Maintenance guide:

Working environments:

Android Studio:

For this app project **Android Studio** was used as a development environment. Android Studio is the official integrated development environment (IDE) for developing Android applications. It's built on JetBrains' IntelliJ IDEA software and provides all the tools developers need to design, develop, and debug Android apps.

Firebase:

Firebase is a comprehensive platform developed by Google for building and managing mobile and web applications. It offers a wide array of tools and services that simplify the backend development process, allowing developers to focus on creating the user-facing parts of their apps. Firebase is especially popular for Android development, but it supports iOS and web platforms as well. In Firebase, there are two main types of databases designed for app development: **Realtime Database** and **Cloud Firestore**. Both are NoSQL databases intended for use with mobile and web applications.

Our database consists:

Users:

1. username
2. Password
3. Id
4. Phon number
5. Type
6. Age of child
7. City
8. Email

Chat - rooms:

1. Message
2. Date
3. Time
4. Sender email

Community:

1. Name
2. Disabilities
3. Hobbies
4. Nodes - users
5. Pets

Event:

1. Id
2. Date

3. Description
4. Location
5. Time
6. Time stamp
7. Title
8. The user that create the event and his user id

Offer help:

1. Id
2. Date
3. Description
4. Location
5. Time
6. Time stamp
7. Title
8. The user that create the offer and his user id

Request help:

1. Id
2. Date
3. Description
4. Location
5. Time
6. Time stamp
7. Title
8. The user that create the *request* and his user id

Requirements for environments:

1. Internet connection
2. For vs code need to install the following packages:
   Prettier
   Dart
3. Android studio
4. Firebase

Maintenance:

1. Adding more functionalities
2. Upgrading the display
3. Upgrading the database services based on a mount of the users, events and messages, it stores, so it continue to work in its best performance
4. Improvement and refinement of the community-building algorithm dynamically

Impotent code part :

There are 3 important parts of this project : louvain.dart, louvain_graph:

Louvain:

This part uses the graph we created and calculates the distances and thus matches the users to the communities in a modular way

import 'louvain_graph.dart';

```
class LouvainAlgorithm {
  // Function to calculate modularity based on the current partition of the graph
  double calculateModularity(Graph graph, List<Community> communities) {
    int totalEdges = graph.getTotalEdges(); // Total number of edges in the graph
    double modularity = 0.0;

    // Loop through each community and calculate modularity for each pair of nodes
    for (var community in communities) {
      for (var node1 in community.members) {
        for (var node2 in community.members) {
          bool areConnected = graph.areConnected(node1, node2); // Check if nodes are
connected
          int degree1 = graph.getDegree(node1); // Degree (number of edges) of node1
          int degree2 = graph.getDegree(node2); // Degree of node2

          // Modularity formula: (1 if connected, otherwise 0) minus expected connection
          modularity += (areConnected ? 1 : 0) - (degree1 * degree2) / (2 * totalEdges);
        }
      }
    }

    // Normalize modularity by total edges in the graph
    return modularity / (2 * totalEdges);
  }

  // Function that runs the Louvain algorithm to detect communities
  List<Community> run(Graph graph) {
    // Initialize communities where each node is its own community
    final communities = <Community>[];
    graph.getAdjacencyList().forEach((node, neighbors) {
      communities.add(Community(members: [node], interests: []));
```

```
});

bool improvement = true; // Flag to track if improvement in modularity is found
double currentModularity = calculateModularity(graph, communities); // Initial modularity

// Loop until no improvement in modularity is found
while (improvement) {
 improvement = false;

 // Loop over each community and each node in the community
 for (var community in communities) {
  for (var node in community.members) {
   var bestCommunity = community; // Track the best community for the node
   var bestModularityGain = 0.0;  // Track the best modularity improvement

   // Check all neighboring communities and calculate modularity gain
   for (var neighborCommunity in getNeighboringCommunities(node, communities,
graph)) {
    community.members.remove(node); // Temporarily remove node from current
community
    neighborCommunity.members.add(node); // Add node to neighboring community

    // Recalculate modularity after moving the node
    double newModularity = calculateModularity(graph, communities);

    // If modularity improves, keep track of the best community for the node
    double modularityGain = newModularity - currentModularity;
    if (modularityGain > bestModularityGain) {
     bestModularityGain = modularityGain;
     bestCommunity = neighborCommunity;
     improvement = true;
    }
```

```
          // Revert the node back to the original community if modularity did not improve
          neighborCommunity.members.remove(node);
          community.members.add(node);
        }

        // If a better community was found, move the node to that community
        if (bestCommunity != community) {
          community.members.remove(node);
          bestCommunity.members.add(node);
          currentModularity += bestModularityGain; // Update the current modularity
        }
      }
    }
  }

  return communities; // Return the detected communities
 }

 // Helper function to get neighboring communities for a given node
 List<Community> getNeighboringCommunities(String node, List<Community> communities,
Graph graph) {
  // Find neighboring communities based on the connections of the node in the graph
  Map<String, int> neighbors = graph.getNeighbors(node);
  return communities.where((community) => community.members.any((member) =>
neighbors.containsKey(member))).toList();
 }
}

class Community {
 List<String> members;  // List of members (nodes) in the community
 List<String> interests; // Optional: list of interests related to the community
```

```
  Community({
    required this.members,
    required this.interests,
  });
}
```

Louvain Graph:

This part builds the graph - adding arcs and nodes to the graph structure by using a Adjacency list with weights

```
class Graph {
  final Map<String, Map<String, int>> adjacencyList = {}; // Adjacency list with weights

  // Function to add a node to the graph
  void addNode(String nodeId) {
    if (!adjacencyList.containsKey(nodeId)) {
      adjacencyList[nodeId] = {};
    }
  }

  // Function to add an edge between two nodes in the graph with a weight
  void addEdge(String nodeId1, String nodeId2, int weight) {
    if (adjacencyList.containsKey(nodeId1) && adjacencyList.containsKey(nodeId2)) {
      adjacencyList[nodeId1]?[nodeId2] = weight;  // Add the weight between node1 and node2
      adjacencyList[nodeId2]?[nodeId1] = weight;  // Ensure the graph is bidirectional
    }
  }

  // Function to get the adjacency list of the graph
  Map<String, Map<String, int>> getAdjacencyList() {
    return adjacencyList;
  }
```

```
  // Function to get the total number of edges in the graph
  int getTotalEdges() {
    int edges = 0;
    adjacencyList.forEach((node, neighbors) {
      edges += neighbors.length;
    });
    return edges ~/ 2; // Divide by 2 because each edge is counted twice
  }

  // Function to get the neighbors of a specific node
  Map<String, int> getNeighbors(String node) {
    return adjacencyList[node] ?? {};
  }

  // Function to get the degree (number of connections) of a specific node
  int getDegree(String node) {
    return adjacencyList[node]?.length ?? 0;
  }

  // Function to check if two nodes are connected
  bool areConnected(String node1, String node2) {
    return adjacencyList[node1]?.containsKey(node2) ?? false;
  }
}
```