Web Technology / Full Stack Development

RESTful API Design • CRUD Operations • MVC Architecture • Node.js and Express Basics

1. RESTful API Design

1.1 Introduction

An API (Application Programming Interface) is a set of rules that lets one software component use the services of another. A web API exposes the functions of a server over HTTP so that browsers, mobile apps and other servers can use them.

REST (Representational State Transfer) is an architectural style for designing web services, proposed by Roy Fielding in his doctoral thesis in 2000. In REST, everything is a resource identified by a URI, and clients manipulate resources by exchanging representations of them (usually JSON) using the standard HTTP methods. An API that follows the REST principles is called a RESTful API.

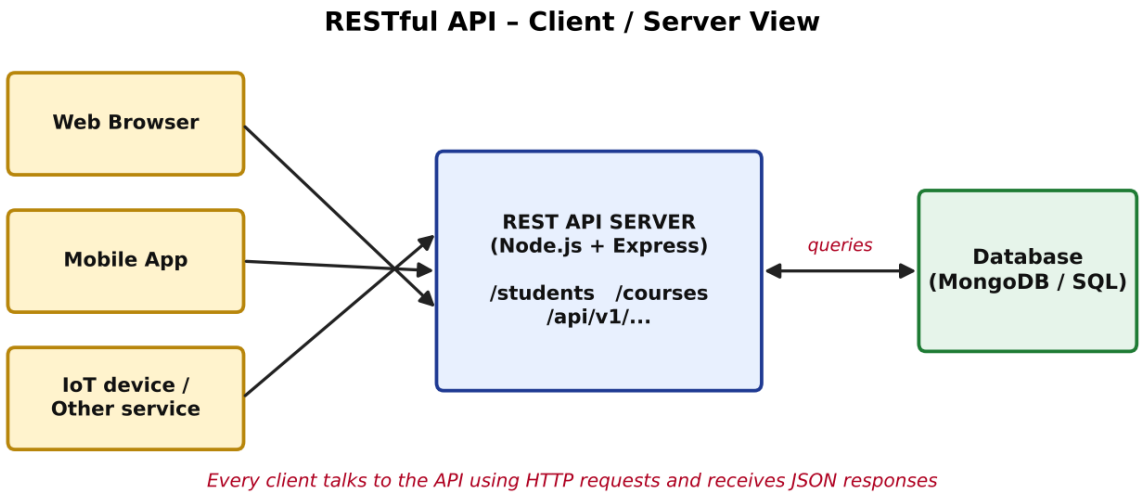


Fig. 1 RESTful API – client/server view

1.2 Architectural Constraints of REST

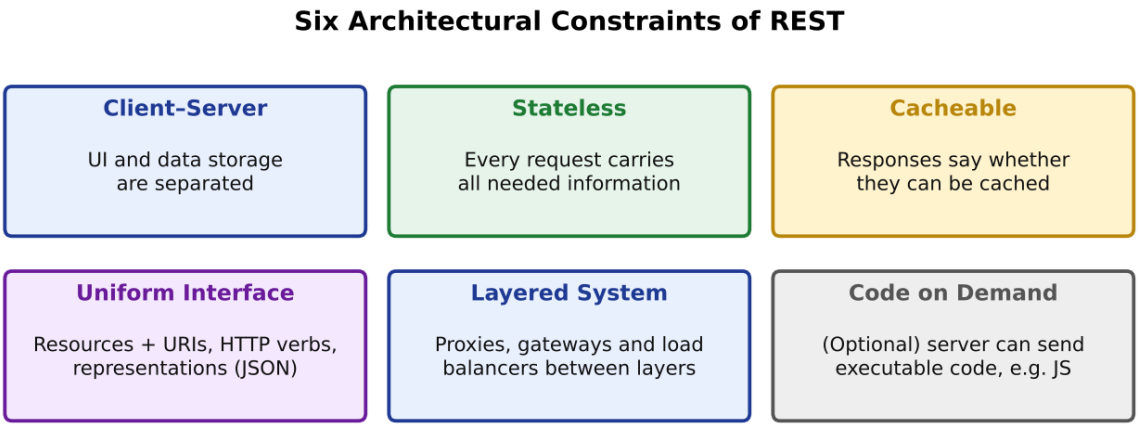


Fig. 2 Constraints of REST

Constraint	Meaning
Client–Server	The user interface (client) and data storage (server) are separate, so each can evolve independently.
Stateless	The server stores no client context between requests; every request contains all the information needed (e.g. token in header). This improves scalability.
Cacheable	Responses must declare whether they can be cached (Cache-Control, ETag) to reduce load and latency.
Uniform Interface	A consistent way to access resources: identification by URI, manipulation through representations, self-descriptive messages and (optionally) HATEOAS links.
Layered System	Client cannot tell whether it talks to the real server or an intermediary (proxy, load balancer, API gateway).
Code on Demand (optional)	Server may send executable code such as JavaScript to the client.

1.3 Resources and URIs

A resource is any thing the API exposes – a student, a course, an order. Each resource is addressed by a unique URI. A collection is a group of resources (/students) and an item is one member of it (/students/7).

URI design rules

- **Use nouns, not verbs:** the HTTP method already tells the action.
- **Use plural names for collections:** /students, /courses.
- **Use lowercase letters and hyphens:** /course-registrations (not /CourseRegistrations).
- **Show hierarchy with slashes:** /students/7/courses lists the courses of student 7.

- **Use query strings for filtering, sorting and paging:** /students?dept=CSE&sort=name&page=2&limit=10.
- **No file extensions and no trailing slash.**

Poor design	Good design
GET /getAllStudents	GET /students
POST /createStudent	POST /students
POST /deleteStudent?id=7	DELETE /students/7
GET /student/7/getCourses	GET /students/7/courses
GET /Students.json	GET /students (with Accept: application/json)

1.4 HTTP Methods

Method	Purpose	Safe	Idempotent	Body
GET	Retrieve a resource or collection	Yes	Yes	No
POST	Create a new resource (or run a non-idempotent action)	No	No	Yes
PUT	Replace an entire resource (or create at a known URI)	No	Yes	Yes
PATCH	Partially update a resource	No	No*	Yes
DELETE	Remove a resource	No	Yes	Usually no
HEAD / OPTIONS	Headers only / supported methods (used in CORS)	Yes	Yes	No

Safe means the method does not change server data. Idempotent means repeating the same request has the same effect as sending it once (calling DELETE /students/7 twice leaves the same final state). *PATCH is not guaranteed to be idempotent.

1.5 HTTP Status Codes

Code	Name	Typical use in a REST API
200	OK	Successful GET, PUT, PATCH
201	Created	POST created a resource; send Location header
204	No Content	Successful DELETE or update with no body
304	Not Modified	Cached copy is still valid
400	Bad Request	Malformed JSON or invalid input
401	Unauthorized	Missing or invalid authentication
403	Forbidden	Authenticated but not permitted
404	Not Found	Resource does not exist

Code	Name	Typical use in a REST API
409	Conflict	Duplicate or version conflict
422	Unprocessable Entity	Validation failed
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Unexpected server failure
503	Service Unavailable	Server overloaded or down for maintenance

Classes: 1xx informational, 2xx success, 3xx redirection, 4xx client error, 5xx server error.

1.6 Request and Response Structure

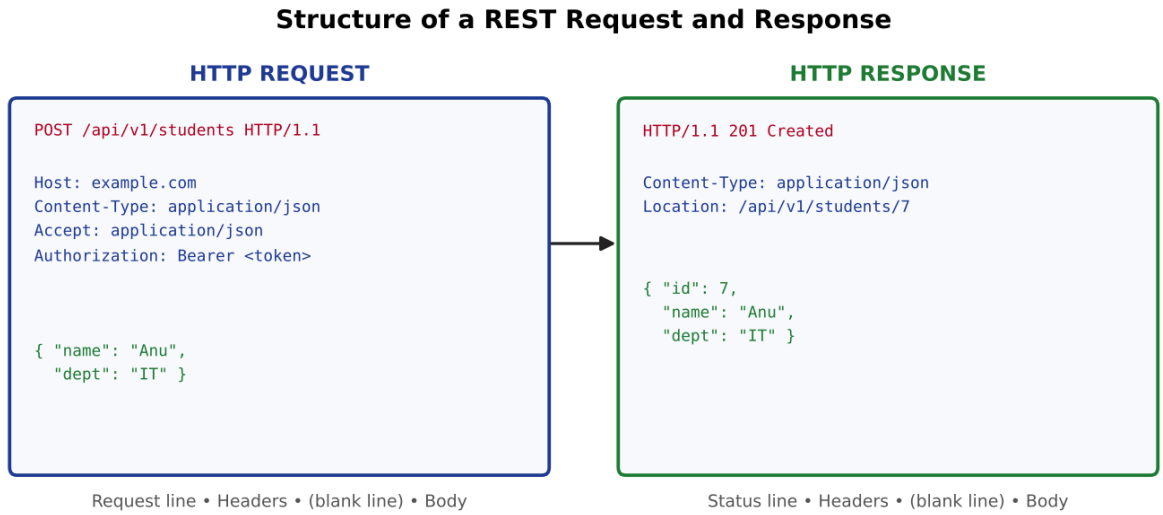


Fig. 3 Structure of a REST request and response

- **Content-Type:** format of the body being sent (application/json).
- **Accept:** format the client wants back (content negotiation).
- **Authorization:** credentials or token, e.g. Bearer <JWT>.
- **Location:** URI of a newly created resource.

JSON (JavaScript Object Notation) is the most common representation because it is lightweight, human-readable and natively understood by JavaScript. XML is also possible.

1.7 Best Practices in API Design

Practice	Description
Versioning	Put the version in the URI (/api/v1/students) or a header so old clients keep working
Pagination	Never return huge lists: ?page=2&limit=10 and include total count
Filtering and sorting	?dept=CSE&sort=-marks through query parameters
Consistent error format	{ "error": "Student not found", "code": 404 } for every failure
Use correct status codes	201 for creation, 404 for missing resource, not 200 for everything

Practice	Description
HTTPS everywhere	Encrypts tokens and data in transit
Authentication	Token based (JWT / OAuth 2.0) in the Authorization header
Validation	Validate and sanitise all input on the server
Rate limiting	Restrict calls per client to prevent abuse (429)
CORS	Allow only trusted origins to call the API from browsers
Documentation	Describe endpoints with OpenAPI / Swagger
HATEOAS (optional)	Include links to related actions in the response

1.8 REST vs SOAP

REST	SOAP
Architectural style	Protocol with strict standards
Uses HTTP methods and URIs	Uses a single endpoint and HTTP POST
JSON, XML, text	XML only (SOAP envelope)
Lightweight, fast, cacheable	Heavier, no caching
Description by OpenAPI (optional)	Described by WSDL (mandatory)
Popular for web and mobile apps	Used in enterprise systems needing WS-Security and transactions

1.9 Example – Designing an API for a Student Resource

Operation	Method and URI	Success status
List all students	GET /api/v1/students	200 OK
List with filter and paging	GET /api/v1/students?dept=CSE&page=1&limit=10	200 OK
Get one student	GET /api/v1/students/7	200 OK / 404
Create a student	POST /api/v1/students	201 Created
Replace a student	PUT /api/v1/students/7	200 OK
Update some fields	PATCH /api/v1/students/7	200 OK
Delete a student	DELETE /api/v1/students/7	204 No Content
Courses of a student	GET /api/v1/students/7/courses	200 OK

2. CRUD Operations

2.1 Definition

CRUD stands for Create, Read, Update and Delete – the four basic operations performed on persistent data in almost every application. A RESTful API exposes these operations over HTTP, and the server translates them into database operations (SQL or NoSQL).

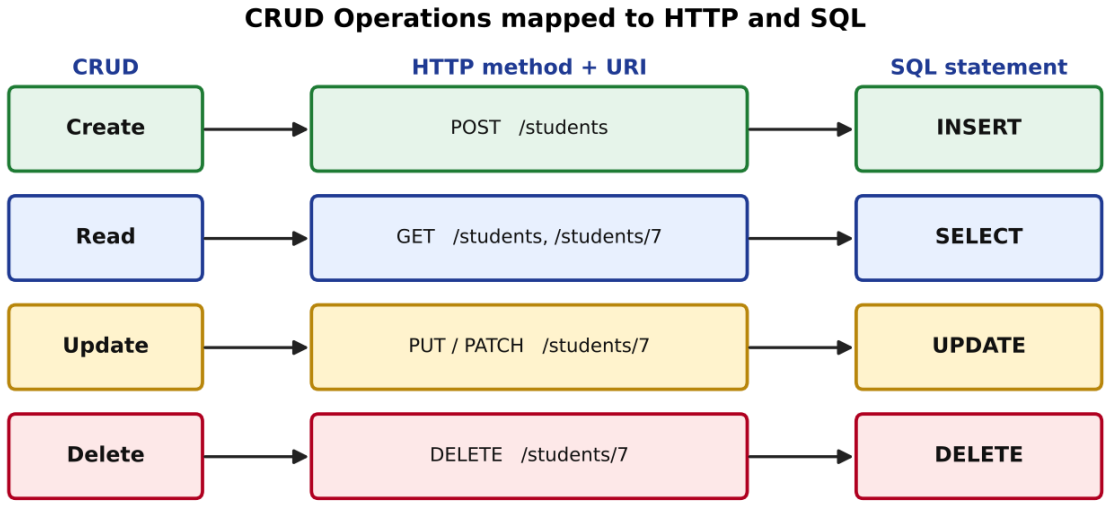


Fig. 4 CRUD operations mapped to HTTP and SQL

CRUD	HTTP	SQL	MongoDB (Mongoose)
Create	POST /students	INSERT INTO	Student.create(), insertOne()
Read	GET /students, GET /students/:id	SELECT	Student.find(), findById()
Update	PUT / PATCH /students/:id	UPDATE	findByIdAndUpdate(), updateOne()
Delete	DELETE /students/:id	DELETE FROM	findByIdAndDelete(), deleteOne()

2.2 Create – POST

The client sends the new resource in the request body. The server validates it, stores it, generates an id, and responds with 201 Created, a Location header and the created resource.

```
POST /api/v1/students
Content-Type: application/json

{ "name": "Anu", "dept": "IT", "year": 3 }

---- Response ----
HTTP/1.1 201 Created
Location: /api/v1/students/7
{ "id": 7, "name": "Anu", "dept": "IT", "year": 3 }
```

2.3 Read – GET

```
GET /api/v1/students/7          ->  200 OK
{ "id": 7, "name": "Anu", "dept": "IT", "year": 3 }

GET /api/v1/students?dept=IT    ->  200 OK
[ { "id": 7, ... }, { "id": 9, ... } ]

GET /api/v1/students/999        ->  404 Not Found
{ "error": "Student not found" }
```

2.4 Update – PUT and PATCH

```
PUT /api/v1/students/7            (all fields must be sent)
{ "name": "Anu R", "dept": "CSE", "year": 3 }      ->  200 OK

PATCH /api/v1/students/7          (only changed fields)
{ "dept": "CSE" }                                    ->  200 OK
```

PUT	PATCH
Replaces the whole resource	Modifies only the given fields
Client sends the full representation	Client sends only the changes
Idempotent	Not necessarily idempotent
Missing fields may be reset or removed	Missing fields are left unchanged

2.5 Delete – DELETE

```
DELETE /api/v1/students/7   ->  204 No Content
DELETE /api/v1/students/7   ->  404 Not Found   (already removed)
```

2.6 POST vs PUT

POST	PUT
Sent to the collection (/students); server chooses the id	Sent to a specific item (/students/7); client knows the URI
Not idempotent – repeating creates duplicates	Idempotent – repeating gives the same result
Usually returns 201 Created	Returns 200 OK (or 201 if it created the item)

2.7 CRUD in SQL for the Student Table

```
INSERT INTO student(name, dept, year) VALUES ('Anu', 'IT', 3);   -- Create
SELECT * FROM student WHERE id = 7;                              -- Read
UPDATE student SET dept = 'CSE' WHERE id = 7;                    -- Update
DELETE FROM student WHERE id = 7;                                -- Delete
```

The complete implementation of these operations in Express is given in Section 5.

3. MVC Architecture

3.1 Introduction

Model–View–Controller (MVC) is a software design pattern that divides an application into three interconnected parts so that the data, the user interface and the control logic are kept separate. It was first used in Smalltalk and is now the standard structure of web frameworks such as Spring MVC, ASP.NET MVC, Django, Ruby on Rails, Laravel and Express-based applications.

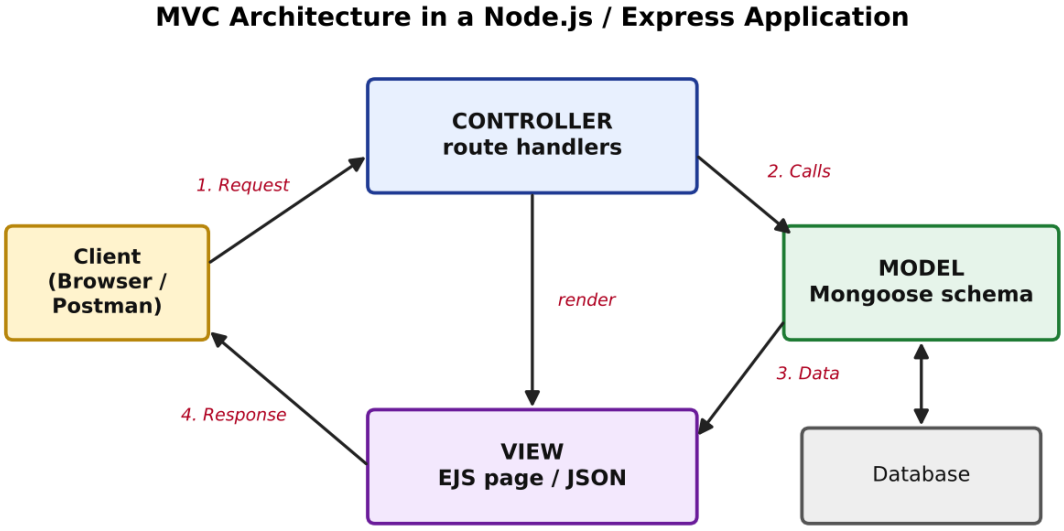


Fig. 5 MVC architecture in an Express application

3.2 Components of MVC

Component	Responsibility	In Node.js / Express
Model	Represents the data and business rules; validates data; talks to the database	Mongoose schema / Sequelize model, DAO functions
View	Presents data to the user; contains no business logic	EJS / Pug / Handlebars template, React front end, or the JSON response of a REST API
Controller	Receives the request, calls the model, selects the view / builds the response	Route handler functions (e.g. studentController.js)

3.3 Flow of Control

1. The client sends an HTTP request, e.g. GET /students/7.
2. The router maps the URL and method to a controller function.
3. The controller validates the input and asks the model for data.
4. The model queries the database and returns the result to the controller.
5. The controller passes the data to the view (renders a template) or sends it directly as JSON.

6. The response is sent back to the client.

3.4 Advantages and Disadvantages

Advantages	Disadvantages
Separation of concerns – easy to understand and maintain	More files and structure for very small applications
Parallel development: UI team and back-end team work independently	Controllers can become too large ('fat controllers') if not designed well
Model and logic can be reused with different views (web, mobile, API)	Tight coupling between view and model data format may need extra effort
Easier unit testing of each layer	Learning curve for beginners

3.5 MVC Project Structure in Express

```
student-app/
├── package.json
├── server.js              // starts the server
├── app.js                 // creates Express app, registers middleware and
routes
├── config/
│   └── db.js              // database connection
├── models/
│   └── Student.js         // MODEL
├── controllers/
│   └── studentController.js   // CONTROLLER
├── routes/
│   └── studentRoutes.js   // URL -> controller mapping
├── views/                 // VIEW (only for server-rendered pages)
│   └── students.ejs
├── middleware/
│   └── errorHandler.js
└── public/                // static files (css, js, images)
```

In a REST API the 'view' is simply the JSON representation sent by the controller. In a server-rendered application the view is an HTML template. In a single-page application (React, Angular) the view lives completely in the browser and the Express server acts only as controller and model.

3.6 MVC vs Servlet/JSP Model 2

Servlet / JSP (Model 2)	Node.js / Express
Controller: Servlet	Controller: route handler functions
View: JSP page	View: EJS/Pug template or JSON
Model: JavaBean + DAO using JDBC	Model: Mongoose / Sequelize model
Mapping through web.xml or @WebServlet	Mapping through app.get(), router.post() etc.
Thread per request on a servlet container	Single event loop, non-blocking I/O

4. Node.js Basics

4.1 What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript outside the browser. It was created by Ryan Dahl in 2009 and is built on Google's V8 JavaScript engine (written in C++) together with the libuv library, which provides the event loop and asynchronous I/O. Node.js allows JavaScript to be used on both client and server, so a full application can be written in one language.

Features

- **Asynchronous and event-driven:** I/O calls do not block the program; callbacks/promises handle the results.
- **Single-threaded event loop:** one thread handles thousands of concurrent connections with low memory.
- **Fast:** V8 compiles JavaScript to machine code.
- **npm:** the world's largest package ecosystem.
- **Cross-platform:** Windows, Linux and macOS.
- **Good for:** REST APIs, real-time apps (chat, streaming), microservices. **Not ideal for:** heavy CPU-bound computation, which blocks the event loop.

4.2 Node.js Architecture

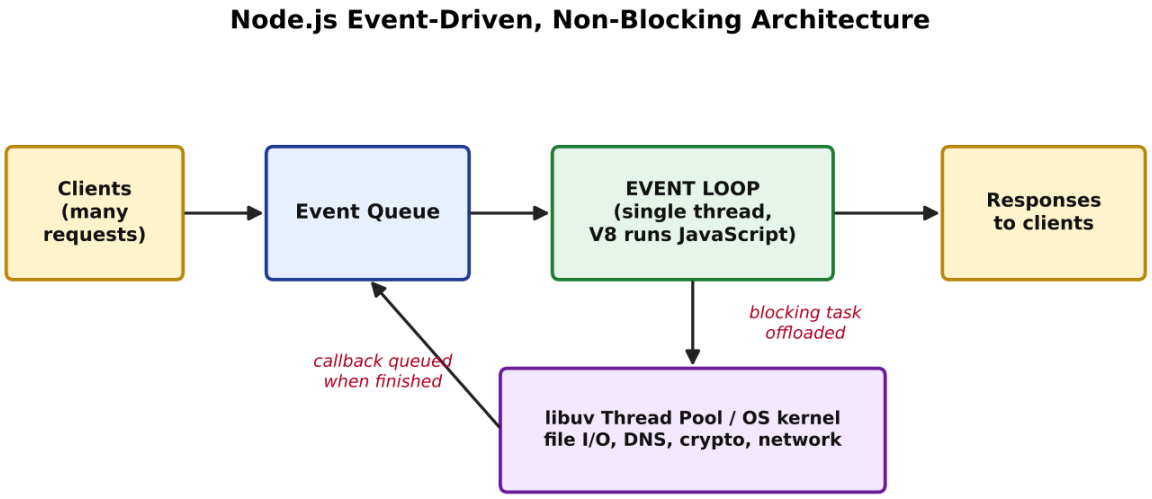


Fig. 6 Node.js event-driven, non-blocking architecture

1. Clients send requests; they are placed in the event queue.
2. The single-threaded event loop picks requests one by one and runs the JavaScript handler.
3. Non-blocking operations are completed immediately. Blocking operations (file system, DNS, crypto, database calls) are offloaded to the libuv thread pool or the operating system.

4. When the task finishes, its callback is placed back in the event queue.
5. The event loop executes the callback and the response is sent to the client.

Phases of the event loop

Phase	Work done
Timers	Runs callbacks of setTimeout() and setInterval()
Pending callbacks	Some system-level callbacks
Poll	Retrieves new I/O events and runs I/O callbacks
Check	Runs setImmediate() callbacks
Close callbacks	Handles close events such as socket.on('close')

process.nextTick() callbacks and resolved promises (microtasks) run between phases, before the loop moves on.

4.3 Node.js vs Traditional Server Model

Node.js	Traditional (thread per request)
One thread, event loop	A new thread or process for every request
Non-blocking I/O	Blocking I/O
Low memory per connection	More memory as threads increase
Excellent for I/O-heavy workloads	Better for CPU-heavy work using many cores
JavaScript on server and client	Different languages (Java, PHP, C#)

4.4 Installing and Running Node.js

```
node -v                  # check version
npm -v                   # check npm version
node                     # starts the REPL (interactive shell)
node app.js              # runs a script
```

```
// app.js
console.log("Hello from Node.js");
```

4.5 npm and package.json

npm (Node Package Manager) installs and manages third-party libraries. Each project has a package.json file describing the project, its scripts and dependencies; installed packages are stored in the node_modules folder and exact versions are locked in package-lock.json.

```
npm init -y                      # create package.json
npm install express              # add a dependency
npm install --save-dev nodemon   # development-only dependency
npm start                        # runs the 'start' script
```

```
{
  "name": "student-app",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": { "express": "^5.0.0", "mongoose": "^8.0.0" }
}
```

4.6 Modules

A module is a separate file whose code is private unless it is exported. Node.js supports CommonJS modules (require / module.exports) and ES modules (import / export, enabled by "type": "module" in package.json or the .mjs extension).

```
// mathUtil.js  (CommonJS)
function add(a, b) { return a + b; }
module.exports = { add };

// main.js
const { add } = require("./mathUtil");
console.log(add(2, 3));     // 5
```

Important core modules

Module	Purpose
fs	Read, write and manage files
http / https	Create web servers and make HTTP requests
path	Handle file and directory paths
os	Information about the operating system
events	EventEmitter class for event-driven code
url / querystring	Parse URLs and query strings
crypto	Hashing, encryption, random values
stream	Process data in chunks (large files, network)

4.7 Asynchronous Programming

Because Node.js has a single thread, a slow operation must never block it. Asynchronous code is written in three styles:

(a) Callbacks

```
const fs = require("fs");
fs.readFile("data.txt", "utf8", (err, data) => {
  if (err) return console.error(err);
  console.log(data);
});
console.log("This line runs first");   // readFile does not block
```

Deeply nested callbacks lead to 'callback hell', which promises and async/await solve.

(b) Promises

```
const fsp = require("fs").promises;
fsp.readFile("data.txt", "utf8")
   .then(data => console.log(data))
   .catch(err => console.error(err));
```

(c) async / await

```
async function show() {
  try {
    const data = await fsp.readFile("data.txt", "utf8");
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
show();
```

Synchronous (blocking)	Asynchronous (non-blocking)
fs.readFileSync() waits until the file is read	fs.readFile() returns immediately and calls back later
Stops all other requests meanwhile	Other requests are served meanwhile
Acceptable only at start-up or in scripts	Required inside servers

4.8 Events and EventEmitter

```
const EventEmitter = require("events");
const emitter = new EventEmitter();

emitter.on("login", user => console.log(user + " logged in"));   // listener
emitter.emit("login", "Ravi");                                   // trigger
```

4.9 A Web Server using the http Module

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

const server = http.createServer((req, res) => {
  if (req.method === "GET" && req.url === "/hello") {
    res.writeHead(200, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ message: "Hello, Node.js" }));
  } else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("Not Found");
  }
});

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

The raw http module is low-level: routing, parsing JSON bodies, cookies, static files and error handling must all be coded by hand. Express adds these features with a clean API.

5. Express Basics

5.1 Introduction

Express.js is a minimal, fast and flexible web application framework for Node.js. It sits on top of the http module and provides routing, middleware, request/response helpers, template-engine support and static file serving. It is the 'E' in the MEAN and MERN stacks (MongoDB, Express, Angular/React, Node).

Features

- Simple and powerful routing for every HTTP method.
- Middleware pipeline for logging, parsing, authentication and error handling.
- Template engines (EJS, Pug, Handlebars) for server-side views.
- Static file serving, JSON helpers and easy integration with databases.

Setting up

```
mkdir student-app && cd student-app
npm init -y
npm install express
```

5.2 Hello World in Express

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

app.get("/", (req, res) => {
  res.send("Hello, Express!");
});

app.listen(PORT, () => console.log(`Listening on http://localhost:${PORT}`));
```

5.3 Routing

Routing decides which function handles a request, based on the HTTP method and the URL path. Syntax: app.METHOD(path, handler).

```
app.get("/students", listStudents);          // GET    /students
app.post("/students", createStudent);        // POST   /students
app.put("/students/:id", replaceStudent);    // PUT    /students/7
app.patch("/students/:id", updateStudent);   // PATCH  /students/7
app.delete("/students/:id", deleteStudent);  // DELETE /students/7
```

Reading input from the request

Property	Contains	Example
req.params	Route parameters (:id)	/students/7 -> req.params.id = "7"
req.query	Query string values	/students?dept=CSE -> req.query.dept
req.body	Parsed request body (needs express.json())	{ name: "Anu" }
req.headers	HTTP headers	req.headers.authorization
req.cookies	Cookies (needs cookie-parser)	req.cookies.token
req.method, req.url, req.path	Method and URL information	GET, /students

Sending output with the response

Method	Purpose
res.send(data)	Sends text, HTML, buffer or object
res.json(obj)	Sends JSON with Content-Type application/json
res.status(code)	Sets the HTTP status code (chainable)
res.sendStatus(204)	Sets status and sends its text
res.redirect(url)	Redirects the client
res.render(view, data)	Renders a template (EJS, Pug)
res.sendFile(path)	Sends a file
res.cookie(name, value)	Sets a cookie
res.set(header, value)	Sets a response header

Grouping routes with express.Router

```
// routes/studentRoutes.js
const router = require("express").Router();
const ctrl = require("../controllers/studentController");

router.route("/").get(ctrl.getAll).post(ctrl.create);
router.route("/:id").get(ctrl.getOne).put(ctrl.replace).patch(ctrl.update).de
lete(ctrl.remove);

module.exports = router;

// app.js
app.use("/api/v1/students", require("./routes/studentRoutes"));
```

5.4 Middleware

Middleware functions are functions that have access to the request object (req), the response object (res) and the next function. Each middleware can run code, change req/res, end the

request–response cycle by sending a response, or pass control to the next middleware by calling next(). An Express application is essentially a chain of middleware, executed in the order in which it is registered.

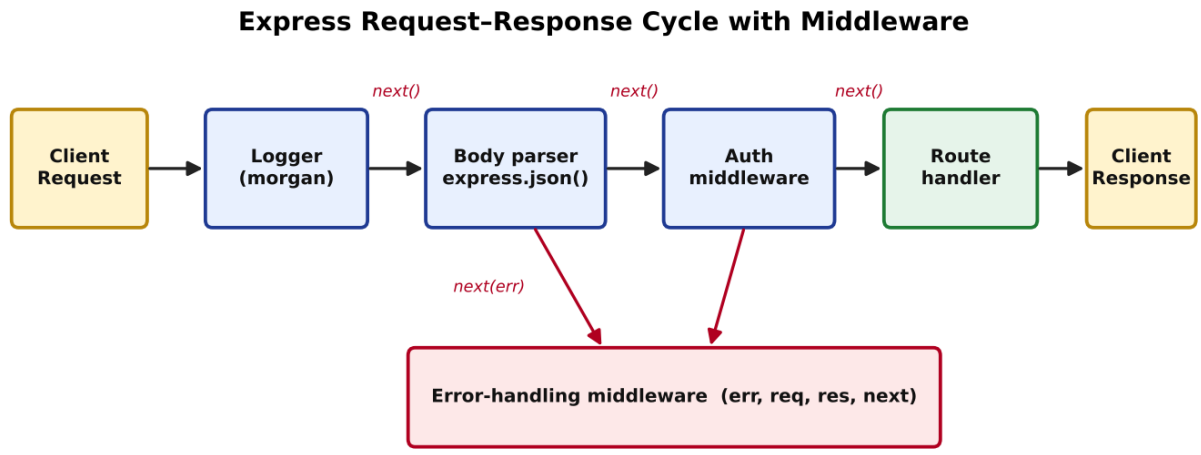


Fig. 7 Request–response cycle with middleware

```
// custom logging middleware
app.use((req, res, next) => {
  console.log(`${new Date().toISOString()}  ${req.method} ${req.url}`);
  next();                          // pass control; otherwise the request hangs
});
```

Type	Description	Example
Application-level	Bound to the app with app.use() or app.METHOD()	app.use(logger)
Router-level	Bound to an express.Router instance	router.use(auth)
Built-in	Shipped with Express	express.json(), express.urlencoded(), express.static()
Third-party	Installed from npm	morgan (logging), cors, helmet (security headers), cookie-parser
Error-handling	Has four arguments (err, req, res, next)	Central error handler

```
app.use(express.json());                            // parse JSON bodies
app.use(express.urlencoded({ extended: true }));    // parse HTML form bodies
app.use(express.static("public"));                  // serve files from /public
app.use(require("cors")());                         // allow cross-origin calls
```

Route-specific middleware can be placed before the handler, for example app.get("/admin", authMiddleware, adminHandler). Order matters: express.json() must be registered before the routes that read req.body.

5.5 Error Handling

```
// 404 handler – placed after all routes
app.use((req, res) => {
  res.status(404).json({ error: "Route not found" });
});

// central error-handling middleware – placed last, must have 4 parameters
app.use((err, req, res, next) => {
  console.error(err);
  res.status(err.status || 500).json({ error: err.message || "Server error"
});
});
```

Errors are passed to this handler by calling next(err) or by throwing in synchronous code. In asynchronous handlers use try/catch and call next(err) (Express 5 also forwards rejected promises automatically).

5.6 Serving Views with a Template Engine (EJS)

```
app.set("view engine", "ejs");              // views are looked up in /views
app.get("/students-page", (req, res) => {
  res.render("students", { students: ["Anu", "Ravi"] });
});

<!-- views/students.ejs -->
<ul>
  <% students.forEach(s => { %>
    <li><%= s %></li>
  <% }) %>
</ul>
```

5.7 Building a RESTful CRUD API (in-memory data)

The following complete program implements all CRUD operations for the student resource using an array in place of a database, which is enough to understand the logic.

```
const express = require("express");
const app = express();
app.use(express.json());

let students = [{ id: 1, name: "Anu", dept: "IT" }];
let nextId = 2;

// READ all (with optional filter)
app.get("/api/v1/students", (req, res) => {
  const { dept } = req.query;
  res.json(dept ? students.filter(s => s.dept === dept) : students);
});

// READ one
app.get("/api/v1/students/:id", (req, res) => {
  const s = students.find(x => x.id === Number(req.params.id));
  if (!s) return res.status(404).json({ error: "Student not found" });
  res.json(s);
});
```

```
// CREATE
app.post("/api/v1/students", (req, res) => {
  const { name, dept } = req.body;
  if (!name || !dept) return res.status(400).json({ error: "name and dept
required" });
  const s = { id: nextId++, name, dept };
  students.push(s);
  res.status(201).location(`/api/v1/students/${s.id}`).json(s);
});

// UPDATE (partial)
app.patch("/api/v1/students/:id", (req, res) => {
  const s = students.find(x => x.id === Number(req.params.id));
  if (!s) return res.status(404).json({ error: "Student not found" });
  Object.assign(s, req.body);
  res.json(s);
});

// DELETE
app.delete("/api/v1/students/:id", (req, res) => {
  const i = students.findIndex(x => x.id === Number(req.params.id));
  if (i === -1) return res.status(404).json({ error: "Student not found" });
  students.splice(i, 1);
  res.sendStatus(204);
});

app.listen(3000, () => console.log("API on port 3000"));
```

Testing the API

Use Postman, Thunder Client (VS Code) or curl:

```
curl http://localhost:3000/api/v1/students
curl -X POST http://localhost:3000/api/v1/students \
     -H "Content-Type: application/json" \
     -d '{"name":"Ravi","dept":"CSE"}'
curl -X PATCH http://localhost:3000/api/v1/students/2 \
     -H "Content-Type: application/json" -d '{"dept":"IT"}'
curl -X DELETE http://localhost:3000/api/v1/students/2
```

5.8 CRUD API with MVC and MongoDB (Mongoose)

In real applications the data is stored in a database and the code is split into model, controller and routes as shown in Section 3.5. Mongoose is an ODM (object–document mapper) for MongoDB – the NoSQL equivalent of the ORM concept: schemas map to collections and documents to objects.

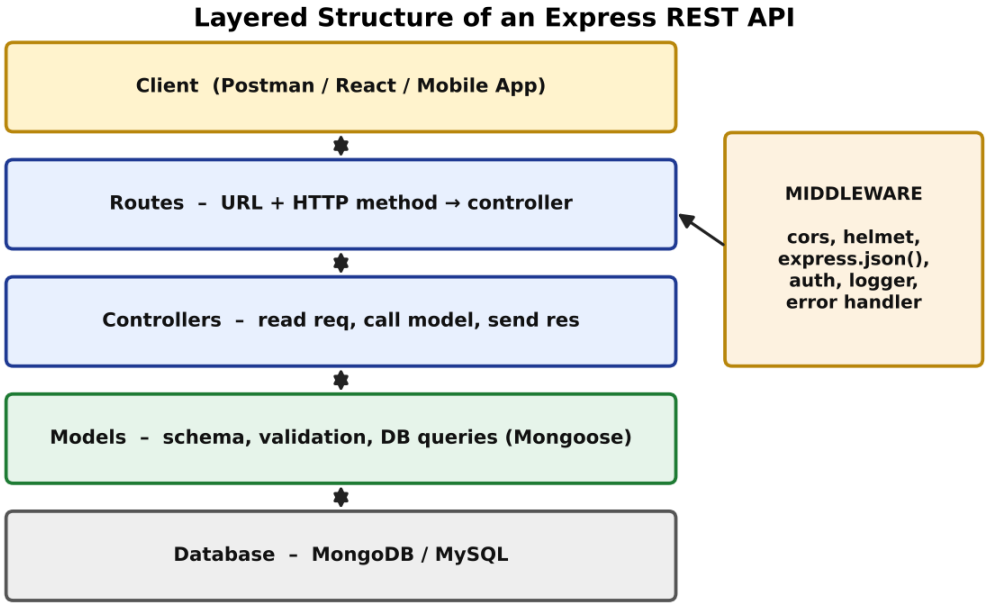


Fig. 8 Layered structure of an Express REST API

config/db.js

```
const mongoose = require("mongoose");
module.exports = async function connectDB() {
  await mongoose.connect(process.env.MONGO_URI ||
"mongodb://127.0.0.1:27017/college");
  console.log("MongoDB connected");
};
```

models/Student.js (Model)

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

const studentSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  dept: { type: String, required: true },
  year: { type: Number, min: 1, max: 4, default: 1 },
}, { timestamps: true });

module.exports = mongoose.model("Student", studentSchema);
```

controllers/studentController.js (Controller)

```
const Student = require("../models/Student");

exports.getAll = async (req, res, next) => {
  try {
    const filter = req.query.dept ? { dept: req.query.dept } : {};
    res.json(await Student.find(filter));
  } catch (e) { next(e); }
};

exports.getOne = async (req, res, next) => {
  try {
    const s = await Student.findById(req.params.id);
    if (!s) return res.status(404).json({ error: "Student not found" });
    res.json(s);
```

```
  } catch (e) { next(e); }
};

exports.create = async (req, res, next) => {
  try {
    const s = await Student.create(req.body);
    res.status(201).json(s);
  } catch (e) { e.status = 400; next(e); }      // validation error
};

exports.update = async (req, res, next) => {
  try {
    const s = await Student.findByIdAndUpdate(req.params.id, req.body,
                                              { new: true, runValidators:
true });
    if (!s) return res.status(404).json({ error: "Student not found" });
    res.json(s);
  } catch (e) { next(e); }
};

exports.remove = async (req, res, next) => {
  try {
    const s = await Student.findByIdAndDelete(req.params.id);
    if (!s) return res.status(404).json({ error: "Student not found" });
    res.sendStatus(204);
  } catch (e) { next(e); }
};
```

app.js and server.js

```
// app.js
const express = require("express");
const app = express();
app.use(express.json());
app.use("/api/v1/students", require("./routes/studentRoutes"));   // router
from Section 5.3
app.use((req, res) => res.status(404).json({ error: "Route not found" }));
app.use((err, req, res, next) =>
  res.status(err.status || 500).json({ error: err.message }));
module.exports = app;

// server.js
const app = require("./app");
const connectDB = require("./config/db");
connectDB().then(() => app.listen(3000, () => console.log("Server
started")));
```

For a relational database (MySQL/PostgreSQL) the same structure is used with an ORM such as Sequelize or Prisma, or a driver like mysql2; only the model layer changes.

5.9 Environment Variables and Configuration

Passwords, API keys and database URLs must not be written in the source code. They are placed in a .env file (excluded from version control) and loaded with the dotenv package: require("dotenv").config(); then read using process.env.MONGO_URI or process.env.PORT.

5.10 Securing the Express API

Concern	Typical solution
Authentication	JSON Web Token (jsonwebtoken) or sessions (express-session) with hashed passwords (bcrypt)
Secure HTTP headers	helmet middleware
Cross-origin requests	cors middleware with an allow-list of origins
Brute force / abuse	express-rate-limit
Input validation	express-validator, Joi or Mongoose schema validation
Injection attacks	Never build queries from raw input; sanitise request data

5.11 Express vs Servlet-based Development

Express (Node.js)	Servlet / JSP
JavaScript, single-threaded event loop	Java, multi-threaded container
Routes defined in code: app.get('/x', fn)	Mappings in web.xml or annotations
Middleware chain	Filters
npm packages, quick prototyping	Maven/Gradle, strongly typed, enterprise ready
res.json() for APIs	response.getWriter() / JSON libraries

6. Important Questions

6.1 Two-Mark Questions

1. What is REST?

An architectural style for web services in which resources are identified by URIs and manipulated through standard HTTP methods using representations such as JSON.

2. List the constraints of REST.

Client–server, stateless, cacheable, uniform interface, layered system and (optional) code on demand.

3. What does stateless mean in REST?

The server keeps no client state between requests; each request carries all the information required.

4. Differentiate PUT and PATCH.

PUT replaces the whole resource; PATCH changes only the supplied fields.

5. What is an idempotent method? Give examples.

A method whose repeated execution has the same effect as one execution – GET, PUT, DELETE.

6. Which status code is returned after successfully creating a resource?

201 Created, with a Location header.

7. What is CRUD? Map it to HTTP methods.

Create, Read, Update, Delete – POST, GET, PUT/PATCH and DELETE.

8. Define MVC.

A design pattern separating an application into Model (data), View (presentation) and Controller (control logic).

9. What is the role of the controller?

It receives requests, calls the model and returns a view or response.

10. What is Node.js?

A JavaScript runtime built on the V8 engine that uses an event-driven, non-blocking I/O model.

11. What is the event loop?

The mechanism that picks queued events and callbacks and runs them on a single thread, enabling asynchronous I/O.

12. What is npm?

Node Package Manager – a tool and registry to install and manage JavaScript packages.

13. Define middleware in Express.

A function with access to req, res and next that runs during the request–response cycle.

14. What is the use of express.json()?

A built-in middleware that parses JSON request bodies into req.body.

15. How are route parameters accessed in Express?

Through req.params, e.g. req.params.id for the route /students/:id.

16. What is Mongoose?

An ODM library that maps JavaScript objects to MongoDB documents using schemas and models.

6.2 Long-Answer Questions (13/16 marks)

1. Explain the architectural constraints of REST. Discuss the principles of designing a RESTful API with suitable URIs, methods and status codes.
2. Explain CRUD operations and show how they are mapped to HTTP methods and database operations. Differentiate PUT, PATCH and POST.
3. Describe the MVC architecture with a neat diagram. Explain how MVC is implemented in an Express application.
4. Explain the architecture of Node.js and the working of the event loop. How does it differ from a thread-per-request server?

5. Discuss asynchronous programming in Node.js using callbacks, promises and async/await with examples.
6. What is middleware in Express? Explain its types and the request–response cycle with examples.
7. Develop a RESTful API using Node.js and Express to perform CRUD operations on a student resource.
8. Explain how an Express application is connected to MongoDB using Mongoose and organised using routes, controllers and models.
9. Compare REST with SOAP, and Express-based development with Servlet/JSP-based development.

— End of Notes —