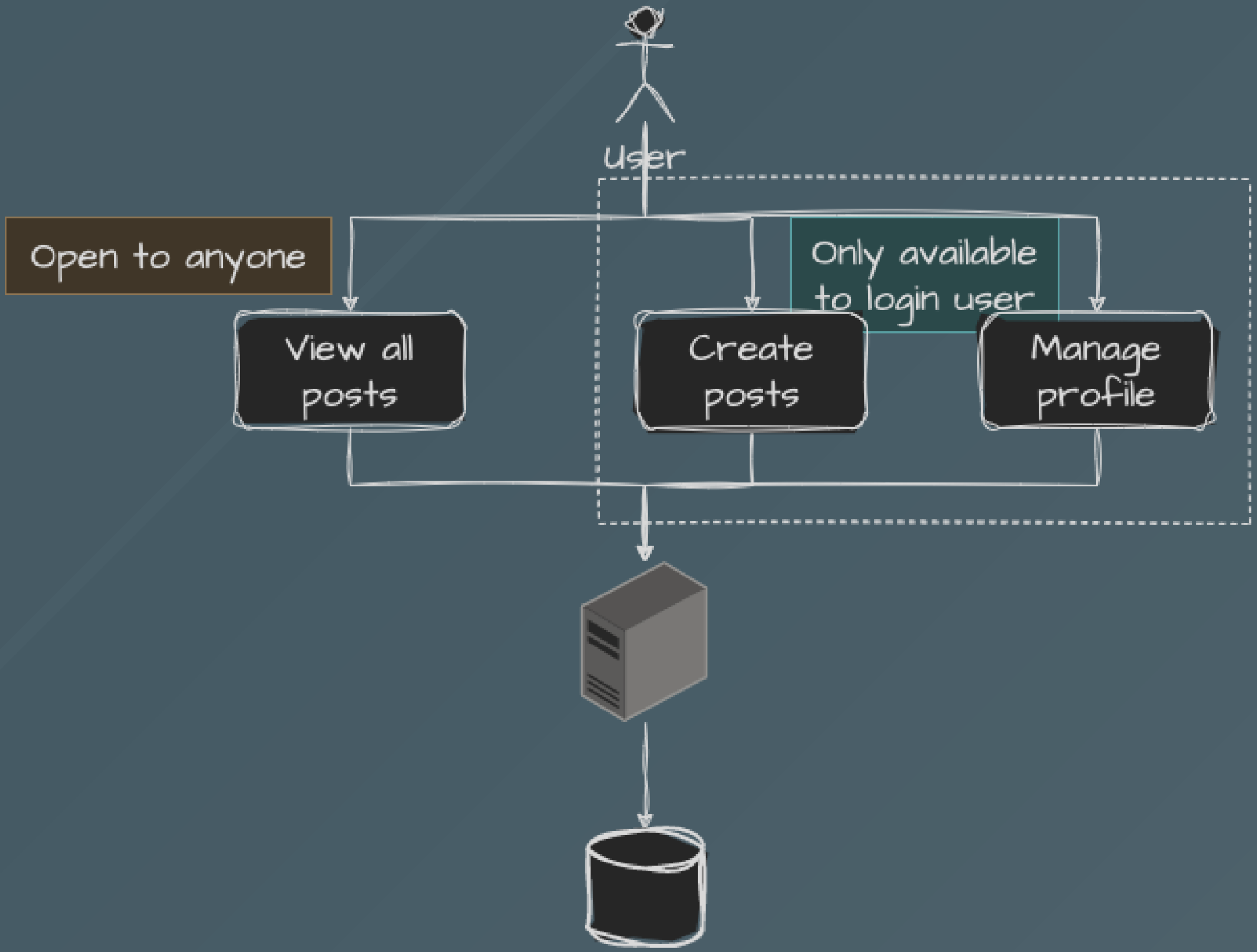
Authentication & Authorization

- **Authentication (AuthN):** *Who are you?* (verify identity)
- **Authorization (AuthZ):** *What are you allowed to do?* (verify access)

Authentication

- How do we verify the identity of a user?
- How do we keep the identity of a user across requests?

> “ We need a way to attach an identity to each request. ”



Two common strategies

A) Cookie + Session (stateful)

- Browser stores a **session id** in a cookie
- Server stores **session data** (user id, roles, etc.)

B) Token (JWT) (stateless-ish)

- Client stores a **JWT**
- Server verifies the **signature** and reads claims

Part 1 - Cookies + Sessions

Cookies

- Small key/value data stored in the **browser**
- Sent by the browser to the server on matching requests
- Created via HTTP response header: `Set-Cookie: ...`

Common attributes

- `Domain`, `Path`, `Expires/Max-Age`

Cookie security flags

- `HttpOnly`
  ✅ JavaScript cannot read it (mitigates token theft via XSS)
- `Secure`
  ✅ only sent over HTTPS
- `SameSite=Strict | Lax | None`
  ✅ reduces CSRF
  ⚠️ `SameSite=None` requires `Secure`

> **Rule of thumb:** for auth cookies: `HttpOnly; Secure; SameSite=Lax` (or `Strict` if possible)

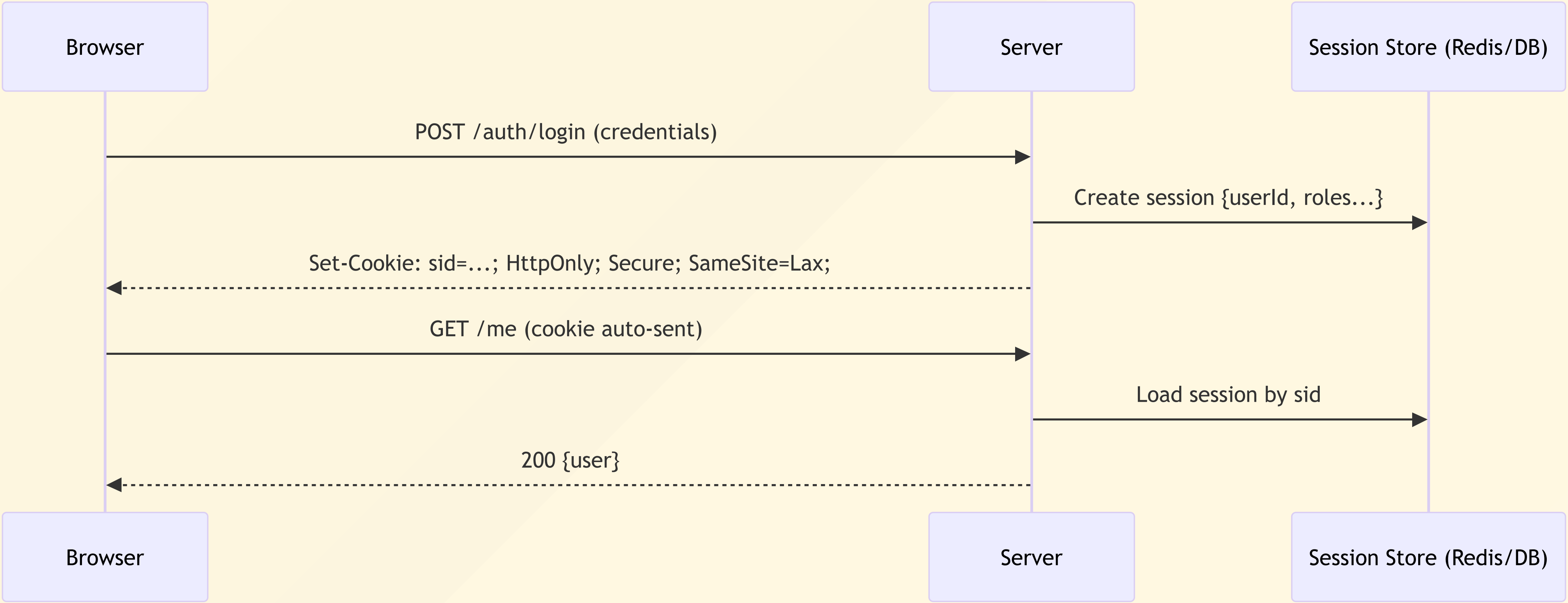
Sessions

- **Session id** in cookie
- **Session data** on server (memory/Redis/DB)
- Server uses session id → loads user context

Why sessions are “stateful”

Because the server must keep session data somewhere.

Session login flow


Browser
Server
Session Store (Redis/DB)
POST /auth/login (credentials)
Create session {userId, roles...}
Set-Cookie: sid=...; HttpOnly; Secure; SameSite=Lax;
GET /me (cookie auto-sent)
Load session by sid
200 {user}
Browser
Server
Session Store (Redis/DB)

Session lifecycle

- **Regenerate session id after login**
  (prevents *session fixation*)
- **Rotate/refresh** session expiration on activity (optional)
- **Logout = destroy session on server**
  and clear cookie (`Max-Age=0`)

Scaling sessions

- ❌ In-memory session store breaks in multi-instance deployments
- ✅ Use a shared store (Redis is common)

Load balancer options

- Sticky sessions (simpler, less flexible)
- Shared session store (recommended)

Potential vulnerabilities

- **XSS:** attacker runs JS in your page
  → can steal tokens from `localStorage` / JS-readable cookies
- **CSRF:** attacker tricks browser into sending cookies to your site
  → possible when auth uses cookies
- **Session fixation:** attacker forces a known session id
  → fix by regenerating session after login

CSRF (why cookie auth is vulnerable)

If the browser auto-sends cookies, a malicious site can cause:

- A hidden form submit
- An image request
- A fetch request (CORS may block reading response, but request can still be sent)

Result: request reaches your server *with* valid cookies.

Pros & Cons: Cookies + Sessions

✅ Pros

- Easy logout/invalidation (destroy server session)
- Sensitive data stays on server
- Simple mental model for browser apps

❌ Cons

- Requires session store (Redis/DB) to scale
- CSRF concerns (cookie auto-sent)
- Cookies sent on every request (overhead)
- Limited to native mobile apps (no browser cookies)

Code Example

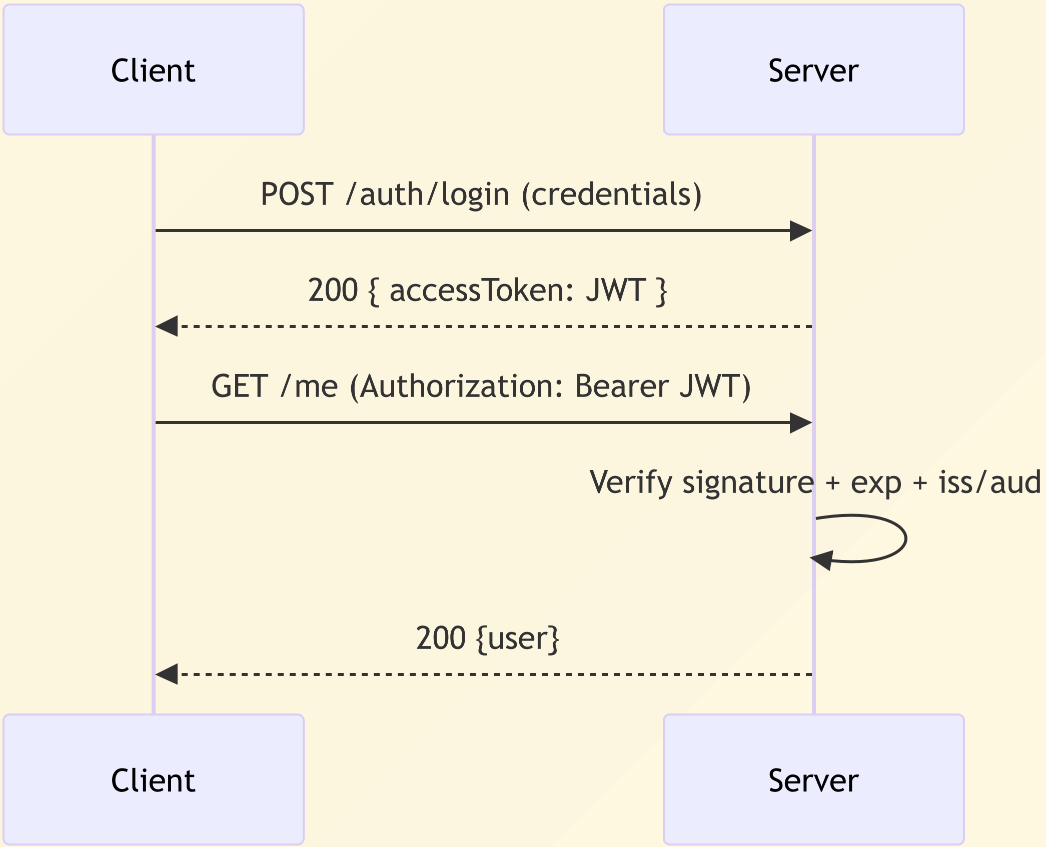
Part 2 — JSON Web Tokens (JWT)

JWT (what it is)

JWT is an open standard (RFC 7519) defining a compact and self-contained way to securely transmit information between parties as JSON.

Key point: JWT is typically **signed** (integrity), not encrypted (confidentiality).

JWT login flow


Client
Server
POST /auth/login (credentials)
200 { accessToken: JWT }
GET /me (Authorization: Bearer JWT)
Verify signature + exp + iss/aud
200 {user}
Client
Server

JWT structure

- **Header**: algorithm, token type
- **Payload**: claims (user id, expiration, roles, etc.)
- **Signature**: Header + Payload + Secret (or private key)

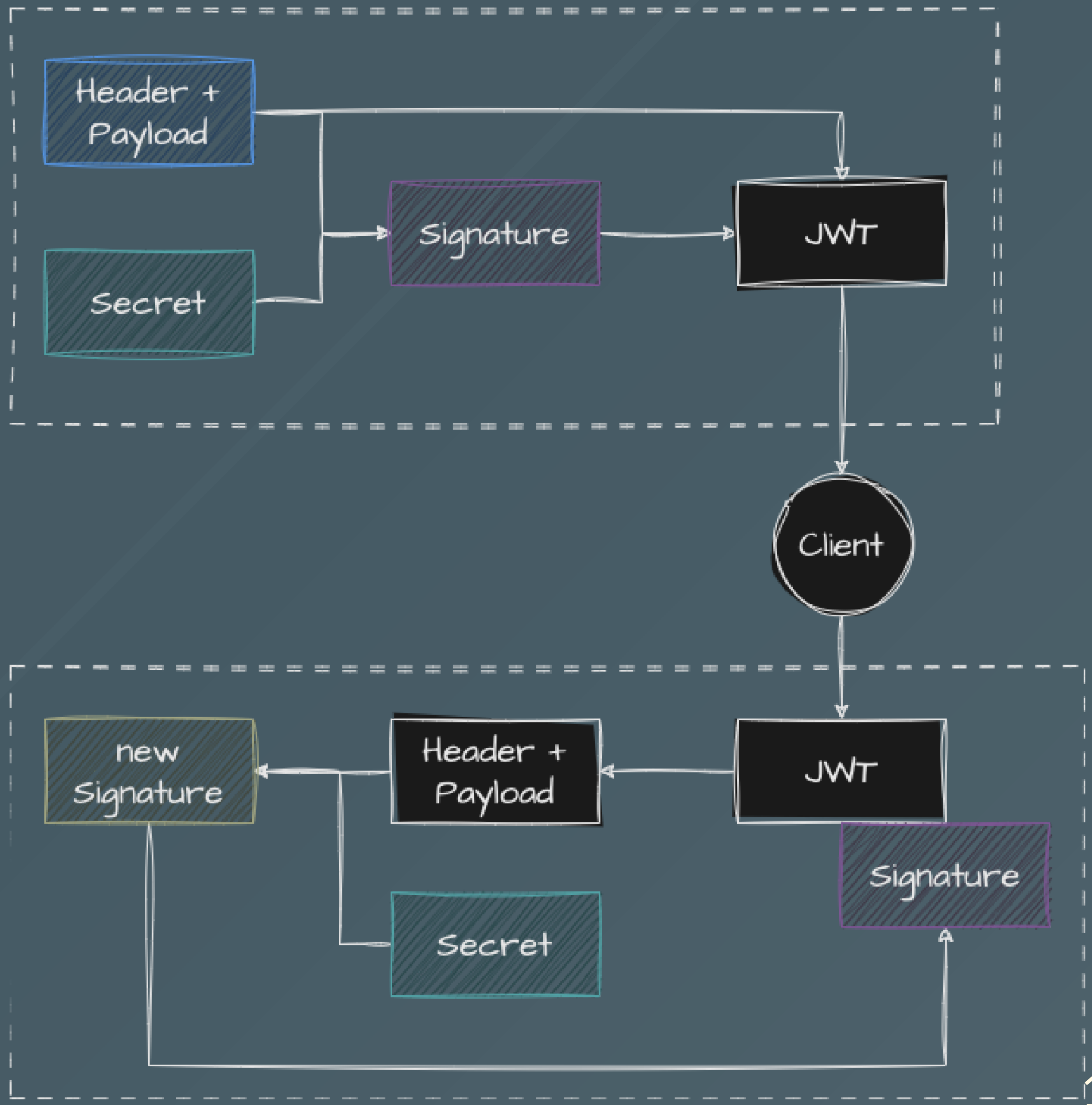
Common claims

- `sub` (subject / user id), `exp`, `iat`, `iss`, `aud`
- `roles` / `scope` (for authorization)

> ⚠️ Put **no secrets** in the payload. Anyone can decode it.

Signing and Verifying

- The secret is only known to the server.



Where to store/send JWT?

Option 1: `Authorization: Bearer <token>`

- ✅ avoids CSRF by default (browser won’t auto-attach)
- ✅ common for APIs/mobile apps
- ⚠️ vulnerable if token is stored in `localStorage` and you have XSS

Option 2: HttpOnly cookie

- ✅ mitigates token theft via XSS
- ⚠️ brings CSRF back (need SameSite + CSRF token)

Session vs JWT (decision table)

Topic	Session + Cookie	JWT (Bearer)
State	Server-side	Client-side (token)
Logout	Easy (destroy session)	Harder (wait expiry / revoke strategy)
Scaling	Needs shared store (Redis/DB)	Easier (verify token)
CSRF	Higher risk	Low by default
XSS impact	Cookie HttpOnly helps	Depends on storage (avoid localStorage)
Best for	Traditional web apps	APIs, mobile, microservices

Disadvantages of JWT

- **Compromised secret/key:** all tokens become forgeable
- **Data visibility:** payload is readable (not encrypted)
- **Revocation is harder:** needs strategy (short exp / refresh / denylist)
- **Overhead:** larger than a session id cookie

Avoid: putting tokens in URLs (URLs leak via logs, history, referrers)

JWT in Node.js

- jsonwebtoken
- passport-jwt

Part 3 — Authorization (AuthZ)

Authorization patterns

1. **Route-level:** must be logged in
2. **RBAC:** role-based access control (admin/user)
3. **Resource ownership:** user can only access their own data

Example rule:

- user can read an order if `user.role === 'admin' || order.userId === user.id`

Summary (rules of thumb)

- **Browser web app:** session-cookie is simplest; use secure cookie flags + CSRF protection
- **APIs/mobile/microservices:** JWT bearer is common; short expiry + refresh strategy
- Always implement **AuthZ** explicitly (roles + resource ownership)
- Secure defaults:
  - never put tokens in URLs
  - avoid storing tokens in `localStorage`
  - validate `exp/iss/aud` and lock down algorithms