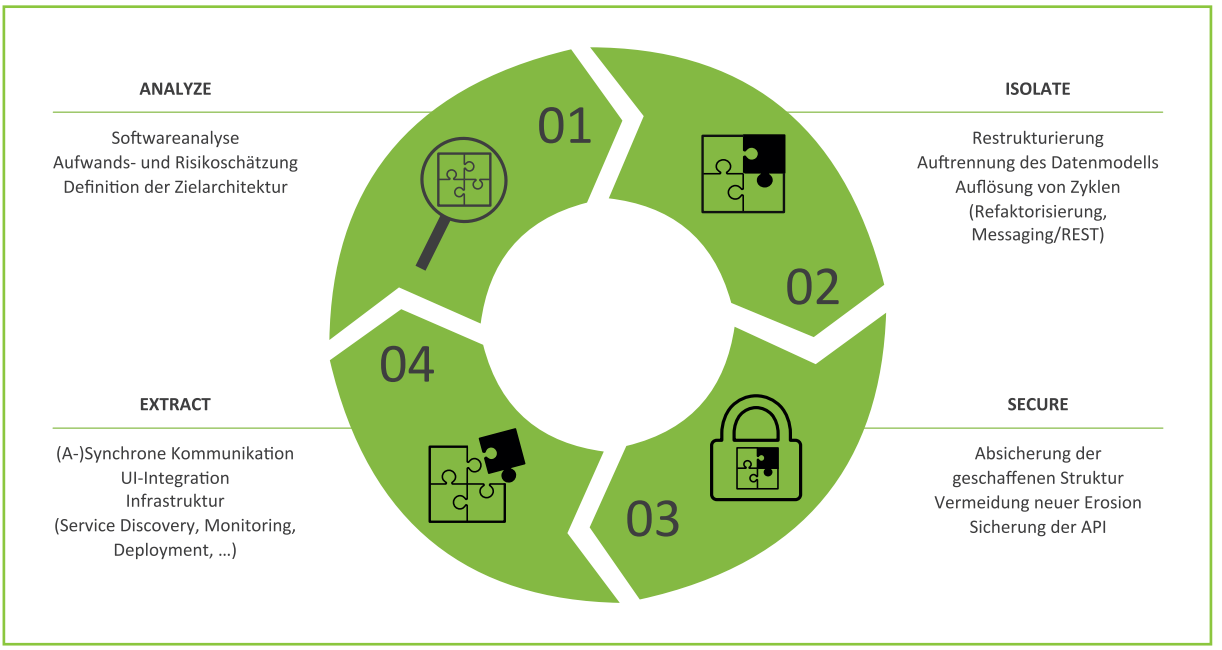
Wir (z)erlegen einen Monolithen …

oder wie wir Anwendungen fit für die Zukunft machen

Die Entscheidung ist getroffen. Unser über Jahre gewachsenes, monolithisches System soll nun restrukturiert und damit fit für die Zukunft gemacht werden. Während diese Entscheidung fachlich und geschäftlich motiviert sein sollte, stellt sich ziemlich schnell die Frage des "Wie?". Dieser Artikel soll einen Überblick über das prinzipielle Vorgehen, die Planung der Refaktorisierung, dessen Umsetzung und die wichtigsten Fallstricke und Entscheidungen geben, um am Ende nicht vor einem gescheiterten Projekt zu stehen.

Wer kennt es nicht, das monolithische Legacy-System? Seit vielen Jahren entwickelt, implementiert es eine große Menge von Geschäftslogik und ist so ein sehr wichtiger Bestandteil eines Unternehmens. Daher ist es umso wichtiger, rasch Neuerungen ausrollen und die spezifischen Anwendungsteile individuell skalieren zu können. Doch schnell kann eine unstrukturierte, schwerfällige und monolithisch deployte Anwendung diesen Prozess verlangsamen bzw. gar unmöglich machen.

Abbildung 1:
Das AISE-Prozessmodell



Aus dieser Situation gibt es verschiedene, mal mehr, mal weniger sinnvolle Auswege. Einer wäre die komplette Neuimplementierung der Anwendung. Doch kommt dieses Vorgehen mit hohem Aufwand und Risiko einher. Außerdem ist selbst hierbei nicht garantiert, dass die neue Anwendung besser strukturiert sein wird und unsere Probleme lösen kann. Stattdessen wollen wir das bestehende System schrittweise und gezielt strukturieren und somit die bereits investierte Arbeit honorieren.

Analyze – Isolate – Secure – Extract – Repeat

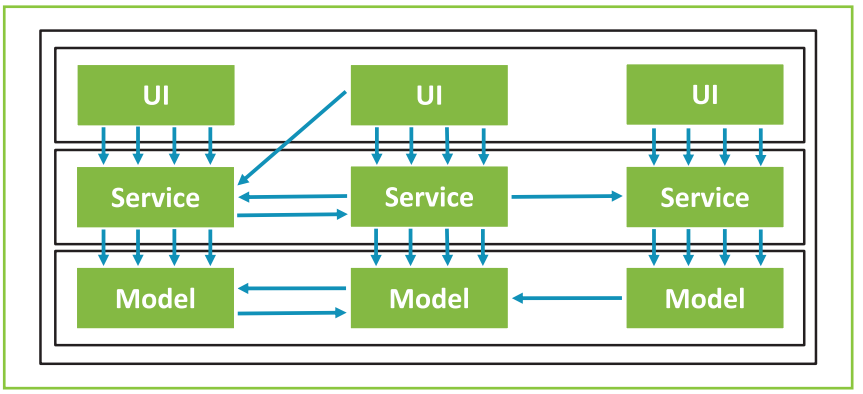
Die Restrukturierung eines monolithischen Legacy-Systems und gegebenenfalls der Extraktion einzelner Microservices kann, wie bereits angedeutet, auf unterschiedliche Art und Weise erfolgen. Da wäre Variante 1, den Monolithen in einem großen Schritt komplett zu restrukturieren bzw. zu zerlegen und anschließend das Ergebnis (hoffentlich) erfolgreich auszurollen. Diese Variante gleicht einer Neuimplementierung. Die Nachteile liegen auf der Hand, denn der langwierige Prozess schränkt die Lieferfähigkeit ein und bringt ein hohes Risiko des Scheiterns mit sich.

Stattdessen nutzen wir hier einen intern entwickelten und erprobten iterativen Prozess, der sich an dem Chicken-Little-Prinzip orientiert. Dabei erfolgt die Migration des Altsystems in einzelnen, wiederum unterteilten Iterationen. Die Iterationen konzentrieren sich auf jeweils eine Fachlichkeit im System und haben zum Ziel, diese Fachlichkeit zunächst in ein eigenständiges Modul zu isolieren und, falls dies im konkreten Fall sinnvoll ist, in einen Micoservice zu extrahieren. Die vier Schritte Analyze, Isolate, Secure und Extract (AISE) sind als Prozess in Abbildung 1 dargestellt.

Die jeweiligen Schritte bauen aufeinander auf, sind aber in sich abgeschlossen und haben jeweils eine deployfähige Applikation als Ergebnis. Das hat zum Vorteil, dass individuell je Fachlichkeit entschieden werden kann, was aus der geschäftlichen Perspektive sinnvoll ist und nur dies umgesetzt werden muss. Außerdem kann somit schnell auf neue Erkenntnisse reagiert und diese in den Prozess eingebunden werden.

Im Folgenden schauen wir uns die vier Schritte genauer an.

Abbildung 2:
Fachliche Abhängigkeiten in einer technisch strukturierten Anwendung



Hinweis: Als Beispiel wird die Identifizeriung und Herauslösung einer Subdomäne aus dem Open-Source e-Commerce-System "shopizer" (Webseite: www.shopizer.com) beschrieben, welches in Java implementiert ist und darüber hinaus auf dem Spring-Framework, Java Server Pages sowie JPA basiert.

Analyze – Eine Bestandsaufnahme

Die Analysephase stellt die Grundlage für die eigentliche Refaktorisierung dar. In der Analyse ist es unsere Aufgabe, die im Betrieb auftretenden Symptome (siehe Artikel "Agilität heißt permanente Modernisierung" Seite 6 ff.) auf technische Ursachen zu untersuchen. Dabei ist eine mögliche Lösung, dass es sinnvoll ist, eine oder mehrere Subdomänen als Microservices bereitzustellen. Diese Bewertung sollte unter Einbeziehung aller Stakeholder und unter besonderer Berücksichtigung der geschäftlichen Aspekte erfolgen.

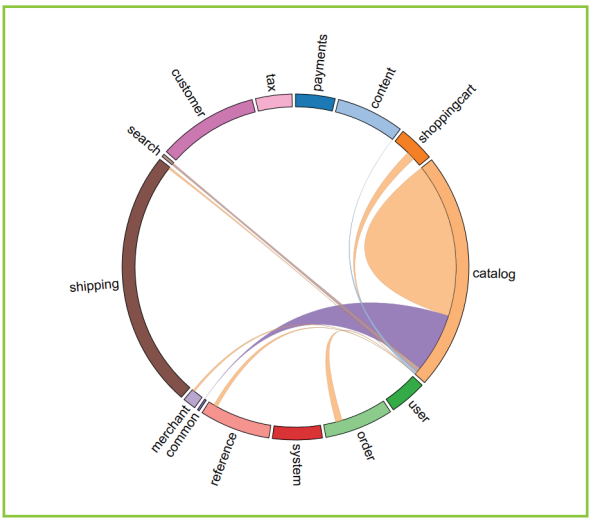
Hat man sich dafür entschieden, Komponenten zu extrahieren, gilt es, in Absprache mit den Domänenexperten und anderen Entwicklern zu identifizieren, wie sich die gedachte, ideelle Struktur einer fachlichen Zerlegung in realer Codestruktur manifestiert. Das Wissen über die betroffenen Anwendungsteile und die Größe der Subdomäne ist außerdem notwendig, um später eine sinnvolle Aufwandsschätzung erstellen zu können.

Der Großteil der Anwendung ist primär nach technischen Gesichtspunkten strukturiert, beispielsweise wie bei einer 3-Schicht-Architektur nach technischen Schichten. Daraus folgt unweigerlich, dass sich die viel stärker zusammenhängenden fachlichen Aspekte über die primäre Struktur verteilen. Die Auftrennung in Microservices erfolgt jedoch anhand fachlicher Subdomänen, was der Abhängigkeitsstruktur zwischen den einzelnen Klassen nahekommt (Abbildung 2).

Um einen guten Überblick über die Subdomänen zu bekommen, ist neben der Analyse der vorliegenden Dokumentation besonders das Gespräch mit anderen Entwicklern und die Analyse der Softwarestrukturen mit jQAssistant (Webseite: jqassistant.org) hervorzuheben. Bei jQAssistant handelt es sich um ein Werkzeug, welches den Java-Byte Code als Neo4j-Graphdatenbank aufbereitet und so beispielsweise die Call-Hierarchien abfragbar macht. Die Analyse macht sich dies zu Nutze, um ausgehend von der implementierten Geschäftslogik der betrachteten Subdomäne (typischerweise ist diese im Domain-Layer anzutreffen) zu bestimmen, welche Klassen im Data- und Presentation-Layer Bestandteil der Subdomäne sind und so auch refaktorisiert werden müssen.

Mit geschickten Abfragen und einer Visualisierung der Abhängigkeiten zwischen den einzelnen Subdomänen lässt sich so mit jQAssistant der Impact des Herausziehens einer Subdomäne bestimmen und ein Überblick über die allgemeinen Abhängigkeitsstrukturen aufbauen. Das vereinfachte Chord-Diagramm in Abbildung 3 zeigt alle ein- und ausgehenden Abhängigkeiten der

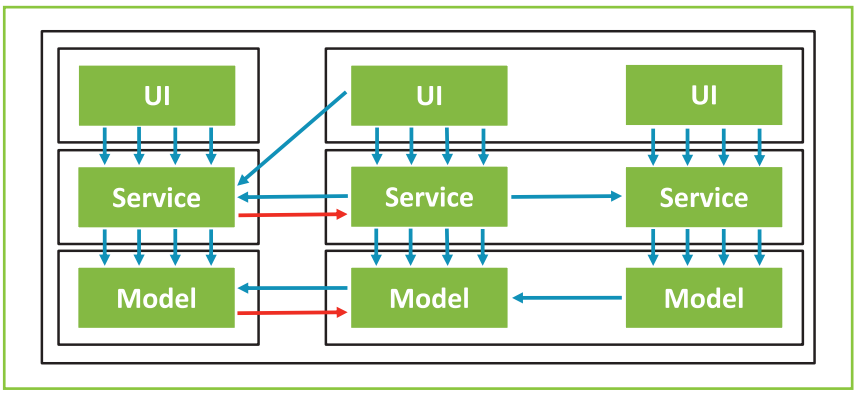
Abbildung 3:
Abhängigkeitsstrukturen der Katalog- und Such-Subdomäne als Chord-Diagramm



zwei Subdomänen "Katalog" und "Suche". Man erkennt, dass die Katalog- und Such-Komponente, wenn auch durch die Paketierung in Java getrennt, sehr stark zusammenhängen (die Suche gehört exklusiv zum Katalog) und auch fachlich gesehen zusammengehören.

Schlussendlich hat die Analyse zur Aufgabe, die kurz- (Isolation) und mittelfristige (Extraktion) Architektur der Anwendung zu definieren, um so in kontrollierten Schritten und mit einem konkreten Ziel die langfristige (komplette Zerlegung nach mehreren Iterationen) Architektur zu erreichen.

Abbildung 4:
Zyklische Abhängigkeiten zwischen Subdomänen



Im Folgenden wird die Herauslösung der Subdomäne "Katalog" beschrieben. Diese soll als eigenständiger Microservice parallel zu den im monolithischen Kern verbleibenden Subdomänen (Order, Payment, Shipping, etc.) betrieben werden können.

Isolate – Der Weg zum strukturierten Monolithen

Das Ziel der Isolation ist, die ausgewählte Subdomäne in ein eigenes Maven-Modul herauszulösen. Das System stellt weiterhin einen Deployment-Monolithen dar, jedoch ist die Hürde und das Risiko, diese Struktur zu erreichen, nicht so hoch wie bei der direkten Extraktion in einen eigenständigen Microservice. Statt sich bereits um Themen wie Transaktionskontexte und zusätzliche Technologien (Service Discovery, Messaging, etc.) Gedanken machen zu müssen, liegt hier der Fokus auf den implementierten fachlichen Prozessen.

Wie wir in der Analyse sehen konnten, existieren zwischen dem Katalog und dem Kern zyklische Abhängigkeiten. Ein direktes Herausziehen des Katalogs in ein eigenständiges Maven-Modul würde damit zum Brechen des Builds führen. Die oberste Priorität von AISE ist jedoch, die Buildfähigkeit so wenig wie möglich zu riskieren und somit liefer- und weiterentwicklungsfähig zu bleiben. Daher werden identifizierte Klassen des Katalogs zunächst in eine parallele Paketstruktur überführt. Verbotene Abhängigkeiten (also vom Katalog zum Kern) können dann weiterhin mit jQAssistant gefunden und durch Refaktorisierung des Codes aufgelöst werden (Abbildung 4).

Zur Auflösung der kritischen Abhängigkeiten stehen verschiedene Möglichkeiten zur Verfügung, die je nach Fall mal mehr, mal weniger passend sind:

1. Code-Duplikation… ist zwar nicht im Sinne des Don't Repeat Yourself (DRY)-Prinzips, kann aber ein probates Mittel sein und trifft am ehesten den Gedanken von Microservices.

2. Shared Libraries… ist ebenfalls nicht im Sinne der Microservice-Architektur, da dies zu einer indirekten Kopplung der Services führt, ist aber bei komplexem, nicht fachlichem und stabilem Code eine gute Möglichkeit zur Aufwandsminimierung bei Bugs.

3. Eventing... dient der Umkehr von Kommunikationsbeziehungen. Dabei werden Konsumenten bei einer Zustandsänderung nicht mehr direkt vom Produzenten aufgerufen, sondern durch Events benachrichtigt.

4. Remote-Kommunikation… entkoppelt die Services auf Java-Ebene durch REST, wenn fachliche Aspekte abgerufen werden sollen und diese Abhängigkeiten auf keinem anderen Weg aufgelöst werden können. Zu beachten ist hierbei, dass dies besonders dann auftreten kann, wenn der verbliebene Kern noch eine große Menge anderer, verschiedener Fachlichkeiten implementiert. Diese Abhängigkeitsbeziehungen sollten in folgenden Iterationen betrachtet werden und gegebenenfalls zu einer weiteren Zerlegung führen.

An diesem Punkt können durchaus Lösungen implementiert werden, welche die Komplexität der Implementierung erhöhen, die aber bei dem Schritt der Extraktion nur noch minimale Anpassungen erfordern. Bei der Auswahl der Lösungen ist demnach Weitblick wichtig.

Schnell wird man auch an den Punkt kommen, dass ungewollte Abhängigkeiten zwischen den Datenmodellen bestehen. Um diese aufzulösen, kommt man, solange die Information tatsächlich benötigt wird, langfristig nicht herum, über das Thema der Datenreplikation nachzudenken.

Dabei werden benötigte Daten des Kerns in den Katalog durch einen Eventing-Mechanismus repliziert. Wichtig ist dabei zu beachten, dass es jeweils nur ein führendes System geben darf.

Wird eine Entität im Kern kreiert, aktualisiert oder gelöscht, publiziert dieser ein passendes Event mit grundlegenden Informationen der Entität (zumindest der ID zur Identifikation) als Data Transfer Object (DTO), worauf interessierte Services lauschen können. An dieser Stelle ist es noch nicht notwendig, einen Event-Broker wie Apache Kafka einzusetzen. Stattdessen bieten Spring mit Application Events und JavaEE mit CDI-Events ausreichend Möglichkeiten zur Implementierung dieses Szenarios.

Da die DTOs universell für alle (zukünftigen) Services dienen sollen, ist es ausreichend, nur grundlegende Information zu publizieren. Zusätzlich von einem Service benötigte Informationen können dann über eine der zwei folgenden Varianten nachgeholt werden.

1. On-Demand… holt die benötigte Information per REST, sobald sie benötigt wird und persistiert sie gegebenenfalls.

2. On-Event… holt alle benötigten Informationen per REST bei Eintreffen eines Create- bzw. Update-Events.

Die Wahl der Strategie ist abhängig von den Aspekten der Verfügbarkeit, der Aktualität der Daten und den möglichen Latenzen und sollte vor dem Hintergrund der späteren Extraktion betrachtet werden.

Der lauschende Service definiert je Entität eine Duplikat-Entität und verwaltet diese selbst in eigenen Tabellen. Sinnvoll ist es nun, Transaction-

Komplexität ist kein Hindernis: Effektiver Einsatz von Werkzeugen und Methoden bringt Sie sicher ans Ziel.

Boundaries einzuführen, um bereits jetzt mögliche Konsistenzprobleme feststellen zu können. Dabei läuft das Empfangen des Events nicht in der Transaktion, von der das Event ausgeht.

Sind diese Schritte erledigt, kann ein neues Maven-Modul angelegt und die betroffenen Klassen in dieses verschoben werden. Da wir keine zyklischen Abhängigkeiten mehr haben, kann die Anwendung weiterhin gebaut werden.

Darüber hinaus stellen die JSP-Views ein Problem dar, da diese aufgrund der benötigten Artefaktstruktur zum Deployment nicht in einem eingepackten JAR liegen dürfen, sondern im WAR vorhanden sein müssen. Hier ist es mit Hinblick auf die bevorstehende Extraktion legitim, diese im Kern zu belassen und nur durch die Ordnerstruktur abzutrennen.

An dieser Stelle wurden die Abhängigkeiten vom Katalog zum Kern durch geschickte Refaktorisierung aufgelöst. Jedoch ist auch die entgegengesetzte Richtung von Interesse. Im Sinne des Domain-Driven Designs sollte hier, besonders im Hinblick auf die bevorstehende Extraktion, auch entkoppelt werden. Das bedeutet, dass zum einen Methodenaufrufe vom Kern zum Katalog immer durch definierte (Java)-Schnittstellen erfolgen sollten. Die stabile API ermöglicht bereits im Deployment-Monolithen, Änderungen und Erweiterungen im Katalog sicher durchführen zu können. Zum anderen sollte auch die Datenreplikation vom Katalog zum Kern erfolgen, um den Katalog so autark wie möglich zu gestalten. Dadurch unterliegen alle Katalog internen Services und Entitäten dem Entwicklungsprozess des Katalogs. Die definierte API schafft klare Kontrakte sowohl im Sinne der Schnittstellen als auch der Austauschobjekte.

Secure – Altes Haus, neues Dach

Die vorangegangene Isolation der Subdomäne in ein eigenes Maven-Modul bringt viele Vorzüge zur Vermeidung ungewollter Abhängigkeiten mit sich. Jedoch sollte nicht die Gefahr der erneuten Erosion unterschätzt werden.

Es ist an dieser Stelle wichtig, das aufgebaute Wissen über die Anwendung zu nutzen und Abhängigkeitsbeziehungen sowohl innerhalb eines Moduls, als auch zwischen Modulen zu dokumentieren und abzusichern. Dafür bietet sich wiederum jQAssistant an, welches ausführbare Architekturdokumentation ermöglicht. So kann man mittels Constraints (Cypher-Abfragen) definieren, welche Abhängigkeitsstrukturen erlaubt beziehungsweise verboten sind und bei Verletzungen dieser den Build fehlschlagen lassen.

Es können verschiedene Aspekte der Anwendungsstruktur abgesichert werden. Dazu zählen zum Beispiel die Folgenden:

- Absicherung der erlaubten Abhängigkeiten zwischen Modulen
- Absicherung der Art der Abhängigkeiten, z. B. nur API-Zugriffe
- Absicherung der Architektur innerhalb eines Moduls

Extract – Aus Eins mach Zwei

Die auserwählte Subdomäne befindet sich in einem eigenen Maven-Modul und die Kommunikation zwischen den Subdomänen erfolgt in beide Richtungen über klare Schnittstellen. Für Projekte, in denen die vollständige Extraktion nicht gewinnbringend ist, ist nun ein guter Zeitpunkt, um zu betrachten, welche weiteren Subdomänen noch isoliert werden können, um und damit langfristig die Qualität der Architektur hoch zu halten. Nach dem Durchlaufen mehrerer Iterationen wäre das Resultat ein wohlstrukturierter und damit wartbarer Monolith.

Wir wollen allerdings noch einen Schritt weiter weg vom Deployment-Monolithen gehen. Dafür müssen zunächst die konkreten Abhängigkeiten vom Kern zum Katalog aufgelöst werden. Die zuvor definierten Java-Interfaces des Katalogs weichen REST-Schnittstellen. Das soll in diesem Rahmen nicht weiter ausgeführt werden.

Neben der Umstellung der synchronen Kommunikation muss ebenfalls die asynchrone Kommunikation umgestellt werden. Anstatt von Spring Application Events kann die Nutzung eines Event Brokers wie Apache Kafka eine Möglichkeit darstellen. Die bisher genutzten DTOs können weiterhin verwendet werden, könnten aber auch durch Service interne Implementierungen ersetzt werden, um die Kopplung weiter zu minimieren.

Folgende Punkte sind dringend zu beachten:

1. Eventual Consistency: Durch asynchrones Eventing können zeitweise inkonsistente Zustände auftreten, z. B. sind benötigte Daten noch nicht vorhanden. Die Lösung von deratigen Problemen ist abhängig von fachlichen Anforderungen und sollte für jeden Fall individuell bewertet werden. Folgende Lösungsmöglichkeiten sind denkbar:

- Abfrage fehlender Daten per REST
- Nutzung von vorhandenen Fallback-Daten (bspw. ältere Daten)
- Anzeigen einer Fehlermeldung

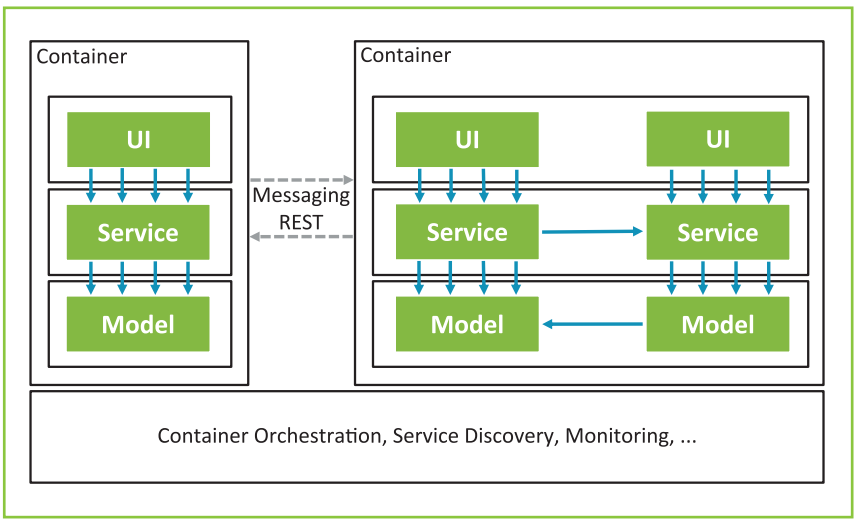
2. Infrastrukturprobleme: Durch die Auftrennung der Anwendung in eigenständige Services können Netzwerkprobleme oder die mangelnde Verfügbarkeit eines benötigten Services zu fehlenden Information führen. Je nach Anwendungsfall und Anforderungen an die Datenaktualität kann eine der folgenden Varianten eine Lösung darstellen:

- Wiederholung der REST-Anfrage
- Nutzung von vorhandenen Fallback-Daten (bspw. ältere Daten)
- Fehlertolerante UI
- Anzeigen einer Fehlermeldung

Die aufgeführten Fälle und Lösungen sind nicht abschließend, sondern sollen ein Gefühl dafür vermitteln, welche Probleme durch die Zerlegung auftreten könnten.

Bestehen keine konkreten Abhängigkeiten mehr, ist das Thema der Infrastruktur prägend. Viele Fragen, die im (Deployment-) Monolithen nie von Relevanz waren, stellen sich jetzt. Wichtig ist dabei wieder, planvoll vorzugehen und die Buildfähigkeit so wenig wie möglich zu gefährden. Das bedeutet, alle Schritte, welche noch innerhalb des Deployment-Monolithen erledigt werden können, müssen umgesetzt werden bevor zwei Deployments entstehen.

Abbildung 5:
Die Microservice-Landschaft



Service Discovery ist ein weiteres wichtiges Thema, welches zuvor noch nicht notwendig war. Schließlich müssen sich die einzelnen Services auch gegenseitig referenzieren können. Service Registries wie Eureka können bei einer großen Zahl an Services viel Aufwand abnehmen und sind quasi Pflicht, sobald die Orchestrierung automatisch lastabhängig erfolgt. Bei einer geringen Anzahl an Services kann die Service Discovery manuell durch Konfiguration gelöst werden. Die Konfiguration kann zur Vereinfachung bereits im Deployment-Monolithen vorbereitet werden.

Bei der Konfiguration der Anwendung ist im Allgemeinen darauf zu achten, dass mit dem Einführen getrennter Deployments jeder Service eigene Konfigurationsdateien benötigt. Diese sind im besten Fall disjunkt, können also auch leicht getrennt werden. Bei Überschneidungen in der Konfiguration der Services ist entweder die Duplikation möglich oder man nutzt einen Konfigurationsserver, welcher die Konfigurationen leicht verwaltbar macht und die benötigten Informationen den Services zur Verfügung stellt.

Da in der Konfiguration typischerweise auch Informationen über die verwendeten Datenbankschemata hinterlegt sind, stellt sich außerdem die Frage, wie und wo unabhängige Services ihre Daten persistieren. Prinzipiell gibt es dafür drei Varianten:

1. Eigene Tabellen je Service… beschreibt die Verwendung des gleichen Schemas für alle Services. Eine Trennung erfolgt nur durch die Nutzung unterschiedlicher Tabellen je Service. Dadurch ist es leicht möglich, dass ein Service Zugriff auf die Daten eines anderen Services erlangt. Ebenfalls können sich Services gegenseitig durch die Verwendung der gleichen Datenbank in ihrer Performance beeinflussen. Diese Variante sollte daher nur eine Notlösung darstellen.

2. Eigenes Schema je Service… ähnelt dem vorhergehenden Prinzip, grenzt sich jedoch insofern ab, dass nun für jeden Service ein eigenes Schema in der gleichen Datenbank zur Verfügung steht. Durch die exklusive Nutzung der Schemata ermöglicht diese Variante eine korrekte Trennung der Daten. Bei verteilten Systemen sind allerdings weiterhin Faktoren wie Latenz und Durchsatz zu bedenken. Diese können zur gegenseitigen Beeinflussung der Services führen.

3. Eigene Datenbank je Service… stellt Microservices in Reinform dar, da hier Themen wie Ausfallsicherheit, Latenz und Durchsatz immer Service-individuell behandelt werden können. Darüber hinaus wird perspektivisch auch die Wahl einer anderen Datenbanktechnologie ermöglicht (z. B. Documentstore oder Graphendatenbank).

Da die User Experience nicht unter der neuen Architektur leiden darf, werden Lösungen zur Integration der Oberflächen verschiedener Services benötigt. Im Folgenden sind drei Möglichkeiten kurz dargestellt.

1. Monolithische UI… ist eine weniger aufwendige Variante. Die UI spricht dabei durch ein API-Gateway die REST-Endpunkte der einzelnen Services an.

2. Client-Side UI Composition… implementiert Service-spezifische UI-Komponenten, welche durch Page Skeletons eingebunden werden.

3. Server-Side Page Fragment Composition… implementiert Service-spezifische UI-Komponenten, welche durch Page Templates server-seitig aggregiert werden.

In unserem Fall bietet sich die dritte Variante an, da langfristig weitere Fachlichkeiten aus dem Kern extrahiert werden sollen und dieser somit nur noch der View-Aggregation dient. Um diese Aufteilung sinnvoll zu bewerkstelligen, ist es an der Zeit, den Deployment-Monolithen zu spalten. Das bedeutet, dass die extrahierte Komponente als ein eigenständiges Maven-Modul gebaut und separat deployed werden kann. Ist dann die Service-Discovery korrekt (gegebenenfalls manuell) konfiguriert und die Infrastruktur aufgebaut, können die Anwendungen unabhängig voneinander starten und liefern für den Nutzer dennoch das gewohnte Bild.

Für den Architekten ergibt sich dagegen ein neues, ähnlich wie in Abbildung 5 gezeigtes Bild.

Alles kann – Nichts muss

Herzlichen Glückwunsch! Aus einer monolithischen, technisch strukturierten Anwendung konnte erfolgreich eine Fachlichkeit in einen eigenständigen Microservice herausgelöst und somit der Weg für die Zukunft geebnet werden. Themen wie die automatische Orchestrierung der Services zum Load Balancing, beispielsweise mit Docker und Kubernetes, Monitoring und Logging wurden noch nicht betrachtet.

Die bisherige Refaktorisierung hat eine gute Grundlage geschaffen, um die Anwendung Schritt für Schritt aufzuteilen und sie bei Bedarf in der Cloud zu betreiben.

Wichtig ist dabei immer abzuschätzen, ob das Einführen einer neuen Technologie einen Mehrwert für die Anwendung darstellt oder es sich nur um eine technische Spielerei handelt. Das ist individuell zu entscheiden und gilt nicht nur für konkrete Technologien, sondern auch bei der Frage, ob beispielsweise die Extraktion in einen eigenständigen Microservice überhaupt notwendig ist.

Hier zeigt sich die Stärke des AISE-Prozesses: Er gibt Architekten und Entwicklern eine Methodik an die Hand, um zielgerichtet und in kleinen, funktionserhaltenden Schritten, die bestehende Anwendung fit für die Zukunft zu machen.

Stephan Pirnbaum
Berater
buschmais GbR

stephan.pirnbaum@buschmais.com
twitter.com/spirnbaum

Stephan Pirnbaum ist Berater bei buschmais. Seine fachlichen Schwerpunkte liegen in der Konzeption und Entwicklung von Java-Applikationen im Unternehmensumfeld sowie der Analyse von Legacy-Anwendungen mit jQAssistant.