Backend Series – Complete Notes

Day 1: What is Backend? + Node.js Setup

- What is Backend?
- - Backend refers to the server-side of an application.
  - It handles business logic, database operations, user authentication, and API responses.
- Backend Responsibilities:
- - Interact with databases (e.g., MongoDB, MySQL)
  - Serve data to frontend via APIs
  - Handle user sessions, security, and server-side logic

Frontend vs Backend:

Frontend	Backend
User Interface	Business Logic
HTML, CSS, JS	Node.js, Python, Java
Runs in browser	Runs on server

Why Use Node.js for Backend?

- - Allows JavaScript outside the browser
  - Built on Chrome's V8 engine
  - Non-blocking, event-driven architecture
  - Fast and scalable

Node.js Setup Steps

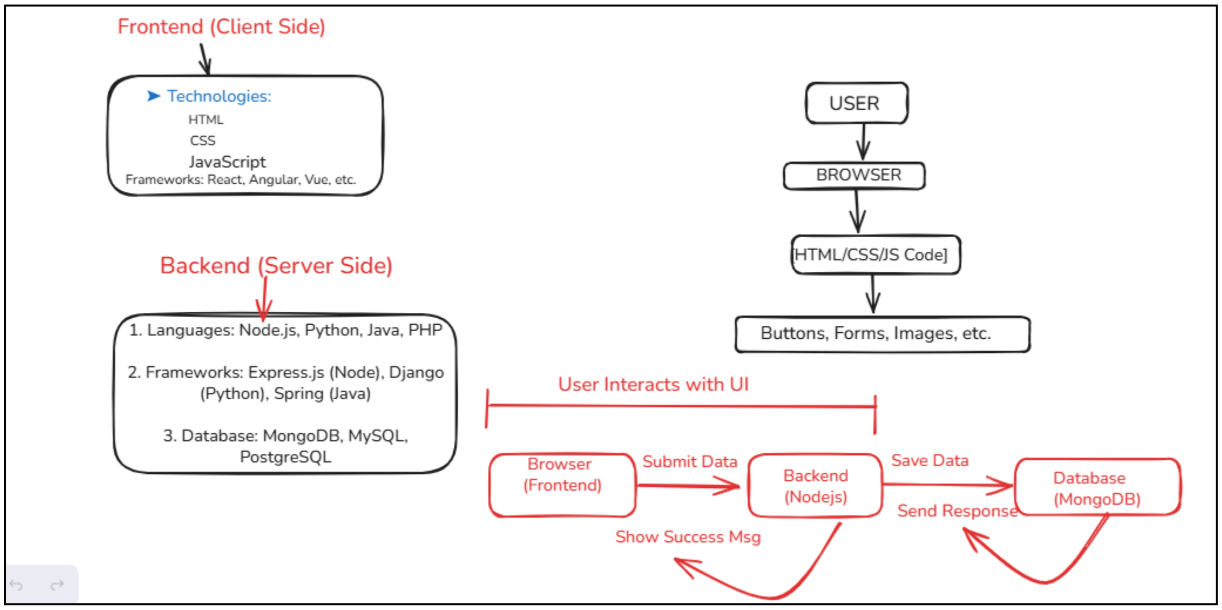
1. 1. Download Node.js
   2. Check installation:
   ```
   node -v
   npm -v
   ```
   3. Create file: index.js with console.log('Hello')
   4. Run using: node index.js

Day 2: Intro to Node.js and npm

- What is Node.js?

- - Runtime environment for running JavaScript on server-side
  - Single-threaded, event-driven model
  - Used for API development, real-time apps, microservices

  What is npm?
- - Node Package Manager
  - Manages packages, scripts, and dependencies



Useful npm Commands:

Command	Purpose
npm init -y	Create package.json
npm install express	Install express package
npm start	Run custom start script
npm uninstall	Remove a package

Example Code (Express Server):

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

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

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

Node.js File System (fs) Module

Introduction

The `fs` module in Node.js allows you to interact with the file system on your computer. It comes built-in with Node.js, so no installation is required.

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

You can use:

- **Synchronous** methods (blocking)
- **Asynchronous** methods (non-blocking)
- **Promise-based** API using `fs.promises`

Commonly Used Methods

1. Reading Files

Asynchronous (Non-blocking)

```
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Synchronous (Blocking)

```
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data);
```

---

2. Writing Files

`fs.writeFile()` – Overwrites content

```
fs.writeFile('file.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
  console.log('File written successfully');
});
```

`fs.appendFile()` – Adds content to end of file

```
fs.appendFile('file.txt', '\nAppended text', (err) => {
  if (err) throw err;
  console.log('Text appended');
});
```

Synchronous versions

```
fs.writeFileSync('file.txt', 'Some content');
fs.appendFileSync('file.txt', '\nMore content');
```

---

3. Deleting Files

```
fs.unlink('file.txt', (err) => {
  if (err) throw err;
  console.log('File deleted');
});

// Sync
fs.unlinkSync('file.txt');
```

---

4. Renaming/Moving Files

```
fs.rename('old.txt', 'new.txt', (err) => {
  if (err) throw err;
  console.log('File renamed');
});

// Sync
fs.renameSync('old.txt', 'new.txt');
```

---

5. Creating & Reading Directories

Create Directory

```
fs.mkdir('myDir', (err) => {
  if (err) throw err;
  console.log('Directory created');
});
```

Read Directory

```
fs.readdir('myDir', (err, files) => {
  if (err) throw err;
  console.log(files);
});
```

---

6. Remove Directory

```
fs.rmdir('myDir', (err) => {
  if (err) throw err;
  console.log('Directory removed');
});
```

Understanding Asynchronous Programming in Node.js

---

1 What is Asynchronous Programming?

- **Synchronous**: Code runs line by line. One task must finish before the next starts.

- **Asynchronous**: Code does **not** wait for a task to finish. It continues to the next task.

---

2 Why Node.js is Asynchronous?

Node.js is designed to be **non-blocking**. Instead of waiting for a task (like file reading or HTTP requests), Node handles other operations and returns results when available.

---

3 The Event Loop (Behind the Scenes)

The **Event Loop** is the heart of Node.js async behavior.

```
┌──────────────────────────────┐
│      Call Stack              │
├──────────────────────────────┤
│      Callback Queue          │
└──────────────────────────────┘
          ↓
     Event Loop □
```

If the call stack is empty, the event loop picks a task from the queue and runs it.

---

4 Handling Asynchronous Code in Node.js

A. Callback

A **callback** is a function passed as an argument that gets executed later.

```
function greet(name, callback) {
  console.log("Hello", name);
  callback();
}

greet("Baljinder", () => {
  console.log("Welcome to Node.js!");
});
```

B. Promises

A **Promise** represents a value that may be available now, later, or never.

```
const getUser = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve("User Data"), 2000);
  });
};

getUser().then(data => console.log(data));
```

C. Async/Await (Best Practice)

Modern and readable way of writing asynchronous code.

```
const getUser = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve("User Data"), 2000);
  });
};

async function showUser() {
  const user = await getUser();
  console.log(user);
}
showUser();
```

---

5 File System Example (`fs` Module)

Callback Style

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

fs.readFile('file.txt', 'utf-8', (err, data) => {
  if (err) return console.log(err);
  console.log("File Content:", data);
});
```

Promise Style

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

fs.readFile('file.txt', 'utf-8')
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

Async/Await Style

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

async function readFile() {
  try {
    const data = await fs.readFile('file.txt', 'utf-8');
    console.log(data);
  } catch (err) {
    console.log("Error:", err);
  }
}

readFile();
```

---

6 Simulating an API Call

```
function fakeApi(url) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Fetched data from ${url}`);
    }, 2000);
  });
}

async function fetchData() {
  const data = await fakeApi("https://example.com/api");
  console.log(data);
}
fetchData();
```

---

7 Error Handling

```
async function riskyOperation() {
  try {
    throw new Error("Something went wrong");
  } catch (error) {
    console.error("Caught:", error.message);
  }
}

riskyOperation();
```

---

8 Callback Hell vs Clean Async Code

Callback Hell:

```
doTask1(() => {
  doTask2(() => {
    doTask3(() => {
      console.log("All tasks done");
    });
  });
});
```

Better with Async/Await:

```
async function runTasks() {
  await doTask1();
  await doTask2();
  await doTask3();
  console.log("All tasks done");
}
```

---

9 Real-World Example

```
function getUser() {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: 1, name: "Baljinder" }), 1000);
  });
}

function getPostsByUser(userId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(["Post 1", "Post 2"]), 1000);
  });
}

async function main() {
  const user = await getUser();
  console.log("User:", user);

  const posts = await getPostsByUser(user.id);
  console.log("Posts:", posts);
}

main();
```