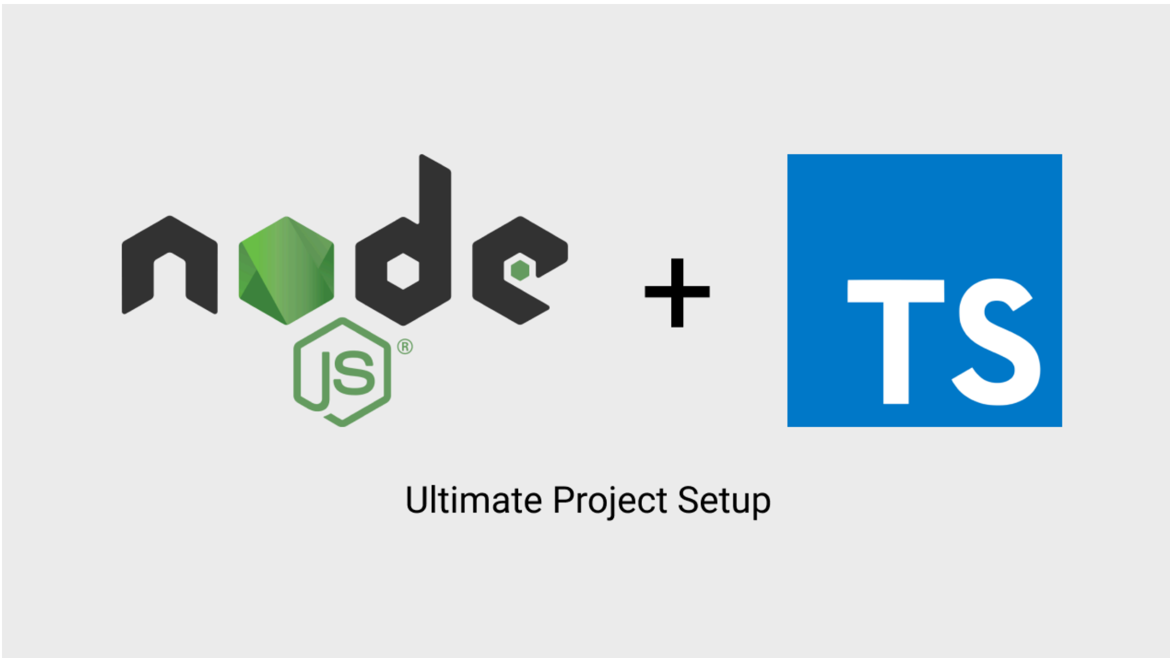
Node.js + Typescript Course

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to build server-side and network applications using JavaScript.

Table of contents

1. Introduction

What is Node.js?



node_ts.png

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to build server-side and network applications using JavaScript. It is built on Chrome's V8 JavaScript engine, enabling fast and scalable development. Unlike traditional JavaScript, which typically runs in the browser, Node.js executes JavaScript code outside of the browser, making it ideal for backend development, real-time applications, and much more.

Key features of Node.js:

- Asynchronous and event-driven
- Non-blocking I/O model
- Lightweight and efficient
- Uses the same language (JavaScript) on both frontend and backend

```
// A simple Node.js server in TypeScript

import * as http from 'http';

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

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

Why Node.js?

Node.js is widely adopted for a variety of reasons, making it one of the most popular tools for backend development:

1. **Single Language for Fullstack Development**: With Node.js, developers can use JavaScript for both the frontend and backend, making it easier for full-stack development and reducing the context switching between different languages.
2. **Fast Execution with V8 Engine**: The V8 engine, which powers Node.js, compiles JavaScript to native machine code, providing highly efficient execution.
3. **Non-blocking, Asynchronous Model**: Node.js uses an event-driven, non-blocking I/O model, which allows it to handle many connections concurrently without being resource-intensive. This makes Node.js particularly suitable for I/O-heavy applications, like web servers or real-time applications.
4. **Large Ecosystem**: The Node Package Manager (NPM) is the largest ecosystem of open-source libraries, which makes adding functionalities like database interaction, authentication, or real-time communication easy.
5. **Scalability**: Node.js is designed with scalability in mind, allowing developers to build applications that handle numerous requests efficiently.

```
// Demonstrating non-blocking behavior in Node.js
import * as fs from 'fs';

console.log('Start reading file...');
fs.readFile('example.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('File content:', data);
});
console.log('Do other work while file is being read.');
```

In the above example, Node.js doesn't block the execution while reading the file, allowing the server to perform other tasks.

What Can Node.js Do?

Node.js can be used for a wide range of applications, making it a versatile tool in modern web development. Here are some common use cases:

1. **Web Servers**: Node.js is commonly used to build web servers. Its ability to handle multiple requests simultaneously with non-blocking I/O makes it highly efficient for web applications.

2. **API Development**: Node.js is an excellent choice for building RESTful APIs and microservices, thanks to its speed and the extensive support for libraries through NPM.

3. **Real-time Applications**: Real-time features such as chat applications, live data streaming, and collaborative tools can be efficiently built using Node.js. Libraries like Socket.io make it easy to implement WebSocket-based communication.

4. **Command Line Tools**: You can use Node.js to write command-line tools and scripts that automate tasks. With access to file systems, processes, and child processes, Node.js can handle various automation tasks.

5. **Serverless Functions**: Many cloud providers offer serverless computing using Node.js as the runtime. This allows you to write lightweight functions that automatically scale and handle HTTP requests.

```typescript
// Simple REST API with Node.js and Express in TypeScript

import express from 'express';

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

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

app.get('/api', (req, res) => {
  res.json({ message: 'Welcome to the API!' });
});

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

With the above example, you can create a simple web server that responds to different routes. This is a very basic example of how Node.js can power web applications or APIs.

This introduction sets the stage for understanding Node.js by explaining its core concepts, benefits, and practical use cases.

2. Getting Started

Download Node.js

To get started with Node.js, you'll first need to download and install it on your system. Node.js comes with its package manager, **npm** (Node Package Manager), which allows you to install libraries and dependencies for your project.

1. Visit the official Node.js website: https://nodejs.org/ (https://nodejs.org/)
2. You'll see two versions available:
    - **LTS (Long Term Support)**: This version is recommended for most users as it is stable and gets security updates for a longer time.
    - **Current**: This is the latest version with new features, but it may not be as stable as the LTS version.
3. Download the appropriate installer for your operating system (Windows, macOS, or Linux).
4. Run the installer and follow the setup instructions.

Once installed, you can verify the installation by opening your terminal (or Command Prompt) and typing:

```
node -v
```

This should display the version of Node.js installed.

You can also verify npm by running:

```
npm -v
```

This will show the version of npm that was installed alongside Node.js.

Command Line Interface (CLI)

Node.js provides a simple command-line interface (CLI) for running JavaScript and TypeScript code. You can interact with Node.js through your terminal or command prompt.

1. Running Node.js REPL (Read-Eval-Print Loop)

Node.js comes with a REPL, which allows you to execute JavaScript code interactively.

To start the REPL, just type `node` in your terminal and press Enter:

```
$ node
Welcome to Node.js v16.13.0.
Type ".help" for more information.
> console.log('Hello, Node.js!');
Hello, Node.js!
>
```

Here, you can execute any JavaScript code, and Node.js will evaluate it immediately. To exit the REPL, type `.exit` or press `Ctrl + C` twice.

2. Running Node.js Scripts

To execute a JavaScript or TypeScript file with Node.js, you can use the following command:

```
node <filename.js>
```

For example, if you have a file `app.js` that contains:

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

You can run it with:

```
node app.js
```

This will output:

```
Hello, World!
```

3. Installing Global Packages

Node.js also uses npm for package management. You can install Node.js packages globally (so they are accessible anywhere) using npm:

```
npm install -g typescript
```

In this example, we installed **TypeScript** globally, allowing you to use the `tsc` (TypeScript Compiler) command.

Initiate the Node.js File

Before creating your first Node.js project, it's a good practice to set up a project directory and initialize it using npm. This will allow you to manage your dependencies and set up your project structure.

1. Create a New Project Directory

Open your terminal and create a new directory for your project:

```
mkdir my-node-app
cd my-node-app
```

2. Initialize the Project with npm

To initialize a new Node.js project, use the following command:

```
npm init
```

This command will ask you several questions about your project (like project name, version, entry point, etc.), but you can skip through them by pressing Enter or manually configure them.

Alternatively, you can use the `-y` flag to accept the default settings:

```
npm init -y
```

This will generate a `package.json` file, which is a configuration file that stores metadata about your project, including dependencies, scripts, and more.

The package.json file will look something like this:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

3. Create Your First Node.js File

Now, create a new file called index.ts (or index.js if you're using plain JavaScript):

```
touch index.ts
```

Inside this file, you can write your first Node.js code using TypeScript:

```
const message: string = 'Hello, TypeScript with Node.js!';
console.log(message);
```

4. Run the TypeScript File

To run TypeScript with Node.js, you need to transpile the TypeScript code into JavaScript using the TypeScript compiler (tsc).

First, install TypeScript as a dev dependency:

```
npm install typescript --save-dev
```

You can now compile the index.ts file into JavaScript:

```
npx tsc index.ts
```

This will generate a JavaScript file (`index.js`), which can then be executed using Node.js:

```
node index.js
```

The output will be:

```
Hello, TypeScript with Node.js!
```

5. Automate the Compilation and Running (Optional)

To make the process of compiling and running TypeScript easier, you can add a script to your `package.json`:

```
{
  "scripts": {
    "start": "tsc && node index.js"
  }
}
```

Now, you can compile and run your project with one command:

```
npm start
```

This will compile the TypeScript and run the generated JavaScript file.

This section guides you through the process of downloading and setting up Node.js, familiarizing you with the command line interface, and creating your first Node.js project.

3. Node Package managers

Node.js uses various package managers to handle external libraries and dependencies, which are essential for building modern applications. These package managers simplify the process of installing, updating, and managing the libraries your project requires. In this section, we'll cover what a package is, and introduce the most commonly used package managers: **NPM**, **PnPm**, and **Yarn**.

What is a Package?

In the Node.js ecosystem, a **package** refers to a reusable piece of code or library that performs a specific task or provides specific functionality. Packages are distributed as modules, which can include utilities, frameworks, middleware, or even entire applications.

A typical package may consist of:

- JavaScript/TypeScript code
- A `package.json` file containing metadata (name, version, dependencies, etc.)
- Additional resources like README files, license agreements, and configuration files

Packages can be installed locally (only available to the project) or globally (available system-wide). They are often hosted in package registries like the official **npm registry.**

Example of a `package.json` file:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "description": "A simple Node.js app",
  "main": "index.js",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "typescript": "^4.4.3"
  },
```

```
  "scripts": {
    "start": "node index.js",
    "build": "tsc"
  }
}
```

In this example, the `express` package is listed as a runtime dependency, while `typescript` is a development dependency.

NPM (Node Package Manager)

NPM (Node Package Manager) is the default package manager for Node.js, and it comes pre-installed when you install Node.js. NPM allows developers to easily share and reuse code and manage dependencies.

Key Features:

- **Largest Package Registry**: NPM has the largest repository of open-source Node.js libraries available for installation.
- **Dependency Management**: It manages both direct and transitive dependencies through a `package.json` file.
- **Scripts**: NPM allows you to define custom scripts in `package.json` to automate tasks like testing, building, or deploying.
- **Versioning**: NPM makes it easy to update or lock dependencies to specific versions, ensuring that your project uses stable versions of packages.

Common NPM Commands:

- Install a package locally:

  ```
  npm install <package-name>
  ```

- Install a package globally:

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

- Add a package as a development dependency:

```
npm install <package-name> --save-dev
```

- Install all dependencies listed in `package.json`:

```
npm install
```

- Remove a package:

```
npm uninstall <package-name>
```

- Running a custom script defined in `package.json`:

```
npm run <script-name>
```

Example:

```
npm install express
```

This command installs `express`, a web framework, and updates `package.json` with the new dependency.

c. PnPm

PnPm is an alternative package manager for Node.js that aims to optimize disk space and speed. It addresses some of the inefficiencies of NPM and Yarn by storing package dependencies in a shared, central directory, and linking them into each project. This can significantly reduce disk usage, especially when multiple projects have similar dependencies.

Key Features:

- **Efficient Disk Usage**: PnPm uses a content-addressable storage to store every package only once, even if multiple projects depend on different versions of the package.

- **Performance**: It is often faster than both NPM and Yarn, especially for monorepo projects.
- **Strict Dependency Management**: PnPm ensures that the dependency tree is more predictable, catching issues like missing or incompatible peer dependencies early on.

Installing PNPM

PNPM is a fast, disk space-efficient package manager for Node.js. More information on https://pnpm.io/installation (https://pnpm.io/installation)

1. **Install PNPM**:

- Using Powershell:

```
Invoke-WebRequest https://get.pnpm.io/install.ps1 -UseBasicParsing |
Invoke-Expression
```

- On POSIX systems, you can use curl or wget:

```
curl -fsSL https://get.pnpm.io/install.sh | sh -
```

If you don't have curl installed, you would like to use wget:

```
wget -qO- https://get.pnpm.io/install.sh | sh -
```

2. **Verify Installation**: After the installation completes, verify it by typing:

```
pnpm -v
```

This should display the installed PNPM version.

3. **Updating pnpm**: To update pnpm, run the `self-update` command:

```
pnpm self-update
```

To install Node.js with PNPM on your system, follow the steps below:

Installing Node.js

Node.js is a JavaScript runtime that allows you to run JavaScript code on your server or local machine. Here's how to install it:

use

- Install and use the specified version of Node.js. Install the LTS version of Node.js:

```
pnpm env use --global lts
```

- Or if you prefer a specific Install Node.js v16:

```
pnpm env use --global 16
```

Common PnPm Commands:

- Install a package locally:

  ```
  pnpm install <package-name>
  ```

- Install a package globally:

  ```
  pnpm add -g <package-name>
  ```

- Add a development dependency:

  ```
  pnpm add <package-name> --save-dev
  ```

- Install all dependencies listed in `package.json`:

  ```
  pnpm install
  ```

Example:

```
pnpm install express
```

This command installs `express`, just like NPM, but PnPm will link the dependencies in an optimized way, saving disk space.

Advantages:

- Faster installation times in larger projects.
- Prevents projects from installing unnecessary duplicate dependencies.
- Strictness in dependency resolution ensures fewer conflicts.

d. Yarn

Yarn is another package manager that was originally created by Facebook to address some performance and security concerns they found in NPM. Yarn improves on speed and consistency, offering a lockfile system and better offline support.

Key Features:

- **Fast Installation**: Yarn caches every package it downloads, so it never needs to download the same package again.
- **Lockfile for Consistency**: Yarn generates a `yarn.lock` file to ensure that all developers working on a project install exactly the same package versions.
- **Parallel Installations**: Yarn installs packages in parallel, which can significantly reduce installation times.
- **Offline Mode**: Once a package is installed, it can be reinstalled without an internet connection, thanks to Yarn's caching system.

Common Yarn Commands:

- Install a package locally:

  ```
  yarn add <package-name>
  ```

- Install a package globally:

```
yarn global add <package-name>
```

- Add a development dependency:

```
yarn add <package-name> --dev
```

- Install all dependencies listed in `package.json`:

```
yarn install
```

Example:

```
yarn add express
```

This command adds `express` as a dependency to your project using Yarn.

Advantages:

- **Faster** than NPM due to parallel package installation.
- **Offline Mode** allows reinstallation of previously installed packages without an internet connection.
- **Deterministic**: The `yarn.lock` file ensures that every installation will result in the same file structure.

Comparison

Common Commands

COMMAND	NPM	PNPM	YARN
init	`npm init`	`pnpm init`	`yarn init`
install from package.json	`npm install`	`pnpm install`	`yarn`
add package	`npm install <package> [--location=global]`	`pnpm add <package> [--global]`	`yarn add <package>`
add package as devDependencies	`npm install <package> --save-dev`	`pnpm add <package> --save-dev`	`yarn add <package> --dev`
remove package	`npm uninstall <package> [--location=global]`	`pnpm uninstall <package> [--global]`	`yarn [global] remove <package>`
remove package as devDependencies	`npm uninstall <package> --save-dev`	`pnpm uninstall <package> --save-dev`	`yarn remove <package> --dev`
audit vulnerable dependencies	`npm list --depth 0 [--location=global]`	`pnpm list --depth 0 [--global]`	`yarn [global] list --depth 0`
Run	`npm run <script-name>`	`pnpm run <script-name>`	`yarn run <script-name>`
build	`npm build`	`pnpm build`	`yarn build`
test	`npm test`	`pnpm test`	`yarn test`

Features

Feature	NPM	PnPm	Yarn
Speed	Standard	Very fast	Fast
Disk Usage	Can be heavy on disk space	Optimized with central storage	Efficient with caching
Offline Mode	Limited	No, but faster for cached files	Yes
Dependency Tree	Standard	Strict	Flexible but with lockfiles
Parallel Installs	No	Yes	Yes

In conclusion, NPM, PnPm, and Yarn all have their strengths and weaknesses, and your choice of package manager depends on your specific needs. **NPM** is the default and easiest to use, **PnPm** optimizes disk space and speed for large projects, and **Yarn** offers performance improvements and more deterministic installs.

3.1 PnPm Cheatsheet

Here's a cheat sheet for using PNPM to perform common tasks. PNPM is a fast and efficient package manager for Node.js projects, and this guide will help you get started with some of the most frequently used commands.

1. Initialization

- **Initialize a new project (create `package.json`):**

  ```
  pnpm init
  ```

2. Installing Packages

- **Install all dependencies listed in `package.json`:**

  ```
  pnpm install
  ```

- **Install a specific package (e.g., `lodash`):**

  ```
  pnpm add lodash
  ```

- **Install a package as a development dependency:**

  ```
  pnpm add eslint --save-dev
  ```

  or

  ```
  pnpm add -D eslint
  ```

- **Install a specific version of a package:**

  ```
  pnpm add lodash@4.17.20
  ```

- **Install dependencies without modifying `package.json` (useful for CI/CD):**

```
pnpm install --frozen-lockfile
```

3. Removing Packages

- **Uninstall a package:**

```
pnpm remove lodash
```

4. Running Scripts

- **Run a script defined in `package.json`:**

```
pnpm run <script_name>
```

Example:

```
pnpm run build
```

- **Run a package binary without installing it globally:**

```
pnpm dlx <package_name>
```

Example:

```
pnpm dlx create-react-app my-app
```

5. Working with Global Packages

- **Install a package globally:**

```
pnpm add -g eslint
```

- **List globally installed packages:**

```
pnpm list -g --depth 0
```

6. Managing Dependencies

- **Update all dependencies to the latest versions:**

```
pnpm update --latest
```

- **Install dependencies without running `preinstall` and `postinstall` scripts:**

```
pnpm install --ignore-scripts
```

7. Managing the PNPM Cache

- **Clear the PNPM cache:**

```
pnpm cache clean
```

8. Workspaces

- **Create a new workspace:**

```
pnpm init
```

- **Add a package to a workspace:**

```
pnpm add <package_name> -w
```

- **Run a command in all workspace packages:**

```
pnpm -r <command>
```

Example:

```
pnpm -r build
```

9. Linking Packages

- **Link a package globally:**

```
pnpm link
```

- **Link a package locally within a project:**

```
pnpm link <package_name>
```

10. Miscellaneous

- **Check for outdated packages:**

```
pnpm outdated
```

- **List all installed packages:**

```
pnpm list
```

4. Typescript Configuration

TypeScript Configuration for a Node.js App with tsx

To configure TypeScript in a Node.js project, and add `tsx` (a tool that allows running TypeScript files without needing to transpile manually), we need to make sure the project has the appropriate dependencies and configurations in place. Additionally, we'll create scripts for starting, developing, and building the app.

1. Setting Up the Node.js App

Start by initializing a new Node.js project if you haven't done so already:

```
mkdir my-node-app
cd my-node-app
npm init -y
```

2. Installing Dependencies

Next, you'll need to install TypeScript, `tsx`, and types for Node.js:

```
npm install typescript tsx @types/node --save-dev
```

- **typescript**: The TypeScript compiler (to compile `.ts` files to JavaScript).
- **tsx**: A package that allows you to run TypeScript and JSX (React) files directly without needing to compile.
- **@types/node**: Provides TypeScript definitions for Node.js modules (like `http`, `fs`, etc.).

3. Creating the tsconfig.json

TypeScript requires a `tsconfig.json` file for configuration. This file tells the TypeScript compiler how to transpile the `.ts` files. You can create it by running the following command:

```
npx tsc --init
```

Then, modify the `tsconfig.json` file to suit a typical Node.js environment:

`tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES6",                     // Target ES6 or later to
support modern JavaScript features
    "module": "CommonJS",                 // Use CommonJS modules for
Node.js
    "rootDir": "./src",                   // Source files location
    "outDir": "./dist",                   // Output directory for
compiled files
    "esModuleInterop": true,              // Enables support for
`import` in CommonJS
    "strict": true,                       // Enables strict type-
checking options
    "skipLibCheck": true,                 // Skips type checking of
declaration files
    "moduleResolution": "node",           // For resolving node modules
    "resolveJsonModule": true,            // Enable importing `.json`
files
    "allowSyntheticDefaultImports": true, // Allow default imports from
modules
    "forceConsistentCasingInFileNames": true // Ensure consistent casing
in file names
  },
  "include": ["src/**/*.ts"],             // Include all TypeScript
files in the `src` folder
  "exclude": ["node_modules"]             // Exclude `node_modules`
}
```

4. Project Structure

Here's an example of how your project structure should look:

```
/my-node-app
  ├── /src
  │     ├── index.ts
  ├── /dist                     # Will be generated after build
  ├── package.json
  ├── tsconfig.json
  └── node_modules
```

5. Creating Scripts for Start, Dev, and Build

Next, let's modify the `package.json` file to include scripts for starting, developing, and building the app:

`package.json`:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "main": "dist/index.js",
  "scripts": {
    "start": "node dist/index.js",              // Start the compiled
JavaScript code
    "dev": "tsx watch src/index.ts",            // Start the app in
development mode with hot-reloading
    "build": "tsc"                              // Compile TypeScript
files to JavaScript
  },
  "devDependencies": {
    "@types/node": "^18.0.0",
    "tsx": "^3.12.7",
    "typescript": "^4.4.3"
  }
}
```

Explanation of the scripts:

- **start**: This runs the app using node after the TypeScript files have been compiled to JavaScript (found in the dist directory).
- **dev**: This runs the app directly with tsx, allowing you to start the TypeScript app in development mode without needing to compile first. tsx watch enables hot-reloading so that the server automatically reloads whenever you make changes.
- **build**: This runs the TypeScript compiler (tsc), which compiles the TypeScript files from the src directory into JavaScript in the dist directory.

6. Example TypeScript File

Now, let's create a simple index.ts file in the src folder to demonstrate the setup:

src/index.ts:

```
import http from 'http';

const hostname = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, TypeScript with Node.js!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

7. Running the App

Development Mode:

To run the app in development mode with hot-reloading (using tsx), run:

```
npm run dev
```

This will start the server, and any changes made to your TypeScript files will automatically restart the server.

Build and Start:

To compile your TypeScript code to JavaScript and then run the app:

1. First, build the app:

   ```
   npm run build
   ```

   This will transpile the TypeScript files and output them to the `dist` directory.

2. Then, run the app:

   ```
   npm start
   ```

This will execute the compiled JavaScript files in the `dist` directory.

Summary

1. **Install Dependencies**: Install `typescript`, `tsx`, and `@types/node` to set up TypeScript and run TypeScript files in Node.js without manual compilation.
2. **`tsconfig.json`**: Configure TypeScript settings for a Node.js project with `"module": "CommonJS"` and `"target": "ES6"`.
3. **Scripts**:
   - `start`: Runs the compiled app using `node`.
   - `dev`: Runs the app directly in TypeScript with `tsx` in watch mode.
   - `build`: Compiles TypeScript files to JavaScript.
4. **Project Structure**: Keep TypeScript source files in a `src` directory and output compiled files into a `dist` directory.

This setup provides an efficient workflow for TypeScript development in a Node.js environment, leveraging `tsx` for development and `tsc` for building production-ready code.

5. Modules

Here's the updated version of the **Modules** documentation that uses **TypeScript** to illustrate how to work with modules in both **CommonJS** and **ES6** formats.

Node.js supports two module systems: **CommonJS** (the original module system used by Node.js) and **ES6 Modules** (the standardized modern module system). This document illustrates how to use modules in **TypeScript**, with examples for both module systems.

What is a Module in Node.js?

A **module** in Node.js is a reusable block of code that can be imported into other parts of your application. Modules can contain functions, objects, or entire classes that are encapsulated for reuse across multiple files.

Key points:

- **CommonJS**: The traditional module system for Node.js.
- **ES6 Modules**: A modern, standardized JavaScript module system.
- **TypeScript**: TypeScript adds static typing to JavaScript, enhancing modules with strong types.

Built-in Modules

Node.js provides built-in modules that offer useful utilities such as working with files, creating HTTP servers, handling paths, and more. You can include and use these modules directly in TypeScript.

Here's how you can use built-in modules in TypeScript:

```
import * as http from 'http';
import * as fs from 'fs';

const server = http.createServer((req, res) => {
  fs.readFile('index.html', (err, data) => {
    if (err) {
      res.writeHead(404, { 'Content-Type': 'text/plain' });
```

```
      res.end('File Not Found');
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    }
  });
});

server.listen(8080, () => {
  console.log('Server running at http://localhost:8080/');
});
```

Include Modules

TypeScript supports both CommonJS and ES6 module systems. Here's how to include modules in each system.

1. CommonJS Syntax in TypeScript

You can use `require()` to import modules in a **CommonJS** format:

```
const http = require('http');  // CommonJS syntax
```

2. ES6 Module Syntax in TypeScript

You can use `import` in TypeScript to leverage **ES6 Modules**:

```
import * as http from 'http';  // ES6 module syntax
```

TypeScript automatically understands the syntax and compiles it into appropriate module systems based on your configuration.

Create Your Own Modules

In TypeScript, you can create custom modules in both **CommonJS** and **ES6** formats. When using TypeScript, you'll also benefit from static typing and type definitions for better clarity and error-checking.

1. Create a CommonJS Module in TypeScript

In **CommonJS**, you define and export functions or objects using `module.exports`. TypeScript supports this syntax and compiles it accordingly.

Example: CommonJS Module in TypeScript

mathOperations.ts:

```
function add(a: number, b: number): number {
  return a + b;
}

function subtract(a: number, b: number): number {
  return a - b;
}

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

In this example, the `add` and `subtract` functions are defined and exported using `module.exports`. You can include these functions in other files using `require()`.

2. Create an ES6 Module in TypeScript

With **ES6 modules**, TypeScript allows you to use the `export` keyword to export functionality. This is more modern and aligns with JavaScript's ES6 standards.

Example: ES6 Module in TypeScript

mathOperations.ts:

```
export function add(a: number, b: number): number {
  return a + b;
}

export function subtract(a: number, b: number): number {
```

```
  return a - b;
}
```

Here, the `export` keyword is used to expose the `add` and `subtract` functions to other files.

Include Your Own Module

Once you've created your custom module, you can include it in other files. TypeScript supports importing both **CommonJS** and **ES6** modules.

1. Including a CommonJS Module in TypeScript

To include a **CommonJS** module in TypeScript, use `require()`:

app.ts:

```
const math = require('./mathOperations');

const result = math.add(10, 5);
console.log(`Addition result: ${result}`);
```

In this example, we use `require()` to import the `mathOperations.ts` file and then use the `add` function from the imported module.

2. Including an ES6 Module in TypeScript

To include an **ES6 module** in TypeScript, use `import`:

app.ts:

```
import { add, subtract } from './mathOperations';

console.log(`Addition: ${add(10, 5)}`);
console.log(`Subtraction: ${subtract(10, 5)}`);
```

Here, we use the `import` syntax to bring in the `add` and `subtract` functions from `mathOperations.ts`. TypeScript's static typing ensures that only the exported members are accessible, and type checking is enforced.

Folder Structure:

Your project might have the following folder structure when working with modules in TypeScript:

```
/my-project
  ├── /modules
  │      └── mathOperations.ts
  ├── app.ts
  └── tsconfig.json
```

In this structure, you can import your custom modules like this:

```
import { add } from './modules/mathOperations';  // ES6
```

```
const math = require('./modules/mathOperations');  // CommonJS
```

tsconfig.json Configuration:

To ensure TypeScript compiles your code correctly, you'll need a `tsconfig.json` file, which defines TypeScript compiler options, including how to handle modules.

Here's an example configuration:

tsconfig.json:

```
{
  "compilerOptions": {
     "target": "ES6", /* Set the JavaScript language version for emitted
JavaScript and include compatible library declarations. */
     "module": "CommonJS", /* Specify what module code is generated. */
     "rootDir": "./src", /* Specify the root folder within your source
files. */
     "outDir": "./dist", /* Specify an output folder for all emitted
files. */
     "forceConsistentCasingInFileNames": true, /* Ensure that casing is
correct in imports. */
     "esModuleInterop": true, /* Emit additional JavaScript to ease
support for importing CommonJS modules. This enables
'allowSyntheticDefaultImports' for type compatibility. */
     "strict": true, /* Enable all strict type-checking options. */
```

```
    "skipLibCheck": true /* Skip type checking all .d.ts files. */
  },
  "include": ["./**/*.ts"],
  "exclude": ["node_modules", "dist"]
}
```

This configuration ensures that TypeScript compiles `.ts` files into the `dist` directory, and it compiles based on the module system you specify (`commonjs` or `esnext` for ES6).

Summary:

1. **What is a Module?**
   - A reusable block of code in Node.js. Modules help organize code efficiently.
2. **Built-in Modules**:
   - Node.js provides various built-in modules like `http`, `fs`, and `path`.
3. **Include Modules**:
   - Use `require()` for **CommonJS** modules.
   - Use `import` for **ES6** modules.
4. **Create Your Own Modules**:
   - **CommonJS**: Use `module.exports` to export code.
   - **ES6**: Use the `export` keyword to export code.
5. **Include Your Own Module**:
   - **CommonJS**: Use `require()` to import modules.
   - **ES6**: Use `import` to bring in modules.
6. **TypeScript Integration**:
   - TypeScript adds static typing to both **CommonJS** and **ES6** modules.

- Use `tsconfig.json` to configure the TypeScript compiler based on your preferred module system.

6. Node.js Globals

Node.js provides several built-in global objects. These are available globally and don't require importing any modules. They can be accessed from any part of your application. Understanding these global objects is crucial for writing effective and efficient Node.js applications.

What are Globals?

In Node.js, **Globals** are objects or functions that are available throughout the entire runtime without the need to import or require any additional modules. They can be accessed directly within any module or file without needing to reference them explicitly.

However, it's important to note that Node.js aims to minimize the use of global objects as it encourages modularity and explicit dependencies. Using global objects should be done with caution to avoid issues with maintainability and scope.

Common Node.js Global Objects

The following are some of the most commonly used global objects in Node.js:

i. __dirname

The `__dirname` global object represents the directory name of the current module (i.e., the file containing the running code). It provides the absolute path to the directory where the currently executing file is located.

This is especially useful for handling file paths dynamically. For example, you may need to access resources or files relative to your module, no matter where your application is being run from.

```
import * as path from 'path';
import * as fs from 'fs';

// Example: Use __dirname to handle file paths dynamically
const filePath = path.join(__dirname, 'data', 'example.txt');

// Checking if the file exists before reading
if (fs.existsSync(filePath)) {
```

```
  const fileContent = fs.readFileSync(filePath, 'utf-8');
  console.log('File Content:', fileContent);
} else {
  console.log(`File not found at ${filePath}`);
}

// Output __dirname value
console.log('Current directory:', __dirname);
```

- **Use case**: File path operations that need the relative path from the current module's directory, such as reading or writing files within the application.

ii. __filename

The `__filename` global object provides the absolute path of the current file being executed, including the file name itself. It can be useful in logging, debugging, or when you need to dynamically work with the path of the current script.

```
import * as path from 'path';

// Example: __filename in action
console.log('Full file path:', __filename);

// Extract directory name from __filename
const currentDir = path.dirname(__filename);
console.log('Directory name:', currentDir);

// Extract base file name (e.g., 'index.js') from __filename
const baseFileName = path.basename(__filename);
console.log('Base file name:', baseFileName);

// Extract file extension (e.g., '.js')
const fileExtension = path.extname(__filename);
console.log('File extension:', fileExtension);

// Construct a path relative to this file
```

```
const relativePath = path.join(currentDir, 'config', 'settings.json');
console.log('Relative file path:', relativePath);
```

- **Use case**: Dynamic operations involving the file path of the current file, like logging its location, or performing actions based on the current file's directory.

iii. exports

In Node.js, `exports` is an object that is used to expose functionalities from a module to other files. It's used to define the public interface of the module. You can export variables, functions, or classes to be available for other files using the `exports` object.

```
// Example: Using exports in a module
export const sayHello = () => {
  console.log('Hello, World!');
};
```

In another file, you can import this function as follows:

```
import { sayHello } from './moduleFile';
sayHello();  // Output: Hello, World!
```

- **Use case**: Sharing functions, variables, or classes from one file to be used in other parts of your application.

iv. module

The `module` object represents the current module and provides information about it. It's an object that contains properties like `exports` (for exporting the module's content) and `filename` (to get the absolute path of the module file).

```
// Example: Inspecting the module object
console.log(module);  // Output: Module details (such as exports, id, filename)
```

Using `module.exports`, you can overwrite the default exports object.

v. require()

require() is a built-in function used to include modules. It is used to load and use modules from other files or node modules. In TypeScript, you typically use the import syntax, but require() is still relevant in some use cases like when working with older JavaScript modules.

```
// Example: Using require() to load a module
const fs = require('fs');
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);
```

In TypeScript, the equivalent of require() is typically:

```
import * as fs from 'fs';
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);
```

ES Global Objects

In addition to the Node.js-specific global objects, Node.js also exposes many of the ECMAScript (ES) global objects that are part of the JavaScript language itself. These objects are available in all JavaScript environments (browser, server-side, etc.), and they include objects like:

- **console**: Used for printing information, warnings, or errors to the terminal.

  ```
  console.log('This is a message');
  console.error('This is an error');
  ```

- **setTimeout()/setInterval()**: Used to execute a function after a specified delay or repeatedly at intervals.

  ```
  setTimeout(() => {
    console.log('This message appears after 2 seconds');
  }, 2000);

  setInterval(() => {
  ```

```
  console.log('This message repeats every 3 seconds');
}, 3000);
```

- `Promise`: A global object that represents an asynchronous operation that may complete at some point in the future.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise resolved!');
  }, 1000);
});

promise.then(message => {
  console.log(message);  // Output: Promise resolved!
});
```

- `Buffer`: Available globally in Node.js, the `Buffer` class is used to handle binary data.

```
const buffer = Buffer.from('Hello, world!', 'utf8');
console.log(buffer.toString());  // Output: Hello, world!
```

- `process`: A Node.js-specific object that provides information about the current process, as well as methods for controlling it.

```
console.log(process.env);  // Outputs environment variables
process.exit(1);  // Terminates the process with an error code
```

These objects are available globally in Node.js as part of the underlying JavaScript language and the Node.js runtime.

This section on **Node.js Globals** introduces developers to essential global objects they will frequently encounter when building applications in Node.js. It provides a solid foundation for understanding the Node.js environment and its integration with ECMAScript features.

7. HTTP Module

The HTTP module in Node.js provides the functionality to create web servers and handle HTTP requests and responses. Additionally, Node.js supports HTTP/2 and HTTPS for improved performance and secure communication. Each protocol offers unique features and is widely used to handle web traffic.

HTTP

The HTTP (Hypertext Transfer Protocol) module in Node.js is used to create and manage basic web servers. It provides functionality to create a server, listen for incoming requests, and respond with data. This is one of the core modules in Node.js, allowing developers to set up a web server with minimal setup.

Key Features:

- Handling basic HTTP requests and responses (GET, POST, PUT, DELETE, etc.)
- Managing request headers and response data
- Easy access to URL and query parameters

Example: Basic HTTP Server in Node.js with TypeScript

```
import * as http from 'http';

// Create a basic HTTP server
const server = http.createServer((req, res) => {
  // Set the response header and status code
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Send a response based on the URL
  if (req.url === '/') {
    res.end('Welcome to the homepage!');
  } else if (req.url === '/about') {
    res.end('About us page');
  } else {
    res.end('404 - Page not found');
```

```
  }
});

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

Explanation:

- **`http.createServer()`**: Creates an HTTP server that listens for incoming requests.
- **`req.url`**: Captures the URL of the incoming request, which allows routing to different responses.
- **`res.writeHead()`**: Sets the HTTP status code and headers before sending the response.
- **`res.end()`**: Ends the response and sends it to the client.

Use Cases:

- Lightweight web servers for testing or small applications.
- Handling basic APIs and static file serving.

HTTP/2

HTTP/2 is a major revision of the HTTP protocol that introduces several improvements over HTTP/1.1, including better performance and reduced latency. Node.js provides native support for HTTP/2 via the `http2` module. Key benefits include multiplexing (allowing multiple requests and responses to be sent over a single connection), header compression, and server push.

Key Features:

- **Multiplexing**: Multiple requests and responses over a single TCP connection.
- **Header Compression**: Compresses HTTP headers to reduce overhead.

- **Server Push**: The server can push resources (like CSS, JS files) to the client before the client even requests them.

Example: Basic HTTP/2 Server in Node.js with TypeScript

```
import * as http2 from 'http2';
import * as fs from 'fs';

// Load SSL certificates for secure connection
// learn how to generate server-key.pem & server-cert.pem from 7.1
Install OpenSSL on Windows
const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem')
};

// Create an HTTP/2 server
const server = http2.createSecureServer(options);

server.on('stream', (stream, headers) => {
  // Respond to HTTP/2 requests
  const path = headers[':path'];

  if (path === '/') {
    stream.respond({
      'content-type': 'text/html',
      ':status': 200
    });
    stream.end('<h1>Welcome to the HTTP/2 Server</h1>');
  } else if (path === '/about') {
    stream.respond({
      'content-type': 'text/html',
      ':status': 200
    });
    stream.end('<h1>About Us</h1>');
  } else {
    stream.respond({
      ':status': 404
```

```
    });
    stream.end('404 - Not Found');
  }
});

// Listen on port 8443
server.listen(8443, () => {
  console.log('HTTP/2 server is running on https://localhost:8443');
});
```

Explanation:

- **`http2.createSecureServer()`**: Creates a secure HTTP/2 server using SSL certificates.
- **`stream.respond()`**: Sends headers and status code for each incoming request.
- **`stream.end()`**: Ends the response and sends data to the client.

Use Cases:

- Modern web applications that require faster loading times and efficient resource loading.
- Applications that need to support Server Push for improved user experience.

HTTPS

HTTPS (Hypertext Transfer Protocol Secure) is an extension of HTTP that provides secure communication over the network using SSL/TLS encryption. In Node.js, the `https` module is used to create secure servers. HTTPS ensures that all data exchanged between the server and the client is encrypted, protecting sensitive information from being intercepted.

Key Features:

- **SSL/TLS Encryption**: Secure communication using certificates.

- **Secure Data Transmission:** Protects sensitive information (like login credentials, payment info).

Example: Basic HTTPS Server in Node.js with TypeScript

```
import * as https from 'https';
import * as fs from 'fs';

// Load SSL certificates
const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem')
};

// Create a basic HTTPS server
const server = https.createServer(options, (req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Respond to requests based on URL
  if (req.url === '/') {
    res.end('Welcome to the secure homepage!');
  } else if (req.url === '/about') {
    res.end('About us page (secure)');
  } else {
    res.end('404 - Secure page not found');
  }
});

// Listen on port 443 (default for HTTPS)
server.listen(443, () => {
  console.log('HTTPS server is running on https://localhost');
});
```

Explanation:

- **SSL Certificates**: The `options` object includes the private key and certificate required for secure connections.

- `https.createServer()`: Creates an HTTPS server using SSL/TLS encryption.
- **Port 443**: The standard port for HTTPS, ensuring that all communication is encrypted.

Use Cases:

- Websites or APIs that need to handle sensitive data securely, such as login systems, financial transactions, or private communications.
- Any application that wants to ensure security and privacy by default.

Summary

- **HTTP**: Ideal for building simple web servers or APIs where performance and security are not primary concerns. However, it is commonly used for local development and testing.
- **HTTP/2**: Offers significant performance improvements over HTTP/1.1, including multiplexing and server push, which makes it ideal for modern web applications.
- **HTTPS**: Provides a secure communication layer with SSL/TLS encryption, essential for any application handling sensitive user data. It ensures the integrity and privacy of data exchanged between client and server.

These modules enable developers to create flexible, high-performance, and secure applications for various use cases.

7.1 Install OpenSSL on Windows

You can generate the `server-key.pem` and `server-cert.pem` files on Windows using OpenSSL. Here's a step-by-step guide to generate the self-signed certificates on Windows.

Configuring OpenSSL on Windows involves downloading and installing the appropriate binaries, setting up environment variables, and verifying the installation. Below is a step-by-step guide to help you through the process.

1. Understanding OpenSSL

OpenSSL is an open-source toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is widely used for securing communications over computer networks and for managing cryptographic keys and certificates.

2. Downloading OpenSSL

OpenSSL doesn't provide official Windows binaries, but trusted third-party providers offer precompiled versions:

Shining Light Productions

1. **Visit the Website:**
   - Go to Shining Light Productions OpenSSL Binaries (https://slproweb.com/products/Win32OpenSSL.html).
2. **Choose the Right Installer:**
   - **Version:** Select the latest stable version compatible with your needs.
   - **Architecture:** Choose between Win32 (32-bit) or Win64 (64-bit) based on your system.
   - **Light vs. Full:** The "Light" version includes the essential components, while the "Full" version contains additional tools.
3. **Download the Installer:**

- Click on the appropriate `.exe` installer to download.

3. Installing OpenSSL

Using Shining Light Productions Installer

1. **Run the Installer:**
   - Double-click the downloaded `.exe` file.
2. **Follow the Installation Wizard:**
   - **License Agreement:** Accept the terms and proceed.
   - **Destination Directory:** Choose the installation path (default is usually fine).
   - **Select Components:**
     - Ensure that the "OpenSSL binaries" and "OpenSSL libraries" are selected.
     - Optionally, include the "Copy OpenSSL DLLs to the Windows system directory" if desired.
   - **Install:** Click `Install` to begin the process.
3. **Complete Installation:**
   - Finish the wizard once installation is complete.

5. Configuring Environment Variables

To use OpenSSL from the Command Prompt, you need to add its `bin` directory to the system `PATH`.

1. **Locate OpenSSL Installation Directory:**
   - Typically, it's installed in `C:\OpenSSL-Win64` or `C:\OpenSSL-Win32`.
2. **Add to PATH:**
   - **Open System Properties:**

    - Press `Win + X`, select `System`, then `Advanced system settings`.
  - **Environment Variables:**
    - Click on `Environment Variables`.
  - **Edit PATH:**
    - Under `System variables`, find and select `Path`, then click `Edit`.
  - **Add New Entry:**
    - Click `New` and enter the path to the OpenSSL `bin` directory, e.g., `C:\OpenSSL-Win64\bin`.
  - **Save Changes:**
    - Click `OK` on all dialogs to apply the changes.
3. **Verify PATH Update:**
  - Open a new Command Prompt window and run:

```
echo %PATH%
```

  - Ensure the OpenSSL `bin` path is listed.

6. Verifying the Installation

1. **Open Command Prompt:**
  - Press `Win + R`, type `cmd`, and press `Enter`.
2. **Check OpenSSL Version:**
  - Run the command:

```
openssl version
```

  - You should see output similar to:

```
OpenSSL 1.1.1k  25 Mar 2021
```

3. **Basic Test:**

- Generate a test key:

```
openssl genrsa -out test.key 2048
```

- If the command executes without errors and `test.key` is created, OpenSSL is functioning correctly.

7. Common Configuration Tasks

Generating a Private Key and Certificate Signing Request (CSR)

Step 1. Generate Private Key:

```
openssl genrsa -out myprivate.key 2048
```

Step 2. Generate CSR:

```
openssl req -new -key myprivate.key -out myrequest.csr
```

```
- Follow the prompts to enter information like Country, State,
Organization, etc.
```

Step 3: Generate the Private Key and Self-Signed Certificate

1. Generate the Private Key (server-key.pem)

Run the following command to generate a 2048-bit RSA private key:

```
openssl genrsa -out server-key.pem 2048
```

This command will create the `server-key.pem` file, which contains the private key.

2. Generate the Certificate Signing Request (CSR)

Next, generate a Certificate Signing Request (CSR) using the private key:

```
openssl req -new -key server-key.pem -out server-csr.pem
```

When prompted, enter the required information. Here's an example:

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:MyCompany
Organizational Unit Name (eg, section) []:Development
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:admin@mycompany.com
```

For the **Common Name**, use `localhost` if you're generating the certificate for local development.

3. Generate the Self-Signed Certificate (server-cert.pem)

Finally, generate a self-signed certificate using the CSR and the private key:

```
openssl x509 -req -in server-csr.pem -signkey server-key.pem -out
server-cert.pem -days 365
```

This will create the `server-cert.pem` file, which is valid for 365 days.

Step 4: Use the Certificates in Your Node.js Application

You should now have the following files in your working directory:

- `server-key.pem`: The private key.
- `server-cert.pem`: The self-signed certificate.

These files can be used in your Node.js HTTPS and HTTP/2 applications.

Final Notes:

- You can use these certificates for local development. When using them in a browser, it will show a security warning because the certificates are self-signed and not trusted by default.

- In production, you'll need to obtain SSL certificates from a trusted Certificate Authority (CA), such as Let's Encrypt or a paid provider like DigiCert or GlobalSign.

8. File System Module

Node.js provides a built-in module called `fs` (File System), which allows you to interact with the file system on your server. With this module, you can perform various file operations such as reading, writing, updating, deleting, and renaming files. You can also use Node.js as a file server to handle file requests from clients.

Node.js as a File Server

Using Node.js as a file server, you can read and serve files to users, or allow file manipulations like creating, updating, deleting, or renaming files. The `fs` module provides both asynchronous and synchronous methods for these operations. It's important to use asynchronous methods whenever possible to prevent blocking the event loop.

Below are some common operations you can perform with the `fs` module:

i. Read Files

To read the contents of a file, you can use the `fs.readFile()` method asynchronously. This method reads the entire file into memory.

Example: Reading a File Asynchronously

```
import * as fs from 'fs';
import * as http from 'http';
import * as path from 'path';

// Define the path to the data directory and the file
const dataDir = path.join(__dirname, 'data');
const filePath = path.join(dataDir, 'example.txt');

// Ensure the data directory exists
if (!fs.existsSync(dataDir)) {
  fs.mkdirSync(dataDir, { recursive: true });
  console.log('Data directory created.');
}

// Example: Serving a file using Node.js
```

```
const server = http.createServer((req, res) => {
  // Reading the requested file asynchronously
  fs.readFile(filePath, 'utf-8', (err, data) => {
    if (err) {
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      res.end('File not found');
    } else {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(data);
    }
  });
});

// Listen on port 3000
server.listen(3000, () => {
  console.log('Server is running at http://localhost:3000');
});
```

Explanation:

- **fs.existsSync()**: Checks if the data directory exists.
- **fs.mkdirSync()**: Creates the data directory if it doesn't exist.
- **path.join()**: Constructs the file paths in a cross-platform manner.

Use Case:

- Serve files like HTML, CSS, JavaScript, or other resources to clients in a web server.

ii. Create Files

You can create new files in Node.js using fs.writeFile() or fs.appendFile(). The writeFile() method creates a new file and writes content to it, overwriting the file if it already exists, while appendFile() adds data to the end of an existing file or creates the file if it doesn't exist.

Example: Creating a New File

```
import * as fs from 'fs';
import * as path from 'path';

const dataDir = path.join(__dirname, 'data');
const filePath = path.join(dataDir, 'newfile.txt');

// Ensure the data directory exists
if (!fs.existsSync(dataDir)) {
  fs.mkdirSync(dataDir, { recursive: true });
}

// Creating and writing to a file
fs.writeFile(filePath, 'Hello, this is a newly created file!', (err) =>
{
  if (err) throw err;
  console.log('File created and written successfully');
});
```

Example: Appending Data to a File

```
import * as fs from 'fs';
import * as path from 'path';

const dataDir = path.join(__dirname, 'data');
const filePath = path.join(dataDir, 'newfile.txt');

// Appending data to an existing file
fs.appendFile(filePath, '\nAppending more content to the file.', (err)
=> {
  if (err) throw err;
  console.log('Content appended successfully');
});
```

Use Case:

- Creating new files for logging, storing data, or dynamically generating files.

iii. Update Files

You can update a file by either writing to it again (overwriting the content) using `fs.writeFile()` or appending new content to the existing file using `fs.appendFile()`.

Example: Overwriting a File

```
import * as fs from 'fs';
import * as path from 'path';

const dataDir = path.join(__dirname, 'data');
const filePath = path.join(dataDir, 'newfile.txt');

// Overwriting the content of a file
fs.writeFile(filePath, 'This content replaces the old one.', (err) => {
  if (err) throw err;
  console.log('File content overwritten successfully');
});
```

Example: Appending to a File

```
import * as fs from 'fs';
import * as path from 'path';

const dataDir = path.join(__dirname, 'data');
const filePath = path.join(dataDir, 'newfile.txt');

// Appending content to a file
fs.appendFile(filePath, '\nMore content being added.', (err) => {
  if (err) throw err;
  console.log('File updated with new content');
});
```

Use Case:

- Updating logs, modifying data files, or appending new data to reports.

iv. Delete Files

You can delete files using the `fs.unlink()` method.

Example: Deleting a File

```
import * as fs from 'fs';
import * as path from 'path';

const dataDir = path.join(__dirname, 'data');
const filePath = path.join(dataDir, 'newfile.txt');

// Deleting a file
fs.unlink(filePath, (err) => {
  if (err) throw err;
  console.log('File deleted successfully');
});
```

Use Case:

- Removing temporary files or cleaning up old data files that are no longer needed.

v. Rename Files

To rename a file, use the `fs.rename()` method.

Example: Renaming a File

```
import * as fs from 'fs';
import * as path from 'path';

const dataDir = path.join(__dirname, 'data');
const oldFilePath = path.join(dataDir, 'oldfile.txt');
const newFilePath = path.join(dataDir, 'newfile.txt');

// Renaming a file
fs.rename(oldFilePath, newFilePath, (err) => {
  if (err) throw err;
```

```
  console.log('File renamed successfully');
});
```

Use Case:

- Organizing files, managing file versions, or moving files within the same directory.

Complete Example: Node.js HTTPS File Server with CRUD Operations

Below is an enhanced example that:

- Reads and writes files from the `./src/data` folder.
- Creates the `data` folder when the server starts if it doesn't exist.
- Provides endpoints for all CRUD operations.
- Demonstrates how to interact with the server using HTTP methods.

Project Structure

```
HTTPS/
├── src/
│   ├── data/
│   └── index.ts
├── package.json
├── tsconfig.json
└── node_modules/
```

Step-by-Step Implementation

1. **Initialize the Project**

   Ensure you have initialized your project and installed necessary dependencies.

   ```
   npm init -y
   npm install typescript tsx --save-dev
   ```

2. **Configure TypeScript (`tsconfig.json`)**

Create a `tsconfig.json` in your project root with the following content:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "moduleResolution": "node",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "skipLibCheck": true
  },
  "include": ["src"],
  "exclude": ["node_modules"]
}
```

3. **Update `package.json` Scripts**

   Add the following scripts to your `package.json`:

```
{
  "name": "https-file-server",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "tsx src/index.ts",
    "build": "tsc",
    "serve": "node dist/index.js"
  },
  "devDependencies": {
    "typescript": "^4.0.0",
    "tsx": "^4.19.1"
  }
}
```

4. **Create the HTTPS Server (`src/index.ts`)**

```
import * as https from 'https';
import * as fs from 'fs';
import * as path from 'path';
import { IncomingMessage, ServerResponse } from 'http';
import { URL } from 'url';

// Define the path to the data directory
const dataDir = path.join(__dirname, 'data');

// Ensure the data directory exists
if (!fs.existsSync(dataDir)) {
  fs.mkdirSync(dataDir, { recursive: true });
  console.log('Data directory created.');
}

// Define SSL certificates
const options = {
  key: fs.readFileSync(path.join(__dirname, '..', 'server-key.pem')),
  cert: fs.readFileSync(path.join(__dirname, '..', 'server-cert.pem'))
};

// Create HTTPS server
const server = https.createServer(options, (req: IncomingMessage, res:
ServerResponse) => {
  const parsedUrl = new URL(req.url || '',
`https://${req.headers.host}`);
  const pathname = parsedUrl.pathname;

  // Set CORS headers (optional, for API usage)
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT,
DELETE, OPTIONS');
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type');

  // Handle preflight requests
  if (req.method === 'OPTIONS') {
    res.writeHead(204);
```

```
    res.end();
    return;
  }

  // Routing
  if (pathname === '/' && req.method === 'GET') {
    // Read the empty file
    const filePath = path.join(dataDir, 'data.txt');
    fs.readFile(filePath, 'utf-8', (err, data) => {
      if (err) {
        // If file doesn't exist, send an empty response
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ message: 'File is empty or does not
exist.' }));
      } else {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ content: data }));
      }
    });
  } else if (pathname === '/create' && req.method === 'POST') {
    // Create or overwrite the file with posted data
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', () => {
      const filePath = path.join(dataDir, 'data.txt');
      fs.writeFile(filePath, body, (err) => {
        if (err) {
          res.writeHead(500, { 'Content-Type': 'application/json' });
          res.end(JSON.stringify({ error: 'Failed to write data.' }));
        } else {
          res.writeHead(201, { 'Content-Type': 'application/json' });
          res.end(JSON.stringify({ message: 'Data saved successfully.'
}));
        }
      });
    });
```

```
  } else if (pathname === '/read' && req.method === 'GET') {
    // Read the file with data
    const filePath = path.join(dataDir, 'data.txt');
    fs.readFile(filePath, 'utf-8', (err, data) => {
      if (err) {
        res.writeHead(404, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ error: 'File not found.' }));
      } else {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ content: data }));
      }
    });
  } else if (pathname === '/update' && req.method === 'PUT') {
    // Update the file by appending data
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', () => {
      const filePath = path.join(dataDir, 'data.txt');
      fs.appendFile(filePath, body, (err) => {
        if (err) {
          res.writeHead(500, { 'Content-Type': 'application/json' });
          res.end(JSON.stringify({ error: 'Failed to update data.'
}));
        } else {
          res.writeHead(200, { 'Content-Type': 'application/json' });
          res.end(JSON.stringify({ message: 'Data updated
successfully.' }));
        }
      });
    });
  } else if (pathname === '/delete' && req.method === 'DELETE') {
    // Delete the file
    const filePath = path.join(dataDir, 'data.txt');
    fs.unlink(filePath, (err) => {
      if (err) {
        res.writeHead(404, { 'Content-Type': 'application/json' });
```

```
        res.end(JSON.stringify({ error: 'File not found.' }));
      } else {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ message: 'File deleted successfully.'
}));
      }
    });
  } else if (pathname === '/rename' && req.method === 'PUT') {
    // Rename the file
    const oldPath = path.join(dataDir, 'data.txt');
    const newPath = path.join(dataDir, 'renamedData.txt');
    fs.rename(oldPath, newPath, (err) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ error: 'Failed to rename file.' }));
      } else {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ message: 'File renamed successfully.'
}));
      }
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ error: 'Route not found.' }));
  }
});

// Listen on port 8443 (using a non-privileged port for development)
const PORT = process.env.PORT || 8443;
server.listen(PORT, () => {
  console.log(`HTTPS server is running on https://localhost:${PORT}`);
});
```

Explanation:

1. **Directory Setup:**

- `dataDir`: Specifies the path to the `data` folder within `./src/data`.
- **Directory Creation:** Checks if the `data` directory exists; if not, it creates it using `fs.mkdirSync()`.

2. **SSL Certificates:**
   - Ensure that your `server-key.pem` and `server-cert.pem` files are located in the project root (one level above `src`).
   - Adjust the paths accordingly if your certificates are stored elsewhere.
3. **Server Routing:**
   - **`/` (GET):** Reads and returns the content of an empty or non-existent `data.txt` file.
   - **`/create` (POST):** Receives data in the request body and saves it to `data.txt`.
   - **`/read` (GET):** Reads and returns the content of `data.txt`.
   - **`/update` (PUT):** Appends data to `data.txt`.
   - **`/delete` (DELETE):** Deletes `data.txt`.
   - **`/rename` (PUT):** Renames `data.txt` to `renamedData.txt`.
4. **CORS Headers:**
   - Added CORS headers to allow cross-origin requests, which is useful if you interact with the API from different origins.
5. **Port Configuration:**
   - The server listens on port `8443` by default to avoid permission issues. You can change this by setting the `PORT` environment variable.

Interacting with the Server

You can interact with the server using tools like **curl**, **Postman**, or any HTTP client.

1. **Read the Empty File (`/` Endpoint)**

   Request:

```
curl -k https://localhost:8443/
```

Response:

```
{
  "message": "File is empty or does not exist."
}
```

2. **Create/Overwrite the File (`/create` Endpoint)**

   Request:

```
curl -k -X POST https://localhost:8443/create -d "This is the initial content of the file."
```

   Response:

```
{
  "message": "Data saved successfully."
}
```

3. **Read the File with Data (`/read` Endpoint)**

   Request:

```
curl -k https://localhost:8443/read
```

   Response:

```
{
  "content": "This is the initial content of the file."
}
```

4. **Update the File by Appending Data (`/update` Endpoint)**

   Request:

```
curl -k -X PUT https://localhost:8443/update -d " Appended content."
```

Response:

```
{
  "message": "Data updated successfully."
}
```

5. **Read the Updated File (`/read` Endpoint)**

   Request:

```
curl -k https://localhost:8443/read
```

   Response:

```
{
  "content": "This is the initial content of the file. Appended
content."
}
```

6. **Delete the File (`/delete` Endpoint)**

   Request:

```
curl -k -X DELETE https://localhost:8443/delete
```

   Response:

```
{
  "message": "File deleted successfully."
}
```

7. **Rename the File (`/rename` Endpoint)**

   Note: Ensure that `data.txt` exists before attempting to rename it.

   Request:

```
curl -k -X PUT https://localhost:8443/rename
```

Response:

```
{
  "message": "File renamed successfully."
}
```

Post-Rename Read:

After renaming, attempting to read `data.txt` will result in a 404 error unless you update the endpoints accordingly.

Security Considerations

- **Self-Signed Certificates:** The above setup uses self-signed certificates, which are suitable for development and testing. Browsers will display a security warning when accessing the server. For production environments, obtain certificates from a trusted Certificate Authority (CA) like Let's Encrypt (https://letsencrypt.org/).
- **Data Validation:** The current implementation does not include data validation. In a production setting, validate and sanitize incoming data to prevent security vulnerabilities.
- **Error Handling:** Enhanced error handling can be implemented to provide more detailed responses and logging.

Additional CRUD Operations Example

To further expand the server's capabilities, you can add more granular CRUD operations based on file names or manage multiple files within the `data` directory.

Example: CRUD Operations with Dynamic File Names

```
import * as https from 'https';
import * as fs from 'fs';
import * as path from 'path';
import { IncomingMessage, ServerResponse } from 'http';
import { URL } from 'url';
```

```
// Define the path to the data directory
const dataDir = path.join(__dirname, 'data');

// Ensure the data directory exists
if (!fs.existsSync(dataDir)) {
  fs.mkdirSync(dataDir, { recursive: true });
  console.log('Data directory created.');
}

// Define SSL certificates
const options = {
  key: fs.readFileSync(path.join(__dirname, '..', 'server-key.pem')),
  cert: fs.readFileSync(path.join(__dirname, '..', 'server-cert.pem'))
};

// Utility function to parse JSON body
const parseRequestBody = (req: IncomingMessage): Promise<any> => {
  return new Promise((resolve, reject) => {
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', () => {
      try {
        const parsed = JSON.parse(body);
        resolve(parsed);
      } catch (err) {
        reject(err);
      }
    });
  });
};

// Create HTTPS server
const server = https.createServer(options, async (req: IncomingMessage,
res: ServerResponse) => {
  const parsedUrl = new URL(req.url || '',
```

```
`https://${req.headers.host}`);
  const pathname = parsedUrl.pathname;
  const method = req.method;

  // Set CORS headers (optional, for API usage)
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE,
OPTIONS');
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type');

  // Handle preflight requests
  if (method === 'OPTIONS') {
    res.writeHead(204);
    res.end();
    return;
  }

  // Route: /file?name=filename.txt
  const fileName = parsedUrl.searchParams.get('name');
  if (!fileName) {
    res.writeHead(400, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ error: 'File name is required as a query
parameter (e.g., ?name=filename.txt)' }));
    return;
  }

  const filePath = path.join(dataDir, fileName);

  // Routing based on method
  switch (method) {
    case 'GET':
      // Read File
      fs.readFile(filePath, 'utf-8', (err, data) => {
        if (err) {
          res.writeHead(404, { 'Content-Type': 'application/json' });
          res.end(JSON.stringify({ error: 'File not found.' }));
        } else {
          res.writeHead(200, { 'Content-Type': 'application/json' });
```

```
          res.end(JSON.stringify({ content: data }));
        }
      });
      break;

    case 'POST':
      // Create File
      try {
        const body = await parseRequestBody(req);
        fs.writeFile(filePath, body.content, (err) => {
          if (err) {
            res.writeHead(500, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ error: 'Failed to write data.' }));
          } else {
            res.writeHead(201, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ message: 'Data saved successfully.'
}));
          }
        });
      } catch (error) {
        res.writeHead(400, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ error: 'Invalid JSON data.' }));
      }
      break;

    case 'PUT':
      // Update File by Appending
      try {
        const body = await parseRequestBody(req);
        fs.appendFile(filePath, body.content, (err) => {
          if (err) {
            res.writeHead(500, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ error: 'Failed to update data.'
}));
          } else {
            res.writeHead(200, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ message: 'Data updated
successfully.' }));
```

```
        }
      });
    } catch (error) {
      res.writeHead(400, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ error: 'Invalid JSON data.' }));
    }
    break;

  case 'DELETE':
    // Delete File
    fs.unlink(filePath, (err) => {
      if (err) {
        res.writeHead(404, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ error: 'File not found.' }));
      } else {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ message: 'File deleted successfully.'
}));
      }
    });
    break;

  default:
    res.writeHead(405, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ error: 'Method not allowed.' }));
    break;
  }
});

// Listen on port 8443 (using a non-privileged port for development)
const PORT = process.env.PORT || 8443;
server.listen(PORT, () => {
  console.log(`HTTPS server is running on https://localhost:${PORT}`);
});
```

Explanation:

1. **Dynamic File Handling:**
   - **Query Parameter:** The server expects a `name` query parameter to specify the file name (e.g., `?name=example.txt`).
2. **HTTP Methods:**
   - **GET:** Reads the specified file.
   - **POST:** Creates a new file with the provided content.
   - **PUT:** Appends content to the existing file.
   - **DELETE:** Deletes the specified file.
3. **Data Directory:**
   - Ensures the `./src/data` directory exists upon server startup.
4. **JSON Parsing:**
   - Implements a utility function `parseRequestBody` to parse JSON data from the request body.
5. **CORS Support:**
   - Allows cross-origin requests, which is beneficial for API consumption from different domains.

Interacting with the Enhanced Server

1. **Create a New File (`POST /file?name=example.txt`)**

   Request:

```
curl -k -X POST https://localhost:8443/file?name=example.txt \
  -H "Content-Type: application/json" \
  -d '{"content": "This is the initial content."}'
```

   Response:

```
{
  "message": "Data saved successfully."
```

```
}
```

2. **Read the File (`GET /file?name=example.txt`)**

   Request:

   ```
   curl -k https://localhost:8443/file?name=example.txt
   ```

   Response:

   ```
   {
     "content": "This is the initial content."
   }
   ```

3. **Update the File by Appending Data (`PUT /file?name=example.txt`)**

   Request:

   ```
   curl -k -X PUT https://localhost:8443/file?name=example.txt \
     -H "Content-Type: application/json" \
     -d '{"content": " Appended content."}'
   ```

   Response:

   ```
   {
     "message": "Data updated successfully."
   }
   ```

4. **Read the Updated File (`GET /file?name=example.txt`)**

   Request:

   ```
   curl -k https://localhost:8443/file?name=example.txt
   ```

   Response:

   ```
   {
     "content": "This is the initial content. Appended content."
   ```

```
}
```

5. **Delete the File (`DELETE /file?name=example.txt`)**

   Request:

   ```
   curl -k -X DELETE https://localhost:8443/file?name=example.txt
   ```

   Response:

   ```
   {
     "message": "File deleted successfully."
   }
   ```

6. **Error Handling:**

   - **Missing File Name:**

     Request:

     ```
     curl -k https://localhost:8443/file
     ```

     Response:

     ```
     {
       "error": "File name is required as a query parameter (e.g., ?
     name=filename.txt)"
     }
     ```

   - **File Not Found:**

     Request:

     ```
     curl -k https://localhost:8443/file?name=nonexistent.txt
     ```

     Response:

     ```
     {
       "error": "File not found."
     ```

```
}
```

- **Invalid JSON Data:**

  Request:

  ```
  curl -k -X POST https://localhost:8443/file?name=example.txt \
    -H "Content-Type: application/json" \
    -d 'Invalid JSON'
  ```

  Response:

  ```
  {
    "error": "Invalid JSON data."
  }
  ```

Security Considerations

- **Input Validation:**
  - Validate the `name` query parameter to prevent directory traversal attacks (e.g., using `../` to access unauthorized directories).
  - Implement checks to ensure that only permitted file types are accessed or manipulated.
- **Authentication & Authorization:**
  - Implement authentication mechanisms to restrict access to the file operations.
  - Use authorization to control what actions authenticated users can perform.
- **Error Messages:**
  - Avoid exposing sensitive information in error messages. Provide generic error responses to prevent information leakage.
- **Rate Limiting:**
  - Implement rate limiting to prevent abuse of the API endpoints.

Final Checklist

Before running your enhanced server, ensure the following:

1. **File Existence:**
   - `src/index.ts` exists and contains the updated TypeScript code.
   - `src/data` directory is created automatically when the server starts.
2. **Dependencies Installed:**
   - Run `npm install` to ensure all dependencies are installed.
3. **TypeScript Configuration:**
   - `tsconfig.json` is properly configured as shown above.
4. **SSL Certificates:**
   - Ensure `server-key.pem` and `server-cert.pem` are placed in the project root (`..` relative to `src/index.ts`).
5. **Scripts Set Up:**
   - `package.json` has the necessary scripts (`start`, `build`, `serve`).
6. **Run the Server Correctly:**
   - Use `npm run start` to execute your server with `tsx`.
7. **Permissions:**
   - Using port `8443` avoids the need for administrative rights. Change the port only if necessary.
8. **Environment Variables:**
   - If using environment variables for configuration, ensure they're correctly set before running the server.

Running the Server

1. **Start the Server:**

   ```
   npm run start
   ```

2. **Expected Output:**

   ```
   Data directory created.
   HTTPS server is running on https://localhost:8443
   ```

3. **Interact with the Server:**

   - **Accessing `/` to Read an Empty File:**

     ```
     curl -k https://localhost:8443/
     ```

     Response:

     ```
     {
       "message": "File is empty or does not exist."
     }
     ```

   - **Creating, Reading, Updating, Deleting, and Renaming Files:**

     - Use the provided endpoints (`/create`, `/read`, `/update`, `/delete`, `/rename`) with appropriate HTTP methods as demonstrated in the examples above.

Troubleshooting Tips

If you encounter issues while running the server, consider the following troubleshooting steps:

1. "Permission Denied" or "EACCES" Error on Port 443

- **Cause:** Binding to port `443` requires elevated privileges.
- **Solution:**
  - **Use a Higher Port:** Change to a port like `8443` for development.

- **Run as Administrator:** If you must use port `443`, run your terminal or command prompt with administrative rights.

2. "Module Not Found" Error

- **Cause:** Node.js (or `tsx`) can't locate the specified module or file.
- **Solution:**
  - **Verify File Paths:** Ensure that all file paths in your code are correct.
  - **Check File Extensions:** Ensure that your TypeScript files have the `.ts` extension.
  - **Run from Correct Directory:** Execute the server from the project root where `package.json` is located.

3. SSL Certificate Errors in Browser

- **Cause:** Self-signed certificates are not trusted by browsers.
- **Solution:**
  - For development, proceed by accepting the risk in the browser.
  - For production, obtain certificates from a trusted CA like Let's Encrypt (https://letsencrypt.org/).

4. JSON Parsing Errors

- **Cause:** Sending invalid JSON data in the request body.
- **Solution:**
  - Ensure that the data sent in `POST` and `PUT` requests is valid JSON.
  - Use tools like Postman to format your requests correctly.

5. File Not Found Errors

- **Cause:** Attempting to read or manipulate a file that doesn't exist.

- **Solution:**
  - Ensure that the file exists before performing operations.
  - Use appropriate error handling to manage such scenarios gracefully.

6. CORS Issues

- **Cause:** Cross-Origin Resource Sharing (CORS) restrictions.
- **Solution:**
  - Ensure that the server sets the appropriate CORS headers as shown in the example.
  - Adjust the headers based on your security requirements.

Security Best Practices

While implementing file operations, it's crucial to adhere to security best practices to protect your server and data.

1. Input Validation

- **Sanitize File Names:**
  - Prevent directory traversal by validating and sanitizing the `name` query parameter.
  - Reject file names containing `../` or other potentially malicious patterns.
- **Restrict File Types:**
  - Allow only specific file types to be created or manipulated.

2. Authentication & Authorization

- **Implement Authentication:**
  - Use authentication mechanisms (e.g., JWT, OAuth) to restrict access to authorized users.
- **Role-Based Access Control (RBAC):**

  - Define roles and permissions to control what actions authenticated users can perform.

3. Error Handling

- **Avoid Detailed Error Messages:**
  - Do not expose stack traces or sensitive error details to clients.
  - Log detailed errors on the server side while sending generic messages to clients.

4. Rate Limiting

- **Prevent Abuse:**
  - Implement rate limiting to restrict the number of requests a client can make within a specific timeframe.

5. Secure Data Storage

- **Protect Sensitive Data:**
  - Encrypt sensitive data stored in files.
  - Set appropriate file permissions to restrict unauthorized access.

6. Regular Updates

- **Keep Dependencies Updated:**
  - Regularly update Node.js and all dependencies to patch known vulnerabilities.
- **Monitor Security Advisories:**
  - Stay informed about security advisories related to the tools and libraries you use.

Conclusion

By following the steps outlined above, you can create a robust and secure HTTPS server in Node.js that performs comprehensive CRUD operations on files within the `./src/data` directory. This setup is ideal for development and testing purposes. For production environments, ensure that you implement additional security measures and obtain trusted SSL certificates.

If you encounter any specific issues or have further questions, feel free to ask for more detailed assistance!

9. Node.js Events

Node.js is built around an **event-driven architecture**, which is one of its core strengths. Events play a significant role in Node.js by enabling asynchronous and non-blocking communication between different parts of an application. The `EventEmitter` class, part of the `events` module, allows you to create, emit, and listen for events in a Node.js application.

In this section, we'll explore how Node.js events work, how to create and use custom events, and practical examples of their usage.

What are Events in Node.js?

In Node.js, an event refers to a specific action or occurrence, such as a user request, a timer completion, or the availability of data. The system "emits" these events, and functions called **listeners** or **handlers** react to them.

Node.js provides the `events` module, which includes the `EventEmitter` class. Objects created from `EventEmitter` can trigger and handle various events. This allows asynchronous, event-based programming, where actions happen in response to certain triggers rather than sequential execution.

The EventEmitter Class

The `EventEmitter` class is the core of event handling in Node.js. It allows you to:

- Register event listeners
- Emit (trigger) events
- Remove event listeners

The basic pattern for working with `EventEmitter` involves creating an event emitter instance, defining listeners for specific events, and emitting those events when needed.

Example: Basic Usage of EventEmitter

```
import { EventEmitter } from 'events';
```

```
// Create an instance of EventEmitter
const eventEmitter = new EventEmitter();

// Define a listener for the 'greet' event
eventEmitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});

// Emit the 'greet' event
eventEmitter.emit('greet', 'Alice');  // Output: Hello, Alice!
```

Explanation:

- **`eventEmitter.on()`**: Registers a listener for the `greet` event.
- **`eventEmitter.emit()`**: Emits (triggers) the `greet` event, causing the registered listener to execute.

Registering Multiple Event Listeners

You can register multiple listeners for the same event. Each listener will be executed in the order in which it was registered.

Example: Multiple Event Listeners

```
import { EventEmitter } from 'events';

const eventEmitter = new EventEmitter();

// Register multiple listeners for the same event
eventEmitter.on('greet', (name) => {
  console.log(`Listener 1: Hello, ${name}!`);
});

eventEmitter.on('greet', (name) => {
  console.log(`Listener 2: How are you, ${name}?`);
});
```

```
// Emit the 'greet' event
eventEmitter.emit('greet', 'Bob');

// Output:
// Listener 1: Hello, Bob!
// Listener 2: How are you, Bob?
```

Handling Event with Arguments

When emitting an event, you can pass additional arguments to the listeners. The arguments can be used to customize the event's behavior.

Example: Passing Arguments to Event Handlers

```
import { EventEmitter } from 'events';

const eventEmitter = new EventEmitter();

// Register a listener with multiple arguments
eventEmitter.on('status', (code, message) => {
  console.log(`Status Code: ${code}, Message: ${message}`);
});

// Emit the 'status' event with arguments
eventEmitter.emit('status', 200, 'OK');  // Output: Status Code: 200,
Message: OK
```

One-Time Event Listeners

Sometimes, you may want a listener to respond to an event only once. The `once()` method allows you to register a listener that is automatically removed after it is triggered for the first time.

Example: One-Time Event Listener

```
import { EventEmitter } from 'events';
```

```
const eventEmitter = new EventEmitter();

// Register a one-time listener for the 'greet' event
eventEmitter.once('greet', (name) => {
  console.log(`Hello for the first and only time, ${name}!`);
});

// Emit the 'greet' event twice
eventEmitter.emit('greet', 'Charlie');  // Output: Hello for the first
and only time, Charlie!
eventEmitter.emit('greet', 'Charlie');  // No output, listener is
removed after the first emit
```

Removing Event Listeners

To remove a specific event listener, use the `removeListener()` or `off()` methods. These methods allow you to remove a previously registered listener.

Example: Removing Event Listeners

```
import { EventEmitter } from 'events';

const eventEmitter = new EventEmitter();

const greetListener = (name: string) => {
  console.log(`Hello, ${name}!`);
};

// Register the listener
eventEmitter.on('greet', greetListener);

// Emit the 'greet' event
eventEmitter.emit('greet', 'David');  // Output: Hello, David!

// Remove the listener
eventEmitter.off('greet', greetListener);
```

```
// Emit the 'greet' event again (no output this time)
eventEmitter.emit('greet', 'David');
```

EventEmitter Inheritance

You can inherit from the `EventEmitter` class to create custom classes that emit events. This is useful when you want to add event-driven behavior to your own objects.

Example: Custom Class Inheriting EventEmitter

```
import { EventEmitter } from 'events';

class MyEmitter extends EventEmitter {
  log(message: string) {
    console.log(message);
    this.emit('logged', { message });
  }
}

const myEmitter = new MyEmitter();

// Register a listener for the 'logged' event
myEmitter.on('logged', (data) => {
  console.log(`Event emitted with message: ${data.message}`);
});

// Call the custom method which emits the 'logged' event
myEmitter.log('This is a custom event');  // Output:
                                          // This is a custom event
                                          // Event emitted with message:
This is a custom event
```

Handling Errors in Event Emitters

When an error occurs within an event emitter, it's a common practice to emit an `'error'` event. If there are no listeners for the `'error'` event, Node.js will throw the error and crash the program. Always ensure there is a listener for error events when emitting errors.

Example: Emitting and Handling Errors

```
import { EventEmitter } from 'events';

const eventEmitter = new EventEmitter();

// Register an error event listener
eventEmitter.on('error', (err) => {
  console.error('Error occurred:', err);
});

// Emit an error event
eventEmitter.emit('error', new Error('Something went wrong!'));

// Output: Error occurred: Error: Something went wrong!
```

Built-in Node.js Event Emitters

Many Node.js core modules use the `EventEmitter` class internally to handle asynchronous operations. Some examples include:

- **`http`**: The HTTP module uses events to handle incoming requests and responses.
- **`net`**: The Networking module uses events for managing socket connections.
- **`fs`**: The File System module emits events when files are opened, read, or closed.

Example: Using Events in the http Module

```
import * as http from 'http';

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, Event-Driven World!');
});

// Register a listener for the 'connection' event
```

```
server.on('connection', () => {
  console.log('New connection established');
});

// Start the server
server.listen(3000, () => {
  console.log('Server listening on http://localhost:3000');
});
```

10. Node.js Errors

Error handling is an essential part of building reliable applications, and Node.js provides several mechanisms for dealing with errors in an efficient way. By understanding the different types of errors and how to handle them, you can write more robust applications.

Node.js uses both synchronous and asynchronous error handling mechanisms. Since Node.js is event-driven and non-blocking, errors can occur in both synchronous code (where they can be caught immediately) and asynchronous code (where error handling is more complex).

In this section, we will cover:

- Common types of errors in Node.js
- Error handling in both synchronous and asynchronous code
- The `Error` object and custom error classes
- Best practices for error handling

a. Types of Errors in Node.js

In Node.js, errors can generally be categorized into several types:

1. **Operational Errors**: Errors that occur during the normal operation of a program. These include network issues, file not found errors, permission errors, etc.
2. **Programmer Errors**: Errors caused by bugs in the code, such as accessing an undefined variable or calling a function that doesn't exist.
3. **System Errors**: Errors that occur due to issues outside of the application, such as hardware failures or network problems.

b. The Error Object

In Node.js, all errors are represented as instances of the built-in `Error` class. The `Error` object provides information about what went wrong and can be thrown and caught using the try/catch mechanism in synchronous code.

Example: Throwing and Catching Errors

```
try {
  throw new Error('Something went wrong!');
} catch (err) {
  console.error('Caught an error:', err.message);
}
```

Properties of the Error Object:

- **message**: A human-readable description of the error.
- **name**: The name of the error (defaults to 'Error').
- **stack**: A stack trace that shows where the error occurred in the code.

c. Handling Synchronous Errors

Synchronous code runs in a blocking fashion, so errors can be caught immediately using try/catch blocks.

Example: Handling Synchronous Errors

```
try {
  // Some synchronous code that could throw an error
  const result = JSON.parse('{ malformed JSON }');
} catch (err) {
  console.error('Error occurred during parsing:', err.message);
}
```

In synchronous code, try/catch blocks are straightforward to use and help to prevent the application from crashing.

d. Handling Asynchronous Errors

Handling errors in asynchronous code is more challenging because of Node.js's non-blocking nature. There are several patterns for handling errors in asynchronous operations, such as callbacks, promises, and async/await.

1. Error Handling in Callbacks

When working with callbacks, errors are typically passed as the first argument to the callback function. This is often referred to as the "error-first" callback pattern.

```
import * as fs from 'fs';

// Example: Handling errors in a callback
fs.readFile('nonexistentfile.txt', 'utf-8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err.message);
    return;
  }
  console.log('File content:', data);
});
```

2. Error Handling in Promises

When using promises, errors can be caught using .catch().

```
// Example: Handling errors in a promise
const asyncOperation = new Promise((resolve, reject) => {
  const errorOccurred = true;
  if (errorOccurred) {
    reject(new Error('Something went wrong!'));
  } else {
    resolve('Operation succeeded');
  }
});

asyncOperation
  .then((result) => {
    console.log(result);
  })
  .catch((err) => {
```

```
    console.error('Error in promise:', err.message);
  });
```

3. Error Handling with async/await

With `async/await`, error handling is similar to synchronous code. You can use `try/catch` blocks to handle errors in asynchronous code.

```
async function readFileAsync() {
  try {
    const data = await fs.promises.readFile('nonexistentfile.txt', 'utf-
8');
    console.log('File content:', data);
  } catch (err) {
    console.error('Error reading file:', err.message);
  }
}

readFileAsync();
```

e. Custom Error Classes

Node.js allows you to create custom error classes that extend the built-in `Error` class. This can be useful for defining specific error types and improving error categorization.

Example: Creating a Custom Error Class

```
class ValidationError extends Error {
  constructor(message: string) {
    super(message);
    this.name = 'ValidationError';
  }
}

try {
  throw new ValidationError('Invalid input data');
} catch (err) {
  if (err instanceof ValidationError) {
```

```
    console.error('Caught a validation error:', err.message);
  }
}
```

Explanation:

- ValidationError: This custom error class inherits from the Error class and can be used to signal validation errors.

f. Uncaught Exceptions

When an error occurs but is not caught by any error-handling code, Node.js emits an uncaughtException event. Unhandled errors can cause the application to crash. You can listen for uncaughtException events to log the error, clean up resources, and exit the application gracefully.

Example: Handling Uncaught Exceptions

```
process.on('uncaughtException', (err) => {
  console.error('Uncaught exception:', err.message);
  process.exit(1);  // Exit the process after handling the error
});

// Simulate an uncaught exception
throw new Error('This is an uncaught exception');
```

Warning:

- Catching uncaughtException should be done cautiously. It's better to handle errors locally in the code, rather than relying on this global handler, as catching an uncaught exception could leave your application in an inconsistent state.

g. Unhandled Promise Rejections

In Node.js, when a promise is rejected and no .catch() handler is provided, the runtime emits an unhandledRejection event. Failing to handle promise rejections can lead to memory leaks or unintended behavior.

Example: Handling Unhandled Promise Rejections

```
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled promise rejection:', reason);
  // Optionally exit the process
  process.exit(1);
});

// Simulate an unhandled promise rejection
new Promise((_, reject) => reject(new Error('This promise was rejected
but not handled')));
```

h. Best Practices for Error Handling in Node.js

1. **Handle Errors Explicitly**: Always handle errors explicitly in your code using `try/catch`, `.catch()`, or callbacks.

2. **Fail Fast**: When encountering critical errors, especially programmer errors (like referencing an undefined variable), fail fast by crashing the application and logging the issue.

3. **Graceful Shutdown**: When handling errors, especially in production, gracefully shut down the application after logging the error to avoid further corruption or inconsistent states.

4. **Use Custom Error Classes**: Create custom error classes for specific categories of errors to make it easier to catch and handle them appropriately.

5. **Log Errors**: Log errors in a consistent and informative format to help with debugging and post-mortem analysis.

6. **Avoid Using `process.exit()` Recklessly**: Exiting the process abruptly without proper cleanup (e.g., closing database connections) can cause issues. Always ensure a graceful shutdown when exiting due to errors.

11. Node intergration with Databases

Introduction to Node.js Integration with Databases

Node.js is a powerful, event-driven JavaScript runtime that excels in building fast and scalable network applications. One of its core strengths lies in its ability to seamlessly integrate with a variety of databases, both relational (SQL-based) and non-relational (NoSQL-based). Databases are fundamental to almost every application, as they store, retrieve, and manipulate the data that powers web services, APIs, and applications.

Integrating Node.js with databases is crucial for building robust applications that require data persistence, user authentication, analytics, and more. Whether you're developing a simple CRUD (Create, Read, Update, Delete) application or a complex enterprise-grade system, understanding how to work with databases in Node.js is essential.

Why Use Node.js with Databases?

Node.js's non-blocking, asynchronous nature makes it an excellent choice for database integration, especially for I/O-heavy tasks like querying databases, where it can handle many concurrent operations without being bogged down by waiting for responses. Here's why Node.js pairs well with databases:

1. **Asynchronous Operations**: Node.js's asynchronous design allows multiple database queries to run in parallel without blocking the main event loop, resulting in faster performance and scalability.

2. **Support for Multiple Databases**: Node.js supports a wide variety of databases, including:

   - **Relational Databases** (SQL): MySQL, PostgreSQL, Microsoft SQL Server (MSSQL), SQLite, etc.

   - **Non-relational Databases** (NoSQL): MongoDB, Redis, CouchDB, etc.

3. **Active Ecosystem**: The Node.js ecosystem has many libraries and ORM (Object Relational Mapping) tools that make it easier to interact with databases, such as

`mongoose` for MongoDB, `sequelize` for SQL databases, and `TypeORM` for various databases.

4. **JSON and JavaScript Compatibility**: With JSON as a common data format and JavaScript used across the stack, integration between Node.js and NoSQL databases (like MongoDB) is often seamless, reducing the need for complex data transformation.

Types of Databases You Can Use with Node.js

1. Relational Databases (SQL-based)

Relational databases store data in structured formats like tables with predefined schemas. SQL (Structured Query Language) is used to interact with the data.

- **MySQL**: One of the most popular open-source databases used for web applications. It is known for its reliability and scalability.
- **PostgreSQL**: A powerful, open-source object-relational database known for its extensibility, standards compliance, and strong support for complex queries.
- **Microsoft SQL Server (MSSQL)**: A robust enterprise-grade relational database management system from Microsoft, commonly used in large organizations.
- **SQLite**: A lightweight, self-contained SQL database engine often used for local storage in smaller applications or development environments.

2. Non-relational Databases (NoSQL-based)

NoSQL databases are designed to handle unstructured or semi-structured data and are known for their flexibility, scalability, and high performance.

- **MongoDB**: A document-based NoSQL database where data is stored in flexible, JSON-like documents. It is highly scalable and ideal for handling large volumes of unstructured data.
- **Redis**: An in-memory key-value store known for its blazing-fast read/write performance, often used for caching and real-time data processing.
- **CouchDB**: Another document-based NoSQL database, CouchDB uses a distributed architecture and stores data as JSON documents, making it suitable for distributed

systems.

Node.js Database Integration Approaches

There are two main approaches for integrating Node.js with databases:

1. Using Query Builders and Raw Queries

For developers who prefer greater control over SQL queries or who need to work directly with database-specific features, raw queries or query builders like **Knex.js** are great options. These allow developers to write SQL queries directly or through a builder that generates SQL dynamically.

Example: Raw Query with MySQL

```
const mysql = require('mysql2/promise');

async function fetchTodos() {
  const connection = await mysql.createConnection({ /* connection
details */ });
  const [rows] = await connection.execute('SELECT * FROM todos');
  return rows;
}
```

11.1 Node.js Database Integration Approaches

There are two main approaches for integrating Node.js with databases:

1. Using Query Builders and Raw Queries

For developers who prefer greater control over SQL queries or who need to work directly with database-specific features, raw queries or query builders like **Knex.js** are great options. These allow developers to write SQL queries directly or through a builder that generates SQL dynamically.

Example: Raw Query with MySQL

```
const mysql = require('mysql2/promise');

async function fetchTodos() {
  const connection = await mysql.createConnection({ /* connection
details */ });
  const [rows] = await connection.execute('SELECT * FROM todos');
  return rows;
}
```

2. Using ORM (Object-Relational Mapping)

ORMs abstract database interactions, allowing developers to work with objects and models instead of raw SQL queries. This approach is particularly useful for those who want to avoid writing raw SQL and focus on the business logic.

Popular Node.js ORMs:

- **Drizzle**: It's the only ORM with both relational and SQL-like query APIs, providing you the best of both worlds when it comes to accessing your relational data.
- **Sequelize**: A promise-based ORM for MySQL, PostgreSQL, and SQLite that provides a comprehensive set of features, including model definition, associations, and transactions.

- **TypeORM**: A versatile ORM that supports a wide variety of databases, including MySQL, PostgreSQL, SQLite, and MongoDB. It works well with TypeScript and supports decorators for defining models.
- **Mongoose**: A popular ODM (Object Data Modeling) library for MongoDB that allows for defining schemas, models, and relationships in a simple and efficient way.

MSSQL

mssql

MSQL Todo CRUD App

Prerequisites

Make sure you have SQL Schema for `todo_tbl`: Here's the schema that corresponds to the table used in this code:

```
CREATE TABLE todo_tbl (
  id INT IDENTITY(1,1) PRIMARY KEY,          -- IDENTITY is used for
auto-increment in MSSQL
  task NVARCHAR(255) NOT NULL,               -- NVARCHAR is used for
variable-length Unicode strings
  completed BIT DEFAULT 0,                   -- BIT is used for boolean-
like columns (0 = false, 1 = true)
  created_at DATETIME DEFAULT GETDATE()      -- DATETIME for timestamps,
GETDATE() returns current timestamp
);
```

Step 1: Install Dependencies

```
npm install mssql
npm install typescript @types/node @types/mssql --save-dev
```

** Step 2: Configure MSSQL Connection and Implement CRUD Functions**

Create a TypeScript file to handle database operations.

1. **Create `index.ts`**

```
import sql from 'mssql';

// MSSQL Connection Configuration
const config: sql.config = {
  user: 'sa', // Replace with your SQL Server username
  password: '@StrongPassword', // Replace with your SQL Server password
  server: 'localhost', // Server name or IP address
  database: 'todo_db',
  options: {
    encrypt: false, // Set to true if you're using Azure or SSL
    trustServerCertificate: true, // Change to true for local dev /
self-signed certs
  },
  pool: {
```

```
    max: 10,
    min: 0,
    idleTimeoutMillis: 30000,
  },
};

// Initialize Connection Pool
const poolPromise: Promise<sql.ConnectionPool> = new
sql.ConnectionPool(config)
  .connect()
  .then((pool: sql.ConnectionPool) => {
    console.log('Connected to MSSQL');
    return pool;
  })
  .catch((err: Error) => {
    console.error('Database Connection Failed! Bad Config: ', err);
    throw err;
  });

// Pool and Request Initialization (Extracted)
let pool: sql.ConnectionPool;
let request: sql.Request;

async function initDBConnection(): Promise<void> {
  pool = await poolPromise;
  request = pool.request(); // Reuse the same request object for all
operations
}

// Create a new Todo in `todo_tbl`
async function createTodo(task: string, completed: boolean):
Promise<void> {
  try {
    request.input('task', sql.NVarChar(255), task);
    request.input('completed', sql.Bit, completed);
    const result: sql.IResult<{ id: number }> = await request.query(
      'INSERT INTO todo_tbl (task, completed, created_at) VALUES (@task,
@completed, GETDATE()); SELECT SCOPE_IDENTITY() as id;'
```

```
    );
    console.log('Todo Created with ID:', result.recordset[0].id);
  } catch (err: any) {
    console.error('Error creating todo:', err);
  }
}

// Fetch all Todos from `todo_tbl`
async function fetchAllTodos(): Promise<void> {
  try {
    const result: sql.IResult<any> = await request.query('SELECT * FROM
todo_tbl');
    console.log('All Todos:', result.recordset);
  } catch (err: any) {
    console.error('Error fetching todos:', err);
  }
}

// Fetch Todo by ID from `todo_tbl`
async function fetchTodoById(id: number): Promise<void> {
  try {
    request.input('id', sql.Int, id);
    const result: sql.IResult<any> = await request.query('SELECT * FROM
todo_tbl WHERE id = @id');

    if (result.recordset.length > 0) {
      console.log('Todo:', result.recordset[0]);
    } else {
      console.log(`Todo with ID ${id} not found.`);
    }
  } catch (err: any) {
    console.error('Error fetching todo by ID:', err);
  }
}

// Delete Todo by ID from `todo_tbl`
async function deleteTodoById(id: number): Promise<void> {
  try {
```

```
    request.input('id', sql.Int, id);
    const result: sql.IResult<any> = await request.query('DELETE FROM
todo_tbl WHERE id = @id');

    if (result.rowsAffected[0] > 0) {
      console.log(`Todo deleted with ID: ${id}`);
    } else {
      console.log(`No Todo found with ID: ${id}`);
    }
  } catch (err: any) {
    console.error('Error deleting todo:', err);
  }
}

// Execute CRUD operations
(async () => {
  try {
    await initDBConnection(); // Initialize the pool and request
    await createTodo('Learn MSSQL', false);
    await fetchAllTodos();
    await fetchTodoById(1);
    await deleteTodoById(1);
    await fetchAllTodos(); // To verify deletion
  } catch (err) {
    console.error('Error during CRUD operations:', err);
  }
})();
```

Step 3: Run the App

```
npm run dev
```

PostgreSQL

postgres

MySQL Todo CRUD App

Prerequisites

Make sure you have SQL Schema for `todo_tbl`: Here's the schema that corresponds to the table used in this code:

```
CREATE TABLE todo_tbl (
  id SERIAL PRIMARY KEY,                -- SERIAL auto-increments the ID
  task VARCHAR(255) NOT NULL,           -- Task is a string with a max
length of 255 characters
  completed BOOLEAN DEFAULT FALSE,      -- Boolean, default is FALSE
  created_at TIMESTAMPTZ DEFAULT NOW()  -- Timestamp with timezone,
defaulting to the current timestamp
);
```

Step 1: Install Dependencies

```
npm install pg
npm install typescript @types/node @types/pg --save-dev
```

download the PostgresSQL Server (https://www.postgresql.org/download/)

Step 2: PostgreSQL Connection and CRUD Functions

Create a file `index.ts`.

```
import { Pool } from 'pg';

// PostgreSQL Connection
const pool = new Pool({
  user: 'admin',
  host: 'localhost',
  database: 'todo_db',
  password: '@StrongPassword',
  port: 5432,
});

// Create a new Todo item in the `todo_tbl` table
async function createTodo(task: string, completed: boolean) {
  const result = await pool.query(
    'INSERT INTO todo_tbl (task, completed, created_at) VALUES ($1, $2,
NOW()) RETURNING *',
    [task, completed]
  );
```

```
  console.log('Todo Created:', result.rows[0]);
}

// Fetch all Todos from the `todo_tbl` table
async function fetchAllTodos() {
  const result = await pool.query('SELECT * FROM todo_tbl');
  console.log('All Todos:', result.rows);
}

// Fetch a Todo by ID from the `todo_tbl` table
async function fetchTodoById(id: number) {
  const result = await pool.query('SELECT * FROM todo_tbl WHERE id =
$1', [id]);
  console.log('Todo by ID:', result.rows[0]);
}

// Delete a Todo by ID from the `todo_tbl` table
async function deleteTodoById(id: number) {
  await pool.query('DELETE FROM todo_tbl WHERE id = $1', [id]);
  console.log('Todo deleted with ID:', id);
}

// Execute CRUD operations
(async () => {
  await createTodo('Learn PostgreSQL', false);
  await fetchAllTodos();
  await fetchTodoById(1);
  // await deleteTodoById(1);
})();
```

Step 3: Run the App

```
npm run dev
```

MySQL

mysql

MySQL Todo CRUD App

Prerequisites

Make sure you have SQL Schema for todo_tbl: Here's the schema that corresponds to the table used in this code:

```
CREATE TABLE todo_tbl (
    id INT IDENTITY(1,1) PRIMARY KEY,
    task NVARCHAR(255) NOT NULL,
    completed BIT NOT NULL,
    created_at DATETIME DEFAULT GETDATE()
);
```

Step 1: Install Dependencies

```
npm install mysql2 typescript @types/node
```

download MySQL Server (https://dev.mysql.com/downloads/installer/)

Step 2: MySQL Connection and CRUD Functions

Create a file `mysql-todo.ts`.

```
import mysql, { PoolOptions, RowDataPacket, ResultSetHeader } from
'mysql2/promise';

// Define connection pool options
const access: PoolOptions = {
  host: 'localhost',
  user: 'admin',
  password: '@StrongPassword',
  database: 'todo_db',
};

// Create a connection pool
const pool = mysql.createPool(access);

// CRUD Operations using `todo_tbl`

// Create a Todo item in `todo_tbl`
async function createTodo(task: string, completed: boolean) {
  const [result] = await pool.execute<ResultSetHeader>(
    'INSERT INTO todo_tbl (task, completed, created_at) VALUES (?, ?,
NOW())',
    [task, completed]
  );
  console.log('Todo Created:', result);
}

// Fetch all Todos from `todo_tbl`
async function fetchAllTodos() {
  const [todos] = await pool.query<RowDataPacket[]>(
    'SELECT * FROM todo_tbl'
  );
  console.log('All Todos:', todos);
}
```

```
// Fetch Todo by ID from `todo_tbl`
async function fetchTodoById(id: number) {
  const [todo] = await pool.query<RowDataPacket[]>(
    'SELECT * FROM todo_tbl WHERE id = ?',
    [id]
  );
  console.log('Todo by ID:', todo[0]);
}

// Delete Todo by ID from `todo_tbl`
async function deleteTodoById(id: number) {
  const [result] = await pool.execute<ResultSetHeader>(
    'DELETE FROM todo_tbl WHERE id = ?',
    [id]
  );
  console.log('Todo deleted with ID:', id);
}

// Execute CRUD operations
(async () => {
  try {
    await createTodo('Learn TypeScript', false);
    await createTodo('Learn JavaScript', false);
    await createTodo('Learn C#', false);
    await fetchAllTodos();
    // Uncomment the following lines if you want to fetch or delete by
ID
    // await fetchTodoById(1);
    // await deleteTodoById(1);
  } catch (error) {
    console.error('Error:', error);
  }
})();
```

Step 3: Run the App

```
npm run dev
```

MongoDB

mongodb

MongoDB Todo CRUD App

Prerequisites

1. **Install MongoDB**:

- Download and install MongoDB (https://www.mongodb.com/try/download/community) and ensure it's running on your local machine.

- Alternatively, you can use a cloud MongoDB service like MongoDB Atlas (https://www.mongodb.com/atlas).

2. **Install Dependencies**:

- You'll need `mongodb` and TypeScript-related dependencies.

```
npm install mongodb typescript @types/node
```

MongoDB Schema

Unlike MySQL, MongoDB doesn't require a predefined schema. A document in the `todo_tbl` collection might look like this:

```
{
  "_id": ObjectId("6139db485f13b720f07b7a71"),
  "task": "Learn TypeScript",
  "completed": false,
  "created_at": ISODate("2021-09-09T10:20:00Z")
}
```

MongoDB will automatically create the collection and documents as needed.

Step 1: Create MongoDB CRUD App

Create a file named `mongodb-todo.ts`.

```
import { MongoClient, ObjectId } from 'mongodb';

// Encode special characters in the password
const encodedPassword = encodeURIComponent('@StrongPassword');

//'mongodb://admin:@StrongPassword@localhost:27017/todo_db?
authSource=admin'

// MongoDB connection string
const uri = `mongodb://admin:${encodedPassword}@localhost:27017/todo_db?
authSource=admin`;
```

```
// MongoDB database and collection
const dbName = 'todo_db';
const collectionName = 'todo_collection';  // Changed collection name to
`todo_collection`

// Initialize MongoDB client with proper options
const client = new MongoClient(uri);

async function connectToMongoDB() {
  try {
    await client.connect();
    console.log('Connected to MongoDB');
  } catch (error:any) {
    console.error('Error connecting to MongoDB:', error.message);
    throw error;
  }
}

// Create a new Todo item in `todo_collection`
async function createTodo(task: string, completed: boolean) {
  try {
    const db = client.db(dbName);
    const collection = db.collection(collectionName);
    const result = await collection.insertOne({
      task,
      completed,
      created_at: new Date(),
    });
    console.log('Todo Created:', result.insertedId);
  } catch (error:any) {
    console.error('Error creating todo:', error.message);
  }
}

// Fetch all Todos from `todo_collection`
async function fetchAllTodos() {
  try {
```

```
    const db = client.db(dbName);
    const collection = db.collection(collectionName);
    const todos = await collection.find({}).toArray();
    console.log('All Todos:', todos);
  } catch (error:any) {
    console.error('Error fetching todos:', error.message);
  }
}

// Fetch a Todo by ID from `todo_collection`
async function fetchTodoById(id: string) {
  try {
    const db = client.db(dbName);
    const collection = db.collection(collectionName);
    const todo = await collection.findOne({ _id: new ObjectId(id) });
    if (todo) {
      console.log('Todo by ID:', todo);
    } else {
      console.log(`Todo with ID ${id} not found.`);
    }
  } catch (error:any) {
    console.error('Error fetching todo by ID:', error.message);
  }
}

// Delete a Todo by ID from `todo_collection`
async function deleteTodoById(id: string) {
  try {
    const db = client.db(dbName);
    const collection = db.collection(collectionName);
    const result = await collection.deleteOne({ _id: new ObjectId(id)
});
    if (result.deletedCount > 0) {
      console.log(`Todo deleted with ID: ${id}`);
    } else {
      console.log(`No Todo found with ID: ${id}`);
    }
  } catch (error:any) {
```

```
    console.error('Error deleting todo:', error.message);
  }
}

// Run MongoDB CRUD Operations
(async () => {
  try {
    await connectToMongoDB();

    // Example operations (uncomment to test)

    // await createTodo('Learn MongoDB', false);
    // await createTodo('Learn JavaScript', false);
    // await createTodo('Learn TypeScript', false);

    await fetchAllTodos();

    // Example ID (replace with an actual ID from your database)
    // const exampleId = '6139db485f13b720f07b7a71'; // Replace with a
valid ObjectId
    // await fetchTodoById(exampleId);

    // Uncomment to delete by ID
    // await deleteTodoById(exampleId);

    // await fetchAllTodos();
  } catch (error:any) {
    console.error('Error:', error.message);
  } finally {
    await client.close(); // Ensure MongoDB client is closed when done
  }
})();
```

Step 2: Run the App

```
npx ts-node mongodb-todo.ts
```

12. Worker Threads in Node.JS (Multithreading)

Overview

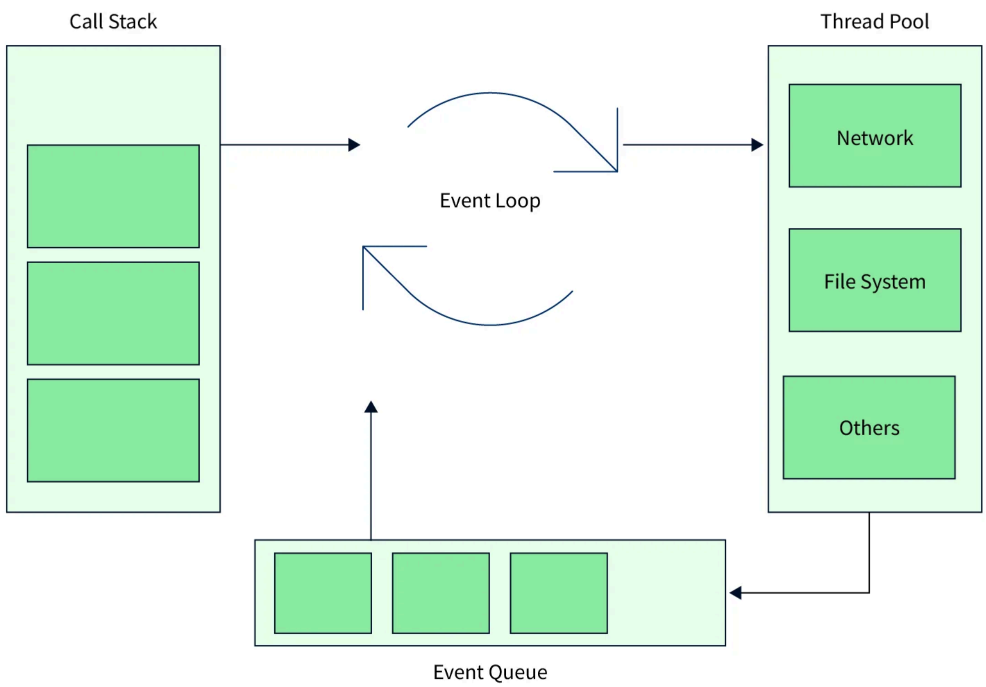
Node.js runs the javascript code using only a single thread, which means only one statement can be executed at a time. However, Nodejs itself is multithreaded and supplies hidden threads in js using the **libuv library**, which handles operations like network requests or reading a file from a disk. Nodejs introduced the **worker threads module**, which allows us to create `multiple tasks in parallel` and execute those tasks. These worker threads divide CPU-bound tasks to many workers for optimization and do not block the main thread.

Introduction

- Threads are like a small process: they have their instruction pointer and can conduct one Javascript task at a time. Threads reside within a process's memory and they don't have their memory. The execution of threads is similar to that of processes.
- Javascript is single-threaded by default which means only one thread is available to perform all the operations. This was the main drawback of the javascript which led to the implementation of server-side Asynchronous I/O, which would reduce the race among the thread in a multithreading environment.
- Lets us now see how javascript has evolved. Javascript has introduced a new concept to prevent the limitation we are facing, that is, Web Workers. Web Workers (threads in js) are mainly used to perform CPU-intensive Javascript tasks. They can perform long-running tasks without affecting or blocking the main execution thread.
- Using a Constructor, we can create a worker object, which will run a js file. This js file includes code to run the worker thread; these workers run in some other global context that is different from the current window. However, Node.js built-in asynchronous I/O operations are more efficient than Workers can be. Web workers are not of much help with I/O-intensive work.

Need for Parallel Processing

Imagine if we do CPU-intensive tasks synchronously, such as a loop with a large number of iterations, fetching data from an API or external resource, and handling file system I/O. This will create a large block of synchronous code that will block the rest of the code. Meanwhile, if another request is sent, it will be waiting for a much larger period before the response is sent back to the user.



event_loop.png

Hence, there is a need for parallel computation where the incoming requests don't have to wait. But due to the single-threaded nature of Node.js, the main loop can take up only one task at a time. Hence worker threads are required to provide the parallel computation. The event loop takes on the synchronous part of all the processes, and the time taking asynchronous part is assigned to a worker thread provided by the **libuv.** The threads then parallel execute the asynchronous task in the background and respond to the event loop once the process is completed.

Worker Threads in Node JS

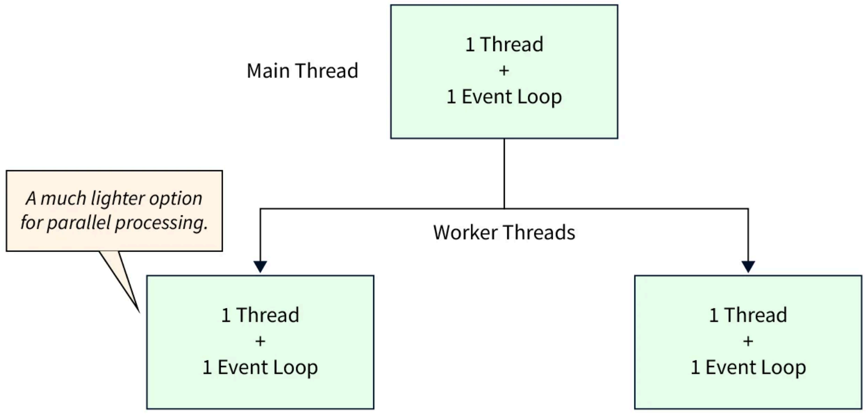
When a Node.js process is executed, it runs:

- **One Process:** It is the Node.js global object that can be accessed from anywhere and has details about what's being executed at a time.
- **One Thread:** Javascript being single-threaded, has only one main thread for the execution of tasks at a given time.
- **One Event Loop:** This plays an important role in the asynchronous nature and non-blocking I/O of Node.js, by offloading tasks to the system kernel whenever possible through promises, callbacks and async/ await.
- **One JS Engine Instance:** This is a computer program that executes js code.
- **One Node.js Instance:** This is a computer program that executes Node.js code.

In simple words, Node runs on **one main thread**, and there can only be a single task executing at a time in the event loop. However, there is a downside: if we have CPU-intensive code, like reading a file from the disk or any complex calculations in a large dataset, then this can block the main thread and hence another process from being executed.

A function is known as **blocking** if the main event loop must wait until it has finished executing the following command. A ***Non-blocking function*** will allow the main event loop to continue executing further code and alerts the main loop once it has finished by calling a *callback*.

It's important to differentiate between *CPU operations* and *I/O (input/output) operations*, as only I/O operations are run *concurrently because they are executed asynchronously*. So the main *goal of Worker threads in js is to enhance the performance of CPU-intensive operations, **and not I/O operations***.



threads.png

Hence worker threads in js have the following properties:

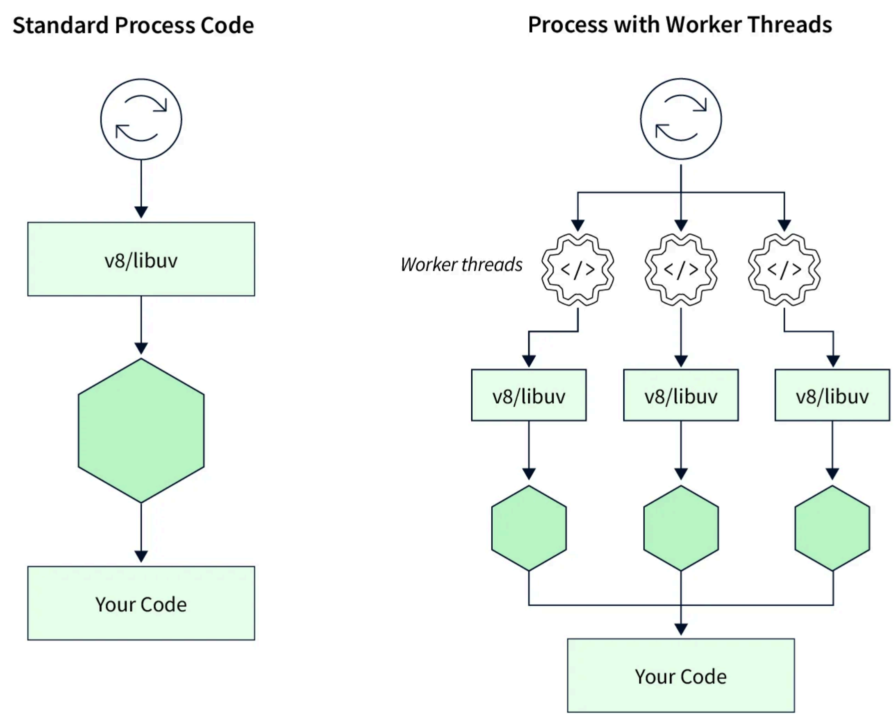
1. One process
2. Multiple threads
3. One event loop per thread
4. One JS Engine Instance per thread
5. One Node.js Instance per thread

How do Worker Threads Work?

Worker threads in js were submitted as an experimental feature in Node.js version v10, but they became stable in version v12 of Node.js. Worker threads provide us with APIs to deal with CPU-intensive tasks without blocking the main thread; hence they maintain the responsive application. Unlike clusters or child_process, worker threads share memory among themselves by transferring ArrayBuffer instances.

The worker thread gives CPU-intensive tasks to another thread while keeping the main thread available to new user requests. When we create a new worker thread, it is associated with a new event loop, so the main thread's event loop can take any further requests.

Worker threads in js don't have any synchronization mechanism like any multi-threaded programming language. Chrome's V8 engine is used to run the Node process. Each worker thread is isolated from other threads. The V8 engine allows us to create isolated V8 runtimes.V8 isolates the instances with their own Javascript heaps and micro-task queues. V8 isolates exist in fully distinct states. Objects from one isolation may not be used in another. The embedder can build several isolates and use them in different threads at the same time. At any given time, only one thread can enter an isolate.



workerthreads.png

12.1 Implementing worker threads

To utilize worker threads in Node.js using TypeScript and TSX, follow the steps below.

Project Setup

1. **Initialize the Project:**

   Create a new directory for your project and initialize it with `npm`.

   ```
   mkdir worker-threads-ts
   cd worker-threads-ts
   npm init -y
   ```

2. **Install Dependencies:**

   Install the necessary dependencies, including TypeScript and TSX.

   ```
   npm install typescript tsx --save-dev
   ```

3. **Initialize TypeScript Configuration:**

   Generate a `tsconfig.json` file with the following command:

   ```
   npx tsc --init
   ```

   Update the `tsconfig.json` to support ES Modules and other necessary settings:

   ```
   {
     "compilerOptions": {
       "target": "ES2020",
       "module": "ESNext",
       "moduleResolution": "node",
       "outDir": "./dist",
       "rootDir": "./src",
       "strict": true,
       "esModuleInterop": true,
       "forceConsistentCasingInFileNames": true,
   ```

```
    "skipLibCheck": true
  },
  "include": ["src"]
}
```

4. **Update `package.json`:**

   Add a script to run your TypeScript files using TSX.

   ```
   {
     "scripts": {
       "start": "tsx src/main.ts"
     }
   }
   ```

5. **Project Structure:**

   Create a `src` directory to house your TypeScript files.

   ```
   mkdir src
   ```

Basic Worker Thread Example

Below is a simple example demonstrating how to create a worker thread that executes a separate TypeScript file.

1. Create the Worker Script (`worker.ts`):

```
// src/worker.ts
import { parentPort } from 'worker_threads';

// Define the type of tasks the worker can handle
type Task = number;

// Function to calculate factorial
function factorial(n: number): number {
  if (n === 0 || n === 1) return 1;
  return n * factorial(n - 1);
```

```
}

// Listen for messages from the main thread
parentPort?.on('message', (task: Task) => {
  console.log(`Worker received task: ${task}`);

  const result = factorial(task);

  // Send the result back to the main thread
  parentPort?.postMessage(result);
});
```

2. Create the Main Script (`main.ts`):

```
// src/main.ts
import { Worker } from 'worker_threads';
import path from 'path';
import { fileURLToPath } from 'url';

// Helper to get __dirname in ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Define the type for worker results
type WorkerResult = number;

// Function to run a worker
function runWorker(task: number): Promise<WorkerResult> {
  return new Promise((resolve, reject) => {
    const workerPath = path.resolve(__dirname, './worker.ts');
    const worker = new Worker(workerPath, {
      // Specify the TypeScript file; TSX handles transpilation
      // No additional execArgv needed for TSX
    });

    // Send task to the worker
```

```
    worker.postMessage(task);

    // Listen for messages from the worker
    worker.on('message', (result: WorkerResult) => {
      resolve(result);
    });

    // Handle errors
    worker.on('error', reject);

    // Handle worker exit
    worker.on('exit', (code) => {
      if (code !== 0)
        reject(new Error(`Worker stopped with exit code ${code}`));
    });
  });
}

// Example usage
async function main() {
  const number = 10;
  console.log(`Calculating factorial of ${number} in worker thread...`);

  try {
    const result = await runWorker(number);
    console.log(`Factorial of ${number} is ${result}`);
  } catch (err) {
    console.error(err);
  }
}

main();
```

3. Run the Application:

Execute the following command to run the main script using the `start` script defined in `package.json`:

```
npm run start
```

Output:

```
Calculating factorial of 10 in worker thread...
Worker received task: 10
Factorial of 10 is 3628800
```

Explanation:

- **worker.ts:**
  - Uses TypeScript's type annotations for better type safety.
  - Listens for messages (`Task` type) from the main thread.
  - Calculates the factorial of the received number and sends the result back.
- **main.ts:**
  - Creates a new worker thread by instantiating the `Worker` class with the path to `worker.ts`.
  - Sends a number to the worker thread to compute its factorial.
  - Awaits the result and logs it to the console.
  - Includes error handling for worker errors and exit codes.

Communication Between Main Thread and Worker

Worker threads communicate with the main thread using messages. They can send and receive messages using the `postMessage` method and the `message` event.

Example: Sending and Receiving Multiple Messages

1. Update `worker.ts` to Handle Multiple Tasks:

```
// src/worker.ts
import { parentPort } from 'worker_threads';
```

```
// Define the types for tasks and messages
type Task = number | 'exit';
type Message = number;

// Function to calculate factorial
function factorial(n: number): number {
  if (n === 0 || n === 1) return 1;
  return n * factorial(n - 1);
}

// Listen for messages from the main thread
parentPort?.on('message', (task: Task) => {
  if (task === 'exit') {
    parentPort?.close();
  } else {
    console.log(`Worker received task: ${task}`);
    const result = factorial(task);
    parentPort?.postMessage(result);
  }
});
```

2. Update `main.ts` to Send Multiple Tasks:

```
// src/main.ts
import { Worker } from 'worker_threads';
import path from 'path';
import { fileURLToPath } from 'url';

// Helper to get __dirname in ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Define the type for worker results
type WorkerResult = number;
```

```
// Function to run a worker
function runWorker(task: number | 'exit'): Promise<WorkerResult> {
  return new Promise((resolve, reject) => {
    const workerPath = path.resolve(__dirname, './worker.ts');
    const worker = new Worker(workerPath, {
      // Specify the TypeScript file; TSX handles transpilation
    });

    // Send task to the worker
    worker.postMessage(task);

    // Listen for messages from the worker
    worker.on('message', (result: WorkerResult) => {
      resolve(result);
    });

    // Handle errors
    worker.on('error', reject);

    // Handle worker exit
    worker.on('exit', (code) => {
      if (code !== 0)
        reject(new Error(`Worker stopped with exit code ${code}`));
    });
  });
}

// Example usage
async function main() {
  const tasks: number[] = [5, 7, 10];
  const promises = tasks.map(runWorker);

  try {
    const results = await Promise.all(promises);
    tasks.forEach((task, index) => {
      console.log(`Factorial of ${task} is ${results[index]}`);
    });
  } catch (err) {
```

```
    console.error(err);
  }
}

main();
```

3. Run the Application:

```
npm run start
```

Output:

```
Worker received task: 5
Worker received task: 7
Worker received task: 10
Factorial of 5 is 120
Factorial of 7 is 5040
Factorial of 10 is 3628800
```

Explanation:

- **worker.ts:**
  - Enhanced to handle multiple tasks by accepting an array of numbers.
  - Listens for the `'exit'` message to gracefully close the worker.
- **main.ts:**
  - Sends multiple tasks to separate worker threads by mapping over the `tasks` array.
  - Awaits all promises using `Promise.all` and logs each result accordingly.

Practical Example: CPU-Intensive Task

To illustrate the benefits of worker threads, let's compare a CPU-intensive task executed in the main thread versus using worker threads.

Task: Calculate the Fibonacci number for a large `n` (e.g., `n = 40`).

1. Without Worker Threads (Single-Threaded)

fibonacci.ts:

```
// src/fibonacci.ts
function fibonacci(n: number): number {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

const n = 40;
console.log(`Fibonacci of ${n} is ${fibonacci(n)}`);
```

Run the Script:

Execute the script using TSX.

```
tsx src/fibonacci.ts
```

Output:

```
Fibonacci of 40 is 102334155
```

Note: While this computation is running, the main thread is blocked and cannot handle other tasks.

2. Using Worker Threads (Multithreaded)

worker-fibonacci.ts:

```
// src/worker-fibonacci.ts
import { parentPort } from 'worker_threads';

// Function to calculate Fibonacci
function fibonacci(n: number): number {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
// Listen for messages from the main thread
parentPort?.on('message', (n: number) => {
  const result = fibonacci(n);
  parentPort?.postMessage(result);
});
```

main-fibonacci.ts:

```
// src/main-fibonacci.ts
import { Worker } from 'worker_threads';
import path from 'path';
import { fileURLToPath } from 'url';

// Helper to get __dirname in ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Define the type for worker results
type WorkerResult = number;

// Function to run the Fibonacci worker
function runFibonacci(n: number): Promise<WorkerResult> {
  return new Promise((resolve, reject) => {
    const workerPath = path.resolve(__dirname, './worker-fibonacci.ts');
    const worker = new Worker(workerPath, {
      // Specify the TypeScript file; TSX handles transpilation
    });

    // Send task to the worker
    worker.postMessage(n);

    // Listen for messages from the worker
    worker.on('message', (result: WorkerResult) => {
      resolve(result);
    });
```

```
    // Handle errors
    worker.on('error', reject);

    // Handle worker exit
    worker.on('exit', (code) => {
      if (code !== 0)
        reject(new Error(`Worker stopped with exit code ${code}`));
    });
  });
}

// Example usage
async function main() {
  const n = 40;
  console.log(`Calculating Fibonacci of ${n} using worker thread...`);

  const fibPromise = runFibonacci(n);

  // Meanwhile, main thread can perform other tasks
  console.log('Main thread is free to perform other tasks.');

  try {
    const result = await fibPromise;
    console.log(`Fibonacci of ${n} is ${result}`);
  } catch (err) {
    console.error(err);
  }
}

main();
```

3. Run the Script:

```
tsx src/main-fibonacci.ts
```

Output:

```
Calculating Fibonacci of 40 using worker thread...
Main thread is free to perform other tasks.
Fibonacci of 40 is 102334155
```

Explanation:

- **Single-Threaded Approach:**
  - The main thread is blocked while calculating the Fibonacci number, making the application unresponsive during the computation.
- **Worker Thread Approach:**
  - The computation is offloaded to a worker thread.
  - The main thread remains free to handle other tasks, enhancing responsiveness and efficiency.

Conclusion

Worker threads in Node.js provide a powerful way to handle CPU-intensive tasks without blocking the main event loop. By leveraging multithreading with TypeScript and TSX, applications can achieve better performance, efficient resource utilization, and improved scalability. Understanding and implementing worker threads is essential for building high-performance Node.js applications that can handle a wide range of workloads effectively.