Milestone 3 - Game API Design Document
Golf Card Game - Multiplayer Web App
Team Members: Andre Achtar-Zadeh, Mikias Berhane, Ahmad Harris, Sana Yasini
Date: 11/12/2025

Brief Game Description:
Our project is a multiplayer Golf Card Game where 2-4 players compete to end with the lowest total score after 9 rounds. Each player has a hidden 3x4 grid of cards and takes turns flipping and swapping cards with the deck or discard pile to reduce their score. The server handles all game logic, validates every action, manages turns, updates scores, and broadcasts real-time updates to players using Socket.io.

API Endpoints
4 Game Management endpoints
3 Game Action endpoints
Full authorization rules
Full validation rules
Full state update logic
Full socket.io events

2.1. Game Management Endpoints (4 required)

Endpoint 1: Create Game

1. Endpoint Name:
Create Game

2. Method + Route:
POST /api/games

3. Purpose:
Creates a new game with a join code, assigns the host, and prepares the game for players to join.

4. Authorization:
Must be logged in
User becomes the game host
No other restrictions

5. Request Body:
```
{
  “maxPlayers”: 4
}
```

6. Validation Checks:
User is logged in
maxPlayers is between 2-4

7. State Updates:
Insert row in games with:
id (UUID)
join_code (random 6-12 chars)
host_id
status = “waiting”
round_number = 0
current_turn = null
deck_count = 52

8. Success Response:
201 Created
```
{
  “game_id”: “<uuid>”,
  “join_code”: “ABC123”,
  “status”: “waiting”
}
```

9. Error Cases:
400 invalid maxPlayers
401 not logged in

10. Socket.io Events:
None; game not started yet

Endpoint 2: Join Game

1. Endpoint Name:
Join Game

2. Method + Route:
POST /api/games/join

3. Purpose
Player joins an existing game via join code.
4. Authorization
Must be logged in

Game must be in "waiting" state

Game must not be full

5. Request Body
```
{
  "joinCode": "ABC123"
}
```

6. Validation Checks
Game exists with this join code

status = "waiting"

Player not already in the game

Game not full (max 4 players)

7. State Updates
Insert a row into game_players linking user ↔ game

Assign seat number based on join order

8. Success Response
 200 OK
```
{
  "joined": true,
  "game_id": "<uuid>"
}
```

9. Error Cases
 401 not logged in
 404 join code invalid
 409 game full
10. Socket.io Events
 game:player:joined (public)
Sent to all players already in the game

Contains updated player list & seats
Endpoint 3: Start Game
1. Endpoint Name:
 Start Game
2. Method + Route:
 POST /api/games/:game_id/start
3. Purpose:
 Deals cards, builds deck, sets round_number to 1, and begins the first turn.

4. Authorization:
Must be logged in

Must be in game_players

Must be host of this game

Game must be in waiting state

5. Request Body:
None
6. Validation Checks:
Game exists

User is host

2–4 players

No cards already exist for this game (prevents double-start)

7. State Updates:
Insert 52 cards into cards table using card_defs

Set zone = 'deck', order_in_deck = shuffled position

Deal 12 cards to each player → zone = 'grid', row, col, face_up = false

Set round_number = 1

Set current_turn = seat 1 player

Insert into turns table: { action: "game_started" }

8. Success Response:
202 Accepted
```
{
  "status": "playing",
  "round_number": 1
}
```

9. Error Cases:
403 not host
409 game already started
10. Socket.io Events:

game:state:update (public)

game:hand:update (private to each player)

game:notification:private (private to a specific player – used to send warnings or error messages that shouldn’t be broadcast to everyone)

Endpoint 4: Get Game State
1. Endpoint Name:
 Get Game State
2. Method + Route:
 GET /api/games/:game_id/state
3. Purpose:
 Returns filtered game state for the requesting player (public + private data).
4. Authorization:
Must be logged in

Must be in this game

5. Request Body:
 None
6. Validation Checks:
Game exists

User in game_players

7. State Updates:
 None (read only)
8. Success Response:
 200 OK
 Includes:
Public state: all face-up cards, scores, turn

Private state: your grid including hidden cards

9. Error Cases:
 403 forbidden
 404 game not found
10. Socket.io Events:
 None
2.2 Game Action Endpoints (3)
Endpoint 5: Flip Card
1. Endpoint Name:
 Flip Card

2. Method + Route:
 POST /api/games/:game_id/flip-card
3. Purpose:
 Current player flips one hidden card in their 3×4 grid.
4. Authorization:
Logged in

In game

Must be current_turn player

Game must be "playing"

5. Request Body:
```
{
  "row": 1,
  "col": 2
}
```

6. Validation Checks:
Card exists at (row, col)

Card belongs to this player

Card is face_down

It is the player’s turn

7. State Updates:
Update card → face_up = true

Recalculate player's score

Insert turn log { action: "flip_card" }

8. Success Response:
 202 Accepted
9. Error Cases:
 400 invalid flip
 403 not your turn
10. Socket.io Events:
game:state:update

game:hand:update (private)

Endpoint 6: Draw & Swap Card
1. Endpoint Name:
 Draw and Swap Card
2. Method + Route:
 POST /api/games/:game_id/draw-and-swap
3. Purpose:
 Player draws from deck/discard and either discards or swaps with their grid card.
4. Authorization:
Logged in

In game

Must be current_turn player

5. Request Body:
```
{
  "source": "deck",
  "swap": { "row": 0, "col": 3 },
  "discard_only": false
}
```

6. Validation Checks:
deck_count > 0 or discard pile has cards

swap pos valid

card at swap belongs to player

move legal under game rules

7. State Updates:
Pull top card from deck/discard

If discard_only == true: put drawn card in discard and end turn

Else:

    Put drawn card → (row, col)

    Moved-out card → discard

Recompute score

Advance turn

Insert { action: "draw_and_swap" }

8. Success Response:
 202 Accepted
9. Error Cases:
 400 invalid swap
 404 empty deck
10. Socket.io Events:
game:state:update

game:hand:update

game:turn:changed

Endpoint 7: End Turn / Knock
1. Endpoint Name:
 End Turn
2. Method + Route:
 POST /api/games/:game_id/end-turn
3. Purpose:
 Ends the player's turn. If knocking is enabled, can signal last round.
4. Authorization:
Logged in

In game

Must be current_turn

5. Request Body:
```
{
  "knock": true
}
```

6. Validation Checks:
Turn belongs to user

Game = playing

Player hasn’t already knocked/ended prematurely

7. State Updates:
Insert { action: "end_turn", payload: { knock } }

If knocking triggers final rotation: set a flag

If round ends:

- Compute all scores
- Insert into round_scores
- Update total_points
- If round < 9 → start next round
- If round = 9 → set game.status = "ended"

8. Success Response:
 202 Accepted
9. Error Cases:
 409 round already ended
 403 not your turn
10. Socket.io Events
game:turn:changed

game:round:ended (if applicable)

3. Socket.io Events (4)
Event 1: game:state:update
Scope: all players
 Trigger: any public state change
 Data:
face-up cards

discard pile top

current_turn

scores

round_number

Event 2: game:player:joined

Scope: all players
 Trigger: player joins
 Data:
updated player list

seat numbers

Event 3: game:hand:update (Private)
Scope: one player only
 Trigger: grid changes (deal, flip, swap)
 Data:
full 3×4 grid

face_up flags

private cards

player’s round score
Event 4: game:turn:changed
Scope: all players
 Trigger: end turn, draw-swap, etc.
 Data:
previous_player

next_player

round_number

Event 5 (optional): game:round:ended
Scope: all players
 Trigger: end of round
 Data:
round scores

total scores

next round number

game_over flag

4. Common Pitfalls & How We Avoid Them
Trusting client → Server validates everything

Broadcasting private cards → We use private events

Vague validation → We list every rule explicitly

Race conditions → Server is the single source of truth

5. Conclusion

This design document outlines all required endpoints, validation logic, and real-time events for our Golf Card Game. The server ensures fairness, consistency, and proper turn order by validating every move and maintaining the game state.