

# Design

FileSender



## **SURF**

Product Manager: William van Santen

Mentor: Rogier Spoor

## **Windesheim University of Applied Sciences**

Supervising Teacher: Rob Kaesehagen

Student: Aaron Jonk (s1170298)

**Date:** 17/04/2025

**Version:** 1.2

## Version management

Version	Date	What
0.1	12/02/2025	Draft of this document
0.2	13/02/2025	Finish wireframe designs & user flows
0.3	14/02/2025	Edit wireframe. Finish ERD & C4-models
1.0	18/02/2025	Update wireframes based on feedback from Signal group chat. Add captions to all images. Added missing links to prototypes.
1.1	05/04/2025	Replace ERD with “Models” section.
1.2	17/04/2025	Restructure, add “Filesystem Structure” & “Download Process” sections.

## Distribution

Version	Date	Recipient
0.3	14/02/2025	FileSender signal group chat
1.0	18/02/2025	Publish on Codeberg
1.1	05/04/2025	Publish on Codeberg
1.2	17/04/2025	Publish on Codeberg

# Contents

<b>1</b>	<b>Models</b>	<b>4</b>
1.1	File	4
<b>2</b>	<b>Filesystem Structure</b>	<b>5</b>
2.1	File Upload	6
2.2	Chunked File Upload	7
<b>3</b>	<b>Download Process</b>	<b>9</b>
3.1	File System API	9
3.2	Service Worker API	10
<b>4</b>	<b>UI-Design</b>	<b>12</b>
4.1	Upload with no JavaScript	13
4.2	Upload with JavaScript	14
4.2.1	First variant	14
4.2.2	Second variant	15
4.3	Transfers	16
4.4	Guest Vouchers	17
4.5	Download	18
<b>5</b>	<b>User Flows</b>	<b>19</b>
5.1	User Flow 1	19
5.2	User Flow 2	20
<b>6</b>	<b>C4-Models</b>	<b>21</b>
6.1	Level 1 (System Context Diagram)	21
6.2	Level 2 (Container Diagram)	22

# 1 Models

In the new FileSender design, we no longer use a traditional ERD. Since the application does not rely on relational data and is instead backed by the filesystem, there is no need to define complex relationships between entities. Instead, we define standalone data models that describe the structure of data stored per file or user.

Each model corresponds to data saved in a metadata file on disk. These files live alongside the uploaded data and are formatted in JSON, making them easy to read and modify. This section outlines the currently defined data models in the system.

## 1.1 File

The File model is used to represent a group of files uploaded by a user. It captures all relevant metadata about the upload, including ownership size, naming, and lifecycle.

This model is saved as a JSON file stored next to the actual uploaded files in the user's directory. Below is an example overview of the fields included in this model:

Field	Type	Description
<b>ID</b>	String	A unique identifier to distinguish the upload.
<b>UserID</b>	String	ID of the user who owns the upload.
<b>DownloadCount</b>	Int	Number of times the files have been downloaded
<b>ByteSize</b>	Int	Total size in bytes of all uploaded files
<b>FileName</b>	String	Path or display name of the file
<b>Chunked</b>	Bool	If the file is chunked or not
<b>Partial</b>	Bool	If the chunked file is fully uploaded or not
<b>ExpiryDate</b>	Timestamp	The date and time when the upload will expire and no longer be accessible
<b>CreationDate</b>	Timestamp	The date and time when the upload was created

Table 1

This model provides all the data required to manage files within FileSender, including basic transfer functionality, expiry handling, and download tracking.

## 2 Filesystem Structure

The FileSender application supports both chunked and non-chunked file uploads. As a result, the underlying filesystem structure plays a critical role in how data is managed, stored, and accessed.

At runtime, the application reads a state directory value from the environment variables, which determines the root working directory on disk. Inside this directory, subdirectories are created per user, where each directory is named using a hashed string derived from the user's ID. This hashing serves two purposes: it prevents directory name collisions and ensures that directory names are safe (sanitized) for filesystem usage. Additionally, it obfuscates the actual user ID, thereby enhancing privacy and security.

Each user's state directory contains one or more uploads, each consisting of file data and an associated metadata file.

```
/[state_directory]
├── [hashed_user_id]
```

Figure 1

## 2.1 File Upload

When a user initiates a file upload, the application first generates a random 16-byte identifier, encoded in Base64. This identifier uniquely represents the file and is used to name both the binary file and its corresponding metadata.

Within the user's hashed state directory:

- A metadata file is created, named after the file ID, and contains JSON data describing the uploaded file.
- A binary file is also created, with the same file ID as its name, containing the raw file data.

To retrieve a file, the client must provide both the hashed user ID and the file ID. Using this information, the system locates the metadata file to read the file name and related metadata, then streams the (optionally encrypted) file contents back to the client.

```
/[state_directory]
├── [hashed_user_id]/
│   ├── [file_id].meta
│   └── [file_id].bin
```

Figure 2

### Non chunked file upload

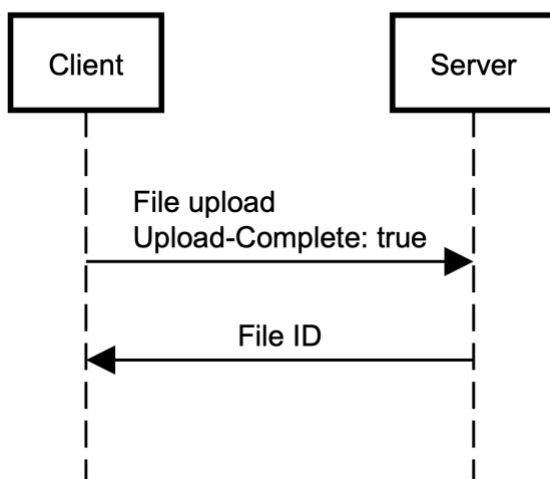


Figure 3

## 2.2 Chunked File Upload

For large files or unstable connections, the FileSender application supports chunked uploads. The upload process inspects the Upload-Complete header to determine if the upload is final or ongoing. If this header is set to false, the metadata file marks the file as partially uploaded and sets a flag indicating that it's a chunked upload.

The response also includes a Location header, specifying the endpoint to which remaining chunks should be uploaded.

When uploading chunks:

- Each request must include the Upload-Offset header, indicating the byte offset of the current chunk.
- The application creates a directory named after the file ID (if not already present).
- Each chunk is saved in a separate binary file within this directory, named after the chunk's offset (hexadecimal format).

This approach is heavily inspired by the Resumable Uploads for HTTP (IETF draft)<sup>1</sup>, keeping the door open for possible resumable upload support in the future and aligning the system with upcoming standards.

```
/[state_directory]
├── [hashed_user_id]/
│   ├── [file_id].meta
│   ├── [file_id].bin           # Chunk at offset 0
│   └── [file_id]/
│       ├── 00001000.bin       # Chunk at offset 4096 (example)
│       ├── 00002000.bin       # Chunk at offset 8192 (example)
│       └── ...
```

Figure 4

---

<sup>1</sup> Kleidl, M., Zhang, G., & Pardue, L. (2025, April 8). *Resumable Uploads for HTTP*. IETF Datatracker. Retrieved April 15, 2025, from <https://datatracker.ietf.org/doc/draft-ietf-httpbis-resumable-upload/>

## Chunked file upload

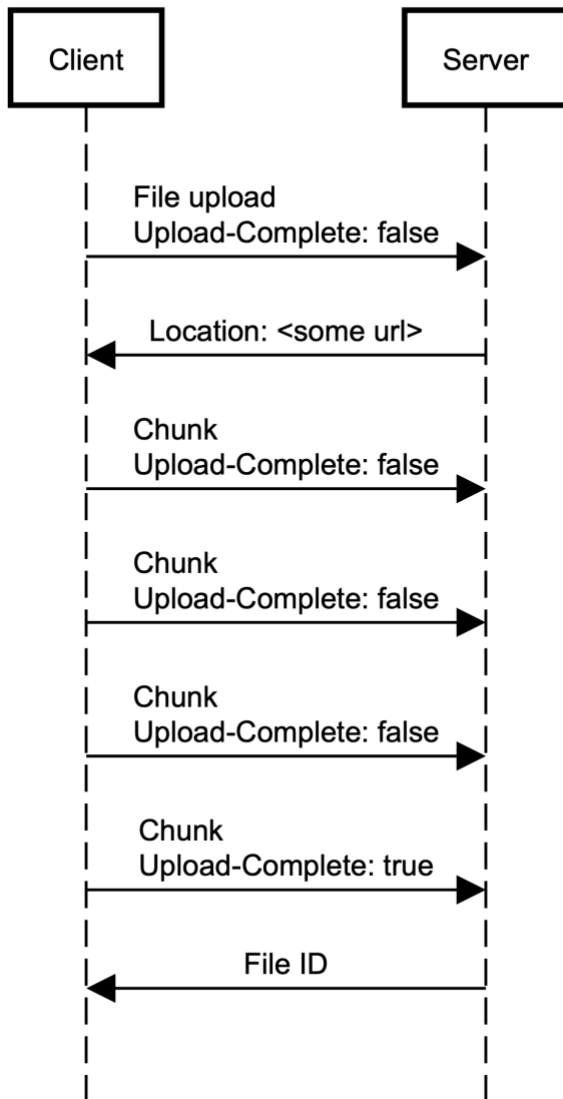


Figure 5



## 3 Download Process

The new download process for FileSender introduces added complexity. One of the primary requirements of this release is supporting encrypted, chunked downloads of files up to 1TB in size. This introduces a challenge: it is not feasible to load a file of that size entirely into memory through the browser. Even files as small as 8GB pose serious limitations when handled in memory by JavaScript alone.

To address this, the download flow must be fully stream-based at every stage. One of the key technologies enabling this is the Service Worker API.

### 3.1 File System API

During development, we explored the File System API, an experimental browser feature that showed up in the Mozilla documentation. This API allows direct streaming of data to a user's filesystem, eliminating the need for more involved solutions through service workers. While promising, the File System API is currently only supported by Chromium-based browsers and remains in an experimental phase<sup>2</sup>.


Browser compatibility													
<a href="#">Report problems with this compatibility data</a> • <a href="#">View data on GitHub</a>													
	Desktop					Mobile					Other		
	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS	Deno
<code>showSaveFilePicker</code> 	✓ 86	✓ 86	✗ No	✓ 72	✗ No	✓ 132	✗ No	✓ 87	✗ No	✗ No	✓ 132	✗ No	✗ No

Figure 6

A particularly interesting capability of this API is support for persistent storage. This makes it possible to not only pause downloads, but also to fully resume them across browser sessions. For example, a user could start a download, close their browser, and continue

<sup>2</sup> Window: `showSaveFilePicker()` method – Web APIs | MDN. (2025, March 17). MDN Web Docs. Retrieved April 17, 2025, from [https://developer.mozilla.org/en-US/docs/Web/API/Window/showSaveFilePicker#browser\\_compatibility](https://developer.mozilla.org/en-US/docs/Web/API/Window/showSaveFilePicker#browser_compatibility)

the process days later. Though not implemented yet, this opens the door to future improvements.

## 3.2 Service Worker API

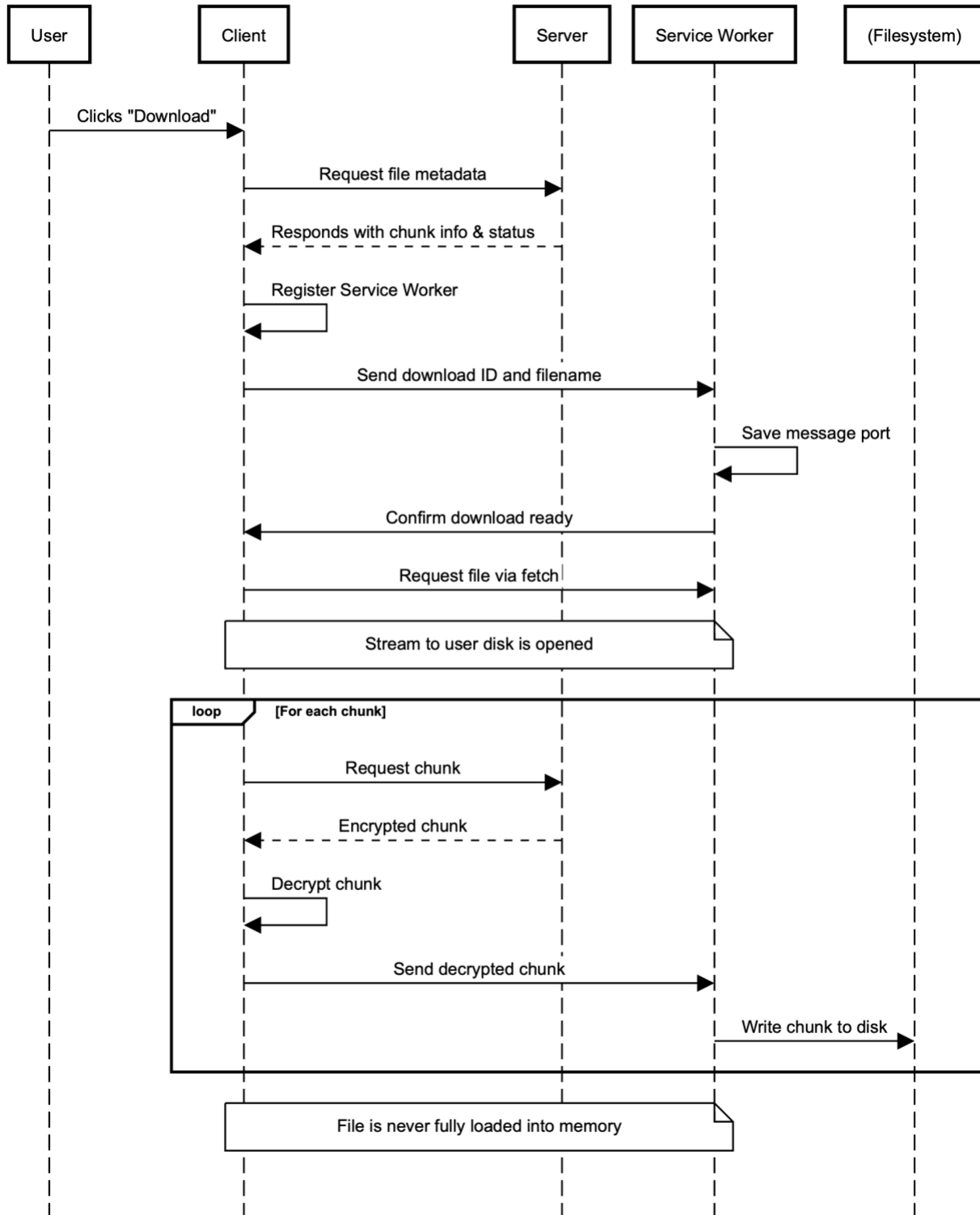
We use the Service Worker API to enable chunk-by-chunk streaming of encrypted files. When a user clicks “download”, the client first queries metadata about the requested file. It checks if the file was uploaded in chunks and verifies whether the upload is complete.

Once confirmed, the client registers a service worker. This acts as a middle layer between the browser and the FileSender backend. A communication channel is then established between the client and the service worker. The client provides the service worker with a random download ID and the name of the file to be saved locally. The service worker stores this information and notifies the client that the download endpoint is ready, even if no data has been transferred yet.

At this point, the client requests the full file from the service worker. The service worker begins a download stream to disk, and the client starts retrieving the file chunks sequentially. Each chunk is downloaded and decrypted on the client side. After decryption, the chunk is passed to the service worker, which streams it directly to the file being written.

This method enables secure downloading of encrypted files without ever holding the complete file in memory. The result is a memory-efficient and scalable approach to downloading large files in the browser.

## Service Worker-Based Chunked Download



## 4 UI-Design

UI design is a crucial aspect of the web application, as it determines what the end user will see and directly impacts the user experience.

Although the current scope of the project primarily focuses on the technical (backend) side of the application rather than its UI design, it is still essential to create a visual representation of how the application should look. This helps us effectively translate the written requirements and features into a clear, visual concept.

To achieve this, we have created wireframes / low-fidelity designs. These designs have been prototyped:

1. No JavaScript version, designed for users accessing the application without JavaScript
2. Two JavaScript-enabled upload variants, demonstrating different approaches to file uploads: one with individual file progress bars and another with a single, unified loader.

All these designs are interactive prototypes, allowing users to easily navigate through the interface and experience the intended workflows.

## 4.1 Upload with no JavaScript

The first prototype is designed for users accessing the application without JavaScript. Supporting browsers without JavaScript (or users who prioritize privacy) is a key requirement for the FileSender project. To address this, we have created a separate prototype, complete with designs, to demonstrate how the application would function without JavaScript. You can view the prototype [here](#).

The design prioritizes simplicity, focusing on basic clicks and form submissions. Since features like progress bars would require JavaScript, they are intentionally omitted. As a result, the UI is clean, straightforward, and easy to navigate.

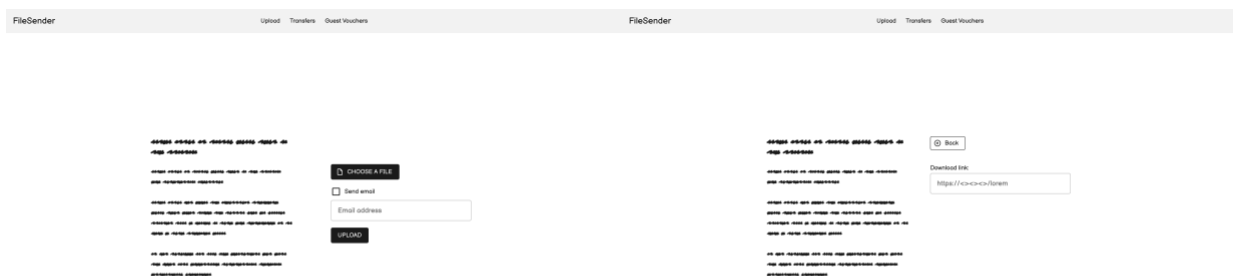


Figure 7

Figure 8

## 4.2 Upload with JavaScript

The JavaScript-enabled designs are enhanced with interactive elements, such as progress bars during file uploads, providing users with real-time feedback.

### 4.2.1 First variant

The first variant features a prototype where each selected file has its own individual progress bar. You can view the prototype [here](#).

The concept behind this design is to give users full control over each file during the upload process. Users can pause or cancel individual uploads mid-transfer. For example, if the first file is uploading and the user realizes the second file is incorrect, they can immediately stop the upload of the second file without affecting the rest of the transfer.

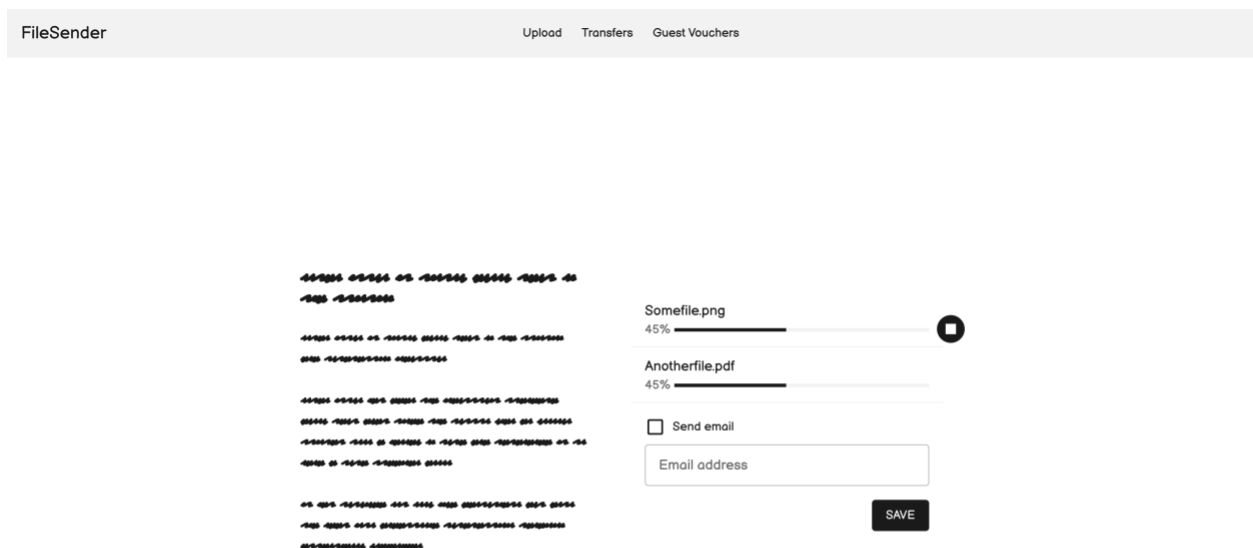


Figure 9

### 4.2.2 Second variant

The second variant takes a different approach. It uses a single, circular loader representing the overall upload progress. You can view this prototype [here](#).

In this design, there is no individual file control; instead, users can only pause or stop the upload as a whole. This approach simplifies the user experience by reducing complexity and focusing on the overall progress rather than individual files.

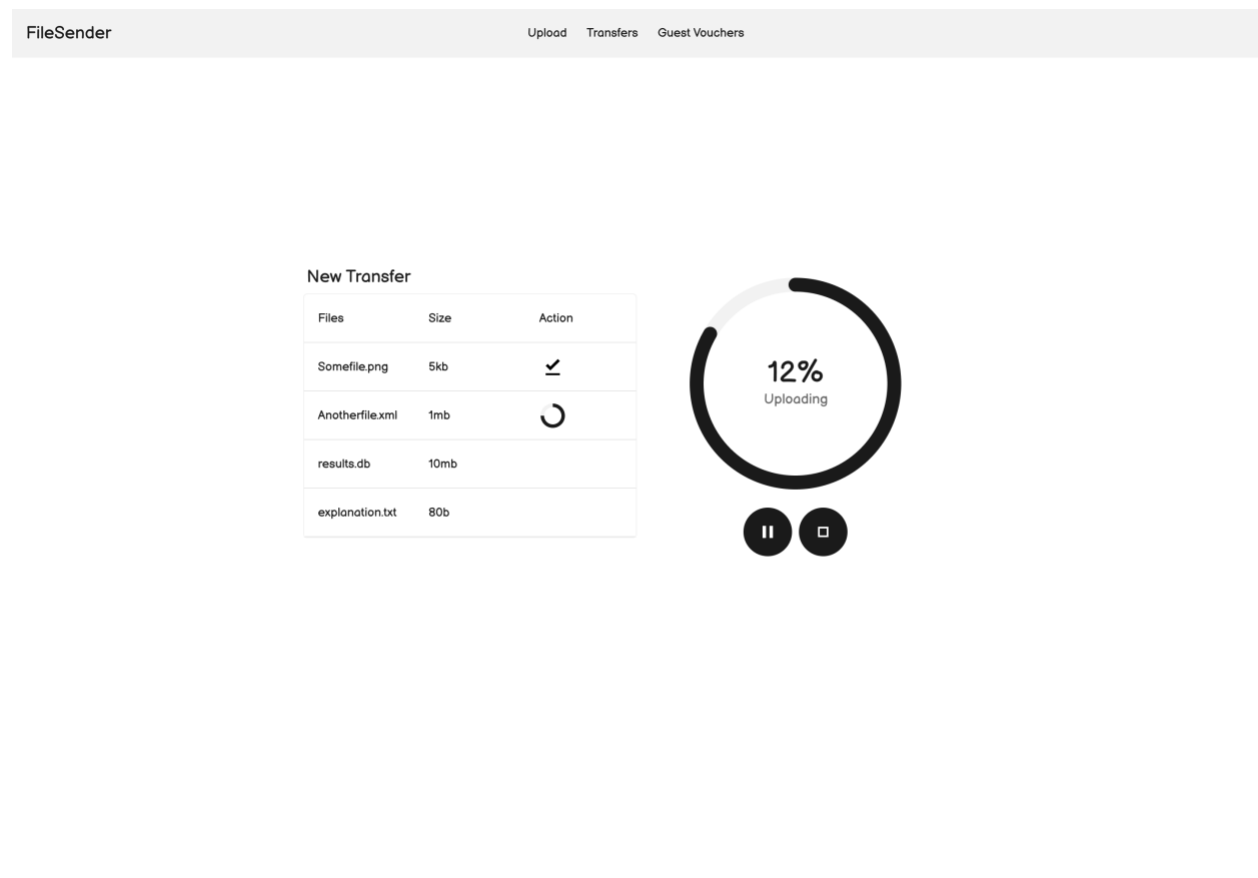


Figure 10

## 4.3 Transfers

This section provides a simple and concise overview of transfers created by the currently authenticated user. The designs for this feature are consistent for both non-JavaScript and JavaScript users.

The transfer overview displays a small list of recent transfers. Clicking on any transfer redirects the user to a detailed view of that specific transfer. The details page includes information such as the download count, list of uploaded files, and access logs.

You can explore the prototype of the transfer pages by visiting any of the upload prototypes and selecting "Transfers" from the navigation bar.

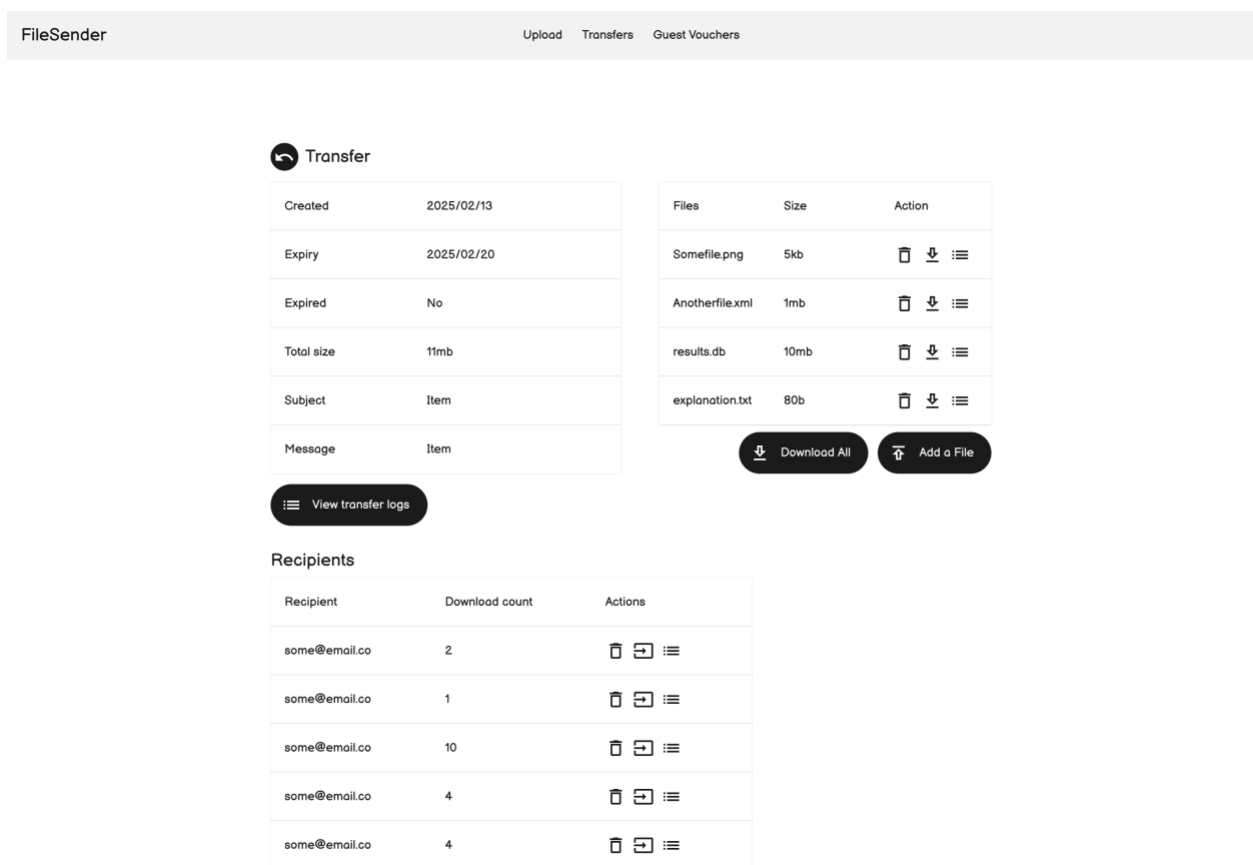


Figure 11



## 4.4 Guest Vouchers

This section features a simple, one-page design. The page includes a form allowing users to send invitations for guests to use the FileSender instance. Although the specific sending options are not detailed in the wireframe, they will be available on this page.

Additionally, there is a small table displaying guest vouchers created by the currently authenticated user. The table provides options for managing vouchers, including:

- Deleting a guest voucher
- Viewing voucher activity logs

This design offers a straightforward interface, enabling end users to easily manage guest access.

FileSender

UploadTransfersGuest Vouchers

Create Voucher

Email address

Subject (optional)

Message (optional)

Send

Options

Guest	Created	Expires	Actions
some@surf.nl	2025/02/13	2025/02/20	<div><div></div><div></div></div>
Item	Item	Item	<div><div></div><div></div></div>
Item	Item	Item	<div><div></div><div></div></div>
Item	Item	Item	<div><div></div><div></div></div>
Item	Item	Item	<div><div></div><div></div></div>

Figure 12

## 4.5 Download

The download page is where end users are redirected after clicking a link in an email or receiving a direct download link.

On this page, users can view the list of files included in the transfer. They can choose to download files individually, selecting and downloading specific files one by one. Alternatively, they can download all the files at once with a single button press.

You can view the download page prototype [here](#).

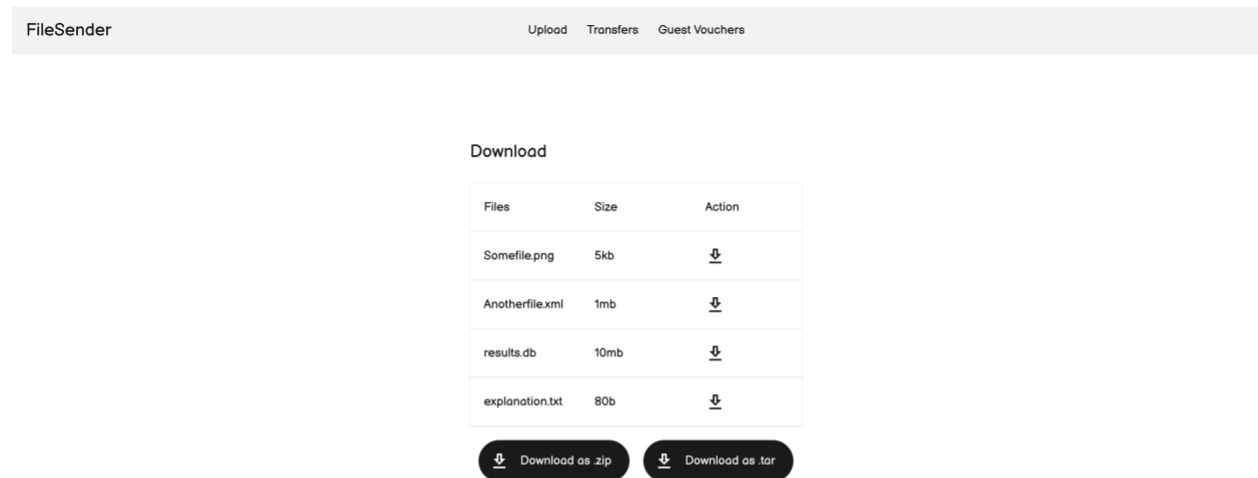


Figure 13

## 5 User Flows

With the designs in place, we can now create user flows. We will develop two distinct user flows, each with a different goal. These user flows help us visualize and map out the processes users follow to accomplish specific tasks. Additionally, within our user flows, the labels for decisions and actions correspond directly to the view names used in the wireframe designs.

### 5.1 User Flow 1

The goal of this user flow is to upload a file and then check whether someone has downloaded it. This flow outlines the steps a user follows from initiating an upload to reviewing download activity.

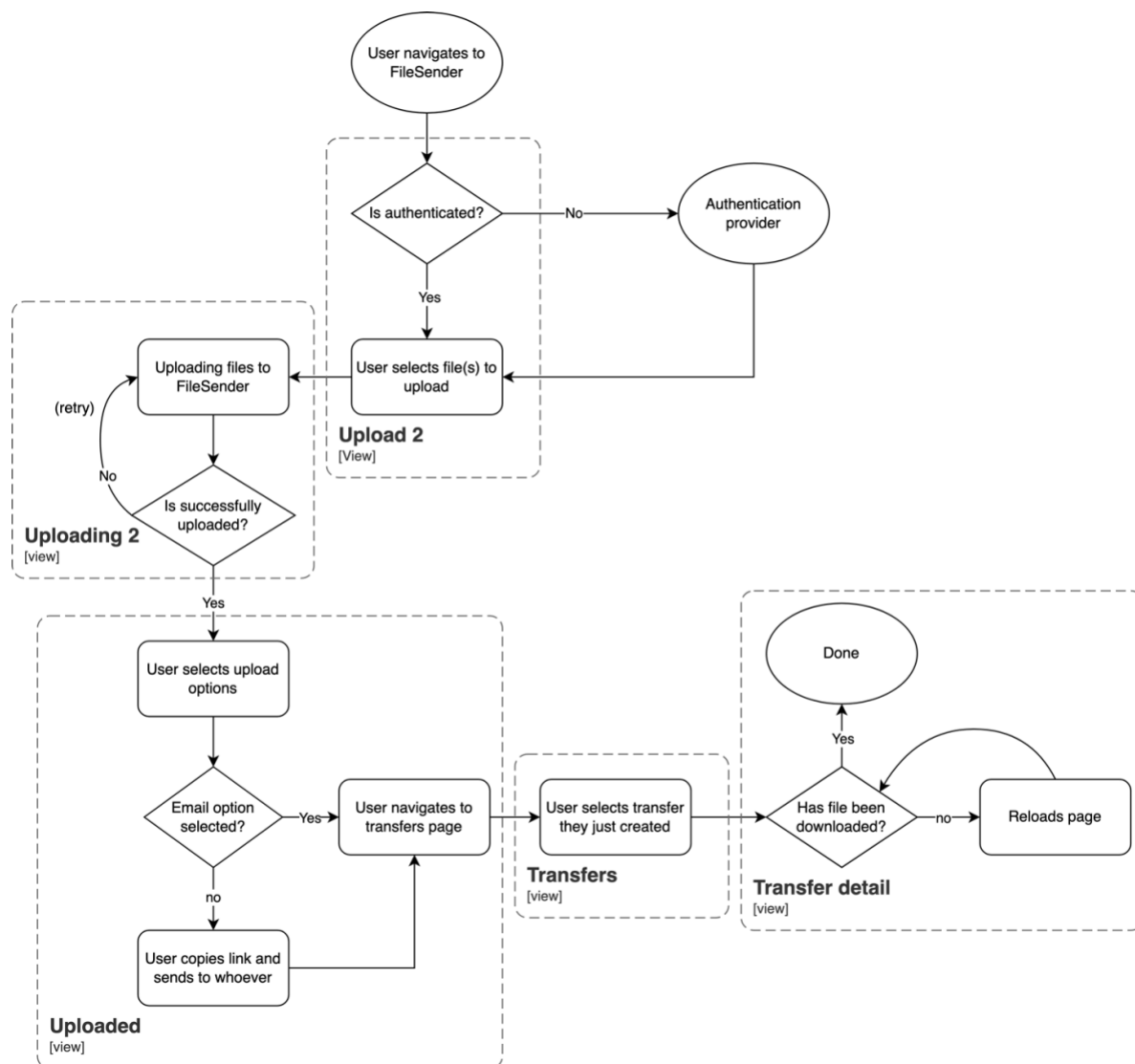


Figure 14

## 5.2 User Flow 2

The goal of the second user flow is to create a guest voucher. This flow maps the steps required to generate and manage an invitation for a guest user.

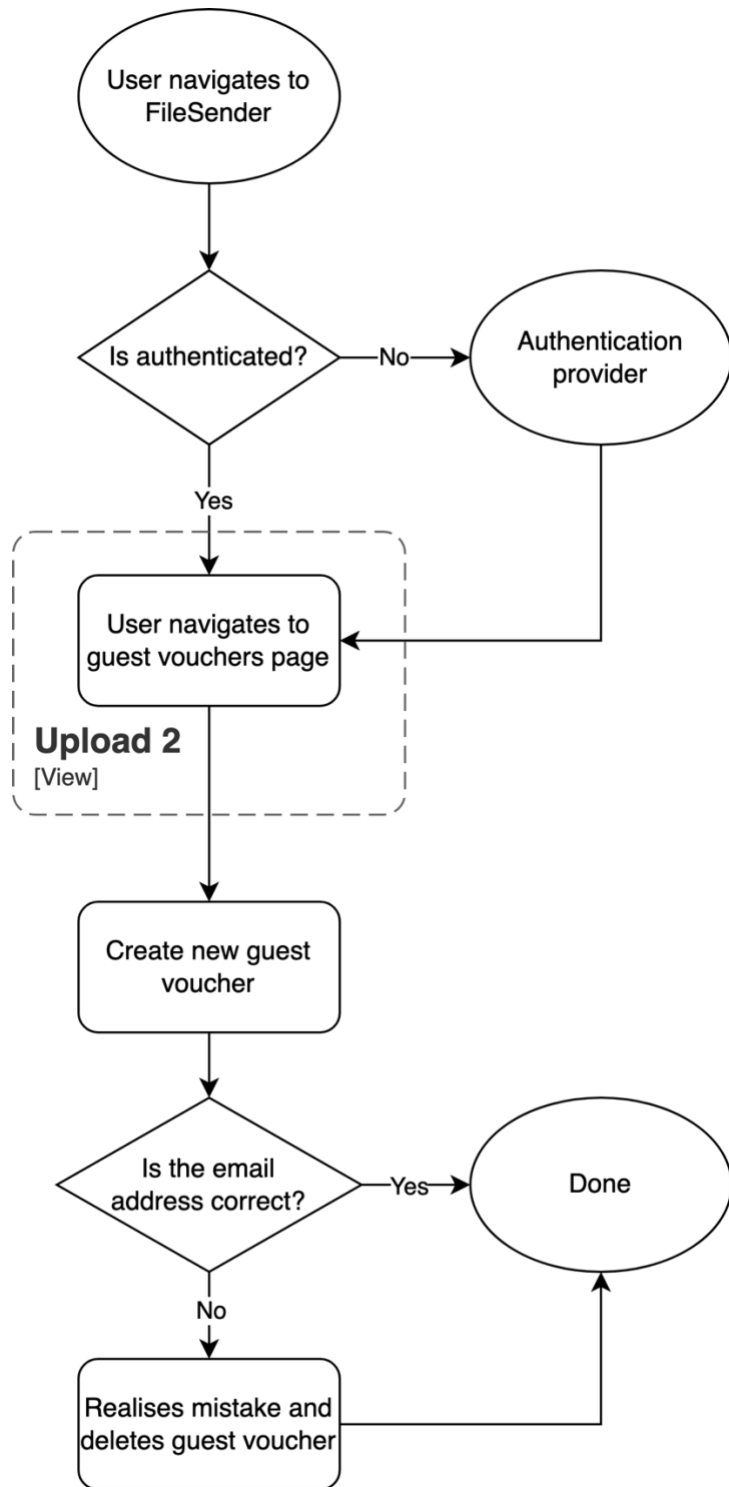


Figure 15

## 6 C4-Models

To provide an overview of the system architecture and map external services, we are using the C4 model. The C4 model consists of four levels, but for this project, we will only be designing the first two levels: the System Context Diagram and the Container Diagram. The remaining levels may be addressed during the wrap-up phase, as they require a completed application.

### 6.1 Level 1 (System Context Diagram)

Level 1 offers an overview of the system's context, highlighting its connections to external systems.

In its current design, the new FileSender application has only one external service: the email SMTP service, which is used to send email notifications. Email notifications are a future functional requirement (Should Have), and this integration is already represented within the C4 model.

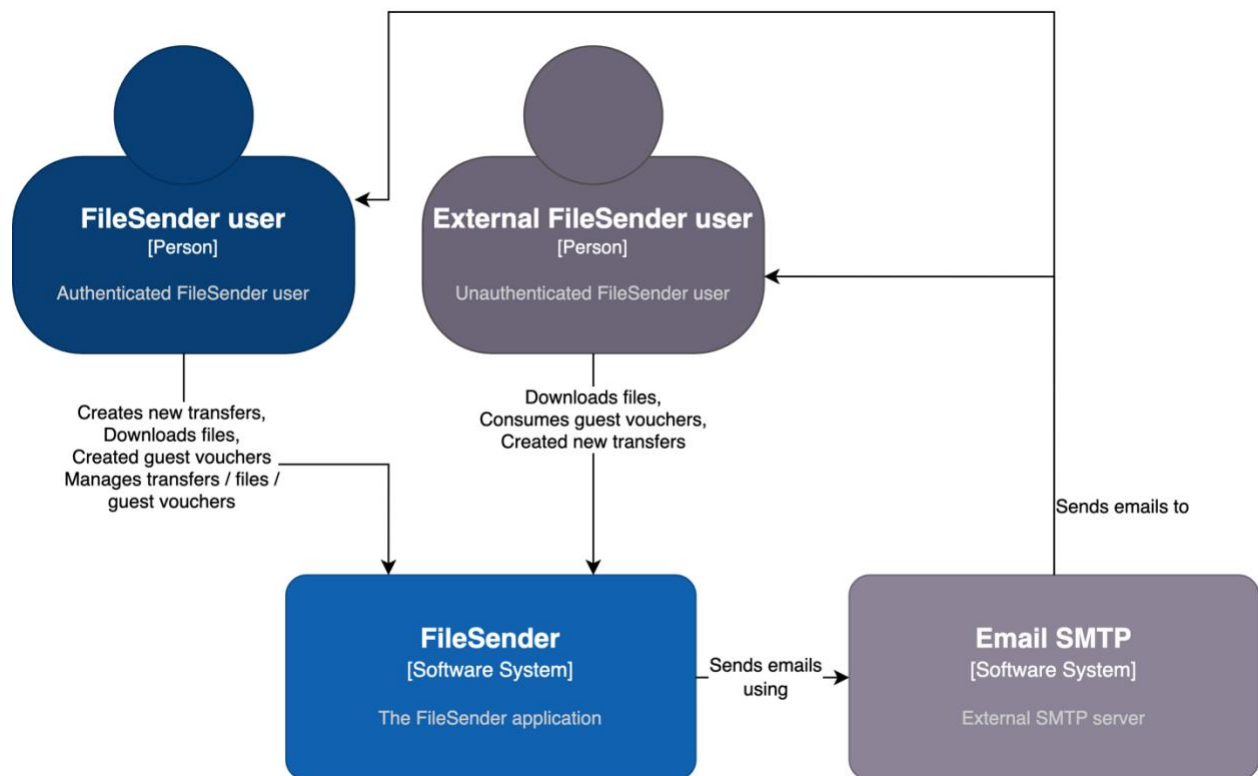


Figure 16

## 6.2 Level 2 (Container Diagram)

Level 2 provides a high-level view of the various containers within the system. It illustrates the system's architecture, showing how different components communicate with one another and clarifying the responsibilities of each component within the overall structure.

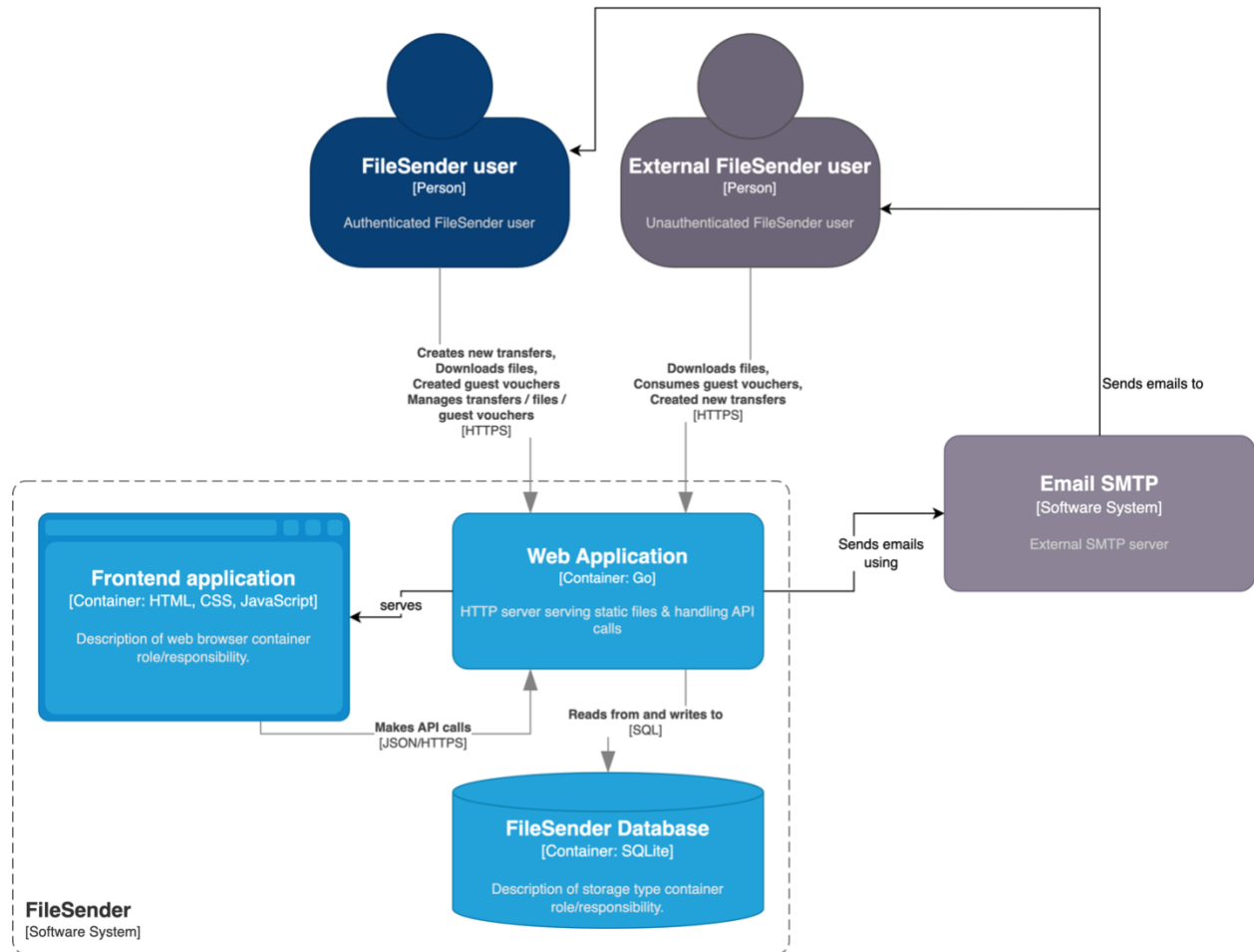


Figure 17