Datenbankanbindung vorbereiten: Pro & Contra Analyse und Schritt-für-Schritt-Anleitung

Ihr Name

4. Februar 2025

Inhaltsverzeichnis

1. Pro & Contra Analyse von Datenbankoptionen

Für das Gym Progress Tracking System stehen grundsätzlich zwei Hauptvarianten zur Verfügung: relationale Datenbanken und NoSQL-Datenbanken. Im Folgenden werden einige Optionen analysiert:

1.1 Relationale Datenbanken (z.B. PostgreSQL, MySQL)

Vorteile:

- **ACID-Konformität:** Garantiert Konsistenz, Integrität und Zuverlässigkeit der Daten.
- **Strukturierte Datenmodellierung:** Ideal, wenn die Daten (z.B. Nutzerdaten, Trainingshistorien) gut strukturiert und in Beziehung zueinander stehen.
- **Komplexe Abfragen:** SQL bietet mächtige Abfragemöglichkeiten, die für Berichte und Analysen nützlich sind.
- **Bewährte Technologie:** Viele Tools, Frameworks und ORMs (z.B. Sequelize, TypeORM) unterstützen relationale Datenbanken.

Nachteile:

- **Schema-Fixierung:** Änderungen am Datenmodell können aufwändig sein, wenn sich Anforderungen ändern.
- **Horizontale Skalierung:** Kann komplexer sein als bei einigen NoSQL-Lösungen, obwohl vertikale Skalierung oft ausreichend ist.

1.2 NoSQL-Datenbanken (z.B. MongoDB, Firebase)

Vorteile:

- **Flexibles Schema:** Ermöglicht schnelle Iterationen, wenn sich die Datenstruktur noch ändern kann.
- **Einfache horizontale Skalierung:** Gut geeignet für Anwendungen, die hohe Schreib-/Lesevolumen verarbeiten.

Nachteile:

- **Weniger strenge Konsistenz:** Oft nicht voll ACID-konform (obwohl neuere Versionen in bestimmten Konfigurationen Transaktionen unterstützen).
- **Eingeschränkte Abfragemöglichkeiten:** Komplexe relationale Abfragen sind oft schwieriger umzusetzen.

1.3 Empfehlung

Für das Gym Progress Tracking System empfehle ich die Verwendung einer relationalen Datenbank, und zwar insbesondere **PostgreSQL**. Gründe:

- Die Daten (Nutzer, Trainingshistorie, Geräteinformationen) sind gut strukturiert und stehen in klaren Beziehungen zueinander.
- PostgreSQL bietet fortschrittliche Features wie JSON-Support, was bei Bedarf flexibel eingesetzt werden kann.
- Es gibt umfangreiche Unterstützung durch ORMs wie Sequelize oder TypeORM, die die Entwicklung vereinfachen.

2. Schritt-für-Schritt-Anleitung zur Einrichtung der Datenbankanbindung

Diese Anleitung führt Sie durch die Einrichtung von PostgreSQL und das Erstellen eines einfachen Benutzer-Modells.

2.1 Schritt 1: PostgreSQL installieren und konfigurieren

1. **Download und Installation:**
   Besuchen Sie die PostgreSQL-Website (`https://www.postgresql.org/`) und laden Sie die passende Version für Ihr Betriebssystem herunter. Folgen Sie den Installationsanweisungen.
2. **PostgreSQL starten:**
   Nach der Installation starten Sie den PostgreSQL-Server. Unter Windows können Sie beispielsweise über den "pgAdminöder den Dienst-Manager sicherstellen, dass der Server läuft.
3. **Erstellen einer Datenbank:**
   Öffnen Sie pgAdmin oder verwenden Sie die Kommandozeile (psql), um eine neue Datenbank zu erstellen. Beispiel in psql:

   ```
   CREATE DATABASE gym_tracking;
   ```

2.2 Schritt 2: Node.js-Backend mit PostgreSQL verbinden

1. **Passendes Node.js-Paket installieren:**
   Installieren Sie ein Paket, um mit PostgreSQL zu kommunizieren. Häufig verwendete Pakete sind `pg` (node-postgres) oder ein ORM wie Sequelize. Für diese Anleitung verwenden wir `pg`:

   ```
   npm install pg
   ```

2. **Erstellen Sie eine Datenbankverbindungsdatei:**
   Erstellen Sie im Backend-Ordner eine Datei, z. B. `db.js`. Fügen Sie den folgenden Beispielcode ein:

```
const { Pool } = require('pg');

const pool = new Pool({
  user: 'IhrDBBenutzer',          // Ersetzen Sie dies durch Ihren Datenbankbenutz
  host: 'localhost',
  database: 'gym_tracking',        // Der Name Ihrer Datenbank
  password: 'IhrDBPasswort',       // Ersetzen Sie dies durch Ihr Datenbankpasswor
  port: 5432,                      // Standardport von PostgreSQL
});

module.exports = pool;
```

2.3 Schritt 3: Erstellen eines Benutzer-Modells

Sie können nun ein einfaches Modell erstellen, um Benutzerdaten zu speichern. Hier zeigen wir, wie Sie einen Endpunkt in Ihrer `server.js` hinzufügen, der einen neuen Benutzer registriert.

1. **Erstellen Sie einen Registrierungs-Endpunkt:**
   Öffnen Sie Ihre `server.js` im Backend-Ordner und fügen Sie folgenden Code hinzu:

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const pool = require('./db');  // Verbindung zur Datenbank
const app = express();
const PORT = process.env.PORT || 5000;

// Middleware, um JSON-Daten zu verarbeiten
app.use(express.json());

// Registrierungs-Endpunkt
app.post('/api/register', async (req, res) => {
  const { name, email, password } = req.body;
  try {
    // Passwort hashen
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Benutzer in der Datenbank speichern
    const newUser = await pool.query(
      'INSERT INTO users (name, email, password) VALUES ($1, $2, $3) RETURNING *'
      [name, email, hashedPassword]
    );
```

```
    // Optional: JWT-Token generieren
    const token = jwt.sign(
      { userId: newUser.rows[0].id },
      'IhrGeheimerSchlüssel',  // Ersetzen Sie dies durch einen sicheren Schlüsse
      { expiresIn: '1h' }
    );

    res.json({ message: 'Benutzer erfolgreich registriert', token });
  } catch (error) {
    console.error(error.message);
    res.status(500).send('Serverfehler');
  }
});

// Beispiel-Login-Endpunkt (optional)
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
  try {
    // Suchen Sie den Benutzer in der Datenbank
    const user = await pool.query('SELECT * FROM users WHERE email = $1', [email]
    if (user.rows.length === 0) {
      return res.status(401).json({ error: 'Ungültige Anmeldedaten' });
    }

    // Vergleichen Sie das Passwort
    const validPassword = await bcrypt.compare(password, user.rows[0].password);
    if (!validPassword) {
      return res.status(401).json({ error: 'Ungültige Anmeldedaten' });
    }

    // JWT-Token generieren
    const token = jwt.sign(
      { userId: user.rows[0].id },
      'IhrGeheimerSchlüssel',
      { expiresIn: '1h' }
    );

    res.json({ message: 'Login erfolgreich', token });
  } catch (error) {
    console.error(error.message);
    res.status(500).send('Serverfehler');
  }
});

app.listen(PORT, () => {
  console.log(`Server läuft auf Port ${PORT}`);
});
```

2. **Hinweis:**
   Sie müssen in Ihrer Datenbank (PostgreSQL) eine Tabelle namens `users` anlegen, die zumindest die Spalten `id` (Primärschlüssel), `name`, `email` und `password` enthält. Dies können Sie z. B. über pgAdmin oder psql durchführen:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL
);
```

3. Zusammenfassung

- Wir haben die benötigten Pakete `jsonwebtoken` und `bcryptjs` installiert.
- Wir haben eine Datenbankverbindungsdatei (`db.js`) erstellt, um unser Node.js-Backend mit PostgreSQL zu verbinden.
- Wir haben einen einfachen Registrierungs-Endpunkt in der `server.js` implementiert, der:
  - Eingaben entgegennimmt,
  - das Passwort sicher mit `bcryptjs` hasht,
  - die Daten in der Datenbank speichert,
  - und optional ein JWT-Token zurückgibt.
- Denken Sie daran, dass Sie die `users`-Tabelle in Ihrer PostgreSQL-Datenbank erstellen müssen.

Diese Anleitung bietet einen ausführlichen Überblick, wie Sie die Datenbankanbindung vorbereiten und ein einfaches Benutzer-Modell implementieren. Dies ist ein wesentlicher Schritt, um die Nutzerregistrierung und -authentifizierung im weiteren Verlauf sicher und effizient umzusetzen.