Chapter 01: Introduction to Node.js

1. What is Node.js?

- **Node.js** is a JavaScript runtime environment built on Chrome's **V8 engine**.
- It allows JavaScript to run outside the browser, making it a powerful tool for various applications beyond web development.

2. Key Features of Node.js

a) Runtime Environment

- Node.js provides a **runtime environment** to execute JavaScript code outside the browser.
- It is powered by the **V8 engine**, the same engine used in Google Chrome.

b) Event-Driven Architecture

- Node.js follows an **event-driven architecture**, efficiently handling asynchronous operations.
- This design allows it to manage multiple tasks without waiting for one to complete before starting another.

c) Asynchronous I/O (Non-blocking I/O)

- Node.js performs **asynchronous I/O operations**, meaning tasks like reading/writing files or making network requests do not block the execution of other operations.
- This is different from traditional synchronous programming, where tasks are executed sequentially.

3. Development History of Node.js

a) Creation and Early Development

- **Ryan Dahl** developed Node.js in **2009**.
- Initially, he experimented with **SpiderMonkey**, Mozilla’s JavaScript engine, but later adopted Google’s **V8 engine** for its performance.

- The project was originally named **web.js** but was later renamed **Node.js** to reflect its broader potential.

b) Support from Joyent

- **Joyent**, a technology company, saw potential in Node.js and supported its development.

4. Comparison with Traditional Servers

- Before Node.js, most web servers used the **Apache HTTP Server**, which followed a **blocking I/O** model, limiting the number of concurrent connections.
- Node.js introduced a **non-blocking I/O** model, allowing it to handle multiple requests efficiently with fewer system resources.

5. The Evolution of NPM (Node Package Manager)

- **NPM** is a package manager for Node.js, allowing developers to install and manage libraries easily.
- Developed by **Joyent** in **2010**, it was initially available for **macOS and Linux**.
- In **2011**, Microsoft collaborated to bring NPM to **Windows**.

6. Leadership and Community Transitions

a) Leadership Changes

- In **2012**, **Ryan Dahl** stepped down, and **Isaac Z. Schlueter**, the creator of NPM, took over Node.js development.

b) The io.js Fork and Reunification

- In **2014**, due to internal disagreements, **Fedor Indutny** forked Node.js and created **io.js**.
- In **2015**, the Node.js and io.js communities resolved their conflicts, merging back into a single project.

c) Formation of the OpenJS Foundation

- By **2019**, the **JS Foundation** and **Node.js Foundation** merged, forming the **OpenJS Foundation**, ensuring long-term community-driven development.

7. Present and Future of Node.js

- Node.js continues to evolve with strong community support, making it one of the most popular JavaScript runtimes.
- Its ecosystem, powered by **NPM**, provides a vast collection of libraries for building scalable and efficient applications.

Namaste Node.js - Episode 2 Summary

Chapter 02: JavaScript on the Server

1. Servers in Node.js

- A **server** is a system that provides resources, data, services, or programs to other computers (clients) over a network.
- In Node.js, a server is primarily used to handle and respond to **client requests** over the **HTTP protocol**.
- Node.js servers are highly efficient due to their **event-driven, non-blocking I/O architecture**, which allows:
  - Handling multiple client requests simultaneously.
  - Avoiding the need to create new threads for each request.
  - Reducing resource overhead and improving application performance.

2. The V8 JavaScript Engine

- The **V8 engine** is an open-source JavaScript engine developed by **Google**, primarily used in **Chrome** and **Node.js**.
- It is optimized for high performance through **Just-In-Time (JIT) compilation**, which compiles JavaScript into **machine code** at runtime.
- Key features of V8 include:
  - **JIT Compilation**: Converts JavaScript into optimized machine code dynamically during execution.
  - **Garbage Collection**: Automatically manages memory, preventing leaks and maintaining performance.
  - **Efficient Execution**: Optimizes frequently used code paths to improve runtime speed.

3. Node.js Code Conversion: High-Level to Machine Code

- JavaScript is a **high-level interpreted language**, meaning it needs to be converted into **machine code** before execution by the CPU.
- The **V8 engine** manages this conversion through the following steps:
  i. **Parsing**:
     - V8 reads the JavaScript code and checks for **syntax errors**.
     - Converts the code into an **Abstract Syntax Tree (AST)**.
  ii. **Intermediate Representation (IR) Generation**:
     - The AST is transformed into an **Intermediate Representation (IR)**, a lower-level, platform-independent form of the code.
  iii. **Just-In-Time (JIT) Compilation**:
     - The IR is compiled into **machine code** dynamically at runtime.
     - V8 continuously **optimizes the machine code** based on runtime performance, ensuring efficient execution.
- This conversion process enables **Node.js applications** to run at high speed while maintaining flexibility and scalability.

Namaste Node.js - Episode 3 Summary

Getting Started with Node.js: From Installation to Writing Your First JavaScript Code

Downloading and Installing Node.js

Step 1: Download Node.js

1. Visit the official Node.js website.
2. You will see two versions available for download:
   - **LTS (Long-Term Support)**: Recommended for most users as it is more stable.
   - **Current**: Includes the latest features but may be less stable.
3. Click on the **LTS** version to download the installer.

Step 2: Install Node.js

1. Run the installer that you downloaded.
2. Follow the installation wizard:
   - Accept the license agreement.
   - Choose the installation path (default is usually fine).
   - Ensure that the option **"Add to PATH"** is checked (this allows you to use Node.js from the command line).
3. Complete the installation by clicking **Finish**.

Step 3: Verify Installation

1. Open **Command Prompt** (CMD) or **Terminal**.
2. Type the following command to check the installed Node.js version:

```
node -v
```

4.2. Writing Your First JavaScript Code in Visual Studio Code (VS Code)

Step 1: Install Visual Studio Code

1. Visit the Visual Studio Code website.
2. Download the installer for your operating system (Windows, macOS, or Linux).
3. Run the installer and follow the prompts to install VS Code.

Step 2: Open VS Code and Create a New File

1. Launch Visual Studio Code.
2. Open a new file by clicking on `File > New File` or use the shortcut `Ctrl + N`.
3. Save the file with a `.js` extension, for example, `app.js`, by clicking on `File > Save As`.

Step 3: Write Your First JavaScript Code

1. In the newly created `.js` file, type the following code:

```
console.log("Hello, World!");
```

2. Save the file.

Step 4: Run the Code in VS Code Terminal

1. Open the integrated terminal in VS Code by clicking on `View > Terminal` or using the shortcut `Ctrl + ``.
2. Ensure that you are in the correct directory where your `.js` file is saved.
3. Run the code by typing the following command in the terminal:

```
node app.js
```

4. You should see the output `Hello, World!` printed in the terminal.

Writing and Running JavaScript Code in Command Prompt (CMD)

Step 1: Open Command Prompt

1. Press `Win + R`, type `cmd`, and press `Enter` to open the Command Prompt.

Step 2: Navigate to Your Project Directory

1. Use the `cd` command to change the directory to where you want to save your JavaScript file:

```
cd path\to\your\directory
```

Step 3: Create a JavaScript File

1. Create a new JavaScript file using any text editor (like Notepad) or directly from CMD:

```
echo console.log("Hello, World!") > app.js
```

Step 4: Run the JavaScript Code

1. Execute the JavaScript file using Node.js by typing:

```
node app.js
```

2. The output `Hello, World!` should appear in the Command Prompt.

Namaste Node.js - Episode 4 Summary

Importing and Exporting in Node.js

Importing and Using

To import a function from a module, use the `require` function. Here's an example:

```
// In file app.js
const greet = require('./greet');
console.log(greet('World')); // Output: Hello, World!
```

Exporting Multiple Functions/Variables

To export multiple functions or variables, attach them to the `module.exports` object:

```
// In file utils.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;

module.exports = { add, subtract };
```

Importing Multiple Exports

To import multiple exports from a module, use destructuring:

```
// In file app.js
const { add, subtract } = require('./utils');
console.log(add(5, 3));      // Output: 8
console.log(subtract(5, 3)); // Output: 2
```

How `require` Works

When you use `require` to import a module, Node.js executes the code in the module file. Only the properties of `module.exports` are exposed to the importing file.
Example:

```
// In file data.js
const secret = 'hidden';
const publicData = 'visible';

module.exports = publicData;
// In file app.js
const data = require('./data');
console.log(data); // Output: visible
console.log(secret); // Error: secret is not defined
```

In the example above, `secret` is not accessible outside `data.js` because it was not exported using `module.exports`.

.mjs vs. .cjs Modules

.cjs Modules (CommonJS)

- **File Extension:** `.js` or `.cjs`
- **Module System:** CommonJS
- **Usage:** `require()` and `module.exports`

Example:

```
// In file common.js
module.exports = function() { /* ... */ };
```

.mjs Modules (ES Modules)

- **File Extension:** `.mjs`
- **Module System:** ES Modules (ESM)
- **Usage:** `import` and `export`

Example:

```
// In file module.mjs
export function greet(name) {
  return `Hello, ${name}!`;
}
```

Importing in .mjs

```
// In file app.mjs
import { greet } from './module.mjs';
console.log(greet('World')); // Output: Hello, World!
```

Differences Between .mjs and .cjs

- **Syntax:** `.cjs` uses `require` and `module.exports`, while `.mjs` uses `import` and `export`.
- **Compatibility:** `.mjs` is the standard ES Module syntax and is compatible with modern JavaScript, while `.cjs` is used for legacy Node.js modules.
- **Use Case:** Use `.mjs` for new projects or when using features of ES Modules, and `.cjs` for legacy code or when using the CommonJS module system.

Summary

- `module.exports` is used to export functions, objects, or variables from a module.
- `require` imports these exports into another module.
- `.cjs` modules use CommonJS syntax (`require` and `module.exports`), while `.mjs` modules use ES Module syntax (`import` and `export`).
- These kind of pattern used for import export in Nodejs

Episode-05 Diving into the NodeJS github repo

Understanding IIFE, Module Privacy, and `require` in Node.js

Overview

This repository provides an in-depth look into Immediately Invoked Function Expressions (IIFE) in Node.js, how module variables and functions can be kept private, and the detailed mechanism of the `require` statement in Node.js.

Immediately Invoked Function Expression (IIFE)

An Immediately Invoked Function Expression (IIFE) is a design pattern in JavaScript that executes a function immediately after its creation. It is often used to create a private scope and avoid polluting the global namespace.

Syntax

```
(function() {
    // Code inside here is private
})();
```

Parameters

IIFEs can accept parameters, allowing them to use values from the outer scope without exposing them:

```
(function(param) {
    console.log(param);
})(5); // Logs 5
```

In this example, `param` is only accessible within the IIFE and does not affect the outer scope.

Use in Node.js

In Node.js, IIFEs can be particularly useful for managing module scope and creating private variables and functions. Here are a few key points on how IIFEs are used in Node.js:

1. **Encapsulation**: IIFEs allow developers to encapsulate logic and variables within a module. This means that variables and functions defined inside the IIFE are not accessible from outside the module, helping to avoid potential conflicts and unintentional usage.

2. **Avoiding Globals**: By wrapping code in an IIFE, you prevent variables and functions from leaking into the global namespace. This is especially important in Node.js, where global scope pollution can lead to hard-to-debug issues.
3. **Creating Module-like Structures**: Although Node.js modules use CommonJS for encapsulation, IIFEs can still be useful for creating self-contained code blocks within modules or libraries.

Example in Node.js

Here is an example demonstrating how an IIFE can be used to create private variables and functions within a Node.js module:

```
// myModule.js
const myModule = (function() {
    const privateVariable = 'I am private';

    function privateFunction() {
        console.log('This is a private function');
    }

    return {
        publicFunction: function() {
            console.log('This is a public function');
            privateFunction(); // Accesses private function
        }
    };
})();

module.exports = myModule;
```

In this example:

- `privateVariable` and `privateFunction` are private to the IIFE and cannot be accessed from outside the module.
- `publicFunction` is exposed and accessible when the module is required elsewhere.

Module Privacy in Node.js

In Node.js, each module has its own scope, which means that variables and functions defined in a module are private to that module by default. You can control what is exposed to other modules using the `module.exports` or `exports` object.

Example

```
// myModule.js
const privateVariable = 'I am private';

function privateFunction() {
    console.log('This is a private function');
}
```

```
module.exports = {
    publicFunction: function() {
        console.log('This is a public function');
    }
};
```
In this example:

- `privateVariable` and `privateFunction` are private to `myModule.js`.
- Only `publicFunction` is exposed and accessible when the module is required elsewhere.

`require` Mechanism in Node.js

The `require` statement in Node.js is used to import modules. Here's a detailed five-step mechanism of how `require` works:

1. **Resolve**: Node.js determines the full path of the module. It first checks if the module is a core module (like `fs` or `path`). If not, it looks in `node_modules` directories, and if the module is a file, it uses the provided path.

2. **Load**: Node.js reads the file content and loads it into memory. If the file is in JavaScript, it will be treated as a script.

3. **Wrap**: Node.js wraps the module code in a function to provide local scope. This function looks like:

```
4. (function (exports, require, module, __filename, __dirname) {
5.     // Module code
   });
```

6. **Compile**: Node.js compiles the module code to machine code or JavaScript code (if using a JavaScript file).

7. **Execute**: The compiled code is executed within the context of the wrapped function. The module code is run, and `module.exports` is populated with the exported values.

Example

```
// index.js
const myModule = require('./myModule');

myModule.publicFunction(); // Logs: 'This is a public function'
```
In this example, `require('./myModule')` goes through the above five-step mechanism to load and execute `myModule.js`.

Conclusion

Understanding IIFE, module privacy, and the `require` mechanism is crucial for writing clean and maintainable code in Node.js. This repository aims to provide clarity on these concepts for better development practices.

- please star the Repository if you liked it and share with others:)

Episode-06 libuv and async IO

Understanding `Synchronous` and `Asynchronous`

Overview

libuv

Libuv is a library written in the programming language C that helps nodejs to improve efficiency while running tasks parallelly. However, nodejs already have async API's. It uses Libuvs's thread pools if async API is not available in nodejs and processes are blocking the operations. Libuv doesn't perform the task itself, it only manages the operations.

- Event-driven asynchronous I/O model is integrated.
- It allows the CPU and other resources to be used simultaneously while still performing I/O operations, thereby resulting in efficient use of resources and network.
- It facilitates an event-driven approach wherein I/O and other activities are performed using callback-based notifications.

Synchronous Javascript

In synchronous programming, operations are performed one after the other, in sequence. So, basically each line of code waits for the previous one to finish before proceeding to the next. This means that the program executes in a predictable, linear order, with each task being completed before the next one starts.

Asynchronous Javascript

Asynchronous programming, on the other hand, allows multiple tasks to run independently of each other. In asynchronous code, a task can be initiated, and while waiting for it to complete, other tasks can proceed. This non-blocking nature helps improve performance and responsiveness, especially in web applications.

Understanding Node.js: V8, libuv, and File Operations

Overview

Node.js leverages the V8 engine and libuv library to provide efficient handling of both synchronous and asynchronous operations. This document explains the fundamental concepts of V8, libuv, and their interactions with file operations, including correct and incorrect usages of `fs.readFileSync` and `fs.readFile`.

Key Concepts

V8 Engine

- **Purpose**: Executes JavaScript code.
- **Behavior**: Runs JavaScript synchronously in a single thread, processing one line of code at a time.

libuv Library

- **Purpose**: Manages asynchronous operations, including I/O tasks, timers, and more.
- **Behavior**: Handles non-blocking operations and manages the event loop, allowing Node.js to perform tasks asynchronously without blocking the main thread.

Code Breakdown

Incorrect Usage of `fs.readFileSync`

```
fs.readFileSync('./file.txt', 'utf-8', (err, data) => {
    console.log("File data fetched synchronously: ", data);
});
```

- **Issue**: `fs.readFileSync` is a synchronous method and does not accept a callback function.
- **Behavior**:
  - The file is read synchronously and its content is returned immediately.

- The callback provided is ignored.
- Data is not stored or printed because it is not captured.

Correct Usage of `fs.readFileSync`:

```
try {
    const dataSync = fs.readFileSync('./file.txt', 'utf-8');
    console.log("File data fetched synchronously: ", dataSync);
} catch (err) {
    console.error("Error reading file synchronously: ", err);
}
```

- **Explanation**:
  - `fs.readFileSync` reads the file synchronously, blocking the event loop until completion.
  - The content is stored in `dataSync` and logged.

Synchronous Operations

```
console.log("Hello! Async");

function mulFn(x, y) {
    const result = x * y;
    return result;
}

var a = 1078698;
var b = 20986;
var c = mulFn(a, b);
console.log("Multiplication result is: ", c);
```

- **Execution**:
  - `console.log` prints "Hello! Async" immediately.
  - `mulFn` calculates the product of `a` and `b` and logs the result.

Asynchronous Operations

`crypto.pbkdf2`

```
crypto.pbkdf2("myownpassword", "salt", 5000, 50, "sha512", (err, key) => {
    console.log("Key is generated: ", key);
    console.log("Key in hex format: ", key.toString('hex'));
    console.log("Key in base64 format: ", key.toString('base64'));
});
```

- **Execution**:
  - Asynchronous operation managed by libuv.
  - `crypto.pbkdf2` performs key derivation without blocking the main thread.
  - The callback is invoked once the key is generated.

`https.get`

```
https.get("https://dummyjson.com/products/1", (res) => {
    console.log("Fetched data successfully!");
});
```

- **Execution**:
  - Asynchronous HTTP request managed by libuv.
  - The request does not block the event loop.
  - The callback is called when the data is fetched.

setTimeout

```
setTimeout(() => {
    console.log("settimeout called after 5 sec");
}, 5000);
```

- **Execution**:
  - Asynchronous timer managed by libuv.
  - `setTimeout` schedules a callback to be executed after 5 seconds.
  - The event loop continues processing other tasks.

fs.readFile

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

- **Execution**:
  - Asynchronous file read managed by libuv.
  - The file is read without blocking the main thread.
  - The callback is executed when the file read operation is complete.

Example Code Explained

```
console.log("Hello world");
var a = 1078698;
var b = 20986;

// Runs immediately when the call stack of the main thread is empty
setTimeout(() => {
    console.log("call me ASAP");
}, 0);

function mulFn(x, y) {
    const result = x * y;
    return result;
}

var c = mulFn(a, b);
console.log("multiplication result is: ", c);
```

- **Explanation**:
  - `console.log("Hello world");` executes immediately, printing "Hello world".
  - `setTimeout(() => { console.log("call me ASAP"); }, 0);` schedules a callback to be executed as soon as the call stack is empty. Even though the

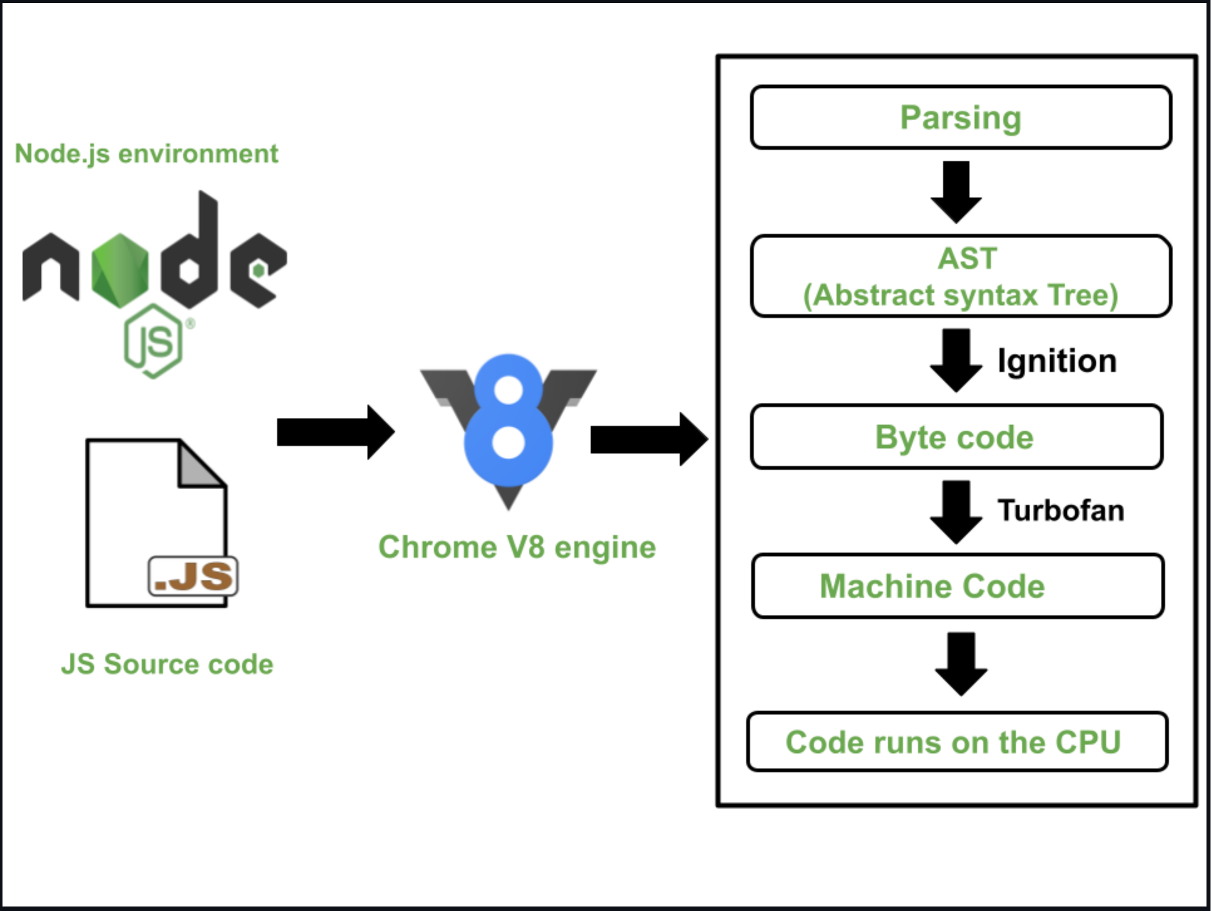
timeout is set to 0 milliseconds, the callback is placed in the event queue and will run after the current stack of synchronous code has completed.
   - `mulFn(x, y)` performs a synchronous multiplication of `a` and `b`, returning the result, which is then logged immediately.
   - The output of the `setTimeout` callback ("call me ASAP") appears after all the synchronous code has executed, illustrating the non-blocking nature of asynchronous operations.

Summary

- **Synchronous Code**: Operations like `readFileSync`, `console.log`, and `mulFn` execute sequentially and block the main thread.
- **Asynchronous Code**: Operations like `crypto.pbkdf2`, `https.get`, `setTimeout`, and `fs.readFile` are managed by libuv and do not block the event loop, allowing other operations to continue running.

The integration of V8 and libuv enables Node.js to handle multiple tasks efficiently, combining synchronous and asynchronous processing in a single-threaded environment. EOF

V8 JavaScript Engine: Code Execution Phases



1. Parsing Stage

- **Lexical Analysis:** V8 reads the JavaScript code and breaks it down into tokens, which are small chunks like keywords, operators, and identifiers.
- **Syntax Analysis:** The tokens are then arranged into a structure called an Abstract Syntax Tree (AST), which represents the code's structure and logic.

2. Ignition (Interpreter)

- **Bytecode Generation:** V8 converts the AST into bytecode, a simpler, intermediate form of the code that’s easier to run.
- **Execution:** The Ignition interpreter runs this bytecode directly, allowing the code to start executing quickly, but not yet fully optimized.

3. Profiling

- **Hotspot Detection:** As the code runs, V8 watches for parts of the code that are used a lot, called "hot" functions. It gathers data on these parts to decide if they should be optimized.

4. TurboFan (Optimizing Compiler)

- **Optimization:** For the frequently used "hot" code, V8 uses the TurboFan compiler to turn the bytecode into highly optimized machine code, making it run much faster.
- **Deoptimization:** If the assumptions used to optimize the code turn out to be wrong (like a variable type changing), V8 can revert the code back to a slower, but safer, execution mode.

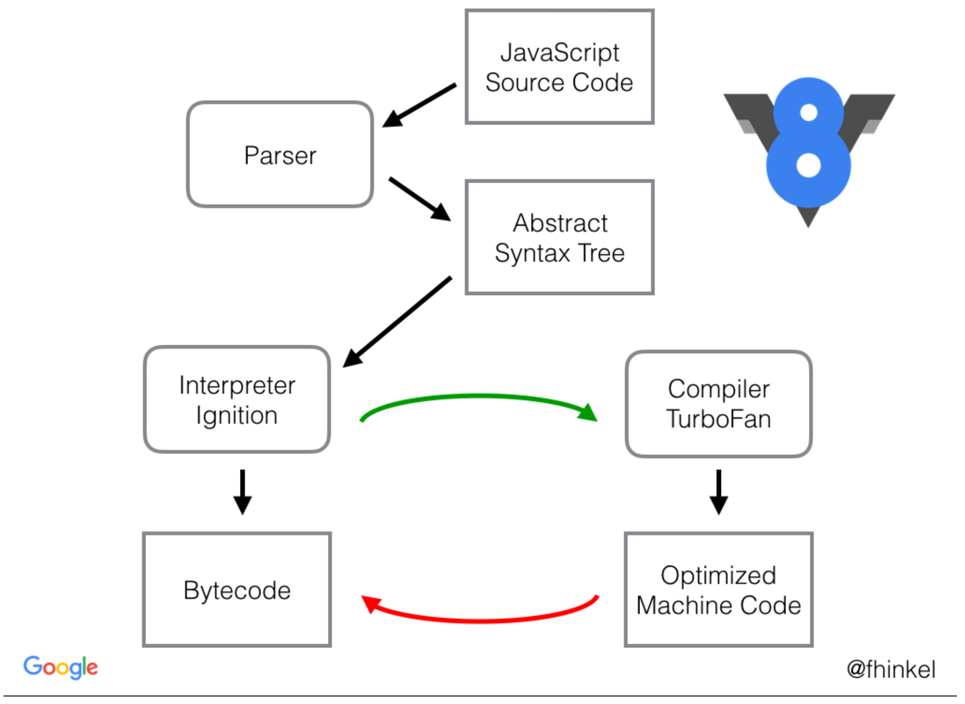
5. Garbage Collection

- **Memory Management:** V8 regularly cleans up memory by removing data that the program no longer needs, making sure that memory usage stays efficient.

6. Final Execution

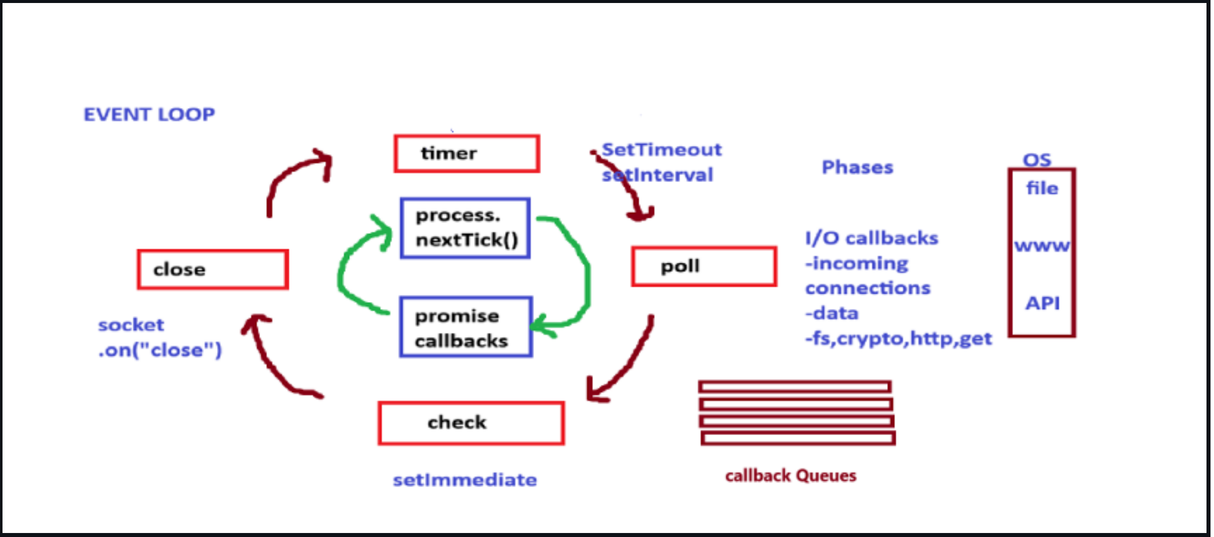
- **Execution:** The optimized code continues to run efficiently until the program finishes or until changes in the code require adjustments.

This step-by-step process ensures that JavaScript code is executed as quickly and efficiently as possible. """



Understanding libuv and event loop

The event loop in LIBUV operates in four major phases:



1 **Timers Phase:** In this phase, all callbacks that were set using `setTimeout` or `setInterval` are executed. These timers are checked, and if their time has expired, their corresponding callbacks are added to the callback queue for execution.
2 **Poll Phase:** After timers, the event loop enters the Poll phase, which is crucial because it handles I/O callbacks. For instance, when you perform a file read operation using `fs.readFile` , the callback associated with this I/O operation will be executed in this phase. The Poll phase is responsible for handling all I/Orelated tasks, making it one of the most important phases in the event loop.
3 **Check Phase:** Next is the Check phase, where callbacks scheduled by the `setImmediate` function are executed. This utility API allows you to execute callbacks immediately after the Poll phase, giving you more control over the order of operations.
4 **Close Callbacks Phase:** Finally, in the Close Callbacks phase, any callbacks associated with closing operations, such as socket closures, are handled. This phase is typically used for cleanup tasks, ensuring that resources are properly released.

- *Event Loop Cycle with `process.nextTick()` and `Promises`*

One More Important Note

When the event loop is empty and there are no more tasks to execute, it enters the `poll phase` and essentially waits for incoming events

Thread pool in libuv

Whenever there's an asynchronous task, V8 offloads it to libuv. For example, when reading a file, libuv uses one of the threads in its thread pool. The file system (fs) call is assigned to a thread in the pool, and that thread makes a request to the OS. While the file is being read, the thread in the pool is fully occupied and cannot perform any other tasks. Once the file reading is complete, the engaged thread is freed up and becomes available for other operations. For instance, if you're performing a cryptographic operation like hashing, it will be assigned to another thread. There are certain functions for which libuv uses the thread pool.

```
In Node.js, the default size of the thread pool is 4 threads:
    UV_THREADPOOL_SIZE=4
```

Q: Suppose you have a server with many incoming requests, and users are hitting APIs. Do these APIs use the thread pool?

A : No.

In the libuv library, when it interacts with the OS for networking tasks, it uses sockets. Networking operations occur through these sockets. Each socket has a socket descriptor, also known as a file descriptor (although this has nothing to do with the file system).

When an incoming request arrives on a socket, and you want to write data to this connection, it involves blocking operations. To handle this, a thread is created for each request. However, creating a separate thread for each connection is not practical, especially when dealing with thousands of requests.

Instead, the system uses efficient mechanisms provided by the OS, such as `epoll` (on Linux) or `kqueue` on macOS★. These mechanisms handle multiple file descriptors (sockets) without needing a thread per connection:

Here How it works:

- `epoll` (**Linux**) and `kqueue` (**macOS**) are notification mechanisms used to manage many connections efficiently.
- When you create an `epoll` or `kqueue` descriptor, it monitors multiple file descriptors (sockets) for activity.
- The OS kernel manages these mechanisms and notifies libuv of any changes or activity on the sockets.
- This approach allows the server to handle a large number of connections efficiently without creating a thread for each one.

The kernel-level mechanisms, like epoll and kqueue , provide a scalable way to manage multiple connections, significantly improving performance and resource utilization in a high-concurrency environment.

Important points to follow:

1 **DON'T BLOCK THE MAIN THREAD**

- Don't use `sync` methods
- Don't do operations on `heavy JSON Object` it will make load on main thread.
- Avoid complex `Regular Expression`.
- Avoid `Complex calculations` and big or `infine loops`.

2 **Data Structures is important**

- `epoll` - *Red Balck tree*
- `timers` -*min heap*

3 **Naming is very Important**

4 **There's is always lots to learn**

Creating a Server

Q: What is a server?

A: Understanding Servers: Hardware and Software

- What is `Server` ? The term "server" can refer to both hardware and software, depending on the context.
  - `Hardware` : A physical machine (computer) that provides resources and services to other computers (clients) over a network
  - `Software` : An application or program that handles requests and delivers data to clients.

- Deploying an Application on a Server
  - When someone says "deploy your app on a server," they usually mean:

1 **Hardware Aspect** : You need a physical machine (server) to run your application. This machine has a CPU, RAM, storage, etc.

2 **Operating System OS** : The server hardware runs an operating system like Linux or Windows. Your application runs on this OS.

3 **Server Software** : The software (e.g., a web server like Apache or an application server built with Node.js) that handles requests from users.

Client-Server Architecture

The term "client" refers to someone accessing a server. Imagine a user sitting at a computer wanting to access a file from a server. For this, the client needs to open a socket connection (not to be confused with WebSocket). Every client has an IP address, and every server has an IP address as well. The client could be a web browser.

To access the file, the client opens a socket connection. On the server side, there should be an application that is listening for such requests, retrieves the requested file, and sends it back to the client.

There can be multiple clients, and each client creates a socket connection to get data. After the data is received, the socket connection is closed. If the client needs to make another request, a new socket connection is created, data is retrieved, and the connection is closed again

Q: Can I create multiple servers?

A : Yes, you can create multiple HTTP servers.

Now, suppose a user is sending a request. How do we know which server it should go to?

When I mention creating an HTTP server, it means we are setting up two different Node.js applications. The distinction between these servers is defined by a port, which is a 4-digit number (e.g., port 3000).

For example, suppose an HTTP server with IP address `102.209.1.3`is running on port `3000` . This combination of IP address and port number (`102.209.1.3:3000`) indicates which specific HTTP server the request should be routed to.
Essentially, this means there’s a single computer (the server) that can run multiple applications, each with its internal servers. The port number determines which application or server the request is directed to.

Socket vs WebSockets

When a user makes a request to a website, a socket connection is established between the client and the server. This connection is typically used for a single request-response cycle: the client sends a request, the server processes it, sends back the response, and then the socket is closed. This process involves opening a new connection for each request.

On the other hand, WebSockets introduce a more efficient method by allowing the connection to remain open. This means that after the initial connection is established, it stays active, allowing for continuous communication between the client and server. Both the client and server can send and receive data at any time without the need to re-establish the connection. This persistent connection is ideal for real-time applications, where continuous interaction is required, such as in chat applications, online gaming, or live updates.

Creating a Server

```
localhost:999
const http = require("node:http");
const port = 999;
const server = http.createServer(function (req, res) {
    res.end("server Created")
})
server.listen(port, () => {
    console.log("Server running on port " + port)
})
```

Now, I want to handle different responses for the URL `localhost:3000/getsecretdata`

```
const http = require("node:http");
const port = 999;
const server = http.createServer(function (req, res) {
    if (req.url === "/getSecretData") {
        res.end("You are a human and the the secret so chill")
    }
    res.end("server Created")
})

server.listen(port, () => {
    console.log("Server running on port " + port)
})
```

```
We use Express to create a server. Express is a framework built on top of Node.js
that makes our lives easier.
```

Creating a Server Databases - SQL & NoSQL

Q: what is a database?

A: (Through Wikipedia) In computing, a **database is an organized collection of data or a type of data store based on the use of a **database management system** (**DBMS**), the software that interacts with end users, applications, and the database itself to capture and analyze the data. The DBMS additionally encompasses the core facilities provided to administer the database. The sum total of the database, the DBMS and the associated applications can be referred to as a **database system**. Often the term "database" is also used loosely to refer to any of the DBMS, the database system or an application associated with the database.

Types Of Databases:

1 Relational DB - `MySQL, PostgreSQL`

Relational databases like MySQL and PostgreSQL use structured tables with predefined schemas, making them ideal for handling complex queries and transactions. They ensure data integrity through ACID properties and are widely used for applications requiring robust, relational data models.

2 NoSQL DB -`MongoDB`
MongoDB is a NoSQL database that stores data in flexible, JSON-like documents, allowing for dynamic schemas. It's highly scalable and ideal for handling large volumes of unstructured or semi-structured data, making it popular for modern web applications

3 In-memory DB - `Redis`
Redis is an in-memory database known for its high-speed data processing capabilities. It supports various data structures like strings, hashes, and lists, making it suitable for caching, real-time analytics, and message brokering.

4 Distributed SQL DB - `Cockroach DB`
Cockroach DB is a distributed SQL database designed to scale horizontally across multiple nodes while providing strong consistency and ACID transactions. It's ideal for applications requiring high availability and resilience across different geographic locations.

5 Time Series DB - `Influx DB`
Influx DB is a time series database optimized for handling high write and query loads, particularly for time-stamped data. It's commonly used for monitoring, real-time analytics, and IoT applications where time-based data is crucial.

6 OO DB - `db4o`
db4o is an object-oriented database that stores data as objects, closely aligning with object-oriented programming languages. It simplifies development by allowing direct storage and retrieval of objects without the need for conversion to relational tables.

7 Graph DB: `Neo4j`
Neo4j is a graph database that excels at handling complex relationships between data entities. It uses a graph structure with nodes, relationships, and properties, making it ideal for applications like social networks, recommendation engines, and fraud detection.

8 Hierarchical DB - `IBM IMS`
IBM IMS is a hierarchical database that organizes data in a tree-like structure with parent-child relationships. It's used primarily in legacy systems for high performance transaction processing and is known for its reliability in handling large-scale, mission-critical applications.

9 Network DB - `IDMS`
IDMS (Integrated Database Management System) is a network database that represents data using a graph of record types and set relationships. It allows more complex relationships than hierarchical databases and is often used in legacy systems requiring high performance.

10 Cloud DB -`Amazon RDS`
Amazon RDS (Relational Database Service) is a managed cloud database service that supports multiple relational database engines, including MySQL, PostgreSQL, and Oracle. It automates tasks like backups, patching, and scaling, making it easy to deploy and manage databases in the cloud

Most commonly used databases are: 1 **Relational DB** 2 **NoSQL DB**

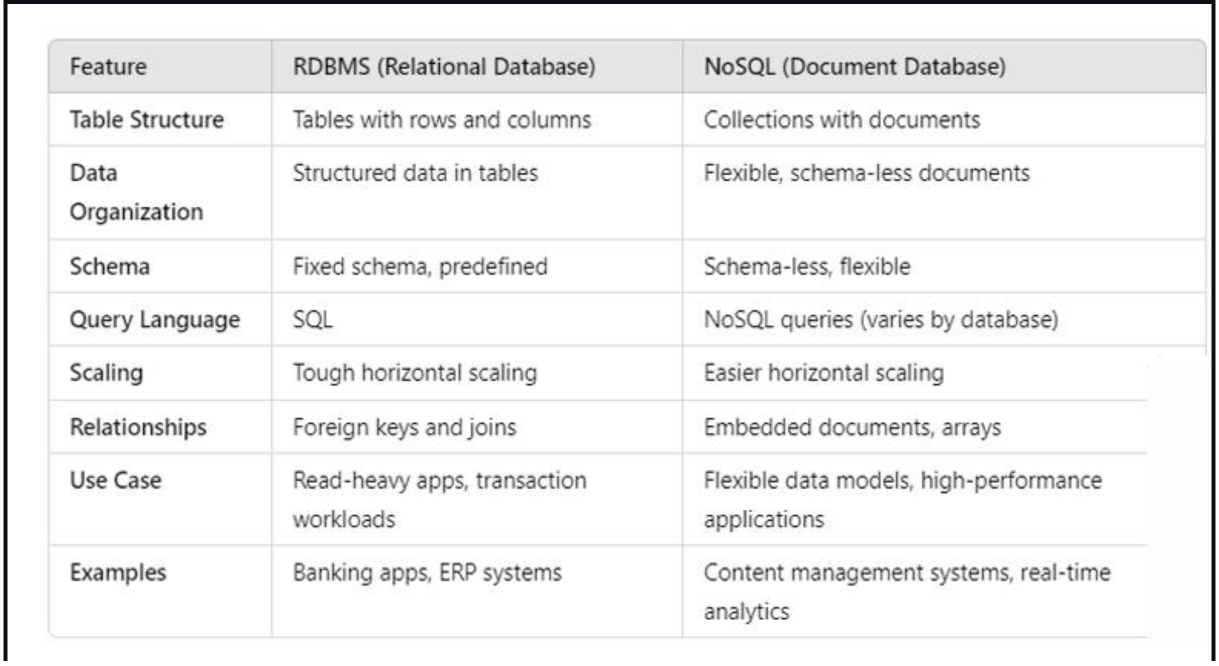
RDBMS (MySQL, PostgreSQL)

Relational Database Management Systems (RDBMS) like MySQL and PostgreSQL are popular choices for managing structured data.

NoSQL and MongoDB

NoSQL databases can be classified into four main types: 1 *Document Databases* 2 *Key-Value Databases* 3 *Graph Databases* 4 *Wide-Column Databases* 5 *Multi-Model Databases*

SQL vs NoSQL



Feature	RDBMS (Relational Database)	NoSQL (Document Database)
Table Structure	Tables with rows and columns	Collections with documents
Data Organization	Structured data in tables	Flexible, schema-less documents
Schema	Fixed schema, predefined	Schema-less, flexible
Query Language	SQL	NoSQL queries (varies by database)
Scaling	Tough horizontal scaling	Easier horizontal scaling
Relationships	Foreign keys and joins	Embedded documents, arrays
Use Case	Read-heavy apps, transaction workloads	Flexible data models, high-performance applications
Examples	Banking apps, ERP systems	Content management systems, real-time analytics

Creating a database & mongodb

MongoDB Setup and Connection Guide

This guide walks you through the steps to create a MongoDB Atlas cluster, set up a user, retrieve the connection string, and connect the database using MongoDB Compass.

1. Creating a MongoDB Atlas Account and Setting up a Cluster

Step 1: Go to MongoDB Website

1. Visit the MongoDB Atlas website.
2. Sign up or log in if you already have an account.

Step 2: Create a Free M0 Cluster

1. After signing in, go to the **Atlas** dashboard.
2. Click on the **Create a New Cluster** button.
3. Choose a free tier by selecting the **M0 Sandbox** cluster.
4. Choose a cloud provider (AWS, Google Cloud, or Azure) and a region (select a region close to your location for optimal performance).
5. Click **Create Cluster**.

This process may take a few minutes. MongoDB will notify you once the cluster is ready.

2. Creating a Database User

Step 3: Set Up a Database User

1. After the cluster is created, you will need to set up a user to access the database.
2. Navigate to the **Database Access** tab on the left side of the Atlas dashboard.
3. Click **Add New Database User**.
4. Enter a username and password for the new user. Keep these credentials secure, as you will need them to connect to the database.
5. Choose **Read and write to any database** as the role.
6. Click **Add User**.

3. Setting Network Access

Step 4: Configure Network Access

1. Navigate to the **Network Access** tab.
2. Click **Add IP Address**.
3. Select **Allow access from anywhere** if you want to access your database from any IP address, or add your specific IP address.
4. Click **Confirm**.

4. Getting the Connection String

Step 5: Retrieve the Connection String

1. Go to the **Clusters** tab on the left side.
2. Click on **Connect** next to your cluster.
3. Choose **Connect your application**.
4. Copy the connection string that looks something like this:

```
mongodb+srv://<username>:<password>@cluster0.mongodb.net/<dbname>?retryWrit
es=true&w=majority
```

- Replace `<username>` and `<password>` with the credentials of the user you created.
- Replace `<dbname>` with the name of the database you want to connect to.

5. Installing and Connecting MongoDB Compass

Step 6: Install MongoDB Compass

1. Download and install MongoDB Compass.
2. Once installed, open MongoDB Compass.

Step 7: Connect to Your Cluster

1. In MongoDB Compass, you will be prompted to enter the connection string.
2. Paste the connection string you copied earlier from MongoDB Atlas.
3. Replace `<password>` with the password of the user you created.
4. Click **Connect**.

You should now be successfully connected to your MongoDB cluster and able to manage your database locally using MongoDB Compass.

6. Verifying Connection

Step 8: Check Connection in MongoDB Compass

1. Once connected, MongoDB Compass will show a list of your databases in the cluster.
2. Click on your database to manage collections, documents, and perform queries.
3. You can create new databases or collections using the **New Database** button.

7. Creating and Managing Databases (Optional)

Step 9: Create a New Database

1. After connecting, click on the **Databases** tab in MongoDB Compass.
2. Click **Create Database**.
3. Enter a **Database Name** and **Collection Name** for your new collection (table).
4. Click **Create Database** to create a new database and collection in your MongoDB cluster.

Step 10: Managing Collections

1. Once inside a database, you can add, delete, or update documents (records) inside a collection.
2. To add a document, click **Insert Document**.
3. Add your data in JSON format and click **Insert**.

8. Troubleshooting

Step 11: Handling Connection Issues

- **Whitelist IP Address**: Ensure that your current IP address is whitelisted in **Network Access** settings in MongoDB Atlas.
- **User Authentication**: Make sure the correct username and password are used in the connection string.
- **Connection Timeout**: Check your internet connection or cluster region if the connection is timing out.

Step 12: Resetting MongoDB Compass Configuration (if needed)

1. If you face persistent connection issues, try resetting MongoDB Compass settings.

2. Go to **Settings → Reset Compass** to return to the default configuration.

Conclusion

By following this guide, you have successfully:

- Installed MongoDB Compass.
- Connected it to your MongoDB Atlas cluster.
- Verified the connection and managed databases.
- Handled basic troubleshooting issues.

Now, you are ready to start working with MongoDB locally or integrate it into your applications.

Database Connection and MongoDB CRUD Operations Overview

This document covers how to connect to a MongoDB server and perform CRUD (Create, Read, Update, Delete) operations using the MongoDB Node.js library.

1. Connecting to the Database Server

To connect to a MongoDB, first install the MongoDB Node.js driver:

```
npm install mongodb
```

Database Connection

```
const { MongoClient } = require('mongodb');
// Connection URL
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

// Database Name
const dbName = 'Namaste-Nodejs';

async function main() {
  await client.connect();
  console.log('Connected successfully to server');
  const db = client.db(dbName);
  const collection = db.collection('User');

  return 'done.';
}
main()
  .then(console.log)
  .catch(console.error)
  .finally(() => client.close());
```

2. Introduction to CRUD

CRUD operations are fundamental to interacting with databases. Each operation corresponds to a specific action that can be performed on data stored in the database. Here’s a brief overview of each operation:

1.1 Create

- **Definition**: The Create operation adds new documents to a MongoDB collection.
- **Purpose**: Used to insert new records or entries into a database.
- **Example Use Cases**: Adding a new user profile, creating a new product listing, or recording a new transaction.

```
const  data  = {
firstname:  "Akshad",
lastname:  "Jaiswal",
city:  "Pune",
phoneNumber:  "88526587",
}
//Create
const insertData = await collection.insertMany([data])
console.log("data inserted = ", insertData)
```

1.2 Read

- **Definition**: The Read operation retrieves documents from a MongoDB collection.
- **Purpose**: Used to query and obtain data stored in the database.
- **Example Use Cases**: Fetching user details, listing products, or generating reports based on stored data.

```
//Read
const  findData  =  await  collection.find({}).toArray();
console.log("All data :", findData)
```

1.3 Update

- **Definition**: The Update operation modifies existing documents in a MongoDB collection.
- **Purpose**: Used to change the values of specific fields in existing documents.
- **Example Use Cases**: Updating user information (like email or password), changing the status of an order, or modifying product details.

```
// Update
const updateData = await collection.updateOne({ _id: new
ObjectId('67066d6a3be8f41630d5dae4') }, { $set: { firstname: "Mint" } })
console.log("Updated document ", updateData)
```

1.4 Delete

- **Definition**: The Delete operation removes documents from a MongoDB collection.
- **Purpose**: Used to permanently delete records that are no longer needed.
- **Example Use Cases**: Deleting a user account, removing a product that is out of stock, or purging outdated transaction records.

```
//delete
const deletedata = await collection.deleteOne({ _id: new
ObjectId('670668562c6bd11e25050c13') })
console.log("deleted data=>", deletedata)
```

3. Summary of CRUD Operations

Operation	Description	Example Use Case
Create	Adds new documents to a collection	Adding a new user profile
Read	Retrieves documents from a collection	Fetching user details
Update	Modifies existing documents in a collection	Updating user information
Delete	Removes documents from a collection	Deleting a user account

4. Conclusion

Understanding CRUD operations is essential for effectively managing data in a MongoDB database. These operations enable users to create, retrieve, modify, and delete records, allowing for comprehensive data manipulation and management. Mastery of these operations forms the foundation for building applications that rely on database interactions.

Microservices vs Monolith - How to build a project

Software Project Development in the Industry

Waterfall Model Overview

The Waterfall Model is a sequential software development process that consists of a series of steps. Each step must be completed before moving on to the next. This model is one of the traditional approaches used in the software industry for project development.

Steps in the Waterfall Model

1. **Requirement**
   - Gather and analyze all project requirements.
   - Document the functional and non-functional requirements.
   - Stakeholders provide input to ensure a clear understanding of the project's needs.
   - **Roles Involved**: Business Analysts, Project Managers, Stakeholders, Product Owners
2. **Design**
   - Create a system and software design based on the requirements.
   - Architectural design: High-level system structure.
   - Detailed design: Specifies how each component will work.
   - **Roles Involved**: Solution Architects, UX/UI Designers, System Designers, Technical Leads
3. **Development**
   - Actual coding of the software begins based on the design documents.
   - Development teams implement functionalities and integrate different modules.
   - **Roles Involved**: Software Developers, Backend/Frontend Developers, Database Administrators, DevOps Engineers
4. **Testing**
   - Perform testing to identify defects and bugs.
   - Different testing methods include unit testing, integration testing, system testing, and acceptance testing.

    - **Roles Involved**: Quality Assurance (QA) Engineers, Testers, Test Leads, Automation Engineers

5. **Deployment**
    - Once testing is successful, the software is deployed to a live environment.
    - The deployment phase includes user training and system configuration.
    - **Roles Involved**: DevOps Engineers, System Administrators, Release Managers, IT Support

6. **Maintenance**
    - Ongoing support for the software post-deployment.
    - Fixing any issues that arise and implementing necessary updates or enhancements.
    - **Roles Involved**: Support Engineers, Maintenance Team, Developers, IT Support, Customer Support

Project Building Strategies

Overview

There are two main architectural approaches to building software projects: Monolith and Microservices. Each has its own characteristics, advantages, and challenges. Understanding the parameters that differentiate these architectures is essential for selecting the right approach for a given project.

1. Monolith Architecture

- A single unified codebase where all components are interconnected.
- Easier to deploy since the entire application is packaged and released as a single unit.
- Suitable for smaller projects or applications with tightly coupled components.

2. Microservices Architecture

- A distributed system where different functionalities are separated into individual services.
- Each service can be developed, deployed, and scaled independently.
- Ideal for large, complex projects where different teams handle different services.

Comparison Parameters

Parameter	Monolith Architecture	Microservices Architecture
Development Speed	Faster for small projects; a single codebase is easy to manage.	May be slower initially due to service setup and communication.
Code Repo	Single code repository for the entire project.	Multiple repositories for individual services.
Scalability	Limited to scaling the entire application.	Fine-grained scaling; each service can be scaled independently.
Tech Stack	Typically a unified stack across the project.	Allows different tech stacks for different services.
Infra Cost	Lower for small projects with simpler requirements.	Higher due to separate services and infrastructure overhead.
Complexity	Simpler for smaller projects but grows complex with size.	Higher complexity due to distributed nature and inter-service communication.
Fault Isolation	Failures can affect the entire application.	Better fault isolation; issues in one service do not impact others.
Testing	Easier to perform end-to-end testing in a single environment.	Requires testing multiple independent services and integration.
Ownership	Centralized; a single team usually manages the entire application.	Distributed; different teams can own different services.
Maintenance	Easier for small projects but harder as the project grows.	More manageable for large projects with well-defined services.
Revamps	Difficult to change or refactor large monoliths.	Easier to revamp individual services without affecting others.
Debugging	Easier in a single codebase but can be challenging for large apps.	More difficult due to distributed logging and monitoring.

Parameter	Monolith Architecture	Microservices Architecture
Developer Experience	Easier for small teams working on a single codebase.	Better for large teams as they can work independently on different services.

Conclusion

Choosing between Monolith and Microservices depends on project size, team structure, and specific requirements. Monolith architecture is simpler for smaller projects, while Microservices offer greater flexibility and scalability for larger, complex projects.

DevTinder Project Development (LLD HLD)

Development Approach

The DevTinder app is being built by following a structured development cycle similar to how projects are managed in a company. The process includes gathering requirements, high-level design (HLD), and low-level design (LLD).

1. Requirements Gathering

- **Understanding the Project**:
    - **Project Name**: DevTinder
    - **Concept**: A platform similar to Tinder, but specifically designed for developers to connect and collaborate.
- **Features**:
    - **User Account Management**:
        - Create an account.
        - Signup and login functionality.
        - Update user profile.
    - **Developer Exploration**:
        - Feed page to explore developer profiles.
        - Ability to send connection requests.
    - **Connections Management**:
        - View matches (mutual connections).
        - List of sent and received requests.
        - Update profile information.
    - **Additional Features**:
        - More features to be added according to development needs.

2. High-Level Design (HLD)

- **Tech Planning**:
    - **Architecture**: The app will follow a microservices architecture with separate services for the frontend and backend.
    - **Tech Stack**:
        - **Frontend**: React.js
        - **Backend**: Node.js
        - **Database**: MongoDB

- **Development Team Roles**:
  - Once the features and tech planning are finalized, the software development team comes into the picture, including SDE1, SDE2, and backend teams.
- **Note**:
  - Proper planning is essential, as it makes the development process smoother and coding easier.

3. Low-Level Design (LLD)

- **Database Design**:
  - **Collections**:
    a. **User**:
       - Fields: `firstname`, `lastname`, `email`, `password`, `age`, `gender`, etc.

    b. **ConnectionRequest Collection**:
       - Fields: `fromUserId` (sender's user ID), `toUserId` (receiver's user ID), `status` (e.g., pending, accepted, rejected, ignored).

API Design

What are REST APIs?

- **REST (Representational State Transfer)** is an architectural style for designing networked applications. It relies on a stateless, client-server communication protocol, most commonly HTTP.
- **REST APIs** are web services that allow for interaction between a client and a server using standard HTTP methods. They enable the creation, retrieval, updating, and deletion of resources through well-defined endpoints.

How REST APIs Work

1. **Client Request**:
   - The client sends a request to the server via HTTP methods such as GET, POST, PUT, PATCH, or DELETE.
   - Each HTTP method corresponds to a specific operation:
     - **GET**: Retrieve data from the server.
     - **POST**: Create new data on the server.
     - **PUT**: Update existing data (entirely replace the resource).
     - **PATCH**: Partially update existing data.
     - **DELETE**: Remove data from the server.
2. **Server Response**:

   - The server processes the request and returns a response.
   - The response usually includes a status code (e.g., 200 for success, 404 for not found) and, if applicable, data in JSON or XML format.

3. **Stateless Communication**:
   - Each request from the client to the server must contain all the information needed to understand and process the request.
   - The server does not store any state about the client session between requests.

Difference Between PUT and PATCH

- **PUT**: Updates the entire resource with new data. If any fields are missing, they may be reset or removed.
- **PATCH**: Only updates specific fields of the resource, leaving the other fields unchanged.

REST APIs Needed

1. **User Management APIs**:
   - `POST /signup`: Register a new user
   - `POST /login`: Authenticate and login the user
   - `POST /profile`: Create a user profile
   - `GET /profile`: Retrieve user profile details
   - `PATCH /profile`: Update user profile
   - `DELETE /profile`: Delete user profile

2. **Connection Management APIs**:
   - `POST /sendRequest`: Send a connection request (ignore/interested)
   - `POST /reviewRequest`: Accept or reject a connection request
   - `GET /request`: Retrieve the list of requests (sent and received)
   - `GET /connections`: Get a list of established connections

Next Steps

1. Finalize the database schema and API design.
2. Implement the backend services for the listed APIs.
3. Develop the frontend to interact with these backend services.
4. Test the application for feature completion and bug fixes.

DevTinder Project

Overview

DevTinder is a MERN (MongoDB, Express, React, Node.js) application designed to help developers connect and collaborate. The project uses a microservices architecture, which divides the application into two main services:

1. **Frontend**: Handles the user interface and client-side logic.
2. **Backend**: Manages server-side logic, APIs, and interactions with the database.

DevTinder Backend

Repository Setup

- Created a new repository for the backend service: DevTinder Backend GitHub Repository
- Initialized the repository using `npm init`, which generated a `package.json` file for managing project dependencies and configurations.

Learning Journey

1. What is Express Framework?

- **Express** is a minimal, flexible, and robust web application framework for Node.js that simplifies server-side development.
- Explored the official website to understand its purpose and features: Express.js Official Website

2. Installing Express

- Installed Express in the project using npm:

```
npm install express
```

- This added the Express framework to the package.json file under dependencies and created the node_modules folder to store the installed packages.

3. Understanding Key Files and Folders in a Node.js Project

When working on a Node.js project, several important files and folders are created to manage dependencies, configurations, and project structure. Here's a breakdown of these key components:

1. `node_modules`

- **Description**:
  - The `node_modules` folder contains all the installed npm packages and their dependencies.
  - It is automatically generated when packages are installed using npm.
- **Purpose**:
  - Stores all the libraries and dependencies required by the project.
  - Can be large because it includes every package and sub-dependency specified in the project.
- **Note**:
  - You typically don't need to manually edit this folder.
  - If deleted, it can be recreated by running `npm install`.

2. `package.json`

- **Description**:
  - The `package.json` file acts as the manifest file for a Node.js project, containing metadata such as project name, version, description, author, and dependencies.
- **Key Fields**:
  - `"name"`: The name of the project.
  - `"version"`: The current version of the project.
  - `"description"`: A brief description of the project.
  - `"dependencies"`: Lists npm packages required for the project.
  - `"devDependencies"`: Lists packages needed for development but not in production.
  - `"scripts"`: Defines custom npm commands for running tasks (e.g., `"start": "node app.js"`).
- **Common Commands**:
  - `npm init`: Creates a new `package.json` file.
  - `npm install <package>`: Adds a package to the project's dependencies.
  - `npm install <package> --save-dev`: Adds a package to `devDependencies`.

3. `package-lock.json`

- **Description**:
  - The `package-lock.json` file is automatically generated when npm modifies the `node_modules` tree or `package.json`.
  - Ensures consistent installation of dependencies by locking the versions.

- **Purpose**:
  - Prevents issues caused by updates to dependencies or sub-dependencies.
  - Provides a detailed description of the dependency tree and specific versions installed.

4. Other Common Files

- **`.gitignore`**:
  - Specifies files and directories that should be ignored by Git. Commonly used to exclude `node_modules` and other generated files.
- **`README.md`**:
  - A markdown file that serves as documentation for the project, typically including instructions for installation, usage, and contribution.

server

Created the basic server using Express

```
const express = require("express")
const app = express();
const port = 3000;

app.use("/test", (req, res) => {
    res.send("Server started ")
})

app.use("/main", (req, res) => {
    res.send("another route")
})

app.listen(port, () => {
    console.log("Server started running on port " + port)
})
```

Understanding the `-g` Flag in `npm install`

What is `-g` in `npm install`?

The `-g` flag stands for "global" and is used with the `npm install` command to install packages globally on your system, rather than locally within a specific project.

Local vs. Global Installation

Local Installation (Default)

- When you run `npm install <package>`, the package is installed locally in the `node_modules` directory of the current project.

- The package is only accessible within that project and is added to the project's `package.json` dependencies.

Global Installation (`-g` Flag)

- When you run `npm install -g <package>`, the package is installed globally on your system.
- This makes the package accessible from the command line in any directory.
- Global installation is typically used for packages that provide command-line tools (e.g., `npm`, `nodemon`, `eslint`).

Example Usage

To install a package globally:

```
npm install -g <package>
npm install -g nodemon
```

Routing and Request Handlers

DevTinder Backend

Project Overview

DevTinder is a MERN application for developers to connect and collaborate. This repository contains the backend code for the project.

Repository Link

- DevTinder Backend Repository

HTTP Methods

1. POST

- Used to create a new resource
- Request body contains the data to be created
- Example: Creating a new user account

2. GET

- Used to retrieve a resource
- Request query parameters can be used to filter or sort data
- Example: Retrieving a list of users

3. PATCH

- Used to partially update a resource
- Request body contains the changes to be made
- Example: Updating a user's profile information

4. DELETE

- Used to delete a resource
- Example: Deleting a user account

5. PUT

- Used to replace a resource entirely
- Request body contains the new data
- Example: Updating a user's entire profile information

Notes

- HTTP methods can be used to perform CRUD (Create, Read, Update, Delete) operations on resources
- Understanding the differences between these methods is crucial for building a robust and scalable backend

API Testing with Postman

What is Postman?

- Postman is a popular API testing tool that allows you to send HTTP requests and view responses in a user-friendly interface.

Why use Postman?

- Postman provides an easy way to test and debug APIs, making it an essential tool for backend development.

How to use Postman?

- Download and install Postman from the official website
- Create a new request by selecting the HTTP method (e.g., GET, POST, PUT, DELETE) and entering the API endpoint URL
- Add request headers, query parameters, and body data as needed
- Send the request and view the response in the Postman interface

Advanced Routing in Node.js

Overview

Routing in Node.js allows you to define how the server responds to various HTTP requests. Advanced routing techniques can be used to create dynamic and flexible routes by using special characters like `+`, `?`, `*`, and regular expressions.

Special Characters in Routing

1. `+` (Plus)

- The `+` character matches one or more occurrences of the preceding character.
- Example:

```
app.get('/ab+c', (req, res) => {
  res.send('Route matched: /ab+c');
});
```

The route /ab+c would match:

- /abc
- /abbc
- /abbbc, and so on.

`?` (Question Mark)

- The ? character makes the preceding character optional in an Express route pattern.
- Example:

```
app.get('/ab?c', (req, res) => {
  res.send('Route matched: /ab?c');
});
```

This route will match:

- /abc
- /ac (since b is optional).

* (Asterisk)

- The * character matches any sequence of characters in an Express route.
- Example:

```
app.get('/a*cd', (req, res) => {
  res.send('Route matched: /a*cd');
});
```

This route will match:

- /acd
- /abcd
- /axyzcd, etc.

Regular Expressions

- Regular expressions (regex) can be used in Express routing to match complex patterns.
- Examples:

```
app.get(/a/, (req, res) => {
  res.send('Route matched any path containing "a"');
});
```

This route will match:

- /abc
- /a123
- /123a, etc.

DevTinder Project - Learning Notes

Code Demonstration Link

- DevTinder Backend Repository

Middlewares and Error Handlers in Express

Route Handlers in Express

Overview

In Express, route handlers are functions that handle requests to a specific endpoint. You can use multiple route handlers for a single route, control the flow with `next`, and even wrap handlers into arrays for better modularity.

1. Multiple Route Handlers

- Express allows defining multiple route handlers for a single route. Each handler can execute different logic or perform different tasks before sending a response.
- Example:

```
app.get('/example', (req, res, next) => {
  console.log('First handler');
  next(); // Pass control to the next handler
}, (req, res) => {
  res.send('Second handler');
});
```

- In the above example:
- The first handler logs a message and then calls next().
- The second handler sends a response after the first one completes

Understanding `next` and `next()` in Express

Overview

In Express, `next` is a callback function that allows you to control the flow of middleware functions and route handlers. It helps in moving the request to the next middleware or route handler in the stack.

1. What is `next`?

- `next` is a function provided by Express, used to pass control to the next middleware function or route handler.
- It must be called within a middleware function for the request to proceed further.
- If not called, the request will be left hanging, and the server won't send a response.

2. How to Use `next()`

- **Basic Usage**:

```
app.get('/example', (req, res, next) => {
  console.log('First handler');
  next(); // Passes control to the next middleware function or route handler
}, (req, res) => {
  res.send('Second handler');
});
```

- In the above example:
- The first function logs a message and then calls next() to proceed to the next handler.
- The second function sends a response after the first handler completes

3. Using `next()` to Skip Route Handlers in Express

In Express, you can use the `next()` function to pass control to the next middleware function or route handler. By passing the string `'route'` as an argument to `next()`, you can skip the remaining route handlers for a particular route.

How to Use `next('route')` to Skip Handlers

- When `next('route')` is called, Express will skip the remaining handlers for the current route and move on to the next matching route handler.
- This is useful when certain conditions are met, and you want to bypass specific middleware or handlers.
- **Basic Usage**:

```
app.get('/skip', (req, res, next) => {
  console.log('This handler will be skipped');
  next('route'); // Skips to the next matching route handler
}, (req, res) => {
  res.send('You will not see this response because the handler is skipped');
});

// Next matching route handler
app.get('/skip', (req, res) => {
  res.send('Skipped to this route handler');
});
```

- Here, the second handler will be skipped, and the request will be passed directly to the third handler.

Middlewares in Express.js

1. What is Middleware?

- **Middleware** is a function that has access to the request (`req`), response (`res`), and the next middleware function in the request-response cycle.
- Middleware functions can:
  - Execute any code.
  - Modify the request and response objects.
  - End the request-response cycle by sending a response.
  - Call the next middleware function in the stack using `next()`.

2. Why Do We Need Middleware?

- **Modularity**: Middleware helps in separating concerns like authentication, logging, validation, etc., into reusable functions.
- **Pre-processing**: Middleware can be used to modify or check the request before it reaches the route handler.
- **Error Handling**: Middleware is essential for catching errors and handling them gracefully without stopping the application.
- **Authorization/Authentication**: Middleware ensures that only authorized users can access certain routes.
- **Request Logging**: Middleware can log request details for monitoring or debugging.

3. How Express.js Handles Middlewares Behind the Scenes

- When an HTTP request is received, Express executes all middleware functions in the order they are defined.
- Each middleware function can:
  - **Pass control** to the next middleware by calling `next()`.
  - **End the request-response cycle** by sending a response.
- Middleware functions are executed sequentially unless `next()` is invoked, which passes control to the next middleware or route handler.
- Behind the scenes, Express creates a **middleware stack** and processes it in order. If `next()` is not called, the request gets stuck and no further processing occurs.

Example of Middleware Flow

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

```
// Middleware 1: Request Logger
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to the next middleware
});
// Middleware 2: Authorization Check
//Handle user authentication for all admin routes using middlewares
app.use("/admin", (req, res, next) => {
    const token = "999";
    const isAuthorizedAdmin = token === "999";
    if (!isAuthorizedAdmin) {
        res.status(401).send("Unauthorized Admin")
    } else {
        next();
    }
})
app.get("/admin/getAllData", (req, res) => {
    res.send("All data Generated")
})
app.get("/admin/deleteData", (req, res) => {
    res.send("Data Deleted")
})
app.listen(3000, () => console.log('Server is running on port 3000'));
```

HTTP Status Codes

HTTP status codes are standard response codes returned by web servers to indicate the result of a client's HTTP request. These codes help both the client and server understand what happened with the request and whether it was successful or encountered an error.

Categories of HTTP Status Codes:

- **1xx Informational**: The request was received, and the process is continuing.
- **2xx Success**: The request was successfully received, understood, and accepted.
- **3xx Redirection**: Further action is required to complete the request.
- **4xx Client Error**: The request contains bad syntax or cannot be fulfilled.
- **5xx Server Error**: The server failed to fulfill a valid request.

Common HTTP Status Codes

1xx Informational

- **100 Continue**: The server has received the request headers, and the client should proceed to send the request body.

2xx Success

- **200 OK**: The request was successful, and the server responded with the requested data.
- **201 Created**: The request was successful, and a new resource was created.
- **204 No Content**: The request was successful, but there is no content to send in the response.

3xx Redirection

- **301 Moved Permanently**: The resource has been permanently moved to a new URL. All future requests should use the new URL.
- **302 Found**: The resource has been temporarily moved to a different URL, but future requests should still use the original URL.
- **304 Not Modified**: The resource has not been modified since the last request, so the client can use the cached version.

4xx Client Error

- **400 Bad Request**: The server could not understand the request due to invalid syntax.
- **401 Unauthorized**: The client must authenticate itself to get the requested response.
- **403 Forbidden**: The client does not have permission to access the requested resource.
- **404 Not Found**: The server cannot find the requested resource. This usually occurs when the URL is incorrect.

5xx Server Error

- **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.
- **502 Bad Gateway**: The server, acting as a gateway, received an invalid response from the upstream server.
- **503 Service Unavailable**: The server is currently unavailable, usually due to being overloaded or down for maintenance.

How to Use HTTP Status Codes in Express.js

In Express, you can send status codes using `res.status()` followed by the appropriate code:

```
app.get('/example', (req, res) => {
  res.status(200).send('Success');
});
```

```
app.get('/error', (req, res) => {
  res.status(404).send('Not Found');
});
```

Difference Between `app.use()` and `app.all()` in Express.js

Feature	`app.use()`	`app.all()`
Purpose	Mounts middleware functions or sub-routers to all or specific routes	Matches all HTTP methods (GET, POST, PUT, DELETE, etc.) on a specific route
Path Requirement	Can be used with or without a path	Requires a specific path
Applies to	All HTTP methods by default	All HTTP methods but only for the defined path
Common Use Case	Applying middleware logic across multiple routes or specific paths	Handling all HTTP methods (GET, POST, etc.) on one route
Functionality	Middleware is invoked sequentially until the next middleware or route handler is reached	Executes for any HTTP method (GET, POST, PUT, etc.) on the specified path

Examples

1. `app.use()` Example

```
// Middleware applied to all routes
app.use((req, res, next) => {
  console.log('Request received');
  next();
});

// Middleware applied to a specific path
app.use('/user', (req, res, next) => {
  console.log('User path accessed');
  next();
});
```

2. `app.all()` Example

```
// Match all HTTP methods on '/about' route
app.all('/about', (req, res) => {
  res.send('This route handles all HTTP methods');
});
```

Error-Handling Middleware in Express.js

Error-handling middleware in Express is used to catch and manage errors that occur during the processing of requests. It allows the application to respond with appropriate error messages and status codes.

Defining Error-Handling Middleware

- Error-handling middleware is defined with **four** parameters: `(err, req, res, next)`.
- Express identifies it as an error handler because it includes the `err` parameter as the first argument.

Example of Error-Handling Middleware

```
// Define error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error details
  res.status(500).send('Something went wrong!'); // Send a 500 Internal Server
Error response
});
```

Databse, Schema & Models | Mongoose

Code Demonstration Link

- DevTinder Backend Repository

1. Database Connection

This guide demonstrates how to connect to MongoDB directly using a connection string URL without using a `.env` file. This setup is convenient for testing but is not secure for production as it exposes sensitive information.

1. Install Mongoose

Make sure Mongoose is installed in your project:

```
npm install mongoose
```

2. Database Connection code:

```
const mongoose = require('mongoose');
const express = require('express');
const app = express();
const PORT = 3000;

// MongoDB connection string URL
const databaseUrl =
'mongodb+srv://<username>:<password>@cluster0.mongodb.net/myDatabase?retryWrites=t
rue&w=majority';

// Connect to MongoDB using Mongoose
mongoose.connect(databaseUrl, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
  .then(() => {
    console.log('Connected to MongoDB');
    // Start the server only after database connection is successful
    app.listen(PORT, () => console.log(`Server running on
http://localhost:${PORT}`));
  })
  .catch((err) => {
    console.error('Database connection error:', err);
  });
```

Important Note: Always ensure that the database connection is established before starting the server. This setup is best for development or testing but not for production, as it can expose credentials.

2. Database Schema in Mongoose

A **schema** in Mongoose defines the structure of documents in a MongoDB collection, including fields, data types, and validation. Using schemas allows for structured data models, making it easier to handle data validation and consistency within MongoDB.

What is a Database Schema?

- A Mongoose schema outlines the fields and their types for a document in a MongoDB collection.
- Schemas allow you to apply constraints, set defaults, and define validation rules for each field.
- Mongoose schemas enable structured, schema-driven data handling similar to traditional SQL databases.

Example: Creating a User Schema

Below is an example of defining a simple user schema using Mongoose:

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

// Define the User schema
const userSchema = new mongoose.Schema({
  firstname: { type: String, required: true },
  lastname: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 0 },
  gender: { type: String, enum: ['Male', 'Female', 'Other'] }
});

// Create a User model from the schema
const User = mongoose.model('User', userSchema);

module.exports = User;
```

3. Saving a Document with a Schema in Mongoose

Once a schema is defined in Mongoose, you can create and save documents to MongoDB based on that schema. Mongoose provides a straightforward way to add documents with its `.save()` method, ensuring the data adheres to the schema's structure and validation rules.

Example Schema: User Schema

First, define a simple `User` schema to structure the user documents:

```
const mongoose = require('mongoose');
const User = require('./models/User'); // Adjust the path to where your User model
is defined
```

```
// Connect to MongoDB (replace with your connection string)
mongoose.connect('mongodb+srv://<username>:<password>@cluster0.mongodb.net/myDatab
ase', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
  .then(() => {
    console.log('Connected to MongoDB');

    // Create an instance of the User model
    const user = new User({
      firstname: 'Akshad',
      lastname: 'Jaiswal',
      email: 'Akshad@example.com',
      age: 22,
      gender: 'Male'
    });

    // Save the document to the database
    return user.save();
  })
  .then(doc => console.log('Document inserted:', doc))
  .catch(err => console.error('Error:', err))
  .finally(() => mongoose.disconnect());
```

3. Automatic Fields Added by MongoDB

MongoDB automatically adds certain fields to documents within collections. Understanding these fields is crucial for effective data management. This document outlines the two key automatic fields: `_id` and `__v`.

1. `_id` Field

Description

The `_id` field is a unique identifier for each document in a MongoDB collection. It serves as the primary key for the document.

Characteristics

- **Type**: `ObjectId`
- **Uniqueness**: Each document must have a unique `_id` value. If you attempt to insert a document with a duplicate `_id`, MongoDB will return an error.
- **Auto-generated**: If not provided, MongoDB generates this field automatically when a document is created.

Example

```
{
  "_id": ObjectId("60d5b6f0d89a3c52a8d7c331"),
  "name": "John Doe"
}
```

Understanding the __v Field in MongoDB

The __v field is an automatic field added by MongoDB when using Mongoose to manage document versions. This README provides a detailed overview of the __v field, including its purpose, characteristics, and examples.

2. __v Field

The __v field is used to track the version of a document in a MongoDB collection. It helps manage concurrent updates and prevents overwriting changes made by other operations.

Characteristics

- **Type**: `Number`
- **Purpose**: The primary purpose of the __v field is to implement versioning for documents in Mongoose. It allows Mongoose to maintain the integrity of data during updates.
- **Auto-incremented**: Each time a document is updated, the __v field increments automatically. This ensures that Mongoose can identify the version of the document and handle any conflicts that arise from concurrent updates.

Example

Document Structure

When you create a document in a MongoDB collection using Mongoose, the __v field is added automatically. Here's an example of a document with the __v field:

```
{
  "_id": ObjectId("60d5b6f0d89a3c52a8d7c331"),
  "name": "John Doe",
  "__v": 0
}
```

Diving into APIs

Code Demonstration Link

- DevTinder Backend Repository

This README serves as a reference for API concepts and operations encountered while developing the DevTinder app. It includes notes on data handling, database interactions, and key differences between JavaScript and JSON objects.

Table of Contents

---

1. JavaScript Object vs JSON Object

Key Differences

Feature	JavaScript Object	JSON (JavaScript Object Notation)
Definition	A collection of key-value pairs in JavaScript	A text-based data format for representing structured data
Data Types Supported	Any JavaScript type (string, number, boolean, array, object, function, etc.)	Limited to strings, numbers, booleans, arrays, objects, and null
Syntax	Property names do not need to be in quotes	Property names must be in double quotes

Feature	JavaScript Object	JSON (JavaScript Object Notation)
Usage	Primarily used within JavaScript code for manipulation	Commonly used for data interchange between systems
Parsing Requirement	Not required in JavaScript, as it's native	Needs `JSON.parse()` to convert to a JavaScript object
Stringification Requirement	`JSON.stringify()` is used to convert to JSON format	Already in string format, no conversion needed for transmission
Functions Allowed	Can include functions as values	Does not support functions
Comments	Can contain comments	Does not allow comments

---

2. Receiving Data Through POST API

Overview

The POST API is used to receive data from the client. In the DevTinder app, this data is often user information that needs to be stored in the database.

Process

- **Step 1**: Set up a route that handles incoming POST requests.
- **Step 2**: Extract data from the request body. Ensure data validation and sanitize inputs to prevent issues such as SQL injection or XSS.
- **Step 3**: Once validated, structure the data appropriately before pushing it into the database.
- **Step 4**: Handle errors or success responses based on database operations to provide feedback to the client.

```
app.use(express.json());
//signup api for signing the user
app.post("/signup", async (req, res) => {
    const data = req.body;
    const user = new User(data)
    try {
        await user.save();
        res.send("User added successfully")

    } catch (err) {
        res.status(400).send("Error in saving the user:" + err.message)
    }
})
```

3. Retrieving Users from the Database

Overview

Fetching users from the database is essential for listing or displaying information on the frontend.

Process

- **Step 1**: Set up a GET API route that triggers the database query for retrieving users.
- **Step 2**: Define query filters (if any) based on the request parameters. For example, you may want to fetch users with specific attributes.
- **Step 3**: Execute the query and structure the results before sending them back in the API response.
- **Step 4**: Ensure the response format is JSON, allowing the client to process and display data as needed.

```
// Feed API - get all the users form the database
app.get("/feed", async (req, res) => {

    try {
        const users = await User.find({})
        if (users.length === 0) {
            res.send("No user found")
        } else {
            console.log(users);
            res.send(users)
        }
    }
    catch (err) {
        res.status(400).send("Something went wrong")
    }

})
```

4. Handling Duplicate Documents with findOne()

Overview

When `findOne()` is used with duplicate email documents in the database, MongoDB will return the **first matching document** it finds based on the collection's internal document ordering.

Key Points

- **Ordering**: MongoDB does not guarantee a specific order unless specified with an index. The document returned will typically be the first found, according to the insertion order.
- **Best Practice**: To avoid duplicates, enforce uniqueness in fields like email by using unique indexes.

```
    //user API to find the single user by by email
    app.get("/user", async (req, res) => {
        //getting user from body
        const userEmail = req.body.emailId;
        try {
            const users = await User.findOne({ emailId: userEmail })
            if (users.length === 0) {
                res.status(400).send("User not found")
            } else {

                // console.log(users)
                res.send(users)
            }
        }
        catch (err) {
            res.status(400).send("Something went wrong")
        }
    })
```

---

5. Delete API - Removing Documents from Database

Overview

The Delete API is used to remove specific documents from the database, allowing for efficient data management.

Key Method: `findByIdAndDelete()`

- Accepts the document's `_id` to identify and delete the target document.

- Returns the deleted document or `null` if no document is found.
- **Usage**: Primarily used for deleting user profiles or unwanted data from the collection.

```
    //delete user API - deleting a user by its id
app.delete("/user", async (req, res) => {
    const userId = req.body.userId;

    try {
        const users = await User.findByIdAndDelete(userId);
        res.send("User deleted Successfully")

    } catch (err) {
        res.status(400).send("Something went wrong")
    }
})
```

6. PATCH vs PUT API

Feature	PATCH	PUT
Purpose	To partially update a document	To completely replace a document
Required Data	Only the fields that need updating	All fields, even if only one field changes
Typical Use Case	Updating a few fields, like changing a user's email	Replacing or re-uploading an entire document
Database Interaction	Updates specific fields, leaving others unchanged	Replaces the document with a new version
HTTP Response Code	Typically 200 (OK) or 204 (No Content)	Typically 200 (OK) or 204 (No Content)

7. Updating Data with PATCH API

Overview

The PATCH API is used to update specific fields within a document without affecting other data.

Key Method: `findByIdAndUpdate()`

- Accepts the document's `_id` and an object with updated data.
- Only the fields in the update object will be modified; others remain unchanged.
- **Use Case**: Ideal for updating user information like username, bio, or profile picture without affecting other fields.

```
// patch user API - updating the data of user
app.patch("/user", async (req, res) => {
    const userId = req.body.userId;
    const data = req.body;

    try {
        const user = await User.findByIdAndUpdate({ _id: userId }, data, {
returnDocument: "before" });
        console.log(user)
        res.send("User updated successfully")

    } catch (err) {
        res.status(400).send("Something went wrong")
    }
})
```

This README is part of the DevTinder backend documentation. Expand upon these notes as you dive deeper into API handling and database management.

DevTinder Project - Data Sanitization and Schema Validations in Mongoose

Code Demonstration Link

- DevTinder Backend Repository

Overview

In today's learning, I explored data sanitization and schema validation features in Mongoose. These features help enforce data integrity, improve data consistency, and add extra layers of validation to ensure that the data saved in MongoDB adheres to specific rules.

Schema Types in Mongoose

Mongoose provides various schema types and properties that can be used to enforce data validation and sanitization. Key properties include:

1. `required`

- Ensures that a field must be provided before a document is saved.

```
firstName: {
        type: String,
        `required: true,`
        minLength: 3,
        maxLenght: 50
    }
```

2. `unique`

- Specifies that the value in the field must be unique across the collection.

```
emailId: {
        type: String,
        lowercase: true,
        required: true,
       `unique: true,`
        trim: true
    }
```

3. `default`

- Sets a default value for a field if no value is provided.

```
about: {
        type: String,
        `default: "Dev is in search for someone here"`
    }
```

4. `lowercase`

- Converts the string value to lowercase before storing it in the database.

```
emailId: {
        type: String,
        `lowercase: true,`
        required: true,
        unique: true,
        trim: true
    }
```

5. `trim`

- Removes leading and trailing whitespace from a string before saving it.

~~~
emailId: {
        type: String,
        lowercase: true,
        required: true,
        unique: true,
       `trim: true`
   }

### 6. `minLength` and `maxLength`
- Ensures that the length of a string field falls within a specified range.
```javascript
firstName: {
        type: String,
        required: true,
        `minLength: 3,`
        `maxLenght: 50`
    }
~~~

7. `min` and `max`

- Sets minimum and maximum values for numerical fields.

```
age: {
        type: Number,
        required: true,
        `min: 18`
    }
```

8. `validate`

- Allows for custom validation logic to be applied to a field. This can include custom functions for more complex validation needs.

```
gender: {
        type: String,
        required: true,
        trim: true,
       ` validate(value) {
            if (!["male", "female", "others"].includes(value)) {
                throw new Error("Not a valid gender (Male , Female and other)")
            }
        }`
    }
```

9. `timestamps`

- Automatically adds `createdAt` and `updatedAt` fields to the schema, tracking when the document was created and last modified.

```
{
   `timestamps: true`
}
```

Custom Validator Function

Custom validators can be used to enforce more complex validation logic. For example, a custom validator for the `age` field can ensure that only values meeting a specific condition are accepted.

```
gender: {
        type: String,
        required: true,
        trim: true,
       `validate(value) {
            if (!["male", "female", "others"].includes(value)) {
                throw new Error("Not a valid gender (Male , Female and other)")
            }
        }`
    }
```

API-Level Validations

PATCH API for Selected Field Updates

- **Field-Level Validation**: API-level validation ensures that only specific fields can be updated in a PATCH request. This limits changes to approved fields and improves security.
- **Selective Updates**: Enables users to update only allowed fields while maintaining the integrity of other data.
- **Benefits**:
    - Prevents unintended updates.
    - Minimizes errors by restricting updates to specified fields.
    - Enhances data security by controlling what data is modifiable.

```
        const ALLOWED_UPDATES = [
            "photoURL",
            "about",
            "gender",
            "skills",
            "firstName",
            "lastName",
            "age"
        ];

        const isUpdateAllowed = Object.keys(data).every((k) =>
ALLOWED_UPDATES.includes(k));

```

```
•         if (!isUpdateAllowed) {
•             throw new Error("Update Not Allowed")
          }
```

Conclusion

Schema validations and data sanitization in Mongoose provide powerful tools to maintain data quality and integrity. These features enable building robust applications by ensuring that only valid data is stored in the database, reducing the need for manual checks and potential data-related errors.

Key Validation Methods in Validator.js

Now focus on learning how to use the `validator.js` library for advanced data validation in the DevTinder app. Validator.js provides powerful and simple methods to validate user input fields, ensuring data accuracy and security.

1. Validating Email IDs

- **Method**: `isEmail`
- Ensures that the provided email address is in a valid format.
- Helps prevent invalid or malformed email addresses from being stored in the database.

```
validate(value) {
          if (!validator.isEmail(value)) {
              throw new Error("Invalid Email :" + value)
          }
      }
```

2. Validating Photo URLs

- **Method**: `isURL`
- Validates whether a string is a properly formatted URL.
- Useful for checking the validity of photo URLs uploaded by users.

```
validate(value) {
          if (!validator.isURL(value)) {
              throw new Error("Invalid URL :" + value)
          }
      }
```

3. Validating Password Strength

- **Method**: `isStrongPassword`

- Checks if the password meets specific criteria for strength, such as:
  - Minimum length.
  - Inclusion of uppercase and lowercase letters.
  - Numbers and special characters.
- Ensures that users create secure passwords to protect their accounts.

```
validate(value) {
         if (!validator.isStrongPassword(value)) {
             throw new Error("Enter a strong password :" + value)
         }
     }
```

Benefits of Using Validator.js

- **Improved Data Integrity**: Ensures only valid data is stored in the database.
- **Enhanced Security**: Prevents common vulnerabilities caused by invalid inputs.
- **Ease of Use**: Simple methods for complex validations reduce development time.
- **Standard Compliance**: Ensures data adheres to industry standards (e.g., email formatting, URL structure).

Conclusion

Using `validator.js` enhances the robustness of input validation in the DevTinder app. It simplifies the process of ensuring data consistency and security, particularly for critical fields like email, photo URLs, and passwords.

DevTinder Project - Password Encryption and Authentication

Code Demonstration Link

- DevTinder Backend Repository

Overview

This document outlines the key learnings and best practices for encrypting passwords and managing authentication in the DevTinder app. The process involves validating user input, securely storing passwords, and verifying credentials during login.

---

1. Signup Data Validation

Step 1: Validate User Input

Before processing any user-provided data, it is essential to validate it to maintain security and data integrity.

1. **Helper Functions**:
    - Create reusable helper functions to check individual fields (e.g., email, password strength).
    - These functions ensure the input meets the required format or constraints, simplifying validations throughout the app.
2. **Validator Library**:
    - Use the `validator` library for comprehensive and pre-built validation functions.
    - Common validations:
        - Check if the email is valid.
        - Ensure the password meets specific security criteria (e.g., length, character requirements).
        - Validate optional fields like profile URLs or phone numbers.

```
const validator = require("validator")
const validateSignupData = (req) => {
    const { firstName, lastName, emailId, password } = req.body;

    if (!firstName || !lastName) {
        throw new Error("Enter a vaid first or last name")
    } else if (!validator.isEmail(emailId)) {
        throw new Error("Enter a valid Email ID")
    } else if (!validator.isStrongPassword(password)) {
        throw new Error("Enter a strong password")
    }
}
module.exports = {
    validateSignupData
}
```

---

2. Password Encryption

Why Encrypt Passwords?

Storing plaintext passwords is highly insecure. Encrypting passwords ensures that even if the database is compromised, sensitive user data remains protected.

Step 2: Hashing Passwords

1. **Using `bcryptjs`**:
    - Install the `bcryptjs` package, which provides robust password hashing functionalities.
    - It is lightweight and specifically designed for Node.js applications.
2. **`bcrypt.hash`**:
    - The `bcrypt.hash` function is used to transform the plaintext password into an irreversible hashed string.
    - **Arguments**:
        - **Plain Password**: The raw password entered by the user during signup.
        - **Salt Rounds**: Determines the complexity of the hashing process. A higher value makes the hash stronger but increases computation time. (Recommended value: 10–12 for general use cases.)

```
 const passwordHash = await bcrypt.hash(password, 10)
```

Step 3: Storing Hashed Passwords

- After hashing, store the hashed string in the database instead of the plaintext password.
- The hashed string cannot be reverted to the original password, ensuring it remains secure even if exposed.

---

3. Login Authentication

Step 4: Verifying Passwords

1. `bcrypt.compare`:
   - The `bcrypt.compare` function is used during login to match:
     - The plaintext password provided by the user.
     - The hashed password stored in the database.
   - This ensures the user credentials are validated without ever exposing the original password.

```
const isValidPassword = await bcrypt.compare(password, user.password)
```

2. **Advantages of Using `bcrypt.compare`**:
   - Secure: Ensures no plaintext password is exposed during the process.
   - Efficient: Designed to handle comparisons quickly without compromising security.

Benefits of Encrypting Passwords

1. Enhanced Security

- Passwords are stored as hashed strings, reducing the risk of sensitive information being exposed in case of a database breach.

2. Compliance

- Password hashing complies with security standards and regulations (e.g., GDPR, HIPAA) that mandate secure handling of user data.

3. Trustworthiness

- Protecting user credentials builds trust and ensures users feel confident in the platform's security.

Conclusion

By implementing input validation and password encryption, the DevTinder app ensures that sensitive user data is handled securely. Utilizing robust libraries like `bcryptjs` and `validator` minimizes risks and aligns with best practices for modern web application security.

DevTinder Project -Authentication, JWT & Cookies

Code Demonstration Link

- [DevTinder Backend Repository]()

Overview

This document details the authentication process in the DevTinder app, focusing on JSON Web Tokens (JWT), cookie management, and security best practices. Understanding these concepts ensures secure user authentication and efficient session management.

---

1. Authentication

Key Concepts:

- Authentication verifies the user's identity by matching the email and password with database records.
- Upon successful login:
  i. The server generates a **JWT token**.
  ii. The token is sent back to the client and stored in a **cookie**.
  iii. Subsequent requests use the token for validation.

2. Cookies

What are Cookies?

- Cookies are small pieces of data stored on the client-side and sent to the server with every request.

JWT in Cookies

- Storing JWT tokens in cookies ensures that the server can validate user sessions without relying on client-side storage like `localStorage`.
- **Validation**:
  - Every request includes the token stored in cookies, allowing the server to verify its authenticity.

Cookie Expiry

- Cookies can have an expiration time to enforce session time limits, improving security and user control.

3. JWT (JSON Web Tokens)

Key Steps in Authentication Using JWT:

1. **Password Validation**:
   - After verifying the email and password, proceed to generate a JWT.
2. **Token Creation**:
   - A JWT token is created using the `jsonwebtoken` package with `jwt.sign`.
   - The token includes:
     - **Header**: Information about the token type and signing algorithm.
     - **Payload**: Contains user-specific data (e.g., user ID).
     - **Signature**: Ensures the token's integrity.

```
 //create a jwt token
    const token = await jwt.sign({ _id: user._id }, "999@Akshad", { expiresIn:
"1d" })
```

3. **Sending Token in Cookies**:
   - The generated token is added to cookies and sent back to the user as part of the response.

---

4. `res.cookie` Method

Cookie Setup:

- The `res.cookie` method is used to send cookies from the server to the client.
- **Setup**:
  - Install `cookie-parser` middleware to parse and manage cookies easily.
- Cookies can include settings like:
  - **Expiration Time**: Defines when the cookie will expire.
  - **HTTPOnly**: Ensures cookies are inaccessible via client-side scripts, reducing XSS risks.

```
res.cookie("token", token, { expires: new Date(Date.now() + 8 * 3600000) })
```

---

5. JWT.io

Understanding JWT Components:

1. **Header (Red)**:
   - Contains metadata about the token, such as type (`JWT`) and signing algorithm (e.g., `HS256`).
2. **Payload (Purple)**:
   - Contains claims, which are statements about the user (e.g., user ID).
3. **Signature (Blue)**:
   - Verifies that the token was not tampered with and is signed by the server.

---

6. Security Concepts

Cookie Hijacking or Stealing

- **Definition**:
  - Attackers can intercept or steal cookies via network vulnerabilities or client-side attacks.
- **Mitigation**:
  - Use `Secure` and `HTTPOnly` attributes for cookies.
  - Implement HTTPS to encrypt cookie transmission.
  - Rotate tokens periodically to limit exposure.

---

7. Middleware: `UserAuthentication`

Purpose:

- Verifies the JWT token sent in cookies for every protected route.
- Ensures only authenticated users can access restricted resources.

Process:

- The middleware uses `jwt.verify` to validate the token.
- If the token is invalid or missing, the request is denied with an appropriate error response.

```
const jwt = require("jsonwebtoken")
const User = require("../Models/user")
const userAuth = async (req, res, next) => {
    try {
        const { token } = req.cookies
        if (!token) {
            throw new Error("Not a Vaid token !!")
        }

        const deocodedObj = await jwt.verify(token, "999@Akshad")
        const { _id } = deocodedObj;

        const user = await User.findById(_id);
        if (!user) {
            throw new Error("User Not Found")
        }
        req.user= user;
        next();
    } catch (err) {
        res.status(400).send("ERROR : " + err.message)
    }
}
module.exports = {
    userAuth
}
```

What are `Schema.methods` in Mongoose?

Definition:

- `Schema.methods` is an object where you can define custom instance methods for a schema.
- These methods are available on all documents created with the schema.

Use Cases:

1. **Password Management**:
   - Use methods to hash passwords, compare hashed passwords during login, or perform other password-related operations.
2. **Token Generation**:
   - Attach functions to generate JWT tokens for authentication directly to the schema.
3. **Custom Business Logic**:
   - Add reusable methods for specific operations on schema instances, such as calculating derived data or formatting outputs.

```
// for jwt
    userSchema.methods.getjwt = async function () {
    const user = this;
    const token = await jwt.sign({ _id: this._id }, "999@Akshad", { expiresIn:
"1d" })

    return token;
}

// for password validation
userSchema.methods.validatePassword = async function (passwordInputByUser) {
    const user = this;
    const passwordHash = user.password;
    const isValidPassword = await bcrypt.compare(passwordInputByUser,
passwordHash);
    return isValidPassword;

}
```

Advantages of Using `Schema.methods`

1. **Encapsulation**:
    - Encapsulate document-specific logic within the schema for better modularity and readability.
2. **Reusability**:
    - Define functions once and reuse them across all instances of the schema.
3. **Integration with Mongoose Models**:
    - Work seamlessly with Mongoose's querying and data manipulation features.

Common Use Cases in DevTinder

1. **Authentication**:
    - Attach methods to hash passwords during signup and compare passwords during login.
2. **JWT Integration**:
    - Add a method to generate JWT tokens after successful password validation.
3. **Data Transformation**:
    - Create methods to sanitize or format data before sending it to the client.

DevTinder Project - API Development and Express Router

Code Demonstration Link

- DevTinder Backend Repository

Overview

Today's learning focused on finalizing the DevTinder API endpoints and structuring the application using Express Router for better modularity and maintainability. The APIs were categorized into different routers based on their functionality.

---

1. DevTinder API Endpoints

Auth Router

Handles user authentication, including signup, login, and logout.

- **POST `/signup`**: Register a new user.
- **POST `/login`**: Authenticate a user and issue a token.
- **POST `/logout`**: Revoke the user's session.

Profile Router

Manages user profile-related operations.

- **GET `/profile/view`**: Retrieve the profile information of the logged-in user.
- **PATCH `/profile/edit`**: Update user profile details.
- **PATCH `/profile/password`**: Change the user's password.

Connection Request Router

Handles connection requests between users with various statuses:

- **Status Options**: `ignore`, `interested`, `accepted`, `rejected`.
- **Endpoints**:

- **POST `/request/send/intrested/:userId`**: Send a connection request to another user.
- **POST `/request/ignored/:userId`**: Mark a request as ignored.
- **POST `/request/review/accepted/:requestId`**: Accept a connection request.
- **POST `/request/review/rejected/:requestId`**: Reject a connection request.

User Router

Handles operations related to connections, requests, and the user feed.

- **GET `/user/connections`**: Get a list of connections for the logged-in user.
- **GET `/user/requests/received`**: Retrieve a list of received connection requests.
- **GET `/user/feed`**: Get a list of suggested users to connect with.

---

2. Structuring with Express Router

Key Concepts:

1. **Creating a Routes Folder**:
   - Organize the API endpoints into separate route files for each functionality (e.g., `authRoutes.js`, `profileRoutes.js`).
2. **Using Express Router**:
   - Leverage the `express.Router()` to define routes in a modular way.
   - Each route file exports its router, which is then mounted to a specific path in the main application file.

```
const express = require("express");
const profileRouter = express.Router();
const { userAuth } = require("../Middlewares/auth");


//profile API to get the profile details
profileRouter.get("/profile", userAuth, async (req, res) => {
    const user = req.user;
    res.send(user);
});

module.exports = profileRouter;
```

Benefits of Using Express Router:

- **Modularity**: Separate files for each router improve code organization and readability.
- **Scalability**: Easier to maintain and expand as the application grows.
- **Reusability**: Common middleware and logic can be reused across routes.

Overview

Today's learning focuses on building key APIs for the DevTinder app:

1. A **Logout API** to securely log users out.
2. A **Profile/Edit API** to update user information while maintaining validation and security.

---

1. Logout API

Key Features:

- **Purpose**: Log users out by clearing their authentication cookies.
- **Implementation**:
    - Use the `res.cookie` method to set the cookie storing the JWT token to `null`.
    - Ensure the cookie is securely cleared by setting appropriate attributes (e.g., `httpOnly`, `secure`).

```
  authRouter.post("/logout", async (req, res) => {
    res
        .cookie("token", null, {
            expires: new Date(Date.now())
        })
        .send("User Logged out successfully")
})
```

Benefits:

- Ensures a secure and seamless logout process.
- Prevents unauthorized access by invalidating the session.

---

2. Profile/Edit API

Key Features:

- **Purpose**: Allow users to update their profile information.
- **Validation**:
  - Implement checks to ensure only specific fields (e.g., `firstName`, `about`, `profileURL` etc) can be updated.
  - Prevent updates to sensitive or immutable fields like `password` or `_id`.
- **Security**:
  - Validate all incoming data to ensure it meets predefined criteria (e.g., length, format).
  - Sanitize inputs to prevent injection attacks or unintended updates.

```
  profileRouter.patch("/profile/edit", userAuth, async (req, res) => {
    try {
        if (!validateEditFields(req)) {
            throw new Error("Invalid Edit request")
        }
        const loggedInUser = req.user;
        Object.keys(req.body).forEach(key => (loggedInUser[key] = req.body[key]))
        await loggedInUser.save();
        res.json({
            message: ` ${loggedInUser.firstName}, your profile updated 
successfully`,
            data: loggedInUser
        })
    }
    catch (err) {
        res.status(400).send("ERROR : " + err.message);
    }
})
```

Benefits:

- Maintains data integrity by restricting updates to allowable fields.
- Provides a secure mechanism for users to manage their profile information.

Conclusion

This structured approach to API development ensures that the DevTinder backend is organized, maintainable, and scalable. Using Express Router and clearly defined endpoints simplifies future feature additions and debugging.

DevTinder Project - Logical DB and Compound Indexes

Code Demonstration Link

- DevTinder Backend Repository Project

Overview

Today's learning covers advanced concepts in MongoDB and Mongoose, including creating a connection request schema with strict validations, building an API to handle connection requests, and using `.pre` middleware for additional checks.

---

1. Connection Request Schema

Key Features:

- **Schema Fields**:
  - `fromUserId`: The ID of the user sending the connection request.
  - `toUserId`: The ID of the user receiving the connection request.
  - `status`: The status of the request, restricted to specific values.
- **Status Validation**:
  - Only the following statuses are allowed:
    - `ignore`
    - `interested`
    - `accepted`
    - `rejected`
  - Any other value will throw a validation error using the `enum` type.
- **Custom Validators**:
  - Ensure only the predefined status values are accepted.
  - Provide meaningful error messages for invalid inputs.

---

2. API to Send Connection Requests

Endpoint:

- `POST /request/send/:status/:toUserId`

Key Validations:

1. **Allowed Statuses**:
   - Only "ignored" and "interested" statuses are allowed for this API.
2. **Duplicate Requests**:
   - Ensure that only one connection request can exist between two users, regardless of the direction (`fromUserId` to `toUserId` or vice versa).
   - If a request already exists, respond with a status indicating it was sent previously.
3. **Database Operation**:
   - Adds the `fromUserId`, `toUserId`, and `status` to the database upon successful validation.

---

3. `.pre` Middleware in Mongoose Schema

Key Logic:

- **Prevent Self-Requests**:
  - Use `.pre` middleware in the Mongoose schema to validate that the `fromUserId` is not the same as the `toUserId`.
  - Throw an error if the condition (`fromUserId !== toUserId`) is not met.

Benefits of Using `.pre` Middleware:

- Centralized validation logic directly within the schema.
- Ensures the database remains consistent by preventing invalid operations at the schema level.

4. Indexing in MongoDB

What is an Index?

- An index is a data structure that improves the speed of data retrieval operations on a database table or collection.
- It acts like a "table of contents" for your database, enabling faster searches.

Real-World Example:

- Searching for a common name like "Virat" among hundreds of entries can take significant time without an index.
- An index allows the database to find all matching entries much faster, reducing API response times.

Automatic Indexing:

- Fields marked with `unique: true` in a schema automatically have an index created for them by MongoDB.
    - Example: Unique fields like `email` or `username`.

Custom Indexing:

- Use `index: true` in the schema to create indexes for fields that are frequently queried but not marked as unique.
- Indexing non-unique fields like `firstname` can significantly improve query performance.

---

5. Compound Indexes

What are Compound Indexes?

- Compound indexes are indexes created on multiple fields in a collection.
- They improve the performance of queries that filter or sort on multiple fields.

Use Cases:

- Queries involving multiple fields, such as filtering users by `city` and `age`.
- Sorting results efficiently when multiple conditions are applied.

Benefits:

- Faster data retrieval for complex queries.
- Reduced latency in API responses.

6. Limitations and Trade-Offs of Indexing

Why Not Create Too Many Indexes?

1. **Storage Overhead**:
   - Each index consumes additional disk space, increasing the overall size of the database.
2. **Write Performance Impact**:
   - Creating and maintaining indexes adds overhead to write operations (insert, update, delete).
3. **Index Maintenance**:
   - The database must update all relevant indexes whenever a document is modified, leading to slower write operations.

Best Practices:

- Index only the fields that are queried frequently.
- Use compound indexes strategically to cover complex queries.
- Regularly monitor and optimize indexes to balance performance and resource usage.

Conclusion

These enhancements improve the robustness of the DevTinder app by ensuring data integrity and adding meaningful validations. The use of enums, compound validations, and middleware demonstrates effective use of Mongoose and MongoDB features for scalable application development.

DevTinder Project - ref, populate and Thought process for writing API's

Code Demonstration Link

- DevTinder Backend Repository Project

Overview

Today's focus is on using `ref`, `populate`, and implementing a structured thought process for writing APIs. The work involves building an API to review and update the status of a connection request.

API Details

Endpoint:

- **POST** `/request/review/:status/:requestId`

Purpose:

- To accept or reject a connection request by updating its status based on specific conditions.

Thought Process for API Development

Steps:

1. **Getting the Logged-In User**:
   - Identify the user making the request using authentication mechanisms (e.g., JWT or session).
2. **Retrieving Parameters**:
   - Extract `status` and `requestId` from the API parameters.

3. **Validating Input**:
   - Ensure that the `status` parameter only contains valid values:
     - Allowed values: `accepted` or `rejected`.
   - Return an error response if the `status` is invalid.
4. **Searching for the Connection Request**:
   - Query the database to find a connection request that:
     - Matches the provided `requestId`.
     - Has the `toUserId` field equal to the logged-in user's ID.
     - Has the `status` set to `interested`.
5. **Updating the Status**:
   - If a matching connection request is found, update its `status` field to the new value provided in the parameters.
   - Respond with a success message or the updated connection request data.

```
requestRouter.post(
"/request/review/:status/:requestId",
userAuth,
async (req, res) => {
    try {
    const loggedInUser = req.user;
    const { status, requestId } = req.params;

    //Validate Status
    const allowedStatuses = ["accepted", "rejected"];
    if (!allowedStatuses.includes(status)) {
        return res.status(400).json({
        message: "Invalid Status or Status not allowed",
        success: false,
        });
    }

    //validating the request
    const connectionRequest = await ConnectionRequestModel.findOne({
        _id: requestId,
        toUserId: loggedInUser._id,
        status: "intrested",
    });

    if (!connectionRequest) {
        return res.status(404).json({
        message: "request not found ",
        success: false,
        });
    }

    connectionRequest.status = status;
    const data = await connectionRequest.save();

    res.status(200).json({
        message: "Connection request " + status,
        data,
        success: true,
    });
    } catch (error) {
    res.status(400).send("ERROR:" + error.message);
    }
}
);
```

Key Concepts

1. `ref` and `populate` in Mongoose

- `ref`:
    - Defines relationships between collections in MongoDB.
    - Used to reference documents from other collections (e.g., linking `toUserId` and `fromUserId` to the User collection).
- `populate`:
    - Populates referenced fields with actual document data instead of just IDs.
    - Simplifies retrieving related data in a single query.

API Details (UserSide)

Endpoint:

- **GET** `/user/requests/received`

Purpose:

- Fetch all pending connection requests where the logged-in user is the recipient (`toUserId`) and the request status is `interested`.

Thought Process for API Development

Steps:

1. **Setting up the User Router**:
    - Create a dedicated router for user-related operations to improve modularity and organization.
2. **Authenticating the API**:

    - Apply authentication middleware to ensure the user is logged in before processing the request.
    - Use tokens (e.g., JWT) to validate and retrieve the logged-in user.

3. **Fetching Connection Requests**:
    - Query the `ConnectionRequest` collection to retrieve all requests with:
        - `toUserId` matching the logged-in user's ID.
        - `status` set to `interested`.
4. **Building Relationships Between Schemas**:
    - Use `ref` in the `ConnectionRequest` schema to reference the `User` schema for the `fromUserId` field.
    - Populate the `fromUserId` field to retrieve the related user's data.
5. **Returning Specific Fields**:
    - Use `populate` to fetch only the `firstName` and `lastName` of the `fromUserId` user.
    - Exclude unnecessary fields to optimize the response payload.

```
//Setting Reference
fromUserId: {
  type: mongoose.Schema.Types.ObjectId,
  ref: "User",
  required: true,
},

//Popuating Data
userRouter.get("/user/requests/recieved", userAuth, async (req, res) => {
try {
    const loggedInUser = req.user;
    const connectionRequests = await ConnectionRequestModel.find({
    toUserId: loggedInUser._id,
    status: "intrested",
    }).populate("fromUserId", ["firstName", "lastName"]);
    if (connectionRequests) {
    return res.status(200).json({
        connectionRequests,
    });
    }
} catch (error) {
    res.status(400).send("ERROR:" + error.message);
}
});
```

API Details

Endpoint:

- **GET** `/user/connections`

Purpose:

- Fetch all accepted connection requests where the logged-in user is either the sender (`fromUserId`) or the receiver (`toUserId`).

Thought Process for API Development

Steps:

1. **Fetching Connection Requests**:
   - Query the `ConnectionRequest` collection to retrieve requests that:
     - Have `status` set to `"accepted"`.
     - Include the logged-in user's ID in either the `toUserId` or `fromUserId` fields.
2. **Data Relationships**:
   - Use relationships between the `ConnectionRequest` and `User` schemas to retrieve additional user data if necessary.
   - Leverage Mongoose's `populate` method to enhance the data with user details.
3. **Returning Results**:
   - Filter and return all matching connection requests as part of the response.
   - Optimize the response payload by including only relevant fields (e.g., user names, IDs).

```
userRouter.get("/user/connections", userAuth, async (req, res) => {
  try {
    const loggedInUser = req.user;

    const connectionRequests = await ConnectionRequestModel.find({
      $or: [
        { toUserId: loggedInUser._id, status: "accepted" },
        { fromUserId: loggedInUser._id, status: "accepted" },
      ],
    }).populate("fromUserId", USER_SAFE_DATA);

    const data = connectionRequests.map((row) => row.fromUserId);

    res.status(200).json({
      data,
    });
  } catch (error) {
    res.status(400).send("ERROR :" + error.message);
  }
});
```

DevTinder Project - API Development: Feed API and Pagination

Code Demonstration Link

- DevTinder Backend Repository Project

Overview

Today's focus is on enhancing the **Get /user/feed** API by implementing pagination. The API is designed to retrieve user profiles while filtering out specific users and limiting the number of results per request.

API Details

Endpoint:

- **GET** `/user/feed`

Purpose:

- Fetch profiles of other users while excluding:
  - The logged-in user.
  - Existing connections.
  - Ignored users.
  - Users who have already received a connection request.

Thought Process for API Development

1. Filtering Out Unwanted Users

- Use a **Set data structure** to collect and store the following user IDs:
  - `fromUserId` and `toUserId` from all connection requests associated with the logged-in user.
  - This prevents duplicate filtering and ensures an optimized query.
- Use **MongoDB operators**:
  - **$nin**: Exclude multiple user IDs in a single query.
  - **$ne**: Ensure the logged-in user's profile is not included in the query results.

2. Implementing Pagination

- **Why Pagination?**
  - If there are **1000+ users**, returning all users at once would be inefficient.
  - Pagination ensures that only a specific number of users are retrieved per request.
- **Fetching `page` and `limit` from Request Parameters**:
  - `page`: Determines which page of users the client is requesting.
  - `limit`: Defines the maximum number of users to be returned.
- **Calculating the Skip Value**:
  - The formula for calculating skipped records:
  - `skip = (page - 1) * limit`
  - `skip`: Number of users to ignore before returning results.
- **Applying Skip and Limit in the Query**:
  - `skip()` is used to move past the number of users from previous pages.
  - `limit()` ensures only the requested number of users is retrieved.

Key Concepts

1. Set-Based Filtering for Efficient Querying

- Using a **Set** prevents duplicates and ensures only valid user IDs are excluded from the query.
- This helps in maintaining fast query performance.

2. MongoDB Query Optimization

- **$nin**: Excludes all users in the filter list.
- **$ne**: Ensures the logged-in user's profile does not appear in the results.

3. Pagination for Better Performance

- **Pagination prevents excessive data loading**, reducing server response time.
- Allows for smooth user experience with "Load More" or infinite scrolling features.

```
 userRouter.get("/user/feed", userAuth, async (req, res) => {
try {
  const loggedInUser = req.user;

  const page = parseInt(req.query.page || 1);
  let limit = parseInt(req.query.limit || 10);
  limit = limit > 50 ? 50 : limit;
  const skip = (page - 1) * limit;

  const connectionRequest = await ConnectionRequestModel.find({
    $or: [{ fromUserId: loggedInUser._id }, { toUserId: loggedInUser._id }],
  }).select("fromUserId toUserId");

  const hideUsersFromFeed = new Set();
  connectionRequest.forEach((req) => {
    hideUsersFromFeed.add(req.fromUserId.toString());
    hideUsersFromFeed.add(req.toUserId.toString());
  });

  const users = await User.find({
    $and: [
      { _id: { $nin: Array.from(hideUsersFromFeed) } },
      { _id: { $ne: loggedInUser._id } },
    ],
  })
    .select(USER_SAFE_DATA)
    .skip(skip)
```

```
      .limit(limit);

    res.send(users);
  } catch (error) {
    res.status(400).send("ERROR: " + error.message);
  }
});
module.exports = userRouter;
```

Benefits of This Approach

1. **Improved Performance**:
   - Querying a subset of users per request optimizes response times.
2. **Scalability**:
   - Ensures that the API remains efficient even with a large number of users.
3. **Better User Experience**:
   - Users receive paginated data, making the browsing experience smooth.

Conclusion

The **Get /user/feed** API now supports **pagination**, ensuring that user profiles are efficiently retrieved while maintaining a smooth and scalable user experience. Filtering with **MongoDB queries** and **pagination techniques** enhances the performance and usability of the DevTinder app.