Dokumentation

Beschreibung

Inhalt der Anwendung

In dieser Anwendung haben Nutzer die Möglichkeit, private und öffentliche Notizen zu erstellen. Diese kann man dann in einer Suche suchen und sich anzeigen lassen. Die Notizen können in Markdown oder mit HTML-Tags versehen werden. Unter anderem kann man private Notizen durch das Teilen des Links anderen Nutzern zeigen.

Gruppenmitglieder

Moritz Foglia und Alexander Afanasjew

Verwendete Technologien

Wir haben den T3-Stack verwendet. Dieser umfasst folgende Komponenten:

1. **Next.js**: Als React-Framework bietet Next.js serverseitiges Rendering und statische Seiten-Erzeugung, was die Leistung und Benutzerfreundlichkeit der Anwendung verbessert.
2. **TypeScript**: Durch die Typisierung erhöht TypeScript die Codequalität und erleichtert die Wartung, da Fehler frühzeitig und einfacher erkannt werden.
3. **tRPC**: Für die Erstellung von Endpunkten und APIs wurde tRPC eingesetzt. Es ermöglicht typsichere APIs ohne die Notwendigkeit, zusätzliche Code-Generatoren oder Typdefinitionen zu verwenden. Über den Kontext “ctx”, haben geschützte Prozeduren im Backend Zugriff auf die User-Session und den Prisma-Client.
4. **Prisma**: Als ORM (Object-Relational Mapping) wurde Prisma genutzt, um die Datenbankabfragen zu vereinfachen und eine typsichere Interaktion mit der Datenbank zu gewährleisten. Durch den Prisma-Client ist die ganze Anwendung gegen SQL-Injections geschützt, da dieser nur Prepared Statements benutzt.
5. **Tailwind CSS**: Für das Styling der Benutzeroberfläche wurde Tailwind CSS verwendet. Es ermöglicht eine schnelle und konsistente Gestaltung durch vordefinierte Klassen.
6. **Auth.js**: Für die Authentifizierungs-Middleware wurde NextAuth.js verwendet. Es bietet eine flexible und einfache Möglichkeit, verschiedene Authentifizierungsstrategien (z. B. OAuth, E-Mail, Credentials) zu integrieren und sich nahtlos in Next.js-Anwendungen zu integrieren. NextAuth.js unterstützt sowohl die clientseitige als auch die serverseitige Authentifizierung.
7. **Docker:** Zum Plattformunabhängigen Aufbauen unserer Anwendung
8. **Mailhog:** Zum Abfangen der E-Mails
9. **Treafik:** Zum Aufsetzen einer Reverse-Proxy und einer Rate Limit Middleware
10. **Postgres:** Datenbank Server

Zusätzlich haben wir auch Bibliotheken importiert, welche wir vorher auf die Kriterien, die in der Vorlesung genannt wurden, geprüft haben:

1. **React-Markdown**: Zum Rendern der Markdown/HTML-Syntax
2. **Rehype-Raw**: React-Markdown-Plugin zum verarbeiten von HTML innerhalb von Markdown
3. **Remark-GFM**: React-Markdown Plugin zur Unterstützung von GitHub Flavored Markdown
4. **Dompurify**: Zur Sanitisierung der Markdown und HTML eingabe, zum Schutz vor XSS
5. **Bcrypt**: Zum Hashen und Salten von Passwörtern
6. **Zod**: Zur validierung von Eingaben
7. **tailwind typographie plugin**: Wird in der tailwind.config.ts als plugin initialisiert. Es ist dafür da, dass die gerenderte Notiz gestylt ist.
8. **zxcvbn-ts**:Bibliothek zum Testen der Passwortstärke
9. **playwright:** Bibliothek zum Tests schreiben
10. **nodemailer:** Bibliothek zum Versenden von E-Mails

Infrastruktur

CI/CD

Über die Security Einstellung in unserem Repository haben wir die Dependabot alerts eingeschaltet, wodurch wir Benachrichtigungen bekommen, wenn es eine Schwachstelle für eine Dependencie gibt. Außerdem werden für diese Dependencies Pullrequests erstellt, um sie zu aktualisieren.

Für den Frontend- und Backend-Test nutzen wir Playwright, dort testen wir im Frontend bei der Registrierung, ob die Felder alle richtig erkannt werden und für das Backend testen wir, ob der richtige Error geworfen wird, wenn man ein zu kurzes Passwort eingibt.

In der ci-cd.yml Datei legen wir fest, dass diese Tests bei jedem push und bei jeder pull-request auf main oder einen feature/ Branch ausgeführt werden. Um genau zu sein, wird unsere Anwendung einmal komplett über docker compose gebaut und dann werden die Tests durchgeführt.

Docker-Compose Datei:

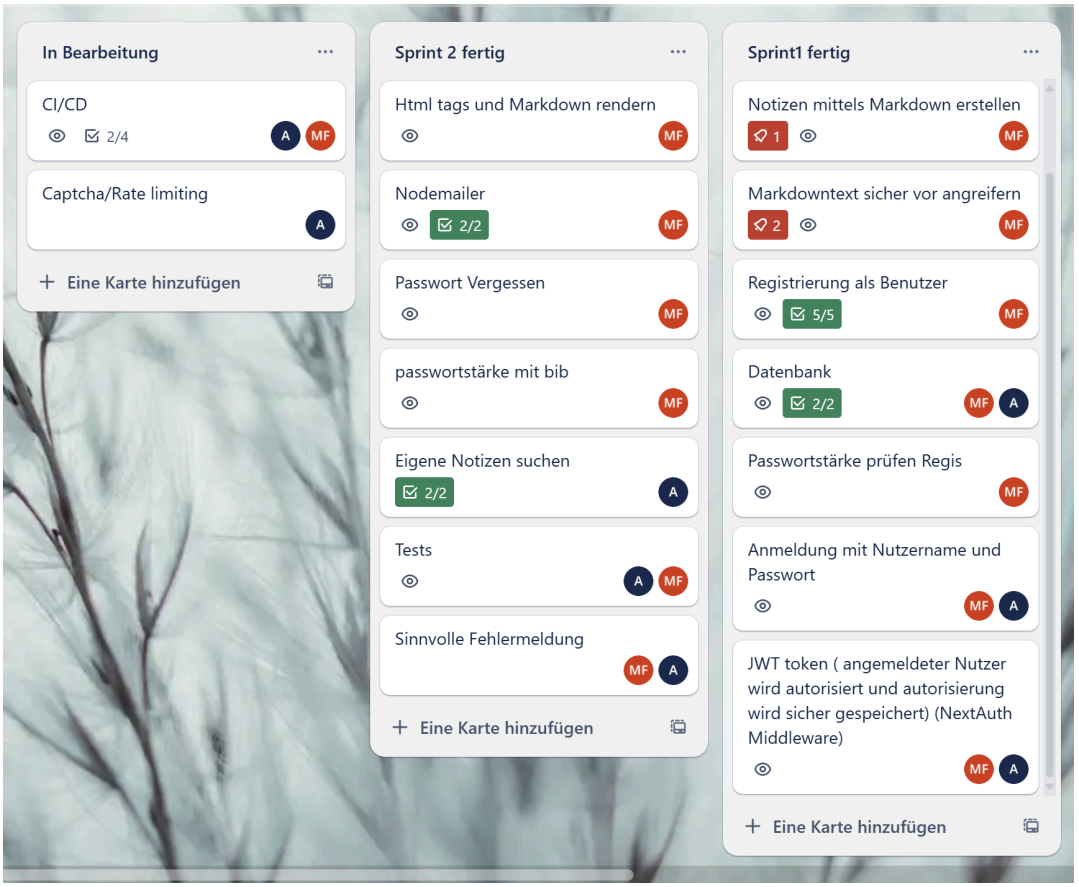
1. Starten von Mailhog, zum Abfangen von E–Mails
2. Aufsetzen einer Postgres-Datenbank und Angabe eines Volumes zum Speichern der Daten.
3. Aufsetzen der App über eine Dockerfile, welche die nötigen Dependencies installiert, die Anwendung baut und über “server.js” auf dem Port 3000 intern laufen lässt.
4. Ein Service zum Aktualisieren der Datenbank (dbPusher) wird benötigt, da in der Dockerfile das Verwenden des Push-Befehls nicht möglich ist.
5. Treafik wird verwendet, um eine Reverse-Proxy mit Ratelimit aufzusetzen. Zugriff auf die Anwendung erfolgt über http://notes.localhost/.

Verwendete IDE

Wir haben beide Visual Studio Code benutzt, das ESLint-Plugin wird für Fehlererkennung und Prüfung der Codequalität verwendet. Die Prettier Erweiterung dient der einheitlichen Formatierung des Codes.

Struktur des Entwicklungsprozesses

Wir haben ein Kanban-Board auf Trello erstellt, wo wir alle Features eingetragen haben, diese haben wir dann aufgeteilt und nach und nach in Sprints bearbeitet. Außerdem haben wir auf GitHub mit Pullrequests gearbeitet, ergo man konnte erst sein neues Feature auf den Main-Branch mergen, wenn der andere aus der Gruppe es überprüft hat. Außderdem ist der Main-Branch geschützt, sodass man nicht einfach darauf pushen kann.



Funktionen

Registrierung

1. Formulareingabe
    a. Benutzer muss einen Usernamen, eine E-Mail und zweimal das gleiche Passwort eingeben
2. Frontend-Validierung
    a. Im Frontend wird anhand des Boolean isFormValid überprüft, ob der Nutzer die Felder korrekt ausgefüllt hat.
    b. Mit der zxcvbn-ts Bibliothek wird die Passwortstärke überprüft und anhand eines Balken wird dargestellt, ab wann das Passwort sicher genug ist.
3. Button drücken
    a. Sobald isFormValid true ist, ist der Button drückbar.
    b. Wenn man den Button drückt, aber das Passwort zu schwach ist, bekommt man eine dementsprechende Antwort.
    c. Falls alles klappt, wird man zum Login weitergeleitet.
4. Backend-Validierung
    a. Mit Zod wird die Eingabe überprüft, damit fehlerhafte oder unvollständige Eingaben schon vor der Verarbeitung erkannt werden
    b. die Optionen für die zxcvbn-ts Bibliothek werden so gesetzt, dass sie das Passwort auch mit Wörtern aus dem deutschen Duden überprüft
    c. Falls nicht alle Felder ausgefüllt sind, wird ein Error geschmissen
    d. Falls die Passwortstärke nicht ausreicht, wird ein Error geschmissen, welcher den Grund zurück gibt, warum das Passwort zu schlecht ist
    e. Falls alles passt, wird das Passwort mit bcrypt gehashed und ein salt angehängt.
    f. Es wird gecheckt, ob der Username oder die E-Mail schon vergeben sind.
    g. Falls alles passt, wird der User erstellt
5. Schutzmechanismen
    a. Durch das Verwenden von Prisma werden Prepared Statements benutzt. Damit werden SQL-Injections verhindert.
    b. Die Sicherstellung von starken Passwörtern durch zxcvbn-ts sorgt dafür, dass Accounts nicht so einfach geknackt werden.
    c. Durch einen Rate-Limiter ist die Registrierung vor Brute-Force Angriffen geschützt

Login

1. Formulareingabe
    a. Benutzer muss seinen Benutzernamen und sein Passwort eingeben
2. Frontend-Validierung
    a. Es wird geprüft, ob die Felder im Formular ausgefüllt wurden, dadurch wird der Button zum Einloggen freigeschaltet
3. Backend-Validierung
    a. Die durch die Auth.js Konfiguration (Credentials-Provider) bereitgestellte signIn Funktion wird verwendet, um zu prüfen, ob ein Passwort und Nutzername zwischen 1 und 100/200 Zeichen vorhanden sind. Dann werden die Nutzerdaten aus der Datenbank geholt (Error, falls Nutzer nicht vorhanden) und der Passwort-Hash wird über bcrypt mit dem eingegebenen Passwort geprüft.
    b. Erfolg -> Login (User bekommt Session),
    Kein Erfolg -> Error
4. Schutzmechanismen
    a. Durch das Verwenden von Prisma werden Prepared Statements benutzt. Damit werden SQL-Injections verhindert.
    b. Durch einen Rate-Limiter ist das Login vor Brute-Force Angriffen geschützt
    c. Auth.js verwendet spezielle CSRF Token zum Schutz gegen CSRF

Login mit Discord

1. Weiterleitung an Discord
    a. Auth.js leitet den Benutzer zur Discord-OAuth2-Seite weiter
    b. Discord zeigt die Berechtigungsabfrage an (z.B. Zugriff auf Profilinformationen)
2. Benutzer stimmt zu
    a. Der Benutzer authentifiziert sich bei Discord und genehmigt den Zugriff
    b. Discord leitet den Benutzer mit einem Authorization Code zurück an deine Anwendung
3. Token-Austausch
    a. Auth.js tauscht den Authorization Code gegen einen Access Token und ggf. einen Refresh Token aus
    b. Die Tokens können nur ausgetauscht werden, wenn die richtige Kombination aus Client-ID und Client-Secret vorliegt, diese müssen in der .env vorhanden sein
4. Benutzerinformationen
    a. Mit dem Access Token ruft Auth.js die Benutzerinformationen von Discord ab (z.B. Benutzername, E-Mail)
5. Sitzung erstellen
    a. Auth.js erstellt eine Sitzung für den Benutzer und speichert diese sicher, entweder als JWT oder in einer Datenbank
6. Schutzmechanismen
    a. Alle Anfragen an Discord und deine Anwendung laufen über HTTPS, um die Daten während der Übertragung zu schützen
    b. Discord verwendet das OAuth2-Standardprotokoll, das sicherstellt, dass Tokens nur autorisierten Clients zugänglich sind

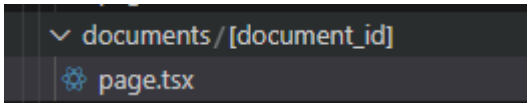
Autorisierung

1. Die öffentlich aufrufbaren Elemente/Seiten der Anwendung werden durch einen Regex-Matcher in middleware.ts festgelegt. Wenn auf eine nicht erlaubte Ressource zugegriffen wird, leitet Auth.js die Anfrage zu /auth/signin weiter.
2. Beim Authentifizieren eines Benutzers generiert Auth.js einen jwt-Token, der die UUID des Nutzers enthält. Dieser wird bei der Anmeldung überprüft, um dem User eine Session mit UUID zu geben..
3. Der tRPC Kontext fragt bei jedem tRPC-Request die auth() Funktion ab, welche die Session des anfragenden Nutzers zurückgibt.
4. Bei einer Public Procedure muss der Nutzer keine Session haben, bei einer Protected Procedure schon. Dadurch wird der Nutzer autorisiert oder eben nicht.

Notizen Erstellung

1. Formulareingabe
    a. Benutzer gibt Titel und den Inhalt in ein Textfeld ein
    b. Benutzer kann mit einem Button die Privatsphäre der Notiz einstellen
2. Notizenvorschau
    a. Bevor der Inhalt in der Vorschau gerendert wird, wird er in dem useEffect mit der DOMPurify.sanatize-Funktion gesäubert, wodurch schädliche HTML-Tags entfernt werden (Schutz vor XSS).
    b. Der Inhalt wird dann mit der Bibliothek ReactMarkdown und den Plugins rehypeRaw und remarkGfm gerendert.
3. Notizerstellung
    a. Beim Absenden des Fomurlars, wird der Inhalt vorsichtshalber nochmal mit der DOMPurify.sanatize-Funktion gesäubert und dann ans Backend geschickt.
4. Trpc-Route
    a. Die Route ist eine protectedProcedure (nur authentifizierte Nutzer können auf die Route zugreifen)
    b. mit Zod validieren wir die Eingaben zur Route, damit fehlerhafte oder unvollständige Eingaben schon vor der Verarbeitung erkannt werden
    c. dann überprüfen wir erneut ob der user eine Session besitzt
    d. und daraufhin erstellen wir die Notiz mit einem Prepared Statement von Prisma. Dadurch werden SQL-Injections verhindert.

Notizen anzeigen/ Linkstruktur

1. Anzeige
    a. Rendering, durch die React-Komponente Note wird die Notiz mit Hilfe von ReactMarkdown und den Plugins rehypeRaw und remarkgfm gerendert.
    b. Der Inhalt wird vor dem Rendern mit der DOMPurify.sanatize-Funktion gesäubert (Schutz vor XSS).
2. Linkstruktur
    a. 

Durch diese Ordnerstruktur benutzen wir dynamisches Routing von NextJs. Der Ordner “documents” repräsentiert eine Route in der Anwendung. Der Ordner [document_id] zeigt an, dass dieser Teil der URL dynamisch ist. Wenn man nun eine Notiz erstellt hat, pusht der Router einen zu der URL documents/[note.id]. Die Note.id repräsentiert die UUID der Notiz. Die dynamische Route erstellt automatisch einen Parameter, welcher an die Document-Seite übergeben wird, der ist in unserem Fall die uuid der Notiz und diese senden wir dann ans Backend, um die Notiz zu holen
3. Trpc-Route
    a. Die Route ist eine protectedProcedure (nur authentifizierte Nutzer können auf die Route zugreifen)
    b. Als erstes wird die Eingabe wieder mit Zod validiert, die Eingabe muss eine uuid sein
    c. Dann werden mit Prisma per Prepared Statement die Daten geholt, falls keine Notiz unter dieser UUID existiert, wird ein entsprechender Error geworfen.
    d. Bei Erfolg wird am Ende die Notiz zurückgegeben
4. Sicherheit
    a. Dadurch, dass die Linkstruktur so aufgebaut ist, dass sie mit der UUID der Notiz verbunden ist, ist es einem Angreifer nicht effektiv möglich, gezielt eine private Notiz eines anderen Users zu lesen, da es zu viele mögliche UUIDs gibt.

Notiz Suchen

1. Notizen können auf der Seite /documents/search über ein Eingabefeld gesucht werden.
    a. Beim Eingeben wird die Eingabe encodiert und der NextJs-Router navigiert einen zur selben URL mit der Eingabe als neuen Suchparameter (Schutz vor URL-Struktur-Manipulation und XSS)
    b. Beim Laden der Seite wird geprüft, ob der Suchbegriff in Titeln der Notizen des Nutzers vorhanden ist, die UUID und der Titel dieser wird dann zurückgegeben.
    c. Die oben genannte Prüfung erfolgt über den Prisma-Client, welcher Prepared Statements verwendet, um vor SQL-Injections zu schützen.
    d. Für jede vorhandene Notiz wird der Titel und ein Link, der zur Notiz führt, angezeigt

Reset Password

1. Passwort zurücksetzen – Initialisierung:
    a. Auf der Sign-In-Seite gibt es den Link „Passwort vergessen“.
    b. Ein Klick darauf leitet auf eine Seite weiter, wo die E-Mail-Adresse zweimal eingegeben werden muss.
2. Prüfungen:
    a. Beide E-Mails müssen übereinstimmen.
    b. Es wird geprüft, ob die E-Mail einem existierenden Benutzer zugeordnet ist.
    c. E-Mail nicht zugeordnet: Keine E-Mail wird verschickt, aber dieselbe Erfolgsmeldung wird angezeigt wie bei einer gültigen E-Mail, das dient dazu, einem Angreifer nicht die vorhandenen Nutzer E-Mails preiszugeben.
    d. E-Mail zugeordnet: Es wird eine E-Mail mit einem Link zur Passwort-Zurücksetzung versendet
3. Link zur Passwortzurücksetzung:
    a. Der Link hat die Struktur: /auth/reset-password/[token].
    b. Der Token wird generiert mit crypto.randomByte(32).toString("hex").
4. Token-Eigenschaften:
    a. Eine Stunde gültig.
    b. Wird in der Datenbank gespeichert und mit dem Benutzer verknüpft.
5. Aufruf des Links:
    a. Der Token wird aus der URL gelesen und mit der Funktion getToken überprüft.
    b. Ungültiger Token: Fehlermeldung wird angezeigt.
    c. Gültiger Token: Der Benutzer kann sein Passwort zurücksetzen.
6. Passwort zurücksetzen:
    a. Beide Passwörter müssen übereinstimmen und stark genug sein (zxcvbn-Prüfung)
    b. Vor dem Zurücksetzen wird erneut geprüft, ob der Token gültig ist.
    c. Das neue Passwort wird gehasht.
    d. Der Benutzer wird in der Datenbank aktualisiert.
    e. Der Token wird aus der Datenbank gelöscht.
    f. Der Benutzer wird zur Sign-In-Seite weitergeleitet.
7. Schutz:
    a. Brute-Force Schutz durch Rate Limiter und Länge des Tokens

Weitere Schwachstellen und deren Vorbeugung:

1. DDOS-Angriffe:
    a. Eine Reverse Proxy wird mittels Treafik aufgesetzt, diese stellt eine Ratelimit Middleware zur Verfügung. Siehe docker-compose.yml
2. Umgebungsvariablen im Repository:
    a. Die Umgebungsvariablen werden im Projekt in einer .env Datei gespeichert, welche in der .gitignore eingetragen ist..

Datenschutz:

An sensiblen Daten werden nur die E-Mail und das Passwort des Nutzers in der Datenbank gespeichert. Das Passwort ist mit bcrypt sicher gehasht, damit die Passwörter der Nutzer bei einem Datenbankleck nicht kompromittiert sind. Die E-Mail Adresse hat einen geringeren Wert für Angreifer und wird deshalb ohne Hash gespeichert.