Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

REAL-TIME COMMUNICATION & BACKGROUND JOBS (WEEKS 9-10)

SETTING UP SOCKET.IO WITH EXPRESS.JS

CHAPTER 1: INTRODUCTION TO WEBSOCKETS & SOCKET.IO

1.1 What is WebSockets?

WebSockets is a **real-time communication protocol** that enables **bidirectional, persistent connections** between a client and a server. Unlike traditional HTTP, WebSockets allow servers to **send updates to clients** without waiting for a new request.

- **Why Use WebSockets?**

✓ Provides **low-latency real-time updates**.
✓ Reduces server load compared to frequent HTTP polling.
✓ Ideal for **chat apps, live notifications, gaming, and collaboration tools**.

- **WebSockets vs. HTTP**

Feature	HTTP	WebSockets
Connection Type	Request-response	Persistent

Communication	One-way (client to server)	Two-way (server & client)
Performance	Requires multiple requests	Faster with a single connection

---

1.2 What is Socket.io?

Socket.io is a **Node.js library** that simplifies **WebSocket communication** between clients and servers. It provides:
✓ **Automatic reconnection** if the connection is lost.
✓ **Built-in event handling** for custom messages.
✓ **Room-based communication** for broadcasting messages.

📌 **Installing Socket.io for Express.js:**

npm install express socket.io

✓ express ⟶ Web framework for handling HTTP requests.
✓ socket.io ⟶ Enables real-time bidirectional communication.

CHAPTER 2: SETTING UP SOCKET.IO IN AN EXPRESS.JS SERVER

2.1 Creating a Basic Express.js Server with Socket.io

📌 **Step 1: Import Required Modules (server.js)**

```
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");

});


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

✓ Creates an **HTTP server** and attaches Socket.io to it.

2.2 Listening for WebSocket Connections

📌 Step 2: Handling WebSocket Events (server.js)

```
io.on("connection", (socket) => {

  console.log("A user connected:", socket.id);


  socket.on("disconnect", () => {

    console.log("User disconnected:", socket.id);

  });

});
```

✓ Logs when a **user connects or disconnects**.

📌 **Start the Server:**

node server.js

✓ The **Socket.io server is now running on port 3000**.

CHAPTER 3: CONNECTING A CLIENT TO THE SOCKET.IO SERVER

3.1 Setting Up a Client-Side Connection

📌 **Create an HTML file (index.html) for a simple client:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Socket.io Client</title>
  <script src="https://cdn.socket.io/4.0.1/socket.io.min.js"></script>
</head>
<body>
  <h2>Socket.io Client</h2>
  <script>
    const socket = io("http://localhost:3000");

    socket.on("connect", () => {
      console.log("Connected to server:", socket.id);
    });
```

```
    socket.on("disconnect", () => {
      console.log("Disconnected from server");
    });
  </script>
</body>
</html>
```

📌 **Open index.html in a browser**

✓ Client connects **automatically to the server**.

✓ Console logs Connected to server: <socket_id>.

CHAPTER 4: SENDING & RECEIVING MESSAGES IN REAL-TIME

4.1 Implementing Message Exchange Between Server & Client

📌 **Step 1: Modify server.js to Handle Messages**

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

  socket.on("chatMessage", (message) => {
    console.log("Message received:", message);
    io.emit("chatMessage", message); // Broadcast message to all clients
  });
```

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

✓ Listens for "chatMessage" and **broadcasts it to all clients**.

📌 Step 2: Update index.html to Send & Display Messages

```
<input type="text" id="messageInput" placeholder="Type a message">
<button onclick="sendMessage()">Send</button>
<ul id="messages"></ul>

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

  function sendMessage() {
    const message = document.getElementById("messageInput").value;
    socket.emit("chatMessage", message);
  }
```

```
  socket.on("chatMessage", (message) => {
    const li = document.createElement("li");
    li.textContent = message;
    document.getElementById("messages").appendChild(li);
  });
</script>
```

📌 **Run the Express.js server (server.js)**
📌 **Open multiple index.html clients in the browser**
✓ Messages **broadcast in real-time** between all connected clients.

CHAPTER 5: BROADCASTING & ROOMS IN SOCKET.IO

5.1 Broadcasting Messages to All Clients

📌 **Example: Sending Messages to Everyone Except the Sender**

```
socket.broadcast.emit("chatMessage", "A new user has joined the chat");
```

✓ Notifies all users **except the sender**.

5.2 Creating Private Chat Rooms

📌 **Example: Joining & Broadcasting to a Room**

```
socket.on("joinRoom", (room) => {
  socket.join(room);
  socket.to(room).emit("message", `User joined room: ${room}`);
```

```
});
```

✓ Users in the **same room receive messages**.

📌 **Example: Sending a Message to a Room**

```
socket.to("room1").emit("message", "Hello, Room 1!");
```

✓ Only users in room1 receive the message.

CHAPTER 6: SECURING WEBSOCKETS WITH AUTHENTICATION

6.1 Adding JWT Authentication to Socket.io

📌 **Modify server.js to Verify JWT Tokens**

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


io.use((socket, next) => {
  const token = socket.handshake.auth.token;
  if (!token) return next(new Error("Authentication error"));

  jwt.verify(token, "your_secret_key", (err, decoded) => {
    if (err) return next(new Error("Invalid token"));
    socket.user = decoded;
    next();
  });
});
```

✓ Clients **must send a valid JWT token** to connect.

📌 **Client-Side (Passing Token When Connecting)**

```
const socket = io("http://localhost:3000", {
  auth: { token: "YOUR_JWT_TOKEN" }
});
```

✓ **Unauthorized users cannot connect** to the WebSocket.

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

Challenges Faced by Slack

✓ Handling **millions of real-time messages per second**.
✓ Ensuring **low-latency message delivery**.

Solutions Implemented

✓ Used **WebSockets for persistent connections**.
✓ Implemented **chat rooms for private & group messages**.
✓ Used **JWT authentication** to secure user sessions.

🔹 **Key Takeaways from Slack's WebSocket Strategy:**
✓ **WebSockets enable efficient real-time communication.**
✓ **Rooms improve chat performance by isolating message delivery.**
✓ **Secure authentication prevents unauthorized access.**

📝 **Exercise**

✅ Implement **Socket.io in an Express.js API**.
✅ Build a **real-time chat app** using WebSockets.
✅ Implement **user authentication with JWT** in Socket.io.
✅ Use **chat rooms to create private conversations**.

Conclusion

✓ Socket.io enables real-time communication using WebSockets.
✓ Express.js can integrate WebSockets for chat, notifications, and live updates.
✓ Rooms allow targeted messaging for private conversations.
✓ Adding authentication secures WebSocket connections.

Implementing Real-Time Chat Applications with Express.js & Socket.io

Chapter 1: Introduction to Real-Time Chat Applications

1.1 What is a Real-Time Chat Application?

A real-time chat application enables users to **send and receive messages instantly** without needing to refresh the page. It is widely used in:

✓ **Messaging apps (WhatsApp, Telegram, Messenger)**
✓ **Customer support chat systems**
✓ **Collaboration tools (Slack, Microsoft Teams)**

🔹 **Why Use WebSockets for Real-Time Communication?**

✓ **Bi-directional communication** – Server and client can send/receive messages.
✓ **Low latency** – Faster than traditional HTTP polling.
✓ **Efficient use of network resources** – Maintains a persistent connection.

Chapter 2: Setting Up a Real-Time Chat Application

2.1 Installing Required Dependencies

📌 **Step 1: Create a Node.js Project and Install Packages**

mkdir chat-app

cd chat-app

npm init -y

npm install express socket.io http cors dotenv mongoose

✓ express → Web framework for handling requests.
✓ socket.io → Enables real-time communication using WebSockets.
✓ http → Required for integrating Express with socket.io.
✓ mongoose → Stores chat messages in MongoDB.
✓ cors → Handles cross-origin requests.

2.2 Setting Up Express & Socket.io Server

📌 Step 2: Create server.js and Configure Socket.io

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

const http = require("http");

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

const cors = require("cors");


const app = express();

const server = http.createServer(app);

const io = new Server(server, {

  cors: { origin: "*" }

});


app.use(cors());
```

```
app.get("/", (req, res) => res.send("Chat Server Running"));


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

  socket.on("sendMessage", (data) => {
    io.emit("receiveMessage", data); // Broadcast message to all users
  });

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

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

✓ **Creates a WebSocket connection** using socket.io.
✓ **Listens for incoming messages** and **broadcasts them to all clients**.

CHAPTER 3: CREATING A FRONTEND FOR REAL-TIME CHAT

3.1 Setting Up a React Frontend

📌 Step 3: Create a React App and Install Dependencies

npx create-react-app chat-client

cd chat-client

npm install socket.io-client

✓ socket.io-client allows **connecting to the WebSocket server**.

3.2 Connecting React to the WebSocket Server

📌 Step 4: Modify App.js in React Client

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

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

function App() {
  const [message, setMessage] = useState("");
  const [messages, setMessages] = useState([]);

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

```
  }, []);


  const sendMessage = () => {
    socket.emit("sendMessage", message);
    setMessage(""); // Clear input after sending
  };


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

```
export default App;
```

✓ **Listens for incoming messages** and updates the UI dynamically.
✓ **Sends messages using socket.emit()**.

📌 **Run the React Client:**

```
npm start
```

✓ Opens the **chat app in the browser** at http://localhost:3000.

CHAPTER 4: STORING MESSAGES IN MONGODB

4.1 Setting Up MongoDB for Chat Messages

📌 **Step 5: Create models/Message.js for Message Schema**

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

const MessageSchema = new mongoose.Schema({
  text: { type: String, required: true },
  timestamp: { type: Date, default: Date.now }
});

module.exports = mongoose.model("Message", MessageSchema);
```

✓ Defines a **schema for storing chat messages**.

📌 **Step 6: Connect to MongoDB in server.js**

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


mongoose.connect("mongodb://127.0.0.1:27017/chatDB", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"));
```

✓ Stores messages **in a MongoDB database**.

4.2 Storing Messages in Database

📌 Step 7: Modify server.js to Save Messages

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


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


  socket.on("sendMessage", async (data) => {
    const newMessage = new Message({ text: data });
    await newMessage.save();
    io.emit("receiveMessage", data);
  });
});
```

✓ Saves each **message in MongoDB** before broadcasting.

CHAPTER 5: BEST PRACTICES FOR SECURE REAL-TIME CHAT

- **1. Implement Authentication for Users**

✓ Use **JWT authentication** to secure chat messages.

- **2. Prevent Spam & Abuse**

✓ Use **Rate Limiting** to prevent spam messages.

- **3. Encrypt Messages for Privacy**

✓ Use **end-to-end encryption** for private chats.

- **4. Store Chat History in MongoDB**

✓ Archive older messages **efficiently for scalability**.

Case Study: How WhatsApp Implements Real-Time Chat

Challenges Faced by WhatsApp

✓ Handling **millions of concurrent users**.
✓ Ensuring **end-to-end encryption** for privacy.
✓ Delivering messages **instantly across devices**.

Solutions Implemented

✓ Used **WebSockets (Socket.io)** for fast message delivery.
✓ Implemented **message encryption** for security.
✓ Used **MongoDB for scalable chat storage**.

- **Key Takeaways from WhatsApp's Chat Architecture:**

✓ **WebSockets enable fast & real-time communication**.

✓ **Secure authentication prevents unauthorized access.**

✓ **MongoDB allows efficient chat message storage.**

📝 Exercise

✅ Create a **real-time chat server using Express.js & Socket.io.**

✅ Develop a **React client to send and receive messages.**

✅ Store chat messages **in MongoDB for persistence.**

✅ Implement **authentication before allowing users to chat.**

Conclusion

✓ **WebSockets (Socket.io) enable real-time communication.**

✓ **Express.js provides a backend for chat applications.**

✓ **MongoDB stores chat messages efficiently.**

✓ **Authentication secures user messages in real-time chat.**

HANDLING WEBSOCKET EVENTS & BROADCASTING IN EXPRESS.JS WITH SOCKET.IO

CHAPTER 1: INTRODUCTION TO WEBSOCKETS & BROADCASTING

1.1 What are WebSockets?

WebSockets enable **real-time, bidirectional communication** between a client and a server over a **single persistent connection**. Unlike HTTP, WebSockets **do not require multiple request-response cycles**, making them ideal for **real-time applications**.

- **Why Use WebSockets?**

✓ **Reduces latency** by maintaining a persistent connection.

✓ **Enables real-time communication** for chats, live notifications, and stock updates.

✓ **More efficient than traditional polling** which sends repeated HTTP requests.

- **Common Use Cases for WebSockets:**

Use Case	Example Application
Live Chat	WhatsApp, Slack, Messenger
Real-time Notifications	Facebook, Twitter alerts
Stock Market Updates	Trading platforms
Live Gaming & Multiplayer	Online games (Chess, Poker)

CHAPTER 2: SETTING UP WEBSOCKETS WITH EXPRESS.JS & SOCKET.IO

2.1 Installing Socket.io in Express.js

📌 Install Required Packages

npm install express socket.io cors

✓ express → Web framework to serve APIs and WebSockets.
✓ socket.io → Enables real-time WebSocket connections.
✓ cors → Allows communication between different origins.

2.2 Creating a WebSocket Server with Express.js

📌 Step 1: Set Up Express & Socket.io in server.js

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

const http = require("http");

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

const cors = require("cors");


const app = express();

app.use(cors());


const server = http.createServer(app);

const io = new Server(server, { cors: { origin: "*" } });
```

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

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

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

✓ **Creates a WebSocket server** that listens for client connections.
✓ Uses socket.on("connection", callback) to handle **new WebSocket connections**.
✓ Logs **user connection and disconnection events**.

📌 **Run the WebSocket Server**

```
node server.js
```

✓ Starts the WebSocket server on **port 3000**.

CHAPTER 3: HANDLING WEBSOCKET EVENTS IN EXPRESS.JS

3.1 Sending & Receiving Messages Between Client and Server

📌 **Modify server.js to Listen for Incoming Messages**

```
io.on("connection", (socket) => {
```

```
  console.log(`User Connected: ${socket.id}`);

  socket.on("message", (data) => {
    console.log(`Message from ${socket.id}: ${data}`);
    io.emit("message", data); // Broadcast message to all clients
  });

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

✓ socket.on("message", callback) listens for incoming messages.
✓ io.emit("message", data) **broadcasts messages** to all connected clients.

📌 Client-Side JavaScript to Send & Receive Messages

```
const socket = io("http://localhost:3000");

socket.on("connect", () => {
  console.log("Connected to WebSocket Server");
});

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

```
  console.log("New Message:", data);
});

// Sending a message to the server
socket.emit("message", "Hello, WebSocket!");
```

✓ Sends "Hello, WebSocket!" message to the server.
✓ Listens for incoming **messages from other clients**.

📌 **Console Output When a Message is Sent**

User Connected: e123456

Message from e123456: Hello, WebSocket!

CHAPTER 4: BROADCASTING WEBSOCKET EVENTS IN EXPRESS.JS

4.1 What is Broadcasting?

Broadcasting allows sending **messages to multiple clients simultaneously** without needing each client to send a request.

🔹 **Types of WebSocket Broadcasting:**

Broadcast Type	Description	Example Use Case
io.emit()	Sends message to **all clients**	Global notifications
socket.broadcast.emit()	Sends message to **all clients except sender**	Chat messages

io.to(room).emit()	Sends message to **specific room/channel**	Private chat rooms

---

4.2 Broadcasting Messages to All Clients

📌 Modify server.js to Broadcast Messages

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

  socket.on("chatMessage", (data) => {
    io.emit("chatMessage", data); // Broadcast message to all users
  });
});
```

✓ Uses io.emit() to **send messages to all clients**.

📌 Client-Side Code to Handle Chat Messages

```
socket.on("chatMessage", (data) => {
  console.log("New Chat Message:", data);
});

socket.emit("chatMessage", "Hello Everyone!");
```

✓ Messages sent by **one client are received by all**.

4.3 Broadcasting Messages to Specific Clients

📌 Sending Messages to Everyone Except Sender

```
socket.broadcast.emit("notification", "New User Joined the Chat!");
```

✓ Alerts **all users except the sender**.

📌 Sending Messages to a Specific Room (Chat Rooms)

```
socket.join("room1"); // Join a chat room
io.to("room1").emit("message", "Welcome to Room 1!");
```

✓ Messages are **only sent to clients in "room1"**.

CHAPTER 5: IMPLEMENTING WEBSOCKET EVENTS IN A CHAT APP

5.1 Implementing a Real-Time Chat Application

📌 Modify server.js to Handle Chat Events

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

  socket.on("joinRoom", (room) => {
    socket.join(room);
    console.log(`${socket.id} joined room ${room}`);
  });

  socket.on("chatMessage", (data) => {
```

```
    io.to(data.room).emit("chatMessage", { user: socket.id, message: data.message });
  });

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

✓ **Users join a room and send messages** to that specific room.

📌 **Client-Side Code for Chat Room Communication**

```
const socket = io("http://localhost:3000");

socket.emit("joinRoom", "room1"); // Join Room 1

socket.on("chatMessage", (data) => {
  console.log(`[${data.user}] says: ${data.message}`);
});

socket.emit("chatMessage", { room: "room1", message: "Hello Room 1!" });
```

✓ Sends messages **only to users in the same room**.

Case Study: How Slack Uses WebSockets for Real-Time Communication

Challenges Faced by Slack

✓ Managing **millions of simultaneous messages**.
✓ Ensuring **real-time synchronization across devices**.

Solutions Implemented

✓ Used **WebSockets for instant message delivery**.
✓ Implemented **broadcasting for workspace-wide notifications**.
✓ Designed **room-based messaging** for channels & private chats.

🔹 **Key Takeaways from Slack's Architecture:**
✓ **WebSockets enable instant messaging & presence tracking.**
✓ **Broadcasting allows real-time notifications across multiple users.**
✓ **Efficient room management ensures scalability.**

📝 Exercise

✅ Implement **WebSocket broadcasting** to all connected users.
✅ Build a **real-time notification system** using socket.broadcast.emit().
✅ Create **chat rooms** where users can join and message only within rooms.
✅ Deploy your **WebSocket server using Heroku or Vercel**.

Conclusion

✓ WebSockets provide real-time, bidirectional communication.
✓ Express.js & Socket.io simplify WebSocket event handling.
✓ Broadcasting enables sending messages to multiple users.
✓ Room-based messaging allows private chat functionality.

IMPLEMENTING TASK QUEUES WITH REDIS & BULLMQ IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO TASK QUEUES

1.1 What is a Task Queue?

A **task queue** is a system that allows applications to **process tasks asynchronously**, rather than executing them immediately. Task queues are useful when handling:

✓ **Background jobs** (e.g., sending emails, processing images).
✓ **Heavy computations** that should not block the main request.
✓ **Rate-limited operations** (e.g., API calls, scheduled tasks).

- **Why Use Task Queues?**

✓ Improves **performance** by running tasks in the background.
✓ Handles **delayed or scheduled jobs** efficiently.
✓ Ensures **scalability** by distributing tasks among workers.

- **Example Use Cases for Task Queues:**

Use Case	Example
Email Notifications	Sending order confirmation emails asynchronously.
Data Processing	Processing large CSV files without blocking the API.
Scheduled Jobs	Running daily reports at a specific time.

CHAPTER 2: INTRODUCTION TO REDIS & BULLMQ

2.1 What is Redis?

Redis is an **in-memory data store** used as a database, cache, and **message broker for task queues**. It enables:

✓ **High-speed message passing** between task producers and workers.

✓ **Data persistence,** ensuring tasks are not lost after server restarts.

📌 Install Redis on Your System

- **Mac/Linux:**
- brew install redis # For Mac
- sudo apt install redis-server # For Ubuntu
- **Windows:** Install from Redis for Windows.

📌 Start Redis Server:

redis-server

✓ This starts a **local Redis instance**.

2.2 What is BullMQ?

BullMQ is a **job queue library** for Node.js that uses Redis to handle background tasks efficiently.

🔹 Features of BullMQ:

✓ **Fast and reliable task processing.**

✓ Supports **delayed and scheduled jobs**.

✓ Provides **worker concurrency control**.

📌 Install BullMQ in Express.js Project

npm install bullmq

✓ Installs BullMQ for managing **task queues in Express.js**.

CHAPTER 3: SETTING UP BULLMQ WITH EXPRESS.JS & REDIS

3.1 Creating a Job Queue with BullMQ

📌 Step 1: Initialize Redis Connection (queue.js)

```
const { Queue } = require("bullmq");


const redisConnection = {
  host: "127.0.0.1",
  port: 6379
};


const emailQueue = new Queue("emailQueue", { connection: redisConnection });


module.exports = emailQueue;
```

✓ Creates a **queue named emailQueue** in Redis.
✓ Uses **BullMQ to manage task execution**.

3.2 Adding Jobs to the Queue

📌 Step 2: Create an API Route to Add Jobs (routes/jobs.js)

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

const emailQueue = require("../queue");


const router = express.Router();


router.post("/send-email", async (req, res) => {
  const { email, subject, message } = req.body;


  await emailQueue.add("sendEmail", { email, subject, message });


  res.json({ message: "Email task added to queue" });
});


module.exports = router;
```

✓ When a user **requests /send-email,** the email job is **added to the queue**.

📌 Example Request (Using Postman or cURL):

```
POST http://localhost:3000/jobs/send-email

Content-Type: application/json
```

```
{
  "email": "user@example.com",
  "subject": "Welcome!",
  "message": "Thanks for signing up."
}
```

📌 **Expected Response:**

```
{ "message": "Email task added to queue" }
```

CHAPTER 4: PROCESSING JOBS WITH BULLMQ WORKERS

4.1 Creating a Worker to Process Jobs

📌 **Step 3: Define a Worker to Process Queued Jobs (worker.js)**

```
const { Worker } = require("bullmq");

const redisConnection = {
  host: "127.0.0.1",
  port: 6379
};

// Define a worker to process jobs
const emailWorker = new Worker("emailQueue", async (job) => {
  console.log(`Processing email for ${job.data.email}...`);
```

```
  await sendEmail(job.data);
  console.log("Email sent successfully!");
}, { connection: redisConnection });

// Simulated email sending function
async function sendEmail({ email, subject, message }) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`Email sent to ${email}: ${subject} - ${message}`);
      resolve();
    }, 2000);
  });
}
```

✓ The **worker listens to emailQueue** and processes jobs as they arrive.
✓ Uses sendEmail() function to **simulate sending an email**.

📌 **Start the Worker Process in a New Terminal Window:**

```
node worker.js
```

✓ Starts processing queued tasks.

📌 **Example Worker Log Output:**

```
Processing email for user@example.com...
```

Email sent successfully!

CHAPTER 5: IMPLEMENTING DELAYED & SCHEDULED JOBS

5.1 Scheduling Jobs for Future Execution

📌 **Example: Adding a Job with a 10-Second Delay**

```
router.post("/schedule-email", async (req, res) => {
  const { email, subject, message, delay } = req.body;

  await emailQueue.add("sendEmail", { email, subject, message }, { delay });

  res.json({ message: `Email scheduled to be sent in ${delay / 1000} seconds` });
});
```

✓ Allows scheduling **emails to be sent later**.

📌 **Example Request:**

```
POST http://localhost:3000/jobs/schedule-email
Content-Type: application/json
{
  "email": "user@example.com",
  "subject": "Reminder",
```

```
  "message": "Don't forget your appointment!",
  "delay": 10000
}
```

✓ Schedules an email **10 seconds after request**.

📌 **Worker Log Output (after 10 seconds):**

```
Processing email for user@example.com...
Email sent successfully!
```

CHAPTER 6: MONITORING TASK QUEUES WITH BULL BOARD

6.1 Installing Bull Board

Bull Board provides a **dashboard to monitor queued jobs**.

📌 **Install Bull Board**

```
npm install @bull-board/express bullmq
```

📌 **Integrate Bull Board in Express (server.js)**

```
const { createBullBoard } = require("@bull-board/api");
const { ExpressAdapter } = require("@bull-board/express");
const { BullMQAdapter } = require("@bull-board/api/bullMQAdapter");

const serverAdapter = new ExpressAdapter();
```

createBullBoard({ queues: [new BullMQAdapter(emailQueue)], serverAdapter });

app.use("/admin/queues", serverAdapter.getRouter());

✓ Opens a **dashboard at http://localhost:3000/admin/queues** to monitor jobs.

Case Study: How LinkedIn Uses Task Queues

Challenges Faced by LinkedIn

✓ Handling **millions of email notifications** daily.
✓ Processing **background tasks** without blocking user experience.

Solutions Implemented

✓ Used **Redis-based task queues** for delayed job processing.
✓ Implemented **BullMQ for scalable background tasks**.
✓ Monitored job execution with **real-time dashboards**.

🔹 **Key Takeaways from LinkedIn's Job Processing Strategy:**
✓ **Task queues improve system responsiveness** by running jobs asynchronously.
✓ **Delayed tasks ensure notifications are sent at optimal times**.
✓ **Monitoring job execution helps detect failures early**.

📝 **Exercise**

✅ Set up **Redis and BullMQ** in an Express.js project.
✅ Create an API route to **queue background jobs**.

✅ Implement a **worker process to handle jobs**.
✅ Add **delayed job scheduling** to execute tasks at a future time.
✅ Integrate **Bull Board** to monitor job queues visually.

Conclusion

✓ BullMQ enables efficient task queuing in Express.js.
✓ Redis stores jobs and ensures fast task execution.
✓ Workers process background jobs asynchronously.
✓ Task scheduling and monitoring improve system efficiency.

USING NODE-CRON FOR SCHEDULED JOBS IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO SCHEDULED JOBS

1.1 What is a Scheduled Job?

A **scheduled job** is a task that runs **automatically at a specified time interval** without manual intervention. In Node.js applications, scheduled jobs are used for:

✓ **Automated data backups**
✓ **Sending email notifications**
✓ **Generating reports**
✓ **Running database cleanups**

1.2 What is node-cron?

node-cron is a **task scheduler for Node.js** that allows developers to run **cron jobs** inside Express.js applications.

🔹 **Why Use node-cron?**

✓ Runs **automated background tasks** without manual execution.
✓ Uses **cron syntax** for scheduling jobs.
✓ Lightweight and easy to integrate with **Express.js & MongoDB**.

📌 **Installing node-cron in an Express.js Project:**

npm install node-cron

✓ Installs the node-cron package for scheduling jobs.

CHAPTER 2: SETTING UP NODE-CRON IN EXPRESS.JS

2.1 Creating a Basic Cron Job

📌 Modify server.js to Schedule a Job Every Minute:

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

const cron = require("node-cron");


const app = express();


cron.schedule("* * * * *", () => {

  console.log("Task is running every minute");

});


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

✓ The cron job **executes a task every minute**.

📌 Start the Express Server:

```
node server.js
```

✓ **Every minute,** "Task is running every minute" is logged in the console.

2.2 Understanding Cron Syntax

Cron jobs use a **5-part time format**:

```
* * * * *  command_to_run

- - - - -

|||||

|||| +---- Day of the week (0-7, Sunday=0)

||| +------ Month (1-12)

|| +-------- Day of the month (1-31)

| +---------- Hour (0-23)

+------------ Minute (0-59)
```

📌 **Common Cron Patterns:**

Schedule	Cron Syntax	Example Usage
Every minute	* * * * *	Log data every minute
Every hour	0 * * * *	Run cleanup every hour
Every day at midnight	0 0 * * *	Daily database backup
Every Monday at 8 AM	0 8 * * 1	Weekly reports

📌 **Example: Run Job at 2 AM Daily**

```
cron.schedule("0 2 * * *", () => {
  console.log("Running a job at 2 AM daily");
});
```

✓ Executes **every day at 2 AM**.

CHAPTER 3: RUNNING AUTOMATED JOBS IN EXPRESS.JS

3.1 Automating Email Notifications

📌 Step 1: Install nodemailer for Email Sending

```
npm install nodemailer
```

📌 Step 2: Create an Automated Email Scheduler

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


const transporter = nodemailer.createTransport({
  service: "gmail",
  auth: { user: "your-email@gmail.com", pass: "your-password" }
});


cron.schedule("0 9 * * *", () => {
  const mailOptions = {
    from: "your-email@gmail.com",
    to: "user@example.com",
    subject: "Daily Notification",
    text: "Good morning! This is your daily email."
  };


  transporter.sendMail(mailOptions, (error, info) => {
    if (error) console.error("Error sending email:", error);
```

```
    else console.log("Email sent:", info.response);
  });
});
```

✓ Sends an **automated email at 9 AM daily**.

3.2 Automating Database Cleanup Jobs

📌 **Example: Removing Inactive Users Every Week**

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

cron.schedule("0 0 * * 0", async () => {
  const result = await User.deleteMany({ active: false });
  console.log(`Deleted ${result.deletedCount} inactive users`);
});
```

✓ Deletes **inactive users every Sunday at midnight**.

CHAPTER 4: MANAGING CRON JOBS IN PRODUCTION

4.1 Preventing Duplicate Jobs

Running multiple instances of an Express app **can trigger duplicate jobs**. Use a database or Redis to **ensure only one instance runs the job**.

📌 **Using a Locking Mechanism with MongoDB**

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

const JobLog = require("./models/JobLog");


cron.schedule("0 0 * * *", async () => {

  const existingJob = await JobLog.findOne({ job: "database_cleanup", date: new Date().toISOString().slice(0, 10) });


  if (existingJob) return console.log("Job already ran today");


  await User.deleteMany({ active: false });

  await JobLog.create({ job: "database_cleanup", date: new Date().toISOString().slice(0, 10) });


  console.log("Database cleanup job executed.");

});
```

✓ Prevents **duplicate job execution** across multiple servers.

4.2 Logging Job Executions

📌 Example: Save Job Executions in a Log File

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


cron.schedule("0 0 * * *", () => {
```

```
  const log = `Job executed on ${new Date().toISOString()}\n`;
  fs.appendFile("job-log.txt", log, (err) => {
    if (err) console.error("Error logging job:", err);
  });
});
```

✓ **Tracks cron job execution history** in job-log.txt.

CHAPTER 5: BEST PRACTICES FOR USING NODE-CRON

- **1. Always Use Proper Cron Syntax**

✓ Incorrect syntax **can cause jobs to fail**.

```
cron.schedule("0 0 * * *", () => console.log("Job running at midnight"));
```

- **2. Monitor & Restart Failed Jobs**

✓ Use **PM2 process manager** to restart failed cron jobs.

```
npm install -g pm2
pm2 start server.js
pm2 save
```

- **3. Secure API Jobs with Authentication**

✓ Ensure **only authorized users trigger jobs**.

```
app.post("/trigger-job", authenticateToken, (req, res) => {
  // Run a scheduled job manually
});
```

Case Study: How Netflix Uses Scheduled Jobs for Content Management

Challenges Faced by Netflix

✓ Managing **millions of user watch histories**.
✓ Automating **content recommendations & cleanup**.

Solutions Implemented

✓ Used **cron jobs to delete old cache data**.
✓ Implemented **scheduled recommendation updates**.
✓ Automated **subscription renewal reminders** via email.

🔹 **Key Takeaways from Netflix's Strategy:**
✓ **Scheduled jobs optimize backend processes**.
✓ **Automated tasks improve efficiency & user experience**.
✓ **Logging job execution ensures reliability**.

📝 Exercise

✅ Implement a **cron job that logs a message every minute**.
✅ Schedule a job that **sends an email reminder every day at 8 AM**.
✅ Set up a **weekly job to clean up old database records**.
✅ Ensure jobs **run only once per day** using MongoDB locking.

Conclusion

✓ node-cron automates scheduled tasks efficiently in Express.js.

✓ Cron jobs improve database performance & automate notifications.

✓ Logging & locking mechanisms prevent duplicate job execution.

✓ Following best practices ensures reliable and secure scheduled jobs.

Optimizing API Performance with Caching

Chapter 1: Introduction to API Caching

1.1 What is API Caching?

API caching is the process of **storing frequently requested data** in memory or storage to reduce redundant computations and improve API response times. Instead of **fetching data from the database repeatedly**, the API retrieves cached data when possible.

- **Why Use Caching?**

✓ **Reduces server load** by minimizing database queries.
✓ **Improves response time** by serving pre-fetched data.
✓ **Enhances scalability** by reducing computational overhead.

- **Types of API Caching:**

Type	Description	Example
Memory Cache	Stores data in memory for fast retrieval.	Redis, Node.js Cache
Database Cache	Caches query results to avoid repeated execution.	PostgreSQL Query Cache
CDN Caching	Caches API responses globally for fast delivery.	Cloudflare, AWS CloudFront

Chapter 2: Implementing In-Memory Caching in Express.js

2.1 Using node-cache for Basic Caching

📌 Step 1: Install node-cache Package

npm install node-cache

📌 Step 2: Set Up In-Memory Caching in Express.js

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

const NodeCache = require("node-cache");


const app = express();

const cache = new NodeCache({ stdTTL: 60 }); // Cache expires after 60 seconds


app.get("/api/data", (req, res) => {
  const cachedData = cache.get("data");


  if (cachedData) {
    return res.json({ source: "cache", data: cachedData });
  }


  const apiData = { message: "Hello from API!" }; // Simulated API Response

  cache.set("data", apiData); // Store data in cache
```

```
  res.json({ source: "server", data: apiData });
});

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

✓ **Reduces database queries** by caching the API response.
✓ cache.get() **retrieves cached data if available**.
✓ cache.set() **stores new data in cache**.

📌 **Example Request:**

GET http://localhost:5000/api/data

✓ First request fetches **data from the server**; subsequent requests **fetch from cache**.

CHAPTER 3: USING REDIS FOR HIGH-PERFORMANCE CACHING

3.1 What is Redis?

Redis is an **in-memory key-value store** that provides **fast caching** for API responses. It supports **data persistence** and **automatic cache expiration**.

📌 **Step 1: Install Redis and redis npm package**

```
npm install redis dotenv
```

📌 **Step 2: Set Up Redis Connection in server.js**

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

const client = redis.createClient();
```

client.connect().then(() => console.log("Connected to Redis"));

✓ Establishes **a Redis connection for caching API responses**.

3.2 Caching API Responses with Redis

📌 Step 3: Modify API Endpoint to Use Redis Caching

```
app.get("/api/data", async (req, res) => {
  const cachedData = await client.get("data");

  if (cachedData) {
    return res.json({ source: "cache", data: JSON.parse(cachedData) });
  }

  const apiData = { message: "Hello from API!" }; // Simulated API Response
  await client.setEx("data", 60, JSON.stringify(apiData)); // Store data in Redis for 60 sec

  res.json({ source: "server", data: apiData });
});
```

✓ Stores data in Redis using setEx(key, expiry, value).
✓ Retrieves **cached data instantly** for subsequent API requests.

CHAPTER 4: IMPLEMENTING DATABASE QUERY CACHING

4.1 Why Cache Database Queries?

✓ **Reduces repetitive queries** that slow down APIs.
✓ **Prevents database overload** during high traffic.
✓ **Speeds up API responses** significantly.

📌 **Step 1: Cache Database Query Results in Redis**

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

mongoose.connect("mongodb://127.0.0.1:27017/myDB", {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

const User = mongoose.model("User", new mongoose.Schema({ name: String }));

app.get("/api/users", async (req, res) => {
  const cachedUsers = await client.get("users");
```

```
  if (cachedUsers) {
    return res.json({ source: "cache", data: JSON.parse(cachedUsers) });
  }

  const users = await User.find();
  await client.setEx("users", 300, JSON.stringify(users)); // Cache for 5 minutes

  res.json({ source: "server", data: users });
});
```

✓ **Caches user data for 5 minutes** to prevent excessive database calls.

📌 **Example Request:**

GET http://localhost:5000/api/users

✓ First request queries **MongoDB**, subsequent requests **fetch from Redis**.

CHAPTER 5: IMPLEMENTING API RATE LIMITING & PERFORMANCE OPTIMIZATION

5.1 Using express-rate-limit to Prevent Overuse

Rate limiting prevents excessive API requests that can **overload servers**.

📌 Step 1: Install express-rate-limit

```
npm install express-rate-limit
```

📌 Step 2: Apply Rate Limiting in server.js

```
const rateLimit = require("express-rate-limit");


const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 10, // Limit to 10 requests per minute
  message: "Too many requests. Please try again later."
});


app.use("/api", limiter);
```

✓ **Prevents API overuse** by limiting requests per IP address.

5.2 Using Content Delivery Networks (CDN) for API Caching

CDNs like **Cloudflare, AWS CloudFront** cache API responses globally.

📌 Steps to Enable API Caching with Cloudflare:

1. Go to **Cloudflare Dashboard**.
2. Navigate to **Caching → Configuration**.
3. Set **Caching Level** to **Standard** for API requests.

4. Enable **Edge Cache TTL** to store responses globally.

✓ **Speeds up API response time for global users**.

Case Study: How Twitter Uses API Caching

Challenges Faced by Twitter

✓ Handling **millions of real-time requests per second**.
✓ Ensuring **low-latency data retrieval**.
✓ Preventing **server overload during peak usage**.

Solutions Implemented

✓ Used **Redis caching** for frequently accessed API responses.
✓ Implemented **CDN caching (Cloudflare) for global access**.
✓ Enabled **rate limiting to prevent API abuse**.

🔹 **Key Takeaways from Twitter's API Caching Strategy:**
✓ **Caching reduces database load & improves API response times.**
✓ **Rate limiting prevents API overuse & security risks.**
✓ **CDN caching speeds up API responses for international users.**

📝 **Exercise**

✅ Implement **basic in-memory caching** using node-cache.
✅ Use **Redis to cache API responses** for database-heavy endpoints.
✅ Apply **rate limiting in an Express.js API** using express-rate-limit.
✅ Configure **CDN caching for a public API** using Cloudflare.

Conclusion

✓ Caching improves API response time & scalability.

✓ Redis provides persistent in-memory caching for Express.js APIs.

✓ Rate limiting prevents excessive requests & secures APIs.

✓ Using CDN caching speeds up API responses for global users.

ASSIGNMENT:

BUILD A REAL-TIME NOTIFICATION SYSTEM USING SOCKET.IO.

Assignment Solution: Build a Real-Time Notification System Using Socket.io

Step 1: Setting Up the Project

1.1 Install Required Dependencies

📌 **Initialize a Node.js project and install dependencies**

mkdir real-time-notifications

cd real-time-notifications

npm init -y

npm install express socket.io http mongoose cors dotenv

✓ express → Web framework for handling HTTP requests.
✓ socket.io → Enables real-time WebSocket connections.
✓ mongoose → Handles database operations in MongoDB.
✓ cors → Allows communication between different origins.
✓ dotenv → Manages environment variables securely.

Step 2: Creating the Express.js Server with Socket.io

2.1 Setting Up Express & WebSocket Server

📌 **Create server.js and set up WebSocket connections**

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

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

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

// Store connected users
let onlineUsers = {};

// Socket.io Connection Handling
io.on("connection", (socket) => {
```

```
  console.log(`User Connected: ${socket.id}`);

  // Register user when they connect
  socket.on("registerUser", (userId) => {
    onlineUsers[userId] = socket.id;
    console.log(`User Registered: ${userId} -> ${socket.id}`);
  });

  // Send Notification to a Specific User
  socket.on("sendNotification", ({ receiverId, message }) => {
    if (onlineUsers[receiverId]) {
      io.to(onlineUsers[receiverId]).emit("receiveNotification", { message });
      console.log(`Notification sent to User ${receiverId}`);
    }
  });

  // Broadcast Notifications to All Users
  socket.on("broadcastNotification", (message) => {
    io.emit("receiveNotification", { message });
    console.log("Broadcast Notification:", message);
  });
```

```
  // Handle user disconnect
  socket.on("disconnect", () => {
    Object.keys(onlineUsers).forEach((key) => {
      if (onlineUsers[key] === socket.id) delete onlineUsers[key];
    });
    console.log(`User Disconnected: ${socket.id}`);
  });
});

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

✓ **Stores connected users** in an object for sending targeted notifications.
✓ **Handles user registration, notification sending, and disconnection events**.

Step 3: Creating a Notification Model in MongoDB

3.1 Defining a Notification Schema

📌 Create models/Notification.js to store notifications in MongoDB

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

```
const NotificationSchema = new mongoose.Schema({
  receiverId: { type: String, required: true },
  message: { type: String, required: true },
  isRead: { type: Boolean, default: false },
  timestamp: { type: Date, default: Date.now }
});

module.exports = mongoose.model("Notification", NotificationSchema);
```

✓ Stores **notifications with user ID, message, and timestamp**.

Step 4: Implementing REST API Endpoints for Notifications

4.1 Creating Notification Routes

📌 **Create routes/notifications.js to handle API requests**

```
const express = require("express");
const Notification = require("../models/Notification");

const router = express.Router();

// Fetch all notifications for a user
router.get("/:userId", async (req, res) => {
  try {
```

```
    const notifications = await Notification.find({ receiverId: req.params.userId }).sort({ timestamp: -1 });
    res.json(notifications);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Mark a notification as read
router.put("/:id/read", async (req, res) => {
  try {
    await Notification.findByIdAndUpdate(req.params.id, { isRead: true });
    res.json({ message: "Notification marked as read" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

module.exports = router;
```

✓ Allows users to **fetch and mark notifications as read**.

📌 Step 5: Add Routes to server.js

```
const notificationRoutes = require("./routes/notifications");
```

```
app.use("/api/notifications", notificationRoutes);
```

Step 5: Creating a React Frontend for Notifications

5.1 Installing Dependencies for React

📌 Set up a React project and install Socket.io client

```
npx create-react-app notifications-client

cd notifications-client

npm install socket.io-client axios
```

✓ socket.io-client → Connects to WebSocket server.
✓ axios → Fetches notifications from the backend API.

5.2 Connecting React to the WebSocket Server

📌 Modify App.js to Handle Notifications

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

import io from "socket.io-client";

import axios from "axios";


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


function App() {

  const [notifications, setNotifications] = useState([]);

  const userId = "user123"; // Simulated logged-in user ID
```

```
  useEffect(() => {
    socket.emit("registerUser", userId); // Register user on connection

    socket.on("receiveNotification", (data) => {
      setNotifications((prev) => [...prev, data]);
    });

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

  const sendNotification = () => {
    socket.emit("sendNotification", { receiverId: userId, message: "You have a new message!" });
  };

  return (
    <div>
      <h2>Real-Time Notifications</h2>
      <button onClick={sendNotification}>Send Test Notification</button>
      <ul>
```

```
          {notifications.map((notif, index) => (
            <li key={index}>{notif.message}</li>
          ))}
        </ul>
      </div>
    );
}

export default App;
```

✓ Registers **users to receive notifications**.
✓ Updates UI when **new notifications arrive**.

📌 **Run the React Client:**

```
npm start
```

✓ Opens the **notification system in the browser** at http://localhost:3000.

Step 6: Testing the Notification System

6.1 Testing Notifications with Postman

📌 **Send a Notification via API Request**

- **Method:** POST
- **URL:** http://localhost:5000/api/notifications
- **Body (JSON):**

```
{
```

```
  "receiverId": "user123",
  "message": "This is a test notification!"
}
```

✓ Should **store the notification in MongoDB** and **send it via WebSocket**.

📌 **Retrieve Notifications for a User**

- **Method:** GET
- **URL:** http://localhost:5000/api/notifications/user123

✓ Should **return all notifications** for the user.

Case Study: How Facebook Uses Real-Time Notifications

Challenges Faced by Facebook

✓ Managing **millions of notifications per second**.
✓ Ensuring **real-time delivery without delays**.

Solutions Implemented

✓ Used **WebSockets for instant updates**.
✓ Stored **unread notifications in a database**.
✓ Allowed users to **fetch old notifications asynchronously**.

🔹 **Key Takeaways from Facebook's Notification System:**
✓ **WebSockets enable instant alerts.**
✓ **Database persistence ensures notification history.**
✓ **Scalability is key to handling large user bases.**

Conclusion

✓ Socket.io provides real-time notifications using WebSockets.
✓ MongoDB stores notifications for persistence.
✓ Express.js handles WebSocket events and API requests.
✓ React updates UI dynamically when new notifications arrive.