Jenseits der grünen Wiesen und der ESBs

Integration mal anders

Die grünen Wiesen sind alle schon zugebaut – heute muss sich praktisch jedes Projekt mit Integration in der einen oder anderen Weise beschäftigen. Abseits der üblichen Pfade wie ESBs gibt es einige Architekturansätze, mit reden Hilfe man mit diesen Herausforderungen umgehen kann.

von Eberhard Wolff

Oft wird die Integration verschiedener Systeme durch einen Enterprise Service Bus (ESB) gelöst, über den dann alle Systeme ihre Daten austauschen können. Von einem ESB wird dann erwartet, dass alle Integrationsprobleme verschwinden. Aber am Ende stellt sich heraus, dass es doch nur eine Technologie ist, die mit Verstand verwendet werden muss. Ein ESB ist eine Software, die verschiedene Netzwerkprotokolle und Applikationsprotokolle unterstützt, um so die Dienste verschiedener Anwendungen anzuschließen und zu integrieren. Für die Kommunikation werden jeweils spezialisierte Adapter genutzt. Auch die Transformation von Daten zwischen verschiedenen Formaten ist möglich. Zusätzlich kann oft eine Orchestrierung mithilfe einer Process Engine erfolgen, die den Ablauf eines komplexeren Dienstes koordiniert, der aus den bereits vorhandenen Diensten zusammengesetzt werden kann. Die Dienste können in einem Verzeichnis registriert werden, um so die Verwaltung zu vereinfachen und eine Verteilung der Dienste auf andere Server zu ermöglichen. Ein ESB hat also alle Features, mit denen Dienste koordiniert und integriert werden. An sich ist also damit das Thema der Integration gelöst. Aber in der Realität gibt es beim Einsatz von ESBs einige Herausforderungen:

- Damit die Dienste Daten austauschen können, müssen die Datenmodelle ineinander überführt oder es muss im Idealfall ein allgemeines Datenmodell definiert werden. Aber die Anwendungen können fachlich bedingt unterschiedliche Sichten auf die Daten haben. Für das Liefern von Waren sind andere Daten eines Kunden relevant als für die Erstellung einer Rechnung. Ein allgemeines Datenmodell zu finden, kann schwer sein und bringt in einigen Situationen auch keinen echten Vorteil.
- Ein ESB ist ein Netzwerkdienst, und die Dienste sind ebenfalls auf verschiedenen Rechnern installiert. Eine Nutzung eines ESBs widerspricht damit der ersten Regel für verteilte Objekte (Verteile deine Objekte nicht! [1]), denn die Dienste werden durch den ESB genau zu solchen verteilten Objekten. Daher hat jede Aktion eine hohe Latenz, weil die Kommunikation mit entfernten Rechnern über ein Netzwerk stattfindet.
- Der ESB kann nur einen bestimmten Datendurchsatz schaffen, weil eben ein Netzwerk für die Kommunikation genutzt wird.
- Der ESB ist eine zentrale Kommunikationsinfrastruktur, die nur bis zu einem bestimmten Punkt skaliert.
- Jedes System, das an den ESB angeschlossen wird, hat nur eine technische Abhängigkeit zum ESB (**Abb. 1**). Fachlich kann es aber mit vielen Systemen kommunizieren. Dadurch können sich nahezu beliebige fachliche Abhängigkeiten ergeben (im Extremfall wie in **Abb. 2**). So kann eine fachliche Änderung an einer Komponente Auswirkungen auf zahlreiche andere Komponenten haben, die schwer zu durchschauen sind. Änderungen am System werden so schwer umzusetzen oder werden wegen des Risikos ganz unterlassen. Dieses Vorgehen widerspricht dem sonst üblichen Ansatz, Abhängigkeiten zu managen und darauf einen Fokus bei der Architektur zu legen. Letztendlich können durch diese Abhängigkeiten die Beziehungen so komplex werden, dass das eigentliche Ziel des ESBs völlig pervertiert wird: Statt leichterer Änderbarkeit und Anpassbarkeit wird das System immer schwerer zu warten und zu ändern. Übrigens gibt es gänzlich andere Ansätze für Integrationen: Der Aufruf einer Webseite bei Amazon führt beispielsweise zu dem Aufruf mehrerer Systeme, die jeweils einen Teil der Seite beisteuern. Diese Systeme kommunizieren dann aber nicht mehr untereinander. Dadurch entsteht keine Bus-Struktur, sondern eher eine Art Seestern, bei der die Seite nur von den Services abhängt, die aber selber keine weiteren Abhängigkeiten mehr haben. Dadurch können die einzelnen Bestandteile des Systems leicht geändert werden, weil die Änderungen nur lokal begrenzte Auswirkungen haben.
- ESBs sind komplex. Alleine bei Web Services gibt es mehr als 40 Standards, die ein ESB unterstützen kann, und es gibt zahlreiche weitere Protokolle und Standards. Daher ist ein ESB zwar sehr mächtig, aber auch schwierig zu handhaben.
- Der ESB ist oft monolithisch. Er bietet viele Dienste, und auch wenn nicht alle genutzt werden, muss dennoch immer das gesamte Produkt installiert und bezahlt werden.

Mehr zum Thema

Mehr zum Thema Messaging mit RabbitMQ gibt es auf JAXenter unter http://www.jaxenter.de/artikel/4016.

Eine weitere umfassende Kritik am ESB bietet [2].

Der Einsatz eines ESB ist trotz dieser Probleme machbar. Dann sollten allerdings die Performanceimplikationen bei der Architektur beachtet und nur die Teile des ESB genutzt werden, die wirklich notwendig sind. Das Management der Abhängigkeiten zwischen den Anwendungen darf ebenfalls nicht vernachlässigt werden, nur weil nun eine Integrationsplattform genutzt wird. Wichtig ist außerdem, dass der ESB so genutzt wird, dass fachliche Ereignisse ausgetauscht werden, also zum Beispiel „Eine Bestellung ist eingetroffen". Daraufhin kann dann eine Komponente zum Beispiel eine Rechnung erstellen. Wird nun ein Bonusprogramm eingeführt, kann das ebenfalls auf solche fachlichen Nachrichten reagieren und für die Bestellung einen Bonus gutschreiben.

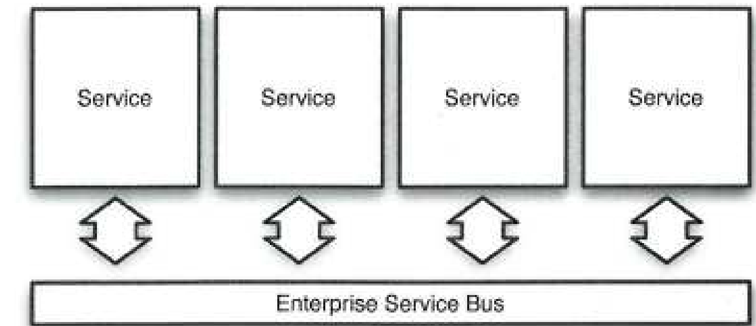

Abb. 1: Integration mit einem ESB: technische Sicht, saubere Abhängigkeiten

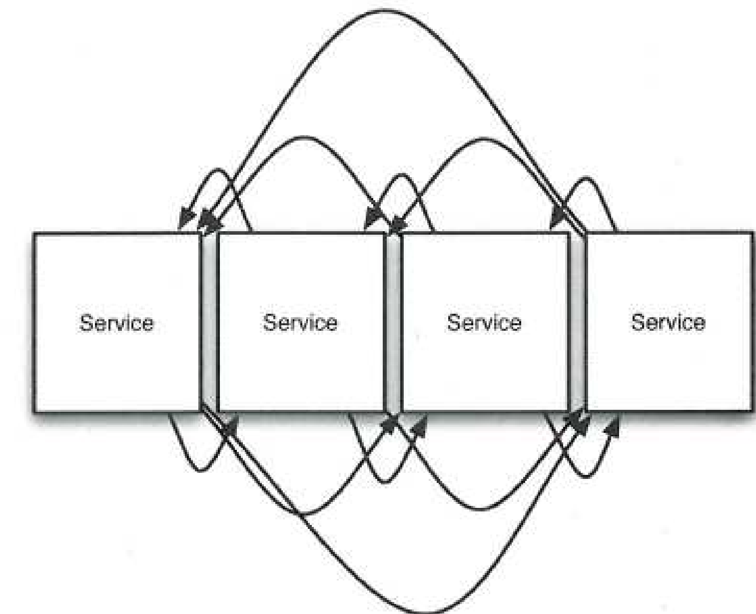

Abb. 2: Integration mit einem ESB: fachlich redet jeder mit jedem/Chaos

Alternative: Integration leichtgewichtig und flexibel

Ein Ansatz, mit Integration anders umzugehen, ist die Nutzung eines individuellen Technologie-Stacks anstelle eines ESB. Dadurch ist die Lösung kein Monolith, sodass jene Technologien genutzt werden können, die tatsächlich benötigt werden. Außerdem können verschiedene Technologien mit ihren jeweiligen Vorteilen und Einsatzszenarien kombiniert werden (**Abb. 3**). Dabei müssen die einzelnen Features, die sonst ein ESB anbietet, durch andere Technologien implementiert werden.

Für die Übertragung der Nachrichten kann eine Message-oriented Middleware genutzt werden, die auf JMS basiert. Dazu zählen zum Beispiel ActiveMQ aus dem Apache-Projekt, WebSphere MQ von IBM, SonicMQ, HornetQ von JBoss usw. Eine Alternative ist eine Message-oriented Middleware wie RabbitMQ [3] oder Apache Qpid, die den AMQP-Standard implementiert. Diese Komponente stellt die Übertragung von Nachrichten zur Verfügung, die auch die Basis eines ESB ist. Aber solche Lösungen sind viel leichtgewichtiger: Sie können in Extremfällen sogar auf eingebetteten Devices laufen.

Die Logik für die Integration kann mit verschiedenen Mechanismen aufgebaut werden. Wenn sie in Java implementiert wird, können Frameworks wie Spring Integration oder Apache Camel genutzt werden. Sie bieten Abstraktionen an, die zu den Enterprise Integration Patterns [4] passen. Diese Patterns basieren auf der asynchronen Verarbeitung von Daten. Zu ihnen zählen zum Beispiel Router, Translator oder Adapter. Diese Patterns werden in den Frameworks direkt durch passende Konstrukte unterstützt, sodass recht leicht Integrationslogik implementiert werden kann.

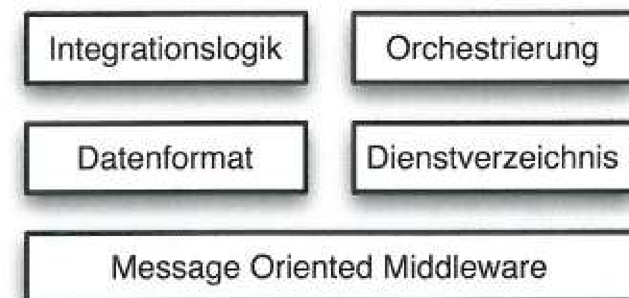

Abb. 3: Technologie-Stack statt ESB

Für die Komposition von Services kann beispielsweise Activiti oder jBPM genutzt werden. Im Gegensatz zu anderen Business Process Engines fokussieren diese Projekte auf eine gute Unterstützung innerhalb eigener An-

wendungen. Das bedeutet, dass sie wie ein Framework in einer Anwendung eingebettet laufen kann. Gleichzeitig können Abläufe grafisch modelliert werden, und sie unterstützen auch einige Standards wie BPMN 2.0.

Die Logik und auch die Abläufe könnten mit einer Skriptsprache wie Groovy implementiert werden. Dadurch kann das System manipuliert werden, während es läuft – der Code muss nur neu geladen werden. Dabei verschwimmt aber die Grenze zwischen Konfiguration und Code. Die Änderung eines Geschäftsprozesses mit einer darauf spezialisierten Ausdrucksform wie BPMN ist eher eine Konfiguration. Wenn es in Groovy geschrieben wird, ist es Code und kann nahezu beliebiges tun. Oft sind die Releaseprozesse für Konfigurationsänderungen auch anders als jene für Codeänderungen, weil geänderter Code intensiver getestet wird. Daher kann eine solche Vermischung von Code und Konfiguration sehr problematisch sein, weil sie diese Sicherheitsmechanismen unterläuft.

Dienstverzeichnisse zu implementieren ist ebenso möglich. So kann bei der Spring-JMS-Abstraktion der Empfänger einer Nachricht mit einem *DestinationResolver* ermittelt werden. Es ist dann möglich, als Empfänger einer Nachricht den Namen eines Dienstes anzugeben. Welcher konkrete technische Kanal genutzt wird, kann beim Versenden der Nachricht aus dem Dienstverzeichnis ausgelesen werden. Ebenso ist es möglich, die Daten durch einen *MessageConverter* zum Beispiel transparent für den Entwickler von Java in XML oder JSON zu übersetzen. Diese Formate sind allgemein nicht besonders effizient, sodass Alternativen wie Protocol Buffer [5], Apache Thrift oder Apache Avro betrachtet werden sollten. Diese Protokolle sind ebenfalls plattform- und sprachunabhängig, versprechen aber eine wesentlich effizientere Kodierung der Daten, also JSON oder XML, sodass sich die Performance der Integrationslösung verbessert.

In bestimmten Situationen können auch ganz andere Lösungen genutzt werden. So ist es ja nicht ungewöhnlich, Daten aus einem System, zum Beispiel über Nacht, in ein anderes System zu replizieren oder anderweitige größere Datenmengen in einem Rutsch zu übertragen, zu bearbeiten oder zu synchronisieren. In solchen Situationen ist es kaum sinnvoll, einen Message-oriented Middleware zu nutzen, weil sie für die Bearbeitung großer Datenmengen durch die Verteilung und Performance keine optimale Lösung ist. Stattdessen können in solchen Szenarien Batches helfen, die direkt auf die Massendaten optimiert sind. Hier finden Frameworks wie Spring Batch ihren Anwendungskontext, die typische Probleme bei der Batch-Verarbeitung wie Wiederanlauffähigkeit, Optimierungen bezüglich der Transaktionen oder auch Logging adressieren.

Alternative: Ad-hoc-Integration

Statt einen zentralisierten Enterprise Service Bus einzuführen, können natürlich auch die Systeme jeweils mit eigenen Protokollen und Ansätzen integriert werden. Oft schlagen Organisationen sowieso diesen Weg ein. Dadurch können je nach Einsatzkontext die jeweils passenden Technologien gewählt und genutzt werden. Dazu zählen neben der bereits erwähnten Message-oriented Middleware und Batches auch Web Services nach dem SOAP- oder REST-Ansatz. REST ist ein Akronym für „Representational State Transfer" und bezeichnet Systeme, die typischerweise auf HTTP aufsetzen. Jedes fachliche Objekt hat einen URL, zum Beispiel http://adesso.de/kunde/42. Darauf kann dann mit HTTP-Methoden wie *GET* zum Lesen oder *DELETE* zum Löschen zugegriffen werden. Gerade mit REST kann eine Integrationslösung ganz anders aufgebaut werden:

- Durch seinen URL kann ein Objekt weltweit eindeutig identifiziert werden. Durch Links können einfach Beziehungen zwischen Objekten hergestellt werden, die auch auf verschiedenen Systemen liegen können. Im Extremfall kann über Links durch die Daten des gesamten Systems navigiert werden, sodass ein Client nur wenig Wissen über das System haben muss. Er muss nur irgendwo anfangen und kann dann alle Daten schrittweise finden. Dadurch kann das System flexibel geändert werden, ohne dass die Clients informiert werden müssen.
- REST kann verschiedene Datenrepräsentationen unterstützen, sodass die Daten unterschiedlichen

Was ist mit SOA passiert?

Integration und ESB passen gut zum Thema SOA (Service-oriented Architecture). Unter dem Begriff „SOA" versteht man die Aufteilung der IT in verschiedene Services, die dann komponiert werden können. Dieser Ansatz ist mit vielen Versprechungen und Vorschusslorbeeren gestartet, aber mittlerweile macht sich Ernüchterung breit. Warum ist das so?

- SOA erzeugt selber keinen Geschäftswert. Es erlaubt eine größere Flexibilität durch den Austausch und die Komposition von Services. Dadurch kann sich eine bessere Time to Market ergeben, was ein Wettbewerbsvorteil sein kann. Das ist aber keine sehr direkte Folge, sondern ein langfristiges Ergebnis, das außerdem in seinem wirtschaftlichen Nutzen schwer zu beziffern ist.
- Außerdem bedeutet SOA letztendlich eine komplette Umgestaltung der IT-Landschaft. Das ist aufwändig, dauert lange und hat ein hohes Risiko.
- Diese beiden Faktoren bedeuten, dass einem hohen Aufwand nur ein niedriger Nutzen entgegensteht. Daher ist ein Return on Investment (ROI) nur langfristig zu erreichen, was den Ansatz wenig attraktiv macht.

Dennoch ist die SOA-Idee im Grundsatz nicht schlecht. Dienste bereitzustellen ist ein guter Ansatz für Integration. Von daher können sicher immer noch einzelne Dienste im Rahmen einzelner Projekte implementiert oder auf der Basis vorhandener Systeme zur Verfügung gestellt werden. Nur die komplette Umstellung auf einen Schlag innerhalb eines eigenen Projekts ist wohl wenig sinnvoll. Dazu gibt es auch einen interessanten Blog-Post [7].

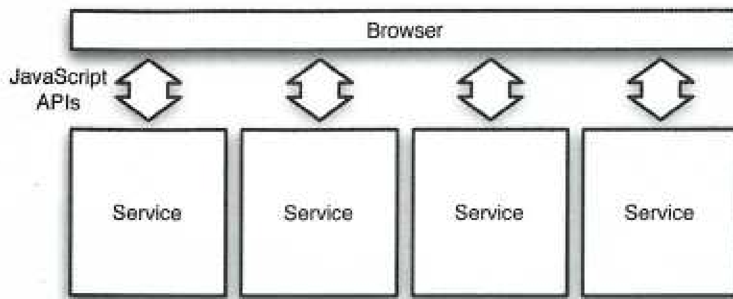

Abb. 4: Integration im Client

Clients in einem jeweils geeigneten Format angeboten werden können.
- Außerdem werden typische Herausforderungen wie der Umgang mit Fehlern elegant gelöst. So gibt es idempotente Methoden wie *DELETE*, die bei mehrfachem Aufruf dasselbe Ergebnis produzieren. Einige Methoden wie *GET* sind sogar „sicher", sie erzeugen keine Seiteneffekte und können beliebig oft wiederholt werden, ohne dass sich das System ändert. Dadurch kann bei einem Netzwerkfehler oft dieselbe Aktion noch einmal ausgeführt werden, um zu gewährleisten, dass sie tatsächlich ausgeführt wird.

REST ist interessant, weil das World Wide Web auf diesem Architekturansatz basiert, und dadurch die Skalierbarkeit dieses Ansatzes außer Frage steht. Außerdem ist er flexibel zum Beispiel bezüglich der Verteilung der Objekte oder der verwendeten Datenrepräsentation. Daher lohnt sich ein Blick auf diesen Ansatz in jedem Fall [6].

Integration auf dem Client

Integration auf dem Client (**Abb. 4**) ist ein weiterer Ansatz, der ebenfalls aus dem Web heraus motiviert ist. Dabei ist die Integration der Systeme nicht mehr im Backend, sondern auf dem Client. Im Web ist das nichts ungewöhnliches: Beispiele sind das Einbinden von Karten aus Google Maps, Fotos aus Flickr und Videos aus YouTube. Diese Dienste werden über JavaScript direkt aus dem Browser heraus angesprochen. Dieser Ansatz wird Mashup genannt. Die Backends bieten nur ein API an, bleiben ansonsten unverändert. Der Vorteil dieses Ansatzes ist, dass die Integration das Backend-System nur wenig beeinflusst. Für eine Geschäftsanwendung ist es nicht schwierig, eine Basis für Mashups zu bieten, die Kunden, Bestellungen oder Waren anzeigen kann. Anschließend können aus diesen Teilen dann Anwendungen zusammengestellt werden. Die Backend-Systeme können dann immer noch als Silos arbeiten und müssen nicht integriert werden. Stattdessen zeigt der Client zum Beispiel die jeweils für den aktuellen Kunden passenden Bestellungen automatisch an.

Dieser Ansatz ist ein Teil eines Trends zum verstärkten Einsatz von JavaScript. Im Bereich Integration bietet beispielsweise Google ein JavaScript API zur Automatisierung über verschiedene Google-Produkte hinweg an [8]. Ebenfalls nutzt die NoSQL-Datenbank CouchDB, um Views, Anfragen und auch sonstige Logik zu definieren. Dienste, die Daten als JSON anbieten, können ebenfalls direkt per JavaScript angesteuert werden. Durch das neue WebSockets API, das als Teil von HTML 5 standardisiert wird, können JavaScript-Clients in Browsern auch Nachrichten direkt an das Backend schicken und umgekehrt das Backend auch direkt an den Browser. Produkte wie Kaazing [9] erlauben die direkte Anbindung eines JavaScript-Clients beispielsweise an JMS-Infrastrukturen. Mit diesem verliert die Integration der Backend-Systeme untereinander an Relevanz, da die damit verbundenen Herausforderungen am Client gelöst werden. Natürlich ist dabei der Endnutzer im Fokus. Eine Datenkonsolidierung der Systeme untereinander ist so nicht möglich.

Fazit

Integration nur mit dem Ansatz ESB zu verfolgen, ist nicht immer angemessen und führt oft auch nicht zu den gewünschten Resultaten. Zudem ist die Einführung eines ESB oft kostenaufwändig und dauert recht lange. Ansätze, die auf einen flexiblen Technologie-Stack, ganz andere Technologien oder Integration im Client setzen, können sinnvolle Alternativen sein. Das ist ein weiteres Beispiel für die generelle Aussage, dass es eben kein Silver Bullet gibt, mit dem alle Probleme gelöst werden können. Der Einsatz einer Technologie ist immer ein Trade-off zwischen verschiedenen Zielen und ein Kompromiss. Die Kunst besteht darin, die verschiedenen Technologien zu kennen und dann den richtigen Ansatz für das jeweilige Szenario zu nutzen. Und das Vorstellen solcher Alternativen war das Ziel dieses Artikels.

Eberhard Wolff (Twitter: @ewolff) arbeitet als Architecture and Technology Manager für die adesso AG in Berlin. Er ist Java Champion, Autor einiger Fachbücher und regelmäßiger Sprecher auf verschiedenen Konferenzen. Sein Fokus liegt auf Java, Spring und den Cloud-Technologien.

Links & Literatur

[1] http://martinfowler.com/bliki/FirstLaw.html

[2] http://www.infoq.com/presentations/soa-without-esb

[3] Wolff, Eberhard: „RabbitMQ, AMQP und Spring", Java Magazin 7.2011

[4] Hohpe, Gregor; Woolf, Bobby: „Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Addison-Wesley, 2003: http://www.eaipatterns.com/

[5] http://code.google.com/p/protobuf/

[6] Tilkov, Stefan: „REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien", dpunkt, 2011

[7] http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html

[8] http://code.google.com/googleapps/appsscript/

[9] http://kaazing.com/