GetraenkeIO: Eine Getränkelagerverwaltung mit Bestandsstatistik für Vereinsheime

Technischer Bericht: CL-TR-2025-42, Juli 2025

Dotzler Martin, Ehrles Andreas, Taach Eduard, Wegerer Nikolas,
Weinhut Justin, Christoph P. Neumann
CyberLytics-Lab an der Fakultät Elektrotechnik, Medien und Informatik
Ostbayerische Technische Hochschule Amberg-Weiden
Amberg, Deutschland

***Zusammenfassung*—GetraenkeIO ist eine webbasierte Software zur Verwaltung des Getränkelagers in Vereinen oder anderen Einrichtungen. Sie ermöglicht es registrierten Benutzern, Getränke eigenständig zu buchen, während Administratoren mit erweiterten Rechten Zugriff auf Bestandsstatistiken und Verwaltungsfunktionen haben. Das System unterstützt die effiziente Verwaltung des Lagers.**

Technologisch basiert die Anwendung auf einem Python-Backend, das über eine REST-API mit einem React-basierten Frontend kommuniziert. Die Lagerbestände und andere relevante Daten werden in einer relationalen PostgreSQL-Datenbank gespeichert. Docker wird verwendet, um die Anwendung flexibel bereitzustellen und eine einfache Skalierbarkeit zu gewährleisten.

***Index Terms*—Docker, React, TypeScript, Python, PostgreSQL, Container, Cloud**

I. Einleitung und Projektziele

In vielen Vereinsheimen stellt die Getränkeverwaltung nach wie vor eine organisatorische Herausforderung dar. Die Ausgabe und Bezahlung erfolgen häufig auf Vertrauensbasis, wodurch es schwierig ist, den Überblick über Lagerbestände und Einnahmen zu behalten. Auch gängige Hilfsmittel wie Strichlisten oder Excel-Tabellen sind fehleranfällig und erfordern eine regelmäßige, manuelle Pflege.

GetraenkeIO bietet hierfür eine digitale und einfach bedienbare Lösung. Die webbasierte Anwendung stellt eine benutzerfreundliche Oberfläche bereit, über die Vereinsmitglieder selbstständig Getränke auswählen und ihre Käufe erfassen können. Im Hintergrund verwaltet das System automatisch den Lagerbestand, dokumentiert Zahlungen und erstellt grundlegende Statistiken.

Verantwortliche erhalten so einen klaren Überblick über Bestände und Konsumverhalten, während die alltägliche Verwaltung spürbar vereinfacht wird. Als besonderes Merkmal integriert GetraenkeIO alle relevanten Funktionen in einer kompakten Anwendung, die speziell auf den Einsatz in kleinen Organisationen zugeschnitten ist.

II. Verwandte Arbeiten

Die Verwaltung von Getränken ist in vielen Vereinen eine notwendige Aufgabe. Daher ist es nicht verwunderlich, dass es für eine leichtere Handhabung dieses Vorgangs bereits bestehende Anwendungen gibt, welche eine Vereinfachung versprechen. Bei diesem Vergleich interessieren uns vor allem die Unterschiede, insbesondere jene im Bereich der Monetarisierung. Alle drei hier betrachteten Anwendungen basieren auf einer teilweise zahlungspflichtigen Strategie. So ist es bei der Getränke Zähler App [1], der Getränkeliste App [2] und dem Getränkewart 2.0 [3] jeweils möglich je nach Anbieter mit bis zu 3 bis 10 Nutzern die Anwendung jeweils kostenlos zu nutzen. Sollte die App jedoch von mehr Benutzern verwendet werden, so werden jeweils monatliche oder jährliche Kosten für die Nutzung fällig. Dies ist ein großer Unterschied zu GetraenkeIO, da diese nicht nur kostenfrei nutzbar ist, sondern dank der MIT Lizenz sogar Open Source. Im Gegenteil zu den drei anderen genannten Lösungen ist es bei GetraenkeIO deshalb ebenfalls möglich die Software auf eigener Hardware laufen zu lassen und so die komplette Kontrolle über die eigenen Daten zu erhalten.

III. Architekturziele

A. Kontextabgrenzung

Das System zur Getränkeverwaltung in Vereinsheimen soll von folgenden Nutzern genutzt werden:

1) **Benutzer**: Der reguläre Benutzer des Systems, der in einem Vereinsheim o.ä. ab und zu ein Getränk trinkt und dieses im System buchen möchte, damit er seine Zeche bezahlen kann.
2) **Admin**: (Auch Getränkewart) des Vereinsheims, der sich um den Getränkenachschub kümmert und regelmäßig das Geld für die Getränke kassiert.

Den Benutzern des Systems wird eine Web-Oberfläche zur Interaktion bereitgestellt, die folgende Funktionalität bietet.
Benutzer können:

- sich registrieren und einloggen.
- eine Übersicht über alle verfügbaren Getränke anzeigen.
- ein bestimmtes Getränk buchen.
- ihren Getränkeverlauf einsehen.

Admins können alles was der Benutzer kann und zusätzlich:

- Guthaben von Benutzern aufladen.
- Getränkedaten (Bestand, Preis, verfügbare Getränke,...) verwalten.
- Statistiken über den Getränkeverbrauch einsehen.

Nicht Teil des Systems sind:

- Funktionalität zur Beschaffung von Getränken.
- Verwaltung des tatsächlichen Kassenstandes.

B. Rahmenbedingungen

- Zum Betreiben des Systems muss eine **Internetverbindung** bestehen.
- Es gibt nur **einen Administrator** pro installiertem System, dessen Passwort bei der Installation gesetzt wird.

C. Architekturstil

Als Architekturstil für die Applikation wurde eine dreischichtige Architektur bestehend aus:

- **Frontend**: React Web-App
- **Backend**: Python REST-API
- **Datenbank**: Relationale PostgreSQL Datenbank

gewählt.

IV. Architektur von GetraenkeIO

A. Technologie-Stack

Die GetraenkeIO-Anwendung besteht aus einer dreischichtigen Architektur aus Frontend, Backend und Datenhaltungsschicht. Jede Schicht soll von den anderen abgegrenzt in einem eigenen Container [4] laufen. Das Frontend besteht aus einer React-Anwendung [5]. Als Backend wird eine Python-Anwendung [6] verwendet. Diese benutzt das Web-Framework FastAPI [7] und stellt damit dem Frontend eine RESTful-Schnittstelle [8] zum Datenaustausch zur Verfügung. Zur Datenhaltung wird die relationale Datenbank PostgreSQL [9] genutzt. Die Kommunikation zwischen Datenbank und Backend übernimmt das Framework SQLModel [10], welches mittels Object-Relational Mapping (ORM) die Datenbanktabellen als Python-Objekte zur Verfügung stellt. In Abbildung 1 werden die Architekturbausteine und deren Beziehungen grafisch dargestellt.

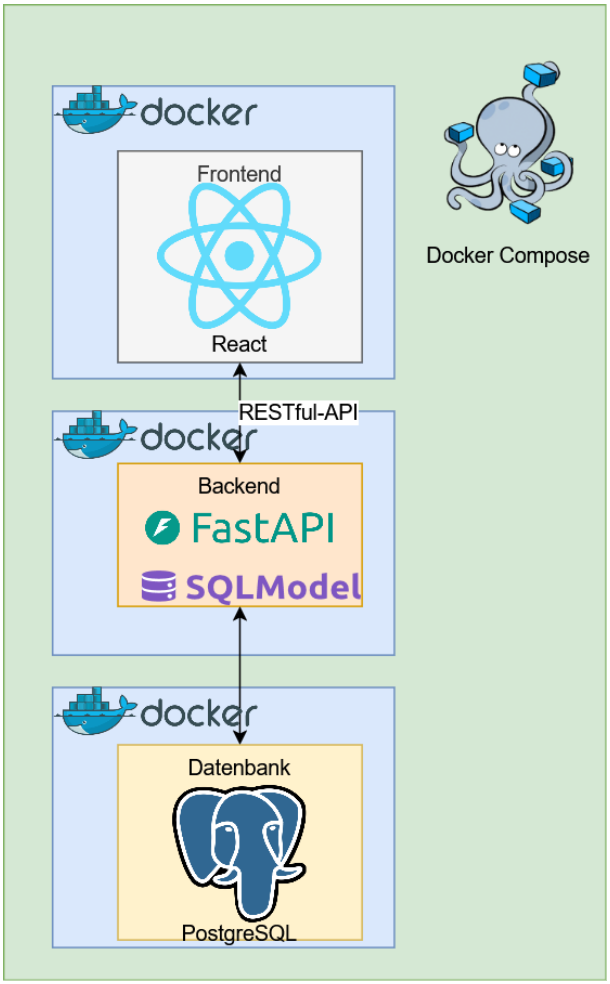

Abbildung 1. Bausteinsicht über den Technologie-Stack und die verwendeten Technologien und deren Zusammenspiel.

B. Frontend

Das Frontend der Anwendung wurde mit dem Framework **React** umgesetzt. Zusammen mit dem Tool **Vite** wird eine schnelle Projektinitialisierung als auch ein sehr performantes Hot Module Replacement bereitgestellt. Dadurch können Änderungen am Code durch einfaches Speichern in Echtzeit im Browser dargestellt werden. Dies führt zu einer deutlich angenehmeren und zeiteffizienteren Entwicklung.
Als Programmiersprache wurde **TypeScript** gewählt. Im Vergleich zu JavaScript bietet TypeScript eine statische Typisierung von Variablen, Funktionen und anderen Komponenten. Dies ermöglicht eine frühzeitige Fehlererkennung - was sich positiv auf die Entwicklungszeit auswirkt - als auch bessere Lesbarkeit des Codes und vereinfachte Wartungen bei größeren und komplexeren Projekten.
Für die bessere Gestaltung der Benutzeroberfläche wird **Inline-CSS** in den React-Komponenten verwendet. Das heißt CSS-Stile liegen nicht in eigenen Dateien sondern werden mithilfe der `style={{}}`-Syntax direkt im JSX/TSX-Code eingebunden. Auf separate Dateien und externe Styling-Frameworks wie **TailwindCSS** wurde bewusst verzichtet, um die Syntax möglichst komponentennah und unkompliziert zu halten.
Die Kommunikation mit dem Backend erfolgt ausschließlich über eine Rest-API. Hierfür wird die Bibliothek **Axios** verwendet. Diese bietet eine einfache Anbindung an diverse Endpunkte. Die empfangenen Daten liegen im JSON-Format vor und werden von den einzelnen Komponenten individuell verarbeitet und angezeigt.
Für die zentrale Zustandsverwaltung wird die Bibliothek **Redux** verwendet. Dadurch lassen sich globale Zustände und Daten einfach effizient verwalten. Im Falle des Logins bzw. der Registierung sorgt ein globaler Zustand dafür, dass nicht eingeloggte Benutzer keinen Zugriff auf geschützte Routen haben, um unerlaubten Zugriff zu vermeiden. Die Nutzerrrollen des angemeldeten Users werden auf die gleiche Art und Weise gespeichert. So hat der Nutzer zudem nur Zugriff auf die für seine Rolle vorgesehenen Seiten und Funktionen. Die Integration erfolgt mithilfe des **react-redux**-Bindings, wodurch der State über das gesamte Frontend hinweg einheitlich zugänglich ist.
Abbildung 2 zeigt die Getränkeübersichtsseite des Frontends.

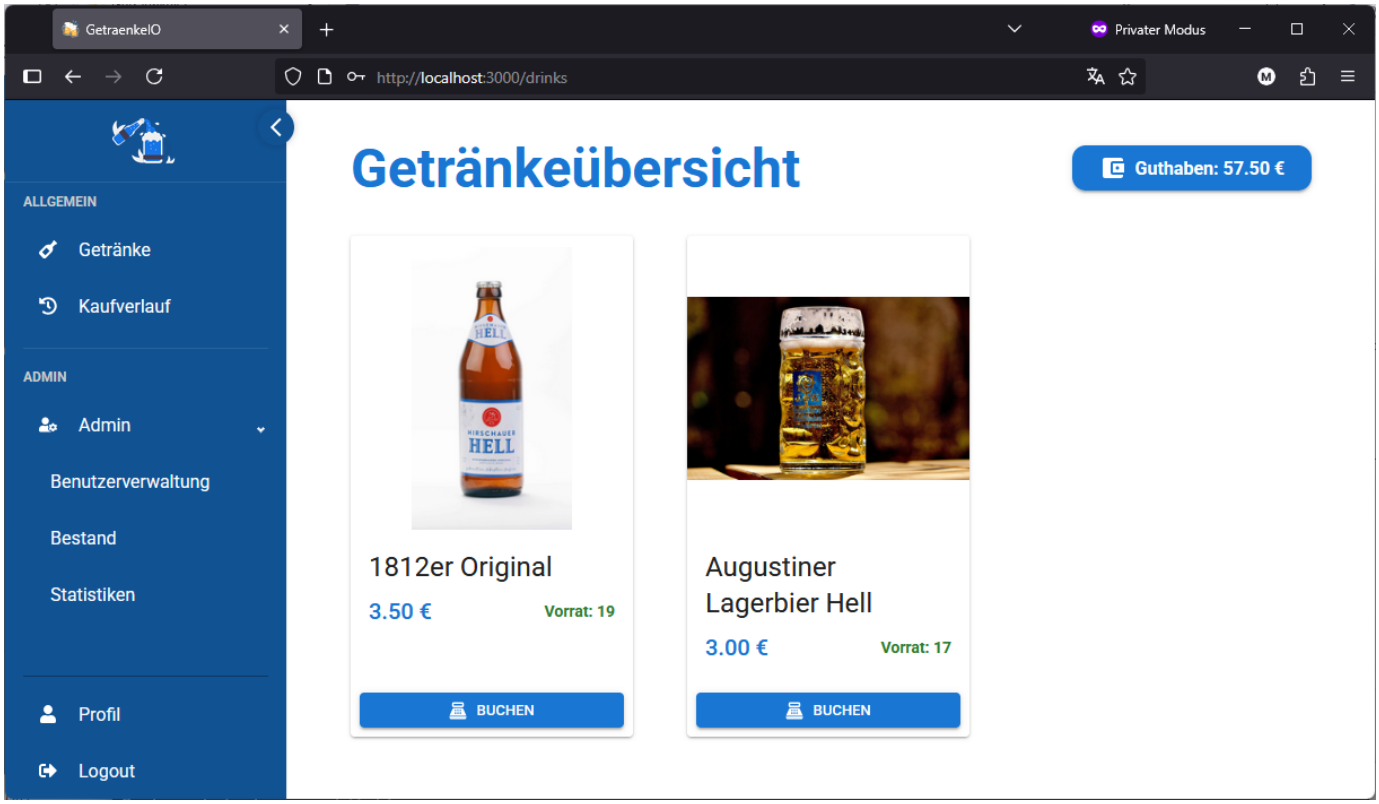

Abbildung 2. Zentrale Seite mit Getränkeübersicht, Guthabenanzeige und der Möglichkeit ein Getränk zu buchen. Aktuell ist ein Admin eingeloggt, daher sind links in der Sidebar auch die Menüs für den Admin sichtbar.

1) Struktur und Komponenten: Der Sourcecode ist in verschiedene Ordner unterteilt. Zusammengehörige Funktionalitäten sind dabei gruppiert, um die Wartbarkeit zu verbessern. Im Folgenden werden die wichtigsten Ordner / Dateien und deren Aufgaben beschrieben.

src/api Enthält die zentrale `axiosInstance` über welche alle Http-Requests abgewickelt werden.

src/components Beinhaltet wiederverwendbare UI-Komponenten, z. B.: `AuthSidebar` `HomeSidebar` und `ProtectedLayout`.

src/features Redux-Slices zur Authentifizierung sowie Methoden, die verschiedenen API-Calls ausführen.

src/models Definiert gemeinsame TypeScript-Interfaces, die über die ganze Anwendung hinweg verwendet werden.

src/pages Eigenständige Seiten wie `Login` und `Register`, die unabhängig vom Layout (ohne Sidebar) fungieren.

src/store Konfiguration des zentralen Redux-Stores für das globale State-Management.

src/views Seiten, die nach dem Einloggen angezeigt werden können und das `ProtectedLayout` sowie die `HomeSidebar` verwenden.

src/App.tsx Hauptkomponente der Anwendung. Hier wird das grundlegende Routing sowie das globale Layout (z. B. Sidebar, Protected Routes) definiert.

src/main.tsx Einstiegspunkt der React-App. Hier wird der Redux-Provider eingerichtet und die Anwendung in das DOM eingebunden.

C. Backend

Das Backend von GetraenkeIO ist als Python Applikation geschrieben. Mithilfe der Frameworks FastAPI wird eine REST-API für das Frontend zur Verfügung gestellt.

FastAPI ist ein Framework, das speziell zum Entwickeln von REST-APIs erstellt wurde und sehr intuitiv zu bedienen ist. Es ist damit möglich mit relativ wenigen Zeilen Code einen funktionierenden Endpunkt, inklusive Validierung der Werte zu erzeugen. Ebenfalls unterstützt es verschiedene Middlewares. Konkret wurde die CORS-Middleware genutzt. Diese ermöglicht es Cross-Origin Resource Sharing (CORS) Anfragen vom Frontend entgegenzunehmen. Eine Besonderheit von FastAPI ist, dass eine API-Dokumentation in Form eines OpenAPI-Dokuments [11] automatisiert erstellt wird. Diese wird auf der Route */docs* als interaktive Oberfläche mittels SwaggerUI [12] zur Verfügung gestellt. Dies erleichtert die Abstimmung zwischen den Entwicklerteams von Front-und Backend.

Für die Verwaltung der Datenbankverbindung wurde das Framework SQLModel gewählt. Dieses wurde vom gleichen Entwickler wie FastAPI entwickelt. Deshalb ist eine Integration dieser beiden Technologien sehr gut möglich. SQLModel ist ein ORM-Framework. Das Datenmodell für die Datenbank wird von SQLModel auf Basis von den im Python-Code erzeugten Datenklassen (Models) erstellt. Ebenfalls müssen für die

gängigen Create, Read, Update und Delete (CRUD) Operationen keine direkten SQL-Abfragen erzeugt werden, es genügt das zugehörige Model zu ändern und an die Datenbanksession weiterzugeben.

1) Struktur und Komponenten: Der Quellcode der Backend-App gliedert sich in mehrere Komponenten (Ordner/Dateien), deren Funktion im Folgenden erklärt wird.

app/api/routes Logik für alle Routen, der REST-Schnittstelle.

app/api/dependencies.py Funktionen/Abhängigkeiten, welche von mehreren Endpunkt-Implementierungen aus *app/api/routes* verwendet werden (Benutzer-Authentifizierung, Zugriff auf Datenbank-Session, Überprüfung ob aktueller Benutzer Admin-Rechte hat).

app/core Sämtliche Logik zur Konfiguration der Anwendung (Datenbank, Passwort-Hash-Funktion, Lesen der Einstellungen aus Umgebungsvariablen).

app/crud Logik zur Interaktion mit der Datenbank (Erstellen, Lesen, Schreiben und Löschen von den Datenmodellen aus */app/models*).

app/models Datenmodelle für die in der Datenbank gespeicherten Daten. Ebenfalls die zugehörigen Datenmodelle, die in den Endpunkten (GET/POST) verwendet werden und die Logik zum validieren der Daten.

app/tests Komponenten-Tests für die Anwendung, welche mithilfe von pytest [13] automatisch bei einem *push* ins Git-Repository ausgeführt werden.

main.py Initialisierung und Start-Up der Anwendung.

requirements.txt Enthält alle genutzten Pakete, um diese automatisiert installieren zu können.

.env.example Beispielhafte .env-Datei, welche die zur Konfiguration nutzbaren Umgebungsvariablen und Beispielwerte für diese enthält.

2) REST-Schnittstelle: Die einzige Schnittstelle zwischen Front-und Backend ist die REST-Schnittstelle. Diese stellt alle relevanten Daten und Informationen für das Frontend im JSON-Format bereit. Im folgenden werden alle wichtigen Endpunkte genauer beschrieben:

GET /users Gibt eine Liste mit Details über alle Benutzer zurück. Ist nur für den Admin nutzbar.

GET /users/me Gibt Informationen über den aktuell eingeloggten Benutzer zurück.

GET /users/{user_name} Gibt Informationen über den Benutzer mit dem Name *user_name* zurück. Ist nur für den Admin nutzbar.

POST /users/ Endpunkt zum Erstellen eines neuen Benutzers bei seiner Registrierung.

POST /users/{user_id}/recharges Endpunkt zum Aufladen des Guthabens eines Benutzers. Ist nur für den Admin nutzbar.

GET /users/{user_id}/recharges Gibt alle Aufladungen eines Benutzers zurück. Ist für den Benutzer selbst und den Admin benutzbar.

GET /drinks Gibt eine Liste mit allen Daten der aktuell vorhandenen Getränke zurück.

GET /drinks/{drink_id} Gibt das Getränk mit der angegebenen Getränke-ID zurück.

POST /drinks Erlaubt es dem Admin ein neues Produkt zu erstellen. Dazu müssen alle Attribute des neuen Getränks in einem JSON-Objekt mitgegeben werden. Ausgenommen davon ist die ID, welche bei der Erstellung des Eintrags in der Getränketabelle automatisch mithilfe eines UUID-Generators erstellt wird.

PUT /drinks/{drink_id} Lässt Administratoren die Attribute der Getränke ändern. Dabei können beliebige Kombinationen an Attributen eines Getränks gleichzeitig geändert werden. Lediglich die ID ist hierüber nicht veränderbar. Zum ändern der gewünschten Attribute werden diese Entsprechend mit neuen Werten als JSON-Objekt an die API übergeben. Ist ein Attribut nicht im Objekt vorhanden, so bleibt dort der Ursprüngliche Wert erhalten.

DELETE /drinks/{drink_id} Lässt Administratoren ein Getränk löschen. Dabei wird dem Nutzer nach erfolgreicher Löschung eine Kopie des Objekts zurückgegeben.

POST /transactions Ermöglicht es Nutzern ein Getränk zu kaufen. Dabei wird das Guthaben des Nutzers um den Gesamtpreis verringert und die Stückzahl des Getränks entsprechend angepasst. Zudem wird die Transaktion in einer separaten Tabelle protokolliert.

GET /transactions Gibt einem Admin alle getätigten Transaktionen als Liste zurück.

GET /transactions/me Gibt Nutzern alle von ihnen durchgeführte Transaktionen als Liste zurück.

D. Datenhaltung

Zur Datenhaltung in der Produktionsumgebung wird eine relationale PostgreSQL-Datenbank verwendet. Die Kommunikation mit dieser findet über das SQLModel-Framework statt. Da ein ORM-Framework verwendet wird, könnte die Datenbank relativ einfach gegen alle anderen von diesem Framework unterstützen Datenbanken ausgetauscht werden. In der Anfangsphase der Entwicklung wurde diese Möglichkeit genutzt, um SQLite [14] als Entwicklungsdatenbank verwenden zu können und durch die einfache Konfiguration schnell ein lauffähiges System erzeugen zu können.

Die Daten der Getränkeverwaltung werden in unserer Datenbank, wie in Abbildung 3 beschrieben, auf fünf verschiedene Tabellen aufgeteilt. So werden aktuelle Nutzer und Getränke in jeweils einzelnen Tabellen abgespeichert. Abseits davon sind die weiteren Tabellen zur Überwachung der getätigten Aktionen verantwortlich. Somit wird das Aufladen von Guthaben in der recharge Tabelle und jeder Kauf in der transaction Tabelle abgespeichert. Um jedoch sowohl die ursprünglichen Preise, als auch weitere möglicherweise veränderte Attribute der Getränke für spätere Analysezwecke korrekt vorliegen zu haben, wird für jedes veränderte Getränk, das gekauft wird, ebenfalls ein Eintrag in der drinkhistory Tabelle erstellt.

V. Herausforderungen & Probleme

Im Laufe der Entwicklung unseres Projekts sind uns einige Probleme und Herausforderungen begegnet, aus denen wir

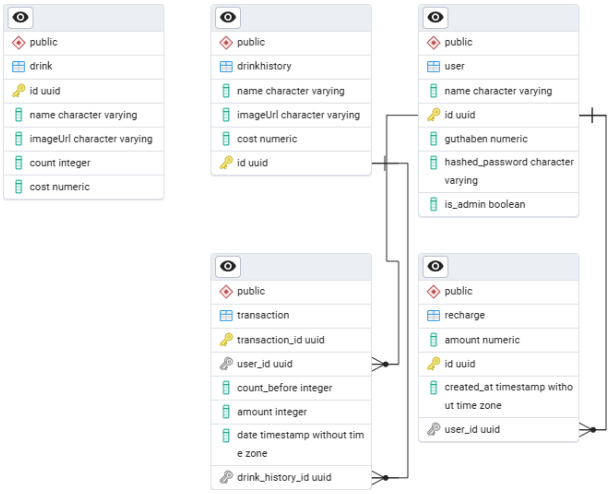

Abbildung 3. ER-Diagramm zur PostgreSQL Datenbank

lernen konnten.

1. Authentifizierungsfehler bei API-Anfragen

Ein technisches Problem war, dass die Getränke aus dem Backend nicht geladen wurden. Grund dafür war, dass die Anfragen teilweise ohne Authentifizierung abgeschickt wurden. Das lag daran, dass verschiedene HTTP-Bibliotheken verwendet wurden (einmal Axios, einmal fetch), und nicht immer die richtigen Zugangsdaten mitgesendet wurden. Nachdem wir das mit dem jeweiligen Teammitglied besprochen haben, wurde der Fehler schnell behoben.

2. Fehlermeldungen beim Erstellen des Frontend-Docker-Containers

Beim Docker-Build des Frontends sind wir auf Probleme gestoßen, weil der TypeScript-Compiler auf ungenutzte Importe gestoßen ist. Diese waren im Code zwar nicht relevant, haben aber den Buildprozess gestört. Nachdem diese Importe bereinigt wurden, konnte das Frontend erfolgreich als Container gebaut werden.

3. Zugriffsrechte im Frontend nicht korrekt umgesetzt

Ein weiterer Fehler betraf die Zugriffsbeschränkung für Admin-Funktionen. Es war möglich, auch ohne Admin-Rechte gewisse Admin-Routen im Frontend aufzurufen, solange man eingeloggt war. Das wurde intern schnell erkannt und korrigiert.

4. CORS-Konfiguration unvollständig

Bei der Kommunikation zwischen Frontend und Backend kam es zu Problemen mit CORS (Cross-Origin Resource Sharing). Eine benötigte Origin war nicht in der Konfiguration des Servers freigegeben, was zu abgelehnten Anfragen führte. Dieser Fehler wurde durch Ergänzen der fehlenden Origin behoben.

VI. Fazit und Ausblick

Wir haben für unser Empfinden eine gute Lösung für eine webbasierte Getränkeverwaltung für Vereine oder auch andere Einrichtungen geschaffen. Unsere gesteckten Ziele wurden weitestgehend erreicht. Sowohl unsere *Muss-* als auch *Soll-Anforderungen* wurden vollumfänglich erfüllt. Die traditionellen Probleme der manuellen Verwaltung durch Strichlisten und Excel-Tabellen werden hierbei durch unser System abgelöst und bieten stattdessen eine automatisierte digitale Lösung.

Ein Hauptaugenmerk bei unserem Projekt lag auf der Benutzerfreundlichkeit. Vereinsmitglieder können ohne administrative Unterstützung eigenständig Getränke buchen. Administratoren erhalten dagegen einen umfangreichen Überblick über Getränkebestände, Konsumverhalten und Umsatz durch das integrierte Statistik- und Verwaltungssystem. Die von uns gewählte dreischichtige Architektur mit React im Frontend, FastAPI im Backend und einer PostgreSQL-Datenbank hat sich als robust und skalierbar erwiesen.

Eine Möglichkeit, *GetraenkeIO* weiter zu optimieren, wäre die Integration von Zahlungsdienstleistern wie PayPal, die eine automatisierte Guthabenaufladung ermöglichen würde. Benutzer könnten ihr Guthaben selbstständig ohne Adminunterstützung aufladen. Dies würde den Verwaltungsaufwand für Administratoren deutlich reduzieren und die Kassenlage des Vereins oder der Einrichtung verbessern, da Zahlungen direkt in Echtzeit eingingen.

Literatur

[1] Matthias Aigner. *Getränke Zähler App*. URL: https://drinkscounter.com/de/getraenke-zaehler-app.html.
[2] iqmeta GmbH. *Getränkeliste App*. URL: https://iqmeta.com/gliste/.
[3] sparse creations GmbH. *Getränkewart 2.0*. URL: https://getraenkewart.com/.
[4] IBM. *Was sind Container?* 2025. URL: https://www.ibm.com/de-de/topics/containers.
[5] Facebook. *React*. [Online]. URL: https://react.dev/.
[6] Guido van Rossum. *Python*. [Online]. URL: https://www.python.org.
[7] Sebastián Ramírez. *FastAPI*. [Online]. URL: https://fastapi.tiangolo.com/.
[8] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
[9] Andrew Yu und Jolly Chen. *PostgreSQL: Open Source Relational Database*. [Online]. URL: https://www.postgresql.org/.
[10] Sebastián Ramírez. *SQLModel*. 2025. URL: https://sqlmodel.tiangolo.com/.
[11] The Linux Foundation. *OpenAPI-Spezifikation*. 2025. URL: https://spec.openapis.org/oas/latest.html.
[12] SMARTBEAR. *Swagger*. 2025. URL: https://swagger.io/.
[13] Holger Krekel und pytest Entwicklungteam. *Pytest*. [Online]. URL: https://pytest.org/.
[14] Richard Hipp. *SQLite: Features*. [Online]. URL: https://sqlite.org/features.html.