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.

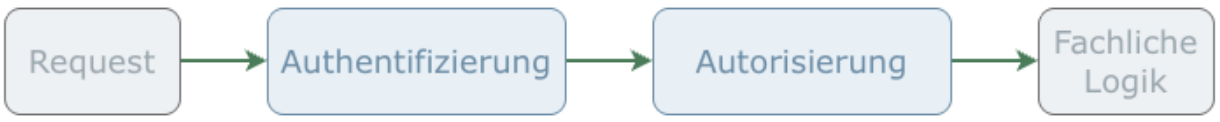

Abbildung 1: Abstrahierte Darstellung der Trennung

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

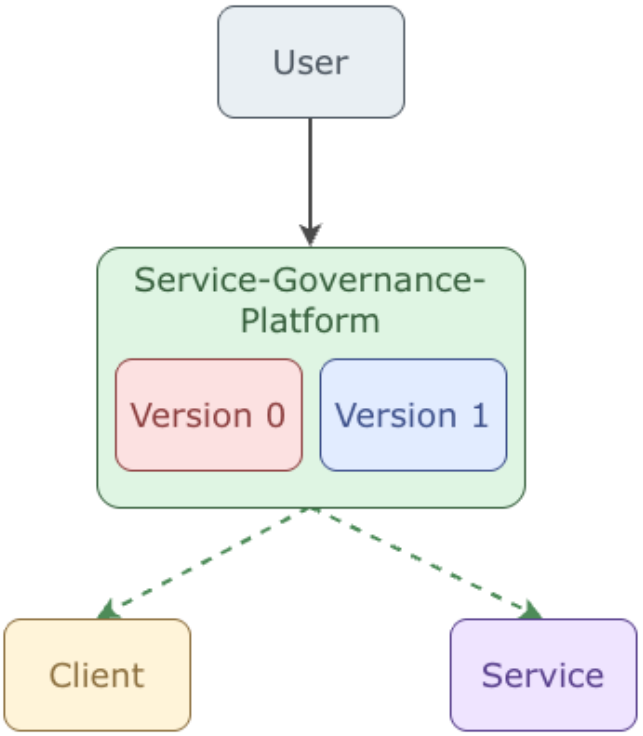

Abbildung 2: Abstrahierte Darstellung der versionsbasierten Konfigurationsrotation

🔌 **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