Architektur auf mehreren Ebenen

Wie kommt man zu Self-Contained Systems?

Phillip Ghadir

Self-Contained Systems bezeichnet ein Konzept für Softwarebausteine, das klare Integrationsregeln auf der Makro-Ebene definiert: Systeme werden in eigenständige Teilsysteme zerlegt, die zu einem Gesamtsystem integriert werden. Häufig wird vorgeschlagen, Systeme entlang von Domänen in Self-Contained Systems zu zerlegen. In diesem Artikel wird ein alternativer Ansatz für das Schneiden einer Architektur in föderierte Self-Contained Systems vorgestellt.

Bereits in [Til11] haben Stefan Tilkov und ich beschrieben, dass es sinnvoll ist, große Systeme in Teilsysteme zu zerlegen, und erstmals erläutert, was wir im Allgemeinen unter Mikro- und Makro-Architektur verstehen. Aus verschiedenen Vorträgen, Experimenten und Projekten kristallisierte sich irgendwann der Name für solche Teilsysteme in einem komplexen Gesamtsystem heraus: Self-Contained Systems.

Mikro- und Makro-Architektur

Die Kernidee ist einfach: Architekturregeln werden Ebenen zugeordnet und separat definiert. Bausteine unseres Systems können wir sowohl als Black-Box betrachten und damit von deren internem Aufbau abstrahieren als auch als White-Box, bei der die interne Struktur sichtbar gemacht wird.

Ein wesentlicher Faktor für die Architektur großer Softwaresysteme ist das Identifizieren passender Bausteintypen, für welche auf Black-Box-Ebene isoliert die Regeln für die Entwicklung und Evolution der Architektur definiert werden.

Der nächste wesentliche Faktor ist, für die Entwicklung und Evolution einzelner Bausteine unterschiedliche Regeln verwenden zu können, sofern die Zusicherungen auf der Black-Box-Ebene eingehalten werden.

Die Regeln auf der Black-Box-Ebene der relevanten Bausteintypen bezeichnen wir als Makro-Architektur. Die Regeln für die interne Umsetzung der einzelnen Bausteine zählen zur Mikro-Architektur.

Eigenschaften von Self-Contained Systems

Ein Self-Contained System ...

- ist eine eigenständige Webanwendung.
- wird von einem eigenständigen Team realisiert.
- wird größtenteils asynchron mit Um-Systemen gekoppelt.
- kapselt sowohl die Daten als auch die Logik, die nötig ist, die Hauptanwendungsfälle autark zu erbringen.
- bringt seine eigene Benutzungsschnittstelle mit, die nur die eigenen Anwendungsfälle unterstützt.
- inkludiert keine Fachlogik von Um-Systemen, darf aber von technischen Basis-Bibliotheken abhängen.

Für Self-Contained Systems fordern wir deren strenge Isolation, die sicherstellt, dass eine Verwendung nur über definierte Schnittstellen erfolgen kann. Self-Contained Systems beinhalten sowohl ihre Benutzungsschnittstellen als auch die Verantwortung für die Verwaltung der zugehörigen Daten.

Die Integration von Self-Contained Systems erfolgt typischerweise per HTTP sowie per Nachrichtenaustausch. Bei Bedarf kann neben einer Webanwendung für menschliche Nutzer auch ein Web-API für die Integration mit anderen Systemen ergänzt werden. Innerhalb von Benutzerinteraktionen werden Um-Systeme niemals synchron integriert, sondern ausschließlich asynchron.

Wie genau ein Self-Contained System realisiert ist, bleibt erst einmal offen.

Bezug zu Microservices

[Gha14b] beschreibt die Eigenschaften von Microservices. Technisch betrachtet sind Microservices Single-purpose-Monolithen, die über Remote-Schnittstellen – im Allgemeinen über HTTP – verwendet werden können. Eine ausführliche Behandlung des Themas bietet [Wol15].

Das Konzept der Self-Contained Systems ist auf der Makro-Ebene orthogonal zu einem Microservices-Ansatz, sodass beide gleichermaßen unabhängig voneinander wie auch in Kombination miteinander verwendet werden können.

Insbesondere aufgrund des aktuellen Interesses an Microservices referenziert die öffentliche Beschreibung der Self-Contained Systems [SCS] eben diese für die Ebene der Mikro-Architektur.

Gemeinsamkeiten

Sowohl Self-Contained Systems als auch Microservices sind Ansätze zur expliziten Isolation von Systemen. Während monolithische Systeme ab einer gewissen Größe anfällig dafür sind, dass zwischen Systemteilen unerwünschte – und zum Teil unentdeckte – strukturelle Abhängigkeiten entstehen, führt die Isolation von Self-Contained Systems und Microservices dazu, dass Abhängigkeiten ausschließlich zu exponierten Schnittstellen entstehen können.

Grundwissen Softwarearchitektur

Softwarearchitektur hat mit den wesentlichen Entscheidungen zu tun, die die Softwarestruktur und die Regeln zur Entwicklung und Evolution eines Softwaresystems betreffen.

Der iSAQB e.V. definiert im Lehrplan zum Foundation Level [CPSA-F] die Grundlagen zur Entwicklung und Dokumentation von Softwarearchitekturen. Unter anderem spielt dort die Zerlegung eines Systems in Bestandteile eine Rolle.

Abstraktion ist ein wesentliches Instrument für die Architektur-Arbeit als solche und für die Kommunikation von Ergebnissen. Abstraktion lässt sich durch das konsistente Betrachten von Belangen und Darstellen in entsprechenden Architektur-Sichten erreichen.

Für den Entwurf der statischen Struktur eines Systems erlaubt die Black-Box-Betrachtung von Bausteinen das grobe Zuordnen von Verantwortung und Abstrahieren von den Details, wie die Verantwortung erfüllt werden kann. Im Gegensatz dazu erlaubt die White-Box-Betrachtung von Bausteinen zu beschreiben, wie eine Verantwortung erfüllt werden soll, ohne dabei das gesamte System betrachten zu müssen.

Dadurch unterstützt dieser Ansatz aktiv die Entkopplung von Implementierungsdetails und verbessert so mittel- bis langfristig die Evolution des Systems.

Unterschiede

Ein wesentliches Unterscheidungsmerkmal ist die Größe. Für Microservices gibt es die Forderung der Größenbeschränkung. Für Self-Contained Systems gibt es keine spezielle Anforderung bezüglich der Größe.

Ihre Größenbeschränkung macht Microservices zu perfekt anpassbaren Bausteinen. Konsequenzen von Änderungen und der Umfang der Implementierung lassen sich schnell überblicken.

Allerdings bestünde ein großes System potenziell aus so vielen Microservices, dass deren Anzahl die Beherrschbarkeit drastisch reduziert. Self-Contained Systems adressieren diesen Nachteil dadurch, dass sie mehr Verantwortung als ein Microservice kapseln und eine Systemzerlegung somit mit erheblich weniger Self-Contained Systems auskommt.

Zudem beinhalten Self-Contained Systems eine Benutzungsschnittstelle, wohingegen verschiedene Microservices-Ansätze sich dazu nicht äußern. Darüber hinaus erwartet man für Self-Contained Systems eine definierte Autonomie, während es für Microservices durchaus zulässig ist, miteinander föderiert zu werden.

Für die Integration eines Self-Contained Systems im Front-End bietet sich Resource-oriented Client-Architecture [ROCA] an.

Der Klassiker: Zerlegung in Domänen

Ein üblicher Vorschlag für den Entwurf eines Self-Contained Systems ist, über strategisches Domain-Driven Design abgegrenzte Kontexte (= Bounded Contexts) zu identifizieren und diese als Self-Contained Systems zu realisieren.

Im Zentrum eines Bounded Context stehen typischerweise zentrale Entitäten und Services. Entitäten kapseln bekanntermaßen Daten, die innerhalb einer Domäne verwaltet werden.

Im strategischen Domain-Driven Design liegt der Fokus auf dem Identifizieren von isolierbaren Kontexten und dem Integrieren von solchen Kontexten über definierte Integrationsstrategien. [Eva04] bietet eine Übersicht über den Grad der Autonomie, den die verschiedenen Integrationsstrategien zulassen (s. Abb. 1).

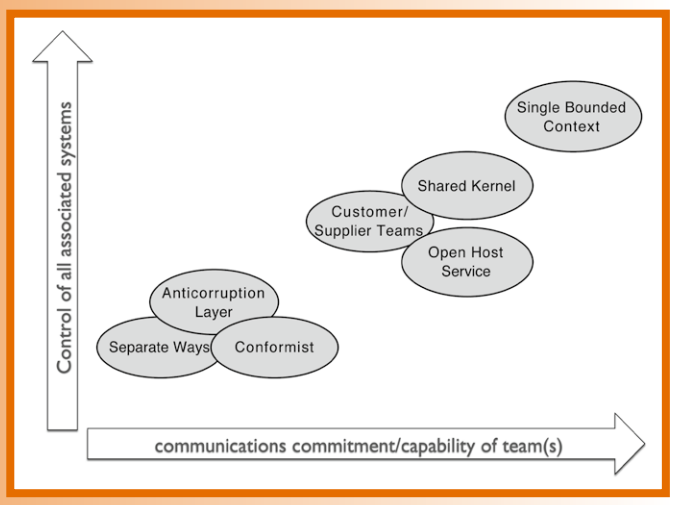

Abb. 1: Anforderung der Kontext-Beziehungen nach Evans

Die Umsetzung eines Bounded Context als Self-Contained System führt recht natürlich zu einer passenden Kapselung und hohen Isolation.

Self-Contained Systems – mehr als eine Kerndomäne

In Diskussionen begegnet mir manchmal die irrige Meinung, ein Self-Contained System entspräche einem fachlichen Kern. Diese Ansicht ist gleich auf mehrere Weisen irreführend:

- Die Implementierung eines Domänenmodells ist frei von technischer Anbindung. Im Sinne des Domain-Driven Design ist ein Self-Contained System eher eine Applikation, die zusätzlich noch technische Aspekte, anwendungsspezifische Anwendungsfälle sowie eine Benutzungsschnittstelle implementiert.
- Die Implementierung eines Self-Contained Systems kann mit Hilfe der verschiedensten Komponenten und insbesondere auch durch die Verwendung von remote genutzten Microservices realisiert werden. Die Integration via Remote-Services kommt im Domain-Driven Design eher bei der Integration von abgegrenzten Bereichen zum Tragen.
- Domain-Driven Design unterscheidet neben Kerndomänen auch generische und unterstützende Domänen. Dennoch liegt der Fokus auf dem fachlichen Verständnis und dessen korrekter Umsetzung von isolierbaren fachlichen Kontexten. Die Integration über die Kontextgrenzen hinweg erfolgt typischerweise dort in der Domänenschicht, während diese bei Self-Contained Systems typischerweise auch in der Applikations- oder Benutzerinteraktionsschicht erfolgt.

Wenn der Kontext zum Problem wird

Sowohl [SCS] als auch [Til11] legen nahe, sich beim Schnitt an Bounded Contexts zu orientieren. Das Umsetzen einer fachlichen Domäne in einem Self-Contained System ist vielversprechend, da es ermöglicht, die gewünschte Fachlichkeit angemessen und isoliert umzusetzen. Fachlich zusammenhängende Daten und Logik sind in einem System gekapselt.

Veränderungen am Markt und eine Reduktion der Organisationstiefe können allerdings dazu führen, dass gewisse Geschäftsfunktionen an andere Organisationen ausgelagert werden müssen.

Wenn dann Funktionalität innerhalb eines Self-Contained Systems auf den dort verwalteten Daten operiert und Teile davon in Zukunft unter anderer Hoheit ausgeführt werden müssen, erfordert dies unter Umständen eine recht umfangreiche Migration, sofern zuvor verlaufsorientierte Daten mit Bestandsdaten gekapselt sind.

Was zuvor innerhalb eines Kontexts „auf dem kurzen Dienstweg“ autorisiert werden konnte, muss jetzt für Externe geöffnet werden. Die Domäne selbst wird aufgrund eines Business Process Outsourcing erneut unterteilt.

Dieses Risiko kann beispielsweise dadurch adressiert werden, indem ein Self-Contained System intern mit Hilfe von Microservices realisiert wird. Darüber hinaus können wir Softwerker versuchen, einige solcher organisatorischen Änderungen vorauszusehen.

Leichter Fokus-Shift

Anstatt uns vollständig auf die Zerlegung in Domänen zu konzentrieren und unseren Fokus auf „fachliche Implementierungs-

details“ zu lenken, können wir alternativ von fachlichen Aufgaben ausgehen, die sich zum Auslagern an andere eignen.

In der Unternehmensarchitektur kennt man häufig ein Konstrukt namens Geschäftsfunktion oder Business Service. Mit solchen sind üblicherweise Geschäftsprozesse assoziiert, die sich unterteilen lassen in:

- Kontrollflüsse,
- Datenflüsse,
- Aktionen,
- Aktivitäten und
- Nachrichten/Events.

Die Geschäftsprozesse – von denen ich hier rede – sind üblicherweise eher abstrakt als ausführbar.

Statt nun ein Self-Contained System um zentrale Entitäten zu stricken, kann man einen relevanten Teilprozess als Microservice oder – falls größer – als Self-Contained System realisieren. Durch Kapselung eines geschäftsrelevanten Teilprozesses entsteht ein Service, der von einer anderen organisatorischen Einheit erbracht werden kann ohne größere Auswirkungen auf die Softwarearchitektur.

Der Fokus-Shift führt zu einer stärkeren Isolierung von Bestands- und Verlaufsdaten. Auch fachliche Teilprozesse kapseln benötigte Daten lokal. Dadurch ist das System weniger anfällig gegen das Auslagern von Geschäftsprozessen, durch das andernfalls Datenkopplungen entstehen könnten.

Die Integration erfolgt bevorzugt über das Propagieren von Ereignissen.

Bootstrapping

Üblicherweise sind Domänenmodelle einfach verständlich. Der Code ist lesbar und drückt aus, was fachlich zu geschehen hat. Für Self-Contained Systems gilt dies nicht immer – insbesondere dann, wenn wir den Fokus für das Schneiden auf Basis von Teilprozessen legen, die ausgegliedert werden könnten.

Ein verteiltes System kann beliebig fragil sein. Und das korrekte Aufsetzen eines aus Self-Contained Systems bestehenden verteilten Systems kann beliebig aufwendig sein. Insbesondere für die Remote-Abhängigkeiten muss das System korrekt aufgesetzt werden. Dabei kann beliebig Vieles schiefgehen – egal ob man das manuell oder automatisiert aufsetzt.

Der Vorteil beim automatisierten Aufsetzen ist allerdings, dass man recht schnell Feedback zum Setup erhält. Auch können notwendige Anpassungen sukzessive weiterentwickelt werden – und das Setup bleibt wiederholbar und anpassbar.

Das Auslagern von Funktionalität auf andere Organisationseinheiten erfordert dann eine Anpassung der Setup-Routinen, die unweigerlich zum Self-Contained System gehören. Werden eigene Services durch externe ersetzt, müssen die Systeme nur richtig miteinander verdrahtet werden.

Wer sich für das ganze Setup-Thema interessiert, wird im DevOps-Umfeld fündig. Eine Einführung dazu bietet zum Beispiel [Wol14].

Stabilität und Widerstandsfähigkeit

In dieser Kolumne haben wir uns schon mit Mustern zur Stabilität von verteilten Systemen beschäftigt (siehe [Gha14a]). Verteilte Systeme sind anfälliger für Störungen in der Umgebung.

Während wir bei statisch gebundenen Bibliotheken stets sicher sein können, dass die Funktionalität korrekt erreichbar ist, weiß man das bei Remote-Aufrufen nie wirklich. Daher ist es gängige Praxis – innerhalb eines Self-Contained Systems – mit technischen Fehlern zu rechnen und bei deren Auftreten dennoch in einem definierten Zustand arbeiten zu können.

Während klassische monolithische Systeme recht gut mit statischer Code-Analyse vermessen werden können, ist dies bei Self-Contained Systems eher unmöglich. Die Verteilung bringt Abhängigkeiten erst zur Infrastruktur. Einiges an Bindung entsteht zur Setup-/Konfigurationszeit. Manches an Bindung entsteht erst zur Laufzeit.

Vieles, was für das fachliche Verständnis nötig wäre, liegt weder in der alleinigen Hoheit eines Source-Projekts noch passiert es innerhalb eines Prozesses. Daher gewinnen Logging, Tracing und Monitoring eine noch höhere Bedeutung als für monolithische Systeme.

Organisationsstruktur

Sowohl im Domain-Driven Design (für eine Domäne) als auch für Self-Contained Systems wird gefordert, dass sich ein Team um die Entwicklung kümmern soll. Durch den engeren Fokus ist ein Team für die Entwicklung eines Domänenmodells tendenziell kleiner als ein Team, das die Entwicklung eines Self-Contained System verantwortet.

Darüber hinaus erfordert die Entwicklung eines Self-Contained Systems zusätzlich zu dem nötigen Know-how für die Entwicklung und Umsetzung des Domänenmodells auch noch Expertise in Front-End-Technologien, um sowohl die Benutzerinteraktionen als auch die Themen der Front-End-Integration zu beherrschen.

Zudem gibt es bei der Entwicklung und Betreuung von Self-Contained Systems die Herausforderung, Authentisierung und Autorisierung zu integrieren – aber nicht deren Benutzungsschnittstellen. Das bringt üblicherweise organisatorische Abhängigkeiten mit sich, die sich ebenfalls im Setup widerspiegeln.

Fazit

Microservice-Architekturen erlauben eine Entkopplung der Softwarestrukturen von konkreten Organisationsstrukturen. Für die Entwicklung sehr umfangreicher Systeme fehlt unserer Meinung nach aber ein Konzept zur Aggregation von Microservices zu etwas Grobgranularerem und Beherrschbarerem.

Self-Contained Systems springen hier in die Bresche und bieten ähnliche Möglichkeiten der Entkopplung, aber erlauben gleichzeitig, große Systeme in eine besser beherrschbare Anzahl von eigenständigen Subsystemen zu zerlegen. Eine Herausforderung liegt dabei im Ausbalancieren der Granularität der Self-Contained Systems und deren Anzahl.

Eine kritische Herausforderung ist hier die Integration von Services. Ein häufiger Ansatz ist die Integration im Frontend – das heißt, in der Oberfläche. Self-Contained Systems können dabei von den Mechanismen profitieren, die das World Wide Web so erfolgreich gemacht haben. Der ROCA-Stil ist für die Integration im Frontend vielversprechend (siehe [ROCA]).

Literatur und Links

[CPSA-F] iSAQB e.V., Lehrplan zum Certified Professional for Software Architecture – Foundation Level, `http://www.isaqb.org/wp-content/uploads/2015/05/isaqb-Lehrplan-foundation-v3-MAI-2015-DE.pdf`

[Eva04] E. Evans, Domain-Driven Design, Addison-Wesley, 2004

[Gha14a] Ph. Ghadir, Hystrix – Wider den Totalausfall, in: JavaSPEKTRUM, 03/2014
[Gha14b] Ph. Ghadir, Micro-Services in Java realisieren – Teil 1, in: JavaSPEKTRUM, 04/2014
[ROCA] `http://roca-style.org`
[SCS] Self-Contained Systems, `http://scs-architecture.org`
[Til11] Ph. Ghadir, St. Tilkov, Softwarearchitektur im Großen, Online-Themenspecial Architekturen 2010, OBJEKTspektrum
`http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2011/Architekturen/tilkov_ghadir_OS_Architekturen_11.pdf`
[Wol4] E. Wolff, Continuous Delivery – Der pragmatische Einstieg, dpunkt.verlag, 2014
[Wol15] E. Wolff, Microservices – Grundlagen flexibler Architekturen, dpunkt.verlag, 2015

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com

Java User Groups Termine

Mitte Januar bis 23. März 2016

Bitte senden Sie Ihre Termine an: susanne.herl@sigs-datacom.de

Schweiz; Infos unter http://www.jug.ch

26.01./28.01.2016 – Luzern/Bern – Hacking with Lambdas, Streams and the new Date and Time API, Michael Inden
26.01.2016 – St. Gallen – Konfiguration mit Apache Tamaya, Anatole Tresch
09.02.2016 – St. Gallen – Low overhead production time profiling and diagnostics; Marcus Hirt
17.02./18.02.2016 – Luzern/Zürich, Monadic Java, Mario Fusco

Darmstadt; Infos unter http://www.jug-da.de

18.02.2016 – Hack the website! – Ort: Deutsche Telekom AG, T-Online-Allee 1, 64295 Darmstadt

Erlangen – Nürnberg; Infos unter http://www.jug-ern.de

18.02.2016, 18.30 Uhr – Programm und Ort werden noch bekannt gegeben. Vorschläge bitte an info@jug-n.de!

Frankfurt; Infos unter http://usergroups.rheinmainrocks.de

28.01.2016, 19.00 Uhr – Independent Game Developers Rhein-Main – Ort: Sanid GmbH, Kruppstr. 105, 60388 Frankfurt

München; Infos unter http://www.jugm.de

14.03.2016 – Software Performance in DevOps – Eine Perspektive aus Forschung und Praxis mit Andreas Brunnert

Münster; Infos unter http://jug-muenster.de

27.01.2016 – Kubernetes/Docker, Marc Sluiter, 3.01.08 Konferenzraum 2
17.02.2016 – MongoDB, Tobias Trelle, 3.01.08 Konferenzraum 2

Stuttgart; Infos unter http://www.jugs.org

07.03.2016, 18.30 Uhr – Agile Nacht Stuttgart

Ostfalen; Infos unter http://www.jug-ostfalen.de

08.03.2016, 09.30 Uhr – JavaLand im Phantasialand Brühl