

Portfolio Prüfung

25. Januar 2025

Inhaltsverzeichnis

1	Konzept der Microservices	2
1.1	Globale Architekturübersicht	3
1.1.1	Regionalisierte Cluster und Globale Lastverteilung	3
2	Modellierung	7
3	Prototyp	9
4	Test der API-Calls	10
5	Apache Camel	11

1 Konzept der Microservices

Für das generelle Konzept des **XmasWishes-System** schlage ich eine klassische 3-Tier-Architektur. So lässt sich zunächst die Verarbeitung sowie die Verwaltung von Geschenkswünschen in einer skalierbaren Weise darstellen. Die Herangehensweise ist dabei in mehrere Ebenen unterteilt:

- **Frontend:** Diese Schicht ist dafür verantwortlich, die Interaktion mit den Benutzern zu ermöglichen. Zum Beispiel wird ein Formular bereitgestellt, in das die Benutzer ihre Wünsche eintragen können. Die Eingaben werden validiert, um sicherzustellen, dass keine ungültigen oder unvollständigen Daten verarbeitet werden. Zusätzlich könnte die Benutzeroberfläche einfache visuelle Elemente wie die Anzahl der aktuell eingereichten Wünsche oder sonstige Statusanzeigen des Weihnachtsmannes beinhalten; um den Benutzern ein direktes Feedback zu geben.
- **Backend:** Hier liegt die Logik des Systems. Die Anwendungsschicht empfängt die Benutzerdaten von der Präsentationsschicht, prüft diese auf Vollständigkeit und speichert sie in der Datenbank. Darüber hinaus kümmert sich diese Schicht um Zustandsänderungen der Wünsche, beispielsweise wenn ein Wunsch von „formuliert“ in „in Bearbeitung“ wechselt. Auch komplexe Prozesse wie die asynchrone Verarbeitung oder Benachrichtigungen an andere Systeme werden hier gesteuert.
- **Datenbank:** Diese Schicht ist für die dauerhafte Speicherung der Daten verantwortlich. Im Szenario von XmasWishes werden alle Wünsche in der Datenbank gespeichert. Dazu gehören Indizes, der genaue Wunschttext, der Wohnort des Wünschenden, eventuelle Zeitstempel als auch der Name des Kindes (klein-gebliebenen Erwachsenen). Durch die Nutzung von Indizes kann die Suche nach bestimmten Wünschen, etwa nach Status oder Name, effizient gestaltet werden. Backups und Replikationen können dafür sorgen, dass keine Daten verloren gehen und die Verfügbarkeit auch bei Hardware-Ausfällen sichergestellt ist.

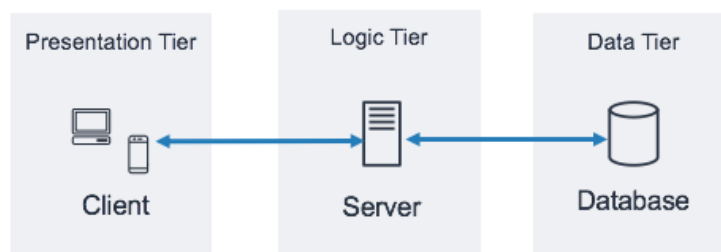


Abbildung 1: 3-Tier-Architektur

1.1 Globale Architekturübersicht

Geografisch verteilte Cluster sind eine bewährte Strategie, um eine skalierbare und performante IT-Infrastruktur bereitzustellen, die global effizient arbeitet. Die Idee dahinter ist, die Last auf verschiedene Regionen oder Kontinente zu verteilen, sodass jede Region ein eigenständiges Cluster betreibt. Besonders in Spitzenzeiten, wie der Weihnachtszeit mit hohem Anfrageaufkommen, kann die Belastung so horizontal skaliert werden. Jedes Cluster dient als unabhängige Einheit und sorgt für maximale Verfügbarkeit sowie eine effiziente Verarbeitung der Anfragen.

1.1.1 Regionalisierte Cluster und Globale Lastverteilung

Jedes Cluster ist geografisch an eine bestimmte Region oder einen Kontinent gebunden und wird strategisch so positioniert, dass die durchschnittliche Latenzzeit für die Nutzer minimiert wird. Innerhalb eines Clusters kommunizieren Webserver, die nach einer 3-Tier-Architektur aufgebaut sind, mit einer gemeinsamen Datenbank. Diese Webserver werden als Microservices betrieben und geografisch so verteilt, dass sie möglichst nah bei den Nutzern sind. Die physische Nähe der Server reduziert nicht nur die Antwortzeiten, sondern verbessert auch die Nutzererfahrung, insbesondere bei Echtzeitanfragen oder datenintensiven Vorgängen.

Damit Nutzer effizient mit dem richtigen Cluster verbunden werden, wird DNS-Geolokation eingesetzt. DNS-Dienste mit Geolokationsunterstützung analysieren die IP-Adresse eines Nutzers, um dessen geografischen Standort zu ermitteln. Basierend auf dieser Analyse wird die Anfrage an den Load-Balancer des nächstgelegenen Clusters weitergeleitet. Beispielsweise wird ein Nutzer aus Europa automatisch mit dem Europa-Cluster verbunden, während ein Nutzer aus den USA das nordamerikanische Cluster verwendet.

Neben der DNS-Geolokation kommen globale Load-Balancer zum Einsatz, um Anfragen dynamisch zu verteilen. Diese Load-Balancer leiten Anfragen basierend auf Kriterien wie Latenz, Netzwerkstabilität oder Verfügbarkeit an das optimale Cluster weiter. Dadurch wird sichergestellt, dass Nutzer immer die bestmögliche Verbindung erhalten. Zudem unterstützen diese Systeme Failover-Szenarien, sodass bei einem Ausfall eines Clusters Anfragen an ein benachbartes Cluster weitergeleitet werden können.

Ein zentraler Aspekt geografisch verteilter Cluster ist die Synchronisation der Datenbanken, um konsistente Daten global verfügbar zu halten. MongoDB bietet hierfür zwei wesentliche Mechanismen:

- Datenbankreplikation und Synchronisation
 - **Sharding:** Die Daten werden in Partitionen aufgeteilt, die geografisch verteilt werden können. Jede Region hat somit Zugriff auf die relevanten Daten und kann Anfragen lokal verarbeiten.
 - **Replica Sets:** Änderungen an einer Datenbank innerhalb eines Clusters werden automatisch auf die anderen Replikate synchronisiert, um Konsistenz zu gewährleisten und Redundanz zu schaffen.

Für eine global skalierte Architektur wäre hier Sharding ein passenderer Ansatz. Um sicherzustellen, dass ein Nutzer stets mit dem richtigen Cluster verbunden wird, können verschiedene Maßnahmen ergriffen werden.

Eine Möglichkeit ist die DNS-Regionserkennung, bei der DNS-Anfragen automatisch an das geografisch nächstgelegene Cluster weitergeleitet werden. Dadurch wird gewährleistet, dass die Anfragen möglichst effizient und mit geringer Latenz verarbeitet werden. Eine weitere Option sind regionale Subdomains, wie beispielsweise eu.example.com oder us.example.com, die eine explizite Zuordnung der Anfragen zu einem bestimmten Cluster ermöglichen. Zusätzlich können APIs spezifische Endpunkte für jede Region bereitstellen, um sicherzustellen, dass die Verarbeitung der Anfragen lokal erfolgt und den Anforderungen der jeweiligen Region entspricht. Diese Ansätze zusammen schaffen eine robuste und effiziente Infrastruktur für die globale Verteilung der Nutzeranfragen.

Darüber hinaus ermöglicht die geografische Verteilung der Cluster nicht nur eine geringere Latenz, sondern auch eine verbesserte Fehlertoleranz. Fällt ein Cluster in einer Region aus, können die anderen Cluster weiterhin ohne Unterbrechung arbeiten, was die Gesamtzuverlässigkeit des Systems erheblich steigert. Die Last wird dabei durch Load Balancer effizient auf die verfügbaren Webserver innerhalb eines Clusters verteilt, während die Cluster selbst durch ein zentrales System miteinander verbunden bleiben, das für die Synchronisation und zentrale Verarbeitung verantwortlich ist. Diese Verteilung bietet außerdem die Möglichkeit, die Serverressourcen dynamisch anzupassen, sodass saisonale Spitzen, wie sie zur Weihnachtszeit auftreten, problemlos abgefangen werden können, ohne dass die Benutzererfahrung darunter leidet.

Die Webserver einer Ebene werden alle mittels eines gemeinsamen Load Balancers angesteuert, der die eingehenden Anfragen effizient auf die verfügbaren Server verteilt. Dabei können je nach gewünschtem Prinzip verschiedene Verteilungsstrategien verwendet werden, wie zum Beispiel Round Robin, bei dem die Anfragen nacheinander an die Server weitergeleitet werden, oder Least Connections, bei dem der Server mit der geringsten Auslastung priorisiert wird. Dies sorgt dafür, dass die Last gleichmäßig verteilt wird und Engpässe bei stark frequentierten Servern vermieden werden. Der Einsatz eines Load Balancers erhöht nicht nur die Verfügbarkeit des Systems, sondern stellt auch sicher, dass

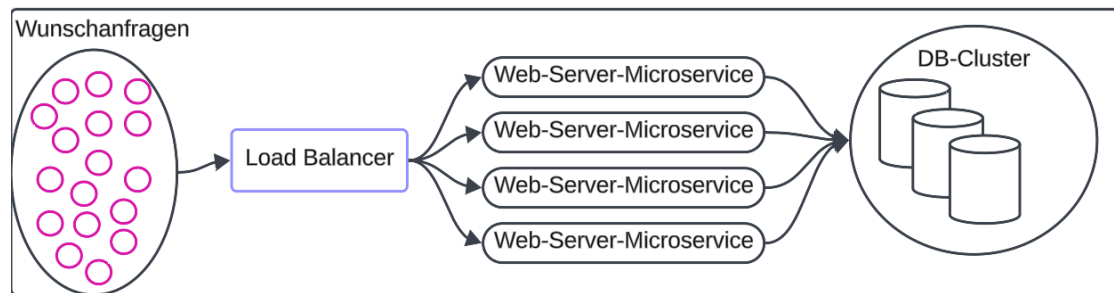


Abbildung 2: Regionales Konzept der Anfragenverwaltung

Ausfälle einzelner Server für die Benutzer nahezu unbemerkt bleiben, da die Anfragen automatisch auf andere verfügbare Instanzen umgeleitet werden können.

Alle Webserver in einer Region teilen sich einen gemeinsamen Datenbank-Cluster, der für die Speicherung und Verwaltung der regionalen Daten verantwortlich ist. Dieser Cluster nutzt Technologien wie Replikation, um eine hohe Datenverfügbarkeit und Ausfallsicherheit zu gewährleisten. Gleichzeitig ermöglicht das Sharding des Datenbankclusters die horizontale Skalierung, sodass große Datenmengen effizient verarbeitet und gespeichert werden können. Die gemeinsame Nutzung eines Datenbank-Clusters innerhalb einer Region gewährleistet zudem die Konsistenz der Daten, da alle Webserver auf denselben Datenbestand zugreifen können. Dies ist besonders wichtig, um sicherzustellen, dass Benutzeranfragen unabhängig vom Webserver, der sie bearbeitet, immer auf aktuelle und konsistente Daten zugreifen.

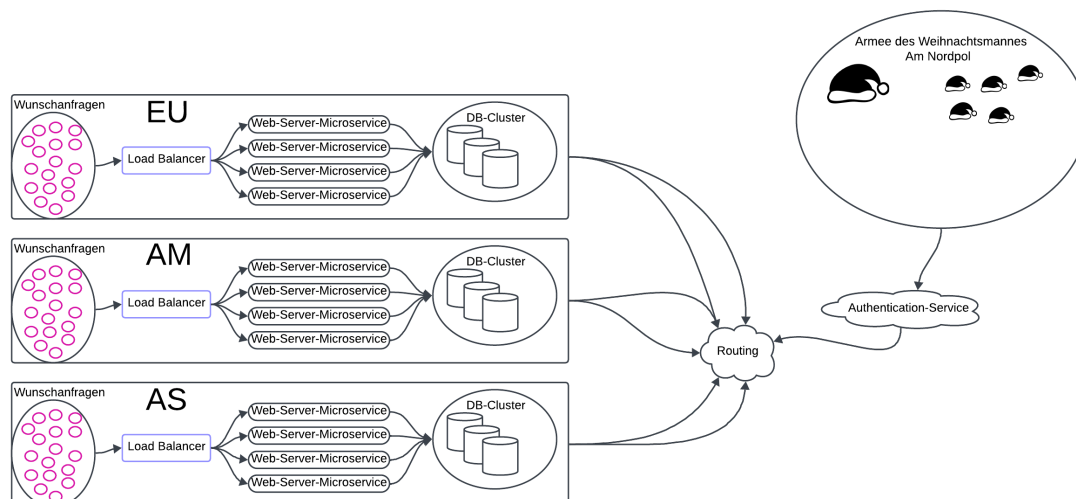


Abbildung 3: Gesamtkonzept

Das gesamte Konzept, wie in 3 skizziert, ermöglicht somit eine beliebige Skalierung über verschiedene Regionen. Die geografische Verteilung der Serverinfrastruktur stellt sicher,

dass jede Region unabhängig skaliert werden kann, um den jeweiligen Anforderungen gerecht zu werden, während die zentrale Steuerung und Verarbeitung über den Nordpol erfolgt. Die andere Seite des IT-Systems, die für den Weihnachtsmann und seine Elfen vorgesehen ist, wird ebenfalls berücksichtigt. Der Weihnachtsmann und seine Elfen können vom Nordpol aus auf die zentralen Datenbanken zugreifen, um die Wünsche der Kinder einzusehen, zu priorisieren und zu bearbeiten. Dieser Zugriff erfolgt über ein API-Gateway, das die Kommunikation zwischen der operativen Ebene (regionalen Clustern) und der zentralen Verarbeitungsebene sicherstellt.

Je nachdem, wie streng der Weihnachtsmann in Bezug auf Datenschutz und DSGVO-Vorgaben ist, könnte ein Authentifizierungsservice dazwischengeschaltet werden, um sicherzustellen, dass nur autorisierte Nutzer Zugriff auf die Daten haben. Für den aktuellen Prototyp wird jedoch davon ausgegangen, dass der Weihnachtsmann seinen Elfen vertraut und auf die Implementierung einzelner Nutzeraccounts für die Elfen verzichtet. Dies vereinfacht die Architektur und ermöglicht eine schnellere Umsetzung des Systems, ohne dabei den Fokus auf die Funktionalität der Wunschbearbeitung zu verlieren. In einer späteren Erweiterung könnte dieser Aspekt jedoch berücksichtigt werden, um auch den Anforderungen an eine detaillierte Rollen- und Rechteverwaltung gerecht zu werden. Die Nordpol-Bewohnern könnten dann je nach Rolle die Datenbankeinträge verwalten und den Status der einzelnen Wünsche je nach Bedarf anpassen.

2 Modellierung

Ich habe mich für Node.js als Grundlage für das Backend entschieden, weil es eine äußerst effiziente und skalierbare Plattform bietet. Mit seinem eventgesteuerten, nicht blockierenden I/O-Modell eignet sich Node.js hervorragend für Anwendungen, die viele parallele Anfragen verarbeiten müssen. Dadurch kann man sicherstellen, dass das Backend selbst bei hohem Anfrageaufkommen reaktionsschnell bleibt.

Ein weiterer Grund für meine Wahl ist die Modularität von Node.js. Mit dem Node Package Manager (npm) hat man Zugriff auf eine Vielzahl von Bibliotheken und Tools, die mir die Entwicklung erleichtern. Für die Erstellung der RESTful APIs kann Express.js verwendet werden; ein leichtgewichtiges Framework, das mir klare und flexible Routing-Mechanismen bietet. Für die Datenbankanbindung wäre Mongoose eine Variante, die es ermöglicht, MongoDB-Datenmodelle einfach zu definieren und zu validieren.

Für die Datenbank habe ich mich bewusst für MongoDB entschieden, da sie perfekt zu den Anforderungen meiner Anwendung passt. Als dokumentenbasierte NoSQL-Datenbank bietet sie die Flexibilität, Daten in JSON-ähnlichen Dokumenten zu speichern, was ideal ist, da sich die Datenstruktur während der Entwicklung ändern kann. Diese Flexibilität ermöglicht es mir, schnell auf neue Anforderungen zu reagieren, ohne aufwendige Datenbankmigrationen durchführen zu müssen.

Ein weiterer großer Vorteil von MongoDB ist die Skalierbarkeit. Mit Sharding wird es ermöglicht, die Daten auf mehrere Server zu verteilen, wodurch die Datenbank horizontal skaliert und große Datenmengen effizient verarbeitet werden können. Durch die Replikation der Daten wird zudem eine hohe Verfügbarkeit gewährleistet, sodass selbst bei einem Ausfall eines Knotens die Daten weiterhin zugänglich bleiben. Für jede Region wird ein eigenes MongoDB-Cluster implementiert, das die Daten regional konsistent hält und dennoch die Möglichkeit bietet, sie später zentral zu aggregieren.

Als Load Balancer habe ich NGINX zur Verfügung gestellt, da es eine leistungsstarke und bewährte Lösung für die Verteilung von Anfragen ist. NGINX erlaubt es mir, die eingehenden Anfragen effizient auf die verfügbaren Node.js-Server zu verteilen, wobei man verschiedene Strategien wie Round Robin oder Least Connections nutzen kann. Mit Round Robin ist sichergestellt, dass alle Server gleichmäßig ausgelastet werden, während Least Connections sicherstellt, dass die am wenigsten ausgelasteten Server bevorzugt werden, was bei hoher Last von Vorteil ist.

Zusätzlich dient NGINX als Proxy-Server. Das bedeutet, dass es die Anfragen der Benutzer entgegennimmt, sie an die entsprechenden Node.js-Container weiterleitet und die Antworten zurück an die Benutzer sendet. Dadurch bleibt die interne Infrastruktur geschützt, und man kann die Kommunikation zwischen Benutzer und Server optimieren. Für jede Region wird ein eigener NGINX-Load-Balancer eingerichtet, der die Anfragen lokal verteilt.

Docker wird als Grundlage für die Containerisierung der Anwendung verwendet. Mit Docker kann jeder Node.js-Microservice in einem eigenen Container betrieben werden, wodurch die Dienste voneinander isoliert und unabhängig voneinander skaliert werden können. Diese Isolation ermöglicht es, die Anwendung problemlos auf verschiedene Systeme zu deployen, ohne dass Konflikte zwischen den Diensten auftreten.

Ein wesentlicher Vorteil von Docker ist die einfache Skalierbarkeit. Bei Bedarf können zusätzliche Instanzen eines Microservices durch das Starten weiterer Container bereitgestellt werden, die automatisch vom NGINX-Load-Balancer integriert werden. Mit Docker Compose lassen sich die verschiedenen Container orchestrieren, darunter die Node.js-Microservices, die MongoDB-Cluster und die NGINX-Load-Balancer. Diese Orchestrierung erlaubt es, die gesamte Infrastruktur mit einem einzigen Befehl zu starten und zentral zu verwalten. Docker gewährleistet nicht nur eine hohe Skalierbarkeit und Portabilität der Anwendung, sondern bietet auch konsistente und wiederholbare Builds, was die Wartbarkeit und Weiterentwicklung der Infrastruktur erheblich vereinfacht.

3 Prototyp

Für den Prototypen habe ich nur einen Layer der Architektur implementiert. Dies entspricht dann einem möglichen Pfad innerhalb einer Regionalen Cluster. Also einem Microwebservice, einer MongoDB und einem Loadbalancer.

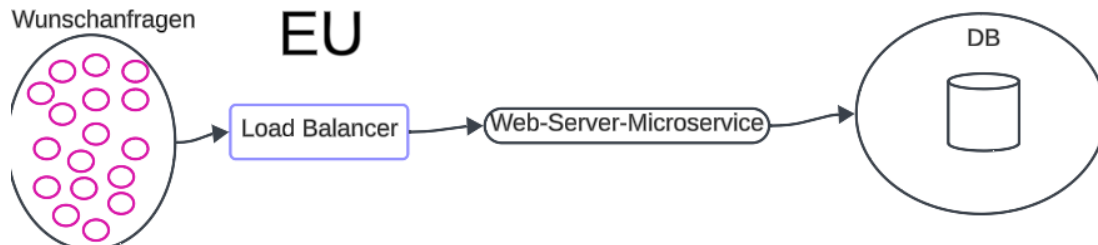


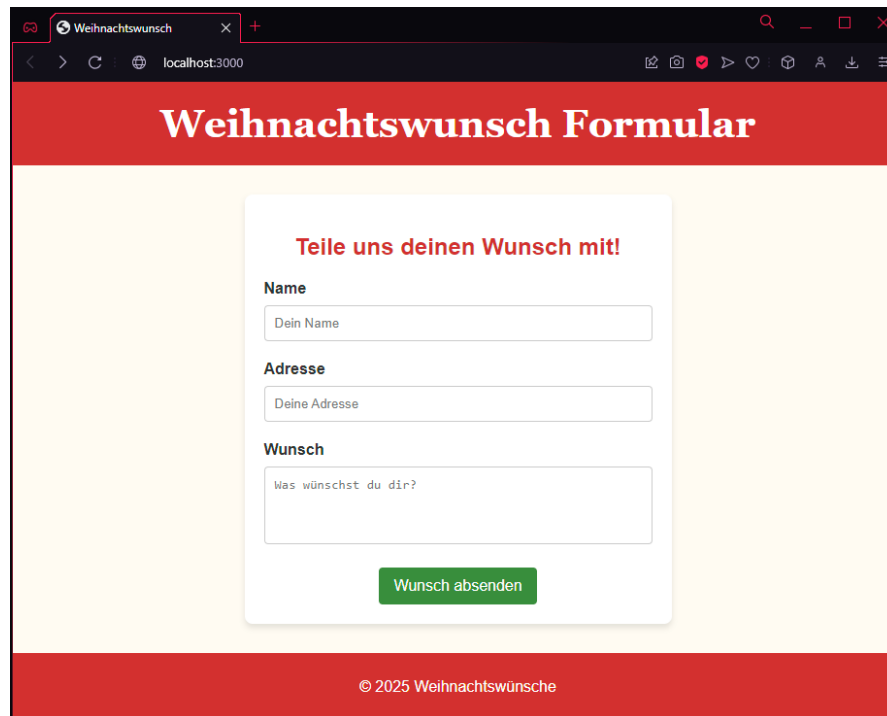
Abbildung 4: Implementierung einer Schicht

Die Webserver laufen mithilfe von Docker containern und können zum BSP mit Docker compose einfach gestartet werden. Für Testzwecke lässt sich der Webserver auch einzeln starten und läuft und ist dann mit localhost erreichbar. Da jeder Webserver in einem eigenen Docker-container läuft, kann man diese beliebig skalieren und je nach Wunschaufkommen dazu schalten. So könnte man im Sommer wenn eher weniger Wünsche geäußert werden das ganze auf ein Minimum reduzieren. Die einzelnen Datenbanken laufen ebenfalls in Docker containern die mit der Docker-Compose zusammen gestartet werden können.

```
simon@simon: /mnt/c/Users/Simon_Szulik/PycharmProjects/Software-Architecture-4-Enterprises/Uebung02/app$ node server.js
(node:7319) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
(Use 'node --trace-warnings ...' to show where the warning was created)
(node:7319) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
Server running on http://localhost:3000
```

Abbildung 5: Webserver Test

Hinter dem Webserver liegt dann eine einfache html-form die eine Post-Routine enthält. Mit dieser können Nutzer dann ihre Wünsche senden. Diese werden dann je nachdem auf welchen Server zugegriffen wird mit dem Loadbalancer verteilt und in die Datenbank geschrieben.



The screenshot shows a web browser window with the title 'Weihnachtswunsch'. The address bar shows 'localhost:3000'. The page has a red header with the text 'Weihnachtswunsch Formular'. Below the header is a white form with a red title 'Teile uns deinen Wunsch mit!'. The form contains three input fields: 'Name' with placeholder text 'Dein Name', 'Adresse' with placeholder text 'Deine Adresse', and 'Wunsch' with placeholder text 'Was wünschst du dir?'. Below the 'Wunsch' field is a green button labeled 'Wunsch absenden'. At the bottom of the page is a red footer with the text '© 2025 Weihnachtswünsche'.

Abbildung 6: Wunschtemplate

4 Test der API-Calls

Um die Latenz und allgemeine Performance der API-Calls zu berechnen, können zwei verschiedene Aktionen in Betracht gezogen werden. Zum einen könnte man die Latenz messen, wenn der Weihnachtsmann und seine Helfer den Status vor Geschenken ändern und diese beispielsweise zum Versand freigeben. Diese Art von Latenz erscheint jedoch weniger relevant, da alle Anfragen vom Nordpol kommen und das System in der aktuellen Implementierung kein Rollensystem oder ähnliche Authentifizierungsmechanismen besitzt. Viel interessanter ist es, die Performance eines Inserts in die Datenbank zu testen, da dies dabei hilft, potenzielle Bottlenecks im System aufzudecken und den Load-Balancer besser zu optimieren. Ein umfassender Stresstest über das gesamte System würde allerdings die vollständige Implementierung der geografischen Schichten voraussetzen, was in dieser Testphase noch nicht erfolgt ist. Daher habe ich die von mir lokal aufgesetzte Schicht getestet und eine Zeit lang API-Calls in einer Schleife versendet, um die Latenz und Gesamtzeit der Inserts in die Datenbank zu messen. Für diesen Test

habe ich lokal 1.000.000 API-Calls betrachtet, die alle den gleichen Endpunkt aufrufen, und dabei gemessen, wie lange es dauert, bis die Daten erfolgreich in die Datenbank überführt wurden. Das Testergebnis ist allerdings stark von der Rechnerqualität und Geschwindigkeit abhängig, insbesondere von der Prozessorleistung, der Festplattengeschwindigkeit (z. B. SSD vs. HDD) und der verfügbaren Netzwerkbandbreite, falls die MongoDB nicht lokal läuft. Diese haben ziemlich genau 23 Minuten gebraucht, was eine durchschnittliche Latenz von 1,38 MS für einen API-Call entspricht.

5 Apache Camel

Apache Camel ist ein leistungsfähiges Open-Source-Framework, das es ermöglicht, Daten zwischen verschiedenen Systemen und Anwendungen zu übertragen und zu transformieren. Es basiert auf sogenannten Enterprise Integration Patterns (EIP), die bewährte Methoden zur Verarbeitung und Integration von Daten beschreiben. Das Framework arbeitet auf der Basis von sogenannten Routen, die den Fluss der Daten von einer Quelle zu einem Ziel definieren. Eine Route beschreibt, wo die Daten herkommen (z. B. aus einem Dateisystem), welche Transformationen durchgeführt werden (z. B. Parsing oder Konvertierung) und wo die Daten schließlich gespeichert oder weitergeleitet werden (z. B. in eine Datenbank oder API).

In unserem Fall wird eine Ordnerstruktur im Webserver angelegt, in dem Nutzer Dateien ablegen können. Apache Camel wird dann so konfiguriert, dass es ein bestimmtes Verzeichnis überwacht, in dem die eingescannten Wunschbriefe abgelegt werden. Jede neue Datei wird automatisch von Camel erkannt und weiterverarbeitet. Der Inhalt der Datei wird zunächst ausgelesen und in das JSON-Format konvertiert. Anschließend wird eine HTTP-POST-Anfrage an die API des Node.js-Backends gesendet, die die extrahierten Daten (wie Name, Adresse und Wunsch) empfängt und in der Datenbank speichert.

Das Node.js-Backend wird um einen zusätzlichen API-Endpunkt erweitert, der die von Apache Camel gesendeten Daten empfängt. Dieser Endpunkt nimmt die HTTP-POST-Anfragen entgegen, validiert die Daten und speichert sie in der MongoDB-Datenbank. Der API-Endpunkt überprüft dabei, ob alle erforderlichen Felder (z. B. Name, Adresse, Wunsch) vorhanden sind, und gibt im Fehlerfall eine entsprechende Antwort zurück.