LECTURE 18

Request Handling and Routing

From one route to dozens: reading a request fully, and organizing a growing Express project.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. The anatomy of an HTTP request: method, URL, headers, and body
2. Routes for GET, POST, PUT, PATCH, and DELETE
3. Route parameters and query strings
4. Parsing a JSON request body with express.json()
5. Serving static files, and organizing routes with express.Router
6. Handling unmatched routes (404) and centralizing error handling

ANATOMY OF A REQUEST

Every HTTP Request Has Four Parts

```
POST /api/books HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Accept: application/json

{"title": "Clean Code", "author": "Robert C. Martin"}
```

Method (POST), URL/path (/api/books), headers (Content-Type, Accept), and body (the JSON payload).

Where to Find Each Part in Express

Request part	Where to find it
Method	req.method
Path	req.path (or req.url, which includes the query string)
Headers	req.headers (e.g. req.headers['content-type'])
Body	req.body (requires middleware)
Route parameters	req.params
Query string	req.query

What Is Routing?

1. Matching an incoming request's method and path to the code that should handle it

2. Express gives you one method per HTTP verb, each taking a path and a handler function

The Five Methods You'll Use Constantly

Method	Typical purpose	Example
GET	Read/fetch data -- should never change anything	Fetch a list of books
POST	Create something new	Add a new book
PUT	Replace an existing resource entirely	Overwrite a book's full record
PATCH	Partially update an existing resource	Change only a book's price
DELETE	Remove a resource	Delete a book

One app.METHOD() per Verb

```
app.get('/api/books', (req, res) => res.send('All books'));
app.post('/api/books', (req, res) => res.send('A new book was created'));

app.put('/api/books/:id', (req, res) =>
  res.send(`Book ${req.params.id} was fully replaced`));
app.patch('/api/books/:id', (req, res) =>
  res.send(`Book ${req.params.id} was partially updated`));
app.delete('/api/books/:id', (req, res) =>
  res.send(`Book ${req.params.id} was deleted`));
```

Method + Path = Endpoint

NOTE

The same path (/api/books/:id) can have completely different handlers depending on the method. Express matches on the combination of method AND path together, not the path alone -- this pairing is often called an endpoint.

GET Requests Should Be Safe

TIP

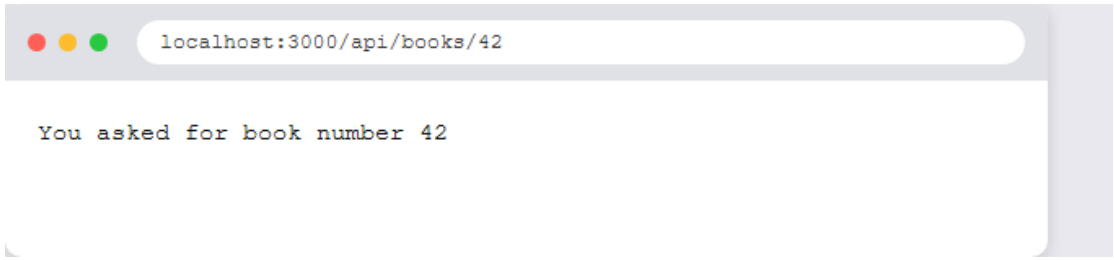
Calling a GET request should never change data on the server. This isn't enforced by the language -- it's a convention other developers, browsers, and tools all rely on. Breaking it can cause surprising bugs, since browsers may pre-fetch or cache GET requests.

Route Parameters vs. Query Strings

1. Two different ways of getting extra information out of a URL -- important not to confuse them
2. Route parameters: named placeholders built into the path itself, marked with a colon (:) -- identify WHICH resource
3. Query strings: optional ?key=value pairs at the end of a URL -- filter or modify a request

The Exact Code, Live

```
// URL: /api/books/42
app.get('/api/books/:id',
  (req, res) => {
  console.log(req.params.id);
  // "42"
  res.send(
    `You asked for book
    number ${req.params.id}`
  );
});
```



More Than One Parameter

```
// URL: /api/authors/12/books/42
app.get('/api/authors/:authorId/books/:bookId', (req, res) => {
  console.log(req.params.authorId); // "12"
  console.log(req.params.bookId);   // "42"
});
```

Filtering, Sorting, Pagination

```
// URL: /api/books?genre=fiction&sort=title&page=2
app.get('/api/books', (req, res) => {
  console.log(req.query.genre); // "fiction"
  console.log(req.query.sort);  // "title"
  console.log(req.query.page);  // "2" (always a string!)
  res.send('Filtered book list');
});
```

Everything Arrives as a String

WARNING

req.params and req.query always arrive as strings, even if they look like a number. req.query.page is "2", not 2. If you need a number, convert it explicitly: Number(req.query.page) or parseInt(req.query.page, 10).

Route Parameter vs. Query String

	Route parameter	Query string
Syntax	/books/:id -> /books/42	/books?genre=fiction
Purpose	Identifies a specific resource	Filters/modifies a request
Required?	Usually required to match the route	Usually optional
Access in Express	req.params	req.query

What Is Middleware?

1. A function that runs DURING the request-response cycle, before your route handler, typically to inspect or transform the request

2. By default, Express does NOT automatically parse a JSON body -- req.body would be undefined without extra setup

3. express.json() is a built-in middleware that parses an incoming body as JSON and attaches it to req.body

Registering express.json()

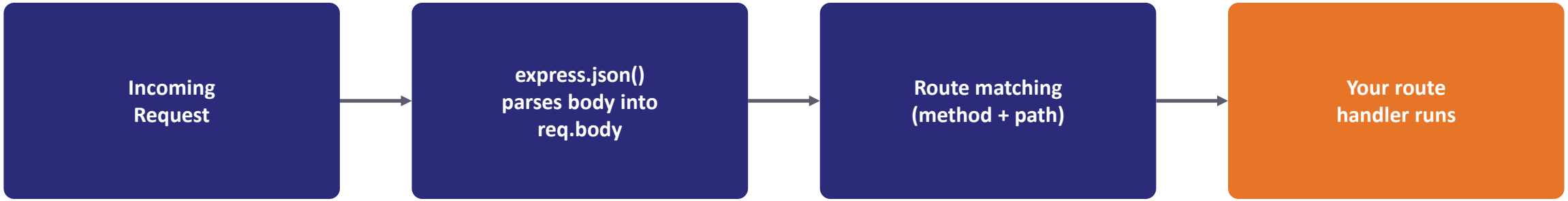
```
const express = require('express');
const app = express();

app.use(express.json()); // apply to every incoming request

app.post('/api/books', (req, res) => {
  const { title, author } = req.body;
  res.status(201).send(`Created "${title}" by ${author}`);
});
```

app.use(express.json()) must be called BEFORE any route handler that expects to read req.body.

Where Middleware Fits



app.use(...) registers middleware that runs for (by default) every incoming request, before Express tries to match it to a route.

The #1 Beginner Mistake

WARNING

If you forget app.use(express.json()) and the client sends a JSON body, req.body will be undefined, and destructuring it (const { title } = req.body) will throw an error. If req.body seems empty, check this first.

Files That Don't Change Per-Request

1. Images, CSS stylesheets, client-side JavaScript bundles, and so on
2. express.static() is a built-in middleware that serves an entire folder of files automatically
3. You don't need to write a route yourself -- Express matches the URL to the file

express.static('public')

```
app.use(express.static('public'));

// my-project/
// |-- index.js
// `-- public/
//     |-- logo.png
//     `-- style.css
//
// A request to /logo.png is served from public/logo.png automatically.
```

Routers and Controllers

1. Keeping every route in one file becomes hard to manage as an app grows past a handful of routes
2. express.Router() -- a mini, standalone Express app you define in a separate file and plug into your main app
3. Router file: defines URL paths and methods; Controller file: contains the actual handler logic

routes/books.js

```
const express = require('express');
const router = express.Router();
const booksController = require('../controllers/booksController');

router.get('/', booksController.getAllBooks);
router.get('/:id', booksController.getBookById);
router.post('/', booksController.createBook);

module.exports = router;
```

controllers/booksController.js

```
exports.getAllBooks = (req, res) => {
  res.send('List of all books');
};

exports.getBookById = (req, res) => {
  res.send(`Book with id ${req.params.id}`);
};

exports.createBook = (req, res) => {
  res.status(201).send(`Created book: ${req.body.title}`);
};
```

index.js -- Mounting the Router

```
const express = require('express');
const app = express();
const booksRouter = require('./routes/books');

app.use(express.json());
app.use('/api/books', booksRouter); // mount at this path prefix

app.listen(3000);
```

router.get('/:id', ...) inside routes/books.js therefore actually handles GET /api/books/:id -- the prefix is added automatically.

What Happens With No Matching Route?

1. Express needs to be told explicitly what to do, or it sends a generic, unhelpful default response

2. Place a catch-all handler AFTER all your other routes -- Express checks routes in the order they're defined

A Catch-All 404 Handler

```
// ...all your real routes go above this...

app.use((req, res) => {
  res.status(404).send('Sorry, that page was not found.');
});
```

Centralized Error-Handling Middleware

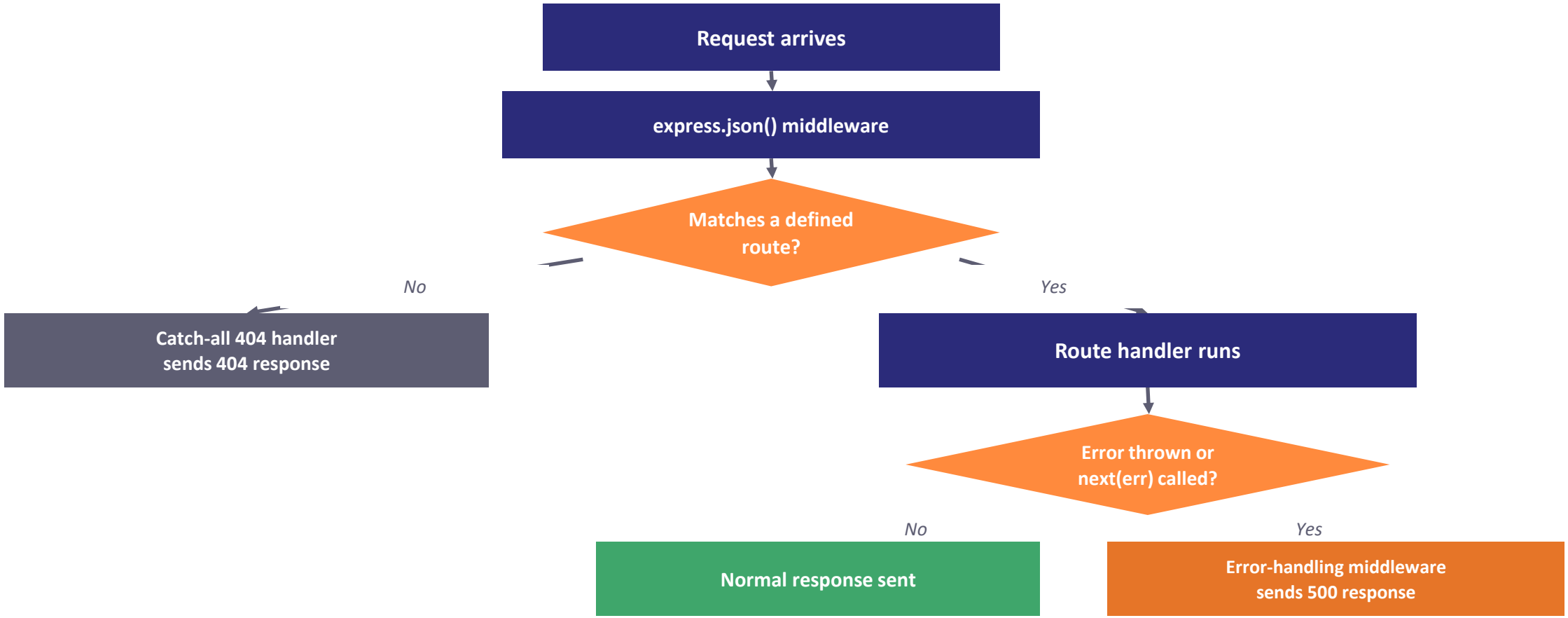
1. Instead of try/catch and custom error responses in every route, Express supports error-handling middleware in one place
2. You recognize it because it takes FOUR parameters instead of the usual two or three, with err first
3. next(err) skips all remaining normal routes/middleware and jumps straight to the nearest error handler

next(err) and the 4-Parameter Handler

```
app.get('/api/books/:id', (req, res, next) => {
  try {
    if (req.params.id === '0') throw new Error('Invalid book id');
    res.send(`Book ${req.params.id}`);
  } catch (err) {
    next(err); // pass along to the error-handling middleware
  }
});

app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong on our end.');
});
```

Request Lifecycle: Match, Handle, or 404



Order Matters

TIP

Middleware and routes are checked top-to-bottom in the order you register them with app.use()/app.get()/etc. Your 404 handler and error-handling middleware must always come LAST, or they'll swallow requests meant for routes defined below them.

KEY TAKEAWAYS

Lecture 18 in Six Points

1. Every HTTP request has a method, URL/path, headers, and optionally a body -- Express exposes all of these on req.
2. Express provides one method per HTTP verb (get/post/put/patch/delete); routing matches on method AND path together.
3. Route parameters (req.params, from :name) identify a specific resource; query strings (req.query, from ?key=value) filter or modify a request -- both always arrive as strings.
4. express.json() middleware must be registered with app.use() before your routes can read a JSON body via req.body.
5. express.static('folder') serves static files directly, without individual routes; express.Router() splits routes into separate files, typically paired with controllers.
6. Register a catch-all 404 handler and a 4-argument error-handling middleware LAST, after all your real routes.