

002-Bibliothek

Backend Aufgabe
Anforderungsniveau: Leicht

1 Ziel der Aufgabe

Ziel dieser Aufgabe ist es, Ihre Kenntnisse in praxistauglicher Backend-Entwicklung mit Node.js und TypeScript zu prüfen. Sie erstellen eine REST-API zur Verwaltung einer *Buch-Bibliothek* inklusive CRUD-Funktionalität. Die Daten werden aktuell *in Memory* gespeichert—optional kann später eine JSON-Datei als Persistenz verwendet werden.

2 Aufgabenbeschreibung

2.1 Projektaufbau

1. Legen Sie ein neues Node.js-Projekt an und installieren Sie Abhängigkeiten:

```
mkdir backend-book
cd backend-book
npm init -y
npm install express uuid
npm install -D typescript ts-node-dev @types/express @types/node @types/uuid
```

2. Erzeugen Sie eine `tsconfig.json` (z.B. via `npx tsc -init`) mit:
3. Strukturieren Sie den Quellcode:

```
backend-book/
  src/
    controllers/
      bookController.ts
    models/
      book.ts
    routes/
      bookRoutes.ts
    services/
      bookService.ts
  data/                                % optional: JSON-Datei für Persistenz
    books.json
  index.ts
package.json
tsconfig.json
.gitignore
README.md
```

2.2 Ressourcenmodell „Book“

Erstellen Sie in `src/models/book.ts` das Interface `Book`:

```
export interface Book {
  id: string;           % UUID
  title: string;        % mind. 3 Zeichen
  author: string;       % mind. 3 Zeichen
  isbn: string;         % 10-13 Ziffern
  publishedDate: string; % ISO (YYYY-MM-DD)
  available: boolean;   % Verfügbar oder nicht
  createdAt: Date;      % Erstellungsdatum
  updatedAt: Date;      % Letzte Aktualisierung
}
```

Neue Bücher erhalten automatisch `id = uuidv4()`, `createdAt = new Date()`, `updatedAt = new Date()`.

2.3 Service-Schicht

Implementieren Sie in `src/services/bookService.ts` Funktionen zur Verwaltung der Bücher *in Memory*.

Optional: JSON-Datei als Persistenz In `src/data/books.json` starten Sie mit `[]`. Beim Start laden Sie die Datei via `fs.readFileSync`; nach jedem CRUD-Vorgang schreiben Sie synchron zurück via `fs.writeFileSync`. Achten Sie auf Fehlerbehandlung bei Dateioperationen.

2.4 Routing

In `src/routes/bookRoutes.ts` legen Sie die Routen fest:

```
import { Router } from 'express';
import {
  fetchAllBooks, fetchBookById,
  createNewBook, updateExistingBook, deleteBookById
} from '../controllers/bookController';

const router = Router();

router.get('/books', fetchAllBooks);
router.get('/books/:id', fetchBookById);
router.post('/books', createNewBook);
router.put('/books/:id', updateExistingBook);
router.delete('/books/:id', deleteBookById);

export default router;
```

2.5 Server-Setup

In `src/index.ts` initialisieren Sie Express:

```
import express, { Request, Response, NextFunction } from 'express';
import bookRoutes from './routes/bookRoutes';

const app = express();
const PORT = process.env.PORT || 4000;

app.use(express.json()); % Body-Parser für JSON

// API-Routen
app.use('/api', bookRoutes);

// Globaler Error-Handler (unten)
app.use((err: any, req: Request, res: Response, next: NextFunction) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Interner Serverfehler' });
});

// 404-Handler
app.use((req: Request, res: Response) => {
  res.status(404).json({ message: 'Ressource nicht gefunden' });
});

app.listen(PORT, () => {
  console.log('Server läuft: http://localhost:${PORT}');
});
```

3 Anforderungen

- **TypeScript-Typisierung:** Kein Einsatz von `any`. Alle Interfaces und Funktionen sind strikt typisiert.
- **CRUD-Endpunkte:**
 - GET `/api/books` 200 OK + JSON-Array
 - GET `/api/books/:id` 200 OK + JSON oder 404 Not Found
 - POST `/api/books` 201 Created + JSON oder 400 Bad Request
 - PUT `/api/books/:id` 200 OK + JSON oder 400 Bad Request bzw. 404 Not Found
 - DELETE `/api/books/:id` 204 No Content oder 404 Not Found
- **Validierung:**
 - `title` und `author` mind. 3 Zeichen.
 - `isbn` 10–13 Ziffern.
 - `publishedDate` ISO-Format (YYYY-MM-DD) (Prüfung via `Date.parse`).
 - `available` muss Boolean sein.
 - Bei Fehlern 400 Bad Request mit:
 - {

```

    "message": "Ungültige Eingabedaten",
    "errors": ["...", "..."]
  }

```

- **Fehler- und Statuscodes:**
 - 201 Created für erfolgreiches Anlegen.
 - 400 Bad Request bei Validierungsfehlern.
 - 404 Not Found wenn Ressource nicht existiert.
 - 200 OK für erfolgreiche Abfragen/Aktualisierung.
 - 204 No Content für erfolgreiches Löschen.
 - 500 Internal Server Error im globalen Error-Handler.
- **Saubere Code-Organisation:**
 - Trennung von *Models*, *Services*, *Controllers*, *Routes*.
 - Keine Business-Logik in Controllern.
- **Dokumentation:**
 - README.md mit:
 - * **Installation:** `npm install`
 - * **Startskripte:**
 - Dev: `npm run dev` (z. B. mit `ts-node-dev src/index.ts`)
 - Prod: `npm run build && node dist/index.js`
 - * **API-Beschreibung** aller Endpunkte (z. B. cURL-Beispiele).

3.1 Optionale Erweiterungen (Kann-Kriterien)

1. JSON-Datei als Persistenz

- Legen Sie `src/data/books.json` an (Initial: `[]`).
- Beim Start: Datei einlesen (`fs.readFileSync`).
- Nach jeder Änderung: Datei aktualisieren (`fs.writeFileSync`).
- Bewältigen Sie Race-Conditions und Fehler.

2. Datenbankbindung (z. B. MySQL)

- Integrieren Sie MySQL (z. B. mit `TypeORM` oder `Prisma`).
- Definieren Sie ein Schema und führen Sie Migrations beim Start durch.
- Passen Sie die Service-Funktionen an, um die DB zu nutzen.

3. Swagger/OpenAPI-Dokumentation

- Installieren Sie `swagger-jsdoc` und `swagger-ui-express`.
- Dokumentieren Sie alle Endpunkte in JSDoc oder YAML/JSON.
- Stellen Sie die Swagger-UI unter `/api-docs` bereit.

4. CORS-Konfiguration

- Installieren Sie `cors` und erlauben Sie nur bestimmte Ursprünge, z. B. `http://localhost:3000`
- ```

import cors from 'cors';
app.use(cors({ origin: 'http://localhost:3000' }));

```

## 5. Unit- und Integrationstests

- Schreiben Sie Unit-Tests für `bookService` (z. B. mit Jest).
- Schreiben Sie Integrationstests mit `supertest`, um die API-Endpoints zu prüfen.

Viel Erfolg bei der Umsetzung!

*Stand: 31. Mai 2025*