LECTURE 19

HTTP Status Codes and Headers

The three-digit codes and metadata headers that tell every client exactly what happened — and how to set them yourself in Express.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Understand the five status code families (1xx–5xx) and what each broadly means
2. Learn the exact meaning and correct use of the most common specific status codes
3. Identify key request headers: Accept, Content-Type, Authorization, User-Agent
4. Identify key response headers: Content-Type, Cache-Control, Set-Cookie, Location
5. Understand content negotiation — how client and server agree on a response format

OVERVIEW

Every Response Carries Two Things

1. A three-digit status code that summarizes what happened, at a glance
2. A set of headers — key-value metadata about the request or response
3. Setting these correctly in your own Express routes is what makes your API behave the way browsers, other servers, and mobile apps expect

Five Families, By First Digit

The first digit tells you, at a glance, what kind of outcome occurred — before you even read the rest of the response.

STATUS CODE FAMILIES

What Each Family Means

Family	Meaning	At Fault
1xx Informational	Request received and understood; processing continues (rare in typical apps)	—
2xx Success	Request received, understood, and accepted successfully	—
3xx Redirection	Further action needed, usually following a different URL	—
4xx Client Error	Bad syntax, invalid data, or something the client did wrong	Client
5xx Server Error	The server failed on an otherwise-valid request	Server

Where to Start Debugging

NOTE

A useful habit: 4xx means "look at what the client sent" — a bad request, a missing field, an unauthenticated user. 5xx means "look at your server code" — an unhandled exception, a database that's down, a bug. This distinction guides where you start debugging.

Setting a Status Code in Express

```
res.status(201).json({ message: 'Book created successfully' });
```

If you never call .status(), Express defaults to 200.

Success and Redirection Codes

Code	Name	When to Use It
200	OK	Default success code with a body — successful GET, PUT, PATCH
201	Created	A new resource was created — the standard response to a POST that creates something, often paired with a Location header
204	No Content	Success, but nothing to send back — a typical DELETE response
301 / 302	Moved / Found	301 = permanently moved (update links); 302 = temporarily moved — the new URL is in the Location header

Client and Server Error Codes

Code	Name	When to Use It
400	Bad Request	Malformed request — broken JSON or a required field missing
401	Unauthorized	No valid credentials provided — "you need to log in"
403	Forbidden	Authenticated, but not allowed to do this
404	Not Found	No resource exists at this URL
409	Conflict	Conflicts with the resource's current state — e.g. email already taken
422	Unprocessable Entity	Well-formed request, but the data fails validation rules
500	Internal Server Error	Something broke on the server — an unhandled exception or bug

401 vs. 403 — A Very Common Mix-Up

WARNING

401 Unauthorized really means "I don't know who you are" — not logged in, missing or invalid credentials. 403 Forbidden means "I know who you are, but you're not allowed to do this" — logged in, but insufficient permissions, like a regular user hitting an admin-only route. Many students use these interchangeably; exams and real APIs do not.

400 vs. 422

WARNING

400 typically means the request itself is malformed at a structural level — broken JSON, wrong content type. 422 means the request was structurally fine and understood, but the data inside it fails validation — e.g. age: -5, or a missing required field in otherwise-valid JSON. Not every API distinguishes these strictly, but you should understand the difference conceptually.

Validating a POST and Responding 201 / 400

```
app.post('/api/books', (req, res) => {
  const { title, author } = req.body;

  if (!title || !author) {
    return res.status(400).json({
      error: 'title and author are required'
    });
  }

  const newBook = { id: 101, title, author };

  res.status(201)
     .location(`/api/books/${newBook.id}`)
     .json(newBook);
});
```

location() sets the Location header — where the new resource now lives.

204 No Content and 404 Not Found

```
app.delete('/api/books/:id', (req, res) => {
  // ... delete the book here ...
  res.status(204).send(); // nothing to return
});

app.get('/api/books/:id', (req, res) => {
  const book = null; // imagine none found
  if (!book) {
    return res.status(404).json({
      error: 'Book not found'
    });
  }
  res.status(200).json(book);
});
```

204 has no body; 404 means nothing exists at that URL.

Key Request Headers

Header	Purpose
Accept	Which content type(s) the client can handle in the response
Content-Type	The format of the request body — what express.json() checks before parsing req.body
Authorization	Credentials proving who the client is, e.g. Bearer <token>
User-Agent	Identifies the client software — browser, OS, or a tool like curl

Reading Headers in Express

```
app.get('/api/books', (req, res) => {
  console.log(req.headers['user-agent']);
  console.log(req.get('Accept'));
  res.send('ok');
});
```

req.headers uses lowercase keys; req.get() is case-insensitive.

Key Response Headers

Header	Purpose
Content-Type	Format of the response body — set automatically by res.json() / res.send()
Cache-Control	How long the response may be reused before it must be re-fetched
Set-Cookie	Tells the browser to store a cookie (covered next lecture)
Location	Where a redirect points, or where a newly created resource now lives

Setting Response Headers

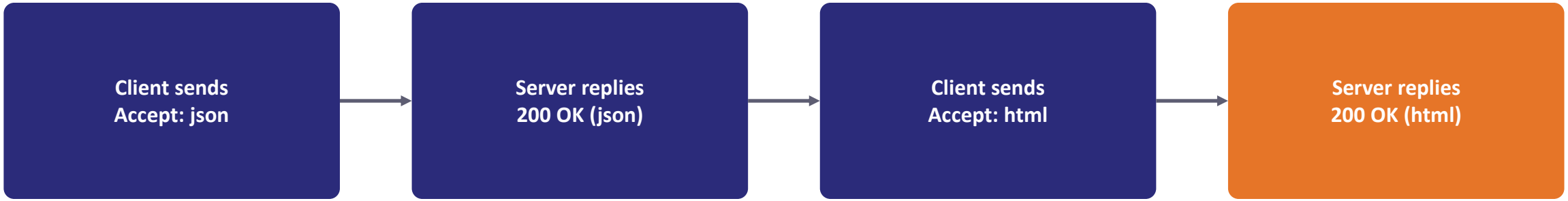
```
app.get('/api/report', (req, res) => {
  res.set('Cache-Control', 'no-store');
  res.json({ generatedAt: new Date().toISOString() });
});

app.get('/old-path', (req, res) => {
  res.redirect(301, '/new-path');
});
```

res.redirect(301, ...) sets both the 301 status and the Location header automatically.

Client and Server Agree on a Format

1. The client states its preferred format(s) via the Accept header
2. The server decides how to respond based on that — and on what it's actually able to produce
3. A browser navigating to a URL sends Accept: text/html, ... — a fetch() call from a JS app might send Accept: application/json instead

A Content-Negotiated Exchange



Same server, same route family — the Accept header the client sends decides which format the response comes back in.

res.format() in Express

```
app.get('/books', (req, res) => {
  res.format({
    'application/json': () => {
      res.json({ books: ['Clean Code', 'Pragmatic Prog.'] });
    },
    'text/html': () => {
      res.send('<h1>Books</h1>');
    },
    default: () => {
      res.status(406).send('Not Acceptable');
    }
  });
});
```

res.format() runs the callback matching the client's Accept header; unmatched types fall to default.

You're Already Doing This

TIP

Even without res.format(), you perform a simpler form of content negotiation every time you choose res.json() vs. res.send() vs. res.render() — you are deciding, on the server side, what format to return. Full content negotiation just makes that decision dynamic, based on what the client actually asked for.

KEY TAKEAWAYS

Lecture 19 in Six Points

1. Status codes fall into five families by their first digit: 1xx informational, 2xx success, 3xx redirection, 4xx client error, 5xx server error.
2. Know the exact use of 200, 201, 204, 301/302, 400, 401, 403, 404, 409, 422, and 500 — especially 401 (not authenticated) vs. 403 (not authorized), and 400 (malformed) vs. 422 (failed validation).
3. Key request headers: Accept (what the client wants back), Content-Type (format of the request body), Authorization (credentials), User-Agent (identifies the client).
4. Key response headers: Content-Type (response format), Cache-Control (caching rules), Set-Cookie (store a cookie), Location (redirect / new resource URL).
5. Content negotiation lets a client and server agree on a response format using the Accept header; Express supports this with res.format().
6. Set status codes explicitly with res.status(code), and always choose the code that most accurately describes what happened.