Design Document: Authentication and Department Management

Overview

This document outlines the design and functionality of the user authentication and department management system. The system enables the creation and signing in of users, assigning them to departments within a company structure in a Firebase Firestore database. The key modules involved are `authentication.js` and `page.js`. The design leverages Firebase Authentication for user management and Firestore for company and department data.

System Components

1. **Firebase Setup**
   The system is configured with Firebase, which provides the backend services for authentication and Firestore database access.
2. **Core Functionalities**
   - **User Creation and Authentication**: Users can sign up with email and password, along with additional data such as name, phone, role, and department.
   - **Department Assignment**: New users are added to specific departments based on a selection made during signup.
   - **Error Handling**: The system provides feedback on common errors (e.g., invalid email, email already in use).

Detailed Code Breakdown

`authentication.js`

This file handles the main backend logic for authentication and interacting with Firestore to manage users and departments.

1. Fetching Departments

```
export const getDepartments = async () => {
  try {
    const departmentsRef = collection(db,
"Company/mh3VZ5IrZjubXUCZL381/Departments");
    const departmentsSnapshot = await getDocs(departmentsRef);
    const departments = departmentsSnapshot.docs.map(doc =>
doc.data().name);
```

```
    return departments;
  } catch (error) {
    console.error("Error fetching departments:", error);
    throw error;
  }
};
```

- **Purpose**: Fetches the list of department names from Firestore. These department names are displayed in a dropdown during user creation.
- **Process**:
    - Retrieves documents in the `Departments` collection.
    - Maps each document to extract the `name` field and returns a list of department names.

2. Creating a New User

```
export const createUser = async (email, password, additionalData) => {
  try {
    const userCredential = await createUserWithEmailAndPassword(auth,
email, password);
    const user = userCredential.user;

    const employeeRef = doc(db,
"Company/mh3VZ5IrZjubXUCZL381/Employees", user.uid);
    await setDoc(employeeRef, {
      name: additionalData.name,
      email: user.email,
      phone: additionalData.phone,
      role: additionalData.role,
      departments: additionalData.departments,
      companyName: additionalData.companyName,
    });

    if (additionalData.departments) {
      const departmentsRef = collection(db,
"Company/mh3VZ5IrZjubXUCZL381/Departments");
      const q = query(departmentsRef, where("name", "==",
additionalData.departments));
      const querySnapshot = await getDocs(q);
      if (!querySnapshot.empty) {
        const departmentDoc = querySnapshot.docs[0];
        const departmentRef = doc(db,
`Company/mh3VZ5IrZjubXUCZL381/Departments`, departmentDoc.id);
        await updateDoc(departmentRef, {
          users: arrayUnion(employeeRef),
        });
      } else {
        throw new Error("No matching department found");
      }
    }
  } catch (error) {
    console.error("Error creating user or adding to department:",
error);
    return error.code;
  }
};
```

- **Purpose**: Creates a new user in Firebase Authentication and saves additional data to Firestore under the `Employees` collection. It also updates the corresponding department with the new user.
- **Process**:
  - Calls `createUserWithEmailAndPassword` to register a new user.
  - Saves user details such as name, phone, role, department, and company name in Firestore.
  - Searches for the selected department in Firestore and updates its `users` array to include a reference to the new employee.

3. Signing In a User

```
export const signInUser = async (email, password, companyName) => {
  try {
    const userCredential = await signInWithEmailAndPassword(auth,
email, password);
    const user = userCredential.user;

    const userDocRef = doc(db,
"Company/mh3VZ5IrZjubXUCZL381/Employees", user.uid);
    const userDoc = await getDoc(userDocRef);
    if (!userDoc.exists()) {
      throw new Error("No user found for the given company.");
    }

    const userData = userDoc.data();
    if (userData.companyName !== companyName) {
      throw new Error("Company name does not match.");
    }

    return userData;
  } catch (error) {
    console.error("Error signing in:", error);
    throw error;
  }
};
```

- **Purpose**: Authenticates a user by their email and password, then checks if the user belongs to the correct company.
- **Process**:
    - Calls `signInWithEmailAndPassword` to authenticate the user.
    - Retrieves the user document from Firestore and verifies the company name.

`page.js`

This file contains the frontend logic, handling user interaction such as signing up and signing in, and displaying available departments.

1. State Management and Department Fetching

```
const [departments, setDepartments] = useState<string[]>([]); //
Departments list state
const [selectedDepartment, setSelectedDepartment] = useState("");
const [errorMessage, setErrorMessage] = useState("");

useEffect(() => {
  const fetchDepartments = async () => {
    try {
      const departmentsList = await getDepartments();
      if (departmentsList.length > 0) {
        setDepartments(departmentsList);
        setSelectedDepartment(departmentsList[0]);
      } else {
        setErrorMessage("No departments available.");
      }
    } catch (error) {
      setErrorMessage("Failed to fetch departments");
    }
  };
  fetchDepartments();
}, []);
```

- **Purpose**: Fetches department names upon page load and displays them in a dropdown for selection during user sign-up.
- **Process**:
  - Calls `getDepartments()` and sets the `departments` state with the fetched department names.
  - Handles errors by updating `errorMessage` if the fetch fails.

2. Handling User Sign-Up

```
const handleSignUp = async () => {
  try {
    const additionalData = {
      name,
      phone,
      departments: selectedDepartment,
      role,
      companyName,
    };
    const result = await createUser(email, password, additionalData);
    if (result) {
      handleErrorMessages(result);
    } else {
      alert("User created successfully!");
    }
  } catch (error) {
    setErrorMessage("Error creating user");
  }
};
```

- **Purpose**: Sends sign-up data to `createUser()` to create a new employee and associate them with a department.
- **Process**:
  - Collects form data and passes it to `createUser()`.
  - Displays error messages in case of sign-up failure.

3. Handling User Sign-In

```
const handleSignIn = async () => {
  try {
    const result = await signInUser(email, password, companyName);
    alert("User signed in successfully!");
  } catch (error: any) {
    handleErrorMessages(error.message);
  }
};
```

- **Purpose**: Sends login credentials to `signInUser()` for authentication and company name verification.
- **Process**:
  - Verifies login details and shows an error if the login fails (e.g., incorrect company name).

Error Handling

Errors are caught and handled using `try...catch` blocks in both `authentication.js` and `page.js`. For user sign-up, potential Firebase errors (e.g., invalid email, weak password) are mapped to user-friendly messages, which are displayed in the UI. If department fetching fails, an appropriate message is shown.

Conclusion

This system allows for the secure creation of employees within an organizational structure, ensuring that new users are linked to their appropriate departments. The design is scalable, allowing for multiple departments and employees, with error handling in place for common issues during authentication and user management.

File Upload and Management System

Overview

The system enables users to upload files to specific directories in Firebase Storage and manage file references in Firestore. Users can select the upload target (either a department or a buyer's quote) within a company's organizational structure. The system also handles file replacement with user confirmation, lists all files (including those in subdirectories), and allows for file deletion while updating Firestore accordingly.

System Components

Firebase Setup

The system is configured with Firebase, utilizing:

- **Firebase Storage** for storing uploaded files.
- **Firestore** for managing metadata and associations with companies, departments, buyers, and quotes.

Core Functionalities

- **File Upload**: Users can upload files to specific directories corresponding to departments or buyer quotes.
- **Directory Management**: Users can specify custom directories and select companies, departments, buyers, and quotes for precise file placement.
- **File Replacement**: The system checks for existing files with the same name and prompts users for confirmation before replacing them.
- **File Listing**: Displays all files in Firebase Storage, including those in subdirectories.
- **File Deletion**: Allows users to delete files from Firebase Storage and updates Firestore to remove references.
- **Data Fetching**: Retrieves data for companies, departments, buyers, and quotes from Firestore to populate selection options.

Detailed Code Breakdown

UploadPage Component

This component handles the main frontend logic, managing user interactions such as selecting files, directories, upload targets, and displaying files.

State Management and Data Fetching

```
// State declarations
const [file, setFile] = useState<File | null>(null);
const [directory, setDirectory] = useState<string>("");
const [companies, setCompanies] = useState<Company[]>([]);
const [departments, setDepartments] = useState<Department[]>([]);
const [buyers, setBuyers] = useState<Buyer[]>([]);
const [quotes, setQuotes] = useState<Quote[]>([]);
const [selectedCompany, setSelectedCompany] = useState<Company |
null>(null);
const [selectedDepartment, setSelectedDepartment] =
useState<Department | null>(null);
const [selectedBuyer, setSelectedBuyer] = useState<Buyer |
null>(null);
const [selectedQuote, setSelectedQuote] = useState<Quote |
null>(null);
const [fileList, setFileList] = useState<FileItem[]>([]);
const [uploadTarget, setUploadTarget] = useState<"Department" |
"Buyer">("Department");
```

1. **Purpose**: Manages the state of the component, including the selected file, directory, companies, departments, buyers, quotes, and upload target.
   Process:
   - Uses `useState` hooks to manage local state.
   - Utilizes `useEffect` hooks to fetch data from Firestore when dependencies change (e.g., selected company, upload target).
   - Fetches companies on component mount and updates departments or buyers based on the selected company and upload target.

Handling File Selection and Directory Input

```
// Handle directory input change
const handleDirectoryChange = (e: React.ChangeEvent<HTMLInputElement>)
=> {
  setDirectory(e.target.value);
};

// Handle file selection
const handleFileChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  if (e.target.files && e.target.files[0]) {
    setFile(e.target.files[0]);
  }
};
```

2. **Purpose**: Updates the state when the user selects a directory or file to upload.
   Process:
   - `handleDirectoryChange` updates the `directory` state when the user inputs a directory.
   - `handleFileChange` updates the `file` state when the user selects a file.

Handling File Upload with Replacement and Warning

```
const handleUpload = async () => {
  if (!file || !selectedCompany) {
    alert("Please select a file and company first.");
    return;
  }

  // Validation based on upload target
  if (uploadTarget === "Department" && !selectedDepartment) {
    alert("Please select a department.");
    return;
  }

  if (uploadTarget === "Buyer") {
    if (!selectedBuyer) {
      alert("Please select a buyer.");
      return;
    }
    if (!selectedQuote) {
      alert("Please select a quote.");
      return;
    }
  }

  const dir = directory ? `${directory}/` : "";
  let storagePath = `${dir}${file.name}`;

  // Determine storage path based on upload target
  if (uploadTarget === "Department") {
    storagePath =
`Company/${selectedCompany.id}/Departments/${selectedDepartment!.id}/$
{dir}${file.name}`;
```

```
  } else if (uploadTarget === "Buyer") {
    storagePath =
`Company/${selectedCompany.id}/Buyers/${selectedBuyer!.id}/Quotes/${se
lectedQuote!.id}/${dir}${file.name}`;
  }

  const storageRef = ref(storage, storagePath);

  try {
    // Check if file exists
    const existingFileSnapshot = await getDownloadURL(storageRef)
      .then(() => true)
      .catch((error) => {
        if (error.code === "storage/object-not-found") {
          return false;
        } else {
          throw error;
        }
      });

    if (existingFileSnapshot) {
      const confirmReplace = window.confirm(
        `A file named "${file.name}" already exists. Do you want to
replace it?`
      );

      if (!confirmReplace) {
        alert("Upload canceled.");
        return;
      }

      // Delete existing file
      await deleteObject(storageRef);
      console.log(`Deleted existing file: ${file.name}`);

      // Remove old file reference from file list
      setFileList((prevList) =>
        prevList.filter((file) => file.fullPath !==
storageRef.fullPath)
      );
    }

    // Proceed with the upload
    const uploadTask = uploadBytesResumable(storageRef, file);

    uploadTask.on(
      "state_changed",
      (snapshot) => {
        // Handle progress
        const progress = (snapshot.bytesTransferred /
snapshot.totalBytes) * 100;
        console.log("Upload is " + progress + "% done");
      },
      (error) => {
        console.error("Upload failed", error);
      },
      async () => {
        // Handle successful upload
        const downloadURL = await
getDownloadURL(uploadTask.snapshot.ref);
        setFileList((prevList) => [
          ...prevList,
          { name: file!.name, url: downloadURL, fullPath:
storageRef.fullPath },
        ]);

        // Add file reference to Firestore
        if (uploadTarget === "Department") {
          await addFileToFirestore(
            selectedCompany!.id,
            "Departments",
            selectedDepartment!.id,
            null,
            file!.name,
            storageRef.toString()
          );
        } else if (uploadTarget === "Buyer") {
          await addFileToFirestore(
            selectedCompany!.id,
            "Buyers",
            selectedBuyer!.id,
            selectedQuote!.id,
            file!.name,
            storageRef.toString()
          );
        } else {
          throw new Error("Invalid upload target");
        }
      }
    );
  } catch (error) {
    console.error("Error handling upload", error);
  }
};
```

3. **Purpose**: Manages the file upload process, including checking for existing files, prompting for replacement, uploading the file, and updating Firestore.
   Process:
   - Validates that all necessary selections are made.
   - Constructs the storage path based on the selected options.
   - Checks if a file with the same name already exists at the target location.
   - Prompts the user for confirmation if a file exists.
   - Deletes the existing file if confirmed.
   - Uploads the new file using `uploadBytesResumable`.
   - Updates the `fileList` state and Firestore with the new file reference.

Listing Files from Firebase Storage

```
// Recursive function to list all files, including those in
subdirectories
const listFilesRecursive = async (dirRef: StorageReference) => {
  const res = await listAll(dirRef);
  const allFiles = await Promise.all(
    res.items.map(async (itemRef) => {
      const downloadURL = await getDownloadURL(itemRef);
      return { name: itemRef.name, url: downloadURL, fullPath:
itemRef.fullPath };
    })
  );

  // Recursively get files from subdirectories
  for (const folderRef of res.prefixes) {
    const folderFiles = await listFilesRecursive(folderRef);
    allFiles.push(...folderFiles);
  }

  return allFiles;
};

// Function to list files from Firebase Storage
const listFiles = async () => {
  const listRef = ref(storage, "/"); // Start from the root

  try {
    const files = await listFilesRecursive(listRef);
    setFileList(files);
  } catch (error) {
    console.error("Error listing files", error);
  }
};

// List files when component mounts
useEffect(() => {
  listFiles();
}, []);
```

4. **Purpose**: Retrieves and displays all files from Firebase Storage, including those in subdirectories.
   Process:
   - Defines a recursive function `listFilesRecursive` to traverse directories.
   - Fetches files and folders using `listAll`.
   - Retrieves download URLs for files.
   - Updates the `fileList` state with all files found.

Deleting Files from Firebase Storage and Updating Firestore

```
// Function to delete a file from Firebase Storage
const deleteFile = async (fileFullPath: string) => {
  const fileRef = ref(storage, fileFullPath);

  try {
    // Delete the file from Firebase Storage
    await deleteObject(fileRef);
    console.log(`Deleted file from Storage: ${fileFullPath}`);

    const pathParts = fileFullPath.split("/");
    const companyId = pathParts[1];

    // If deleting a Department file
    if (pathParts[2] === "Departments") {
      const departmentId = pathParts[3];
      const fileName = pathParts[pathParts.length - 1];

      const fileDocRef = doc(
        db,
        "Company",
        companyId,
        "Departments",
        departmentId,
        "files",
        fileName
      );
      await deleteDoc(fileDocRef);
      console.log(`Deleted Firestore document: ${fileName}`);

    } else if (pathParts[2] === "Buyers") {
      // If deleting a Buyer's Quote file
      const buyerId = pathParts[3];
      const quoteId = pathParts[5];
      const filePath = "gs://your-app.appspot.com/" + fileFullPath;

      const quoteDocRef = doc(
        db,
        "Company",
        companyId,
        "Buyers",
        buyerId,
        "Quotes",
        quoteId
      );

      // Remove the file path from the PDFs array
      await updateDoc(quoteDocRef, {
        PDFS: arrayRemove(filePath),
      });
      console.log(`Removed file path from PDFs array in Firestore:
${filePath}`);
    }

    // Update the file list
    setFileList((prevList) => prevList.filter((file) => file.fullPath
!== fileFullPath));
  } catch (error) {
    console.error("Error deleting file or updating Firestore", error);
  }
};
```

5. **Purpose**: Deletes a file from Firebase Storage and removes its reference from Firestore.
   Process:
   - Identifies the file to delete using its full path.
   - Deletes the file from Firebase Storage.
   - Parses the file path to determine if it's associated with a department or buyer.
   - For department files, deletes the corresponding Firestore document.
   - For buyer quote files, removes the file path from the `PDFS` array in the quote document.
   - Updates the `fileList` state to reflect the deletion.

Adding File References to Firestore

```
const addFileToFirestore = async (
  companyId: string,
  collectionType: "Departments" | "Buyers",
  subCollectionId: string,
  subSubCollectionId: string | null,
  fileName: string,
  filePath: string
) => {
  if (collectionType === "Departments") {
    // For Departments, add a new document to the "files" collection
    const filesDocRef = doc(
      db,
      "Company",
      companyId,
      "Departments",
      subCollectionId,
      "files",
      fileName
    );
    await setDoc(filesDocRef, { fileName, filePath });
  } else if (collectionType === "Buyers") {
    if (!subSubCollectionId) {
      throw new Error("Quote ID is required when uploading to
Buyers.");
    }
    // For Buyers, append the filePath to the PDFs array in the Quote
document
    const quoteDocRef = doc(
      db,
      "Company",
      companyId,
      "Buyers",
      subCollectionId,
      "Quotes",
      subSubCollectionId
    );

    // Use arrayUnion to add the filePath to the PDFs array
    await updateDoc(quoteDocRef, {
      PDFS: arrayUnion(filePath),
    });
  } else {
    throw new Error("Invalid collection type");
  }
};
```

6. **Purpose**: Updates Firestore with references to the uploaded files.
   Process:
   - Determines whether to update departments or buyers based on `collectionType`.
   - For departments, creates a new document in the `files` subcollection.
   - For buyers, adds the file path to the `PDFS` array in the quote document.

Error Handling

Errors are handled using `try...catch` blocks throughout the component:

- **File Upload**: Catches errors during the upload process, including file existence checks and storage operations.
- **File Deletion**: Handles errors when deleting files from Firebase Storage and updating Firestore.
- **Data Fetching**: Catches errors when fetching companies, departments, buyers, and quotes from Firestore.
- **User Feedback**: Alerts and console logs are used to inform the user of success or failure of operations.

Conclusion

This file upload and management system provides a robust solution for uploading files to Firebase Storage with precise control over the destination directory. It supports organizational structures by allowing uploads to specific departments or buyer quotes within a company. The system ensures data integrity by managing file references in Firestore and provides a user-friendly interface for file management, including listing and deleting files. Error handling mechanisms are in place to enhance reliability and user experience.