Software Architectures for Enterprises
Aufgabe 2

Prof. Dr. Peter Sturm
Universität Trier
Professur für Wirtschaftsinformatik I

Nico Klose
1537568

1. Aufgabe 1: Modellierung

Das XmasWishes-System soll ein modernes, global skalierbares und robustes Softwaresystem werden, das die Weihnachtswünsche von Kindern und Erwachsenen verwaltet. Ziel ist es, saisonale Lastspitzen effizient zu bewältigen, geographisch verteilte Daten zu verarbeiten und gleichzeitig Datenschutz- und Verfügbarkeitsanforderungen zu erfüllen. In dieser Ausarbeitung wird ein Architekturentwurf für das System entwickelt, der die funktionalen und nicht-funktionalen Anforderungen berücksichtigt.

1.1 Anforderungen

1.1.1 Funktionale Anforderungen

1. **Erfassung der Wünsche:** Kinder und Erwachsene sollen ihre Wünsche jederzeit und weltweit eintragen können.
2. **Statusverwaltung:** Wünsche durchlaufen vier Stati:
   - Formuliert
   - In Bearbeitung
   - In Auslieferung
   - Unter dem Weihnachtsbaum
3. **Geografische Verteilung:** Daten aus verschiedenen Regionen müssen effizient verarbeitet werden

1.1.2 Nicht-funktionale Anforderungen

1. **Skalierbarkeit:** Das System muss saisonale Lastspitzen, insbesondere im letzten Quartal, problemlos bewältigen
2. **Verfügbarkeit:** Hohe Systemverfügbarkeit ist erforderlich, um weltweite Anfragen zuverlässig zu bedienen
3. **Performance:** Minimale Latenzzeiten bei der Datenerfassung und Statusaktualisierung
4. **Datenschutz:** Strikte DSGVO-Konformität, insbesondere bei der Verarbeitung personenbezogener Daten

1.1.3 Einschränkungen

- Die tatsächliche Geschenkproduktion und -auslieferung wird abstrahiert
- Der Fokus liegt ausschließlich auf der Aufnahme und Verarbeitung der Wünsche

1.2 Architekturkonzept

Das XmasWishes-System wird als verteilte Architektur mit modularen Komponenten konzipiert, um den Anforderungen an Skalierbarkeit, Flexibilität und globaler Verfügbarkeit gerecht zu werden. Der Entwurf basiert auf einer Kombination aus Microservices und einer Event-Driven Architecture (EDA), um die Verarbeitung von geographisch verteilt anfallenden Daten effizient zu bewältigen.

1.2.1 Frontend

Das Frontend bildet die benutzerfreundliche Schnittstelle des Systems und ermöglicht den Nutzern ihre Wünsche einzureichen oder den Status ihrer Wünsche in Echtzeit zu verfolgen. Diese Komponente muss plattformübergreifend konzipiert sein, um sowohl auf Webbrowsern als auch auf mobilen Endgeräten konsistent nutzbar zu sein.

1.2.2 Backend

Das Backend ist modular aufgebaut und in verschiedene Microservices unterteilt, die jeweils klar definierte Aufgaben übernehmen:

- **Wünsche-Service:**
  - Verarbeitet und speichert eingereichte Wünsche
  - Verfolgt den Status der Wünsche über den gesamten Lebenszyklus
- **Elfen-Service:**
  - Bereitet die Wünsche für die Geschenkproduktion vor und leitet diese weiter
- **Rentier-Service:**
  - Stellt Statusinformationen zur Auslieferung der Wünsche bereit und überwacht die Lieferprozesse.

1.2.3 Kommunikation zwischen den Komponenten

Die Kommunikation innerhalb des Systems erfolgt asynchron, um die Microservices voneinander zu entkoppeln und eine effiziente Verarbeitung von Ereignissen sicherzustellen:

- Ereignisse werden über einen zentralen Event-Broker verteilt
- Dies ermöglicht eine hohe Verfügbarkeit und Skalierbarkeit, insbesondere bei Lastspitzen

1.2.4 Datenhaltung und Replikation

Die Daten des Systems, einschließlich der Nutzerwünsche und Statusinformationen, werden in einer global replizierten Datenbank gespeichert

2. Aufgabe 2: Konkretisierung

Im Frontend wird React für die Webanwendung eingesetzt, ein Framework, das durch seine komponentenbasierte Architektur und das virtuelle DOM eine effiziente und flexible Entwicklung ermöglicht. React garantiert eine benutzerfreundliche und plattformübergreifende Nutzererfahrung, die auf allen gängigen Browsern responsiv und leistungsstark bleibt. Ergänzend dazu wird Flutter für die Entwicklung mobiler Anwendungen verwendet. Flutter ermöglicht eine plattformübergreifende App-Entwicklung mit einer konsistenten Nutzererfahrung auf iOS- und Android-Geräten. Durch die Nutzung einer gemeinsamen Codebasis wird die Entwicklung beschleunigt und die Wartung erleichtert.

Das Backend des Systems ist modular als Microservices-Architektur aufgebaut und nutzt Node.js als Laufzeitumgebung. Node.js überzeugt durch seine eventgesteuerte, nicht blockierende Architektur, die speziell für I/O-intensive Anwendungen wie das XmasWishes-System geeignet ist. Es erlaubt die parallele Verarbeitung zahlreicher Anfragen, was besonders in Lastspitzen entscheidend ist. Für die Kommunikation zwischen Frontend und Backend wird GraphQL eingesetzt. Diese Abfragesprache ermöglicht präzise und flexible Datenabfragen, wodurch die Netzwerklast reduziert und die API-Effizienz gesteigert wird. GraphQL bietet zudem eine hohe Erweiterbarkeit, da neue Felder und Funktionen nahtlos hinzugefügt werden können, ohne bestehende Dienste zu beeinträchtigen.

Die Kommunikation zwischen den Microservices erfolgt über Apache Kafka, das als Event-Broker dient. Kafka ermöglicht eine asynchrone, entkoppelte Kommunikation zwischen Diensten und gewährleistet eine zuverlässige Verarbeitung großer Datenmengen in Echtzeit. Nachrichten werden persistent gespeichert, was die Integrität des Systems erhöht und die Wiederherstellung bei Dienstausfällen erleichtert. Dank der horizontalen Skalierbarkeit von Kafka können auch saisonale Lastspitzen, wie sie in der Weihnachtszeit auftreten, problemlos bewältigt werden.

Für die Datenhaltung wird MongoDB als NoSQL-Datenbank eingesetzt. MongoDB zeichnet sich durch ein flexibles JSON-basiertes Datenmodell aus, das sich leicht an dynamische Anforderungen anpassen lässt. Die Datenbank unterstützt eine globale Replikation, wodurch weltweit schnelle Lese- und Schreibzugriffe sichergestellt werden. Ihre hohe Performance und die Fähigkeit zur horizontalen Skalierung machen MongoDB zur idealen Wahl für ein System mit global verteiltem Nutzeraufkommen.

Diese Architektur und die verwendeten Technologien sind in Abbildung 2.1 visualisiert, welche die Zusammenhänge und Datenflüsse zwischen den Frontend-, Backend- und Datenbank-Komponenten sowie die asynchrone Kommunikation über den Event-Broker darstellt.

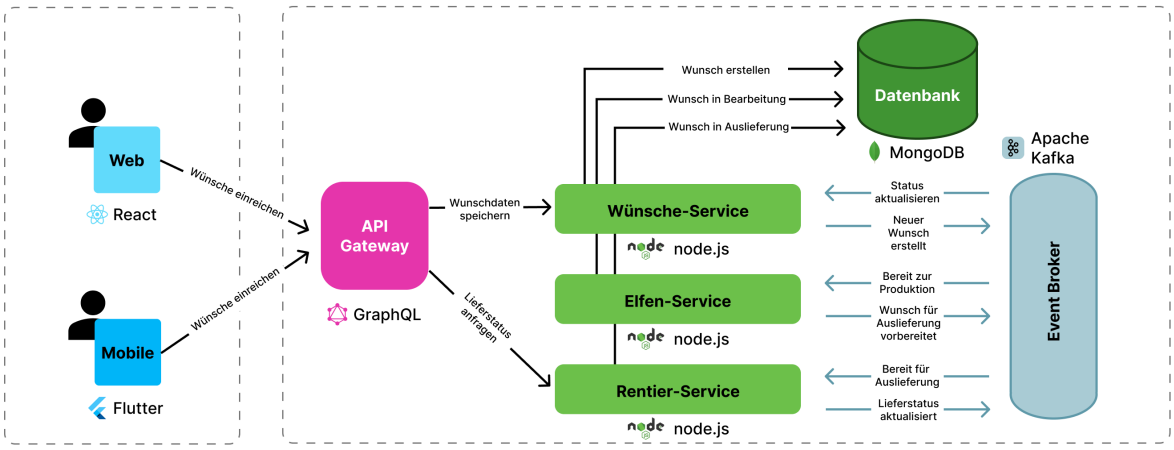

Abbildung 2.1: Architekturentwurf

3. Aufgabe 3: Prototyp

3.1 Kurzbeschreibung

Der entwickelte Prototyp des XmasWishes-Systems setzt die in den vorherigen Aufgaben beschriebenen Architektur- und Technologieentscheidungen erfolgreich um. Ziel war es, die grundlegenden Funktionen des Systems zu implementieren und durch Leistungstests die maximale Anzahl von API-Aufrufen pro Sekunde zu ermitteln. Der Schwerpunkt lag auf der Entwicklung von Microservices für die Verwaltung und Verarbeitung von Wünschen sowie auf der Nutzung von Apache Kafka als Event-Broker und MongoDB als Datenbank. Für die Bereitstellung der Services wurde Docker verwendet.

Das System besteht aus mehreren Komponenten. Das Frontend wurde mit React entwickelt und bietet eine einfache Benutzeroberfläche, die es den Nutzern ermöglicht, ihre Wünsche einzutragen und den Status ihrer Wünsche abzufragen. Aufgrund von CORS-Problemen konnte das Frontend jedoch nicht direkt mit dem Backend kommunizieren, was die vollständige Funktionalität des Prototyps eingeschränkt hat.

Im Backend wurden zwei Microservices implementiert. Der Wish-Service dient der Entgegennahme und Speicherung von Wünschen. Er bietet eine GraphQL-API mit zwei Hauptfunktionen: Die Mutation createWish ermöglicht es, einen neuen Wunsch in der MongoDB-Datenbank zu speichern und ein entsprechendes Ereignis an Apache Kafka zu senden. Die Query getWishStatus erlaubt es, den aktuellen Status eines Wunsches anhand seiner ID aus der Datenbank abzufragen. Der Elfen-Service konsumiert die vom Wish-Service gesendeten Kafka-Events. Sobald ein Event vom Typ WishCreated erkannt wird, wird der Status des entsprechenden Wunsches in der Datenbank von „Formuliert“ zu „In Bearbeitung“ geändert.

Für die Kommunikation zwischen den Microservices wurde Apache Kafka verwendet, das sich als zuverlässiger Event-Broker erwies. Der Wish-Service sendet Events an ein Kafka-Topic namens wish-events, während der Elfen-Service auf dieses Topic lauscht und die Ereignisse verarbeitet. MongoDB wurde als Datenbank genutzt, um die Wünsche zu speichern. Die Datenstruktur umfasst die Felder „name“, „wishText“ und „status“, die den Namen des Wünschenden, den Wunschtext und den aktuellen Status des Wunsches darstellen.

Die gesamte Architektur wurde mithilfe von Docker containerisiert, einschließlich der Microservices, Kafka und MongoDB. Mit Docker Compose konnten die Services einfach gestartet und lokal orchestriert werden. Dies vereinfachte den Entwicklungsprozess und die Bereitstellung des Prototyps.

3.2 API Tests

3.2.1 createWish

Die durchgeführten Tests zur Ermittlung der maximalen Verarbeitungsrate der API bei unterschiedlichen Parallelitätswerten (10 bis 1000 parallele Anfragen) zeigen eine solide Stabilität bei moderater bis hoher Last, jedoch deutliche Grenzen bei extremer Belastung. Bis zu 500 parallelen Anfragen konnte die API alle Anfragen erfolgreich verarbeiten, wobei die höchste Verarbeitungsrate von 89,87 Requests pro Sekunde (RPS) bei 500 parallelen Anfragen erreicht wurde. Die Erfolgsrate blieb in diesem Bereich stabil bei 100%, und die Testdauer lag nur leicht über der geplanten Zeit, was auf eine effiziente Verarbeitung hinweist.

Mit steigender Parallelität zeigte sich jedoch eine zunehmende Belastung des Systems. Bei 1000 parallelen Anfragen sank die Verarbeitungsrate drastisch auf 28,22 RPS, und die Erfolgsrate fiel auf 72,4%, da 276 Anfragen fehlschlugen. Gleichzeitig verlängerte sich die Testdauer deutlich auf 35,43 Sekunden, was auf eine Überlastung des Servers hinweist. Diese Ergebnisse zeigen, dass die API unter moderaten Lastbedingungen gut skaliert, jedoch bei extremer Last an Kapazitätsgrenzen stößt.

Insgesamt erreicht die API ihre optimale Leistung bei Parallelitätswerten von 100 bis 500 parallelen Anfragen, mit einer Spitzenrate von knapp 90 RPS. Sie zeigt eine robuste Stabilität bis zu einer moderaten Last, ist jedoch für extrem hohe Belastungen nicht ausgelegt.

3.2.2 getWishStatus

Die Tests für die Route ‘getWishStatus‘ zeigen eine hohe Stabilität und Skalierbarkeit der API. Die Verarbeitungsrate stieg mit zunehmender Last von 109,88 RPS (10 Threads, 10 Anfragen pro Thread) auf einen Spitzenwert von 148,59 RPS (100 Threads, 100 Anfragen pro Thread). Über alle Tests hinweg wurden keine Fehler oder Ausfälle beobachtet, was auf eine robuste und effiziente Verarbeitung hinweist.

Im Vergleich zu ‘createWish‘ zeigt ‘getWishStatus‘ eine bessere Anpassung an steigende Parallelität und Last, da die Verarbeitungsrate linear mit der Anzahl der Threads und Anfragen pro Thread anstieg. Dies deutet darauf hin, dass die Route weniger ressourcenintensiv ist und auch unter hoher Last zuverlässig skaliert. Mit einer maximalen Verarbeitungsrate von 148,59 RPS bei maximal getesteter Last eignet sich die Route gut für Szenarien mit hoher Abfragefrequenz.

4. Aufgabe 4: Apache Camel

Die Implementierung der Aufgabe erfolgt mithilfe einer Apache Camel-Anwendung, die eingesannte Briefe in verschiedenen Formaten verarbeitet und die extrahierten Daten in das XmasWishes-System integriert. Der gesamte Workflow basiert auf der modularen Struktur der Camel-Routen und individuellen Prozessoren, die die Verarbeitung der Dateien und die Transformation der Daten sicherstellen.

Zu Beginn werden die Dateien in einem definierten Eingangsordner (input) abgelegt. Der FileProcessor übernimmt die Erkennung des Dateiformats und die entsprechende Verarbeitung. Textdateien (.txt) werden direkt ausgelesen, während bei PDF-Dateien (.pdf) der Text mithilfe von Apache PDFBox extrahiert wird. Für Bilder (.jpg oder .png) wird Tesseract OCR verwendet, um handschriftliche oder gedruckte Inhalte in lesbaren Text umzuwandeln. Fehlerhafte Dateien, etwa solche mit nicht unterstützten Formaten, werden dabei robust behandelt und entsprechend geloggt.

Nach der Textextraktion übernimmt der JsonProcessor die Umwandlung des Textes in eine JSON-Struktur. Hierbei werden relevante Informationen wie name und wishText aus dem extrahierten Inhalt analysiert. Falls bestimmte Schlüssel fehlen, werden Standardwerte wie Ünknown"für den Namen oder "No wish provided"für den Wunschtext gesetzt. Das fertige JSON wird im Body der Nachricht gespeichert, während die extrahierten Felder zusätzlich als Header zur Verfügung gestellt werden.

Die Camel-Route steuert den gesamten Verarbeitungsprozess. Sie liest Dateien aus dem Eingangsordner, wendet die Prozessoren zur Verarbeitung an und übermittelt die erzeugten JSON-Daten an die XmasWishes-API. Dies erfolgt über eine dynamisch erzeugte GraphQL-Mutation, die die Felder name und wishText an das Zielsystem sendet. Nach erfolgreicher Verarbeitung werden die Dateien im Ordner success_wishes abgelegt. Dabei wird das ursprüngliche Dateiformat ignoriert, und die Ergebnisse werden konsistent als .txt gespeichert. Falls bei der Verarbeitung Fehler auftreten, werden die Dateien in den Ordner error_wishes verschoben.