Purwadhika
Digital Technology School

AI Fullstack Software Development

Asynchronous Processing and Real-Time Communication

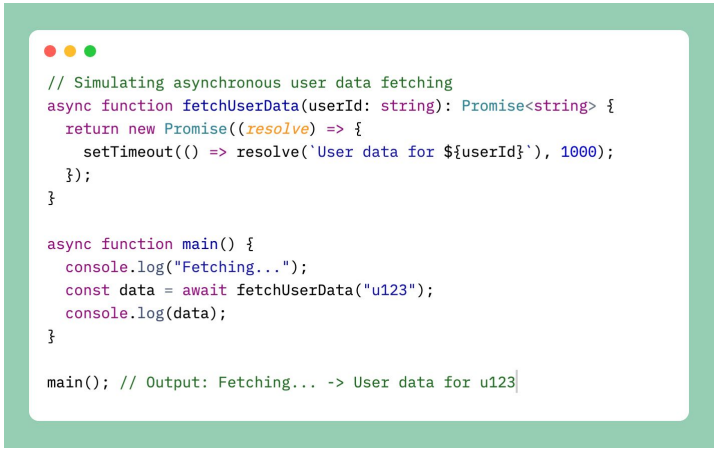
Outline

Purwadhika
Digital Technology School

- Asynchronous Processing
- Scheduling & Queue
- Communication with Socket.io

What is Asynchronous Processing?

- Asynchronous processing allows programs to perform multiple tasks concurrently without blocking the main thread.
- In Node.js, most operations (e.g., database calls, network requests, or I/O) are non-blocking, meaning the program can continue executing while waiting for those operations to complete.



```
// Simulating asynchronous user data fetching
async function fetchUserData(userId: string): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(`User data for ${userId}`), 1000);
  });
}

async function main() {
  console.log("Fetching...");
  const data = await fetchUserData("u123");
  console.log(data);
}

main(); // Output: Fetching... -> User data for u123
```

Introduction Scheduling & Queue

Task scheduling and queue management are crucial concepts in web development, especially in scenarios where you need to handle background jobs, long-running processes, or asynchronous tasks. These concepts help improve the performance, scalability, and responsiveness of web applications.

Task Scheduling

Task scheduling involves executing tasks or jobs at specified intervals or at predetermined times.

Use Cases:

- Periodic maintenance tasks (e.g., database cleanup, log rotation).
- Sending scheduled emails or notifications.
- Running background processes at specific times.

Implementation:

- In web development, task scheduling is often achieved using a task scheduler or cron jobs.
- On Linux systems, cron jobs can be used to schedule recurring tasks.
- In web frameworks like Laravel (PHP), Django (Python), or NodeJS, built-in schedulers or third-party packages facilitate task scheduling.

Queue Management

A queue is a data structure that manages tasks or jobs in a first-in, first-out (FIFO) order. Queue management involves handling the processing of tasks in an orderly manner.

Use Cases:

- Asynchronous processing of time-consuming tasks (e.g., image processing, file uploads).
- Handling background jobs to improve response times for user-facing actions.
- Decoupling components of a system to enhance scalability.

Implementation:

- When a web application receives a request that involves a time-consuming task, instead of processing it immediately, the task is added to a queue.
- A separate worker process or multiple worker processes continuously monitor the queue and execute tasks in the order they were added.
- Popular queue systems include Redis, RabbitMQ, and Apache Kafka.

Implement it in your Code

Create a directory structure:

Task Scheduling

In **src/cron/scheduleTask.ts**, define a simple scheduled task using **node-cron**:

```
import cron from 'node-cron';

export const scheduleTask = () => {
  // Schedule a task to run every minute
  cron.schedule('* * * * *', () => {
    console.log('Task running every minute.');
    // Add your task logic here
  });
};
```

*Note : Don’t forget to install `**node-cron**` first

Queue

In **src/queues/queueManager.ts**, define a simple queue using **bull**.

```
import Queue from 'bull';

const exampleQueue = new Queue('exampleQueue', {
  redis: {
    host: 'localhost',
    port: 6379,
  },
});

export default exampleQueue;
```

Final Code

```
import express from 'express';
import { scheduleTask } from './cron/scheduleTask';
import exampleQueue from './queues/queue';

const app = express();
const port = 3000;

// Express route
app.get('/', (req, res) => {
  res.send('Hello, world!');
});

// Start the scheduled task
scheduleTask();

// Example of using the queue
app.get('/enqueue', async (req, res) => {
  // Enqueue a job to the "exampleQueue"
  await exampleQueue.add('exampleJob', { data: 'some data' });
  res.send('Job enqueued!');
});

// Start the Express server
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

Create the Web Server

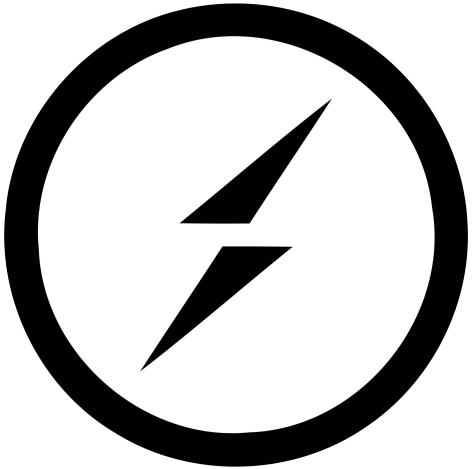
To start building a real-time app, we need a web server capable of handling HTTP and WebSocket connections.

```
import express, { Application } from "express";
import cors from "cors";

const app: Application = express();
app.use(cors());
app.use(express.json());

app.get("/", (_, res) => res.send("Server is running 🚀"));
export default app;
```

What is Socket.IO?

- Now that we have our web server, we add **Socket.IO** for real-time communication.
- **Socket.IO** provides a WebSocket-like protocol that allows event-based bi-directional communication between server and client.

https://socket.io/



Adding Real-Time Capabilities

- We'll use asynchronous event handlers to send and receive chat messages in real time
- Every connected client gets a unique socket ID.
- Using events like connection and disconnect, we track user activity asynchronously.

```
import http from "http";
import { Server } from "socket.io";
import app from "./app";
import { registerChatHandlers } from "./socket/chat.socket";

const PORT = 5000;
const server = http.createServer(app);

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

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

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

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

Building the Chat Handler

To handle real-time messages, we define a Socket.IO event handler. Whenever a client emits an event like **sendMessage**, the server listens and then broadcasts it asynchronously to all connected clients.

- `socket.on()` listens for client events asynchronously.
- `io.emit()` broadcasts to all connected clients.
- The message transfer happens instantly without blocking other operations.

```
import { Server, Socket } from "socket.io";
import { ChatMessage } from "../types/message.type";

export function registerChatHandlers(io: Server, socket: Socket) {
  socket.on("sendMessage", (msg: ChatMessage) => {
    console.log("📩 New message:", msg);
    io.emit("receiveMessage", msg); // Broadcast to all users
  });
}
```

Frontend Connection Setup

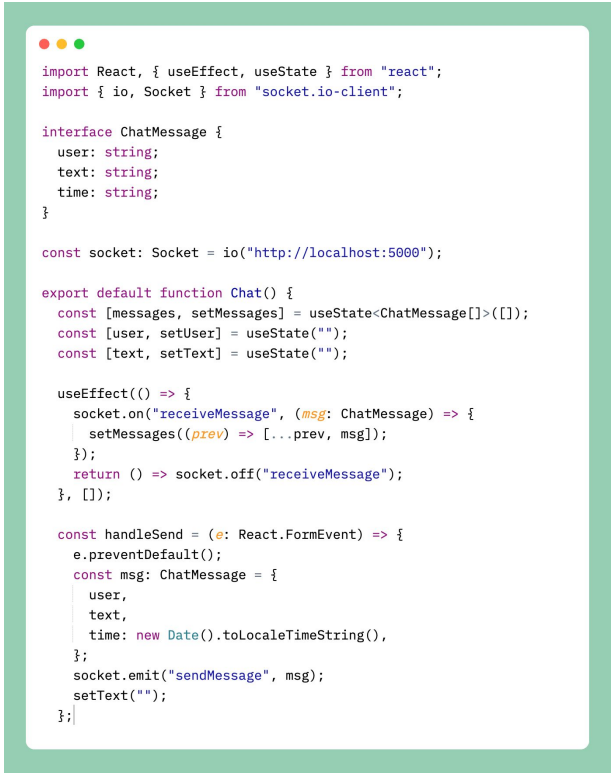
The frontend connects to the backend using the Socket.IO client library, establishing a WebSocket connection automatically. Every user gets a persistent connection that enables sending and receiving events in real time.

```
import { io, Socket } from "socket.io-client";

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

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

The client now connects asynchronously to the server. Once the connection is established, events can flow in both directions instantly.

Implementing Real-Time Messaging

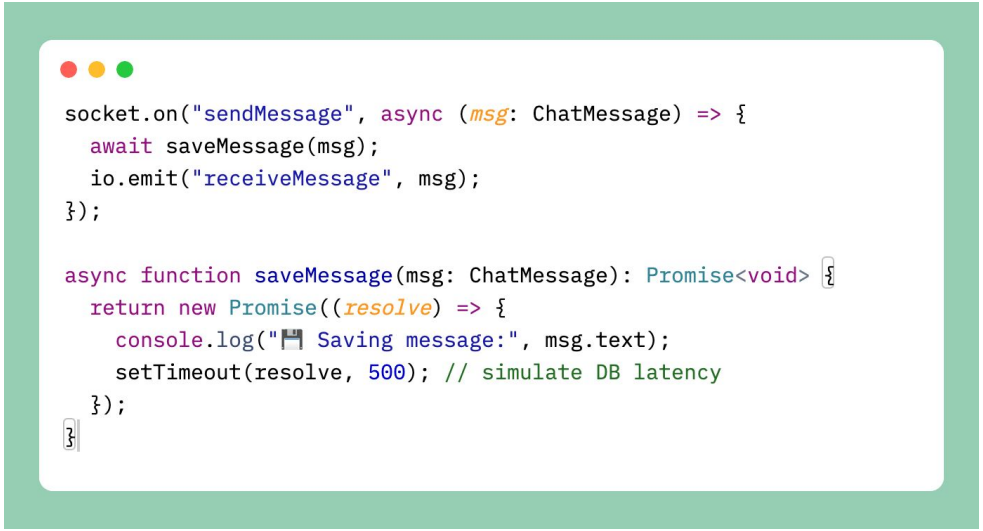
Now that both sides can communicate, we create a React chat component to handle message input and display. All message sends and receives happen asynchronously through socket events.



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

interface ChatMessage {
  user: string;
  text: string;
  time: string;
}

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

export default function Chat() {
  const [messages, setMessages] = useState<ChatMessage[]>([]);
  const [user, setUser] = useState("");
  const [text, setText] = useState("");

  useEffect(() => {
    socket.on("receiveMessage", (msg: ChatMessage) => {
      setMessages((prev) => [...prev, msg]);
    });
    return () => socket.off("receiveMessage");
  }, []);

  const handleSend = (e: React.FormEvent) => {
    e.preventDefault();
    const msg: ChatMessage = {
      user,
      text,
      time: new Date().toLocaleTimeString(),
    };
    socket.emit("sendMessage", msg);
    setText("");
  };
```

How Async Works Behind the Scenes

Under the hood, every message event goes through the Node.js event loop. While one client's message is being processed (or stored in DB), others can still send messages — this is non-blocking I/O.



Demo

Chitchat App : https://github.com/BagasDhitya/chitchat-app

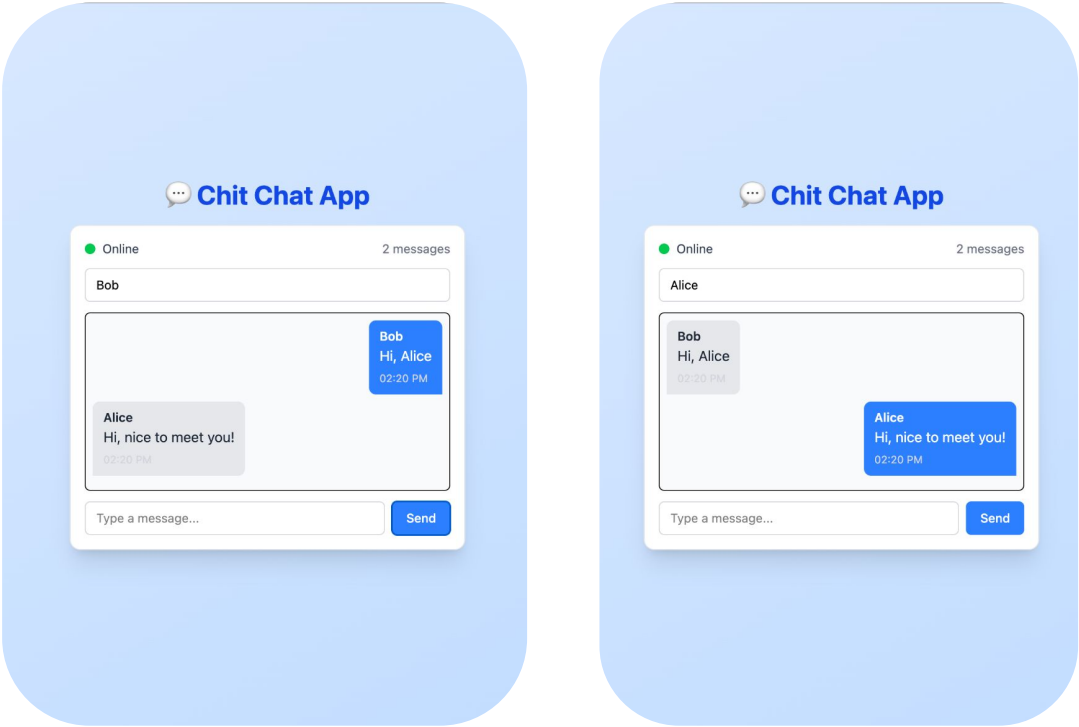
This project is organized into two main folders:

- api
- web

To get started, follow these steps:

1. Navigate to each folder and run `npm install` to install the dependencies.
2. In the `.env` file, set the port to **5000**.
3. After that, run `npm run dev` in both folders to start the development servers.

Result


Chit Chat App
Online
2 messages
Bob
Bob
Hi, Alice
02:20 PM
Alice
Hi, nice to meet you!
02:20 PM
Type a message...
Send
Chit Chat App
Online
2 messages
Alice
Bob
Hi, Alice
02:20 PM
Alice
Hi, nice to meet you!
02:20 PM
Type a message...
Send

Exercise

In this task, you will implement an article **scheduling feature** for the **Blog App** you previously built by extending the **article data model**, allowing users to set a **publish date**, and creating a **background scheduler** that **automatically publishes articles** at the **scheduled time**.

Thank you