KAPITEL 3

Den Monolithen aufteilen

In Kapitel 2 haben wir uns darum gekümmert, wie wir die Migration auf eine Microservices-Architektur durchdenken können. Genauer gesagt, haben wir untersucht, ob es überhaupt eine gute Idee ist, und wenn ja, wie Sie Ihre neue Architektur ausrollen und sicherstellen, dass Sie in die richtige Richtung laufen.

Wir haben darüber gesprochen, wie ein guter Service aussieht und warum kleinere Services für uns besser sein können. Aber wie gehen wir damit um, wenn wir schon eine große Zahl von Anwendungen haben, die diesen Mustern nicht folgen? Wie teilen wir diese monolithischen Anwendungen auf, ohne alles neu zu schreiben und mit einem großen Knall umzusetzen?

Im Rest dieses Kapitels werden wir eine Reihe von Migrations-Patterns vorstellen und Tipps liefern, die Ihnen dabei helfen können, eine Microservices-Architektur umzusetzen. Wir schauen uns Patterns an, die für fertige Software von dritter Seite, für Altsysteme oder weiter einzusetzende (und zu erweiternde) Monolithen nützlich sind. Damit inkrementelle Roll-outs funktionieren, müssen wir sicherstellen, dass wir weiter mit der bestehenden monolithischen Software arbeiten können.

Denken Sie daran, dass wir bei unserer Migration inkrementell vorgehen wollen. Die Microservices-Architektur soll in kleinen Schritten aufgebaut werden, sodass wir aus dem Prozess lernen und bei Bedarf die Richtung ändern können.

Ändern wir den Monolithen, oder lassen wir es bleiben?

Als Teil Ihrer Migration müssen Sie sich schon ziemlich zu Beginn überlegen, ob Sie planen (und dazu in der Lage sind), den bestehenden Monolithen zu ändern, oder ob Sie das nicht wollen.

Gibt es für Sie die Möglichkeit, das bestehende System zu verändern, haben Sie in Bezug auf die verschiedenen Patterns die größtmögliche Auswahl. In manchen Situationen wird es aber harte Beschränkungen geben, die Ihnen diese Möglichkeit

verwehren. Bei dem bestehenden System kann es sich um ein Fremdprodukt handeln, für das Sie keinen Quellcode besitzen, oder es ist in einer Technologie geschrieben, für die Sie bei sich kein Know-how mehr haben.

Es kann auch sanftere Treiber geben, die Sie davon abhalten, das bestehende System anzupassen. Es ist möglich, dass sich der aktuelle Monolith in einem solch schlechten Zustand befindet, dass die Veränderungskosten einfach zu hoch wären – dann sollten Sie Ihre Verluste niedrig halten und neu beginnen (allerdings fürchte ich, wie schon geschrieben, dass manche viel zu schnell zu diesem Schluss kommen). Ein anderer Grund kann darin bestehen, dass viele andere Personen weiterhin am Monolithen arbeiten und Sie ihnen nicht im Weg sein wollen. Manche Patterns, wie zum Beispiel das weiter unten vorgestellte *Branch by Abstraction*, können solche Probleme mindern, aber dennoch können Ihnen die Auswirkungen auf andere noch zu groß sein.

Ich erinnere mich an eine Situation, in der ich mit Kollegen zusammen dabei half, ein System mit hoher Rechenlast zu skalieren. Die zugrunde liegenden Berechnungen wurden durch eine C-Bibliothek ausgeführt. Unsere Aufgabe war es, die verschiedenen Eingaben zusammenzubringen, sie an die Bibliothek zu übergeben und dann die Ergebnisse abzuspeichern. Die Bibliothek selbst bestand quasi nur aus Problemen. Speicherlecks und ein furchtbar ineffizientes API-Design waren nur zwei der Hauptursachen für Probleme. Monatelang fragten wir nach dem Quellcode der Bibliothek, um diese Probleme zu beheben, aber wir wurden abgewiesen.

Viele Jahre später traf ich mich mit dem Projektsponsor und fragte ihn, warum wir die zugrunde liegende Bibliothek nicht ändern durften. Da erst gestand mir der Sponsor, dass sie den Quellcode verloren hatten, es ihnen aber zu peinlich war, es uns zu erzählen! Passen Sie also auf, dass Ihnen das nicht passiert.

Hoffentlich sind Sie also aktuell in einer Position, in der Sie mit der Codebasis des bestehenden Monolithen arbeiten und sie anpassen können. Aber was ist, wenn wir das nicht können – geht es dann nicht weiter? Ganz im Gegenteil – hier kann uns eine Reihe von Patterns helfen, auf die wir bald zurückkommen werden.

Ausschneiden, einfügen oder reimplementieren?

Auch wenn Sie Zugriff auf den Code des bestehenden Monolithen haben, ist zu Beginn der Migration zu Ihren neuen Microservices nicht immer klar, was mit dem vorhandenen Code passieren soll. Sollten wir ihn so, wie er ist, kopieren oder die Funktionalität neu implementieren?

Ist die bestehende Codebasis des Monolithen ausreichend gut faktorisiert, können Sie eventuell viel Zeit sparen, indem Sie den Code selbst kopieren. Entscheidend ist hier, zu verstehen, dass der Code aus dem Monolithen *herauskopiert* werden muss, die Funktionalität aus dem Monolithen selbst aber nicht entfernt werden darf (zumindest jetzt noch nicht). Warum? Weil wir mehr Optionen haben, wenn wir die Funktionalität noch eine gewisse Zeit im Monolithen belassen. Wir können so

notfalls zurückrollen oder beide Implementierungen parallel laufen lassen. War die Migration irgendwann erfolgreich und sind Sie glücklich damit, können Sie die Funktionalität dann entfernen.

Den Monolithen refaktorieren

Ich habe festgestellt, dass die größte Hürde für den Einsatz existierenden Codes aus dem Monolithen für Ihren neuen Microservice darin besteht, dass die vorhandene Codebasis traditionell nicht rund um Businessdomänenkonzepte organisiert ist. Viel häufiger sind technische Kategorisierungen (denken Sie an all die Model/View/Controller-Paketnamen, die Sie beispielsweise schon gesehen haben). Versuchen Sie, Businessdomänenfunktionalität zu kopieren, kann das schwierig sein – die bestehende Codebasis passt nicht zu dieser Kategorisierung, daher kann es schon ein Problem sein, den passenden Code überhaupt zu finden!

Wollen Sie Ihren bestehenden Monolithen entlang von Businessdomänengrenzen neu organisieren, empfehle ich Ihnen *Effektives Arbeiten mit Legacy Code* von Michael Feathers (mitp, 2010). In seinem Buch definiert Michael das Konzept eines *Seam* – also einer Stelle, an der Sie das Verhalten eines Programms ändern können, ohne das bestehende Verhalten anpassen zu müssen. Im Grunde definieren Sie einen Seam rund um den zu ändernden Code, arbeiten an einer neuen Implementierung des Seam und tauschen ihn dann aus, nachdem die Änderung durchgeführt wurde. Er stellt Techniken vor, um sicher mit Seams zu arbeiten und dadurch zu helfen, die Codebasis aufzuräumen.

Wenngleich Michaels Konzept der Seams ganz allgemein auf viele Bereiche angewendet werden kann, passt es besonders gut bei Kontextgrenzen, die wir in Kapitel 1 behandelt haben. *Effektives Arbeiten mit Legacy Code* mag sich nicht direkt auf Domain-Driven-Design-Konzepte beziehen, aber Sie können die Techniken aus dem Buch nutzen, um Ihren Code anhand dieser Prinzipien zu organisieren.

Ein modularer Monolith?

Haben Sie einmal damit begonnen, Ihre bestehende Codebasis aufzuräumen, ist ein offensichtlicher nächster in Betracht zu ziehender Schritt, Ihre neu identifizierten Seams zu nehmen und sie als eigene Module zu extrahieren, womit Sie Ihren Monolithen zu einem *modularen Monolithen* machen. Sie haben immer noch eine einzelne Deployment-Einheit, aber diese besteht aus mehreren statisch verlinkten Modulen. Das genaue Wesen dieser Module hängt von Ihrem zugrunde liegenden Technologie-Stack ab – bei Java würde mein modularer Monolith aus mehreren JAR-Dateien bestehen, bei einer Ruby-App könnte es eine Sammlung von Ruby-Gems sein.

Wie wir schon kurz am Anfang des Buchs erwähnt haben, bringt das Aufteilen eines Monolithen in Module, an denen sich unabhängig voneinander entwickeln lässt, möglicherweise eine Reihe von Vorteilen mit sich, während gleichzeitig viele der Herausforderungen einer Microservices-Architektur vermieden werden – das

kann für viele Organisationen sehr verlockend sein. Ich habe schon mit einigen Teams gesprochen, die ihren Monolithen in einen modularen Monolithen umgewandelt haben, um von dort aus eventuell zu einer Microservices-Architektur zu wechseln – und die dann festgestellt haben, dass der modulare Monolith schon einen Großteil ihrer Probleme löst!

Inkrementelles Überarbeiten

Ich tendiere immer dazu, zunächst die bestehende Codebasis aufzubereiten, bevor ich damit beginne, Funktionalität einfach neu zu implementieren. Diesen Rat habe ich auch in meinem Buch *Building Microservices* bereits gegeben. Manchmal stellen Teams fest, dass ihnen dadurch schon so viele Vorteile geboten werden, dass sie Microservices gar nicht mehr benötigen!

Aber ich muss akzeptieren, dass in der Realität nur sehr wenige Teams darangehen, ihren Monolithen zu refaktorieren, bevor sie sich Microservices zuwenden. Stattdessen scheint es üblicher zu sein, dass Teams nach dem Ermitteln der Verantwortlichkeiten für die neu erstellten Microservices damit beginnen, eine neue Implementierung dieser Funktionalität unter Verzicht auf alten Code vorzunehmen.

Aber begeben wir uns damit nicht gerade wieder in Gefahr, die Probleme zu wiederholen, die mit Big-Bang-Rewrites verbunden sind, wenn wir unsere Funktionalität reimplementieren? Entscheidend ist, sicherzustellen, dass Sie immer nur kleine Funktionalitätselemente gleichzeitig neu schreiben und diese überarbeitete Funktionalität regelmäßig an Ihre Kunden liefern. Wenn es ein paar Tage oder Wochen dauert, das Verhalten des Service neu zu implementieren, ist das wahrscheinlich in Ordnung. Scheint sich der Zeitrahmen eher auf mehrere Monate auszudehnen, würde ich mein Vorgehen überdenken.

Migrations-Patterns

Ich habe viele Techniken kennengelernt, die im Rahmen einer Microservices-Migration eingesetzt wurden. Im Rest dieses Kapitels werden Sie diese Patterns kennenlernen. Wir werden besprechen, wo sie nützlich sein können und wie sie sich implementieren lassen. Denken Sie daran – wie bei allen Patterns gilt auch hier, dass das nicht immer »gute« Ideen sind. Bei jedem Pattern habe ich versucht, Ihnen genug Informationen mitzugeben, damit Sie entscheiden können, ob sie in Ihrem Kontext sinnvoll einsetzbar sind.

Stellen Sie sicher, dass Sie die Vor- und Nachteile jedes dieser Patterns verstanden haben. Sie sind nicht immer der »richtige« Weg.

Wir werden uns zunächst Techniken anschauen, mit denen man den Monolithen migrieren und ihn integrieren kann – dabei geht es vor allem darum, wo sich der

Anwendungscode befindet. Als Erstes schauen wir uns eine der nützlichsten und am häufigsten eingesetzten Techniken an: die Strangler Fig Application.

Pattern: Strangler Fig Application

Eine Technik, die häufig beim Umschreiben von Systemen zum Einsatz kommt, nennt sich *Strangler Fig Application* (*http://bit.ly/2p5xMKo*). Martin Fowler hat als Erster dieses Pattern dokumentiert, wobei er von einem bestimmten Feigentyp (Fig) inspiriert wurde, der sich selbst in den oberen Zweigen von Bäumen aussäht. Die Feige wächst dann Richtung Boden, um Wurzeln zu schlagen, und umhüllt den Baum teilweise. Der bestehende Baum ist dabei zunächst nur eine stützende Struktur für die neue Feige, aber zum Ende hin stirbt er ab und verrottet, während die neue Feige nun stabil genug ist, sich selbst zu stützen.

Im Kontext von Software ist die Parallele, dass unser neues System zuerst vom bestehenden System unterstützt wird und es umhüllt. Die Idee dahinter ist, dass Altes und Neues nebeneinander existieren können und dass das neue System Zeit zum Wachsen hat, bis es potenziell das alte System vollständig ersetzt. Entscheidender Vorteil bei diesem Pattern ist (wie wir noch sehen werden), dass es unser Ziel einer inkrementellen Migration hin zu einem neuen System unterstützt. Zudem haben wir so die Möglichkeit, die Migration zu pausieren oder sogar ganz zu stoppen und trotzdem aus dem bisher ausgelieferten neuen System Vorteile zu ziehen.

Wie wir gleich sehen werden, versuchen wir beim Implementieren dieser Idee für unsere Software, nicht nur inkrementelle Schritte hin zu unserer neuen Anwendungsarchitektur vorzunehmen, sondern auch sicherzustellen, dass jeder Schritt problemlos rückgängig zu machen ist, um sein Risiko zu minimieren.

Wie es funktioniert

Das *Strangler Fig Pattern* kam zwar häufig zum Einsatz, um von einem monolithischen System zu einem anderen zu migrieren, aber hier geht es um eine Migration von einem Monolithen zu einer Reihe von Microservices. Dazu kann das Kopieren von Code aus dem Monolithen gehören (sofern möglich), aber auch das Reimplementieren der fraglichen Funktionalität. Erfordert die Funktionalität zudem ein Persistieren eines Status, muss man sich überlegen, wie dieser Status zum neuen Service und möglicherweise auch wieder zurück migriert werden kann. Die Aspekte rund um die Daten werden wir uns in Kapitel 4 anschauen.

Beim Implementieren eines Strangler Fig Pattern gibt es drei Schritte (siehe Abbildung 3-1). Zuerst identifizieren Sie die Teile des bestehenden Systems, die Sie migrieren wollen. Sie werden sich überlegen müssen, welche Teile Sie zuerst angehen wollen, wobei Sie auf die Abwägungen aus Kapitel 2 zurückgreifen können. Dann müssen Sie diese Funktionalität in Ihrem neuen Microservice implementieren. Ist er bereit, müssen Sie die Aufrufe an den Monolithen zu Ihrem schicken neuen Microservice umleiten können.

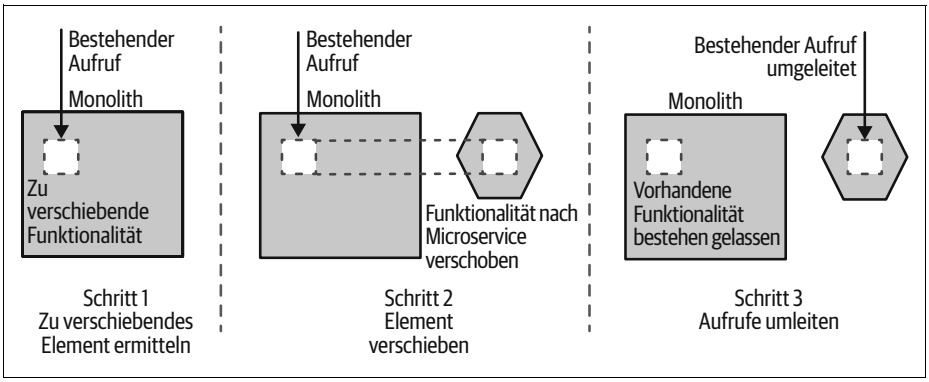

Abbildung 3-1: Ein Überblick über das Strangler Fig Pattern

Es lohnt sich, darauf hinzuweisen, dass die neue Funktionalität auch nach dem Deployen in eine Produktivumgebung technisch gesehen erst dann live ist, wenn der Aufruf an die verschobene Funktionalität umgeleitet wird. Das heißt, Sie könnten sich die Zeit nehmen, diese Funktionalität ordentlich zu bauen und länger an der Implementierung zu sitzen. Sie könnten diese Änderungen in die Produktivumgebung bringen und sich dabei sicher sein, dass sie noch nicht verwendet wird. Das erlaubt Ihnen, die Deployment- und Management-Aspekte Ihres neuen Service auf die Reihe zu bringen. Implementiert Ihr neuer Service dann irgendwann die gleiche Funktionalität wie Ihr Monolith, können Sie sich überlegen, ein Pattern wie *Parallel Run* (siehe weiter unten) einzusetzen, um zu prüfen, ob die neue Funktionalität wie gewünscht arbeitet.

Es ist wichtig, die Konzepte *Deployment* und *Release* voneinander zu trennen. Nur weil Software in eine gegebene Umgebung deployt wurde, heißt das nicht, dass sie auch wirklich von Kunden eingesetzt wird. Indem Sie die beiden Dinge als getrennte Konzepte behandeln, behalten Sie sich die Möglichkeit vor, Ihre Software in der Produktivumgebung zu überprüfen, bevor sie zum Einsatz kommt, womit Sie das Risiko beim Roll-out reduzieren. Strangler Fig, Parallel Run, Canary Release und Ähnliche sind Patterns, die darauf bauen, dass *Deployment* und *Release* getrennte Aktivitäten sind.

Entscheidend bei diesem Ansatz mittels einer Strangler-Anwendung ist nicht nur, dass wir inkrementell neue Funktionalität auf das neue System migrieren, sondern diese Änderung auch sehr einfach wieder rückgängig machen können, wenn es erforderlich ist. Denken Sie daran – wir machen alle Fehler, daher wollen wir Techniken nutzen, die es uns erlauben, Fehler nicht nur so günstig wie möglich zu machen (daher die vielen kleinen Schritte), sondern sie auch schnell wieder zu beheben.

Wird die zu extrahierende Funktionalität auch von anderer Funktionalität im Monolithen eingesetzt, müssen Sie deren Aufrufe ebenfalls ändern. Wir werden dafür weiter unten im Kapitel noch ein paar Techniken vorstellen.

Wo wir es einsetzen

Das Strangler Fig Pattern ermöglicht Ihnen, Funktionalität zu Ihrer neuen Systemarchitektur hin zu verschieben, ohne Ihr bestehendes System anzufassen oder gar Änderungen daran vorzunehmen. Das hat Vorteile, wenn am bestehenden Monolithen noch von anderen gearbeitet wird, da sich dadurch Konflikte vermeiden lassen. Außerdem ist es nützlich, wenn es sich beim Monolithen um ein Black-Box-System handelt – wie zum Beispiel Software von Fremdherstellern oder einen SaaS-Service.

Gelegentlich können Sie eine komplette End-to-End-Scheibe mit Funktionalität auf einmal extrahieren, wie in Abbildung 3-2 dargestellt. Das vereinfacht das Extrahieren deutlich (abgesehen von den Daten, auf die wir später noch eingehen werden).

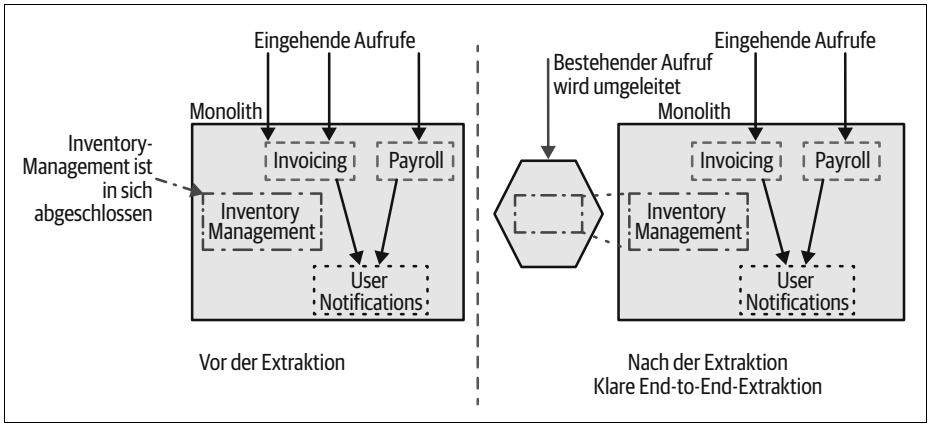

Abbildung 3-2: Geradlinige End-to-End-Extraktion der Inventory-Management-Funktionalität

Um eine solche saubere End-to-End-Extraktion durchzuführen, verlockt es vielleicht, größere Funktionalitätsbereiche zu übernehmen, um diesen Prozess zu vereinfachen. Das kann zu einem schwierigen Balanceakt führen – indem Sie größere Scheiben mit Funktionalität extrahieren, nehmen Sie mehr Arbeit auf sich, vereinfachen aber gleichzeitig eventuell die Herausforderungen bei der Integration.

Wollen Sie nur einen kleineren Happen zu sich nehmen, denken Sie vielleicht an »flachere« Extraktionen wie die in Abbildung 3-3. Hier extrahieren wir die Payroll-Funktionalität, obwohl sie andere Funktionalität nutzt, die im Monolithen verbleibt – in diesem Beispiel die Möglichkeit, User Notifications zu verschicken.

Statt auch die User-Notification-Funktionalität neu zu implementieren, stellen wir sie unserem neuen Microservice aus dem Monolithen heraus bereit – etwas, für das wir offensichtlich Änderungen am Monolithen selbst vornehmen müssen.

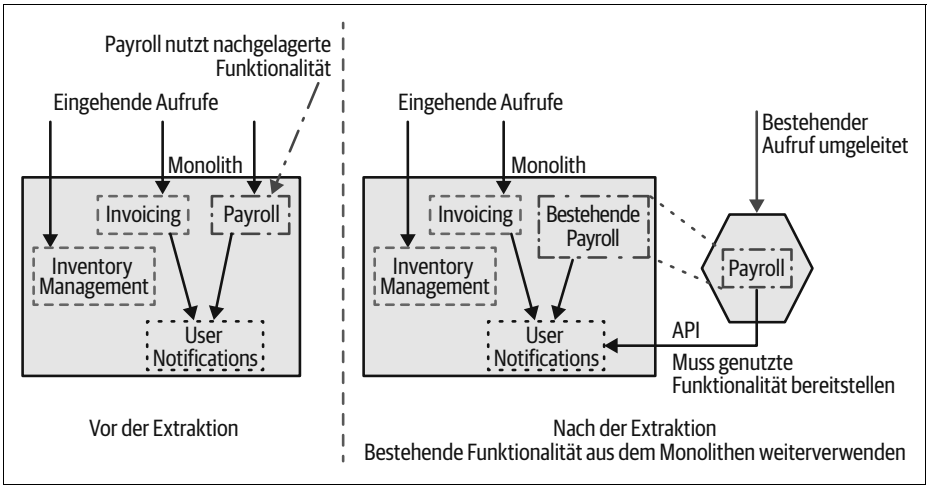

Abbildung 3-3: Funktionalität extrahieren, die immer noch den Monolithen nutzen muss

Damit der Strangler funktioniert, müssen Sie aber die eingehenden Aufrufe für die fragliche Funktionalität eindeutig auf das Element abbilden können, das Sie verschieben wollen. So würden wir in Abbildung 3-4 idealerweise die Fähigkeit zum Verschicken von User Notifications an unsere Kunden in einen neuen Service verschieben. Aber Notifications werden aufgrund verschiedenster eingehender Aufrufe an den bestehenden Monolithen verschickt. Daher können wir die Aufrufe von außerhalb des Systems nicht einfach alle umleiten. Stattdessen müssen wir uns eine Technik wie die im Abschnitt »Pattern: Branch by Abstraction« auf Seite 106 anschauen.

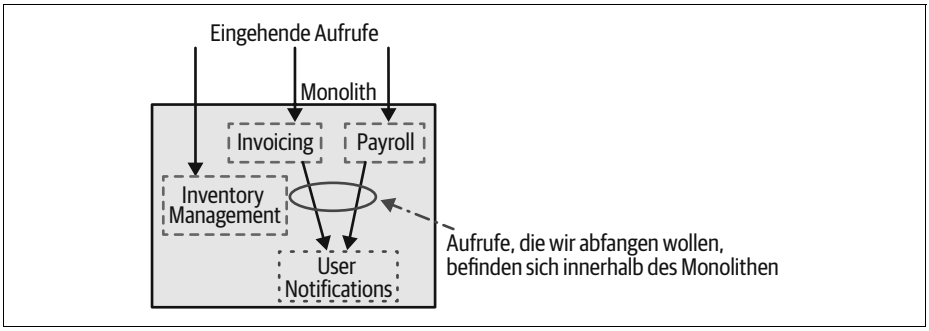

Abbildung 3-4: Das Strangler Fig Pattern funktioniert nicht so gut, wenn sich die zu verschiebende Funktionalität tiefer im bestehenden System befindet.

Auch werden Sie über die Natur der Aufrufe an das bestehende System nachdenken müssen. Wie wir bald sehen werden, ist ein Protokoll wie HTTP sehr aufgeschlossen für Umleitungen. HTTP selbst hat das Konzept einer transparenten Umleitung schon eingebaut, und es können Proxys genutzt werden, um den Inhalt eines eingehenden Requests auszuwerten und ihn entsprechend weiterzuleiten. Andere Arten von Protokollen, wie zum Beispiel manche RPCs, können weniger

gut damit umgehen. Je mehr Arbeit Sie in der Proxy-Schicht haben, um den eingehenden Aufruf zu verstehen und möglicherweise umzuwandeln, desto weniger nützlich wird diese Option.

Trotz dieser Einschränkungen hat sich die Strangler Fig Application schon oft als sehr nützliche Migrationstechnik erwiesen. Aufgrund ihrer Überschaubarkeit und des einfachen Ansatzes für inkrementelle Änderungen ist es oft meine erste Anlaufstelle bei den Überlegungen zum Migrieren eines Systems.

Beispiel: HTTP Reverse Proxy

HTTP besitzt einige interessante Features, unter anderem lässt es sich sehr leicht abfangen und so umleiten, dass dies für das aufrufende System völlig transparent geschieht. Ein bestehender Monolith mit einer HTTP-Schnittstelle ist daher für eine Migration durch das Strangler Fig Pattern sehr gut geeignet.

In Abbildung 3-5 sehen wir ein bestehendes monolithisches System, das eine HTTP-Schnittstelle bereitstellt. Diese Anwendung oder zumindest die HTTP-Schnittstelle kann ohne Benutzeroberfläche sein, dafür wird sie von einem UI aufgerufen. Auf jeden Fall ist das Ziel das gleiche: einen *HTTP Reverse Proxy* zwischen den eintreffenden Aufrufen und dem Monolithen einzufügen.

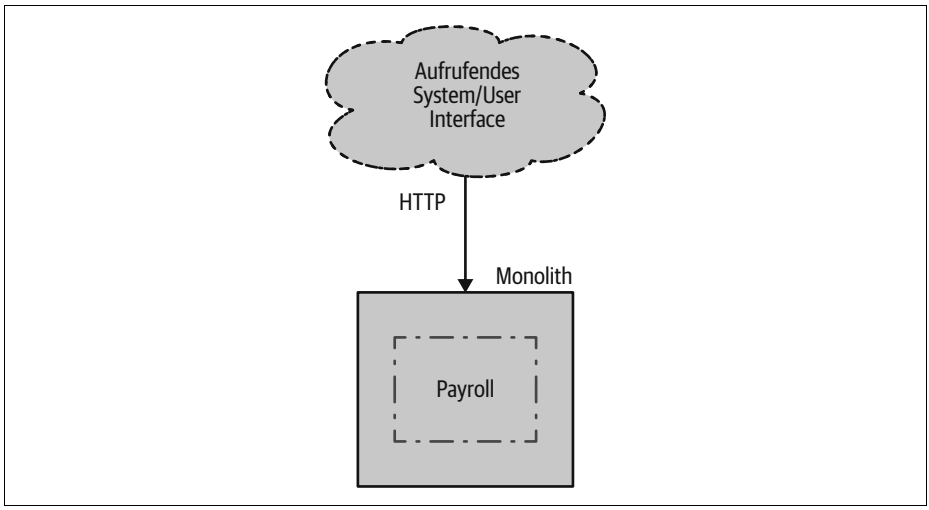

Abbildung 3-5: Ein einfacher Überblick über einen HTTP-getriebenen Monolithen vor der Implementierung eines Stranglers

Schritt 1: Proxy einfügen

Sofern Sie nicht schon einen passenden HTTP-Proxy im Einsatz haben, den Sie für Ihre Zwecke verwenden können, schlage ich vor, ihn als *Allererstes* aufzusetzen (siehe Abbildung 3-6). In diesem ersten Schritt wird der Proxy einfach alle Aufrufe ohne Änderung weiterleiten.

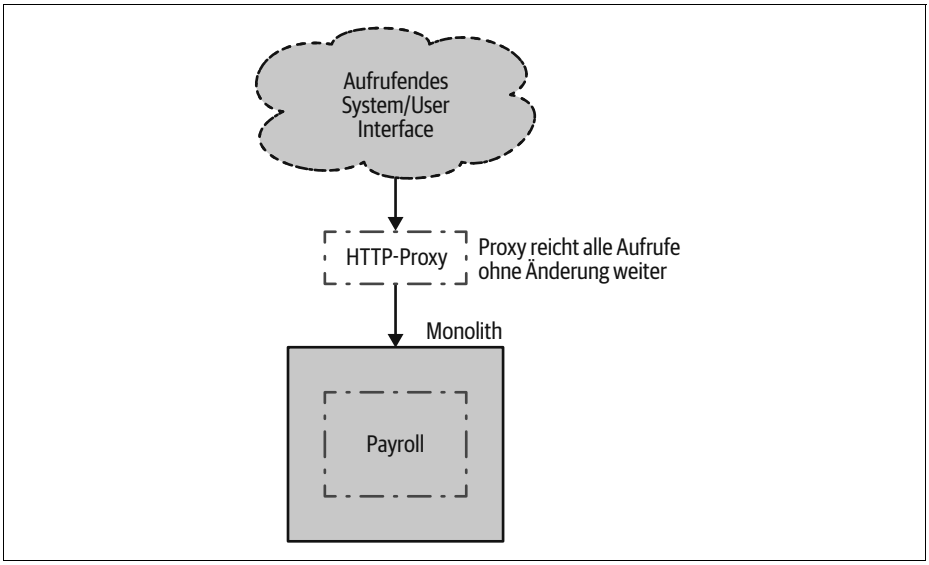

Abbildung 3-6: Schritt 1: Einen Proxy zwischen Monolith und aufrufendem System einfügen

Dieser Schritt wird es Ihnen erlauben, die Auswirkungen eines zusätzlichen Netzwerk-Hops zwischen den Aufrufen und dem Monolithen abzuschätzen, erforderliches Monitoring für Ihre neue Komponente einzurichten und diese einfach mal eine Weile laufen zu lassen. Mit Blick auf Latenzgesichtspunkte fügen wir für alle Aufrufe einen neuen Netzwerk-Hop und einen weiteren Prozess hinzu. Bei einem anständigen Proxy und Netzwerk werden Sie nur eine minimale Auswirkung auf die Latenz erhalten (vielleicht im Rahmen von ein paar Millisekunden), aber wenn das nicht der Fall ist, haben Sie jetzt die Möglichkeit, die Notbremse zu ziehen und das Problem zu identifizieren, bevor Sie weitermachen.

Haben Sie vor Ihrem Monolithen schon einen Proxy im Einsatz, können Sie diesen Schritt überspringen – stellen Sie aber sicher, dass Sie verstehen, wie dieser Proxy so umkonfiguriert werden kann, dass er später die Aufrufe umleitet. Ich schlage vor, zumindest mit der Redirection zu experimentieren, um sicherzustellen, dass sie wie gewünscht funktioniert, bevor Sie sich darauf verlassen, dass das später noch erledigt werden kann. Es wäre eine unschöne Überraschung, wenn Sie erst kurz vor dem Liveschalten herausfinden, dass das unmöglich ist!

Schritt 2: Funktionalität migrieren

Ist unser Proxy dort, wo er sein soll, können Sie als Nächstes damit beginnen, Ihren neuen Microservice zu extrahieren (siehe Abbildung 3-7).

Dieser Schritt lässt sich nochmals in mehrere Stufen unterteilen. Zuerst richten Sie einen grundlegenden Service ein und lassen ihn laufen, ohne dass irgendwelche Funktionalität implementiert ist. Ihr Service muss die Aufrufe für die entsprechende

Funktionalität annehmen, aber Sie können im Moment nur ein `501 Not Implemented` zurückliefern. Aber selbst auf dieser Stufe würde ich den Service bereits in die Produktivumgebung deployen. Damit können Sie sich schon mit dem produktiven Deployment-Prozess vertraut machen und den Service im relevanten System testen. Ihr neuer Service ist jetzt noch nicht *releast*, da Sie die aktuellen Aufrufe noch nicht dorthin umleiten. Im Endeffekt trennen wir so das Software-Deployment vom Software-Release – ein übliches Vorgehen, auf das wir später noch zu sprechen kommen.

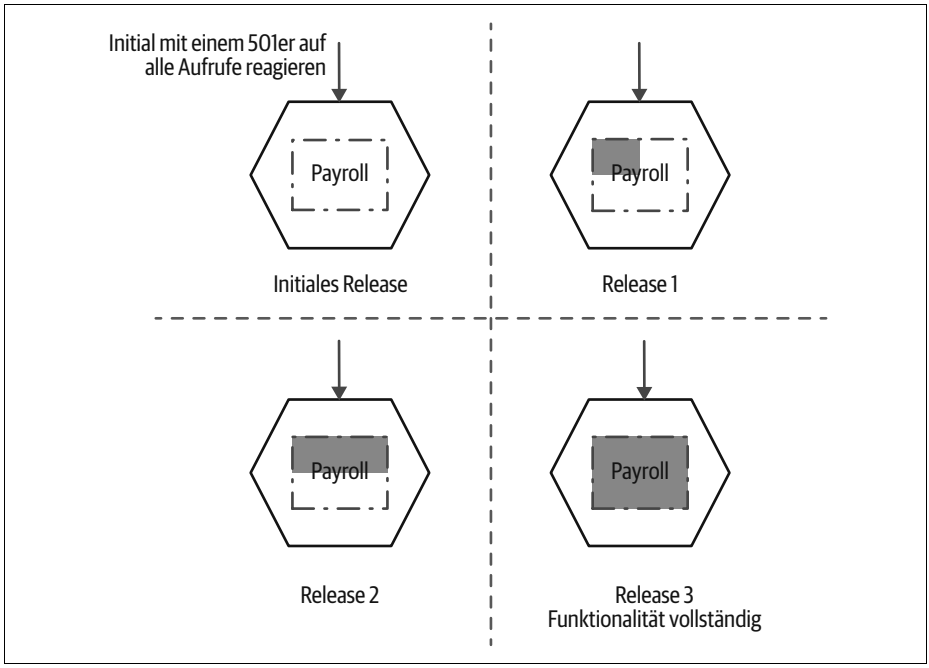

Abbildung 3-7: Schritt 2: Inkrementelles Implementieren der zu verschiebenden Funktionalität

Schritt 3: Aufrufe umleiten

Ist der Umzug der gesamten Funktionalität abgeschlossen, konfigurieren Sie den Proxy so um, dass er die Aufrufe umleitet (siehe Abbildung 3-8). Geht das aus welchen Gründen auch immer schief, können Sie die Umleitung wieder zurücknehmen – bei den meisten Proxys ist das ein schneller und einfacher Prozess, womit Sie ein schnelles Rollback erhalten.

Sie können sich dazu entscheiden, die Umleitung mit so etwas wie einem Feature-Toggle zu implementieren, durch den Ihr gewünschter Konfigurationsstatus offensichtlicher wird. Mit einem Proxy zum Umleiten der Aufrufe haben Sie auch die Möglichkeit, ein inkrementelles Roll-out der neuen Funktionalität durch ein *Canary Rollout* oder sogar einen vollständigen *Parallel Run* (den wir später noch besprechen) umzusetzen.

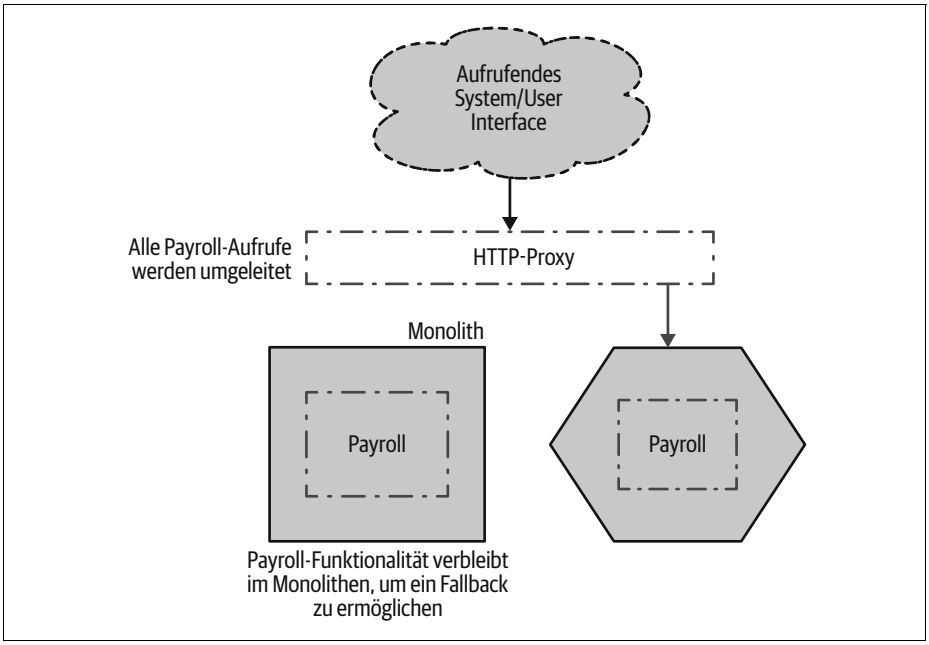

Abbildung 3-8: Schritt 3: Die Aufrufe für die Payroll-Funktionalität umleiten und die Migration damit abschließen

Daten?

Bisher haben wir noch nicht über die Daten gesprochen. Was passiert in Abbildung 3-8, wenn unser neu migrierter Payroll-Service auf die Daten zugreifen muss, die zurzeit in der Datenbank des Monolithen vorgehalten werden? Möglichkeiten dafür werden wir in Kapitel 4 besprechen.

Proxy-Optionen

Es hängt teilweise vom durch den Monolithen verwendeten Protokoll ab, wie Sie den Proxy implementieren. Nutzt der bestehende Monolith HTTP, ist das schon mal ein guter Anfang. Es ist so verbreitet, dass Sie bei der Umleitung unter sehr vielen Optionen wählen können. Ich würde einen dedizierten Proxy wie NGINX vorschlagen, der genau für solche Anwendungsfälle geschaffen wurde und der unglaublich viele Umleitungsmechanismen bietet, die umfassend getestet sind und auch eine gute Performance liefern.

Manche Umleitungen werden einfacher sein als andere. Denken Sie an Redirections anhand von URI-Pfaden, vielleicht für den Einsatz von REST-Ressourcen. In Abbildung 3-9 verschieben wir die gesamte Invoice-Ressource zu unserem neuen Service, was sich leicht aus dem URI-Pfad parsen lässt. Wenn jedoch das bestehende System Informationen über die Art der aufgerufenen Funktionalität irgendwo im Hauptteil

der Anfrage vergräbt (vielleicht in einem Form-Parameter), muss unsere Redirection-Regel in der Lage sein, einen Parameter im POST einzuschalten – etwas, das möglich, aber komplizierter ist. Es lohnt sich auf jeden Fall, die für Ihren Proxy möglichen Optionen unter die Lupe zu nehmen, falls Sie sich solch einer Situation gegenübersehen.

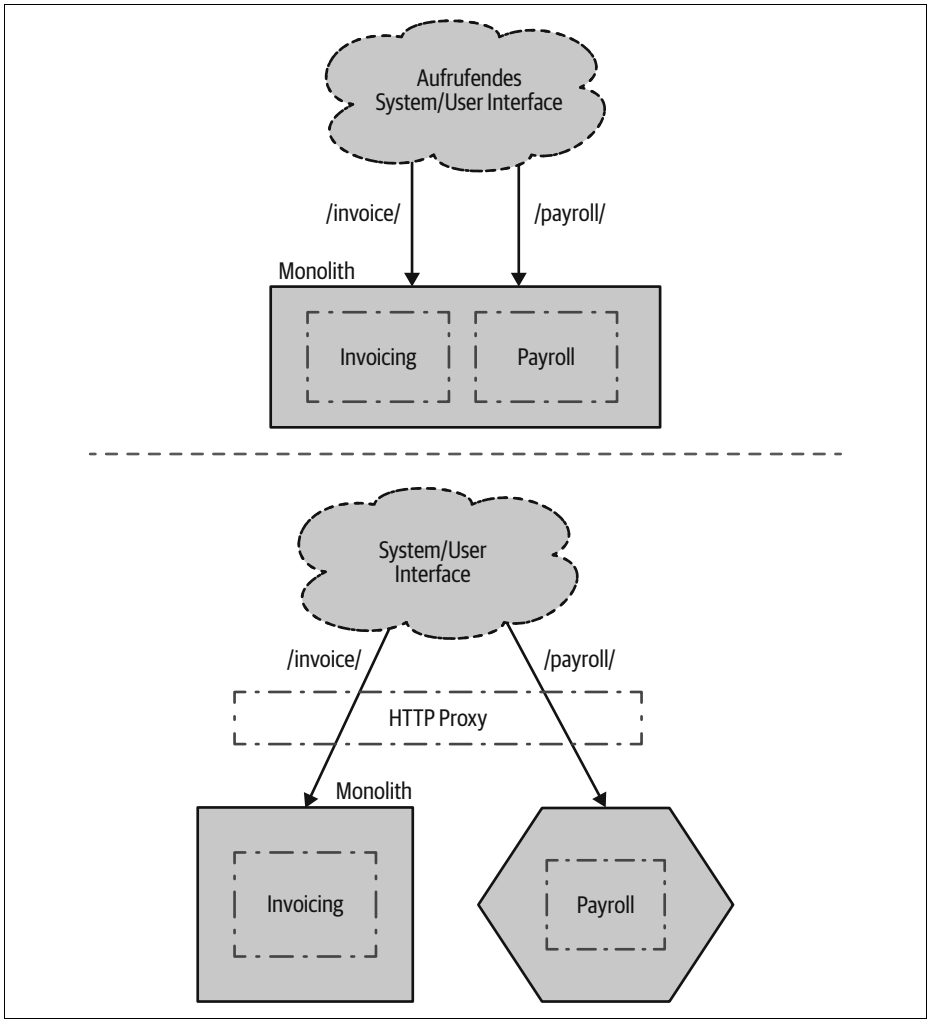

Abbildung 3-9: Umleitung bei Ressourcen

Ist das Eingreifen und Umleiten komplexer oder nutzt der Monolith ein weniger unterstütztes Protokoll, sind Sie vielleicht geneigt, selbst etwas zu programmieren, aber Sie sollten dabei sehr vorsichtig sein. Ich habe schon einige Netzwerk-Proxys per Hand geschrieben (in Java und Python). Es mag zwar sein, dass das mehr über meine Coding-Fähigkeiten denn über etwas anderes aussagt, aber die Proxys waren immer fürchterlich ineffizient und brachten deutliche Verzögerungen ins System

ein. Heutzutage würde ich in solchen Situationen eher darüber nachdenken, einen dedizierten Proxy um das gewünschte Verhalten zu erweitern – NGINX erlaubt Ihnen beispielsweise, in Lua geschriebenen Code einzusetzen, um eigenes Verhalten hinzuzufügen.

Inkrementelles Roll-out

Wie Sie in Abbildung 3-10 sehen können, ermöglicht diese Technik über eine Folge von kleinen Schritten Änderungen an der Architektur, die jeweils parallel zu anderen Arbeiten am System erfolgen können.

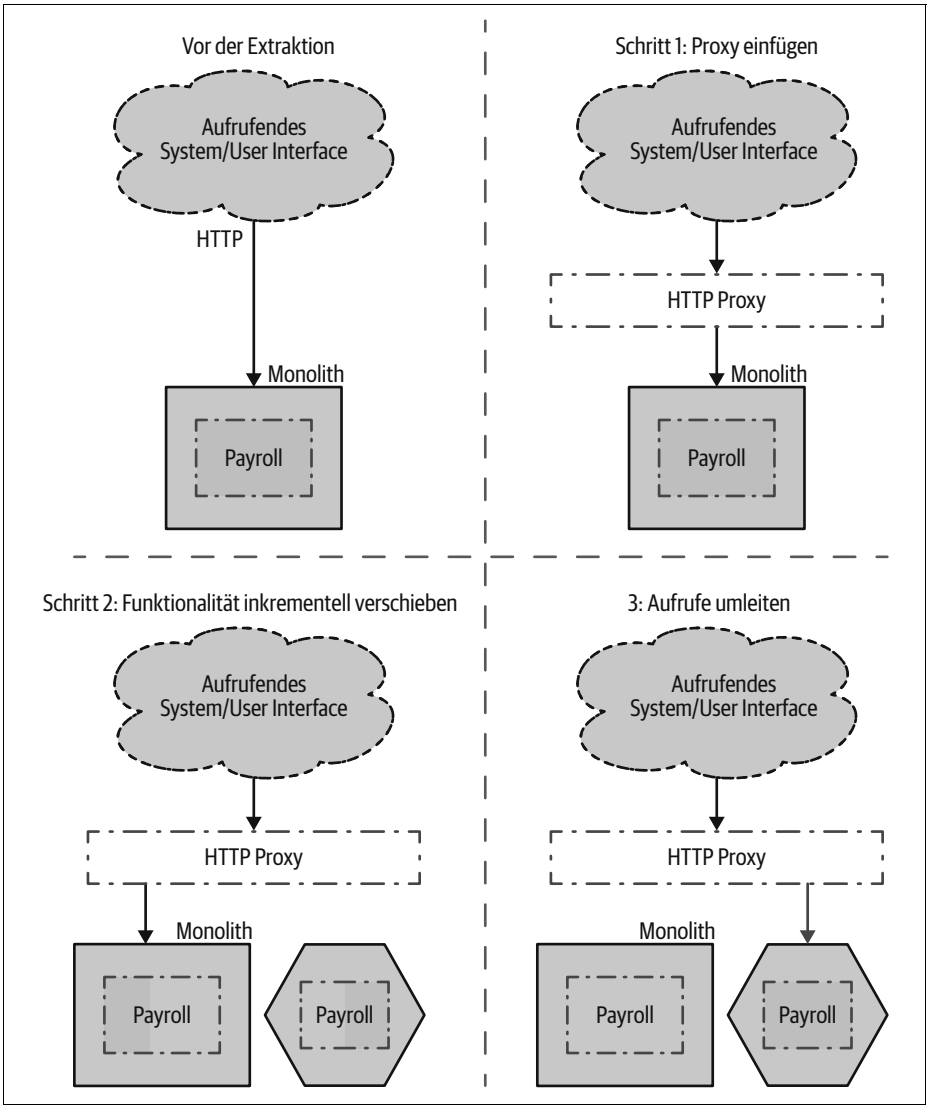

Abbildung 3-10: Ein Überblick über das Implementieren eines HTTP-basierten Stranglers

Vielleicht befürchten Sie, dass der Wechsel zu einer neuen Implementierung der Payroll-Funktionalität immer noch zu groß ist – dann können Sie in kleineren Schritten vorgehen. Sie könnten sich beispielsweise überlegen, nur einen Teil der Payroll-Funktionalität zu migrieren und die entsprechenden Aufrufe unterschiedlich umzuleiten – ein Teil des Verhaltens ist im Monolithen implementiert, ein Teil im Microservice (siehe Abbildung 3-11). Das kann zu Problemen führen, wenn die Funktionalität sowohl im Monolithen als auch im Microservice auf dieselben Daten zugreifen muss, denn dann ist eine gemeinsame Datenbank notwendig – mit allen damit verbundenen Problemen.

Es ist aber kein Big-Bang-Plattformwechsel nötig, bei dem alles angehalten wird. So wird es viel einfacher, diese Arbeit in Stufen zu unterteilen, die parallel zu anderen Arbeiten ausgeliefert werden. Statt Ihr Backlog in »Feature-« und »Technik«-Storys zu unterteilen, führen Sie die Arbeiten zusammen. Machen Sie beim Architekturwechsel inkrementelle Schritte, während Sie gleichzeitig neue Features ausliefern!

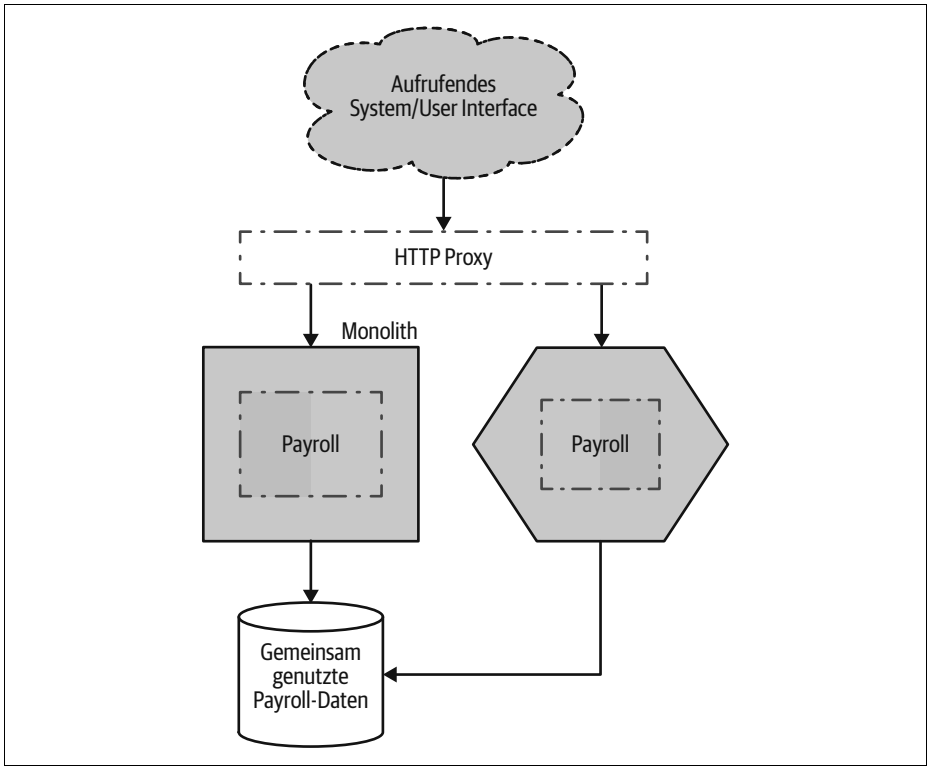

Abbildung 3-11: Zugriff auf gemeinsam genutzte Daten

Protokolle wechseln

Sie könnten Ihren Proxy auch nutzen, um das Protokoll zu wechseln. Vielleicht stellen Sie aktuell eine SOAP-basierte HTTP-Schnittstelle bereit, aber Ihr neuer

Microservice wird stattdessen eine gRPC-Schnittstelle unterstützen. Sie könnten dann den Proxy so konfigurieren, dass er Requests und Responses entsprechend umwandelt (siehe Abbildung 3-12).

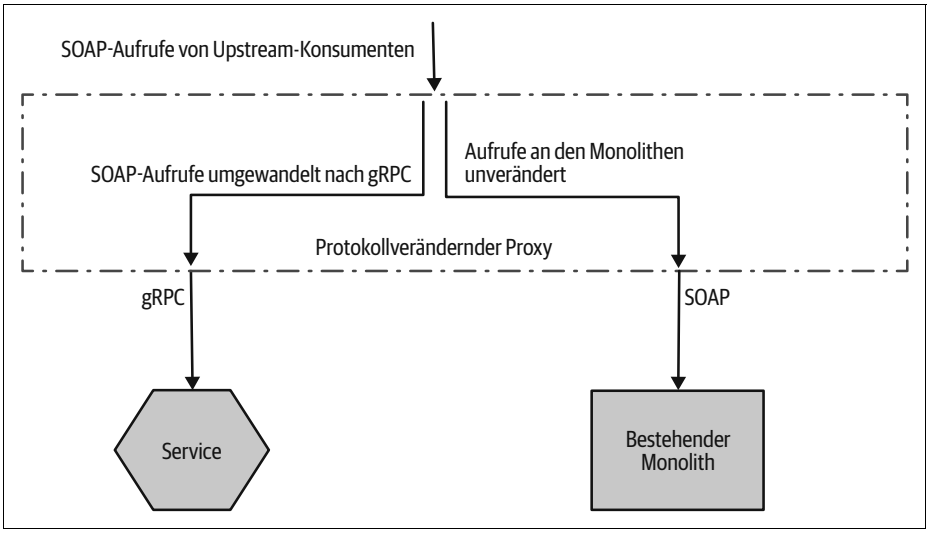

Abbildung 3-12: Einsatz eines Proxys, um das Kommunikationsprotokoll zu ändern (als Teil einer Strangler-Migration)

Bei diesem Ansatz habe ich so meine Bedenken, vor allem aufgrund der Komplexität und Logik, die im Proxy selbst aufgebaut werden muss. Für einen einzelnen Service mag das nicht so schlimm sein, aber wenn Sie das Protokoll für viele Services anpassen, geschieht immer mehr im Proxy. Wir optimieren typischerweise auf eine unabhängige Deploybarkeit unserer Services, aber wenn wir eine gemeinsame Proxy-Schicht haben, an der mehrere Teams arbeiten müssen, kann das das Erstellen und Deployen von Änderungen ausbremsen. Wir müssen darauf achten, dass wir keine neue Schicht einführen, um die es Streit geben kann. Es gibt bei Diskussionen zu Microservices das oft zitierte Mantra: »Halte die Verbindungen dumm und die Endpunkte klug.« Wir wollen die Menge an Funktionalität reduzieren, die in gemeinsame Middleware-Schichten gesteckt wird, da dies die Feature-Entwicklung wirklich verlangsamen kann.

Wollen Sie das verwendete Protokoll migrieren, empfehle ich eher, das Mapping im Service selbst vorzunehmen – indem er sowohl das alte wie auch das neue Kommunikationsprotokoll unterstützt. Innerhalb des Service werden Aufrufe über das alte Protokoll einfach auf das neue abgebildet (siehe Abbildung 3-13). Damit vermeiden Sie, Änderungen in Proxy-Schichten vornehmen zu müssen, die auch von anderen Services genutzt werden, und der Service hat die volle Kontrolle darüber, wie sich diese Funktionalität mit der Zeit ändert. Sie können Microservices als eine Sammlung von Funktionalität an einem Netzwerkendpunkt betrachten. Vielleicht stellen Sie die gleiche Funktionalität verschiedenen Konsumenten auf unter-

schiedlichen Wegen bereit – indem Sie unterschiedliche Nachrichten- oder Request-Formate innerhalb dieses Service unterstützen, können Sie die verschiedenen Anforderungen der aufrufenden Konsumenten bedienen.

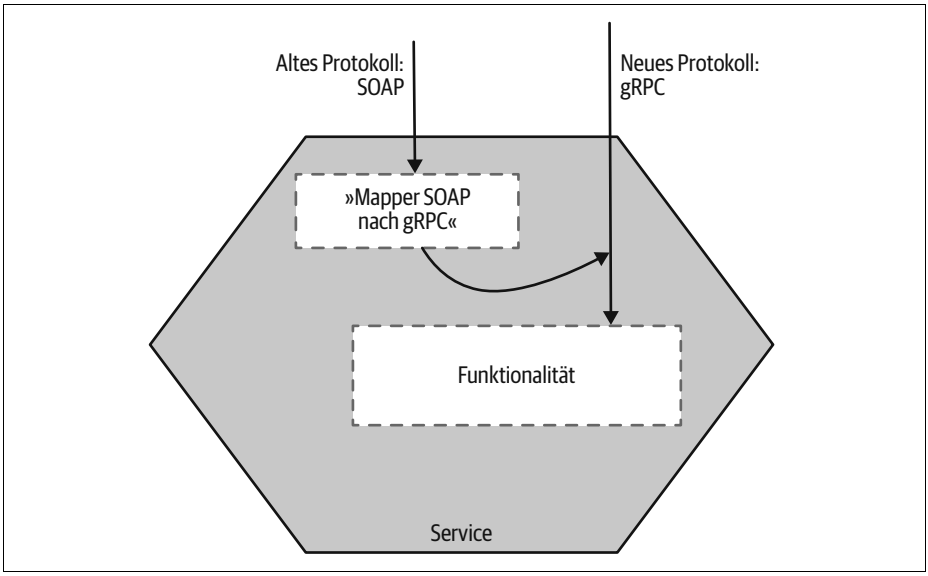

Abbildung 3-13: Wollen Sie den Protokolltyp ändern, überlegen Sie sich, ob Sie einen Service seine Fähigkeiten über mehrere Protokolltypen anbieten lassen.

Indem Sie das Mapping für servicespezifische Requests und Responses in den Service verlagern, bleibt die Proxy-Schicht einfacher und generischer. Zudem haben Sie durch einen Service, der beide Typen von Endpunkten unterstützt, mehr Zeit, anfragende Konsumenten zu migrieren, bevor Sie die alte API möglicherweise abschalten.

Service Meshes

Bei Square wurde ein hybrider Ansatz für dieses Problem verfolgt.[1] Dort entschied man sich dafür, vom selbst entwickelten RPC-Mechanismus für die Service-to-Service-Kommunikation wegzugehen und gRPC zu übernehmen – ein gut unterstütztes Open-Source-RPC-Framework mit einem großen Ökosystem. Damit das so schmerzlos wie möglich über die Bühne ging, wollte Square die Änderungen in jedem Service so gering wie möglich halten. Dazu nutzte die Firma ein Service Mesh.

Bei einem *Service Mesh* (siehe Abbildung 3-14) kommuniziert jede Serviceinstanz mit anderen Serviceinstanzen über ihren eigenen dedizierten lokalen Proxy. Jede Proxy-Instanz kann passend für die Serviceinstanz konfiguriert werden, mit der sie zusammenarbeitet. Sie können diese Proxys zudem über eine Control Plane zentral kontrollieren und überwachen. Da es keine zentrale Proxy-Schicht gibt, vermeiden

1 Eine umfassendere Erläuterung finden Sie in »The Road to an Envoy Service Mesh« (*https://squ.re/2nts1Gc*) von Snow Petterson im Entwicklerblog von Square.

Sie die Fallstricke einer gemeinsam genutzten »klugen« Verbindung – jeder Service kann für seinen eigenen Teil der Service-to-Service-Verbindung verantwortlich sein, wenn das erforderlich ist. Es sei darauf hingewiesen, dass die Firma aufgrund der Art und Weise, wie sich ihre Architektur entwickelt hat, ihr eigenes Service Mesh auf Basis des Open-Source-Proxys Envoy erschuf, der ihren Anforderungen entsprach, statt auf fertige Lösungen wie Linkerd oder Istio zurückzugreifen.

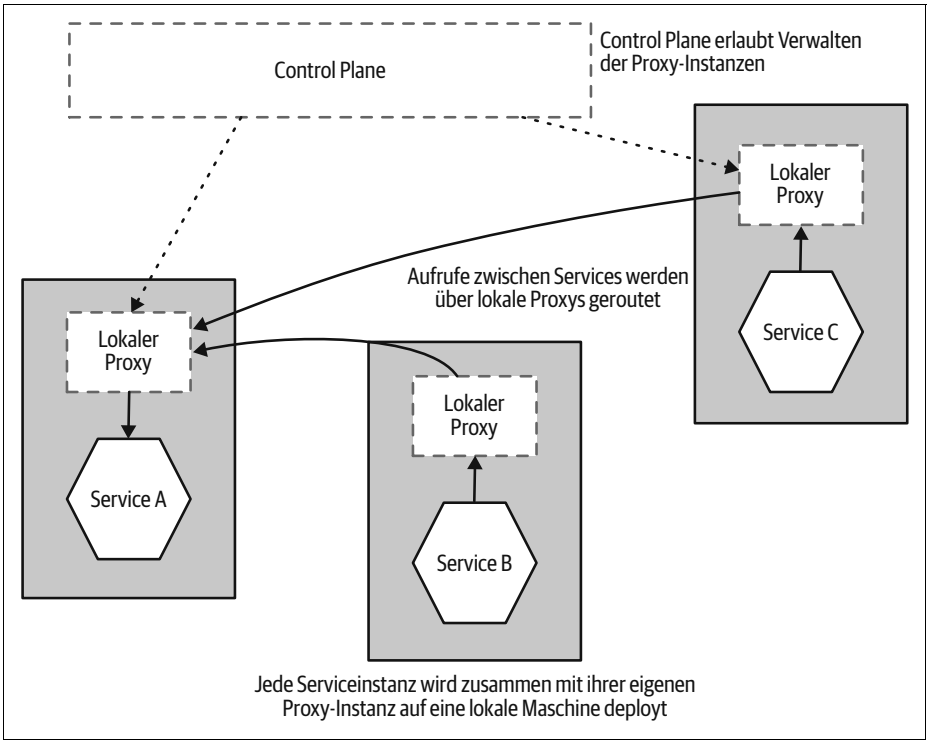

Abbildung 3-14: Überblick über ein Service Mesh

Service Meshes werden immer beliebter, und konzeptionell ist diese Idee meiner Meinung nach sehr gut. Sie können häufig auftretende Probleme bei der Service-to-Service-Kommunikation behandeln. Mich verunsichert lediglich ein wenig, dass es trotz der vielen Arbeit durch sehr kluge Menschen ziemlich lange gedauert hat, bis sich die Tools in diesem Bereich stabilisiert haben. Istio scheint der klare Sieger zu sein, aber es ist nicht die einzige Option – jede Woche scheinen neue Tools zu entstehen. Ich rate dazu, dem Bereich der Service Meshes möglichst noch ein wenig mehr Zeit zu geben, bis er sich stabilisiert hat, bevor Sie eine Wahl treffen.

Beispiel: FTP

Ich habe zwar recht lange über den Einsatz des Strangler-Patterns für HTTP-basierte Systeme gesprochen, aber nichts hält Sie davon ab, auch andere Formen von

Kommunikationsprotokollen abzufangen und umzuleiten. Die schweizerische Immobilienfirma Homegate hat eine Abwandlung dieses Musters genutzt, um die Art und Weise zu verändern, wie die Kunden neue Immobilienangebote einstellen.

Die Kunden von Homegate haben ihre Angebote per FTP hochgeladen, wobei ein bestehendes monolithisches System die hochgeladenen Dateien verarbeitete. Die Firma wollte gern zu Microservices wechseln und dabei auch einen neuen Upload-Mechanismus unterstützen. Statt nur Batch-FTP-Uploads anzubieten, sollte eine REST-API zum Einsatz kommen, die zu einem bald zu ratifizierenden Standard passte.

Die Immobilienfirma wollte aus Kundensicht nichts verändern – die Anpassungen sollten nahtlos vonstattengehen. FTP wurde also weiterhin als der Mechanismus gebraucht, über den Kunden mit dem System zumindest aktuell noch interagieren. Die Firma hat dann FTP-Uploads abgefangen (indem Änderungen im FTP-Serverlog erkannt wurden) und neu hochgeladene Dateien an einen Adapter weitergeleitet, der sie in Requests für die neue REST-API umwandelte (siehe Abbildung 3-15).

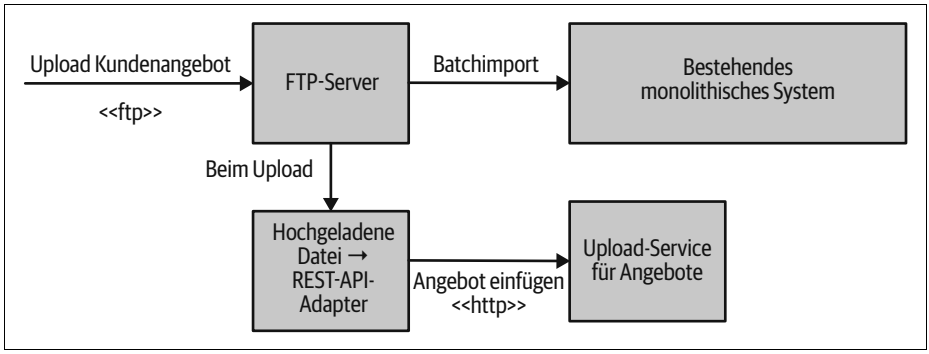

Abbildung 3-15: Abfangen eines FTP-Uploads und Umleiten zu einem neuen Angebotsservice für Homegate

Aus Kundensicht hatte sich der Upload-Prozess nicht verändert. Der Vorteil war aber, dass der neue Service, der sich um die hochgeladenen Dateien kümmerte, die Daten schneller veröffentlichen konnte, sodass die Kunden ihre Angebote schneller live schalten konnten. Es gab dann einen Plan, die neue REST-API später direkt den Kunden zur Verfügung zu stellen. Interessanterweise waren in dieser Zeit beide Upload-Mechanismen verfügbar. So konnte das Team sicherstellen, dass beide Wege korrekt funktionierten. Das ist ein tolles Beispiel für ein Pattern, das wir später im Abschnitt »Pattern: Parallel Run« auf Seite 114 behandeln werden.

Beispiel: Message Interception

Bisher haben wir uns das Abfangen von synchronen Aufrufen angeschaut. Aber wie sieht es aus, wenn Ihr Monolith durch eine andere Art von Protokoll gesteuert wird, zum Beispiel durch das Empfangen von Nachrichten über einen Message

Broker? Das grundlegende Muster ist das gleiche – wir brauchen eine Methode, die die Aufrufe abfängt und sie an unseren neuen Microservice weiterleitet. Der Hauptunterschied liegt in der Art des Protokolls selbst.

Inhaltsbasiertes Routing

In Abbildung 3-16 empfängt unser Monolith eine Vielzahl von Nachrichten, von denen wir nur eine Untermenge abfangen müssen.

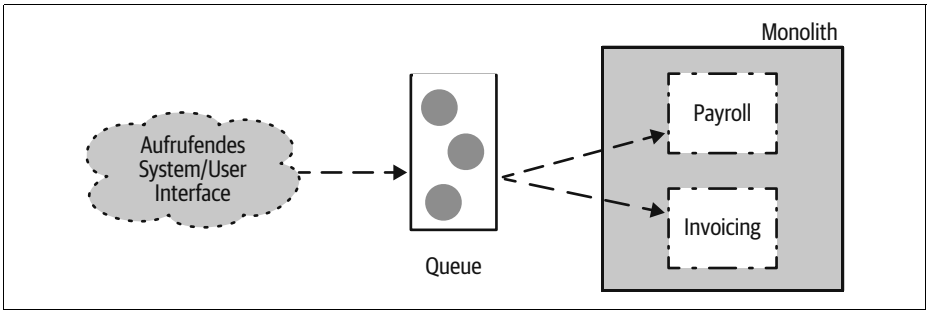

Abbildung 3-16: Ein Monolith empfängt Aufrufe über eine Queue.

Am einfachsten wäre es, alle Nachrichten abzufangen und abhängig vom Ziel herauszufiltern (siehe Abbildung 3-17). Das ist im Prinzip eine Implementierung des Content-based Router Pattern, das in *Enterprise Integration Patterns* beschrieben ist.[2]

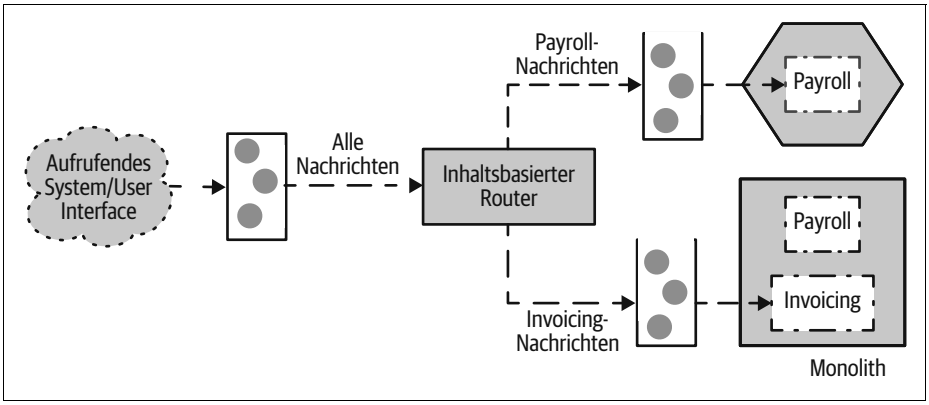

Abbildung 3-17: Ein inhaltsbasierter Router fängt Nachrichtenaufrufe ab.

Diese Technik erlaubt uns, den Monolithen unangetastet zu lassen, aber wir fügen eine weitere Queue in den Anforderungsweg ein, die für zusätzliche Latenz sorgen kann und um die wir uns auch noch kümmern müssen. Die andere Frage ist, wie »klug« wir unsere Messaging-Schicht machen. In Kapitel 4 von *Building Microsystems* habe ich über die Herausforderungen gesprochen, die entstehen können,

2 Bobby Woolf und Gregor Hohpe, *Enterprise Integration Patterns* (Addison-Wesley, 2003).

wenn es zu viele kluge Elemente im Netz zwischen Ihren Services gibt, da sich das System dadurch schlechter verstehen und ändern lässt. Stattdessen rate ich Ihnen, dem Mantra der »klugen Endpunkte und dummen Verbindungen« zu folgen. Der inhaltsbasierte Router stellt ohne Frage ein »kluges Verbindungselement« dar – er lässt die Komplexität beim Lenken der Aufrufe zwischen unseren Systemen wachsen. In manchen Situationen ist das eine ausgesprochen nützliche Technik, aber es liegt an Ihnen, eine gute Balance zu finden.

Selektiver Konsum

Eine Alternative wäre, den Monolithen zu ändern und ihn die Nachrichten ignorieren zu lassen, die von unserem neuen Service empfangen werden sollen (siehe Abbildung 3-18). Hier teilen sich unser neuer Service und unser Monolith die gleiche Queue, und lokal nutzen sie irgendeine Form von Mustererkennung, um nur auf die Nachrichten zu lauschen, die sie interessieren. Solch eine Form von Filter ist eine häufige Anforderung bei Message-basierten Systemen, und sie lässt sich über so etwas wie einen Message Selector in JMS oder entsprechende Technologien auf anderen Plattformen implementieren.

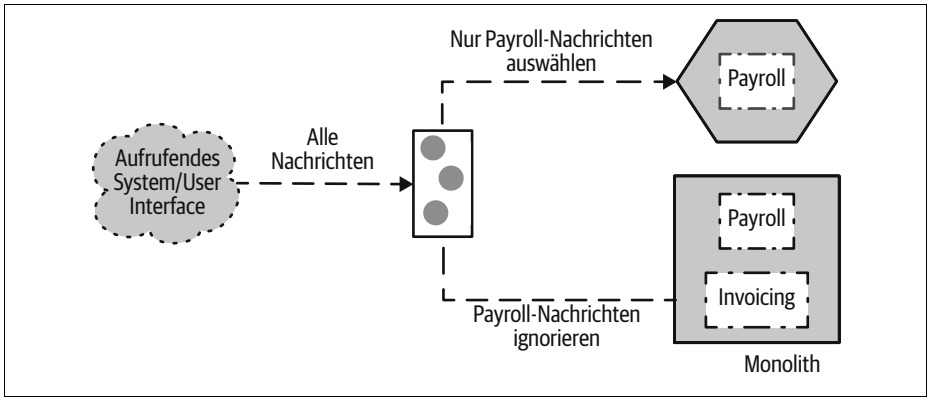

Abbildung 3-18: Der Router ist »dumm«, dafür filtern die Zielsysteme die für sie relevanten Nachrichten heraus.

Durch diesen Filteransatz benötigen Sie keine zusätzliche Queue, dafür sehen Sie sich aber anderen Herausforderungen gegenüber. So kann die zugrunde liegende Messaging-Technologie dazu in der Lage sein, solch eine einzelne Queue Subscription gemeinsam zu nutzen – oder auch nicht (allerdings ist das ein so verbreitetes Feature, dass es mich schon wundern würde). Wollen Sie die Aufrufe umleiten, sind zwei Änderungen notwendig, die ausreichend gut koordiniert sind. Ihr Monolith muss das Lesen der Aufrufe für den neuen Service beenden, und der Service muss sie übernehmen. Genauso sind für ein Rückgängigmachen der Umleitung ebenfalls zwei Änderungen erforderlich.

Je mehr unterschiedliche Konsumenten Sie für die gleiche Queue haben und je komplexer die Filterregeln werden, desto problematischer kann das Ganze werden.

Sie können sich leicht eine Situation vorstellen, in der zwei Konsumenten aufgrund überlappender Regeln die gleichen Nachrichten empfangen oder umgekehrt – manche Nachrichten werden ganz ignoriert. Aus diesem Grund würde ich den selektiven Konsum nur bei wenigen Konsumenten und/oder einfachen Filterregeln in Betracht ziehen. Ein inhaltsbasierter Routing-Ansatz wird sinnvoller sein, wenn die Anzahl an Konsumenten wächst, auch wenn Sie dann die schon erwähnten Nachteile in Kauf nehmen müssen – insbesondere das Problem der »schlauen Verbindungen«.

Die zusätzliche Komplexität durch entweder diese Lösung oder durch das inhaltsbasierte Routing entsteht dadurch, dass wir bei einer asynchronen Request-Response-Kommunikation sicherstellen müssen, dass wir den Request an den Client zurückleiten können, ohne dass er etwas von der Änderung bemerkt. Es gibt andere Optionen für das Routen von Aufrufen in Message-getriebenen Systemen, von denen viele bei der Implementierung des Strangler Fig Pattern für die Migration helfen können. Ich lege Ihnen dafür *Enterprise Integration Patterns* als sehr gute Ressource ans Herz.

Andere Protokolle

Ich hoffe, Sie haben aus diesem Beispiel mitgenommen, dass es viele Möglichkeiten gibt, Aufrufe an Ihren existierenden Monolithen umzuleiten, auch wenn Sie unterschiedliche Protokollarten einsetzen. Was tun Sie, wenn Ihr Monolith durch einen Batch-File-Upload gesteuert wird? Sie fangen die Batchdateien ab, extrahieren die Aufrufe, die Sie umleiten wollen, und entfernen sie aus der Datei, bevor Sie sie weiterleiten. Manche Mechanismen können diesen Prozess wirklich verkomplizieren, und es ist viel einfacher, wenn Sie so etwas wie HTTP einsetzen, aber mit ein wenig kreativem Denken kann das Strangler Fig Pattern in erstaunlich vielen Situationen eingesetzt werden.

Andere Beispiele für das Strangler Fig Pattern

Das Strangler Fig Pattern ist in allen Situationen hilfreich, in denen Sie ein bestehendes System nach und nach auf eine neue Plattform bringen wollen. Sein Einsatzzweck ist nicht nur auf Teams beschränkt, die Microservices-Architekturen implementieren. Das Pattern wurde schon lange Zeit genutzt, bevor Martin Fowler es 2004 niederschrieb. Bei meinem früheren Arbeitgeber ThoughtWorks haben wir es oft verwendet, um mit seiner Hilfe monolithische Systeme neu aufzubauen. Paul Hammant hat auf seinem Blog eine Liste von Projekten erstellt (*http://bit.ly/2paBpyP*), in denen dieses Pattern zum Einsatz kam. Dazu gehört ein Blotter[3] einer

3 Als Blotter (von engl.: Löschpapier) wird im Handel mit Wertpapieren und Commodities eine temporäre Aufstellung der offenen Positionen bezeichnet. Der Begriff stammt aus der Zeit vor der Einführung des computerbasierten Tradings, als Händler die offenen Geschäfte auf ihrer Schreibunterlage notierten. (Quelle: *https://de.wikipedia.org/wiki/Blotter_(Trading)*)

Handelsfirma, eine Buchungsanwendung einer Fluggesellschaft, ein Ticketingsystem eines Eisenbahnunternehmens und ein Kleinanzeigenportal.

Verhaltensänderung während der Migration

Hier und an anderer Stelle im Buch konzentriere ich mich auf Patterns, die ich ausgewählt habe, weil man mit ihnen ein bestehendes System inkrementell auf eine Microservices-Architektur migrieren kann. Einer der Hauptgründe dafür ist, dass Sie so die Migrationsaufgaben mit weiterhin auszuliefernden Features mischen können. Aber es gibt trotzdem ein Problem, das auftritt, wenn Sie das Systemverhalten ändern oder erweitern wollen, das gerade aktiv migriert wird.

Stellen Sie sich beispielsweise vor, wir nutzen das Strangler Fig Pattern, um unsere bestehende Payroll-Funktionalität aus unserem Monolithen herauszuschneiden. Das Pattern erlaubt uns, dies in mehreren Schritten zu tun, wobei wir theoretisch jeden Schritt auch wieder rückgängig machen können. Würden wir einen neuen Payroll-Service an unsere Kunden ausliefern und ein Problem darin finden, könnten wir die Aufrufe auch wieder zurück an das alte System lenken. Das funktioniert gut, wenn die Payroll-Funktionalität von Monolith und Microservice identisch ist – aber was, wenn wir das Payroll-Verhalten als Teil der Migration verändert haben?

Würden im Payroll-Microservice ein paar Fehler behoben sein, die nicht in die entsprechende Funktionalität des Monolithen zurückportiert worden sind, würde ein Rollback auch dazu führen, dass diese Fehler wieder auftauchen. Das kann noch problematischer werden, wenn Sie den Payroll-Microservice um neue Funktionalität erweitert haben – ein Rollback würde dann dafür sorgen, dass Ihre Kunden bestimmte Features nicht mehr erhalten.

Hier gibt es keine einfache Lösung. Erlauben Sie Veränderungen an der zu übertragenden Funktionalität, bevor die Migration abgeschlossen ist, müssen Sie in Kauf nehmen, dass Rollbacks schwieriger werden. Erlauben Sie keinerlei Änderungen während der Migration, ist es einfacher. Aber je länger die Migration dauert, desto schwerer kann es werden, einen »Feature Freeze« in diesem Teil des Systems durchzuhalten – gibt es die Anforderung, dass sich ein Teil Ihres Systems ändert, wird diese Anforderung nicht einfach wieder verschwinden. Je länger es dauert, bis die Migration abgeschlossen ist, desto mehr Druck werden Sie bekommen, »nur mal eben dieses Feature mit reinzunehmen, wo Sie doch sowieso gerade daran arbeiten«. Je kleiner jede Migration ist, desto weniger Druck werden Sie diesbezüglich haben.

Versuchen Sie beim Migrieren von Funktionalität, alle Änderungen am zu übertragenden Verhalten auszumerzen – verzögern Sie neue Features oder Bugfixes, bis die Migration abgeschlossen ist, wenn Sie können. Ansonsten erschweren Sie die Möglichkeit, Änderungen an Ihrem System zurückzurollen.