My Friday - Engagement Management System

Table of Contents

Overview

My Friday is a comprehensive engagement management system designed for audit firms to manage client engagements, RFIs (Requests for Information), templates, and team collaboration. The application supports both auditor and client user types with role-based access control.

Application Purpose

The system enables audit firms to:

- **Manage Engagements**: Track audit engagements from creation through completion
- **Handle RFIs**: Manage requests for information with client responses and attachments
- **Collaborate**: Real-time chat and team collaboration features
- **Automate Workflows**: Automated email notifications, engagement recreation, and daily syncs
- **Integrate**: Seamless integration with My Review app for checklist management
- **Support Clients**: Client portal for viewing engagements and responding to RFIs

Key Capabilities

- **Engagement Management**: Create, edit, and manage audit engagements with 6-stage workflow
- **Client Management**: Manage client entities, entity types, and engagement types
- **RFI Management**: Handle requests for information with responses, attachments, and tracking
- **Template System**: Reusable RFI templates for engagements
- **Team Collaboration**: Team-based access control, real-time chat, and user management
- **Email Notifications**: Automated email notifications for various engagement events
- **Integration**: Seamless integration with My Review app for checklist management
- **File Management**: Upload, store, and manage files with Firebase Storage
- **Activity Logging**: Comprehensive audit trail for all actions
- **PWA Support**: Progressive Web App with offline capabilities

---

Technology Stack

Frontend

- **React** 18.2.0 - UI framework
- **React Router** 6.21.1 - Routing
- **Material-UI (MUI)** 5.15.1 - Component library
- **Firebase** 10.7.1 - Authentication and Firestore
- **Axios** 1.6.2 - HTTP client
- **React Firebase Hooks** 5.1.1 - Firebase integration hooks
- **Styled Components** 5.3.10 - Styling
- **Date-fns** 2.30.0 - Date manipulation
- **React PDF** 7.3.3 - PDF rendering
- **Workbox** 6.6.0 - Service worker and PWA support
- **Moment.js** 2.30.1 - Date manipulation and timezone handling

Backend

- **Firebase Cloud Functions** 4.3.1 - Serverless functions

- **Express** 4.18.2 - HTTP server framework
- **Firebase Admin SDK** 11.11.1 - Server-side Firebase operations
- **Node.js** 20 - Runtime environment
- **Nodemailer** 6.9.7 - Email sending
- **Google APIs** 131.0.0 - Google Drive and Google Docs integration
- **Pub/Sub** 4.5.0 - Message queuing for scheduled tasks
- **Moment.js** 2.30.1 - Date manipulation and timezone handling
- **Moment Timezone** 0.5.45 - Timezone support

Project Structure

```
friday/
├── build/                    # Production build output
├── functions/                # Firebase Cloud Functions
│   ├── index.js              # Main functions file
│   ├── adminUtils.js         # Admin utility functions
│   ├── userUtils.js          # User management utilities
│   ├── emailUtils.js         # Email sending utilities
│   ├── clientWeeklyEmails.js # Weekly email processing
│   ├── engagementRecreation.js # Engagement recreation logic
│   ├── pubSubUtils.js        # Pub/Sub utilities
│   └── serviceAccountKey*.json # Service account keys
├── public/                   # Static assets
├── src/
│   ├── components/           # Reusable React components
│   ├── globalFunctions/      # Shared utility functions
│   ├── hooks/                # Custom React hooks
│   ├── styles/               # Styled components
│   ├── sidebar/              # Sidebar component
│   ├── App.js                # Main app component
│   ├── firebase.js           # Firebase configuration
│   ├── AuthContext.js        # Authentication context
│   ├── Login.js              # Login page
│   ├── EngagementsHome.js    # Engagements routing
│   ├── ClientsHome.js        # Clients routing
│   ├── SettingsHome.js       # Settings routing
│   └── [Other feature files]
├── firebase.json             # Firebase configuration
├── package.json              # Frontend dependencies
└── README.md                 # This file
```

Authentication System

Authentication Methods

The application supports two authentication methods:

1. **Google Sign-In** (`signInWithGoogle`)
    - Uses Firebase Google Auth Provider
    - Validates user exists in Firestore before allowing access
    - Updates user UID if not previously set
2. **Email/Password** (`logInWithEmailAndPassword`)
    - Standard Firebase email/password authentication
    - Validates user exists in Firestore
    - Supports password reset functionality

Authentication Flow

1. User attempts to sign in (Google or Email/Password)
2. Firebase Authentication validates credentials
3. System checks if user exists in `friday/users/list` collection
4. Validates user status (must be "active")
5. If user is inactive or not found, logout is triggered
6. For auditors, automatically attempts to authenticate with My Review app
7. User data is loaded into AuthContext for app-wide access

User Types

The system distinguishes between two main user types:

Auditor Users

- **Type**: `type: "auditor"` in Firestore
- **Authentication**: Google Sign-In or Email/Password
- **Roles**: Partner, Manager, or Clerk (stored in `role` field)
- **Access**: Full access to assigned engagements, can create/manage engagements
- **Features**:
    - View all engagements (filtered by team assignment)
    - Create and edit engagements

    - Manage clients (Partners/Managers only)
    - Access Settings (Partners/Managers only)
    - Manage RFI templates
    - View all RFI items
    - Push checklists to My Review app
- **Data Access**: Can see all engagements they're assigned to, regardless of client

Client Users

- **Type**: `type: "client"` in Firestore
- **Authentication**: Email/Password only (created by auditors)
- **Roles**: `admin` or `user` (stored in `clients` array per client)
- **Access**: Limited to their assigned client's engagements
- **Features**:
    - View only engagements for their assigned client(s)
    - Respond to RFI items assigned to them
    - Upload attachments to RFI responses
    - View engagement details and files
    - Cannot create or edit engagements
    - Cannot access Settings, Clients, or Templates
- **Data Access**:
    - Filtered by `client_access` custom claims
    - Can only see RFI items where `users` array includes their `cloud_id`
    - Admin users see all RFIs for their client; regular users see only assigned RFIs

User Type Differentiation:

```
// Auditor User Document
{
  type: "auditor",
  role: "KBCSEHD91PdSh4StXOZe", // Partner role ID
  role_name: "Partner",
  teams: ["team-id-1"],
  status: "active"
}

// Client User Document
{
  type: "client",
  clients: [
    {
      cloud_id: "client-id-1",
```

```
        name: "Client Name",
        role: "admin", // or "user"
        status: "active"
      }
    ],
    status: "active"
  }
```

Access Control Flow:

```
User Login
    ↓
Check User Type (auditor/client)
    ↓
┌─────────────────┬─────────────────┐
│   Auditor       │   Client        │
│   - Load all    │   - Load only   │
│     assigned    │     client's    │
│     engagements │     engagements │
│   - Full access │   - Filtered    │
│     to features │     RFI access  │
└─────────────────┴─────────────────┘
```

Authentication Context

The `AuthContext` provides app-wide access to:

- `currentUser` : Currently authenticated user data
- `allUsers` : List of all auditor users
- `allUsersMap` : Map of users by email for quick lookup
- `userGroups` : User permission groups/roles
- `teams` : Team information
- `templates` : RFI templates
- `entityTypes` : Client entity types
- `engagementTypes` : Engagement type definitions

Custom Claims for Client Users

Client users have custom claims set on their Firebase Auth user to control access:

Custom Claims Structure:

```
  {
    type: "client",
    client_access: ["client-id-1", "client-id-2", ...]
  }
```

How Custom Claims Work:

1. When a client user is created via `/add-client-user`, custom claims are set
2. The `client_access` array contains IDs of all clients the user can access
3. When a user is added to additional clients, the array is updated
4. When a user is removed from a client, the client ID is removed from the array
5. Frontend can access claims via `user.getIdTokenResult().claims`
6. Backend verifies claims via `admin.auth().verifyIdToken()` to validate access

Custom Claims Flow:

```
Client User Created/Updated
    ↓
Backend Function (addClientUser/updateClientUser)
    ↓
Get/Set Custom Claims
    ├─ type: "client"
    └─ client_access: [clientIds]
    ↓
Firebase Auth User Updated
    ↓
User Token Contains Claims
    ↓
Frontend/Backend Validates Access
```

Firebase & Firestore Configuration

Firebase Projects

The application uses **two Firebase projects**:

1. **Friday Project** (Primary)
    - Project ID: `friday-a372b`
    - Storage Bucket: `friday-a372b.appspot.com`
    - Used for: Main application data, authentication, storage

2. **My Review Project** (Secondary)
    - Project ID: `audit-7ec47` (referenced in code)
    - Used for: Integration with My Review app

Firestore Structure

The Firestore database is organized under the `friday` collection:

```
friday/
├── settings/                    # App settings and configuration
├── users/
│   ├── list/                   # User documents (auditors and clients)
│   └── permissions/             # User role/permission definitions
├── teams/
│   └── list/                   # Team definitions
├── clients/
│   ├── list/                   # Client entity documents
│   ├── entity_types/           # Entity type definitions
│   └── engagement_types/        # Engagement type definitions
├── engagements/
│   └── list/                   # Engagement documents
│       └── [engagementId]/
│           ├── sections/        # Engagement sections
│           ├── rfis/           # RFI items for this engagement
│           └── last_read_messages/ # Last read tracking
├── templates/
│   └── list/                   # RFI template documents
└── logs/
    └── list/                   # System logs
```

Firestore Features

- **Persistent Local Cache**: Enabled with unlimited cache size
- **Multi-tab Manager**: Synchronizes cache across browser tabs
- **Real-time Listeners**: Uses `onSnapshot` for real-time data updates
- **Offline Support**: Works offline with cached data

Firebase Storage

Firebase Storage is used for all file uploads in the application. Files are organized in a hierarchical structure for easy access and management.

Storage Bucket

- **Bucket Name**: `friday-a372b.appspot.com`
- **Access**: Files are accessed via download URLs (signed URLs for backend, public URLs for frontend)

Storage Path Structure

Files are organized by purpose and context:

1. RFI Response Attachments

```
Path:
{clientId}/{engagementId}/{rfiId}/{questionId}/{fileName}_{timestamp}
Example: client-123/eng-456/rfi-789/question-
101/document.pdf_010124_143022
```

- **Purpose**: Attachments uploaded to RFI responses
- **Upload Function**: `uploadFilesToFirebase()` in `src/globalFunctions/uploadFilesToApi.js`
- **Access**: Both auditors and client users can upload/view

2. Engagement Master Files

```
Path: LockhatInc/{clientId}/{engagementId}/{fileName}_{timestamp}
Example: LockhatInc/client-123/eng-456/engagement-letter.pdf_010124_143022
```

- **Purpose**: Engagement-level files (engagement letters, post-RFI documents, etc.)
- **Upload Function**: `uploadMasterFilesToFirebase()` in `src/globalFunctions/uploadFilesToApi.js`
- **Access**: Primarily auditors, but client users can view

3. User CVs/Resumes

```
Path: Users/{userId}/{fileName}
Example: Users/user-789/john-doe-cv.pdf
```

- **Purpose**: Auditor user CVs/resumes
- **Upload Endpoint**: `POST /upload-user-cv` (backend)
- **Access**: Only auditors can upload; visible in user profiles

4. Email Attachments (Temporary)

```
Path: temp_email_attachments/{fileName}
Example: temp_email_attachments/attachment-123.pdf
```

- **Purpose**: Temporary storage for email attachments before allocation
- **Upload Endpoint**: `POST /mail-receive` (backend)
- **Access**: Admin users can allocate to RFIs

5. Engagement Details Files

```
Path: EngagementDetails/{engagementId}/{fileName}
Example: EngagementDetails/eng-456/details.pdf
```

- **Purpose**: Engagement-specific detail files
- **Upload Endpoint**: `POST /upload-engagement-details-file` (backend)
- **Access**: Auditors and client users can view

File Upload Process

Frontend Upload (Client-Side):

```
// 1. Create storage reference
const storageRef = ref(storage, `${clientId}/${engagementId}/${rfiId}/${c

// 2. Upload with progress tracking
const uploadTask = uploadBytesResumable(storageRef, file);

// 3. Get download URL after upload
const url = await getDownloadURL(uploadTask.snapshot.ref);

// 4. Save URL to Firestore RFI document
```

Backend Upload (Server-Side):

```
// 1. Create file reference in storage bucket
const blob = storage_bucket.file(`${path}/${fileName}`);

// 2. Create write stream
const blobStream = blob.createWriteStream({ metadata: { contentType } });

// 3. Generate signed URL (expires in future)
const url = await blob.getSignedUrl({ action: 'read', expires: '03-19-212
```

File Retrieval

Frontend Retrieval:

- Files are stored with download URLs in Firestore documents
- URLs are retrieved directly from Firestore (no additional API call needed)
- URLs are public or signed (depending on upload method)

Backend Retrieval:

- Uses Firebase Admin SDK to generate signed URLs
- Signed URLs expire after specified date (typically far future)
- Used for email attachments and file downloads

Storage Permissions

Firebase Storage Security Rules (should be configured):

- **RFI Attachments**: Users can read/write files in their client/engagement paths
- **Engagement Files**: Auditors can write, both can read
- **User CVs**: Only the user or admins can access
- **Email Attachments**: Only admins can access

Access Control Logic:

1. **Client Users**: Can only access files for their assigned clients
2. **Auditors**: Can access files for all engagements they're assigned to
3. **File Path Validation**: Backend validates client/engagement access before generating URLs

Storage Best Practices

1. **File Naming**: Files are renamed with timestamp to prevent conflicts
    - Format: `{originalName}_{DDMMYY_HHmmss}`
2. **File Size Limits**: Consider implementing size limits (typically 10-50MB per file)
3. **File Type Validation**: Validate file types before upload
4. **Cleanup**: Consider implementing cleanup for orphaned files
5. **Signed URLs**: Use signed URLs for sensitive files with expiration dates

---

My Review App Integration

Overview

My Friday integrates with **My Review** (My Work Review) app to enable auditors to push engagement checklists and retrieve reviews. The integration uses cross-project authentication.

Integration Endpoints

My Review API Endpoint:

`https://us-central1-audit-7ec47.cloudfunctions.net/api`

Authentication Flow

1. When an auditor logs into Friday, the app automatically attempts to authenticate with My Review
2. Friday sends the user's ID token to My Review's custom token generation endpoint:

   ```
   POST https://us-central1-audit-7ec47.cloudfunctions.net/api/generate-custom-token
   Body: { idToken: <friday_user_id_token> }
   ```

3. My Review validates the token and returns a custom token
4. Friday uses `signInWithCustomToken` to authenticate with My Review's Firebase project
5. This enables seamless access to My Review features from within Friday

Integration Features

1. **Get My Review Checklists** (`/get-mwr-checklists`)
   - Retrieves available checklists from My Review
   - Requires MWR key from settings
   - **Endpoint**: `https://us-central1-audit-7ec47.cloudfunctions.net/api/api/get-checklists`
   - **Authentication**: Bearer token (`MWR_SERVER_KEY`) + MWR key from settings
   - **Headers**: `X-Server-Server: true` (indicates server-to-server communication)

2. **Get My Review Reviews** (`/get-mwr-reviews`)
   - Retrieves reviews from My Review
   - Server-to-server communication with API key
   - **Endpoint**: `https://us-central1-audit-7ec47.cloudfunctions.net/api/api/friday-get-reviews`
   - **Authentication**: Bearer token (`MWR_SERVER_KEY`) + MWR key from settings
   - **Headers**: `X-Server-Server: true`

3. **Push Checklist to My Review** (`/push-mwr-checklist`)
   - Pushes engagement checklist to My Review
   - Enables checklist management in My Review app

- **Endpoint**: `https://us-central1-audit-7ec47.cloudfunctions.net/api/api/push-mwr-checklist`
- **Authentication**: Bearer token (`MWR_SERVER_KEY`) + MWR key from settings
- **Headers**: `X-Server-Server: true`
- **Error Responses**:
  - `401` : Unauthorized (invalid MWR key or server key)
  - `501` : Checklist already exists in review
  - `500` : Internal server error

Chat Synchronization

When a `review_link` is added to an engagement, the chat system automatically synchronizes messages from both Friday and My Review:

Chat Sync Process:

1. **Friday Chat**: Subscribes to `friday/engagements/list/{engagementId}/chat`
2. **My Review Chat**: If `review_link` exists, subscribes to `myworkreview/reviews/list/{review_link}/chat`
3. **Message Merging**: Both chat streams are merged and sorted by `datetime`
4. **Real-time Updates**: Uses Firestore `onSnapshot` listeners for real-time synchronization
5. **Message Identification**: Messages are tagged with `isFridayMessage: true` or `isReviewMessage: true`

Chat Sync Flow:

```
Engagement Has review_link
    ↓
Open Engagement Chat
    ↓
Subscribe to Friday Chat
    ├─ friday/engagements/list/{engagementId}/chat
    └─ Real-time listener (onSnapshot)
    ↓
Subscribe to My Review Chat (if review_link exists)
    ├─ myworkreview/reviews/list/{review_link}/chat
    └─ Real-time listener (onSnapshot)
    ↓
Merge Messages
    ├─ Combine both chat arrays
    ├─ Sort by datetime
```

```
    └─ Tag with source (Friday/Review)
    ↓
Display Unified Chat
```

Chat Sync Details:

- **Location**: `src/components/EngagementChat.js`
- **Active Tab**: Only syncs My Review chat in main chat tab (`activeTab === 0`), not in team chat tab
- **Message Format**: Both chats use the same message structure:
  - `user_email` : User email
  - `datetime` : Message timestamp
  - `message` : Message content
  - `photo` : User photo (from Friday's user map)
  - `isFridayMessage` or `isReviewMessage` : Source identifier
- **Caching**: Chat messages are cached in `EngagementsView.js` for quick loading
  - Caches last 10 messages from each chat
  - Merges Friday and My Review chats when `review_link` exists

Chat Sync Requirements:

- Engagement must have `review_link` field set (My Review review ID)
- User must be authenticated with both Friday and My Review Firebase projects
- My Review Firebase project must be accessible

Engagement Review Link

Review Link Field:

- **Field Name**: `review_link`
- **Type**: String (My Review review ID)
- **Location**: `friday/engagements/list/{engagementId}/review_link`
- **Purpose**: Links engagement to a My Review review for checklist and chat synchronization

Previous Year Review Link:

- **Field Name**: `previous_year_review_link`
- **Type**: String (My Review review ID)
- **Location**: `friday/engagements/list/{engagementId}/previous_year_review_link`
- **Purpose**: Stores the review link from the previous year's engagement
- **Usage**: Used during engagement recreation to preserve review link history

- **Recreation Behavior**: When engagement is recreated, `review_link` is set to `null` and `previous_year_review_link` is set to the old `review_link`

Review Link in Recreation:

- When engagement is recreated:
  - `review_link` : Set to `null` (new engagement starts without review link)
  - `previous_year_review_link` : Set to old engagement's `review_link`
  - This preserves the link to the previous year's review

Dependencies Between Friday and My Review

Authentication Dependencies:

1. **Custom Token Generation**:
   - Friday sends user's ID token to My Review
   - My Review validates and returns custom token
   - Friday uses custom token to authenticate with My Review Firebase
   - **Dependency**: My Review must have `/generate-custom-token` endpoint accessible
2. **Service Account Keys**:
   - Friday requires `serviceAccountKeyMyReview.json` to access My Review Firebase
   - Used for server-side operations (checklists, reviews)
   - **Dependency**: Service account must have access to My Review Firebase project

API Dependencies:

1. **MWR API Key**:
   - Stored in `friday/settings` document under `keys.mwr`
   - Required for all My Review API calls
   - **Dependency**: Must be set in settings for integration to work
2. **MWR Server Key**:
   - Stored in environment variable `MWR_SERVER_KEY`
   - Used as Bearer token for server-to-server communication
   - **Dependency**: Must be set in backend environment variables
3. **API Endpoints**:
   - My Review must have these endpoints accessible:
     - `/api/generate-custom-token`

        - `/api/api/get-checklists`
        - `/api/api/friday-get-reviews`
        - `/api/api/push-mwr-checklist`
    - **Dependency**: All endpoints must be deployed and accessible

Data Dependencies:

1. **Review Link**:
    - Engagement `review_link` field must match a valid My Review review ID
    - **Dependency**: Review must exist in My Review Firebase project
2. **Chat Synchronization**:
    - Friday chat: `friday/engagements/list/{engagementId}/chat`
    - My Review chat: `myworkreview/reviews/list/{review_link}/chat`
    - **Dependency**: Both Firestore collections must be accessible
    - **Dependency**: User must be authenticated with both Firebase projects
3. **User Data**:
    - Friday uses its own user map for chat message display
    - My Review chat messages use Friday's user map for photos/names
    - **Dependency**: User emails must match between Friday and My Review

Firebase Project Dependencies:

1. **My Review Firebase Project**:
    - Project ID: `audit-7ec47`
    - Must be accessible from Friday's backend
    - **Dependency**: Service account must have access
2. **Firestore Collections**:
    - Friday must be able to read/write to My Review Firestore:
        - `myworkreview/reviews/list/{reviewId}/chat`
        - `myworkreview/reviews/list/{reviewId}/checklists`
    - **Dependency**: Firestore security rules must allow access

Configuration Dependencies:

1. **Frontend Environment Variables**:
    - `REACT_APP_MY_REVIEW_FIREBASE_*` (all My Review Firebase config values)
    - **Dependency**: Must be set for frontend to connect to My Review

2. **Backend Environment Variables**:
    - `MWR_SERVER_KEY` : Bearer token for API calls
    - **Dependency**: Must be set for backend API calls to work
3. **Service Account Keys**:
    - `functions/serviceAccountKeyMyReview.json` : My Review Firebase admin access
    - **Dependency**: Must be present and valid

Error Handling:

- If My Review authentication fails: User can still use Friday, but My Review features are unavailable
- If API calls fail: Error messages shown to user, operations fail gracefully
- If chat sync fails: Only Friday chat is shown, My Review chat is skipped

Configuration

- My Review Firebase config is stored in environment variables (see Environment Variables section)
- Service account key: `functions/serviceAccountKeyMyReview.json`
- MWR API key stored in `friday/settings` document under `keys.mwr`
- MWR Server key stored in backend environment variable `MWR_SERVER_KEY`

Google Drive Integration

Overview

The application integrates with **Google Drive** for engagement letter generation. This allows dynamic generation of PDF engagement letters from Word document templates.

Google Drive Setup

Service Account: Uses `serviceAccountCloudKey.json` for authentication
Scopes Required:

- `https://www.googleapis.com/auth/drive` - Drive file access
- `https://www.googleapis.com/auth/documents` - Google Docs API access
- `https://www.googleapis.com/auth/drive.file` - File creation access

Drive Folder: Engagement letters are temporarily stored in Google Drive folder ID: `1DcOh90HMoy4Ntm6K7jl0EW-A8S-uIBwG`

Engagement Letter Generation Flow

Endpoint: `POST /generate-engagement-letter`

Process Flow:

```
1. Download .docx Template from Firebase Storage
    ↓
2. Upload to Google Drive (convert to Google Docs)
    ↓
3. Find and Replace Template Variables:
   - {client} → Client name
   - {year} → Engagement year
   - {address} → Client address
   - {date} → Commencement date
   - {email} → Client email(s)
   - {engagement} → Engagement type
    ↓
4. Export Google Docs as PDF
    ↓
5. Stream PDF back to client
    ↓
6. Cleanup: Delete temporary Google Drive file
```

Template Variables:

- `{client}` - Client name
- `{year}` - Engagement year
- `{address}` - Client address
- `{date}` - Engagement commencement date
- `{email}` - Client master email(s)(formatted as `[email1, email2]`)
- `{engagement}` - Engagement type name

Error Handling:

- Retries on rate limit errors (up to 3 attempts with exponential backoff)
- Logs errors and sends developer notifications
- Cleans up Google Drive files even on error

Usage in Email Generation:

- Engagement letters are generated as PDF attachments for email
- Used in engagement info gathering and initial engagement emails
- Function: `createAndDownloadEngagementLetter()` in `functions/index.js`

Progressive Web App (PWA) & Service Worker

Overview

The application is configured as a **Progressive Web App (PWA)** with service worker support for offline capabilities and automatic updates.

Service Worker Configuration

Location: `src/service-worker.js`, `src/serviceWorkerRegistration.ts`

Features:

- **Precaching**: Static assets are precached for offline access
- **Cache Strategies**:
  - **Cache First**: For static assets (images, fonts, CSS, JS)
  - **Network First**: For API calls
  - **Stale While Revalidate**: For dynamic content
- **Offline Support**: App works offline with cached data
- **Update Notifications**: Notifies users when new version is available

Service Worker Update Flow

Update Detection:

1. Service worker checks for updates on app load
2. New service worker is installed in background
3. User is notified when update is ready
4. User can refresh to activate new version

Implementation (`ServiceWorkerUpdateListener.tsx`):

- Listens for service worker update events
- Events: `onupdateinstalling`, `onupdatewaiting`, `onupdateready`
- Provides `skipWaiting()` method to force update activation

Update Notification (`App.js`):

- Shows snackbar when new version is available
- Provides "Refresh" button to reload app
- Checks version from `friday/settings` document

Version Management:

- App version stored in `package.json` (currently 1.1.863)
- Version checked against `friday/settings.version` in Firestore
- Mismatch triggers update notification

Offline Functionality

Firestore Offline Support:

- Persistent local cache enabled (unlimited size)
- Multi-tab synchronization
- Works offline with cached data
- Syncs when connection is restored

Online Status Detection:

- Hook: `useOnlineStatus()` in `src/hooks/useOnlineStatus.tsx`
- Shows notification when offline
- Displays: "You are connected with limited functionality"

PWA Features

- **Installable**: Can be installed as a web app
- **Offline Access**: Works with cached data when offline
- **Fast Loading**: Assets cached for quick subsequent loads
- **Update Notifications**: Automatic update detection and notification

---

Backend Firebase Functions

Architecture Overview

The backend uses Firebase Cloud Functions with Express.js for HTTP endpoints and Pub/Sub for scheduled tasks. Functions are organized into:

1. **HTTP Endpoints** - Express routes for direct API calls
2. **Pub/Sub Functions** - Scheduled tasks triggered by Cloud Scheduler

3. **Email Processing Pipeline** - Asynchronous email generation and sending
4. **Engagement Recreation System** - Automated engagement creation

Function Endpoints

All backend functions are exposed through Express routes in `functions/index.js`:

Engagement Management

- `POST /recreate-engagement` - Recreate engagement with updated structure
- `POST /upload-engagement-template` - Upload engagement letter template
- `POST /upload-engagement-details-file` - Upload files to engagement
- `POST /generate-engagement-letter` - Generate engagement letter from template
- `POST /upload-engagement-letter` - Upload signed engagement letter
- `POST /upload-post-rfi` - Upload post-RFI documents
- `POST /create-engagement-files-zip` - Create ZIP archive of engagement files
- `POST /delete-engagement` - Delete engagement

Client & User Management

- `POST /add-client-user` - Add new client user
- `POST /update-client-user` - Update client user details
- `POST /delete-client-user` - Delete client user
- `POST /replace-client-user-engagements` - Replace user across engagements

RFI Management

- `POST /send-rfi-reminder` - Send reminder email for RFI items
- `POST /upload-response-attachment` - Upload attachment to RFI response

My Review Integration

- `POST /get-mwr-checklists` - Get checklists from My Review
- `POST /get-mwr-reviews` - Get reviews from My Review
- `POST /push-mwr-checklist` - Push checklist to My Review

Authentication & Utilities

- `GET /hello` - Health check/test endpoint (returns "Hello From Friday")
- `POST /generate-custom-token` - Generate custom token for My Review

- `POST /upload-user-cv` - Upload user CV/resume
- `POST /reset-password-email` - Send password reset email
- `POST /test-client-emails` - Test email functionality

Email Processing

- `POST /mail-receive` - Receive and process incoming emails

Pub/Sub Scheduled Functions

The backend uses Google Cloud Pub/Sub for scheduled tasks. These functions are triggered by Cloud Scheduler and handle automated processes:

1. Daily Backup (`pubSub_dailyBackup`)

- **Topic**: `daily-backup`
- **Schedule**: Daily (configured in Cloud Scheduler)
- **Functionality**:
  - Exports entire Firestore database to Google Cloud Storage
  - Backup bucket: `gs://fridaydailybackups`
  - Uses Firestore Admin API for export
  - Logs failures to `friday/logs/list` collection
  - Sends developer email notifications on failure
- **Configuration**: Standard memory (512MB)

2. Daily User Sync (`pubSub_dailyUserSync`)

- **Topic**: `daily-user-sync`
- **Schedule**: Daily (configured in Cloud Scheduler)
- **Functionality**:
  - Syncs internal auditor users from external Google Apps Script
  - **External API**: `https://script.google.com/macros/s/AKfycbyqQp6ZLZn_vSNi4dxZxWHa1yEoFEcKMoWunP3Mytx2sXL9N0g/exec`
  - Requires `USER_EMAIL_SYNC_KEY` environment variable
  - **Process Flow**:
    1. Fetches user list from Google Apps Script (name, email, photo)
    2. Compares with existing Firestore users
    3. Adds new users (defaults to "Clerk" role, "active" status)
    4. Removes users not in external list (marks as "deleted" status)
    5. Updates photo URLs and names for existing users

6. Deletes Firebase Auth users for removed users

- **Concurrency**: Uses PQueue with concurrency: 10
- **Error Handling**: Sends developer email on failure

User Sync Flow Diagram:

```
External Google Apps Script
    ↓ (Daily Trigger)
Pub/Sub: daily-user-sync
    ↓
Fetch User List (name, email, photo)
    ↓
Compare with Firestore Users
    ↓
┌──────────────────┬──────────────────┐
│  New Users       │  Removed Users   │
│  - Add to DB     │  - Mark deleted  │
│  - Set role      │  - Delete Auth   │
└──────────────────┴──────────────────┘
    ↓
Update Photos & Names
    ↓
Complete
```

3. Daily Engagement Check (`pubSub_dailyEngagementsCheck`)

- **Topic**: `daily-engagement-check`
- **Schedule**: Daily
- **Memory**: 1GB, Timeout: 540 seconds
- **Functionality**:
  - **Stage Transitions**: Moves engagements from "Info Gathering" to "Commencement" when commencement date is reached
  - **Custom Notifications**: Processes engagement-specific notification triggers
  - Checks all active engagements (excluding "Close Out" stage and 100% complete)
  - Sends custom engagement notifications based on configured triggers
- **Concurrency**: PQueue with concurrency: 5

4. Daily RFI Notifications (`pubSub_dailyRfiNotifications`)

- **Topic**: `daily-rfi-notifications`
- **Schedule**: Daily
- **Memory**: 1GB, Timeout: 540 seconds

- **Functionality**:
  - Processes RFI deadline notifications
  - Sends reminder emails for overdue RFI items
  - Cleans up processed notifications
- **Concurrency**: PQueue with concurrency: 2

5. Daily Manager/Clerk Notifications (`pubSub_dailyManagerClerkNotifications`)

- **Topic**: `daily-manager-clerk-notifications`
- **Schedule**: Daily
- **Functionality**:
  - Consolidates daily RFI notifications for managers and clerks
  - Reads from `friday/engagements/client_sent_notifications` collection
  - Groups notifications by engagement
  - Sends consolidated email to engagement team members
  - Deletes processed notifications after sending
- **Concurrency**: PQueue with concurrency: 4

6. Weekly Client Engagement Emails (`pubSub_weeklyClientEngagementEmails`)

- **Topic**: `weekly-client-engagement-emails`
- **Schedule**: Weekly (configured in Cloud Scheduler)
- **Memory**: 1GB, Timeout: 540 seconds
- **Functionality**:
  - Processes all active engagements in stages: Commencement, Team Execution, Partner Review, Finalization
  - Generates weekly summary emails for client users
  - **Email Types**:
    - Individual user emails (for non-admin users with assigned RFIs)
    - Admin user email (first admin gets all RFIs)
    - Master summary email (sent to master email with section summaries)
  - Calculates working days outstanding for each RFI
  - Groups RFIs by section and user assignment
- **Concurrency**: PQueue with concurrency: 30, Batch size: 10

Weekly Email Flow:

```
Pub/Sub: weekly-client-engagement-emails
    ↓
Get Active Engagements (Commencement → Finalization)
    ↓
For Each Engagement:
```

```
    ├─ Get Client Users
    ├─ Get RFIs (client_status != "Sent")
    ├─ Get Sections
    └─ Process RFIs by User Assignment
        ↓
    ┌─────────────────────────┬─────────────────────────┐
    │  Admin Users            │  Regular Users          │
    │  - First admin gets     │  - Individual emails    │
    │    all RFIs             │    with assigned RFIs   │
    └─────────────────────────┴─────────────────────────┘
    ↓
Generate Master Summary Email
    ↓
Send All Emails
```

7. Yearly Engagement Creation (`pubSub_yearlyEngagementCreation`)

- **Topic**: `yearly-engagement-creation`
- **Schedule**: Yearly (configured in Cloud Scheduler)
- **Memory**: 2GB, Timeout: 540 seconds
- **Functionality**:
  - Finds all engagements with `repeat_interval: "yearly"`
  - Queues engagement recreation in batches
  - Publishes to `process-engagement-batch` topic for processing
- **Batch Size**: 20 engagements per batch

8. Monthly Engagement Creation (`pubSub_monthlyEngagementCreation`)

- **Topic**: `monthly-engagement-creation`
- **Schedule**: Monthly (configured in Cloud Scheduler)
- **Memory**: 2GB, Timeout: 540 seconds
- **Functionality**:
  - Finds all engagements with `repeat_interval: "monthly"`
  - Queues engagement recreation in batches
  - Publishes to `process-engagement-batch` topic for processing
- **Batch Size**: 20 engagements per batch

9. Process Engagement Recreation Batch (`pubSub_processEngagementRecreationBatch`)

- **Topic**: `process-engagement-batch`
- **Triggered By**: Yearly/Monthly engagement creation functions

- **Memory**: 2GB, Timeout: 540 seconds
- **Functionality**:
  - Processes batches of engagements for recreation
  - Creates new engagement for next period (year/month)
  - Copies engagement structure, RFIs, sections
  - Updates dates (commencement, completion)
  - Resets engagement letter and post-RFI statuses
  - **Repeat Interval Management**:
    - **New Engagement**: Inherits the same `repeat_interval` (yearly/monthly) from the original engagement
    - **Original Engagement**: Updated to `repeat_interval: "once"` after successful recreation
    - This ensures the new engagement will be recreated again next period, while the old one won't be recreated
  - Logs recreation status to `friday/recreationLogs/list`
  - Handles errors and logs failed engagements
  - **Error Handling**: If recreation fails, deletes the new engagement and reverts the original engagement's `repeat_interval` back to its original value
- **Concurrency**: PQueue with concurrency: 5

Engagement Recreation Flow:

```
Yearly/Monthly Pub/Sub Trigger
    ↓
Get Engagements (repeat_interval: "yearly"/"monthly")
    ↓
Queue in Batches (20 per batch)
    ↓
Pub/Sub: process-engagement-batch
    ↓
For Each Engagement:
    ├─ Calculate New Dates (year/month)
    ├─ Check if Engagement Already Exists
    │   └─ If exists: Update original to "once" and skip
    ├─ Create New Engagement Document
    │   ├─ Copy all fields from original
    │   ├─ Set new year/month
    │   ├─ Set new commencement date
    │   ├─ Reset stage to "Info Gathering"
    │   ├─ Reset progress to 0
    │   ├─ Reset engagement letter status
    │   ├─ Reset post-RFI status
```

```
    │   └─ **Keep same repeat_interval** (yearly/monthly)
    ├─ Copy Sections (processSection)
    │   ├─ Create new section in new engagement
    │   ├─ Get all RFIs for section from old engagement
    │   └─ For each RFI:
    │       ├─ Check recreate_with_attachments flag
    │       ├─ If true: Copy files & responses
    │       ├─ Create RFI with reset statuses
    │       └─ Update section_cloud_id to new section
    ├─ Copy Checklists
    ├─ **Update Original Engagement**
    │   └─ Set repeat_interval: "once"
    └─ Log Recreation Status
```

Repeat Interval Update Process:

```
Original Engagement (repeat_interval: "yearly"/"monthly")
    ↓
Recreation Process Starts
    ↓
Create New Engagement
    ├─ repeat_interval: "yearly"/"monthly" (inherited)
    ├─ year: next year
    ├─ month: next month (if monthly)
    └─ All other fields copied
    ↓
Copy Sections & RFIs
    ↓
Update Original Engagement
    └─ repeat_interval: "once" ✅
    ↓
Result:
    ├─ Original: repeat_interval = "once" (won't be recreated)
    └─ New: repeat_interval = "yearly"/"monthly" (will be recreated next
period)
```

Section and RFI Recreation Process:

The recreation process copies sections and RFIs in a structured manner:

Section Recreation:

1. **Process Sections in Batches**: Sections are processed in batches of 100
2. **Create New Section**: Each section is copied to the new engagement with a new `cloud_id`

3. **Link RFIs to New Section**: RFIs are queried by `section_cloud_id` and linked to the new section

RFI Recreation:

1. **Query RFIs by Section**: RFIs are fetched for each section using `section_cloud_id`
2. **Check `recreate_with_attachments` Flag**:
   - If `recreate_with_attachments === true` : RFI files and responses are copied
   - If `recreate_with_attachments === false` or undefined: Only RFI structure is copied (no attachments)
3. **Copy Files (if flagged)**:
   - Queries `friday/engagements/list/{oldEngagementId}/files` where `question_id == rfiId`
   - Copies all files to `friday/engagements/list/{newEngagementId}/files`
   - Preserves original file document IDs
   - Updates `filesCount` in the new RFI
4. **Copy Responses (if flagged)**:
   - Queries `friday/engagements/list/{oldEngagementId}/responses` where `question_id == rfiId`
   - Copies all responses to `friday/engagements/list/{newEngagementId}/responses`
   - Preserves original response document IDs
   - Updates `responseCount` in the new RFI
5. **Create New RFI**:
   - Copies all RFI fields from original
   - **Resets Status Fields**:
     - `status: 0` (waiting)
     - `auditor_status: "Requested"`
     - `client_status: "Pending"`
     - `date_requested: newCommencementDate`
     - `deadline_date: ""`
     - `flag: false`
     - `ml_query: false`
     - `date_resolved: ""`
     - `lastMessageBy: ""`
     - `teamChatCount: 0`
   - **Updates Section Reference**:
     - `section_cloud_id: newSectionCloudId` (links to new section)
     - `engagement_rfi_cloud_id: originalRfiId` (preserves link to original RFI)

- **Preserves Original RFI ID**: New RFI uses the same document ID as the original
- **Removes Timestamps**: Deletes `lastUpdatedTimeStamp` and `lastTeamChatTimestamp`

RFI Recreation with Attachments Flow:

```
For Each Section:
    ↓
Create New Section (new cloud_id)
    ↓
Get RFIs for Section (section_cloud_id)
    ↓
For Each RFI:
    ├─ Check recreate_with_attachments flag
    │   ↓
    │   ┌──────────────────┬──────────────────┐
    │   │   true           │   false/undefined│
    │   │   - Copy files   │   - Skip files   │
    │   │   - Copy         │   - Skip         │
    │   │     responses    │     responses    │
    │   │   - Update       │   - Set counts   │
    │   │     counts       │     to 0         │
    │   └──────────────────┴──────────────────┘
    ├─ Create RFI Document
    │   ├─ Copy all fields
    │   ├─ Reset statuses
    │   ├─ Update section_cloud_id
    │   └─ Preserve original RFI ID
    └─ Link to New Section
```

File and Response Copying:

- **Files**: Copied from `friday/engagements/list/{oldEngagementId}/files` to `friday/engagements/list/{newEngagementId}/files`
- **Responses**: Copied from `friday/engagements/list/{oldEngagementId}/responses` to `friday/engagements/list/{newEngagementId}/responses`
- **Preservation**: Original document IDs are preserved for both files and responses
- **Error Handling**: If file/response copy fails, logs error but continues processing

Checklist Recreation:

- Checklists are copied separately from sections/RFIs
- Processed in batches of 500

- **Resets Checklist Fields**:
  - `datetime: newCommencementDate`
  - `friday_pushed: { status: false, pushedBy: "", pushedDate: "" }`
  - `resolved: false`
  - `resolvedBy: ""`
  - `resolvedDate: ""`
  - `responses: []`

Error Handling:

- If recreation fails at any point:
  1. Delete the newly created engagement
  2. Revert original engagement's `repeat_interval` back to its original value (yearly/monthly)
  3. Log failure status with error details
  4. This ensures the original engagement can be retried in the next scheduled run
- **File/Response Copy Errors**: If individual file or response copy fails, logs error but continues processing other files/responses

10. Email Content Generation (`pubSub_generateEmailContent`)

- **Topic**: `email-content-generation`
- **Triggered By**: Various functions that need to send emails
- **Memory**: 1GB, Timeout: 300 seconds
- **Functionality**:
  - Generates email content based on type
  - **Supported Email Types**:
    - `clientSignup` - Client user signup confirmation
    - `passwordReset` - Password reset link
    - `clientReminder` - Client reminder notifications
    - `clientEngagement` - Engagement notifications
    - `engagementInfoGathering` - Info gathering stage emails
    - `engagementNotification` - Custom engagement notifications
    - `clerkRfiNotification` - RFI notifications for clerks/managers
    - `rfiDeadlineNotification` - RFI deadline reminders
    - `clientWeeklyEngagement` - Weekly engagement summaries
    - `clientMasterWeeklyEngagement` - Master weekly summaries
    - `engagementFinalization` - Finalization notices
    - `engagementFiles` - Engagement file notifications
    - `bugReportDeveloper` - Developer bug reports

    - Publishes generated email to `email-sending` topic

11. Process Emails (`pubSub_processEmails`)

- **Topic**: `email-sending` (subscription name from env)
- **Triggered By**: Email content generation function
- **Memory**: 512MB, Timeout: 300 seconds
- **Functionality**:
    - Processes email queue with rate limiting
    - **Rate Limiting**: Max 500 emails per day (configurable)
    - Handles email attachments (CVs, engagement letters, files)
    - Downloads and attaches files from Firebase Storage
    - Sends emails via Nodemailer SMTP
    - Tracks daily email count and resets at midnight
- **Email Queue**: In-memory queue processed sequentially

Email Processing Pipeline:

```
Function Needs to Send Email
    ↓
Publish to: email-content-generation
    ↓
Generate Email Content (HTML, subject, etc.)
    ↓
Publish to: email-sending
    ↓
Check Daily Email Limit (< 500)
    ↓
Process Attachments (if any)
    ↓
Send via Nodemailer SMTP
    ↓
Complete
```

Firestore Trigger Functions

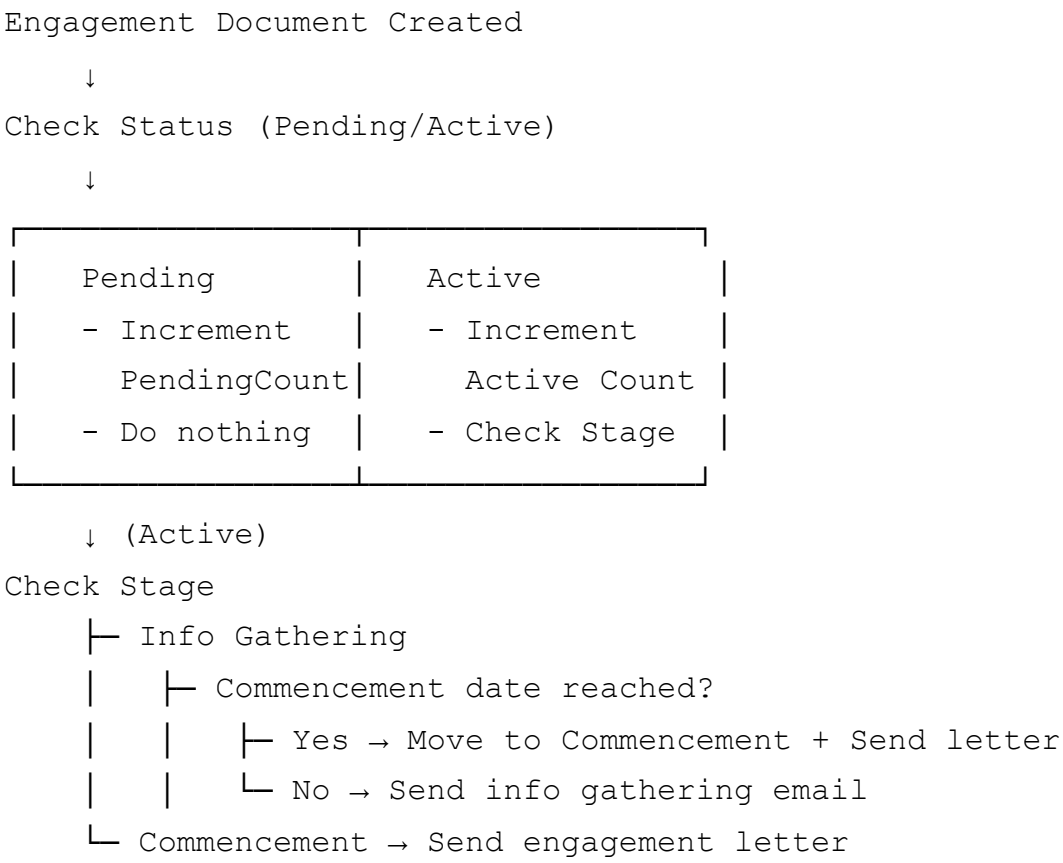
The backend uses Firestore triggers to automatically respond to document changes. These functions are triggered when documents are created, updated, or deleted:

1. Engagement Create (`engagementCreate`)

- **Trigger**: `onCreate` on `friday/engagements/list/{engagementId}`

- **Memory**: 2GB, Timeout: 540 seconds
- **Functionality**:
  - **Pending Engagements**: Increments `PendingCounts` for the team
  - **Active Engagements**:
    - Increments active counts and fees for the client
    - Increments stage count for the team
    - **Info Gathering Stage**:
      - If commencement date is today or past: Moves to "Commencement" stage and sends engagement letter
      - If commencement date is future: Sends info gathering email
    - **Commencement Stage**: Sends engagement letter to client
- **Concurrency**: PQueue with concurrency: 10
- **Error Handling**: Sends developer email on failure

Engagement Create Flow:



2. Engagement Update (`engagementUpdate`)

- **Trigger**: `onUpdate` on `friday/engagements/list/{engagementId}`
- **Memory**: 1GB, Timeout: 540 seconds
- **Functionality**:
  - **Stage Changes**: Updates counts and fees when stage changes
  - **Status Changes**:
    - Pending → Active: Decrements pending count, increments active count

        - If in "Info Gathering": Sends info gathering email
    - **Commencement Date Changes**: Updates all RFI `date_requested` fields
    - **Stage to Commencement**: Sends engagement letter if active
    - **Stage to Finalization**: Sends finalization notice to client
    - **Post-RFI Status Changes**:
        - AFS "Sent": Generates and sends draft AFS email
        - Journals "Sent": Generates and sends draft journals email
    - **Date Adjustments**: If engagement becomes active with past commencement date, updates to current date
- **Concurrency**: PQueue with concurrency: 5
- **Error Handling**: Sends developer email on failure

Engagement Update Flow:

```
Engagement Document Updated
    ↓
Check What Changed
    ├─ Stage Changed
    │   ├─ Update counts/fees
    │   ├─ Finalization → Send finalization notice
    │   └─ Commencement (Active) → Send engagement letter
    ├─ Status Changed (Pending → Active)
    │   ├─ Update counts
    │   └─ Info Gathering → Send info gathering email
    ├─ Commencement Date Changed
    │   └─ Update all RFI dates
    ├─ Post-RFI AFS "Sent"
    │   └─ Generate and send draft AFS email
    └─ Post-RFI Journals "Sent"
        └─ Generate and send draft journals email
```

3. Engagement Delete (`engagementDelete`)

- **Trigger**: `onDelete` on `friday/engagements/list/{engagementId}`
- **Functionality**:
    - **Pending Engagements**: Decrements `PendingCounts` for the team
    - **Active Engagements**: Decrements stage count for the team
    - **Info Gathering Stage**: Decrements `InfoGathering.{team}` count
    - **Other Stages**: Decrements `{stage}.{team}` count
- **Concurrency**: PQueue with concurrency: 3
- **Error Handling**: Sends developer email on failure

4. Client Inactive Update (`clientInactiveUpdate`)

- **Trigger**: `onUpdate` on `friday/clients/list/{clientId}`
- **Functionality**:
  - **Status Change to Inactive**:
    - Updates all engagements with `repeat_interval != "once"` to `repeat_interval: "once"`
    - Deletes engagements in "Info Gathering" stage with 0% progress
  - **Purpose**: Prevents future engagement recreation for inactive clients
- **Concurrency**: PQueue with concurrency: 3
- **Error Handling**: Sends developer email on failure

Client Inactive Flow:

```
Client Status Changed to Inactive
    ↓
Find All Engagements for Client
    ↓
┌─────────────────────┬─────────────────────┐
│   Repeat != Once    │   Info Gathering    │
│   - Update to       │   + 0% Progress     │
│     "once"          │   - Delete          │
└─────────────────────┴─────────────────────┘
```

5. RFI Update Monitor (`rfiUpdateMonitor`)

- **Trigger**: `onUpdate` on `friday/engagements/list/{engagementId}/rfis/{rfiId}`
- **Memory**: 2GB, Timeout: 540 seconds
- **Functionality**:
  - **Client Status Changed to "Sent"**:
    - Creates notification in `friday/engagements/client_sent_notifications`
    - Includes engagement team members (excluding Partners)
    - Used by daily manager/clerk notifications
  - **Deadline Date Changed**:
    - If deadline set: Creates/updates RFI notification for assigned client users
    - If deadline removed: Removes RFI notification
    - Notifies client admins and assigned users
- **Concurrency**: PQueue with concurrency: 3
- **Error Handling**: Logs errors (no email sent to avoid spam)

RFI Update Flow:

```
RFI Document Updated
    ↓
Check What Changed
    ├─ Client Status → "Sent"
    │    └─ Create notification for team members
    └─ Deadline Date Changed
        ├─ Deadline Set → Create/update notification
        └─ Deadline Removed → Remove notification
```

6. Team Users Update Monitor (`teamUsersUpdateMonitor`)

- **Trigger**: `onUpdate` on `friday/teams/list/{teamId}`
- **Functionality**:
  - **Users Removed from Team**:
    - Finds all engagements for the team containing removed users
    - Removes users from engagement `user` arrays
    - Processes in batches of 200 engagements
  - **Purpose**: Maintains data consistency when team membership changes
- **Batch Size**: 200 engagements per batch
- **Error Handling**: Sends developer email on failure

Team Users Update Flow:

```
Team Document Updated
    ↓
Check if Users Removed
    ↓
Find Engagements with Removed Users
    ↓
Remove Users from Engagement Arrays
    ↓ (Batch of 200)
Update Firestore
```

Custom Claims on Authentication

The system uses Firebase Custom Claims to manage client user access control:

Client User Custom Claims

When a client user is created or updated, custom claims are set on their Firebase Auth user:

```
{
  type: "client",
  client_access: [clientId1, clientId2, ...]  // Array of client IDs user
}
```

Custom Claims Flow

1. **User Creation** (`addClientUser`):
    - Creates Firebase Auth user with custom claims
    - Sets `type: "client"`
    - Sets `client_access: [clientId]` (array of accessible client IDs)
2. **User Update** (`updateClientUser`):
    - Retrieves existing custom claims
    - Updates `client_access` array when user is added to new clients
    - Preserves existing client access
3. **User Deletion** (`deleteClientUser`):
    - Removes client ID from `client_access` array
    - If no clients remain, user may be deactivated

Custom Claims Usage

- **Frontend**: Can be accessed via `user.getIdTokenResult()` to check `client_access`
- **Backend**: Verified via `admin.auth().verifyIdToken()` to validate client access
- **Security**: Prevents client users from accessing other clients' data

Custom Claims Structure:

```
Firebase Auth User
├─ uid: "user-uid"
├─ email: "user@example.com"
└─ customClaims:
    ├─ type: "client"
    └─ client_access: ["client-id-1", "client-id-2"]
```

Email Receiving System

The system can receive and process incoming emails:

Email Receive Endpoint (`/mail-receive`)

- **Method**: POST
- **Authentication**: Bearer token (`EXPECTED_BEARER_TOKEN`)
- **Functionality**:
  - Receives email data from external email service
  - Parses email content (subject, body, from, to, cc, bcc, date)
  - Processes attachments (base64 encoded)
  - Stores attachments in Firebase Storage (`temp_email_attachments/`)
  - Saves email to Firestore `emails` collection
  - Marks email as `allocated: false` (pending allocation)

Email Allocation (`EmailReceiveDialog` component)

- **Location**: `src/components/EmailReceiveDialog.js`
- **Functionality**:
  - Displays unallocated emails from `emails` collection
  - Allows allocation to:
    - Client
    - Engagement
    - RFI Question
  - **Allocation Types**:
    - `clientResponse` - Client response to RFI
    - `manualResponse` - Manual response entry
  - Uploads attachments to engagement/RFI
  - Marks email as `allocated: true`

Email Receiving Flow:

```
External Email Service
    ↓ (POST /mail-receive)
Parse Email Data
    ├─ Subject, Body, From, To
    ├─ Attachments (base64)
    └─ Date
    ↓
Store Attachments in Firebase Storage
    ↓
Save to Firestore (emails collection)
    ↓
Mark as allocated: false
```

```
    ↓
Admin Allocates Email
    ├─ Select Client
    ├─ Select Engagement
    └─ Select RFI Question
    ↓
Upload Attachments to RFI
    ↓
Mark as allocated: true
```

Function Configuration

- **Standard API**: 512MB memory, 9-minute timeout
- **High Memory API**: 4GB memory, 9-minute timeout (for large file operations)
- **CORS**: Configured for specific origins (localhost, Netlify deployments)
- **PQueue**: Used for concurrent processing with configurable concurrency limits

---

Application Overview

User Interface Structure

The application follows a standard web application structure with:

Main Navigation:

- **NavBar** (`src/components/NavBar.js`): Top navigation bar component providing main app navigation. Includes:
  - Logo and app name
  - Logout button
  - Clients link (for auditors)
  - Settings link (for Partners)
  - My Review link (for auditors)
  - Mobile menu for small screens

Layout Components:

- **ProtectedLayout** (`src/components/ProtectedLayout.js`): Main layout wrapper for authenticated routes. Provides consistent layout structure (header, sidebar, content area) for all protected pages. Ensures users are authenticated before accessing content.
- **ManagerPartnerLayout** (`src/components/ManagerPartnerLayout.js`): Layout wrapper that restricts access to Manager and Partner roles only. Used for settings and administrative

pages that require elevated permissions.

- **AuditorLayout** (`src/components/AuditorLayout.js`): Layout wrapper for auditor-only routes. Ensures only auditor users (not clients) can access certain pages.
- **PartnerLayout** (`src/components/PartnerLayout.js`): Layout wrapper for Partner-only routes. Restricts access to highest privilege level for sensitive operations.

Main Pages:

- **Home** (`src/Home.js`): Dashboard/home page
- **Login** (`src/Login.js`): Authentication page
- **ForgotPassword** (`src/ForgotPassword.js`): Password reset page
- **Splash** (`src/Splash.js`): Loading/splash screen

Frontend Routing System

Routing Library:

- **React Router DOM** v6.21.1 - Declarative routing for React applications
- Uses `BrowserRouter` for client-side routing
- Implements nested routes with `Routes` and `Route` components
- Supports lazy loading for code splitting

Routing Setup:

1. **Root Router** (`src/index.js`):
    - Wraps entire app with `<BrowserRouter>`
    - Handles top-level routes: `/` and `/app/*`
    - Both routes render the `<App />` component
2. **App Router** (`src/App.js`):
    - Main application routing logic
    - Implements route protection and nested routing
    - Uses lazy loading for major components (ClientsHome, SettingsHome, EngagementsHome, Login, ForgotPassword)
    - Wraps routes in `<Suspense>` for loading states

Routing Structure:

```
BrowserRouter (index.js)
    ↓
App Component (App.js)
```

```
    ↓
┌──────────────────┬──────────────────┐
│  Public Routes   │ Protected Routes │
│  - /login        │  - RequireAuth   │
│  - /password-    │    - Protected   │
│    reset         │      Layout      │
│  - /unauthorized │      - Routes    │
└──────────────────┴──────────────────┘
```

Nested Routing Pattern:

- Each major section has its own routing component:
  - **EngagementsHome** (`src/EngagementsHome.js`): Handles all engagement routes
  - **ClientsHome** (`src/ClientsHome.js`): Handles all client routes
  - **SettingsHome** (`src/SettingsHome.js`): Handles all settings routes
  - **RFIHome** (`src/RFIHome.js`): Handles all RFI routes

Example Nested Route Structure:

```
// ClientsHome.js
<Routes>
  <Route path="/" element={<ClientsView />} />
  <Route path="/new" element={<ClientsDetails />} />
  <Route path="/:id" element={<ClientsDetails />} />
  <Route path="/entity-types/*" element={<EntityTypesHome />} />
  <Route path="/engagement-types/*" element={<EngagementTypesHome />} />
</Routes>
```

Lazy Loading:

- Major components are lazy-loaded for performance:

```
const LazyClientsHome = lazy(() => import('./ClientsHome'));
const LazySettingsHome = lazy(() => import('./SettingsHome'));
const LazyEngagementsHome = lazy(() => import('./EngagementsHome'));
```

- Wrapped in `<Suspense>` with `<Loader>` fallback
- Reduces initial bundle size and improves load time

Route Protection Hierarchy:

```
Public Routes (no protection)
    ↓
```

```
RequireAuth (authentication check)
    ↓
ProtectedLayout (main layout wrapper)
    ↓
NonClerkRoute (blocks Clerk role)
    ↓
ManagerPartnerLayout (restricts to Manager/Partner)
    ↓
Actual Route Component
```

Application Routes

Public Routes:

- `/` - Home/Landing page
- `/login` - Login page
- `/password-reset` - Password reset page
- `/unauthorized` - Unauthorized access page

Protected Routes (Auditors):

- `/app` - Main dashboard (defaults to engagements)
- `/app/engagements` - Engagements list view
- `/app/engagements/:team/:id` - Edit engagement
- `/app/engagements/:team/:id/:engagementId` - View engagement details
- `/app/engagements/new` - Create new engagement
- `/app/clients` - Clients list view
- `/app/clients/:id` - Client details/edit
- `/app/clients/new` - Create new client
- `/app/clients/entity-types/*` - Entity types management
- `/app/clients/engagement-types/*` - Engagement types management
- `/app/rfi` - RFI management
- `/app/rfi/:id` - RFI details
- `/app/rfi/new` - Create new RFI
- `/app/templates` - Templates management
- `/app/settings/*` - Settings (Partners/Managers only)

Protected Routes (Clients):

- `/app/engagements` - Client engagements view (filtered)
- `/app/engagements/client/:id/:engagementId` - Client engagement details

Route Parameters:

- `:team` - Team ID for engagement routes
- `:id` - Client ID or engagement ID
- `:engagementId` - Specific engagement ID
- `*` - Wildcard for nested routes

Route Protection:

- **RequireAuth** (`src/components/RequireAuth.js`): Base authentication check - redirects to `/login` if not authenticated
- **NonClerkRoute** (`src/NonClerkRoute.js`): Blocks Clerk access - redirects to `/unauthorized` if user is Clerk
- **TeamAuthRoute** (`src/TeamAuthRoute.js`): Team-based access control - ensures user has access to specific team's engagements
- **ManagerPartnerLayout**: Restricts to Manager/Partner roles - used for settings and administrative pages

---

Core Features

1. Engagements Management

Location: `src/EngagementsHome.js` , `src/EngagementsView.js` , `src/EngagementDetailView.js` , `src/EngagementsEdit.js`

Functionality:

- View all engagements with filtering and grouping
- Create new engagements
- Edit engagement details (dates, client, team, etc.)
- View engagement details with sections and RFIs
- Manage engagement files and documents
- Generate engagement letters
- Track engagement status and milestones
- Real-time chat for engagements
- Activity logging for all changes

Key Components:

- **EngagementsHome** (`src/EngagementsHome.js`): Routing component for engagements
- **EngagementsView** (`src/EngagementsView.js`): Main list view for auditors with DataGrid

- **EngagementsClientView** (`src/EngagementsClientView.js`): Client-facing engagements view
- **EngagementDetailView** (`src/EngagementDetailView.js`): Detailed engagement view with sections and RFIs
- **EngagementsEdit** (`src/EngagementsEdit.js`): Create/edit engagement form
- **EngagementItem** (`src/components/EngagementItem.js`): Individual engagement item component

Engagement Stages and Workflow

Engagements progress through **6 distinct stages**:

1. Info Gathering (`InfoGathering`)

- **Description**: Initial stage before engagement begins
- **Auto-Transition**: Automatically moves to "Commencement" when commencement date is reached (via daily Pub/Sub)
- **User Transition**: Cannot manually set to this stage if commencement date has passed
- **Features**:
  - Can set engagement details
  - RFIs are not yet sent to client
  - Days outstanding calculation returns 0

2. Commencement (`Commencement`)

- **Description**: Engagement has started, RFIs can be sent to client
- **Auto-Transition**: From "Info Gathering" when commencement date is reached
- **User Transition**: Can move to "Team Execution"
- **Features**:
  - Engagement letter can be sent
  - RFIs become active
  - Days outstanding calculation begins

3. Team Execution (`TeamExecution`)

- **Description**: Team is actively working on the engagement
- **User Transition**: Can move to "Partner Review" or back to "Commencement"
- **Features**:
  - Full RFI management
  - Team collaboration features
  - Progress tracking

4. Partner Review (`PartnerReview`)

- **Description**: Engagement is under partner review
- **User Transition**: Can move to "Finalization" or back to "Team Execution"
- **Features**:
  - Partner review workflows
  - Final checks before completion

5. Finalization (`Finalization`)

- **Description**: Engagement is being finalized
- **User Transition**: Can move to "Close Out" (Partners only)
- **Features**:
  - Final document preparation
  - Client rating available (for client admins)
  - Completion workflows

6. Close Out (`CloseOut`)

- **Description**: Engagement is complete and ready for collection
- **User Transition**: Only Partners can set this stage
- **Requirements**:
  - Engagement letter must be marked as "Accepted" by auditor
  - Or engagement progress must be 0
- **Features**:
  - Engagement is considered closed
  - No further changes allowed
  - Ready for billing/collection

Stage Transition Rules:

Allowed Transitions:

```
Info Gathering → Commencement (auto or manual)
Commencement → Team Execution (manual)
Team Execution → Partner Review (manual)
Team Execution → Commencement (manual, back)
Partner Review → Finalization (manual)
Partner Review → Team Execution (manual, back)
Finalization → Close Out (Partners only)
```

Restrictions:

- Clerks cannot change engagement stage
- Cannot set to "Info Gathering" if commencement date has passed
- Cannot set to "Close Out" unless engagement letter is accepted or progress is 0
- Cannot change stage if engagement status is "Pending"

Implementation: `EngagementStageSelection` component (`src/components/EngagementStageSelection.js`)

Engagement Status Options

Engagement Status (`status` field):

- `Pending`: Engagement is pending approval
- `Active`: Engagement is active and in progress

Engagement Client Status (`client_status` field):

- `Pending`: Client has not responded
- `Partially Sent`: Client has partially responded
- `Sent`: Client has sent all responses

Engagement Auditor Status (`auditor_status` field):

- `Requested`: RFI has been requested from client
- `Reviewing`: Auditor is reviewing response
- `Queries`: Auditor has queries on response
- `Received`: Auditor has accepted the response

Engagement Letter Status:

- **Client Status**: `Pending`, `Sent`
- **Auditor Status**: `Pending`, `Queries`, `Accepted`

Post-RFI Status:

- **Client Status**: `Pending`, `Queries`, `Accepted`
- **Auditor Status**: `Pending`, `Sent`

RFI Status Workflow

RFI Status Values (`status` field - numeric):

- `0`: "waiting" - RFI is waiting for response
- `1`: "completed" - RFI has been completed/accepted

RFI Status Options (`rfi_status` field - string):

- `Pending`: RFI is pending
- `Active`: RFI is active

- `Inactive` : RFI is inactive

RFI Status Flow:

```
RFI Created (status: 0, "waiting")
    ↓
Client Responds
    ↓
Auditor Reviews
    ↓
Auditor Accepts (status: 1, "completed")
    OR
Auditor Requests Changes (status: 0, "waiting")
```

Status Change Rules:

- Only auditors (not clerks) can change RFI status
- Status can only be changed to "completed" if:
  - RFI has files uploaded OR
  - RFI has responses
- When status changes to "completed":
  - `date_resolved` is set to current timestamp
  - `auditor_status` is set to "Received"
  - `client_status` is set to "Pending" (for new cycle)
- When status changes back to "waiting":
  - `date_resolved` is cleared
  - `auditor_status` is updated based on current state

Implementation: `handleUpdateQuestionStatus()` in `EngagementItem` component

6. Real-Time Chat

Location: `src/components/EngagementChat.js`

Functionality:

- Real-time chat for engagements
- Team chat for RFIs
- File attachments in chat
- Message replies
- Message search
- Chat history
- Integration with My Review chat

Key Components:

- **EngagementChat** (`src/components/EngagementChat.js`): Main chat component for engagement team and client communication. Provides real-time chat with Firestore listeners, file upload support, message history, and synchronized chat from both Friday and My Review when review_link is set.
- **InputComponent**: Chat input component with file upload support. Handles message composition, file attachments, drag-and-drop, and message sending. Provides rich text input with attachment capabilities.
- **QuickViewAttachment** (`src/components/QuickViewAttachment.jsx`): Component for previewing file attachments. Displays file previews (images, PDFs, documents) in a quick-view modal without full download.

7. Email Management

Location: `src/components/EmailReceiveDialog.js`

Functionality:

- Receive emails from external email service
- Allocate emails to clients, engagements, and RFIs
- Upload email attachments to RFIs
- View email content and attachments
- Filter allocated/unallocated emails

Key Components:

- **EmailReceiveDialog** (`src/components/EmailReceiveDialog.js`): Dialog component for allocating received emails to engagements, RFIs, and questions. Allows admin users to process incoming emails, attach them to specific RFI questions, and manage email attachments. Handles email-to-RFI workflow.

8. PDF Editor

Location: `src/components/PDFEditorDialog.tsx` , `src/components/pdfEditor/`

Functionality:

- PDF viewing and annotation
- Area highlighting
- Text highlighting
- PDF editing capabilities
- Mobile and desktop support

Key Components:

- **PDFEditorDialog** (`src/components/PDFEditorDialog.tsx`): PDF editor dialog
- **PdfHighlighter** (`src/components/pdfEditor/PdfHighlighter.tsx`): PDF highlighter component

UI Components Reference

Common UI Components

Layout Components:

- **ProtectedLayout** (`src/components/ProtectedLayout.js`): Main layout wrapper with NavBar
- **ManagerPartnerLayout** (`src/components/ManagerPartnerLayout.js`): Layout for Manager/Partner routes
- **AuditorLayout** (`src/components/AuditorLayout.js`): Layout for auditor routes
- **PartnerLayout** (`src/components/PartnerLayout.js`): Layout for Partner routes
- **Header** (`src/components/Header.js`): Page header with back link
- **NavBar** (`src/components/NavBar.js`): Top navigation bar

Dialog Components:

- **ConfirmDialog** (`src/components/ConfirmDialog.js`): Confirmation dialog
- **EmailReceiveDialog** (`src/components/EmailReceiveDialog.js`): Email allocation dialog
- **EngagementChecklistDialog** (`src/components/EngagementChecklistDialog.js`): RFI checklist dialog
- **EngagementChat** (`src/components/EngagementChat.js`): Engagement chat dialog
- **EngagementActivityDialog** (`src/components/EngagementActivityDialog.js`): Activity log dialog
- **EngagementRatingDialog** (`src/components/EngagementRatingDialog.js`): Engagement rating dialog
- **EngagementSwitchStageDialog** (`src/components/EngagementSwitchStageDialog.js`): Stage change dialog
- **RFIEngagementItemDialog** (`src/RFIEngagementItemDialog.js`): RFI item dialog
- **RFIEngagementPostRFIDialog** (`src/RFIEngagementPostRFIDialog.js`): Post-RFI dialog
- **PDFEditorDialog** (`src/components/PDFEditorDialog.tsx`): PDF editor dialog

Form Components:

- **InputField** (`src/components/InputField.js`): Text input field
- **SelectField** (`src/components/SelectField.js`): Dropdown select field
- **SearchField** (`src/components/SearchField.js`): Search input field
- **DatePicker**: Date selection (MUI DatePicker)

Display Components:

- **Loader** (`src/components/Loader.js`): Loading spinner
- **LoaderStacked** (`src/components/LoaderStacked.js`): Stacked loading indicator
- **EmptyContainer** (`src/components/EmptyContainer.js`): Empty state display
- **StatusLabel** (`src/components/StatusLabel.js`): Status badge/label
- **EngagementProgressLabel** (`src/components/EngagementProgressLabel.js`): Progress indicator
- **CircularProgressLabel** (`src/components/CircularProgressLabel.js`): Circular progress
- **LinearProgressLabel** (`src/components/LinearProgressLabel.js`): Linear progress
- **UserAvatar** (`src/components/UserAvatar.js`): User avatar display

File Components:

- **ResponseAttachment** (`src/components/ResponseAttachment.js`): Response attachment display
- **ResponseAttachmentPreview** (`src/components/ResponseAttachmentPreview.js`): Attachment preview
- **RFIResponseAttachmentPreview** (`src/components/RFIResponseAttachmentPreview.js`): RFI attachment preview
- **QuickViewAttachment** (`src/components/QuickViewAttachment.jsx`): Quick view attachment
- **uploadPostRFIFiles** (`src/components/uploadPostRFIFiles.js`): Post-RFI file upload

RFI Components:

- **AttachRFIDialog** (`src/components/AttachRFIDialog.js`): Attach RFI dialog
- **AttachRFIQuestionDialog** (`src/components/AttachRFIQuestionDialog.js`): Attach RFI question dialog
- **EditRFIQuestionDialog** (`src/components/EditRFIQuestionDialog.js`): Edit RFI question dialog
- **AttachRFIClientUserDialog** (`src/components/AttachRFIClientUserDialog.js`): Assign client user dialog
- **AttachRFIQuestionClientUserDialog** (`src/components/AttachRFIQuestionClientUserDialog.js`): Assign client user to

question dialog

- **AttachBulkDeadlinesRFIDialog** (`src/components/AttachBulkDeadlinesRFIDialog.js`): Bulk deadline dialog
- **RFIItemReminderDialog** (`src/components/RFIItemReminderDialog.js`): RFI reminder dialog

Engagement Components:

- **EngagementItem** (`src/components/EngagementItem.js`): Individual engagement item
- **EngagementCardItem** (`src/components/Engagements/EngagementCardItem.js`): Engagement card
- **EngagementStageSelection** (`src/components/EngagementStageSelection.js`): Stage selection dropdown
- **EngagementTeamUsersDialog** (`src/components/EngagementTeamUsersDialog.js`): Team users dialog
- **EngagementSectionOrderDialog** (`src/components/EngagementSectionOrderDialog.js`): Section order dialog
- **EngagementDateChoiceDialog** (`src/components/EngagementDateChoiceDialog.js`): Date selection dialog
- **EngagementRatingComponent** (`src/components/EngagementRatingComponent.js`): Rating component

Client Components:

- **ClientManageUserDialog** (`src/components/ClientManageUserDialog.js`): Client user management dialog
- **ClientRiskAssessmentDialog** (`src/components/ClientRiskAssessmentDialog.js`): Risk assessment dialog
- **ClientTestEmailDialog** (`src/components/ClientTestEmailDialog.js`): Test email dialog

Template Components:

- **RFITemplateSelectField** (`src/components/RFITemplateSelectField.js`): Template selection field

Settings Components:

- **SettingsEngagementNotificationTriggerDialog** (`src/components/SettingsEngagementNotificationTriggerDialog.js`): Notification trigger dialog

Other Components:

- **ColorPickerDialog** (`src/components/ColorPickerDialog.js`): Color picker dialog
- **ChooseFilesFromOtherEngagementsDialog** (`src/components/ChooseFilesFromOtherEngagementsDialog.js`): File selection dialog
- **ImportReviewQueriesDialog** (`src/components/ImportReviewQueriesDialog.js`): Import queries dialog
- **PushMWRDialog** (`src/components/PushMWRDialog.js`): Push to My Review dialog
- **EngagementChecklistPushMyReviewDialog** (`src/components/EngagementChecklistPushMyReviewDialog.js`): Push checklist dialog
- **GetRFIReport** (`src/components/GetRFIReport.js`): RFI report generator
- **RFIReport** (`src/components/RFIReport.js`): RFI report display
- **AiTool** (`src/components/AiTool.js`): AI tool integration

Route Protection Components:

- **RequireAuth** (`src/components/RequireAuth.js`): Authentication check wrapper
- **UnAuthorizedRoute** (`src/components/UnAuthorizedRoute.js`): Unauthorized access page

PDF Editor Components (`src/components/pdfEditor/`):

- **PdfHighlighter** (`PdfHighlighter.tsx`): Main PDF highlighter
- **Highlight** (`Highlight.tsx`): Text highlight component
- **AreaHighlight** (`AreaHighlight.tsx`): Area highlight component
- **PdfLoader** (`PdfLoader.tsx`): PDF loader component
- **Tip** (`Tip.tsx`): Tooltip component
- **Popup** (`Popup.tsx`): Popup component

Common/Utility Components:

- **Header** (`src/components/Header.js`): Page header component providing consistent header layout across pages. Includes title, breadcrumbs, and action buttons.
- **Loader** (`src/components/Loader.js`): Loading spinner component. Displays loading state during data fetching or async operations.
- **LoaderStacked** (`src/components/LoaderStacked.js`): Stacked loading indicator for multiple async operations. Shows progress for multiple concurrent loading states.
- **EmptyContainer** (`src/components/EmptyContainer.js`): Component for displaying empty states. Shows message when no data is available (e.g., no engagements, no RFIs).
- **UserAvatar** (`src/components/UserAvatar.js`): Component for displaying user avatars/photos. Shows user profile pictures with fallback to initials or default avatar.

- **StatusLabel** (`src/components/StatusLabel.js`): Reusable component for displaying status badges/labels. Shows colored status indicators (active, pending, completed, etc.).
- **SearchField** (`src/components/SearchField.js`): Reusable search input component. Provides consistent search interface with debouncing and search icon.
- **SelectField** (`src/components/SelectField.js`): Reusable dropdown/select component. Provides consistent select interface with validation and error handling.
- **InputField** (`src/components/InputField.js`): Reusable text input component. Provides consistent input interface with validation, labels, and error messages.
- **ConfirmDialog** (`src/components/ConfirmDialog.js`): Reusable confirmation dialog component. Displays confirmation prompts for destructive actions (delete, remove, etc.).
- **SeparatorLine** (`src/components/SeparatorLine.js`): Visual separator component. Provides horizontal line dividers for UI layout.
- **ExpandMore** (`src/components/ExpandMore.js`): Expand/collapse icon component. Used for collapsible sections and accordions.
- **CircularProgressLabel** (`src/components/CircularProgressLabel.js`): Circular progress indicator with label. Shows percentage progress in circular format.
- **LinearProgressLabel** (`src/components/LinearProgressLabel.js`): Linear progress indicator with label. Shows percentage progress in linear format.
- **EngagementProgressLabel** (`src/components/EngagementProgressLabel.js`): Specialized progress indicator for engagements. Displays engagement completion percentage with custom styling.

File/Attachment Components:

- **ResponseAttachment** (`src/components/ResponseAttachment.js`): Component for displaying RFI response attachments. Shows attached files in RFI responses with download/preview options.
- **ResponseAttachmentLoad** (`src/components/ResponseAttachmentLoad.js`): Loading component for response attachments. Shows loading state while fetching attachment data.
- **ResponseAttachmentPreview** (`src/components/ResponseAttachmentPreview.js`): Preview component for response attachments. Provides preview functionality for response files.
- **RFIResponseAttachmentPreview** (`src/components/RFIResponseAttachmentPreview.js`): Specialized preview component for RFI response attachments. Handles RFI-specific attachment preview logic.
- **ResponseChoiceBox** (`src/components/ResponseChoiceBox.js`): Component for selecting response options. Provides choice/option selection interface for RFI responses.
- **uploadPostRFIFiles** (`src/components/uploadPostRFIFiles.js`): Component for uploading post-RFI files. Handles file upload for post-RFI document attachments.

PDF Editor Components (continued):

- **PDFEditorDialog** (`src/components/PDFEditorDialog.tsx`): Dialog wrapper for PDF editor. Provides modal interface for PDF viewing and annotation.
- **HighlightLayer** (`src/components/pdfEditor/HighlightLayer.tsx`): Layer component for managing multiple highlights. Handles rendering and interaction of multiple highlight overlays.
- **MouseSelection** (`src/components/pdfEditor/MouseSelection.tsx`): Component for mouse-based text selection in PDFs. Handles mouse drag selection for highlighting text.
- **MouseMonitor** (`src/components/pdfEditor/MouseMonitor.tsx`): Component for monitoring mouse events in PDF viewer. Tracks mouse position and interactions for PDF features.
- **MobileResizableRectangle** (`src/components/pdfEditor/MobileResizableRectangle.tsx`): Component for resizable rectangular selections on mobile. Enables area selection on touch devices.
- **TipContainer** (`src/components/pdfEditor/TipContainer.tsx`): Container component for managing tooltips in PDF editor. Handles tooltip positioning and display logic.
- **Spinner** (`src/components/pdfEditor/Spinner.tsx`): Loading spinner for PDF operations. Shows loading state during PDF loading or processing.
- **ConvertDataUrlFile** (`src/components/pdfEditor/ConvertDataUrlFile.tsx`): Utility component for converting data URLs to files. Handles file format conversions for PDF operations.

Activity/Engagement Components:

- **EngagementActivityDialog** (`src/components/EngagementActivityDialog.js`): Dialog for displaying engagement activity log. Shows chronological list of all activities/events for an engagement.
- **EngagementSwitchStageDialog** (`src/components/EngagementSwitchStageDialog.js`): Dialog for switching engagement stages. Provides confirmation and validation when changing engagement stage.

Component Patterns

Data Display:

- **MUI DataGrid**: Used for list views (engagements, clients, users). Provides sorting, filtering, pagination, and inline editing capabilities.
- **Custom Cards**: Used for engagement items and RFI items. Provides visual card-based layout with hover effects and quick actions.
- **Tables**: Used for RFI checklists and data display. Provides structured data presentation with sorting and filtering.

Form Patterns:

- **Multi-step Forms**: Used for engagement creation. Breaks complex forms into manageable steps with progress indicators.

- **Dialog Forms**: Used for editing RFIs, clients, users. Provides modal interface for form editing without navigation.
- **Inline Editing**: Used in DataGrid for quick edits. Allows direct editing of cell values without opening dialogs.

Real-time Updates:

- **Firestore Listeners**: Used for real-time data updates. Provides automatic UI updates when data changes in Firestore.
- **Chat Components**: Real-time chat with Firestore listeners. Enables instant message delivery and synchronization.
- **Activity Feeds**: Real-time activity logging. Shows live updates of user actions and system events.

File Handling:

- **Drag & Drop**: Used for file uploads. Provides intuitive file upload interface with drag-and-drop support.
- **File Preview**: Used for attachment previews. Enables quick viewing of files without full download.
- **PDF Viewer**: Used for PDF viewing and annotation. Provides PDF rendering with highlighting and annotation capabilities.

2. Clients Management

Location: `src/ClientsHome.js`, `src/ClientsView.js`, `src/ClientsDetails.js`, `src/EntityTypesHome.js`, `src/EngagementTypesHome.js`

Functionality:

- Manage client entities (create, edit, view)
- Define entity types (e.g., Corporation, Partnership)
- Define engagement types
- Assign client users to engagements
- View client engagement history
- Manage client users (add, remove, update roles)

Key Components:

- **ClientsHome** (`src/ClientsHome.js`): Routing component for clients
- **ClientsView** (`src/ClientsView.js`): List of all clients with DataGrid
- **ClientsDetails** (`src/ClientsDetails.js`): Client detail/edit form
- **EntityTypesHome** (`src/EntityTypesHome.js`): Entity type management
- **EngagementTypesHome** (`src/EngagementTypesHome.js`): Engagement type management

- **ClientManageUserDialog** (`src/components/ClientManageUserDialog.js`): Dialog for managing client users

3. RFI (Request for Information) Management

Location: `src/RFIHome.js` , `src/RFIView.js` , `src/components/EngagementChecklistDialog.js` , `src/RFIEngagementItemDialog.js`

Functionality:

- Create RFI items from templates or ad-hoc
- Assign RFI items to client users
- Track RFI responses and attachments
- Set deadlines and track outstanding days
- Send RFI reminders via email
- Filter and search RFI items
- Push checklists to My Review app
- Real-time team chat for RFIs
- Flag important RFIs
- Mark RFIs as ML queries
- Bulk deadline management

Key Components:

- **RFIHome** (`src/RFIHome.js`): Routing component for RFI management
- **RFIView** (`src/RFIView.js`): RFI list view
- **EngagementChecklistDialog** (`src/components/EngagementChecklistDialog.js`): Main RFI checklist dialog
- **RFIEngagementItemDialog** (`src/RFIEngagementItemDialog.js`): Individual RFI item dialog
- **RFIEngagementPostRFIDialog** (`src/RFIEngagementPostRFIDialog.js`): Post-RFI dialog
- **AttachRFIDialog** (`src/components/AttachRFIDialog.js`): Dialog for attaching RFIs
- **EditRFIQuestionDialog** (`src/components/EditRFIQuestionDialog.js`): Dialog for editing RFI questions

RFI Status and Workflow

RFI Status Values:

- **Numeric Status** (`status` field):
  - `0` : "waiting" - Waiting for client response

- `1` : "completed" - Auditor has accepted the response

RFI Status Options:

- **String Status** (`rfi_status` field):
  - `Pending` : RFI is pending
  - `Active` : RFI is active and awaiting response
  - `Inactive` : RFI is inactive

RFI Response Workflow:

```
RFI Created (status: 0)
    ↓
Assigned to Client Users
    ↓
Client Responds (uploads files/responses)
    ↓
Auditor Reviews Response
    ↓
┌─────────────────┬─────────────────┐
│   Accept        │   Request       │
│   (status: 1)   │   Changes       │
│   "completed"   │   (status: 0)   │
│                 │   "waiting"     │
└─────────────────┴─────────────────┘
```

RFI Status Change Rules:

- Only auditors (not clerks) can change RFI status
- Cannot change to "completed" without:
  - Files uploaded (`filesCount > 0`) OR
  - Responses provided (`responseCount > 0`)
- When marked as "completed":
  - `date_resolved` is set to current timestamp
  - `auditor_status` is set to "Received"
  - `client_status` is reset to "Pending"
- When marked back as "waiting":
  - `date_resolved` is cleared
  - `auditor_status` is updated based on current state

Days Outstanding Calculation:

- Calculated from `date_requested` to `date_resolved` (or current date)

- Uses working days (excludes weekends)
- Returns 0 if engagement stage is "Info Gathering"
- Formula: `calculateWorkingDays(date_requested, date_resolved || current_date)`

Implementation: `handleUpdateQuestionStatus()` in `EngagementItem` component

4. Templates Management

Location: `src/TemplatesHome.js`, `src/TemplateDetails.js`

- Create and manage RFI templates
- Define template sections and questions
- Reuse templates across engagements
- Edit template structure

5. Settings

Location: `src/SettingsHome.js`, `src/SettingsView.js`

User Management (`SettingsUsers.js`)

- View all auditor users
- Create/edit user accounts
- Assign roles and permissions
- Manage user status (active/inactive)
- Upload user CVs

Team Management (`SettingsTeamsView.js`)

- Create and manage teams
- Assign users to teams
- Team-based access control

RFI Templates (`RFIHome.js`)

- Manage RFI templates
- Create template sections and questions

Engagement Notifications (`SettingsEngagementNotifications.js`)

- Configure notification settings

- Email notification preferences

Integrations (`SettingsIntegrations.js` **)**

- Manage external integrations
- API key configuration

Recreation Logs (`SettingsRecreationLogs.js` **)**

- View engagement recreation history
- Track system changes

6. Email Notifications

The system sends automated emails for:

- Client engagement notifications
- RFI reminders
- RFI deadline notifications
- Engagement finalization
- Weekly engagement summaries
- Password reset
- Client signup confirmations

Email Configuration: Uses Nodemailer with SMTP configuration

7. Email Receiving & Allocation

Location: `src/components/EmailReceiveDialog.js` , `functions/index.js` (`/mail-receive`)

The system can receive and process incoming emails:

- **Email Receiving**:
    - Receives emails via POST endpoint `/mail-receive`
    - Authenticates with Bearer token
    - Parses email content (subject, body, from, to, cc, bcc, date)
    - Processes attachments (base64 encoded)
    - Stores attachments in Firebase Storage (`temp_email_attachments/`)
    - Saves email to Firestore `emails` collection
    - Marks email as `allocated: false` (pending allocation)

- **Email Allocation**:
  - Displays unallocated emails in `EmailReceiveDialog` component
  - Allows allocation to:
    - Client
    - Engagement
    - RFI Question
  - **Allocation Types**:
    - `clientResponse` - Client response to RFI (uploads attachments to RFI)
    - `manualResponse` - Manual response entry
  - Uploads attachments to engagement/RFI
  - Marks email as `allocated: true`

Use Case: Allows processing of client email responses and automatically attaching them to RFI items.

8. Activity Logging

Location: `src/hooks/useLogQuestionActivity.js`, `src/components/EngagementActivityDialog.js`

The system maintains comprehensive activity logs for engagements and RFIs:

RFI Activity Logging:

- **Location**: `friday/engagements/list/[engagementId]/rfis/[rfiId]/activity`
- **Logged Events**:
  - Status changes (waiting → completed, completed → waiting)
  - File uploads
  - Response additions
  - Flag status changes
  - ML Query status changes
  - Deadline date updates
  - Client status updates
  - Auditor status updates
- **Activity Document Structure**:

```
{
  message: "Team status changed to Accepted",
  user_email: "user@example.com",
  dateTime: Timestamp
}
```

Engagement Activity Logging:

- **Location**: `friday/engagements/list/[engagementId]/activity`
- **Logged Events**:
  - Engagement field changes (dates, client, team, etc.)
  - RFI additions/removals
  - Stage changes
  - All questions marked as accepted
  - Recreation events
- **Activity Document Structure**:

```
{
  message: "Changed commencement_date from 2024-01-01 to 2024-02-01",
  user_email: "user@example.com",
  dateTime: Timestamp
}
```

Usage:

- Hook: `useLogQuestionActivity()` provides `logActivity()` and `logEngagementActivity()`
- Component: `EngagementActivityDialog` displays activity history
- View: Users can view activity timeline for engagements and RFIs

Implementation: `useLogQuestionActivity` hook in `src/hooks/useLogQuestionActivity.js`

9. Engagement Recreation Intervals

Location: `src/EngagementsEdit.js`, `functions/engagementRecreation.js`

Engagements can be configured with **repeat intervals** to automatically recreate for future periods:

Repeat Interval Options:

- `yearly` - Engagement is recreated every year
- `monthly` - Engagement is recreated every month
- `once` - Engagement is a one-time engagement (no recreation)

Recreation Process:

1. **Yearly Recreation**:
   - Triggered by Pub/Sub: `yearly-engagement-creation`
   - Creates new engagement for next year

- Updates commencement date by 1 year
- Copies all sections, RFIs, and structure
- Resets engagement letter and post-RFI statuses

2. **Monthly Recreation**:
   - Triggered by Pub/Sub: `monthly-engagement-creation`
   - Creates new engagement for next month
   - Updates commencement date by 1 month
   - Handles year rollover (December → January)
   - Copies all sections, RFIs, and structure
   - Resets engagement letter and post-RFI statuses

3. **Recreation Rules**:
   - Checks if engagement already exists for new period
   - If exists, marks original as `once` (prevents duplicate creation)
   - Logs recreation status to `friday/recreationLogs/list`
   - Handles errors and logs failed recreations

Recreation Logging:

- **Location**: `friday/recreationLogs/list`
- **Log Structure**:

```
{
  engagementId: "original-engagement-id",
  clientId: "client-id",
  engagementTypeId: "engagement-type-id",
  status: "pending" | "completed" | "failed",
  details: {
    repeatInterval: "yearly" | "monthly",
    oldMonth: "January",
    newMonth: "February",
    oldYear: "2024",
    newYear: "2025"
  },
  timestamp: "2024-01-01 12:00:00"
}
```

Implementation: `processEngagement()` in `functions/engagementRecreation.js`

10. Business Rules and Validation

Location: `src/EngagementsEdit.js`, `src/components/EngagementStageSelection.js`

The system enforces several business rules and validation logic:

Engagement Creation Rules

Required Fields:

- `team` - Must be selected
- `year` - Must be 4 digits
- `engagement_type` - Must be selected
- `client` - Must be selected
- `repeat_interval` - Must be selected (yearly, monthly, once)
- `commencement_date` - Must be provided
- `fee` - Must be greater than 0
- `user` - At least one user must be assigned

Validation Rules:

- **Duplicate Check**: Cannot create engagement if one already exists for same client, engagement type, year, and month
- **Date Validation**: Commencement date cannot be before today (for new engagements)
- **Stage Validation**: Cannot set stage to "Info Gathering" if commencement date has passed
- **Close Out Validation**: Cannot set to "Close Out" unless:
  - Engagement letter is marked as "Accepted" by auditor, OR
  - Engagement progress is 0

Engagement Stage Rules

Stage Transition Restrictions:

- Clerks cannot change engagement stage
- Cannot change stage if engagement status is "Pending"
- Cannot set to "Info Gathering" if commencement date has passed
- Only Partners can set stage to "Close Out"
- Stage transitions must follow allowed progression path

Allowed Stage Progression:

```
Info Gathering → Commencement → Team Execution → Partner Review → Finalization → Close Out
```

Backward Transitions:

- Team Execution → Commencement (allowed)
- Partner Review → Team Execution (allowed)
- Other backward transitions are not allowed

RFI Status Rules

Status Change Validation:

- Cannot change to "completed" without files or responses
- Only auditors (not clerks) can change RFI status
- Cannot change status if engagement is closed
- Status changes are logged to activity

Engagement Recreation Rules

Recreation Validation:

- Checks if engagement already exists for new period
- Prevents duplicate creation
- Validates dates and month/year combinations
- Handles year rollover for monthly engagements

Implementation: `checkFields()`, `checkIfEngagementExists()`, `isStageDisabled()` functions

Database Structure

Main Collections

`friday/users/list`

User documents with fields:

- `email` : User email address
- `name` : User full name
- `type` : "auditor" or "client"
- `role` : Role ID (for auditors)
- `status` : "active" or "inactive"
- `uid` : Firebase Auth UID
- `authProvider` : "google" or "email"
- `clients` : Array of client associations (for client users)

- `cv_url` : URL to uploaded CV (for auditors)

`friday/users/permissions`

Role/permission definitions:

- `name` : Role name (e.g., "Partner", "Manager", "Clerk")
- `key` : Role key
- `roles` : Permission object

`friday/clients/list`

Client entity documents:

- `name` : Client name
- `entity_type` : Entity type ID
- `status` : "active" or "inactive"
- `users` : Array of client user IDs
- `engagements` : Array of engagement IDs

`friday/engagements/list`

Engagement documents:

- `client_id` : Client ID
- `engagement_type` : Engagement type ID
- `year` : Engagement year
- `month` : Engagement month (for monthly engagements)
- `commencement_date` : Start date
- `completion_date` : End date
- `team` : Team ID
- `status` : Engagement status ("Pending" or "Active")
- `stage` : Engagement stage (Info Gathering, Commencement, etc.)
- `users` : Array of assigned user IDs
- `client_users` : Array of assigned client user IDs
- `repeat_interval` : "yearly", "monthly", or "once"
- `progress` : Engagement progress percentage (0-100)
- `client_progress` : Client progress percentage (0-100)
- `fee` : Engagement fee
- `engagement_letter` : Engagement letter details (file, statuses)

- `post_rfi` : Post-RFI document details
- `post_rfi_journal` : Post-RFI journal details
- `feedback` : Client feedback/rating
- `clerksCanApprove` : Boolean flag for clerk approval permissions
- `rfis_linked` : Array of linked RFI template IDs

`friday/engagements/list/[engagementId]/rfis`

RFI item documents:

- `key` : RFI key/template reference
- `name` : RFI question name/text
- `question` : RFI question text
- `date_requested` : Request date (YYYY-MM-DD HH:mm format)
- `date_resolved` : Resolution date (set when status changes to 1)
- `deadline` : Response deadline
- `deadline_date` : Deadline date
- `status` : Numeric status (0 = "waiting", 1 = "completed")
- `users` : Array of assigned client user IDs
- `responses` : Array of response objects
- `files` : Array of file attachments
- `filesCount` : Number of files uploaded
- `responseCount` : Number of responses
- `days_outstanding` : Calculated outstanding days
- `auditor_status` : Auditor status (Requested, Reviewing, Queries, Received)
- `client_status` : Client status (Pending, Partially Sent, Sent)
- `flag` : Boolean flag for important RFIs
- `ml_query` : Boolean flag for ML queries
- `section_cloud_id` : Section ID this RFI belongs to
- `engagement_rfi_cloud_id` : RFI cloud ID
- `recreate_with_attachments` : Boolean flag for recreation

`friday/engagements/list/[engagementId]/rfis/[rfiId]/activity`

RFI activity log documents:

- `message` : Activity message
- `user_email` : User who performed the action
- `dateTime` : Timestamp of the activity

`friday/engagements/list/[engagementId]/activity`

Engagement activity log documents:

- `message` : Activity message
- `user_email` : User who performed the action
- `dateTime` : Timestamp of the activity

`friday/engagements/list/[engagementId]/sections`

Engagement section documents:

- `name` : Section name
- `key` : Section key
- `order` : Section order

`friday/recreationLogs/list`

Engagement recreation log documents:

- `engagementId` : Original engagement ID
- `clientId` : Client ID
- `engagementTypeId` : Engagement type ID
- `status` : Recreation status ("pending", "completed", "failed")
- `details` : Recreation details (old/new month, year, interval)
- `timestamp` : Recreation timestamp

`friday/templates/list`

Template documents:

- `name` : Template name
- `sections` : Array of section objects
- Each section contains questions

`friday/emails`

Email documents (from email receiving):

- `subject` : Email subject
- `from` : Sender email
- `to` : Recipient email(s)
- `cc` : CC email(s)

- `bcc` : BCC email(s)
- `date` : Email date
- `body` : Email body (HTML)
- `attachmentUrls` : Array of attachment URLs
- `allocated` : Boolean flag (false = pending allocation)

`friday/settings`

Settings document:

- `version` : Current app version
- `keys` : Object containing API keys (e.g., `keys.mwr`)

`friday/logs/list`

System log documents:

- `log_time` : Timestamp
- `email` : User email
- `review` : Log category
- `message` : Log message

User Roles & Permissions

Auditor Roles

1. **Partner** (`KBCSEHD91PdSh4StXOZe`)
    - Full access to all features
    - Can access Settings
    - Can manage users and teams
2. **Manager** (`hXbxDR3GfCHtrtjrn2wC`)
    - Can access Settings
    - Can manage engagements
    - Cannot access certain admin features
3. **Clerk** (`xNexLKjl1m3kPTG8BBaF`)
    - Limited access

- Cannot access Clients or Settings
- Can view and work on assigned engagements

Route Protection

The application uses multiple layers of route protection:

1. RequireAuth (`src/components/RequireAuth.js`)

- **Purpose**: Base authentication check
- **Logic**: Verifies `currentUser.cloud_id` exists
- **Action**: Redirects to `/login` if not authenticated
- **Applies To**: All protected routes

2. NonClerkRoute (`src/NonClerkRoute.js`)

- **Purpose**: Blocks Clerk role from accessing certain routes
- **Logic**: Checks `currentUser.role_name !== "Clerk"`
- **Blocked Routes**:
  - `/app/clients/*` - Client management
  - `/app/settings/*` - Settings (also blocked by ManagerPartnerLayout)
- **Action**: Redirects to `/app/unauthorized` if Clerk tries to access

3. ManagerPartnerLayout (`src/components/ManagerPartnerLayout.js`)

- **Purpose**: Restricts Settings to Manager/Partner roles only
- **Logic**: Checks `role_name === "Manager" || role_name === "Partner"`
- **Blocked Routes**: `/app/settings/*`
- **Action**: Redirects to `/app/unauthorized` if Clerk tries to access

4. TeamAuthRoute (`src/TeamAuthRoute.js`)

- **Purpose**: Team-based access control for engagements
- **Logic**: Verifies user is assigned to the engagement's team
- **Applies To**: `/app/engagements/:team/:id` (edit engagement)
- **Action**: Redirects if user not on engagement team

5. Client Access Control

- **Purpose**: Restricts client users to their assigned clients
- **Implementation**:

    - `isClientAuthorized(clientId)` hook checks if client is in user's `clients` array
    - `handleFilterRFIItemsForClient()` filters RFIs by assignment
- **Applies To**: All engagement views for client users
- **Action**: Shows only authorized engagements/RFIs

Inline Permission Checks in Components

In addition to route-level protection, components use **inline permission checks** to conditionally render features and control access. These checks use the `useGetUserType()` hook.

useGetUserType Hook

Location: `src/hooks/useGetUserType.js`

This hook provides permission checking functions:

- `isAuditor` - User is an auditor
- `isClient` - User is a client user
- `isPartner` - User is a Partner
- `isManager` - User is a Manager
- `isClerk` - User is a Clerk
- `isClientAdmin(clientId)` - Client user is admin for specific client
- `isClientAuthorized(clientId)` - Client user has access to specific client

Common Inline Permission Patterns

1. Conditional Rendering Based on User Type:

```
const { isAuditor, isClient, isClerk, isPartner } = useGetUserType();

// Only show for auditors
{isAuditor && <AuditorOnlyComponent />}

// Hide for clerks
{!isClerk && <ManagerPartnerFeature />}

// Show different UI for clients
{isClient ? <ClientView /> : <AuditorView />}
```

2. Feature Access Control:

```
// Example: Engagement rating (only client admins)
if (!isClientAdmin(engagement.client)) {
    setBarMessage("Only client admins can rate the engagement experience"
    return;
}

// Example: Edit permissions (auditors except clerks)
const canEdit = isAuditor && !isClerk;

// Example: Clerk approval check
const canClerkApprove = isAuditor && !isClerk ? true : engagement?.clerks
```

3. Field-Level Permissions:

```
// Example: DataGrid column editability
editable: (isAuditor && !isClerk) ? true : false

// Example: Button visibility
{isAuditor && !isClerk && <EditButton />}

// Example: Menu item visibility
{isPartner && <DeleteMenuItem />}
```

4. Action-Level Permissions:

```
// Example: Status change permissions
if (isAuditor) return; // auditor cannot change client status
if (!isAuditor || auditorReadOnly || !canClerkApprove) return; // client
if (currentStatus === 1 && isClerk) return; // clerk cannot change accept
```

Examples of Inline Permission Checks

EngagementItem Component (`src/components/EngagementItem.js`):

- **Line 347**: Only client admins can see rating dialog when engagement is 100% complete
- **Line 390-392**: Only auditors (not clerks) can see certain action buttons
- **Line 527**: Only auditors (not clerks) can see certain menu items
- **Line 807**: Deadline field is only editable by auditors (not clerks)
- **Line 977**: Only auditors (not clerks) can perform certain actions
- **Line 1021**: Only client admins can rate engagements
- **Line 1201**: Auditors cannot change client status

- **Line 1211**: Clients cannot change team status
- **Line 1442**: Clerks cannot perform certain actions

EngagementChecklistDialog Component (`src/components/EngagementChecklistDialog.js`):

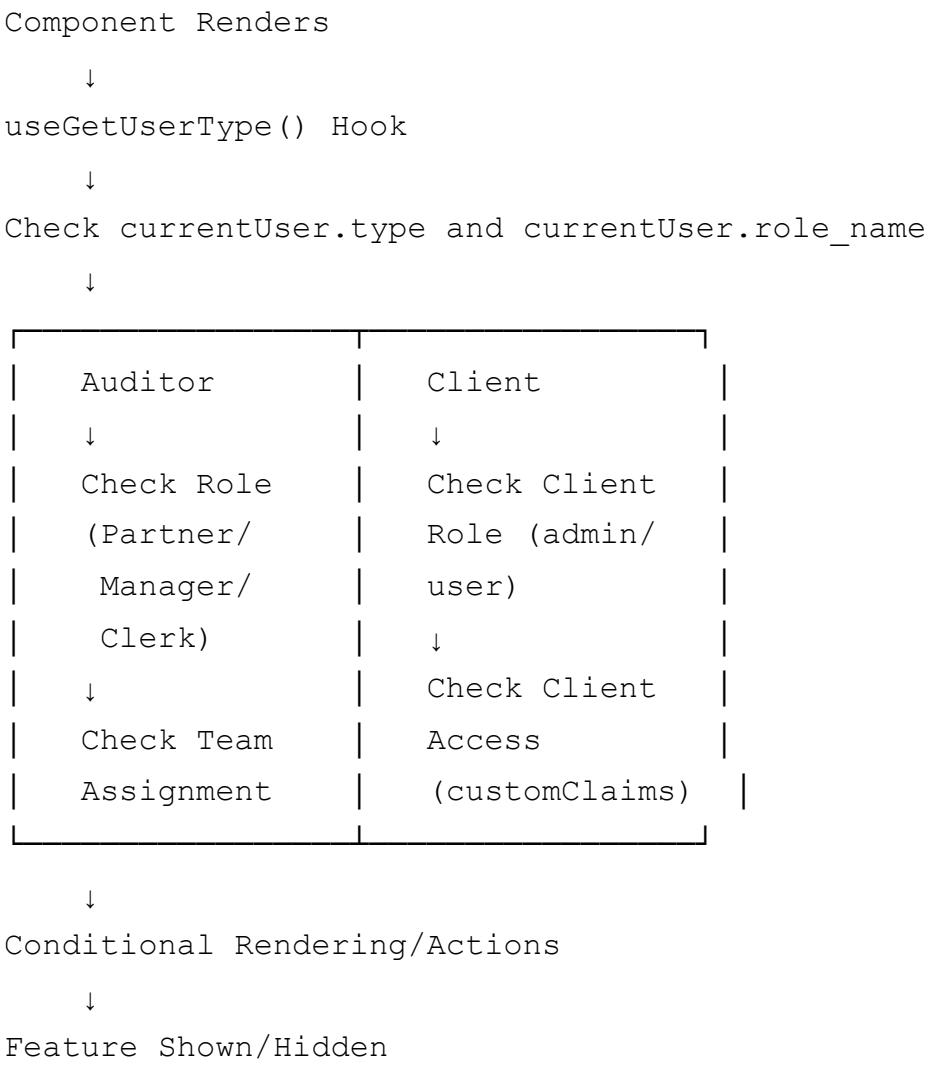
- **Line 765-789**: Complex permission logic for resolving/unresolving highlights:
  - Partners can always resolve/unresolve
  - Managers can resolve if not already resolved
  - Clerks cannot unresolve
  - Users can only unresolve their own partial resolutions

EngagementStageSelection Component (`src/components/EngagementStageSelection.js`):

- **Line 14**: Only Partners can set stage to "Close Out"
- **Line 18**: Clerks cannot change engagement stage
- **Line 23**: Cannot set stage to "Info Gathering" if commencement date has passed

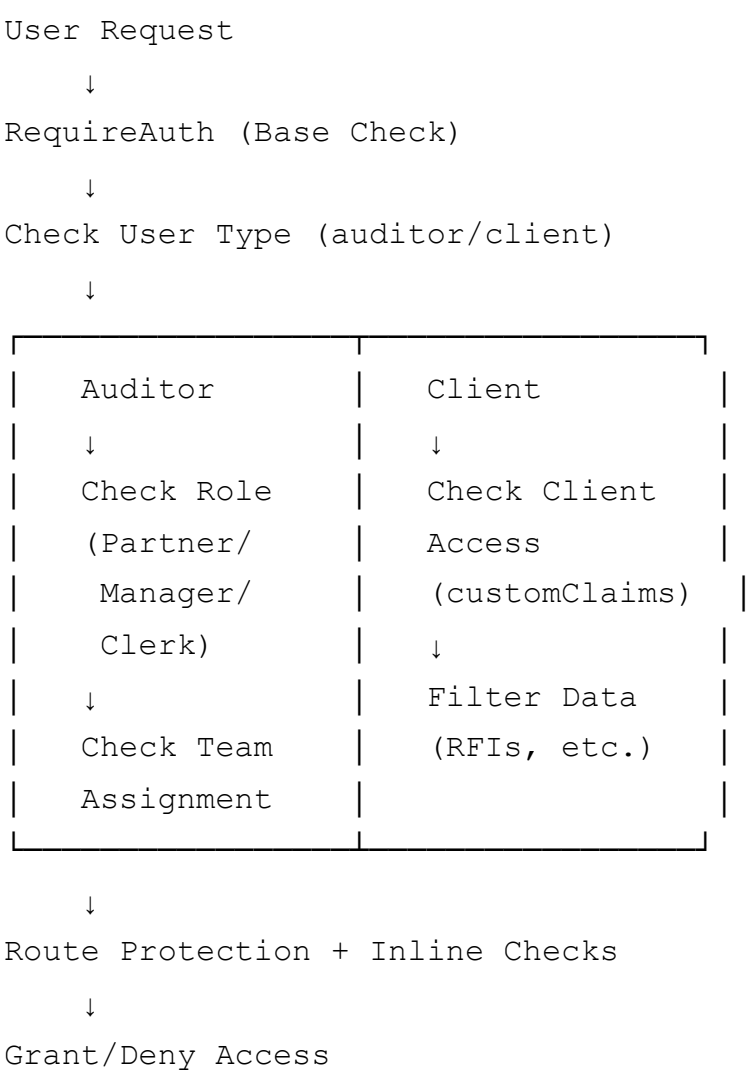
Permission Check Flow

Component-Level Permission Check:



Permission System Architecture

Permission Check Flow:



Permission Levels:

1. **Public**: Login page, password reset
2. **Authenticated**: Any logged-in user (auditor or client)
3. **Auditor Only**: Engagements, Clients, Settings (with role restrictions)
4. **Client Only**: Client engagement views (filtered)
5. **Role-Based**: Partner/Manager only (Settings, Client management)
6. **Team-Based**: Engagement access based on team assignment
7. **Client-Based**: Client user access based on `client_access` custom claims
8. **Inline Component Checks**: Feature-level permissions within components

Client Users

Client User Access Control

Client users have restricted access based on:

1. **Custom Claims**: `client_access` array in Firebase Auth custom claims
2. **Firestore User Document**: `clients` array contains client associations
3. **RFI Assignment**: Can only see RFIs where `users` array includes their `cloud_id`

Client User Permissions

The system distinguishes between **Admin** and **User** roles for client users. Each role has different capabilities and access levels.

Admin Role (`role: "admin"` in `clients` array):

What They Can See:

- ✅ All RFIs for their client's engagements (regardless of assignment)
- ✅ All engagement details and files
- ✅ All sections and questions
- ✅ Engagement progress and status
- ✅ All client users for their client
- ✅ Master weekly summary emails (section-level summaries)

What They Can Do:

- ✅ Respond to any RFI for their client (not just assigned ones)
- ✅ Upload attachments to any RFI response
- ✅ Rate engagement experience (when engagement is 100% complete)
- ✅ View engagement rating dialog
- ✅ Access all engagement features available to client users

What They Cannot Do:

- ❌ Create or edit engagements
- ❌ Access Settings, Clients, or Templates
- ❌ Edit engagement details
- ❌ Change engagement status
- ❌ Assign RFIs to other users

User Role (`role: "user"` in `clients` array):

What They Can See:

- ✅ Only RFIs explicitly assigned to them (`users` array includes their `cloud_id`)
- ✅ Engagement details for their client (but filtered RFI view)
- ✅ Only sections/questions they're assigned to
- ✅ Their own responses and attachments

What They Can Do:

- ✅ Respond to RFIs assigned to them only
- ✅ Upload attachments to assigned RFI responses
- ✅ View engagement details (filtered view)

What They Cannot Do:

- ❌ See RFIs not assigned to them
- ❌ Respond to unassigned RFIs
- ❌ Rate engagement experience (only admins can)
- ❌ View engagement rating dialog
- ❌ Create or edit engagements
- ❌ Access Settings, Clients, or Templates
- ❌ See all RFIs for their client

Permission Check Examples:

RFI Visibility:

```
// Admin: sees all RFIs
if (isClientAdmin(clientId)) {
  // Show all RFIs for client
}

// User: sees only assigned RFIs
else {
  // Filter: rfi.users.includes(currentUser.cloud_id)
  const filteredRFIs = rfis.filter(rfi =>
    rfi.users.includes(currentUser.cloud_id)
  );
}
```

Engagement Rating:

```
// Only client admins can rate
if (!isClientAdmin(engagement.client)) {
  setBarMessage("Only client admins can rate the engagement experience");
  return;
}
// Show rating dialog
```

Weekly Email Distribution:

- **Admin Users**: Receive master summary email with all RFIs grouped by section
- **User Role**: Receive individual email with only their assigned RFIs

Implementation Details:

RFI Filtering (`handleFilterRFIItemsForClient` in `src/globalFunctions/clientInfo.js`):

```
// Filters RFI section items based on user assignment
// Admin: no filtering (sees all)
// User: only sees items where users array includes their cloud_id
const updatedRfiList = rfiList.map((rfiItem) => ({
  ...rfiItem,
  section_items: rfiItem.section_items.filter((sectionItem) =>
    sectionItem?.users?.includes(currentUserCloudId)
  ),
}));
```

Client Admin Check (`isClientAdmin` in `src/hooks/useGetUserType.js`):

```
const isClientAdmin = (clientId) => {
  return currentUser.type === "client" &&
    currentUser.clients.find(client =>
      client.cloud_id === clientId &&
      client.role === "admin"
    )
}
```

Client User Route Access

- **Allowed Routes**:
  - `/app/engagements` - View client engagements (filtered)
  - `/app/engagements/client/:id/:engagementId` - View engagement details
- **Blocked Routes**:
  - `/app/clients/*` - Cannot access client management
  - `/app/settings/*` - Cannot access settings
  - `/app/engagements/new` - Cannot create engagements
  - `/app/engagements/:team/:id` - Cannot edit engagements

Client User Data Filtering

Engagements Filtering:

```
// Only engagements where client matches user's client_access
engagements.filter(engagement =>
  currentUser.clients.some(client =>
    client.cloud_id === engagement.client &&
    client.status === "active"
  )
)
```

RFI Filtering:

```
// For admin: all RFIs for client
// For user: only RFIs where users array includes their cloud_id
if (isClientAdmin) {
  // Show all RFIs
} else {
  // Filter: rfi.users.includes(currentUser.cloud_id)
}
```

Implementation: `handleFilterRFIItemsForClient()` in `src/globalFunctions/clientInfo.js`

Environment Variables

Frontend (.env)

```
# Friday Firebase Configuration
REACT_APP_FRIDAY_FIREBASE_API_KEY=
REACT_APP_FRIDAY_FIREBASE_AUTH_DOMAIN=
REACT_APP_FRIDAY_FIREBASE_PROJECT_ID=friday-a372b
REACT_APP_FRIDAY_FIREBASE_STORAGE_BUCKET=friday-a372b.appspot.com
REACT_APP_FRIDAY_FIREBASE_MESSAGING_SENDER_ID=
REACT_APP_FRIDAY_FIREBASE_APP_ID=
REACT_APP_FRIDAY_FIREBASE_MEASUREMENT_ID=

# My Review Firebase Configuration
REACT_APP_MY_REVIEW_FIREBASE_API_KEY=
REACT_APP_MY_REVIEW_FIREBASE_AUTH_DOMAIN=
REACT_APP_MY_REVIEW_FIREBASE_PROJECT_ID=audit-7ec47
```

```
REACT_APP_MY_REVIEW_FIREBASE_STORAGE_BUCKET=
REACT_APP_MY_REVIEW_FIREBASE_MESSAGING_SENDER_ID=
REACT_APP_MY_REVIEW_FIREBASE_APP_ID=
REACT_APP_MY_REVIEW_FIREBASE_MEASUREMENT_ID=
```

Backend (functions/.env)

```
# Email Configuration
EMAIL_HOST=
EMAIL_PORT=
EMAIL_USER=
EMAIL_PASSWORD=

# My Review Integration
MWR_SERVER_KEY=

# User Sync Configuration
USER_EMAIL_SYNC_KEY=

# Email Processing
EMAIL_SENDING_SUBSCRIPTION_NAME=email-sending

# Bearer Token for Email Receiving
EXPECTED_BEARER_TOKEN=7f7d4fa3-ea07-4541-9ed2-b432a6859bb8

# Other service keys as needed
```

Setting Up a New Staging/Development Environment

This section provides a comprehensive guide for setting up a new staging or development environment from scratch. This is essential when creating a new Firebase project or setting up a separate environment for testing.

Prerequisites

- Node.js 20 or higher
- npm or yarn
- Firebase CLI installed globally (`npm install -g firebase-tools`)
- Google Cloud account with billing enabled

- Google Drive account for integration
- Access to My Review Firebase project (for integration)

Step 1: Create New Firebase Project

1. **Go to Firebase Console**: https://console.firebase.google.com/
2. **Click "Add Project"**
3. **Enter Project Name**: e.g., `friday-staging` or `friday-dev`
4. **Enable Google Analytics** (optional but recommended)
5. **Select Analytics Account** (if enabled)
6. **Click "Create Project"**
7. **Wait for project creation** (takes ~1 minute)

Step 2: Enable Firebase Services

In the Firebase Console, enable the following services:

Authentication:

1. Go to **Authentication** → **Get Started**
2. Enable **Email/Password** provider
3. Enable **Google** provider
    - Add authorized domains (localhost, your staging domain)
    - Configure OAuth consent screen if needed

Firestore Database:

1. Go to **Firestore Database** → **Create Database**
2. Choose **Start in production mode** (we'll add rules later)
3. Select **Cloud Firestore location** (choose closest to your users)
4. Click **Enable**

Storage:

1. Go to **Storage** → **Get Started**
2. Choose **Start in production mode** (we'll add rules later)
3. Select **Storage location** (should match Firestore location)
4. Click **Done**

Cloud Functions:

1. Go to **Functions** → **Get Started**
2. Review billing requirements (Cloud Functions require Blaze plan)
3. Enable billing if not already enabled

Step 3: Set Up Firestore Security Rules

1. Go to **Firestore Database → Rules**
2. Add the following security rules (adjust based on your needs):

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Helper function to check if user is authenticated
    function isAuthenticated() {
      return request.auth != null;
    }

    // Helper function to check if user is active
    function isUserActive() {
      return isAuthenticated() &&
             get(/databases/$(database)/documents/friday/users/list/$(req
    }

    // Users collection
    match /friday/users/list/{userId} {
      allow read: if isAuthenticated();
      allow write: if isAuthenticated() && request.auth.uid == userId;
    }

    // Engagements collection
    match /friday/engagements/list/{engagementId} {
      allow read: if isAuthenticated();
      allow write: if isAuthenticated() && isUserActive();

      // RFIs subcollection
      match /rfis/{rfiId} {
        allow read: if isAuthenticated();
        allow write: if isAuthenticated() && isUserActive();
      }
    }

    // Add more rules as needed for your collections
  }
}
```

3. Click **Publish**

Note: These are basic rules. Adjust based on your security requirements and user roles.

Step 4: Set Up Firestore Indexes

The application requires several composite indexes. Create them as needed:

Method 1: Automatic (Recommended)

- Run queries in the app
- Firebase will show error messages with links to create missing indexes
- Click the links to create indexes automatically

Method 2: Manual

1. Go to **Firestore Database → Indexes**
2. Click **Create Index**
3. Add indexes for common queries:
    - Collection: `friday/engagements/list`
        - Fields: `team` (Ascending), `status` (Ascending), `stage` (Ascending)
    - Collection: `friday/engagements/list`
        - Fields: `client` (Ascending), `status` (Ascending), `year` (Ascending)
    - Collection: `friday/users/list`
        - Fields: `type` (Ascending), `status` (Ascending)
    - Add more indexes as needed based on your queries

Common Indexes Needed:

```
{
  "indexes": [
    {
      "collectionGroup": "list",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "team", "order": "ASCENDING" },
        { "fieldPath": "status", "order": "ASCENDING" },
        { "fieldPath": "stage", "order": "ASCENDING" }
      ]
    },
    {
      "collectionGroup": "list",
      "queryScope": "COLLECTION",
      "fields": [
        { "fieldPath": "client", "order": "ASCENDING" },
```

```
        { "fieldPath": "status", "order": "ASCENDING" },
        { "fieldPath": "year", "order": "ASCENDING" }
      ]
    }
  ]
}
```

Step 5: Set Up Firebase Storage Security Rules

1. Go to **Storage → Rules**
2. Add the following security rules:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    // Helper function to check authentication
    function isAuthenticated() {
      return request.auth != null;
    }

    // RFI Response Attachments
    match /{clientId}/{engagementId}/{rfiId}/{questionId}/{fileName} {
      allow read: if isAuthenticated();
      allow write: if isAuthenticated();
    }

    // Engagement Master Files
    match /Engagement Templates/{entityType}/{fileName} {
      allow read: if isAuthenticated();
      allow write: if isAuthenticated();
    }

    // User CVs
    match /User CVs/{userId}/{fileName} {
      allow read: if isAuthenticated();
      allow write: if isAuthenticated() && request.auth.uid == userId;
    }

    // Email Attachments (Temporary)
    match /temp_email_attachments/{fileName} {
      allow read: if isAuthenticated();
      allow write: if false; // Only backend can write
    }
```

```
    // Engagement Details Files
    match /EngagementDetails/{engagementId}/{fileName} {
      allow read: if isAuthenticated();
      allow write: if isAuthenticated();
    }
  }
}
```

3. Click **Publish**

Step 6: Set Up Google Drive Integration

Create Google Drive Folder:

1. Go to **Google Drive**: https://drive.google.com/
2. Create a new folder for engagement letters (e.g., "Friday Engagement Letters - Staging")
3. **Right-click folder → Share → Get link**
4. Copy the **Folder ID** from the URL (e.g., `1DcOh90HMoy4Ntm6K7jl0EW-A8S-uIBwG`)

Set Up Google Cloud Service Account:

1. Go to **Google Cloud Console**: https://console.cloud.google.com/
2. Select your Firebase project (or create a new one)
3. Go to **IAM & Admin → Service Accounts**
4. Click **Create Service Account**
5. Enter **Service account name**: `friday-drive-service`
6. Click **Create and Continue**
7. Grant **Editor** role (or more specific roles)
8. Click **Continue → Done**
9. Click on the created service account
10. Go to **Keys** tab → **Add Key → Create New Key**
11. Choose **JSON** format
12. Download the key file → Save as `functions/serviceAccountCloudKey.json`

Enable Google Drive API:

1. Go to **APIs & Services → Library**
2. Search for **Google Drive API**
3. Click **Enable**
4. Search for **Google Docs API**
5. Click **Enable**

Grant Folder Access:

1. Open the downloaded service account JSON file
2. Copy the **client_email** value
3. Go back to Google Drive folder
4. **Right-click folder → Share**
5. Paste the service account email
6. Grant **Editor** access
7. Click **Send**

Update Code:

- Update `functions/index.js` with your folder ID:

```
const folderId = 'YOUR_FOLDER_ID_HERE';
```

Step 7: Set Up Service Account Keys

Friday Firebase Service Account:

1. Go to **Firebase Console → Project Settings → Service Accounts**
2. Click **Generate New Private Key**
3. Download JSON file → Save as `functions/serviceAccountKey.json`

My Review Firebase Service Account:

1. Go to **My Review Firebase Console** (separate project)
2. Follow same steps as above
3. Save as `functions/serviceAccountKeyMyReview.json`

Google Cloud Service Account:

- Already created in Step 6, saved as `functions/serviceAccountCloudKey.json`

⚠️ **Security Note**: Never commit these files to version control. Add to `.gitignore`:

```
functions/serviceAccountKey*.json
.env
.env.local
```

Step 8: Set Up Environment Variables

Frontend Environment Variables (.env file in root):

Create `.env` file in the root directory:

```
# Friday Firebase Configuration
REACT_APP_FRIDAY_FIREBASE_API_KEY=your-api-key
REACT_APP_FRIDAY_FIREBASE_AUTH_DOMAIN=your-project.firebaseapp.com
REACT_APP_FRIDAY_FIREBASE_PROJECT_ID=your-project-id
REACT_APP_FRIDAY_FIREBASE_STORAGE_BUCKET=your-project.appspot.com
REACT_APP_FRIDAY_FIREBASE_MESSAGING_SENDER_ID=your-sender-id
REACT_APP_FRIDAY_FIREBASE_APP_ID=your-app-id
REACT_APP_FRIDAY_FIREBASE_MEASUREMENT_ID=your-measurement-id

# My Review Firebase Configuration
REACT_APP_MY_REVIEW_FIREBASE_API_KEY=my-review-api-key
REACT_APP_MY_REVIEW_FIREBASE_AUTH_DOMAIN=my-review-project.firebaseapp.cc
REACT_APP_MY_REVIEW_FIREBASE_PROJECT_ID=my-review-project-id
REACT_APP_MY_REVIEW_FIREBASE_STORAGE_BUCKET=my-review-project.appspot.com
REACT_APP_MY_REVIEW_FIREBASE_MESSAGING_SENDER_ID=my-review-sender-id
REACT_APP_MY_REVIEW_FIREBASE_APP_ID=my-review-app-id
REACT_APP_MY_REVIEW_FIREBASE_MEASUREMENT_ID=my-review-measurement-id

# MUI License (if using premium features)
REACT_APP_MUI=your-mui-license-key

# Public URL (for service worker)
PUBLIC_URL=/
```

How to Get Firebase Config Values:

1. Go to **Firebase Console → Project Settings → General**
2. Scroll to **Your apps** section
3. Click **Web app** icon (</>) or **Add app → Web**
4. Register app (or use existing)
5. Copy config values from the code snippet

Backend Environment Variables (.env file in functions/):

Create `.env` file in `functions/` directory:

```
# Email Configuration
AUDIT_EMAIL_USER=your-email@gmail.com
AUDIT_EMAIL_PASS=your-app-password

# Google Apps Script User Sync
```

```
USER_EMAIL_SYNC_KEY=your-google-apps-script-key

# My Review API Key
MWR_SERVER_KEY=your-my-review-api-key

# Email Receiving Bearer Token (generate a secure random token)
EXPECTED_BEARER_TOKEN=your-secure-random-token-here

# Port (optional, defaults to 5000)
PORT=5000
```

How to Get These Values:

- **Email Credentials**: Use Gmail App Password or SMTP credentials
- **USER_EMAIL_SYNC_KEY**: Get from Google Apps Script that provides user list
- **MWR_SERVER_KEY**: Get from My Review project settings
- **EXPECTED_BEARER_TOKEN**: Generate a secure random token (e.g., UUID)

Generate Secure Bearer Token:

```
# Using Node.js
node -e "console.log(require('crypto').randomUUID())"

# Or use online UUID generator
```

Step 9: Update CORS Configuration

Update `functions/index.js` with your staging domain:

```
const allowedOrigins = [
    'http://localhost:3000',
    'https://localhost:3000',
    'localhost:3000',
    'https://your-staging-domain.netlify.app',
    'https://www.your-staging-domain.netlify.app',
    'https://your-staging-domain.netlify.app/',
    'https://us-central1-your-project-id.cloudfunctions.net'
];
```

Step 10: Set Up Cloud Functions

Initialize Firebase Functions:

```
cd functions
npm install
```

Deploy Functions:

```
# Login to Firebase
firebase login

# Set Firebase project
firebase use your-project-id

# Deploy all functions
firebase deploy --only functions

# Or deploy specific function
firebase deploy --only functions:api
```

Set Environment Variables for Functions:

```
# Set email credentials
firebase functions:config:set audit.email.user="your-email@gmail.com"
firebase functions:config:set audit.email.pass="your-app-password"

# Set user sync key
firebase functions:config:set user.email.sync.key="your-key"

# Set My Review API key
firebase functions:config:set mwr.server.key="your-key"

# Get all config
firebase functions:config:get
```

Note: For newer Firebase projects, use Secret Manager instead:

```
# Create secrets
firebase functions:secrets:set AUDIT_EMAIL_USER
firebase functions:secrets:set AUDIT_EMAIL_PASS
firebase functions:secrets:set USER_EMAIL_SYNC_KEY
firebase functions:secrets:set MWR_SERVER_KEY
```

Step 11: Set Up Pub/Sub Topics

Create Pub/Sub Topics:

1. Go to **Google Cloud Console → Pub/Sub → Topics**
2. Create the following topics:
    - `daily-backup`
    - `daily-user-sync`
    - `daily-engagement-check`
    - `daily-rfi-notifications`
    - `daily-manager-clerk-notifications`
    - `weekly-client-engagement-emails`
    - `yearly-engagement-creation`
    - `monthly-engagement-creation`
    - `process-engagement-recreation-batch`
    - `email-content-generation`
    - `email-sending`

Or use gcloud CLI:

```
# Set project
gcloud config set project your-project-id

# Create topics
gcloud pubsub topics create daily-backup
gcloud pubsub topics create daily-user-sync
gcloud pubsub topics create daily-engagement-check
gcloud pubsub topics create daily-rfi-notifications
gcloud pubsub topics create daily-manager-clerk-notifications
gcloud pubsub topics create weekly-client-engagement-emails
gcloud pubsub topics create yearly-engagement-creation
gcloud pubsub topics create monthly-engagement-creation
gcloud pubsub topics create process-engagement-recreation-batch
gcloud pubsub topics create email-content-generation
gcloud pubsub topics create email-sending
```

Step 12: Set Up Cloud Scheduler Jobs

Create Cloud Scheduler Jobs:

1. Go to **Google Cloud Console → Cloud Scheduler**

2. Click **Create Job** for each scheduled task:

Daily Backup:

- Name: `daily-backup`
- Frequency: `0 2 * * *` (2 AM daily)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `daily-backup`
- Payload: `{}`

Daily User Sync:

- Name: `daily-user-sync`
- Frequency: `0 3 * * *` (3 AM daily)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `daily-user-sync`
- Payload: `{}`

Daily Engagement Check:

- Name: `daily-engagement-check`
- Frequency: `0 4 * * *` (4 AM daily)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `daily-engagement-check`
- Payload: `{}`

Daily RFI Notifications:

- Name: `daily-rfi-notifications`
- Frequency: `0 5 * * *` (5 AM daily)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `daily-rfi-notifications`
- Payload: `{}`

Daily Manager/Clerk Notifications:

- Name: `daily-manager-clerk-notifications`
- Frequency: `0 6 * * *` (6 AM daily)

- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `daily-manager-clerk-notifications`
- Payload: `{}`

Weekly Client Engagement Emails:

- Name: `weekly-client-engagement-emails`
- Frequency: `0 9 * * 1` (9 AM every Monday)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `weekly-client-engagement-emails`
- Payload: `{}`

Yearly Engagement Creation:

- Name: `yearly-engagement-creation`
- Frequency: `0 0 1 1 *` (Midnight on January 1st)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `yearly-engagement-creation`
- Payload: `{}`

Monthly Engagement Creation:

- Name: `monthly-engagement-creation`
- Frequency: `0 0 1 * *` (Midnight on 1st of each month)
- Timezone: `UTC`
- Target: **Pub/Sub**
- Topic: `monthly-engagement-creation`
- Payload: `{}`

Step 13: Set Up Backup Bucket

Create Cloud Storage Bucket:

1. Go to **Google Cloud Console → Cloud Storage → Buckets**
2. Click **Create Bucket**
3. Name: `your-project-dailybackups` (e.g., `friday-staging-dailybackups`)
4. Location: Choose same as Firestore
5. Storage class: **Standard**
6. Access control: **Uniform**

7. Click **Create**

Update Code:

- Update `functions/index.js`:

```
const backup_bucket = "gs://your-project-dailybackups";
```

Step 14: Set Up Initial Data

Create Initial Firestore Documents:

1. **Settings Document:**
   - Path: `friday/settings`
   - Data:

```
{
  "version": "1.1.863",
  "keys": {
    "mwr": "your-my-review-api-key"
  }
}
```

2. **User Permissions (Roles):**
   - Path: `friday/users/permissions`
   - Create documents for:
     - Partner
     - Manager
     - Clerk
   - Structure:

```
{
  "name": "Partner",
  "key": "partner",
  "roles": {
    // Permission object
  }
}
```

3. **Teams:**

- Path: `friday/teams/list`
- Create team documents as needed

4. **Entity Types:**
   - Path: `friday/clients/entity_types`
   - Create entity type documents (e.g., Corporation, Partnership)
5. **Engagement Types:**
   - Path: `friday/clients/engagement_types`
   - Create engagement type documents

Step 15: Update Code References

Update Hardcoded Values:

1. **Google Drive Folder ID:**
   - File: `functions/index.js`
   - Search for: `const folderId = '1DcOh90HMoy4Ntm6K7jl0EW-A8S-uIBwG';`
   - Replace with your folder ID
2. **Storage Bucket:**
   - File: `functions/index.js`
   - Search for: `storageBucket: 'friday-a372b.appspot.com'`
   - Replace with your bucket name
3. **Backup Bucket:**
   - File: `functions/index.js`
   - Search for: `const backup_bucket = "gs://fridaydailybackups";`
   - Replace with your backup bucket
4. **My Review API Endpoint:**
   - File: `src/App.js`
   - Search for: `https://us-central1-audit-7ec47.cloudfunctions.net/api/generate-custom-token`
   - Update if My Review project changes
5. **Google Apps Script URL:**
   - File: `functions/userUtils.js`

- Search for: Google Apps Script URL
- Update if script URL changes

Step 16: Test the Setup

Test Frontend:

```
npm start
```

- Should start on `http://localhost:3000`
- Check browser console for errors
- Try logging in

Test Backend:

```
cd functions
npm run serve
```

- Functions should start locally
- Test API endpoints

Test Firebase Connection:

- Try creating a test user in Firebase Console
- Try creating a test document in Firestore
- Try uploading a test file to Storage

Test Google Drive:

- Try generating an engagement letter
- Check if file appears in Google Drive folder

Test Pub/Sub:

- Manually trigger a Pub/Sub topic
- Check function logs for execution

Step 17: Deploy to Staging

Deploy Frontend:

```
# Build for production
npm run build
```

```
# Deploy to Netlify (or your hosting)
# Or use Firebase Hosting
firebase deploy --only hosting
```

Deploy Backend:

```
firebase deploy --only functions
```

Update CORS:

- Add your staging domain to `allowedOrigins` in `functions/index.js`
- Redeploy functions

Step 18: Verify Everything Works

Checklist:

- ☐ Firebase Auth working (login/logout)
- ☐ Firestore reads/writes working
- ☐ Storage uploads/downloads working
- ☐ Cloud Functions deployed and accessible
- ☐ Pub/Sub topics created
- ☐ Cloud Scheduler jobs created and enabled
- ☐ Google Drive integration working
- ☐ Email sending working (test email)
- ☐ My Review integration working (if applicable)
- ☐ CORS configured correctly
- ☐ Environment variables set
- ☐ Service account keys in place
- ☐ Security rules configured
- ☐ Indexes created

Common Issues and Solutions

Issue: "Permission denied" errors

- **Solution**: Check Firestore/Storage security rules
- **Solution**: Verify user authentication
- **Solution**: Check user status is "active"

Issue: CORS errors

- **Solution**: Add domain to `allowedOrigins` in `functions/index.js`
- **Solution**: Redeploy functions after CORS update

Issue: Google Drive "403 Forbidden"

- **Solution**: Verify service account has access to folder
- **Solution**: Check Google Drive API is enabled
- **Solution**: Verify service account key is correct

Issue: Functions not executing

- **Solution**: Check Pub/Sub topics exist
- **Solution**: Verify Cloud Scheduler jobs are enabled
- **Solution**: Check function logs for errors

Issue: Email not sending

- **Solution**: Verify SMTP credentials are correct
- **Solution**: Check email service status
- **Solution**: Verify environment variables are set

Issue: Index errors

- **Solution**: Create missing indexes (Firebase will provide links)
- **Solution**: Wait for indexes to build (can take several minutes)

Environment-Specific Configuration

Development Environment:

- Use local Firebase emulators
- Use test Google Drive folder
- Use test email addresses
- Disable scheduled jobs (or use longer intervals)

Staging Environment:

- Use separate Firebase project
- Use separate Google Drive folder
- Use staging email addresses
- Enable scheduled jobs with test intervals

Production Environment:

- Use production Firebase project
- Use production Google Drive folder

- Use production email addresses
- Enable all scheduled jobs with production intervals

Development Setup

Prerequisites

- Node.js 20 (for functions)
- npm or yarn
- Firebase CLI: `npm install -g firebase-tools`
- Firebase project access

Installation

1. **Clone the repository**

   ```
   git clone <repository-url>
   cd friday
   ```

2. **Install frontend dependencies**

   ```
   npm install
   ```

3. **Install backend dependencies**

   ```
   cd functions
   npm install
   cd ..
   ```

4. **Set up environment variables**

   - Create `.env` file in root directory with frontend variables
   - Create `functions/.env` with backend variables
   - Add service account keys to `functions/` directory

5. **Login to Firebase**

   ```
   firebase login
   ```

6. **Set Firebase project**

```
firebase use friday-a372b
```

Running Locally

Frontend:

```
npm start
```

Runs on http://localhost:3000

Backend Functions (Emulator):

```
cd functions
npm run serve
```

Full Firebase Emulator Suite:

```
firebase emulators:start
```

Building for Production

```
npm run build
```

Outputs to `build/` directory

Deployment

Frontend Deployment

The app is configured for deployment on **Netlify**:

- Build command: `npm run build`
- Publish directory: `build`
- Redirects configured in `public/_redirects`

Backend Deployment

Deploy Firebase Functions:

```
cd functions
firebase deploy --only functions
```

Deploy specific function:

```
firebase deploy --only functions:api
```

Firebase Hosting (Alternative)

```
firebase deploy --only hosting
```

Service Account Keys

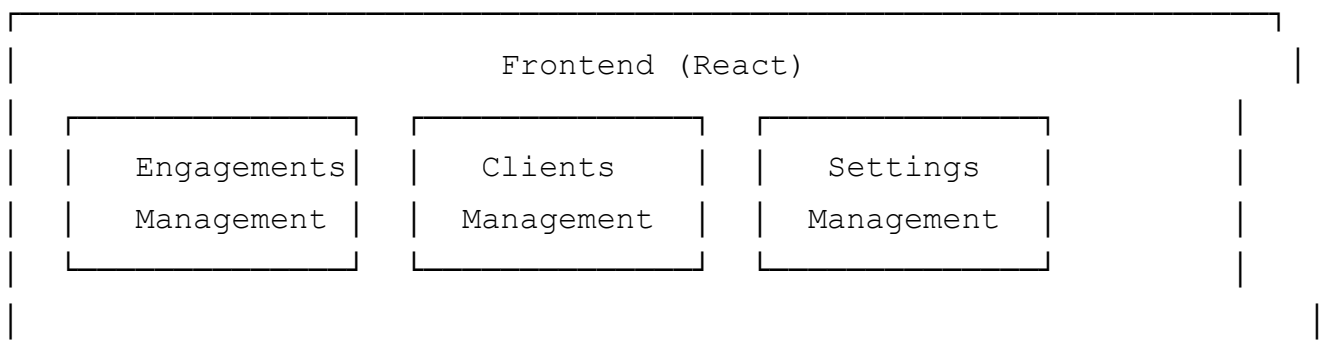
Required service account keys in `functions/`:

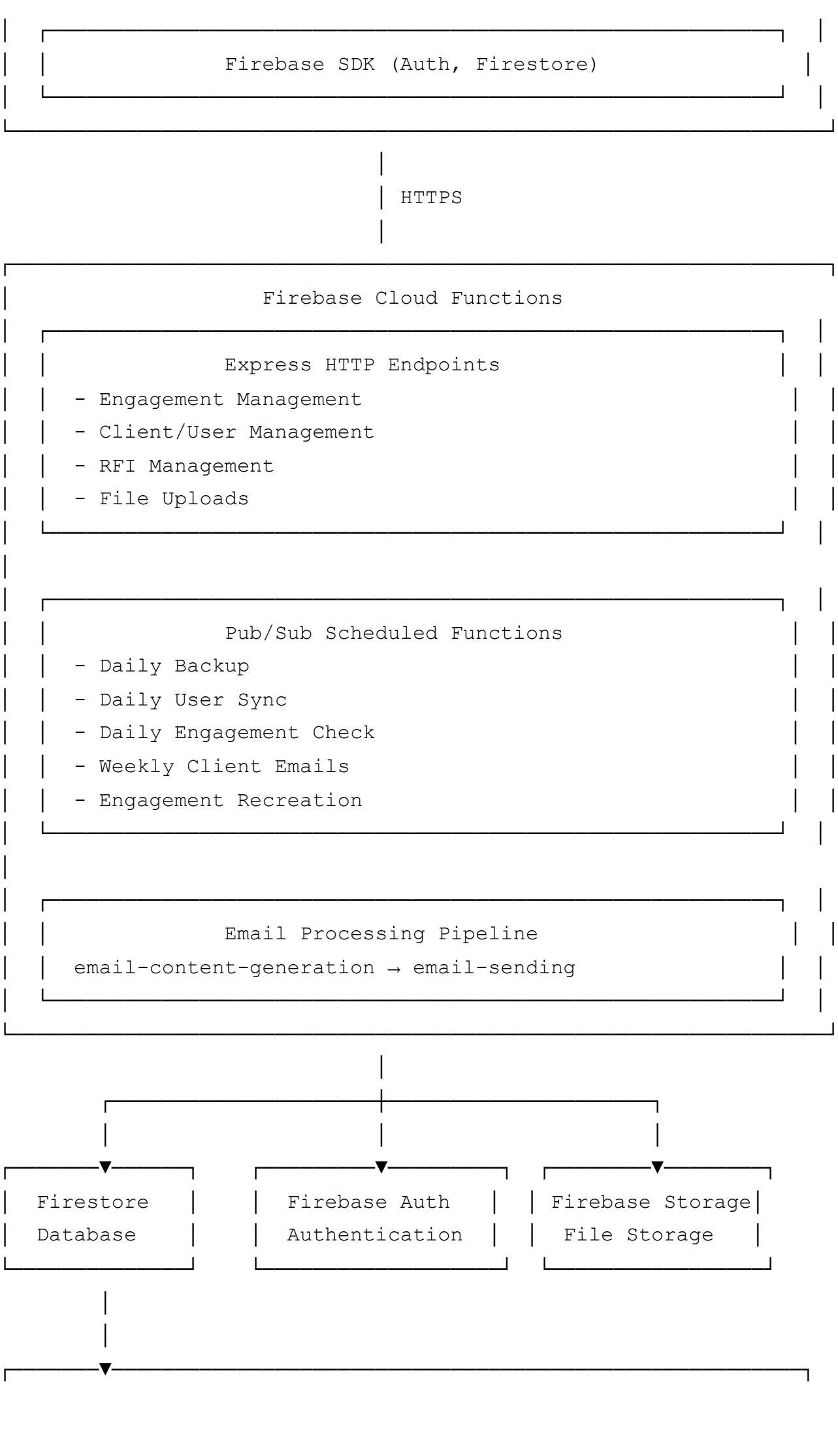
- `serviceAccountKey.json` - Friday Firebase admin
- `serviceAccountKeyMyReview.json` - My Review Firebase admin
- `serviceAccountCloudKey.json` - Google Cloud/Drive access

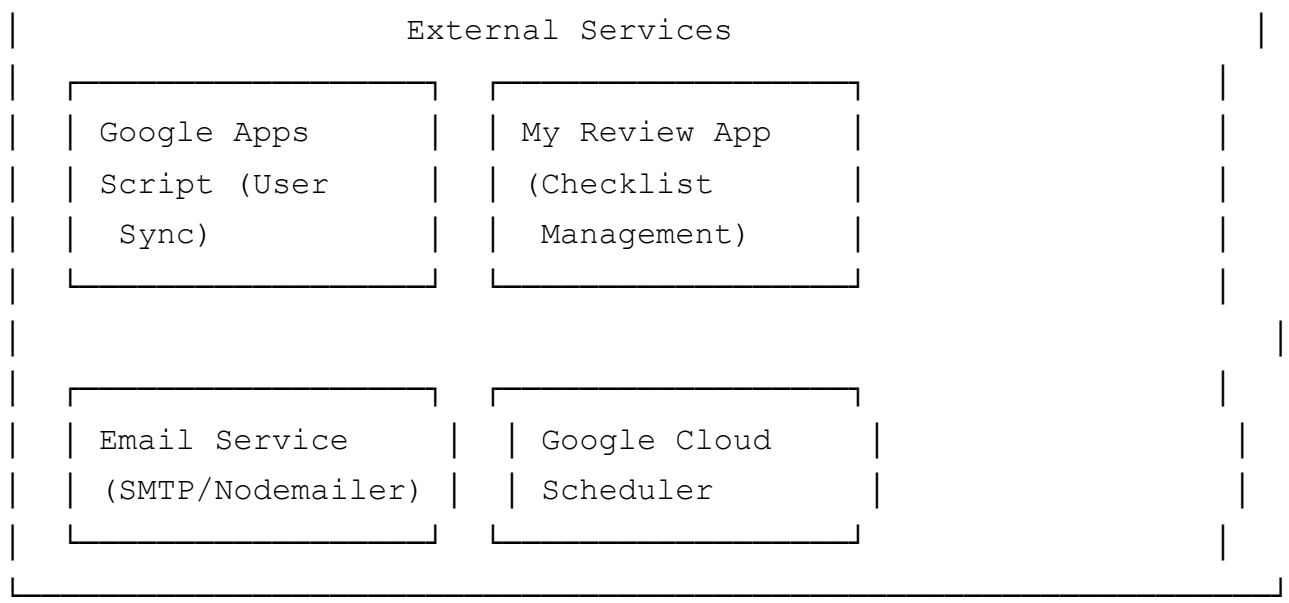
⚠️ **Security Note**: Never commit service account keys to version control. Use environment variables or secure secret management in production.

System Architecture

High-Level Architecture Diagram




Firebase SDK (Auth, Firestore)
HTTPS
Firebase Cloud Functions
Express HTTP Endpoints
- Engagement Management
- Client/User Management
- RFI Management
- File Uploads
Pub/Sub Scheduled Functions
- Daily Backup
- Daily User Sync
- Daily Engagement Check
- Weekly Client Emails
- Engagement Recreation
Email Processing Pipeline
email-content-generation → email-sending
Firestore Database
Firebase Auth Authentication
Firebase Storage File Storage



Data Flow Diagram

User Authentication Flow:

```
User Login (Google/Email)
    ↓
Firebase Auth
    ↓
Check Firestore User Exists
    ↓
Validate User Status (active/inactive)
    ↓
Load User Data to AuthContext
    ↓
For Auditors: Authenticate with My Review
    ↓
Set Custom Claims (for client users)
    ↓
Access Granted
```

Engagement Creation Flow:

```
User Creates Engagement
    ↓
Frontend: EngagementsEdit Component
    ↓
Save to Firestore (friday/engagements/list)
    ↓
Create Sections (subcollection)
```

```
    ↓
Create RFIs from Template (subcollection)
    ↓
Assign Client Users
    ↓
Send Initial Email to Client
    ↓
Engagement Active
```

RFI Response Flow:

```
Client User Responds to RFI
    ↓
Update RFI Document (responses array)
    ↓
Upload Attachments (Firebase Storage)
    ↓
Update RFI Status
    ↓
Notify Engagement Team (Pub/Sub)
    ↓
Send Email Notifications
```

Cloud Scheduler Configuration

The following Cloud Scheduler jobs need to be configured in Google Cloud Console:

1. **Daily Backup**
    - Schedule: `0 2 * * *` (2 AM daily)
    - Pub/Sub Topic: `daily-backup`
    - Timezone: UTC
2. **Daily User Sync**
    - Schedule: `0 3 * * *` (3 AM daily)
    - Pub/Sub Topic: `daily-user-sync`
    - Timezone: UTC
3. **Daily Engagement Check**
    - Schedule: `0 4 * * *` (4 AM daily)
    - Pub/Sub Topic: `daily-engagement-check`
    - Timezone: UTC

4. **Daily RFI Notifications**
    - Schedule: `0 5 * * *` (5 AM daily)
    - Pub/Sub Topic: `daily-rfi-notifications`
    - Timezone: UTC
5. **Daily Manager/Clerk Notifications**
    - Schedule: `0 6 * * *` (6 AM daily)
    - Pub/Sub Topic: `daily-manager-clerk-notifications`
    - Timezone: UTC
6. **Weekly Client Engagement Emails**
    - Schedule: `0 9 * * 1` (9 AM every Monday)
    - Pub/Sub Topic: `weekly-client-engagement-emails`
    - Timezone: UTC
7. **Yearly Engagement Creation**
    - Schedule: `0 0 1 1 *` (Midnight on January 1st)
    - Pub/Sub Topic: `yearly-engagement-creation`
    - Timezone: UTC
8. **Monthly Engagement Creation**
    - Schedule: `0 0 1 * *` (Midnight on 1st of each month)
    - Pub/Sub Topic: `monthly-engagement-creation`
    - Timezone: UTC

Note: These schedules are examples. Actual schedules should be configured based on business requirements and timezone preferences.

Security Measures

Authentication Security

Firebase Authentication:

- **Google Sign-In**: OAuth 2.0 authentication via Firebase
- **Email/Password**: Secure password-based authentication
- **Custom Claims**: Role-based access control via Firebase custom claims
- **Token Validation**: All API requests validate Firebase ID tokens

- **User Status Check**: Active/inactive user status validation

Implementation:

- Frontend: Firebase Auth SDK with `onAuthStateChanged` listener
- Backend: Firebase Admin SDK token verification
- Custom claims set for client users: `{ type: "client", client_access: [clientId] }`

API Security

CORS Configuration:

- **Allowed Origins**: Whitelist of allowed domains
- **Current Allowed Origins**:
  - `http://localhost:3000` (development)
  - `https://localhost:3000` (development)
  - `https://myworkreview.netlify.app` (production)
  - `https://fridayinc.netlify.app` (production)
  - `https://www.fridayinc.netlify.app` (production)
  - `https://us-central1-friday-a372b.cloudfunctions.net` (functions)
- **Configuration**: `functions/index.js` - `allowedOrigins` array
- **Note**: Add new domains to `allowedOrigins` array when deploying to new domains

Bearer Token Authentication:

- **Email Receiving Endpoint**: `/mail-receive` requires Bearer token
- **Token**: `EXPECTED_BEARER_TOKEN` (stored in `functions/index.js`)
- **Validation**: Checks `Authorization: Bearer <token>` header
- **Error Responses**: 401 (no token), 403 (invalid token)

Data Security

Firestore Security Rules:

- **User Access**: Users can only access their own data
- **Client Access**: Client users can only access their assigned clients
- **Team Access**: Team-based access control for engagements
- **Status Validation**: Only active users can access the system

Firebase Storage Security Rules:

- **Path-Based Access**: Files stored in client-specific paths
- **Signed URLs**: Time-limited signed URLs for file access
- **Upload Permissions**: Only authenticated users can upload
- **Download Permissions**: Users can only download files they have access to

Service Account Keys:

- **Location**: `functions/serviceAccountKey*.json`
- **Security**: Never commit to version control
- **Access**: Use environment variables or secure secret management
- **Rotation**: Regularly rotate service account keys

Environment Variables

Sensitive Configuration:

- **Firebase Config**: Stored in `.env` file (frontend)
- **Service Account Keys**: Stored as JSON files (backend)
- **API Keys**: Stored in Firestore `friday/settings` document
- **Email Credentials**: Stored in environment variables (backend)

Best Practices:

- Use `.env` files for local development
- Use Firebase Functions environment variables for production
- Never commit `.env` files or service account keys
- Use Firebase Secret Manager for sensitive data

Performance Optimizations

Frontend Optimizations

Code Splitting:

- **Lazy Loading**: Major routes loaded on-demand
- **React.lazy()**: Used for route-level code splitting
- **Dynamic Imports**: Components loaded as needed
- **Bundle Size**: Reduced initial bundle size

Caching Strategies:

- **Firestore Cache**: Persistent local cache (unlimited size)

- **Multi-tab Sync**: Cache synchronized across browser tabs
- **Service Worker**: Precaching of static assets
- **Cache First**: Static assets served from cache
- **Stale While Revalidate**: Dynamic content with background updates

React Optimizations:

- **useMemo**: Memoized calculations for expensive operations
- **useCallback**: Memoized callbacks to prevent re-renders
- **React.memo**: Memoized components for performance
- **Debouncing**: Debounced search and input handlers

Backend Optimizations

Concurrent Processing:

- **PQueue**: Used for concurrent processing with configurable limits
- **Concurrency Limits**: 5-10 concurrent operations per queue
- **Batch Operations**: Firestore batch writes for multiple operations
- **Queue Management**: `queue.onIdle()` to wait for completion

Database Optimizations:

- **Indexed Queries**: Firestore queries use indexed fields
- **Query Limits**: Limited queries with pagination where needed
- **Real-time Listeners**: Efficient `onSnapshot` usage
- **Unsubscribe**: Proper cleanup of listeners

Function Configuration:

- **Memory**: 512MB (standard), 2GB (high memory), 4GB (large files)
- **Timeout**: 9 minutes (540 seconds) for long-running operations
- **Cold Start**: Minimized dependencies for faster cold starts

Network Optimizations

Request Batching:

- **Batch Writes**: Multiple Firestore writes in single batch
- **Parallel Requests**: Concurrent API calls where possible
- **Request Deduplication**: Prevent duplicate requests

Asset Optimization:

- **Image Optimization**: Optimized images for web

- **Font Loading**: Optimized font loading strategies
- **Static Assets**: Served from CDN (Netlify)

Error Handling and Logging

Frontend Error Handling

Error Boundaries:

- **React Error Boundaries**: Catch and handle React errors
- **Fallback UI**: Display error messages to users
- **Error Reporting**: Log errors to console and Firestore

User Feedback:

- **Snackbar Notifications**: User-friendly error messages
- **Loading States**: Show loading indicators during operations
- **Validation Errors**: Field-level validation with error messages

Error Logging:

- **Console Logging**: Development error logging
- **Firestore Logs**: Error logs stored in `friday/logs/list`
- **Activity Logs**: User actions logged to activity collections

Backend Error Handling

Function Error Handling:

- **Try-Catch Blocks**: Comprehensive error handling in functions
- **Error Responses**: Proper HTTP status codes (400, 401, 403, 500)
- **Error Logging**: Console logging and Firestore error logs
- **Developer Notifications**: Email notifications for critical errors

Error Types:

- **Validation Errors**: 400 Bad Request
- **Authentication Errors**: 401 Unauthorized
- **Authorization Errors**: 403 Forbidden
- **Not Found Errors**: 404 Not Found
- **Server Errors**: 500 Internal Server Error

Error Notification:

- **Developer Emails**: Critical errors sent to developers
- **Function Logs**: Firebase Functions logs
- **Firestore Logs**: Error logs in `friday/logs/list` collection

Activity Logging

RFI Activity Logs:

- **Location**: `friday/engagements/list/[engagementId]/rfis/[rfiId]/activity`
- **Logged Events**: Status changes, file uploads, responses, deadline updates
- **Structure**: `{ message, user_email, dateTime }`

Engagement Activity Logs:

- **Location**: `friday/engagements/list/[engagementId]/activity`
- **Logged Events**: Field changes, RFI additions/removals, stage changes
- **Structure**: `{ message, user_email, dateTime }`

Recreation Logs:

- **Location**: `friday/recreationLogs/list`
- **Logged Events**: Engagement recreation attempts and results
- **Structure**: `{ engagementId, clientId, status, details, timestamp }`

System Logs:

- **Location**: `friday/logs/list`
- **Logged Events**: System errors, warnings, and important events
- **Structure**: `{ log_time, email, review, message }`

Backup and Recovery

Automated Backups

Daily Backup Function:

- **Trigger**: Pub/Sub topic `daily-backup`
- **Schedule**: Daily (configured via Cloud Scheduler)
- **Process**:
    1. Exports Firestore database to Cloud Storage
    2. Stores backup in `gs://fridaydailybackups` bucket

3. Creates timestamped backup files
4. Maintains backup history

Backup Storage:

- **Bucket**: `gs://fridaydailybackups`
- **Location**: Google Cloud Storage
- **Retention**: Configure retention policy as needed
- **Access**: Restricted to service account

Backup Verification:

- **Manual Verification**: Check backup bucket for recent backups
- **Automated Checks**: Monitor backup function logs
- **Restore Testing**: Periodically test restore procedures

Manual Backup Procedures

Firestore Export:

```
# Export Firestore database
gcloud firestore export gs://fridaydailybackups/export-$(date +%Y%m%d)
```

Storage Backup:

```
# Backup Firebase Storage
gsutil -m cp -r gs://friday-a372b.appspot.com gs://fridaydailybackups/stc
```

Recovery Procedures

Firestore Restore:

1. Identify backup file from backup bucket
2. Use `gcloud firestore import` command
3. Verify data integrity after restore
4. Test application functionality

Storage Restore:

1. Identify backup from backup bucket
2. Use `gsutil -m cp -r` to restore files
3. Verify file permissions and access

4. Test file access in application

Point-in-Time Recovery:

- **Limitation**: Firestore backups are point-in-time snapshots
- **Frequency**: Daily backups allow recovery to previous day
- **Data Loss**: Data created after last backup may be lost

Disaster Recovery Plan

Recovery Steps:

1. **Assess Damage**: Identify affected systems and data
2. **Stop Services**: Prevent further data loss
3. **Restore Backups**: Restore from most recent backup
4. **Verify Integrity**: Test application and data integrity
5. **Resume Services**: Gradually resume normal operations
6. **Document Incident**: Document recovery process and lessons learned

Recovery Time Objectives (RTO):

- **Target**: Restore within 24 hours
- **Backup Frequency**: Daily backups
- **Recovery Testing**: Quarterly recovery drills

Troubleshooting Guide

Authentication Issues

Problem: User cannot log in

- **Check**: User exists in `friday/users/list` collection
- **Check**: User status is "active" (not "inactive")
- **Check**: Firebase Auth user exists
- **Check**: Email matches between Firestore and Firebase Auth
- **Solution**: Verify user document and status, check Firebase Console

Problem: Client user cannot access client data

- **Check**: Custom claims set correctly (`type: "client"` , `client_access: [clientId]`)
- **Check**: User document has correct `clients` array
- **Check**: Client status is "active" in user document

- **Solution**: Verify custom claims, check user document structure

Problem: My Review integration fails

- **Check**: MWR API key in `friday/settings` document (`keys.mwr`)
- **Check**: My Review API endpoint is accessible
- **Check**: Service account key for My Review is valid
- **Check**: User is auditor (not client)
- **Solution**: Verify API key, check service account key, test API endpoint

Data Access Issues

Problem: User cannot see engagements

- **Check**: User has correct role/permissions
- **Check**: Engagement team matches user's team
- **Check**: Client user has access to engagement's client
- **Check**: Engagement status is "Active" (not "Pending")
- **Solution**: Verify user permissions, check engagement document

Problem: Client user cannot see RFIs

- **Check**: RFI has user's `cloud_id` in `users` array
- **Check**: Client user is admin OR assigned to RFI
- **Check**: Engagement is active and not closed
- **Solution**: Verify RFI assignment, check user role

Problem: Files cannot be uploaded

- **Check**: Firebase Storage bucket exists and is accessible
- **Check**: Storage security rules allow uploads
- **Check**: User has authentication token
- **Check**: File size is within limits
- **Solution**: Verify Storage bucket, check security rules, test upload

Backend Function Issues

Problem: Cloud Functions not executing

- **Check**: Functions are deployed (`firebase deploy --only functions`)
- **Check**: Function logs in Firebase Console
- **Check**: Pub/Sub topics exist and are configured
- **Check**: Cloud Scheduler jobs are enabled
- **Solution**: Check deployment status, review function logs, verify Pub/Sub topics

Problem: Daily sync not working

- **Check**: Google Apps Script endpoint is accessible
- **Check**: `USER_EMAIL_SYNC_KEY` environment variable is set
- **Check**: Pub/Sub topic `daily-user-sync` exists
- **Check**: Cloud Scheduler job is enabled
- **Solution**: Test Google Apps Script endpoint, verify environment variables

Problem: Email notifications not sending

- **Check**: SMTP credentials are correct
- **Check**: Email service is accessible
- **Check**: Nodemailer configuration is correct
- **Check**: Function logs for email errors
- **Solution**: Verify SMTP credentials, test email sending, check logs

Performance Issues

Problem: App is slow to load

- **Check**: Network connection speed
- **Check**: Firestore query performance
- **Check**: Large data sets being loaded
- **Check**: Service worker cache issues
- **Solution**: Optimize queries, implement pagination, clear cache

Problem: Functions timeout

- **Check**: Function memory allocation
- **Check**: Function timeout settings
- **Check**: Large data processing
- **Check**: External API response times
- **Solution**: Increase memory/timeout, optimize processing, add retries

Integration Issues

Problem: Google Drive integration fails

- **Check**: Service account key for Google Drive is valid
- **Check**: Google Drive API is enabled
- **Check**: Drive folder permissions
- **Check**: Rate limits not exceeded
- **Solution**: Verify service account key, check API status, review rate limits

Problem: Email receiving not working

- **Check**: Bearer token is correct (`EXPECTED_BEARER_TOKEN`)
- **Check**: Email service is sending to correct endpoint
- **Check**: CORS configuration allows email service origin
- **Check**: Function is deployed and accessible
- **Solution**: Verify Bearer token, check endpoint URL, test endpoint

Common Error Messages

"Not allowed by CORS"

- **Cause**: Request origin not in `allowedOrigins` array
- **Solution**: Add origin to `allowedOrigins` in `functions/index.js`

"Unauthorized: No Bearer token provided"

- **Cause**: Email receiving endpoint missing Bearer token
- **Solution**: Include `Authorization: Bearer <token>` header

"User not found"

- **Cause**: User document doesn't exist in Firestore
- **Solution**: Verify user exists in `friday/users/list` collection

"Engagement not found"

- **Cause**: Engagement document doesn't exist
- **Solution**: Verify engagement exists in `friday/engagements/list` collection

"Permission denied"

- **Cause**: User doesn't have required permissions
- **Solution**: Check user role and permissions, verify route protection

Debugging Tips

Frontend Debugging:

- Use React DevTools for component inspection
- Check browser console for errors
- Use Network tab to inspect API calls
- Check Firestore cache in Application tab

Backend Debugging:

- Check Firebase Functions logs in Firebase Console
- Review Firestore logs collection (`friday/logs/list`)
- Check email notifications for system errors
- Use Firebase Emulator for local testing

Database Debugging:

- Use Firebase Console to inspect Firestore data
- Check document structure and field values
- Verify indexes are created for queries
- Test queries in Firebase Console

Getting Help

Resources:

- **Firebase Console**: Check function logs and Firestore data
- **Firestore Logs**: Review `friday/logs/list` collection
- **Email Notifications**: Check developer email for system errors
- **Function Logs**: Review Firebase Functions logs in Console

Support Checklist:

1. Check Firebase Console for errors
2. Review Firestore logs collection
3. Check email notifications for system errors
4. Verify environment variables are set
5. Test API endpoints manually
6. Review function logs for detailed errors

Version Information

- **Current Version**: 1.1.863 (from package.json)
- **React Version**: 18.2.0
- **Firebase SDK**: 10.7.1
- **Node.js**: 20 (functions)

Last Updated: 7 November 2025
Maintained By: Prolific Solutions