Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Practical No. 9
Study and implementation of node.js

Problem Statement 1: Database Connectivity using SQL or Oracle

Write a Node.js program that connects to an Oracle/SQL database, retrieves data from a table, and displays the results

Install the Oracle driver (you'll need Oracle Instant Client installed on your machine):

```
npm install oracledb
```

Script (`oracle-query.js`):

```
// oracle-query.js
const oracledb = require('oracledb');

// Adjust these to your DB credentials and host
const dbConfig = {
  user: 'YOUR_DB_USER',
  password: 'YOUR_DB_PASSWORD',
  connectString: 'HOSTNAME:PORT/SERVICE_NAME'
  // e.g. 'localhost:1521/XEPDB1'
};

async function run() {
  let connection;

  try {
    // 1) Establish connection
    connection = await oracledb.getConnection(dbConfig);
    console.log('Connected to Oracle!');

    // 2) Execute a simple query
```

```
    const result = await connection.execute(
      `SELECT id, name, created_at
         FROM your_table
        WHERE ROWNUM <= 10`
    );

    // 3) Display results
    console.log('Query Results:');
    console.table(result.rows);

  } catch (err) {
    console.error('Error:', err);
  } finally {
    if (connection) {
      try {
        await connection.close();
        console.log('Connection closed');
      } catch (closeErr) {
        console.error('Error closing connection:', closeErr);
      }
    }
  }
}

run();
```

Run it:

```
node oracle-query.js
```

Problem Statement 2: Middleware (Express.js)

Q1: What is middleware in Node.js, particularly in the context of Express.js?
A1: In Express.js, middleware refers to functions that sit "in the middle" of the HTTP request–response cycle. Each middleware function receives the `Request` and `Response` objects (and a `next` callback), and can:

1. **Inspect or modify the request (`req`) or response (`res`) objects.**
2. **Terminate the request by sending a response (e.g., `res.send()`), or**
3. **Pass control to the next middleware in the stack by calling `next()`.**

Middleware is used for cross-cutting concerns like logging, authentication, parsing request bodies, error handling, and more.

Q2: How do you create custom middleware in Express.js?
A2: To create custom middleware, you write a function with the signature `(req, res, next)`. For example:

```
// logger.js
function logger(req, res, next) {
  console.log(`[${new Date().toISOString()}] ${req.method}
${req.url}`);
  next(); // Pass control to the next middleware
}

module.exports = logger;
```

Then you register it in your app:

```
const express = require('express');
const logger = require('./logger');

const app = express();

// Apply to all routes
app.use(logger);
```

```
// Or apply to a specific route
// app.get('/users', logger, (req, res) => { … });

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

app.listen(3000);
```

Q3: Explain how middleware is executed in order in an Express.js application.
A3:

1. **Registration order matters. Express processes middleware in the exact sequence you call `app.use()` or attach middleware to routes.**

2. **Linear flow: When a request comes in, Express starts with the first middleware. Each one runs in turn—if it calls `next()`, the next middleware executes; if it sends a response (or calls `next(err)`), it may short-circuit or jump to error-handling middleware.**

3. **Route handlers as middleware: Route handlers (e.g., `app.get('/path', handler)`) are just middleware that match specific HTTP methods and paths. They fit into the same chain.**

4. **Error middleware: Functions defined with four arguments `(err, req, res, next)` only run if an earlier middleware passes an error to `next(err)`. They too run in registration order.**

Example sequence:

```
app.use(mw1);
app.use(mw2);
app.get('/data', mw3, (req, res) => {
  res.send('Done');
});
app.use(errorHandler);
```

1. **mw1 runs**
2. **mw2 runs**
3. **Path matches `/data`, so mw3 runs**
4. **Handler runs and sends response**
5. **errorHandler is skipped (no error)**

Problem Statement 3: File System (fs) Module

Q1: How do you read and write files using the `fs` module in Node.js?
A1:

Reading a file (asynchronously):

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


fs.readFile('input.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File contents:', data);
});
```

Writing a file (asynchronously):

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


const content = 'Hello, Node.js!';

fs.writeFile('output.txt', content, 'utf8', (err) => {

  if (err) {

    console.error('Error writing file:', err);

    return;

  }

  console.log('File written successfully');

});
```

You can also use the Promise-based API under `fs.promises` for `async/await` style:

```
const { promises: fsp } = require('fs');


async function demo() {

  const data = await fsp.readFile('input.txt', 'utf8');

  console.log(data);


  await fsp.writeFile('output.txt', 'Some new content',
'utf8');

  console.log('Done');

}
```

```
demo().catch(console.error);
```

Q2: What is the difference between `fs.readFile()` and `fs.readFileSync()`?

A2:

- **`fs.readFile(path, [options], callback)`**
  - **Asynchronous: Non-blocking; Node can handle other tasks while reading.**
  - **You supply a callback `(err, data)` that fires when the read completes.**
- **`fs.readFileSync(path, [options])`**
  - **Synchronous: Blocking; the thread waits until the file is fully read.**
  - **Returns the file contents directly (or throws on error).**
  - **Use only for small files or scripts where blocking is acceptable.**

Q3: How can you check if a file or directory exists in Node.js?

A3:

- **Deprecated: `fs.exists(path, callback)` is deprecated because it can lead to race conditions.**

Recommended (callback):

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

```
fs.access('some/path', fs.constants.F_OK, (err) => {
  if (err) {
    console.log('Does not exist');
  } else {
    console.log('Exists');
  }
});
```

Recommended (Promise):

```
const { promises: fsp, constants } = require('fs');

async function check(path) {
  try {
    await fsp.access(path, constants.F_OK);
    console.log('Exists');
  } catch {
    console.log('Does not exist');
  }
}

check('some/path');
```

Synchronous check:

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


if (fs.existsSync('some/path')) {
  console.log('Exists');
} else {
  console.log('Does not exist');
}
```

Q4: How do you handle file operations in an asynchronous manner?
A4:

1. **Callbacks (Node style): Use `fs.readFile`, `fs.writeFile`, etc., passing a callback to handle the result or error.**

Promises / `async`–`await`: Use `fs.promises` (or wrap callbacks with `util.promisify`):

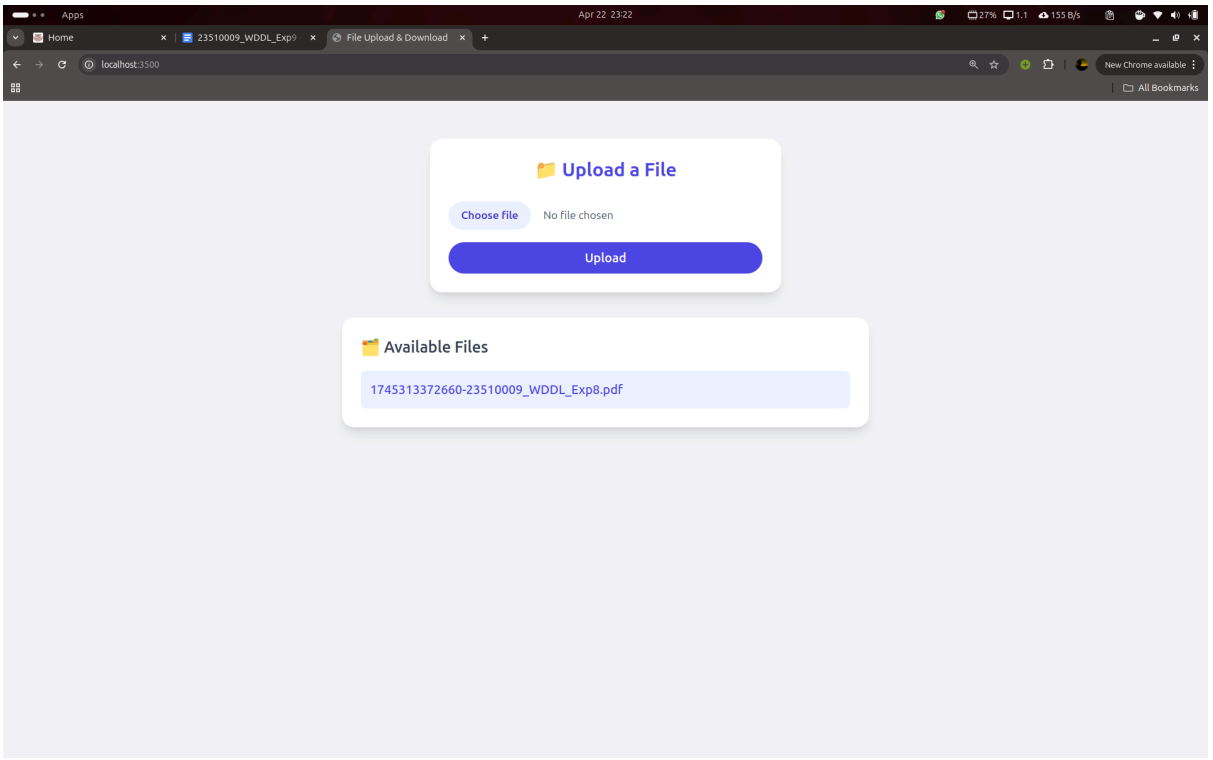
```
const { promises: fsp } = require('fs');


async function processFiles() {
  try {
    const data = await fsp.readFile('input.txt', 'utf8');
    const transformed = data.toUpperCase();
    await fsp.writeFile('output.txt', transformed, 'utf8');
```

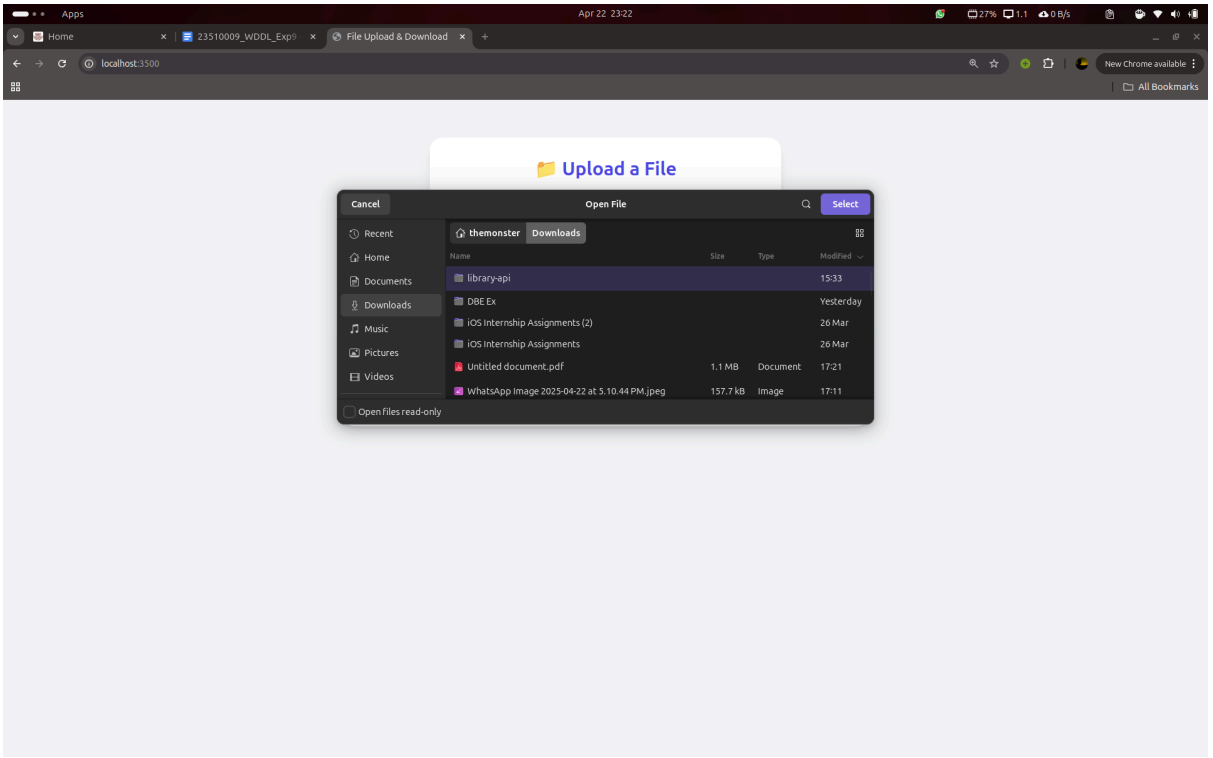
```
    console.log('All done');
  } catch (err) {
    console.error('File operation failed:', err);
  }
}

processFiles();
```

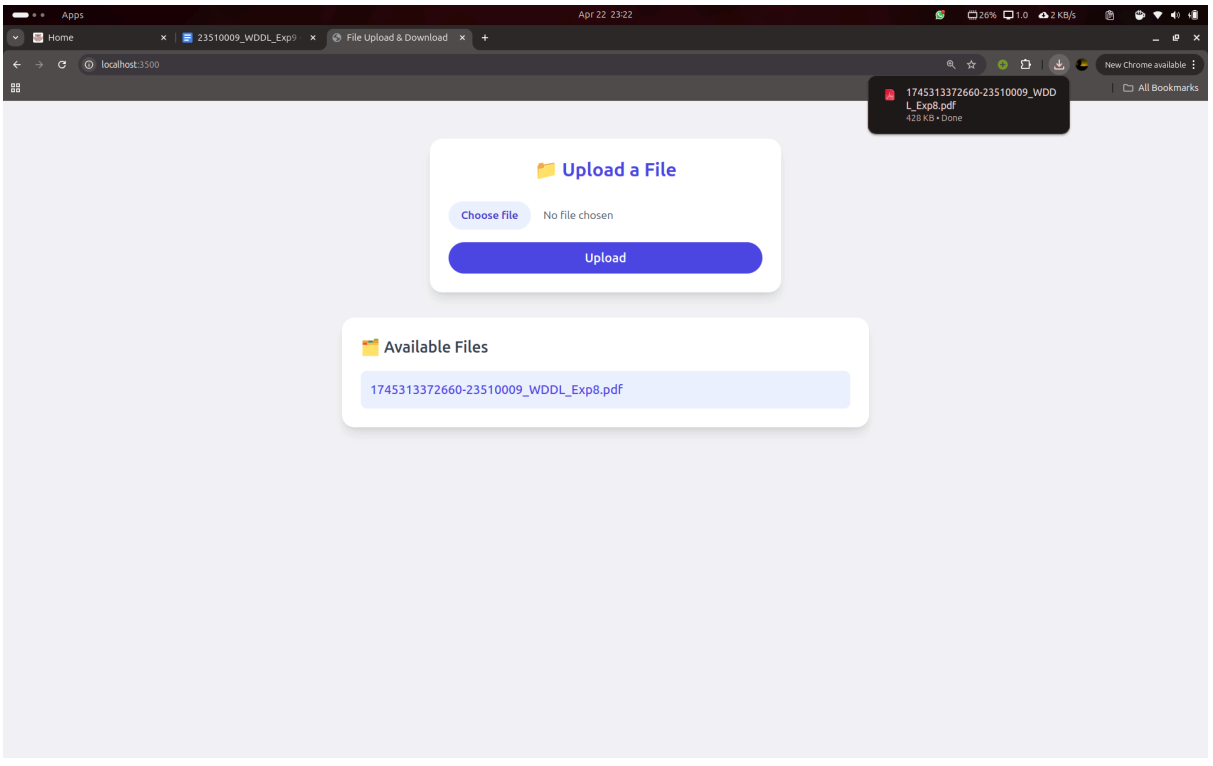
Streams: For large files, use `fs.createReadStream` and `fs.createWriteStream` to process data in chunks without loading the entire file into memory:

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

const reader = fs.createReadStream('large.txt', 'utf8');
const writer = fs.createWriteStream('out.txt');
reader.on('data', chunk => {
  // transform chunk if needed
  writer.write(chunk);
});
reader.on('end', () => {
  writer.end();
  console.log('Streamed copy complete');
});
```

Problem Statement 4: File Upload and Download API



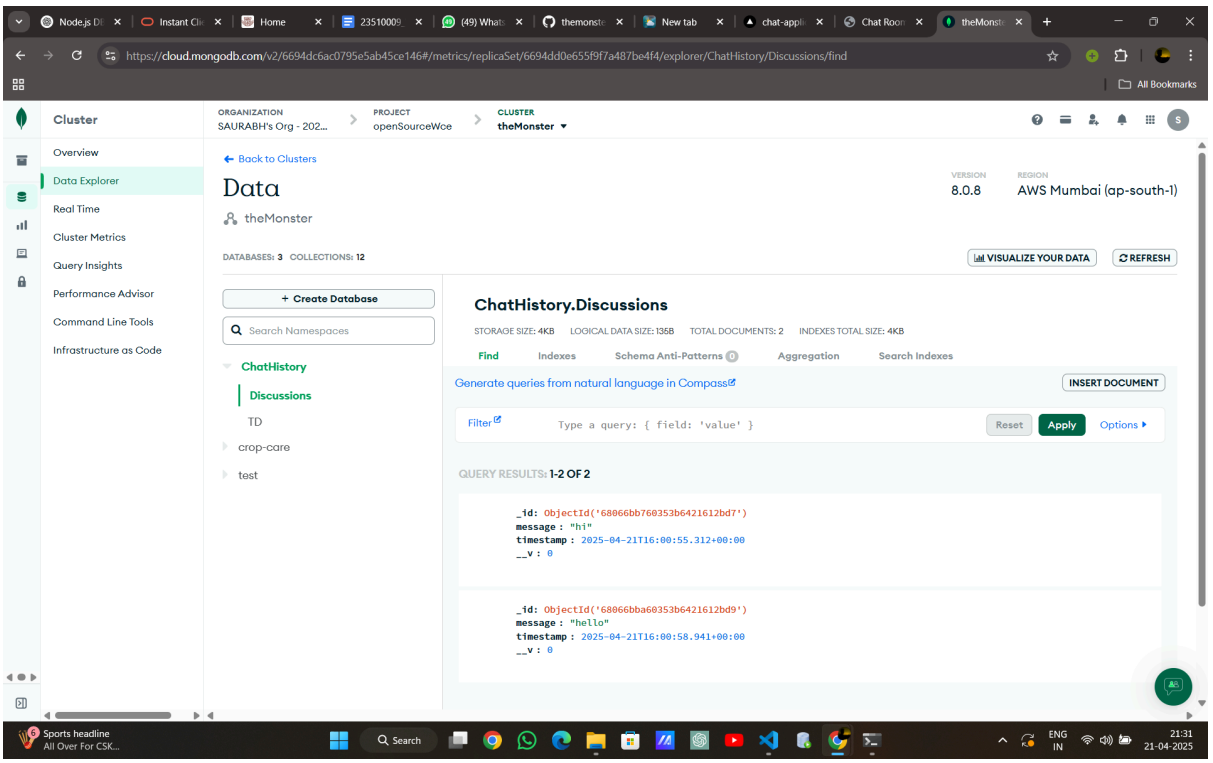

Apps
Apr 22 23:22
Home
23510009_WDDL_Exp9
File Upload & Download
localhost:3500
New Chrome available
All Bookmarks
Upload a File
Cancel
Open File
Select
Recent
Home
Documents
Downloads
Music
Pictures
Videos
themonster
Downloads
Name
Size
Type
Modified
library-api
15:33
DBE Ex
Yesterday
iOS Internship Assignments (2)
26 Mar
iOS Internship Assignments
26 Mar
Untitled document.pdf
1.1 MB
Document
17:21
WhatsApp Image 2025-04-22 at 5.10.44 PM.jpeg
157.7 kB
Image
17:11
Open files read-only

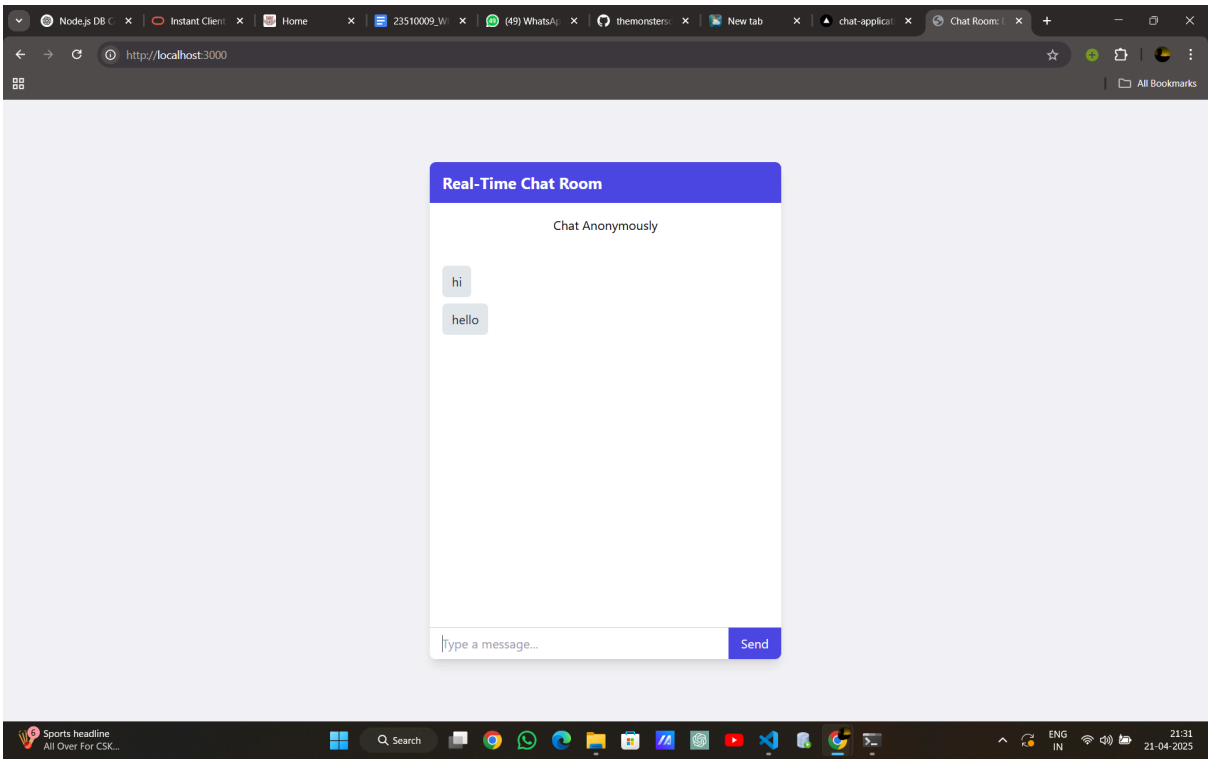

Apps
Apr 22 23:22
Home
23510009_WDDL_Exp9
File Upload & Download
localhost:3500
New Chrome available
All Bookmarks
1745313372660-23510009_WDDL_Exp8.pdf
428 KB • Done
Upload a File
Choose file
No file chosen
Upload
Available Files
1745313372660-23510009_WDDL_Exp8.pdf

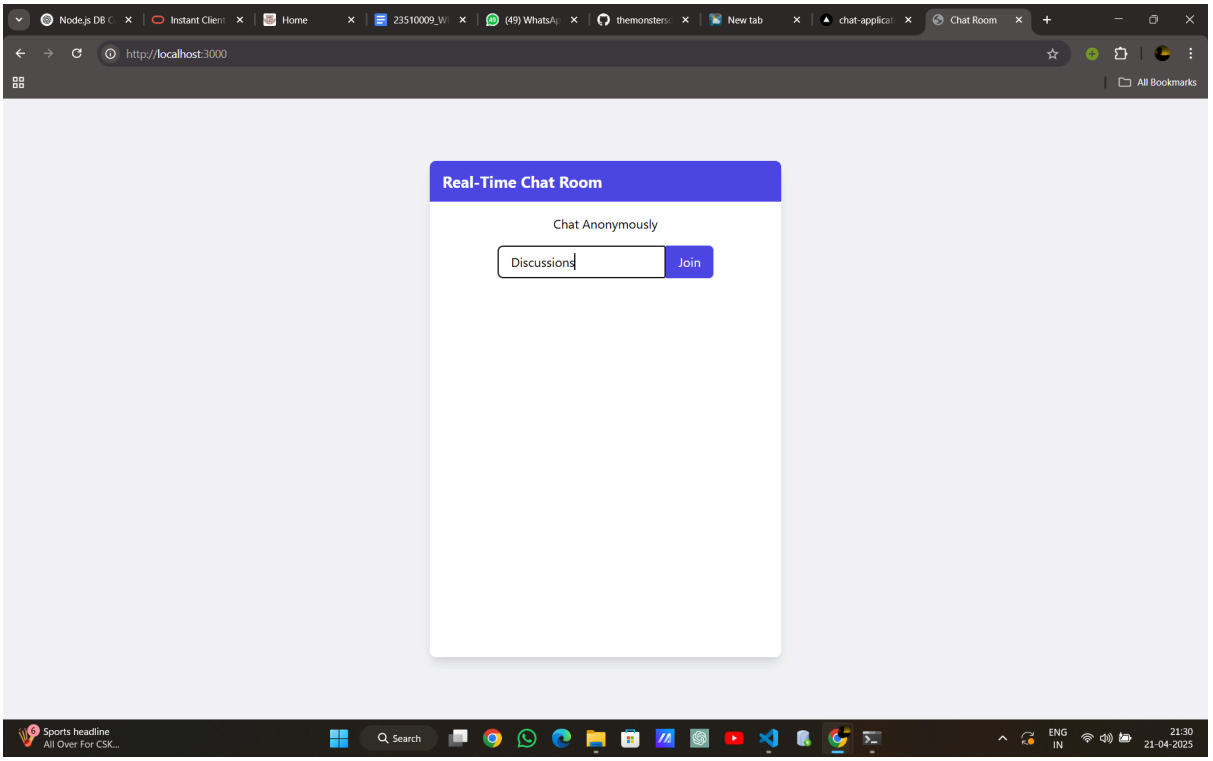
Problem Statement 5: Real-time Chat Application with Socket.io

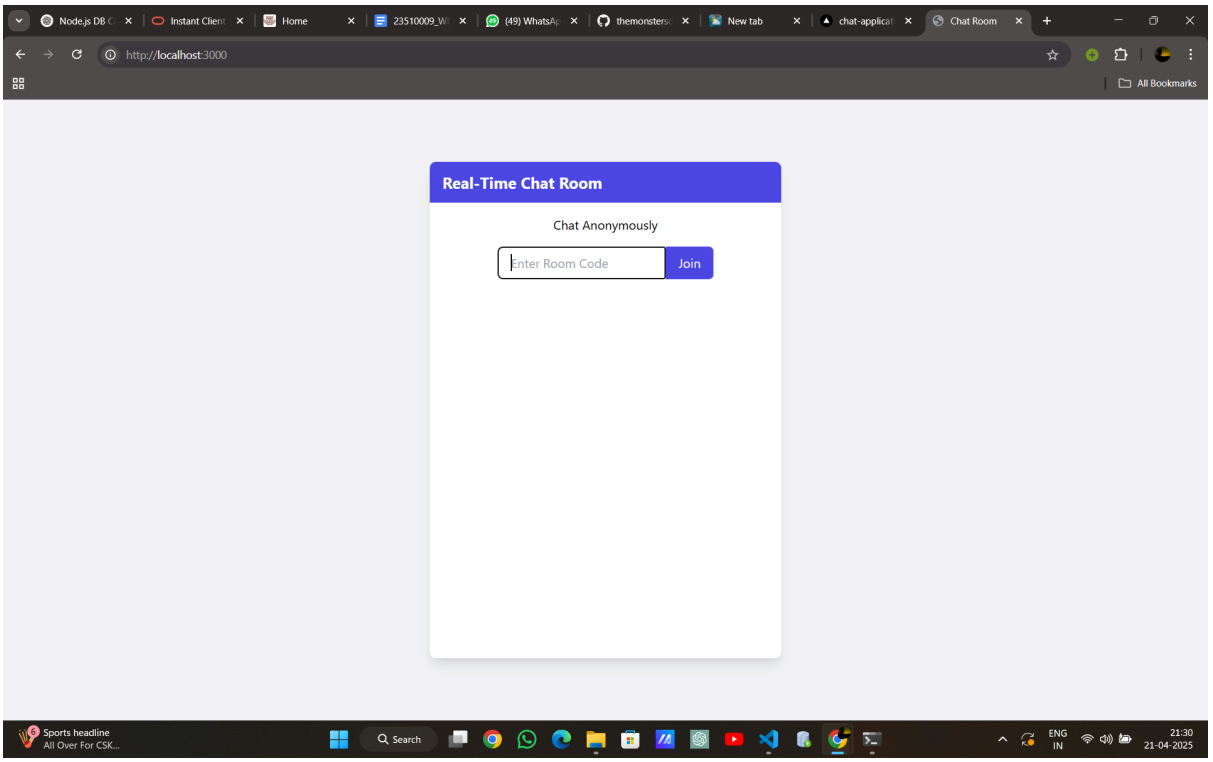
SourceCode:

https://github.com/themonstersd13/WddlExp9https://github.com/themonstersd13/WddlExp9






Node.js DB C
Instant Client
Home
23510009_W
(49) WhatsAp
themonsters
New tab
chat-applicat
Chat Room
http://localhost:3000
All Bookmarks
Real-Time Chat Room
Chat Anonymously
Discussions
Join
Sports headline
All Over For CSK...
Search
ENG
IN
21:30
21-04-2025


Node.js DB C
Instant Client
Home
23510009_W
(49) WhatsAp
themonsters
New tab
chat-applicat
Chat Room
http://localhost:3000
All Bookmarks
Real-Time Chat Room
Chat Anonymously
Enter Room Code
Join
Sports headline
All Over For CSK...
Search
ENG
IN
21:30
21-04-2025