Week 2 - Web Fundamentals & Core Modules - Part 2

Amr A Khllaf

August 15, 2025

Contents

1 API (Application Programming Interface)

1.1 Overview

An API (Application Programming Interface) is a structured interface that allows different software applications to communicate with each other. It serves as a mediator, enabling systems to exchange data and functionality regardless of their underlying implementations.

1.2 Key Characteristics

- Provides a contract of interaction with clearly defined methods and data structures
- Enables seamless integration between disparate systems and services
- Abstracts complex implementation details behind intuitive interfaces
- Facilitates secure sharing of functionality across different platforms

1.3 Common Types of APIs

1.3.1 Web APIs

- Accessible over HTTP/HTTPS protocols
- **RESTful APIs**: Use standard HTTP methods (GET, POST, PUT, DELETE) for resource manipulation
- **GraphQL APIs**: Provide flexible, query-based data retrieval with reduced over-fetching
- **SOAP APIs**: Employ XML-based messaging protocol with formal contracts (WSDL)

1.3.2 Library/SDK APIs

- Included within software development kits or packages
- Expose pre-built functions, classes, and methods
- Accelerate development through reusable components
- Examples: TensorFlow API, jQuery API

1.3.3 Operating System APIs

- Enable applications to interface with system resources
- Provide access to:
  - File system operations
  - Network communication protocols
  - Hardware interfaces
  - System services

1.4 Benefits

- Promotes modular and maintainable software architecture
- Reduces development complexity and time-to-market
- Enables powerful third-party integrations and ecosystem growth
- Supports scalable distributed systems

2 JSON (JavaScript Object Notation)

2.1 Overview

JSON is a lightweight data interchange format that has become the standard for API communication due to its simplicity and efficiency.

2.2 Key Features

- Human-readable text format with simple syntax
- Machine-parsable with native support in most programming languages
- Language-independent while based on JavaScript object notation
- Self-describing and easy to understand

2.3 Structure

```json
{
  "name": "Example Object",
  "properties": ["lightweight", "readable", "efficient"],
  "isDataFormat": true,
  "usage": {
    "primary": "API communication",
    "secondary": "Configuration files"
  }
}
```

2.4 Relationship with APIs

JSON has become the de facto standard for API data exchange because:

- It's more compact and readable than XML alternatives
- Native browser support makes it ideal for web APIs
- Parsing overhead is minimal compared to other formats
- Schema validation tools (like JSON Schema) enable robust contract enforcement

When APIs transmit data between systems, JSON provides the universally understood language that both sides can easily process and interpret.

2.5 Difference between JSON.stringify() and JSON.parse()

- `JSON.stringify()`: Converts a JavaScript object into a JSON string.
- `JSON.parse()`: Converts a JSON string back into a JavaScript object.
- We use `JSON.stringify()` when we want to send data to a server or store it in a file, and `JSON.parse()` when we want to read that data back into our application.
- Example:

```
const jsonString = JSON.stringify({ name: "Alice", age: 30 });
console.log(jsonString); // '{"name":"Alice","age":30}'

const jsonObject = JSON.parse(jsonString);
console.log(jsonObject); // { name: 'Alice', age: 30 }
```

---

3 CRUD With HTTP (getAllUsers & Add User)

3.1 Overview

- This section covers how to perform CRUD (Create, Read, Update, Delete) operations using HTTP methods in a RESTful API context.
- We will implement two key functionalities: retrieving all users and adding a new user.

3.2 Implementation

We'll create four essential RESTful API endpoints to manage user data:

1. **GET All Users** - Retrieve the complete list of users
2. **POST User** - Add a new user to the system
3. **PUT User** - Update an existing user's information
4. **DELETE User** - Remove a user from the system

Each endpoint will use the appropriate HTTP method that corresponds to its CRUD operation, following RESTful principles:

Operation	HTTP Method	Endpoint	Description
Create	POST	/users	Add a new user
Read	GET	/users	Retrieve all users
Update	PUT	/users/:id	Modify existing user data
Delete	DELETE	/users/:id	Remove a user from the system

Let's implement these endpoints with proper request handling, response formatting, and error management.

```
const http = require("http"); // Import the http module

const users = [
  // Sample user data
  {
    id: 1,
    name: "Amr",
    email: "amr@example.com",
    password: "123456",
  },
];
```

```
const server = http.createServer((req, res) => {
  // Create the HTTP server
  const { url, method } = req; // Destructure the request URL and method

  // 1. Get All Users
  if (url == "/users" && method == "GET") {
    res.writeHead(200, { "Content-Type": "application/json" }); // Set
↪   response headers
    res.end(JSON.stringify(users));
  }

  // 2. Add Users
  else if (url == "/users" && method == "POST") {
    let body = ""; // Initialize an empty string to collect the request
     ↪  body (Like Cartona)
    // Collect the request body data
    // data is an EventEmitter
    req.on("data", (chunk) => {
      body += chunk.toString(); // Convert Buffer to string
      // body += JSON.parse(chunk.toString()); // Convert Buffer to
       ↪  string
    });
    req.on("end", () => {
      // When the request ends, parse the collected data
      const newUser = JSON.parse(body);
      newUser.id = users.length + 1; // Generate new ID for every new user
      users.push(newUser);
      res.writeHead(201, { "Content-Type": "application/json" });
      res.end(JSON.stringify(newUser)); // Respond with the newly created
↪   user
      res.end(JSON.stringify({ message: "User added successfully" }));
    });
  }
});

server.listen(3000, (err) => {
  // Start the server and listen on port 3000
  if (err) {
    console.error("Error starting server:", err);
  } else {
    console.log("Server listening on port 3000...");
  }
});
```

- This code sets up an HTTP server that listens for incoming requests on port 3000. It handles two main routes: retrieving all users and adding a new user. The server responds with the appropriate status codes and JSON data based on the request type (GET or POST).
- **res:** has an `on` method to listen for events, such as `data` and `end`, when processing incoming request data.

```
else if (url == "/users" && method == "POST") {
    res.end(JSON.stringify({ message: "Success" }));
    req.on("data", (chunk) => {
      console.log(chunk);
    });
  }
  // Output: <Buffer ...> like: <Buffer 7b 22 6e 61 6d 65 22 3a 22 41 6c
  ↪  69 63 65 22 2c 22 61 67 65 22 3a 33 30 7d>
```

- **data:** is an EventEmitter that emits events when data is available to read from the stream.

---

4 Thunder Client & Status Code

- **Thunder Client** is a lightweight REST API client extension for Visual Studio Code, allowing developers to test APIs directly from the editor.
- It provides an intuitive interface for sending HTTP requests and viewing responses, making API development and testing more efficient.
- **Status Codes** are issued by a server in response to a client's request made to the server. They represent the outcome of the request and are categorized into five classes:
  - **1xx (Informational):** Request received, continuing process.
  - **2xx (Successful):** The action was successfully received, understood, and accepted.
  - **3xx (Redirection):** Further action needs to be taken in order to complete the request.
  - **4xx (Client Error):** The request contains bad syntax or cannot be fulfilled.
  - **5xx (Server Error):** The server failed to fulfill a valid request.

4.1 HTTP Response Methods

When sending HTTP responses in Node.js, there are two equivalent approaches for setting headers and status codes:

4.1.1 Method 1: Using writeHead()

```
res.writeHead(201, { "Content-Type": "application/json" });
res.end(JSON.stringify(newUser));
```

4.1.2 Method 2: Using Individual Methods

```
res.setHeader("Content-Type", "application/json");
res.statusCode = 201;
res.end(JSON.stringify(newUser));
```

4.1.3 Difference Between Methods

- **writeHead()**: Sets the status code and multiple headers in a single call. More concise and often preferred for simple cases.

- **Individual methods**: Allow setting headers and status code separately. This approach provides more flexibility when headers need to be modified at different points in your code before sending the response.

Both approaches achieve the same result, so the choice depends on your coding style and specific requirements.

---

5 Update User

5.1 Overview

Updating existing user data is a fundamental CRUD operation implemented through the HTTP PUT method in RESTful APIs.

5.2 StartsWith and Split String Methods

5.2.1 String.prototype.startsWith()

The `startsWith()` method determines whether a string begins with the characters of a specified string, returning a boolean value.

```
const url = "/users/123";
console.log(url.startsWith("/users/")); // true
```

This method is particularly useful for route handling in web servers, allowing flexible pattern matching for URL paths without requiring exact matches.

5.2.2 String.prototype.split()

The `split()` method divides a string into an ordered list of substrings based on a specified separator, returning them as an array.

```
const url = "/users/123";
const parts = url.split("/");
console.log(parts); // ['', 'users', '123']
console.log(parts[2]); // '123'
```

When handling RESTful URLs, `split()` is invaluable for extracting parameters from URL paths, such as IDs or resource identifiers.

5.2.3 Combined Usage in API Routes

These methods work together effectively when implementing dynamic API endpoints:

```
// Extract user ID from URL like '/users/42'
if (url.startsWith("/users/")) {
  const urlId = parseInt(url.split("/")[2]); // Extract user ID which is
  ↪ 123
  // Process request with extracted urlId
}
```

This pattern provides a clean way to handle parameterized routes without complex regex or external routing libraries.

5.3 Implementation

To update a user, we need to:

1. Identify the user by ID from the URL
2. Receive the updated data in the request body
3. Find and modify the specified user in our data store

```
// 3. Update User
else if (url.startsWith('/users/') && method === "PUT") {
  const urlId = parseInt(url.split('/')[2]); // Extract user ID from URL

  let body = "";
  req.on("data", (chunk) => {
    body += chunk.toString();
  });

  req.on("end", () => {
    const updatedData = JSON.parse(body);

    // Find user by ID
    const userIndex = users.findIndex(user => user.id === urlId);

    if (userIndex !== -1) {
      // Update user with new data while preserving ID
      users[userIndex] = { ...users[userIndex], ...updatedData };

      res.writeHead(200, { "Content-Type": "application/json" });
      res.end(JSON.stringify(users[userIndex]));
    } else {
      // User not found
      res.writeHead(404, { "Content-Type": "application/json" });
      res.end(JSON.stringify({ message: "User not found" }));
    }
  });
}
```

- This Line of Code is like:

```
users[userIndex] = { ...users[userIndex], ...updatedData };
// This is like:
users[userIndex].name = updatedData.name;
users[userIndex].email = updatedData.email;
users[userIndex].password = updatedData.password;
```

- Explain the first line.

The first line uses the spread operator (`...`) to create a new object that combines the existing user data with the updated data. This approach is concise and ensures that only the specified fields are updated while preserving any other existing fields in the user object.

- The Second Object will contain the updated user data, which will replace (Override) the corresponding fields in the original user object.

5.4 Key Considerations

- The `startsWith()` method enables flexible URL pattern matching
- We extract the user ID from the URL path to identify the target resource
- Error handling provides appropriate feedback when users don't exist
- Partial updates are supported by merging existing data with new properties

6 Delete User

6.1 Splice Method

The `splice()` method is a powerful array manipulation function in JavaScript that changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

6.1.1 Syntax

```
array.splice(startIndex, deleteCount, item1, item2, ...)
```

6.1.2 Parameters

- **startIndex**: The index at which to start changing the array
- **deleteCount**: (Optional) Number of elements to remove from the array
- **item1, item2, . . .**: (Optional) Elements to add to the array

6.1.3 Return Value

- Returns an array containing the deleted elements (if any)

6.1.4 Examples

Example 1: Removing Elements

```
const fruits = ["apple", "banana", "cherry", "date"];
const removed = fruits.splice(1, 2); // Remove 2 elements starting at
↪  index 1

console.log(fruits); // ["apple", "date"]
console.log(removed); // ["banana", "cherry"]
```

Example 2: Replacing Elements

```
const colors = ["red", "green", "blue"];
colors.splice(1, 1, "yellow", "orange"); // Replace 1 element at index 1
↪  with 2 new elements

console.log(colors); // ["red", "yellow", "orange", "blue"]
```

Example 3: Adding Elements Without Removing

```
const numbers = [1, 2, 5];
numbers.splice(2, 0, 3, 4); // Insert elements at index 2, without
↪  removing any

console.log(numbers); // [1, 2, 3, 4, 5]
```

Example 4: Usage in Delete User API In our delete user implementation, `splice()` is used to remove a specific user from the array:

```
const deletedUser = users.splice(userIndex, 1)[0];
```

This line does three things:

1. Removes one element at position `userIndex` from the `users` array
2. Returns an array containing the removed element
3. We access the first (and only) element with `[0]` to get the actual user object

6.1.5 When to Use

- Modifying arrays in-place when you need to add or remove elements at specific positions
- Implementing operations that require extracting a portion of an array while simultaneously modifying it
- Managing collections of objects in memory, such as user records in a simple API

6.2 Overview

Deleting user records is an essential CRUD operation implemented through the HTTP DELETE method in RESTful APIs. This operation permanently removes a specific user from the system.

6.3 Implementation

To delete a user, we need to:

1. Extract the user ID from the URL path
2. Find and remove the specified user from our data store
3. Return an appropriate response based on the operation's success

```
// 4. Delete User
else if (url.startsWith('/users/') && method === "DELETE") {
  const urlId = parseInt(url.split('/')[2]); // Extract user ID from URL

  // Find user by ID
  const userIndex = users.findIndex(user => user.id === urlId);

  if (userIndex !== -1) {
    // Remove user from array
    const deletedUser = users.splice(userIndex, 1)[0];

    res.writeHead(200, { "Content-Type": "application/json" });
    res.end(JSON.stringify({
      message: "User deleted successfully",
      deletedUser
    }));
  } else {
    // User not found
    res.writeHead(404, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ message: "User not found" }));
  }
}
```

- We can do it like this also:

```
// 4. Delete User
  else if (url.startsWith("/users/") && method === "DELETE") {
    const urlId = parseInt(url.split("/")[2]);

    const userIndex = users.findIndex((user) => user.id == urlId);

    if (userIndex == -1) {
      res.statusCode = 404;
      return res.end(JSON.stringify({ message: "User not found" }));
    }
    users.splice(userIndex, 1);
    res.statusCode = 200;
    res.end(JSON.stringify({ message: "User Deleted successfully" }));
```

6.4 Key Points

- The DELETE method should be idempotent - multiple identical requests should have the same effect as a single request
- Successful deletion operations typically return HTTP status code 200 (OK) or 204 (No Content)
- The response may include the deleted resource data or simply a confirmation message
- Proper error handling for non-existent resources prevents confusing client experiences

6.5 Best Practices

- Always verify the existence of the resource before attempting deletion
- Consider using soft deletes (marking as inactive rather than removing) for important data
- Implement appropriate authentication and authorization checks before allowing deletion
- Return clear error messages when deletion fails due to constraints or dependencies

---

7 Enhancement The Code (SendResponse)

- Abstract the response sending logic into a separate function to reduce code duplication and improve maintainability

```
function sendResponse(res, statusCode, data) {
  res.writeHead(statusCode, { "Content-Type": "application/json" });
  res.end(JSON.stringify(data));
}
```

- Apply it in the all CRUD:

```
// 1. Create User
else if (url === "/users" && method === "POST") {
  let body = "";
  req.on("data", chunk => {
    body += chunk.toString();
  });
  req.on("end", () => {
    const newUser = JSON.parse(body);
    users.push(newUser); // This will add the new user to the in-memory
↪  array not in the JSON file
    sendResponse(res, 201, { message: "User created successfully", user:
     ↪  newUser });
  });
}

// 2. Read User
else if (url.startsWith("/users/") && method === "GET") {
  const urlId = parseInt(url.split("/")[2]);
  const user = users.find(user => user.id === urlId);
  if (user) {
    sendResponse(res, 200, user);
  } else {
    sendResponse(res, 404, { message: "User not found" });
  }
}

// 3. Update User
else if (url.startsWith("/users/") && method === "PUT") {
  const urlId = parseInt(url.split("/")[2]);
  const userIndex = users.findIndex(user => user.id === urlId);
  if (userIndex !== -1) {
    let body = "";
    req.on("data", chunk => {
      body += chunk.toString();
    });
    req.on("end", () => {
      const updatedUser = JSON.parse(body);
      users[userIndex] = { ...users[userIndex], ...updatedUser };
      sendResponse(res, 200, { message: "User updated successfully", user:
       ↪  users[userIndex] }); // Send updated user data
    });
  } else {
    sendResponse(res, 404, { message: "User not found" });
  }
}
```

```
// 4. Delete User
else if (url.startsWith("/users/") && method === "DELETE") {
  const urlId = parseInt(url.split("/")[2]);
  const userIndex = users.findIndex(user => user.id === urlId);
  if (userIndex !== -1) {
    users.splice(userIndex, 1);
    sendResponse(res, 200, { message: "User deleted successfully" });
  } else {
    sendResponse(res, 404, { message: "User not found" });
  }
}
```

- The `sendResponse` function can be reused in all CRUD operations to maintain consistency and reduce code duplication.
- The Parameters are:
  - `res`: The HTTP response object, which is used to send the response back to the client.
  - `statusCode`: The HTTP status code to send.
  - `data`: The data to include in the response body.
- This function sets the response headers and sends the JSON data back to the client, making it easier to manage responses across different endpoints.

---

8 Store Data Into JSON File

- We have a problem when the server is restarted, as all data stored in memory will be lost. To persist user data, we can store it in a JSON file.

8.1 Implementation Steps

1. **Install the `fs` module**: This module is built into Node.js and allows us to interact with the file system.
2. **Create a function to read users from the JSON file**: This function will read the user data from the file and parse it into a JavaScript object.
3. **Create a function to write users to the JSON file**: This function will take the user data and write it to the file in JSON format.
4. **Update the CRUD operations to use these functions**: Instead of manipulating the in-memory `users` array directly, we will read from and write to the JSON file.

```
const fs = require("fs");
const path = require("path");

const dataFilePath = path.join(__dirname, "data", "users.json"); // Path to
↪ the JSON file

// Read users from JSON file
function readUsersFromFile() {
  try {
    const data = fs.readFileSync(dataFilePath);
    return JSON.parse(data);
  } catch (error) {
    console.error("Error reading users from file:", error);
    return [];
  }
}

// Write users to JSON file
function writeUsersToFile(users) {
  try {
    fs.writeFileSync(dataFilePath, JSON.stringify(users, null, 2));
  } catch (error) {
    console.error("Error writing users to file:", error);
  }
}
```

8.2 Understanding `path.join(__dirname, "data", "users.json")`

8.3 Parameters

- **`__dirname`**
  - A built-in Node.js variable that returns the absolute path of the directory where the current file is located.
  - Example: If your file is `/home/amr/project/server.js`, `__dirname` will be `/home/amr/project`.
- **`"data"`**
  - A folder name inside your project directory.
  - Indicates that you want to go into the `data` subdirectory.
- **`"users.json"`**
  - The file name that stores the users' data.
  - Specifies the target file inside the `data` folder.

8.4 Examples

1. With `"data"` (file inside a `data` folder)

project/ server.js data/ users.json

```
const dataFilePath = path.join(__dirname, "data", "users.json");
```

2. Without `"data"` (file in the root directory)

project/ server.js users.json

```
const dataFilePath = path.join(__dirname, "users.json");
```

- Note: If you remove "data" but the file is actually inside the data folder, Node.js will throw: `Error: ENOENT: no such file or directory, open '/path/to/your/project/data/users.js`
- Applying the same logic on our CRUD operations, we need to ensure that all data manipulations are reflected in the JSON file as well.

```
const fs = require("fs");
let users = JSON.parse(fs.readFileSync("./users.json"));

const server = http.createServer((req, res) => {
  // Create the HTTP server
  const { url, method } = req; // Destructure the request URL and method

  function sendResponse(statusCode, msg) {
    res.statusCode = statusCode;
    return res.end(JSON.stringify(msg));
  }

  // 1. Get All Users
  if (url == "/users" && method == "GET") {
    // res.writeHead(200, { "Content-Type": "application/json" });
    // res.end(JSON.stringify(users));
    sendResponse(200, users);
  }

  // 2. Add User
  else if (url == "/users" && method == "POST") {
    let cartona = "";
    req.on("data", (chunk) => {
      // console.log(chunk);
      cartona += chunk;
    });
    req.on("end", () => {
      let newUser = JSON.parse(cartona);
      // Assign a new id to the user
      newUser.id = users.length + 1;
      // console.log(cartona);
      users.push(newUser); // This will be add in RAM
      fs.writeFileSync("users.json", JSON.stringify(users));
      // res.statusCode = 201;
      // // res.end(JSON.stringify(newUser));
      // res.end(JSON.stringify({ message: "User added successfully" }));

      sendResponse(200, { message: "User added successfully" });
    });
  }

  // 3. Update User
  else if (url.startsWith("/users/") && method === "PUT") {
    const urlId = parseInt(url.split("/")[2]);
    const userIndex = users.findIndex((user) => user.id == urlId);

    if (userIndex == -1) {
      // res.statusCode = 404;

      // return res.end(JSON.stringify({ message: "User not found" }));
      sendResponse(404, { message: "User not found" });
    }
    req.on("data", (chunk) => {
      const updatedData = JSON.parse(chunk);
      users[userIndex].name = updatedData.name;
      users[userIndex].email = updatedData.email;
      users[userIndex].password = updatedData.password;

      fs.writeFileSync("users.json", JSON.stringify(users));

      // res.statusCode = 200;
      // res.end(JSON.stringify({ message: "User updated successfully"
      ↪  }));
      sendResponse(200, { message: "User updated successfully" });
    });
  }

  // 4. Delete User
  else if (url.startsWith("/users/") && method === "DELETE") {
    const urlId = parseInt(url.split("/")[2]);

    const userIndex = users.findIndex((user) => user.id == urlId);

    if (userIndex == -1) {
      // res.statusCode = 404;
      // return res.end(JSON.stringify({ message: "User not found" }));
      sendResponse(404, { message: "User not found" });
    }
    users.splice(userIndex, 1);
    fs.writeFileSync("users.json", JSON.stringify(users));
    // res.statusCode = 200;
    // res.end(JSON.stringify({ message: "User Deleted successfully" }));
    sendResponse(200, { message: "User Deleted successfully" });
  } else {
    // res.statusCode = 404;
    // res.end(JSON.stringify({ message: "Route Not Found!" }));
    sendResponse(404, { message: "Route Not Found!" });
  }
});

server.listen(3000, (err) => {
  if (err) {
    console.log("Error Starting Server", err);
  } else {
    console.log("Server listening on port 3000...");
  }
});
```

- This code demonstrates a simple CRUD (Create, Read, Update, Delete) API for managing users in a JSON file using Node.js.
- It uses the built-in `fs` module to read and write the JSON file synchronously.
- The server listens on port 3000 and handles different HTTP methods (GET, POST, PUT, DELETE) for the `/users` endpoint.
- The API allows clients to perform CRUD operations on the user data stored in the JSON file.

---

9 More About FS Module

The `fs` (File System) module in Node.js provides an API for interacting with the file system. It allows you to read, write, and manipulate files and directories. Here are some key features and methods of the `fs` module:

10 More About FS Module

The `fs` (File System) module in Node.js provides an API for interacting with the file system. It allows you to read, write, and manipulate files and directories. Here are some key features and methods of the `fs` module:

1. **Reading Files**:

- `fs.readFileSync(path, options)`: Reads the contents of a file synchronously.

```
const fs = require("fs");
const data = fs.readFileSync("users.json", "utf8");
console.log(data); // Output: {"users":[{"id":1,"name":"John"}]}
```

- `fs.readFile(path, options, callback)`: Reads the contents of a file asynchronously.

```
const fs = require("fs");
fs.readFile("users.json", "utf8", (err, data) => {
  if (err) throw err;
  console.log(data); // Output: {"users":[{"id":1,"name":"John"}]}
});
```

2. **Writing Files**:

- `fs.writeFileSync(path, data, options)`: Writes data to a file synchronously.

```
const fs = require("fs");
fs.writeFileSync("output.txt", "Hello World", "utf8");
// Creates or overwrites output.txt with "Hello World"
```

- `fs.writeFile(path, data, options, callback)`: Writes data to a file asynchronously.

```
const fs = require("fs");
fs.writeFile("output.txt", "Hello World", "utf8", (err) => {
  if (err) throw err;
  console.log("File written successfully");
});
```

3. **Appending to Files**:

- `fs.appendFileSync(path, data, options)`: Appends data to a file synchronously to the end of the file.

```
const fs = require("fs");
fs.appendFileSync("log.txt", "\nNew log entry", "utf8");
// Adds "New log entry" to the end of log.txt
```

- `fs.appendFile(path, data, options, callback)`: Appends data to a file asynchronously.

```
const fs = require("fs");
fs.appendFile("log.txt", "\nAsync log entry", "utf8", (err) => {
  if (err) throw err;
  console.log("Data appended to file");
});
```

4. **Copying Files**:

- `fs.copyFileSync(src, dest, flags)`: Copies a file synchronously.

```
const fs = require("fs");
fs.copyFileSync("source.txt", "destination.txt");
// Copies source.txt to destination.txt
```

- `fs.copyFile(src, dest, flags, callback)`: Copies a file asynchronously.

```
const fs = require("fs");
fs.copyFile("source.txt", "destination.txt", (err) => {
  if (err) throw err;
  console.log("File copied successfully");
});
```

5. **Removing Files and Directories**:

- `fs.unlinkSync(path)`: Deletes a file synchronously.

```
const fs = require("fs");
fs.unlinkSync("temporary.txt");
// temporary.txt is now deleted
```

- `fs.unlink(path, callback)`: Deletes a file asynchronously.

```
const fs = require("fs");
fs.unlink("temporary.txt", (err) => {
  if (err) throw err;
  console.log("File deleted successfully");
});
```

- `fs.rmSync(path, options)`: Recursively removes files and directories synchronously.

```
const fs = require("fs");
fs.rmSync("directory", { recursive: true, force: true });
// Removes directory and all its contents
```

- `fs.rmdirSync(path, options)`: Removes a directory synchronously.

```
const fs = require("fs");
fs.rmdirSync("emptyDirectory");
// Removes the empty directory
```

- `fs.rmdir(path, options, callback)`: Removes a directory asynchronously.

```
const fs = require("fs");
fs.rmdir("emptyDirectory", (err) => {
  if (err) throw err;
  console.log("Directory removed successfully");
});
```

6. **Working with Directories**:

- `fs.mkdirSync(path, options)`: Creates a directory synchronously.

```
const fs = require("fs");
fs.mkdirSync("newFolder");
// Creates a directory named "newFolder"
```

- `fs.mkdir(path, options, callback)`: Creates a directory asynchronously.

```
const fs = require("fs");
fs.mkdir("newFolder", (err) => {
  if (err) throw err;
  console.log("Directory created successfully");
});
```

- `fs.mkdtempSync(prefix, options)`: Creates a temporary directory synchronously.

```
const fs = require("fs");
const tempDir = fs.mkdtempSync("prefix-");
console.log("Temporary directory created:", tempDir);
```

- `fs.readdirSync(path, options)`: Reads the contents of a directory synchronously.

```
const fs = require("fs");
const files = fs.readdirSync("./");
console.log(files); // Output: ['file1.txt', 'file2.js', 'folder1', ...]
```

- `fs.readdir(path, options, callback)`: Reads the contents of a directory asynchronously.

```
const fs = require("fs");
fs.readdir("./", (err, files) => {
  if (err) throw err;
  console.log(files); // Output: ['file1.txt', 'file2.js', 'folder1', ...]
});
```

7. **File Metadata**:

- `fs.statSync(path)`: Returns metadata about a file synchronously.

```
const fs = require("fs");
const stats = fs.statSync("users.json");
console.log(stats.size); // Output: file size in bytes, e.g., 1024
console.log(stats.isFile()); // Output: true
console.log(stats.isDirectory()); // Output: false
```

- `fs.stat(path, callback)`: Returns metadata about a file asynchronously.

```
const fs = require("fs");
fs.stat("users.json", (err, stats) => {
  if (err) throw err;
  console.log(`Last modified: ${stats.mtime}`);
  // Output: Last modified: 2023-06-15T14:12:55.000Z
});
```

8. **Error Handling**:

```
const fs = require("fs");
try {
  const data = fs.readFileSync("nonexistent.txt", "utf8");
} catch (err) {
  console.error("Error reading file:", err.message);
  // Output: Error reading file: ENOENT: no such file or directory, open
   ↪  'nonexistent.txt'
}

fs.readFile("nonexistent.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Async error:", err.message);
    return; // Output: Async error: ENOENT: no such file or directory,
     ↪  open 'nonexistent.txt'
  }
  console.log(data);
});
```

In the context of the CRUD API example, the `fs` module is used to read the initial user data from a JSON file, and all changes (additions, updates, deletions) are written back to the file to persist the data.

---

11 Sending Files Using HTTP Module & Important Concept

11.1 Overview

In this section, we will explore how to send files over HTTP using the built-in `http` module in Node.js. This is particularly useful for building APIs that need to serve files, such as images, documents, or any other type of file.

11.1.1 Node-Snippets VsCode Extension

- Node-Snippets: is a Visual Studio Code extension that provides a collection of useful code snippets for Node.js development.
- It helps developers write code faster by providing ready-to-use snippets for common tasks, such as creating HTTP servers, handling requests, and working with the file system.

11.2 Function Chaining

Function chaining is a programming pattern where multiple method calls are linked together in a single statement. Each method returns an object that contains the next method in the chain.

11.2.1 Key Benefits

- Improves code readability by reducing variable declarations
- Creates more compact and expressive code
- Enables fluent interfaces for better API design

11.2.2 Example in JavaScript

```javascript
// Without chaining
const numbers = [1, 2, 3, 4, 5];
const filtered = numbers.filter((num) => num > 2);
const doubled = filtered.map((num) => num * 2);
const sum = doubled.reduce((total, num) => total + num, 0);

// With chaining
const result = [1, 2, 3, 4, 5]
  .filter((num) => num > 2)
  .map((num) => num * 2)
  .reduce((total, num) => total + num, 0);
```

11.2.3 Implementation in Custom Classes

To enable chaining in your own classes, ensure that methods return `this`:

```javascript
class QueryBuilder {
  constructor() {
    this.query = {};
  }

  where(key, value) {
    this.query[key] = value;
    return this;
  }

  limit(count) {
    this.query.limit = count;
    return this;
  }

  execute() {
    console.log(`Executing query: ${JSON.stringify(this.query)}`);
    return this;
  }
}

// Usage
const query = new QueryBuilder()
```

```
    .where("status", "active")
    .where("age", 30)
    .limit(10)
    .execute();
```

11.2.4 Simple Chaining Nested Function Example:

Function chaining can also be implemented using nested functions that return objects with methods:

```
function x() {
  return {
    y: function (z) {
      console.log(z);
      return this; // Enables further chaining
    },
    z: function () {
      console.log("Additional method");
      return this;
    },
  };
}

// Chaining multiple method calls
x().y("Hello, World!").z();
// Output:
// Hello, World!
// Additional method
```

This pattern is commonly used in many JavaScript libraries like jQuery, Lodash, and modern JavaScript features like Promise chains.

Function chaining creates cleaner, more maintainable code when used appropriately.

11.3 Difference between writeHead, setHeader, and statusCode

When sending HTTP responses in Node.js, there are multiple ways to set status codes and headers:

11.3.1 writeHead Method

```
res.writeHead(200, { "Content-Type": "application/json" });
```

- Sets both status code and multiple headers in a single call
- Cannot be called after headers have been sent
- More concise when setting multiple headers at once

11.3.2 Individual Methods

```
res.statusCode = 200;
res.setHeader("Content-Type", "application/json");
```

- Sets status code and headers separately

- More flexible when headers need to be set conditionally
- Allows adding headers at different points in the code
- `setHeader` can be called multiple times to set different headers

11.3.3 Key Differences

1. **Order of operations**: `writeHead` must be called before any response data is sent
2. **Flexibility**: Individual methods allow for more granular control
3. **Readability**: `writeHead` is more compact for simple cases
4. **Modification**: Individual headers can be modified after setting with `setHeader`, but not with `writeHead`

Both approaches are valid and achieve the same result. Choose based on your specific code structure and requirements.

12 Sending Files Using HTTP Module

12.1 Introduction

Sending files to clients is a fundamental feature of web servers. The Node.js HTTP module provides mechanisms to serve various file types with appropriate content types and efficient data streaming.

12.2 Implementation Approach

- We can do it like this:

```
var http = require("http");
const fs = require("fs");
const server = http.createServer(function (request, response) {
  if (request.url == "/" && request.method == "GET") {
    const html = fs.readFileSync("./index.html");

    response.write(html);
    response.end(html);
  }
});

server.listen(3000);

console.log("Server running at http://127.0.0.1:3000/");
```

- Difference between response.write and response.end:
  - `response.write` is used to send a chunk of the response body to the client, while - `response.end` is used to signal that the response is complete. You can call `response.write` multiple times to send multiple chunks, **but you must call `response.end` exactly once to finish the response.**
- Example:

```
response.write("Hello, ");
response.write("World!");
response.end();
```

- `response.end()` must be called to complete the response and send it to the client.
- If you forget to call `response.end()`, the client will hang indefinitely, waiting for the response to finish.

12.2.1 Ending HTML and CSS and JS and Image Files

To properly end responses for different file types, we need to set the correct content type and ensure the response is completed.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    ↪  />
    <title>Document</title>
    <link rel="stylesheet" href="./style.css" />
  </head>
  <body>
    <h1>Lorem ipsum dolor sit amet.</h1>
    <img src="/image" alt="" />
    <script src="./test.js"></script>
  </body>
</html>
```

```javascript
const http = require("http");
const fs = require("fs");
const path = require("path");

const server = http.createServer((request, response) => {
  if (request.url === "/" && request.method === "GET") {
    const html = fs.readFileSync("./index.html");
    response.setHeader("Content-Type", "text/html");
    response.end(html);
  } else if (request.url === "/style.css" && request.method === "GET") {
    const css = fs.readFileSync("./style.css");
    response.setHeader("Content-Type", "text/css");
    response.end(css);
  } else if (request.url === "/test.js" && request.method === "GET") {
    const js = fs.readFileSync("./test.js");
    response.setHeader("Content-Type", "text/javascript");
    response.end(js);
  } else if (request.url === "/image" && request.method === "GET") {
    const img = fs.readFileSync("./FB_IMG_1577686813012.jpg");
    response.setHeader("Content-Type", "image/jpeg");
    response.end(img);
  } else {
    response.statusCode = 404;
    response.end("Not Found");
  }
});

server.listen(3000);
console.log("Server running at http://127.0.0.1:3000/");
```

12.2.2 What Happens Behind the Scenes

1. **Browser Request Flow:**
   - When you navigate to `http://127.0.0.1:3000/`, the browser sends a GET request for `/`
   - After receiving the HTML, the browser parses it and finds references to CSS, JS, and image files
   - The browser automatically sends additional requests for each resource
2. **Server Response Process:**
   - For each request, the server:
     1. Identifies the requested resource path
     2. Reads the corresponding file from disk
     3. Sets the appropriate Content-Type header for the browser to interpret correctly
     4. Sends the file content in the response
3. **Content-Type Importance:**
   - Without proper Content-Type headers, browsers might:
     - Display text files instead of rendering HTML
     - Show raw CSS code instead of applying styles
     - Display binary data as text instead of showing images
4. **Optimization Opportunities:**
   - Using `response.end(data)` directly instead of `response.write(data)` followed by `response.end()` is more efficient
   - Setting correct MIME types helps the browser process resources properly
   - Error handling should be added for missing files
   - For larger files, streaming with `fs.createReadStream().pipe(response)` would be more memory-efficient
5. **Resource Loading Sequence:**
   - HTML loads first (critical path)
   - CSS files load next (render blocking)
   - Images load as they're encountered
   - JavaScript executes based on placement in HTML

This approach creates a simple but functional static file server that handles multiple file types with appropriate content types.

12.2.3 Basic File Sending

To send files using the HTTP module, we need to:

1. Read the file from the file system
2. Set the appropriate content type header
3. Stream the file contents to the client

```
const http = require("http");
const fs = require("fs");
const path = require("path");

const server = http.createServer((req, res) => {
  // Determine file path based on request
  const filePath = path.join(__dirname, "public", "example.pdf");

  // Set content type based on file extension
  res.setHeader("Content-Type", "application/pdf");

  // Create read stream and pipe to response
  const fileStream = fs.createReadStream(filePath);
  fileStream.pipe(res);

  // Handle stream errors
  fileStream.on("error", (error) => {
    res.statusCode = 404;
    res.end("File not found or error occurred");
  });
});

server.listen(3000, () => {
  console.log("Server listening on port 3000");
});
```

12.2.4 Content Type Mapping

For a robust file server, we should map file extensions to appropriate MIME types:

```
const contentTypes = {
  ".html": "text/html",
  ".css": "text/css",
  ".js": "text/javascript",
  ".json": "application/json",
  ".png": "image/png",
  ".jpg": "image/jpeg",
  ".gif": "image/gif",
  ".pdf": "application/pdf",
};

// Get content type based on file extension
const ext = path.extname(filePath);
const contentType = contentTypes[ext] || "application/octet-stream";
```

12.3 Advanced Features

12.3.1 Content-Disposition Header

To force browser to download rather than display a file:

```
res.setHeader("Content-Disposition", 'attachment;
↪  filename="download.pdf"');
```

12.3.2 Range Requests

For large file support with partial content requests:

```
if (req.headers.range) {
  const { range } = req.headers;
  const parts = range.replace(/bytes=/, "").split("-");
  const start = parseInt(parts[0], 10);
  const end = parts[1] ? parseInt(parts[1], 10) : fileSize - 1;

  res.writeHead(206, {
    "Content-Range": `bytes ${start}-${end}/${fileSize}`,
    "Accept-Ranges": "bytes",
    "Content-Length": end - start + 1,
    "Content-Type": contentType,
  });

  fs.createReadStream(filePath, { start, end }).pipe(res);
} else {
  // Normal file serving
}
```

12.4 Best Practices

1. **Error handling**: Always handle file read errors gracefully
2. **Content types**: Set the correct MIME type for each file type
3. **Streaming**: Use streams instead of loading entire files into memory
4. **Caching**: Implement appropriate caching headers for static files
5. **Security**: Validate file paths to prevent directory traversal attacks

Implementing these practices ensures efficient and secure file delivery through your Node.js HTTP server.

---

13 Get Data From HTML Form

13.1 HTML Form Structure

Forms in HTML provide a structured way for users to submit data to servers. The form's `action` attribute specifies where to send the data, while the `method` attribute defines how it should be sent.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
     ↪  />
    <title>Document</title>
    <link rel="stylesheet" href="./style.css" />
  </head>
  <body>
    <form action="/addUser" method="post">
      <input type="text" name="username" placeholder="Enter your username"
       ↪  />
      <input type="email" name="email" placeholder="Enter your email" />
      <button type="submit">Add User</button>
    </form>
  </body>
</html>
```

13.2 Handling Form Data with Node.js

When a form is submitted, the server needs to parse the incoming data. Node.js provides the `querystring` module to handle this efficiently.

13.2.1 The Query String Module (Core)

The `querystring` module parses URL query parameters or form data into JavaScript objects:

```javascript
const queryString = require("querystring");

// Parse URL query parameters
const parsedUrl = new URL(req.url, `http://${req.headers.host}`);
const queryParams = queryString.parse(parsedUrl.search);

console.log(queryParams);
```

13.2.2 Complete Server Implementation

This example demonstrates how to:

1. Serve HTML, CSS, JavaScript, and image files
2. Process form submissions
3. Parse form data using the querystring module
4. Redirect after form submission

```javascript
const http = require("http");
const fs = require("fs");
const qs = require("querystring");

const server = http.createServer((request, response) => {
  const { url, method } = request;

```

```
// Serve HTML file
if (url === "/" && method === "GET") {
  const html = fs.readFileSync("./index.html");
  response.setHeader("Content-Type", "text/html");
  response.write(html);
}
// Serve CSS file
else if (url === "/style.css" && method === "GET") {
  const css = fs.readFileSync("./style.css");
  response.setHeader("Content-Type", "text/css");
  response.write(css);
}
// Serve JavaScript file
else if (url === "/test.js" && method === "GET") {
  const js = fs.readFileSync("./test.js");
  response.setHeader("Content-Type", "text/javascript");
  response.write(js);
}
// Serve image file
else if (url === "/image" && method === "GET") {
  const img = fs.readFileSync("./FB_IMG_1577686813012.jpg");
  response.setHeader("Content-Type", "image/jpeg");
  response.write(img);
}
// Process form submission
else if (url === "/addUser" && method === "POST") {
  let body = "";

  // Collect data chunks as they arrive
  request.on("data", (chunk) => {
    body += chunk.toString();
  });

  // Process the complete request body
  request.on("end", () => {
    // Raw form data (e.g., "username=John&email=john@example.com")
    console.log("Raw form data:", body);

    // Parsed form data as object (e.g., { username: "John", email:
    ↪  "john@example.com" })
    const formData = qs.parse(body);
    console.log("Parsed form data:", formData);

    // Here you would typically save the data or perform other
    ↪  operations

    // Redirect back to the home page
    response.writeHead(302, { Location: "/" });
    response.end();
    return;
```

```
      });
    }

    // End the response if not already ended
    response.end();
  });

  server.listen(3000, () => {
    console.log("Server running at http://127.0.0.1:3000/");
  });
```

This implementation demonstrates how to handle different types of requests, parse form data, and properly set content types for various file types, ensuring a complete and efficient web server.