Understanding and Implementing Frontend File Operations in Supabase Storage

Supabase Storage provides a robust and scalable solution for storing and serving large files, integrating seamlessly with the Supabase backend. This report outlines how files and folders are managed within Supabase Storage and details the implementation of common file operations—creation (upload), renaming, and deletion—directly from a frontend application using the Supabase JavaScript client library.

1. Understanding Supabase Storage Structure:

Supabase Storage organizes digital assets into a hierarchical structure comprising buckets, files, and folders [1]. Files, which can be any type of media such as images, videos, and documents, are stored outside the database to optimize performance and manage storage efficiently [1]. For security, HTML files are served as plain text, preventing the execution of potentially malicious scripts [1]. Folders serve as organizational tools, allowing developers to structure their files in a manner that aligns with their project's requirements, much like organizing files on a local computer [1]. There are no predefined rules for folder organization, offering flexibility to suit various project needs [1]. Buckets are the top-level containers that hold both files and folders, acting as "super folders" [1]. A key function of buckets is to define distinct security and access rules for the assets they contain [1]. For instance, a project might utilize separate buckets for user-uploaded videos and user profile pictures, each with its own set of access permissions [1]. It is important to note that the naming conventions for files, folders, and buckets must adhere to AWS object key naming guidelines, avoiding the use of unsupported characters [1]. This structured approach allows for intuitive management of cloud-based assets, mirroring familiar file system concepts.

Within Supabase Storage, the organization of files and folders within buckets relies on the concept of object keys [6]. While the user interface, such as the Supabase Dashboard, presents a visual hierarchy of folders, these folders are not distinct physical entities in the underlying storage system [6]. Instead, the appearance of a folder structure is derived from the file's path, or object key, within the bucket [6]. The forward slash character / in a file path is interpreted by the UI to create a directory-like representation [6]. For example, when a file is uploaded with the path images/profile/user1.png, it is stored as an object with the key images/profile/user1.png within the designated bucket. The system then parses this key and visually renders images and profile as folders, providing a familiar navigational experience for users. This method of organization is common in object storage

systems, including Amazon S3, with which Supabase Storage maintains compatibility [1]. This implies that operations that appear to target folders are often implemented by acting on the files whose object keys share a common prefix.

Considering the underlying storage mechanism, folders in Supabase Storage are not stored differently from files [6]. Both are essentially objects within a bucket, identified by their unique keys [6]. The key for a file includes its name and any preceding folder structure, separated by forward slashes. Therefore, a folder does not consume storage space independently; only the files contained within the paths that visually represent the folder contribute to storage usage [6]. The existence of a folder is implied by the presence of files whose paths include the folder name as a prefix. This understanding is crucial when performing operations such as renaming or deleting folders, as these actions will involve manipulating the object keys (file paths) of the files associated with that conceptual folder rather than interacting with a separate folder entity.

2. Frontend File Operations in Supabase Storage:

To interact with Supabase Storage from a frontend application, the Supabase JavaScript client library provides a set of methods for performing file operations.

2.1. Uploading New Files:

The upload method, available through the supabase.storage.from('bucket_name') interface, is used to upload files to a specified bucket [7]. This method requires the destination path within the bucket, including the desired filename, and the file object itself [8]. If the specified folders in the path do not exist, Supabase Storage will implicitly create them based on the object key structure [8]. For smaller files (under 6MB), the standard upload method is suitable, utilizing the multipart/form-data format [7]. While standard uploads can handle files up to 5GB, for larger files, it is recommended to use resumable uploads via the TUS protocol for enhanced reliability [7]. The upload method also offers an upsert option, which, when set to true, will overwrite an existing file at the specified path [7]. However, it is generally advisable to upload to a new path to avoid potential CDN propagation delays that might lead to serving stale content temporarily [7]. Additionally, the content type of the uploaded file can be automatically detected based on the file extension or explicitly set using the contentType option during the upload process [7].

TypeScript

```typescript
import { createClient, SupabaseClient } from '@supabase/supabase-js';

const supabaseUrl = 'YOUR_SUPABASE_URL';
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY';
const supabase: SupabaseClient = createClient(supabaseUrl, supabaseKey);

async function uploadFile(bucketName: string, filePath: string, file: File): Promise<any> {
  const { data, error } = await supabase.storage
    .from(bucketName)
    .upload(filePath, file);

  if (error) {
    console.error('Error uploading file:', error);
    return error;
  } else {
    console.log('File uploaded successfully:', data);
    return data;
  }
}

// Example usage:
const bucket = 'your-bucket-name';
const path = 'images/my-image.png';
const fileInput = document.getElementById('file-upload') as HTMLInputElement;
const selectedFile = fileInput.files?..;

if (selectedFile) {
  const uploadResult = await uploadFile(bucket, path, selectedFile);
  // Handle uploadResult (e.g., display success message or error)
}
```

The ability of the upload method to interpret paths with forward slashes and implicitly create the corresponding structure simplifies the file upload process. Developers do not need to explicitly create folders before uploading files into them; the path specified during the upload operation automatically handles the creation of the visual

folder hierarchy within the storage [8].

2.2. Renaming Files:

Supabase Storage does not provide a dedicated "rename" operation for files [1]. Instead, renaming a file is accomplished by using the move method to relocate the file to a new path within the same bucket, effectively changing its name [1]. The move method requires two parameters: fromPath, which specifies the current path of the file, and toPath, which defines the desired new path, including the new filename [10].

TypeScript

```
import { createClient, SupabaseClient } from '@supabase/supabase-js';

const supabaseUrl = 'YOUR_SUPABASE_URL';
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY';
const supabase: SupabaseClient = createClient(supabaseUrl, supabaseKey);

async function renameFile(bucketName: string, oldPath: string, newPath: string): Promise<any> {
  const { data, error } = await supabase.storage
    .from(bucketName)
    .move(oldPath, newPath);

  if (error) {
    console.error('Error renaming file:', error);
    return error;
  } else {
    console.log('File renamed successfully:', data);
    return data;
  }
}

// Example usage:
const bucket = 'your-bucket-name';
const currentFilePath = 'images/my-image.png';
const newFilePath = 'images/new-image-name.png';
```

```
const renameResult = await renameFile(bucket, currentFilePath, newFilePath);
// Handle renameResult (e.g., display success or error)
```

The absence of a specific rename function underscores the nature of object storage, where a file's identity is intrinsically linked to its path or object key. Renaming, in this context, is conceptually equivalent to creating a new object with the desired name and then deleting the original one. The move operation efficiently handles this process on the server side [9].

2.3. Deleting Files:

To delete one or more files from a Supabase Storage bucket, the remove method is used [1]. This method, accessed via supabase.storage.from('bucket_name').remove(), takes an array of file paths as its argument, specifying the files to be deleted [13].

TypeScript

```
import { createClient, SupabaseClient } from '@supabase/supabase-js';

const supabaseUrl = 'YOUR_SUPABASE_URL';
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY';
const supabase: SupabaseClient = createClient(supabaseUrl, supabaseKey);

async function deleteFile(bucketName: string, filePath: string): Promise<any> {
  const { data, error } = await supabase.storage
    .from(bucketName)
    .remove([filePath]);

  if (error) {
    console.error('Error deleting file:', error);
    return error;
  } else {
    console.log('File deleted successfully:', data);
    return data;
  }
}
```

```
// Example usage:
const bucket = 'your-bucket-name';
const filePathToDelete = 'images/my-image.png';

const deleteResult = await deleteFile(bucket, filePathToDelete);
// Handle deleteResult (e.g., update UI or display confirmation)
```

It is crucial to understand that once an object (file) is deleted using the remove method, it is permanently removed from the storage and cannot be recovered [12]. Therefore, it is important to implement appropriate confirmation mechanisms in the frontend to prevent accidental data loss.

3. Frontend Folder Operations in Supabase Storage:

Since folders in Supabase Storage are conceptual and based on file paths, operations like renaming and deleting folders require manipulating the paths of the files they contain.

3.1. Renaming Folders:

Renaming a folder in Supabase Storage involves iterating through all the files whose paths begin with the old folder name and then moving each of these files to a new path that reflects the new folder name [6]. This process effectively updates the conceptual folder structure.

TypeScript

```
import { createClient, SupabaseClient } from '@supabase/supabase-js';

const supabaseUrl = 'YOUR_SUPABASE_URL';
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY';
const supabase: SupabaseClient = createClient(supabaseUrl, supabaseKey);

interface FileObject {
  name: string;
}

async function renameFolder(bucketName: string, oldFolderPath: string, newFolderPath: string):
```

```
Promise<any> {
  try {
    // List all files in the old folder
    const { data: files, error: listError } = await supabase.storage
      .from(bucketName)
      .list(oldFolderPath);

    if (listError) {
      console.error('Error listing files in folder:', listError);
      return listError;
    }

    if (files && files.length > 0) {
      for (const file of files as FileObject) {
        const oldPath = `${oldFolderPath}/${file.name}`;
        const newPath = `${newFolderPath}/${file.name}`;
        // Move each file to the new path
        const { error: moveError } = await supabase.storage
          .from(bucketName)
          .move(oldPath, newPath);

        if (moveError) {
          console.error(`Error moving file ${oldPath} to ${newPath}:`, moveError);
          return moveError;
        }
        console.log(`Moved ${oldPath} to ${newPath}`);
      }
    }
    console.log(`Folder ${oldFolderPath} renamed to ${newFolderPath}`);
    return { message: `Folder ${oldFolderPath} renamed to ${newFolderPath}` };
  } catch (error) {
    console.error('Error renaming folder:', error);
    return error;
  }
}

// Example usage:
const bucket = 'your-bucket-name';
const currentFolderPath = 'images/old-folder';
```

```
const newFolderPath = 'images/new-folder';

const renameFolderResult = await renameFolder(bucket, currentFolderPath, newFolderPath);
// Handle renameFolderResult
```

Renaming a folder is not an atomic operation but rather a series of individual file move operations. This implies that for folders containing a large number of files, the process might take some time, and there is a possibility of partial failure if some of the move operations encounter errors.

3.2. Deleting Folders:

Deleting a folder conceptually in Supabase Storage requires identifying and deleting all files whose paths have the folder name as a prefix [1].

TypeScript

```
import { createClient, SupabaseClient } from '@supabase/supabase-js';

const supabaseUrl = 'YOUR_SUPABASE_URL';
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY';
const supabase: SupabaseClient = createClient(supabaseUrl, supabaseKey);

interface FileObject {
  name: string;
}

async function deleteFolder(bucketName: string, folderPath: string): Promise<any> {
  try {
    // List all files in the folder
    const { data: files, error: listError } = await supabase.storage
      .from(bucketName)
      .list(folderPath);

    if (listError) {
      console.error('Error listing files in folder:', listError);
```

```
    return listError;
  }

  if (files && files.length > 0) {
    const filePathsToDelete = (files as FileObject).map(file => `${folderPath}/${file.name}`);
    // Delete all files in the folder
    const { data: deleteData, error: deleteError } = await supabase.storage
      .from(bucketName)
      .remove(filePathsToDelete);

    if (deleteError) {
      console.error('Error deleting files in folder:', deleteError);
      return deleteError;
    }
    console.log('Folder and its contents deleted successfully:', deleteData);
    return deleteData;
  } else {
    console.log('Folder is empty or does not exist.');
    return { message: 'Folder is empty or does not exist.' };
  }
 } catch (error) {
  console.error('Error deleting folder:', error);
  return error;
 }
}

// Example usage:
const bucket = 'your-bucket-name';
const folderToDelete = 'images/old-folder';

const deleteFolderResult = await deleteFolder(bucket, folderToDelete);
// Handle deleteFolderResult
```

Similar to renaming, deleting a folder involves a bulk operation on the files within it. This reinforces the understanding that folders are logical constructs based on file paths in the object storage system.

4. Implementing File Operations in the Frontend (Best Practices):

To ensure proper and error-free implementation of file operations in the frontend,

several best practices should be followed.

4.1. Initializing the Supabase Client:

The first step is to correctly initialize the Supabase client in your frontend application using your project URL and anonymous API key [7]. This client instance will be used to interact with Supabase Storage.

TypeScript

```
import { createClient, SupabaseClient } from '@supabase/supabase-js';

const supabaseUrl = 'YOUR_SUPABASE_URL';
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY';
const supabase: SupabaseClient = createClient(supabaseUrl, supabaseKey);
```

4.2. Handling Asynchronous Operations:

The methods provided by the Supabase client library for storage operations are asynchronous and return Promises [15]. It is essential to handle these operations using async/await syntax to ensure that the code execution waits for the completion of the storage operation before proceeding. This helps in managing the flow of data and handling potential errors effectively, as demonstrated in the code snippets above.

4.3. Error Handling:

Implementing robust error handling is crucial for a smooth user experience [7]. Always wrap your storage operation calls within try...catch blocks or check the error object returned by the Supabase client to catch potential issues such as network connectivity problems, insufficient permissions, or file not found errors. Display informative error messages to the user to help them understand and resolve any problems.

4.4. Security Considerations (RLS Policies):

Security is paramount when dealing with file storage. Supabase Storage leverages Row Level Security (RLS) policies defined in the backend to control access to buckets and objects [1]. These policies determine who can perform which actions (upload,

rename, delete, download) on specific resources. For example, you can define a policy that allows only authenticated users to upload files to a particular bucket or restrict the deletion of files to their respective owners [12]. Ensure that your frontend implementation respects these policies by only allowing authorized users to initiate file operations. The frontend code initiates the requests, but the actual authorization and access control are enforced by the RLS policies configured in the Supabase backend. Therefore, establishing comprehensive and appropriate RLS policies is fundamental to securing your storage.

5. Conclusion:

Performing file operations in Supabase Storage from a frontend application involves understanding the underlying storage structure and utilizing the methods provided by the Supabase JavaScript client library. While files are managed directly through dedicated functions for upload, rename (move), and delete, folder operations are conceptual and require manipulating the paths of the files contained within those folders. By adhering to best practices such as proper client initialization, handling asynchronous operations, implementing error handling, and leveraging Row Level Security policies, developers can seamlessly integrate file management capabilities into their frontend applications while maintaining data integrity and security.

Works cited

1. Storage Quickstart | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/guides/storage/quickstart
2. Storage | Supabase - Vercel, accessed March 18, 2025, https://zone-www-dot-9obe9a1tk-supabase.vercel.app/docs/guides/storage
3. Storage | Supabase Docs - Vercel, accessed March 18, 2025, https://docs-k54u0gmi1-supabase.vercel.app/docs/guides/storage
4. Supabase self-hosted storage guide — Restack, accessed March 18, 2025, https://www.restack.io/docs/supabase-knowledge-supabase-self-hosted-storage
5. Supabase tutorial guide — Restack, accessed March 18, 2025, https://www.restack.io/docs/supabase-knowledge-supabase-tutorial-guide
6. Buckets and auto folders : r/Supabase - Reddit, accessed March 18, 2025, https://www.reddit.com/r/Supabase/comments/1foxl0f/buckets_and_auto_folders/
7. Standard Uploads | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/guides/storage/uploads/standard-uploads
8. JavaScript: Upload a file | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/reference/javascript/storage-from-upload
9. JS Client library to interact with Supabase Storage - GitHub, accessed March 18, 2025, https://github.com/supabase/storage-js

10. JavaScript: Move an existing file | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/reference/javascript/storage-from-move
11. Storage | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/guides/storage
12. Delete Objects | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/guides/storage/management/delete-objects
13. JavaScript: Delete files in a bucket | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/reference/javascript/storage-from-remove
14. How to secure file uploads in Supabase storage? - bootstrapped.app, accessed March 18, 2025, https://bootstrapped.app/guide/how-to-secure-file-uploads-in-supabase-storage
15. Uploading files to storage - Build Real-time Web Apps with Supabase and JavaScript, accessed March 18, 2025, https://app.studyraid.com/en/read/12469/403033/uploading-files-to-storage
16. Storage Buckets | Supabase Docs, accessed March 18, 2025, https://supabase.com/docs/guides/storage/buckets/fundamentals
17. Authenticated Supabase Storage With Policies | by Yogesh Manikkavasagam | Medium, accessed March 18, 2025, https://medium.com/@yogeshmulecraft/authenticated-supabase-storage-with-policies-db65bbc9fa63