Advisory Report

FileSender

SURF
Product Manager: William van Santen
Mentor: Rogier Spoor

Windesheim University of Applied Sciences
Supervising Teacher: Rob Kaesehagen
Student: Aaron Jonk (s1170298)

Date: 25/02/2025

Version: 1.1

Version Management

Version	Date	What
0.1	07/02/2025	Write introduction & advise on programming language
0.2	11/02/2025	Write advise on testing framework
0.3	14/02/2025	Advise on which database to use
1.0	18/02/2025	Edit based on feedback. Add PHP & Selenium to comparison table. Show SQLite performance in numbers.
1.1	25/02/2025	Add additional factors to the decision matrix for database advice.
1.2	05/04/2025	Add “Storage Approach” advise
1.3	11/04/2025	Add “Encryption” advise

Distribution

Version	Date	Recipient
0.3	14/02/2025	FileSender signal group chat
1.0	18/02/2025	Publish on Codeberg
1.1	25/02/2025	Publish on Codeberg
1.2	05/04/2025	Publish on Codeberg
1.3	11/04/2025	Publish on Codeberg

Contents

1 Introduction and Context 4

2 Programming Language 5

3 Testing Framework 7

4 Storage Approach 9

5 Database 10

1 Introduction and Context

This advisory report is prepared for the rewrite and implementation of new and existing functionality within FileSender version 4. During this reimplementation, recommendations will be provided, supported by the research presented in this document. These recommendations will cover various aspects, from the technologies to be used in the rewrite, such as the programming language, to the testing framework, and guidance on the future development of FileSender.

When relevant, these recommendations will be supported by a decision matrix. This matrix assigns scores to different alternatives based on how well they meet specific requirements. By summing up the scores for each option across all requirements, a total score is calculated, making it clear which option is most suitable for the project. Each option is scored per requirement on a scale from 0 to 10. The score indicates how well the option satisfies the requirement and how important that requirement is for the project.

- A score of 0 means the option does not meet the requirement at all.
- A score of 10 means the option fully meets the requirement and the requirement is highly important.
- For example, if an option meets a requirement but the requirement is not crucial, it might receive a score of 6.

2 Programming Language

The current versions of FileSender (versions 1, 2, and 3) are written in PHP. The first decision to make for the reimplementation of FileSender is selecting the appropriate programming language. The application needs to be fast, memory-safe, secure, and capable of serving static files as well as creating potential API endpoints.

The technologies being considered for comparison are PHP, Node.js (JavaScript), Python, Golang, and Rust.

	PHP	Node.js	Python	Golang	Rust
Performance (response-times)	0	3	0	7	10
Performance (memory usage)	3	0	3	10	7
Security-first design	0	0	0	10	7
Capable of serving static files	10	10	10	10	10
Good developer experience	7	7	7	5	7
Preferred by SURF	0	0	0	1	0
Total	20	20	20	43	31

Table 1

When evaluating the performance of the programming languages, Golang and Rust are closely matched[1]. As a result, PHP, Node.js and Python are no longer considered suitable candidates for the rewrite.

Another crucial factor to consider is security, which is the primary reason for reimplementing FileSender. Due to Go's simplicity, its built-in garbage collection, and its extensive standard library, Go is preferred over Rust in this regard[2]. In addition, Go's standard library provides a robust set of tools that minimize the need for external

[1] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2021). Ranking Programming Languages by Energy Efficiency. In GitHub. Universidade da Beira Interior. Accessed on 7 February 2025. https://haslab.github.io/SAFER/scp21.pdf Page 16

[2] Arundel, J. (2025, 3 February). Rust vs Go in 2025 — Bitfield Consulting. Bitfield Consulting. Accessed on 7 February 2025. https://bitfieldconsulting.com/posts/rust-vs-go

dependencies, reducing the attack surface and enhancing the security posture of the application.

Furthermore, Go is particularly well-suited for building enterprise software due to its strong concurrency model, scalability, and ease of development. Its minimal runtime dependencies and straightforward tooling make it an excellent choice for creating high-performance, maintainable enterprise applications[3].

Lastly, it's worth mentioning that I have no prior experience with Go, which slightly lowers its score in terms of developer experience. However, despite this, Golang stands out as the best overall option. Therefore, Golang is the recommended programming language for the FileSender rewrite.

[3] *Case Studies – The Go Programming Language*. Accessed on 24 February 2025. https://go.dev/solutions/case-studies

3 Testing Framework

With the programming language chosen for FileSender version 4, it's important to establish a proper testing strategy for the new application. Go's built-in unit testing library, included in its standard library, will be utilized for application unit tests. These tests can conveniently run within the CI/CD pipeline, making them an ideal fit for our development workflow.

In addition to unit tests, end-to-end testing will be critical. These tests should simulate different browsers and configurations, such as disabling JavaScript in the browser, to ensure FileSender's robustness. To achieve this, we have evaluated three interesting libraries: Playwright, Cypress, and Puppeteer.

	Playwright	Cypress	Puppeteer	Selenium
Can be run in pipelines	1	1	1	1
Support turning JavaScript off in browser	10	0	10	10
Support for Chromium, Firefox & WebKit (Safari) testing	10	0	3	10
Total	21	1	14	21

Table 2

The first and perhaps most crucial feature is the ability to run tests within CI/CD pipelines. Automated tests should be executed with every pull request, ensuring continuous integration. Fortunately, all three options are compatible with CI/CD pipelines.

The second key requirement is the ability to disable JavaScript during testing. This is a critical feature for simulating real-world user scenarios where JavaScript may be disabled. Cypress falls short in this area, as it is heavily dependent on JavaScript for its testing framework. Although there is an open issue discussing this feature, no active development is underway to address it.

The third requirement is extensive browser support, covering Chromium, Firefox, and WebKit (Safari). Playwright and Selenium excel in this category, providing full support for all three major browsers. Puppeteer, on the other hand, offers limited compatibility, and Cypress lacks support beyond Chromium-based browsers.

Differences between Playwright and Puppeteer

Here are the differences between Playwright and Puppeteer.

Parameter	Playwright	Puppeteer
Browser Support	Supports multiple browsers including Chrome, Firefox, and WebKit.	Focuses mainly on Chrome and Chromium-based browsers, with limited support for others.

Figure 1, source

In the end, both Selenium and Playwright remain as the top choices, having received the same score in the comparison. However, Playwright stands out due to its easy learning curve and efficient parallel execution, which generally makes it faster than Selenium[4].

Given these advantages, Playwright is the most suitable option, offering robust and flexible testing capabilities for FileSender version 4.

[4] Idowu Omisola. (2023, January 11). *Playwright vs. Selenium in 2025: In-Depth Comparison*. @ZenRowsHQ; ZenRows. Accessed on 18 February 2025. https://www.zenrows.com/blog/playwright-vs-selenium#playwright-vs-selenium

4 Storage Approach

In the initial stages of the FileSender reimplementation, the design included a database to manage data storage. However, as development progressed, it became clear that introducing a database at this point adds unnecessary complexity. After reevaluating the application's data model and requirements, we advise moving away from using a database entirely and instead adopting a filesystem-based approach for data storage.

Using the filesystem simplifies the architecture and avoids the need for a separate database layer. Each user will have their own folder on the server. Their uploaded files can be stored in this folder, either directly or within subdirectories if desired. Next to each uploaded file, a separate metadata file can be placed. This metadata file can contain all required information about the file.

This structure supports all necessary functionality without the overhead of managing a database. It also provides a number of practical benefits:

1. The system becomes much simpler to develop, deploy, and maintain, since there is no need to configure or manage a database.
2. All files and related data are directly accessible on the disk, which makes backup up or syncing the application straightforward using tools like rsync or rclone.
3. The use of plain files and folders improves transparency, making it easy to inspect or debug individual items.
4. Read and write operations on the filesystem are fast and reliable, especially since FileSender is not expected to be write-intensive.
5. The structure closely mirrors how users typically think about and organize their files, making it intuitive to work with during development.

If, in the future the application evolves to require more complex data handling such as relationships between users, search indexing, or analytics, it will still be possible to integrate a database later on. Go's architecture allows for this kind of flexibility without requiring major rewrites. In that case, the reader is advised to refer to section 5, which already covers recommendations on which database software to use, with SQLite being the preferred choice. For now, though, the filesystem-based approach offers a simpler, more robust, and easier-to-maintain foundation for FileSender.

5 Database

The database is a critical component of the new FileSender application. Our goal is to keep the setup, usage, and maintenance as simple as possible. Go applications offer flexibility with databases, allowing easy swaps in the future. Therefore, the current recommendation is based on present needs and is not a permanent decision. We compared MySQL, PostgreSQL, and SQLite using a decision matrix.

	MariaDB	PostgreSQL	SQLite
Small library size	5	5	10
Scalable	3	5	0
Not an external application (built-in)	0	0	3
Performance	10	7	5
Favorable (open source) licensing	10	10	10
Installation process	7	6	10
Total Score	35	33	38

Table 3

SQLite scores the highest in the decision matrix. Our recommendation is based on the following key factors:

Simplicity & Integration
SQLite has a small library size and is embedded within the application, eliminating the need for external database configuration during setup.

Ease of Installation
Unlike MariaDB and PostgreSQL, SQLite does not require a separate installation process or service, making deployment significantly easier.

Favorable Open-Source Licensing
All three databases have permissive open source licensing models, ensuring no restrictions on usage or distribution.

Low Maintenance
Since SQLite does not run as a separate database server, it minimizes administrative overhead and simplifies maintenance.

While SQLite scored lowest in scalability and performance, this is not a concern for our current use case. The ability to switch databases in Go provides future flexibility if performance becomes an issue. Additionally, our use case is not write-intensive, so SQLite's serialized write operations are unlikely to create bottlenecks.

Furthermore, benchmarks show that SQLite is already quite fast, handling around 5,000 INSERT operations per second. This level of performance is more than sufficient for our application's needs[5].

In conclusion, we choose SQLite for its simplicity, easy integration, and minimal maintenance. The decision is flexible, and we can revisit it if our performance or scalability needs evolve.

[5] Toxigon. (2025, January 22). *SQLite Performance Benchmarks: 2025 Edition Insights.* Toxigon. Accessed on 18 February 2025. https://www.toxigon.com/sqlite-performance-benchmarks-2025-edition

6 Encryption

An important part of the FileSender application is the encryption and decryption of files on the client side (in the browser). In the new version of FileSender, this functionality should be fully supported without imposing constraints on file size. Specifically, users must be able to encrypt and upload files as large as 1TB.

To support such large files, the application must rely on a streaming or piped approach to processing file contents. This ensures that file data is not fully loaded into memory but rather read and processed in chunks. These chunks must be encrypted before being passed along for further chunking and uploading. Depending on the capabilities of the chosen encryption library, we may either encrypt a continuous byte stream or encrypt individual chunks after splitting. Both models are valid and are considered in this comparison.

For this decision, we compared three options: the browser-native SubtleCrypto API, and two external libraries: Age and Libsodium. The comparison evaluates each option across six criteria relevant to FileSender's requirements.

	SubtleCrypto	Age	Libsodium
Streaming Encryption	10	10	10
Streaming Decryption	10	Chunked (10)	Chunked (10)
Browser Support	10	8	10
Performance	10	7	7
Size	10	10	10
Secure	0	5	5
Total Score	50	50	52

All three options support encrypting data in either a streaming or chunked manner, which is essential for handling large files efficiently in the browser. The most significant differences arise in terms of cryptographic safety, performance consistency, and future readiness.

SubtleCrypto provides fast, native cryptographic operations with full browser support and zero dependency size. However, it is limited to legacy algorithms such as AES-GCM, AES-CBC, AES-CTR, and RSA-OAEP. While these algorithms are still widely supported, they are no longer recommended for modern security practices in 2025[6]. AES, particularly in modes like CBC and GCM, has known risks and limitations when misused, and SubtleCrypto requires developers to manually manage parameters like IVs and authentication tags. Additionally, RSA-OAEP is only suitable for asymmetric encryption and cannot be used directly for encrypting large file contents. These factors justify a score of 0 in the "Secure" category, as SubtleCrypto does not offer modern, forward-secure encryption primitives or safe-by-default APIs.

Age is a simplified encryption format that uses X25519 and ChaCha20-Poly1305 under the hood. It offers a safer abstraction than SubtleCrypto and reduces the chance of insecure usage patterns. While Age supports encryption in the browser, its current limitations include the lack of full streaming decryption pipeline and slightly reduced performance compared to native APIs.

Libsodium offers a complete cryptographic solution, including a ChaCha20-Poly1305 API, which is specifically designed for secure and authenticated streaming encryption. It is implemented in WebAssembly, providing high performance and security. Although it introduces a small initialization cost and adds external library weight, Libsodium is by far the most robust and future-ready option. It handles chunked or streamed encryption securely and correctly without burdening the developer with low-level cryptographic decisions.

Considering these criteria and the nature of the FileSender application, Libsodium is the recommended encryption library due to its balance of security, performance, and streaming capabilities.

[6] Hcardona. (2023, December 9). *Padding Oracle Attack: Are You Vulnerable?* Winmill. https://www.winmill.com/incorrect-aes-implementation-leaves-system-vulnerable