Web Authentication Course

Flavio Copes

Updated August 6, 2026

Contents

Secure a web API with password authentication, server-side sessions, authorization, CSRF defenses, OAuth and OIDC, passkeys, recovery, MFA, and operational controls.

What you'll learn: Design, implement, test, and operate a vendor-neutral authentication system that protects each user's Books API resources and supports modern login and recovery paths.

Authentication foundations

Separate identity and authorization, threat-model the system, and create secure sessions.

Separate identity, authentication, and authorization

Distinguish an account identifier, proof of control, session state, and permission checks before building login endpoints.

Authentication verifies a claim about who is calling. Authorization decides what that authenticated identity may do. An account record and a session are related but different things.

Keep four questions separate:

1. **Identity:** which account is this?
2. **Authentication:** what proof did the caller present?
3. **Session:** how does the server remember that proof across requests?
4. **Authorization:** may this account perform this action on this resource?

Our project secures the Books API. A user signs in, receives a session, and can manage books they own. The server must derive the owner from the trusted session, not from a user ID submitted in the request body.

This request value is untrusted:

```
{ "title": "The Hobbit", "ownerId": "user-42" }
```

The handler should ignore `ownerId`. It gets the user ID from the validated session and writes that value itself.

A successful login answers only the authentication question. It does not prove the user is an administrator or owns book 17. Make the authorization decision again at every protected operation.

This separation also improves failures. A missing session is an authentication failure. A valid session requesting someone else's book is an authorization failure. A database outage is neither.

Draw signup, login, authenticated request, authorization decision, and logout as separate steps. Label the trusted input at every boundary.

Threat-model the login system

Identify assets, attackers, entry points, trust boundaries, and abuse cases before selecting authentication mechanisms.

Security work starts by asking what must be protected and how it could fail, not by adding a fashionable login library.

Protect credentials, sessions, private books, recovery channels, and administrative actions. Consider credential stuffing, enumeration, session theft, CSRF, XSS, insecure direct object references, reset-link theft, and compromised dependencies.

Draw the data flow before listing controls:

```
browser → application → session store
                      user database
                      email provider
                      identity provider
```

Every arrow crosses a trust boundary. Ask what data crosses it, who controls the input, and what evidence the receiver requires.

Write concrete abuse cases. “An attacker changes `/books/17` to `/books/18` and reads another user's book” is testable. “Authorization might fail” is too vague.

Rank risks by likely impact and exposure. Credential stuffing on a public login deserves different controls from a malicious database administrator. Include recovery too: how will you revoke stolen sessions, rotate a provider secret, or notify affected users?

Do not assume one control solves a whole threat. `HttpOnly` reduces direct cookie theft through JavaScript, but it does not prevent actions made by injected code. Rate limiting slows guesses, but it does not make a reused password safe.

Create a table with asset, attacker goal, entry point, prevention, detection, and recovery. Include one unauthenticated attack and one signed-in user attacking another account.

Choose server-side sessions

Use a random opaque cookie value that maps to server-side session state and understand the tradeoff against self-contained tokens.

For a browser-first application, an opaque server-side session is a strong default. The browser stores only a meaningless identifier; the server owns identity and session state.

The cookie might contain `7xK...pQ`, but that value carries no user data. On every request, the server uses it to find a record like this:

```
session hash → user ID, created time, last-used time, expiry, revoked time
```

Generate identifiers with a cryptographically secure random source or a trusted session library. Store only a hash of the identifier when feasible, plus user ID, creation, expiry, and revocation data.

Hashing limits what a database-only leak reveals. The raw value stays in the cookie. The server hashes the presented value and looks up the matching record, much like a password-reset token.

Server-side sessions cost a database or cache lookup and shared state across application instances. In return, logout, account suspension, password reset, and “sign out every device” can take effect immediately.

Self-contained tokens fit some architectures, but they do not remove security work. You still need key rotation, expiry, audience checks, safe browser storage, and often a revocation strategy. Choose them when the architecture needs those specific properties, not to avoid one lookup.

Design the users and sessions tables without putting passwords, roles, email addresses, or personal data inside the cookie value. Show how deleting one session record invalidates its cookie.

Set a secure session cookie

Use HTTPS and restrictive cookie attributes so scripts, insecure transport, and unnecessary cross-site requests cannot expose the session.

A session identifier is temporarily equivalent to the authentication proof that created it. Protect it as a credential.

Use an `HttpOnly`, `Secure`, appropriately `SameSite` cookie, no broad Domain, and a controlled lifetime. A `__Host-` name adds browser-enforced restrictions when its requirements are met. It

requires `Path=/`, so choose the host-bound prefix or narrower path scoping deliberately rather than claiming both. Serve the entire authenticated application over HTTPS.

```
Set-Cookie: __Host-session=opaque-random-value; Path=/; HttpOnly; Secure; SameSite=Lax; Max-A
```

Each attribute solves a different problem:

- `HttpOnly` prevents ordinary browser JavaScript from reading the value.
- `Secure` sends it only over HTTPS.
- `SameSite` limits some cross-site requests.
- `Path=/` makes a `__Host-` cookie available to the whole origin.
- omitting `Domain` keeps subdomains from setting or receiving it.

These controls reduce exposure. They do not replace server validation, CSRF protection, or XSS prevention. Injected code may be unable to read an `HttpOnly` cookie but can still submit authenticated actions from the page.

Expire the browser cookie and revoke the server record on logout. Clearing only the cookie leaves a copied token valid. Revoking only the record is safe, but the browser keeps sending a useless credential until it expires.

Set the cookie through an HTTPS-capable development environment or use a documented local exception without weakening production. Inspect its flags and scope in DevTools, revoke the server record, and confirm the next request fails.

Check your understanding: authentication foundations

Check identity, authentication, authorization, threat modeling, sessions, cookies, HTTPS, and trusted implementation boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Password authentication

Design signup, hash passwords safely, implement login, and resist automated attacks.

Design safe signup

Normalize account identifiers, validate input, allow password managers, and create accounts without leaking whether another user exists.

Signup creates security-critical records from untrusted input. Decide which identifier is canonical and which fields require later verification.

Normalize email according to a documented conservative policy, enforce database uniqueness, allow long passphrases and password-manager paste, and avoid arbitrary composition rules. Never silently truncate a password. Return a generic outcome where enumeration risk matters.

Let the database settle concurrent requests:

```
CREATE UNIQUE INDEX users_email_unique
ON users (normalized_email);
```

An application-level "does this email exist?" check can race. Two requests may both see no row and then insert. The unique constraint is the final invariant; the handler translates its error into the product's chosen response.

Keep the original email for display and delivery if needed, and store a normalized form for lookup. Avoid aggressive provider-specific rewriting, such as removing dots or tags, unless your product owns that policy and its consequences.

Create the account in an unverified state. Sending an email proves only that a message was attempted. The account becomes verified after a valid, scoped, unexpired token is consumed.

Password rules should work with password managers. Allow paste, spaces, Unicode, and at least 64 characters. As a current baseline, treat passwords shorter than 15 characters as weak when MFA is absent and shorter than 8 when MFA is required. Check known-compromised choices without sending the full password to another service.

Define the signup request schema, unique constraint, duplicate behavior, and verification state for the Books API. Race two requests for the same email and inspect the final rows.

Hash passwords with a trusted library

Store password verifiers with a current password-hashing algorithm and library instead of encryption or a fast general-purpose hash.

Passwords must not be stored in plaintext or reversibly encrypted. A database compromise should still make guessing expensive.

Use a maintained library implementing Argon2id with parameters chosen from current OWASP guidance for your environment. Let the library generate a unique salt and encode its algorithm and parameters with the result. Never invent a password-hashing construction.

```
// Library API names differ. Use the maintained library you selected.
const passwordHash = await passwordHasher.hash(password)
await users.create({ id, email, passwordHash })
```

The encoded value normally contains the algorithm, salt, and work parameters. The salt does not need to be secret. It prevents two equal passwords from producing equal stored verifiers and defeats one precomputed table for every account.

Tune the work factor on production-like hardware. A login must be affordable for a real user but expensive to repeat at attacker scale. Measure concurrent logins too; a setting that looks fine once can exhaust CPU or memory under load.

Parameters age. After a successful login, inspect the encoded verifier. If it uses an older algorithm or weaker settings, hash the supplied password again with the current policy and replace the stored value. This upgrades active accounts without storing plaintext.

A server-side pepper can add protection when it lives outside the database, but it creates a key-management and rotation problem. Do not add one unless you can operate it safely.

Benchmark the chosen parameters on production-like hardware. Record single-login latency, concurrent behavior, library version, current policy, and the rehash trigger.

Implement login with generic errors

Compare password hashes safely, avoid user enumeration, rotate session state, and preserve consistent observable behavior.

Login receives an identifier and secret, finds a verifier, and establishes a new authenticated session only after a successful comparison.

Use the library's verify function and a dummy verifier path for unknown accounts when appropriate. Return the same public error for an unknown account and wrong password. On success, create a fresh session identifier and do not accept a client-supplied pre-login session as authenticated.

Keep the visible contract boring:

```
{ "error": "Invalid email or password" }
```

The status, response body, redirect, and broad timing behavior should not reveal whether the account exists, is disabled, or used the wrong password. Internal logs may record a safe reason code, but never the password or session token.

Handle failures in this order:

1. Validate input shape and length.
2. Apply cheap request-size, network, and global abuse limits.
3. Load the account or a dummy verifier.
4. Run the password library's comparison.
5. Apply account-specific policy without changing the public outcome.
6. Create a new session on success.

The early limits protect CPU and memory from requests that should never reach an expensive password hash. Account-specific throttling still happens after a uniform real-or-dummy verification path so the response does not become an account-existence oracle.

Do not reuse an anonymous session identifier after login. Rotation prevents an attacker who supplied or learned the earlier value from inheriting the authenticated session.

A database or hashing-service failure is not "invalid credentials." Return a generic temporary failure and alert internally so an outage does not look like thousands of bad passwords.

Test success, unknown email, wrong password, disabled account, storage failure, and a client-supplied pre-login cookie. Only success may create a fresh session.

Defend against automated login

Add layered throttling, monitoring, breached-password checks, and optional MFA without creating a permanent account-lockout attack.

Attackers reuse leaked credentials and automate guesses. Password correctness alone is not an adequate abuse boundary.

Rate-limit by account and network signals, add progressive delays, log suspicious patterns, and notify users of meaningful risk. Block known-compromised passwords during creation or change without sending the full password to a third party. Avoid simple permanent lockouts an attacker can trigger.

Different attacks leave different evidence:

- **Brute force:** many passwords against one account.

- **Password spraying:** one password against many accounts.
- **Credential stuffing:** leaked email/password pairs tried at scale.

One IP limit misses distributed attacks and can punish a school or office behind shared NAT. One account limit lets an attacker lock out a victim. Combine short-window network, account, device, and global signals with progressive responses.

Start with small delays and monitoring. Escalate to temporary challenges or step-up authentication when risk rises. Keep a clear recovery path for the real user and avoid telling the attacker whether an account exists.

Record reason codes, rate-limit decisions, and coarse network information with a retention policy. Alert on changes in success rate, distributed failures, and attempts against high-value accounts. Never log submitted passwords.

Test controls with concurrent requests. A counter updated only in process memory can reset across instances or deployments and fail exactly when traffic spreads out.

Design thresholds and responses for one failed login, repeated failures for one account, and distributed attempts across many accounts. Explain the denial-of-service tradeoff of every limit.

Check your understanding: password authentication

Check signup validation, password policy, Argon2id, salts, login comparison, generic errors, session rotation, throttling, and credential stuffing defenses.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Sessions and authorization

Authenticate requests, enforce ownership, prevent CSRF, and manage session trust.

Authenticate every request

Resolve a session cookie into trusted identity context and reject missing, expired, revoked, or malformed sessions consistently.

Each protected HTTP request is independent. Middleware must validate the presented session before handlers rely on an identity.

Read the cookie, validate the opaque identifier's encoding and length, hash it if the stored lookup key is hashed, load the server record, check expiry and revocation, and attach a minimal identity context. Authority comes from the server record, never by parsing claims from an opaque token. Do not place session tokens in URLs or logs.

Keep the middleware result small:

```
type AuthContext = {
  userId: string
  sessionId: string
  authenticatedAt: Date
}
```

Do not copy roles or account state into long-lived request-independent memory. Authorization-relevant data can change. Load it at the correct boundary or include a version that forces stale sessions to refresh.

Distinguish absence from infrastructure failure. A missing or invalid session normally becomes `401 Unauthorized`. A session-store outage is a server failure; treating it as anonymous can accidentally expose a route designed for both public and private results.

Run authentication before reading protected request data or starting expensive work. The handler should receive either a trusted context or no execution at all.

Cache carefully. A shared HTTP cache must never serve one authenticated user's response to another. Mark private responses correctly and include identity in any application cache key.

Add middleware to the Books API and protect create, update, delete, and private list routes. Revoke a session between two requests and confirm no protected handler runs on the second one.

Authorize resource ownership

Enforce ownership in server-side queries so one authenticated user cannot read or modify another user's books by changing an ID.

Authentication does not make every resource available to every signed-in user. Object identifiers are not authorization tokens.

Query or mutate by both resource ID and trusted owner ID. Return a deliberately chosen not-found or forbidden response without leaking unnecessary existence detail. Apply the rule to reads, writes, exports, and bulk routes.

```
UPDATE books
SET title = ?
WHERE id = ? AND owner_id = ?
```

Check the affected-row count. Zero means the book did not exist *for this owner*. The handler should not run a second unrestricted query merely to produce a more detailed error unless the product truly needs that disclosure.

Reads need the same predicate:

```
SELECT id, title
FROM books
WHERE id = ? AND owner_id = ?
```

Filtering a returned object in JavaScript is too late if the query already loaded another user's private data. Put the boundary as close to the data operation as possible.

Indirect paths are easy to miss. Search, exports, attachments, counts, batch updates, background jobs, and WebSocket subscriptions all need ownership or role checks. A UI that hides another user's button is not authorization; an attacker can call the route directly.

For administrative access, make the elevated rule explicit and auditable. Do not scatter `isAdmin || ownerId === userId` throughout handlers. Centralize the policy and test both sides.

Create two users and test every resource route with the other user's book ID. Include list, export, update, and delete paths, and verify both the response and database state.

Protect cookie requests from CSRF

Use SameSite cookies, origin checks, and anti-CSRF tokens where needed for state-changing browser requests.

Browsers automatically attach matching cookies, including when a request begins on another site. That behavior creates cross-site request forgery risk.

SameSite is useful defense in depth but may not cover every application flow. Validate Origin or Fetch Metadata where appropriate and use a proven anti-CSRF token pattern for state-changing requests. Never use GET for actions that change state.

The attacker does not need to read the response if submitting the request changes data.

A synchronizer-token flow puts an unpredictable value in the legitimate page and requires it in the request:

```
<input type="hidden" name="csrfToken" value="session-bound-token">
```

The server compares it with a value tied to the session. A cross-site page can submit a form, but the same-origin policy normally prevents it from reading the protected page to obtain the token.

Check the `Origin` header for sensitive browser requests and reject a mismatched origin. Fetch Metadata headers can provide another signal. Define what happens when older or non-browser clients omit them instead of silently accepting every absence.

CSRF tokens do not cure XSS. Script running on your origin may read page tokens and act as the user. Keep output escaping and Content Security Policy in the wider defense.

Document every cookie-authenticated state transition, not just Books resource routes. Include logout, password and email changes, passkey or MFA enrollment and removal, and federated link or unlink. Login itself can need login-CSRF protection. OAuth callbacks use their transaction's `state` and OIDC `nonce` checks rather than an ordinary form token.

Test missing token, wrong token, wrong origin, a valid same-origin request, and a GET request that must remain read-only.

Rotate, revoke, and step up sessions

Renew session identifiers after privilege changes, implement logout, expire idle sessions, and require fresh proof for sensitive actions.

A session has a lifecycle. It should not remain equally trusted forever after one login.

Rotate identifiers after authentication and privilege changes, set idle and absolute expiry, revoke on logout and password reset, and let users end other sessions. Require recent authentication or MFA for sensitive actions such as email, password, or recovery changes.

Idle and absolute expiry solve different problems. Idle expiry closes abandoned sessions after inactivity. Absolute expiry limits the lifetime even when the session stays active.

Track lifecycle fields on the server:

```
created_at
last_seen_at
authenticated_at
expires_at
revoked_at
```

Update `last_seen_at` at a controlled interval rather than writing on every request. Use `authenticated_at` to decide whether the proof is fresh enough for a sensitive action.

Rotation should be atomic: create the new identifier, invalidate the old one, then send the new cookie. Test concurrent requests during rotation so an old request cannot restore the previous credential.

"Logout all devices" revokes every session for the account, including the current one unless the product says otherwise. Password reset should normally do the same or present that choice clearly.

Step-up authentication is action-specific. Reading a book may use the existing session, while changing recovery details requires a fresh password, passkey, or MFA proof.

Add logout-current and logout-all endpoints plus a recent-auth requirement for changing the account email. Verify old identifiers fail after each transition.

Check your understanding: sessions and authorization

Check session lookup and rotation, logout, ownership authorization, CSRF, XSS, cookie boundaries, and step-up authentication.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Federated login and passkeys

Use OIDC, authorization code with PKCE, safe account linking, and WebAuthn.

Separate OAuth from login

Understand OAuth authorization, OpenID Connect identity, and why an access token for one API is not automatically a login credential for another.

OAuth lets a client obtain delegated access to a protected resource. OpenID Connect adds an identity layer for authentication.

Use a provider's documented OIDC flow when the product goal is "sign in." Validate tokens for the intended issuer, audience, signature, and lifetime. Do not invent identity meaning from an arbitrary OAuth access token.

Imagine a calendar application asking permission to read your events. OAuth answers: "May this client call the calendar API with this scope?" It does not, by itself, define how your application should identify the person.

OIDC adds an ID token and standardized identity claims. The authorization server issues that token to a specific client after an authentication flow.

Keep artifacts in their lanes:

- the authorization code is short-lived and exchanged once
- the ID token describes an authentication event for the client
- the access token authorizes calls to its resource server
- the refresh token obtains new tokens and needs stronger storage controls

Do not send a provider access token to an unrelated API and call the response "logged in." Its audience, scopes, and validation rules belong to the resource server it was issued for.

Use a maintained OIDC library. Discovery, key rotation, signature algorithms, nonce, issuer, and audience validation are protocol work, not application string parsing.

Draw the browser, Books app, authorization server, and resource server for a federated login. Label each artifact, its audience, and the component allowed to receive it.

Use authorization code with PKCE

Start a modern authorization flow with state and PKCE, validate the callback, and exchange a one-time code without exposing tokens in URLs.

The authorization code flow sends the browser back with a short-lived code. PKCE binds that code to the client instance that began the flow.

Generate a secure state value and PKCE verifier, store them in a short-lived transaction, send the derived challenge, and validate state exactly on callback. Use exact registered redirect URIs and exchange the code server-side where the application architecture supports it.

The browser-visible request contains the challenge, not the verifier:

> **Diagram: Authorization code flow with PKCE.** The client stores a verifier, sends only its challenge through the browser, receives a code, and proves possession of the verifier during the token exchange. After the callback, the client sends the original verifier to the token endpoint. A party that steals only the authorization code cannot complete the exchange without it.

`state` and PKCE protect different links. `state` ties the callback to the browser transaction and helps prevent login CSRF. PKCE ties the code to the client that started the flow. Use both according to the provider and library guidance.

Store the transaction server-side or in a protected, short-lived mechanism. Bind it to the authorization server that was selected, and include the intended post-login destination after validating it as a safe local path. A client that supports several issuers must validate the callback issuer, or use distinct redirect URIs, so a response from one provider cannot be confused with a transaction started at another. Never accept an arbitrary callback `returnTo` URL and redirect to it.

Consume the transaction once. A second callback with the same state, code, or verifier should fail even inside the expiry window.

Implement a fake provider flow in tests. Cover wrong state, wrong verifier, wrong issuer, reused code, expired transaction, and an external return URL.

Link federated identities safely

Attach provider identities to local accounts without taking over an existing account through an unverified or colliding email address.

A provider subject identifier is the stable external identity key. Email can change and is not always sufficient proof that two accounts are the same person.

Store provider plus issuer-specific subject, verify the provider's required claims, and require authenticated confirmation before linking to an existing account. Treat link and unlink as sensitive operations with audit events and recovery implications.

A useful uniqueness key is:

```
issuer + subject
```

The displayed provider name is not enough. Two issuers could use the same subject string, and one provider may operate several issuers.

Do not automatically join accounts because email text matches. Providers differ in how they verify addresses, organizations can recycle addresses, and claims can change. If the user is already signed in, ask for recent authentication before linking. If not, require proof for both accounts through a deliberate merge flow.

Prevent one external identity from linking to two local users with a database uniqueness constraint. Record who initiated the link, when it happened, and which session approved it.

Unlinking can remove the user's only login method. Before allowing it, verify that another usable credential and recovery path remain. Notify the account through an existing trusted channel.

Design local users and external identities tables and the flow for linking a second provider. Test a colliding email and an attempt to link an already-linked subject.

Add passkeys with WebAuthn

Understand registration and authentication ceremonies, server challenges, relying-party scope, public-key storage, and recovery planning.

Passkeys use WebAuthn public-key credentials. The authenticator keeps the private key; the server stores credential identifiers and public-key material.

The server generates an unpredictable challenge for each ceremony and verifies origin, relying-party ID, challenge, signature, and required flags with a maintained WebAuthn library. Registration and authentication need HTTPS in production. Plan more than one credential and a secure recovery path.

Registration and authentication are separate ceremonies.

During registration, the server creates options with a fresh challenge and relying-party identity. The authenticator creates a credential for that relying party. The server verifies the response before storing the credential ID, public key, and metadata needed by its library.

During authentication, the server creates another challenge. The authenticator signs over that challenge and origin-bound data. The server verifies the signature with the stored public key.

Store challenges for a short time and consume them once. Bind each one to the session or account, ceremony type, expected origin, relying-party ID, and user-verification policy. A registration response cannot satisfy an authentication ceremony, and a challenge started in one browser session cannot be redeemed in another. Verify user presence and user verification according to policy, and ensure the returned credential and user handle belong to the intended account. Check the library's counter and backup-state behavior without assuming every authenticator uses counters the same way.

Passkeys resist phishing because credentials are scoped to a relying-party ID. They do not remove account-lifecycle work. Users change devices, sync providers, lose authenticators, and need to name or remove credentials.

Require recent authentication to add a passkey to an existing account. Otherwise, a stolen session

can become a durable new login method.

Test sign-in, replayed challenge, a challenge swapped between two sessions, wrong ceremony type, wrong origin, wrong relying-party ID, credential removal, and an attempt to enroll through an old session.

Check your understanding: federated login and passkeys

Check OAuth versus OIDC, authorization code flow, PKCE, state, redirect validation, account linking, WebAuthn ceremonies, and passkey recovery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Recovery and operations

Verify email, reset passwords, design MFA recovery, monitor, test, and respond.

Verify email ownership

Confirm that a user controls an email address with a short-lived single-use token without granting more authority than verification requires.

An email address should not be treated as verified merely because signup accepted its syntax.

Generate a strong random token, store only its hash with purpose and expiry, send the raw token in one HTTPS link, and consume it once. Rate-limit resends and return generic responses that do not reveal account existence.

Treat verification as a narrowly scoped capability. A token created to verify an email must not also reset a password, sign a user in, or approve an email change. Store enough state to enforce that boundary:

```
token_hash | user_id | target_email | version | purpose | expires_at | consumed_at
```

Hashing the token protects users if this table leaks: the raw value exists only in the email and the browser request. On redemption, look up the hash, verify the exact target address, current version, purpose, and expiry, then mark the token consumed and that address verified in one transaction. Two simultaneous clicks should not both succeed.

An email-change flow needs an additional decision. Confirming the new address proves control of that address, but it does not prove the request came from the account owner. Require a recent authenticated session before starting the change, notify the old address, and increment the pending-address version when it changes. Define resend policy explicitly: either keep earlier links valid until expiry or atomically invalidate them when issuing the new token.

Links also leak through logs, analytics, referrers, and support screenshots. Avoid third-party scripts on the redemption page, never log query strings containing tokens, and replace the URL after consuming it.

Implement the request and consume endpoints, then test expiration, replay, simultaneous redemption, resend invalidation, and changing the pending email before redemption. The final assertion should check the exact authority granted: only the intended address becomes verified.

Build password recovery

Create an enumeration-resistant reset flow with hashed single-use tokens, normal password policy, session invalidation, and user notification.

Password recovery is an alternate authentication path and is often the easiest route around a strong login system.

Return the same request response for existing and unknown accounts, generate a short-lived token, store its hash, and apply the normal password hashing policy on reset. Do not automatically log the user in afterward; revoke existing sessions and notify the account owner.

Keep the public request path deliberately boring:

```
POST /forgot-password
→ "If an account exists, we sent instructions."
```

The status, body, and approximate timing should be similar whether the address exists. Still rate-limit by account and network signals. A generic response prevents easy enumeration; it does not make an unlimited email-sending endpoint safe.

The reset token should be random, short-lived, single-use, and bound to one user and one purpose. Store its hash rather than the raw value. When the user submits a new password, consume the token and replace the password hash atomically. If either operation can commit alone, concurrent requests can reuse the same link.

Build the link from a configured or allowlisted canonical HTTPS origin, never an untrusted `Host` header. Keep third-party scripts off the reset page, send `Referrer-Policy: no-referrer`, and make sure request logs do not capture token-bearing query strings. Replace the visible URL after the server has accepted the token.

Do not weaken the normal password policy during recovery, and do not send a new password by email. After success, revoke the user's existing sessions so a thief who already has a session cannot remain signed in. Let the user log in normally, and send a notification that explains how to report an unexpected reset without including secrets.

Operational failures need explicit behavior. If the email provider is down, avoid claiming that a message was sent while silently dropping the request. Record a safe delivery job or return a retryable generic result without exposing whether the account exists.

Test unknown and known addresses, an altered host header, expired and reused tokens, two concurrent reset submissions, password replacement, session invalidation, delivery failure, and notifications. A good integration test proves the old password and every old session stop working after the transaction commits.

Design MFA and recovery

Add a second factor and recovery mechanism without making the fallback weaker than the primary authentication path.

MFA reduces damage from a stolen password, but enrollment, replacement, and recovery become new sensitive workflows.

Prefer phishing-resistant factors such as passkeys where the product allows them. Protect factor changes with recent authentication, notify the user, provide one-time recovery codes stored as hashes, and avoid security questions as a secret factor.

A second factor helps only when it is independent enough from the first. A password plus a code sent to the same compromised email account is weaker than a password plus a passkey held on another device. TOTP applications resist password reuse, but users can still type a code into a phishing site. Passkeys bind authentication to the site origin and provide stronger phishing resistance.

Enrollment is a security-sensitive state transition. Require recent proof with an existing enrolled factor when one is available, prove possession of the new factor, and notify the user through an existing channel. Do not mark TOTP active merely because a secret was displayed; require one valid code first. Protect recoverable TOTP seeds like credentials, accept only a small documented clock window, and reject replay of a code already used in that window. Factor removal and replacement deserve at least the same protection as enrollment.

Recovery codes are credentials, not convenience text. Generate high-entropy one-time values, show them once, store hashes, and consume each atomically. Avoid security questions: their answers are often public, guessable, reusable, and hard to rotate.

Support can accidentally become a universal bypass. Document what support staff may verify and what they may never override. High-value accounts may need delayed recovery, multiple proofs, or manual review, but invented personal trivia is not strong evidence.

Draw the complete lifecycle: enroll, challenge, add a second device, regenerate recovery codes, lose a device, remove a factor, and recover the account. For every arrow, state what proof is required, what notification is sent, which sessions are revoked, and how an attacker might abuse the transition.

Operate and test authentication

Monitor security events, review active sessions, rotate secrets, patch dependencies, test abuse cases, and prepare an incident response for credential or session compromise.

Authentication is a living production system. Algorithms, providers, dependencies, threats, and user sessions change after launch.

Log safe security events without credentials, alert on meaningful anomalies, let users review and revoke sessions, rotate signing and provider secrets, and keep auth libraries current. Test enumeration, CSRF, XSS impact, authorization, rate limits, reset, session theft, and rollback behavior.

Logs should describe decisions without becoming a second credential store. Useful events include login success and failure, lockout or challenge activation, session creation and revocation, password reset, factor changes, and authorization denial. Record stable internal identifiers and coarse context; never record passwords, raw session IDs, reset links, authorization codes, or recovery codes.

Alert on patterns tied to an abuse hypothesis, not every failure. A burst across many accounts suggests password spraying. Many networks failing against one account suggests targeted guessing. A new country immediately followed by factor removal suggests takeover. Thresholds need production tuning and a way for support to see why a user was challenged.

Users should be able to inspect active sessions with understandable device and recent-activity information, then revoke one or all of them. Administrators need an audited emergency path for global revocation. Secret rotation needs overlap rules: introduce the new signing key, accept the old key only for a bounded period, then remove it. A runbook should say who can perform each step.

Test properties across the whole system, not just endpoint success cases. Compare observable responses for account enumeration, replay reset and OAuth callbacks, race two token redemptions, attempt CSRF on every cookie-authenticated mutation, mutate resource identifiers, simulate a

stolen session, and fail the session store and email provider. Verify that outages fail closed where authority is uncertain without turning every dependency problem into a user lockout.

Write two short incident playbooks: one for a stolen session database and one for a compromised OAuth client secret. Include detection evidence, containment, credential or session revocation, user communication, recovery, and the follow-up change that prevents recurrence. Then turn the key checks into a release checklist owned by a named team rather than an unactioned document.

Check your understanding: recovery and operations

Check email verification, reset tokens, MFA recovery, security events, session review, secret rotation, tests, dependency updates, and incident response.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/web-authentication/.