



Secure File Storage

Students:

- Ido Ben Nun 209202225
- Bar Cohen 316164938

Table of Contents

Links	3
GitHub	3
Render	3
Introduction	4
Background	4
Project Design	5
<i>Problem Statement</i>	5
File Encryption and Decryption	5
User Authentication	5
File Management	5
Secure Communication	5
User Interface	5
<i>Methodology</i>	5
Implementation	6
React	6
Firebase Storage	6
AES Encryption	6
Firebase Authentication	6
Access Tokens and Refresh Tokens	7
<i>Code Structure</i>	7
Authentication Module	7
File Encryption/Decryption Module	8
File Management Module	8
User Interface	9
<i>Challenges Faced</i>	9
Key Management	9
Integration with Firebase	9
Process Overview	10
Performance Considerations	11
Results and Analysis	11
<i>Encryption and Decryption Speed</i>	11
<i>File Upload and Download Time</i>	12
<i>Security Analysis</i>	12
<i>Implications</i>	12
Improvement Suggestions	13

<i>Advanced Encryption Techniques</i>	13
<i>Multi-Factor Authentication (MFA)</i>	13
<i>Improved User Interface</i>	13
<i>Performance Optimization</i>	13
<i>Enhanced Key Management</i>	13
Conclusion	14
Declaration of Language Model Use	14
System Overview	15
Start.....	15
Successful Registration	16
Login.....	17
File Management	17
Actions	18
Testing	18
Registration	18
Login.....	18
Token management.....	18
File upload	19
File download	19
File management	19
References	20

Links

GitHub

- **Back-end:**
https://github.com/Bar1996/SFS_Back.git
- **Front-end:**
https://github.com/Bar1996/SFS_Front.git

Render

- **Back-end:**
<https://sfs-back.onrender.com/>
- **Front-end:**
<https://sfs-front.onrender.com/>

Introduction

Nowadays, the importance of secure file storage, especially in the field of cyber security, is very high. Secure file storage is not only about keeping the data safe, but also about protecting the privacy and the rights of the users, preventing unauthorized access and protecting sensitive information.

As cyber threats continue to evolve, so do the technologies used to protect against potential threats. This project focuses on developing a secure file storage system for a React application, using Firebase Storage and AES encryption for file security. Implementing such a system is a necessary step to ensure that user data remains protected from unauthorized access.

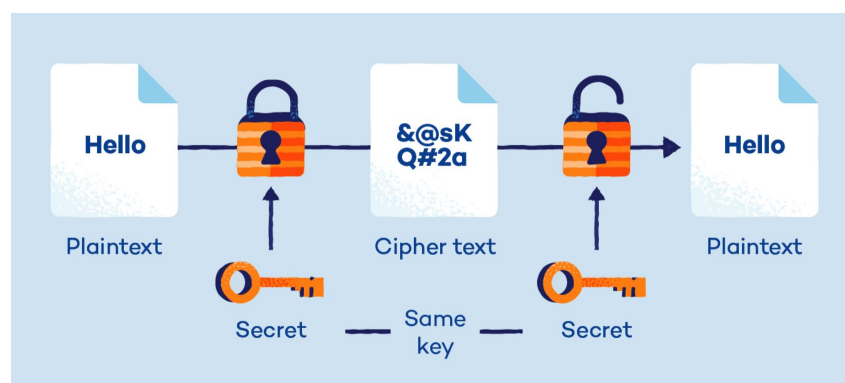
Background

The field of secure file storage has seen significant progress over the past few years, driven by the need to protect sensitive information from cyber threats.

Simple storage solutions often rely on basic encryption mechanisms or, in some cases, no encryption at all, leaving the data vulnerable to unauthorized access.

Existing solutions for secure file storage on the web usually include a combination of encryption algorithms and cloud-based storage services. For example, many modern applications use cloud storage providers such as OneDrive, Google Drive, Dropbox, etc. that offer built-in encryption to protect user data. However, these solutions act as black boxes, where users have limited control over the encryption keys and the storage process itself.

This project aims to offer a secure and user-friendly alternative by implementing the system using React, and Firebase Storage. The use of AES (Advanced Encryption Standard), a symmetric encryption algorithm widely regarded as secure and efficient, ensures that files are encrypted before storage and can only be decrypted by authorized users. This approach not only improves the security of stored data but also provides users with greater control and transparency over their files.



Project Design

Problem Statement

With the increasing reliance on web applications for day-to-day activities, the need to securely store files on browser platforms is a must. Users often store sensitive information, such as personal documents, financial records and confidential work files, on their devices.

However, many storage solutions lack strong security measures or have limited control over how files are encrypted and accessed. The main challenge this project faces is developing a secure file storage system that not only encrypts files before they are stored but also ensures that only authorized users can access and manage those files.

File Encryption and Decryption

Implement a secure file storage system using AES encryption, ensuring that all files are encrypted before being stored in Firebase Storage and can only be decrypted by authorized users.

User Authentication

Incorporate user authentication that includes access tokens, ensuring that only authenticated users (the file owner) can access their stored files.

File Management

Develop functionality that allows users to upload, download, delete and manage their files through a user-friendly interface.

Secure Communication

Implement secure communication protocols (such as HTTPS via Render) for uploading files from the web application to the cloud storage.

User Interface

Create a simple and convenient user interface for file management, making the system accessible and easy to use for all users.

Methodology

The project was developed using React, a framework that allows the development of multiplatform web applications. Firebase Storage is used for cloud-based file storage due to its scalability, ease of integration with React, and secure file system support. The encryption of the files is handled using AES (Advanced Encryption Standard), a symmetric key encryption algorithm known for its efficiency and security. User authentication is managed using Firebase Authentication, which provides secure and reliable methods for verifying user identities.

Implementation

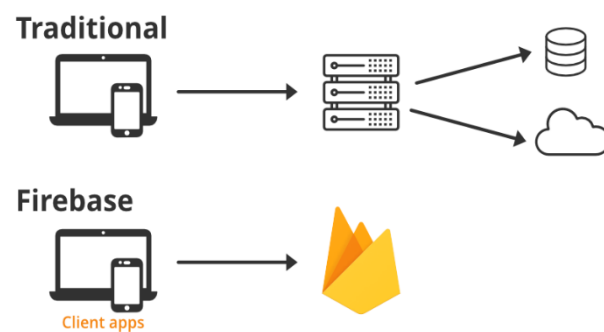
Implementing a secure file storage system includes several main components, each designed to ensure the confidentiality, integrity, and accessibility of user files. The following sections detail the technologies used, the code structure and the challenges encountered in the development process.

React

The project was developed using React, a popular framework for building web applications. React enables convenient development by enabling reuse of components in different web browsers, ensuring an efficient development process for cross-browser compatibility.

Firebase Storage

We chose Firebase Storage as the cloud-based storage solution for this project. Firebase offers strong security features, scalability and great integration with React. It also provides built-in support for file uploads, downloads and metadata management, making it a suitable choice for this project.



The traditional approach compared to Firebase approach

AES Encryption

AES-256-CBC stands for "Advanced Encryption Standard with a 256-bit key in Cipher Block Chaining (CBC) mode." It is used to encrypt files before they are stored in Firebase. AES is a symmetric encryption algorithm that is widely recognized for its security and efficiency. The encryption and decryption processes are handled on the back-end, ensuring that files are never stored insecurely on the device or in the cloud storage.

```
src > routes > JS file_route.js > decryptFile
12  const algorithm = 'aes-256-cbc';
13  const key = Buffer.from(process.env.ENCRYPTION_KEY, 'hex');
14  const ivLength = 16;
```

Firebase Authentication

Firebase Authentication is used in addition to manage user authentication and ensure that only authorized users can access files stored in Firebase Storage. Firebase authentication supports flexibility in managing user access.

Access Tokens and Refresh Tokens

The system uses token-based authentication to manage user sessions. When a user logs in, the backend generates an access token and a refresh token. The access token is a short-lived token that grants the user access to the system. This token is sent with each request to authenticate the user and ensure they have the necessary permissions to perform actions like uploading, downloading, or deleting files.

```
src > common > .js auth_middleware.js > ...
1  const jwt = require("jsonwebtoken");
2
3  const authMiddleware = async (req, res, next) => {
4    const authHeader = req.headers["authorization"];
5    const token = authHeader && authHeader.split(" ")[1];
6    if (token == null) {
7      return res.status(401).send("No token provided");
8    }
9    jwt.verify(token, process.env.TOKEN_SECRET, (err, user) => {
10     if (err) {
11       return res.status(403).send("Invalid token");
12     }
13     req.body.user = user;
14     next();
15   });
16 };
17
18 module.exports = authMiddleware;
```

Token-based authentication middleware code – back-end side

Code Structure

The back-end project consists of several main components:

Authentication Module

This module handles user registration, login, and authentication. It ensures that only authenticated users can interact with the file storage system. The authentication process is implemented using a combination of Firebase Authentication and a token-based system, which securely manages user credentials and session states.

```
src > routes > .js auth_route.js > ...
1  const express = require('express');
2  const router = express.Router();
3  const auth_controller = require('../controllers/auth_controller.js');
4
5  router.post('/signup', auth_controller.SignUpWithEmailAndPassword);
6  router.post('/login', auth_controller.LoginWithEmailAndPassword);
7  router.post('/post_email', auth_controller.PostEmail);
8  router.post('/post_password', auth_controller.PostPassword);
9  router.get('/refresh', auth_controller.refresh);
10 router.get('/logout', auth_controller.logout);
11
12 module.exports = router;
```

File Encryption/Decryption Module

This module is responsible for encrypting files before they are uploaded to Firebase Storage and decrypting them when they are downloaded. The module uses the AES encryption algorithm, with the encryption key securely managed within the application. The encryption process converts files into an encrypted format that can only be decrypted by the correct key, ensuring that the data remains confidential.

```
src > routes > js file_route.js > [⌕] upload
27   function encryptFile(buffer) {
28     const iv = crypto.randomBytes(ivLength);
29     const cipher = crypto.createCipheriv(algorithm, key, iv);
30     let encrypted = cipher.update(buffer);
31     encrypted = Buffer.concat([encrypted, cipher.final()]);
32     return { iv, encrypted };
33   }
34
35   function decryptFile(encryptedBuffer, iv) {
36     const decipher = crypto.createDecipheriv(
37       algorithm,
38       key,
39       Buffer.from(iv, "hex")
40     );
41     let decrypted = decipher.update(encryptedBuffer);
42     decrypted = Buffer.concat([decrypted, decipher.final()]);
43     return decrypted;
44   }
```

File Management Module

This module provides the core functionality for uploading, downloading, deleting, and managing files within the application. It interacts with Firebase Storage to perform these operations, ensuring that files are securely transferred and stored. The module also handles file metadata, such as file names, upload dates, and file sizes.

```
src > routes > js file_route.js > ...
46 > router.post("/upload", upload.single("file"), middleware, async (req, res) => { ...
101 > });
102 > async function downloadFileAsBuffer(url) { ...
109 > }
110 > router.get("/download/:fileName", middleware, async (req, res) => { ...
148 > });
149 > router.get("/files", middleware, async (req, res) => { ...
173 > });
174 > router.delete("/:fileName", middleware, async (req, res) => { ...
203 > });
204 > router.patch("/rename/:fileName", middleware, async (req, res) => { ...
266 > });
267 >
268 > module.exports = router;
```


User Interface

The user interface (Front-end) is designed to be simple and convenient, allowing users to easily manage their files. The interface includes screens for authentication, file browsing, file upload/download, and file management.

```
src > js Home.js > Home > newFileName
15  const Home = () => {
42  >   const handleFileChange = (e) => { ...
44  >   };
45  >   const handleUpload = async () => { ...
81  >   };
82  >   const handleDownload = async (fileName) => { ...
103 >   };
104 >   const handleDelete = async (fileName) => { ...
122 >   };
123 >   const handleRename = async (fileName) => { ...
148 >   };
149 >   const cancelRename = () => { ...
152 >   };
153 >   const formatDate = (dateString) => { ...
156 >   };
157 >   const formatSize = (size) => { ...
164 >   };
165 >   const handleLogout = async () => { ...
181 >   };
182 >   return ( ...
323 >   );
324 > };
```

Challenges Faced

Several challenges were encountered during the implementation process:

Key Management

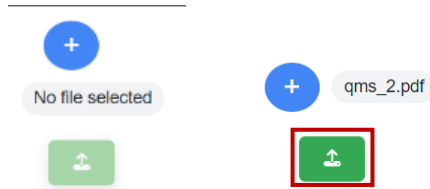
Ensuring the secure management of encryption keys was a significant challenge. The keys must be stored securely within the application and cannot be exposed to unauthorized users. This challenge was addressed by securely storing the keys in the application's secure storage module, such as the device's secure key store.

Integration with Firebase

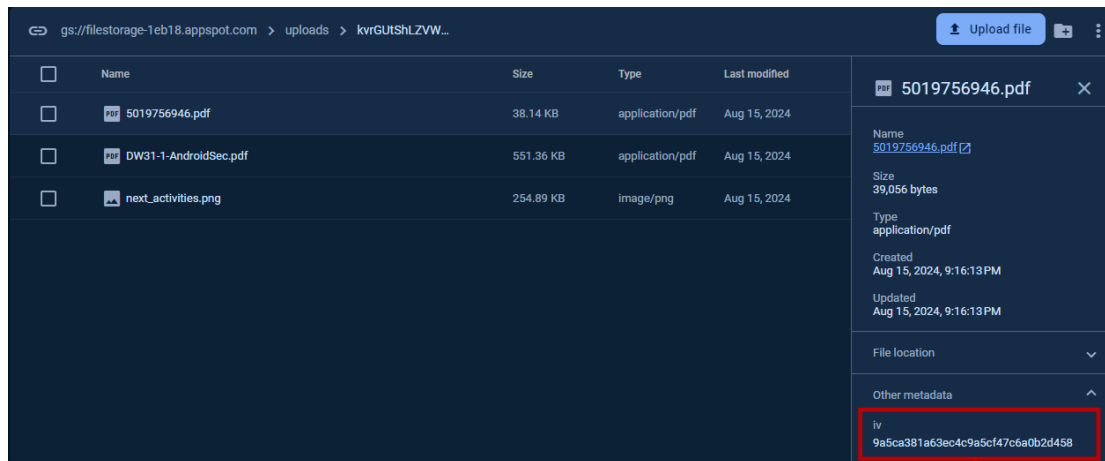
Integrating the AES encryption process with Firebase Storage required careful consideration. The files had to be encrypted before being uploaded and decrypted after being downloaded, which added complexity to the file management process. This has been addressed by simplifying the encryption/decryption logic and integrating with the Firebase file handling API.

Process Overview

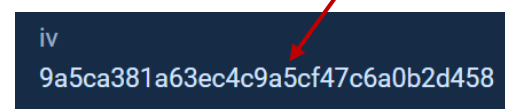
1. **File Upload:** The user selects a file to upload via the user interface in the React frontend. Once the file is selected and the upload is initiated, the file is sent to the backend server.



2. **Encryption:** In the backend, the system first generates a unique encryption key and initial vector (IV) for the file. The file is then encrypted using the AES (Advanced Encryption Standard) algorithm. AES uses both the encryption key and the IV to transform the file's contents into a secure, unreadable format.



Firebase Storage Interface

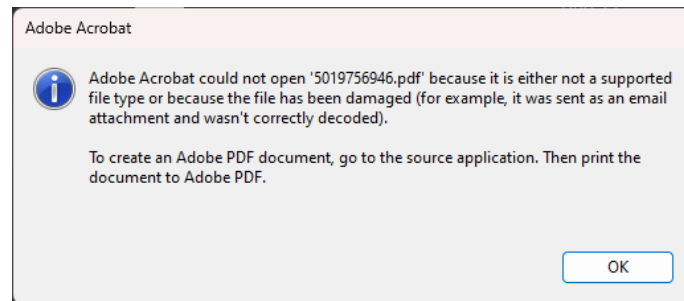


3. **Decryption:** To decrypt the file and make it readable, the user must be authenticated and connected to the system. Once authenticated, the backend handles the decryption process. The system retrieves the necessary encryption key and the IV stored securely within the system.



Download the file via the interface ensures that the user is authenticated

4. **File Download:** When a user attempts to download an encrypted file without proper authentication, he will receive an unreadable file. This is a direct result of the encryption applied to the file during the upload process. The encryption process uses a combination of an encryption key and an initial vector (IV) to encode the file's contents.



Downloading an encrypted file that cannot be opened

Performance Considerations

Encrypting and decrypting files can be complex, especially for large files. Optimizing the performance of these operations was essential to ensure a smooth user experience. This was achieved by optimizing the AES encryption process and implementing efficient in-app file handling techniques.

Results and Analysis

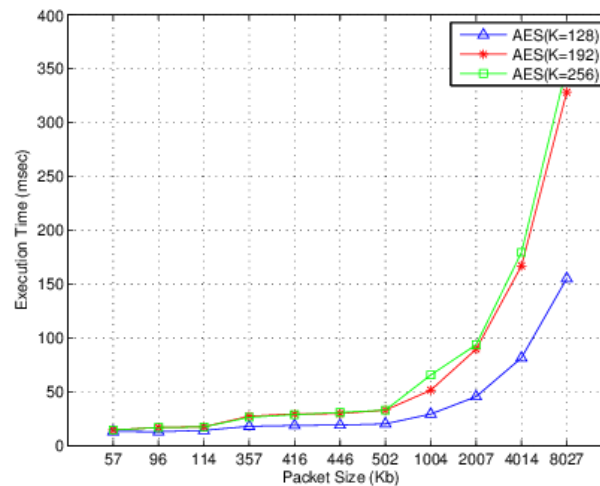
In our secure file storage system, we have considered several key factors, including encryption/decryption speed, file upload/download time, and overall security. Analyzing these results provides insights into the effectiveness of the system and its potential impact on the field of cyber security.

Encryption and Decryption Speed

One of the critical performance metrics was the speed of the AES encryption and decryption processes. The system was tested with various file sizes to determine the impact of file size on encryption and decryption times. The results showed that while AES encryption is generally efficient, the time required increases with the size of the file. For small to medium-sized files (up to 10 MB), the encryption and decryption times were negligible, typically under a few seconds. However, for larger files, the process took longer, which could impact the user experience.

File Upload and Download Time

One of the considerations was the speed of the AES encryption and decryption processes. The system was tested with different file sizes to determine the effect of file size on encryption and decryption times. The results showed that while AES encryption is generally efficient, the time required increases with file size. For small to medium files (up to 10 MB), encryption and decryption times were negligible, typically under a few seconds. However, for larger files, the process took longer, which could affect the user experience.



Execution time for different file size in AES algorithm

Security Analysis

The main goal of the system was to improve security by encrypting files before storage and ensuring that only authorized users could access them. The security of AES encryption is effective against common attacks, confirming it as a strong choice for file encryption. In addition, the use of Firebase Authentication provided a strong layer of protection, preventing unauthorized access to the stored files.

The system's architecture also included secure communication channels (HTTPS via Render) for the entire file transfer process, further ensuring that data is not intercepted during upload. The combination of encryption, secure storage, and authentication effectively reduced the risks associated with unauthorized access and data breaches.

Implications

The results prove that the secure file storage system is effective in protecting user data while maintaining a balance between security and performance. The use of AES encryption and Firebase Storage has proven to be a reliable solution for securing files on web platforms. The design and implementation of this system is inspired by the field of cyber security, especially in the context of web application development, where information security is often a significant concern.

Improvement Suggestions

While the secure file storage system performed well in its current implementation, there are several areas where improvements could be made to enhance its functionality, security, and user experience.

Advanced Encryption Techniques

While AES encryption is extremely secure, there are more advanced encryption techniques or hybrid models that combine encryption algorithms to improve security even further. For example, implementing asymmetric encryption for key exchange and symmetric encryption for data can provide an additional layer of security, especially in environments where key management is a concern.

Multi-Factor Authentication (MFA)

Integrating multi-factor authentication (MFA) into a system can significantly improve user authentication security. MFA requires users to provide two or more authentication factors, which reduces the likelihood of unauthorized access even if one factor is accepted. This can be relevant for users who store highly sensitive information.

Improved User Interface

While the current user interface works well, improving the design and adding functionality could improve the overall user experience. Integrating richer navigation, visual feedback during file operations, and customization options can make the system more user-friendly.

Performance Optimization

Although the system performed well with relatively large files, optimization of the encryption and file handling processes may further reduce processing times for extremely large files. This could include investigating more efficient encryption algorithms or optimizing the existing AES implementation to better handle larger datasets.

Enhanced Key Management

Secure key management remains a critical aspect of system security. Implementing a more sophisticated key management system by integrating with cloud-based key management services such as Google Cloud KMS offers additional security and scalability. This will allow for more secure storage and rotation of encryption keys, reducing the risk of key compromise.

Conclusion

The secure file storage system developed in this project ensures the preservation of user data. By leveraging AES encryption, Firebase Storage, and secure authentication methods, the system effectively addresses key security challenges associated with file storage on the devices using the service.

The project highlights the importance of secure file storage in the wider context of cyber security, especially as privacy measures become increasingly central to personal and professional activities. The application not only demonstrates the feasibility of building a secure storage solution within a web application but also contributes important insights into the challenges and considerations involved in such an effort.

The proposed improvements, such as advanced encryption techniques, multi-factor authentication, and improved key management, offer clear paths for future development. By addressing these areas, the system can become even stronger, offering even greater security and usability.

In conclusion, this project highlights the critical role of secure file storage in protecting user data and privacy, making a significant contribution to the field of cyber security. As the digital landscape continues to evolve, the principles presented in this project will remain essential to maintaining information in a more connected world.

Declaration of Language Model Use

For the analysis and report writing of this project, an AI language model was employed to assist in the following ways:

ChatGPT:

- Assisted in researching relevant information related to secure file storage systems and encryption methods.
- Provided suggestions for improving the report, ensuring that each section aligned with the project requirements.

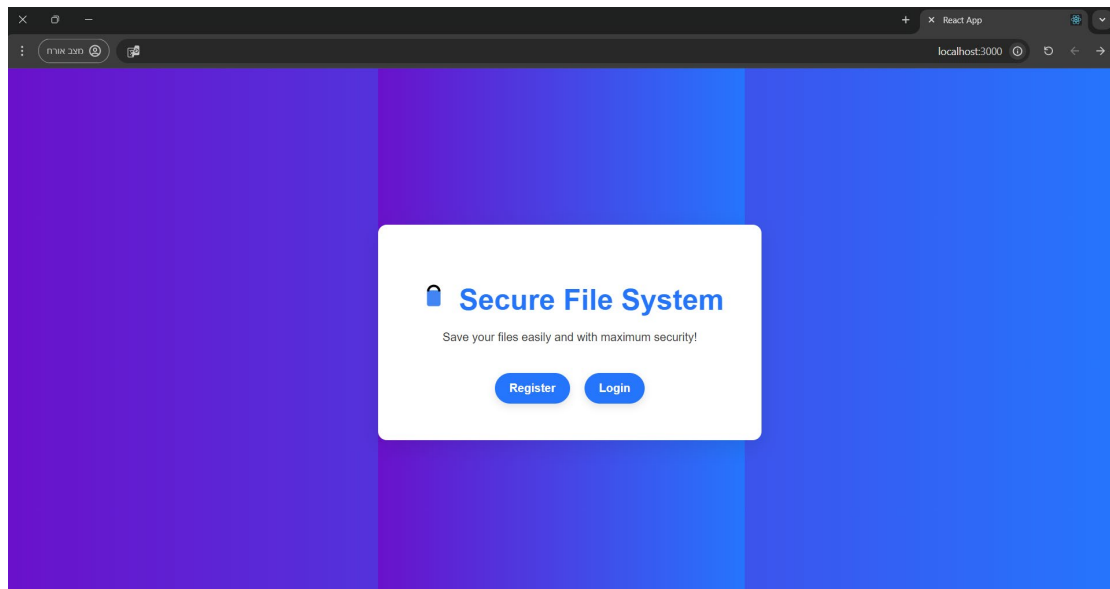
Claude:

- After the initial draft of the project was completed, Claude was used to review the content for completeness, ensuring that all critical aspects were covered according to the project guidelines.

System Overview

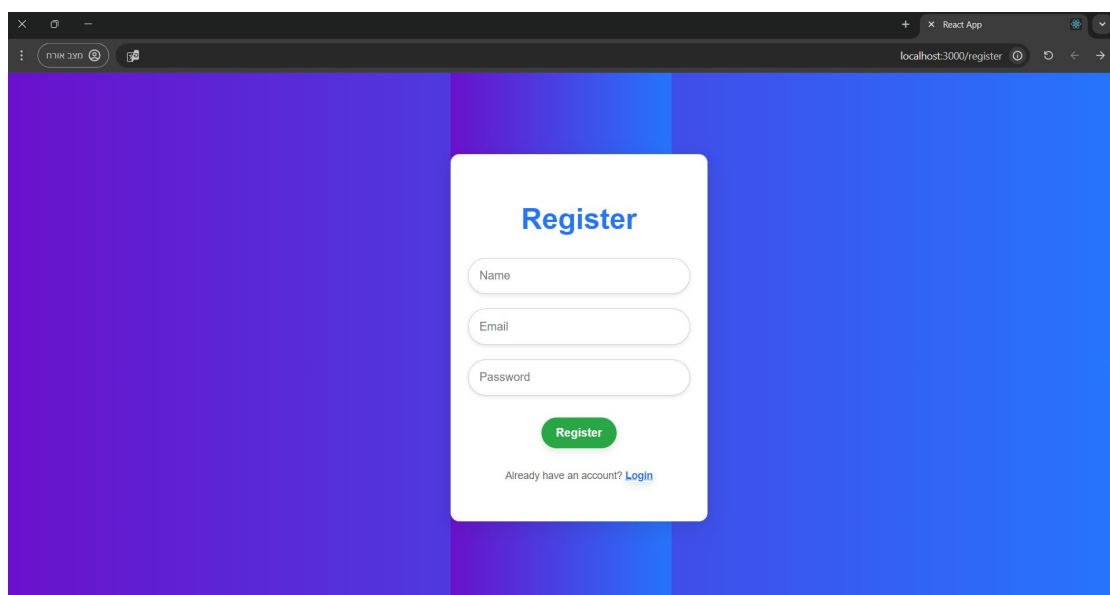
Start

The start page provides users with two main options: Register or Login. This screen introduces the secure file storage system with a modern design, inviting users to either sign up for a new account or log in to their existing account.



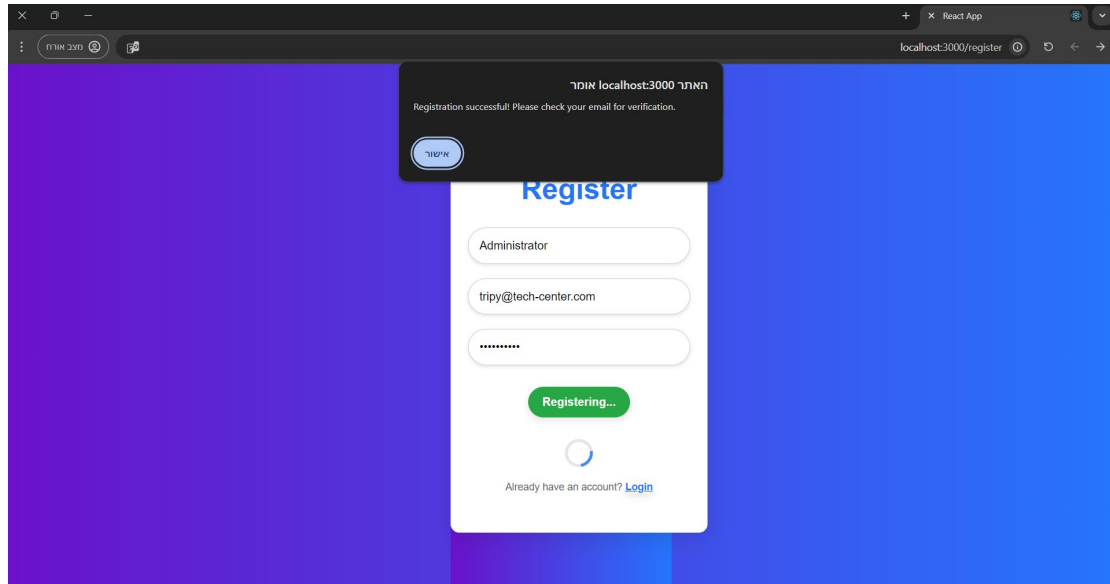
Register

The registration page allows new users to create an account by entering their name, email, and password. A clear button labeled "Register" is centered below the input fields, with an option for users who already have an account to switch to the login page.

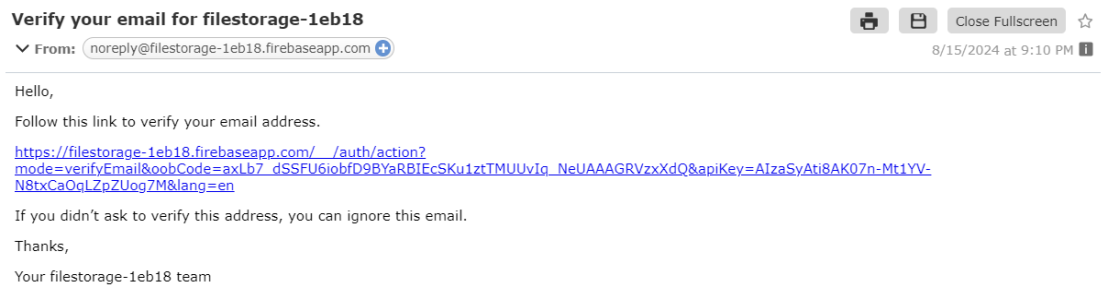


Successful Registration

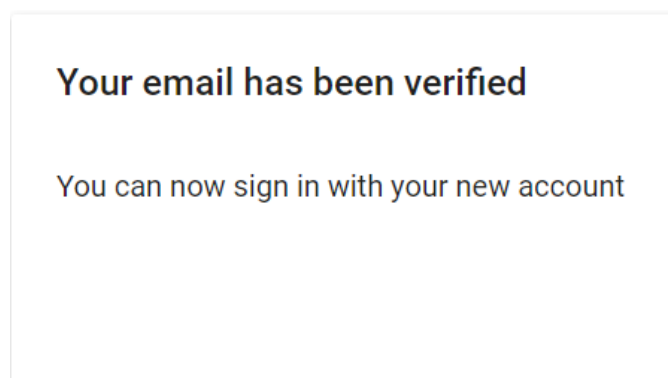
After completing the registration form and submitting the information, users are notified with a success message, asking them to check their email for verification. Afterwards, the user can login with his email and password.



- Verify Email address.

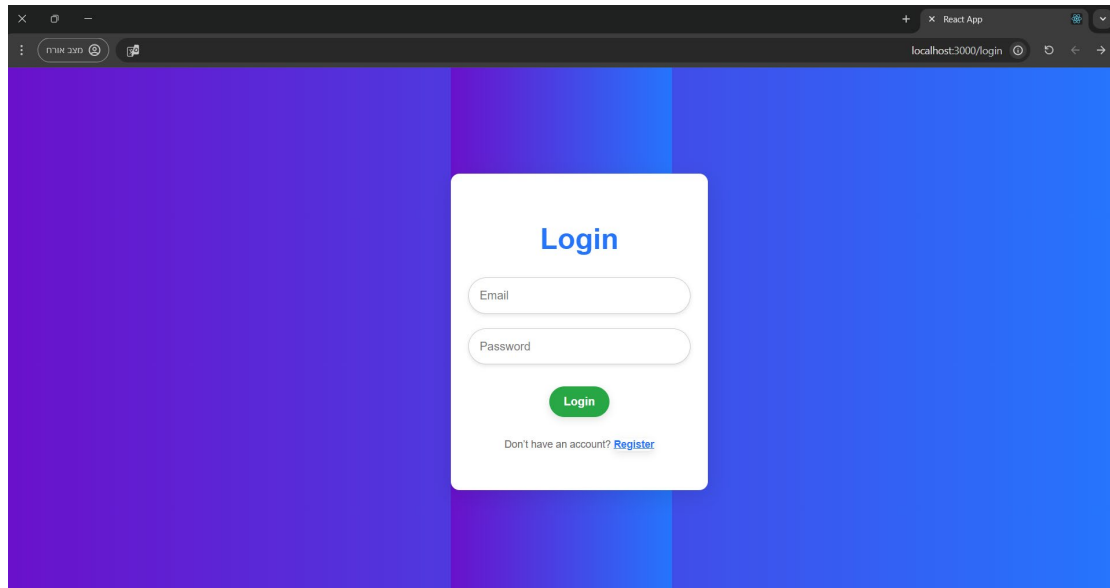


- Open the link to verify Email address.



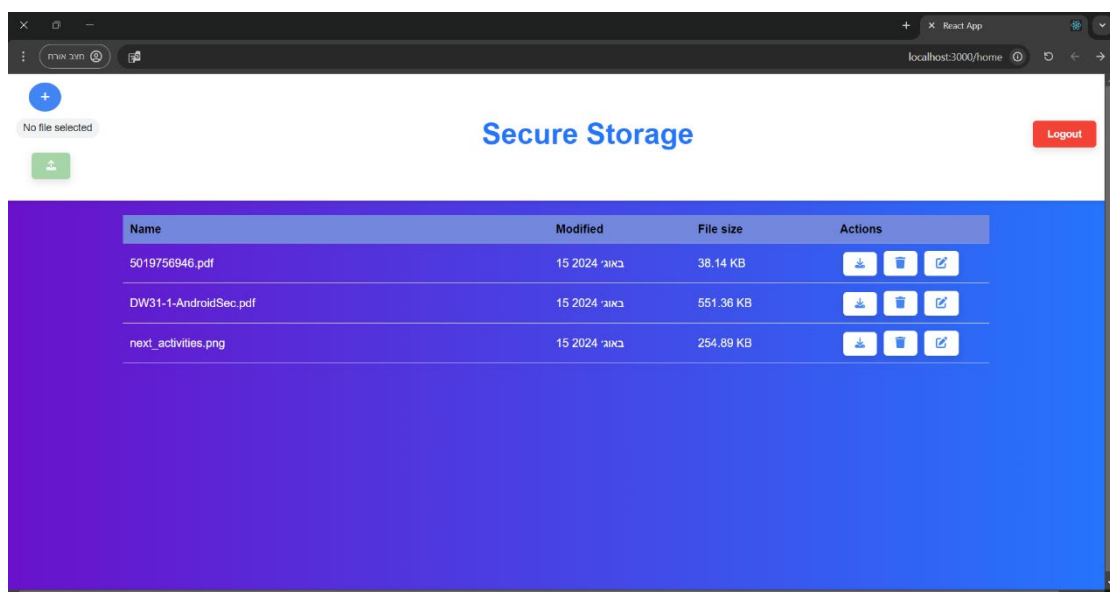
Login

The login page provides users with a form to enter their email and password to access their account. Below the form, users have the option to switch to the registration page if they don't have an account.





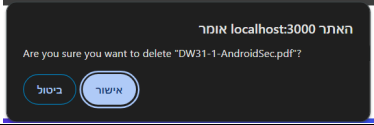


File Management

Once logged in, users are directed to the dashboard, where they can view their uploaded files. The dashboard displays file names, modification dates, file sizes, and a set of action buttons for downloading, renaming, or deleting files. The user interface is designed for easy file management, with a logout button in the top right corner for sessions management.



Actions

File Actions

	Download file	
	Delete file (after confirmation alert)	
	Rename file (after save new name)	

Testing

To ensure that the secure file storage system works properly, we conducted tests using Jest and Supertest. These tests are designed to verify features such as registration, login, file management and secure communication.

A detailed list of the tests that were performed:

Registration

Email verification: checking whether the email address is available for registration, returns a success message when the email is available.

Password submission: checking whether the system accepts the user's password during registration.

User registration: simulation of a complete user registration process by sending user information (email, password and name). The test confirmed that the user was successfully created in the Firebase database and that the email was manually verified for testing purposes.

Login

User login: simulated user login by sending the email and password to the server. The test verified that the login process worked properly by checking that access and refresh tokens were returned in response.

Token management

Access Token Refresh: checking the system's ability to refresh an expired access token using the refresh token, ensuring that the user's session remains secure and uninterrupted.

User Logout: Simulate the logout process to ensure that the user's session was successfully terminated and the system returned a confirmation message.

File upload

File upload: Check the system's ability to handle file uploads. The file was successfully encrypted and uploaded to cloud storage, and the system returned a URL to access the file.

File download

Download files: Make sure the system handles file downloads correctly. The test confirmed that the downloaded file can be decoded and will match the contents of the original file.

File management

File list: Tested the system's ability to list all files uploaded by the user. The response was verified to ensure that the list of files was accurate.

Rename File: Simulate renaming a file to confirm that the system has successfully updated the file name.

Delete file: Verified that the system can delete files from cloud storage and return a success message.

All files

83.17% Statements

262/315

57.14% Branches

32/56

96.15% Functions

25/26

83.17% Lines

262/315

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
SFS	<div><div></div></div>	100%	30/30	100%	0/0	100%	2/2	100%	30/30
SFS/src/common	<div><div></div></div>	83.33%	10/12	66.66%	4/6	100%	2/2	83.33%	10/12
SFS/src/controllers	<div><div></div></div>	74.8%	98/131	57.89%	22/38	100%	10/10	74.8%	98/131
SFS/src/helpers	<div><div></div></div>	70%	7/10	50%	1/2	100%	1/1	70%	7/10
SFS/src/routes	<div><div></div></div>	88.63%	117/132	50%	5/10	90.9%	10/11	88.63%	117/132

References

- [1] "What is AES-256-CBC?":
<https://docs.anchormydata.com/docs/what-is-aes-256-cbc>
- [2] "Comparative Analysis of AES and RSA with Other Encryption Techniques for Secure Communication":
https://www.researchgate.net/publication/379987144_Comparative_Analysis_of_AES_and_RSA_with_Other_Encryption_Techniques_for_Secure_Communication
- [3] "Client Side Encryption in Firebase Database and Storage":
<https://medium.com/hackernoon/client-side-encryption-in-firebase-database-60dd55abadb2>
- [4] "Firebase Security and Rules":
<https://firebase.google.com/docs/rules>
- [5] "Firebase Storage Documentation":
<https://firebase.google.com/docs/storage>