Introduction to Node JS | Cheat Sheet:

Concepts in Focus

1. MERN Stack

MERN stands for **MongoDB**, **Express JS**, **React JS** and **Node JS**.

It is a **JavaScript Stack** that is used for easier & faster deployment of full-stack web applications.

2. Node JS

Node JS is a JavaScript environment that executes JavaScript code outside a web browser.

Why Node JS?

- Cross-Platform (Windows, Linux, Mac OS X, etc.)
- Huge number of third-party packages
- Open Source
- Massive Community

3. Running JavaScript Using Node JS

We can run JavaScript using Node JS in 2 ways.

- Node REPL (Similar to browser console)
- Node CLI

3.1 Node REPL (Read-Eval-Print-Loop)

Type node in the terminal and press Enter.

root@123# node

```
Welcome to Node.js v12.18.3.
Type ".help" for more information.
> const a = 1
undefined
> const b = 2
undefined
> a+b
3
>
```

Type .exit and press Enter to exit from the Node REPL.

```
root@123# node
Welcome to Node.js v12.18.3.
Type ".help" for more information.
> const a = 1
undefined
> const b = 2
undefined
> a+b
3
> .exit
root@123:/home/workspace#
```

3.2 Node CLI

We can write JavaScript to a file and can run using Node CLI.

//index.js

```
const greetings = (name) => {
  console.log(`Hello ${name}`);
};

greetings("Raju");
greetings("Abhi");
```

Output:

```
1root@123# node index.js
2. Hello Raju
3. Hello Abhi
```

Note:

Save the file whenever the code changes.

4. Module:

In Node JS, each JavaScript file is treated as a separate module. These are known as the **Common JS/Node JS** Modules.

To access one module from another module, we have Module Exports.

- Common JS Module Exports
    - Default Exports
    - Named Exports
- Modern JS Module Exports
    - Default Exports
    - Named Exports

4.1 Common JS Module Exports

4.1.1 Default Exports

Exporting Module

The
module.exports
is a special object included in every JavaScript file in the Node JS application by default.

//calculator.js

```
const add = (a, b) => {
  return a + b;
};
module.exports = add;
```

Importing Module:

To import a module which is the local file, use the **require()** function with the **relative path** of the module (file name).

//index.js

```
const add = require("./calculator");

console.log(add(6, 3));
```

OutPut:

1.root@123# node index.js

2. 9

4.1.2 Named Exports

We can have multiple named exports per module.

Exporting Module

//calculator.js

```
const add = (a, b) => {
  return a + b;
};
const sub = (a, b) => {
  return a - b;
};


exports.add = add;
exports.sub = sub;
```

Importing Module:

//index.js

```
const { add, sub } = require("./calculator");
console.log(add(6, 3));
console.log(sub(6, 3));
```

Output:

root@123# node index.js

9

3

4.2 Modern JS Module Exports

Modern JS Modules are known as **ES6 Modules**.

The **export** and **import** keywords are introduced for exporting and importing one or more members in a module.

4.2.1 Default Exports

Exporting Module

//calculator.mjs

```
const add = (a, b) => {
  return a + b;
};
export default add;
```

Importing Module:

//index.mjs

```
import add from "./calculator.mjs";
console.log(add(6, 3));
```

OutPut:

```
root@123# node index.mjs
9
```

4.2.2 Named Exports

Exporting Module

//calculator.mjs

```
export const add = (a, b) => {
  return a + b;
};
export const sub = (a, b) => {
  return a - b;
};
```

Importing Module:

//index.mjs

```
import { add, sub } from "./calculator.mjs";
```

```
console.log(add(6, 3));
console.log(sub(6, 3));
```

Output:

```
root@123# node index.mjs
9
3
```

Note:
We need to specify .mjs extension while importing ES6 Modules.
We may or may not need to specify .js while importing Common JS Modules.

Common JS Module Exports | Reading Material:

Concepts in Focus

- Default Exports
  - Exporting a variable while defining
  - Exporting a variable after defining
  - Exporting a value or an expression
  - Exporting a function while defining
  - Exporting a function after defining
  - Exporting a class while defining
  - Exporting a class after defining

- Named Exports
  - Exporting multiple variables while defining
  - Exporting multiple variables after defining
  - Exporting multiple values and expressions
  - Exporting multiple functions while defining
  - Exporting multiple functions after defining
  - Exporting multiple classes while defining
  - Exporting multiple classes after defining

Let's see different scenarios that we may come across while exporting.

1. Default Exports

With Default Exports, we can import modules with any name.

1.1 Exporting a variable while defining

We cannot export boolean, number, string, null, undefined, objects, and arrays while defining.

Example:

//sample.js

```
module.exports = let value = 5;
```

//index.js

```
const num = require("./sample.js");
```

Output:

```
root@123# node index.js
/index.js:3
module.exports = let value = 5;
                 ^
SyntaxError: Unexpected identifier
    at wrapSafe (internal/modules/cjs/loader.js:1053:16)
    ...
```

1.2 Exporting a variable after defining

We can export boolean, number, string, null, undefined, objects, and arrays after defining.

Example:

//sample.js

```
let value = 5;

module.exports = value;
```

//index.js

```
const value = require("./sample.js");

console.log(value);
```

Output:

```
root@123# node index.js
5
```

1.3 Exporting a value or an expression

We can export a value or an expression directly.

Example:

//sample.js

```
module.exports = 5 * 3;
```

//index.js

```
const result = require("./sample.js");

console.log(result);
```

Output:

```
root@123# node index.js
15
```

1.4 Exporting a function while defining

We can export a function while defining.

Example:

//sample.js

```
module.exports = function (num1, num2) {
  return num1 + num2;
};
```

//index.js

```
const sum = require("./sample.js");

console.log(sum(2, 6));
```

Output:

```
root@123# node index.js
8
```

1.5 Exporting a function after defining

We can export a function after defining.

Example:

//sample.js

```
function sum(num1, num2) {
  return num1 + num2;
}
module.exports = sum;
```

//index.js

```
const sum = require("./sample.js");

console.log(sum(2, 6));
```

Output:

```
root@123# node index.js
8
```

1.6 Exporting a class while defining

We can export a class while defining.

Example:

//sample.js

```
module.exports = class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
};
```

//index.js

```
const StudentDetails = require("./sample.js");

const studentDetails = new StudentDetails("Ram", 15);

console.log(studentDetails);

console.log(studentDetails.name);
```

Output:

```
root@123# node index.js
StudentDetails { name: 'Ram', age: 15 }
Ram
```

1.7 Exporting a class after defining

We can export a class after defining.

Example:

//sample.js

```
class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
module.exports = StudentDetails;
```

//index.js

```
const StudentDetails = require("./sample.js");

const studentDetails = new StudentDetails("Ram", 15);

console.log(studentDetails);

console.log(studentDetails.name);
```

Output:

```
root@123# node index.js
StudentDetails { name: 'Ram', age: 15 }
Ram
```

2. Named Exports

2.1 Exporting multiple variables while defining

We cannot export boolean, number, string, null, undefined, objects, and arrays while defining.

Example:

//sample.js

```
exports.value = let value = 5;
exports.studentName = let studentName = "Rahul";
```

//index.js

```
const { value, studentName } = require("./sample");

console.log(value);

console.log(studentName);
```

Output:

```
root@123# node index.js
exports.value = let value = 5;
                ^^^^^

SyntaxError: Unexpected identifier
    at wrapSafe (internal/modules/cjs/loader.js:1053:16)
```

2.2 Exporting multiple variables after defining

We can export multiple variables after defining.

Example:

//sample.js

```
let value = 5;

exports.value = value;

let studentName = "Rahul";

exports.studentName = studentName;
```

//index.js

```
const { value, studentName } = require("./sample");

console.log(value);

console.log(studentName);
```

Output:

```
root@123# node index.js
5
Rahul
```

2.3 Exporting multiple values and expressions

We can export multiple values and expressions.

Example:

//sample.js

```
let value = 2;

exports.sum = 2 + 3;

exports.sub = 3 – value;
```

//index.js

```
const { sum, sub } = require("./sample");

console.log(sum);

console.log(sub);
```

Output:

```
root@123# node index.js
5
1
```

2.4 Exporting multiple functions while defining

We can export multiple functions while defining.

Example:

//sample.js

```
exports.sum = function (num1, num2) {
  return num1 + num2;
};
exports.sub = function sub(num1, num2) {
  return num1 - num2;
};
```

//index.js

```
const { sum, sub } = require("./sample");
console.log(sum(2, 6));
console.log(sub(8, 3));
```

Output:

```
root@123# node index.js
8
5
```

2.5 Exporting multiple functions after defining

We can export multiple functions after defining.

Example:

//sample.js

```
function sum(num1, num2) {
  return num1 + num2;
}
exports.sum = sum;
function sub(num1, num2) {
  return num1 - num2;
}
exports.sub = sub;
```

//index.js

```
const { sum, sub } = require("./sample");
console.log(sum(2, 6));
console.log(sub(8, 3));
```

Output:

```
root@123# node index.js
8
5
```

2.6 Exporting multiple classes while defining

We can export multiple classes while defining.

Example:

//sample.js

```
exports.studentDetails = class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
};

exports.carDetails = class CarDetails {
  constructor(name, age) {
    this.name = name;
    this.speed = age;
  }
};
```

//index.js

```
const { studentDetails, carDetails } = require("./sample.js");

const newStudentDetails = new studentDetails("Ram", 15);
console.log(newStudentDetails);
console.log(newStudentDetails.name);

const newCarDetails = new carDetails("Alto", "60kmph");
console.log(newCarDetails);
console.log(newCarDetails.name);
```

Output:

```
root@123# node index.js
StudentDetails { name: 'Ram', age: 15 }
Ram
CarDetails { name: 'Alto', speed: '60kmph' }
Alto
```

2.7 Exporting multiple classes after defining

We can export multiple classes after defining.

Example:

//sample.js

```
class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
exports.studentDetails = StudentDetails;

class CarDetails {
  constructor(name, age) {
    this.name = name;
    this.speed = age;
  }
}

exports.carDetails = CarDetails;
```

//index.js

```
const { studentDetails, carDetails } = require("./sample.js");

const newStudentDetails = new studentDetails("Ram", 15);
console.log(newStudentDetails);
console.log(newStudentDetails.name);

const newCarDetails = new carDetails("Alto", "60kmph");
console.log(newCarDetails);
console.log(newCarDetails.name);
```

Output:

```
root@123# node index.js
StudentDetails { name: 'Ram', age: 15 }
Ram
CarDetails { name: 'Alto', speed: '60kmph' }
Alto
```

ES6 Module Exports | Reading Material

Concepts in Focus

Let's see different scenarios that we may come across while exporting.

1. Default Exports

With Default Exports, we can import modules with any name.

1.1 Exporting a variable while defining

We cannot export boolean, number, string, null, undefined, objects, and arrays while defining.

Example:

//sample.mjs

```
export default let value = 5;
```

//index.mjs

```
import value from "./sample.mjs";
console.log(value);
```

Output:

```
root@123# node index.mjs
(node:31964) ExperimentalWarning: The ESM module loader is experimental.
file:///index.mjs:1
```

```
export default let value = 5;
               ^^^
SyntaxError: Unexpected strict mode reserved word
```

1.2 Exporting a variable after defining

We can export boolean, number, string, null, undefined, objects, and arrays after defining.

Example:

//sample.mjs

```
let a = 5;
export default a;
```

//index.mjs

```
import a from "./sample.mjs";
console.log(a);
```

Output:

```
root@123# node index.mjs
(node:32665) ExperimentalWarning: The ESM module loader is experimental.
5
```

1.3 Exporting a value or an expression

We can export a value or an expression directly.

Example:

//sample.mjs

```
export default 5 * 3;
```

//index.mjs

```
import result from "./sample.mjs";

console.log(result);
```

Output:

```
root@123# node index.mjs
(node:4071) ExperimentalWarning: The ESM module loader is experimental.
15
```

1.4 Exporting a function while defining

We can export a function while defining.

Example:

//sample.mjs

```
export default function (num1, num2) {
  return num1 + num2;
}
```

//index.mjs

```
import sum from "./sample.mjs";

console.log(sum(2, 6));
```

Output:

```
root@123# node index.mjs
(node:4278) ExperimentalWarning: The ESM module loader is experimental.
8
```

1.5 Exporting a function after defining

We can export a function after defining.

Example:

//sample.mjs

```
function sum(num1, num2) {
  return num1 + num2;
}
export default sum;
```

//index.mjs

```
import sum from "./sample.mjs";
```

```
console.log(sum(2, 6));
```

Output:

```
root@123# node index.mjs
(node:4462) ExperimentalWarning: The ESM module loader is experimental.
8
```

1.6 Exporting a class while defining

We can export a class while defining.

Example:

//sample.mjs

```
export default class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

//index.mjs

```
import StudentDetails from "./sample.mjs";

const newStudentDetails = new StudentDetails("Ram", 15);
console.log(newStudentDetails);
console.log(newStudentDetails.name);
```

Output:

```
root@123# node index.mjs
(node:1035) ExperimentalWarning: The ESM module loader is experimental.
StudentDetails {name: "Ram", age: 15}
Ram
```

1.7 Exporting a class after defining

We can export a class after defining.

Example:

//sample.mjs

```
class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
export default StudentDetails;
```

//index.mjs

```
import StudentDetails from "./sample.mjs";

const newStudentDetails = new StudentDetails("Ram", 15);
console.log(newStudentDetails);
console.log(newStudentDetails.name);
```

Output:

```
root@123# node index.mjs
(node:1575) ExperimentalWarning: The ESM module loader is experimental.
StudentDetails {name: "Ram", age: 15}
Ram
```

2. Named Exports

2.1 Exporting multiple variables while defining

We can export boolean, number, string, null, undefined, objects, and arrays while defining.

Example:

//sample.mjs

```
export let value = 5;
export let studentName = "Rahul";
```

//index.mjs

```
import { value, studentName } from "./sample.mjs";
```

```
console.log(value);

console.log(studentName);
```

Output:

```
root@123# node index.mjs
(node:1770) ExperimentalWarning: The ESM module loader is experimental.
5
Rahul
```

2.2 Exporting multiple variables after defining

We can export multiple variables after defining in an Object format.

Example:

//sample.mjs

```
let value = 5;
const studentName = "Rahul";

export { value, studentName };
```

//index.mjs

```
import { value, studentName } from "./sample.mjs";

console.log(value);
console.log(studentName);
```

Output:

```
root@123# node index.mjs
(node:2437) ExperimentalWarning: The ESM module loader is experimental.
5
Rahul
```

2.3 Exporting multiple functions while defining

We can export multiple functions while defining.

Example:

//sample.mjs

```
export function sum(num1, num2) {
  return num1 + num2;
}

export function sub(num1, num2) {
  return num1 - num2;
}
```

//index.mjs

```
import { sum, sub } from "./sample.mjs";

console.log(sum(4, 2));

console.log(sub(4, 2));
```

Output:

```
root@123# node index.mjs
(node:2954) ExperimentalWarning: The ESM module loader is experimental.
6
2
```

2.4 Exporting multiple functions after defining

We can export multiple functions after defining.

Example:

//sample.mjs

```
function sum(num1, num2) {
  return num1 + num2;
}

function sub(num1, num2) {
  return num1 - num2;
}

export { sum, sub };
```

//index.mjs

```
import { sum, sub } from "./sample.mjs";
```

```
console.log(sum(4, 2));

console.log(sub(4, 2));
```

Output:

```
root@123# node index.mjs
(node:3276) ExperimentalWarning: The ESM module loader is experimental.
6
2
```

2.5 Exporting multiple classes while defining

We can export multiple classes while defining.

Example:

//sample.mjs

```
export class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

export class CarDetails {
  constructor(name, speed) {
    this.name = name;
    this.speed = speed;
  }
}
```

//index.mjs

```
import { StudentDetails, CarDetails } from "./sample.mjs";

const newStudentDetails = new StudentDetails("Ram", 15);
console.log(newStudentDetails);
console.log(newStudentDetails.name);

const newCarDetails = new CarDetails("Alto", "60kmph");
```

```
console.log(newCarDetails);
console.log(newCarDetails.name);
```

Output:

```
root@123# node index.mjs
(node:3517) ExperimentalWarning: The ESM module loader is experimental.
StudentDetails { name: 'Ram', age: 15 }
Ram
CarDetails { name: 'Alto', speed: '60kmph' }
Alto
```

2.6 Exporting multiple classes after defining

We can export multiple classes after defining.

Example:

//sample.mjs

```
class StudentDetails {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

class CarDetails {
  constructor(name, speed) {
    this.name = name;
    this.speed = speed;
  }
}

export { StudentDetails, CarDetails };
```

//index.mjs

```
import { StudentDetails, CarDetails } from "./sample.mjs";
```

```
const newStudentDetails = new StudentDetails("Ram", 15);
console.log(newStudentDetails);
console.log(newStudentDetails.name);

const newCarDetails = new CarDetails("Alto", "60kmph");
console.log(newCarDetails);
console.log(newCarDetails.name);
```

Output:

```
root@123# node index.mjs
(node:3841) ExperimentalWarning: The ESM module loader is experimental.
StudentDetails {name: "Ram", age: 15}
Ram
CarDetails {name: "Alto", speed: "60kmph"}
Alto
```

Introduction to Node JS | Part 2 | Cheat Sheet:

Concepts in Focus

- Core Modules
    - Path
- Package
    - Node Package Manager (NPM)
- Steps to create a Node JS Project
- Third-Party Packages
    - date-fns

1. Core Modules

The Core Modules are inbuilt in Node JS.

Some of the most commonly used are:

Module	Description
path	Handles file paths
fs	Handles the file system
url	Parses the URL strings

1.1 Path

The path module provides utilities for working with file and directory paths. It can be accessed using:

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

Example:

//index.js

```
const path = require("path");
const filePath = path.join("users", "ravi", "notes.txt");

console.log(filePath);
```

Output:

```
root@123# node index.js
users/ravi/notes.txt
```

Note:

Many developers prefer Common JS Modules to ES6 syntax as ES6 syntax is in the experimental phase.

2. Package

A package is a directory with one or more modules grouped.

2.1 Node Package Manager (NPM)

NPM is the package manager for the Node JS packages with more than one million packages.

It provides a command line tool that allows you to publish, discover, install, and develop node programs.

2.1.1 CLI

NPM CLI sets up the Node JS Project to organize various modules and work with third-party packages.

Command	Description
npm init -y	Initializes a project and creates a

Command	Description
	package.json file
npm install <package-name> --save	Installs the third-party packages

3. Steps to create a Node JS Project

Run the below commands in the terminal.

1. Create a new directory/folder.

mkdir myapp

2. Move into the created folder.

cd myapp

3. Initialize the project.

npm init -y

4. Third-Party Packages

The Third-Party Packages are the external Node JS Packages.

They are developed by Node JS developers and are made available through the Node ecosystem.

4.1 date-fns

It is a third-party package for manipulating JavaScript dates in a browser & Node.js.

Installation Command:

npm install date-fns –save

4.1.1 addDays

It adds the specified number of days to the given date.

Example:

//index.js

```
const addDays = require("date-fns/addDays");
```

```
const result = addDays(new Date(2021, 0, 11), 10);

console.log(result);
```

Output:

```
1.root@123# node index.js
2.2021-01-21T00:00:00.000Z
```

Note:

While creating the Date() object, we have to provide the month index from (0-11), whereas we will get the output considering Jan=1 and Dec=12.

Introduction to Express JS | Cheat Sheet

Concepts in Focus:

1. HTTP Server

- Works with the HTTP requests and responses
- Handles the different paths
- Handles the query parameters
- Sends the content as HTML, CSS, etc. as an HTTP response
- Works with the databases

1.1 Server-side Web Frameworks

The Server-side Web Frameworks take care of all the above requirements.

Some of the Web Frameworks are:

- **Express (Node JS)**
- Django (Python)

- Ruby on Rails (Ruby)
- Spring Boot (Java)

2. Express JS

It is a free and open-source Server-side Web Application Framework for Node JS.

It provides a robust set of features to build web and mobile applications quickly and easily.

Installation Command:

npm install express –save

3. Network call using Express JS

1. **Creating Express server instance**

   const express = require("express");

   const app = express();

2. Assigning a port number

app.listen(3000);

3.1 Handling HTTP Request

Syntax:

app.METHOD(PATH, HANDLER)

- **METHOD** is an HTTP request method, in lowercase like get , post , put , and delete.
- **PATH** is a path on the server.
- **HANDLER** is the function executed when the PATH is matched with the requested path.

3.1.1 GET Request

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

app.get("/", (request, response) => {
  response.send("Hello World!");
```

```
});
app.listen(3000);
```

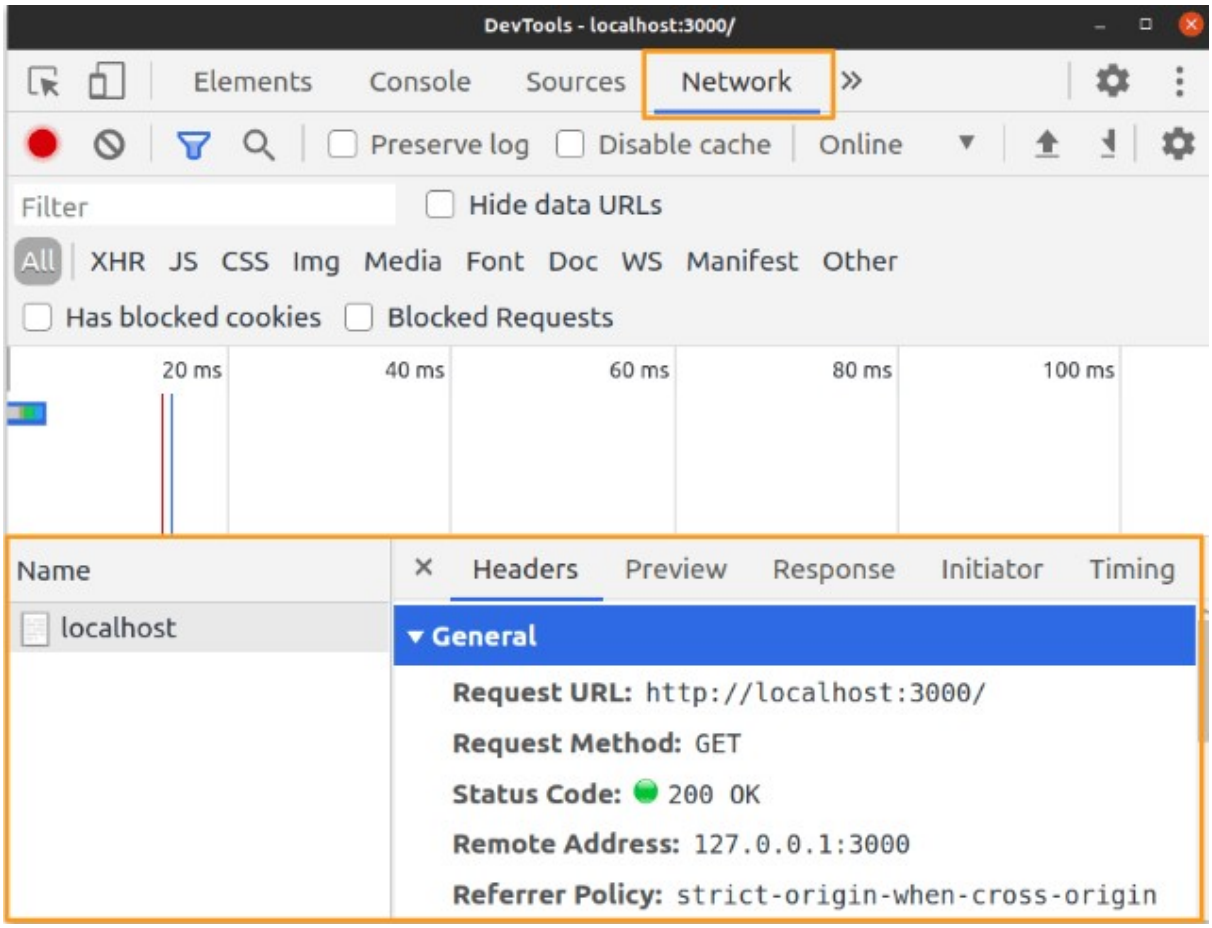
Note:

Whenever the code changes, we need to restart the server to reflect the changes we made.

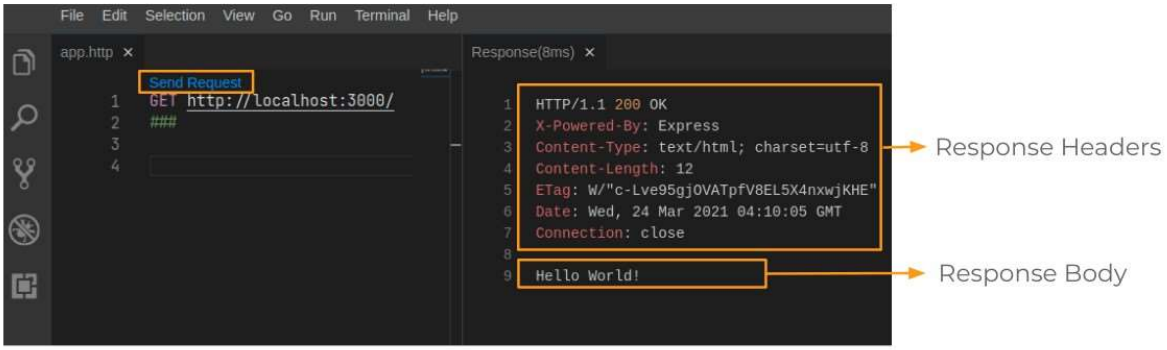
4. Testing Network Calls

We can test the Network calls in two ways.

1. Browser Network Tab.

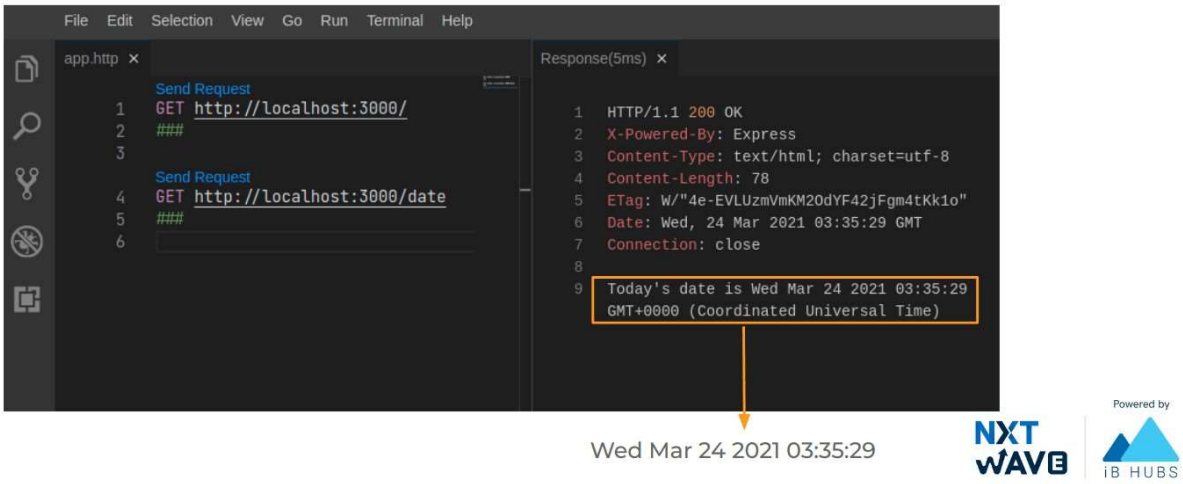


1. Clicking the Send Request in the app.http file.



5. Network Call to get a Today's Date

```
const express = require("express");
const app = express();
app.get("/date", (request, response) => {
  let date = new Date();
  response.send(`Today's date is ${date}`);
});
app.listen(3000);
```



6. Network Call to get HTML content as an HTTP Response

6.1 Sending file as an HTTP Response

Syntax:

```
response.sendFile(PATH, {root: __dirname });
```

- **PATH** is a path to the file which we want to send.
- **__dirname** is a variable in Common JS Modules that returns the path of the folder where the current JavaScript file is present.

```
const express = require("express");
const app = express();
app.get("/page", (request, response) => {
  response.sendFile("./page.html", { root: __dirname });
});
app.listen(3000);
```

Introduction to Express JS | Part 2

Concepts in Focus

1. Application Programming Interface (API)

An API is a software intermediary that allows two applications to talk to each other.

For example, OLA, and UBER use Google Maps API to provide their services.

All the Network calls that we added are also the APIs.

2. Database

Express apps can use any database supported by Node JS.

There are many popular options, including **SQLite**, PostgreSQL, MySQL, Redis, and MongoDB.

3. SQLite

The SQLite provides a command-line tool **sqlite3**.

It allows the user to enter and execute SQL statements against an SQLite database.

3.1 SQLite CLI

3.1.1 Listing Existing Tables

The
.tables
command is used to get the list of tables available in the SQLite database.

3.1.2 Selecting Table Data

Syntax:
SELECT * from <table>

4. SQLite Methods

4.1 Open

The SQLite
open()
method is used to connect the database server and provides a connection object to operate on the database.

Syntax:

```
open({
  filename: DATABASE_PATH,
  driver: SQLITE_DATABASE_DRIVER,
});
```

It returns a promise object. On resolving the promise object, we will get the database connection object.

4.2 Executing SQL Queries

SQLite package provides multiple methods to execute SQL queries on a database.

Some of them are:

- **all()**
- get()
- run()
- exec(), etc.

4.2.1 all()

db.all(SQL_QUERY);

The
all()
method is used to get multiple rows of data.

5. SQL Third-party packages

We can use
sqlite
and

sqlite3
node packages to connect SQLite Database from Node JS.

Installation Commands

npm install sqlite --save
npm install sqlite3 –save

6. Connecting SQLite Database from Node JS to get the books from Goodreads Website

1. Install the SQL third-party packages sqlite and sqlite3 .
2. Initialize the SQLite Database

6.1 SQLite Database Initialization

```
const express = require("express");
const path = require("path");
const { open } = require("sqlite");
const sqlite3 = require("sqlite3");
const app = express();
const dbPath = path.join(__dirname, "goodreads.db");
let db = null;
const initializeDBAndServer = async () => {
  try {
    db = await open({
      filename: dbPath,
      driver: sqlite3.Database,
    });
    app.listen(3000, () => {
      console.log("Server Running at http://localhost:3000/");
    });
  } catch (e) {
    console.log(`DB Error: ${e.message}`);
    process.exit(1);
  }
};
initializeDBAndServer();
```

6.2 Goodreads Get Books API

```
app.get("/books/", async (request, response) => {
  const getBooksQuery = `
    SELECT
      *
    FROM
      book
    ORDER BY
      book_id;`;
  const booksArray = await db.all(getBooksQuery);
  response.send(booksArray);
});
```

Introduction to Express JS | Part 3

Concepts in Focus

- SQLite Methods
  - get()
  - run()
- Node JS Third-party packages
  - Nodemon
- GoodReads API
  - Get Book
  - Add Book
  - Update Book
  - Delete Book
  - Get Author Books

1. SQLite Methods

The SQLite package provides multiple methods to execute SQL queries on a database.

Some of them are:

- all()
- **get()**
- **run()**
- exec(), etc.

1.1 get()

The

get()
method is used to get a single row from the table.

Syntax:

db.get(SQL_QUERY);

1.2 run()

The
run()
method is used to create or update table data.

Syntax:

db.run(SQL_QUERY);

2. Node JS Third-party packages

2.1 Nodemon

The Nodemon is a tool that **restarts the server automatically** whenever we make changes in the file.

Installation Command:

npm install -g nodemon

Note:
The -g indicates that the nodemon will be installed globally in the environment. While executing the file, replace the node with the nodemon. For example, nodemon index.js.

3. GoodReads APIs

- Get Book
- Add Book
- Update Book
- Delete Book
- Get Author Books

3.1 Get Book

We can use
/books/:bookId/
as a path to identify a single book resource, where
bookId

is a path parameter.

For example,

In http://localhost:3000/books/1/ , the bookId is 1.

```
app.get("/books/:bookId/", async (request, response) => {
  const { bookId } = request.params;
  const getBookQuery = `
    SELECT
      *
    FROM
      book
    WHERE
      book_id = ${bookId};`;
  const book = await db.get(getBookQuery);
  response.send(book);
});
```

Note: Any String can be used a Path Parameter.

3.2 Add Book

To add a book to the Database, you need to send a request body in JSON format.

- The express.json() is used to recognize the incoming request object as JSON Object and parses it.
- The request.body is used to get the HTTP Request body.

```
app.post("/books/", async (request, response) => {
  const bookDetails = request.body;
  const {
    title,
    authorId,
    rating,
    ratingCount,
    reviewCount,
    description,
    pages,
    dateOfPublication,
    editionLanguage,
    price,
```

```
    onlineStores,
  } = bookDetails;
  const addBookQuery = `
    INSERT INTO
      book
(title,author_id,rating,rating_count,review_count,description,pages,date_of_publicati
on,edition_language,price,online_stores)
    VALUES
      (
        '${title}',
         ${authorId},
         ${rating},
         ${ratingCount},
         ${reviewCount},
        '${description}',
         ${pages},
        '${dateOfPublication}',
        '${editionLanguage}',
         ${price},
        '${onlineStores}'
      );`;

  const dbResponse = await db.run(addBookQuery);
  const bookId = dbResponse.lastID;
  response.send({ bookId: bookId });
});
```

The
dbResponse.lastID
provides the primary key of the new row inserted.

3.3 Update Book

We can use books/:bookId/ as a path to identify a single book resource,
where :bookId is the path parameter.
For example,
http://localhost:3000/books/1/.

```
app.put("/books/:bookId/", async (request, response) => {
  const { bookId } = request.params;
```

```
  const bookDetails = request.body;
  const {
    title,
    authorId,
    rating,
    ratingCount,
    reviewCount,
    description,
    pages,
    dateOfPublication,
    editionLanguage,
    price,
    onlineStores,
  } = bookDetails;
  const updateBookQuery = `
    UPDATE
      book
    SET
      title='${title}',
      author_id=${authorId},
      rating=${rating},
      rating_count=${ratingCount},
      review_count=${reviewCount},
      description='${description}',
      pages=${pages},
      date_of_publication='${dateOfPublication}',
      edition_language='${editionLanguage}',
      price=${price},
      online_stores='${onlineStores}'
    WHERE
      book_id = ${bookId};`;
  await db.run(updateBookQuery);
  response.send("Book Updated Successfully");
});
```

The request.params provides the parameters passed through the request.

Note: The strings sent through the APIs must be wrapped in quotes.

3.4 Delete Book

```
app.delete("/books/:bookId/", async (request, response) => {
  const { bookId } = request.params;
  const deleteBookQuery = `
    DELETE FROM
      book
    WHERE
      book_id = ${bookId};`;
  await db.run(deleteBookQuery);
  response.send("Book Deleted Successfully");
});
```

3.5 Get Author Books

```
app.get("/authors/:authorId/books/", async (request, response) => {
  const { authorId } = request.params;
  const getAuthorBooksQuery = `
    SELECT
     *
    FROM
     book
    WHERE
      author_id = ${authorId};`;
  const booksArray = await db.all(getAuthorBooksQuery);
  response.send(booksArray);
});
```

REST APIs

Concepts in Focus

- Get Books API
    - Filtering Books
- REST APIs
    - Why Rest Principles?
    - REST API Principles

1. Get Books API

Let's see how to add **Filters** to **Get Books API**

1.1 Filtering Books

- Get a specific number of books
- Get books based on search query text
- Get books in the sorted order

1.1.1 Get a specific number of books

To get specific number of books in certain range we use **limit** and **offset**.

Offset is used to specify the **position** from where rows are to be selected.

Limit is used to specify the **number of rows**. and many more... Query parameters starts with **?** (question mark) followed by key value pairs separated by **&** (ampersand)

Example :

http://localhost:3000/books/?limit=2

http://localhost:3000/books/?offset=2&limit=3

http://localhost:3000/authors/20/books/?offset=2

Note :

The query parameters are used to sort/filter resources.
The path parameters are used to identify a specific resource(s)

1.1.2 Get books based on search query text

We provide query text to **search_q** key

search_q = potter

1.1.3 Get books in the sorted order

We provide sorted order to **order** key

Ascending: **ASC** DESCENDING: **DESC**

order = ASC

order = DESC

Filtering GET Books API

app.get("/books/", async (request, response) => {

```
  const {
    offset = 2,
    limit = 5,
    order = "ASC",
    order_by = "book_id",
    search_q = "",
  } = request.query;
  const getBooksQuery = `
    SELECT
      *
    FROM
     book
    WHERE
     title LIKE '%${search_q}%'
    ORDER BY ${order_by} ${order}
    LIMIT ${limit} OFFSET ${offset};`;
  const booksArray = await db.all(getBooksQuery);
  response.send(booksArray);
});
```

Note:

We can skip or add slash while appending query parameters to the URL

http://localhost:3000/books/?offset=2&limit=3
is same as
http://localhost:3000/books?offset=2&limit=3

2. REST APIs

REST: Representational State Transfer

REST is a **set of principles** that define how Web standards, such as HTTP and URLs, are supposed to be used.

2.1 Why Rest Principles?

Using Rest Principles improves application in various aspects like **scalability**, **reliability** etc

2.2 REST API Principles

- Providing unique ID to each resource
- Using standard methods like GET, POST, PUT, and DELETE
- Accept and Respond with JSON

  and many more...

Debugging Common Errors

Concepts in Focus

- Importing Unknown Modules
- Starting Server in Multiple Terminals
- Starting Server outside the myapp
- Accessing Wrong URL
- Missing Function Call
- Importing Unknown File

1. Importing Unknown Modules

When we try to import unknown modules

root@123:/.../myapp# nodemon index.js

[nodemon] 2.0.7

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): *.*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node index.js`

internal/modules/cjs/loader.js:968

  throw err;

  ^

Error: Cannot find module 'expresses'

Require stack:

- /home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js

    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:965:15)

    at Function.Module._load (internal/modules/cjs/loader.js:841:27)

    at Module.require (internal/modules/cjs/loader.js:1025:19)

    at require (internal/modules/cjs/helpers.js:72:18)

    at Object.<anonymous> (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js:1:17)

    at Module._compile (internal/modules/cjs/loader.js:1137:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1157:10)

    at Module.load (internal/modules/cjs/loader.js:985:32)

    at Function.Module._load (internal/modules/cjs/loader.js:878:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:71:12) {

  code: 'MODULE_NOT_FOUND',

  requireStack: [

    '/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js'

  ]

}

[nodemon] app crashed - waiting for file changes before starting...

2. Starting Server in Multiple Terminals

When we try to start the server in multiple terminals

root@123:/.../myapp# nodemon index.js

[nodemon] 2.0.7

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): *.*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node index.js`

events.js:292

throw er; // Unhandled 'error' event

^

Error: listen EADDRINUSE: address already in use :::3000

at Server.setupListenHandle [as _listen2] (net.js:1313:16)

at listenInCluster (net.js:1361:12)

at Server.listen (net.js:1447:7)

at Function.listen (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/application.js:618:24)

at initializeDBAndServer (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js:16:9)

Emitted 'error' event on Server instance at:

at emitErrorNT (net.js:1340:8)

at processTicksAndRejections (internal/process/task_queues.js:84:21) {

code: 'EADDRINUSE',

errno: 'EADDRINUSE',

syscall: 'listen',

address: '::',

port: 3000

}

[nodemon] app crashed - waiting for file changes before starting...

When you get **address already in use :::3000** error

Step 1: Kill the currently running process in the terminal with Ctrl + C

Step 2: Run lsof -i :port_number in your CCBP IDE Terminal

Step 3: Run kill -9 process_id in your CCBP IDE Terminal

Example :

```
root@123:/.../...part-3/myapp# nodemon index.js
[nodemon] starting `node index.js`
events.js:292
      throw er; // Unhandled 'error' event
      ^
Error: listen EADDRINUSE: address already in use :::3000
.
.
.
^C

root@123:/.../...part-3# lsof -i :3000

COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
node    9841 root   20u  IPv6 345981      0t0  TCP *:3000 (LISTEN)

root@123:/.../...part-3# kill -9 9841
root@123:/.../...part-3#
```

3. Starting Server outside the myapp

When we try to start the server outside myapp using node

```
root@123:/.../...part-3# node index.js
internal/modules/cjs/loader.js:968
  throw err;
  ^
Error: Cannot find module '/.../...Part-3/index.js'
```

root@123:/.../...part-3#

When we try to start the server outside myapp using nodemon

root@123:/.../...part-3# nodemon index.js

Usage: nodemon [nodemon options] [script.js] [args]

See "nodemon --help" for more.

root@123:/.../...part-3#

4. Accessing Wrong URL

When we try to Send Request for wrong URLs

HTTP/1.1 404 Not Found

X-Powered-By: Express

Content-Security-Policy: default-src 'none'

X-Content-Type-Options: nosniff

Content-Type: text/html; charset=utf-8

Content-Length: 148

Date: Sun, 04 Apr 2021 11:50:56 GMT

Connection: close

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Error</title>

</head>

<body>

<pre>Cannot GET /bookss/1/</pre>

</body>

</html

5. Missing Function Call

When we miss calling the function express and trying to access methods returned from the function express

root@123:/.../...part-3/myapp# nodemon index.js

[nodemon] 2.0.7

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): *.*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node index.js`

/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js:26

app.get("/books/", async (request, response) => {

^

TypeError: app.get is not a function

at Object.<anonymous> (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js:26:5)

at Module._compile (internal/modules/cjs/loader.js:1137:30)

at Object.Module._extensions..js (internal/modules/cjs/loader.js:1157:10)

at Module.load (internal/modules/cjs/loader.js:985:32)

at Function.Module._load (internal/modules/cjs/loader.js:878:14)

at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:71:12)

at internal/main/run_main_module.js:17:47

[nodemon] app crashed - waiting for file changes before starting...

6. Importing Unknown File

When we try to import an unknown file

root@123:/.../...part-3/myapp# nodemon index.js

[nodemon] 2.0.7

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): *.*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node index.js`

Server Running at http://localhost:3000/

(node:668) UnhandledPromiseRejectionWarning: Error: SQLITE_ERROR: no such table: book

(node:668) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 1)

(node:668) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

Debugging Common Errors | Part 2

Concepts in Focus

- Request Methods
- Missing colon(:)
- Accessing Path Parameters
- Query Formatting
- Replacing SQLite Methods
- HTTP Request URL
    - Missing '?'
    - Missing ‘&’ between Query Parameters
    - Replacing '&' with ',' in Query Parameters
- Accessing Unknown Database
- Accessing Wrong Port Number

1. Request Methods

When we try to request with a wrong method

HTTP/1.1 404 Not Found

X-Powered-By: Express

Content-Security-Policy: default-src 'none'

X-Content-Type-Options: nosniff

Content-Type: text/html; charset=utf-8

Content-Length: 148

Date: Sun, 04 Apr 2021 11:50:56 GMT

Connection: close

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Error</title>

</head>

<body>

<pre>Cannot POST /books/</pre>

</body>

</html

2. Missing colon(:)

When we miss semicolon while setting Path Parameters

HTTP/1.1 404 Not Found

X-Powered-By: Express

Content-Security-Policy: default-src 'none'

X-Content-Type-Options: nosniff

Content-Type: text/html; charset=utf-8

Content-Length: 148

Date: Sun, 04 Apr 2021 11:50:56 GMT

Connection: close

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Error</title>

</head>

<body>

<pre>Cannot GET /books/</pre>

</body>

</html>

3. Accessing Path Parameters

When we are trying to access path parameters with the wrong name

root@123:/.../...part-3/myapp# nodemon index.js

[nodemon] 2.0.7

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): *.*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node index.js`

Server Running at http://localhost:3000/

(node:905) UnhandledPromiseRejectionWarning: ReferenceError: bookId is not defined

at /home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/index.js:46:19

at Layer.handle [as handle_request] (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/layer.js:95:5)

at next (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/route.js:137:13)

at Route.dispatch (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/route.js:112:3)

at Layer.handle [as handle_request] (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/layer.js:95:5)

at /home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/index.js:281:22

at param (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/index.js:354:14)

at param (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/index.js:365:14)

at Function.process_params (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/index.js:410:3)

at next (/home/workspace/nodejs/sessions/Introduction-to-Express-JS-Part-2/myapp/node_modules/.pnpm/express@4.17.1/node_modules/express/lib/router/index.js:275:10)

(node:905) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 1)

(node:905) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

4. Query Formatting

When we miss embedding expression into string literal properly in the query

root@123:/.../...part-3/myapp# nodemon index.js

[nodemon] 2.0.7

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): *.*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node index.js`

Server Running at http://localhost:3000/

(node:1023) UnhandledPromiseRejectionWarning: Error: SQLITE_ERROR: no such table: book

(node:1023) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 1)

(node:1023) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

5. Replacing SQLite Methods

When we use one SQLite method instead of another SQLite method we may get an unexpected response

HTTP/1.1 200 OK

X-Powered-By: Express

Content-Type: application/json; charset=utf-8

Content-Length: 34

ETag: W/"22-fiAy4921oZAEOAA97wnoLQyL2Dw"

Date: Sun, 04 Apr 2021 12:46:07 GMT

Connection: close

{

"stmt":{},

"lastID":0,

"changes":0

}

6. HTTP Request URL

6.1 Missing ?

When we miss ? before starting query parameters

```
HTTP/1.1 404 Not Found
X-Powered-By: Express
Content-Security-Policy: default-src 'none'
X-Content-Type-Options: nosniff
Content-Type: text/html; charset=utf-8
Content-Length: 148
Date: Sun, 04 Apr 2021 11:50:56 GMT
Connection: close

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot GET
http://localhost:3000/books/offset=2&limit=3&search_q=potter&order_by=price&or
der=DESC</pre>
</body>
</html>
```

6.2 Missing & between Query Parameters

When we miss & between query parameters

```
root@123:/.../...part-3/myapp# nodemon index.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Server Running at http://localhost:3000/
string
```

(node:1408) UnhandledPromiseRejectionWarning: Error: SQLITE_ERROR: unrecognized token: "3offset"
(node:1408) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 1)
(node:1408) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

6.3 Replacing & with , in Query Parameters

When we replace & with , in Query Parameters

root@123:/.../...part-3/myapp# nodemon index.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Server Running at http://localhost:3000/
string
(node:1452) UnhandledPromiseRejectionWarning: Error: SQLITE_ERROR: near ",": syntax error
(node:1452) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 1)
(node:1452) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.

7. Accessing Unknown Database

When we are trying to access an unknown Database it will not show any error instead it creates a new database with the given name

root@123:/.../myapp# sqlite3 goodread.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite>.tables
sqlite>

8. Accessing Wrong Port Number

When we are trying to access the wrong port number, we may get the below notification

The connection was rejected. Either the requested service isn't running on the requested server/port, the proxy settings in vscode are misconfigured, or a firewall is blocking requests. Details: RequestError: connect ECONNREFUSED 127.0.0.1:4000.

Authentication:

Concepts in Focus

- Installing Third-party package bcrypt
- Goodreads APIs for Specified Users
    - Register User API
    - Login User API

1. Installing Third-party package bcrypt

Storing the passwords in plain text within a database is not a good idea since they can be misused, So Passwords should be encrypted

bcrypt
package provides **functions** to perform operations like **encryption**, **comparison**, etc

- **bcrypt.hash()** uses various processes and encrypts the given password and makes it unpredictable
- **bcrypt.compare()** function compares the password entered by the user and hash against each other

Installation Command

root@123:~/myapp# npm install bcrypt --save

2. Goodreads APIs for Specified Users

We need to maintain the list of users in a table and provide access only to that specified users

User needs to be registered and then log in to access the books

- Register User API
- Login User API

2.1 Register User API

Here we check whether the user is a new user or an existing user. Returns "User Already exists" for existing user else for a new user we store encrypted password in the DB

```
app.post("/users/", async (request, response) => {
  const { username, name, password, gender, location } = request.body;
  const hashedPassword = await bcrypt.hash(request.body.password, 10);
  const selectUserQuery = `SELECT * FROM user WHERE username = '${username}'`;
  const dbUser = await db.get(selectUserQuery);
  if (dbUser === undefined) {
    const createUserQuery = `
      INSERT INTO
        user (username, name, password, gender, location)
      VALUES
        (
          '${username}',
          '${name}',
          '${hashedPassword}',
          '${gender}',
          '${location}'
        )`;
```

```
    const dbResponse = await db.run(createUserQuery);
    const newUserId = dbResponse.lastID;
    response.send(`Created new user with ${newUserId}`);
  } else {
    response.status = 400;
    response.send("User already exists");
  }
});
```

2.2 Login User API

Here we check whether the user exists in DB or not. Returns "Invalid User" if the user doesn't exist else we compare the given password with the DB user password

```
app.post("/login", async (request, response) => {
  const { username, password } = request.body;
  const selectUserQuery = `SELECT * FROM user WHERE username = '${username}'`;
  const dbUser = await db.get(selectUserQuery);
  if (dbUser === undefined) {
    response.status(400);
    response.send("Invalid User");
  } else {
    const isPasswordMatched = await bcrypt.compare(password, dbUser.password);
    if (isPasswordMatched === true) {
      response.send("Login Success!");
    } else {
      response.status(400);
      response.send("Invalid Password");
    }
  }
});
```

Status Codes:

Status Codes	Status Text ID
200	OK
204	No Response
301	Moved Permanently
400	Bad Request
403	Forbidden
401	Unauthorized

Authentication Part 2:

Authentication Mechanisms

- Token Authentication mechanism
  - Access Token
  - How Token Authentication works?
- JWT
  - How JWT works?
  - JWT Package
- Login User API by generating the JWT Token
- How to pass JWT Token?
- Get Books API with Token Authentication

1. Authentication Mechanisms

To check whether the user is logged in or not we use different Authentication mechanisms

Commonly used Authentication mechanisms:

- Token Authentication
- Session Authentication

2. Token Authentication mechanism

We use the Access Token to verify whether the user is logged in or not

2.1 Access Token

Access Token is a set of characters which are used to identify a user

Example:

It is used to **verify** whether a user is Valid/Invalid

2.2 How Token Authentication works?

- Server generates token and certifies the client
- Client uses this token on every subsequent request
- Client don’t need to provide entire details every time

3. JWT

JSON Web Token is a standard used to create access tokens for an application This access token can also be called as JWT Token

3.1 How JWT works?

Client: Login with username and password **Server:** Returns a JWT Token **Client:** Sends JWT Token while requesting **Server:** Sends Response to the client

3.2 JWT Package

Jsonwebtoken package provides jwt.sign and jwt.verify functions

- jwt.sign() function takes payload, secret key, options as arguments and generates JWTToken out of it
- jwt.verify() verifies jwtToken and if it’s valid, returns payload. Else, it throws an error

root@123root@123:.../myapp# npm install jsonwebtoken

4. Login User API by generating the JWT Token

When the user tries to log in, verify the Password. Returns **JWT Token** if the password matches else return **Invalid Password** with status code **400**.

```
app.post("/login", async (request, response) => {
  const { username, password } = request.body;
  const selectUserQuery = `SELECT * FROM user WHERE username = '$
{username}'`;
  const dbUser = await db.get(selectUserQuery);
  if (dbUser === undefined) {
    response.status(400);
    response.send("Invalid User");
  } else {
    const isPasswordMatched = await bcrypt.compare(password, dbUser.password);
    if (isPasswordMatched === true) {
      const payload = {
        username: username,
```

```
      };
      const jwtToken = jwt.sign(payload, "MY_SECRET_TOKEN");
      response.send({ jwtToken });
    } else {
      response.status(400);
      response.send("Invalid Password");
    }
  }
});
```

5. How to pass JWT Token?

We have to add an authorization header to our request and the JWT Token is passed as a **Bearer token**

GET http://localhost:3000/books?
offset=2&limit=3&search_q=the&order_by=price&order=DESC
Authorization: bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoicmFodWwiLCJnZW5kZXIiOiJNYWxlIiwibG9jYXRpb24iOiJoeWRlcmFiYWQiLCJpYXQiOjE2MTc0MzI0MDd9.Eqevw5QE70ZAVrmOZUc6pflUbeI0ffZUmQLDHYplU8g

6. Get Books API with Token Authentication

Here we check for JWT Token from Headers. If JWT Token is not present it returns an **Invalid Access Token** with status code **401** else **verify** the **JWT Token**.

```
app.get("/books/", (request, response) => {
  let jwtToken;
  const authHeader = request.headers["authorization"];
  if (authHeader !== undefined) {
    jwtToken = authHeader.split(" ")[1];
  }
  if (jwtToken === undefined) {
    response.status(401);
    response.send("Invalid Access Token");
  } else {
```

```
  jwt.verify(jwtToken, "MY_SECRET_TOKEN", async (error, payload) => {
    if (error) {
      response.send("Invalid Access Token");
    } else {
      const getBooksQuery = `
          SELECT
            *
          FROM
           book
          ORDER BY
            book_id;`;
      const booksArray = await db.all(getBooksQuery);
      response.send(booksArray);
    }
  });
 }
});
```

Authentication Part 3:

Middleware functions

- Multiple Middleware functions
- Logger Middleware Implementation
  - Defining a Middleware Function
  - Logger Middleware Function
  - Get Books API with Logger Middleware
- Authenticate Token Middleware
- Get Books API with Authenticate Token Middleware
- Passing data from Authenticate Token Middleware
- Get User Profile API with Authenticate Token Middleware

1. Middleware functions

Middleware is a special kind of function in Express JS which accepts the request from

- the user (or)
- the previous middleware

After processing the request the middleware function

- sends the response to another middleware (or)
- calls the API Handler (or)
- sends response to the user

app.method(Path, middleware1, handler);

Example

```
const jsonMiddleware = express.json();
app.use(jsonMiddleware);
```

It is a **built-in middleware function** it recognizes the incoming request object as a JSON object, parses it, and then calls handler in **every API call**

1.1 Multiple Middleware functions

We can pass multiple middleware functions

app.method(Path, middleware1, middleware2, handler);

2. Logger Middleware Implementation

2.1 Defining a Middleware Function

```
const middlewareFunction = (request, response, next) => {};
```

2.2 Logger Middleware Function

```
const logger = (request, response, next) => {
  console.log(request.query);
  next();
};
```

The **next** parameter is a function passed by Express JS which, when **invoked**, executes the **next succeeding function**

2.3 Get Books API with Logger Middleware

```
app.get("/books/", logger, async (request, response) => {
  const getBooksQuery = `
   SELECT
    *
   FROM
    book
   ORDER BY
    book_id;`;
  const booksArray = await db.all(getBooksQuery);
  response.send(booksArray);
});
```

3. Authenticate Token Middleware

In Authenticate Token Middleware we will verify the JWT Token

```
const authenticateToken = (request, response, next) => {
  let jwtToken;
  const authHeader = request.headers["authorization"];
  if (authHeader !== undefined) {
    jwtToken = authHeader.split(" ")[1];
  }
  if (jwtToken === undefined) {
    response.status(401);
    response.send("Invalid JWT Token");
  } else {
    jwt.verify(jwtToken, "MY_SECRET_TOKEN", async (error, payload) => {
      if (error) {
        response.status(401);
        response.send("Invalid JWT Token");
```

```
    } else {
      next();
    }
   });
 }
};
```

4. Get Books API with Authenticate Token Middleware

Let's Pass Authenticate Token Middleware to Get Books API

```
app.get("/books/", authenticateToken, async (request, response) => {
 const getBooksQuery = `
  SELECT
   *
  FROM
   book
  ORDER BY
   book_id;`;
 const booksArray = await db.all(getBooksQuery);
 response.send(booksArray);
});
```

5. Passing data from Authenticate Token Middleware

We **cannot directly pass** data to the next handler, but we can send data through the **request** object

```
const authenticateToken = (request, response, next) => {
 let jwtToken;
 const authHeader = request.headers["authorization"];
 if (authHeader !== undefined) {
```

```
  jwtToken = authHeader.split(" ")[1];
 }
 if (jwtToken === undefined) {
  response.status(401);
  response.send("Invalid JWT Token");
 } else {
  jwt.verify(jwtToken, "MY_SECRET_TOKEN", async (error, payload) => {
   if (error) {
    response.status(401);
    response.send("Invalid JWT Token");
   } else {
    request.username = payload.username;
    next();
   }
  });
 }
};
```

6. Get User Profile API with Authenticate Token Middleware

We can access request variable from the request object

```
app.get("/profile/", authenticateToken, async (request, response) => {
 let { username } = request;
 const selectUserQuery = `SELECT * FROM user WHERE username = '${username}'`;
 const userDetails = await db.get(selectUserQuery);
 response.send(userDetails);
});
```