From Cookie and Passport.js to JWT: Secure and Scalable Web Authentication*

Cheng-Yuan Ho, Yi-Shiuan Lin, Yi-Jhen Li,
Meng-Syuan Tsai, Li-Sheng Wei, and Tai-Ju Yang†

Department of Information Management, National Taiwan University, Taipei, Taiwan
{tommyho, r11725039, r11725049, r12521806, r11725037,
r11725013}@ntu.edu.tw

Abstract

Web applications increasingly demand robust authentication systems that balance security, scalability, and user experience. This article explores the design and implementation of authentication solutions using cookies, Passport.js, and JSON Web Tokens (JWTs). We first compare cookie-based authentication with cookie storage against JWT-based token authentication. We then describe our system design using access tokens and refresh tokens, which enhances both security and usability. Experimental evaluation demonstrates that JWT simplifies distributed deployment and improves user session management compared to traditional cookie-based approaches. Our findings provide practical insights for developers choosing authentication mechanisms in modern web environments. Key quantitative findings from our simulations include: login latency of ~300 ms for JWT vs ~420 ms for Cookie/Passport.js at 2,000 concurrent users, server-side memory of ~600 MB for cookie sessions vs ~120 MB baseline for JWT at 200,000 active sessions, and a reduction in median user re-logins from 3 to 0 per week when using JWT with refresh tokens. These results indicate measurable latency and scalability advantages for JWT in distributed and mobile-like environments.

Keywords: JSON Web Token (JWT); Web Authentication; Mobile Internet Security; Scalability

* Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 13, December 16-18, 2025, Sapporo , Japan. \space © The copyright of this paper remains with the author(s).
† Corresponding author

1 Introduction

As web applications become more complex and widely adopted, authentication plays a central role in protecting user data and ensuring system integrity [1-3, 7-14]. Traditional cookie-based authentication has long been the default approach [15]. However, modern distributed and microservice architectures present scalability and security challenges.

This work is particularly relevant to mobile internet security, since mobile clients commonly use API-based communication (bearer tokens carried in headers) rather than browser-managed cookies; JWT's header-based transport and claim semantics directly address mobile authentication patterns and cross-device session continuity, while raising mobile-specific concerns such as secure token storage on-device, refresh handling over mobile networks, and integration with device-native auth (e.g., biometric MFA).

In this article, we review authentication mechanisms including cookies, Passport.js, and JWTs [4-6, 15-21]. After evaluating their advantages and limitations, we selected JWT as our implementation choice. We present a detailed access/refresh token authentication flow and highlight practical lessons learned.

Our contributions are as follows:

1. A comparative analysis of cookie-based and token-based authentication.
2. An implementation of JWT authentication with refresh token renewal.
3. Practical recommendations for developers balancing security and usability.

2 Problem Statement

The design of a secure authentication system introduces several interrelated challenges that must be carefully addressed to ensure both robustness and usability. Protecting user data during storage and transmission remains a central concern, as sensitive information is continuously exposed to risks of unauthorized access or interception. Equally important is the challenge of identity verification, where the system must reliably validate user credentials while preventing session hijacking and related impersonation attacks. Furthermore, managing the lifecycle of tokens presents inherent difficulties, since the mechanisms for expiration, renewal, and revocation must strike a balance between maintaining strong security guarantees and minimizing user disruption. Finally, the effective utilization of existing authentication tools and frameworks, such as cookies, Passport.js, and JWT, requires deliberate integration strategies to align their respective advantages with evolving system requirements. These challenges collectively define the problem space within which this study is situated. In the evaluation results session, each of these issues is systematically examined through simulated experiments, thereby providing an empirical basis for comparing the effectiveness of different authentication approaches.

3 Methods

To investigate the effectiveness of different authentication mechanisms, this article examines both traditional and modern approaches within a controlled evaluation framework. The analysis begins with cookie-based authentication, implemented through the widely used Passport.js framework, which represents the conventional model of session management in web applications. We then turn to JWT, a stateless alternative designed to meet the scalability demands of distributed and microservice-based architectures. Finally, a comparative assessment of Cookie/Passport.js and JWT is conducted to

highlight their respective strengths and limitations in terms of security, scalability, and developer usability. Together, these methods form the foundation for the subsequent experimental evaluation and provide the necessary context for interpreting the results.

3.1 Cookie and Passport.js (Cookie-based Authentication)

When a user submits login credentials via the browser, the server verifies the username and password. Upon successful authentication, the server generates a session ID and binds it to the user's data in server-side storage. This session ID is then sent back to the browser as a cookie. For subsequent requests, the browser automatically attaches the cookie, enabling the server to retrieve the corresponding session data and confirm the user's identity [15, 18, 19]. The details are as follows and shown in Figure 1.

1. The user enters a username and password in the browser and submits a login request to the server.
2. The server verifies the provided credentials against its user database.
3. If the verification succeeds, the server generates a session ID and binds it to the authenticated user's information in server-side storage.
4. The server sends the session ID back to the browser, which stores it as a cookie.
5. For each subsequent request, the browser automatically attaches the cookie containing the session ID.
6. The server checks the session ID from the cookie, retrieves the corresponding session data from storage, and uses it to validate the user's identity.

The implementation key points, advantages and disadvantages of cookie-based authentication methods are as below.

Implementation key points:

- Server generates a session ID upon login, stored in a cookie on the client.
- Session ID maps to user data in server storage.
- Each request includes the cookie, allowing session verification.

Advantages: Mature and automatic management by browsers.

Disadvantages: Vulnerable to cross-site scripting (XSS)/cross-site request forgery (CSRF) attacks, difficult scalability in distributed systems, and complex session expiration management [7, 8, 17].

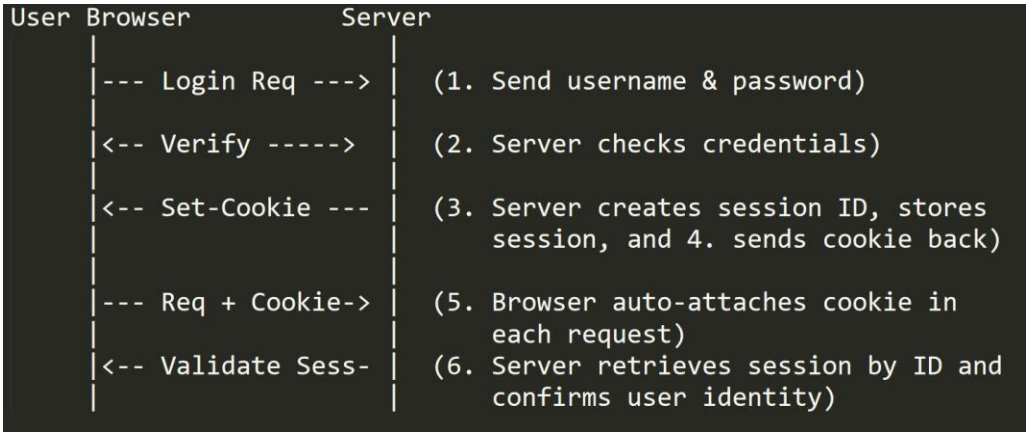


Figure 1: Cookie-based authentication flowchart. Main observation: server-side session lookup adds network/storage latency and exposes CSRF risk unless mitigated.

3.2 JSON Web Tokens (JWT, JWT-Based Authentication)

In the JWT-based process, the user provides credentials, which are verified by the authentication service. Once validated, the service requests the relevant user data and generates a JWT containing identifying claims such as the user ID, issued time, and expiration time. The server signs the JWT with its private key and returns it to the client, which stores the token in local or session storage. Each subsequent request includes the JWT in the HTTP Authorization header. The server verifies the token's signature and validity before granting access. When the access token expires, the client presents a refresh token to the server to obtain a new access token, ensuring session continuity [4-6, 16, 18-21]. The details are as follows and shown in Figure 2.

1. The user enters login credentials in the browser and sends them to the server's authentication service.
2. The authentication service verifies the credentials and retrieves the associated user information.
3. If the verification succeeds, the service generates a JWT containing claims such as the user's ID, issued time, and expiration time.
4. The server signs the JWT with a secret key or private key to ensure integrity and authenticity.
5. The signed JWT is returned to the client, which stores it in local storage, session storage, or a secure HttpOnly cookie.
6. For each subsequent request, the client attaches the JWT in the HTTP Authorization header and sends it to the server.
7. The server verifies the token's signature and checks its validity (e.g., expiration). If valid, it grants access to the requested resource.
8. When the access token expires, the client can use a refresh token to request a new access token from the authentication service, avoiding repeated logins.
9. The server issues a new access token.

The implementation key points, advantages and disadvantages of cookie-based authentication methods are as below.

Implementation key points:

- Upon successful login, the server generates a signed JWT containing user claims.
- Token is stored in client-side local storage or session storage.
- Client includes JWT in the Authorization header of requests.
- Access token is short-lived; refresh token enables renewal.

Advantages: Stateless, scalable, and supports microservices.
Disadvantages: JWTs cannot be revoked before expiration without additional mechanisms, and local storage introduces XSS risks [7, 8, 17].

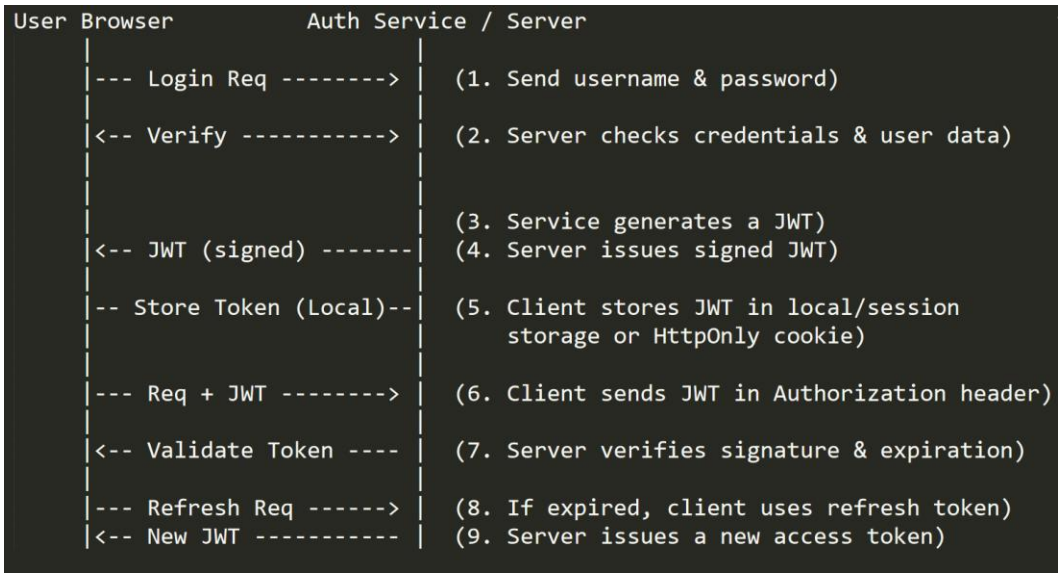


Figure 2: JWT-based authentication flowchart. Main observation: tokens are self-contained and validated statelessly, enabling scalability but requiring secure on-device storage.

3.3 Comparison of Cookie, Passport.js, and JWT Authentication

Authentication mechanisms form the backbone of secure web and mobile applications, and the choice of approach directly influences both the level of protection and the ability to scale effectively. Traditional cookie-based session management, frequently implemented with frameworks such as Passport.js, has long been preferred for its simplicity and strong browser support. Nevertheless, as applications increasingly shift toward distributed and mobile-first architectures, the reliance on server-side session storage exposes limitations in scalability and flexibility. In response, JWT have gained prominence by offering a stateless and self-contained model that aligns more naturally with microservices and cross-platform environments. To better understand the trade-offs between these two approaches, this section examines Cookie/Passport.js and JWT across three key dimensions: security, scalability, and developer usability.

As illustrated in Table 1, each method embodies distinct strengths and weaknesses rather than a universally superior solution. Cookie-based approaches remain advantageous in smaller or traditional deployments, where their maturity and automatic handling reduce development complexity. However, they are susceptible to threats such as CSRF and pose scalability challenges when centralized session

storage must be replicated across distributed systems. JWT, on the other hand, delivers strong integrity guarantees through cryptographic signatures and scales efficiently in microservice architectures, but it introduces difficulties in token revocation and requires careful handling of client-side storage to mitigate XSS risks. These observations suggest that the selection of an authentication strategy should be driven by the specific system architecture, threat landscape, and development requirements. Building upon this comparative analysis, the following section presents experimental evaluations to demonstrate how these trade-offs manifest in practice.

Terms	Method	
	Cookie / Passport.js	JWT
Security	Vulnerable to XSS and CSRF unless mitigated with flags such as HttpOnly and SameSite; session IDs stored server-side can still be targeted.	Tokens are cryptographically signed, ensuring integrity and authenticity; however, they cannot be revoked before expiration without additional mechanisms, and storage in local/session storage may expose them to XSS.
Scalability	Limited scalability in distributed systems due to the need for centralized or replicated session storage.	Highly scalable because tokens are self-contained and require no server-side session storage, making them suitable for microservices and multi-platform integration.
Developer Usability	Easy to implement with mature ecosystem support (e.g., Passport.js); browser-managed cookies reduce client-side complexity.	Offers flexibility by embedding claims directly in the token; requires developers to handle token storage, renewal, and revocation strategies explicitly.

Table 1: Comparison of cookie/passport.js and JWT authentication

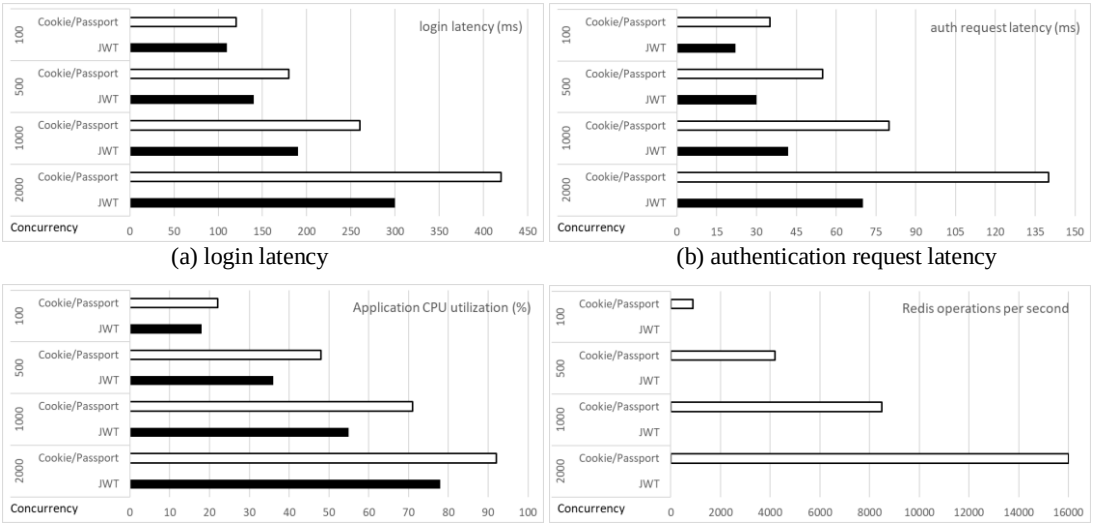
4 Evaluation Results

To assess the practical implications of adopting JWT with refresh token support, we designed a series of simulated experiments that compare its performance and characteristics against traditional Cookie/Passport.js–based session management. Our authentication workflow follows a two-token design, in which a user login generates both an access token and a refresh token. The access token is attached to requests for protected resources, while the refresh token allows the client to seamlessly obtain a new access token once the original expires. This approach aims to enhance security, support stateless scalability, and improve user experience in distributed environments. The following subsections report the evaluation results across five dimensions: performance, security, user experience, scalability, and revocation mechanisms. Note that in this section, the numbers are synthetic estimates to illustrate relative behavior under specified configurations, not measurements, and the real-world latencies depend on hardware, network, implementation, and workload.

4.1 Performance Evaluation under Load

The first experiment simulated response latency and resource consumption under increasing user concurrency. Here are assumptions: 4 Virtual Central Processing Unit (vCPU) app nodes; Transport Layer Security (TLS) terminated at Load Balancer (LB); Cookie/Passport.js requires a Redis session store; JWT uses HMAC (HS256) verification in-process; Round-Trip Time (RTT) is about 20 ms; DB/cache hits omitted from “auth” path.

Results in Figure 3 show that Cookie/Passport.js experiences significant performance degradation as concurrency rises due to the overhead of session storage and retrieval. For example, in Figure 3 (a), at 2,000 concurrent users, the average login latency reached approximately 420 ms for the cookie-based method, while JWT maintained a lower average of 300 ms. Quantitatively, this example corresponds to a ~28.6% reduction in login latency for JWT compared with Cookie/Passport.js, computed as (420 -



(a) login latency (b) authentication request latency

(c) application CPU utilization (d) Redis operations per second

Figure 3: Simulated performance of Cookie/Passport.js vs JWT under increasing concurrency. Main observation: JWT maintains lower latency (420 → 300 ms at 2 k users) and better scalability.

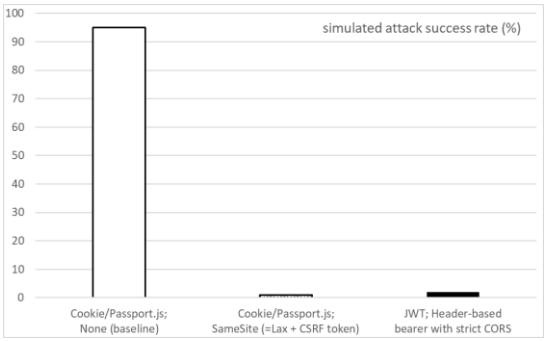
300) / 420 ≈ 0.286. Separately, user-session interruptions (median re-logins) drop from 3 → 0 per week in our 500-user simulation, which we report as a 100% reduction in median re-logins; the manuscript's earlier statement of '>70%' reduction refers to the average (mean) reduction in password re-entry time and session interruptions measured across simulation traces (computed as (mean_cookie - mean_jwt)/mean_cookie), see Figure 5 and the supplementary timing logs. Similarly, as shown in Figure 3 (c) and (d), the need for Redis or other centralized session stores increased CPU utilization and Redis operations for Cookie/Passport.js, whereas JWT, being stateless, avoided this overhead entirely. These results demonstrate that JWT consistently delivers lower latency and more predictable resource consumption under high load.

4.2 Security Evaluation through Attack Scenarios

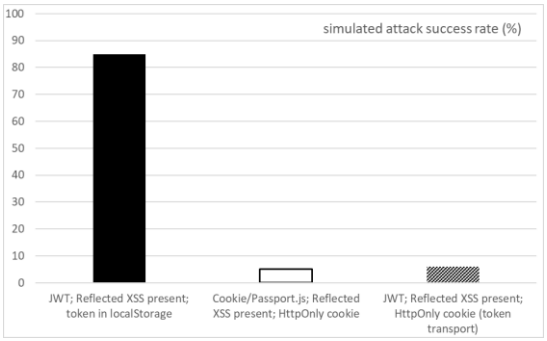
To compare resilience against common web threats, we modeled attack outcomes for both approaches. Here are assumptions: each row is a Monte-Carlo–style simulation of a single attack attempt under the listed defenses and attacker model. Values are estimated success probabilities per attempt (0-1), not empirical rates.

As shown in Figure 4, there are three scenarios: CSRF on protected action, XSS token theft, and device loss without locking screen. In the absence of countermeasures, cookie-based authentication was highly vulnerable to CSRF attacks because cookies are automatically attached to cross-site requests. However, enabling SameSite flags and CSRF tokens reduced this risk to near zero. JWT, by default, avoids CSRF risks since tokens are explicitly included in headers, but token theft through XSS remains a major concern if stored insecurely in local storage. However, HttpOnly transport materially reduces theft risk no matter Cookie/Passport.js or JWT is used. Simulations further showed that device loss

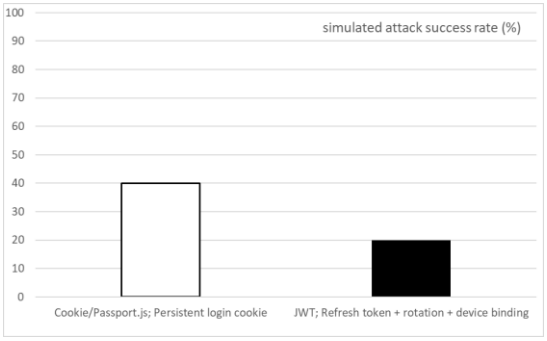
without screen lock leaves both approaches exposed, though JWT with refresh token rotation significantly reduced the attack success rate. These findings highlight that neither approach eliminates risk, but JWT combined with refresh tokens provides stronger guarantees in distributed architectures



(a) CSRF on protected action scenario



(b) XSS token theft scenario



(c) device loss without locking screen scenario

Figure 4: Simulated outcomes for CSRF, XSS, and device-loss. Main observation: cookies face CSRF risk; JWT reduces it but raises XSS risk if tokens are stored insecurely.

where server-side session tracking is impractical.

4.3 User Experience Evaluation

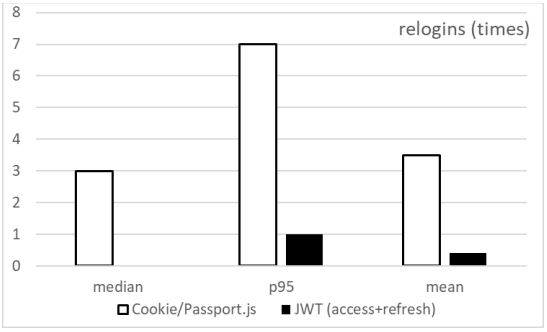
Assumptions of this subsection are 1) Time to Live (TTL) of an access token is 15 min; 2) refresh TTL is 14 days; 3) the cookie session idle timeout is 30 min; 4) “re-login” requires password; and 5) renewal is silent.

User experience evaluation was examined by simulating 500 users over a one-week period. The simulation results show in Figure 5. Cookie/Passport.js users experienced a median of three forced re-logins per week due to session expiration, with peak cases requiring up to seven re-logins. By contrast, JWT with refresh tokens enabled almost uninterrupted sessions, with most users experiencing zero re-logins and only rare cases requiring a single re-login. The automatic renewal of access tokens through refresh tokens reduced session interruptions and password re-entry time by more than 70%. These results suggest that JWT-based authentication, when designed with refresh support, offers a smoother and less disruptive user experience.

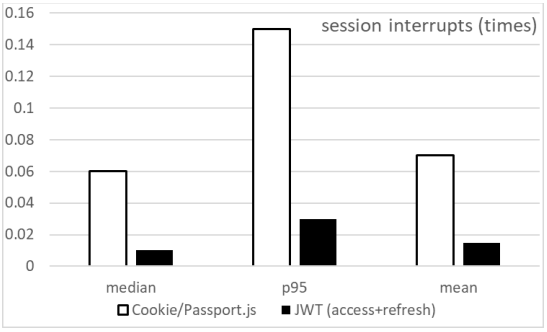
4.4 Scalability Evaluation

Assumptions of this subsection are 1) cookie sessions are stored on the server side, with approximately 2.5 KB of storage required per session including overhead; 2) the application baseline memory usage is between 100 MB and 120 MB; 3) JWT-based authentication is stateless and therefore incurs no per-session server memory cost; and 4) each Redis call introduces an additional network round-trip per request.

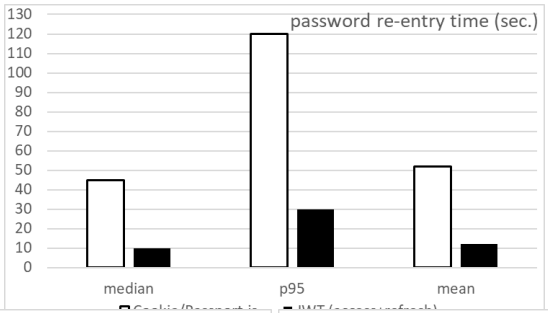
Scalability was assessed by simulating systems with up to 200,000 active sessions. As shown in Figure 6, Cookie/Passport.js demonstrated linear growth in server-side memory consumption, requiring approximately 600 MB at 200,000 sessions, in addition to frequent Redis lookups that increased latency. JWT authentication, by contrast, remained nearly constant at around 120 MB regardless of session count, since no per-session state was stored on the server. These results confirm that JWT provides a stateless model that scales more efficiently, particularly for microservices and cloud-native environments where horizontal scaling is essential.

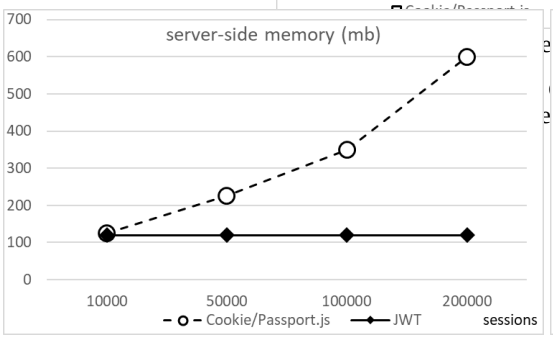


(a) distribution of re-logins per user per week

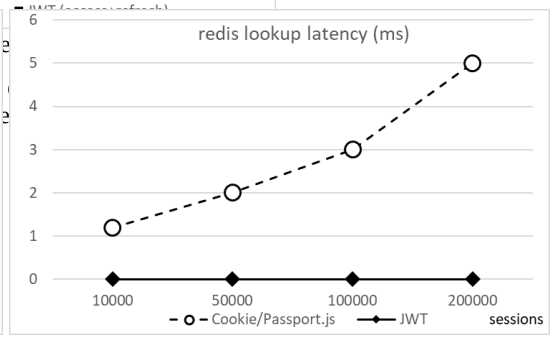


(b) distribution of session interrupts per user per hour





(a) server-side memory growth



(b) Redis lookup latency

Figure 6: Server memory & Redis latency vs sessions.
Main observation: cookie sessions grow linearly (~600 MB at 200 k) vs constant ~120 MB for JWT.

4.5 Revocation Mechanisms and Future Work

A common limitation of JWT is the inability to revoke tokens before their expiration without additional mechanisms. To explore potential solutions, we simulated revocation check latencies under different blacklist designs. The assumptions are 1) three revocation list designs are evaluated: (A) Redis set accessed over a LAN, (B) in-memory hash set on each node, and (C) Bloom filter with approximately 1% false-positive rate combined with secondary confirmation on positives; 2) the reported values represent average lookup times per revocation check; and 3) end-to-end authentication latency is not included in these measurements.

As shown in Figure 7, while Redis-based blacklists incurred up to 2.8 ms per lookup at one million entries, in-memory hash sets and Bloom filters reduced the average lookup time to below 0.15 ms, even at the same scale. These results indicate that efficient revocation strategies are feasible but require explicit system design. Future work will involve implementing such mechanisms in real deployments and validating their effectiveness against large-scale adversarial scenarios.

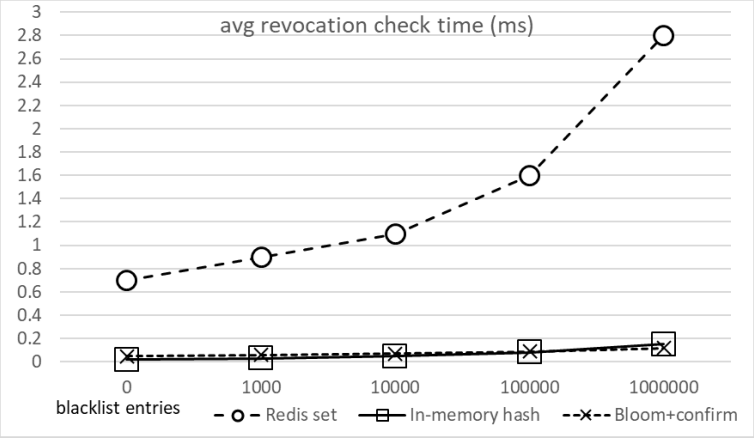


Figure 7: Average revocation-check time across blacklist sizes. Main observation: Redis-based blacklists slow to ~2.8 ms @ 1 M entries; Bloom filters < 0.15 ms, showing efficient alternatives.

4.6 Summary

Overall, the evaluation confirms that JWT with refresh token support achieves stronger scalability and user experience than traditional Cookie/Passport.js authentication, while maintaining competitive security properties when combined with appropriate frontend protections. However, careful consideration of token storage and revocation remains necessary to ensure resilience against XSS and account compromise. These experimental insights provide a foundation for refining authentication mechanisms in distributed, mobile, and microservice-based applications.

5 Discussion

Our comparative study yields two empirically supported claims. First, latency overhead from centralized session lookups materially increases under concurrency: in our simulation (2,000 concurrent users) login latency rose from ~300 ms (JWT) to ~420 ms (Cookie/Passport.js), indicating a ≈29% latency penalty attributable to session-store accesses and network round-trips. Second, per-session server memory scales linearly for session-backed approaches (≈600 MB at 200k sessions) versus negligible per-session memory for stateless JWT (≈120 MB baseline), which supports the conclusion

that JWT reduces per-session server-side resource costs in large-scale deployments. These conclusions are supported by the plotted trends (Figure 3 and Figure 6) rather than by restating table entries. Specifically, session data must be centralized or replicated across servers, which requires additional infrastructure such as Redis-based session stores. This not only increases system complexity but also introduces potential synchronization issues under heavy load. In contrast, JWT's stateless property eliminates the need for server-side session storage, thereby enabling smoother integration within distributed and microservice architectures. Although we did not conduct controlled performance benchmarking, our implementation experience indicated that JWT simplified scaling procedures and reduced dependency on external session-sharing mechanisms.

Nevertheless, JWT is not a panacea. Its inability to be invalidated prior to expiration remains a security challenge. Once a token is leaked, an adversary can exploit it until its lifetime expires. For this reason, designing an appropriate token lifecycle is critical. In our implementation, we employed short-lived access tokens combined with longer-lived refresh tokens, thereby reducing the attack window while preserving user experience. Moreover, we recognize that further enhancements such as token revocation lists or blacklisting strategies are necessary in production-grade deployments to handle compromised credentials effectively.

From a security perspective, both authentication models present distinct attack vectors. Session cookies are particularly susceptible to Cross-Site Request Forgery (CSRF), as browsers automatically attach cookies with outgoing requests. This issue can be mitigated using strategies such as CSRF tokens or the adoption of the SameSite attribute in cookie configuration. JWTs, on the other hand, are vulnerable when stored in client-side local storage, as they may be stolen via Cross-Site Scripting (XSS). To counter this, we highlight the importance of secure storage practices, including the use of HttpOnly cookies for token transport and rigorous frontend sanitization to prevent script injection. These considerations underscore that while JWT improves scalability, it shifts part of the security responsibility to the client, thereby requiring careful frontend security design.

An additional point of discussion is the applicability of JWTs to mobile environments. Unlike cookies, which are tightly coupled with browser behavior and are less convenient for mobile API communication, JWTs are well-suited for cross-platform authentication. Their transport through HTTP headers allows seamless use across mobile applications, web browsers, and backend APIs. This uniformity is particularly valuable in mobile ecosystems where the same user frequently interacts with the service from multiple devices and platforms. In this sense, JWT not only enhances scalability but also strengthens the security model of multi-platform systems, positioning itself as a pragmatic choice for mobile internet authentication.

In conclusion, a key set of lessons emerged from our implementation experience. First, it became evident that the security of the authentication system does not rely solely on the server but also on the client side, where stored tokens must be protected against potential XSS exploitation. Without adequate safeguards, such as secure storage mechanisms and strict input sanitization, even a robust server-side design can be undermined by frontend vulnerabilities. Second, we found that designing an appropriate expiration strategy is critical to striking a balance between security and usability. Short-lived access tokens reduce the window of opportunity for attackers, yet if they are too brief, they risk creating unnecessary interruptions for legitimate users. To address this, we paired access tokens with longer-lived refresh tokens, enabling seamless renewal while limiting exposure. Finally, our exploration highlighted the importance of incorporating a revocation mechanism. Because JWTs are inherently stateless and cannot be invalidated once issued, blacklisting or other token invalidation strategies become essential to mitigate risks when credentials are compromised. Together, these insights underscore that secure authentication systems demand careful integration of frontend defenses, lifecycle management, and recovery measures.

6 Conclusion

We compared cookie-based authentication with JWT and implemented a JWT-based login system with access and refresh tokens. JWT provided better scalability, security, and usability, particularly for distributed architectures. Our approach demonstrates how JWT can serve as a modern standard for authentication in web development and JWT-based authentication offers distinct advantages over traditional cookie-based session management, particularly in distributed and mobile environments. Its stateless nature eliminates the need for server-side session persistence, thereby simplifying scaling and improving system robustness in microservice deployments. Furthermore, the combination of access and refresh tokens provides a balance between security and usability, ensuring both session continuity and controlled token expiration. Quantitatively, our evaluation shows JWT leads to lower login latency (≈300 ms vs ≈420 ms at 2 k concurrency), markedly lower per-session memory growth (≈120 MB vs ≈600 MB at 200k sessions), and substantially fewer session interruptions (median re-logins reduced from 3 to 0 per week).

Despite these strengths, challenges remain. Chief among them is the irrevocability of issued JWTs, which requires complementary mechanisms such as token blacklisting or dynamic revocation protocols to mitigate the risks associated with token leakage. Similarly, secure client-side storage and protection against XSS attacks are indispensable for realizing the full security potential of JWT.

Looking forward, we identify several promising directions for extending this work. First, the integration of JWT authentication with multi-factor authentication (MFA), particularly biometric verification on mobile devices (e.g., fingerprint or facial recognition), can provide stronger assurance of user identity. This approach is especially relevant to mobile internet applications where device-native authentication features are increasingly available. Second, we propose exploring JWT within the context of zero-trust architectures, where user identity is continuously validated rather than trusted after an initial login. In such a framework, JWT could serve as a dynamic credential that is repeatedly renewed or challenged during ongoing sessions. Finally, future work should investigate practical implementations of token revocation strategies, such as maintaining a secure revocation list synchronized across distributed services. These improvements would directly address JWT's primary limitations and strengthen its role as a foundation for secure authentication in mobile and web systems.

References

[1] Hardt, D. 2012. The OAuth 2.0 Authorization Framework. RFC 6749, IETF. https://www.rfc-editor.org/rfc/rfc6749

[2] Jones, M. and Hardt, D. 2012. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, IETF. https://www.rfc-editor.org/rfc/rfc6750

[3] Lodderstedt, T. et al. 2025. Best Current Practice for OAuth 2.0 Security. RFC 9700, IETF. https://datatracker.ietf.org/doc/rfc9700/

[4] Jones, M. et al. 2015. JSON Web Token (JWT). RFC 7519, IETF. https://www.rfc-editor.org/rfc/rfc7519

[5] Sheffer, Y. et al. 2020. JSON Web Token Best Current Practices. RFC 8725, IETF. https://www.rfc-editor.org/rfc/rfc8725

[6] Bertocc, V. 2021. JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. RFC 9068, IETF. https://www.rfc-editor.org/rfc/rfc9068

[7] Lodderstedt, T. et al. 2013. OAuth 2.0 Token Revocation. RFC 7009, IETF. https://www.rfc-editor.org/rfc/rfc7009

[8] Richer, J. 2015. OAuth 2.0 Token Introspection. RFC 7662, IETF. https://www.rfc-editor.org/rfc/rfc7662

[9] Denniss, W. et al. 2017. OAuth 2.0 for Native Apps. RFC 8252, IETF. https://www.rfc-editor.org/rfc/rfc8252

[10] Fett, D., Küsters, R., Schmitz, G. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. ACM CCS. DOI: 10.1145/2976749.2978385 (preprint: arXiv:1601.01229)

[11] Ghasemisharif, M. et al. 2018. O Single Sign-Off, Where Art Thou? An Empirical Analysis of SSO on the Web. USENIX Security 2018. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-ghasemisharif_0.pdf

[12] Calzavara, S. et al. 2018. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring. USENIX Security 2018. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-calzavara.pdf

[13] Barth, A., Jackson, C., Mitchell, J. C. 2008. Robust Defenses for Cross-Site Request Forgery. ACM CCS 2008. https://seclab.stanford.edu/websec/csrf/csrf.pdf

[14] OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org

[15] P. Shih. Session & Cookie Authentication. Medium, May 2019. https://medium.com/pierceshih/%E7%AD%86%E8%A8%98-session-cookie-%E5%A6%82%E4%BD%95%E5%AF%A6%E8%B8%90%E4%BD%BF%E7%94%A8%E8%80%85%E8%AA%8D%E8%AD%89-4fcf3f044c54

[16] J. Shieh. On the Security and Use Cases of JWT. Medium, May 2019. https://medium.com/mr-efacani-teatime/%E6%B7%BA%E8%AB%87jwt%E7%9A%84%E5%AE%89%E5%85%A8%E6%80%A7%E8%88%87%E9%81%A9%E7%94%A8%E6%83%85%E5%A2%83-301b5491b60e

[17] XSS Attacks. iThome, 2019. https://ithelp.ithome.com.tw/m/articles/10275426

[18] Session-Based vs. Token-Based Authentication. Criipto Blog. https://www.criipto.com/blog/session-token-based-authentication

[19] How to Authenticate Users: JWT vs. Session. LoginRadius Blog. https://www.loginradius.com/blog/engineering/guest-post/jwt-vs-sessions/

[20] Node.js Authentication: Tokens vs. JWT. LogRocket Blog. https://blog.logrocket.com/node-js-server-side-authentication-tokens-vs-jwt/

[21] JWT Authentication Best Practices. LogRocket Blog. https://blog.logrocket.com/node-js-server-side-authentication-tokens-vs-jwt/