COLLEGE CODE : 9528

COLLEGE NAME : SCAD COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT : COMPUTER SCIENCE AND ENGINEERING

STUDENTNMID : BE96D422034491CBF0B482D5A1BD21C0

Roll no : 952823104145

DATE : 26.09.2025

Completed the project named as:

Phase 3

TECHNOLOGYPROJECTNAME :

NODEJS BACKEND FOR CONTACT FORM

SUBMITTED By,

NAME: C.SELVA NANTHINI

MOBILE NO: 7604999577

IBM-FE - NodeJS Backend for Contact Form

Overview

This document provides a structured and practical guide to building a Node.js (Express) backend to support an IBM-FE contact form. It covers setup, implementation, data storage, testing, and version control. Clear explanations and code snippets make it accessible for students and developers.

Project setup

Step-by-Step Setup

```
1. Install prerequisites: Node.js (18+ LTS), npm/yarn, Git.
2. Initialize project:
mkdir ibm-fe-backend && cd ibm-fe-backend
npm init -y
3. Install dependencies:
npm i express cors helmet dotenv mongoose joi
npm i -D nodemon jest supertest eslint
4. Setup useful middleware: morgan, express-rate-limit, helmet.
5. Add package.json scripts:
"start": "node src/server.js"
"dev": "nodemon src/server.js"
"test": "jest --runInBand"
```

Setting up a Node.js project for the contact form begins with initializing a new project using npm init or yarn init, creating the foundation for dependency management. Essential dependencies such as Express.js (for building the REST API) and body-parser (for handling JSON or URL-encoded form submissions) should be installed right away. Organizing the project structure is also critical—typically with dedicated folders for routes, controllers, and configuration. A .gitignore file should be included to keep unnecessary files, like node_modules, out of version control.

Core Features Implementation

1. Input Validation & Sanitization - Use Joi or express-validator to check name, email, and phone formats.

2. Controllers & Services - controllers/contacts.js: CRUD operations. - services/contacts.js: database logic.

3. Error Handling - Central error middleware sends JSON {error: 'message'} with HTTP status codes.

The core features of the Node.js backend revolve around processing contact form submissions reliably. This includes validating input fields to ensure required data (like name, email, and message) is present and formatted correctly. For example, the email address should follow a proper format, and the message field should not be empty. Input validation prevents errors and reduces the likelihood of malicious data being submitted. With Express middleware, these checks can be integrated seamlessly before data is processed further.

Data Storage (Local State / Database)

Local State (in-memory)
- Good for demos and prototypes. Fast but not persistent.

Database (Production)
- MongoDB with Mongoose: Flexible schema, great for contact form.
- PostgreSQL/MySQL: Relational with strong consistency.
- SQLite: Lightweight, file-based option.

Example: Mongoose Contact Model

```
const mongoose = require('mongoose');
const ContactSchema = new mongoose.Schema({
name: { type: String, required: true, trim: true },
email: { type: String, required: true, lowercase: true, unique: true },
phone: { type: String, required: true }
}, { timestamps: true });
module.exports = mongoose.model('Contact', ContactSchema);
```

For long-term storage and scalability, integrating a database is essential. Options like MongoDB or PostgreSQL can store user submissions securely and allow administrators to retrieve and analyze them later. MongoDB, in particular, works well with Node.js due to its flexible schema design and JSON-like structure. Implementing database storage ensures that contact form submissions are not only received but also available for further actions like reporting, customer support, or marketing insights. The choice of database should depend on project requirements such as scalability, complexity, and integration with other services.

Testing Core Features

- Use Jest for unit testing and SuperTest for API integration tests. - Example: test POST /contacts for success and validation errors. - Use Postman for manual testing against local/staging environments.

```
const request = require('supertest'); const app =
require('../src/server');

test('POST /contacts creates', async () => { const res =
await request(app) .post('/contacts') .send({name:'Test',
email:'t@x.com', phone:'9876543210'});
expect(res.statusCode).toBe(201); });
```

Testing is a critical step to ensure the backend functions as expected. Unit tests can be written for individual components like input validation, ensuring that invalid emails or missing message fields trigger the correct error responses. Integration tests should verify that the complete flow—from sending a POST request to the backend to receiving a success response—works without issues. Tools like Jest or Mocha are commonly used in the Node.js ecosystem for such testing.

Version Control with GitHub

1. Initialize repo: git init
2. Add .gitignore (node_modules, .env)
3. Branching: main, develop, feature/*
4. Commit messages: feat(api): add POST route
5. GitHub Actions for CI: run npm test on push/pull request

Version control is essential for maintaining a clean development process. Using Git with GitHub ensures that all changes to the backend code are tracked and can be rolled back if needed.Branching strategies such as feature branches or pull requests should be adopted to keep the main or production branch stable. By pushing the project to GitHub early, the team gains additional benefits like access to GitHub Actions for CI/CD pipelines, automated testing, and deployment workflows. Maintaining the project in a public or private GitHub repository also improves transparency and accountability while ensuring that no work is lost if a local environment fails.