

Express & REST APIs

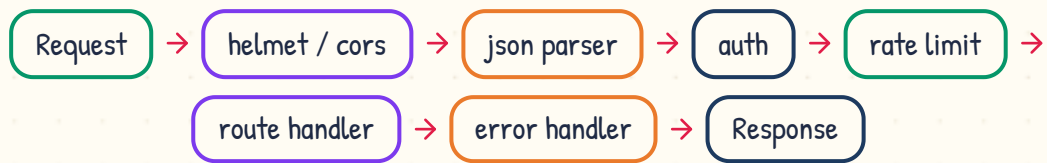
Middleware, routing, validation, errors, REST design, GraphQL

1

Express basics and middleware

Definition

Express is a minimal Node.js web framework. A request flows through a **chain of middleware** `(req, res, next)` until one sends a response.



A small production-shaped API

```
import express from 'express'
import helmet from 'helmet'
import cors from 'cors'
import rateLimit from 'express-rate-limit'

const app = express()
app.use(helmet()) // security headers
app.use(cors({ origin: process.env.WEB_ORIGIN }))
app.use(express.json({ limit: '100kb' }))
app.use('/api', rateLimit({ windowMs: 60_000, limit: 100 }))

app.get('/api/users/:id', async (req, res, next) => {
  try {
    const user = await User.findById(req.params.id)
    if (!user) return res.status(404).json({ error: 'User not found' })
    res.json(user)
  } catch (err) {
    next(err) // to the error handler
  }
})

// error handler: 4 arguments, registered last
app.use((err, req, res, next) => {
  console.error(err)
  res.status(err.status ?? 500).json({ error: err.expose ? err.message :
  'Internal error' })
})

app.listen(process.env.PORT ?? 3000)
```

- ✎ Order matters: middleware runs in the order it is registered.
- ✎ Call `next()` to continue or `next(err)` to jump to the error handler; forgetting both hangs the request.
- ✎ Express 5 forwards rejected promises from async handlers automatically; in Express 4 wrap them or use try/catch.
- ✎ Organise with `express.Router()` per resource, plus controller/service layers.

Method & path	Action	Success code
GET /users	list (with ?page=&limit=&sort=)	200
GET /users/42	read one	200 / 404
POST /users	create	201 + Location header
PUT /users/42	replace	200 / 204
PATCH /users/42	partial update	200
DELETE /users/42	delete	204

Code	Meaning
400	bad request / validation failed
401	not authenticated
403	authenticated but not allowed
404	not found
409	conflict (duplicate email)
422	semantically invalid
429	too many requests
500	server error
503	unavailable

- ✎ Nouns in URLs (/orders/7/items), HTTP verbs for actions; plural resources.
- ✎ **Idempotent**: GET, PUT, DELETE (same effect when repeated). POST is not → use an **Idempotency-Key** for payments.
- ✎ Version the API (/v1/...), paginate lists (cursor-based for large data), use one consistent error shape.



REST is stateless: every request carries its auth (token or cookie).

Validate with Zod, protect with middleware

```
import { z } from 'zod'

const CreateUser = z.object({
  name: z.string().min(1).max(80),
  email: z.string().email(),
  age: z.number().int().min(18).optional(),
})

const validate = (schema) => (req, res, next) => {
  const result = schema.safeParse(req.body)
  if (!result.success) return res.status(400).json({ error: 'Invalid input',
details: result.error.flatten() })
  req.body = result.data
  next()
}

const requireAuth = (req, res, next) => {
  const token = req.cookies.access_token
  try {
    req.user = jwt.verify(token, process.env.JWT_SECRET)
    next()
  } catch {
    res.status(401).json({ error: 'Not authenticated' })
  }
}

const requireRole = (...roles) => (req, res, next) =>
  roles.includes(req.user.role) ? next() : res.status(403).json({ error:
'Forbidden' })

router.post('/users', requireAuth, requireRole('admin'), validate(CreateUser),
createUser)
```

Common mistake

Returning raw error messages or stack traces to clients leaks internals. Log details on the server and send a generic message plus a request id.

Uploads, cron jobs and real-time

- File uploads: `multer` (multipart) with size and type limits; store in S3, keep only the key in the DB; or let the client upload directly with a pre-signed URL.
- Cron jobs: `node-cron` for simple schedules; with many instances use a queue (BullMQ) or a lock so the job runs once.
- Real-time: WebSockets via `ws` or Socket.IO; Server-Sent Events for one-way server → client updates.
- Webhooks (e.g. payments): verify the signature, respond fast, process asynchronously, and make the handler idempotent.

	REST	GraphQL
Endpoints	many (one per resource)	one (<code>/graphql</code>)
Shape of data	fixed by the server	client asks for exact fields
Over/under-fetching	common	avoided
Caching	easy with HTTP caching	needs client cache (Apollo, urql)
Watch out for	many round trips	N+1 queries (use DataLoader), query cost limits

Revision

Middleware chain with `next()`, error handler last, validate every input (Zod), auth → role checks, correct status codes, idempotent methods, versioning and pagination, safe error messages.

Q. What is middleware in Express?

A. A function (req, res, next) in the request pipeline that can read/modify the request, end the response, or call next() to continue.

Q. How does Express handle errors?

A. Call next(err) (or throw in async handlers on Express 5) to reach an error-handling middleware with four parameters, registered last.

Q. PUT vs PATCH?

A. PUT replaces the whole resource; PATCH changes only the given fields.

Q. Which HTTP methods are idempotent?

A. GET, PUT, DELETE (and HEAD, OPTIONS). POST is not, so payments use an idempotency key.

Q. 401 vs 403?

A. 401: not authenticated (missing or invalid credentials). 403: authenticated but not allowed.

Q. How do you validate request bodies?

A. With a schema library such as Zod or Joi in middleware; reject with 400 and clear field errors.

Q. What makes an API RESTful?

A. Resources identified by URLs, standard HTTP methods and status codes, stateless requests, and representations such as JSON.

Q. REST vs GraphQL?

A. REST has many fixed-shape endpoints and simple HTTP caching; GraphQL has one endpoint where clients choose fields, avoiding over-fetching but needing N+1 and cost protection.

Q. How do you paginate large lists?

A. Limit/offset for small data; cursor-based (after=lastId) for large or changing data.

Q. How do you run a cron job safely with many instances?

A. Use a distributed lock or a queue scheduler (e.g. BullMQ) so only one instance runs each job.