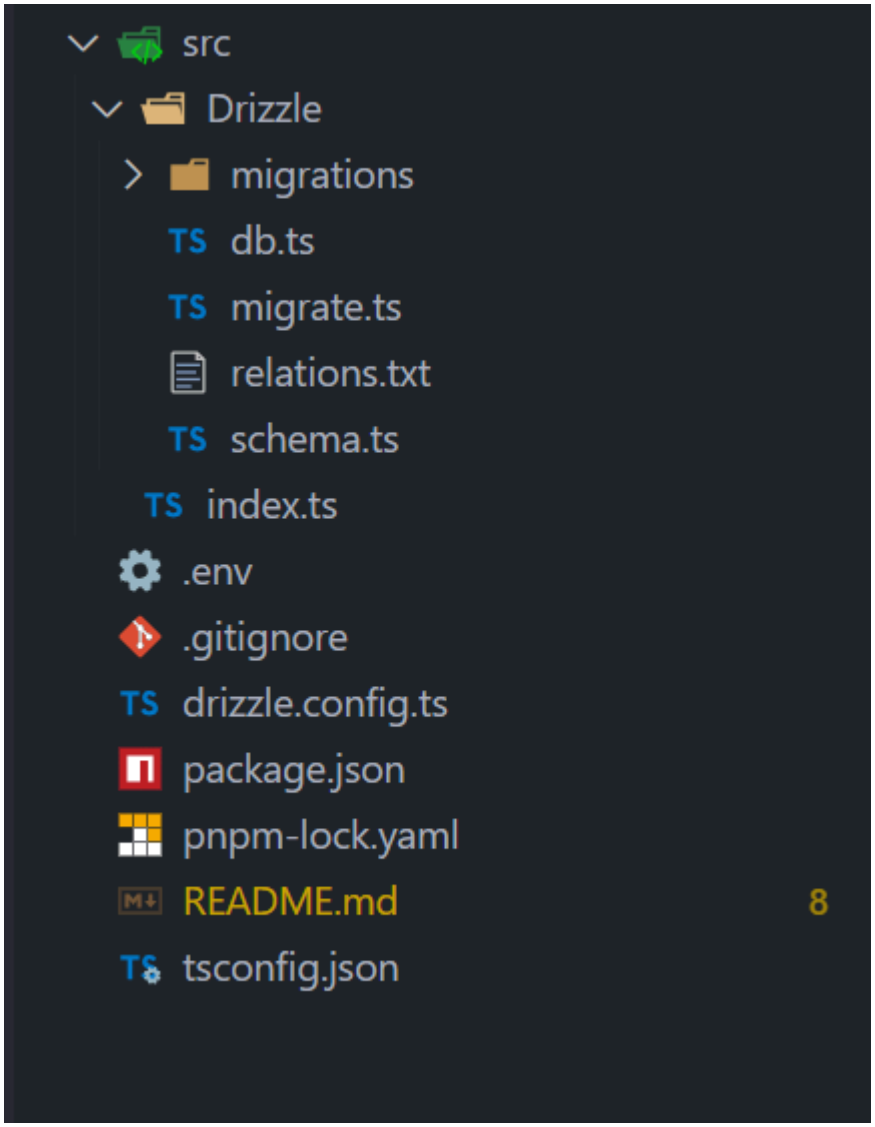
Car Rental Management System DB

Setting up Drizzle ORM

- Drizzle ORM is a headless TypeScript ORM
- It’s the only ORM with both relational and SQL-like query APIs, providing you best of both worlds when it comes to accessing your relational data.

At the end, you should have a folder structure like this:



Step 1: Setup TypeScript

- Use the guide available on the Readme.md on gitHub: setup ts with tsx

step 2: Insatll Drizzle packages (dependancies and dev dependancies)

```
pnpm add drizzle-orm pg dotenv
pnpm add -D drizzle-kit tsx @types/pg
```

You should now have them installed and available in ***package.json***

step 3: Connecting to a database

- Adding a connectoin string. Create a file on the root of the project called: ***.env*** and add the following connection.

```
# postgres://username:password@localhost:5432/mydatabase
Database_URL=postgres://postgres:yourpass@localhost:5432/car_rental-db
```

- The connection should be ignored from tracking by git.
- On the root dir, create **.gitignore **and add the following:

```
node_modules

dist

.env
```

step 4: Write the schema

- Create the **Drizzle **folder in **src **folder
- Create a file called ***schema.ts***
- This file will contain the definitions to the tables and the relationships
- Paste the following table schema to the file

```
import { relations } from "drizzle-orm";
import { text, varchar, serial, pgTable, decimal, integer, boolean, date } from
```

```
"drizzle-orm/pg-core";


// customer table
export const CustomerTable = pgTable("customer", {
    customerID: serial("customerID").primaryKey(),
    firstName: varchar("FirstName", { length: 50 }).notNull(),
    lastName: varchar("LastName", { length: 50 }).notNull(),
    email: varchar("Email", { length: 100 }).notNull().unique(),
    phoneNumber: text("PhoneNumber"),
    address: varchar("Address", { length: 255 })
});

// Location Table
export const LocationTable = pgTable("location", {
    locationID: serial("LocationID").primaryKey(),
    locationName: varchar("LocationName", { length: 100 }).notNull(),
    address: text("Address").notNull(),
    contactNumber: varchar("ContactNumber", { length: 20 })
});

// car table
export const CarTable = pgTable("car", {
    carID: serial("CarID").primaryKey(),
    carModel: varchar("CarModel", { length: 100 }).notNull(),
    year: date("Year").notNull(),
    color: varchar("Color", { length: 30 }),
    rentalRate: decimal("RentalRate", { precision: 10, scale: 2 }).notNull(),
    availability: boolean("Availability").default(true),
    locationID: integer("LocationID").references(() => LocationTable.locationID, {
onDelete: "set null" })
})

// Reservation Table
export const ReservationTable = pgTable("reservation", {
    reservationID: serial("ReservationID").primaryKey(),
    customerID: integer("CustomerID").notNull().references(() =>
CustomerTable.customerID, { onDelete: "cascade" }),
    carID: integer("CarID").notNull().references(() => CarTable.carID, { onDelete:
"cascade" }),
    reservationDate: date("ReservationDate").notNull(),
    pickupDate: date("PickupDate").notNull(),
    returnDate: date("ReturnDate")
});

//Booking Table
export const BookingsTable = pgTable("bookings", {
    bookingID: serial("BookingID").primaryKey(),
    carID: integer("CarID").notNull().references(() => CarTable.carID, { onDelete:
"cascade" }),
    customerID: integer("CustomerID").notNull().references(() =>
CustomerTable.customerID, { onDelete: "cascade" }),
    rentalStartDate: date("RentalStartDate").notNull(),
    rentalEndDate: date("RentalEndDate").notNull(),
    totalAmount: decimal("TotalAmount", { precision: 10, scale: 2 })
});
```

```
// Payment Table
export const PaymentTable = pgTable("payment", {
    paymentID: serial("PaymentID").primaryKey(),
    bookingID: integer("BookingID").notNull().references(() =>
BookingsTable.bookingID, { onDelete: "cascade" }),
    paymentDate: date("PaymentDate").notNull(),
    amount: decimal("Amount", { precision: 10, scale: 2 }).notNull(), //
{precision: 10, scale: 2} means 10 digits total, 2 of which are after the decimal
point. i.e // 12345678.90
    paymentMethod: text("PaymentMethod")
});

// Maintenance Table
//
export const MaintenanceTable = pgTable("maintenance", {
    maintenanceID: serial("MaintenanceID").primaryKey(),
    carID: integer("CarID").notNull().references(() => CarTable.carID, { onDelete:
"cascade" }),
    maintenanceDate: date("MaintenanceDate").notNull(),
    description: varchar("Description", { length: 255 }),
    cost: decimal("Cost", { precision: 10, scale: 2 })
});


// Insurance Table

export const InsuranceTable = pgTable("insurance", {
    insuranceID: serial("InsuranceID").primaryKey(),
    carID: integer("CarID").notNull().references(() => CarTable.carID, { onDelete:
"cascade" }),
    insuranceProvider: varchar("InsuranceProvider", { length: 100 }).notNull(),
    policyNumber: varchar("PolicyNumber").notNull(),
    startDate: date("StartDate").notNull(),
    endDate: date("EndDate")
});


// RELATIONSHIPS

// CustomerTable Relationships - 1 customer can have many reservations and bookings
export const CustomerRelations = relations(CustomerTable, ({ many }) => ({
    reservations: many(ReservationTable),
    bookings: many(BookingsTable)
}))

// LocationTable Relationships -  1 location can have many cars
export const LocationRelationships = relations(LocationTable, ({ many }) => ({
    cars: many(CarTable)
}))

// CarTable Relationships - 1 car can have many reservations, bookings,
maintenance, and insurance
export const CarRelations = relations(CarTable, ({ many, one }) => ({
    location: one(LocationTable, {
        fields: [CarTable.locationID],
        references: [LocationTable.locationID]
    }),
```

```
    reservations: many(ReservationTable),
    bookings: many(BookingsTable),
    maintenanceRecords: many(MaintenanceTable),
    insurancePolicies: many(InsuranceTable)
}));

// ReservationTable Relationships - 1 reservation belongs to 1 customer and 1 car
export const ReservationRelations = relations(ReservationTable, ({ one }) => ({
    customer: one(CustomerTable, {
        fields: [ReservationTable.customerID],
        references: [CustomerTable.customerID]
    }),
    car: one(CarTable, {
        fields: [ReservationTable.carID],
        references: [CarTable.carID]
    })
}))

// BookingsTable Relationships - 1 booking belongs to 1 customer and 1 car, and can
have many payments
export const BookingsRelations = relations(BookingsTable, ({ one, many }) => ({
    customer: one(CustomerTable, {
        fields: [BookingsTable.customerID],
        references: [CustomerTable.customerID]
    }),
    car: one(CarTable, {
        fields: [BookingsTable.carID],
        references: [CarTable.carID]
    }),
    payments: many(PaymentTable)
}))

// PaymentTable Relationships - 1 payment belongs to 1 booking
export const PaymentRelations = relations(PaymentTable, ({ one }) => ({
    booking: one(BookingsTable, {
        fields: [PaymentTable.bookingID],
        references: [BookingsTable.bookingID]
    })
}))

// MaintenanceTable Relationships - 1 maintenance record belongs to 1 car
export const MaintenanceRelations = relations(MaintenanceTable, ({ one }) => ({
    car: one(CarTable, {
        fields: [MaintenanceTable.carID],
        references: [CarTable.carID]
    })
}));

// InsuranceTable Relationships - 1 insurance policy belongs to 1 car
export const InsuranceRelations = relations(InsuranceTable, ({ one }) => ({
    car: one(CarTable, {
        fields: [InsuranceTable.carID],
        references: [CarTable.carID]
    })
}));
```

Step 5: Creating a connection to the database

- Create a file ***db.ts***
- ***db.ts*** will be responsible for connection to the database anytime we need to make any operation. Have the code below:

```
import "dotenv/config"

import { drizzle } from "drizzle-orm/node-postgres"
import { Client } from "pg"
import * as schema from "./schema"

export const client = new Client({
    connectionString: process.env.Database_URL as string
})

const main = async () => {
    await client.connect()
}
main().then(() => {
    console.log("Connected to the database")
}).catch((error) => {
    console.error("Error connecting to the database:", error)
})


const db = drizzle(client, { schema, logger: true })

export default db
```

Step 6: Configure Drizzle for your project

On the root of the project, create a file: ***drizzle.config.ts** , * the file will be used to configer drizzle for postgres

```
import "dotenv/config";
import { defineConfig } from "drizzle-kit";

export default defineConfig({
    dialect: "postgresql", // means we are using PostgreSQL
    schema: "./src/Drizzle/schema.ts", // path to the schema file
    out: "./src/Drizzle/migrations", // path to the migrations folder
    dbCredentials: { // database connection details
        url: process.env.Database_URL as string
    },
    verbose: true, // enables detailed logging
    strict: true, // enables strict mode for type safety, i.e. it will throw an
```

```
error if there are any issues with the schema
});
```

Step 7: Ready for Migrations

- Back to the *Drizzle* folder, create a file and name it ***migrate.ts***
- The file is responsible for migrations of the schema to the postgres server.

```
import "dotenv/config";
import { migrate } from "drizzle-orm/node-postgres/migrator";
import db, { client } from "./db"

async function migration() {
    console.log("......Migrations Started......");
    await migrate(db, { migrationsFolder: __dirname + "/migrations" });
    await client.end();
    console.log("......Migrations Completed......");
    process.exit(0); // 0 means success
}

migration().catch((error) => {
    console.error("Migration failed:", error);
    process.exit(1); // 1 means an error occurred
});
```

Step 8: Adding scripts to generate and migrate the schema

- On your ***package.json*** file, add two scripts to generate and migrate the schema we have created.

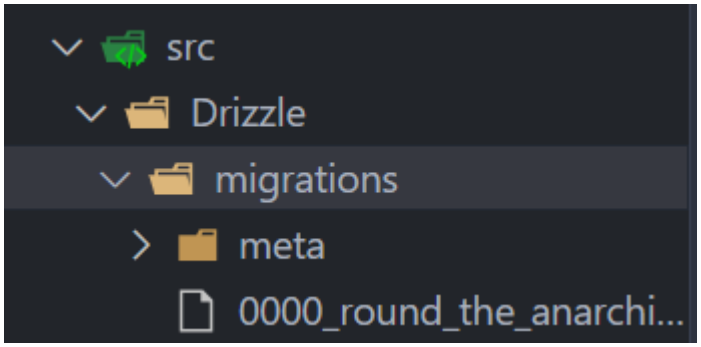
```
"generate":"drizzle-kit generate",
"migrate":"tsx src/drizzle/migrate.ts"
```

Step 8: Run the generate command

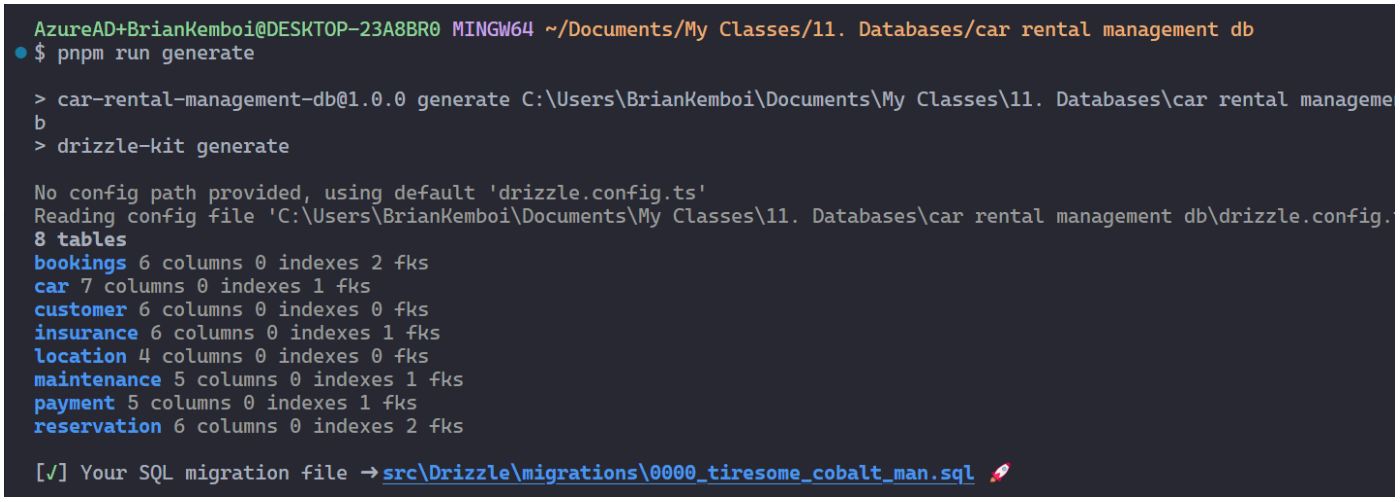
- Its now time to generate the migrations, on your terminal, run:

```
pnpm run generate
```

- A folder called migrations is then created:



- This is what you will see on your terminal:



Step 9: Migrate your Shema to Postgres Server

- Its now time to migrate our schema to Postgres Server on our ,local machine:
- On your terminal, run the command:

```
pnpm run migrate
```

- The command will excecute the connection to the database, creation of all the tables and closing the connection.

Seeding the Database

Seeding is the process of adding data to the tables created. The purpose of seeding is to help us query the data.

step 1: create a file inside the drizzle folder and name it seed.ts

- Add the following data into the newly created file:

```
import { relations } from "drizzle-orm";
import { text, varchar, serial, pgTable, decimal, integer, boolean, date }
from "drizzle-orm/pg-core";


// customer table
export const CustomerTable = pgTable("customer", {
    customerID: serial("customerID").primaryKey(),
    firstName: varchar("FirstName", { length: 50 }).notNull(),
    lastName: varchar("LastName", { length: 50 }).notNull(),
    email: varchar("Email", { length: 100 }).notNull().unique(),
    phoneNumber: text("PhoneNumber"),
    address: varchar("Address", { length: 255 })
});

// Location Table
export const LocationTable = pgTable("location", {
    locationID: serial("LocationID").primaryKey(),
    locationName: varchar("LocationName", { length: 100 }).notNull(),
    address: text("Address").notNull(),
    contactNumber: varchar("ContactNumber", { length: 20 })
});

// car table
export const CarTable = pgTable("car", {
    carID: serial("CarID").primaryKey(),
    carModel: varchar("CarModel", { length: 100 }).notNull(),
    year: date("Year").notNull(),
    color: varchar("Color", { length: 30 }),
    rentalRate: decimal("RentalRate", { precision: 10, scale: 2 }).notNull(),
    availability: boolean("Availability").default(true),
    locationID: integer("LocationID").references(() =>
LocationTable.locationID, { onDelete: "set null" })
})

// Reservation Table
export const ReservationTable = pgTable("reservation", {
    reservationID: serial("ReservationID").primaryKey(),
    customerID: integer("CustomerID").notNull().references(() =>
CustomerTable.customerID, { onDelete: "cascade" }),
    carID: integer("CarID").notNull().references(() => CarTable.carID, {
onDelete: "cascade" }),
```

```
    reservationDate: date("ReservationDate").notNull(),
    pickupDate: date("PickupDate").notNull(),
    returnDate: date("ReturnDate")
});

//Booking Table
export const BookingsTable = pgTable("bookings", {
    bookingID: serial("BookingID").primaryKey(),
    carID: integer("CarID").notNull().references(() => CarTable.carID, {
onDelete: "cascade" }),
    customerID: integer("CustomerID").notNull().references(() =>
CustomerTable.customerID, { onDelete: "cascade" }),
    rentalStartDate: date("RentalStartDate").notNull(),
    rentalEndDate: date("RentalEndDate").notNull(),
    totalAmount: decimal("TotalAmount", { precision: 10, scale: 2 })
});

// Payment Table
export const PaymentTable = pgTable("payment", {
    paymentID: serial("PaymentID").primaryKey(),
    bookingID: integer("BookingID").notNull().references(() =>
BookingsTable.bookingID, { onDelete: "cascade" }),
    paymentDate: date("PaymentDate").notNull(),
    amount: decimal("Amount", { precision: 10, scale: 2 }).notNull(), //
{precision: 10, scale: 2} means 10 digits total, 2 of which are after the
decimal point. i.e // 12345678.90
    paymentMethod: text("PaymentMethod")
});

// Maintenance Table
//
export const MaintenanceTable = pgTable("maintenance", {
    maintenanceID: serial("MaintenanceID").primaryKey(),
    carID: integer("CarID").notNull().references(() => CarTable.carID, {
onDelete: "cascade" }),
    maintenanceDate: date("MaintenanceDate").notNull(),
    description: varchar("Description", { length: 255 }),
    cost: decimal("Cost", { precision: 10, scale: 2 })
});


// Insurance Table

export const InsuranceTable = pgTable("insurance", {
    insuranceID: serial("InsuranceID").primaryKey(),
    carID: integer("CarID").notNull().references(() => CarTable.carID, {
onDelete: "cascade" }),
    insuranceProvider: varchar("InsuranceProvider", { length: 100
}).notNull(),
    policyNumber: varchar("PolicyNumber").notNull(),
    startDate: date("StartDate").notNull(),
    endDate: date("EndDate")
});


// RELATIONSHIPS
```

```
// CustomerTable Relationships - 1 customer can have many reservations and
bookings
export const CustomerRelations = relations(CustomerTable, ({ many }) => ({
    reservations: many(ReservationTable),
    bookings: many(BookingsTable)
}))

// LocationTable Relationships -  1 location can have many cars
export const LocationRelationships = relations(LocationTable, ({ many }) => ({
    cars: many(CarTable)
}))

// CarTable Relationships - 1 car can have many reservations, bookings,
maintenance, and insurance
export const CarRelations = relations(CarTable, ({ many, one }) => ({
    location: one(LocationTable, {
        fields: [CarTable.locationID],
        references: [LocationTable.locationID]
    }),
    reservations: many(ReservationTable),
    bookings: many(BookingsTable),
    maintenanceRecords: many(MaintenanceTable),
    insurancePolicies: many(InsuranceTable)
}));

// ReservationTable Relationships - 1 reservation belongs to 1 customer and 1
car
export const ReservationRelations = relations(ReservationTable, ({ one }) =>
({
    customer: one(CustomerTable, {
        fields: [ReservationTable.customerID],
        references: [CustomerTable.customerID]
    }),
    car: one(CarTable, {
        fields: [ReservationTable.carID],
        references: [CarTable.carID]
    })
}))

// BookingsTable Relationships - 1 booking belongs to 1 customer and 1 car,
and can have many payments
export const BookingsRelations = relations(BookingsTable, ({ one, many }) =>
({
    customer: one(CustomerTable, {
        fields: [BookingsTable.customerID],
        references: [CustomerTable.customerID]
    }),
    car: one(CarTable, {
        fields: [BookingsTable.carID],
        references: [CarTable.carID]
    }),
    payments: many(PaymentTable)
}))

// PaymentTable Relationships - 1 payment belongs to 1 booking
export const PaymentRelations = relations(PaymentTable, ({ one }) => ({
    booking: one(BookingsTable, {
```

```
        fields: [PaymentTable.bookingID],
        references: [BookingsTable.bookingID]
    })
}))

// MaintenanceTable Relationships - 1 maintenance record belongs to 1 car
export const MaintenanceRelations = relations(MaintenanceTable, ({ one }) =>
({
    car: one(CarTable, {
        fields: [MaintenanceTable.carID],
        references: [CarTable.carID]
    })
}));

// InsuranceTable Relationships - 1 insurance policy belongs to 1 car
export const InsuranceRelations = relations(InsuranceTable, ({ one }) => ({
    car: one(CarTable, {
        fields: [InsuranceTable.carID],
        references: [CarTable.carID]
    })
}));
```

step 2: Add a script in your package.json to exceute the seed

Add the following script to package.json file

```
"seed": "tsx src/Drizzle/seed.ts"
```

- when exceuted, it will run a file in seed.ts file

step 3: Excetute the seed command

In your terminal, run the following command:

```
pnpm run seed
```

- The command will insert all the records in the tables and will be available for query

Perfom CRUD Operations

Select

- Select all Users

```
const getAllCustomers = async () => {
    return await db.query.CustomerTable.findMany()
}
```

- get customer by ID

```
const getCustomerById = async (customerID: number) => {
    return await db.query.CustomerTable.findFirst({
        where: eq(CustomerTable.customerID, customerID)
    })
}
```

- Customer with reservation

```
const getCustomerWithReservations = async (customerID: number) => {
    return await db.query.CustomerTable.findFirst({
        where: eq(CustomerTable.customerID, customerID),
        with: {
            reservations: true
        }
    })
}
```

- customer with bookings

```
const getCustomerWithBookings = async (customerID: number) => {
    return await db.query.CustomerTable.findFirst({
        where: eq(CustomerTable.customerID, customerID),
        with: {
            bookings: {
                columns: {
                    carID: true,
                    rentalStartDate: true,
                    rentalEndDate: true,
                    totalAmount: true
                }
```

```
            }
        }
    })
}
```

- Using select to fetch specific details

```
const getCustomerWithSelectedDetails = async (customerID: number) => {
    return await db.select({
        firstName: CustomerTable.firstName,
        lastName: CustomerTable.lastName,
        email: CustomerTable.email,
        phoneNumber: CustomerTable.phoneNumber
    })
        .from(CustomerTable)
        .where(eq(CustomerTable.customerID, customerID));
}
```

- Fetch Locations with all cars available

```
const getLocationsWithCars = async () => {
    return await db.query.LocationTable.findMany({
        with: {
            cars: {
                columns: {
                    carModel: true,
                    color: true,
                    rentalRate: true,
                    availability: true
                }
            }
        }
    })
}
```

Fetch maintenance for all cars

```
const getCarsWithMaintenance = async () => {
    return await db.query.CarTable.findMany({
        with: {
            maintenanceRecords: {
                columns: {
                    description: true,
                    maintenanceDate: true,
                    cost: true
                }
            }
        }
```

```
    })
}
```

Insert

- Insert a new record of a user

```
const newCustomer = {
    firstName: "Brian",
    lastName: "Kemboi",
    email: "kemboi@gmail.com",
    phoneNumber: "0712345678",
    address: "10 River Rd"
};

const insertCustomer = async (customer: TICustomer) => {
    const insertedCustomer = await
db.insert(CustomerTable).values(customer).returning();
    return insertedCustomer;
}
```

Update

- Update a user

```
const updateCustomer = async (email: string, updatedData: Partial<TICustomer>) => {
    const updatedCustomer = await db.update(CustomerTable)
        .set(updatedData)
        .where(eq(CustomerTable.email, email))
        .returning();
    return updatedCustomer;
}
```

Delete

- Delete a customer

```
const deleteCustomer = async (customerID: number) => {
    const deletedCustomer = await db.delete(CustomerTable)
        .where(eq(CustomerTable.customerID, customerID))
        .returning();
```

```
    return deletedCustomer;
}
```

Special Characters

- Using like %

```
const searchCustomersByName = async (name: string) => {
    return await db.query.CustomerTable.findMany({
        where: like(CustomerTable.firstName, `%${name}%`)
    })
}
```