Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

DEPLOYMENT, TESTING & ADVANCED TOPICS (WEEKS 11-12)

DEPLOYING TO HEROKU, VERCEL, AND AWS

CHAPTER 1: INTRODUCTION TO DEPLOYMENT

1.1 What is Deployment?

Deployment is the process of **making an application accessible on the internet** by hosting it on a cloud platform. It involves **configuring servers, databases, and runtime environments** to ensure smooth functionality.

- **Why is Deployment Important?**

✓ Allows **users to access applications from anywhere**.
✓ Ensures **scalability, security, and reliability**.
✓ Enables **automatic updates and maintenance**.

- **Popular Cloud Platforms for Deployment:**

Platform	Best For	Example Use Case
Heroku	Simple deployment	Small web apps & APIs

Vercel	Serverless deployment	Frontend & full-stack apps
AWS (Amazon Web Services)	Scalable cloud infrastructure	Large-scale applications

---

CHAPTER 2: DEPLOYING AN EXPRESS.JS APP TO HEROKU

2.1 Installing the Heroku CLI

📌 Step 1: Install Heroku CLI

Download and install **Heroku CLI** from Heroku's website.

📌 Step 2: Login to Heroku

```
heroku login
```

✓ Opens a browser window to authenticate your Heroku account.

2.2 Preparing the Express App for Deployment

📌 Step 3: Ensure package.json has a Start Script

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

✓ Ensures Heroku knows how to **start the app**.

📌 Step 4: Add a Procfile in the Project Root

```
web: node server.js
```

✓ Informs Heroku **how to run the app**.

2.3 Deploying to Heroku

📌 **Step 5: Initialize Git Repository**

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

✓ Prepares the project for deployment.

📌 **Step 6: Create a New Heroku App**

```
heroku create my-express-app
```

✓ Creates a **new app on Heroku** with a unique URL.

📌 **Step 7: Deploy the App**

```
git push heroku main
```

✓ Pushes the **code to Heroku** for deployment.

📌 **Step 8: Open the App**

```
heroku open
```

✓ Opens the deployed **Express.js app in a browser**.

📌 **Example Live URL:**

https://my-express-app.herokuapp.com

✓ **Express API is now live on Heroku!**

CHAPTER 3: DEPLOYING AN EXPRESS.JS APP TO VERCEL

3.1 Installing Vercel CLI

📌 **Step 1: Install Vercel CLI Globally**

npm install -g vercel

✓ Installs Vercel's deployment tool.

📌 **Step 2: Login to Vercel**

vercel login

✓ Authenticates with **your Vercel account**.

3.2 Preparing the Express App for Deployment

📌 **Step 3: Modify server.js for Vercel**

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Hello from Vercel!");
});

module.exports = app;
```

✓ Exports app for Vercel's **serverless deployment**.

📌 **Step 4: Create vercel.json Configuration File**

```
{
 "version": 2,
 "builds": [{ "src": "server.js", "use": "@vercel/node" }],
 "routes": [{ "src": "/(.*)", "dest": "server.js" }]
```

}

✓ Configures **Vercel to deploy the Express app**.

3.3 Deploying to Vercel

📌 **Step 5: Initialize Vercel in the Project Directory**

vercel

✓ **Follow the prompts** to deploy the app.

📌 **Step 6: View Deployed App**

Vercel provides a **live deployment URL** like:

https://my-express-app.vercel.app

✓ **Your API is now live on Vercel!**

CHAPTER 4: DEPLOYING AN EXPRESS.JS APP TO AWS (AMAZON WEB SERVICES)

4.1 Setting Up an EC2 Instance for Deployment

📌 **Step 1: Create an AWS EC2 Instance**

1. Go to **AWS Console → EC2**.
2. Click **Launch Instance** and select **Ubuntu 20.04 LTS**.
3. Choose instance type **t2.micro (Free Tier)**.
4. Set up **security group (allow SSH, HTTP, and HTTPS access)**.
5. Click **Launch** and download the **.pem key file**.

📌 **Step 2: Connect to EC2 Instance via SSH**

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

✓ Connects to the **AWS server** remotely.

4.2 Installing Node.js & Setting Up Express.js

📌 Step 3: Install Node.js & Git on the EC2 Instance

sudo apt update

sudo apt install -y nodejs npm git

✓ Installs **Node.js and npm** on AWS.

📌 Step 4: Clone Your Express.js App

git clone https://github.com/your-repo/express-app.git

cd express-app

npm install

✓ Retrieves and **sets up your API** on the EC2 instance.

4.3 Running the Express App on AWS

📌 Step 5: Start the Server

node server.js

✓ API runs but **stops when SSH disconnects**.

📌 Step 6: Use pm2 for Process Management

npm install -g pm2

pm2 start server.js

pm2 save

pm2 startup

✓ Ensures **server runs in the background** even after logout.

📌 **Step 7: Configure Security Group for API Access**

1. Go to **AWS Console → EC2 → Security Groups**.
2. Allow **port 3000 for public access**.

📌 **Step 8: Access Deployed API**

http://your-ec2-ip:3000

✓ Your **Express API is now live on AWS!**

CHAPTER 5: BEST PRACTICES FOR CLOUD DEPLOYMENT

- **1. Use Environment Variables for Security**

✓ Store API keys **in a .env file**.

DATABASE_URL=mongodb+srv://username:password@cluster.mongodb.net/

- **2. Use Process Managers for Stability**

✓ **PM2** keeps servers **running in production**.

pm2 restart server.js

- **3. Monitor Logs & Performance**

✓ Use logging tools like **AWS CloudWatch** or **Heroku Logs**.

- **4. Enable Auto-Scaling for High Traffic**

✓ AWS **Auto Scaling Groups** adjust **server capacity dynamically**.

Case Study: How Netflix Uses Cloud Deployment

Challenges Faced by Netflix

✓ Handling **millions of user requests globally**.
✓ Ensuring **scalable infrastructure for video streaming**.

Solutions Implemented

✓ Used **AWS EC2 instances** to host APIs.
✓ Deployed **serverless microservices on AWS Lambda**.
✓ Implemented **CDN caching for fast content delivery**.

- **Key Takeaways from Netflix's Cloud Deployment Strategy:**

✓ **Auto-scaling ensures performance under high traffic.**
✓ **Using CDNs reduces server load and speeds up API responses.**
✓ **Cloud-based deployments improve reliability and uptime.**

📝 Exercise

✅ Deploy an **Express.js app on Heroku** and access its public URL.
✅ Deploy an **Express API on Vercel** using the CLI.
✅ Set up an **EC2 instance on AWS** and deploy an Express.js server.
✅ Use **PM2 to keep the Node.js server running on AWS**.

Conclusion

✓ **Heroku is great for simple Express.js deployments.**
✓ **Vercel offers serverless deployment for fast APIs.**
✓ **AWS provides scalable infrastructure for large applications.**
✓ **Following best practices ensures smooth cloud deployments.**

USING DOCKER & KUBERNETES FOR CONTAINERIZATION

CHAPTER 1: INTRODUCTION TO CONTAINERIZATION

1.1 What is Containerization?

Containerization is a **virtualization method** that packages applications and their dependencies into **lightweight, portable containers**. Unlike traditional **virtual machines (VMs)**, containers share the **host OS kernel**, making them more efficient and faster.

- **Why Use Containers?**

✓ **Portability** – Runs consistently across different environments.
✓ **Scalability** – Easily scale applications across multiple instances.
✓ **Efficiency** – Uses fewer resources compared to VMs.
✓ **Faster Deployment** – No need to reconfigure dependencies.

- **Containers vs. Virtual Machines (VMs)**

Feature	Containers	Virtual Machines (VMs)
Speed	Faster startup	Slower boot time
Resource Usage	Lightweight, shares OS kernel	Heavy, includes OS per VM
Scalability	Easily scalable	Requires additional VMs
Portability	Works anywhere with Docker	OS-dependent

📌 **Example: Running an Application Without Containers**

- Install Node.js, Express, MongoDB manually.
- Configure dependencies & environment variables.
- Troubleshoot differences between development & production.

📌 **Example: Running the Same Application in a Container**

- Use docker-compose to define everything.
- One command starts everything (docker-compose up).
- Runs identically across different machines.

CHAPTER 2: INTRODUCTION TO DOCKER

2.1 What is Docker?

Docker is a **containerization platform** that allows developers to build, ship, and run applications in isolated environments. It ensures that software **runs the same, regardless of where it is deployed**.

🔹 **Why Use Docker?**

✓ Eliminates **"It works on my machine"** problems.
✓ Automates **application setup and deployment**.
✓ Provides **efficient resource usage**.

📌 **Installing Docker:**

Download Docker from Docker Official Site.

📌 **Verify Installation:**

```
docker --version
```

✓ Confirms **Docker is installed**.

CHAPTER 3: CREATING A DOCKERIZED EXPRESS.JS APP

3.1 Writing a Dockerfile

A **Dockerfile** is a script that tells Docker **how to build an image** for an application.

📌 **Create a Dockerfile in Your Project:**

```
# Use Node.js base image
FROM node:16

# Set the working directory inside the container
WORKDIR /app

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

# Copy application files
COPY . .

# Expose the port the app runs on
EXPOSE 5000

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

✓ **Defines container steps** for setting up the app.

3.2 Building and Running the Docker Image

📌 Step 1: Build the Docker Image

docker build -t my-express-app .

✓ **Creates a Docker image** named my-express-app.

📌 Step 2: Run the Docker Container

docker run -p 5000:5000 my-express-app

✓ **Starts a container** that runs the Express.js app on port **5000**.

📌 Step 3: Verify Running Containers

docker ps

✓ Lists all **active Docker containers**.

3.3 Using docker-compose to Manage Multi-Container Apps

For real-world apps, you need **multiple services** (API, database, caching). docker-compose simplifies multi-container setups.

📌 Create a docker-compose.yml File:

```
version: '3'

services:

 app:

  build: .

  ports:

   - "5000:5000"
```

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

✓ Defines **two containers**:
✓ app (Express.js server)
✓ mongo (MongoDB instance)

📌 **Run Everything with One Command:**

```
docker-compose up
```

✓ Starts **both the Express.js app and MongoDB together**.

CHAPTER 4: INTRODUCTION TO KUBERNETES

4.1 What is Kubernetes?

Kubernetes (K8s) is a **container orchestration tool** that manages **deployments, scaling, and networking** of containerized applications.

🔹 **Why Use Kubernetes?**

✓ **Automates container deployment** across multiple machines.
✓ **Handles scaling dynamically** based on traffic.
✓ **Self-healing** – Restarts failed containers automatically.

📌 **Kubernetes vs. Docker**

Feature	Docker	Kubernetes

Scope	Runs single containers	Manages multiple containers
Scaling	Manual	Automatic
Networking	Basic	Advanced load balancing

4.2 Setting Up a Kubernetes Cluster

📌 **Step 1: Install Minikube for Local Kubernetes Testing**

minikube start

✓ Starts a **local Kubernetes cluster**.

📌 **Step 2: Verify Cluster is Running**

kubectl get nodes

✓ Confirms **Kubernetes is running**.

CHAPTER 5: DEPLOYING AN EXPRESS.JS APP ON KUBERNETES

5.1 Creating a Kubernetes Deployment

📌 **Create deployment.yaml** to Define a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: express-app
spec:
  replicas: 2
```

```
 selector:
  matchLabels:
   app: express
 template:
  metadata:
   labels:
    app: express
  spec:
   containers:
    - name: express-container
      image: my-express-app
      ports:
       - containerPort: 5000
```

✓ Creates **two replicas** of the Express.js app.

📌 **Apply the Deployment to Kubernetes**

```
kubectl apply -f deployment.yaml
```

✓ Deploys the **containerized Express.js app** to Kubernetes.

5.2 Exposing the App with a Service

📌 **Create service.yaml to Expose Express App**

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: express-service
spec:
  type: NodePort
  selector:
    app: express
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
      nodePort: 30001
```

✓ Allows the app to be accessed via **port 30001**.

📌 **Apply the Service Configuration**

```
kubectl apply -f service.yaml
```

📌 **Access the App on Kubernetes**

```
minikube service express-service --url
```

✓ **Opens the deployed Express app** in the browser.

Case Study: How Netflix Uses Docker & Kubernetes

Challenges Faced by Netflix

✓ Handling **millions of containerized services**.
✓ Ensuring **zero downtime deployments**.

Solutions Implemented

✓ Used **Docker to package microservices**.
✓ Deployed **millions of containers** using Kubernetes.
✓ Implemented **auto-scaling** based on traffic.

🔹 **Key Takeaways from Netflix's Container Strategy:**
✓ **Docker ensures portability & consistency.**
✓ **Kubernetes automates scaling & self-healing.**
✓ **Microservices architecture improves performance.**

📝 Exercise

✅ Build a **Dockerfile** for a simple Express.js app.
✅ Use **docker-compose** to set up an Express.js & MongoDB project.
✅ Deploy an Express.js app on **Kubernetes using Minikube**.
✅ Configure Kubernetes **auto-scaling for high-traffic scenarios**.

Conclusion

✓ **Docker enables efficient containerization of Node.js applications.**
✓ **Kubernetes manages, scales, and orchestrates multiple containers.**
✓ **Using Docker & Kubernetes improves deployment efficiency and reliability.**
✓ **Cloud-native applications use Kubernetes for auto-scaling & fault tolerance.**

MANAGING ENVIRONMENT VARIABLES & SECRETS IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO ENVIRONMENT VARIABLES & SECRETS

1.1 What are Environment Variables?

Environment variables are **key-value pairs** stored outside the application code to configure an application dynamically. They are used to **store sensitive information and application settings securely**.

- **Why Use Environment Variables?**

✓ **Security** – Protects sensitive data (API keys, database credentials).

✓ **Portability** – Allows code to run on different environments (dev, staging, production).

✓ **Configurability** – Easily update settings without modifying the code.

- **Examples of Environment Variables:**

Variable	Purpose	Example
PORT	Defines the server port	3000
MONGO_URI	MongoDB database connection string	mongodb://localhost:27017/mydb

JWT_SECRET	Secret key for authentication tokens	mySuperSecretKey123
API_KEY	External API authentication key	1234-5678-ABCD

CHAPTER 2: USING .ENV FILES IN EXPRESS.JS

2.1 Installing dotenv for Managing Environment Variables

📌 **Install the dotenv package to load environment variables**

npm install dotenv

✓ dotenv **loads environment variables from a .env file** into process.env.

📌 **Create a .env File in the Project Root Directory**

PORT=5000

MONGO_URI=mongodb://127.0.0.1:27017/mydb

JWT_SECRET=mySuperSecretKey123

✓ Defines **environment variables** in a secure file.

2.2 Loading Environment Variables in Express.js

📌 **Modify server.js to Use Environment Variables**

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

const express = require("express");

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

```
const app = express();
const PORT = process.env.PORT || 3000;
const MONGO_URI = process.env.MONGO_URI;

// Connect to MongoDB
mongoose.connect(MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"))
 .catch(err => console.error("MongoDB Connection Error:", err));

app.get("/", (req, res) => {
  res.send(`Server running on port ${PORT}`);
});

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ Loads **database credentials & server port** from .env.

📌 **Start the Server:**

```
node server.js
```

✓ The server **runs on the port defined in .env**.

CHAPTER 3: SECURING ENVIRONMENT VARIABLES

3.1 Adding .env to .gitignore

📌 Prevent .env from Being Uploaded to GitHub

```
# Add this to .gitignore
.env
```

✓ Ensures **sensitive data is never committed** to version control.

3.2 Using Environment Variables in Production

🔹 1. Setting Variables in Linux/MacOS

```
export PORT=5000
export MONGO_URI="mongodb://127.0.0.1:27017/mydb"
node server.js
```

✓ Sets variables **without needing a .env file**.

🔹 2. Setting Variables in Windows (PowerShell)

```
$env:PORT="5000"
$env:MONGO_URI="mongodb://127.0.0.1:27017/mydb"
node server.js
```

✓ Useful for **deploying on cloud platforms (AWS, DigitalOcean, Heroku)**.

CHAPTER 4: MANAGING SECRETS SECURELY

4.1 Storing Secrets in AWS Secrets Manager

Cloud services like **AWS Secrets Manager** store environment variables securely.

📌 Steps to Use AWS Secrets Manager for Express.js

1. **Create a Secret** in AWS Secrets Manager (e.g., MongoDB credentials).
2. **Retrieve Secret Dynamically in Express.js**

```
const AWS = require("aws-sdk");

const secretsManager = new AWS.SecretsManager();


async function getSecret(secretName) {
  const secret = await secretsManager.getSecretValue({ SecretId: secretName }).promise();
  return JSON.parse(secret.SecretString);
}
```

✓ Keeps sensitive data **outside the codebase**.

4.2 Using .env.example for Shared Configurations

📌 Create a Sample .env.example File for Teams

```
PORT=your_port_here

MONGO_URI=your_database_uri_here

JWT_SECRET=your_secret_here
```

✓ Helps **teams share required environment variables** without exposing secrets.

4.3 Encrypting Environment Variables

📌 Example: Encrypting and Decrypting Secrets

openssl enc -aes-256-cbc -salt -in .env -out .env.enc -k mypassword

✓ Encrypts .env for **secure storage**.

To decrypt:

openssl enc -aes-256-cbc -d -in .env.enc -out .env -k mypassword

✓ Ensures only **authorized users can access sensitive data**.

Case Study: How Netflix Manages API Secrets Securely

Challenges Faced by Netflix

✓ Handling **millions of API requests securely**.
✓ Protecting **sensitive credentials (API keys, database URIs, authentication tokens)**.

Solutions Implemented

✓ **Stored API secrets in AWS Secrets Manager**.
✓ **Used encrypted environment variables** for internal services.
✓ **Automated key rotation for sensitive tokens**.

- **Key Takeaways from Netflix's Security Strategy:**

✓ **Never store secrets in source code repositories.**
✓ **Use cloud-based secret managers for high-security**

applications.

✓ Apply encryption for highly sensitive data.

📝 Exercise

✅ Implement **dotenv** in an Express.js project to manage environment variables.

✅ Add .env to .gitignore and **test loading variables dynamically**.

✅ Store **MongoDB credentials in an AWS Secrets Manager** and retrieve them.

✅ Encrypt and decrypt an .env file for **extra security**.

Conclusion

✓ Environment variables enhance security & configurability in Express.js apps.

✓ Secrets should never be committed to Git repositories.

✓ Cloud-based secret management (AWS Secrets Manager) improves security.

✓ Encrypting sensitive data prevents unauthorized access.

WRITING UNIT & INTEGRATION TESTS WITH JEST & SUPERTEST

CHAPTER 1: INTRODUCTION TO TESTING IN NODE.JS

1.1 Why Test Your Express.js APIs?

Testing is crucial for **ensuring application reliability** by catching bugs early and validating functionality. **Unit and integration tests** help maintain code quality, prevent regressions, and improve application stability.

- **Types of Testing in Express.js APIs:**

Type	Description	Example
Unit Testing	Tests individual functions	Testing a utility function
Integration Testing	Tests multiple components together	API requests hitting the database
End-to-End (E2E) Testing	Simulates real user interactions	Testing a full login flow

- **Why Use Jest & Supertest?**

✓ **Jest** – JavaScript testing framework for unit and integration tests.

✓ **Supertest** – Library for making HTTP requests to Express.js APIs.

📌 **Installing Jest & Supertest in an Express.js Project:**

npm install --save-dev jest supertest

✓ Installs Jest for **test execution** and Supertest for **API testing**.

CHAPTER 2: SETTING UP JEST IN AN EXPRESS.JS PROJECT

2.1 Configuring Jest in package.json

📌 Modify package.json to Include Jest Configuration:

```
"scripts": {
  "test": "jest"
}
```

✓ Now, run tests using:

```
npm test
```

✓ Jest automatically **detects test files (.test.js or .spec.js)**.

CHAPTER 3: WRITING UNIT TESTS WITH JEST

3.1 Testing a Utility Function

📌 Create a Simple Utility Function (utils/math.js)

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

module.exports = add;
```

✓ **Basic addition function** for testing.

📌 Write Unit Test for Addition (tests/math.test.js)

```
const add = require("../utils/math");
```

```
test("adds 2 + 3 to equal 5", () => {
  expect(add(2, 3)).toBe(5);
});
```

✓ Uses Jest's expect() function for assertion.

📌 **Run the Test:**

```
npm test
```

✓ Should output:

```
PASS tests/math.test.js
✓ adds 2 + 3 to equal 5
```

✓ Confirms **correct function execution**.

CHAPTER 4: WRITING INTEGRATION TESTS FOR EXPRESS.JS APIS

4.1 Setting Up an Express API for Testing

📌 **Create server.js with a Simple API Endpoint**

```
const express = require("express");
const app = express();

app.use(express.json());

app.get("/api/greet", (req, res) => {
  res.json({ message: "Hello, World!" });
});
```

module.exports = app; // Export app for testing

✓ **Returns a simple JSON response**.

4.2 Writing an Integration Test with Supertest

📌 Create a Test File (tests/server.test.js)

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

test("GET /api/greet should return Hello, World!", async () => {
  const res = await request(app).get("/api/greet");

  expect(res.status).toBe(200);
  expect(res.body.message).toBe("Hello, World!");
});
```

✓ Uses Supertest's request() to **call the API endpoint**.
✓ Validates **status code and response data**.

📌 Run the Test:

```
npm test
```

✓ Outputs:

```
PASS tests/server.test.js
✓ GET /api/greet should return Hello, World!
```

✓ Confirms **API response works correctly**.

Chapter 5: Testing CRUD Operations in Express.js

5.1 Setting Up a Basic CRUD API

📌 **Modify server.js to Include User Management Routes**

```
const users = [{ id: 1, name: "Alice" }];


app.get("/api/users", (req, res) => res.json(users));


app.post("/api/users", (req, res) => {
  const newUser = { id: users.length + 1, name: req.body.name };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

✓ Provides **GET and POST API routes** for users.

5.2 Writing Tests for CRUD API

📌 **Modify tests/server.test.js to Include CRUD Tests**

```
test("GET /api/users should return all users", async () => {
  const res = await request(app).get("/api/users");


  expect(res.status).toBe(200);
```

```
  expect(res.body.length).toBeGreaterThan(0);
});

test("POST /api/users should create a new user", async () => {
  const newUser = { name: "Bob" };
  const res = await request(app).post("/api/users").send(newUser);

  expect(res.status).toBe(201);
  expect(res.body.name).toBe("Bob");
});
```

✓ Ensures **new users are created correctly**.

📌 **Run Tests:**

```
npm test
```

✓ Confirms CRUD API **functions correctly**.

CHAPTER 6: TESTING EXPRESS.JS WITH A MONGODB DATABASE

6.1 Setting Up a Mongoose Test Database

📌 **Modify server.js to Connect to MongoDB**

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

mongoose.connect("mongodb://127.0.0.1:27017/testDB", {
  useNewUrlParser: true,
```

```
  useUnifiedTopology: true
});
```

✓ Uses a **test database for running API tests**.

6.2 Writing Tests for Database Queries

📌 Create a Mongoose Model (models/User.js)

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

const UserSchema = new mongoose.Schema({
  name: String
});

module.exports = mongoose.model("User", UserSchema);
```

✓ Defines a **User model for testing database operations**.

📌 Modify tests/server.test.js to Include Database Tests

```
const User = require("../models/User");

beforeAll(async () => {
  await mongoose.connection.dropDatabase(); // Clear database before tests
});
```

```
test("POST /api/users should save user in the database", async () => {
  const res = await request(app).post("/api/users").send({ name: "Charlie" });

  const savedUser = await User.findOne({ name: "Charlie" });
  expect(savedUser).not.toBeNull();
});
```

✓ Uses **beforeAll() to reset the database before tests**.

📌 **Run Database Tests:**

```
npm test
```

✓ Confirms data **is stored correctly in MongoDB**.

CHAPTER 7: BEST PRACTICES FOR JEST & SUPERTEST

- **1. Reset Test Data Before Running Tests**

✓ Use **beforeAll() and afterAll()** to clear data.

- **2. Mock External API Calls**

✓ Use jest.fn() to **mock API responses**.

```
const fetchData = jest.fn().mockResolvedValue({ name: "Test User" });
```

- **3. Use Separate Test Databases**

✓ Connect to **a dedicated MongoDB test instance** instead of production.

- **4. Measure Code Coverage with Jest**

✓ Run tests with code coverage reports:

npm test -- --coverage

Case Study: How Airbnb Uses Automated API Testing

Challenges Faced by Airbnb

✓ Handling **millions of API requests**.
✓ Ensuring **data integrity across microservices**.

Solutions Implemented

✓ **Wrote unit tests for all API endpoints**.
✓ **Implemented integration tests for user authentication flows**.
✓ **Used CI/CD pipelines to run Jest tests automatically**.

- **Key Takeaways from Airbnb's Testing Strategy:**

✓ **Automated tests prevent regressions.**
✓ **Testing databases ensure data integrity.**
✓ **API performance improves with efficient testing.**

Conclusion

✓ **Jest is ideal for unit testing Express.js functions.**
✓ **Supertest simplifies API testing in Express.js applications.**
✓ **Database tests ensure data integrity in MongoDB-based applications.**
✓ **Best practices like mocking and coverage reports improve test quality.**

IMPLEMENTING CI/CD PIPELINES USING GITHUB ACTIONS

CHAPTER 1: INTRODUCTION TO CI/CD AND GITHUB ACTIONS

1.1 What is CI/CD?

CI/CD (Continuous Integration and Continuous Deployment) are software development practices that automate and streamline the process of integrating code changes, testing, and deploying applications.

- **Continuous Integration (CI):**
  - The process of **automatically integrating code changes** from multiple contributors into a shared repository.
  - The goal is to detect integration issues early through **automated testing** of code before merging.
- **Continuous Deployment (CD):**
  - After CI ensures the code is error-free, **CD automates the release process** by deploying code changes to production automatically.
  - CD focuses on **speed and reliability** in delivering code to end-users.

📌 **Example:** A developer pushes code changes to a GitHub repository. **GitHub Actions** automatically runs tests, builds the project, and deploys it if all tests pass.

📌 **Exercise:** Define **CI/CD** in your own words and explain the difference between **Continuous Integration** and **Continuous Deployment**.

Chapter 2: Introduction to GitHub Actions

2.1 What is GitHub Actions?

GitHub Actions is an automation tool built into GitHub that allows developers to define workflows for automating tasks like **building, testing, and deploying code**. GitHub Actions simplifies the creation of **CI/CD pipelines** by providing an easy-to-use YAML configuration to define workflows.

- **Key Features of GitHub Actions:**
  - **Workflow Automation:** Automate processes like **pull request reviews, issue triage,** and **CI/CD pipelines**.
  - **Integrations with GitHub Events:** Trigger actions based on **repository events** (e.g., pushes, pull requests, releases).
  - **Customizable Workflows:** Create reusable workflows for different use cases (e.g., testing, deployment).
  - **Matrix Builds:** Run jobs in parallel for different environments or configurations (e.g., different versions of Node.js).

📌 **Example:** A GitHub Actions workflow might be triggered every time you push a new commit to your repository, automatically running unit tests and deploying the code if everything passes.

📌 **Exercise:** Define **GitHub Actions** in your own words and list three benefits of using it for automating CI/CD pipelines.

Chapter 3: Setting Up a GitHub Actions Workflow

3.1 Creating Your First GitHub Action

To start using GitHub Actions, you need to create a **workflow file** in your repository. This file defines all the steps required for the pipeline, such as building, testing, and deploying your application.

🔹 **Steps to Set Up a GitHub Action:**

1. **Create the .github/workflows directory** in your repository.
2. Inside the **workflows folder**, create a new **YAML file** (e.g., ci-cd-pipeline.yml).
3. Define the workflow configuration using **GitHub Actions syntax**.

📌 **Example of a Simple Workflow:**

```
name: CI/CD Pipeline

on:
 push:
  branches:
   - main
jobs:
 build:
  runs-on: ubuntu-latest
  steps:
   - name: Checkout code
    uses: actions/checkout@v2
   - name: Set up Node.js
    uses: actions/setup-node@v2
    with:
```

```
    node-version: '14'
- name: Install dependencies
  run: npm install
- name: Run tests
  run: npm test
```

This workflow is triggered on every push to the main branch, checks out the code, sets up Node.js, installs dependencies, and runs tests.

📌 **Exercise:** Define **workflow file** in your own words and list three essential components you must include in a basic GitHub Actions workflow.

3.2 Understanding Workflow Syntax

GitHub Actions workflows are defined using **YAML syntax**. Here's a breakdown of the key components:

🔹 **Key Workflow Components:**

- **name:** Describes the name of the workflow (e.g., CI/CD Pipeline).
- **on:** Specifies the event that triggers the workflow (e.g., push, pull_request).
- **jobs:** Defines the individual tasks to be run in the pipeline (e.g., build, deploy).
- **runs-on:** Defines the type of virtual machine or container used (e.g., ubuntu-latest).
- **steps:** Lists the individual steps (actions or commands) to be executed in a job.

📌 **Example:** In the YAML example, the workflow is triggered on **push events to the main branch**. The job runs on an **Ubuntu runner**, performs the steps for checking out the code, setting up Node.js, installing dependencies, and running tests.

📌 **Exercise:** Define **jobs and steps** in GitHub Actions in your own words and list three steps you might include in a **build job**.

CHAPTER 4: ADVANCED CI/CD PIPELINES WITH GITHUB ACTIONS

4.1 Running Tests in Parallel (Matrix Builds)

Matrix builds in GitHub Actions allow you to run the same set of jobs across multiple environments or configurations in parallel. This can significantly reduce the testing time for projects with dependencies on various versions of programming languages or tools.

- **Example of a Matrix Build:**

```
jobs:
 test:
  runs-on: ubuntu-latest
  strategy:
   matrix:
    node: [12, 14, 16]
  steps:
   - name: Checkout code
    uses: actions/checkout@v2
   - name: Set up Node.js
```

```
    uses: actions/setup-node@v2
    with:
      node-version: ${{ matrix.node }}
  - name: Install dependencies
    run: npm install
  - name: Run tests
    run: npm test
```

In the above example, the test job will run three times in parallel with **Node.js versions 12, 14, and 16**.

📌 **Exercise:** Define **matrix builds** in your own words and list three scenarios where matrix builds would be useful in a CI/CD pipeline.

4.2 Continuous Deployment (CD) with GitHub Actions

Once your code is successfully tested and validated, you can use GitHub Actions to automatically deploy it to your staging or production environments.

🔹 **Steps to Set Up Continuous Deployment:**

1. Set up a **deployment environment** (e.g., AWS, Heroku, DigitalOcean).
2. Create a **deployment action** in your workflow that automates the process of deploying your application to the desired environment.

📌 **Example: Deploying to AWS S3:**

```
jobs:
```

```
deploy:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Install AWS CLI
      run: |
        curl "https://d1vvhvl2y92vvt.cloudfront.net/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
        unzip awscliv2.zip
        sudo ./aws/install
    - name: Deploy to S3
      run: |
        aws s3 sync ./build s3://my-bucket-name --delete
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

This workflow will deploy the built code to an **AWS S3 bucket** after all tests pass.

📌 **Exercise:** Define **continuous deployment** in your own words and list three tools or platforms you can integrate with GitHub Actions for deployment.

CHAPTER 5: TROUBLESHOOTING AND OPTIMIZING GITHUB ACTIONS PIPELINES

5.1 Debugging Failed Workflows

Sometimes workflows can fail due to various issues like incorrect configurations or errors in the code. GitHub Actions provides logs that can help you identify what went wrong.

🔹 **Common Reasons for Workflow Failures:**

- Syntax errors in the workflow YAML file.
- Missing dependencies or incorrect environment configurations.
- Permissions issues with accessing external services (e.g., AWS, Docker).

📌 **Example:** Check the logs provided by GitHub to pinpoint the exact step where the error occurred and adjust the workflow accordingly.

📌 **Exercise:** Define **troubleshooting in CI/CD** in your own words and list three common problems you might encounter while setting up a CI/CD pipeline with GitHub Actions.

5.2 Optimizing GitHub Actions Workflows

To ensure your pipelines run efficiently and reliably, consider the following best practices:

🔹 **Best Practices for Optimizing Workflows:**

- **Cache dependencies**: Use caching to **avoid downloading dependencies every time** the pipeline runs.

- **Use reusable workflows**: Define common workflows that can be shared across multiple repositories.
- **Limit parallel jobs**: Avoid running too many jobs in parallel if resource usage is a concern.

📌 **Example:** Caching Node.js dependencies in a workflow:

```
- name: Cache node modules
  uses: actions/cache@v2
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-
```

📌 **Exercise:** Define **caching dependencies** in your own words and list three ways it can help optimize a CI/CD pipeline.

Conclusion

Implementing **CI/CD pipelines with GitHub Actions** can significantly enhance your development process by automating the integration, testing, and deployment stages. GitHub Actions makes it easier to build and manage workflows, while providing flexibility to support various configurations and deployment environments.

LOGGING & MONITORING EXPRESS.JS APIS

CHAPTER 1: INTRODUCTION TO LOGGING & MONITORING

1.1 What is Logging?

Logging is the process of **recording system events, errors, and user activities** in an application. Logs provide insights into:

✓ **Errors & Debugging** – Identify and fix issues quickly.

✓ **Performance Monitoring** – Track slow API responses.

✓ **Security Auditing** – Detect unauthorized access attempts.

1.2 What is Monitoring?

Monitoring is the process of **tracking API performance and usage metrics** to ensure the system is functioning properly.

- **Why Monitor APIs?**

✓ Detect **slow API endpoints**.

✓ Identify **high traffic & bottlenecks**.

✓ Ensure **uptime & reliability**.

- **Difference Between Logging & Monitoring**

Feature	Logging	Monitoring
Purpose	Tracks errors & events	Tracks API performance & uptime
Stored Data	Logs files or databases	Metrics in a dashboard
Usage	Debugging & auditing	Performance analysis

CHAPTER 2: IMPLEMENTING LOGGING IN EXPRESS.JS

2.1 Using console.log() for Basic Logging

📌 Example: Log API Requests in server.js

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

const app = express();


app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
});


app.get("/", (req, res) => {
  res.send("Welcome to Express.js!");
});


app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ Logs **all incoming requests** to the console.

📌 Example Log Output in Console:

```
[2024-02-27T12:00:00Z] GET /
```

2.2 Using morgan for Request Logging

📌 Step 1: Install morgan for HTTP Logging

npm install morgan

📌 Step 2: Configure Morgan Middleware in server.js

```
const morgan = require("morgan");


app.use(morgan("combined")); // Logs in Apache format
```

✓ Logs detailed HTTP request information.

📌 Example Log Output in Console:

```
127.0.0.1 - - [27/Feb/2024:12:00:00 +0000] "GET / HTTP/1.1" 200 -
```

2.3 Writing Logs to a File with winston

📌 Step 1: Install winston for File-Based Logging

npm install winston

📌 Step 2: Configure winston Logger (logger.js)

```
const { createLogger, transports, format } = require("winston");


const logger = createLogger({
  format: format.combine(
    format.timestamp(),
    format.json()
  ),
  transports: [
```

```
    new transports.File({ filename: "logs/error.log", level: "error" }),
    new transports.File({ filename: "logs/combined.log" })
  ]
});

module.exports = logger;
```

✓ Stores logs in logs/error.log and logs/combined.log.

📌 **Step 3: Use Logger in server.js**

```
const logger = require("./logger");

app.use((req, res, next) => {
  logger.info(`Request: ${req.method} ${req.url}`);
  next();
});

app.get("/", (req, res) => {
  logger.info("Home route accessed");
  res.send("Welcome to Express.js!");
});
```

✓ Logs **requests & responses** to files.

📌 **Example Log Entry (logs/combined.log):**

{ "timestamp": "2024-02-27T12:00:00Z", "level": "info", "message": "Request: GET /" }

Chapter 3: Implementing Error Logging

3.1 Handling Errors in Express.js

📌 Example: Logging Errors Using Middleware

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

✓ Logs errors **to the error.log file**.

📌 Example Log Entry (logs/error.log):

{ "timestamp": "2024-02-27T12:00:00Z", "level": "error", "message": "Error: Database connection failed" }

Chapter 4: Implementing API Monitoring

4.1 Using express-status-monitor for Basic Monitoring

📌 Step 1: Install express-status-monitor

```
npm install express-status-monitor
```

📌 Step 2: Configure Express Status Monitor

```
const statusMonitor = require("express-status-monitor");

app.use(statusMonitor());
```

✓ Access **API performance metrics at http://localhost:5000/status**.

📌 **Metrics Available in /status Dashboard:**

✓ **Response time** for each request.

✓ **Memory usage & CPU load**.

✓ **Active WebSocket connections**.

4.2 Using Prometheus for API Monitoring

📌 Step 1: Install prom-client for Metric Collection

```
npm install prom-client
```

📌 Step 2: Configure Prometheus in server.js

```
const client = require("prom-client");
const collectDefaultMetrics = client.collectDefaultMetrics;

// Collect basic system metrics
collectDefaultMetrics({ timeout: 5000 });

const requestCounter = new client.Counter({
  name: "http_requests_total",
  help: "Total number of requests",
  labelNames: ["method", "route", "status"]
});

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

```
  res.on("finish", () => {
    requestCounter.labels(req.method, req.url,
res.statusCode).inc();
  });
  next();
});

// Expose metrics endpoint
app.get("/metrics", async (req, res) => {
  res.set("Content-Type", client.register.contentType);
  res.end(await client.register.metrics());
});
```

✓ Metrics available at **http://localhost:5000/metrics**.

📌 **Example Prometheus Metrics Output:**

```
# HELP http_requests_total Total number of requests
# TYPE http_requests_total counter
http_requests_total{method="GET",route="/",status="200"} 5
```

4.3 Visualizing Metrics with Grafana

1. **Install Grafana** (Download here).
2. **Connect Prometheus as a data source**.
3. **Create dashboards to visualize API performance**.

✓ Monitors **API response times, request counts, and system usage**.

Chapter 5: Best Practices for Logging & Monitoring

- **1. Separate Logs by Type**

✓ Use different files for **requests, errors, and system logs**.

- **2. Use Log Rotation to Manage Large Logs**

✓ Prevents logs from growing indefinitely.

npm install logrotate-stream

- **3. Monitor API Latency & Errors**

✓ Set up alerts for **high response times or frequent errors**.

- **4. Store Logs Securely**

✓ Use **cloud logging services (AWS CloudWatch, Loggly, Papertrail)**.

Case Study: How Uber Uses Logging & Monitoring

Challenges Faced by Uber

✓ Handling **millions of API requests per second**.
✓ Detecting **slow responses & API failures**.

Solutions Implemented

✓ **Used structured logging (Winston) for debugging.**
✓ **Implemented Prometheus for real-time monitoring.**
✓ **Set up alerting on API failures & latency spikes.**

- **Key Takeaways from Uber's Strategy:**

✓ **Logging is crucial for debugging large-scale systems.**

✓ **Monitoring detects performance bottlenecks early.**

✓ **Automated alerts prevent service outages.**

📝 Exercise

- ✅ Set up **Winston for structured logging** in an Express.js app.
- ✅ Implement **error logging middleware** to track API failures.
- ✅ Use **Prometheus & Grafana** to monitor API request metrics.
- ✅ Configure **log rotation & cloud logging for scalability**.

Conclusion

✓ **Logging tracks API activity, debugging errors, and auditing security.**

✓ **Monitoring ensures APIs perform efficiently & detect failures.**

✓ **Using winston, morgan, and express-status-monitor improves observability.**

✓ **Prometheus & Grafana visualize API health & request statistics.**

Final Capstone Project: Develop & deploy a Full-Stack Express.js App with Authentication, WebSockets, and Database Integration.

Final Capstone Project: Full-Stack Express.js App with Authentication, WebSockets, and Database

Step 1: Setting Up the Express.js Backend

1.1 Install Required Dependencies

📌 Create a new Node.js project and install dependencies

mkdir fullstack-app

cd fullstack-app

npm init -y

npm install express mongoose cors dotenv bcryptjs jsonwebtoken socket.io http

✓ express → Web framework for handling API requests.
✓ mongoose → Connects to MongoDB.
✓ cors → Allows frontend-to-backend communication.
✓ dotenv → Stores environment variables.
✓ bcryptjs → Hashes passwords for secure authentication.
✓ jsonwebtoken → Enables JWT-based authentication.
✓ socket.io → Provides real-time WebSocket communication.

Step 2: Setting Up Express Server with WebSockets

📌 Create server.js and set up the server

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

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

```
const http = require("http");
const { Server } = require("socket.io");
const mongoose = require("mongoose");
const cors = require("cors");

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

app.use(cors());
app.use(express.json());

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

// WebSocket Setup
io.on("connection", (socket) => {
  console.log("User connected:", socket.id);
```

```
  socket.on("sendMessage", (data) => {
    io.emit("receiveMessage", data);
  });

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

app.get("/", (req, res) => {
  res.send("Full-Stack App Backend Running");
});

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

✓ **Connects to MongoDB**.
✓ **Handles real-time messaging** with Socket.io.

Step 3: Implementing User Authentication (JWT)

📌 Create models/User.js for User Schema

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

```
const UserSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

module.exports = mongoose.model("User", UserSchema);
```

✓ Defines a **User schema with hashed passwords**.

📌 **Create routes/auth.js for Authentication Routes**

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

const router = express.Router();

// Register User
router.post("/register", async (req, res) => {
  try {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
```

```
    const newUser = new User({ username, email, password: hashedPassword });
    await newUser.save();
    res.json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Login User
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({ message: "Invalid credentials" });
    }

    const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET, { expiresIn: "1h" });
    res.json({ token, user: { username: user.username, email: user.email } });
  } catch (error) {
```

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

module.exports = router;
```

📌 **Modify server.js to Use Authentication Routes**

```
const authRoutes = require("./routes/auth");
app.use("/api/auth", authRoutes);
```

✓ Enables **JWT authentication for user login & registration**.

Step 4: Creating the Frontend with React.js

4.1 Setting Up React

📌 **Create a React App and Install Dependencies**

```
npx create-react-app frontend
cd frontend
npm install socket.io-client axios react-router-dom
```

✓ socket.io-client ⟶ Connects to WebSocket server.
✓ axios ⟶ Fetches data from backend APIs.
✓ react-router-dom ⟶ Enables page navigation.

4.2 Implementing User Login in React

📌 **Modify App.js to Include Login & WebSockets**

```
import { useState, useEffect } from "react";
```

```
import io from "socket.io-client";
import axios from "axios";

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

function App() {
  const [user, setUser] = useState(null);
  const [message, setMessage] = useState("");
  const [chat, setChat] = useState([]);

  useEffect(() => {
    socket.on("receiveMessage", (data) => {
      setChat((prev) => [...prev, data]);
    });

    return () => socket.off("receiveMessage");
  }, []);

  const loginUser = async () => {
    const response = await
axios.post("http://localhost:5000/api/auth/login", {
      email: "test@example.com",
      password: "password123"
```

```
    });
    setUser(response.data.user);
  };

  const sendMessage = () => {
    socket.emit("sendMessage", { username: user.username, text: message });
    setMessage("");
  };

  return (
    <div>
      {!user ? (
        <button onClick={loginUser}>Login</button>
      ) : (
        <>
          <h2>Welcome, {user.username}</h2>
          <input type="text" value={message} onChange={(e) => setMessage(e.target.value)} />
          <button onClick={sendMessage}>Send</button>
          <ul>
            {chat.map((msg, index) => (
              <li key={index}>{msg.username}: {msg.text}</li>
```

```
            ))}
          </ul>
        </>
      )}
    </div>
  );
}

export default App;
```

✓ Logs in a **user & enables real-time chat** using WebSockets.

Step 5: Deploying the Full-Stack App

5.1 Deploy Backend on Heroku

📌 **Push the code to GitHub & deploy to Heroku**

```
git init
git add .
git commit -m "Initial commit"
heroku create fullstack-app
git push heroku main
```

✓ **Backend is now live** on Heroku.

5.2 Deploy Frontend on Vercel

📌 Install Vercel CLI & Deploy

npm install -g vercel

vercel

✓ **Frontend is now live** on Vercel.

Final Project Summary

- **Features Implemented:**

✓ **JWT Authentication** (Login & Registration).
✓ **Real-Time Chat with WebSockets**.
✓ **MongoDB Database Integration**.
✓ **React Frontend with Socket.io Support**.
✓ **Deployed Backend (Heroku) & Frontend (Vercel)**.