2-tier architachture

client

- it is the presentation layer which is also called ui;

server

- It is the combination of both hardware and software which accepts request, process is and sends back a response.

client ----------request----------> server

client <---------response---------- server

- this is called as 1 req-res cycle, after completing one cycle we can send more request

3-tier architechture

client/presentation/ui layer -------- request + data --> **server/application layer/ business layer** ----- data query -------> **database layer**

```
                                https
```

client/presentation/ui layer <--- response ---------- **server/application layer/ business layer** <--- response --------- **database layer**

technologies to build presetation layer

HTML, CSS, JS --> Native languages reactjs,vue, angularjs, etc.

technologies to build application layer

java, python, c, c#, c++, .net, etc

technologies save data

SQL and NoSQL

Nodejs

- It is not a language
- Nodejs is a runtime enviroment which executes js program outside the browser.
- Node.js is a open-source, cross-platform javascript runtime environment to execute js files outside the browser.
-

v8 + ***c++*** ====> NodeJS

Software Architecture & Node.js Notes

Software Architecture Patterns

2-Tier Architecture

Structure:

- **Client (Presentation Layer)**: The user interface layer that users interact with
- **Server**: Combination of hardware and software that accepts requests, processes them, and sends back responses

Communication Flow:

```
Client ——————request——————→ Server
Client ←——————response——————— Server
```

This is called one **request-response cycle**. After completing one cycle, the client can send more requests.

Key Points:

- Direct communication between client and server
- Server handles both business logic and data management
- Simpler architecture but less scalable

3-Tier Architecture

Structure:

```
Client/Presentation/UI Layer ——request + data——→ Server/Application/Business Layer ——data query—
—→ Database Layer
                            <---------- response————————                                    ←——res
ponse——
```

Layers Explained:

1. **Client/Presentation/UI Layer**
   - Handles user interface and user experience
   - Technologies: HTML, CSS, JavaScript (native languages)
   - Frameworks: React.js, Vue.js, Angular.js, etc.
2. **Server/Application/Business Layer**
   - Contains business logic and application processing
   - Technologies: Java, Python, C, C#, C++, .NET, Node.js, etc.
3. **Database Layer**

- Handles data storage and retrieval
- Technologies: SQL databases (MySQL, PostgreSQL) and NoSQL databases (MongoDB, Redis)

Advantages:

- Better separation of concerns
- More scalable and maintainable
- Independent scaling of each layer

Node.js Overview

What is Node.js?

- **NOT a programming language**
- **Runtime environment** that executes JavaScript programs outside the browser
- **Open-source, cross-platform** JavaScript runtime environment
- Built on Google's **V8 JavaScript engine** combined with **C++**

`V8 + C++ = Node.js`

- c++ is act as a API.

Key Features

- Executes JavaScript on the server side
- Event-driven, non-blocking I/O model
- Single-threaded but highly efficient
- Large ecosystem through npm (Node Package Manager)

JavaScript Promises & Async/Await

Promises

A Promise represents the eventual completion (or failure) of an asynchronous operation.

Basic Promise Structure:

```
let promise = new Promise((resolve, reject) => {
    // Asynchronous operation
    if (condition) {
        resolve("Success!");
    } else {
        reject("Error!");
    }
});

promise
    .then(result => console.log(result))
    .catch(error => console.log(error));
```

Real Example - Fetch API:

```
let promise = fetch("https://jsonplaceholder.typicode.com/posts");

promise
    .then(response => {
        console.log(response);
        return response.json(); // Returns another promise
    })
    .then(data => {
        console.log(data); // Actual JSON data
    })
    .catch(error => {
        console.log(error);
    });
```

Async/Await

Key Points:

- `async` is used in function declaration
- `await` is used inside the function body
- `async` function always returns a promise
- `await` suspends function execution until the promise resolves

Syntax:

```
async function getTodos() {
    try {
        let response = await fetch("https://jsonplaceholder.typicode.com/posts");
        let jsonData = await response.json();
        console.log(jsonData);
        return jsonData;
    } catch (error) {
        console.log("Error:", error);
    }
}

// Function call
getTodos();
```

Execution Order Example:

```
console.log(1); // Executes first

async function getTodos() {
    console.log(2); // Executes second
    let output = await fetch("url"); // Waits for response
    console.log(3); // Executes after fetch completes
}

getTodos();
console.log(4); // Executes third (doesn't wait for async function)
```

Output: 1, 2, 4, 3

Promise vs Async/Await Comparison

With Promises (Callback Hell):

```
fetch("url")
    .then(response => response.json())
    .then(data => {
        console.log(data);
        // More nested operations...
    })
    .catch(error => console.log(error));
```

With Async/Await (Cleaner):

```
async function fetchData() {
    try {
        let response = await fetch("url");
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log(error);
    }
}
```

Technology Stack Summary

Layer	Technologies
Frontend	HTML, CSS, JavaScript, React, Vue, Angular
Backend	Node.js, Java, Python, C#, .NET, PHP
Database	MySQL, PostgreSQL (SQL), MongoDB, Redis (NoSQL)

Best Practices

1. **Error Handling**: Always use try-catch with async/await or .catch() with promises
2. **Architecture**: Choose the right architecture pattern based on project requirements
3. **Separation of Concerns**: Keep presentation, business logic, and data layers separate
4. **Asynchronous Programming**: Use async/await for better code readability