1. Project Overview

This document serves as the complete technical blueprint for the **Fruit Cards** real-time multiplayer card game. It is designed to be an extremely detailed guide for a developer of any skill level, explaining every component, data model, and interaction from the ground up.

2. System Architecture

The game will be built using a **client-server architecture**. This means the game's core logic runs on a central server, and each player's device (the client) connects to it. This approach is essential for a real-time multiplayer game to ensure all players see the same information and to prevent cheating.

- **Frontend (Client):** The user interface. It will be a single-page application built with **React** and **TypeScript** for type safety, using **Vite** for a fast development experience. Styling will be handled with **Tailwind CSS**, and navigation between pages will be managed by **React Router**.
- **Backend (Server):** The "brain" of the game. A **Node.js** server using the **Express** framework will handle all the game rules, player connections, and data. It will use the **Socket.IO** library to enable fast, real-time communication.
- **Database:** A local **SQLite** database will store game and player data. This is crucial for our session persistence feature, as it allows players to reconnect and resume a game.

3. Frontend Implementation Details

The frontend will be a single-page application, which means the browser will only load one index.html file. As the user navigates, React will dynamically change what is shown on the page without reloading.

3.1 Component Structure and Logic

The application is built from several components. Each component is a self-contained piece of the user interface.

- <App.tsx>: This is the root component where everything else is rendered. It will be responsible for setting up the **routing** for our different pages (like the landing page, lobby, and game) using React Router. It also initializes our global state, such as the Socket.IO connection, using a React Context.
- <LandingPage.tsx>: This is the first screen the player sees.
  - **State:** It will use local React state hooks (useState) to manage the text typed into the nickname and room code input fields.
  - **Logic:** When the player clicks "Create Room," this component will take the nickname and send a createRoom event to the backend via our Socket.IO connection. When the player clicks "Join Room," it will send a joinRoom event with

the nickname and room code.

- <LobbyPage.tsx>: This page is where players wait before a game starts.
    - **State:** It will receive and display a list of all players in the room from the server. It will also manage the state for the leader's controls, like the number of card types chosen.
    - **Logic (Leader):** The player who created the room is the leader. This component will show them a control panel to select the fruit types. When they click "Start Game," the component will send a startGame event to the backend with the chosen types.
    - **Logic (Player):** For non-leaders, this component simply displays a list of players and a message indicating they are waiting for the game to start.
- <GamePage.tsx>: This is the main screen where the game is played.
    - **State:** This is the most complex component. It will manage the player's own hand of cards, the number of cards in each opponent's hand, who the current player is, and the countdown timer. All this information is received from the backend via Socket.IO events.
    - **Logic:** When it's the player's turn, they can click on a card to select it. The component will then send a passCard event to the server. It will also visually show the turn timer counting down.
- <GameOverPage.tsx>: This page appears when the game ends.
    - **State:** It will display the winner's nickname and the winning card type, which are received from the server.
    - **Logic:** It will have buttons to "Play Again" (which sends a startGame event if you're the leader) or "Return to Home" (which navigates back to the landing page).

3.2 State Management and Context

In React, useState is great for a component's local data. But for information that many components need to share, like the Socket.IO connection, passing it down through props becomes complicated (a problem known as "prop drilling").
We will solve this using React's **Context API**. We will create a SocketContext that holds the Socket.IO connection object. Any component that needs to talk to the server can simply access this context without having to receive the object from a parent component.

3.3 Localization

To support both English and Arabic, we will use a library like **i18next**. We will create two JSON files, en.json and ar.json, that contain all the text used in the UI. The application will check the user's browser language setting and load the appropriate file.

4. Backend Implementation Details

The backend is the master of the game. It controls all the rules, timers, and data, so a player's actions are never trusted and always validated by the server.

4.1 Express Server and Socket.IO

First, we set up an **Express** server. This server's main job is to serve our frontend application when a user first visits the page. The real work is done by **Socket.IO**, which creates a persistent, real-time connection between the server and each player.
The server will have a main file, src/index.js, that does the following:

1. **Starts the Express server** to listen for HTTP requests.
2. **Initializes the database connection** by calling a function from our src/db/database.js file.
3. **Sets up the Socket.IO server**, connecting it to the Express server.
4. **Defines the Socket.IO event handlers** by calling a function from src/socket/socketHandlers.js, which listens for all events from the clients.

4.2 Socket.IO Event Bus: The Central Nervous System

The heart of our real-time game is the Socket.IO event bus. Think of it like a telegraph system where the frontend and backend send each other messages (events) with specific payloads (the data). This section details every single one of those messages.

Server-to-Client Events (Backend emits, Frontend on)

These events are sent from the server to the client. The frontend code listens for these events to update the player's view of the game.

- roomCreated
    - **Purpose:** This event confirms that a new game room has been successfully created on the server.
    - **Payload:** { roomId: string }
    - **Action:** When a player creates a room, the server immediately sends this event back to that specific player. The frontend should take the roomId from the payload, store it (e.g., in a state variable), and navigate the user to the lobby page.
- playerJoined
    - **Purpose:** This event informs all players in a room that a new player has connected.
    - **Payload:** { playerId: string, nickname: string, isLeader: boolean }
    - **Action:** The server broadcasts this event to everyone in the room. The frontend on each player's device listens for this and adds the new player's details (nickname, leader status) to its list of players in the lobby.
- playerLeft
    - **Purpose:** This event tells everyone that a player has left the room, either by closing the tab or manually exiting.
    - **Payload:** { playerId: string }
    - **Action:** The server broadcasts this event. The frontend should remove the player with the matching playerId from its player list display.

- gameStarted
    - **Purpose:** This event marks the beginning of the game. It's the signal to all clients to transition from the lobby to the main game screen.
    - **Payload:** { initialHands: { [playerId: string]: Card[] } }
    - **Action:** The server sends this event to all players in the room. The payload contains each player's initial hand of cards. The frontend should take its own hand from this object, save it to a state variable, and then navigate to the <GamePage>.
- turnUpdate
    - **Purpose:** This is the most frequently used event. It synchronizes the game state across all players at the start of every new turn.
    - **Payload:** { currentTurnId: string, timerDuration: number, cardPassed?: Card }
    - **Action:** The server broadcasts this event to all players. The frontend should update its state to reflect the new currentTurnId, reset its turn timer UI with timerDuration, and, if cardPassed is present, it should visually "pass" that card from the previous player to the current one. This is how everyone knows whose turn it is and which card was just passed.
- gameWinner
    - **Purpose:** This event signals that a player has won the game.
    - **Payload:** { winnerId: string, winnerNickname: string, winningCardType: string }
    - **Action:** The server sends this to all players. The frontend should display a message announcing the winner, show the winning fruit type, and navigate to the <GameOverPage>.
- sessionRestored
    - **Purpose:** This is for the persistence feature. It allows a player who has disconnected to reconnect and resume the game.
    - **Payload:** { gameState: object } (This object will contain all the necessary data: player hands, room status, etc.)
    - **Action:** The server sends this to the specific client that just reconnected. The frontend should use this comprehensive gameState object to completely restore its view of the game, including player hands, the current turn, and the timer.
- roomNotFound
    - **Purpose:** To handle the error case where a player attempts to join a room that doesn't exist.
    - **Payload:** None.
    - **Action:** The frontend should display a user-friendly error message indicating that the room code is invalid.
- notYourTurn
    - **Purpose:** A security event to inform a player that they have tried to make a move when it is not their turn.
    - **Payload:** None.
    - **Action:** The frontend should display a warning or error message to the user. This is a crucial validation step to prevent cheating.

Client-to-Server Events (Frontend emits, Backend on)

These events are sent from the player's device to the backend. The backend listens for them, validates the request, and updates the game state accordingly.

- createRoom
  - **Purpose:** A player's request to create a new game room.
  - **Payload:** { nickname: string }
  - **Action:** The backend receives this, generates a unique room ID and a sessionId, creates entries in the GameRooms and Players database tables, and then sends the roomCreated event back to the client.
- joinRoom
  - **Purpose:** A player's request to join an existing game room.
  - **Payload:** { roomId: string, nickname: string, sessionId?: string }
  - **Action:** The backend checks if the roomId exists. If it does, a new player is added to the database. The server then sends a playerJoined event to all clients in the room, letting them know a new player has joined.
- startGame
  - **Purpose:** The leader's request to start the game.
  - **Payload:** { cardTypes: string[], timerDuration: number }
  - **Action:** The backend validates that the sender is the room's leader. If so, it deals the cards, updates the GameRooms database table, and broadcasts the gameStarted event to all players.
- passCard
  - **Purpose:** The player's action of passing a card.
  - **Payload:** { cardId: string }
  - **Action:** The backend first validates that it is the sender's turn and that the cardId exists in their hand. If the action is valid, the server updates the database to reflect the new hands, checks for a win condition, and broadcasts a turnUpdate event.
- requestSession
  - **Purpose:** A player's attempt to rejoin a game after a disconnection.
  - **Payload:** { sessionId: string }
  - **Action:** The backend looks up the sessionId in the database. If a matching player is found, it updates their isOnline status, and sends a sessionRestored event with the current game state to the client.
- leaveRoom
  - **Purpose:** A player's voluntary exit from a room.
  - **Payload:** { roomId: string }
  - **Action:** The backend removes the player from the database and broadcasts a playerLeft event to the remaining players in the room.

4.3 Database Schema (SQLite)

We will use a lightweight database called **SQLite** to store our data in a single file named game.db. This is perfect for a small-scale application like this because it's easy to set up and manage. We will have two tables.

- GameRooms table:
  - roomId: A unique code (like "ABCD") for each room.
  - leaderId: The ID of the player who created the room.
  - status: The current state of the game (e.g., 'lobby', 'in_progress').
  - cardTypes: A string representing a JSON array of the fruit types chosen by the leader (e.g., ["apple", "banana"]). We store it as a string because SQLite doesn't have a specific type for arrays.
  - timerDuration: How long each turn is.
  - currentPlayerIndex: An integer to track whose turn it is in the turn order.
  - playerIds: A JSON array of the player IDs in the room, stored as a string.
- Players table:
  - playerId: A unique ID for each player.
  - sessionId: A unique ID used for session persistence. We will save this in the player's local storage.
  - nickname: The player's chosen name.
  - hand: A string representing a JSON array of the cards in the player's hand.
  - roomId: The ID of the room the player is in.
  - isLeader: A true/false value to know who is the leader.
  - isOnline: A true/false value to track if a player is currently connected.

4.4 Database Implementation (SQLite)

All database logic will be in the src/db/database.js file. This centralizes our database code, making it easy to manage.

4.4.1 Initialization

When the server starts, we'll create the tables if they don't exist. The sqlite3.Database constructor will automatically create the game.db file for us.

```
// src/db/database.js
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('./game.db');

const initDb = () => {
  // We use `db.serialize()` to make sure our commands run in order.
  db.serialize(() => {
    // We use `CREATE TABLE IF NOT EXISTS` so we don't get an error
    // if the tables already exist.
    db.run(`CREATE TABLE IF NOT EXISTS GameRooms (
      roomId TEXT PRIMARY KEY UNIQUE,
      leaderId TEXT,
      status TEXT,
      cardTypes TEXT,
      timerDuration INTEGER,
      currentPlayerIndex INTEGER,
      playerIds TEXT
    )`);

    db.run(`CREATE TABLE IF NOT EXISTS Players (
      playerId TEXT PRIMARY KEY UNIQUE,
      sessionId TEXT UNIQUE,
      nickname TEXT,
      hand TEXT,
      roomId TEXT,
      isLeader BOOLEAN,
      isOnline BOOLEAN,
      FOREIGN KEY(roomId) REFERENCES GameRooms(roomId)
    )`);
  });
};

module.exports = { db, initDb };
```

4.4.2 CRUD Operations

CRUD stands for Create, Read, Update, Delete. These are the main operations we'll need for our data. Each function will use a Promise, which is a modern way to handle asynchronous operations and avoid "callback hell."

```
// src/db/database.js - additional functions

// Inserts a new room into the database.
const createRoom = (roomId, leaderId) => {
    return new Promise((resolve, reject) => {
        const stmt = db.prepare("INSERT INTO GameRooms (roomId, leaderId, status, playerIds)
VALUES (?, ?, ?, ?)");
        const initialPlayers = JSON.stringify([]);
        // We use `run` to execute a statement that doesn't return data.
        stmt.run(roomId, leaderId, 'lobby', initialPlayers, function(err) {
            if (err) {
                return reject(err);
            }
            // `this.lastID` is useful for tables with auto-incrementing IDs, but here we use a custom
one.
```

```
            resolve(this.changes);
        });
        stmt.finalize(); // Always clean up the prepared statement.
    });
};

// Adds a new player to the Players table.
const addPlayer = (playerId, sessionId, nickname, roomId, isLeader) => {
    return new Promise((resolve, reject) => {
        const stmt = db.prepare("INSERT INTO Players (playerId, sessionId, nickname, hand,
roomId, isLeader, isOnline) VALUES (?, ?, ?, ?, ?, ?, ?)");
        const initialHand = JSON.stringify([]);
        // `isOnline` is set to 1 (true) for the new player.
        stmt.run(playerId, sessionId, nickname, initialHand, roomId, isLeader ? 1 : 0, 1,
function(err) {
            if (err) {
                return reject(err);
            }
            resolve(this.changes);
        });
        stmt.finalize();
    });
};

// Fetches all players in a specific room.
const getPlayersInRoom = (roomId) => {
    return new Promise((resolve, reject) => {
        // `db.all` is used when we expect to get multiple rows back.
        db.all("SELECT * FROM Players WHERE roomId = ?", [roomId], (err, rows) => {
            if (err) {
                return reject(err);
            }
            resolve(rows);
        });
    });
};

// Fetches a player using their session ID for reconnection.
const getPlayerBySessionId = (sessionId) => {
    return new Promise((resolve, reject) => {
        // `db.get` is used when we expect a single row.
        db.get("SELECT * FROM Players WHERE sessionId = ?", [sessionId], (err, row) => {
            if (err) {
```

```
            return reject(err);
          }
          resolve(row);
        });
    });
};

// ... more functions for updating and deleting data
```

5. Core Game Logic

The backend is where all the game rules are enforced.

5.1 Game Initialization

When the leader starts the game, the server will:

1. **Generate a Deck:** Create an array of card objects, where each card has an id and a type (e.g., id: '1a', type: 'apple'). The number of cards is based on the number of players and card types.
2. **Shuffle the Deck:** A simple shuffling algorithm will reorder the cards randomly.
3. **Deal Cards:** The server will iterate through the list of players and deal an equal number of cards to each, with the first player getting one extra card.
4. **Set Turn Order:** The order in which players joined the room will determine the turn order. The player with the extra card will start.
5. **Start the Timer:** The server will start a new timer for the first player's turn.

5.2 Turn Management

The backend will use setTimeout or setInterval to create a server-side timer for each player's turn.

- **On a Valid Move:** When a player passes a card, the server validates the move. If it's valid, the timer is cleared, the game state is updated, and a turnUpdate event is broadcast to all clients. A new timer is started for the next player.
- **On Timeout:** If the timer runs out before the player makes a move, the server will automatically select a random card from their hand, pass it to the next player, and broadcast the turnUpdate event.

5.3 Win Condition

After a card is passed, the server will check the receiving player's hand. It will iterate through their cards to see if all cards belong to a single fruit type. If so, a gameWinner event is broadcast, and the game ends.

6. Session Management & Reconnection

This is a key feature that relies on our database.

1. **Initial Connection:** When a player joins a room for the first time, the server generates a unique sessionId and saves it along with the player's data in the database. It then sends this ID to the client, which stores it in localStorage.
2. **Disconnection:** If a player's connection drops, the server marks their isOnline status as false in the database but does not remove them from the room.
3. **Reconnection:** When the player reconnects, the frontend will automatically send their stored sessionId to the server. The server looks up this ID in the database, finds the player, and restores their session by marking their isOnline status as true and sending them the latest game state.

7. File Structure

This structure provides a clear separation of concerns, making the project easier to manage and scale.

7.1 Frontend (/frontend)

```
/frontend
├── public/
│   ├── index.html
│   └── ... (images, fonts, etc.)
├── src/
│   ├── assets/
│   │   ├── fruits/
│   │   │   ├── apple.svg
│   │   │   ├── banana.svg
│   │   │   └── ...
│   │   └── sounds/
│   │       ├── card_pass.mp3
│   │       ├── turn_start.mp3
│   │       └── ...
│   ├── components/
│   │   ├── common/
│   │   │   ├── Button.tsx
│   │   │   └── Modal.tsx
│   │   ├── Card.tsx
│   │   ├── Player.tsx
│   │   └── TurnTimer.tsx
│   ├── context/
│   │   └── SocketContext.tsx
│   ├── hooks/
│   │   └── useSocket.ts
│   ├── pages/
│   │   ├── GamePage.tsx
│   │   ├── GameOverPage.tsx
│   │   ├── LandingPage.tsx
│   │   └── LobbyPage.tsx
│   ├── App.tsx
│   ├── main.tsx
│   ├── index.css
│   └── i18n.ts
├── package.json
└── vite.config.js
```

7.2 Backend (/backend)

```
/backend
├── src/
│   ├── db/
│   │   └── database.js
│   ├── services/
│   │   ├── gameManager.js
│   │   └── playerManager.js
│   ├── socket/
│   │   └── socketHandlers.js
│   ├── index.js
│   └── routes.js
├── package.json
└── package-lock.json
```