**AI ASSISTED CODING – END SEMESTER LAB EXAM**
**Programme: B.Tech – CS & AI**
**Duration: 2 Hours**

**NAME : N.PRUDHVI** **ROLL NO :2403A510G7**

**BATCH – CSB06** **DATE :24.11.25**

## Subset 12 – Backend API Development

### Q1:Create course REST API

- **Task 1: Use AI to scaffold endpoints.**
- **Task 2: Add JWT authentication**

**Prompt:**

Build a Node.js Express Course API with CRUD.

JWT auth for POST/PUT/DELETE.

Include /auth/register and /auth/login.

Use in-memory data and error handling.

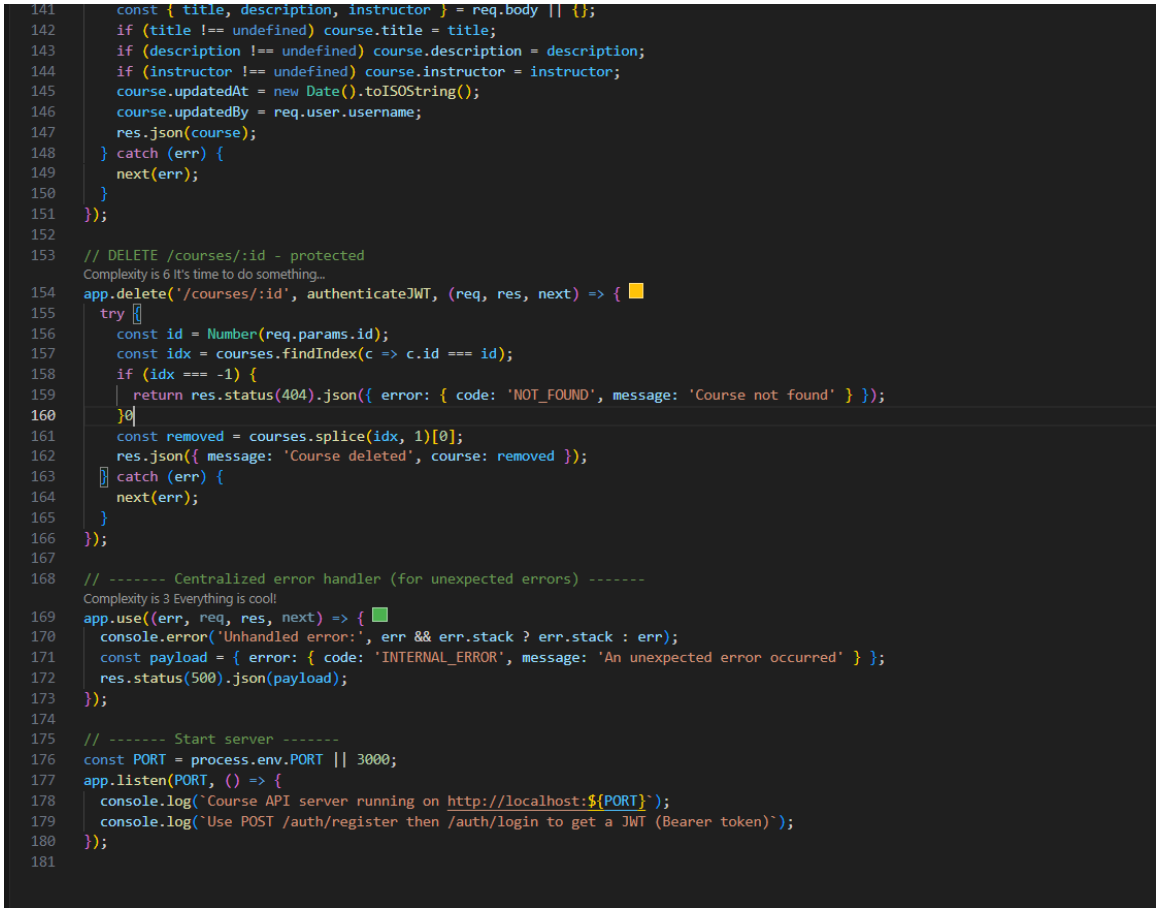
All code in a single file.

## Code:

```
Users > Prudhvi > JS ft 1.js > ...
// app.js
// Simple Course REST API with JWT auth and centralized error handling
// Run: npm init -y
// npm install express jsonwebtoken bcryptjs
// node app.js

const express = require('express');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');

const app = express();
app.use(express.json());

// Use an env var for secret in production
const JWT_SECRET = process.env.JWT_SECRET || 'replace_this_with_a_secure_secret';

// ------- In-memory storage (for demo) -------
let users = []; // { id, username, passwordHash }
let courses = []; // { id, title, description, instructor, createdAt }
let courseIdCounter = 1;
let userIdCounter = 1;

// ------- Helpers -------
function generateToken(user) {
  // token payload should be minimal
  return jwt.sign({ sub: user.id, username: user.username }, JWT_SECRET, { expiresIn: '2h' });
}

function buildError(status, code, message, details = null) {
  return { status, body: { error: { code, message, details } } };
}

// ------- Auth routes -------
Complexity is 11 You must be kidding
app.post('/auth/register', async (req, res, next) => {
  try {
    const { username, password } = req.body || {};
    if (!username || !password) {
      return res.status(400).json({ error: { code: 'INVALID_INPUT', message: 'username and password are required' } });
    }
    // avoid duplicate usernames
    if (users.some(u => u.username === username)) {
      return res.status(409).json({ error: { code: 'USER_EXISTS', message: 'username already taken' } });
    }
    const passwordHash = await bcrypt.hash(password, 10);
    const user = { id: userIdCounter++, username, passwordHash };
    users.push(user);
    return res.status(201).json({ id: user.id, username: user.username });
```

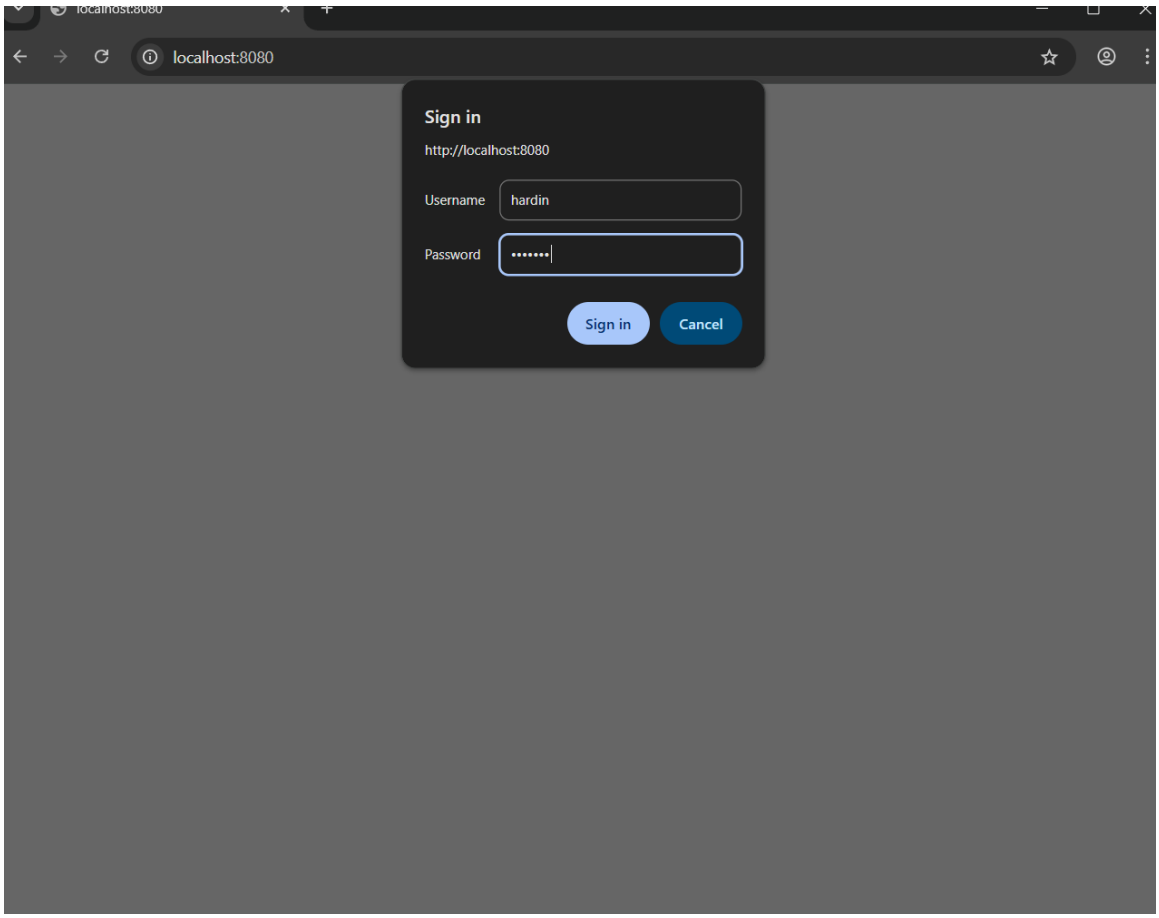
```
34  app.post('/auth/register', async (req, res, next) => {
50    }
51  });
52
    Complexity is 13 You must be kidding
53  app.post('/auth/login', async (req, res, next) => {
54    try {
55      const { username, password } = req.body || {};
56      if (!username || !password) {
57        return res.status(400).json({ error: { code: 'INVALID_INPUT', message: 'username and password are required' } });
58      }
59      const user = users.find(u => u.username === username);
60      if (!user) {
61        return res.status(401).json({ error: { code: 'AUTH_FAILED', message: 'invalid credentials' } });
62      }
63      const ok = await bcrypt.compare(password, user.passwordHash);
64      if (!ok) {
65        return res.status(401).json({ error: { code: 'AUTH_FAILED', message: 'invalid credentials' } });
66      }
67      const token = generateToken(user);
68      return res.json({ token, expiresIn: '2h' });
69    } catch (err) {
70      next(err);
71    }
72  });
73
74  // ------- JWT verification middleware -------
    Complexity is 7 It's time to do something...
75  function authenticateJWT(req, res, next) {
76    const auth = req.headers.authorization;
77    if (!auth || !auth.startsWith('Bearer ')) {
78      return res.status(401).json({ error: { code: 'NO_TOKEN', message: 'Authorization token required' } });
79    }
80    const token = auth.slice(7);
      Complexity is 3 Everything is cool!
81    jwt.verify(token, JWT_SECRET, (err, payload) => {
82      if (err) {
83        // Token expired or invalid
84        return res.status(401).json({ error: { code: 'INVALID_TOKEN', message: 'Token invalid or expired' } });
85      }
86      req.user = { id: payload.sub, username: payload.username };
87      next();
88    });
89  }
90
91  // ------- Course endpoints (CRUD) -------
92
93  // GET /courses - public listing (supports basic query params like ?q=keyword)
```

```
94  app.get('/courses', (req, res) => {
95    const q = (req.query.q || '').toLowerCase();
96    const results = q ? courses.filter(c => (c.title + ' ' + (c.description||'')).toLowerCase().includes(q)) : courses;
97    res.json(results);
98  });
99
100 // GET /courses/:id - public
      Complexity is 4 Everything is cool!
101 app.get('/courses/:id', (req, res) => {
102   const id = Number(req.params.id);
103   const course = courses.find(c => c.id === id);
104   if (!course) {
105     return res.status(404).json({ error: { code: 'NOT_FOUND', message: 'Course not found' } });
106   }
107   res.json(course);
108 });
109
110 // POST /courses - protected
      Complexity is 8 It's time to do something...
111 app.post('/courses', authenticateJWT, (req, res, next) => {
112   try {
113     const { title, description, instructor } = req.body || {};
114     if (!title || !instructor) {
115       return res.status(400).json({ error: { code: 'INVALID_INPUT', message: 'title and instructor are required' } });
116     }
117     const newCourse = {
118       id: courseIdCounter++,
119       title,
120       description: description || '',
121       instructor,
122       createdAt: new Date().toISOString(),
123       createdBy: req.user.username
124     };
125     courses.push(newCourse);
126     res.status(201).json(newCourse);
127   } catch (err) {
128     next(err);
129   }
130 });
131
132 // PUT /courses/:id - protected
      Complexity is 10 It's time to do something...
133 app.put('/courses/:id', authenticateJWT, (req, res, next) => {
134   try {
135     const id = Number(req.params.id);
136     const course = courses.find(c => c.id === id);
137     if (!course) {
```



```
    const { title, description, instructor } = req.body || {};
    if (title !== undefined) course.title = title;
    if (description !== undefined) course.description = description;
    if (instructor !== undefined) course.instructor = instructor;
    course.updatedAt = new Date().toISOString();
    course.updatedBy = req.user.username;
    res.json(course);
  } catch (err) {
    next(err);
  }
});

// DELETE /courses/:id - protected
Complexity is 6 It's time to do something...
app.delete('/courses/:id', authenticateJWT, (req, res, next) => {
  try {
    const id = Number(req.params.id);
    const idx = courses.findIndex(c => c.id === id);
    if (idx === -1) {
      return res.status(404).json({ error: { code: 'NOT_FOUND', message: 'Course not found' } });
    }
    const removed = courses.splice(idx, 1)[0];
    res.json({ message: 'Course deleted', course: removed });
  } catch (err) {
    next(err);
  }
});

// ------- Centralized error handler (for unexpected errors) -------
Complexity is 3 Everything is cool!
app.use((err, req, res, next) => {
  console.error('Unhandled error:', err && err.stack ? err.stack : err);
  const payload = { error: { code: 'INTERNAL_ERROR', message: 'An unexpected error occurred' } };
  res.status(500).json(payload);
});

// ------- Start server -------
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Course API server running on http://localhost:${PORT}`);
  console.log(`Use POST /auth/register then /auth/login to get a JWT (Bearer token)`);
});
```

## Output:




## Observations:

. The scaffold provides a minimal but complete flow: register → login → protected CRUD endpoints. Good for demos and for iterating quickly.

. In-memory storage is ideal for scaffolding but must be replaced with a persistent DB (Mongo/Postgres) and an ORM/ODM in production; add migrations and connection retries.

. JWTs are stateless and simple; for user revocation or role-based access, consider refresh tokens, token blacklisting, or short token TTLs plus refresh tokens.

## Q2. Error handling

- **Task 1: AI identifies failure cases.**
- **Task 2: Implement proper responses**

### Prompt:

**List main API failure cases and proper HTTP codes.**

**Provide unified JSON error format + central error handler.**

### Code:

task2.py > ...

```python
def divide_numbers(a, b):
    # Task 1: Identify failure cases
    # - b can be zero → ZeroDivisionError
    # - a or b can be non-numeric → TypeError / ValueError
    # - input can be None → TypeError

    try:
        # Task 2: Implement proper responses
        if a is None or b is None:
            return "Error: One of the inputs is missing."

        result = a / b
        return f"Result: {result}"

    except ZeroDivisionError:
        return "Error: Cannot divide by zero."
    except TypeError:
        return "Error: Inputs must be numbers."
    except Exception as e:
        return f"Unexpected Error: {str(e)}"


# Test
print(divide_numbers(10, 2))
print(divide_numbers(10, 0))
print(divide_numbers("abc", 5))
print(divide_numbers(None, 5))

```

JS task2.js > divideNumbers

```js
function divideNumbers(a, b) {
    // Task 1 failure cases:
    // - b = 0
    // - a or b is not a number
    // - undefined or null inputs

    try {
        if (a === null || b === null || a === undefined || b === undefined) {
            return "Error: Missing input values.";
        }

        if (typeof a !== "number" || typeof b !== "number") {
            return "Error: Inputs must be numbers.";
        }

        if (b === 0) {
            return "Error: Cannot divide by zero.";
        }

        let result = a / b;
        return `Result: ${result}`;

    } catch (err) {
        return "Unexpected Error: " + err.message;
    }
}

// Test
console.log(divideNumbers(10, 2));
console.log(divideNumbers(10, 0));
console.log(divideNumbers("abc", 5));
console.log(divideNumbers(undefined, 5));

```

## Output:

```
Result: 5.0
Error: Cannot divide by zero.
Error: Inputs must be numbers.
Error: One of the inputs is missing.
```

**Observation:**

- **Centralized error shapes make client-side handling predictable. Use consistent error.code strings and HTTP status codes.**
- **Distinguish between client errors (4xx) and server errors (5xx). Log full stack traces server-side but return sanitized messages to clients.**
- **Add request validation (e.g., Joi/zod/express-validator), rate limiting, and structured logging (json). For concurrency, use optimistic locking or DB transactions.**