S&N INVENT

Microservice-Referenzarchitektur

Whitepaper

Jochen Kirchner

20.07.2020

Inhaltsverzeichnis

Vorbemerkung

Microservice-Referenzarchitektur

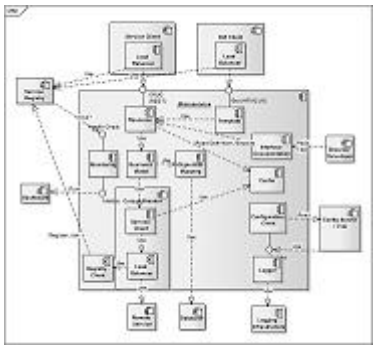

Abbildung 1: Microservice-Referenzarchitektur

Der Beitrag geht zunächst auf konzeptionelle Ansätze und einen sinnvollen Einsatz einer Microservice-Architektur ein und beschreibt anschließend eine Referenzarchitektur zur Abbildung funktionaler und nicht funktionaler Anforderungen.

1 Microservices als Architektur-Stil

Der Begriff Microservices steht für einen Architektur-Stil, der insbesondere etablierte Patterns in sich vereinigt und eine Weiterentwicklung des SOA-Architekturstils darstellt. Im Gegensatz zu SOA, das eine Wiederverwendbarkeit von Services in einem Gesamt-Unternehmen fokussiert, stehen Microservices für die Erstellung und Wiederverwendung von Services in einem Projekt, bei dem im Kontext einer fachlichen Domain (beispielsweise im fachlichen Kontext einer Abteilung oder Team) eigenständige Funktionsblöcke abgebildet werden. Der Microservice-Architektur-Stil folgt insbesondere folgenden (im Detail ggf. zu adaptierenden) Prinzipien:

- **Single-Responsibility-Principle**
  Wichtigstes Prinzip eines Microservice ist sein in sich abgeschlossener und von anderen Microservices unabhängiger Funktionsumfang – meist abgeleitet aus abgegrenzten Business Capabilities oder Business (Sub-) Domains.
- **Modularität und Minimierung von Abhängigkeiten**
  Anwendungen werden als Menge kleiner Services erstellt, die jeweils in einem separaten Prozess laufen und über leichtgewichtige Mechanismen (HTTP, REST) kommunizieren. Microservices sind grundsätzlich unabhängig von anderen Services, sie können daher einzeln ersetzt werden. Beim Design sollte allerdings darauf geachtet werden, dass keine impliziten Abhängigkeiten (Datenmodelle, Ablauflogik, Technologien, etc.) entstehen.
- **Unabhängige, agile Teams**
  Jeder Service wird (nach dem DevOps-Prinzip und Produktgedanken) jeweils von einem unabhängigen, fachlich spezialisierten Team entwickelt, ausgeliefert und in Produktion betreut. Der abgeschlossene, begrenzte Funktionsumfang ermöglicht die unabhängige Entwicklung von anderen Parteien in kleinen Teams, sowie die Skalierung von parallelen Entwicklungsteam bei mehreren Funktionsblöcken.
- **Skalierbarkeit**
  Microservices sind einzeln skalierbar. So können ressourcen-intensive Bestandteile einer Anwendung unabhängig und idealerweise automatisch skaliert werden.
- **Time-to-Market**
  Die Entwicklung und Auslieferung von Microservices bedient sich etablierter Verfahren, wie Continuous Delivery zum automatisierten Erstellen, Testen und Ausliefern von Softwarekomponenten. In Kombination mit den kleinen, schnell erstellbaren und testbaren Microservices sind sehr schnelle Auslieferungen möglich, sofern man Abhängigkeiten zu anderen Komponenten begrenzt bzw. vermeidet
- **The right tool for the job**
  Microservices können in der für sie optimalen Technologie erstellt oder bei Bedarf durch eine neue Technologie ersetzt werden. Die Verantwortung dafür liegt nicht bei einer zentralisierten Governance, sondern beim dezentral für den Microservice zuständigen Team.
- **Dezentralisierte Datenhaltung**
  Microservices bevorzugen eine unabhängige Datenhaltung, z.B. als dezentralisierte, logische Datenbank. Auch hier kann die optimale Technologie eingesetzt werden.

Änderungen der Datenstruktur können im Idealfall unabhängig von bestehenden Serviceschnittstellen umgesetzt werden. Referenzielle Integritäten zu anderen Datentöpfen werden über asynchrone Abgleichmechanismen abgebildet, um Zugriffsblockaden zu vermeiden.

- **Design for failure**
  Microservices werden immer gegen bekannte Fehlerfälle (Timeout, fehlerhaftes Verhalten, …) so abgesichert, dass jederzeit ein definiertes, nicht blockierendes Verhalten möglich ist und sie keinen Single-Point-of-Failure in einer verteilten Architektur darstellen. Microservices bedienen sich dabei verschiedener Stabilisierungs-Patterns für insbesondere Service-zu-Service Kommunikation wie Ab-„Schottung“(Bulkheads), Wiederholung (Retry-on-Failure) und Abbruch zu zeitintensiver Operationen (Circuit-Breaker), die gleichfalls zum Monitoring kritischer Pfade als auch zu Justierung im laufenden Betrieb genutzt werden können.
- **Optionales User Interface**
  Microservices enthalten optional eine Benutzerschnittstelle, die auf die begrenzte Funktionalität des Microservice ausgelegt ist. Microservices können daher sowohl als reine Backen-Services oder auch als modularer Bestandteil einer größeren Anwendung (Portal) erstellt werden. In letzterem Fall sind jedoch Rahmenbedingungen wie Styleguide und technische Einbindung in einen Anwendungsrahmen zu klären, die insbesondere zur Laufzeit der Anwendung den alleinigen Austausch des Microservices (Hot Replacement) ermöglichen. Hier sei auf gängige Standards wie Portlet Specification (JSR-168) oder Web Components verwiesen.

Microservices ermöglichen insgesamt einen vertikalen Anwendungsschnitt, indem abgegrenzte Funktionen als Gesamtpaket von Anwendungslogik, Datenhaltung und Anwendungsoberfläche erstellt, gewartet und ausgetauscht werden können.

Für den Software-Architektur-Still stellt sich die Frage, welche Vor- und Nachteile sich ergeben und wann dieser in Projekten eingesetzt werden sollte.

1.1 Vorteile von Microservices

Gegenüber monolithischen Anwendungen (gemeinsame Auslieferung in einem Paket, keine inhaltliche funktionale Trennung) ergeben sich deutlich Vorteile:

- **Geschwindigkeit:**
  Microservices sind schnell erstellbar, änderbar, deploybar (nutzen Continuous Integration Infrastrukturen) und auch ersetzbar. Damit können Anforderungen von Markt- oder Kundenseite schnell umgesetzt und nach Einsatz optimiert werden.
- **Agiles Vorgehen und flexible Teams:**
  Organisatorisch werden pro Microservice kleine, unabhängige Teams aufgesetzt, die gemäß DevOps das gesamte Spektrum von fachlichem Design, Entwicklung, Produktionseinsatz bis Monitoring übernehmen. Inhaltliche und zeitliche Abhängigkeiten zwischen Teams (z. B.: Entwicklung, Datenbank-Experten, Betrieb) sind im Idealfall nicht vorhanden.

- **Skalierung einzelner Services:**
  Im Gegensatz zur Skalierung ganzer Webanwendungen können einzelne Services mit Hilfe ihres Monitorings einzeln und automatisiert horizontal skaliert werden.
- **Das richtige Werkzeug pro Aufgabenstellung:**
  Anstatt zentraler Vorgaben kann jeder Microservice mit passender Technologie (Programmiersprache, Datenbanktyp) erstellt bzw. später ersetzt werden.
- **Risikobegrenzung im Fehlerfall:**
  Bei auftretenden Fehlern ist nur eine einzelne Microservice-Instanz betroffen, wodurch die Verfügbarkeit des gesamten Systems steigt.
  Dabei ist jedoch zu beachten, dass Services miteinander kommunizieren und auch das Netzwerk von Ausfällen oder Fehlern betroffen sein kann. Daher müssen diese Fehlersituationen durch gezielte Maßnahmen abgefangen werden.
- **Canary Deployment:**
  Beim Ausrollen neuer Microservice-Versionen kann zunächst ein Prototyp-Betrieb bzw. Produktiv-Test gegen wenige und schrittweise weitere Instanzen erfolgen. Weiterhin ist ein Parallelbetrieb mehrerer Versionen möglich, die über die Service-Registry verwaltet werden.

1.2 Risiken von Microservices

In Summe sind das gute Argumente für eine moderne, cloud-fähige und agile Software-Architektur. Aber was sollte man dabei an Risiken beachten:

- Insgesamt entstehen eine **höhere Komplexität** (Entwicklung, Deployment, Betrieb) und ein größerer Overhead für ein verteiltes System.
- Die **Stabilität/Verfügbarkeit des Netzwerks** ist nicht immer gegeben. Es sind vorbeugende Maßnahmen bei Nichtansprechbarkeit aufzurufender Services zu treffen.
- Das **Monitoring** und Nachverfolgen von (insbesondere kaskadierenden) Fehlern in verteilten Umgebungen ist ein Muss, aber komplex und zeitaufwendig.
- **Höhere Fixkosten und Komplexität im Betrieb** (höherer Anzahl an Modulen, unterschiedliche Versionen, unterschiedliche Laufzeitumgebungen für verschiedene Technologien).
- Integrationstests können aufwendig und komplex sein.

1.3 Entscheidungskriterien für Microservices

Wurde im Projekt die Entscheidung für eine agile Vorgehensweise getroffen, ist eine modulare Architektur die Voraussetzung für kleine, agile, unabhängig arbeitende Teams mit abgegrenzter Funktionalität. Dies lässt sich sowohl mit einer modularen Struktur innerhalb einer großen Anwendung, als auch mit Microservices erreichen. Sollen zudem zeit- und ressourcenintensive Regressionstest vermieden werden, was bei größeren Anwendungen unvermeidlich ist, so besteht die erweiterte Anforderung darin, diese Module technisch unabhängig ausliefern zu können, was wiederum für die Erstellung von Microservices spricht.

Ein weiterer Nachteil großer Anwendungen kann darin bestehen, dass eine gesamte Anwendung für Performance-Anforderungen optimiert werden muss, die nur einen Bruchteil der

Funktionalität betreffen. In diesem Fall sind Microservices von Vorteil, da sie sich einzeln je nach Bedarf beliebig skalieren lassen und so Kosten partiell nur nach Bedarf anfallen.

Bei der Einführung von Microservices steigt mit der Anzahl von Services und der Häufigkeit von Releases die Komplexität in Entwicklung, Release-Management und Betrieb. Die Komplexität mindern Service-Mesh-Systeme wie Istio (Kubernetes-Aufsatz). Sie ermöglichen sowohl die Konfiguration und das Monitoring der Service-zu-Service-Kommunikation als auch die Planung und Justierung eines Canary Deployments für unterschiedliche Service-Versionen und übernehmen Sicherheitsaspekte wie Authentifizierung und Autorisierung bei Service-Zugriffen.

Aus kommerzieller Sicht sind bei Microservices zunächst höhere Fixkosten zu erwarten. Allerdings können wirtschaftliche Erfolge durch Agilität und Geschwindigkeit die notwendigen Kosten rechtfertigen.

Kommt man in der Abwägung zu einem negativen Ergebnis, kann ein monolithischer Ansatz mit guter interner Modularisierung eine gute Alternative und Basis für einen späteren Übergang zu einer Microservice-Architektur sein.

2 Microservice Referenzarchitektur

Fällt die Entscheidung zugunsten einer Microservice-Architektur aus, müssen nicht-funktionale Aspekte und das Zusammenspiel in einer verteilten Anwendung detailliert konzipiert werden. Folgende Abbildung zeigt in einer Referenzarchitektur die innerhalb und außerhalb eines Microservice abzubildenden Aspekte und Funktionen.

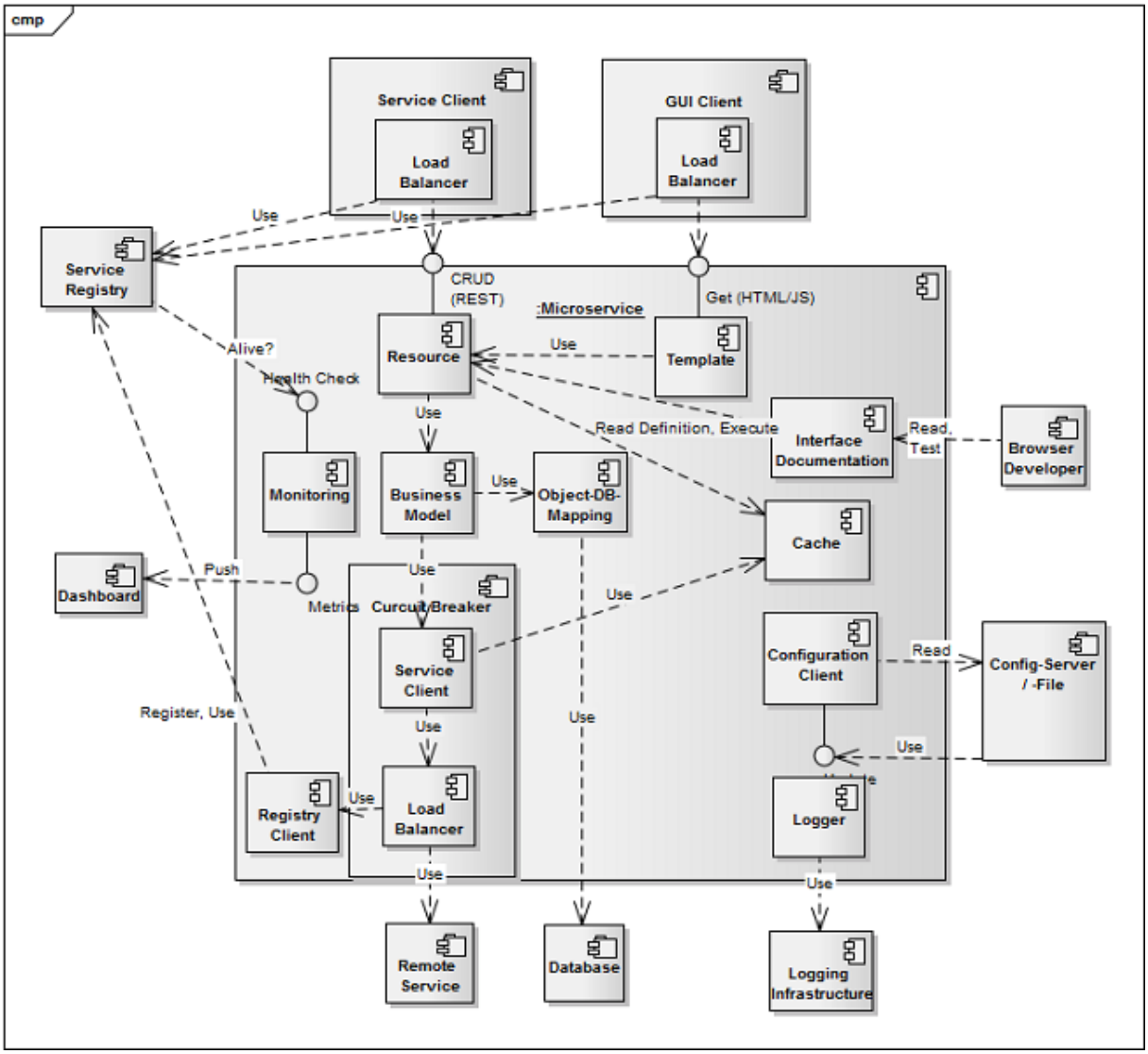

Abbildung 2: Referenzarchitektur samt Aspekten und Funktionen innerhalb und außerhalb eines Microservice

2.1 Bestandteile des Microservices

Ein Microservice besteht mindestens aus einer Komponente, die über ein leichtgewichtiges Protokoll wie HTTP oder REST (HTTP + XML/JSON) Schnittstellen zur Verfügung stellt. Optional werden zusätzlich eine Datenhaltung und/oder eine Benutzeroberfläche bereitgestellt.

2.2 Dokumentation

Zwingend erforderlich ist eine aktuelle Dokumentation der Service-Schnittstellen, optimalerweise als Weboberfläche des Webservice auslesbar und testbar. Für REST-Schnittstellen wird meist das Framework OpenAPI Specification (OAS) (vormals Swagger) verwendet, das allerdings nur die Datenstrukur und den Endpunkt pro Schnittstelle dokumentieren kann. Die Anwendungslogik (Welche Ausgangsdaten werden aufgrund welche Eingabedaten geliefert) ist zusätzlich zu dokumentieren.

2.3 Tracing und Monitoring

Für den Betrieb einer Microservice-Architektur sind Auswertungen über Verfügbarkeit, Performance und fachliche Prozesse unverzichtbar. Frameworks wie Metrics lesen über definierte Messpunkte Aufrufraten, Antwortzeiten, Fehlerraten und Timeouts und senden diese Daten in bestimmten Zeitabständen an Auswertungs-Dashboards wie Prometheus und Graphite.

Lebenswichtige Funktionen können über HealthCheck-Schnittstellen per Ja/Nein-Abfrage geprüft werden.

Über Log-Dateien können instanz-übergreifend weitere Auswertungen über Frameworks wie ELK gewonnen werden. Dazu sind Log-Inhalte und -Formate festzulegen. Frameworks wie Spring Boot bieten darüber hinaus Tracing Funktionen, die das kombinierte Logging innerhalb einer Aufrufsequenz mehrerer Services ermöglichen.

2.4 Service-Registry

Für den Aufruf der Microservice-Instanzen von Client-Anwendungen oder von anderen Services müssen die Endpunkte der aktuell verfügbaren Instanzen und Versionen über eine Service-Registry abrufbar sein. Die einzelne Service-Instanz meldet sich meist bei Start oder Stopp bei der Registry an. In bestimmten Zeitabständen ruft die Service-Registry eine HealthCheck Schnittstelle des Microservice auf und speichert im Negativfall einen Nichtverfügbarkeitsstatus. Bei dynamischer Skalierung der Microservice-Instanzen ist die Service-Registry jeweils auf dem aktuellen Stand.

2.5 Load Balancing

Im Gegensatz zu klassischen Webanwendungen werden aufgrund der dynamischen Skalierung der Microservice-Instanzen Load-Balancer mit Zugriff auf die Service-Registry eingesetzt. Es werden ggf. client-seitige Load-Balancer bevorzugt, da statt zweier Anfragen über das Netzwerk (serverseitiger Load-Balancer plus Service-Schnittstelle) nur der direkte Aufruf der Microservice-Schnittstelle notwendig ist. Bei der Vielzahl von Aufrufen (Clients zu Microservices bzw. zwischen Microservices) wird so ein erhebliches Maß an Netzwerkverkehr vermieden.

2.6 Widerstandsfähigkeit gegen Fehlersituationen

Die verteilte Architektur stellt erhöhte Anforderungen an einzelne Microservices für insbesondere Service-zu-Service-Kommunikationen. Sie dürfen nicht zum Single-Point-of-Failure werden, indem bei einem Request ein vollständiges oder teilweise blockierendes Verhalten erfolgt. Dies kann in technischer Form ein Timeout, eine zu lange Antwortzeit oder ein technischer Fehler sein. Auch kann aus fachlicher Sicht eine unpassende Antwort erfolgen, mit der der Aufrufer in seinem Kontext nicht fortfahren kann.

Aktuelle Frameworks bieten verschiedene Lösungen an, fehlerkritische Quellcode-Passagen (beim Client-Aufruf von Microservices oder bei Microservice-zu-Microservice-Aufrufen) in „Schotten“ (einzelnen Threads) einzukapseln, damit auftretende Fehler sich nicht auf die Stabilität des Microservices oder der Umgebung auswirken können. Auf Fehlerfälle und ihre Wiederholungsrate wird aktiv mit einem definierten Verhalten (z.B. Default-Daten oder alternativer Aufruf einer anderen Schnittstelle) reagiert, sodass ein Service möglichst in jeder Situation eine inhaltlich minimal verwertbare Antwort liefert.

Die Kommunikation von Frontend-Anwendungen zu Microservices und die Service-zu-Service-Kommunikation sollte möglichst asynchron erfolgen. Dies vermeidet sowohl Blockaden beim synchronen Zugriff auf gleiche Datensätze, ermöglicht einem Konsumenten aber auch die gleichzeitige parallele Anfrage an mehrere Schnittstellen. Das Eintreffen des ersten Ergebnisses kann beispielsweise als Ergebnis ausreichen, anstatt alle Anfragen sequenziell auszuführen und auf die Zwischenergebnisse zu warten. Insbesondere auf die temporäre Nichtverfügbarkeit einzelner Microservices aufgrund von Netzwerkausfällen oder Deployments wird so unkritisch für den Aufrufer reagiert.

2.7 Konfiguration

Für den Einsatz von Microservices in Entwicklungs-, Test- und Produktionsumgebungen, sowie für das Deployment mehrerer Instanzen in mehreren virtuellen Umgebungen oder Docker-Containern müssen Konfigurationsparameter sowohl beim Start gelesen, als auch während der Laufzeit aktualisiert werden. Ein Teil der Frameworks bietet dazu zentralisierte Lösungen wie beispielsweise Spring Cloud Config.

2.8 Caching

Ebenfalls zur Reduzierung von Anfragen über das Netzwerk und zur Optimierung von Antwortzeiten können an mehreren Architekturpunkten Anfrageergebnisse gecacht werden, wiebei eingehenden Anfragen eines Clients oder bei Weiterleitung des Aufrufs an andere Webservices. Je nach eingesetztem Framework können beispielsweise Größe des Caches oder zeitliche Aktualität der Daten vorkonfiguriert bzw. zur Laufzeit zurückgesetzt werden.

Autor

Jochen Kirchner ist IT-Berater bei der S&N Invent GmbH. Er studierte als Dipl.Ing. Elektrotechnik in Friedberg und als Wirtschaftsingenieur in Mainz. Zunächst war er mehrere Jahre als Softwareentwickler für Desktop und Webanwendungen tätig erst Visual Basic/ASP, seit 1998 Java/Java Script. Seit 1998 arbeitet er als technischer Projektleiter und Software-Architekt für Banken, Sparkassen Versicherungen und Luftfahrtunternehmen. In seiner verbleibenden Zeit beschäftigt er sich vorrangig mit der Entwicklung in neuen Technologien.

Bei S&N Invent leitet er seit 1994 das Kompetenzmanagement, welches Know-How in neuen Technologien, Methodiken und fachlichen Themen für zukünftige Projekte erarbeitet und den Projektmitarbeitern vermittelt.

Seine Freizeit verbringt er mit seiner Familie im nördlichen Frankfurter Umland und widmet sich kreativen Hobbies wie Modellbau, Fotografie und Musik.

Über S&N Invent

S&N Invent ist ein bundesweit tätiges IT-Unternehmen mit einem umfassenden Leistungsportfolio über die gesamte Wertschöpfungskette des IT-Lifecycles. Wir entwickeln gemeinsam mit unseren Kunden Lösungen, setzen Projekte um und schaffen damit digitale Mehrwerte. Das Leistungsspektrum reicht von klassischen Mainframe-Architekturen über moderne Java/JEE Web- und Portalarchitekturen bis hin zu neuesten Technologien im Cloud- und Mobile-Bereich. Agile Projekte mit hohem Qualitätsanspruch, gewährleistet durch modernes Continuous Quality Management, sind für uns in zahlreichen Projekten gelebter Standard. Die S&N Invent GmbH ist ein Unternehmen der S&N Group. Insgesamt sind in den Gesellschaften der S&N Group ca. 380 feste Mitarbeiterinnen und Mitarbeiter an acht Standorten in Deutschland sowie dem Nearshore-Standort Budapest beschäftigt. Damit können wir unsere Kunden mit umfassender Kompetenz und regionaler Nähe bestens betreuen. Gleichzeitig sind wir so in der Lage, große und komplexe Projekte in Time und Budget erfolgreich umsetzen zu können.

Abbildungsverzeichnis