10 Serviceorientierte Architekturen (SOA) und Microservices

Mit kräftiger Unterstützung von Alex Heusingfeld und Stefan Tilkov

SOA is a Lifestyle.

Ann Thomas Manes, in [SOAX 07]

Fragen, die dieses Kapitel beantwortet:

- Was ist SOA und warum gibt es sie?
- Wie funktionieren Services?
- Wie wird SOA in der Praxis umgesetzt?
- Was sind Microservices?
- Wie unterscheiden sich Microservices von klassischer SOA?
- Welche Rolle spielen SOA und Microservices für Softwarearchitekten?

Der Begriff „Microservices“ bezeichnet einen Architekturstil, dessen Ziel die Erhöhung der Wandelbarkeit und Flexibilität von Software durch Modularisierung in der Entwicklung und im Betrieb ist. Dabei werden bestehende Konzepte wie beispielsweise *Continuous Delivery* mit neuen Ansätzen kombiniert. Das Konzept adressiert häufige Probleme großer Systeme, bringt aber auch neue Herausforderungen mit sich. Da dieser Ansatz vor allem bei Internetunternehmen zum Einsatz kommt, stellt sich die Frage, ob er auch für eher klassische Unternehmen tragen kann.

Anstatt die gesamte Funktionalität in einem einzelnen System zu implementieren, wird sie in fachlich zusammengehörende Gruppen aufgeteilt und diese als eine Menge von kleineren Systemen, sogenannten Microservices, entwickelt. Jeder Microservice hat dabei seine eigene Ablaufumgebung, ist also unabhängig von anderen Microservices, und kommuniziert mit diesen nur über das Netzwerk.

Die Fokussierung auf die Fachlichkeit und den „Service“-Begriff erweckt den Eindruck, dass der Microservice-Ansatz mit den Konzepten von SOA verwandt ist. Und wirklich wurden einige der Ideen hinter Microservices auch in der SOA-Community diskutiert, weshalb wir uns in diesem Kapitel zunächst die Konzepte von serviceorientierten Architekturen genauer ansehen wollen, um die Unterschiede zwischen SOA und dem Microservice-Architekturstil besser verstehen zu können.

10.1 Was ist SOA?

Warum gibt es SOA?

SOA ist ein Business-Thema

Als Motivation für SOA skizzieren wir die Situation vieler Unternehmen hinsichtlich (externer) Einflussfaktoren und (interner) Informationsverarbeitung: Unternehmen existieren, um Mehrwert zu erwirtschaften, und zwar durch wertschöpfende Geschäftsprozesse. Dabei wirken im Wesentlichen vier Einflussfaktoren auf die Unternehmen und deren Geschäftsprozesse:

1. Die Ansprüche von Kunden und Partnern steigen: Der Druck globaler Märkte auf Unternehmen nimmt zu. Um weiterhin profitabel zu bleiben, müssen Unternehmen ihre Geschäftsprozesse häufig an wechselnde Einflüsse von Märkten und Kunden anpassen. Flexibilisierung und Modularisierung von Geschäftsprozessen wird zur Grundlage unternehmerischen Handelns.
2. Viele Geschäftsprozesse funktionieren ausschließlich mit IT-Unterstützung. Ohne IT sind viele Unternehmen nicht handlungs- beziehungsweise lebensfähig. Denken Sie beispielsweise an die gesamte Finanz- und Telekommunikationsbranche. Leider hemmt die IT oftmals die (so dringend benötigte) Flexibilität und Agilität von Unternehmen, weil sie sich nicht schnell oder flexibel genug an geänderte Prozesse adaptieren kann.
3. Bestehende IT-Systeme sind häufig Applikationsmonolithen, unflexible und über lange Jahre gewachsene starre Anwendungssilos. Die zugehörigen IT-Organisationen sind als Fürstentümer organisiert und orientieren sich weniger an der Wertschöpfung des Gesamtunternehmens als an eigener Macht und Einfluss.
4. IT-Budgets werden oftmals für die Entwicklung oder Modernisierung einzelner Anwendungen vergeben, ohne direkten Bezug zu deren aktuellen oder künftigen Wertschöpfung. Dadurch entstehen die sogenannten „Millionengräber“. Budgets sollten sich an der Wertschöpfung der IT-unterstützten Dienste und Geschäftsprozesse orientieren.

Für Unternehmen folgt daraus die Notwendigkeit, die Abstimmung zwischen Geschäft (*Business*) und IT zu verbessern: In Zukunft muss die IT hochflexible Geschäftsprozesse effektiv und schnell unterstützen. Dazu bedarf es der Abkehr von Anwendungsmonolithen und der Schaffung flexibler Softwareeinheiten – eben Services.

Genau deshalb (und nur deshalb!) brauchen Unternehmen SOA: Es geht um unternehmerische Flexibilität, um die Fähigkeit, als Unternehmen auf veränderliche Marktbedingungen schnell mit wertschöpfenden Geschäftsprozessen antworten zu können. SOA wird ausschließlich durch diese geschäftliche Argumentation getrieben – Technologie ist nur Mittel zum Zweck.

Diese beiden Perspektiven von SOA (geschäftlich/organisatorisch und technisch) führen unserer Erfahrung nach häufig zu Missverständnissen bei der Kommunikation zwischen den beteiligten Personen über die Bedeutung fachlicher Services und deren Umsetzung in technischen Diensten. Unserer Meinung nach gehören zu SOA unbedingt beide Perspektiven – keine kann alleine funktionieren.

Tabelle 10.1 SOA-Definitionen aus verschiedenen Perspektiven

Perspektive (Stakeholder)	SOA ist ...
Manager	eine Menge von IT-Assets („Fähigkeiten"), aus denen wir Lösungen für unsere Kunden und Partner aufbauen können.
Enterprise-IT-Architekt	Architekturprinzipien und -muster für übergreifende Modularisierung, Kapselung, lose Kopplung, Wiederverwendbarkeit, Komponierbarkeit, Trennung von Verantwortlichkeiten.
Projektleiter	ein Ansatz für parallele Entwicklung mehrerer Projekte oder Aufgaben.
Entwickler, Programmierer	ein Programmiermodell, basierend auf Standards wie Web-Services, REST, XML, HTTP.
Administrator, Operator	ein Betriebsmodell für Software, bei dem wir nur noch Teile unserer Anwendungen (Services) selbst betreiben und viele Services von Partnern nutzen.
Strategieberater, Softwarehersteller	ein unklar definiertes Hype-Thema, das in den Ohren unserer Kunden vielversprechend klingt. Ein gigantischer Beratungs- und Werkzeugmarkt voller Chancen und Risiken.

SOA muss in Unternehmen übergreifend wirken, sowohl auf Geschäftsprozesse/Organisation als auch auf IT. Daher haben sehr unterschiedliche Beteiligte mit SOA zu tun und jeweils ihre spezifische Auffassung von „Was ist SOA?". Einige Beispiele (in Anlehnung an [Lublinsky 07]) finden Sie in Tabelle 10.1.

Verstehen Sie SOA als Synthese dieser Meinungen:

- Eine serviceorientierte Architektur (SOA) ist eine unternehmensweite IT-Architektur mit Services (Diensten) als zentralem Konzept.
- Services realisieren oder unterstützen Geschäftsfunktionen.
- Services sind lose gekoppelt.
- Services sind selbstständig betriebene, verwaltete und gepflegte Softwareeinheiten, die ihre Funktion über eine implementierungsunabhängige Schnittstelle kapseln.
- Zu jeder Schnittstelle gibt es einen Service-Vertrag, der die funktionalen und nichtfunktionalen Merkmale (Metadaten) der Schnittstelle beschreibt. Zu jeder Schnittstelle kann es mehrere, voneinander unabhängige, Implementierungen geben.
- Die Nutzung von Services geschieht über (entfernte) Aufrufe („Remote Invocation").
- SOA setzt so weit wie möglich auf offene Standards, wie XML, SOAP[1], WSDL und andere.
- Services sind bereits von ihrem Entwurf her *integrierbar*.[2]

In den folgenden Abschnitten wollen wir diese Definition etwas erläutern. Zur Einordnung stellt Bild 10.1 den Begriff „Service" in den Kontext von Unternehmen, Geschäftsprozess und Implementierung.

[1] SOAP: *http://www.w3.org/TR/SOAP*, WSDL: *http://www.w3.org/TR/WSDL*

[2] Dieses Versprechen ist in der Praxis oft schwer einzulösen, weil Service-Anbieter und Service-Nutzer sich nur schwer auf eine gemeinsame Semantik von Daten und Dokumenten einigen können.

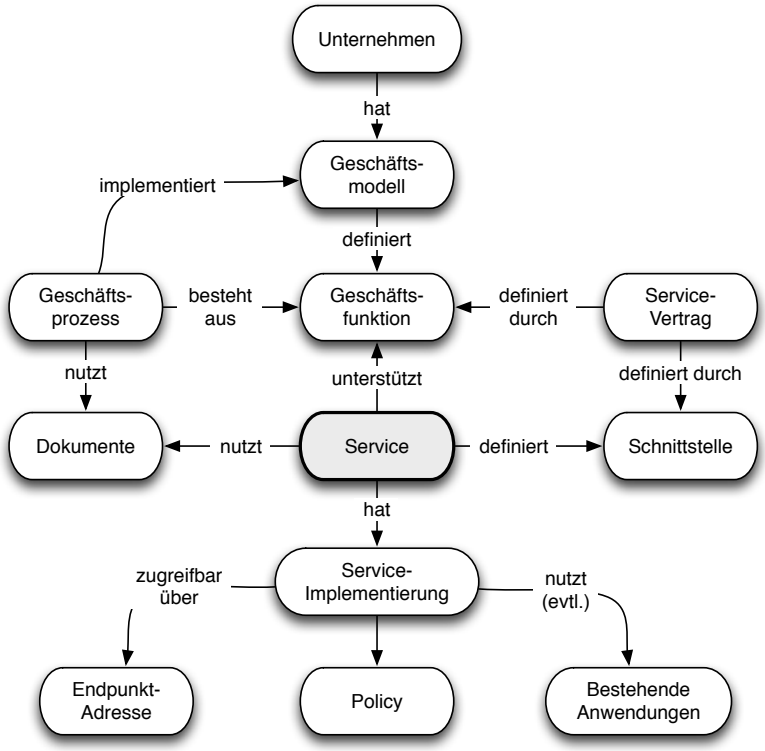

Bild 10.1 Services im Kontext von Unternehmen und Implementierung

Services als zentrales Konzept

In SOA übernehmen Services in hohem Maße die Rolle bisheriger „Anwendungen“: Unternehmen planen, entwerfen, entwickeln, testen und betreiben Services, nicht Anwendungen. Outsourcing erfolgt auf Basis von Services, nicht von Hardware oder Anwendungen. Abteilungen verantworten Services, keine Anwendungen.

Services realisieren Geschäftsfunktionen

Services realisieren Geschäftsfunktionen, die für ihre Nutzer einen geschäftlichen Mehrwert darstellen. Unter solchen Geschäftsfunktionen verstehen wir Aktivitäten, die ein Unternehmen direkt oder indirekt in seiner Wertschöpfung unterstützen – sei es bei der Fertigung eines Produkts oder dem Erbringen einer Dienstleistung. Beispiele für Services sind Bestellannahme, Kostenkalkulation, Angebotserstellung, Produktionsplanung, Vertragsmanagement, Kundenverwaltung, Abrechnung, Mahnwesen.

Warum beschränken wir Services auf Geschäftsfunktionen? Es gibt doch auch technische Dienste, beispielsweise „Datensicherung“. Solche technischen Dienste können sinnvolle und wichtige Funktionen erbringen, tragen jedoch nicht direkt zur Wertschöpfung eines Unternehmens bei. Eine aus wirtschaftlicher Sicht sinnvolle SOA hat geschäftliche Flexibilität (business agility) zum Ziel und orientiert sich an wertschöpfenden Services.

Lose Kopplung

Der Begriff „Kopplung“ bezeichnet den Grad der Abhängigkeiten zwischen zwei oder mehr „Dingen“. Je mehr Abhängigkeiten zwischen Services bestehen, desto enger sind sie aneinandergekoppelt. Kopplung bezieht sich dabei auf alle Arten der Abhängigkeiten von Services, beispielsweise:

- Zeitliche Abhängigkeit: Ein Service kommuniziert synchron mit einem anderen, d. h., beide Services müssen zeitgleich in Betrieb sein.
- Örtliche Abhängigkeit: Ein Service ruft einen anderen Service unter einer bestimmten Adresse auf – der aufgerufene Service darf diese Adresse nicht ändern.
- Struktur- oder Implementierungsabhängigkeit: Die Implementierung eines Service verwendet die Implementierung eines anderen Service direkt, anstatt über die „offizielle“ Schnittstelle zu kommunizieren. Somit ist die Implementierung des aufgerufenen Service nicht mehr austauschbar.
- Datenabhängigkeit: Ein Service nutzt eine (interne) Datenstruktur eines anderen Service, interpretiert beispielsweise die ersten acht Stellen eines Datenbankschlüssels als Artikelnummer. Somit kann die interne Repräsentation dieser Daten nicht mehr geändert werden.

Lose Kopplung von Services bringt eine hohe Unabhängigkeit und damit Flexibilität einzelner Services mit sich. Services in einer SOA sollten lose gekoppelt sein; mit anderen Worten: nur so eng miteinander verflochten, wie unbedingt notwendig.

Das Idealziel – lose Kopplung in allen Dimensionen – ist dabei weder immer erreichbar noch immer wünschenswert. Zeitliche Entkopplung beispielsweise erfordert asynchrone Kommunikation und die ist schwieriger zu implementieren als synchrone.

Einen weiteren Aspekt möchten wir hervorheben, nämlich das dynamische Binden: Ein Service-Nutzer („Service-Consumer“ oder „Dienstnutzer“) ermittelt die Adresse seines Service-Anbieters (auch „Service-Provider“ oder „Dienstanbieter“ genannt) in der Regel erst zur Laufzeit. Hierbei unterstützen ihn spezialisierte Verzeichnisdienste („Registries“).

Schnittstelle und Implementierung

Jeder Service stellt für seine Nutzer eine fest definierte Schnittstelle bereit. Intern, das heißt nach außen unsichtbar, besitzt jeder Service eine Implementierung, die letztendlich für die eigentliche Arbeit des Service verantwortlich zeichnet. Die Implementierung eines Service besteht wiederum aus Geschäftslogik und zugehörigen Daten.

Services bieten ihre Dienste ausschließlich über diese Schnittstellen an. Service-Nutzer sollen und dürfen keinerlei Annahmen über das Innenleben (die Implementierung) von genutzten Services treffen. Dieses Geheimnisprinzip stellt die Unabhängigkeit und lose Kopplung einzelner Services sicher.

Serviceorientierte Architekturen setzen dieses Konzept durchgängig ein: Sämtliche Services (sowohl Geschäfts- als auch technische Services) bieten klar definierte Schnittstellen an, für die es eine oder mehrere Implementierungen geben kann. Die Nutzer eines Service besitzen ausschließlich eine Abhängigkeit zur Schnittstelle des genutzten Service.

Service-Vertrag und Metadaten

In einer SOA gibt es eine Vielzahl von Metadaten, d. h. Daten über Daten. Dazu zählen funktionale und nichtfunktionale Aspekte, beispielsweise

- Beschreibungen der Service-Schnittstellen, der möglichen Operationen und der an ihnen ausgetauschten Informationen;
- Sicherheitsanforderungen und Berechtigungen;
- Performance-Anforderungen;
- organisatorische Zuordnungen;
- die Adresse, unter der ein Service aufrufbar ist (auch „Endpunkt-Adresse“ genannt).

Es gibt zwei Arten solcher Metadaten: funktionale und nichtfunktionale. Zu den funktionalen Metadaten zählen die Schnittstellendefinitionen und die Definition der ausgetauschten Informationsobjekte. Nichtfunktionale Metadaten beschreiben Aspekte wie Sicherheitsrestriktionen, Transaktionseigenschaften oder den Zuverlässigkeitsgrad der Nachrichtenzustellung. Nichtfunktionale Eigenschaften eines Service heißen auch Policies. Schnittstelle und Policy zusammen bilden den Service-Vertrag.

Ein Service kann durchaus mit identischer Schnittstelle, aber mit verschiedenen Policies (d. h. in unterschiedlichen nichtfunktionalen Ausprägungen) angeboten werden. Betrachten Sie als Beispiel die Kursabfrage von Wertpapieren:

- Eine Policy (die kostenfreie „Community Edition“ dieses Service) liefert die um 15 Minuten gegenüber der Börse verzögerten Kursdaten.
- Eine zweite Policy (die kostenpflichtige „Premium Edition“) liefert über die identische Schnittstelle die Kursdaten in Echtzeit.

(Selbstbeschreibende) Dokumente

Service-Aufrufe übergeben ihre Daten in Form von *Dokumenten*, also strukturierten *Datenbehältern*. Meist verwenden Services dazu XML-Dokumente, deren Aufbau durch spezifische XML-Schemata vom Service-Provider vorgegeben wird. Selbstbeschreibende Formate schützen dabei vor Fehlern, sowohl bei der Interpretation der Dokumente als auch bei fachlichen Erweiterungen. Vergleichen Sie beispielsweise den folgenden Methodenaufruf

```
createInvoice( 20070905, 42, 84, "K-SQ-42")
```

mit dem Dokument(fragment):

```
<invoice>
         <date>2007-09-05</date>
         <productid>42</productid>
         <quantity>84</quantity>
         <customerid>K-SQ-42</customerid>
</invoice>
```

Zum Verständnis des Methodenaufrufs müssen Sie die Signatur dieser Methode kennen, da der Methodenaufruf nicht selbstbeschreibend ist. Die Übergabe zusätzlicher Informationen (beispielsweise Adressdaten zur Rechnung) erfordert Codeänderungen in der betroffenen Klasse.

Sie können hingegen dem Dokument durchaus einige neue Daten hinzufügen, ohne dass die betroffenen Services geändert werden müssen: Ein Service-Consumer kann, beispielsweise mit XPath, aus dem Dokument genau die für ihn notwendigen Informationen extrahieren. Zusätzliche Daten stören nicht.

Falls Sie jetzt an Performance denken: korrekt – die in selbstbeschreibenden Dokumenten enthaltene Informationsredundanz kostet Laufzeit, verursacht durch Parsing-Aufwände und zusätzlichen Datentransfer. Andererseits gewinnen Sie eine erhebliche Flexibilität, der in SOA große Bedeutung zukommt.

10.2 So funktionieren Services

In diesem Abschnitt wollen wir erklären, wie Services einer SOA technisch funktionieren. Hierbei soll unterstrichen werden, warum die Unterscheidung zwischen Service-Anbieter (Provider) und Service-Nutzer (Consumer) wichtig ist.

Die Zusammenarbeit zwischen Consumer und Provider benötigt eine ganze Menge Vorbereitung:

- Der Provider definiert seine funktionale Schnittstelle in einer maschinenlesbaren Form (etwa: WSDL, Web-Service Description Language). Diese Schnittstelle wird Bestandteil des Service-Vertrags.
- Ein Provider implementiert diese Schnittstelle und fügt die nichtfunktionalen Eigenschaften dieser Implementierung (ihre „Policies“) als Metadaten dem Service-Vertrag hinzu.
- Der Provider stellt die Implementierung des Service in einer Laufzeitumgebung zur Verfügung. Dazu legt er die Adresse (Endpunkt-Adresse) fest.
- Schließlich publiziert der Provider all diese Informationen in einem Service-Verzeichnis. Nun können Service-Consumer auf diese Informationen zugreifen.
- Ein Service-Consumer sucht (und findet) im Service-Verzeichnis die Definition der Provider-Schnittstelle.
- Unter Nutzung der Provider-Schnittstelle entwickelt der Consumer einen eigenen Service, der den Provider „verwendet“, d. h. zur Laufzeit bestimmte Operationen der Provider-Schnittstelle aufruft.

Dieses Zusammenspiel zeigt Bild 10.2.

Es gibt viele technische Varianten, mit denen Sie diese Art der Zusammenarbeit implementieren können. Einige davon (aber nicht die einzigen!) sind Web-Services, REST, CORBA oder nachrichtenbasierte Systeme.

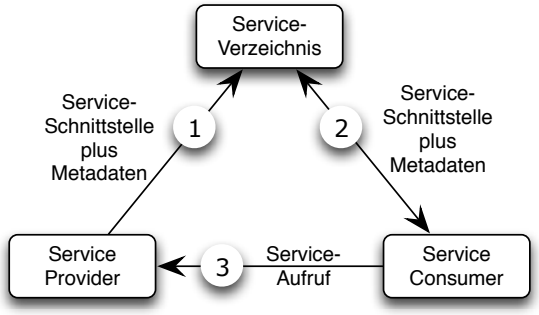

Bild 10.2
Das Service-Dreieck: Consumer, Provider, Verzeichnis

10.3 SOA in der Praxis

Wenn man den SOA-Begriff verwendet, sollte man klar zwischen den Ideen und Konzepten, die beispielsweise im SOA-Manifesto beschrieben werden, und der „SOA-Realität" unterscheiden. Services bilden den Kern von SOA – jedoch gehören noch einige weitere Bestandteile und Konzepte dazu, ohne die im wahrsten Sinne „nichts läuft". Einige möchten wir Ihnen nachfolgend vorstellen.

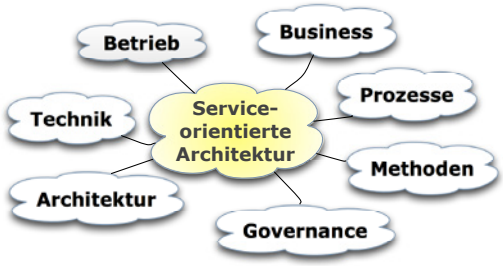

Bild 10.3
Was gehört noch zu SOA?

Architekturstile

Sie können SOA mit unterschiedlichen Architekturstilen aufbauen:

- schnittstellenorientiert: Ähnlich dem konventionellen Remote Procedure Call (RPC) definieren Sie für sämtliche Funktionen Ihrer Services eigene Operationen. Meist folgen diese Operationen dem (synchronen) Kommunikationsmuster *Request-Response*, gelegentlich sogar mit dem Verzicht auf Rückgabewerte (*one-way operations*). Die Operationen erhalten in ihren Signaturen Parameter.
- nachrichtenorientiert: Bei einer nachrichtenorientierten SOA stehen die ausgetauschten Informationen im Vordergrund. In der Regel spricht man hier von Dokumenten, die durch Nachrichten übertragen werden – anders formuliert: Nachrichten enthalten als Nutzdaten (*Payload*) ein Dokument und zusätzlich Adressinformationen und andere Metadaten.

Dieser Architekturstil kommt häufig bei asynchron gekoppelten Systemen zum Einsatz, im Zusammenhang mit entsprechender Kommunikationsinfrastruktur (*message-oriented middleware*, MOM).[3]

- ressourcenorientiert: Eine Alternative für Service-Orientierung im Stil des Web ist REST. Services stellen hierbei sog. Ressourcen zur Verfügung, die von anderen Services konsumiert und ggf. verändert werden können. Die Interaktion zwischen Services erfolgt hierbei über eine einheitliche und minimalistische Schnittstelle, die durch die Verwendung von Hypermedia lose aneinandergekoppelt sind (siehe [Tilkov-15]).

XML & Co.

Markup-Sprachen auf der Basis von XML bilden immer noch die *lingua franca* von SOAP-basierenden Services, obwohl Alternativen wie JSON bereitstehen und im modernen Webumfeld weiter verbreitet sind.

Als Softwarearchitekt in SOA-Projekten sollten Sie über ein grundlegendes Verständnis von XML-Techniken verfügen. Dazu zähle ich die folgenden:

- XML, Attribute, Entitäten und „Content“, Wohldefiniert- und Wohlgeformtheit, XML-Namensräume,
- XML-Schema zur Beschreibung konkreter XML-Sprachen und deren Syntax,
- XPath zur Navigation und Selektion innerhalb von XML-Dokumenten.

Ein verbreitetes Format zum Nachrichtenaustausch innerhalb einer SOA stellt SOAP dar. Das Gegenstück zur Beschreibung von Services heißt WSDL (Web-Service Description Language).

Ablaufsteuerung und Koordination von Services

Services sollen koordiniert ablaufen und durch ihr Zusammenwirken die Unternehmensprozesse unterstützen. Diese Koordination kann entweder durch eine zentrale Steuerung (im Sinne eines Dirigenten, der die Musiker eines Orchesters koordiniert) erfolgen oder aber durch föderierte Intelligenz (im Sinne eines Balletts, in dem sich Tänzerinnen und Tänzer eigenständig koordinieren).

Beide Varianten, in der SOA-Terminologie *Orchestrierung* beziehungsweise *Choreographie* genannt, kommen in SOA zur Ablaufsteuerung von Services zum Einsatz – die zugehörigen Standards heißen BPEL[4] und WS-CDL[5].

Technische Infrastruktur für SOA

Eingangs haben wir bereits Registries als Verzeichnisse von Service-Metadaten erwähnt. Solche Registries übernehmen zur Laufzeit die Vermittlung von Anfragen zwischen Service-Consumern und Service-Providern. Vorher müssen die Service-Provider eine solche Registry mit den notwendigen Metadaten über den jeweiligen Service befüllen. Eine Registry kann darüber hinaus interessante Informationen über das Nutzungsverhalten von Services sammeln, die Einhaltung von Service-Verträgen überprüfen oder zwischen unterschiedlichen Policies von Consumern und Providern vermitteln.

---

[3] Eine geniale Einführung in diesen Architekturstil gibt [Hohpe+03] als „Integrationsmuster“.

[4] BPEL: *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*

[5] WS-CDL: *http://www.w3.org/TR/ws-cdl-10/*

Andererseits bilden Registries auch die Grundlage von Service-Wiederverwendung: Zur Entwicklungszeit können sich Service-Designer und -Entwickler über vorhandene Service-Verträge informieren. Manche Verzeichnisse bieten außerdem die Möglichkeit, sämtliche weiteren Dokumente über Services zu verwalten, etwa Anforderungsdokumente, Testberichte oder Ähnliches. Diese Werkzeugkategorie heißt dann „Service-Repository“ und könnte grundsätzlich von einer Registry getrennt ablaufen.

Die dritte Kategorie von Werkzeugen, die wir ansprechen möchten, ist der Enterprise-Service-Bus (ESB). Ein ESB vernetzt sämtliche technischen Beteiligten einer SOA miteinander. Möchte ein Service-Nutzer mit einem Service-Provider in Kontakt treten, so übernimmt der ESB die gesamten Details der Kommunikation – inklusive notwendiger Zusatzleistungen. Auf den ersten Blick ähnelt dieser Ansatz dem Request-Broker aus CORBA, er verfügt jedoch über einige zusätzliche Facetten:

- Technische Verbindung aller Komponenten, inklusive der notwendigen Netzwerk- und Protokolldetails.
- Ein Service-Bus muss Brücken zwischen heterogenen Technologien schlagen: Service-Consumer sind möglicherweise in anderen Programmiersprachen entwickelt als Service-Provider. Der ESB abstrahiert zusätzlich von Betriebssystemen und Middleware-Protokollen.
- Kapselung verschiedener Kommunikationskonzepte: Der ESB ermöglicht beispielsweise die Kommunikation synchroner und asynchroner Komponenten miteinander oder aber die Übersetzung zwischen verschiedenen Protokollen.
- Bereitstellung technischer Dienste: In einer „echten“ SOA müssen neben den eigentlichen (wertschöpfenden) Services noch technische Dienste vorhanden sein, wie etwa Logging, Autorisierung oder Nachrichtentransformation. Diese werden über den Service-Bus angeboten (oder durch ihn vermittelt.).

Governance und Management

So wie Unternehmen früher Dutzende von Anwendungen (Programmen, Systemen) eingesetzt haben, können in einer SOA schnell viele Hundert verschiedene Services entstehen. Den Überblick zu behalten, deren Weiterentwicklung (oder Abschaltung) gezielt zu steuern, Qualitätsvorgaben zu überprüfen und die Services ggfs. auch extern anzubieten – all das bedarf einer Management- oder Steueraufgabe: SOA Governance.

Dies erfolgt meist im Rahmen des Enterprise Architecture Management (EAM).

10.4 SOA und Softwarearchitektur

Am Anfang dieses Kapitels schrieben wir, dass SOA ein Business-Thema ist. Hier nochmals zur Wiederholung:

> ... Es geht um unternehmerische Flexibilität, um die Fähigkeit, auch unter veränderlichen Marktbedingungen als Unternehmen wertschöpfende Geschäftsprozesse anbieten zu können. ... Technik ist nur Mittel zum Zweck.

Insofern schätzen wir den Zusammenhang von SOA, Softwarearchitektur und anderen Disziplinen wie folgt ein: Vornehmlich gehört SOA in den organisatorischen Bereich der Unternehmen.

- Die Annäherung von IT und Business muss organisatorisch vollbracht werden – hier haben Softwarearchitekten keinen nennenswerten Einfluss.
- Anschließend gilt es, die IT-Gesamtarchitektur der Unternehmen serviceorientiert zu organisieren. Hierzu erhalten Softwarearchitekten im Idealfall passende Vorgaben von Enterprise-Architekten – im schlimmsten Fall darf ein (armer) Softwarearchitekt eines kleinen Entwicklungsprojekts die Weichen der Gesamt-SOA stellen (eine undankbare Aufgabe, weil kurzfristige Projektziele häufig von langfristigen Unternehmens- und SOA-Zielen abweichen).
- Danach sollten Unternehmen eine wirkungsvolle und durchgängige „Regierung" für die Gesamt-IT sowie ihre SOA etablieren – neudeutsch als „Governance" bekannt.

Deshalb können Softwarearchitekten nur in sehr geringem Maße die Wege einer unternehmensübergreifenden SOA bestimmen. Andererseits bestimmen technische Entwurfsentscheidungen darüber, ob Service-Implementierungen den Erwartungen ihrer Nutzer gerecht werden – und solche Entscheidungen treffen Softwarearchitekten.

Um es mathematisch zu formulieren: Für den Erfolg von SOA sind Softwarearchitekten eine notwendige, aber nicht hinreichende Bedingung.

10.5 Microservices

Wie eingangs erwähnt, geht es bei Microservices vor allem um die Modularisierung und Flexibilisierung von Software. Allerdings endet diese nicht bei der Entwicklung, sondern geht über das Deployment und Testen bis hin zum Betrieb der Software. Während Änderungsanforderungen durch den Einsatz agiler Methoden oft noch schnell in die Software eingebaut werden können, wird deren Auslieferung in Produktion meist durch lange und komplexe Prozesse verzögert. Microservices treten an, dies zu ändern.

Allem voran gilt es, die organisatorischen Weichen zu stellen und den Fokus von technologiegeprägten Teams (Architekten, DBAs, UI-Entwickler, Tester, Administratoren) auf Produkte bzw. Fachlichkeiten zu setzen und dies durch funktionsübergreifende Teams zu unterstützen. Diese Teams sollen sich von Anfang bis Ende, von Konzeption bis Produktion um „ihr Produkt" kümmern – auf den Punkt gebracht durch den von Amazon geprägten Satz „*You build it, you run it*".

Hierbei kommt dem Systemschnitt respektive der Aufteilung der Microservices eine wichtige Bedeutung zu. Wurde eigentlich zusammengehörende Fachlichkeit (Business Domain) in mehrere Microservices aufgeteilt, so müssen bei der Entwicklung einer neuen fachlichen Anforderung mehrere Microservices gleichzeitig geändert und im schlimmsten Fall die Änderungen aufeinander abgestimmt werden. Aus diesem Grund sollten Sie Microservices grundsätzlich mit möglichst hoher fachlicher Kohäsion bauen[6]. Ob dies gegeben ist, kann man oft an den Kommunikationsmustern oder der Kopplung eines Microservices erkennen. Im besten Fall kann er seine Funktionalität ohne Kommunikation zu anderen Diensten erfüllen.

Spätestens wenn Microservices mit anderen Systemen kommunizieren, ist dies ein systemübergreifendes Thema, das nicht für einen Microservice allein respektive innerhalb eines Teams entschieden werden sollte[7]. Dies ist ein Beispiel dafür, dass Entscheidungen im Microservice-Architekturstil auf unterschiedlichen Ebenen getroffen werden.

Domänen-, Makro- und Mikroarchitektur

Wird eine Anwendung oder ein System aus mehreren Microservices zusammengesetzt, so spricht man häufig von einem „*System of Systems*". Wir können Architekturentscheidungen für ein solches System auf drei unterschiedlichen Ebenen treffen, die jeweils eine andere Perspektive auf unser „System of Systems" vermitteln:

- *Domänenarchitektur*
  Sie definiert, welche fachlichen Blöcke – Domänen, Subdomänen und Komponenten – es im *System of Systems* gibt und wie sie zusammenhängen. Jede Komponente bietet eine definierte fachliche Funktionalität und deckt alle Aspekte ab, die notwendig sind, um diese Funktionalität bereitzustellen. Dies impliziert, dass jede Komponente ein autonomes System mit eigenem Lebenszyklus ist, welches alle technischen Einheiten wie Persistenz, Benutzeroberfläche und Logik beinhaltet. Die Domänenarchitektur macht jedoch explizit keine Vorgaben zu den technischen Einheiten. Diese Aspekte werden auf Makro- und Mikroarchitekturebene definiert.

  Die Dimension der Domänenarchitektur kann sehr unterschiedlich sein und mit der Zeit wachsen. Allerdings darf sie auch bei einem „Microservices im Kleinen"-Anfangsprojekt nicht vernachlässigt werden, da sie die große Perspektive des *System of Systems* vorgibt.
- *Makroarchitektur*
  Auf dieser Ebene finden Sie alle system- und serviceübergreifenden Entscheidungen und Regeln. Als Beispiele können dies die Plattform (Hardware, OS etc.), Kommunikationsprotokolle (RESTful HTTP, Atom, AMQP etc.), Applikationsmetriken, Anforderungen zum Monitoring, Deployment-Spezifika oder generelle Muster für die Integration der Systeme sein.

[6] Hohe Kohäsion ist grundsätzlich eine gute Idee – siehe Kapitel 4.
[7] Um dabei Konsistenz innerhalb des gesamten Systemverbunds sicherzustellen.

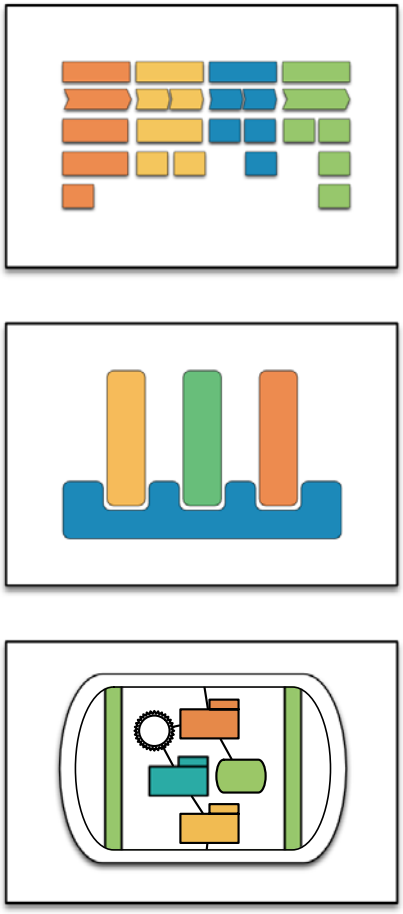

Bild 10.4
„Architekturebenen für Microservices“
(Quelle: Oliver Tigges, innoQ)

Es ist also eine Abstraktionsebene, die vor allem das Zusammenspiel zwischen den Systemen regeln soll, dabei aber nur das notwendige Minimum vorgibt, um jedem System den maximalen Freiraum zu geben, die technisch beste Lösungsstrategie für seine spezifische fachliche Anforderung zu wählen.

- *Mikroarchitektur*
  Die Mikroarchitektur legt letztlich unter Beachtung der Vorgaben von Domain- und Makroarchitektur die Konzepte und Technologien für ein einzelnes System fest. Anders als bei Domain- und Makroarchitektur können diese theoretisch für jedes System unterschiedlich sein. Diese Freiheit bringt die Flexibilität, für die gegebenen fachlichen Anforderungen die passendsten Entscheidungen zu treffen und die effizientesten Lösungsstrategien zu wählen. Auf dieser Ebene wird auch entschieden, aus welchen Microservices unsere Komponente bestehen soll.

Die Unterscheidung von Mikro- und Makroarchitektur erlaubt es Softwarearchitekten, sich bewusst zu machen, welche Elemente der Architektur sich auch an Unternehmensarchitektur bzw. an Softwarearchitekten anderer Systeme richten und wann eine Abstimmung mit Umsystemen entfallen kann.

Dezentralisierung

Grundsätzlich verfolgt der Microservice-Architekturstil das Ziel der Dezentralisierung. Die Erfahrung zeigt, dass Zentralisierung – ob technisch (z. B. ESB) oder fachlich (z. B. kanonisches Datenmodell) – zu Flaschenhälsen auf mehreren Dimensionen (zeitliche Verzögerung, inhaltliche Einigung, Koordinierungsaufwand etc.) führt und somit Handlungs- und Reaktionsfähigkeit sowie die Wandelbarkeit von Software einschränkt. Zudem sind lokale Entscheidungen meist effektiver als globale Abstraktionen (Subsidiaritätsprinzip). Dabei werden bewusst gewisse Redundanzen in Kauf genommen (z. B. Mehrfachaufwände durch ähnliche Schnittstellenkonsumenten in unabhängig entwickelten Systemen). Jedoch erlauben ein regelmäßiger Abgleich zwischen Systemen sowie daraus abgeleitete Konventionen die nachträgliche Konsolidierung, idealerweise basierend auf kontinuierlicher Kommunikation (Technical Validation) zwischen den verschiedenen Teams. So findet nicht nur ein regelmäßiger Wissensaustausch zwischen den Teams statt, sondern die technischen Lösungskonzepte eines Teams können auch von anderen Teams nochmal validiert werden.

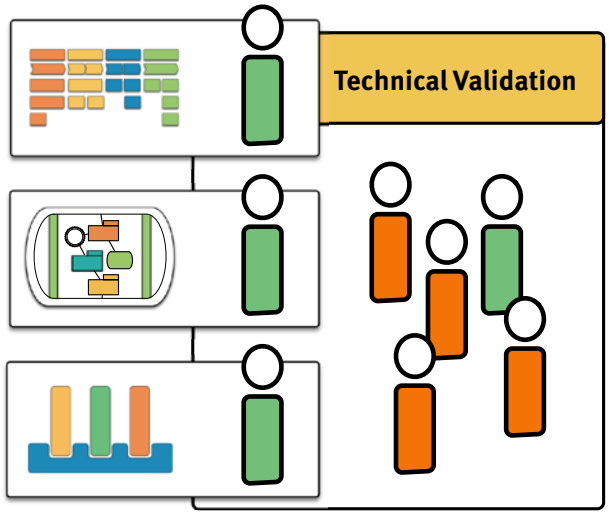

Bild 10.5
„Technical Validation“: Erfahrungsaustausch zwischen Teams
(Quelle: Oliver Tigges, innoQ)

Beispiel: Datenhoheit

Eine andere Form der Dezentralisierung ist der Verzicht auf die Definition eines gemeinsamen Datenmodells. Im starken Kontrast zum unternehmensübergreifenden Domänenmodell wird diese Entscheidung bei Microservices auf der Mikroarchitekturebene getroffen. Für eine bestimmte Entität ist immer genau ein führendes System („Master“) zuständig. Dennoch gibt es ggf. domänenspezifische Ergänzungen, die den Master-Datensatz referenzieren. Somit entsteht ein virtuelles Datenmodell aus der Gesamtheit (Kombination von Master-Datensatz und domänenspezifische Ergänzungen) – jedoch bestimmt jede Domäne ihre eigene Sicht (ein beliebiger Ausschnitt des virtuellen Datenmodells).

In der Konsequenz hat jeder Microservice sein eigenes Datenmodell. Hierbei ist es selbstverständlich, dass Entitäten wie bspw. ein Kunde in mehreren Systemen benötigt werden. Allerdings kann jeder Service eine andere Sicht auf die Entität haben, wodurch diese jeweils andere Attribute hat. Am Beispiel wäre für einen Produktkatalog die Rechnungsadresse eines

Kunden nicht relevant, während sich das Rechnungssystem nicht für das Alter des Kunden interessiert. Beide Systeme verwenden aber den Namen und die Kundennummer, welche meist von einem CRM-System, dem führenden System für die Entität „Kunde“ verwaltet werden.

Diese Form der Datenduplikation wird gerne in Kauf genommen, da hierdurch große Flexibilität gewonnen wird. Sie können damit Services genau so entwerfen, dass sie zu einem bestimmten Use-Case passen. Szenarien wie „Service X liefert die drei Attribute, die wir benötigen, also müssen wir ihn benutzen, denn im unternehmensübergreifenden Modell ist kein anderer Service hierfür vorgesehen, auch wenn er darüber hinaus viel mehr Daten liefert und drei andere Backendservices anfragt, um diese (überflüssigen) Daten zu beschaffen“ werden dadurch explizit vermieden. Dies trägt nicht nur zu einer effizienteren Nutzung der Ressourcen bei, sondern verhindert auch längere Laufzeiten durch unnötige Aufrufkaskaden. Letztere haben meist auch noch negativen Einfluss auf die Verfügbarkeit der Anwendung.

Den Sorgen um die Datenintegrität bei solcher Datenduplikation begegnen Sie mit asynchronem Abgleich der Daten. Im obigen Beispiel wäre das CRM-System beispielsweise das führende System für Kundenstammdaten (Kundennummer, Name, Adressen, etc.) und könnte Änderungsbenachrichtigungen zu seinen Kundendatensätzen in einem Feed bereitstellen oder diese via Publish/Subscribe als Nachrichten an „Abonnenten“ verschicken.

Integration von Microservices

Auch bei der Integration von Services sind die unterschiedlichen Konzepte zwischen typischen SOA- und Microservices-Infrastrukturen erkennbar. Viele SOA-Projekte beinhalten die Einführung von Produkten wie eines Enterprise Service Bus (ESB), eines Orchestrierungs- oder BPM-Produkts oder anderer Middleware. Die Integration von Microservices erfolgt im Gegensatz dazu nicht über mächtige Middleware-Komponenten. Die Orchestrierung wird auch hierbei bewusst dezentralisiert, so dass die interagierenden Services die notwendige Logik für ihre Interaktion mitbringen und Middleware nur als Kommunikationskanal für den Datenfluss dient. Das Resultat wird gemeinhin als „smart endpoints, dumb pipes“ bezeichnet. Dies erzeugt eine höhere Autonomie und nützt dem Ziel, die Microservices möglichst unabhängig weiterzuentwickeln und in Produktion zu bringen. Ebenso setzen Microservices auf einfache Kommunikationsprotokolle wie beispielsweise AMQP oder RESTful HTTP. REST bringt mit Hypermedia bzw. HATEOAS bereits viele Eigenschaften mit, die für die Implementierung von abwärtskompatiblen, lose gekoppelten Schnittstellen wichtig sind.

Der Grund, warum Microservices gerade jetzt eine große Popularität bei Softwarearchitekten erfahren, liegt wohl auch darin, dass viele der schwierigen Herausforderungen von verteilten Systemen wie beispielsweise die Automatisierung von Betriebsprozessen bereits seit geraumer Zeit durch Bewegungen wie DevOps angegangen werden und die meisten Herausforderungen als handhabbar gelten.

Microservices in jedem Fall?

Wie alle Architekturansätze passen Microservices nicht in jeder Situation! Um ein Gesamtsystem, das eine gewisse Größe übersteigt und parallel von mehreren, möglichst eigenständigen Teams entwickelt werden soll, umzubauen, müssen diese Teams die Produktionsverantwortung für „ihren“ Service übernehmen können und wollen – dann kann sich der technische Mehraufwand lohnen. Eine ausgereifte, automatisierte Infrastruktur für Deployment, Betrieb und Monitoring ist ebenfalls eine notwendige Voraussetzung, um einem der Hauptkritik-

punkte an Microservices, der „Komplexität durch die Aufteilung“, zu begegnen. Wenn die einzelnen Services miteinander kommunizieren müssen, sind eine Überwachung und Koordination immer sinnvoll. Allerdings ist zu hoher Kommunikationsaufwand zwischen den Services auch ein Hinweis, den Service-Schnitt noch einmal auf den Prüfstand zu stellen.

Zusätzliche Probleme drohen, wenn die Granularität der Microservices zu klein gerät: Die entstehende Mengenkomplexität kann schwer handhabbar sein, weil in der Service-Welt zur Zeit keine Ordnungs- oder Modularisierungskonzepte entstehen: Stellen Sie sich ein objektorientiertes System ohne Packages oder Namensräume vor ... Tausende von Klassen ohne weitere Ordnung ☹ Diese Komplexität sollten Sie in Microservices unbedingt vermeiden.

Alexander Heusingfeld ist Senior Consultant bei innoQ. Als Berater, Entwickler und Architekt unterstützt er Kunden durch seine langjährige Kenntnis von Java und JVM-basierten Systemen. Meist beschäftigt er sich dabei mit dem Design, der Evaluierung und Implementierung von Architekturen für Enterprise Application Integration (EAI) und modernen Webanwendungen. Er trägt sehr gerne zu OpenSource-Projekten bei, spricht bei Konferenzen und Java User Groups und bloggt gelegentlich unter http://goldstift.de/.

10.6 Weiterführende Literatur

[SOAX 07] sammelt Grundlagen- und Erfahrungsberichte vieler Experten zu SOA und verwandten Themen. Deckt auf fast 900 Seiten sämtliche Bereiche von SOA ab, organisatorische wie technische.

[Lublinsky 07] beschreibt SOA als Architekturstil, kurz und prägnant. Lesenswert, enthält viele Referenzen. Gut finde ich seinen Ansatz einer *Pattern-Language* für SOA.

[Melzer 07] erläutert die grundlegenden Aspekte, die hinter SOA und Web-Services stecken.

Anmerkung: Teile dieses Kapitels entstanden auf Basis des „Einmaleins der SOA“ von Stefan Tilkov und Gernot Starke aus [SOAX 07].

[Tilkov 15] Das REST-Buch, mittlerweile in der 3. Auflage. Hilfreich für alle, die mit der Integration in modernen Websystemen zu tun haben.