# Lean & Clean – Microservice in der Praxis

## Agilität

Auch wenn wir uns über die Entwicklung neuer Anwendungen von der "grünen Wiese" weg freuen, so wenig ist der Begriff aus Sicht einer Organisation passend. Oft existieren vorhandene oder gewünschte Organisationsstrukturen. Mit zunehmender von innen oder außen gewünschter Dynamik ändern sich stetig Wünsche, Struktur, Kommunikation oder Zuständigkeiten. Das Problem ist somit weniger die heutig bekannte Struktur, sondern 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 Anforderungen von morgen 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 die vorhandene oder gewünschte Organisationstruktur, dessen Kommunikation und Prozesse ein gutes Mittel das Bild auf die Architektur zu schärfen. Ein erster Entwurf sollte sich somit auf die unterschiedlichen Rollen bzw. Abteilungen und dessen spezifische Domäne (DDD auch Bounded Context genannt) und Begriffe auseinandersetzen. In der genaueren Analyse einer spezifischen Domäne sollten Sie Zuständigkeiten, Begriffe, Prozesse und die Kommunikation im innerin und nach außen erarbeiten. Nicht sellten 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.

An dieser Stelle sei dem interessierten Leser wärmstens Techniken und Praktiken aus [DDD](https://de.wikipedia.org/wiki/Domain-Driven\_Design) empfohlen.

Für unsere zukünftige Anwendung ergibt sich nach einer ersten Analyse folgendes Organigramm.

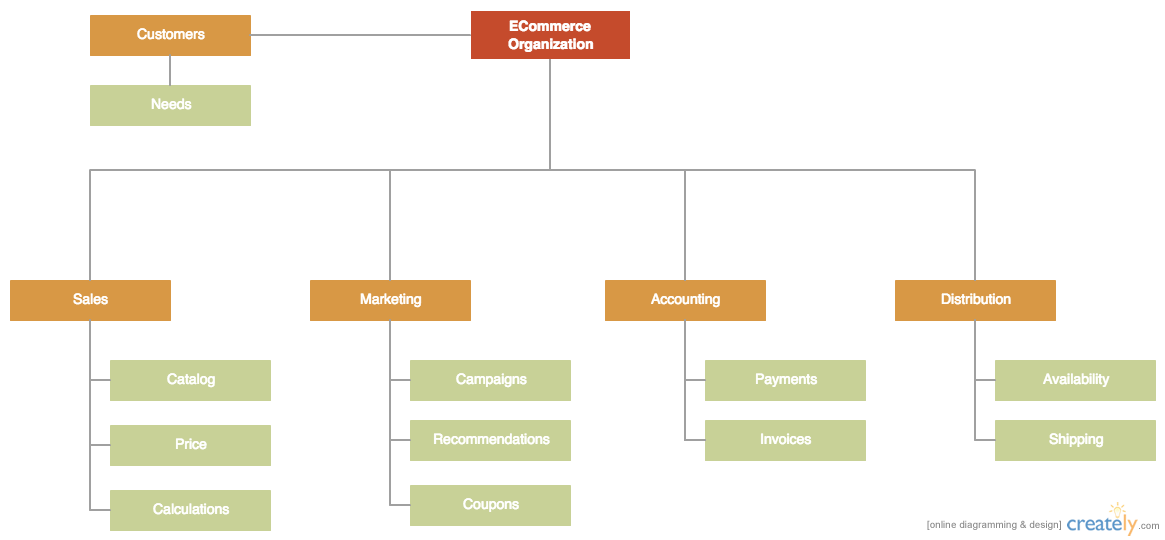


Abbildung 1 ![Organisationsstruktur](images/organisation-structure.png)

Wem es nicht gleich gelingt einen ersten Entwurf in angemessener Zeit aus der Vogelperspektive zu erarbeiten, kann sich 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 Refaktorieren sollte bewusst in Kauf genommen werden. Schrittweises Erarbeiten und Anpassung durch Erkenntnissen aus der Fachdomäne soll Sie und Ihre Kollegen vor bewussten oder unbewussten Annahmen, oder schlimmer Umsetzungen, schützen.

## Umsetzung Architektur

Die Umsetzung unserer ECommerce-Anwendung lässt bewusst einige Aspekte und gewünschte Funktionen in der tatsächlichen Umsetzung vorerst außen vor. Diese werden in zukünftigen Entwicklungsschritten, nach ersten Nutzungsanalysen und damit besserem Verständnis der Teildomänen, integriert. Vermeiden Sie die Planung und Umsetzung vieler "unreifer Baustellen" mit hohen fachlichen oder technischen Risiken und konzentrieren Sie sich auf die stabile Umsetzung der Kernfunktionen.

Laut Wikipedia - „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) lehnt sich unsere erabeitete Architektur im ersten Entwurf vor allem an die Organisationsstruktur (Abbildung 1) und Benutzeroberfläche (Abbildung 2) an.

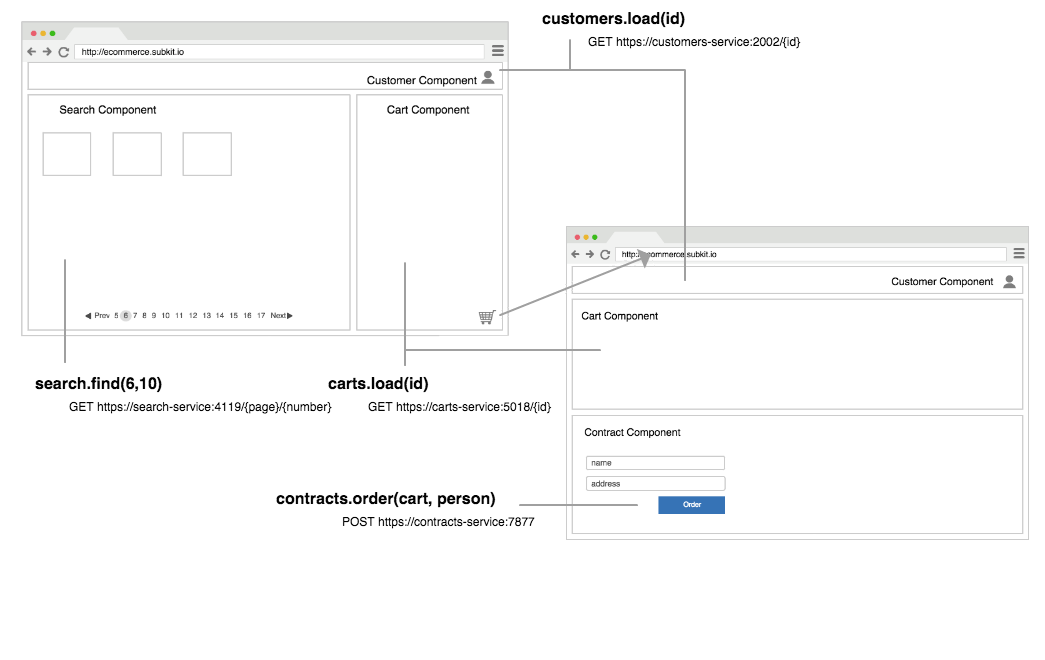


Abbildung 2 ![Composite UI](images/composite-ui.png)

Mit unserem bisherigem Know-How der Fachdomänen und der vorhandenen Resourcen 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 parallele stattfindene Entwicklung und Veröffentlichung von Funktionalität bei beinahe störungsfreiem Betrieb waren zusätzliche Gründe, die Gesamtanwendungen in 4 Teilbereiche und innerhalb der Teilbereiche in kleine spezialisierte Dienste (Microservices) zu teilen.

\* Catalog - Artikelverwaltung (CMS)

\* Search - Aggregation und Indezierung von Produktdaten

\* Cart - Berechnung von Preisen

\* Contract - Bestellung von Artikeln

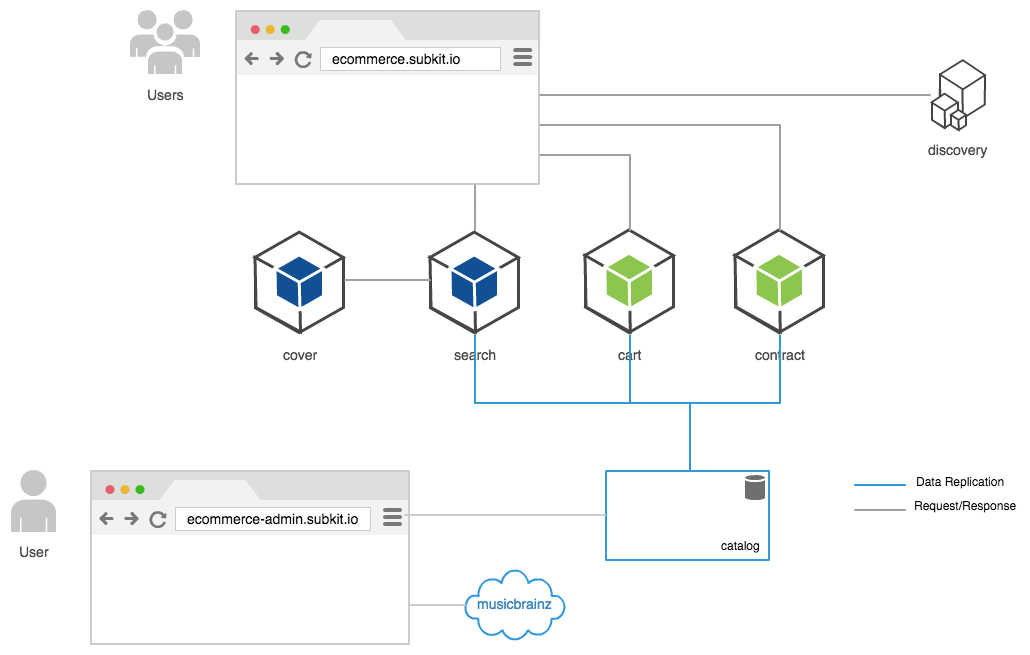


Abbildung 3 ![Architektur](images/architecture.png)

Jeder Microservice, dessen UI-Komponente und deren Integration wurde 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 Microservice im Betrieb durch mindestens 2 Prozesse ausgeführt.

Um eine höhere Ausfallsicherheit oder die Verteilung von Workload zu gewährleisten, können mehrere Prozesse eines Microservice gestartet werden. An dieser Stelle mag jeder inzwischen reflexartig an Load-Balancer oder Cluster denken. Doch diese haben sind im täglichen Betrieb mit mehreren unabhängigen Veröffentlichungen pro Tag meist als zu träge und komplex erwiesen. Zudem erfordern diese meist Spezialwissen, ziehen ungewollte Abhändigkeiten und Seiteneffekte nach sich und bilden letztlich einen "Single Point of Failure".

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

\* Garantien bzw. Zusicherungen einer Zustandsverwaltung und Zustandsübertragung [Data Replication]()

\* Fehlertoleranz gegenüber Ausfall und Überlastung [Service Discovery, Graceful-Shutdown, Rate-Limits]()

\* Reduzierung von Abhängigkeiten durch operative Datenhoheit [Atom Ops, Cache, Webhooks]()

\* Aggegration von Operativen-, Reporting-, Stamm- und historischen Daten [Admins]()

\* Prozesskonfiguration [ENV-Vars]()

\* Instrumentation und Monitoring [Logging]()

\* DevOp und Infrastruktur [Infrastruktur](Infrastructure.md)

Anforderungen und Risikomanagement

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."

Entscheidend ist, viele sonst vorborgende Annahmen, mögliche inhaltliche oder technische Fehler bzw. Probleme in Entwicklung und Betrieb von Anfang an explizit zu machen. Der Kompromiss aus technischem Aufwand und damit verbundene Kosten und der eigentlich 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 komplexe Entscheidung.

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

## Stabilität und Verfügbarkeit

In traditionellen serviceorientierten Architekturen wird die Kommunikation, sowohl zwischen Frontend und Backend als auch zwischen verschieden Diensten, meist über in Konfigurationsdateien hinterlegten Adressen realisiert. Diese enge Koppelung führt in der Praxis häufig zu Problemen bei der Weiterentwicklung der Systeme, 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 von ihrem Wesen her 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. Es folgt auch daraus, 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, wieviele Instanzen von Service B laufen, unter welchen Addressen 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 Breakers oder des Round-Robin-Verfahrens. Eine beispielhafte Implementierung im Rahmen eines AngularJS-Services ist in Listing A1 dargestellt.

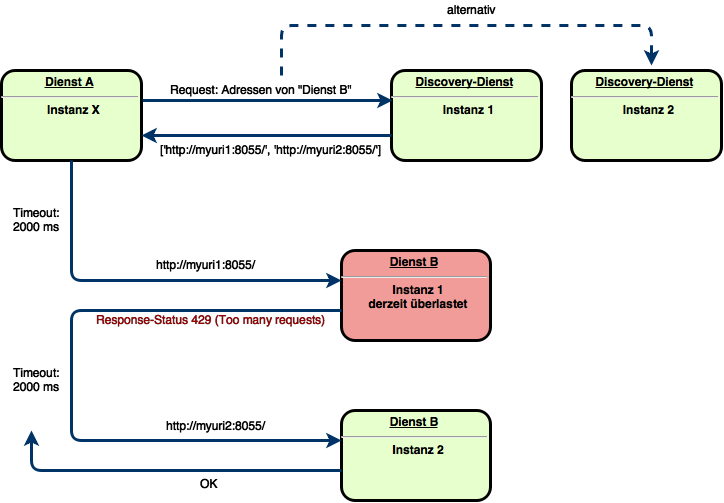


Abbildung 4 ![ServiceDiscovery](images/service-discovery.png)

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.

### Orchestration

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, die Integration soweit wie möglich nach "vorne" zu verlagern, im Idealfall direkt in die UI. Auf diese Weise kann ein allzu dichtes Netz an Abhängigkeiten 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 Orchastration 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 Orchstration 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 Architekturen gibt aus diesem Grund eine oder mehrere meist 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, "geht nichts mehr". Ebenso ist die Evolution des Datenschemas eine heikle Angelegenheit, betrifft doch eine Änderung meist auch alle abhängigen Komponenten, 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 sinvoll, eine lokale Kopie der Daten zu haben, sei es nur im Speicher, 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 der "Master-Daten" 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 Inkonsistenzen wird zum elementaren Bestandteil der Programmlogik. In unserem CD-Shop-Beispiel 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. In Listing A2 wird zudem die gesamte 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 [Data Replikation 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 Entwicklungsprozess sollten Sie die nötige Infrastruktur an allen Stellen 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 Entwicklern und Operations 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 deregistriet und in das Instrumentations- und Monitoring System ausgewertet werden. So ergeben sich 3 einfache Schritte für die Veröffentlichung.

\* Code und Dokumentation

\* Version, Build und Publish

\* Konfiguration und Deployment

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

\* Log to StdOut

\* Konfiguration via Env-Var

\* Graceful-Shutdown

\* Request Rate Limits

\* Verwendung temporärer Zustandsspeicher (z.B. In-Memory Cache)

\* Health-Check als HTTP Endpunkt