

In der veralteten V0-Version der Web-Component-Spezifikation sollten Komponenten noch mit Hilfe des

### Listing 3: Shadow DOM

```
<body>
<hello-component name="r10"></hello-component>

<script>
  const template = document.createElement('template');
  template.innerHTML = `
    <style>
      /* the style will only be used within the shadow dom */
      p {
        background: rebeccapurple;
        color: white;
      }
    </style>
  <p>Hello to <span id="content"></span></p>
  `;

  class HelloComponent extends HTMLElement {
    constructor() {
      super();
      //create shadow dom (open mode)
      this.attachShadow({mode: 'open'})
    }

    connectedCallback() {
      //clone template into shadow dom
      this.shadowRoot.appendChild(template.content.cloneNode(true));
      //id's can be used without causing conflicts
      this.shadowRoot
          .querySelector('#content')
          .innerText = this.getAttribute('name');
    }
  }

  //register the hello-component
  customElements.define('hello-component', HelloComponent);
</script>
</body>
```

### Listing 4: Slots

```
<body>
<hello-component>
  <!-- add child element as slot -->
  <h1 slot="title">R10</h1>
</hello-component>

<script>
  const template = document.createElement('template');
  template.innerHTML = `
    <p>Hello to <slot name="title"></slot></p>
  `;

  class HelloComponent extends HTMLElement {
    constructor() {
      super();
      this.attachShadow({mode: 'open'})
    }

    connectedCallback() {
      //clone template into shadow dom. Slot will be rendered automatically
      this.shadowRoot.appendChild(template.content.cloneNode(true));
    }
  }

  //register the hello-component
  customElements.define('hello-component', HelloComponent);
</script>
</body>
```

HTML Import APIs modularisiert werden. Dieses API ist aber mittlerweile als obsolet gekennzeichnet. Vor allem in älteren Beiträgen wird dennoch von HTML Imports anstatt von ES Modules geredet, was leicht zu Missverständnissen führen kann.

## Anwendungsfälle für Web Components

Der hauptsächliche Anwendungsfall für Web Components liegt darin, Framework-unabhängige UI-Komponenten zu erstellen. Durch die breite Browserunterstützung ist das bereits heute möglich. Auch die Integration nativer Web Components in die verschiedenen Single Page Application (SPA) Frameworks ist mittlerweile sehr gut. So können sowohl in Angular als auch in Vue.js Web Components ohne Einschränkungen verwendet werden. Die Unterstützung in React ist aktuell noch verbesserungswürdig [9]. Eine genaue Zusammenstellung über den aktuellen Status der Web-Component-Unterstützung verschiedener Frameworks lässt sich unter [10] nachlesen.

Weiterhin ist es durchaus möglich, ganze Webanwendungen ausschließlich auf Basis von Web Components zu entwickeln, also komplett auf externe Bibliotheken und SPA Frameworks zu verzichten. Dieser Framework-lose Ansatz ist, zumindest in der Theorie, durchaus attraktiv. Schließlich muss man sich nicht mehr an etwaige Drittanbieter binden, sondern lediglich einen offiziellen Standard nutzen. Leider sind die APIs der Web-Component-Spezifikation aber vergleichsweise rudimentär. Deshalb muss in der Regel mehr Code geschrieben werden als bei den gängigen SPA Frameworks. Abhilfe können hier leichtgewichtige Bibliotheken wie lit-html [11] schaffen, die aber natürlich wieder eine gewisse Abhängigkeit zur Folge haben.

## Fazit

Web Components sind mittlerweile eine breit unterstützte Technologie. Der Standard hilft dabei, typische Probleme bei der Entwicklung webbasierter Applikationen zu umgehen. Dazu zählt vor allem die Framework-übergreifende Wiederverwendbarkeit von Komponenten. Selbstverständlich bedeutet das aber nicht, dass beliebige Teile einer Anwendung automatisch austauschbar sind. In erster Linie eignet sich die Spezifikation für die Umsetzung technisch getriebener Komponenten (z. B. spezielle Buttonelemente oder modale Dialoge).

Vor allem das Custom Element API ist an vielen Stellen sehr rudimentär, sodass es für die Entwicklung von eigenen Web Components hilfreich sein kann, auf externe Bibliotheken zu setzen. Hier bieten zum Beispiel die Frameworks Stencil [12] und Skate [13] vielversprechende Ansätze. Mit ihnen lassen sich native und damit austauschbare Web Components erstellen, ohne die oftmals umständlichen APIs direkt zu verwenden.

Web Components werden vor allem von Google vorangetrieben, das mit Chrome einen der am meisten verbreiteten Browser am Markt besitzt. Es lohnt sich also definitiv, einen näheren Blick auf die Spezifikation zu werfen.

In diesem Sinne: Stay tuned.



**Sven Kölpin** ist Enterprise-Entwickler, Speaker und Autor bei der open knowledge GmbH in Oldenburg. Schwerpunkt und Leidenschaft ist die Konzeption und Entwicklung von Webanwendungen.

## Links & Literatur

[1] https://jqueryui.com

[2] https://de.wikipedia.org/wiki/Polyfill

[3] https://www.polymer-project.org

[4] https://www.webcomponents.org

[5] https://developers.google.com/web/fundamentals/web-components/customelements

[6] https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Template-Instantiation.md

[7] https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML

[8] https://medium.com/@emilio_martinez/shadow-dom-open-vs-closed-1a8cf286088a

[9] https://custom-elements-everywhere.com/libraries/react/results/results.html

[10] https://custom-elements-everywhere.com

[11] https://lit-html.polymer-project.org

[12] https://stenciljs.com

[13] https://github.com/skatejs/skatejs/

Hochdynamische Anwendungen mit Containern und Service Meshes

# Java Microservices: Ab in die Cloud

Microservices-basierte Anwendungen in Java zu realisieren, ist dank Spring Boot und Eclipse MicroProfile denkbar einfach. Dies gilt sowohl für die Implementierung der Services selbst als auch für die grundlegende Unterstützung notwendiger Querschnittsdienste wie Logging, Tracing und Security. Was aber, wenn zusätzliche Flexibilität bezüglich Skalierung und Plattform gefragt ist? Bevor es mit der Anwendung in die Cloud geht, müssen zunächst noch ein paar Hausaufgaben erledigt werden.

von Arne Limburg und Lars Röwekamp

In den Artikeln „Vom Java-EE-Monolithen zu Microservices“ (S. 26) und „Java Microservices im Praxischeck“ (S. 34) haben wir gezeigt, mit welchen Herausforderungen ein Enterprise-Java-Entwickler beim Umstieg von monolithischen Architekturen hin zu Microservices rechnen muss und wie er diesen mit Hilfe von Spring Boot oder MicroProfile technologisch begegnen kann. In diesem Artikel wollen wir uns der Frage widmen, welche zusätzlichen infrastrukturellen Hilfsmittel benötigt werden, um eine Microservices-basierte Anwendung – beispielsweise in der Cloud – sinnvoll betreiben zu können.

## Flexibilität und Agilität

Einen einzelnen Microservice zu implementieren und alleinstehend zum Laufen zu bekommen, ist heutzutage dank guter Framework-Unterstützung kein wirkliches Problem mehr. Die eigentliche Herausforderung besteht darin, eine fachlich komplexe Anwendung als Netz von Dutzenden von Microservices inklusive der dafür notwendigen Infrastruktur dynamisch aufzubauen und zu betreiben. Steigt die Last, sollen die Services automatisch skalieren. Fallen Service aus oder werden gerade unter geänderter Adresse neu gestartet, sollen potenzielle Consumer davon nichts mitbekommen, sondern automatisch an die neue Adresse geroutet oder mit alternativen Services verbunden werden.

Aber nicht nur die Technologie bringt neue Herausforderungen mit sich, sondern auch der damit einhergehende agile Entwicklungsprozess und die durch ihn ermöglichte Geschwindigkeit von der Anforderung bis hin zur Bereitstellung und Liveschaltung eines neuen fachlichen Features. Langwierig beim Operation-Team zu beantragende Plattformkomponenten sind ebenso tabu wie ressourcen- und zeitfressende, anwendungsweite Test- oder Integration-Stages.

Um dieser Anforderungen Herr zu werden, bedarf es einer leichtgewichtigen Virtualisierung der Anwendungskomponenten, die es den Entwicklern erlaubt, die von ihnen implementierte Software in kurzen Iterationen produktionsnah zu testen und bereitzustellen. Zusätzlich wird ein Mechanismus benötigt, der die virtualisierten Komponenten automatisiert orchestriert, überwacht und bei Bedarf korrigierend eingreift. Und letztendlich gilt es noch, die Inter-Service-Kommunikation in den Griff zu bekommen.

Für die erste Herausforderung – leichtgewichtige Virtualisierung – haben sich in den letzten Jahren Container hervorgetan. Die zweite Herausforderung wird entsprechend von Werkzeugen zur Containerorchestrierung und deren Management übernommen. Als dritte und letzte Infrastrukturkomponente kommen sogenannte Service Meshes ins Spiel, die die Aufgabe eines Infrastrukturlayers für Inter-Service-Kommunikation übernehmen.

## Container, wohin man schaut

Seit 2013 das Open-Source-Projekt Docker veröffentlicht wurde, sind Container nicht mehr wegzudenken.

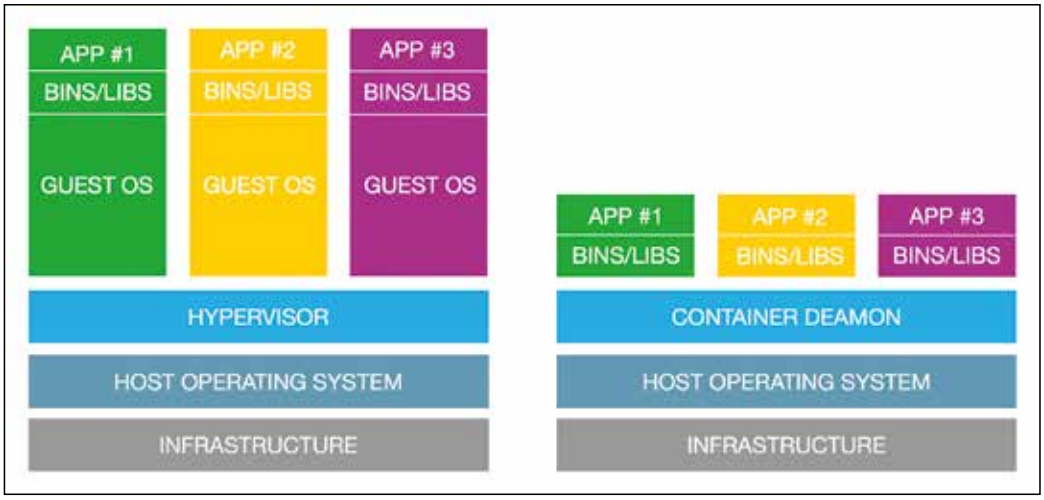


Abb. 1: Virtualisierung vs. Container

Aber warum brauchen wir überhaupt Container? Und welchen Vorteil bringen sie uns speziell im Umfeld von Microservices? Schließlich gibt es doch seit Jahren Virtualisierung.

Zunächst einmal soll die Frage geklärt werden, warum wir überhaupt Virtualisierung im Umfeld von Microservices benötigen. Die Grundidee der Virtualisierung besteht darin, den Anwendungs- und Plattformkomponenten eine identische und stabile Ablaufumgebung von der Entwicklung bis hin zur Produktion zur Verfügung zu stellen und so negative Seiteneffekte bei der Übergabe der Software von einer Stage zur nächsten zu vermeiden. Ohne Virtualisierung müssten entweder alle Stages – inklusive OS, Plattformkomponenten und Zugriffsrechten – identisch aufgebaut sein, was faktisch kaum zu realisieren ist, oder es müsste ein nicht zu unterschätzender manueller Aufwand bei der Migration der Komponenten von einer Stage zur nächsten betrieben werden. Genau das widerspricht aber dem agilen Konzept und macht alle Vorteile der Microservices in Hinblick auf eine schnelle Umsetzung und Bereitstellung neuer Features zunichte.

So weit, so gut. Welchen Vorteil bringen dann aber Container im Vergleich zur klassischen Virtualisierung mit sich? Bei der Virtualisierung besteht das Gesamtpaket aus einer VM, inklusive Betriebssystem und Anwendung. Ein physikalischer Server mit mehreren virtuellen Maschinen würde also entsprechend aus einem Hypervisor und mehreren darauf aufsetzenden Gastbetriebssystemen bestehen (**Abb. 1**). Diese strikte Trennung der einzelnen VMs inklusive all ihrer Komponenten hat aus Sicht der Sicherheit durchaus ihre Vorteile. Betrachtet man dagegen die benötigten Ressourcen, kommt dieses Modell allerdings relativ schwergewichtig daher und scheint somit für Microservices-basierte Anwendungen mit sehr, sehr vielen zu virtualisierenden Komponenten nicht wirklich gut geeignet zu sein.

Hier kommen nun Container ins Spiel. Im Gegensatz zur klassischen Virtualisierung teilen sich Container den Betriebssystemkernel untereinander lesend. Zusätzlich erhält jeder Container Zugriff auf einen eigenen, schreibenden Bereich. Container benötigen somit kein Gastbetriebssystem und sind daher deutlich leichtgewichtiger. Dies bringt nicht nur zur Laufzeit Vorteile mit sich, sondern hilft auch dabei, den Container einfacher von A nach B zu verschieben.

Ein weiterer Vorteil der Container ergibt sich durch deren schnelle Bootsequenz. Je nach Aufbau stehen die innerhalb der Container beherbergten Microservices binnen Sekunden zur Verfügung, was dem agilen Vorgehen mit seinen sehr kurzen Testiterationen sehr entgegenkommt. VMs dagegen benötigen teilweise Minuten zum Hochfahren.

Container können aus mehreren Layern aufgebaut sein, von denen alle bis auf den obersten immutable sind. Dies hilft dabei, das Risiko einer Manipulation der Container zu minimieren, und kann die Build-Geschwindigkeit deutlich erhöhen, da beim Neubau des Containers lediglich der Writable-Layer ausgetauscht werden muss. Wie immer steckt natürlich auch bei der Verwendung von Containern der Teufel im Detail. Da die Container von der Entwicklung bis zur Produktion zum Einsatz kommen, sollte man sich vor deren Einsatz ausgiebig mit der Materie auseinandersetzen. Zwar gibt es mittlerweile – insbesondere für Docker – sehr viele offene Repositories mit vorgefertigten Containern, diese entsprechen aber nicht immer den eigenen (Sicherheits-)Anforderungen.

Bedeutet das nun, dass jeder Entwickler auch automatisch zum Container- bzw. Docker-Spezialisten ausgebildet werden muss? Man benötigt im Team auf jeden Fall Experten, die in der Lage sind, Container zu bauen. Und zwar so, dass diese der selbstauferlegten oder unternehmensweit vorgegebenen Policy bezüglich Sicherheit, Ressourcenverbrauch, Performance etc. entsprechen. Mit dem Bauen allein ist es aber noch lange nicht getan. Die Container wollen auch betrieben, skaliert und gemanagt werden. Womit wir beim nächsten Thema wären: Containermanagementplattformen.

## Wider das Chaos: Containermanagementplattformen

Auf Umgebungen, in denen lediglich einige wenige Anwendungen laufen, die wiederum nur aus vereinzelten

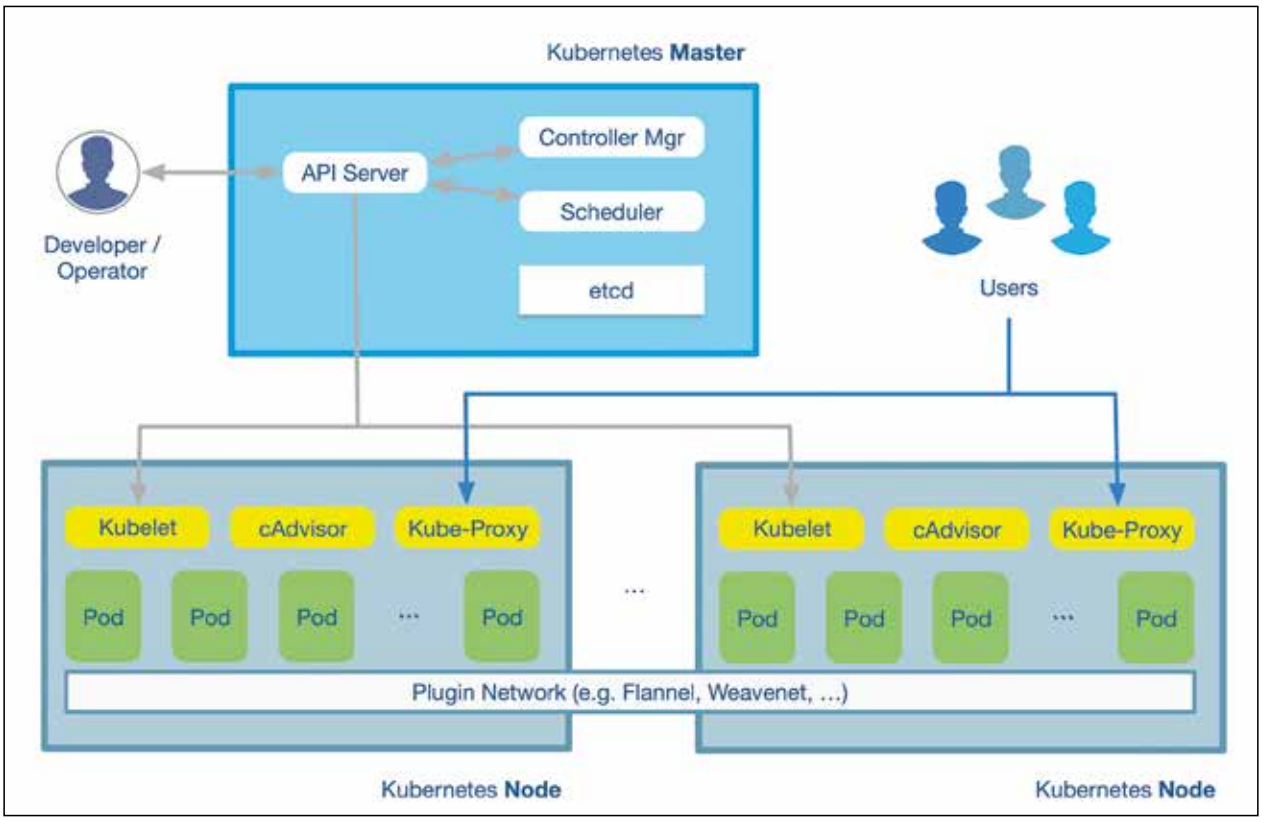


Abb. 2: Kubernetes-Architektur Quelle: https://de.wikipedia.org/wiki/Kubernetes#/media/File:Kubernetes.png

Containern bestehen, ist das Management der Container noch manuell zu bewerkstelligen. Anders dagegen sieht es aus, wenn, wie in der Praxis durchaus üblich, mehrere hundert Services mehrfach skaliert in Tausenden von Container laufen sollen. Derartige Szenarien lassen sich manuell nicht mehr beherrschen. Automatisierung von Aufgaben wie Deployment, Management, Skalierung, Networking und Verfügbarkeit ist nicht mehr nur Kür, sondern wird zur Pflicht. Hier kommen Containermanagementplattformen ins Spiel. Zu deren wesentlichen Aufgaben zählen:

- Provisionierung und Deployment von Containern
- Redundanz und Verfügbarkeit von Containern sicherstellen
- Aufrufen/Entfernen von Containern zur Lastverteilung über verteilte Hostinfrastruktur
- Verschieben von Containern bei Problemen auf einem Host
- Ressourcenallokation zwischen Containern
- öffentliche Bereitstellung von Services, die innerhalb der Container laufen
- Service Registry und Service Discovery inklusive Load Balancing
- Health-Monitoring von Containern und Hosts
- Konfiguration von Anwendungen bezüglich der benötigten Container

Die Orchestrierung der Anwendungen erfolgt typischerweise mit Hilfe eines YAML oder JSON Files, das angibt, welche Container benötigt werden, wo sie zu finden sind (zum Beispiel in einem internen Repository oder auf Docker Hub), wie die Kommunikation zwischen den Containern aussehen soll, welche Storage-Systeme gemountet und wo die Logs der Container abgelegt werden sollen. Teams branchen und versionieren diese Konfigurationen, sodass die Anwendungen auf unterschiedlichen Entwicklungs- und Testumgebungen deployt werden können, bevor sie in Produktion gehen.

Für die Komposition einiger weniger Docker-Container reicht die Verwendung von Docker Compose. Wächst die Anwendung und damit auch der Anspruch an Automatisierung, wird ein Wechsel zu einer echten Containermanagementplattform notwendig. Docker selbst bietet mit Docker Swarm eine hausinterne Lösung an. Der Platzhirsch unter den Containermanagementplattformen ist – mit einer entsprechend breiten Unterstützung in der Cloud – allerdings Kubernetes.

Ursprünglich von Google als Seitenprojekt des Project Borg ins Leben gerufen, ist Kubernetes mittlerweile der De-facto-Standard für Docker-Orchestrierung. Als **das** Vorzeigeprojekt der Cloud Native Computing Foundation wird es von namhaften Größen wie Google, Amazon, Microsoft, IBM, Intel, Cisco und RedHat unterstützt. Mithilfe von Kubernetes lassen sich Containercluster über verschiedene Umgebungen (on-Premise, Public-, Private-, Hybrid-Cloud) hinweg effizient verwalten.

Container werden als Pods – eine Art Basiseinheit – verwaltet, die auf der gleichen physikalischen oder virtuellen Umgebung (aka. Nodes) ausgeführt werden

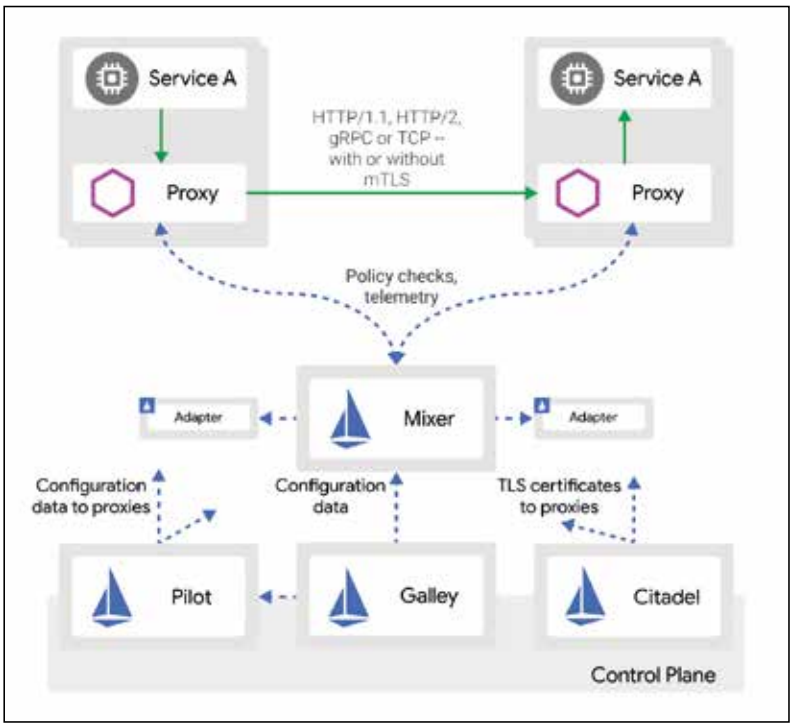


Abb. 3: Istio-Architektur [4]

(**Abb. 2**). Container innerhalb eines Pods teilen sich IP-Adresse, IPC, Hostname und etliche andere Ressourcen, wobei Kubernetes für alle Pods automatisiert Umgebungen mit ausreichender Kapazität findet und dort die Container des Pods startet. Dank Nodes und Pods erhalten Entwickler bzw. Administratoren eine Abstraktionsebene, die ihnen hilft, Container inklusive ihrer Workloads sowie der von ihnen benötigten Plattformdienste zu gruppieren und als Gruppe zu verwalten.

Das Gute an Containermanagementplattformen wie Docker Swarm oder Kubernetes ist, dass sie in jeder Umgebung eingesetzt werden können, in denen auch Container lauffähig sind. Und das ist heutzutage fast überall – von traditioneller On-Premise-Serverinfrastruktur bis hin zur Public-Cloud. Dass Containermanagementplattformen einen ernstzunehmenden Markt darstellen, erkennt man daran, dass mittlerweile nahezu alle großen Cloudanbieter gemanagte Container bzw. Kubernetes-Plattform-Services im Angebot haben. Der Schritt von On-Premise zur Cloud – und gegebenenfalls zurück – ist somit nur ein paar Klicks entfernt.

Dank Container und Containermanagementplattform haben wir die Grundlagen geschaffen, auf Microservices basierende, dynamische Anwendungsumgebungen einfach und effizient auf beliebigen Umgebungen – von On-Premise bis Public-Cloud – bereitzustellen. Aber sind wir damit schon am Ziel? Was ist mit den im zu Beginn angesprochenen Artikel „Vom Java-EE-Monolithen zu Microservices“ genannten Querschnittsaufgaben wie Tracing, Resilience, Security? Liegen sie auch in der Verantwortung der Containermanagementplattform? Gibt es bessere Alternativen? In der jüngeren Vergangenheit kristallisierten sich Service Meshs als mögliche Lösung heraus.

## Service Mesh

Ein Service Mesh ist ein Infrastrukturlayer für Inter-Service-Kommunikation. Ziel des Einsatzes eines Service Meshs ist es, besagte Kommunikation zuverlässiger, schneller und sicherer zu machen. In der Regel geschieht das dadurch, dass der Service Mesh als Proxy um jeden einzelnen Microservice herum gelegt wird, indem der Service Mesh in Kubernetes als sogenanntes Sidecar mit installiert wird. Jede eingehende und ausgehende Kommunikation erfolgt dann über diesen Proxy. Dadurch können verschiedene Aufgaben, die wir in den beiden vorherigen Artikeln beschrieben haben, durch einen solchen Service Mesh übernommen werden. Dabei kann der Proxy einerseits Informationen über ein- und ausgehende Requests sammeln und an eine zentrale Instanz übermitteln, um so Tracing, Monitoring, Billing und Quota-Aufgaben zu übernehmen. Andererseits kann er direkt auf eingehende Requests reagieren und sie beispielsweise beantworten, ohne dass der Request zum tatsächlichen Microservice durchdringt. So können z. B. ein Throttling oder Securityanforderungen (unabhängig vom Microservice) realisiert werden. Nicht zuletzt kann der Proxy auch ausgehende Requests direkt beantworten, ohne dass sie den Container tatsächlich verlassen. Auf diese Weise kann der Service Mesh Caching- und Resilience-Aufgaben übernehmen.

Ein Beispiel für einen solchen Service Mesh ist Istio [1]. Wie die gerade genannten Themen von Istio übernommen werden können und welche Auswirkungen das auf die Microservices hat, wollen wir hier beschreiben.

## Istio

Istio besteht aus mehreren Komponenten. Als Proxy wird in jedem Microservice eine erweiterte Version des Open-Source-Proxys Envoy verwendet [2]. Diese wird, wie bereits oben erwähnt, in Kubernetes als Sidecar deployt. Das heißt, dass neben dem Applikationscontainer im selben Pod ein weiterer Container deployt wird, der den Proxy enthält [3] („Pods that run multiple containers that need to work together“).

Die zentrale Komponente Istio Pilot dient dann den Proxies als Service Discovery und versorgt sie mit Konfigurationen zu Routing, Security, Traffic Management und Resilience.

Zur Laufzeit kommuniziert der Envoy Proxy mit dem Istio Mixer. Dieser ist einerseits dafür verantwortlich, Informationen zu Monitoring-, Tracing-, Auditing-, Billing- und weiteren Zwecken pro Request zu sammeln. Andererseits übermittelt er auch Informationen an den

Envoy Proxy. Auf diese Weise können Authorization-, Quota- und Throttling-Anforderungen realisiert werden.

Darüber hinaus bietet Istio die Infrastruktur zur Realisierung von Securityanforderungen wie Transport Layer Security (TLS), Client-Authentication (mit JWT) und Role-based Access Control. Die dazu benötigte Zertifikatsinfrastruktur wird von der Komponente Citadel bereitgestellt. **Abbildung 3** veranschaulicht die beschriebenen Zusammenhänge.

Der Service Mesh hat also die Aufgabe, wiederkehrende Themen aus der Implementierung eines einzelnen Microservice herauszulösen und unabhängig davon einmalig zu realisieren. Aber welche Aufgaben kann ein Service Mesh tatsächlich übernehmen? Das wollen wir in den nächsten Abschnitten am Beispiel Istio im Detail betrachten.

## Authentication, Authorization und Access Control

Wenn wir mit einem Server kommunizieren, möchten wir nicht, dass jemand den Verkehr mitliest. Deshalb wird in der Regel verschlüsselte Kommunikation verwendet. Gleichzeitig wollen wir sicher sein, dass es sich bei der Gegenstelle tatsächlich um den Server handelt, den wir erwarten. Wir überprüfen sein Zertifikat daher gegen eine sogenannte Trusted Authority. Eine vertrauenswürdige Stelle versichert uns, dass es sich tatsächlich um den richtigen Server handelt. Diese Kommunikationsmechanismen sind etabliert und werden unter Transport Layer Security (TLS) zusammengefasst. Es ist der Mechanismus, der im Internet verwendet wird, sobald die aufgerufene Seite das HTTPS-Protokoll verwendet.

In einer Microservices-Architektur möchten wir aber einen Schritt weiter gehen. Nicht nur der Client möchte sicher sein, dass es sich um den richtigen Server handelt, sondern auch der Server möchte gegebenenfalls wissen, mit welchem Client er kommuniziert. Bei diesem Mechanismus handelt es sich um einen Spezialfall von TLS, nämlich der Mutual TLS. Das heißt, dass der Server auch die Identität des Clients verifiziert und ihn so authentifiziert.

Istio kann so konfiguriert werden, dass bei Inter-Service-Kommunikation Mutual TLS entweder optional oder verpflichtend ist. Über die so gestaltete automatisierte Clientauthentifizierung kann dann im zweiten Schritt eine Autorisierung stattfinden und über Policies festgelegt werden, welcher Server mit welchem kommunizieren darf.

Zusätzlich zu dieser Service to Service Authorization bietet Istio auch Endbenutzer-Authentication mit JWT an. Dabei lässt sich konfigurieren, für welche Aufrufe ein JWT benötigt wird, und natürlich auch, wer das JWT ausgestellt haben muss. Calls, die den Aufruf ohne JWT durchführen, kommen gar nicht erst beim Microservice an, sondern werden direkt von Istio geblockt.

Weitergehend ist in Istio auch eine sehr feingranulare, rollenbasierte Zugriffssteuerung möglich. Pro Resource-Pfad und HTTP-Methode können Rollen angegeben werden, denen ein Zugriff erlaubt ist. Diesen Rollen können dann Benutzer zugeordnet werden.

## Tracing

Im Artikel „Java Microservices im Praxischeck“ haben wir bereits erläutert, dass zum Tracing innerhalb eines Microservice grundsätzlich drei Dinge notwendig sind:

1. Vergabe von Trace- und Span-IDs
2. Weitergabe aller Trace-IDs aus eingehenden Requests an ausgehende Requests
3. Schicken der Informationen an den Tracing-Server (alternativ kann dieser sie sich beim Service abholen)

Zumindest die Punkte 1 und 3 kann Istio bei richtiger Konfiguration übernehmen. Die Microservices selbst müssen dann nur noch dafür sorgen, dass die entsprechenden Header von den eingehenden Requests an die ausgehenden durchgereicht werden [5].

## Resilience

Im Artikel „Java Microservices im Praxischeck“ haben wir verschiedene Resilience-Patterns wie Time-out, Retry, Circuit Breaker, Bulkhead und Fallback kennengelernt. Mit einem Service Mesh wie Istio müssen die meisten dieser Patterns nicht mehr in der Applikation selbst implementiert werden. Stattdessen kann der Service Mesh so konfiguriert werden, dass er die entsprechenden Aufgaben übernimmt. Dass z. B. ein Retry ausgeführt wurde, ist dann für den Microservice komplett transparent.

Es ist sicherlich nicht sinnvoll, die gleiche Konfiguration in jedem Microservice immer wieder zu wiederholen. Von daher ist es sinnvoll, bei den übergreifenden Themen einen Service Mesh in Betracht zu ziehen.

Wenn es um komplexere Resilience-Patterns wie Bulkhead oder Circuit Breaker geht, ist es allerdings so, dass sie in Istio komplett anders funktionieren als in Hystrix [6]. Es lässt sich aber dasselbe (oder zumindest ein ähnliches) Verhalten in Istio konfigurieren, sodass Hystrix praktisch nicht mehr benötigt werden würde. Das einzige Hystrix-Feature, das es in Istio nicht gibt, ist das Fallback. Hier ist es aber auch sinnvoll, dass die Entscheidung über das Fallback in der Applikation und nicht im Sidecar getroffen wird. Schließlich handelt es sich dabei um eine Businessentscheidung.

## Testen von Resilience

Im Allgemeinen ist es sehr kompliziert, das Resilience-Verhalten einer Applikation zu testen. Der einfache Fall, dass ein Nachbar-Service nicht verfügbar ist, ist dabei noch die leichteste Übung. Aber wie teste ich, ob sich mein Service korrekt verhält, wenn der Nachbar-Service langsam ist (und dabei z. B. von Zeit zu Zeit in ein Retry läuft)?

Durch den Envoy Proxy hat Istio alle Mittel an der Hand, um die komplette Bandbreite der Resilience-Patterns zu testen. Und tatsächlich gibt es in Istio auch die Möglichkeit der „Fault Injection“, das heißt, man kann im laufenden Betrieb den Traffic verlangsamen, einen bestimmten Prozentsatz an Requests fehlschlagen, also in ein Time-out laufen, oder bestimmte Fehlercodes zurückgeben lassen. Auf diese Weise kann man testen, ob sich ein Service in solchen Situationen korrekt verhält.

## Traffic Routing

Kernfeature von Istio ist es, den Traffic, der zwischen den Services stattfindet, zu kontrollieren. Dadurch ist es z. B. ein Leichtes, eine Darstellung darüber zu erhalten, wer welchen Service aufruft und wie oft. Zudem gibt es beim Routing mit Istio einige Optionen zur Steuerung des Traffics. So kann beim Routing z. B. nicht nur der aktuelle Zustand der Services berücksichtigt werden, sondern etwa auch der aktuelle Benutzer, der den Service verwenden möchte. Auf diese Weise können bestimmte Service-Versionen z. B. zunächst nur für bestimmte Nutzergruppen freigeschaltet oder es kann eine Mandantenfähigkeit implementiert werden, indem die Benutzer eines Mandanten auf einen anderen Service geroutet werden als die Benutzer eines anderen Mandanten.

Nicht nur durch benutzerbasiertes Routing kann der Livegang eines Service sehr sanft erfolgen. Es gibt außerdem das Feature des Traffic Mirroring. Dabei kann der Traffic, der bisher auf Version 1 des Service ging, vor dem Livegang von Version 2 zunächst für einen bestimmten Zeitraum dupliziert und an Version 1 und 2 geschickt werden. Dabei lässt sich konfigurieren, dass zunächst weiterhin die Antworten von Version 1 zum Aufrufer geschickt werden. So kann zunächst ohne Gefahr überprüft werden, ob sich Version 2 korrekt verhält, bevor dann die Antworten von Version 2 tatsächlich zum Aufrufer geroutet werden. Zu guter Letzt gibt es noch das Feature des Traffic Shiftings. Dabei kann zunächst nur ein Teil des Traffics zur neuen Version geroutet werden, um zu überprüfen, wie sich dieser (z. B. auch in punkto Performance) verhält. Eine Migration kann so schrittweise erfolgen.

## Service Mesh: ein Resümee

Sollte man nun den Schritt hin zum Service Mesh gehen oder nicht? Es ist sicherlich nicht sinnvoll, die gleiche Konfiguration in jedem Microservice immer wieder zu wiederholen. Von daher ist es sinnvoll, bei den übergreifenden Themen einen Service Mesh in Betracht zu ziehen. Das gilt z. B. für die Kommunikation zwischen Microservice und Tracing-Server. Durch den Einsatz von Istio kann diese beispielsweise zentral konfiguriert werden. Dennoch ist gerade das Tracing ein gutes Beispiel dafür, wie sich Istio und Microservices-Bibliotheken (hier OpenTracing bzw. Zipkin) gut ergänzen. Istio ist nicht in der Lage zu erkennen, welcher ausgehende Request zu welchem vorherigen Eingangs-Request gehört. Daher kann Istio auch nicht automatisch die benötigten Header vom Eingangs-Request auf den ausgehenden durchschleifen. Diese Aufgabe kann aber die OpenTracing-Bibliothek innerhalb des Microservice wunderbar übernehmen, sodass der Service-Entwickler sich im Normalfall darum nicht kümmern muss.

Ähnlich verhält es sich beim Thema Resilience. Retry-, Time-out-, Circuit-Breaker- und Bulkhead-Funktionalität kann Istio sehr gut übernehmen. Wenn es aber darum geht, ein Fallback zu definieren, so muss fachlich entschieden werden, was das bedeutet. Eine Realisierung innerhalb des Microservice ist allein deshalb schon sinnvoll. Daher ist es auch nicht weiter schlimm, dass Istio dieses Feature gar nicht anbietet. Ob allein dafür der Einsatz eines Standards wie der des Microprofile-Fault-Tolerance-Standards oder einer Bibliothek wie Hystrix gerechtfertigt ist, muss dann jedes Team selbst entscheiden.

Beim Thema Security hängt viel davon ab, an welchen Stellen Securityentscheidungen innerhalb der Ar-

chitektur getroffen werden. Gibt es eine zentrale Stelle, die entscheidet, welcher Service mit welchem kommunizieren darf? Dann ist es sicherlich sinnvoll, diese auch durch Istio abzusichern. Werden die Benutzer und Rollen zentral verwaltet und wird auch an zentraler Stelle festgelegt, welche Ressource von welcher Rolle mit welcher HTTP-Methode aufgerufen werden darf, dann kann man auch das über Istio konfigurieren. Wenn Teile dieser Entscheidungen beim Service-Team liegen, ist es sinnvoller, die Zugriffssteuerung im Service zu realisieren und die dort vorhandenen Standards und Frameworks zu verwenden. Unabhängig davon ist es aus Securitysicht immer sinnvoll, das Mutual-TLS-Feature von Istio zu verwenden.

Für einige Aufgaben ist ein Service Mesh (wie z. B. Istio) also in jedem Fall sinnvoll. Es gilt allerdings anzumerken, dass nicht alle der hier erwähnten Istio-Features bereits „stable“ sind. Einige befinden sich noch im Zustand Alpha oder Beta. Es ist aber nur eine Frage der Zeit, wann auch sie stable und damit production-ready sind – ein aktueller Stand kann unter [7] eingesehen werden. Wen der Alpha- oder Betastatus einiger Features nicht stört oder wer nur Features benötigt, die bereits stable sind, der sollte bei der Wahl des Service Meshs auf jeden Fall Istio in die engere Auswahl nehmen.

## Fazit

Microservices in Java zu implementieren, ist dank Spring Boot oder Eclipse MicroProfile denkbar einfach. Sie dagegen zu Dutzenden oder gar Hunderten zu betreiben, stellt schon eine größere Herausforderung dar.

Container helfen, Microservices und die von ihnen benötigten Komponenten so zu verpacken, dass sie auf beliebigen Umgebungen automatisiert deployt werden können und dort dann stabil laufen. Nur so kann von der Entwicklung bis hin zur Bereitstellung ein agiles Vorgehen gewährleistet werden.

Bei großen Anwendungen geht die notwendige Automatisierung deutlich über die Bereitstellung einer CI/CD Pipeline hinaus. Containermanagementplattformen übernehmen Aufgaben wie Health-Monitoring, automatisches Recovery, Workload Balancing, Ressourcenallokation und vieles mehr. Mittlerweile bieten alle großen Cloudanbieter gemanagte Containermanagementplattformen (allen voran Managed Kubernetes) an. Der Schritt von On-Premise hin zur Cloud ist so mit überschaubarem Aufwand zu realisieren.

Eine Herausforderung bleibt nach wie vor die Beantwortung der Frage, welche Komponenten für die Verwaltung der Querschnittsaufgaben wie Logging, Tracing und Security verantwortlich zeichnen. Service Meshs können hier dank Proxy einen Großteil der Aufgaben transparent für den Entwickler übernehmen. Natürlich nicht nur in der Cloud, sondern auch on-premise oder in hybriden Umgebungen.

**Arne Limburg** ist Softwarearchitekt bei der open knowledge GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

**Lars Röwekamp** ist Gründer des IT-Beratungs- und Entwicklungsunternehmens open knowledge GmbH und beschäftigt sich im Rahmen seiner Tätigkeit als „CIO New Technologies“ mit der eingehenden Analyse und Bewertung neuer Software- und Technologietrends. Ein besonderer Schwerpunkt seiner Arbeit liegt derzeit in den Bereichen Enterprise und Mobile Computing, wobei neben Design- und Architekturfragen insbesondere die Real-Life-Aspekte im Fokus seiner Betrachtung stehen.



## Links & Literatur

[1] https://istio.io/

[2] https://www.envoyproxy.io/

[3] https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#understanding-pods

[4] Quelle: https://istio.io

[5] https://istio.io/docs/tasks/telemetry/distributed-tracing/#understanding-what-happened

[6] https://blog.christianposta.com/microservices/comparing-envoy-and-istio-circuit-breaking-with-netflix-hystrix/

[7] https://istio.io/about/feature-stages/

Interview mit Uwe Friedrichsen

# „Die Effekte der Verteilung werden sehr oft stark unterschätzt"

Software-Architektur im Jahr 2019 – welche Bedeutung hat sie, welchen Entwicklungen ist sie ausgesetzt, welche Herausforderungen muss sie bewältigen? Antworten gibt Uwe Friedrichsen, Software-Architekt bei codecentric und Sprecher auf dem Microservices Summit 2019, im Interview.

**JAXenter: Die Rolle des Software-Architekten wird immer wieder mal in Frage gestellt. „Sind sie in Zeiten der DevOps-Bewegung noch nötig? Muss nicht jeder Entwickler ein Architekt sein?" Und so weiter…. Deshalb beginnen wir mal mit einem Plädoyer für den Berufsstand: Weshalb sind dezidierte Software-Architekten deiner Meinung nach auch im Jahr 2019 noch sinnvoll? Wo siehst du ganz persönlich deine Hauptaufgabe als Software-Architekt?**

**Uwe Friedrichsen:** Die Aussicht, dass jeder Entwickler die Rolle eines Architekten übernehmen kann, klingt verlockend. Aus meiner Erfahrung ist das aber nicht der Fall. Die Rollen „Entwickler" und „Architekt" verlangen sehr unterschiedliche Skillsets, und aus meiner Erfahrung verfügen nur relativ wenige Entwickler über das Skillset und die Erfahrungen, die man jenseits der reinen Entwicklung für Architekturarbeit benötigt. Das ist aber auch vollständig in Ordnung, denn umgekehrt sieht das häufig ähnlich aus.

Leider wird die Debatte um das Thema immer noch von vielen nicht zielführenden Vorstellungen beherrscht. Das geht von falsch verstandener Agilität über Technologie-Bingo bis hin zu den vollständig sinnlosen Karrieremodellen vieler Unternehmen, in denen der Architekt „über" dem Entwickler steht. Wir brauchen beide Skills in Projekten, keiner ist „besser" oder „wichtiger" als der andere, und nur sehr wenige Personen vereinigen aus meiner Erfahrung beide Skillsets. Entsprechend bin ich der Überzeugung, dass wir auch im Jahr 2019 noch Architekten brauchen – und diese werden aus den zuvor skizzierten Gründen häufig weiterhin dezidierte Personen sein.

Zu deiner zweiten Teilfrage: Aus meiner Sicht besteht Architekturarbeit im Kern aus den folgenden Aktivitäten: Das Problem und den Kontext (über die verschiedenen Stakeholder-Gruppen hinweg) ganzheitlich verstehen, Lösungsoptionen identifizieren, die Vor- und Nachteile („Tradeoffs") der einzelnen Optionen im gegebenen Kontext herausarbeiten (denn keine Lösung hat nur Vorteile) und letztlich den beteiligten Personengruppen helfen, die bestmöglichen Entscheidungen zu treffen, indem man sie dabei unterstützt, die Situation sowie die Optionen mit ihren Vor- und Nachteilen möglichst ganzheitlich zu verstehen. Obwohl der Satz jetzt recht lang war, ist das eine extrem kompakte Beschreibung der Aufgaben, die man aus meiner Sicht als Architekt(in) wahrnehmen sollte. Entsprechend beschreibt das auch meine Hauptaufgabe als Software-Architekt.

In der konkreten Ausgestaltung kommt da natürlich noch ganz viel dazu, z.B. wann man welche Aktivitäten in Abhängigkeit vom gegebenen Kontext in welchem Umfang wahrnimmt oder wie man mit den anderen Stakeholder-Gruppen, insbesondere auch dem Entwicklerteam zusammenarbeitet. Aber das würde den Rahmen hier bei weitem sprengen. Wichtig ist dabei aus meiner Sicht nur, dass es keine einzelne wichtigste Hauptaufgabe gibt. Wenn man überhaupt etwas als besonders wichtig darstellen möchte, dann ist es die Ganzheitlichkeit der Betrachtung und des Handelns.

**JAXenter: Software-Architektur beschäftigt sich traditionell stark mit Backend-Problematiken: Provisionierung, Skalierbarkeit, Resilienz, etc. Im Umfeld der sogenannten Serverless-Bewegung spricht man nun aber immer mehr von Managed Backends und Backend as a Service. Wird das Backend als solches für Architekten also immer uninteressanter?**

**Uwe Friedrichsen:** Aus meiner Sicht nein. Serverless vergrößert ja die Menge möglicher Lösungsoptionen. Architekturarbeit bedeutet ja nicht, alles bis ins letzte Detail zu designen und jedes Rad wieder und wieder neu zu erfinden, sondern für eine gegebene Aufgabenstellung im Rahmen der gesetzten Rahmenbedingungen eine möglichst sinnvolle Lösung zu finden.

An der Stelle stellt Serverless mit Fokus auf Managed Services für mich eine Bereicherung dar. Wir haben einen ganz neuen Typus an Kauflösungen an die Hand bekommen, der ganz andere Eigenschaften als traditionelle Standard-Software hat, nämlich eher der Unix-Philosophie „Mache nur eine Sache und die gut“ folgt sowie „Integration first“ und „Pay per use“ ist. Das gibt mir in der Architekturarbeit ganz neue, spannende Optionen, die ich bislang in der Form nicht hatte.

Außerdem hilft uns Serverless noch an einer anderen Stelle. Schauen wir einmal auf die derzeit immer weiter explodierende technische Komplexität heutiger Individualentwicklungsprojekte: Microservices, Spring Boot, Docker, Kubernetes, Istio, ELK, SPA, React, React native, Prometheus, Jenkins, Webpack, Kafka, Cassandra, PostgreSQL, Swagger, GraphQL, und, und, und. Das ist ein heute typischer technischer Minimal-Stack. Und obgleich „minimal“ an der Stelle schon nach Hohn und Spott klingt, ist es damit ja noch nicht getan, denn zu dem Stack gesellen sich noch Dutzende weiterer Tools und Frameworks. Das war ja nur die offensichtliche Spitze des Technologieeisbergs.

Wenn wir dagegen die Notwendigkeit stellen, immer schneller auf sich verändernde Marktbedingungen zu reagieren, dann wird klar, dass sich das nicht durchhalten lässt. Wir müssen die Fertigungstiefe reduzieren, wenn wir ohne Qualitätsverluste schnell agieren können wollen. Genau hier hilft Serverless: Indem wir die nicht differenzierenden Teile der Anwendungslogik als Managed Services dazukaufen, können wir unsere Kapazitäten auf die differenzierenden Teile fokussieren und da einen richtig guten Job machen.

Also nein, ich finde nicht, dass Architekturarbeit im Backend weniger interessant wird. Ich finde sie sogar spannender durch die neuen Optionen, die Serverless bietet.

**JAXenter: Simon Wardley hat die wilde These aufgestellt, dass Container und Kubernetes nur eine Randerscheinung in der Geschichte der Softwareentwicklung darstellen und bald schon obsolet werden könnten, da Serverless – wie einst Software – die Welt verschlingt. Wie stehst du dazu?**

**Uwe Friedrichsen:** Die These finde ich persönlich überhaupt nicht wild. Tatsächlich sehe ich das ähnlich, wobei man etwas genauer hinschauen muss, wie Simon Wardley das aus meiner Sicht meint: Container und Container-Scheduler wie Kubernetes werden weiter existieren, nur werden sie „unsichtbar“ werden, sprich hinter höherwertigen Abstraktionen verschwinden.

AWS Fargate als Managed Container-Scheduler deutet den Weg an: In Zukunft werde ich mich nicht mehr mit den Komplexitäten von Docker und Kubernetes herumschlagen. Stattdessen werde ich meinen Anwendungscode schreiben, der als eigene Einheit, sprich Service bereitgestellt werden soll. An Metadaten gebe ich nur mit, auf welche fachlichen Events der Service reagieren soll. Die CI/CD-Pipeline sorgt dafür, dass am Ende ein fertiger (für uns unsichtbarer) Container in einem Repository bereitsteht, und der Scheduler kümmert sich darum, dass der Container immer dann verfügbar ist, wenn er gebraucht wird. Im Hintergrund mögen immer noch Docker, Kubernetes oder was auch immer werkeln, aber für unser Denkmodell ist das irrelevant, weil wir als Entwickler auf einer ganz anderen

# „Container und Container-Scheduler wie Kubernetes werden weiter existieren, nur werden sie unsichtbar werden."

Abstraktionsebene mit der Infrastruktur interagieren werden.

Um noch eine alternative Entwicklung zu skizzieren: Es ist auch vorstellbar, dass sich Serverless Functions in Richtung vollständiger, tiefer Anwendungsentwicklung weiterentwickeln werden. Dafür sind sie derzeit eigentlich weder gedacht noch geeignet, aber das hat uns in der IT bekanntermaßen noch nie davon abgehalten, es trotzdem zu tun. In dem Fall würden, getrieben vom Bedarf, mächtigere Function-IDEs, bessere Strukturierungs- und Debug-Möglichkeiten, Laufzeitoptimizer, etc. für Functions entstehen.

Das würde sich dann ein wenig wie CICS für synchrone Verarbeitung und TSO/JCL für asynchrone Verarbeitung auf dem Host anfühlen. Klingt das jetzt sehr schräg? Ja, auf den ersten Blick vielleicht schon. Aber wir laufen derzeit gerade wieder durch eine weitere Technologie-Evolutionswelle in der IT – inklusive der typischen Effekte wie Technologieexplosion, Überforderung vieler betroffener Entwickler und mehr. Eine ähnliche Evolutionswelle gab es auf dem Host vor ca. 30 Jahren, und das Ergebnis findet man z.B. in CICS und TSO/JCL. Und was sich derzeit mit Step Functions und Co. andeutet, zeigt ganz stark in die gleiche Lösungsrichtung.

Persönlich bin ich mir noch nicht sicher, in welche Richtung sich das alles mit Serverless entwickeln wird. Allerdings teile ich die These, dass Container und deren Scheduler, wie wir sie heute kennen, in nicht ferner Zukunft von höherwertigen Abstraktionen abgelöst werden.

**JAXenter: Viele Jahre lang war der große, möglichst komplette Application Server das Ideal und die Basis vieler Software-Architekturen. Hat diese Metapher in Zeiten der Cloud und der Microservices ausgedient?**
**Uwe Friedrichsen:** Aus meiner Sicht ein klares „Jein". In einem echten Cloud-Native-Szenario hat der klassische Application Server natürlich keine Zukunft mehr. Aber Cloud Native ist am Ende auch nur ein Architekturstil unter vielen mit spezifischen Vor- und Nachteilen, und nicht jedes Unternehmen muss diesen Stil umsetzen. Es wird weiterhin Unternehmen bzw. Bereiche innerhalb von Unternehmen geben, die mit anderen Stilen besser fahren werden. Dazu gehört die Nutzung von Application Servern.

Rein inhaltlich wird es aus meiner Sicht daher immer noch einen Platz für umfassende Application Server geben. Der wird aber sehr viel kleiner sein als in der Vergangenheit.

**JAXenter: Und dann die Blockchain – von der immer wieder zu hören ist, dass sie das gesamte Internet revolutionieren könnte. Siehst du das kommen? Oder bleibt die Blockchain eine mögliche Lösung für einen bestimmten Anwendungsfall?**
**Uwe Friedrichsen:** Nun ja, eine differenzierte Antwort auf die Frage würde recht lang, und ich habe bereits einige lange Antworten gegeben. Also halte ich es hier einmal kurz: Die Revolution ist aus meiner Sicht ausgeblieben und wird auch nicht stattfinden. Blockchain und Smart Contracts werden ihre Nische finden, aber das Internet revolutionieren? Nein, das sehe ich nicht.

**JAXenter: Was ist momentan dein persönliches Steckenpferd: Welches Architekturthema liegt dir besonders am Herzen – und warum?**
**Uwe Friedrichsen:** Im Bereich der Architektur gibt es zwei Themen: Das erste Thema ist, was Architekturarbeit überhaupt bedeutet und warum wir sie brauchen – eine Art Refokussierung. Ich beobachte sehr viele, häufig hitzige Debatten, die aus meiner Sicht leider regelmäßig am Ziel vorbeischießen, weil sie die eigentliche Kernfrage, nämlich das „Warum", nicht stellen.

# „Mit Microservices schaffen wir eine stark verteilte Anwendungs-Landschaft, und die Effekte der Verteilung werden sehr oft unterschätzt."

Da ich im Laufe meines Berufslebens immer wieder erlebt habe, wie viel Mehrwert Architekturarbeit stiften kann, wenn man sie sauber auf ihren eigentlichen Daseinszweck, auf das „Warum", ausrichtet, habe ich damit begonnen, dieses Thema intensiver zu bearbeiten. So habe ich z.B. auf dem Software Architecture Summit einen Workshop über essentielle Architekturarbeit gehalten, und es werden noch diverse weitere Workshops unterschiedlichen Umfangs zu dem Thema folgen. Mein Ziel ist es, den Teilnehmern zu vermitteln, wie sie mit auf ihren eigentlichen Zweck ausgerichteter Architekturarbeit echten Mehrwert schaffen können.

Mein derzeitiges zweites Architekturthema ist fachliches Design in verteilten Systemen. Seit mehreren Jahren sind Microservices der populärste Architekturstil, und im Zusammenspiel mit Containern und Schedulern wie Kubernetes ergeben sich damit interessante Optionen. Allerdings schaffen wir mit Microservices auch eine stark verteilte Anwendungslandschaft, und die Effekte der Verteilung werden sehr oft stark unterschätzt.

Die geschaffenen Anwendungslandschaften sollen ja üblicherweise zuverlässig verfügbar sein, kurze Antwortzeiten haben, im Fehlerfall ein robustes Verhalten zeigen, ohne Wartungsfenster auskommen, und so weiter. Dem steht das nicht-deterministische Verhalten verteilter Systeme und speziell entfernter Aufrufe im Weg. Mit jedem zusätzlichen entfernten Aufruf steigt die Wahrscheinlichkeit, dass wir die geforderten Eigenschaften der Anwendungslandschaft nicht einhalten werden.

Als Gegenmaßnahmen werden dann in der Regel entweder generische Maßnahmen wie z.B. der Einsatz eines Service Meshes (wie Istio oder Linkerd) oder technische Resilienzmuster auf Anwendungsebene (etwa Circuit Breaker oder Backpressure) genannt. Das ist aber nur die halbe Wahrheit. Auch wenn die genannten Maßnahmen grundsätzlich sinnvoll sind, können sie ihr Potential nur ausspielen, wenn der fachliche Schnitt zwischen den verschiedenen Services stimmt.

Ein simples Beispiel, das man fast täglich in der Praxis antrifft: Service A erhält eine externe Anfrage. Um diese Anfrage beantworten zu können, benötigt er auf fachlicher Ebene Informationen von Service B. Sollte Service B jetzt nicht verfügbar sein, kann Service A seine Anfrage nicht beantworten, sprich er ist ebenfalls nicht verfügbar. Und sollte Service B langsam antworten, antwortet Service A ebenfalls langsam, und so weiter. Auf technischer Ebene kann ich jetzt so viele Circuit Breaker (mit oder ohne Service Mesh) zwischen Service A und Service B einziehen wie ich will – alles, was diese mir im Fehlerfall sagen, ist: Service A ist jetzt auch kaputt!

Der Grund dafür liegt im fachlichen Design. Die Fachlichkeit ist so zwischen den Services aufgeteilt worden, dass ein Problem in Service B sich immer in Service A fortpflanzt, egal was wir auf technischer Ebene an Gegenmaßnahmen ergreifen. Deshalb hat gutes fachliches Design gerade in verteilten Systemen eine zentrale Bedeutung. Im Laufe der Zeit habe ich aber gelernt, dass die meisten verbreiteten Design-Methoden nicht geeignet sind, um verteilte Systeme fachlich zu strukturieren, weil sie die nicht-deterministische Kommunikation zwischen Prozessgrenzen nicht beachten.

Entsprechend bin ich auf die Suche gegangen und habe geschaut, wie man fachliches Design für verteilte Systeme sinnvoll gestalten kann. Interessanterweise bin ich dabei auf ganz alte Design-Praktiken aus den frühen 70er Jahren gestoßen, die ich allerdings ein wenig auf den veränderten Kontext anpassen musste – in den frühen 70er Jahren hatte man es ja noch nicht mit verteilten Systemen zu tun.

Über diese Herausforderungen beim Design verteilter Systemlandschaften sowie die Ansätze, die ich dazu gefunden habe, erzähle ich derzeit viel. Ich bilde mir nicht ein, dass ich die perfekte Lösung kenne. Aber ich hoffe, dass ich die Diskussion, wie man das Thema angehen sollte, damit ein wenig nach vorne bringe. Das Thema ist in der heute immer mehr verteilten Anwendungswelt wichtiger denn je – und aus meiner Sicht bei weitem nicht zufriedenstellend gelöst.

**Uwe Friedrichsen** ist ein langjähriger Reisender in der IT-Welt. Als CTO und Fellow der codecentric AG ist er stets auf der Suche nach innovativen Ideen und Konzepten. Seine aktuellen Schwerpunktthemen sind (verteiltes) Systemdesign und die IT von (über)morgen. Er teilt und diskutiert seine Ideen regelmäßig auf Konferenzen, als Autor von Artikeln, Blogposts, Tweets und im direkten Gespräch.

## Vom Umgang mit funktionalen Anforderungen

# Req4Arc

Der saubere Start hat geklappt. Wir haben uns auf Ziele, Stakeholder und Scope geeinigt. Jetzt gilt es, die Anforderungen zu verstehen und zu klären – und zwar sowohl die funktionalen Anforderungen als auch Qualitätsanforderungen und Randbedingungen.

von Dr. Peter Hruschka und Dr. Gernot Starke

In diesem Artikel konzentrieren uns auf die funktionalen Anforderungen. Dabei interessieren uns besonders die für die Architektur relevanten – und weniger eine vollständige Liste aller benötigten Funktionen. Sie müssen wissen, welche funktionalen Anforderungen Ihre Architekturentscheidungen besonders prägen werden.

Woher jedoch können wir diese relevanten von den weniger relevanten unterscheiden? Lassen Sie uns dazu auf die verschiedenen Granularitäten funktionaler Anforderungen schauen.

### Granularität von Anforderungen

Ihre Stakeholder werden Ihnen manchmal sehr grobgranulare Anforderungen nennen („das Navisystem soll in der nächsten Version auch einen Spurassistenten haben“) und manchmal sehr feingranulare Anforderungen („die erwartete Ankunftszeit für die aktive Route soll auffällig und gelb unterlegt rechts unten im Kombidisplay dargestellt werden“).

Für Sie als Architekt(in) ist es wichtig, sich einen Überblick über die gewünschte Funktionalität zu verschaffen. Leider können wir unsere Stakeholder nicht davon abhalten, alles, was sie wollen, in beliebiger Reihenfolge und Granularität von sich zu geben. Wenn Sie Details zu hören bekommen, sollten Sie nachfragen, zu welchem größeren Thema diese denn gehören. Sie wollen die unterschiedlichen Granularitäten (funktionaler) Anforderungen in eine Hierarchie bringen, wie in **Abbildung 1** aufgezeigt. Für Ihre Architekturarbeit konzentrieren Sie sich dann zuerst auf die oberste Ebene.

Die grobgranularen Anforderungen hätten Sie gerne vollständig, damit Sie einen Überblick über das Gesamtsystem haben und entscheiden können, welche Teile Sie in der Implementierung vorziehen und welche Sie zurückstellen wollen.

### Denken in Prozessen – wenn möglich

Wie kommen wir zu einer grobgranularen Zerlegung des gesamten Systems, die wir möglichst getrennt voneinander analysieren und implementieren können? Nach welchen Kriterien sollen wir gliedern oder zerlegen?

Viele der Analysemethoden der letzten Jahrzehnte schlagen – unter unterschiedlichen Namen – vor, eine Zerlegung in Prozesse vorzunehmen. Diese Prozesse

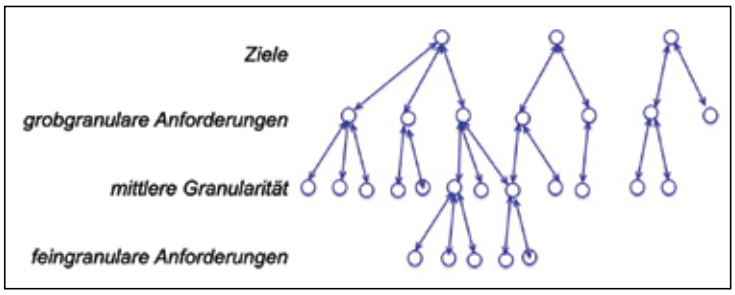


Abb. 1: Granularität von Anforderungen – von Zielen zu funktionalen Details

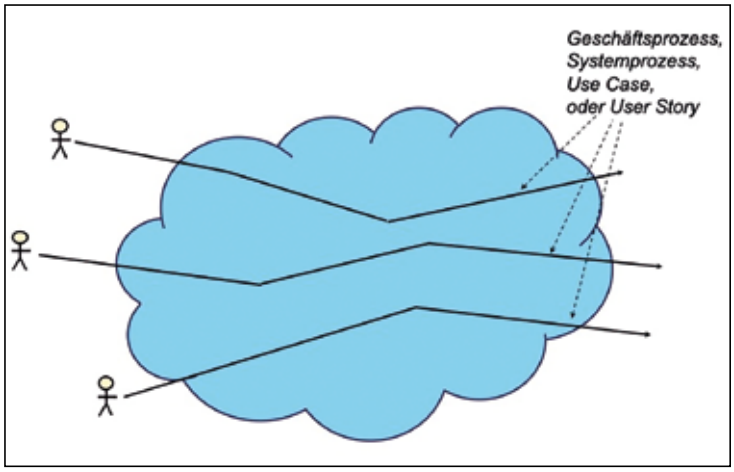


Abb. 2: Grobgranulare Anforderungen – gliedern nach Prozessen

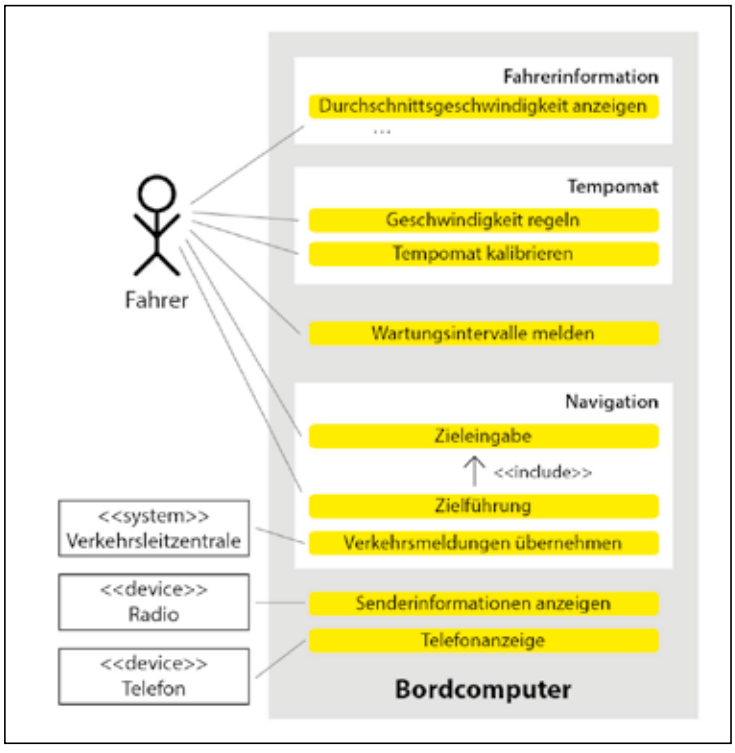


Abb. 3: Use-Case-Diagramm als Überblick über funktionale Anforderungen

werden aus dem Kontext (d. h. aus der Systemumgebung) getriggert und stellen vergröbert den gesamten Ablauf von den Eingaben bis hin zu den Ergebnissen dar (**Abb. 2**).

Diese Prozesse werden durch Auslöser aus der Umgebung gefunden und sind somit unabhängig von jeder internen Strukturierung Ihres Systems. Außerdem stellen diese externen Trigger sicher, dass es jemanden in der Systemumgebung (im Kontext) gibt, der diesen Prozess haben will. Das sind zwei der INVEST-Kriterien, die Bill Wake [1] für gute Stories aufgestellt hat: unabhängig voneinander und wertvoll.

Für Sie als Architekt(in) ist dieser Überblick einiges wert: Selbst wenn Sie im ersten Release nur einen Teil davon implementieren wollen, wissen Sie doch, wo die Reise hingeht – was Kurzsichtigkeit bei Designentscheidungen verhindert.

Die zweitbeste Möglichkeit zur Grobgliederung von Anforderungen – wenn Sie ein bestehendes System ergänzen oder erweitern – ist die Gliederung nach der bekannten Struktur dieses Vorgängersystems. Sie können dann neue Anforderungen zu spezifischen Subsystemen oder bestehenden Komponenten erfassen. Diese Anforderungsgliederung erschwert jedoch innovative Ideen zum Produkt, weil Sie die gegebene architekturelle Struktur wahrscheinlich nicht hinterfragen oder anzweifeln werden, sondern nur lokale Ergänzungen oder Änderungen spezifizieren.

## Schreiben oder zeichnen?

Wenn Sie in Prozessen denken, können Sie nach Lust und Laune wählen, ob Sie Umgangssprache zur Darstellung dieser Prozesse bevorzugen oder lieber grafische Darstellungen.

Umgangssprachlich bewährt sich die Formel, die Mike Cohn für Stories vorgeschlagen hat [2]: „Als <Nutzer> möchte ich <Funktion>, sodass <Vorteil>." Dazu zwei konkrete Beispiele:

- „Als Fahrer möchte ich auf Wartungsintervalle hingewiesen werden, damit mein Fahrzeug immer fahrtüchtig bleibt."
- „Als Fahrer möchte ich das Tempo mit dem Tempomat konstant halten, um nicht selbst das Gaspedal bedienen zu müssen und in geschwindigkeitsbeschränkten Bereichen die Höchstgeschwindigkeit nicht zu überschreiten."

Diese sprachliche Formel gewährleistet, dass es erstens jemanden gibt (den Nutzer), der die Funktion wirklich haben will und sie erklärt zweitens auch den Grund dafür (den Vorteil). Als Entwicklerteam erlaubt Ihnen dieser Vorteil, Ihren Nutzern bei Bedarf Alternativen zu der Funktion vorzuschlagen und vermeidet somit die zu frühe Festlegung auf Lösungsdetails.

Das Gleiche kann man gemäß Ivar Jacobson [3] natürlich auch in Form von Use Cases darstellen, mit einem Strichmännchen als Auslöser (Actor) und der Prozessbezeichnung in der Ellipse, die diesen Use Case repräsentiert (**Abb. 3**).

## Mehr Details zu den Prozessen

Viele gefundene Prozesse sind sicherlich noch zu vage, zu abstrakt, um schon Klarheit über die funktionalen Anforderungen zu haben. Wenn Sie entschieden haben, was zuerst implementiert werden soll, stehen Ihnen viele Ausdruckmöglichkeiten zur Verfügung, um die funktionalen Details zu spezifizieren. Sie können den Prozess mit Aktivitätsdiagrammen, Datenflussdiagrammen, BPMN, Ereignisprozessketten, Szenarien, kleiner geschnittenen Stories u. v. a. m. weiter präzisieren. Beispiele zu vielen Notationen finden Sie in [4].

## Empfehlungen

Nutzen Sie „große Prozesse“ als Ansatz zur Strukturierung der Requirements Ihres Systems. Diese sind unabhängig voneinander, wegen der externen Trigger für irgendjemanden in der Systemumgebung wertvoll und können somit unabhängig voneinander implementiert werden.

Warten Sie noch ein oder zwei Folgen der Kolumne ab – weil wir Ihnen aus dem Domain-driven Design noch einige Patterns zum besseren Verständnis funktionaler Anforderungen vorstellen werden (und Sie dann die so genannten Bounded Contexts zur Strukturierung Ihres Systems heranziehen können).

Streben Sie anfangs mehr nach Überblick über die Gesamtfunktionalität als nach zu vielen Details – auch wenn die Nutzer versucht sind, Ihnen viele Details zu erzählen.

## Fazit

Mit dem sauberen Start (Scope und Kontext) haben wir für unser System die Außenschnittstellen klar definiert. Jetzt kommen die wesentlichen funktionalen Anforderungen hinzu – wobei Sie in dieser Folge die Ende-zu-Ende-Prozesse und einige Möglichkeiten zur Beschreibung von Funktionen (Stories und Use Cases) kennengelernt haben. Im Grunde können Sie jetzt mit Architekturarbeit loslegen. In der nächsten Folge stellen wir die anforderungsbezogenen Teile von Domain-driven Design vor, die uns bei komplexer Fachlichkeit sehr helfen können. Bis dahin alles Gute.

**Streben Sie nach Überblick über die Gesamtfunktionalität –auch wenn die User Ihnen viele Details erzählen.**

**Dr. Peter Hruschka** (www.b-agile.de) und Dr. Gernot Starke (innoQ-Fellow, www.gernotstarke.de) haben vor einigen Jahren www.arc42.de ins Leben gerufen, das freie Portal für Softwarearchitektur, und nun auch www.req42.de, das freie Portal für agiles Requirements Engineering. Sie sind als Gründungsmitglieder sowohl im IREB (www.ireb.org) als auch im iSAQB (www.isaqb.org) vertreten sowie Autoren mehrerer Bücher rund um Requirements Engineering, Softwarearchitektur und Entwicklungsprozesse.



## Links & Literatur

[1] Wake, Bill: „INVEST in Good Stories, and SMART Tasks“: https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

[2] Cohn, Mike: „User Stories Applied“, Addison Wesley, 2004

[3] Jacobson, Ivar: „Use-Cases 2.0“: https://www.ivarjacobson.com/publications/white-papers/use-case-ebook

[4] Hruschka, Peter: „Business Analysis und Requirements Engineering“, Carl Hanser Verlag, 2. Auflage 2019

Im Interview: Anna-Maria Schaum, Head of Business Intelligence & Controlling bei MyHammer

# „Was wir brauchen, sind Vorbilder"

In unserer Artikelserie „Women in Tech" stellen wir inspirierende Frauen vor, die erfolgreich in der IT-Branche Fuß gefasst haben. Heute im Fokus: Anna-Maria Schaum, Head of Business Intelligence & Controlling bei MyHammer.

Die Tech-Industrie wird von Männern dominiert – so weit, so schlecht. Doch langsam, aber sicher bekommt der sogenannte Boys Club Gesellschaft von begabten Frauen: Immer mehr Frauen fassen in der Branche Fuß.

Aus diesem Grund wollen wir hier spannenden und inspirierenden Frauen die Möglichkeit geben, sich vorzustellen und zu erzählen, wie und weshalb sie den Weg in die Tech-Branche gewählt haben. Aber auch Themen wie Geschlechtervorurteile, Herausforderungen oder Förderungsmöglichkeiten kommen zur Sprache.

## Was hat dein Interesse für die Tech-Branche geweckt?

Ich bin schon sehr früh ein Zahlenfreak gewesen. Das war mir früher gar nicht so bewusst, bis ich vor ein paar Jahren bei meinem letzten Umzug über alte Zettel gestolpert bin. Ich hatte dort meine Einnahmen aus dem Mathe-Nachhilfeunterricht dokumentiert. Bei diesem Anblick ist mir klar geworden, dass ich auch damals schon in sehr rudimentärer Form das gemacht habe, was ich auch heute noch liebe: mit Zahlen umgehen. Sei es, sie erstmal zu erfassen, sie dann auswertbar zu machen oder eben die richtigen Rückschlüsse auf Basis von Daten zu ziehen.

An der TU Berlin habe ich schließlich BWL studiert. Meine Schwerpunkte lagen im operativen und strategischen Controlling sowie der Organisation und Unternehmensführung. Nach dem Studium war ich dann als Junior Analyst bei Coca-Cola tätig und habe mich zum Team Lead hochgearbeitet. Bei MyHammer bin ich als Senior Controller eingestiegen und durfte nach meiner Elternzeit als Head of Business Intelligence & Controlling zurückkehren.

## Ein Tag in Anna-Marias Leben

In meiner Position bin ich vor allem dafür verantwortlich, die Ressourcen in meinem Team zu managen sowie die unterschiedlichen Anforderungen aller Stakeholder aufzunehmen und für deren Umsetzung zu priorisieren. Mein Team kümmert sich wiederum um alle Datenbelange des Unternehmens. Selbst habe ich bisher noch nichts entwickelt, aber ich bin ja noch jung ;).

## Vorbilder und Förderer

Unterstützt hatten bzw. haben mich immer sehr stark meine direkten Vorgesetzten, beides Männer :). Ein konkretes Vorbild habe ich nicht, aber ich bewundere alle Frauen, die es schaffen, den Spagat zwischen Familie

## Was wir vor allem brauchen, sind Vorbilder, die allgegenwärtig sind und keine Ausnahmen, die die Regel bestätigen.

und Karriere zu bestehen. Das war tatsächlich die bisher größte Herausforderung für mich.

Steine in den Weg gelegt hat mir zum Glück niemand. Ganz im Gegenteil: meine Karriere wurde immer wieder durch andere indirekt gesteuert, indem man in mir mehr gesehen hat, als ich selbst mir zugetraut hätte.

### Hindernisse auf dem Weg

Nach dem Abschluss meines Studiums wurde ich zur Berufsberatung der Agentur für Arbeit eingeladen. Dort hat man mir ziemlich offen gesagt, dass ich als Frau nie Karriere machen werde, da ich ohnehin bald Kinder bekomme. Mir als einer frisch diplomierten Uniabsolventin, das muss man sich mal vorstellen.

In meinem Team heute sind wir tatsächlich paritätisch besetzt, auch wenn ich nicht gezielt darauf hingearbeitet habe. Auch wenn wir Stellen für meinen Bereich neu besetzen mussten, war der Anteil von weiblichen Bewerbern recht hoch. Ich denke die Hürden sind nicht zwingend ein Tech-Thema, sondern eher allgemeinerer Natur: Als Frau kämpft man auch heute noch gegen die üblichen Vorurteile: „Die wird doch sicher bald schwanger“, „Sie hat Kinder, die sind doch sicher häufig krank“.

Ich fürchte die Debatte um Frauen in der Tech-Branche wird uns noch sehr lange beschäftigen. Bestimmte Stereotypen sind auch heute noch so tief in der Erziehung verankert und dabei vor allem unbewusst, sodass es schwer ist, hier anzugreifen. Mein fünfjähriger Sohn meint, Frauen würden weniger gut Auto fahren und rosa ist eine „Mädchen-Farbe“, das habe ich ihm ganz sicher nicht „beigebracht“, es ist trotzdem da.

### Hoffnung für die Zukunft

Je mehr Frauen im Tech-Bereich arbeiten, desto selbstverständlicher wird dies. Was wir vor allem brauchen, sind Vorbilder, die allgegenwärtig sind und keine Ausnahmen, die die Regel bestätigen. Für Arbeitgeber darf sich gar nicht die Frage stellen, ob man eine Frau einstellt, weil sie ja später schwanger werden könnte.

Für Frauen, die in die Tech-Branche einsteigen möchten, habe ich den Ratschlag, dass man sich tatsächlich von niemandem reinreden lassen sollte. Macht lieber das, was euch Spaß macht!



**Anna-Maria Schaum** ist Head of Business Intelligence & Controlling bei MyHammer. Mit ihrem Team ist sie damit u. a. für alle Aufgaben rund um Datenanalyse und -visualisierung verantwortlich. Seit 2012 ist sie bereits für MyHammer tätig und hat die Führungsposition seit 2014 inne. Das Besondere: Die Mutter von zwei Kindern übt die Tätigkeit in Teilzeit aus. Vor ihrem aktuellen Engagement war Anna-Maria Schaum u. a. bei SAP und Coca-Cola tätig. Zuvor studierte sie BWL mit Schwerpunkt Controlling und Unternehmensführung an der TU Berlin.

Ein Sündenbock hilft nicht! Blameless Post Mortems

# DevOps Stories

Agilität, Management 3.0, New Work oder DevOps – all diese Bewegungen verändern die Art und Weise, wie Softwareentwicklung organisiert wird. Wenn man ihren Denkmustern und Ideen konsequent folgt, beeinflussen sie stark die Kultur des Unternehmens. Über solche Kulturveränderungen berichtet die Kolumne DevOps Stories. Heute belauschen wir das MusicStore-Team beim morgendlichen Stand-up.

von Konstantin Diener

Das MusicStore-Team steht morgens beim Stand-up zusammen:

**Lukas:** *„Heute Nacht gab es vermehrt Alerts im Produktivsystem."*

**Julia:** *„Wann hat das angefangen?"*

**Lukas:** *„Gestern gegen 21:00 Uhr. Zumindest sieht es im Chatverlauf danach aus."*

**Erik:** *„Was war da los?"*

**Lukas:** *„Es gab scheinbar Probleme mit den Kreditkartenzahlungen, die ..."*

**Manuela:** *„Das passt, ich habe hier drei Beschwerden. Die Kunden wollten ein Abo mit der Kreditkarte bezahlen, es hat aber nicht geklappt."*

**Ruben:** *„Heute Nacht war Jens mit der Rufbereitschaft dran, oder?"*

**Lukas:** *„Ja."*

**Christian:** *„Wo ist der schon wieder?"*

**Ruben:** *„Er hat doch gestern gesagt, dass er heute Morgen zum Zahnarzt muss."*

**Christian:** *„Super Timing!"*

**Lukas:** *„Durch den Verlauf in unserem ChatOps-Tool können wir doch eigentlich ganz gut nachvollziehen, was da passiert ist. Gegen 21:30 Uhr wurde Jens informiert, und er hat direkt mit der Fehlersuche begonnen. Um 23:40 Uhr hat er festgestellt, dass das Problem offenbar mit seiner Änderung am Bezahlprozess zu tun hat."*

**Christian:** *„Dafür hat der ernsthaft über zwei Stunden gebraucht?"*

**Lukas:** *„Die neue API-Version verhält sich offenbar anders als die bisherige. Als das klar war, hat er die Änderung zurückgerollt."*

**Julia:** *„Es sieht hier aber danach aus, dass die Kartenzahlungen dann immer noch nicht funktionierten, oder?"*

**Lukas:** *„Korrekt. Durch das Zurückrollen der API-Version ist irgendwie der API Key der Testumgebung auch auf Production gelandet. Und aus gutem Grund lassen sich damit keine echten Zahlungen durchführen."*

**Manuela:** *„Jetzt funktioniert's aber wieder, oder?"*

**Lukas:** *„Genau. Jens hat um 01:05 Uhr den richtigen API Key auf dem Produktivsystem gesetzt, und seitdem funktioniert es wieder."*

**Manuela:** *„Alles klar, dann setzte ich jetzt gleich eine Info an unsere Kunden auf. Sie sollen es einfach nochmal versuchen. Aber jetzt ist maximale Transparenz wichtig."*

**Christian:** *„Das ist doch peinlich. Die Kunden sind sauer, weil ihr Abo nicht funktioniert hat! Und du teilst ihnen jetzt mit, dass es daran lag, dass bei uns Dilettanten arbeiten, die Bugs in Produktion pushen, niemanden um Hilfe fragen und dann auch noch so dämlich sind, den falschen API Key zu verwenden? Jens bekommt das einfach nicht auf die Reihe ..."*

**Ruben:** *„Ich finde, Jens hat sehr professionell reagiert. Und du glaubst doch wohl nicht, dass er dich um Hilfe bittet, wenn du so über ihn redest!"*

## In komplexen Systemen passieren Fehler ...

Im MusicStore-Team ist ein Fehler passiert, der Endkunden verärgert und vermutlich auch zu Umsatzeinbußen geführt hat. Es gibt nicht wenige Menschen in der IT-Branche, die in einer solchen Situation wie Christian denken: Der Fehler hätte vermieden werden können. Wenn wir uns nur ausreichend anstrengen, passieren keinerlei Fehler.

Nach Meinung von Werner Vogels, CTO bei Amazon, ist diese Ansicht nicht ganz richtig. Von ihm stammt das Zitat „Everything fails all the time". Dahinter steckt die Überzeugung, dass IT-Systeme heute so komplex sind, dass sich Fehler nicht per se vermei-

Wenn die Person nicht von der Richtigkeit der Entscheidung überzeugt gewesen wäre, hätte sie anders entschieden. Eine Bestrafung der Person stellt also keine sinnvolle Reaktion auf einen Fehler dar.

den lassen. Entscheidend ist der professionelle Umgang mit diesen Fehlern.

Einen Beweis für diese Ansicht liefert ein Fehler, der im Oktober 2018 bei GitHub aufgetreten ist [1]. Durch den Austausch von Netzwerkhardware entstand zwischen verschiedenen Standorten ein Verbindungsausfall von 43 Sekunden, der in letzter Konsequenz zu über 24 Stunden mit (Teil-)Ausfällen der GitHub-Produkte führte. GitHub ist mit Sicherheit kein kleines Unternehmen, beschäftigt einige der besten Softwareingenieure der Welt und ist in der Entwicklung und im Betrieb großer Softwarelösungen definitiv kein Anfänger.

## Den „Schuldigen" zu bestrafen, erhöht nicht die Sicherheit, sondern verringert sie

Der Bericht über den Ausfall ist gut geschrieben. Neben einigen anderen Aspekten fällt einer besonders auf: An keiner Stelle steht „Es war die Schuld der Netzwerkingenieure (o. a.) und wir haben sie entlassen".

Auch dieser Teil von Christians Reaktion auf die nächtlichen Probleme ist in vielen Organisationen immer noch weit verbreitet: Wir suchen den Schuldigen und bestrafen ihn. Seiner Meinung nach hätten andere Kollegen es gar nicht zu diesem Fehler kommen lassen. Dass es ein Problem gab, lag einzig und allein an Jens' Inkompetenz. Dabei ist es nicht unwahrscheinlich, dass ein ähnlicher Fehler auch anderen im Team hätte passieren können. Wir Menschen lassen uns in solchen Situationen allzu oft vom fundamentalen Attributionsfehler [2] („Meine Fehler entstehen durch den Kontext, die der anderen durch Inkompetenz oder schlechte Absichten") oder vom Rückschaufehler („Rückblickend war dieses Ereignis vorhersehbar") [3] leiten.

Aus der Überzeugung, dass die Ursache für den Fehler in einer bestimmten Person liegt, wird in der Regel die Schlussfolgerung gezogen, dass man diese Person nur feuern oder von kritischen Systemen fernhalten muss, um Fehler in Zukunft zu vermeiden. Damit geht auch die Überzeugung einher, dass Sicherheit sich durch Abschreckung steigern lässt. Wenn man den Verursacher schon nicht entlässt oder abschottet, muss man ihn auf jeden Fall gut sichtbar bestrafen, damit er und andere sich nicht noch einmal eine solche Dummheit erlauben.

Dieser komplette Denkansatz geht davon aus, dass der Verursacher vorsätzlich oder grob fahrlässig einen Fehler begangen hat [4]. Tatsächlich ist es aber oft so, dass derjenige der Meinung war, dass ...

- ... das, was als Folge seiner Handlung eingetreten ist, nicht möglich sei.
- ... das, was passiert ist, nichts mit seinen Aktionen zu tun hat.
- ... die Erreichung des gemeinsamen Ziels jede Art von Risiko wert ist.

Kurz gesagt: Wenn die Person nicht von der Richtigkeit der Entscheidung überzeugt gewesen wäre, hätte sie anders entschieden. Eine Bestrafung der Person stellt also keine sinnvolle Reaktion auf einen Fehler dar. Tatsächlich verringert man in der Regel mit einem solchen Schritt die Sicherheit, statt sie zu erhöhen.

Obwohl Jens beim Zahnarzt ist, kann das Music-Store-Team relativ gut nachvollziehen, was sich in der vergangenen Nacht zugetragen hat. Möglich wird das, weil Jens jeden seiner Schritte im Gruppenchattool des Teams dokumentiert hat. Wenn er Angst vor einer Bestrafung haben müsste, würde er vermutlich nicht selbst

noch die Beweise in Form eines akribischen Protokolls liefern. Allgemein lässt sich feststellen, dass die Mitarbeiter in Organisationen mit einer ausgeprägten Bestrafungskultur nicht ehrlich sind, sondern versuchen, ihre Fehler zu vertuschen. Ohne detaillierte Informationen über die Entstehung werden der Mitarbeiter und seine Kollegen ähnliche Fehler in Zukunft kaum vermeiden können.

## Blameless Postmortems helfen, Fehler bestmöglich zu verstehen und sie damit zukünftig zu verhindern

Systeme werden nur sicherer, wenn alle Beteiligten offen über Fehler und deren Entstehung sprechen können und keine Strafe fürchten müssen. Dieses Prinzip einer Blameless Culture ist in der Luftfahrt [5], [6] und der Medizin schon länger bekannt. Hier wird ein Fehler als Möglichkeit gesehen, das System zu stärken.

IT-Unternehmen wie Google [7], Etsy [4], GitHub oder AWS haben deshalb für die systematische Analyse von Fehlern sogenannte Blameless Postmortems etabliert. Diese Postmortems werden bei Google in den folgenden Fällen durchgeführt:

- user-visible downtime or degradation beyond a certain threshold
- data loss of any kind
- on-call engineer intervention (release rollback, rerouting of traffic, etc.)
- a resolution time above some threshold
- a monitoring failure (which usually implies manual incident discovery)

Im Rahmen des Postmortems analysieren die beteiligten Personen den Fehler und seine Entstehung. Sie schreiben den zeitlichen Verlauf, den Kontext und die abgeleiteten Maßnahmen dann in einem Dokument auf, das von anderen Ingenieuren in der Firma gegengelesen und allen Teams zugänglich gemacht wird. Ein Beispiel findet sich unter [8]. Am Ende handelt es sich bei einem solchen Postmortem um eine spezielle, technisch orientierte Retrospektive. Unter anderem wird hier auch festgehalten, was im Kontext des Fehlers gut gelaufen ist.

Jens' Protokoll im Chattool stellt eine gute Ausgangsbasis für ein Postmortem dar, weil der zeitliche Verlauf sich offenbar klar erkennen lässt. Und es gibt offensichtlich Dinge, die gut funktioniert haben: Jens als Diensthabender ist sehr zügig über den Fehler informiert worden, er konnte seine Aktionen vom Gruppenchattool aus durchführen und der Deployment-Prozess ließ ihn zügig eine neue Version ausrollen, als er die Ursache identifiziert hatte. Das MusicStore-Team muss sich allerdings die Frage stellen, warum ihr Softwareentwicklungsprozess solche Fehler zulässt, wie sie Jens unterlaufen sind. Warum kann er das veränderte Laufzeitverhalten des APIs erst in Produktion sehen? Wie ist es möglich, dass durch ein Softwareupdate ein falscher API Key im Produktivsystem landet? Das Team sollte sich mit diesen Fragen beschäftigen, statt Jens die Schuld zuzuschieben.

Dieser Prozess fand statt, als Jens von seinem Arzttermin zurückkam. Als Ergebnis hat das Team in zusätzliche Tests und ein besseres Management der API Keys investiert.

**Konstantin Diener** ist CTO bei cosee. Er ist überzeugt, dass die meisten Probleme ihre Ursachen im Prozess und nicht bei den Menschen in der Organisation haben. Retrospektiven werden bei cosee sehr regelmäßig zur Analyse eingesetzt. Experimente mit Postmortems folgen.

https://cosee.biz @onkelkodi @coseeaner



## Links & Literatur

[1] GitHub-Blog: https://github.blog/2018-10-30-oct21-post-incident-analysis/

[2] „Wie kann man nur …!? – Wieso Entwickler sich mit Psychologie beschäftigen sollten"; Kolumne „DevOps Stories"; Java Magazin 8.2018; https://jaxenter.de/devops-stories-entwickler-psychologie-73738

[3] https://de.wikipedia.org/wiki/R%C3%BCckschaufehler

[4] https://codeascraft.com/2012/05/22/blameless-postmortems/

[5] https://www.eurocontrol.int/articles/just-culture

[6] https://www.vcockpit.de/recycled/vc-magazin-auszuege/details/news/just-culture-non-punitive-culture.html

[7] https://landing.google.com/sre/sre-book/chapters/postmortem-culture/

[8] https://landing.google.com/sre/sre-book/chapters/postmortem/

Querschnittsfunktionen mit Angular implementieren

# Die Angular-Abenteuer

Querschnittsfunktionen – das sind diese lästigen, meist technischen Anforderungen, die es immer und immer wieder zu berücksichtigen gilt. Beispiele dafür sind unter anderem Authentifizierung, Protokollierung oder die Behandlung von Fehlern. Natürlich möchte man die dafür nötigen Methodenaufrufe nicht ständig wiederholen müssen. Idealerweise werden sie automatisch aktiv.

von Manfred Steyer

In diesem Artikel zeige ich drei Mechanismen von Angular, die genau das auf elegante Art erlauben: Guards, HTTP Interceptors und Direktiven. Alle gezeigten Beispiele können unter [1] nachgelesen werden.

## Guards

Mit Guards können sich Angular-Anwendungen über Routenwechsel informieren lassen. Dabei handelt es sich lediglich um Services mit vorgegebenen Methoden, die der Router zu bestimmten Zeitpunkten aufruft. Diese Methoden können auch ins Routing eingreifen: Ihr zurückgelieferter Wert bestimmt, ob der Router den angeforderten Routenwechsel tatsächlich durchführen darf. Kann die Methode ihre Entscheidung augenblicklich bekannt geben, liefert sie einen Boolean. Um die Entscheidung hinauszuzögern, liefert sie zunächst lediglich ein *Observable<boolean>* oder einen *Promise<boolean>*. Steht die Entscheidung später fest, kann sie über diese Mechanismen den Router benachrichtigen. Dieses Vorgehen ist beispielsweise notwendig, wenn zur Entscheidungsfindung ein Web-API zu konsultieren oder Rücksprache mit dem Benutzer zu halten ist.

Für unterschiedliche Arten von Guards definiert Angular auch unterschiedliche Interfaces, die es zu implementieren gilt (Tabelle 1).

Wie man sich beim Betrachten dieser Interfaces denken kann, lassen sich gleich einige Arten von Querschnittsfunktionen mit Guards implementieren. Das nachfolgende Beispiel dient dem Schützen von Routen. Möchte ein Benutzer eine Route aktivieren, für die ihm die Berechtigungen fehlen, soll er auf die Log-in-Seite zurückgesendet werden (**Abb. 1**).

Dies dient weniger der Sicherheit, zumal Sicherheit bei browserbasierten SPAs immer im Backend zu realisieren ist. Vielmehr fördert es die Benutzerfreundlichkeit, da die Anwendung den Benutzer im Fall des Falles hierdurch zur Anmeldung auffordern kann. Beim Guard handelt es sich um einen einfachen Service, der den Typ *CanActivate* implementiert. Neben dem hier

| Interface | Methode | Beschreibung |
|---|---|---|
| CanActivate | canActivate | Legt fest, ob die gewünschte Route aktiviert werden darf |
| CanActivateChild | canActivateChild | Legt fest, ob bzw. welche Child Routes einer Route aktiviert werden dürfen |
| CanLoad | canLoad | Legt fest, ob ein Modul per Lazy Loading geladen werden darf |
| CanDeactivate | canDeactivate | Legt fest, ob eine Route deaktiviert werden darf |

Tabelle 1: Interfaces für Guards

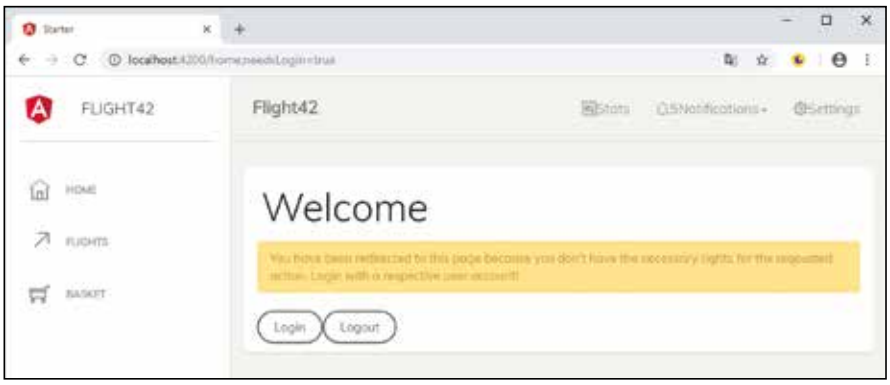


Abb. 1: Routen mit Guards schützen

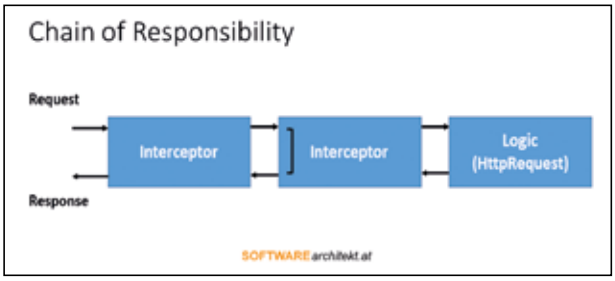


Abb. 2: Chain of Responsibility

aus Platzgründen nicht näher beschriebenen *AuthService* zum Anmelden von Benutzern lässt er sich auch dem Router injizieren. Damit leitet er Benutzer mit fehlenden Berechtigungen auf eine andere Route weiter (Listing 1).

Die von *CanActivate* vorgegebene gleichnamige Methode erhält vom Router einen *ActiveRouteSnapshot*, der über die gewünschte Route informiert, sowie einen *RouterStateSnapshot*, der über die aktuelle Route Auskunft gibt. Sie wendet sich an den *AuthService*, um herauszufinden, ob der aktuelle Benutzer angemeldet ist. Ist dem so, retourniert sie *true* und erlaubt somit die gewünschte Aktivierung. Ansonsten leitet sie mit der Methode *navigate* des Routers den Benutzer auf die Route „home“ weiter. Dabei übergibt sie einen Parameter, der die dahinterliegende *HomeComponent* wissen lässt, dass der Benutzer aufgrund fehlender Berechtigungen bei ihr gestrandet ist.

Um den Guard für eine Route zu aktivieren, muss sie mit ihrer Eigenschaft *canActivate* lediglich darauf verweisen. Damit eine Route mehrere solcher Guards nutzen kann, handelt es sich bei *canActivate* um ein Array (Listing 2).

## HTTP Interceptors

Der HTTP Client von Angular bietet mit Interceptoren einen Einsprungspunkt. Jeder bereitgestellte Interceptor hat die Möglichkeit, ausgehende HTTP-Anfragen sowie eingehende HTTP-Antworten zu manipulieren. Auf diese Weise lassen sich Anfragen um Authentifizierungsinformationen erweitern, aber auch Caching-Strategien implementieren oder Datenformate wie XML oder CSV verarbeiten. Die Idee hinter diesen Interzeptoren folgt dem Muster der Chain of Resposibility (**Abb. 2**).

### Listing 1

```
@Injectable({ providedIn: 'root'})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {
  }
  canActivate(route: ActivatedRouteSnapshot, state:
                                          RouterStateSnapshot): boolean
  {
    if (this.authService.isLoggedIn) {
      return true;
    }
    else {
      this.router.navigate(['/home', {needLogin: true}]);
      return false;
    }
  }
}
```

### Listing 2

```
const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: '',
    component: FlightBookingComponent,
    children: [
      {
        path: 'flight-search',
        component: FlightSearchComponent
      },
      {
        path: 'passenger-search',
        component: PassengerSearchComponent,
        canActivate: [AuthGuard]

      },
      {
        path: 'flight-edit/:id',
        component: FlightEditComponent,
      }

    ]
  }
];
```

Dieses Muster sieht vor, dass eine Aktion mit Zusatzlogiken erweitert wird. Letztere werden in Klassen untergebracht, deren Objekte zu einer Kette zusammengeschlossen werden. Am Ende dieser Kette befindet sich die eigentliche Aktion. Jedes dieser Objekte kümmert sich um seine Aufgabe und kann an das nächste Kettenglied delegieren.

Ein Beispiel für einen Interceptor, der jede ausgehenden Anfrage um einen beispielhaften Authorization-Header erweitert, findet sich in Listing 3. Außerdem leitet er den Benutzer auf die Startseite um, wenn die Antwort auf einen Securityfehler schließen lässt.

Wie das Beispiel zeigt, handelt es sich bei Interceptors um Services, die das Interface HttpInterceptor implementieren. Es gibt lediglich die Methode *intercept* vor, an die Angular die aktuelle Anfrage sowie das nächste Glied der Kette weitergibt. Um die Kontrolle an dieses weiterzugeben, ruft der Interceptor die Methode *next* auf. Das Ergebnis dieser Methode ist ein Observable mit der HTTP-Antwort. Dieses lässt sich mit den üblichen RxJS-Operatoren bearbeiten.

Bevor der hier gezeigte Interceptor die Anfrage um einen Authorization-Header erweitert, prüft er, ob es sich um ein vertrauenswürdiges Web-API handelt. Auf diese Weise stellt er sicher, dass solche vertrauenswürdigen Informationen nicht in die falschen Hände geraten.

Außerdem ist hier zu beachten, dass das API rund um den HTTP Client mit Immutables arbeitet. Das bedeutet, dass Interceptoren die Headers-Auflistung aber auch die Anfrage nicht verändern können. Stattdessen müssen sie diese Objekte klonen und im Rahmen dessen verändern. Bei den Kopfzeilen kümmert sich darum die Methode *set*. Anstatt die Auflistung zu verändern, liefert sie eine Kopie mit der zusätzlichen Kopfzeile. Bei der Anfrage kommt für dieselbe Aufgabe die Methode *clone* zum Einsatz.

Damit Angular den Interceptor verwendet, muss die Anwendung ihn für das Token *HTTP_INTERCEPTORS* registrieren (Listing 4).

### Listing 3

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private router: Router) {
  }

  intercept(req: HttpRequest<any>, next: HttpHandler):
                                        Observable<HttpEvent<any>> {

    if (req.url.startsWith('http://www.angular.at')) {
      const headers = req.headers.set('Authorization', 'asfjsdfjjkjru==');
      req = req.clone({ headers });
    }

    return next.handle(req).pipe(
      catchError(resp => this.handleError(resp))
    );
  }

  handleError(resp: HttpErrorResponse): Observable<HttpEvent<any>> {

    if (resp.status === 401 || resp.status === 403) {
      this.router.navigate(['/home', {needsLogin: true}]);
    }
    return throwError(resp);

  }
}
```

### Listing 4

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CityPipe,
  ],
  exports: [
    CityPipe,
  ]
})
export class SharedModule {
  static forRoot():
                                        ModuleWithProviders {
    return {
      ngModule: SharedModule,
      providers: [{
        provide: HTTP_
                        INTERCEPTORS,
        useClass: AuthInterceptor,
        multi: true
      }]
    }
  }
}
```

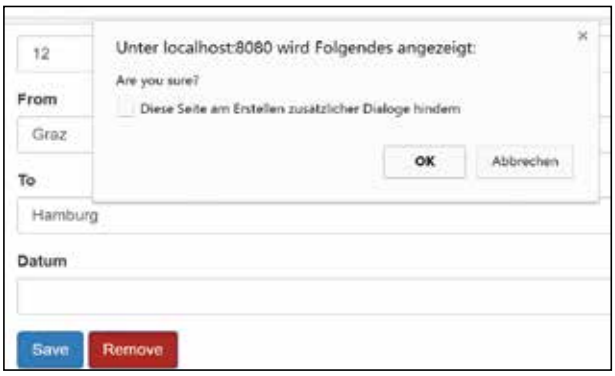


Abb. 3: Direktive für kritische Aktionen

Die Option *multi* ist hier auf *true* zu setzen, um Angular zu informieren, dass es mehrere Interceptors geben kann. All diese bilden die oben erwähnte Kette. Die Reihenfolge der Registrierungen repräsentieren die Reihenfolge der Glieder innerhalb der Kette.

## Direktiven

Bei Direktiven handelt es sich um die kleinen Geschwister der Komponenten. Sie fügen Verhalten zu bestimmten Elementen hinzu, ohne ein Template anzuzeigen. Die gewünschten Elemente adressieren sie über CSS-Selektoren. Das bedeutet, dass sie alle Elemente der Anwendung, die bestimmte Eigenschaften aufweisen, erweitern können. Genau deswegen eignen sie sich wunderbar für die Umsetzung von Querschnittsfunktionen.

Zur Demonstration soll eine für kritische Aktionen gedachte Alternative zum *click*-Ereignis bereitgestellt werden. Diese gibt zunächst nur eine Warnmeldung aus und stößt den hinterlegten Event Handler nur an, wenn diese bestätigt wurde (**Abb. 3**).

Eine Direktive definiert sich ähnlich wie eine Komponente: Es handelt sich dabei um eine Klasse, die Bindings aufweisen kann. Metadaten sind über den Dekorator Directive bereitzustellen. Dieser hat fast alle Eigenschaften, die man auch von Component kennt – lediglich templatebezogene Eigenschaften fehlen, zumal Direktiven eben keine Templates haben. Zu diesen Eigenschaften, die man vergeblich suchen wird, zählen *template* bzw. *templateUrl*, *styles* bzw. *styleUrls* und *viewProviders*.

Die hier betrachtete Direktive verwendet einen Selektor, der sämtliche Elemente mit dem Attribut *flightClickWithWarning* adressiert. Die Verwendung von Camel Case ist hier ebenso üblich wie der Einsatz eines projektspezifischen Präfix. Für letzteren fällt hier die Wahl abermals auf *flight* (Listing 5).

Der Name *flightClickWithWarning* wird nicht nur für den Selektor herangezogen, sondern auch für das Event, das nach einer eventuellen Bestätigung aufzurufen ist.

### Listing 5

```
@Directive({
  selector: '[flightClickWithWarning]'
})
export class FlightClickWithWarningDirective implements OnInit {

  // Darzustellende Warnung
  @Input() warning: string = 'Are you sure?';

  // Event-Handler, der nach Besätigung der Warnung
  // auszuführen ist
  @Output() flightClickWithWarning = new EventEmitter();

  constructor(
    private elementRef: ElementRef,
    private renderer: RendererV2) {

      // elementRef: Verweis auf aktuelles Element
      // renderer: Services zum Verändern von Elementen

  }

  ngOnInit() {
    // Warnung: Direkter DOM-Zugriff!
    // this.elementRef.nativeElement.setAttribute('class', 'btn btn-danger');

    // Indirekter DOM-Zugriff über Renderer
    this.renderer.setAttribute(this.elementRef.nativeElement, 'class', 'btn
btn-danger');
  }

  [...]
}
```

Eine Direktive definiert sich ähnlich wie eine Komponente: Es handelt sich dabei um eine Klasse, die Bindings aufweisen kann. Metadaten sind über den Dekorator Directive bereitzustellen.

Eine solche Vorgehensweise ist üblich, zumal sie es erlaubt, im selben Atemzug sowohl die Direktive auf eine Komponente anzuwenden als auch eine erste Bindung festzulegen:

```
<button
  (flightClickWithWarning)="remove()"
  [warning]="...">Remove</button>
```

Die Direktive lässt sich die aktuelle *ElementRef* injizieren. Dabei handelt es sich um ein Objekt, das das aktuelle Element referenziert. Das ist jenes Element, auf das die Direktive angewandt wurde. Im Fall des letzten Beispiels handelt es sich dabei um das *button*-Element.

Zum Verändern des Buttons lässt sie sich auch den aktuellen Renderer injizieren. Dessen Nutzung demonstriert der Lifecycle Hook *ngOnInit*. Seine Aufgabe besteht darin, die Klassen *btn* und *btn-danger* zum Button hinzuzufügen. Diese lassen es in einem warnenden Rot erstrahlen. Wie der Kommentar zeigt, könnte diese Aufgabe eine Direktive ohne Renderer bewerkstelligen, zumal eine *ElementRef* über ihre Eigenschaft *nativeElement* direktiven Zugriff auf das zugrundeliegende DOM-Element gewährt. Diese Vorgehensweise klappt jedoch nur, wenn Angular auf klassische Weise im Hauptthread des Browsers ausgeführt wird. Kommt Angular zum Beispiel serverseitig, in einer nativen Anwendung oder in einem Web-Worker zur Ausführung, steht das DOM-Element nicht zur Verfügung.

Um dem gerecht zu werden, sieht Angular vor, dass jede Plattform ihren eigenen Renderer definiert. Dieser kümmert sich um das korrekte Modifizieren von Elementen, wie dem im betrachteten Fall demonstrierten Hinzufügen von Klassen. Neben dem Verändern der adressierten Elemente müssen Direktiven häufig auch deren Ereignisse behandeln. Im hier verwendeten Fallbeispiel hat die *FlightClickWithWarningDirective* beispielsweise die Aufgabe, auf das *click*-Ereignis zu reagieren. Hierfür kommen *HostListener*, die ein Ereignis mit einer Methode verknüpfen, zum Einsatz:

```
@HostListener('click', ['$event'])
  handleClick($event): void {
    if (confirm(this.warning)) {
      this.flightClickWithWarning.emit();
    }
  }
```

Der hier betrachtete *HostListener* bringt bei jedem *click*-Ereignis des adressierten Elements die Methode *handleClick* zur Ausführung. Diese gibt einen Warndialog aus und stößt – sofern dieser bestätigt wird – das Ereignis *flightClickWithWarning* an. Wie auch Komponenten sind Direktiven über ein Modul zu deklarieren und auf Wunsch auch über *exports* anderen Modulen zur Verfügung zu stellen.

## Fazit

Idealerweise werden Querschnittsfunktionen automatisch aktiv, ohne dass der Programmcode sie immer und immer wieder anstoßen muss. Guards ermöglichen das im Zuge des Routings. Sie stellen Logiken bereit, die beim Routenübergang greifen und diese können den Routenvorgang auch unterbinden. Damit lassen sich zum Beispiel Autorisierungslogiken implementieren. HTTP-Interzeptoren werden hingegen bei jedem HTTP-Aufruf tätig und können ausgehende Anfragen sowie eingehende Antworten manipulieren. Anwendungsgebiete sind das Übermitteln von Authentifizierungsinformationen, Caching oder Fehlerbehandlungen. Die ein wenig im Schatten der Komponenten stehenden Direktiven erlauben, zusätzliches Verhalten zu allen Elementen der Anwendung hinzuzufügen, die bestimmte Eigenschaften aufweisen.

**Manfred Steyer** unterstützt als Trainer und Berater Softwareteams im gesamten deutschen Sprachraum bei der Entwicklung mit Angular. Er wurde als Google Developer Expert ausgezeichnet und schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. In seinem aktuellen Buch zu Angular behandelt er die vielen Seiten des populären JavaScript-Frameworks aus der Feder von Google.

www.softwarearchitekt.at

## Links & Literatur

[1] https://github.com/manfredsteyer/CrossCutting.git

Der neue ValueType in Java

# Warum Werttypen wichtig sind

Bisher kennt Java zwei verschiedene Datentypen: primitive Datentypen und Objekte. In den kommenden Releases wird es einen neuen Typ geben, den ValueType. Für die tägliche Programmierung sollte es keinen Unterschied zwischen Objekten und Werttypen geben, unter der Haube sind sie jedoch ziemlich unterschiedlich. ValueTypes sind wie Objekte, aber funktionieren wie primitive Datentypen. Das heißt, sie sind effektiv.

von Peter Verhas

Andere Sprachen nutzen ebenfalls diese Art der Datenverarbeitung, obwohl sie sie normalerweise anders bezeichnen. In diesem Artikel beschreibe ich die Problematik von Objekten, die es notwendig macht, ValueTypes (oder Werttypen) in Java aufzunehmen. Anschließend erkläre ich, was ValueTypes sind, und gehe schließlich auf ihre Problematik ein. Ja, auch Werttypen unterliegen einer Problematik, und der Grund dafür ist ein sehr elementarer: Das Leben ist kein Ponyhof, und man kann nicht alles haben. Es gibt bei der Verwendung von ValueTypes gegenüber Objekten Vorteile, aber ebenso existieren Anwendungen beziehungsweise Programmierkonstrukte, bei denen Objekte besser passen. Im letzten Teil des Artikels erkläre ich außerdem, warum es Einschränkungen bei Werttypen gibt, wie z. B. keine Vererbung, Mangel an generischen Elementen und Unveränderlichkeit.

## Objekte in Java

Objekte in Java sind kleine Speicherteile, die in der Regel in einem Segment des Speichers namens Heap gespeichert werden. Der Speicher wird beim Erstellen des Objekts zugewiesen und freigegeben, wenn das Objekt nicht mehr verwendet und „weggeräumt“ wird (Garbage Collection). Während der Lebensdauer des Objekts kann der Speicher, der das Objekt repräsentiert, im Verlauf des Garbage-Collection-Prozesses von einem Ort zum anderen verschoben werden. Auf diese Weise verwaltet Java den Speicher und stellt sicher, dass unabhängig davon, wie Objekte erstellt und zerstört werden, der Speicher nicht segmentiert wird. Andere Sprachen mit Garbage Collector, die den „Objektmüll“ zwar wegräumen, aber nicht komprimieren, laufen Gefahr, dass lange laufende Prozesse den Speicher segmentieren. Aber: Komprimieren ist ein CPU-intensiver Prozess, deshalb konkurriert Java GC kaum mit der Geschwindigkeit der Garbage Collection (GC) der Programmiersprache Go. Vor allem aber macht die Tatsache, dass sich die Objekte im Speicher bewegen, Zeiger nutzlos. Es wird zwar irgendwohin verwiesen, die Daten können sich jedoch nach einer Komprimierungsphase von GC bereits irgendwo anders befinden. Das ist der Grund, warum Java keine Zeiger verwendet – Java verwendet Referenzen.

Man könnte sich fragen, worin der Unterschied zwischen Zeiger und Referenz besteht. Kurz gesagt: Bei Referenzen handelt es sich um verwaltete Zeiger. Wenn GC ein Objekt im Speicher verschiebt, müssen alle Referenzen aktualisiert werden, damit sie jederzeit auf das richtige Objekt verweisen. Die Integration von nativem Code ist ebenfalls etwas umständlich. Java GC kann nicht wissen, wohin im durch den nativen Code verwalteten Speicher der Zeiger kopiert wurde. Außerdem gibt es in Java keine Zeigerarithmetik. Es existieren zwar Arrays, aber ansonsten kann man die schöne Zeigerarithmetik vergessen, an die man sich bei der Programmierung in C oder C++ vielleicht bereits gewöhnt hat. Der Vorteil ist, dass auch durch fehlerhafte Zeigerwertberechnungen verursachte Speicherfehler ausgeschlossen werden können.

Dennoch befinden sich die Objekte im Speicher, und wenn der Prozessor damit rechnen muss, muss das Objekt vom Hauptspeicher zum Prozessor gelangen. In den guten alten Zeiten, in denen die CPUs mit 4 MHz liefen, war das kein Problem. Die Geschwindigkeit des Speicherzugriffs war vergleichbar mit der Geschwindigkeit des Prozessors. Heute laufen Prozessoren mit 4 GHz, und der Speicherzugriff ist kaum schneller als früher. Die Technologie ist nicht schlecht, es handelt sich hierbei einfach um Physik. Man muss nur die Zeit berechnen, die benötigt wird, um von der CPU zum Speicher und mit Lichtgeschwindigkeit wieder zurück zu gelangen, das ist alles. Es gibt nur einen Weg, die Geschwindigkeit zu erhöhen, nämlich den Speicher näher an die Verarbeitungseinheit zu bringen. Und genau das ist es, was moderne CPUs auszeichnet: Sie verfügen über Speichercaches auf der CPU selbst. Leider ist nicht nur die CPU-Geschwindigkeit, sondern auch der Speicherbedarf gestiegen. Früher hatten wir 640 kB auf einem Computer, was für alles reichen musste. Heute hat mein Mac 16 GB. Und hier kommt wieder die Physik ins Spiel: Man kann nicht 16 GB oder mehr auf die CPU legen, da dort kein Platz ist und es auch keine effektive Methode gibt, das System entsprechend zu kühlen. Und wir wollen die CPU schließlich zum Rechnen nutzen, nicht zum Kochen.

Wenn die CPU Speicherplatz braucht, wird im Cache gespeichert. Wenn ein Programm etwas von einem Speicherort abruft, ist es wahrscheinlich, dass es bald etwas vom anderen Speicherort abruft und so weiter. Deshalb liest die CPU ganze Speicherseiten in den Cache ein. Im Cache können sich viele Seiten aus verschiedenen Speicherbereichen befinden. Wenn wir auf ein Element in einem Array zugreifen wollen, kann das lange dauern (aus CPU-Sicht bedeutet „lange dauern“ ein paar Dutzend Nanosekunden), wenn wir jedoch ein zweites Element aus dem Array brauchen, befindet es sich bereits im Cache. Das ist sehr effektiv, es sei denn, es handelt sich um ein Array von Objekten. In diesem Fall ist das Array selbst ein zusammenhängender Bereich von Referenzen. Die CPU, die auf das zweite Element im Array zugreift, hat die Referenz im Cache, das Objekt selbst kann sich jedoch auf einer völlig anderen Seite befinden als auf der, auf der sich das erste Objekt befand. Es existieren einige Techniken zur Optimierung des Speicherlayouts, die dieses Problem teilweise lösen, aber der Knüller wäre es, wenn die Objekte im Speicher wie die Hühner auf der Stange aufgereiht gespeichert würden. Nur geht das leider nicht, denn selbst wenn sie nacheinander aufgereiht wären, wären sie durch den sogenannten Objektheader getrennt.

Der Objektheader befindet sich einige Bytes vor dem Objektspeicher, der das Objekt beschreibt. Er beinhaltet die Sperre, die in synchronisierten Anweisungen verwendet wird, und auch den Typ des Objekts. Wenn wir eine Referenz in einer Variable haben, die z. B. vom Typ *Serializable* ist, kennen wir den tatsächlichen Typ des Objekts nicht von der Variable selbst. Wir müssen uns Zugriff auf das Objekt verschaffen, damit die JRE den tatsächlichen Typ des Objekts lesen kann. In Java hilft dieser Objektheader bei Vererbung und Polymorphie. Auch betragen die wenigen Bytes in den 32-Bit-Implementierungen 12 Bytes und 16 Bytes auf der 64-Bit-Architektur. Das bedeutet, dass ein Integer einen 32-Bit-int-Wert und zusätzlich 128 Bit an administrativen Bits speichert – ein Verhältnis von 4:1.

## ValueType

Werttypen versuchen, dieses Problem anzugehen. Ein ValueType ist insofern etwas wie eine Klasse, als er Felder und Methoden haben kann. ValueTypes werden im Rahmen des Valhalla-Projekts unter JEP 169 für Java entwickelt. Derzeit ist eine Early-Access-Version verfügbar. Diese Version ist ein Abkömmling der Version Java 11 und hat noch Einschränkungen. Die Syntax ist noch vorläufig (mit einigen Schlüsselwörtern, die mit einem doppelten Unterstrich beginnen und nicht in der endgültigen Version enthalten sein können), und auch einige Funktionen sind nicht implementiert. Dennoch gibt es die Möglichkeit, das Ganze auszuprobieren.

Der ValueType unterscheidet sich von einem Objekt dadurch, dass er keinen Objektheader und keine Identität hat, es keine Referenzen auf ihn gibt, Werttypen unveränderlich sind und es keine Vererbung zwischen Werttypen und somit auch keine Polymorphie gibt. Manches davon – wie das Fehlen eines Objektheaders – sind Implementierungsdetails, anderes wiederum Designentscheidungen. Kommen wir zu einigen Merkmalen von ValueTypes.

## Keine Identität

ValueTypes haben keine Identität. Wenn wir es mit Objekten zu tun haben, können zwei davon identisch sein. Im Grunde genommen sprechen wir nur zweimal über

das gleiche Objekt und Objekte können gleich sein. Im letztgenannten Fall haben wir zwei verschiedene Objekte – es handelt sich jedoch um Instanzen der gleichen Klasse, und die Methode *equals()* gibt *true* zurück, wenn wir sie vergleichen. Die Identität wird in Java mit dem Operator == überprüft, die Gleichheit, wie bereits erwähnt, mit der Methode *equals()*.

Primitive Datentypen wie *Byte*, *char*, *short*, *int*, *long*, *float*, *double* oder *boolean* haben ebenfalls keine Identität. In diesem Fall ist das ziemlich offensichtlich. Es ist Unsinn, zu sagen, dass zwei Boolesche Werte beide *true*, aber dennoch unterschiedliche Instanzen sind. Als logische Werte haben auch Zahlen wie Null, Eins oder Pi keine Instanzen. Sie sind Werte. Wir können sie mit dem Operator == auf Gleichheit und nicht auf Identität prüfen. Die Idee der Werttypen ist es, den Satz dieser acht primitiven Werte um programmdefinierte Typen zu erweitern, die ebenfalls Werte darstellen.

## Keine Referenzen

Die Werte werden in den Variablen und nicht im Heap gespeichert, und wenn die Bitdarstellung des Werts sich in einer Variable befindet, wird der Compiler den Typ erkennen. Ebenso erkennt er, dass die Bits in einer Variablen als 32-Bit-Ganzzahl mit Vorzeichen behandelt werden sollten, wenn die Variable vom Typ *int* ist. Es ist auch wichtig, zu beachten, dass, wenn ein ValueType als Argument an eine Methode übergeben wird, die Methode die „Kopie“ des ursprünglichen ValueTypes erhält. Das liegt daran, dass Java alle Argumente nach Wert und nie nach Referenz übergibt, was sich mit der Einführung von Werttypen auch nicht ändert. Wenn ein ValueType als Argument an einen Methodenaufruf übergeben wird, werden alle Bits des ValueTypes in die lokale beziehungsweise in die Argumentvariable der Methode kopiert.

## Kein Objektheader

Da Werttypen Werte sind, die in den Variablen und nicht im Heap gespeichert werden, benötigen sie keinen Header. Der Compiler weiß einfach, was für ein Typ eine Variable ist und wie das Programm mit den Bits in dieser Variable umgehen soll. Das ist von entscheidender Bedeutung, wenn die ValueType-Arrays ins Spiel kommen. Genau wie bei primitiven Datentypen werden beim Erstellen eines Arrays mit einem Werttyp die Werte im Speicher nacheinander gepackt. Das bedeutet, dass wir bei ValueTypes nicht das Problem haben werden, das bei Objektarrays auftritt. Es gibt keine Referenzen auf die einzelnen Elemente des Arrays, und sie können nicht im Speicher verteilt werden. Wenn die CPU das erste Element lädt, lädt sie alle Elemente, die sich auf der gleichen Speicherseite befinden. Der Zugriff auf nachfolgende Elemente nutzt den Vorteil des Prozessorcache.

## Keine Vererbung

Es könnte eine Vererbung zwischen Werttypen geben, aber es wäre äußerst schwierig, sie durch den Compiler zu verwalten. Außerdem würde das Ganze nicht viele Vorteile bringen. Ich wage zu behaupten, dass das Zulassen von Vererbung nicht nur Probleme für den Compiler verursachen, sondern auch unerfahrene Programmierer dazu anregen würde, Konstrukte zu erstellen, die mehr Schaden als Nutzen brächten. In der kommenden Java-Version, die ValueTypes unterstützt, wird es keine Vererbung zwischen ValueTypes und auch nicht zwischen Klassen und ValueTypes geben. Dabei handelt es sich um eine Designentscheidung. Statt uns in vielen Erklärungen zu verlieren, betrachten wir einfach einige Beispiele.

Enthält eine Klasse *C* ein Feld vom Typ einer anderen Klasse *P*, enthält sie ebenfalls eine Referenz auf diese andere Klasse. Es kann auch sein, dass *C* der Klasse *P* untergeordnet ist. Das ist kein Problem. Beispielsweise gibt es eine verknüpfte Liste mit *P*-Instanzen. Es existiert ein Feld namens Next, das entweder „Null“ ist oder den Verweis auf das nächste *P* in der Liste enthält. Wenn die Liste auch Instanzen von *C* enthalten kann (zur Erinnerung: *C* erweitert *P*), dann hat C auch eine Referenz auf das nächste *P*. Die Liste kann *C*-Instanzen enthalten, da, wie die Vererbung impliziert, ein *C* auch ein *P* ist.

Wie sieht es aus, wenn *P* ein ValueType ist? Wir können die Elemente nicht miteinander verbinden. Es gibt keine Referenz auf das nächste *P*, da es keine Referenz auf einen ValueType gibt. Klassen können über Feldwerte gegenseitig aufeinander referenzieren. Werttypen implementieren nur Containment. Wenn ein ValueType ein Feld hat, das einem anderen ValueType entspricht, enthält er alle Bits dieses anderen ValueTypes. Ein Werttyp kann daher niemals ein Feld enthalten, das der Typ selbst ist. Denn das würde bedeuten, dass sich der Werttyp selbst enthält, was eine unendliche Rekursion in der Definition des Typs ist. Ein solcher Typ wäre unendlich groß, daher ist er in der Spezifikation ausdrücklich verboten. Wenn ValueTypes voneinander erben könnten, wäre die Einschränkung komplexer. In diesem Fall könnten wir nicht einfach sagen, dass sich ein ValueType nicht selbst enthalten darf. Es hätte jeder andere Werttyp verboten werden müssen, der vom aktuellen Werttyp abstammt.

Ein Werttyp kann niemals ein Feld enthalten, das der Typ selbst ist. Denn das würde bedeuten, dass sich der Werttyp selbst enthielte: eine unendliche Rekursion in der Definition des Typs.

Man muss versuchen, sich eine Variable vorzustellen, die vom Typ *V* ist. Eine solche Variable sollte groß genug sein, um alle Bits von *V* zu beinhalten, aber auch groß genug, um alle Bits von *K* aufzunehmen, wenn es möglich wäre, Wertobjekte zu erweitern. *K* würde dann *V* hypothetisch erweitern. In diesem Fall enthält *K* alle Bits von *V* und seine eigenen. Wie viele Bits sollte eine Variable vom Typ *V* haben? Die Anzahl der Bits von *V*? Dann könnten wir keinen *K*-Wert darin speichern, *K* würde nicht passen. Alle Variablen vom Typ *V* sollten also groß genug sein, um auch *K* zu enthalten. Aber wir sollten nicht bei *K* aufhören, da es mehr ValueTypes geben könnte, die *K* erweitern – unter der Annahme, dass eine Vererbung stattgefunden hat, was aber nicht der Fall ist. In diesem Fall sollte eine Variable vom Typ *V* so viele Bits haben, wie das größte untergeordnete Element von *V* haben könnte, was zum Zeitpunkt der Kompilierung unbekannt ist. Bekannt wird es nur und erst, wenn alle ValueTypes geladen sind.

## Keine Polymorphie

Da es keine Vererbung gibt, kann es keine Werttyppolymorphie geben. Es existieren jedoch noch weitere Gründe, die darauf hindeuten, dass es nicht sinnvoll ist, eine Polymorphie für Werttypen zu implementieren. Betrachten wir das obige Beispiel und stellen wir uns die Variable vor, die alle Bits des größten untergeordneten Elements von *V* enthalten kann. Riefen wir eine Methode für diese Variable auf, welche sollte es (während der Laufzeit) sein? Die Variable enthält keine Informationen über den Typ des tatsächlichen ValueTypes. Ist es *V*, ist es *K* oder ein anderes untergeordnetes Element? Der Compiler muss wissen, welche Methode er aufrufen soll, da es keine Headerinformationen gibt, die den Typ signalisieren würden, der zu diesem Zeitpunkt in der Variable ist.

**Listing 1**

```
package javax0.valuetype;

public __ByValue class Point {
  public int x;
  public int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Point pushedRight(int d) {
    return __WithField(this.x, x+d);
  }

  public String toString() {
    return "[" + x + "," + y + "]";
  }
}
```

## Unveränderlichkeit

Unveränderlichkeit ist eine Designentscheidung, aber eine nachvollziehbare, sozusagen natürliche. Werttypen in Java sind unveränderlich. Unveränderlichkeit ist in der Regel eine gute Sache. Unveränderliche Objekte helfen dabei, auf saubere und threadsichere Weise Code zu schreiben. Unveränderlichkeit löst nicht alle Probleme, aber sie ist oft praktisch. Auch wenn man *int* als Zahl sieht, ist es ziemlich offensichtlich, dass man ihren Wert nicht ändern kann. Wenn eine Variable den ganzzahligen Wert 2 enthält, kann man den in der Variable gespeicherten Wert ändern, aber nicht den Wert selbst auf 3. Könnte man, würde im ganzen Universum plötzlich 2 mal 2 mal 2 gleich 9 ergeben. Ähnlich sieht es bei Werttypen aus. Man kann den Inhalt der Variable ändern, die den ValueType enthält, nicht jedoch den ValueType selbst. Wenn man ein einzelnes Bit ändert, hat man diesem Ansatz zufolge einen neuen Werttyp angelegt und den neuen Wert anstelle des alten gespeichert. Schauen wir uns das einfache Beispiel in Listing 1 an.

Dies ist ein einfacher ValueType, der eine vorläufige Syntax verwendet, die mit der Version *build 11-lworldea+0-2018-07-30-1734349.david.simms.valhalla* der Early-Access-Version von Java kompiliert wurde. Damit wird das „Schieben“ eines Punkts entlang der X-Achse möglich. Das Hauptprogramm, das den Werttyp verwendet, macht das, was in Listing 2 zu sehen ist.

Wenn wir *pushedRight* aufrufen, hat die Variable *a* einen neuen Wert. Der Punkt *(3,4)* hat sich jedoch nicht bewegt, der zweidimensionale Raum wurde nicht verzerrt. Dieser Punkt bleibt dort für immer, nur der Variablenwert wird geändert. Wenn wir jetzt versuchen, die Zeile *a = a.pushedRight(1);* auf *a.x = 4;* zu ändern, erhalten wir einen Kompilierungsfehler, der besagt:

```
/.../src/javax0/valuetype/Main.java:7: error: cannot assign a value to final
                                        variable x
```

Dabei ist zu beachten, dass das Feld *x* nicht als endgültig deklariert wurde, sondern automatisch als endgültig gilt, da es sich in einem Werttyp befindet.

Die Unveränderlichkeit als Merkmal steht in starkem Zusammenhang mit der Tatsache, dass es keine Referenzen auf einen ValueType geben kann. Java könnte uns theoretisch erlauben, ein Feld eines ValueTypes zu modifizieren. Das Ergebnis wäre aber im Wesentlichen das gleiche: Wir erhalten einen anderen Wert. Auf diese Weise ist die Unveränderlichkeit bei Werttypen keine Einschränkung. Es geht nur darum, wie wir unser Programm schreiben und wie wir ValueTypes sehen. Wenn man sie als Werte (wie Zahlen) betrachtet, die von Natur aus unveränderlich sind, wäre das sozusagen eine gesunde Denkweise.

## Probieren Sie es aus

Sie können den Early-Access-Release ausprobieren, indem Sie die JEP-Homepage [1] besuchen und außerdem die EA-Version [2] herunterladen. Die Builds sind für Linux, macOS und Windows für die x64-Plattform verfügbar. Die Installation ist zwar nicht so einfach wie bei den Produkten für den Handel, aber für erfahrene Java-Entwickler sollten einige Umgebungsvariableneinstellungen und das manuelle Extrahieren und Verschieben von Dateien an den richtigen Ort kein Problem sein. Sie können versuchen, Eclipse, IntelliJ oder Notepad zu verwenden, um den Quellcode zu bearbeiten, aber zumindest konnte ich mit der IntelliJ-2019.EA-Edition den Code nicht erstellen. Obwohl es sich bei dem Code um einen Fork aus dem Java-11-Quellcode handelt, erkennt IntelliJ ihn als Java 13, was übrigens ein kleiner Hinweis darauf ist, wo wir in der freigegebenen Version Werttypen erwarten können. Die Kompilierung des Codes kann manuell durch Ausführen des Befehls *javac* über die Befehlszeile und dann des Befehls *java* zum Starten der JVM erfolgen. Ich bin mit dem IntelliJ-extrahierten Ant-Skript klargekommen, obwohl ich mit Maven besser vertraut bin als mit Ant. Beim Herumspielen habe ich einige unerklärliche Kompilierungsfehler festgestellt, wobei der Code letztendlich fehlerhaft war. Nach Änderung des Codes funktionierte mit der Kompilierung alles gut. Andere Male, als der Fehler definitiv nicht bei mir lag, waren die Fehlermeldungen gut beschrieben und leicht verständlich.

## Zusammenfassung

Java entwickelt sich rasant. In den vergangenen zwei Jahren hat sich viel getan, und im Moment sind viele neue Entwicklungen im Gange, die uns in der Zukunft zur Verfügung stehen. Dazu gehört das Merkmal ValueType. Dieser Artikel beschreibt die wichtigsten Aspekte dieses Merkmals und gibt einen kurzen Ausblick auf diese neue Technologie. Java ist zudem eine der wichtigsten Programmiersprachen, vielleicht die zweithäufigste Programmiersprache im professionellen Umfeld nach COBOL. Als Java-Entwickler ist einem der Arbeitsplatz so gut wie sicher. Ständiges Lernen ist jedoch unabdingbar, wenn man professionell und up to date sein möchte. Glücklicherweise handelt es sich bei Java um Open Source und es existieren eine frei verfügbare Mailingliste, Dokumentationen, Quellcode und Testversionen von zukünftigen Versionen, lange bevor sie veröffentlicht werden. Ein leidenschaftlicher Entwickler sollte sich diese ansehen, die EA-Versionen zum Experimentieren verwenden und bereits zum Zeitpunkt des GA-Releases für den Handel die neuen Möglichkeiten kennen. Also los! Schnappen Sie sich die Valhalla-EA-Version von Java und probieren Sie sie aus.

**Peter Verhas** ist Senior Software Architect bei EPAM Schweiz. Er hat mehr als zehn Jahre Erfahrung in der Java-Entwicklung und mehr als zwanzig Jahre Erfahrung mit C und anderen Programmiersprachen. Zudem ist er Autor der Bücher „Java Projects“, „Mastering Java 9“ und „Java 9 Programming By Example“. Er bloggt außerdem regelmäßig in englischer Sprache (bei DZONE, Java Code Geeks und seinem eigenen Blog Javax0.wordpress.com). Peter hat einen Master in Elektrotechnik und studierte an der TU Budapest, der TU Wien und der TU Delft. Er arbeitete für Unternehmen wie Digital Equipment Corporation, T-Mobile und unterstützte die Telekommunikations- und Finanzbranche. Er war kurzzeitig Lehrer an der TU Budapest. Peter veröffentlicht auch Open-Source-Programme auf GitHub und ist Autor des ScriptBasic-Interpreters.

www.github.com/verhas

### Listing 2

```
package javax0.valuetype;

public class Main {

  public static void main(String[] args) {
    Point a = new Point(3,4);
    a = a.pushedRight(1);
    System.out.println(a);
  }
}
```

## Links & Literatur

[1] https://openjdk.java.net/jeps/169

[2] https://jdk.java.net/valhalla/