# Lean & Clean – Microservices in der Praxis

Auch wenn wir uns über die Entwicklung neuer Anwendungen von der "grünen Wiese" weg freuen, so ist doch der Begriff aus Sicht einer Organisation nicht ganz passend. Oft existieren vorhandene oder gewünschte Organisationsstrukturen. Mit zunehmender Dynamik ändern sich stetig Wünsche, Struktur, Kommunikation oder Zuständigkeiten. Das Problem ist somit weniger die heutig bekannte Struktur, sondern mehr die Fähigkeit, auf morgige Anforderungen schnell, kostengünstig und verlässlich zu reagieren.

## Analyse und Entwurf

Mit Microservice-Architekturen sollen Anwendungen entstehen, die dynamische Geschäftsmodelle unterstützen und zukünftige Anforderungen leicht und flexibel technisch unterstützen. Dabei liegt der Schlüssel einer effizienten Anforderungsanalyse im Zerlegen von großen Aufgaben in kleinere Aufgaben. In den meisten Fällen spiegeln Organisationen diese Art der Aufgabenverteilung bzw. Zuständigkeiten. In aller Regel etablieren sich mehr oder minder starre Kommunikationsstrukturen und Prozesse zwischen ihnen.

Oft ist der Blick auf diese Strukturen und Prozesse ein gutes Mittel, das Bild auf die Architektur zu schärfen. Ein erster Entwurf kann sich somit auf die unterschiedlichen Rollen bzw. Abteilungen und deren Begriffe und spezifische Domänen, in [DDD](https://de.wikipedia.org/wiki/Domain-Driven\_Design) auch Bounded Contexts genannt, auseinandersetzen. In der genaueren Analyse einer spezifischen Domäne sollten Sie Zuständigkeiten, Begriffe, Prozesse und die Kommunikation im inneren und nach außen erarbeiten. Nicht selten können Sie innerhalb einer Domäne abermals Teilaufgaben erkennen. Die meist vorhandene, technisch neutrale hierarchische Struktur von Aufgaben, Teilaufgaben und Teil-Teilaufgaben ist meist ein guter Anfang für einen ersten Architekturentwurf.

Um Ihnen den praktischen Einstieg etwas zu erleichtern, haben wir im Folgenden vor, einige Methoden und Praktiken Anhand einer Bespielhaften ECommerce-Anwendung zu erläutern. Unsere Umsetzungen finden Sie auf GitHub (https://github.com/ServiceMashup). Für unsere zukünftige Anwendung ergibt sich nach der ersten Analyse folgendes Organigramm:

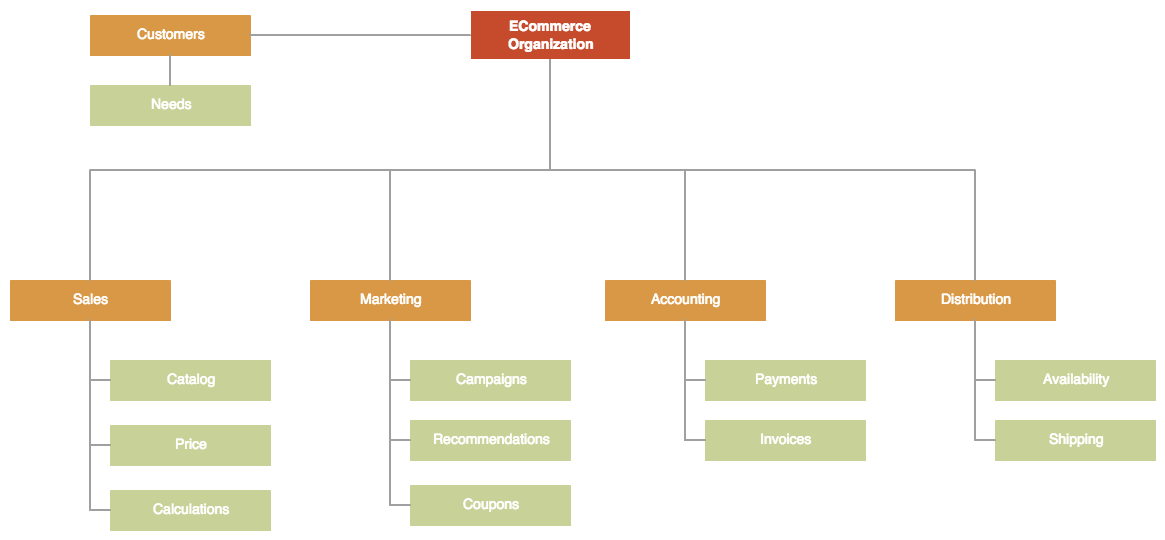


Abbildung 1 ![Organisationsstruktur](images/organisation-structure.png)

Wenn es Ihnen im ersten Anlauf nicht gleich gelingt, einen ersten Entwurf in angemessener Zeit aus der Vogelperspektive zu erarbeiten, kann man sich vorerst auf einen Teilbereich konzentrieren. Ein Entwurf, wie der Begriff bereits prägt, hat keinesfalls den Anspruch auf Vollständigkeit. Vielmehr ist die Erarbeitung innerhalb eines angemessenen Zeitfensters entscheidend. Nacharbeit und möglicherweise verbundenes Refakturieren sollte bewusst in Kauf genommen werden. Schrittweises Erarbeiten und Anpassung durch Erkenntnisse aus der Fachdomäne sollen Sie und Ihre Kollegen vor bewussten oder unbewussten Annahmen, oder schlimmer, deren Umsetzungen, schützen.

## Agile Architektur

Die Implementierung unserer ECommerce-Anwendung lässt bewusst einige Aspekte und gewünschte Funktionen vorerst außen vor. Diese werden in zukünftigen Entwicklungsschritten, nach ersten Nutzungsanalysen und damit besserem Verständnis der Teildomänen, integriert. Somit versuchen wir, die Planung und Implementierung vieler "unreifer Baustellen" mit hohen fachlichen oder technischen Risiken zu vermeiden und uns auf die stabile Realisierung der Kernfunktionen zu konzentrieren.

Architektur: „Strukturen eines Softwaresystems: Softwareteile, die Beziehungen zwischen diesen und die Eigenschaften der Softwareteile und ihrer Beziehungen“ [Paul Clements - Wikipedia](<https://de.wikipedia.org/wiki/Softwarearchitektur>)

Unsere erarbeitete Architektur orientiert sich im ersten Entwurf vor allem an die Benutzeroberflächen (Abbildung 2) und Organisationsstruktur (Abbildung 1).

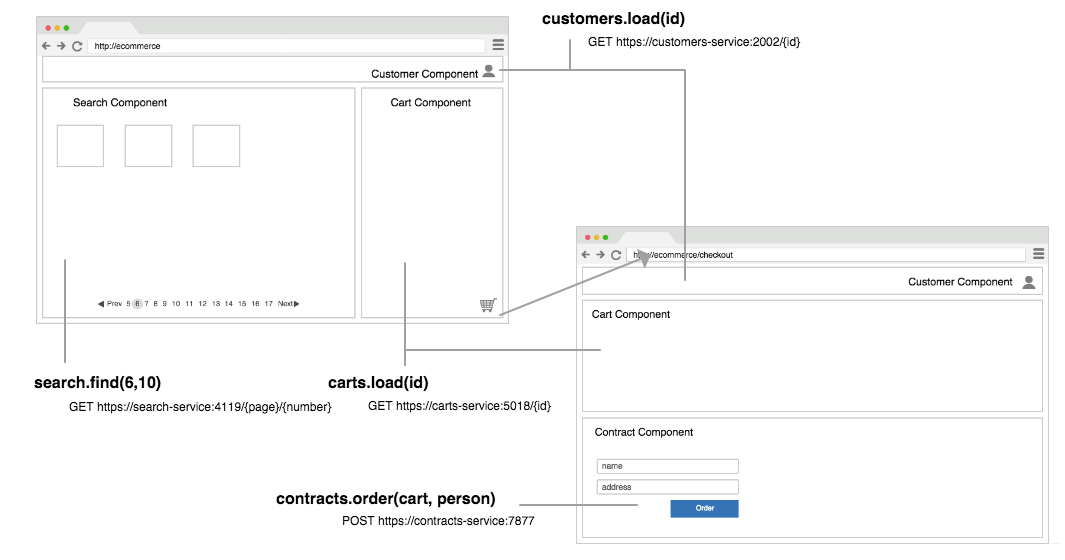


Abbildung 2 ![Composite UI](images/composite-ui.png)

Mit unserem bisherigem Know-How der Fachdomänen und der vorhandenen Ressourcen haben wir uns im ersten Schritt für eine Umsetzung der Kernfunktionen nach (Abbildung 2 und Abbildung 3) innerhalb eines Zeitfensters von 2-3 Wochen entschieden. Die völlig unabhängige und damit parallel stattfindende Entwicklung und Veröffentlichung von Funktionalität bei beinahe störungsfreiem Betrieb waren zusätzliche Gründe, die Gesamtanwendungen in 4 Teilbereiche und innerhalb der Teilbereiche nach Bedarf in weitere kleine spezialisierte Dienste (Microservices) zu teilen.

* Catalog - Artikelverwaltung (Data-Warehouse und Content-Management)
* Search - Aggregation und Indizierung von Artikeldaten
* Cart - Berechnung von Preisen des Warenkorbs
* Contract - Bestellung von Artikeln

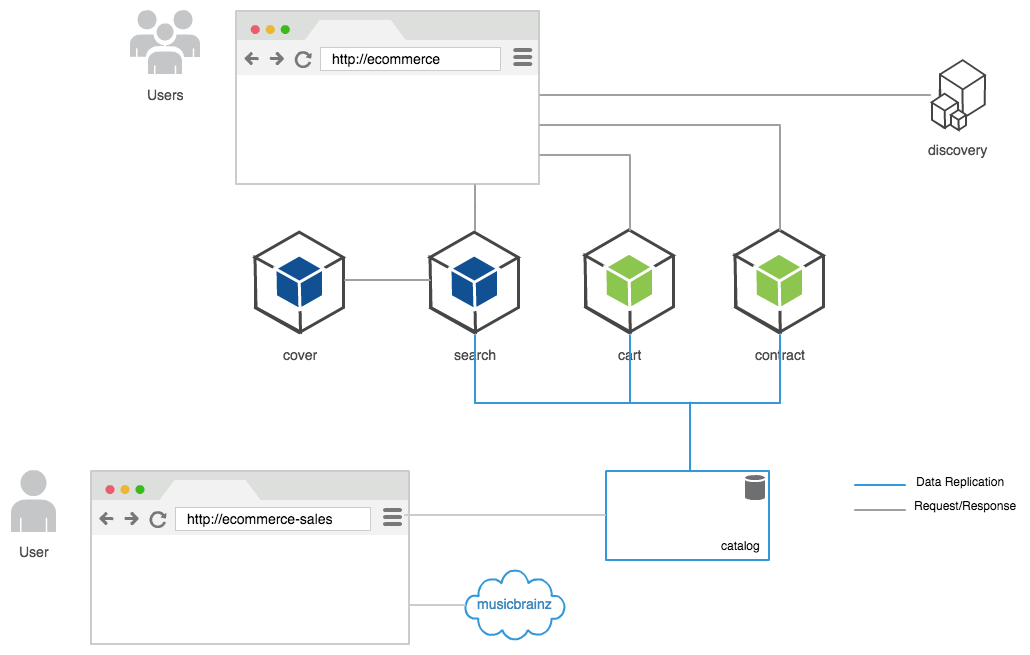


Abbildung 3 ![Architektur](images/architecture.png)

Jeder Microservice, dessen UI-Komponente und deren Integration wird dabei so umgesetzt, dass dieser im Rahmen seines Normalbetriebs völlig unabhängig von anderen Diensten seine spezialisierte Dienstleistung anbieten muss. Um die stetige Aktualisierung und Neustarts von Microservices zu ermöglichen, wird jeder Service im Betrieb durch mindestens 2 Prozesse ausgeführt. Dies verschafft uns eine höhere Ausfallsicherheit sowie Möglichkeiten zur Verteilung von Workload.

Bei der Ausführung mehrerer Instanzen mag jeder inzwischen reflexartig an Load-Balancer oder Cluster denken. Doch diese haben sich im täglichen Betrieb mit mehreren unabhängigen Veröffentlichungen pro Tag meist als zu träge und komplex erwiesen. Zudem erfordern sie oft Spezialwissen, ziehen ungewollte Abhängigkeiten und Seiteneffekte nach sich und bilden letztlich einen "Single Point of Failure".

Wer mehr als einen Prozess eines Dienstes betreiben muss, steht vor besonderen Herausforderungen an Entwicklung und Betrieb. Wir müssen uns mit einigen wichtigen technischen Aspekten verteilter Anwendungen auseinander setzen.

* Garantien bzw. Zusicherungen einer Zustandsverwaltung und Zustandsübertragung
* Fehlertoleranz gegenüber Ausfall und Überlastung
* Reduzierung von Abhängigkeiten durch operative Datenhoheit
* Aggregation von Operativen-, Reporting-, Stamm- und historischen Daten
* Prozesskonfiguration und Veröffentlichung
* Instrumentation und Monitoring
* DevOps und Infrastruktur

Systemanforderungen und Risikomanagement

Sie sollten von Anfang an versuchen, verborgene Annahmen, mögliche inhaltliche oder technische Fehler bzw. Probleme in Entwicklung und Betrieb explizit zu Kommunizieren und als Teil eines zukünftigen Service-Level-Agreement zu betrachten. So, oder so ähnlich, kann eine Teilanforderung aus Anwendersicht an die Gesamtanwendung beschrieben sein:

"Wie unsere Kunden unserer ECommerce-Organisation es gewohnt sind, ist ein 24/7 Betrieb mit 99,9% Gesamtverfügbarkeit gewünscht. Zudem muss eine stetige Weiterentwicklung und Veröffentlichung von Softwareteilen im laufenden Betrieb gewährleistet werden. Priorität, da umsatzrelevant, hat der tatsächliche Kauf (verbindliche Bestellung) von Produkten des Warenkorbs."

Der Kompromiss aus technischem Aufwand (und damit verbundener Kosten) und eigentlichem Nutzen entsteht in der transparenten Kommunikation. D.h. vor der vermeidlich technisch perfekten Lösung, steht das Gespräch „Ziele und Nutzen“ gegen „Aufwände und Kosten“ an. Nach dem [Pareto-Prinzip](https://de.wikipedia.org/wiki/Paretoprinzip) ist ein "für den Anfang gut genug" nicht die perfekte aber weniger aufwändige Entscheidung.

Wer vor sehr komplexen Teildomänen mit vielen fachlichen oder technischen Herausforderungen steht, dem sei hiermit empfohlen, auf ein "klassisches [Monolith First](http://martinfowler.com/bliki/MonolithFirst.html)"-Architekturmodell zurückzugreifen. So können Sie mehr über die Fachdomäne erfahren und mögliche technische Risiken minimiert werden.

## Stabilität und Verfügbarkeit

Oft wird die Kommunikation, sowohl zwischen Frontend und Backend als auch zwischen verschieden Diensten über in Konfigurationsdateien hinterlegte Adressen realisiert. Diese Art der strikten Koppelung führt in der Praxis häufig zu Problemen bei der schrittweisen Weiterentwicklung der Teilsysteme, z.B. wenn eine neue Version eines Dienstes veröffentlicht werden soll, welche nicht mehr abwärtskompatibel ist, während es jedoch noch weitere Dienste im System gibt, die von ersterem abhängen. Ebenso kann hier eine horizontale Skalierung nur durch vorgeschaltete Load-Balancer oder direkt auf DNS-Ebene realisiert werden, was wiederum die Komplexität der zu pflegenden Infrastruktur erhöht. Verzichtet man auf solche Maßnahmen, setzt man gleichzeitig die Verfügbarkeit des Gesamtsystems aufs Spiel: In einem Netz aus voneinander abhängigen Diensten dürfte dann kein einziger dieser Dienste ausfallen.

Microservices verfolgen einen anderen Ansatz: Zum einen sind Dienste im Optimalfall völlig autark und haben so wenig wie möglich (im besten Fall keine) externe Abhängigkeiten, zum anderen sind sie derart gestaltet, dass sie selbst bei einem Ausfall dieser Abhängigkeiten völlig oder zumindest nur leicht eingeschränkt operabel bleiben. Ist der Partner-Dienst dann zu einem späteren Zeitpunkt wieder verfügbar, können noch anstehende Operationen abgeschlossen oder veraltete Daten auf den neuesten Stand gebracht werden. Daraus folgt, dass Kommunikation zwischen Diensten im Regelfall asynchron und nicht im Kontext einer Benutzerinteraktion stattfindet. Daraus folgt ebenfalls, dass die Verfügbarkeit veralteter Daten als immer noch besser zu bewerten ist als die Nicht-Verfügbarkeit jeglicher Daten oder sogar das Auftreten eines Fehlerfalls, im Falle des eCommerce-Beispiels schon rein aus geschäftlichem Interesse: Jeder Auftrag ist kostbar, auch wenn er vielleicht momentan aufgrund eines technischen Ausfalls nicht sofort verarbeitet werden kann. Für Microservice-Architekturen bietet sich eine Reihe von Maßnahmen zur Fehlerkompensation und Ausfallsicherheit an:

## Service Discovery und Fallbacks

Nehmen wir an, Service A möchte seinen eigenen Datenbestand aktualisieren und muss dazu Service B aufrufen. In diesem Fall ist es für Service A im Grunde irrelevant, wie viele Instanzen von Service B laufen, unter welchen Adressen sie erreichbar sind, und welche der laufenden Instanzen nun tatsächlich aufgerufen werden soll. Das einzige Interesse von Service A ist, eine HTTP-Verbindung zu einer intakten Instanz von Service B zu erlangen. Anstatt nun Service A über Konfigurationsdaten mit sämtlichen Zugangsinformationen zu Service B auszustatten, können diese Informationen auch zur Laufzeit ermittelt werden. Dazu benötigt man einen Dienst, der als Registry fungiert und Daten über sämtliche im System laufende Dienste anbietet. Von dieser Registrierungsstelle können nach Bedarf die aktuellen Adressen eines Dienstes anhand seines Namens erfragt werden. Die somit erlangte Liste von Adressen kann für eine bestimmte Zeit im Speicher vorgehalten werden. Möchte man nun eine Verbindung herstellen, probiert man die Adressen systematisch durch. Antwortet die Gegenstelle nicht, so wird versucht, eine Verbindung mit der nächsten Adresse aus der Liste herzustellen (siehe Abbildung A1). Diesen ersten naiven Ansatz kann man dann weiter ausbauen, z.B. durch den Einsatz eines Circuit-Breaker-Pattern, Verteilung von Request via Round-Robin, oder Shard-Verfahrens. Eine beispielhafte Implementierung im Rahmen eines AngularJS-Services ist in Listing A1 dargestellt.

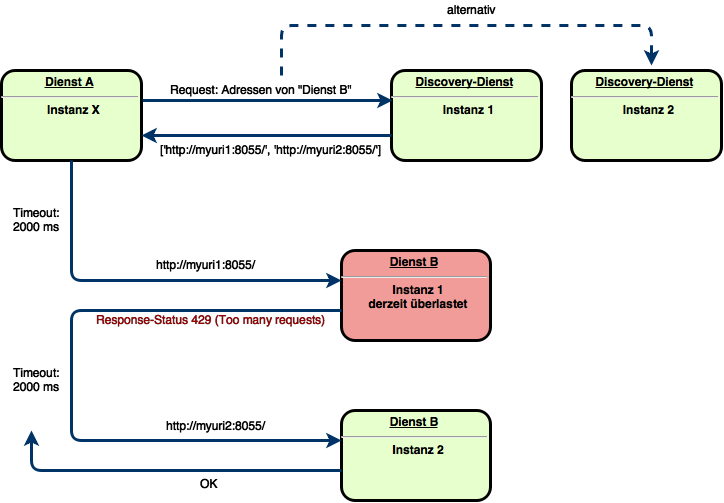


Abbildung 4 ![ServiceDiscovery](images/service-discovery.png)

Listing A1 [Service Discovery in AngularJS](listings/angular-service-discovery.txt)

Die Funktion `ajax` holt zuerst vom Discovery-Service sämtliche verfügbaren URLs für den angegebenen Service-Namen und erzeugt daraus eine Liste von Funktionen, in denen jeweils der HTTP-Aufruf an die entsprechende URL initiiert wird. Die eigentliche Fallback-Logik besteht aus der Verkettung dieser Funktionen, wobei die nächste immer nur dann aufgerufen wird, wenn der vorherige Aufruf zu einem Fehler geführt hat. Dies lässt sich in JavaScript elegant durch die Verkettung von Promises implementieren und geschieht in der Funktion `invokeUntilResolved`. Die Funktion `getServiceUrls` funktioniert nach dem gleichen Muster, schließlich muss es auch vom Discovery-Dienst mehrere Instanzen geben. Zusätzlich werden die Antworten dieser Aufrufe noch in einem lokalen Cache abgelegt, um bei weiteren Anfragen mit demselben Dienstnamen die Liste nicht noch einmal anfordern zu müssen.

Das Auffinden und Auswählen von Services ist eine Querschnittsfunktionalität, die höchstwahrscheinlich unverändert in mehreren Diensten zum Einsatz kommen wird. Aus diesem Grunde bietet es sich an, diese Funktion durch ein importierbares SDK zentral für alle Teams zur Verfügung zu stellen, z.B. als NuGet-Package, NPM-Modul oder Bower-Paket.

## Integration in der UI

Beim Zusammenspiel verteilter Dienste stellt sich immer die Frage nach dem richtigen Schnitt der Funktionalitäten sowie nach dem Management gegenseitiger Abhängigkeiten. Einen Königsweg gibt es wohl nicht, jedoch kann eine Faustregel sein, Systemzustände und dessen Integration über einzelne Dienste hinweg soweit wie möglich an die UI und somit nah an den Nutzer zu verlagern. Auf diese Weise kann ein allzu dichtes Netz an Abhängigkeiten und Zustandssynchronisierung zwischen Diensten vermieden werden, und die Eigenständigkeit von Backend-Diensten wird größtmöglich gewahrt. Zum Beispiel ist es unserer Meinung nach meist keine gute Idee, eine "Web-API" zur Verfügung zu stellen, in der verschiedene Funktionalitäten zusammenlaufen. Es ist hier zu bevorzugen, dass die UI selbst mit den einzelnen Diensten kommuniziert und dann deren Daten aggregiert darstellt. Dieses Vorgehen macht einzelne Komponenten meist viel leichter austauschbar. Es kann aber auch Argumente geben, die gegen eine Orchestration in der Benutzeroberfläche sprechen, z.B. wenn Dienste aus Sicherheitsgründen nicht direkt öffentlich angesprochen werden können. In unserem Beispiel sind der Produktkatalog, der Einkaufswagen sowie die Bestellannahme weitestgehend voneinander getrennt. In der Nutzeroberfläche werden Daten aus verschiedenen Bereichen komponentenorientiert angezeigt (hier z.B. in verschiedenen AngularJS-Direktiven). Kommen im Laufe der Zeit weitere Dienste hinzu (wie z.B. eine Wunschliste oder ein Couponsystem), können diese integriert werden, ohne die Funktionalität vorhandener Services zu beeinträchtigen. Ein weiterer Punkt spricht für die Orchestration in der Nutzeroberfläche: Dadurch, dass hier der eigentliche Kontrollfluss stattfindet, ist es einfacher zu gewährleisten, dass bei Schreiboperationen die Gesamtheit aller Daten an ihrem vorgesehenen Zielort ankommt. Denn die UI kann sämtliche benötigte Daten aggregieren und an den entsprechenden Dienst als Komplettpaket abliefern, sodass dieser sich nicht noch zusätzlich um weitere Daten bemühen muss, was ja wieder eine potenzielle Fehlerquelle wäre.

## Datenreplikation

In den meisten Fällen bestehen Dienste nicht nur aus Logik, sondern brauchen in irgendeiner Form auch Zugriff auf Daten, sei es, dass sie neue Daten in das System einspeisen, oder bereits vorhandene Daten weiterverarbeiten. In klassischen Layer-basierten Architekturen gibt es aus diesem Grund hochverfügbar ausgelegte zentralisierte relationale Datenbanken, um die sich Services wie Planeten in einem Sonnensystem gruppieren. Die Herausforderungen, die sich dabei stellen, sind die Behandlung von Nebenläufigkeit, Skalierung und Ausfallsicherheit. Ist die Datenbank Offline, in Wartung bzw. Update oder durch konkurrierende Anfragen überlastet, "geht nichts mehr". Meist ist die Evolution des Datenschemas eine heikle Angelegenheit, betrifft doch eine Änderung meist auch alle abhängigen Komponenten. Es entstehen unklare Seiteneffekte, was Entwickler meist vor diesem Schritt zurückschrecken lässt. Ein solch starkes Abhängigkeitsgefüge ist Microservice-Architekturen nicht zuträglich, zur Autarkie eines Dienstes gehört hier auch die alleinige Hoheit über sämtliche benötigten Daten. Um diese zu erlangen, ist es sinnvoll, eine lokale Kopie der Daten zu haben, sei es nur im Speicher, im Dateisystem oder gar in einer lokalen Datenbank, die nur der jeweiligen Service-Instanz zugänglich ist. Es handelt sich dabei jedoch um temporäre Daten. Beim Herunterfahren des Dienstes oder der Veröffentlichung einer neuen Version sind diese verloren. Ebenso kann es sein, dass die Kopie der Daten nicht mehr dem aktuellen Stand entspricht, falls sich diese in der Zwischenzeit geändert haben sollten. Dies erfordert ein Umdenken auf Seite der Entwickler verglichen mit dem zentralen Ansatz: Die Behandlung von temporären Inkonsistenzen wird zum elementaren Bestandteil des Gesamtsystems. In unserer ECommerce-Anwendung kommt etwa dem SearchService die Aufgabe zu, auf Anfrage die aktuell angebotenen Produkte herauszugeben. Dazu wird bei jedem Start des Dienstes eine Replikation des Produktkatalogs, hier eine als "Data-Warehouse" dienende CouchDB, im lokalen Speicher erstellt (siehe Listing A2). Gleichzeitig werden diese Produktdaten durch Aufrufe an den CoverService mit Bildinformationen angereichert. Sind die Daten irgendwann zu alt, kann ein erneuter Replikationsdurchlauf angestoßen werden. Alternativ dazu könnte der Service in bestimmten Intervallen eigenständig seinen Bestand aktualisieren oder auch über einen WebHook (<https://de.wikipedia.org/wiki/WebHooks>) über Aktualisierungen einer Datenquelle informiert werden. In Listing A2 wird eine Daten-Replikation in einer anonymen Funktion ausgeführt, welche im Fehlerfall in zeitlichen Abständen automatisch wiederholt wird. Hierzu benutzen wir das NuGet-Package [ReliabilityPatterns](https://www.nuget.org/packages/ReliabilityPatterns/).

Listing A2 [Datenreplikation in .NET](listings/data-replication-dotnet.txt)

Eine etwas andere Strategie verfolgt der CoverService. Dieser dient als Proxy zu einem externen Dienst, von dem URLs zu CD-Cover-Bildern abgefragt werden können. Die Informationen werden hier nicht komplett repliziert, sondern erst auf Anfrage geholt. Von da an wird dann bei Folgeanfragen eine im lokalen Cache vorgehaltene Version des Datensatzes zurückgegeben. Ist der externe Dienst nicht erreichbar oder liefert er keine verwertbaren Daten, so wird hier einfach ein Standardbild als Fallback verwendet, da die Informationen nicht geschäftskritisch sind: Der Benutzer ist immer noch in der Lage, die CD zu bestellen, auch wenn das Coverbild nicht angezeigt werden kann.

## Infrastruktur

Zur effizienten Unterstützung eines agilen und verteilten Entwicklungsprozesses sollten Sie die nötigen Infrastrukturprozesse an allen relevanten Punkten automatisieren. Jedes manuelle Eingreifen benötigt spezielle Kenntnisse und Fähigkeiten die nicht immer zur Verfügung stehen könnten. Ziel muss das Ausliefern beliebiger Komponenten auf Knopfdruck ohne besondere Kenntnisse sein. Um Entwicklung und Betrieb völlige technologische Freiheit zu ermöglichen, hat sich die Verwendung von VM Containern als besonders flexibel erwiesen. Dabei kann jede Anwendungskomponente automatisiert in beliebig vielen unterschiedlichen Version mittels HTTP API jederzeit veröffentlicht, im Service-Discovery registriert und deregistriert und in das Instrumentations- und Monitoring System ausgewertet werden. So ergeben sich drei einfache Schritte für die Veröffentlichung:

* Code und Dokumentation
* Version, Build und Publish
* Konfiguration und Deployment

In unserer Umgebung haben sich für uns folgende Best-Practices und Technologien in der Umsetzung von Microservices in eine automatisierte Infrastruktur ergeben:

* Docker als VM-Container
* Consul.io für automatisches Service-Discovery und Health-Checks
* Shipyard für Continuous Delivery via HTTP-API
* FluentD als Container-Log-Collector
* ElasticSearch-LogStash-Kibana für das Monitoring

Jeder Microservice muss folgende Bedingungen für einen nahtlosen Betrieb erfüllen:

* Log to StdOut
* Konfiguration via Umgebungsvariablen
* HTTP-Kommunikation via Service-Discovery, Timeouts und Fallbacks
* Verwendung temporärer Zustandsspeicher (z.B. In-Memory Cache)
* Health-Check als HTTP Endpunkt
* Graceful-Shutdown (nach Bedarf)
* Request Rate Limits (nach Bedarf)

## Fazit

Microservices ermöglichen kleine selbstorganisierende Entwicklungsteams mit hoher Effizienz in der Weiterentwicklung innerhalb der Teildomäne. Dabei stehen stets Ziele und Nutzen im Vordergrund. Jedes Team hat innerhalb weniger klarer Regeln absolute Freiheit über die zu verwendeten Technologien und dessen Umsetzungen. Darüber hinaus wachsen Fachabteilung und Entwicklung näher zusammen, Entwicklungs-Teams übernehmen selbstständig Verantwortung für die angebotenen speziellen Dienstleistungen und im Ideal werden kontinuierlich innovative Anwendungen effizient umgesetzt.

Um Microservice-Architekturen effizient in neuen oder bestehenden Anwendungen umzusetzen, ist ein Umdenken gegenüber traditionellen technischen Konzepten erforderlich:

* Aggregate statt Relationen
* Replikation statt Normalisierung
* horizontale statte vertikale Skalierung
* Asynchronität statt Transaktionen
* Distribution statt Zentralisierung

Versuchen Sie von Anfang an ein agiles Vorgehensmodell zu verfolgen. Voraussetzung dazu ist die Umsetzung von Continuous Integration und Continuous Delivery samt der dazu nötigen Infrastruktur.

Sollten Sie eine neue Anwendung mit Microservices umsetzen wollen, beginnen Sie den Teil mit dem höchsten Kundennutzen. Wenn Sie jedoch vorhaben Ihre bestehende Anwendungsstruktur in Richtung Microservices auszubauen, sollten Sie nicht gleich mit dem geschäftskritischen Bereich beginnen, da diese den höchsten fachlichen Anforderungen unterliegen. Suchen sie sich eher einen Teilbereich bei dem Sie sich auf die neuen technischen Herausforderungen fokussieren und Erfahrungen sammeln können wie Microservices in Ihrer Organisation optimal eingesetzt werden können.