

**Batch:D-2**

**Roll No.:16010123325**

**Experiment / assignment / tutorial No.\_7\_\_**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

**Title: Create a RESTful API server in Express and Node.js. Implementation + Testing application using postman/Thunderclient**

**AIM:** Create a RESTful API server in Express and Node.js. Implementation + Testing application using postman/Thunderclient

**Problem Definition:** To develop and test a RESTful API using Node.js and Express.js that performs CRUD (Create, Read, Update, Delete) operations on student data. The API should handle JSON requests and responses, and be tested using Thunder Client to verify proper backend functionality.

**Resources used:**

- Software:
  - Node.js
  - Visual Studio Code
  - Thunder Client (VS Code Extension)
- Libraries:
  - Express.js
  - Body-Parser (for JSON parsing)

---

**Expected OUTCOME of Experiment:**

Successful implementation of a RESTful API that performs CRUD operations using Express.js and Node.js, with verification through Thunder Client showing correct HTTP responses and JSON data for each operation.

**CO 3: Test the concepts and components of various front-end, back-end web app development technologies & frameworks using web development tools.**

---

**Books/ Journals/ Websites referred:**

1. Official Express.js Documentation – <https://expressjs.com>
2. MDN Web Docs – REST API Basics and HTTP Methods

**Pre Lab/ Prior Concepts:**

Thunder Client is a lightweight REST API client available as an extension inside VS Code. It allows developers to test their backend APIs by sending HTTP requests (GET, POST, PUT, DELETE) and receiving responses directly in JSON format. It simplifies testing without needing an external tool like Postman.

**Methodology:**

1. Initialize a Node.js project using `npm init -y`.
2. Install Express.js using `npm install express`.
3. Create a `server.js` file to define CRUD (Create, Read, Update, Delete) routes.
4. Run the server using `node server.js`.
5. Use Thunder Client to test each API endpoint.

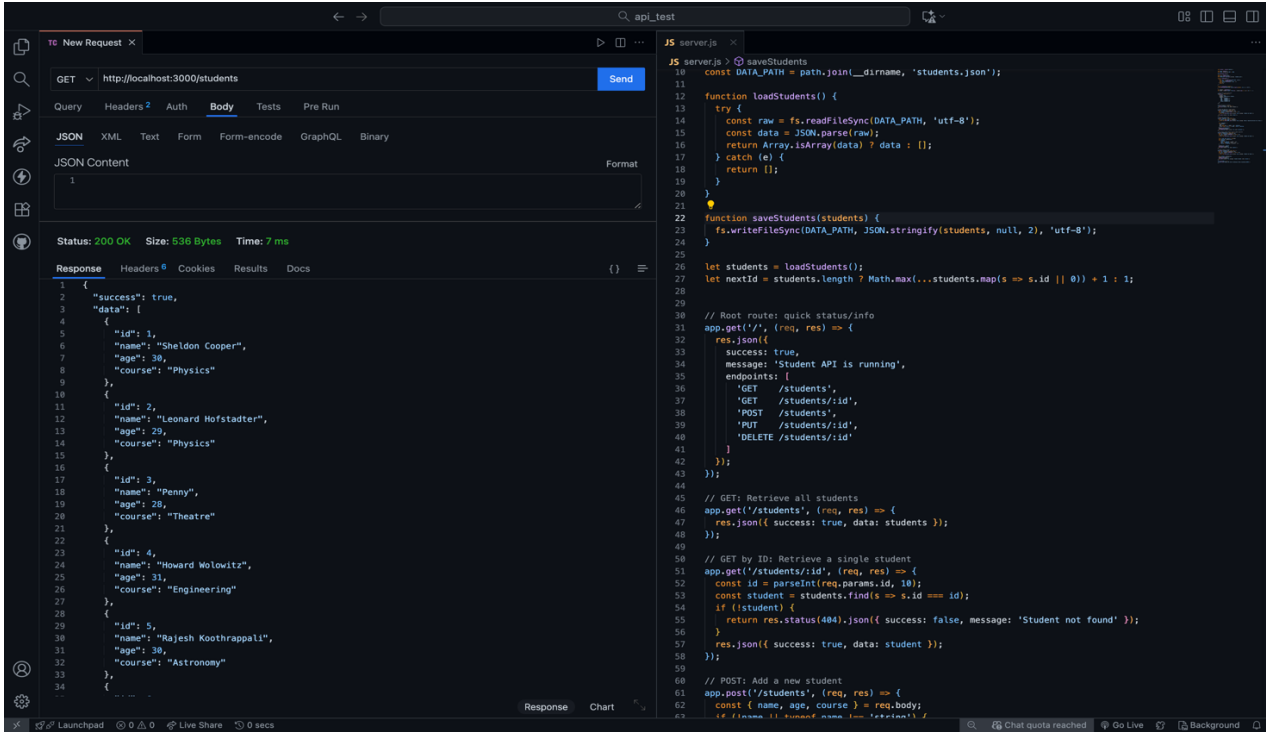
**Implementation Details:**

The implementation includes defining RESTful endpoints using Express.js. Each endpoint corresponds to a CRUD operation on a simple student dataset.

**Steps for execution:**

1. Start the server using the command: `node server.js`
2. Open Thunder Client from the VS Code sidebar.
3. Perform the following operations:
  - GET: Retrieve all students.
  - POST: Add a new student.
  - GET by ID: Retrieve a single student.
  - PUT: Update student details.
  - DELETE: Remove a student.

## GET Request – Fetch All Students



The screenshot shows a REST client interface in VS Code. The request is a GET to `http://localhost:3000/students`. The response is a 200 OK status with a JSON body containing an array of student objects. The server code on the right shows the `loadStudents` function and the `app.get('/students')` route handler.

```

// REST Client Request
GET http://localhost:3000/students

// REST Client Response
Status: 200 OK Size: 536 Bytes Time: 7 ms
{
  "success": true,
  "data": [
    {
      "id": 1,
      "name": "Sheldon Cooper",
      "age": 38,
      "course": "Physics"
    },
    {
      "id": 2,
      "name": "Leonard Hofstadter",
      "age": 29,
      "course": "Physics"
    },
    {
      "id": 3,
      "name": "Penny",
      "age": 28,
      "course": "Theatre"
    },
    {
      "id": 4,
      "name": "Howard Wolowitz",
      "age": 31,
      "course": "Engineering"
    },
    {
      "id": 5,
      "name": "Rajesh Koothrappali",
      "age": 30,
      "course": "Astronomy"
    }
  ]
}

// Server Code (server.js)
const DATA_PATH = path.join(__dirname, 'students.json');

function loadStudents() {
  try {
    const raw = fs.readFileSync(DATA_PATH, 'utf-8');
    const data = JSON.parse(raw);
    return Array.isArray(data) ? data : [];
  } catch (e) {
    return [];
  }
}

function saveStudents(students) {
  fs.writeFileSync(DATA_PATH, JSON.stringify(students, null, 2), 'utf-8');
}

let students = loadStudents();
let nextId = students.length ? Math.max(...students.map(s => s.id || 0)) + 1 : 1;

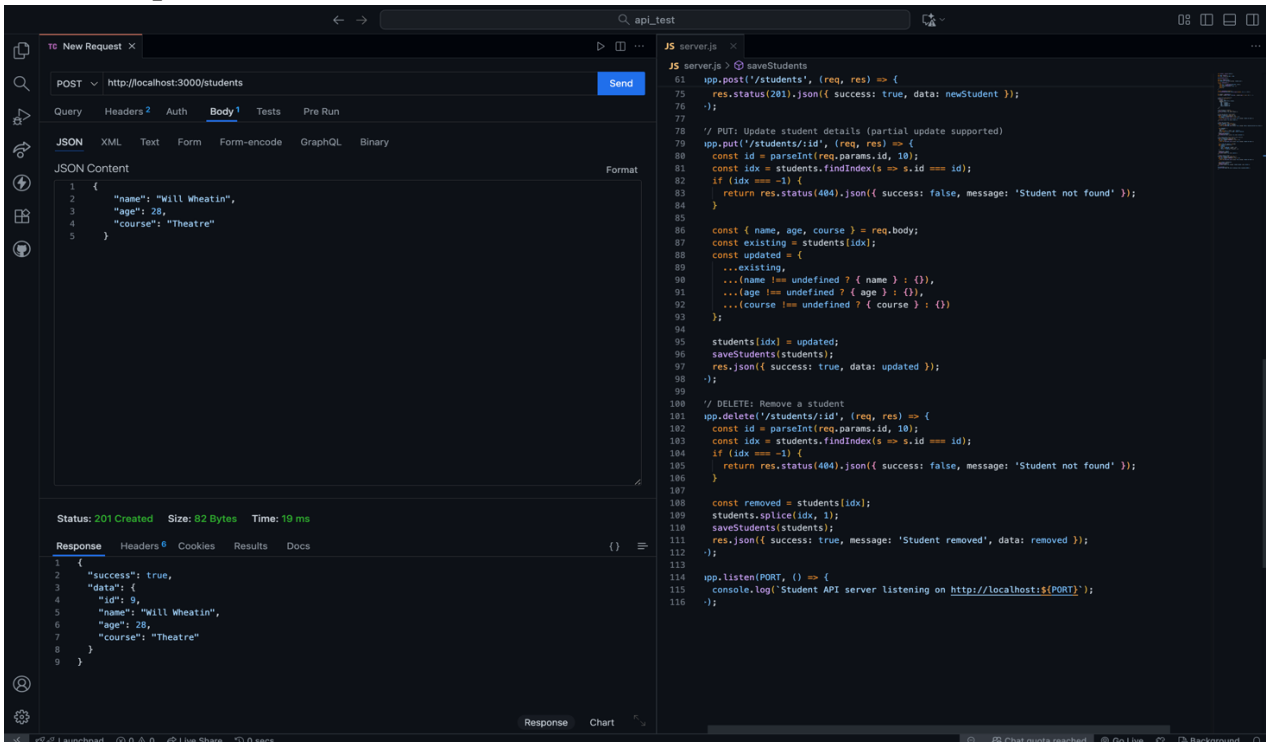
// Root route: quick status/info
app.get('/', (req, res) => {
  res.json({
    success: true,
    message: 'Student API is running',
    endpoints: [
      'GET /students',
      'GET /students/:id',
      'POST /students',
      'PUT /students/:id',
      'DELETE /students/:id'
    ]
  });
});

// GET: Retrieve all students
app.get('/students', (req, res) => {
  res.json({ success: true, data: students });
});

// GET by ID: Retrieve a single student
app.get('/students/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const student = students.find(s => s.id === id);
  if (!student) {
    return res.status(404).json({ success: false, message: 'Student not found' });
  }
  res.json({ success: true, data: student });
});

// POST: Add a new student
app.post('/students', (req, res) => {
  const { name, age, course } = req.body;
  // ... (code for saving new student)
});
  
```

## POST Request – Add New Student



The screenshot shows a REST client interface in VS Code. The request is a POST to `http://localhost:3000/students` with a JSON body containing a new student object. The response is a 201 Created status with a JSON body containing the new student object. The server code on the right shows the `app.post('/students')` route handler.

```

// REST Client Request
POST http://localhost:3000/students
{
  "name": "Will Wheatin",
  "age": 28,
  "course": "Theatre"
}

// REST Client Response
Status: 201 Created Size: 82 Bytes Time: 19 ms
{
  "success": true,
  "data": {
    "id": 6,
    "name": "Will Wheatin",
    "age": 28,
    "course": "Theatre"
  }
}

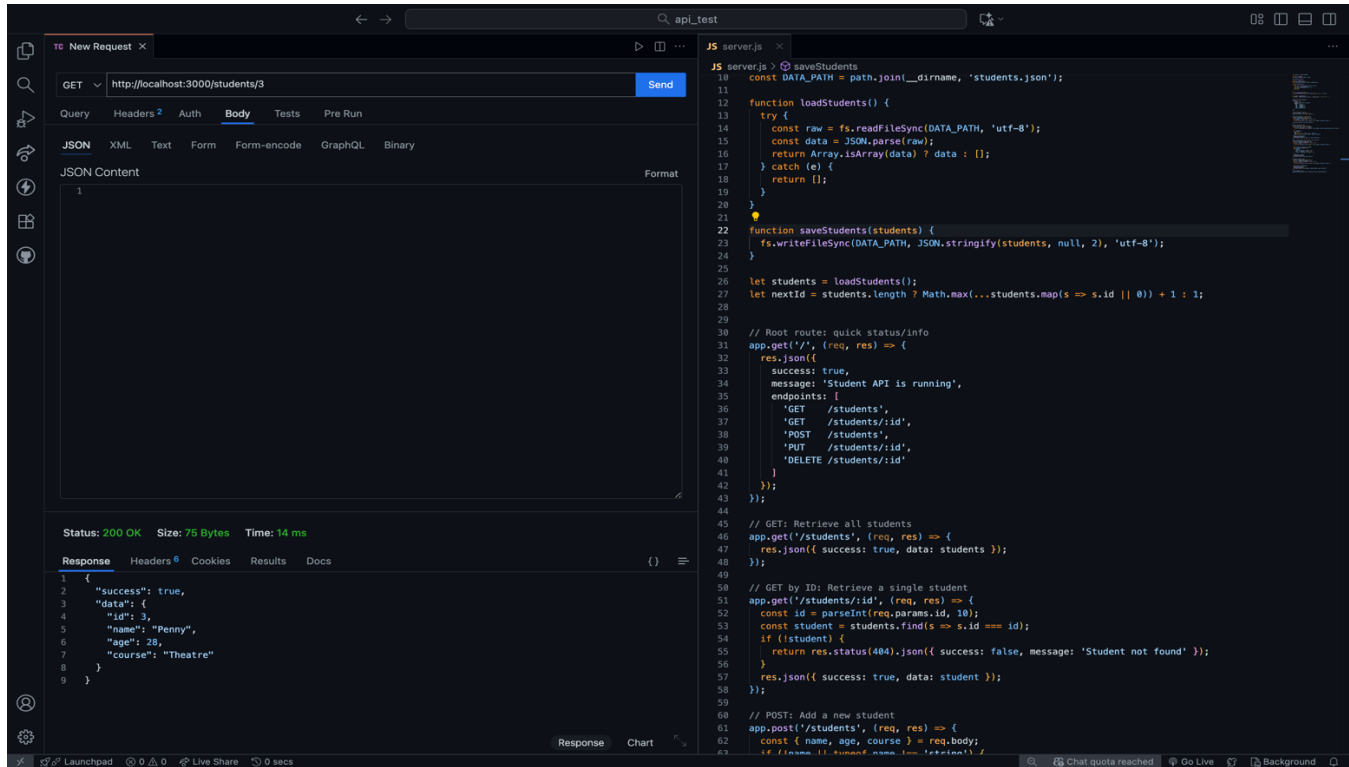
// Server Code (server.js)
app.post('/students', (req, res) => {
  res.status(201).json({ success: true, data: newStudent });
});

// PUT: Update student details (partial update supported)
app.put('/students/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const idx = students.findIndex(s => s.id === id);
  if (idx === -1) {
    return res.status(404).json({ success: false, message: 'Student not found' });
  }
  const { name, age, course } = req.body;
  const existing = students[idx];
  const updated = {
    ...existing,
    ...(name !== undefined ? { name } : {}),
    ...(age !== undefined ? { age } : {}),
    ...(course !== undefined ? { course } : {})
  };
  students[idx] = updated;
  saveStudents(students);
  res.json({ success: true, data: updated });
});

// DELETE: Remove a student
app.delete('/students/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const idx = students.findIndex(s => s.id === id);
  if (idx === -1) {
    return res.status(404).json({ success: false, message: 'Student not found' });
  }
  const removed = students[idx];
  students.splice(idx, 1);
  saveStudents(students);
  res.json({ success: true, message: 'Student removed', data: removed });
});

app.listen(PORT, () => {
  console.log('Student API server listening on http://localhost:1(PORT)');
});
  
```

## GET by ID Request – Fetch Single Student



The screenshot displays a REST client (Postman) and a code editor (VS Code) side-by-side. The REST client shows a GET request to `http://localhost:3000/students/3` with a status of 200 OK. The response is a JSON object representing a student with ID 3, name 'Penny', age 28, and course 'Theatre'.

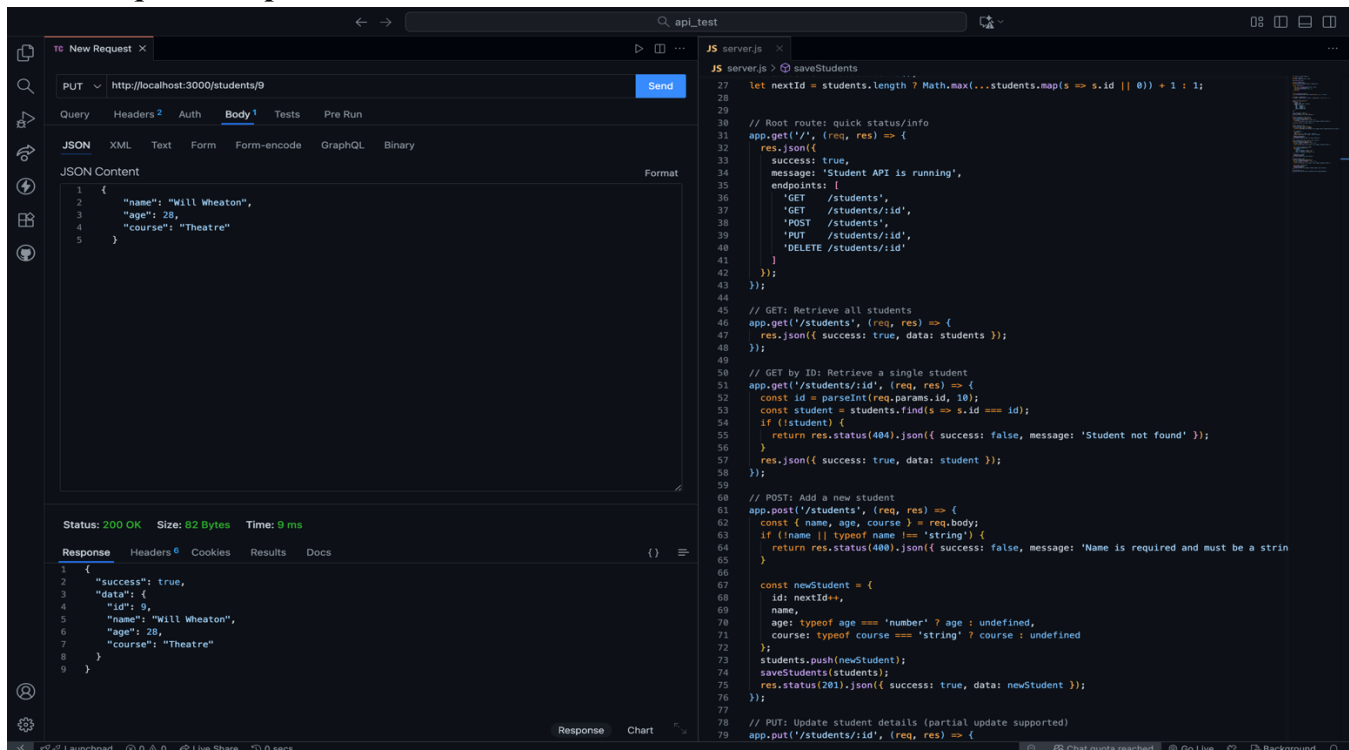
The code editor shows the Node.js server code. The relevant part for the GET by ID request is as follows:

```

// GET by ID: Retrieve a single student
app.get('/students/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const student = students.find(s => s.id === id);
  if (!student) {
    return res.status(404).json({ success: false, message: 'Student not found' });
  }
  res.json({ success: true, data: student });
});

```

## PUT Request – Update Student Data



The screenshot displays a REST client (Postman) and a code editor (VS Code) side-by-side. The REST client shows a PUT request to `http://localhost:3000/students/9` with a status of 200 OK. The request body is a JSON object representing a student with ID 9, name 'Will Wheaton', age 28, and course 'Theatre'.

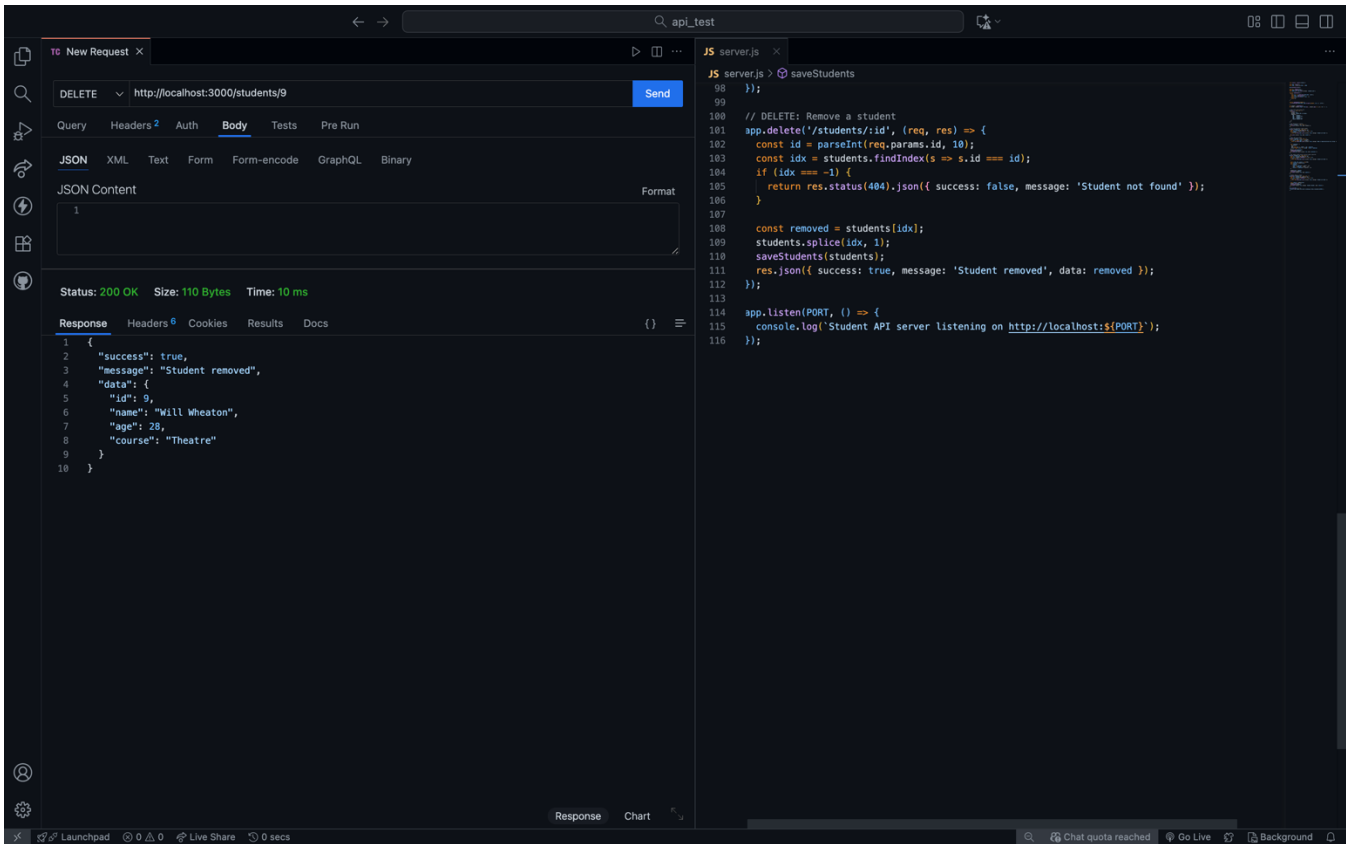
The code editor shows the Node.js server code. The relevant part for the PUT request is as follows:

```

// PUT: Update student details (partial update supported)
app.put('/students/:id', (req, res) => {
  const { name, age, course } = req.body;
  if (!name || typeof name !== 'string') {
    return res.status(400).json({ success: false, message: 'Name is required and must be a string' });
  }
  const newStudent = {
    id: nextId++,
    name,
    age: typeof age === 'number' ? age : undefined,
    course: typeof course === 'string' ? course : undefined
  };
  students.push(newStudent);
  saveStudents(students);
  res.status(201).json({ success: true, data: newStudent });
});

```

## DELETE Request – Remove Student Record



The screenshot displays the Thunder Client interface with a new request configured for a DELETE operation on the endpoint `http://localhost:3000/students/9`. The request body is empty. The response status is `200 OK` with a size of `110 Bytes` and a time of `10 ms`. The response body is a JSON object:

```

{
  "success": true,
  "message": "Student removed",
  "data": {
    "id": 9,
    "name": "Will Wheaton",
    "age": 28,
    "course": "Theatre"
  }
}

```

On the right, the `server.js` file shows the implementation of the delete endpoint:

```

// ...
98 });
99
100 // DELETE: Remove a student
101 app.delete('/students/:id', (req, res) => {
102   const id = parseInt(req.params.id, 10);
103   const idx = students.findIndex(s => s.id === id);
104   if (idx === -1) {
105     return res.status(404).json({ success: false, message: 'Student not found' });
106   }
107
108   const removed = students[idx];
109   students.splice(idx, 1);
110   saveStudents(students);
111   res.json({ success: true, message: 'Student removed', data: removed });
112 });
113
114 app.listen(PORT, () => {
115   console.log('Student API server listening on http://localhost:${PORT}');
116 });

```

### Conclusion:

In this experiment, a RESTful API was successfully implemented using Express.js and Node.js. The API handled CRUD operations effectively and was tested using Thunder Client. All endpoints returned the expected responses, demonstrating the core functionality of RESTful services.