Web Technologies and Full Stack Development

Streams, Express and REST APIs

The notes for the lesson “Streams, Express and REST APIs”: chapters 6.3 to 6.5, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

`3 chapters` `Chapters 6.3 to 6.5`

About these notes

These are the notes for the University free n easy video lesson **Streams, Express and REST APIs** on Web Technologies and Full Stack Development: chapters 6.3 to 6.5, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.3 File system, streams and event emitter

Three Node tools sit between your code and the disk. The **fs** module opens, reads, writes and deletes files. A **stream** carries the same bytes in small chunks, so a 1 GB file never has to sit in RAM. **EventEmitter** is the base pattern both rely on: an object emits a named event, and the functions listening for that name run. You meet all three in a log-writing server, a file-upload handler, a backup script, or anywhere a program moves data bigger than its memory.

> You need callbacks, the event loop and microtasks from 6.1, plus arrow functions and promises from an earlier lesson.

The fs module: three flavours of the same call

Every file operation in Node lives in `fs`. The same job is offered in three shapes, and you pick one per call site.

- Callback: `fs.readFile(path, cb)`. The last argument is `(err, data)` and it runs later, after the disk answers.
- Synchronous: the name ends in `Sync`, as in `fs.readFileSync(path, 'utf8')`. It returns the value directly, throws on failure, and stops the single thread until the disk is done.

- Promises: `require('fs/promises')`, or `require('fs').promises`, gives the same functions returning promises, so `const text = await fs.readFile(p, 'utf8')` works.

CALL	WHAT IT DOES
readFile, writeFile	whole file in, whole file out
appendFile	adds bytes to the end
unlink	deletes a file
mkdir	creates a folder; {recursive:true} makes parents
readdir	lists names inside a folder
stat	size, mtime, isDirectory()

`readFile` returns a Buffer unless you pass an encoding. Pass `'utf8'` for text; keep the Buffer for images and video.

Join paths with `path.join('data','logs','app.txt')`. It picks the right separator for the operating system and collapses duplicate slashes. String concatenation with `'/'` breaks on Windows and happily produces `data//logs`.

Streams: move the file in chunks

A stream pushes data through in pieces instead of one big value. There are four kinds.

- **Readable**: a source, such as `fs.createReadStream`.
- **Writable**: a sink, such as `fs.createWriteStream`.
- **Duplex**: both at once, such as a TCP socket.
- **Transform**: a duplex that changes the bytes, such as gzip.

`fs.createReadStream('big.mp4', {highWaterMark: 64 * 1024})` opens the file and emits a `'data'` event per chunk. `highWaterMark` is the chunk size in bytes; the default is 64 KB = 65536 bytes.

`read.pipe(write)` wires the two together: each chunk is written to the destination, and the end of the source closes the destination. pipe returns the destination, so calls chain.

Backpressure makes this safe. When the write stream's internal buffer is full, `write()` returns false and pipe pauses the read stream until the writer emits `'drain'`. A fast disk never floods a slow disk or a slow network; the reader simply waits.

pipe does not forward errors. Attach handlers yourself, or use `stream.pipeline(read, write, cb)`, which destroys both sides and calls back once with the error.

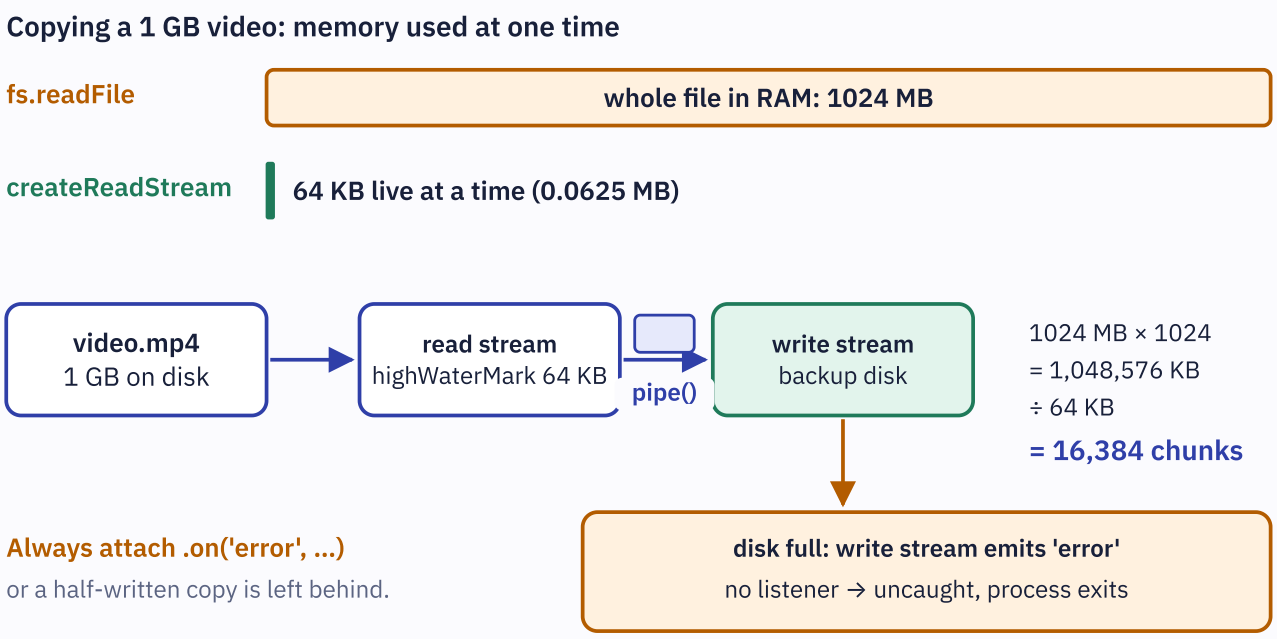

Figure 6.3.1 readFile holds the whole 1024 MB file in memory, while a read stream piped to a write stream keeps only one 64 KB chunk live and repeats 16,384 times. If the disk fills and no error listener is attached, the process exits.

EventEmitter: on, emit, once

`require('events')` gives the EventEmitter class. Streams, http servers and sockets are all EventEmitters underneath.

`emitter.on('name', fn)` registers a listener. `emitter.emit('name', arg)` calls every listener for that name at once, in registration order, passing arg. `once('name', fn)` runs the listener a single time and then removes it.

To build your own, extend the class. `class Job extends EventEmitter {}`, then `const j = new Job(); j.on('done', n => console.log(n)); j.emit('done', 42);` prints 42.

`'error'` is special. If you `emit('error', err)` and nobody is listening, Node throws err as an uncaught exception and the process exits. A stream that hits a disk failure emits `'error'`; with no handler, your server dies.

`'close'` is not `'error'`. It only means the file handle was released, and it can arrive after a clean end or after a failure. Never treat `'close'` as proof of success.

WORKED EXAMPLE

Question. A Node script copies a 1 GB video file to a backup disk. (a) How much RAM does `fs.readFile` need for this copy? (b) With `createReadStream` at `highWaterMark: 64 * 1024`, how many chunks are read and how much memory is live per step? (c) The backup disk fills up halfway. What happens if no 'error' listener is attached?

1. 1 GB = 1024 MB, the binary unit that `stat` and file sizes use.
2. readFile must hold the whole file before the write starts, so memory ≈ 1024 MB.
3. Convert to kilobytes: 1024 MB × 1024 KB/MB = 1,048,576 KB.
4. Chunks = 1,048,576 KB ÷ 64 KB = 16,384 chunks.
5. Memory live per step ≈ 64 KB = 0.0625 MB. Check the ratio: 1024 MB ÷ 0.0625 MB = 16,384, the same number of chunks.
6. (c) The write stream emits 'error'. With zero listeners on that event, Node throws it as an uncaught exception and the process exits, leaving a half-written copy.

Answer. readFile needs about 1024 MB of RAM; the stream reads 16,384 chunks of 64 KB and keeps about 64 KB live at a time; a missing 'error' listener ends the process.

EXAM TRAPS

- **readFile is asynchronous**, so it does not block the event loop. Its problem is memory: the whole file is buffered. readFileSync is the one that blocks.
- **pipe() does not carry errors.** If the destination disk fails, only the write stream's 'error' reports it. Listen on it, or use stream.pipeline.
- **'error' is not 'close'.** 'close' only means the handle was released, and an unhandled 'error' throws and kills the process.
- **Missing encoding.** readFile without 'utf8' returns a Buffer, so logging text shows hexadecimal bytes.
- **on() when you meant once().** A completion handler registered with on() fires again the next time that same emitter emits the event.

CHECK YOURSELF

– `REMEMBER` **Name the three flavours of an fs call and one difference between them.** `2 marks`

Callback (gets (err, data) later), Sync (returns the value, throws on error, blocks the thread), and promises (returns a promise, used with await).

– REMEMBER **What does highWaterMark control, and what is its default?** 2 marks

The chunk size in bytes for a stream. Default 64 KB = 65536 bytes.

– UNDERSTAND **Why does piping a 1 GB file use far less memory than readFile?** 3 marks

readFile buffers the entire file (about 1024 MB). A stream keeps roughly one chunk (about 64 KB) live and pauses while the writer's buffer is full, which is backpressure.

– UNDERSTAND **Why is the 'error' event special on an EventEmitter?** 3 marks

With no listener registered, emit('error', e) throws e as an uncaught exception, and the process exits instead of continuing silently.

– APPLY **A 512 MB file is read with highWaterMark set to 64 KB. How many chunks?** 4 marks

512 × 1024 = 524,288 KB; 524,288 ÷ 64 = 8,192 chunks.

– APPLY **With `const fs = require('fs/promises')`, write the call that appends 'ok' to log.txt and the call that reads it back as text.** 4 marks

await fs.appendFile('log.txt', 'ok'); then const t = await fs.readFile('log.txt', 'utf8'); — the 'utf8' argument makes t a string, not a Buffer.

– ANALYZE **Four requests each run readFile on a 1.5 GB file at the same moment. Estimate the memory and say what you would change.** 5 marks

4 × 1536 MB = 6144 MB ≈ 6 GB of RAM, which usually ends in an out-of-memory crash. (Above 2 GiB, readFile refuses the file outright with ERR_FS_FILE_TOO_LARGE.) Replace readFile with createReadStream and pipe the chunks to the response, so each request holds about 64 KB at a time.

BEYOND THE SYLLABUS

Not for the exam. Real code uses `stream.pipeline` and `stream/promises` instead of hand-wired pipe plus listeners, because they destroy both sides on failure. Node streams are async iterables, so `for await (const chunk of readStream)` works and reads like a loop. The WHATWG web streams standard now carries the same idea into browsers, Deno and Node, so a Transform written for one can often run in the others. Search for 'Node stream backpressure' and 'web streams TransformStream' next. Sending files over HTTP with streams belongs to 6.4.

6.4 Express.js: routing, middleware and static files

Express is a small package that wraps Node's built-in HTTP server. It lets you define routes and middleware as ordinary JavaScript functions. Every request walks down an ordered chain of functions until one function sends a response. This gives clean URL matching, body parsing, static file serving and error handling without writing the raw server code by hand. Express is used for REST APIs, server-rendered pages and the back end of full-stack apps.

> You need: 6.1 for Node's `http` server and callbacks, and HTTP methods (an earlier lesson), URLs and status codes.

The app and the middleware chain

`express` is a package that wraps Node's `http` server. You create an application object with `const app = express()` and start it with `app.listen(3000)`. Port 3000 is a number, not a string. Every incoming request enters a list of functions called middleware. Each function receives `(req, res, next)`: `req` is the request, `res` is the response, and `next` is a function that passes control to the following middleware. A function that sends a response ends the chain for that request.

Middleware runs in registration order. A logger can print the method and URL, then call `next()`. A body parser can read the request body, then call `next()`. A route handler matches a method and path, reads inputs, and calls `res.send` or `res.json`. If no function sends a response, the client waits until the server times out.

Express is one of several Node.js frameworks. Others include Fastify, Koa and NestJS. Express is the one exams use because its routing and middleware are small enough to trace by hand.

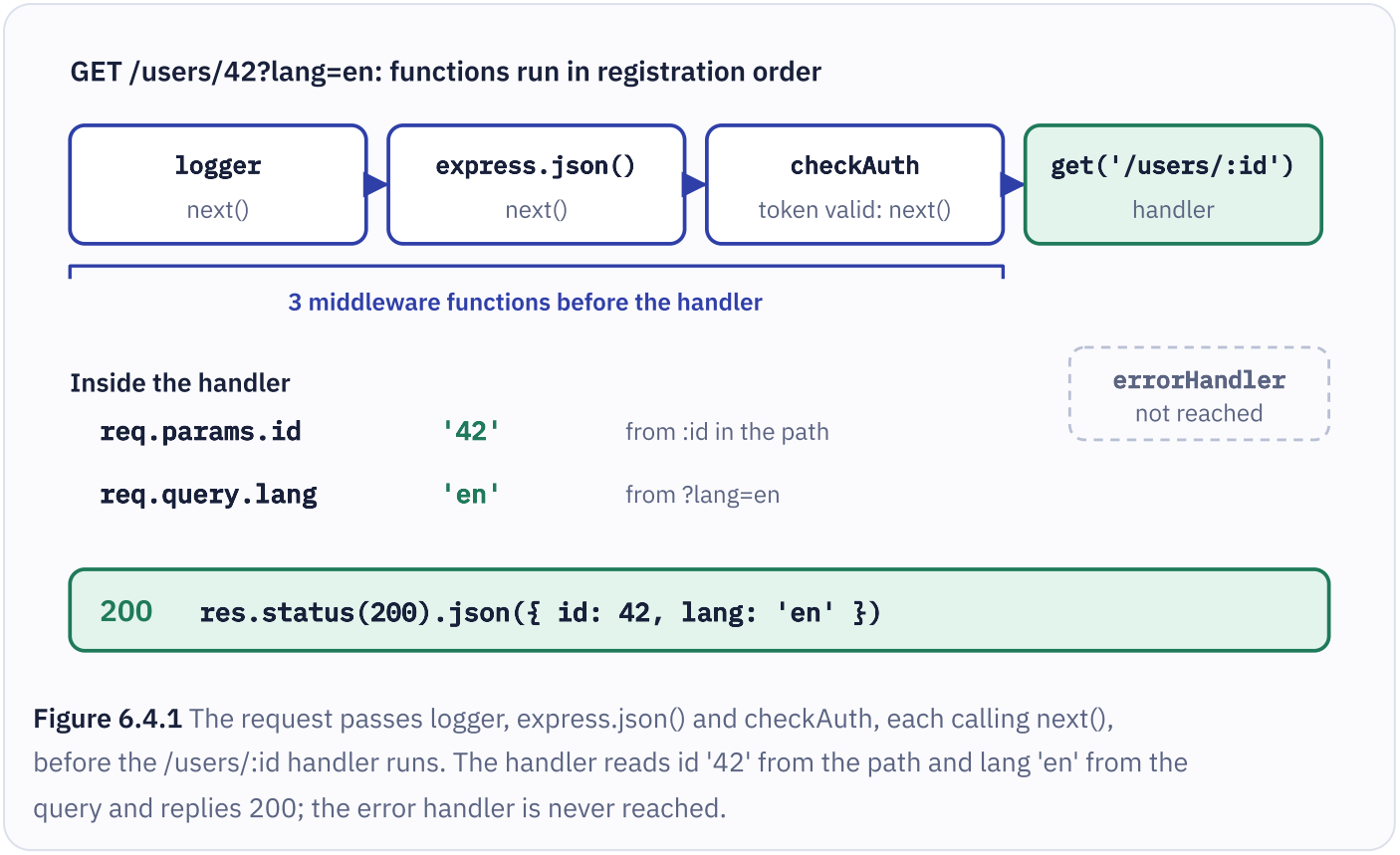

Figure 6.4.1 The request passes logger, express.json() and checkAuth, each calling next(), before the /users/:id handler runs. The handler reads id '42' from the path and lang 'en' from the query and replies 200; the error handler is never reached.

Routes, parameters and query strings

Routes map an HTTP method and a URL path to a handler. The methods are `app.get`, `app.post`, `app.put`, `app.patch` and `app.delete`. The handler signature is `(req, res) => { ... }`. A path can contain a named parameter: `app.get('/users/:id', handler)`. A request for `/users/42` gives `req.params.id` equal to the string `'42'`. Parameters are always strings; convert with `Number(req.params.id)` before arithmetic.

Query strings come after `?` and are read from `req.query`. For `/users/42?lang=en`, `req.query.lang` is `'en'`. A repeated key can become an array. Query values are also strings.

Response helpers shorten common work. `res.send('text')` sends text or a buffer. `res.json(obj)` sets the JSON content type and sends JSON. `res.status(404)` sets the status code and returns `res`, so `res.status(404).json({ error: 'not found' })` works. `res.redirect('/login')` sends a 302 redirect by default.

Middleware, routers and static files

`app.use(fn)` registers middleware for every request, or for a mount path when written as `app.use('/api', fn)`. Order matters. If `express.json()` is registered above a route, `req.body` is filled before the handler runs. If it is registered below the route, `req.body` is `undefined` inside the handler.

Body parsers: `express.json()` reads JSON request bodies; `express.urlencoded({ extended: true })` reads HTML form bodies. The flag `extended: true` allows nested objects. Both parsers call `next()` after parsing.

`express.Router()` makes a mini-app. You attach routes to the router, then mount it with `app.use('/api', router)`. A router route `/users` mounted at `/api` answers `/api/users`.

Static files are served with `app.use(express.static('public'))`. A request for `/logo.png` returns `public/logo.png` if it exists. Put static middleware above routes only when you want files to win; otherwise below.

Error middleware takes four arguments: `(err, req, res, next)`. Express identifies it by that arity. A three-argument function is ordinary middleware and is never called with an error. Database calls inside handlers are covered in 6.6.

WORKED EXAMPLE

Question. An Express app registers, in this order: `app.use(logger)`, `app.use(express.json())`, `app.use(checkAuth)`, `app.get('/users/:id', handler)`, `app.use(errorHandler)`. A client with a valid token sends `GET /users/42?lang=en`. State `req.params.id`, `req.query.lang`, whether `express.json()` runs before the handler, the response status, and how many middleware functions run before the handler.

```
const express = require('express');
const app = express();
const logger = (req, res, next) => { console.log(req.method, req.url); next(); };
const checkAuth = (req, res, next) => { if (req.headers.authorization) next(); else res
app.use(logger);
app.use(express.json());
app.use(checkAuth);
app.get('/users/:id', (req, res) => {
  const id = Number(req.params.id);
  res.status(200).json({ id: id, lang: req.query.lang });
});
app.use((err, req, res, next) => { res.status(500).json({ error: 'server error' }); });
app.listen(3000);
```

1. The path is `/users/42` and the query string is `lang=en`.
2. The route pattern `/users/:id` matches. The `:id` segment captures `42`. `req.params.id` is the string `'42'`, whose numeric value is 42.
3. `req.query.lang` is the string `'en'`.
4. Registration order is logger, `express.json()`, `checkAuth`, route, error handler. Both `express.json()` and `checkAuth` sit above the route, so `express.json()` runs before the handler.
5. The middleware before the handler are logger, `express.json()` and `checkAuth`: 3 functions.
6. The token is valid, `checkAuth` calls `next()`, the handler runs and sends `res.status(200).json({ id: 42, lang: 'en' })`. The status is 200.

Dry run

STAGE	REQ.PARAMS.ID	REQ.QUERY.LANG	NEXT()	STATUS
logger	—	—	yes	—
express.json()	—	—	yes	—
checkAuth	—	—	yes	—

STAGE	REQ.PARAMS.ID	REQ.QUERY.LANG	NEXT()	STATUS
handler	'42'	'en'	no	200
res.json	—	—	—	200

Output

```
{"id":42,"lang":"en"}
```

Answer. `req.params.id` is the string `'42'` with numeric value 42, `req.query.lang` is `'en'`, `express.json()` runs before the handler, 3 middleware functions run first, and the response status is 200.

EXAM TRAPS

- Forgetting `next()` inside a middleware leaves the request hanging with no response; the browser spins until timeout.
- Moving `express.json()` below a route makes `req.body` undefined inside that route's handler.
- An error handler written with three arguments is treated as ordinary middleware and is never called for errors; it needs `(err, req, res, next)`.
- Confusing `req.params` with `req.query`: `/users/:id` fills params, while `?lang=en` fills query.
- Forgetting that route and query values are strings: `'42' + 1` gives `'421'`, not 43.

CHECK YOURSELF

– `REMEMBER` **What does `next()` do in an Express middleware function?** 2 marks

It passes control to the next middleware or route handler in the registration order.

– `UNDERSTAND` **Why must `app.use(express.json())` be registered before a POST route that reads `req.body`?** 3 marks

Middleware runs in registration order. The body parser fills `req.body` and calls `next()`. If it is below the route, the route handler runs first and sees `req.body` as undefined.

– APPLY **The route is `app.get('/books/:isbn', handler)`. A request is `GET /books/978?lang=en`. What are `req.params.isbn` and `req.query.lang`?** 3 marks

`req.params.isbn` is the string `'978'`; `req.query.lang` is the string `'en'`.

– APPLY **An app registers `app.get('/ping', handler)` and then `app.use(express.json())`. A client sends `GET /ping` with a JSON body. What is `req.body` inside the handler?** 3 marks

`req.body` is undefined because `express.json()` is registered below the route, so the handler runs before the parser.

– ANALYZE **A developer writes `app.use((err, req, res) => { res.status(500).send('fail'); })`. Why does Express not call it when a route throws an error?** 4 marks

Express identifies error middleware by its four arguments. This function has three, so Express treats it as ordinary middleware. It needs the signature `(err, req, res, next)`.

– ANALYZE **A file `public/logo.png` exists, but `app.use(express.static('public'))` is registered after a catch-all route. A request for `/logo.png` gets the catch-all response. Explain the result and the fix.** 4 marks

Middleware runs in order. The catch-all route runs before the static middleware and sends a response, so the file is never served. Register `express.static('public')` above the catch-all route.

– REMEMBER **Name four Node.js web frameworks.** 2 marks

Express, Fastify, Koa and NestJS.

BEYOND THE SYLLABUS

Not for the exam. Express 5 changes some error handling for async handlers, and newer projects often choose Fastify for speed or NestJS for structure. Production apps add Helmet for headers, rate limiting, request logging and schema validation. Search for Express error handling, Express Router best practices, Helmet, and OpenAPI with Express to go further.

6.5 RESTful APIs: HTTP methods, JSON and CRUD

REST is a way to design a web API around things, not actions. A book, a user and an order are resources. Each resource gets a URL made of plural nouns, and the HTTP method says what to do with it. POST /books creates a book, GET /books/7 reads book 7, and JSON carries the data in both directions. Browsers, mobile apps and React pages all call such APIs. The style exists because one small set of methods and status codes lets any client talk to any server without a private vocabulary.

> You need: Express routing and handlers from 6.4, and the HTTP methods and status codes from an earlier lesson.

Resources, URLs and HTTP methods

A resource is a named thing: a book, a customer, an order. A collection is a resource that holds many items. /books is the collection; /books/7 is one item inside it. Use plural nouns and no verbs. /getBooks is wrong because GET already says read; /books?author=gupta is right for filtering.

Five methods cover CRUD. CRUD means Create, Read, Update, Delete.

METHOD	URL	ACTION	BODY
GET	/books/7	read one	none
POST	/books	create	full JSON
PUT	/books/7	replace	full JSON
PATCH	/books/7	partial update	changed fields
DELETE	/books/7	remove	none

The id belongs in the path. The body holds the data, not the identity. POST /books returns the new item and a Location header. A header is a name-value line in a request or response. PUT replaces the whole item; a field you leave out is cleared or set to its default. PATCH changes only the fields you send. A common exam mistake is using PUT for a partial update.

CRUD routes for /books and their success codes

Action	Request	Status	Safe	Idempotent
Create	`POST /books`	**201 Created**	no	no
List	`GET /books`	**200 OK**	yes	yes
Read	`GET /books/7`	**200 OK**	yes	yes
Replace	`PUT /books/7`	**200 OK**	no	yes
Delete	`DELETE /books/7`	**204 No Content**	no	yes
Missing	`GET /books/99`	**404 Not Found**	yes	yes

The same POST sent twice:



Figure 6.5.1 Each CRUD action has its own method and success code: 201 for create, 200 for list, read and replace, 204 for delete and 404 for a missing book. Only GET is safe; GET, PUT and DELETE are idempotent, so two identical POSTs make books 7 and 8.

Safe, idempotent and status codes

A method is safe if the client is not asking the server to change any resource. GET is safe, even though the server may still write a log line. A method is idempotent if sending the same request twice leaves the same final state as sending it once. GET, PUT and DELETE are idempotent. PUT and DELETE are not safe because they change state. POST is neither: two POST /books calls create two books, with ids 7 and 8. PUT /books/7 twice still leaves one book 7. DELETE /books/7 twice leaves it deleted; the second call may return 404, but the state is the same. PATCH is not safe, and it is not guaranteed idempotent, because the patch document may describe a change relative to the current value (for example 'increment stock by 1').

Status codes are three-digit numbers that summarise the result.

CODE	MEANING	WHEN TO USE
200	OK	request worked, body has data
201	Created	new resource; send Location
204	No Content	success, no body, common for DELETE
400	Bad Request	bad JSON or missing field
401	Unauthorized	not logged in or bad token

CODE	MEANING	WHEN TO USE
403	Forbidden	logged in but not allowed
404	Not Found	no such resource
409	Conflict	clashes with existing state, e.g. duplicate email
500	Internal Server Error	server bug

Use 201 on create, not 200. Use 204 on delete when there is nothing to return.

JSON, collections and stateless requests

JSON is a text format for objects and arrays. Content-Type: application/json tells the server that the request body is JSON. Accept: application/json asks the server for a JSON response. Express needs the express.json() middleware to read JSON bodies, then res.json() sends JSON back. See 6.4.

GET /books returns an array; GET /books/7 returns one object. Query strings, the part after ?, filter and page: /books?author=gupta&page=2&limit=10. State limit and page in the API documentation. Version the path: /api/v1/books. A new incompatible shape goes to /api/v2.

REST is stateless: each request carries everything needed, such as the token. The server does not remember the previous request. This lets any server in a pool answer any request. Database work is in 6.6; protecting routes is in 6.8.

WORKED EXAMPLE

Question. Design the CRUD routes for a books API and give the success status for each. Then say how many books exist after POST /books is sent twice with the same JSON, and name the safe and idempotent methods.

1. Create: POST /books with JSON. The database returns id 7. Respond 201 Created, Location: /books/7, and the new book.
2. List: GET /books returns 200 OK with an array of books.
3. Read one: GET /books/7 returns 200 OK with one object. GET /books/99 for a missing book returns 404 Not Found.
4. Replace: PUT /books/7 with the full JSON object returns 200 OK and the replaced book.
5. Delete: DELETE /books/7 returns 204 No Content with an empty body.
6. Safe methods: GET only. Idempotent methods: GET, PUT and DELETE. POST is neither safe nor idempotent, and PATCH is not safe either. Two identical POSTs create id 7 and id 8, so 2 books exist.

Dry run

CALL	RESULT
POST /books	201, id 7
GET /books/7	200
POST twice	2 books

Answer. Create returns 201, list and read return 200, a missing read returns 404, replace returns 200, delete returns 204, and two identical POSTs leave 2 books. Only GET is safe; GET, PUT and DELETE are idempotent; POST is neither.

EXAM TRAPS

- Using PUT for a partial update; that is PATCH. PUT replaces the whole resource.
- Writing verbs into the URL such as /getBooks; use the method plus a plural noun, such as GET /books.
- Returning 200 instead of 201 on create. Create should return 201 with a Location header.
- Putting the id in the JSON body instead of the path. The URL /books/7 identifies the resource.
- Saying POST is idempotent or DELETE is not. Two POSTs create two books; two DELETEs leave the same final state.

CHECK YOURSELF

– REMEMBER **Which HTTP method creates a resource, and which status code should it return?** 2 marks

POST creates a resource and should return 201 Created, with a Location header.

– UNDERSTAND **Why is POST not idempotent while PUT is?** 3 marks

Two POST /books calls create two new books with different ids. Two PUT /books/7 calls replace the same book 7, so the final state is the same.

– APPLY **A client sends DELETE /books/7 twice. The first returns 204. What does the second probably return, and what is the final state?** 3 marks

The second probably returns 404 Not Found because book 7 is already gone. The final state is still that book 7 does not exist.

– ANALYZE **A route PUT /books/7 accepts only {title}. What is wrong, and what should change?** 4 marks

PUT should replace the whole book, so a title-only body is a partial update. Change the route to PATCH /books/7 for partial updates, or send the full object with PUT.

– REMEMBER **Which status code means not logged in, and which means logged in but not allowed?** 2 marks

401 Unauthorized means not logged in or a bad token. 403 Forbidden means logged in but not allowed.

– APPLY **Write the URL to list page 2 with 10 books per page, filtered by author gupta, API version 1.** 3 marks

/api/v1/books?author=gupta&page=2&limit=10

– ANALYZE **Why should the id not be in the JSON body for PUT /books/7?** 3 marks

The URL already identifies book 7. An id in the body can disagree with the path, so the server would have two identities for one resource. The path is the authority.

BEYOND THE SYLLABUS

Not for the exam. In industry, OpenAPI describes REST endpoints so client code and docs can be generated. Idempotency keys let a client retry a POST safely; a UPI payment retry can otherwise create two debits. GraphQL and gRPC solve overlapping problems with different trade-offs. Search for OpenAPI specification, idempotency keys, JSON:API and HTTP caching next.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
stream	A channel that moves data in small chunks instead of one big buffer, so memory stays flat and work starts early.	6.3
pipe()	readable.pipe(writable) forwards each chunk and ends the destination, but it does not forward errors to the destination.	6.3
EventEmitter	Node's on(event, listener) and emit(event, args) pattern; fs streams, http servers and your own objects all use it.	6.3
middleware	A function (req, res, next) registered with app.use that runs in order and either calls next() or sends the response.	6.4
express.static	Built-in middleware that serves every file in a folder directly, with no route written for it.	6.4
REST	An API style where a URL names a resource (/books/7) and the HTTP method names the action on it.	6.5

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- UNDERSTAND **What is middleware in Express? Describe the order in which application-level, router-level, built-in and error-handling middleware execute, and how next() controls the flow.** 12 marks

  signature (req, res, next); registration order equals execution order; next() passes on, next('route') skips, next(err) jumps to error middleware; four-argument error handler placed last; res.send ends the chain.

- UNDERSTAND **Define REST and show how the HTTP methods map to CRUD operations on a /books collection, including the status codes each should return.** 12 marks

  resources as nouns, methods as verbs; statelessness and uniform interface; GET /books and GET /books/7 with 200 and 404; POST /books with 201 and Location header; PUT and PATCH, DELETE with 204; idempotency of GET, PUT, DELETE.

Apply

- APPLY **Write an Express application that serves a public folder as static files and exposes GET, POST, PUT and DELETE routes on /books backed by an in-memory array, returning correct status codes and JSON bodies.** 15 marks

  express() and app.use(express.json()); app.use(express.static('public')); route parameters req.params.id; validation returning 400; 201 on create, 404 when the id is missing, 204 on delete; listen on process.env.PORT.

- APPLY **Write a Node program that copies a 2 GB file using fs.createReadStream and fs.createWriteStream with pipe(). Explain what happens if the source fails and how to fix it.** 12 marks

  chunked transfer, flat memory; pipe() forwards data and calls end() on destination; errors on the readable are not forwarded by pipe alone; attach 'error' on each stream or use stream.pipeline / finished; destroy the destination so no half-written file is left.

Glossary (terms used in this lesson)

Backpressure	A stream telling its writer to slow down because the internal buffer is full, so memory does not grow without limit.
Idempotent	A request that can be repeated with the same server state resulting, which is true of GET, PUT and DELETE but not POST.
next()	The third argument of an Express middleware; calling it hands the request to the next matching function in the chain.
req.body	The parsed request body; it stays undefined unless express.json() or a similar parser middleware ran first.

Sources and further reading

Textbooks

- Mario Casciaro, Luciano Mammino, *Node.js Design Patterns*: Chapters on the Node platform, the event loop, streams and modules.
- Ethan Brown, *Web Development with Node and Express*: Routing, middleware, static files, REST APIs and deployment.
- Evan M. Hahn, *Express in Action*: Express routing, middleware and building APIs.
- Shannon Bradshaw, Eoin Brazil, Kristina Chodorow, *MongoDB: The Definitive Guide*: Documents, collections, queries and the MongoDB shell.

Open web resources

Each link below was opened by the book script on 11 Nov 2026 and returned a page; the pages were not read line by line.

- Node.js API documentation: nodejs.org/docs/latest/api
- Express web framework documentation: expressjs.com
- npm documentation: docs.npmjs.com
- MongoDB manual: www.mongodb.com/docs/manual
- OWASP Top Ten web application security risks: owasp.org/www-project-top-ten
- Docker get started guide: docs.docker.com/get-started

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Web Technologies and Full Stack Development (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.