Software Requirements Specification

Team : CacheME

Project : Secure File Sharing Platform

Version 5

Table of Contents

Introduction **3**
- 1.1 Purpose 4
- 1.2 Scope 4
- 1.3 Document Overview 4

User Stories and Characteristics **6**
- 2.1 User Stories & Characteristics 6
- 2.2 Use Case Diagram 10
  - Authentication and Access Control 10
  - File Access and Control 11
  - File Collaboration and Sharing 12
  - User Administration 13

Functional Requirements **14**
- Functional Requirements 14

Service Contracts **18**
- 4.1 Overview 18
- 4.2 API Gateway Service 19
  - API Specification 19
  - Routes 19
  - Data Format 19
  - Error Handling 20
- 4.3 File Service 20
  - API Specification 20
  - Endpoints 20
  - Data Format 21
  - Example Request (Send File) 21
  - Error Handling 22
- 4.4 Vault Service (Key MAnagement) 22
  - API Specification 22
  - Endpoints 22
  - Data Format 23
  - Error Handling 23
- 4.5 User Service 23
  - API Specification 23
  - Endpoints 24
  - Data Format 25
  - Error Handling 25

4.6 Cross-Service Communication 25
Protocols 25
Data Flow Examples 26
File Sharing Flow 26
Key Storage Flow 26
Key Retrieval Flow 26
Register Flow 26
Login Flow 26
4.7 Testing Requirements 26
4.8 Versioning 27
Domain Model 28
Domain Model 29
Architectural Requirements 29
6.1 Quality Requirements 31
6.2 Architectural Mapping 34
6.3 Architectural Diagram 36
6.4 Design Patterns 37
6.5 Constraints 38
Technology Requirements 40
7.1 Technology Stack Overview 41
7.2 Technology Choices 42
Deployment Model 51
Deployment Model 52
Resources 53
Resources 54

1 Introduction

This section gives a brief description to the documentation and project: Secure File Sharing Platform.

1.1 Purpose

This Software Requirements Specification (SRS) document outlines the functional and non-functional requirements for the Secure File Share platform. The platform is a cloud-based secure file sharing web application developed to ensure confidentiality, integrity, and controlled access to digital files. The platform is intended for individuals and/or organizations that require a reliable, encrypted medium for sharing documents in a privacy-focused environment.

The project has been initiated by Southern Cross Solutions. Development of the Secure Share platform is being carried out by the CacheME team.

This document serves as a foundational reference for anyone needing to understand the system's capabilities and expectations.

1.2 Scope

Secure Share will provide users with a platform to upload, encrypt, share, and manage files securely via a modern web interface. Key features include:

- End-to-end encryption
- Role-based access control
- File activity logging and audit trails
- Sharing
- User interface
- Cloud-based file storage

1.3 Document Overview

The document includes:

- Overall description
- Functional Requirements

- Non-functional Requirements
- Service Contracts
- Architectural Requirements
- Technologies

2

User Stories and Characteristics

Definition of user stories that represent key interactions that users would have with the system as well as acceptance criterias for each user story.

2.1 User Stories & Characteristics

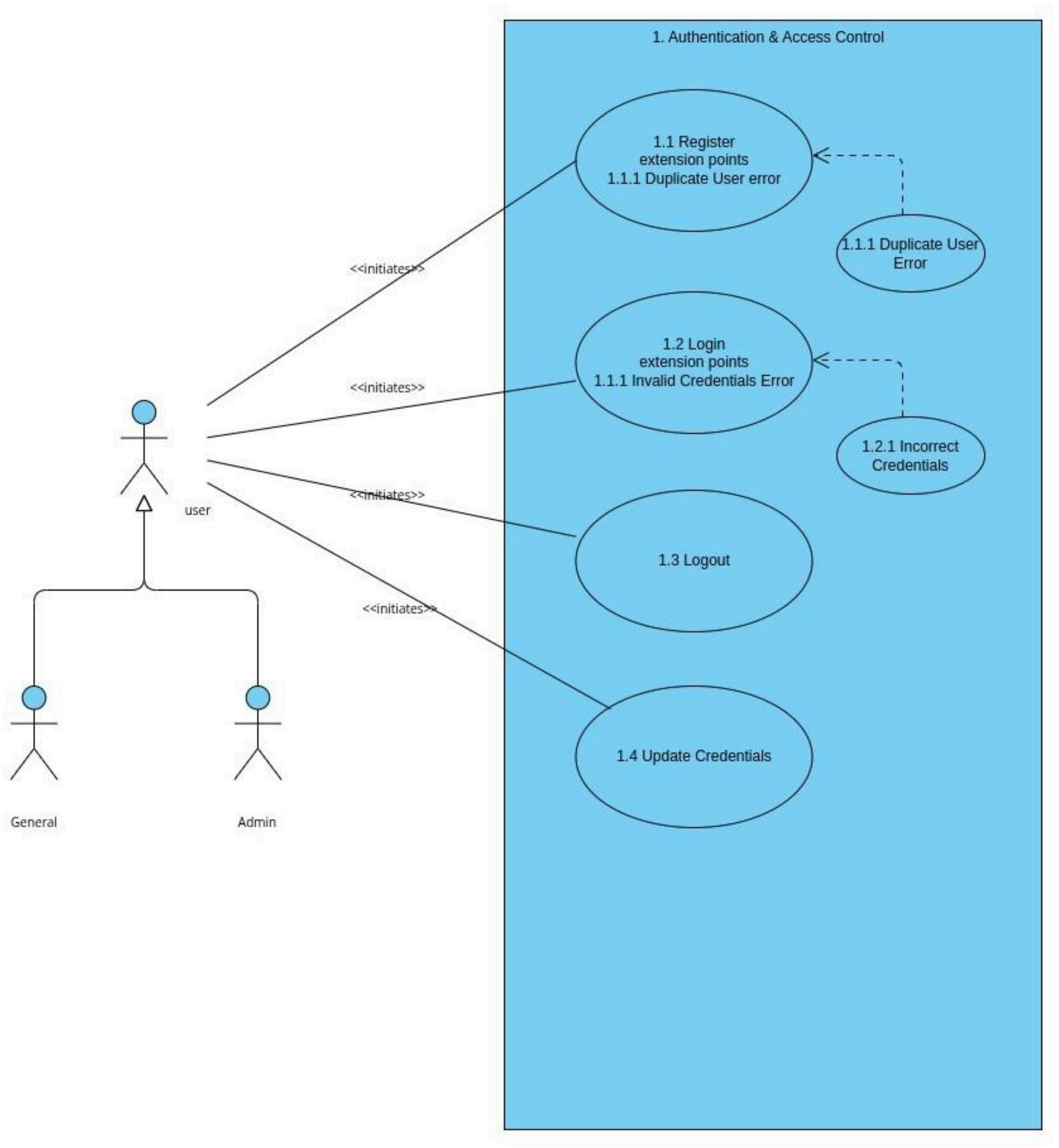
User Stories	Acceptance criteria
File Upload As a secure file sharing platform user I want to be able to securely upload my documents onto the platform so that they can be securely stored.	Given that I see the upload slot when I copy in or attach a file for upload, then the system should encrypt the file and store it securely.
File download As a secure file sharing platform user I want to be able to securely download files sent by other users to me and files I stored on the platform so that I can have the information that is needed.	Given that a user sees that files were sent to them by a peer or if they want to download previously stored files, when they click they choose the file to download, then the file should be downloaded into their system.
File sharing As a secure file sharing platform user I want to be able to securely share my files with other users so that I can send them information I need them to have.	Given that a user has the username or email of the recipient when they click the send button, then the recipient should be alerted that someone is trying to send over a file and they can either accept the file then receive the file securely or deny the request then never receive the file.
File Viewing As a secure file sharing platform user I want the ability to view files on my device before I download them or upload them so that I know that they contain information I need.	Given that a user can see all the files they have on the platform they should be able to When they click a file, Then they should be able to see the contents of the file before they download or upload it.
Access Control logs As a secure file sharing platform user I want the be able to see who downloaded my file, who I shared the file with, how many files I have uploaded or downloaded, So that I can	Given that a user is on the platform, when they press the button to access the logs, then they should be shown information about either specific files or general information about the files that have been uploaded and downloaded.

be extra sure that my files are when they belong and I have not lost any files	
Revoke file access from a user As a secure file sharing platform user I want to be able to revoke a person's access to my file. So that they do not have unlimited access to my file	Given that a user is sending a file to someone when they want to manage access to a file , then they should be prompted to remove other users.
Sent a file to multiple people As a secure file sharing platform user I want to be able to share files with multiple people, so that I don't have to send files individually one-by-one to every person	Given a user wants to share a file, When they press the send, Then they should be prompted on whether they want to send it to a single person or a group of people and whether it should be a download, or view-access.
Restore Access Logs on File Deletion As a secure file sharing platform user I want to be able to view who accessed my files even after I delete them This is for forensic or auditing reasons	Given a user has deleted a file, When they go to the deleted files tab and press on the delete file. Then they should be able to see how many people accessed that file before it was deleted.
View Notifications As a secure file sharing platform user I want to be able to view my notifications.	Given a user has new notifications , when accessing their notifications through a bell icon they should be able to view their notifications.
Delete Notifications As a secure file sharing platform user I want to be able to delete my notifications.	Given a user has notifications they want to delete, when accessing their notifications through a bell icon they should be able to delete their notifications by pressing the 'x' on the right upper-side of the notification .
Approve/Decline file shares As a secure file sharing platform user I want to be able to approve/decline file shares from other users.	Given a user wants to approve/decline a file transfer, they should be able to click on the respective button in notifications.
Delete/Restore files As a secure file sharing platform user I want to be able to delete/restore files.	Given a user wants to delete/restore a file, they would navigate to the trash page allowing them to either permanently delete file(s) or restore them. If they wish to clear

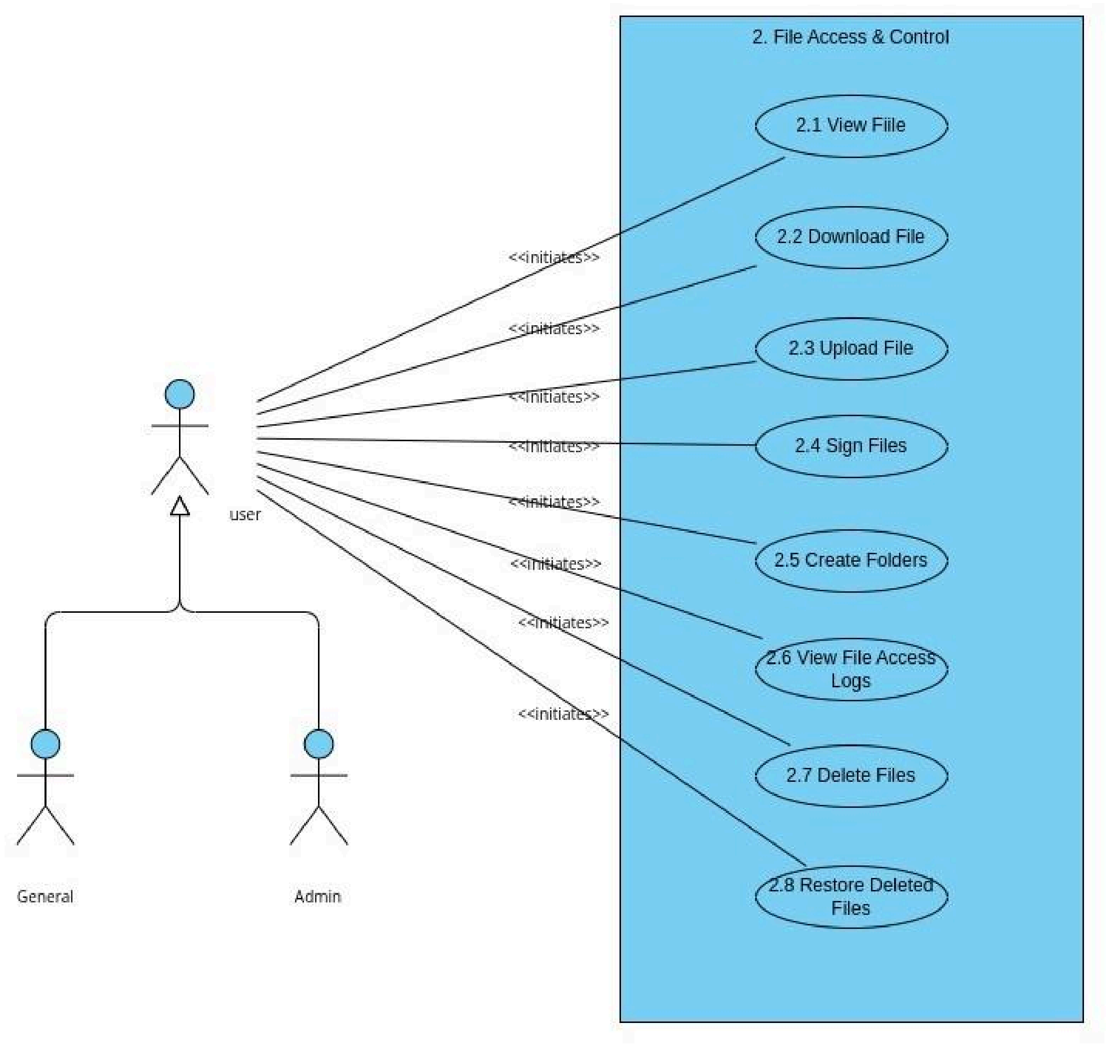
	the trash page they can do so by clearing trash.
Bulk uploads As a secure file sharing platform user I want to be able to upload multiple files at once.	Given a user wants to upload multiple files at once, they can add the files to the dialogue according to their preference (drag and drop / selection). They can then upload and view the progress bar during the upload.
Search for files As a secure file sharing platform user I want to be able to search for a specific file by file detail.	Given a user wants to search for a file, they should be able to use the search bar at the top of the page to look for their desired file by file detail.
View File Details for a specific file As a secure file sharing platform user I want to be able to view a file's details in order to ensure that it is the desired file I want.	Given a user wants to view a file's details, they should be able to use a right-click menu to get access to the file's details.
View Activity Logs for a specific file As a secure file sharing platform user I want to be able to view a file's activity details in order to keep track of activities involving my file.	Given a user wants to view a file's activity log, they should be able to use a right-click menu to get access to the file's activity log.
Forget password As a secure file sharing platform user, I want to be able to change my password due to feeling at risk, or not remembering my password.	Given a user wants to change their password, they would be prompted to provide the recovery key that was given upon registration. Their files would be re-encrypted. They would then have to wait for an email regarding their file re-encryption process.
Export Activity Logs As a secure file sharing platform user, I want to be able to export the access logs according to the filter or not from the platform.	Given a user wants to export their activity logs then can filter according to their liking and export in pdf or csv format.

2.2 Use Case Diagram

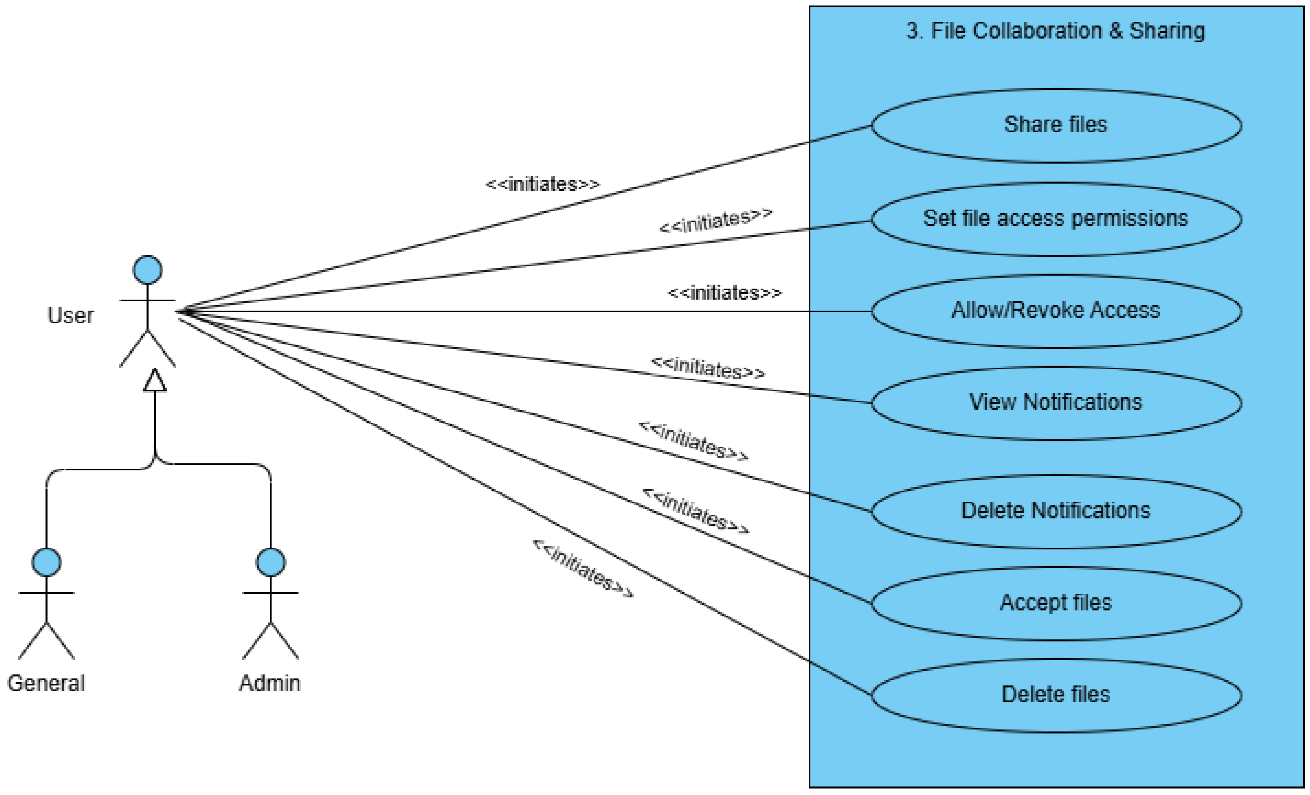
Authentication and Access Control


1. Authentication & Access Control
1.1 Register extension points 1.1.1 Duplicate User error
1.1.1 Duplicate User Error
1.2 Login extension points 1.1.1 Invalid Credentials Error
1.2.1 Incorrect Credentials
1.3 Logout
1.4 Update Credentials
<<initiates>>
<<initiates>>
<<initiates>>
<<initiates>>
user
General
Admin

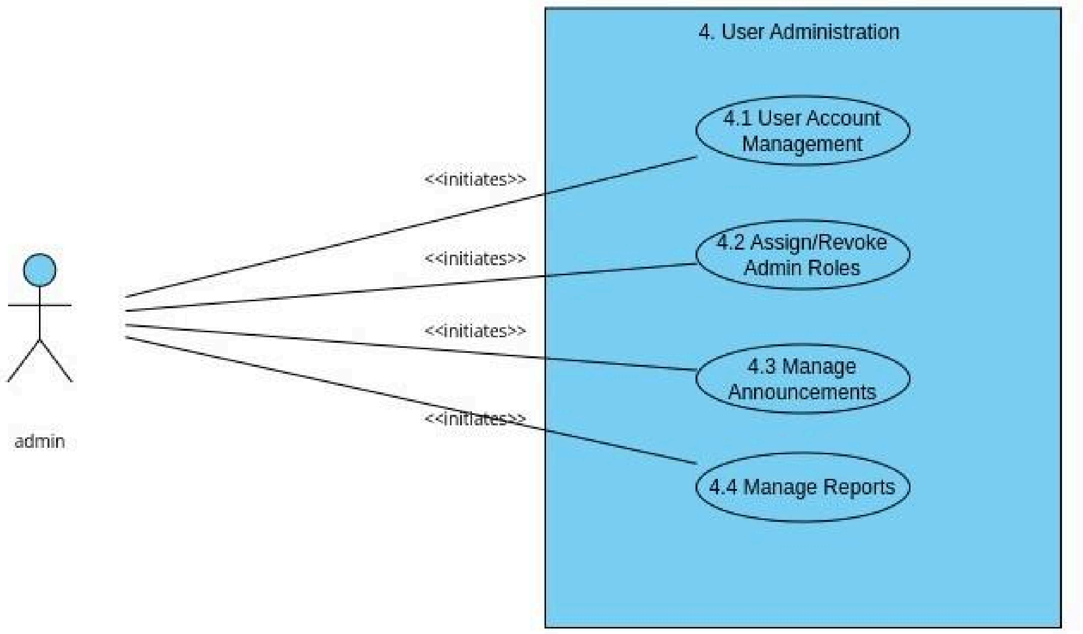
File Access and Control



File Collaboration and Sharing



User Administration



3

Functional Requirements

These are the functional requirements (what a system should do) for the Secure file sharing platform.

Functional Requirements

The Secure File Sharing platform shall (Version 4):

FR1. User Authentication & Account Management

FR1.1 Users shall be able to register an account with a secure password.

FR1.2 Users shall be able to log in using registered credentials.

FR1.3 The system shall support multi-factor authentication.

FR1.4 Users shall be able to “forget password” in order to change their password with losing files.

FR2. File Upload and Download

FR2.1 Users shall be able to upload files to the platform.

FR2.2 Users shall be able to download files from the platform.

FR2.3 The system shall display upload/download progress with cancel support.

FR3. End-to-End File Encryption

FR3.1 Files shall be encrypted on the client side before upload.

FR3.2 Files shall be decrypted on the client side after download.

FR3.3 Only the sender and intended recipient(s) shall have access to encryption keys.

FR4. File Sharing and Access Control

FR4.1 Users shall be able to share files with specific users.

FR4.2 The sender shall be able to set permissions:

- view-only (cannot download)
- Full access (view + download)

FR4.3 The sender shall be able to revoke access at any time for “view-only” files.

FR5. Digital Signing

FR5.1 The system shall be able to digitally sign files.

FR6. Access Logs and Audit Trails

FR6.1 All file-related events (upload, view, share, download) shall be logged.

FR6.2 Authorized users shall be able to view access logs.

FR6.3 Each log entry shall record time, user, and action type.

FR7. Advanced Sharing Options

FR7.1 Sharing with multiple recipients simultaneously shall be supported.

FR8. Administrative Controls

FR8.1 Administrators shall be able to monitor all users.

FR8.2 Administrators shall be able to manage user accounts:

- Remove user accounts (block | deletion)
- Restrict accounts for policy violations
- Unblock users

FR8.3 Administrators shall be able to make announcements.

FR8.4 Administrators shall be able to manage and create reports.

FR9. Notifications

FR9.1 The system should allow users to approve/disapprove notifications for transactions.

FR10. File Organization

FR10.1 Users shall be able to create and organize folders.

FR10.2 The system shall support file moving, and deletion.

FR10.3 Users shall be able to search for files.

FR10.4 The system shall support bulk file actions.

FR10.5 Users shall be able to sort their files.

FR11. Error Handling/Detection

FR11.1 The system shall display meaningful error messages for failed operations.

FR11.2 The system shall validate input fields.

FR11.3 The system shall provide confirmations for operations.

FR11.4 The system shall include a user feedback/report issue.

FR12. Key Management

FR12.1 The system shall generate key pairs for each user.

FR12.2 The system shall store public keys shall be stored in a database.

FR12.3 The system shall store private keys in a vault.

FR13. Cloud Storage Integration

FR13.1 The system shall use open-source cloud infrastructure for backend file management.

FR14. Session Management

FR14.1 The system shall support secure time-outs and auto logout for inactive users.

4

Service Contracts

4.1 Overview

This document defines the service contracts between major components of the secure file sharing platform. The contracts specify APIs, data formats, communication protocols, and error handling to ensure reliable integration between services.

4.2 API Gateway Service

API Specification

- **Base URL**: /api
- **Protocol**: REST over HTTP/HTTPS
- **Authentication**: JWT in Authorization header

NOTE: The parts lead to the service endpoints discussed in other sections.

Routes

```
/api/users/*
/api/files/*
/api/contact/*
/api/vault/*
/api/notifications/*
/api/health
```

Data Format

- **Request/Response**: JSON
- **Error Responses**:
  ```
  {
    "success": false,
    "message": "Error description",
  ```

```
  "code": "ERROR_CODE"
}
```

Error Handling

- **400**: Bad Request
- **401**: Unauthorized
- **403**: Forbidden
- **404**: Not Found
- **500**: Internal Server Error
- **Timeout**: 30 seconds

4.3 File Service

API Specification

- **Base URL**: `http://file-service:8081` (or via API Gateway)
- **Protocol**: REST over HTTP/HTTPS

Endpoints

- **Notification handling**
  - `POST /notifications` - gets all user notifications
  - `POST /notifications/markAsRead` - marks a specific notification as read
  - `POST /notifications/respond` - adds a rejected or accepted response for a file transfer.
  - `POST /notifications/clear` - clears all the user's notifications
  - `POST /notifications/add` - adds a new notification to a user.
- **File Metadata**
  - `POST /metadata` - gets metadata for a specific user
  - `POST /getFileMetadata` - gets file metadata for a specific user
  - `POST /getNumberOfFiles` - gets the number of files associated with a specific user.

- `POST /addPendingFiles` - adds a sent file into partially received (receiver can reject the file).
- `POST /getPendingFiles` - gets all the files that are partially received.
- `POST /deleteFile` - permanently deletes a file from server
- `POST /addTags` - adds tags about a file such as received etc.
- `POST /addUser` - adds a specific user to the user file sharing database.
- `POST /removeTags` - removes tags relating to specific file.

- **File Send and receive**
  - `POST /send` - Send file to recipient
  - `POST /sendByView` - Send view-only file
  - `POST /download` - downloads a specific file to user computer
  - `POST /downloadSentFile` - Download sent file
  - `POST /downloadViewFile` - Download view-only file
  - `POST /getSharedViewFiles`- gets view only files for specific user.
  - `POST /addSentFiles`- adds a sent file for specific receiver
  - `POST /getSentFiles`- gets sent files for specific receiver
  - `POST /changeMethod`- changes a view-only file to full access and vice-versa.
- **File access and revocation**
  - `POST /addAccesslog` - adds access log to a specific file
  - `POST /getAccesslog` - gets access log for specific file
  - `POST /revokeViewAccess` - revokes access to a specific file for a specific user.
  - `POST /getViewFileAccessLogs` - gets access logs for a view-only file
- **Folder Creation and upload**
  - `POST /createFolder` - Create folder
  - `POST /updateFilePath` - Move file
  - `POST /upload` - uploads a file to storage

Data Format

- **File Uploads**: Multipart/form-data
- **Other Requests**: JSON
- **Headers**:
  - `x-nonce`: Base64-encoded nonce for file encryption
  - `Content-Type`: `application/json` or `multipart/form-data`

Example Request (Send File)

```
POST /api/files/send
Content-Type: multipart/form-data

form-data:
  fileid: "file123"
  userId: "user123"
  recipientUserId: "user456"
  metadata: JSON.stringify({
    fileNonce: "base64...",
    keyNonce: "base64...",
    ikPublicKey: "base64...",
    spkPublicKey: "base64...",
    ekPublicKey: "base64...",
    opk_id: "opk123",
    encryptedAesKey: "base64...",
    signature: "base64...",
    fileHash: "base64...",
    viewOnly: false
  })
  encryptedFile: <binary data>
```

Error Handling

- **400**: Missing required fields
- **403**: Access denied (for view-only files)
- **404**: File not found
- **500**: Internal server error
- **Timeout**: 60 seconds for file operations

4.4 Vault Service (Key MAnagement)

API Specification

- **Base URL**: `http://vault-service:8443` (or via API Gateway)
- **Protocol**: REST over HTTP/HTTPS

Endpoints

- `GET /health` - Service health check

- `POST /store-key` - Store key bundle
- `GET /retrieve-key` - Retrieve key bundle
- `DELETE /delete-key` - Delete key bundle

Data Format

- **Request/Response**: JSON
- **Key Bundle Structure**:
```
{
  "encrypted_id": "user123",
  "spk_private_key": "base64...",
  "ik_private_key": "base64...",
  "opks_private": [
    {"opk_id": "opk1", "private_key": "base64..."},
    ...
  ]
}
```

Error Handling

- **400**: Invalid key bundle
- **404**: Key not found
- **500**: Vault operation failed
- **Timeout**: 10 seconds

4.5 User Service

API Specification

- **Base URL**: `http://user-service:3000` (or via API Gateway)
- **Protocol**: REST over HTTP/HTTPS

Endpoints

Method	Endpoint	Auth Required	Description
POST	/register	No	Register a new user
POST	/login	No	User login
POST	/logout	Yes	User logout
GET	/profile	Yes	Get user profile
DELETE	/profile	Yes	Delete user profile
POST	/token_refresh	Yes	Refresh authentication token
PUT	/profile	Yes	Update user profile
POST	/verify-password	Yes	Verify user password
POST	/send-reset-pin	Yes	Send password reset PIN
POST	/change-password	Yes	Change user password
GET	/public-keys/:userId	Yes	Get public keys for a user
GET	/getUserId/:email	Yes	Get user ID from email
POST	/get-token	Yes	Get user token
GET	/token-info	Yes	Get user info from token
GET	/notifications	Yes	Get notification settings
PUT	/notifications	Yes	Update notification settings
POST	/avatar-url	Yes	Update avatar URL

Data Format

- **Request/Response**: JSON
- **Registration Example**:

```
{
  "username": "user1",
  "email": "user@example.com",
  "password": "securepassword",
  "ik_public": "base64...",
  "spk_public": "base64...",
  "opks_public": ["base64..."],
  "nonce": "base64...",
  "signedPrekeySignature": "base64...",
  "salt": "base64...",
  "ik_private_key": "base64...",
  "spk_private_key": "base64...",
  "opks_private": ["base64..."]
}
```

Error Handling

- **400**: Missing required fields
- **409**: User already exists
- **500**: Database operation failed
- **Timeout**: 15 seconds

4.6 Cross-Service Communication

Protocols

- **Primary**: REST (JSON)
- **Alternative**: gRPC (for performance-critical paths)

Data Flow Examples

File Sharing Flow

1. UI -> API Gateway: `POST /api/files/download`
2. API Gateway -> File Service: Download encrypted file
3. UI -> API Gateway: `GET /api/users/public-keys/{recipientId}`
4. UI -> API Gateway: `POST /api/files/send` (with encrypted payload)
5. API Gateway -> File Service: Store sent file
6. File Service -> Metadata DB: Record transaction

Key Storage Flow

1. UI -> API Gateway: `POST /api/vault/store-key`
2. API Gateway -> Vault Service: stores key bundle
3. Vault Service -> HashiCorp Vault: stores secrets

Key Retrieval Flow

1. UI -> API Gateway: `GET /api/vault/retrieve-key`
2. API Gateway -> Vault Service: Retrieve key bundle
3. Vault Service -> HashiCorp Vault: Get secrets

Register Flow

1. UI -> API Gateway: `POST /api/users/register`
2. API Gateway -> Supabase: stores the credentials
3. Then Key Storage Flow

Login Flow

1. UI -> API Gateway: `POST /api/users/register`
2. API Gateway -> Supabase: retrieves the credentials
3. Then Key Retrieval Flow

4.7 Testing Requirements

Each service contract must be verified with:

1. Unit tests for individual endpoints
2. Integration tests for cross-service workflows
3. Performance tests for file operations

4. Security tests for encryption/decryption flows

Test cases should verify:

- Correct data formats
- Proper error handling
- Authentication/authorization
- Encryption/decryption correctness
- Performance under load

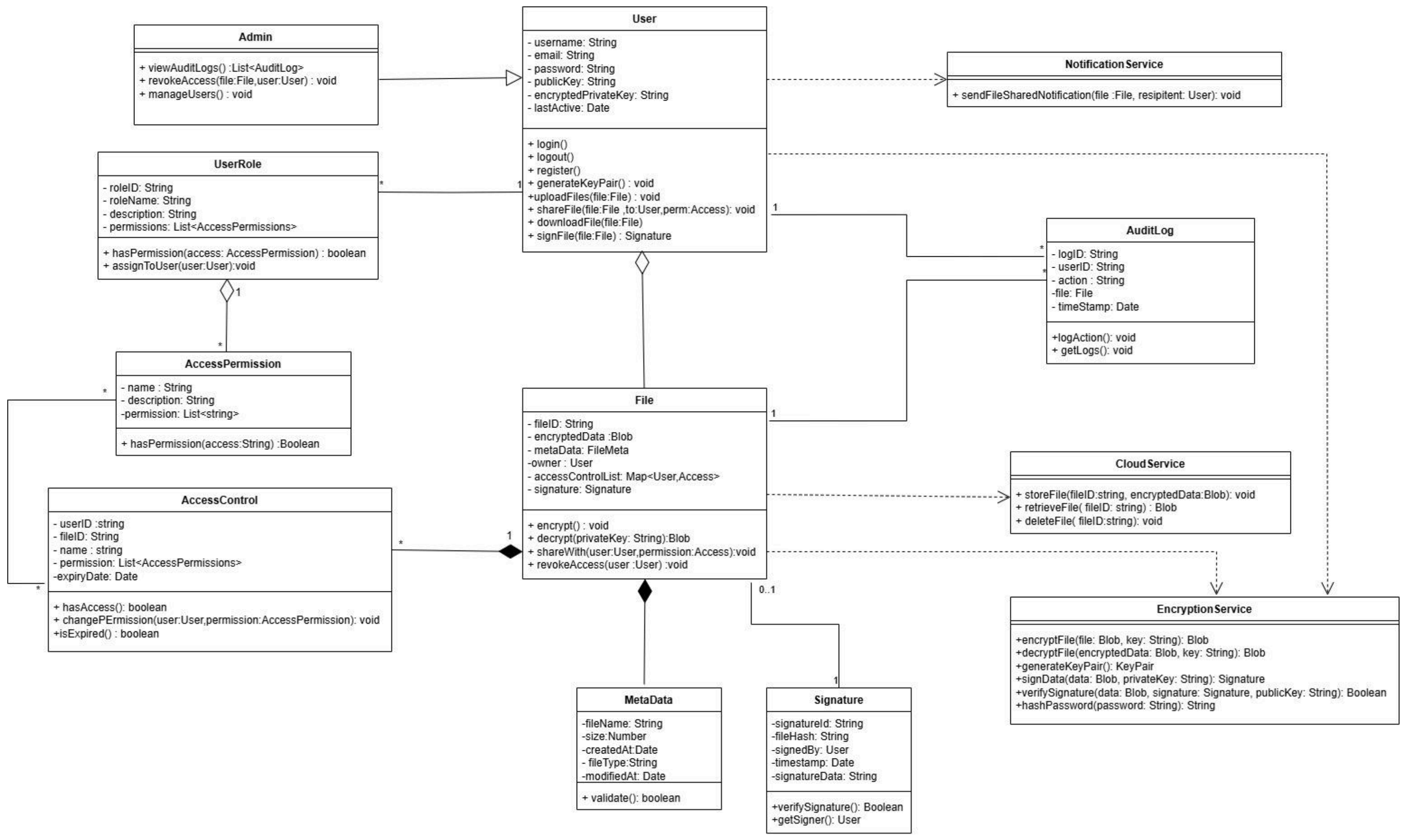
4.8 Versioning

All APIs would follow semantic versioning (v1, v2, etc.) with:

- Version in URL path (/v1/api/files)
- Backward compatibility for at least one previous version
- Deprecation notices in documentation

5 Domain Model

Domain Model


Admin
+ viewAuditLogs() :List<AuditLog>
+ revokeAccess(file:File,user:User) : void
+ manageUsers() : void
User
- username: String
- email: String
- password: String
- publicKey: String
- encryptedPrivateKey: String
- lastActive: Date
+ login()
+ logout()
+ register()
+ generateKeyPair() : void
+uploadFiles(file:File) : void
+ shareFile(file:File ,to:User,perm:Access): void
+ downloadFile(file:File)
+ signFile(file:File) : Signature
NotificationService
+ sendFileSharedNotification(file :File, resipitent: User): void
UserRole
- roleID: String
- roleName: String
- description: String
- permissions: List<AccessPermissions>
+ hasPermission(access: AccessPermission) : boolean
+ assignToUser(user:User):void
*
1
1
*
1
*
AuditLog
- logID: String
- userID: String
- action : String
-file: File
- timeStamp: Date
+logAction(): void
+ getLogs(): void
AccessPermission
- name : String
- description: String
-permission: List<string>
+ hasPermission(access:String) :Boolean
*
*
File
- fileID: String
- encryptedData :Blob
- metaData: FileMeta
-owner : User
- accessControlList: Map<User,Access>
- signature: Signature
+ encrypt() : void
+ decrypt(privateKey: String):Blob
+ shareWith(user:User,permission:Access):void
+ revokeAccess(user :User) :void
1
CloudService
+ storeFile(fileID:string, encryptedData:Blob): void
+ retrieveFile(fileID: string) : Blob
+ deleteFile(fileID:string): void
AccessControl
- userID :string
- fileID: String
- name : string
- permission: List<AccessPermissions>
-expiryDate: Date
+ hasAccess(): boolean
+ changePErmission(user:User,permission:AccessPermission): void
+isExpired() : boolean
*
1
0..1
1
EncryptionService
+encryptFile(file: Blob, key: String): Blob
+decryptFile(encryptedData: Blob, key: String): Blob
+generateKeyPair(): KeyPair
+signData(data: Blob, privateKey: String): Signature
+verifySignature(data: Blob, signature: Signature, publicKey: String): Boolean
+hashPassword(password: String): String
MetaData
-fileName: String
-size:Number
-createdAt:Date
- fileType:String
-modifiedAt: Date
+ validate(): boolean
Signature
-signatureId: String
-fileHash: String
-signedBy: User
-timestamp: Date
-signatureData: String
+verifySignature(): Boolean
+getSigner(): User

Architectural Requirements

6 Architectural Requirements

6.1 Quality Requirements

NF1: Security Requirements

Description
For a secure file sharing platform, the security of the system is paramount ensuring the protection of sensitive data, prevention of unauthorized access, and compliance with data protection regulations. It encompasses all measures taken to protect the system and its data from unauthorized access, use, disclosure, disruption, modification, or destruction.

Quantification
To properly quantify security, adherence to industry-standard encryption protocols and compliance with data protection regulations will be assessed. The system's resilience against common attack vectors will be measured through penetration testing. Access control mechanisms will be evaluated for their effectiveness in restricting unauthorized access.

Targets

NFR1.1: Sensitive user credentials must be encrypted using industry-standard algorithms (AES-256 for symmetric encryption and x3dh for asymmetric operations).
NFR1.2: Multi-factor authentication must be enforced where configured.
NFR1.3: Tokens should expire after an hour..
NFR1.4: No unencrypted file or key material shall be stored on the server.
NFR1.5: Compliance with data protection regulations is mandatory.
NFR1.6: Permissions shall be enforced at the backend and shall not be bypassable via client modifications.
NFR1.7: Administrative interfaces must be restricted to authenticated and authorized users only.
NFR1.8: The system must log all critical and warning-level errors with full context for debugging, and these logs must be auditable.
NFR1.9: Keys must be generated using a secure random number generator.
NFR1.10: The system must ensure zero exposure of private keys to the server.
NFR1.11: Private keys should be stored in a sealed vault.
NFR1.12: Signing operations must comply with digital signature standards.

NFR1.13: Administrator actions must be auditable and traceable, capturing who performed what action and when.

NF2: Performance Requirements

Description
The system must execute operations efficiently and respond promptly to user interactions, ensuring a smooth and responsive experience without undue delays.

Quantification
Performance will be measured through load testing, response time monitoring for critical operations, and throughput analysis. Metrics will include the time taken to complete specific tasks under defined loads.

Targets

NFR2.1: File uploads of 10mb files should not take longer than 30 seconds.
NFR2.2: Digital signature verification must be deterministic and complete within 3 seconds.

NF3: Availability

Description
It concerns the ability of the system to perform its required functions under stated conditions for a specified period of time, and to be accessible and operational when needed.

Quantification
Availability will be measured by system uptime percentage and the success rate of automated data purging processes.

Targets

NFR3.1: The system should have an overall uptime of 95% over 7 days.
NFR3.2: Scheduled maintenance should take no longer than an hour.

NF4: Usability Requirements

Description
The system's user interface must be intuitive, accessible, and provide clear and actionable feedback to users, facilitating efficient and error-free interaction.

Quantification
Usability will be measured through user feedback, evaluation of task completion times for new users, and the clarity and helpfulness of error messages. Formal UX reviews will also quantify these aspects.

Targets

NFR4.1: The user interface must visually distinguish between different permission levels clearly, aiding users in understanding their capabilities.
NFR4.2: All error messages must be user-friendly, localizable, and provide actionable guidance to help users resolve issues.

NF5: Latency Requirements

Description
The system must provide timely feedback and notifications to users, minimizing perceived delays in interaction and information delivery.

Quantification
Latency will be measured by monitoring the delay between a system event (e.g. user action, internal trigger) and the corresponding visual update or notification delivery.

Targets

NFR5.1: Progress indicators must update in real-time (less than 500 ms delay).
NFR5.2: Notification delivery (email/push/pop) must occur within 30 seconds of the triggering event.

6.2 Architectural Mapping

Requirements	***Architectural Strategies***	***Architectural Pattern***
Security	- Enforce client-side encryption using E2EE. - Use Zero-Trust principles for inter-service communication. Isolate key management logic and vaulting in a dedicated service. - Log and monitor critical/auth actions centrally. - Restrict administrative interface	**End-to-End Encryption (E2EE):** Ensures file contents are never exposed to the server, addressing NFR1.1, NFR1.5, NFR1.11. **Zero-Trust Architecture:** Authenticates all interactions independently, supporting NFR1.2, NFR1.3, NFR1.7,. **Secure Key Management:** Isolates cryptographic processes and storage, fulfilling NFR1.10, NFR1.12. **Client-Side Logic Enforcement:** Prevents server from processing encrypted content, enforcing separation of duties (NFR1.11).
Performance	- Separate responsibilities via microservices. - Use asynchronous communication for non-critical tasks. - Isolate and scale critical operations (e.g., uploads, signing).	**Microservices-Based Deployment:** Decouples logic, improving throughput and load handling (NFR2.1, NFR2.2). **Event-Driven Communication:** Handles operations like logging and notifications asynchronously, reducing bottlenecks (NFR2.2, NF5).
Availability	- The site should have a good uptime - Scheduled maintenance should not have an effect on the availability of the site	**Microservices-Based Deployment:** Fault isolation ensures other services will continue to run without fault.
Usability	- Structure UI using Model-View-Controller. - Separate presentation, logic, and data handling for maintainability and clarity. - Provide localized, accessible error messages. - Distinguish user roles visually in the UI. - Ensure the client handles encryption tasks with minimal	**Model-View-Controller (MVC):** Organizes UI into clear layers: the Model manages data, the View renders UI, and the Controller handles user input and updates. This structure supports NFR4.1 and NFR4.2 by making the UI intuitive, easier to maintain, and testable. **Client-Side Logic Enforcement:** Requires UX design that simplifies complex encryption processes (NFR4.1, NFR4.2). **Zero-Trust Architecture:** Enforces role aware UI constraints and backend enforcement of

	friction.	permissions (NFR4.1).
Latency	- Use pub/sub or messaging queues for notifications. - Ensure low-latency operations have dedicated channels. - Push progress updates from client/server in real time.	**Event-Driven Communication:** Ensures near real-time updates for notifications and progress (NFR5.1, NFR5.2). **Microservices-Based Deployment:** Allows low-latency services (e.g., notifications) to remain performant independently of heavier services (NFR5.2).

6.3 Architectural Diagram

6.3.1 Architectural Patterns

Microservices architecture

Reasoning:

Allows for isolation of services. Allowing for independent scalability and flexibility in languages, as we plan to use GO for encryption services.

MVC Architecture

Reasoning:

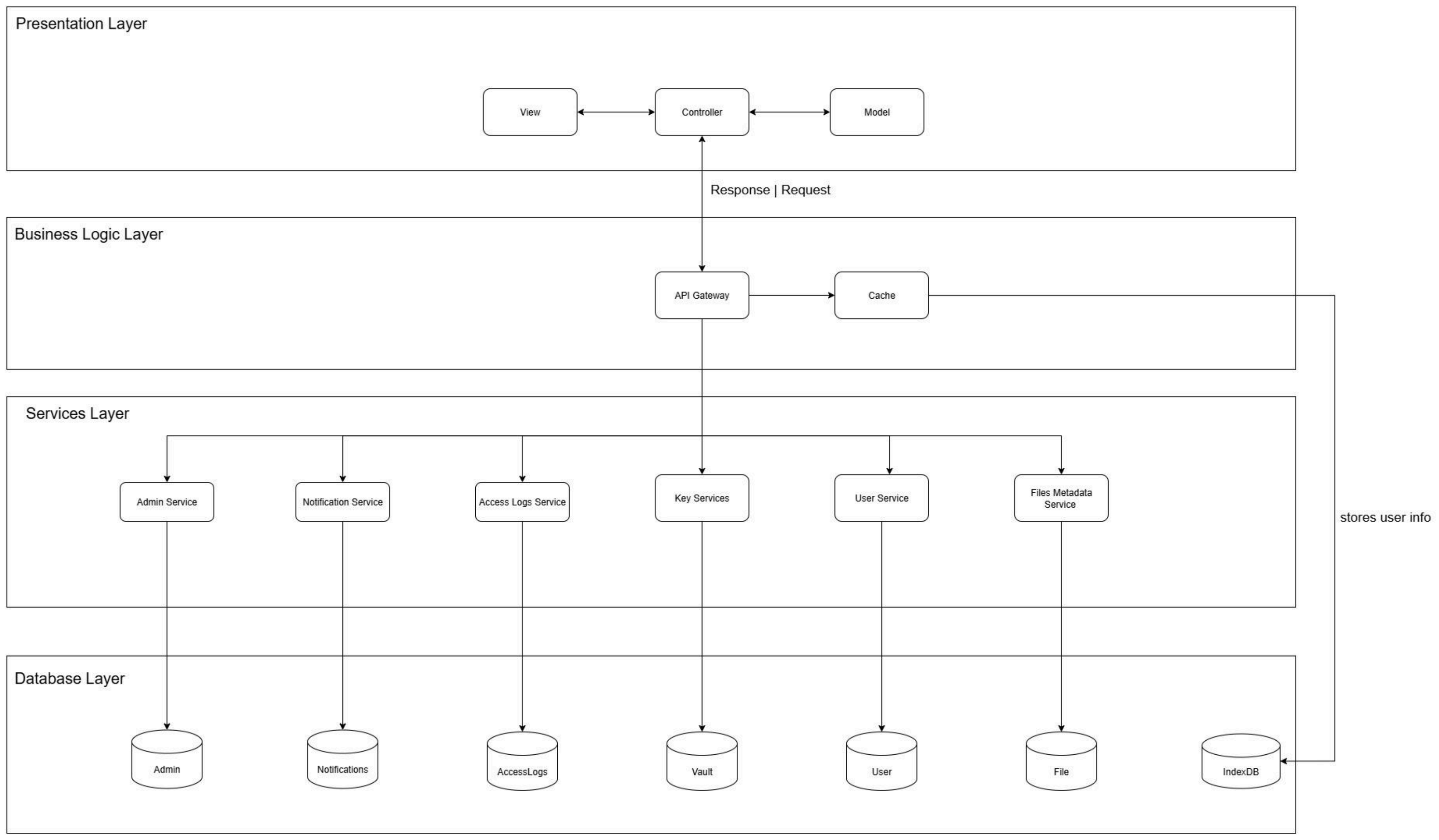
Allows us to organize UI, and creates a separation of concerns. This will also help down the line with mobile development.

Layered Architecture

Reasoning:

Allows us to scale appropriately and allows maintainability as each layer has a distinct responsibility.

6.3.2 Architectural Diagram



6.4 Design Patterns

The following design patterns were selected (subject to change) for the following functionality:

- **Decorator Pattern** – For logging, auditing, and digital signature.,
- **Prototype Pattern** – For managing users.,
- **Observer Pattern** – To support real-time notifications and updates.,
- **Proxy Pattern** – For access verification and controlled resource access.,
- **Command Pattern** – To queue and execute file operations like uploads/downloads.

6.5 Constraints

End-to-End Encryption (E2EE):

All files must be encrypted on the client before upload and decrypted only after download. This ensures that server-side components cannot access the file contents in plaintext. It impacts system design by limiting the ability to process, inspect, or search file contents on the server.

Zero-Trust Architecture:

The system assumes no implicit trust between components. Each request must be authenticated and authorized independently. File contents remain 'invisible' to the server, reinforcing the separation of concerns between storage, access control, and identity.

Secure Key Management:

Public/private key pairs must be securely generated, distributed, and stored. Only the sender and intended recipient(s) should have access to decryption keys. This constraint requires strong encryption standards, secure key exchange implementations, and safeguards for preventing unauthorized access.

Microservices-Based Deployment:

The architecture is composed of independently deployable microservices. This requires containerized environments (e.g. Docker), secure services communication. It also enforces separation of concerns between components such as authentication, file storage, and notifications.

Scalability and Elasticity:

Each microservice should be able to grow or shrink on its own depending on how much it's being used. For example, if many users are uploading files, only the upload service needs to scale. This helps the system handle high traffic without slowing down.

Client-Side Logic Enforcement:

Due to E2EE, operations such file previews, or metadata extraction cannot occur on the server. These must be handled entirely on the client, placing architectural limits on server responsibilities.

Event-Driven Communication:

Actions like file upload, sharing, or download must trigger asynchronous events (e.g., logging, auditing, notifications). This requires an event-driven architecture using message queues or pub/sub systems.

Data Residency and Compliance:

The platform must comply with South Africa's data protection laws, especially POPIA. This means storing personal data securely and only in allowed regions, and deleting logs when no longer needed.

7

Technology Requirements

7.1 Technology Stack Overview

Technology	Description
Next.js	Main frontend framework, bootstrapped with create-next-app for React-based web application development.
Tailwind CSS	Utility-first CSS framework for fast and responsive UI styling.
Express.js + Node.js	Backend API server and routing logic for handling HTTP requests and business logic implementation.
Golang	Handles large file uploads efficiently for high performance.
Postgres	Database management system allowing for data saving.
Supabase	Backend-as-a-Service platform providing authentication and user management.
Electron	Desktop application
HashiCorp	To store keys in a sealed vault
GitHub Actions	Handles automated testing
OwnCloud	Handles the cloud storage of files
Cypress + Testify + Pytest + Jest	Testing frameworks

7.2 Technology Choices

This section evaluates at least three technology options per system component. For each, we provide an overview, pros and cons, and a clear justification for the selected choice.

Frontend Framework

Overview

The frontend manages the user interface and system interaction. It must be performant and easily styled

Framework	Pros	Cons
React + Next.js + Tailwind CSS	Reusable component-based architecture, large ecosystem, and cross-platform consistency.	Fast-evolving ecosystem, learning curve
Angular	Built-in features, strong CLI, two-way binding	Heavier setup, complex for small teams
Vue.js	Simple syntax, lightweight, great for rapid dev	Smaller community, less opinionated

✅**Selected**

React + Next.js + Tailwind CSS

Justification

Combines React’s modular design with Next.js’s server-side rendering and Tailwind CSS’s utility-first styling, making it ideal for scalable and SEO-friendly apps

Backend Framework

Overview

The backend handles services ,such as file encryption and key management, and database/API operations.

Framework	Pros	Cons
Express.js (Node.js)	Lightweight, RESTful APIs, full JS stack	Manual error handling, scalability limits
Spring Boot (Java)	Robust, scalable, microservices-ready	Heavy, steeper learning curve
ASP.NET Core (C#)	High performance, scalable, secure	Windows-first, steep learning curve

✅ Selected

Express.js (Node.js)

Justification:

Lightweight and fast for REST APIs, fits with full-stack JavaScript, and ideal for rapid development and modular encryption/middleware logic.

Data Storage

Overview

The platform needs persistent, secure, and real-time data storage.

Storage	Pros	Cons
Supabase	Open-source, real-time sync, built-in auth	Limited analytics
Firebase	Scalable, auth & hosting	Vendor lock-in, pricing at

	included	scale
MongoDB	Schema-less, scalable	Schema validation can be complex

✅ **Selected**

Supabase

Justification

SQL-based, with real-time features, file storage, and auth. Aligns with open-source philosophy and simplifies integration with frontend and backend.

Code Editor / IDE

Overview

Tool used to write, debug, and maintain source code.

IDE	**Pros**	**Cons**
VS Code	Lightweight, extensions, Git integration	Lacks full IDE depth
Visual Studio	Powerful for .NET, enterprise-grade	Heavy, Windows-only
Vim/Neovim	Fast, customizable	Steep learning curve

✅ **Selected**

VS Code

Justification

Perfect balance of power and simplicity. Ideal for JavaScript/Node.js workflows, team collaboration, and containerized dev environments.

CI/CD Pipeline

Overview

Automates testing, building, and deployment workflows.

Tool	Pros	Cons
GitHub Actions	Native GitHub integration, easy to configure	Slower builds
Azure DevOps	Full DevOps suite, approval flows	Complex setup
Octopus Deploy	Strong CD, visual pipelines	Paid, separate CI tool needed

✅ **Selected**

GitHub Actions

Justification

Built directly into GitHub, supports branch-based workflows and integrates seamlessly with our source control and testing tools.

Version Control & Collaboration

Overview

Tracks changes, enables collaboration, and supports CI/CD integrations.

Platform	Pros	Cons
GitHub	Open-source community, CI/CD integration	Basic project management
GitLab	Rich features, self-hosted option	Resource heavy
Bitbucket	Good for small teams	Dated UI, smaller ecosystem

✅ **Selected**

GitHub

Justification

Centralized version control, integrated with GitHub Actions for CI/CD, and supports team workflows with issues, pull requests, and project boards.

Cloud Infrastructure

Overview

Used for encrypted file storage, sharing, and collaboration.

Platform	**Pros**	**Cons**
Nextcloud	Open-source, secure, API support	May need performance tuning
OwnCloud	Lightweight, fast updates	Inconsistent releases
Dropbox	Easy integration, stable	Not privacy-focused

✅ **Selected**

OwnCloud

Justification

Self-hosted, open-source, secure APIs. Ideal for a privacy-first application with full control over file storage and user access.

Testing Stack

Overview

Ensures application reliability, correctness, and integration safety.

Stack	**Pros**	**Cons**

Jest + Pytest + Cypress +Testify	Supports all levels of testing, good TS support	Heavy in large monorepos
Vitest + Testify	Fast, Vite-compatible	Smaller community
Mocha + Chai	Mature, customizable	Requires more setup

✅ Selected

Jest + Pytest + Cypress

Justification

Covers unit, integration, and E2E testing, with wide community support and tools for both frontend and backend testing.

Encryption & Authentication

Overview

Ensures data privacy, secure key exchange, and authenticated user sessions.

Combination	Pros	Cons
AES-256 + RSA + JWT	- Strong encryption and widely adopted standards - JWT supports stateless auth - RSA is well understood and secure	- RSA is slower for large key sizes - No forward secrecy - JWT tokens require proper storage and expiration handling
AES-256 + X3DH + Auth Cookies	- End-to-end encryption with forward secrecy - Secure session cookies resist XSS - X3DH is ideal for asynchronous secure messaging	- Requires more complex session handling - Cookies can be affected by CSRF without proper protection - X3DH can be complex to implement
AES-256 + X3DH + JWT	- Combines modern	- JWT can be stolen if not

	encryption with forward secrecy - Stateless, scalable auth with JWT - Ideal for APIs and microservices	secured properly - X3DH requires key pre-distribution - Requires careful session expiration and renewal logic

✅ **Selected**

AES-256 + X3DH + JWT

Justification

This combination provides strong encryption (AES-256), a modern and secure key exchange mechanism (X3DH) suitable for asynchronous communication, and JWT-based session management, which enables stateless and scalable authentication. Together, they fulfill both the security and usability requirements of the Secure Share platform.

Secrets Management & Key Storage

Overview

Secure storage and management of encryption keys, API credentials, and sensitive configuration.

Platform	**Pros**	**Cons**
HashiCorp Vault	- Enterprise-grade security - Dynamic secrets - Access control & audit logs	- Complex initial setup - Requires proper HA config
AWS Secrets Manager	- Native AWS integration - Auto-rotation - IAM integration	- Vendor lock-in - Per-secret pricing - Regional limitations
Azure Key Vault	- Microsoft ecosystem integration - Managed HSM - Certificate management	- Azure-specific - Complex RBAC model - Higher latency

✅ **Selected**

HashiCorp Vault

Justification

HashiCorp Vault provides enterprise-grade security with fine-grained access control, audit logging, and dynamic secrets generation. As an open-source solution, it avoids vendor lock-in while offering enterprise-level security features, making it ideal for our sensitive key management needs.

Vault Service Implementation Language

Overview

Programming language used to implement secure interaction with the vault service for key management.

Language	**Pros**	**Cons**
Python	- Extensive crypto/security libraries - Clean, readable syntax - Strong HashiCorp SDK	- Slower execution - GIL limitations for threading
Go	- Performant - Strongly typed - Native HashiCorp language	- Steeper learning curve - Less extensive library ecosystem
Node.js	- JavaScript ecosystem consistency - Async by default - Fast development	- Less mature crypto libraries - Dependency management complexity

✅ **Selected**

Python

Justification

Python was selected for the Vault service implementation due to its comprehensive security-focused libraries, excellent HashiCorp Vault SDK support, and readable syntax that enhances security audit capabilities. For security-critical components like key management, Python's robust ecosystem provides the right balance of security, maintainability, and development speed.

Desktop Application

Overview

Used for building an admin desktop application.

Platform	Pros	Cons
Electron	Cross-platform, uses web technologies (HTML/CSS/JS), large community, easy integration with Node.js	Can be memory-heavy, larger app size
NW.js	Similar to Electron, flexible	Smaller community, less documentation
Qt (C++)	Lightweight, secure, small bundle size	Rust knowledge required, newer ecosystem

✅ Selected

Electron

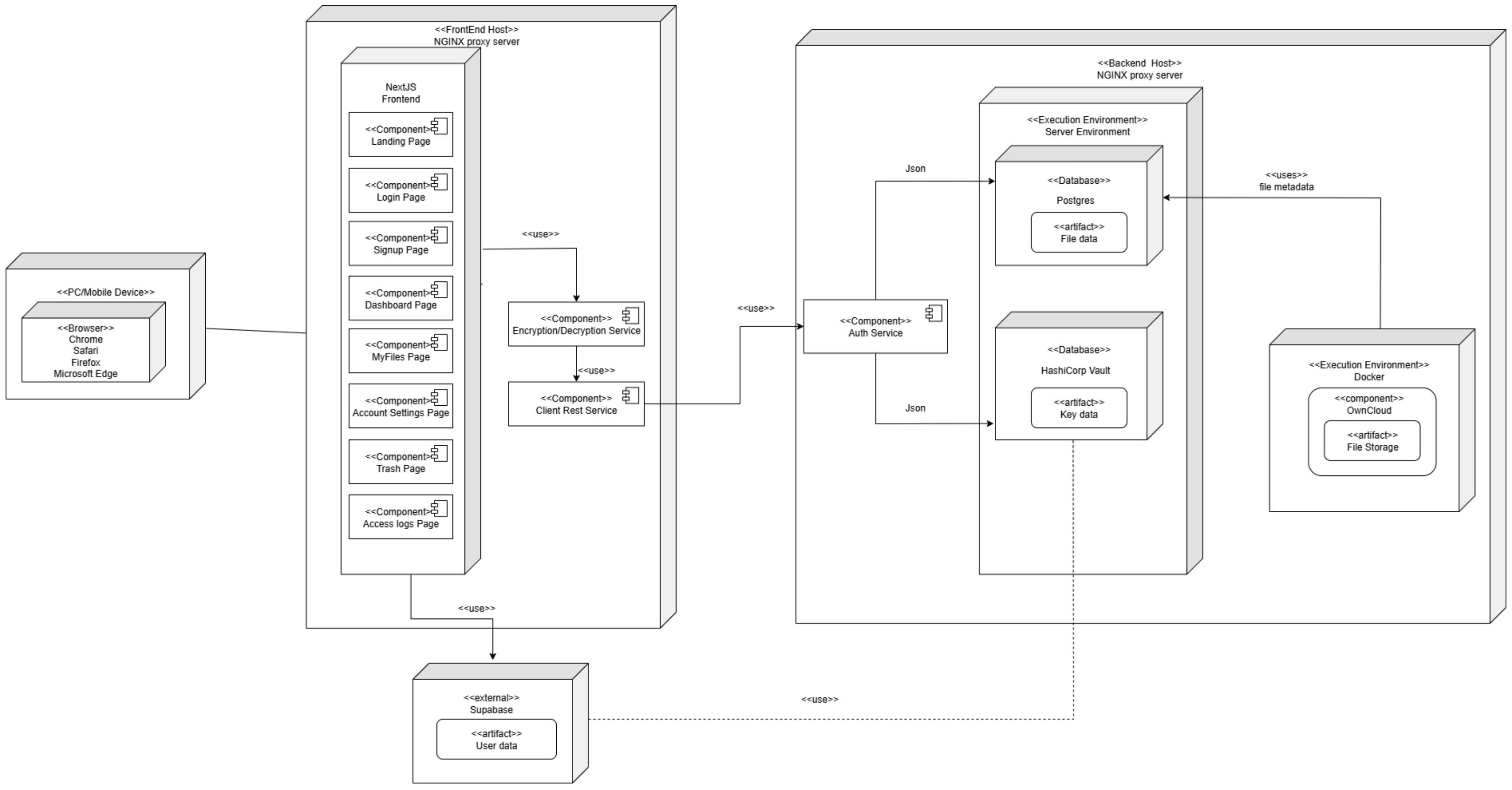
Justification

Cross-platform, uses familiar web technologies, and integrates seamlessly with Node.js. Ideal for building a secure, feature-rich desktop client quickly while maintaining full control over the UI and backend logic.

8

Deployment Model

Deployment Model



<<FrontEnd Host>>
NGINX proxy server
NextJS Frontend
<<Component>> Landing Page
<<Component>> Login Page
<<Component>> Signup Page
<<Component>> Dashboard Page
<<Component>> MyFiles Page
<<Component>> Account Settings Page
<<Component>> Trash Page
<<Component>> Access logs Page
<<use>>
<<Component>> Encryption/Decryption Service
<<use>>
<<Component>> Client Rest Service
<<PC/Mobile Device>>
<<Browser>> Chrome Safari Firefox Microsoft Edge
<<use>>
<<external>> Supabase
<<artifact>> User data
<<use>>
<<Backend Host>>
NGINX proxy server
<<use>>
<<Component>> Auth Service
Json
Json
<<Execution Environment>> Server Environment
<<Database>> Postgres
<<artifact>> File data
<<Database>> HashiCorp Vault
<<artifact>> Key data
<<uses>> file metadata
<<Execution Environment>> Docker
<<component>> OwnCloud
<<artifact>> File Storage

9 Resources

Resources

Versions of **SRS (Software Requirements Documentation)** document :

- Version 1
- Version 2
- Version 3
- Version 4