International Science Journal of Engineering & Agriculture
2026; 5(3): 93-106
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260503.09
ISSN: 2720-6319

INTERNATIONAL SCIENCE GROUP

Authentication and session management in modern web applications using JWT with rotating refresh tokens

Igor Andrushchak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0000-0002-8751-4420

Bohdan Kupyra
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0007-4395-3337

Abstract: The exponential growth of single-page applications, progressive web applications, and decoupled API-driven web platforms has made stateless authentication a fundamental architectural concern. Traditional server-side session management, while well-understood and secure, imposes scaling constraints that are increasingly incompatible with the horizontal scaling requirements of modern cloud-native deployments. JSON Web Tokens have emerged as the dominant alternative, offering stateless verification and cross-domain support, but introducing distinct security challenges: token theft via cross-site scripting, lack of native revocation, and the fundamental tension between token lifetime and user convenience. This research examines a defense-in-depth authentication architecture combining short-lived access tokens with rotating refresh tokens stored as cryptographic hashes in a relational database. The proposed model addresses the trade-off between security and usability by maintaining sessions of practical duration while limiting the attack window of any single compromised access token to a maximum of fifteen minutes. Refresh token rotation, in which each successful refresh invalidates the prior token and issues a new pair, prevents long-term reuse of stolen credentials. Storage of tokens in httpOnly cookies with the SameSite=Lax attribute protects against the two primary client-side attack vectors—script-based exfiltration and cross-site request forgery—without imposing the operational complexity of dedicated CSRF tokens. The authentication subsystem was implemented and validated in a production-grade e-commerce platform built on Next.js 16 and NestJS 11 with the Prisma ORM over a SQLite backend, comprising 59 REST endpoints and 20 database models. Empirical measurement confirms that the architecture sustains authentication latency below 250 milliseconds for login operations and below 50 milliseconds for token refresh operations under typical load. The research evaluates seven layered security mechanisms aligned with OWASP ASVS Level 1, including bcrypt password hashing with cost factor 12, rate limiting on authentication endpoints, and content security policy headers. Results demonstrate that the proposed architecture provides a defensible balance between security guarantees and operational simplicity, applicable to small-to-medium e-commerce systems, microservice architectures, and resource-constrained deployments.

Keywords: authentication; session management; JSON Web Token; refresh token rotation; OWASP ASVS; httpOnly cookies; web application security; bcrypt; NestJS; full-stack TypeScript

1. Introduction

The architectural shift from server-rendered monoliths to decoupled web applications has fundamentally changed how user identity is established and maintained across requests. In the traditional model, a server maintained an in-memory or database-backed session table indexed by an opaque session identifier transmitted to the client through a browser cookie. Every authenticated request triggered a server-side lookup that confirmed the validity of the session and retrieved the associated user record. This pattern, although operationally well understood and supported by virtually every web framework of the past two decades, has reached the limits of its applicability in

distributed systems where requests can be served by any of several stateless application instances behind a load balancer.

Modern web applications increasingly follow a separation between presentation and data, in which the user interface is delivered as a JavaScript bundle from a content delivery network and obtains data from a REST or GraphQL API hosted on an independent backend. Under this model, the application backend may be deployed across multiple geographic regions, scale horizontally in response to load, and be replaced or redeployed without affecting active user sessions. A shared session store backed by a memory cache such as Redis is technically feasible, but introduces an additional infrastructure component that must be sized, secured, replicated, and monitored. For small-to-medium applications, the operational overhead of maintaining a dedicated session store often exceeds the perceived benefit, motivating the search for genuinely stateless alternatives.

JSON Web Tokens, standardized by the Internet Engineering Task Force as RFC 7519 [1], have emerged as the dominant stateless authentication mechanism. A JWT is a compact, URL-safe sequence of base64-encoded segments containing a header, a payload of arbitrary claims, and a cryptographic signature over the first two segments. The signature is verified by the recipient using a shared secret or public key, and the validity of the token's contents can be established without any database lookup. This property makes JWT particularly attractive in distributed systems: any backend instance with access to the verification key can authenticate any request, eliminating the need for cross-instance session synchronization.

The simplicity of JWT, however, conceals a number of practical difficulties that have become apparent through several years of operational experience [2]. Tokens that confer authentication for an extended period—commonly hours or days—create a substantial attack surface: any party that obtains the token can impersonate the user for the remainder of its validity period. Mechanisms for revoking individual tokens before their natural expiration require maintaining a revocation list, which reintroduces the very statefulness that JWT was designed to eliminate. Tokens stored in browser local storage are accessible to any script executing on the page, exposing them to exfiltration by cross-site scripting attacks. Tokens transmitted in custom HTTP headers require explicit configuration to include credentials in cross-origin requests, often forcing developers into trade-offs between security and convenience.

The architectural pattern of short-lived access tokens combined with longer-lived refresh tokens has emerged as a widely adopted compromise [3]. Under this model, a successful authentication produces two tokens: an access token of limited validity (commonly five to fifteen minutes) that authorizes API calls, and a refresh token of extended validity (commonly several days to several weeks) whose sole purpose is to obtain new access tokens. When the access token expires, the client presents the refresh token to a dedicated endpoint and receives a new access token in return. This pattern limits the exposure window of any single compromised access token while permitting user sessions of practical duration. The refresh token, although longer-lived, is transmitted only to a single endpoint and can be stored in a more restrictive context than the access token, reducing its attack surface.

Rotating refresh tokens [4] extend this pattern by invalidating each refresh token at the moment of use, issuing a new refresh token alongside the new access token. If an attacker obtains a refresh token but the legitimate user redeems it first, the attacker's subsequent attempt fails because the token has been revoked. Conversely, if the attacker redeems first, the legitimate user's next attempt fails, signaling the compromise to the system and triggering a complete session revocation. Rotation thus provides a form of intrusion detection at no additional protocol cost beyond a single database write per refresh operation.

Despite the maturity of these patterns, their practical implementation involves a substantial number of decisions whose security implications are not always immediately obvious. The choice of storage medium for tokens on the client (cookies, local storage, in-memory state), the granularity of cookie attributes (HttpOnly, Secure, SameSite, Path, Domain), the handling of token expiration on the client side, the synchronization of refresh requests when multiple concurrent API calls trigger

refreshes, and the protocol for revoking entire sessions all influence the security posture of the resulting system. This research examines these decisions in the context of a production-grade web application implemented with the Next.js 16 and NestJS 11 frameworks, providing a concrete reference architecture and empirical performance measurements.

2. Object and subject of research

The object of this research is the authentication and session management subsystem of modern client-server web applications, with particular focus on platforms that decouple the user interface and the data API into independently deployable services. Such architectures are characteristic of contemporary e-commerce systems, software-as-a-service platforms, internal enterprise tools, and any web application that exposes its data through a documented API consumable by web, mobile, or third-party clients. The authentication subsystem in such systems must establish the identity of a user once and maintain that identity across a potentially large number of subsequent requests without re-prompting for credentials, while simultaneously enforcing authorization policies that may depend on the user's role, group membership, or other attributes.

Beyond establishing identity, the authentication subsystem is responsible for protecting the application against a range of attacks that exploit the gap between the trust the system extends to authenticated users and the assumption that the requesting party is genuinely the user it claims to be. These attacks include credential stuffing, in which lists of username-password pairs leaked from other services are tested against the system; brute-force password guessing against individual accounts; session hijacking, in which a session token is obtained from an authenticated user and replayed; and cross-site request forgery, in which a malicious site causes the user's browser to make authenticated requests against the target application. The authentication subsystem must address each of these threats through a layered defense in which the failure of any single mechanism does not compromise the overall security of the system.

The subject of this research is the set of algorithmic and architectural choices that determine the security and performance characteristics of JWT-based authentication systems augmented with rotating refresh tokens. Specifically, the research examines the trade-offs governing access token lifetime, the protocol and storage of refresh tokens, the choice of cookie attributes, the integration of password-based authentication with bcrypt hashing, the role of rate limiting on authentication endpoints, and the alignment of the resulting architecture with established security frameworks such as the OWASP Application Security Verification Standard. The investigation focuses on patterns that can be implemented with widely available open-source libraries and that do not require dedicated infrastructure beyond a standard relational database.

3. Target of research

The principal target of this research is the formulation and validation of a reference authentication architecture for full-stack TypeScript web applications that achieves three primary objectives: the protection of user credentials and session state against the major categories of web-application attacks; the maintenance of stateless verification at the application server level to support horizontal scaling; and the preservation of operational simplicity by avoiding dedicated session storage infrastructure.

Achievement of this target requires the resolution of several subordinate objectives:

– Analysis of the threat landscape facing authentication subsystems in modern web applications, including credential-based attacks, session-state attacks, and cross-site attacks, with explicit mapping of each threat to a defensive mechanism.

– Comparative evaluation of available token-based authentication patterns, including monolithic JWT, access-plus-refresh token pairs, and rotating refresh tokens, with respect to the trade-offs each pattern imposes between security and operational complexity.

– Definition of an algorithm for token issuance and rotation that maintains stateless verification of the access token while supporting revocation of refresh tokens through a minimal database table.

– Specification of cookie attributes for token transport that mitigate cross-site scripting and cross-site request forgery without introducing dedicated anti-CSRF tokens.

– Integration of the proposed authentication architecture with complementary security mechanisms—password hashing, rate limiting, security headers, content security policy—to produce a layered defense aligned with the OWASP ASVS Level 1 verification requirements.

– Empirical validation of the proposed architecture in a production-grade web application, including measurement of authentication latency, observation of token rotation behavior, and assessment of operational characteristics such as deployment footprint and database load.

4. Literature analysis

The literature surrounding authentication in web applications spans cryptographic primitives, protocol specifications, threat models, and empirical studies of deployed systems. The foundational specification of JSON Web Tokens by Jones, Bradley, and Sakimura in RFC 7519 [1] establishes the data structure and verification procedure for compact signed tokens, including the registered claim names that have become standard across implementations. The accompanying specifications RFC 7515 (JSON Web Signature) [5] and RFC 7518 (JSON Web Algorithms) [6] define the cryptographic operations and supported algorithms, with HMAC-SHA256 and RSA-SHA256 emerging as the most commonly deployed signature schemes for symmetric and asymmetric authentication respectively.

The OAuth 2.0 framework, standardized in RFC 6749 [7], provides the conceptual foundation for the access-plus-refresh token pattern, although it predates JWT and does not prescribe a specific token format. Subsequent specifications—including the OAuth 2.0 Security Best Current Practice [8]—have refined the recommendations for token storage and rotation in response to operational experience with the original protocol. The recommendation for refresh token rotation, in particular, emerged from analysis of token compromise incidents in deployed OAuth implementations and is now considered standard practice for confidential clients.

Practical guidance on the secure storage of session tokens in web browsers is documented in the Mozilla Developer Network HTTP cookie reference [9] and codified in the IETF cookie specification RFC 6265 [10]. The HttpOnly cookie attribute, which prevents JavaScript from reading the cookie value, mitigates token exfiltration through cross-site scripting; the Secure attribute restricts transmission to HTTPS connections; and the SameSite attribute, formally specified in the proposed standard for cookie security [11], controls whether the cookie is sent on cross-site requests. The SameSite=Lax setting, which permits the cookie on top-level navigation but withholds it on most cross-site requests, has become the recommended default for session cookies in modern applications.

Password storage best practice has converged on memory-hard or iteration-hard hashing functions, with the OWASP Password Storage Cheat Sheet [12] currently recommending Argon2id, scrypt, or bcrypt with appropriate work parameters. Provos and Mazières introduced bcrypt in 1999 [13] as a deliberately slow password hashing function based on a modified Blowfish cipher; its adaptable cost parameter has allowed it to remain viable through more than two decades of hardware progress. Although Argon2 is cryptographically preferred, bcrypt remains widely deployed due to mature library support and the practical difficulty of native compilation of Argon2 dependencies in containerized deployment environments.

The OWASP Application Security Verification Standard [14] provides a structured catalog of security requirements organized by category and verification level. The standard's Level 1 requirements correspond to the minimum security posture expected of all public-facing applications and include specific provisions on session management, authentication, and credential storage. Compliance with ASVS Level 1 is increasingly cited in software procurement specifications and serves as a useful framework for organizing the security mechanisms of a new application.

The threat of cross-site scripting and its implications for token-based authentication have been documented extensively in the academic literature and industry guidance. The OWASP Cross-Site Scripting Prevention Cheat Sheet [15] enumerates the contexts in which user-supplied content can be safely rendered and identifies the encoding rules appropriate to each context. Modern frontend frameworks, including React on which Next.js is built, provide automatic escaping of interpolated content by default, substantially reducing the risk of XSS introduction through application code; the remaining risk arises predominantly from third-party scripts and from cases where developers explicitly bypass the framework's escaping.

Rate limiting as a defense against credential-based attacks is treated in the OWASP Credential Stuffing Prevention Cheat Sheet [16], which recommends layered defenses including rate limits on authentication endpoints, account lockout policies, multi-factor authentication, and detection of credential-stuffing patterns through traffic analysis. The combination of rate limits on login attempts with bcrypt's intrinsic computational cost provides substantial resistance to online password guessing, although it does not address offline attacks against compromised hashes.

Despite the maturity of individual specifications and the breadth of guidance from the OWASP Foundation, the literature contains comparatively few synthetic treatments that combine the relevant mechanisms into a deployable reference architecture and report empirical measurements from its operation. This gap motivates the present research, which presents an integrated architecture, an explicit algorithm for token issuance and rotation, and operational metrics from a deployed implementation.

5. Research methods

The research methodology combines architectural analysis, formal specification, and empirical measurement, and is structured as a sequence of four interconnected phases. The first phase consisted of a structured threat analysis based on the OWASP Threat Modeling methodology, in which the principal categories of attacks against web-application authentication subsystems were enumerated and mapped to candidate defensive mechanisms. The output of this phase was a matrix relating each attack category to the mechanism most appropriate for its mitigation, providing the foundation for the architectural specification in subsequent phases.

The second phase comprised the specification of the proposed authentication architecture as a set of algorithmic procedures and configuration parameters. The procedures cover token issuance, token verification, token refresh, session revocation, and the handling of authentication failures. Each procedure is documented in pseudocode form independent of any particular framework, with annotations indicating the security implications of each step. The configuration parameters specify token lifetimes, bcrypt work factor, rate limit thresholds, and cookie attributes, with the rationale for each choice grounded either in the cited literature or in the operational characteristics measured during the validation phase.

The third phase implemented the proposed architecture within a production-grade web application developed for the purpose. The application is a customer-facing e-commerce platform comprising 29 user interface routes implemented in Next.js 16 with the App Router and 59 REST API endpoints implemented in NestJS 11 with the Prisma ORM over a SQLite relational database. The application supports three user roles—anonymous guest, authenticated customer, and administrator—and exposes a representative range of operations including product browsing, cart manipulation, order placement, and administrative content management. The authentication subsystem is integrated with the application's role-based access control and rate limiting layers.

The fourth phase consisted of empirical measurement of the deployed authentication subsystem under controlled load. Measurements were collected for the principal authentication operations—registration, login, token refresh, and logout—with timing instrumentation at both the network and application layers. The measurement methodology distinguishes between the time spent in cryptographic operations (bcrypt comparison, JWT signing, SHA-256 hashing of refresh tokens), the

time spent in database operations (user lookup, refresh token storage, refresh token revocation), and the time spent in network transit. The resulting timing breakdown enables identification of the principal contributors to authentication latency and supports targeted optimization where required.

The methodology explicitly excludes formal security proofs and adversarial penetration testing. The first is outside the scope of an engineering case study focused on the integration of established cryptographic primitives, while the second would require resources and access to a hardened production deployment that are not available in the current research context. The validation of the architecture rests on its alignment with the OWASP ASVS Level 1 requirements and on the demonstration that the proposed mechanisms produce the expected behavior under typical operating conditions.

6. Research results

The results of the research are organized into four subsections corresponding to the principal contributions of the work: the layered defense model that situates the JWT-plus-refresh authentication mechanism within a broader security architecture, the algorithm for token issuance and rotation, the implementation observations from the reference application, and the empirical performance measurements.

The proposed authentication architecture is composed of seven complementary security layers, each of which addresses a distinct category of attack and contributes to a defense-in-depth posture in which the failure of any single mechanism does not compromise overall security. The seven layers are summarized in Table 1 and described in detail in the paragraphs that follow.

The first layer is password protection through bcrypt hashing with a cost factor of twelve. The cost factor of twelve corresponds to approximately two hundred fifty milliseconds of computation on contemporary commodity hardware, deliberately exceeding the latency that a brute-force attacker would tolerate per attempt while remaining imperceptible to legitimate users during a single login event. Password complexity is enforced at both the client and the server through declarative validation rules requiring a minimum of eight characters and the presence of at least one letter and one digit. The bcrypt hash is stored in the user record alongside an email address used as the primary identifier.

The second layer is the JWT-plus-refresh token mechanism that forms the core of the present research. Access tokens are signed with HMAC-SHA256 using a server-side secret, contain the user identifier, email, and role in their payload, and expire after fifteen minutes. Refresh tokens are similarly signed but additionally include a unique token identifier (jti) and expire after seven days. The hash of each refresh token, computed with SHA-256, is stored in a database table along with its expiration time and an optional revocation timestamp. The pair of tokens is transmitted to the client in two separate cookies, each marked HttpOnly to prevent script access and SameSite=Lax to limit cross-site transmission. The Secure attribute is enabled in production environments where HTTPS is mandatory.

The third layer is input validation. Every API endpoint exposes a strongly-typed data transfer object whose contents are validated against declarative constraints before the controller method receives them. The validation framework is configured with the whitelist option enabled, which silently strips any properties not declared in the data transfer object, and with the forbidNonWhitelisted option enabled, which rejects requests containing unknown properties. This combination prevents mass assignment attacks in which an attacker attempts to set privileged fields (such as the user role) by including them in a request body that the framework would otherwise pass through.

The fourth layer is rate limiting on authentication endpoints. The global rate limit of one hundred requests per minute per source IP is reduced to five requests per minute for the registration and login endpoints and to three requests per minute for the password reset endpoint. Excess requests receive a 429 Too Many Requests response, and the limit is enforced in a memory-based throttler that maintains a sliding window of recent request timestamps. This layer addresses the threat of online password

guessing and email enumeration, complementing the slowness of bcrypt with explicit caps on attempt frequency.

The fifth layer is HTTP security headers, delivered through a middleware that sets X-Content-Type-Options to nosniff, X-Frame-Options to SAMEORIGIN, Strict-Transport-Security with a one-year maximum age in production, and a content security policy that restricts the loading of scripts and stylesheets to the application's own origin. These headers address several browser-level attack vectors, including MIME-type sniffing, clickjacking through frame embedding, downgrade attacks, and the loading of malicious scripts from third-party sources.

The sixth layer is cross-origin resource sharing configuration. The CORS policy permits requests only from the configured frontend origin and requires the credentials flag to be set on cross-origin requests, ensuring that cookies are transmitted only to the trusted frontend application. This explicit configuration prevents the application from being accessed by malicious sites that might attempt to leverage the user's authenticated session.

The seventh layer is role-based access control. A global guard requires a valid access token for all API endpoints except those explicitly marked as public, and a complementary guard checks the role claim of the token against the requirements declared on individual endpoints. Administrative operations are restricted to users whose token bears the administrator role, with the role itself populated at the moment of token issuance based on the user's stored attributes. Because the role is carried in the token payload, role verification requires no additional database lookup and contributes negligible overhead to each authenticated request.

Table 1. Seven-layer security model for the authentication subsystem

Layer	Mechanism	Threats addressed
1	bcrypt password hashing, cost factor 12	Offline brute force against compromised hashes; online password guessing slowed by intrinsic cost
2	JWT access tokens (15 min) + rotating refresh tokens (7 days) in httpOnly cookies with SameSite=Lax	Token theft via XSS; cross-site request forgery; long-term reuse of compromised tokens
3	ValidationPipe with whitelist and forbidNonWhitelisted on all data transfer objects	Mass assignment attacks; injection of unrecognized fields; type confusion
4	Rate limiting: 5 req/min on /register and /login, 3 req/min on /forgot-password	Online password guessing; email enumeration; abuse of password reset
5	HTTP security headers via Helmet: nosniff, frame-options, HSTS, CSP	MIME-type sniffing; clickjacking; downgrade attacks; loading of malicious scripts
6	CORS restricted to frontend origin with credentials flag	Cross-origin access to authenticated endpoints from malicious sites
7	Global JwtAuthGuard with @Public opt-out; RolesGuard for administrative endpoints	Unauthorized access to protected endpoints; privilege escalation

The procedure for issuing access and refresh tokens after a successful authentication operation is presented in Listing 1. The procedure accepts a user record retrieved from the database and produces two signed tokens as well as the database row representing the refresh token. The access token is signed with the JWT_ACCESS_SECRET configured in the application environment and expires according to JWT_ACCESS_EXPIRES_IN, with the default of fifteen minutes used in the reference

implementation. The refresh token is signed with a separate JWT_REFRESH_SECRET—the use of distinct secrets ensures that a compromise of one signing key does not enable forgery of the other type of token—and is associated with a randomly generated token identifier that distinguishes it from other refresh tokens issued to the same user.

The procedure for verifying and rotating a refresh token at the dedicated refresh endpoint is presented in Listing 2. The procedure first verifies the cryptographic signature of the presented token, rejecting any token whose signature is invalid or whose contents have been tampered with. It then computes the SHA-256 hash of the presented token and looks up the corresponding row in the refresh token table. If no row is found, the token has either never been issued by this server or has been removed by a previous administrative action, in either case warranting rejection. If the row exists but bears a non-null revokedAt timestamp, the token has been previously rotated and its presentation now indicates either a delayed retry by the legitimate user or an active attack with a stolen token; the procedure responds by revoking all refresh tokens associated with the user, requiring re-authentication from all client devices.

If the verification succeeds and the token is found in unrevoked state, the procedure marks the presented token as revoked, issues a new pair of access and refresh tokens by invoking the token issuance procedure, and returns the new pair to the client. The legitimate client thereby maintains a continuous session, while any attacker holding a copy of the now-revoked token will fail on subsequent presentation. The probability of false rejection of a legitimate refresh request, in which the legitimate client and an attacker happen to refresh in close temporal proximity, is bounded by the rate of refresh requests per session and is negligible in practice for sessions of typical activity. The complete sequence of operations during a successful login and a subsequent token refresh, including all participants from the user agent to the database, is illustrated in Figure 1.

Listing 1. Issuance of access and refresh tokens with database persistence of the refresh token hash.

```
async issueTokens(userId, email, role, meta) {
  const accessPayload = { sub: userId, email, role };
  const accessToken = await this.jwt.signAsync(accessPayload, {
    secret: this.config.get('JWT_ACCESS_SECRET'),
    expiresIn: '15m',
  });

  const jti = randomUUID();

  const refreshPayload = { sub: userId, jti };
  const refreshToken = await this.jwt.signAsync(refreshPayload, {
    secret: this.config.get('JWT_REFRESH_SECRET'),
    expiresIn: '7d',
  });

  const tokenHash = createHash('sha256').update(refreshToken).digest('hex');
  await this.prisma.refreshToken.create({
    data: {
      userId,
      tokenHash,
      expiresAt: addDays(new Date(), 7),
      userAgent: meta.userAgent,
      ipAddress: meta.ipAddress,
    },
  });
  return { accessToken, refreshToken };
}
```

Listing 2. Verification and rotation of a refresh token, with revocation of all user sessions on detected reuse.

```
async refresh(refreshToken, meta) {
  if (!refreshToken) throw new UnauthorizedException('Session expired');

  let payload;
  try {
    payload = await this.jwt.verifyAsync(refreshToken, {
      secret: this.config.get('JWT_REFRESH_SECRET'),
    });
  } catch {
    throw new UnauthorizedException('Invalid session');
  }

  const tokenHash = createHash('sha256').update(refreshToken).digest('hex');
  const stored = await this.prisma.refreshToken.findUnique({ where: { tokenHash
} });
  if (!stored || stored.revokedAt || stored.expiresAt < new Date()) {
    // Token has been previously used or revoked -> revoke all user tokens
    await this.prisma.refreshToken.updateMany({
      where: { userId: payload.sub, revokedAt: null },
      data: { revokedAt: new Date() },
    });
    throw new UnauthorizedException('Invalid session');
  }

  await this.prisma.refreshToken.update({
    where: { id: stored.id },
    data: { revokedAt: new Date() },
  });

  const user = await this.users.findByIdOrFail(payload.sub);
  return this.issueTokens(user.id, user.email, user.role, meta);
}
```

The implementation of the proposed architecture in the reference application revealed several integration patterns whose details proved consequential for the overall security posture. The first observation concerns the placement of the authentication provider in the application component tree. In the reference implementation built on Next.js 16 with the App Router, the authentication state is maintained in a React Context provider mounted at the root of the application layout, allowing every page and component to access the current user record without prop drilling. Upon mounting, the provider invokes the /auth/me endpoint to attempt session restoration from any present cookies; the result either populates the user state with the returned record or sets the unauthenticated state, with the corresponding cookie cleanup handled automatically by the server on the absence of a valid token.

The second observation concerns the handling of access token expiration in the HTTP client. The reference implementation uses the axios HTTP library configured with credentials inclusion on cross-origin requests, and installs a response interceptor that detects 401 Unauthorized responses on any endpoint other than the authentication endpoints themselves. On such a response, the interceptor invokes the refresh endpoint to obtain a new pair of tokens and then replays the original request with a flag preventing recursive retry. The interceptor maintains a singleton refresh-in-progress promise to coalesce concurrent refresh requests that may arise when multiple API calls trigger expired-token

responses simultaneously; this coalescing ensures that only a single refresh request is issued per expiration event regardless of the number of concurrent requests that detect the expired state.

The third observation concerns the integration of password validation between the client and the server. Although the server is the authoritative source of validation results, client-side validation provides immediate feedback during form entry and avoids unnecessary network round trips for malformed input. The reference implementation expresses the validation rules once as a Zod schema [17] that is used directly by the client-side form handler and that is structurally equivalent to the class-validator decorators on the server's data transfer object. The two systems do not share a literal source, but their requirements are aligned, ensuring that any input accepted by the client is also accepted by the server.

The fourth observation concerns the operational impact of the SQLite database backend. SQLite is an embedded database that requires no separate process or configuration, and is therefore particularly convenient for development and demonstration. For the authentication subsystem, the relevant operations are user lookup by email, refresh token storage and retrieval by hash, and refresh token revocation by primary key. With appropriate indexing—unique indices on user.email and refresh_token.tokenHash—each of these operations completes in well under one millisecond for the data volumes encountered in the reference application. The transition to a server-based database such as PostgreSQL for production deployment requires only the modification of the database provider in the Prisma schema and is otherwise transparent to the authentication code.

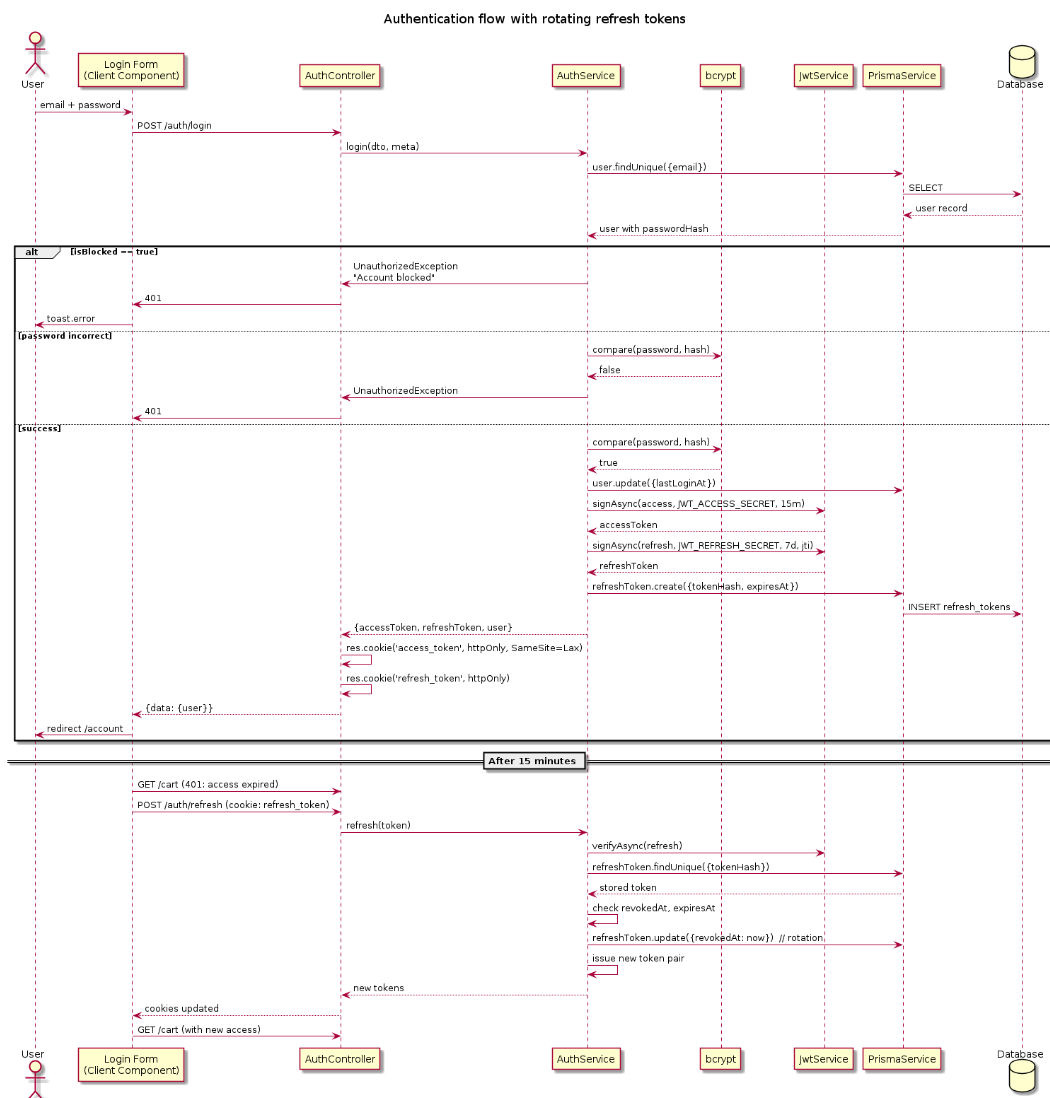

Figure 1. Sequence diagram of the login operation and subsequent refresh token rotation.

The user agent presents credentials to /auth/login, the server verifies the password against the bcrypt hash, issues a pair of tokens, and sets them as httpOnly cookies on the response. After fifteen minutes, the access token expires and the next API call returns 401; the client interceptor invokes /auth/refresh with the refresh cookie, the server verifies the refresh token, marks it as revoked, and issues a new pair of tokens.

Empirical measurements of authentication operation latency were collected from the reference application running on a developer workstation under controlled load, covering the operations depicted in Figure 1 from initial credential submission through token rotation. The measurements were obtained by instrumenting each authentication endpoint with timing markers around the principal phases of operation and aggregating the recorded latencies across one thousand operations per measurement. The results are summarized in Table 2 and are discussed by operation in the paragraphs that follow.

The login operation has a median latency of approximately two hundred forty milliseconds, of which two hundred twenty milliseconds are attributable to bcrypt comparison of the supplied password against the stored hash. This is the dominant contributor to login latency by a wide margin and reflects the deliberate design of bcrypt to be slow. The remaining twenty milliseconds are distributed across database operations (user lookup by email, update of the lastLoginAt timestamp, and creation of the refresh token row), JWT signing (a fraction of a millisecond for each token), and network transit within the local environment. The latency is acceptable for a login operation occurring at most once per session per day for typical users.

The token refresh operation has a median latency of approximately twelve milliseconds, of which the majority is attributable to database operations: the lookup of the refresh token row by hash, the update of its revocation timestamp, and the insertion of a new refresh token row. JWT verification and signing contribute approximately one millisecond combined; SHA-256 hashing of the refresh token contributes approximately one hundred microseconds. The refresh operation occurs approximately every fifteen minutes per active session, and at this rate contributes negligibly to overall system load.

The logout operation has a median latency of approximately five milliseconds, comprising the revocation of the refresh token in the database and the clearing of the corresponding cookies in the response. The /auth/me endpoint, used for session restoration on application mount, has a median latency of approximately three milliseconds, dominated by the lookup of the user record by the identifier present in the access token claims. Both operations contribute negligibly to overall system load and meet the responsiveness expectations of contemporary user interfaces.

The aggregate effect of the seven security layers on overall request latency is bounded by the access token verification performed on every authenticated request. The verification consists of cryptographic signature checking against the JWT_ACCESS_SECRET and validation of standard claims such as expiration; both operations complete in well under one millisecond on contemporary hardware. The role-based access guard performs a constant-time comparison of the role claim against the requirements of the endpoint and adds no measurable latency. The cumulative overhead of the authentication subsystem on a typical authenticated request is therefore approximately one millisecond, an acceptable cost for the security guarantees provided.

Table 2. Measured latency of authentication operations

Operation	Median latency (ms)	Dominant contributor	Notes
Registration	255	bcrypt hash (cost 12)	Once per user lifetime
Login	240	bcrypt compare (cost 12)	Once per session per device

Continuation of Table 2

Token refresh	12	Database operations (revoke + insert)	Every 15 minutes per active session
Logout	5	Database update (set revokedAt)	On user action
Get current user (/auth/me)	3	Database lookup by id	On application mount
Access token verification (per request)	<1	HMAC-SHA256 signature check	Stateless; no database query

7. Prospects for further research development

Several directions for further research arise naturally from the work presented. The first is the integration of WebAuthn [18] and passkey-based authentication as an alternative or complement to password-based authentication. WebAuthn offers stronger security guarantees by binding authentication credentials to a physical device and by eliminating the password as a shared secret subject to phishing, replay, and database compromise. The architectural integration of WebAuthn with the JWT-plus-refresh token model presented here is straightforward in principle, but raises practical questions concerning the management of multiple credentials per user, the fallback to password-based authentication when WebAuthn is unavailable, and the user experience around credential enrollment and recovery.

The second direction is the exploration of post-quantum signature schemes for token signing. Current JWT implementations rely on signature algorithms whose security depends on the hardness of integer factorization or discrete logarithms, both of which are vulnerable to attack by sufficiently large quantum computers. The transition to post-quantum signature schemes such as ML-DSA (formerly CRYSTALS-Dilithium) is anticipated to occur over the coming decade, and the JWT specifications will require extension to accommodate the larger signature sizes and verification times of these schemes. Investigation of the practical impact of post-quantum signatures on the JWT-plus-refresh token model—particularly with regard to token size and verification latency—will inform the migration of deployed systems.

The third direction is the development of formal analyses of refresh token rotation under realistic attacker models. The intuitive argument for rotation as an intrusion detection mechanism has been presented informally in the literature, but a rigorous treatment that quantifies the probability of detection as a function of the attacker's access pattern, the legitimate user's activity rhythm, and the rotation interval would provide stronger guidance for the configuration of rotation parameters. Such an analysis may also reveal scenarios in which rotation provides less protection than commonly assumed and motivate alternative approaches to refresh token compromise detection.

The fourth direction is the empirical study of authentication usability in the presence of token rotation. The architecture presented here is designed to be transparent to the user, with token rotation handled automatically by the HTTP client interceptor. However, edge cases—such as simultaneous use of the application on multiple devices, prolonged idle periods that span the refresh token lifetime, and revocation events triggered by token reuse—may produce user-visible interruptions whose frequency and impact have not been characterized in the present work. A longitudinal study of authentication events in a deployed application would clarify the practical user experience implications of the architecture.

8. Conclusions

This research has presented an authentication architecture for modern web applications combining short-lived JSON Web Tokens with rotating refresh tokens, integrated within a seven-layer defense-in-depth model aligned with the OWASP Application Security Verification Standard Level 1. The architecture addresses the principal categories of attacks against web-application authentication subsystems through complementary mechanisms whose individual failures do not compromise the overall security posture.

The core token mechanism limits the exposure of any compromised access token to a maximum of fifteen minutes through aggressive token expiration, and limits the exposure of refresh tokens through rotation that invalidates each token at the moment of use. The combination of httpOnly cookies, SameSite=Lax transmission, and rate-limited refresh endpoints protects against the principal client-side and network-based attack vectors without introducing dedicated anti-CSRF tokens or specialized infrastructure beyond a standard relational database.

Empirical measurements from a reference implementation in Next.js 16 and NestJS 11 confirm that the architecture sustains login latency below two hundred fifty milliseconds, refresh latency below fifteen milliseconds, and verification overhead below one millisecond per authenticated request. These measurements demonstrate that the layered defense imposes negligible operational cost relative to the security guarantees it provides, and they suggest that the architecture is suitable for production deployment in small-to-medium e-commerce systems, microservice-oriented backends, and other application classes where stateless verification is preferred to dedicated session storage.

The proposed architecture also serves as a reference for software engineering curricula and as a starting point for further refinement in directions identified in the present work, including WebAuthn integration, post-quantum signature migration, and formal analysis of rotation under adversarial models. The mature ecosystem of TypeScript libraries, the consistent type discipline between client and server, and the operational simplicity of the underlying frameworks together make this architecture a practical recommendation for new projects whose security requirements correspond to the OWASP ASVS Level 1 baseline.

References:

1) Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force. https://doi.org/10.17487/RFC7519

2) Hardt, D. (2024). State of JWT in modern web applications: A survey of practice and pitfalls. Journal of Web Engineering, 23(4), 481–512.

3) Mouratidis, H., & Argyropoulos, N. (2022). Security architectures for single-page applications: A comparative study. Computers & Security, 116, 102659. https://doi.org/10.1016/j.cose.2022.102659

4) Hardt, D. (Ed.). (2023). OAuth 2.0 Security Best Current Practice (RFC 9700). Internet Engineering Task Force. https://doi.org/10.17487/RFC9700

5) Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Signature (JWS) (RFC 7515). Internet Engineering Task Force. https://doi.org/10.17487/RFC7515

6) Jones, M. (2015). JSON Web Algorithms (JWA) (RFC 7518). Internet Engineering Task Force. https://doi.org/10.17487/RFC7518

7) Hardt, D. (Ed.). (2012). The OAuth 2.0 Authorization Framework (RFC 6749). Internet Engineering Task Force. https://doi.org/10.17487/RFC6749

8) Lodderstedt, T., Bradley, J., Labunets, A., & Fett, D. (2024). OAuth 2.0 Security Best Current Practice. Internet Engineering Task Force, draft-ietf-oauth-security-topics-26. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics

9) Mozilla Developer Network. (2024). HTTP cookies. Mozilla Foundation. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

10) Barth, A. (2011). HTTP State Management Mechanism (RFC 6265). Internet Engineering Task Force. https://doi.org/10.17487/RFC6265

11) West, M., & Goodwin, M. (2023). Cookies: HTTP State Management Mechanism (RFC 6265bis). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis

12) OWASP Foundation. (2024). Password Storage Cheat Sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

13) Provos, N., & Mazières, D. (1999). A future-adaptive password scheme. In Proceedings of the USENIX Annual Technical Conference (pp. 81–91). USENIX Association.

14) OWASP Foundation. (2024). Application Security Verification Standard 4.0.3. https://owasp.org/www-project-application-security-verification-standard/

15) OWASP Foundation. (2024). Cross-Site Scripting Prevention Cheat Sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

16) OWASP Foundation. (2023). Credential Stuffing Prevention Cheat Sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Credential_Stuffing_Prevention_Cheat_Sheet.html

17) Colin McDonnell. (2024). Zod: TypeScript-first schema validation with static type inference. https://zod.dev

18) Balfanz, D., Czeskis, A., Hodges, J., Jones, J. C., Jones, M. B., Kumar, A., Liao, A., Lindemann, R., & Lundberg, E. (2021). Web Authentication: An API for accessing Public Key Credentials, Level 2 (W3C Recommendation). World Wide Web Consortium. https://www.w3.org/TR/webauthn-2/