Experiment 6

Aim:

Building a RESTful API with Express. Create routes for CRUD operations (Create, Read, Update, Delete) on a mock dataset. Use Express middleware for request handling and Validation.

Theory:

Overview of Express's RESTful APIs

A standardized mechanism for creating web services that communicate via HTTP methods like GET, POST, PUT, and DELETE is offered by RESTful APIs (Representational State Transfer). Express.js makes it easier to create RESTful APIs by providing user-friendly request handling, middleware support, and routing. Developers may effectively define API endpoints with Express, guaranteeing scalability and maintainability. Web and mobile apps frequently employ RESTful APIs because they facilitate smooth communication between clients and servers.

Body-Parser in Express Framework

When working with JSON or URL-encoded data, Express.js uses the body-parser middleware to handle incoming request bodies. Data extraction from POST and PUT requests need it. Important features include of:

• JSON Data Parsing: Transforms JSON payloads into JavaScript objects that may be accessed by req.body.
• Managing URL-Encoded Data: Parses application/x-www-form-urlencoded data to support form submissions.
• Enhancing Request Processing: Facilitates effective access to request data via middleware and route handlers.

Body-parser is still frequently used for improved request body handling even if Express 4.16+ comes with built-in express.json() and express.urlencoded().

CORS and Middleware in Express for Validation and Request Processing

In Express, middleware serves as a link between the final response and an incoming request. Functions in the middleware can change request and response objects, end requests, or transfer control to the following function.

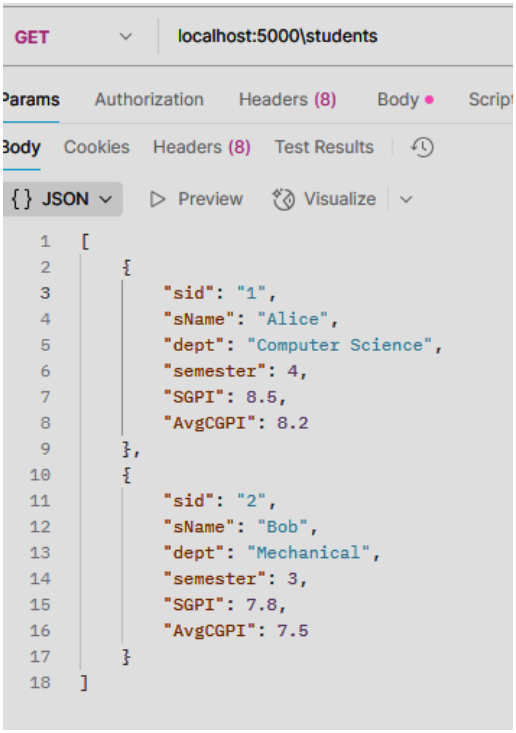
- Typical use scenarios include the following:
  Verifying that users have the appropriate permissions is known as authentication and authorization.
  • Logging and Debugging: Monitoring HTTP requests using Morgan.
  • Error handling: identifying and politely handling mistakes.
  • Data validation: Verifying input fields prior to request processing.

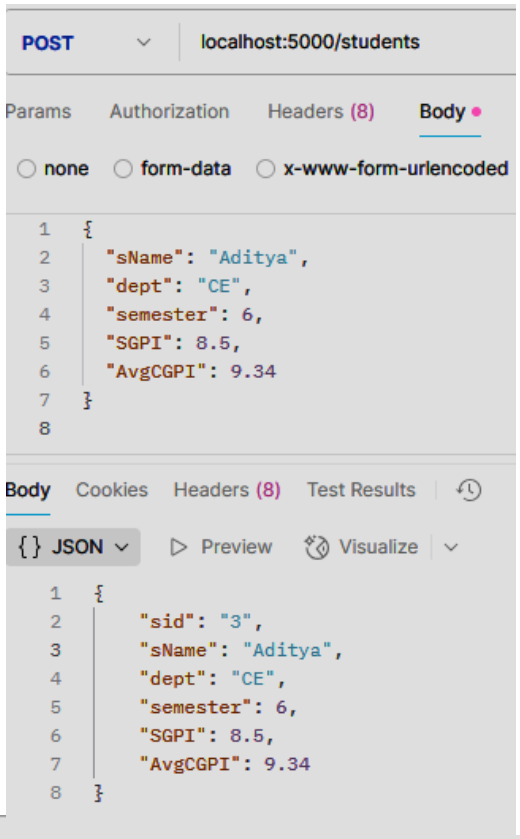
- CORS (Cross-Origin Resource Sharing) is a middleware used to allow or restrict cross-origin requests. By default, browsers enforce the Same-Origin Policy, which prevents requests from different domains. The cors package in Express allows developers to control access to their API by specifying allowed origins, methods, and headers. Key features of CORS include:
  - Enabling or Restricting API Access: Define allowed domains using options like origin: '*' (allow all) or a specific domain.
  - Supporting Custom Headers: Specify necessary headers for authentication and security.
  - Handling Preflight Requests: Automatically responds to OPTIONS requests for CORS-enabled endpoints. Using middleware and CORS properly ensures secure and efficient
  - request handling in Express applications.

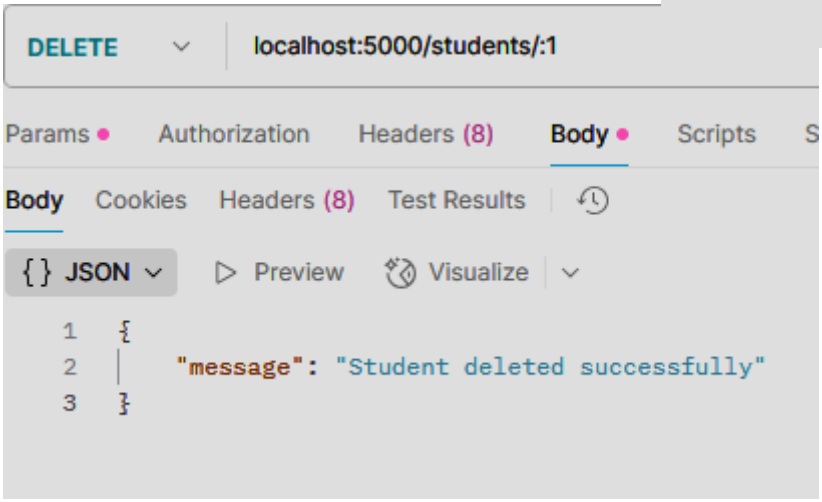
Sample Code:

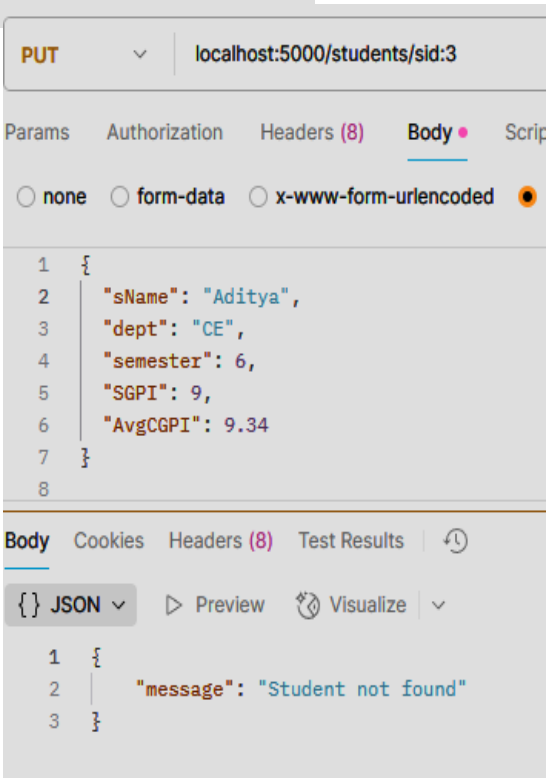
//Server.js

```
const express = require("express");
const bodyParser = require("body-parser"); const cors = require("cors")
const app = express(); const PORT = 5000;
app.use(cors());
app.use(bodyParser.json());


let students = [
{ sid: "1", sName: "Alice", dept: "Computer Science", semester: 4, SGPI: 8.5, AvgCGPI: 8.2 },
{ sid: "2", sName: "Bob", dept: "Mechanical", semester: 3, SGPI: 7.8, AvgCGPI: 7.5 }
]

const validateStudent = (req, res, next) => {
const { sName, dept, semester, SGPI, AvgCGPI } = req.body;
if (!sName || !dept || semester == null || SGPI == null || AvgCGPI == null) {
return res.status(400).json({ error: "All fields are required: sName, dept, semester, SGPI, AvgCGPI" });
}
if (typeof semester !== "number" || semester < 1 || semester > 8) {
return res.status(400).json({ error: "Semester must be a number between 1 and 8" });
}
if (typeof SGPI !== "number" || SGPI < 0 || SGPI > 10) {
return res.status(400).json({ error: "SGPI must be between 0 and 10" });
}
if (typeof AvgCGPI !== "number" || AvgCGPI < 0 || AvgCGPI > 10) {
return res.status(400).json({ error: "AvgCGPI must be between 0 and 10" });
}
next();
};
app.post("/students", validateStudent, (req, res) => {
const newStudent = { sid: String(students.length + 1), ...req.body }; students.push(newStudent);
res.status(201).json(newStudent);
});
app.get("/students", (req, res) => { res.json(students);
});
app.get("/students/:sid", (req, res) => {
const student = students.find(s => s.sid === req.params.sid);
if (!student) return res.status(404).json({ message: "Student not found" }); res.json(student);
});
app.put("/students/:sid", validateStudent, (req, res) => {
const index = students.findIndex(s => s.sid === req.params.sid);
if (index === -1) return res.status(404).json({ message: "Student not found" });

students[index] = { ...students[index], ...req.body }; res.json(students[index]);
});
app.delete("/students/:sid", (req, res) => {
students = students.filter(s => s.sid !== req.params.sid); res.json({ message: "Student deleted successfully"
});
});

app.delete("/students/:sid", (req, res) => {
students = students.filter(s => s.sid !== req.params.sid); res.json({ message: "Student deleted successfully"
});
});
app.listen(PORT, () => console.log(`Server running on port :{PORT}`
```









Conclusion:

Express.js simplifies the development of RESTful APIs by offering robust routing, middleware integration, and request handling mechanisms. The body-parser middleware efficiently processes request bodies, while CORS enables secure cross-origin requests. Middleware enhances functionality by enabling logging, validation, authentication, and error handling. Together, these components make Express a powerful framework for building scalable web application