

Backend · concept board

Auth & Web Security

Sessions, JWT, OAuth, hashing, OWASP Top 10, CORS, CSP

1. Authentication vs authorization

2. Storing passwords

3. Sessions vs JWT

4. OAuth 2.0 and OpenID Connect

5. Common attacks (OWASP)

6. Browser security: CORS, cookies, headers

Draw on any slide, then Present (arrow keys) or export the board as a PDF.

1. Authentication vs authorization

Definition

Authentication = who are you? (login). Authorization = what may you do? (roles, permissions, ownership).

Fail authentication → 401; fail authorization → 403.

- RBAC: permissions by role (admin, doctor, patient).
- Always check ownership too: "can this user see *this* order?" — missing checks cause IDOR bugs.
- Authorize on the server for every request; hiding a button is not security.

2. Storing passwords

Hash with bcrypt / argon2 — never encrypt or plain-hash

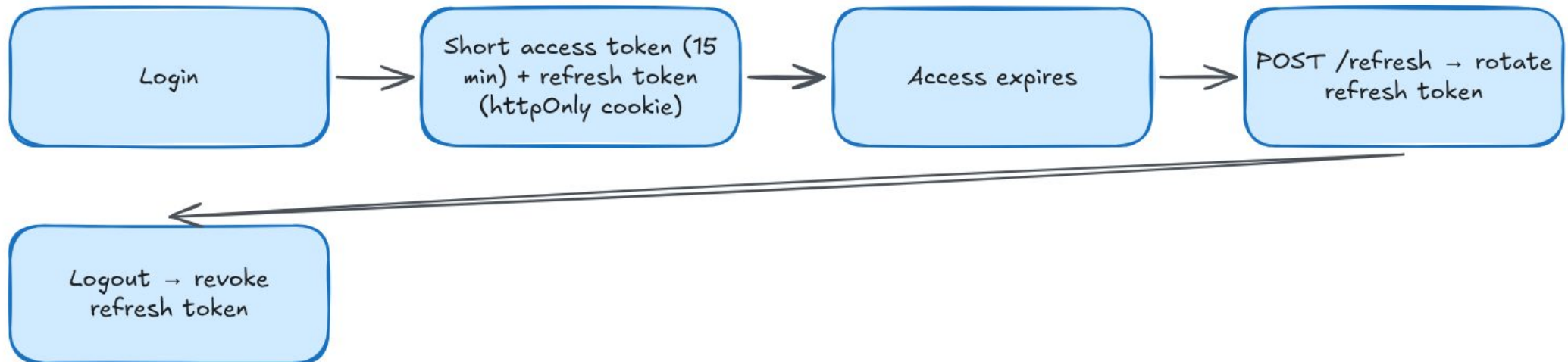
```
import bcrypt from 'bcrypt'
```

```
const hash = await bcrypt.hash(password, 12) // salt is generated and stored inside the hash  
const ok = await bcrypt.compare(attempt, hash) // constant-time compare
```

- Use slow, salted algorithms: argon2id or bcrypt. *Never MD5/SHA-256 alone.*
- Rate-limit login and OTP endpoints; lock out or add delays after failures.
- Same error for "no such user" and "wrong password" (no user enumeration).
- Offer MFA (TOTP / passkeys) for sensitive apps.

3. Sessions vs JWT

	Server session	JWT
State	stored on the server (Redis/DB); cookie holds only an id	claims inside a signed token
Logout / revoke	easy: delete the session	hard: valid until it expires
Scaling	needs a shared store	any server can verify the signature
Best for	classic web apps	APIs, mobile, microservices



Access + refresh token pattern

Common mistake

JWTs are signed, not encrypted — never put secrets in the payload. Always verify the signature and exp, and pin the algorithm (reject alg: none).

4. OAuth 2.0 and OpenID Connect

OAuth 2.0 lets an app act on a user's behalf without their password (authorization). OpenID Connect adds an ID token on top, for login with Google/GitHub (authentication).

1. App redirects to the provider with `client_id`, `redirect_uri`, `scope`, `state` and a PKCE challenge.
2. User logs in and consents.
3. Provider redirects back with a one-time code.
4. Server exchanges the code (+ PKCE verifier) for tokens.
5. App reads the ID token / user info and creates its own session.

Tip

Use the Authorization Code flow with PKCE; state protects against CSRF on the callback. Libraries: Auth.js (NextAuth), Passport, Clerk, Cognito.

5. Common attacks (OWASP)

Attack	What happens	Defence
XSS	attacker script runs in your page	escape output (React does), avoid dangerouslySetInnerHTML or sanitise (DOMPurify), CSP, httpOnly cookies
CSRF	another site makes the browser send a request with your cookies	SameSite=Lax/Strict cookies, CSRF tokens, check Origin
SQL / NoSQL injection	input changes the query	parameterised queries / ORM, validate types, reject \$ operators in input
Broken access control / IDOR	/orders/124 shows someone else's order	check ownership on every request
SSRF	server fetches an attacker-chosen internal URL	allow-list hosts, block internal IP ranges
Clickjacking	your page is framed and clicks are hijacked	X-Frame-Options / CSP frame-ancestors
Secrets leaked	keys in git or frontend bundles	env vars / secret manager, rotate keys

Injection: never build queries with strings

```
// × vulnerable: id = "1 OR 1=1"
db.query('SELECT * FROM users WHERE id = ${req.params.id}')

// ✓ parameterised
db.query('SELECT * FROM users WHERE id = ?', [req.params.id])
```

6. Browser security: CORS, cookies, headers

Definition

Same-origin policy: a page can only read responses from the same scheme + host + port. CORS is the server opting in to let other origins read its responses (Access-Control-Allow-Origin). CORS does not protect your API from non-browser clients.

- Non-simple requests (JSON body, custom headers, PUT/DELETE) trigger a preflight OPTIONS request.
- Credentialed requests can't use *; list exact origins.
- Cookie flags: HttpOnly (no JS access), Secure (HTTPS only), SameSite=Lax (CSRF defence).
- Headers: HSTS, CSP, X-Content-Type-Options: nosniff, Referrer-Policy — helmet sets sensible defaults.

Revise

Hash passwords with argon2/bcrypt, short access + rotating refresh tokens in httpOnly cookies, authorize every request (ownership!), parameterised queries, escape output + CSP, SameSite cookies, CORS is an opt-in not a firewall.

Rapid-fire Q&A (1/2)

Authentication vs authorization?

Authentication verifies identity (login); authorization decides what that identity may do (roles, ownership).

How should passwords be stored?

Hashed with a slow, salted algorithm such as argon2id or bcrypt — never plain, encrypted or fast-hashed.

Session vs JWT?

Sessions keep state on the server and are easy to revoke; JWTs are signed, stateless tokens that scale easily but are hard to revoke, so keep them short-lived with refresh tokens.

What is XSS and how do you prevent it?

Injected script running in your page. Escape output, avoid raw HTML or sanitise it, use a CSP, and keep tokens in httpOnly cookies.

What is CSRF?

Another site causes the browser to send an authenticated request with your cookies. Defend with SameSite cookies, CSRF tokens and Origin checks.

How do you prevent SQL injection?

Parameterised queries or an ORM — never concatenate user input into SQL.

Rapid-fire Q&A (2/2)

What does CORS do?

It lets a server allow specific other origins to read its responses in the browser. It is not an access-control mechanism for the API itself.

What is a preflight request?

An `OPTIONS` request the browser sends before a non-simple cross-origin request to check that the server allows it.

OAuth vs OpenID Connect?

OAuth 2.0 is for delegated authorization (access tokens); OpenID Connect adds an ID token for authentication (login with Google).

What is IDOR?

Insecure Direct Object Reference: changing an id in the URL shows another user's data because ownership isn't checked.

What do `HttpOnly`, `Secure` and `SameSite` do?

`HttpOnly` hides the cookie from JS, `Secure` sends it only over HTTPS, `SameSite` limits sending it on cross-site requests.