SES - Aspose Report

The Secure File-Sharing Server enables authenticated users to manage, share, and publish files via a React frontend and a Node.js/Express REST backend.

Instead of relying on a database to store the data structure and file permissions, we instead use unix shell to establish filesystem to store users folders and metadata.

Team - Gr03

- Duarte Cardoso, up202400037
- João Torre Pereira, up202108848
- Pedro Magalhães, up202108756
- Tomás Romão, up202400393

How to run

Note: use node version v22.13.0 or similar

To start both backends, in the project root folder:

- `docker-compose build`
- `docker-compose up`

And to start the frontend:

- `cd frontend/`
- `npm install`
- `npm run dev`

Introduction

As internal collaboration grows, our organization needed an in-house file-sharing solution balancing convenience and robust security. By using Linux containers per user (rather than a centralized database), we achieve strong isolation, simpler metadata handling, and easier rollback. This report summarizes requirements, design decisions, implementation choices (focusing on containers), and security validation, emphasizing how containerization influenced each phase.

System Requirements

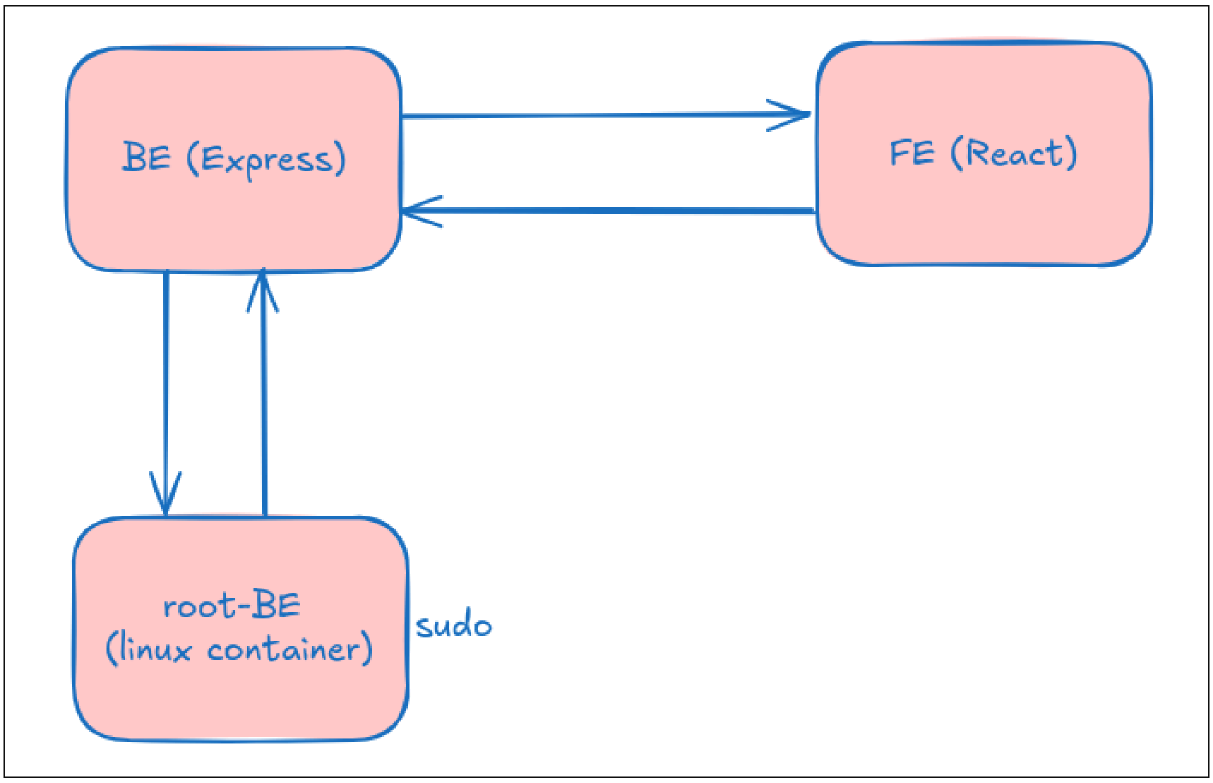
- **Functional:**
  - GUI file operations (create, delete, rename, move) in a directory tree.
  - Shell-style batch operations via a browser terminal.
  - File/directory sharing with view/edit permissions.
  - Static “public_html” for each user, served as a personal web page.
- **Non-Functional:**
  - The system should be easy to use and responsive.
  - The backend API should be RESTful and extensible.
  - The service should be reliable and available for multiple users.
- **Security:**
  - Password-based login
  - Access control: only owners or explicitly shared users can view/edit.
  - Protect against client-side attackers and malicious authenticated users; trust container runtime over a hardened host OS.

High-Level System Design

Architecture:

Three components:

1. **Frontend (React):** GUI to display user directory tree, file grid and shell emulator.
2. **Backend (Node.js/Express):** Backend with authentication responsibility as well the communicating the user GUI though a API.
3. **Root Backend (Node.js/Express):** Sudo running backend with no local communication with backend for high privileges responsibility like users creation and file permissions handling.



Both Backend and Root Backend runs on the same machine and file system, the difference is that Root Backend runs with sudo privileges. This is essential to create and manage unix users as well the file permissions. Backend opens a an external port, and the communicating with root-backend is done locally with a unix socket.

Unix based system Decision

We firstly started by using mongoDB to store our files paths with the users permissions. But this separated the permissions from the data storing and the server filesystem. There is a risk to have not updated permissions with the existing data or vice versa. We decided to get rid of the entire database, that was only responsible for storing users and theirs permissions, for a well established system in unix. With that, database dependency and all the attacks in that area are mitigated.

Unix users are created for each system user, with their corresponded home folder to store the desired files.

Because unix files permissions are basic, and do not allow a simple share logic, we use ACL to create ACE entries for the file with the users how have access to its. ACL is incredible helpful because it allow us to define which specific permissions (read, write, execute) we want for each specific user. ACL also disposes of a list with more specific permissions.

With this approach, our app is essentially a client to interact with a unix system, and we rely on the already secure unix system as our logic handler.

Shell Emulator

Our shell emulator is implemented by sending the pretending command to our API and the backend executing it in the name of the authenticated user. The response is later sent back to user. This works well, since it encapsulates the commands with the right privileges.

However, there was a UX / security trade off discussion here. Commands like `cd` work, but the user expects the next command to be executed inside the folder he went to. This is not the case in our implementation because each API call means one separate execution, so the next command runs inside the default path.

In order to increase user UX, we need to be able to create a certain threat with the user running shell. But this increases the security risks, for example another authenticated user having access to that user shell/thread. So, based on this, we decided not to implement that nice to have feature.

Note that users use a specific shell with limited binaries access to not be available to run every command.

JWT Authentication and Cookies

Since we already have no database, there is no proper way to store server sessions. So we decided to use JWT with a 24 hour expiration period. This is secure and simplifies the implementation. A user authenticates by inserting his username and password which are then used to login into that unix user. In case of the success a JWT token is created with his username and sent to the frontend to be used in authenticated only endpoints.

We started by storing the token in the browser local storage API. This is fine since the username is encoded with a server secret, but it is vulnerable to be stolen via JS attacks. Because of this, we migrated to Cookies with `httpOnly` flag and `Same Origin Policy`.

For an extra protecting layer in the registering process, we match the inserted password with a pawned password database. This is done using haveibeenpwned.com API and only the first 5 SHA-1 characters of the password encoding are sent thought the network in order to protect the source password.

React PDF Viewer package migration

Our frontend allows users to view code, images and pdf files. To implemented the pdf viewer we use <>. After the implementation and running `npm install` we discovered this package had high vulnerabilities.

Searching for a version with a fix we realized the package was impossible to maintain.

Because of that, we were forced to migrate the package and refactor some code. With this change we ended up with 0 package vulnerabilities in the 3 modules of our implementation at this date.

Lesson learned: Always check for packages know vulnerabilities and maintenance status, and choosing only the ones with the best score.

Files viewer

Files are not public to the internet and so we created an endpoint to consume the file blob only if the user is authenticated or has permission.

We created a backend route that checks if the user is logged in before allowing them to access any files. When a user tries to view a file, behind the scenes the frontend sends a request to our file loading endpoint. This endpoint verifies the authentication from the user using middleware, and then checks if the user actually owns the file or if the file has been shared with him. If the user does not have the right permissions, the server responds with an error and the file is not sent, with an error display.

On the root backend, we also make sure that users can only access files within their own home directory. We do this by resolving the file path and checking that it starts with the directory from the user, which prevents path traversal attacks. If the file exists and the user is allowed to access it, we stream the file back to the frontend as a blob. This way, the file never becomes public and is only accessible to users who are supposed to see it.

On the frontend, we built a generic file viewer component that can handle different file types, like images, videos, PDFs, and code files using already existing libraries (without know security concerns). When the user opens a file, the component fetches the file blob from the backend and displays it using the appropriate viewer.

Overall, this approach keeps user files private and secure, while still making it easy to view and interact with files through the web application.

File Sharing

Each user has files belonging to them, but we wanted to implement it so that they could share files with others existing users. As such, a permission-based system was implemented where either read only or read/write capabilities were conferred from the owner to a selected user.

The goal of this system was that the read only users can not tamper with the file so they would be only able to view or download it, but the read/write users could view, download and edit the file as the system would allow for. To implement these permissions, they need to be both managed in the ACLs and keeping track at an application level.

This feature consists of a dropdown in the UI where the owner of the file can write whoever he wants to share the file with and select the permissions (read, read/write). In this implementation, a json file was used as database to store every "share" that exists and the needed details such as who is the owner, with whom it was shared with and the permission that was granted. Doing it this way provides us with a neat and lightweight file that can be secured with permissions at the file level.

The decision behind implementing permissions at an OS level with the `setfacl` ensures more safety in the event that even if the app security is bypassed, no unauthorized file accessing may happen. Given that these permissions are persistent, they are safe from compromised code. Validation on if a user truly owns the file also happens, so that no user may share a file they do not own or even share it to a user that does not exist. Lastly, the file paths are saved at the time of sharing as is the timestamp of when a file is shared. This may be of use in the event a file is tampered with and records must be checked.

Using Helmet for HTTP security

We chose to use Helmet because it provides a simple way to secure our Express application by setting various HTTP headers. Helmet acts as a collection of middleware functions that help protect the app from some well-known web vulnerabilities by default. For example, it can prevent clickjacking by setting the X-Frame-Options header, and it helps mitigate cross-site scripting (XSS) attacks by controlling which sources are allowed to load scripts and other resources through the Content Security Policy (CSP).

One of the main advantages of Helmet is that it is easy to integrate and configure. By just adding app.use(helmet()) to our Express app, we get a solid baseline of security headers without having to manually set each one. We can also customize Helmet to fit our needs, such as allowing our frontend to embed certain pages in iframes during development by adjusting the frame-guard and CSP settings.

Helmet provides instructions in how to use different security measures in their documentation at https://helmetjs.github.io/

Viewing HTML profile content securely

Our application allows users to upload HTML files which are rendered as their own profile presentation. Naturally, we created some security measures to protect the users against well known attacks.

To let users view their HTML profile pages, we created a secure endpoint that presents the profile file only if it exists for the requested user. When someone visits a profile preview page, our frontend loads the HTML profile from the user inside an iframe, which fetches the content from our profile files endpoint. If the profile does not exist or there is an error, the user sees a clear error message.

All user profiles are public, and we provide a page where anyone can see a list of all existing usernames and view their profile pages if they want. To display a profile, we contact the backend endpoint using the username in the request. The server then looks for an HTML file with that username in the profiles directory. If the file is found, it is sent back and displayed in the browser.

For security, we use Helmet to set important HTTP headers. We configure a Content Security Policy (CSP) to only allow scripts, styles, and other resources from our own site or trusted frontend, which helps prevent XSS attacks. We also use frame-guard and CSP frame-ancestors from Helmet to control which sites can embed our pages in iframes, protecting against clickjacking.

Because of these security measures, there is no way for a script to get executed when visiting profile page from an existing user, even if a user tries to upload a malicious HTML file. The CSP blocks any external or inline scripts, so the profiles are safe to view.