deti

ICM

Flutter project

Heart Serious Game

Trabalho realizado por:

Diogo Falcão (Nº 108712) e José Gameiro (Nº108840) – LEI

Índex

Introduction

In the class of ICM (Introduction to Mobile Computing), we're asked to work on a first project, which consists in developing a fully functional mobile application, using the framework flutter. The framework Flutter is a versatile and dynamic framework developed by Google, that emerged as a game-changer in the universe of mobile app development. Offering a robust set of tools and an intuitive programming environment, Flutter empowers developers to create stunning cross-platform applications with ease.

Project Context

For this project, under the guidance of our instructor, our group conceived the idea of creating the Heart Serious Game (HS Game). In this innovative concept, two players engage in a challenge facilitated by WearOS smartwatches equipped with heart rate monitoring capabilities. The smartwatches record and transmit the participants' heart rate values to a mobile device, where they are visualized in real-time. As the game progresses, the mobile application stores these values in a local database. Ultimately, victory is claimed by calculating the average heart rate, and choosing which player scored the highest average throughout the gameplay period. This unique approach not only integrates cutting-edge technology but also adds an immersive and competitive element to the gaming experience.

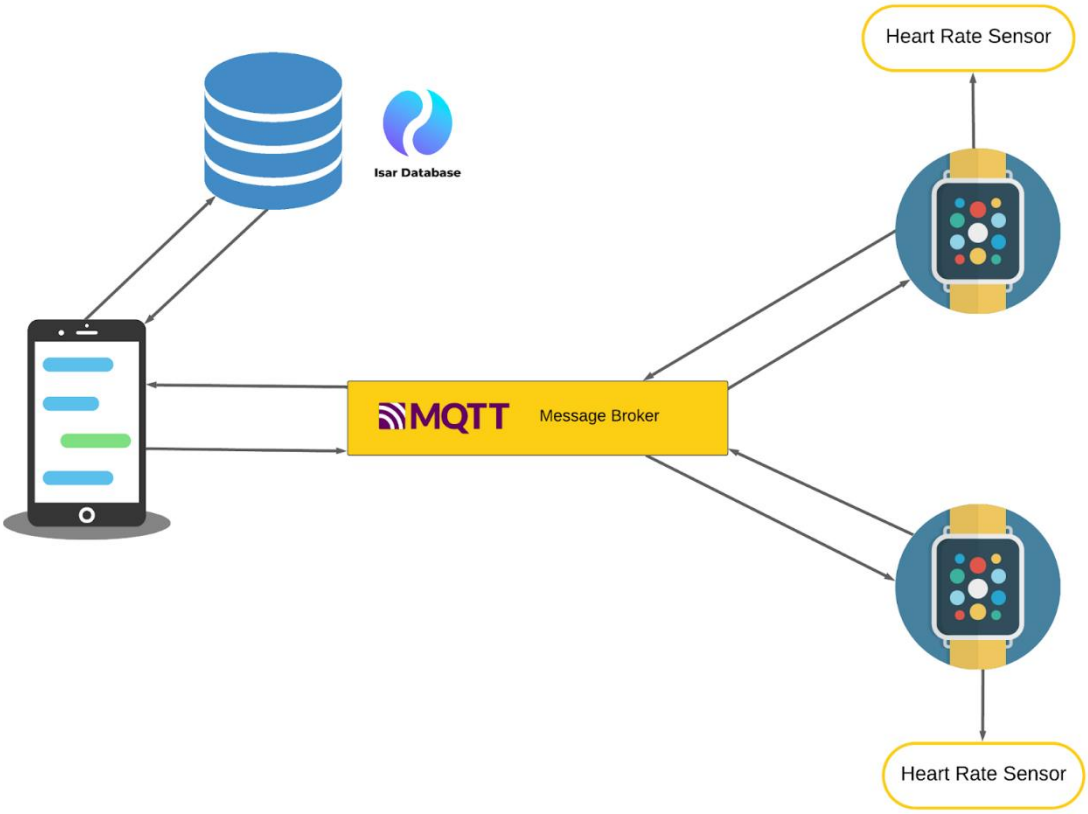
Requirements

Our group defined the following requirements for us to make our system fully functional:

- WearOS Integration:
  - The mobile application must be compatible with WearOS devices.
  - Ensure seamless communication and integration between the WearOS smartwatches and the mobile application.
- Heart Rate Sensor:
  - The mobile application must interface with the heart rate sensors embedded in the WearOS smartwatches.
  - Ensure accurate and real-time retrieval of heart rate data from the sensors.
  - Give permission to use health data retrieval and the app is good to go!
- Message Broker:
  - Implement a message broker system to facilitate communication between the two players' smartwatches and the mobile device.
  - Ensure reliable message delivery and synchronization between the devices.
  - Use MQTT or similar protocols for efficient message routing and delivery.
- Device interaction and Explicit communication:

    - Enable explicit communication and interaction between the mobile device and the paired smartwatches.
    - Establish a connection protocol to link the mobile application with the WearOS devices securely.
    - Implement device discovery and pairing mechanisms to connect the two smartwatches with the mobile device.
- Display real time feedback of the game:
    - Develop a user interface that provides real-time feedback of the ongoing game, including the beginning of the game and the connection status of other devices in every user screen (Smartwatch or Smartphone);
    - Display the heart rate data collected from the smartwatches in a visually engaging manner.
    - Implement dynamic visualizations or animations to reflect changes in heart rate during gameplay.

Architecture Diagram and Workflow



The architecture of our system comprises five essential elements: two smartwatches (with the capability of reading the heart rate), a Message Queuing Telemetry Transport (MQTT) broker, one mobile device, and a database powered by Isar. Each element plays a crucial role in facilitating seamless communication, data exchange, and storage within our application ecosystem.

We developed two distinct Flutter projects to accommodate the varying mechanisms and interfaces of the mobile device and smartwatch. This delineation was necessitated by the fundamental differences between the two devices; for instance, while the smartwatch is responsible for reading heart rate values, the mobile phone solely receives them.

The workflow embedded within our architecture unfolds as follows:

- Device Connection and configuration
    - Initially, each device connects to the MQTT broker deployed in the network.
    - Upon connection establishment, the devices send configuration messages to the broker, containing pertinent information such as their unique identifiers (IDs) and device types (phone or smartwatch).
    - The MQTT broker efficiently routes these configuration messages to the appropriate destinations.
    - Simultaneously, the mobile device captures and stores this configuration data in the Isar database, ensuring a comprehensive record of all connected devices.
- Heart Rate measure and transmission:
    - Once all three devices (two smartwatches and one mobile device) are successfully connected and configured, the heart rate measurement process begins.
    - The smartwatches initiate heart rate monitoring and periodically transmit the measured data to the MQTT broker.
    - Leveraging the lightweight and efficient nature of MQTT protocol, the broker swiftly relays the heart rate data to the designated recipients, including the mobile device.
    - Upon receiving the heart rate data, the mobile device processes and stores it in the Isar database, associating each data entry with the corresponding smartwatch ID.
- Data Storage and visualization
    - The Isar database, residing within the mobile device, serves as the repository for all heart rate data collected during the gameplay.
    - Leveraging the capabilities of Isar, the mobile application efficiently manages and organizes the heart rate data, ensuring optimal storage performance and data integrity.
    - The mobile application retrieves the stored heart rate data from the Isar database, presenting it to the user in a visually intuitive and informative manner.
    - Through dynamic visualizations and real-time feedback, users can monitor their heart rate fluctuations and track their performance during gameplay.

MQTT was selected as the messaging protocol due to its lightweight nature and efficient message routing capabilities. It’s publish-subscribe architecture facilitates seamless communication between multiple devices, ensuring fast and reliable data transmission. MQTT's support for Quality of Service (QoS) levels enables us to prioritize message delivery based on the application's requirements, enhancing overall reliability and robustness.

Isar was chosen as the database solution for its speed, reliability, and simplicity. Its reactive design and support for object-oriented database operations align well with the needs of a mobile application with real-time data requirements. Isar's performance optimizations, such as lazy loading and background indexing, ensure efficient data storage and retrieval, even on resource-constrained mobile devices. Additionally, Isar's compatibility with Flutter and its seamless integration with Dart programming language make it an ideal choice for our mobile application development needs.

Widget tree

Since we've created two distinct flutter projects (one for a mobile device and another for a smartwatch) we'll divide this section in 2, analyzing the widget tree of the mobile project and the smartwatch project separately.

Mobile Project (phone_main folder)

Main Widget (main.dart)

This widget defines the main entry point of the Flutter application and constructs the basic structure of the user interface; it represents the root widget of the application. It configures the material design properties, such as theme and title, and specifies the initial screen to be displayed using the MaterialApp widget.

Within the MaterialApp widget, there is a ChangeNotifierProvider wrapping the MQTTAppState instance. This provider is essential for managing the state of the MQTT connection throughout the application, ensuring that changes in the MQTT state are propagated to dependent widgets.

MQTT App State (mqttappstate.dart)

This class is responsible for managing the state related to MQTT connection and receiving messages within the Flutter application. It utilizes the ChangeNotifier pattern to notify listeners of any state changes, ensuring that the UI reflects the most up-to-date information. It serves as a central repository for managing MQTT-related state and facilitating communication between different components of the Flutter application.

MQTT Manager (mqttmanager.dart)

Responsible for managing MQTT communication within the application. It interacts with the mqtt_client package to establish connections, send and receive messages, and handle various MQTT events. The class includes several instance variables:

- currentState: Represents the current state of the MQTT connection, managed by an instance of MQTTAppState.

- isarService: Manages interactions with the Isar database, including saving user data and heart rate information.
- client: Represents the MQTT client used for communication with the MQTT broker.
- identifier, host, and topic: Store essential connection parameters such as client identifier, host address, and topic to subscribe/publish messages.
- users: Maintains a list of users and their associated heart rate data.

The class includes methods to initialize the MQTT client, establish and terminate connections, publish messages, and handle various MQTT events such as connection status changes, message reception, and disconnection. Key methods include:

initializeMQTTClient: Configures the MQTT client with connection settings and callbacks.

- connect: Initiates a connection to the MQTT broker.
- disconnect: Terminates the MQTT connection and removes device information.
- publish: Publishes a message to the MQTT broker.
- onConnected: Handles successful connection events, subscribes to the MQTT topic, and processes received messages.
- onDisconnected: Handles disconnection events and updates the connection state.

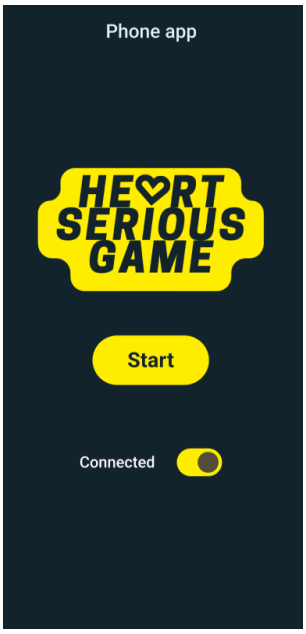
Overall, MQTTManager encapsulates the logic for managing MQTT communication, ensuring seamless interaction between the application and the MQTT broker.

MQTT View (mqttview.dart)

Serves as a child of the main widget and encapsulates elements related to MQTT communication and device connection status. The widget includes several instance variables and methods:

- _hostTextController, _messageTextController, _topicTextController: Controllers for text fields used to input MQTT connection parameters.
- currentAppState: Holds the current state of the MQTT connection.
- manager: Manages MQTT communication through the MQTTManager class.
- isarService: Facilitates interactions with the Isar database.
- The build method constructs the UI components within a SingleChildScrollView and a Column. It includes an app bar, connection state text, and the main content area.
- The mainColumn method renders the main content of the widget, displaying an image, connection status messages, and a connection button based on the MQTT connection state.
- The _buildConnecteButtonFrom method dynamically creates the connection button based on the MQTT connection state. It also displays information about connected devices and controls for starting and stopping the MQTT connection.
- The _prepareStateMessageFrom method generates human-readable connection status messages based on the MQTT connection state.
- The _configureAndConnect and _disconnect methods handle configuring and connecting/disconnecting from the MQTT broker, respectively.

Overall, MQTTView provides a visual representation of MQTT connection status and facilitates user interaction with MQTT communication within the Flutter application.

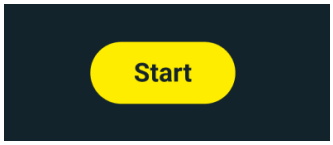


mqttappsatet.dart

This code defines a Dart class named MQTTAppState along with an enum called MQTTAppConnectionState. The class extends ChangeNotifier from the flutter/cupertino.dart package, enabling it to notify listeners about changes in its state. The class encapsulates the state management for an MQTT (Message Queuing Telemetry Transport) application. It includes attributes to store the connection state, received text, message history, and connected devices. Additionally, it provides methods to update these attributes and query information about the connected devices.

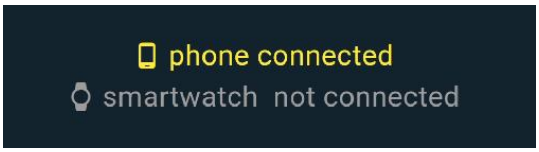
Yellow Button (yellowbutton.dart)

This widget, named yellowButton, is a simple function that constructs a yellow-colored button with custom styling and text. It encapsulates the creation of an ElevatedButton widget with specific properties and behavior. We created a function in a different file to avoid repetition, because it's used in the Mqtt View widget and in the End Page widget (which will be discussed later in this report).



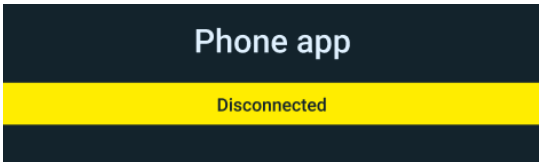
Connected Device (deviceconnected.dart)

This widget is a simple function that builds a row containing an icon and text representing the connection status of a device. It encapsulates the creation of a row layout with an icon and text, providing a visual indication of whether the device is connected or not. Like the Yellow Button widget, this one is only used in the Mqtt View.



App Bar (appbar.dart)

This widget, named appBar, is also a simple function that constructs the app bar for the Flutter application. It encapsulates the creation of an AppBar widget with specific properties and styling.



Connection State Text (connectionstate.dart)

This code defines a function called `connectionStateText` in Flutter, which is used to display a text status related to connection states.

Countdown pages (countdown.dart)

The `CountdownWidget` is a Flutter StatefulWidget displaying a countdown timer. It initializes with attributes for a database service and a build context. Upon initialization, it starts the countdown and triggers vibration feedback. The countdown logic is managed through a periodic timer. Once the countdown reaches 1, a custom vibration pattern is initiated, and the widget navigates to a new page. The UI includes a colored container, an image corresponding to the countdown value, and text displaying the current countdown value.

game

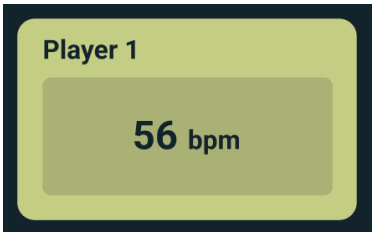
This code defines a StatelessWidget named GamePage. The GamePage widget contains attributes for the database service (isarService) and the build context (context). In the build method, the widget constructs the UI for the game page. It includes a container with a specific background color, containing a SingleChildScrollView widget. Inside the SingleChildScrollView, it displays widgets generated by the appBar and columnGameState functions.

columngame.dart

The code defines a Flutter widget function called `columnGameState`. This function creates a vertical scrollable column of UI elements, including a countdown timer, player information fetched from a database service (`isarService`), and a button to navigate back to the first page of the app. The player information is displayed dynamically using the `PlayerWidget` based on the data fetched from the service.

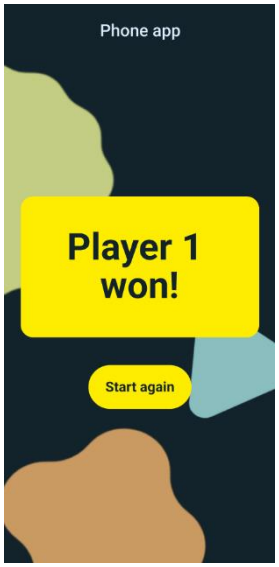
playerwidget.dart

This code defines a Flutter widget called PlayerWidget. It is a stateful widget responsible for displaying player information, including their name and heart rate. The widget receives parameters for colors (brightColor and darkColor), player name, and heart rate. In its state, it retrieves the MQTT app state using the Provider package. In its build method, it constructs the UI with a colored container and displays the player's name and heart rate using appropriate styling.



winner.dart

The code defines a StatelessWidget called GamePage, which encompasses attributes for a database service (isarService) and a build context (context). Within its build method, the widget constructs the user interface for the game page, consisting of a container with a defined background color enclosing a SingleChildScrollView widget. The SingleChildScrollView displays widgets generated by the appBar and columnGameState functions.



usar.dart

This Dart code defines a class called User, annotated with @collection from the Isar package, indicating that instances of this class can be stored in a database collection. The class contains fields for an optional ID, a nullable name, and a nullable list of double values representing heart rates. Additionally, it includes a constructor to initialize these fields. The part directive imports the generated code file (user.g.dart) containing necessary functions for database operations.

userservice.dart

This Dart code defines an IsarService class responsible for interacting with an Isar database. It initializes a late Future variable to hold the Isar instance and provides methods for various database operations such as saving, retrieving, updating, and deleting users. Additionally, it includes methods for obtaining user-related data like heart rate lists and usernames. The openDB method initializes the Isar database instance by opening it with the necessary schemas and directory path, while ensuring only one instance exists. Each database operation is performed within a write transaction for data consistency and integrity.

This service implements:

- saveUser(User newUser): Saves a new user to the Isar database.
- listenUser(): Listens to changes in the user collection and yields a stream of user lists.

- getUserById(int id): Retrieves a user by ID from the Isar database.
- getIdByName(String name): Retrieves the ID of a user by name from the Isar database.
- getHeartRateList(User user): Retrieves the list of heart rates of a user.
- getHeartRateListById(int id): Retrieves the list of heart rates of a user by ID.
- getLastHeartRate(int id): Retrieves the last heart rate of a user by ID.
- getUserName(int id): Retrieves the name of a user by ID.
- addHeartRate(User user, double heartRate): Adds a heart rate to a user's heart rate list.
- getAllUser(): Retrieves all users from the Isar database.
- UpdateUser(User user): Updates an existing user in the Isar database.
- cleanAllUser(): Deletes all users from the Isar database.
- deleteUser(int userid): Deletes a user from the Isar database based on user ID.

SmartWatch Project (wear_mqtt folder)

main.dart

The code initializes a Flutter Material app with a theme, state management via Provider, and a custom MQTT view wrapped in scrolling constraints.

heartbeats.dart

The `GamePage` class constructs a Flutter widget representing a game page with a customized app bar, heart rate measurement indicator, player name display, and a yellow button for stopping, all styled with specified colors and fonts.

endpage.dart

The `EndPage` class, a StatefulWidget, renders an end screen in a Flutter application, featuring a background image, an animated blur effect controlled by a timer, a custom app bar titled "WearOS app," and a yellow button labeled "Start again" for restarting the game. The page's appearance is styled with specified colors and fonts, providing a cohesive and visually appealing user interface for the end phase of the game.

mqttmanager.dart

The `MQTTManager` class facilitates listening and publishing to a specific topic using the MQTT protocol in a Flutter application. It initializes an MQTT client with the provided host and identifier, manages connection and disconnection, and handles message publishing and subscription. The class utilizes the `mqtt_client` package for MQTT functionality and `logger` package for logging. It also interacts with the `MQTTAppState` to manage the application's connection state and device information, dynamically updating device status based on received messages and ensuring a maximum device limit. Additionally, it implements callbacks for various MQTT events such as successful connection, subscription confirmation, and unsolicited disconnection, providing a robust and responsive MQTT communication layer within the application.

mqtt_appstate.dart

Like the one in the phone_main folder.

deviceconnected.dart

Like the one in the phone_main folder.

yellowbutton.dart

Like the one in the phone_main folder.

mqtt_view.dart

The `MQTTView` class is a StatefulWidget in a Flutter application, responsible for displaying a view for MQTT connection management and interaction. The class initializes controllers for input fields, manages the MQTT connection state, and handles connection and disconnection events. It also provides methods for configuring and connecting to the MQTT broker, starting heartbeat measurement, and disconnecting from the broker. The view includes UI elements such as an app bar, connection state text, editable fields for MQTT parameters, device connection status indicators, a button for starting the heartbeat, and a switch for toggling the connection state. Additionally, it utilizes the `provider` package for state management, `logger` package for logging, and `workout` package for heartbeat measurement. The UI elements are styled using specified colors and fonts, providing a coherent and user-friendly MQTT interaction interface within the application.


WearOS app
Disconnected
Connection is non-existent!

Dependencies in WearOS app:

1. bounce
2. flutter_animate
3. flutter_lints
4. flutter_svg
5. loading_animation_widget
6. logger
7. mqtt_client
8. provider
9. uuid
10. workout

Dependencies in Phone app:

1. animated_background
2. bounce
3. isar
4. isar_flutter_libs
5. isar_generator
6. loading_animation_widget
7. logger
8. motion
9. mqtt_client
10. path_provider
11. provider
12. uuid
13. vibration

Conclusion and final note

Wrapping up our project in the Introduction to Mobile Computing (ICM) class, where we dove into building a mobile app with Flutter, was quite the experience. Flutter, by Google, turned out to be a real game-changer for us. It's user-friendly and packed with features that let us make an app that not only works across platforms but also looks and feels great. We put a lot of effort into the small things - like choosing the right icons and making sure the app felt alive with smooth animations and feedback when you touch something. This wasn't just about getting a good grade; we really got into it, paying close attention to the user experience. Learning Flutter was a big part of the project. It's safe to say we learned a ton, not just about coding, but also about design MQTT broker and Isar, and how to make an app feel just right.