🏗️ Microservice Platform — Beitrag in Unternehmensumgebungen

Diese Dokumentation fasst eine Auswahl meiner technischen Beiträge innerhalb einer umfangreichen Microservice-Plattform zusammen. Der Fokus liegt auf konkreten Architektur- und Engineering-Arbeiten, die sowohl **Plattformstabilität**, **Entwicklerproduktivität** als auch die **Betriebszuverlässigkeit** nachhaltig verbessert haben.

> Hinweis:
> Die ausgewählten Beispiele sind bewusst so gestaltet, dass sie **keine geschäftlichen, vertraulichen oder internen Details** preisgeben. Sie zeigen ausschließlich meine methodische, analytische und technische Arbeitsweise sowie meinen strukturierten Ansatz zur Problemlösung in komplexen Plattformumgebungen.

Schwerpunkte dieser Zusammenstellung:

1. **Service-Governance-Plattform** – Schaffung einer zentralen, einheitlichen Steuerungsoberfläche für Services und Infrastruktur
2. **Service-zu-Service-Authentifizierung** – zuverlässige Authentifizierung und Autorisierung ohne zusätzliche Infrastrukturkomponenten
3. **Gateway-Performanceanalyse** – deutliche Stabilitäts- und Durchsatzsteigerung durch nachvollziehbare Performanceanalyse
4. **TCP-„Connection reset by peer"-Analyse** – Beseitigung eines langjährigen Infrastrukturproblems durch präzise Netzwerkdiagnose

1. Service-Governance-Plattform

⭐ Kurzüberblick

- **Problem**: Werkzeuge und Informationen lagen verteilt vor – es fehlte eine zentrale, konsistente Steuerungsoberfläche.
- **Beitrag**: Aufbau einer **rollenbasierten Governance-Konsole**, die **Monitoring**, **Konfiguration** und **Plattform-Infos** in einer UI bündelt.
- **Wirkung**: Vereinheitlichte Developer-Workflows, schnellere Diagnosepfade und deutlich weniger Abstimmungsaufwand.

🧭 Hintergrund

Mit dem Wachstum der Microservice-Landschaft entstand die Notwendigkeit einer zentralen, produktisierten Governance-Plattform, die alle entwicklungsrelevanten Plattformfunktionen in einer Benutzeroberfläche bündelt und so Transparenz, Standardisierung und Steuerbarkeit der gesamten Architektur verbessert.

Sie sollte insbesondere die verstreuten Werkzeuge und Informationen zusammenführen und dadurch die Entwicklungs- und Diagnoseprozesse deutlich vereinfachen.

🛠️ Technische Umsetzung

Die technische Umsetzung folgte einem modularen Ansatz. Die Plattform integrierte bestehende Unternehmenskomponenten wie der **Identity-Provider**, das **API-Gateway**, die **Konfigurationsverwaltung** sowie die Oberflächen der **Messaging-** und **Storage-Services**.

Die wichtigsten Funktionen:

- **Zentrale UI** für alle Services und Instanzen – **rollenbasiert** gefiltert
- **Monitoring** auf Service- und Instanzebene (**OS**, **JVM**, **Applikationsmetriken**)
- **Konfigurationsmanagement** (z. B. dynamische Log-Level & Service-Konfigurationsdateien)
- **Service-Authentifizierung** und **-Autorisierung**
- Operationen für integrierte **Cache-**, **Messaging-** und **Storage-Komponenten**
- **Einheitliche Oberfläche** zur Unterstützung aller entwicklungsrelevanten Prozesse

Die Plattform wurde als **modulare Webanwendung** mit klar getrennten Verantwortlichkeiten umgesetzt. Frontend und Backend bildeten eigenständige Schichten, während das Backend über eine standardisierte REST-API funktionierte.

Die technische Umsetzung folgte drei übergeordneten Prinzipien:

- **Integration in den bestehenden Microservice-Verbund**:
  Die Plattform wurde als regulärer Service betrieben und nutzte sämtliche vorhandenen Plattformmechanismen.
- **Konsistente Nutzung vorhandener Plattformdienste**:
  Persistenz, Caching und Messaging wurden vollständig über bestehende Unternehmenskomponenten realisiert, um Redundanzen zu vermeiden und die Entwicklungsgeschwindigkeit zu erhöhen.

- **Kommunikation über etablierte Schnittstellen und Events**:
  Die Interaktion erfolgte ausschließlich über REST-APIs und etablierte Event-Mechanismen. Dadurch blieb die Architektur modular, erweiterbar und unabhängig von konkreten Infrastrukturdetails.

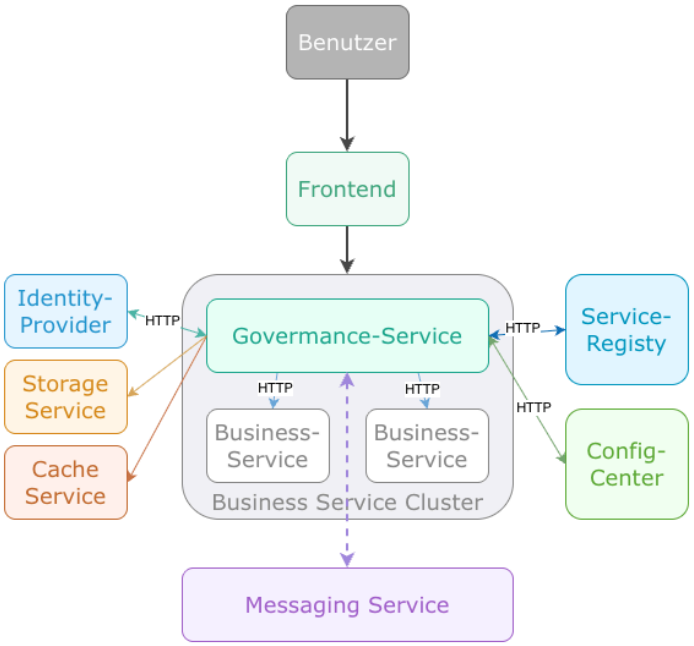

Abbildung 1: Abstrahierte Architektur der Governance-Plattform-Architektur

Ergebnisse

- **Zentrale Sichtbarkeit** über alle Services, Instanzen und Plattformkomponenten
- **Reduzierter Abstimmungs- und Betriebsaufwand** durch konsolidierte Informationen
- **Verbesserte Steuerbarkeit** durch rollenbasierte Kontrollen und standardisierte Konfiguration
- **Höhere Plattformstabilität** durch klare, einheitliche Governance-Mechanismen
- **Schnellere Entscheidungen**, da technische und organisatorische Daten in einer Oberfläche zusammenlaufen

2. Service-zu-Service-Authentifizierung

Kurzüberblick

- **Problem**: Einige interne Dienste benötigten fein granular kontrollierte Zugriffsregeln – selbst bei Aufrufen aus dem internen Netzwerk. Gleichzeitig mussten Sicherheitsmechanismen wie Regeländerungen oder Schlüsselrotation unterbrechungsfrei funktionieren.
- **Beitrag**: Entwicklung eines leichtgewichtigen, hochperformanten Authentifizierungs- und Autorisierungsmoduls für Service-zu-Service-Kommunikation.
- **Wirkung**: Klare, nachvollziehbare und erweiterbare Sicherheitsmechanismen innerhalb der Service-Landschaft – inklusive dynamischer Konfiguration und unterbrechungsfreier Schlüsselrotation.

Hintergrund

Innerhalb der Microservice-Landschaft gab es Dienste, deren Schnittstellen trotz interner Kommunikation besonders geschützt werden mussten. Dabei zeigte sich:

- Gateway-Authentifizierung deckt nicht alle **internen Aufrufwege** ab.
- Verschiedene Dienste benötigten **unterschiedliche Sicherheitsregeln**.
- Sicherheit durfte **keine spürbare Latenz** verursachen.
- Regeln und Schlüssel mussten **ohne Neustarts** aktualisiert werden können.
- Anfragen durften weder beim erstmaligen Einsatz noch während der Rotation von Schlüsseln oder Regeln unterbrochen werden.

Gleichzeitig sollte das Sicherheitsmodell vollständig in die Service-Governance-Plattform integriert werden, damit:

- Richtlinien, Ausnahmen und Standardwerte **zentral verwaltet** werden konnten
- Deployments und Regelanpassungen transparent **nachvollziehbar** waren
- und Dienste **automatisch** die jeweils gültigen Konfigurationen erhielten

Ziel war ein konsistentes, erweiterbares Sicherheitsmodell, das sich nahtlos in alle Services einfügt – ohne zusätzliche Infrastruktur und ohne die Komplexität schwergewichtiger Frameworks.

🛠️ Technische Umsetzung

Um eine robuste und zugleich leichtgewichtige Lösung zu erreichen, orientierte sich das Design konzeptionell an **etablierten Sicherheits-Frameworks** wie **Spring Security**. Dabei wurden jedoch nur die für die Plattform relevanten Prinzipien übernommen – insbesondere das **Filter-Chain-Modell** und das **Voter-basierte** Entscheidungsverfahren.

Die Implementierung wurde bewusst vereinfacht, um:

- die Latenz im Hochlastbetrieb minimal zu halten
- nur tatsächlich benötigte Authentifizierungs- und Autorisierungsregeln abzubilden
- Erweiterbarkeit über Plugins zu ermöglichen,
- und die Integration in bestehende Services ohne zusätzlichen Mechanismen.

Durch diese gezielte Reduktion entstand ein Mechanismus, der sowohl performant als auch plattformkompatibel blieb, ohne die Komplexität vollständiger Sicherheitsframeworks mitzuführen.

🔐 Trennung von Authentifizierung und Autorisierung

Die Sicherheitslogik wurde in zwei vollständig getrennte Verantwortungsbereiche aufgeteilt:

- Authentifizierung – Identifikation des aufrufenden Dienstes
- Autorisierung – Prüfung, ob der identifizierte Dienst die konkrete Operation ausführen darf

Diese Trennung erhöhte Transparenz, Testbarkeit und Erweiterbarkeit des Moduls.

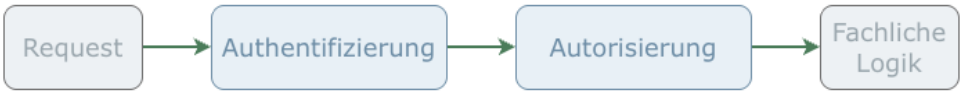

Abbildung 2: Abstrahierte Darstellung der Trennung

🧩 Kettenbasiertes Verantwortungsmodell

Sowohl Authentifizierung als auch Autorisierung bestanden aus **Verantwortlichkeitsketten („Chains“)**. Jeder Schritt war über Interfaces angebunden und konnte einzeln erweitert oder ersetzt werden.

- Neue Prüfmethoden konnten als zusätzliche **Chain-Elemente** eingebunden werden.
- Nicht benötigte Regeln wurden einfach nicht registriert.
- Die Reihenfolge der Kettenglieder war klar definiert und deterministisch.

⚖️ Entscheidungslogik mit Voter-Modell

Für die Autorisierung wurde ein schlankes, erweiterbares Voter-Modell eingesetzt. Jeder Voter bewertete eine bestimmte Regel, und eine übergeordnete Entscheidungslogik aggregierte die Einzelergebnisse zu einem finalen Zugriffsurteil.

Dieses Modell ermöglichte:

- klare Trennung einzelner Prüfregeln
- flexible Erweiterbarkeit
- eine nachvollziehbare, stabil aggregierte Entscheidung

🔄 Unterbrechungsfreie Schlüssel- und Regelrotation

Eine zentrale Anforderung war die Fähigkeit, Sicherheitskonfigurationen wie Schlüssel oder Regeldefinitionen während des Betriebs zu aktualisieren, ohne laufende Anfragen zu beeinträchtigen.

Das Modul implementierte dafür ein versioniertes, kompatibles Aktualisierungsmodell, bei dem:

- neue Konfigurationen sofort aktivierbar waren
- bestehende Versionen für eine kurze Übergangszeit parallel gültig blieben
- veraltete Versionen anschließend manuell oder automatisch ausliefen

Der Ansatz orientierte sich an etablierten Best Practices moderner Cloud-Plattformen und stellte sicher:

- kontinuierliche Verfügbarkeit
- kontrollierte und nachvollziehbare Einführung neuer Sicherheitskonfigurationen
- keine Unterbrechung laufender Requests während der Rotation

Die zentrale Verwaltung über die Service-Governance-Plattform erlaubte zusätzlich:

- einheitliche Konfiguration für alle Dienste
- Auditierbarkeit von Änderungen
- versionsbasiertes Ausrollen gültiger Regeln an alle betroffenen Clients

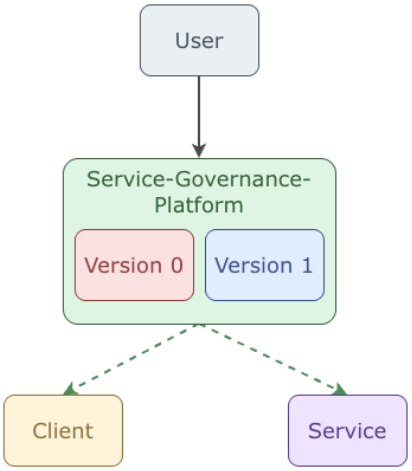

Abbildung 3: Abstrahierte Darstellung der versionsbasierten Konfigurationsrotation

🔌 Erweiterbarkeit & Integration

- **Plugin-Architektur**:
  Authentifizierungs-, Autorisierungslogik sowie einzelne Voter-Regeln konnten als optionale Plugins ergänzt werden.
- **Tiefe Integration in die Governance-Plattform**:
  Die Verwaltung erfolgte zentral über die Service-Governance-Plattform.
- **Konfigurierbarkeit pro Dienst**:
  Jeder Dienst konnte eigene Regeln übernehmen.
- **Framework-neutraler Ansatz**:
  Die Komponente war zwar im Java-Ökosystem implementiert, jedoch nicht an Spring gebunden.

📈 Ergebnisse

- **Signifikant erhöhte Sicherheit** in Service-zu-Service-Kommunikation
- **Feingranulare Autorisierung** bis auf API-Ebene
- **Sehr niedrige Latenz**, da alle Checks hochoptimiert und modular waren
- **Unterbrechungsfreie Schlüssel-Updates**, vergleichbar mit modernen Cloud-Plattformen
- **Einfache Erweiterbarkeit**, da alle Regeln über Plugins ergänzt werden konnten
- **Konsistente Plattformmechanik**, vollständig integriert in die Governance-Plattform

🚀 3. Performanceeinbruch im Gateway – Analyse und Optimierung

🧭 Hintergrund

Nach der Anbindung eines neuen Nutzungskanals traten am zentralen API-Gateway deutliche Performanceeinbrüche auf: Der Durchsatz sank, einzelne Requests blockierten länger als erwartet und die Latenzverteilung wurde zunehmend instabil. Da der Bereich geschäftskritisch war, musste die Ursache schnell eingegrenzt werden.

🔍 Analyse

Ich führte eine **systematische Analyse** der Gateway-Metriken und Trace-Daten durch. Dabei fiel ein Kanal mit außergewöhnlich vielen blockierenden Aufrufen auf. Die Untersuchung der relevanten **Codepfade** und **Prozessabläufe** deutete darauf hin, dass sich ein möglicher Engpass im Kommunikationspfad zwischen dem Gateway und dem Authentifizierungsdienst befunden haben könnte.

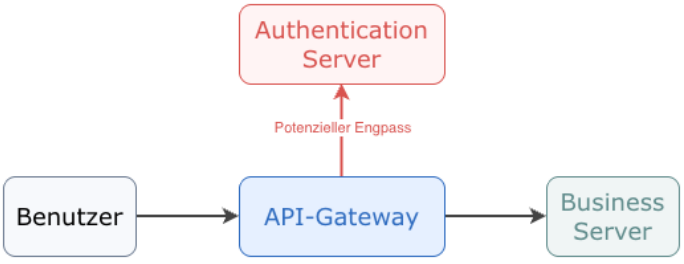

Abbildung 4: Potenzieller Engpass

🔧 Maßnahmen

Um diese Beobachtungen reproduzierbar zu validieren, entwickelte ich eine **kompakte Testumgebung auf Komponentenebene**. Sie kombinierte ein **kontrollierbares Lastmodell**, **klar definierte Testketten** und **reproduzierbare Messpunkte** und bildete die **Grundlage** für eine strukturierte Analyse des kritischen Request-Pfads.

Für die technische Analyse setzte ich einen kombinierten Werkzeug- und Methodenstapel ein:

- **wrk** für realitätsnahe Last- und Request-Simulation
- **Docker** zur standardisierten Bereitstellung einer Testumgebung
- **JProfiler** zur Identifikation blockierender oder ineffizienter Abschnitte
- **JMH** für Microbenchmarks einzelner Codepfade
- **Confluence** zur sauberen Dokumentation aller Messpunkte, Testketten und Analyseergebnisse

Die Untersuchungen zeigten mehrere Ursachen, die gemeinsam zur Performanceverschlechterung beitrugen:

- **Unnötige** oder **blockierende Synchronisation**, die unter Last blockierende Effekte erzeugte
- **Synchrone Log-Ausgaben**, die IO-bedingte Verzögerungen verursachten
- **Ineffiziente UUID-Generierung unter Java 8**, die bekannte Entropie- und Performanceprobleme aufweist
- **Wachsende Routing-Tabellen** im Gateway erhöhten die Lookup-Kosten

Auf Basis der Analyse leitete ich die notwendigen Maßnahmen ein. Ich übernahm die **technische Federführung**: identifizierte die kritischen Stellen, definierte die Lösungsansätze und nahm selbst einen Teil der **Code- und Konfigurationsanpassungen** vor. Weitere Änderungen wurden von Teamkollegen umgesetzt, wobei ich die technische Richtung, Priorisierung und Verifikation koordinierte.

Die Wirksamkeit aller Anpassungen überprüfte ich in **mehreren Lasttest-Iterationen** und stimmte die Ergebnisse teamübergreifend ab.

📈 Ergebnisse

- Rund **200 %** höherer Durchsatz und **deutlich stabilere Latenzverteilung**
- **Etablierung** eines reproduzierbaren **Analyse- und Testverfahrens** für zukünftige Gateway-Untersuchungen
- Veröffentlichung eines **technischen Leitfadens**, der teamübergreifend zur schnelleren Identifikation ähnlicher Engpässe beiträgt

Als Weiterentwicklung dieser Arbeit entstanden zwei **Open-Source-Projekte**:

- **gateway-bottleneck-lab** – offiziell veröffentlichte Open-Source-Version meiner Gateway-Testmethodik
- **uuid-benchmark** – Analyse verschiedener UUID-Implementierungen und deren Performanceverhalten

📎 **Weiterführend: Gateway-Bottleneck-Lab** (Open Source)

🌐 4. „Connection reset by peer“ (java.io.IOException) – Analyse eines systematischen TCP-Problems

🧭 Hintergrund

Die Ausnahme `java.io.IOException: Connection reset by peer` trat bereits auf, bevor ich dem Team beitrat. Auf den Grafana-Dashboards war deutlich zu erkennen, dass zahlreiche Dienste bzw. deren Client-Komponenten diese Ausnahme regelmäßig auswarfen. Das Muster war auffällig:

- gehäufte Ausnahmen etwa alle **30 Minuten**
- deutliche Zunahme nach **Leerlaufphasen**

Dank Retry-Mechanismen entstand zwar keine Geschäftsunterbrechung, dennoch stellte das Verhalten ein operatives **Risiko** dar und führte zu erheblichem **Log-Rauschen** im Monitoring.

Da ich durch meine technische Ausbildung mit Netzwerk- und TCP/IP-Mechanismen vertraut bin, bat mich mein Teamleiter um eine strukturierte Analyse.

🔍 Analyse

Die Fehlermeldung deutete bereits auf eine **serverseitig abgebrochene TCP-Verbindung** hin. Zwei Beobachtungen unterstützten diese Annahme:

- Anstieg nach Inaktivität → typisch für **Idle-Timeouts**
- Einsatz von **Connection Pools** → RST beim Wiederverwenden eines „toten“ Sockets ist ein bekanntes Muster

Aus meiner Sicht lag die Ursache vermutlich im Infrastruktur-Layer, insbesondere bei Idle-Eviction-Mechanismen eines Load Balancers. Das Ops-Team war jedoch zunächst zurückhaltend, da:

- C#-basierte Services dieses Verhalten nie gezeigt hatten
- Java-Services erst später migrierten und daher im Verdacht standen
- „Connection reset by peer“ häufig fälschlich als Anwendungsfehler interpretiert wird

Fehlerhafte Client-Implementierungen schloss ich nicht vollständig aus — aber wenn der Server eine Verbindung ohne **FIN** beendet, führt das zwangsläufig zu einem **RST**. Damit war klar: **Ich brauchte Beweise auf TCP-Ebene**.

🔧 Maßnahmen

Mit gezielten Paketmitschnitten in Wireshark analysierte ich den Datenverkehr. Die Ergebnisse bestätigten die Vermutung eindeutig:

> Nach rund 30 Minuten Inaktivität wurden Verbindungen im Connection Pool serverseitig beendet, jedoch ohne das übliche FIN-Handshake. Der Client interpretierte dies korrekt als „Connection reset by peer“.

Damit ließ sich dem Ops-Team eine **klare**, **technisch fundierte Argumentation** vorlegen:

- Serverseitiges Beenden der Verbindung **ohne FIN**
- Verhalten eindeutig passend zu einer **Idle-Eviction-Konfiguration** eines Load Balancers

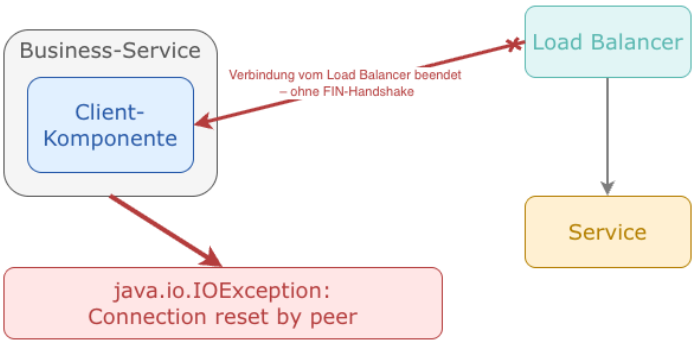

Abbildung 5: Serverseitiger Verbindungsabbruch ohne FIN-Handshake führt zur Ausnahme

Die Load-Balancer-Konfiguration wurde daraufhin überprüft und die Ursache bestätigt: Eine fehlerhafte **Eviction-Konfiguration** führte zu systematischen Verbindungsabbrüchen.

Nach Korrektur der Einstellung trat das Problem nicht mehr auf.

📈 Ergebnisse

- Diese **seit Monaten auftretenden** Ausnahmen `java.io.IOException: Connection reset by peer` verschwanden vollständig
- Das Risiko versteckter Infrastrukturprobleme wurde nachhaltig reduziert und das störende Log-Rauschen auf den Monitoring-Dashboards vollständig beseitigt.
- Die Zusammenarbeit mit dem Ops-Team verbesserte sich spürbar durch das entstandene gemeinsame technische Verständnis