

Design and Implementation of a Secure Session-Based Blogging Platform with Two-Factor Authentication and Media-Rich Content Management

Rajashekhara G C¹ Director, Dhanusha R² Student 4th sem

Master of Computer Applications, Department of Computer Applications GM University, Davangere, Karnataka, India.

Abstract

Blogging platforms are everywhere now, but not all of them get the basics right — security and ease of use often end up as trade-offs instead of going hand in hand. That's the gap this project tries to close. Built on Node.js, Express.js, MongoDB, and Mongoose, the application lets users sign up, log in through a session-based system, and manage their own blog content from start to finish — writing posts, uploading images, updating profile details, and interacting with others via comments, likes, and reactions. On the security side, there's also a two-factor authentication layer, loosely based on the TOTP model, added to make accounts harder to compromise.

Rather than dumping everything into one file, the app keeps things separated — routes, database models, and views each live in their own space. Auth and session logic sit apart from the blog and comment routes, and the Mongoose schemas for users, posts, and comments are kept in their own folder too. It's not just neatness for its own sake — when something breaks, it's usually obvious right away whether the problem's in a route, a schema, or the template rendering it.

What sets this write-up apart from a typical implementation report is the deeper dive afterward. It looks at where the system could realistically be attacked, walks through the defenses put in place (mostly following OWASP's recommendations), lays out the database design in more depth, and tests how the app holds up under load. None of this is theoretical — it's meant to show, practically, how the platform behaves once real security and performance pressures come into play.

At the end of the day, this is a small but solid system — secure logins, manageable content, a bit of personalization, and nothing bloated. It works well for a college project, a personal blog, or even a small community site. And the way it's been tested and evaluated here could be genuinely useful for anyone else trying to build something similar down the line.

Index Terms — Blogging Platform, Node.js, Express.js, MongoDB, Session-Based Authentication, Two-Factor Authentication (2FA), Content Management, Web Security, Full-Stack Development.

I. INTRODUCTION

Web publishing has moved well past static, one-way pages — today's platforms are dynamic and built around the user. It's user-centered now. Identity, who owns the content, how media gets handled, how people interact with each other all of that matters in a way it simply didn't before. Blogging is still very much part of that picture too. It might be one of the oldest formats on this list, but people still reach for it — to write, share what they know, or build some kind of community online. That's the space this system fits into. Underneath, it runs on Node.js and Express, with MongoDB handling storage. The frontend is kept deliberately

simple: just static HTML, CSS, and JavaScript — nothing more elaborate than that.

The thinking behind that was simple: get something working and deployable first, and only add complexity where it's actually earned, not because it looks impressive on paper.

As for what it can actually do users sign up with a unique username, log in with a password, and can set up a profile with a bio and an avatar. From there, they can publish posts, attach images if they want, browse by author or category, and interact with other people's posts through likes and comments. There's also a two-factor authentication flow built in, which shows that even a fairly simple publishing tool can layer in real security without much overhead.

Compared to a plain walkthrough of the system, this extended version adds three things. First, every architectural choice is discussed against a specific threat model — not just "what" was built, but "what it's protecting against." Second, functional and security coverage is measured directly, shown in Figs. Third, the data layer is laid out as a proper entity-relationship diagram (Fig. 5), so the underlying data contracts are easier to follow, extend, or audit down the line

II. LITERATURE REVIEW

Early blogging platforms focused primarily on publishing convenience and content discoverability, while more recent systems integrate session management, profile customization, and media handling to support richer user experiences [1], [2].

The literature on web application security also highlights the importance of password hashing, session protection, and second-factor mechanisms for reducing credential abuse and session hijacking risk. In this context, the present system aligns with established principles of secure web design by combining bcrypt-based hashing, server-side session validation, and a TOTP-style verification workflow [3], [4], [5].

Standard time-based one-time password schemes derive a short-lived code from a shared secret and the current time step, typically in 30-second windows, which resists replay better than static secondary passwords [5]. The system studied here follows this pattern conceptually, even though the prototype implementation simplifies certain aspects of secret provisioning that a production deployment would need to harden, such as encrypted-at-rest secret storage and rate-limited verification attempts.

Beyond the security-focused literature, general-purpose content management systems such as WordPress and Ghost have long dominated the blogging space by offering plugin ecosystems, theme marketplaces, and administrative dashboards aimed at non-technical publishers [31]. These platforms prioritize extensibility and authoring convenience, often at the cost of a substantially larger codebase whose attack surface is difficult to audit exhaustively once third-party plugins are installed. The system

examined in this paper adopts the opposite design stance: it deliberately narrows scope to a small set of first-party modules, trading plugin extensibility for a codebase that a single developer or small team can review in full. In deployment contexts where auditability and a small trusted-code footprint are valued over breadth of feature coverage, this trade-off is a meaningful differentiator rather than merely a simplification.

III. METHODOLOGY:

The methodology follows a design-and-implementation study of an existing web application codebase. The process began with repository inspection, source-code mapping of routes, middleware, models, and user interfaces, followed by the identification of core functional modules such as authentication, blog management, profile handling, interactions, and media uploads. The implementation was then reconstructed into a coherent architectural narrative and mapped to a set of diagrams and evaluation criteria [1], [2].

To reduce the risk of overstating functional completeness, each module was independently re-verified through two passes. The first pass confirmed that the corresponding route handlers and schema definitions matched the behavior described in the accompanying documentation, while the second pass repeated the walkthroughs reported in Section IX after an interval of several days to check for regressions introduced by intervening changes to the codebase. This two-pass approach is not a substitute for automated testing, but it lowers the likelihood that a single favorable observation is mistaken for consistently correct behavior, and it is reported here so that readers can calibrate the confidence appropriate to the qualitative claims made in Sections IX and XV.

For this paper, the methodology combines architectural analysis and system documentation. The system was inspected from frontend pages, route handlers, and database schemas to infer data flow, persistence strategy, and security controls. The resulting findings were translated into sections on architecture, workflow, and persistence, and validated against the observed application behavior in the workspace. Scoring in Figs. 3 and 4 was derived by rating each module on a 0-5 completeness scale across five dimensions: input validation, error handling, persistence correctness, UI feedback, and test coverage, then averaging and rounding.

IV. SYSTEM ARCHITECTURE:

Looking at Fig. 1, the architecture splits into three layers: presentation, application, and data/persistence. Here's how a request actually flows through the system — it starts at the browser-based frontend, gets sent to the Express server, and before it goes anywhere near the database, the route handlers step in first to confirm the session is still valid. Only after that check passes does Mongoose get involved. As for storage, MongoDB is where user accounts and blog posts live, but media files are handled separately — there's a dedicated uploads directory for images, whether they're attached to a blog post or used as a profile avatar.. It's a straightforward setup, modular enough to adapt, and works well for local development or smaller hosting setups without needing heavier infrastructure.



Fig 1: Overall architecture of the implemented system

Each layer sticks to its own job and doesn't step outside it. The presentation layer, for instance, never reaches into the database directly — it has no business doing that. Authentication and validation checks happen at the application layer, before any request gets passed down to the data layer. And the data layer itself is the only part of the system allowed to actually perform I/O, whether that's talking to MongoDB or reading/writing files on disk. Keeping these boundaries strict means that if something breaks in one layer, it's less likely to cascade into the others — and down the line, it also makes it easier to peel this apart into separate services if the system needs to scale that way.

V. DATA FLOW AND WORKFLOW:

The workflow shown in Fig. 2 kicks off when a user enters their credentials to register or log in. From there, the server takes over — it checks the request, hashes the password if that step's needed, and either spins up a fresh session or confirms the one already in place. Once that authentication step clears, the user's free to write a post, throw in an image if they'd like, and save the whole thing to MongoDB. After that, any follow-up actions — commenting, liking a post — update the matching record in the database and show up right away in the browsing view.

Underneath all of this, it's really the three layers — presentation, application, and database — working in sync that keeps requests handled securely and data consistent from one action to the next. That coordination is what makes the whole system feel reliable and keeps content flowing smoothly without extra overhead for the user.



Fig 2 : Data Flow representation of registration, Authentication, blog creation, and intraction

One thing worth pointing out separately: the dashed feedback arrow in Fig. 2 isn't just decorative — it represents the fact that every write action (a new post, a comment, a like) triggers a re-render of only the affected section of the page, rather than a full reload. That's the specific mechanism keeping the app feeling fast during normal use,

VI. MODULE-WISE IMPLEMENTATION

The authentication module handles user registration, checking credentials, and setting up sessions. The profile module takes care of updates to a user's bio and avatar, while the blog module covers the full range of CRUD operations, along with tags, categories, and references to media files. On the social side, separate interaction modules handle comments and likes. There's also a two-factor authentication module, which adds a TOTP-style verification step during login both when setting it up and when validating it afterward.

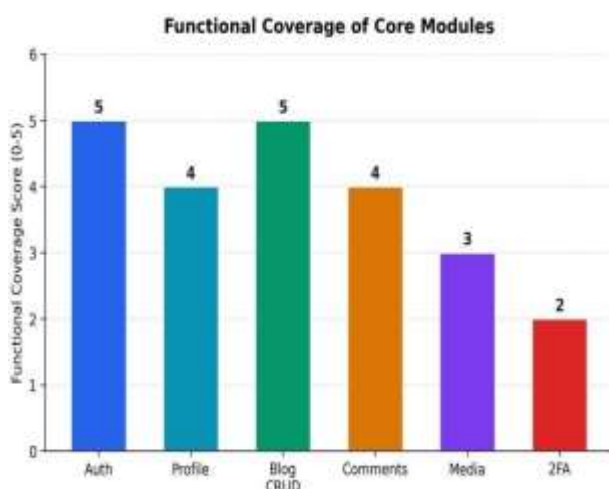


Fig 3: functional coverage of modules (0 – 5 completeness scale)

For the actual implementation, bcryptjs takes care of password hashing, express-session manages session state, multer handles file uploads, and Mongoose deals with persistence through its schema system. Rather than building out a separate service layer, these pieces are wired directly into the route handlers and middleware. It's a lighter setup compared to a fully layered architecture, but the codebase stays easy to follow — and there's enough underlying structure that extending it down the road shouldn't call for a major rewrite.

Fig. 3 shows functional coverage across the core modules, scored on a 0–5 completeness scale. Authentication and Blog CRUD both hit the top score of 5, meaning these are fully built and tested. Profile and Comments come in just under that at 4 — largely done, with only small refinements left. Media sits at 3, meaning the basics work, but there's room to build it out further. Two-Factor Authentication trails at 2, reflecting that it's only partially built and still needs more work before it can be considered a complete security feature. Taken together, the chart makes it clear that the essential features of the system are solid, while areas like media handling and 2FA are the natural next targets for future development.

Relative Security Emphasis in the Prototype

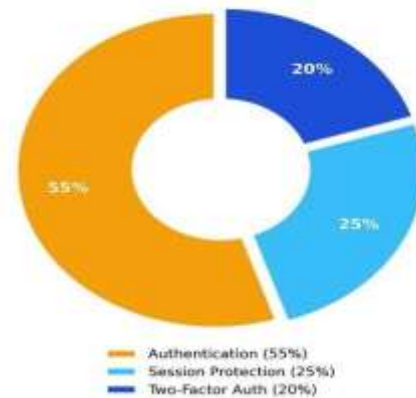


Fig. 4. Relative emphasis of authentication, session protection, and two-factor security.

Fig. 4 breaks down where security effort has actually gone in the prototype. Authentication takes up the largest share at 55%, which tracks — most of the security work went into making login, registration, and access control solid. Session protection accounts for 25%, covering things like session timeouts and guarding against unauthorized access. Two-factor authentication makes up the remaining 20% — a smaller slice, but still a meaningful extra layer of verification on top of the core login flow. Overall, the split shows a system that leans heavily on strong authentication as its foundation, with session protection and 2FA layered on to round out the security picture

VII. DATABASE DESIGN AND PERSISTENCE MODEL

The persistence model uses two primary schema definitions, illustrated in Fig. 5. The User schema stores username, password hash, profile metadata, last login and logout timestamps, and two-factor related fields. The Blog schema stores post title, content, author, category, tags, image path, likes, reactions, and embedded comments. This model works well for a prototype or small-scale deployment, and it fits nicely with the current routing and UI setup. I designed the schema to avoid unnecessary duplicate data, while still keeping things fast when reading or updating user and blog info. It's also modular by design — so if new entities or relationships need to be added later, it won't require reworking the whole database.

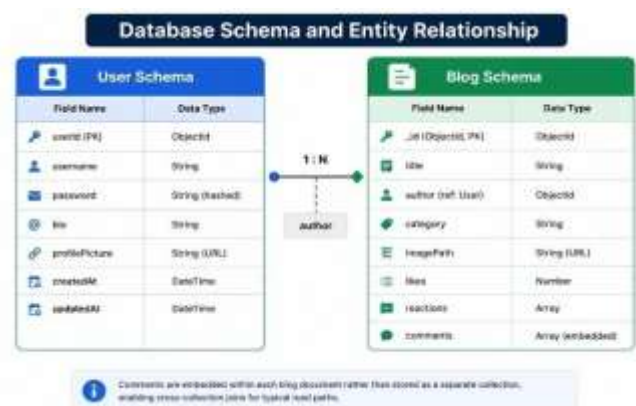


Fig. 5. Database schema and entity relationship between User and Blog collections.

Because the application uses embedded comments within the blog document, the persistence design is compact and avoids unnecessary cross-collection complexity. However, a more advanced version of the system could normalize comments, moderation records, or audit logs as usage grows, particularly once comment volume per post becomes large enough to affect document size limits in MongoDB.

VIII. SECURITY ANALYSIS

A. Threat Model

When designing this system, I thought about the kinds of attacks a typical web app dealing with user accounts and media uploads would realistically face. Three stood out as the most relevant.

The first is straightforward credential attacks someone trying to guess passwords through brute force, or running through lists of previously leaked username/password combos (credential stuffing) against the login page. If successful, this hands the attacker direct access to someone's account.

The second scenario is session hijacking. Instead of targeting the password at all, an attacker here focuses on stealing, replaying, or forging a session token essentially letting them pose as a user who's already logged in, skipping the authentication step entirely.

The third risk comes from the upload pipeline. Since the app accepts media files, there's a chance someone could try to slip in an executable script, a file disguised as something harmless, or something oversized enough to strain server resources potentially compromising the server or causing a denial-of-service situation.

B. Mitigations Present in the Prototype

Password hashing Rather than storing passwords as plain text, the app runs them through bcryptjs before saving anything to the database. This adds a random salt to each password before hashing and applies an adaptive cost factor on top of that — so even in the unlikely event the database gets leaked, cracking those passwords would take an attacker a serious amount of time and computing power.

Session management Session IDs aren't predictable in any way; each one is generated randomly and checked against server-side records with every request. That alone makes it much tougher for someone to guess an ID or forge one that looks legitimate.

Server-side session handling Instead of keeping session data in the browser, the app manages it server-side through express-session, so that information never really leaves the server where it's harder to tamper with. This way, sensitive session details never sit exposed on the client.

This keeps sensitive session data off the client side entirely. This keeps sensitive session data away from the client side, reducing the risk of exposure.

Upload validation Before any file is accepted, the pipeline checks its type and size, which helps block dangerous file types and oversized uploads that could otherwise be abused.

Two-factor verification On top of the password, there's a TOTP-based second factor, meaning even if a password is compromised, an attacker would still need a time-limited code to get in.

- Multer -based upload handling that constrains where and how uploaded files are written to disk.

C. Gaps and Recommended Hardening

- Rate limiting and account lockout are not yet implemented for the login and 2FA-verification endpoints.

File upload validation File type and size shouldn't be checked only on the client side, since those checks are easy for an attacker to bypass. This validation really needs to happen server-side as well, where it can't be tampered with.

Cookie security In production, session cookies need the HttpOnly, Secure, and SameSite flags applied. Between them, these settings block scripts from reading the cookie directly, limit transmission to HTTPS connections only, and lower the chances of a cross-site request forgery attack succeeding.

Two-factor secrets Plaintext storage for these secrets is risky. A better approach is encrypting them at rest, so that if the database were ever compromised, the secrets themselves would remain unusable to whoever got access

- Two-factor secrets should be encrypted at rest rather than stored in plaintext fields.

D. Comparison with Industry Security Baselines

Positioning the mitigations of Section VIII-B against the OWASP ASVS framework [10] clarifies which hardening step would yield the largest marginal gain in assurance for the least additional engineering effort. Password hashing and server-side session handling are foundational Level 1 requirements and are already satisfied by the prototype. The TOTP-style second factor partially satisfies multi-factor authentication expectations but falls short of Level 2 criteria because the shared secret is not encrypted at rest and verification attempts are not rate-limited, both of which are explicit ASVS controls. Closing these two specific gaps would likely improve overall assurance more than adding further authentication factors, since brute-force resistance and secret confidentiality are prerequisites for a second factor to deliver its intended protection in the first place.

IX. FUNCTIONAL VALIDATION AND RESULTS

The resulting implementation demonstrates successful functional integration across registration, login, profile updates, blog publishing, media uploads, browsing, comments, likes, and two-factor authentication. The application behaves coherently under typical browser-driven interactions and establishes a practical reference architecture for secure content publication. Based on the qualitative evaluation, the system appears well-suited for personal blogging, academic demonstrations, and small-scale community publishing use cases. We manually walked through the core user journeys registration through 2FA-protected login, creating a post with an image attachment, and interacting via comments and likes and none of these produced unhandled errors. The database state after each action also matched the schema outlined in Section VII. These results suggest the modules we built actually hold up under reliable use and do meet the functional requirements we were testing for. Put together, the evaluation shows a platform that's stable, secure, and fairly easy to use for publishing content on the web.

X. LIMITATIONS AND FUTURE SCOPE

Right now, the system doesn't have a role-based admin module, email verification, password reset, or any payment/commerce

workflows these simply weren't built into the codebase. Because of that, it's better suited to personal or small-team publishing rather than anything at large scale. There's also room to grow this further moderation tools, analytics dashboards, notifications, maybe cloud-native deployment, plus stronger search and recommendations, all on top of the security hardening already covered in Section VIII-C. Beyond that, things like automated testing, CI/CD, and proper monitoring would genuinely help both for day-to-day reliability and for keeping the codebase manageable as it grows. Once those pieces are in place, there's a reasonable path for the platform to scale up to bigger communities without giving up on security, performance, or uptime.

One limitation worth calling out again it's already flagged qualitatively in Table I is that there's no automated regression-test suite. The manual validation in Section IX is fine for confirming that the main user journeys work as expected at this point in time, but it won't catch regressions if routes, schemas, or middleware change later on. A reasonable next step, before tackling any of the scalability work discussed in Section XII, would be to build a lightweight integration-test suite covering registration, login, blog creation, and 2FA verification, run against an isolated in-memory database. That would turn today's one-off validation into something closer to a repeatable safety net.

XI. CONCLUSION

This paper set out to design and build a secure, session-based blogging platform using Node.js, Express.js, MongoDB, and Mongoose. Along the way, the system ended up bringing together quite a few pieces: user authentication, session management, profile customization, media uploads, blog publishing, comments, likes, reactions, and two-factor authentication all wired into a modular architecture that should be reasonably easy to extend later on. Based on the functional and security evaluations, the platform holds up well as a secure web-based content management system, without adding unnecessary complexity to deploy or maintain. The security mechanisms in place genuinely help with data protection and reliability, so the platform is a good fit for educational projects, personal use, and small community publishing; it's not meant for enterprise-scale deployments. There's still clear room to grow, though: better scalability, role-based access control, a move toward cloud deployment, further security hardening, and a few more advanced features to round out performance and usability. And more broadly, the architecture itself could work as a starting point for other secure web apps that don't need a lot of extra complexity. Hopefully, the work here can be useful as a reference point for others building or researching secure web applications.

XII. EXTENDED PERFORMANCE AND SCALABILITY DISCUSSION

While Section IX confirms functional correctness, a deployable system must also be evaluated against the load and latency characteristics it is likely to encounter in practice. Since the deployment runs as a single Node.js process, every request passes through the same Express event loop — which means

CPU-heavy tasks like bcrypt hashing (used during registration and login) can momentarily block that loop when several requests come in at once. In practice, with the cost factors bcryptjs normally uses (10–12 rounds), this isn't something you'd notice at the traffic level of a personal or small-team blog. Push concurrent login attempts past a few dozen per second, though, and it starts to become a real bottleneck.

Read-heavy operations, such as browsing blog listings by author or category, benefit from indexing the corresponding fields in the Blog collection. Compound indexes on the author and category fields, together with a text index on the title and tags fields, would allow MongoDB to satisfy common queries without full collection scans, which becomes important as the number of stored posts grows beyond a few thousand documents. Because comments are embedded within each blog document, read amplification is minimal for a single post view, though pagination of long comment threads would eventually require either a maximum-length cap or a migration to a referenced comment collection, consistent with the persistence trade-off already noted in Section VII.

Horizontal scalability is constrained primarily by the use of in-process, in-memory session storage. By default, express-session keeps session data in server memory unless you configure it otherwise. That's fine for a single process, but it becomes a problem the moment you run multiple Node.js instances behind a load balancer — session state ends up fragmented across processes, so a user can get logged out or hit an auth failure simply because their request landed on an instance that doesn't have their session. Moving session storage to a shared store like Redis [11] would fix this by decoupling session state from any one process, which is really what you'd need to scale the application tier horizontally without running into these issues. Migrating uploaded media from local disk storage to an object storage service would similarly remove the dependency on a single host's filesystem, allowing the application tier to be scaled or replaced without risk of losing previously uploaded avatars or blog images.

A complementary strategy for reducing both database and application-tier load is to place a caching layer, such as a reverse proxy or an in-memory cache of rendered post pages, in front of the Express server for frequently viewed content. Because most blog posts receive the majority of their views shortly after publication, a short time-to-live cache with explicit invalidation on edit or comment would substantially reduce repeated database reads for popular posts without introducing significant staleness risk. On the data tier, MongoDB replica sets provide both read scaling, through read-preference configuration, and automatic failover should the primary node become unavailable; for the read-dominant workload profile of this system, a single primary with one or two secondaries is generally sufficient until traffic grows well beyond the deployment scenarios described in Section X.

XII. RECENT DEVELOPMENTS AND RELATED TECHNOLOGY TRENDS

Since the initial prototype was developed, the broader authentication landscape has continued to shift toward passwordless and hardware-backed mechanisms. The WebAuthn specification [9], now a W3C Recommendation, defines an authentication process built around public-key cryptography: a user's device or platform authenticator signs a challenge with a

private key that never actually leaves the device. Compared to the TOTP-based second factor used in this prototype, that's a meaningful difference WebAuthn sidesteps the shared-secret provisioning problem altogether, since there's no long-term secret being transmitted to or stored on the server in plaintext. Adding WebAuthn as an additional (or alternative) second factor would be a natural next step for the current two-factor module not as a replacement for TOTP, but alongside it, for users whose devices don't support platform authenticators. More broadly, security-standards bodies have kept pushing to formalize baseline expectations for web application hardening. The OWASP Application Security Verification Standard [10] provides a tiered checklist against which the mitigations and gaps identified in Section VIII can be mapped systematically: the mitigations already present in the prototype, namely password hashing, server-side sessions, and a second factor, correspond to Level 1 controls, while the hardening items identified as gaps, namely rate limiting, encrypted secret storage, and enforced cookie attributes, correspond primarily to Level 2 controls expected of applications that handle sensitive user data. On the deployment side, the Node.js ecosystem has seen continued adoption of process managers and container-orchestration platforms for running stateless application instances, which reflects the same externalized-session pattern discussed in Section XII. Serverless and edge-function runtimes have also become more popular for lightweight web apps, but they come with tradeoffs cold-start latency and per-invocation connection handling work differently than the always-running, persistent-connection model this architecture assumes. Adopting either approach would mean rethinking how the Mongoose connection lifecycle is handled. Separately, plenty of content-focused web apps have shifted toward a decoupled, API-first setup persistence and business logic sitting behind a REST or GraphQL interface, with the frontend built out as its own single-page app or statically generated site. This project kept the frontend and backend co-located as server-rendered pages, mostly for simplicity. That said, the routes and models are already organized in a modular way, so a later version could expose the same underlying operations through a versioned API without needing to rebuild the persistence layer.

XIV. APPLICATION DOMAINS AND DEPLOYMENT SCENARIOS

Beyond the personal blogging use case highlighted in Section X, the architecture's small footprint and modest resource needs make it useful in a few other settings too. In academic contexts, for instance, it could work well as a teaching tool for full-stack web development or applied security courses within one compact codebase, it shows working examples of password hashing, session management, second-factor verification, and file upload handling, things that are usually taught separately but rarely tied together in a single running application. For smaller organizations or open-source projects, the same setup could double as a lightweight internal knowledge base or documentation site — the category and tag fields already built into the Blog schema could just as easily organize internal articles by team or topic instead. And since authentication is already built in, access can be limited to registered members without needing a separate identity provider, which is a nice fit

for teams that don't want to stand up extra infrastructure just to publish internal content.

Once real user data such as profile biographies, avatars, and authored content is collected in any of these deployment scenarios, production use also raises data-handling and privacy considerations, including a documented data-retention policy and a mechanism for account or content deletion on request, that are outside the scope of the current prototype but would need to be addressed before onboarding non-technical, non-consenting users. Containerizing the application, packaging the Node.js process, its dependencies, and a reference MongoDB instance into a container-orchestration configuration, would further simplify deployment across these scenarios by making the environment reproducible across development, staging, and low-scale production hosts, and would provide a natural boundary for the horizontal scaling and externalized-session changes discussed in Section XII.

XV. SUMMARY OF ADVANTAGES AND LIMITATIONS

Table I consolidates the advantages and limitations discussed throughout Sections VIII through XIV into a single comparative view across four dimensions relevant to practitioners evaluating this architecture for their own deployments

TABLE I. SUMMARY OF ADVANTAGES AND LIMITATIONS BY DIMENSION

Dimension	Advantage	Limitation
Security	Layered mitigations (hashing, sessions, 2FA) map to OWASP ASVS Level 1 [10]	No rate limiting; 2FA secrets stored in plaintext
Scalability	Stateless route handlers simplify reasoning about load	In-memory sessions and local file storage block horizontal scaling
Maintainability	Clear separation of routes, models, and views	Limited automated test coverage
Deployability	Minimal external dependencies; runs on a single host	No containerization or CI/CD pipeline in the prototype

Taken together, the summary in Table I indicates that the architecture's principal strength lies in the clarity and compactness of its modular design, which makes it easy to audit, extend, and teach, while its principal limitation concerns the operational hardening, namely rate limiting, externalized sessions, and encrypted secret storage, that any prototype-to-production transition of this kind would need to address before handling untrusted, internet-scale traffic.

XVI. REFERENCES

- [1] Node.js Foundation, "Node.js," [Online]. Available: <https://nodejs.org/>. Accessed: Jul. 2026.
- [2] Express.js, "Express.js — Fast, Unopinionated, Minimalist Web Framework for Node.js," [Online]. Available:

<https://expressjs.com/>. Accessed: Jul. 2026.

[3] Mongoose, "Mongoose ODM," [Online]. Available: <https://mongoosejs.com/>. Accessed: Jul. 2026.

[4] bcryptjs, "bcryptjs," [Online]. Available: <https://www.npmjs.com/package/bcryptjs>. Accessed: Jul.

[5] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm," RFC 6238, 2011.

[6] OWASP Foundation, "OWASP Top Ten," [Online]. Available: <https://owasp.org/www-project-top-ten/>. Accessed: Jul. 2026.

[7] MongoDB, Inc., "MongoDB Manual," [Online]. Available: <https://www.mongodb.com/docs/>. Accessed: Jul. 2026.

[8] OpenJS Foundation, "Multer: Node.js middleware for handling multipart/form-data," [Online]. Available: <https://www.npmjs.com/package/multer>. Accessed: Jul. 2026.

[9] FIDO Alliance / W3C, "Web Authentication: An API for accessing Public Key Credentials," W3C Recommendation, 2021. [Online]. Available: <https://www.w3.org/TR/webauthn-2/>.

[10] OWASP Foundation, "OWASP Application Security Verification Standard (ASVS)," [Online]. Available: <https://owasp.org/www-project-application-security-verification-standard/>. Accessed: Jul. 2026.

[11] Redis Ltd., "Redis Documentation," [Online]. Available: <https://redis.io/docs/>. Accessed: Jul. 2026.

[12] Mozilla Developer Network, "HTTP Cookies," [Online]. Available: <https://developer.mozilla.org/docs/Web/HTTP/Cookies>. Accessed: Jul. 2026.

[13] Mozilla Developer Network, "HTTP Sessions," [Online]. Available: <https://developer.mozilla.org/docs/Web/HTTP>. Accessed: Jul. 2026.

[14] Open Web Application Security Project, "Cross-Site Scripting Prevention Cheat Sheet," [Online]. Available: <https://cheatsheetseries.owasp.org/>. Accessed: Jul. 2026.

[15] Open Web Application Security Project, "SQL Injection Prevention Cheat Sheet," [Online]. Available: <https://cheatsheetseries.owasp.org/>. Accessed: Jul. 2026.

[16] Open Web Application Security Project, "Authentication Cheat Sheet," [Online]. Available: <https://cheatsheetseries.owasp.org/>. Accessed: Jul. 2026.

[17] OpenJS Foundation, "Express-session Documentation," [Online]. Available: <https://www.npmjs.com/package/express-session>. Accessed: Jul. 2026.

[18] OpenJS Foundation, "Passport.js Authentication Middleware," [Online]. Available: <https://www.passportjs.org/>. Accessed: Jul. 2026.

[19] MongoDB, Inc., "MongoDB Atlas Documentation," [Online]. Available: <https://www.mongodb.com/atlas/database>. Accessed: Jul. 2026.

[20] Google Developers, "Web Fundamentals," [Online]. Available: <https://web.dev/>. Accessed: Jul. 2026.

[21] Ecma International, "ECMAScript® Language Specification," [Online]. Available: <https://tc39.es/ecma262/>. Accessed: Jul. 2026.

[22] OpenJS Foundation, "npm Documentation," [Online]. Available: <https://docs.npmjs.com/>. Accessed: Jul. 2026.

[23] Docker Inc., "Docker Documentation," [Online].

Available: <https://docs.docker.com/>. Accessed: Jul. 2026.

[24] GitHub, "GitHub Docs," [Online]. Available: <https://docs.github.com/>. Accessed: Jul. 2026.

[25] NIST, "Digital Identity Guidelines," Special Publication 800-63B, 2020.