Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

AUTHENTICATION, SECURITY & REAL-TIME APPLICATIONS (WEEKS 9-10)

IMPLEMENTING JWT AUTHENTICATION IN NODE.JS

CHAPTER 1: INTRODUCTION TO JWT AUTHENTICATION

1.1 Understanding JWT (JSON Web Token)

JWT (**JSON Web Token**) is a compact and self-contained authentication mechanism used to securely transfer information between parties as a **JSON object**. It is widely used for **user authentication and authorization** in web applications.

Why Use JWT for Authentication?

✓ **Stateless Authentication** – No need to store session data on the server.

✓ **Secure and Compact** – Encodes user information in a token.

✓ **Supports Cross-Origin Requests** – Ideal for APIs and microservices.

✓ **Scalable** – Works efficiently in distributed architectures.

1.2 Structure of a JWT Token

A JWT consists of **three parts,** separated by dots (.):

Header.Payload.Signature

- **Header** – Contains metadata about the algorithm and token type.
- **Payload** – Stores user-specific claims (e.g., user ID, email).
- **Signature** – Ensures the token's integrity using a secret key.

Example of a JWT Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsIm5hbWUiOiJKb2huIERvZSIsImlhdCI6MTYyNTUyNzg5OX0.dfaVbKxLhJ9-WqXdzEzAg9xaPkz6yzKjYV2N1X3D45Q

CHAPTER 2: SETTING UP JWT AUTHENTICATION IN NODE.JS

2.1 Installing Required Packages

To implement JWT authentication in **Node.js and Express,** install the necessary dependencies:

npm init -y

npm install express jsonwebtoken bcryptjs body-parser dotenv cors

Package Overview:

- **express** – Web framework for handling requests.
- **jsonwebtoken** – Used to generate and verify JWTs.
- **bcryptjs** – For hashing passwords securely.
- **body-parser** – Parses incoming request data.

- **dotenv** – Loads environment variables.
- **cors** – Handles cross-origin requests.

2.2 Setting Up Express Server

Create a file server.js and set up an Express server:

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

const express = require('express');

const bodyParser = require('body-parser');


const app = express();

app.use(bodyParser.json());


const PORT = process.env.PORT || 5000;

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

CHAPTER 3: USER AUTHENTICATION WITH JWT

3.1 Creating User Model and Mock Database

For simplicity, we use an **in-memory user database** (without an actual database connection).

Create a file users.js:

```
const users = []; // Temporary storage for users
```

```
module.exports = users;
```

3.2 Implementing User Registration with Password Hashing

Create a file auth.js for authentication logic:

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

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const users = require('./users');


const router = express.Router();

const SECRET_KEY = "mysecretkey"; // Replace with environment variable in production


// Register a new user

router.post('/register', async (req, res) => {

  const { name, email, password } = req.body;


  // Check if the user already exists

  const existingUser = users.find(user => user.email === email);

  if (existingUser) {
```

```
    return res.status(400).json({ message: "User already exists" });
  }

  // Hash the password
  const hashedPassword = await bcrypt.hash(password, 10);

  // Create a new user
  const newUser = { id: users.length + 1, name, email, password: hashedPassword };
  users.push(newUser);

  res.status(201).json({ message: "User registered successfully" });
});

module.exports = router;
```

Explanation:

- **bcrypt.hash(password, 10)** – Hashes the password securely before storing it.
- **Saves users in an array (users.js)** – In real applications, this should be stored in a **database**.

3.3 Implementing User Login and Token Generation

Add login functionality to auth.js:

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

  // Find user by email
  const user = users.find(user => user.email === email);
  if (!user) {
    return res.status(400).json({ message: "Invalid credentials" });
  }

  // Compare passwords
  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) {
    return res.status(400).json({ message: "Invalid credentials" });
  }

  // Generate JWT token
  const token = jwt.sign({ userId: user.id, email: user.email },
SECRET_KEY, { expiresIn: '1h' });
```

```
  res.json({ message: "Login successful", token });
});
```

Explanation:

✓ **Checks if the user exists** in the database.

✓ **Verifies password using bcrypt.compare()**.

✓ **Generates a JWT token valid for 1 hour**.

CHAPTER 4: PROTECTING ROUTES USING JWT MIDDLEWARE

4.1 Creating Authentication Middleware

To secure routes, create authMiddleware.js:

```
const jwt = require('jsonwebtoken');
const SECRET_KEY = "mysecretkey";

module.exports = function (req, res, next) {
  const token = req.header('Authorization');
  if (!token) return res.status(401).json({ message: "Access Denied" });

  try {
    const verified = jwt.verify(token.replace("Bearer ", ""), SECRET_KEY);
    req.user = verified;
    next();
```

```
  } catch (error) {
    res.status(400).json({ message: "Invalid Token" });
  }
};
```

4.2 Protecting Routes with Middleware

Modify server.js to include protected routes:

```
const authMiddleware = require('./authMiddleware');

app.get('/dashboard', authMiddleware, (req, res) => {
  res.json({ message: `Welcome User ${req.user.userId}` });
});
```

Now, only authenticated users can access /dashboard.

Case Study: How Netflix Uses JWT for Secure User Authentication

Background

Netflix, a global video streaming platform, requires a **secure authentication system** to allow users to access personalized content.

Challenges

- **Preventing unauthorized access** to user accounts.
- **Ensuring secure session management** without storing sessions on servers.

- **Scalability** to support millions of users simultaneously.

Solution: Implementing JWT for Authentication

✓ Used **JWT tokens** for **stateless authentication,** eliminating session storage overhead.
✓ Implemented **role-based access control** using JWT claims.
✓ Improved **user experience** by allowing seamless logins across devices.

By leveraging JWT, **Netflix ensures fast, scalable, and secure authentication** for its millions of users.

Exercise

1. Modify the JWT token expiration time from **1 hour to 30 minutes** in auth.js.
2. Implement a new **protected route** /profile that only logged-in users can access.
3. Extend the JWT payload to include **user role (admin, user),** and restrict an **admin-only route**.

Conclusion

In this section, we explored:
✓ **What JWT is and how it works for authentication**.
✓ **Setting up JWT authentication in a Node.js and Express application**.
✓ **Hashing passwords securely using bcrypt.js**.
✓ **Protecting routes using authentication middleware**.

✓ A real-world case study on how Netflix uses JWT for secure authentication.

HASHING PASSWORDS WITH BCRYPT.JS

CHAPTER 1: INTRODUCTION TO PASSWORD HASHING

1.1 Understanding Password Security

In modern web applications, **password security** is one of the most critical aspects of **user authentication**. Storing passwords in **plain text** is a major security risk, as it exposes sensitive data in case of a **database breach**.

Instead of storing passwords directly, developers use **hashing** to convert passwords into an **irreversible, encrypted format**.

✓ **Hashing** – Converts a password into a fixed-length string using a cryptographic algorithm.
✓ **Salting** – Adds a random string (salt) to the password before hashing to make attacks harder.
✓ **Hash Verification** – Ensures that only valid passwords match the stored hash.

A strong hashing algorithm prevents **brute-force attacks** and ensures **password confidentiality**.

1.2 Why Use bcrypt.js?

bcrypt.js is a popular library for password hashing in Node.js because it offers:

- **Adaptive hashing** – Increases security by making hashing computationally expensive.
- **Built-in salting** – Prevents **rainbow table attacks** by adding random values.

- **Easy integration** – Works seamlessly with Node.js authentication systems.

To install bcrypt.js, run:

npm install bcryptjs

Then, require it in your project:

const bcrypt = require('bcryptjs');

CHAPTER 2: HASHING PASSWORDS USING BCRYPT.JS

2.1 How Hashing Works in bcrypt.js

bcrypt follows a **multi-step process**:

1. Generate a **salt** (random value).
2. Hash the password with the salt using the **bcrypt algorithm**.
3. Store the hashed password securely in the database.

2.2 Example: Hashing a Password

```
const bcrypt = require('bcryptjs');


const password = "mySecurePassword";


bcrypt.genSalt(10, (err, salt) => {
  bcrypt.hash(password, salt, (err, hash) => {
    if (err) throw err;
```

```
    console.log("Hashed Password:", hash);
  });
});
```

✓ genSalt(10) generates a **salt with 10 rounds of encryption**.
✓ hash(password, salt) encrypts the password using bcrypt.
✓ The final hashed password is stored in the database.

2.3 Synchronous vs. Asynchronous Hashing

Asynchronous Hashing (Recommended for Performance):

```
bcrypt.genSalt(10)
  .then(salt => bcrypt.hash(password, salt))
  .then(hash => console.log("Hashed Password:", hash))
  .catch(err => console.error(err));
```

Synchronous Hashing (Blocks Execution, Use with Caution):

```
const salt = bcrypt.genSaltSync(10);
const hash = bcrypt.hashSync(password, salt);
console.log("Hashed Password:", hash);
```

✓ Asynchronous methods prevent **blocking other operations**, making them ideal for production.
✓ Synchronous methods should only be used in **scripts or testing**.

CHAPTER 3: VERIFYING PASSWORDS WITH BCRYPT.JS

3.1 How Password Verification Works

When a user logs in:

1. The provided **plain-text password** is hashed using the same algorithm.
2. The new hash is **compared** with the stored hash.
3. If both hashes **match**, authentication is successful.

3.2 Example: Comparing a Plain Password with a Hashed Password

```
const storedHash = "$2a$10$Xfz9ZuvHoJlfCudVnPM1P.qxxnp.J9XSkjoRkVZhEK5qgYOFXyPha"; // Example hash


bcrypt.compare("mySecurePassword", storedHash, (err, result) => {
  if (result) {
    console.log("Password Matched! User Authenticated.");
  } else {
    console.log("Invalid Password.");
  }
});
```

✓ compare(plainPassword, storedHash) checks if the password matches the stored hash.
✓ The function returns true if the password is **valid** and false if **incorrect**.

3.3 Synchronous vs. Asynchronous Comparison

Asynchronous Password Comparison (Recommended):

```
bcrypt.compare("wrongPassword", storedHash)
  .then(result => console.log(result ? "Login Successful" : "Invalid Password"))
  .catch(err => console.error(err));
```

Synchronous Password Comparison:

```
const isMatch = bcrypt.compareSync("mySecurePassword", storedHash);
console.log(isMatch ? "Login Successful" : "Invalid Password");
```

✓ **Asynchronous comparison** is better for **handling multiple user requests** efficiently.

CHAPTER 4: SALTING IN BCRYPT.JS FOR ENHANCED SECURITY

4.1 What is Salting?

A **salt** is a **random string** added to the password before hashing. It makes it difficult for attackers to use **precomputed attacks** such as **rainbow tables**.

Without salting, two identical passwords will have the **same hash**:

```
const hash1 = bcrypt.hashSync("password123", 10);
const hash2 = bcrypt.hashSync("password123", 10);
console.log(hash1 === hash2); // Output: false (because salt is different)
```

✓ Each hash is **unique**, even for the same input password.

4.2 Choosing the Right Salt Rounds

The **higher the salt rounds**, the **stronger** the encryption but the **slower** the process.

- **8-10 rounds** → Suitable for most applications.
- **12+ rounds** → Recommended for highly sensitive data.

Example of manually defining **salt rounds**:

```
const saltRounds = 12;

bcrypt.hash("myPassword", saltRounds, (err, hash) => {
  console.log(hash);
});
```

✓ **Higher salt rounds** increase security but may **slow down authentication**.

Case Study: How a FinTech Company Strengthened User Security with bcrypt.js

Background

A **FinTech startup** handling **online transactions** needed to secure user accounts against **brute-force attacks** and **database breaches**.

Challenges

✓ Users often reused **weak passwords** across different services.
✓ A previous **security breach** exposed customer credentials.
✓ Their system used **plain-text passwords**, making them vulnerable.

Solution: Implementing bcrypt.js for Password Hashing

The development team adopted bcrypt.js to:

✓ **Hash all user passwords** before storing them in the database.
✓ **Use 12 salt rounds** to strengthen encryption.
✓ **Implement secure login verification** using bcrypt's compare method.

Results

- **100% prevention** of password leaks in future breaches.
- **Stronger encryption,** reducing brute-force risks.
- **Increased user trust,** boosting customer sign-ups.

By using bcrypt.js, the company **hardened authentication security** and improved compliance with **security best practices**.

Exercise

1. Write a Node.js script that:
    - Takes a password as input.
    - Hashes it using bcrypt.js with **10 salt rounds**.
    - Prints the hashed password.
2. Modify the script to:
    - Accept a **user input password**.
    - Compare it with a **stored hash**.
    - Print "Login Successful" if it matches, otherwise print "Invalid Password".

Conclusion

In this section, we explored:

✓ **How bcrypt.js securely hashes passwords to prevent leaks.**

✓ **How to implement password verification in user authentication.**

✓ **The importance of salting to protect against attacks.**

ROLE-BASED ACCESS CONTROL (RBAC) IN NODE.JS

CHAPTER 1: INTRODUCTION TO ROLE-BASED ACCESS CONTROL (RBAC)

1.1 Understanding Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is a security mechanism that restricts system access based on predefined user roles. Instead of assigning permissions directly to users, RBAC **groups permissions into roles**, making it easier to manage access levels.

RBAC is widely used in **web applications, enterprise software, and APIs** to control user privileges effectively.

Key Features of RBAC:

✓ **User Roles** – Assigning users predefined roles (Admin, Editor, User, etc.).
✓ **Permissions** – Defining what actions each role can perform.
✓ **Access Control** – Restricting endpoints based on user roles.

Example Use Case:

- **Admin** → Can create, update, and delete users.
- **Editor** → Can modify content but cannot manage users.
- **User** → Can view content but cannot modify it.

RBAC improves **security, scalability, and maintainability** in applications by **simplifying permission management**.

CHAPTER 2: SETTING UP ROLE-BASED ACCESS CONTROL IN NODE.JS

2.1 Installing Required Dependencies

To implement **RBAC in Node.js,** we need:

- **Express.js** – To create API routes.
- **JSON Web Tokens (JWT)** – For authentication and role verification.
- **bcrypt.js** – To hash user passwords securely.
- **dotenv** – To store environment variables.
- **mongoose** – To manage user roles in MongoDB.

Install the required packages:

```
npm install express mongoose jsonwebtoken bcryptjs dotenv
```

2.2 Connecting to MongoDB

Create a **MongoDB connection** file **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;
```

✓ The .env file stores the **MongoDB URI**:

```
MONGO_URI=mongodb://localhost:27017/rbacDB

JWT_SECRET=mysecretkey
```

CHAPTER 3: DEFINING USER ROLES AND PERMISSIONS

3.1 Creating a User Schema with Roles

Define a **User schema** in **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 },
   role: { type: String, enum: ['admin', 'editor', 'user'], default: 'user' }
});


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

✓ Each user is assigned a **role** (admin, editor, or user).
✓ The default role is **"user"** for new registrations.

CHAPTER 4: IMPLEMENTING ROLE-BASED AUTHENTICATION

4.1 User Registration with Hashed Passwords

Create a **user registration route** in **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();

// Register a new user
router.post('/register', async (req, res) => {
  try {
    const { name, email, password, role } = req.body;

    // Hash password
    const hashedPassword = await bcrypt.hash(password, 10);
```

```
    const newUser = new User({ name, email, password: hashedPassword, role });
    await newUser.save();

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

module.exports = router;
```

✓ **Passwords are securely hashed** using bcryptjs.
✓ **Users can specify their role** (if allowed).

4.2 User Login & Token Generation

Modify routes/auth.js to include **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, role: user.role }, process.env.JWT_SECRET, { expiresIn: '1h' });

    res.json({ message: 'Login successful', token });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ **JWT tokens** store the user ID and role.
✓ Tokens expire after **1 hour** for security.

CHAPTER 5: PROTECTING ROUTES WITH ROLE-BASED ACCESS CONTROL

5.1 Middleware for Authentication and Role Checking

Create **middleware/authMiddleware.js**:

```
const jwt = require('jsonwebtoken');

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

```
exports.authenticate = (req, res, next) => {
  const token = req.header('Authorization');
  if (!token) return res.status(403).json({ error: 'Access Denied' });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // Attach user data to request
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid Token' });
  }
};

exports.authorize = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ error: 'Unauthorized Access' });
    }
    next();
  };
};
```

✓ **authenticate** ensures only logged-in users can access certain routes.
✓ **authorize** restricts actions to specific roles (e.g., authorize('admin')).

5.2 Protecting API Endpoints

Create **routes/protectedRoutes.js**:

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

const { authenticate, authorize } = require('../middleware/authMiddleware');


const router = express.Router();


// Public Route

router.get('/public', (req, res) => res.json({ message: "Accessible by anyone" }));


// Protected Route (Only Authenticated Users)

router.get('/user', authenticate, (req, res) => res.json({ message: "Welcome, user!" }));


// Admin Only Route

router.get('/admin', authenticate, authorize('admin'), (req, res) => res.json({ message: "Admin access granted" }));
```

```
module.exports = router;
```

✓ **/public** – Open to all users.
✓ **/user** – Requires authentication.
✓ **/admin** – Requires **admin** privileges.

Case Study: How an E-Learning Platform Implemented RBAC for Course Management

Background

An online learning platform needed **role-based access control** for different user types:

✓ **Admins** → Manage users and courses.
✓ **Instructors** → Create and update courses.
✓ **Students** → Enroll and complete courses.

Challenges

- **Unauthorized course modifications** by students.
- **Inconsistent role assignments** leading to security risks.
- **Difficulty in scaling permissions** as new features were added.

Solution: Implementing RBAC with Mongoose & JWT

The development team:

✓ Defined **roles and permissions** for Admins, Instructors, and Students.
✓ Used **JWT for authentication** and role-based access control.
✓ Created **middleware to restrict access** to API endpoints.

Results

- **Improved security,** preventing unauthorized course modifications.
- **Better user management,** reducing admin overhead.
- **Scalability,** allowing new roles to be added easily.

This case study demonstrates how **RBAC enhances security and access management** in web applications.

Exercise

1. Modify the **User model** to include an additional role: "moderator".
2. Add an API route /editor that only **editors and admins** can access.
3. Implement role-based access control for **editing and deleting users**.

Conclusion

✓ We implemented **role-based authentication** using **Mongoose & JWT**.
✓ We created **protected routes** to restrict access based on user roles.
✓ We demonstrated how **RBAC improves security** in web applications.

USING SOCKET.IO FOR REAL-TIME COMMUNICATION

CHAPTER 1: INTRODUCTION TO REAL-TIME COMMUNICATION WITH SOCKET.IO

1.1 What is Socket.io?

Socket.io is a **JavaScript library** that enables **real-time, bidirectional, and event-driven communication** between clients and servers. Unlike traditional HTTP requests, which follow a **request-response cycle**, Socket.io uses **WebSockets** to establish persistent connections, allowing data to be exchanged instantly between users and servers.

Key Features of Socket.io

- **Full-duplex communication** – Data flows both ways simultaneously.
- **Low latency** – Messages are transmitted instantly without waiting for a response.
- **Event-driven model** – Supports custom events for efficient messaging.
- **Automatic reconnection** – Handles disconnections and retries automatically.
- **Works in browsers and Node.js** – Can be used for both frontend and backend applications.

1.2 How Does Socket.io Work?

Socket.io operates using **WebSockets,** but it also provides fallback mechanisms like **long polling** for older browsers. The communication process follows these steps:

1. **Client connects to the server** via WebSockets.
2. **Server acknowledges the connection** and establishes a two-way link.
3. **Both client and server exchange messages** in real time.
4. **The connection stays open** unless manually closed or interrupted.

Comparison: HTTP vs. WebSockets (Socket.io)

Feature	**HTTP Requests**	**WebSockets (Socket.io)**
Communication	One-way (Client to Server)	Bidirectional
Latency	Higher due to request-response	Lower with real-time updates
Use Case	Static pages, APIs	Chat apps, notifications, live updates
Connection Persistence	Closes after response	Stays open

CHAPTER 2: SETTING UP SOCKET.IO IN A NODE.JS PROJECT

2.1 Installing Required Packages

Before using **Socket.io**, ensure you have **Node.js** installed. Then, create a new project:

mkdir socketio-chat

cd socketio-chat

npm init -y

Install **Express.js** (for creating a server) and **Socket.io**:

npm install express socket.io

2.2 Setting Up a Basic Express Server with Socket.io

Create a new file server.js and add the following code:

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

const http = require('http');

const { Server } = require('socket.io');


const app = express();

const server = http.createServer(app);

const io = new Server(server);


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

   res.send('Socket.io Server Running');

});
```

```
io.on('connection', (socket) => {
  console.log('A user connected');

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

server.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Explanation:

✓ **Creates an Express server** using http.createServer().
✓ **Initializes a Socket.io server** with new Server(server).
✓ **Listens for client connections and disconnections** using .on('connection', callback).
✓ **Starts the server on port 3000**.

CHAPTER 3: IMPLEMENTING REAL-TIME MESSAGING WITH SOCKET.IO

3.1 Creating a Basic Chat Application

Now, let's modify our server to **broadcast chat messages** to all connected users.

Update server.js with Chat Functionality

```
io.on('connection', (socket) => {
  console.log('A user connected');

  socket.on('chat message', (msg) => {
    console.log('Message received:', msg);
    io.emit('chat message', msg); // Broadcast message to all users
  });

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

3.2 Setting Up the Client (Frontend)

Create an **HTML file (index.html)** to send and display chat messages:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Chat App</title>
```

```
  <script src="https://cdn.socket.io/4.0.1/socket.io.min.js"></script>
</head>
<body>
  <h1>Real-Time Chat</h1>
  <input id="message" type="text" placeholder="Type a message">
  <button onclick="sendMessage()">Send</button>
  <ul id="messages"></ul>

  <script>
    const socket = io();

    function sendMessage() {
      const msg = document.getElementById('message').value;
      socket.emit('chat message', msg);
    }

    socket.on('chat message', (msg) => {
      const list = document.getElementById('messages');
      const item = document.createElement('li');
      item.textContent = msg;
      list.appendChild(item);
```

```
    });
  </script>
</body>
</html>
```

How It Works:

✓ **Clients send messages using socket.emit('chat message', msg).**
✓ **Server listens for the event and broadcasts the message to all users.**
✓ **Messages appear instantly in all connected clients.**

CHAPTER 4: ADVANCED FEATURES OF SOCKET.IO

4.1 Broadcasting Events to Specific Users

If you want to send a message to **only a specific client,** use **socket.emit()** instead of broadcasting with io.emit().

Example: Sending Private Messages

```
socket.on('private message', (data) => {
  const { recipientId, message } = data;
  io.to(recipientId).emit('private message', message);
});
```

- **recipientId** is the unique socket ID of the recipient.
- This enables **one-on-one messaging** instead of broadcasting to all users.

4.2 Handling User Disconnections and Reconnections

Socket.io automatically **reconnects users** if the connection is lost. However, we can also **handle disconnections manually**:

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

Case Study: How WhatsApp Uses WebSockets for Real-Time Messaging

Background

WhatsApp, a globally used messaging platform, requires **instant message delivery** with minimal delays.

Challenges

- Millions of concurrent users sending messages.
- Ensuring messages reach recipients even in **low-network conditions**.
- Handling **user presence and real-time status updates**.

Solution: Using WebSockets for Instant Messaging

✓ **Persistent WebSocket connections** – No need for repeated HTTP requests.
✓ **End-to-end encryption** – Secure message delivery.
✓ **Message queues** – Store messages if a user is offline and deliver them later.

Results

- **99.99% message delivery success rate**.

- **Reduced network traffic,** improving efficiency.
- **Instant read receipts and live typing indicators**.

This case study highlights how **Socket.io and WebSockets power real-time chat applications**.

Exercise

1. What is the main advantage of WebSockets over HTTP requests?
2. Write a simple Express server that sends a real-time notification to all connected users.
3. How does Socket.io handle automatic reconnection when a user disconnects?

Conclusion

In this section, we explored:

✓ **How Socket.io enables real-time communication**.

✓ **How to build a real-time chat app using Node.js and WebSockets**.

✓ **Advanced Socket.io features like private messaging and event broadcasting**.

IMPLEMENTING A REAL-TIME CHAT APPLICATION IN NODE.JS

CHAPTER 1: INTRODUCTION TO REAL-TIME CHAT APPLICATIONS

1.1 Understanding Real-Time Communication

Real-time communication refers to **instant data exchange** between clients and servers without requiring the client to refresh the page. This technology is essential for applications such as **messaging apps, live notifications, and online gaming**.

Traditional HTTP-based applications rely on **request-response cycles**, where a client requests data and the server responds. However, real-time applications require:

✓ **Persistent Connections** – Clients stay connected to the server.
✓ **Bidirectional Communication** – Both server and client can send/receive messages at any time.
✓ **Low Latency** – Messages are delivered with minimal delay.

1.2 Why Use WebSockets for Real-Time Chat?

WebSockets enable **full-duplex communication**, meaning data flows in both directions continuously without repeated HTTP requests.

Feature	WebSockets	Traditional HTTP
Latency	Very Low	High
Data Flow	Bidirectional	Request-Response
Connection	Persistent	Short-lived

Ideal For	Chats, live updates	Static content

By using **Socket.IO**, a WebSocket library for Node.js, we can easily implement real-time messaging in our chat application.

CHAPTER 2: SETTING UP THE CHAT APPLICATION

2.1 Installing Required Packages

To build a real-time chat application, install the required dependencies:

npm init -y

npm install express socket.io http cors

Package Overview:

- **express** – Creates a backend server.
- **socket.io** – Enables WebSocket communication.
- **http** – Required to integrate Express with WebSockets.
- **cors** – Allows cross-origin requests from different clients.

2.2 Setting Up an Express Server with WebSockets

Create a file server.js and set up a WebSocket server:

const express = require('express');

const http = require('http');

const socketIo = require('socket.io');

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


const app = express();
const server = http.createServer(app);
const io = socketIo(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"]
  }
});

app.use(cors());

app.get('/', (req, res) => {
  res.send('Chat server is running...');
});

io.on('connection', (socket) => {
  console.log('A user connected:', socket.id);

  socket.on('message', (data) => {
```

```
    io.emit('message', data); // Broadcast message to all clients
  });

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

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

2.3 Explanation of Code

✓ Creates an Express server (server.js).
✓ Integrates Socket.IO for WebSocket communication.
✓ Listens for client connections (io.on('connection', callback)).
✓ Handles incoming messages (socket.on('message', callback)).
✓ Broadcasts messages to all connected clients using io.emit().
✓ Handles user disconnections (socket.on('disconnect', callback)).

CHAPTER 3: CREATING A FRONTEND CHAT CLIENT

3.1 Setting Up a Basic HTML Chat Interface

Create a file index.html and add the following:

```
<!DOCTYPE html>
<html>
<head>
  <title>Chat Application</title>
  <script src="https://cdn.socket.io/4.0.0/socket.io.min.js"></script>
</head>
<body>
  <h2>Real-Time Chat</h2>
  <div id="chat-box"></div>
  <input type="text" id="message" placeholder="Type a message..."
/>
  <button onclick="sendMessage()">Send</button>

  <script>
    const socket = io("http://localhost:3000");

    socket.on("message", (data) => {
      const chatBox = document.getElementById("chat-box");
      const messageElement = document.createElement("p");
      messageElement.textContent = data;
```

```
        chatBox.appendChild(messageElement);
    });


    function sendMessage() {
        const message =
document.getElementById("message").value;
        socket.emit("message", message);
        document.getElementById("message").value = "";
    }
  </script>
</body>
</html>
```

3.2 Explanation of the Client-Side Code

✓ Connects to WebSocket server (io("http://localhost:3000")).
✓ Listens for incoming messages (socket.on("message", callback)).
✓ Displays received messages inside the chat box.
✓ Sends new messages to the server (socket.emit("message", message)).

CHAPTER 4: ENHANCING THE CHAT APPLICATION

4.1 Displaying Usernames in Messages

Modify the server.js to include usernames:

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

  socket.on('message', (data) => {
    io.emit('message', `${socket.id}: ${data}`); // Add username to message
  });

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

4.2 Sending Private Messages

Modify server.js to support private messages:

```
socket.on('privateMessage', ({ recipientId, message }) => {
  io.to(recipientId).emit('message', `(Private) ${socket.id}: ${message}`);
});
```

✓ Now, users can send private messages to specific users.

Case Study: How WhatsApp Implements Real-Time Chat with WebSockets

Background

WhatsApp, one of the most widely used messaging platforms, requires **real-time message delivery** with minimal latency.

Challenges

- **Ensuring instant message delivery to millions of users.**
- **Maintaining stable WebSocket connections over long periods.**
- **Handling high message loads efficiently.**

Solution: Using WebSockets for Real-Time Messaging

✓ Implemented **WebSocket-based messaging for low-latency communication**.
✓ Used **message queues** to store and forward messages when a recipient is offline.
✓ Optimized **server performance to handle millions of active users** simultaneously.

This approach allows WhatsApp to **maintain real-time, efficient, and scalable chat services** globally.

Exercise

1. Modify the chat application to **allow users to enter a username before sending messages**.
2. Implement a feature where **messages are displayed with timestamps**.
3. Extend the chat application to support **multiple chat rooms**.

Conclusion

In this section, we explored:

✓ **How WebSockets enable real-time chat applications.**

✓ **Setting up a Node.js server with Socket.IO for bidirectional communication.**

✓ **Creating a frontend chat client that interacts with the WebSocket server.**

✓ **Enhancing the chat app with usernames and private messaging.**

✓ **A case study on how WhatsApp implements real-time chat.**

HANDLING WEBSOCKET EVENTS IN NODE.JS

CHAPTER 1: INTRODUCTION TO WEBSOCKETS

1.1 Understanding WebSockets and Real-Time Communication

WebSockets are a communication protocol that enables **full-duplex, real-time** communication between a client and a server over a **single, long-lived connection**. Unlike traditional **HTTP requests**, which follow a **request-response** model, WebSockets allow **continuous, bidirectional communication**, making them ideal for **real-time applications** such as:

✓ **Live chat applications** (e.g., WhatsApp Web, Slack).
✓ **Stock market updates** (real-time price tracking).
✓ **Online gaming** (multiplayer interactions).
✓ **Collaborative document editing** (Google Docs-like applications).

1.2 How WebSockets Differ from HTTP

Feature	HTTP	WebSockets
Connection Type	Short-lived (request-response)	Persistent (long-lived)
Communication	One-way (client requests, server responds)	Bidirectional (both client & server can send messages anytime)
Latency	Higher due to repeated requests	Lower due to continuous connection

Best For	Static pages, APIs, file downloads	Live chats, notifications, real-time data

To implement WebSockets in **Node.js**, we use the **ws (WebSocket) library**. Install it using:

npm install ws

Then, import it in your project:

const WebSocket = require('ws');

CHAPTER 2: SETTING UP A WEBSOCKET SERVER IN NODE.JS

2.1 Creating a WebSocket Server

A WebSocket server listens for incoming client connections and facilitates real-time communication.

Example: Basic WebSocket Server in Node.js

```
const WebSocket = require('ws');


const server = new WebSocket.Server({ port: 8080 });


server.on('connection', (socket) => {
  console.log('Client connected');


  socket.send('Welcome to the WebSocket server!');
```

```
  socket.on('message', (message) => {
    console.log('Received:', message);
    socket.send(`You said: ${message}`);
  });

  socket.on('close', () => {
    console.log('Client disconnected');
  });
});

console.log('WebSocket server running on ws://localhost:8080');
```

✓ **new WebSocket.Server({ port: 8080 })** creates a WebSocket server on port **8080**.
✓ **.on('connection', callback)** handles new client connections.
✓ **.send()** sends a message to the client.
✓ **.on('message', callback)** listens for incoming messages.
✓ **.on('close', callback)** detects when a client disconnects.

CHAPTER 3: CONNECTING A WEBSOCKET CLIENT

3.1 Creating a WebSocket Client in JavaScript

WebSocket clients can be implemented in **web browsers** using the built-in WebSocket API.

Example: WebSocket Client in a Web Page

```
<!DOCTYPE html>
<html>
<head>
  <title>WebSocket Client</title>
</head>
<body>
  <h2>WebSocket Client</h2>
  <button onclick="sendMessage()">Send Message</button>
  <script>
    const socket = new WebSocket('ws://localhost:8080');

    socket.onopen = () => {
      console.log("Connected to WebSocket server");
    };

    socket.onmessage = (event) => {
      console.log("Message from server:", event.data);
    };

    socket.onclose = () => {
      console.log("Connection closed");
```

```
    };

    function sendMessage() {
        socket.send("Hello from the client!");
    }
  </script>
</body>
</html>
```

✓ The **WebSocket client** connects to the **WebSocket server**.
✓ When a message is received from the server, it is **logged to the console**.
✓ Clicking the **"Send Message"** button sends a **message to the server**.

CHAPTER 4: HANDLING WEBSOCKET EVENTS IN NODE.JS

4.1 Understanding WebSocket Events

WebSockets use several key **events** to handle communication:

Event	Description
connection	Fired when a new client connects to the WebSocket server.
message	Triggered when the client or server sends a message.
close	Fired when the client disconnects.

error	Triggered if an error occurs in the connection.

4.2 Handling Messages and Errors

Example: Handling Messages and Errors in WebSocket Server

```
const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 8080 });

server.on('connection', (socket) => {
  console.log('Client connected');

  socket.on('message', (message) => {
    console.log('Received:', message);
    socket.send(`Echo: ${message}`);
  });

  socket.on('error', (err) => {
    console.error('WebSocket error:', err);
  });

  socket.on('close', () => {
    console.log('Client disconnected');
```

```
  });
});
```

✓ The **.on('message', callback)** event processes **incoming messages**.
✓ The **.on('error', callback)** event handles **connection errors**.

CHAPTER 5: BROADCASTING MESSAGES TO MULTIPLE CLIENTS

5.1 Sending Messages to All Connected Clients

In many applications, such as **chat rooms**, messages must be **broadcasted** to all connected clients.

Example: Broadcasting Messages to All Clients

```
server.on('connection', (socket) => {
  console.log('Client connected');

  socket.on('message', (message) => {
    console.log('Received:', message);

    server.clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(`Broadcast: ${message}`);
      }
    });
```

```
  });
});
```

✓ server.clients.forEach() loops through **all connected clients**.
✓ Only clients with **readyState === WebSocket.OPEN** receive messages.

Case Study: How a Stock Trading Platform Used WebSockets for Real-Time Updates

Background

A stock trading platform needed to **deliver real-time stock price updates** to thousands of users. Using **traditional polling (repeated HTTP requests)** caused:

✓ **High server load** due to frequent requests.
✓ **Delayed stock prices,** reducing user experience.
✓ **Inefficient bandwidth usage,** increasing operational costs.

Solution: Implementing WebSockets

The development team implemented **WebSockets for live stock updates,** allowing the server to:

✓ **Push real-time stock prices** to connected clients instantly.
✓ **Reduce server load** by maintaining **persistent connections**.
✓ **Optimize bandwidth** by only sending data **when prices changed**.

Results

- **Stock prices updated in real-time,** improving user experience.
- **80% reduction in server load,** leading to cost savings.

- **Scalable architecture,** handling thousands of concurrent users.

This case study highlights how **WebSockets improve performance and scalability** in real-time applications.

Exercise

1. Create a **WebSocket server** that sends a "Hello, Client!" message when a client connects.
2. Modify the server to handle **incoming messages** and echo them back.
3. Implement **broadcasting,** so all connected clients receive a message when one client sends a message.

Conclusion

In this section, we explored:

✓ **How WebSockets enable real-time communication**.

✓ **How to set up a WebSocket server in Node.js**.

✓ **How to handle WebSocket events, including connection, messaging, and errors**.

✓ **How to broadcast messages to multiple clients**.

ASSIGNMENT:

BUILD A REAL-TIME NOTIFICATION SYSTEM USING WEBSOCKETS

SOLUTION GUIDE: BUILD A REAL-TIME NOTIFICATION SYSTEM USING WEBSOCKETS IN NODE.JS

Step 1: Set Up the Project

1.1 Create a New Project Directory

mkdir real-time-notifications

cd real-time-notifications

1.2 Initialize a Node.js Project

npm init -y

This generates a package.json file.

1.3 Install Dependencies

npm install express socket.io cors dotenv

✓ **Express** – Web framework for handling HTTP requests.
✓ **Socket.io** – WebSocket library for real-time communication.
✓ **CORS** – Allows cross-origin requests from clients.
✓ **dotenv** – Manages environment variables.

Step 2: Set Up the Express and WebSocket Server

2.1 Create a Basic Express Server with Socket.io

1. Create **server.js**:

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

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

```
const { Server } = require('socket.io');
const cors = require('cors');
require('dotenv').config();

const app = express();
const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: '*',
    methods: ['GET', 'POST']
  }
});

const PORT = process.env.PORT || 5000;

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

app.get('/', (req, res) => {
  res.send('Real-Time Notification System is Running!');
});
```

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

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

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

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

✓ **Socket.io** listens for client connections.
✓ **Users send notifications** using 'send-notification'.
✓ **Broadcast notifications** to all clients using 'receive-notification'.
✓ Handles **user disconnections** gracefully.

Step 3: Create the Frontend Client (HTML + JavaScript)

3.1 Simple Web Client for Receiving Notifications

Create **index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Real-Time Notifications</title>
  <script src="https://cdn.socket.io/4.0.1/socket.io.min.js"></script>
</head>
<body>
  <h2>Real-Time Notification System</h2>
  <button onclick="sendNotification()">Send Notification</button>
  <ul id="notifications"></ul>

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

    socket.on('connect', () => {
      console.log('Connected to WebSocket server');
    });
```

```
    socket.on('receive-notification', (data) => {
      console.log('New Notification:', data);
      const li = document.createElement('li');
      li.textContent = data.message;
      document.getElementById('notifications').appendChild(li);
    });

    function sendNotification() {
      const notification = { message: `New alert at ${new Date().toLocaleTimeString()}` };
      socket.emit('send-notification', notification);
    }
  </script>
</body>
</html>
```

✓ **Clients receive notifications in real-time** using receive-notification.
✓ **Users can send notifications** using the sendNotification() function.
✓ **New notifications appear instantly** on the webpage.

Step 4: Testing the Notification System

4.1 Start the WebSocket Server

node server.js

✓ The server will run on http://localhost:5000.

4.2 Open Multiple Browser Windows

- Open **index.html** in multiple tabs.
- Click the "Send Notification" button in one tab.
- The notification should appear **in all open tabs** instantly.

Step 5: Enhancing the Notification System

5.1 Save Notifications in a Database (MongoDB)

1. Install Mongoose:
2. npm install mongoose
3. Connect to MongoDB in server.js:
4. const mongoose = require('mongoose');
5.
6. mongoose.connect('mongodb://localhost:27017/notificationsDB', {
7. useNewUrlParser: true,
8. useUnifiedTopology: true
9. })
10. .then(() => console.log('MongoDB Connected'))
11. .catch(err => console.error('MongoDB Connection Error:', err));
12. Create **Notification Schema**:

```
13. const Notification = mongoose.model('Notification', new
    mongoose.Schema({
14.         message: String,
15.   timestamp: { type: Date, default: Date.now }
16.       }));
17. Save notifications when received:
18.       socket.on('send-notification', async (data) => {
19.         console.log('New Notification:', data);
20.         await Notification.create(data);
21.         io.emit('receive-notification', data);
22.       });
```

✓ **Notifications are stored in MongoDB**, ensuring persistence.
✓ Users receive **both real-time and saved notifications**.

Case Study: How a News Portal Used WebSockets for Real-Time Alerts

Background

A news portal wanted to deliver **instant breaking news alerts** to readers **without page refreshes**.

Challenges

- **Traditional polling (AJAX requests)** caused **server load issues**.
- **Delays in delivering news updates** reduced user engagement.

Solution: Implementing WebSockets for Real-Time Notifications

The development team:

✓ Replaced **AJAX polling with WebSockets** for instant updates.
✓ **Optimized database storage** for saving alerts.
✓ Created **WebSocket clients for mobile & desktop browsers**.

Results

- **70% reduction in server requests,** improving efficiency.
- **Real-time updates** increased user engagement.
- **Faster news delivery,** making the site more competitive.

This case study shows how **WebSockets provide instant data updates** while reducing server load.

Exercise

1. Modify the system to allow **only authenticated users** to send notifications.
2. Store notifications in **MongoDB** and display past alerts to new users.
3. Extend the client UI to **show timestamps** for each notification.

Conclusion

✓ We built a **real-time notification system** using **Socket.io & WebSockets**.
✓ Users **send and receive notifications instantly**.
✓ We enhanced the system by **storing notifications in MongoDB**.