

Express & REST APIs interview questions

25 questions · 11 easy · 12 medium · 2 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is Express, and why use it instead of the plain `http` module?

Easy



#express #basics

Express is a minimal, unopinionated **web framework** for Node. It wraps Node's `http` server and adds:

- **routing** by method and path, with URL params (`/users/:id`)
- a **middleware** pipeline for cross-cutting concerns
- helpers like `res.json()`, `res.status()`, `req.query`, `req.params`
- a huge ecosystem (helmet, cors, multer, passport)

Without it you would parse URLs, bodies and headers by hand. It is **unopinionated**: it does not dictate folder structure, ORM or validation, so you choose those yourself.

Express 5 (stable since 2024) adds native handling of rejected promises in route handlers.

? **Likely follow-up:** How does Express compare to Fastify or NestJS?

2

How does the middleware chain and `next()` work in Express?

Easy



#middleware

Express keeps an ordered list of middleware and routes. For each request it runs the matching functions **in the order they were registered**. Each one receives `(req, res, next)` and must either:

- **end the response** (`res.send`, `res.json`), or
- call `next()` to pass control to the next function

`next('route')` skips to the next matching route, and `next(err)` skips straight to error-handling middleware.

If a middleware does neither, the request **hangs** until the client times out. Order matters: `express.json()` must come before routes that read `req.body`, and auth before protected routes.

Example

```
const app = express()

app.use(express.json()) // 1. parse body for every
request
app.use((req, res, next) => { // 2. log
  console.log(req.method, req.path)
  next()
})
app.use('/admin', requireAdmin) // 3. only for /admin/*

app.get('/users/:id', getUser) // 4. route handler ends the
response
```

? Likely follow-up: What happens if you call `next()` after sending a response?

3

How does error-handling middleware work in Express?

Medium 🔥🔥 #errors #middleware

Error middleware is a function with **four arguments**: `(err, req, res, next)`. Express recognises it by the argument count, so you must declare all four even if you do not use `next`.

It runs when a handler throws synchronously or calls `next(err)`. Register it **after all routes**, so it catches errors from everything above.

A good central handler:

- maps known errors to status codes (validation → 400, not found → 404)
- logs unexpected errors with the request ID
- hides stack traces and internals from clients in production

Example

```
class AppError extends Error {
  constructor(status, message) { super(message); this.status = status }
}

app.get('/users/:id', async (req, res) => {
  const user = await db.users.find(req.params.id)
  if (!user) throw new AppError(404, 'User not found')
  res.json(user)
})

app.use((err, req, res, next) => {
  const status = err.status ?? 500
  if (status === 500) logger.error(err)
  res.status(status).json({ error: status === 500 ? 'Internal error' : err.message })
})
```

4

How do you handle errors in async route handlers?

Medium



#errors #async

In **Express 4**, a rejected promise inside an `async` handler is not caught: the error middleware never runs, the request hangs and you get an unhandled rejection. Fixes:

- wrap every handler in `try/catch` and call `next(err)`
- use a small `asyncHandler` wrapper (below) or the `express-async-errors` package

In **Express 5**, rejected promises are forwarded to `next(err)` automatically, so a plain `throw` inside an async handler works.

Either way, keep one central error middleware instead of sending error responses from every route.

Example

```
// Express 4 helper
const asyncHandler = (fn) => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next)

app.get('/orders', asyncHandler(async (req, res) => {
  const orders = await Order.find({ userId: req.user.id })
  res.json(orders)
})))
```

5

What is the difference between `req.params`, `req.query` and `req.body`?

Easy



#routing #request

- `req.params`: named parts of the **path**. For route `/users/:id` and URL `/users/42`, `req.params.id` is `'42'`. Use them to **identify a resource**.
- `req.query`: the **query string**. `/users?role=admin&page=2` gives `{ role: 'admin', page: '2' }`. Use for filtering, sorting, pagination.
- `req.body`: the **request payload** of POST/PUT/PATCH. Empty unless a body parser like `express.json()` runs first.

All values from params and query are **strings** (or arrays), so convert and validate them before use.

Example

```
app.use(express.json())

// PATCH /users/42?notify=true body: { "name": "Asha" }
app.patch('/users/:id', (req, res) => {
  const id = Number(req.params.id) // 42
  const notify = req.query.notify === 'true'
  const { name } = req.body // 'Asha'
  res.json({ id, notify, name })
})
```

6

What is REST? What makes an API RESTful?

Easy



#rest #api design

REST (Representational State Transfer) is an **architectural style** for APIs over HTTP. Key ideas:

- **resources** identified by URLs, named with nouns: `/users/42/orders`, not `/getOrders`
- **HTTP methods** express the action: GET reads, POST creates, PUT/PATCH update, DELETE removes
- **stateless**: each request carries everything needed (e.g. a token); the server keeps no session between requests
- **representations**: clients get JSON (or other formats) describing the resource
- correct **status codes** and **cacheable** responses
- a **uniform interface** (HATEOAS in the strict definition)

In practice most "REST" APIs are **resource-oriented JSON over HTTP** and skip HATEOAS.

Likely follow-up: How would you design the URLs for users, their orders, and order items?

7

Explain the main HTTP methods. Which are safe and which are idempotent?

Easy



#http #rest

- **GET**: read. Safe and idempotent.
- **POST**: create or trigger an action. Neither safe nor idempotent: two calls can create two orders.
- **PUT**: replace a resource. Idempotent.
- **PATCH**: partial update. Not guaranteed idempotent (e.g. "increment by 1").
- **DELETE**: remove. Idempotent: deleting twice leaves the same state (the second call may return 404).
- **HEAD/OPTIONS**: metadata and CORS preflight. Safe.

Safe = does not change state. **Idempotent** = **repeating the request has the same effect as doing it once**. This matters for retries: clients and proxies can retry idempotent requests safely. For POST, use an **Idempotency-Key** header so retries do not double-charge.

🔗 Likely follow-up: How would you implement an Idempotency-Key for payments?

8

Which HTTP status codes should an API use, and when?

Easy



#http #rest

- **200 OK** success with a body; **201 Created** after POST (add a **Location** header); **204 No Content** success with no body (DELETE)
- **301/308** permanent redirect; **304 Not Modified** for cache validation
- **400 Bad Request** malformed or invalid input; **422** is sometimes used for validation errors
- **401 Unauthorized** **not authenticated** (missing or bad token); **403 Forbidden** authenticated but not allowed
- **404 Not Found**; **409 Conflict** (duplicate email, version conflict); **429 Too Many Requests**
- **500** unexpected server error; **502/503/504** upstream or availability problems

Never return 200 with `{ error: ... }` in the body: clients, monitoring and caches rely on the code.

9

PUT vs PATCH?

Easy



#http #rest

PUT replaces the **whole resource** with the body you send. Fields you leave out are reset or removed. It is idempotent, and can also create a resource at a client-chosen URL.

PATCH applies a **partial update**: only the fields you send change. Formats include a plain JSON object (JSON Merge Patch) or a list of operations (JSON Patch).

Rule of thumb: **PUT sends the full object, PATCH sends only changes**. Most CRUD apps use PATCH for edit forms.

Example

```
const user = { id: 1, name: 'Asha', email: 'a@x.io', role: 'admin' }

const put = (current, body) => ({ id: current.id, ...body })           //
replace
const patch = (current, body) => ({ ...current, ...body })           //
merge

console.log(put(user, { name: 'Asha K' }))    // email and role are gone
console.log(patch(user, { name: 'Asha K' }))  // only name changed
```

10

How do you paginate an API? Offset vs cursor pagination?

Medium



#pagination #api design

Offset pagination: `?page=3&limit=20` → `LIMIT 20 OFFSET 40`.

- simple, supports "jump to page 7" and total counts
- gets **slow** on deep pages (the DB still scans skipped rows) and items can be **skipped or duplicated** when rows are inserted or deleted between requests

Cursor (keyset) pagination: `?after=<cursor>&limit=20`, where the cursor encodes the last item's sort key → `WHERE (created_at, id) < (...) ORDER BY created_at DESC, id DESC LIMIT 20`.

- **fast at any depth** using an index, and stable while data changes
- no random page jumps

Use offset for small admin tables, cursor for feeds, infinite scroll and large datasets.

Always cap `limit`.

Example

```
const rows = Array.from({ length: 7 }, (_, i) => ({ id: i + 1 }))
const encode = (id) => 'cur_' + id // real APIs: base64 of
the sort key
const decode = (c) => Number(c.slice(4))

function list(after, limit = 3) {
  const start = after ? rows.findIndex((r) => r.id === decode(after)) + 1
  : 0
  const items = rows.slice(start, start + limit)
  const last = items.at(-1)
  return { items: items.map((r) => r.id), next: items.length === limit ?
  encode(last.id) : null }
}
const p1 = list()
const p2 = list(p1.next)
console.log(p1.items, p2.items, list(p2.next).items)
```

11

How do you validate request input in an Express API?

Medium



#validation #security

Never trust the client: validate **params, query and body** at the edge, before business logic runs.

Use a schema library such as **zod**, Joi or express-validator (or JSON Schema with Ajv):

- check types, required fields, formats (email, UUID), ranges and lengths
- **strip unknown fields** so clients cannot set `role: 'admin'` (mass assignment)
- return **400** with clear field-level messages
- **use the parsed output**, not the raw `req.body`, so types are coerced and safe

Validation at the API boundary does not replace database constraints (unique, not null); keep both.

Example

```
const { z } = require('zod')

const validate = (schema) => (req, res, next) => {
  const result = schema.safeParse(req.body)
  if (!result.success) return res.status(400).json({ errors:
result.error.flatten() })
  req.body = result.data      // typed, unknown keys stripped
  next()
}

const CreateUser = z.object({ email: z.string().email(), name:
z.string().min(1).max(80) })
app.post('/users', validate(CreateUser), createUser)
```

12

What is CORS, and how do you configure it in Express?

Medium



#cors #security

By default, the browser's **same-origin policy** stops JavaScript on `https://app.com` from reading responses from `https://api.com`. **CORS** lets the server opt in, using response headers like `Access-Control-Allow-Origin`.

For "non-simple" requests (JSON body, custom headers, PUT/DELETE), the browser first sends an `OPTIONS` **preflight** request to ask permission.

In Express, use the `cors` package with an explicit allowlist. With cookies you need `credentials: true` and you **cannot use the * wildcard** origin.

CORS is enforced by browsers only; it does not stop curl or server-to-server calls, so it is not access control.

Example

```
const cors = require('cors')

app.use(cors({
  origin: ['https://app.example.com', 'https://admin.example.com'],
  credentials: true,           // allow cookies
  methods: ['GET', 'POST', 'PATCH', 'DELETE'],
  maxAge: 600,                // cache preflight for 10 minutes
}))
```

🔗 **Likely follow-up:** Why do you see a CORS error in the browser but the request works in Postman?

13

How do you implement rate limiting?

Medium



#rate limiting #security

Rate limiting caps how many requests a client (by IP, user ID or API key) can make in a time window, protecting against abuse, brute force and overload.

Common algorithms:

- **fixed window:** count per minute; simple but allows bursts at window edges
- **sliding window:** smoother counting over the last N seconds
- **token bucket:** tokens refill at a steady rate; each request spends one, so short bursts are allowed

In Express, `express-rate-limit` works for one instance; with several instances, `store counters in Redis` so all instances share them. Return **429** with a `Retry-After` header. Use stricter limits on login and password-reset routes.

Example

```
function tokenBucket(capacity, refillPerSec) {
  let tokens = capacity, last = Date.now()
  return () => {
    const now = Date.now()
    tokens = Math.min(capacity, tokens + ((now - last) / 1000) *
refillPerSec)
    last = now
    if (tokens < 1) return false
    tokens -= 1
    return true
  }
}
const allow = tokenBucket(3, 1)
console.log([1, 2, 3, 4, 5].map(() => (allow() ? 200 : 429)))
```

14

What does it mean for an API to be stateless, and why does it matter?

Easy



#rest #scaling

A stateless server keeps **no client session in its own memory** between requests. Every request carries what is needed to handle it: an auth token or session ID, plus the parameters.

State still exists, but it lives in shared stores (database, Redis), not inside one server process.

Why it matters:

- **any instance can handle any request**, so you can add servers behind a load balancer freely
- restarts and deploys do not log users out
- easier caching and retries

A server-side session in Redis is fine; a session in a process's memory is what breaks horizontal scaling.

15

How do you structure a larger Express API?

Medium



#architecture

Separate layers so that each has one job:

- **routes**: map URLs and methods to controllers, attach middleware
- **controllers**: handle HTTP: read `req`, call a service, send the response
- **services**: business logic, with **no knowledge of req or res**, so they are reusable and easy to unit test
- **repositories / models**: database access
- **middleware**: auth, validation, error handling
- **config**: env parsing in one place

Many teams group by **feature** (`users/`, `orders/`) with these layers inside each, rather than one giant `controllers/` folder. Keep `app.js` (building the app) separate from `server.js` (`listen`) so tests can import the app.

Example

```
// users/users.controller.js
exports.create = async (req, res) => {
  const user = await usersService.register(req.body)
  res.status(201).location(`/users/${user.id}`).json(user)
}

// users/users.service.js
exports.register = async ({ email, name }) => {
  if (await usersRepo.findByEmail(email)) throw new AppError(409, 'Email taken')
  return usersRepo.insert({ email, name })
}
```

? Likely follow-up: How would you test the service without a database?

16

REST vs GraphQL: what are the trade-offs?

Medium 🔥🔥 #graphql #api design

REST: many endpoints, each returning a fixed shape.

- simple, uses HTTP caching (CDN, ETag) naturally, easy to monitor per endpoint
- can **over-fetch** (too many fields) or **under-fetch** (several round trips for related data)

GraphQL: one endpoint; the client sends a query describing **exactly the data it needs**, typed by a schema.

- great for complex UIs and many client types (web, mobile)
- harder HTTP caching, risk of expensive nested queries (needs depth limits, cost analysis), **N+1** problems solved with DataLoader

Choose REST for simple CRUD and public APIs; GraphQL when many clients need different views of connected data.

17

How do you version a REST API?

Easy 🔥🔥 #api design

Versioning lets you make **breaking changes** without breaking existing clients. Options:

- **URL path:** `/v1/users`, the most common; obvious and easy to route and cache

- **header:** `Accept: application/vnd.myapp.v2+json` or a custom `API-Version` header; cleaner URLs, harder to test in a browser
- **query param:** `?version=2`; simple but easy to forget

Good practice: **avoid breaking changes where you can** (add fields, do not rename or remove them), so versions are rare. When you do release one, announce a deprecation period (e.g. `Deprecation` / `Sunset` headers) and run both versions side by side.

18

How do you cache API responses with `Cache-Control` and ETags?

Hard



#caching #http

Cache-Control tells browsers and CDNs whether and how long to cache:

- `public, max-age=300`: anyone may cache for 5 minutes
- `private`: only the user's browser (personal data)
- `no-store`: never cache (sensitive data); `no-cache`: cache, but revalidate first

ETag is a fingerprint of the response. The client sends it back in `If-None-Match`; if unchanged, the server replies **304 Not Modified** with no body, **saving bandwidth** (but not the server lookup). Express sets weak ETags on `res.send` by default.

For heavy queries, also cache on the server (Redis) with a TTL and invalidate on writes.

Example

```
app.get('/products/:id', async (req, res) => {
  const product = await getProduct(req.params.id)
  res.set('Cache-Control', 'public, max-age=60, stale-while-
revalidate=300')
  res.json(product) // Express adds an ETag and answers 304 when it
matches
})

app.get('/me', auth, (req, res) => {
  res.set('Cache-Control', 'private, no-store')
  res.json(req.user)
})
```

19

How do you handle file uploads in Express?

Medium



#uploads

Uploads arrive as `multipart/form-data`, which `express.json()` does not parse. Use **multer** (or busboy) to stream the parts.

Good practice:

- set **size and count limits** to avoid memory and disk exhaustion
- check the real file type (magic bytes), not just the extension or `Content-Type` header
- generate your own file names; never use the client's path
- store files in **object storage** (S3, GCS), not on the server's disk, so instances stay stateless
- for large files, let the client **upload directly to storage using a presigned URL**, and keep the API out of the data path

Example

```
const multer = require('multer')
const upload = multer({
  storage: multer.memoryStorage(),
  limits: { fileSize: 5 * 1024 * 1024, files: 1 },
})

app.post('/avatar', auth, upload.single('avatar'), async (req, res) => {
  if (!['image/png', 'image/jpeg'].includes(req.file?.mimetype)) {
    return res.status(400).json({ error: 'PNG or JPEG only' })
  }
  const url = await s3Upload(`avatars/${req.user.id}.png`,
    req.file.buffer)
  res.status(201).json({ url })
})
```

20

How do you approach logging in an Express API?

Easy



#logging #observability

- use a **structured logger** (pino, winston) that writes JSON, not `console.log`; it is faster and searchable
- log every request (method, path, status, duration) with middleware such as `pino-http`

- attach a **request ID** (or trace ID) to every log line so you can follow one request across services
- use levels: `debug`, `info`, `warn`, `error`
- **never log secrets**: passwords, tokens, full card numbers, personal data; redact these fields
- write to stdout and let the platform ship logs to a central system (ELK, Datadog, CloudWatch)

Pair logs with metrics and tracing (OpenTelemetry) for full observability.

21

What is HATEOAS?

Medium 🔥 #rest

HATEOAS (Hypermedia As The Engine Of Application State) is the REST constraint where responses **include links to related actions and resources**, so clients discover what they can do next instead of hard-coding URLs.

For example, an order response might include links to `self`, `cancel` and `items`, and the `cancel` link only appears when cancelling is allowed.

In practice few APIs implement it fully. Knowing what it is, and that it is the part of REST most "RESTful" APIs skip, is usually enough for interviews. Pagination `next` links are its most common real-world form.

Example

```
{
  "id": 81,
  "status": "pending",
  "_links": {
    "self": { "href": "/orders/81" },
    "items": { "href": "/orders/81/items" },
    "cancel": { "href": "/orders/81/cancel", "method": "POST" }
  }
}
```

What are webhooks, and how do you receive them safely?

Hard



#webhooks #security

A webhook is an **HTTP callback**: instead of you polling a provider ("is the payment done?"), the provider POSTs an event to your URL when something happens (Stripe, GitHub).

Receiving them safely:

- **verify the signature** (HMAC of the raw body with a shared secret) and reject old timestamps to stop forged and replayed events
- respond **2xx quickly**, then process in a background job; slow handlers cause timeouts and retries
- make handling **idempotent**: providers retry and may send the same event twice, so store processed event IDs
- do not assume events arrive in order

Sending them: sign payloads, retry with exponential backoff, and log delivery attempts.

Example

```
const crypto = require('node:crypto')

app.post('/webhooks/pay', express.raw({ type: 'application/json' })),
(req, res) => {
  const expected = crypto.createHmac('sha256',
process.env.WEBHOOK_SECRET)
    .update(req.body).digest('hex')
  const given = Buffer.from(req.get('X-Signature') ?? '')
  if (given.length !== expected.length ||
    !crypto.timingSafeEqual(given, Buffer.from(expected))) return
res.sendStatus(401)
  queue.add('payment-event', JSON.parse(req.body))
  res.sendStatus(200)
})
```

? **Likely follow-up:** Why must the signature be computed on the raw body rather than parsed JSON?

23

What should an API error response look like?

Easy



#errors #api design

Consistent and machine-readable, so clients can handle errors generically:

- the right **HTTP status code**
- a stable **error code** for programs (`EMAIL_TAKEN`), plus a human message
- **field-level details** for validation errors
- a **request ID** to match with server logs

A standard option is **RFC 9457 Problem Details** (`application/problem+json`), with `type`, `title`, `status`, `detail`).

Never leak stack traces, SQL or internal hostnames: `detailed in logs, generic to clients` for 500 errors.

Example

```
{
  "type": "https://api.example.com/errors/validation",
  "title": "Invalid request body",
  "status": 400,
  "errors": [{ "field": "email", "message": "Must be a valid email" }],
  "requestId": "req_8f2a1c"
}
```

24

How do you test an Express API?

Medium



#testing

- **Unit tests** for services and helpers, with dependencies (repositories, external APIs) mocked. Fast and many.
- **Integration tests** that send real HTTP requests to the app using **supertest**, against a real test database (often in Docker or Testcontainers). These catch wiring bugs: middleware order, validation, status codes.
- **Contract tests** or an OpenAPI spec check so clients and server agree on shapes.

Export the `app` without calling `listen`, so supertest can use it directly. Reset or seed the DB between tests so they are `independent and repeatable`.

Example

```
const request = require('supertest')
const app = require('../src/app')

test('POST /users rejects an invalid email', async () => {
  const res = await request(app).post('/users').send({ email: 'nope',
    name: 'A' })
  expect(res.status).toBe(400)
  expect(res.body.errors.fieldErrors.email).toBeDefined()
})
```

25

Which security middleware and settings would you add to a production Express app?

Medium 🔥🔥 #security #production

- `helmet()`: sets security headers (CSP, HSTS, `X-Content-Type-Options`, frame protection)
- `cors` with an explicit origin allowlist
- body size limits: `express.json({ limit: '100kb' })`
- rate limiting, stricter on auth routes
- input validation on every route
- `app.disable('x-powered-by')` (helmet does this) to hide the framework
- `app.set('trust proxy', 1)` behind a load balancer, so `req.ip` and secure cookies work correctly
- secure cookie flags (`httpOnly`, `secure`, `sameSite`)
- HTTPS everywhere, with TLS usually terminated at the load balancer

And keep dependencies patched: **most real breaches come from known vulnerabilities**, not clever attacks.