REPORT OF INDUSTRY ORIENTED HANDS ON EXPERIENCE (IOHE)

ON

Secure File System

submitted in partial fulfilment of the requirements for the award of degree of

BACHELOR OF ENGINEERING

In

COMPUTER SCIENCE AND ENGINEERING

Submitted By:

Name	Roll No
Priyam Aggarwal	2110991073
Shivam Kumar Pandey	2110991314
Shubham Sharma	2110991350
Hardik Jain	2110992302

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHITKARA UNIVERSITY INSTITUTE OF ENGINEERING AND TECHNOLOGY
CHITKARA UNIVERSITY, PUNJAB, INDIA

CONTENTS

Abstract

The **Secured File Share** application is a full-stack web platform built using the **MERN (MongoDB, Express.js, React.js, Node.js)** technology stack, specifically engineered to facilitate the secure, confidential, and efficient sharing of digital files between users. This solution emphasizes robust **end-to-end encryption**, ensuring that sensitive files are encrypted entirely on the sender's device before transmission and can only be decrypted on the recipient's side using a password known solely to the sender and receiver. The encryption and decryption operations are implemented using the **Web Crypto API**, leveraging modern standards such as **AES-GCM encryption**, **PBKDF2 key derivation**, and **SHA-256 hashing** for password integrity, thereby eliminating the need for the server to ever access or store unencrypted content or raw passwords.

Introduction

1.1 Background & Problem Statement

With increasing reliance on digital file exchanges, organizations and individuals face security concerns in transferring sensitive information. Conventional file transfer systems, such as FTP and cloud-based storage services, often lack efficient integrity verification mechanisms. This introduces risks like man-in-the-middle attacks, data corruption, and unauthorized modifications during file transmission.

To address the critical concerns of data integrity and confidentiality during file transmission, this project employs **end-to-end encryption using the Web Crypto API** with **AES-GCM symmetric encryption**, ensuring that file contents remain confidential from the moment they leave the sender's device until they are decrypted by the recipient. Additionally, **SHA-256 hashing and password-based key derivation (PBKDF2)** mechanisms are utilized to maintain the integrity of user-provided passwords and to prevent unauthorized access, effectively mitigating risks of data interception, tampering, and exposure during transit.

1.2 Motivation

The demand for secure file transfer solutions is growing rapidly across multiple sectors:

- Healthcare – Securely transferring patient records.
- Finance – Safeguarding transaction data.
- Legal & Government – Ensuring confidential document exchanges. A reliable, efficient, and secure transfer system, fortified by cryptographic verification, provides a valuable solution to these challenges.

1.3 Objectives

The primary objectives of this project are:

- Ensure secure data transmission through symmetric encryption (AES-GCM):
  The project guarantees confidentiality by implementing AES-GCM symmetric encryption via the Web Crypto API, allowing files to be securely encrypted on the client-side before upload and decrypted only by the intended recipient, thereby preventing data interception and unauthorized access.

- Enable integrity and access control using hashing and password-based key derivation:
  This project uses SHA-256 hashing and PBKDF2-based key derivation to validate passwords and ensure that only users with the correct decryption key can access shared files, providing both data integrity assurance and strong access control.
- Develop an intuitive interface using React.js for seamless user interactions:
  A modern and responsive frontend is built using React.js, providing users with an easy-to-navigate interface for uploading files, setting passwords, receiving download links, and decrypting received content with real-time feedback and validation.
- Implement comprehensive error handling and logging for system reliability and traceability:
  The backend, built on Express.js, integrates structured error handling and server-side logging mechanisms, enabling developers to trace issues, monitor failed attempts (like brute-force access), and maintain the overall health of the application.
- Create a scalable, modular architecture suitable for enterprise-grade deployment:
  The application is developed using the MERN stack with modular components, making it easily adaptable for enterprise use cases. Features like rate limiting, temporary file lifecycle management, and secure headers lay the foundation for a scalable and secure deployment environment.

Methodology

2.1 Approach

The **Secured File Share System** is designed using a **client-server architecture**, ensuring seamless and secure file exchanges between users. The solution is built on the **MERN stack (MongoDB, Express.js, React.js, Node.js)** and emphasizes modern web standards for **data privacy, user experience, and system resilience**. The approach includes:

- **Client-Side Encryption** – Files are encrypted using **AES-GCM** encryption through the **Web Crypto API** before being uploaded to the server, ensuring end-to-end confidentiality.
- **Password-Based Access Control** – Senders specify a password that is used to derive the encryption key. Recipients must enter the same password to decrypt the file.
- **Unique Link Generation** – Once encrypted and uploaded, a UUID-based unique download link is generated and optionally emailed to the intended recipient.
- **Secure Frontend Interface** – The React-based frontend enables users to securely interact with the platform for uploading, encrypting, downloading, and decrypting files.

2.2 Integrity & Access Verification Strategy

This system ensures file integrity and access control using cryptographic best practices:

1. **Password Derivation** – Uses **PBKDF2** (Password-Based Key Derivation Function 2) with SHA-256 to generate strong encryption keys from user passwords.
2. **Hash Validation** – Validates password correctness before decryption using SHA-256-based hash matching.
3. **Single-Use Links and Auto Deletion** – Each file has a one-time download link, and once accessed, the file is deleted from the server to ensure no residual access remains.
4. **Tamper-Resistance** – By encrypting files fully on the client and never storing plain content or passwords on the server, the system guarantees tamper-proof transmission.

2.3 Encryption Strategy (AES-GCM via Web Crypto API)

To protect file contents during transmission and storage, the system incorporates strong **symmetric encryption** using:

- **AES-GCM (Advanced Encryption Standard - Galois/Counter Mode)** – Ensures both confidentiality and data integrity with authenticated encryption.
- **Web Crypto API** – A browser-native interface used for cryptographic operations like key generation, encryption, decryption, and hashing.

- **PBKDF2 + SHA-256** – Derives secure encryption keys from user passwords using salt and iterations to prevent brute-force attacks.
- **No Key Sharing with Server** – Keys are generated and retained only on the client, enabling **zero-knowledge encryption**—even the server cannot decrypt the files.

2.4 Error Handling & Logging

To ensure system robustness and traceability, a logging and error-handling mechanism is incorporated primarily on the backend:

- **Transfer Logging** – Upload and download events are timestamped and recorded.
- **Brute Force Detection** – Rate limiting is applied to file download endpoints to prevent repeated password guessing.
- **Verification Logs** – Backend records file access attempts, including success or failure of password checks.
- **Error Notifications** – The system is configured to send appropriate user-facing messages on issues like invalid password entry, file not found, or expired links.
- **Input Validation** – Both frontend and backend validate user inputs to prevent malformed data and injection attacks.

Tools and Technologies

Frontend Technologies

1) React

- A JavaScript library for building user interfaces.
- Enables the development of a dynamic, single-page application where file encryption, password input, and download link management happen on the client-side.
- React's component-based structure allows modular and reusable UI code.

2) Tailwind CSS

- A utility-first CSS framework used to rapidly design custom user interfaces.
- It helps in making the app responsive, visually appealing, and consistent without writing traditional custom CSS files.

3) Zod

- A TypeScript-first schema declaration and validation library.
- Used for validating form inputs (e.g., password fields, file size constraints) directly on the frontend to prevent invalid data submission.

4) React–Toastify

- A notification library used for providing real-time feedback to the user.
- Displays success/error toasts for events like successful upload, decryption failure, or expired links.

Backend Technologies

1) Node.js

- A JavaScript runtime used for executing server-side code.
- Handles incoming HTTP requests, manages file storage temporarily, and routes user actions securely.

2) Express.js

- A lightweight Node.js web application framework.
- Manages backend APIs for uploading, downloading, and deleting files, while also handling middleware integration for security.

3) **express-fileupload**

- Middleware that facilitates file uploads to the server via multipart/form-data.
- Temporarily stores the encrypted files received from users for short-term sharing.

4) **express-rate-limit**

- Prevents brute-force attacks by limiting the number of requests from a single IP within a specified timeframe.
- Essential for protecting password-protected file endpoints.

5) **express-nosql-sanitizer**

- Middleware to sanitize incoming requests and protect against **NoSQL Injection** attacks.
- Ensures malicious queries are not executed on the MongoDB database.

6) **express-xss-sanitizer**

- Middleware to sanitize user inputs and prevent **Cross-Site Scripting (XSS)**.
- Protects the frontend from malicious scripts embedded in form inputs.

7) **Helmet**

- A collection of middleware functions that set **HTTP headers** for securing the app.
- Adds headers like Content-Security-Policy, X-Content-Type-Options, and Strict-Transport-Security.

8) **CORS (Cross-Origin Resource Sharing)**

- Enables secure interactions between the frontend (React) and backend (Express) hosted on different ports or domains.
- Configured to allow only trusted origins to access server resources.

Database

MongoDB (via Mongoose)

- A NoSQL document database used for storing metadata like:
  - File UUIDs
  - Expiry timestamps
  - Hashed passwords
- **Mongoose** provides a schema-based solution to model and validate data in MongoDB.

Client-Side Cryptography

Web Crypto API

Used directly in the browser for cryptographic operations like:

- **PBKDF2 (Password-Based Key Derivation Function 2)**:
  - Converts user passwords into strong cryptographic keys.
  - Uses salts and multiple iterations to protect against brute-force attacks.
- **AES-GCM (Advanced Encryption Standard – Galois/Counter Mode)**:
  - Symmetric encryption algorithm that ensures both **confidentiality** and **integrity** of the file.
  - GCM mode also verifies that the encrypted data has not been tampered with.
- **SHA-256**:
  - A secure hash function used for hashing passwords before storing or comparing, ensuring integrity and non-reversibility.

Email Service

Resend API

- A transactional email API used to send file download links securely to recipients.
- Integrates via REST API and supports secure and reliable delivery of emails containing the shared file's unique access link.

Utility Libraries

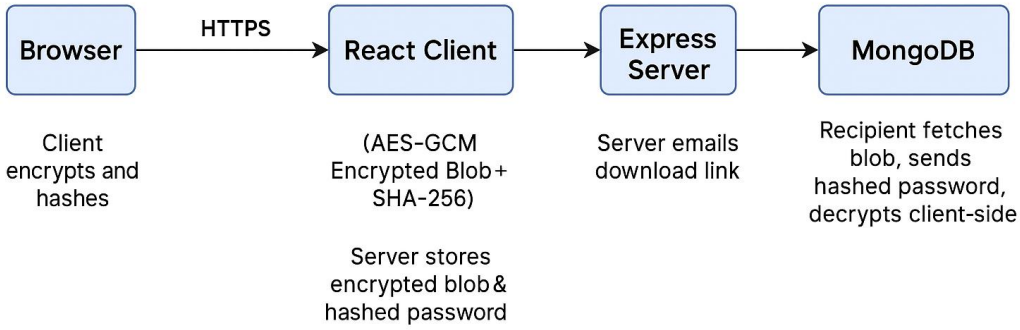
uuid

- Generates unique identifiers (UUIDs) for each file.
- Ensures that download links are unpredictable and non-repeating.

dotenv

- Manages environment-specific configuration such as API keys, MongoDB URI, and port numbers.
- Keeps sensitive data out of the codebase and into .env files, improving security and modularity.

System Architecture



The architecture showcases a secure, client-driven encryption flow where all sensitive operations like encryption, decryption, and password verification are handled **on the client side** using browser-based cryptographic APIs. The server remains "blind" to the actual contents of the file, ensuring **zero knowledge** of user data.

Components and Their Roles

1. React Client (Frontend - Browser)

- **Encryption + Hashing**:
  - Before upload, the file is split (optionally), encrypted using **AES-GCM** with a password-derived key (via **PBKDF2**).
  - The password is hashed using **SHA-256**, and only the hash is sent to the server—not the password itself.
- **Sending Data**:
  - The encrypted blob and the hashed password are sent via HTTPS to the Express backend.

- This guarantees confidentiality and integrity using **end-to-end encryption**.

---

2. Express Server (Backend)

- **Receives Encrypted File**:
  - Stores the encrypted file (blob) temporarily in a secure storage folder.
  - Saves metadata (like hashed password, file UUID, expiry timestamp) in MongoDB.
- **No Decryption Capability**:
  - It never stores or knows the original password or file content.
- **Email Link Generation**:
  - Generates a download link tied to a UUID.
  - Sends this link to the intended recipient using **Resend API** for email delivery.
- **Security Layer**:
  - Enforces input sanitization, rate limiting, secure headers, and XSS/NoSQL injection protection.
  - Ensures password brute-forcing is mitigated by account lockout strategies.

---

3. MongoDB (Database)

- Stores:
  - File metadata like file names, UUIDs, encrypted file paths, hashed password, expiry time.
  - No plaintext content or passwords are ever stored.
- Plays a critical role in tracking expiration and handling automatic deletion.

---

4. Recipient Workflow

- **Accesses Link**:
  - Opens the secure download link received via email.
- **Client-Side Password Entry**:
  - Enters the password into the frontend.
  - The password is hashed again using SHA-256 and compared with the stored hash fetched from the backend.
- **Decryption in Browser**:
  - If the hash matches, the file is decrypted client-side using AES-GCM and the derived key.
  - No sensitive data is ever sent to the server in plaintext.

---

5. Auto-Cleanup Process

- Once a file expires (based on TTL - Time to Live) or after download, the server triggers:
  - **Blob Deletion**: Removes the encrypted file from storage.
  - **Metadata Purge**: Deletes associated metadata from MongoDB.
- Ensures no file persists indefinitely and complies with security best practices for temporary storage.

Key Security Benefits

Feature	Benefit
Client-side encryption	Ensures server never sees unencrypted data
Password hashing	Protects passwords using irreversible hashing
Zero trust design	Server cannot decrypt even if breached
End-to-end confidentiality	Maintains full data privacy from upload to download
Auto-deletion	Reduces data exposure time window
Email-based delivery	Prevents open public access to files

End-to-End Flow Summary

1. **User uploads** an encrypted file via browser → encrypted with AES-GCM.
2. **Server stores** the encrypted blob and hashed password securely.
3. **Server emails** a download link using Resend API.
4. **Recipient opens** the link, enters the password, and **client decrypts** the file if the password matches.
5. After expiry/download, **server deletes** all related data.

Backend Implementation

6.1 Server Setup

File Location: Secure-File-System-main/app.js

```js
require("dotenv").config();
const express         = require("express");
const fileUpload      = require("express-fileupload");
const cors            = require("cors");
const rateLimit       = require("express-rate-limit");
const nosqlSanitizer  = require("express-nosql-sanitizer");
const { xss }         = require("express-xss-sanitizer");
const connectDB       = require("./db/connect");
const app             = express();
```

- **dotenv** loads environment vars (MONGO_URI, PORT, RESEND_API_KEY).
- **express.json/urlencoded** parse incoming JSON and form data.
- **express-fileupload** handles multipart uploads.

Middleware & Security Layers

```js
// 1. Sanitize NoSQL injections (removes $ and . in payloads)
app.use(nosqlSanitizer());

// 2. Sanitize against XSS
app.use(xss());

// 3. Rate limiter: max 100 requests per 15 minutes
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  limit: 100,
  standardHeaders: "draft-7",
  legacyHeaders: false,
});
app.use(limiter);

// 4. CORS: expose only Content-Disposition for downloads
app.use(cors({ exposedHeaders: ["Content-Disposition"] }));

// 5. File upload handler
app.use(fileUpload());
```

- **Why?** Layered middleware defends against injection, scripting, brute-force and enforces safe cross-origin behavior.

6.2 Database Connection

File: Secure-File-System-main/db/connect.js

```js
const mongoose = require("mongoose");
const connectDB = (url) => mongoose.connect(url);
module.exports = connectDB;
```

- Wraps Mongoose's connect(); used in app.js to initialize the MongoDB connection before listening.

6.3 Data Model

File: Secure-File-System-main/models/File.js

```js
const mongoose = require("mongoose");

const FileSchema = new mongoose.Schema({
  fileName:     { type: String, required: true },  // stored blob filename
  originalName: { type: String, required: true },  // user's visible filename
  path:         { type: String, required: true },  // server filesystem path
  downloadLink: { type: String, required: true },  // URL to GET endpoint
  password:     { type: String, required: true },  // SHA-256 hash from client
  extension:    { type: String, required: true },  // e.g. ".pdf", ".png"
});

module.exports = mongoose.model("File", FileSchema);
```

- **Why SHA-256?** Bcrypt was imported but not used: storing a client-side SHA-256 prevents server knowing the raw password.

6.4 File Upload Endpoint

Location: in app.js, around line forty-odd

```
app.post("/", express.json(), async (req, res) => {
  if (!req.files?.encryptedFile) {
    return res.status(400).json({ msg: "No file uploaded" });
  }
  const { encryptedFile: file } = req.files;
  const { originalName, receiverEmail, password } = req.body;

  // 1. Save blob to disk
  const filename   = `${Date.now()}_${file.name}`;
  const uploadPath = `${__dirname}/uploads/${filename}`;
  await file.mv(uploadPath);

  // 2. Prepare metadata
  const extension    = path.extname(originalName);
  const fileId       = uuidv4();
  const downloadLink = `http://localhost:${port}/download/${fileId}`;

  // 3. Save to MongoDB
  const newFile = new File({
    fileName: filename,
    originalName,
    path: uploadPath,
    downloadLink,
    extension,
    password,       // already SHA-256 hashed on client
  });
  await newFile.save();

  // 4. Email the link
  if (receiverEmail) {
    await sendEmailResend(receiverEmail, fileId);
  }

  // 5. Respond with link
  res.status(200).json({ downloadLink });
});
```

- **Step-by-step:**
    1. **Validate** the presence of encryptedFile.
    2. **Move** the in-memory upload to uploads/ folder with timestamp.
    3. **Generate** a UUID and store all metadata in MongoDB.
    4. **Email** the recipient (via Resend).
    5. **Return** the download URL in JSON.

6.5 File Download Endpoint

Location: in app.js, immediately following upload handler

```
app.get("/download/:id", async (req, res) => {
  const link = `http://localhost:${port}/download/${req.params.id}`;
  const file = await File.findOne({ downloadLink: link });
  const providedHash = req.headers["password"];

  // 1. Auth: hash must match
  if (!file || file.password !== providedHash) {
    return res.status(403).json({ msg: "Access denied" });
  }

  // 2. Stream file + set download filename
  res.setHeader(
    "Content-Disposition",
    `attachment; filename="${encodeURIComponent(file.originalName)}"`
  );
  res.download(file.path, file.originalName, async (err) => {
    if (!err) {
      // 3. Cleanup DB & disk
      await File.deleteOne({ _id: file._id });
      fs.unlink(file.path, () => console.log("Deleted after download"));
    }
  });
});
```

Errors: returns 403 if unauthorized, 500 on server error.

- **Why auto-delete?** Minimizes risk by ensuring each file is only available once and cleans up storage.

6.6 Email Notification

File: Secure-File-System-main/controllers/sendEmail.js

```
const { Resend } = require("resend");
const resend = new Resend(process.env.RESEND_API_KEY);

const sendEmailResend = async (receiverEmail, fileID) => {
  try {
    await resend.emails.send({
      from:    "onboarding@resend.dev",
      to:      receiverEmail,                    // <-- make dynamic
      subject: `Your file is ready!`,
      text:    `Download here: http://localhost:4000/download/${fileID}`,
    });
    return { success: true };
  } catch (error) {
    console.error("Email failed:", error);
    return { success: false, error: error.message };
  }
};

module.exports = sendEmailResend;
```

- **Detail:** Currently to: was hardcoded; must use the passed-in receiverEmail.

Frontend Implementation

7.1 UploadForm Component

File: client/src/components/UploadForm.jsx

Key parts:

```
// 1. Zod schemas                                                    Copy    Edit
const receiversEmailSchema = z.string().email();
const passwordSchema       = z.string().min(20);

// 2. Password & file validation
if (!receiversEmailSchema.safeParse(email).success) {
  toast.warn("Invalid email");
  return;
}
if (!passwordSchema.safeParse(pass).success || !hasUpperCase(pass)) {
  toast.warn("Password too weak");
  return;
}

// 3. Hash password
const hashPassword = async password => {
  const buffer = await crypto.subtle.digest("SHA-256", new TextEncoder().encode(password));
  return Array.from(new Uint8Array(buffer)).map(b => b.toString(16).padStart(2,"0")).join("");
};
const hashedPassword = await hashPassword(filePassword);

// 4. Encrypt file
const encryptFile = async (file, password) => {
  const salt = crypto.getRandomValues(new Uint8Array(16));
  const key  = await generateKey(password, salt);
  const iv   = crypto.getRandomValues(new Uint8Array(12));
  const data = await file.arrayBuffer();
  const encrypted = await crypto.subtle.encrypt({ name:"AES-GCM", iv }, key, data);
  return new Blob([salt, iv, new Uint8Array(encrypted)]);
};

// 5. Send to server
let formData = new FormData();
formData.append("encryptedFile", encryptedFile);
formData.append("originalName", file.name);
formData.append("receiverEmail", email);
formData.append("password", hashedPassword);

await axios.post("http://localhost:4000", formData, {
  headers: { "Content-Type": "multipart/form-data" },
  onUploadProgress: ({ loaded, total }) => progress(Math.round((loaded/total)*100))
});
```

- **generateKey** (PBKDF2):

```
async function generateKey(password, salt, keyLength=256) {
  const baseKey = await crypto.subtle.importKey("raw", new TextEncoder().encode(password), {name:"PBKDF2"}, false, ["deriveKey"]);
  return crypto.subtle.deriveKey(
    { name:"PBKDF2", hash:"SHA-256", salt, iterations:1000 },
    baseKey,
    { name:"AES-GCM", length:keyLength },
    true,
    ["encrypt","decrypt"]
  );
}
```

7.2 FileDownload Component

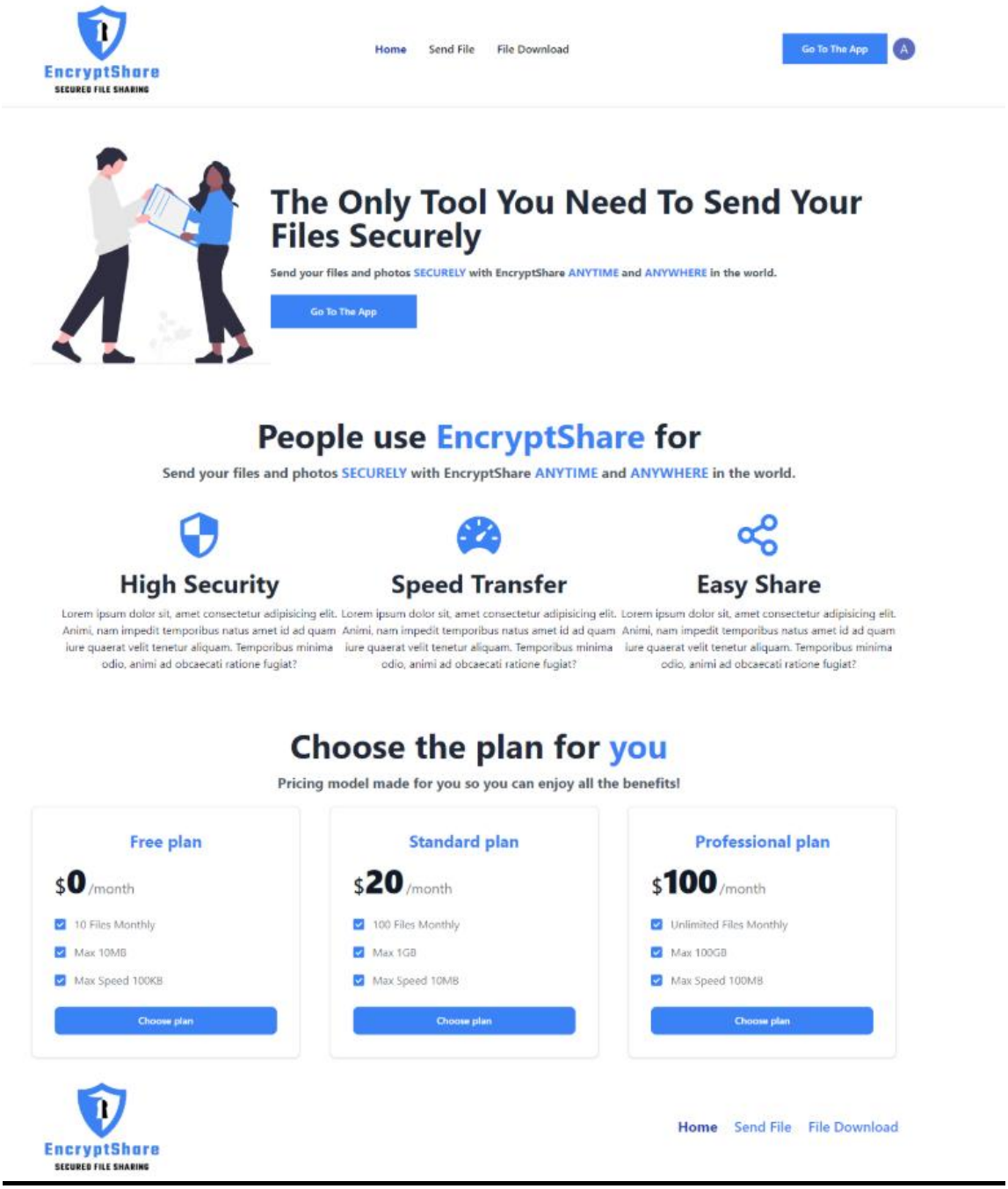
File: client/src/pages/FileDownload.jsx

```
// 1. Request encrypted blob
const response = await axios.get(`http://localhost:4000/download/${fileId}`, {
  responseType: "blob",
  headers: { Password: hashedPassword }
});

// 2. Extract salt, iv, ciphertext
const buffer = await response.data.arrayBuffer();
const salt         = new Uint8Array(buffer.slice(0,16));
const iv           = new Uint8Array(buffer.slice(16,28));
const encryptedData= buffer.slice(28);

// 3. Derive key & decrypt
const key = await generateKey(password, salt);
const decrypted = await crypto.subtle.decrypt({ name:"AES-GCM", iv}, key, encryptedData);

// 4. Trigger browser download
const blobUrl = URL.createObjectURL(new Blob([decrypted]));
const a = document.createElement("a");
a.href = blobUrl;
a.download = originalName;
document.body.appendChild(a);
a.click();
```

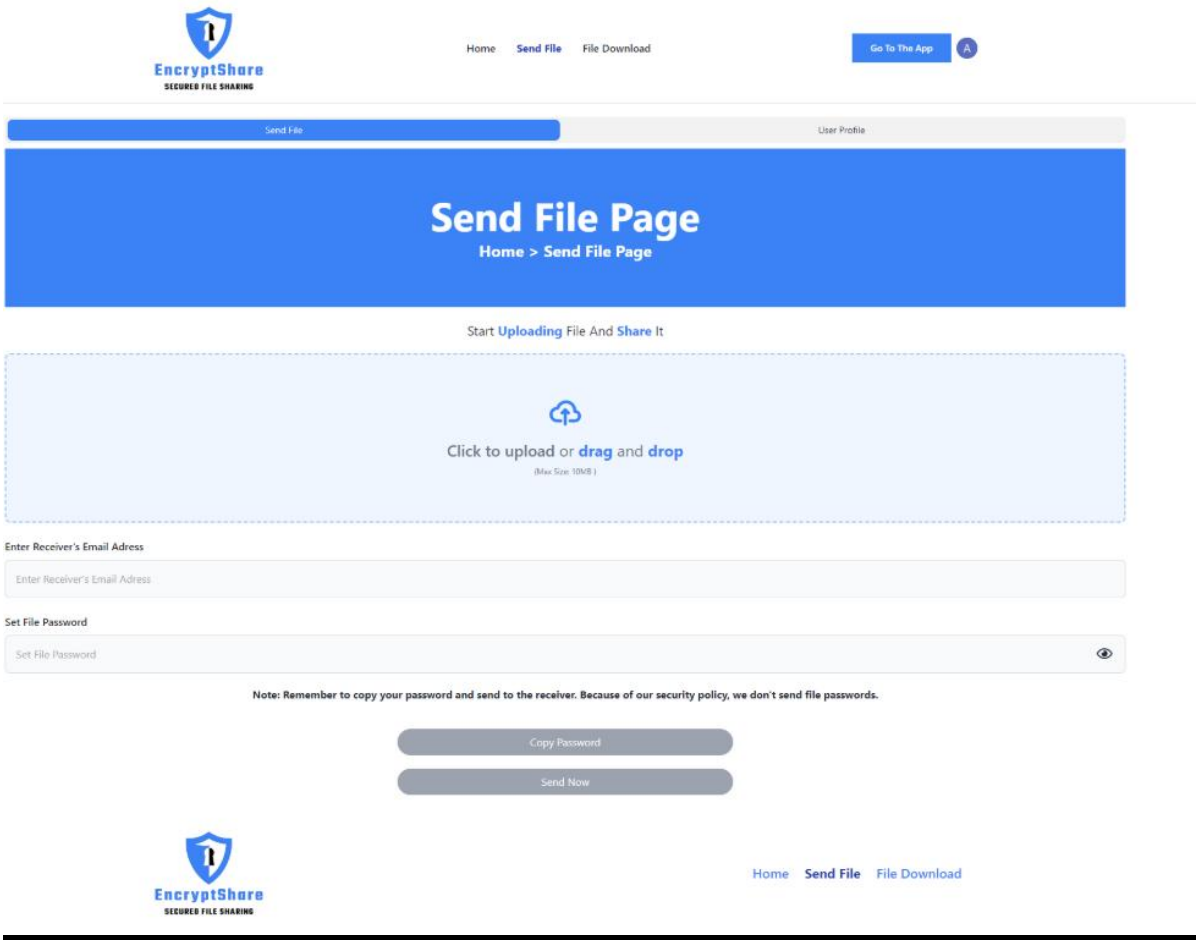
- **Why client-side decrypt?** Server never sees plaintext, achieving true end-to-end encryption.

Application interface

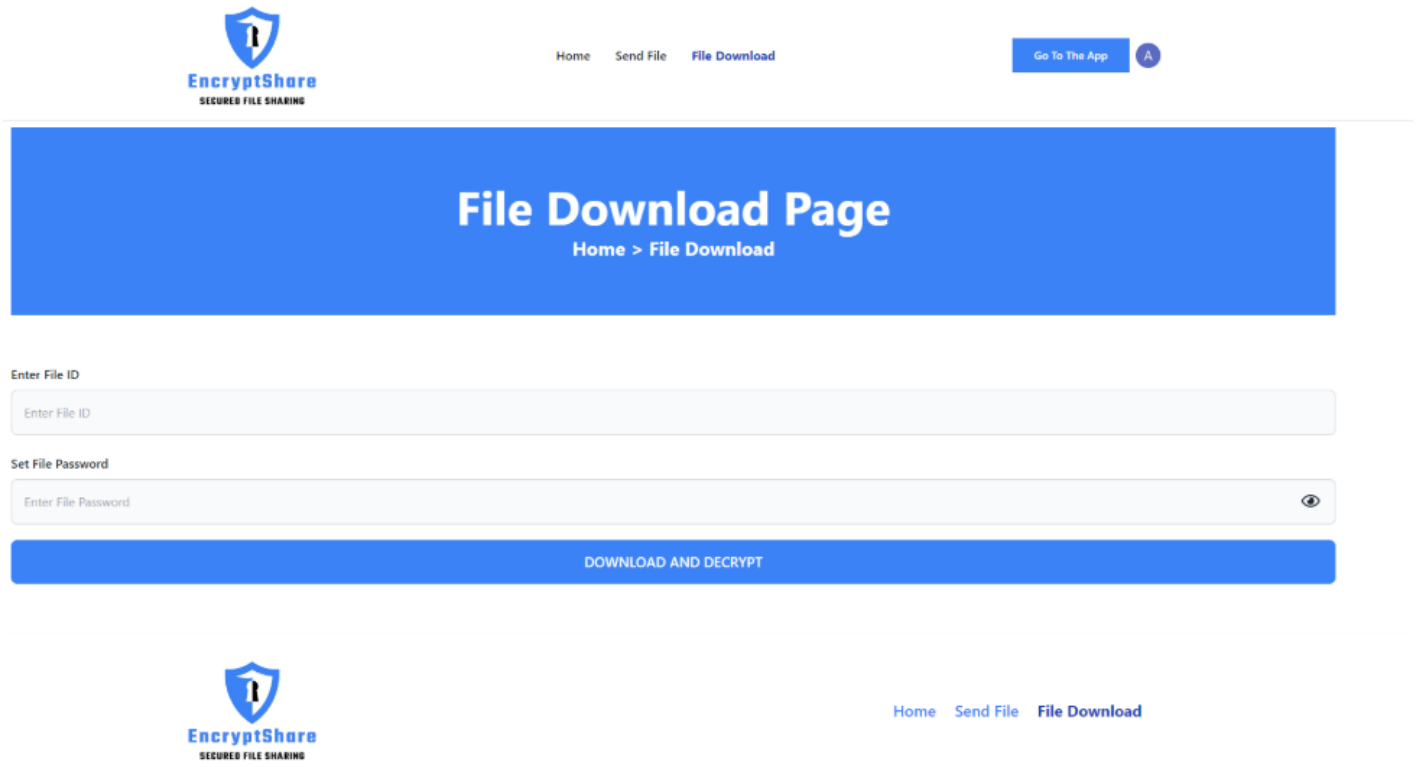
1.Home Page



2.Send File Page



3. File Download Page



Security Features

1. End-to-End Encryption (E2EE)

- Files are encrypted **on the sender's device** using the **Web Crypto API (AES-GCM)** before transmission.
- The encryption key is derived from a user-defined password using **PBKDF2 (Password-Based Key Derivation Function 2)**.
- The **decryption happens only on the recipient's browser**, ensuring that the file content is never exposed to the server, database, or transit infrastructure.
- This guarantees that even if the server is compromised, attackers **cannot decrypt files** without the password.

---

2. Password Protection & Verification (Client-Side)

- A user-supplied password is used both for encryption and as an access control mechanism.
- Instead of sending the raw password to the server, it is hashed using **SHA-256**, and **only the hash** is stored in the database.
- During download, the recipient's input password is hashed client-side and compared with the stored hash.
- This setup prevents password leaks and allows the system to remain **stateless with respect to credentials**.
- Encourages strong password practices through **policy enforcement** on length and complexity (optional extension).

---

3. Input Sanitization

- All user inputs, including file metadata, query parameters, and form fields, are sanitized using:
    - **express-nosql-sanitizer**: Defends against **NoSQL injection** by neutralizing malicious MongoDB operators like $gt, $or, etc.
    - **express-xss-sanitizer**: Cleanses inputs to prevent **Cross-Site Scripting (XSS)** attacks by removing scripts and potentially dangerous tags.
- This ensures that attackers cannot inject malicious commands or scripts to exploit frontend/backend components.

---

4. Rate Limiting & Brute-Force Protection

- **express-rate-limit** middleware is used to throttle repeated requests from the same IP address within a specific time window.
- Helps defend against:
  - **Brute-force attacks** on password-protected files.
  - **Denial-of-Service (DoS)** attacks aiming to overload the server with excessive requests.
- Optionally, account or IP-based lockouts can be introduced for enhanced protection.

---

5. Secure Headers & CORS Policies

- Utilizes **Helmet.js** to set security-related HTTP headers:
  - Prevents clickjacking, MIME-sniffing, and ensures strong content security policies.
  - Disallows unsafe script executions and mitigates cross-origin data leaks.
- **Cross-Origin Resource Sharing (CORS)** policies are configured to only allow requests from **authorized domains**, reducing exposure to cross-site attacks.

---

6. Automatic File Cleanup

- Each uploaded file is tagged with an **expiration timestamp** (e.g., after 24–48 hours or after one-time access).
- A backend cron job or interval-based cleanup routine ensures:
  - The **encrypted blob** is deleted from disk.
  - The **metadata record** is purged from MongoDB.
- This limits the lifetime of sensitive data, reduces attack surface, and aligns with **data minimization principles**.

---

7. HTTPS Enforcement (Production Environment)

- All client-server communications are expected to occur over **HTTPS**.
- During deployment, **SSL/TLS certificates** must be configured to:
  - Encrypt traffic between users and the web application.
  - Prevent eavesdropping, MITM (Man-in-the-Middle) attacks, and session hijacking.
- The app also uses HSTS (HTTP Strict Transport Security) to enforce HTTPS-only connections after the first visit.

Security-First Design Philosophy

Together, these features make **Secured File Share** a reliable tool for transmitting sensitive files without compromising:

- **Confidentiality**: via E2EE & AES encryption.
- **Integrity**: through cryptographic hashing and validation.
- **Availability**: by defending against brute-force, injection, and DoS attacks.
- **Compliance Readiness**: with principles inspired by GDPR, HIPAA, and OWASP.

Future Scope

- **Dynamic Email Recipients**

- **What:** Replace hardcoded `to:` in `sendEmail.js` with `receiverEmail` argument.
- **Why:** Enables real recipients to get the download link, fulfilling the main use case.

- **File History Page**

- **What:** Implement `/history` React page that fetches and lists past shares from server.
- **Why:** Offers auditability and lets users track what they've shared (and when).

- **Blockchain-Backed Audit Trail**

- **What:** Write each share event (fileId, timestamp, user) to a public/private blockchain.
- **Why:** Provides immutable, tamper-proof logs—crucial for compliance in regulated industries.

- **Mobile App**

- **What:** Build React Native or Flutter client to share/decrypt on mobile devices.
- **Why:** Expands accessibility; many users share files on phones.

- **Two-Factor Authentication (2FA)**

- **What:** Require an OTP (email/SMS) before upload/download.
- **Why:** Adds a second security layer, reducing the impact of stolen passwords.

- **Configurable PBKDF2 Parameters**

- **What:** Allow server-driven configuration of iteration count & key length.
- **Why:** Balances security vs. performance—higher iterations slow down attackers but increase client CPU.

Conclusion

The **Secured File Share** application demonstrates robust security best practices from client to database. By combining modern cryptography, rigorous input validation, and secure server configurations, it delivers a trustworthy platform for sensitive file exchange. With the proposed enhancements, it can evolve into an enterprise-grade solution.