**Node.**

Node.js is an open-source, cross-platform runtime environment that allows you to run JavaScript code on the server side. It uses the V8 JavaScript engine and is designed for building scalable, high-performance, event-driven, and non-blocking applications, such as web servers and APIs.

**Npm init.**

npm init is a command used in Node.js to create a new **package.json** file, which serves as a metadata file for your project. This file contains important details like the project name, version, description, main file, dependencies, and scripts.

**Usage:**

* Run npm init in your terminal.
* It will prompt you to input various details about your project.
* You can skip prompts with defaults using npm init -y.

The package.json file is crucial for managing dependencies and scripts in a Node.js project.

**Package.json.**

package.json is a configuration file in a Node.js project that contains metadata and information about the project. It helps manage the project’s dependencies, scripts, and other settings.

**Key Sections of package.json:**

1. **Basic Metadata**:
   * "name": The project name.
   * "version": The project version.
   * "description": A brief description of the project.
   * "main": The entry point file (e.g., index.js).
2. **Scripts**:
   * "scripts": Custom terminal commands for running tasks.
   * Example:

"scripts": {

"start": "node index.js",

"test": "jest"

}

1. **Dependencies**:
   * "dependencies": Lists packages required for the project to run.
   * "devDependencies": Lists packages needed only for development.
   * Example:

"dependencies": {

"express": "^4.18.2"

},

"devDependencies": {

"nodemon": "^2.0.22"

}

1. **Keywords**:
   * "keywords": An array of keywords related to the project.
2. **Author**:
   * "author": The creator’s name.
3. **License**:
   * "license": The project’s license (e.g., MIT).

**Package-lock.json.**

package-lock.json is a file automatically generated by npm when you run commands like npm install. It ensures consistent dependency versions are installed across different environments, even if the versions in package.json use loose ranges (e.g., ^1.0.0 or ~1.0.0).

**Key Features:**

1. **Exact Versions**:
   * It locks the exact versions of all installed packages, including nested dependencies.
2. **Faster Installs**:
   * Speeds up future installations by avoiding version resolution steps.
3. **Ensures Consistency**:
   * Ensures the same dependency tree for all collaborators and deployment environments.
4. **Immutable**:
   * Should not be manually edited; it’s maintained automatically by npm.

**Structure:**

* Lists every installed package, their resolved versions, and their dependencies.
* Includes metadata like integrity hashes for package verification.

**Example:**

{

"name": "my-node-app",

"lockfileVersion": 2,

"requires": true,

"packages": {

"": {

"name": "my-node-app",

"version": "1.0.0",

"dependencies": {

"express": "^4.18.2"

}

},

"node\_modules/express": {

"version": "4.18.2",

"resolved": "https://registry.npmjs.org/express/-/express-4.18.2.tgz",

"integrity": "sha512-randomhash",

"dependencies": {

"body-parser": "^1.19.0"

}

}

}

}

**Do You Need to Commit It to Git?**

Yes, it’s recommended to commit package-lock.json to your version control system (e.g., Git) to ensure consistent builds.

**Fs module.**

The fs module in Node.js is a built-in module that provides functionality to interact with the file system, allowing you to read, write, update, delete, and manage files and directories.

**How to Import:**

const fs = require('fs');

**Common Methods:**

1. **Reading Files**:
   * fs.readFileSync(path, encoding): Synchronous file read.
   * fs.readFile(path, encoding, callback): Asynchronous file read.

const data = fs.readFileSync('example.txt', 'utf8');

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

1. **Writing Files**:
   * fs.writeFileSync(path, data, options): Synchronous file write.
   * fs.writeFile(path, data, options, callback): Asynchronous file write.

javascript

Copy code

fs.writeFileSync('example.txt', 'Hello, World!');

fs.writeFile('example.txt', 'Hello, Async!', (err) => {

if (err) throw err;

});

1. **Appending Files**:
   * fs.appendFileSync(path, data): Synchronous append.
   * fs.appendFile(path, data, callback): Asynchronous append.

javascript

Copy code

fs.appendFile('example.txt', '\nAppended Text', (err) => {

if (err) throw err;

});

1. **Deleting Files**:
   * fs.unlinkSync(path): Synchronous delete.
   * fs.unlink(path, callback): Asynchronous delete.

javascript

Copy code

fs.unlink('example.txt', (err) => {

if (err) throw err;

console.log('File deleted');

});

1. **Working with Directories**:
   * fs.mkdirSync(path): Create a directory synchronously.
   * fs.mkdir(path, callback): Create a directory asynchronously.
   * fs.readdirSync(path): Synchronous read directory contents.
   * fs.rmdirSync(path): Synchronous remove directory.

javascript

Copy code

fs.mkdir('newFolder', (err) => {

if (err) throw err;

});

**Modes:**

* **Synchronous**: Blocking operations that wait until the task is completed.
* **Asynchronous**: Non-blocking operations that use callbacks or promises.

**Advanced Usage:**

* Use fs.promises for Promise-based methods:

javascript

Copy code

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

async function readFile() {

const data = await fs.readFile('example.txt', 'utf8');

console.log(data);

}

readFile();

The fs module is essential for file and directory operations in Node.js applications!

**Os module.**

The os module in Node.js is a built-in module that provides operating system-related utility methods and properties. It allows you to interact with the underlying operating system and retrieve information about it.

**How to Import:**

javascript

Copy code

const os = require('os');

**Common Methods and Properties:**

1. **System Information**:
   * os.type(): Returns the operating system name (e.g., 'Linux', 'Windows\_NT').
   * os.platform(): Returns the platform (e.g., 'win32', 'darwin', 'linux').
   * os.arch(): Returns the CPU architecture (e.g., 'x64', 'arm').

javascript

Copy code

console.log(os.type()); // 'Linux'

console.log(os.platform()); // 'linux'

console.log(os.arch()); // 'x64'

1. **CPU Information**:
   * os.cpus(): Returns an array of objects containing information about each CPU core.
   * os.loadavg(): Returns an array of load averages (only for Unix-like systems).

javascript

Copy code

console.log(os.cpus()); // [{ model: 'Intel', speed: 2600, times: {...} }]

1. **Memory Information**:
   * os.totalmem(): Returns the total system memory in bytes.
   * os.freemem(): Returns the free system memory in bytes.

javascript

Copy code

console.log(os.totalmem()); // Total memory in bytes

console.log(os.freemem()); // Free memory in bytes

1. **Network Information**:
   * os.networkInterfaces(): Returns an object with network interfaces and their addresses.

javascript

Copy code

console.log(os.networkInterfaces());

1. **User Information**:
   * os.userInfo(): Returns information about the current user.

javascript

Copy code

console.log(os.userInfo());

1. **System Uptime**:
   * os.uptime(): Returns the system uptime in seconds.

javascript

Copy code

console.log(os.uptime()); // Uptime in seconds

1. **Path and Directory Information**:
   * os.homedir(): Returns the current user's home directory.
   * os.tmpdir(): Returns the default directory for temporary files.

javascript

Copy code

console.log(os.homedir()); // User's home directory

console.log(os.tmpdir()); // Temporary directory

1. **Endianness**:
   * os.endianness(): Returns the byte order of the CPU ('BE' for Big-Endian, 'LE' for Little-Endian).

javascript

Copy code

console.log(os.endianness()); // 'LE'

**Example Usage:**

javascript

Copy code

const os = require('os');

console.log('OS Type:', os.type());

console.log('Platform:', os.platform());

console.log('CPU Cores:', os.cpus().length);

console.log('Free Memory:', os.freemem() / 1024 / 1024, 'MB');

console.log('Total Memory:', os.totalmem() / 1024 / 1024, 'MB');

console.log('Home Directory:', os.homedir());

The os module is useful for gathering system-level information and building platform-specific functionality.

**Import / export**

**import export**

In JavaScript, **import** and **export** are used to share and reuse code between files or modules. They are part of the ES6 module system.

### ****Exporting:****

Allows you to make variables, functions, or classes available to other files.

1. **Named Export**:
   * Allows you to export multiple values.
   * Each must be imported with the same name.

javascript

Copy code

// file: math.js

export const add = (a, b) => a + b;

export const subtract = (a, b) => a - b;

1. **Default Export**:
   * Allows you to export one value as the default export.
   * Can be imported with any name.

javascript

Copy code

// file: math.js

const multiply = (a, b) => a \* b;

export default multiply;

### ****Importing:****

Allows you to bring exported code into a file.

1. **Import Named Exports**:
   * Use curly braces {} to specify the names.

javascript

Copy code

import { add, subtract } from './math.js';

console.log(add(5, 3)); // 8

console.log(subtract(5, 3)); // 2

1. **Import Default Export**:
   * No curly braces are needed; use any name.

javascript

Copy code

import multiply from './math.js';

console.log(multiply(5, 3)); // 15

1. **Import All as an Object**:
   * Use \* to import everything as an object.

javascript

Copy code

import \* as math from './math.js';

console.log(math.add(5, 3)); // 8

console.log(math.subtract(5, 3)); // 2

### ****CommonJS (Node.js):****

Before ES6, Node.js used CommonJS modules with require() and module.exports.

1. **Export**:

javascript

Copy code

// file: math.js

module.exports = {

add: (a, b) => a + b,

subtract: (a, b) => a - b,

};

1. **Import**:

javascript

Copy code

const math = require('./math.js');

console.log(math.add(5, 3)); // 8

console.log(math.subtract(5, 3)); // 2

### Comparison:

| **Feature** | **ES6 Modules** | **CommonJS** |
| --- | --- | --- |
| Syntax | import/export | require/module.exports |
| Static/Dynamic | Static imports | Dynamic require() |
| Browser Support | Native in modern browsers | Needs bundler like Webpack for browsers |

**ES6 modules** are now standard and recommended for modern JavaScript projects.

**Server in Node.js**

In Node.js, a **server** is a program that listens for and responds to client requests. You can create servers using the built-in http or https modules, or frameworks like **Express.js** for additional features.

**Creating a Basic HTTP Server:**

1. **Using the http Module**:

const http = require('http');

const server = http.createServer((req, res) => {

// Set response headers

res.writeHead(200, { 'Content-Type': 'text/plain' });

// Send response data

res.end('Hello, Node.js Server!');

});

// Start the server

server.listen(3000, () => {

console.log('Server is running on http://localhost:3000');

});

1. **Explanation**:
   * **http.createServer()**: Creates an HTTP server.
   * **Callback Function (req, res)**:
     + req: Represents the incoming request object (e.g., URL, headers, etc.).
     + res: Represents the response object to send data back to the client.
   * **res.writeHead()**: Sets HTTP response status and headers.
   * **res.end()**: Ends the response and sends data to the client.

**Creating a Basic HTTPS Server:**

To create a secure HTTPS server, you need an SSL/TLS certificate.

const https = require('https');

const fs = require('fs');

// Read SSL/TLS certificate and key

const options = {

key: fs.readFileSync('server-key.pem'),

cert: fs.readFileSync('server-cert.pem'),

};

const server = https.createServer(options, (req, res) => {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello, Secure Node.js Server!');

});

server.listen(3001, () => {

console.log('HTTPS server is running on https://localhost:3001');

});

**Using Express.js (Simpler and More Feature-Rich):**

Express is a web framework for Node.js that simplifies server creation.

const express = require('express');

const app = express();

// Define a route

app.get('/', (req, res) => {

res.send('Hello, Express.js Server!');

});

// Start the server

app.listen(3000, () => {

console.log('Express server is running on http://localhost:3000');

});

**Key Concepts:**

1. **Routing**:
   * Define how the server responds to different URLs or HTTP methods.
   * Example:

javascript

Copy code

app.get('/about', (req, res) => res.send('About Page'));

1. **Middleware**:
   * Functions executed during the request-response cycle.
   * Example: Parsing JSON data.

app.use(express.json());

1. **Request and Response**:
   * req: Contains information about the HTTP request.
   * res: Used to send a response back to the client.
2. **Static Files**:
   * Serve HTML, CSS, and JavaScript files.
   * Example:

app.use(express.static('public'));

**Choosing Between http and Express.js:**

* **Use http**: For lightweight, low-level custom servers.
* **Use Express.js**: For easier routing, middleware, and advanced features like templating and RESTful APIs.

**JSON, REST API and other types of API’s**

**JSON**, **REST API**, and **other types of APIs** in the context of Node.js and other backend technologies:

**JSON (JavaScript Object Notation):**

* **Definition**: A lightweight data-interchange format that's easy for humans to read and machines to parse.
* **Usage**: Commonly used to exchange data between clients and servers.
* **Example**:

{

"name": "John Doe",

"age": 30,

"skills": ["JavaScript", "Node.js"]

}

* **In Node.js**: JSON is often used in REST APIs to send and receive data.

const data = { name: 'John', age: 30 };

const jsonData = JSON.stringify(data); // Convert object to JSON

const objData = JSON.parse(jsonData); // Convert JSON to object

**REST API (Representational State Transfer API):**

* **Definition**: A set of rules for creating web services that use HTTP requests to perform CRUD operations.
* **Key Features**:
  + **Stateless**: Each request is independent.
  + **Resource-based**: Operates on resources (e.g., users, products).
  + **HTTP Methods**:
    - GET: Retrieve data.
    - POST: Create data.
    - PUT: Update data.
    - DELETE: Remove data.
* **In Node.js**: Using **Express.js** to build a REST API:

const express = require('express');

const app = express();

app.use(express.json());

// Routes

app.get('/users', (req, res) => res.json([{ id: 1, name: 'John' }]));

app.post('/users', (req, res) => res.status(201).json(req.body));

app.listen(3000, () => console.log('REST API running on http://localhost:3000'));

**Other Types of APIs:**

1. **GraphQL API**:
   * **Definition**: A query language for APIs that lets clients request only the data they need.
   * **Example**:

query {

user(id: 1) {

name

email

}

}

* + **In Node.js**: Using **Apollo Server**:

const { ApolloServer, gql } = require('apollo-server');

const typeDefs = gql`

type User { id: ID!, name: String! }

type Query { user(id: ID!): User }

`;

const resolvers = { Query: { user: () => ({ id: 1, name: 'John' }) } };

const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => console.log(`GraphQL running at ${url}`));

1. **SOAP API (Simple Object Access Protocol)**:
   * **Definition**: A protocol for exchanging structured information using XML.
   * **Example**: Often used in enterprise-level applications.
   * **In Backend**: Handled using libraries like soap in Node.js.

const soap = require('soap');

const url = 'http://example.com/wsdl';

soap.createClient(url, (err, client) => {

client.myFunction({ param: 'value' }, (err, result) => console.log(result));

});

1. **WebSocket API**:
   * **Definition**: A protocol for real-time, full-duplex communication between client and server.
   * **Use Cases**: Chat apps, live notifications, gaming.
   * **In Node.js**: Using ws module:

const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 3000 });

server.on('connection', ws => {

ws.on('message', message => console.log(`Received: ${message}`));

ws.send('Hello Client!');

});

1. **gRPC (Google Remote Procedure Call)**:
   * **Definition**: A high-performance RPC framework that uses Protocol Buffers (protobuf) for data serialization.
   * **Use Cases**: Microservices, real-time systems.
   * **In Backend**: Handled using grpc in Node.js.

const grpc = require('@grpc/grpc-js');

const protoLoader = require('@grpc/proto-loader');

const packageDefinition = protoLoader.loadSync('service.proto');

const service = grpc.loadPackageDefinition(packageDefinition);

const server = new grpc.Server();

server.addService(service.MyService.service, {

myMethod: (call, callback) => callback(null, { message: 'Hello' }),

});

server.bindAsync('localhost:50051', grpc.ServerCredentials.createInsecure(), () => server.start());

1. **RESTful API with JSON-RPC**:
   * **Definition**: A remote procedure call protocol encoded in JSON.
   * **Example**:

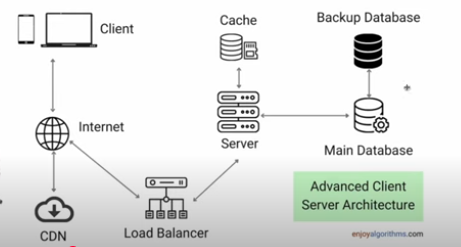
{ "jsonrpc": "2.0", "method": "getData", "params": { "id": 1 }, "id": 1 }

**Comparison of API Types:**

| **API Type** | **Data Format** | **Protocol** | **Use Case** | **Complexity** |
| --- | --- | --- | --- | --- |
| REST API | JSON, XML | HTTP | General-purpose | Simple |
| GraphQL | JSON | HTTP | Dynamic data querying | Medium |
| SOAP API | XML | HTTP, SMTP | Enterprise systems | Complex |
| WebSocket API | Binary/Text | WebSocket | Real-time communication | Medium |
| gRPC | Protobuf | HTTP/2 | Microservices, high-performance | High |

**Which One to Use?**

* **REST API**: Most common for web and mobile apps.
* **GraphQL**: For flexible querying and modern apps.
* **SOAP**: Legacy enterprise systems.
* **WebSocket**: Real-time applications.
* **gRPC**: High-performance and microservices.

****

**Methods**

### 1. ****Using REST API (HTTP Requests)****:

REST APIs use HTTP methods (GET, POST, etc.) to share data between systems.

#### Example (Node.js with Express):

**Server:**

javascript

Copy code

const express = require('express');

const app = express();

app.use(express.json()); // Parse JSON body

app.get('/data', (req, res) => {

res.json({ message: 'Hello, this is data from the server!' });

});

app.post('/data', (req, res) => {

console.log(req.body); // Log the received data

res.json({ message: 'Data received successfully!' });

});

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

**Client:**

javascript

Copy code

const fetch = require('node-fetch');

fetch('http://localhost:3000/data', { method: 'GET' })

.then(response => response.json())

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

fetch('http://localhost:3000/data', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ name: 'John', age: 30 }),

}).then(response => response.json())

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

### 2. ****Using WebSocket (Real-Time Communication)****:

WebSocket enables two-way, real-time communication between client and server.

#### Example (Node.js with ws):

**Server:**

javascript

Copy code

const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 3000 });

server.on('connection', ws => {

ws.on('message', message => {

console.log(`Received: ${message}`);

ws.send('Hello Client!');

});

});

**Client:**

javascript

Copy code

const WebSocket = require('ws');

const client = new WebSocket('ws://localhost:3000');

client.on('open', () => {

client.send('Hello Server!');

});

client.on('message', message => {

console.log(`Received: ${message}`);

});

### 3. ****Using Message Queues (e.g., RabbitMQ, Kafka)****:

Message queues facilitate asynchronous data sharing between systems.

#### Example (Node.js with amqplib for RabbitMQ):

**Producer:**

javascript

Copy code

const amqp = require('amqplib');

async function sendMessage() {

const connection = await amqp.connect('amqp://localhost');

const channel = await connection.createChannel();

const queue = 'dataQueue';

const message = 'Hello, RabbitMQ!';

await channel.assertQueue(queue);

channel.sendToQueue(queue, Buffer.from(message));

console.log('Message sent:', message);

setTimeout(() => connection.close(), 500);

}

sendMessage();

**Consumer:**

javascript

Copy code

const amqp = require('amqplib');

async function receiveMessage() {

const connection = await amqp.connect('amqp://localhost');

const channel = await connection.createChannel();

const queue = 'dataQueue';

await channel.assertQueue(queue);

channel.consume(queue, msg => {

console.log('Received:', msg.content.toString());

channel.ack(msg);

});

}

receiveMessage();

### 4. ****Using Files (Local or Cloud Storage)****:

Data can be shared using files stored locally or in cloud services like AWS S3.

#### Example (Writing and Reading JSON in Node.js):

**Write Data to File:**

javascript

Copy code

const fs = require('fs');

const data = { name: 'John', age: 30 };

fs.writeFileSync('data.json', JSON.stringify(data));

console.log('Data written to file.');

**Read Data from File:**

javascript

Copy code

const fs = require('fs');

const rawData = fs.readFileSync('data.json');

const data = JSON.parse(rawData);

console.log('Data read from file:', data);

### 5. ****Using Shared Databases****:

Databases like MongoDB or MySQL can be used to share data between systems.

#### Example (Node.js with MongoDB):

**Insert Data:**

javascript

Copy code

const { MongoClient } = require('mongodb');

async function insertData() {

const client = new MongoClient('mongodb://localhost:27017');

await client.connect();

const db = client.db('testdb');

const collection = db.collection('data');

await collection.insertOne({ name: 'John', age: 30 });

console.log('Data inserted');

await client.close();

}

insertData();

**Retrieve Data:**

javascript

Copy code

const { MongoClient } = require('mongodb');

async function fetchData() {

const client = new MongoClient('mongodb://localhost:27017');

await client.connect();

const db = client.db('testdb');

const collection = db.collection('data');

const result = await collection.findOne({ name: 'John' });

console.log('Data fetched:', result);

await client.close();

}

fetchData();

### 6. ****Using GraphQL (Flexible Data Queries)****:

GraphQL allows sharing only the requested data.

#### Example (Node.js with Apollo Server):

**Server:**

javascript

Copy code

const { ApolloServer, gql } = require('apollo-server');

const typeDefs = gql`

type Query {

hello: String

}

`;

const resolvers = {

Query: {

hello: () => 'Hello, GraphQL!',

},

};

const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => console.log(`Server ready at ${url}`));

**Client:**

javascript

Copy code

const fetch = require('node-fetch');

fetch('http://localhost:4000/', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

query: '{ hello }',

}),

})

.then(response => response.json())

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

### 7. ****Using Shared Memory (Inter-Process Communication)****:

For high-performance, low-latency data sharing within the same system.

#### Example (Using Redis for Data Sharing):

**Set Data:**

javascript

Copy code

const redis = require('redis');

const client = redis.createClient();

client.set('key', 'value', redis.print);

**Get Data:**

javascript

Copy code

const redis = require('redis');

const client = redis.createClient();

client.get('key', (err, reply) => {

console.log(reply); // Output: value

});

### Summary of Methods:

| **Method** | **Best Use Case** | **Example Technology** |
| --- | --- | --- |
| REST API | Web apps and client-server communication | Express.js, Flask |
| WebSocket | Real-time apps (chat, notifications) | ws, Socket.IO |
| Message Queues | Asynchronous communication | RabbitMQ, Kafka |
| Files | Sharing structured data | Local/Cloud storage |
| Shared Databases | Persistent and scalable data storage | MongoDB, MySQL |
| GraphQL API | Dynamic querying | Apollo, GraphQL.js |
| Shared Memory | High-speed IPC | Redis, Memcached |

Each method has its own strengths and is suited for specific scenarios based on requirements like scalability, latency, or complexity.

**REST Methods**

### REST Methods Overview

RESTful APIs use HTTP methods to perform operations on resources. Common methods include:

1. **GET**: Retrieve data from the server.
2. **POST**: Send data to the server to create a new resource.
3. **PUT**: Update an existing resource entirely.
4. **PATCH**: Update a part of an existing resource.
5. **DELETE**: Remove a resource from the server.

### Example: REST Methods with Flag Data

Imagine we are building an API for managing country flags.

#### **Server Implementation (Node.js with Express)**

const express = require('express');

const app = express();

app.use(express.json());

let flags = [

{ id: 1, country: 'India', flag: '🇮🇳' },

{ id: 2, country: 'USA', flag: '🇺🇸' },

];

// GET all flags

app.get('/flags', (req, res) => {

res.json(flags);

});

// GET a specific flag

app.get('/flags/:id', (req, res) => {

const flag = flags.find(f => f.id === parseInt(req.params.id));

if (!flag) return res.status(404).send('Flag not found');

res.json(flag);

});

// POST a new flag

app.post('/flags', (req, res) => {

const newFlag = {

id: flags.length + 1,

country: req.body.country,

flag: req.body.flag,

};

flags.push(newFlag);

res.status(201).json(newFlag);

});

// PUT to update a flag completely

app.put('/flags/:id', (req, res) => {

const flag = flags.find(f => f.id === parseInt(req.params.id));

if (!flag) return res.status(404).send('Flag not found');

flag.country = req.body.country;

flag.flag = req.body.flag;

res.json(flag);

});

// PATCH to update part of a flag

app.patch('/flags/:id', (req, res) => {

const flag = flags.find(f => f.id === parseInt(req.params.id));

if (!flag) return res.status(404).send('Flag not found');

if (req.body.country) flag.country = req.body.country;

if (req.body.flag) flag.flag = req.body.flag;

res.json(flag);

});

// DELETE a flag

app.delete('/flags/:id', (req, res) => {

const flagIndex = flags.findIndex(f => f.id === parseInt(req.params.id));

if (flagIndex === -1) return res.status(404).send('Flag not found');

flags.splice(flagIndex, 1);

res.status(204).send(); // No Content

});

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

#### **Client Code for REST Methods**

Here’s how a client (e.g., Node.js script) can interact with the API using fetch.

**Install node-fetch if needed**:

npm install node-fetch

**Client Code:**

const fetch = require('node-fetch');

const apiUrl = 'http://localhost:3000/flags';

// GET all flags

async function getAllFlags() {

const response = await fetch(apiUrl);

const data = await response.json();

console.log('All Flags:', data);

}

// GET a specific flag

async function getFlagById(id) {

const response = await fetch(`${apiUrl}/${id}`);

if (response.ok) {

const data = await response.json();

console.log(`Flag with ID ${id}:`, data);

} else {

console.log(`Flag with ID ${id} not found.`);

}

}

// POST a new flag

async function createFlag(country, flag) {

const response = await fetch(apiUrl, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ country, flag }),

});

const data = await response.json();

console.log('New Flag Created:', data);

}

// PUT to update a flag

async function updateFlag(id, country, flag) {

const response = await fetch(`${apiUrl}/${id}`, {

method: 'PUT',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ country, flag }),

});

const data = await response.json();

console.log(`Flag with ID ${id} Updated:`, data);

}

// PATCH to update part of a flag

async function patchFlag(id, partialUpdate) {

const response = await fetch(`${apiUrl}/${id}`, {

method: 'PATCH',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(partialUpdate),

});

const data = await response.json();

console.log(`Flag with ID ${id} Partially Updated:`, data);

}

// DELETE a flag

async function deleteFlag(id) {

const response = await fetch(`${apiUrl}/${id}`, { method: 'DELETE' });

if (response.ok) console.log(`Flag with ID ${id} Deleted`);

else console.log(`Flag with ID ${id} not found.`);

}

// Example Usage

(async () => {

await getAllFlags();

await createFlag('Japan', '🇯🇵');

await getFlagById(3);

await updateFlag(3, 'Japan Updated', '🏳️');

await patchFlag(3, { flag: '🇯🇵' });

await deleteFlag(3);

})();

**HTTP response codes**

HTTP response codes indicate the status of a request made to a server. These codes are grouped into five categories based on their purpose.

**1. Informational (100–199)**

Used to indicate that a request is being processed.

* **100 Continue**: Server received the initial part of the request; client should continue.
* **101 Switching Protocols**: Server agrees to switch protocols as requested by the client.

**2. Success (200–299)**

Indicates the request was successful.

* **200 OK**: Request succeeded; response contains the data (default for GET).
* **201 Created**: Request succeeded; a new resource was created (default for POST).
* **202 Accepted**: Request accepted but not yet processed.
* **204 No Content**: Request succeeded, but no content is returned.

**3. Redirection (300–399)**

Indicates that further action is needed to complete the request.

* **301 Moved Permanently**: Resource has been moved to a new URL permanently.
* **302 Found**: Temporary redirection to another URL.
* **304 Not Modified**: Client can use cached data as the resource hasn't changed.

**4. Client Errors (400–499)**

Indicates issues with the client request.

* **400 Bad Request**: Request is malformed or invalid.
* **401 Unauthorized**: Authentication is required or has failed.
* **403 Forbidden**: Client doesn’t have permission to access the resource.
* **404 Not Found**: Resource not found at the given URL.
* **405 Method Not Allowed**: HTTP method is not supported for the resource.
* **429 Too Many Requests**: Client has sent too many requests in a short time.

**5. Server Errors (500–599)**

Indicates issues with the server.

* **500 Internal Server Error**: Generic server error.
* **501 Not Implemented**: Server doesn’t support the requested functionality.
* **502 Bad Gateway**: Invalid response from an upstream server.
* **503 Service Unavailable**: Server is temporarily overloaded or under maintenance.
* **504 Gateway Timeout**: Server acting as a gateway didn’t get a response in time.

**Common Use Cases in REST APIs**

| **Scenario** | **Response Code** | **Meaning** |
| --- | --- | --- |
| Successful data retrieval | 200 OK | Data retrieved successfully. |
| Resource creation | 201 Created | A new resource was successfully created. |
| Resource update | 200 OK | Resource updated successfully. |
| Partial update (PATCH) | 200 OK / 204 No Content | Partial resource update successful. |
| Deleting a resource | 204 No Content | Resource deleted successfully. |
| Client sends invalid data | 400 Bad Request | Client needs to fix the request. |
| Unauthorized access | 401 Unauthorized | Client must authenticate. |
| Requesting a non-existent resource | 404 Not Found | Resource does not exist at the specified URL. |
| Server-side error | 500 Internal Server Error | Server failed to process the request. |

**Example in Node.js with Express:**

app.get('/example', (req, res) => {

res.status(200).json({ message: 'Success' }); // 200 OK

});

app.post('/example', (req, res) => {

if (!req.body.name) {

res.status(400).json({ error: 'Bad Request: Name is required' }); // 400 Bad Request

} else {

res.status(201).json({ message: 'Resource created', data: req.body }); // 201 Created

}

});

app.delete('/example/:id', (req, res) => {

const resourceExists = false; // Simulate missing resource

if (!resourceExists) {

res.status(404).json({ error: 'Not Found: Resource does not exist' }); // 404 Not Found

} else {

res.status(204).send(); // 204 No Content

}

});

By using these codes appropriately, APIs can clearly communicate the result of a request to the client.

**Packagelock.json.**

**Packagelock.json.**

**Packagelock.json.**

**Packagelock.json.**

**Packagelock.json.**

**Packagelock.json.**