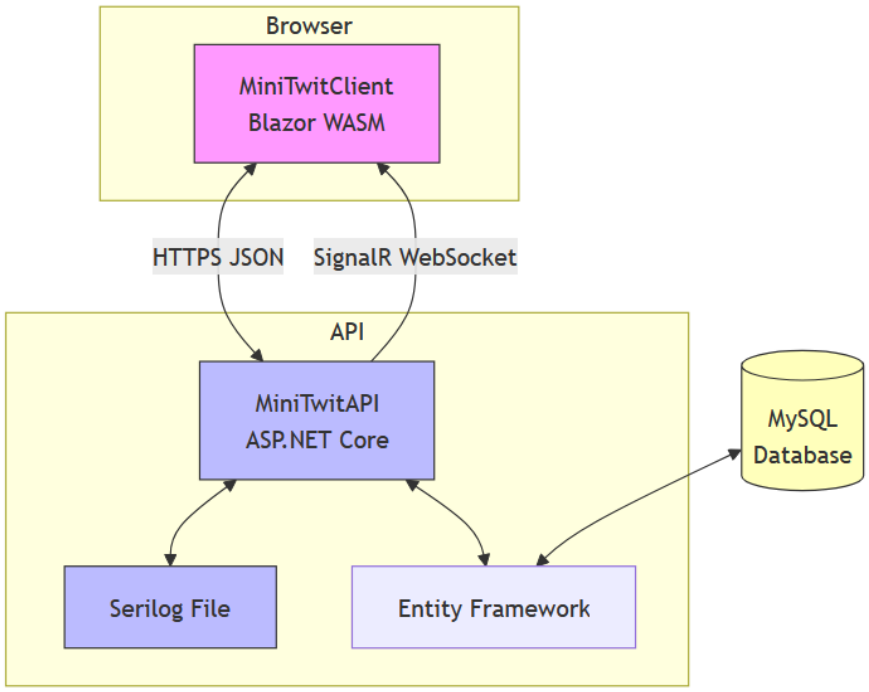
DevOps, Software Evolution and Software Maintenance



Adrian Kari, adrka@itu.dk
Lukas André Rasmussen, lanr@itu.dk
Mathias Hvolgaard Darling Larsen, mhvl@itu.dk

GitHub:
https://github.com/Lukski175/MiniTwit-FS

30th May 2025

Contents

1 System's Perspective

1.1 Design & Architecture

This section explains the overall structure, design decisions, and internal interactions of *MiniTwitFS*, which consists of two top-level applications:

- **MiniTwitAPI** – an ASP.NET Core Web API that exposes functions that talks to our Database.
- **MiniTwitClient** – a Blazor WebAssembly Application that runs entirely in the browser and communicates with the API.

The system implements a simplified Twitter-style micro-blog ("MiniTwitFS") where users can publish short messages, follow one another in real time, and view their own timelines.

1.1.1 Architectural Style

Single-Page Client The Blazor WebAssembly front-end is downloaded once and then runs entirely in the browser. Navigation and state changes are handled client-side, so the UI behaves as a Single Page Application (SPA).

REST + *occasional* SignalR All operations (e.g. `/msgs`, `/follows`, timeline queries) are standard REST requests over HTTPS. SignalR is currently used only for a secondary channel that streams live log messages to a developer page.

API Structure Inside **MiniTwitAPI** the code is organised as different layers. *Presentation* (controllers and hubs) → *Application* (DTOs, use-case services) → *Domain* (entities, business rules) → *Infrastructure*. Each layer has no compile-time dependency on an outer layer. All infrastructure (MySQL via EF Core, Serilog logging, authentication) is provided through dependency injection at startup.

1.1.2 High-Level Component View

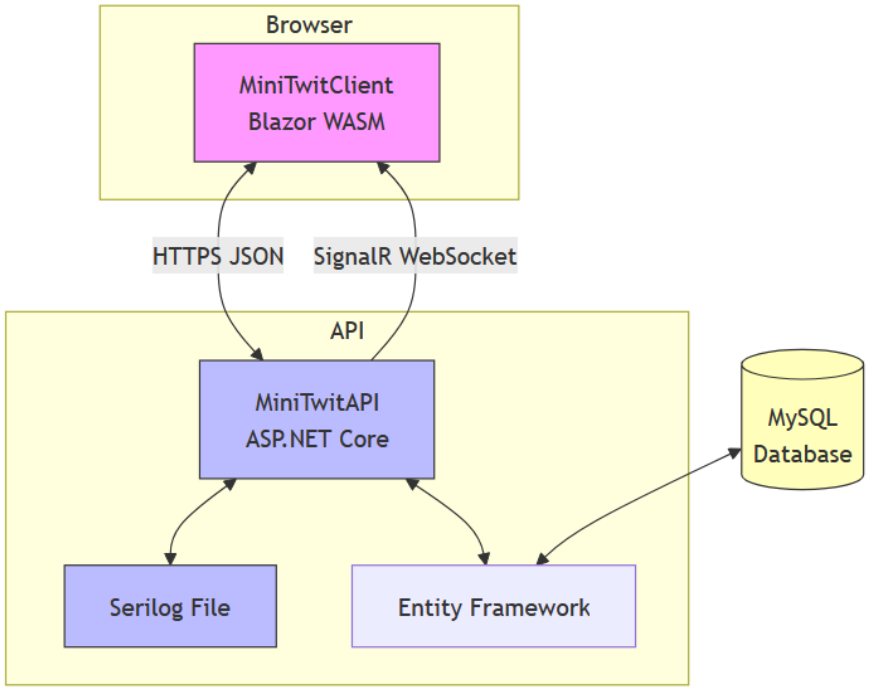


Figure 1: System Architecture

Explanation MiniTwitClient communicates with the API through CORS-enabled HTTPS endpoints, authenticating with JSON Web Tokens (JWT). When a user posts a new message, the API stores the record in the database and broadcasts a *MessageAdded* event over SignalR. Connected admin clients update their logs instantly without refreshing the page. See Figure 1 for a visualization.

1.1.3 Internal Structure of `MiniTwitAPI`

- **Controllers** Request endpoints that validates input and implements use-cases such as PostMessage, FollowUser, and GetTimeline.
- **Domain Models** Database Entities: User, Message, and Follower. Also DTO's and request models, e.g. AddMessageRequest and LoginRequest.
- **Infrastructure** AppDbContext (EF Core with migrations), JWT authentication middleware, ILogger and SignalR.
- **Configuration** The base *appsettings.json* is overridden by *appsettings.Development.json*

or *appsettings.Production.json* according to the DOTNET_ENVIRONMENT variable.

The database connection string, JWT signing key and the simulator's Basic-auth header are *not* checked into Git, instead they are injected at runtime via DigitalOcean "App secrets".

Injections, mappings, authentication, and different services and rules, are all setup on startup through the *Program.cs* file.

The migration command is executed on production release as part of the release pipeline, in order to guarantee that the runtime database schema always matches the current code.

Serilog logs are written to the console during development and to DigitalOcean's log stream in production, driven by the *Serilog:WriteTo* section.

Figure 2 shows a class diagram of the API MinitwitController.

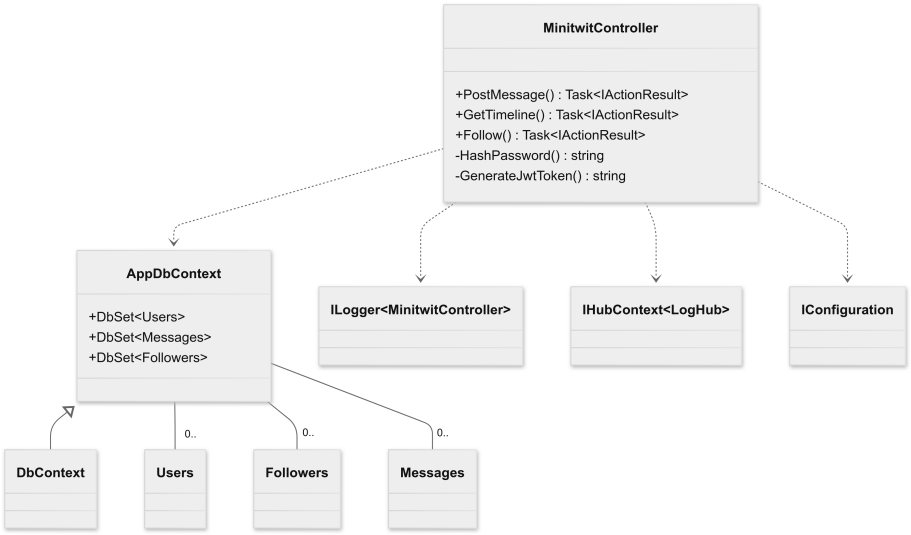


Figure 2: Class diagram of MinitwitController in the API

1.1.4 Internal Structure of `MiniTwitClient`

The client is compiled from C# Razor components to WebAssembly. A lightweight state–container service called UserState keeps the currently authenticated user (name and JWT). Components inject this service and read its properties.

- **Pages** / register , /login, /public, /timeline, /user/username.
- **Shared Components** NavMenu, MessageCard, FollowButton, and toast notifications.

- **Services** MinitwitController (typed REST client), optional WebSocketService for live log streaming, Tailwind CSS for styling, and small JS-interop helpers.
- **Configuration** The HTML page sets a window.appConfig.apiEndpoint value, where the Blazor code reads that value through JavaScript interop, so the same build of the site works in every environment without being rebuilt. It is setup so that the value gets overwritten both when running locally, or when built in the pipeline, with the correct values - controlled by the project settings.

  If the user has logged in, the generated JSON Web Token is stored in sessionStorage. When the page reloads the token is re-read, and, if it is present, a Bearer header is attached to every HTTP request.

  Tailwind CSS compiles the code into an index.html, and bundled CSS files, when pushing to production;

  > *Tailwind automatically removes all unused CSS when building for production, which means your final CSS bundle is the smallest it could possibly be.* [7]

Figure 3 shows a class diagram example from the client.

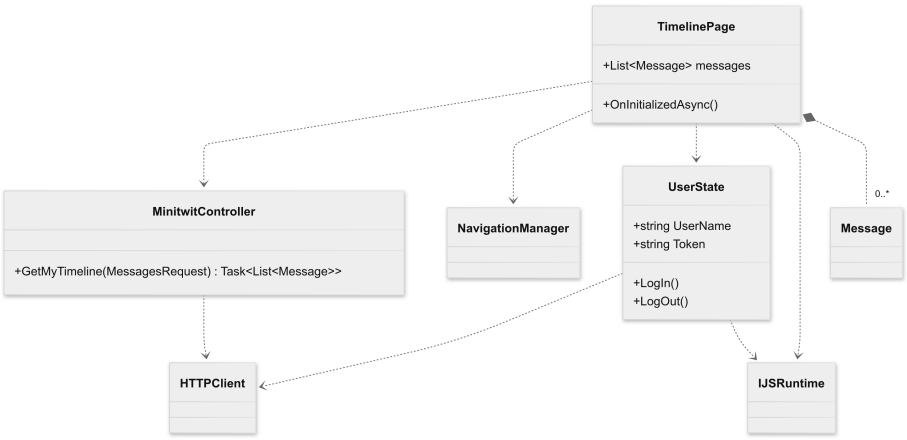


Figure 3: Class diagram of Timeline page

1.1.5 Typical Runtime Interactions

When a real user wants to interact with our system, they do so through our web client. Here, the user is restricted by the user interface we have provided, however the requests that the user does still goes through all of the systems.

Figure 4 shows a sequence diagram, depicting the different calls between systems, in a successful post-message request from the client.

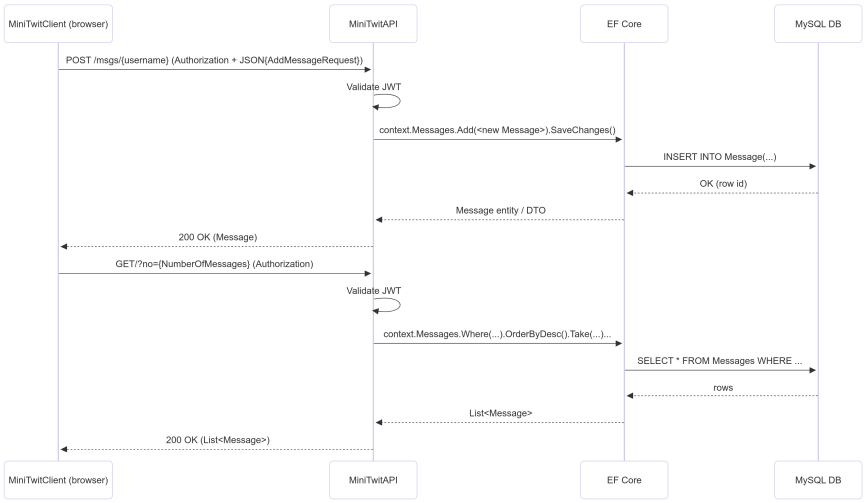


Figure 4: Client doing post request to the API, and the response they receive

However, the simulator connects directly to the API, uses a Basic authorization header, and doesn't use the response, which can be seen in figure 5.

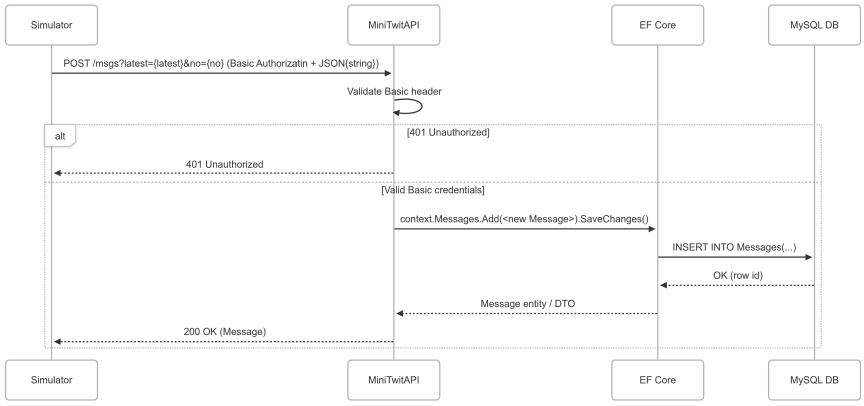


Figure 5: Simulator doing post request to the API

1.1.6 Authentication & Authorization - Current vs Target

Current implementation

- **Client users** Receive a JSON Web Token (JWT) at login. Defaults to Simulator Basic authorization when not logged in. All requests carry the current token.

- **Simulator** Uses a hard-coded Basic auth header that is accepted by a custom middleware.
- **Endpoint protection.** Controllers are simply decorated with `[Authorize]`, which validates the JWT; however, the middleware lets any request with the simulator Basic authorization pass, giving it the same privileges as a logged-in user.

Identified shortcomings

- The Basic credential leaks to every unauthenticated visitor via network traces.
- Any attacker who copies the header can post, follow, or delete on behalf of any user, e.g. through Postman.

Target implementation

- **No Basic fallback in the client.** Anonymous browsers send **NO** Authorization header. Authenticated browsers sends a JWT only.
- **Dual authentication schemes.** The API registers both `JwtBearer` (for browsers) and `SimulatorBasic` (for the simulator). A dedicated handler validates the secret header and tags requests as *simulator* for logging.
- **Explicit endpoint rules.** Write actions are protected by something similar to `[Authorize(AuthenticationSchemes = "Bearer, SimulatorBasic", Policy = "UserMatchesRoute")]` meaning a request is allowed only when either a valid JWT whose subject matches the {`username`} route *or* the simulator header is present. Public GET endpoints are marked `[AllowAnonymous]`.

Why this matters. Removing the Basic fallback closes the inadvertent backdoor while still letting the simulator operate. Separate schemes enable clearer auditing, future rate-limits, and rotation of the simulator secret without touching user flows. Current setup is not secure at all.

1.1.7 DigitalOcean Setup

Our system is deployed on DigitalOcean using an IaaP App Platform where the API is configured as a web service and the client as a static site. The platform has a reference to our repository on GitHub, where the clients output directory looks for a build in the dist/wwwroot located in the MiniTwitClient project. Additionally, we have a MySQL database set up as well, which is attached to the platform, and the SnapShooter add-on to backup the database.

1.2 Dependencies, Technologies & Rationale

All dependencies and technologies are shown on Table 1 and Table 2.

Technology	**Citation**	**Description**
Microsoft ASP.NET Core 8 (Web API)	[16]	A framework for building RESTful web services using C# and .NET. Compared to Node/Express, it provides static typing, built-in dependency injection, and higher baseline throughput in Linux containers.
Entity Framework Core 8 (Code-First)	[8]	An object–relational mapper that maps .NET objects to SQL tables and handles migrations. Compared to lightweight mappers like Dapper, its automatic migrations and LINQ queries remove boilerplate while still allowing raw SQL when required.
Blazor WebAssembly (stand-alone)	[9]	Runs C# directly in the browser via WebAssembly with no JavaScript bundle. Compared to a React+TypeScript front-end, it keeps the entire stack in one language and reuses existing DTOs.
SignalR	[10]	Real-time library that pushes server updates to clients over WebSockets or graceful fallbacks.
Tailwind CSS	[7]	A utility-class framework applied directly in markup. Compared to Bootstrap, its purge step removes unused styles, shrinking the CSS bundle and keeping component markup concise.
Docker	[6]	Packages the app and its dependencies into containers, ensuring identical development and production environments. Compared to shell scripts on raw VMs, it eliminates "works on my machine" drift and enables one-command integration test environments.

Table 1: Technology stack with citations and rationale.

Technology	Citation	Description
DigitalOcean	[15]	Cloud provider offering computing, storage, and managed MySQL. We chose it over Azure and Google Cloud because it was the only service we could get up and running successfully.
GitHub Actions	[5]	CI/CD that runs tests, lints, builds images, and deploys on every pull request. Compared to Jenkins, it requires no self-hosted server or plugin maintenance.
MySQL	[11]	Relational database engine powering production. Compared to PostgreSQL, the team already knew its tooling and DigitalOcean offers it as a fully managed service.
xUnit	[1]	Unit-testing framework for .NET with a clean attribute syntax. Compared to MSTest or NUnit, it executes tests in parallel out of the box and integrates tightly with the .NET CLI.
Serilog	[12]	Structured logging library that writes JSON to files, Seq, or Grafana/Loki. Compared to the default Microsoft logger, it produces queryable, schema-less logs without extra adapters.
SnapShooter	[2]	Backup service creating scheduled and on-demand snapshots of volumes and databases. Compared to a hand-rolled `mysqldump`+cron, it handles retention policies and one-click restores for us.

Table 2: Technology stack with citations and rationale.

See Figure 6 for a diagram of the entire system and their dependencies to each other.

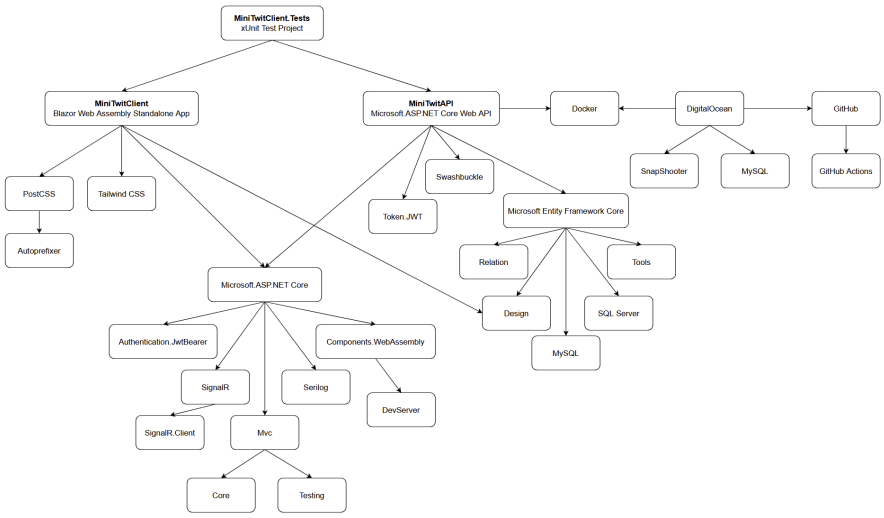


Figure 6: Dependency diagram of the entire system.

2 Process' perspective

2.1 Continuous Integration/Continuous Deployment

For deploying the system, our CI/CD pipeline consists of GitHub Actions workflows that trigger deployments on DigitalOcean. When changes are pushed to the main branch on GitHub, the deploy-api.yml workflow is triggered if there are updates to the API, while the build-client.yml workflow is triggered if there are changes to the client. If deploy-api.yml detects changes to the client, it cancels itself, allowing build-client.yml to take over. This workflow removes the existing dist folder (containing the previous build), creates a new client build in that folder, and then uses the GitHub Actions bot to push the updated build back to the main branch. The reason for a bot to push to main is that the updating of the dist folder doesn't change the actual repository, but only the instance that exists in the workflow. Then, at last, it triggers the deploy-client.yml workflow, which immediately deploys the client to DigitalOcean.

See Figure 7 for a sequence diagram illustrating the continuous deployment process using GitHub Actions. It is worth noting that Step 8, which cancels the workflow, stops the current process triggered by the push to the main branch. This occurs because another push to main by another developer, triggers a new workflow run, effectively restarting the process.

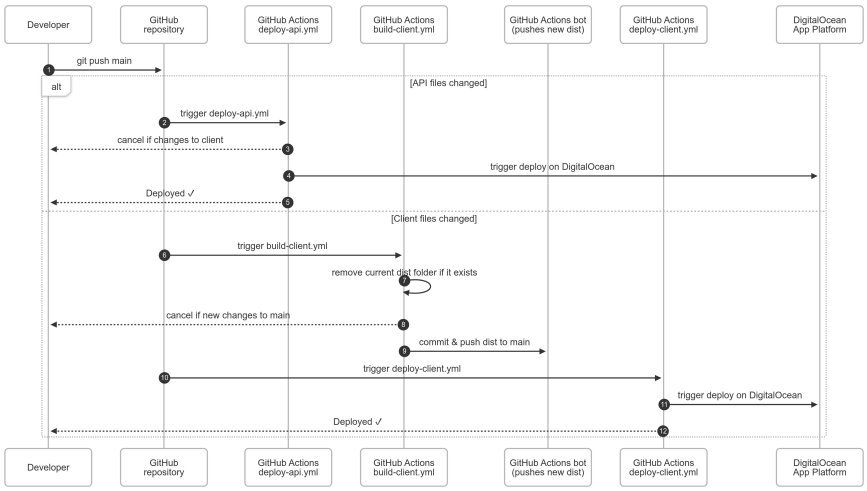


Figure 7: Sequence diagram of the deployment process using GitHub Actions.

2.1.1 Scaling & upgrade strategy

Our deployment runs on DigitalOcean App Platform, which enables us to apply horizontal scaling, zero-downtime upgrades, and rollbacks.

It uses the blue-green rollout strategy. creating a new set of containers, that gets updated and traffic transferred to them. With the option to roll-back upon failure [4].

Every build pushed by GitHub Actions is tagged with the Git SHA. This means we can redeploy the previous SHA, instantly restoring the last known-good state.

Each container can be monitored for average CPU utilization, meaning the amount of instances running at a time is constantly changing, depending on settings, optimizing our instance utilization [3].

2.2 Monitoring

At the moment we rely only on the basic CPU/RAM graphs that DigitalOcean App Platform provides. In the future, it would be wise to implement a full monitoring stack. Here we could either use a built-in addon to the App Platform, such as 360 Monitoring, or do the Prometheus + Grafana stack inspired by the course example:

Collection layer (Prometheus)

- **Application metrics** – expose the built-in /metrics endpoint from ASP.NET Core via the *prometheus-net.AspNetCore* package; record HTTP latency, status-code counts, and SignalR connection count.
- **Client metrics** – ship front-end Web Vitals (Cumulative Layout Shift, First Input Delay, Largest Contentful Paint) to Prometheus with the *@grafana/web-vitals-exporter* script.
- **Infrastructure exporters** – run the *mysql_exporter* for DB metrics

Visualisation layer (Grafana)

- Run Grafana in a Docker container and point it at a folder that contains dashboard JSON files. When the container restarts it automatically reloads the same dashboards.
- The main dashboard shows five things side-by-side: request rate, error rate, 95th-percentile latency, database queries per second, and basic CPU / memory use.
- Alert rules as simple YAML files. For example, send an alert if the 95th-percentile latency stays above 500-ms for five minutes, or if the number of open MySQL connections hits the limit.

Benefit This stack would give a single place for viewing *application*, *database*, and *host* telemetry, with reproducible dashboards under version control and alert rules. In practice it would cut mean-time-to-detect and provide data for capacity planning.

2.3 Logging

As mentioned earlier, we chose to implement our own log viewer. The API sends lines from the logs to the client, which is loaded 50 lines at a time in a textbox, which loads more when scrolling up. SignalR is used for updating the logs in real time. It can be accessed on the `/logs` page, only to administrators. An administrator account can be accessed through this login:

$$helgeandmircea \tag{1}$$

$$sesame0uvr3toi \tag{2}$$

2.4 Security

Every request to MiniTwit-FS travels over HTTPS, then clears JWT authentication and role checks before it reaches the controllers. All database access flows through Entity Framework Core, which compiles parameterised queries and blocks SQL-injection attempts by design [13]. User passwords are stored as SHA-256 hashes in the database.

Sessions ride on HttpOnly cookies that expire after 30 minutes of idle time and work only from whitelisted origins set by CORS. Input DTOs carry data-annotation rules—length limits, required fields, email regexes so malformed payloads are rejected by the client.

Secrets such as DB credentials and the JWT key stay in environment variables, and any that reach the logs are masked. SnapShooter [2] takes a full database snapshot every night and keeps a seven-day archive, so we can easily restore data.

2.4.1 Pentest Tool

OWASP ZAP [17] logged 13 issues—three medium risk, none high. The mediums are all missing headers: Content-Security-Policy, anti-clickjacking (X-Frame-Options/frame-ancestors), and a publicly reachable index.html. Everything else is low or informational. Fix with a strict CSP (default-src 'self'), X-Frame-Options: DENY, HSTS, SameSite=Lax, X-Content-Type-Options: nosniff, hide backup files, then rerun ZAP to clear the medium findings.

3 Reflection Perspective

3.1 Deployment

During the initial process of refactoring MiniTwit into an ASP.NET Web API and Blazor application, we had a hard time with deployment, and spend upwards of three weeks trying to get deployment to work on Azure and Google Cloud. Three weeks after the simulator started, we finally got deployment up and running on DigitalOcean, and the simulator started on our service, and ran pretty smoothly. This resulted in us being a couple of weeks behind on the simulator. Though while we had trouble with this, we could still work on the project locally, so we didn't fall too far behind.

3.2 Configuration

We built the API first, and the client after, and thought most of the configuration was the same. We found out, however, that Blazor does not copy appsettings.json into the wwwroot bundle, as we initially thought, so the client could not read the API base-URL in production. We fixed this in *Commit 7fe71a2* by creating a *inject-config.js* that runs before building the wwwroot, and just updates the endpoint value in index.html.

3.3 Tailwind

Using Tailwind and Blazor greatly improved our speed in terms of creating a custom front-end, however the initial setup of getting Tailwind to work took longer than expected.

3.4 Database

Initially, we wanted to use SQLite, as it is a light-weight single file database, that can quickly do read/writes - even though it uses a write-lock, meaning other processes wait in queue [14]. However, we had trouble setting it up with our system, and instead pivoted to using MySQL. This is also a better choice for future proofing the system, as it can have multiple API instances update rows, through row locking - and has a better backup strategy coupled with DigitalOcean [11].

3.5 API problems

After going live we discovered that the API in production rejected simulator calls, which we found by seeing an empty *Follows* table - and fixed it late in *Commit d3516ed*. This slip shows why monitoring and integration tests matter - a monitoring stack notifying when we have large rejected calls would have warned us earlier, and tests would have failed before going into production.

3.6 Branch protection

One teammate had a broken local environment and started pushing directly to main for every test run. In a real production setting we would enforce a "main is protected" rule with PR checks and external review-apps, for example *Snyck* for automated security testing.

3.7 ChatGPT

ChatGPT was used for boiler-plate Razor code, generating Mermaid diagrams, revising our report, and also used often when debugging errors, basically as a search engine. This proved useful and greatly enhanced our production speed and ability to compare options.

3.7.1 If we had to do it again

Looking back, our biggest delays came from infrastructure problems rather than application code. If we were to restart the MiniTwitFS project tomorrow, we would keep the .NET/ASP.NET Core + Blazor tech stack (still the fastest path for us) but change the order and discipline of our work:

Start with a API refactor After refactoring the Flask prototype, we would create a bare-bones ASP.NET Core API, and wire it into the continuous-integration pipeline before touching the Blazor client. Every subsequent commit would require the API test-suite to pass before merging to main.

Tag and document every milestone Instead of single big deploys, we would use GitHub Releases at the end of each sprint, attach a short changelog, and potentially create a matching staging environment.

DigitalOcean Droplets App Platform solved our immediate hosting pain, but also hid useful knobs (timeouts, custom health probes, custom deployment). Next time we would utilize Droplets, run Docker Swarm, and treat the Swarm file as infrastructure-as-code. That gives us full control over our update strategy and lets us customize the deployment.

Fix authentication from the outset The simulator's back-door Basic credential never got fixed, and should have never been an issue - we should never

have made the client act as the simulator.

Introduce a dedicated identity service For future scaling we would break authentication into a tiny "Auth API" that issues JWTs and manages roles. The main API would then accept Role=moderator tokens without owning password logic.

These changes would have shaved at least three weeks off the project timeline and given us cleaner release artefacts, faster feedback from the simulator, and simpler rollbacks when a refactor went sideways.

References

[1] *About xUnit.net*. 2025. URL: https://xunit.net/.

[2] *Back up your servers, databases, and applications with confidence.* 2025. URL: https://snapshooter.com/.

[3] DigitalOcean. *How to Scale Apps in App Platform*. 2025. URL: https://docs.digitalocean.com/products/app-platform/how-to/scale-app/?utm_source=chatgpt.com.

[4] DigitalOcean. *My app deployment failed because of a health check*. 2025. URL: https://docs.digitalocean.com/support/my-app-deployment-failed-because-of-a-health-check/?utm_source=chatgpt.com.

[5] GitHub. *Automate your workflow from idea to production*. 2025. URL: https://github.com/features/actions.

[6] Solomon Hykes. *Develop faster. Run anywhere.* 2025. URL: https://www.docker.com/.

[7] Tailwind Labs. *Rapidly build modern websites without ever leaving your HTML*. 2025. URL: https://tailwindcss.com/.

[8] Microsoft. *Entity Framework Core*. 2024. URL: https://learn.microsoft.com/en-us/ef/core/.

[9] Microsoft. *Launch your idea to the web fast with Blazor*. 2025. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor.

[10] Microsoft. *Real-time ASP.NET with SignalR*. 2025. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/signalr.

[11] *MySQL Database*. 2025. URL: https://www.mysql.com/products/enterprise/database/.

[12] Serilog. *Why Serilog?* 2025. URL: https://serilog.net/.

[13] *SQL Queries*. 2025. URL: https://learn.microsoft.com/en-us/ef/core/querying/sql-queries?tabs=sqlserver.

[14] *SQLite*. 2025. URL: https://www.sqlite.org/appfileformat.html.

[15] *The simplest cloud that scales with you.* 2025. URL: https://www.digitalocean.com/.

[16] Rick Anderson Tim Deschryver. *Tutorial: Create a controller-based web API with ASP.NET Core*. 2025. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-9.0&tabs=visual-studio.

[17] *Zed Attack Proxy (ZAP)*. 2025. URL: https://github.com/zaproxy/zaproxy.