By Anton Kucherenko

Node.js Backend For Educational Course Management System

Project Overview

This project aims to develop a backend system using Node.js that effectively manages educational courses. It will facilitate student&course creation, student enrollments, assignment submissions and grading, all through a RESTful API interfacing with a SQLite database.

Used Technologies

- Node.js - Core technology for building the server-side application.
- Express.js - Used to create and manage server, routes, and API endpoints.
- SQLite - Database engine for storing and retrieving application data.
- Npm (Node package manager)
- Sqlite3 - Facilitates interactions between the Node.js application and the SQLite database.
- Nodemon - Used in development for automatically restarting the server upon code changes.
- Fs - Used for reading and writing files, such as SQL scripts for database initialization and data manipulation.
- Express-validator - Used for request data validation and sanitization.
- Postman - Used for testing and documenting the API.

API Structure

- **General API Information:**

Base URL: http://localhost:3000/api

- **Standard Endpoints for Each Entity:**

Entities: students, courses, assignments, enrollments, submissions, grades

- GET /api/[entity] - Retrieve all entities.

- POST /api/[entity] - Create a new entity.

- PATCH /api/[entity]/:id - Update an existing entity.

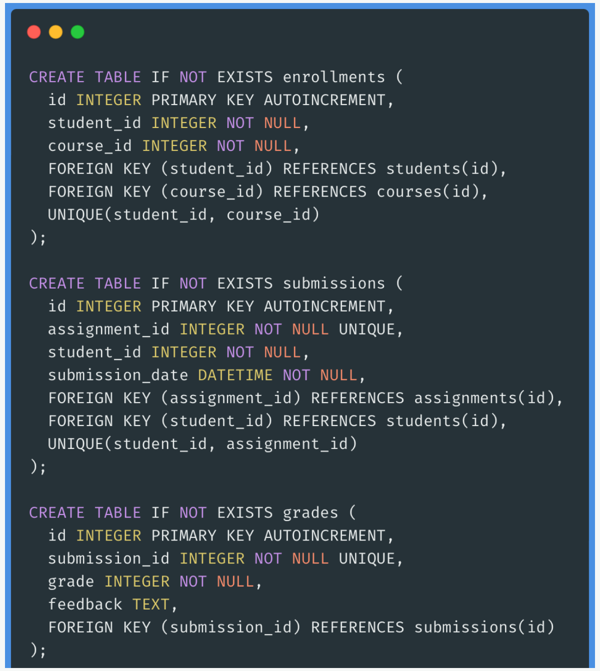
- DELETE /api/[entity]/:id - Delete an entity.

- **Unique Endpoints for Assignments:**

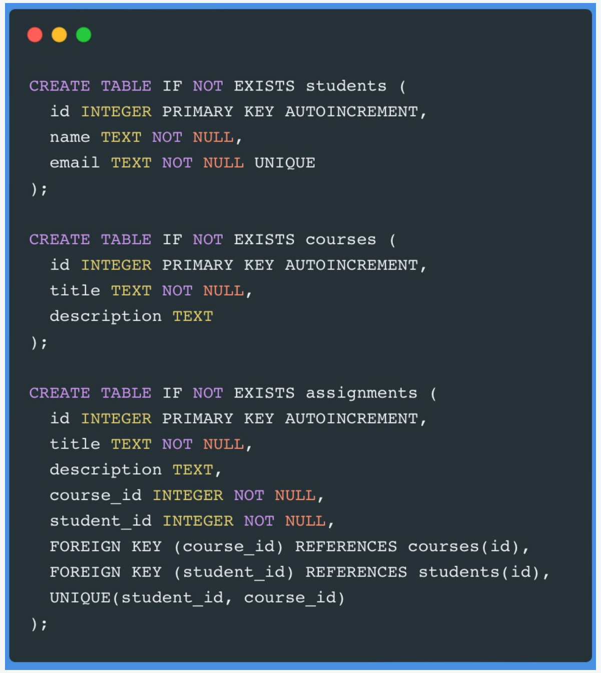
- GET /api/assignments/course/:courseId – Retrieve assignments for a specific course.

- GET /api/assignments/assignments?course_id=1&student_id=1 - Retrieve assignments filtered by both course_id and student_id.

Database Schema



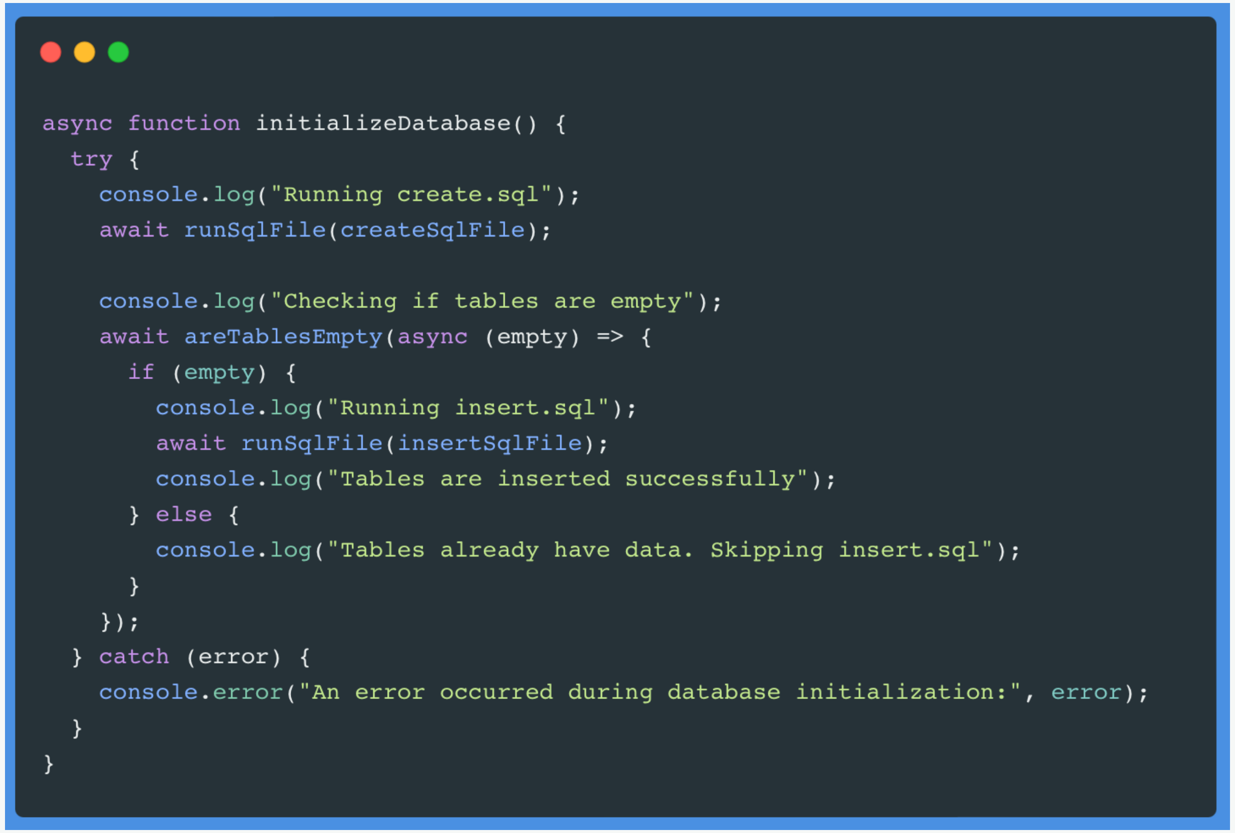
```
CREATE TABLE IF NOT EXISTS enrollments (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  student_id INTEGER NOT NULL,
  course_id INTEGER NOT NULL,
  FOREIGN KEY (student_id) REFERENCES students(id),
  FOREIGN KEY (course_id) REFERENCES courses(id),
  UNIQUE(student_id, course_id)
);

CREATE TABLE IF NOT EXISTS submissions (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  assignment_id INTEGER NOT NULL UNIQUE,
  student_id INTEGER NOT NULL,
  submission_date DATETIME NOT NULL,
  FOREIGN KEY (assignment_id) REFERENCES assignments(id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  UNIQUE(student_id, assignment_id)
);

CREATE TABLE IF NOT EXISTS grades (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  submission_id INTEGER NOT NULL UNIQUE,
  grade INTEGER NOT NULL,
  feedback TEXT,
  FOREIGN KEY (submission_id) REFERENCES submissions(id)
);
```



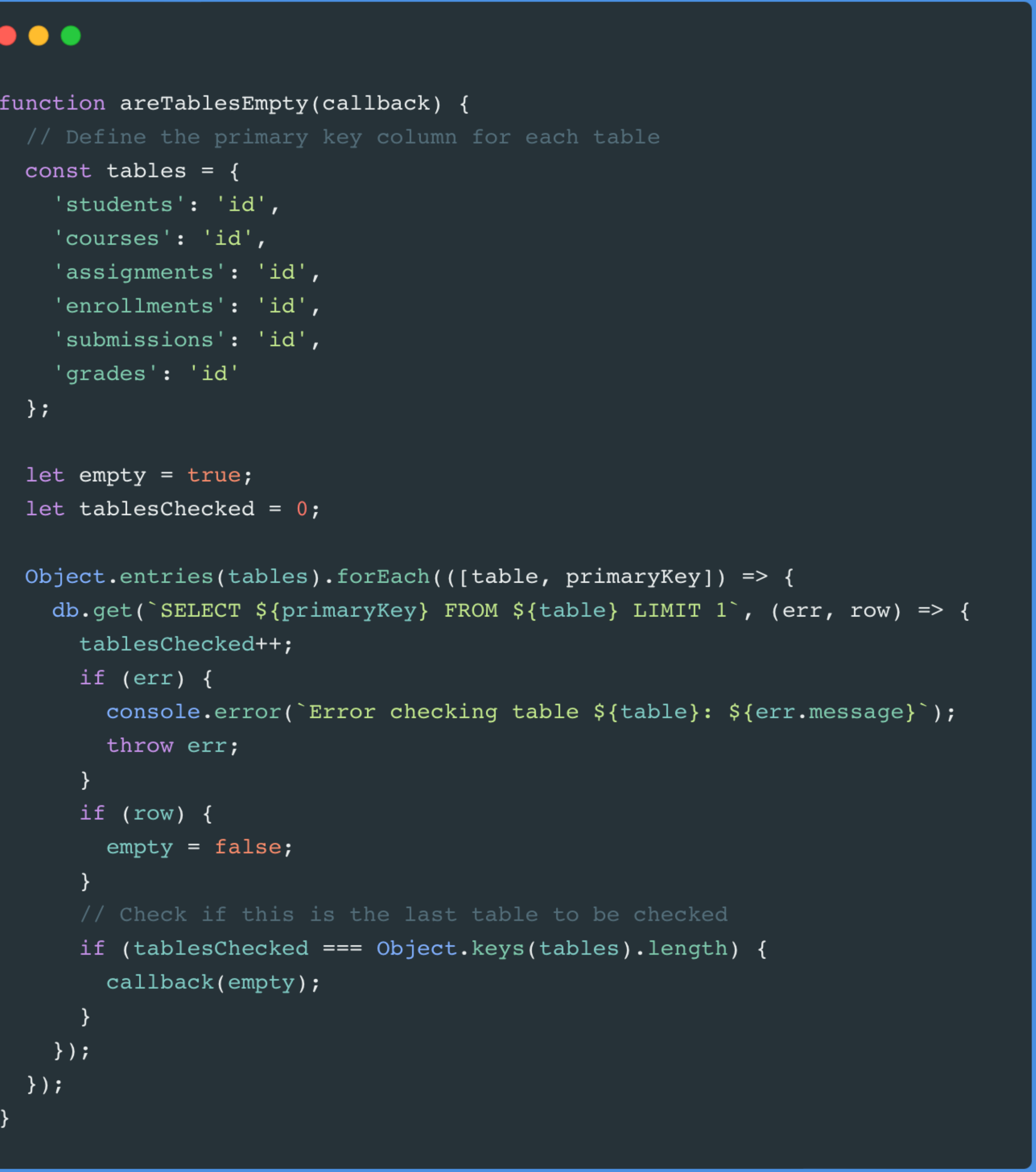
```
CREATE TABLE IF NOT EXISTS students (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  email TEXT NOT NULL UNIQUE
);

CREATE TABLE IF NOT EXISTS courses (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  description TEXT
);

CREATE TABLE IF NOT EXISTS assignments (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  description TEXT,
  course_id INTEGER NOT NULL,
  student_id INTEGER NOT NULL,
  FOREIGN KEY (course_id) REFERENCES courses(id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  UNIQUE(student_id, course_id)
);
```

Database Initialization

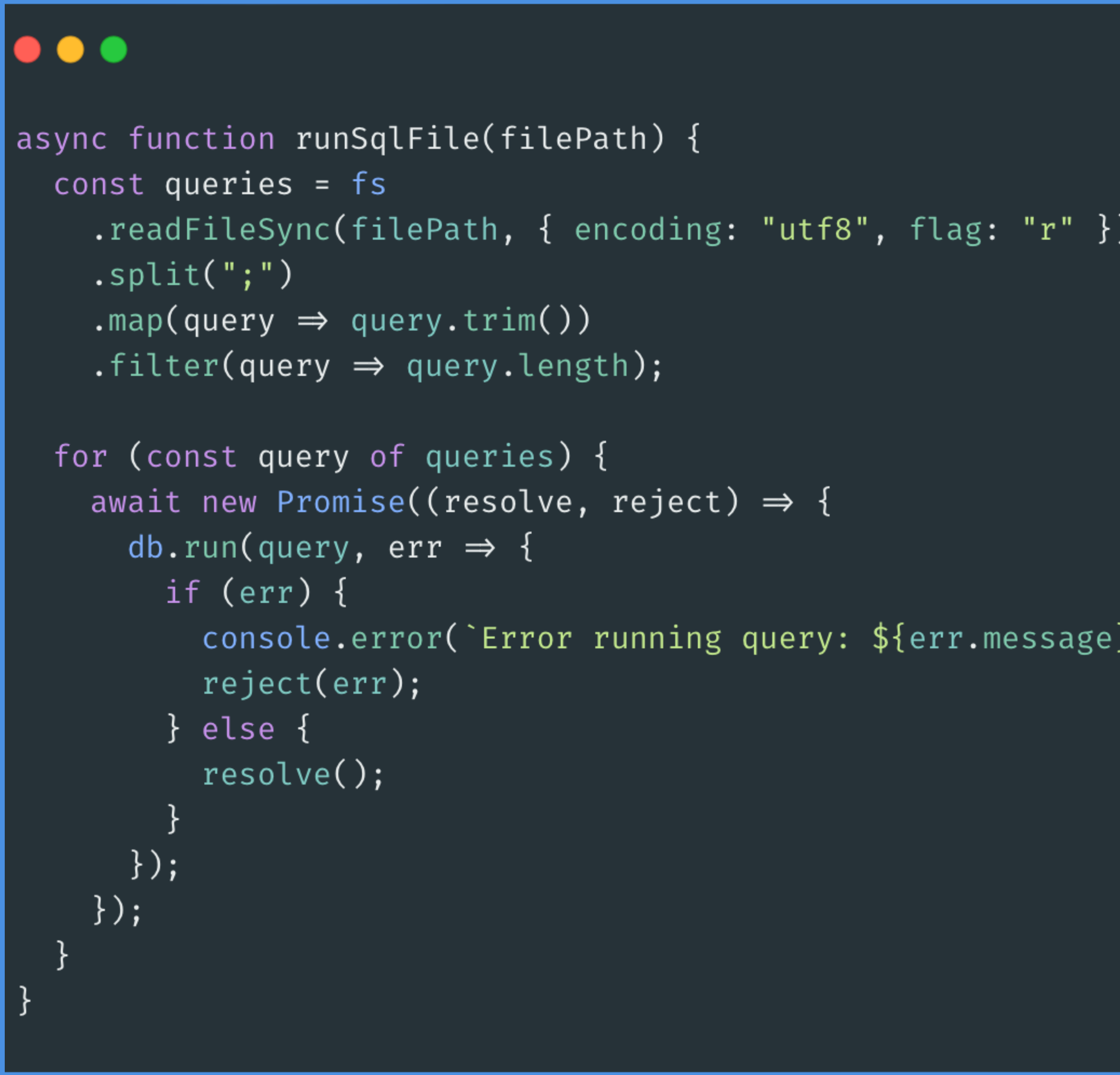
- The initializeDatabase() function is crucial for setting up the database schema and inserting initial data. It checks if the tables are empty and populates them if necessary.
- The runSqlFile() function reads and executes SQL commands from .sql files, ensuring that the database structure and initial dataset are correctly established.
- This process is vital for the smooth functioning of the API, as it prepares the database with the necessary structure and data for the API endpoints to interact with.



```
async function initializeDatabase() {
  try {
    console.log("Running create.sql");
    await runSqlFile(createSqlFile);

    console.log("Checking if tables are empty");
    await areTablesEmpty(async (empty) => {
      if (empty) {
        console.log("Running insert.sql");
        await runSqlFile(insertSqlFile);
        console.log("Tables are inserted successfully");
      } else {
        console.log("Tables already have data. Skipping insert.sql");
      }
    });
  } catch (error) {
    console.error("An error occurred during database initialization:", error);
  }
}
```



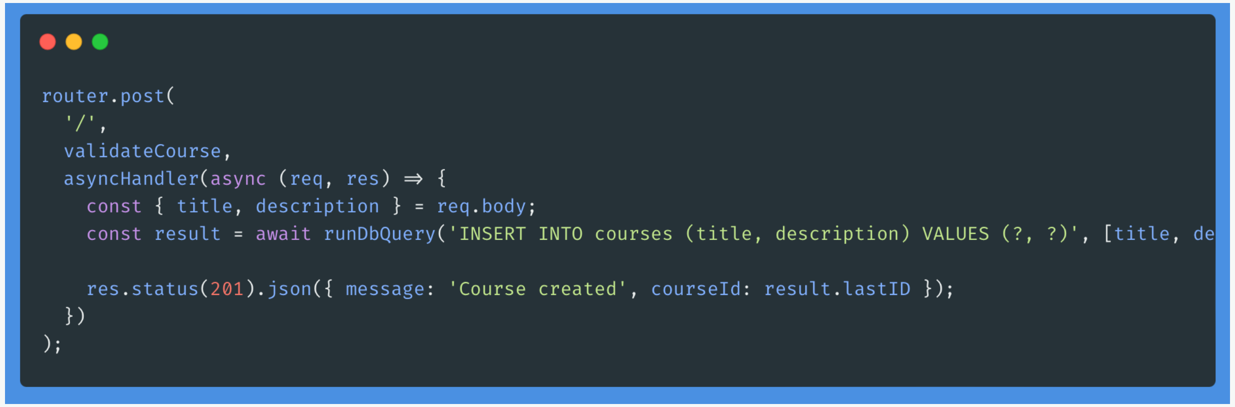
```
function areTablesEmpty(callback) {
  // Define the primary key column for each table
  const tables = {
    'students': 'id',
    'courses': 'id',
    'assignments': 'id',
    'enrollments': 'id',
    'submissions': 'id',
    'grades': 'id'
  };

  let empty = true;
  let tablesChecked = 0;

  Object.entries(tables).forEach(([table, primaryKey]) => {
    db.get(`SELECT ${primaryKey} FROM ${table} LIMIT 1`, (err, row) => {
      tablesChecked++;
      if (err) {
        console.error(`Error checking table ${table}: ${err.message}`);
        throw err;
      }
      if (row) {
        empty = false;
      }
      // Check if this is the last table to be checked
      if (tablesChecked === Object.keys(tables).length) {
        callback(empty);
      }
    });
  });
}
```



```
async function runSqlFile(filePath) {
  const queries = fs
    .readFileSync(filePath, { encoding: "utf8", flag: "r" }
    .split(";")
    .map(query => query.trim())
    .filter(query => query.length);

  for (const query of queries) {
    await new Promise((resolve, reject) => {
      db.run(query, err => {
        if (err) {
          console.error(`Error running query: ${err.message
          reject(err);
        } else {
          resolve();
        }
      });
    });
  }
}
```

API Endpoint Creation ~ Adding a Course

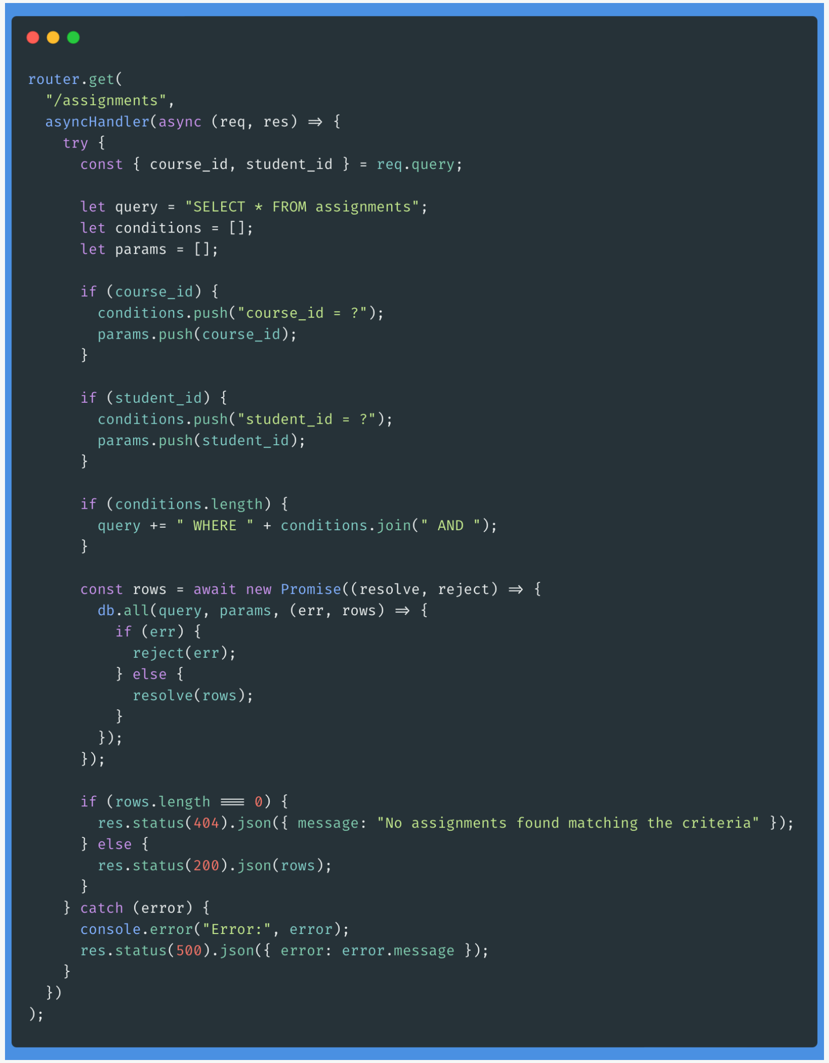
- The POST /api/courses endpoint is used to insert new course data into the database. It handles incoming requests that include course details like title and description.
- The validateCourse middleware validates the request data to ensure that all required fields are present and correctly formatted, enhancing data integrity.
- This endpoint is an example of how the application allows users to add new data to the database, showcasing the CREATE operation in CRUD.



```
router.post(
  '/',
  validateCourse,
  asyncHandler(async (req, res) => {
    const { title, description } = req.body;
    const result = await runDbQuery('INSERT INTO courses (title, description) VALUES (?, ?)', [title, de

    res.status(201).json({ message: 'Course created', courseId: result.lastID });
  })
);
```

Dynamic Query Handling in Assignments

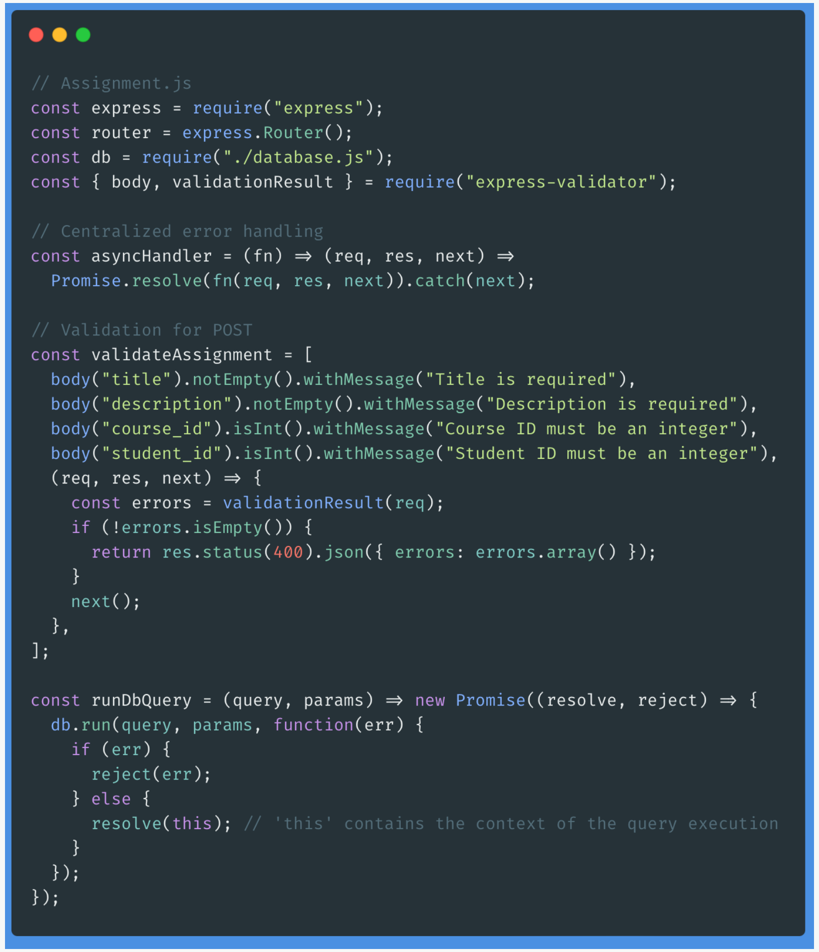
- The /api/assignments/assignments endpoint dynamically constructs an SQL query based on the provided query parameters (course_id, student_id).
- This approach allows for more complex queries and the ability to filter assignments based on different criteria.
- It demonstrates the application's capacity to provide specific data to the user, depending on their needs, showcasing an advanced level of backend functionality.



```
router.get(
  "/assignments",
  asyncHandler(async (req, res) => {
    try {
      const { course_id, student_id } = req.query;

      let query = "SELECT * FROM assignments";
      let conditions = [];
      let params = [];

      if (course_id) {
        conditions.push("course_id = ?");
        params.push(course_id);
      }

      if (student_id) {
        conditions.push("student_id = ?");
        params.push(student_id);
      }

      if (conditions.length) {
        query += " WHERE " + conditions.join(" AND ");
      }

      const rows = await new Promise((resolve, reject) => {
        db.all(query, params, (err, rows) => {
          if (err) {
            reject(err);
          } else {
            resolve(rows);
          }
        });
      });

      if (rows.length === 0) {
        res.status(404).json({ message: "No assignments found matching the criteria" });
      } else {
        res.status(200).json(rows);
      }
    } catch (error) {
      console.error("Error:", error);
      res.status(500).json({ error: error.message });
    }
  })
);
```

Error Handling in Express

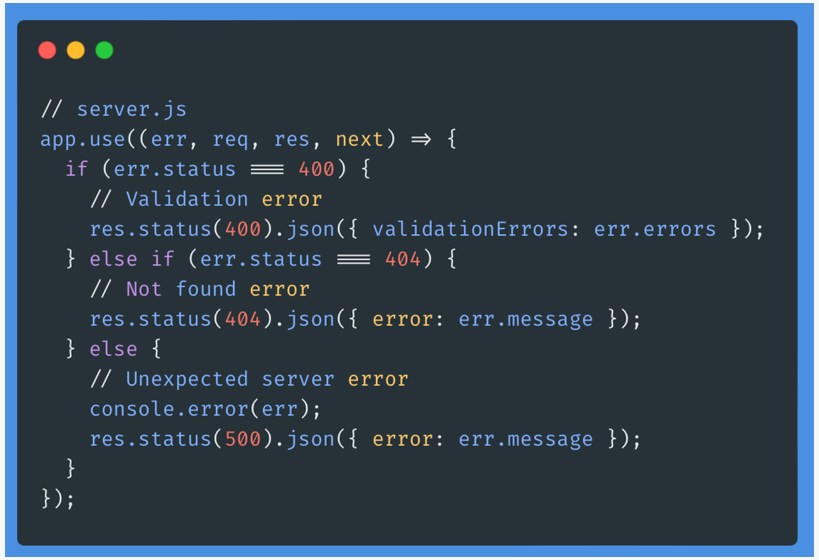
- This slide delves into the error handling mechanisms implemented in an Express.js application.
- The asyncHandler is a higher-order function wrapping asynchronous route handlers. It catches any errors that occur in an async function and passes them to the next middleware, centralizing error handling and improving code readability.
- The validateAssignment array is an example of request validation using express-validator. It checks if the request body contains valid data for creating or updating an assignment. If validation fails, it returns a 400 Bad Request response with details of the validation errors.
- The runDbQuery function is a promise-based approach to executing database queries. It ensures any SQL errors are caught and handled appropriately, avoiding unhandled exceptions and server crashes.



```
// Assignment.js
const express = require("express");
const router = express.Router();
const db = require("./database.js");
const { body, validationResult } = require("express-validator");

// Centralized error handling
const asyncHandler = (fn) => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next);

// Validation for POST
const validateAssignment = [
  body("title").notEmpty().withMessage("Title is required"),
  body("description").notEmpty().withMessage("Description is required"),
  body("course_id").isInt().withMessage("Course ID must be an integer"),
  body("student_id").isInt().withMessage("Student ID must be an integer"),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    next();
  },
];

const runDbQuery = (query, params) => new Promise((resolve, reject) => {
  db.run(query, params, function(err) {
    if (err) {
      reject(err);
    } else {
      resolve(this); // 'this' contains the context of the query execution
    }
  });
});
```

Error Handling in Express

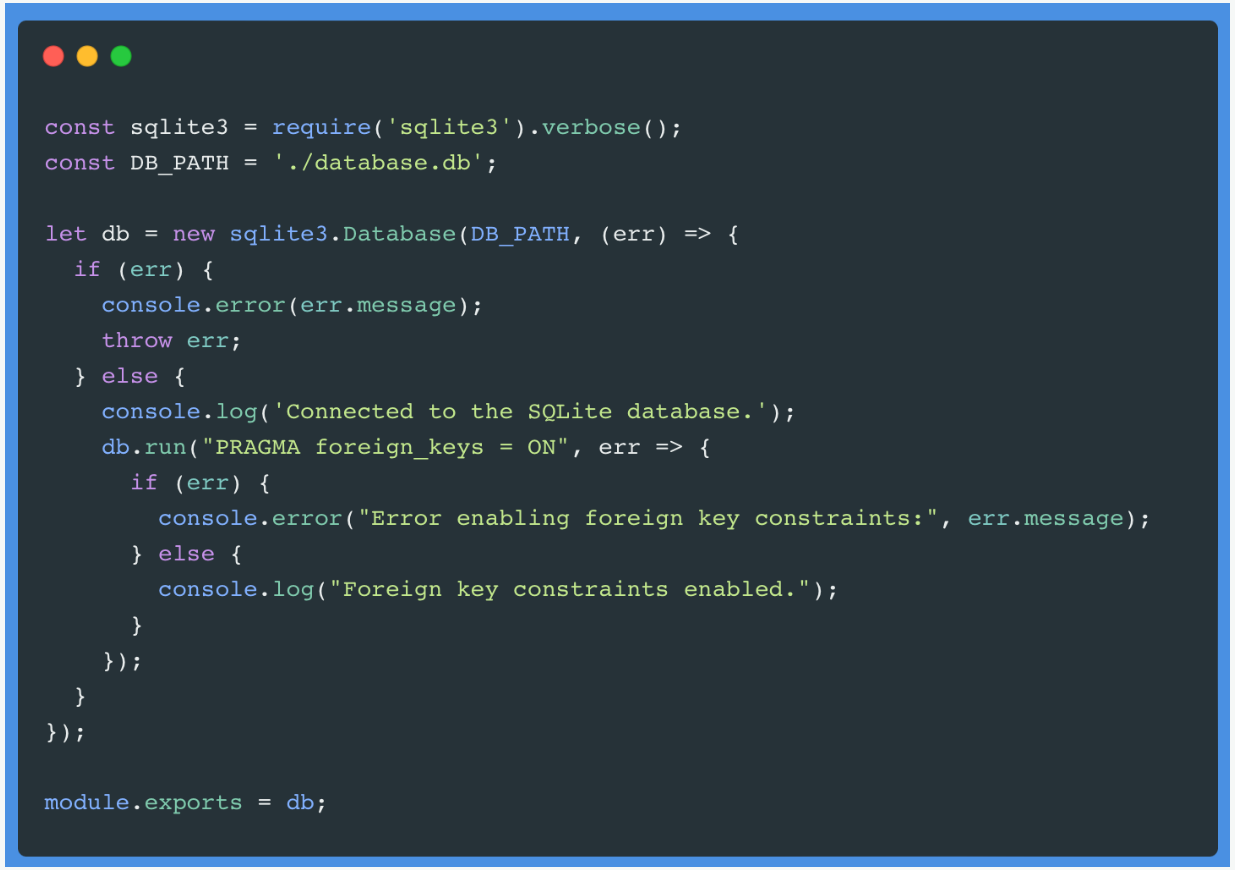
- The app.use middleware at the end captures all errors passed through next(). It differentiates between 400 Validation Error, 404 Not Found Error, and other server errors (marked as 500 Internal Server Error). This middleware logs the error and sends a structured JSON response to the client, making the API more robust and user-friendly.
- This comprehensive error handling strategy enhances the API's reliability by providing meaningful feedback to the client and ensuring that errors do not cause the server to crash unexpectedly.



```javascript
// server.js
app.use((err, req, res, next) => {
  if (err.status === 400) {
    // Validation error
    res.status(400).json({ validationErrors: err.errors });
  } else if (err.status === 404) {
    // Not found error
    res.status(404).json({ error: err.message });
  } else {
    // Unexpected server error
    console.error(err);
    res.status(500).json({ error: err.message });
  }
});
```

Foreign Key Constraint Handling

- This slide shows how foreign key constraints are enforced in the SQLite database to maintain data integrity.
- The PRAGMA foreign_keys = ON command ensures that the relationships between tables (like students, courses, assignments) are strictly maintained.
- This setup prevents orphan records and maintains consistency in the database, which is crucial for relational databases.
- It exemplifies the application’s adherence to database best practices, ensuring that the data remains reliable and meaningful.



```
const sqlite3 = require('sqlite3').verbose();
const DB_PATH = './database.db';

let db = new sqlite3.Database(DB_PATH, (err) => {
  if (err) {
    console.error(err.message);
    throw err;
  } else {
    console.log('Connected to the SQLite database.');
    db.run("PRAGMA foreign_keys = ON", err => {
      if (err) {
        console.error("Error enabling foreign key constraints:", err.message);
      } else {
        console.log("Foreign key constraints enabled.");
      }
    });
  }
});

module.exports = db;
```

Challenges Faced

Database Connection and Data Retrieval

- **Challenge**: Initial difficulties in connecting SQLite database with Node.js/Express application and retrieving data.
- **Impact**: Hindered the ability to interact with the database and display data through API endpoints.

Application Crashes on Data Modification

- **Challenge**: Application crashed when modifying data, due to repeated data insertion attempts.
- **Impact**: Inconsistent application behavior and development delays.

Foreign Key Constraints Issues

- **Challenge**: Encountered problems with foreign key constraints, leading to errors during data insertion and updates.
- **Impact**: Affected database integrity and application functionality.

Error Handling Complications

- **Challenge**: Issues with implementing robust error handling mechanisms.
- **Impact**: Difficulty in diagnosing and resolving errors, leading to user experience issues.

Solutions Implemented

Resolving Database Connection

Solution: Implemented reliable methods for establishing database connections and retrieving data.

Outcome: Achieved a stable connection, enabling smooth data operations.

Preventing Application Crashes

Solution: Added checks to verify if tables are empty before inserting data from insert.sql.

Outcome: Prevented unnecessary data insertions, enhancing application stability.

Addressing Foreign Key Constraints

Solution: Adjusted the code and the order of operations to handle foreign key constraints effectively while keeping them active.

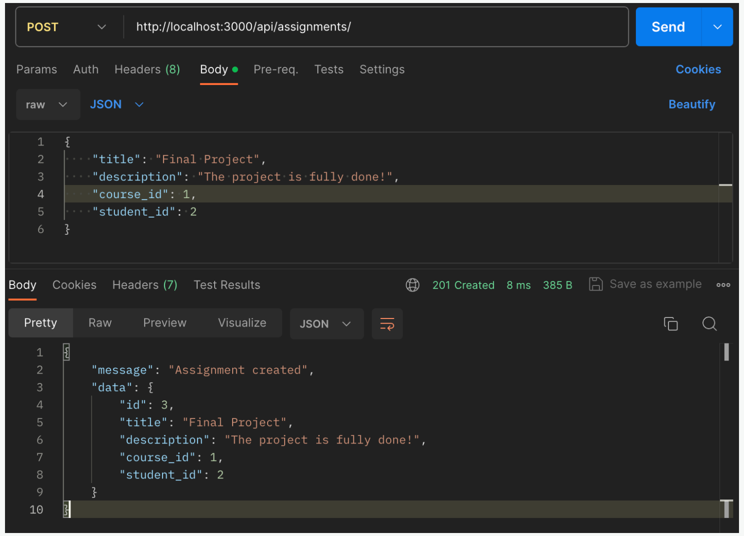
Outcome: Maintained database integrity without compromising on relational data constraints.

Improving Error Handling

Solution: Enhanced error handling to identify and respond to issues more effectively.

Outcome: Better error diagnosis and resolution, leading to a more robust application.

API Test Calls



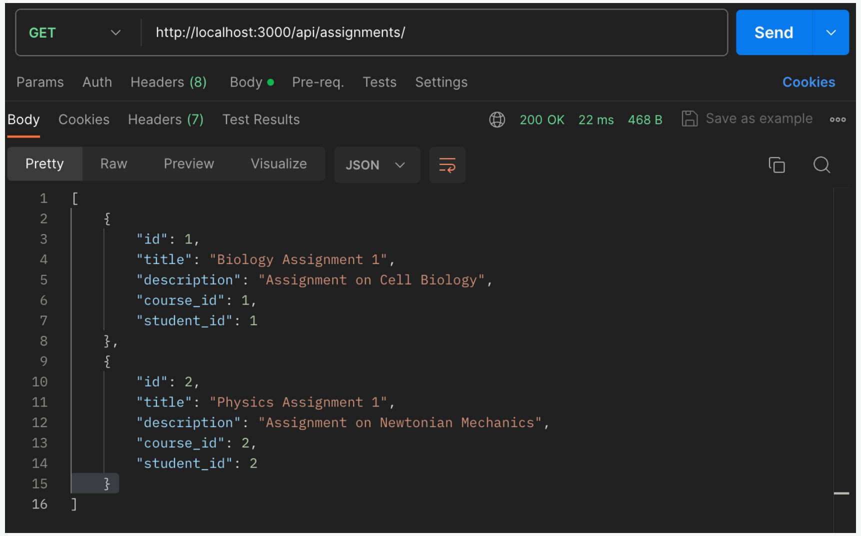
POST http://localhost:3000/api/assignments/

```
{
    "title": "Final Project",
    "description": "The project is fully done!",
    "course_id": 1,
    "student_id": 2
}
```

201 Created 8 ms 385 B

JSON

```
{
    "message": "Assignment created",
    "data": {
        "id": 3,
        "title": "Final Project",
        "description": "The project is fully done!",
        "course_id": 1,
        "student_id": 2
    }
}
```



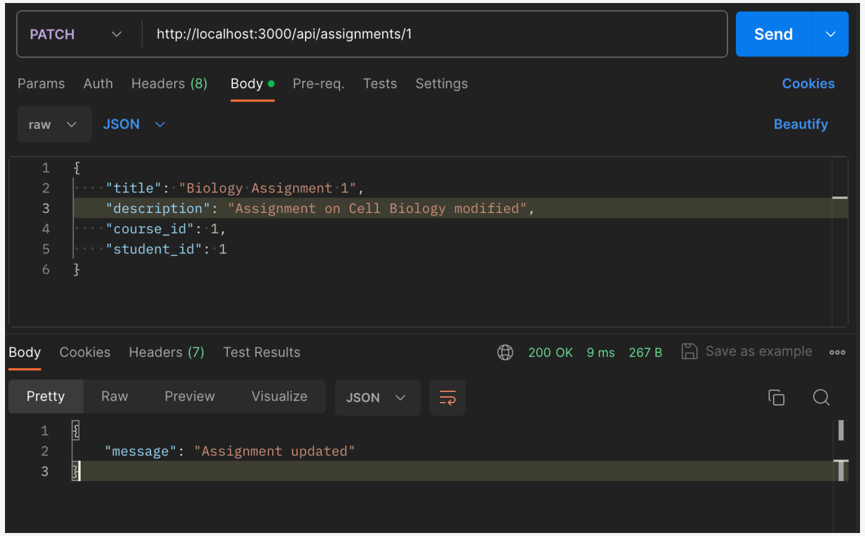
GET http://localhost:3000/api/assignments/

200 OK 22 ms 468 B

JSON

```
[
    {
        "id": 1,
        "title": "Biology Assignment 1",
        "description": "Assignment on Cell Biology",
        "course_id": 1,
        "student_id": 1
    },
    {
        "id": 2,
        "title": "Physics Assignment 1",
        "description": "Assignment on Newtonian Mechanics",
        "course_id": 2,
        "student_id": 2
    }
]
```

API Test Calls

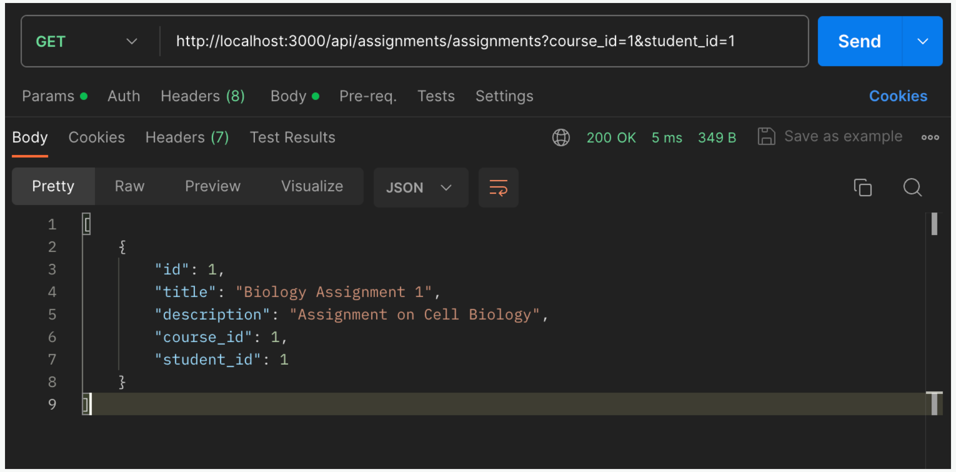


PATCH http://localhost:3000/api/assignments/1

```
{
    "title": "Biology Assignment 1",
    "description": "Assignment on Cell Biology modified",
    "course_id": 1,
    "student_id": 1
}
```

200 OK 9 ms 267 B

```
{
    "message": "Assignment updated"
}
```



GET http://localhost:3000/api/assignments/assignments?course_id=1&student_id=1

200 OK 5 ms 349 B

```
[
    {
        "id": 1,
        "title": "Biology Assignment 1",
        "description": "Assignment on Cell Biology",
        "course_id": 1,
        "student_id": 1
    }
]
```

Timetable

Overall Duration:

Total Time: Approximately 20 hours

Spent: 3 Days

Day 1: Initial Setup and Planning

Setting up the Node.js environment.

Creating basic project structure.

Planning the database schema and API endpoints.

Day 2: Core Development

Implementing the database schema in SQLite.

Developing CRUD operations for students, courses, assignments, enrollments, submissions, grades.

Testing API endpoints individually for functionality.

Day 3: Refinement and Testing

Refining code and optimizing API responses.

Conducting comprehensive testing.

Finalizing the project and preparing documentation.

Key Achievements:

Successfully integrated Node.js with SQLite for a RESTful API.

Developed a fully functional API catering to a range of operations.

Ensured robust error handling and validation across endpoints.

Summary

Project Overview:

- Developed a robust Node.js backend application with Express framework.
- Implemented a RESTful API serving various entities like students, courses, assignments, etc.
- Integrated SQLite database for data storage with foreign key constraints for data integrity.

Key Achievements:

- Successfully set up an automated database initialization process.
- Implemented dynamic query handling for complex searches.
- Established comprehensive error handling for reliable API responses.
- Ensured data validation for all incoming requests.

Technologies Used:

- Node.js and Express for server and API development.
- SQLite for database management.
- Express-validator for request validation.
- Async/await patterns for handling asynchronous operations.

Challenges and Solutions:

- Encountered and resolved issues related to database connections and foreign key constraints.
- Overcame challenges in dynamic query construction and error handling.
- Adapted code to handle different error scenarios gracefully.

Learning Outcomes:

- Gained proficiency in Node.js and Express for building scalable backend solutions.
- Deepened understanding of database operations and data integrity in SQLite.
- Enhanced skills in error handling and data validation in a RESTful API context.

Conclusion

This project has been an insightful experience. It presented numerous challenges, particularly in integrating a Node.js application with a SQLite database and developing a dynamic RESTful API. These challenges improved my problem-solving capabilities. The knowledge gained from this endeavor is valuable and will undoubtedly be beneficial in future projects. I am thankful for the learning opportunity this project and the course provided and look forward to applying these skills in new and exciting contexts.