LECTURE 20

Cookies and Sessions

How a stateless protocol remembers you — logged in, mid-checkout, and everything in between.

CSC336 Web Technologies

OVERVIEW

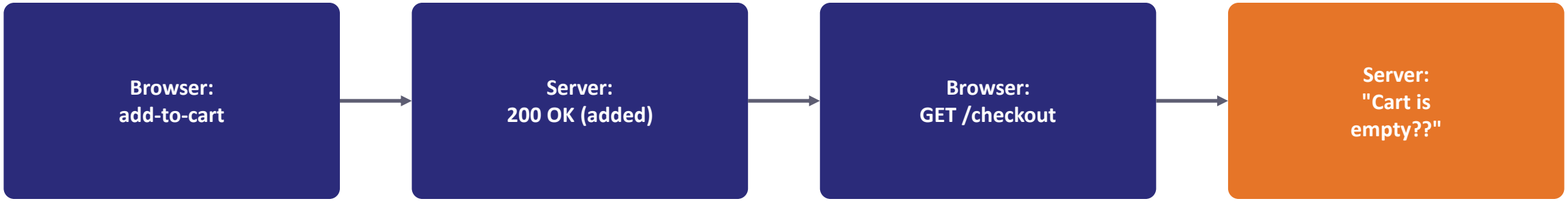
In This Lecture

1. Understand why HTTP is stateless, and why applications need a way to manage state
2. Learn how cookies are created and understand their key attributes
3. Learn how sessions work: session identifiers and server-side session stores
4. Compare session-based state management with pure cookie-based state
5. Walk through a full login/logout flow, including session expiry
6. Understand basic security considerations for cookies and sessions

The Statelessness of HTTP

1. Stateless means each request is handled with no memory of any previous request
2. Intentional — it's a big reason the web scales so well; a stateless server doesn't track every visitor it's ever seen
3. But real applications clearly DO remember you: logged in, cart contents, preferences

Without Help, the Server Forgets



Without extra help, the server has already forgotten the add-to-cart request by the time /checkout arrives — each request stands completely alone.

Cookies and Sessions

1. State management: a mechanism that lets a stateless protocol FEEL stateful by carrying identifying information along with each request

2. Lets the server connect separate requests together as belonging to the same visitor

3. The two main tools are cookies and sessions — often used together

What Is a Cookie?

1. A small key-value pair, generally capped around 4KB, that a server asks the browser to store
2. The browser automatically resends it with every future request to that same server
3. The basic building block that makes state management possible on the web

Creating a Cookie

```
app.get('/set-theme', (req, res) => {
  res.cookie('theme', 'dark'); // sets Set-Cookie
  res.send('Theme preference saved!');
});

app.get('/get-theme', (req, res) => {
  console.log(req.cookies); // needs cookie-parser
  res.send('Check the console for your theme.');
});
```

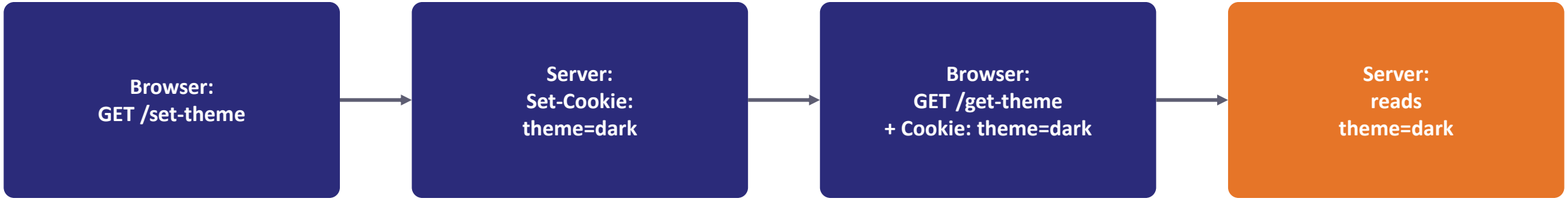
res.cookie() works out of the box; reading req.cookies needs extra middleware (next slide).

req.cookies Needs cookie-parser

NOTE

Express does not parse incoming cookies into req.cookies by default — you need the small cookie-parser package (npm install cookie-parser, then app.use(require('cookie-parser')())) to read cookies sent by the browser. Setting cookies with res.cookie(), however, works out of the box.

Setting and Reading a Cookie



The browser stores the cookie once, then attaches it automatically to every later request to that server — no extra client-side code required.

Cookie Attributes That Matter

Attribute	Meaning
Expires / Max-Age	When the cookie is deleted — without it, a session cookie (gone when the browser closes); with it, persistent
Path	Restricts the cookie to requests under a specific path, e.g. Path=/admin
HttpOnly	Blocks client-side JavaScript (document.cookie) from reading the cookie — defends against XSS
Secure	The cookie is only ever sent over HTTPS — always use this in production
SameSite	Controls cross-site sending: Strict, Lax (common default), or None — a major CSRF defense

Setting Attributes

```
app.get('/login', (req, res) => {
  res.cookie('sessionId', 'abc123', {
    maxAge: 24 * 60 * 60 * 1000, // 24 hours
    httpOnly: true,
    secure: true,  // HTTPS only
    sameSite: 'lax'
  });
  res.send('Logged in!');
});
```

maxAge is in milliseconds; without it (or Expires) the cookie disappears when the browser closes.

Always Set HttpOnly on Sensitive Cookies

WARNING

A cookie without HttpOnly can be read — and stolen — by any JavaScript running on the page, including malicious injected scripts. Any cookie that identifies a logged-in user, like a session ID, should always be set with HttpOnly: true.

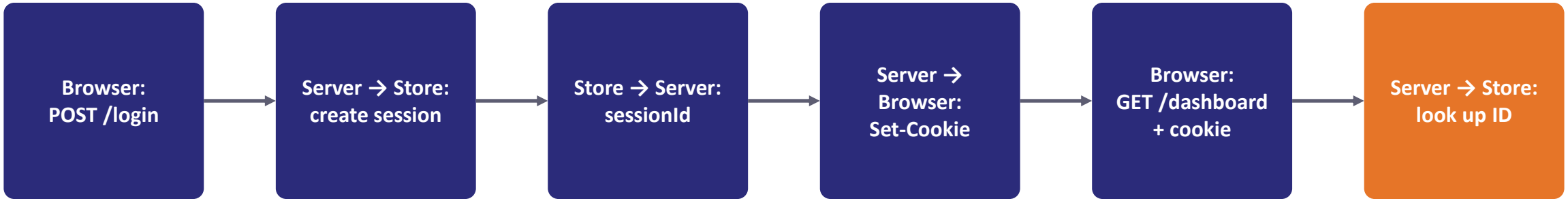
Why Not Just Store Data in the Cookie?

1. Cookies are capped around 4KB — too small for real application data
2. They're visible to (and can be tampered with by) the user unless specially protected
3. You generally don't want sensitive data like "this user is an admin" sitting on the client

How a Session Works

1. The session identifier (session ID) is generated by the server and sent to the browser as a cookie
2. The actual session data lives in a session store on the server — memory for testing, a database or Redis in production
3. On every request, the browser resends the session ID cookie; the server looks it up in its store

Login Creates a Session



The cookie only ever carries an opaque ID — the real data (who's logged in) stays server-side in the session store.

express-session Setup

```
const express = require('express');
const session = require('express-session');
const app = express();
app.use(express.json());

app.use(session({
  secret: 'a-long-random-secret-string',
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true,
    secure: false, // true in production
    maxAge: 30 * 60 * 1000 // 30 min
  }
}));
```

Login and Dashboard Routes

```
app.post('/login', (req, res) => {
  // verify credentials against a DB first
  req.session.userId = 7;
  req.session.username = req.body.username;
  res.send('Logged in!');
});

app.get('/dashboard', (req, res) => {
  if (!req.session.userId) {
    return res.status(401).send('Please log in first.');
  }
  res.send(`Welcome back, ${req.session.username}!`);
});
```

Logout Route

```
app.post('/logout', (req, res) => {
  req.session.destroy(() => {
    res.send('Logged out.');
  });
});

app.listen(3000);
```

req.session.destroy() removes the session from the store — nothing is left for the old cookie to reference.

Where Does the Data Actually Live?

	Cookie-based state	Session-based state
Where the data lives	In the browser (inside the cookie)	On the server (in the session store)
Size limit	Small (~4KB per cookie)	Effectively unlimited
Can the user tamper?	Yes, unless cryptographically signed	No — the user only holds an opaque ID
Server needs storage?	No	Yes — memory, database, or Redis
Good for	Small, low-sensitivity values (theme, language)	Anything sensitive or substantial (login, cart)

Signed Cookies: A Middle Ground

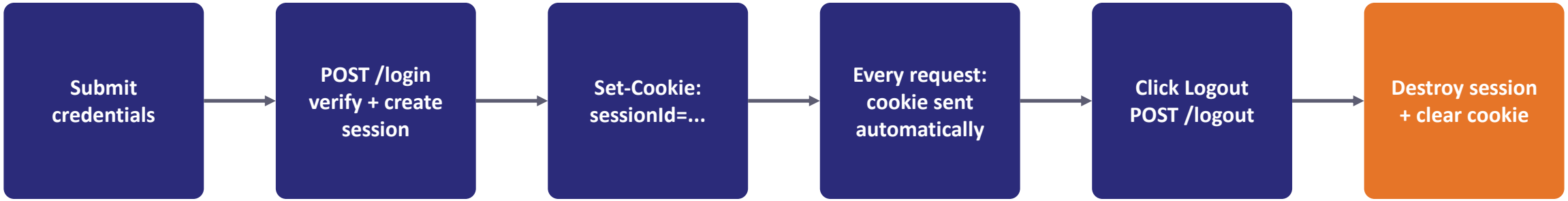
TIP

Signed cookies keep the actual data in the cookie, but cryptographically sign it so the server can detect tampering. This adds tamper-detection but not secrecy — the data is still visible to the user, just not undetectably editable. Still not the right choice for genuinely secret data.

Putting It Together

1. Login: the user submits credentials to a login route
2. The server verifies the credentials against a database
3. If valid, it creates a new session, stores data like the user's ID, and sends the session ID back as a cookie
4. Authenticated requests: the browser attaches the session cookie automatically; the server looks up the session on each one
5. Logout: destroys the session on the server, and typically clears the cookie on the client too
6. Expiry: maxAge stops the browser from resending an old cookie; a good session store also expires the server-side data independently

The Full Login → Logout Sequence



Even if someone later obtained the old session cookie, it would no longer correspond to a valid session once destroyed.

Basic Security Considerations

- Always set HttpOnly on session cookies — closes off a major theft vector (XSS)
- Always use Secure (and HTTPS) in production — the session ID is never sent in plain text
- Set a reasonable SameSite value and a reasonable maxAge — sessions that never expire are a standing risk
- Never store highly sensitive raw data (like plaintext passwords) inside a session, even server-side

SECURITY

Always Destroy the Session on Logout

WARNING

Destroy the session on the server side, not just by clearing the cookie in the browser — otherwise, an attacker who somehow captured the old cookie value could still use it after the user thought they had logged out.

KEY TAKEAWAYS

Lecture 20 in Six Points

1. HTTP is stateless: each request is handled independently, with no built-in memory of previous requests from the same client.
2. A cookie is a small piece of data the server asks the browser to store and resend automatically, set via the Set-Cookie header.
3. Key cookie attributes: Expires/Max-Age (lifetime), Path (scope), HttpOnly (blocks client-side JS), Secure (HTTPS only), and SameSite (cross-site sending).
4. A session keeps the actual data on the server (in a session store) and uses a session identifier, held in a cookie, as the lookup key.
5. A login/logout flow creates a session at login, checks it on each authenticated request, and destroys it at logout — sessions should always expire.
6. Always combine HttpOnly, Secure, and a sensible SameSite setting on session cookies, and always destroy sessions server-side on logout.