

**Projektarbeit
in der Angewandten Informatik**

Spotify 2.0

**Dominic Meyer
Friedemann Taubert
Dennis Krischal**

Abgabedatum: 21. August 2024

Prof. Dr. Marcel Spehr

Kurzfassung

Diese Projektarbeit entwickelt eine Konzeptstudie zu einer Musik-Streaming-Plattform, die sich an der bekannten App Spotify orientiert. Die Arbeit zielt darauf ab, verschiedene Technologien, die in der Systemarchitektur solcher Plattformen eingesetzt werden, zu vergleichen und ihre Eignung für spezifische Anforderungen zu bewerten. Dies umfasst unter anderem Datenbankmanagementsysteme, Message Broker, In-Memory-Datenbanken, Fileserver und Authentifizierungsmechanismen. Anhand von Implementierungen und Tests werden die Vor- und Nachteile der verschiedenen Technologien aufgezeigt. Die Ergebnisse bieten wertvolle Erkenntnisse für die Auswahl der geeigneten Komponenten bei der Entwicklung von skalierbaren und leistungsfähigen Streaming-Plattformen.

Abstract

This project explores the architectural design and component selection for a music streaming platform inspired by Spotify. The study aims to compare various technologies commonly used in such systems, including database management systems, message brokers, in-memory databases, file servers, and authentication mechanisms. Through practical implementation and testing, the work evaluates the strengths and weaknesses of these technologies, providing insights into their suitability for specific requirements in scalable and high-performance streaming platforms. The findings contribute to the informed selection of components in the development of robust and efficient streaming services.

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Listings	VI
1 Einleitung	1
1.1 Zielsetzung	1
2 Grundlagen	2
2.1 Datenbanken Modul	2
2.1.1 MySQL	2
2.1.2 PostgreSQL	3
2.1.3 SQLite	3
2.1.4 MongoDB	4
2.1.5 Cassandra	5
2.2 Message Broker	5
2.2.1 Apache Kafka	6
2.2.2 RabbitMQ	7
2.2.3 Apache ActiveMQ	8
2.3 Authentifizierung und Autorisierung	9
2.3.1 OpenID Connect	10
2.3.2 SAML	11
2.3.3 OAuth2.0	12
2.4 Frameworks für API's	13
2.5 In-Memory-Datenbanken	13
2.5.1 Redis	14
2.5.2 Memcached	15
2.5.3 SAP HANA	16
2.6 Fileserver	17
2.6.1 Amazon S3	17
2.6.2 GlusterFS	18
2.6.3 Network File System	19
2.6.4 FileZilla	19
2.7 Infrastruktur als Code mit Terraform	20
2.7.1 Was ist Terraform	20
2.7.2 Grundprinzipien und Funktionsweise	21
2.7.3 Vorteile von Terraform	21
2.7.4 Beispiel einer Terraform-Konfiguration	21
2.7.5 Best Practices und Herausforderungen	22
2.7.6 Fazit	22

3	Konzept	23
3.1	Architektur layout	23
3.1.1	Datenbanken Vergleich	23
3.1.2	Aufbau des Vergleiches	24
3.1.3	Fileserver Vergleich	25
3.1.4	Cache Vergleich	26
3.1.5	Messagebroker	27
3.1.6	Authenticator	28
3.1.7	Loadbalancer	30
3.2	Konzept - Beschreibung	31
3.2.1	Nutzergruppen und deren Konzepte	31
3.2.2	Frontend - Benutzeroberflächen und Plattformen	32
3.2.3	Backend - Systemkomponenten und Infrastruktur	32
3.2.4	Top-K Problem	33
3.3	Konzept für labels	35
4	Umsetzung	36
4.1	Datenbackend	36
4.1.1	Postgree Cluster	36
4.1.2	MinIo Cluster	37
4.1.3	redis Db	37
4.2	Api	38
4.2.1	Komponenten der API	38
4.2.2	Kommunikation mit der Datenbank über EFCore	39
4.2.3	JWT-Authentifizierung	39
4.2.4	Aufsetzen des Dockers	40
4.3	Loadbalancer	41
4.4	Einführung	41
4.5	Konfiguration des Loadbalancers	41
4.5.1	Grundlegende Konfiguration	41
4.5.2	Round-Robin-Verfahren	42
4.5.3	Aufsetzen Mit Terraform	42
4.6	Frontend-UI	43
4.6.1	Login	43
4.6.2	Registrierung	44
4.6.3	Header	44
4.6.4	Suchergebnisse	44
4.6.5	Footer	45
4.7	Installationsanleitung zum testen des Systems	45
4.7.1	Clonen des Repositorys	45
5	Zusammenfassung und Ausblick	46

Abbildungsverzeichnis

3.1	System Architektur ¹	23
3.2	DB Vergleich ²	25
3.3	Docker Container Nummern ³	30
3.4	Ngnx Round-Robin Konfiguration ⁴	31
4.1	Login ⁵	43
4.2	Registration ⁶	44
4.3	Header ⁷	44
4.4	Header ⁸	44

Listings

2.1	Model der “counter“ Tabelle ⁹	21
3.1	NuGet Package Manager Console Commands	23
3.2	User Datenstruct	24
3.3	Erstellen der TestDaten	24
4.1	NGINX Loadbalancer Konfiguration	36
4.2	NGINX Loadbalancer Konfiguration	37
4.3	NGINX Loadbalancer Konfiguration	37
4.4	Dockerfile Asp.net	40
4.5	Terraform Script für 2 Api's	40
4.6	NGINX Loadbalancer Konfiguration	41
4.7	NGINX Loadbalancer Konfiguration	42

1 Einleitung

Im Rahmen dieses Projekts wurde eine hochperformante Musikstreaming-Plattform konzipiert und implementiert, die das Potenzial hat, die Art und Weise, wie wir Musik konsumieren, zu revolutionieren. Die Architektur basiert auf einem Microservices-Ansatz und nutzt einen Tech-Stack, der speziell auf die Anforderungen von skalierbaren, cloud-nativen Anwendungen zugeschnitten ist. Terraform ermöglicht eine effiziente Verwaltung der Infrastruktur, während .NET MAUI für die Entwicklung der Desktop-Anwendung zum Einsatz kommt. Die Kernfunktionalitäten der Plattform werden von einer ASP.NET Web API bereitgestellt, die auf zwei PostgreSQL-Datenbanken zugreift, um eine hohe Verfügbarkeit und Datenkonsistenz zu gewährleisten. Redis wird als In-Memory-Datenbank eingesetzt, um häufig benötigte Daten zu cachen und die Antwortzeiten zu minimieren. NGINX fungiert als Load Balancer und verteilt den eingehenden Traffic gleichmäßig auf die beiden API-Instanzen.

1.1 Zielsetzung

Im Rahmen dieser Arbeit haben wir uns zum Ziel gesetzt, eine Musikstreaming-Plattform zu entwickeln, die uns ermöglicht, die in unserem Studium erworbenen theoretischen Kenntnisse in die Praxis umzusetzen. Dabei stand im Vordergrund, ein tieferes Verständnis für die Herausforderungen bei der Entwicklung skalierbarer und hochverfügbarer Software-Systeme zu entwickeln. Außerdem haben wir verschiedene Technologien miteinander verglichen um eine möglichst gute Entscheidung, bei der zu verwendeten Technologien, zu treffen. Dabei haben wir uns auf die folgenden vier Aspekte konzentriert:

Skalierbarkeit: Wie können wir eine Anwendung gestalten, die mit einer steigenden Anzahl von Nutzern und Datenmengen problemlos zurechtkommt?

Wartbarkeit: Welche Maßnahmen sind erforderlich, um eine möglichst einfache Wartbarkeit der Plattform zu gewährleisten?

Ausfallsicherheit: Welche Strategien können eingesetzt werden, um die Auswirkungen von Ausfällen zu minimieren?

2 Grundlagen

2.1 Datenbanken Modul

Datenbankenmodule sind spezialisierte Softwarekomponenten, die für die Verwaltung, Speicherung und Abruf von Daten innerhalb eines Datenbanksystems verantwortlich sind. Sie bieten die notwendige Infrastruktur und Funktionalität, um Daten effizient und sicher zu verwalten. Datenbankenmodule umfassen eine Vielzahl von Technologien und Architekturen, die je nach Anwendungsfall und Anforderungen variieren können.

Im Folgenden wird auf spezifische Beispiele für Datenbankenmodule eingegangen, um die Vielfalt und unterschiedlichen Einsatzmöglichkeiten dieser Technologien zu veranschaulichen. Hierbei werden MySQL, PostgreSQL, SQLite, MongoDB und Cassandra betrachtet.

2.1.1 MySQL

MySQL ist ein relationales Datenbankmanagementsystem (RDBMS), das ursprünglich von der schwedischen Firma MySQL AB entwickelt wurde. Es wurde 2008 von Sun Microsystems übernommen, die später von Oracle Corporation gekauft wurden. MySQL ist eine der beliebtesten Datenbanken der Welt und wird von zahlreichen Webanwendungen und Plattformen wie WordPress, Joomla und Drupal verwendet.

Architektur und Design

MySQL folgt dem klassischen Client-Server-Modell. Es unterstützt verschiedene Speicher-Engines, die unterschiedliche Funktionalitäten und Leistungsmerkmale bieten, wie InnoDB und MyISAM. InnoDB ist besonders für seine Unterstützung von ACID-Transaktionen und Fremdschlüsselreferenzen bekannt.

Features

- **Skalierbarkeit und Flexibilität:** MySQL kann auf großen Systemen mit vielen CPUs und Speicher laufen und unterstützt eine große Anzahl von Tabellen und Datensätzen.
- **Sicherheit:** MySQL bietet robuste Sicherheitsmechanismen wie SSL-Unterstützung, Verschlüsselung und Benutzerrechteverwaltung.
- **Replikation:** MySQL unterstützt Master-Slave-Replikation sowie Multi-Master-Replikation für hohe Verfügbarkeit und Lastverteilung.

Einsatzgebiete

MySQL wird häufig in Webanwendungen, E-Commerce-Plattformen und datenintensiven Anwendungen verwendet. Durch seine weit verbreitete Verwendung und Unterstützung durch die Community ist es eine bevorzugte Wahl für viele Entwickler.

2.1.2 PostgreSQL

PostgreSQL ist ein objektrelationales Datenbankmanagementsystem (ORDBMS), das eine erweiterte Funktionalität gegenüber traditionellen RDBMS bietet. Es wurde in den 1980er Jahren an der University of California, Berkeley, entwickelt und ist heute eine Open-Source-Software mit einer großen und aktiven Community.

Architektur und Design

PostgreSQL ist für seine robuste Architektur bekannt. Es unterstützt komplexe Abfragen, Transaktionen, und erweiterbare Datentypen. Das System ist darauf ausgelegt, Datenintegrität und Zuverlässigkeit zu gewährleisten.

Features

- **Erweiterbarkeit:** PostgreSQL ermöglicht es Entwicklern, eigene Datentypen, Operatoren und Funktionen zu definieren.
- **SQL-Standards:** Es unterstützt eine breite Palette von SQL-Standards, einschließlich fortgeschrittener Funktionen wie Common Table Expressions (CTEs) und Window Functions.
- **Replikation und Hochverfügbarkeit:** PostgreSQL bietet sowohl synchrone als auch asynchrone Replikation und Tools wie pgpool-II und Patroni zur Unterstützung der Hochverfügbarkeit.

Einsatzgebiete

PostgreSQL wird oft in Bereichen eingesetzt, die eine hohe Datenintegrität und komplexe Abfragen erfordern, wie Finanzdienstleistungen, wissenschaftliche Anwendungen und Geoinformationssysteme (GIS).

2.1.3 SQLite

SQLite ist ein leichtgewichtiges, serverloses, selbstständiges SQL-Datenbankmanagementsystem. Es wurde von D. Richard Hipp entwickelt und ist in viele Anwendungen eingebettet, von Browsern bis hin zu mobilen Apps.

Architektur und Design

SQLite speichert die gesamte Datenbank in einer einzelnen Datei auf der Festplatte. Es ist ACID-konform und verwendet die Programmiersprache C für die Implementierung.

Features

- **Einfachheit und Leichtigkeit:** SQLite benötigt keine Installation oder Konfiguration. Es kann direkt in Anwendungen eingebettet werden.
- **Leistung:** Trotz seiner Leichtigkeit bietet SQLite beeindruckende Leistung für viele Anwendungen.
- **Portabilität:** Die gesamte Datenbank kann als eine Datei leicht kopiert, gesichert und übertragen werden.

Einsatzgebiete

SQLite wird häufig in Anwendungen verwendet, die eine einfache, eingebettete Datenbanklösung benötigen, wie mobile Apps, kleine Webanwendungen und eingebettete Systeme.

2.1.4 MongoDB

MongoDB ist ein NoSQL-Datenbankmanagementsystem, das 2007 von MongoDB Inc. (ehemals 10gen) entwickelt wurde. Es verwendet ein dokumentenorientiertes Datenmodell und bietet hohe Flexibilität und Skalierbarkeit.

Architektur und Design

MongoDB speichert Daten in BSON (Binary JSON) Dokumenten, die es ermöglichen, komplexe und verschachtelte Datenstrukturen abzubilden. Es unterstützt horizontale Skalierung durch Sharding und Replikation für hohe Verfügbarkeit.

Features

- **Flexibles Datenmodell:** MongoDB erlaubt das Speichern von heterogenen Datensätzen und dynamischen Schemas.
- **Skalierbarkeit:** Durch Sharding kann MongoDB große Datenmengen über viele Server hinweg verteilen.
- **Hohe Verfügbarkeit:** Automatische Replikation und Failover-Mechanismen gewährleisten die kontinuierliche Verfügbarkeit der Daten.

Einsatzgebiete

MongoDB wird oft in Big-Data-Anwendungen, Echtzeit-Analysen, Content-Management-Systemen und Internet-of-Things (IoT) Anwendungen verwendet.

2.1.5 Cassandra

Apache Cassandra ist ein verteiltes NoSQL-Datenbankmanagementsystem, das ursprünglich von Facebook entwickelt und später als Open-Source-Projekt an die Apache Software Foundation übergeben wurde. Es ist für seine hohe Skalierbarkeit und Verfügbarkeit bekannt.

Architektur und Design

Cassandra verwendet eine Masterless-Architektur, bei der alle Knoten gleichberechtigt sind und Daten in einem verteilten Hash-Tabellenmodell gespeichert werden. Dies ermöglicht eine hohe Fehlertoleranz und horizontale Skalierbarkeit.

Features

- **Skalierbarkeit:** Cassandra kann nahtlos skaliert werden, indem neue Knoten hinzugefügt werden, ohne dass Ausfallzeiten oder Leistungseinbußen auftreten.
- **Hohe Verfügbarkeit:** Dank seiner Masterless-Architektur und der Unterstützung für Multi-Datacenter-Replikation bietet Cassandra eine ausgezeichnete Verfügbarkeit.
- **Leistung:** Cassandra ist für schnelle Schreibvorgänge optimiert und kann große Mengen von Daten effizient verarbeiten.

Einsatzgebiete

Cassandra wird häufig in Anwendungen verwendet, die hohe Verfügbarkeit, schnelle Schreibvorgänge und Skalierbarkeit erfordern, wie Echtzeit-Datenanalyse, Messaging-Dienste und große E-Commerce-Plattformen.

2.2 Message Broker

Ein Message Broker ist eine Middleware, die das Senden und Empfangen von Nachrichten zwischen verteilten Systemen und Anwendungen erleichtert. Er übernimmt die Rolle eines Vermittlers, der Nachrichten von einem Produzenten entgegennimmt und sie an einen oder mehrere Konsumenten weiterleitet. Message Broker sind besonders nützlich, um Systeme zu entkoppeln, die Zuverlässigkeit der Kommunikation zu erhöhen und die Skalierbarkeit zu verbessern. Sie unterstützen verschiedene Kommunikationsmuster wie Punkt-zu-Punkt und Publish-Subscribe.

Im Folgenden werden drei prominente Beispiele für Message Broker vorgestellt: Apache Kafka, RabbitMQ und Apache ActiveMQ. Diese Systeme bieten unterschiedliche Funktionen und Stärken, die sie für verschiedene Anwendungsfälle und Anforderungen geeignet machen.

2.2.1 Apache Kafka

Apache Kafka ist eine verteilte Streaming-Plattform, die ursprünglich von LinkedIn entwickelt und später als Open-Source-Projekt der Apache Software Foundation bereitgestellt wurde. Kafka ist darauf ausgelegt, große Mengen an Daten in Echtzeit zu verarbeiten und zu übertragen. Es wird häufig für das Sammeln, Analysieren und Überwachen von Datenströmen verwendet und findet Anwendung in Bereichen wie Log-Analyse, Echtzeit-Überwachung und Event-Sourcing.

Architektur und Komponenten

Die Architektur von Kafka basiert auf einem Cluster-Design, das aus mehreren Brokern besteht, die zusammenarbeiten, um Daten zu verwalten. Die Hauptkomponenten von Kafka sind:

- **Producer:** Anwendungen, die Daten an Kafka-Topics senden.
- **Consumer:** Anwendungen, die Daten aus Kafka-Topics lesen.
- **Broker:** Server, die Daten speichern und verwalten.
- **Zookeeper:** Ein Koordinationsdienst, der zur Verwaltung von Kafka-Cluster-Metadaten verwendet wird.

Daten in Kafka werden in Topics organisiert, und jedes Topic ist in mehrere Partitions unterteilt, was eine parallele Verarbeitung und eine höhere Skalierbarkeit ermöglicht.

Hauptmerkmale

- **Hohe Durchsatzrate:** Kafka ist für die Handhabung von Millionen von Nachrichten pro Sekunde optimiert.
- **Skalierbarkeit:** Kafka kann horizontal skaliert werden, indem zusätzliche Broker hinzugefügt werden.
- **Persistenz:** Nachrichten werden auf der Festplatte gespeichert und können über einen konfigurierbaren Zeitraum aufbewahrt werden.
- **Fehler-Toleranz:** Kafka repliziert Daten über mehrere Broker hinweg, um Ausfallsicherheit zu gewährleisten.
- **Genau einmalige Zustellung:** Durch die Implementierung von Transaktionen stellt Kafka sicher, dass Nachrichten genau einmal zugestellt werden.

Anwendungsfälle

Apache Kafka eignet sich ideal für Szenarien, in denen eine hohe Durchsatzrate und Latenzanforderungen eine Rolle spielen:

- Echtzeit-Analyse von Datenströmen
- Log- und Ereignisüberwachung
- Integration von verschiedenen Datenquellen und Systemen
- Event-getriebene Architekturen

2.2.2 RabbitMQ

RabbitMQ ist ein weit verbreiteter Open-Source-Nachrichtenbroker, der auf dem Advanced Message Queuing Protocol (AMQP) basiert. RabbitMQ wurde von der Firma Pivotal Software entwickelt und ist bekannt für seine Flexibilität und Unterstützung mehrerer Messaging-Protokolle. Es bietet robuste Funktionen zur Verwaltung von Nachrichtenwarteschlangen und zur Implementierung verteilter Systeme.

Architektur und Komponenten

RabbitMQ verwendet eine Server-Client-Architektur, bei der der Server (Broker) die Nachrichten speichert und verwaltet, während Clients als Publisher oder Consumer fungieren. Die Hauptkomponenten von RabbitMQ sind:

- **Exchange:** Verteilt Nachrichten an die entsprechenden Warteschlangen basierend auf Routing-Regeln.
- **Queue:** Speicherung von Nachrichten, bis sie von einem Consumer abgerufen werden.
- **Binding:** Regeln, die die Verknüpfung von Exchanges und Queues definieren.
- **Publisher:** Anwendungen, die Nachrichten an Exchanges senden.
- **Consumer:** Anwendungen, die Nachrichten aus Queues empfangen und verarbeiten.

Hauptmerkmale

- **Flexibilität:** Unterstützung für verschiedene Messaging-Protokolle (z.B. AMQP, MQTT, STOMP).
- **Erweiterbarkeit:** RabbitMQ kann durch Plugins erweitert werden, um zusätzliche Funktionen zu bieten.
- **Zuverlässigkeit:** Nachrichten können persistent gespeichert werden, um Datenverluste zu vermeiden.
- **Fein abgestimmte Kontrolle:** Erlaubt detaillierte Konfigurationen für Routing, Persistenz und Zustellung von Nachrichten.

- **Management-UI:** Eine webbasierte Benutzeroberfläche zur Überwachung und Verwaltung des Systems.

Anwendungsfälle

RabbitMQ ist besonders nützlich in Szenarien, in denen Nachrichtenvermittlungs- und Warteschlangenverwaltung eine Rolle spielen:

- Asynchrone Verarbeitung von Aufgaben
- Lastverteilung von Arbeitseinheiten über mehrere Dienste
- Entkopplung von Anwendungen in Microservice-Architekturen
- Implementierung von Workflows und Geschäftsprozessen

2.2.3 Apache ActiveMQ

Apache ActiveMQ ist ein weiterer beliebter Open-Source-Nachrichtenbroker, der von der Apache Software Foundation entwickelt wurde. ActiveMQ unterstützt mehrere Messaging-Protokolle und ist bekannt für seine hohe Performance und Flexibilität. Es kann sowohl als klassischer Nachrichtenbroker als auch als Message-Orientierte-Middleware (MOM) eingesetzt werden.

Architektur und Komponenten

ActiveMQ verwendet ebenfalls eine Broker-basierte Architektur, bei der der Broker für das Management und die Speicherung der Nachrichten verantwortlich ist. Die Hauptkomponenten von ActiveMQ sind:

- **Broker:** Verwaltet Nachrichten, Verbindungen und Weiterleitungen.
- **Producer:** Anwendungen, die Nachrichten an den Broker senden.
- **Consumer:** Anwendungen, die Nachrichten vom Broker empfangen.
- **Destination:** Zielorte für Nachrichten, die als Queues (für Punkt-zu-Punkt-Kommunikation) oder Topics (für Publish-Subscribe-Kommunikation) konfiguriert werden können.
- **Transport Connector:** Ermöglicht die Kommunikation über verschiedene Protokolle (z.B. TCP, HTTP).

Hauptmerkmale

- **Protokollunterstützung:** Unterstützung für JMS (Java Message Service), AMQP, MQTT, STOMP und andere Protokolle.
- **Hohe Verfügbarkeit:** Kann in Clustern konfiguriert werden, um Ausfallsicherheit und Skalierbarkeit zu gewährleisten.
- **Persistente Speicherung:** Nachrichten können auf Festplatte gespeichert werden, um Datenverlust zu vermeiden.
- **Flexible Deployment-Optionen:** Kann als eigenständiger Server, eingebetteter Broker oder in der Cloud betrieben werden.
- **Management-Tools:** Web-basierte Konsole und JMX-Unterstützung zur Überwachung und Verwaltung des Brokers.

Anwendungsfälle

ActiveMQ ist vielseitig einsetzbar und eignet sich für verschiedene Anwendungsfälle:

- Integration von verschiedenen Systemen und Anwendungen
- Asynchrone Nachrichtenübermittlung
- Implementierung von Event-Driven-Architekturen
- Unterstützung von verteilten Systemen und Cloud-Anwendungen
- Anbindung und Verwaltung von IoT-Geräten

Insgesamt bieten Apache Kafka, RabbitMQ und Apache ActiveMQ leistungsstarke Lösungen für das Management von Nachrichten in verteilten Systemen. Jedes dieser Systeme hat seine eigenen Stärken und eignet sich für unterschiedliche Anwendungsfälle, abhängig von den spezifischen Anforderungen an Durchsatz, Latenz, Zuverlässigkeit und Skalierbarkeit.

2.3 Authentifizierung und Autorisierung

Mechanismen für Authentifizierung und Autorisierung sind entscheidende Sicherheitsprozesse, die sicherstellen, dass nur berechtigte Benutzer Zugriff auf bestimmte Ressourcen haben. Authentifizierung ist der Prozess, bei dem die Identität eines Benutzers überprüft wird, z.B. durch Eingabe eines Passworts oder mittels biometrischer Daten. Autorisierung bestimmt anschließend, welche Ressourcen oder Aktionen diesem authentifizierten Benutzer erlaubt sind, basierend auf vordefinierten Rechten und Rollen.

Im Folgenden wird auf Beispiele wie OpenID Connect (OIDC), Security Assertion Markup Language (SAML) und OAuth 2.0 referenziert, um diese Konzepte zu veranschaulichen. OIDC erweitert das OAuth 2.0-Protokoll und bietet benutzerfreundliche Authentifizierung, SAML ermöglicht Single Sign-On (SSO) und Austausch von Authentifizierungsinformationen zwischen verschiedenen Diensten, während OAuth 2.0 flexible Autorisierungsmechanismen bereitstellt, um Drittanwendungen den sicheren Zugriff auf Benutzerdaten zu erlauben.

2.3.1 OpenID Connect

OpenID Connect (OIDC) ist ein Authentifizierungsprotokoll, das auf dem OAuth 2.0-Protokoll aufbaut und von der OpenID Foundation entwickelt wurde. OIDC erweitert OAuth 2.0, um Endbenutzern eine einfache Möglichkeit zur Authentifizierung bei webbasierten und mobilen Anwendungen zu bieten. Während OAuth 2.0 primär für Autorisierung gedacht ist, fügt OIDC Mechanismen hinzu, um die Identität der Benutzer zu bestätigen.

Grundlagen

OIDC verwendet OAuth 2.0, um den Authentifizierungsprozess zu steuern. Der Prozess beginnt damit, dass der Benutzer bei einem Identity Provider (IdP) authentifiziert wird. Der IdP erstellt daraufhin ein ID-Token, das Informationen über den Benutzer enthält. Diese Informationen werden dann an die anfragende Anwendung (Client) zurückgesendet.

Hauptkomponenten

1. **ID-Token:** Ein JWT (JSON Web Token), das die Authentifizierungsinformationen des Benutzers enthält.
2. **UserInfo Endpoint:** Eine API, über die zusätzliche Informationen über den Benutzer abgefragt werden können.
3. **Authorization Endpoint:** Die URL, an die die Authentifizierungsanfragen gesendet werden.
4. **Token Endpoint:** Die URL, über die Access-Tokens angefordert werden können.

Ablauf einer Authentifizierung

1. Der Benutzer navigiert zur Anwendung, die eine Authentifizierung erfordert.
2. Die Anwendung leitet den Benutzer zum Authorization Endpoint des IdP weiter.
3. Der Benutzer authentifiziert sich beim IdP (z.B. durch Eingabe von Benutzername und Passwort).
4. Der IdP erstellt ein ID-Token und sendet es zurück an die Anwendung.
5. Die Anwendung überprüft das ID-Token und autorisiert den Zugriff basierend auf den enthaltenen Informationen.

Vorteile

- **Interoperabilität:** OIDC ist weit verbreitet und wird von vielen großen Identitätsanbietern unterstützt.
- **Erweiterbarkeit:** Dank des modularen Designs kann OIDC leicht an spezifische Anforderungen angepasst werden.
- **Sicherheit:** OIDC nutzt bewährte Sicherheitsmechanismen von OAuth 2.0 und ergänzt sie um zusätzliche Sicherheitsfeatures.
-

2.3.2 SAML

SAML (Security Assertion Markup Language) ist ein XML-basiertes Framework für die Austausch von Authentifizierungs- und Autorisierungsdaten zwischen Parteien, insbesondere zwischen einem Identity Provider (IdP) und einem Service Provider (SP). SAML ermöglicht Single Sign-On (SSO), indem es die Identität eines Benutzers von einem IdP an verschiedene SPs weiterleitet.

Grundlagen

SAML definiert drei Rollen:

1. **Principal:** Der Benutzer, der sich authentifizieren möchte.
2. **Identity Provider (IdP):** Die Instanz, die die Identität des Benutzers bestätigt.
3. **Service Provider (SP):** Die Anwendung oder der Dienst, der auf die Identität des Benutzers zugreift.

Hauptkomponenten

1. **Assertions:** XML-Dokumente, die Authentifizierungs- und Autorisierungsinformationen enthalten.
2. **Protocol:** Regeln für den Austausch von Assertions.
3. **Bindings:** Methoden zur Übertragung der Assertions (z.B. HTTP POST, HTTP Redirect).
4. **Profiles:** Spezielle Implementierungen für bestimmte Anwendungsfälle (z.B. Web Browser SSO Profile).

Ablauf einer Authentifizierung

1. Der Benutzer navigiert zu einem SP und versucht, auf eine Ressource zuzugreifen.
2. Der SP leitet den Benutzer zum IdP weiter.
3. Der Benutzer authentifiziert sich beim IdP.
4. Der IdP erstellt eine SAML-Assertion und sendet sie zurück an den SP.
5. Der SP validiert die Assertion und gewährt dem Benutzer Zugriff auf die Ressource.

Vorteile

- **Interoperabilität:** SAML wird von vielen Unternehmen und Organisationen genutzt und ist ein Industriestandard für SSO.
- **Sicherheit:** SAML bietet starke Sicherheitsfeatures, darunter digitale Signaturen und Verschlüsselung.
- **Flexibilität:** SAML unterstützt verschiedene Authentifizierungs- und Autorisierungsszenarien.

2.3.3 OAuth2.0

OAuth 2.0 ist ein Autorisierungsframework, das Drittanwendungen den Zugriff auf Ressourcen eines Benutzers ermöglicht, ohne dessen Passwort zu teilen. OAuth 2.0 ist weit verbreitet und bildet die Grundlage für viele moderne Authentifizierungs- und Autorisierungslösungen, einschließlich OIDC.

Grundlagen

OAuth 2.0 definiert vier Rollen:

1. **Resource Owner:** Der Benutzer, der die Ressourcen besitzt.
2. **Client:** Die Anwendung, die Zugriff auf die Ressourcen benötigt.
3. **Resource Server:** Der Server, der die Ressourcen hostet.
4. **Authorization Server:** Der Server, der die Autorisierung durchführt und Tokens ausgibt.

Hauptkomponenten

1. **Access Token:** Ein Token, das den Zugriff auf die Ressourcen ermöglicht.
2. **Authorization Grant:** Ein Berechtigungsnachweis, der dem Client den Erhalt eines Access Tokens erlaubt.
3. **Authorization Endpoint:** Die URL, an die der Benutzer zur Autorisierung weitergeleitet wird.
4. **Token Endpoint:** Die URL, über die der Client Access Tokens anfordert.

Ablauf einer Authentifizierung

1. Der Benutzer gibt dem Client die Erlaubnis, auf seine Ressourcen zuzugreifen (z.B. durch Einwilligung bei einer OAuth-Dialogbox).
2. Der Client erhält einen Authorization Grant und tauscht diesen gegen ein Access Token beim Authorization Server ein.
3. Der Client verwendet das Access Token, um auf die Ressourcen des Benutzers auf dem Resource Server zuzugreifen.

Vorteile

- **Flexibilität:** OAuth 2.0 kann in einer Vielzahl von Szenarien verwendet werden, einschließlich Webanwendungen, mobile Apps und IoT.
- **Sicherheit:** Durch die Trennung von Autorisierung und Authentifizierung können Passwörter sicher bleiben.
- **Interoperabilität:** OAuth 2.0 ist ein weit akzeptierter Standard und wird von vielen großen Plattformen unterstützt.

2.4 Frameworks für API's

2.5 In-Memory-Datenbanken

In-Memory-Datenbanken sind spezielle Datenbanksysteme, die Daten vollständig im Arbeitsspeicher (RAM) speichern, anstatt auf Festplatten oder anderen permanenten Speichergeräten. Dies ermöglicht extrem schnelle Lese- und Schreibvorgänge, da der Zugriff auf Daten im RAM wesentlich schneller ist als der Zugriff auf Daten auf herkömmlichen Festplatten. In-Memory-Datenbanken sind ideal für Anwendungen, die sehr hohe Performance und geringe Latenzzeiten erfordern, wie Echtzeit-Datenanalysen, Caching, Sitzungsmanagement und mehr.

Im Folgenden werden wir drei prominente Beispiele für In-Memory-Datenbanken näher betrachten: Redis, Memcached und SAP HANA. Diese Beispiele verdeutlichen die Vielfalt und

Flexibilität von In-Memory-Datenbanken und zeigen, wie sie in verschiedenen Anwendungsfällen eingesetzt werden können, um die Effizienz und Geschwindigkeit von Datenverarbeitungsprozessen zu optimieren.

2.5.1 Redis

Redis (Remote Dictionary Server) ist eine Open-Source, in-memory Datenstruktur-Speicher, der als Datenbank, Cache und Message Broker verwendet wird. Es wurde von Salvatore Sanfilippo entwickelt und ist bekannt für seine hohe Performance, Flexibilität und Unterstützung für verschiedene Datenstrukturen.

Architektur und Funktionsweise

Redis speichert Daten vollständig im Arbeitsspeicher, was extrem schnelle Lese- und Schreibvorgänge ermöglicht. Es verwendet eine einfache Schlüssel-Wert-Speicherarchitektur, unterstützt jedoch auch komplexere Datenstrukturen wie Listen, Mengen, geordnete Mengen, Hashes, Bitmaps und HyperLogLogs.

Redis kann auf einem Einzelserver laufen oder in einer verteilten Umgebung konfiguriert werden, um hohe Verfügbarkeit und Skalierbarkeit zu gewährleisten. Es bietet eine Vielzahl von Mechanismen zur Datenpersistenz, darunter Snapshots (RDB) und Append-only Files (AOF).

Anwendungsfälle

- **Caching:** Redis wird häufig zur Zwischenspeicherung von Daten verwendet, um die Lesezugriffsgeschwindigkeit zu erhöhen und die Last auf Backend-Datenbanken zu reduzieren.
- **Session Management:** Aufgrund seiner schnellen Zugriffsgeschwindigkeit ist Redis ideal für das Speichern von Sitzungsdaten in Webanwendungen.
- **Real-Time Analytics:** Die Fähigkeit von Redis, komplexe Datenstrukturen zu handhaben, macht es nützlich für Echtzeitanalysen und die Verarbeitung von Streaming-Daten.
- **Message Queuing:** Redis kann als Message Broker verwendet werden, um Nachrichten zwischen verschiedenen Diensten zu verteilen.

Vorteile und Herausforderungen

Vorteile

- Sehr hohe Geschwindigkeit und niedrige Latenzzeiten.
- Unterstützung für verschiedene Datenstrukturen.
- Einfache Skalierbarkeit und hohe Verfügbarkeit.

Nachteile

- Da Redis im Arbeitsspeicher arbeitet, kann es bei großen Datenmengen teuer werden.
- Die Verwaltung der Persistenzmechanismen kann komplex sein.

2.5.2 Memcached

Memcached ist ein verteiltes In-Memory-Caching-System, das hauptsächlich zur Verbesserung der Anwendungsleistung durch Reduzierung der Datenbanklast verwendet wird. Es wurde ursprünglich von Danga Interactive für LiveJournal entwickelt, ist aber mittlerweile weit verbreitet in verschiedenen Webanwendungen und Diensten.

Architektur und Funktionsweise

Memcached speichert Daten als einfache Schlüssel-Wert-Paare im Arbeitsspeicher. Es verwendet ein einfaches Protokoll für das Setzen, Abrufen und Löschen von Werten, was es extrem schnell und effizient macht. Memcached ist darauf ausgelegt, horizontal skalierbar zu sein, indem es mehrere Server in einem Cluster kombiniert, um mehr Speicherplatz und höhere Verfügbarkeit zu bieten.

Die Daten in Memcached sind flüchtig, was bedeutet, dass sie bei einem Neustart des Servers verloren gehen. Es bietet keine eingebaute Datenpersistenz oder erweiterte Datenstrukturen, was es einfacher und schneller macht, aber auch weniger flexibel im Vergleich zu anderen In-Memory-Datenbanken wie Redis.

Anwendungsfälle

- **Web Caching:** Memcached wird häufig verwendet, um häufig abgefragte Daten wie HTML-Seiten, API-Antworten und Datenbankabfragen zwischenspeichern, um die Ladezeiten zu verkürzen.
- **Datenbankentlastung:** Durch das Zwischenspeichern von Abfrageergebnissen kann Memcached die Last auf Backend-Datenbanken erheblich reduzieren und deren Leistung verbessern.
- **Session Storage:** In Webanwendungen wird Memcached oft zur Speicherung von Benutzersitzungsdaten verwendet.

Vorteile und Herausforderungen

Vorteile

- Sehr hohe Geschwindigkeit und niedrige Latenzzeiten.
- Einfache Implementierung und Verwaltung.
- Horizontale Skalierbarkeit.

Nachteile

- Flüchtigkeit der Daten, was bedeutet, dass Daten bei einem Serverneustart verloren gehen.
- Unterstützung nur für einfache Schlüssel-Wert-Paare und keine komplexen Datenstrukturen.

2.5.3 SAP HANA

SAP HANA (High-Performance Analytic Appliance) ist eine In-Memory-Datenbank- und Anwendungsplattform von SAP, die für die Verarbeitung großer Datenmengen in Echtzeit entwickelt wurde. HANA kombiniert Transaktions- und Analytik-Funktionen in einer einzigen In-Memory-Datenbank.

Architektur und Funktionsweise

SAP HANA verwendet eine spaltenorientierte In-Memory-Datenbankarchitektur, die speziell für hohe Datenverarbeitungs- und Abfragegeschwindigkeiten optimiert ist. Daten werden im Arbeitsspeicher gehalten, was extrem schnelle Zugriffszeiten ermöglicht. HANA unterstützt sowohl OLTP (Online Transaction Processing) als auch OLAP (Online Analytical Processing) in einem einzigen System, wodurch eine Echtzeit-Datenanalyse möglich wird.

Die Architektur von HANA umfasst auch erweiterte Funktionen wie Datenkomprimierung, Multicore-Prozessor-Unterstützung und parallele Verarbeitung. Dies ermöglicht es HANA, große Datenmengen effizient zu verarbeiten und komplexe Abfragen schnell auszuführen.

Anwendungsfälle

- **Echtzeit-Analytik:** HANA wird häufig für Echtzeit-Datenanalysen in verschiedenen Branchen eingesetzt, einschließlich Finanzen, Einzelhandel und Gesundheitswesen.
- **Datenintegration:** HANA bietet Funktionen zur Integration und Verwaltung von Daten aus verschiedenen Quellen, was es zu einer umfassenden Plattform für Datenmanagement und -analyse macht.
- **Datenintegration:** SAP HANA ist die Grundlage für viele SAP-Anwendungen, einschließlich S/4HANA, und ermöglicht die Verarbeitung großer Mengen transaktionaler und analytischer Daten in Echtzeit.

Vorteile und Herausforderungen

Vorteile

- Extrem schnelle Datenverarbeitung und Abfragegeschwindigkeit.
- Unterstützung für sowohl transaktionale als auch analytische Workloads.
- Integration und Verwaltung großer Datenmengen aus verschiedenen Quellen.

Nachteile

- Hohe Kosten für Implementierung und Betrieb.
- Komplexität der Verwaltung und Wartung.
- Erfordert spezielle Hardware und umfangreiche Schulung für Benutzer und Administratoren.

2.6 Fileserver

Ein Fileserver ist ein zentraler Server in einem Netzwerk, der Benutzern Speicherplatz für Dateien zur Verfügung stellt und den Zugriff auf diese Dateien verwaltet. Er ermöglicht es mehreren Benutzern, von verschiedenen Geräten aus auf die gleichen Dateien zuzugreifen und diese gemeinsam zu nutzen, was die Zusammenarbeit und Datenverwaltung erheblich erleichtert.

Im Folgenden werden verschiedene Beispiele von Fileserver-Lösungen näher erläutert: Amazon S3 ist ein skalierbarer Objektspeicher-Service von Amazon Web Services, der durch hohe Verfügbarkeit und Sicherheit besticht. GlusterFS hingegen ist ein verteiltes Dateisystem, das große Datenmengen über verschiedene Speicherressourcen hinweg verwaltet und sich durch hohe Skalierbarkeit und Flexibilität auszeichnet. NFS (Network File System) ist ein Dateisystemprotokoll, das den Zugriff auf Dateien über ein Netzwerk so ermöglicht, als wären sie lokal gespeichert. FileZilla schließlich ist ein freier FTP-Client und -Server, der die Übertragung von Dateien über das Internet unterstützt und sich durch Benutzerfreundlichkeit und Sicherheit auszeichnet.

Diese Beispiele bieten verschiedene Lösungen für unterschiedliche Anforderungen und Szenarien im Bereich der Dateispeicherung und -verwaltung.

2.6.1 Amazon S3

Amazon Simple Storage Service (Amazon S3) ist ein skalierbarer Objektspeicher-Service, der von Amazon Web Services (AWS) angeboten wird. Es ermöglicht Unternehmen und Entwicklern, Daten in nahezu unbegrenzter Menge sicher zu speichern, zu verwalten und abzurufen.

Architektur und Funktionsweise

Amazon S3 basiert auf dem Konzept von Buckets und Objekten. Ein Bucket ist ein Container für Objekte, die die eigentlichen Daten darstellen. Jedes Objekt besteht aus den Daten, Metadaten und einer eindeutigen Kennung.

Vorteile

- **Skalierbarkeit:** S3 skaliert automatisch mit dem Datenwachstum.
- **Sicherheit:** Daten können verschlüsselt und Zugriffsrechte granular definiert werden.
- **Verfügbarkeit und Langlebigkeit:** AWS garantiert eine hohe Verfügbarkeit und Datenhaltbarkeit.
- **Integration:** S3 integriert sich nahtlos mit anderen AWS-Diensten wie Lambda, EC2 und Glacier.

Typische Anwendungsfälle

- **Datensicherung und Archivierung:** Unternehmen nutzen S3 zur Sicherung und langfristigen Archivierung großer Datenmengen.
- **Webhosting:** S3 kann als Host für statische Websites dienen.
- **Big Data:** Speicherung und Analyse großer Datenmengen für Data-Warehouse- und Machine-Learning-Anwendungen.

2.6.2 GlusterFS

GlusterFS ist ein verteiltes Dateisystem, das darauf abzielt, große Datenmengen über verschiedene Speicherressourcen hinweg zu verwalten. Es wird hauptsächlich in großen IT-Umgebungen und Rechenzentren eingesetzt.

Architektur und Funktionsweise

GlusterFS verwendet eine skalierbare Architektur, die auf sogenannten Bricks basiert, wobei jeder Brick eine Speichereinheit (normalerweise ein Verzeichnis auf einem Server) darstellt. Diese Bricks werden zu einem GlusterFS-Volume zusammengefasst.

Vorteile

- **Skalierbarkeit:** Durch das Hinzufügen von Bricks kann die Kapazität und Leistung nahezu linear erweitert werden.
- **Hochverfügbarkeit:** Daten werden redundant gespeichert, um Ausfallsicherheit zu gewährleisten.
- **Flexibilität:** Unterstützt verschiedene Konfigurationsoptionen, einschließlich Replikation und Verteilung von Daten.

Typische Anwendungsfälle

- **Virtualisierung:** Speicherlösungen für virtuelle Maschinen in Rechenzentren.
- **Big Data und Analyse:** Speicherung und Verwaltung großer Datenmengen in verteilten Systemen.
- **Medien und Streaming:** Bereitstellung großer Multimedia-Dateien und Streaming-Inhalte.

2.6.3 Network File System

Network File System (NFS) ist ein Dateisystemprotokoll, das von Sun Microsystems entwickelt wurde und es ermöglicht, Dateien über ein Netzwerk so zu verwenden, als wären sie auf der lokalen Festplatte gespeichert.

Architektur und Funktionsweise

NFS verwendet das Client-Server-Modell, bei dem der NFS-Server den Dateizugriff auf seinen Speichermedien bereitstellt und NFS-Clients diese Dateien über das Netzwerk nutzen können.

Einsatzmöglichkeiten

- **Einfachheit:** Leicht zu implementieren und zu verwalten.
- **Kompatibilität:** Unterstützt eine Vielzahl von Betriebssystemen und Netzwerkkombinationen.
- **Transparenz:** Bietet transparenten Dateizugriff über das Netzwerk.

Typische Anwendungsfälle

- **Home Directories:** Zentrale Verwaltung von Benutzerverzeichnissen in Unternehmensnetzwerken.
- **Datenspeicherung:** Bereitstellung gemeinsamer Speicherbereiche für Anwendungen und Daten.
- **Backup und Recovery:** Nutzung als Ziel für Backup-Daten und Wiederherstellungsszenarien.

2.6.4 FileZilla

FileZilla ist ein freier und plattformunabhängiger FTP-Client und -Server, der zur Übertragung von Dateien über das Internet genutzt wird. Es unterstützt verschiedene Protokolle wie FTP, FTPS und SFTP.

Architektur und Funktionsweise

FileZilla besteht aus zwei Hauptkomponenten: dem FileZilla Client und dem FileZilla Server. Der Client ermöglicht den Zugriff auf entfernte Dateisysteme, während der Server den Zugriff auf lokale Dateien von entfernten Standorten aus ermöglicht.

Vorteile

- **Benutzerfreundlichkeit:** Intuitive Benutzeroberfläche und einfache Konfiguration.
- **Sicherheit:** Unterstützung von sicheren Protokollen wie FTPS und SFTP.
- **Plattformunabhängigkeit:** Verfügbar für verschiedene Betriebssysteme, darunter Windows, macOS und Linux.

Typische Anwendungsfälle

- **Website-Management:** Hochladen und Verwalten von Dateien auf Webservern.
- **Datentransfer:** Sichere Übertragung von Dateien zwischen verschiedenen Systemen.
- **Remote-Zugriff:** Zugriff auf Dateien von entfernten Standorten aus zur Verwaltung und Bearbeitung.

2.7 Infrastruktur als Code mit Terraform

Infrastruktur als Code (IaC) ist ein Konzept, das die Bereitstellung und Verwaltung von IT-Infrastrukturen durch maschinenlesbare Definitionsdateien ermöglicht, anstatt durch physische Hardwarekonfiguration oder interaktive Konfigurationstools. Dieses Paradigma ermöglicht es DevOps-Teams, Infrastruktur in Versionierungssysteme einzubinden, Änderungen nachzuvollziehen und automatisiert in verschiedenen Umgebungen auszurollen. Ein prominentes Werkzeug für IaC ist Terraform.

2.7.1 Was ist Terraform

Terraform ist ein Open-Source-Werkzeug, das von HashiCorp entwickelt wurde. Es ermöglicht Benutzern, Infrastrukturressourcen wie virtuelle Maschinen, Netzwerke und Speicher über eine deklarative Konfigurationssprache, die HashiCorp Configuration Language (HCL), zu definieren und zu verwalten. Terraform unterstützt eine Vielzahl von Anbietern, darunter große Cloud-Anbieter wie AWS, Azure und Google Cloud, sowie On-Premise-Lösungen.

2.7.2 Grundprinzipien und Funktionsweise

Terraform arbeitet nach dem Prinzip der Deklaration von Endzuständen. Benutzer definieren in Konfigurationsdateien den gewünschten Zustand ihrer Infrastruktur. Terraform interpretiert diese Dateien und erzeugt einen Ausführungsplan, der die notwendigen Schritte enthält, um den aktuellen Zustand der Infrastruktur in den gewünschten Zustand zu überführen. Die Hauptkomponenten von Terraform sind:

- **Provider:** Stellen die Verbindung zu Infrastrukturplattformen her und bieten Ressourcen an, die verwaltet werden können.
- **Ressourcen:** Elementare Einheiten der Infrastruktur, die durch Terraform verwaltet werden, wie z.B. virtuelle Maschinen oder Datenbanken.
- **Module:** Wiederverwendbare und strukturierte Gruppen von Ressourcen, die komplexe Infrastrukturen abstrahieren und verwalten.

2.7.3 Vorteile von Terraform

Terraform bietet mehrere Vorteile für die Verwaltung moderner IT-Infrastrukturen:

- **Plattformunabhängigkeit:** Unterstützt zahlreiche Cloud- und On-Premise-Provider.
- **Versionierung:** Infrastrukturkonfigurationen können versioniert und wie Quellcode verwaltet werden.
- **Automatisierung:** Reduziert menschliche Fehler und steigert die Effizienz durch Automatisierung.
- **Konsistenz:** Stellt sicher, dass alle Umgebungen identisch konfiguriert sind.
- **Skalierbarkeit:** Erleichtert die Verwaltung großer und komplexer Infrastrukturen.

2.7.4 Beispiel einer Terraform-Konfiguration

Ein einfaches Beispiel für eine Terraform-Konfigurationsdatei, die eine EC2-Instanz in AWS erstellt, sieht wie folgt aus:

```
1  provider "aws" {
2      region = "us-west-2"
3  }
4
5  resource "aws_instance" "example" {
6      ami           = "ami-0c55b159cbfafa1f0"
7      instance_type = "t2.micro"
8
9      tags = {
10         Name = "example-instance"
11     }
12  }
13
```

Listing 2.1: Model der “counter“ Tabelle ¹

In dieser Konfiguration wird ein AWS-Provider definiert und eine EC2-Instanz mit spezifischen Attributen erstellt. Die Konfigurationssprache HCL ermöglicht eine klare und lesbare Struktur, die leicht erweiterbar und wartbar ist.

2.7.5 Best Practices und Herausforderungen

Bei der Nutzung von Terraform sollten einige Best Practices beachtet werden:

- **Modularisierung:** Komplexe Konfigurationen sollten in Module aufgeteilt werden, um Wiederverwendbarkeit und Wartbarkeit zu erhöhen.
- **State Management:** Terraform speichert den Zustand der Infrastruktur in einer State-Datei. Diese Datei sollte sicher verwaltet und regelmäßig gesichert werden.
- **Versionskontrolle:** Terraform-Konfigurationsdateien sollten in Versionskontrollsystemen wie Git verwaltet werden.
- **Testing:** Infrastrukturänderungen sollten vor dem Ausrollen in Produktionsumgebungen gründlich getestet werden.

Gleichzeitig gibt es Herausforderungen, wie z.B. die Handhabung von Sensordaten, die Sicherheit der State-Datei und die Koordination von Teams, die parallel an der Infrastruktur arbeiten.

2.7.6 Fazit

Terraform hat sich als mächtiges Werkzeug für die Verwaltung von Infrastruktur etabliert. Durch seine deklarative Natur, die Unterstützung vieler Plattformen und die Möglichkeit der Automatisierung bietet es erhebliche Vorteile für moderne DevOps-Teams. Trotz der Herausforderungen, die mit der Verwaltung und Skalierung komplexer Infrastrukturen verbunden sind, ist Terraform ein unverzichtbares Werkzeug für die Umsetzung von Infrastruktur als Code und die Erreichung von Effizienz und Konsistenz in IT-Betriebsprozessen.

¹Quelle: Selbsterstellte Abbildung

3 Konzept

3.1 Architektur layout

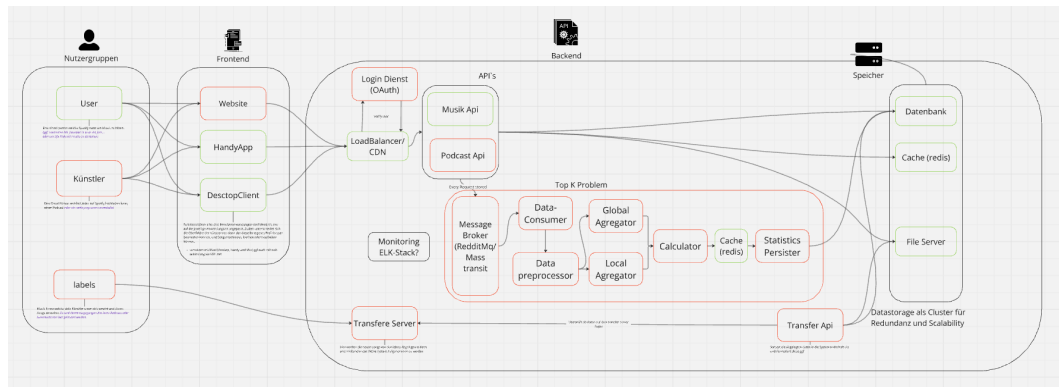


Abbildung 3.1: System Architektur¹

3.1.1 Datenbanken Vergleich

In diesem Kapitel wird das Modul der Datenpersistierung betrachtet. Hierfür werden die Im Kapitel 2.1 beschriebenen Datenbank Modelle verglichen. Im Folgenden wird aber auf die SQLite Datenbank verzichtet, da bei dieser sehr große Setup Schwierigkeiten aufgetreten sind.

Verwendete Pakete

Um den folgenden test durchführen zu können Müssen im projekt die pakete zur Kommunikation mit den datenbanken systemen insstaliert werden. Dafür müssen folgende befehle in der NuGet-Paket-Manager-Console ein:

```

1      Install-Package CassandraSharpDriver
2
3      Install-Package MongoDB.Driver
4      Install-Package MongoDB.Bson
5
6      Install-Package MySql.Data
7
8      Install-Package Npgsql
    
```

Listing 3.1: NuGet Package Manager Console Commands

¹Quelle: Selbsterstellte Abbildung

3.1.2 Aufbau des Vergleiches

Um alle Datenbanken sinnvoll miteinander vergleichen zu können, werden überall die gleichen Daten gespeichert. Wir haben uns hier für einen einfachen User-Datensatz entschieden, bestehend aus einer UserID und einem UserName. Um die User-Daten einfach generieren zu können, besteht der UserName lediglich aus dem Wort Üserü und der UserID.

```
1 public struct User
2 {
3
4     public int Id { get; set; }
5     public string Name { get; set; }
6
7     public User(int id, string name)
8     {
9         Id = id;
10        Name = name;
11    }
12
13 }
```

Listing 3.2: User Datenstruct

Zu Beginn des Tests werden 5.000 Test Datensätze erstellt.

```
1 List<User> _users = new List<User>();
2 int[] getList = new int[datacapGet];
3
4 Console.WriteLine("Start Preparing Date");
5
6 for (int i = 0; i < datacapSafe; i++)
7 {
8     User thmpUser = new(i, "User" + i);
9     _users.Add(thmpUser);
10 }
11 Random random = new Random();
12
13 for (int j = 0; j < datacapGet; j++)
14 {
15     getList[j] = random.Next(0, datacapSafe);
16 }
17
18 Console.WriteLine("End prepare data");
19
```

Listing 3.3: Erstellen der TestDaten

Im obigen Codebeispiel werden die 5.000 User generiert und ein Array mit zufälligen User-IDs im Rahmen der generierten User-IDs erstellt.

Die generierten Daten werden alle nacheinander in die Datenbank gespeichert und auch wieder ausgelesen. Dabei werden die Zeiten gestoppt und anschließend werden die gesammelten Ergebnisse gegenübergestellt.

Cassandra	Safe	Time: 13,476414
MongoDB	Safe	Time: 4,6615249
MySql	Safe	Time: 17,5786693
PostGress	Safe	Time: 11,9587992

Cassandra	Get	Time: 11,9411682
MongoDB	Get	Time: 9,8808754
MySql	Get	Time: 15,7678724
PostGress	Get	Time: 4,7328605

Abbildung 3.2: DB Vergleich²

Im Bild ist zu sehen, dass MongoDB mit insgesamt 14,54 Sekunden die schnellste aller getesteten Datenbanken ist, gefolgt von PostgreSQL mit 16,68 Sekunden. Die anderen beiden Datenbanksysteme haben einen deutlich größeren Abstand. Trotz der höheren Geschwindigkeit von MongoDB haben wir uns für PostgreSQL entschieden, da wir beim Schreiben der Tests weniger Probleme mit der Herstellung der Verbindung und dem Aufsetzen des Docker Containers hatten. Zudem wird PostgreSQL vom Entity Framework Core unterstützt, was das Einrichten einer komplexeren relationalen Datenbank erheblich vereinfacht. Das Entity Framework Core unterstützt zudem die Basis-Kommunikation sowie die Versionierung der Datenbank von Haus aus, sodass während der Entwicklung kein SQL verwendet werden muss.

3.1.3 Fileserver Vergleich

Der Vergleich von Lasstests für verschiedene Fileserverssysteme wie MinIO, GlusterFS und NFS macht unter bestimmten Bedingungen wenig Sinn, da die Eignung eines Systems stark von den spezifischen Anforderungen und Einsatzszenarien abhängt. Lasstests messen in erster Linie die Leistungsfähigkeit eines Systems hinsichtlich der Verarbeitungsgeschwindigkeit und des Durchsatzes, bieten jedoch keine umfassende Bewertung der Gesamteignung oder -effizienz eines Systems im realen Einsatz. Jedes Fileserverssystem bringt seine eigenen Stärken und Schwächen mit, die sich nicht immer in den Ergebnissen eines standardisierten Tests widerspiegeln.

MinIO, GlusterFS und NFS haben unterschiedliche Architekturen und Anwendungsfälle. MinIO ist ein hochskalierbares, objektbasiertes Speichersystem, das für Cloud-native Anwendungen und große Datenmengen optimiert ist. Es bietet hohe Verfügbarkeit und hohe Leistung für objektbasierte Speicherung und eignet sich gut für moderne, verteilte Systeme. GlusterFS hingegen ist ein verteiltes, Netzwerkdateisystem, das eine skalierbare und redundante Speicherlösung für große Datenmengen und verteilte Workloads bietet. NFS, als älteres Protokoll, ist darauf ausgelegt, als Netzwerkdateisystem in weniger verteilten und komplexen Umgebungen zu fungieren und bietet eine solide, aber oft weniger skalierbare Lösung für Netzwerkdateisysteme.

Der Einsatz eines Lasstests könnte daher irreführende Ergebnisse liefern, die nicht unbedingt die praktischen Anforderungen und die Eignung der jeweiligen Systeme für bestimmte

²Quelle: Selbsterstellte Abbildung

Aufgaben widerspiegeln. Zum Beispiel könnte ein Lasttest MinIO in einem sehr leistungsstarken, hochverfügbaren Setup zeigen, während GlusterFS in einem verteilten Szenario mit vielen Knoten besser abschneiden könnte. In der Praxis spielen jedoch auch Faktoren wie Benutzerfreundlichkeit, Integration in bestehende Systeme, Unterstützung für spezifische Features und Kosten eine entscheidende Rolle bei der Auswahl des geeigneten Systems.

Nach sorgfältiger Überlegung und Bewertung haben wir uns letztlich für MinIO entschieden. Der Hauptgrund für diese Wahl war die hohe Flexibilität und Skalierbarkeit, die MinIO für unsere Anwendungen bietet. Die Fähigkeit, große Mengen an objektbasierten Daten effizient zu verwalten und die einfache Integration in moderne DevOps-Umgebungen waren entscheidend. Außerdem bietet MinIO eine benutzerfreundliche API und umfangreiche Funktionen für Datenmanagement und -sicherheit, die ideal zu unseren Anforderungen passen und schnell und einfach zu Implementieren waren. Diese praktischen Vorteile haben sich als wichtiger erwiesen.

3.1.4 Cache Vergleich

Im Rahmen meines Projekts habe ich einen Vergleich von drei führenden In-Memory-Datenbanken durchgeführt: Redis, Memcached und Hazelcast. Diese Technologien wurden hinsichtlich ihrer Leistung, Flexibilität und Einsatzmöglichkeiten analysiert, um die bestmögliche Lösung für unser Anwendungsgebiet zu identifizieren. Jede dieser In-Memory-Datenbanken bringt spezifische Vor- und Nachteile mit sich, weshalb eine fundierte Auswahl getroffen werden muss, basierend auf den Anforderungen des Systems.

Redis ist eine der am weitesten verbreiteten In-Memory-Datenbanken und zeichnet sich durch seine Vielseitigkeit und Funktionsvielfalt aus. Es bietet Unterstützung für verschiedene Datentypen, darunter Strings, Hashes, Listen, Mengen und geordnete Mengen. Darüber hinaus verfügt Redis über eingebaute Mechanismen zur Replikation, Persistenz sowie über eine hohe Verfügbarkeit und automatische Failover-Mechanismen. Ein besonders herausragendes Merkmal von Redis ist seine Unterstützung für atomare Operationen, wodurch komplexe Anwendungslogik direkt auf Datenebene effizient umgesetzt werden kann. Diese Funktionen machen Redis zur bevorzugten Wahl für Anwendungsfälle, bei denen nicht nur ein einfacher Cache, sondern auch komplexe Datenstrukturen und konsistente Datenmanipulationen erforderlich sind.

Memcached hingegen ist ein äußerst leichtgewichtiges und performantes Caching-System, das sich durch seine Einfachheit und Geschwindigkeit auszeichnet. Es wurde speziell für das Caching von Schlüssel-Wert-Paaren entwickelt und bietet im Gegensatz zu Redis eine eingeschränktere Funktionalität, da es nur Strings als Datentyp unterstützt. Aufgrund dieser Einfachheit ist Memcached besonders effizient in Szenarien, in denen keine komplexen Datenoperationen erforderlich sind, sondern lediglich große Mengen an Daten schnell im Speicher gehalten werden müssen. Memcached skaliert sehr gut horizontal, was es zu einer ausgezeichneten Wahl für den Einsatz als reines Cache-System macht. Allerdings fehlen ihm fortgeschrittene Funktionen wie Persistenz oder die Möglichkeit zur Replikation, was es weniger robust im Vergleich zu Redis macht, insbesondere wenn Daten auch nach einem Neustart des Systems erhalten bleiben müssen.

Hazelcast stellt eine weitere interessante Option dar, da es nicht nur eine In-Memory-Datenbank, sondern ein vollständiges In-Memory-Datenverarbeitungssystem bietet. Hazelcast unterstützt verteilte Datenstrukturen und ermöglicht die Speicherung von Daten in einem Cluster, wobei es sich automatisch um die Verteilung der Daten auf verschiedene Knoten kümmert. Neben dem Caching bietet Hazelcast Unterstützung für fortgeschrittene Funktionen wie verteilte Sperrmechanismen, MapReduce-Prozesse und Event-Streaming. Dies macht es zu einer leistungsstarken Lösung für verteilte Systeme, die mehr als nur einen schnellen Datenspeicher benötigen, sondern auch integrierte Mechanismen zur Datenverarbeitung und Synchronisation. Allerdings bringt diese Komplexität auch einen höheren Konfigurationsaufwand und eine größere Systemlast mit sich, was Hazelcast in einfachen Caching-Szenarien möglicherweise überdimensioniert macht.

In meiner Analyse hat sich gezeigt, dass die Wahl der richtigen In-Memory-Datenbank stark von den Anforderungen des jeweiligen Systems abhängt. Redis bietet eine hervorragende Mischung aus Geschwindigkeit, Flexibilität und erweiterten Funktionen, was es zur optimalen Lösung für Anwendungen mit anspruchsvollen Datenverarbeitungsanforderungen macht. Wenn jedoch lediglich ein schnelles und leichtgewichtiges Caching-System benötigt wird, ohne die Notwendigkeit für fortgeschrittene Datentypen oder Persistenz, ist Memcached eine hervorragende Alternative. Hazelcast schließlich überzeugt insbesondere in hochgradig verteilten Systemen, in denen neben Caching auch weitere verteilte Datenoperationen benötigt werden, ist jedoch komplexer in der Implementierung und Wartung.

Insgesamt bietet jede dieser Technologien spezifische Vorteile, die in unterschiedlichen Szenarien zur Geltung kommen. Die Entscheidung für eine dieser Lösungen hängt daher maßgeblich davon ab, ob der Fokus auf einfacher Skalierbarkeit und Geschwindigkeit (Memcached), vielseitiger Datenspeicherung und -manipulation (Redis) oder einem verteilten, robusten Datenverarbeitungssystem (Hazelcast) liegt.

3.1.5 Messagebroker

Für die Implementierung eines zuverlässigen und skalierbaren Messaging-Systems in unserem Projekt bietet sich RabbitMQ als bevorzugter Messagebroker an. RabbitMQ ist ein etabliertes und weit verbreitetes System, das auf dem Advanced Message Queuing Protocol (AMQP) basiert. Seine Stärken liegen insbesondere in der Kombination aus Flexibilität, Zuverlässigkeit und einem umfangreichen Funktionsspektrum, das sich hervorragend für moderne verteilte Systeme eignet.

Ein wesentlicher Grund für die Wahl von RabbitMQ ist seine Fähigkeit, eine garantierte Nachrichtenzustellung sicherzustellen. In Szenarien, in denen Nachrichten zwischen verschiedenen Diensten hin und her gesendet werden, ist es entscheidend, dass keine Nachrichten verloren gehen. RabbitMQ bietet Mechanismen wie die Bestätigung des Empfangs (Acknowledgements) und das zuverlässige Speichern von Nachrichten in Warteschlangen. Diese Funktionen gewährleisten, dass Nachrichten auch bei temporären Dienstaussfällen oder Verbindungsproblemen nicht verloren gehen, sondern zu einem späteren Zeitpunkt erneut zugestellt werden können.

Darüber hinaus bietet RabbitMQ umfangreiche Möglichkeiten, Nachrichtenflüsse flexibel zu gestalten. Durch den Einsatz von Exchange-Typen und Routing-Mechanismen können Nach-

richten gezielt an eine oder mehrere Warteschlangen weitergeleitet werden, was besonders in komplexeren Architekturen von großem Nutzen ist. Diese Flexibilität erlaubt es uns, den Nachrichtenfluss auf die spezifischen Anforderungen unseres Systems zuzuschneiden, sei es durch direkte Punkt-zu-Punkt-Kommunikation oder durch das Broadcasten von Nachrichten an mehrere Konsumenten.

Ein weiterer Aspekt, der für RabbitMQ spricht, ist seine hohe Integrationsfähigkeit in verschiedene Umgebungen und Programmiersprachen. RabbitMQ ist nicht nur plattformunabhängig, sondern verfügt auch über eine breite Unterstützung für verschiedene Programmiersprachen und Frameworks, was es zu einer vielseitigen Wahl für Projekte macht, die auf unterschiedliche Technologien zurückgreifen.

Obwohl andere Messagebroker, wie Apache Kafka oder Redis, ebenfalls eine gewisse Popularität genießen, gibt es klare Gründe, warum RabbitMQ in unserem Szenario die bessere Wahl ist. Apache Kafka beispielsweise ist für Anwendungen mit extrem hohen Durchsatzanforderungen und für die Verarbeitung von Streaming-Daten ausgelegt. Diese Eigenschaften sind jedoch in unserem Projekt weniger relevant, da unser Fokus auf einer robusten, zuverlässigen und gut strukturierten Nachrichtenverarbeitung liegt, anstatt auf der Bewältigung von Millionen von Nachrichten pro Sekunde. Kafka wäre somit in Hinblick auf die Anforderungen unseres Systems überdimensioniert und komplexer in der Implementierung. Redis hingegen, obwohl sehr performant, ist primär als In-Memory-Datenbank und Cache konzipiert. Seine Funktionalität als Messagebroker ist weniger ausgereift als die von RabbitMQ, besonders wenn es um die komplexen Anforderungen an Nachrichtenzustellung, Bestätigungen und Routing geht.

Zusammenfassend lässt sich sagen, dass RabbitMQ aufgrund seiner Stabilität, Flexibilität und breiten Unterstützung als Messagebroker die ideale Wahl für unser Projekt darstellt. Es bietet eine ausgereifte Infrastruktur, um Nachrichten sicher und effizient zwischen unseren Diensten zu verteilen, und stellt sicher, dass keine Daten verloren gehen. Seine Fähigkeit, sich an unterschiedlichste Anwendungsfälle anzupassen, sowie die einfache Integration in unsere bestehende Systemlandschaft machen RabbitMQ zu einem unverzichtbaren Bestandteil unserer Architektur.

3.1.6 Authenticator

OAuth 2.0, SAML (Security Assertion Markup Language), und OpenID Connect sind drei bedeutende Protokolle im Bereich der Authentifizierung und Autorisierung, die jeweils ihre eigenen Stärken und Schwächen haben. Dennoch gibt es mehrere Gründe, warum OAuth 2.0 im modernen Kontext oft als überlegen betrachtet wird, insbesondere wenn man es mit SAML und OpenID Connect vergleicht.

Einfachheit und Flexibilität

OAuth 2.0 zeichnet sich durch seine Einfachheit und Flexibilität aus. Es ist als Framework konzipiert, das es Entwicklern ermöglicht, Autorisierungsmechanismen einfach in ihre Anwendungen zu integrieren, ohne sich mit komplexen XML-Strukturen auseinandersetzen zu müssen, wie es bei SAML der Fall ist. Stattdessen setzt OAuth 2.0 auf das schlankere und

leichter handhabbare JSON, was die Implementierung und Wartung von OAuth 2.0-basierten Lösungen deutlich vereinfacht. Darüber hinaus unterstützt OAuth 2.0 eine breite Palette von Anwendungsszenarien, von mobilen Apps über Webanwendungen bis hin zu serverseitigen Implementierungen, was es zu einem äußerst vielseitigen Werkzeug macht.

Fokus auf Autorisierung statt Authentifizierung

Ein weiterer Vorteil von OAuth 2.0 ist sein klarer Fokus auf die Autorisierung, im Gegensatz zu SAML, das in erster Linie ein Authentifizierungsprotokoll ist. Während SAML darauf abzielt, Identitäten zu verifizieren und Benutzern den Zugang zu Anwendungen zu gewähren, erlaubt OAuth 2.0 es Drittanbietern, im Namen des Benutzers auf Ressourcen zuzugreifen, ohne dass das Passwort des Benutzers weitergegeben werden muss. Dies bietet eine erhöhte Sicherheit und bessere Kontrolle über den Zugriff auf sensible Daten. Benutzer können spezifische Berechtigungen erteilen und diese jederzeit wieder zurückziehen, was eine granularere Kontrolle über den Datenzugriff ermöglicht.

Moderne Nutzungsszenarien und Unterstützung für mobile Geräte

In einer Welt, in der mobile Geräte und Cloud-Dienste dominieren, ist die Fähigkeit von OAuth 2.0, solche modernen Nutzungsszenarien zu unterstützen, ein wesentlicher Vorteil. OAuth 2.0 ist so konzipiert, dass es nahtlos mit mobilen Anwendungen funktioniert, indem es unter anderem Unterstützung für Zugriffstoken und Refresh-Tokens bietet, die speziell auf die Herausforderungen mobiler Anwendungen zugeschnitten sind. SAML hingegen wurde in einer Zeit entwickelt, als Webbrowser die primäre Plattform für den Zugriff auf Anwendungen waren, und ist daher weniger gut geeignet für die Anforderungen moderner, mobiler Umgebungen.

Breite Akzeptanz und Integration in moderne APIs

OAuth 2.0 hat sich als Standard für den Zugriff auf APIs etabliert. Viele der weltweit größten und bekanntesten Dienste, wie Google, Facebook und GitHub, setzen auf OAuth 2.0, um Drittanbieteranwendungen den Zugang zu ihren APIs zu ermöglichen. Die breite Akzeptanz von OAuth 2.0 bedeutet, dass Entwickler auf eine Fülle von Ressourcen, Bibliotheken und Tools zugreifen können, um die Implementierung zu erleichtern. SAML und OpenID Connect haben zwar auch ihre Einsatzbereiche, jedoch ist die Verbreitung von OAuth 2.0 gerade im Bereich der API-Interaktion unübertroffen.

Weiterentwicklung durch OpenID Connect

Während OpenID Connect oft als eigenständiges Protokoll betrachtet wird, ist es tatsächlich eine Erweiterung von OAuth 2.0, die Authentifizierungsfunktionen hinzufügt. OpenID Connect kombiniert die Stärken von OAuth 2.0 mit einem standardisierten Authentifizierungsmechanismus, der sicherstellt, dass Benutzer eindeutig identifiziert werden können. Dies macht OpenID Connect zu einer idealen Wahl für Anwendungen, die sowohl Authentifizierung als auch Autorisierung benötigen. Da OpenID Connect direkt auf OAuth 2.0 aufbaut,

profitiert es von der Flexibilität und den modernen Funktionen, die OAuth 2.0 bietet, während es gleichzeitig die Schwächen von reinem OAuth 2.0 in Bezug auf Authentifizierung adressiert.

Interoperabilität und Zukunftssicherheit

Da OAuth 2.0 als Framework konzipiert ist, ist es sehr anpassungsfähig und zukunftssicher. Es unterstützt verschiedene Flows (z.B. Authorization Code, Implicit, Client Credentials) und kann leicht an neue Anwendungsfälle und Technologien angepasst werden. SAML, das hauptsächlich in älteren, unternehmensinternen Umgebungen verwendet wird, kann schwerfällig sein und ist weniger flexibel in der Anpassung an neue Technologien. OpenID Connect erweitert die Möglichkeiten von OAuth 2.0 noch weiter und ermöglicht es, Identitätsinformationen sicher über verschiedene Domains hinweg auszutauschen, was eine hohe Interoperabilität gewährleistet.

Zusammenfassend lässt sich sagen, dass OAuth 2.0 gegenüber SAML und OpenID Connect aufgrund seiner Flexibilität, Einfachheit, modernen Nutzungsszenarien und breiten Akzeptanz klare Vorteile bietet. Während SAML in traditionellen Unternehmensumgebungen weiterhin relevant sein mag und OpenID Connect eine starke Option für Authentifizierungsanforderungen darstellt, ist OAuth 2.0 in der heutigen digitalen Landschaft oft die bevorzugte Wahl, insbesondere wenn es um die Integration von modernen Web- und mobilen Anwendungen geht.

3.1.7 Loadbalancer

Im Rahmen dieses Projekts wurde Nginx als Loadbalancer eingesetzt, um die eingehenden Anfragen auf zwei ASP.NET APIs gleichmäßig zu verteilen. Der Loadbalancer verwendet den Round-Robin-Algorithmus, um die Anfragen abwechselnd zwischen den beiden APIs hin und her zu leiten. Dadurch wird sichergestellt, dass keine der beiden APIs überlastet wird, sondern beide möglichst gleichmäßig ausgelastet sind.

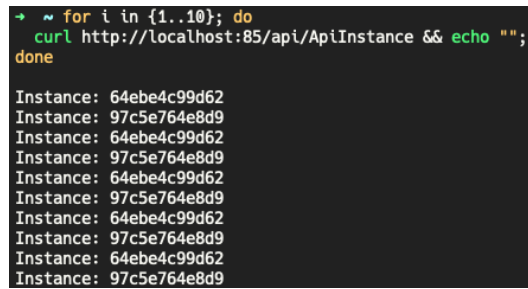
Die Wahl von Nginx als Loadbalancer fiel aufgrund seiner weiten Verbreitung und der einfachen Implementierung. Nginx bietet zudem eine zuverlässige und effiziente Möglichkeit, Lastverteilung zu realisieren.

Nginx wechselt zwischen den beiden API-Containern hin und her ohne einen Container und seine Lasten zu berücksichtigen. Daher gibt es Neben dem Round-Robin-Algorithmus auch noch weitere Algorithmen die aber nicht weiter betrachtet wurden da der Round-Robin-Algorithmus für unseren Zweck der Veranschaulichung ausreichend ist.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
336e6d8e9c9	nginx:latest	"/docker-entrypoint..."	8 minutes ago	Up 8 minutes	80/tcp, 0.0.0.0:85-85/tcp	loadbalancer
16816e4c12	postgres:10.6	"/docker-entrypoint.sh"	8 minutes ago	Up 8 minutes	5432/tcp, 0.0.0.0:5432-5432/tcp	postgres
91c5c764d8d9	api:latest	"/usr/bin/docker-ent..."	8 minutes ago	Up 8 minutes	8080/tcp, 0.0.0.0:8080-8080/tcp	api_container-2
6e0e4c9e0c2	api:latest	"/usr/bin/docker-ent..."	8 minutes ago	Up 8 minutes	8080/tcp, 0.0.0.0:8080-8080/tcp	api_container-1
63321fca705	postgres:10.6	"/docker-entrypoint.sh"	8 minutes ago	Up 8 minutes	5432/tcp, 0.0.0.0:5432-5432/tcp	postgres
16816e4c12	postgres:10.6	"/docker-entrypoint.sh"	8 minutes ago	Up 8 minutes	5432/tcp, 0.0.0.0:5432-5432/tcp	postgres
63321fca705	nginx:latest	"/usr/bin/docker-ent..."	8 minutes ago	Up 8 minutes	80/tcp, 0.0.0.0:80-80/tcp	nginx

Abbildung 3.3: Docker Container Nummern ³

³Quelle: Selbsterstellte Abbildung



```
→ ~ for i in {1..10}; do
  curl http://localhost:85/api/ApiInstance && echo "";
done
Instance: 64ebe4c99d62
Instance: 97c5e764e8d9
Instance: 64ebe4c99d62
Instance: 97c5e764e8d9
Instance: 64ebe4c99d62
Instance: 97c5e764e8d9
Instance: 64ebe4c99d62
Instance: 97c5e764e8d9
Instance: 64ebe4c99d62
Instance: 97c5e764e8d9
```

Abbildung 3.4: Ngnx Round-Robin Konfiguration⁴

3.2 Konzept - Beschreibung

In der Abbildung 3.1 ist das Layout der Systemarchitektur dargestellt, das in drei Hauptgruppen unterteilt ist: die Nutzergruppe, das Frontend und das Backend. Diese Struktur stellt sicher, dass die unterschiedlichen Anforderungen und Funktionalitäten der verschiedenen Benutzergruppen sowie die technischen Anforderungen des Systems effizient und übersichtlich umgesetzt werden können.

3.2.1 Nutzergruppen und deren Konzepte

Die Nutzergruppe besteht aus drei spezifischen Benutzerrollen: dem normalen Benutzer, dem Künstler und dem Label. Jede dieser Rollen hat unterschiedliche Rechte und Funktionalitäten innerhalb des Systems, die auf ihre spezifischen Bedürfnisse und Aufgaben abgestimmt sind.

Der normale Benutzer besitzt die geringsten Rechte innerhalb des Systems. Er kann Musik suchen, anhören und eigene Playlists erstellen, was ihm eine personalisierte und individuelle Musikerfahrung ermöglicht. Der Fokus liegt hier auf dem Konsum von Inhalten.

Der Künstler hingegen hat erweiterte Rechte. Neben den Funktionen eines normalen Benutzers kann er Musik hochladen, eigene Lieder verwalten und zusätzliche Informationen wie Beschreibungen oder Hintergründe zu seinen Songs veröffentlichen. Diese Informationen werden direkt mit den entsprechenden Liedern verknüpft und bieten den Hörern tiefere Einblicke in die Werke des Künstlers. Diese Rolle zielt darauf ab, Künstlern eine Plattform zu bieten, auf der sie ihre Musik nicht nur präsentieren, sondern auch aktiv mit ihrem Publikum interagieren können.

Das Label repräsentiert eine größere Musikfirma oder ein Produktionsunternehmen, das eine Vielzahl von Künstlern unter Vertrag hat und große Mengen an Musikdateien verwaltet. Labels haben eine besondere Stellung im System, da sie in der Lage sind, umfangreiche Musikbibliotheken hochzuladen. Um den Aufwand sowohl für das Label als auch für unser System zu minimieren, wird ihnen ein spezieller Upload-Mechanismus zur Verfügung gestellt, der den Prozess optimiert und eine Überlastung des Systems verhindert.

⁴Quelle: Selbsterstellte Abbildung

3.2.2 Frontend - Benutzeroberflächen und Plattformen

Das Frontend umfasst alle Zielsysteme, auf denen unsere Anwendung verfügbar sein soll. Dazu gehören eine Website, eine mobile App und eine Desktop-Anwendung. Diese sollen auf folgenden Plattformen bereitgestellt werden:

- **Website:** Die Hauptanlaufstelle für Nutzer, die über den Browser auf unsere Plattform zugreifen.
- **Mobile App:** Sowohl für Android- als auch für iOS-Geräte. Diese App ermöglicht es Nutzern, auch unterwegs auf die Musikbibliothek zuzugreifen.
- **Desktop-Anwendung:** Diese wird für Windows, Linux und macOS verfügbar sein, um eine breite Nutzerschaft abzudecken.

Alle drei Plattformen bieten sowohl dem normalen Benutzer als auch dem Künstler umfassenden Zugriff auf die jeweiligen Funktionen. Die Anfragen, die von diesen Frontends ausgehen, werden zunächst an einen Load Balancer weitergeleitet (siehe Kapitel 3.1.7), um eine gleichmäßige Verteilung der Last sicherzustellen und die Systemstabilität zu gewährleisten.

3.2.3 Backend - Systemkomponenten und Infrastruktur

Das Backend bildet das Herzstück der Systemarchitektur. Der erste Ansprechpartner für eingehende Anfragen ist der Load Balancer, der die Last auf verschiedene Instanzen verteilt, um die Systemressourcen effizient zu nutzen und eine Überlastung einzelner Komponenten zu vermeiden. In einem realen Anwendungsfall wird hier ein Nginx-Server eingesetzt, der den „Least Connected“-Algorithmus verwendet, um sicherzustellen, dass neue Anfragen an die am wenigsten belastete API-Instanz weitergeleitet werden.

Bevor eine Anfrage jedoch an die API weitergeleitet wird, muss sie durch einen Authentifizierungsdienst validiert werden, um die Sicherheit des Systems zu gewährleisten. Hierbei wird OAuth3 verwendet, um sicherzustellen, dass nur autorisierte Anfragen weiterverarbeitet werden (siehe Kapitel 2.3.3).

Nach erfolgreicher Authentifizierung wird die Anfrage an die API weitergeleitet, die auf dem ASP.NET-Framework basiert. Dieses Framework bietet von Haus aus zahlreiche Vorteile, die in späteren Kapiteln ausführlicher beschrieben werden. Die API interagiert anschließend mit dem Datenspeicher, der in drei Hauptkategorien unterteilt ist:

1. **Datenbank:** Wir haben uns für eine PostgreSQL-Datenbank entschieden, die als Cluster eingerichtet ist, um hohe Zuverlässigkeit zu gewährleisten. Jede Instanz der Datenbank hat einen Zwilling, eine exakte Kopie, die sicherstellt, dass im Falle eines Ausfalls einer Instanz die Daten weiterhin verfügbar sind. Dadurch wird das Risiko eines Datenverlusts minimiert und die Systemzuverlässigkeit erheblich gesteigert.

perl

2. **Fileserver:** Der Fileserver basiert auf MinIO und ist ebenfalls als Cluster mit einer redundanten Instanz eingerichtet. Dies garantiert, dass auch bei einem Ausfall des primären Fileservers weiterhin auf die gespeicherten Dateien zugegriffen werden kann.
3. **In-Memory Cache:** Hier kommt Redis zum Einsatz, allerdings nur in einer einzelnen Instanz. Da der In-Memory-Cache hauptsächlich dazu dient, die zuletzt geladenen Lieder zu speichern, um die Ladezeiten bei wiederholten Anfragen zu verkürzen, stellt ein Ausfall keine signifikante Beeinträchtigung des Gesamtsystems dar. Ein Ausfall führt lediglich zu einer etwas längeren Ladezeit für den Endbenutzer.

Die letzte Komponente, an die die API Daten sendet, ist ein Message Broker, der sich im abgekapselten Bereich des sogenannten „Top-K Problems“ befindet.

3.2.4 Top-K Problem

Die Verarbeitung des Top-K Problems stellt einen schwierigen Punkt der Systemarchitektur dar, da hier die eigentliche Berechnung der am häufigsten gehörten Lieder erfolgt, und diese Berechnungen sowohl in Echtzeit als auch präzise nach einer längeren Zeit zur Verfügung stehen müssen. Diese Phase muss effizient, skalierbar und in der Lage sein, eine große Menge an Echtzeitdaten zu verarbeiten. Um dies zu erreichen, kommen mehrere spezialisierte Komponenten und Techniken zum Einsatz.

Datenkonsum und Vorverarbeitung

Nachdem der Message Broker (wie Apache Kafka) die Daten über gehörte Lieder entgegengenommen hat, werden diese von einem dedizierten Datenkonsument abgerufen. Dieser Datenkonsument ist in der Regel ein Teil eines Stream-Processing-Frameworks wie Apache Flink oder Apache Spark Streaming. Der Konsument liest die Datenströme kontinuierlich und in Echtzeit aus den Kafka-Topics, in denen die Liedwiedergaben gespeichert sind.

Bevor die eigentliche Top-K Berechnung beginnt, erfolgt eine Vorverarbeitung der Daten. Dies kann das Filtern von Duplikaten, das Bereinigen der Daten oder das Anreichern der Daten mit zusätzlichen Informationen (z. B. Metadaten zu den Liedern) umfassen. Die Vorverarbeitungsschicht stellt sicher, dass nur relevante und qualitativ hochwertige Daten in den folgenden Verarbeitungsstufen verwendet werden.

Aggregation und Zwischenberechnungen

Nachdem die Daten vorverarbeitet wurden, beginnt der Aggregationsprozess. Hierbei wird die Anzahl der Wiedergaben für jedes Lied gezählt. Diese Aggregation kann in mehreren Stufen erfolgen:

Lokale Aggregation: Zunächst werden die Wiedergabezahlen auf lokaler Ebene (z. B. pro Partition oder pro Knoten im Stream-Processing-Cluster) zusammengefasst. Diese lokale Aggregation reduziert die Menge der zu verarbeitenden Daten erheblich, bevor sie auf globaler Ebene zusammengeführt werden.

Globale Aggregation: Nach der lokalen Aggregation werden die Teilergebnisse zusammengeführt, um eine globale Sicht auf die Wiedergabezahlen zu erhalten. Diese globale Aggregation kann in regelmäßigen Intervallen oder nach bestimmten Ereignissen erfolgen, um eine konsistente und aktuelle Liste der meistgehörten Lieder zu gewährleisten.

Berechnung der Top-K Liste

Sobald die Aggregation abgeschlossen ist, erfolgt die eigentliche Berechnung der Top-K Lieder. Dieser Schritt kann in zwei Hauptphasen unterteilt werden:

Sortierung: Die aggregierten Wiedergabezahlen werden nach ihrer Häufigkeit sortiert. Da das Ziel darin besteht, nur die K am häufigsten gehörten Lieder zu identifizieren, wird die Sortierung in absteigender Reihenfolge durchgeführt.

Selektion der Top-K: Aus der sortierten Liste werden dann die obersten K Einträge ausgewählt. Diese Auswahl repräsentiert die Lieder, die aktuell am häufigsten gehört werden. Je nach Implementierung kann dieser Prozess sehr effizient gestaltet werden, indem Heaps oder spezielle Datenstrukturen wie B-Bäume verwendet werden, die schnell auf die Top-K Elemente zugreifen können, ohne die gesamte Liste sortieren zu müssen.

Caching und Zwischenspeicherung

Um die Effizienz weiter zu steigern und die Latenz bei Anfragen nach der Top-K Liste zu minimieren, wird häufig ein Caching-Mechanismus implementiert. Hierbei können die Ergebnisse der Top-K Berechnungen in einem In-Memory-Datenspeicher wie Redis oder Memcached zwischengespeichert werden. Dies ermöglicht es, häufige Anfragen direkt aus dem Cache zu beantworten, ohne jedes Mal eine neue Berechnung durchführen zu müssen.

Skalierung und Fehlerbehandlung

Die Verarbeitung des Top-K Problems muss skalierbar sein, um mit der steigenden Anzahl von Benutzern und Datenströmen Schritt halten zu können. Dies wird durch horizontale Skalierung des Stream-Processing-Clusters erreicht, bei der zusätzliche Knoten hinzugefügt werden, um die Last zu verteilen.

Darüber hinaus ist es wichtig, Mechanismen zur Fehlerbehandlung und Wiederherstellung zu implementieren. Da es sich um ein Echtzeitsystem handelt, müssen Ausfälle von Komponenten oder Knoten robust abgefangen und kompensiert werden können. Apache Flink und Spark Streaming bieten beispielsweise integrierte Checkpointing- und Wiederherstellungsmechanismen, die den Zustand der Verarbeitung regelmäßig sichern und im Fehlerfall wiederherstellen können. Speicherung der Ergebnisse

Nach der Berechnung und eventuellen Zwischenspeicherung der Top-K Liste werden die Ergebnisse in eine PostgreSQL-Datenbank übertragen. Diese Datenbank speichert nicht nur die aktuellen Top-K Lieder, sondern kann auch historische Daten speichern, um Trends über Zeiträume hinweg zu analysieren. Ein Index auf der Lied-ID oder den Wiedergabezahlen ermöglicht schnelle Abfragen und gewährleistet, dass die Datenbank selbst bei hoher Last performant bleibt.

3.3 Konzept für labels

Ein Label ist typischerweise ein großes Musikunternehmen, das täglich eine erhebliche Menge an Musik produziert und verwaltet. Wenn ein Label eine große Anzahl von Musikdateien gleichzeitig in unser System hochladen möchte, könnte dies zu erheblichen Belastungsspitzen führen oder sogar Systemausfälle verursachen, insbesondere wenn gleichzeitig eine hohe Nutzerzahl auf unseren Dienst zugreift. Um diese Risiken zu minimieren und eine stabile Systemleistung zu gewährleisten, wurde im Backend ein spezieller Mechanismus implementiert.

Zentraler Bestandteil dieses Mechanismus ist ein dedizierter File Transfer Server. Auf diesen Server können die Labels ihre Musikdateien sowie alle zugehörigen Metadaten hochladen, die für die Verwaltung und Darstellung der Musikstücke im System notwendig sind. Dieser Prozess entlastet das Hauptsystem und ermöglicht es den Labels, ihre Inhalte ohne Zeitdruck und unabhängig von der aktuellen Systemlast hochzuladen.

Auf der anderen Seite gibt es eine speziell entwickelte Transfer-API, die regelmäßig den File Transfer Server überprüft, um festzustellen, ob neue Dateien zum Import bereitstehen. Sobald die Systemlast niedrig genug ist, beispielsweise in den späten Nachtstunden, wenn die Anzahl aktiver Nutzer gering ist, beginnt die Transfer-API damit, die hochgeladenen Musikdateien in unser zentrales Datenbank-Backend zu integrieren. Dieser zeitlich optimierte Importprozess stellt sicher, dass das System jederzeit leistungsfähig bleibt und gleichzeitig die Musik der Labels reibungslos in das System eingepflegt wird.

Durch diese Architektur wird nicht nur die Stabilität und Verfügbarkeit des Systems gewährleistet, sondern auch den Labels eine flexible und effiziente Möglichkeit geboten, ihre Musikproduktion in unser System zu integrieren. Diese Lösung stellt sicher, dass auch bei einem plötzlichen Anstieg der hochgeladenen Inhalte keine Überlastung oder Verzögerung auftritt, was insbesondere bei der Verwaltung von umfangreichen Musikbibliotheken von entscheidender Bedeutung ist.

4 Umsetzung

Um einen Proof of Concept unserer Systemarchitektur zu erreichen, haben wir ein Testprojekt umgesetzt. Um den Umfang hierbei nicht zu groß zu gestalten, haben wir nur die grün hinterlegten Elemente unseres Systems implementiert, wie in Abb. 3.1 zu sehen ist.

Um die nicht-funktionale Anforderung der Wartbarkeit umzusetzen, haben wir uns entschieden, die gesamte Docker-Umgebung mittels Terraform aufzusetzen. In den folgenden Kapiteln werden alle umgesetzten Elemente beschrieben, einschließlich ihrer Terraform-Komponenten, um sie als Docker-Container zur Verfügung zu haben.

4.1 Datenbackend

4.1.1 Postgree Cluster

Für das Projekt wurde eine einfache PostgreSQL-Datenbank eingerichtet. Da der Aufwand für den Aufbau eines Clusters zur Sicherstellung der Ausfallsicherheit zu groß war, haben wir darauf verzichtet. Abgesehen davon ist das Einrichten eines solchen Containers relativ einfach und kann mit folgendem Terraform-Skript durchgeführt werden:

```
1      # Primary PostgreSQL Server
2      resource "docker_image" "postgres" {
3          name = "postgres:latest"
4      }
5
6      resource "docker_container" "primary_postgres" {
7          image = docker_image.postgres.image_id
8          name  = "primary_postgres"
9          ports {
10             internal = 5432
11             external = 5432
12         }
13         env = [
14             "POSTGRES_DB=mydb",
15             "POSTGRES_USER=user",
16             "POSTGRES_PASSWORD=password",
17         ]
18
19         # Mount custom postgresql.conf and pg_hba.conf
20         volumes {
21             host_path      = "${abspath(path.module)}/postgresql.conf"
22             container_path = "/etc/postgresql/data/postgresql.conf"
23         }
24         volumes {
25             host_path      = "${abspath(path.module)}/pg_hba.conf"
26             container_path = "/etc/postgresql/data/pg_hba.conf"
27         }
```

```
28
29     networks_advanced {
30         name = docker_network.custom_network.name
31     }
32 }
33
```

Listing 4.1: NGINX Loadbalancer Konfiguration

4.1.2 Minlo Cluster

Für das Projekt wurde ein einfacher MinIO-Server eingerichtet. Da der Aufwand für die Einrichtung eines Clusters zur Sicherstellung der Ausfallsicherheit zu hoch war, haben wir darauf verzichtet. Ansonsten ist das Einrichten eines solchen Containers relativ unkompliziert und kann mit folgendem Terraform-Skript durchgeführt werden:

```
1  #Min.io
2  resource "docker_image" "minio_image" {
3      name = "minio/minio:latest"
4  }
5
6  resource "docker_container" "minio_container" {
7      image = docker_image.minio_image.image_id
8      name  = "minio"
9      env = [
10         "MINIO_ROOT_USER=minio",
11         "MINIO_ROOT_PASSWORD=minio123"
12     ]
13     ports {
14         internal = 9000
15         external = 9000
16     }
17     command = ["server", "/data"]
18
19     networks_advanced {
20         name = docker_network.custom_network.name
21     }
22 }
23
```

Listing 4.2: NGINX Loadbalancer Konfiguration

4.1.3 redis Db

Auch das Einrichten einer Redis-Datenbank ist relativ einfach und kann mit Terraform durchgeführt werden:

```
1  #Redis
2  resource "docker_image" "redis" {
3      name = "redis:latest"
4  }
5
6  resource "docker_container" "redis" {
7      name = "redis"
8  }
```

```
8         image = docker_image.redis.image_id
9         ports {
10             internal = 6379
11             external = 6379
12         }
13
14         networks_advanced {
15             name = docker_network.custom_network.name
16         }
17     }
18 }
```

Listing 4.3: NGINX Loadbalancer Konfiguration

4.2 Api

4.2.1 Komponenten der API

Login und Registrierung

Login: Der Login-Prozess ermöglicht es Benutzern, sich in die Anwendung einzuloggen und einen JWT-Token zu erhalten. Dieser Token dient zur Authentifizierung bei zukünftigen Anfragen. Der Login-Controller prüft die Anmeldedaten, verifiziert sie anhand der in der Datenbank gespeicherten Informationen und gibt im Erfolgsfall einen Token zurück.

Registrierung: Die Registrierung erlaubt es neuen Benutzern, sich zu registrieren, indem sie ihre Informationen an die API senden. Der Registrierungs-Controller speichert die neuen Benutzerdaten sicher in der PostgreSQL-Datenbank. Nach der erfolgreichen Registrierung wird ein Token zurückgegeben.

SongListeSearch

Der SongListeSearch-Controller bietet Endpunkte zur Suche nach Songs in der Datenbank. Benutzer können nach Songtiteln oder Künstlernamen suchen. Dieser Controller verarbeitet Suchanfragen, filtert die Datenbankergebnisse entsprechend der Benutzeranfrage und gibt die relevanten Songs zurück.

Song Controller

Der Song-Controller ermöglicht das Abrufen von einzelnen Liedern.

4.2.2 Kommunikation mit der Datenbank über EFCore

Entity Framework Core (EFCore):

EFCore ist ein ORM (Object-Relational Mapper), das es ermöglicht, auf einfache Weise mit relationalen Datenbanken zu interagieren. In unserem Fall verwenden wir EFCore, um mit einer PostgreSQL-Datenbank zu kommunizieren. EFCore verwaltet die Datenbankzugriffe und ermöglicht es uns, C#-Objekte mit den Datenbanktabellen zu verknüpfen.

Datenbank-Migrationen

Datenbank-Migrationen sind ein wesentliches Werkzeug, um die Datenbankstruktur an Änderungen im Code anzupassen. EFCore bietet ein Migrationstool, mit dem Änderungen an den Entitäten in SQL-Befehle umgewandelt werden können, die auf die Datenbank angewendet werden. Migrationen helfen bei der Versionierung der Datenbank. Bei einem Upgrade von einer Version auf die nächste minimieren sie auch das Risiko von Datenverlust.

4.2.3 JWT-Authentifizierung

JWT (JSON Web Token):

JWT ist ein Standard für die sichere Übertragung von Informationen zwischen Parteien als JSON-Objekt. In unserer API verwenden wir JWTs zur Authentifizierung und Autorisierung.

Generierung des Tokens:

Nach erfolgreicher Authentifizierung eines Benutzers wird ein JWT-Token erstellt. Dieses Token enthält verschlüsselte Informationen über den Benutzer, z.B. die Benutzer-ID und Rollen. Das Token wird an den Benutzer zurückgegeben und bei zukünftigen API-Anfragen als Header mitgeschickt.

Validierung des Tokens:

Bei jeder Anfrage an geschützte Endpunkte überprüft die API das JWT-Token, um sicherzustellen, dass der Benutzer authentifiziert ist. Wenn das Token gültig ist, wird die Anfrage bearbeitet. Andernfalls wird eine Fehlermeldung zurückgegeben.

Sicherheit:

Es ist wichtig, dass das JWT sicher erstellt und übertragen wird. Verwenden Sie starke Geheimnisse für die Token-Signierung und stellen Sie sicher, dass das Token nur über HTTPS übertragen wird, um Sicherheitsrisiken zu minimieren.

4.2.4 Aufsetzen des Dockers

Im Sinne der Ausfallsicherheit haben wir zwei API-Container instanziiert. Dies lässt sich ebenfalls sehr einfach mit Terraform umsetzen. Zunächst benötigt man ein Docker-Skript zum Erstellen des Docker-Images. Dieses Skript wird jedoch bereits bei der Erstellung der ASP.NET API mitgeliefert:

```
1      FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
2      # USER app
3      WORKDIR /app
4      EXPOSE 8080
5      EXPOSE 8081
6
7      FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
8      ARG BUILD_CONFIGURATION=Release
9      WORKDIR /src
10     COPY ["Spotify_Api.csproj", "."]
11     RUN dotnet restore "./Spotify_Api.csproj"
12     COPY . .
13     WORKDIR "/src/."
14     RUN dotnet build "./Spotify_Api.csproj" -c $BUILD_CONFIGURATION -o /
app/build
15
16     FROM build AS publish
17     ARG BUILD_CONFIGURATION=Release
18     RUN dotnet publish "./Spotify_Api.csproj" -c $BUILD_CONFIGURATION -o /
app/publish /p:UseAppHost=false
19
20     FROM base AS final
21     WORKDIR /app
22     COPY --from=publish /app/publish .
23     ENTRYPOINT ["dotnet", "Spotify_Api.dll"]
24
25
```

Listing 4.4: Dockerfile Asp.net

Anschließend muss man nur folgendes Terraform-Skript ausführen, um zwei Instanzen der API als Docker-Container bereitzustellen:

```
1      # Build Docker Image for API
2      resource "docker_image" "Api_Image" {
3          name          = "api_image:latest"
4          build {
5              context    = "./Spotify_Api"
6          }
7      }
8
9      # Deploy Two API Containers
10     resource "docker_container" "Api_container" {
11         count = 2
12         image = docker_image.Api_Image.image_id
13         name  = "Api_container_${count.index + 1}"
14         ports {
15             internal = 8080
16             external = "${8080 + count.index}"
17         }
18     }
```

```
19
20     networks_advanced {
21         name = docker_network.custom_network.name
22     }
23
24     depends_on = [
25         docker_container.primary_postgres
26     ]
27 }
28
```

Listing 4.5: Terraform Script für 2 Api's

4.3 Loadbalancer

4.4 Einführung

Loadbalancing ist eine Technik, die verwendet wird, um den Netzwerkverkehr gleichmäßig auf mehrere Server zu verteilen, um eine hohe Verfügbarkeit und Skalierbarkeit zu gewährleisten. In diesem Kapitel werden wir eine Loadbalancer-Lösung implementieren, die auf NGINX basiert und das Round-Robin-Verfahren verwendet, um Anfragen gleichmäßig auf mehrere Backend-Server zu verteilen.

4.5 Konfiguration des Loadbalancers

Die Konfiguration eines Loadbalancers in NGINX erfolgt durch die Bearbeitung der Konfigurationsdatei. Die Konfigurationsdatei von NGINX befindet sich in der Regel unter `/etc/nginx/nginx.conf`. Alternativ können Sie auch eine spezifische Konfigurationsdatei für Ihre Site im Verzeichnis `/etc/nginx/sites-available/` erstellen.

4.5.1 Grundlegende Konfiguration

Erstellen Sie eine neue Konfigurationsdatei oder bearbeiten Sie die bestehende Datei und fügen Sie die folgenden Konfigurationszeilen hinzu:

```
1     events {}
2
3     http {
4         upstream backend {
5             server Api_container_1:8080;
6             server Api_container_2:8080;
7         }
8
9         server {
10             listen 85;
11
12             location / {
13                 proxy_pass http://backend;
14                 # proxy_set_header Host $host;
```



```
15         # proxy_set_header X-Real-IP $remote_addr;
16         # proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
17         # proxy_set_header X-Forwarded-Proto $scheme;
18     }
19 }
20 }
21
```

Listing 4.6: NGINX Loadbalancer Konfiguration

In dieser Konfiguration:

- Der **upstream**-Block definiert eine Gruppe von Backend-Servern, auf die die Anfragen verteilt werden.
- Der **server**-Block konfiguriert den NGINX-Server, der Anfragen empfängt und sie an die Backend-Server weiterleitet.
- Die **proxy_pass**-Direktive leitet die Anfragen an die **upstream**-Gruppe weiter.
- Die **proxy_set_header**-Direktiven sorgen dafür, dass bestimmte Header-Informationen an die Backend-Server weitergegeben werden.

4.5.2 Round-Robin-Verfahren

Standardmäßig verwendet NGINX das Round-Robin-Verfahren, um Anfragen gleichmäßig auf die in der **upstream**-Gruppe definierten Server zu verteilen. Dies bedeutet, dass die Anfragen nacheinander an die Server in der Reihenfolge gesendet werden, in der sie definiert sind.

4.5.3 Aufsetzen Mit Terraform

Um den Loadbalancer als Dockercontainer zur Verfügung zu haben wurde folgendes terraform script verwendet:

```
1     # Load balancer / Reverse Proxy
2     resource "docker_image" "nginx" {
3         name = "nginx:latest"
4     }
5
6     resource "docker_container" "nginx" {
7         image = docker_image.nginx.name
8         name = "loadbalancer"
9         ports {
10             internal = 85
11             external = 85
12         }
13
14         volumes {
15             host_path      = "${abspath(path.module)}/nginx.conf"
16             container_path = "/etc/nginx/nginx.conf"
17         }
18
19         networks_advanced {
```

```
20         name = docker_network.custom_network.name
21     }
22
23     depends_on = [
24         docker_container.Api_container
25     ]
26 }
27
```

Listing 4.7: NGINX Loadbalancer Konfiguration

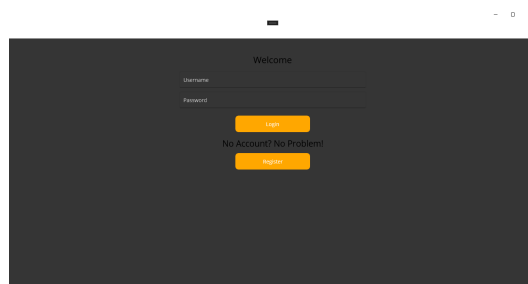
4.6 Frontend-UI

In diesem Kapitel erstellen wir eine MAUI-App, die eine plattformübergreifende Lösung für moderne mobile und Desktop-Anwendungen bietet. MAUI (Multi-platform App UI) von Microsoft ermöglicht es Entwicklern, mit einer einzigen Codebasis Apps für verschiedene Plattformen wie Android, iOS, Windows und macOS zu erstellen. Diese Flexibilität reduziert die Entwicklungszeit und den Wartungsaufwand erheblich, da wir keine separaten Codebasen für jede Plattform pflegen müssen.

Wir werden eine MAUI-App entwickeln, die eine Login-Seite, eine Registrierungsseite, eine Home-Seite, eine Header-Sektion mit einer Suchleiste und einem Such-Button, eine Seite für Suchergebnisse sowie einen Footer mit dem Titel des aktuell gespielten Liedes umfasst. MAUI bietet uns die Möglichkeit, eine konsistente Benutzererfahrung über verschiedene Geräte hinweg zu gewährleisten.

4.6.1 Login

Die Login-Seite ist der erste Interaktionspunkt für Benutzer. In `LoginPage.xaml` gestalten wir das Layout mit Eingabefeldern für den Benutzernamen und das Passwort sowie einem Login-Button und einem Weiterleitungs-Button zur Registrierung.

Abbildung 4.1: Login¹

¹Quelle: Selbsterstellte Abbildung

4.6.2 Registrierung

Ähnlich der Login-Seite enthält die Registrierungsseite (`RegistrationPage.xaml`) Eingabefelder für Benutzername, E-Mail-Adresse, Passwort und Passwortwiederholung sowie einen Registrierungs-Button.

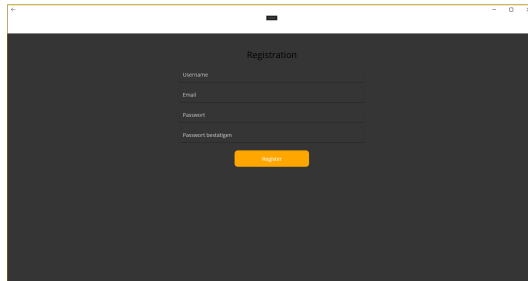


Abbildung 4.2: Registration²

4.6.3 Header

Der Header besteht aus einem Textfeld für die Suche sowie einem Such-Button. Angedeutet sind hier auch ein Profil- und ein Einstellungs-Button, die jedoch nur auf leere, nicht funktionierende Seiten verweisen, da diese für den Prototyp irrelevant sind.



Abbildung 4.3: Header³

4.6.4 Suchergebnisse

Wird der Suchbutton im Header bestätigt, gelangt man zu einer Seite mit einer Auflistung aller gefundenen Lieder. Dies ist eine Scroll-View, da je nach Eingabe der Bildschirm nicht ausreicht, um alle Lieder darzustellen.



Abbildung 4.4: Header⁴

wird hier auf den Play Button hinter einem lied gedrückt, startet ein Song.

²Quelle: Selbsterstellte Abbildung

³Quelle: Selbsterstellte Abbildung

⁴Quelle: Selbsterstellte Abbildung

4.6.5 Footer

Hier werden Informationen des aktuellen Songs dargestellt. Leider funktioniert dies nicht, da es bis zu letzter Probleme bei der Übertragung der Mp3 Datei von der Api zum Client gab.

4.7 Installationsanleitung zum testen des Systems

Im Folgenden wird beschrieben, wie die benötigten Komponenten installiert und das System gestartet werden können.

****Voraussetzungen****

Bevor mit der Installation begonnen wird, müssen folgende Softwarekomponenten auf dem System installiert sein:

- Terraform: Für die automatische Verwaltung und das Deployment der Docker-Container.
 - .NET SDK: Für die Kompilierung und den Betrieb der Anwendung.
 - .NET MAUI: Für die plattformübergreifende Entwicklung mobiler Anwendungen.
 - ASP.NET Web API: Für die Bereitstellung der Schnittstellen.
- **Installationsschritte****

Alle erforderlichen Docker-Container werden von Terraform aufgesetzt und automatisch gestartet. Es ist wichtig, dass Sie sicherstellen, dass Sie sich während der gesamten Installation immer im Verzeichnis „FinalProject“ befinden.

1. Navigieren Sie zunächst in den Ordner „FinalProject“:

```
1 cd FinalProject
2
```

2. Wechseln Sie anschließend in das Verzeichnis der API und erstellen Sie eine Release-Version:

```
1 cd Spotify_Api
2 dotnet publish -c Release
3
```

3. Führen Sie Terraform aus, um die Docker-Container zu starten:

```
1 cd ..
2 terraform init
3 terraform apply
4
```

****Ausführung auf einem Mac****

Um das Projekt auf einem Mac auszuführen, verwenden Sie den folgenden Befehl, nachdem Sie in das entsprechende Verzeichnis gewechselt haben:

```
1 cd Spotify2
2 dotnet build -t:Run -f net8.0-maccatalyst
3
```

****API-Testing****

Um die API separat zu testen, können Sie den integrierten Swagger-Dienst über die folgende URL aufrufen:

<http://localhost:85/swagger/index.html>

Dieser Endpunkt ermöglicht es Ihnen, die verfügbaren API-Routen einzusehen und direkt anzusprechen, um die Funktionalität zu überprüfen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine umfassende Analyse sowie die Umsetzung eines Spotify 2.0 Prototyps durchgeführt. Dabei standen sowohl technische als auch konzeptionelle Herausforderungen im Mittelpunkt, die es zu bewältigen galt. Der Schwerpunkt lag insbesondere auf der Entwicklung einer skalierbaren, wartbaren und flexiblen Architektur, die den spezifischen Anforderungen eines modernen Musik-Streaming-Dienstes gerecht wird. Die Integration verschiedener Technologien, wie Redis, Terraform, Docker, MinIO, PostgreSQL und Nginx, spielte eine zentrale Rolle in der Architektur des Systems.

Der Prototyp bietet bereits wesentliche Grundfunktionen, jedoch befindet er sich noch in einem frühen Stadium und ist keineswegs abgeschlossen. So fehlen derzeit noch wichtige Funktionen, die für die Nutzererfahrung entscheidend sind. Insbesondere der Musik-Player, der eine zentrale Komponente des Systems darstellt, muss noch vollständig implementiert werden. Auch essentielle Features wie eine Logout-Funktion, eine optimierte Listenansicht und die reibungslose Wiedergabe von Songs müssen weiterentwickelt werden.

Darüber hinaus wurde deutlich, dass die aktuelle Datenbankarchitektur, die auf einer einzigen PostgreSQL-Instanz basiert, langfristig nicht ausreichend ist. Um die Ausfallsicherheit und Skalierbarkeit zu gewährleisten, sollte ein Datenbank-Cluster implementiert werden, der gegenüber Störungen und Ausfällen besser abgesichert ist. Dies würde nicht nur die Verfügbarkeit des Systems erhöhen, sondern auch eine Grundlage für zukünftige Erweiterungen schaffen.

Der Prototyp hat somit das Potenzial, sich zu einem robusten und funktionsreichen System zu entwickeln, wenn diese noch offenen Punkte adressiert werden. Im weiteren Verlauf des Projekts gilt es, die bestehende Architektur zu verfeinern, die Implementierung der fehlenden Komponenten voranzutreiben und gleichzeitig sicherzustellen, dass das System auch in einem produktiven Umfeld stabil und skalierbar bleibt.