Love Letter

Distributed Systems

Emanuele Lamagna

January 2025

Contents

Chapter 1

Introduction

1.1 Original game

Love Letter[1] is a famous card game introduced in 2012 by Seiji Kanai. In this game each player aims to deliver a love letter to the Princess with the assistance of relatives and acquaintances.

At the beginning of each game each player draws a card. The first player draws a second card from the deck and decides which of the two to play. Each card has a different effect, which can be protection against other players or

an attack. Here is the complete list:

- 1, Guard: "Player may choose another player and name a card other than the Guard. If the chosen player's hand contains the same card as named, that player is eliminated from the round."
- 2, Priest: "Player may privately see another player's hand."
- 3, Baron: "Player may choose another player and privately compare hands. The player with the lower-value card is eliminated from the round."
- 4, Handmaid: "Player cannot be affected by any other player's cards until their next turn."
- 5, Prince: "Player may choose any player (including themselves) to discard their hand and draw a new one."
- 6, King: "Player may trade hands with another player."
- 7, Countess: "Does nothing when played, but if the player has this card and either the King or the Prince in their hand, this card must be played immediately."
- 8, Princess: "If the player plays or discards this card for any reason, they are eliminated from the round."

When the card is played, it is placed in front of the player so that everyone can see it. The player then ends the round and the next one begins. The goal of the game is to get to the last turn (when there are no more cards to draw from the deck) with the card with the highest number. However, there are other conditions of defeat or victory:

- a player is eliminated if, for any reason, he plays or discards the princess or if he is a victim of the baron's effect
- a player wins (early) if he is the only one left alive

1.2 App version

The goal of the mobile app is to replicate the gaming experience offered by the physical game, allowing you to create rooms of players (from 1 to 4) from which to start the games. The player who creates the room is called 'host'

and can start the game when there are at least two players connected. When the game starts, the players begin to alternate in the various turns, and at the end they return to the initial room. Obviously, compared to the physical game, it is necessary to carefully manage the user connections, since it is a real-time game.

Chapter 2

Requirements

The project must satisfy various requirements, divided into functional and non-functional.

2.1 Functional requirements

- Possibility of creating a room by any player.
- Possibility of entry into a room by any player using the room code.
- Possibility for host player to start game.
- Possibility for each player to leave the room or game.
- Correct management of turns, taking into account various dynamics including eliminated players.
- Correct functioning of card effects. This also includes the ability to choose a player as a target.
- Need to have in real time the list of cards in the player's hand and the list of cards played by other players.
- Correct handling of player disconnection. When a player is offline, he must receive a warning. Other players, in the same way, must understand (at a graphical level) that he is not online at that moment. It is therefore necessary to create a mechanism for sending heartbeats to verify in real time whether a player is online or not (other players must already see the player offline within about 3-4 seconds).

2.2 Non-functional requirements

- Optimization of performance related to the composition of screens and widgets. This involves limiting screen rebuilds, in favor of rebuilds of individual widgets (for example, when a player draws, the entire screen is not rebuilt, but only the graphic part that shows the cards in the player's hand).
- Fault tolerance and reliability: the system should perform its intended functions without errors or failures, and should continue to function properly even if such failures do occur.
- Concurrency control and data consistency: the system must ensure that there are no race conditions and that the data always remains consistent and valid.
- High scalability: the system should be, in an optimistic view, very scalable.
- Simple and responsive UI: a player must be able to immediately understand various aspects including the cards he has, the player whose turn it is and how many cards remain in the deck.

2.3 Technology choices

As for the app development, it has been decided to use **Flutter**[2] for many reasons:

- the framework that represents the state of the art of mobile applications
- previous experience of use and, generally, ease of use
- possibility of cross-platform execution

As for the management of the application states, it has been decided to use **BLoC**[3], a package that has become very popular in recent years. BLoC makes it really easy to manage widget rebuilds, making it easy to meet the aforementioned requirement.

As for the backend part, it was decided to use **Supabase**[4]. Supabase is an open-source platform that provides a ready-to-use backend infrastructure for web and mobile applications. It is represents an open-source alternative to Firebase. Supabase simplifies development by offering tools and services

that cover various backend aspects, such as a PostreSQL database, Realtime and Storage. It also has the ability to self-host.

In particular, Supabase was chosen for its **Realtime**[5] feature. Realtime enables real-time updates to the application by using PostgreSQL's logical replication. It allows clients to subscribe to changes (INSERT, UPDATE, DELETE) in the database and receive live updates without needing to refresh or poll the server. The key features are:

- Live Data Sync: automatically synchronizes data changes between the database and connected clients.
- Channels: allows subscribing to specific database tables or rows for granular updates.
- WebSocket-Based: uses WebSockets for low-latency, efficient real-time communication.
- Flexible Integration: can be used with frontends built in React, Flutter, Vue, or other frameworks.

Using Realtime we can count on excellent management of many of the established requirements:

- Fault Tolerance and Reliability. Supabase employs several strategies to ensure reliability and fault tolerance in its services, especially for real-time capabilities. In terms of reliability, Supabase operates a globally distributed real-time server infrastructure that is designed to be highly available. This distribution reduces latency and ensures that services remain operational even in the event of localized failures. Supabase also supports cross-region data replication through its "Publications" feature. This approach mitigates the risk of data loss or downtime, as if one region goes down, data can be recovered from another. In terms of fault tolerance, Supabase's architecture is designed to avoid single points of failure, ensuring that the system can continue to function even if one component fails. This helps maintain service uptime in the event of hardware or software issues [6].
- Concurrency and Data Consistency. For concurrency, PostgreSQL uses a transaction isolation model based on MVCC (Multiversion Concurrency Control), which allows multiple transactions to read and write to the database at the same time, without interfering with each other: each transaction sees a "snapshot" of the data at the time it started,

avoiding locks on readers [7]. Conflicting writes are resolved through locking mechanisms. Regarding data consistency, PostgreSQL ensures that each operation or group of operations respects the ACID properties (Atomicity, Consistency, Isolation, Durability). Only committed changes are sent to clients (Logical Decoding) [8]. In addition, real-time notifications sent to clients respect the logical order of changes made in the database, preventing inconsistencies in the received data.

- High Scalability. Supabase's high scalability is primarily supported by the PostgreSQL architecture, a highly scalable and very used database.

It is necessary to use the package **supabase_flutter**[9] in the Flutter app to achieve this integration.

Chapter 3

Design

The goal of the application is very clear: to be intuitive, simple and allow players to enjoy an excellent gaming experience.

3.1 App architecture

The application architecture is called Pine Architecture[10]. This definition, introduced by Angelo Cassano in 2022, is based on the Clean Architecture[11] concept that has been around for years, but adapting it to the morphology of Flutter applications.

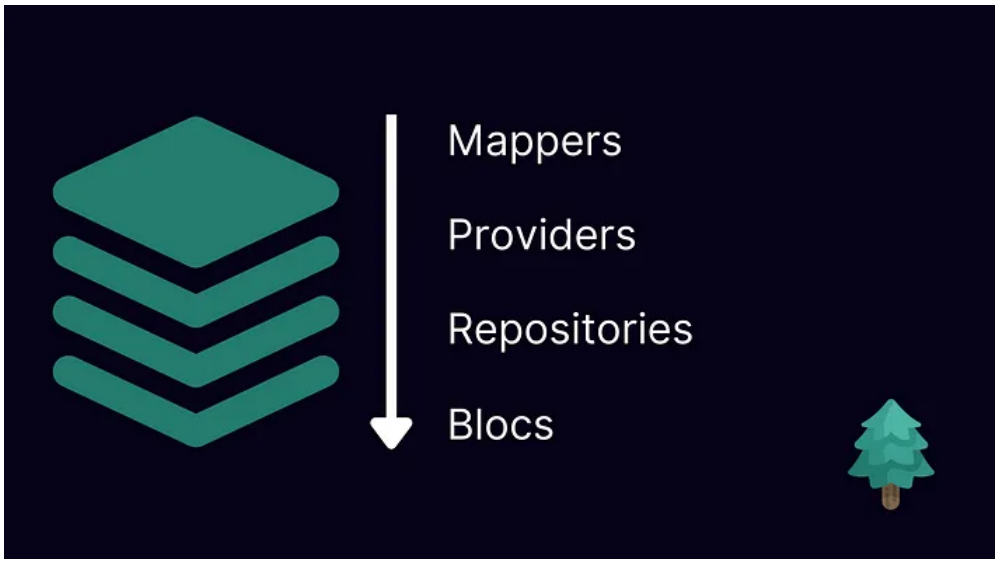
The idea is to build a Widget tree (hence the name Pine), and all dependencies are injected in a cascade. This tree contains, from top to bottom:

- mappers, to map original data (e.g. from APIs) into data to be presented (the models)
- providers, to retrieve data inside the app
- repositories, which act as an abstract layer towards the data layer
- blocs, which are the actual state manager of the app, and also act as viewmodels

A widget can call a function of a bloc, or use:

- a builder, to automatically manage rebuilds depending on the states of the bloc they refer to
- a listener, to manage what to do when the bloc is in a certain state and/or with certain data

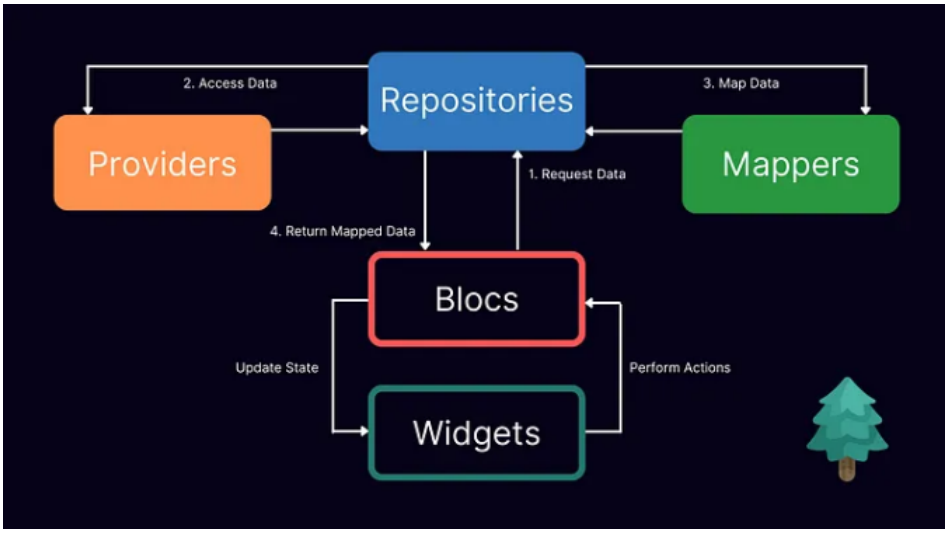
- a consumer, which allows you to do both of the above



When a bloc is called, it emits a series of states. These states are custom, and generally the following are adopted:

- init: the initial state of the bloc
- try: it is emitted when a function is called in the bloc and allows the widgets, in the meantime, to show a loading
- result: it is emitted when the data is ready (for example, the repository has returned it). The widget can then show it.
- empty: it is emitted when the data is ready, but is empty. The widget must handle this case.
- error: it is emitted if something goes wrong. The widget must handle and show the error.

Inside the blocs will be injected the repository to refer to, so when it has to make a call it queries that repository. In the repository, in turn, will be injected the services to call (for example a database helper or an API service).



3.2 App behavior

It was decided to design a series of screens:

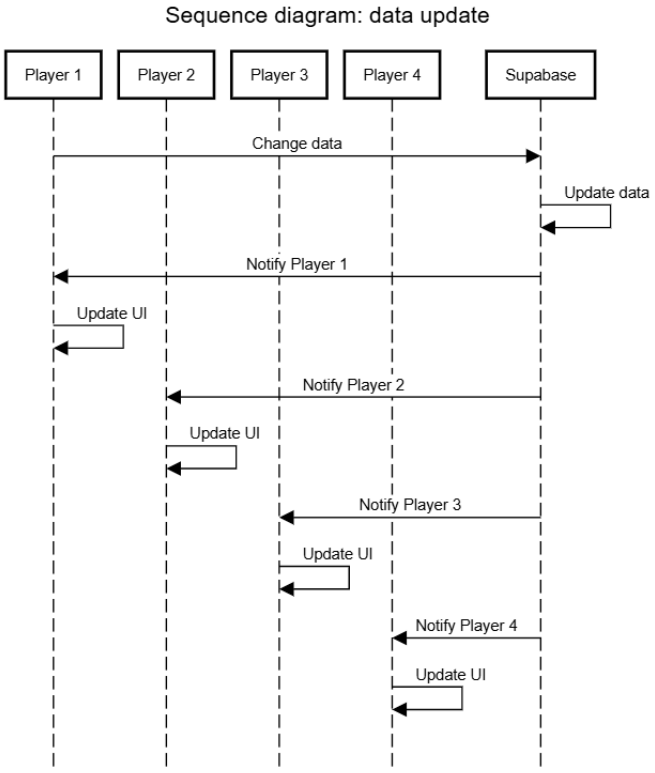
- **Home Screen**. In this screen the player can decide whether to create a room or join an existing room (this is managed via two different buttons). In both cases a dialog will appear asking for the player's name and, if the intended action is to join a room, also the code of the latter.

- **Room Screen**. This screen is accessed via the Home Screen and, as we will see later, via the Game Screen at the end of the game. In this screen each player can see the list of players in the room, the room code and a button to, if necessary, leave the room and return to the Home Screen. The host (the person who created the room) will also see a button that allows them to start the game. This button will be unlocked only when there are 2 to 4 active players (so not temporarily offline). The player can understand if another player is offline because a loading gif will appear next to the player in question. If the offline player is himself, then he will see a full-screen loading gif that will warn him that he is offline.

- **Game Screen**. This screen is only accessible from the Room Screen. The screen is divided into three sections: the game bar at the top, which shows the turn number, the cards remaining in the deck, and the

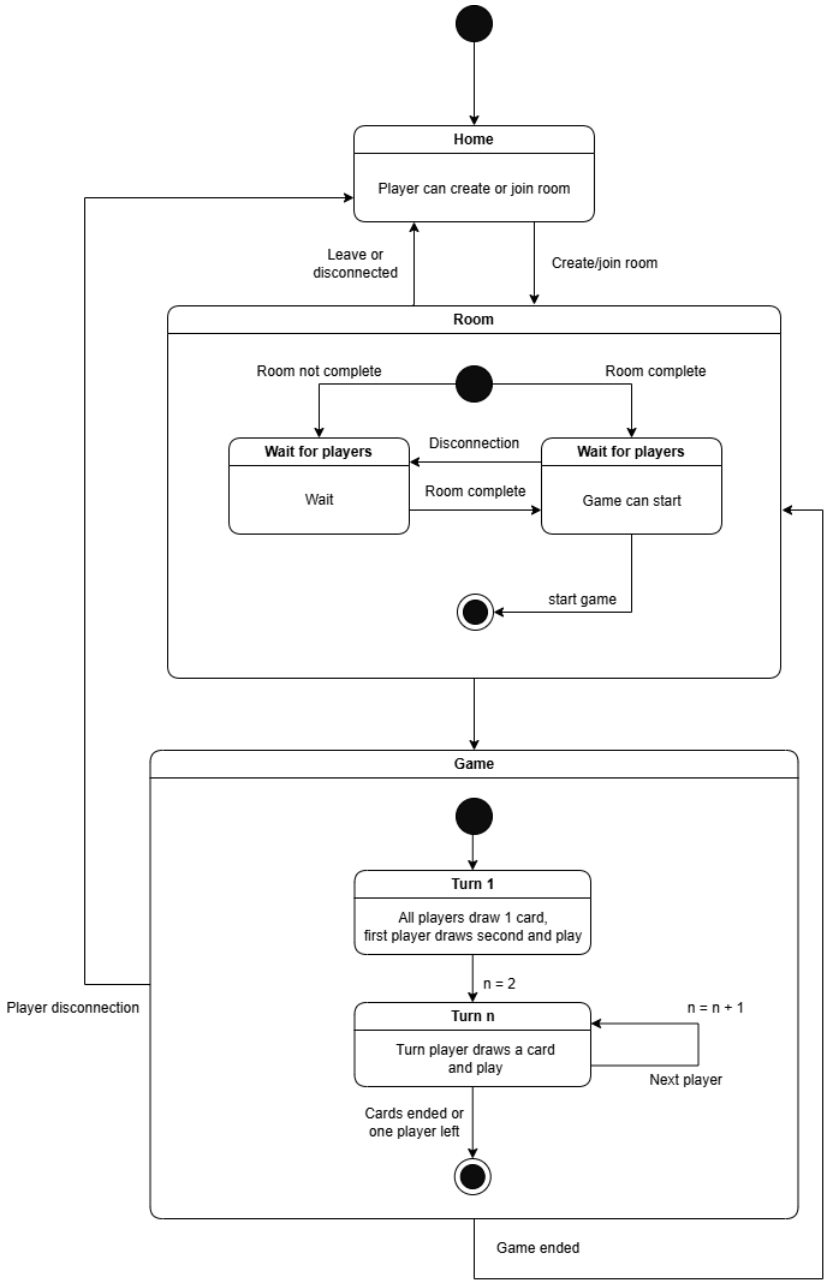
button to draw a card if the player needs to do so; the player boards section in the center, which shows the cards played by the players; the player's hand at the bottom, which shows the cards currently in the player's hand and which, if necessary, he can play. When a player, for any reason, is eliminated, his slot in the board list is darkened and the cards he had in his hand are discarded. Also in this screen, if a player loses connection, a loading gif will be shown (if it is another player) or a full-screen loading (if it is the player himself). If a player is offline for more than n seconds, they are eliminated and, graphically, it is treated as a normal game elimination. When a game ends and there are one or more winners, a dialog appears. Once the confirm is pressed, the player returns to the Room Screen, where a new game can start.

3.3 Interactions



Above is shown the sequence diagram for the general interactions between

the app and the backend. From the scheme it is easy to understand that the data can be updated by any player (for example when he draws a card, when he activates an effect and so on). Then on the backend side the database is updated, and this triggers notifications to all listeners of the modified table. Below, instead, the application state diagram is shown:



3.4 Database

After establishing what the app should do, it was essential to study the generic structure of the database. After several attempts, it was decided to adopt the following tables:

- **game**: this table contains data related to a game
- **player**: this table contains data about players within the game
- **player_status**: this table contains information about the status of players

Why separate the player table and the player_status table and not merge them into a single table? The tables are all realtime, so when they are modified, updates are sent to the listeners. Since each instance of the application is designed to send heartbeats every second, if there was only one table we would have continuous rebuilds throughout the game screen, and for each player. By keeping them separate, we can manage these very frequent rebuilds only for a small portion of the screen, namely the one related to the possible loading.

Chapter 4

Implementation details

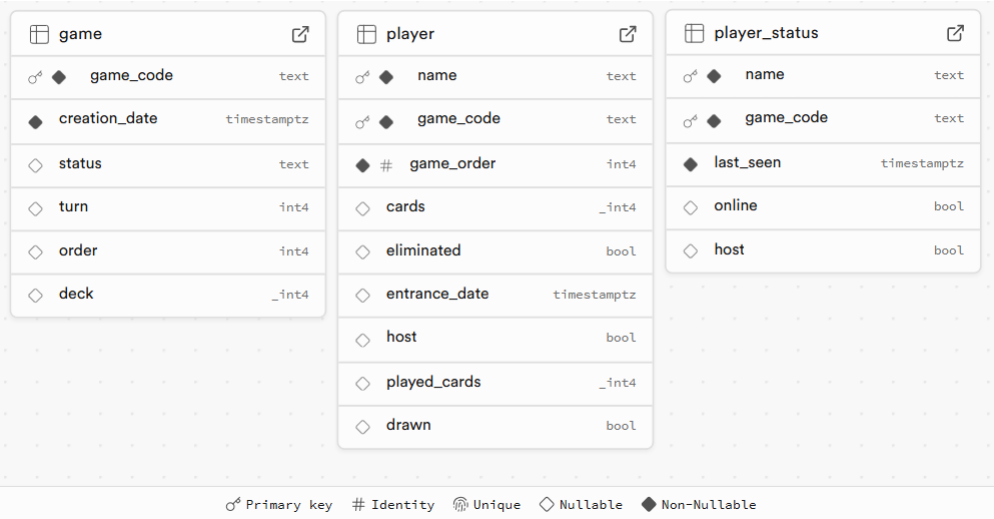
We can divide this section into two parts: Supabase Cloud and Flutter App.

4.1 Supabase Cloud

This section is relatively simple, but contains everything that is needed to best manage data synchronization between the various instances of the app.

4.1.1 Tables

As previously mentioned, the PostgreSQL database only has the following three tables:



All tables are realtime, so any change notifies all listeners. Obviously, a listener, as we will see in the next section, can listen to changes in a table only under certain conditions (for example, if the game_code field is that of the game in which it is inserted).

4.1.2 Functions

There are three database functions:

- **add_player_with_name**: it is responsible for inserting players into the room. This process is performed in this function with the aim of avoiding race conditions. In fact, otherwise, they could occur in various situations, for example:
  - initial room entrance: in this case the following error could occur: two players check if there is still room in the room; both are given a positive response, since at that moment there are only 3 out of 4 people available; after checking, both are added to the room, thus having 5 people, one more than the allowed limit. By putting the two operations (check and insert) in a single method we make the process atomic, avoiding this type of error.
  - post-game room return: In this case the following error may occur: the game is over, players must confirm the end and return to the room screen. Two players press the button 'together'. As in the previous case, both players take control, but this time on the host (the first player who returns to the room screen after a game becomes the host). The error of having two hosts in a game may then occur. Again, the atomic function solves the problem.
- **check_turn_player**: it is the function that takes care of assigning the turn to the correct player. It is not easy to determine this, since it is necessary to take into account, in addition to the order of the players depending on the turn, also the elimination of one or more players. To simplify the matter, in addition to the integer variable 'turn', the integer variable 'order' is also used, which initially starts equal to 'turn', but then, if the turn passes to an eliminated player, it is increased without touching 'turn'. This mechanism allows the regular flow of turns, accompanied by the succession of players managed by 'order'
- **handle_game_start**: it is the function responsible for initializing the game. A trigger is placed on this function: in fact, when the 'sta-

tus' parameter of the game table changes (this happens when the host presses the start game button), this function is triggered.

- **update_player_status**: This is the function that represents the heartbeat of a player. When a player calls this function, the last seen date is updated, and it is also checked to see if other players have been offline for more than n seconds. If so, they are taken offline. Players who have been offline for more than m seconds are eventually deleted, and thus removed from the player table.
  It would be more correct to run a job that handles everything, and it was initially developed this way. However, the limits of the free plan of Supabase do not allow this behavior in the long run, so it was decided to handle it this way.

4.2 Flutter App

As mentioned above, the app is based on the Pine architecture. For more details, see the "Design" chapter. It was decided to create, inside the app, a bloc for each table of the remote database (so game, player and player_status). More specifically, cubits were used: this is a lightweight version of the classic blocs[12]. Here is the list of cubits:

- **GameCubit**: handles interactions with the game table. Contains functions for:
  - check the existance of a room
  - create a room
  - start a game
  - draw from the deck
  - update the turn and/or the order
  - reset the game
  - update the game
- **PlayerCubit**: handles interactions with the player table. Contains functions for:
  - watch the hand of a player
  - swap the cards of two players
  - play/discard a card from a player hand

  - get infos about one on more players
  - insert a player
  - reset one or more players
  - eliminate a player
  - update a player

- **StatusCubit**: handles interactions with the player_status table. Contains functions for:
  - send the heartbeat of a player
  - insert a player status
  - reset a player status
  - update a player status

All functions can be called from the UI to perform certain operations. The update operations (updateGame, updatePlayer and updateStatus) are the only ones that emit states. What does this mean? Let's look at a concrete example.

```
  Future<void> updateGame(
    List<Map<String, dynamic>> data,
    String gameCode,
  ) async {
  emit(const TryGameState());
  for (var item in data) {
    if (item['game_code'] == gameCode) {
      final game = Game(
        gameCode: gameCode,
        creationDate: DateTime.parse(item["creation_date"]),
        status: item["status"],
        turn: item["turn"],
        order: item["order"],
        deck: (item["deck"] as List<dynamic>)
          .map((e) => e as int)
          .toList(),
      );
      emit(ResultGameState(game: game));
    }
  }
}
```

This is the updateGame function, which is used, as the name suggests, to update a game. During its course it emits two states: 'try', when it starts, and 'result', when it has the data ready. On the UI side there are listeners on the remote tables that, when they detect changes, call these functions. Thanks to the emission of states it is possible, in a simple and organized way, to show the various loading steps: for example, when the 'try' state is emitted it is possible to show a loading with spinner or percentage, while in the 'result' state the final data will be shown.

In the screens, then, we have listeners. The listener below, for example, is present in the GameScreen, and listens for changes in the game table only for rows with a certain game code.

```
_gameSubscription = supabase
    .from('game')
    .stream(primaryKey: ["game_code"])
    .eq("game_code", widget.gameCode)
    .listen((List<Map<String, dynamic>> data) async {
      await context.read<GameCubit>()
        .updateGame(data, widget.gameCode);
    });
```

In addition, in the RoomScreen and GameScreen we have timers for sending heartbeats:

```
Timer.periodic(
  const Duration(seconds: 1),
  (timer) => updateLastSeen(),
);
```

updateLastSeen is the function that takes care of calling the function in the StatusCubit and managing, in case of connection loss, the return to the HomeScreen.

Chapter 5

Validation

The application was developed in Flutter, so by its nature it can run on multiple platforms. Tests were carried out on the following platforms:

- Android (various versions of the SDK)
- Microsoft Edge
- Google Chrome
- Windows

No problems related to real-time data synchronization were found on any of the tested platforms. The only problems encountered were graphical ones, in particular related to screen size and widget adaptation.

In Flutter apps there are various types of tests: unit tests, widget tests, integration tests. It was not possible to test this type of application through unit tests because in Flutter, by default, you cannot make API or database calls during their execution. Running mock tests would not have made sense, since the heart of the project is the synchronization with Supabase. It was therefore decided to run an integration test. This is a test in which the application is launched on a device (physical or virtual) and in which interactions with widgets are controlled via code (for example, pressing a button, writing values in a textfield, etc.). It was decided to test the creation of a room by a player. Other tests would have had to include very complex systems, since there is a synchronization of real-time players and the possible disconnection. Further more complex tests were beyond the time and scope of the project, but they will be discussed in the last section, relating to possible future developments.

Of course, numerous manual tests were also carried out in various ways:

- through multiple physical devices
- through multiple emulators
- mix of them

It was deemed necessary to have the application installed by several users outside of the development, to test and evaluate the gaming experience.

Chapter 6

Deployment

To test the application there are two ways:

- connection to Supabase Cloud
- use of Supabase Self-Hosting

In the first case, just skip to point 5. In the second case, however, there is a series of steps to follow:

1. Install and run Supabase[13]:

   ```
   # Get the code
   git clone --depth 1 https://github.com/supabase/supabase

   # Go to the docker folder
   cd supabase/docker

   # Copy the fake env vars
   cp .env.example .env

   # Pull the latest images
   docker compose pull

   # Start the services (in detached mode)
   docker compose up -d
   ```

   Nota: in windows il comando 'cp' è sostituito dal comando 'copy'.

2. Run the given 'format.py' script in the folder that contains the new 'supabase' folder. Supabase is an open source and community driven

platform, so some errors may occur in the latest versions released. The error in question is related to the pooler: running the script changes the related file to make sure everything works correctly.

3. Access Supabase Studio. You can access Supabase Studio through the API gateway on port 8000. For example:

   `http://<your-ip>:8000`

   or

   `localhost:8000`

   if you are running Docker locally. You will be prompted for a username and password. By default, the credentials (that you can also find in .env file) are:

   ```
   Username: supabase
   Password: this_password_is_insecure_and_should_be_updated
   ```

   You should change these credentials as soon as possible. More infos in the docs[13].

4. Import tables and functions. In Supabase Studio, go to 'SQL Editor' in the left panel and copy the given SQL file content. Run it and the necessary tables and functions will be created. After running the SQL code, the last thing to do is to set the newly created tables as 'realtime': this is a process that cannot, at present, be automated. It is therefore necessary to go to the left in the menu, select 'Table Editor' and click on each of the three tables. At the top right a button will appear with the words 'Realtime off': by pressing the button the table will become realtime. This step is also possible by entering the table edit (via the three dots next to the table name).

5. Change the supabaseUrl and the ANON_KEY in the credentials file ("assets/files/credentials.txt"). For the supabaseUrl you have to use

   `http://<your-ip>:8000`

   For the ANON_KEY you have to use the one you have in the .env file.

6. Launch the app. You can launch via VS Code or Android Studio. It's also possible to create a release version apk, and then normally install in physical devices or emulators. The command is:

   ```
   flutter build apk --release
   ```

   After generating the apk, you just need to run it on an Android device and install the application. It can then be launched like any other application.

Chapter 7

Conclusion

This project allowed me to delve into many aspects of distributed systems, both at a strictly practical level (such as heartbeat/disconnection management and distributed concurrency) and at an applied theoretical level (how Supabase automatically handles the major challenges of distributed computing).

7.1 Future developments

There are some developments that, due to the amount of hours required by the project, have remained incomplete or to be done. Much of the time was spent designing the database, learning about realtime capabilities, and integrating those capabilities into a Flutter app with that kind of architecture. Some of these are:

- **Integration Test**. As already discussed in the 'Validation' chapter, some integration tests were performed to evaluate the correct entry into the room. However, performing in-depth tests of this type would have required an excessive amount of hours, so it was decided to report a basic example.
- **Row Level Security**. This is a feature that allows to define data access rules at the level of a single row in a table. In practice, RLS allows to determine who can read, write, update, or delete certain rows based on specific conditions, such as the authenticated user or column values. It is managed through SQL rules defined on the database. For example, it is possible to configure a rule to ensure that a user can only access records that he or she created (using a column such as user_id). This is a very interesting feature, which makes the application much

more secure. However, adding this feature to the tables would have implied the use of a registration and a login in the application, to ensure that the calls were authenticated. In addition to this, it would have been necessary to significantly change the organization of the updates by the users, to comply with this feature.

In short, the application offers many possibilities for future development, both in terms of graphics and synchronization rules. However, I consider the work done satisfactory, which allowed me to learn a lot and bring a concrete example of real-time implementation and integration between Flutter and Supabase.

Bibliography

[1] Love Letter, https://boardgamegeek.com/boardgame/129622/love-letter

[2] Flutter, https://docs.flutter.dev/

[3] BLoC, https://bloclibrary.dev/

[4] Supabase, https://supabase.com/docs

[5] Supabase Realtime, https://supabase.com/docs/guides/realtime

[6] Ably: Pusher vs Supabase Realtime, https://ably.com/compare/pusher-vs-supabase

[7] MVCC in PostgreSQL, https://postgrespro.com/blog/pgsql/5967899

[8] Understanding ACID Compliance in PostgreSQL, https://www.timescale.com/learn/understanding-acid-compliance

[9] Supabase Flutter package, https://pub.dev/packages/supabase_flutter

[10] Pine: A lightweight architecture helper for your Flutter Projects, https://angeloavv.medium.com/pine-a-lightweight-architecture-helper-for-your-flutter-projects-1ce69ac63f74

[11] Complete Guide to Clean Architecture, https://www.geeksforgeeks.org/complete-guide-to-clean-architecture/

[12] Cubit State Management in Flutter, https://medium.com/@muhammadnaqeeb/cubit-state-management-in-flutter-cb3d357fd0f0

[13] Supabase Self-Hosting with Docker, https://supabase.com/docs/guides/self-hosting/docker

[14] Row Level Security, https://supabase.com/docs/guides/database/postgres/row-level-security