Syllabus:

Introduction to Middleware: Route, Express Route Methods - GET, POST, PUT, DELETE, Route Paths-strings, string patterns or regular expressions, Route parameters.

Route Handlers - as a form of a function, an array of functions, or combinations of both.

Response methods - download, end, json, redirect, render, send, sendFile, sendStatus.

Introduction to Middleware:

Middleware in Express refers to functions that process requests before reaching the route handlers. These functions can modify the request and response objects, end the request-response cycle, or call the next middleware function. Middleware functions are executed in the order they are defined. They can perform tasks like authentication, logging, or error handling. Middleware helps separate concerns and manage complex routes efficiently.

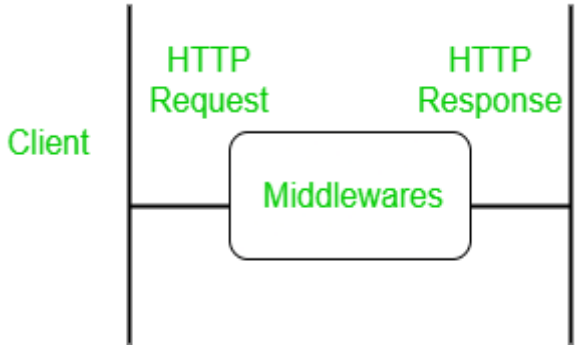


Figure: ***Middleware working***

Syntax:

```
app.use((req, res, next) => {
    console.log('Middleware executed');
    next();
});
```

- **(req, res, next) => {}:** This is the middleware function where you can perform actions on the request and response objects before the final handler is executed.
- **next():** This function is called to pass control to the next middleware in the stack if the current one doesn't end the request-response cycle.

What Middleware Does in Express.js

Middleware functions in Express.js can perform several important tasks:

1. **Execute Code:** Middleware can run any code when a request is received.
2. **Modify Request and Response**: Middleware can modify both the request (req) and response (res) objects.
3. **End the Request-Response Cycle:** Middleware can send a response to the client, ending the cycle.
4. **Call the Next Middleware:** Middleware can call next() to pass control to the next function in the middleware stack.

How Middleware Works in Express.js?

In Express.js, middleware functions are executed sequentially in the order they are added to the application. Here's how the typical flow works:

1. **Request arrives at the server**.
2. **Middleware functions** are applied to the request, one by one.
3. Each middleware can either:
   - **Send a response** and end the request-response cycle.
   - **Call** next() to pass control to the next middleware.
4. If no middleware ends the cycle, the **route handler** is reached, and a final response is sent.

Types of Middleware:

ExpressJS offers different types of middleware and you should choose the middleware based on functionality required.

1. Application-level middleware:

Application-level middleware in Express.js refers to middleware functions that are applied to the entire application rather than being tied to a specific route. These middleware functions are executed for every incoming request, regardless of the specific route being accessed. They are set up using the app.use() method.

```
const express = require('express')
const app = express()
app.use(express.json()); // Parses JSON data for every incoming request

// App level middleware
app.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})
```

2. Router-level middleware:

Router-level middleware in Express.js refers to middleware functions that are bound to a specific instance of the Express Router. Unlike application-level middleware, which is applied globally to the entire application, router-level middleware is specific to a particular set of routes defined by an instance of the Router.

```
const router = express.Router();
// Router-level middleware
router.use((req, res, next) => {
  console.log('This middleware runs for routes defined by the router.');
  next(); // Pass control to the next middleware or route handler
});
// Mount the router at a specific path
app.use('/myapp', router);
```

In this example, the router-level middleware is applied to all routes defined within the router instance. The app.use('/myapp', router) line specifies that the router is mounted at the path '/myapp', so the middleware and routes defined within the router will only be active for requests that start with '/myapp'.

3. Error-handling middleware:

Error handling middleware refers to a type of middleware that is specifically designed to handle errors that occur during the processing of a request. When an error occurs in an Express application, it can be caught by an error handling middleware, allowing you to handle and respond to the error in a centralized and organized way.

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something went wrong!')
})
```

Error-handling middleware always takes four arguments.

err: The error object.

req: The request object.

res: The response object.

next: The next middleware function.

You must provide four arguments to identify it as an error-handling middleware function. Even if you don't need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.

4. Built-in middleware:

Built-in middleware are middleware functions that are included with the Express framework by default. These middleware functions provide common and essential functionalities that can be used in web applications. Express.js simplifies the development process by using these built-in middleware, making it easy for developers to handle common tasks without having to implement everything from scratch.

Here are some examples of built-in middleware in Express.js:

1. ***express.json():*** This middleware parses incoming JSON data from the request body and makes it available in req.body.

2. ***express.urlencoded():*** This middleware parses incoming URL-encoded form data from the request body and makes it available in req.body.

3. ***express.static():*** This middleware serves static files (e.g., images, CSS, JavaScript) from a specified directory.

For example, express.static() serves files like images, and express.json() helps parse incoming JSON data.

```
app.use(express.static('public')); // Serves static files from the "public" folder
app.use(express.json()); // Parses JSON payloads in incoming requests
```

5. Third-party middleware:

Third-party middleware are middleware functions that are not part of the core Express framework but are created by external developers or third-party libraries. These middleware functions add additional features, functionalities, or integrations to an Express application, allowing developers to extend the capabilities of their applications by using existing third-party solutions.

Here are a few examples of popular third-party middleware in Express.js:

1. cookie-parser: Middleware for parsing cookies attached to incoming requests and making the parsed data available in req.cookies.

2. body-parser: A middleware that parses incoming request bodies in different formats, such as JSON and URL-encoded data, and makes the parsed data available in req.body.

3. express-session: Middleware for handling user sessions, allowing you to store and retrieve data associated with a user across multiple requests.

4.multer: Middleware for handling multipart/form-data, commonly used for file uploads.
5.passport: Middleware for validating and sanitizing input data in request bodies, query parameters, and headers.
For example, the morgan middleware logs HTTP requests, and body-parser helps parse incoming request bodies for easier handling of form data.

```
const morgan = require('morgan');
app.use(morgan('dev')); // Logs HTTP requests using the "dev" format

const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true })); // Parses URL-encoded bodies
```

Steps to Implement Middleware in Express:

Step 1: Initialize the Node.js Project: `npm init -y`

Step 2: Install the required dependencies. `npm install express`

Step 3: Set Up the Express Application *Filename: index.js*

```
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;

app.get('/', (req, res) => {
  res.send('<div><h2>Welcome to KMIT</h2><h5>Tutorial on Middleware</h5></div>');
});

app.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

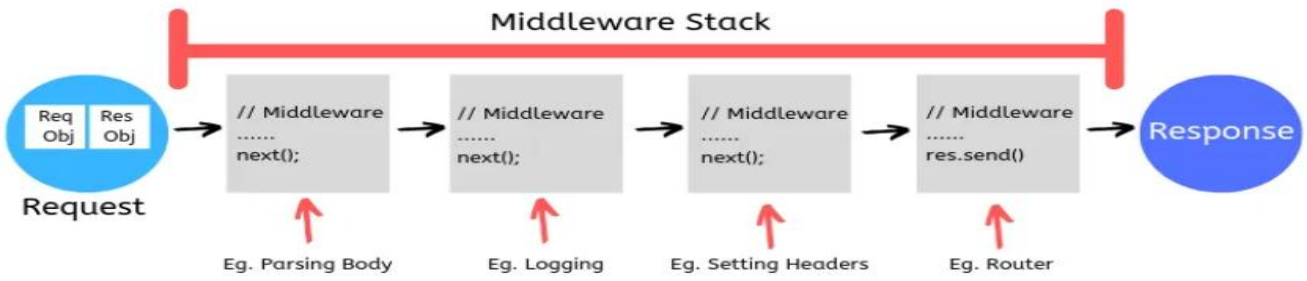
Step 4: Start the Application: `node index.js`

Output: *When you navigate to http://localhost:3000/, you will see:*

```
Welcome to GeeksforGeeksTutorial on Middleware
```

Middleware Chaining

Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser. So, before sending the response back to the browser the different middleware processes the request.
The next() function in the express is responsible for calling the next middleware function if there is one.
Modified requests will be available to each middleware via the next function



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

// Middleware 1: Log request method and URL
app.use((req, res, next) => {
  console.log(`${req.method} request to ${req.url}`);
  next();
});

// Middleware 2: Add a custom header
app.use((req, res, next) => {
  res.setHeader('X-Custom-Header', 'Middleware Chaining Example');
  next();
});

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

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

- **Middleware 1:** Logs the HTTP method and URL of the incoming request.
- **Middleware 2:** Sets a custom header X-Custom-Header in the response.
- **Route Handler:** Sends a "Hello, World!" message as the response.
-

Output: When a client makes a GET request to **http://localhost:3000/**, the server responds with:

Hello, World!

Advantages of using Middleware

- **Modularity**: Breaks down complex tasks into smaller, manageable functions.
- **Reusability:** Middleware functions can be reused across different routes or applications.
- **Maintainability**: Organizes code logically, making it easier to manage and update.
- **Error Handling:** Centralizes error handling, improving the application's robustness.
- **Performance Optimization**: Allows for tasks like caching, compression, and security checks to be handled efficiently.

Best Practices for Middleware in ExpressJS

- Always call next() to pass control to the next middleware unless you are ending the request-response cycle.
- Use specific middleware for different tasks (e.g., authentication, logging) to keep the code modular and clean.
- Place error-handling middleware at the end of the middleware stack to catch any unhandled errors.
- Avoid using synchronous code in middleware to prevent blocking the event loop.

ROUTE

- **Routing** describes how URI endpoints in an application react to client requests.
- Routing is the most important component of a web application because, without a proper API to handle requests and the ability to know the pieces of the puzzle, you will end up with a code base that no developer wants to look at.

Introduction:

- The word "routing" describes the process of deciding how an application will respond to a client request for a particular path and HTTP request type. (GET, POST, and so on).
- You specify routing using Express app object methods that are equivalent to HTTP methods, such as an app.get() for GET requests and app.post() for POST requests, etc.
- When the application gets a request to the specified route (endpoint) and HTTP method, it calls the callback function specified by these routing methods (also known as "handler functions").
- To put it another way, the application essentially "listens" for requests that match the route(s) and method(s) specified, and when it finds a match, it executes the callback function specified.
- In actuality, the callback functions that are input to the routing methods can be multiple.
- With numerous callback functions, it is crucial to pass control to the next callback by passing next as an parameter to the callback method and then calling next() inside the function body.

What is Routing?

Routing, put simply, is deciding which process is launched whenever a user navigates to a specific URL.

What are the Routing Methods?

A route function is derived from one of the HTTP methods and is attached to an express class instance.

Routing methods are defined as follows: **app.METHOD(PATH,CALLBACK)**

- This method instructs the server to execute the next CALLBACK function and send an HTTPMETHOD request if the user navigates to PATH.

Here, the following verbs are used in place of METHOD:

Get: To process GET requests (that is, to request/GET data from a given resource).
Post: To send info to a server to update an old resource or create a new resource.
Put: To send info to a server to create/update a resource.

- POST requests vary from PUT requests in that the latter are idempotent.
- This implies that if a PUT request is made multiple times, it has no additional effect.
- In comparison, calling a POST method more than once will cause side effects in your programme.
- As a result, keep in mind that POST requests should only be made once.

Delete: To remove a specific resource from the database.
Head: Like a GET request, the HEAD method asks that the target resource transfer a representation of its state, but without the representation data enclosed in the response body.This helps retrieve representation metadata from the response header without transferring the complete representation.

Connect: The connect method instructs the intermediary to create a TCP/IP tunnel to the origin server specified by the request destination. It is frequently used to secure communications via one or more HTTP proxies.
Options: The options method asks the destination resource to send the HTTP methods it supports. This can be used to test a web server's functionality by requesting * instead of a particular resource.
Trace: The trace method asks that the received request be transferred to the target resource in the response body. A client can then see what modifications or additions (if any) have been made by intermediaries.
Patch: This function requests that the target resource modify its state based on the partial update specified in the request's representation. This can save bandwidth by updating a portion of a file or document without transferring the complete thing.

Examples of the most frequently used are:

```
// GET method route
app.get('/', (request, response) => {
  res.send('GET request to the homepage')
})

// POST method route
app.post('/', (request, response) => {
  res.send('POST request to the homepage')
})
```

All HTTP request mechanisms are supported by Express, like get, post, put, delete etc.

app.all() is a special routing technique that loads middleware functions at a path for all HTTP request methods. ,
For example,, for requests to the route /xyz, whether using DELETE, POST, GET, OR PUT or any other HTTP request method supported by the http module, the following handler is performed.

```
app.all('/xyz', (request, response, next) => {
  console.log('Authentication or any other xyz section')

  // pass control to the next handler
  next()
})
```

Express Route Methods - GET, POST, PUT, DELETE

In Express.js, HTTP methods determine how a server handles client requests. They define the type of operation to perform on a resource, such as retrieving, creating, updating, or deleting data. Express allows defining routes that handle requests efficiently and in an organized manner.

Methods of HTTP Requests

Here are the different types of http requests

1. GET Method:

- The GET method is an HTTP request used by a client to retrieve data from the server.
- It takes two parameters: the URL to listen on and a callback function with req (client request) and res (server response) as arguments.
  Syntax: `app.get("URL",(req,res)=>{})`

In the above syntax

- app.get – defines a GET route in Express.
- "URL" – the path the route listens to.
- (req, res) => {} – callback function where req is the client request and res is the server response.

2. POST Method:

- The POST method sends data from the client to the server, usually to store it in a database.
- It takes two parameters: the URL to listen on and a callback function with req (client request) and res (server response).
- The data sent is available in the request body and must be parsed as JSON.
  Syntax: `app.post("URL",(req,res)=>{})`

In the above syntax

- app.post – defines a POST route in Express.
- "URL" – the path the route listens to.
- (req, res) => {} – callback function where req contains client data and res sends the server response.

3. PUT Method:

- The PUT method updates existing data in the database.
- It takes two parameters: the URL to listen on and a callback function with req (client request containing updated data in the body) and res (server response).
  Syntax: `app.put("URL",(req,res)=>{})`

In the above syntax:

- app.put – defines a PUT route in Express.
- "URL" – the path the route listens to.
- (req, res) => {} – callback function where req contains the client's updated data (usually in the body) and res sends the server response.

4. DELETE Method:

- The DELETE method removes data from the database.
- It takes two parameters: the URL to listen on and a callback function with req (containing the ID of the item to delete in the body) and res (server response).
  Syntax: `app.delete("URL",(req,res)=>{})`

In the above syntax

- app.delete – defines a DELETE route in Express.

- "URL" – the path the route listens to.
- (req, res) => {} – callback function where req contains data (e.g., ID to delete) and res sends the server response.

5. PATCH Method:

- The PATCH method is used to partially update data on the server.
- It takes two parameters: the URL to listen on and a callback function with req (containing the data to update, usually in the body or URL parameters) and res (server response).

Syntax: `app.patch("URL", (req, res) => {})`

In the above syntax:

- **app.patch:** defines a PATCH route in Express.
- **"URL":** the path the route listens to.
- **(req, res) => {}:** callback function where req contains the data to update (e.g., fields to change) and res sends the server response.
-

Example-1: Implementation of above HTTP methods.

```
const express = require('express');
const app = express();
const PORT = 3000;
app.use(express.json());

app.get("/", (req, res) => {
    console.log("GET Request Successfull!");
    res.send("Get Req Successfully initiated");
})

app.put("/put", (req, res) => {
    console.log("PUT REQUEST SUCCESSFUL");
    console.log(req.body);
    res.send(`Data Update Request Recieved`);
})

app.post("/post", (req, res) => {
    console.log("POST REQUEST SUCCESSFUL");
    console.log(req.body);
    res.send(`Data POSt Request Recieved`);
})

app.delete("/delete", (req, res) => {
    console.log("DELETE REQUEST SUCCESSFUL");
    console.log(req.body);
    res.send(`Data DELETE Request Recieved`);
})

app.listen(PORT, () => {
    console.log(`Server established at ${PORT}`);
})
```

Example-2: Book Store API – using Express(RestAPI-Postman) **FileName: app.js**

```
//install dependencie:  npm init -y and npm i express

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

const port=3000;
app.listen(port,()=>
{
    console.log(`server is running on port number ${port}`);

})

/*express.json():-Whenever someone sends data to your server (like through a POST or PUT request)
in JSON format,this middleware converts it into a JavaScript object automatically —
so you can easily access it using req.body.
*/
app.use(express.json())

let books=[
    {
        id: '1',
        title: 'book 1'
    },
    {
        id:'2',
        title:'book 2'
    }
]

//intro route
app.get('/',(req,res)=>{
    res.json({
        message:"welcome to boook store api"
    });

//get all books
app.get("/get",(req,res)=>{
    res.json(books);

});//now goto postman create new request with 'GET' for get all books


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

    const book=books.find((item)=> item.id==req.params.id);

    if(book)
    {
        res.status(200).json(book)
```

```
    }
    else
    {
        res.status(404).json(
            {
                 message: 'Book not found! Please try with a different ID'
            }
    );
    }
}
);
//add book
app.post('/add', (req,res)=>{
     const newbook={
        id: books.length + 1,
        title: `book ${books.length + 1}`
     }
      books.push(newbook)
      //res.status(200).json(newbook)
                res.status(200).json(
                {
                    data: newbook,
                     message: "new book is added successfully"
                }
            )
});
//update book detials
app.put('/update/:id',(req,res)=>
{
    const findcurrentbook=books.find((bookItem) =>bookItem.id==req.params.id);
    if(findcurrentbook)
    {
        findcurrentbook.title=req.body.title || findcurrentbook.title
        res.status(200).json({
            message: `book with id ${req.params.id} updated successfully`,
            data: findcurrentbook
        })
    }
    else
    {
              res.status(404).json({
            message: "book not found"
        });
    }

//delete a book
app.delete('/delete/:id', (req, res) => {
  const index = books.findIndex(b => b.id === req.params.id);
  if (index !== -1) {
    const deleted = books.splice(index, 1);
    res.json(
        {
```

```
            message: 'Book deleted successfully',
            data: deleted[0] });
  }
  else
    {
    res.status(404).json(
        {
         message: 'Book not found'
        });
  }
});
```

Run: node app.js

Output on terminal: server is running on port number 3000

Output-1: Method: **GET** URL: **http://localhost:3000/**

```
{
    "message": "welcome to boook store api"
}
```

Output-2 (get all books): Method: **GET** URL: **http://localhost:3000/get**

```
[
    {
        "id": "1",
        "title": "book 1"
    },
    {
        "id": "2",
        "title": "book 2"
    }
]
```

Output-3 (get single book): Method: **GET** URL: **http://localhost:3000/get/2**

```
{
    "id": "2",
    "title": "book 2"
}
```

Output-4 (add book): Method: **POST** URL: **http://localhost:3000/add**

```
{
    "data": {
        "id": 3,
        "title": "book 3"
    },
    "message": "new book is added successfully"
}
```

Output-5 (get all books): Method: **GET** URL: **http://localhost:3000/get**

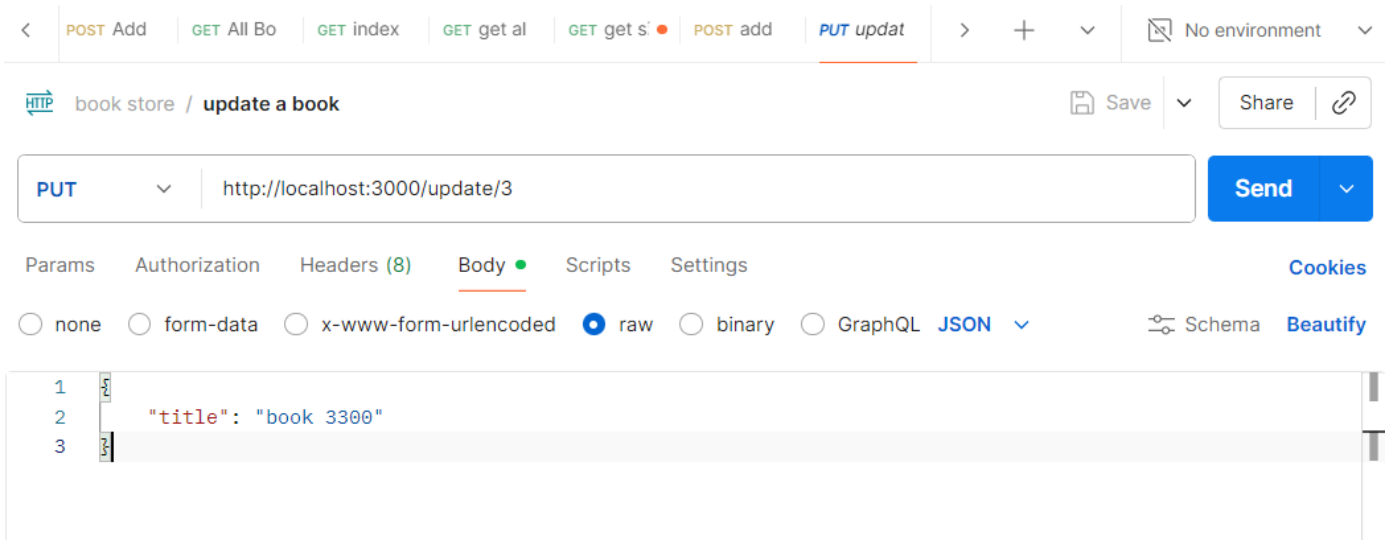
```
[
    {
        "id": "1",
        "title": "book 1"
    },
    {
        "id": "2",
```

```
        "title": "book 2"
    },
    {
        "id": 3,
        "title": "book 3"
    }
]
```

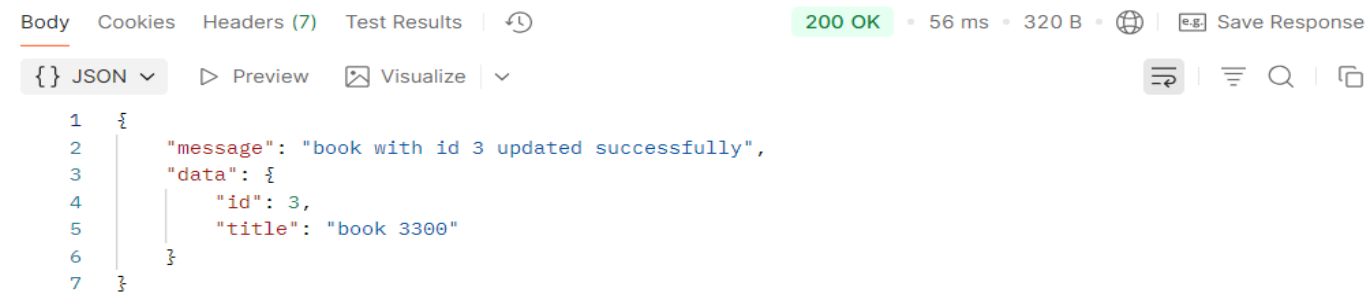

Output-6 (Update book details): Method: **PUT** URL: **http://localhost:3000/update/3**

In post man app, for add book details or update book details, select body and select radio button(raw) and enter the book update details (Example:selected book id: 3)

Input:



Output:



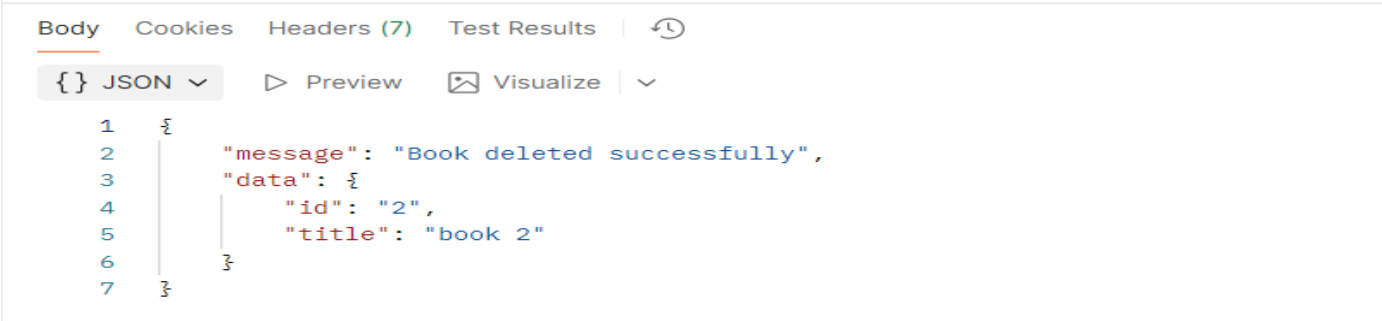
Output-7 (get all books): Method: **GET** URL: **http://localhost:3000/get**


```
[
    {
        "id": "1",
        "title": "book 1"
    },
    {
        "id": "2",
        "title": "book 2"
    },
```

```
    {
        "id": 3,
        "title": "book 3300"
    }
]
```

Output-8 (delete single book): Method: **DELETE** URL: http://localhost:3000/delete/2

(Example : delete the 2nd Book)



Output-9 (get all books): Method: **GET** URL: http://localhost:3000/get

Route Paths:-strings, string patterns or regular expressions, Route parameters.

- A **Route Path** in Express.js defines the **URL pattern** that the application should respond to. When a user sends a request to that specific path, Express matches it and executes the corresponding route handler.
- The endpoints at which requests may be made are defined by route pathways in conjunction with a request method. Strings, string patterns, or regular expressions can all be used as route pathways.
- The symbols ?, +, *, and () are subsets of their equivalents in regular expressions. By string-based pathways, the hyphen (-) and the dot (.) are taken literally.

If you need to use the dollar character ($) in a path string, enclose it escaped within ([and]).

Note:

- Express uses path-to-regexp to match the route paths; for a list of all possible ways to define a route path, see the path-to-regexp documentation. Even though it doesn't enable pattern matching, Express Route Tester is a useful tool for testing fundamental Express routes.
- The route path does not include query strings.

Each route path can be:

- a string
- a string pattern
- or a regular expression

General Syntax: **app.METHOD(PATH, HANDLER)**

METHOD – HTTP method (GET, POST, PUT, DELETE, etc.)

PATH – URL path (can be string, string pattern, or regex)

HANDLER – Function executed when route matches

1. Route Paths as Strings

Definition: A string path defines a **fixed URL** — it matches the route **exactly** as written.
Use this when you know the exact path name.

Example 1: Simple String Path

```
app.get('/home', (req, res) => {
  res.send('Welcome to the Home Page');
});
```

Matches → `/home` **Does not match →** `/homepage`, `/Home`

Example 2: Multiple Fixed Routes

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

app.get('/contact', (req, res) => {
  res.send('Contact Page');
});
```

Each path is fixed and must be typed exactly.

Example 3: Nested Path

```
app.get('/user/profile', (req, res) => {
  res.send('User Profile Page');
});
```

Matches → `/user/profile` **Does not match →** `/user` **or** `/profile`

Example 4: Case-Sensitive Path

```
app.get('/Login', (req, res) => {
  res.send('Case-sensitive route');
});
```

Matches → `/Login` **Does not match →** `/login` **(if case sensitive routing enabled)**

2. Route Paths as String Patterns

Definition: String patterns let you use **special characters** in the path string to match **variations** of URLs.

Common Special Characters

Symbol	Meaning	Example	Matches
?	Optional character	/ab?cd	/abcd or /acd
+	One or more of the same character	/ab+cd	/abcd, /abbcd, /abbbcd
*	Any character(s) in that position	/ab*cd	/abcd, /abXcd, /ab123cd
()	Grouping	/a(bc)?d	/ad, /abcd

Example 1: Using "?" (Optional Character)

```
app.get('/ab?cd', (req, res) => {
  res.send('Matched route with optional b');
});
```

- ✓ **Matches → /abcd, /acd**

Example 2: Using "+" (One or More Occurrences)

```
app.get('/ab+cd', (req, res) => {
  res.send('Matched route with one or more b's');
});
```

- ✓ **Matches → /abcd, /abbcd, /abbbcd**

Example 3: Using "*" (Wildcard)

```
app.get('/ab*cd', (req, res) => {
  res.send('Matched route with anything in between ab and cd');
});
```

- ✓ **Matches → /abcd, /abXcd, /ab123cd, /abHelloWorldcd**

Example 4: Grouping "( )"

```
app.get('/a(bc)?d', (req, res) => {
  res.send('Matched route with optional "bc"');
});
```

- ✓ **Matches → /ad, /abcd**

3. Route Paths as Regular Expressions

Definition: Regular expressions (regex) give you **full control** to match complex URL patterns.
They start and end with '/' and are written **without quotes**.

Use this when you want to handle URLs that follow a particular pattern rather than a fixed name.

Example 1: URL Ending with a Word

```
app.get(/.*fly$/, (req, res) => {
  res.send('This route ends with "fly"');
});
```

✓ **Matches → `/butterfly`, `/dragonfly`** **Does not match → `/flyer`, `/flying`**

Example 2: URL Starting with a Word

```
app.get(/^\/api/, (req, res) => {
  res.send('All API routes start with /api');
});
```

✓ **Matches → /api, /api/users, /api/data** **Does not match → /user/api**

Example 3: Exact Numeric Path

```
app.get(/^\/[0-9]+$/, (req, res) => {
  res.send('Path contains only numbers');
});
```

✓ **Matches → `/123`, `/45678`** **Does not match → `/abc`, `/12a`**

Example 4: URL with Letters Only

```
app.get(/^\/[A-Za-z]+$/, (req, res) => {
  res.send('Letters-only path matched');
});
```

✓ **Matches → /home, /welcome, /Express** **Does not match → /home123, /abc_def**

Final Example: **File name :app.js**

```
// Import Express
const express = require('express');
const app = express();
const port = 3000;

// String Route Path
app.get('/home', (req, res) => {
  res.send('String path matched → /home');
});

// String Pattern Route Path
// Example: /ab?cd → Matches /abcd or /acd
app.get('/ab?cd', (req, res) => {
  res.send('String pattern matched → /abcd or /acd');
});

// Example: /ab+cd → Matches /abcd, /abbcd, /abbbcd
app.get('/ab+cd', (req, res) => {
  res.send('String pattern matched → /abcd, /abbcd, etc.');
});
```

```
// Regular Expression Route Path
// Example: Matches any path that ends with "fly"
app.get(/.*fly$/, (req, res) => {
  res.send('Regex path matched → ends with "fly" (e.g., /butterfly, /dragonfly)');
});

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

Run using: node app.js
Open your browser or Postman and test the following URLs:

- http://localhost:3000/home
- http://localhost:3000/abcd
- http://localhost:3000/acd
- http://localhost:3000/abbcd
- http://localhost:3000/butterfly

You'll see different responses depending on the **route path type** matched.

4. Route parameters:

- ✓ The values supplied at their place in the URL are captured by route parameters, also known as URL segments.
- ✓ The names of the route parameters supplied in the path are used as the keys for the captured values in the **req.params object**.
- ✓ In Express.js, route parameters are used to capture dynamic values from the URL.
- ✓ They are specified in the route path as a placeholder preceded by a colon ':' .
- ✓ When a request is made to a route that matches the path pattern, Express.js automatically extracts the value of the parameter from the URL and adds it to the request object as req.params.

Syntax:

```
app.METHOD('/route/:parameterName', (req, res) => {
   res.send(req.params.parameterName);
});
```

Here are a few ways to work with route parameters in Express.js:

1. Accessing Route Parameters in a Route Handler:

- ✓ In the route handler function, you can access the route parameter value using **req.params.parameterName.**
- ✓ For example, **if the route path is /users/,** you **can access the value of id using req.params.id.**

Example:

```
app.get('/students/:id', (req, res) => {
 res.send(`Student ID is: ${req.params.id}`);
});
```

Try:

- http://localhost:3000/students/101 → *Output:* `Student ID is: 101`
- http://localhost:3000/students/200 → *Output:* `Student ID is: 200`

Explanation: `:id` is a placeholder for a value that appears in the same position in the URL.

2. Using Multiple Route Parameters in a Single Route:

- ✓ You can define multiple route parameters in a single route by separating them with slashes in the route path.
- ✓ For example, if you want to capture both user ID and book ID, you can define the route path as /users//books/.
- ✓ Then, you can access the values of both parameters using **req.params.userId** and **req.params.bookId**, respectively.

Example:

```
app.get('/users/:userId/books/:bookId', (req, res) =>
  {
      const userId = req.params.userId;
      const bookId = req.params.bookId;
      res.send(`User ID: ${userId}, Book ID: ${bookId}`);
});
```

Explanation:

- `:userId` and `:bookId` are **route parameters**.
- They act as **placeholders** in the URL that Express automatically fills with the actual values from the request.

URL: http://localhost:3000/users/10/books/55
Output: **User ID: 10, Book ID: 55**

(or)

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params);
});
```

URL: http://localhost:3000/users/10/books/500
Output:

```
{
  "userId": "10",
  "bookId": "500"
}
```

3. Defining Optional Route Parameters:

- ✓ You can define optional route parameters by adding a question mark after the parameter name in the route path.
- ✓ For example, if you want to capture an optional parameter status, you can define the **route path as /users//?.**
- ✓ If the status parameter is not present in the URL, **req.params**.the status will be undefined.

Example:

```
app.get('/student/:name?', (req, res) => {
  if (req.params.name)
    res.send(`Student Name: ${req.params.name}`);
  else
    res.send('No name provided');
});
```

Try:

- `/student/Ravi` → `Student Name: Ravi`
- `/student` → `No name provided`

Explanation: `?` makes the parameter optional.

4. Defining Route Parameters with Regular Expression Patterns:

- ✓ You can define route parameters with regular expression patterns to restrict the type of value that can be captured.
- ✓ For example, if you want to capture only numeric values for the user ID, you can define the **route path as /users/(\d+)**.
- ✓ The regular expression pattern \d+ matches one or more digits.

Example:

```
app.get('/users/:id(\\d+)', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
```

Explanation:

- `:id` → is a **route parameter**.
- `(\\d+)` → is a **regular expression** that restricts the parameter to **digits only**.
  - `\\d` → means a digit (0–9)
  - `+` → means one or more digits

So this route will **only match numeric user IDs**.

URL: http://localhost:3000/users/123
Output: User ID: 123
URL: http://localhost:3000/users/abc
Output: `Cannot GET /users/abc`
(because abc is not a number — it fails the \d+ regex check)

5. Using Middleware to Handle Common Functionality for Specific Route Parameters:

- ✓ You can define middleware that will be executed only for routes that have a certain route parameter.
- ✓ This can be useful for performing validation or pre-processing on the parameter value before executing the route handler function.
- ✓ You can define middleware functions using the **app.param() method**.
- ✓ For example, you can define a middleware function to validate **the user ID parameter** like this:

Example:

```
app.param('id', (req, res, next, id) => {
  if (isValidUserId(id)) {
    req.params.id = id;
    next();
  } else {
    res.status(400).send('Invalid user ID');
  }
});
```

What It Does:

- `app.param('id', ...)` registers a **param middleware** that automatically runs **whenever a route contains the parameter `:id`**.
- It's often used for:
  - **Validation** (checking if the ID format is correct)
  - **Preloading data** (like fetching a user from a database)

Explanation of Each Argument:

Parameter	Meaning
`'id'`	The route parameter name to watch (`:id`)
`(req, res, next, id)`	Callback executed when a route with `:id` is matched
`id`	The actual value passed in the URL
`next()`	Passes control to the next middleware or route handler

Example:

```
// Example validator
function isValidUserId(id) {
  return /^\d+$/.test(id); // Only digits allowed
}

// Param middleware
app.param('id', (req, res, next, id) => {
  if (isValidUserId(id)) {
    next();
  } else {
    res.status(400).send('Invalid user ID');
  }
});

// Route using :id
app.get('/users/:id', (req, res) => {
  res.send(`User ID is valid: ${req.params.id}`);
});
```

Example Output:

Request	Output
`GET /users/123`	✓ `User ID is valid: 123`
`GET /users/abc`	✗ `Invalid user ID` (400 Bad Request)

- This middleware function will be executed only for routes that have the: id route parameter.
- It checks whether the id parameter is valid, and if it is, it **sets req.params.id** and calls **next()** to pass control to the next middleware function or route handler function.
- If the id parameter is not valid, it sends a 400 Bad Request response.

Route Handlers - as a form of a function, an array of functions, or combinations of both.

- A single HTTP transaction can be roughly described by the Request and Response cycle.
  - The server receives a message from the client.
  - After receiving the request, the server reads the info. (request headers, URL path, HTTP method, query parameters, cookies, data or payload, etc.).
  - The client receives a reply from the server. The status number, headers, content encoding, and any returned data are all included
  - The HTTP transaction is finished once the answer has been returned.
- Multiple middleware functions can be used to process a request, with the ability to bypass the remaining callbacks by invoking the next('route').
- This can be useful to set conditions on a route and then move on to other routes if necessary.

Route handlers can take various forms, such as

1. a single function,
2. an array of functions, or
3. combination of both.

1. Single Function as Route Handler

A simple route handler where one function handles the entire request–response cycle.

Example 1:

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

app.get('/home', (req, res) => {
  res.send('Welcome to Home Page!');
});
app.listen(3000, () => console.log('Server running on port 3000'));
```

Explanation:

- `app.get('/home', ...)` handles **GET** requests to `/home`.
- The single callback `(req, res)`:
  - Receives the request.
  - Sends `"Welcome to Home Page!"` as the response.
- No `next()` call here — it ends the request–response cycle.

Output:

URL → `http://localhost:3000/home`
Shows → `Welcome to Home Page!`

Example 2:

```
app.post('/login', (req, res) => {
  res.send('Login Successful!');
});
```

Explanation:

- This route handles **POST** requests to `/login`.
- It immediately sends back `"Login Successful!"` as the response.
- Used for form submissions or login operations.

Output:

URL → `http://localhost:3000/login`
Shows → `Login Successful!`

Example 3:

```
app.delete('/user', (req, res) => {
  res.send('User deleted successfully!');
});
```

Explanation:

- Handles **DELETE** requests to `/user`.
- When triggered, it responds with `"User deleted successfully!"`.
- Commonly used for deleting a record.

Output:

URL → `http://localhost:3000/user`
Shows : `User deleted successfully!`

2. An array of functions

➢ Here, multiple functions are executed **in sequence** (middleware chain).
Each function can modify the request, perform checks, or log info before sending a response.

Example 1:

```
const express = require('express');
const app = express();
app.listen(3000, () => {
  console.log('Server running on port 3000');
});
const logTime = (req, res, next) => {
  console.log('Time:', Date.now());
  next();
};
const checkUser = (req, res, next) => {
  console.log('Checking user authentication...');
  next();
};
const sendResponse = (req, res) => {
  res.send('User authenticated and request handled!');
};

app.get('/profile', [logTime, checkUser, sendResponse]);
```

Explanation:

1. **`logTime`** → Logs the current timestamp, then calls next`()`.
2. **`checkUser`** → Logs an authentication check, then calls next().
3. **`sendResponse`** → Sends the final message.

Each middleware is executed **in sequence**.

Console Output:

```
Server running on port 3000
Time: 1761960011475
Checking user authentication...
```

Browser Output: **URL: http://localhost:3000/profile**

```
User authenticated and request handled!
```

Example 2:

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

app.listen(3000, () => {
  console.log('Server running on port 3000');
});

function step1(req, res, next) {
  console.log("Step 1 complete");
  next();
}

function step2(req, res, next) {
  console.log("Step 2 complete");
  next();
}

function step3(req, res) {
  res.send("All steps completed successfully!");
}

app.get('/steps', [step1, step2, step3]);
```

Explanation:

- Three middleware functions executed in order.
- `step1` and `step2` call `next()` to move forward.
- `step3` sends the final response and ends the cycle.

Console Output:

```
Server running on port 3000
Step 1 complete
Step 2 complete
```

Browser Output: **URL: http://localhost:3000/steps**

All steps completed successfully!

Example 3:

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

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
app.get('/verify', [
  (req, res, next) => {
    console.log('Middleware 1');
    next();
  },
  (req, res, next) => {
    console.log('Middleware 2');
    next();
```

```
  },
  (req, res) => {
    res.send('Verification complete!');
  }
]);
```

Explanation:

- Three inline middleware functions.
- Each one executes in order.
- The last one sends the response.

Console Output:

```
Server running on port 3000
Middleware 1
Middleware 2
```

Browser Output: **URL:** http://localhost:3000/verify

Verification complete!

3. Combination of both

A combination of independent functions and arrays of functions can handle a route.

Example 1:

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

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
const mid1 = (req, res, next) => {
  console.log('Middleware 1');
  next();
};
const mid2 = (req, res, next) => {
  console.log('Middleware 2');
  next();
};
app.get('/combo', [mid1, mid2], (req, res) => {
  res.send('Combination route executed successfully!');
});
```

Explanation:

- `[mid1, mid2]` runs first.
- Each logs a message and calls `next()`.
- Final function sends `"Combination route executed successfully!"`.

Console Output:

```
Server running on port 3000
Middleware 1
Middleware 2
```

Browser Output:

URL: http://localhost:3000/combo

```
Combination route executed successfully!
```

Example 2:

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

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
function logRequest(req, res, next) {
  console.log(`Request for ${req.url}`);
  next();
}
function auth(req, res, next) {
  console.log('Authenticating...');
  next();
}
app.get('/dashboard', logRequest, [auth], (req, res) => {
  res.send('Welcome to the Dashboard!');
});
```

Explanation:

- `logRequest` runs first → logs request info.
- `[auth]` runs next → simulates authentication.
- Final callback sends `"Welcome to the Dashboard!"`.

Console Output:

```
Server running on port 3000
Request for /dashboard
Authenticating...
```

Browser Output: URL: http://localhost:3000/dashboard

```
Welcome to the Dashboard!
```

Example 3

```
const express = require('express');
const app = express();
app.listen(3000, () => {
  console.log('Server running on port 3000');
});

app.post('/submit', [
  (req, res, next) => {
    console.log('Step 1');
    next();
  }
], (req, res, next) => {
  console.log('Step 2');
  next();
}, (req, res) => {
  res.send('Form Submitted!');
});
```

Explanation:

1. First function in array → logs `"Step 1"`, calls `next()`.
2. Second → logs `"Step 2"`, calls `next()`.
3. Final → sends `"Form Submitted!"`.

Console Output:

```
Server running on port 3000
Step 1
Step 2
```

In Post man:(because this is post request)

Method: POST URL: http://localhost:3000/submit

```
Form Submitted!
```

Summary Table

Type	Example	Flow
Single Function	`/home`, `/login`, `/user`	One function handles request and sends response
Array of Functions	`/profile`, `/steps`, `/verify`	Multiple middlewares executed in order
Combination	`/combo`, `/dashboard`, `/submit`	Mix of arrays + single functions, executed sequentially

Response methods - download, end, json, redirect, render, send, sendFile, sendStatus.

The request-response cycle can be ended by using the methods on the response object (res) in the accompanying table to send a response to the client. The client request will be abandoned if none of these techniques are used by a route handler.

In Express, a few of the response methods are as follows:

res.send(): to send data

res.status(): Specify HTTP response code

res.json(): to return the JSON data We are going to look at the syntax and examples in detail in the later section of the article.

What are Response Methods?

The Response object (response) provides the HTTP response that an Express app sends when it receives an HTTP request.

Let's look at some properties of the response object.

Properties	Description
res. app	It keeps track of the express application instance that is utilizing the middleware.
res.headersSent	Whether the app supplied HTTP headers with the response is indicated by this Boolean value.
res. locals	It designates an object with response local variables that are focused on the request.

Different Types of Response Methods

Method	Description
`res.send()`	Sends text, HTML, or objects as response.
`res.json()`	Sends a JSON response.
`res.redirect()`	Redirects to another route or website.
`res.sendFile()`	Sends a file as a response.
`res.download()`	Sends a file for the user to download.
`res.end()`	Ends the response without sending any data.
`res.render()`	Renders a template view (used with EJS, Pug, etc.).
`res.sendStatus()`	Sets the status code and sends its message.

1. res.send(): Most people are already familiar with the res.send() method. You can answer HTTP requests with a variety of data types using **res. send().** sends a response of **text, HTML, or even an object** directly to the browser.

Syntax: res.send([body])

Example:

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

app.get('/send', (req, res) => {
  res.send('<h1>Hello Student!</h1><p>This response is sent using
<b>res.send()</b></p>');
});

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

Console Output: **Server running at http://localhost:3000/send**

URL: http://localhost:3000/send

Browser Output:

Hello Student!

This response is sent using res.send()

2. res.json(): It is used for sending a JSON response. This function uses JSON.stringify() to transform the parameter into a JSON string and sends a response (with the correct content type).

Syntax: res.json([body])

Example:

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

const app = express();

app.get('/json', (req, res) => {
  res.json({
    message: 'Welcome to JSON Response',
    success: true,
```

```
    user: { name: 'John', role: 'Student' }
  });
});

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

Console Output: Server running at http://localhost:3000/json
URL: http://localhost:3000/json
Browser Output:

```
{
  "message": "Welcome to JSON Response",
  "success": true,
  "user": { "name": "John", "role": "Student" }
}
```

3. res.redirect(): redirects to a URL derived from the given path with the specified status, which is a positive number that matches an HTTP status code. Status defaults to "302 "Found" if it is not given.

Syntax: res.redirect([status,] path)

Example:

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

app.get('/home', (req, res) => {
  res.send('You are now on the Home Page!');
});

app.get('/redirect', (req, res) => {
  res.redirect('/home');
});

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

Console Output: Server running at http://localhost:3000/redirect
URL: http://localhost:3000/rdirect
Browser Output:

You are now on the Home Page!

4. res.sendFile(): Transfers the file at the specified path. Sets the Content-Type response HTTP header field based on the extension of the filename. The path should be a complete path to the file unless the root option is provided in the options object.

Syntax: res.send file(path [, options] [, fn])

Example: Create File: Create a folder public and add a file info.txt

info.txt: Keshav Memorial Institute of Technology

app.js:

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

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

app.get('/file', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'info.txt'));
});

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

Console Output: Server running at http://localhost:3000/file
URL: http://localhost:3000/file
Browser Output:

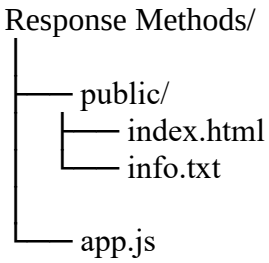
Keshav Memorial Institute of Technology

5. res.download():

The file at the path is transferred as an "attachment." Browsers will typically prompt the user to download. The path argument is used by default to derive the Content-Disposition header "filename=" parameter, although this can be modified with the filename parameter. If the path is relative, it will be based on the process's current working directory or the root option, if specified.

Syntax: res.download(path [, filename] [, options] [, fn])

Example: **Create File: Create a folder** public **and** add a file info.txt



File Name: app.js

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

// 🔽 Define the download route FIRST
app.get('/download', (req, res) =>
 {
  const filePath = path.join(__dirname, 'public', 'info.txt');
  console.log('Trying to download:', filePath);

  res.download(filePath, 'student_notes.txt', (err) => {
    if (err) {
      console.error(' Error while downloading:', err);
```

```
      res.status(500).send('Error downloading the file.');
    }
  });
});

// 🔹 Optional: root route just to check server is working
app.get('/', (req, res) => {
  res.send('Server is running. Try /download');
});

// 🔹 Start server
app.listen(3017, () => {
  console.log('✅ Server running at http://localhost:3017');
});
```

File Name: index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Download Example</title>
</head>
<body style="font-family: Arial; text-align: center; margin-top: 50px;">
  <h1>📄 Download Student Notes</h1>
  <p>Click the button below to download your notes file.</p>

  <a href="/download">
    <button style="padding: 10px 20px; font-size: 16px; cursor: pointer;">
      Download Notes
    </button>
  </a>
</body>
</html>
```

File Name: info.txt

Keshav Memorial Institute of Technology

Run: **node app.js**

Console Output: Server running at http://localhost:3017/download

URL: http://localhost:3017/download

Expected result:

The browser will immediately start downloading a file named **student_notes.txt** (which is actually your `info.txt`).

6. res.end():

The response procedure will end using this method. This method was inspired by Node core, notably the response. end() method of HTTP.ServerResponse. Use to swiftly terminate the response with no data. If you want to send the response with data, instead use res. send() and res.json().

Syntax: res.end([data] [, encoding])

Example: app.js

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

app.get('/end', (req, res) => {
  res.write('Response is partially written...');
  res.end('Now connection is closed using res.end()');
});

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

Console Output: Server running at http://localhost:3000/end

URL: http://localhost:3000/end

Browser Output:

Response is partially written...Now connection is closed using res.end()

EJS (Embedded JavaScript)

EJS stands for **Embedded JavaScript Templates**.: It allows you to **generate HTML dynamically** by embedding JavaScript code inside your HTML files.

EJS is a **template engine** —a tool that lets you **insert dynamic data (variables, arrays, objects, etc.) into HTML pages** before sending them to the browser.

Why use EJS in Node.js?

- It helps render **dynamic content** (e.g., user names, product lists, etc.).
- It integrates easily with **Express.js**.
- You can use normal **HTML + JS** syntax (easy to learn).

Installation

In your Node.js + Express project folder, run: **npm install ejs**

Example: EJS Project Structure

```
myapp/
│
├── views/
│   ├── index.ejs
│   └── about.ejs
│
├── app.js
├── package.json
```

What Are EJS Tags?

EJS (Embedded JavaScript) uses **special tags** inside HTML to tell the template engine **where to run JavaScript code or show data**.

All EJS tags start with `<%` and end with `%>` (between them, you write JavaScript or variables).

1.Scriptlet Tag — <% %>

Purpose: Run JavaScript code inside the template
Does not print anything to HTML page.

Example

```
<% let age = 20; %>
<% if (age >= 18) { %>
  <p>You are an adult.</p>
<% } else { %>
  <p>You are a minor.</p>
<% } %>
```

Explanation:

- Inside `<% %>` you can write normal JS (`if`, `for`, variables, etc.)
- EJS executes it but **doesn't print** anything unless you use `<%= %>`.

2. Output Tag — <%= %>

Purpose: Print (display) a **variable or value** on the web page
Escapes HTML (for safety — avoids malicious code).

Example

```
<% let name = "Rakesh"; %>
<h2>Welcome, <%= name %></h2>
```

Output (HTML)

```
<h2>Welcome, Rakesh</h2>
```

If `name` contained `<script>` tags, they would be escaped automatically to prevent XSS (cross-site scripting).

3. Unescaped Output Tag — <%- %>

Purpose: Print raw HTML (without escaping special characters)
Use carefully — it can make your page vulnerable if data is user-provided.

Example

```
<% let htmlData = "<b>Hello Bold Text</b>"; %>
<p><%- htmlData %></p>
```

Output (HTML)

Hello Bold Text

Note: If you used `<%= htmlData %>` instead, you'd see `<b>Hello Bold Text</b>` as plain text.

4.Comment Tag — <%# %>
Purpose: **Write comments inside EJS that** won't appear in final HTML

Example

<%# This is a comment and will not appear in output %>

<h3>Visible content</h3>

Output (HTML)
<h3>Visible content</h3>

5. Literal Tag — <%% %>

Purpose: Display `<%` or `%>` symbols as text on the web page.

Example

Use EJS syntax like <%%= name %> to print variables.

Output (HTML)

Use EJS syntax like <%= name %> to print variables.

Combining Tags Example

```
<% let students = ["Rakesh", "Meena", "Arjun"]; %>
<ul>
  <% students.forEach(student => { %>
    <li><%= student %></li>
  <% }) %>
</ul>
```

Output

```
<ul>
  <li>Rakesh</li>
  <li>Meena</li>
  <li>Arjun</li>
</ul>
```

Tag	Name	Example	Purpose	Output
`<% %>`	**Scriptlet Tag**	`<% if(age > 18){ %> Adult <% } %>`	Runs JS code (no output)	Not printed
`<%= %>`	**Output Tag**	`<%= name %>`	Prints escaped variable	Text only

Tag	Name	Example	Purpose	Output
`<%-%>`	**Unescaped Output Tag**	`<%- htmlData %>`	Prints raw HTML	HTML rendered
`<%#%>`	**Comment Tag**	`<%# This won't show %>`	Internal comment	Hidden
`<%%%>`	**Literal Tag**	`<%%= name %>`	Prints `<%= name %>` literally	`<%= name %>` text

7. res.render()

If you use Express in conjunction with a template engine like Pug or EJS, this technique will automatically compile the templates to standard HTML and deliver client feedback.

Syntax: res.render(view [, locals] [, callback]

- **locals:** an object called locals, whose properties specify local variables for the view.
- **a callback function:** The method does not execute an automatic response but instead returns the rendered string and any potential errors if they are provided. When an error occurs, the method internally calls next(err).
- **View:** The string file path of the view file to be rendered is the view argument.

Exampl-1:

Step 1: Create Folder Structure

```
myapp/
│
├── views/
│   ├── index.ejs
│   └── about.ejs
│
├── app.js
└── package.json
```

Step 2: Install Dependencies: Run these commands in your terminal:

```
npm init -y
npm install express ejs
```

Step 3: app.js

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

// Step 1: Tell Express that we are using EJS template engine
app.set('view engine', 'ejs');
/*Whenever I use res.render(), find an EJS file with that name in the views folder."
By default, Express will look for .ejs files inside a folder called views/
— located in the same directory as your app.js.
*/
```

```
// Step 2: Define a route that uses res.render()
app.get('/', (req, res) => {
  res.render('index', { name: 'John', age: 25 });

});

app.get('/about', (req, res) => {
  res.render('about', { college: 'KMIT', city: 'Hyderabad' });
});

// Step 3: Start the server
app.listen(3010, () => console.log('Server running on http://localhost:3010'));
```

index.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title>Home Page</title>
</head>
<body>
  <h1>Welcome <%= name %></h1>
  <p>Your age is <%= age %></p>
</body>
</html>
```

home.ejs:

```
<!DOCTYPE html>

<html>
<head>
  <title>About Page</title>
</head>
<body>
  <h1>About <%= college %></h1>
  <p>Located in <%= city %></p>
</body>
</html>
```

How to Run

Step 1: Open terminal inside project folder and run:

node app.js

Step 2: Open browser and go to:

http://localhost:3000

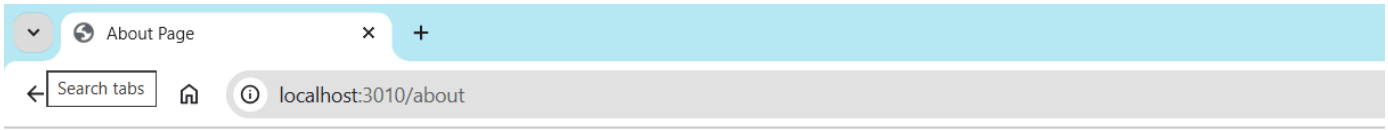
You'll see your EJS page dynamically rendered!

Browser Output: URL: http://localhost:3010/

Welcome John

Your age is 25

URL: http://localhost:3010/about



About KMIT

Located in Hyderabad

Summary

Concept	Description
File Extension	`.ejs`
Default Folder	`views/`
Setup	`app.set('view engine', 'ejs')`
Run Command	`node app.js`
Server URL	`http://localhost:3000`
Tags	`<% %>, <%= %>, <%- %>, <%# %>, <%% %>`

Example-2:

Step 1: Create Folder Structure

```
myapp/
│
├── templates/
│   ├── index.ejs
│   └── about.ejs
│
├── app.js
└── package.json
```

app.js

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

app.set('view engine', 'ejs');  // tells Express to use EJS

app.set('views', './templates'); // tells Express where EJS files are stored
//This line changes the default folder where Express looks for your EJS files.
//“Instead of the default /views folder, look inside the /templates folder for my .ejs files.”

app.get('/', (req, res) => {
  res.render('home', { name: 'JOHN' });//Express automatically knows to look for a file called:
});

app.get('/about', (req, res) => {
  res.render('about', { city: 'Hyderabad' });
});

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

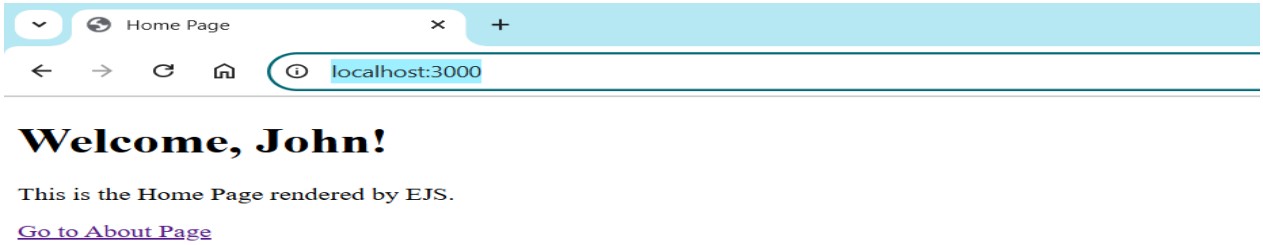
home.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title>Home Page</title>
</head>
<body>
  <h1>Welcome, <%= name %>!</h1>
  <p>This is the Home Page rendered by EJS.</p>
  <a href="/about">Go to About Page</a>
</body>
</html>
```

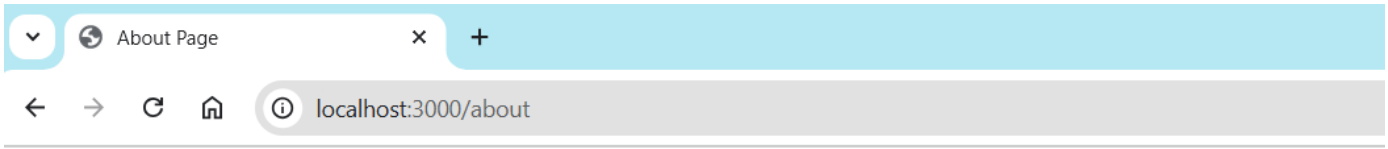
about.ejs:

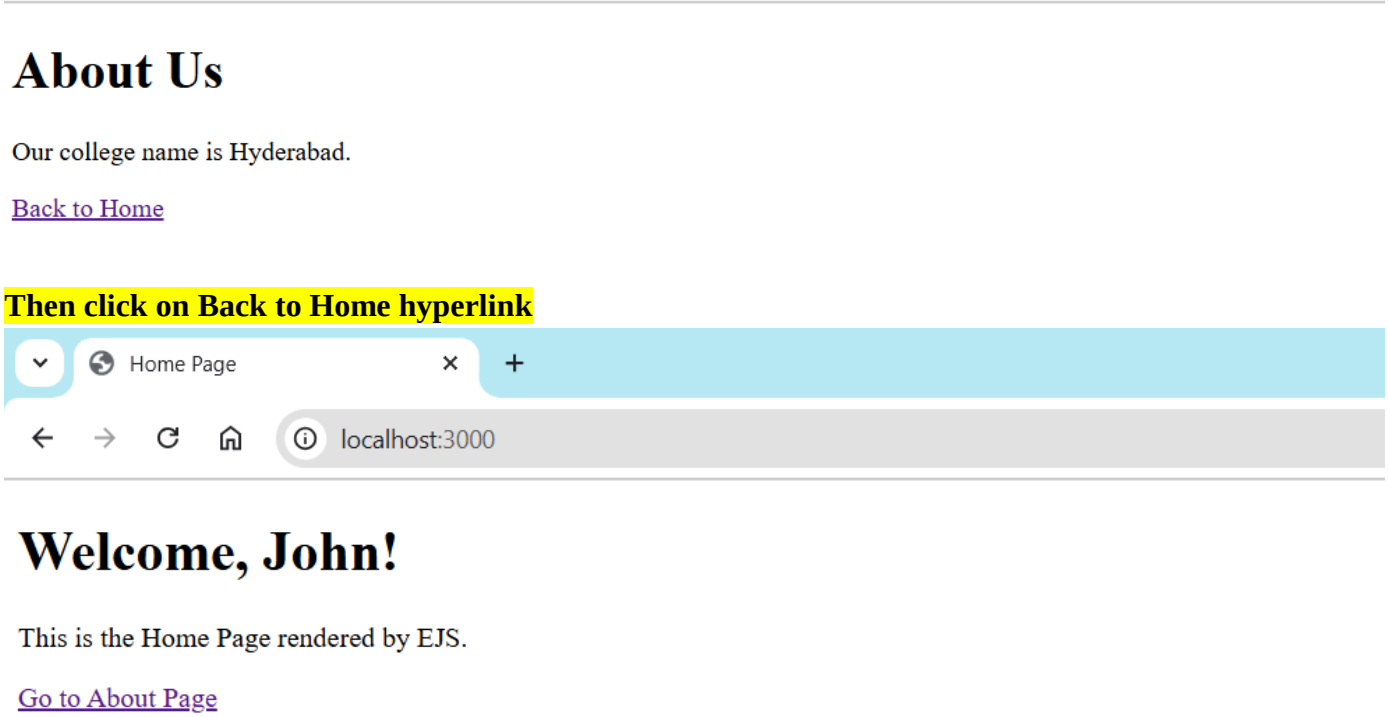
```
<!DOCTYPE html>
<html>
<head>
  <title>About Page</title>
</head>
<body>
  <h1>About Us</h1>
  <p>Our college name is <%= city %>.</p>
  <a href="/">Back to Home</a>
</body>
</html>
```

Browser output: URL: http://localhost:3000/



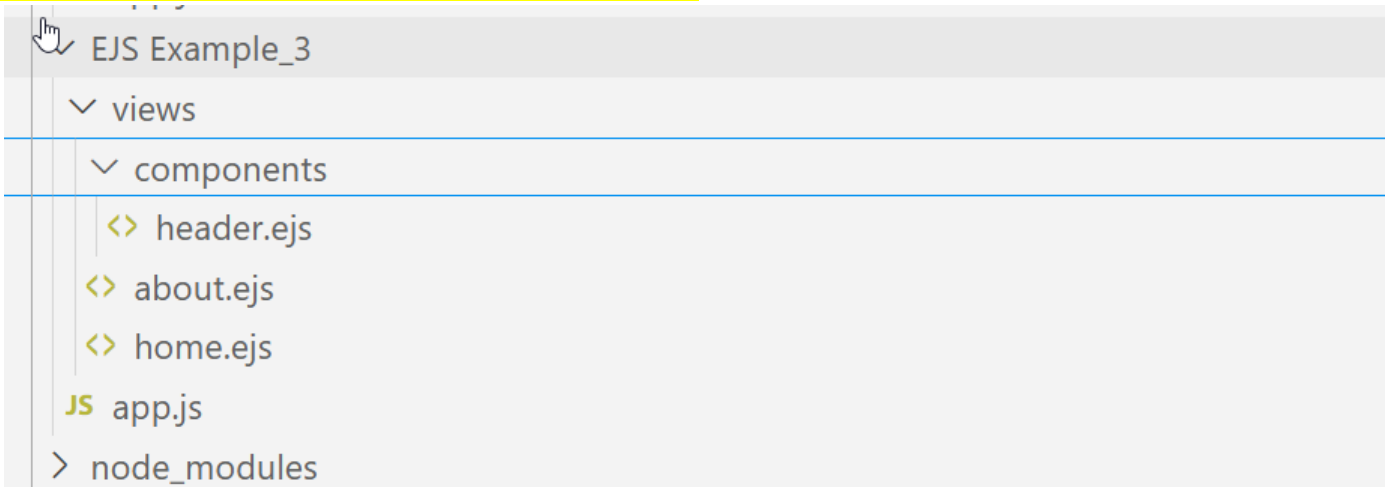
Then click on goto about page hyperLink





Then click on Back to Home hyperlink

Example-3: how to include a ejs file into other ejs file



app.js

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

```
const app=express();

//set the view engine as ejs

app.set('view engine','ejs')

//set the directory for the views

app.set('views',path.join(__dirname,'views'));

const products=[
   {   id: 1,
       title: 'product_1'
   },
   {   id: 2,
       title: 'product_2'
   },
   {   id: 3,
       title: 'product_3'
   },
   {   id: 4,
       title: 'product_4'
   }
]

app.get('/',(req,res)=>{
   res.render('home',{title: 'home',products:products})
});

app.get('/about',(req,res)=>{
   res.render('about',{title:'About page'})
})

const port=3090;

app.listen(port,()=>{
console.log(`sever is running on port numbet ${port}`)
})
```

header.js:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <header>EJS Template Demo</header>
</body>
</html>
```

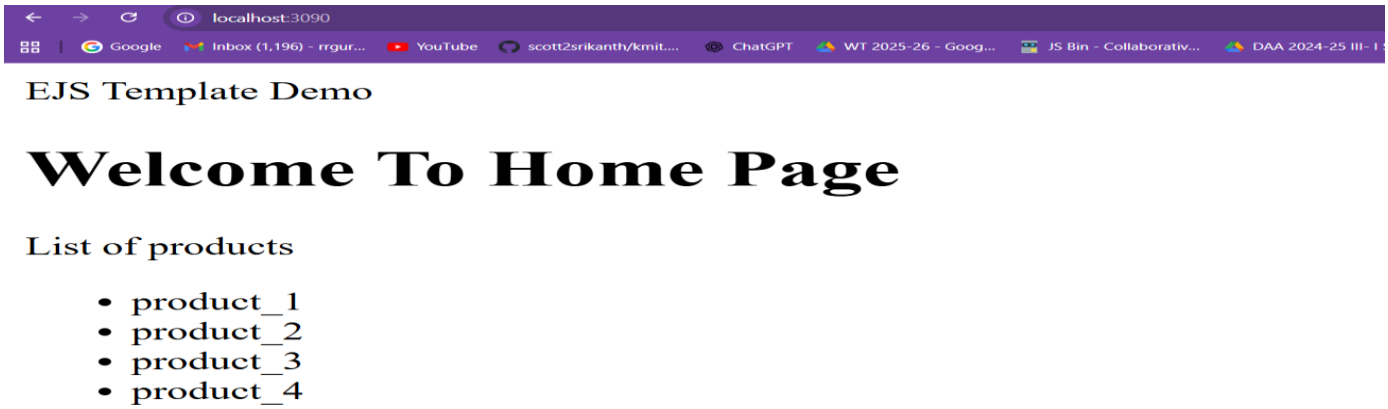
about.js:

```
<%- include('components/header.ejs') %>

<h1>this is our about page</h1>
```

home.js:

```
<%- include ('components/header.ejs')  %>
<h1>Welcome To Home Page</h1>
<lable>List of products</lable>

<ul> <% products.forEach(product=> {  %>
    <li><%= product.title %> </li>
<% })  %>
</ul>
```

Browser output: URL: http://localhost:3090/



Browser output: URL: http://localhost:3090/about

8. res.sendStatus()

Represents the status of the response, res. status defines the HTTP response code. If an unidentified status code is provided, the response body will just contain the code's numeric value.
Each code will represent a different response state.

- 5xx: Server error
- 4xx: Error in clients
- 3xx: Redirects
- 2xx: Success
- 1xx : Information

Syntax: res.status(statusCode)

Example:

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

app.get('/status', (req, res) => {
  res.sendStatus(404); // Sends: "Not Found"
});

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

Console Output: 'Server running at http://localhost:3000/status
URL: **http://localhost:3000/status**
Browser Output: **Not Found**

Express.Router:

The express.Router() function is used to create a new router object. This function is used when you want to create a new router object in your program to handle requests.

Syntax: **express.Router([options])**

Optional Parameters:

- **case-sensitive:** This enables case sensitivity.
- **mergeParams:** It preserves the req.params values from the parent router.
- **strict:** This enables strict routing.

Return Value:
This function returns the New Router Object.

Explanation and Example
Modular, mountable route controllers can be made using the **express.Router** class. A Router instance is frequently referred to as a "mini-app" because it is an entire middleware and navigation system.

The example that follows shows how to build a router as a module, load a middleware function, define a few routes, and mount the router module on a path in the main app.

We must first build a separate module, instantiate an instance of app.Router, and use module.exports to export the instance before we can use app.Router().

In the app subfolder, create a router file called "**birds.js**" and fill it with the following information:

```
const express = require('express')
const router = express.Router()

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
```

```
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Then, load the router module in the app:

```
const birds = require('./birds')

// ...

app.use('/birds', birds)
```

The app will now be able to handle requests to /birds and /birds/about, as well as call the timeLog middleware function that is specific to the route.

Example : Let's create a file named **index.js**

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

// Multiple routing
const router1 = express.Router();
const router2 = express.Router();
const router3 = express.Router();

router1.get('/user', function (req, res, next) {
   console.log("User Router Working");
   res.end();
});

router2.get('/admin', function (req, res, next) {
   console.log("Admin Router Working");
   res.end();
});

router2.get('/student', function (req, res, next) {
   console.log("Student Router Working");
   res.end();
});

app.use(router1);
app.use(router2);
app.use(router3);
```

```
app.listen(PORT, function (err) {
   if (err) console.log(err);
   console.log("Server listening on PORT", PORT);
});
```

Browser URL: Now make a GET request to

1. http://localhost:3000/user,
2. http://localhost:3000/admin, and
3. http://localhost:3000/student,

then you will see the following output on your console:

Output:

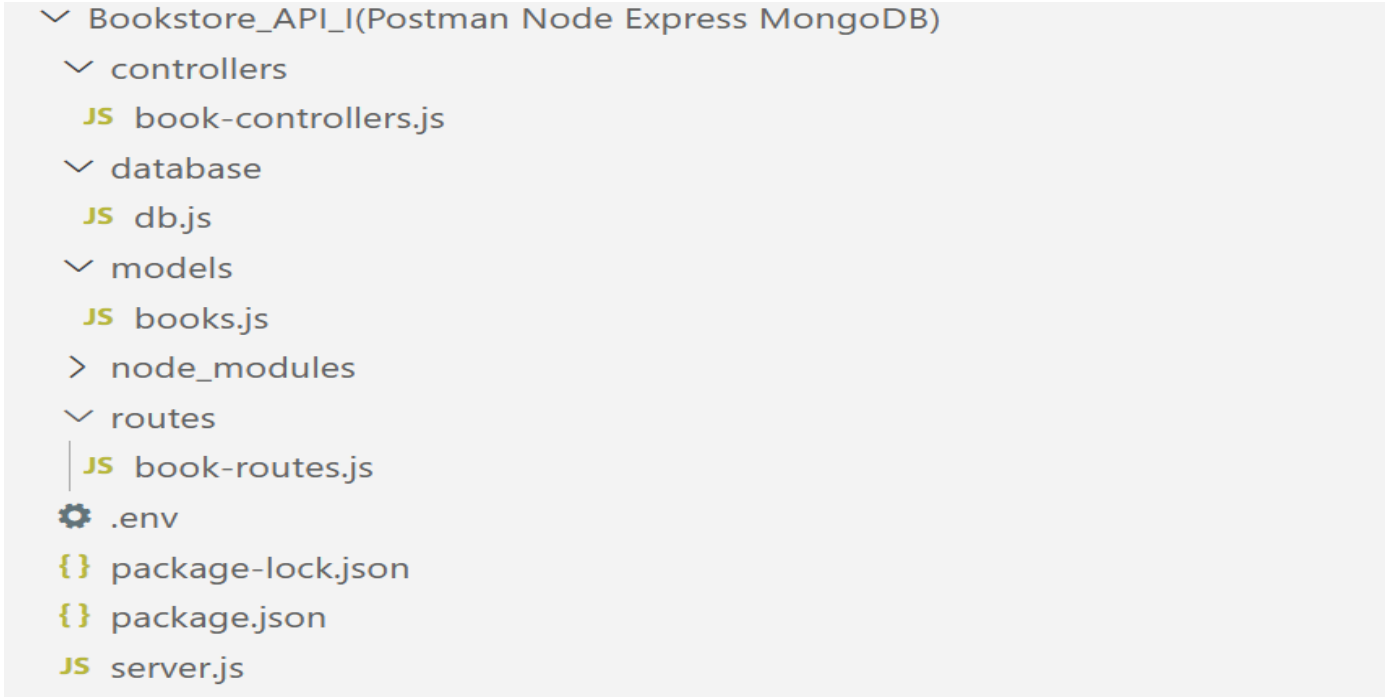
Server listening on PORT 3000
User Router Working
Admin Router Working
Student Router Working

Project-1: Book Store API: (Only Backend development) using post man (REST API)

You need to install below dependencies:

1. npm installation: npm init –y
2. install express: npm install express
3. dotenv installation: npm install dotenv
4. nodemon with devdependencies: npm install nodemon ---save—dev
5. mongoose installation : npm install mongoose

Project Structure: Note: Follow the same structure for Every Project



Setup Package.json file with dev dependencies:

```
{
  "name": "bookstore_api",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "server.js",
    "dev": "nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "dotenv": "^17.2.3",
    "express": "^5.1.0",
    "mongoose": "^8.19.2"
  },
  "devDependencies": {
    "nodemon": "^3.1.10"
  }
}
```

Note: in script we need to add below statements

```
    "start": "server.js",
    "dev": "nodemon server.js"
```

And

Change to server.js from index.js in "main": "server.js" (this is Optional) here in this project main javascript code written in server.js

---

Run project with command: npm run dev (autoexection)

Filename: db.js

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

const connectToDB= async()=>
{
    try{
        await mongoose.connect("mongodb://127.0.0.1:27017/BookStore")
        //here BookStore is the data base name
      onsole.log("mongoDB connected successfully!");
    }
    catch(error )
    {
        console.error("MongoDB Connection Failed", error)
        process.exit(1);
    }
};
module.exports=connectToDB;
```

Filename: server.js:

```
require('dotenv').config();
const express= require('express');
const connectToDB=require("./database/db.js");
const bookRoutes=require('./routes/book-routes.js')

const app=express();
const PORT=process.env.PORT || 3009;

//connect to database
connectToDB();

//middleware ->expess.json
app.use(express.json());

//routes here books is the collection name,it is going to create in mongoBD under
//BookStore Databse
app.use("/api/books",bookRoutes)
app.listen(PORT,()=>{
    console.log(`server is conected on port number${PORT}`)
})
```

Filename: book-routes.js

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

const{
    getAllBooks,
    getsingleBookId,
    addNewBook,
    updateBook,
    deletebook
   }=require('../controllers/book-controllers');

//create express router
const router= express.Router()

// All routes are related to books
router.get('/get', getAllBooks);          //  Handler added
router.get('/get/:id', getsingleBookId);  //  Handler added
router.post('/add', addNewBook);          //  Handler added
router.put('/update/:id', updateBook);    //  Handler added
router.delete('/delete/:id', deletebook); //  Handler added

module.exports=router;
```

Filename: book-model.js

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

//create schema(Model) for Book
const bookschema=new mongoose.Schema(
    {
        title: {
            type: String,
            required: [true,"Book title is required"],
            trim : true,
            maxLength: [100,"Book title can not be more than 100  characters"]
        },
        author: {
            type: String,
            required: [true,"author name is required"],
            trim : true
        },
        year:{
            type: Number,
            requireq: [true,'publication year is required'],
            min : [1000,'year must be atleast 1000'],
            max: [new Date().getFullYear(),'Year can not be in the future']
        },
        createdAt:{
            type: Date,
            default: Date.now
        }
    }
)
module.exports=mongoose.model("BOOK",bookschema);
```

Filename: book-controllers.js

```
const Book=require("../models/book-model");
//.. means "go up one folder level" – it refers to the parent directory.

const getAllBooks=async(req,res)=>{
    try{
        const allBooks=await Book.find({});
        if(allBooks?.length > 0)
    //The ? in allBooks?.length is called the optional chaining operator in JavaScript
(introduced in ES2020).
     //This is a safe way to access the property .length only if allBooks is not null or
undefined.
     //It's equivalent to: if (allBooks && allBooks.length > 0)
           {
            res.status(200).json(
                {
                    success: true,
                    message: 'List of Books Fetched successfully',
                    data: allBooks
```

```
                }
            )

        }
        else {
      res.status(404).json({
        success: false,
        message: 'No books found in the database'
      });
    }
    }
    catch(e){
        console.log(e);
        res.status(500).json(
            {
                sucess: false,
                message: 'something went wrong please try again'
            }
        )

    }
};

const getsingleBookId=async(req,res)=>{
    try{
            const getCurrentBookId=req.params.id;
            const bookDetailsById=await Book.findById(getCurrentBookId)
            //in postman url:
http://localhost:3009/api/books/get/68ffa7903d75fc5b9e39e13a
            //Here getting book based on id,we will take the id number from mongodb or it
is available in getallbooks route
            if(!bookDetailsById)
            {
                return res.status(400).json({
                    success: false,
                    message: 'Book with current is not found! please try with proper book
id '
                });
            }
            else {
                res.status(200).json({
                    success: true,
                    data: bookDetailsById
                })
            }
    }
    catch(e)
    {
        console.log(e);
        res.status(500).json({
            success: false,
            message: "something went wrong! please tyr again"
```

```
        })
    }
};

const addNewBook=async(req,res)=>{
    const newBookformData=req.body;
    const newlyCreatedBook=await Book.create(newBookformData);
    try{
        if(newBookformData)
        {
        res.status(201).json({
            success: true,
            message:"book added successfully",
            data: newlyCreatedBook
            })
        }
    }catch(e)
     {
        console.log(e);
        res.status(500).json(
            {
                success:false,
                message: 'something went wrong! please try agin'
            }
        )

    }
};

const updateBook=async(req,res)=>{
    try
    {
       const updatedBookdata=req.body;
       const getCurrentBookId=req.params.id;
       const updateBook=await Book.findByIdAndUpdate(getCurrentBookId,
                           updatedBookdata,
                           {
                               new: true
                           });
            if(!updateBook)

                {
                    res.status(404).json(
                        {
                            success: false,
                            message: 'Book is not found with this ID'
                        }
                    )
                }
                else{
                    res.status(200).json(
                        {
```

```
                            success: true,
                             message: 'Book is updated successfully' ,
                             data: updateBook
                        }
                    )
                }
    }catch(e)
    {
        console.log(e);
        res.status(500).json({
            success: false,
            Message:  'something went wrong! please try agin'
        })

    }
};

const deletebook=async(req,res)=>{
    try{
        const getCurrentBookId=req.params.id;
        const deletebook=await Book.findByIdAndDelete(getCurrentBookId);
        if(!deletebook)
        {
            res.status(404).json(
                {
                    success: false,
                    Message: 'Book is not Found with Id'
                }
            )
        }
        else{
            res.status(200).json(
                {
                    success: true,
                    data: deletebook
                }
            )
        }
    }
    catch(e)
    {
        console.log(e);
        res.status(500).json({
            success: false,
            message: "Something went wrong! Please try again"
        })

    }
};

module.exports={
    getAllBooks,
```

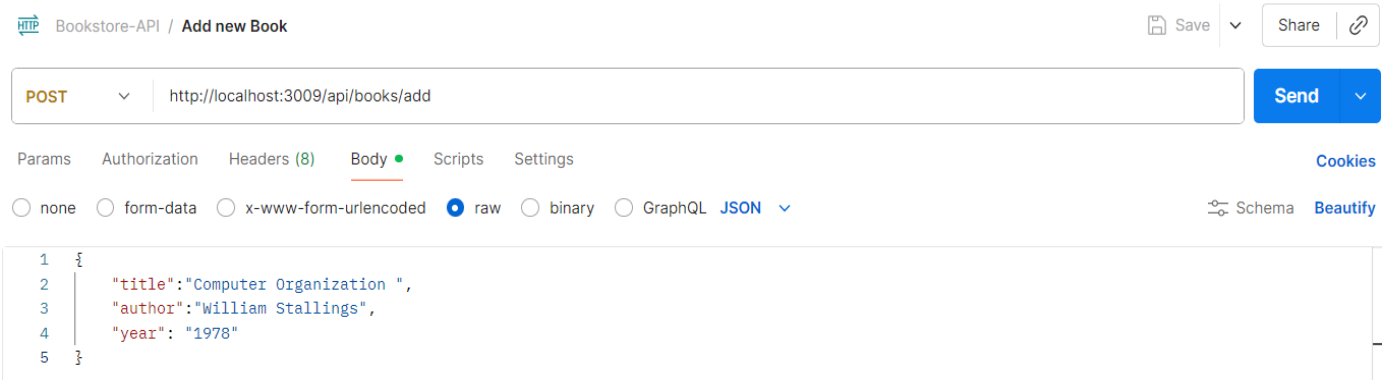
```
    getsingleBookId,
    addNewBook,
    updateBook,
    deletebook
};
```

Run Project: `npm run dev`

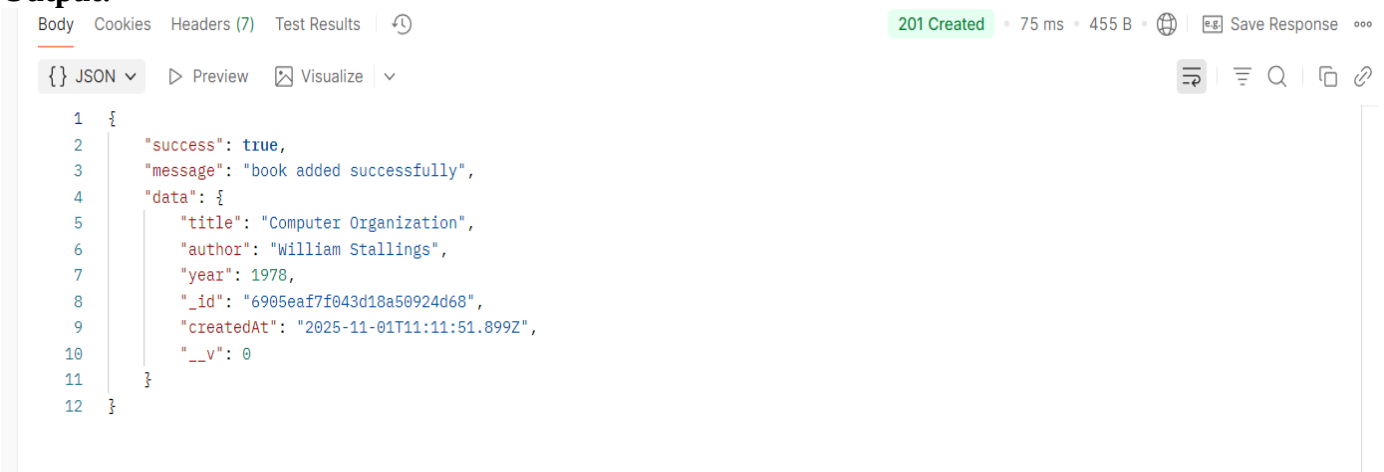
```
> bookstore_api@1.0.0 dev
> nodemon server.js
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
[dotenv@17.2.3] injecting env (0) from .env -- tip: 🔑 add access controls to secrets:
https://dotenvx.com/ops
server is conected on port number3009
mongoDB connected successfully!
```

Now goto postman: create collection for your project and create request for each route and save

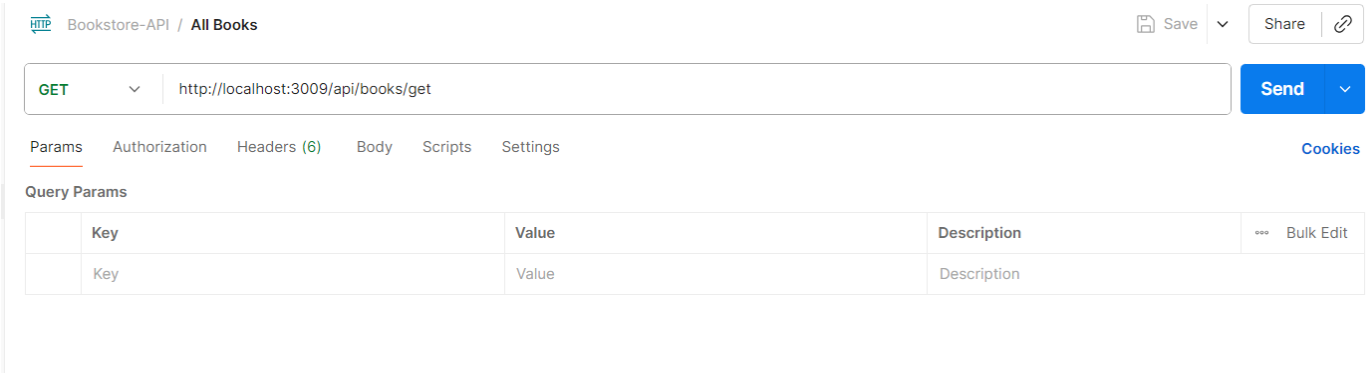
Step-1: add Book request name is Add new Book (REST API Creation)



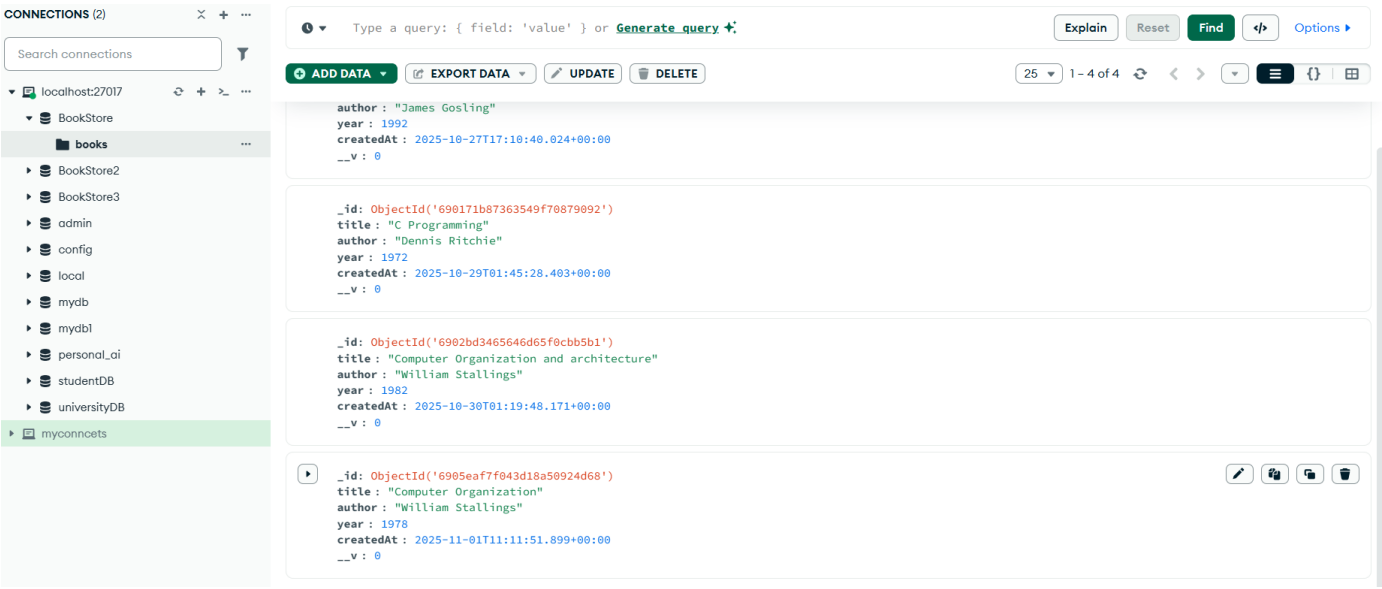
Output:



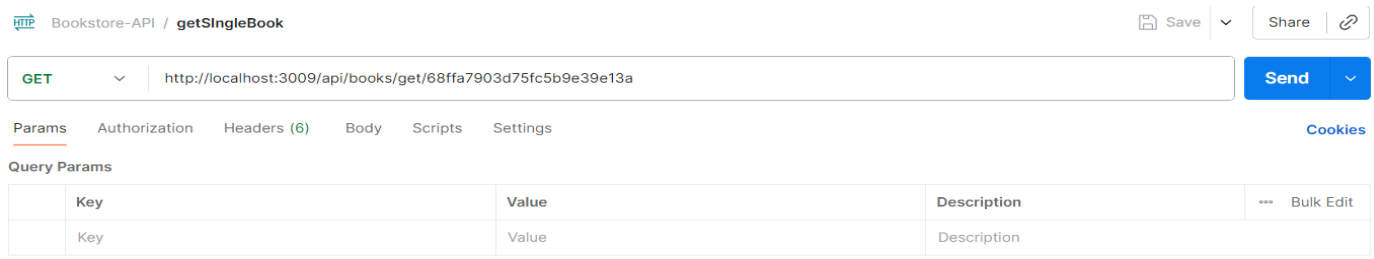
Step-2: get all books



Now goto mongoDB check the collection (previously added three books)



Step-3: get a single book (based id , here Book id which is taken from mongoDB collection



Output:

Step-4: delete a book based on id

Bookstore-API / Dele single book

Save Share

DELETE http://localhost:3009/api/books/delete/68ffa7903d75fc5b9e39e13a Send

Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Output:

Body Cookies Headers (7) Test Results 200 OK 7 ms 398 B Save Response

{} JSON Preview Visualize


```
{
    "success": true,
    "data": {
        "_id": "68ffa7903d75fc5b9e39e13a",
        "title": "java Book",
        "author": "James Gosling",
        "year": 1992,
        "createdAt": "2025-10-27T17:10:40.024Z",
        "__v": 0
    }
}
```


Now check in monogDB Collection:

Type a query: { field: 'value' } or Generate query Explain Reset Find Options

ADD DATA EXPORT DATA UPDATE DELETE 25 1 - 3 of 3


```
_id: ObjectId('690171b87363549f70879092')
title : "C Programming"
author : "Dennis Ritchie"
year : 1972
createdAt : 2025-10-29T01:45:28.403+00:00
__v : 0

_id: ObjectId('6902bd3465646d65f0cbb5b1')
title : "Computer Organization and architecture"
author : "William Stallings"
year : 1982
createdAt : 2025-10-30T01:19:48.171+00:00
__v : 0

_id: ObjectId('6905eaf7f043d18a50924d68')
title : "Computer Organization"
author : "William Stallings"
year : 1978
createdAt : 2025-11-01T11:11:51.899+00:00
__v : 0
```


Computer Organization and architecture

Step-5: update Book Details: based on ID

Bookstore-API / update book

Save Share

PUT http://localhost:3009/api/books/update/6902bd3465646d65f0cbb5b1 Send

Params Authorization Headers (8) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Schema Beautify

```
{
    "title":"Computer Organization and architecture ",
    "author":"William Stallings",
    "year": "1992"
}
```

Output:

Body Cookies Headers (7) Test Results 200 OK 20 ms 472 B Save Response

JSON Preview Visualize


```
{
    "success": true,
    "message": "Book is updated successfully",
    "data": {
        "_id": "6902bd3465646d65f0cbb5b1",
        "title": "Computer Organization and architecture",
        "author": "William Stallings",
        "year": 1992,
        "createdAt": "2025-10-30T01:19:48.171Z",
        "__v": 0
    }
}
```


Now check in monogDB Collection:

Type a query: { field: 'value' } or Generate query Explain Reset Find

ADD DATA EXPORT DATA UPDATE DELETE 25 1 - 3 of 3


```
_id: ObjectId('690171b87363549f70879092')
title : "C Programming"
author : "Dennis Ritchie"
year : 1972
createdAt : 2025-10-29T01:45:28.403+00:00
__v : 0
```

```
_id: ObjectId('6902bd3465646d65f0cbb5b1')
title : "Computer Organization and architecture"
author : "William Stallings"
year : 1992
createdAt : 2025-10-30T01:19:48.171+00:00
__v : 0
```

```
_id: ObjectId('6905eaf7f043d18a50924d68')
title : "Computer Organization"
author : "William Stallings"
year : 1978
createdAt : 2025-11-01T11:11:51.899+00:00
__v : 0
```

Project-2 : NodeJS with mongoDB

```
/*
Project: Connect MongoDB with Node.js
 Step 1: Create a New Folder

mkdir node-mongo-demo
cd node-mongo-demo

 Step 2: Initialize Node.js Project
npm init -y

This creates a default package.json file.

 Step 3: Install Mongoose
npm install mongoose

*/

// Import mongoose
import mongoose from "mongoose";

// MongoDB connection URI (local database)
const uri = "mongodb://127.0.0.1:27017/studentDB";
//127.0.0.1 is not shown here because it’s a virtual internal address (reserved by the
system).

/*
mongodb://127.0.0.1:27017/studentDB
|      |           |         |
|      |           |         └── Database name (studentDB)
|      |           └──────────── Port (27017 = default MongoDB port)
|      └──────────────────────── Host / IP (your computer)
└─────────────────────────────── Protocol / Driver name

*/


// Connect to MongoDB
//uri: Uniform Resource Identifier.
mongoose.connect(uri, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB connected successfully!"))
.catch((err) => console.error("Connection failed:", err));

// Create a schema (structure of data)
const studentSchema = new mongoose.Schema({
  name: String,
  age: Number,
  course: String
});
```

```
// Create a model (represents a collection)
const Student = mongoose.model("Student", studentSchema);

// Insert one document
const addStudent = async () => {
  const s1 = new Student({
    name: "Rakesh Reddy",
    age: 21,
    course: "B.Tech"
  });
/*  Explanation of above functionality
----------------------------------------
->const addStudent = async () => { ... }- This creates an asynchronous arrow function
named addStudent.

The word async means:
->Inside this function, we will perform operations that take time (like talking to
MongoDB).
->Because MongoDB operations (like .save()) are asynchronous, we must use await inside.

const s1 = new Student({ ... })
->Here we are creating a new document (record) in memory.
->Student is the model that was created earlier using:
    const Student = mongoose.model("Student", studentSchema);
That model represents your MongoDB collection (students).

->When you write:

new Student({...})
You are saying: “Hey Mongoose, make a new student document using this data.”
*/

  await s1.save();
  console.log("Student data saved!");
};

// Fetch all documents
const getStudents = async () => {
  const students = await Student.find();
  console.log("All Students:", students);
};

// Run both functions
addStudent().then(() => getStudents());
```

Explanation of above Code:

```
/*
| Step | Concept                | Description                                          |
| ---- | ---------------------- | ---------------------------------------------------- |
| 1    | **mongoose.connect()** | Connects Node.js to MongoDB                          |
| 2    | **Schema**             | Defines structure of data (like a table blueprint)   |
| 3    | **Model**              | A JavaScript class that interacts with the collection |
| 4    | **save()**             | Adds new document to MongoDB                         |
| 5    | **find()**             | Reads documents from MongoDB                         |

*/

/*
1.mongoose.connect(uri, {...})

This is the main function used to connect your Node.js application to your MongoDB
database.

It takes two arguments:  uri → the database connection string (where your MongoDB lives)
Example: "mongodb://127.0.0.1:27017/studentDB"

An options object → { useNewUrlParser: true, useUnifiedTopology: true }

2.useNewUrlParser: true
This tells Mongoose to use MongoDB's new connection string parser.
The old parser was deprecated (removed in newer MongoDB drivers).

 Why we need this:
The new parser understands more modern connection strings like:
mongodb+srv://username:password@cluster.mongodb.net/myDB
(used in MongoDB Atlas cloud)

⚙ If you don't include it, you might see a warning like:

DeprecationWarning: current URL string parser is deprecated

3. useUnifiedTopology: true

✅ Meaning:

This tells Mongoose to use MongoDB's new unified topology engine to manage connections.

 In simple words:

MongoDB connections are complex (they have servers, replica sets, monitoring threads,
etc.).

The old system had multiple internal engines to handle all that.

The new unified topology combines all those into one clean, modern system.
```

✅ Benefits:

Faster and more stable connections

Handles server failover automatically

Reduces memory leaks and warnings

No “open handle” issues when you close your app

⚙️ Without this option, you might see a warning like:

DeprecationWarning: current Server Discovery and Monitoring engine is deprecated

*/

/*
What’s Happening Here: "Student" → is called the model name.
studentSchema → is the schema (the structure/blueprint of your documents).

When you define this model, Mongoose automatically creates (or connects to) a MongoDB collection — but it changes the name slightly.

🧠 Mongoose Naming Rule
Mongoose automatically lowercases and pluralizes your model name to form the collection name in MongoDB.

So:

Model Name (in code)	Actual Collection (in MongoDB)
"Student"	students
"User"	users
"Person"	people
"City"	cities
"Category"	categories

Summary:

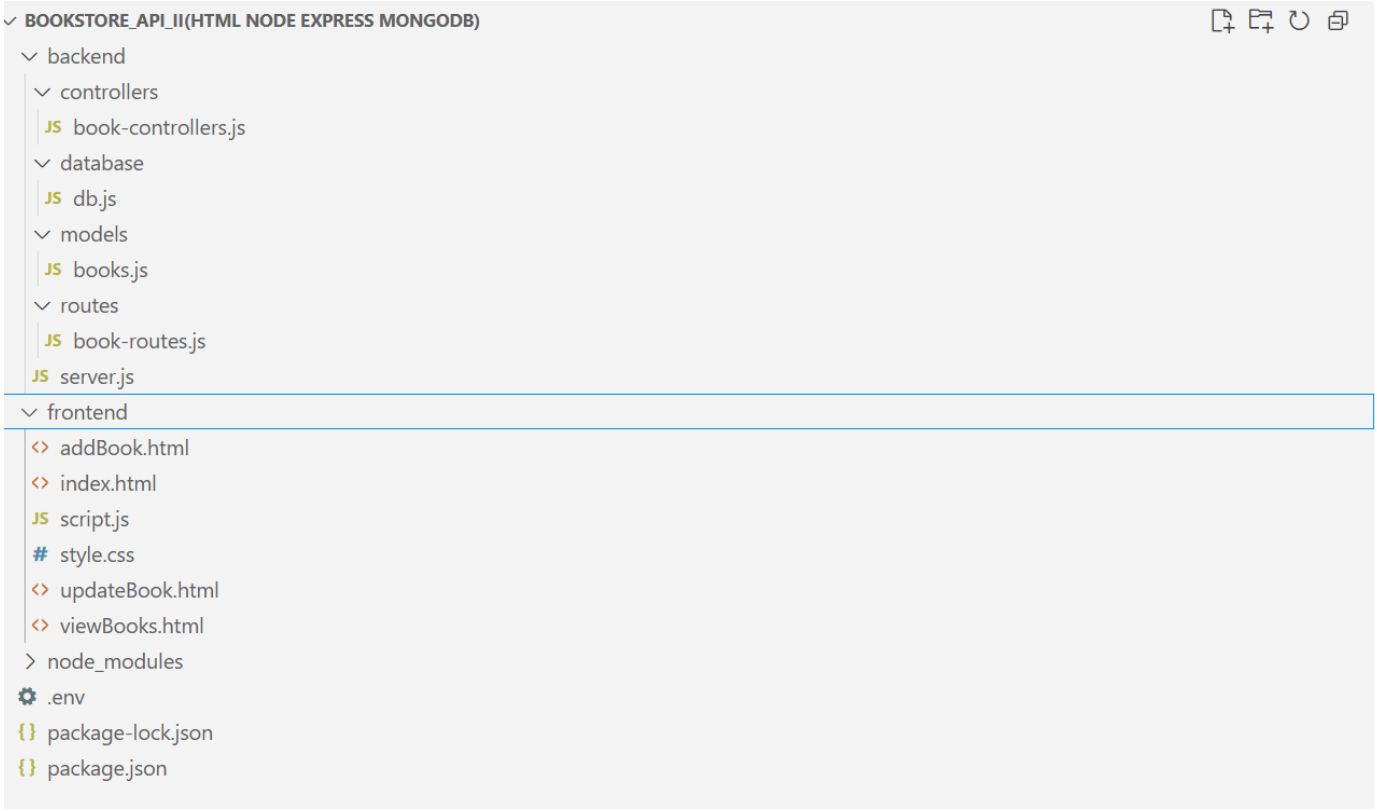
Node.js + Mongoose → good for small scripts, learning, or backend database logic only.

Express.js + Mongoose → used for real applications, where users send data via HTTP requests.
*/

Project-3: Book store API (add books, get all books, update a book details, delete a book, get single book)

Using: (Frontend (Html, CSS, Bootstrap, JS) ,Backend(Node.js,Express.js and mongo DB)

Project Structure:



Frontend files:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Book Management</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <header>
    <h1>📚 Book Management System</h1>
    <nav>
      <a href="index.html" class="active">Home</a>
      <a href="addBook.html">➕ Add Book</a>
    </nav>
  </header>

  <main>
    <section id="books-section">
      <h2>Available Books</h2>
      <div id="books-container" class="book-grid"></div>
    </section>
```

```
  </main>

  <footer>
    <p>© 2025 Book Manager | Created by Rakesh</p>
  </footer>

  <script src="script.js"></script>
</body>
</html>
```

style.css

```
/* Global Styles */
* {
  box-sizing: border-box;
  font-family: 'Poppins', sans-serif;
  margin: 0;
  padding: 0;
}

body {
  background-color: #f4f6f8;
  color: #333;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
}

/* Header */
header {
  background: linear-gradient(90deg, #0078d7, #00b4db);
  color: white;
  padding: 1rem 2rem;
  text-align: center;
  box-shadow: 0 4px 6px rgba(0,0,0,0.1);
}

header h1 {
  margin-bottom: 0.5rem;
}

nav a {
  color: white;
  text-decoration: none;
  margin: 0 1rem;
  font-weight: 500;
}

nav a:hover,
nav a.active {
  text-decoration: underline;
}
```

```css
/* Main Content */
main {
  flex: 1;
  padding: 2rem;
}

/* Book Grid */
.book-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(260px, 1fr));
  gap: 1.5rem;
  margin-top: 1.5rem;
}

/* Book Card */
.book-card {
  background: white;
  border-radius: 10px;
  padding: 1.5rem;
  box-shadow: 0 2px 6px rgba(0,0,0,0.1);
  transition: transform 0.2s, box-shadow 0.2s;
}

.book-card:hover {
  transform: translateY(-4px);
  box-shadow: 0 4px 12px rgba(0,0,0,0.15);
}

.book-card h3 {
  color: #0078d7;
  margin-bottom: 0.5rem;
}

.book-card p {
  margin-bottom: 0.3rem;
}

.book-card button {
  border: none;
  padding: 0.5rem 0.8rem;
  border-radius: 6px;
  cursor: pointer;
  font-weight: 500;
  margin-right: 0.5rem;
  transition: 0.2s;
}

.book-card button:first-of-type {
  background-color: #ff4d4d;
  color: white;
}
```

```
.book-card button:last-of-type {
  background-color: #0078d7;
  color: white;
}

.book-card button:hover {
  opacity: 0.85;
}

/* Forms */
.book-form {
  background: white;
  max-width: 400px;
  margin: 2rem auto;
  padding: 2rem;
  border-radius: 10px;
  box-shadow: 0 2px 8px rgba(0,0,0,0.1);
}

.book-form label {
  display: block;
  margin-bottom: 0.4rem;
  font-weight: 600;
}

.book-form input {
  width: 100%;
  padding: 0.6rem;
  margin-bottom: 1rem;
  border-radius: 6px;
  border: 1px solid #ccc;
}

.btn {
  background-color: #0078d7;
  color: white;
  padding: 0.7rem;
  width: 100%;
  border: none;
  border-radius: 8px;
  font-weight: 600;
  cursor: pointer;
  transition: background 0.3s;
}

.btn:hover {
  background-color: #005fa3;
}

/* Footer */
footer {
  text-align: center;
```

```
  background: #222;
  color: #ddd;
  padding: 1rem 0;
  font-size: 0.9rem;
}
/* ===== Update Book Page Styling ===== */

.form-section {
  display: flex;
  flex-direction: column;
  align-items: center;
  margin-top: 3rem;
}

.form-section h2 {
  color: #0078d7;
  margin-bottom: 1.5rem;
  font-size: 1.8rem;
  text-align: center;
}

/* Stylish Form */
.book-form {
  background: #ffffff;
  width: 100%;
  max-width: 450px;
  padding: 2rem;
  border-radius: 12px;
  box-shadow: 0 4px 15px rgba(0,0,0,0.1);
  display: flex;
  flex-direction: column;
  gap: 1rem;
}

.book-form label {
  font-weight: 600;
  margin-bottom: 0.3rem;
  color: #333;
}

.book-form input {
  padding: 0.7rem;
  border: 1px solid #ccc;
  border-radius: 6px;
  outline: none;
  font-size: 1rem;
  transition: border-color 0.3s ease;
}

.book-form input:focus {
  border-color: #0078d7;
  box-shadow: 0 0 5px rgba(0,120,215,0.3);
```

```
}

/* Submit Button */
.btn {
  background: linear-gradient(90deg, #0078d7, #00b4db);
  color: white;
  font-weight: 600;
  padding: 0.8rem;
  border: none;
  border-radius: 8px;
  cursor: pointer;
  transition: transform 0.2s ease, box-shadow 0.2s ease;
}

.btn:hover {
  transform: translateY(-2px);
  box-shadow: 0 3px 10px rgba(0,0,0,0.2);
  opacity: 0.95;
}

/* Footer */
footer {
  background: #222;
  color: #ddd;
  padding: 1rem;
  text-align: center;
  font-size: 0.9rem;
  margin-top: 3rem;
}
```

addBook.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Add Book</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <header>
    <h1>➕ Add a New Book</h1>
    <nav>
      <a href="index.html">🏠 Home</a>
    </nav>
  </header>

  <main>
    <form id="addBookForm" class="book-form">
      <label>Title:</label>
      <input type="text" id="title" required />
```

```
      <label>Author:</label>
      <input type="text" id="author" required />

      <label>Year:</label>
      <input type="number" id="year" required />

      <button type="submit" class="btn">Add Book</button>
    </form>
  </main>

  <footer>
    <p>© 2025 Book Manager</p>
  </footer>

  <script>
    document.getElementById("addBookForm").addEventListener("submit", async (e) => {
      e.preventDefault();

      const book = {
        title: document.getElementById("title").value,
        author: document.getElementById("author").value,
        year: document.getElementById("year").value,
      };

      const res = await fetch("/api/books/add", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(book),
      });

      const result = await res.json();

      if (result.success) {
        alert("✅ Book added successfully!");
        window.location.href = "index.html";
      } else {
        alert("❌ Failed to add book!");
      }
    });
  </script>
</body>
</html>
```

updateBook.html:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Update Book</title>
```

```
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <header>
    <h1>📖 Update Book Details</h1>
    <nav>
      <a href="index.html">🏠 Home</a>
      <a href="addBook.html">➕ Add Book</a>
    </nav>
  </header>

  <main>
    <section class="form-section">
      <h2>Edit Book Information</h2>
      <form id="updateBookForm" class="book-form">
        <label for="title">Title:</label>
        <input type="text" id="title" required />

        <label for="author">Author:</label>
        <input type="text" id="author" required />

        <label for="year">Year:</label>
        <input type="number" id="year" required />

        <button type="submit" class="btn">Update Book</button>
      </form>
    </section>
  </main>

  <footer>
    <p>© 2025 Book Manager | All Rights Reserved</p>
  </footer>

  <script>
    const urlParams = new URLSearchParams(window.location.search);
    const bookId = urlParams.get("id");

    // Fetch the book details by ID
    async function loadBook() {
      try {
        const res = await fetch(`/api/books/get/${bookId}`);
        const result = await res.json();

        if (result.success && result.data) {
          const book = result.data;
          document.getElementById("title").value = book.title;
          document.getElementById("author").value = book.author;
          document.getElementById("year").value = book.year;
        } else {
          alert("Book not found!");
        }
      } catch (err) {
```

```
        console.error("Error fetching book:", err);
        alert("Error fetching book details.");
      }
    }

    // Handle the update form
    document.getElementById("updateBookForm").addEventListener("submit", async (e) => {
      e.preventDefault();

      const updatedBook = {
        title: document.getElementById("title").value,
        author: document.getElementById("author").value,
        year: document.getElementById("year").value,
      };

      try {
        const res = await fetch(`/api/books/update/${bookId}`, {
          method: "PUT",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(updatedBook),
        });

        const result = await res.json();

        if (result.success) {
          alert("✅ Book updated successfully!");
          window.location.href = "index.html";
        } else {
          alert("❌ Failed to update book.");
        }
      } catch (err) {
        console.error("Error updating book:", err);
        alert("Error while updating book.");
      }
    });

    loadBook();
  </script>
</body>
</html>
```

viewBooks.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>All Books</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h2>All Books in Store</h2>
  <div id="booksContainer"></div>
```

```
  <a href="addBook.html">➕ Add New Book</a>
  <script src="script.js"></script>
</body>
</html>
```

script.js:

```
async function fetchBooks() {
  const res = await fetch("/api/books/get");
  const data = await res.json();

  const container = document.getElementById("books-container");
  container.innerHTML = "";

  if (!data.success || data.data.length === 0) {
    container.innerHTML = "<p>No books found.</p>";
    return;
  }

  data.data.forEach((book) => {
    const div = document.createElement("div");
    div.classList.add("book-card");
    div.innerHTML = `
      <h3>${book.title}</h3>
      <p><strong>Author:</strong> ${book.author}</p>
      <p><strong>Year:</strong> ${book.year}</p>
      <button onclick="deleteBook('${book._id}')">🗑 Delete</button>
      <button onclick="window.location.href='updateBook.html?id=${book._id}'">✏
Update</button>
    `;
    container.appendChild(div);
  });
}

async function deleteBook(id) {
  if (confirm("Are you sure you want to delete this book?")) {
    const res = await fetch(`/api/books/delete/${id}`, { method: "DELETE" });
    const data = await res.json();
    alert(data.message || "Deleted");
    fetchBooks();
  }
}

fetchBooks();
```

Backend File:

.env

```
PORT=3009
MONGO_URI=mongodb://127.0.0.1:27017/BookStore_II
```

db.js:

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

const connectToDB= async()=>
{
    try{
        await mongoose.connect("mongodb://127.0.0.1:27017/BookStore2")
        console.log("mongoDB connected successfully!");
    }
    catch(error )
    {
        console.error("MongoDB Connection Failed", error)
        process.exit(1);
    }
};
module.exports=connectToDB;
```

server.js

```
const express = require("express");
const path = require("path");
const dotenv = require("dotenv");
const connectDB = require("./database/db");
const bookRoutes = require("./routes/book-routes");

dotenv.config();
const app = express();

app.use(express.json());

// ✅ Connect to MongoDB
connectDB();

// ✅ Serve static frontend files
app.use(express.static(path.join(__dirname, "../frontend")));

// ✅ API routes
app.use("/api/books", bookRoutes);

// ✅ Default route to serve index.html
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../frontend", "index.html"));
});

// ✅ Start the server
const PORT = process.env.PORT || 3009;
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
```

book-routes.js

```
const express=require('express')
const{
    getAllBooks,
    getsingleBookId,
    addNewBook,
    updateBook,
    deletebook
   }=require('../controllers/book-controllers');

//create express router
const router= express.Router()

// All routes are related to books
router.get('/get', getAllBooks);           //  Handler added
router.get('/get/:id', getsingleBookId);   //  Handler added
router.post('/add', addNewBook);           //  Handler added
router.put('/update/:id', updateBook);     //  Handler added
router.delete('/delete/:id', deletebook); //  Handler added

module.exports=router;
```

book-model.js

```
const mongoose=require('mongoose');
const bookschema=new mongoose.Schema(
    {
        title: {
            type: String,
            required: [true,"Book title is required"],
            trim : true,
            maxLength: [100,"Book title can not be more than 100  characters"]
        },
        author: {
            type: String,
            required: [true,"author name is required"],
            trim : true
        },
        year:{
            type: Number,
            requireq: [true,'publication year is required'],
            min : [1000,'year must be atleast 1000'],
            max: [new Date().getFullYear(),'Year can not be in the future']
        },
        createdAt:{
            type: Date,
            default: Date.now
        }
    }
)
module.exports=mongoose.model("BOOK",bookschema);
```

book-controllers.js

```
const Book=require("../models/book-model");
//.. means "go up one folder level" – it refers to the parent directory.

const getAllBooks=async(req,res)=>{
    try{
        const allBooks=await Book.find({});
        if(allBooks?.length > 0)
    //The ? in allBooks?.length is called the optional chaining operator in JavaScript
(introduced in ES2020).
     //This is a safe way to access the property .length only if allBooks is not null or
undefined.
     //It's equivalent to: if (allBooks && allBooks.length > 0)
           {
            res.status(200).json(
                {
                    success: true,
                    message: 'List of Books Fetched successfully',
                    data: allBooks
                }
            )

        }
        else {
      res.status(404).json({
        success: false,
        message: 'No books found in the database'
      });
    }
    }
    catch(e){
        console.log(e);
        res.status(500).json(
            {
                sucess: false,
                message: 'something went wrong please try again'
            }
        )

    }
};

const getsingleBookId=async(req,res)=>{
    try{
            const getCurrentBookId=req.params.id;
            const bookDetailsById=await Book.findById(getCurrentBookId)
            //in postman url:
http://localhost:3009/api/books/get/68ffa7903d75fc5b9e39e13a
            //Here getting book based on id,we will take the id number from mongodb or it
is available in getallbooks route
            if(!bookDetailsById)
            {
```

```
                return res.status(400).json({
                    success: false,
                    message: 'Book with current is not found! please try with proper book
id '
                });
            }
            else {
                res.status(200).json({
                    success: true,
                    data: bookDetailsById
                })
            }
    }
    catch(e)
    {
        console.log(e);
        res.status(500).json({
            success: false,
            message: "something went wrong! please tyr again"
        })
    }
};

const addNewBook=async(req,res)=>{
    const newBookformData=req.body;
    const newlyCreatedBook=await Book.create(newBookformData);
    try{
        if(newBookformData)
        {
        res.status(201).json({
            success: true,
            message:"book added successfully",
            data: newlyCreatedBook
            })
        }
    }catch(e)
     {
        console.log(e);
        res.status(500).json(
            {
                success:false,
                message: 'something went wrong! please try agin'
            }
        )

    }
};

// ✅ Update Book by ID
const updateBook = async (req, res) => {
  try {
    const { id } = req.params;
```

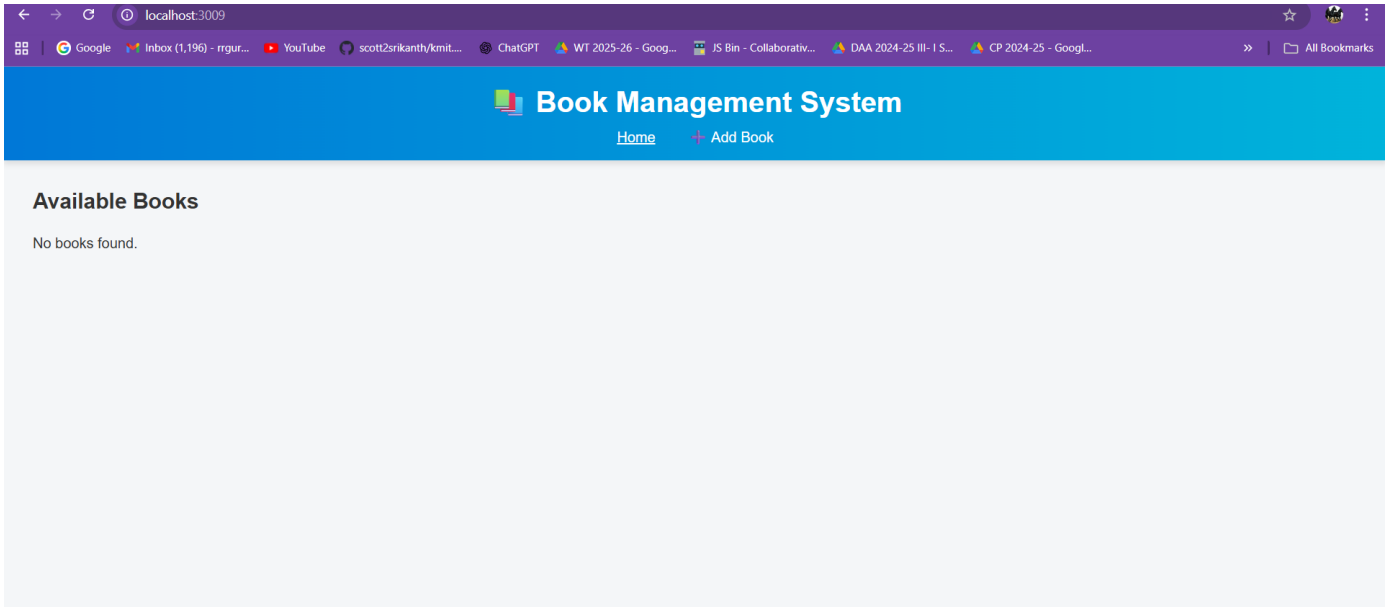
```
    const updatedData = req.body;

    const book = await Book.findByIdAndUpdate(id, updatedData, {
      new: true, // returns the updated document
    });

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

    res.status(200).json(
        {
            success: true,
             message: "Book updated successfully",
             data: book
            }
        );
  } catch (err)
  {
    console.error("Update error:", err);
    res.status(500).json(
        {
            success: false,
            message: "Error updating book"
        });
  }
};


const deletebook=async(req,res)=>{
    try{
        const getCurrentBookId=req.params.id;
        const deletebook=await Book.findByIdAndDelete(getCurrentBookId);
        if(!deletebook)
        {
            res.status(404).json(
                {
                    success: false,
                    Message: 'Book is not Found with Id'
                }
            )
        }
        else{
            res.status(200).json(
                {
                    success: true,
                    data: deletebook
```

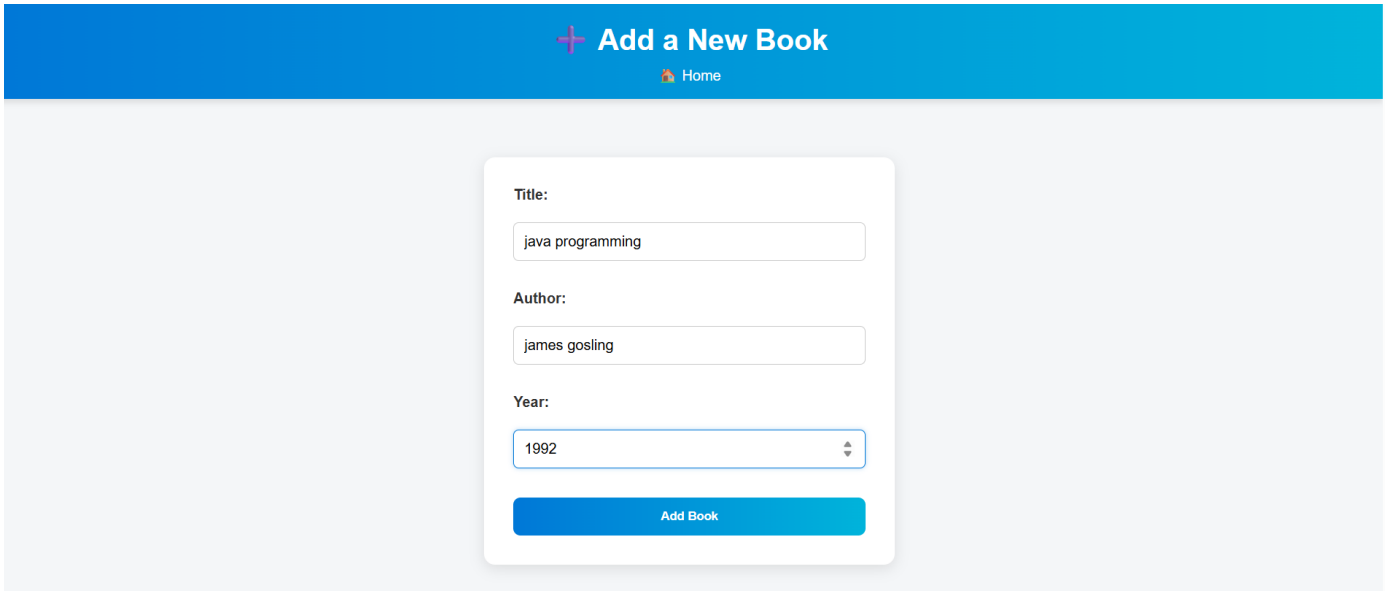
```
                }
            )
        }
    }
    catch(e)
    {
        console.log(e);
        res.status(500).json({
            success: false,
            message: "Something went wrong! Please try again"
        })

    }
};

module.exports={
    getAllBooks,
    getsingleBookId,
    addNewBook,
    updateBook,
    deletebook
};
```

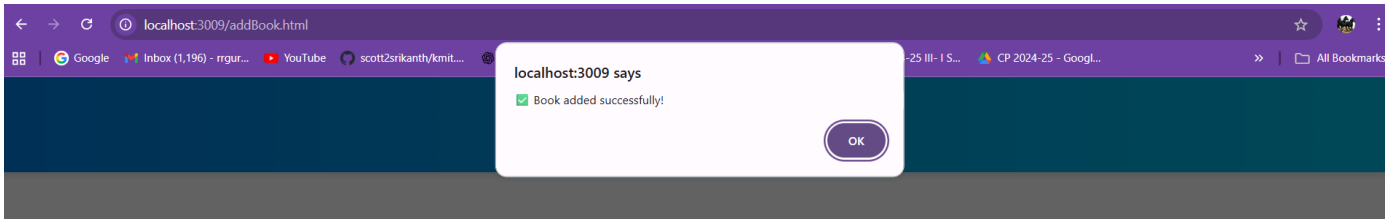
Run: D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\ExpressJS\Projects\Bookstore_API_II(Html Node Express MongoDB)\backend> npm run dev

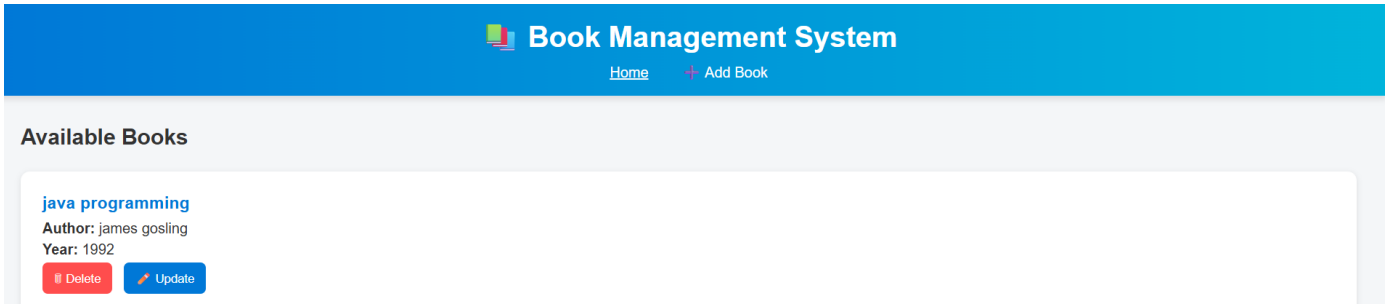
```
> bookstore_api-with-frontend@1.0.0 dev

> nodemon backend/server.js

[nodemon] 3.1.10

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

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

[nodemon] watching extensions: js,mjs,cjs,json

[nodemon] starting `node backend/server.js`

[dotenv@17.2.3] injecting env (2) from .env -- tip: 📡 add observability to secrets: https://dotenvx.com/ops

Server running on http://localhost:3009

mongoDB connected successfully!Browser url: localhost
```

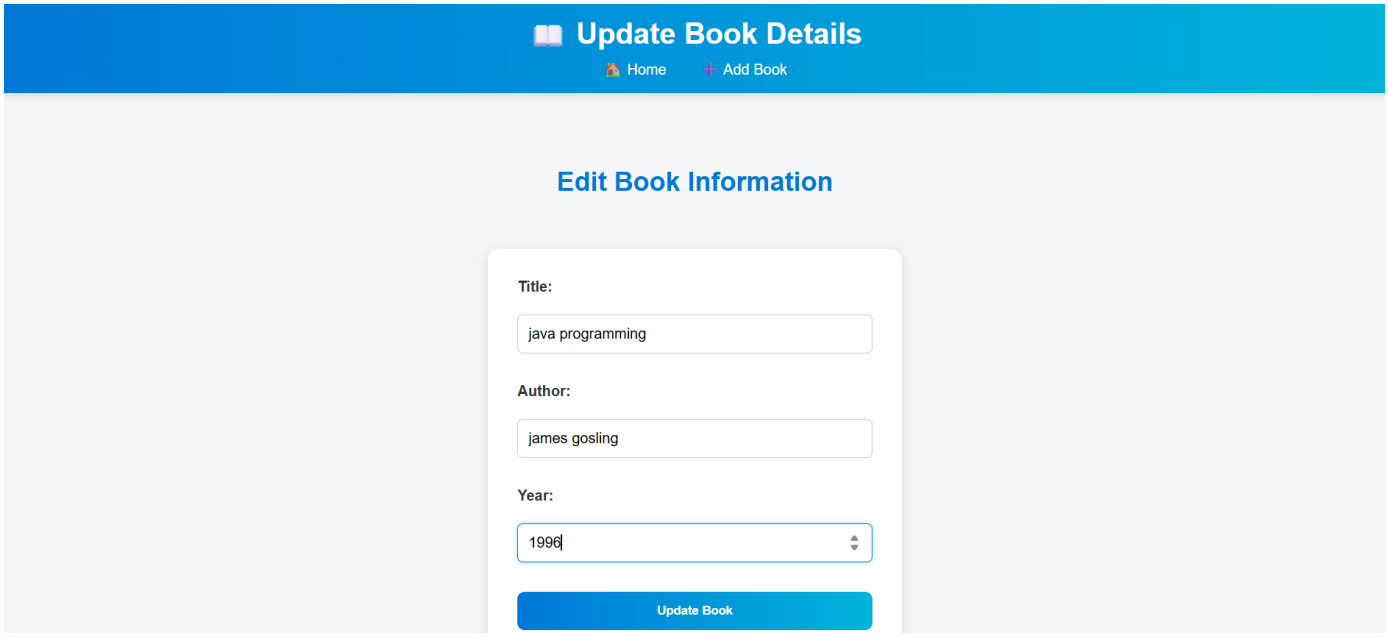
Browser URL:

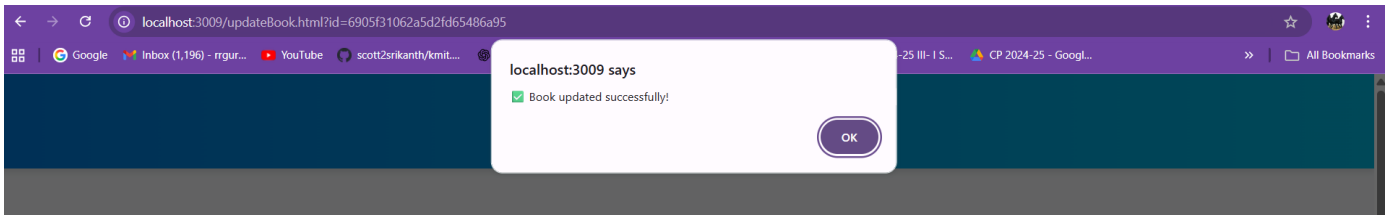

localhost:3009
Book Management System
Home
Add Book
Available Books
No books found.

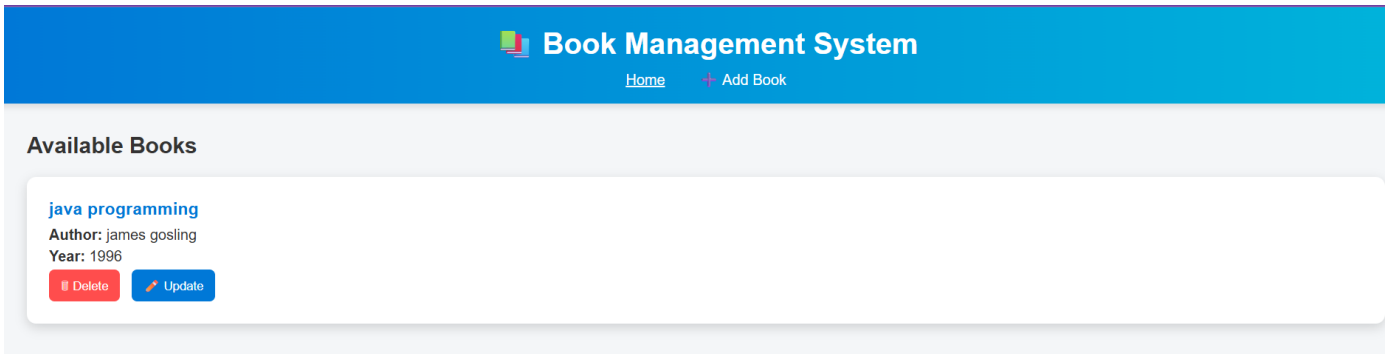

Add a New Book
Home
Title:
java programming
Author:
james gosling
Year:
1992
Add Book


localhost:3009/addBook.html
localhost:3009 says
Book added successfully!
OK

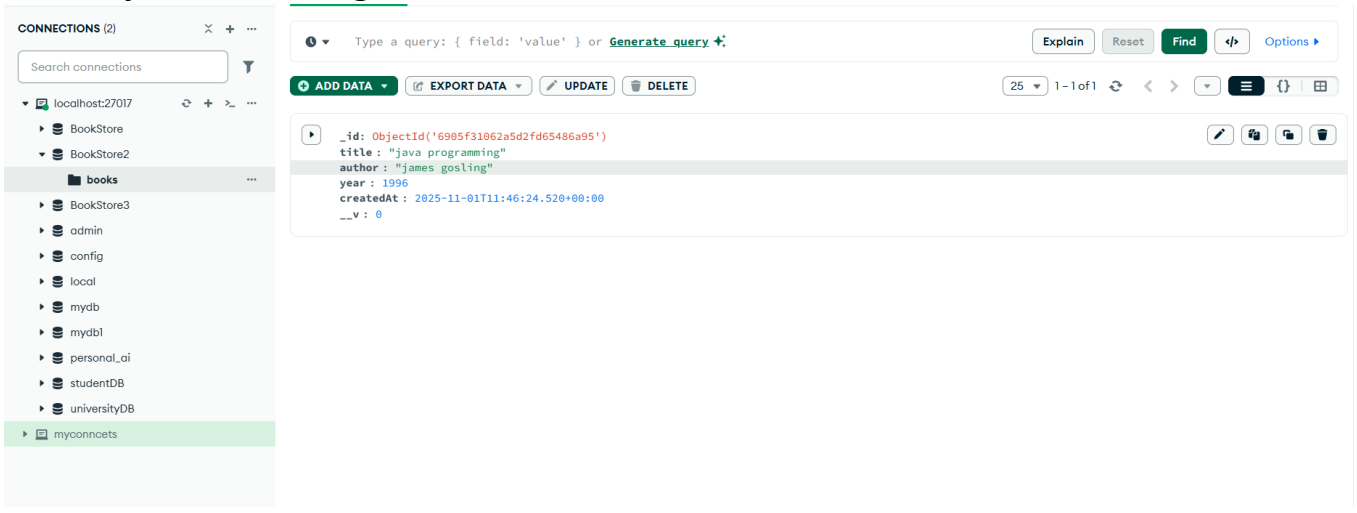

Book Management System
Home
Add Book
Available Books
java programming
Author: james gosling
Year: 1992
Delete
Update

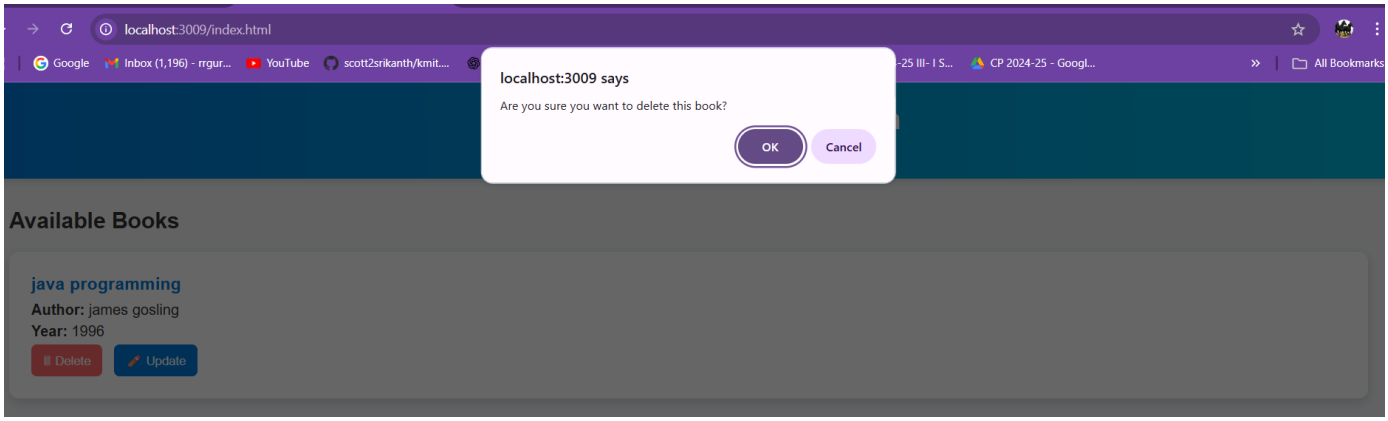


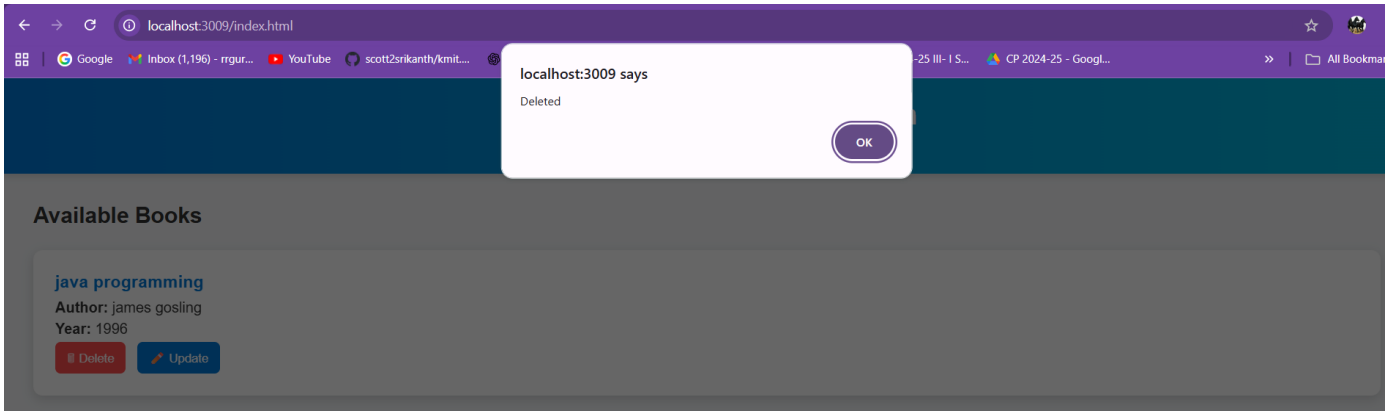


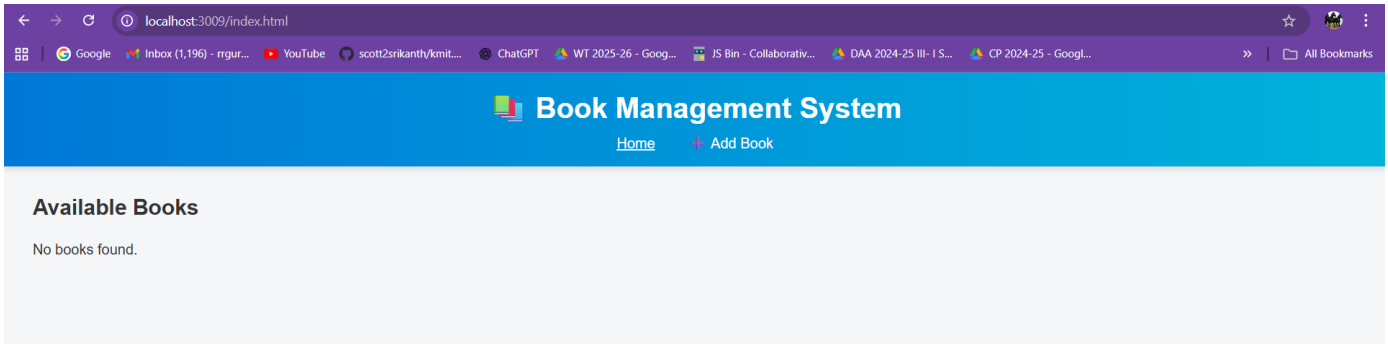


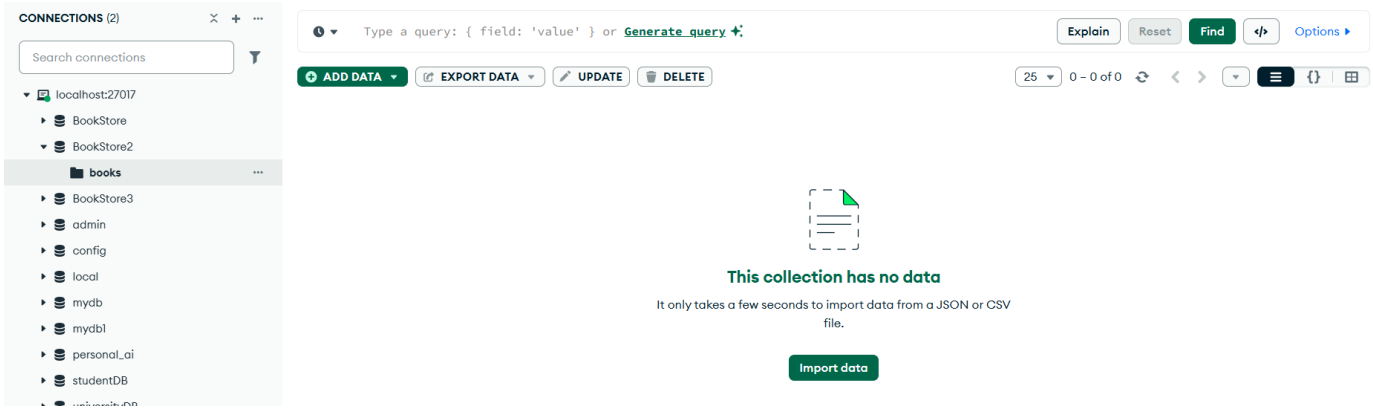
Parallely check In mongoDB also




localhost:3009/index.html
localhost:3009 says
Are you sure you want to delete this book?
OK
Cancel
Available Books
java programming
Author: james gosling
Year: 1996
Delete
Update


localhost:3009/index.html
localhost:3009 says
Deleted
OK
Available Books
java programming
Author: james gosling
Year: 1996
Delete
Update


localhost:3009/index.html
Book Management System
Home
Add Book
Available Books
No books found.


CONNECTIONS (2)
Search connections
localhost:27017
BookStore
BookStore2
books
BookStore3
admin
config
local
mydb
mydb1
personal_ai
studentDB
Type a query: { field: 'value' } or Generate query
Explain
Reset
Find
Options
ADD DATA
EXPORT DATA
UPDATE
DELETE
This collection has no data
It only takes a few seconds to import data from a JSON or CSV file.
Import data