Siemens Energy - BETI 2024

Realisierung eines Webauftritts

Socialmedia RPG

Echo

Ogulcan Kuecuek

Leon Woenckhaus

Nick Hildebrandt

Aaron Turyabahika

Andre Seiler

27. Februar 2025

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CRUD engl. ”Create, Read, Update, Delete”, die Funktionen, die ein Nutzer benötigt, um Daten anzulegen und zu verwalten. 3

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

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

SSR engl. ”Server Side Rendering”, ist Technik, Webseiten auf dem Server zu rendern, bevor sie an den Client-Browser gesendet werden. 3

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

Für die Umsetzung von Echo war eine detaillierte Ressourcenplanung erforderlich, um eine effiziente und produktive Arbeitsumgebung zu gewährleisten. Dabei wurden sowohl technische als auch personelle Ressourcen berücksichtigt, um sicherzustellen, dass die Entwicklung des Projekts ohne Verzögerungen oder Engpässe erfolgen konnte.

Ein zentraler Aspekt war die Bereitstellung der notwendigen Hardware und Software. Da das Projekt vollständig digital entwickelt wird, wurde für jedes Teammitglied ein voll ausgestatteter Arbeitsplatz eingerichtet. Dies umfasste jeweils einen Laptop mit Docking-Station, um eine flexible Nutzung zu ermöglichen, sowie zusätzliche Bildschirme, um die

Produktivität während der Entwicklung zu steigern. Zur optimalen Bedienbarkeit wurden zudem Maus und Tastatur bereitgestellt. Insgesamt wurden fünf Arbeitsplätze für das Kernteam eingerichtet, um eine reibungslose Zusammenarbeit zu gewährleisten.

Für die physische Infrastruktur wurde am ersten Tag ein eigener Arbeitsraum gemäß DGUV Vorschrift 3 (DGUV V3) eingerichtet, um eine sichere und ergonomische Arbeitsumgebung zu gewährleisten. Dies beinhaltete die Einrichtung von Stromanschlüssen, ergonomischen Sitzplätzen und einer strukturierten Arbeitsorganisation, um sowohl den gesundheitlichen Anforderungen als auch den technischen Anforderungen gerecht zu werden. Dieser separate Raum erlaubte es dem Team, ungestört an der Umsetzung von Echo zu arbeiten und eine konzentrierte Entwicklungsatmosphäre zu schaffen.

Neben der internen Hardware- und Infrastrukturplanung waren auch externe personelle Ressourcen erforderlich, um die Umsetzung des Projekts effizient voranzutreiben. Insbesondere für spezifische Backend- und Datenbankentwicklungen wurde der externe Entwickler Friedrich Waag engagiert, der als freiberuflicher Berater mit einer Vergütung von 100 Euro pro Stunde tätig war. Seine Expertise trug maßgeblich zur Optimierung der Backend-Architektur und der REST-API-Integration bei, sodass eine skalierbare und performante Infrastruktur gewährleistet werden konnte.

Zusätzlich zu den offensichtlichen Ressourcen wie Hardware, Software und Arbeitsraum wurde auch auf die Bereitstellung essenzieller Tools und Entwicklungsumgebungen geachtet. Die Softwareausstattung umfasste eine Kombination aus Nuxt.js, Prisma, Tailwind CSS und Git für die Versionskontrolle, um eine effiziente und gut dokumentierte Entwicklung sicherzustellen. Die Nutzung einer cloudbasierten Entwicklungsumgebung erleichterte die Zusammenarbeit im Team und die Nachverfolgbarkeit von Änderungen.

2.3 Kostenplanung

Die folgende Tabelle gibt einen Überblick über die geschätzten Gesamtkosten für die Entwicklung der Web-App Echo über einen Zeitraum von sechs Wochen. Die Kosten setzen sich aus Personalkosten, Bürokosten sowie der benötigten Hardware für die Arbeitsplätze zusammen. Während die Personalkosten die größten Ausgaben darstellen, wurden auch die infrastrukturellen Anforderungen für eine produktive Entwicklungsumgebung berücksichtigt.

Mit der detaillierten Aufstellung der Kosten ist die Planung abgeschlossen. Alle wesentlichen Ausgaben wurden berücksichtigt, sodass eine realistische Einschätzung der finanziellen Anforderungen vorliegt. Damit steht einer effizienten Umsetzung der Web-App Echo aus wirtschaftlicher Sicht nichts mehr im Weg.

Ressource	**Kosten (€)**	**Gesamtkosten (€)**
Personalkosten		
Entwickler (5x6 Wochen, 40 Std./Woche, 30 €/Std.)	30	36.000
Externer Entwickler (5 Std. á 100 €)	100	500
Bürokosten		
Büromiete (30x100 m², 1,5 Monate)	30/m²	4.500
Hardware für 5 Arbeitsplätze		
Höhenverstellbarer Schreibtisch	400	2.000
Ergonomischer Bürostuhl	300	1.500
Laptop (Entwicklungsgerät)	1.200	6.000
Monitor (24 Zoll)	150	750
Dockingstation	150	750
Tastatur	50	250
Maus	30	150
Gesamtkosten		**51.400**

Tabelle 1: Gesamtkosten für die Entwicklung der Web-App Echo

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
13.01.2025 - 15.01.2025	Detaillierte Planung des Projekts und des Projektumfangs
16.01.2025 - 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 - 20.02.2025	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
21.02.2025	Cold Freeze: Keine neuen Features werden hinzugefügt, Fokus auf Stabilisierung und Fehlerbehebung
23.02.2025	Hard Freeze: Endgültiger Stopp aller Änderungen am Code
25.02.2025	Präsentation des Projekts

Tabelle 2: Zeitplan der Projektentwicklung

3 Zielplattform und Implementierung

Die Plattform Echo wurde als moderne Web-App konzipiert, um eine möglichst breite Nutzerbasis zu erreichen und eine barrierefreie Nutzung zu ermöglichen. Eine Web-App hat den entscheidenden Vorteil, dass sie direkt über den Webbrowser genutzt werden kann, ohne dass eine separate Installation oder ein spezifisches Betriebssystem erforderlich ist. Dies reduziert die Einstiegshürde für neue Nutzer erheblich, da sie lediglich einen Browser öffnen und sich anmelden müssen, um auf die Plattform zuzugreifen. Gleichzeitig ermöglicht dieses Konzept eine plattformspezifische Unabhängigkeit, sodass Nutzer mit Windows, macOS, Linux oder mobilen Betriebssystemen wie Android und iOS gleichermaßen auf Echo zugreifen können.

Während viele Social-Media-Plattformen als native Apps entwickelt werden, bedeutet dies oft einen hohen Entwicklungsaufwand, da für verschiedene Plattformen wie iOS und Android jeweils eigenständige Anwendungen erstellt und gepflegt werden müssen. Mit einer Web-App als Ausgangspunkt wird dieser Aufwand minimiert, da Änderungen und neue Funktionen zentral implementiert und sofort für alle Nutzer verfügbar gemacht werden können. Zudem profitieren Nutzer von einer konsistenten Benutzererfahrung, unabhängig davon, welches Gerät sie verwenden. Dadurch bleibt Echo nicht nur technisch schlank, sondern auch für Entwickler und Administratoren leicht wartbar.

Langfristig ist jedoch geplant, die Plattform um eine mobile App zu erweitern, um eine noch tiefere Integration in mobile Betriebssysteme zu ermöglichen. Dafür wurde von Anfang an eine REST-API als Kommunikationsschnittstelle entwickelt, die die Grundlage für eine spätere native App bilden kann. Die API ermöglicht es, dass sowohl die Web-App als auch zukünftige mobile Anwendungen mit derselben Datenquelle arbeiten, was eine flexible Weiterentwicklung erlaubt. Statt zwei völlig getrennte Anwendungen zu verwalten, kann die Web-App als primäre Plattform fungieren, während mobile Anwendungen als alternative Clients auf dieselben Funktionen und Daten zugreifen.

Diese Strategie ermöglicht es, Echo zunächst mit minimalem Entwicklungsaufwand einer großen Nutzerschaft bereitzustellen und gleichzeitig eine skalierbare Infrastruktur für zukünftige Erweiterungen zu schaffen. Indem die Plattform von Anfang an als Web-App konzipiert wurde, kann sie schneller wachsen, einfacher gewartet werden und bleibt für alle Nutzer unabhängig vom verwendeten Endgerät jederzeit zugänglich.

3.1 Architekturdesign

Die Architektur von Echo basiert auf einem modernen Single-Page Application (SPA)-Ansatz, bei dem die gesamte Anwendung innerhalb eines einzigen HTML-Dokuments läuft und der Inhalt dynamisch durch JavaScript nachgeladen wird. Anders als bei Server-

Side Rendering (SSR), bei dem jede Seite auf dem Server generiert und an den Client ausgeliefert wird, lädt eine SPA die benötigten Daten asynchron über eine API, sodass die Nutzererfahrung deutlich flüssiger ist. Dies bedeutet, dass Seitenübergänge innerhalb der Anwendung ohne vollständiges Neuladen des Browsers erfolgen, was zu einer schnelleren Reaktionszeit und einer optimierten Performance führt.

Ein wesentlicher Vorteil dieser Architektur ist ihre hohe Skalierbarkeit. Da die Server nicht für jede Seitenanfrage eine vollständig gerenderte HTML-Seite liefern müssen, reduziert sich die Last auf den Server erheblich. Stattdessen werden nur die notwendigen JSON-Daten über eine API geladen, was nicht nur Bandbreite spart, sondern auch für Nutzer mit langsamen oder instabilen Internetverbindungen von Vorteil ist. Anstatt große Mengen an HTML und CSS erneut zu übertragen, kann die Web-App gezielt nur die benötigten Datenfragmente (Snippets) nachladen, wodurch die Ladezeiten minimiert und Inhalte selbst bei schlechter Verbindung progressiv aufgebaut werden können.

Die Umsetzung von Echo erfolgt mit Nuxt, einem leistungsstarken Vue.js-Framework, das ursprünglich für SSR (Server-Side Rendering) optimiert wurde, aber auch perfekt als reines SPA-Framework genutzt werden kann. In unserem Fall wird auf SSR verzichtet, da die Anwendung als dynamische Web-App funktioniert und die Inhalte in Echtzeit über eine REST-API nachlädt. Nuxt ermöglicht eine modulare Entwicklung, in der Komponenten klar strukturiert sind und eine optimierte Code-Splitting-Strategie verwendet wird, um unnötiges Laden von nicht benötigten Skripten zu vermeiden.

Für die Verwaltung der Datenbank setzen wir auf Prisma, ein modernes ORM (Object-Relational Mapping), das als Schnittstelle zwischen der Anwendung und der relationalen SQL-Datenbank fungiert. Prisma generiert automatisch eine typsichere API, die es ermöglicht, Datenbankabfragen direkt in TypeScript zu schreiben, ohne rohe SQL-Statements nutzen zu müssen. Dadurch wird nicht nur die Lesbarkeit und Wartbarkeit des Codes verbessert, sondern auch Sicherheitsrisiken wie SQL-Injections verhindert. Prisma kümmert sich außerdem um Datenbankmigrationen, sodass Änderungen an der Datenstruktur ohne aufwendige manuelle Anpassungen umgesetzt werden können.

Als Frontend-Framework kommt Vue.js zum Einsatz, das eine komponentenbasierte Architektur bietet und durch seine reaktive Datenbindung eine besonders dynamische Benutzererfahrung ermöglicht. Vue.js erlaubt es, die Benutzeroberfläche in kleine, wiederverwendbare Komponenten aufzuteilen, die unabhängig voneinander geladen und aktualisiert werden können. Dies passt perfekt zu einer SPA-Architektur, da der gesamte Inhalt der Anwendung effizient verwaltet und nur das nachgeladen wird, was tatsächlich benötigt wird.

Unter der Haube verwendet Nuxt für die Serverlogik H3, einen leichtgewichtigen HTTP-Server, der speziell für serverlose Umgebungen und Edge-Computing optimiert wurde.

H3 wird genutzt, um REST-API-Endpunkte innerhalb der Anwendung bereitzustellen, wodurch Daten zwischen dem Frontend und der Datenbank effizient ausgetauscht werden können. Da H3 direkt in Nuxt integriert ist, kann es nahtlos mit den vorhandenen Vue-Komponenten und Prisma-Datenbanken interagieren, ohne dass zusätzliche externe Server benötigt werden.

Die Benutzeroberfläche von Echo wurde mit einem klaren Fokus auf Modernität und Performance entwickelt. Um eine ansprechende, reaktionsschnelle und flexible Gestaltung zu gewährleisten, setzen wir auf Nuxt UI und Tailwind CSS, zwei leistungsstarke Technologien, die speziell für moderne Web-Apps optimiert sind. Diese Kombination erlaubt es uns, eine ästhetisch ansprechende und zugleich hochfunktionale Oberfläche zu erstellen, die sich dynamisch an verschiedene Bildschirmgrößen und Endgeräte anpasst.

Nuxt UI ist eine komponentenbasierte UI-Bibliothek, die speziell für Nuxt entwickelt wurde und zahlreiche fertige, hochoptimierte UI-Elemente bietet. Dies ermöglicht es uns, schnell einheitliche und benutzerfreundliche Komponenten zu erstellen, die sich perfekt in das Vue/Nuxt-Ökosystem integrieren. Durch die Nutzung von Nuxt UI entfällt die Notwendigkeit, grundlegende UI-Komponenten wie Buttons, Modale, Formulare oder Karten von Grund auf neu zu schreiben, was die Entwicklungszeit erheblich reduziert und gleichzeitig eine konsistente Nutzererfahrung gewährleistet.

Zusätzlich setzen wir auf Tailwind CSS, ein utility-first CSS-Framework, das es ermöglicht, hochgradig anpassbare und performante Designs zu erstellen. Anstatt klassische Stylesheets mit vielen vordefinierten CSS-Klassen zu schreiben, können wir mit Tailwind direkt im Markup flexible und skalierbare Stile definieren, die sich dynamisch an die Anforderungen der Anwendung anpassen. Dies reduziert den Overhead von ungenutztem CSS, verbessert die Ladezeiten und erleichtert das Styling von Komponenten erheblich. Ein weiterer Vorteil von Tailwind ist die eingebaute Responsivität, wodurch sich die Benutzeroberfläche automatisch an verschiedene Bildschirmgrößen anpasst und sowohl auf Desktops, Tablets als auch Smartphones optimal dargestellt wird.

Durch die Kombination aus Nuxt UI und Tailwind CSS bietet Echo eine moderne, minimalistische und zugleich intuitive Benutzeroberfläche, die nicht nur schnell lädt, sondern auch leicht anpassbar und erweiterbar ist. Besonders in Verbindung mit der SPA-Architektur sorgt dies für eine flüssige Nutzererfahrung, da UI-Komponenten nicht ständig neu geladen, sondern dynamisch aktualisiert werden. Insgesamt ermöglicht diese Technologieauswahl eine konsistente, ästhetische und performante Oberfläche, die sowohl für Erstnutzer als auch erfahrene Anwender einfach und intuitiv bedienbar bleibt.

3.2 Datenmodell

Eine Datenbank ist das Rückgrat einer jeden Social-Media-Plattform, da sie sämtliche Informationen über Benutzer, Beiträge, Kommentare, Communities und Interaktionen verwaltet. Ohne eine gut strukturierte Datenbank wäre es unmöglich, Inhalte effizient abzurufen, Nutzeraktivitäten nachzuverfolgen oder die Plattform langfristig skalierbar zu halten. Besonders in sozialen Netzwerken entstehen durch die zahlreichen Interaktionen zwischen den Nutzern viele Relationen, also Verknüpfungen zwischen unterschiedlichen Datensätzen. Ein Benutzer kann beispielsweise mehrere Beiträge veröffentlichen, Kommentare hinterlassen, Likes verteilen oder sich mit Communities verbinden. Diese komplexen Verbindungen stellen hohe Anforderungen an die Datenbankarchitektur, da sie sowohl leistungsfähig als auch konsistent sein muss.

Eine relationale SQL-Datenbank eignet sich besonders gut für ein solches System, da sie Daten in klar definierten Tabellen organisiert und über Fremdschlüssel (Foreign Keys) strukturierte Beziehungen zwischen den einzelnen Einträgen ermöglicht. Dies stellt sicher, dass jeder Benutzer eindeutig identifizierbar ist, Beiträge mit ihren jeweiligen Autoren verknüpft sind und Kommentare den entsprechenden Posts zugeordnet werden. Ein weiterer Vorteil von SQL ist die Möglichkeit, über mächtige JOIN-Abfragen mehrere Tabellen effizient zu durchsuchen, ohne redundante Daten zu speichern. Die Wahl einer relationalen Datenbank für Echo erlaubt es uns, Daten strukturiert, performant und sicher zu verwalten, was insbesondere bei wachsender Nutzerzahl entscheidend ist.

Eine der größten Herausforderungen bei der Entwicklung einer relationalen Datenbank für eine Social-Media-Plattform besteht darin, die Vielzahl an Beziehungen effizient zu handhaben. Eine falsche Modellierung kann schnell zu langsamen Abfragen, inkonsistenten Daten oder doppelten Einträgen führen. Gleichzeitig muss sichergestellt werden, dass Benutzeraktionen wie das Löschen eines Kontos oder das Entfernen eines Beitrags nicht zu verwaisten Datensätzen führen. Ein weiteres Problem ist die Sicherheit, da direkter Zugriff auf die Datenbank potenziell anfällig für Angriffe wie SQL-Injections ist. Um diese Risiken zu minimieren und gleichzeitig eine flexible, leicht erweiterbare Struktur zu gewährleisten, haben wir uns für den Einsatz eines Object-Relational Mappers (ORM) entschieden.

Ein ORM ist eine Abstraktionsschicht, die den direkten Umgang mit SQL Befehlen überflüssig macht und stattdessen Datenbankoperationen über eine objektorientierte API ermöglicht. Dies verbessert nicht nur die Lesbarkeit und Wartbarkeit des Codes, sondern bietet auch integrierte Sicherheitsmechanismen, die automatisch SQL-Injections verhindern. Für Echo verwenden wir Prisma, ein modernes ORM für TypeScript und Node.js, das sich durch seine typsichere API, einfache Migrationsverwaltung und starke Performance auszeichnet. Prisma erlaubt es uns, unser Datenbankschema in einer klar definier-

ten Schema-Datei zu beschreiben, aus der automatisch SQL-Strukturen generiert werden. Dies erleichtert nicht nur die Verwaltung der Datenbank, sondern sorgt auch für eine konsistente Entwicklung über alle Teammitglieder hinweg.

Das Datenbankschema von Echo umfasst mehrere zentrale Tabellen, die durch Relationen miteinander verknüpft sind. Die wichtigste Entität ist die Benutzer (User)-Tabelle, in der grundlegende Informationen wie id, username, email und password gespeichert sind. Zusätzlich gibt es optionale Felder für profileImage, backgroundImage und bannerImage, die auf eine separate Image-Tabelle verweisen. Ein Benutzer sammelt Erfahrungspunkte (xp) und kann verschiedene Auszeichnungen (Award) erhalten, die in einer @relation(ÜserAwards")-Tabelle verwaltet werden.

Beiträge (Post) sind eine weitere zentrale Entität in der Datenbank. Jeder Post ist über den userId-Fremdschlüssel mit einem Benutzer verknüpft und enthält Informationen wie title, text, imageId und createdAt. Beiträge können in einer Community (Community) veröffentlicht werden, die ebenfalls eine eigene Tabelle mit communityName, description und adminUserId besitzt. Jeder Post kann wiederum mehrere Kommentare (Comment) enthalten, die mit postId dem jeweiligen Beitrag zugewiesen sind. Um sicherzustellen, dass alle Relationen konsistent bleiben, nutzen wir für viele Tabellen onDelete: Cascade, wodurch beispielsweise beim Löschen eines Posts auch automatisch alle zugehörigen Kommentare entfernt werden.

Zusätzlich zur Speicherung der Beiträge und Kommentare enthält die Datenbank Mechanismen zur Verwaltung von Benutzerinteraktionen. Ein Nutzer kann Beiträge und Kommentare liken, wodurch eine Like-Tabelle benötigt wird, die userId und postId oder commentId miteinander verbindet. Außerdem haben wir ein Community Mitgliedschaftssystem, das eine @relation zwischen User und Community verwaltet, sodass ein Nutzer Mitglied in mehreren Gruppen sein kann.

Ein besonderes Feature von Echo ist das Award-System, mit dem Nutzer Auszeichnungen für besondere Leistungen erhalten können. Hierzu gibt es eine eigene Award-Tabelle, die sowohl mit Benutzern als auch mit Communities verknüpft sein kann. Jeder Award hat ein awardName, ein awardImageId und kann von einem Nutzer oder Administrator verliehen werden. Durch diese Art der Gamification wird das Engagement der Nutzer gefördert und langfristige Interaktionen innerhalb der Plattform unterstützt.

4 Benutzeroberfläche

Die Gestaltung der Benutzeroberfläche von Echo basiert auf der Analyse und Kombination bewährter Konzepte aus bestehenden Social-Media-Plattformen, um eine intuitive und ansprechende Nutzererfahrung zu schaffen. Inspirieren ließen wir uns dabei insbesondere von Plattformen wie Reddit, Discord, Instagram, Bluesky und Steam. Jede dieser Plattformen besitzt bestimmte Eigenschaften, die für unser Design relevant waren, aber auch Schwachstellen, die wir verbessern wollten.

Von Reddit haben wir das Konzept der Communities übernommen, welches es Nutzern ermöglicht, themenbezogene Gruppen zu erstellen, sich zu vernetzen und Inhalte auszutauschen. Allerdings ist Reddit als klassische Forenstruktur eher unübersichtlich, weshalb wir eine modernere Umsetzung gewählt haben. Discord bietet mit seinen Servern eine ähnliche Funktion, allerdings ist der soziale Aspekt nicht sofort ersichtlich, da die Plattform primär als Chat-Dienst konzipiert ist. Instagram hingegen zeichnet sich durch eine minimalistische und intuitive Benutzerführung aus, bei der zentrale Funktionen wie der Feed, die Suche und das Profil über eine feste Leiste erreichbar sind. Diese klare Menüstruktur war eine wesentliche Inspiration für die Umsetzung unserer eigenen Navigation. Zusätzlich haben wir aus Steam die Idee der Badges und Belohnungssysteme übernommen, um den Gamification-Aspekt unserer Plattform hervorzuheben.

Die Benutzeroberfläche von Echo ist in mehrere klar strukturierte Bereiche unterteilt. Eine stets präsente Navigationsleiste ermöglicht den schnellen Zugriff auf zentrale Funktionen. Der Hauptinhalt der Seite konzentriert sich auf den dynamischen Post-Feed, der durch Infinite Scrolling eine flüssige Nutzungserfahrung bietet. Icons statt Text machen die Navigation und Interaktion intuitiver, da Nutzer gängige Symbole schneller erfassen können als Textbeschriftungen. So wird beispielsweise das Einstellungen-Symbol als Zahnrad dargestellt, während ein Plus-Symbol zum Erstellen neuer Beiträge verwendet wird. Die Farbgebung und visuelle Gestaltung basieren auf modernen Usability-Richtlinien, wobei Nuxt UI-Elemente anstelle individueller CSS-Lösungen genutzt werden, um Konsistenz und eine optimierte User Experience zu gewährleisten.

Ein zentrales Element der Benutzerführung ist die NavigationBar, die dauerhaft am oberen oder seitlichen Bildschirmrand sichtbar bleibt. Sie enthält die wichtigsten Menüpunkte, darunter Feed, Communities und Suche. Je nach Login-Status unterscheidet sich der angezeigte Inhalt: Eingeloggte Nutzer erhalten Zugriff auf ihr Profil, während Gäste einen Anmelden-Button sehen. Die Icons sind mit line-md- und heroicons-Symbolen umgesetzt, um eine moderne und konsistente Darstellung zu gewährleisten.

Die Navigation orientiert sich an den besten Praktiken moderner Social-Media-Apps, insbesondere an Instagram und Bluesky, wo eine reduzierte und leicht verständliche Struktur

die Interaktion mit der Plattform erleichtert. Da die Communities eine zentrale Rolle in Echo spielen, haben wir diese prominent in der Navigation platziert, sodass Nutzer schnell neue Gruppen entdecken und ihnen beitreten können. Zudem ist die Suchfunktion fest integriert, um Inhalte und Nutzer jederzeit auffindbar zu machen.

4.1 Login

Die Login-Seite von Echo bildet den zentralen Zugangspunkt für registrierte Nutzer und ermöglicht eine sichere und reibungslose Authentifizierung. Sie wurde so gestaltet, dass sie eine intuitive Bedienung, eine effiziente Validierung von Eingaben und eine schnelle Anmeldung ermöglicht, während gleichzeitig moderne Sicherheitsstandards eingehalten werden.

Beim Laden der Seite wird dem Nutzer ein einfaches, aber funktionales Anmeldeformular präsentiert. Dieses besteht aus zwei Haupteingabefeldern: E-Mail-Adresse und Passwort. Der Nutzer gibt seine Anmeldeinformationen ein, und nach Absenden des Formulars (onSubmit()) werden die Daten an die API /api/logins gesendet, um die Authentifizierung durchzuführen.

Die Eingabefelder sind mit einer Validierung versehen, um Fehleingaben direkt abzufangen. Folgende Regeln werden überprüft:

1. Die E-Mail-Adresse darf nicht leer sein und muss einem gültigen Format entsprechen
2. Das Passwort muss mindestens 10 Zeichen lang sein
3. Falls ein Feld ungültig ist, erhält der Nutzer eine sofortige visuelle Rückmeldung

Sobald der Nutzer auf „Anmelden“ klickt, wird die Anfrage an den Server geschickt. Während der Anfrage wird der submitting-State auf true gesetzt, um Mehrfachklicks zu verhindern. Sollte die Anmeldung erfolgreich sein, werden die Benutzerdaten über loadMe() geladen, und der Nutzer wird zur Startseite oder einer zuvor aufgerufenen Seite weitergeleitet. Falls ein Fehler auftritt – etwa durch falsche Zugangsdaten oder einen Serverfehler – wird eine entsprechende Fehlermeldung über useToast() angezeigt.

Die Oberfläche folgt einem klaren und minimalistischen Design, das sich nahtlos in das Gesamtbild von Echo einfügt. Die Login-Maske ist zentral ausgerichtet und verwendet Nuxt UI-Komponenten für eine moderne, ansprechende Darstellung. Eine zusätzliche Passwort-Vergessen-Funktion oder die Möglichkeit zur Neuregistrierung könnte durch ergänzende Buttons bereitgestellt werden, um die Nutzerfreundlichkeit weiter zu verbessern.

4.2 Registrierung

Die Registrierungsseite in Echo bietet neuen Nutzern die Möglichkeit, sich einfach und sicher auf der Plattform zu registrieren. Dabei wird ein klar strukturiertes und benutzerfreundliches Anmeldeformular bereitgestellt, das eine schnelle und fehlerfreie Eingabe der benötigten Informationen gewährleistet.

Beim Betreten der Seite wird dem Nutzer ein drei Felder umfassendes Formular präsentiert, in das er folgende Informationen eingeben muss:

1. Benutzername – Darf nicht leer sein und maximal 15 Zeichen enthalten
2. E-Mail-Adresse – Sie muss ein gültiges Format haben
3. Passwort – Es muss mindestens 10 Zeichen lang sein

Alle Eingaben werden durch eine Echtzeitvalidierung überprüft. Falls eine Eingabe nicht den Anforderungen entspricht, wird eine Fehlermeldung direkt unter dem jeweiligen Feld angezeigt. Dies geschieht über eine validate()-Funktion, die sicherstellt, dass alle Felder ausgefüllt und korrekt formatiert sind.

Sobald der Nutzer das Formular absendet (onSubmit()), werden die eingegebenen Daten über eine asynchrone Anfrage an die API /api/users gesendet. Während der Registrierung wird submitting auf true gesetzt, um eine doppelte Übermittlung der Daten zu verhindern. Ist die Registrierung erfolgreich, wird der Nutzer automatisch eingeloggt, indem seine Benutzerdaten über loadMe() geladen und er auf die Startseite weitergeleitet wird.

Falls es zu einem Fehler kommt – etwa weil die E-Mail-Adresse bereits vergeben ist oder der Server ein Problem meldet – wird dies über useToast() als visuelle Fehlermeldung an den Nutzer weitergegeben. Dadurch bleibt die Nutzererfahrung flüssig und transparent.

Das Design der Registrierungsseite ist bewusst minimalistisch und modern, um eine einfache Handhabung zu gewährleisten. Die Oberfläche nutzt Nuxt UI-Komponenten, um eine optisch ansprechende und responsive Darstellung zu ermöglichen.

4.3 Feed Ansicht

Die Feed-Ansicht in „Echo“ ist das zentrale Element der Benutzeroberfläche und bietet den Nutzern eine chronologische Übersicht über Beiträge von anderen Nutzern und Communities. Sie ist so gestaltet, dass sie eine flüssige und intuitive Navigation ermöglicht und dabei moderne Designprinzipien beachtet. Der Hauptaufbau der Seite besteht aus einer Navigation, einem zentralen Content-Bereich und einem Mechanismus zum dynamischen Nachladen von Beiträgen.

Im oberen Bereich der Seite befindet sich die navigations Leiste, die es den Nutzern ermöglicht, schnell zwischen verschiedenen Bereichen der Plattform zu wechseln. Die Navigation ist kontextabhängig: Angemeldete Nutzer sehen Links zu ihrem Profil, während nicht eingeloggte Nutzer stattdessen eine Anmelden-Schaltfläche erhalten. Die Navigation bietet direkten Zugriff auf den Feed, die Communities und eine Suchfunktion. Die Symbole für die einzelnen Menüpunkte sind in einem modernen, minimalistischen Stil gehalten und stammen aus der line-md- und heroicons-Bibliothek.

Der zentrale Bereich der Seite wird durch die PostFeed-Komponente dargestellt, die für das Laden und Anzeigen der Beiträge zuständig ist. Hierbei werden Beiträge von der API /api/users/feed abgerufen und in einer strukturierten Liste angezeigt. Das Nachladen der Beiträge erfolgt dynamisch: Wenn der Benutzer nach unten scrollt, erkennt die Anwendung über die useIntersectionObserver-Funktion von VueUse, ob das Ende der Liste erreicht ist. Falls noch weitere Beiträge verfügbar sind, werden automatisch neue Posts nachgeladen, wodurch ein nahtloses Infinite-Scrolling-Erlebnis entsteht.

Jeder Beitrag wird innerhalb einer separaten Post-Karte dargestellt, die neben dem Beitragstext auch Informationen über den Autor, die Anzahl der Likes und Kommentare sowie eventuelle Auszeichnungen enthält. Diese Karten sind mit weiteren Komponenten wie CardPost, CardUser und CardAward modular aufgebaut, um eine bessere Wiederverwendbarkeit und Wartbarkeit zu gewährleisten. Das Styling der Feed-Elemente erfolgt durch Nuxt UI und TailwindCSS, wodurch ein klares und gut strukturiertes Layout erreicht wird.

Technisch basiert der Feed auf einer reaktiven Datenstruktur. Die geladenen Beiträge werden innerhalb eines ref()-Arrays gespeichert und durch v-for dynamisch gerendert. Die Funktion fetchPosts() holt die Daten asynchron aus dem Backend und überprüft, ob weitere Beiträge vorhanden sind. Falls keine neuen Inhalte mehr existieren, wird das Infinite Scrolling automatisch gestoppt, um unnötige API-Anfragen zu vermeiden. Fehler beim Laden werden über eine useToast()-Benachrichtigung dem Benutzer mitgeteilt.

4.4 Communities Ansicht

Die Community-Ansicht in Echo stellt einen zentralen Bereich der Plattform dar, in dem Nutzer bestehende Communities entdecken und verwalten sowie neue Gruppen erstellen können. Diese Ansicht wurde speziell darauf ausgelegt, eine einfache Navigation, schnelle Interaktionen und eine klare Trennung zwischen eigenen und empfohlenen Communities zu ermöglichen.

Beim Laden der Seite werden die verfügbaren Communities über die API /api/communities abgerufen und dynamisch in einer Liste von Community-Karten dargestellt. Zusätzlich werden für eingeloggte Nutzer ihre eigenen Communities über /api/users/[id]/communities

geladen. Die Community-Ansicht besteht aus zwei Haupttabs: Entdecken (für alle Communities) und Meine (für vom Nutzer beigetretene Communities). Die Implementierung nutzt ein reaktives computed()-Property, um die angezeigten Communities zu verwalten und entsprechend zu filtern.

Die Navigation innerhalb der Community-Ansicht erfolgt über eine Tab-Leiste, die je nach Login-Status unterschiedliche Inhalte anzeigt. Ist ein Nutzer angemeldet, hat er Zugriff auf einen zusätzlichen Tab „Meine“, in dem nur die Communities angezeigt werden, in denen er Mitglied ist. Dies ermöglicht eine schnelle und intuitive Verwaltung der eigenen Gruppen. Für die Darstellung der einzelnen Community-Karten wird die CardCommunity-Komponente verwendet, die Informationen zur Community, eine Vorschau des Community-Feeds und interaktive Elemente wie den Beitreten und Verlassen-Button enthält.

Eine zentrale Funktion dieser Ansicht ist die Möglichkeit, neue Communities zu erstellen. Diese Funktion wird durch ein dynamisches Formular bereitgestellt, das über FormError und FormSubmitEvent validiert wird. Nutzer können einen Namen und eine Beschreibung für ihre Community eingeben, und die Daten werden bei erfolgreicher Validierung an die API gesendet. Während der Bearbeitung eines neuen Eintrags wird der isEditing-State genutzt, um zwischen Erstellungsmodus und regulärer Ansicht zu wechseln.

Technisch basiert die Implementierung auf Nuxt 3 mit serverseitigem Fetching, sodass die Community-Daten effizient geladen und verwaltet werden. Die API-Anfragen werden mit useFetch() ausgeführt, wobei durch den immediate-Parameter sichergestellt wird, dass der Abruf für eingeloggte Nutzer sofort erfolgt. Fehler oder Netzwerkprobleme werden durch useToast() dem Nutzer als Benachrichtigung angezeigt, um eine verbesserte Usability und Fehlerbehandlung zu gewährleisten.

Die visuelle Darstellung der Communities folgt einem modernen, kartenbasierten Design, das an Plattformen wie Reddit und Discord erinnert. Jede Community wird in einer individuellen Karte (CardCommunity.vue) dargestellt, die neben Namen und Beschreibung auch Symbole für Community-Aktivitäten enthält. Nutzer können über ein einfaches Icon-basiertes Menü einer Community beitreten, verlassen oder sie durchsuchen, wodurch die Plattform sowohl für neue als auch für erfahrene Nutzer leicht zugänglich bleibt.

4.5 Community Homepage

Die Community-Homepage einer spezifischen Community in Echo bietet den Nutzern einen zentralen Ort, um sich über eine bestimmte Gruppe zu informieren, Diskussionen zu verfolgen und mit anderen Mitgliedern zu interagieren. Sie ist so gestaltet, dass sie eine Mischung aus Übersichtlichkeit, Interaktivität und intuitiver Navigation bietet.

Beim Aufruf einer Community-Seite wird zunächst die Community-ID aus der URL extrahiert, um die zugehörigen Informationen aus der API /api/communities/[id] abzurufen. Der Nutzer erhält auf der Hauptseite der Community eine detaillierte Übersicht, die aus mehreren zentralen Komponenten besteht. Im oberen Bereich befindet sich eine Community-Kopfzeile, die den Namen, die Beschreibung und das Bannerbild der Community anzeigt. Falls der Nutzer noch kein Mitglied ist, wird ihm die Möglichkeit geboten, der Community über einen Folgen-Button beizutreten. Der aktuelle Status, ob der Nutzer bereits Mitglied ist oder nicht, wird über eine API-Abfrage ermittelt und durch einen reaktiven State verwaltet.

Unterhalb der Kopfzeile befindet sich der Community-Feed, der alle aktuellen Beiträge der Gruppe enthält. Diese Beiträge werden mithilfe der CardPost-Komponente gerendert, die für die einheitliche Darstellung von Posts innerhalb der Plattform verantwortlich ist. Wie bereits in der allgemeinen Feed-Ansicht von Echo wird auch hier Infinite Scrolling genutzt, um die Ladezeiten zu optimieren und ein nahtloses Nutzererlebnis zu gewährleisten. Die Posts werden aus der API /api/communities/[id]/feed geladen und dynamisch aktualisiert.

Ein weiterer essenzieller Bestandteil der Community-Homepage ist die Mitgliederliste, die über die CardUser-Komponente dargestellt wird. Hier sehen Nutzer eine Übersicht der aktivsten Mitglieder innerhalb der Community. Dies dient nicht nur dazu, die Vernetzung innerhalb der Community zu fördern, sondern verleiht auch dem sozialen Aspekt der Plattform mehr Bedeutung. Mitglieder können sich gegenseitig Awards verleihen, ein Feature, das durch die CardAward-Komponente realisiert wird und den Gamification-Aspekt von Echo unterstreicht.

Administratoren einer Community haben zusätzliche Steuerungsmöglichkeiten. Über eine spezielle Administrationsleiste können sie Beiträge moderieren, Mitglieder verwalten und die Community-Einstellungen bearbeiten. Die Berechtigungen dafür werden durch eine computedAsync()-Funktion überprüft, die den aktuellen Nutzer mit der Community-Admin-Rolle abgleicht.

Technisch basiert die Community-Seite auf einer modularen und reaktiven Architektur. Alle relevanten Daten wie Posts, Mitglieder und Einstellungen werden in separaten fetch()-Anfragen geladen und in reaktiven States verwaltet. Fehler bei der Datenabfrage werden durch useToast() abgefangen und als visuelle Hinweise an den Nutzer ausgegeben, um eine benutzerfreundliche Fehlerbehandlung zu gewährleisten.

4.6 Post Homepage

Die Post-Homepage in Echo stellt eine zentrale Anlaufstelle für die detaillierte Betrachtung eines spezifischen Beitrags dar. Hier können Nutzer nicht nur den vollständigen Inhalt eines Posts lesen, sondern auch Kommentare hinterlassen, Likes vergeben und an Diskussionen teilnehmen. Diese Seite wurde so konzipiert, dass sie eine übersichtliche Darstellung des Hauptbeitrags mit einer dynamisch geladenen Kommentar-Sektion kombiniert.

Beim Aufrufen eines spezifischen Posts wird die Post-ID aus der URL extrahiert und die zugehörigen Daten über die API /api/posts/[id] abgerufen. Dies umfasst nicht nur den Beitrag selbst, sondern auch die Benutzerdaten des Autors, sodass relevante Informationen direkt auf der Seite angezeigt werden können. Sollte es zu einem Ladefehler kommen – beispielsweise, wenn der Post nicht existiert oder die Verbindung zur API fehlschlägt – wird eine Fehlermeldung über useToast() angezeigt und der Nutzer automatisch auf die Startseite weitergeleitet.

Die Seite ist in zwei Hauptbereiche unterteilt: Im oberen Teil befindet sich die Post-Detailansicht, die den vollständigen Inhalt des Beitrags inklusive Titel, Autor und Anzahl der Likes präsentiert. Dieser Abschnitt nutzt die CardPost-Komponente, die bereits im Feed und in der Community-Ansicht verwendet wird, um eine konsistente Darstellung der Beiträge innerhalb der Plattform sicherzustellen.

Darunter befindet sich die Kommentar-Sektion, die eine zentrale Rolle für die Interaktion mit dem Post spielt. Kommentare werden aus der API /api/posts/[id]/comments abgerufen und in einer scrollbaren Liste dargestellt. Dank der useIntersectionObserver-Funktion werden neue Kommentare automatisch nachgeladen, sobald der Nutzer ans Ende der Liste scrollt. Dies sorgt für eine flüssige und effiziente Nutzung der Seite, ohne dass manuelle Ladeaktionen erforderlich sind.

Ein Kommentarfeld ermöglicht es angemeldeten Nutzern, direkt mit anderen in Austausch zu treten. Der Eingabetext wird über einen ref()-State verwaltet, und das Abschicken eines Kommentars wird durch die submitComment()-Funktion gesteuert. Hierbei wird sichergestellt, dass leere Kommentare ignoriert werden und Shift+Enter für Zeilenumbrüche genutzt werden kann, um eine bessere Benutzerfreundlichkeit zu gewährleisten.

Interaktive Elemente wie das Liken eines Beitrags oder Kommentars werden in Echtzeit aktualisiert, sodass Nutzer direkt Feedback über ihre Aktivitäten erhalten. Die technischen Funktionen sind stark an das bereits existierende System für den Post-Feed angelehnt, jedoch mit einer spezifischen Fokussierung auf eine tiefere Diskussionsebene.

4.7 Benutzer Homepage

Die Benutzer-Homepage in Echo ist das persönliche Profil jedes Nutzers, auf dem sich alle relevanten Informationen zu seiner Aktivität innerhalb der Plattform befinden. Sie dient als zentraler Ort für die Verwaltung von Benutzerinformationen, Interaktionen mit anderen und die Darstellung der persönlichen Erfolge und Community-Beiträge.

Beim Aufruf eines Benutzerprofils wird die Benutzer-ID aus der URL extrahiert und über die API /api/users/[id] die entsprechenden Informationen abgerufen. Neben den grundlegenden Daten wie Benutzername, Profilbild und Bio werden auch Beiträge, verfolgte Communities und erhaltene Auszeichnungen angezeigt. Die Seite unterscheidet zwischen zwei Ansichten: Das eigene Profil mit erweiterten Bearbeitungsfunktionen und die Ansicht fremder Profile, die öffentlich zugängliche Informationen bereitstellt. Ein isAdmin-State überprüft, ob der aktuell eingeloggte Nutzer sein eigenes Profil betrachtet, um entsprechende Bearbeitungsfunktionen freizuschalten.

Im oberen Bereich der Profilseite wird eine Header-Sektion angezeigt, die den Namen, das Profilbild und einen Folgen-/Entfolgen-Button enthält. Der aktuelle Beziehungsstatus zwischen den Nutzern wird über die API /api/users/[id]/isfollowing überprüft und dynamisch in isFollowing gespeichert. Sollte es beim Laden der Benutzerdaten zu einem Fehler kommen – etwa weil das Profil nicht existiert – wird eine Fehlermeldung über useToast() angezeigt, und der Nutzer wird zur Startseite umgeleitet.

Darunter befindet sich eine mehrteilige Übersicht, die die Aktivität des Nutzers in verschiedenen Bereichen darstellt. Die Post-Liste zeigt alle erstellten Beiträge des Nutzers in einer sortierten Ansicht und verwendet die CardPost-Komponente, die bereits aus dem Feed bekannt ist. Die Community-Sektion zeigt alle Gruppen, denen der Benutzer angehört, und nutzt die CardCommunity-Komponente, während die Auszeichnungs-Sektion (CardAward) besondere Erfolge des Nutzers hervorhebt.

Nutzer haben die Möglichkeit, ihr Profilbild oder ihre Bio zu bearbeiten, sofern sie sich auf ihrer eigenen Profilseite befinden. Änderungen an den Benutzerdaten werden über ein dynamisches Formular umgesetzt, das mit FormError und FormSubmitEvent validiert wird. Falls gewünscht, kann der Nutzer sein Konto auch vollständig löschen, wobei eine deleteMe-Funktion genutzt wird, um die Aktion sicher durchzuführen.

4.8 Echo Explorer

Der Echo Explorer ist die zentrale Anlaufstelle für Nutzer, um neue Inhalte, Communities und Personen innerhalb der Plattform zu entdecken. Anstelle einer einfachen Suchfunktion bietet der Explorer eine dynamische Umgebung, in der Nutzer gezielt nach Beiträgen suchen oder durch Vorschläge neue Communities und interessante Profile kennenlernen

können. Die Oberfläche kombiniert eine leistungsfähige Suche mit einer intuitiven Navigation, um ein nahtloses und interaktives Erlebnis zu schaffen.

Im Mittelpunkt des Echo Explorers steht das zentrale Suchfeld, in das Nutzer schlüssel Wörter eingeben können, um relevante Inhalte zu finden. Die Suchfunktion ist in drei Hauptkategorien unterteilt: Benutzer, Communities und Beiträge. Diese Kategorien werden in Tabs mit aussagekräftigen Icons dargestellt, sodass Nutzer mühelos zwischen ihnen wechseln können.

Ein besonderer Aspekt des Echo Explorers ist das dynamische Nachladen der Ergebnisse. Mithilfe der useIntersectionObserver-Funktion wird erkannt, wenn ein Nutzer das Ende einer Liste erreicht, und automatisch die nächste Seite der Ergebnisse nachgeladen. Dies sorgt für eine flüssige und unterbrechungsfreie Navigation ohne die Notwendigkeit von manuellen Lade-Buttons. Zudem wird sichergestellt, dass nur so viele API-Anfragen gestellt werden, wie tatsächlich benötigt werden, um die Performance der Plattform zu optimieren.

Die Suchergebnisse werden in einer modularen Kartenansicht präsentiert, die sich an das Design der Plattform anpasst. Benutzerprofile werden über die CardUser-Komponente dargestellt und enthalten Profilbild, Namen und eine Folgen-Schaltfläche, um direkte Interaktion zu ermöglichen. Communities nutzen die CardCommunity-Komponente, die eine Kurzbeschreibung und eine Beitrittsoption bietet. Beiträge erscheinen in der CardPost-Ansicht, die eine Vorschau des Inhalts sowie die wichtigsten Interaktionsmetriken anzeigt.

Neben der gezielten Suche bietet der Echo Explorer auch Vorschläge basierend auf Trends und persönlichen Interessen. Dies ermöglicht es Nutzern, auch ohne eine spezifische Suchanfrage interessante Inhalte zu entdecken. Diese Vorschläge können beispielsweise auf beliebten Posts, häufig beigetretenen Communities oder neu erstellten Benutzerprofilen basieren.

Ein besonderes Augenmerk liegt auf der Usability und der nahtlosen Navigation. Falls eine Suchanfrage keine Treffer liefert oder ein Netzwerkfehler auftritt, wird der Nutzer durch eine visuelle Benachrichtigung mit useToast() darüber informiert. Zudem wird die Suchhistorie in der URL gespeichert, sodass Nutzer durch die Zurück-Navigation in ihrem Browser zu vorherigen Suchergebnissen zurückkehren können.

5 API-Übersicht

Um eine moderne, flexible und skalierbare Architektur zu gewährleisten, haben wir uns entschieden, eine API (Application Programming Interface) zu entwickeln, um die Kommunikation zwischen unserer relationalen SQL-Datenbank und den verschiedenen Frontend Anwendungen zu ermöglichen. Diese Entscheidung bietet uns zahlreiche Vorteile, insbesondere im Hinblick auf Modularität, Portabilität und Skalierbarkeit. Durch die Trennung von Datenbank- und Frontend-Logik entsteht eine klare Struktur, die die Wartbarkeit und Erweiterbarkeit unserer Software deutlich vereinfacht. Jede Komponente kann unabhängig voneinander weiterentwickelt werden, ohne dass dies negative Auswirkungen auf andere Bereiche der Anwendung hat.

Mit einer API schaffen wir zudem die Grundlage für Portabilität, da unsere Datenbank problemlos mit verschiedenen Arten von Anwendungen kommunizieren kann. Neben unserem bestehenden Frontend auf Basis des Nuxt-Frameworks lässt sich die API auch einfach in zukünftig geplante mobile oder Desktop-Anwendungen integrieren. Dadurch wird unsere Architektur langfristig flexibel und anpassungsfähig. Gleichzeitig erlaubt uns dieser Ansatz, die Anwendung zu kalieren, um steigenden Anforderungen an Datenverarbeitung und Benutzerinteraktion gerecht zu werden. Durch die Möglichkeit, die Last auf mehrere Instanzen zu verteilen, erreichen wir eine deutlich bessere Performance und können Wachstum effektiv unterstützen.

Die API basiert auf dem REST-Prinzip (Representational State Transfer), einem etablierten Architekturstil für verteilte Systeme. REST definiert klare Regeln für die Kommunikation zwischen Client und Server und ermöglicht eine ressourcenorientierte Datenübertragung über standardisierte HTTP-Methoden wie GET, POST, PUT und DELETE. Jede Ressource innerhalb der API wird über eine eindeutige URL identifiziert und durch eine klar definierte Schnittstelle angesprochen. Ein zentrales Merkmal der REST-Architektur ist ihre Statelessness, bei der jede Anfrage alle notwendigen Informationen enthält, sodass der Server keine Sitzungsdaten zwischen den Anfragen speichern muss. Dies erhöht die Skalierbarkeit und vereinfacht das Lastmanagement erheblich. Die Verwendung von Standardprotokollen macht die Integration und Weiterentwicklung der API zudem deutlich einfacher und sorgt für eine hohe Interoperabilität mit anderen Systemen.

Die Kommunikation zwischen dem Frontend und unserer API erfolgt über das JSON-Format (JavaScript Object Notation). JSON ist ein leichtgewichtiges Datenformat, das speziell für den Datenaustausch zwischen Systemen entwickelt wurde. Es ist einfach lesbar, sowohl für Menschen als auch für Maschinen, und wird von nahezu allen modernen Programmiersprachen unterstützt. Die Struktur von JSON basiert auf Schlüssel-Wert-Paaren, die es ermöglichen, komplexe Datenstrukturen wie Objekte, Arrays und verschachtelte Hierarchien abzubilden.

Für die technische Umsetzung setzen wir auf das Nuxt-Framework in Kombination mit dem H3-Server. H3 ist ein leichtgewichtiger, moderner Webserver, der speziell für Node-basierte Anwendungen entwickelt wurde und integraler Bestandteil von Nuxt 3 ist. Er bildet die Basis für unsere API und ermöglicht uns, serverseitige Logik und API-Endpunkte nahtlos in derselben Codebasis zu verwalten. Diese enge Integration reduziert die Komplexität unserer Anwendung und bietet uns eine leistungsstarke Umgebung für die Entwicklung. H3 unterstützt dabei asynchrone Programmierung sowie Middleware-basierte Architekturen, was uns erlaubt, flexible und skalierbare APIs zu entwickeln. Dies macht die Kombination aus einer REST-basierten API und der leistungsstarken Nuxt-Plattform zu einer robusten und zukunftssicheren Lösung für unsere Anwendung.

5.1 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.

5.2 GET: `/api/awards/[id]`

Diese Route wird verwendet, um eine spezifische Auszeichnung (Award) anhand ihrer `id` abzurufen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

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

Antwort:

- Ein Award-Objekt mit den im System gespeicherten Details.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID der Auszeichnung fehlt.
- `400 No Id specified`: Die ID wurde nicht korrekt angegeben.
- `404 Award not found`: Die gesuchte Auszeichnung existiert nicht.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

5.3 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.

5.4 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.

5.5 POST: `/api/awards/search`

Diese Route wird verwendet, um Auszeichnungen (Awards) anhand einer Suchanfrage zu durchsuchen.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Body-Parameter:**
  - `query` (erforderlich, string): Suchbegriff für den Namen der Auszeichnung
  - `communityId` (optional, integer): Falls angegeben, wird die Suche auf eine bestimmte Community beschränkt

Antwort:

- Eine Liste von Award-Objekten, die dem Suchbegriff entsprechen.

Fehlerfälle:

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

5.6 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.

5.7 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.

5.8 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.

5.9 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.

5.10 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.

5.11 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.

5.12 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.

5.13 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.

5.14 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.

5.15 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.

5.16 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.

5.17 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.

5.18 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.

5.19 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.

5.20 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.

5.21 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.

5.22 GET: `/api/communities/[id]/awards`

Diese Route wird verwendet, um die Auszeichnungen (Awards) einer bestimmten Community anhand ihrer `id` abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community, deren Auszeichnungen abgerufen werden sollen
- **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 der angegebenen Community.

Fehlerfälle:

- `400 Invalid pagination parameters`: Die angegebenen Paginierungsparameter sind ungültig.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.
- `404 Community not found`: Die angegebene Community existiert nicht.

5.23 GET: `/api/communities/[id]/isfollowing`

Diese Route wird verwendet, um zu überprüfen, ob der eingeloggte Benutzer einer bestimmten Community folgt.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community, deren Follow-Status überprüft werden soll

Antwort:

- Ein Boolean-Wert, der angibt, ob der Benutzer der Community folgt.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Id parameter is missing`: Die Community-ID fehlt.
- `404 Community not found`: Die angegebene Community existiert nicht.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

5.24 GET: `/api/communities/[id]/posts`

Diese Route wird verwendet, um die Beiträge (Posts) einer bestimmten Community anhand ihrer `id` abzurufen. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID der Community, deren Beiträge abgerufen werden sollen
- **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 der angegebenen Community.

Fehlerfälle:

- `400 Id parameter is missing`: Die ID der Community fehlt.
- `404 Community not found`: Die angegebene Community existiert nicht.
- `400 Invalid pagination parameters`: Die angegebenen Paginierungsparameter sind ungültig.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

5.25 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.

5.26 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.

5.27 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.

Zur Sitzungsverwaltung im Webbrowser wird ein Cookie names "key"verwendet, um die Benutzeridentität zu speichern. Es wird geprüft, ob ein gültiger Login-Eintrag vorhanden ist. Falls ja, wird dieser wiederverwendet, ansonsten wird ein neuer Eintrag erstellt. Das key-Cookie wird anschließend gesetzt und enthält Informationen zur Sitzung. Dieses Cookie ist 24 Stunden lang gültig, wird nur über HTTPS übertragen und ist mit „SameSite: Strict“ übertragen, um Cross-Site-Request-Forgery (CSRF) zu verhindern.

Die Authentifizierungs-Middleware liest das key-Cookie bei jeder eingehenden Anfrage aus und überprüft es in der Datenbank. Wenn ein passender Login-Eintrag existiert und dieser jünger als 24 Stunden ist, wird die Sitzung als gültig betrachtet und im event.context gespeichert. Falls der Eintrag älter ist, wird er gelöscht und die Anfrage als nicht authentifiziert betrachtet. Um die Konsistenz der Datenbank sicherzustellen, werden ältere Login-Einträge regelmäßig bereinigt, sodass nur gültige Sitzungen bestehen.

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.

5.28 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.

5.29 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.

5.30 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.

5.31 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.

5.32 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.

5.33 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.

5.34 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.

5.35 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.

5.36 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.

5.37 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.

5.38 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.

5.39 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.

5.40 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.

5.41 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.

5.42 GET: `/api/users/me`

Diese Route wird verwendet, um das Profil des aktuell eingeloggten Benutzers abzurufen.

Anmeldung: Anmeldung erforderlich.

Eingabe: Keine Eingabe erforderlich.

Antwort:

- Ein Benutzerobjekt mit den Profildaten des eingeloggten Nutzers.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.

Hinweis: Für den Zugriff auf die Benutzerdaten im Frontend kann das Composable `useAuth` verwendet werden.

5.43 GET: `/api/users/[id]/followers`

Diese Route wird verwendet, um die Follower eines bestimmten 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, dessen Follower abgerufen werden sollen
- **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, die dem Benutzer folgen.

Fehlerfälle:

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

5.44 GET: `/api/users/[id]/following`

Diese Route wird verwendet, um die Benutzer abzurufen, denen ein bestimmter Benutzer anhand seiner `id` folgt. Die Pagination erfolgt über die Parameter `page` und `limit`.

Anmeldung: Keine Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers, dessen gefolgten Benutzer abgerufen werden sollen
- **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, denen der Benutzer folgt.

Fehlerfälle:

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

5.45 GET: `/api/users/[id]/isfollowing`

Diese Route wird verwendet, um zu überprüfen, ob der eingeloggte Benutzer einem bestimmten Benutzer folgt.

Anmeldung: Anmeldung erforderlich.

Eingabe:

- **Routen-Parameter:**
  - `id` (erforderlich, integer): ID des Benutzers, dessen Follow-Status überprüft werden soll

Antwort:

- Ein Boolean-Wert, der angibt, ob der eingeloggte Benutzer dem Benutzer mit der angegebenen ID folgt.

Fehlerfälle:

- `401 Unauthorized`: Der Benutzer ist nicht eingeloggt.
- `400 Id parameter is missing`: Die Benutzer-ID fehlt.
- `404 UserId not found`: Der angegebene Benutzer existiert nicht.
- `400 Database request failed`: Fehler bei der Datenbankabfrage.

5.46 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.

5.47 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.

5.48 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.

5.49 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.

5.50 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.

5.51 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.

5.52 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.

5.53 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.

6 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.

6.1 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.

6.2 Automatische kontinuierliche Bereitstellung

Die Continuous Integration (CI) und Continuous Deployment (CD)-Pipeline von Echo ermöglicht eine automatisierte, zuverlässige und kontinuierliche Bereitstellung der Anwendung. CI/CD ist ein essenzielles Konzept in der modernen Softwareentwicklung, das sicherstellt, dass neue Code-Änderungen schnell getestet und bereitgestellt werden können, ohne manuelle Eingriffe. Continuous Integration bezieht sich dabei auf das automatische Bauen und Testen des Codes bei jeder Änderung, während Continuous Deployment bedeutet, dass die getesteten Änderungen automatisch auf den Produktionsserver ausgerollt werden. Diese Automatisierung verringert die Wahrscheinlichkeit von Fehlern, beschleunigt den Entwicklungsprozess und sorgt für eine stabile und jederzeit verfügbare Anwendung.

Das bereitgestellte Deploy-Skript (deploy.sh) ist der zentrale Bestandteil der Deployment-Pipeline und übernimmt die automatisierte Bereitstellung von Echo auf dem Produktionsserver. Das Skript stellt sicher, dass die neueste Version der Anwendung aus dem Git-Repository geladen, installiert und konfiguriert wird. Zu Beginn des Skripts wird überprüft, ob es mit Root-Rechten ausgeführt wird, da für bestimmte Operationen wie das Starten von Systemdiensten und das Schreiben in /opt/echo administrative Berechtigungen erforderlich sind. Zusätzlich stellt das Skript sicher, dass das System unter Debian läuft und systemd als Init-System verwendet wird, da Echo als systemd-Service betrieben wird.

Als Nächstes prüft das Skript, ob alle erforderlichen Abhängigkeiten (git, node, npm) installiert sind, bevor es mit dem Deployment fortfährt. Falls der echo.service bereits aktiv ist, wird dieser gestoppt, um eine saubere Aktualisierung der Anwendung zu gewährleisten. Danach wird der bestehende Anwendungsordner gelöscht und die neueste Version aus dem Repository frisch geklont. Dadurch wird sichergestellt, dass das Deployment immer auf

der neuesten stabilen Version basiert. Anschließend werden die Umgebungsvariablen aus der Datei env in das Hauptverzeichnis kopiert, um die Anwendung mit den richtigen Konfigurationen zu starten.

Nachdem der Code aktualisiert wurde, installiert das Skript alle notwendigen Node.js-Abhängigkeiten mit npm install und führt eine Prisma-Migration (npx prisma migrate dev –name dev) durch. Diese Migration sorgt dafür, dass alle Änderungen an der Datenbankstruktur automatisch angewendet werden, sodass die Anwendung immer mit der aktuellen Datenbankversion kompatibel bleibt. Danach wird der Code mit npm run build kompiliert, um die Anwendung für den Produktivbetrieb bereitzustellen.

Im Anschluss daran werden die Konfigurationsdateien für systemd und nginx in die entsprechenden Verzeichnisse kopiert. Die Datei echo.service stellt sicher, dass die Anwendung als hintergrundlaufender Dienst gestartet wird, während die nginx.conf-Konfiguration als Reverse Proxy dient, um die Anfragen über HTTPS weiterzuleiten. Das Skript setzt die korrekten Dateiberechtigungen, indem es den Besitzer des Anwendungsordners auf www-data ändert, sodass der Webserver und der Node.js-Dienst darauf zugreifen können.

Zuletzt führt das Skript ein systemctl daemon-reload aus, um die systemd Konfigurationsdateien neu einzulesen, und aktiviert sowie startet echo.service und nginx.service. Dadurch wird sichergestellt, dass die Anwendung automatisch beim Systemstart geladen wird und stets verfügbar bleibt. Falls Echo bereits läuft, wird der Service mit systemctl restart neu gestartet, um die aktualisierte Version der Anwendung direkt bereitzustellen.

Zusätzlich sorgt die NGINX-Konfiguration für eine sichere und optimierte Bereitstellung der Web-App. Die Konfiguration enthält eine automatische HTTP zu HTTPS Weiterleitung, moderne TLS-Settings und verschiedene Sicherheitsheader wie Strict-Transport-Security und X-Frame-Options, um Angriffe wie Clickjacking und Man-in-the-Middle-Angriffe zu verhindern. Außerdem implementiert sie ein Rate-Limiting, um Missbrauch zu reduzieren, und ein effizientes Reverse Proxy Setup, das die Anfragen an den Nuxt-Server weiterleitet.

Die Datei env enthält die wichtigsten Umgebungsvariablen für die Anwendung, einschließlich des Ports (8080), der Datenbank-URL (file:/opt/echo/prisma/dev.db) sowie des Speicherpfads für Bilder (/opt/echo/images). Diese Datei wird vom Skript automatisch an den richtigen Ort kopiert, sodass die Anwendung mit den richtigen Konfigurationswerten gestartet werden kann.

6.3 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.