Source code

github link: https://github.com/snehildiv/reportapp

TypeScript

```typescript
import jsPDF from 'jspdf';
import axios from 'axios';
import React, { useState } from 'react';
import { v4 as uuidv4 } from 'uuid';
import firebase from 'firebase/compat/app';
import 'firebase/compat/firestore';

import { firebaseConfig } from './firebaseConfig';

// Initialize Firebase
firebase.initializeApp(firebaseConfig);
const firestore = firebase.firestore();

interface Report {
  id: string;
  fileName: string;
  code: string;
  executionResult: string;
}

const App: React.FC = () => {
  const [files, setFiles] = useState<File[]>([]);
  const [reports, setReports] = useState<Report[]>([]);

  const handleFileUpload = (event: React.ChangeEvent<HTMLInputElement>) => {
    const uploadedFiles = Array.from(event.target.files || []);
    setFiles((prevFiles) => [...prevFiles, ...uploadedFiles]);
  };

  const generateReport = async () => {
    const reportPromises = files.map(async (file) => {
      const code = await readFile(file);
      const executionResult = await executeCode(code);

      return {
        id: uuidv4(),
        fileName: file.name,
        code,
        executionResult,
      };
    });

    const generatedReports = await Promise.all(reportPromises);
    setReports(generatedReports);
    // Store the reports in Firebase Firestore
    generatedReports.forEach((report) => {
      firestore.collection('reports').doc(report.id).set(report);
    });
  };

  const readFile = (file: File): Promise<string> => {
    return new Promise<string>((resolve, reject) => {
      const reader = new FileReader();
      reader.onload = () => {
        resolve(reader.result as string);
      };
      reader.onerror = reject;
      reader.readAsText(file);
    });
  };

  const executeCode = async (code: string): Promise<string> => {
    // Make an API call to execute the code using Piston API or any other similar service
    // Return the execution result as a string
    // You can use Axios for making API requests
    // Example using Axios:
    const response = await axios.post('https://piston-api.example.com/execute', { code });
    return response.data;
  };

  const downloadReport = (report: Report) => {
    // Convert report data into PDF format and initiate download
    // You can use a library like jsPDF or pdfmake for generating PDFs
    // Example: jsPDF implementation
    const doc = new jsPDF();
    doc.text(`File: ${report.fileName}`, 10, 10);
    doc.text(`Report Generated: ${new Date().toLocaleString()}`, 10, 20);
    doc.text(`Code:\n${report.code}`, 10, 30);
    doc.text(`Execution Result:\n${report.executionResult}`, 10, 50);
    doc.save(`${report.fileName}_report.pdf`);
  };

  return (
    <div>
      <header>
        <img src="C:\Users\divya\code-report-app\src\logo.jpeg" alt="Logo" />
      </header>
      <input type="file" multiple onChange={handleFileUpload} />
      <button onClick={generateReport}>Generate Report</button>
      <table>
        <thead>
          <tr>
            <th>File Name</th>
            <th>Report</th>
            <th>Action</th>
          </tr>
        </thead>
        <tbody>
          {reports.map((report) => (
            <tr key={report.id}>
              <td>{report.fileName}</td>
              <td>{report.executionResult}</td>
              <td>
                <button onClick={() => downloadReport(report)}>Download</button>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
};

export default App;
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

```
import React from 'react';
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Css

```
.App {
  text-align: center;
}

.App-logo {
  height: 40vmin;
  pointer-events: none;
}

@media (prefers-reduced-motion: no-preference) {
  .App-logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}

```

```
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
    monospace;
}
```

Video demo: https://drive.google.com/file/d/10-gNnwuoTgkrHUzIZ6dZe_ciE8jXfVEr/view?usp=drivesdk

Documentation of WebApp

1. Set up Firebase project:
   - Create a new project on the Firebase console (https://console.firebase.google.com/).
   - Enable Firebase Authentication and Firestore for your project.
   - Configure Firebase rules to secure access to your data.
   - Obtain the Firebase configuration values (API keys, project ID, etc.) for later use.
2. Set up React.js project:
   - Set up a new React.js project using Create React App (CRA) or your preferred tool.
   - Install required dependencies, including `firebase`, `react-router-dom`, and any additional libraries you plan to use.
   - Create the necessary components and routes for your web app, such as file upload, report display, etc.
3. Configure Firebase in your React app:
   - Import the Firebase SDK and initialize it with the Firebase configuration obtained in Step 1.
   - Set up Firebase Authentication to allow user registration and login.
   - Set up Firebase Firestore to store user data and reports.
4. Implement user authentication:
   - Create authentication components and screens, such as a registration form and a login form.
   - Use Firebase Authentication to handle user registration, login, and logout.
   - Manage user sessions and authentication state in your React app.
5. Implement file upload functionality:
   - Create a file upload component that allows users to select and upload code files.
   - Use Firebase Cloud Storage to store the uploaded files.
   - Save the file metadata and reference to the user's Firestore document for later retrieval.
6. Generate code file reports:
   - Set up a backend process or use a code analysis library to generate reports for the uploaded files.
   - Depending on your specific requirements, you can use tools like ESLint, TypeScript Compiler API, or custom analysis scripts to extract relevant information from the code files.
   - Store the generated reports in Firestore, associating them with the corresponding user and file metadata.
7. Display reports to users:

- Create a component to fetch and display the reports for each user.
- Use Firebase Firestore to retrieve the user's file metadata and associated reports.
- Render the reports in a readable format within your React app.

8. Additional features and enhancements:
   - Implement user access control to ensure users can only access their own reports.
   - Enhance the UI/UX with styling frameworks like Material-UI or Tailwind CSS.
   - Implement search or filtering functionality to allow users to find specific reports.
   - Add error handling and validation to provide a robust user experience.

The approach taken to design a web app using React.js and TypeScript with a backend on Firebase that allows users to get a report for each uploaded code file involved several key steps. Firstly, the Firebase project was set up, enabling Firebase Authentication and Firestore. Next, the React.js project was created, and the necessary dependencies were installed. Firebase was then configured in the React app, with Firebase Authentication handling user registration and login, and Firestore used to store user data and reports. User authentication, file upload functionality, report generation, and report display were implemented, leveraging Firebase services such as Cloud Storage and Firestore. Additional features and enhancements, including user access control, UI/UX improvements, search/filtering functionality, and error handling, were also considered.

Challenges

---

1. Integration complexity: Integrating React.js, TypeScript, and Firebase can sometimes be complex due to the learning curve associated with each technology. Understanding the intricacies of Firebase services and how to connect them to React components can pose a challenge.
2. File analysis and report generation: Implementing code analysis and generating reports for uploaded code files can be challenging. It requires selecting appropriate libraries or tools, configuring them correctly, and extracting relevant information from the code files effectively.
3. Security and access control: Ensuring that users can only access their own reports and protecting sensitive user data requires careful implementation of security rules and access control mechanisms in Firebase.

Suggestions for improvement:

1. Clear documentation and examples: Providing detailed documentation and examples specifically tailored to integrating React.js, TypeScript, and Firebase would be helpful. This would assist developers in understanding the integration process, handling Firebase services, and implementing features efficiently.

2. Code analysis libraries: Researching and recommending specific code analysis libraries or tools that can be easily integrated with Firebase would simplify the process of generating reports. Providing clear examples or guides for integrating these tools into the app would also be beneficial.
3. Step-by-step tutorials: Creating step-by-step tutorials or guides that walk developers through the process of setting up the app, implementing features, and handling common challenges would be valuable. These tutorials could cover authentication, file upload, report generation, and display functionalities.
4. Error handling and feedback: Improving error handling and providing meaningful feedback to users when errors occur during file upload, report generation, or other processes would enhance the user experience. Clear error messages and prompts for corrective actions can help users understand and resolve issues more effectively.
5. Performance optimization: Offering performance optimization techniques, such as code splitting, lazy loading, and data caching, would help improve the speed and responsiveness of the web app. This could be particularly useful when dealing with large code files or generating complex reports.