**IDT4M Last Mile Health Interview Programming Exercise Documentation by Pacharo Simukonda**

## 1. Overview

This document provides a detailed explanation of the development process for the Android App and Web Dashboard as part of the IDT4M Last Mile Health Interview Programming Exercise. The goal was to create an Android app that interacts with a REST API to manage disease case data and a web dashboard to visualize the same data.

The following sections describe the implementation steps, tools used, and key features of both projects. Screenshots of the interfaces are included for clarity.

## 2. Android App Documentation

### 2.1 Purpose

The Android app allows users to:

- READ disease case data from the restdb.io database and display it in a readable format.
- FILTER records based on **Disease Classification**.
- CREATE new records and post them to the database.

### 2.2 Tools and Technologies

- Programming Language: Kotlin
- Development Environment: Android Studio
- API Communication: Retrofit
- UI Components: RecyclerView, EditText, Spinner, Button
- Architecture: MVVM (Model-View-ViewModel)

### 2.3 Implementation Steps

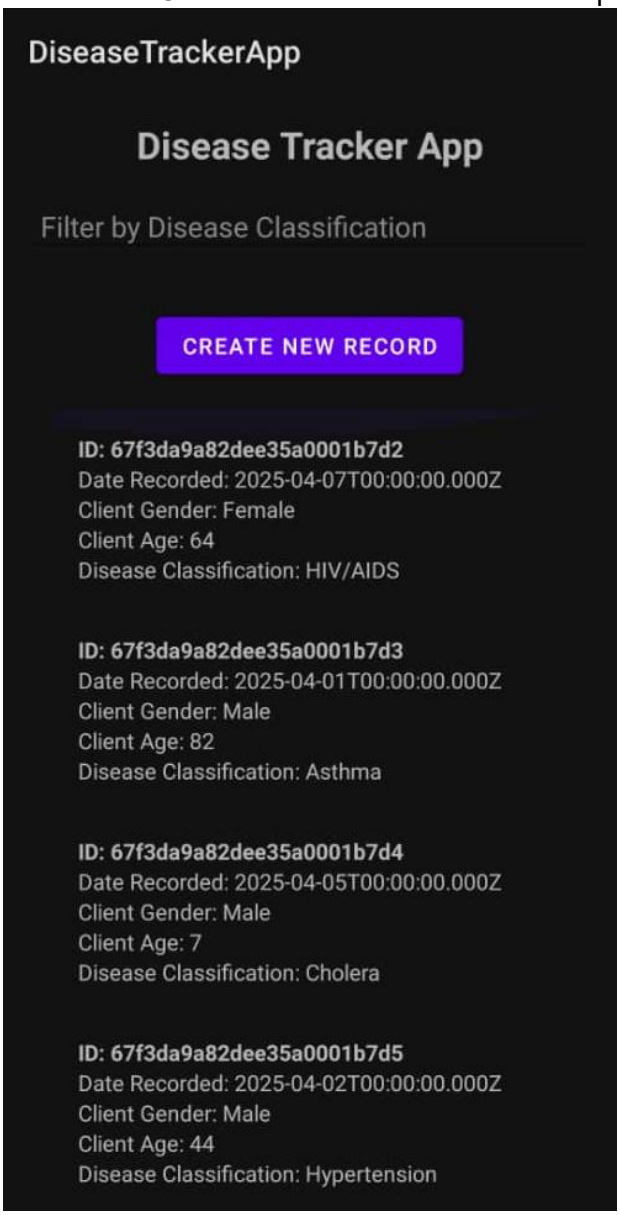
#### Step 1: Setting Up the Project

- Installed Android Studio and created a new Kotlin-based Android project.
- Added necessary dependencies in the **build.gradle** file:
    - Retrofit for API calls.
    - Gson for JSON parsing.
    - Lifecycle-aware components (ViewModel, LiveData) for managing UI-related data.

#### Step 2: Designing the User Interface

- Used a **RecyclerView** to display the list of disease cases.
- Added a filter field (**EditText**) for filtering records by **Disease Classification**.

- Included a "Create New Record" button that opens a dialog for adding new records.




*The main screen of the Android app showing the list of disease cases, filter field, and "Create New Record" button.*

**Step 3: Fetching Data**

- The DiseaseCase data class represents the structure of the database fields

```
data class DiseaseCase(
    @SerializedName("_id") val id: String? = null,
    @SerializedName("Date Recorded") val dateRecorded: String = getCurrentDate(),
    @SerializedName("Client Gender") val clientGender: String? = null,
    @SerializedName("Client Age") val clientAge: Int? = null,
    @SerializedName("Disease Classification") val diseaseClassification: String? = null
) {
```

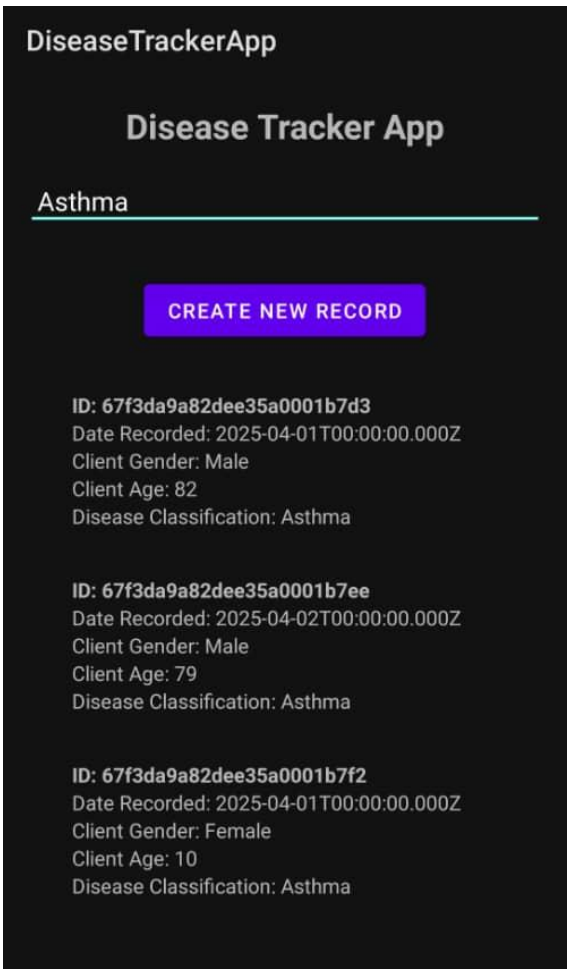
- Configured Retrofit to fetch data from the restdb.io API using a **GET** request.

```
@GET("rest/cases-disease")
suspend fun getCases(@Header("x-apikey") apiKey: String): Response<List<DiseaseCase>>
```

- Displayed the fetched data in the **RecyclerView** using a custom adapter (**DiseaseCaseAdapter**).

**Step 4: Filtering Records**

- Implemented filtering functionality using a **TextWatcher** to dynamically update the **RecyclerView** based on user input.

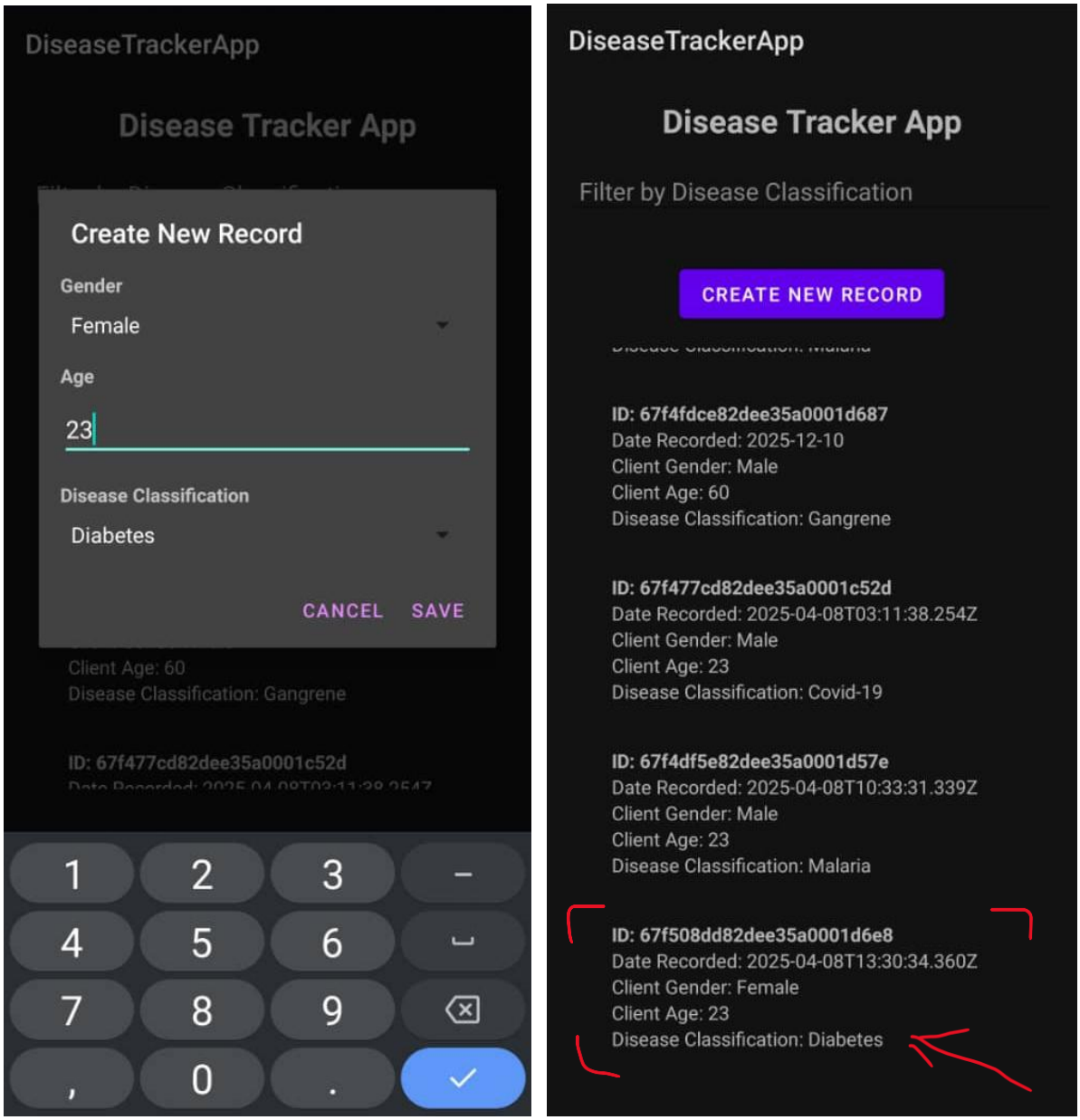



The filtered list of disease cases after entering a query in the filter field.

**Step 5: Creating New Records**

- Created a dialog with input fields for **Client Gender**, **Client Age**, and **Disease Classification**.
- Automatically generated the **Date Recorded** field using **SimpleDateFormat**.
- Validated user input before sending data to the API using a **POST** request.

```
@POST("rest/cases-disease")
suspend fun createCase(
    @Header("x-apikey") apiKey: String,
    @Body case: DiseaseCase
): Response<DiseaseCase>
```




*The dialog for creating a new record, showing input fields for gender, age, and disease classification.*

**3. Web Dashboard Documentation**

**3.1 Purpose**

The web dashboard visualizes disease case data disaggregated by:

- **Client Gender**
- **Disease Classification**

It uses charts and tables to provide meaningful insights into the data.

**3.2 Tools and Technologies**

- Frontend Framework: HTML, Tailwind CSS
- Charting Library: Chart.js
- API Communication: Fetch API

- Styling: Tailwind CSS for responsive design

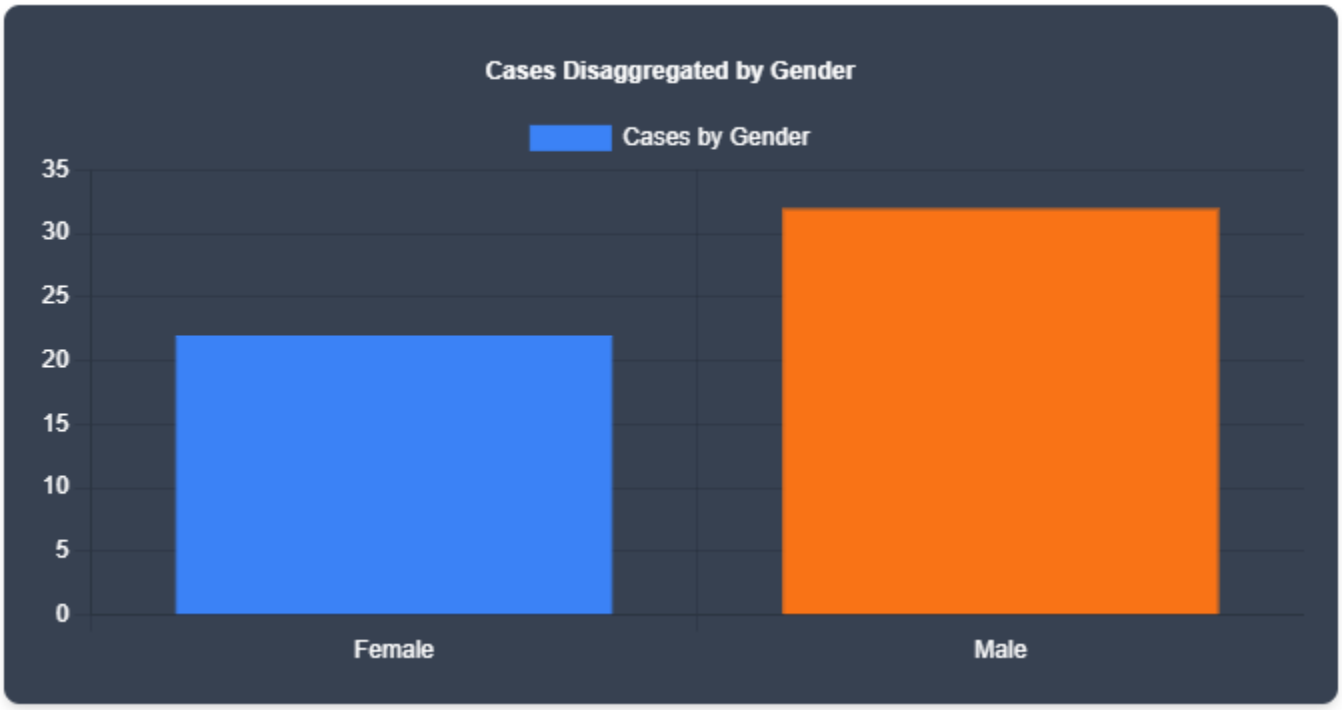
### 3.3 Implementation Steps

#### Step 1: Setting Up the Project

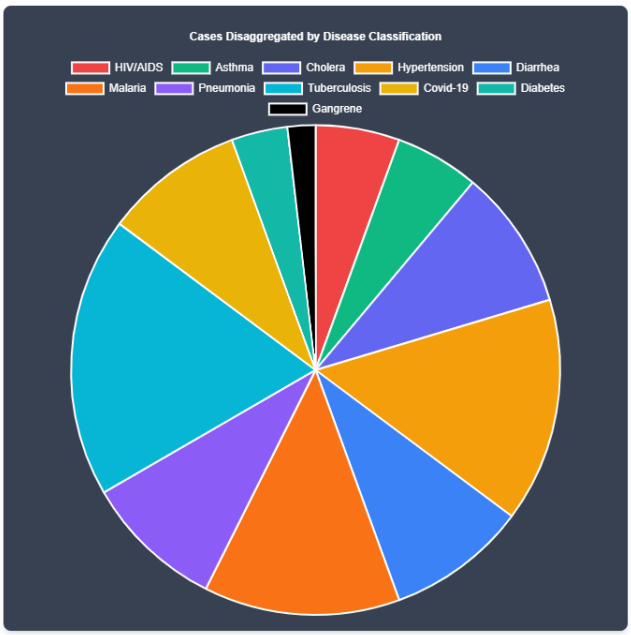
- Created a folder structure with **index.html**, **script.js**, and optional **style.css**.
- Included libraries like Tailwind CSS and Chart.js via CDN links in the **<head>** section.

#### Step 2: Designing the User Interface

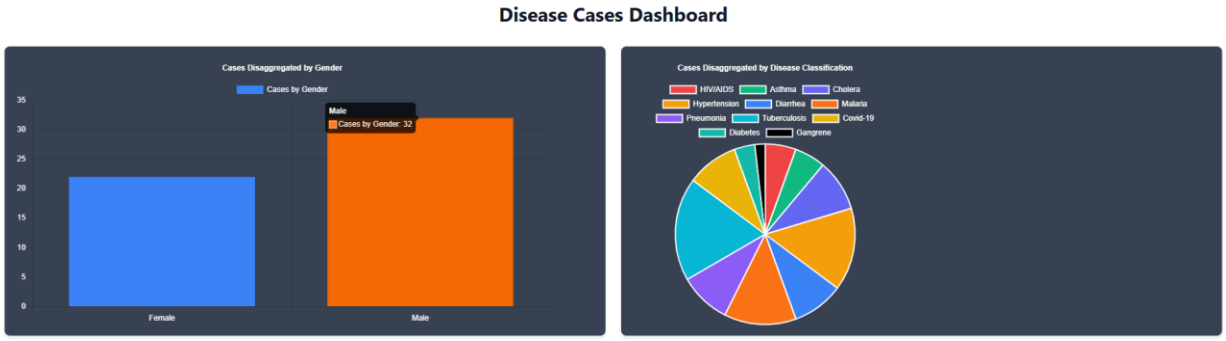
- Used a responsive grid layout to display two charts side-by-side:
    - A bar chart for cases disaggregated by gender.




    - A pie chart for cases disaggregated by disease classification.



*The web dashboard layout showing two charts side-by-side.*




### Step 3: Fetching Data

- Used the Fetch API to retrieve data from the restdb.io database.

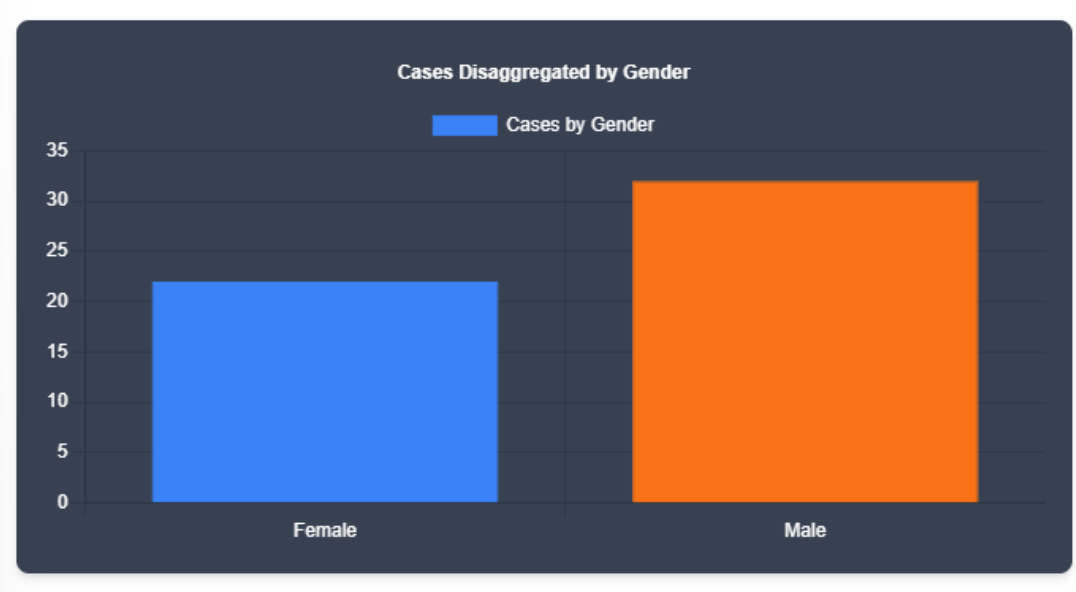
```
const fetchData = async () :Promise<void> => {  Show usages  Pacharo Simukonda
    try {
        const response :Response = await fetch( input: 'https://incldigitrans-5881.restdb.io/rest/cases-disease', init: {
            method: 'GET',
            headers: {
                'Content-Type': 'application/json',
                'x-apikey': '67f3e466636df6b1f15d955a',
                'cache-control': 'no-cache'
            }
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        processData(data);
    } catch (error) {
        console.error('Error fetching data:', error);
        document.getElementById( elementId: 'app').innerHTML = `<p class="text-red-500 text-center">Failed to load data.</p>`;
    }
};
```

- Processed the data to group it by **Client Gender** and **Disease Classification**.

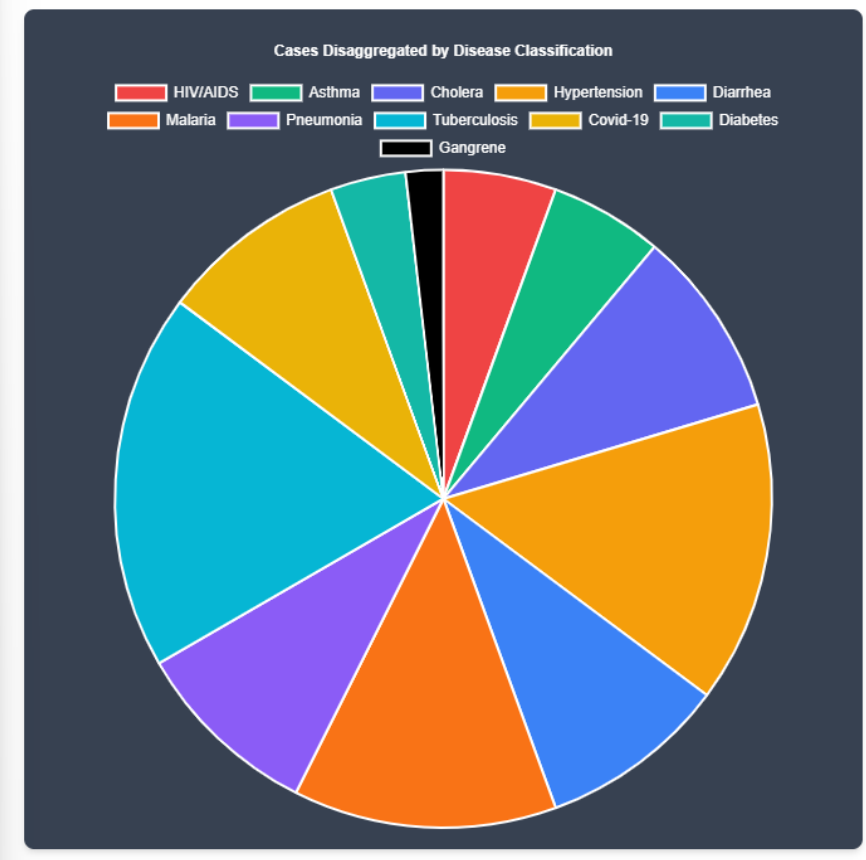
```
const processData = (data) :void => {  Show usages  Pacharo Simukonda
    if (!Array.isArray(data) || data.length === 0) {
        console.error('No data available or invalid data format:', data);
        document.getElementById( elementId: 'app').innerHTML = `<p class="text-yellow-500 text-center">No data available to display.</p>`;
        return;
    }

    const genderCounts :{} = {};
    const diseaseCounts :{} = {};

    data.forEach(record => {
        genderCounts[record['Client Gender']] = (genderCounts[record['Client Gender']] || 0) + 1;

        diseaseCounts[record['Disease Classification']] = (diseaseCounts[record['Disease Classification']] || 0) + 1;
    });

    renderCharts(genderCounts, diseaseCounts);
};
```

**Step 4: Rendering Charts**

- Used Chart.js to render:
    - A bar chart for cases disaggregated by gender.
    - A pie chart for cases disaggregated by disease classification.
- Customized the charts with colors, labels, and legends for better readability.




*Bar chart showing cases disaggregated by gender.*

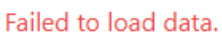



*Pie chart showing cases disaggregated by disease classification.*

**4. Testing and Debugging**

- Tested both the Android app and web dashboard thoroughly:
    - Verified that data was fetched and displayed correctly.
    - Ensured filters and forms worked as expected.
    - Checked visualizations for accuracy and responsiveness.

4.1Web Dashboard




Error message displayed when the app/dashboard fails to load data.

**5. Installation and Usage**

**5.1 Android App**

1. Clone the repository: git clone https://github.com/huntsolo-dev/IDT4M-LMH-Programming-Exercise.git
2. Open the project in Android Studio.
3. Sync Gradle dependencies and run the app on an emulator or physical device.

**5.2 Web Dashboard**

1. Clone the repository: git clone https://github.com/huntsolo-dev/IDT4M-LMH-Programming-Exercise.git
2. Open the **index.html** file in a browser to view the dashboard.

**6. Challenges Faced and Solutions**

- API Integration:
    - *Challenge:* Understanding how to authenticate and interact with the REST API, including the use of headers like **x-apikey** for secure access and ensuring correct endpoint configuration.
    - *Solution:* Researched the restdb.io documentation and Swagger API specification to understand the required headers, endpoints, and request/response formats. Implemented authentication by including the **x-apikey** header in all API requests. For Android, used Retrofit to streamline API calls and handle JSON parsing, while for the web dashboard, utilized the Fetch API to perform GET and POST requests. Added robust error handling to manage scenarios like invalid API keys, network issues, or empty responses, ensuring a smooth user experience.
- Charts Implementation:
    - *Challenge*: Creating interactive and visually appealing charts for the web dashboard.
    - *Solution:* Used Chart.js for bar and pie charts, customizing colors, labels, and responsiveness for better user experience.
- Learning Curve of Kotlin:
    - *Challenge:* Adapting to Kotlin syntax and Android-specific concepts like MVVM architecture, LiveData, and coroutines.
    - *Solution:* Followed tutorials, articles, documentation, and hands-on practice to grasp Kotlin fundamentals and Android development best practices.

- RecyclerView Setup:
    - *Challenge:* Binding data to RecyclerView efficiently in the Android app.
    - *Solution:* Created a custom adapter and ViewHolder to manage data binding and improve performance.
- Error Handling:
    - *Challenge:* Providing meaningful feedback to users when API calls fail or return empty data.
    - *Solution:* Implemented error messages in the web dashboard to inform users about issues like network errors or invalid responses.
- Dynamic Filtering:
    - *Challenge:* Implementing real-time filtering of records based on user input.
    - *Solution:* Used **TextWatcher** in Android and JavaScript event listeners in the web dashboard to dynamically filter data.
- Date Handling:
    - *Challenge:* Auto-generating and formatting dates for new records in the required format.
    - *Solution:* Used **SimpleDateFormat** in Android and JavaScript's **Date** object to generate and validate timestamps.
- Responsive Design:
    - *Challenge:* Ensuring the web dashboard looked good on all screen sizes.
    - *Solution:* Used Tailwind CSS for responsive layouts and tested the dashboard on desktop and mobile devices.
- Version Control:
    - *Challenge:* Managing code changes and collaborating effectively.
    - *Solution:* Used Git and GitHub for version control, creating branches for features and submitting pull requests for code reviews.
- Testing and Debugging:
    - *Challenge:* Identifying and fixing bugs during development.
    - *Solution:* Used Android Studio's Logcat for debugging and browser developer tools (I used Google Chrome) for the web dashboard, along with thorough testing of all features.

**7. Conclusion**

The Android app and web dashboard successfully meet the requirements of the IDT4M Last Mile Health Interview Programming Exercise. Both projects demonstrate a structured approach to problem-solving, effective use of modern technologies, and attention to user experience.

**7.1 Key Features Achieved**:

- Android app: Displays, filters, and creates disease case records.
- Web dashboard: Visualizes data using interactive charts.