Technical Documentation for Share2Teach Project

1. Project Overview

Share2Teach is a web application built using **SvelteKit** that facilitates user interactions such as file uploads, content management, and user analytics. The project leverages **SQL Server** for database operations and **Docker** for containerization.

Key Technologies

- **Frontend**: Svelte, Tailwind CSS, FlowBite
- **Backend**: SvelteKit API, Node.js
- **Database**: SQL Server on Microsoft Azure
- **Deployment**: Docker, Vercel

Functionality

Front-End

- **Overview**: The front-end serves as the user interface where users can manage and interact with content.
- **Key Features**:
    - User forms for file uploads and document interaction such as rating.
    - Interactive elements such as charts for website analytics, tables, and navigation components that makes the website intuitive.
    - Integration with back-end APIs to fetch and display data dynamically.

Back-End

- **Overview**: The back-end API handles user authentication, file management, database interactions, and returns JSON responses to the front-end.

- **Key Features**:
    - RESTful APIs for user management, file operations, subject management, and more.
    - Interaction with SQL Server for data persistence and CRUD operations.

Data Flow: From Front end to Back end:

- ❖ When a user interacts with the front-end (e.g., submitting a form, clicking a button, or navigating to a different section of the application), an event is triggered. This event is captured by Svelte’s reactive system.
- ❖ The front-end sends an HTTP request (e.g., GET, POST, PUT, DELETE) to the back-end API endpoint.
- ❖ The API endpoint URL, request method, headers (e.g., Content-Type), and any necessary payload (e.g., form data or JSON object) are specified within the request.
- ❖ The back end (built with SvelteKit’s server-side API capabilities) receives the request, validates it, and processes the required action. If the request involves fetching data (e.g., fetching user files or subjects), the back end interacts with the SQL Server database. It executes queries or stored procedures to retrieve the relevant information. The back end assembles the data into a structured JSON response, often including information such as status codes and error messages if needed.
- ❖ The JSON response is sent back to the front-end. It typically contains the data requested (e.g., list of files, user details) or a confirmation message (e.g., file uploaded successfully). Error handling and status codes are also included to help the front-end handle scenarios where something goes wrong.
- ❖ Upon receiving the response, Svelte’s reactive system updates the relevant state or variables associated with the component that triggered the request. The reactivity in Svelte automatically updates the UI based on the changes to these variables. For example, if the response contains a list of files, the component bound to display the files will re-render to show the updated list dynamically. Svelte’s built-in reactivity ensures that any changes to the state (such as new data from the server) immediately reflect on the user interface without needing a full-page refresh.
- ❖ If the API call fails (e.g., network issues, server errors), the front-end catches the error and displays appropriate feedback to the user (e.g., a modal popup or an error message).

Code Quality

Standards

- **Code Style**:
  - **ESLint**: Enforces consistent coding style across JavaScript and TypeScript files.
  - **Prettier**: Auto-formats code to maintain readability.
  - **Indentation**: To make code more readable

Testing

- **API Testing with Postman**:
  - Postman was used extensively to test API endpoints for verifying their functionality and responses.
  - Various requests (GET, POST, PUT, DELETE) were made to simulate real-world usage, and the responses were checked for correctness, including status codes and returned data.
  - Test scenarios included successful cases, as well as handling errors like invalid input or unauthorized access.
- **Console Logs**:
  - Console logs were used in the back-end code to debug and monitor the flow of data, interactions with the database, and error handling.
  - By inspecting console output during API calls, issues were identified and resolved, ensuring the application behaves as expected.
- These methods helped verify the overall functionality and reliability of the API endpoints and interactions between the front-end and back-end components.

Best Practices

- **Class and DAO Structure**:
  - ★ Classes and Data Access Object (DAO) classes were implemented to manage different entities and handle interactions with the database. This approach ensures that the code is organized, modular, and easy to maintain.
  - ★ DAO classes are responsible for database operations, keeping the database logic separate from the business logic.

- **Function Usage**:

- ★ Functions were utilized to encapsulate specific tasks, promoting code reusability and simplifying debugging.
- ★ This helps maintain a clean codebase where each function performs a single, well-defined task.

- **File Organization**:
  - ★ The project files were neatly organized into appropriate folders based on functionality. For example, folders for API routes, database models, utility functions, and front-end components.
  - ★ This structure supports separation of concerns and makes the project easier to navigate and maintain.

Performance

Front-End Optimizations

- **Lazy Loading**: Components are loaded dynamically as needed to improve initial load time.
- **Code Splitting**: Utilizes SvelteKit's built-in features for splitting JavaScript bundles.
- **Data Caching**: Uses client-side caching strategies for data fetched from APIs.

Back-End Optimizations

- **Efficient API Calls**: Reduces the number of database calls per API request using batching techniques.
- **Caching Mechanisms**: Server-side caching for frequently accessed endpoints.

Deployment

Front-End Deployment

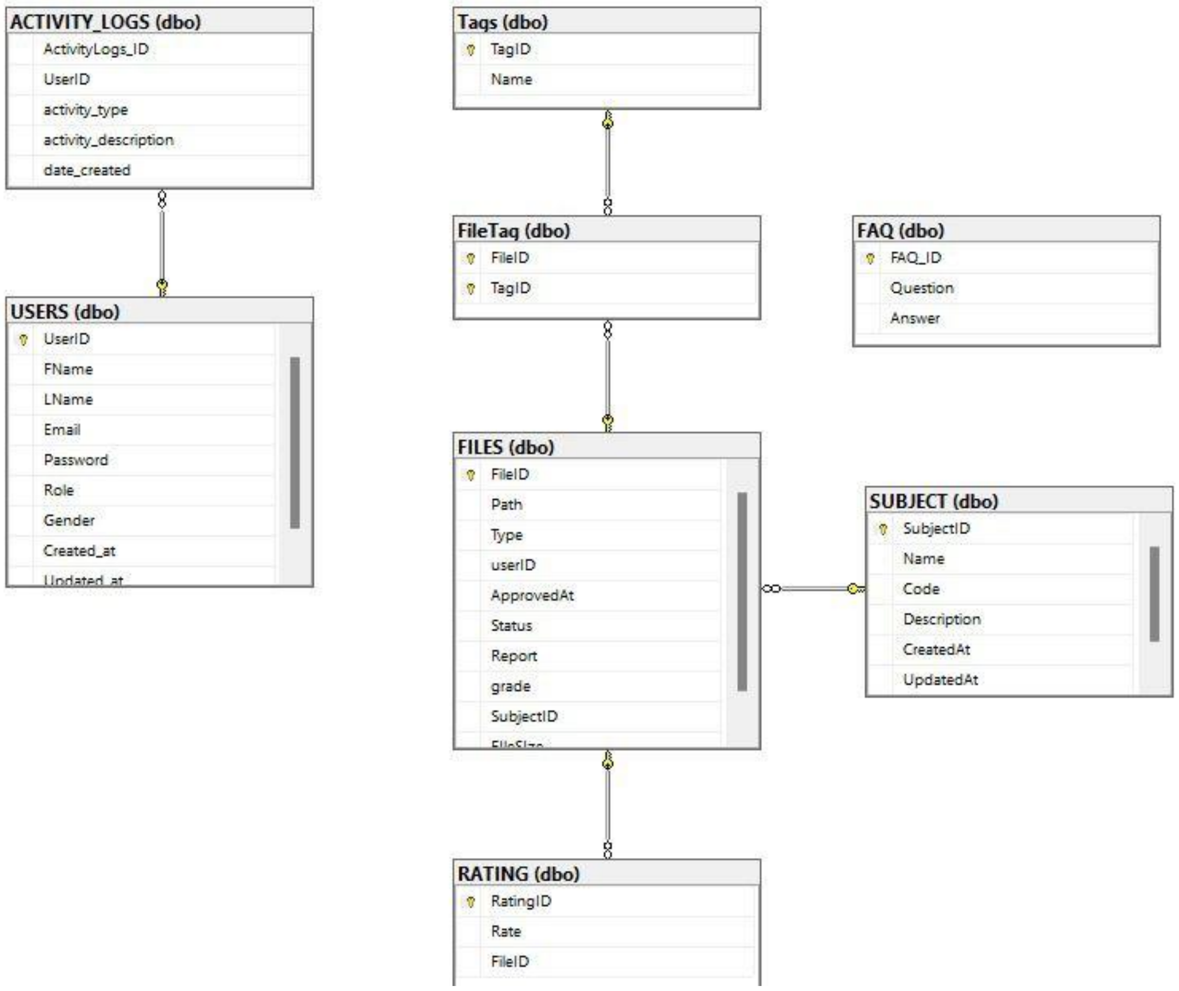
- **Platform**: Deployed on **Vercel Build Process**:
  - o SvelteKit builds the project using Vite, producing optimized static files.
- **Environment Configurations**:
  - o Environment variables managed via .env files for different environments for example database connectionstring and azure blob storage access key

Back-End Deployment

- **Platform**: Hosted using **Docker** and deployed to Vercel
- **Docker Configuration**:
    - The dockerfile sets up the image, installs dependencies, and runs the server.
- **CI/CD Pipeline**:
    - GitHub Actions or another CI/CD tool automates build and deployment.
- **Scaling and Load Balancing**:
    - Configurations for scaling instances and managing traffic using Azure's built-in load balancer.

Schemas and Diagrams

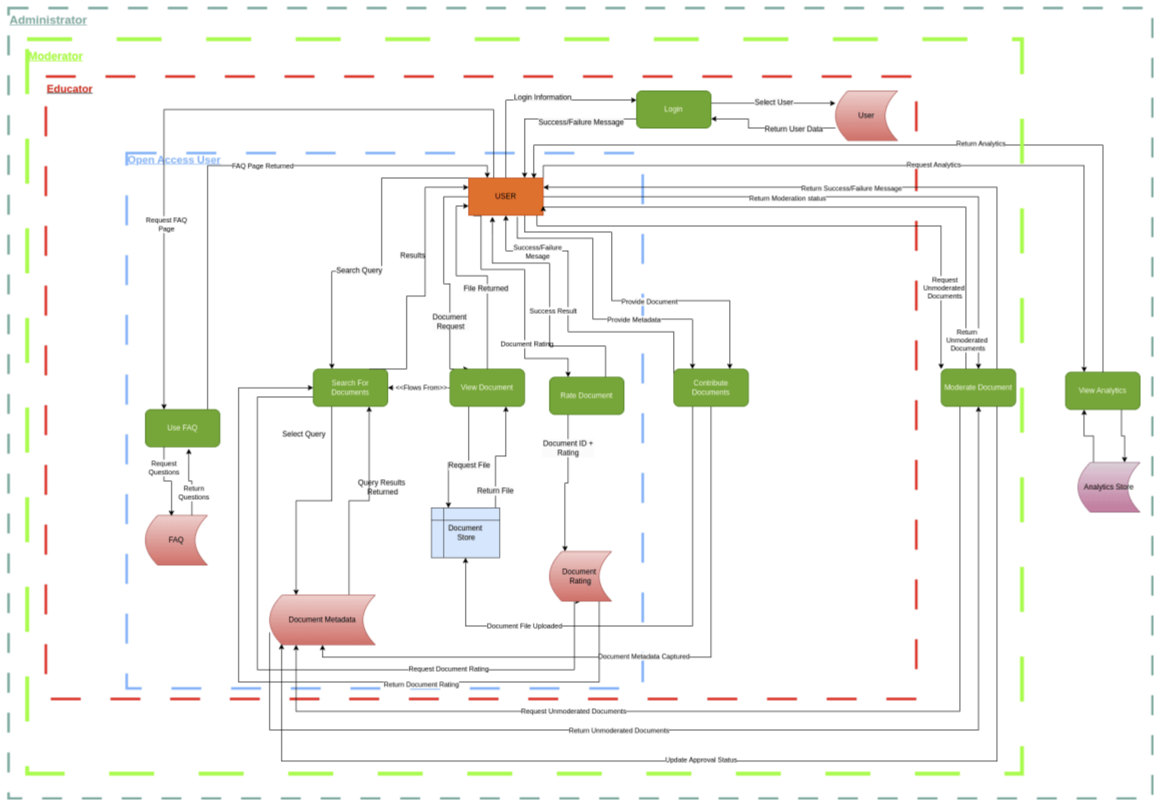
Database Schema



API Design

```
export async function POST({ request, locals }) {
    //upload the file to blob storage
    const formData = await request.formData(); // Parse the JSON body

    const fileDAO = new FileDAO();
    const doc = formData?.get('documents')
    console.log(doc)
    console.log("running entrries")
    console.log(formData.entries())

    if (!doc) {
        console.error('No file uploaded or file is missing in the form data');
        return json({ error: 'No file uploaded' }, { status: 400 });
    }

    const fileBuffer = Buffer.from(await doc?.arrayBuffer());
    const fileName = doc?.name;
    const contentType = doc?.type;
    const fileSize = doc?.size;
    console.log("running buffer")
    console.log(fileBuffer)
    console.log("We have the user ID logged in: "+locals.userID)

    try {
        await fileDAO.connect(); // Ensure you connect to the database
        var file = new File(
            Number(locals.userID),
            formData.get('path'),
            doc?.type,
            null,
            'pending',
            formData.get('report'),
            formData.get('grade'),
            Number(formData.get('subjectID')),
            fileSize
        );
        //console.log(file)
        await fileDAO.uploadFile(file);

        const fileServer = new FileServerDAO();
        await fileServer.connect();
        await fileServer.uploadToBlobStorage(fileBuffer, formData.get('path'),
contentType)

        return json({ message: 'File created successfully' }, {
            status: 200,
            headers: {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*', // Local address
                'Access-Control-Allow-Methods': 'POST, GET, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type',
            }
        });

    } catch (error) {
        console.error('Error creating File:', error);
        return json({ error: 'Internal Server Error' }, { status: 500 });
    }

}
```

Flow Diagrams


Administrator
Moderator
Educator
Open Access User
USER
Login
User
Login Information
Success/Failure Message
Select User
Return User Data
Use FAQ
FAQ
Request FAQ Page
FAQ Page Returned
Request Questions
Return Questions
Search For Documents
Search Query
Results
Select Query
Query Results Returned
Document Metadata
<<Flows From>>
View Document
Document Request
File Returned
Request File
Return File
Document Store
Rate Document
Success/Failure Mesage
Success Result
Document Rating
Document ID + Rating
Document Rating
Request Document Rating
Return Document Rating
Contribute Documents
Provide Document
Provide Metadata
Document File Uploaded
Document Metadata Captured
Moderate Document
Request Unmoderated Documents
Return Unmoderated Documents
Return Success/Failure Message
Return Moderation status
Update Approval Status
View Analytics
Analytics Store
Request Analytics
Return Analytics