



TABLE OF CONTENTS

01

THE GAME

A brief introduction to the project and the game

02

TECHNOLOGIES

Overview of technologies used in the project

03

IMPLEMENTATION

Details of the app and backend implementation

04

DISTRIBUTED ISSUES

Details on how ds issues were handled during the development

05

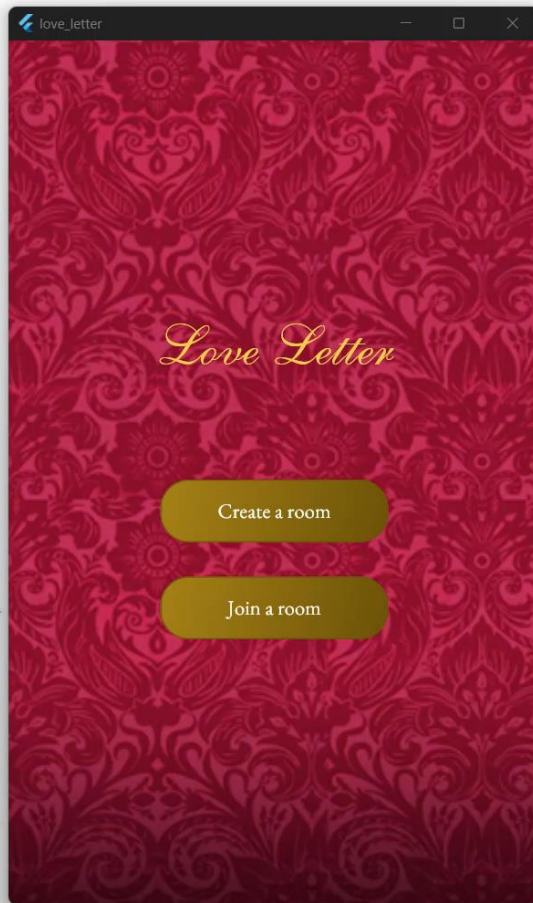
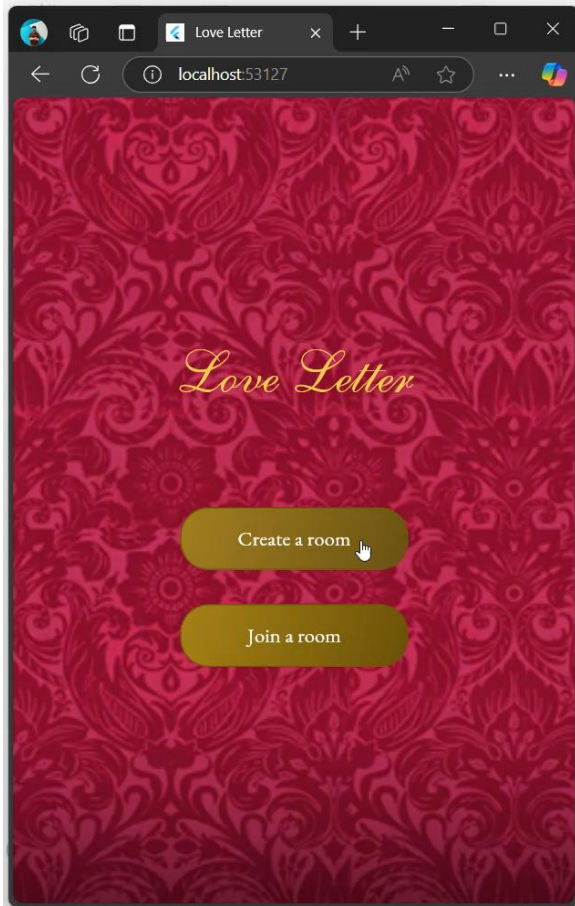
CONCLUSIONS

Conclusions and possible future developments

THE GAME

Love Letter is a turn-based game. Each player starts with a card and takes turns drawing from the deck. Each player activates a card and passes the turn. The effects are varied and allow you to look at cards, eliminate players, and much more. The winner is the one who, at the end of the deck, has the highest card or the only player left alive.







TECHNOLOGIES

For the development of this project, different **technologies** were chosen, and each choice was made for experience or for convenience related to the frameworks.

SUPABASE

Supabase is an open-source platform that provides backend functionality like Firebase, based on PostgreSQL. It offers authentication, real-time SQL database, file storage, and serverless functions. In particular, **Realtime** is a feature of that allows you to listen in real time for changes to your PostgreSQL database data using WebSocket. It supports events such as INSERT, UPDATE, and DELETE, allowing users to automatically synchronize data between clients without the need for manual updates. It offers filters to receive only specific events.



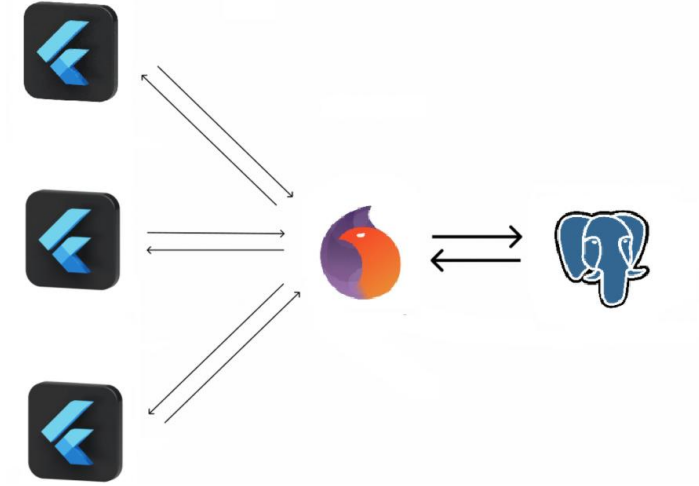


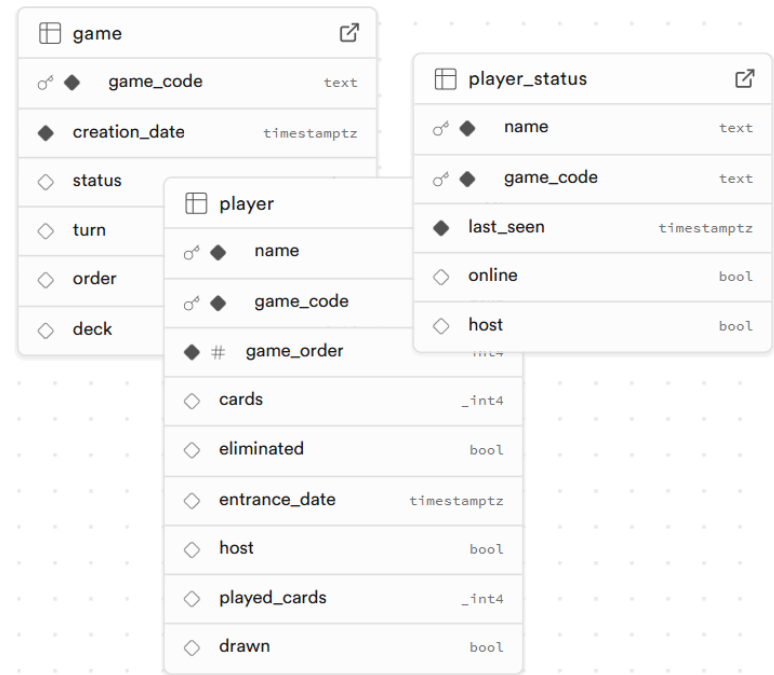
FLUTTER & BLoC

Flutter is a framework that lets you build natively compiled mobile, web, and desktop applications from a single codebase. In particular, **BLoC** (Business Logic Component) is a state management pattern for Flutter that separates business logic from UI. It uses event streams and states to manage user interactions reactively. Additionally, the Supabase package was used for communication between the app and the backend.

IMPLEMENTATION

To develop this application, it was necessary to design both a series of tables and functions on the backend side, and the entire app and the subsequent connection to the backend. Strategies were used in the app to make the rebuilding of the components efficient. Finally, it should be considered that all communication with the server remains transparent to the client, which in fact believes it is performing actions directly on the database.





BACKEND

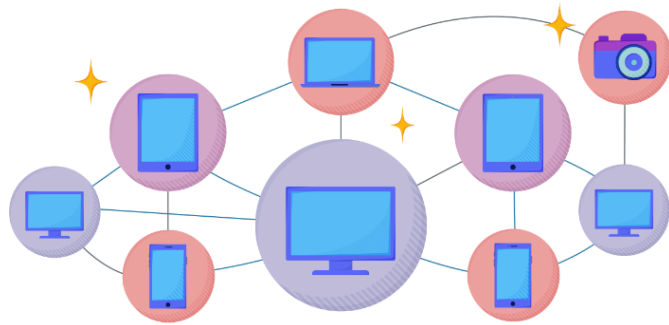
The backend includes three **real-time tables** (image on the side), various **functions** (data entry, game initialization, player status update), a **trigger** to manage the alternation of players between turns and a **job** that, if necessary, deletes inactive games with no connected players

APP

As seen previously, the application has three screens (home, room and game). In the last two each player sends **heartbeats** (every few seconds) to confirm his status: if he goes offline, the backend will no longer receive signals and will declare him offline, until a possible removal from the room (in the first case) or elimination from the game (second case). If a player manages to reconnect, a **fetch** is performed to ensure they have up-to-date data. Table changes are handled by calling **rpc functions** from the backend, to optimally manage concurrency.

```
_statusSubscription = supabase
  .from('player_status')
  .stream(primaryKey: ["name", "game_code"])
  .eq("game_code", widget.gameCode)
  .listen((List<Map<String, dynamic>> data) async {
    await context.read<StatusCubit>().updateStatus(data);
  });

void _startUpdateTimer() {
  _updateTimer = Timer.periodic(
    const Duration(seconds: 1),
    (timer) => updateLastSeen(),
  );
}
```



DISTRIBUTED ISSUES

The project handled various distributed systems **issues** (seen during the course).

Some of these were handled manually, while others are handled automatically by

Supabase Realtime without us having to worry (too much). Obviously, it is important to understand how they are handled.

CONCURRENCY & DATA CONSISTENCY

PostgreSQL efficiently manages **concurrency** thanks to mechanisms such as MVCC and lockings. For **data consistency**, Realtime is based on PostgreSQL's logical replication, but uses an eventual consistency approach: updated data in the system gradually propagates to connected clients, and over time all nodes will converge to the same state. This is a great solution for applications that require real-time updates with tolerance for slight temporary inconsistencies.



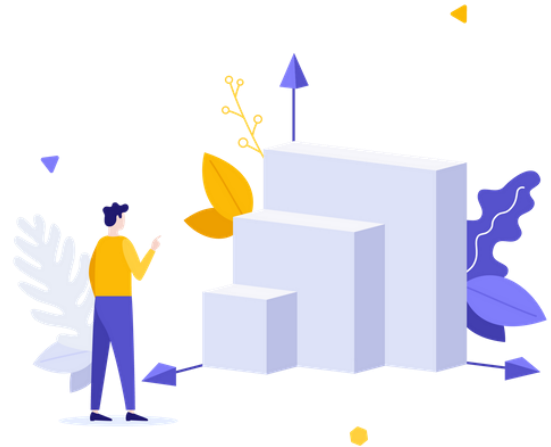


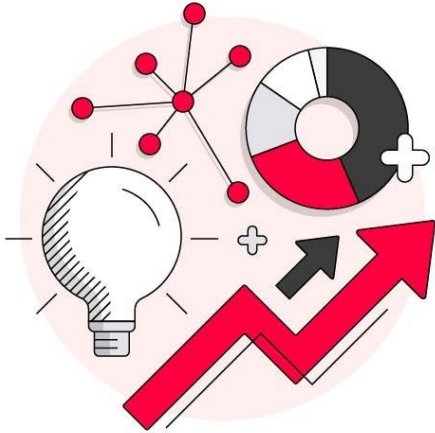
RELIABILITY & AVAILABILITY

To ensure **reliability**, Supabase uses a resilient infrastructure with data replication and automatic failover, supported by a global cluster of real-time servers designed for high availability and fault tolerance. Furthermore, Supabase is committed to ensuring high **availability** of its services, offering a Service Level Agreement (SLA) that ensures 99.9% availability of the total time in each calendar month.

SCALABILITY

Scalability is vital as it ensures the service can handle increased data load or users without compromising performance. Supabase can handle up to 10.000 concurrent clients, 2500 channel joins and 2.500 messages per second. It can be said that the project is implemented to be scalable by design.





CONCLUSIONS

This was a very formative project for the topics covered. Future developments may include the use of **Row Level Security** and **Authentication** (omitted for time reasons, and because it would have involved a login/registration mechanism). As for the game itself, it would be nice to expand the logic to implement newer existing versions (with new cards and mechanisms).