Siemens Energy - BETI 2024

Realisierung eines Webauftritts

Socialmedia RPG

Echo

Ogulcan Kuecuek

Leon Woenckhaus

Nick Hildebrandt

Aaron Turyabahika

Andre Seiler

16. Februar 2025

Inhaltsverzeichnis

1 Projektbeschreibung **1**
1.1 Projektbegründung . . . 2
1.2 Projektabgrenzung . . . 2

2 Projektplanung **4**
2.1 Teamaufbau und Rollen . . . 5
2.2 Ressourcenplanung . . . 5
2.3 Kostenplanung . . . 6
2.4 Zeitplanung . . . 6

3 Zielplattform und Implementierung **8**
3.1 Architekturdesign . . . 8
3.2 Benutzeroberfläche . . . 10
3.3 Datenmodell . . . 11

4 API-Übersicht **13**
4.1 GET: `/api/awards` . . . 13
4.2 POST: `/api/awards` . . . 14
4.3 DELETE: `/api/awards/[id]` . . . 15
4.4 DELETE: `/api/comments/[id]` . . . 16
4.5 GET: `/api/comments/[id]` . . . 16
4.6 GET: `/api/comments` . . . 17
4.7 POST: `/api/comments` . . . 18
4.8 POST: `/api/comments/search` . . . 19
4.9 DELETE: `/api/comments/[id]/like` . . . 20
4.10 POST: `/api/comments/[id]/like` . . . 21
4.11 DELETE: `/api/communities/[id]` . . . 22
4.12 GET: `/api/communities/[id]` . . . 23
4.13 GET: `/api/communities` . . . 24
4.14 POST: `/api/communities` . . . 25
4.15 POST: `/api/communities/search` . . . 26
4.16 GET: `/api/communities/[id]/feed` . . . 27
4.17 DELETE: `/api/communities/[id]/join` . . . 27
4.18 POST: `/api/communities/[id]/join` . . . 28
4.19 GET: `/api/communities/[id]/users` . . . 28
4.20 GET: `/api/images/[id]` . . . 29
4.21 POST: `/api/images/[type]` . . . 30
4.22 POST: `/api/logins` . . . 31

4.23 DELETE: /api/logins . . . 32
4.24 DELETE: /api/posts/[id] . . . 33
4.25 GET: /api/posts/[id] . . . 34
4.26 GET: /api/posts . . . 34
4.27 POST: /api/posts . . . 35
4.28 POST: /api/posts/search . . . 36
4.29 GET: /api/posts/[id]/comments . . . 37
4.30 DELETE: /api/posts/[id]/like . . . 38
4.31 POST: /api/posts/[id]/like . . . 39
4.32 GET: /api/users/feed . . . 40
4.33 DELETE: /api/users/[id] . . . 40
4.34 GET: /api/users/[id] . . . 41
4.35 GET: /api/users . . . 41
4.36 POST: /api/users . . . 42
4.37 POST: /api/users/search . . . 43
4.38 GET: /api/users/streak . . . 43
4.39 GET: /api/users/[id]/awards . . . 44
4.40 GET: /api/users/[id]/comments . . . 45
4.41 GET: /api/users/[id]/communities . . . 45
4.42 DELETE: /api/users/[id]/follow . . . 46
4.43 POST: /api/users/[id]/follow . . . 46
4.44 GET: /api/users/[id]/posts . . . 47
4.45 Datenzugriff und Backend-API Routen . . . 48

5 Abnahmephase **50**
5.1 Bereitstellung . . . 50
5.2 Fazit . . . 51

Abbildungsverzeichnis

1 ERM Schema 12

Abkürzungsverzeichnis

API engl. ”application programming interface”, ist eine Sammlung von Definitionen und Protokollen, die es Softwareanwendungen ermöglichen, miteinander zu kommunizieren. 48

CRUD engl. ”Create, Read, Update, Delete”, die Funktionen, die ein Nutzer benötigt, um Daten anzulegen und zu verwalten. 2, 48

JSON engl. “JavaScript Object Notation” ist ein kompaktes Datenformat, das in einer leicht lesbaren Textform den Datenaustausch zwischen Anwendungen ermöglicht.. 48

RPG engl. ”role-playing games”, sind ein Genre von Spielen, in denen die Spieler in die Rolle eines imaginären Charakters schlüpfen. 1, 2, 8

SSL engl. ”Secure Sockets Layer”, ein Protokoll, das die Sicherheit der Kommunikation über das Internet gewährleistet. 3, 8

SSR engl. ”Server Side Rendering”, ist Technik, Webseiten auf dem Server zu rendern, bevor sie an den Client-Browser gesendet werden. 2, 8

XP engl. ”experience points”, sind ein Konzept in Spielen, das den Fortschritt eines Charakters oder Spielers zeigt. 1

1 Projektbeschreibung

Das Social Media Projekt „Echo“ ist ein innovatives, kompetitives Social Media Netzwerk, das speziell für Gamer, Digital Natives und Content Creator entwickelt wurde. Echo kombiniert die Elemente traditioneller Social Media Plattformen wie Reddit mit Rollenspiel-Mechaniken (engl. ”role-playing games”, sind ein Genre von Spielen, in denen die Spieler in die Rolle eines imaginären Charakters schlüpfen (RPG)), um ein dynamisches und interaktives Nutzererlebnis zu schaffen.

Nutzer sammeln Erfahrungspunkte (engl. ”experience points”, sind ein Konzept in Spielen, das den Fortschritt eines Charakters oder Spielers zeigt (XP)), indem sie Likes und Kommentare auf ihre Posts von anderen Nutzern erhalten. Diese XP sind ein Maß für die Aktivität und Beliebtheit eines Nutzers innerhalb der Plattform. Zusätzlich zu den XP können Nutzer durch sogenannte „Streaks“ weitere Erfahrungspunkte sammeln. Ein Streak entsteht, wenn ein Nutzer über mehrere aufeinanderfolgende Tage hinweg aktiv ist und regelmäßig Inhalte postet oder mit anderen interagiert. Je länger der Streak, desto höher die Belohnung.

Ebenfalls können Benutzer anderen Accounts folgen, um aus diesen personalisierte Inhalte zu bekommen und keine Neuigkeiten mehr zu verpassen. Dieser Wert an sogenannten Followern wird ebenfalls gezählt.

Die gesammelten Erfahrungspunkte ermöglichen es den Nutzern, ihr Profil und das Design der Website individuell anzupassen. Dies umfasst personalisierte Themes, exklusive Avatare und spezielle Layouts, die das Profil einzigartig machen. Ab einer gewissen Anzahl an XP können Nutzer ihr Profilbild, Bannerbild und Hintergrundbild selbst wählen, was zusätzliche Individualisierungsmöglichkeiten bietet.

Ein weiteres zentrales Element von Echo sind die sogenannten Communities, auf denen sich Nutzer sammeln können (Beitreten) und diese Awards durch andere Benutzer verliehen bekommen können, die im Profil angezeigt werden. Auf Communities gibt es genauso wie bei Benutzern die Möglichkeit, Posts mit Kommentaren und Likes zu versehen. Dies fördert die Interaktion und das Gemeinschaftsgefühl innerhalb der Plattform.

Im Mittelpunkt von Echo stehen Gamification-Elemente, Gruppenzugehörigkeit und das Belohnungsgefühl. Nutzer werden durch kontinuierliche Belohnungen und Fortschritte motiviert, aktiv zu bleiben und sich in der Community zu engagieren.

1.1 Projektbegründung

Das Social Media Projekt „Echo“ zeichnet sich durch seine einzigartigen Gamification- und Rollenspiel-Elemente (RPG) aus, die es zu einem besonderen Sammelpunkt für Gamer und die restliche Internetkultur machen. Diese Elemente fördern nicht nur die Interaktivität und das Engagement der Nutzer, sondern schaffen auch ein dynamisches und wettbewerbsorientiertes Umfeld, das die Nutzer motiviert, aktiv zu bleiben und sich kontinuierlich weiterzuentwickeln.

Ein weiteres technisches Highlight von Echo ist der innovative Caching-Algorithmus, der sicherstellt, dass Bilder nicht doppelt gespeichert werden. Beim Hochladen von Bildern werden doppelte Dateien erkannt und durch einen Verweis auf das bereits vorhandene Bild ersetzt. Diese Methode spart nicht nur wertvolle Speicherressourcen, sondern trägt auch zur Schonung der Umwelt bei. Durch die Reduzierung des Speicherbedarfs wird der Energieverbrauch der Server gesenkt, was wiederum den CO2-Ausstoß verringert und somit einen positiven Beitrag zum Umweltschutz leistet. Laut einer Studie des Borderstep Instituts für Innovation und Nachhaltigkeit verursachen Rechenzentren in Deutschland jährlich etwa 13 Millionen Tonnen CO2-Emissionen [1], was die Bedeutung ressourcenschonender Technologien unterstreicht.

Die Motivation für das Projekt war es, eine minimalistische und schnelle Plattform speziell für Gamer zu entwickeln, um die Gaming- und Internetkultur in einen diversifizierten multimedialen Austausch zu stellen. Echo kombiniert technologische Innovation mit einer klaren Zielgruppenausrichtung, um eine Plattform zu schaffen, die sowohl funktional als auch nachhaltig ist. Die Integration von Gamification- und RPG-Elementen macht Echo zu einem einzigartigen Erlebnis für Nutzer, während die ressourcenschonende Technologie die Umweltbelastung minimiert. Diese Kombination aus Innovation und Zielgruppenfokussierung macht Echo zu einer herausragenden Plattform im Bereich der sozialen Medien.

1.2 Projektabgrenzung

Das Social Media Projekt „Echo“ wird durch ein serverseitig gerendertes Frontend realisiert (engl. ”Server Side Rendering”, ist Technik, Webseiten auf dem Server zu rendern, bevor sie an den Client-Browser gesendet werden (SSR)), das eine nahtlose und schnelle Benutzererfahrung gewährleistet. Dazu wird eine API entwickelt, die Daten aus einer relationalen Datenbank über die CRUD-Operationen (engl. ”Create, Read, Update, Delete”, die Funktionen, die ein Nutzer benötigt, um Daten anzulegen und zu verwalten (CRUD)) zur Verfügung stellt. Diese Architektur ermöglicht eine effiziente und skalierbare Datenverwaltung, die den Anforderungen einer dynamischen Social Media Plattform gerecht wird.

Das Projekt wird umfassend dokumentiert, wie in diesem Dokument beschrieben, und zusätzlich durch eine Abschlusspräsentation ergänzt. Diese Präsentation wird die wichtigsten Aspekte und Ergebnisse des Projekts zusammenfassen und visuell ansprechend darstellen.

Für die Vorstellung des Projekts wird „Echo“ auf einem Server im Internet bereitgestellt und über eine eigene Domain mit einem entsprechenden SSL-Zertifikat ((engl. ”Secure Sockets Layer”, ein Protokoll, das die Sicherheit der Kommunikation über das Internet gewährleistet (SSL))) erreichbar gemacht. Dies stellt sicher, dass die Plattform sicher und zuverlässig zugänglich ist und den modernen Sicherheitsstandards entspricht.

2 Projektplanung

Zu Beginn des Projekts haben wir uns als Gruppe zusammengefunden und die grundlegenden Ideen und Funktionalitäten auf mehreren Flipchartblättern diskutiert. In intensiven Debatten haben wir die verschiedenen Aspekte des Projekts durchgesprochen, um am Ende einen sehr abstrakten Mockup zu erstellen, der zeigte, wie unser Projekt aussehen sollte und welche Funktionen bzw. RPG-Elemente es enthalten sollte.

Anschließend haben wir diese Funktionen in kleinere Issues aufgeteilt, die wie folgt beschrieben und aufgebaut sind: Eine kurze, prägnante Beschreibung des vorgeschlagenen Features, die erklärt, was es neu macht und warum es sinnvoll ist. Eine Liste der konkreten Funktionen, die das Feature umfassen wird. Eine Beschreibung des idealen Nutzerflusses für dieses Feature, die erklärt, wie der Nutzer mit dem Feature interagiert. Eine Auflistung der Technologien, die für die Frontend-Implementierung verwendet werden, sowie spezielle Designanforderungen, wie z.B. die Verwendung von Icons. Eine Erklärung, wie die Benutzereinstellungen in der Datenbank gespeichert werden, z.B. durch ein neues Dokument pro Benutzer mit den entsprechenden Feldern, sowie spezifische Anforderungen an die Datenverarbeitung oder Sicherheit.

Diese Issues dienen dabei gleichzeitig auch als Basisdokumentation der Funktionalität, aus der wir dieses Dokument technisch ableiten. Die Dokumentation selbst sowie die Präsentation werden über die Git-Versionskontrolle verwaltet und über Issues getrackt. Diese Issues wurden dank Kategorien wie Kernfunktionalität, optionale Funktionalität, Backend-API und Frontend zugewiesen. Zudem konnten einige Issues andere voraussetzen, wie z.B. dass ein Login die Registrierung voraussetzt. Mehrere Issues bildeten dann Meilensteine, die wir datieren konnten.

Da wir eine agile Arbeitsweise nach Scrum verwenden, gibt es tägliche Standups, in denen jeder sagt, was er gerade getan hat, welche Probleme es dabei gab und was er als nächstes machen wird. Die Entwicklungsarbeit und Versionsverwaltung wird dann durch Git realisiert, mit der Cloud-Implementierung von GitHub, um den Entwicklungsstand aus allen Computern zu synchronisieren. Git ist ein verteiltes Versionskontrollsystem, das es uns ermöglicht, Änderungen am Code effizient zu verfolgen und zusammenzuführen. Damit es zu keinen Konflikten in der gleichzeitigen Bearbeitung von ein und derselben Datei durch mehrere Leute kommt, wurde für jede Aufgabe ein eigener Entwicklungszweig erstellt. Diese wurden nach Beendigung wieder in den Main-Zweig zurückimplementiert, um eine lauffähige Version unseres Projekts stets im Main zu behalten.

Um einen Überblick über laufende und abgeschlossene Aufgaben zu haben, wurde auf GitHub ein Kanban-Board eingerichtet (3 Spalten: Offen, In Bearbeitung und Fertig), bei dem erledigte Aufgaben nach gemeinsamer Bewertung und Verbesserung auf „Fertig“

gestellt wurden. Nachdem wir zunächst die Backend-API gemeinsam mit dem Frontend bearbeitet hatten, mussten wir aufgrund der verschiedenen domänenspezifischen Anforderungen unseren Entwicklungsprozess umstellen und das Frontend und Backend parallel, aber getrennt durch verschiedene Teammitglieder entwickeln, um das jeweilige Können optimal zu allocieren.

2.1 Teamaufbau und Rollen

1. Projektmanagement
    - Nick Hildebrandt
2. Dokumentation
    - Leon Woenckhaus
3. Präsentation
    - Aaron Turyabahika
4. Backend-API
    - Nick Hildebrandt
    - Leon Woenckhaus
5. Frontend
    - Andre Seiler
    - Ogulcan Kuecuek
    - Aaron Turyabahika
6. Deploayment und integration
    - Nick Hildebrandt

2.2 Ressourcenplanung

Detaillierte Planung der benötigten Ressourcen (Hard-/Software, Räumlichkeiten usw.).

Ggfs. sind auch personelle Ressourcen einzuplanen (z.B. unterstützende Mitarbeiter).

Hinweis: Häufig werden hier Ressourcen vergessen, die als selbstverständlich ange- sehen werden (z.B. PC, Büro).

2.3 Kostenplanung

2.4 Zeitplanung

Der Plan sieht vor, dass der Cold Freeze am 10. Februar 2025 und der Hard Freeze am 13. Februar 2025 stattfinden. Die Präsentation war ursprünglich für den 17. Februar 2025 zur Überprüfung angesetzt. Das korrigierte Präsentationsdatum ist nun der 20. Februar 2025, wobei der Cold Freeze auf den 13. Februar 2025 und der Hard Freeze auf den 16. Februar 2025 verschoben wurde. Im Softwareentwicklungsprozess bezeichnet der Cold Freeze den Zeitpunkt, ab dem keine neuen Features mehr hinzugefügt werden. Der Fokus liegt ab diesem Zeitpunkt auf der Stabilisierung und Fehlerbehebung. Der Hard Freeze markiert den endgültigen Stopp aller Änderungen am Code, um sicherzustellen, dass die Software für die Veröffentlichung vorbereitet ist.

Das Projekt wird innerhalb eines festgelegten Zeitrahmens durchgeführt, wobei die tägliche Arbeitszeit auf 8 Stunden pro Person begrenzt ist. Der Projektumfang wurde so geplant, dass die reguläre Arbeitszeit von 8 Stunden pro Tag pro Person ausreicht, um das Projekt abzuschließen. Sollten Teammitglieder bereit sein, zusätzlichen Aufwand zu investieren, können weitere Features und Verbesserungen implementiert werden, die über die ursprünglichen Anforderungen hinausgehen.

Zunächst wurde das Geschäftsmodell und die Grundidee unseres Projektes im Rahmen des betriebswirtschaftlichen Unterrichts in der Woche vom 2. bis 6. Dezember 2024 in Form eines Business Model Canvas geplant. Im Januar wurde viel Zeit in die detaillierte Planung des Projekts und des Projektumfangs investiert. Die Umsetzung begann in der Woche vom 13. bis 19. Januar 2025. In dieser Phase wurde die Grundidee spezifiziert, technische Fähigkeiten erlernt und kollaborative Arbeitsprozesse mit GitHub eingerichtet.

Von 20. Januar bis 12. Februar 2025 erfolgte die technische Umsetzung von Frontend und Backend, wobei auftretende Probleme behandelt und Funktionen weiter spezifiziert wurden. In der Woche vom 13. bis 19. Februar 2025 wurde die Projektdokumentation erstellt und die Präsentation durch die jeweils verantwortlichen Teammitglieder vorbereitet. Gleichzeitig wurde die fertige Version getestet und gemäß unserem Deployment-Plan auf einem Server bereitgestellt.

Zeitraum	**Aktivitäten**
02.12.2024 - 06.12.2024	Planung des Geschäftsmodells und der Grundidee im betriebswirtschaftlichen Unterricht in Form eines Business Model Canvas
Januar 2025	Detaillierte Planung des Projekts und des Projektumfangs
13.01.2025 - 19.01.2025	Spezifizierung der Grundidee, Erlernen technischer Fähigkeiten und Einrichtung kollaborativer Arbeitsprozesse mit GitHub
20.01.2025 - 12.02.2025	Technische Umsetzung von Frontend und Backend, Behandlung auftretender Probleme und weitere Spezifizierung der Funktionen
13.02.2025 - 19.02.2025	Erstellung der Projektdokumentation und Vorbereitung der Präsentation durch die jeweils verantwortlichen Teammitglieder, Testen der fertigen Version und Bereitstellung auf einem Server gemäß dem Deployment-Plan
10.02.2025	Cold Freeze: Keine neuen Features werden hinzugefügt, Fokus auf Stabilisierung und Fehlerbehebung
13.02.2025	Hard Freeze: Endgültiger Stopp aller Änderungen am Code, um die Software für die Veröffentlichung vorzubereiten
16.02.2025	Hard Freeze (korrigiert): Endgültiger Stopp aller Änderungen am Code
20.02.2025	Präsentation des Projekts

Gant Diagramm hier, Tabelle schöner

3 Zielplattform und Implementierung

Zur Auswahl der Zielplattform gehören unter anderem die Programmiersprache, die Datenbank, Client/Server-Architektur und die Hardware.

Unsere Zielplattform für die Serverseite ist Linux, da Linux-basierte Server weit verbreitet sind. Für den Webbrowser setzen wir auf Gecko, WebKit und V8 für serverseitiges Rendering. Das Backend wird mit Node.js auf Linux betrieben.

Node.js gilt als Goldstandard (Quelle hier). Wir haben uns für Node.js statt Deno entschieden, da es zuvor Kompatibilitätsprobleme mit Deno gab. Node.js ist jedoch besser etabliert und verfügt aufgrund seiner längeren Existenz über mehr Dokumentation. Außerdem ist Node.js zum Zeitpunkt des Projektbeginns besser mit Nuxt kompatibel.

zur Auswahl der Zielplattform (u.a. Programmiersprache, Datenbank, Client/Server, Hardware). Zielplatform: Linux Webbrowser gecko, webkit, v8 serverside rendered nodejs Linux backend was ist node node goldstandard (quelle hier) node.js statt deno: Zuvor mit deno kompalitbitätsproblem. Node allerdings besser etabliert, mehr Dokumentation aufgrund längerem bestehen. Node.js ist ausserdem besser kompatibel mit Nuxt zum Zeitpunkt des Projektbeginns.

3.1 Architekturdesign

Für unser Projekt haben wir uns für die Nutzung des Nuxt.js-Frameworks entschieden. Nuxt.js ist ein leistungsstarkes Framework, das auf Vue.js aufbaut und die Entwicklung von serverseitig gerenderten (SSL) und statisch generierten Anwendungen vereinfacht. Durch diese Trennung wird die Wartbarkeit und Erweiterbarkeit der Anwendung erheblich verbessert. Dies ist für uns von großem Interesse, da wir unser Projekt so aufsetzen wollen, dass wir es in der Zukunft um weitere Features erweitern können. So können wir zunächst den Social Media Aspekt des Projekts umsetzen, um darauf die Role-Playing-Game (RPG) Elemente langsam aufzubauen. Dadurch können wir nach der Fertigstellung des Social Media Grundgerüsts jederzeit einen funktionierenden Prototypen präsentieren.

Unsere Wahl für das Framework fiel auf Nuxt.js, da es ein leistungsstarkes Framework ist. Es baut auf Vue.js auf und vereinfacht die Entwicklung von serverseitig gerenderten (SSR) und statisch generierten Anwendungen. Ein zentrales Konzept in Nuxt.js sind die Komponenten. Vue-Komponenten ermöglichen es, die Anwendung in wiederverwendbare und isolierte Module zu unterteilen. Diese Module können mehrfach verwendet und für verschiedene Anwendungsfälle angepasst werden, was die Entwicklung effizienter und die Codebasis übersichtlicher macht. Die Komponenten erlauben uns auch, das Projekt effizient zu erweitern. Daher eignet sich Nuxt.js ideal als Framework für unsere Anforderungen an das Projekt.

Unsere Wahl viel auf Nuxt über andere ähnlich aufgebaute Frameworks wie Next aufgrund folgender Auswahlkriterien: Nuxt.js bietet umfassende Unterstützung für serverseitiges Rendering und statische Seitengenerierung, was es zu einer idealen Wahl für Fullstack-Anwendungen macht. Für Nuxt existiert eine ausführliche Dokumentation und eine Vielzahl an Tutorials. Der Einstieg und die kontinuierliche Weiterentwicklung der Anwendung wird dadurch vereinfacht und Gruppenmitglieder mit weniger Programmier- Erfahrung können schneller in den Workflow eingebunden werden. Nuxt.js ist kompatibel mit den bestehenden Technologien und Umgebungen, die wir nutzen möchten, was eine nahtlose Integration und Migration ermöglicht.

Ggfs. Bewertung und Auswahl von verwendeten Frameworks sowie ggfs. eine kurze Einführung in die Funktionsweise des verwendeten Frameworks.

Prisma/SQLite Backend

Für die Umsetzung des Backends ist eine Datenbank unverzichtbar. SQLite ist für unser Projekt besonders gut geeignet, da es auch bei einer großen Anzahl von Einträgen und Abfragen eine hohe Geschwindigkeit und Effizienz bietet. Allerdings kann die direkte Einbindung von SQLite in PHP-Anwendungen die Anwendung anfällig für SQL-Injections machen. Um dieses Sicherheitsrisiko zu minimieren, haben wir uns entschieden, ein Object-Relational Mapping (ORM) zu verwenden.*(ORM Abkürzung in document handeln)* Aufgrund der vorhandenen Kompatibilität mit Nuxt.js und Node.js haben wir uns für Prisma als ORM entschieden. Prisma fungiert als eine Schicht zwischen der Datenbank und der Anwendung, die es ermöglicht, Datenbankabfragen sicher und effizient durchzuführen. Es bietet eine typsichere API, die die Entwicklung vereinfacht und gleichzeitig die Sicherheit erhöht, indem es SQL-Injections verhindert. Prisma unterstützt zudem die Migration und Verwaltung der Datenbankstruktur, was die Wartung und Weiterentwicklung der Anwendung erleichtert.

Im Rahmen des Deployments wird Nginx als Reverse HTTPS Proxy eingesetzt. Nginx übernimmt dabei die Aufgabe, eingehende Anfragen an die entsprechenden Backend-Server weiterzuleiten und sorgt so für eine effiziente Lastverteilung und erhöhte Sicherheit. Darüber hinaus wird Nginx auch für die Verwaltung der SSL-Zertifikate zuständig sein, um eine sichere HTTPS-Verbindung zu gewährleisten.*(was ist nginx quelle:https://nginx.org/en/)*

Für die automatische Verwaltung und Erneuerung der SSL-Zertifikate nutzen wir das ACME-Protokoll (Automated Certificate Management Environment). ACME ist ein Protokoll, das von der Internet Security Research Group (ISRG) entwickelt wurde und es ermöglicht, SSL/TLS-Zertifikate automatisch zu beziehen und zu erneuern. Dies reduziert den administrativen Aufwand und stellt sicher, dass unsere Zertifikate stets aktuell und sicher sind.(*acme acronym in document handeln, absatz review Nick*)

3.2 Benutzeroberfläche

Entscheidung für die gewählte Benutzeroberfläche (z.B. GUI, Webinterface).Beschreibung des visuellen Entwurfs der konkreten Oberfläche (z.B. Mockups, Menü- führung). Inspirationen: Instagram, Bluesky, Steam, Discord

Der strukturelle Aufbau unserer Seite ist von verschiedenen Social-Media-Plattformen inspiriert. Die Idee, den Nutzern zu erlauben, Communities zu erstellen, ist vom Prinzip der sogenannten Subreddits abgeleitet. Auf der Forenseite Reddit können Subsites erstellt werden, die sich thematisch voneinander abgrenzen und jeweils eigene Regeln haben. Als klassische Forenseite ist Reddit jedoch nicht besonders intuitiv zu bedienen.

Wir haben uns außerdem Discord angesehen, eine Plattform, die eine eigene Desktop-Applikation bietet und vor allem als Voice- und Text-Chat-Programm bekannt ist. Discord ermöglicht es den Nutzern, ihre eigenen sogenannten Server zu erstellen, was besonders in der Gaming-Community ein viel genutztes Feature ist. Allerdings ist es für normale Nutzer, die wenig über die Community-basierte Natur der Server wissen, nicht offensichtlich, dass ein großer Social-Media-Aspekt in dieser Chat-Anwendung integriert ist.

Beide Seiten haben also ihre Mankos bei der Benutzerfreundlichkeit. Bei Instagram hat der Nutzer über eine Leiste einfachen Zugriff auf seinen Feed, sein Profil, die Suche und die Erkundung. Diese Simplizität macht Instagram sehr benutzerfreundlich. Wir wollten durch den Einsatz eines festen Headers auf unserer Seite diese Nutzerfreundlichkeit nachahmen. Die Communities sowie die Community-Suche und -Erkundung sollen direkt eingebunden sein, sodass es für die Nutzer einfach ist, diese Features zu finden und zu bedienen.

Komponenten der Seite von anderen Seiten inspriert und möglichs user friendly gemacht; Profil inspo twitter, follower list Insta, eigenene Ideen zb: Badge hinzufügen, Inspo für badges an sich Steam

Tatsächliche Umsetzung by 'what do we need' Nuxt UI elemente statt CSS

Icons statt text simpler nicht schreiben sondern zeigen Zahnrad einstellungen, plus hinzufügen, Pictogramme simple modern

Steam RPG inspo, Motivation (alle von uns steam user, steam als mehr social statt game installer website)

Ggfs. Erläuterung von angewendeten Richtlinien zur Usability und Verweis auf Corporate Design.

3.3 Datenmodell

Entwurf/Beschreibung der Datenstrukturen (z.B. ERM und/oder Tabellenmodell, XML-Schemas) mit kurzer Beschreibung der wichtigsten (!) verwendeten Entitäten. ERM einfügen Relationelles Datenbankmodell

Beschreibung der angelegten Datenbank (z.B. Generierung von SQL aus Modellierungswerkzeug oder händisches Anlegen), XML-Schemas usw.

Unsere Datenbank wurde mit Prisma als Object-Relational Mapping (ORM) Tool erstellt und migriert. Jedes in unserem Schema geplante Tabellenobjekt wurde als Modell in Prisma definiert, was eine strukturierte und effiziente Datenverwaltung ermöglicht. Ein Vorteil von Prisma ist die automatische Generierung von Zwischentabellen durch die Verwendung von @relations, was den Entwicklungsprozess erheblich vereinfacht und beschleunigt. Als Backend für das ORM dient SQLlite, das eine zuverlässige und performante Grundlage für unsere Datenbankoperationen bietet. Diese Kombination aus Prisma und SQLlite gewährleistet eine robuste und skalierbare Datenbanklösung, die sowohl den aktuellen als auch zukünftigen Anforderungen unseres Projekts gerecht wird.

Um die wichtigsten Beziehungen in unserem Social-Media-Projekt darzustellen, sind die Modelle für Benutzer, Post, Community und Kommentare von zentraler Bedeutung. Die Benutzertabelle enthält beispielsweise eine eindeutige ID, die jeden Nutzer identifiziert, sowie die Nutzerdaten wie Name und E-Mail-Adresse. Darüber hinaus gibt es Querverweise auf andere Tabellen durch Foreign Keys, die beispielsweise auf die Posts eines Nutzers verweisen. Die Post-Tabelle weist für jeden Post ebenfalls eine eindeutige ID auf und enthält einen Foreign Key, der auf den Autor des Posts verweist. Prisma erstellt automatisch Zwischentabellen zwischen diesen Tabellen, in denen die Post-IDs und Benutzer-IDs hinterlegt werden. Diese automatische Generierung von Zwischentabellen durch Prisma gewährleistet eine effiziente und strukturierte Verwaltung der Datenbankbeziehungen, was die Handhabung und Abfrage unserer Social-Media-Daten erheblich vereinfacht.

(Bild ERM here)

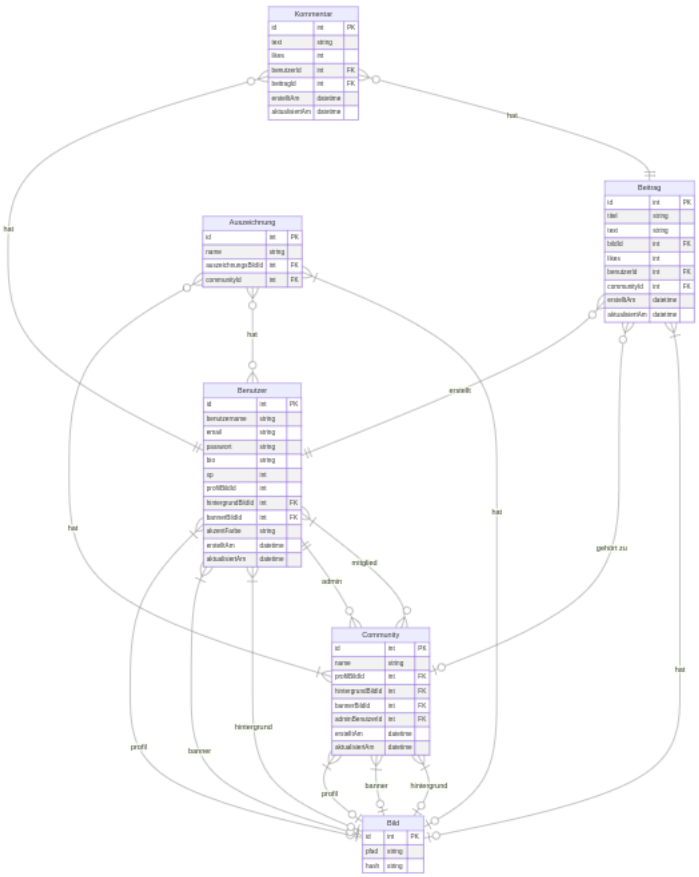

Abbildung 1: ERM Schema

4 API-Übersicht

Diese Dokumentation beschreibt die verschiedenen Routen der `/api/awards`-API. Jede Methode wird mit einer technischen Einleitung erklärt, gefolgt von Details zur Anmeldung, Eingabeparametern, der erwarteten Antwort und möglichen Fehlerfällen.

4.1 GET: `/api/awards`

Diese Route wird verwendet, um eine paginierte Liste aller Awards abzurufen. Die Abfrage erfolgt über die Prisma-Methode `findMany` in der `award`-Tabelle. Die zurückgegebenen Felder werden durch die `awardSelect`-Definition bestimmt. Falls keine Einträge gefunden werden, wird ein `404 Not Found` zurückgegeben. Bei einem Fehler in der Datenbankabfrage wird ein `400 Database request failed` ausgegeben.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung. Standardwert: 1
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite. Standardwert: 10

Antwort:

- Eine Liste von Award-Objekten mit folgenden Feldern:
  - `id` (integer)
  - `awardName` (string)
  - `awardImage` (string)
  - `adminUserId` (integer)
  - `community` (Objekt)
  - `createdAt` (datetime)
  - `updatedAt` (datetime)

Fehlerfälle:

- `404 Not Found`: Es wurden keine Einträge gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.2 POST: `/api/awards`

Diese Route dient zur Erstellung oder Aktualisierung eines Awards. Falls der Body-Parameter `id` angegeben ist, wird der entsprechende Eintrag aktualisiert. Falls keine `id` übergeben wird, wird ein neuer Eintrag erstellt. Die Erstellung erfolgt über die Prisma-Methode `create`, während die Aktualisierung über `update` durchgeführt wird.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `id` (optional, integer): ID des zu aktualisierenden Awards
  - `awardName` (erforderlich, string): Name des Awards
  - `awardImageId` (optional, integer): ID des Award-Bildes
  - `communityId` (erforderlich, integer): ID der zugehörigen Community
  - `userId` (optional, integer): ID des zu verknüpfenden Benutzers

Antwort:

- Das erstellte oder aktualisierte Award-Objekt mit folgenden Feldern:
  - `id`, `awardName`, `awardImage`, `adminUserId`, `community`, `createdAt`, `updatedAt`

Fehlerfälle:

- `400 Bad Request`: Fehlende Pflichtfelder wie `awardName` oder `communityId`.
- `400 The user is not the creator of the award`: Der Benutzer ist nicht berechtigt, den Award zu aktualisieren.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.
- `401 Unauthorized`: Der Benutzer ist nicht angemeldet.

4.3 DELETE: `/api/awards/[id]`

Diese Route wird verwendet, um einen Award anhand seiner `id` zu löschen. Vor der Löschung wird geprüft, ob der angemeldete Benutzer der Ersteller des Awards ist. Die Löschung erfolgt über die Prisma-Methode `delete`.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des zu löschenden Awards

Antwort:

- Es wird eine Bestätigungsmeldung mit dem Status `200 OK` zurückgegeben:
  - {`statusCode: 200, statusMessage: Entry with Id [id] was deleted.`}

Fehlerfälle:

- `404 Not Found`: Der Award wurde nicht gefunden.
- `401 Unauthorized`: Der Benutzer ist nicht der Ersteller des Awards.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.4 DELETE: `/api/comments/[id]`

Diese Route wird verwendet, um einen Kommentar anhand seiner `id` zu löschen. Es wird geprüft, ob der Benutzer berechtigt ist, den Kommentar zu löschen. Die Löschung erfolgt über die Prisma-Methode `delete`.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des zu löschenden Kommentars

Antwort:

- Es wird eine Bestätigungsmeldung mit dem Status `200 OK` zurückgegeben:
  - {`statusCode: 200, statusMessage: Entry with Id [id] was deleted.`}

Fehlerfälle:

- `404 Not Found`: Kommentar wurde nicht gefunden.
- `401 Unauthorized`: Der Benutzer ist nicht der Ersteller des Kommentars.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.5 GET: `/api/comments/[id]`

Diese Route wird verwendet, um einen Kommentar anhand seiner `id` abzurufen. Es wird die Prisma-Methode `findUnique` verwendet.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des abzurufenden Kommentars

Antwort:

- Das Kommentar-Objekt mit folgenden Feldern:
  - `id`, `text`, `user`, `post`, `count.likes`, `createdAt`, `updatedAt`

Fehlerfälle:

- `404 Not Found`: Kommentar wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.6 GET: `/api/comments`

Diese Route wird verwendet, um eine Liste von Kommentaren zu einem bestimmten Post abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Query-Parameter:**
  - `postId` (optional, integer): ID des Posts, zu dem die Kommentare gehören
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Kommentar-Objekten mit folgenden Feldern:
  - `id`, `text`, `user`, `post`, `count.likes`, `createdAt`, `updatedAt`

Fehlerfälle:

- `404 Not Found`: Es wurden keine Kommentare gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.7 POST: `/api/comments`

Diese Route dient zur Erstellung oder Aktualisierung eines Kommentars. Falls `id` nicht angegeben ist, wird ein neuer Kommentar erstellt. Falls `id` vorhanden ist, wird der Kommentar aktualisiert.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `id` (optional, integer): ID des zu aktualisierenden Kommentars
  - `text` (erforderlich, string): Text des Kommentars
  - `postId` (erforderlich, integer): ID des zugehörigen Posts

Antwort:

- Das erstellte oder aktualisierte Kommentar-Objekt mit folgenden Feldern:
  - `id`, `text`, `user`, `post`, `count.likes`, `createdAt`, `updatedAt`

Fehlerfälle:

- `400 Bad Request`: Fehlende Pflichtfelder wie `text` oder `postId`.
- `400 The user is not the creator of the comment`: Der Benutzer ist nicht berechtigt, den Kommentar zu aktualisieren.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.
- `401 Unauthorized`: Der Benutzer ist nicht angemeldet.

4.8 POST: `/api/comments/search`

Diese Route wird verwendet, um Kommentare anhand einer Suchanfrage zu durchsuchen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `query` (erforderlich, string): Der Suchbegriff
  - `userId` (optional, integer): ID des Benutzers, dessen Kommentare durchsucht werden sollen
  - `postId` (optional, integer): ID des Posts, zu dem die Kommentare gehören

Antwort:

- Eine Liste von Kommentar-Objekten mit folgenden Feldern:
  - `id`, `text`, `user`, `post`, `count.likes`, `createdAt`, `updatedAt`

Fehlerfälle:

- `400 Bad Request`: Die Suchanfrage fehlt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.9 DELETE: `/api/comments/[id]/like`

Diese Route wird verwendet, um ein Like von einem Kommentar zu entfernen. Die Route überprüft, ob der Benutzer eingeloggt ist und ob der Kommentar existiert. Nach erfolgreicher Überprüfung wird das Like des angemeldeten Benutzers entfernt.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Kommentars, von dem das Like entfernt werden soll

Antwort:

- Das aktualisierte Kommentar-Objekt mit den folgenden Feldern:
  - `id`, `text`, `user`, `post`, `count.likes`, `createdAt`, `updatedAt`

Fehlerfälle:

- `404 Not Found`: Der Kommentar wurde nicht gefunden.
- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.10 POST: `/api/comments/[id]/like`

Diese Route wird verwendet, um einen Kommentar zu liken. Die Route überprüft, ob der Benutzer eingeloggt ist und ob der Kommentar existiert. Nach erfolgreicher Überprüfung wird dem Kommentar ein Like hinzugefügt.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Kommentars, das geliked werden soll

Antwort:

- Das aktualisierte Kommentar-Objekt mit den folgenden Feldern:
  - `id`, `text`, `user`, `post`, `count.likes`, `createdAt`, `updatedAt`

Fehlerfälle:

- `404 Not Found`: Der Kommentar wurde nicht gefunden.
- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.11 DELETE: `/api/communities/[id]`

Diese Route wird verwendet, um eine Community anhand ihrer `id` zu löschen. Es wird geprüft, ob der Benutzer eingeloggt ist und ob er der Administrator der Community ist. Nach erfolgreicher Prüfung wird die Community gelöscht.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der zu löschenden Community

Antwort:

- {`statusCode: 200, statusMessage: Entry with Id [id] was deleted.`}

Fehlerfälle:

- `404 Not Found`: Die Community wurde nicht gefunden.
- `401 Unauthorized`: Der Benutzer ist nicht der Administrator der Community.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.12 GET: `/api/communities/[id]`

Diese Route wird verwendet, um eine Community anhand ihrer `id` abzurufen. Es wird die Prisma-Methode `findUnique` verwendet.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der abzurufenden Community

Antwort:

- Das Community-Objekt mit folgenden Feldern:
  - `id`, `communityName`, `description`, `profileImage`, `backgroundImage`, `bannerImage`, `adminUserId`, `createdAt`, `updatedAt`, `count.posts`, `count.users`

Fehlerfälle:

- `404 Not Found`: Die Community wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.13 GET: `/api/communities`

Diese Route wird verwendet, um eine Liste von Communities abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Community-Objekten mit folgenden Feldern:
  - `id`, `communityName`, `description`, `profileImage`, `backgroundImage`, `bannerImage`, `adminUserId`, `createdAt`, `updatedAt`, `count.posts`, `count.users`

Fehlerfälle:

- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.14 POST: `/api/communities`

Diese Route dient zur Erstellung oder Aktualisierung einer Community. Falls `id` nicht angegeben ist, wird eine neue Community erstellt. Falls `id` vorhanden ist, wird die Community aktualisiert.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `id` (optional, integer): ID der zu aktualisierenden Community
  - `communityName` (erforderlich, string): Name der Community
  - `description` (optional, string): Beschreibung der Community
  - `bannerImageId` (optional, integer): ID des Banner-Bildes
  - `backgroundImageId` (optional, integer): ID des Hintergrundbildes
  - `profileImageId` (optional, integer): ID des Profilbildes

Antwort:

- Das erstellte oder aktualisierte Community-Objekt mit denselben Feldern wie in der GET-Antwort beschrieben.

Fehlerfälle:

- `400 Bad Request`: Fehlende Pflichtfelder wie `communityName`.
- `404 Not Found`: Die Community wurde nicht gefunden.
- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt oder nicht der Administrator der Community.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.15 POST: `/api/communities/search`

Diese Route wird verwendet, um Communities anhand einer Suchanfrage zu durchsuchen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `query` (erforderlich, string): Der Suchbegriff
  - `userId` (optional, integer): ID des Benutzers, dem die Community zugeordnet ist

Antwort:

- Eine Liste von Community-Objekten mit denselben Feldern wie in der GET-Antwort beschrieben.

Fehlerfälle:

- `400 Bad Request`: Die Suchanfrage fehlt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.16 GET: `/api/communities/[id]/feed`

Diese Route wird verwendet, um den Feed einer Community abzurufen, einschließlich der Beiträge in der Community. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Optional.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community
- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Post-Objekten.

Fehlerfälle:

- `404 Not Found`: Die Community wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.17 DELETE: `/api/communities/[id]/join`

Diese Route wird verwendet, um den Benutzer aus einer Community zu entfernen. Der Benutzer muss eingeloggt sein.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community

Antwort:

- Das aktualisierte Community-Objekt.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.18 POST: `/api/communities/[id]/join`

Diese Route wird verwendet, um den Benutzer einer Community hinzuzufügen. Der Benutzer muss eingeloggt sein.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community

Antwort:

- Das aktualisierte Community-Objekt.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.19 GET: `/api/communities/[id]/users`

Diese Route wird verwendet, um eine Liste von Benutzern in der Community abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community
- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Benutzer-Objekten.

Fehlerfälle:

- `404 Not Found`: Die Community wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.20 GET: `/api/images/[id]`

Diese Route wird verwendet, um ein Bild anhand seiner `id` aus der Datenbank zu laden und das zugehörige Bild direkt aus dem Dateisystem zurückzugeben. Nach erfolgreicher Prüfung in der Datenbank wird das Bild aus dem konfigurierten Verzeichnis geladen und als Binärdaten zurückgegeben.

Falls das Bild nicht existiert oder ein Fehler bei der Datenbankabfrage auftritt, wird eine entsprechende Fehlermeldung zurückgegeben.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des abzurufenden Bildes

Antwort:

- Die Binärdaten des Bildes.

Fehlerfälle:

- `404 Not Found`: Das Bild wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.21 POST: `/api/images/[type]`

Diese Route wird verwendet, um ein neues Bild hochzuladen und zu speichern. Unterstützte Bildtypen sind `profile`, `banner`, `post` und `background`, die jeweils unterschiedliche Maße und Speicherpfade haben. Vor der Speicherung wird das Bild auf doppelte Einträge geprüft, um unnötige Mehrfachspeicherungen zu vermeiden.

Das hochgeladene Bild durchläuft folgende Verarbeitungsschritte: Zunächst wird der Bildtyp überprüft. Das Bild wird anschließend in das .webp-Format konvertiert. Um Duplikate zu vermeiden, wird der MD5-Hash der Datei berechnet und geprüft, ob das Bild bereits existiert. Ist das der Fall, wird die zugehörige Bild-ID zurückgegeben. Andernfalls wird das Bild auf die für den jeweiligen Typ definierten Maße skaliert und im Dateisystem gespeichert. Abschließend werden die Bildmetadaten in der Datenbank hinterlegt.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `type` (erforderlich, string): Typ des Bildes. Mögliche Werte:
    - `profile` – 48x48 Pixel (Profilbilder)
    - `banner` – 1500x250 Pixel (Bannerbilder)
    - `post` – 1000x800 Pixel (Bilder für Beiträge)
    - `background` – 1920x1080 Pixel (Hintergrundbilder)
- **Body-Parameter:**
  - Hochgeladene Bilddatei (erforderlich)

Antwort:

- {`id: [id]`} – Die ID des gespeicherten oder bereits vorhandenen Bildes.

Fehlerfälle:

- `400 Bad Request`: Ungültiger Bildtyp oder fehlende Datei.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.
- `500 Failed to process and save image`: Fehler bei der Bildverarbeitung oder Speicherung.

4.22 POST: `/api/logins`

Diese Route wird verwendet, um einen Benutzer einzuloggen. Der Benutzer muss seine E-Mail-Adresse und sein Passwort angeben. Die E-Mail-Adresse wird auf ein gültiges Format geprüft. Falls der Benutzer existiert und das Passwort korrekt ist, wird ein Login-Eintrag in der Datenbank erstellt. Falls bereits ein gültiger Login-Eintrag innerhalb der letzten 24 Stunden existiert, wird dieser wiederverwendet, andernfalls wird ein neuer Eintrag erstellt. Alle älteren Login-Einträge werden gelöscht.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `email` (erforderlich, string): E-Mail-Adresse des Benutzers
  - `password` (erforderlich, string): Passwort des Benutzers

Antwort:

- Benutzerobjekt ohne das Passwortfeld.

Fehlerfälle:

- `400 Bad Request`: Fehlende E-Mail-Adresse oder Passwort.
- `400 Invalid email format`: Die E-Mail-Adresse hat ein ungültiges Format.
- `400 Invalid email`: Kein Benutzer mit dieser E-Mail-Adresse gefunden.
- `400 Invalid email or password`: Das Passwort ist falsch.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.23 DELETE: `/api/logins`

Diese Route wird verwendet, um den aktuellen Benutzer auszuloggen. Der Login-Eintrag wird aus der Datenbank gelöscht, und das `key`-Cookie wird entfernt. Falls kein aktueller Login gefunden wird, wird ein entsprechender Fehler zurückgegeben.

Anmeldung: Anmeldung erforderlich.

Eingabe: Keine Eingabe erforderlich.

Antwort:

- {`statusCode: 200, statusMessage: Üser logged out"`}

Fehlerfälle:

- `400 No current login`: Es wurde kein aktueller Login gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.24 DELETE: `/api/posts/[id]`

Diese Route wird verwendet, um einen Beitrag anhand seiner `id` zu löschen. Es wird überprüft, ob der Benutzer eingeloggt ist und ob er der Ersteller des Beitrags ist. Nach erfolgreicher Prüfung wird der Beitrag aus der Datenbank entfernt.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des zu löschenden Beitrags

Antwort:

- {`statusCode: 200, statusMessage: Ëntry with Id [id] was deleted."`}

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Beitrags fehlt.
- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt oder nicht der Ersteller des Beitrags.
- `404 Post not found`: Der Beitrag wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.25 GET: `/api/posts/[id]`

Diese Route wird verwendet, um einen Beitrag anhand seiner `id` abzurufen. Nach erfolgreicher Prüfung wird das Beitrag-Objekt zurückgegeben.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des abzurufenden Beitrags

Antwort:

- Das Beitrag-Objekt mit Feldern wie `id`, `text`, `title`, `user`, `community`, `count.likes`, `count.comments`, `createdAt`, und `updatedAt`.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Beitrags fehlt.
- `404 Post not found`: Der Beitrag wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.26 GET: `/api/posts`

Diese Route wird verwendet, um eine Liste von Beiträgen abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Beitrag-Objekten mit Feldern wie `id`, `text`, `title`, `user`, `community`, `count.likes`, `count.comments`, `createdAt`, und `updatedAt`.

Fehlerfälle:

- `404 No posts found`: Es wurden keine Beiträge gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.27 POST: `/api/posts`

Diese Route dient zur Erstellung oder Aktualisierung eines Beitrags. Falls keine `id` im Body angegeben ist, wird ein neuer Beitrag erstellt. Falls eine `id` vorhanden ist, wird der bestehende Beitrag aktualisiert.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `id` (optional, integer): ID des zu aktualisierenden Beitrags
  - `title` (optional, string): Titel des Beitrags
  - `text` (optional, string): Text des Beitrags
  - `imageId` (optional, integer): ID des zugehörigen Bildes
  - `communityId` (optional, integer): ID der zugehörigen Community

Antwort:

- Das erstellte oder aktualisierte Beitrag-Objekt mit Feldern wie `id`, `text`, `title`, `user`, `community`, `count.likes`, `count.comments`, `createdAt`, und `updatedAt`.

Fehlerfälle:

- `400 Title and text or imageId are required`: Fehlende Pflichtfelder.
- `400 User is not part of the community`: Der Benutzer gehört nicht zur Community.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.28 POST: `/api/posts/search`

Diese Route wird verwendet, um Beiträge anhand einer Suchanfrage zu durchsuchen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `query` (erforderlich, string): Der Suchbegriff
  - `communityId` (optional, integer): ID der Community
  - `userId` (optional, integer): ID des Benutzers

Antwort:

- Eine Liste von Beitrag-Objekten mit Feldern wie `id`, `text`, `title`, `user`, `community`, `count.likes`, `count.comments`, `createdAt`, und `updatedAt`.

Fehlerfälle:

- `400 Query string in body is missing`: Die Suchanfrage fehlt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.29 GET: `/api/posts/[id]/comments`

Diese Route wird verwendet, um eine Liste der Kommentare zu einem bestimmten Beitrag abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Beitrags
- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Kommentar-Objekten.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Beitrags fehlt.
- `404 PostId not found`: Der Beitrag wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.30 DELETE: `/api/posts/[id]/like`

Diese Route wird verwendet, um ein Like von einem Beitrag zu entfernen. Die Route überprüft, ob der Benutzer eingeloggt ist und ob der Beitrag existiert. Falls erfolgreich, wird das Like entfernt und dem Benutzer wird ein XP-Punkt abgezogen.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Beitrags

Antwort:

- Das aktualisierte Beitrag-Objekt.

Fehlerfälle:

- `400 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Id parameter is missing`: Die ID des Beitrags fehlt.
- `404 PostId not found`: Der Beitrag wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.31 POST: `/api/posts/[id]/like`

Diese Route wird verwendet, um einen Beitrag zu liken. Die Route überprüft, ob der Benutzer eingeloggt ist und ob der Beitrag existiert. Falls erfolgreich, wird dem Beitrag ein Like hinzugefügt und dem Benutzer wird ein XP-Punkt gutgeschrieben.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Beitrags

Antwort:

- Das aktualisierte Beitrag-Objekt.

Fehlerfälle:

- `400 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Id parameter is missing`: Die ID des Beitrags fehlt.
- `404 PostId not found`: Der Beitrag wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.32 GET: `/api/users/feed`

Diese Route wird verwendet, um den Benutzer-Feed abzurufen, der aus Beiträgen besteht, die von den Benutzern und Communities stammen, denen der aktuelle Benutzer folgt. Falls der Benutzer nicht eingeloggt ist, werden allgemeine Beiträge zurückgegeben.

Anmeldung: Optional.

Eingabe:

- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Beitrag-Objekten.

Fehlerfälle:

- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.33 DELETE: `/api/users/[id]`

Diese Route wird verwendet, um einen Benutzer anhand seiner `id` zu löschen. Es wird geprüft, ob der Benutzer eingeloggt ist und ob er die Berechtigung hat, das Konto zu löschen. Falls die Prüfung erfolgreich ist, wird das Benutzerkonto gelöscht.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des zu löschenden Benutzers

Antwort:

- {`statusCode: 200, statusMessage: Ëntry with Id [id] was deleted."`}

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Benutzers fehlt.
- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt oder nicht der Besitzer des Kontos.
- `404 UserId not found`: Der Benutzer wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.34 GET: `/api/users/[id]`

Diese Route wird verwendet, um ein Benutzerprofil anhand seiner `id` abzurufen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des abzurufenden Benutzers

Antwort:

- Das Benutzerprofil mit Feldern wie `id`, `username`, `email`, `bio`, `xp`, `profileImage`, `createdAt`, und `count.posts`.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Benutzers fehlt.
- `404 User not found`: Der Benutzer wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.35 GET: `/api/users`

Diese Route wird verwendet, um eine Liste von Benutzern abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Benutzerprofilen.

Fehlerfälle:

- `404 No users were found`: Es wurden keine Benutzer gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.36 POST: `/api/users`

Diese Route dient zur Erstellung oder Aktualisierung eines Benutzerprofils. Falls keine `id` im Body angegeben ist, wird ein neues Profil erstellt. Falls eine `id` vorhanden ist, wird das bestehende Profil aktualisiert.

Anmeldung: Optional.

Eingabe:

- **Body-Parameter:**
  - `id` (optional, integer): ID des zu aktualisierenden Benutzers
  - `username` (erforderlich, string): Benutzername
  - `email` (erforderlich, string): E-Mail-Adresse
  - `password` (erforderlich, string): Passwort (mindestens 10 Zeichen)
  - `bio` (optional, string): Benutzerbeschreibung
  - `profileImageId` (optional, integer): ID des Profilbildes

Antwort:

- Das erstellte oder aktualisierte Benutzerprofil.

Fehlerfälle:

- `400 Invalid email format`: Die E-Mail-Adresse hat ein ungültiges Format.
- `400 Password must be at least 10 characters long`: Das Passwort ist zu kurz.
- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.37 POST: `/api/users/search`

Diese Route wird verwendet, um Benutzer anhand einer Suchanfrage zu durchsuchen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `query` (erforderlich, string): Suchbegriff für Benutzername, E-Mail oder Beschreibung

Antwort:

- Eine Liste von Benutzerprofilen.

Fehlerfälle:

- `400 Query string in body is missing`: Die Suchanfrage fehlt.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.38 GET: `/api/users/streak`

Diese Route wird verwendet, um den XP-Streak des Benutzers zu aktualisieren und ihm bei erfolgreichem Abschluss einen XP-Punkt gutzuschreiben. Ein Streak kann nur einmal alle 24 Stunden abgeschlossen werden.

Anmeldung: Anmeldung erforderlich.

Eingabe: Keine Eingabe erforderlich.

Antwort:

- Das aktualisierte Benutzerprofil mit dem aktualisierten XP-Wert.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Streak not available - wait 24h`: Der Streak kann erst nach 24 Stunden erneut abgeschlossen werden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.39 GET: `/api/users/[id]/awards`

Diese Route wird verwendet, um die Awards eines Benutzers anhand seiner `id` abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers
- **Query-Parameter:**
  - `page` (optional, integer): Die aktuelle Seite der Paginierung
  - `limit` (optional, integer): Anzahl der zurückzugebenden Einträge pro Seite

Antwort:

- Eine Liste von Award-Objekten.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Benutzers fehlt.
- `404 UserId not found`: Der Benutzer wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.40 GET: `/api/users/[id]/comments`

Diese Route wird verwendet, um die Kommentare eines Benutzers anhand seiner `id` abzurufen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers

Antwort:

- Eine Liste von Kommentar-Objekten.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Benutzers fehlt.
- `404 UserId not found`: Der Benutzer wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.41 GET: `/api/users/[id]/communities`

Diese Route wird verwendet, um die Communities eines Benutzers abzurufen, denen er beigetreten ist.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers

Antwort:

- Eine Liste von Community-Objekten.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Benutzers fehlt.
- `404 UserId not found`: Der Benutzer wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.42 DELETE: `/api/users/[id]/follow`

Diese Route wird verwendet, um einem Benutzer zu entfolgen. Der Benutzer muss eingeloggt sein.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers, dem entfolgt werden soll

Antwort:

- Das aktualisierte Benutzerprofil des Zielbenutzers.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 You cannot unfollow yourself`: Der Benutzer kann sich nicht selbst entfolgen.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.43 POST: `/api/users/[id]/follow`

Diese Route wird verwendet, um einem Benutzer zu folgen. Der Benutzer muss eingeloggt sein.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers, dem gefolgt werden soll

Antwort:

- Das aktualisierte Benutzerprofil des Zielbenutzers.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 You cannot follow yourself`: Der Benutzer kann sich nicht selbst folgen.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.44 GET: `/api/users/[id]/posts`

Diese Route wird verwendet, um die Beiträge eines Benutzers anhand seiner `id` abzurufen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers

Antwort:

- Eine Liste von Beitrag-Objekten.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID des Benutzers fehlt.
- `404 UserId not found`: Der Benutzer wurde nicht gefunden.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

4.45 Datenzugriff und Backend-API Routen

(to be reworked) Die Backend-API (engl. "application programming interface", ist eine Sammlung von Definitionen und Protokollen, die es Softwareanwendungen ermöglichen, miteinander zu kommunizieren (API)) dient als zentrale Schnittstelle zwischen dem Frontend und dem Backend unserer Anwendung. Ihre Hauptaufgabe besteht darin, Daten wie Profile, Communities und Posts aus der Datenbank abzurufen und diese dem Frontend zur Verfügung zu stellen. Die API definiert dabei klar strukturierte Routen, die festlegen, wie diese Datenabfragen erfolgen und welche Informationen abgerufen werden können. Wir verwenden eine RESTful API (Representational State Transfer), eine Art von Web-API, die auf den Prinzipien des REST-Architekturstils basiert. RESTful APIs nutzen HTTP-Anfragen, um auf Ressourcen zuzugreifen und diese zu manipulieren. Diese Ressourcen werden durch eindeutige URLs identifiziert und können in verschiedenen Formaten wie (engl. "JavaScript Object Notation" ist ein kompaktes Datenformat, das in einer leicht lesbaren Textform den Datenaustausch zwischen Anwendungen ermöglicht. (JSON)) oder XML dargestellt werden. Unsere API übergibt JSON Objekte.

Durch die Nutzung der API-Routen wird die Integration ins Frontend erheblich vereinfacht. Entwickler müssen keine direkten SQL-Abfragen schreiben oder tiefgehende technische Kenntnisse besitzen, um auf die benötigten Daten zuzugreifen. Stattdessen können sie die vorgegebenen API-Endpunkte nutzen, um benutzerfreundlich und effizient die gewünschten Informationen zu erhalten und anzuzeigen.

Ein wesentliches Merkmal einer RESTful API ist ihre Statelessness, was bedeutet, dass jede Anfrage vom Client an den Server alle notwendigen Informationen enthält, um sie zu verstehen und zu verarbeiten. Dies erleichtert die Skalierbarkeit und Zuverlässigkeit der API. Darüber hinaus verwenden RESTful APIs standardisierte HTTP-Methoden wie GET, POST, PUT und DELETE, um CRUD-Operationen (Create, Read, Update, Delete) auf den Ressourcen durchzuführen.

(Jeuses chreisis umsortieren und aufdröseln (up))

Systemkontext: Darstellung, wie das Backend in die Gesamtarchitektur eingebunden ist (z. B. Diagramm mit Datenbank, Frontend, API-Gateway). Technologiestack: Beschreibung der eingesetzten Technologien (z. B. Programmiersprache, Frameworks, Datenbank). Designmuster: Falls zutreffend, z. B. REST, Microservices, etc.

Request/Response-Formate: HTTP-Methoden (GET, POST, PUT, DELETE). Beispielanfragen und -antworten (JSON, XML, etc.). Fehlercodes und Fehlermeldungen.

Um unsere API nutzen zu können, verlassen wir uns auf gängige HTTP-Methoden. HTTP-Methoden sind grundlegende Operationen, die im Hypertext Transfer Protocol (HTTP) verwendet werden, um Anfragen zwischen einem Client und einem Server zu definieren.

Die vier häufigsten HTTP-Methoden sind GET, POST, PUT und DELETE. Unsere Routen benutzen diese Methoden, um Daten aus dem Backend ins Frontend zu übertragen oder umgekehrt neue Daten vom Frontend ins Backend weiterzuleiten.

Die GET-Methode wird verwendet, um Daten vom Server anzufordern. Eine GET-Anfrage ruft Informationen ab, ohne den Zustand des Servers zu verändern. Beispielsweise wird eine GET-Anfrage verwendet, um eine Webseite oder eine API-Ressource abzurufen. Mit der POST-Methode werden Daten an den Server gesendet, um eine neue Ressource zu erstellen. Diese Methode wird häufig verwendet, um Formulardaten oder andere Informationen an den Server zu übermitteln, die dann verarbeitet und gespeichert werden. Die PUT-Methode wird verwendet, um eine vorhandene Ressource auf dem Server zu aktualisieren oder zu ersetzen. Im Gegensatz zu POST, das eine neue Ressource erstellt, wenn sie nicht existiert, überschreibt PUT die vorhandene Ressource vollständig mit den gesendeten Daten. Die DELETE-Methode wird verwendet, um eine Ressource auf dem Server zu löschen. Eine DELETE-Anfrage entfernt die angegebene Ressource und verändert somit den Zustand des Servers.

Unsere API Endpunkte können mit diesen Methoden die Daten Serverseitig auf der Datenbank verändern, oder von der Datenbank abrufen. So kann zum Beispiel über die /users Route bestimmte Daten von den Usern über GET abgerufen werden, oder ein neuer User mit POST erstellt werden. Wenn sich jemand zum ersten mal über das Frontend registriert wird der Account über POST in der Datenbank angelegt.

(Notiz: API einbindung in Feed ansicht, endless scrolling: Der Feed lädt alle 10 Posts die nächsten 10 Posts, wenn der Benutzer das Ende der Seite erreicht.

Der API-Endpunkt für den Feed verwendet Prisma, um die Posts aus der Datenbank abzurufen und unterstützt Pagination durch die Parameter page und limit. Wenn also 10 posts auf einer Page sein sollen, wird das limit auf 10 gesetzt. Die parameter werden über url übergeben. Über diesen Endpunkt können also 10 posts geladen werden. Wenn die pagination als parameter seite 2 übergeben bekommt, werden die nächsten 10 posts der Datenbank geladen. Um einen ëndlosen"feed zu generieren, ohne die komplette Datenbank sondern nur 10 einträge auf einmal zu laden wird mit dieser pagination gearbeitet.

im Frontend wird dazu zunächst auf die API route zugegriffen über eine fetch funktion, um die ersten 10 posts zu laden. Anschließend werden)

Endpunkte: Liste der API-Endpunkte (z. B. GET /users, POST /orders). Beschreibung des Zwecks jedes Endpunkts.

Authentifizierung und Autorisierung: Beschreibung des Sicherheitskonzepts (z. B. OAuth, API-Keys). Zugriffsbeschränkungen und Rollen.

5 Abnahmephase

Im Rahmen der Abnahmephase wurde das Projekt aufgrund seiner überschaubaren Größe manuell getestet. Dabei haben wir gezielt Fehleingaben und doppelte Eingaben provoziert, um sicherzustellen, dass die Anwendung auch bei abweichenden Nutzungsszenarien zuverlässig funktioniert. Diese Tests haben bestätigt, dass die Anwendung robust auf verschiedene Eingaben reagiert und Fehler situationsgerecht behandelt.

Hinsichtlich der Stabilität und Skalierbarkeit erfüllt die Applikation die Anforderungen eines kleineren Projekts. Die Verwendung einer relationalen Datenbank sorgt für eine konsistente Datenhaltung, wobei diese zwingend auf einem Server betrieben werden muss. Die API hingegen kann problemlos auf mehreren Instanzen bereitgestellt werden, um Skalierung zu ermöglichen.

Das Frontend arbeitet vollständig ohne serverseitiges Rendering (SSR) und wird ausschließlich clientseitig gerendert. Dadurch lässt sich das Frontend effizient über ein CDN cachen, was die Ladezeiten optimiert und die Auslieferung der Inhalte beschleunigt. Diese Architektur ermöglicht eine flexible und performante Bereitstellung der Anwendung.

5.1 Bereitstellung

Die Bereitstellung unserer Anwendung erfolgt auf einem Debian 12 Linux-Server beim deutschen Hostinganbieter Hetzner. Um sicherzustellen, dass stets die neueste Version von Node.js verwendet wird, binden wir das offizielle NodeSource APT-Repository ein und beziehen Node.js direkt daraus. Nach der Installation werden die Projektdateien auf den Server übertragen und entsprechend konfiguriert. Der Zugriff auf den Server erfolgt ausschließlich über SSH, abgesichert durch einen SSH-Schlüssel. Dies gewährleistet eine verschlüsselte Verbindung sowohl für die Verwaltung des Servers als auch für die Übertragung der Dateien mittels SFTP.

Für den Betrieb unserer Backend-API nutzen wir einen Node.js-Prozess, der über systemd als Dienst gestartet und verwaltet wird. Systemd sorgt für eine zuverlässige Prozessverwaltung, überwacht den Status und startet den Prozess bei Bedarf automatisch neu. Zur weiteren Absicherung wird der Zugriff auf systemkritische Dateien und Ressourcen durch das Linux-Kernel-Feature "Control Groups"(cgroups) eingeschränkt, wodurch eine präzise Ressourcenverwaltung ermöglicht wird.

Die Kommunikation von außen erfolgt ausschließlich über TLS 1.2 und TLS 1.3, wobei nur sichere Chiffren zugelassen werden. Für den Zugriff wird Nginx als Reverse Proxy eingesetzt, der Anfragen an die Node.js-API weiterleitet. Gleichzeitig sorgt Nginx dafür, dass alle HTTP-Anfragen automatisch auf HTTPS umgeleitet werden. Die SSL-Zertifikate

werden von ACME Certbot bereitgestellt und regelmäßig über einen systemd-timer automatisch verlängert.

Netzwerkseitig wird der Server zusätzlich durch iptables abgesichert. Es sind nur Verbindungen zu den Ports 80 (HTTP) und 443 (HTTPS) erlaubt. Ein denkbarer weiterer Schutz wäre die Konfiguration von iptables mit –limit, um eingehende Anfragen zu begrenzen und damit einfachen DDoS-Angriffen vorzubeugen.

Das Debian-System ist so konfiguriert, dass Sicherheitsupdates automatisch eingespielt werden, einschließlich Aktualisierungen für Node.js. Unser Projekt muss jedoch manuell aktualisiert werden. Die neuen Versionen werden über SFTP hochgeladen und anschließend aktiviert. Dabei kann es zu kurzen Downtimes kommen. Für eine zukünftige Optimierung könnte der Update-Prozess automatisiert oder ein Zero-Downtime-Deployment in Betracht gezogen werden.

5.2 Fazit

Die Entwicklung dieser Anwendung war für uns als unerfahrene Entwickler eine spannende und zugleich herausfordernde Erfahrung. Besonders anspruchsvoll war die Zusammenarbeit in einem Team von sechs Personen, da dies für uns alle das erste Mal in dieser Konstellation war. Unsere ursprüngliche Entwicklungsstrategie sah vor, dass das Frontend und das Backend zeitgleich von derselben Person entwickelt werden, um eine voneinander abweichende Entwicklung zu vermeiden und mögliche Integrationsprobleme zu minimieren.

Diese Strategie stellte sich jedoch schnell als unpraktikabel heraus. Der technische Aufwand und die Einarbeitung in die verschiedenen Technologien waren deutlich komplexer, als wir zunächst angenommen hatten. Daher entschieden wir uns frühzeitig, unser Team in spezialisierte Backend- und Frontend-Entwickler aufzuteilen. Dies erwies sich als kluge Entscheidung, da es uns ermöglichte, die jeweiligen domänenspezifischen Fähigkeiten gezielt einzusetzen. Die Spezialisierung steigerte nicht nur die Codequalität, sondern auch die Effizienz unseres Entwicklungsprozesses.

Eine weitere Herausforderung, die sich durch unsere begrenzte Erfahrung ergab, war die Zeitplanung. Viele Features benötigten in der Umsetzung mehr Zeit als erwartet, was zu Verzögerungen führte. Um dennoch den Überblick zu behalten und das Projekt erfolgreich abzuschließen, haben wir sogenannte Cold Freeze- und Hard Freeze-Meilensteine definiert. Diese Vorgehensweise half uns, die Prioritäten klar zu setzen und uns rechtzeitig auf die wesentliche Kernfunktionalität zu konzentrieren. Ohne diese strikte Fokussierung wäre es kaum möglich gewesen, das Projekt innerhalb des vorgegebenen Zeitrahmens abzuschließen.

Technisch betrachtet haben wir wichtige Erkenntnisse gewonnen, die wir bei zukünftigen Projekten berücksichtigen werden. Wir würden uns beim nächsten Mal klar dafür entscheiden, Frontend und Backend getrennt voneinander zu entwickeln und dafür unterschiedliche Technologien einzusetzen. Für das Backend käme beispielsweise Go in Frage, während das Frontend auf spezialisierte Bibliotheken wie Vue oder SolidJS setzen könnte. Diese Trennung hätte mehrere Vorteile:

- Sie würde eine höhere Performance und bessere Skalierbarkeit ermöglichen.
- Framework-spezifische Einschränkungen, wie sie bei Nuxt auftraten, könnten vermieden werden.
- Die Reduzierung von Abhängigkeiten (Node-Module) würde die Stabilität und Wartbarkeit des Projekts erheblich verbessern.

Ein weiterer Punkt, den wir gelernt haben, betrifft die Verwendung von REST-APIs. Während REST einfach zu implementieren ist, zeigte sich, dass es in Bezug auf Underfetching und Overfetching problematisch sein kann, was die Performance unserer Anwendung deutlich beeinträchtigt. Eine sinnvolle Alternative für zukünftige Projekte wäre der Einsatz von GraphQL, da es die Abfrage von genau den benötigten Daten ermöglicht. Dies würde nicht nur die Datenmenge optimieren, sondern auch die Performance der Anwendung erheblich verbessern.

Ein großer Erfolg war für uns die Einführung eines ORM (Object-Relational Mapping). Es erleichterte die Arbeit mit der Datenbank und erhöhte die Produktivität sowie die Sicherheit erheblich. Durch ORM konnten wir Datenbankabfragen auf einer abstrakteren Ebene formulieren, was potenzielle Sicherheitsrisiken wie SQL-Injections minimierte und die Entwicklungszeit verkürzte.

Aus funktionaler Sicht bietet die aktuelle Version unserer Anwendung bereits eine solide Basis, doch die Möglichkeiten für Erweiterungen sind nahezu unbegrenzt. Zukünftig könnten wir die Anwendung beispielsweise um ein Chat-System erweitern, das die Interaktion der Benutzer fördert. Ebenso ließen sich mehr RPG-Elemente wie ein Skill-Tree einfügen, der den Benutzern individuelle Fortschrittsmöglichkeiten bietet und die Motivation zur aktiven Teilnahme steigert.

Abschließend lässt sich sagen, dass dieses Projekt trotz aller Herausforderungen ein voller Erfolg war. Wir haben nicht nur technisches Wissen hinzugewonnen, sondern auch gelernt, effizienter im Team zu arbeiten, Zeitpläne zu hinterfragen und komplexe Zusammenhänge in der Webentwicklung zu verstehen. Das Projekt war ein wichtiger Meilenstein für uns und bildet eine solide Grundlage für zukünftige Vorhaben. Unsere Erfahrungen werden uns helfen, in kommenden Projekten bessere Entscheidungen zu treffen und uns als Entwickler weiterzuentwickeln.

Literatur

[1] B. Institut, „Rechenzentren in Deutschland: Eine Studie zu Energiebedarf und CO2-Emissionen,“ Borderstep Institut für Innovation und Nachhaltigkeit, 2020, Zugriff am 05. Februar 2025.