DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

TaskStorm: Skalierbare und flexible Datenbanklösung mit NoSQL

Umsetzungsdokumentation

an der Dualen Hochschule Baden-Württemberg Heidenheim

in der Fakultät Wirtschaft

im Studiengang Wirtschaftsinformatik

eingereicht von

Dorian Szenozicska

Yaroslava Weiss

Kevin Weigel

Semester:	WI2023A/4
Abgabedatum:	31.03.2025
Dozent:	Marvin Scharle, M.Sc.
GitHub-Repository:	https://github.com/cypher0003/Task-storm-RealTimeTodos.git

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Gender-Hinweis

Aus Gründen der besseren Lesbarkeit wird bei Personenbezeichnungen und personenbezogenen Hauptwörtern dieser Ausarbeitung teilweise die männliche Form verwendet. Entsprechende Begriffe gelten im Sinne der Gleichbehandlung grundsätzlich für alle Geschlechter. Die verkürzte Sprachform hat nur redaktionelle Gründe und beinhaltet keine Wertung.

Benutzer-Hinweis

Es wird explizit empfohlen, vor der Inbetriebnahme der Anwendung die README-Datei eingehend zu studieren. Diese Dokumentation liefert Informationen bezüglich Installation, Konfiguration, Bedienung des Systems und Hinweise und bildet somit die Grundlage für einen fundierten und fehlerfreien Einsatz. Eine sorgfältige Kontrolle ist unerlässlich, um mögliche Fehlbedienungen zu vermeiden und die Funktionalität der Anwendung optimal nutzen zu können.

Inhaltverzeichnis

I. Abkürzungsverzeichnis

Abkürzung	Bedeutung
ACID	Atomicity, Consistency, Isolation, Durability
AOF	Append-Only File
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DB	Database
EXES	*Expire Seconds* (in Redis)
HTTP	Hypertext Transfer Protocol
JWT	JSON Web Token
MULTI	Redis MULTI-Command
NoSQL	Not Only SQL
npm	*Node Package Manager*
Pub/Sub	Publish/Subscribe
RDB	Redis Database Backup
REST	Representational State Transfer
SQLite	Structured Query Language – Lite

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

1. Einleitung

In modernen Arbeitsumgebungen kommt einer effizienten und jederzeit verfügbaren Aufgabenverwaltung eine hohe Bedeutung zu, da sie sowohl die individuelle als auch die kollaborative Produktivität steigert. Die vorliegende Arbeit befasst sich mit der Entwicklung einer Echtzeit-To-do-Anwendung, die auf WebSockets, Redis, SQLite und Fastify basiert und die Anforderungen einer flexiblen, sicheren und skalierbaren Aufgabenverwaltung erfüllt.

1.1. Ziel des Projekts

Ziel dieses Projekts ist die Entwicklung einer modernen Echtzeit-To-do-Anwendung, die auf WebSockets, Redis, SQLite und Fastify basiert. Die Applikation ermöglicht es Nutzern, Aufgaben zu erstellen, zu verwalten und in Echtzeit mit anderen Nutzern zu teilen. Durch den Einsatz von Redis wird eine schnelle Synchronisation der Daten gewährleistet, während SQLite für die langfristige Speicherung der Aufgaben genutzt wird. Die Kombination dieser Technologien stellt sicher, dass Änderungen an Aufgaben unmittelbar übertragen werden und die Zusammenarbeit zwischen Nutzern effizient gestaltet werden kann.

1.2. Überblick über den Anwendungsfall

Die entwickelte Anwendung adressiert die steigende Nachfrage nach einer flexiblen und effizienten To-do-Listen-Verwaltung mit Echtzeit-Funktionalitäten, die eine nahtlose Zusammenarbeit zwischen den Nutzern ermöglicht. In modernen Arbeits- und Organisationsumgebungen ist es essenziell, Aufgaben synchron und ortsunabhängig zu verwalten, insbesondere im Kontext kollaborativer Teams. Die Anwendung ermöglicht es den Nutzern, Aufgaben zu erstellen, zu

bearbeiten und zu löschen, während ein integriertes Freundschaftssystem die Verwaltung gemeinsamer To-do-Listen unterstützt.

Durch den Einsatz von WebSockets werden To-dos in Echtzeit angelegt, verändert und gelöscht und über die Systemarchitektur an alle verbundenen Nutzer übertragen. Im Vergleich zu klassischen polling-basierten Systemen, die regelmäßig Anfragen an den Server senden müssen und dadurch sowohl die Serverlast als auch die Latenzzeit erhöhen, stellt dies eine wesentliche Verbesserung dar. [1]

Eine weitere essenzielle Funktionalität ist die Offline-Unterstützung durch Redis. Im Falle einer Unterbrechung der Internetverbindung werden die Änderungen lokal zwischengespeichert und bei Wiederherstellung der Verbindung synchronisiert. Die Architektur nutzt die Redis-Persistenzmechanismen wie Append-Only File (AOF) Logging oder Snapshotting, um eine zuverlässige Wiederherstellung und Datenkonsistenz zu gewährleistet. [2] Diese Vorgehensweise gewährleistet eine kontinuierliche Nutzbarkeit der Anwendung, ohne dass Datenverlust oder Inkonsistenzen auftreten.

Ein weiterer zentraler Aspekt der Architektur ist die Verwendung der JWT-Authentifizierung (Bearer Token), die eine sichere Anmeldung und den Schutz der Nutzerdaten gewährleistet. Im Vergleich zu Basic Authentication, bei der Anmeldeinformationen bei jeder Anfrage übertragen werden, bietet JWT den Vorteil, dass nach erfolgreicher Authentifizierung ein verschlüsseltes Token ausgestellt wird, das vom Client gespeichert und bei jeder weiteren Anfrage verwendet werden kann, ohne dass erneute Anmeldedaten übertragen werden müssen. Dadurch wird die Angriffsfläche für Man-in-the-Middle-Angriffe reduziert. [3]

Die Kombination aus Echtzeit-Datenübertragung, Offline-Funktionalität und sicherer Authentifizierung führt zur Entstehung eines leistungsfähigen Systems,

[1] Vgl. (Fette, 2011, S. 14 ff.)
[2] Vgl. (Redis, 2025)
[3] Vgl. (Jones, 2015, S. 13-16)

das eine effiziente und sichere Unterstützung kollaborativer To-do-Verwaltung gewährleistet.

1.3. Bedeutung von NoSQL für das Projekt

NoSQL-Datenbanken bieten eine flexible und skalierbare Alternative zu relationalen Datenbanksystemen, insbesondere für Anwendungen mit hohen Anforderungen an Echtzeitkommunikation und schnelle Datenverarbeitung. Sie zeichnen sich durch ihre horizontale Skalierbarkeit, fehlende Schemarestriktionen und einfache Replikation aus, wodurch sie sich besonders für große verteilte Systeme eignen. [4] Ein von entscheidenden Vorteilen von NoSQL-Datenbanken besteht in der Abwesenheit starrer Tabellenstrukturen. Relationale Datenbanken erzwingen durch ihr Schema eine feste Datenstruktur, während NoSQL-Datenbanken flexible Modelle bieten, die sich an sich ändernde Anforderungen anpassen können.[5]

In der vorliegenden To-do-Anwendung übernimmt Redis die Speicherung und Verteilung von Echtzeit-Updates, SQLite als relationale Datenbank für die langfristige Speicherung sowohl der Aufgaben und Nutzerbeziehungen als auch Nutzerverwaltung fungiert. In der vorliegenden Arbeit wird die hybride Architektur als ein wesentlicher Aspekt der IT-Infrastruktur erörtert, der eine performante Verarbeitung und Skalierbarkeit gewährleistet. Dies führt dazu, dass die Anwendung auch bei einer steigenden Anzahl von Nutzern effizient bleibt. [6]

Die Kombination von Fastify, WebSockets und Redis ermöglicht eine moderne, kollaborative und ressourcenschonende Lösung für das Aufgabenmanagement, die insbesondere für Teams oder Freundeskreise geeignet ist, die in Echtzeit zusammenarbeiten möchten.[7]

---

[4] Vgl. (Kaufmann, 2023, S. 252)
[5] Vgl. (Kaufmann, 2023, S. 252)
[6] Vgl. (Moniruzzaman, 2013)
[7] Vgl. (Redis, 2025); (Usunov & Chalkin, 2024); (Suranga, 2022)

2. Datenbanktechnologie & Systemarchitektur

Die Anwendung zeichnet sich durch eine hybride Datenbankarchitektur aus, welche relationale sowie NoSQL-Datenbanktechnologien integriert. Für die persistente Speicherung der To-do-Listen und Nutzerbeziehungen wird SQLite als relationale Datenbank verwendet, während Redis für die Echtzeit-Datenverarbeitung eingesetzt wird.

2.1.1. Auswahl der NoSQL-Datenbank

Für die Echtzeit-Synchronisation von To-do-Listen wurde Redis als NoSQL-Datenbank ausgewählt. Die Entscheidung für Redis als NoSQL-Datenbank basiert auf mehreren Faktoren. Redis ist eine In-Memory-Datenbank, die eine hohe Lese- und Schreibgeschwindigkeit bietet und damit für Echtzeitanwendungen optimiert ist. Eine ihrer wichtigsten Eigenschaften ist die Pub-/Sub-Architektur, die es ermöglicht, Datenänderungen unmittelbar an alle verbundenen Nutzer zu übertragen. Dadurch werden Verzögerungen minimiert und eine nahtlose Zusammenarbeit in Echtzeit gewährleistet. [8]

Ein weiterer Vorteil von Redis liegt in der Möglichkeit, temporäre Daten zu speichern und diese später mit einer persistenten Datenbank zu synchronisieren. Falls ein Nutzer offline ist, können Änderungen zwischengespeichert und nach Wiederherstellung der Verbindung synchronisiert werden. Diese Funktionalität trägt zur signifikanten Verbesserung der Benutzererfahrung bei und reduziert das Risiko von Datenverlusten erheblich. [9]

[8] Vgl. (Redis, 2025)
[9] Vgl. (Kim, 2024)

2.1.2. Datenmodellierung & Schema-Design

Das Datenmodell der Anwendung wurde so konzipiert, dass sowohl die Echtzeit-Synchronisation als auch die persistente Datenspeicherung optimal unterstützt werden. In SQLite sind die To-do-Listen und Nutzerbeziehungen in relationalen Tabellen organisiert. Jede To-do-Aufgabe wird durch eine eindeutige Identifikationsnummer repräsentiert und ist mit einem Nutzer oder einer Freundesgruppe verknüpft. Zudem werden Statusinformationen wie "pending" oder "done" gespeichert.

Redis hingegen speichert temporäre und Echtzeit-relevante Daten. Hierzu gehören aktive Sessions, ausstehende Änderungen und Echtzeit-Benachrichtigungen über Änderungen an To-do-Listen. Die Daten werden in Form von Schlüssel-Wert-Paaren[10] gespeichert, wobei jeder Nutzer über eine separate Redis-Instanz für seine aktiven To-Dos verfügt.

Die Kombination aus SQLite für persistente Daten und Redis für schnelle Zugriffe ermöglicht eine robuste Architektur, die sowohl Geschwindigkeit als auch Datenintegrität gewährleistet. Die Integration zusätzlicher NoSQL-Technologien zur Steigerung der Skalierbarkeit wird durch die modulare Struktur der Architektur problemlos ermöglicht.

2.1.3. Beschreibung der Funktionsweise der gewählten NoSQL-Datenbank

Im Rahmen der vorliegenden Arbeit wird Redis als NoSQL-Datenbank eingesetzt, um die Anforderungen an eine flexible, skalierbare und leistungsfähige Echtzeit-Datenverarbeitung zu erfüllen. Die schemalose Datenhaltung erlaubt die flexible und dynamische Speicherung von Daten als Schlüssel-Wert-Paare oder

[10] Vgl. (Kaufmann, 2023, S. 254)

Listen, wodurch eine unkomplizierte Anpassung der Datenstruktur an veränderte Anforderungen ermöglicht wird.[11][12]

Die Integration von Redis in die Server-Infrastruktur erfolgt mittels zweier dedizierter Verbindungen: Die Verbindung "redisPublisher" dient der Publikation von Nachrichten. Bei der Erstellung, Aktualisierung oder Löschung von To-dos werden die entsprechenden Datenänderungen zunächst in der relationalen Datenbank (SQLite) persistiert. Im Anschluss werden sie über definierte Redis-Channels, wie z.B. "workspace:<workspaceId>", publiziert. Die zweite Verbindung, "redisSubscriber", dient dem Abonnieren dieser Nachrichten, wodurch die empfangenen Daten unmittelbar an alle verbundenen WebSocket-Clients weitergeleitet werden. Die Integration erfolgt im Fastify-Server mittels der Dekoration des Server-Objekts über "fastify.decorate('redis', { redisPublisher, redisSubscriber })". Dadurch erhalten sämtliche Module und Services innerhalb der Anwendung Zugriff auf Redis-Funktionalitäten, ohne dass redundante Verbindungen aufgebaut werden müssen.

Das System der Echtzeitkommunikation beruht auf dem Publish-Subscribe-Verfahren (Pub/Sub), welches von Redis unterstützt wird. Bei Auftreten relevanter Ereignisse, wie etwa der Erstellung eines neuen To-dos, werden diese Ereignisse mittels der Funktion "sendToRedis" in den entsprechenden Redis-Channel publiziert. Die WebSocket-Clients, welche den betreffenden Channel abonniert haben, reagieren in Echtzeit auf diese Nachrichten und gewährleisten somit eine unmittelbare Synchronisation der Anwendungsdaten. Für den Fall, dass zum Zeitpunkt der Nachrichtenübermittlung keine aktiven Clients verbunden sind, erfolgt eine zusätzliche Zwischenspeicherung der Nachrichten im Redis-Cache. Dies dient dazu, verpasste Updates bei erneuter Verbindung nachträglich auszuliefern. Durch diesen Mechanismus werden ressourcenintensive Verfahren wie Polling vermieden und Latenzzeiten deutlich reduziert.

[11] Vgl. (Hecht & Jablonski, 2011, S. 336-341)
[12] Vgl. (Redis, 2025)

Neben der Übertragung im Echtzeitmodus fungiert Redis zudem als Cache, wobei beispielsweise Benutzerprofile über die Funktion "cacheUserProfile" temporär gespeichert werden. Das Ziel besteht darin, redundante Datenbankabfragen zu minimieren und die Performance der Anwendung zu optimieren. Funktionen wie "cacheWorkspaceTodos" und "cacheWorkspaceMembers" ermöglichen das Zwischenspeichern von weiteren relevanten Daten, wie etwa To-dos und Mitgliederinformationen eines Workspaces, und stellen so einen schnellen Zugriff auf häufig benötigte Informationen sicher. Zusätzlich erfolgt beim Systemstart eine initiale Synchronisation der in SQLite gespeicherten Daten mit dem Redis-Cache durch die Funktion "syncDatabaseToRedis", um die sofortige Verfügbarkeit relevanter Echtzeitdaten sicherzustellen.

Die Kombination der In-Memory-Speicherung, des effizienten Pub/Sub-Mechanismus und der gezielten Caching-Strategien resultiert in einer signifikanten Steigerung der Performance und Skalierbarkeit und gewährleistet eine robuste sowie flexible Datenverarbeitung. Die Trennung von Echtzeit- und Persistenzschicht ermöglicht zudem eine effiziente Echtzeitdatenverarbeitung bei gleichzeitiger langfristiger Datensicherheit und verdeutlicht, inwiefern moderne NoSQL-Datenbanken effektiv zur Realisierung leistungsfähiger und adaptiver Systeme beitragen können, insbesondere in kollaborativen Anwendungsszenarien.

2.1.4. Vergleich: NoSQL vs. relationale Datenbank

Die vorliegende Lösung zeichnet sich durch den Einsatz einer hybriden Methodik aus, die sowohl eine relationale Datenbank (diese wird als SQLite bezeichnet), als auch eine NoSQL-Datenbank (die Bezeichnung lautet hier Redis) einschließt. Dieser Ansatz ermöglicht es, die jeweiligen Stärken beider Technologien gezielt zu nutzen, um sowohl den Anforderungen an persistente Datenspeicherung (strukturierte Daten, ACID-Transaktionen) als auch an schnelle Echtzeitkommunikation (Pub/Sub, Caching) gerecht zu werden. Die Vorzüge von SQLite liegen insbesondere in der geringen Komplexität und dem niedrigen Ressourcenverbrauch. Bei einer Zunahme der Nutzerzahlen stößt die Software jedoch schnell

an ihre Skalierungsgrenzen. In solchen Fällen kann ein Wechsel zu relationalen Datenbanksystemen wie MySQL oder PostgreSQL in Erwägung gezogen werden, die für höhere Lasten konzipiert sind.[13][14] Als primär im Arbeitsspeicher agierendes System verarbeitet Redis Key-Value-Datenstrukturen äußerst effizient, unterstützt Pub/Sub-Funktionen und eignet sich damit hervorragend für Szenarien mit hohen Echtzeitanforderungen sowie intensiven Lese- und Schreibzugriffen..[15] Die Verbindung dieser beiden Ansätze führt zur Entstehung eines Systems, das sowohl die ACID-Eigenschaften relationaler Datenbanken als auch die Vorteile in Bezug auf Geschwindigkeit und Flexibilität einer NoSQL-Lösung vereint.

Die nachfolgende tabellarische Gegenüberstellung wichtiger Eigenschaften von SQLite (relationale Datenbank) und Redis (NoSQL-Datenbank) veranschaulicht die Unterschiede zwischen beiden Ansätzen.

Kriterium	SQLite (relational)	Redis (NoSQL)
Datenmodell	Relationales Modell (Tabellen, Spalten, Beziehungen).	Key-Value-Modell (Speicherung von Strings, Listen, Sets, Hashes etc.).
Transaktionssicherheit	ACID-Konformität (Atomicity, Consistency, Isolation, Durability).	Unterstützt Transaktionen über MULTI/EXEC, aber weniger umfangreiche ACID-Garantien, meist Eventual Consistency.
Skalierbarkeit	Leichtgewichtig, jedoch eingeschränkte Skalierbarkeit; für Anwendungen mit sehr vielen Nutzern potenziell	Hochskalierbar durch verteilte In-Memory-Knoten; horizontale Skalierung möglich, aber Datenbankgröße hängt stark vom

[13] Vgl. (Hipp, 2023)
[14] Vgl. (Redis, 2025)
[15] Vgl. (Hecht & Jablonski, 2011, S. 336-341)

	Wechsel auf MySQL/PostgreSQL notwendig.[16]	verfügbaren Arbeitsspeicher ab.
Performance	Gute Lese-/Schreibrate für kleinere und mittlere Datenmengen (Dateibasierte DB), bei großen Datenmengen und konkurrierenden Zugriffen jedoch Performanceeinbußen.	Sehr hohe Leistung bei Lese- und Schreiboperationen durch In-Memory-Ansatz; ideal für Caching und Echtzeit-Szenarien.[17]
Datenintegrität	Stark durch das relationale Schema und Foreign-Key-Beziehungen (kann so Datenanomalien vorbeugen).	Weniger restriktiv; Datenintegrität muss häufig in der Applikationslogik sichergestellt werden.
Einsatzszenarien	Geeignet für kleinere Projekte, eingebettete Anwendungen und Prototypen. Mit Upgrades auf MySQL/PostgreSQL für skalierbarere, komplexere Systeme.	Echtzeitanwendungen (Chat, Pub/Sub), hochfrequente Datenzugriffe (Analytics, Caching) und als unterstützende DB in Microservice-Architekturen.
Persistence	Daten werden persistent auf dem Dateisystem abgelegt; Backup-Strategien sind entsprechend einfach (Kopie der Datenbankdatei).	Standardmäßig In-Memory; Persistenzoptionen vorhanden (RDB, AOF), jedoch höherer Aufwand für Voll-Backups je nach Konfiguration.[18]
Komplexität der Integration	Einfache Integration, da viele Frameworks und	Ebenfalls in vielen Umgebungen verfügbar, aber

[16] Vgl. (Hipp, 2023)
[17] (Hecht & Jablonski, 2011, S. 336-341)
[18] Vgl. (Redis, 2025)

	Sprachen SQLite "out of the box" unterstützen.	Konfiguration teils komplexer als eine Single-File-DB wie SQLite.[19]

Tabelle 1. Vergleichstabelle NoSQL vs. relationale Datenbank

[19] Vgl. (Redis, 2025)

3. Systemarchitektur & Entwicklung

In diesem Kapitel ist eine Erläuterung der Software- und Systemarchitektur sowie der technischen Umsetzung dargestellt, gefolgt von einer Erläuterung der vorteilhaften Nutzung einer NoSQL-Datenbank. Abschließend erfolgt eine Bewertung, inwieweit die definierten Anforderungen aus dem Anforderungskatalog erfüllt sind.

3.1.1. Software- und Systemarchitektur

Die zugrunde liegende Architektur nutzt grundlegende Konzepte eines ereignisbasierten Kommunikationsmodells. In diesem Modell veröffentlichen spezifische Komponenten (wie Benutzeraktionen) Ereignisse, die über eine vermittelnde Instanz, den sogenannten Event Bus, an alle registrierten Abonnenten weitergeleitet werden. Die Vorgehensweise wird in Abbildung 1 veranschaulicht, wobei Komponenten Ereignisse in den Bus einspeisen und andere Komponenten diese über entsprechende Abonnements empfangen können. Durch diese lose Kopplung entfällt die Notwendigkeit, dass Sender und Empfänger sich direkt kennen oder miteinander kommunizieren, was insbesondere bei verteilter Software von Vorteil ist. [20]

[20] Vgl. (O. Haase, S. 15)

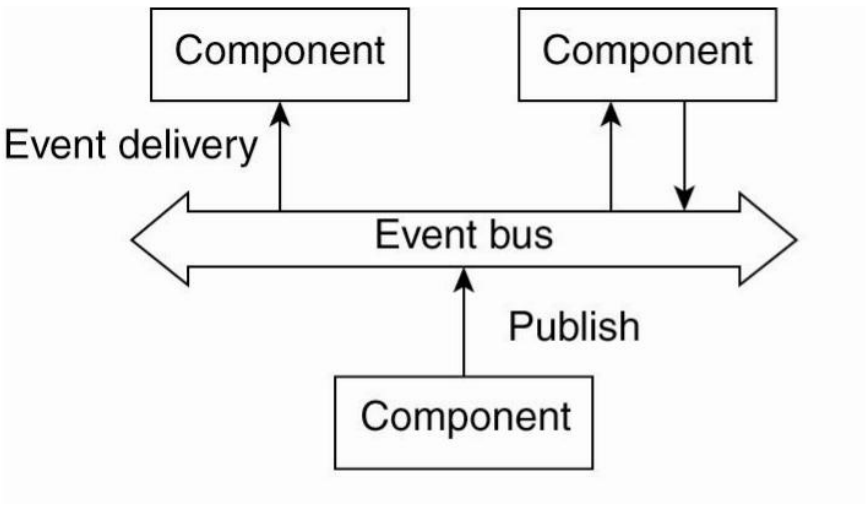

Abbildung 1. Ereignisbasierte Architektur[21]

In der vorliegenden Anwendung wird Redis als Event Bus genutzt. Das Anlegen oder Ändern eines neuen To-dos durch einen Nutzer löst eine "Publish-Operation" aus, die das Ereignis in einen spezifischen Channel (z. B. "workspace:<workspaceId>") einsteuert. In der Folge werden alle anderen Komponenten, die diesen Channel abonniert haben, in Echtzeit über die erfolgte Änderung benachrichtigt. Durch die Kombination mit WebSockets ist es möglich, diese Ereignisnachrichten unmittelbar an alle verbundenen Clients weiterzuleiten. In der Konsequenz sehen sämtliche Nutzer in einem Workspace dieselben, stets aktuellen Informationen.

Aus technischer Perspektive erfolgt die Publikation von Ereignissen durch den Aufruf eines Publishers, der die Daten der Änderung an den entsprechenden Redis-Channel übermittelt. Auf der Seite der Empfänger agiert ein Subscriber, der alle Nachrichten aus dem abonnierten Channel entgegennimmt und sie an die jeweilige Komponente im Backend oder direkt an die geöffneten WebSocket-Verbindungen im Frontend weiterleitet. Das Prinzip der Publish/Subscribe-Kommunikation bietet nicht nur eine performante Verteilung von Ereignissen, sondern auch eine flexible Erweiterbarkeit. Bei Bedarf können neue Komponenten sich

[21] Vgl. (Tanenbaum S., 2007)

ebenfalls auf einen Channel registrieren, ohne dass eine Anpassung der bestehenden Struktur erforderlich ist.

Im Kontext der To-do-Verwaltung bedeutet dies konkret, dass der Anwender, welcher ein neues To-do in einem Workspace anlegt, lediglich ein einzelnes Ereignis in Redis veröffentlicht. Alle anderen Nutzer, die im selben Workspace angemeldet sind und denselben Channel abonniert haben, erhalten daraufhin augenblicklich eine Benachrichtigung. Sie können ihre Benutzeroberfläche ohne Verzögerung aktualisieren. In diesem Prozess fungieren Redis (als Ereignisbussystem) und WebSockets (als Transportmechanismus zum Client) in Kombination als klassische Realisierung einer ereignisbasierten Architektur. Diese Architektur gewährleistet eine automatisierte Verteilung von Änderungen an alle relevanten Teilnehmer. [22]

Die Anwendung demonstriert den kombinierten Einsatz einer klassischen relationalen Datenbank (SQLite) und einer NoSQL-Datenbank (Redis) innerhalb eines mehrschichtigen Architekturmodells, das eine klare Trennung zwischen Frontend, Backend und Datenbankstrukturen ermöglicht. Während das Frontend die Benutzerschnittstelle bereitstellt, übernimmt das Backend die Aufgaben der Geschäftslogik und steuert sämtliche Datenbankzugriffe. Das Besondere an dieser Verbindung ist die Kombination relationaler und NoSQL-Konzepte: SQLite wird vorwiegend für dauerhafte Speicherung und transaktional sichere Operationen genutzt, während Redis eine schemalose, hochperformante Verwaltung von Echtzeit- und Caching-Daten gewährleistet.

Die NoSQL-Prinzipien manifestieren sich insbesondere in der Flexibilität und Geschwindigkeit der Redis-Integration. Im Gegensatz zum strengen Tabellenmodell relationaler Systeme, bei dem Daten in Form von Key-Value-Paaren, Listen oder Hashes gespeichert werden, erlaubt Redis eine flexible Strukturierung, die eine rasche Anpassung an neue Anforderungen ermöglicht. Die In-Memory-Bereitstellung und die Unterstützung eines Pub/Sub-Mechanismus ermöglichen eine effiziente Verteilung von Nachrichten an verbundene Clients. Nutzer können auf

[22] Vgl. (O. Haase, S. 15)

verpasste Aktualisierungen im Cache zurückgreifen, was beim Wiederverbinden zu einem lückenlosen Informationszugang führt.

Die Kommunikation zwischen den Softwarekomponenten erfolgt grundsätzlich über zwei Wege: Erstens über HTTP-Endpunkte (REST-API) für zentrale CRUD-Operationen und zweitens via WebSockets für den Echtzeitaustausch von Daten. Dadurch ist das Frontend in der Lage, bei Bedarf kontinuierlich aktuelle Informationen, etwa neue To-dos oder Änderungen in Workspaces, zu empfangen. Bei jeder Datenänderung (Erstellen, Aktualisieren, Löschen) bedient sich der Server sowohl der relationalen als auch der NoSQL-Datenbank, um das Zusammenspiel aus Persistenz und Echtzeit-Update abzubilden.

In der Gesamtschau ergibt sich demnach eine architektonische Diversifikation: Das Frontend initiiert Anfragen an das Backend, das die Logik für Datenbankzugriffe und Verarbeitungsprozesse beinhaltet, während die resultierenden Daten je nach Verwendungszweck entweder in SQLite (persistente Relationstabellen) oder in Redis (flexible Zwischenspeicherung und Publikation) verwaltet werden. Die klar definierten Schnittstellen zwischen diesen Schichten erleichtern die Wartung und Skalierung der Anwendung und demonstrieren die gezielte Nutzung von NoSQL-Konzepten, um reaktionsschnelle und vielseitige Systeme zu realisieren.

3.1.2. Technische Umsetzung

Hinsichtlich der technischen Realisierung wurde ein Ansatz verfolgt, bei dem eine Trennung von Frontend und Backend bei gleichzeitiger nahtloser Interaktion beider Segmente gewährleistet wird. Im Frontend kommen die gängigen Web-Technologien (HTML, CSS, TypeScript, NextJS als Framework) zum Einsatz, ergänzt durch eine reaktive Komponente für den Echtzeit-Datenaustausch via WebSockets. Dadurch wird sichergestellt, dass Statusänderungen (beispielsweise neue To-dos oder aktualisierte Benutzerinformationen) unmittelbar angezeigt werden, ohne dass ein erneutes Laden oder kontinuierliches Abfragen notwendig ist.

Das Backend nutzt ein modulares Konzept in Form von unterschiedlichen Services (wie Authentifizierung, Freundesverwaltung, To-do-Management), die im

Zusammenspiel mit Fastify und Node.js ausgeführt werden. Die Kreativität manifestiert sich insbesondere in der Zusammenarbeit einzelner Module, um einen möglichst flexiblen und zugleich robusten Betrieb zu gewährleisten. Durch die Kombination von Publish-Subscribe-Mechanismen (Redis), klassischen REST-Endpoints (HTTP) und Echtzeitkommunikation (WebSockets) werden sowohl Standard-CRUD-Operationen als auch latenzarme Datenaktualisierungen realisiert, ohne dabei die Codebasis unnötig zu verkomplizieren.

Obwohl der Fokus des Projekts auf einem NoSQL-Ansatz liegt und Redis insbesondere in der To-do-Verwaltung für Echtzeit-Updates zum Einsatz kommt, wird in ausgewählten Bereichen. wie Benutzer- oder Freundschaftsverwaltung, auf eine relationale Datenbank zurückgegriffen. Diese Vorgehensweise ist zum einen zeitlich motiviert. Relationale Datenbanken und ihre etablierten Ökosysteme lassen sich häufig schneller integrieren. Zum anderen weisen sie für bestimmte Anwendungsfälle mit klaren relationalen Abhängigkeiten eine geringere Umsetzungs- und Testkomplexität auf. relationale Strukturen erlauben zudem eine unkomplizierte Abbildung von Beziehungen zwischen Benutzern und Workspaces, was unter engen Zeitvorgaben eine robuste Alternative zu einer vollumfänglichen NoSQL-Lösung in allen Teilbereichen darstellt. Die Verwendung von Redis in der To-do-Verwaltung sowie bei der Echtzeitkommunikation per WebSocket erlaubt dennoch einen beschleunigten und bidirektionalen Austausch von Daten, was dem Fokus auf NoSQL und flexiblen Publish-Subscribe-Mechanismen entspricht. Eine vollständig auf NoSQL fokussierte Architektur wäre zwar grundsätzlich denkbar, jedoch hätte dies insbesondere im Umgang mit komplexeren relationalen Strukturen zu einem signifikant höheren Implementierungsaufwand geführt. Dieser wäre im Rahmen des aktuellen Projektumfangs nicht umsetzbar. Die gewählte hybride Lösung ermöglicht die zuverlässige Implementierung zentraler Funktionen und gewährleistet zugleich ein ausgewogenes Zusammenspiel verschiedener Technologien. Somit wurde eine solide Grundlage für Erweiterungen und zukünftige Anpassungen geschaffen.

Im Rahmen der Entwicklungsumgebung wurden Werkzeuge wie Visual Studio Code, Git und Node Package Manager (npm) eingesetzt. Visual Studio Code unterstützt die Qualität und Lesbarkeit des Codes, während Git die revisionssichere

Verwaltung sämtlicher Projektdateien gewährleistet. Durch den Einsatz von npm-Skripten können verschiedene Aufgaben (beispielsweise das Starten des Servers oder das Aktualisieren von Abhängigkeiten) automatisiert werden, wodurch kurze Feedbackzyklen ermöglicht werden. Dieses Setup erlaubt es, Funktionalitäten schnell zu erweitern und iterative Verbesserungen an Frontend sowie Backend ohne größere Umstellungen vorzunehmen.

3.1.3. Vorteilhafte Nutzung der NoSQL-Datenbank

Die Entscheidung, im Rahmen dieses Projekts auf die Verwendung einer NoSQL-Datenbank zu setzen, basiert maßgeblich auf den Anforderungen an die Echtzeitkommunikation sowie die hohe Zugriffsgeschwindigkeit. Da Redis sämtliche Daten im Arbeitsspeicher vorhält, weisen Lese- und Schreibzugriffe typischerweise sehr geringe Latenzzeiten auf.[23] Insbesondere beim Anlegen oder Aktualisieren von To-dos, für das eine unverzügliche Benachrichtigung aller verbundenen Clients erforderlich ist, ist dieser Performancegewinn von zentraler Bedeutung.

Darüber hinaus bietet die schemalose Datenhaltung von Redis signifikante Vorteile, da sie die flexible Ablage und Anpassung von Schlüssel-Wert-Paaren, Listen oder Hashes ermöglicht. Im Gegensatz zu strikt relationalen Systemen ist keine Vorabfestlegung von Tabellen- oder Spaltendefinitionen erforderlich, was eine dynamische Erweiterung oder Änderung der Datenstruktur ermöglicht. Dies beschleunigt den Entwicklungsprozess und erleichtert spätere Anpassungen bei sich ändernden Anforderungen.[24]

Gegen eine rein relationale Lösung bestehen signifikante Einwände. Zwar bieten relationale Datenbanken Replikationsfunktionen und Trigger, jedoch sind diese Funktionen häufig nicht auf den hohen Durchsatz ausgelegt, der in

[23] Vgl. (Hecht & Jablonski, 2011, S. 336-341)
[24] Vgl. (Equant)

hochfrequenten Kollaborationsszenarien erforderlich ist.[25] Redis hingegen verfügt über einen speziell zugeschnittenen Pub/Sub-Mechanismus, der die unmittelbare Verteilung von Nachrichten an alle verbundenen Clients gewährleistet (siehe Abbildung 2).

Ist kein Client aktiv verbunden, speichert Redis eingehende Aktualisierungen zwischen und liefert sie bei Wiederverbindung unmittelbar aus. In einer rein relationalen Umgebung wäre die Entwicklung eines eigenen Mechanismus erforderlich, der in Bezug auf den Implementierungsaufwand und die Wartbarkeit deutlich komplexer wäre (siehe Abbildung 3).

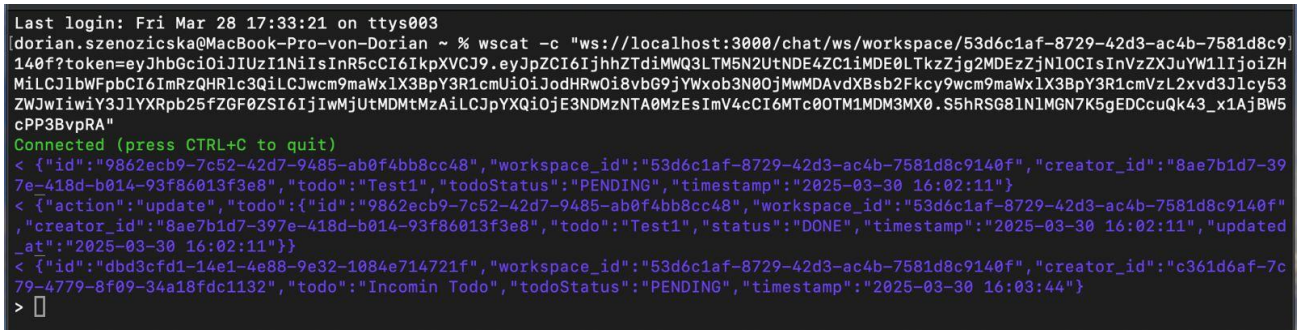

Abbildung 2. Nachrichtenempfang bei aktiver Websocketverbildung

Abbildung 3. Nachrichtenempfang bei verpassten Nachrichten über Websocket

[25] Vgl. (Hecht & Jablonski, 2011, S. 336-341)

Relationale Systeme verfügen zwar teilweise über integrierte Cache-Funktionen (z. B. Query-Caches), jedoch ist festzustellen, dass diese in Bezug auf die Konsistenz der Daten und die Sicherheit der Transaktionen deutlich weniger leistungsfähig sind. Relationale Systeme fokussieren sich jedoch stärker auf Datenkonsistenz und Transaktionssicherheit als auf hochfrequente In-Memory-Abfragen. Der In-Memory-Ansatz von Redis ermöglicht eine signifikante Reduktion der Bearbeitungszeiten für wiederkehrende Anfragen und bietet somit einen substanziellen Geschwindigkeitsvorteil für häufig genutzte Daten.[26]

Im Gegensatz dazu erfordern relationale Datenbanken bei Änderungen an der Datenstruktur (z. B. Hinzufügen neuer Spalten) oft einen zeit- und ressourcenintensiven Migrationsprozess. Im Gegensatz dazu ermöglicht Redis durch seine schemalose Natur eine unmittelbare Erweiterung oder Modifizierung, wodurch das System an neue Anforderungen angepasst werden kann, ohne dass aufwendige DB-Migrationen durchgeführt werden müssen.

Die Kombination der beiden Systeme trägt maßgeblich zur Robustheit und Leistungsfähigkeit der Anwendung bei und erlaubt es, spontane Skalierungsanforderungen sowie wechselnde Datenmodelle effektiv zu bewältigen.[27]

3.1.4. Erfüllung der Anforderungen aus dem Anforderungskatalog

Im Rahmen der vorliegenden Anwendung sind alle als "Must-have" (P0) klassifizierten Anforderungen aus dem bereitgestellten Anforderungskatalog erfolgreich umgesetzt, sodass der Kernnutzen für die im Dokument definierten Nutzerrollen (Members und Workspace-Admins) gewährleistet ist (siehe Abbildungen 4 und 5). Die implementierten CRUD-Funktionen für To-dos, das integrierte Freundschaftssystem inklusive der gemeinsam nutzbaren To-do-Listen sowie das Echtzeit-Update über WebSockets gewährleisten eine effektive, kollaborative

[26] Vgl. (Obe & Hsu, 2015)
[27] Vgl. (Equant)

Nutzung. Mittels JWT-Authentifizierung wird die Registrierung und Anmeldung der Nutzer ermöglicht. So kann sichergestellt werden, dass das Kernziel der sicheren Zugriffskontrolle auf persönliche und geteilte To-dos erreicht wird. Das Konzept der Workspaces, innerhalb derer mehrere Personen gleichzeitig Aufgaben anlegen und bearbeiten können, wurde ebenfalls umgesetzt. Die Performance- und Skalierbarkeitsoptimierung wird durch den Einsatz von Redis als Komponente im Arbeitsspeicher sowie eines Caching-Mechanismus erzielt, während die langfristige Datensicherheit in SQLite gewährleistet ist.

Darüber hinaus sind verschiedene weitere Anforderungen, die dem "Should-have"-Bereich (P1) zuzuordnen sind, teilweise umgesetzt worden. Bei den Benutzerfunktionen sind sämtliche P1-Features realisiert, mit Ausnahme der Suchfunktion in To-do-Listen. Zusätzlich ist von den "Could-have"-Funktionen (P2) bereits das Benutzerprofil mit Profilbildern verfügbar. Funktionen, die dem "Should-have"-Bereich (P1) zuzuordnen sind, wie "Kategorisierung & Filter", "Kommentare in To-dos", "Dark Mode", "Aufgabenzuweisung" oder die vorgesehene Kalender-Synchronisation, sind hingegen noch nicht realisiert und befinden sich im Backlog (siehe Abbildung 5).

Im administrativen Bereich sind beide als P0 klassifizierten Kernfunktionen, insbesondere die Rollen- und Rechteverwaltung sowie die Mitgliederverwaltung im Workspace, erfolgreich implementiert. Allerdings sind weder die als "Should-have" (P1) definierten noch die als "Could-have" (P2) klassifizierten Anforderungen realisiert. Dies betrifft unter anderem Features wie "Sicherheitsüberwachung & Audit-Logs", "Papierkorb & Wiederherstellung von Daten" oder einen möglichen "Gastzugang für To-dos".

Systemarchitektur & Entwicklung

Die Nichterfüllung dieser weiterführenden Anforderungen ist dabei vor allem auf zeitliche Restriktionen sowie die Priorisierung der Stabilisierung des Kernfunktionsumfangs zurückzuführen. Zwar erlaubt die gewählte modulare Software-Architektur prinzipiell eine nachträgliche Integration zusätzlicher Features, etwa durch die reaktive Auslegung des Frontends oder den Einsatz von Redis-Pub/Sub im Backend, jedoch wurde im aktuellen Entwicklungszyklus entschieden, sich auf P0 und zentrale P1-Funktionen zu konzentrieren.

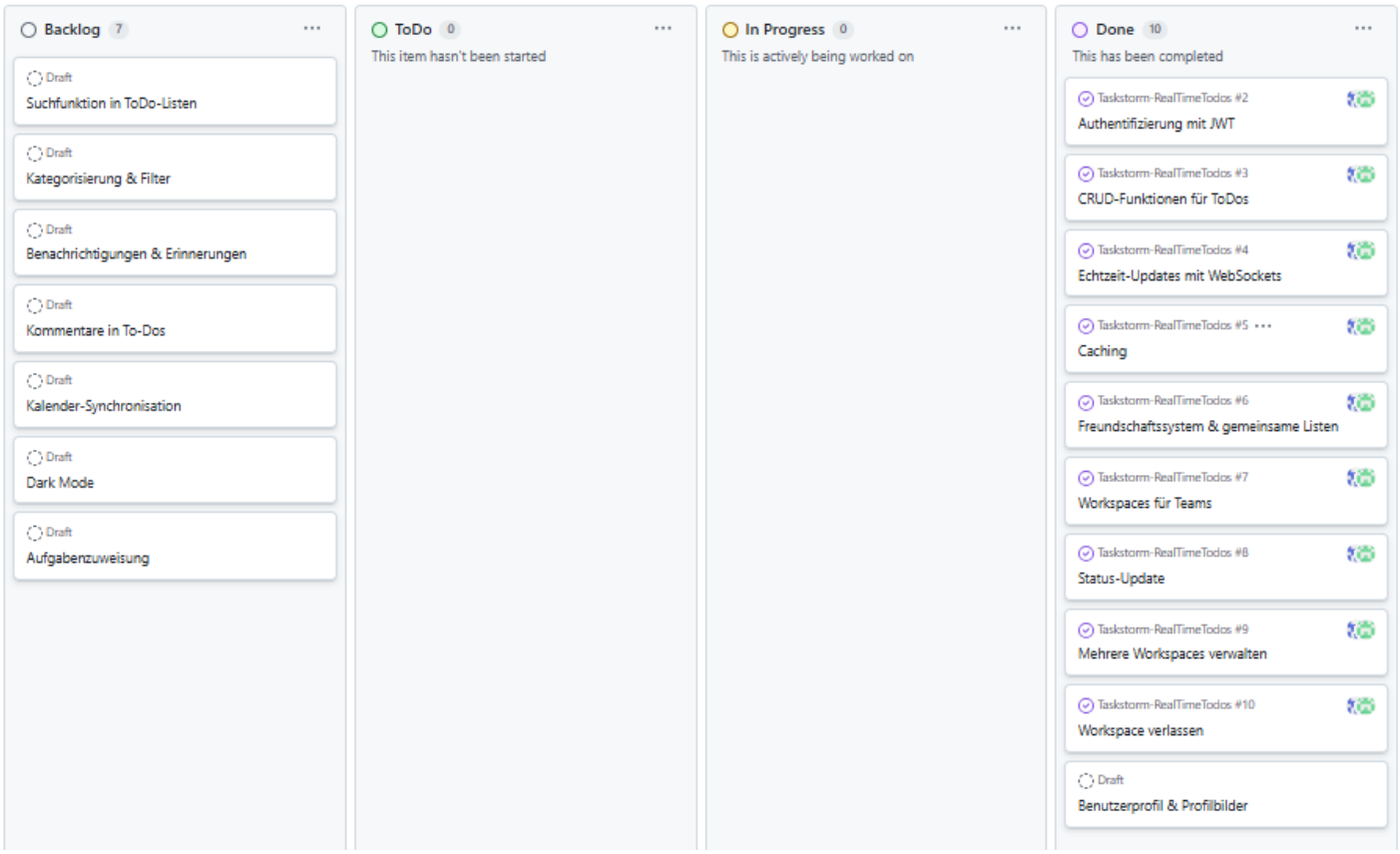

Abbildung 4. Überblick über geplante und umgesetzte Funktionen im GitHub-Projektboard - Member

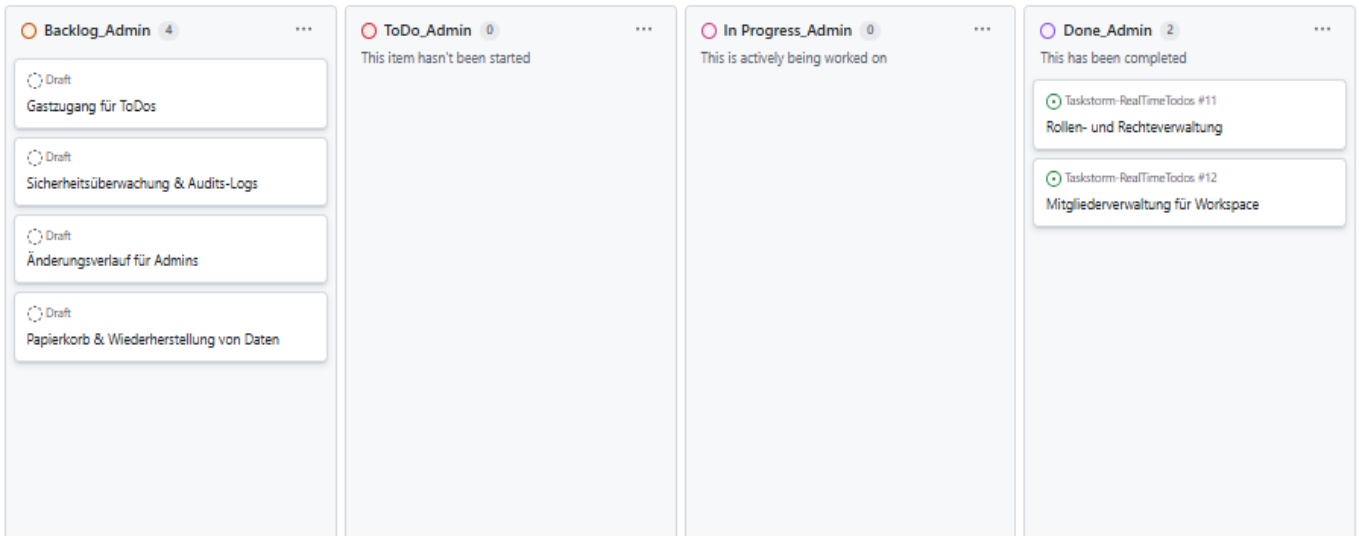

Abbildung 5. Überblick über geplante und umgesetzte Funktionen im GitHub-Projektboard - Admin

Die entwickelte Applikation realisiert die im Anforderungskatalog aufgeführten Kernfunktionen in entscheidenden Bereichen und ermöglicht dadurch eine funktionale, kollaborative To-do-Verwaltung in Echtzeit. Die priorisierte Implementierung umfasst wesentliche Aspekte der individuellen Nutzung, ebenso wie zentrale Anforderungen der kollaborativen Organisation. Die zugrunde liegende modulare Architektur ermöglicht es, dass in künftigen Entwicklungszyklen zusätzliche Funktionen ohne tiefgreifende Umstrukturierungen eingebunden werden können. Das resultierende Rahmenkonzept erweist sich damit als leistungsfähig und erfüllt die Kernanforderungen in hohem Maße, bietet aber gleichzeitig Raum für weitergehenden Optimierungsbedarf.

4. Evaluierung & Fazit

Die nachfolgenden Abschnitte widmen sich der abschließenden Evaluation des entwickelten To-do-Systems und beleuchten nochmals zentrale Kriterien wie Performance, Datenintegrität und Nutzerfreundlichkeit. In diesem Zusammenhang erfolgt eine kritische Prüfung der gewählten Datenbanktechnologien.

4.1. Bewertungskriterien und Evaluierung der Umsetzung

Die Entwicklung einer kollaborativen Echtzeit-To-do-Anwendung ist mit einer zentralen Herausforderung verbunden, nämlich der Sicherstellung von sowohl hoher Zugriffsgeschwindigkeit als auch zuverlässiger Datenspeicherung. Obwohl eine rein relationale Datenbank wie SQLite ACID-Konformität und robuste Langzeit-Persistenz gewährleistet, zeigt sie in Szenarien mit sehr hohem Datenaufkommen und unmittelbaren Aktualisierungen eine geringere Performance. In solchen Fällen bietet sich der Einsatz von Redis an. Als In-Memory-Datenbank mit Pub/Sub-Funktionalität ermöglicht Redis nahezu verzögerungsfreie Lese- und Schreibzugriffe im Millisekundenbereich sowie eine umgehende Benachrichtigung sämtlicher verbundener Clients bei Änderungen (z. B. an To-dos). Ein weiterer Vorteil von Redis ist, dass es nicht an ein starres Datenbank-Schema gebunden ist. Key-Value-Paare oder Listen können dynamisch angelegt und an sich wandelnde Anforderungen angepasst werden, ohne kostspielige Migrationsprozesse wie in strikt relationalen Systemen.

In dieser Architektur übernimmt SQLite die permanente Datenspeicherung, indem es alle relevanten Informationen (z. B. To-dos, Nutzerbeziehungen) konsistent und dauerhaft ablegt. Die ACID-Eigenschaften von SQLite verhindern Inkonsistenzen, während Redis kurzfristige Änderungen puffert, die bei einer Offline-Phase eines Nutzers zwischenzeitlich entstehen. Nach Wiederherstellung der Verbindung werden die verpassten Aktualisierungen automatisch nachgeladen, ohne dass zusätzliche Mechanismen in der relationalen Datenbank erforderlich

sind. Die Kombination dieser beiden Technologien überwindet die Einschränkungen einzelner Ansätze, wie sie in der Nutzung von SQLite oder Redis zu beobachten sind, und gewährleistet, dass sämtliche Anforderungen an eine schnelle Kommunikation, ein flexibles Datenmanagement und eine zuverlässige Persistenz in vollem Umfang erfüllt werden.

4.2. Fazit & Weiterentwicklungen

Die vorgestellte To-do-Anwendung kombiniert das Beste aus beiden Welten, indem sie Redis als In-Memory-Datenbank und SQLite als relationales Datenbanksystem einsetzt. Dadurch wird eine zuverlässige Echtzeitkommunikation mit minimalen Latenzzeiten realisiert, während zugleich die dauerhafte Speicherung und Integrität der Daten sichergestellt ist. Insbesondere im Kontext der kollaborativen Aufgabenverwaltung erweist sich dieser Ansatz als leistungsstarkes Fundament, da Änderungen wie das Hinzufügen oder Aktualisieren von To-dos umgehend bei allen teilnehmenden Nutzern sichtbar sind.

Langfristig bieten sich verschiedene Möglichkeiten, das System zu erweitern. Hierzu zählen etwa eine tiefere Offline-Funktionalität mit umfangreicherer Zwischenspeicherung komplexer To-dos, die Anbindung externer Kalenderdienste oder die Integration fortgeschrittener Reporting-Optionen, wodurch der Nutzen für Teams weiter gesteigert werden könnte. Darüber hinaus wäre die Implementierung eines erweiterten Rollen- und Rechtemanagements vorstellbar, was die Anwendung insbesondere in professionellen Umgebungen noch attraktiver gestalten würde. Die Trennung von Echtzeit- und Persistenzschicht, die durch die modulare Architektur sichergestellt wird, gewährleistet die Flexibilität des Systems, um auf steigende Benutzerzahlen und wechselnde Anforderungen zu reagieren, während es sowohl in Bezug auf die Performance als auch auf die Zuverlässigkeit überzeugt.

IV. Literaturverzeichnis

Equant. (kein Datum). *Hybrid Approaches: Combining SQL and NoSQL*. Abgerufen am 10. März 2025 von https://equant.org/articles/hybrid-database/#:~:text=Hybrid%20Approaches%3A%20Combining%20SQL%20and%20NoSQL&text=One%20common%20approach%20is%20to,databases%20for%20various%20data%20types.

Fette, I. &. (2011). *The WebSocket Protocol.* Internet Engineering Task Force (IETF). Abgerufen am 6. Februaar 2025 von https://datatracker.ietf.org/doc/html/rfc6455

Hecht, R., & Jablonski, S. (2011). NoSQL Evaluation. 336-341. Abgerufen am 10. März 2025

Hipp, D. R. (2023). *About SQLite*. Abgerufen am 1. März 2025 von https://www.sqlite.org/about.html#:~:text=SQLite%20is%20a%20compact%20library,And%20some%20compiler%20optimizations

Jones, M. B. (2015). *JSON Web Token (JWT).* Internet Engineering Task Force (IETF) . Abgerufen am 6. Februar 2025 von https://datatracker.ietf.org/doc/html/rfc7519

Kaufmann, M. &. (2023). *SQL- & NoSQL-Datenbanken.* Springer Vieweg Berlin, Heidelberg. Abgerufen am 6. Februar 2025 von https://link.springer.com/book/10.1007/978-3-662-67092-7

Kim, N. (Juni 2024). *Why Is Redis a Distributed Swiss Army Knife*. Abgerufen am 4. Februar 2025 von https://newsletter.systemdesign.one/p/redis-use-cases

Moniruzzaman, A. B. (2013). NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. (I. J. Application, Hrsg.) 1-14. Abgerufen am 4. Februar 2025

O. Haase, P. D. (kein Datum). Verteilte Systeme. Verteilte Architekturen. 15. Abgerufen am 25. März 2025 von http://www-home.fh-konstanz.de/~haase/lehre/versy/slides/v3VerteilteArchitekturen.pdf

Obe, R., & Hsu, L. (2015). *PostgreSQL: Up and Running.* Media, O'Reilly. Abgerufen am 10. März 2025 von https://github.com/faisalbasra/postgres_books/blob/master/002-OReilly.PostgreSQL.Up.and.Running.2nd.Edition.2014.12.pdf

Redis. (2025). *Redis configuration file example*. Abgerufen am 10. März 2025 von https://redis.io/docs/latest/operate/oss_and_stack/management/config-file/

Redis. (2025). *Redis Data Integration*. Abgerufen am 10. März 2025 von https://redis.io/docs/latest/integrate/redis-data-integration/quick-start-guide/

Redis. (2025). *Redis persistance*. Abgerufen am 10. März 2025 von https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/

Redis. (2025). *Redis persistence*. Abgerufen am 6. Februar 2025 von https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/

Redis. (2025). *Redis Pub/Sub*. Abgerufen am 6. Februar 2025 von https://redis.io/docs/latest/develop/interact/pubsub/

Suranga, S. (2022). *Using WebSockets with Fastify*. Abgerufen am 10. März 2025 von https://blog.logrocket.com/using-websockets-with-fastify/

Tanenbaum S., &. v. (2007). *Verteilte Systeme: Prinzipien und Paradigmen.* Pearson Studium. Abgerufen am 25. März 2025

Usunov, E., & Chalkin, A. (2024). *fastify-redis-channels*. Abgerufen am 10. März 2025 von https://github.com/hearit-io/fastify-redis-channels

Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir unseren Anforderungskatalog mit dem Titel

"TaskStorm: Skalierbare und flexible Datenbanklösung mit NoSQL"

selbständig verfasst und keine anderen als die angegebenen Quellen und erlaubten Hilfsmittel benutzt habe. Alle wörtlichen Zitate in der Arbeit wurden durch Anführungszeichen eindeutig gekennzeichnet. Die Arbeit wurde in gleicher oder ähnlicher Form bei keiner anderen Prüfung vorgelegt.

Der Textteil der Arbeit umfasst 3973 Wörter.

Heidenheim an der Brenz, den 30.03.2025