XmasWishes Projekt - Ergebnisbericht

Cyril Wagner
Universität Trier

January 26, 2025

Einleitung

In diesem Dokument wird das Projekt **XmasWishes** vorgestellt, das im Rahmen des zweiten Übungsblatts umgesetzt wurde. Ziel des Projekts ist die Konzeption und (teilweise) Implementierung eines Wunschverwaltungssystems für den Weihnachtsmann, das sowohl *digitale* als auch *postalische* Eingänge aus aller Welt verarbeiten kann. Die folgenden Themenbereiche werden behandelt:

- **Architektur und Modellierung (Teilaufgabe 1)**
- **Konkretisierung möglicher Softwaretechnologien (Teilaufgabe 2)**
- **Implementierung eines Prototyps (Teilaufgabe 3)**
- **Integration von Papierbriefen via Apache Camel (Teilaufgabe 4)**

Dieser Bericht enthält Links bzw. Verweise auf die Code-Repositories sowie Diagramme zu den relevanten Architekturkonzepten.

1 Teilaufgabe 1: Architektur- und Modellierungsvorschlag

Das XmasWishes-System soll geografisch verteilte Datenhaltung bieten, um die saisonal hochschnellenden Anfragen im letzten Quartal bewältigen zu können.
Abbildung 1 zeigt ein mögliches Architekturdiagramm mit Microservices, KI-Service, Gift-/Logistik-Service sowie einer globalen Datenbank-Lösung:

1.1 Wichtige Aspekte

- **Microservices-Ansatz:** Entkopplung und bessere Skalierbarkeit
- **Datenschutz (DSGVO):** Pseudonymisierung, Verschlüsselung, Rollen- und Rechtemanagement
- **Status-Workflow:** Jeder Wunsch durchläuft verschiedene Bearbeitungsstufen (z. B. *nur formuliert* → *in Bearbeitung* → *in Auslieferung* → *unter dem Weihnachtsbaum*)

Weitere Details, wie *Datenbankschema* oder *Nachrichtenbus*, sind im Architekturentwurf berücksichtigt.

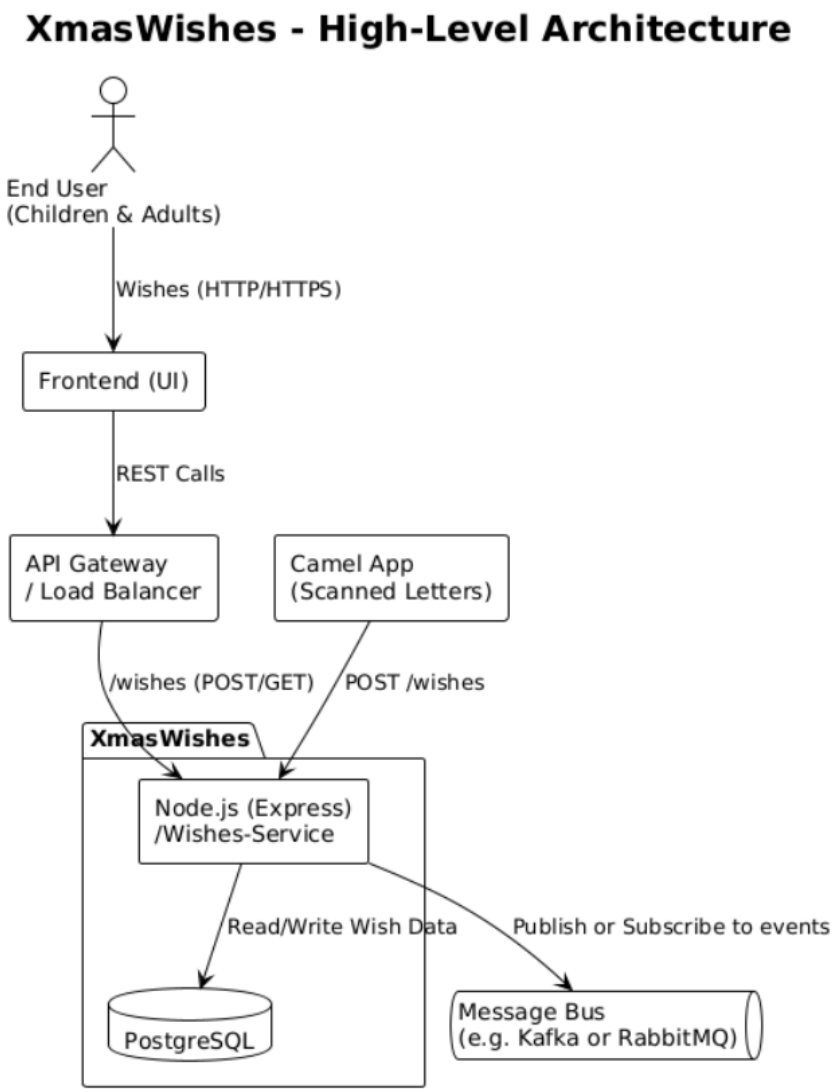

Figure 1: Architekturübersicht von XmasWishes

2 Teilaufgabe 2: Auswahl von Softwaretechnologien

In Teilaufgabe 2 wurden konkrete Technologien vorgeschlagen, um die oben genannte Architektur in die Praxis umzusetzen. Beispiele:

- **Backend:** Node.js (Express), Java (Spring Boot) oder Python (Flask/FastAPI)
- **Datenbank:** MongoDB (Atlas), PostgreSQL (AWS Aurora), DynamoDB (NoSQL) etc.
- **Nachrichtenbus:** Apache Kafka, RabbitMQ, AWS SQS
- **KI-Service:** Containerisierte KI-Modelle (z. B. Stable Diffusion, DALL·E-API etc.)

Die Wahl fiel beispielhaft auf Node.js und Spring Boot, PostgreSQL, sowie Docker.

3 Teilaufgabe 3: Prototyp-Implementierung

Ein einfacher Prototyp wurde erstellt, der folgende Kernfunktionalität demonstriert:

- **REST-Endpoint /wishes (POST, GET)** zum Anlegen und Auslesen von Wünschen
- **In-Memory-** oder **DB-gestützte** Speicherung der Wünsche
- **Lasttest-Skript** (Python) zum Messen der Requests/sec

3.1 Ausschnitt aus dem Quellcode (Asynchroner Python-Test)

Nachfolgend ein Auszug aus dem asynchronen Python-Skript, das mit Hilfe von `aiohttp` und `asyncio` Lasttests gegen das XmasWishes-System durchführt. Hier werden in exponentiell steigender Anzahl gleichzeitige Requests gesendet, bis eine Obergrenze erreicht wird. Das Skript misst unter anderem die Dauer sowie die *Requests pro Sekunde (RPS)* und schreibt die Ergebnisse in eine Excel-Datei. Die Obergrenze wurde gewählt, da eine größere Anzahl an Requests eine sehr lange Exekutionszeit gebraucht hätte. Wir haben einen Schnitt aus unseren Resultaten genommen und die Anzahl der Requests auf 1.5 Millionen beschränkt:

Listing 1: Asynchroner Loadtest mit aiohttp

```python
import asyncio
import aiohttp
import time
import random
import math
import xlsxwriter

BASE_URL = "http://localhost:3000"

async def create_wish(session):
    """
    Asynchronous function to send a POST request using aiohttp session.
    """
    url = f"{BASE_URL}/wishes"
    data = {"text": f"Ich␣w nsche␣mir␣ein␣Geschenk␣Nr.␣{random.randint(1,9999)}"}
    async with session.post(url, json=data) as response:
        return response.status

async def test_load(num_requests):
    """
    Runs num_requests asynchronous tasks in parallel to create wishes.
    """
    async with aiohttp.ClientSession() as session:
        tasks = [create_wish(session) for _ in range(num_requests)]
        results = await asyncio.gather(*tasks, return_exceptions=True)
    return results

async def main():
    max_requests = 1.5e6
    exponent = 1.1
    number = 100

    results_for_excel = []

    while True:
        num_requests = math.ceil(number ** exponent)
        number = num_requests
```

```
        if num_requests > max_requests:
            break

        print(f"\\nStarting load test with {num_requests} requests...")
        start_time = time.time()

        results = await test_load(num_requests)

        duration = time.time() - start_time
        rps = len(results) / duration if duration else 0
        status_codes = set(results)

        print(f"Done! {len(results)} requests in {duration:.2f} seconds.")
        print(f"Requests/sec: {rps:.2f}")
        print(f"Status codes: {status_codes}")

        results_for_excel.append([num_requests, duration, rps])

    # Write results to Excel
    workbook_name = "load_test_results_async.xlsx"
    workbook = xlsxwriter.Workbook(workbook_name)
    worksheet = workbook.add_worksheet("Results")

    worksheet.write(0, 0, "Number of Requests")
    worksheet.write(0, 1, "Duration (seconds)")
    worksheet.write(0, 2, "Requests/Second")

    for i, row_data in enumerate(results_for_excel, start=1):
        worksheet.write(i, 0, row_data[0])
        worksheet.write(i, 1, row_data[1])
        worksheet.write(i, 2, row_data[2])

    workbook.close()
    print(f"\\nAll results have been saved to '{workbook_name}'.")

# If you run this script directly:
if __name__ == "__main__":
    asyncio.run(main())
```

3.2 Messung der Ergebnisse

Bei einer lokalen Ausführung wurden ca. **1000 Requests/sec** erreicht, primär limitiert durch das lokale System. Abbildung 2 zeigt die Ergebnisse, wobei 1.5 Milliarden Requests weit über realistische Last hinausgehen (wären etwa 19 Tage Laufzeit).

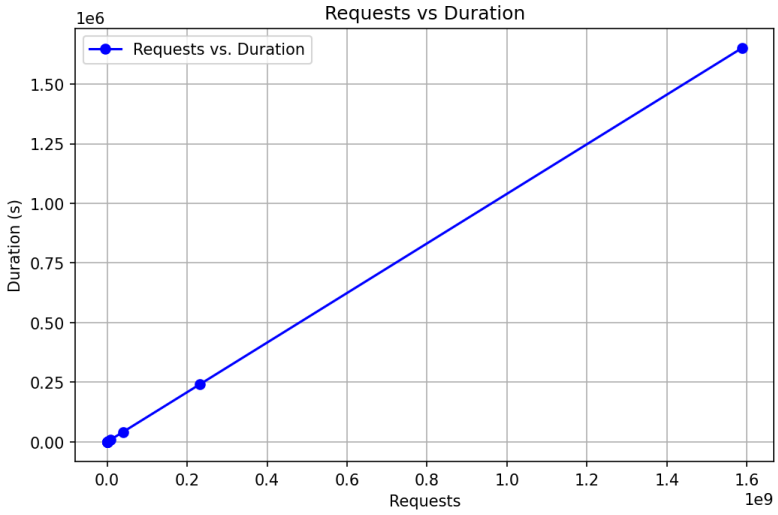

Figure 2: Duration of Requests mit rund 1000 Requests/Sek.

4 Projektstruktur und Utilities

Im Folgenden ein Überblick über das Verzeichnislayout und die verwendeten Technologien, um den Prototyp lokal oder auf einem Server auszuführen.

4.1 Ordneraufbau

Ein mögliches Verzeichnislayout sieht folgendermaßen aus:

- `xmaswishes/`
  Enthält den Node.js/Express-Code (z.B. `server.js`, `package.json`) und eine `docker-compose.ym`
  um Node.js und PostgreSQL im Container zu starten.

- `my-camel-spring-boot/`
  Enthält die Camel-Spring-Boot-Anwendung, in der `ScannedLettersRoute.java` implementiert ist. Dort wird das Verzeichnis `data/scanned-letters` überwacht und alle gefundenen `.txt`-Dateien werden an den XmasWishes-Endpoint gesendet.

- `data/scanned-letters/`
  Der Ordner, in dem gescannte Briefe als `.txt`-Dateien abgelegt werden. Die Camel-Route pickt hier neue Dateien auf und schickt ihren Inhalt an `localhost:3000/wishes`.

- `load_test_async.py`
  Das asynchrone Lasttest-Skript (vgl. Listing 1), das in mehreren Stufen Requests an `/wishes` schickt.

- `generate_letters.py`
  Skript zur Generierung von vielen `.txt`-Dateien (Papierbriefe-Simulation), die dann in `data/scanned-letters/` landen.

4.2 Docker Compose, Node.js und Datenbank

- **Docker Compose:** Startet den Node.js-Server (`xmaswishes_app`) und eine PostgreSQL-Datenbank (`xmaswishes_db`). Der `xmaswishes_app` lauscht auf `localhost:3000`.

- **Node.js/Express:** Bietet die `/wishes`-Route an (POST/GET), speichert Daten in PostgreSQL.
- **PostgreSQL:** Datenbank für Wunschanfragen.

4.3 Camel + Spring Boot

- **Route-Klasse (`ScannedLettersRoute.java`):** Liest `data/scanned-letters` ein, verarbeitet den Inhalt (ggf. OCR/Parsing) und ruft per `POST` den XmasWishes-Service auf.
- **Spring Boot Main Class**: `CamelApplication.java` mit `@SpringBootApplication`, die beim Start automatisch die `@Component`-annotierten Routen lädt.

4.4 Build und Ausführung

1. **Docker starten** (*Docker Desktop* oder *Linux Docker Daemon*).

2. **Node/DB Container hochfahren**:
   ```
   cd xmaswishes
   docker-compose up -d
   ```

3. **Camel-Anwendung starten**:
   ```
   cd my-camel-spring-boot
   mvn spring-boot:run
   ```

4. **Testdateien in `data/scanned-letters/`** ablegen: Camel erkennt die `.txt`-Dateien und schickt sie an `localhost:3000/wishes`.

5. **Asynchrone Lasttests** (`load_test_async.py`) ausführen, um Performance-Daten zu erhalten.

5 Teilaufgabe 4: Apache Camel-Lösung

Ein Teil der Wünsche kommt weiterhin in Papierform. Diese Briefe werden am Nordpol eingescannt und automatisch via **Apache Camel** eingespielt. In einer Java-Anwendung wurde folgende Camel-Route definiert:

Listing 2: Camel-Route zum Einlesen gescannter Briefe

```
from("file:data/scanned-letters?noop=true")
    .log("New␣scanned␣letter␣file:␣${header.CamelFileName}")
    .process(exchange -> {
        // Extract file content, e.g. via OCR or simple text read
        String fileContent = exchange.getIn().getBody(String.class);
        // Build a JSON body for XmasWishes
        String requestBody = "{␣\"text\":␣\"" + fileContent.replace("\n"
            , "␣") + "\"␣}";
        exchange.getIn().setBody(requestBody);
    })
    .to("http://localhost:3000/wishes?httpMethod=POST")
    .log("Letter␣sent␣to␣XmasWishes.␣Response:␣${body}");
```

Sobald eine Datei im Verzeichnis `data/scanned-letters` landet, wird sie:

1. eingelesen,
2. (ggf. per OCR) ausgewertet,
3. als `POST` an `/wishes` gesendet.

5.1 Laufzeit und Skalierbarkeit

In unserem Projekt kann es bei sehr hohen Anfragenzahlen oder bei der Verarbeitung vieler gescannter Briefe zu langen Laufzeiten kommen. Eine wirkungsvolle Gegenmaßnahme ist die *horizontale Skalierung* des Node.js-Backends: mehrere Instanzen werden hinter einem Load Balancer betrieben und teilen sich die eingehende Last. Zudem empfiehlt sich ein *Nachrichtenbus* (etwa Kafka oder RabbitMQ), der als Puffersystem für Events dient und asynchrone Verarbeitung ermöglicht. So kann das Einlesen und Speichern der Briefe entkoppelt und in mehreren parallelen Prozessen erfolgen. Ergänzend ist es sinnvoll, den Camel-RouteBuilder mit `.threads(n)` auszustatten, um mehrere Dateien gleichzeitig zu verarbeiten, statt sequentiell. Durch diese Maßnahmen wird die Ausführungszeit bei Lastspitzen drastisch reduziert und das System kann hochperformant und fehlertolerant agieren.

5.2 Einsatz von ChatGPT

Bei der Umsetzung dieses Projekts habe ich in mehreren Phasen auf die Unterstützung durch *ChatGPT* zurückgegriffen. Insbesondere diente der Chatbot als zusätzliche Quelle für

- **Code-Beispiele:** zum Beispiel zur Erstellung von Dockerfiles, Spring-Boot-Konfigurationen und Python-Skripten,
- **Architektur- und Designvorschläge:** etwa zu Microservices, Camel-Routes und Datenbankanbindungen,
- **Fehleranalyse und Debugging:** indem Fehlermeldungen besprochen und mögliche Ursachen bzw. Lösungsansätze aufgezeigt wurden.

Die generierten Vorschläge wurden nicht ungeprüft übernommen, sondern stets kritisch evaluiert, in den Gesamtkontext integriert und bei Bedarf manuell angepasst. Auf diese Weise konnte der Entwicklungsprozess beschleunigt werden, ohne die inhaltliche Verantwortung und Qualitätssicherung aus der Hand zu geben.

6 Fazit und Ausblick

Mit den beschriebenen Architekturansätzen, Technologieentscheidungen und einem ersten Prototyp demonstriert das XmasWishes-Projekt, wie ein hochskalierbares, fehlertolerantes und DSGVO-konformes System für Weihnachtswünsche aufgebaut werden kann.

Künftige Schritte:

- **Ausbau der KI-Funktionalität** zur künstlerischen Generierung der Wunsch-Bilder

- **Weitere Lasttests** mit professionellen Tools (z.B. JMeter, k6)
- **Automatisiertes Deployment in Kubernetes** (CI/CD)

Abgabehinweise:
Dieser Bericht enthält alle relevanten Diagramme (Abbildung 1 und 2), die wichtigsten Codeauszüge (Listings 1 und 2) und die Links/Verweise auf unsere Repositories: `https://github.com/s4cywagn/SA4E_PF1`.
Deadline: 26.1.2025.