Complete Game Workflow Documentation

Version: 2.0
Last Updated: October 30, 2025
Document Type: Complete System Reference

Table of Contents

System Overview

The SerpentRace game system uses a **hybrid architecture**:

- **REST API**: Game creation, joining, and initial setup
- **WebSocket (Socket.IO)**: Real-time gameplay, card interactions, and game state updates
- **Redis**: Session management, pending states, turn tracking
- **PostgreSQL**: Persistent game data, user profiles, decks

Technology Stack

- **Backend**: Node.js + TypeScript + Express
- **Real-time**: Socket.IO (WebSocket library)
- **Database**: TypeORM + PostgreSQL
- **Cache/Session**: Redis
- **Authentication**: JWT tokens (Access + Refresh)

Game Flow Diagram

```
┌──────────────────────────────────────────────────────────────────────────┐
│                          GAME LIFECYCLE                                  │
└──────────────────────────────────────────────────────────────────────────┘


1. GAME CREATION (REST)
   │
   ├─ POST /api/games/start
   │  ├─ Gamemaster creates game with deck selection
   │  ├─ Game Code generated (6 characters)
   │  └─ Game state: "waiting"
   │
   ▼

2. PLAYER JOINING (REST + WebSocket)
   │
   ├─ POST /api/games/join
   │  ├─ Validate game code
   │  ├─ Check game type (PUBLIC/PRIVATE/ORGANIZATION)
   │  ├─ Add player to game.players[]
   │  └─ Return gameToken for WebSocket auth
   │
   ├─ WebSocket Connection
   │  ├─ Connect to /game namespace
   │  ├─ Emit: game:join with gameToken
   │  ├─ Server validates & joins rooms
   │  └─ Broadcast: game:player-joined to all
   │
   ├─ Emit: game:ready (player marks ready)
   └─ Broadcast: game:player-ready to all
   │
   ▼

3. GAME START (REST)
   │
   ├─ POST /api/games/:gameId/start
   │  ├─ Only gamemaster can start
   │  ├─ Check minimum players (2+)
   │  ├─ Generate board (100 fields with pattern)
   │  ├─ Shuffle and assign turn sequence
```

```
   │  ├─ Initialize player positions (all at 0)
   │  └─ Game state: "active"
   │
   ├─ Broadcast: game:started
   │  ├─ Board data sent to all players
   │  ├─ Turn sequence revealed
   │  └─ First player notified
   │
   ▼

4. GAMEPLAY LOOP (WebSocket)
   │
   ├─ DICE ROLL
   │  ├─ Emit: game:dice-roll { diceValue: 1-6 }
   │  ├─ Broadcast: game:dice-rolled
   │  ├─ Calculate new position
   │  ├─ Update player position
   │  ├─ Broadcast: game:player-moved
   │  └─ Check field type at new position
   │
   ├─ SPECIAL FIELD LANDING (positive/negative/luck)
   │  ├─ Wait 2 seconds (frontend animation)
   │  ├─ Draw card from appropriate deck
   │  ├─ Broadcast: game:card-drawn
   │  └─ Branch based on card type:
   │
   │     ┌─ QUESTION CARD (types 0-4)
   │     │  ├─ Emit to player: game:card-drawn-self (60s)
   │     │  ├─ Player answers
   │     │  ├─ Emit: game:card-answer
   │     │  ├─ Broadcast: game:answer-submitted
   │     │  ├─ Validate answer
   │     │  ├─ Broadcast: game:answer-validated
   │     │  ├─ Determine if guess required:
   │     │  │  ├─ Positive field + correct = GUESS
   │     │  │  ├─ Negative field + wrong = GUESS
   │     │  │  ├─ Positive field + wrong = NO MOVEMENT
   │     │  │  └─ Negative field + correct = NO MOVEMENT
   │     │  │
   │     │  └─ IF GUESS REQUIRED:
```

```
    │     │     ├─ Emit: game:position-guess-request (30s)
    │     │     ├─ Show pattern modifier, dice, stepValue
    │     │     ├─ Player guesses position
    │     │     ├─ Emit: game:position-guess
    │     │     ├─ Broadcast: game:position-guess-broadcast
    │     │     ├─ Calculate with BoardGenerationService
    │     │     ├─ Apply -2 penalty if wrong
    │     │     ├─ Broadcast: game:guess-result
    │     │     ├─ Check win condition (position >= 100)
    │     │     └─ Check secondary landing
    │
    │     ┌─ LUCK CARD (type 6)
    │     │  ├─ Process immediately (no answer)
    │     │  ├─ Broadcast: game:card-result
    │     │  ├─ Apply consequence:
    │     │  │  ├─ MOVE_FORWARD (0): Move X steps
    │     │  │  ├─ MOVE_BACKWARD (1): Move X steps back
    │     │  │  ├─ LOSE_TURN (2): Store X turns to lose
    │     │  │  ├─ EXTRA_TURN (3): Store X extra turns
    │     │  │  └─ GO_TO_START (5): Return to position 1
    │     │  ├─ Broadcast: game:luck-consequence
    │     │  └─ Advance turn (with turn tracking)
    │
    │     └─ JOKER CARD (type 5) - Secondary Landing Only
    │        ├─ Broadcast: game:joker-drawn
    │        ├─ Request gamemaster decision
    │        ├─ Emit to gamemaster: game:gamemaster-decision-request (120s)
    │        ├─ Gamemaster decides (approve/reject)
    │        ├─ Emit: game:gamemaster-decision
    │        ├─ Broadcast: game:gamemaster-decision-result
    │        ├─ Determine if guess required:
    │        │  ├─ Positive field + approved = GUESS
    │        │  ├─ Negative field + rejected = GUESS
    │        │  ├─ Positive field + rejected = NO MOVEMENT
    │        │  └─ Negative field + approved = NO MOVEMENT
    │        │
    │        └─ IF GUESS REQUIRED:
    │           ├─ Emit: game:joker-position-guess-request (30s)
    │           ├─ Player guesses with dice=6
    │           ├─ Emit: game:joker-position-guess
```

```
    │           ├─ Calculate with BoardGenerationService
    │           ├─ Apply -2 penalty if wrong
    │           ├─ Broadcast: game:joker-complete
    │           └─ Check secondary landing
    │
    ├─ TURN ADVANCEMENT
    │  ├─ Check if current player has extra turns
    │  │  └─ IF YES: Same player continues, decrement counter
    │  ├─ Find next player in sequence
    │  ├─ Skip players with turns to lose
    │  │  └─ Broadcast: game:players-skipped
    │  ├─ Broadcast: game:turn-changed
    │  └─ Emit to next player: game:your-turn
    │
    └─ WIN CONDITION
       ├─ Player reaches position >= 100
       ├─ Broadcast: game:ended
       ├─ Update database (finished=true, winner)
       └─ Cleanup all Redis data and disconnect
    │
    ▼

5. GAME END & CLEANUP
    │
    ├─ Clear all Redis keys:
    │  ├─ gameplay:{gameCode}
    │  ├─ game_positions:{gameCode}
    │  ├─ pending_card:{gameCode}:{playerId}
    │  ├─ pending_decision:{gameCode}:{requestId}
    │  ├─ player_extra_turns:{gameCode}:{playerId}
    │  └─ player_turns_to_lose:{gameCode}:{playerId}
    │
    ├─ Force disconnect all players from rooms
    ├─ Emit: game:cleanup-complete
    └─ Database updated with final state
```

REST API Endpoints

Authentication Headers

All authenticated endpoints require:

```
Authorization: Bearer <access_token>
```

1. Create Game

Endpoint: `POST /api/games/start`
Auth: Required
Description: Create a new game session with selected decks

Request Body:

```
{
  deckids: string[];      // Array of deck UUIDs (at least 1)
  maxplayers: number;     // Maximum players (2-10)
  logintype: number;      // 0=PUBLIC, 1=PRIVATE, 2=ORGANIZATION
}
```

Response (200 OK):

```
{
  id: string;                    // Game UUID
  gamecode: string;              // 6-character game code
  maxplayers: number;
  logintype: number;
  boardsize: number;             // Default 100
  createdby: string;             // Gamemaster user ID
  orgid: string | null;          // Organization ID (for ORG games)
  gamedecks: GameDeck[];         // Array of decks with cards
  players: string[];             // Empty array initially
  startdate: Date | null;
  enddate: Date | null;
  finished: boolean;
  winner: string | null;
  totaltiles: number;
}
```

Errors:

- `400`: Invalid input (missing deckids, invalid maxplayers, etc.)
- `401`: Not authenticated
- `404`: Deck not found
- `500`: Internal server error

---

2. Join Game

Endpoint: `POST /api/games/join`
Auth: Optional (depends on game type)
Description: Join an existing game using game code

Request Body:

```
{
  gameCode: string;       // 6-character game code
  playerName?: string;    // Required for PUBLIC, optional for authenticated
}
```

Authentication Requirements by Game Type:

- **PUBLIC (0)**: No auth required, playerName mandatory
- **PRIVATE (1)**: Auth required
- **ORGANIZATION (2)**: Auth + organization membership required

Response (200 OK):

```
{
  id: string;                    // Game UUID
  gamecode: string;              // Game code
  playerName: string;            // Player's display name
  playerCount: number;           // Current number of players
  maxPlayers: number;
  gameType: string;              // "PUBLIC" | "PRIVATE" | "ORGANIZATION"
  isAuthenticated: boolean;
  gameToken: string;             // JWT token for WebSocket authentication
}
```

Game Token Payload:

```
{
  gameId: string;
  gameCode: string;
  playerName: string;
  playerId?: string;      // Only if authenticated
  iat: number;            // Issued at
  exp: number;            // Expires in 24 hours
}
```

Errors:

- `400`: Invalid gameCode format or missing playerName
- `401`: Authentication required (PRIVATE/ORGANIZATION games)
- `403`: Organization membership required or mismatch
- `404`: Game not found
- `409`: Game full or already started
- `500`: Internal server error

3. Start Game Play

Endpoint: `POST /api/games/:gameId/start`
Auth: Required (only gamemaster)
Description: Start the actual gameplay after all players are ready

Path Parameters:

- `gameId`: UUID of the game

Request Body: Empty `{}`

Response (200 OK):

```
{
  message: string;               // "Game started successfully"
  gameId: string;
  playerCount: number;
  game: GameAggregate;           // Full game object
  boardData: {
    gameId: string;
    fields: GameField[];         // 100 fields with positions, types, stepValues
    generationComplete: boolean;
    generatedAt: Date;
  }
}
```

Board Generation:

- Generates 100 fields with pattern-based distribution
- Field types: `regular`, `positive`, `negative`, `luck`
- Each field has `stepValue` (0-3) for movement calculations
- Pattern modifiers by zones:
  - Positions 1-20: +2 (easier start)
  - Positions 21-40: -1 (early challenge)
  - Positions 41-60: +1 (mid-game boost)
  - Positions 61-80: -2 (late challenge)

    - Positions 81-100: +3 (final stretch)

Game State Changes:

- `game.startdate` set to current date
- `game.gamePhase` set to "active"
- Turn sequence randomized
- Player positions initialized to 0

Errors:

- `400`: Invalid gameId
- `403`: Only gamemaster can start game
- `404`: Game not found
- `409`: Game already started, not enough players, or players not ready
- `500`: Board generation failed or internal error

WebSocket Events

Connection & Authentication

Namespace: `/game`

Connection:

```
const socket = io('http://localhost:3000/game', {
  auth: {
    token: gameToken  // JWT token from /game/join endpoint
  }
});
```

Authentication Flow:

1. Client connects with `gameToken` in auth
2. Server validates token using `GameTokenService`
3. Server extracts: `gameId`, `gameCode`, `playerName`, `playerId`

4. Server joins socket to rooms:
    - `game_{gameCode}` (all players)
    - `game_{gameCode}:{playerName}` (individual player)
5. Server emits `authenticated` event

Events:

```
// Client → Server: Initial join
socket.emit('game:join', { gameToken: string });

// Server → Client: Authentication success (renamed from 'authenticated')
socket.on('game:joined', {
  gameCode: string;
  playerName: string;
  message: string;
  gameId: string;
  playerId?: string;
  timestamp: string;
});

// Server → All Players: Player joined
socket.on('game:player-joined', {
  playerId: string;
  playerName: string;
  playerCount: number;
  timestamp: string;
});
```

Pre-Game Events

Ready System

```
// Client → Server: Mark player as ready
socket.emit('game:ready', {
  gameCode: string;
  ready: boolean;
});

// Server → All Players: Player ready status changed
socket.on('game:player-ready', {
  playerId: string;
  playerName: string;
  ready: boolean;
  readyCount: number;
  totalPlayers: number;
  allReady: boolean;
  timestamp: string;
});

// Server → All Players: All players ready (can start game)
socket.on('game:all-ready', {
  message: string;
  readyCount: number;
  totalPlayers: number;
  timestamp: string;
});
```

Player Approval System (Private Games Only)

```
// Server → Pending Player: Waiting for gamemaster approval
socket.on('game:pending-approval', {
  message: string;
  gameCode: string;
  timestamp: string;
});

// Server → Gamemaster: Player requesting to join
socket.on('game:player-requesting-join', {
  playerName: string;
  playerId?: string;
  message: string;
  timestamp: string;
});

// Client → Server: Gamemaster approves player
socket.emit('game:approve-player', {
  gameCode: string;
  playerName: string;
});

// Client → Server: Gamemaster rejects player
socket.emit('game:reject-player', {
  gameCode: string;
  playerName: string;
  reason?: string;
});

// Server → Approved Player: Join approved, can reconnect
socket.on('game:approval-granted', {
  gameCode: string;
  playerName: string;
  message: string;
  timestamp: string;
});

// Server → Rejected Player: Join denied
socket.on('game:approval-denied', {
  gameCode: string;
```

```
  playerName: string;
  reason?: string;
  message: string;
  timestamp: string;
});

// Server → All Players: Player was approved
socket.on('game:player-approved', {
  playerName: string;
  playerId?: string;
  message: string;
  timestamp: string;
});

// Client → Server: Join after approval (private games)
socket.emit('game:join-approved', { gameToken: string });
```

Game State Events

```
// Server → Individual Player: Current game state
socket.on('game:state', {
  // Complete game state object
  gameCode: string;
  players: PlayerPosition[];
  currentTurn: number;
  currentPlayer: string;
  turnSequence: string[];
  // ... additional state data
});

// Server → All Players: Game state update
socket.on('game:state-update', {
  // Updated game state after action
});
```

Chat System

```
// Client → Server: Send chat message
socket.emit('game:chat', {
  gameCode: string;
  message: string;
});

// Server → All Players: Chat message received
socket.on('game:chat-message', {
  playerName: string;
  playerId?: string;
  message: string;
  timestamp: string;
});
```

Player Disconnect Events

```
// Server → All Players: Player disconnected
socket.on('game:player-disconnected', {
  playerName: string;
  playerId?: string;
  message: string;
  timestamp: string;
});

// Server → All Players: Player disconnected during card answer
socket.on('game:player-disconnected-during-card', {
  playerName: string;
  playerId: string;
  message: string;
  timestamp: string;
});

// Client → Server: Leave game voluntarily
socket.emit('game:leave', { gameCode: string });

// Server → All Players: Player left game
socket.on('game:player-left', {
  playerName: string;
  playerId?: string;
  message: string;
  playerCount: number;
  timestamp: string;
});
```

Game Start Notification

```
// Server → All Players: Game started (after REST /start call)
socket.on('game:started', {
  message: string;
  boardData: BoardData;
  turnSequence: string[];        // Array of player IDs in turn order
  currentPlayer: string;         // First player ID
  currentPlayerName: string;
  timestamp: string;
});

// Server → Current Player: Your turn notification
socket.on('game:your-turn', {
  message: string;
  canRoll: boolean;
  isExtraTurn?: boolean;         // True if using extra turn
  timestamp: string;
});
```

Gameplay Events

Dice Roll

```
// Client → Server: Roll dice
socket.emit('game:dice-roll', {
  gameCode: string;
  diceValue: number;  // 1-6
});

// Server → All Players: Dice rolled
socket.on('game:dice-rolled', {
  playerId: string;
  playerName: string;
  diceValue: number;
  timestamp: string;
});

// Server → All Players: Player moved
socket.on('game:player-moved', {
  playerId: string;
  playerName: string;
  oldPosition: number;
  newPosition: number;
  diceValue: number;
  timestamp: string;
});
```

Card Drawing & Question Cards

```
// Server → All Players: Card drawn (everyone sees question)
socket.on('game:card-drawn', {
  playerName: string;
  playerId: string;
  cardType: string;              // "QUIZ", "SENTENCE_PAIRING", etc.
  question: string;              // The question text
  fieldType: string;             // "positive" | "negative" | "luck"
  timestamp: string;
});

// Server → Drawing Player: Interactive card details
socket.on('game:card-drawn-self', {
  cardData: {
    cardid: string;
    question: string;
    type: CardType;
    timeLimit: number;             // 60 seconds
    answerOptions?: QuizOption[];  // For QUIZ (multiple choice)
    sentencePairs?: SentencePair[]; // For SENTENCE_PAIRING (left-right matching)
    words?: string[];              // For SENTENCE_PAIRING (legacy scrambled words)
    // ... type-specific data
  };
  timestamp: string;
});

// SENTENCE_PAIRING Data Structures

/**
 * Sentence pair for left-right matching
 */
interface SentencePair {
  id: string;           // Unique identifier (e.g., "pair_0", "pair_1")
  left: string;         // Left part to match (shown in order)
  right: string;        // Right part (scrambled position)
}

/**
 * Player's answer for sentence pairing
 */
```

```
interface SentencePairingAnswer {
  pairId: string;       // ID of the pair being matched
  leftText: string;     // Left part text
  rightText: string;    // Player's chosen right part
}

// Example: Matching fruits to colors
// Card data sent to player:
{
  sentencePairs: [
    { id: "pair_0", left: "Apple", right: "Yellow" },      // Right parts
    { id: "pair_1", left: "Banana", right: "Orange" },     // are scrambled
    { id: "pair_2", left: "Orange", right: "Red" }
  ]
}

// Player's answer:
{
  answer: [
    { pairId: "pair_0", leftText: "Apple", rightText: "Red" },
    { pairId: "pair_1", leftText: "Banana", rightText: "Yellow" },
    { pairId: "pair_2", leftText: "Orange", rightText: "Orange" }
  ]
}

// Client → Server: Submit answer
socket.emit('game:card-answer', {
  gameCode: string;
  answer: any;  // Type depends on card type:
                // - QUIZ: string (A/B/C/D)
                // - SENTENCE_PAIRING: SentencePairingAnswer[] or string (legacy)
                // - OWN_ANSWER: string
                // - TRUE_FALSE: boolean or "true"/"false"
                // - CLOSER: number
});

// Server → All Players: Answer submitted (before validation)
socket.on('game:answer-submitted', {
  playerName: string;
  playerId: string;
```

```
  answer: any;
  message: string;
  timestamp: string;
});

// Server → All Players: Answer validated
socket.on('game:answer-validated', {
  playerName: string;
  playerId: string;
  isCorrect: boolean;
  correctAnswer: any;
  message: string;
  timestamp: string;
});
```

Position Guessing (Question Cards)

```
// Server → Player: Request position guess
socket.on('game:position-guess-request', {
  message: string;
  currentPosition: number;       // Starting position
  diceRoll: number;              // Original dice value
  fieldStepValue: number;        // Field's step value
  patternModifier: number;       // Zone-based modifier
  timeLimit: number;             // 30 seconds
  timestamp: string;
});

// Client → Server: Submit position guess
socket.emit('game:position-guess', {
  gameCode: string;
  guessedPosition: number;
});

// Server → All Players: Player is guessing (notification)
socket.on('game:player-guessing', {
  playerId: string;
  playerName: string;
  message: string;
  timestamp: string;
});

// Server → All Players: Player's guess broadcast
socket.on('game:position-guess-broadcast', {
  playerId: string;
  playerName: string;
  guessedPosition: number;
  message: string;
  timestamp: string;
});

// Server → All Players: Guess result with full calculation
socket.on('game:guess-result', {
  playerId: string;
  playerName: string;
  guessedPosition: number;
```

```
  actualPosition: number;        // Calculated position
  finalPosition: number;         // After penalty (if wrong)
  guessCorrect: boolean;
  penaltyApplied: boolean;       // -2 if wrong
  calculation: {
    startPosition: number;
    diceRoll: number;
    stepValue: number;
    patternModifier: number;
    calculatedPosition: number;
    penalty: number;             // 0 or -2
  };
  message: string;
  timestamp: string;
});

// Server → All Players: Player didn't move
socket.on('game:no-movement', {
  playerId: string;
  playerName: string;
  reason: string;                // "Wrong answer on positive field"
  message: string;
  timestamp: string;
});

// Server → All Players: Penalty avoided
socket.on('game:penalty-avoided', {
  playerId: string;
  playerName: string;
  message: string;               // "Avoided penalty on negative field"
  timestamp: string;
});
```

Luck Cards

```
// Server → All Players: Luck card consequence applied
socket.on('game:luck-consequence', {
  playerId: string;
  playerName: string;
  consequenceType: string;       // "MOVE_FORWARD" | "MOVE_BACKWARD" | etc.
  value: number;                 // Magnitude of effect
  newPosition?: number;          // For movement consequences
  turnsToLose?: number;          // For LOSE_TURN
  extraTurns?: number;           // For EXTRA_TURN
  message: string;
  timestamp: string;
});
```

Consequence Types:

- `MOVE_FORWARD` (0): Move forward X steps
- `MOVE_BACKWARD` (1): Move backward X steps
- `LOSE_TURN` (2): Lose X turns
- `EXTRA_TURN` (3): Gain X extra turns
- `GO_TO_START` (5): Return to position 1

Joker Cards & Gamemaster Decisions

```
// Server → All Players: Joker card drawn
socket.on('game:joker-drawn', {
  playerName: string;
  playerId: string;
  jokerCard: {
    question: string;
    consequence: Consequence;
  };
  waitingForGamemaster: boolean;
  timestamp: string;
});

// Server → Gamemaster: Decision request
socket.on('game:gamemaster-decision-request', {
  requestId: string;
  playerName: string;
  playerId: string;
  jokerCard: {
    question: string;
    consequence: Consequence;
  };
  timeLimit: number;              // 120 seconds
  timestamp: string;
});

// Client → Server: Gamemaster decision
socket.emit('game:gamemaster-decision', {
  gameCode: string;
  requestId: string;
  decision: 'approve' | 'reject';
});

// Server → All Players: Decision result
socket.on('game:gamemaster-decision-result', {
  playerName: string;
  playerId: string;
  gamemasterName: string;
  decision: string;               // "approve" | "reject"
  approved: boolean;
```

```
  consequence: number | null;
  description: string;
  timestamp: string;
});

// Server → Player: Joker position guess request
socket.on('game:joker-position-guess-request', {
  message: string;
  currentPosition: number;
  diceRoll: number;              // Always 6 for jokers
  fieldStepValue: number;
  patternModifier: number;
  timeLimit: number;             // 30 seconds
  timestamp: string;
});

// Client → Server: Joker position guess
socket.emit('game:joker-position-guess', {
  gameCode: string;
  guessedPosition: number;
});

// Server → All Players: Joker card complete
socket.on('game:joker-complete', {
  playerId: string;
  playerName: string;
  guessedPosition: number;
  actualPosition: number;
  finalPosition: number;         // Current position if didn't move
  guessCorrect: boolean;
  penaltyApplied: boolean;
  moved: boolean;                // Based on field type + decision
  calculation: {
    startPosition: number;
    diceRoll: number;            // 6
    stepValue: number;
    patternModifier: number;
    calculatedPosition: number;
    penalty: number;
  };
```

```
  message: string;
  timestamp: string;
});
```

Joker Flow Logic:

- **Positive field + approved**: Player moves (with guess)
- **Positive field + rejected**: Player doesn't move
- **Negative field + approved**: Player doesn't move (avoided penalty)
- **Negative field + rejected**: Player moves (with guess - penalty)

Turn Advancement & Turn Tracking

```
// Server → All Players: Turn changed
socket.on('game:turn-changed', {
  currentPlayer: string;         // Player ID
  currentPlayerName: string;
  turnNumber: number;
  message: string;
  timestamp: string;
});

// Server → All Players: Extra turn remaining
socket.on('game:extra-turn-remaining', {
  playerId: string;
  playerName: string;
  remainingExtraTurns: number;   // How many left after this one
  message: string;
  timestamp: string;
});

// Server → All Players: Players skipped
socket.on('game:players-skipped', {
  skippedPlayers: Array<{
    playerId: string;
    playerName: string;
    remainingTurnsToLose: number;
  }>;
  message: string;
  timestamp: string;
});
```

Game End & Cleanup

```
// Server → All Players: Game ended
socket.on('game:ended', {
  winner: string;                // Player ID
  winnerName: string;
  message: string;
  finalPositions: PlayerPosition[];
  timestamp: string;
});

// Server → All Players: Cleanup complete
socket.on('game:cleanup-complete', {
  gameCode: string;
  message: string;
  timestamp: string;
});
```

Error Events

```
// Server → Client: Error occurred
socket.on('game:error', {
  message: string;
  code?: string;
  timestamp: string;
});

// Server → All Players: Card error (no cards available)
socket.on('game:card-error', {
  playerName: string;
  playerId: string;
  error: string;
  timestamp: string;
});

// Server → All Players: Joker error
socket.on('game:joker-error', {
  playerName: string;
  playerId: string;
  error: string;
  timestamp: string;
});
```

Interfaces & Data Structures

WebSocket Interfaces

Located in: `src/Application/Services/Interfaces/GameInterfaces.ts`

```typescript
/**
 * Join game room data
 */
export interface JoinGameData {
  gameToken: string;  // JWT token from REST /join endpoint
}

/**
 * Leave game data
 */
export interface LeaveGameData {
  gameCode: string;
}

/**
 * Dice roll data
 */
export interface DiceRollData {
  gameCode: string;
  diceValue: number;  // 1-6
}

/**
 * Player position tracking
 */
export interface PlayerPosition {
  playerId: string;
  playerName: string;
  boardPosition: number;
  turnOrder: number;
}

/**
 * Game chat message
 */
export interface GameChatData {
  gameCode: string;
  message: string;
}
```

```typescript
/**
 * Card answer data
 */
export interface CardAnswerData {
  gameCode: string;
  answer: any;  // Type depends on card type
}

/**
 * Gamemaster decision data
 */
export interface GamemasterDecisionData {
  gameCode: string;
  requestId: string;
  decision: 'approve' | 'reject';
}

/**
 * Field effect calculation request
 */
export interface FieldEffectRequest {
  gameId: string;
  playerId: string;
  playerName: string;
  currentPosition: number;
  card: GameCard;
  field: GameField;
  dice: number;
  guessedPosition?: number;
}

/**
 * Field effect calculation result
 */
export interface FieldEffectResult {
  finalPosition: number;
  stepValue: number;
  dice: number;
  patternModifier: number;
```

```
  consequenceModifier: number;
  guessResult?: GuessResult;
  gamemasterResult?: GamemasterDecisionResult;
  description: string;
  effects: string[];
  turnEffect?: TurnEffect;
}

/**
 * Turn effect (for multi-turn tracking)
 */
export interface TurnEffect {
  type: 'LOSE_TURN' | 'EXTRA_TURN';
  playerId: string;
  value: number;  // Number of turns
}

/**
 * Guess result details
 */
export interface GuessResult {
  guessedPosition: number;
  actualPosition: number;
  isCorrect: boolean;
  penaltyApplied: boolean;
  description: string;
}
```

Pending State Interfaces

Located in: `src/Application/Services/GameWebSocketService.ts`

```typescript
/**
 * Pending card state (stored in Redis)
 * Used for question cards requiring answers
 */
interface PendingCardState {
  playerId: string;
  playerName: string;
  card: GameCard;
  field: GameField;              // Field where card was drawn
  dice: number;                  // Original dice roll
  currentPosition: number;       // Position before card effect
  drawnAt: number;               // Timestamp
  answerGiven?: boolean;         // Has player answered?
  answerCorrect?: boolean;       // Was answer correct?
  requiresGuess?: boolean;       // Does this require position guess?
  guessedPosition?: number;      // Player's guessed position
}

/**
 * Pending gamemaster decision state (stored in Redis)
 * Used for joker cards
 */
interface PendingDecisionState {
  playerId: string;
  playerName: string;
  card: GameCard;
  field: GameField;              // Field where joker was drawn
  dice: number;                  // Always 6 for jokers
  currentPosition: number;       // Position before joker effect
  drawnAt: number;               // Timestamp
  recursionDepth: number;        // Prevent infinite secondary landings
  gamemasterDecided?: boolean;   // Has gamemaster decided?
  gamemasterApproved?: boolean;  // Was it approved?
  guessedPosition?: number;      // Player's guessed position (if required)
}
```

Redis Keys:

```
pending_card:{gameCode}:{playerId}              → PendingCardState (TTL: 60s)
pending_decision:{gameCode}:{requestId}         → PendingDecisionState (TTL: 120s)
player_extra_turns:{gameCode}:{playerId}        → number (extra turns remaining)
player_turns_to_lose:{gameCode}:{playerId}      → number (turns to skip)
```

Domain Aggregates

GameAggregate

Located in: `src/Domain/Game/GameAggregate.ts`

```
/**
 * Main game entity
 */
@Entity('Games')
export class GameAggregate {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'varchar', length: 10, unique: true })
  gamecode: string;  // 6-character unique code

  @Column({ type: 'int' })
  maxplayers: number;

  @Column({ type: 'int', default: LoginType.PUBLIC })
  logintype: LoginType;  // 0=PUBLIC, 1=PRIVATE, 2=ORGANIZATION

  @Column({ type: 'int', default: 50 })
  boardsize: number;

  @Column({ type: 'uuid', nullable: false, name: 'createdBy' })
  createdby: string;  // Gamemaster user ID

  @Column({ type: 'uuid', nullable: true, name: 'organizationid' })
  orgid: string | null;

  @Column({ type: 'jsonb', default: () => "'[]'", name: 'decks' })
  gamedecks: GameDeck[];  // Decks with cards

  @Column({ type: 'simple-array', default: '' })
  players: string[];  // Array of player IDs/names

  @Column({ type: 'timestamp', nullable: true })
  startdate: Date | null;

  @Column({ type: 'timestamp', nullable: true })
  enddate: Date | null;

  @Column({ type: 'boolean', default: false })
```

```
  finished: boolean;

  @Column({ type: 'varchar', nullable: true })
  winner: string | null;

  @Column({ type: 'int', default: 100 })
  totaltiles: number;
}

/**
 * Login type enum
 */
export enum LoginType {
  PUBLIC = 0,        // Anyone can join with playerName
  PRIVATE = 1,       // Requires authentication
  ORGANIZATION = 2   // Requires auth + org membership
}
```

GameField

```typescript
/**
 * Individual board field/tile
 */
export interface GameField {
  position: number;        // 0-100
  type: 'regular' | 'positive' | 'negative' | 'luck';
  stepValue?: number;      // 0-3, used in movement calculation
}

/**
 * Complete board data
 */
export interface BoardData {
  gameId?: string;
  fields: GameField[];     // 100 fields
  generationComplete?: boolean;
  generatedAt?: Date;
  error?: string;
}
```

Field Type Distribution (generated by `BoardGenerationService`):

- **Regular**: ~40% (no special card)
- **Positive**: ~25% (reward card)
- **Negative**: ~25% (penalty card)
- **Luck**: ~10% (instant consequence)

Important: Fields have NO consequences. Consequences come only from **cards**.

GameCard

```
/**
 * Card in a deck
 */
export interface GameCard {
  cardid: string;
  question?: string;
  answer?: any;                // Can be string, number, object, array
  type?: CardType;
  consequence?: Consequence | null;  // Only for LUCK and JOKER cards
  played?: boolean;
  playerid?: string;
}

/**
 * Card types
 */
export enum CardType {
  QUIZ = 0,              // Multiple choice (A/B/C/D)
  SENTENCE_PAIRING = 1,  // Match left parts to right parts
  OWN_ANSWER = 2,        // Free text answer
  TRUE_FALSE = 3,        // Boolean answer
  CLOSER = 4,            // Closest number answer
  JOKER = 5,             // Gamemaster decision (secondary landing only)
  LUCK = 6               // Instant consequence
}
```

Card Answer Formats by Type:

Card Type	Answer Format in Database	Player Answer
QUIZ	`[{answer:"A", text:"...", correct:true}, ...]`	`"A"` (string)
SENTENCE_PAIRING	`[{left:"Apple", right:"Red"}, ...]`	`[{pairId, leftText, ri`

Card Type	Answer Format in Database	Player Answer
SENTENCE_PAIRING (legacy)	`"word1 word2 word3"` (string)	`["word1", "word2", "wo` `"word1 word2 word3"`
OWN_ANSWER	`["answer1", "answer2", ...]` (string[])	`"player answer"` (string
TRUE_FALSE	`true` or `false` (boolean)	`true/false` or `"true"/'`
CLOSER	`{correct: 42, percent: 10}` (object)	`40` (number)
JOKER	Not applicable	Not applicable
LUCK	Not applicable	Not applicable

Consequence

Located in: `src/Domain/Deck/DeckAggregate.ts`

```typescript
/**
 * Card consequence structure
 */
export interface Consequence {
  type: ConsequenceType;
  value?: number;  // Magnitude of effect (default: 1)
}

/**
 * Consequence types
 */
export enum ConsequenceType {
  MOVE_FORWARD = 0,    // Move forward X steps
  MOVE_BACKWARD = 1,   // Move backward X steps
  LOSE_TURN = 2,       // Skip X turns
  EXTRA_TURN = 3,      // Get X extra turns
  GO_TO_START = 5      // Return to position 1
}
```

Multi-Turn Support:

- `LOSE_TURN` with `value=3`: Player skips next 3 turns
- `EXTRA_TURN` with `value=2`: Player gets 2 additional turns after current

GameDeck

```
/**
 * Deck of cards
 */
export interface GameDeck {
  deckid: string;
  decktype: DeckType;
  cards: GameCard[];
}

/**
 * Deck types
 */
export enum DeckType {
  POSITIVE = 0,   // For positive fields
  NEGATIVE = 1,   // For negative fields
  LUCK = 2,       // For luck fields
  JOKER = 3       // For secondary landings
}
```

Complete Game Scenarios

Scenario 1: Question Card on Positive Field (Correct Answer)

```
1. Player at position 20, rolls dice = 4
2. Field at position 24 has stepValue = 2, type = "positive"
3. Player lands on position 24
4. Server waits 2 seconds (frontend animation)
5. Server draws POSITIVE deck card (type = QUIZ)
6. Server broadcasts: game:card-drawn (everyone sees question)
7. Server emits: game:card-drawn-self to player (60s timer)
8. Player answers correctly
9. Server broadcasts: game:answer-submitted
10. Server broadcasts: game:answer-validated { isCorrect: true }
11. Server determines: positive field + correct = GUESS REQUIRED
12. Server emits: game:position-guess-request to player (30s timer)
    - Shows: currentPosition=20, dice=4, stepValue=2, patternModifier=+2
13. Player calculates: 20 + 4 + 2 + 2 = 28, guesses 28
14. Server broadcasts: game:position-guess-broadcast { guessedPosition: 28 }
15. Server calculates: 20 + 4 + 2 + 2 = 28 (correct!)
16. Server broadcasts: game:guess-result {
    guessCorrect: true,
    finalPosition: 28,
    penaltyApplied: false
}
17. Player position updated to 28
18. Server checks: position 28 is not special field
19. Server calls: advanceTurn() → next player's turn
```

Scenario 2: Question Card on Negative Field (Wrong Answer)

```
1. Player at position 50, rolls dice = 5
2. Field at position 55 has stepValue = 1, type = "negative"
3. Player lands on position 55
4. Server waits 2 seconds
5. Server draws NEGATIVE deck card (type = TRUE_FALSE)
6. Server broadcasts: game:card-drawn
7. Server emits: game:card-drawn-self to player
8. Player answers incorrectly
9. Server broadcasts: game:answer-validated { isCorrect: false }
10. Server determines: negative field + wrong = GUESS REQUIRED (penalty test)
11. Server emits: game:position-guess-request
    - Shows: currentPosition=50, dice=5, stepValue=1, patternModifier=+1
12. Player guesses: 57, but actual is: 50 + 5 + 1 + 1 = 57
13. Server calculates actual: 57, player guessed: 57 (CORRECT!)
14. Server broadcasts: game:guess-result {
    guessCorrect: true,
    finalPosition: 57,
    penaltyApplied: false
}
15. Player moves to 57 (avoided penalty despite wrong answer)
16. Server advances turn
```

Scenario 3: Luck Card with Multi-Turn EXTRA_TURN

```
1. Player at position 30, rolls dice = 3
2. Field at position 33 has type = "luck"
3. Player lands on position 33
4. Server draws LUCK deck card
5. Card has: { type: EXTRA_TURN (3), value: 3 }
6. Server broadcasts: game:card-result
7. Server broadcasts: game:luck-consequence {
    consequenceType: "EXTRA_TURN",
    value: 3,
    extraTurns: 3
}
8. Server calls: setPlayerExtraTurns(gameCode, playerId, 3)
   - Redis: player_extra_turns:{gameCode}:{playerId} = 3
9. Server calls: advanceTurn()
10. advanceTurn() checks: getPlayerExtraTurns() = 3 > 0
11. Server emits: game:extra-turn-remaining { remainingExtraTurns: 2 }
12. Same player continues (no turn advance)
13. Player rolls again...
14. After turn ends, advanceTurn() checks again: extraTurns = 2 > 0
15. Process repeats until extraTurns = 0
16. Player gets 4 total turns (1 original + 3 extra)
```

Scenario 4: Luck Card with Multi-Turn LOSE_TURN

```
1. Player A draws luck card: { type: LOSE_TURN (2), value: 2 }
2. Server calls: setPlayerTurnsToLose(gameCode, playerA, 2)
   - Redis: player_turns_to_lose:{gameCode}:{playerA} = 2
3. Server broadcasts: game:luck-consequence {
    consequenceType: "LOSE_TURN",
    turnsToLose: 2
}
4. Server calls: advanceTurn() → skip to Player B

--- Later, when it's Player A's turn in sequence ---

5. advanceTurn() reaches Player A in turn sequence
6. Checks: getPlayerTurnsToLose(playerA) = 2 > 0
7. Decrements: 2 → 1
8. Adds to skippedPlayers array
9. Continues to next player in sequence (Player B)
10. Server broadcasts: game:players-skipped {
    skippedPlayers: [{
      playerName: "Player A",
      remainingTurnsToLose: 1
    }]
}
11. Server broadcasts: game:turn-changed { currentPlayer: "Player B" }

--- Next round ---

12. advanceTurn() reaches Player A again
13. Checks: getPlayerTurnsToLose(playerA) = 1 > 0
14. Decrements: 1 → 0, deletes Redis key
15. Skips Player A again
16. Player A skipped 2 times total
```

Scenario 5: Joker Card on Positive Field (Approved by Gamemaster)

```
1. Player completes question card, lands on position 70 (positive field)
2. Server checks secondary landing
3. Server draws JOKER deck card
4. Server broadcasts: game:joker-drawn { waitingForGamemaster: true }
5. Server emits to gamemaster: game:gamemaster-decision-request (120s timer)
6. Gamemaster approves
7. Server broadcasts: game:gamemaster-decision-result {
    decision: "approve",
    approved: true
}
8. Server determines: positive field + approved = GUESS REQUIRED
9. Server emits: game:joker-position-guess-request
   - Shows: currentPosition=70, dice=6, stepValue=1, patternModifier=-2
10. Player guesses: 75, actual: 70 + 6 + 1 - 2 = 75 (CORRECT!)
11. Server broadcasts: game:joker-complete {
    guessCorrect: true,
    moved: true,
    finalPosition: 75
}
12. Player moves to 75
13. Server checks for tertiary landing (not allowed - recursion depth)
14. Server advances turn
```

Scenario 6: Multiple Players with Turn Tracking

```
Turn Sequence: [PlayerA, PlayerB, PlayerC]

--- Turn 1 ---
PlayerA: Gets EXTRA_TURN (value=2)
- Redis: player_extra_turns:game:PlayerA = 2
- PlayerA plays again

--- Turn 2 (PlayerA extra turn 1) ---
PlayerA: Gets LOSE_TURN (value=1)
- Redis: player_turns_to_lose:game:PlayerA = 1
- Redis: player_extra_turns:game:PlayerA = 1 (still has 1 left)
- advanceTurn() checks: extraTurns = 1 > 0
- PlayerA plays AGAIN (extra turn takes priority)

--- Turn 3 (PlayerA extra turn 2) ---
PlayerA: Normal turn
- Redis: player_extra_turns:game:PlayerA = 0 (deleted)
- advanceTurn() → next player

--- Turn 4 ---
Sequence reaches PlayerA
- Checks: turnsToLose = 1 > 0
- Skip PlayerA, decrement to 0
- Server broadcasts: game:players-skipped
- PlayerB plays

--- Turn 5 ---
PlayerB: Normal turn

--- Turn 6 ---
PlayerC: Normal turn

--- Turn 7 ---
Back to PlayerA (turnsToLose = 0)
- PlayerA plays normally
```

Turn Tracking System

Redis Storage

```
// Extra turns storage
player_extra_turns:{gameCode}:{playerId} → "3"  // Number as string
TTL: None (cleared on cleanup or use)

// Turns to lose storage
player_turns_to_lose:{gameCode}:{playerId} → "2"  // Number as string
TTL: None (cleared on cleanup or use)
```

Turn Tracking Methods

```
class GameWebSocketService {
  // Extra turns
  private async setPlayerExtraTurns(
    gameCode: string,
    playerId: string,
    count: number
  ): Promise<void>;

  private async getPlayerExtraTurns(
    gameCode: string,
    playerId: string
  ): Promise<number>;  // Returns 0 if not found

  private async decrementPlayerExtraTurns(
    gameCode: string,
    playerId: string
  ): Promise<void>;  // Decrements or deletes if reaches 0

  // Turns to lose
  private async setPlayerTurnsToLose(
    gameCode: string,
    playerId: string,
    count: number
  ): Promise<void>;

  private async getPlayerTurnsToLose(
    gameCode: string,
    playerId: string
  ): Promise<number>;  // Returns 0 if not found

  private async decrementPlayerTurnsToLose(
    gameCode: string,
    playerId: string
  ): Promise<void>;  // Decrements or deletes if reaches 0

  // Cleanup
  private async clearPlayerTurnData(
    gameCode: string,
    playerId: string
```

```
  ): Promise<void>;  // Deletes both keys
}
```

Enhanced advanceTurn() Logic

```
private async advanceTurn(gameCode: string): Promise<void> {
  // PHASE 1: Check if current player has extra turns
  const extraTurns = await this.getPlayerExtraTurns(gameCode, currentPlayerId);
  if (extraTurns > 0) {
    await this.decrementPlayerExtraTurns(gameCode, currentPlayerId);
    emit('game:extra-turn-remaining', { remainingExtraTurns: extraTurns - 1 });
    emit('game:your-turn', { isExtraTurn: true });
    return;  // Same player continues
  }

  // PHASE 2: Find next player, skipping those with lost turns
  let nextTurnIndex = (currentTurnIndex + 1) % turnSequence.length;
  const skippedPlayers = [];
  let loopGuard = 0;

  while (loopGuard < turnSequence.length) {
    const candidatePlayerId = turnSequence[nextTurnIndex];
    const turnsToLose = await this.getPlayerTurnsToLose(gameCode, candidatePlayerId);

    if (turnsToLose > 0) {
      await this.decrementPlayerTurnsToLose(gameCode, candidatePlayerId);
      skippedPlayers.push({
        playerId: candidatePlayerId,
        remainingTurnsToLose: turnsToLose - 1
      });
      nextTurnIndex = (nextTurnIndex + 1) % turnSequence.length;
      loopGuard++;
    } else {
      break;  // Found valid player
    }
  }

  // PHASE 3: Update game state
  gameState.currentTurn = nextTurnIndex;
  gameState.currentPlayer = turnSequence[nextTurnIndex];

  // PHASE 4: Notify about skipped players
  if (skippedPlayers.length > 0) {
    emit('game:players-skipped', { skippedPlayers });
```

```
  }

  // PHASE 5: Notify about turn change
  emit('game:turn-changed', { currentPlayer, currentPlayerName });
}
```

Position Guessing Mechanic

Pattern-Based Movement Calculation

Formula:

```
finalPosition = currentPosition + (stepValue × dice) + patternModifier
```

Pattern Modifiers by Position & Field Type:

```
private getPatternModifier(position: number, positiveField: boolean): number {
  // Dynamic pattern-based modifiers for engaging gameplay
  // Sign depends on field type: positive field = positive modifier, negative field = negative mod

  if (position % 10 === 0) {
    return 0;  // Positions ending in 0 (10, 20, 30...) - No modifier
  } else if (position % 10 === 5) {
    return positiveField ? 3 : -3;  // Positions ending in 5 (15, 25, 35...) - ±3 modifier
  } else if (position % 3 === 0) {
    return positiveField ? 2 : -2;  // Positions divisible by 3 (9, 12, 21...) - ±2 modifier
  } else if (position % 2 === 1) {
    return positiveField ? 1 : -1;  // Odd positions (1, 7, 11...) - ±1 modifier
  } else {
    return 0;  // Other even positions - No modifier
  }
}
```

How Field Type is Determined:

- `positiveField = true` when `stepValue > 0` (positive field)
- `positiveField = false` when `stepValue < 0` (negative field)

Why This Design:

- **Dynamic**: Every position has different calculation rules based on patterns
- **Learnable**: Players can recognize patterns (ends in 5, divisible by 3, etc.)
- **Field-Dependent**: Positive fields give positive modifiers, negative fields give negative modifiers
- **Skill-Based**: Requires mental calculation and pattern recognition under time pressure (30s)
- **Not Trivial**: Information is available but requires active processing

Guess Requirement Logic

```
private determineGuessRequirement(
  fieldType: 'regular' | 'positive' | 'negative' | 'luck',
  answerCorrect: boolean
): boolean {
  if (fieldType === 'positive') {
    return answerCorrect;  // Correct = guess for REWARD
  } else if (fieldType === 'negative') {
    return !answerCorrect;  // Wrong = guess for PENALTY
  }
  return false;  // Regular and luck fields never require guess
}
```

Question Card Guess Flow

Matrix:

Field Type	Answer	Guess Required?	Reason
Positive	Correct	**YES**	Reward scenario
Positive	Wrong	NO	No movement
Negative	Correct	NO	Avoided penalty

Field Type	Answer	Guess Required?	Reason
Negative	Wrong	**YES**	Penalty test
Regular	Any	NO	No special fields
Luck	N/A	NO	Instant consequence

Joker Card Guess Flow

Matrix (dice always = 6):

Field Type	Gamemaster	Guess Required?	Movement
Positive	Approved	**YES**	Move if guess
Positive	Rejected	NO	No movement
Negative	Approved	NO	No movement (avoid)
Negative	Rejected	**YES**	Move if guess

Penalty System

- **Wrong guess**: -2 steps from calculated position
- **Minimum position**: 1 (can't go below start)
- **Applied after calculation**: `finalPosition = max(1, calculatedPosition - 2)`

Calculation Examples

Example 1: Position 15 (ends in 5), positive field, dice 4, stepValue 2

```
positiveField = true (stepValue 2 > 0)
patternModifier = 3 (position ends in 5, positive field)
calculation = 15 + (2 × 4) + 3 = 15 + 8 + 3 = 26
```

Example 2: Position 35 (ends in 5), negative field, dice 6, stepValue -1

```
positiveField = false (stepValue -1 < 0)
patternModifier = -3 (position ends in 5, negative field)
calculation = 35 + (-1 × 6) + (-3) = 35 - 6 - 3 = 26
```

Example 3: Position 21 (divisible by 3), positive field, dice 5, stepValue 2

```
positiveField = true (stepValue 2 > 0)
patternModifier = 2 (position divisible by 3, positive field)
calculation = 21 + (2 × 5) + 2 = 21 + 10 + 2 = 33
```

Example 4: Position 20 (ends in 0), any field type, dice 4, stepValue 2

```
patternModifier = 0 (position ends in 0, always 0)
calculation = 20 + (2 × 4) + 0 = 20 + 8 = 28
```

Example 5: Position 7 (odd), negative field, dice 3, stepValue -2

```
positiveField = false (stepValue -2 < 0)
patternModifier = -1 (position is odd, negative field)
calculation = 7 + (-2 × 3) + (-1) = 7 - 6 - 1 = 0 → clamped to 1
```

Error Handling

REST API Error Responses

Format:

```
{
  error: string;  // Human-readable error message
  code?: string;  // Optional error code
}
```

Status Codes:

- `400` : Bad Request (validation errors, missing fields)
- `401` : Unauthorized (authentication required)
- `403` : Forbidden (insufficient permissions, org mismatch)
- `404` : Not Found (game, deck, user not found)
- `409` : Conflict (game full, already started, player already in game)
- `500` : Internal Server Error

WebSocket Error Events

```
socket.on('game:error', {
  message: string;
  code?: string;
  timestamp: string;
});
```

Common Errors:

- Authentication failed
- Invalid game code
- Game already started
- Not your turn
- Invalid answer format
- Timeout expired
- No cards available
- Internal server error

Timeout Handling

Card Answer Timeout (60 seconds):

```
// Server auto-processes as wrong answer
socket.on('game:card-timeout', {
  playerName: string;
  message: string;
  timestamp: string;
});
```

Gamemaster Decision Timeout (120 seconds):

```
// Server auto-rejects joker card
socket.on('game:gamemaster-timeout', {
  requestId: string;
  playerName: string;
  message: string;
  timestamp: string;
});
```

Position Guess Timeout (30 seconds):

```
// Server treats as no movement
socket.on('game:guess-timeout', {
  playerName: string;
  message: string;
  timestamp: string;
});
```

Redis Cleanup

Automatic Cleanup:

- Game end: All keys deleted
- Player disconnect: Pending states cleared
- Timeout: Keys expire naturally (TTL)

Manual Cleanup:

```
// Called on game:ended or disconnect
await cleanupGameData(gameCode, gameId);
```

Keys Cleaned:

```
gameplay:{gameCode}
game_state:{gameCode}
game_board_{gameCode}
game_connections:{gameCode}
game_ready:{gameCode}
game_pending:{gameCode}
game_positions:{gameCode}
pending_card:{gameCode}:{playerId}
pending_decision:{gameCode}:{requestId}
player_extra_turns:{gameCode}:{playerId}
player_turns_to_lose:{gameCode}:{playerId}
```

Card Type Implementation Details

SENTENCE_PAIRING Card Type

Purpose: Test player's ability to match related items (left parts to right parts)

Use Cases:

- Match words to definitions
- Match questions to answers
- Match countries to capitals
- Match names to descriptions
- Match terms to translations

Database Format (NEW)

```
// In DeckAggregate Card.answer field
{
  text: "Match each fruit to its color",
  type: CardType.SENTENCE_PAIRING,  // 1
  answer: [
    { left: "Apple", right: "Red" },
    { left: "Banana", right: "Yellow" },
    { left: "Orange", right: "Orange" },
    { left: "Grape", right: "Purple" }
  ]
}
```

Client Preparation

Server scrambles the **right parts** while keeping left parts in order:

```
// Sent to client in game:card-drawn-self
{
  cardData: {
    question: "Match each fruit to its color",
    type: 1,  // SENTENCE_PAIRING
    sentencePairs: [
      { id: "pair_0", left: "Apple", right: "Purple" },   // Scrambled!
      { id: "pair_1", left: "Banana", right: "Orange" },
      { id: "pair_2", left: "Orange", right: "Red" },
      { id: "pair_3", left: "Grape", right: "Yellow" }
    ],
    timeLimit: 60
  }
}
```

Player Answer Format

```
// Player submits array of matches
{
  answer: [
    { pairId: "pair_0", leftText: "Apple", rightText: "Red" },
    { pairId: "pair_1", leftText: "Banana", rightText: "Yellow" },
    { pairId: "pair_2", leftText: "Orange", rightText: "Orange" },
    { pairId: "pair_3", leftText: "Grape", rightText: "Purple" }
  ]
}
```

Validation Logic

```
// For each correct pair in database:
1. Find player's match for the left part
2. Compare player's chosen right part to correct right part
3. Case-insensitive comparison with trimmed whitespace
4. ALL pairs must match correctly for answer to be correct

// Example validation result:
{
  isCorrect: true,  // Only if ALL pairs match
  submittedAnswer: [...],
  correctAnswer: [...],
  explanation: "✅ Perfect! All 4 pairs matched correctly!
    ✓ \"Apple\" → \"Red\"
    ✓ \"Banana\" → \"Yellow\"
    ✓ \"Orange\" → \"Orange\"
    ✓ \"Grape\" → \"Purple\""
}
```

Partial Match Example

```
// Player gets 2 out of 4 correct:
{
  isCorrect: false,
  explanation: "❌ Only 2/4 pairs correct:
    ✓ \"Apple\" → \"Red\"
    ✗ \"Banana\" → \"Purple\" (should be \"Yellow\")
    ✗ \"Orange\" → \"Yellow\" (should be \"Orange\")
    ✓ \"Grape\" → \"Purple\""
}
```

Legacy Format (Backward Compatibility)

Old Database Format:

```
{
  text: "The quick brown fox jumps over the lazy dog",
  type: CardType.SENTENCE_PAIRING,
  answer: "The quick brown fox jumps over the lazy dog"  // String
}
```

Client Preparation (Legacy):

```
{
  cardData: {
    question: "Arrange the words to form a sentence:",
    type: 1,
    words: ["lazy", "The", "dog", "fox", "over", "quick", "brown", "jumps", "the"],
    timeLimit: 60
  }
}
```

Player Answer (Legacy):

```
{
  answer: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
  // OR
  answer: "The quick brown fox jumps over the lazy dog"
}
```

Frontend Implementation Example

```tsx
// React/Vue component for SENTENCE_PAIRING
interface Props {
  sentencePairs: SentencePair[];
  onSubmit: (answer: SentencePairingAnswer[]) => void;
}

function SentencePairingCard({ sentencePairs, onSubmit }: Props) {
  const [matches, setMatches] = useState<Map<string, string>>(new Map());

  const handleMatch = (pairId: string, leftText: string, rightText: string) => {
    setMatches(prev => new Map(prev).set(pairId, rightText));
  };

  const handleSubmit = () => {
    const answer: SentencePairingAnswer[] = sentencePairs.map(pair => ({
      pairId: pair.id,
      leftText: pair.left,
      rightText: matches.get(pair.id) || ''
    }));
    onSubmit(answer);
  };

  return (
    <div>
      {sentencePairs.map(pair => (
        <div key={pair.id}>
          <span>{pair.left}</span>
          <select onChange={(e) => handleMatch(pair.id, pair.left, e.target.value)}>
            <option value="">-- Select --</option>
            {sentencePairs.map(p => (
              <option key={p.id} value={p.right}>{p.right}</option>
            ))}
          </select>
        </div>
      ))}
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

Database Migration

No migration needed! The implementation supports both formats:

1. **New cards**: Use array of `{left, right}` objects
2. **Existing cards**: Continue working with string format
3. Detection is automatic based on `answer` field type

Creating New Sentence Pairing Cards:

```
-- Example INSERT for new format
INSERT INTO "Cards" (deck_id, text, type, answer)
VALUES (
  'deck-uuid',
  'Match programming languages to their creators',
  1,  -- SENTENCE_PAIRING
  '[
    {"left": "Python", "right": "Guido van Rossum"},
    {"left": "JavaScript", "right": "Brendan Eich"},
    {"left": "C++", "right": "Bjarne Stroustrup"},
    {"left": "Ruby", "right": "Yukihiro Matsumoto"}
  ]'::jsonb
);
```

Appendix: Complete Event Summary

Client → Server Events

Event	Data	Descr
`game:join`	`{ gameToken: string }`	Join g room v auth to
`game:leave`	`{ gameCode: string }`	Leave volunt

Event	Data	Descr
`game:ready`	`{ gameCode: string, ready: boolean }`	Mark p as rea ready
`game:approve-player`	`{ gameCode: string, playerName: string }`	Game approv player (PRIV
`game:reject-player`	`{ gameCode: string, playerName: string, reason?: string }`	Game rejects player (PRIV
`game:join-approved`	`{ gameToken: string }`	Join a approv (PRIV
`game:chat`	`{ gameCode: string, message: string }`	Send messa
`game:action`	`{ gameCode: string, action: string, data?: any }`	Gener game
`game:dice-roll`	`{ gameCode: string, diceValue: number }`	Roll di (1-6)
`game:card-answer`	`{ gameCode: string, answer: any }`	Submi answe
`game:gamemaster-decision`	`{ gameCode: string, requestId: string, decision: string }`	Game decisi joker
`game:position-guess`	`{ gameCode: string, guessedPosition: number }`	Submi positi guess (quest

Event	Data	Descr
`game:joker-position-guess`	`{ gameCode: string, guessedPosition: number }`	Submi positic guess (joker)

Server → Client Events

Event	Audience	Description
`game:joined`	Individual	Successful join, joined rooms
`game:state`	Individual	Current game state sent
`game:pending-approval`	Individual	Waiting for gamemaster approval (PRIVATE)
`game:approval-granted`	Individual	Join request approved (PRIVATE)
`game:approval-denied`	Individual	Join request rejected (PRIVATE)
`game:player-joined`	All	Player joined game
`game:player-left`	All	Player left game
`game:player-disconnected`	All	Player disconnected unexpectedly
`game:player-disconnected-during-card`	All	Player disconnected during card answer
`game:player-requesting-join`	Gamemaster	Player wants to join (PRIVATE)
`game:player-approved`	All	Player was approved by gamemaster
`game:player-ready`	All	Player ready status changed

Event	Audience	Description
`game:all-ready`	All	All players ready
`game:chat-message`	All	Chat message from player
`game:action-result`	All	Generic action result
`game:state-update`	All	Game state updated
`game:started`	All	Game started, board generated
`game:turn-changed`	All	Turn advanced to next player
`game:your-turn`	Individual	Your turn notification
`game:dice-rolled`	All	Dice rolled by player
`game:player-moved`	All	Player moved to new position
`game:card-drawn`	All	Card drawn (question shown)
`game:card-drawn-self`	Individual	Interactive card data
`game:card-result`	All	Card result (for LUCK cards)
`game:card-timeout`	All	Player timed out on card answer
`game:answer-submitted`	All	Answer submitted (pre-validation)
`game:answer-validated`	All	Answer validation result
`game:position-guess-request`	Individual	Request position guess
`game:player-guessing`	All	Player is calculating guess
`game:position-guess-broadcast`	All	Player's guess shown
`game:guess-result`	All	Guess result with calculation
`game:no-movement`	All	Player didn't move

Event	Audience	Description
`game:penalty-avoided`	All	Player avoided penalty
`game:luck-consequence`	All	Luck card consequence applied
`game:joker-drawn`	All	Joker card drawn
`game:gamemaster-decision-request`	Gamemaster	Decision request
`game:gamemaster-decision-result`	All	Decision result
`game:gamemaster-timeout`	All	Gamemaster timed out on decision
`game:joker-position-guess-request`	Individual	Joker position guess request
`game:joker-complete`	All	Joker card processing complete
`game:joker-error`	All	Joker card error
`game:extra-turn-remaining`	All	Player using extra turn
`game:players-skipped`	All	Players skipped (lost turns)
`game:ended`	All	Game ended, winner declared
`game:cleanup-complete`	All	Cleanup finished
`game:error`	Individual	Error occurred
`game:card-error`	All	Card drawing error

End of Documentation