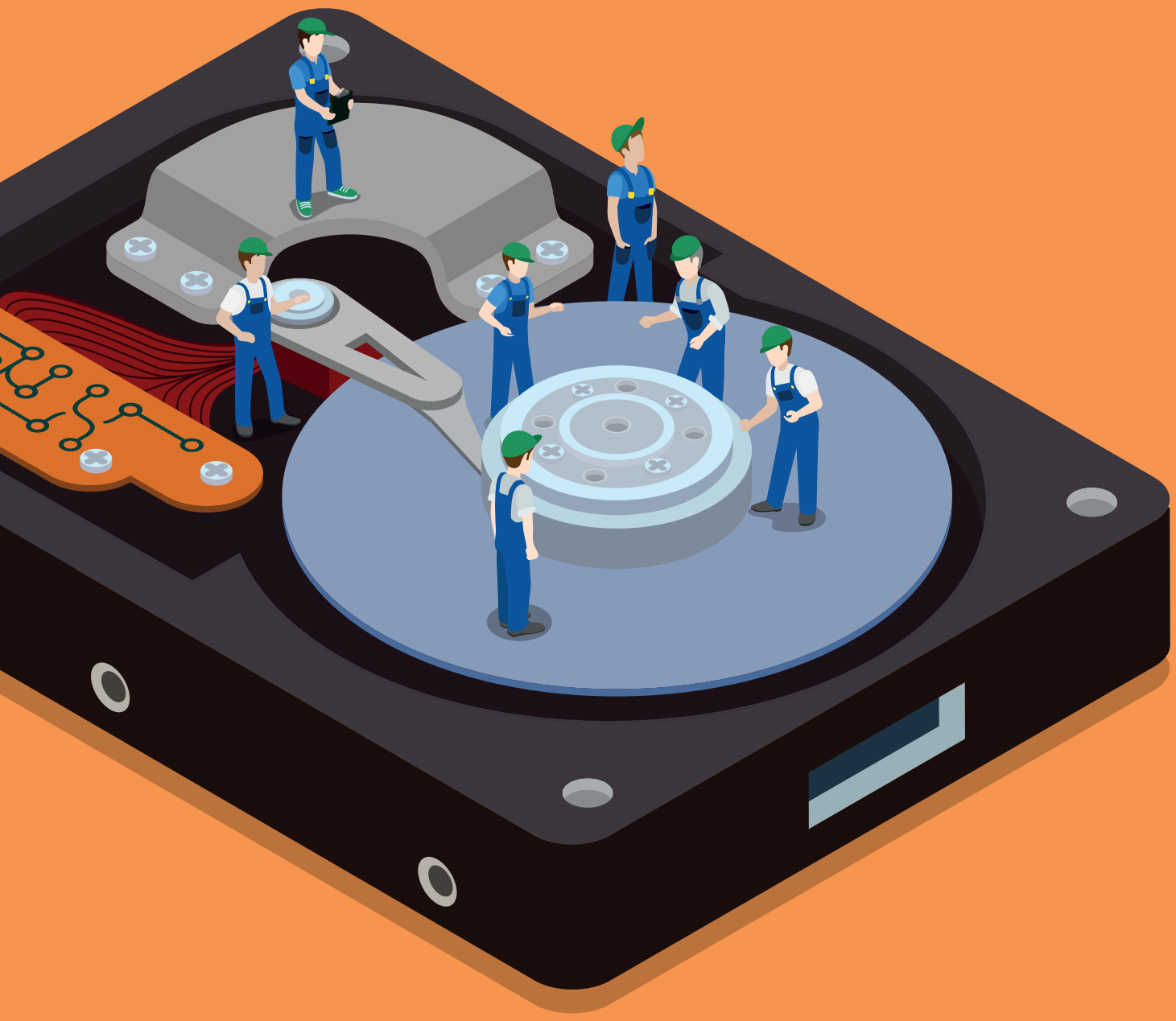

Microservices aus anderen Gründen – ein Erfahrungsbericht

Daniel Clasen und Jan Nonnen, Viaboxx GmbH

Dieser Artikel stellt ein Medizinsoftware-Projekt mit Spring Cloud vor, in dem die gesamte Architektur als komponiertes System realisiert wurde. Er zeigt die Erfahrungen der Autoren, erzählt von den Herausforderungen und hinterfragt kritisch die Vor- und Nachteile der gewählten Architektur. Bei Softwareprojekten in der Medizinbranche sind auch fachliche, rechtliche und prozessbedingte Herausforderungen zu beachten, die Architektur und Design maßgeblich formen und prägen.

Dank Spring Cloud *[1]* ist es heutzutage einfach, mit dem Spring Framework und Java eine Microservice-Architektur aufzubauen. Auch wenn man überall von Microservices hört, sollte man stets überlegen, ob je nach Anforderungen und Projekt so eine Architektur verwendet werden sollte oder nicht. Die Autoren präsentieren ihre Erfahrungen mit dem Einsatz solch einer Architektur in einem Medizinsoftware-Projekt nach dem GAMP-5-Regelwerk *[2]*, das sie mit Spring Cloud Netflix und Angular JS realisiert haben.

In dem Projekt waren Eigenschaften wie Datensicherheit, isolierte Testmöglichkeit und unabhängige Entwicklung wichtiger als die Größe und Hochverfügbarkeit der Komponenten. Durch diese Priorisierung hat sich im Laufe der Zeit die Sicht der Autoren als Software-Entwickler auf Microservices geändert. In anderen Projekten hatten sie bereits Microservices eingesetzt, insbesondere wegen ihrer Unabhängigkeit und dezentralen sowie horizontalen Skalierbarkeit.

Das Medizinprojekt war ein Greenfield-Projekt, das in erster Linie als Informationssystem von einer ganzen Medizinfirma genutzt werden sollte. Bereits im Rahmen der Anforderungserhebung stellte sich jedoch heraus, dass die verschiedenen Abteilungen das frühere Informationssystem komplett unterschiedlich nutzten und verwendete Begriffe anders verstanden. Durch diese Erkenntnis wurde die Idee bestärkt, keinen hochverfügbaren Monolithen zu entwickeln, sondern das Projekt als einen verteilten Workflow zu realisieren und mit einer Microservice-Architektur auch die Einzelbausteine einfacher anpassbar und skalierbar zu gestalten.

Von Microservices zu Modulen

Da man als agiles Unternehmen auch in diesem Projekt agil und iterativ arbeiten wollte, gab es zuerst eine grobe Anforderungserhebung mit Domain Driven Design, um die Begriffe und Abgrenzungen zwischen den Abteilungen zu erkennen und fachliche Schnittstellen zwischen den Komponenten zu definieren. Vor dem ersten Implementierungssprint fanden Verfeinerungssprints statt, um die für die Implementierung benötigten Spezifikationen und Dokumente gemeinsamen mit dem Kunden zu erstellen und aus den fachlichen Schnittstellen technische Komponentenschnittstellen zu entwerfen.

Im Laufe der Zeit entstanden, basierend auf den verschiedenen Abteilungen, fachliche Module, die jeweils ein eigenständiges Frontend besitzen und größere unabhängige Komponenten darstellen. Bei diesem Ansatz gibt es immer den Konflikt zwischen den Software-Entwicklern, die gerne mit möglichst vielen unabhängigen, kleinen, wiederverwendbaren Services arbeiten, und dem Qualitätsmanagement (QM), das sich möglichst wenig installierte Komponenten wünscht, da jede Installation und Änderung jede Menge Dokumentation, Validierung und Aufwand für die Administratoren bedeutet. Im Nachhinein ist das Ergebnis daher eine Mischung aus Microservices und einer dezentralen, serviceorientierten Architektur, die nach den Erfahrung der Autoren gut zusammenspielen und insgesamt eine in ihren Augen moderne, serviceorientierte Architektur darstellen.

Durch den iterativen Projektansatz konnten Anpassungswünsche des Kunden in späteren Iterationen übernommen und es mussten jeweils nur einzelne Services neu herausgegeben und installiert werden. Insbesondere für den Datenschutz stellte sich der Ansatz als sehr gut heraus, so ließen sich kritische Dinge wie Patientendaten in speziellen Services wegkapseln und speichern.

Außerhalb der kritischen, personenbezogenen Services sind diese Daten nur als Pseudonym referenziert. Zudem können Module mehrfach installiert und auch physikalisch getrennt werden. So ist es beispielsweise möglich, je ein Patienten-Modul pro Land zu betreiben, in dem Patientendaten erhoben werden. Das entschärft viele Risiken, die mit personenbezogenen Daten in einer Medizinsoftware einhergehen, und stellt die Einhaltung gesetzlicher Datenschutz-Richtlinien sicher. Nachteilig bei dieser Kapselung von Daten ist, dass Zusammenhänge, die über mehrere Module hinweg aggregiert werden müssen, Kommunikation und somit Netzwerklast zwischen den Modulen verursachen.

Ein weiteres Problem bei der gewählten Architektur und dem agilen Projektansatz ist die Einheitlichkeit beziehungsweise Konsistenz zwischen den Modulen. Da Module teils in unterschiedlichen Feature-Versionen produktiv betrieben werden, muss seitens der Spezifikation und Entwicklung ein besonderes Augenmerk auf

Kompatibilität und Nachziehen von Verbesserungen in anderen Modulen gerichtet und beachtet werden. Ein Feature muss langsam durch die produktiven Module durchsickern, bis es überall einheitlich vorhanden ist, was nachfolgend genauer beleuchtet wird.

Segregation of the UI

Aufgrund der Anforderungen des QM sowie der fachlich relevanten Anforderungen hinsichtlich der Modularität und Unabhängigkeit aller Module fiel die Entscheidung, wie eingangs erwähnt, die Frontends mittels Angular als modulare, eigenständige Web-Apps zu entwickeln. Dass dies einfacher gesagt als getan ist, war bereits vor der Entwicklung klar, wenn auch nicht im vollen Umfang. Die Unterteilung in diverse Frontends bringt zwar den Vorteil mit, dass diese sich später iterativ weiterentwickeln lassen und folglich auch separat getestet werden können, jedoch darf beim Benutzer nicht der Eindruck verschiedener Anwendungen entstehen.

Das Look and Feel sollte also über alle Module hinweg einheitlich sein, sich aber gleichzeitig unabhängig voneinander weiterentwickeln können. Diesen besonderen Anforderungen an die Entwicklung konnte man zum Großteil über die Auslagerung von Frontend-Komponenten wie Eingabefelder, Tabellen-Ansichten, Header, Footer, Layout etc. gerecht werden. Eine Gratwanderung, da durch eine maximale Wiederverwendbarkeit auch ein hoher Abstraktionsgrad entsteht, der ausgiebig getestet werden möchte.

Die ausgelagerte Frontend-Bibliothek wurde in einem separaten „git"-Repository entwickelt, mit „npm" versionisiert und auf einem privaten „npm"-Repository gehostet. Somit können Änderungen an den gängigen Komponenten an einer Stelle implementiert werden, ohne dass dies direkt eine Auswirkung auf alle Frontends hat, denn in jedem Frontend wird eine bestimmte Version der Komponenten-Bibliothek verwendet. Durch die Vermeidung von dupliziertem Code, den man oft in Microservice-Architekturen findet, lässt sich auch der manuelle Testaufwand verringern, da die Komponenten wiederverwendet anstatt neu geschrieben werden.

Wie bei den meisten Angular-Web-Apps findet die Kommunikation mit dem Backend mittels JSON über HTTPS statt. Explizit handelt es sich hier um JSON+HAL, also beschreibende Metadaten und Links auf relevante Ressourcen, die durch Spring HATEOAS *[3]* bei der Serialisierung und Deserialisierung der Kommunikations-Objekte aufbereitet werden. Das REST-API jedes einzelnen Moduls ist zur Wahrung der Kompatibilität bei zukünftigen Änderungen sowohl im Quellcode als auch in der Schnittstelle per URL-Parameter versionisiert. Ein typischer Endpunkt zum Abfragen der Modul-Eigenschaften sieht zum Beispiel so aus: „GET /api/v1/info".

Das REST-API jedes einzelnen Moduls ist im Java-Backend jeweils in ein eigenes JAR ausgelagert, das nur die Ressourcen-Klassen, die Interfaces mit den annotierten Endpunkten und entsprechende Feign-Clients enthält. Dies hat den Vorteil, dass die Definition des REST-API eines Moduls als Dependency in einem anderen Modul genutzt und mithilfe des Feign-Clients direkt angefragt werden kann.

Doch wie fügt sich das nun alles zu einer Anwendung zusammen und woher weiß der Feign-Client, an welchen Host und welche URL die HTTP-Anfragen an ein anderes Modul gehen sollen?

Spring Cloud to the Rescue!

Neben dem Feign-Client kommen auch Zuul als API-Gateway, Eureka als Service-Discovery, Hystrix als Circuit-Breaker und Ribbon als Client-Side-Loadbalancer aus dem Spring-Cloud-Netflix-Stack *[1]* zum Einsatz. Jedes fachliche und technische Modul besitzt einen Typ- („serviceID") und einen Standort-Bezeichner („location"), mit denen er sich an der Service-Discovery anmeldet. Aufgrund dieser beiden Eigenschaften wird dann immer die richtige Microservice-Instanz bei REST-Anfragen angesprochen, was unter anderem Zuul mit einer eigenen Routen-Konfiguration möglich macht. Wird also versucht, Eigenschaften von einem Patienten-Modul in Deutschland abzufragen, sieht der Request so aus: „GET https://domain.com/patients/de/api/v1/info". Dabei wertet das API-Gateway Zuul direkt den angefragten Pfad aus. Aufgrund der enthaltenen Parameter fragt es die Service-Discovery (beziehungsweise einen lokalen Cache) nach einer Liste aller laufenden Instanzen von Modulen mit den Parametern an. Zuul wählt nun mithilfe des Ribbon-Load-Balancers eine der Instanzen aus, an die der Request ohne den Routing-Präfix weitergeleitet wird. Dabei enthält jede Instanz-Information auch das jeweilige Protokoll, Host und Port. Diese Weiterleitung ist kein HTTP-Redirect, sondern wird direkt von Zuul als eine Art Proxy durchgeführt und sieht beispielsweise wie folgt aus: „GET http://prod-01-fra-de.domain.com:8001/api/v1/info". Dabei läuft auf dem Ziel-Host „prod-01-fra-de" das Patienten-Modul auf dem Port 8001. Dies hat wiederum den Vorteil, dass das API-Gateway das einzige Modul ist, das durch die Benutzer erreichbar sein muss. Danach finden alle Anfragen intern statt und eine etwaige Firewall muss nur sicherstellen, dass sich alle beteiligten Hosts unter den verwendeten HTTP(s)-Ports erreichen können.

Auch alle Web-Ressourcen des Frontends werden über diesen Mechanismus durch Zuul als Proxy und letztlich von den einzelnen Services ausgeliefert. Im Browser ist daher das gesamte System unter einer einzigen URL erreichbar, was den angenehmen Vorteil bringt, dass man sich weder im Frontend noch im Backend um CORS kümmern musste.

Segregation of Data?

Die nächste Herausforderung war, die richtige Instanz eines Service zur Abfrage von verknüpften Ressourcen zur Laufzeit zu ermitteln. Wie eingangs erwähnt, gab es regulatorische und datenschutzrechtliche Gründe, gewisse Daten in getrennten Datenbanken zu speichern. Somit entschied man sich, jedem Service die Hoheit über die Verwaltung seiner Daten zu geben, und zwar in Form einer eigenen Datenbank. Diese Anforderung steht jedoch in Konflikt mit der Art und Weise, wie der Kunde in seinem Prozess die Informationen verknüpfen muss.

Dazu ein Beispiel: Ein Befund zu einem medizinischen Fall enthält Daten, die aus so ziemlich allen Services aggregiert werden müssen. Schließlich geht es am Ende um das Wohl eines Patienten, der einer Krankenkasse angehörig ist und einen behandelnden Arzt hat, der den Befund mitteilt und gegebenenfalls eine Medikation passend zur Anamnese veranlasst. Könnten alle diese Informationen nicht mehr aufgelöst und in Relation gebracht werden, da sie sich aus rechtlichen Gründen in diversen Datenbanken befinden, wäre der Befund wertlos. Der Arzt würde den Befund nicht erhalten, der Patient keine Medikation und die Krankenkasse keine Rechnung.

Zur Laufzeit wird ein Lokalisierungs-Service angefragt, um den Standort einer Ressource im gesamten System zu ermitteln. Darauffolgende Requests werden mit dieser Information angereichert, und das gleichermaßen im Frontend wie im Backend.

Diese Trennung der Daten war eine elegante Möglichkeit, den nichttechnischen Anforderungen gerecht zu werden, jedoch stellen sich die Auswirkungen nach nun über zwei Jahren und einigen Iterationen der Entwicklung aus technischer Sicht zunehmend als problematisch dar. Das Sortieren beziehungsweise Suchen nach extern referenzierten und folglich aggregierten Daten kann nicht wie bei einem Monolithen über die Datenbank erfolgen, da sich der dargestellte Datenbestand gar nicht in nur einer Datenbank befindet. Zudem ist bei der Darstellung von Informationen zu beachten, dass der aktuelle Benutzer gegebenenfalls keine Berechtigung hat, die aggregierten Informationen oder Teile davon zu sehen. Mit zunehmender Granularität an Berechtigungen wird es jedoch immer komplizierter, alle Kombinationen zu erkennen und zu testen. Außerdem war es Wunsch der Autoren, aufgrund der allgemeinen SOA-ähnlichen Architektur zu vermeiden, dass sich ein Service um das berechtigungsabhängige Verhalten eines anderen Service kümmern muss.

Segregation of Permissions

Ähnlich wie bei der Trennung der Benutzer-Oberflächen, der Datenhoheit und des Fokus der Services entschieden sich die Autoren, auch die Berechtigungen völlig unabhängig unter den Modulen aufzuteilen. Als Identity-Management-Service mit Active-Directory-Integration und JSON-Web-Token-Provider (JWT) setzen sie in diesem Projekt auf Keycloak. Jeder Request an das Backend jedes einzelnen Moduls muss also ein signiertes Token enthalten, mit dem der agierende Benutzer identifiziert werden kann. Dies hat zur Folge, dass keine herkömmlichen Sessions im Backend verwaltet und keine Informationen des Benutzers, wie seine E-Mail-Adresse, gespeichert werden können. Die E-Mail-Adresse könnte zwar im Token enthalten sein, ist jedoch immer nur transient, da das Active Directory die Datenhoheit über diese Informationen innehat und sich die Daten zu jedem Zeitpunkt ändern können.

Soll nun beispielsweise ein Benutzer benachrichtigt werden, wenn eine bestimmte Aktion durch einen anderen Benutzer ausgeführt wird, muss immer das Identity-Management nach der aktuellen E-Mail-Adresse des Benutzers gefragt werden. Bei den Berechtigungen sieht es ähnlich aus. Zwar sind diese in der Implementierung im Token enthalten, allerdings wertet jedes Modul nur die für sich relevanten Rollen und Rechte aus.

Gibt es nun eine Funktion, die Auswirkungen auf Daten zweier Module hat, muss zunächst immer der andere Service angefragt werden, um zu prüfen, ob der aktuelle Benutzer auf diese Daten zugreifen beziehungsweise die Aktion ausführen darf. Ein Beispiel: Ein medizinischer Fall gilt als abgeschlossen, wenn der daraus resultierende Befund an den behandelnden Arzt versendet wurde. Die

Versenden-Aktion befindet sich im Service, der sich um die Befunde kümmert. Sie hat allerdings auch Auswirkungen auf den Service, der die Fälle verwaltet, und auf den Benachrichtigungs-Service, der die E-Mails oder Faxe versendet.

Die Herangehensweise an die Entwicklung dieser Art von Software war mit dem stark QM-lastigen Prozess, bedingt durch den medizinischen Kontext, an manchen Stellen sehr ungewohnt für das Team. Dies hat sich auch auf die Code-Basis ausgewirkt, schließlich musste man sich Gedanken um Fälle und Ausnahmen machen, die sonst in der Entwicklung à la „Wenn das passiert, steht die Welt eh in Flammen" abgewunken werden. Jedoch entstand nach den ersten Sprints im Team schnell das Verständnis dafür, nicht fragen zu dürfen, ob dieser Edge-Case eintritt, sondern wann und welche Auswirkungen dies am Ende für einen Patienten haben kann.

Zugegeben ist dieses geschaffene Bewusstsein der Verdienst des QM-Teams und dessen – bildlich gesprochenen – vehementen „Auf-die-Finger-Hauens". Das Durchhaltevermögen, diese Prämisse bis an die letzte Stelle im Code zu verfolgen, um eine solche Architektur stemmen zu können, ist jedoch letztendlich genau das, was dieses Projekt ausgemacht hat. All diese harten Schnitte und Trennungen in der Architektur waren aufwendig und haben auf Seiten der Implementierung und Konzeption viel Zeit in Anspruch genommen.

Allerdings zahlt sich der Trade-off inzwischen aus. Obgleich der Kunde ursprünglich gar kein zu 99,99 Prozent verfügbares System haben wollte, will er einen Ausfall durch Updates von nur wenigen Sekunden inzwischen nicht mehr in Kauf nehmen und wünscht sich für die Zukunft einen Blue-Green-Deployment-Prozess *[4]*. Mit Spring Cloud ist das kein Problem und die verteilte Architektur macht es uns noch einfacher. Da die Autoren bereits von Beginn an in der Entwicklung den Fall behandelt haben, dass ein API inkompatibel werden kann oder dass manchmal ein Service nicht schnell genug oder erst gar nicht antwortet, sind alle Module und speziell die Stellen, bei denen es Abhängigkeiten zwischen den Services gibt, auf ein solches Verhalten vorbereitet.

Fazit

Bei vielen der Herausforderungen wie dem Umgang mit den Aggregationen von Datensichten fehlt auch jetzt noch die perfekte Lösung, um sie zu meistern. Überwiegend waren es Kompromisse, die umgesetzt wurden. Diese musste der Kunde tragen und verstehen. Letztlich war das Ziel aller, den Aufwand an Dokumentation, manuellen Tests und externen Validierungen durch CE-Kennzeichnung, TÜV, U.S. Food and Drug Administration (FDA) etc. für die initiale und weiterführende Entwicklung so gering wie möglich zu halten. Das dazu nötige beidseitige Verständnis von Anforderungen, Risiken und Problemstellungen sowie die daraus folgenden Lösungen und Kompromissen wären ohne eine sehr enge Zusammenarbeit mit dem Kunden nicht möglich gewesen. Daher gab es auch auf Kundenseite einige Personen, die aktiv an der Entwicklung und Gestaltung der Software beteiligt waren. So fanden nach jedem Sprint Tests und Feedback-Runden mit Key-Usern statt. Mehrere Stakeholder, wie zum Beispiel der Product-Owner und diverse Abteilungsleiter, gaben bei den Sprint-Planungen essenzielles Fachwissen mit und steuerten die Entwicklung durch Priorisierung der Aufgaben.

Am Ende ist es gelungen, eine Software zu erstellen, die auch – aber eben nicht nur – den QM-relevanten Aspekten gerecht wird und zugleich eine extrem hohe Benutzer-Akzeptanz beim Kunden findet. Dabei ist auch am Ende eine Microservice-orientierte Architektur gewachsen, die geholfen hat, einige der Prozess-inhärenten Probleme in dem Projekt zu lösen, um dafür andere, neue Herausforderungen für die weitere Entwicklung zu schaffen.

Literatur

[1] *https://cloud.spring.io/spring-cloud-netflix*
[2] GAMP 5, ein risikobasierter Ansatz für konforme GxP-computergestützte Systeme, ISPE, 2008
[3] *http://projects.spring.io/spring-hateoas/ & https://tools.ietf.org/id/draft-kelly-json-hal-01.html*
[4] *https://martinfowler.com/bliki/BlueGreenDeployment.html*

Daniel Clasen
daniel.clasen@viaboxx.de

Daniel Clasen ist Software-Entwickler und IT-Nerd mit Herz und Seele. Als Full-Stack-Entwickler – spezialisiert auf moderne UI-Entwicklung und Java-basierte Web-Anwendungen – liebt er jede Art von Herausforderungen, seien sie Software-, Hardware- oder Netzwerk-bezogen. Neben seinem Job als Entwickler und Team-Lead bei der Viaboxx GmbH beteiligt er sich an Open-Source-Projekten, schreibt Blog- sowie Fachartikel und hält Vorträge auf Konferenzen.

Jan Nonnen
jan.nonnen@viaboxx.de

Jan Nonnen ist als Entwickler und Team-Lead schon einige Jahre bei der Viaboxx GmbH aktiv. Daneben ist er bekennender „Software Craftsman" sowie seit dem Jahr 2011 Mitbegründer und Organisator des Bonn Agile Meetups. Er spricht auf Konferenzen und veröffentlicht Fachartikel. Ihn begeistern insbesondere testgetriebene Entwicklung, Clean Code und Software-Design.