01-2014 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

Java hebt ab

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 01

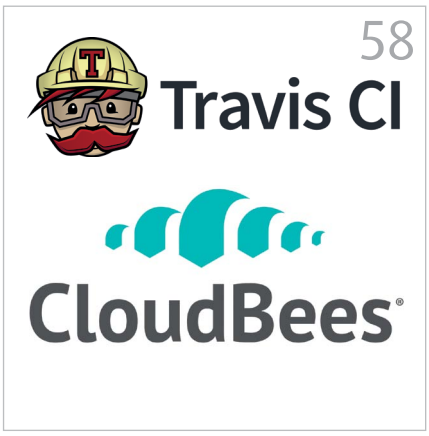
iJUG Verbund

Sonderdruck

Java macht wieder richtig Spaß:
Neuigkeiten von der JavaOne, Seite 8

Interview mit Mark Little über das Wachstum und die Komplexität von Java EE 7, Seite 30

JavaFX oder eine HTML5-basierte Technologie:
Wo und warum der Einsatz von JavaFX sinnvoll ist , Seite 44



Cloud-hosted Continuous Integration mit CloudBees und Travis CI, Seite 58

Einfach skalieren

Leon Rosenberg, anotheria solutions GmbH

Von B2C-Portalen wird vor allem eins erwartet – zu skalieren. Leider wird Skalierbarkeit zu oft zur Technologie-Frage erklärt – die Auswahl einer passenden „Hype-Technologie" würde genügen. Dass dem nicht so ist, belegt spätestens der Live-Betrieb. Anstatt die Technologie-Keule zu schwingen, zeigt der Artikel, wie sich mit intelligenter Architektur und dem Verzicht auf das Datenmodell ein hoch skalierendes, hoch performantes Portal entwickeln lässt. Neben den grundsätzlichen Konzepten sind konkrete Skalierungs-Szenarien und Lösungen dargestellt.

In den dunklen Anfangszeiten des Internets – also Ende des letzten Jahrtausends – wurde die Frage der richtigen Architektur eines Systems zur Datenbank-Wahl degradiert. Beauftragte der CEO eines Unternehmens den Bau eines neuen Portals, stellte sich das Entwicklerteam die Frage, ob man eine Oracle-Enterprise-Lizenz benötige oder ob eine einfache Lizenz ausreichend sei. Ganz wilde Kerle brachten sogar Poet oder Versant (oder andere objektorientierte Datenbanken) ins Spiel. Anschließend wurden Daten-Modelle erstellt, die in den meisten Fällen Datenbank-Modelle waren, bevor man sich überhaupt die Frage stellte, was genau das neue System leisten und wie es funktionieren soll.

Obwohl wir nun, gut zehn Jahre später, um eine ganze Reihe interessanter Entwicklungen in der Software-Entwicklung weiter sind, läuft es heute ganz ähnlich ab, nur dass es nicht um Oracle vs. Informix geht, sondern darum, ob man Mongo, Hadoop oder ElasticSearch nimmt. Zweifelsohne sind das tolle und sehr nützliche Technologien, jedoch sollte deren Wahl nicht vor der Architektur-Entscheidung stehen. Mit anderen Worten: Die Technologie sollte der Architektur dienen, indem sie bestimmte Aufgaben innerhalb dieser Architektur übernimmt. Die Aufgaben sollten allerdings von der Architektur und den Anforderungen diktiert werden.

Der „Technology first"-Ansatz, dem man häufig in der Software-Entwicklung begegnet, ist für die weniger technisch versierten Entscheider zwar attraktiv, führt jedoch selten zu der wirklich guten Lösung für das konkrete Problem. Wenn ein Startup-Unternehmen Mongo, Bootstrap, ElasticSearch, Rails etc. nutzt, kann es damit bestimmt auch seine Probleme lösen und wenn nicht, dann kann ihm keiner vorwerfen, er hätte nicht die fortschrittlichste Technologie benutzt. Der Autor vertritt hingegen den entgegengesetzten Ansatz: „Architecture first". Das bedeutet, dass in erster Linie das konkrete Problem architektonisch gelöst wird und die Technologie nur ein Mittel zur Umsetzung der Architektur ist. Das bedeutet auch, dass die Technologie nur ein Teil der Lösung ist, und zwar nur dann, wenn sie auch zur Lösung beiträgt.

Hierzu ein Beispiel: Jahrelang wurden relationale Datenbanken zur Lösung sämtlicher Probleme – auch für Internet-Portale – herangezogen und jahrelang hatte man damit Probleme, diese dann zu skalieren, sobald das Schema zu komplex wurde. Aus dieser Not heraus wurde die Nachfolge-Generation der RDBMS-Systeme, die NoSQL-Datenbank, geboren. Ob es sich hier um wirkliche Neuentwicklungen oder um die Wiederbelebung ganz alter Ideen handelt, ist hierbei irrelevant.

Der Erfolg der NoSQL-Datenbanken basiert zum Großteil darauf, dass sie die Schwäche der SQL-Datenbanken, nämlich die „Joins", erkannt haben und sie erst gar nicht unterstützen. Nur wenn man seine Architektur so baut, dass man keine „Joins" braucht, sind die SQL-Datenbanken kein Skalierungsproblem mehr.

Was eine Architektur wirklich ist

Bevor wir darüber sprechen, wie man die richtige Architektur findet, welche die üblichen nicht funktionalen Anforderungen wie Flexibilität, Skalierung und Manage-barkeit erfüllt, müssen wir uns die Frage stellen, was eine Architektur überhaupt ist. Die Meinungen darüber gehen weit auseinander. Einige sehen darin eine sehr abstrakte Art der Anforderungsbeschreibung, andere wiederum sagen, die Einteilung des Codes in Packages und das Strukturieren dieser Packages sei die Architektur.

Auf der Seite http://www.sei.cmu.edu/architecture/start/glossary/community.cfm findet man eine große Auswahl verschiedener Definitionen. Der Autor findet diese am passendsten: „Software-Architektur ist die Struktur eines Systems, die aus Komponenten, den extern sichtbaren Eigenschaften dieser Komponenten und den Beziehungen zwischen ihnen besteht."

Bei der Software-Architektur geht es also um die Komponenten eines Systems und darum, wie diese miteinander kommunizieren. Nun will man keine universelle Architektur bauen, also nicht nach dem heiligen Gral der Software-Entwicklung suchen, sondern sich um jene hochskalierbaren B2C-Portale kümmern, bei denen vor allem die Benutzer jede Menge an Zeit verbringen, sei es in Online-Shops, Vergleichsportalen oder sozialen Netzwerken in all ihren Ausprägungen. Diese Portale haben aus Sichtweise der Software-Architektur einige Gemeinsamkeiten:

- Sie haben alle eine hohe „read/write"-Ratio – die Menge der Lese-Zugriffe ist in der Regel deutlich höher als die der Schreibzugriffe (bis zu 90 Prozent)
- Sie haben mehrere klar trennbare Funktionen (wie Nachrichten-System oder Profil-Ansicht)

Abbildung 1: Services richtig schneiden

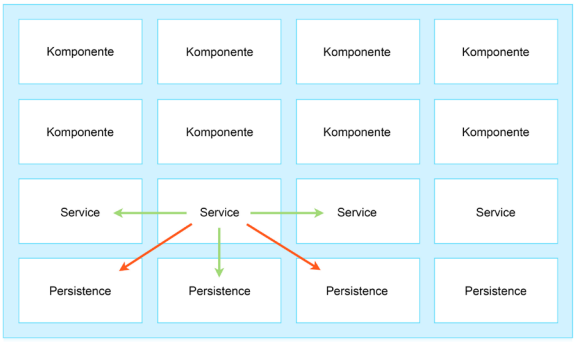


Abbildung 2: Erlaubte (grüne) und verbotene (rote) Kommunikationswege

- Sie unterliegen Peaks, die ein Vielfaches der Normal-Last zu bestimmten Tages-, Wochen- oder Jahreszeiten verursachen
- Sie verändern sich ständig und sehr schnell, sowohl was den Code als auch was den Content angeht

Grundsätzliche Architektur-Prinzipien

Eine sehr populäre Software-Architektur beim Bau solcher Portale ist die Service-orientierte Architektur (SOA). Sie ist etwas in Verruf gekommen durch die Popularität der Web-Services, einer Architektur, die mit SOA wenig zu tun hat, aber gerne mit ihr verwechselt wird. SOA ist viel älter und reifer als Web-Services und bietet – richtig angewandt – Antwort auf viele Skalierungsfragen.

Wenn wir zurück an unsere Architektur-Definition denken, sagt SOA, dass unsere Komponenten Services oder Clients sind, wobei eine konkrete Komponente sowohl das eine als auch das andere gleichzeitig sein kann, und dass die extern sichtbaren Eigenschaften dieser Komponenten die Interfaces sind, die jene Komponenten zur Verfügung stellen. Dabei gibt es zwischen den Komponenten zwei Arten von Beziehungen und folglich auch zwei Arten von Kommunikation:

- *Direkte oder synchrone Kommunikation* Methodenaufrufe, also Inanspruchnahmen eines Service durch einen Client
- *Indirekte oder asynchrone Kommunikation* Benachrichtigungen über Zustandsänderungen (Events), die eine Komponente, ohne zu wissen, ob sie Empfänger hat, überallhin ausstrahlt

Die direkte Kommunikation ähnelt einem Telefonanruf bei einem telefonischen Bestellservice, während die indirekte Kommunikation mehr an einen Börsenticker erinnert, der unabhängig davon läuft, ob ihn nun jemand liest oder nicht. Sowohl die „abfragbaren Methoden" als auch die „ablauschbaren Daten" stellen, Software-architektonisch gesehen, Interfaces dar, also Mittel, über die man mit der Komponente kommunizieren kann.

Isolation bis in die Datenbank

Ein weiteres sehr nützliches Prinzip von SOA ist die Isolation der einzelnen Komponenten voneinander. Das bedeutet unter anderem, dass eine Komponente die Alleinherrschaft über ihre Daten hat und niemand diese Daten verändern darf, ohne dass die Komponente darüber zumindest in Kenntnis gesetzt wurde. Idealerweise soll die Komponente die Datenmanipulation selbst durchführen.

Service-orientierte Architekturen haben viele Vorteile, aber der Wichtigste ist, dass sie kein globales Datenmodell besitzen. Die Interfaces einer Komponente sind alles, was über die Komponente nach außen bekannt ist. Das Innenleben jeder Komponente ist ihr selbst überlassen und nach außen hin unsichtbar.

Manchen fällt es schwer, diesen freiwilligen Verzicht auf ein Datenmodell zu akzeptieren. Mit einer schnellen SQL-Abfrage könnte man eine komplexe Recherche schnell durchführen.

Es stimmt, dass eine Verknüpfung der Daten verschiedener Services auf der Datenbank-Ebene Vorteile für Nachforschungen, Data Mining und statistische Auswertungen bietet. Aber niemand sagt, dass diese Verknüpfungen auf dem Produktionssystem existieren müssen.

Wenn man Daten für statistische Auswertungen braucht, so kann man diese Daten regelmäßig aus dem Produktionssystem überführen und dabei so viele Verknüpfungen herstellen wie nötig. Das Produktionssystem selbst sollte aber von solchen Dritt-Verwendungen freigehalten werden, die in der Regel die Performance deutlich verringern. Der Verzicht auf ein gemeinsames Datenmodell bedeutet für die Systemarchitektur Folgendes:

- Anstelle eines Daten-Modells existiert ein Service-Modell. Man könnte dazu auch Enterprise-Modell sagen, wenn dieses Wort nicht schon mehrfach für alles Mögliche herhalten musste. Das Service-Modell besteht aus den Services in dem System, den Artefakten, die sie verwalten und den Beziehungen zwischen den Artefakten.
- Jeder Service und jede Komponente sind völlig frei in der Wahl ihrer Persistenz. Der Service, der hochstrukturierte Daten hat, kann sie in einer SQL-Datenbank ablegen; derjenige, der eher Blobs verwaltet, seien es Bilder oder Texte, kann das in einer für diese Zwecke passenden Persistenz tun.
- Services werden innerhalb des Systems unterschiedlich belastet. Steigt in einer Datenbank-orientierten Zwei-Tier-Applikation immer die Last auf das Gesamtsystem an, ist bei SOA leicht zu identifizieren, wo die Last genau ansteigt. Das erlaubt es, die Last an genau dieser Stelle effizient durch Optimieren oder Skalieren zu reduzieren.

Damit aber die Vorteile einer SOA sich in vollen Zügen entfalten können, müssen

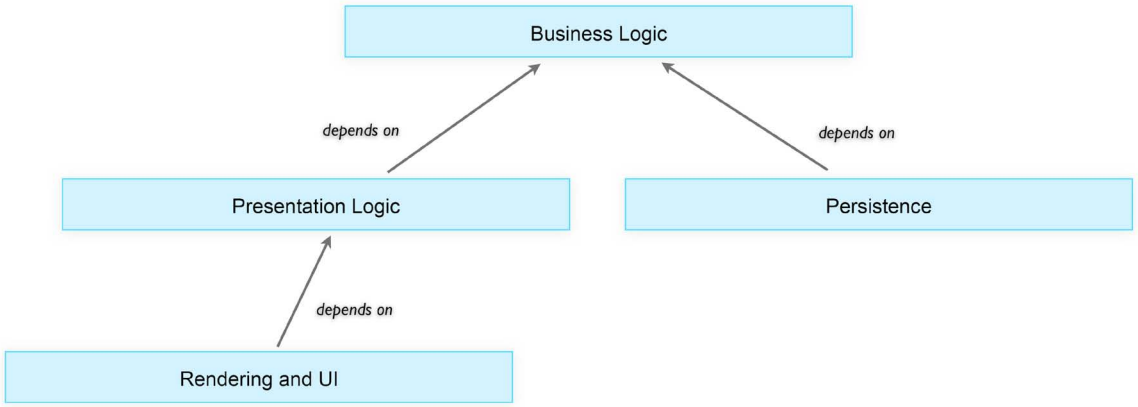


Abbildung 3: Abhängigkeiten zwischen den Layern

die Services richtig geschnitten sein. Große, monströse Services werden leicht zu Applikationen innerhalb der Applikation und zeigen all die Probleme auf, die man nicht haben wollte. Sind die Services hingegen zu klein, ist der Kommunikations-Overhead zu groß, um sinnvoll skalieren zu können. Die richtige Servicegröße findet man am einfachsten, indem man sich zwei Paradigmen der Software-Entwicklung bedient: „Design by Responsibility" und „Layer Isolation". Die Layer Isolation legt den grundsätzlichen Verantwortungsbereich des Service fest: Was ist ein Service (Business Logic) und was nicht (etwa Presentation Logic). Mithilfe des Design by Responsibility können wir eine fachliche Trennung vornehmen, die uns hilft, den Verantwortungsbereich des Service eng zu halten (siehe Abbildung 1). Wenn man die Services identifiziert hat, muss man sich Gedanken darüber machen, wie sie miteinander kommunizieren sollen und dürfen (siehe Abbildung 2).

Ein globales Daten-Modell verhindert

Das übliche Layer-Modell ist vertikal. Oben liegt die Präsentation und unten die Persistenz. Folglich beginnen die Entwickler mit der Persistenz und entwerfen ein Daten-Modell. Tatsächlich muss man bei SOA aber in der Mitte anfangen (siehe Abbildung 3).

Auf diese Weise gelingt es uns, ein wirkliches Service-Modell zu entwerfen, bei dem jede Persistenz nur die Bedürfnisse eines Service erfüllt und auf ihn eingestellt ist. Das führt zu einer strikten Isolation einzelner Persistenzen, die wir für die Skalierung brauchen (siehe Abbildung 4).

Ein weiteres Paradigma, das man unbedingt befolgen sollte, heißt „KISS", „Keep It Simple & Stupid". Auf Software-Architekturen angewandt, muss eine Architektur nur die Komponenten enthalten, die absolut notwendig sind. Alles, was keinen Mehrwert bietet, und dazu können auch die eingangs erwähnten Hype-Technologien gehören, muss wegbleiben. Mit anderen Worten: Alles, was keine Miete zahlt, muss raus, denn jede Technologie hat Wartungskosten, die nur gerechtfertigt sind, wenn die Technologie auch auf das Ergebnis einzahlt.

Gute Architektur erlaubt gezieltes Tuning

Hat man seine Services fertig entworfen und entwickelt, muss man sich die Lastfrage stellen. Oft ist es erst im Live-Betrieb wirklich klar, wie viel Last auf eine Komponente zukommt. Natürlich ist es schön, einen aussagekräftigen Lasttest zu haben, es ist aber schwer, diesen zu entwerfen und umzusetzen, ohne das reale Verhalten der Benutzer zu kennen. Doch wenn man das Benutzerverhalten kennt, ist man bereits im Live-Betrieb. Wie auch immer, es ist überhaupt nicht schlimm, wenn man das Tuning erst im wirklichen Betrieb vornimmt, denn den berühmten Satz über „premature optimization" kennt jeder.

Wichtig ist allerdings, jede Komponente zu überwachen, um die Engstellen rechtzeitig, am besten noch vor dem Überladen, zu erkennen und entsprechend reagieren zu können. Hat man eine Engstelle identifiziert, ergeben sich mehrere Reaktionsmöglichkeiten. Der Artikel geht auf zwei davon, „Caches" und „Loadbalancing", detaillierter ein.

Caches

Wie eingangs erwähnt, reden wir vor allem über B2C-Portale. Diese haben eine gemeinsame Eigenschaft, nämlich einen hohen prozentuellen Anteil von Lese-Zugriffen an der Gesamt-Zugriffszahl. Das spricht dafür, mit Caches zu arbeiten. In der Tat sind Caches ein extrem wirksames Werkzeug, das allerdings auch seine Tücken hat.

Es gibt mehrere Arten von Caches, die zwei Hauptarten sind Query- und Objekt-Caches. Query-Caches, auch „Method-Caches" genannt, sind Caches, die meistens von außen eingesetzt werden. Man speichert also Ergebnisse von Methoden (Queries) und hofft, dass gleiche Eingaben auch immer dieselben Ausgaben erzeugen. Wird die gleiche Abfrage mehrmals ausgeführt, kann man ab dem zweiten Request die komplexe und zeitaufwändige Verarbeitung auslassen und direkt das Ergebnis der vorigen Operation zurückgeben. Der Vorteil von Query-Caches ist, dass sie meistens von der konkreten Architektur und Domäne unabhängig, also als fertiges Produkt leicht integrierbar sind. Dafür haben sie aber eine ganze Reihe von Nachteilen:

- Sie sind ineffektiv bei großen Interfaces, die mehrere Methoden bieten, um an eine Information (also ein Objekt) zu kommen, denn es wird der Pfad und nicht das Ziel im Cache gehalten.
- Sie sind größenproportional zu der Menge der möglichen Anfragen und nicht der möglichen Antworten, daher schwer zu planen und übergewichtig.
- Sie sind nicht elegant.

Bei den Objekt-Caches sieht die Sache ganz anders aus. Sie sind schwerer einzubauen, dafür jedoch um einiges effektiver. Die ideale Implementierung eines Objekt-Cache ist eine Collection (ob „Map" oder „Liste", kommt auf die Anforderungen an), die sämtliche von diesem Service verwalteten Objekte in ihrer Objekt-Form enthält. Idealerweise kann jede Lese-Abfrage mithilfe des Cache vom Service beantwortet werden, ohne dass sich dieser an eine externe Persistenz (also etwa die Datenbank) wenden muss. Eine solche 100-prozentige „Cachebarkeit" ist oft nicht zu erreichen. Dort, wo sie erreicht werden kann, lohnt sie sich allerdings immer. Das beste Beispiel für einen 100-prozentigen Cache sind Benutzerdaten. Sie sind in der Regel sehr klein und daher leicht zu cachen (User-Id, E-Mail, Name, Registrierungszeit), werden jedoch permanent an vielen Stellen der Applikation gebraucht.

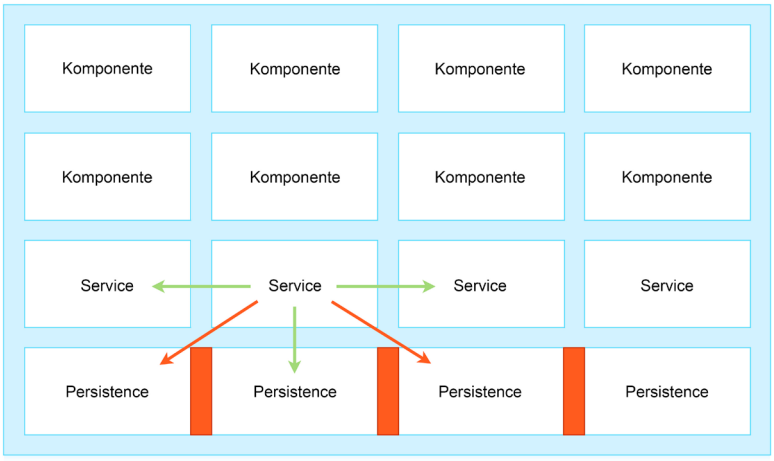


Abbildung 4: Persistenzen isolieren

Cachen, was nicht existiert

Eine Sonderform des Objekt-Cache sind die sogenannten „Null- oder Negativ-Caches". Meistens muss man einen Objekt-Cache erst zur Laufzeit befüllen, um etwa einen Kaltstart der Anwendung oder das Erstellen neuer Objekte (vor allem auch bei mehreren verteilten Instanzen desselben Service) zu ermöglichen. Das bedeutet, dass eine Anfrage nach einem bestimmten Objekt durch den Cache auf die Persistenz durchschlagen kann.

Was passiert aber, wenn der Service permanent nach nicht existierenden Objekten gefragt wird? Im schlimmsten Fall erzeugen diese Anfragen direkten Datenbank-Traffic, führen zur Überladung der Datenbank und der mühevoll aufgebaute Cache ist weitgehend ineffektiv. Negativ-Caches adressieren genau dieses Problem, indem man in ihnen vermerkt, welche Objekte man bereits erfolglos abzufragen versucht hat, und man sich das beim nächsten Request erspart.

Cachen, was sich ändert

Eine andere Unter-Kategorie von Caches sind Expiry-Caches. Dabei nimmt man an, dass ein Objekt oder seine Bestandteile über eine gewisse Zeit einen bestimmten Zustand behalten. Ein typisches Beispiel ist das Profil eines Benutzers bei einem Dating-Dienst. Ob der Benutzer „A" die Änderungen von Benutzer „B" an seinem Profil in 5, 30 oder 60 Sekunden sieht, ist für den Benutzer „A" nicht wichtig. Man kann also den Zustand des Profils von Benutzer „B" für eine – menschlich gesehen – sehr kurze, für das System aber lange Zeit einfrieren und mit dieser Objektversion arbeiten. Auf diese Weise verhindert man, dass viele Anfragen, die sehr wahrscheinlich sowieso das gleiche Ergebnis gebracht hätten (wie oft ändert jemand schon sein Profil), nicht verarbeitet werden müssen. Diese Caching-Technik ist vor allem clientseitig sehr beliebt, um Netzwerk-Traffic zu sparen.

Ein anderes Beispiel, in dem ein Expiry-Cache sinnvoll sein kann, ist folgendes: Wenn während der Verarbeitung einer Anfrage davon auszugehen ist, dass das gleiche Objekt mehrmals angefragt wird, jedoch von so unterschiedlichen Stellen im Code, dass ein Zwischenspeichern des Objekts als Variable unmöglich oder unpraktisch ist. „ThreadLocal" ist ein beliebtes Mittel, um solche Zwischenspeicher zu realisieren.

Grundsätzlich geht es bei den Expiry-Caches um sogenannte „Trade offs". Die Geschwindigkeit der Änderungssichtbarkeit wird gegen die Performance eingetauscht. Bei der Skalierung der Applikation geht es meistens darum, Trade offs zu erzeugen und Ressourcen, die im Überfluss vorhanden sind, gegen die knappen einzutauschen: „RAM vs. CPU" oder „Network bei Caches" sind weitere Beispiele.

Lokale Optimierungspotenziale sind begrenzt

Caches sind ein gutes Mittel, um die Performance einer einzelnen Komponente oder eines Service zu optimieren, aber irgendwann ist jede Optimierung am Ende. Das ist der letzte Moment, an dem man sich die Frage stellen sollte, wie man mehrere Instanzen einer Komponente betreiben kann. Anders ausgedrückt: Wie skaliert man seine Architektur. Unterschiedlich Node-Typen lassen sich unterschiedlich gut skalieren. Dabei gilt die Regel, dass, je weiter vorn und näher am Kunden eine Komponente ist, desto leichter sie skaliert werden kann.

Performance ist aber nicht der einzige Grund, um die grundsätzliche Skalierungsfähigkeit anzustreben. Die Verfügbarkeit eines Systems hängt auch zu großen Teilen davon ab, ob man in der Lage ist, Komponenten mehrfach parallel zu betreiben, sodass der Ausfall eines System-Teils problemlos kompensiert werden kann. Die Kür ist natürlich, das System elastisch betreiben zu können und die Ressourcen-Verwendung dem echten Traffic dynamisch anpassen zu können.

Die Presentation-Ebene, zu der die Web-Apps gehören, die in einem Web-Server oder Servlet-Container laufen und die für das Generieren des Mark-up (HTML, XML oder JSON), also für die Präsentation der Seite, verantwortlich sind, lässt sich in der Regel sehr leicht skalieren. Solange die einzelnen Web-Server entweder komplett „stateless" sind oder ihr State wiederherstellbar ist, also aus Caches besteht, oder ihr State ausschließlich einem User zugeordnet ist und der User immer wieder auf dem gleichen Web-Server landet, können neue Web-Server einfach dazugestellt und wieder entfernt werden. Interessanter wird

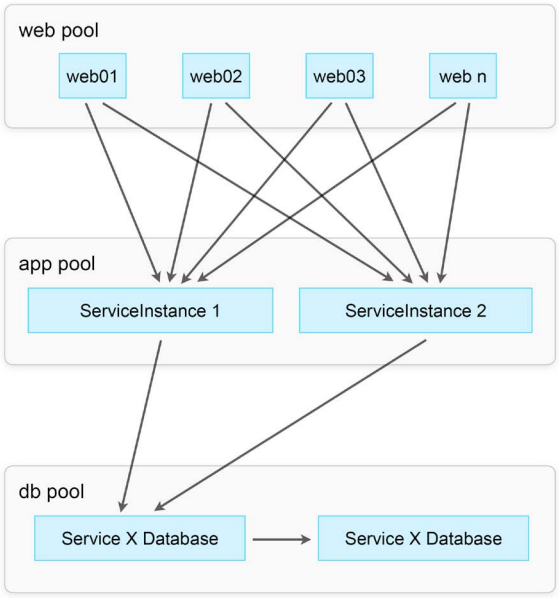


Abbildung 5: Aufrufe auf die Services nach Round-Robin verteilen

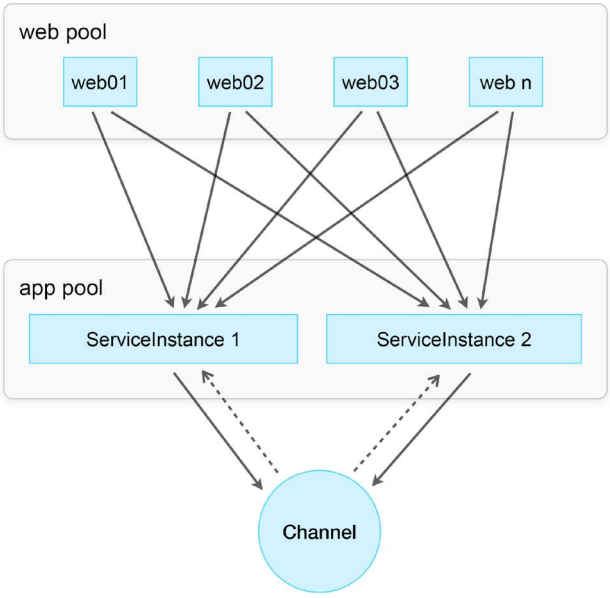


Abbildung 6: Round-Robin mit State-Synchronisation

es auf der Application-Ebene, in der die Services laufen. Bevor wir dazu übergehen, ein Blick auf die Ebene dahinter, die Datenbanken.

Die traurige Wahrheit, was das Skalieren der Datenbank betrifft, besteht darin, dass es nicht funktioniert. Mögen uns die Vertreter diverser Datenbank-Hersteller ständig vom Neuen versuchen zu überzeugen, dass ihre neuesten Features skalieren können – sie tun es nicht, wenn es darauf ankommt. Damit keine Missverständnisse aufkommen: Es gibt viele gute Gründe, eine Datenbank in einem Cluster oder einer Master/Slave-Replikation zu betreiben, aber Performance gehört nicht dazu. Der Hauptgrund, warum Applikationen so schlecht über die Datenbanken skalieren können, liegt darin, dass die Hauptaufgabe der Datenbanken darin besteht, Daten verlässlich zu speichern; Daten wieder auszulesen ist etwas, das sie viel schlechter beherrschen. Wenn wir also nicht über die Datenbank skalieren können, müssen wir dies über die Application-Ebene tun. Es gibt viele Gründe, warum das gut funktioniert:

- Dort ist das meiste Wissen über die Applikation gesammelt. Wenn man also skalieren möchte, tut man es mit dem Wissen über die Applikation und ihre Benutzung.
- Dort kann man mit den Mitteln von Programmiersprachen arbeiten, was sehr viel mächtiger ist, als mit den Mitteln, die einem auf der Datenbank-Ebene zur Verfügung stehen

Skalierungsstrategien

Zum Skalieren der Services stehen mehrere Strategien zur Verfügung. Zuerst einmal ist es wichtig zu definieren, was der Zustand eines Service, also sein State ist. Der State einer Service-Instanz ist die Menge an Informationen, die nur sie kennt und die sie von anderen Instanzen unterscheidet. Eine Instanz eines Service ist meistens eine JavaVM, in der eine Kopie des Service läuft.

Die Informationen, die den State ausmachen, sind meistens die Daten im Cache. Verwaltet ein Service keine Daten, ist er komplett „stateless". Möchte man die Last auf einen Service reduzieren, versucht man, mehrere Instanzen davon zu betreiben. Dabei gibt es verschiedene Strategien, wie man die Last auf die einzelnen Instanzen verteilt. Die einfachste heißt „Round-Robin". Dabei spricht jeder Client mit jeder Service-Instanz. Die Service-Instanzen werden abwechselnd genutzt. Das funktioniert gut, solange die Services „stateless" sind und einfach Aufgaben abarbeiten (etwa E-Mails verschicken, siehe Abbildung 5).

Sofern die Service-Instanzen States besitzen, müssen sie diese untereinander synchronisieren, zum Beispiel indem sie die Änderungen in den States über einen Event-Channel oder ein anderes Publisher/Subscriber-Modell austauschen (siehe Abbildung 6). Dabei kann jede Instanz jede Änderung, die sie auf einem Daten-Objekt vornimmt, anderen Instanzen mitteilen (gestrichelte Linie), sodass diese die Änderung auf ihren States auch durchführen können und so der gemeinsame Service State über alle Instanzen erhalten bleibt.

Diese Methode funktioniert gut bei geringem Traffic, bekommt aber Probleme, wenn die Menge der parallel durchgeführten Änderungen so groß ist, dass Konflikte durch gleichzeitiges Verändern eines Objekts auf mehreren Instanzen wahrscheinlich sind. Um den entgegenzutreten, kann man „Routing" einführen. Darunter versteht man in dem Fall, dass die Service-Instanz kontextsensitiv ausgewählt wird. Der Kontext können in diesem Fall der Client, die Operation oder die Daten sein.

Routing über die Daten oder Sharding ist das mächtigste Instrument. Darunter versteht man einen Routing-Mechanismus, der eine Service-Instanz aufgrund von Operations-Parametern, also den Daten, auswählt. Idealerweise hat man einen primären Parameter wie eine User-Id, den man leicht in eine numerische Form umwandeln kann, sodass man eine Division mit Rest durchführen kann. Dividiert man durch die Anzahl der verfügbaren Instanzen, so bekommt die erste Instanz alle Anfragen mit „rest=0", die nächste mit „rest=1" und so weiter (siehe Abbildung 7).

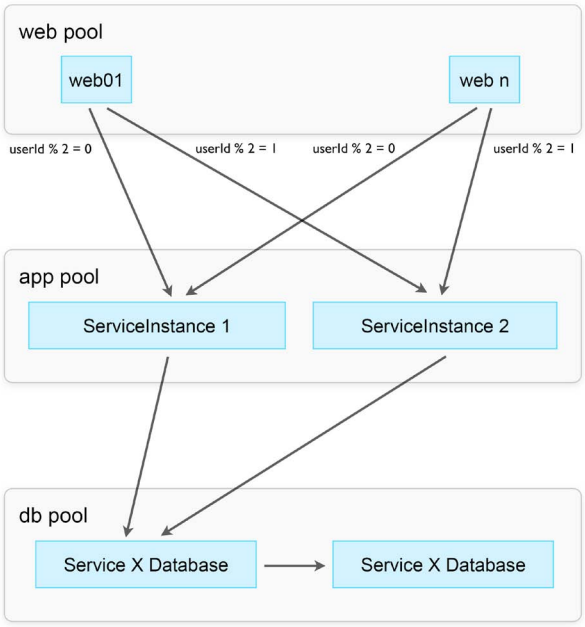


Abbildung 7: Sharding über Modulo

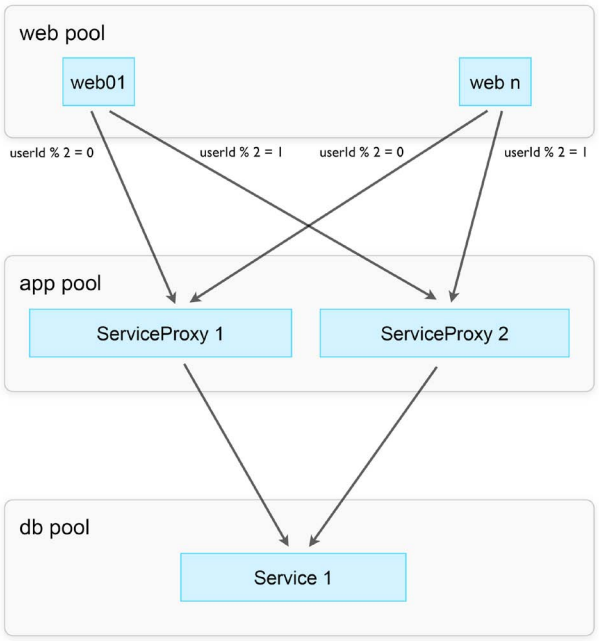


Abbildung 8: Service-Proxies

Diese Methode hat einen sehr nützlichen Nebeneffekt: Da alle Anfragen zum gleichen Benutzer immer auf derselben Instanz landen, wird eine Fragmentierung der Daten erreicht. Das bedeutet kleinere Caches, bei Bedarf gewollt fragmentierte Datenbanken und weiteres Optimierungspotenzial.

Natürlich kann man bei entsprechender Middleware die verschiedenen Skalierungsstrategien kombinieren, indem man beispielsweise die Service-Instanzen über Sharding zu Gruppen zusammenfasst, innerhalb derer man wiederum Round-Robin fährt. Diese weiterführenden Strategien zu diskutieren, würde aber den Rahmen des Artikels sprengen.

Manchmal können die Daten aber nicht „ge-sharded" werden, vor allem dann, wenn eine Operation zwei Datensätze gleichzeitig modifiziert. Das wohl klassischste Beispiel hierfür ist die Zustellung einer Nachricht von Benutzer „A" zu Benutzer „B", bei dem sowohl die Mailbox von Benutzer „A" als auch die von Benutzer „B" beschrieben ist.

Es ist unmöglich, eine Verteilung zu finden, bei der sowohl die Mailbox von Benutzer „A" als auch die von Benutzer „B" immer garantiert auf derselben Service-Instanz ist. Trotzdem gibt es auch hier eine Vielzahl von möglichen Strategien. Die wohl einfachste ist, den Service implizit aufzuteilen, idealerweise durch die Middleware und ohne dass der Client etwas davon merkt (siehe Abbildung 8).

Die Proxies sind dafür da, Lese-Operationen noch vor dem eigentlichen Master-Service zu verarbeiten, und da wir sehr viel mehr Lese- als Schreib-Operation haben, den Master-Service zu entlasten. Die Lese-Operationen finden meistens im Kontext eines Benutzers statt, können also wie gewohnt „ge-sharded" werden. Die verbleibenden Schreib-Operationen schlagen über die Proxies auf den Master durch, da aber sehr viel Last bereits in den Proxies abgearbeitet wurde, können die Schreib-Operationen weniger Schaden anrichten.

Fazit

Der Artikel zeigt, wie man die richtige Architektur findet und welche Mittel zur Verfügung stehen, um diese Architektur zu skalieren. Dabei stellt sich heraus, dass die Zeit, die man in die Wahl und die Einhaltung von Architektur-Paradigmen investiert, eine Investition ist, die sich, wenn es drauf ankommt, mehrfach auszahlt.

Slave-Proxies, Round-Robin-Service-Instanzen und Null-Caches sind Techniken, an die man sicherlich nicht gleich am Anfang der Entwicklung eines neuen Portals denkt. Es ist auch unnötig, sie auf Verdacht einzubauen; wichtig ist, sie zu kennen und eine Architektur zu haben, die den Einsatz solcher Tools erlaubt.

Nicht jedes Portal muss skalieren – aber wenn man es will, muss man es schnell können. Sich dabei ausschließlich auf Technologie zu verlassen, bedeutet, seine Handlungsfreiheit abzugeben und, wenn die Technologiewahl sich als falsch erwiesen hatte, gelegentlich die Scherben seines Systems aufsammeln zu müssen. Hat man im Gegensatz dazu auf die richtige Architektur gesetzt, Prinzipien und Paradigmen aufgestellt und sie befolgt, wird man mit Sicherheit auf jede neue Herausforderung eine Antwort finden können.

Leon Rosenberg
leon@leon-rosenberg.net
twitter: @dvayanu

Leon Rosenberg widmet sich als Software-Architekt seit mehr als zehn Jahren der Entwicklung von hochskalierenden Portalen, darunter Friendscout24, Parship und Allyouneed. Außerdem engagiert er sich bei Open-Source-Projekten im Bereich der Verteilung und des Application Management.

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG *News* und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN

0700 11 36 24 39

ODER BESTELLEN SIE ONLINE

go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr

☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:

Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.