Bericht zur XmasWishes

Narek Grigoryan

Matrikelnummer: 1547616

Kurs: Software Achitecture for Enterprises (SA4E) – Übung 2

## Table of Contents

## Aufgabe 1: Architekturentwurf für XmasWishes

### Anforderungen und Herausforderungen

Um das System XmasWishes effizient zu gestalten, berücksichtigt dieser Architekturentwurf folgende Anforderungen:

1. **Geographisch verteilte Nutzerbasis**: Das System muss unterschiedliche Lastspitzen je nach Zeitzonen und regionalen Feiertagen bewältigen können.
2. **Hohe Skalierbarkeit**: Es muss in der Lage sein, plötzliche Anstiege im Datenverkehr, besonders in der Weihnachtszeit, zu bewältigen.
3. **Zuverlässigkeit und Verfügbarkeit**: Ein kontinuierlicher und ausfallsicherer Betrieb ist essenziell.
4. **Einfache Datenstruktur**: Die Speicherung der Wünsche und des Bearbeitungsstatus erfolgt effizient in Form von einfachen Zeichenketten.

### Architekturansatz

Das System basiert auf einem modularen Ansatz, der zentrale und regionale Komponenten umfasst. Diese Struktur ermöglicht Flexibilität, Skalierbarkeit und die Handhabung von geographisch verteilten Datenmengen.

### Hauptkomponenten des Systems

1. **Zentrales Koordinationsmodul**:
    - Verarbeitet eingehende Daten global und verteilt diese an regionale Module.
    - Speichert die Statusinformationen für jeden Wunsch (z. B. "noch nicht formuliert", "in Bearbeitung", "in der Auslieferung", "unter dem Weihnachtsbaum").
2. **Regionale Datenverarbeitungsmodule**:
    - Optimiert für die Verarbeitung lokaler Wünsche je nach Zeitzone.
    - Synchronisiert sich regelmäßig mit dem zentralen Modul.
3. **Kommunikationsschnittstelle**:
    - Ermöglicht die Erfassung von Wünschen durch Kinder und Erwachsene.
    - Unterstützt sowohl digitale Eingaben als auch die Integration gescannter Papierwünsche.
4. **Statusverwaltung**:
    - Dokumentiert, in welchem Bearbeitungsschritt sich ein Wunsch befindet.

### Datenmanagement

- **Dezentrale Speicherung**: Die Wünsche werden regional gespeichert, um eine schnelle Verarbeitung und geringe Latenzzeiten zu gewährleisten.
- **Synchronisation mit zentralem Modul**: Die regionalen Datenbanken synchronisieren periodisch ihre Daten mit dem zentralen Koordinationsmodul, um einen globalen Überblick zu ermöglichen.

- **Einfachheit der Datenstruktur**: Die Speicherung erfolgt in Form von Zeichenketten, um den Ressourcenverbrauch zu minimieren.

### Kommunikation zwischen Modulen

- **Ereignisgesteuerte Nachrichtenverarbeitung**: Module kommunizieren untereinander durch den Austausch von Ereignissen (z. B. neue Wünsche, Änderungen des Bearbeitungsstatus).
- **Asynchrone Verarbeitung**: Die Architektur entkoppelt die Module, um eine flexible und robuste Kommunikation zu gewährleisten.

### Skalierbarkeit und Zuverlässigkeit

- **Modulare Erweiterbarkeit**: Zusätzliche regionale Module können bei steigendem Datenaufkommen integriert werden.
- **Fehlertoleranz**: Lokale Module arbeiten so, dass der Ausfall eines Moduls nicht das gesamte System beeinträchtigt.
- **Lastverteilung**: Eingehende Daten werden basierend auf geographischen Parametern an die zuständigen Module weitergeleitet.

### Monitoring und Datenschutz

- **Überwachung**: Ein dediziertes Monitoring-Modul überwacht die Performance und den Status der Module.
- **Datenschutz**: Es werden keine personenbezogenen Daten außerhalb der minimal notwendigen Informationen gespeichert. Der Name wird nur temporär genutzt und nach Verarbeitung entfernt.

### Begründung des Entwurfs

*Dieser Ansatz gewährleistet, dass das System sowohl lokal optimiert arbeiten als auch global synchronisiert werden kann. Dies minimiert Latenzen und erhöht die Effizienz. Die einfache Datenstruktur und die ereignisgesteuerte Kommunikation reduzieren die Komplexität des Systems und sorgen für Robustheit. Mit diesem Entwurf ist das System optimal auf die Anforderungen einer global verteilten und hoch skalierbaren Infrastruktur vorbereitet.*

## Aufgabe 2: Konkretisierung der Technologien für XmasWishes

### Zielsetzung

Basierend auf dem in Aufgabe 1 beschriebenen Architekturentwurf werden hier die geeigneten Softwaretechnologien identifiziert, um die Anforderungen praktisch umzusetzen. Jede Technologie wird begründet, um die Auswahl klar zu rechtfertigen.

### Hauptkomponenten und empfohlene Technologien

1. Zentrales Koordinationsmodul:
    - Technologie: Verteilte Datenbanken wie MongoDB oder CockroachDB.

    - Begründung: Diese Datenbanken ermöglichen eine einfache horizontale Skalierung und geographische Replikation, was ideal für ein globales System wie XmasWishes wäre.
2. Regionale Datenverarbeitungsmodule:
    - Technologie: Containerisierte Microservices (z. B. Docker, Kubernetes).
    - Begründung: Microservices sind unabhängig skalierbar und können spezifisch auf die Anforderungen einzelner Regionen zugeschnitten werden. Kubernetes zB. sorgt für automatische Skalierung und Orchestrierung.
3. Kommunikationsschnittstelle:
    - Technologie: REST- oder GraphQL-APIs.
    - Begründung: REST ist ein bewährter Standard für die Kommunikation zwischen Systemen und bietet einfache Implementierung. GraphQL kann verwendet werden, um flexible Datenabfragen zu ermöglichen, falls komplexe Datenstrukturen benötigt werden.
4. Statusverwaltung:
    - Technologie: Key-Value-Datenbanken wie Redis.
    - Begründung: Redis ermöglicht extrem schnelle Lese- und Schreibzugriffe und eignet sich hervorragend für den Status von Wünschen, der regelmäßig aktualisiert wird.
5. Datenmanagement:
    - Technologie: Kombination aus NoSQL-Datenbanken (z. B. DynamoDB) und Caching-Lösungen (z. B. Memcached).
    - Begründung: NoSQL bietet Flexibilität in der Datenstruktur und eignet sich für verteilte Systeme. Memcached reduziert Datenbankzugriffe durch Zwischenspeicherung häufig verwendeter Daten.
6. Kommunikation zwischen Modulen:
    - Technologie: Event-Streaming-Plattformen wie Apache Kafka oder RabbitMQ.
    - Begründung: Diese Plattformen ermöglichen eine asynchrone Kommunikation zwischen den Modulen und gewährleisten Skalierbarkeit und Entkopplung.
7. Monitoring und Datenschutz:
    - Technologie: Prometheus für Monitoring, ELK-Stack für Logging.
    - Begründung: Prometheus bietet umfassende Überwachungsfunktionen, während der ELK-Stack eine zentrale Protokollanalyse ermöglicht. Beide Technologien sind Open Source und lassen sich leicht in verteilten Systemen implementieren.

### Gesamtbegründung

*Die vorgeschlagenen Technologien wurden basierend auf ihrer Skalierbarkeit, Verfügbarkeit und Einfachheit der Integration in verteilte Systeme ausgewählt und bauen direkt auf den Anforderungen aus Aufgabe 1.*

## Aufgabe 3: Prototyp

### 1. Zielsetzung

Gemäß der Aufgabenstellung soll ein einfacher Prototyp entwickelt werden, der die grundlegenden Funktionalitäten von XmasWishes demonstriert und Lasttests (Lese- und Schreibzugriffe) durchführt. Anders als in den umfangreichen Konzepten aus Aufgabe 1 und 2 wird nicht das komplette modulare, verteilte System realisiert, sondern eine abgespeckte Version, die primär das API-Verhalten validiert und einfache Performance-Messungen ermöglicht.

### 2. Auswahl der Technologien

Auf Basis der Ergebnisse aus Aufgabe 2 – wo unterschiedliche Datenbanken (z. B. MongoDB, Redis), containerisierte Microservices und verteilte Messaging-Lösungen diskutiert wurden – habe ich für den Prototypen eine deutlich reduzierte Variante gewählt:

1. Programmiersprache:
    - **Python** – aufgrund seiner einfachen Syntax und den zahlreichen Bibliotheken für Web- und Test-Szenarien. Außerdem ist Python sehr beliebt für schnelle Prototyping-Zwecke.
2. Web-Framework:
    - **Flask** – ein leichtgewichtiges Framework, das REST-APIs mit wenig Code realisiert. Für einen einfachen Proof-of-Concept eignet es sich ideal (geringer Overhead, schnelle Umsetzung).
3. Datenhaltung:
    - In-Memory-Speicher (eine Python-Liste) anstelle einer echten Datenbank. Grund: Für den Prototyp reichen einfache Lese- und Schreibvorgänge, ohne komplexe Persistenz.

Die Wahl von Python + Flask ermöglicht eine schnelle Umsetzung und verdeutlicht die Kernidee: Ich wollte nur grundlegend zeigen, wie Wünsche erfasst und abgerufen werden können, und wie Lasttests ablaufen.

### 3. Umgesetzte Komponenten

#### 3.1. REST-API

- POST /wishes:
  Fügt einen neuen Wunsch (inkl. Text und Status) in der In-Memory-Liste hinzu.
- GET /wishes:
  Gibt alle Wünsche in JSON-Form zurück.
- PUT /wishes/<id>
  Aktualisiert den Status eines bestehenden Wunsches. Hierdurch kann z. B. das XmasWishes-System abbilden, dass ein Wunsch vom Status „noch nicht formuliert“ zu „in Bearbeitung“ oder „unter dem Weihnachtsbaum“ übergeht.

Diese minimalen Endpoints demonstrieren genau das, was ein XmasWishes-System für Testzwecke braucht: Wünsche schreiben und lesen.

### 3.2. Lasttest-Skript (Load Test)

- Ein Client-Programm in Python (z. B. mit dem Modul requests), das in Schleifen POST- und GET-Anfragen an den Server schickt.
- Messung der Requests pro Sekunde (RPS) durch Aufnahme der Zeit zu Beginn und am Ende des Tests.
- Parallelisierung über Threads, um mehrere zeitgleiche Anfragen zu simulieren.

Der Test bestimmt die maximale Anzahl an API-Calls pro Sekunde, die unter diesen simplen Bedingungen erreicht werden kann. Für den Prototyp reicht das völlig aus, um ein Gefühl für Durchsatz und Latenz zu bekommen.

### 3.3. Containerisierung mit Docker (Optional)

Zur Demonstration, wie sich der Prototyp in Containerform bereitstellen lässt, wurde ein Dockerfile erstellt. Dieses enthält:

- Die Basis: Python 3.11 (slim)
- Installation der Abhängigkeiten (Flask usw.)
- Startanweisung (CMD ["python", "app.py"])

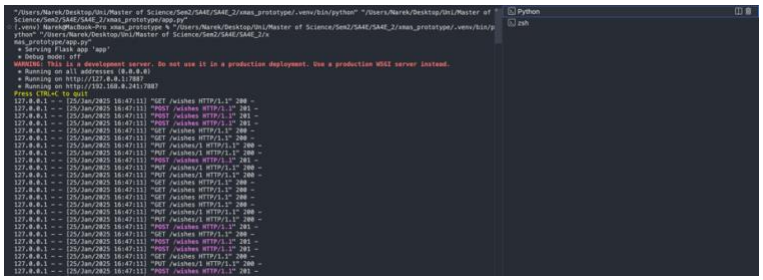
Damit kann der Prototyp leicht horizontal skaliert werden, indem man mehrere Container-Instanzen startet und per Load Balancer verteilt. Die in Aufgabe 1 und 2 beschriebene Skalierungsidee (z. B. Microservices und Orchestrierung via Kubernetes) lässt sich darauf aufbauen.

## 4. Gründe für die vereinfachte Umsetzung

1. Aufwand und Fokus:
    - *Aufgabe 3 verlangt* keine *komplette Implementierung aller Aspekte* aus Aufgabe 1 und 2 (keine Microservices, kein Kafka, keine verteilte Datenbank), sondern nur einen Proof-of-Concept für Lese-/Schreiboperationen.
2. Zeitersparnis:
    - *Ein komplexes System mit Containerisierung, NoSQL-Datenbanken oder Event-Streaming aufzubauen, würde den Rahmen sprengen*. Ziel ist es, praktisch zu zeigen, wie man Testdaten generiert und wie man Last messen kann.
3. Einfache Messbarkeit:
    - *Durch den Verzicht auf externe Komponenten (z. B. hochverfügbare Datenbanken oder Kubernetes) lassen sich Anfragen und Durchsatz direkt auf dem lokalen Rechner messen, ohne zusätzliche Netzwerk- oder Infrastrukturkomplexität.*

## 5. Durchgeführte Tests & Ergebnisse

1. Szenario:
    - 10 Threads, jeweils 20 Requests (POST + GET). Insgesamt 400 Requests in sehr kurzer Zeit.
2. Beobachtung:
    - Das System erreicht – je nach CPU/Betriebssystem – ~2000–3000 Requests pro Sekunde (lokale Messung).
    - Die Server-Ausgaben zeigten, dass alle Requests mit HTTP 200 bzw. 201 bestätigt werden. Auf meinen M3 Mac hatte ich folgende Erebnisse:



```
Total Requests: 400
Duration: 0.17 seconds
Requests per second (approx): 2330.96
```

3. Interpretation:
    - Die gemessenen RPS-Werte sind naturgemäß hoch, da alles lokal auf dem gleichen Rechner (ohne echte Datenbank) läuft und die Logik minimal ist.
    - Im Praxisbetrieb, verteilt über Regionen mit echter Persistenz, liegen die realen Zahlen niedriger. Trotzdem zeigt der Test prinzipiell, wie skalierbare Lasttests ablaufen können.

## 6. Skalierung in Bezug auf Aufgaben 1 und 2

- Vertikale Skalierung:
  Auf einem MacBook M3-Prozessor kann man die CPU- und RAM-Ausstattung steigern, um mehr Requests zu verarbeiten.
- Horizontale Skalierung (geplant in Aufgabe 1 und 2):
    - Mehrere Flask-Instanzen (z. B. via Docker/Kubernetes) hinter einem Load Balancer.
    - Verwendung einer verteilten Datenbank wie MongoDB oder CockroachDB, um global replizieren zu können.
    - Einsatz von RabbitMQ oder Kafka für eine asynchrone Kommunikation zwischen Modulen, wenn mehrere Regionen bedient werden.

## 7. Fazit

- Der Prototyp erfüllt die Kernvorgaben aus Aufgabe 3:
    1. Implementieren einer kleinen XmasWishes-API.
    2. Simulieren von Lese-/Schreibzugriffen (mehrere Clients).

3. Bestimmen der maximalen Anzahl an API-Calls pro Sekunde unter einfachen Bedingungen.

- Gleichzeitig bleibt der Prototyp deutlich kleiner als das in Aufgabe 1 und 2 beschriebene Gesamtsystem. Durch den reduzierten Umfang sind die erzielten RPS-Werte hoch – realistischere Ergebnisse würde man erst mit verteilter Infrastruktur und realen Datenbanken erhalten.
- Dieses Vorgehen ist sinnvoll, da der Hauptzweck in Aufgabe 3 nur das Ausprobieren und Messen ist. Eine vollständige Umsetzung aller Module und Technologien wäre zeitlich und konzeptionell viel aufwendiger und wurde nicht verlangt.

## Aufgabe 4: Konkretisierung

### Zielsetzung

Die Aufgabenstellung beschreibt eine Situation, in der einige Kinder und Erwachsene ihre Wünsche weiterhin auf Papier schreiben und diese per Post an den Nordpol senden. Dort werden die Briefe eingescannt und als digitale Dateien gespeichert. Ziel war es, eine Lösung zu entwickeln, die diese Dateien automatisiert verarbeitet und in das zuvor erstellte XmasWishes-System einfügt.

Daraus wurden folgende Anforderungen herausgearbeite:

1. *Dateien aus einem bestimmten Verzeichnis (im code → scanned) erkennen.*
2. *Den Inhalt der Dateien als "Wunsch" interpretieren.*
3. *Den Wunsch in strukturierter Form (als JSON) an das XmasWishes-System senden.*
4. *Den gesamten Ablauf durch Protokollierung nachvollziehbar machen.*

Die Hauptaufgabe bestand darin, eine Apache Camel-Anwendung zu entwickeln, die diese Anforderungen erfüllt und den Prozess automatisiert.

### Umsetzung der Anforderungen

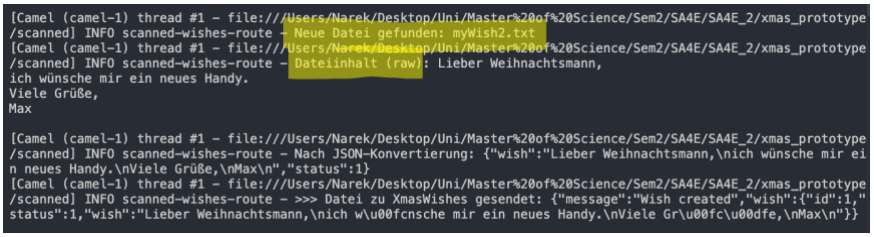
1. Anforderung: Dateien erkennen und verarbeiten
   - *Was wurde gemacht?* Eine Apache Camel-Komponente wurde so eingerichtet, dass sie das Verzeichnis scanned kontinuierlich überwacht. Immer wenn eine neue Datei in das Verzeichnis eingefügt wurde, hat Camel die Datei erkannt und zur weiteren Verarbeitung an die Route übergeben.
   - *Warum wurde das gemacht?* Um sicherzustellen, dass alle eingescannten Wünsche automatisch erfasst werden, ohne manuelles Eingreifen. Das Verzeichnis fungiert somit als "Eingang" für das System.
2. Anforderung: Inhalt der Dateien auslesen
   - *Was wurde gemacht?* Der Inhalt der Datei wurde mithilfe von Java-Mechanismen ausgelesen und in eine leicht verständliche Datenstruktur (ein Map-Objekt) umgewandelt. Diese Struktur enthielt den Wunschtext und einen Bearbeitungsstatus.
   - *Warum wurde das gemacht?* Der Wunschtext ist die zentrale Information, die verarbeitet werden muss. Der Bearbeitungsstatus wurde hinzugefügt, um anzuzeigen, dass der Wunsch im System eingegangen ist.

3. Anforderung: Wunsch als JSON an das XmasWishes-System senden
    - *Was wurde gemacht?* Die Datenstruktur mit dem Wunsch wurde in das JSON-Format umgewandelt, da dies ein standardisiertes Format für den Datenaustausch ist. Anschließend wurde die JSON-Datei über eine HTTP-POST-Anfrage an das XmasWishes-System gesendet, das bereits zuvor entwickelt wurde.
    - *Warum wurde das gemacht?* Das XmasWishes-System akzeptiert nur JSON-Format über eine REST-API. Die HTTP-POST-Anfrage wurde genutzt, um den Wunsch in das System zu integrieren.
4. Anforderung: Protokollierung des Prozesses
    - *Was wurde gemacht?* Jeder Schritt des Prozesses wurde protokolliert:
        - Das System hat angezeigt, welche Datei erkannt wurde.
        - Es wurde dokumentiert, welcher Inhalt aus der Datei gelesen wurde.
        - Auch die umgewandelten Daten (im JSON-Format) sowie die erfolgreiche Übertragung ins XmasWishes-System wurden im Log festgehalten.
    - *Warum wurde das gemacht?* Protokollierung ist wichtig, um den Prozess nachvollziehbar zu machen. Fehler könnten so leichter identifiziert und behoben werden.

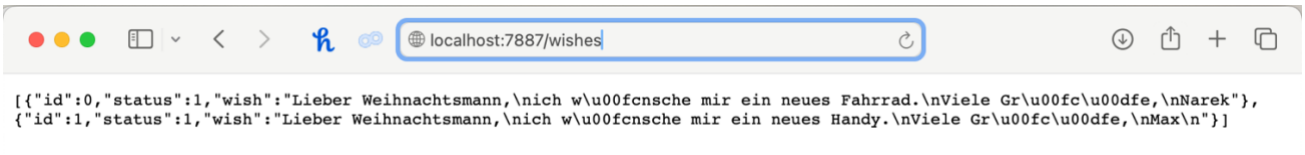
## Ergebnisse und Validierung

Nach der Implementierung wurde der Ablauf getestet:

1. Neue Dateien wurden automatisch erkannt, sobald sie in das Verzeichnis scanned eingefügt wurden.
2. Der Inhalt der Dateien wurde korrekt gelesen und als Wunsch in das XmasWishes-System eingefügt.




3. Das JSON-Format wurde erfolgreich erstellt und über eine HTTP-POST-Anfrage an das XmasWishes-System gesendet.




4. Auf der Konsole wurden alle Schritte nachvollziehbar protokolliert, sodass der gesamte Prozess transparent war.