🎓 Studenten-Notenverwaltung - TypeScript Übung

Teil 1: Backend Setup

Schritt 1: Backend-Projekt initialisieren

```
mkdir student-grades
mkdir student-grades/backend
cd student-grades/backend
npm init -y
```

Schritt 2: Dependencies installieren

```
npm install express mongoose cors
npm install -D typescript @types/express @types/node @types/cors tsx
nodemon
```

Was installieren wir?

- `express`: Web-Framework
- `mongoose`: MongoDB ODM
- `cors`: Erlaubt Frontend-Zugriff
- TypeScript-Pakete für Entwicklung

Schritt 3: TypeScript konfigurieren

Datei erstellen: `tsconfig.json`

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "noEmit": true,
    "moduleResolution": "node"
  },
  "include": ["src"], // Compiler soll nur Dateien in src berücksichtigen
  "exclude": ["node_modules"] // Ignoriert node_modules (eigentlich
```

```
  standardmäßig ausgeschlossen)
  }
```

Schritt 4: Package.json Scripts anpassen

In `package.json` unter `"scripts"` hinzufügen:

```
"scripts": {
  "build": "tsc",
  "start": "tsx --env-file=.env src/server.ts"
}
```

Schritt 5: Ordnerstruktur erstellen

```
mkdir src
mkdir src/models
mkdir src/routes
mkdir src/controllers
mkdir src/config
```

Finale Struktur:

```
student-grades/backend/
├── src/
│   ├── config/
│   │   └── db.ts
│   ├── models/
│   │   └── Student.ts
│   ├── routes/
│   │   └── studentsRouter.ts
│   ├── controllers/
│   │   └── studentsController.ts
│   └── server.ts
├── .env
├── tsconfig.json
└── package.json
```

Schritt 6: Environment-Variablen

Datei erstellen: `.env`

```
MONGO_URL=mongodb://localhost:27017
DATABASE=student_grades
PORT=3000
```

Schritt 7: Datenbankverbindung

Datei erstellen: `src/config/db.ts`

```
import mongoose from "mongoose";

const URL = process.env.MONGO_URL || "";
console.log({URL});


mongoose.connection.on('error', (error) => {
    console.log('DB after initial connection:', error);
});

const connectDB = async () => {
    try {
        await mongoose.connect(URL, {
            dbName: process.env.DATABASE
        });
        console.log('Connected to MongoDB!');
    } catch (error) {
        console.error('Connection error:', error);
    }
};

export default connectDB;
```

BEACHTET : Der Code ist etwas anders als die Datenbankverbindung, die wir sonst genutzt haben.

- Ihr könnt zum Test einmal den alten Code (ohne Typescript) der Datenbankverbindung hier einfügen und schauen, was passiert.

Schritt 8: Student Model mit TypeScript

Datei erstellen: `src/models/Student.ts`

```
import { Document, Schema, model } from "mongoose";

// ✅ TypeScript Konzept: type Definition
// ✅ Konzept: string und number als primitive Typen
// ✅ Konzept: Optionaler Parameter mit ?
type StudentType = {
  name: string;
  grade: number;
  _id?: string;  // Optional, weil MongoDB das automatisch generiert
};

// ✅ Konzept: Typen kombinieren mit & (Intersection)
```

```
// Document ist der Mongoose-Typ für Datenbankdokumente
// enthält Methoden wie .save() und _id
type StudentDocument = StudentType & Document;

// Mongoose Schema
const studentSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  grade: {
    type: Number,
    required: true,
    min: 1,
    max: 6
  }
}, {
  timestamps: true
});

// Model exportieren
const Student = model<StudentDocument>("Student", studentSchema);

// Types exportieren für Verwendung in anderen Dateien
export { Student, StudentType, StudentDocument };
```

TypeScript-Konzepte hier:

- ✅ `type` Definition
- ✅ `string` und `number` Typen
- ✅ Optionaler Parameter `_id?`
- ✅ Objekt-Typisierung

Schritt 9a: Routes mit TypeScript

Datei erstellen: `src/routes/studentsRouter.ts`

```
import express from "express";
import { getAllStudents, createStudent } from
"../controllers/studentsController";

const studentsRouter = express.Router();

studentsRouter
   .get("/", getAllStudents)
   .post("/", createStudent)

export default studentsRouter;
```

Schritt 9b: Students Controller erstellen

```
import { Request, Response } from "express";
import { Student, StudentType } from "../models/Student";

// ✅ TypeScript Konzept: Funktions-Typisierung mit Request & Response
// GET - Alle Studenten abrufen
export const getAllStudents = async (req: Request, res: Response):
Promise<void> => {
  try {
    // ✅ Konzept: Array-Typisierung - students ist vom Typ
StudentDocument[]
    const students = await Student.find();

    res.json(students);
  } catch (error: unknown) {
    // ✅ Konzept: unknown Type für Error-Handling
    // unknown ist sicherer als any - wir müssen den Typ prüfen

    if (error instanceof Error) {
      res.status(500).json({ message: error.message });
    } else {
      res.status(500).json({ message: "Ein unbekannter Fehler ist
aufgetreten" });
    }
  }
};

// POST - Neuen Student erstellen
export const createStudent = async (req: Request, res: Response):
Promise<void> => {
  try {
    // ✅ Konzept: Objekt-Typisierung
    const { name, grade }: StudentType = req.body;

    // Validierung
    if (!name || !grade) {
      res.status(400).json({ message: "Name und Note sind erforderlich" });
      return;
    }

    if (grade < 1 || grade > 6) {
      res.status(400).json({ message: "Note muss zwischen 1 und 6 liegen"
});
      return;
    }

    const newStudent = new Student({ name, grade });
    const savedStudent = await newStudent.save();

    res.status(201).json(savedStudent);
  } catch (error: unknown) {
    if (error instanceof Error) {
```

```
      res.status(500).json({ message: error.message });
    } else {
      res.status(500).json({ message: "Ein unbekannter Fehler ist
aufgetreten" });
    }
  }
};
```

TypeScript-Konzepte hier:

- ✅ Funktions-Typisierung mit explizitem Rückgabetyp Promise
- ✅ unknown Type für Error-Handling
- ✅ Array-Typisierung (implizit)
- ✅ Objekt-Destructuring mit Type

Schritt 10: Server Setup

Datei erstellen: `src/server.ts`

```
import express, { Express } from "express";
import cors from "cors";
import connectDB from "./config/db";
import studentRoutes from "./routes/studentsRouter";

// ✅ TypeScript Konzept: Explizite Typisierung
const app: Express = express();
const PORT: number = parseInt(process.env.PORT || "5000");

// Middleware
app.use(cors());
app.use(express.json());

// Routen
app.use("/api/students", studentRoutes);

// ✅ Konzept: Funktions-Typisierung mit async
async function startServer(): Promise<void> {
  await connectDB();

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

startServer();
```

TypeScript-Konzepte hier:

- ✅ Explizite Typisierung (`Express`, `number`)
- ✅ Funktions-Rückgabetyp `Promise<void>`

Schritt 11: Backend starten

```
npm run start
```

Erwartete Ausgabe:

```
Server läuft auf Port 5000
Connected to MongoDB!
```

Schritt 12: Backend testen (Optional)

Mit curl oder Postman:

```
# Student hinzufügen
curl -X POST http://localhost:5000/api/students \
  -H "Content-Type: application/json" \
  -d '{"name": "Max Mustermann", "grade": 2.5}'

# Alle Studenten abrufen
curl http://localhost:5000/api/students
```

✅ Backend ist fertig!

Was haben wir gelernt?

- ✅ `type` Definitionen
- ✅ Primitive Typen: `string`, `number`
- ✅ `unknown` für sicheres Error-Handling
- ✅ Optionale Parameter mit `?`
- ✅ Array-Typisierung
- ✅ Objekt-Typisierung
- ✅ Funktions-Typisierung

Teil 2: Frontend Setup

Schritt 1: Frontend-Projekt erstellen

```
# Im frontend-Ordner
npm create vite@latest . -- --template react-ts
```

```
npm install
```

Was passiert hier?

- Vite erstellt ein React-Projekt mit TypeScript-Template
- `react-ts` = React + TypeScript

Schritt 2: Projektstruktur vorbereiten

Ordner erstellen:

```
mkdir src/types
mkdir src/components
mkdir src/api
```

Finale Struktur:

```
student-grades/frontend/
├── src/
│   ├── api/
│   │   └── students.ts
│   ├── components/
│   │   ├── StudentList.tsx
│   │   └── AddStudentForm.tsx
│   ├── types/
│   │   └── student.ts
│   ├── App.tsx
│   └── main.tsx
├── package.json
└── tsconfig.json
```

Schritt 3: TypeScript Types definieren

Datei erstellen: `src/types/student.ts`

```
// ✅ TypeScript Konzept: type Definition
// ✅ Konzept: string und number als primitive Typen
// ✅ Konzept: Optionaler Parameter mit ?
export type Student = {
  _id?: string;      // Optional, kommt von MongoDB
  name: string;
  grade: number;
  createdAt?: string;  // Optional, Timestamp von MongoDB
  updatedAt?: string;  // Optional, Timestamp von MongoDB
};
```

```
// ✅ Konzept: Union Types
// Status kann nur einen dieser drei Werte haben
export type LoadingStatus = 'idle' | 'loading' | 'error';
```

TypeScript-Konzepte hier:

- ✅ `type` Definition
- ✅ Primitive Typen: `string`, `number`
- ✅ Optionale Parameter mit `?`
- ✅ **Union Types** für beschränkte Werte

Schritt 5: API-Service erstellen

Datei erstellen: `src/api/students.ts`

```
import type { Student } from "../types/student";

const API_URL = "http://localhost:3000/api/students";

// ✅ TypeScript Konzept: Funktions-Typisierung mit Promise und Array
// Funktion gibt ein Promise zurück, das ein Array von Students enthält
export const fetchStudents = async (): Promise<Student[]> => {
  const response = await fetch(API_URL);

  if (!response.ok) {
    throw new Error("Fehler beim Laden der Studenten");
  }

  // ✅ Konzept: Response-Daten als Array typisieren
  const data: Student[] = await response.json();
  return data;
};

// ✅ Konzept: Funktionsparameter typisieren
// Funktion nimmt ein Student-Objekt ohne _id und gibt ein Student zurück
export const addStudent = async (student: Omit<Student, '_id' | 'createdAt'
| 'updatedAt'>): Promise<Student> => {
  const response = await fetch(API_URL, {
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify(student)
  });

  if (!response.ok) {
    throw new Error("Fehler beim Hinzufügen des Studenten");
  }

  const data: Student = await response.json();
```

```
  return data;
};
```

TypeScript-Konzepte hier:

- ✅ Funktions-Rückgabetyp: Promise<Student[]>, Promise
- ✅ Array-Typisierung: Student[]
- ✅ Omit<> Utility Type (entfernt bestimmte Properties)
- ✅ Funktionsparameter typisieren
- ✅ Explizite Typisierung von Response-Daten

Schritt 6: StudentList Komponente

Datei erstellen: `src/components/StudentList.tsx`

```
import type { Student } from "../types/student";

// ✅ TypeScript Konzept: type für Component Props
// ✅ Konzept: Array-Typisierung
// ✅ Konzept: Optionaler Parameter
type StudentListProps = {
  students: Student[];      // Array von Students
  error?: string;           // Optional: Fehlermeldung
};

// ✅ Konzept: Funktionsparameter mit Props-Type
const StudentList = ({ students, error }: StudentListProps) => {

  // ✅ Konzept: Optionale Werte prüfen
  if (error) {
    return <div className="error">Fehler: {error}</div>;
  }

  if (students.length === 0) {
    return <div className="info">Keine Studenten vorhanden</div>;
  }

  return (
    <div className="student-list">
      <h2>Studenten-Liste</h2>
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Note</th>
          </tr>
        </thead>
        <tbody>
          {/* ✅ Konzept: Array wird gemappt, TypeScript kennt den Typ von
student */}
```

```
            {students.map((student: Student) => (
              <tr key={student._id}>
                <td>{student.name}</td>
                <td>{student.grade}</td>
              </tr>
            ))}
          </tbody>
        </table>
      </div>
    );
};

export default StudentList;
```

TypeScript-Konzepte hier:

- ✅ Props-Type Definition
- ✅ Array-Typisierung `Student[]`
- ✅ Optionaler Parameter `error?`
- ✅ Typisierung in `.map()`

Schritt 7: AddStudentForm Komponente

Datei erstellen: `src/components/AddStudentForm.tsx`

```
import {useState} from "react";
import type { FormEvent, ChangeEvent } from "react";

// ✅ TypeScript Konzept: type für Component Props
// ✅ Konzept: Funktions-Typisierung
type AddStudentFormProps = {
  onStudentAdded: () => void;  // Callback-Funktion ohne Rückgabewert
};

// ✅ Konzept: type für Form-Daten
type FormData = {
  name: string;
  grade: string;  // String, weil Input-Felder immer Strings liefern
};

const AddStudentForm = ({ onStudentAdded }: AddStudentFormProps) => {

  // ✅ Konzept: useState mit Objekt-Typisierung
  const [formData, setFormData] = useState<FormData>({
    name: "",
    grade: ""
  });

  const [isSubmitting, setIsSubmitting] = useState<boolean>(false);
```

```
  // ✅ Konzept: Event-Handler Typisierung mit ChangeEvent
  const handleInputChange = (e: ChangeEvent<HTMLInputElement>): void => {
    const { name, value } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: value
    }));
  };

  // ✅ Konzept: Event-Handler Typisierung mit FormEvent
  const handleSubmit = async (e: FormEvent<HTMLFormElement>): Promise<void>
=> {
    e.preventDefault();
    setIsSubmitting(true);

    try {
      // ✅ Konzept: String zu Number konvertieren mit expliziter
Typisierung
      const gradeAsNumber: number = parseFloat(formData.grade);

      const response = await fetch("http://localhost:3000/api/students", {
        method: "POST",
        headers: {
          "Content-Type": "application/json"
        },
        body: JSON.stringify({
          name: formData.name,
          grade: gradeAsNumber  // Als Number senden
        })
      });

      if (response.ok) {
        // Formular zurücksetzen
        setFormData({ name: "", grade: "" });
        // Parent-Komponente informieren
        onStudentAdded();
      } else {
        alert("Fehler beim Hinzufügen des Studenten");
      }
    } catch (error: unknown) {
      // ✅ Konzept: unknown Type für Error-Handling
      if (error instanceof Error) {
        alert(`Fehler: ${error.message}`);
      } else {
        alert("Ein unbekannter Fehler ist aufgetreten");
      }
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <div className="add-student-form">
      <h2>Neuen Student hinzufügen</h2>
```

```
        <form onSubmit={handleSubmit}>
          <div>
            <label htmlFor="name">Name:</label>
            <input
              type="text"
              id="name"
              name="name"
              value={formData.name}
              onChange={handleInputChange}
              required
            />
          </div>

          <div>
            <label htmlFor="grade">Note (1-6):</label>
            <input
              type="number"
              id="grade"
              name="grade"
              value={formData.grade}
              onChange={handleInputChange}
              min="1"
              max="6"
              step="0.1"
              required
            />
          </div>

          <button type="submit" disabled={isSubmitting}>
            {isSubmitting ? "Wird hinzugefügt..." : "Hinzufügen"}
          </button>
        </form>
      </div>
    );
};

export default AddStudentForm;
```

TypeScript-Konzepte hier:

- ✅ Props-Type mit Funktions-Typisierung
- ✅ Objekt-Type für FormData
- ✅ `useState` mit expliziten Typen
- ✅ Event-Handler: `ChangeEvent`, `FormEvent`
- ✅ `unknown` Type im Error-Handling
- ✅ Explizite Typ-Konvertierung (`parseFloat`)
- ✅ `Promise<void>` Rückgabetyp

Schritt 8: App Komponente

Datei anpassen: `src/App.tsx`

```tsx
import { useState, useEffect } from "react";
import StudentList from "./components/StudentList";
import AddStudentForm from "./components/AddStudentForm";
import type { Student, LoadingStatus } from "./types/student";
import { fetchStudents } from "./api/students";
import "./App.css";

function App() {

  // ✅ TypeScript Konzept: useState mit Array-Typisierung
  const [students, setStudents] = useState<Student[]>([]);

  // ✅ Konzept: Union Type für Status
  const [status, setStatus] = useState<LoadingStatus>('idle');

  // ✅ Konzept: Optionaler Wert
  const [error, setError] = useState<string | undefined>(undefined);

  // ✅ Konzept: Async Funktion typisieren
  const loadStudents = async (): Promise<void> => {
    setStatus('loading');
    setError(undefined);

    try {
      const data: Student[] = await fetchStudents();
      setStudents(data);
      setStatus('idle');
    } catch (err: unknown) {
      // ✅ Konzept: unknown Type prüfen
      if (err instanceof Error) {
        setError(err.message);
      } else {
        setError("Fehler beim Laden der Studenten");
      }
      setStatus('error');
    }
  };

  // Beim ersten Laden
  useEffect(() => {
    loadStudents();
  }, []);

  return (
    <div className="app">
      <h1>📚 Studenten-Notenverwaltung</h1>

      {status === 'loading' && <div className="loading">Laden...</div>}

      <AddStudentForm onStudentAdded={loadStudents} />

      <StudentList students={students} error={error} />
    </div>
```

```
  );
}

export default App;
```

TypeScript-Konzepte hier:

- ✅ `useState` mit Array `Student[]`
- ✅ Union Type `LoadingStatus`
- ✅ Optionale Werte: `string | undefined`
- ✅ Async Funktionen: `Promise<void>`
- ✅ `unknown` Type-Checking
- ✅ Explizite Typisierung von Variablen

Schritt 9: Styling (Optional aber schön)

Datei anpassen: `src/App.css`

```
.app {
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
  font-family: Arial, sans-serif;
}

h1 {
  text-align: center;
  color: #333;
}

.add-student-form {
  background: #f5f5f5;
  padding: 20px;
  border-radius: 8px;
  margin-bottom: 30px;
}

.add-student-form h2 {
  margin-top: 0;
}

.add-student-form form div {
  margin-bottom: 15px;
}

.add-student-form label {
  display: block;
  margin-bottom: 5px;
  font-weight: bold;
}
```

```css
.add-student-form input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ddd;
  border-radius: 4px;
  font-size: 14px;
  box-sizing: border-box;
}

.add-student-form button {
  background: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 16px;
}

.add-student-form button:hover:not(:disabled) {
  background: #0056b3;
}

.add-student-form button:disabled {
  background: #ccc;
  cursor: not-allowed;
}

.student-list {
  background: white;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.student-list h2 {
  margin-top: 0;
  border-bottom: 2px solid #007bff;
  padding-bottom: 10px;
}

table {
  width: 100%;
  border-collapse: collapse;
}

th {
  background: #007bff;
  color: white;
  padding: 12px;
  text-align: left;
}
```

```
td {
  padding: 12px;
  border-bottom: 1px solid #ddd;
}

tr:hover {
  background: #f5f5f5;
}

.error {
  background: #f8d7da;
  color: #721c24;
  padding: 15px;
  border-radius: 4px;
  margin: 20px 0;
}

.info {
  background: #d1ecf1;
  color: #0c5460;
  padding: 15px;
  border-radius: 4px;
  margin: 20px 0;
}

.loading {
  text-align: center;
  padding: 20px;
  font-size: 18px;
  color: #007bff;
}
```

Schritt 10: src/index.css anpassen

Minimales `src/index.css`:

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  background: #f0f0f0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto',
sans-serif;
}
```

Schritt 11: Frontend starten

```
npm run dev
```

Schritt 12: Beide Server gleichzeitig laufen lassen

Terminal 1 (Backend):

```
cd student-grades/backend
npm start
```

Terminal 2 (Frontend):

```
cd student-grades/frontend
npm run dev
```

Schritt 13: Testen!

1. Öffne `http://localhost:5173/` im Browser
2. Füge einen Student hinzu (z.B. "Anna Schmidt", Note: 1.5)
3. Klicke auf "Hinzufügen"
4. Der Student sollte in der Liste erscheinen

✅ Projekt komplett!

Was habt gelernt?

Backend:

- ✅ `type` Definitionen
- ✅ `string`, `number` Typen
- ✅ Optionale Parameter `?`
- ✅ `unknown` für Error-Handling
- ✅ Funktions-Typisierung
- ✅ Promise-Typen

Frontend:

- ✅ `type` für Props, State, Daten
- ✅ **Union Types** (`LoadingStatus`)
- ✅ Array-Typisierung `Student[]`
- ✅ Event-Handler Typen (`ChangeEvent`, `FormEvent`)
- ✅ `useState` mit Typen
- ✅ `unknown` Type-Checking
- ✅ Optionale Props

Deployment-Anleitung für Render.com

Teil 1: Backend auf Render deployen

Schritt 1: Die package.json anpassen

- Diese Zeile hinzufügen: `"start": "node dist/server.js"`

Schritt 2: GitHub Repository erstellen

1. Gehe zu https://github.com
2. Erstelle ein neues Repository: `student-grades-app`
3. Mache es **public**

Schritt 3: Code zu GitHub pushen

Im Hauptverzeichnis `student-grades/`:

```
# Git initialisieren (falls noch nicht geschehen)
git init

# .gitignore erstellen
```

Datei erstellen: `.gitignore` (im Hauptverzeichnis)

```
# Dependencies
node_modules/
**/node_modules/

# Environment
.env
**/.env

# Build
dist/
**/dist/
build/
**/build/

# Logs
*.log
npm-debug.log*

# OS
```

```
.DS_Store
Thumbs.db
```

```
# Dateien hinzufügen
git add .

# Commit
git commit -m "Initial commit"

# Remote hinzufügen (ersetze USERNAME und REPO)
git remote add origin https://github.com/USERNAME/student-grades-app.git

# Push
git push -u origin main
```

Schritt 4: Backend Web Service erstellen

1. Gehe zu https://render.com
2. Im Render Dashboard: Klicke auf **"New +"**
3. Wähle **"Web Service"**
4. Verbinde dein GitHub Repository `student-grades-app`
5. Klicke auf **"Connect"** neben dem Repository

Schritt 5: Backend konfigurieren

Fülle die Felder aus:

Feld	Wert
Name	`student-grades-backend`
Region	Frankfurt (EU Central)
Branch	`main`
Root Directory	`backend`
Runtime	`Node`
Build Command	`npm install && npm run build`
Start Command	`npm start`
Instance Type	`Free`

Schritt 6: Environment Variablen setzen

Scrolle runter zu **"Environment Variables"** und füge hinzu:

Key	Value
MONGO_URL	mongodb+srv://student_admin:DEIN_PASSWORT@cluster0.xxxxx.mongodb.net/?retryWrites=true&w=majority
DATABASE	student_grades
PORT	3000
NODE_ENV	production

⚠ **Wichtig:** Ersetze die MongoDB-URL mit deiner echten URL von Atlas!

Schritt 7: Backend deployen

1. Klicke unten auf **"Create Web Service"**
2. Warte, bis das Deployment fertig ist (2-5 Minuten)
3. Render zeigt den Status und Logs an
4. Nach erfolgreichem Deployment siehst du die URL, z.B.:

```
https://student-grades-backend.onrender.com
```

Schritt 8: Backend testen

```
# Teste die API (ersetze die URL mit deiner)
curl https://student-grades-backend.onrender.com/api/students
```

Erwartete Antwort: `[]` (leeres Array, weil noch keine Studenten vorhanden)

Teil 2: Frontend auf Render deployen

Schritt 1: Frontend API-URL anpassen

Datei ändern: `frontend/src/api/students.ts`

Vorher:

```
const API_URL = "http://localhost:3000/api/students";
```

Nachher:

```
const API_URL = import.meta.env.VITE_API_URL ||
"http://localhost:3000/api/students";
```

Schritt 2: Environment-Datei für Produktion

Datei erstellen: `frontend/.env.production`

BEACHTEN - Im Gegensatz zu einer `.env` wird die `.env.production` comitted und zu Github hochgeladen

- In Render ist dann keine Environment Variable nötig (Vite liest .env.production)

```
VITE_API_URL=https://student-grades-backend.onrender.com/api/students
```

⚠ **Wichtig:** Ersetze die URL mit deiner echten Backend-URL von Render!

Schritt 3: Änderungen committen

```
git add .
git commit -m "Add production environment config"
git push
```

Schritt 4: Frontend Static Site erstellen

1. Im Render Dashboard: Klicke auf **"New +"**
2. Wähle **"Static Site"**
3. Wähle dein Repository `student-grades-app`
4. Klicke auf **"Connect"**

Schritt 5: Frontend konfigurieren

Fülle die Felder aus:

Feld	Wert
Name	`student-grades-frontend`
Branch	`main`
Root Directory	`frontend`
Build Command	`npm install && npm run build`
Publish Directory	`dist`

Schritt 6: Environment Variablen setzen

Unter **"Environment Variables":**

Key	Value
`VITE_API_URL`	`https://student-grades-backend.onrender.com/api/students`

⚠ **Wichtig:** Ersetze mit deiner echten Backend-URL!

Schritt 7: Frontend deployen

1. Klicke auf **"Create Static Site"**
2. Warte auf das Deployment (2-5 Minuten)
3. Nach erfolgreichem Deployment bekommst du eine URL, z.B.:

```
https://student-grades-frontend.onrender.com
```

Teil 4: CORS im Backend konfigurieren

Problem: CORS-Fehler

- **BEACHTEN** Diesen Teil nur anpassen, wenn es CORS-Probleme gibt. Wenn die App funktioniert, nichts machen!

Wenn das Frontend versucht, das Backend anzusprechen, könnte ein CORS-Fehler auftreten.

Lösung: CORS für Render konfigurieren

Datei ändern: `backend/src/server.ts`

Vorher:

```
app.use(cors());
```

Nachher:

```
import cors from "cors";

// CORS Konfiguration
const corsOptions = {
  origin: process.env.FRONTEND_URL || "http://localhost:5173",
  credentials: true
};

app.use(cors(corsOptions));
```

Environment Variable hinzufügen

In Render → Backend Web Service → Environment:

Key	Value
`FRONTEND_URL`	`https://student-grades-frontend.onrender.com`

⚠ Ersetze mit deiner Frontend-URL!

Alternativ für mehrere Frontends:

```
const allowedOrigins = [
  "http://localhost:5173",
  "https://student-grades-frontend.onrender.com"
];

const corsOptions = {
  origin: (origin: string | undefined, callback: Function) => {
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error("Not allowed by CORS"));
    }
  },
  credentials: true
};

app.use(cors(corsOptions));
```

Änderungen pushen

```
git add .
git commit -m "Configure CORS for production"
git push
```

Render wird automatisch neu deployen!

Teil 5: Testen & Troubleshooting

Schritt 1: Vollständiger Test

1. Öffne deine Frontend-URL: `https://student-grades-frontend.onrender.com`
2. Füge einen Student hinzu

3. Überprüfe, ob er in der Liste erscheint

Schritt 2: Logs überprüfen (bei Fehlern)

Backend Logs:

1. Gehe zu Render Dashboard
2. Klicke auf `student-grades-backend`
3. Klicke auf **"Logs"**
4. Suche nach Fehlern

Frontend Logs:

1. Gehe zu Render Dashboard
2. Klicke auf `student-grades-frontend`
3. Klicke auf **"Events"** oder **"Logs"**

Browser Console:

1. Öffne die Frontend-URL
2. Drücke `F12` (Developer Tools)
3. Gehe zu **Console**
4. Suche nach CORS- oder Netzwerkfehlern

Häufige Probleme & Lösungen

× Problem: CORS-Fehler im Browser

Lösung:

- Überprüfe `FRONTEND_URL` im Backend
- Stelle sicher, dass CORS richtig konfiguriert ist
- Redeploy Backend nach CORS-Änderungen

× Problem: Frontend kann Backend nicht erreichen

Lösung:

- Überprüfe `VITE_API_URL` im Frontend
- Stelle sicher, dass Backend online ist (grüner Status in Render)
- Teste Backend-URL direkt im Browser: `https://DEINE-URL/api/students`

× Problem: Free Tier schläft ein

Hinweis:

- Render's Free Tier schläft nach 15 Minuten Inaktivität ein
- Beim nächsten Request dauert es ~30 Sekunden zum Aufwachen
- Lösung: Upgrade auf bezahlten Plan oder akzeptieren

Teil 6: Zusammenfassung

🎉 Was haben wir deployed?

✅ **MongoDB Atlas**: Kostenlose Cloud-Datenbank
✅ **Backend**: Node.js + Express + TypeScript auf Render
✅ **Frontend**: React + TypeScript + Vite auf Render
✅ **CORS**: Korrekt konfiguriert für Produktion
✅ **Environment Variables**: Sicher getrennt von Code

🔄 Updates deployen

Änderungen am Code?

```
git add .
git commit -m "Beschreibung der Änderung"
git push
```

Render erkennt automatisch den Push und deployed neu!

✅ Fertig!
