Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

DEPLOYMENT, TESTING & CAPSTONE PROJECT (WEEKS 11-12)

DEPLOYING NODE.JS APPLICATIONS ON AWS, VERCEL, AND HEROKU

CHAPTER 1: INTRODUCTION TO NODE.JS DEPLOYMENT

1.1 Why Deploy Node.js Applications?

Deploying a **Node.js application** means making it accessible over the internet so that users can interact with it. A local development environment is useful for testing, but to make an application available globally, it must be hosted on a server.

Common Deployment Platforms for Node.js:

✓ **AWS (Amazon Web Services)** – Best for large-scale, flexible, and enterprise applications.

✓ **Vercel** – Ideal for front-end and full-stack applications with serverless functions.

✓ **Heroku** – A beginner-friendly PaaS (Platform as a Service) for quick deployments.

Each of these platforms has different advantages depending on the project **complexity, scalability, and cost considerations**.

CHAPTER 2: DEPLOYING A NODE.JS APPLICATION ON AWS

2.1 Setting Up an AWS EC2 Instance

To deploy a **Node.js application on AWS,** we use **EC2 (Elastic Compute Cloud),** a virtual server that hosts applications.

Step 1: Create an AWS EC2 Instance

1. **Log in to AWS** and go to the **EC2 dashboard**.
2. Click on **Launch Instance**.
3. Choose an **Amazon Machine Image (AMI)**: Select **Ubuntu 22.04 LTS**.
4. Choose an instance type (**t2.micro** for free-tier users).
5. Configure security group:
    - Allow **port 22** (SSH) for remote access.
    - Allow **port 80** (HTTP) and **port 3000** (for the Node.js app).
6. **Launch the instance** and download the private key (.pem file).

2.2 Connecting to the AWS Server

Use SSH to connect to your EC2 instance from your local machine:

ssh -i your-key.pem ubuntu@your-ec2-instance-ip

Update and install Node.js on the EC2 instance:

sudo apt update && sudo apt upgrade -y

curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -

sudo apt install -y nodejs

Verify the installation:

node -v

npm -v

2.3 Deploying the Application on AWS

1. **Transfer your Node.js project to AWS using SCP (from your local machine):**
2. scp -i your-key.pem -r your-node-app ubuntu@your-ec2-instance-ip:/home/ubuntu/
3. **Install dependencies inside the EC2 instance:**
4. cd your-node-app
5. npm install
6. **Run the application:**
7. node server.js
8. **Keep the server running using PM2 (process manager):**
9. npm install -g pm2
10. pm2 start server.js
11. **Access the application in a browser:**
12. http://your-ec2-instance-ip:3000

CHAPTER 3: DEPLOYING A NODE.JS APPLICATION ON VERCEL

3.1 Why Use Vercel?

Vercel is optimized for **front-end applications and serverless Node.js functions**. It is best suited for deploying:

✓ **Static sites** and **full-stack applications**.

✓ **API backends** using **serverless functions**.

✓ **Instant deployment** with GitHub integration.

3.2 Deploying a Node.js API on Vercel

Step 1: Install the Vercel CLI

npm install -g vercel

Step 2: Initialize the Project for Vercel

In your Node.js project directory, run:

vercel login

vercel init

Follow the prompts to configure the deployment settings.

Step 3: Deploy the Application

vercel deploy

After deployment, you will receive a URL like:

https://your-node-app.vercel.app

✓ Vercel automatically sets up a serverless environment for your app.

CHAPTER 4: DEPLOYING A NODE.JS APPLICATION ON HEROKU

4.1 Why Use Heroku?

Heroku is a cloud **PaaS (Platform as a Service)** that simplifies application deployment. It is best suited for:

✓ **Rapid development and deployment.**

✓ **Small-to-medium applications.**

✓ **Beginner-friendly Node.js hosting.**

4.2 Deploying a Node.js App on Heroku

Step 1: Install the Heroku CLI

Download and install the Heroku CLI from Heroku CLI Download.

Step 2: Log in to Heroku

heroku login

Step 3: Create a Heroku Application

heroku create your-app-name

Step 4: Prepare Your Application

Ensure your package.json contains a **start script**:

```
"scripts": {
  "start": "node server.js"
}
```

Commit changes if using Git:

```
git init
git add .
git commit -m "Initial commit"
```

Step 5: Deploy the App to Heroku

git push heroku main

Once deployed, access your application at:

https://your-app-name.herokuapp.com

✓ **Heroku manages scaling and server resources automatically.**

Case Study: How Slack Uses AWS for Scalable Chat Services

Background

Slack, a leading communication platform, required a **highly scalable** infrastructure to support millions of concurrent users.

Challenges

- **Handling millions of chat messages in real-time.**
- **Ensuring high availability and fault tolerance.**
- **Scaling efficiently to meet user demands.**

Solution: Deploying on AWS

✓ Used **AWS EC2 and Load Balancers** to distribute chat server load.
✓ Implemented **Node.js WebSocket servers** for real-time messaging.
✓ Used **AWS Auto Scaling** to manage peak traffic hours.

By leveraging AWS, Slack ensures **high-speed, reliable communication services worldwide**.

Exercise

1. Deploy a simple **"Hello World"** Node.js API on **Vercel**.
2. Deploy a **Node.js + Express application** on **Heroku**.

3. Set up a **Node.js server on AWS EC2** and access it via a public URL.

Conclusion

In this section, we explored:

✓ **Deploying a Node.js application on AWS using EC2 instances.**

✓ **Deploying an API or front-end application using Vercel.**

✓ **Using Heroku for easy, beginner-friendly deployment.**

✓ **How Slack leverages AWS for scalable, real-time chat services.**

USING DOCKER & KUBERNETES FOR CONTAINERIZATION

CHAPTER 1: INTRODUCTION TO CONTAINERIZATION

1.1 Understanding Containerization

Containerization is a lightweight virtualization technique that packages an application and its dependencies together, ensuring that it runs **consistently across different environments**. Instead of relying on a complete virtual machine (VM), **containers share the host operating system**, making them more **efficient and portable**.

✓ **Ensures consistency** – Runs the same way on development, testing, and production.
✓ **Lightweight and fast** – Uses fewer resources compared to VMs.
✓ **Scalable** – Easily deploy and manage multiple instances.
✓ **Isolated environments** – Prevents dependency conflicts between applications.

Two key technologies enable containerization:

- **Docker** – A tool for creating, running, and managing containers.
- **Kubernetes** – A system for **orchestrating** (automating) the deployment, scaling, and management of containerized applications.

CHAPTER 2: GETTING STARTED WITH DOCKER

2.1 What is Docker?

Docker is an open-source containerization platform that allows developers to:

✓ Package applications and dependencies into a **Docker container**.
✓ Ensure applications run **identically on any system**.
✓ Improve **portability** across different environments (e.g., local, cloud, servers).

To install Docker, follow the instructions for your OS:

- **Windows/macOS**: Download from Docker Desktop.
- **Linux**: Install via terminal:

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

Verify Docker installation using:

docker --version

2.2 Creating a Docker Container

To containerize an application, you need a **Dockerfile**, which defines how the container should be built.

Example: Creating a Dockerfile for a Node.js App

1. Create a Dockerfile in your project directory:

```
# Use official Node.js image
FROM node:18

# Set working directory in container
WORKDIR /app
```

```
# Copy package.json and install dependencies
COPY package.json .
RUN npm install

# Copy the rest of the application files
COPY . .

# Expose port for communication
EXPOSE 3000

# Start the application
CMD ["node", "server.js"]
```

2. **Build the Docker image:**

```
docker build -t my-node-app .
```

3. **Run the container:**

```
docker run -p 3000:3000 my-node-app
```

✓ **Docker builds the image** based on the Dockerfile.
✓ **The container runs**, making the app accessible at http://localhost:3000.

CHAPTER 3: MANAGING DOCKER CONTAINERS

3.1 Running, Stopping, and Removing Containers

List all running containers:

docker ps

Stop a running container:

docker stop <container_id>

Remove a container:

docker rm <container_id>

Remove an image:

docker rmi my-node-app

✓ Proper management ensures that **unused containers and images don't consume unnecessary space**.

3.2 Using Docker Compose for Multi-Container Apps

Docker Compose simplifies the management of multiple containers. Instead of running individual docker run commands, you define services in a docker-compose.yml file.

Example: Running Node.js and MongoDB with Docker Compose

```
version: '3'
services:
 app:
  build: .
  ports:
   - "3000:3000"
  depends_on:
   - db
```

```
db:
  image: mongo
  ports:
   - "27017:27017"
```

To start all services:

```
docker-compose up -d
```

✓ This starts **both the Node.js app and MongoDB database** together.

CHAPTER 4: INTRODUCTION TO KUBERNETES

4.1 What is Kubernetes?

Kubernetes (K8s) is an **orchestration platform** for managing containerized applications in **production**. It automates:

✓ **Scaling** – Adjusts the number of running containers based on traffic.
✓ **Load balancing** – Distributes traffic efficiently across containers.
✓ **Self-healing** – Automatically restarts failed containers.
✓ **Rolling updates** – Deploys new versions without downtime.

Kubernetes groups containers into **Pods,** which are the smallest deployable units.

4.2 Setting Up a Kubernetes Cluster

To install Kubernetes locally, use **Minikube**:

```
minikube start
```

```
kubectl version
```

✓ Minikube runs a **single-node Kubernetes cluster** for development.

CHAPTER 5: DEPLOYING APPLICATIONS WITH KUBERNETES

5.1 Writing a Kubernetes Deployment File

A **Deployment** in Kubernetes ensures that the required number of container instances (Pods) are always running.

Example: Deploying a Node.js App in Kubernetes

Create a deployment.yaml file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: node-app
spec:
 replicas: 2
 selector:
  matchLabels:
   app: node-app
 template:
  metadata:
   labels:
    app: node-app
  spec:
```

```
containers:
- name: node-app
  image: my-node-app
  ports:
  - containerPort: 3000
```

✓ **Creates two replicas** of the application for **load balancing**.

Apply the deployment:

kubectl apply -f deployment.yaml

Case Study: How a FinTech Company Used Docker & Kubernetes for Scalability

Background

A **FinTech startup** needed to deploy a **secure, scalable** online banking application. Initial challenges included:

✓ **Inconsistent development environments**, causing bugs in production.
✓ **Manual scaling issues**, leading to slow response times during high traffic.
✓ **Downtime during deployments**, affecting customer experience.

Solution: Implementing Docker & Kubernetes

The team **containerized** their application using **Docker** and deployed it with **Kubernetes**, ensuring:

✓ **Consistent environments** across development, testing, and production.
✓ **Automatic scaling** based on traffic loads.

✓ **Rolling updates,** allowing new features to be deployed **without downtime**.

Results

- **Reduced deployment time from 2 hours to 10 minutes**.
- **99.99% uptime,** improving customer satisfaction.
- **50% reduction in server costs** by optimizing resource usage.

This case study highlights **why Docker & Kubernetes are critical for modern cloud-native applications**.

Exercise

1. Write a Dockerfile to containerize a basic **Node.js application**.
2. Use **Docker Compose** to set up a **Node.js API with MongoDB**.
3. Create a Kubernetes **deployment** that runs **3 replicas** of your Node.js container.
4. Implement a **rolling update strategy** in Kubernetes for updating the app **without downtime**.

Conclusion

In this section, we explored:

✓ **Docker for containerizing applications** and managing them efficiently.

✓ **Kubernetes for automating deployment, scaling, and load balancing**.

✓ **How to deploy and scale Node.js applications using Docker & Kubernetes**.

MANAGING ENVIRONMENT VARIABLES IN NODE.JS

CHAPTER 1: INTRODUCTION TO ENVIRONMENT VARIABLES

1.1 Understanding Environment Variables

Environment variables are key-value pairs used to **store configuration settings** separately from the application code. They help manage:

✓ **Database credentials** – Protects sensitive information like database URLs.
✓ **API keys** – Prevents unauthorized access to external services.
✓ **Port numbers** – Makes it easy to switch between development and production environments.
✓ **Third-party service configurations** – Manages settings for services like email providers, payment gateways, etc.

Using environment variables helps **enhance security, maintainability, and flexibility** by preventing hardcoding of sensitive values into source code.

CHAPTER 2: SETTING UP ENVIRONMENT VARIABLES IN NODE.JS

2.1 Using Built-in Environment Variables

Node.js allows accessing environment variables through process.env.

Example: Accessing System Variables

```
console.log('Node Environment:', process.env.NODE_ENV);
```

console.log('System Username:', process.env.USER);

✓ process.env.NODE_ENV is used to set **development or production modes**.
✓ process.env.USER retrieves the **current system username**.

2.2 Creating Custom Environment Variables

Instead of hardcoding configuration values, store them in a **.env file**.

Example: Creating a .env File

1. Create a **.env** file in the project root:
2. PORT=5000
3. DB_HOST=localhost
4. DB_USER=root
5. DB_PASS=securepassword
6. JWT_SECRET=mysecretkey
7. Install dotenv to load variables from .env:
8. npm install dotenv
9. Use dotenv in the Node.js app:
10. require('dotenv').config();
11.
12. console.log('App running on port:', process.env.PORT);

13.console.log('Database Host:', process.env.DB_HOST);

✓ This loads the **.env file** and makes variables accessible via process.env.

CHAPTER 3: MANAGING ENVIRONMENT VARIABLES IN DIFFERENT ENVIRONMENTS

3.1 Setting Variables for Development and Production

Different environments require different configurations. Common environments include:

✓ **Development** – Uses local database and debugging tools.
✓ **Testing** – Runs automated tests without affecting real data.
✓ **Production** – Uses secure API keys and database credentials.

To switch environments dynamically, use **NODE_ENV**:

Example: Configuring Different Databases Based on NODE_ENV

```
require('dotenv').config();


const config = {
  development: {
    db: process.env.DEV_DB,
    port: process.env.PORT || 3000
  },
  production: {
    db: process.env.PROD_DB,
    port: process.env.PORT || 8000
  }
};
```

```
const currentConfig = config[process.env.NODE_ENV] ||
config.development;


console.log('Current Database:', currentConfig.db);

console.log('Running on Port:', currentConfig.port);
```

✓ **Different databases** are used for development and production.
✓ process.env.NODE_ENV determines the **active configuration**.

Set NODE_ENV in terminal:

```
export NODE_ENV=production

node app.js
```

CHAPTER 4: SECURING ENVIRONMENT VARIABLES

4.1 Preventing Accidental Exposure

✓ **Never commit .env files** to version control (GitHub).
✓ Use a **.gitignore** file to exclude .env:

```
# Ignore environment variables

.env
```

✓ **Use environment variables in deployment** instead of .env files for security.

4.2 Storing Environment Variables in Cloud Services

- **For AWS Lambda**: Set environment variables in AWS Console.
- **For Heroku**: Use heroku config:set KEY=value.
- **For Docker**: Use an env_file.

Example: Setting Environment Variables in Heroku

heroku config:set JWT_SECRET=mysecurekey

Case Study: How a Fintech Startup Used Environment Variables for Secure API Management

Background

A fintech startup needed to **securely store API keys** for banking transactions.

Challenges

- Hardcoded API keys led to **security vulnerabilities**.
- Switching between **sandbox and production keys** was difficult.
- Exposing keys in logs led to **data leaks**.

Solution: Implementing Environment Variables

✓ Used .env files for **local development**.
✓ Configured **AWS Secrets Manager** for production API keys.
✓ Restricted **environment variable access** to authorized services.

Results

- **90% reduction in security incidents** related to exposed keys.
- **Faster deployment,** allowing seamless switching between environments.
- **Better compliance** with financial data security regulations.

This case study shows how **environment variables prevent security breaches** while improving **deployment efficiency**.

Exercise

1. Create a **.env file** and store database credentials.
2. Write a script that loads **environment variables using dotenv**.
3. Modify the script to use different variables for **development and production**.

Conclusion

✓ **Environment variables** enhance **security and maintainability** in applications.
✓ **The .env file** stores **sensitive configurations** securely.
✓ **Switching environments dynamically** allows better **scalability and flexibility**.

WRITING UNIT AND INTEGRATION TESTS WITH JEST

CHAPTER 1: INTRODUCTION TO TESTING IN NODE.JS

1.1 Why Testing is Important?

Testing is a critical part of software development that ensures applications function correctly and reliably. Without testing, small changes in the codebase can introduce unexpected bugs that could break the system.

Types of Testing in Node.js

- **Unit Testing** – Tests individual functions or components in isolation.
- **Integration Testing** – Tests how different parts of the application work together.
- **End-to-End (E2E) Testing** – Simulates real-world user interactions.

In this chapter, we will focus on **unit testing** and **integration testing** using **Jest**, a popular JavaScript testing framework.

CHAPTER 2: SETTING UP JEST FOR TESTING IN NODE.JS

2.1 Installing Jest in a Node.js Project

1. Create a new project and initialize package.json:
2. mkdir jest-testing
3. cd jest-testing

4. npm init -y
5. Install Jest:
6. npm install --save-dev jest
7. Update package.json to include a test script:
8. "scripts": {
9. "test": "jest"
10. }
11. Verify Jest installation by running:
12. npm test

If Jest is installed correctly, it should run without errors.

CHAPTER 3: WRITING UNIT TESTS WITH JEST

3.1 What is Unit Testing?

Unit testing involves testing individual functions or modules in isolation to ensure they work as expected.

3.2 Writing a Simple Function to Test

Create a new file math.js with the following code:

```
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
```

```
}

module.exports = { add, subtract };
```

3.3 Writing Unit Tests with Jest

Create a test file math.test.js in the same directory:

```
const { add, subtract } = require('./math');

test('adds 3 + 5 to equal 8', () => {
  expect(add(3, 5)).toBe(8);
});

test('subtracts 10 - 6 to equal 4', () => {
  expect(subtract(10, 6)).toBe(4);
});
```

3.4 Running Unit Tests

Run Jest from the terminal:

```
npm test
```

Expected output:

```
PASS  ./math.test.js
✓ adds 3 + 5 to equal 8 (3ms)
✓ subtracts 10 - 6 to equal 4 (2ms)
```

✓ Jest **automatically finds files ending in .test.js**.

✓ expect() is used to check if the function returns the expected result.

CHAPTER 4: WRITING INTEGRATION TESTS WITH JEST

4.1 What is Integration Testing?

Integration testing ensures multiple modules or components work together correctly. This is useful for testing:

- **Database interactions**
- **API endpoints**
- **Service dependencies**

4.2 Setting Up an Express.js Server for Testing

Create a new file server.js:

```
const express = require('express');

const app = express();

app.use(express.json());


app.get('/hello', (req, res) => {

  res.json({ message: "Hello, Jest!" });

});


app.post('/add', (req, res) => {

  const { num1, num2 } = req.body;
```

```
  res.json({ result: num1 + num2 });
});

module.exports = app;
```

4.3 Creating a Test File for API Integration

Install supertest, a package that allows testing HTTP requests:

```
npm install --save-dev supertest
```

Create a test file server.test.js:

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

test('GET /hello should return a welcome message', async () => {
  const response = await request(app).get('/hello');
  expect(response.statusCode).toBe(200);
  expect(response.body.message).toBe("Hello, Jest!");
});

test('POST /add should return sum of two numbers', async () => {
  const response = await request(app)
    .post('/add')
    .send({ num1: 5, num2: 7 });
```

```
  expect(response.statusCode).toBe(200);
  expect(response.body.result).toBe(12);
});
```

4.4 Running Integration Tests

Run the test suite:

npm test

Expected output:

PASS ./server.test.js

✓ GET /hello should return a welcome message (8ms)

✓ POST /add should return sum of two numbers (5ms)

✓ The API **responds correctly** with expected data.
✓ Jest **tests multiple components together,** ensuring proper integration.

Case Study: How Airbnb Uses Jest for Testing

Background

Airbnb, a global platform for booking stays, manages thousands of microservices. Each service must be **tested rigorously** to ensure reliability.

Challenges

- Maintaining **test coverage** across multiple services.
- Ensuring **real-time availability** of bookings.
- Reducing **bugs in production**.

Solution: Implementing Jest for Unit and Integration Testing

✓ **Unit testing for core functions** – Ensures price calculations, user authentication, and property listing logic work correctly.
✓ **Integration testing for APIs** – Ensures services like payments and booking confirmations interact correctly.
✓ **Automated CI/CD with Jest** – Every new code update is tested before deployment.

Results

- **40% reduction in production bugs**.
- Faster **feature rollouts with automated tests**.
- Improved **booking reliability,** reducing customer complaints.

This case study highlights how **Jest helps maintain quality assurance in large-scale applications**.

Exercise

1. What is the difference between unit testing and integration testing?
2. Write a Jest test for a function that multiplies two numbers.
3. Modify the server.test.js file to test a DELETE API endpoint.

Conclusion

In this section, we explored:
✓ **How to set up Jest for testing in Node.js**.
✓ **Writing unit tests for individual functions**.
✓ **Performing integration tests for API endpoints using Supertest**.

SETTING UP CI/CD PIPELINES FOR NODE.JS APPLICATIONS

CHAPTER 1: INTRODUCTION TO CI/CD PIPELINES

1.1 What is CI/CD?

CI/CD (**Continuous Integration and Continuous Deployment/Delivery**) is a **DevOps practice** that automates the process of testing, building, and deploying applications. It ensures that code changes are tested and delivered **quickly, reliably, and consistently**.

1.2 Why Use CI/CD in Node.js Applications?

✓ **Automates testing and deployment** – Reduces human error and speeds up the process.
✓ **Ensures code quality** – Every code update is automatically tested.
✓ **Enables faster releases** – Reduces time-to-market for new features.
✓ **Improves team collaboration** – Developers can merge and test code more efficiently.

CHAPTER 2: SETTING UP A CI/CD PIPELINE

2.1 CI/CD Workflow Overview

A typical **CI/CD pipeline** consists of:

1. **Continuous Integration (CI)** – Automates testing of new code before merging.
2. **Continuous Deployment (CD)** – Automatically deploys code to production after passing tests.

3. **Version Control (GitHub/GitLab)** – Manages source code changes.
4. **CI/CD Tool (GitHub Actions, Jenkins, GitLab CI/CD)** – Runs automated workflows.
5. **Deployment (AWS, Heroku, Vercel, or Docker)** – Pushes updates to production.

2.2 Setting Up a CI/CD Pipeline Using GitHub Actions

GitHub Actions is a **built-in CI/CD tool** for GitHub repositories. It automates testing and deployment whenever code is pushed to a repository.

Step 1: Create a Node.js Application

Ensure your project has the following files:

- **server.js** – Your main Node.js application.
- **package.json** – Contains project dependencies and scripts.
- **.github/workflows/ci-cd.yml** – Defines the CI/CD pipeline.

Step 2: Create a GitHub Actions Workflow

Inside your project, create a new folder:

mkdir -p .github/workflows

Create a new workflow file:

touch .github/workflows/ci-cd.yml

Open ci-cd.yml and add the following:

name: Node.js CI/CD Pipeline

```
on:
  push:
    branches:
      - main  # Runs the workflow when code is pushed to main branch
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 16

      - name: Install dependencies
```

```
      run: npm install

    - name: Run tests
      run: npm test

    - name: Build project
      run: npm run build

  deploy:
    needs: build
    runs-on: ubuntu-latest

    steps:
    - name: Deploy to Heroku
      uses: akhileshns/heroku-deploy@v3.12.12
      with:
        heroku_api_key: ${{ secrets.HEROKU_API_KEY }}
        heroku_app_name: "your-heroku-app-name"
        heroku_email: "your-email@example.com"
```

2.3 Explanation of GitHub Actions Workflow

✓ **Triggers on code push or pull requests** to the main branch.
✓ **Runs on Ubuntu** (ubuntu-latest) for consistency.

✓ **Installs dependencies** using npm install.
✓ **Runs automated tests** before deployment (npm test).
✓ **Builds the project** using npm run build.
✓ **Deploys to Heroku** if tests pass.

CHAPTER 3: SETTING UP CI/CD WITH JENKINS

3.1 Installing Jenkins

Jenkins is a popular **self-hosted CI/CD tool**. To install it on an Ubuntu server:

sudo apt update

sudo apt install openjdk-11-jre

wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt update

sudo apt install jenkins

Start Jenkins and get the admin password:

sudo systemctl start jenkins

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Visit **http://your-server-ip:8080** to complete Jenkins setup.

3.2 Configuring a Jenkins CI/CD Pipeline

1. **Create a new pipeline job in Jenkins**.

2. **Connect Jenkins to your GitHub repository**.
3. **Define a Jenkinsfile for CI/CD workflow**.

Example Jenkinsfile

```
pipeline {
  agent any

  stages {
    stage('Checkout Code') {
      steps {
        git branch: 'main', url: 'https://github.com/your-repo.git'
      }
    }
    stage('Install Dependencies') {
      steps {
        sh 'npm install'
      }
    }
    stage('Run Tests') {
      steps {
        sh 'npm test'
      }
    }
```

```
    stage('Deploy') {
      steps {
        sh 'git push heroku main'
      }
    }
  }
}
```

✓ **Automates the build, test, and deployment steps** in Jenkins.

CHAPTER 4: DEPLOYING THE CI/CD PIPELINE TO PRODUCTION

4.1 Connecting CI/CD to AWS Deployment

If deploying to AWS, add deployment commands inside GitHub Actions or Jenkins:

```
scp -i your-key.pem -r . ubuntu@your-ec2-instance:/var/www/node-app
```

```
ssh -i your-key.pem ubuntu@your-ec2-instance "cd /var/www/node-app && npm install && pm2 restart all"
```

✓ **Ensures new code is deployed automatically** to an AWS server.

4.2 Monitoring and Debugging CI/CD Pipelines

Common CI/CD Errors & Fixes:

Issue	Solution
Build fails	Check dependencies in package.json

Tests fail	Review test logs in CI/CD logs
Deployment fails	Ensure correct **API keys and credentials**
Slow deployment	Use **caching in CI/CD** workflows

Case Study: How Netflix Uses CI/CD for Continuous Deployment

Background

Netflix, a global streaming service, releases **thousands of software updates daily**.

Challenges

- **Ensuring smooth feature updates without downtime**.
- **Automatically testing millions of lines of code**.
- **Deploying updates quickly to global servers**.

Solution: Implementing Automated CI/CD Pipelines

✓ Used **GitHub Actions for CI** to run automated tests before merging code.
✓ Implemented **Jenkins Pipelines** for **automated deployment**.
✓ Deployed to **AWS with auto-scaling**, ensuring zero-downtime updates.

Netflix's CI/CD system allows **fast, stable, and automated deployments** for a seamless user experience.

Exercise

1. Modify the **GitHub Actions workflow** to deploy a Node.js app to **Vercel** instead of Heroku.

2. Set up a **basic Jenkins pipeline** for your own Node.js project.
3. Configure a **Slack notification** in CI/CD to alert developers when a build fails.

Conclusion

In this section, we explored:

✓ What CI/CD is and why it's essential for Node.js applications.
✓ Setting up a CI/CD pipeline using GitHub Actions.
✓ Using Jenkins for automated testing and deployment.
✓ Deploying applications automatically to AWS, Heroku, or Vercel.
✓ How Netflix uses CI/CD for global deployments.

API Monitoring and Logging in Node.js

Chapter 1: Introduction to API Monitoring and Logging

1.1 Understanding API Monitoring and Logging

APIs play a crucial role in modern applications by facilitating communication between different systems. Ensuring their **reliability, performance, and security** requires **API monitoring and logging**.

✓ **API Monitoring** – Tracks API performance, uptime, and errors in real-time.
✓ **API Logging** – Records API requests, responses, and errors for debugging and analytics.

Effective monitoring and logging help:

- Detect and resolve **performance bottlenecks**.
- Identify **security threats** and unauthorized access.
- Analyze API usage patterns for **optimization**.

Chapter 2: Setting Up API Logging in Node.js

2.1 Why is API Logging Important?

Logging API requests and responses helps:

✓ Debug **issues in API responses**.
✓ Track **malicious activities or unauthorized access**.
✓ Generate **audit logs** for compliance and security.

2.2 Implementing Logging with Winston

Winston is a popular logging library in Node.js that supports:

- **Multiple logging levels** (info, warning, error).
- **File-based and console logging**.
- **Integration with external logging services** (e.g., AWS, Elasticsearch).

Example: Setting Up Winston for Logging

```
const winston = require('winston');


const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'api.log' })
  ]
});


logger.info("API Logging Initialized");
logger.error("Something went wrong!");
```

✓ Logs are stored in **api.log**, and errors are printed to the **console**.

CHAPTER 3: LOGGING API REQUESTS AND RESPONSES

3.1 Using Middleware to Log API Requests

To log incoming API requests in an **Express.js API**, create middleware:

```
const express = require('express');

const winston = require('winston');

const app = express();


// Logger configuration

const logger = winston.createLogger({

  level: 'info',

  format: winston.format.json(),

  transports: [

    new winston.transports.File({ filename: 'requests.log' })

  ]

});


// Middleware to log API requests

app.use((req, res, next) => {

  logger.info(`${req.method} ${req.url} - ${new
Date().toISOString()}`);

  next();

});
```

```
app.get('/api/test', (req, res) => {
  res.send("API is working!");
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

✓ Each request is logged in **requests.log** with the **HTTP method, endpoint, and timestamp**.

3.2 Logging API Errors

Errors should be logged separately to help in debugging:

```
app.use((err, req, res, next) => {
  logger.error(`Error: ${err.message} - ${new Date().toISOString()}`);
  res.status(500).json({ error: "Internal Server Error" });
});
```

✓ Logs **unexpected API errors** for troubleshooting.

CHAPTER 4: MONITORING API PERFORMANCE WITH PROMETHEUS AND GRAFANA

4.1 Why Use API Monitoring?

API monitoring helps:

✓ Track **response times** and latency.
✓ Detect **downtime and failures** in real-time.
✓ Improve **API performance** by identifying slow endpoints.

4.2 Setting Up Prometheus for API Monitoring

Prometheus is an open-source **monitoring tool** that collects API performance metrics.

Step 1: Install Prometheus

Download and install Prometheus:

```
wget https://github.com/prometheus/prometheus/releases/latest/download/prometheus-*.linux-amd64.tar.gz

tar -xvzf prometheus-*.linux-amd64.tar.gz

cd prometheus-*.linux-amd64
```

Step 2: Configure Prometheus to Monitor Node.js API

Edit prometheus.yml:

```
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: 'node-api'
    static_configs:
      - targets: ['localhost:3000']
```

Start Prometheus:

```
./prometheus --config.file=prometheus.yml
```

✓ **Prometheus collects API metrics every 5 seconds**.

CHAPTER 5: VISUALIZING API METRICS WITH GRAFANA

5.1 Why Use Grafana for API Monitoring?

Grafana is an open-source tool that:

✓ Provides **real-time API performance dashboards**.
✓ Integrates with **Prometheus to display API metrics**.
✓ Allows setting up **alerts for high response times**.

Step 1: Install Grafana

For Linux/macOS:

sudo apt-get install -y grafana

Start Grafana:

sudo systemctl start grafana-server

Access Grafana at http://localhost:3000/, and configure Prometheus as a **data source** to start monitoring API performance.

Case Study: How an E-Commerce Platform Used API Monitoring to Improve Performance

Background

A leading **e-commerce company** faced frequent **API slowdowns** during peak shopping hours. Customers reported **delayed responses**, and sales were impacted.

Challenges

✓ API response times **exceeded 3 seconds** for some endpoints.
✓ The **checkout process crashed** intermittently, leading to abandoned carts.
✓ No **monitoring system** to detect API failures in real time.

Solution: Implementing API Logging and Monitoring

The development team implemented:

✓ **Winston logging** to track API errors.
✓ **Prometheus for API performance monitoring**.
✓ **Grafana dashboards** to visualize response times.

Results

- **API response times improved by 40%** after optimizing slow endpoints.
- **Proactive issue detection,** reducing downtime by **90%**.
- **Increased customer satisfaction,** leading to **higher conversions**.

This case study highlights how **monitoring and logging improve API reliability**.

Exercise

1. Set up **Winston logging** to record API requests in an **Express.js server**.
2. Implement **error logging middleware** to capture failed API requests.
3. Configure **Prometheus to collect API performance metrics**.
4. Use **Grafana to create a dashboard** that tracks API response times.

Conclusion

In this section, we explored:
✓ **How API logging improves debugging and security**.

✓ **How API monitoring detects performance bottlenecks.**

✓ **How to integrate Prometheus and Grafana for real-time API tracking.**

FINAL CAPSTONE PROJECT: DEVELOP & DEPLOY A FULL-STACK NODE.JS APP WITH AUTHENTICATION, REAL-TIME FEATURES, AND A DATABASE

FINAL CAPSTONE PROJECT: FULL-STACK NODE.JS APP WITH AUTHENTICATION, REAL-TIME FEATURES & DATABASE

Step 1: Set Up the Project

1.1 Create a New Project Directory

mkdir fullstack-node-app

cd fullstack-node-app

1.2 Initialize a Node.js Project

npm init -y

1.3 Install Dependencies

npm install express mongoose jsonwebtoken bcryptjs cors dotenv socket.io

✓ **Express.js** – Web framework for the API.
✓ **Mongoose** – ODM for MongoDB.
✓ **JWT** – User authentication.
✓ **bcrypt.js** – Secure password hashing.
✓ **CORS** – Allows API access from frontend.
✓ **dotenv** – Manages environment variables.
✓ **Socket.io** – Enables real-time features.

Step 2: Configure MongoDB Database

2.1 Set Up Database Connection

Create a **.env** file to store database credentials:

```
MONGO_URI=mongodb://localhost:27017/chatApp
JWT_SECRET=mysecretkey
PORT=5000
```

Create **config/db.js**:

```
const mongoose = require('mongoose');
require('dotenv').config();

mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
.catch(err => console.error('MongoDB Connection Error:', err));

module.exports = mongoose;
```

Step 3: Set Up User Authentication (JWT)

3.1 Create a User Model

Create **models/User.js**:

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
```

```
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

module.exports = mongoose.model('User', userSchema);
```

3.2 Implement User Authentication Routes

Create **routes/auth.js**:

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
require('dotenv').config();

const router = express.Router();

// User Registration
router.post('/register', async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ name, email, password:
hashedPassword });
```

```
    await newUser.save();
    res.status(201).json({ message: 'User registered successfully!' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// User Login
router.post('/login', async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });

    if (!user || !await bcrypt.compare(password, user.password)) {
      return res.status(401).json({ error: 'Invalid email or password' });
    }

    const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });

    res.json({ message: 'Login successful', token });
  } catch (error) {
```

```
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

Step 4: Implement Real-Time Chat Using WebSockets

4.1 Create a WebSocket Server

Modify **server.js** to handle WebSocket connections:

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const mongoose = require('./config/db');
const authRoutes = require('./routes/auth');

require('dotenv').config();

const app = express();
const server = http.createServer(app);
const io = new Server(server, {
  cors: { origin: '*' }
});
```

```
app.use(express.json());
app.use('/auth', authRoutes);

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

  socket.on('send-message', (message) => {
    console.log('New Message:', message);
    io.emit('receive-message', message);
  });

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

const PORT = process.env.PORT || 5000;
server.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ Users send messages using send-message.
✓ Messages are **broadcasted** to all connected clients.

Step 5: Create a Frontend with React.js

5.1 Initialize React App

npx create-react-app chat-client

cd chat-client

npm install socket.io-client axios

5.2 Create Chat Component

Modify **src/components/Chat.js**:

```
import React, { useState, useEffect } from 'react';
import io from 'socket.io-client';

const socket = io('http://localhost:5000');

function Chat() {
  const [message, setMessage] = useState('');
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    socket.on('receive-message', (msg) => {
      setMessages((prev) => [...prev, msg]);
    });

    return () => socket.off('receive-message');
```

```
  }, []);

  const sendMessage = () => {
    socket.emit('send-message', message);
    setMessage('');
  };

  return (
    <div>
      <h2>Chat Room</h2>
      <div>
        {messages.map((msg, index) => <p key={index}>{msg}</p>)}
      </div>
      <input type="text" value={message} onChange={(e) =>
setMessage(e.target.value)} />
      <button onClick={sendMessage}>Send</button>
    </div>
  );
}

export default Chat;
```

✓ **Messages appear instantly** for all users.
✓ Users can send and receive messages **in real time**.

Step 6: Deploy the Full-Stack App

6.1 Deploy Backend to Heroku

1. Install the Heroku CLI:
2. npm install -g heroku
3. Login to Heroku:
4. heroku login
5. Initialize Git and push code:
6. git init
7. heroku create chat-app
8. git add .
9. git commit -m "Initial commit"
10. git push heroku master

6.2 Deploy React Frontend to Netlify

1. Build the frontend:
2. npm run build
3. Deploy with Netlify:
4. netlify deploy --prod

Case Study: How a Startup Used WebSockets to Enhance User Engagement

Background

A startup needed a **real-time messaging platform** for customer support.

Challenges

✓ Users **had to refresh the page** to see new messages.
✓ **High server load** due to polling every second.

Solution

✓ Implemented **Socket.io** for real-time updates.
✓ Used **MongoDB** to store chat history.
✓ **Optimized authentication** with JWT.

Results

✓ **50% lower server load**.
✓ **Instant messaging**, improving user experience.

Conclusion

✓ We built a **full-stack app** with **authentication, real-time chat, and a database**.
✓ We deployed the **backend on Heroku** and the **frontend on Netlify**.
✓ This project can be expanded with **file uploads, typing indicators, and chat rooms**. 🚀