Express.js & Node.js Coding Standards

General Principles

- ✓ Write clean, readable, and maintainable code
- ✓ Follow DRY (Don't Repeat Yourself) principle
- ✓ Keep functions and modules small and focused
- ✓ Use async/await over callbacks for better readability

Code Styling

Indentation

- Indent code with 2 spaces. No trailing spaces or mixed tabs.

```
const fetchStudentData = async (req, res) => {
  const { studentId } = req.params;

  if (!studentId) {
    return res.status(400).json({ error: 'Student ID required' });
  }

  try {
    const query = 'SELECT * FROM students WHERE id = ?';
    const [results] = await db.promise().query(query, [studentId]);
    return res.json(results);
  } catch (error) {
    console.error('Database error:', error);
    return res.status(500).json({ error: 'Internal server error' });
  }
};
```

Comments

- Add comments only for complex logic, not for obvious code.
- Use JSDoc for function documentation.

```
/**
 * Calculates student's total purchase amount
 * @param {number} studentId - The ID of the student
 * @returns {Promise<number>} Total amount spent
 */
const calculateTotalPurchases = async (studentId) => {
  const query = `
    SELECT SUM(price) as total
    FROM purchases
    WHERE student_id = ?
  `;
  const [results] = await db.promise().query(query, [studentId]);
  return results[0].total || 0;
};
```

Variable Naming

- Use const by default

```
const PORT = 3000;
const dbConfig = {
  host: 'localhost',
  user: 'root',
  database: 'project'
};
```

- Use let when reassignment is needed

```
let connectionAttempts = 0;
connectionAttempts++;
```

- Never use var

String Handling

- Use template literals for string interpolation

```
const successMessage = `User ${userName} registered successfully`;
console.log(`Server running on port ${PORT}`);
```

- Don't use string concatenation

```
// ❌ Avoid
const message = 'Hello, ' + userName + '!';
```

Naming Conventions

File Naming

- **Route files:** camelCase or kebab-case (e.g., studentRoutes.js, course-routes.js)
- **Controller files:** camelCase (e.g., studentController.js, examController.js)
- **Middleware:** camelCase (e.g., authMiddleware.js, errorHandler.js)
- **Utilities/Helpers:** camelCase (e.g., dbHelper.js, fileUtils.js)
- **Constants:** camelCase file, UPPER_SNAKE_CASE variables (e.g., constants.js containing MAX_FILE_SIZE)

Variable & Function Naming

- **Routes:** Descriptive verbs (e.g., getUserCourses, uploadContent)
- **Database queries:** Start with action verb (e.g., insertStudent, fetchNotices)
- **Middleware functions:** Descriptive names (e.g., validateLogin, checkAuth)

API Route Standards

Route Organization

- Group related routes together
- Use consistent naming patterns
- Always validate input data

```
// ✅ Good
app.post('/register', validateRegistration, registerStudent);
app.post('/login', validateLogin, loginStudent);
app.get('/courses', getCourses);
app.post('/buy-course', checkAuth, buyCourse);

// ❌ Avoid
app.post('/reg', (req, res) => { /* inline logic */ });
app.post('/userlogin', (req, res) => { /* inline logic */ });
```

Error Handling

- Always use try-catch blocks for async operations
- Return appropriate HTTP status codes
- Provide meaningful error messages

```
app.post('/buy-course', async (req, res) => {
  try {
    const { student_id, course_id } = req.body;

    if (!student_id || !course_id) {
      return res.status(400).json({
        error: 'Student ID and Course ID are required'
      });
    }

    // Business logic here

    res.status(200).json({
      success: true,
      message: 'Course purchased successfully'
    });
  } catch (error) {
    console.error('Purchase error:', error);
    res.status(500).json({
      error: 'Failed to purchase course'
    });
  }
```

Database Query Standards

Query Formatting

- Use parameterized queries to prevent SQL injection
- Format multi-line queries for readability

```
// ✅ Good - Parameterized query
const query = `
  SELECT c.*, sc.purchase_date
  FROM courses c
  INNER JOIN student_courses sc ON c.id = sc.course_id
  WHERE sc.student_id = ?
  ORDER BY sc.purchase_date DESC
`;
db.query(query, [studentId], (err, results) => {
  // Handle results
});

// ❌ Avoid - String concatenation (SQL injection risk)
const query = `SELECT * FROM students WHERE email='${email}'`;
```

Connection Management

- Use connection pooling for better performance
- Handle connection errors gracefully

```
const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'root',
  database: 'project'
});

db.connect((err) => {
  if (err) {
    console.error('Database connection failed:', err);
    process.exit(1);
  }
  console.log('✅ MySQL connected successfully');
});
```

Middleware Standards

File Upload Middleware

- Configure storage path and filename generation
- Validate file types and sizes

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/course_content/');
  },
  filename: (req, file, cb) => {
    const timestamp = Date.now();
    const filename = `${timestamp}-${file.originalname}`;
    cb(null, filename);
  }
});

const uploadContent = multer({
  storage: storage,
  limits: { fileSize: 50 * 1024 * 1024 }, // 50MB
  fileFilter: (req, file, cb) => {
    const allowedTypes = ['application/pdf', 'video/mp4'];
    if (allowedTypes.includes(file.mimetype)) {
      cb(null, true);
    } else {
      cb(new Error('Invalid file type'));
    }
  }
});
```

Socket.IO Standards

Event Naming

- Use clear, descriptive event names
- Follow consistent naming pattern

```
// ✅ Good
io.on('connection', (socket) => {
  console.log('User connected:', socket.id);

  socket.on('send-message', (data) => {
    io.emit('new-message', data);
  });

  socket.on('disconnect', () => {
    console.log('User disconnected:', socket.id);
  });
});

// ❌ Avoid
socket.on('msg', (data) => { /* ... */ });
```

Project Structure

project-root/

├── node_modules/

├── uploads/

│ └── chat/

│ └── course_content/

├── admin_courses.html

├── admin_dashboard.html

├── admin_mcq.html

├── admin_notices.html

├── admin_stationary.html

├── admin_upload_content.html

├── student_course_contents.html

├── student_courses.html

├── student_dashboard.html

├── student_exam.html

├── student_group_chat.html

├── student_login.html

├── student_notices.html

├── student_profile.html

├── student_register.html

├── student_stationary.html

├── student_stationary_purchase.html

├── view_courses.html

├── server.

├── package.json

├── package-lock.json

└── index.html

Environment Variables

.env File Usage

- Store sensitive configuration in .env
- Never commit .env to version control

```
// .env
PORT=3000
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=root
DB_NAME=project
JWT_SECRET=your_secret_key

// server.js
require('dotenv').config();

const PORT = process.env.PORT || 3000;
const dbConfig = {
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME
};
```

Best Practices Summary

DO:

- ✓ Use async/await for asynchronous operations
- ✓ Validate all user inputs
- ✓ Use parameterized SQL queries
- ✓ Handle errors appropriately
- ✓ Log important events and errors
- ✓ Use meaningful variable and function names
- ✓ Keep functions small and focused
- ✓ Use environment variables for configuration

DON'T:

- Use synchronous file operations in production
- Concatenate strings in SQL queries
- Ignore error handling
- Use inline callbacks (callback hell)
- Store sensitive data in code
- Write overly complex functions
- Mix concerns (routing + business logic)