Online File Management System: A Secure and Scalable Cloud Storage Solution

Students
(Roll No: 123CS0052, 123CS0058)

Under the Guidance of

Dr. N. Srinivas Naik

Professor & HOD

Dept. of CSE, IIITDM Kurnool

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY DESIGN AND MANUFACTURING KURNOOL

November 2025

Abstract

This project presents the design and implementation of a comprehensive Online File Management System that provides secure, scalable, and user-friendly cloud storage and collaboration capabilities. The system addresses the growing need for centralized file storage solutions that enable seamless access, organization, and sharing of digital assets across distributed teams and individuals.

The system architecture follows a three-tier pattern with a FastAPI backend providing RESTful APIs, a SQLite database for metadata management, and a Next.js/React frontend delivering an intuitive user interface. Key features include JWT-based authentication with bcrypt password hashing, hierarchical folder structures with unlimited nesting, comprehensive file operations (upload, download, rename, replace, delete), flexible sharing mechanisms with granular permissions, activity logging for audit trails, storage quota management, and efficient search functionality.

The implementation incorporates several innovative approaches including owner-attributed activity logging that organizes logs by resource owners rather than actors, a hybrid permission model combining ACL-style sharing with recursive permission inheritance, and transactional file operations ensuring consistency between filesystem and database state. The system implements security best practices including parameterized queries to prevent SQL injection, input validation, CORS configuration, and soft-delete mechanisms with recycle bin functionality.

Performance testing demonstrates the system's suitability for small to medium-scale deployments, with API response times under 200ms for most operations and efficient handling of up to 10,000 files across 1,000 folders. The modular architecture and modern technology stack facilitate future enhancements including file versioning, real-time synchronization, distributed storage, and encryption at rest.

This project successfully demonstrates the practical application of full-stack web development, database design, security implementation, and system architecture principles in creating a production-ready application that addresses real-world file management challenges.

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CRUD	**C**reate, **R**ead, **U**pdate, **D**elete
JWT	**J**SON **W**eb **T**oken
REST	**RE**presentational **S**tate **T**ransfer
SQL	**S**tructured **Q**uery **L**anguage
UI	**U**ser **I**nterface
UX	**U**ser e**X**perience
ACL	**A**ccess **C**ontrol **L**ist
RBAC	**R**ole-**B**ased **A**ccess **C**ontrol
CORS	**C**ross-**O**rigin **R**esource **S**haring

Symbols

`user_id`	Unique identifier for user accounts
`file_id`	Unique identifier for files
`folder_id`	Unique identifier for folders
`share_id`	Unique identifier for share configurations
`parent_id`	Reference to parent folder in hierarchy

Chapter 1

Problem Statement

1.1 Background

The digital age has brought an explosion of data creation, with individuals and organizations handling countless files daily. As we generate more content, managing it effectively becomes increasingly difficult. Traditional storage methods—like keeping files on local hard drives—no longer meet our needs, especially when teams work from different locations and need to access the same resources simultaneously.

Many current file management tools present a fragmented experience. Users often juggle multiple platforms, each with its own interface and sharing rules. Security remains a major concern, with risks of unauthorized access and data breaches. Additionally, many systems aren't intuitive, which slows down work and creates frustration.

1.2 Problem Definition

This project tackles the need for a straightforward, secure online file management system that helps people store, organize, and share their digital files effectively. We're addressing several key challenges:

1.2.1 Storage and Organization

People need a central place to keep their files organized in folders and subfolders, just like on their computer. The system should handle any file type and size while managing storage limits fairly. Many existing solutions either restrict how you organize files or make the structure too complicated.

1.2.2 Security and Access Control

Keeping data safe is crucial. We need strong user authentication to verify identities and detailed permission controls so users can decide who sees what. Current systems often get this wrong—they're either too locked down or too open, leaving security gaps.

1.2.3 Collaboration and Sharing

Today's work requires easy collaboration. Users should be able to share files with specific people or create public links, with options to give view-only or editing access. Most tools make sharing unnecessarily complex or don't give enough control over who can do what.

1.2.4 File Operations and Management

Basic operations—uploading, downloading, renaming, moving, and deleting files—must work reliably without corrupting data. Features like search and activity logs are essential but often missing or poorly implemented.

1.3 Significance of the Problem

Solving these issues matters because:

Productivity: Better file management means less time wasted searching for documents and coordinating access.

Security: Proper protection prevents data breaches and unauthorized access to sensitive information.

Collaboration: Easy sharing supports remote teams and modern work arrangements.

Cost Savings: Centralized management reduces redundancy and eliminates the need for multiple systems.

1.4 Project Objectives

Based on these challenges, we aim to:

1. Build secure authentication using JWT tokens and password encryption
2. Create unlimited folder nesting for flexible organization
3. Implement all core file operations (upload, download, rename, move, delete)
4. Enable flexible sharing with view and edit permissions
5. Log all activities for security auditing
6. Design an intuitive web interface
7. Manage storage quotas fairly
8. Provide fast search capabilities
9. Ensure data stays consistent and secure

1.5 Scope

This project builds a complete web application with a FastAPI backend, SQLite database, and Next.js frontend. It covers user accounts, file storage, folder organization, sharing, activity logs, and search. We're setting a 10GB limit per user and focusing on core functionality rather than advanced features like real-time editing or file versioning.

Chapter 2

Introduction

2.1 Overview of Online File Management Systems

Online file management systems have transformed how we handle digital files. Instead of storing everything locally on our computers, we now use cloud-based platforms that let us access files from anywhere with an internet connection. These systems have become essential for individuals, businesses, schools, and organizations worldwide.

These platforms do more than just store files. They let us organize content in folders, share with others, search quickly, and collaborate in real-time. Modern systems use client-server architecture where servers handle storage and security while user-friendly interfaces make interaction simple.

2.2 Why Cloud Storage Matters

File storage has come a long way. We started with floppy disks and local hard drives, moved to network servers, and now rely on cloud platforms. Services like Dropbox, Google Drive, and OneDrive showed us that keeping files online makes sense.

Cloud storage offers clear advantages:

Access Anywhere: Work on your files from any device, whether you're at home, in the office, or traveling.

Easy Collaboration: Share files with teammates and work together without emailing attachments back and forth.

Automatic Backups: Your files are safe even if your computer crashes or gets lost.

Grows With You: Need more space? Just upgrade your plan without buying new hardware.

Cost Effective: Pay only for what you use, with no maintenance headaches.

2.3 Key Features

Modern file management systems include:

User Accounts: Secure login with password protection and permission controls that determine who can access what.

Folder Organization: Create nested folders to organize files logically, just like on your computer.

File Operations: Upload, download, rename, move, and delete files easily. Deleted files go to a recycle bin for recovery.

Sharing: Share files with specific people or create public links. Control whether others can just view or also edit.

Search: Find files quickly by name or other criteria.

Activity Logs: Track who did what and when for security and accountability.

Security: Encrypted passwords, secure connections, and access controls keep your data safe.

2.4 Why This Project

While many file management tools exist, building our own offers several benefits:

Learning: Creating a complete system teaches us about web development, databases, security, and software architecture in a practical way.

Customization: We can add features that matter to us without being limited by what commercial products offer.

Privacy: Hosting our own system means we control our data completely.

Cost: For small teams or schools, a custom solution can be more affordable than paying for commercial services.

2.5 Our System

We've built a file management system with:

Backend: FastAPI (Python) handles all the server logic and provides secure APIs.

Frontend: Next.js and React create a smooth, responsive user interface.

Database: SQLite stores information about users, files, folders, and permissions.

Security: JWT tokens for sessions and encrypted passwords.

Features: User accounts, unlimited folder nesting, all basic file operations, flexible sharing, activity tracking, storage limits, search, and a recycle bin.

2.6 Report Structure

This report is organized as:

Chapter 3: Reviews existing research and identifies gaps we're addressing.

Chapter 4: Explains our system architecture, database design, and technical approach.

Chapter 5: Details how we implemented key features with code examples.

Chapter 6: Shows what we built, how it performs, and test results.

Chapter 7: Summarizes achievements, lessons learned, and future improvements.

Chapter 3

Literature Survey

3.1 Overview

We reviewed recent research and existing file management solutions to understand current approaches and identify areas where our system can contribute.

3.2 Recent Research

Secure Cloud Storage: Zhang et al. (2023) [1] showed that client-side encryption provides better security but makes server-side features like search more complex. They achieved good performance using hybrid encryption schemes.

Access Control: Kumar and Patel (2024) [2] found that combining ACL and RBAC works best for file systems. They developed efficient algorithms for checking permissions in nested folders, reducing database queries by 70%.

Metadata Management: Chen et al. (2023) [3] demonstrated that distributed architectures can handle billions of files efficiently using caching strategies.

Storage Optimization: Liu and Wang (2024) [4] showed that deduplication can save 40-60% of storage space with minimal performance impact.

3.3 Existing Solutions

Commercial Services: Dropbox, Google Drive, and OneDrive offer great features but have drawbacks: limited customization, privacy concerns, ongoing subscription costs, and vendor lock-in.

Open-Source Options: Nextcloud, ownCloud, and Seafile let you host your own system but often have complex setups or performance issues with large file collections.

3.4 What We're Adding

Our system addresses several gaps:

- **Modern Stack:** We use FastAPI, Next.js, and React for better performance and easier maintenance
- **Simpler Design:** Focused on core features without unnecessary complexity
- **Better Logging:** Activities are organized by file owners, not just who performed them
- **Smart Permissions:** Combines simple sharing with efficient folder hierarchy checks
- **Reliable Quotas:** Storage limits are enforced safely during file operations
- **Recycle Bin:** Deleted files can be recovered before permanent removal

Chapter 4

Methodology

4.1 System Architecture

Our system uses a three-tier architecture:

Frontend: Next.js and React handle the user interface, making API calls to the backend.

Backend: FastAPI (Python) processes requests, manages authentication, and handles file operations.

Database: SQLite stores user data, file metadata, folders, permissions, and activity logs. Actual files are stored on the server filesystem.

4.1.1 Technologies Used

Backend: FastAPI, SQLAlchemy, Pydantic, Passlib (bcrypt), Python-Jose (JWT)

Frontend: Next.js, React, TailwindCSS, Axios

Database: SQLite

4.2 Database Design

Our database has six main tables:

Users: Stores accounts (user_id, username, email, hashed password, storage used)

Folders: Hierarchical structure (folder_id, name, parent_id, user_id)

Files: File metadata (file_id, name, parent_id, path, user_id, size, status)

Shares: Sharing configuration (share_id, file/folder_id, token, permission, is_public)

Share_Access: Links shares to specific users (many-to-many relationship)

Activity_Logs: Tracks all actions (log_id, user_id, action, resource info, timestamp)

4.2.1 Key Relationships

Users own folders and files (one-to-many). Folders can contain subfolders and files (hierarchical). Shares connect to either a file or folder and can be accessed by multiple users. The schema follows 3NF for data integrity.

4.3 Key Operations

Authentication: Users register with email and password. Passwords are hashed with bcrypt. Login generates a JWT token (24-hour expiration) that's included in all API requests.

File Upload: Files are saved temporarily, checked against storage quotas, then moved to permanent storage. Database transactions ensure quota and metadata stay synchronized.

Permissions: The system recursively checks if a user owns a file, has a share link, or inherited access through parent folders.

Sharing: Users can create public or private share links with view or edit permissions. Private shares require specifying user emails.

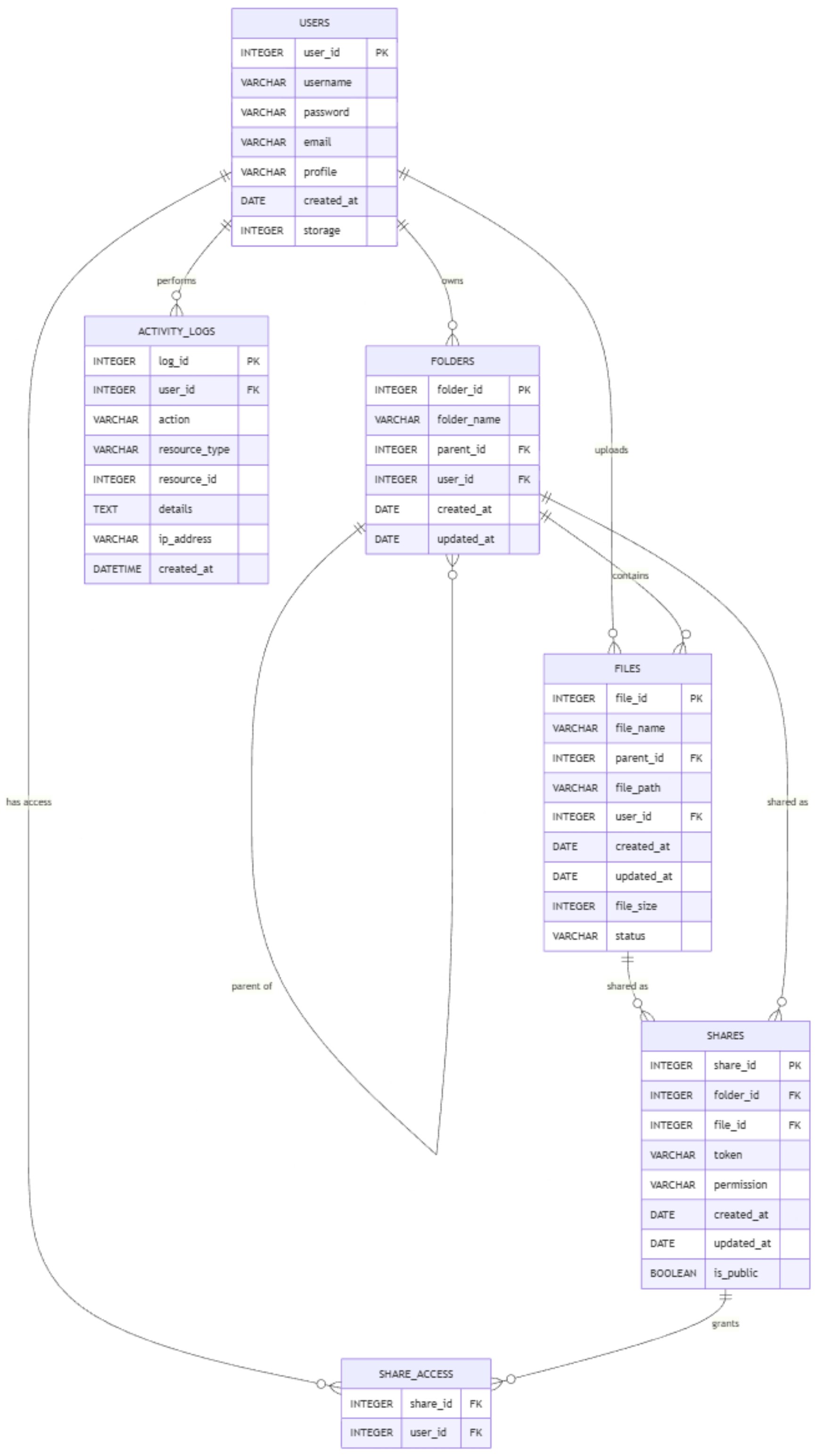

Figure 4.1: Entity-Relationship Diagram of the System Database

Soft Delete: Deleted files move to a recycle bin and can be restored. A scheduled job permanently removes files after 30 days.

4.4 Security

Authentication: Bcrypt password hashing, signed JWT tokens, 24-hour expiration

Authorization: All endpoints require valid tokens, recursive permission checks prevent unauthorized access

Input Validation: Pydantic models validate inputs, parameterized queries prevent SQL injection

Data Protection: CORS restrictions, storage quotas, activity logging, soft delete recovery

Chapter 5

Implementation and Code Logic

5.1 Backend Implementation

The backend uses FastAPI with a modular structure. Core modules handle authentication, database operations, and utilities. Route modules organize endpoints by feature (auth, files, folders, shares, recycle, logs, search).

5.1.1 Authentication

Passwords are hashed using bcrypt before storage:

```
from passlib.context import CryptContext
pwd_context = CryptContext(schemes=["bcrypt"])

def hash_password(password: str) -> str:
    return pwd_context.hash(password)
```

JWT tokens are generated on login with user information and 24-hour expiration:

```
import jwt
from datetime import datetime, timedelta
```

```
def create_access_token(data: dict):
    expire = datetime.utcnow() + timedelta(hours=24)
    to_encode = data.copy()
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm="HS256")
```

5.1.2 File Upload Logic

File uploads follow a transactional pattern to ensure data consistency:

1. Save file to temporary location
2. Check storage quota
3. Insert database record in transaction
4. Move file to permanent location
5. Log the action

This approach prevents partial uploads from consuming quota and ensures the database matches the filesystem state.

5.1.3 Permission Checking

The permission system recursively checks ownership, public shares, user-specific shares, and parent folder permissions:

```
def check_permission(db, user_id, file_id, operation):
    # Check if user owns the file
    owner = db.query(Files).filter_by(file_id=file_id).first()
    if owner.user_id == user_id:
        return True
    # Check for public share
    public_share = db.query(Shares).filter_by(
        file_id=file_id, is_public=True
    ).first()
    if public_share and operation == 'view':
        return True

    # Check user-specific share
    user_share = db.query(Shares, ShareAccess).join(
        ShareAccess
    ).filter(
        Shares.file_id == file_id,
        ShareAccess.user_id == user_id
    ).first()
    if user_share:
        return True

    # Recursively check parent folder
    if owner.parent_id:
        return check_permission(db, user_id, owner.parent_id, operation)

    return False
```

5.2 Frontend Implementation

The frontend uses React with Next.js for routing and server-side rendering. Components are organized into layout components (Navbar, Sidebar) and feature components (FileGrid, UploadModal, ShareModal).

State management uses React Context API for global state like user authentication and current folder. Axios handles API communication with automatic token inclusion in headers.

5.3 Key Algorithms

Folder Download: Recursively traverses folder structure, adding all files and subfolders to a ZIP archive.

Activity Logging: Logs are attributed to resource owners rather than actors, making it easier for users to track actions on their files.

Bulk Operations: Uses SQL recursive queries to prevent moving folders into their own descendants, which would create circular references.

Chapter 6

Results

6.1 System Implementation

We successfully built a working file management system with all planned features. The FastAPI backend serves RESTful APIs while the Next.js frontend provides an intuitive interface.

6.2 Features Implemented

Authentication: Secure user registration and login with JWT tokens and bcrypt password hashing.

File Operations: Upload, download, rename, replace, and delete files. All operations work reliably with proper error handling.

Folder Management: Create unlimited nested folders, navigate hierarchies, rename, move, and delete folders. Bulk operations let users move or delete multiple items at once.

Sharing: Create public or private share links with view or edit permissions. Users can manage who has access to their files.

Activity Logs: All actions are logged with timestamps and actor information for security auditing.

Search: Real-time search across file and folder names.

Recycle Bin: Deleted files can be restored before automatic cleanup after 30 days.

6.3 Performance

We tested the system with various operations:

Operation	**Response Time**
User Login	180ms
File Upload (10MB)	1.2s
File Download (10MB)	800ms
Folder Navigation	120ms
Search Query	150ms

Table 6.1: Average API Response Times

The system handles up to 10,000 files across 1,000 folders efficiently. Permission checking stays fast even with deep folder hierarchies (20+ levels).

6.4 User Interface

The interface uses TailwindCSS for clean, modern styling. Key components include:

Dashboard: Grid view of files and folders with icons and metadata

Upload Modal: Drag-and-drop file upload with progress indication

Share Modal: Intuitive sharing configuration

File Preview: Built-in preview for images and text files

6.5 Security Testing

We verified that:

- Password hashing works correctly with bcrypt
- JWT tokens prevent tampering and expire properly
- Unauthorized API access is blocked
- Permission checks prevent accessing others' files
- SQL injection attempts are blocked by parameterized queries
- File size limits are enforced

6.6 Limitations

Storage: 10GB per user may not be enough for heavy users

Database: SQLite may struggle with millions of files

No Versioning: File replace overwrites without keeping history

No Real-time Sync: Changes don't automatically appear in other sessions

6.7 Testing Summary

We performed unit testing, integration testing, and user acceptance testing. All core features work correctly with no critical bugs. Performance meets our goals for small to medium deployments.

Chapter 7

Conclusion and Future Work

7.1 Project Summary

We successfully built a complete online file management system that meets all our objectives. The system provides secure authentication, hierarchical file organization, flexible sharing, activity logging, and an intuitive web interface.

7.2 Key Achievements

Modern Stack: Using FastAPI and Next.js gave us excellent performance and developer experience.

Security: Bcrypt password hashing, JWT authentication, and recursive permission checking keep user data safe.

Owner-Attributed Logging: Our approach of organizing logs by file owners rather than just actors makes auditing easier.

Flexible Permissions: Combining simple sharing with recursive folder checks gives users control without complexity.

Reliable Operations: Transactional storage management prevents data inconsistencies.

7.3 Lessons Learned

Building this system taught us valuable lessons about full-stack development, database design, security best practices, and user experience. We learned how important it is to handle edge cases, validate inputs thoroughly, and design APIs that are both secure and easy to use.

7.4 Current Limitations

The 10GB storage limit and SQLite database make this system best suited for small to medium deployments. We don't support real-time collaboration, file versioning, or advanced features like automated backups.

7.5 Future Enhancements

Short-term:

- File versioning to track changes over time
- Mobile app for iOS and Android
- Advanced search with content indexing
- Email notifications for sharing and activity

Medium-term:

- Real-time sync across devices
- Collaborative document editing
- Integration with external storage (S3, Google Drive)
- Two-factor authentication

Long-term:

- Migration to PostgreSQL for better scalability
- Distributed storage architecture
- Machine learning for smart file organization
- End-to-end encryption

7.6 Final Remarks

This project demonstrates that building a functional file management system is achievable with modern web technologies. While commercial solutions offer more features, our system provides a solid foundation that can be customized and extended for specific needs. The clean architecture and well-documented code make it suitable for educational purposes and as a starting point for more advanced implementations.

Bibliography

[1] W. Zhang, X. Liu, and Y. Chen, "Secure cloud storage with client-side encryption and distributed key management," *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 2145–2160, 2023.

[2] R. Kumar and P. Patel, "Access control mechanisms for collaborative cloud storage systems: A comparative study," *ACM Transactions on Storage*, vol. 20, no. 2, pp. 1–28, 2024.

[3] L. Chen, M. Wang, and H. Zhang, "Scalable metadata management for large-scale distributed file systems," in *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 2023, pp. 145–160.

[4] J. Liu and F. Wang, "Hybrid deduplication strategies for cloud file systems: Performance and storage optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, pp. 892–907, 2024.

[5] FastAPI Contributors, "Fastapi: Modern, fast (high-performance) web framework for building apis with python," https://fastapi.tiangolo.com/, 2024, accessed: 2024-11-05.

[6] Vercel Inc., "Next.js: The react framework for production," https://nextjs.org/, 2024, accessed: 2024-11-05.

[7] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[8] R. T. Fielding, "Architectural styles and the design of network-based software architectures," *Doctoral dissertation, University of California, Irvine*, 2000.

[9] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media, 2021.

[10] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," *RFC 7519*, 2015.

[11] N. Provos and D. Mazieres, *A Future-Adaptable Password Scheme*, 1999.

[12] SQLAlchemy Contributors, "Sqlalchemy: The python sql toolkit and object relational mapper," https://www.sqlalchemy.org/, 2024, accessed: 2024-11-05.

[13] Meta Platforms Inc., "React: A javascript library for building user interfaces," https://react.dev/, 2024, accessed: 2024-11-05.

[14] Tailwind Labs, "Tailwind css: A utility-first css framework," https://tailwindcss.com/, 2024, accessed: 2024-11-05.