Technologieauswahl

In diesem Kapitel wird der Entwurf des Systems beschrieben. Ziel des Kapitels ist es, einen detaillierten Überblick über die Systemarchitektur und die Technologieauswahl zu geben sowie die einzelnen Komponenten des Systems darzustellen. Es wird erläutert, warum bestimmte Technologien und Architekturen gewählt wurden und wie die verschiedenen Komponenten des Systems miteinander interagieren.

Zunächst wird die Auswahl der verwendeten Technologien und Frameworks begründet. Anschließend wird die Systemarchitektur detailliert beschrieben, gefolgt vom Datenbankentwurf und dem API-Design. Ein weiterer Abschnitt widmet sich der Implementierung der Authentifizierung und Benutzerverwaltung, gefolgt von der Datenvalidierung. Die Backend-Logik wird ebenfalls im Detail erläutert.

Bun

Bun ist eine JavaScript-Runtime Umgebung für den Server, die anders als Node.js oder Deno nicht auf der V8-Engine basiert, sondern auf einer eigenen JavaScript-Engine, welche mithilfe von Apples WebKit Engine implementiert wurde. Bun wurde zudem in einer “low-level general” Programmiersprache namens Zig geschrieben, welche von Rust und C inspiriert ist. Die Entscheidung für Bun fiel aufgrund der hohen Performance und Sicherheit, die durch die Verwendung von Zig und der WebKit-Engine gewährleistet wird. Bun ermöglicht es, serverseitige Anwendungen in JavaScript zu entwickeln und auszuführen. Bun wurde von Jarred Summer entwickelt und ist eine Open-Source-Software, die unter der MIT-Lizenz veröffentlicht wird. Die erste offizielle Version von Bun (Bun 1.0) wurde im September 2023 veröffentlicht.

Funktionen von Bun

Bun bietet eine Reihe von Funktionen, die es zu einer Plattform für die Entwicklung von serverseitigen Anwendungen machen. Dazu gehören:

- Kompatibilität mit Node.js
- Hohe Laufleistung und geringer Speicherverbrauch
- Vereinfachte Modulverwaltung
- TypeScript-Unterstützung
- Web-Standard-APIs
- JSX-Unterstützung
- Watch-Modus für automatisches Neuladen von Änderungen
- Cross-Plattform-Unterstützung

Bun vs. Node.js

Anders als Node.js ist bun nicht auf npm angewiesen und benötigt keine externen Abhängigkeiten zur Ausführung. Stattdessen wird eine integrierte Standardbibliothek verwendet, die Funktionen wie HTTP-Server, Dateisystemzugriff und Netzwerkkommunikation bereitstellt. Dies macht die Entwicklung und Bereitstellung von Anwendungen mit Bun einfacher und sicherer. Bun basiert zudem anders als Node.js nicht auf der von Google entwickelten V8-Engine, sondern auf einer Erweiterung von JavaScriptCore, die von Apple entwickelt und bereitgestellt wird. JSC priorisiert schnellere Startzeiten und geringeren Speicherverbrauch, was zu einer etwas langsameren Ausführungsgeschwindigkeit führt. V8 priorisiert hingegen die Ausführungsgeschwindigkeit mit mehr Runtime-Optimierungen, was zu einem höheren Speicherverbrauch führen kann. Das führt dazu, dass Bun bis zu 4xmal so schnell startet als Node.js

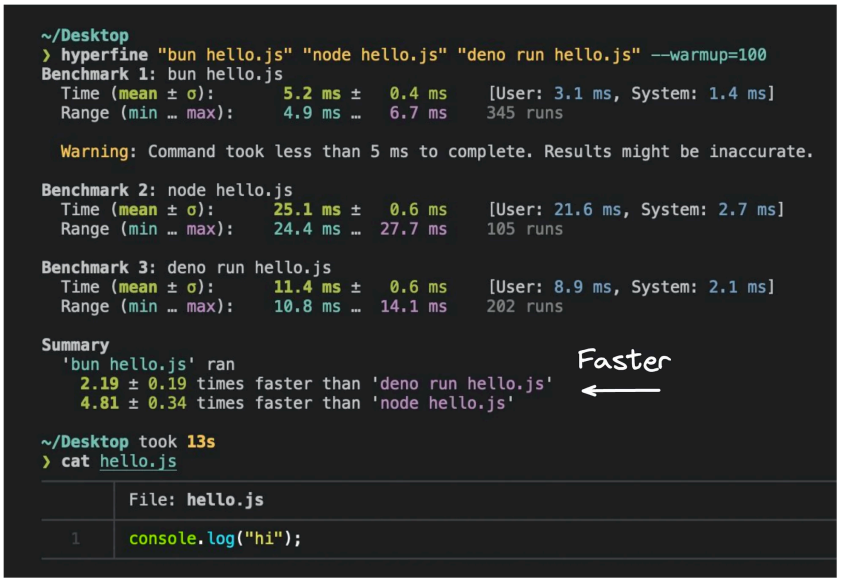

Figure 1: Bun vs. Node.js - Startzeitvergleich, Quelle: Builder.io

Die Benchmark-Ergebnisse, welche in Abbildung 6 gezeigt werden, zeigen eine Verbesserung von mehr als siebzehnmal so schnell wie übliche Paketmanager.

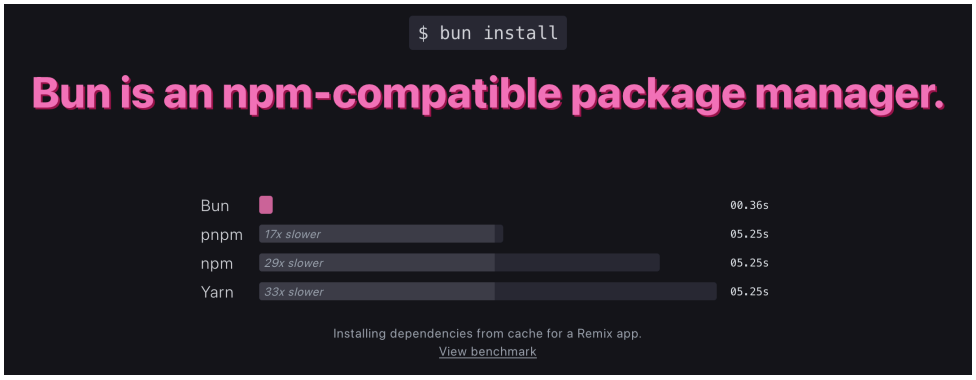

Figure 2: Bun vs. andere Paketmanager - Benchmark-Ergebnisse, Quelle: Bun

Während Node.js eine gute Runtime-Umgebung für JavaScript ist, werden TypeScript Dateien in Node.js nicht direkt unterstützt. TypeScript-Dateien müssen zuerst in JavaScript-Dateien kompiliert werden, bevor sie in Node.js ausgeführt werden können. Bun hingegen unterstützt TypeScript-Dateien direkt, was die Entwicklung von serverseitigen Anwendungen in TypeScript vereinfacht. TypeScript-Dateien können direkt mit dem Befehl bun “Dateiname.ts” ausgeführt werden.

Deshalb wurde Bun als Server-Runtime für die Entwicklung des Webshops gewählt, da es eine hohe Performance, Sicherheit und TypeScript-Unterstützung bietet und die Entwicklung von serverseitigen Anwendungen in JavaScript und TypeScript vereinfacht.

Hono

Hono ist einfaches und ultraschnelles Web-Framework, welches auf jeder JavaScript-Runtime-Umgebung läuft. Entwickelt wurde Hono von Yusuke Wada und ist eine Open-Source-Software, die unter der MIT-Lizenz veröffentlicht wird. Hono wurde speziell für die Entwicklung von Webanwendungen und APIs entwickelt und bietet eine Reihe von Funktionen, die es zu einer leistungsstarken Plattform für die Entwicklung von Webanwendungen machen.

Vorteile von Hono
Die Entscheidung für Hono als Web-Framework wurde aufgrund mehrerer Schlüsselfaktoren getroffen:

- **Ultraschnell und effizient**: Der Router "RegExpRouter" ist besonders schnell und arbeitet nicht mit linearen Schleifen, was eine schnelle und effiziente Routenauflösung ermöglicht. Dies macht Hono ideal für Anwendungen, die eine hohe Geschwindigkeit und geringe Latenz erfordern.
- **Leichtgewichtig und modular**: Hono ist äußerst leichtgewichtig und hat keine externen Abhängigkeiten. Mit dem hono/tiny-Preset beträgt die Größe von Hono weniger als 14 kB, was im Vergleich zu anderen Frameworks sehr kompakt ist. Trotz seiner geringen Größe bietet Hono eine Vielzahl von Middleware- und Hilfsfunktionen, die es einfach machen, leistungsstarke Anwendungen zu entwickeln.
- **Multi-Runtime-Unterstützung**: Hono ist äußerst vielseitig und kann auf einer Vielzahl von Plattformen eingesetzt werden, darunter Cloudflare Workers, Fastly Compute, Deno, Bun, AWS Lambda und Node.js. Dadurch ist es möglich, die gleiche Codebasis auf verschiedenen Plattformen zu verwenden, was die Entwicklung und Wartung von Anwendungen vereinfacht.
- **Middleware inklusive**: Hono bietet eine umfangreiche Sammlung von Middleware, benutzerdefinierten Middleware und Hilfsfunktionen, die es Entwicklern ermöglichen, weniger Code zu schreiben und mehr zu erreichen. Von der Basic Authentication bis zur GraphQL Server-Unterstützung bietet Hono alles, was für die Entwicklung leistungsstarker Webanwendungen erforderlich ist. Zudem auch einen kleinen Validator für die Datenvalidierung, um die Datenintegrität zu gewährleisten.

Hono vs. Express.js
Im Vergleich zu Express.js, einem der beliebtesten Web-Frameworks für Node.js, bietet Hono eine Reihe von Vorteilen:

Vorteile von Hono:

- Mikroservices-Architektur: Hono ist speziell für Mikroservices-Architekturen ausgelegt, was die Skalierbarkeit und Modularität von Anwendungen erleichtert.
- Leistung und Skalierbarkeit: Hono bietet Leistungsbenchmarks und effiziente Anfrageverarbeitung, was besonders für hochskalierbare Anwendungen von Vorteil ist.
- Eingebaute WebSocket-Unterstützung: Hono bietet WebSocket-Unterstützung für die Implementierung von Echtzeitfunktionen an.
- TypeScript-Unterstützung: Hono unterstützt TypeScript nativ, was für Typsicherheit und verbesserte Entwicklerwerkzeuge sorgt.
- Aktive Community-Wartung: Hono wird von einer aktiven Entwicklergemeinschaft gepflegt, was regelmäßige Updates und Verbesserungen gewährleistet.

Deshalb wurde Hono als Web-Framework für die Entwicklung des Webshops gewählt, da es vorallem mit Kombination von Bun ultraschnell, effizient, leichtgewichtig, modular und vielseitig ist.

Framework	Runtime	Durchschnitt	Ping	Query	Body
Hono	bun	184,966.48	234,593.57	185,108.2	135,197.67
Hono	Node	42,699.317	60,797.19	56,645.8	10,654.96
Express	node	16,461.68	17,656.74	16,615.32	15,112.98

Die Ergebnisse sind in req/s gemessen

TypeScript

TypeScript ist eine von Microsoft entwickelte Programmiersprache, die eine strikte Typisierung für JavaScript bietet. Sie erweitert JavaScript um statische Typisierung, Klassen, Interfaces und Module, was die Entwicklung von großen und komplexen Anwendungen erleichtert. TypeScript wird zu JavaScript kompiliert und kann in jedem Browser und auf jedem Betriebssystem ausgeführt werden. Diese Eigenschaften tragen zur Steigerung der Produktivität von Entwicklern und zur Verbesserung der Codequalität bei. TypeScript erweitert JavaScript um zusätzliche Features wie Interfaces, Enums, Generics und Module. Die Vorteile von TypeScript sind:

1. Statische Typisierung TypeScript bietet eine statische Typisierung, die es Entwicklern ermöglicht, Typfehler bereits zur Entwicklungszeit zu erkennen und zu beheben. Dies führt zu weniger Fehlern und einer höheren Codequalität.
2. Moderne JavaScript-Features TypeScript unterstützt moderne JavaScript-Features wie Klassen, Interfaces, Generics und Module, die die Entwicklung von großen und komplexen Anwendungen erleichtern. Darüber hinaus unterstützt TypeScript asynchrone Programmierung, was die Handhabung asynchroner Operationen vereinfacht.

Aufgrund dieser Vorteile wurde TypeScript als primäre Programmiersprache für die Entwicklung des Webshops gewählt.

Zod

Zod ist eine TypeScript-First-Schema-Validierungs-Bibliothek, die es Entwicklern ermöglicht, Datenstrukturen zu definieren und zu validieren. Zod bietet eine einfache und deklarative API zum Definieren von Schemas und zur Validierung von Daten. Zod ist speziell für TypeScript entwickelt und bietet eine nahtlose Integration mit der Sprache. Zod unterstützt eine Vielzahl von Datentypen, Validierungsregeln und Transformationen, die es Entwicklern ermöglichen, komplexe Datenstrukturen zu definieren und zu validieren.

Vorteile von Zod

1. Typsicherheit Zod bietet Typsicherheit auf der Ebene der Datenvalidierung, was es Entwicklern ermöglicht, Datenstrukturen zu definieren und zu validieren, ohne zusätzlichen Code schreiben zu müssen. Dies führt zu weniger Fehlern und einer höheren Codequalität.
2. Einfache API Zod bietet eine einfache und deklarative API zum Definieren von Schemas und zur Validierung von Daten. Die API ist intuitiv und leicht verständlich, was die Entwicklung von Datenvalidierungslogik vereinfacht.
3. Integration mit Frameworks Zod bietet eine nahtlose Integration mit verschiedenen Frameworks und Bibliotheken, was es zu einer vielseitigen Lösung für die Datenvalidierung macht. Besonderns in Kombination mit Backend-Frameworks wie Hono ist Zod eine gute Wahl für die Datenvalidierung.
4. Leistungsfähigkeit Zod zeichnet sich durch hohe Leistung und geringen Overhead aus, was die Validierung großer Datenmengen effizient macht, ohne die Anwendungsleistung zu beeinträchtigen.

Anwendung im Webshop

Zod wird im Webshop für die Validierung von Benutzereingaben, API-Anfragen und Datenbankantworten verwendet. Durch die Verwendung von Zod wird sichergestellt, dass die Datenintegrität gewährleistet ist und sichergestellt wird, dass die vom Benutzer übermittelten Daten

den erwarteten Formaten und Typen entsprechen, bevor diese in die Datenbank gespeichert werden. Zudem erleichert Zod die Fehlerbehandlung und die Rückmeldung an den Benutzer im Falle von Validierungsfehlern, anhand von detaillierten Fehlermeldungen.

Kinde Auth

Kinde Auth ist eine Authentifizierung- und Benutzerverwaltungslösung, die speziell für SaaS-Anwendungen entwickelt wurde. Es bietet eine Vielzahl von Funktionen, die es Entwicklern ermöglichen, Benutzerkonten zu verwalten, Authentifizierung zu implementieren und Zugriffsrechte zu steuern. Dabei wird drauf geachtet, dass höchste Sicherheitsstandards eingehalten werden, um die Benutzerdaten zu schützen. Die Integration von Kinde Auth im Webshop ermöglicht es, Benutzerkonten zu erstellen, sich anzumelden und Zugriffsrechte zu verwalten. Dadurch wird eine robuste und flexible Authentifizierunginfrastruktur bereitgestellt, die den Anforderungen des Webshops entspricht, da im Webshop mit besonders sensiblen Daten gearbeitet wird.

Typen von Authentifizierung bei Kinde Auth

Kinde´s Authentifizierungssystem fokussiert sich hauptsächlich auf die Sicherheit und eine einfache Handhabung für die Benutzer. Es bietet verschiedene Arten von Authentifizierungsmethoden, die je nach Anwendungsfall ausgewählt werden können. Dazu gehören:

- Passwordless
- Single Sign-On (SSO)
- Multi-Faktor-Authentifizierung (MFA)
- Social Login

Für den Webshop wird die Passwordless Authentifizierung implementiert mit der möglichkeit sich über einen Google-Account anzumelden, um die Benutzerfreundlichkeit zu erhöhen und die Sicherheit zu gewährleisten.

Passwordlose Authentifizierung

Zunächst stellt sich die Frage, warum eine Passwordlose Authentifizierung gewählt wurde und was die Vorteile sind. Da es oft vorkommt, dass man sein eigenes Passwort vergisst oder es zu unsicher ist, verwenden Menschen, genauer gesagt zwei-drittel der Menschen, das gleiche Passwort für mehrere Dienste. Dies führt zu einer erhöhten Sicherheitslücke, da bei einem Datenleck die Zugangsdaten für mehrere Dienste kompromittiert sind. Die Passwordlose Authentifizierung bietet eine sichere und benutzerfreundliche Alternative zur herkömmlichen Passwortauthentifizierung. Dadurch fällt die Notwendigkeit für die Benutzer, sich ein Passwort zu merken, weg und erhöht die Sicherheit, da keine Passwörter im Klartext übertragen werden. Stattdessen wird ein einmaliger Token generiert und an den Benutzer gesendet, der zur Authentifizierung verwendet wird. Dieser Token ist nur für eine begrenzte Zeit gültig und kann nicht wiederverwendet werden, was die Sicherheit erhöht.

Verringerung von Sicherheitsrisiken

Die traditionelle Authentifizierung mittels Benutzername und Passwort birgt erhebliche Risiken für Benutzer und Unternehmen in Bezug auf Hackerangriffe und Cyberkriminalität. Laut dem 2022 Data Breach Investigations Report der amerikanischen Firma Verizon entfallen etwa 84 % aller Datenverletzungen auf Schwachstellen bei Anmeldeinformationen. Unsichere Passwörter, wie „password1234“, machen es Hackern leicht, über Brute-Force-Angriffe Zugang zu erhalten. Noch problematischer wird es, wenn Nutzer dieselben Anmeldedaten auf verschiedenen Websites verwenden – ein einziger Datenverstoß kann dann mehrere Systeme gefährden.

Auch menschliche Fehler spielen eine Rolle: Das Notieren von Passwörtern auf Haftnotizen oder das unsichere Weitergeben von Anmeldedaten kann sowohl interne als auch externe Angriffe

begünstigen. Diese Risiken lassen sich durch passwortlose Authentifizierungsmethoden vollständig vermeiden, da sie die beschriebenen Schwachstellen eliminieren.

SQLite & Drizzle ORM

SQLite ist eine relationale Datenbank, die auf SQL basiert und als Datei gespeichert wird. SQLite ist eine leichte und schnelle Datenbank, die keine Server-Infrastruktur erfordert und einfach in Anwendungen integriert werden kann. SQLite ist eine gute Wahl für Anwendungen, die eine lokale Datenbank benötigen und keine komplexe Server-Infrastruktur benötigen. SQLite unterstützt die meisten SQL-Features und bietet eine gute Performance für kleine bis mittlere Datenmengen. Für den Webshop wird SQLite als Datenbank verwendet, da der Webshop ein Prototyp ist und keine komplexe Server-Infrastruktur benötigt. Für einen Produktivbetrieb könnte die Datenbank auf eine relationale Datenbank wie PostgreSQL oder MySQL migriert werden.

Drizzle ORM ist ein Object-Relational Mapping (ORM) Framework für SQLite, das es Entwicklern ermöglicht, Datenbankoperationen in JavaScript durchzuführen. Drizzle ORM bietet eine einfache und deklarative API zum Definieren von Datenmodellen und zur Ausführung von Datenbankabfragen. Drizzle ORM unterstützt die meisten SQL-Features und bietet eine gute Performance für Datenbankoperationen. Drizzle ORM wird im Webshop für die Datenbankoperationen verwendet, um Migrationsdatein zu erstellen und diese zu verfolgen, falls Änderungen getätigt wurden. Zudem wird Drizzle ORM für die Datenbankabfragen und die Datenmodellierung verwendet, um die Datenbankoperationen zu vereinfachen und zu beschleunigen.

Systemarchitektur

Die Systemarchitektur des Webshops setzt sich aus mehreren aufeinander abgestimmten Komponenten zusammen. Diese Komponenten sind so konzipiert, dass sie eine robuste und skalierbare Plattform für die Entwicklung und den Betrieb eines Webshops bieten. Die Hauptkomponenten der Architektur umfassen den Server, das Front-End, die Datenbank und die Authentifizierung.

Darstellung der einzelnen Komponenten und deren Interaktionen

Das beigefügte Diagramm zeigt die Interaktionen zwischen den einzelnen Komponenten des Systems:

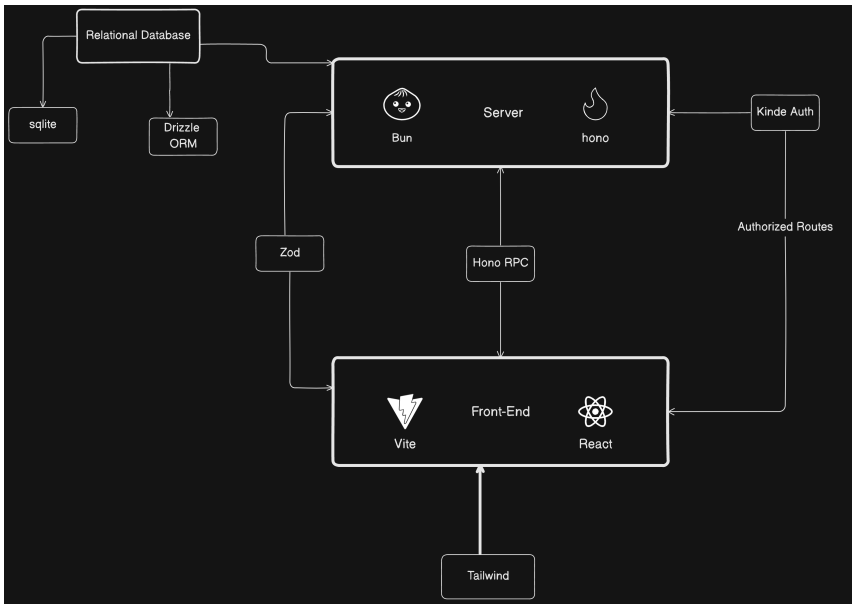

Figure 3: Systemarchitektur des Webshops, Quelle: Eigene Darstellung

- **Relationale Datenbank**: Verbunden mit dem Server über Drizzle ORM.
- **Server**: Implementiert mit Bun und Hono, verwaltet die Kommunikation mit der Datenbank, führt die API-Logik aus und kümmert sich um die Authentifizierung über Kinde Auth.
- **Front-End**: Entwickelt mit Vite und React, kommuniziert mit dem Server über Hono RPC und nutzt Zod zur Datenvalidierung. Tailwind CSS wird für das Styling verwendet.
- **Authentifizierung**: Kinde Auth verwaltet die Benutzeranmeldung und Zugriffsrechte und integriert sich nahtlos in die Serverlogik.

Diese Architektur ermöglicht eine klare Trennung der Verantwortlichkeiten, was die Wartung und Skalierbarkeit des Systems erleichtert. Die Technologieauswahl zielt auf optimale Leistung, Sicherheit und Entwicklerfreundlichkeit ab. Bun und Hono wurden aufgrund ihrer hohen Performance und Effizienz als Server-Frameworks gewählt. SQLite und Drizzle ORM bieten eine einfache, performante Datenhaltung. React und Vite sorgen für ein modernes, flexibles Front-End, Zod für zuverlässige Datenvalidierung und Kinde Auth für sichere Authentifizierung.

Datenbankentwurf

Nachdem die Systemarchitektur beschrieben wurde und die Auswahl der Technologien bekannt sind, wird im folgenden Abschnitt der Datenbankentwurf detailliert erläutert. Der Datenbankentwurf umfasst die Definition der Datenmodelle, die Beziehungen zwischen den Datenmodellen und die Struktur der Datenbanktabellen.

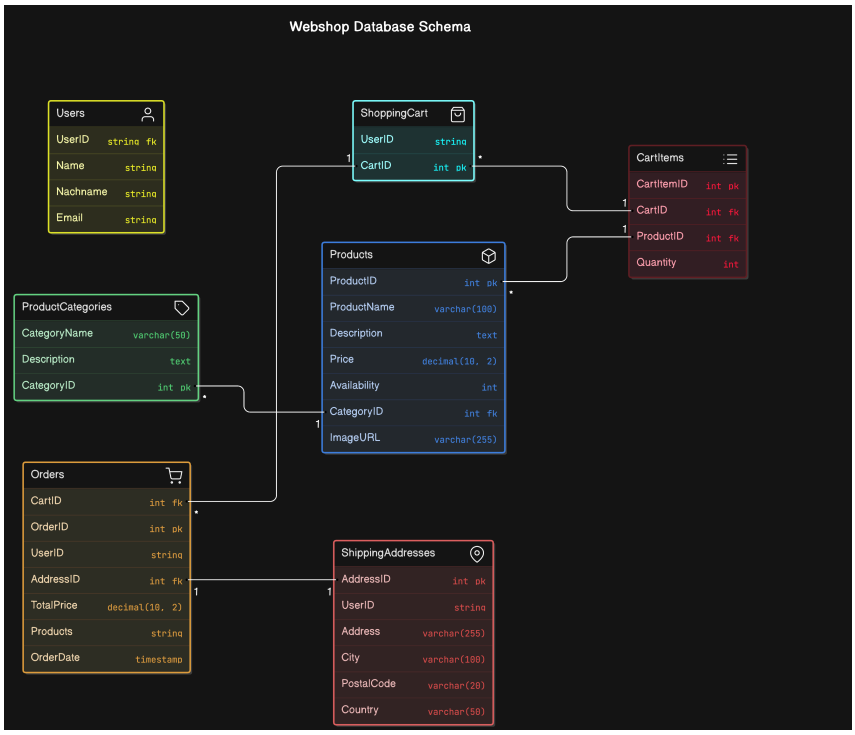

Figure 4: Datenbankmodell des Webshops, Quelle: Eigene Darstellung

Die Tabelle “productCategories” enthält Informationen zu den Produktkategorien, darunter Name und Beschreibung.

Die Tabelle “products” enthält Informationen zu den Produkten im Webshop, darunter Name, Beschreibung, Preis, Anzahl und Kategorie.

Jeder Benutzer hat einen Warenkorb welcher erstellt wird sobald ein Produkt hinzugefügt wird.

Die Tabelle “carts” enthält Informationen zu den Warenkörben, darunter Benutzer-ID und eine Cart-ID.

Um Produkte zu zuordnen wird die Tabelle “cartItems” verwendet, welche Informationen zu den Produkten im Warenkorb enthält, darunter Produkt-ID, Cart-ID und die Anzahl des jeweiligen Produktes.

Nachdem der Benutzer die Bestellung abschicken möchte, wird dieser aufgefordert seine Lieferadresse anzugeben.

Die Tabelle “shippingAddresses” enthält Informationen zu den Lieferadressen, darunter Benutzer-ID, Straße, PLZ und Stadt.

Nach erfolgreicher Bestellung wird die Bestellung in der Tabelle “orders” gespeichert, welche Informationen zu den Bestellungen im Webshop enthält, darunter die Benutzer-ID, die Cart-ID, die Address-ID, den Gesamtpreis, die Produkte welche aus dem Warenkorb bestellt wurden und das Datum der Bestellung.

Dabei wurde die “users” Tabelle absichtlich außen vor gelassen, da diese von Kinde Auth verwaltet wird. Im Webshop wird ausschließlich die user_id abgerufen und gespeichert um die Beziehung zu den anderen Tabellen herzustellen. Damit wird sichergestellt, dass die jeweilige Bestellung dem richtigen Benutzer zugeordnet wird.

SQL-Skripte

Ein Beispiel für ein SQL-Skript zur Erstellung der Tabellen im Webshop mithilfe von SQLite und Drizzle ORM:

```
// Import der benötigten Funktionen aus Drizzle ORM
import { sqliteTable,text,integer,numeric,blob,index } from "drizzle-orm/sqlite-
core";
import { productCategories } from "./productCategories.ts";

// Erstellung der Tabelle "products" mit den jeweiligen Attributen
export const products = sqliteTable(
  "products",
  {
    productID: integer("productID").primaryKey(),
    productName: text("productName").notNull(),
    description: text("description"),
    price: numeric("price").notNull(),
    availability: integer("availability"),
    categoryID: integer("categoryID")
      .references(() => productCategories.categoryID)
      .notNull(),
    image: blob("image"),
  },
  (products) => {
    return {
      categoryIdIndex: index("categoryIdIndex").on(products.categoryID),
    };
  }
);
```

Listing 1: Beispiel für ein SQL-Skript zur Erstellung der Tabelle “products”, Quelle: Eigene Darstellung

Das SQL-Skript erstellt die Tabelle “products” mit den jeweiligen Attributen, welche im Datenbankmodell definiert wurden.

Um die Tabellen zu erstellen und diese in die Datenbank zu speichern, wird das SQL-Skript in SQLite ausgeführt. Drizzle ORM übernimmt die Kommunikation mit der Datenbank und ermöglicht es, Datenbankoperationen in JavaScript durchzuführen. Damit eine SQLite-Datenbank erstellt wird, wird eine “index.ts” Datei erstellt, welche die Datenbankverbindung aufbaut.

```
// index.ts
import { drizzle } from "drizzle-orm/bun-sqlite";
import { Database } from "bun:sqlite";

const sqlite = new Database(process.env.DATABASE_URL!);
export const db = drizzle(sqlite);
```

Listing 2: Erstellung einer SQLite-Datenbankverbindung mit Drizzle ORM, Quelle: Eigene Darstellung

Um die SQL-Skripte zu erstellen, wird ein Drizzle Ordner erstellt, in welchem die SQL-Skripte abgelegt werden. Diese Skripte werden dann von Drizzle ORM ausgeführt, um die Datenbanktabellen zu erstellen und zu verwalten. In der Konsole wird zunächst der Befehl “bun drizzle-kit generate” ausgeführt, um die Migrationsdateien zu erstellen. Anschließend wird die Datei “migrate.ts” mithilfe des Befehls “bun migrate.ts” ausgeführt, um die Migrationsdateien auszuführen und die Datenbanktabellen zu erstellen.

```ts
// migrations.ts
import { migrate } from "drizzle-orm/bun-sqlite/migrator";
import { db } from "./db/index.ts";

async function runMigrations() {
  await migrate(db, { migrationsFolder: "./drizzle" });
  console.log("Migration completed");
}

runMigrations().catch(console.error);
```

Listing 3: Ausführung der Migrationsdateien mit Drizzle ORM, Quelle: Eigene Darstellung

Alle SQL-Skripte und Migrationsdateien werden im Drizzle Ordner abgelegt und von Drizzle ORM verwaltet. Dadurch wird sichergestellt, dass die Datenbanktabellen korrekt erstellt und verwaltet werden und Änderungen an den Datenmodellen nachvollziehbar sind. Zudem werden alle Änderungen an den Datenbanktabellen in den Migrationsdateien protokolliert, um die Datenintegrität zu gewährleisten.

API-Design

Das API-Design des Webshops umfasst die Definition der API-Endpunkte, die HTTP-Methoden und die Datenstrukturen, die von der API verwendet werden. Die API-Endpunkte sind so konzipiert, dass sie eine klare und konsistente Schnittstelle für die Kommunikation zwischen dem Front-End und dem Server bieten. Die API-Endpunkte sind nach REST-Prinzipien gestaltet und verwenden JSON als Datenformat.

1. Kinde Auth API-Endpunkte für Benutzer-Authentifizierung und -Autorisierung: Diese Endpunkte werden für die Anmeldung, Registrierung, Account-Verwaltung und Abmeldung der Benutzer verwendet.

```ts
// auth.ts
import { Hono } from "hono";
import { kindeClient, sessionManager } from "../kinde";
import { getUser } from "../kinde";
export const authRoute = new Hono()
  .get("/login", async (c) => {
    const loginUrl = await kindeClient.login(sessionManager(c));
    return c.redirect(loginUrl.toString());
  })
  .get("/register", async (c) => {
    const registerUrl = await kindeClient.register(sessionManager(c));
    return c.redirect(registerUrl.toString());
  })
  .get("/callback", async (c) => {
    const url = new URL(c.req.url);
    await kindeClient.handleRedirectToApp(sessionManager(c), url);
    return c.redirect("/");
  })
  .get("/logout", async (c) => {
    const logoutUrl = await kindeClient.logout(sessionManager(c));
    return c.redirect(logoutUrl.toString());
  })
  .get("/me", getUser, async (c) => {
    const user = c.var.user;
    return c.json({ user });
  });
```

Listing 4: API-Endpunkte für die Benutzer-Authentifizierung und -Autorisierung, Quelle: Eigene Darstellung

Hier werden die API-Endpunkte für die Benutzer-Authentifizierung und -Autorisierung definiert. Diese Endpunkte ermöglichen es den Benutzern, sich anzumelden, sich zu registrieren, ihren Account zu verwalten und sich abzumelden. Die Endpunkte sind nach REST-Prinzipien gestaltet und verwenden JSON als Datenformat.

2. Bun und Hono Server-Framework API:

Hono baut auf Bun auf und stellt die Routen und Endpunkte für die Webshop-APIs bereit. Diese APIs umfassen GET-, POST-, PUT- und DELETE-Anfragen, die jeweils die entsprechenden Funktionen des Webshops abdecken.

```
  // products.ts
import { Hono } from "hono";
import { zValidator } from "@hono/zod-validator";
import { z } from "zod";
import { db } from "../../db";
import { products as productTable } from "../../db/schema/products";

const productSchema = z.object({
  id: z.number().int().positive().min(1),
  productName: z.string().min(2).max(255),
  description: z.string().max(255).optional(),
  price: z.string(),
  categoryID: z.number().int().positive().min(1),
  availability: z.number().int(),
  image: z.string().optional()
});

type Products = z.infer<typeof productSchema>;
const createPostSchema = productSchema.omit({ id: true });

export const productsRoute = new Hono()
  .get("/", async (c) => {
    const products = await db.select().from(productTable);
    return c.json({ products: products });
  })
  ...
```

Listing 5: API-Endpunkte für die Produktverwaltung, Quelle: Eigene Darstellung

In diesem Code-Ausschnitt wird die Produktroute zum Abfragen aller Produkte definiert. Die Route verwendet Zod zur Validierung der Daten und Drizzle ORM zur Kommunikation mit der Datenbank, um Produkte abzufragen.

3. Zod Datenvalidierung API:

Zod wird zur Validierung der eingehenden Daten in den API-Anfragen verwendet. Dies stellt sicher, dass die Datenstruktur korrekt und vollständig ist, bevor sie weiterverarbeitet wird. Wie in der obigen Abbildung gezeigt, wird ein Zod-Schema für die Produktvalidierung definiert und in den API-Endpunkten verwendet.

Diese Endpunkte und APIs stellen sicher, dass alle wesentlichen Funktionen eines Webshops, wie Benutzerverwaltung, Produktverwaltung, Bestellabwicklung und Datenvalidierung, abgedeckt sind. Die Kombination der genannten Technologien sorgt für eine performante, sichere und gut wartbare Anwendung.

Authentifizierung und Benutzerverwaltung

Verbindung und Handhabung der Authentifizierung

Die Handhabung der Authentifizierung und Benutzerverwaltung findet hauptsächlich in den Kinde Auth API-Endpunkten statt. Die Datei “kinde.ts” konfiguriert mithilfe des Kinde-Clients die Authentifizierung und Autorisierung des Benutzers. Dabei wird der sessionManager aufgerufen und solang die Session gültig ist, wird der Benutzer authentifiziert und muss sich nicht erneut anmelden. Dies geschieht durch den Einsatz von Cookies, die die Session-ID speichern und bei jeder Anfrage an den Server übermitteln.

```
  // kinde.ts
  // Client for authorization code flow
export const kindeClient = createKindeServerClient(
  GrantType.AUTHORIZATION_CODE,
  {
    authDomain: process.env.KINDE_DOMAIN!,
    clientId: process.env.KINDE_CLIENT_ID!,
    clientSecret: process.env.KINDE_CLIENT_SECRET!,
    redirectURL: process.env.KINDE_REDIRECT_URI!,
    logoutRedirectURL: process.env.KINDE_LOGOUT_REDIRECT_URI!,
  }
);
export interface CustomSessionManager {
  getSessionItem(key: string): Promise<string | null>;
  setSessionItem(key: string, value: string): Promise<void>;
  removeSessionItem(key: string): Promise<void>;
  destroySession(): Promise<void>;
}

let store: Record<string, unknown> = {};

export const sessionManager = (c: Context): SessionManager => ({
  async getSessionItem(key: string) {
    const result = getCookie(c, key);
    return result;
  },
  async setSessionItem(key: string, value: unknown) {
    const cookieOptions = {
      httpOnly: true,
      secure: true,
      sameSite: "Lax",
    } as const;
    if (typeof value === "string") {
      setCookie(c, key, value, cookieOptions);
    } else {
      setCookie(c, key, JSON.stringify(value), cookieOptions);
    }
  },
  async removeSessionItem(key: string) {
    deleteCookie(c, key);
  },
  async destroySession() {
    ["id_token", "access_token", "user", "refresh_token"].forEach((key) => {
      deleteCookie(c, key);
    });
  },
});
```

Listing 6: Konfiguration der Kinde Auth-Client und Session-Manager, Quelle: Eigene Darstellung

Um die Benutzer zu authentifizieren, existiert eine getUser-Middleware, die die Benutzerdaten aus der Kinde Auth API abruft und in den API-Endpunkten verwendet wird. Die getUser-Middleware wird in den API-Endpunkten aufgerufen, um die Benutzerdaten abzurufen und zu überprüfen, ob der Benutzer authentifiziert ist. Falls dies nicht der Fall ist, wird eine “Unauthorized”-Fehlermeldung zurückgegeben.

```
// auth.ts
 type Env = {
  Variables: {
    user: UserType;
  };
};
export const getUser = createMiddleware<Env>(async (c, next) => {
  try {
    const manager = sessionManager(c);
    const isAuthenticated = await kindeClient.isAuthenticated(manager); // Boolean:
true or false

    if (!isAuthenticated) {
      return c.json({ error: "Unauthorized" }, 401);
    }
    const user = await kindeClient.getUserProfile(manager);
    c.set("user", user);
    await next();
  } catch (e) {
    console.error(e);
    return c.json({ error: "Unauthorized" }, 401);
  }
});
```

Listing 7: getUser-Middleware zur Benutzerauthentifizierung, Quelle: Eigene Darstellung

Authentifizierung und Autorisierung

Die Authentifizierung und Autorisierung erfolgt über die “auth.ts” Datei, dabei werden Routen definiert, welche die Benutzer zur Anmeldung, Registrierung, Account-Verwaltung und Abmeldung führen. Dabei wird überprüft ob der Benutzer bereits angemeldet ist und falls nicht, wird er aufgefordert sich anzumelden.

```ts
// auth.ts
import { Hono } from "hono";
import { kindeClient, sessionManager } from "../kinde";
import { getUser } from "../kinde";

export const authRoute = new Hono()
  .get("/login", async (c) => {
    const loginUrl = await kindeClient.login(sessionManager(c));
    return c.redirect(loginUrl.toString());
  })
  .get("/register", async (c) => {
    const registerUrl = await kindeClient.register(sessionManager(c));
    return c.redirect(registerUrl.toString());
  })
  .get("/callback", async (c) => {
    const url = new URL(c.req.url);
    await kindeClient.handleRedirectToApp(sessionManager(c), url);
    return c.redirect("/");
  })
  .get("/logout", async (c) => {
    const logoutUrl = await kindeClient.logout(sessionManager(c));
    return c.redirect(logoutUrl.toString());
  })
  .get("/me", getUser, async (c) => {
    const user = c.var.user;
    return c.json({ user });
  });
```

Listing 8: API-Endpunkte für die Benutzer-Authentifizierung und -Autorisierung, Quelle: Eigene Darstellung

Verwaltung der Adminrechte

Die Verwaltung der Adminrechte erfolgt über die Benutzerrollen, die in der Kinde Auth API definiert sind. Dabei wird überprüft, ob der Benutzer die erforderlichen Berechtigungen hat, um auf bestimmte Ressourcen zuzugreifen. Die Benutzerrollen werden in der Kinde Auth API definiert und können für verschiedene Benutzergruppen, wie Administratoren, Moderatoren und Benutzer, konfiguriert werden. Dabei wurde für bestimme Benutzer eine Admin-Rolle erstellt, um auf bestimmte Ressourcen zuzugreifen.

```ts
  // admin.ts
  // Middleware zur Berechtigungsprüfung
const checkIsAdmin = async (c: Context, next: Next) => {
  try {
    const manager = sessionManager(c);
    const isAuthenticated = await kindeClient.isAuthenticated(manager);
    if (!isAuthenticated) {
      return c.json({ error: "Unauthorized" }, 401);
    }
    const permissions = await kindeClient.getPermissions(manager);
    console.log("User Permissions:", permissions);
    const hasPermission = permissions.permissions.some(
      (perm: string) => perm === "isadmin"
    );
    console.log("Has permission:", hasPermission);
    if (!hasPermission) {
      return c.json({ error: "Forbidden" }, 403);
    }
    await next();
  } catch (error) {
    console.error("Error checking permission:", error);
    return c.json({ error: "Internal Server Error" }, 500);
  }
};
export { checkIsAdmin };
```

Listing 9: Middleware zur Berechtigungsprüfung für Adminrechte, Quelle: Eigener Code

Hier wird überprüft, ob der jeweilige Benutzer die erforderlichen Berechtigungen hat (ob “isadmin” wahr ist oder nicht), um auf bestimmte Ressourcen zuzugreifen. Falls der Benutzer nicht über die erforderlichen Berechtigungen verfügt, wird eine “Forbidden”-Fehlermeldung zurückgegeben.

Im folgenden Code-Ausschnitt wird die Middleware checkIsAdmin verwendet, um die Berechtigungen des Benutzers zu überprüfen und sicherzustellen, dass der Benutzer über die erforderlichen Berechtigungen verfügt, um auf die Ressourcen zuzugreifen. Falls der Benutzer beziehungsweise der Admin, erfolgreich Autorisiert wurde, wird der Zugriff auf die Ressourcen gewährt und der Admin kann wie im Code-Ausschnitt gezeigt, auf die Benutzerdaten zugreifen.

```
// admin.ts
adminRoute
.use(checkIsAdmin)
.get("/users", async (c) => {
  try {
    const accessToken = process.env.access_token;

    console.log("Access Token:", accessToken);
    const headers = {
      Accept: "application/json",
      Authorization: "Bearer ${accessToken}",
    };
    console.log("Headers:", headers);

    const response = await fetch("https://webshop.kinde.com/api/v1/users", {
      method: "GET",
      headers: headers,
    });
    console.log("Response:", response);
  }
}
)
```

Listing 10: Ausschnitt des API-Endpunktes für die Verwaltung der Benutzer, Quelle: Eigener Code

Neben dem Zugriff auf die Benutzerdaten, welche die CRUD-Operationen beinhalten, kann der Admin auch auf die Bestellungen, Produkte und Produktkategorien zugreifen. Dabei wird überprüft, ob der Benutzer die erforderlichen Berechtigungen hat, um auf die Ressourcen zuzugreifen. Falls der Benutzer nicht über die erforderlichen Berechtigungen verfügt, wird eine “Forbidden”-Fehlermeldung zurückgegeben.