ChatGPT

Building a Real-Time 4-Player Rook Card Game with Kiro CLI

1. Dev Environment Setup

To build this project, set up the following tools and environment:

- **Node.js and npm:** Install Node.js (preferably LTS version 18+) which includes npm. This is required for both the React frontend and the Node.js backend.
- **Kiro CLI (or Cursor):** Install Kiro CLI (successor to the Amazon Q CLI) and log in with your AWS credentials. This will allow you to use AI prompts to generate code. Ensure you have it configured to use an appropriate model and that it can access your project files.
- **AWS Account & Credentials:** You'll be deploying AWS Lambda functions, API Gateway APIs, DynamoDB tables, etc. Set up AWS CLI with your credentials or ensure Kiro CLI can interface with AWS. Having Administrator or equivalent IAM permissions will simplify setup (you can tighten permissions later).
- **AWS Toolkit (optional):** Tools like AWS SAM CLI or the Serverless Framework can help with local testing and deployment. However, this guide will focus on using Kiro CLI to generate code and CloudFormation (via AWS SAM) for deployment.
- **Project Structure:** Create a project directory. We will have a frontend subdirectory (for the React app) and a backend subdirectory (for Lambda functions and deployment configuration). Initialize a Git repository to track changes if desired. You can use a monorepo or separate repos; this guide assumes a single repository for simplicity.
- **Local Testing:** You'll run the React app locally with `npm start` to test UI changes. For the backend, you can use local unit tests or AWS's web console to invoke functions, but an easier approach is to deploy a dev stage to AWS and test the live endpoints. We'll indicate how to test the HTTP API with `curl` and the WebSocket API with a client like **wscat** (a WebSocket CLI) once deployed.

With the environment ready, we can proceed to craft prompts for Kiro CLI to incrementally build the game.

2. Prompt Engineering Notes

When using Kiro CLI to generate the game, follow these best practices for prompt engineering:

- **One Task per Prompt:** Keep each prompt focused on a specific feature or task. This aligns with an iterative development process. The guide breaks down the game into small components (UI elements, backend functions, etc.) – tackle them one at a time. This makes it easier to identify and fix issues.
- **Be Specific and Descriptive:** Clearly state *what* you want to achieve and *how* if needed. Mention file names, function names, or frameworks to guide the AI. For example, "Create a React component for the game lobby with a form to enter player name" is better than "Make a UI for the game."

- **Provide Context:** Kiro CLI can use your repository context. Open or reference relevant files in the prompt if you want to modify them. For instance, if you want to update `App.js`, you can open that file in the CLI's context and instruct changes. This helps the AI understand the existing structure and avoid conflicts.
- **Iterate and Verify:** After each prompt, **review the code** that Kiro CLI produces. Run the frontend or a portion of the backend to verify it works. If something is not right, you can refine your prompt or manually adjust the code. Keeping prompts small helps reduce mistakes.
- **Use Comments for Clarification:** You can include comments in your prompt (or in code) to explain to the AI the purpose of certain sections. Kiro CLI will ignore code comments in context but they can serve as notes for you and hints for the AI's reasoning.
- **Testing After Each Step:** This guide includes a **Testing** section after each prompt. It's crucial to actually perform these tests (running the app, calling an API, etc.) to confirm the feature works as expected. This will catch issues early.
- **Ask for Help if Stuck:** If Kiro CLI's response is unclear or the code doesn't work, you can prompt it to fix errors or ask for clarification. For example, "The Lambda function returned a 502 error, can you help debug why?" Kiro can then assist in troubleshooting.
- **Keep a Consistent Style:** By using one AI agent (Kiro CLI) throughout, the code style should remain consistent. If you have a particular coding style (for example, using functional React components or a certain project structure), mention it early so the AI adheres to it.
- **Save Progress:** Use version control or Kiro CLI's save features. This way you can rollback if a prompt introduces issues. It's easier to discard or tweak the latest changes than to generate everything in one go.

With these guidelines in mind, let's start building the game step-by-step, writing prompts for Kiro CLI and explaining the expected outcomes.

3. Step-by-Step Implementation with Kiro CLI

Step 1: Initialize React Project and Basic Routing

First, set up the frontend as a React application and ensure we have a basic routing structure for the app pages (Lobby and Game table). We also want the app to be mobile-friendly and PWA-ready.

Prompt:

```
Use **create-react-app** (with TypeScript) to initialize a new React project in
a `frontend` folder named "rook-multiplayer-game". Include PWA support (service
worker and manifest for a mobile-friendly progressive web app). Set up React
Router with two routes:
1. **"/" (Home/Lobby)** – a page where the user can enter a name and either
create a new game or join an existing game by code. This can be a component
`LobbyPage`. It should have an input for the player’s name, a button "Create
Game", and a field + button to join by game code. No actual functionality yet,
just UI.
2. **"/game" (Game Table)** – a page for the game interface. Create a
placeholder component `GamePage` that will later show the game state (players,
```

```
cards, etc.). For now, it can just display a message like "Game Room" and a back
link to lobby.

Use a responsive, mobile-first design: e.g., include a viewport meta tag and
simple CSS to ensure inputs and buttons are easily tappable on mobile. You can
use plain CSS or a lightweight styling solution, but keep it simple (no heavy
component library yet).

After creating the app and components, register a service worker for PWA (use
CRA's default service worker or Workbox). Also add a basic `manifest.json` with
app name "Rook Online" and theme colors. Ensure the app builds and the routes
work (you can use React Router’s BrowserRouter).
```

Explanation: The above prompt will have Kiro CLI create a new React application using Create React App (with the TypeScript template for type safety). It sets up two main pages: a **LobbyPage** for the home route and a **GamePage** for the game route, using React Router for navigation. We specified PWA features: CRA’s default template will include a service worker registration and a `manifest.json` (we might need to tweak manifest and icons manually, but the prompt hints should handle it).

The LobbyPage will contain form elements for entering a player name and game code, and buttons to create or join a game. At this stage, these buttons won’t have functionality (we will wire them up in later steps to our backend). We are focusing on the UI structure and navigation. The GamePage is a placeholder that will eventually show the gameplay (player hands, table, etc.).

After running this prompt, you should have a new React project with a skeleton app. The `App.tsx` (or `App.js`) will likely use `<BrowserRouter>` and define `<Route path="/">` for LobbyPage and `<Route path="/game">` for GamePage. The LobbyPage component might include input fields and some buttons.

Testing: Navigate into the `frontend/rook-multiplayer-game` directory and run `npm start`. This should start the development server. Open `http://localhost:3000/` in your browser (or mobile device emulator). Verify the following: - The home page loads without errors. You should see an input for name, an input for game code, a “Create Game” button, and a “Join Game” button (or similar UI). - Entering a name and clicking “Create Game” currently might just route to the Game page (if the prompt wired it to navigation) or do nothing (if not yet handled – that’s fine for now). Similarly, “Join Game” might just navigate or be a placeholder. - You can manually test routing: go to `http://localhost:3000/game` and confirm the GamePage component displays (e.g., “Game Room” message). Use the browser’s back/forward or a provided link to return to “/”. - The app should be responsive. Try resizing the browser or using dev tools to simulate a mobile screen – the inputs and buttons should still be accessible and well-arranged (e.g., stacked vertically). - Check that the PWA assets are present: there should be a `manifest.json` and the app’s tab icon, etc. While in development CRA might not fully enable the service worker, the files should exist. You can run `npm run build` and serve the production build to test PWA installation later, but for now, ensure no errors related to service worker registration appear.

At this point, we have a basic React frontend set up and ready to be expanded.

Step 2: Render Player Hand and Table Layout (Game UI Skeleton)

Next, we will flesh out the Game page UI. In a Rook game, each player has a hand of cards, and there's a central "table" area where tricks are played. We need to design the layout such that players can see their own cards, see placeholders for other players' cards (face-down), and a central area for the current trick and kitty. We'll also prepare the UI for bidding and score display.

Prompt:

```
Enhance the `GamePage` component to lay out a 4-player Rook card table:

- Display placeholders for four player positions. The local player (the user) is
one of them – assume the user's hand is at the bottom of the screen. Show the
user's name and an area to display their cards.
- For now, use dummy data: create an array representing a hand of cards (e.g.,
["Green14", "Green10", "Green5", "Green1", "Black1", "Rook", ...]) to simulate
the user's cards. Render these as text or simple card-like `<div>` elements.
Sort them in descending order by rank (treat 1 as highest [1] ) and group by color
for visual clarity.
- For the other three players, just show a face-down card back for each card
they might have (since we shouldn't see their cards). For now, assume 13 cards
each. You can represent a card back as a Unicode card back emoji or a simple
rectangle with a pattern.
- Place the players around the "table": e.g., the user at bottom, one opponent
at top, two opponents on the left and right. Label them "Player 1", "Player 2",
etc., for now. Use a flexbox or CSS grid layout so that the table scales on
mobile (vertical orientation might stack opponents above and below, which is
fine).
- Include an area in the center for the current trick pile (cards played in the
current trick). For now, just reserve a `<div>` with text "Current Trick" in the
center.
- Also include an area for the kitty (5 cards set aside). Perhaps a small stack
of 5 face-down cards near the center or corner labeled "Kitty".
- Add a section for showing the current bid and trump once those are determined
(we will update it later). For now, maybe a placeholder text "Bid: -- Trump:
--".
- Style the components with CSS: make sure player names and hands are distinct.
Use responsive sizing so that on a small screen, cards might be smaller. The
layout should not overflow on mobile – cards can wrap to multiple lines if
needed.
- Ensure that the user's cards are clearly visible and identifiable (perhaps
color-code the text or background by suit color: e.g., Green, Black, Red, Yellow
suits). The Rook card can be a special case (maybe a black background or an 🃏
emoji if available).
- Keep all styling mobile-friendly (use relative sizes or flex so it shrinks/
grows).
```

Explanation: This prompt focuses on the **GamePage** UI within React. We are instructing Kiro CLI to create a visual layout for 4 players around a table. It should produce or modify the GamePage component (and possibly some CSS) to have sections for each player's area. The user's own hand is at the bottom, showing actual card values (using dummy card data for now). Other players' hands are represented as face-down cards because we can't see their cards in a real game.

We also mention sorting cards by rank with 1 as highest – Rook's house rules treat 1 as the highest card in each color [1]. We likely won't implement full sorting logic in the UI yet (that can be done either in the backend or with a helper), but the prompt hints at it. The AI might implement a simple sort where it knows or assumes an order (like mapping ranks to values, e.g., 1→15, 14→14, etc., internally).

Key UI elements being added:

- **Player name labels** for each position (for now generic "Player 1" etc., which we will replace with real names once the backend provides them).
- **Card representations:** We might just see text like "Green14" or colored spans. Eventually we could swap these out with actual card images or icons, but text is fine to start.
- **Center Trick area:** where played cards in the current trick will appear.
- **Kitty area:** indicating the kitty pile.
- **Bid/Trump display:** placeholder for now.

The layout is likely done with CSS Grid or Flexbox. Possibly the AI will choose a simple approach like a column for left player, a row for top, etc. On mobile, it may stack things. We want to ensure it's not too large for a phone screen, meaning cards might need to wrap if there are many.

After this step, the GamePage should have a fairly complex JSX structure, but mostly static. We will integrate real data later.

Testing: Run `npm start` (if not already running) and navigate to the game page (you can click the create/join buttons if they were wired to navigate, or manually go to `http://localhost:3000/game`). Verify the following on the UI: - Four player sections are visible (you might see "Player 1", "Player 2", etc., positioned around). On a desktop screen, you might see user at bottom, an opponent at top, and others on sides. On a narrow mobile screen, they might rearrange (possibly top and bottom, or a vertical list – the exact arrangement depends on the CSS). - The user's hand (bottom) should display multiple card identifiers. For example, you might see cards like "Green14, Green10, ... Green1, Black1, Rook" as dummy values. They should be sorted where "Green1" (the 1 card) is considered highest in that suit (likely placed appropriately in order). The Rook card should be present in the dummy hand and displayed distinctively (perhaps just as text "Rook" or an emoji if the prompt did that). - Each opponent's area (top/left/right) should show some number of card backs (like a symbol or rectangles). Likely 13 card-back symbols to represent their hand. This verifies that we cannot see their actual cards. - There should be a central area labeled "Current Trick" (currently empty of cards) and something indicating "Kitty" (maybe a small stack icon or text). These are just placeholders. - A text for "Bid: -- Trump: --" or similar should be visible, indicating those will be filled later. - The layout should adjust if you shrink the window. Ensure that on a mobile-sized screen, the content is still all visible (you might see players stacked or cards wrapping onto a new line – that's okay as long as it's usable). No element should be cut off or require scrolling ideally (except maybe vertically which is fine if needed). - Check the styling: are the suits color-coded? Perhaps the AI gave inline styles or CSS classes. For example, "Green14" might appear in green text. This isn't critical but nice to have. If not done, we can adjust later with CSS. - Overall, confirm that the GamePage component renders without errors in the console. At this point, everything is using dummy data, so no network calls or state from backend yet.

The UI now represents the game table structure. This gives us a framework to plug in real game data in subsequent steps.

Step 3: Add WebSocket Connection Logic (Client Side)

Real-time interaction in our game will be handled via a WebSocket connection to AWS API Gateway. In this step, we will add logic on the frontend to connect to the WebSocket API when the player enters a game, and handle basic connect/disconnect events. We won't have a real endpoint until we deploy, so we will set it up in a way that the URL can be easily configured later (for now maybe use a placeholder).

Prompt:

```
Integrate WebSocket connectivity into the React frontend:

- In the `GamePage` component (or a dedicated context/hook), add logic to
connect to a WebSocket when the component mounts. Use `window.WebSocket`. The
WebSocket URL will be the AWS API Gateway WebSocket endpoint (which we will
define on deploy). For now, create a configuration constant for the WebSocket
URL (e.g., `WS_BASE_URL = "wss://example.com/dev"` as a placeholder).
- When the GamePage loads, open a WebSocket connection to `WS_BASE_URL` **with
query parameters** for the game ID and player (we will have those from when the
user joined or created a game). For now, you can use dummy values or come back
to fill the actual gameId and playerName from state (we will manage state in a
later step). Perhaps assume we have `gameId` and `playerName` in local storage
or in the URL (we will set this up later).
- Implement event handlers: on `open` event of WebSocket, log a message like
"WebSocket connected". On `close`, log "WebSocket disconnected". On `message`,
parse the incoming data (assume JSON) and log it or update a placeholder state.
- Create a React state to track connection status (e.g., `isConnected`) and
maybe last message or game state received. For now, we will just console.log
incoming messages.
- Also handle the scenario where the component unmounts (user leaves the game
page): close the WebSocket connection to clean up.
- Provide some UI indication of connection status on the GamePage, e.g., a small
green dot or "Connected" text when `isConnected` is true, and red or
"Disconnected" if false.
- No actual game logic yet – just ensure the client tries to connect to the
WebSocket and is ready to handle messages.

If needed, store the `WebSocket` object in a React ref so that other parts of
the component can use it to send messages (we will send actual game actions in
later steps). For now, maybe add a dummy “Test Connection” button that sends a
```

```
hardcoded message (like `{"action":"ping","data":"hello"}`) to the server via
WebSocket, just to test sending.
```

Explanation: Here we are adding WebSocket support to the frontend. In our architecture, the front end will connect to an AWS API Gateway WebSocket endpoint for real-time game updates (bids, moves, etc.). We haven't deployed the backend yet, so we'll use a placeholder URL. The actual WebSocket URL will look like `wss://<api-id>.execute-api.<region>.amazonaws.com/<stage>` once deployed.

The prompt instructs Kiro CLI to modify the GamePage (or a related module) to establish a WebSocket connection when the page loads. We mention passing **query parameters** for gameId and playerName – a common pattern is to include them in the connection URL (like `wss://.../Prod?gameId=XYZ&playerName=Alice`). This will later help the backend identify who is connecting [2]. We assume that by the time the user navigates to GamePage, we have a game ID and the user's name stored (we'll likely store these in a global state or localStorage when they create/join a game). The prompt suggests using dummy values for now if needed, which is fine because we will fill actual values in a later step once the join logic is in place.

We also handle WebSocket events: - **onopen:** set `isConnected=true` (and maybe show a "Connected" UI). - **onmessage:** for now just log the event. Later, we'll update game state based on messages. - **onclose:** set `isConnected=false` and possibly attempt reconnection or just alert the user.

We also mention a cleanup: closing the socket if the component unmounts (using `useEffect` cleanup function). This prevents dangling connections.

Additionally, we suggest a test button to send a message. This is optional but can be helpful to verify client-to-server messaging. It could send a JSON payload with an `action` field (since API Gateway WebSocket routes often use a field like "action" to route messages). For example, a ping or a dummy game action. Without a backend echo, we can't fully test it now, but at least we ensure the send function works (it won't throw an error when invoked).

After running this prompt, our frontend will have the logic in place to connect to the WebSocket. The code might be in GamePage or a custom React hook (like `useGameConnection`) that GamePage uses.

Testing: At this stage, we don't have a real WebSocket server to connect to (unless you set up a test echo server). However, we can still verify parts of the functionality: - The app should compile without errors. Check the browser console for any syntax errors or undefined variables related to WebSocket. - If you want to test the connection logic, you can use a public WebSocket echo service (for example, `wss://echo.websocket.org` – though note that service might not be reliable or may have been discontinued). Alternatively, if you have `wscat` or another WebSocket server handy, update `WS_BASE_URL` to that and see if the client connects. - Open the browser console on the Game page. You should see a console log like "WebSocket connected" (assuming the placeholder URL was reachable). If using a dummy URL that's not valid, the console might show connection errors – that's expected until we deploy. The key is that our code attempts the connection without throwing exceptions. - Click the "Test Connection" (if implemented). If the WS is not actually connected (due to placeholder), it might throw an error or just not send. That's okay; once we have a real endpoint, we'll test again. The main thing is that the button calls `ws.send()` properly. You can verify by checking that our code retrieved the WebSocket object and tried to send JSON. - Check that the

UI shows connection status. Likely there's a piece of text or an indicator in the GamePage. It might say "Connected" (if it falsely assumes connection on open event) or "Disconnected" if the socket closed immediately due to no server. If you see it flip quickly or remain disconnected, that's normal for now. - Ensure the WebSocket is closed when leaving the Game page. You can navigate back to "/" and check if any console log says "WebSocket disconnected". Also, in dev tools Network tab, look for any lingering WebSocket connections.

This step sets the stage for real-time updates. We'll revisit the actual URL and integration once we deploy the backend WebSocket API (in a later step).

Step 4: Create Backend API for Game Creation and Joining

We will now begin implementing the backend functionality. The first server capabilities we need are: - An API to **create a new game** (the host uses this to start a lobby). - An API to **join an existing game** (players use this with a game code).

We'll use AWS Lambda functions for each and expose them via HTTP API Gateway endpoints. Let's implement these Lambda handlers and define the DynamoDB schema as needed for these operations.

Prompt:

```
Switch to the backend. Create a new Node.js project in a `backend` folder for
our AWS Lambda functions (use Node.js 18). We need two Lambda functions to
start:
1. **createGame** – Triggered via an HTTP API (POST `/createGame`). This
function should:
   - Generate a new game entry in a DynamoDB table `Games`. The game ID can be a
short unique code (e.g., 6-digit or 4-letter code). Include relevant initial
data: gameId, createdBy (host name), players list (with the host as player0),
and an initial state (e.g., status "LOBBY", no cards dealt yet).
   - The game entry should also include fields for scores (team scores starting
at 0), and any other metadata (like creation timestamp).
   - Return a response with the game ID and the player's assigned seat (host
will be seat 0).
2. **joinGame** – Triggered via HTTP API (POST `/joinGame`). It accepts a gameId
(code) and a player name. This function should:
   - Look up the game in `Games` table. If not found or already full, return an
error.
   - If found and not full, add the new player to the game's players list with
the next available seat number. For 4 players max, seats go 0-3.
   - Update the game record in DynamoDB with the new player. Use a conditional
update (optimistic locking with a version number) to ensure no race conditions
if two join at once [3] .
   - Return a success with the player's assigned seat and maybe current list of
players (or game state snippet).
   - If the game became full (4 players) after this join, also update the game
```

```
status to "FULL" (we'll handle game start in a later step).
   - (We will handle partner selection separately, so at this point teams are not set yet.)

Set up a DynamoDB table schema:
- **Games table:** Partition key `gameId` (string). It will store a JSON object for game state. Fields: gameId, players (array of {seat, name}), hostName, status, scores (maybe an object {team0:0, team1:0}), version (number for concurrency control), etc.
- (We will have separate tables for Hands and Connections later, but for now focus on Games.)

Implement both Lambda handlers in separate files (e.g., `createGame.js` and `joinGame.js`). Use the AWS SDK (DynamoDB DocumentClient) to put/get/update items. Handle errors (e.g., game not found, game full) with appropriate HTTP error codes (400/404).
Make sure to export the handler function for AWS Lambda.
```

Explanation: This prompt instructs the creation of a backend Node.js project and two Lambda function handlers for creating and joining games. We are also defining our data model in DynamoDB as we code these.

Important points in implementation: - **Game ID Generation:** We want a short code for ease of sharing (the user asked for a code to join). Kiro CLI might implement this using a random combination of letters or numbers. It could use `Math.random()` or a library. We should ensure uniqueness; using part of a UUID or a timestamp combined with random can work. DynamoDB will use gameId as primary key. The chance of collision with a 6-digit code exists but is small; we might not worry or we could loop-check existence (not required for this scope). - **DynamoDB Integration:** We mention using AWS SDK's DocumentClient. This requires configuring the region and credentials. Likely, the code will use environment variables for table names. E.g., `process.env.GAMES_TABLE`. We might instruct that in a deployment step, but for now the code can hardcode table name "Games" for simplicity. - **Game Item Structure:** We propose storing players as an array of objects with seat and name. E.g., `players: [{ seat: 0, name: "Alice" }]` after creation, then append others on join. We also include a `version` attribute (starting at 0 or 1) for optimistic locking [3]. The `status` could be "LOBBY", "FULL", "STARTED", etc. Scores can be stored as `{ team0: 0, team1: 0 }`. - **Optimistic Locking:** We referenced using a version number. The `joinGame` function should use a conditional update: only add the player if the version matches what we last saw. This prevents two people joining at the same time causing one to overwrite the other. DynamoDB's `ConditionExpression` can ensure the item's version is a specific value. If it fails, the code can retry by refetching or just throw an error (though unlikely to collide often). - **HTTP Response:** As these are Lambda behind API Gateway (HTTP API), we need to return an object with statusCode, headers, body, etc., or use AWS Lambda Proxy integration format. The prompt doesn't explicitly mention, but we should ensure to return JSON with gameId/seat for frontend use. Possibly, `createGame` returns `{ gameId: "ABCD1234", seat: 0 }`. `joinGame` returns `{ gameId: "ABCD1234", seat: 1, players: [...] }` or similar. - **Game Full Condition:** If players length becomes 4, we mark status "FULL". We won't start dealing here; that will happen later. But marking full can prevent further joins. - **Error Handling:** For example, if a join is attempted on a non-existent code, return 404. If a game is full or some race condition, return 400 with a message. We want the frontend to display such errors (maybe via an alert or message, which we can implement later).

After Kiro CLI runs this prompt, we expect: - A Node.js project (perhaps initialized with `npm init -y` by the AI) in `backend` directory. - A dependency on AWS SDK (for Node 18, AWS SDK v2 is preinstalled in Lambda, but for local usage we might need to install `aws-sdk`). The AI might add it to package.json. - Two files: `createGame.js` and `joinGame.js` exporting `handler`. - Possibly a basic `Games` table schema comment or a separate file (but likely just handled in code for now). - Each handler performing the logic: - `createGame`: generate ID, put item (with ConditionExpression to ensure no existing item with that key maybe). - `joinGame`: get item, modify players array, use Update with ConditionExpression on version, etc.

We haven't created the DynamoDB table itself yet – we'll do that on deploy. But the code assumes it exists.

Testing: We can test these Lambda functions locally by simulating their invocation: - **Unit Test Locally:** Create a temporary test script or use Node REPL to call the handler. For example, in `backend` folder, run:

```
node -e "require('./createGame').handler({ body: JSON.stringify({ playerName:
'Alice' }) }, {}, console.log)"
```

This simulates an event with a body containing a playerName. Since we haven't configured AWS access for local, this will attempt to write to DynamoDB. You can configure AWS SDK to point to a local DynamoDB instance if available. If not, you might see an error. Alternatively, deploy these and test on AWS console: - **Integration Test on AWS:** If you have AWS CLI configured, you can create the DynamoDB table manually:

```
aws dynamodb create-table --table-name Games \
  --attribute-definitions AttributeName=gameId,AttributeType=S \
  --key-schema AttributeName=gameId,KeyType=HASH \
  --billing-mode PAYPERREQUEST
```

Wait for it to be ACTIVE. Then use AWS Console or CLI to invoke the Lambda (after deploying the function code). Since deployment isn't done yet in our process, consider this pseudo-tested until we deploy in Step 15. - **Dry-run logic check:** We can manually inspect the code to ensure logic: - Does `createGame` generate a code (e.g., a 6 char string)? - After running, the DynamoDB should have an item with that gameId and players array length 1. - `joinGame` when given that gameId and a new player name should update the same item's players array to length 2. If you run joinGame again with two more names, by the 4th name it should mark full. - The version number should increment on each update (if we implemented it). We might not explicitly increment version in code yet; possibly we just used it as a condition. We might need to increment it as part of update expression. - Check that error paths return appropriate messages.

Because we can't fully run this without AWS connectivity, at this stage rely on careful review or minimal local tests. We will rely on integration tests after deployment.

We now have fundamental backend APIs for creating and joining games. Next, we will integrate these with our frontend and proceed to game logic.

Step 5: Define Game State Schema (DynamoDB Tables and Data Model)

Before proceeding further with game logic, let's solidify the database schema and ensure our backend knows about all the necessary tables. We have partially done this for the Games table. Now we will outline and (in code) define the other tables and any shared data structures:

- **Games** table: already used for basic info. We will confirm its attributes and possibly add any missing ones needed for gameplay state (trump suit, current turn, etc.). We'll also include the `version` attribute for optimistic locking on updates [3].
- **Hands** table: to store each player's hand of cards (so that we don't reveal other players' cards when retrieving game state). Partition key could be `gameId`, sort key `playerId` or `seat`. This way, each item stores the cards for one player in a game. We'll decide on an identifier for players (could use seat number as unique within game).
- **Connections** table: to track WebSocket connection IDs for players. Partition key can be `gameId`, sort key `connectionId` (or vice versa). We will use this to send messages to all players in a game by querying connections by gameId [4]. Each entry will also store which player (seat or name) it corresponds to.

We will update our backend code to account for these: - Modify the Lambda functions (and future ones) to know the table names via environment variables or constants. - Possibly create a common module for the DynamoDB DocumentClient and table name constants to reuse across functions. - Prepare the data model for game state: e.g., define a TypeScript type or at least document the structure in comments for clarity. Key fields: - `players`: array of {seat, name, team?}. - `scores`: {team0, team1}. - `phase` or `status`: e.g. "BIDDING", "PLAYING", "FINISHED". - `currentTurn`: seat of player whose turn it is. - `currentTrick`: an array of cards played in the ongoing trick (each element could be {seat, card}). - `trump`: the trump suit (once chosen). - `highBid` and `highBidder`: track current bid in bidding phase. - `bidHistory` or passes if needed. - `version`: for concurrency.

We will not fill all of these now, but we should add placeholders in the game item. This will make it easier to update state in later steps without having to add new attributes from scratch (which DynamoDB allows on the fly, but having them upfront is conceptually cleaner).

Prompt:

```
Expand the backend to include our DynamoDB schema and shared structures:

- In the `backend` project, create a config file or constants (e.g.,
`dbConfig.js`) that exports the DynamoDB table names (Games, Hands, Connections)
and initializes a DynamoDB DocumentClient. Use `process.env.GAMES_TABLE`, etc.,
so we can configure table names via environment variables in deployment.
- Update the `createGame` Lambda to use these constants (instead of any
hardcoded table name). Similarly, prepare the `joinGame` to do the same.
- Define the initial game state structure in `createGame`: When creating a new
game item, include fields for things we know we'll need:
  - `gameId` (string, primary key).
  - `players` (array of objects with `seat` and `name`; host is seat 0).
```

```
  - `status` (start as "LOBBY" or "BIDDING_PENDING").
  - `scores` (object with `team0: 0, team1: 0`).
  - `round` (start at 1, to track hand number).
  - `highBid` (start at 0) and `highBidder` (null or empty).
  - `trump` (null until chosen).
  - `currentTurn` (null or perhaps set to seat 1 when game starts, since left of
dealer leads [5]).
  - `currentTrick` (empty array).
  - `version` (number, start at 1).

- We won't set all of these in createGame (some remain null until game
progresses), but include the keys with initial values or placeholders.
- Create the **Hands** table concept in code: For now, when a game is created,
also create 4 items in a `Hands` table for each seat (0-3) with empty `cards`
array (except we might hold off until the game is actually starting).
Alternatively, we can create hand records at deal time. It might be simpler to
create them when dealing, so just note the table usage.
- Prepare for the **Connections** table: We will use it when players connect via
WebSocket. Create a placeholder Lambda function file `connect.js` and
`disconnect.js` (for the $connect and $disconnect routes of the WebSocket API).
In `connect.js`, plan to put an item in Connections table with `gameId`,
`playerSeat` (or name), and `connectionId` (from event.requestContext). In
`disconnect.js`, delete that item. For now, just scaffold these functions (no
full implementation yet, just log the event and basic structure).
- Ensure all these functions (createGame, joinGame, connect, disconnect) are
aware of the table names via config.
- Lastly, increment the `version` in the `joinGame` function’s DynamoDB update.
When a player joins, set `version = version + 1` in the update expression so
that each game update bumps the version number.
```

Explanation: In this step, we are solidifying the backend data model and preparing additional Lambda functions for WebSocket connection management. The prompt has several parts:

- **Database Config:** Instead of repeating table name strings and creating multiple DynamoDB clients, we want a single config. The AI might create a `dbConfig.js` (or similar) where it does:

```javascript
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();
const GAMES_TABLE = process.env.GAMES_TABLE || "Games";
const HANDS_TABLE = process.env.HANDS_TABLE || "Hands";
const CONNECTIONS_TABLE = process.env.CONNECTIONS_TABLE || "Connections";
module.exports = { ddb, GAMES_TABLE, HANDS_TABLE, CONNECTIONS_TABLE };
```

  This allows all our handlers to import these and use the same DocumentClient and table references. During deployment, we’ll set those env vars to actual table names.

- **Game Item Structure:** We enumerate many fields that will be used in gameplay. The createGame handler should initialize as many as reasonable:

- `players`: already adding host.
- `status`: maybe "LOBBY" meaning waiting for players (or we can call it "OPEN").
- `scores`: team scores at 0. We have not set teams yet, but we can still maintain two team scores (they'll remain 0 until we assign teams and start scoring).
- `round`: set to 1.
- `highBid` and `highBidder`: no bidding yet, but we can set highBid = 0, highBidder = null.
- `trump`: null (or empty string).
- `currentTurn`: we might leave null until game starts. The note references left-of-dealer leads the first trick [5], but we won't determine currentTurn until after dealing and trump selection.
- `currentTrick`: [] (empty array).
- `version`: set to 1.

We won't set `team0`/`team1` or partner here; that will come after partner selection. We might also add a boolean like `partnerChosen: false` or store partner info later.

- **Hands Table:** We don't yet create entries in code. The prompt suggests possibly creating empty hand records at game creation, but it also acknowledges we might wait until dealing (which makes sense, since before dealing no one has cards). So likely we won't actually put anything in Hands table in createGame. We will do it when shuffling/dealing in a later step. But we mention its structure: Partition key gameId, sort key seat or player. Each item will have something like { gameId: "XYZ", seat: 0, cards: [...] }. This table allows fetching a single player's hand or all by querying on gameId.

- **Connections Table:** We prepare the connect/disconnect Lambdas:

- `connect.js` handler: On $connect event, API Gateway provides a `connectionId` and query params (gameId, playerName or seat). We will put an item in Connections table, e.g., { gameId: ..., connectionId: ..., player: ... }.

- `disconnect.js`: On $disconnect, remove that item (or mark offline). For now, we can scaffold them (the prompt says just log event, but ideally we implement basic functionality). It's fine if we implement now:

```
// connect.js
const { ddb, CONNECTIONS_TABLE } = require('./dbConfig');
exports.handler = async (event) => {
  const connectionId = event.requestContext.connectionId;
  const gameId = event.queryStringParameters.gameId;
  const player = event.queryStringParameters.player; // this could be name
or seat
  // Put item in connections table
  await ddb.put({ TableName: CONNECTIONS_TABLE, Item: { gameId,
connectionId, player } }).promise();
```

```
  return { statusCode: 200, body: 'Connected.' };
};
```

And similarly for disconnect:

```
// disconnect.js
exports.handler = async (event) => {
  const connectionId = event.requestContext.connectionId;
  // Possibly gameId is not directly provided on disconnect events, so we
may have to store gameId as part of the connectionId key or have a GSI.
  // Simpler approach: use connectionId as primary key of Connections table
(instead of gameId as PK).
  // Let's assume we made connectionId the PK, with gameId as an attribute
or as sort key.
  await ddb.delete({ TableName: CONNECTIONS_TABLE, Key: {
connectionId } }).promise();
  return { statusCode: 200, body: 'Disconnected.' };
};
```

We have a design decision: **Connections table key**. If we use gameId as PK, we need connectionId as sort key (to allow multiple connections per game and to query by game). That's good for broadcasting by game. But deleting by connectionId alone becomes harder because we need gameId to delete the exact key. If we use connectionId as PK, easy to delete, but broadcasting by game requires a scan or GSI. A common solution: use gameId as PK and connectionId as sort key, and also store gameId in each item (which is already the PK). For deletion, we can keep track of gameId in memory from connect step (like store mapping in memory, not ideal) or do a query by a GSI on connectionId.

Simpler: we can cheat by storing connectionId as PK and also store gameId as attribute, and maintain a GSI for gameId -> connectionId query. This is a bit advanced. Alternatively, just use gameId as PK and connectionId as sort, and on disconnect event, query by a reverse index or have the client send a disconnect message before actually disconnecting (not guaranteed).

The AWS blog example likely used gameId as PK and connectionId as sort key [4]. How to handle disconnect: maybe they stored player info in connectionId mapping. But in $disconnect event, we only have connectionId. We can do a **DynamoDB query by connectionId** if we have a global secondary index (GSI) on connectionId (making connectionId a key in GSI). That's probably easiest: define a GSI so we can get item by connectionId.

We'll assume we can do that. We'll note this as a design detail in code comments, perhaps.

For now, the prompt might not dive into that nuance deeply, but our final answer could mention needing to handle disconnect by connectionId lookup.

- **Using Version in joinGame:** We explicitly ask to increment version on update. So joinGame's update expression should include `SET version = :newVersion` with

`:newVersion = oldVersion + 1` (and use ConditionExpression `version = :oldVersion`). DocumentClient can do this via an atomic update and condition.

After applying this step's prompt, our backend codebase should be more structured: - A `dbConfig.js` with shared config. - `createGame.js` and `joinGame.js` using the config and including more fields in game item. - New files `connect.js` and `disconnect.js` with basic logic (and possibly a `default.js` for $default route if needed later). - Possibly updated package.json with AWS SDK (if not added already). - The code should compile (if we run something like `npm install aws-sdk` and lint).

Testing: We can do limited local testing: - Import `createGame.handler` and simulate an event as before. Now it expects environment variables or uses default "Games". If we have a local DynamoDB or AWS creds, it will try to write with more attributes now. You can print the params to verify structure. For example:

```
const evt = { body: JSON.stringify({ playerName: "Alice" }) };
require('./backend/createGame').handler(evt, {})
  .then(res => console.log(res))
  .catch(err => console.error(err));
```

You might add a console.log inside the function to output the `params` sent to DocumentClient to verify all fields. - Similarly for joinGame: simulate an event with a known gameId and a new player name. But to test joinGame, you need an existing item in the Games table. Without a running DynamoDB, you can mock DocumentClient by overriding methods in `dbConfig.ddb` for a dry run. - Ensure that `version` increments. We can test joinGame twice sequentially (simulate 2 joins) and see if second join would increment version to 3, etc.

- Test the connect/disconnect logic by calling them with a dummy event:

  ```
  require('./backend/connect').handler({
    requestContext: { connectionId: "abc", domainName:"test", stage:"dev"},
    queryStringParameters: { gameId:"GAME123", player:"Alice"}
  }, {}, console.log);
  ```

  This will attempt to put to Connections table (which will error if DynamoDB not reachable). But at least see if any exceptions in code (like if queryStringParameters is undefined, etc.).

At this point, we have structured our backend to support further development: - The schema is defined and will be used by subsequent steps (bidding, dealing, etc.). - We have placeholders for WebSocket connect/disconnect which we will fully utilize soon.

Step 6: Implement Player Join Flow & Partner Selection

Now that the backend join logic exists, we can connect the frontend to it, and implement the partner selection feature. The "host picks partner" rule means once all 4 players have joined, the host (player0) chooses one of the other three players to be their partner for the game [6]. We'll handle this in two parts: 1. **Frontend – Lobby integration:** Wire up the "Create Game" and "Join Game" buttons to call our backend API endpoints (using fetch or Axios). Store the returned gameId and seat in state or localStorage, then navigate to the Game page. Pass along the player's name, gameId, and seat to the Game page (perhaps via query params or a global context). 2. **Backend – Partner selection:** When a game is full, the host can select a partner. We'll create a new WebSocket message action (e.g., `"choosePartner"`) that the host's client will send with the chosen player's seat number. We then implement the backend to handle this: - Update the game item: determine teams based on the choice (if host=0 chooses seat X as partner, then team0 = {0, X}, team1 = {the other two}). Mark that partner selection is done, and maybe set `status = "BIDDING"` to start the bidding phase. - We can also assign team IDs to players in the game state or just infer team by seat membership. - Notify all players via WebSocket messages of the teams (so their UI can update to show who's partnered with whom).

1. **Frontend – Partner UI:** On the GamePage, if the game is full but teams not set yet, and the current player is host, display a UI prompt to select a partner. This could be a simple modal or section listing the other three players with a "Choose Partner" button for each. When clicked, send the `"choosePartner"` action through WebSocket with the target's seat.
2. Non-host players might just see "Host is choosing partner" message until teams are decided.
3. Once a partner is chosen, update the UI to indicate teams (e.g., highlight partner names or group players by team).

Let's implement these:

Prompt:

```
Integrate game creation/join on the frontend and add partner selection:

- **Frontend (LobbyPage)**: Implement the onClick handlers for "Create Game" and
"Join Game":
  - For "Create Game": on button click, send a POST request to `/createGame`
endpoint (e.g., using `fetch('/createGame', { method:'POST', body:
JSON.stringify({ playerName }) }`) to our backend API. (In development, this
might be an http://localhost:someport if testing locally; eventually it will be
a CloudFront domain or API Gateway URL – consider making the base API URL
configurable).
    Parse the JSON response which should contain at least `gameId` and maybe
`seat`. Save `gameId`, `playerName`, and `seat` to a React state (maybe context
or a global store) or localStorage. Then navigate the user to `/game`.
  - For "Join Game": similarly, POST to `/joinGame` with `gameId` (entered code)
and `playerName`. Handle errors (if game not found or full, perhaps alert the
user with the error message from response). On success, get `seat` and proceed
to `/game` with state stored.
- We might not have a running backend locally yet; you can simulate by using
dummy responses for now, but structure the code so it will call the real
endpoints when available. Perhaps configure the API base URL in a constant (to
be updated after deployment).
- **Frontend (GamePage)**: When loading the GamePage, retrieve `gameId`,
`playerName`, and `seat` from where we stored them (context or localStorage).
Use these to establish the WebSocket connection (replace the dummy placeholder
```

```
with the actual `?gameId=...&player=...` query).
  Also, display the list of players in the game. We have the names from the join
responses: the host after creating game has only themselves initially, but when
others join, we should update the players list. We will get updates via
WebSocket (we'll implement a handler for a "playerJoined" message to add to the
list).
- **Partner Selection UI**: On the GamePage, if `players.length === 4` and teams
are not yet set (we can track a state `partnerChosen` or check if any player has
a `team` assigned), then:
  - If the current user’s seat is 0 (host), show a prompt: "Choose a partner"
and list the other three players (by name). Each name has a button "Pick
Partner". On click of one, send a WebSocket message: e.g.
`ws.send(JSON.stringify({ action: "choosePartner", seat: <chosenSeat> }))`.
  - If the current user is not host, display a message "Waiting for host to
choose a partner".
- **Backend WebSocket (gameAction)**: Implement a new case in the WebSocket
message handler (we haven't fully written `gameAction` Lambda yet, so do it now
or at least pseudocode):
  - When a message with `action: "choosePartner"` is received (from host),
retrieve the game from DynamoDB. Determine teams: host (0) and chosen seat form
Team0, the other two seats form Team1 [6]. Update the game item: add a `teams`
field or add `team` attribute to each player in the players array. For example,
set `players[i].team = 0 or 1` accordingly. Also update `status` to
"BIDDING" (since after partner selection, bidding will start).
  - Use a conditional write (check version) and increment version as well.
  - Then, broadcast a message to all players about the teams. Possibly the
message could be `{ action: "partnerSelected", teams: {0:<name0>,2:<name2>,
1:<name1>,3:<name3>} }` or simply an updated players list with team info.
- **Frontend Reaction**: Handle the "partnerSelected" message in the WebSocket
onmessage handler. When received, update the state to mark partner selection
complete, update player list with teams, and maybe show a message like "Teams
set: [Host] partnered with [Partner], versus the other two."
  Update UI so that partners are visually indicated (you can group or color-code
team members).

Make sure to handle the case that the host might pick seat 2 as partner (the
default opposite seat scenario) or any other seat – our code should work for any
choice.
```

Explanation: This step connects a lot of pieces: - **Front-End API calls:** We instruct how to call the backend endpoints from the React app. Since our backend likely will be deployed on some domain, in development we might not directly call it (unless CORS is set up). We might assume our front and backend will share a domain or that we’ll handle CORS. The prompt suggests using `fetch('/createGame', ...)` which implies same domain; in a dev environment, we might need to proxy or use a full URL. But these details can be sorted out later (maybe when deploying, we use the CloudFront domain which forwards to API Gateway).

The result: - When user clicks Create, the code will do a fetch. We should implement it as an async function that sets some app state and navigates on success. - We need to store `gameId`, `playerName`, and `seat`. We can use React Context or localStorage. A straightforward way: localStorage, so that if the user refreshes on the game page, we can still retrieve who they are and attempt reconnect. We'll indeed need something for reconnection logic in Step 13. So using localStorage now is wise. - For simplicity, we might use localStorage for now. So: - `localStorage.setItem('rookGameId', gameId)`, same for name and seat (and possibly a playerId if we had one). - Then navigate. Possibly using React Router's `useHistory` to push `/game`.

- **GamePage usage of stored info:** On load, GamePage can pull these from localStorage. We might also pass them via React Router state (some libraries allow passing state when navigating), but localStorage is fine.

- **Updating WebSocket URL:** Replace placeholder with actual URL (which we will have after deployment). But right now, we might not know it. We can keep it configurable. Possibly set `WS_BASE_URL` from an environment variable or define it after deployment. For now, maybe leave a TODO or use a dummy if not known. This is tricky because for local testing we might simulate or skip actual connect.

- **Player list state:** Initially, after creating game, only host is in players. After joinGame, the joiner will know current players (maybe returned by joinGame). We might have joinGame return the updated players list. If we didn't, we could fetch the game state after joining (via a GET endpoint which we haven't built, or rely on WebSocket event). Perhaps easier: modify joinGame Lambda to include `players` in the response. The prompt did hint "maybe return players list". Let's assume we did that (if not, we adjust joinGame to get current players from DB and include them). If joinGame returns the updated players list, the joining client can display all current names. Also, the host's client (and others already connected) need to get notified of the new player. We can handle that via WebSocket message broadcast in joinGame. We have not explicitly done that yet. We might need to modify joinGame to send a WebSocket message using the connections table (like "playerJoined" event). But since this step is big, maybe we can simply retrieve the players list by polling or by the join response for now. Alternatively, implement a quick polling: the GamePage could call a GET /gameState when it loads. But we didn't create such an endpoint specifically. Possibly skip for now and assume the joiner sees all names in their response, and the host gets a "playerJoined" WS message which we'll implement soon (like in Step 8 or 9).

- **Partner Selection UI:**

- For host: We add UI (maybe in GamePage or a subcomponent) that appears when `players.length === 4` and `teams` not set. It lists players except host, each with a button. Possibly by seat or just by name and we know seat from players array.
- On click, calls `send({action:"choosePartner", seat: X})` via WebSocket.

- Non-host: just a message like "Waiting for partner selection".

- **Backend gameAction Lambda (which we might name something like `gameMessage` or so)**: We have not fully built the general message handler Lambda yet. We should create a single Lambda to handle all game-related WebSocket messages (bids, play, etc.), often configured as the `$default` route or specific routes. We can either:

  - Use `$default` route in API Gateway and parse `event.body` to dispatch actions.
  - Or define explicit routes like "choosePartner", "bid", etc. Simpler is a single function with a switch on action.

We'll implement just the partner selection logic in it now for demonstration: - It will get the message from event.body (string) – parse JSON to get `action` and `seat`. - If `action === "choosePartner"`, do the logic: * Retrieve game from Games table (using gameId from event.requestContext or from the Connections table? The Lambda event for message will have connectionId and maybe we need to find gameId from that). * Actually, in a WebSocket message Lambda (for $default route), `event.requestContext` includes connectionId, but not gameId directly. We might need to get the gameId: one way is to have clients send gameId in the message too (which they could). - Perhaps we should include `gameId` in each message payload for clarity. But since every connection is tied to a game (based on connect), we could also look up the gameId by querying the Connections table by connectionId. - The prompt did not explicitly handle that, but likely we need to do it: find the connection in Connections table to get which game and which player is sending this. * For simplicity, let's assume the message includes gameId (the front end knows it, so include it in the JSON payload to avoid an extra DB read). * Then, get game item, update players with team assignments. * Save to DB (condition on version). * Then send out a WebSocket message to everyone with updated teams. We already have connection IDs in Connections table by game, so iterate through those and use ApiGatewayManagementApi (the mgmt API). - We might not fully code the broadcast in this prompt, but mention sending "partnerSelected" with team info.

This is a lot, but focusing only on partner logic for now.

- **Teams assignment logic:** If host = 0 chooses seat X: Team0: {0, X}, Team1: the other two (one of which will be seat (X+1)%4 and the other (X+2)%4, depending on X). Actually, easy way: if host picks seat 1, team0 = [0,1], team1 = [2,3]; if picks 2, team0 = [0,2], team1 = [1,3]; if picks 3, team0 = [0,3], team1 = [1,2]. We update players array by adding `team` property (0 or 1) to each object. Alternatively, store a `teams: {0:[players], 1:[players]}` in game state, but easier to tag players.

Mark `partnerChosen` or just infer from presence of `team` fields or status "BIDDING".

Also we set status "BIDDING" to indicate game is now in bidding phase.

Also perhaps set `currentTurn` for bidding start. Typically bidding starts with host in our rules [7] (we set that variant). So we could set some pointer (like `currentBidder = 0` initially). The PDF said host begins bidding [7], which we followed. We might not do it right now, but soon.

- **WebSocket broadcasting partnerSelected:** We would iterate through connections of that game (which we can get by querying Connections table on gameId, since we set gameId as PK). Then call postToConnection for each to send e.g., `{"action":"partnerSelected","teams":{"0":<name0>,"X":<nameX>}}` or a full updated players list.

The prompt suggests possibly sending the entire players list with team assignments (which might be easier for front end to just replace its state). That could be done as `message = { action: "partnerSelected", players: [...] }`.

- **Frontend onmessage:** We parse JSON, if action === "partnerSelected", we update our state: mark teams (maybe store an array of team0 players and team1 players or just update each player object with a team). Then update UI: e.g., maybe color names of team0 one color, team1 another, or list partners.

Also possibly remove the partner selection UI if it was open.

And possibly move to bidding UI (maybe show something like "Bidding starts. Highest bid so far: ..."). But we haven't built bidding UI yet, that will come soon.

Given the complexity, we may not fully implement the message broadcasting in code in this single prompt (the AI might give pseudocode or a description), but we'll try to be as concrete as possible.

Testing: This end-to-end flow is complex to test fully without deployment. But some partial tests: - After this step, run `npm start` for frontend: - Try create game: It will attempt a fetch to /createGame. If our backend isn't running, you'll get network error. But check that the code is executed (maybe open network console to see request). If working locally with some mock server or after deployment, the response would set localStorage and navigate. - We can simulate a response by temporarily modifying code to skip fetch and directly call navigate with dummy gameId, but it's fine. - The navigation to /game should happen, and localStorage now has gameId, name, seat. - On GamePage, the WebSocket connection will attempt with actual gameId and player. - Since backend isn't up, it might fail to connect. But ensure no runtime errors: the URL might be wrong but code should handle onerror or just show Disconnected. - Check that if 4 dummy players are present (we can simulate by manually modifying the players state in dev tools or code), the host gets partner selection buttons, others get waiting message. - Without actual WS, we can't test the message handling, but we can simulate receiving a partnerSelected event by calling the onmessage handler manually in code or adjusting state manually.

- The backend:

- Could test choosePartner logic by calling the function if we write it. But we might not have fully integrated it. If we created a `gameAction.js` for $default messages, we could simulate an event:

  ```
  require('./backend/gameAction').handler({
    requestContext: { connectionId: "abc" },
    body: JSON.stringify({ action: "choosePartner", gameId: "GAME123", seat:
  2 })
  }, {}, console.log);
  ```

  It would need to find game GAME123 from Dynamo and update. If no Dynamo, we could create a dummy item first or mock DocumentClient methods to see logic path.

- If connect/disconnect were properly storing connections and our game item exists with players, we can test that logic indirectly after deployment by actually performing the flow:

    - Create game (via HTTP) -> then join from other three, -> host picks partner via UI -> see if game state updated in DB and messages sent. This end-to-end test would be done on deployed environment (which we plan in step 15).

At the end of step 6, our system supports the full lobby flow and partner selection: - Users can create/join games. - All four players get into a game and teams are decided by host's choice.

Next steps will cover dealing cards, bidding, etc., now that teams are established.

(Note: We cited [11†L277-L284] which indicated a fixed partnership in one design (0-2 & 1-3), but we allow host to pick any. Our implementation covers any choice. We used that citation to acknowledge the common opposite seating arrangement.)

Step 7: Add Shuffle and Deal Logic (Secure RNG for Cards)

With teams set and the game ready to start, we move on to dealing the cards. We need to simulate a shuffled Rook deck and distribute cards to players: - The Rook deck has 57 cards (1–14 in four suits plus the Rook). According to the rules, we deal out all cards, with **13 cards to each player** (13x4 = 52) and **5 cards to the kitty (nest)** [8] . - We will implement a shuffle using a cryptographically secure random number generator to be fair. In Node, we can use `crypto.randomInt` or `crypto.randomBytes` to shuffle. - After shuffling, assign 13 cards to each of the 4 players and 5 to the kitty. Store each player's hand in the **Hands** table and the kitty in the game state (temporarily, until the bid winner claims it). - This shuffle/deal should be triggered once the game is full and teams are set (after partner selection). We have two options for triggering: 1. Automatically shuffle as soon as partner is chosen (i.e., within the choosePartner handler, after setting teams, then shuffle and save hands). 2. Or have an explicit "start game" action (maybe host triggers it). For simplicity and speed, we'll shuffle immediately after partner selection.

- Backend: Implement shuffle in the gameAction Lambda after partner selection:
- Create an array for the deck: e.g., all combinations of suits ["Red","Green","Yellow","Black"] and ranks 5-14 & 1 (or 1-14 inclusive) plus one "Rook". Actually, if we're including 1–14, that's 14 per suit, plus Rook [9] (the variant uses 1 as high card, not removed). So yes 57 cards.
- Shuffle it with a secure method (Fisher-Yates using crypto).
- Partition the deck: e.g., deck[0:13] -> player0, [13:26] -> player1, [26:39] -> player2, [39:52] -> player3, remaining [52:57] -> kitty.
- Store each player's hand in the **Hands** table: Put or BatchWrite 4 items (gameId + seat as key, with an attribute `cards`: array of card objects or strings). Possibly also store a `handVersion` or something for optimistic updates of hands if needed, but maybe not necessary.
- Update the game item's state: mark trump as not chosen yet, current turn for bidding (set to host seat 0 since they start bidding [7]), and maybe store the kitty cards in the game item (or in a separate table, but game item is fine since kitty will be claimed by winner).
- Also update game `status` to "BIDDING" (if not set already) and maybe `phase = "BIDDING"`.
- Consider storing the deck or played cards somewhere? Not really needed if we manage through moves.

- Increment version and save.

- Notify players of their hand: We cannot send other players' cards to each client (they should not see them). Each client should only get *its own* hand. This means:

- Possibly send a personal WebSocket message to each connection with their cards. API Gateway allows us to `postToConnection` to specific connectionIds. So we can loop through each player's connection and send a message like `{ action: "deal", cards: [ ... ] }` containing *only the cards for that player* (and maybe kitty count or something general).
- Or, since the client might also retrieve their hand via a reconnect or state query, we could rely on clients calling a "getHand" API. But pushing is nicer for real-time.
- We'll implement a loop to send each player's hand privately.

- Also send a general message to all that "deal is done, bidding begins" (with maybe no sensitive info). For example:

  - To each player X: send `{ action: "yourHand", cards: ["Red5", ...] }` (the full list of their 13 cards).
  - To everyone: send `{ action: "gameStart", kittySize: 5, startingPlayer: 0 }` or something to indicate bidding has started (and maybe who starts if not obvious). Everyone knows kitty size = 5 but not contents. We keep kitty contents secret, only revealed to winner.

- Frontend: Handle receiving the hand:

- On "yourHand" message, update the local state for the player's hand (replace dummy hand we had with real ones). The UI should then display actual cards. We can prettify card display later, but text is fine.
- Perhaps sort the hand for display (by suit and rank as before).
- On "gameStart" or similar message, update UI to show that bidding phase is active (e.g., show bidding controls if it's this player's turn to bid).

- Possibly remove any temporary messages like "waiting to start".

- Secure RNG: Using Node's `crypto` ensures unpredictability. We'll instruct use of `crypto.randomInt` in shuffle or similar. That addresses fairness.

After dealing, the next step is bidding which we will handle subsequently.

Prompt:

```
Continue with backend dealing logic once teams are set:

- In the gameAction Lambda (or wherever we handled "choosePartner"), after
updating teams, implement the shuffle and deal:
  - Construct the full Rook deck: for each suit ["Red","Green","Yellow","Black"]
and ranks 1 through 14, plus one "Rook" card. Represent cards as strings or
objects (e.g., "Green14", "Black1", "Rook").
  - Use a **secure random shuffle** (Fisher-Yates algorithm with
`crypto.randomInt` for index generation, instead of Math.random) to shuffle this
```

```
deck thoroughly.
  - Split the shuffled deck into 4 hands of 13 cards and 1 kitty of 5 cards.
Example: `hand0 = deck.slice(0,13)`, `hand1 = deck.slice(13,26)`, `hand2 =
deck.slice(26,39)`, `hand3 = deck.slice(39,52)`, and `kitty = deck.slice(52)`.
  - Store each hand in the DynamoDB **Hands** table. For each player (seat 0-3),
put an item: `{ gameId: ..., seat: ..., cards: [...] }`. Use BatchWrite or
separate puts.
  - Update the **Games** table item for this game: set `status = "BIDDING"`
(game now in bidding phase), set `currentBid` (or `highBid`) to 50 (minimum
starting bid) [10] or 0 if we handle minimum logic on front end, and
`currentBidder` to 0 (host starts bidding). Also include the `kitty` (store the
5 cards or at least store their count and maybe their points; but since in our
variant kitty points go to bidder upfront, we will reveal them to the bidder
later).
  - Also maybe set `trump` to null (to be chosen later by bid winner).
  - Increment the game `version` as usual.

- After dealing, send out WebSocket messages:
  - **Private hand messages:** For each player, send a message only to their
connection with their own cards. For example: `{ action: "deal", cards:
[<card1>, <card2>, ...] }` (13 cards). This allows each client to see their
hand.
  - **Game start message:** Broadcast to everyone an `{ action: "biddingStart",
startingPlayer: 0 }` or similar, to inform them that bidding phase has begun.
(We could also include maybe the minimum bid of 50).
  - Note: Do NOT send the kitty contents to anyone at this point (only the
eventual high bidder will see them).

- Frontend: Handle the new messages:
  - On receiving `action:"deal"` (your own hand), replace the player's hand in
state with the provided cards. Remove any placeholder cards. Update UI to show
these actual cards. Perhaps sort them by suit and rank as before for display.
  - On `action:"biddingStart"`, update the UI to indicate the bidding phase is
live. Possibly show a message like "Bidding starts at 50. Your turn to bid." if
the player is the startingPlayer (host). If not their turn, show "Waiting for
Player X to bid".
  - You might add a UI component for bidding: e.g., if it is this player's turn,
show buttons to "Bid 50" (or higher) and "Pass". We will implement the logic in
the next step, but go ahead and scaffold a simple bidding UI:
    * A section on GamePage for bidding with current high bid and whose turn it
is.
    * If `currentBidder` equals this player's seat, enable input to bid;
otherwise disable.
    * (The actual functionality will be added in Step 8, so it can be non-
functional yet, but the presence of the UI now is okay.)

- Ensure the cards are displayed nicely on the UI (you can reuse the card
rendering from earlier but now with real data). Possibly highlight trump once
```

```
chosen (later).
- Use console logs or UI text to debug: e.g., after dealing, print how many
cards each player got to verify.
```

Explanation: This step extends our "choosePartner" handling to immediately shuffle and deal the cards, then notify players that bidding begins.

Backend (gameAction Lambda): - We add shuffle code using `crypto`. The AI might do:

```
const deck = [];
const suits = ["Red","Green","Yellow","Black"];
for (const suit of suits) {
  for (let r = 1; r <= 14; r++) {
    deck.push(`${suit}${r}`);
  }
}
deck.push("Rook");
// Fisher-Yates:
const { randomInt } = require('crypto');
for (let i = deck.length - 1; i > 0; i--) {
  const j = randomInt(i+1);
  [deck[i], deck[j]] = [deck[j], deck[i]];
}
```

This gives a randomized `deck`. We then slice it.

- **Dealing distribution**: Confirming:
- Each player gets 13.

- 5 go to kitty.

- **Dynamo writes**:

- Possibly do a BatchWrite for 4 hand items. DocumentClient `batchWrite` can batch up to 25 items. That's fine. Or do 4 separate `put` operations in a loop. BatchWrite is more efficient; AI might choose either.
- Hand item structure: { gameId: X, seat: 0, cards: […] }. Use seat as sort key if the table has composite key (PK: gameId, SK: seat). We probably configured Hands with PK=gameId, SK=seat earlier. If not, we can use gameId+seat as a combined key, but likely composite.
- Also store kitty in game: We can store as `game.kitty = [...]`. Only the server and the eventual winner will need to know the kitty contents (the winner will get them). Possibly we might not want to expose kitty to all clients if someone were to fetch game state. But storing in DB is fine since clients don't directly read the game item except through server.
- Or we could store kitty in a separate table or with the hand of some dummy "kitty" seat. But easier: in Games, store kitty as an array.

- We also set `status = "BIDDING"`.
- Set bidding fields:
    - `highBid = 50` (minimum bid to start, or we could leave highBid null and let first bidder set 50. But usually, someone must bid at least 50 or pass. We can treat initial highBid as 0 and enforce first real bid must be >=50).
    - Alternatively, track `currentBid = 0` and define minBid separately. The rules: "bidding starts at 50" means first bid must be at least 50 [10]. We can enforce that in logic rather than storing 50 as default high bid.
    - Might be simpler to set highBid=50 and highBidder=0 (host) at the start to force others to go at least 55. But actually if host doesn't want to bid 50, they can pass.
    - So perhaps better: highBid = 0 initially, and we know minimum allowed is 50. We'll implement logic in Step 8 that if current highBid = 0, then next bid must be >=50.
    - We'll likely implement it that way.
    - Also set `currentBidder = 0` to indicate it's host's turn to act first in bidding.
- Save game with these updates and increment version.
- **WebSocket notifications**:
- **Private "deal" message**: We know each player's connectionId from Connections table (should have been stored on connect).
    - For each players (0-3):
    - Find connection for gameId & player (or seat or name). If our Connections table stores player name and seat, we can match. Possibly easier: in Connections, we stored `player` as name. Maybe we should have stored seat as well.
    - We might query by gameId to get all connections, then filter by seat or player name match to send correct hand.
    - Or if we stored seat in connection item, directly filter.
    - Once identified, use ApiGatewayManagementApi to `postToConnection({ ConnectionId: ..., Data: JSON.stringify({ action:"deal", cards: [...] }) })`.
    - This is 4 send calls.
- **Broadcast "biddingStart":** We can send to all connections a message with `action:"biddingStart", startingPlayer:0` or maybe with highBid or minBid info.
    - We have a list of connectionIds from the query above. We loop through all 4 and send this message to each.
    - Alternatively, we send it once if API Gateway had a concept of broadcast (but it doesn't natively, so we loop).
    - Possibly combine with the above loop, but keep separate logic.

If the AI is thorough, it will incorporate this. If not, we might get a partial implementation in text.

- **Frontend**:
- On receiving `{action:"deal", cards:[...]}`:
  - We update the state, maybe like `playerHand = cards`.
  - Remove dummy placeholders. We will see actual card values (like "Red7", "Yellow1", "Rook", etc.).
  - Possibly sort them. We can call a sort function or rely on the order in the message (which is random deal order, not sorted).
  - We likely want to sort for display: group by suit and by rank (with 1 as highest).
  - Could implement a custom sort: by suit (with a fixed order for suits) and by rank with 1 treated as 15 for comparison [1].
  - Or simply leave unsorted for now. But a nicer UI is to sort. We had earlier sorted dummy by grouping suits. We can reuse that logic.
  - Then re-render shows nice sorted hand.
- On receiving `{action:"biddingStart", startingPlayer:0}`:
  - We know startingPlayer=0 (host seat). If this client's seat is 0, it's their turn; else they wait.
  - We update some state like `currentBid = 0, currentBidder = startingPlayer, highBid = 0`.
  - Show the bidding controls:
  - If current player is currentBidder, enable "Bid" and "Pass" buttons, maybe a dropdown or input to choose bid amount.
  - If not, show something like "Waiting for Player [name0] to bid".
  - We might show the minimum bid (50). Perhaps display "Current high bid: none (minimum 50)".
  - If the prompt suggests scaffolding a bidding UI now: we can add a simple section: "Highest Bid: -- (Starting at 50). Your turn to bid (buttons: Bid 50, Pass)" if it's your turn, or "Waiting for X to bid" if not.
  - We'll implement actual bid logic in next step, but a basic UI now is fine.
- **Testing:**
- Without an actual backend running the logic, we can simulate the messages:
  - In the browser console (with app running), you could do something like: `window.dispatchEvent(new MessageEvent('message', { data: '{"action":"deal","cards":["Red14","Red1",...etc...]"}' } ));` That might not directly call our onmessage unless we wired it via WebSocket event. Instead, better: If our WebSocket code sets `ws.onmessage = (event) => { ... }`, we can call that function manually by retrieving `ws` from state or so. Or modify code to call the handler with a test payload. Simpler: put a temporary hack in code to call the onmessage handler with a test event after some timeout.
  - But let's assume our mental trace is enough:
  - After partner selection, the UI should soon show the player's cards.

- We expect 13 card entries replaced the placeholder backs for our hand, and others remain backs since we don't see theirs.
- A message "Bidding starts. Host to bid first." or similar should appear.

- If we had the backend local, we could try full flow:

    - Use `createGame` via HTTP (maybe using Postman with local Dynamo connected).
    - Connect websockets (maybe using wscat or writing a quick Node script to simulate clients).
    - It's complex, so likely wait until deployment to test fully.

This step completes the dealing process, so each player knows their hand and the game enters the bidding phase. All subsequent game logic (bidding, kitty exchange, play, scoring) will assume the hands and kitty are set.

Step 8: Implement Bidding Phase Logic

Now we tackle the bidding phase of Rook: - **Bidding rules recap:** Starts at 50 points minimum [10]. Players bid in increments of 5 [11]. Turn order is clockwise from the host (with our variant host starts) [7]. A player can also pass. Once you pass, you cannot bid again in that round [12]. Bidding continues until three players have passed, and the remaining high bidder wins the bid. If everyone passes without any bid (i.e., no one bids 50), typically the hand is redealt (but that's rare; we can assume someone will bid 50). - We will implement the messaging and state updates for bidding: - Clients will send WebSocket messages for their actions: e.g., `{ action: "bid", amount: 60 }` or `{ action: "pass" }`. - The backend gameAction Lambda will handle these: * Verify it's that player's turn and the bid is valid (e.g., >= 50 and >= current highBid + 5 if there is a bid). * If valid bid: - Update `highBid` and `highBidder` in the game state. - Move turn to next player (skip those who passed). - Broadcast a message to all players with the new high bid and who made it, and next bidder. * If pass: - Mark that player as passed. We can track passes by maintaining a list of passed seats or a count. - Move turn to next player (skipping passed players). - If passing causes only one player left who hasn't passed, bidding ends. Determine that remaining player as winner (highBidder). - If bidding ends: + Update game state: bidding winner, final bid amount. + Send a message that bidding is over and who won, and instruct the winner to take the kitty and choose trump. + The game then transitions to the kitty/trump phase. * Use conditional writes with version to avoid race (though bids are sequential typically). - The sequence continues until we have a winner.

- **Tracking passed players:** We can add an attribute in game state like `passes` (maybe an array of seats who passed) or just keep a count. Probably easier: an array `passedSeats` or a boolean on each player (like players[x].passed = true).
- Also track `currentBidder` (who's turn it is).

- When 3 players have passed, the remaining one is highBidder. If somehow all 4 passed (no one bid), maybe treat host as winner at 50 by default or redeal. According to some rules, if nobody bids, you could redeal. We can implement a simple rule: if no one bids (highBid stays 0), automatically assign the host with 50 or redeal. But this likely won't happen in serious play. We'll not handle redeal to avoid complexity, just assume someone bids.

- **Frontend bidding UI behavior:**

- Show current high bid and the bidder.

- If it's this player's turn:
    - Provide a control to bid a certain amount. We can simplify by offering a single "Bid +5" button that bids exactly the minimum above current, or an input to choose an amount. However, typically players might jump by more than 5 if they want.
    - Possibly allow the player to input any number >= currentHighBid+5 (or 50 if none yet) and that ends in 0 or 5 (multiple of 5).
    - But simpler: provide two buttons: "Bid [X]" where X = currentHighBid + 5 (or 50 if no bid yet), and "Pass".
    - Player can only bid exactly +5 from high bid with those buttons. This covers most scenarios, but not all (players can jump bid by more than 5). To allow jump bids, we could give a dropdown of increments or an input. Maybe an input field plus "Bid" button.
    - For now, a single increment is fine for simplicity, but we should note real game allows bigger jumps.

- If not player's turn: disable those controls and just show waiting message.

- On clicking "Bid" or "Pass", send WebSocket message accordingly.

- **Backend on "bid":** Check:

- If `amount` < 50 or not >= currentHighBid+5, respond with an error (could send back a message "invalid bid").
- If valid:
    - Update game’s `highBid` and `highBidder` to that.
    - Remove any previous idea of them having passed if they hadn't (no need to remove, just ensure they weren’t marked passed).
    - Advance `currentBidder` to next seat that is not passed and still in game. Usually next seat = (currentBidder+1) mod4, skip any who passed.

- If "pass":

    - Mark currentBidder as passed.
    - Advance to next active bidder.
    - Check if only one left not passed:
    - End bidding: highBidder wins. If highBid was 0 (no one bid at all), we might assign minimum 50 to that winner or handle it. But assume not.
    - Prepare for kitty exchange: we will handle that in Step 9.
    - We should send a message to highBidder that they won the bid and can now see kitty and choose trump after discarding 5.

- **Messages to clients:**

- On each bid: send to all: `{ action: "bidPlaced", seat: X, amount: Y }` so everyone can update display of high bid and know who bid.
- Possibly also send: `{ action: "nextBidder", seat: Z }` to indicate whose turn is next, or include it in the same message.
- On pass: send to all: `{ action: "playerPassed", seat: X }` and maybe also include `nextBidder: Z` or separate message.

- On bidding end: send to all: `{ action: "biddingWon", seat: W, amount: finalBid }`.
- Also specifically to the winner: perhaps instruct them to pick up kitty. But we can broadcast to all something like "Player W won the bid with N points. Waiting for trump selection."

- **Frontend handling:**

- On "bidPlaced": update `highBid` and highlight that player, update UI accordingly (like "Current high bid: Y by Player X").
- On "playerPassed": mark that player as out of bidding (maybe grey them out in UI).
- On "nextBidder": update whose turn indicator. If nextBidder is self, enable bid/pass buttons.
- We might merge "bidPlaced" and "playerPassed" into one "updateBid" message that includes new highBid, highBidder, who passed, etc. But separate is fine too.

- On "biddingWon": notify players that bidding ended, who won, and with what bid.
    - If you are the winner, you know you'll get the kitty and choose trump.
    - If you are not, you wait.
    - Possibly the UI can now transition to showing the kitty to the winner (the backend will send the kitty cards to the winner in next step).
    - But in this step we just mark that bidding phase over. The next step will handle kitty and trump.

We should implement as much as possible now: - The backend gameAction logic for "bid" and "pass". - Updating DynamoDB with conditional check and version increment per action. - The turn rotation logic.

Prompt:

```
Implement the bidding phase:

- **Backend (gameAction Lambda)**: Handle WebSocket messages for bidding:
  - If `action === "bid"`:
    * Parse the bid amount (e.g., `msg.amount`). Identify the player (use
connectionId to lookup their seat via Connections table, or have the client send
seat/gameId as part of message).
    * Load the game (or have it in memory from previous ops) and check that it’s
this player’s turn (`currentBidder == playerSeat`) and that the bid is valid:
`amount >= 50` and `amount >= highBid + 5` and typically `amount % 5 == 0`. If
not valid, you can optionally respond with an error message (or ignore if
invalid).
    * If valid, update the game item: `highBid = amount`, `highBidder =
playerSeat`. Do not mark them passed (they obviously are still in).
    * Reset any previous highBidder's data if needed (not really, just
overwrite).
    * Determine the next bidder: find the next seat in order (currentSeat+1 mod
4) that has not passed. (You should be tracking passed players in the game
```

```
state. Perhaps have an array `passed` of length 4 of booleans, or a list of
passed seats.)
    * Update `currentBidder` to that next seat.
    * If by making this bid the next bidder wraps around to the original bidder,
bidding continues. (We continue until 3 have passed.)
    * Save the game state (increment version).
    * Broadcast a message to all players about the bid: e.g., `{ action:
"bidPlaced", seat: X, amount: Y }` and maybe `{ action: "nextBidder", seat: Z }`
for whose turn is next.
  - If `action === "pass"`:
    * Identify the player (as above) and confirm it's their turn.
    * Update game state: mark that player as passed (e.g., add their seat to a
`passedSeats` list or boolean array).
    * Determine next bidder as above (skip over all players who are passed).
    * Check how many players are still in (not passed). If only one player
remains not passed:
       - Bidding ends. Determine `winner =` the remaining player. The `highBid`
should already be the winning bid (if no one ever bid, we might default it to 50
for the winner).
       - Update game state: set `status = "KITTY"` (phase for kitty exchange),
and store `bidWinner = winnerSeat`.
       - (Do not yet reveal kitty or set trump here, that will be next step.)
       - Broadcast to all: `{ action: "biddingWon", winner: W, amount: highBid }
`.
       - Additionally, for the winner, you can send a prompt to prepare for
kitty: e.g., `{ action: "yourTurnKitty", kitty: [5 cards] }`. Actually, we will
likely send the kitty cards to the winner immediately now.
       - For now, maybe hold off sending kitty until the next step (or go ahead
and send – see below).
    * If not ended (more than one still in):
       - Update `currentBidder` to next active player.
       - Save state.
       - Broadcast to all: `{ action: "playerPassed", seat: X }` and `{ action:
"nextBidder", seat: Z }`.
  - Ensure to use condition expressions on these updates to avoid conflicts and
increment version each time.

- **Frontend:**
  - When a "bidPlaced" message is received, update the UI:
    * Update the displayed high bid and the bidder's name. E.g., "Player X bid
Y".
    * Perhaps highlight the current high bidder.
  - On "playerPassed", mark that player as out (you could gray out their name or
add "(passed)" next to it).
  - On "nextBidder", if it's this player’s seat, enable the bidding input/
buttons; if not, disable them and show "Waiting for Player Z...".
  - If a "biddingWon" message arrives:
    * Announce the winner and bid (e.g., show "Player W won the bid with N
```

```
points!").
    * If this player is the winner, expect that the kitty will be revealed to
them next. If this player is not the winner, inform them to wait while winner
chooses trump.
    * You can clear or hide the bidding controls now.
  - (If we choose to send the kitty immediately to the winner in this step: the
backend would send a message like `{ action: "kitty", cards: [...] }` to the
winner only. In that case, handle that by showing those 5 kitty cards to the
winner in their hand and prompting them to discard. However, we might leave that
to Step 9.)

- **UI controls:** Implement the bid buttons:
  - If it’s the user's turn: show a "Bid" button and a "Pass" button. The "Bid"
button could either open an input or automatically bid the minimum (current high
bid + 5). For simplicity, you can bid the minimum increment. Maybe label it "Bid
[highBid+5]" (or "50" if no bids yet).
  - Clicking "Bid" should send `{action:"bid", amount: newBid}` via WebSocket.
Clicking "Pass" sends `{action:"pass"}`.
  - Disable these buttons when it's not the user's turn or after they act.

- Validate the increments: our game requires multiples of 5. If the user wants a
larger bid, they could click multiple times or we can allow editing the amount.
To keep it simple, one increment at a time is okay.

With this, the bidding phase should operate in real-time: players see bids, know
whose turn it is, and ultimately see who wins the bid. Next, we will handle the
kitty and trump selection by the bid winner.
```

Explanation: This step adds the dynamic bidding process.

- **Backend logic**: We update `gameAction` Lambda:
- Need to identify which player is sending the bid/pass. Likely by `connectionId` to seat mapping. Possibly store seat in the Connections table or send seat in message.
    - If we included gameId and seat in the message from front end, that simplifies identification (though trust issues, but okay).
    - We might do `const playerSeat = message.seat` or query if not provided.
- Check `game.currentBidder` or derive whose turn. Possibly track in game state `currentBidder`.
- Validate amount:
    - `if (game.highBid === 0 ? amount >= 50 : amount >= game.highBid + 5)`
    - If not, could ignore or send an error. Maybe simpler to enforce in front end, so likely minimal error handling in back end.

- On valid bid:

    - `highBid = amount, highBidder = playerSeat`.
    - Mark that player obviously hasn't passed (they wouldn't be bidding if passed).

- Possibly remove them from passed list if they were there (they wouldn't be in passed if it's their turn).
- Determine next bidder:
- `next = (playerSeat + 1) % 4`, while `next` is in passed list, do next++ mod4 until find one not passed.
- It’s possible if only one left, next loop would come back to same person, and we realize others passed? But if only one left not passed, bidding should have ended already, so probably doesn’t happen here.
- Set `currentBidder = next`.
- Save and broadcast.
- Broadcasting:
- `bidPlaced`: includes seat and amount.
- `nextBidder`: includes next seat.

• On pass:

- Mark current as passed (e.g., add to passed list or set players[current].passed = true).
- If we haven't tracked passed before, initialize in game state when bidding started: e.g., `game.passedSeats = []`.
- Determine next similarly.
- Check if `passedSeats.length == 3` (3 passed):
- If yes, find the one not in passed (that's winner).
- If no one bid at all (game.highBid still 0), we might handle that. Possibly treat the last remaining (which might be host if all passed in order) as having bid 50 by default.
  - Alternatively, we could auto-set highBid to 50 for winner if 0.
  - We'll assume someone bid to avoid complexity.
- Set `bidWinner = that seat` (we can also use existing highBidder as winner).
- End bidding phase.
- We may update `status` to something like "KITTY" or "TRUMP" to indicate next stage.
- Send "biddingWon" to all.
- Also likely send kitty to winner next step, but prompt says either now or next.
- The prompt suggests maybe sending winner kitty in next step, so for now just announce winner.
- If not ended:
- Update `currentBidder = next`.
- Save.
- Broadcast "playerPassed" and "nextBidder".

• Version control: each bid/pass update should condition check old version and increment version by 1. Possibly group multiple changes (bid placed and currentBidder change) in one update.

• The code for broadcasting probably uses similar pattern to previous steps: query Connections table for all in game to get their connectionIds, then filter if sending a personal message or send to all.

• **Frontend**:

- We now have to implement the actual event sending:
    - We have in GamePage some state for current high bid, current bidder, and passed list maybe, or we can rely on messages entirely.
    - But we might maintain some for immediate UI.
- The UI controls:
    - The prompt suggests "Bid [amount]" and "Pass" button. Good approach:
    - If no one has bid (highBid = 0): show "Bid 50".
    - If someone has, say highBid=60: show "Bid 65".
    - If current user is highBidder already, they wouldn't have a turn again unless only two left (actually if two left, they alternate, so highBidder can bid again, yes).
    - Actually, if two players left, they keep outbidding each other or pass. Our logic still holds.
    - We can keep it dynamic: "Bid [currentHighBid+5 or 50 if 0]".
    - Also maybe allow a custom input for jump-bidding:
    - It's optional; prompt says keep simple with increment.
    - We'll do increment only for now, mention that jump bidding isn't supported in this UI.
- On receiving messages:
    - "bidPlaced": update highBid and who placed it.
    - "playerPassed": mark passed (maybe in UI cross out their name).
    - "nextBidder": update whose turn. If it's me, enable my buttons; if not, disable.
    - Possibly we can combine these in one message, but likely separate events are fine.
    - The prompt instructs separate, so do separate handlers.
- On "biddingWon":
    - Display the winner and final bid.
    - If this player is winner, maybe highlight that they will get kitty.
    - Remove/hide bidding UI.
- There's a mention: possibly sending kitty to the winner either now or next step. We decide to handle in step 9. So at bidding end, from user's perspective:
    - Winner will soon see kitty and choose trump.
    - Others see "Waiting for winner to choose trump".
    - So we can prepare the UI accordingly:
    - If winner is me: maybe show a message "You won the bid! Waiting for kitty cards..." (which will come momentarily).
    - If not me: "Player X won the bid. Waiting for them to pick trump."
    - We might already have the kitty in the game item (we do from dealing). The backend should now send the kitty to the winner in next step.
    - Possibly we might decide to send kitty immediately after bidding end, but step 9 likely covers that.

- **Testing logic with an example:**

- Suppose players 0,1,2,3; host=0.
- 0 starts, bids 50.
  - backend updates highBid=50, highBidder=0, nextBidder=1.
  - sends bidPlaced(0,50), nextBidder(1).
  - UI: shows "Alice (0) bid 50", highlights Bob (1) turn.
- 1 bids 55:
  - highBid=55, highBidder=1, next=2.
  - messages bidPlaced(1,55), nextBidder(2).
- 2 passes:
  - mark 2 passed.
  - next=3.
  - 2 of 4 passed? Actually passed list [2], not ended (only 1 passed).
  - messages playerPassed(2), nextBidder(3).
- 3 bids 60:
  - highBid=60, highBidder=3, next=0 (since 2 passed, skip 2, go to 0).
  - messages bidPlaced(3,60), nextBidder(0).
- 0 (who bid 50 earlier) now gets turn again (since only 0,1,3 active):
  - 0 could bid 65 or pass. Suppose 0 passes:
  - mark 0 passed (passed list now [2,0]).
  - next=1 (skip 2,0 passed, so 1 is next).
  - still more than one not passed (players 1 and 3 remain).
  - send playerPassed(0), nextBidder(1).
  - 1's turn. Suppose 1 passes:
  - passed list [2,0,1] (3 remains).
  - Only one not passed (3), bidding ends. Winner=3, highBid=60.
  - Update status maybe to "KITTY" or "TRUMP".
  - send playerPassed(1) possibly and then biddingWon(winner:3, amount:60).
  - The sequence of whether to send a separate playerPassed for the last pass or just directly biddingWon can be considered. It's okay to send both for consistency.
  - Then proceed to kitty/trump phase.

That sequence should be handled by our logic.

- *Corner case*: If all 4 pass (no one bids 50):
- According to rules, maybe redeal or force dealer to bid 50. But we have not specified. Could treat host as winner with 50 by default. But let's ignore scenario or treat host as winner at 50 if we want to avoid endless loop.
- Possibly implement: if `highBid` remains 0 after 3 passes, then host (0) is forced as winner with 50. Or we redeal (which we won't implement).
- The simpler: assign host the bid at 50. So if `highBid` is 0 at end, do `highBid=50, highBidder=last remaining (which is host if everyone passed in first round), announce them as winner`.

- We'll not overcomplicate; assume someone bids.

- Implementation detail for passed: Maybe easier to track active players count or something. We'll do passed list as said.

- Now, our backend now covers bidding. After this, the next step will handle giving the kitty to winner and trump selection.

Integration testing: Without a live environment, limited: - We can simulate some sequences by manually calling backend functions or adjusting the game state in DB and sending events in correct order: It's complex to simulate multi-step in unit tests.

Better to rely on careful logic review and test once deployed with multiple clients or simulated connections.

Step 9: Implement Kitty Pickup and Trump Selection

Now the high bidder (bid winner) gets to exchange the kitty and declare trump: - According to rules: - The kitty (5 cards) is given to the high bidder. They add those to their hand (so they have 18 cards total) [13] . - They then must discard any 5 cards back to the kitty (to bring their hand back to 13) [14] . Usually you'd discard point-less or low cards, often you might discard stuff like 5s if they aren't points or off-suit low cards. Also, one cannot discard the Rook in some rules unless it's allowed, but our conversation didn't specify forbidding discarding points. Our house rule says any points in discard still count for bidder's team [15] , so it's okay to discard point cards. - After discarding, the bidder declares one of the four colors as trump [16] . (The Rook automatically is part of trump suit, as per variant [17]). - Trump selection must happen after seeing kitty, because kitty might influence what suit they choose. - Implementation: - Backend: * When bidding ends, we should have the `kitty` stored in game state from dealing. * Send the kitty cards to the winner via WebSocket (private message), so they can see them and decide what to discard. * The winner will then send a message e.g., `{ action: "discardAndTrump", discard: [cards...], trump: "Green" }` (listing the 5 cards they want to throw away and the trump suit they choose). - We could split it into two messages (discard then choose trump), but combining is fine and saves a round-trip. - Alternatively, have them send a "discard" message, then after an ACK or immediate, a "chooseTrump" message. But one combined message is simpler for user (they decide all at once). * The backend gameAction on receiving that: - Verify the 5 discard cards are indeed a subset of the player's cards (they should have the original 13 + kitty 5 = 18, and they must choose exactly 5). - Remove those 5 from their hand. - These 5, plus the original kitty, form the new kitty (the "discard pile"). In our rules, all points in these discarded cards count for the bidder's team automatically [15] . - We should perhaps calculate the points in those discard cards now and add to bidding team's score bank for end-of-hand scoring. However, we can also just remember to add later. We might store in game state something like `kittyPoints` or directly add to a running tally for the team. - Set the `trumpSuit` in game state to the chosen suit. - Update game status to "PLAYING" (trick play phase). - Determine who leads the first trick: rules say the player to left of dealer leads first trick [5] . But since dealer might always be host (depending if we rotate dealer), but in our variant, left of dealer leads. However, often in Rook, the high bidder gets to lead the first trick (since they won bid, they have control). * Actually official tournament rules: after trump named, the *player to left of dealer leads* [18] . In our notes, "Host starts the bid; left of dealer plays first card" was a house rule we listed. This implies the actual card play starts with the player to the left of the original dealer (which was host). * However, some Rook variants let the bid winner lead first trick, but our bullet explicitly says left of dealer leads always. Possibly they separated dealer from high bidder's rights. * To stick to our "house rules": we will have left-of-dealer (seat1 if host=seat0 was dealer) lead first trick, not necessarily the bid winner (if bid winner is not seat1). * This is unusual (commonly, bid winner leads in many games, but we'll follow bullet). *

So determine first trick leader: if we never changed dealer in this first round, dealer=0, so leader=1 (seat 1). * If we had rotated dealer in multiple rounds, it would be seat (dealer+1). * For now, assume first round dealer=0, lead=1. - Set `currentPlayer` (for playing cards) to that leader seat. - Save these updates.

```
  - Send WebSocket messages:
    + Broadcast to all that trump is chosen: `{ action: "trumpChosen", suit: <Color> }`. Everyone now knows trump suit.
    + It might be good to also tell all which cards were discarded? Typically, the discard (kitty) is kept face-down until end of hand and points just count for bidder. Our rules said kitty points belong to bidder and you do **not** have to reveal them [15]. So the discarded cards are not revealed to others (they stay unknown).
    + So we will NOT broadcast discard contents. We keep them hidden.
    + Maybe just broadcast `"trumpChosen"` and maybe something like "Game play begins, Player X leads."
    + Also broadcast or tell the leader to play. Possibly an action: `{ action: "playTurn", seat: <leader> }` or something.
    + We can combine: a `"trumpChosen"` message to all that includes who leads first trick.

  - For the winner specifically, after they send discard/trump and we process:
    + We should update their hand in the Hands table (remove the 5 discards).
    + We might send them back their updated hand (13 cards now) if the front end isn't removing those already when they selected them.
    + The front end likely will remove the discarded from UI as soon as they send them (assuming no error).
    + So maybe not needed to send an update, unless we want to confirm.
```

- Frontend:
  - When bidding winner receives the kitty:
  - We should have sent earlier a message after bidding ended to that winner: maybe `{ action:"kitty", cards:[...5...] }` (we mentioned possibly doing that in bidding end).
  - If not done yet, do it here at start of kitty phase:
    - Actually, better to have done it immediately after bidding ended. Because we want to avoid delay.
    - Possibly we should integrate that: when broadcasting biddingWon, also send a separate message to winner with the kitty cards.
    - If we didn't, we can send now with an event `'yourKitty'`.
  - The winner's UI should show those 5 kitty cards added to their hand (making 18). We should prompt them: "Choose 5 cards to discard".
  - Provide UI to select 5 out of their 18 cards (click to select/discard, maybe highlight them).
  - Once exactly 5 selected, enable a "Confirm Discard & Choose Trump" action:
    - Possibly a dropdown or buttons for trump suits (Red/Green/Yellow/Black).

            - The user picks a suit and hits confirm, sending `{action:"discardAndTrump", discard:[...5...], trump:"Color"}`.
    - If user tries to discard not exactly 5, maybe disable confirm.
    - For other players, during this time: show "Waiting for <winnerName> to choose trump." They don't see the kitty or what's discarded.
    - When a "trumpChosen" message arrives (for everyone):
    - Update UI to display trump suit (maybe show an icon or text indicating which suit is trump).
    - Indicate the next phase: trick play. Possibly show "Trump is X. [LeaderName] leads the first trick."
    - If this player is the leader, highlight that it's their turn to play a card.
    - Winner's front end after sending their discard/trump message:
    - Should remove the discarded cards from their displayed hand.
    - Possibly wait to receive an updated hand or just trust removal is done.
    - Once "trumpChosen" comes back, the winner knows trump and that game moves on.
    - Everyone sets up for play phase:
    - We might hide or disable the UI used for discarding/trump selection.
- To implement sending the kitty: We can do it as soon as bidding ends:
- In bidding end handling (above in step 8's backend, maybe after biddingWon, we could do: if (winnerSeat = W), find `game.kitty` from DB (which we saved from dealing), send to winner connection: `{ action: "kitty", cards: [...] }`. Then perhaps delete `game.kitty` from the game state or mark it taken.

Or we can do it when they send discard? But they'd need to see kitty before discarding, so yes, send immediately after bidding won.

- Summarizing:
- After bidding:
    - Backend:
    - Send kitty to winner.
    - (Game state still has it).
- Winner gets kitty -> discards & picks trump -> sends message.
- Backend on that message:
    - Remove those from hand, update game state, choose trump.
    - Could compute points of discard and add to some `pointsTeam0` etc (store for scoring).
    - Everyone: gets trumpChosen broadcast (with suit).
    - Possibly also some message "play begins, player X leads".
- Then the play phase can begin (Step 10 will handle playing cards).

Prompt:

```
Finalize the kitty exchange and trump selection step:

- **Backend:** After sending the "biddingWon" message in the previous step, send the kitty cards to the winning bidder:
  - Look up the 5-card `kitty` from the game state. Use the Connections table to find the connection for the winner (bidWinner). Send them a private message: `{ action: "kitty", cards: [ ...5 kitty cards... ] }`.
  - (Optionally, you could remove the kitty from the game item now or mark it taken.)
- Now handle the winner's response. Expect a WebSocket message `{ action: "discardAndTrump", discard: [...5 cards...], trump: "<Color>" }` from the bid winner.
  - Verify this message comes from the bidWinner (you can check the connectionId against bidWinner seat).
  - Check that the 5 discard cards are a subset of the player's hand (their hand = original 13 + kitty 5). For safety, ensure `discard.length == 5`.
  - Remove those 5 cards from the winner's hand in the Hands table (update the item for their hand: new cards array = old cards minus discards).
  - These 5 cards are now out of play. (According to our rules, any points in them count for the bidder’s team [15]. We might sum their point values here to add later to score.)
  - Update the Games table: set `trump = <Color>` (the chosen trump suit) and `status = "PLAYING"` (moving to trick-play phase). Also, set an attribute for the first player to lead. Our house rule says the player to the left of the dealer leads the first trick [5]. Assuming dealer = host (seat 0) for this hand, the leader = seat 1. (If we were rotating dealer each hand, we'd calculate accordingly, but for now assume seat1 leads.)
    * You can store `currentPlayer = 1` (the seat who will play first card).
  - Save game state (increment version).

  - Broadcast to all players:
    * `{ action: "trumpChosen", suit: "<Color>" }` to inform everyone of the trump suit.
    * `{ action: "playStart", leader: <seat> }` to inform who starts the first trick.
    (Alternatively, combine these into one message with both trump and leader.)
  - The bid winner does not need a separate message with their updated hand, since they know what they discarded (the front-end can remove those cards from UI). However, you could send them an update if desired (not strictly necessary).

- **Frontend:**
  - When the bid winner receives `{ action: "kitty", cards: [...] }`:
    * Append these 5 cards to their hand. The UI should now show 18 cards for them.
    * Highlight/prompt: "You won the bid! Select 5 cards to discard." Provide UI
```

```
for selecting exactly 5 cards out of their hand. Maybe allow clicking cards to
"move to kitty". Show a count of selected cards.
    * Also provide trump selection UI: perhaps a set of buttons or a dropdown
listing the four suits (Red, Green, Yellow, Black). This can be chosen after or
alongside discarding.
    * Only enable the "Confirm" button when exactly 5 cards are selected and a
trump suit is chosen.
    * On confirm, send the `{ action: "discardAndTrump", discard: [ ...5
chosen... ], trump: "Color" }` message via WebSocket.
  - Other players when receiving "biddingWon":
    * Show a message like "Player X won the bid, awaiting trump selection." They
do nothing during this time.
  - After the winner sends their choice, all players receive `{ action:
"trumpChosen", suit: "<Color>" }`:
    * Update the UI to indicate the trump suit (e.g., display an icon or text of
the trump color).
    * The bid winner's hand in UI should now be back to 13 cards (the front-end
should remove the 5 discards on their side when they sent them, so it should
already show 13).
  - All players then receive `playStart` (with leader seat):
    * Identify which player is leading the first trick. If it's you, highlight
that it's your turn to play a card. If it's not you, show "Waiting for Player Y
to lead."
    * The game now moves to the trick-playing phase. Ensure the UI is ready for
card play: for example, enable the hand cards to be clickable for the player
whose turn it is, etc. (We'll implement actual card play in the next step.)
  - The trump suit should be visibly indicated on the interface (maybe by
coloring the trump suit or a label "Trump: Green", etc.).

- Ensure that players who are not the bid winner never see the kitty cards or
the discarded cards (they remain hidden). They only learn the trump suit.

At this point, the bid winner has integrated the kitty into their hand,
discarded 5 cards, and declared trump. The trump suit is announced to everyone,
and the first trick is ready to be played by the designated leader.
```

Explanation: This finalizes the transition from bidding to play: - We send kitty privately and handle the discard + trump from winner.

- **Backend details:**
- Sending `{action:"kitty", cards:[...5...]}` to winner:
    - We have `game.kitty` from when dealing (should store as array of 5 strings).
    - We use Connections table to find connection of seat=winner (like query by gameId and filter by player seat or name).
    - Then `postToConnection` with that data.

- On `discardAndTrump`:
    - Identify player, ensure it's bidWinner (we have `game.bidWinner` or `game.highBidder`).
    - Validate 5 discards in their hand:
    - We need their hand. Either:
        - We can fetch from Hands table (get item by gameId & seat).
        - Or we could have cached their full 18 cards in game state (not likely, we kept them separate).
    - Safer: fetch from Hands table the winner's hand item, which should contain the 13 original (if not updated after kitty) plus the kitty? Actually, when we dealt, we gave them 13. We haven't updated their hand with kitty on the server side yet.
    - Realize: when kitty was sent, on server side we didn't automatically put those cards into their hand in DB. We should:
        - Possibly update their Hands table entry by adding kitty cards to it when bidding ended. That would reflect they have 18.
        - Or at least we could handle it in the discard message: just consider their hand = DB hand + game.kitty for validation.
    - Implementation: simpler: at bidding end, before sending kitty, update the winner's hand item to include the kitty. So that now their hand record has 18 cards.
    - That means performing an Update on Hands table: add each kitty card to array. DocumentClient supports adding elements to list? Possibly using `SET cards = list_append(cards, :kitty)` if using DocumentClient and lists.
    - That might be good to do.
    - If we do that, then at `discardAndTrump`, we can get their hand from DB and validate removal.
    - Alternatively, we trust the client selection (risky if malicious, but this is a game scenario, not high security).

    - But let's validate anyway.

    - After validation:

    - Remove those from their hand item: update DB to remove those 5 from the list of cards (the order doesn't matter but easier to filter out by values).

    - Now their hand item should have 13 cards again.

    - We might or might not want to store the discard somewhere. We could just throw it away, but maybe record it for scoring:

        - Our rules say any points in discard count for bidding team. So we should accumulate that.
        - We can sum points: Use scoring logic:
        - 5s = 5, 10s = 10, 14s = 10, 1s = 15, Rook = 20 [19] [20].
        - Add to e.g. game.teamScores[winnerTeam] kittyPoints.
        - Or store `game.kittyDiscarded = [cards...]` to maybe sum later.

        - Simpler: could calculate and store `game.kittyPoints` in game state as those points to add to team at scoring time. Possibly add to a field like biddingTeamPointsEarned = that sum.
        - We'll leave the actual adding to total score to the end of hand scoring step, but store for now. Or even simpler, just remember to include them when calculating final score.
        - We'll not dive into actual scoring code now, but mention the idea.
    - Set `trumpSuit`.
    - Determine leader for first trick:
        - Based on left-of-dealer, which is seat1 if dealer=0.
        - If dealer was always seat0 this round, leader = 1.
        - If we had rotated dealer, we'd need to know dealer. But since we didn't implement rotation (score game to 500 across multiple hands, we might do in Step 12, maybe not).
        - We can derive dealer if we stored something like `game.dealerSeat` which we didn't yet. Possibly assume host = dealer always in first hand.
        - We'll proceed with leader = 1 for now as per rule.
    - Save all updates in one or two writes:
        - Could do a TransactWrite: update game (set trump, currentPlayer, etc) and update Hands (remove cards).
        - Or do sequential: update hand, update game.
        - It's fine either way.
    - WebSocket notifications:
        - `trumpChosen` to everyone with suit.
        - `playStart` with leader seat (and maybe name).
        - Possibly only to leader also a prompt "Your turn to play first card".
        - But we can handle turn via common `currentPlayer` state that front end uses.
- **Frontend**:
    - On "kitty": For bid winner:
    - We already have their 13 in state.
    - Append the 5 kitty. That may require adjusting how state is stored: e.g., we had `playerHand` array in state.
    - Now do `setPlayerHand([...playerHand, ...kittyCards])`.
    - UI should update showing 18 cards (we might need CSS to allow wrapping to multiple lines, etc., since more cards).
    - Provide selection UI:
        - Perhaps overlay the kitty cards with a highlight or treat them as part of hand normally (makes no difference for discarding).
        - Let player click 5 cards. We mark selected maybe by adding a CSS class.

    - After each click, count selected; once 5 selected, if they click another we might require deselect first.
    - After selecting 5, choose trump:
    - Could have four suit buttons: Red/Green/Yellow/Black.
    - The user likely picks a trump that they have strong cards in.
    - Press "Confirm" triggers WebSocket send.
  - The user should maybe not be allowed to confirm until exactly 5 are selected and a trump is chosen.

  - Possibly display a reminder of point values or not needed.

  - On "trumpChosen" (all players):

  - Everyone displays "Trump: <Color>" prominently.
  - Possibly could recolor trump suit cards in everyone's hand UI for clarity (like outline them).
  - No one sees the discards or kitty except winner (and winner only saw kitty, not final discard set aside).

  - The winner after sending discards will likely remove them from UI themselves. They could implement that immediately on selection confirm (assuming success).

    - Or wait for backend confirmation message. But since we likely won't send back "discard accepted", we can assume once sent, they remove them. If any error, backend might send an error message (we didn't plan one).
    - We'll assume success and remove immediately.

  - On "playStart" with leader:

  - If leader seat equals your seat, highlight "Your turn to play a card".
  - If not, show "Waiting for <leaderName> to lead".

  - We may set a state `currentPlayerTurn = leader` similar to bidding's currentBidder logic, and reuse logic for enabling card clicks.

  - The actual playing of cards will come in Step 10, but we position the UI accordingly:

    - Possibly show a placeholder area for trick in the center.
    - Leader will have to choose a card to play. We can allow them to click a card now to play it. But we might implement that in Step 10.
    - For now, maybe just note it's their turn.

This completes the setup for trick play.

At this point, all players know trump, and the first player to act is set. The next step would be to implement playing cards and trick logic.

We have to ensure that at the end of this step: - The bid winner's front end successfully integrated kitty and removed discard. - Everyone knows trump suit via "trumpChosen". - Everyone knows who leads via "playStart".

Testing: We can test parts: - Simulate kitty message to winner in browser dev console by calling the onmessage function or simulating event: e.g., `ws.onmessage({ data: JSON.stringify({ action:"kitty", cards:["Red5","Green14",...]}) });` Check that the app appended those to user's hand and showing 18 cards.

- After selecting 5 and trump, ensure the code sends the correct message and no UI errors. We cannot fully test server handling here.
- Simulate receiving trumpChosen: e.g., `onmessage({ data: '{"action":"trumpChosen","suit":"Green"}' })` on all. Check UI indicates "Trump: Green".
- Simulate playStart: e.g., for everyone: `onmessage({ data:'{"action":"playStart","leader":1}'})`. If current user seat is 1, ensure the UI says it's their turn. If not, says waiting.

We then consider step 10 will handle playing out the tricks.

Step 10: Add Card Play Validation and Turn Logic

Now we implement the trick-playing phase: - **Rules for playing cards**: - A trick is led by one player who plays any card from their hand (any suit or trump) [21] . - Other players in clockwise order must follow suit if they have a card of the led suit [22] . If they do not have a card of the led suit, they may play any card (either throw off another suit or play trump) [22] . - If a player holds the Rook and trump is led (trump suit), the Rook is considered part of trump suit and must be played if they have no other trump (Rook is lowest trump) [17] . Essentially Rook follows trump suit. - If a non-trump suit is led and a player only has the Rook in that suit (no card of led suit, Rook is not of that suit because Rook is only trump), they can play Rook as an off-suit or as a trump if they choose to treat it as trump. Actually, under our variant Rook is always trump color, so if led suit != trump and they have Rook but no led suit, they can play Rook (counts as trump) or any other. - No restriction on leading trump; you can lead trump anytime [21] . - Trick winner: the highest card of the led suit wins the trick, unless any trump was played, in which case the highest trump wins [23] . Under our variant, Rook is a trump but ranks lowest in trump, so any other trump (2 and above) beats Rook [17] [24] . - The winner of the trick leads the next trick [25] . - This continues for 13 tricks (since each hand after discards has 13 cards per player). - Implementation: - **Backend (gameAction Lambda)**: Handle an incoming `{ action: "playCard", card: "<Card>" }` message: * Determine the player (connection -> seat). * Validate it is that player's turn (`currentPlayer` in game state == seat). * Validate the card is actually in that player's hand (check Hands table for their cards contains that card). * Determine the suit of the card and the current trick state: - If this is the first card of a new trick (no one has played yet in this trick): + Set `trickSuit` = suit of the played card (or "Trump" if they played a trump or Rook). + Mark that suit in game state (maybe store `currentTrick = [{seat, card}]` and also store `ledSuit`). - If not first card: + Check `if suit of played card == ledSuit or player has no card of ledSuit`. If player has a card of ledSuit but played something else, that's a rule violation (reneging). We should ideally prevent it. + We can enforce by scanning their hand for any card of ledSuit: * If found and they didn't play ledSuit, reject the move (could send an error message to client). * We'll assume players will follow suit since our UI can enforce it too by disabling illegal plays. + Append their card to `currentTrick` list in game state. - Remove the played card from their hand in the Hands table. - Broadcast to all: `{ action: "cardPlayed", seat: X, card: "<Card or maybe hidden??>" }`. + Important: Other players should see the card face-up because once played, everyone sees it. + So yes, we broadcast the actual card. - Update `currentPlayer` to the next player (unless this was the 4th card of the trick). + Next player = (currentPlayer + 1) mod 4, skip none because in play all 4 participate unless someone is out (no one is out during a hand). + If the trick isn't complete, broadcast `{ action: "nextPlayer", seat: Y }` (or we can infer from cardPlayed sequence). * If that was the 4th card played in the trick: - Determine winner of trick: + Evaluate `currentTrick` list (4 entries). Determine ledSuit from first card if not stored, and trump suit is known (game.trump). + Find highest card: # If any card's suit == trump (including Rook treated as trump suit), the highest trump wins. (Compare rank by treating 1 as highest rank (15) and Rook as rank 0 perhaps since it's lowest trump). # If no trump played, highest card of ledSuit wins (again 1 is high). + We need a rank ordering: maybe map rank: 1->15, 14->14,...5->5, Rook->0 if we consider Rook as numeric for comparison. Actually Rook is only in trump suit context. + We'll handle Rook: If Rook was played and another trump was played, that other trump wins because Rook is lowest trump. If Rook is the only trump, and someone played off-suit because they had no led suit, Rook would win the trick because a trump was played. + So treat Rook as rank 0 in trump suit. + Or handle separately: If comparing two trumps and one is Rook, non-Rook wins unless only Rook present. + We'll code accordingly. - Determine winner seat. - Update game state: increment that team's point tally with any points in the 4 cards of the trick: * Sum card points from those 4 cards (5s=5, 10s=10, 14s=10, 1s=15, Rook=20). * Add to some running tally in game (like `teamPoints`). * Also, if these were the last trick of the hand (if players now have no cards left after this trick), note that the winner of last trick also gets the kitty points (but in our variant we already gave kitty to bidder's score automatically via discard rule). * Actually, official rules often say winner of last trick gets the kitty (points in kitty). But we gave those points to bidder as per house rule, so nothing extra for last trick here. - Clear `currentTrick` (for next trick). - Set `currentPlayer = winnerSeat` (they will lead next). - Broadcast to all: `{ action: "trickWon", winner: W, points: P }` maybe including points captured in that trick. - If that was the 13th trick (i.e., each player has now played all cards, hands table should be empty for all): + The hand (round) is over. Move to scoring: # Calculate total points each team got (sum of points from tricks plus any kitty/discard points for bidder). # Determine if bidWinner's team met their bid: - If yes, they score the points they made. - If not, they score **negative** their bid (penalty) [26], and the defending team scores the points they made. # Also check "shoot the moon" bonus: if bid winners took all tricks (opponents score 0), give them 200 points instead of actual 180 (our house rule) [19]. Or if in any scenario all points captured by one team, we apply that rule. # Update cumulative scores (the game is to 500). # Set `status = "HAND_COMPLETE"` or something. # Broadcast a message to all with the hand results: points for each team, updated total scores, maybe who won the game if someone >=500. # If no team reached 500, prepare for next hand (which might involve rotating dealer and starting a new round with shuffle). + We'll implement scoring and next hand in Step 12. + For now, we can broadcast a simple end-of-hand summary and mark game over or waiting for next hand.

- **Frontend**:
- On receiving "cardPlayed":
    - Update the UI to show that card on the table (in the center area representing the trick). We might have placeholders for 4 positions on table corresponding to each seat position.
    - Remove that card from that player's hand in the UI (if we were showing other players' hand backs, we can just decrement their card count or remove a back).
    - Show the actual card face for everyone (everyone sees it).

- On "nextPlayer": highlight that seat as having turn.
  - If it's this player, enable clicking a card to play.
  - If not, ensure this player's hand is not interactive.
- We may not need a separate "nextPlayer" if we infer turn by tracking sequence of cardPlayed messages: but easier to get explicit.
- On "trickWon":
  - Display a message like "Player X won the trick and earned Y points."
  - Possibly clear the table (remove the four cards from UI after a short delay).
  - We might animate the cards going to winner's pile, but not needed now. Just clear them.
  - Update any score display if we show running points (some UIs might not show until end of hand, but we could).
  - Set up for next trick: the winner will lead next (we know from trickWon who won).
  - We can internally set `currentPlayerTurn = winnerSeat`.
  - Perhaps directly call the logic to highlight that it's winner's turn.

- If the hand ended (13 tricks done, everyone out of cards), we expect a message summarizing hand (like "Team A scored X, Team B scored Y").

  - Show those results.
  - Possibly show overall scores (if playing to 500).
  - If someone reached 500, show game over message "Team X wins the game!"
  - Otherwise, prepare UI for next hand or allow starting next hand (maybe the host can press a "Next Hand" button).
  - We will implement that in Step 12.

- We need to enforce follow-suit on front-end too:

- When it's the player's turn and a suit was led (if they are not the leader, `ledSuit` is known from game state or from first card of trick):

  - We should highlight which suit is led (e.g., show an icon of led suit).
  - In player's hand, disable cards that are not of led suit if they have at least one card of led suit.
  - If they have none of led suit, they can play anything.
  - Also, Rook is treated as trump suit always in our variant.
  - So if ledSuit = "Green" and the player has some Green cards:
  - They must play Green, so disable all non-Green *including trump or Rook* for them.
  - If ledSuit is a non-trump and player has none of that suit:
  - They can play any (including trump or Rook).
  - If ledSuit = trump and the player has trump cards (including Rook as trump):
  - They must play trump; if they have Rook and other trumps, they can choose any trump (like Rook or a higher trump; presumably they'd not throw Rook if they have another trump since Rook is low).
  - If they have no trump, they can play anything.
  - The front end can compute this by looking at player’s hand and comparing suits.
  - Mark cards as disabled or hide them so they cannot click an illegal card.
  - This helps enforce the rule so the backend rarely gets an illegal move.
  - We still keep backend check for safety.

- For Rook specifically:

- We treat Rook as suit = trump in our logic. So front-end:

    - If ledSuit = trump and player has Rook, that counts as having trump (so they must play trump, and Rook qualifies as a trump to play).
    - If ledSuit = trump and player plays Rook while having a higher trump, that's allowed but a poor play. They might prefer to throw Rook only if they have no other trump because Rook will lose to any other trump. But that's strategy, not rule.
    - So we allow Rook as an option when following trump suit if they have it (like any other trump).
    - If ledSuit = not trump and they only have Rook (no card of ledSuit), they can play Rook as an off-suit (trump) card. That is allowed.
    - Our enforcement covers that: if they have none of led suit, everything (including Rook/trump) is enabled, which is correct.

- Trick winner determination:

- We'll need to implement rank comparisons:

    - Possibly create a helper function in code to get numeric value:
    - If card is Rook: return maybe a special marker.
    - If card suit == trump:
        - if card is Rook: treat as rank 0 (lowest).
        - else if card rank number: treat 1 as 15, others as face value for now (since 14->14, 13->13...5->5).
    - If card suit != trump and != led suit (and it's not Rook because Rook is trump anyway): they can't win if any trump or any led suit present.
    - We'll approach differently: gather all trump cards played, if any exist, ignore all non-trump for winning.
        - Among trumps, highest wins (with Rook lowest).
        - If no trump played, consider only led suit cards: highest rank wins (with 1 high).
    - We must know led suit:
        - We store it when first card of trick is played, in game state `currentTrickSuit` or so.
    - We'll implement accordingly.

- We will also accumulate points:

- Each trick's points = sum of point cards in it (5,10,14,1 and Rook).

- Add to the team that captured trick:

    - Determine team membership: we know teams from earlier partner selection (players have team attribute or team arrays).
    - If we stored teams in game state or players array with team property, use that.
    - If we didn't store explicitly, we can determine team:

    - If host picks partner step, we could have stored an array or in each player in DB a `team` number.
    - Probably we did store something like players list with team or separate teams field.
    - Let's assume we know team of each seat (0 & chosen partner vs others). Possibly store in game item: `team0Seats = [a,b], team1Seats=[c,d]` or similar.
    - Or simpler: store in game something like `players: [{seat:0, name:..., team:0}, {...}]`.
    - If we did that at partner selection, we can retrieve from game state for scoring.
    - We'll assume we have that.

- We accumulate in game state maybe a running tally like `teamScoresThisHand = {0: X, 1: Y}` for trick points.

- At end-of-hand, these plus kitty points (already counted or separate) will be used.

- At the very end of game (after 13 tricks):

- The prompt hints at scoring and game end:
    - We'll implement that in Step 12, but can stub:
    - After trick 13, broadcast maybe something like `{ action: "handComplete", team0Points: A, team1Points: B, bid: N, bidTeam: T (which team was bidder) }`.
    - Possibly also a message if game continues or game won.
    - We'll cover actual computing in Step 12.
    - For now, maybe just determine that if it's end of first hand, show points and maybe indicate another hand.

So Step 10 will likely implement up through trick outcomes and notify when hand ends, deferring final scoring to later.

Prompt:

```
Implement the trick play (card play) logic:

- **Backend (gameAction)**: Handle a message `{ action: "playCard", card:
"<Card>" }` from a player:
  - Determine the player's seat from the connection (or message).
  - Verify it is that player's turn (`currentPlayer` in game state equals this
seat).
  - Fetch the current trick info from game state (who has played what so far in
this trick, and what the led suit is if any).
  - **Validation**:
    * Check the card is indeed in that player's hand (look up their hand in the
Hands table). If not, ignore or send error.
    * If this is not the first card of the trick (i.e., `ledSuit` is already set
in game state):
       - Determine `ledSuit` (suit of the first card played in this trick,
```

```
stored in game).
       - If the player has any card of `ledSuit` in their hand and the card they
are trying to play is NOT of `ledSuit`, then this is an illegal play (reneging).
You can reject it (and maybe send an error message back). Otherwise, it’s fine.
         + (Note: Rook is considered part of trump suit always [17] , so it does
not count as a card of other suits for following. So if ledSuit is hearts (say)
and player only has Rook and no hearts, they can play Rook.)
    * If validation passes:
       - Remove the card from the player's hand in DynamoDB (update the Hands
table to delete that card from their list).
       - Add this play to the game state's current trick info. If this is the
first card of the trick, set `ledSuit` in game state to the suit of this card
(unless this card is the Rook, in which case `ledSuit = trump` because Rook is
effectively trump).
       - Append `{seat: X, card: "<Card>"}` to a `currentTrick` list in game
state.
       - Broadcast to all players: `{ action: "cardPlayed", seat: X, card:
"<Card>" }` so they see the card on the table.
       - Determine next turn: If this is not the 4th card of the trick yet:
          * Compute `nextSeat = (X + 1) mod 4` (loop to next player). If any
players are out of cards (shouldn't happen until trick 13 is over), skip them.
          * Update `currentPlayer = nextSeat` in game state.
          * Broadcast `{ action: "nextPlayer", seat: nextSeat }`.
       - If this was the 4th card of the trick (everyone has played one):
          * Determine the winner of the trick:
            + Get the `ledSuit` from game state.
            + Identify trump suit from game state (`game.trump`).
            + Among the 4 played cards:
              - If any card's suit is trump (or is Rook, which counts as trump)
 [17] , the highest trump wins the trick. (Remember, in our variant the Rook is the
lowest trump [17] , so if someone played Rook and another played a 5 of trump, the
5 wins because it's a trump higher than Rook.)
              - If no trump was played, the highest card of the ledSuit wins.
Rank order: treat 1 as highest (above 14) [27] , then 14, 13, ... down to 5.
              - We can implement this by assigning values: for trump cards, use
base value (e.g., 5->5, ...14->14, 1->15, Rook->0) and for led suit cards
similarly (but non-led, non-trump cards cannot win over any led or trump).
            + Determine winner seat.
          * Assign trick points: calculate points in these 4 cards (5s=5, 10s &
14s=10, 1s=15, Rook=20). Add these points to a running tally for the winner's
team in game state.
          * Clear the `currentTrick` list (or move it to a trick history if
needed) and reset `ledSuit` for the next trick.
          * Update `currentPlayer = winnerSeat` (winner will lead next).
          * Broadcast to all: `{ action: "trickWon", winner: winnerSeat, points:
<sum> }`.
          * Also broadcast `{ action: "nextPlayer", seat: winnerSeat }` to
indicate the next leader (or combine this info with trickWon).
```

```
          * Check if that was the last trick of the hand (each player should
have played all 13 cards now):
            - You can detect by checking if the Hands table for all players are
empty or if you've completed 13 rounds.
            - If hand is over:
              + Determine total points for each team (use the tallies from trick
wins plus any kitty/discard points for the bidding team).
              + Compare against the bid: see if the bidding team made at least
their bid.
              + Update cumulative scores in the game (game.teamScores).
              + Determine any bonuses: if a team captured all tricks (opponents
0 points), apply the 200-point "shoot the moon" rule [19] .
              + Determine if the game is over (team reaching ≥500 points).
              + Broadcast a summary to all, e.g., `{ action: "handComplete",
team0Points: X, team1Points: Y, team0Total: A, team1Total: B, bid: N, bidTeam:
<0 or 1>, madeBid: true/false }`.
              + If game continues, maybe allow a short pause then automatically
start the next hand or prompt for next hand (we can handle starting next hand in
the next step).
              + If game over, broadcast `{ action: "gameOver", winner: "Team <0
or 1>" }`.

- **Frontend:**
  - On "cardPlayed":
    * Display that card on the table (e.g., show the card image or text at the
position corresponding to the player’s seat). Remove that card from that
player's hand UI. If it's your hand, it disappears (because you played it). If
it's another's, you can just reduce the count or remove one back from their hand
display.
    * Because the card is revealed to all, show its value. We might arrange 4
placeholder slots for the trick (top, left, right, bottom relative positions).
  - On "nextPlayer": highlight the next player’s turn. If it's you, enable your
cards for clicking. If it's not you, ensure your hand cards are disabled.
    * Also, maintain the rule enforcement: if a trick is in progress (ledSuit
known) and it’s your turn, only allow clicking cards of that suit if you have
them. Disable others. (The backend also checks this, but frontend makes it user-
friendly.)
  - On "trickWon":
    * Show a message like "Player X won the trick and got Y points." You might
briefly highlight the winner.
    * Clear the table of the 4 played cards (after a short delay to let players
see final trick).
    * Possibly update a running score display for teams (optional to show mid-
game, or just end of hand).
    * The `nextPlayer` message (or the info in trickWon) will indicate who leads
next; the UI should then highlight that player as the next leader (likely your
next "nextPlayer" event already does this).
  - If "handComplete" arrives:
```

```
    * Display the points each team earned that hand, and cumulative scores.
    * Indicate if the bid was made or set (e.g., "Bid team made their bid" or
"Bid team failed and gets -N points").
    * If the game is not over, prepare for next hand (maybe a "Next Hand" button
for the host or automatically reset after a short break). Possibly reset all
hands and UI for new deal.
  - If "gameOver" arrives:
    * Announce the winning team and perhaps disable further input. You can show
a congratulations message and maybe prevent any further gameplay.

- **Frontend rule enforcement** (on player's turn):
  - If `ledSuit` is known for the current trick and the player has cards of that
suit, highlight those and disable cards of other suits (except Rook counts as
trump suit, not as a off-suit).
  - If player has no cards of ledSuit, they may play anything (enable all).
  - Do similar for trump: if ledSuit = trump and player has trump (including
Rook as trump), they must play trump.
  - This ensures the player can only click legal cards. (The backend still
double-checks for validity.)

At this point, the trick-taking phase should play out correctly: players take
turns playing cards, following suit as required, and the system determines trick
winners and tallies points. When all tricks are done, scores are updated and the
game either proceeds to the next hand or ends if someone reached the victory
condition.
```

Explanation: This step is the most complex, covering the entire play of tricks and scoring of a hand.

Given the length, the AI might not fully implement scoring logic details (like points calculations) explicitly, but should outline them.

The prompt covers: - Back-end `playCard` handling (with suit following, removal of card from hand, etc.) - Trick resolution and broadcasting trick results. - End-of-hand scoring and possibly game termination or next hand. - Front-end updating display on each event: - Display played cards to all. - Indicate whose turn. - Clear trick after it's won. - Show scores and final results.

We need to ensure: - Did we store team info properly to compute team scores? The prompt references "bidTeam". - We likely need to know which team was the bidding team: * If host and partner are team0 by default, maybe team0 = bid team if one of them was high bidder, or team1 if not. * Actually, easier: We know which player (seat) won the bid. We can determine that player's team and call that "bidTeam". * Possibly store `bidTeam` or `bidWinnerSeat` in game state from earlier steps.

- Points calculation:
- Should implement mapping:
    - Rook = 20, 1 = 15, 14 & 10 = 10, 5 = 5, all others (2-4, 6-9, 11-13) = 0.

- Summing for trick and also summing for total.

- Team score update:

- We maintain teamPoints in game for the current hand. Possibly as part of game state or ephemeral.
- But since we can derive from stored captured trick points at end, we may not need to store ephemeral. We can just accumulate in a variable inside Lambda as it processes.

- But since Lambda is stateless (each action separate), better to store partial points in game item after each trick maybe:

  - e.g., `game.teamPointsHand = {0: X, 1: Y}` updated each trick.
  - Or store cumulative points for each team as attributes in game item.
  - This could be updated in trickWon handling.

- End-of-hand:

- We likely determine end-of-hand if each player's hand is empty. We could check if 13 tricks have been played. We might track trick count in game state.

- Maybe simpler: After each trick, check if the hands table for any player is empty. Actually check one player or track a counter:

  - If trickCount == 13 (because each of 4 players had 13 cards).
  - We can store a counter or just infer if `players[0].cards` is empty after trick resolution, that means done (provided we always remove cards).

- Then do scoring:

  - We know:
  - `bidWinnerSeat` and their team,
  - `bidAmount` (the final highBid from game state),
  - teamPoints from `teamPointsHand`.
  - Also kitty/discard points we may have added to bid team's points when discarding (if we did).
  - Actually, if we didn't explicitly add those, they might already be included:
    - If the bidder discarded point cards, we didn't give those to any trick, but our rules gave them to bidding team automatically. So we should add those to teamPoints of bid team now.
    - Possibly we stored them separately as `kittyPoints`.

  - If not, we can recalc: those 5 discard cards plus original kitty if any points, all go to bid team. We can sum them now and add to bid team's points.

  - Then:

  - If bid team's points >= bid, they get all their points.
  - If bid team's points < bid, they "go set": they score -bid points (subtract bid from cumulative), and they do NOT get the points they collected (some rules give them 0 plus -bid, effectively same as just -bid, which is what our PDF says [26]).

- Defenders always keep whatever points they made in tricks.
- So:
  - If made bid: bid team score = points they made; other team score = points they made.
  - If failed bid: bid team score = -bid (not minus points, just minus bid, ignoring what they collected), other team score = points they made (and plus any points bid team collected? Actually if bid fails, the convention is bidding team gets -bid, and doesn't get their points; defending team still gets the points they took) [26] .
  - So effectively, if fail, defending team gets their points, bidding team gets negative bid (the points they took are disregarded).
- "Win-all bonus": if one team captured all points (a sweep), e.g., defenders got 0 points:
  - House rule: if bidding team sweeps, give them 200 points [19] . If defenders sweep (bid team got 0), no specific bonus mentioned, but likely just they get 180 (since all points are 180).
- We can implement: if bid team took all tricks (opponents 0), set their hand score to 200.

- Note that if defenders took all (bid team 0), the bid team definitely failed and gets -bid, defenders get their 180 normally (no bonus given by house rules, presumably).

- Add these hand results to game's cumulative score (game.teamScores total).

- Check if either total >= 500:
- If both >=500, the one with higher wins (rare simultaneous).

- Determine winner team if game over.

- Save the updated cumulative scores and maybe set status to "GAME_OVER" if done or "NEXT_HAND" if continuing.

• Broadcast `handComplete`:

- Provide points of this hand for each team, and new total scores.
- Also say whether bid was made or set (maybe implicit by negative or not).
- Possibly name the bid amount and who had it.
- The front-end will show summary.

• If game continues:

- Possibly automatically start next hand by shuffle again. Or we might wait for a prompt (maybe host clicks "Next Hand" or automatically after a delay).
- We'll likely implement next hand in Step 12.
- For now maybe just signal end of hand.

• If game over:

- Broadcast `gameOver` with winning team or names.
- Possibly end connections or disable input.

Front-end: - On `handComplete`: * Show a summary overlay: "Hand results: Team0 got X, Team1 got Y. Bid was N by Team T, they [made/failed] it." * Show updated total scores for both teams. * Possibly highlight who is in the lead. * If game not over, maybe a "Next Hand will begin shortly..." message.

- On `gameOver`:
  - Display "Team <X> wins the game!" and maybe list player names of that team.
  - Possibly offer a restart or exit option (not required by user story, but nice).
- The UI resets for next hand if any:
  - Clear all trick area, reset hands (we will get a new deal soon).
  - Possibly clear state like passed, trump, etc.
  - If next hand starts automatically, the previous logic (Steps 4-7) would run again to create new game state and so on.
  - We handle that in next step.

We have thus built the core game loop.

Steps 11 and 12 might involve managing score tracking, game loop to next hand or finishing.

Prompt covers up to game result announcement.

Step 10 is done.

We should now finalize the answer, ensuring all sections are properly formatted, comprehensive, and sources cited appropriately.

Let's check citations usage:

We have integrated citations inside explanation: - [27†L23-L31] for Rook lowest trump and must follow trump. - [35†L87-L95], [35†L91-L99], [35†L99-L107] for follow suit and trick rules. - [29†L13-L18] for 1 as highest rank. - [25†L180-L188] for kitty points to bidder. - [25†L188-L196] for bid fail penalty. - [25†L199-L207] for 200 points sweep. - [32†L178-L187] and [32†L189-L197] for dealing distribution and bidding increments (some overlap with [35]). - [11†L277-L284] for fixed seating (though we did variant partner selection). - Possibly [32†L202-L210] used in context of who leads first trick (there it says left of dealer leads). - Already in step 9 we cite [35†L80-L87] for left-of-dealer leads first trick.

We should ensure each citation appears at least once (not repeated unnecessarily). The final text likely already has them scattered in earlier steps. Steps 8-10 rely on previous citations but not many new needed except maybe one to confirm trick points or Rook as lowest trump which we did with [27] and [35].

Check if any new citations needed: - Possibly one for point values? We have [32†L169-L177] listing counters (it mentions 1s, 5s, 10s, 14s, Rook). Or [32†L170-L178] in wiki: That portion says: "Only certain cards have a point value. Each 5 is 5 points, each 10 and 14 is 10 points, and the Rook is 20 points. Each 1 is 15 points [20]." We should cite that when we mention point values, but I see the prompt doesn't explicitly cite one.

Let's incorporate: When describing trick points summing, maybe add a citation: e.g., "Sum points (5s=5, 10s & 14s=10, 1s=15, Rook=20) [20] ."

The prompt touches point values in trick resolution explanation: it says "5s=5, 10s & 14s=10, 1s=15, Rook=20". We should cite [32†L169-L177] there to support official scoring.

Yes, let's add it: In trick resolution: "calculate points in these 4 cards (5s=5, 10s & 14s=10, 1s=15, Rook=20) [20] ."

We should ensure not to cite from multiple sources for the same fact redundantly. It's fine to have multiple references though if needed.

Double-check: - [35†L73-L81] might mention discarding equal number of cards and declare trump (the dealing section mention). Actually [35] lines 73-81 talk about kitty: "The bidder will pick up those kitty cards and then discard equal number... After discarding, bidder declares trump color..." [14] . We should cite that in step 9: Step 9's text: "discard an equal number of cards and declare trump." we should reference [35†L73-L81] to show that was described.

Yes, add a citation in step 9 explanation where it says "discard and then declares the trump suit" with [35†L73-L81].

Check step 9 text: it likely has that concept but didn't cite: "In our rules, any counters in them count... must discard 5 and declare trump." Possibly add [35†L73-L81] after "discard an equal number of cards from their hand and then name the trump" in step 9 explanation (if present).

Alright, I'll finalize with all these elements combined into final output.

Note: This is a comprehensive step-by-step guide. It assumes familiarity with basic programming and AWS services. You can follow these prompts sequentially in Kiro CLI to build the project. Maintain the context (especially for backend prompts, so the code integrates with previous steps). Adjust any configuration variables (like API URLs or AWS resource names) as needed for your environment.

Building a Real-Time 4-Player Rook Card Game with Kiro CLI

1. Dev Environment Setup

Before coding, ensure your environment is ready:

- **Node.js and npm:** Install Node.js (v14 or above) – this provides npm for package management.
- **AWS Account:** You will be deploying to AWS Lambda, API Gateway, and DynamoDB. Make sure you have AWS credentials configured (e.g., via `aws configure`) and appropriate permissions to create resources.

- **Kiro CLI or Cursor IDE:** Make sure Kiro CLI (or your chosen AI coding assistant environment) is installed and authenticated. This will allow you to send the provided prompts and have it generate code.
- **Project Structure:** Create a project directory, e.g. `rook-game`. We will have two main parts: a React frontend and a Node.js backend. You can keep them in one repository (e.g., `frontend/` and `backend/` subfolders).
- **Frontend Dev Server:** We'll use React's dev server (`npm start`) for front-end testing. For the backend, we will rely on AWS deployment (though you can test some logic locally with Node).
- **Tools:** Optionally, have `wscat` (WebSocket CLI) to test the WebSocket API, and an HTTP client (Postman/curl) for the REST API. These help in manual testing of backend endpoints.

With this in place, let's start coding step-by-step.

2. Prompt Engineering Notes

When using Kiro CLI (or Cursor) to generate the code, consider these guidelines:

- **Follow Order:** The prompts below build on each other. Execute them in sequence. The code from earlier steps is context for later steps.
- **Precision in Prompts:** Each prompt clearly specifies what to do. When asking Kiro to create or modify files, mention filenames and relevant content locations if needed. This ensures the AI knows where to put things.
- **Review AI Output:** After each prompt, review the code. Fix small issues or refine the prompt if the output isn't as expected. The guide tries to be precise to minimize misinterpretation.
- **Testing Iteratively:** After major steps, run the application (frontend or backend) to verify. For example, after setting up the React app, run it to ensure it compiles and the basic UI loads. After deploying the backend, test endpoints with sample requests. This iterative testing will catch issues early.
- **Maintain Context for Backend:** The backend prompts assume you keep the previous backend code in context. When you switch to frontend prompts or vice versa, ensure the assistant still has the relevant files open or provided, so it integrates changes correctly.
- **No Hardcoding Secrets:** We will sometimes use placeholder values for things like WebSocket URL or API endpoints. Later, you'll replace these with real values (e.g., after deployment, plug the actual WebSocket API URL into the frontend).
- **Error Handling:** The prompts include basic error handling (like validating moves in the game). The AI will implement checks and may log errors. You can enhance error handling as needed (e.g., more user feedback on invalid actions) once the basic flow works.

With these in mind, let's begin.

3. Step-by-Step Implementation with Kiro CLI

Step 1: Initialize React Project and Basic Routing

First, we create the React frontend and set up pages for the lobby and game table.

Prompt:

```
# Use Kiro CLI to initialize a new React app and set up basic pages/routes.

Create a React app (with TypeScript) in a `frontend` folder named "rook-game-
frontend". Use `create-react-app` or similar.

Once created:
- Add **React Router** for routing.
- Create two pages/components: `LobbyPage` for the path "/" and `GamePage` for
path "/game".
- In `LobbyPage`: create a form or simple inputs for entering a player name and
a game code, plus two buttons: "Create Game" and "Join Game". (No functionality
yet, just UI and event handlers stubbed out.)
- In `GamePage`: for now, just display a placeholder text like "Game Room" and
maybe a back link to go home.
- Set up the router in `App.tsx` to navigate between Lobby and Game pages.
- Ensure the app is styled as a mobile-friendly PWA:
  - Use a meta viewport for mobile.
  - Add a basic manifest (name "Rook Online", theme color, etc.) and ensure the
service worker is registered (you can use CRA's default service worker setup).
- The LobbyPage UI:
  - An input for "Name" (text).
  - An input for "Game Code" (text).
  - A "Create Game" button.
  - A "Join Game" button.
  (We'll hook these to backend later.)
- Use basic CSS to make it look okay on mobile (e.g., inputs and buttons full-
width or large enough to tap).
- Confirm that `npm start` runs without errors and the two routes render
correctly (you can navigate to "/" and "/game" to test).
```

Explanation: This prompt initializes our React project and sets up routing and basic UI. After running this: - The `frontend/rook-game-frontend` app will be generated. - We add React Router (`BrowserRouter`) wrapping our app. - We create `LobbyPage` and `GamePage` components, and configure routes in `App.tsx` (or `App.js` if not TS). - The UI in LobbyPage includes text inputs for player name and game code, and two buttons (Create, Join). Initially, these buttons will not do anything when clicked (we will implement their logic in Step 6 when integrating the backend). - We ensure the app is PWA-ready: CRA usually includes a service worker and manifest. We might need to uncomment or adjust service worker registration (CRA's template often registers it by default or provides an opt-in). - We set a viewport meta tag in public index.html for mobile scaling. - CSS: We probably let CRA default styling be minimal. The prompt suggests making inputs/buttons fill width on mobile; the AI might include some simple CSS (maybe in App.css or via inline styles) to make the lobby form vertically stacked and responsive.

After Kiro CLI generates this: - **Testing:** Run `npm install` and `npm start` in `frontend/rook-game-frontend`. Verify: - The app loads at `http://localhost:3000`. - The lobby page shows inputs and buttons. - Clicking "Create Game" or "Join Game" likely does nothing (maybe just a console.log from stub handler). - Navigating to `/game` route shows "Game Room". - The PWA manifest is present (check the HTML head or `manifest.json`). - There are no errors in console.

At this point, the frontend skeleton is ready for further development.

Step 2: Render Player Hand and Table Layout (Game UI)

Next, we build out the game table UI in the GamePage. This will include placeholders for players' hands and the playing area.

Prompt:

```
# Enhance the GamePage UI to show a 4-player table layout with hands and table
area.

In `GamePage.tsx` (or JSX), implement a basic layout for a 4-player Rook game:

- Represent 4 player positions: your own hand at the bottom, opponent hands at
the top, left, and right.
- For now, use dummy data for cards:
  - Assume each player has 13 cards. For the current player (bottom), create an
array of 13 dummy card identifiers, e.g.
`["Green14","Green10","Green5","Green1","Black1","Red14","Red5","Yellow1","Yellow10","Yellow5","R
(mix of suits and the Rook).
  - For opponents, we won't show actual cards, just a placeholder like a back-
of-card or count of cards.
- Layout:
  - Use a `<div>` or container for the table center (this will later show played
cards).
  - Arrange the player hands around:
    * Bottom (current player) hand: display the dummy cards in a row. Possibly
use small `<div>` or `<span>` per card (just showing e.g. "Green14" or an emoji/
text).
    * Top player: show say 13 facedown cards or an indicator like "13 cards".
    * Left and Right players: similarly show vertical or rotated card backs or
counts.
  - Label each position with a placeholder name or "Player 1/2/3/4". The host
(current user) can be Player 0.
  - Use CSS for positioning: perhaps a flex column for top-bottom, and absolute
positioning or a grid for left/right. Simpler: you can center the table area and
have hands positioned relative.
  - Ensure the bottom hand (yours) is prominently visible and styled for mobile
(maybe wrap cards if too many for screen width).

- Add a section on the side or bottom to display game info:
  - Trump suit (once chosen), Current high bid, Scores, etc. (Just placeholders
```

```
now: e.g., "Trump: TBD, Score: 0-0").

- The opponent hands can simply be represented by "[??]" or card-back emoji
repeated N times. (Text like "🂠" for a card back or a Unicode playing card
back).

- Add this styling to CSS:
  - `.card { display: inline-block; padding: 4px; margin: 2px; border: 1px solid
#ccc; border-radius: 4px; }` for card placeholders.
  - Maybe different background colors for different suits (optional).
  - `.hand { margin: 10px; }` for spacing each player's hand area.

- This is mostly static at this point for UI demonstration. We'll make it
dynamic later.

Ensure that the GamePage renders without errors with this dummy layout.
```

Explanation: Here we create a visual layout for the game table: - We treat the game as 4 fixed positions (0=bottom (you), 1=right, 2=top, 3=left, for example). The prompt might use another ordering but typical clockwise is bottom (self), left, top, right (or similar). They said "opponents top, left, right" which implies: - bottom = current player (player 0), - top = opposite (player 2, partner if fixed teams? Or just opponent), - left = player 3, - right = player 1 (depending how we assign). - They label them "Player 1/2/3/4", might do 0-3.

Given the PDF note that host is seat0 and partner seat2 if fixed, it fits with positions (0 bottom, 2 top). But since we allow host pick partner, these labels are temporary.

- The dummy hand for bottom has actual sample card names including "Rook". This tests how we might display Rook.
- The dummy cards likely will just be text like "Green14". We can style them.
- Opponent hands: possibly show card backs. The prompt suggests using a Unicode playing card back "🂠". That's a nice idea (🂠 might not render on all systems, but it's Unicode for a card back). Or just show "XX XX XX..." or a number "13 cards".
- CSS: They define a `.card` class to style individual card placeholders. Also `.hand` maybe for container.

We need to ensure mobile layout: - The bottom hand will be long, so maybe it wraps if too wide. - We can allow it to wrap by CSS flex-wrap or just letting inline-blocks wrap by width.

The AI might produce a simple layout: Perhaps something like:

```
<div className="game-table">
  <div className="opponent top hand">🂠🂠🂠... (13 times)</div>
  <div className="middle-row">
    <div className="opponent left hand">🂠... (13)</div>
    <div className="table-center">Table Center (played cards here)</div>
    <div className="opponent right hand">🂠... (13)</div>
```

```
  </div>
  <div className="player bottom hand">
    {dummyCards.map(card => <span className="card">{card}</span>)}
  </div>
</div>
```

and some CSS to position: - `.game-table` could be display flex or grid with rows for top, middle, bottom. - Middle row having left and right with space between.

This will give a crude but visible layout: - Top row: 13 card backs horizontally centered. - Middle row: left and right columns with card backs (maybe vertically stacked or just as text in a rotated container?), and a center area for the table where played cards go. - Bottom row: our cards in a scrollable or wrapping container.

We also add a placeholder for game info (like a div showing "Trump: TBD, Current Bid: 0, Score: 0-0"). We will update these later as game progresses.

Testing after this: - Refresh the game page, now it should display the full layout with dummy values. Ensure nothing breaks. - Likely no interactivity yet beyond possibly scrolling if content overflows.

This sets up the front-end game UI, which we'll later populate with real data and actions.

Step 3: Add WebSocket Connection Logic (Client Side)

We need real-time communication, so we set up a WebSocket in the frontend to connect to our backend's WebSocket API. For now, we'll use a placeholder URL and manage connect/disconnect events.

Prompt:

```
# Add WebSocket connection handling in the React frontend (GamePage or
globally).

In the React app, set up a WebSocket connection to the backend:
- We'll eventually have an AWS API Gateway WebSocket endpoint. For now, define a
constant `WS_URL` (placeholder) in a config file or inside GamePage (e.g.,
`const WS_URL = "wss://<placeholder>"`). We will replace this with the actual
URL after deployment.
- Establish the WebSocket connection when the GamePage mounts:
  - Use `useEffect` in GamePage to open `new WebSocket(WS_URL)` when component
mounts, and close it on unmount.
  - Store the WebSocket instance in state or a ref (so we can use it to send
messages later).
- Handle WebSocket events:
  - On open: console.log or display "Connected".
  - On close: if it disconnects, maybe attempt reconnection or show
"Disconnected". (We can do simple log or UI indicator.)
  - On message: parse the JSON message. For now, just log it or set some state
```

```
for testing (later, we'll handle specific messages).
- Provide an indicator in the UI for connection status (for example, a small
green dot or "Connected" text when WebSocket is open, red or "Disconnected" when
closed).
  - You can manage a state `connected` (boolean) that updates on open/close, and
display it in GamePage (maybe in the game info section).
- For testing, add a temporary button on GamePage: "Test Ping". When clicked, if
WebSocket is open, send a test message, e.g. `ws.send(JSON.stringify({ action:
"ping", data: "hello" }))`. (This presumes backend might echo or handle it; it's
just to see that sending works without crash.)
- Ensure the WebSocket connection uses query params to identify the player:
  - We will have player name and possibly game ID. If available (maybe stored in
localStorage or context from the Lobby join step), append them to the URL, e.g.
`wss://.../Prod?gameId=<id>&playerName=<name>`. (If not available yet, leave it
and we'll add when integrating with backend in later steps.)
- Clean up on unmount: close the WebSocket to avoid memory leaks (use
`ws.close()` in the cleanup function of useEffect).

Implement this and test in the browser. (It won't fully connect until we have a
real endpoint, but you should see the console log "Connecting..." then likely an
error because placeholder URL is not real. That's fine. We mainly want no
runtime exceptions in our code.)
```

Explanation: We incorporate client-side WebSocket logic: - Likely put in GamePage since that is the main game screen needing it. Alternatively could use a context or hook global, but GamePage is fine for now. - Steps: - Introduce a constant `WS_URL`. Without a real endpoint, maybe we put something like `const WS_URL = "wss://example.com/rook"`. The prompt says <placeholder> and mentions stage Prod – implying how an API Gateway URL looks (wss://xyz.execute-api.region.amazonaws.com/Prod). - On mount, create `ws = new WebSocket(WS_URL)`. - Use `ws.onopen = () => setConnected(true)` etc., and `ws.onmessage = e => ... JSON.parse(e.data)`. - If we had gameId and playerName from the lobby, we'd attach as query params. However, at this point in the guide, we haven't implemented storing these from the lobby. We will do that in Step 6. So for now, we might leave query empty or assume dummy values. - Show status: the guide suggests a small UI indicator (maybe in the game info bar). - Could be text "● Connected" or just a colored dot. - We'll likely keep it simple: e.g., a `<span>{connected ? "Connected" : "Disconnected"}</span>`.

- Test Ping button:
- Great idea for debugging. The backend currently has no ping route, but if it's open, it might just get 404. It's fine.
- We can still click it to ensure `ws.send` doesn't throw (it might if connection isn't ready or closed).
- Possibly guard: only allow if `ws.readyState === WebSocket.OPEN`.
- The code should handle the scenario that our placeholder isn't reachable. That will trigger onerror or immediate close. We might want to handle onerror as well:
- Could set connected false on error. Or just let close event handle it.

- For now, the console will show error "cannot connect". That's expected until we have actual URL. We just ensure our code doesn't crash from it.

After implementing: - **Testing:** Launch app, open console: - You should see attempt to connect (and likely error or close event). - Perhaps log messages: "WebSocket connected" won't appear due to error, but "WebSocket disconnected" might if we coded onclose. - The UI might flash "Disconnected". - The Test Ping button if clicked might error since no connection or not open; maybe our code should check if `ws.readyState === 1` (OPEN) before sending. If we didn't, clicking might throw but we can catch that in onerror anyway. It's minor since no real connect.

Anyway, once we deploy and plug actual WS URL, this code will handle real events from backend.

Step 4: Create Backend API for Game Creation and Joining

Switching to backend: we implement Lambda functions for creating a game and joining an existing game.

Prompt:

````
# Set up the Node.js backend with Lambda functions for createGame and joinGame, and define the DynamoDB schema.

1. Initialize a Node.js project in a `backend` folder (if not already). Create a `package.json` and install `aws-sdk` (for DynamoDB access) if needed.
2. **DynamoDB Tables**: We will use three tables:
   - `Games` table (primary key: `gameId` (string)). Stores overall game state and metadata.
   - `Hands` table (primary key composite: `gameId` (string) and `playerSeat` (number or string)). Stores each player's hand of cards.
   - `Connections` table (primary key: `connectionId` (string) or composite `gameId` + `connectionId`). Stores active WebSocket connections (to know which connection IDs belong to which game/player).
   We'll create these tables externally (via CloudFormation or AWS Console), but our code will refer to them by name.
   For now, define environment variables or constants for table names (e.g., `GAMES_TABLE`, `HANDS_TABLE`, `CONNECTIONS_TABLE`) so we can easily change them.

3. Create a file `dbConfig.js`:
   - Import AWS SDK DynamoDB.DocumentClient.
   - Export a configured DocumentClient instance.
   - Export the table name constants (either from env or hardcoded for now, like:
     ```js
     const GAMES_TABLE = process.env.GAMES_TABLE || "Games";
     ```
   - We will use these in our Lambda handlers.
````

```
4. **createGame Lambda** (`createGame.js`):
   - This function will be triggered via HTTP API (POST /createGame).
   - Input: It should get a player name from the request body (the host's name).
   - It generates a new unique gameId (e.g., a short code). For simplicity,
generate a random 6-digit or 4-letter code. (Use a simple function or random
bytes and toString(36), etc.)
   - Create a new item in `Games` table with key `gameId`. Attributes to
include:
     * `gameId` (string code)
     * `players` (array with the host as first entry, e.g., `[ { seat: 0, name:
playerName } ]`. We'll assign seats 0-3.)
     * `status` (e.g., "LOBBY" initially, meaning waiting for players)
     * `hostName` (perhaps store separately)
     * `createdAt` (timestamp)
     * Maybe `version` (start at 1) for optimistic locking on updates.
     * Team info not yet (teams decided after partner chosen).
     * (We can add more fields later as needed.)
   - Put this item into DynamoDB (use DocumentClient.put).
   - Return a JSON response with `gameId` and perhaps `playerSeat` (0 for host).
   - (No authentication, so we trust the name input.)

5. **joinGame Lambda** (`joinGame.js`):
   - Triggered via POST /joinGame. Input: a `gameId` and a `playerName`.
   - Look up the game in `Games` table (DocumentClient.get).
   - If not found, return an error (404).
   - If found, check how many players so far. If >=4, return an error (game
full).
   - Determine the next available seat number (e.g., if players array currently
length N, assign seat = N).
   - Update the game item: add the new player to the `players` array. Also, if
now 4 players, you might update status to "FULL" (ready to start game).
   - Use a conditional update to avoid race conditions: e.g., include a
condition that `attribute_not_exists(players[3])` or a version check.
Alternatively, use DocumentClient.update with an expression to push to the
players list only if size<4.
   - (You might use `SET players = list_append(players, :newPlayer), version =
version + 1` with condition `size(players) < 4`.)
   - Return success with assigned seat and current list of players (so client
can see who joined).
   - Also, if game became full (4 players), you might trigger game start later
(but we'll handle game start separately in subsequent steps).

6. Create an **AWS Lambda handler wrapper** for each (since API Gateway uses
Lambda Proxy integration):
   - Each handler function should export `exports.handler = async (event) =>
{ ... }`.
   - Parse `event.body` (JSON) for input.
   - Perform operations as above.
```

```
   - Return an object `{ statusCode: 200, body: JSON.stringify(response) }` (and appropriate headers for CORS if needed, like `Access-Control-Allow-Origin` *).

7. Use the `dbConfig.js` DocumentClient in these handlers.

Make sure to `module.exports` or `exports.handler` properly for deployment.

We'll deploy these Lambdas and test them via HTTP in later steps. For now, just implement the logic and ensure no syntax errors.
```

Explanation: We set up the Node.js backend with two API endpoints and outline DynamoDB tables: - We initialize a Node project with aws-sdk (the AWS Lambda environment already has aws-sdk v2, but including it locally is fine for testing). - We define table names and DocumentClient in `dbConfig.js` for reuse. - We write `createGame.js` Lambda: - Random gameId generation logic (maybe use `Math.random().toString(36).substr(2, 6)` or `uuid` but short code is fine). - Put to Games table: * `gameId` as key. * `players: [ { seat: 0, name } ]` * `status: "LOBBY"` * `version: 1` (for optimistic locking usage). * Possibly `hostName` or reuse players[0]. * Could also store `gameCode` if different from gameId (not needed, we can use gameId as code). - On success, respond with gameId and seat 0.

- `joinGame.js` Lambda:
- Input from body: gameId, playerName.
- Get Games item:
  - If not exists, error (we'll return 404).
  - If exists:
  - Check `players` length. Let n = length.
  - If n >= 4, return error (maybe 400 "Game full").
  - Determine seat = n (so if one player, seat1; if two, seat2, etc.).
  - Append to `players` list the new object `{ seat: seat, name: playerName }`.
  - If after adding, n+1 == 4, we might mark status "FULL".
  - Use DocumentClient.update:
    - Key: gameId.
    - UpdateExpression to push to list: e.g., `SET players[$size] = :newPlayer, #v = #v + :one` (where we use a version attribute to increment for concurrency) or use list_append.
    - ConditionExpression e.g. `attribute_exists(gameId)` AND `size(players) = :prevCount` to ensure no other join happened concurrently (or use version check).
    - Alternatively, simpler: do a get first and then put with condition or transaction, but update with condition is fine.
  - We will likely let the AI choose approach. Possibly it will do a get then put (less efficient but straightforward).
  - After updating:
    - Build response: seat, maybe the updated players list or at least the list of names (so UI can show existing players).
    - For now, perhaps return the whole players array so the client knows current players.

- Return status 200 with JSON.

- We mention also environment variables: likely in Lambda deployment we will set GAMES_TABLE, etc. For local testing, we can default them as above.
- Make sure to handle exceptions (like conditional check failing) by catching and returning appropriate status.
- No WebSocket involvement yet here, that's later. But we note that if game becomes full, maybe we'll trigger game start manually (in our flow, the host will pick partner after join, etc.).

After writing these: - **Testing (the code)**: - We can simulate locally by calling the handler functions with dummy events: * `createGame.handler({ body: JSON.stringify({ playerName: "Alice" }) })` (with proper environment or client configured to local Dynamo if available). But without a running DynamoDB, it won't succeed. We can rely on deploying and testing on AWS, or use DynamoDB Local if we configure it. * The main thing: ensure no syntax errors or reference errors. - The AI should produce correct usage of DocumentClient and updates.

We will later deploy and verify with actual AWS resources (Step 15 covers deployment).

Step 5: Define Game State Schema (DynamoDB Tables and Data Model)

We've partially defined our tables and data structures. Now ensure the schema and any additional fields are in place for game logic synchronization and concurrency control:

Prompt:

```
# Refine the backend data model and prepare for game state management.

Update the backend configuration and Lambdas to incorporate a version number for
optimistic locking and to prepare our game state for real-time updates:

- In `Games` table items, ensure we have a numeric `version` attribute:
  - The version will start at 1 on game creation. We will increment it with each
game state update (join, partner selection, bid, etc.) [3] . This helps ensure we
don't overwrite changes out-of-order.
  - Our Lambda updates (like joinGame) should include a condition like `version
= :expectedVersion` and then do `SET version = version + 1` in the update
expression.
  - Adjust `createGame` to initialize `version: 1`. Adjust `joinGame` to use and
increment version (get the current version from getItem result, use in
ConditionExpression and increment).

- Team assignment: After all 4 players join, we will have teams. By default (if
host picks partner later), we won't assign teams yet in the DB until partner
selection. But if we were doing fixed teams (player0+2 vs 1+3) [6] , we could
store that. For now, plan to store teams after partner selection step.
```

```
- Prepare the `Hands` table usage:
  - We will store each player's cards in `Hands` with `gameId` + `playerSeat`. Initially, no cards are dealt until game start, so we might not create Hand entries until shuffle.
  - But we can create empty hand items on game creation or when players join, if desired (e.g., an empty `cards` array for each). This is optional; we can also create them at deal time.
  - Decide to create `Hands` items when dealing (in a later step). So no action needed now, but keep `HANDS_TABLE` reference in config for future use.

- Prepare the `Connections` table usage:
  - We will populate this when players connect via WebSocket. We'll handle that in WebSocket connect/disconnect Lambdas later. For now, just ensure `CONNECTIONS_TABLE` is in config.
  - Typically, `Connections` table might use `connectionId` as the primary key (and store gameId, playerSeat, etc. as attributes). Or use a composite key with `gameId` as partition and `connectionId` as sort, to easily query all connections in a game [4]. We lean toward gameId as partition so we can query by game to broadcast messages.
  - So define in code (not creating table, just usage assumption): `Connections` table with partition key `gameId` and sort key `connectionId`. We'll use this pattern in our WebSocket lambdas.

- Create placeholders for WebSocket event lambdas:
  - Make files `connect.js` and `disconnect.js` for the $connect and $disconnect routes of the WebSocket API.
  - In `connect.js` handler:
    * When a client connects, API Gateway provides `event.requestContext.connectionId` and any query params (like gameId, playerName).
    * Implement: parse `gameId` and `playerName` from `event.queryStringParameters`.
    * Use DocumentClient.put to add an item to Connections table: with key (gameId, connectionId) and attributes like `playerName` and maybe `seat` if we can get it (we might not know seat from query alone at connect time; perhaps we include seat in query too or look up by name in game).
    * For now, store at least gameId -> connection mapping.
    * Return { statusCode: 200 }.
  - In `disconnect.js` handler:
    * On disconnect, we get connectionId. We need to remove that connection from Connections table.
    * We might not have gameId in the event for $disconnect. If not, we may need to design our table key differently or store a mapping of connectionId -> gameId elsewhere. A simpler way: make `connectionId` the primary key (no sort) in Connections table, and store gameId as an attribute. Then:
      - On connect: put item { connectionId, gameId, playerName, seat }.
      - On disconnect: do DocumentClient.delete({ TableName: CONNECTIONS_TABLE, Key: { connectionId } }).
```

````
    * We'll go with connectionId as the primary key for simplicity (though querying all by gameId will require a secondary index later, but we can manage that).
    * Implement disconnect to delete by connectionId.
    * Return { statusCode: 200 }.

    (We'll revise this schema if needed when broadcasting messages.)

- Adjust `dbConfig.js` to export any new table name or index if required.

- No changes to createGame/joinGame logic besides adding version checks:
  - In joinGame, use `ConditionExpression: version = :v` and increment version by 1 in UpdateExpression [3].
  - Handle ConditionalCheckFailedException by returning an error (409 conflict maybe) if version mismatch (meaning someone else updated game concurrently).
  - Similarly, if two joinGame calls happen, one will fail the condition and can return an error "please refresh or try again".

Implement these changes. We aim to ensure game state consistency and prepare for upcoming real-time operations.```

**Explanation:** We refine backend for concurrency and schema completeness:

- **Versioning (Optimistic Locking)** [3]:
  - Add `version` to game items. createGame sets version=1.
  - In joinGame:
    * When retrieving game, get its version (say `v`).
    * Use update with ConditionExpression `version = :currV` and ExpressionAttributeValues `:currV = v` (the expected current version), and also maybe ensure not full.
    * And in UpdateExpression do `SET players = list_append(players, :newPlayer), version = version + :one` (with `:one = 1`).
    * If condition fails, we catch ConditionalCheckFailed and return an appropriate message.
  - The AI will implement something along these lines, using DocumentClient's condition.
  - If not comfortable with list_append due to type issues, they might do a Get then Put. But prompt encourages condition usage.

- **Team assignment**:
  - The prompt acknowledges possibly storing teams after partner selection. So no code needed now beyond acknowledging plan.

- **Hands table**:
  - We hold off creating hand items until dealing. Possibly mention that in code comments.

- **Connections table schema**:
````

```
  - They debate using gameId as partition vs connectionId as partition. For
broadcasting to all in game, having gameId partition is useful [4]. But
$disconnect event lacks gameId directly, only connectionId.
  - One solution: use connectionId as PK and gameId as attribute, and have a GSI
on gameId for broadcasting queries. Possibly too detailed for now.
  - They choose simplest: connectionId as PK (so easy to delete on disconnect),
but note that to broadcast by game we'll need either scanning or a GSI. It's
fine; we can refine when implementing broadcast.
  - So:
    * connect.js:
      - parse gameId, playerName from event.queryStringParameters (we will
ensure the client passes these).
      - Put item: { connectionId, gameId, playerName, (and maybe seat if
passed) }.
    * disconnect.js:
      - get connectionId from event.requestContext, do
DocumentClient.delete({ connectionId }).
    * They caution if we want gameId as key, more complicated to get on
disconnect. We'll go with connectionId as PK as per prompt, which is simple.

- They mention maybe seat in connect: If client query passes seat too, that
would help. We might do that. The prompt suggests possibly storing seat as well
for clarity.
- They also mention environment table names again in dbConfig.

- Modify joinGame to bump version and condition on version:
  - Suppose initial version=1. After join, version=2, etc.
  - If two join at once, one will fail version cond and we return error (client
could then fetch updated players if needed).

- Possibly also add a condition to ensure players < 4 already. But we can just
check in code.
  - The AI might simply do code logic check plus version cond (which ensures no
other update in between get and update, effectively preventing double join
conflict).

We ensure to mention referencing AWS conditional write source:
- We cite [3] for optimistic locking concept.
- The prompt references it.

After implementation:
- Not easy to "test" concurrency without actual environment, but check that
joinGame now expects version in state.
- We'll trust this until integration testing after deployment.

### Step 6: Implement Player Join Flow & Partner Selection

Now that the backend join logic exists, integrate it with the frontend and add
```

````
the partner selection feature:

**Prompt:**

```text
# Wire up front-end create/join to the backend and implement partner selection
UI and logic.

**Frontend (Lobby -> Game flow):**

- In LobbyPage, implement the onClick for "Create Game":
  - When clicked, make a POST request to `/createGame` REST API. Use `fetch`
with the player name from input.
  - The API endpoint (once deployed) will likely be something like `https://
<api-id>.execute-api.<region>.amazonaws.com/<stage>/createGame`. For now, define
a `API_BASE_URL` constant (or use a placeholder). We can set `API_BASE_URL` in a
config file (similar to WS_URL) and leave it empty or a dummy for now.
  - Example:
    ```js
    const res = await fetch(API_BASE_URL + "/createGame", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ playerName: nameInput })
    });
    const data = await res.json();
    ```
  - Expect `data.gameId` and maybe `data.seat` in response.
  - On success, store the gameId, player name, and seat in local state or
global. For simplicity, use `localStorage` to store these values (so that on
GamePage we can retrieve).
    - e.g., `localStorage.setItem('rookGameId', data.gameId)`, same for name and
seat.
  - Then navigate to `/game` (use `useHistory` or `<Redirect>`). The GamePage
will then load and use the stored info.

- Similarly for "Join Game":
  - Make POST to `/joinGame` with { gameId: codeInput, playerName: nameInput }.
  - Handle errors: if game not found or full, alert the user with message from
response (the backend might send 404 or 400 with an error message).
  - On success, store gameId, player name, and the seat returned.
  - Navigate to `/game`.

- Ensure to handle loading state / disable button while waiting for response,
and basic error UI.

**Frontend (GamePage)**:
- When GamePage loads (useEffect on mount), retrieve gameId, playerName, and
seat from localStorage (those were saved on lobby action).
````
````

```
- Use these to establish the WebSocket connection with query params:
  - e.g., `new WebSocket(WS_URL + "?gameId=" + gameId + "&playerName=" + encodeURIComponent(playerName) + "&seat=" + seat)`.
  - This will let the backend know which game and who this connection is.
- Also, fetch initial game state:
  - We might call a GET `/gameState?gameId=...` endpoint to get current players or status. (We haven't implemented such an endpoint explicitly, but the joinGame response gave us players list.)
  - Alternatively, we can use the data from join response:
    * If this user created the game, they know only themselves initially.
    * If joined, the joinGame response might have included current players.
  - For now, to get current players in game (names, etc.), we can cheat by storing the `players` array from joinGame response in localStorage as well and reading it.
    - Or call a new API (if we implement a quick GET game state lambda).
    - To keep it simple, modify joinGame to return the updated players list and store it.
  - Use this list to display player names on GamePage (like in a sidebar or table info area). E.g., show "Players: Alice (host), Bob, ..." etc.
  - Also, if the game is now full (4 players), the UI should prompt for partner selection.

- **Partner Selection:**
  - We assume fixed teams? Our requirement says host picks partner. So implement:
    * If current player is host (seat 0) and game status is "LOBBY" or waiting (players <4 -> just wait for more players, if players ==4 now, need to pick partner).
    * When 4 players are present (players.length === 4) and teams not chosen yet, show a UI for the host to select a partner:
      - Perhaps list the other 3 players with a "Choose Partner" button next to each name.
      - Host clicks one, we send a WebSocket message, e.g., `ws.send(JSON.stringify({ action: "choosePartner", partnerSeat: X }))` where X is the seat of the chosen partner.
    * Disable partner buttons after clicking (to avoid double send).
    * Other players (not host) when game is full but partner not chosen should see a message "Waiting for host to select partner".

- **Backend (WebSocket "choosePartner"):**
  - We need to implement the logic in our WebSocket message handler (likely in gameAction lambda which we'll do in a later step). For now, just plan:
    * When server receives `choosePartner` from host, it will set teams: host (0) and that chosen partner form Team0, the other two form Team1 [6] .
    * It will update the Games item with team assignments. For example, add `players[].team` fields or a `teams` structure.
    * Then broadcast via WebSocket to all players something like `{ action: "partnerSelected", partner: chosenSeat }` or perhaps the full team info.
```

```
  - For now, simulate this on frontend:
    * When host sends choosePartner, you can optimistically update UI: mark the
chosen partner on host's UI as "Partner".
    * The actual confirmation will come from server (we'll handle in backend
step).
    * On receiving a partner selection message (we'll implement handling in the
onmessage):
      - If `action == "partnerSelected"`:
        + Mark teams in the UI (e.g., color-code names or list team members). We
know host and chosen are Team A, others Team B.
        + Proceed to next stage (which will be dealing cards).
      - Remove the partner selection UI once done.

- Update the GamePage to display teams once set. For example, show "Teams:
[player0 & playerX] vs [others]".

Implement all of the above. This ties the frontend to our backend and sets up
the partner selection step, though final backend handling of choosePartner will
be done in the next step.
```

Explanation: This is a big integration step: - **Front-end Create/Join integration:** - We call the REST endpoints from the lobby form. Since actual API URL isn't known yet (we will have it after deployment), use a placeholder or config that we can update post-deployment. Possibly something like:

```
const API_BASE = "https://<id>.execute-api.<region>.amazonaws.com/Prod";
```

We'll likely leave `<id>` blank or use an env var for local dev (or do nothing, expecting user to fill). - Kiro might define an `api.js` or use fetch directly in component. It's fine. - After fetch, handle success: * If data.gameId exists, store stuff: - They likely will use either React context or simpler localStorage, as the prompt directs localStorage for ease. - So: `localStorage.setItem('gameId', ...)`, 'playerName', 'seat', maybe 'players' if included. * Navigation: since it's a functional component likely with react-router, use `useHistory()` to push "/game". Or use a <Navigate to="/game"/> if using hooks in component.

- Error handling:
  - If fetch returns error (like 404 or 400), catch it.
  - Possibly the response body contains error message. We might do:

    ```
    if(!res.ok) {
       const err = await res.text();
       alert(err || "Failed to join");
       return;
    }
    ```

  - Possibly better to unify error responses as JSON with `message` field in Lambda.

- But for brevity, an alert is fine.

- **Front-end GamePage useEffect:**

- Retrieve localStorage values:
    - `gameId = localStorage.getItem('gameId')`, same for playerName, seat.
- Connect WebSocket with them:
    - e.g. `const ws = new WebSocket(WS_URL + "?gameId="+gameId+"&playerName="+encodeURIComponent(playerName)+"&seat="+seat)`.
    - Then set up ws events (we did earlier).

- Possibly call a GET to retrieve game state:

    - They mention either joinGame gave players list or having an endpoint. We did modify joinGame to return updated players list. Let's assume we did.
    - So we could store that list in localStorage like 'players' (stringified).
    - Then in GamePage, parse that to state (for listing names).
    - If we didn't, we might call a new GET games endpoint.
    - But easier: include `players` in joinGame response.
    - We'll do that: modify joinGame in last step to return the entire players array (list of {seat,name}).
    - Then store `localStorage.setItem('players', JSON.stringify(data.players))`.
    - Then in GamePage, `const players = JSON.parse(localStorage.getItem('players')||"[]")`.
    - This list can be displayed: e.g., `<div>Players: {players.map(p => p.name).join(", ")}</div>` or as a list with seat numbers.
    - That covers showing who’s in game.

- **Partner Selection:**

- If players length==4 and teams not set:

    - To know if teams set, we might look at game status or players structure:
    - Possibly after partner selection, backend will update players objects with a `team` property or set a `teamsChosen` flag.
    - For now, on front end, can track a state `teamsChosen` which is false initially and becomes true after `partnerSelected` message from server.
    - If current user seat=0 (host) and 4 players, show partner choices:
    - The other 3 players (seats 1,2,3 with names).
    - We can list each with a button "Select as Partner".
    - onClick for each:
        - send `ws.send(JSON.stringify({ action:"choosePartner", partnerSeat: thatSeat }))`.
    - Possibly disable the UI after clicking one (since choice made).

    - We'll rely on server to broadcast result to finalize.

- If not host:

- If 4 players but no teams yet, just show "Waiting for host to pick partner".

- On receiving "partnerSelected" (we will implement receiving in next steps when backend sends it):

  - But to pre-empt, we can add a case in ws.onmessage:

    ```
    if(msg.action === "partnerSelected") {
       // e.g., msg.team0 and msg.team1 maybe in payload
       setTeamsChosen(true);
       // hide partner selection UI if any
       // update any team labels if provided
    }
    ```

  - We might not exactly know payload structure yet. But we can plan to handle a message that includes who is on which team.
  - The prompt suggests broadcasting teams or just chosen partner seat. Possibly the backend can compute and send full info: e.g., `{ team0: [0, X], team1: [other two] }`.
  - We'll adapt in Step 7 when implementing it.
  - For now, maybe just log it or mark teamsChosen.

- Also update UI to indicate teams:

  - Could list "Team A: Alice & Charlie; Team B: Bob & Dana" for example.
  - Possibly color code (not required now).
  - As teams are fixed after partner selection, maybe we add to game info display.

- The prompt suggests storing seat on connect query and in localStorage. We have that from join response.

- **Backend**:

- Actually implementing choosePartner in backend not done yet (they say in later step).
- But we can stub in gameAction (the main message handler, which we haven't fully built).
- The prompt says not implementing now, just planning.
- So front-end will send `choosePartner` and likely nothing happens until we do backend in Step 7. But it's fine.

After implementing: - **Test front-end join flow locally**: - We can simulate createGame: * Since we don't have actual API running, if we keep dummy API_BASE_URL, it will fail (likely CORS or network error). * But we can test validation: if you click create without name it might do nothing or we can set required attribute on input. * We might just trust this until we can test with deployed API.

- We can test that localStorage is being set:

  - For example, one could temporarily fake a response: In createGame onClick, instead of calling fetch, assign:

    ```
    const gameId = "TEST123"; localStorage.setItem('gameId', gameId);
    localStorage.setItem('playerName', name);
    localStorage.setItem('seat', "0"); localStorage.setItem('players',
    JSON.stringify([{seat:0,name}, ...]));
    history.push("/game");
    ```

    to simulate.
  - Then on GamePage, it will take those values and attempt to connect WS (which fails, but it's okay).
  - But you'll see Players list showing the names from that fake list.
  - And if you had 4 players in that list, it should show partner selection if you are host.
  - Could simulate that to see the UI.

- Given complexity, it's acceptable to not fully test front-end until integrated with actual back-end in deployment. Just ensure no obvious runtime error (like using undefined values if localStorage keys aren't set).

  - We can avoid errors by defaulting if localStorage missing (e.g., if someone directly goes to /game without going through lobby).

We have integrated the join flow and partner selection UI triggers.

Step 7: Add Shuffle and Deal Logic (Secure RNG for Cards)

Now implement dealing of cards after teams are set. The high bidder will get kitty, but first we need to assign cards to each player and a kitty.

Prompt:

```
# Implement the shuffle and deal of cards in the backend once teams are set.

We assume by this point 4 players are in the game and teams have been chosen
(after `choosePartner`). Now, we handle starting the game by shuffling and
dealing the deck.

**Backend (gameAction Lambda)**:
- We will have a WebSocket message (perhaps triggered by host or automatically
when game is ready) to start the game. For example, after partner selection, the
```

```
server can proceed to shuffle and deal. Let's implement it in the "choosePartner" handling:
  - In the same handler that processes `action: "choosePartner"` (from host), after updating teams, proceed to shuffle the deck and deal cards.

- Construct the Rook deck:
  - The deck has 57 cards: numbers 1-14 in four colors (e.g., Red, Green, Yellow, Black) plus the Rook card [27] .
  - Represent them as strings like "Red14", "Red13", ..., "Red1", similarly for other colors, and "Rook".
  - Use a cryptographically secure shuffle (for fairness). Use Node's `crypto.randomInt` or `crypto.randomBytes` to implement Fisher-Yates shuffle instead of Math.random [3] .
    * Example: for i from deck.length-1 down to 1: j = randomInt(0, i+1); swap deck[i], deck[j].

- After shuffle, deal:
  - Give 13 cards to each of 4 players and 5 cards to the kitty (the last 5 in the shuffled deck).
  - Use the `Hands` table to store each player's hand:
    * For each player seat 0-3, put an item `{ gameId: ..., playerSeat: ..., cards: [array of 13 card strings] }` into DynamoDB (or batchWrite 4 items).
    * Alternatively, store hands in the game state object. But better to use the separate Hands table for sensitive info (we won't send other players' hands to clients).
    * We will also keep the kitty cards in the game state (for now).
  - Update the Games table:
    * Set `status = "BIDDING"` (bidding phase starts).
    * Store the kitty (we can store as `kitty: [5 cards]` in the game item).
    * Set `currentTurn = 0` or whoever starts bidding. Our house rule: host (player0) begins the bidding [7] .
    * Also maybe reset any bid-related fields: `highBid = 0`, `highBidder = null` initially, or we set minimum 50 as context (we know bidding must start at 50).
    * Increment version.

- Broadcast to players:
  - Send each player their hand privately:
    * Use Connections table to get each connectionId for this game.
    * For each player, find their connection and send a message like `{ action: "deal", cards: [ ...13 cards... ] }` **only to that player** (so they see their own hand).
    * For the kitty: do NOT send to everyone. Only the bid winner will eventually see it (during kitty exchange). At deal time, no one gets kitty yet.
  - Announce bidding start:
    * Broadcast `{ action: "biddingStart", startingPlayer: 0 }` to all, indicating the bidding phase has begun and player0 (host) starts with minimum 50 bid.
```

```
    * Everyone can update UI accordingly (enable bidding UI for player0, etc).

- Use `DocumentClient.batchWrite` for writing hands for efficiency (or put in
loop, either okay for 4 items).
- Ensure secure random usage (Node's crypto). Math.random is predictable, so we
prefer crypto for fairness.

**Frontend:**
- Handle the new messages:
  - On `{ action: "deal", cards: [...] }`: if this is for the current player (it
will only be sent to them), update the GamePage state to replace dummy hand with
these actual cards. The UI should now show their real hand. Possibly sort them
by suit and rank for display.
  - On `{ action: "biddingStart", startingPlayer: 0 }`:
    * Show a message "Bidding begins. Minimum bid 50."
    * If `startingPlayer` equals this player's seat, enable their bidding
controls (they act first).
    * Otherwise, show "Waiting for Player X to bid".
  - (The opponents will not get each other's cards, only their own "deal"
message. They will get biddingStart broadcast.)

Implement the backend changes to shuffle and deal in the `choosePartner`
handling, and implement the frontend responses to "deal" and "biddingStart"
messages.
```

Explanation: We integrate shuffle after partner chosen: - The `choosePartner` message handling now does: - Mark teams in game item. - Shuffle deck: * Create deck array of 57 strings (1-14 four suits + "Rook"). * Shuffle using `crypto.randomInt` (the prompt explicitly says to prefer crypto for fairness). - This uses Node's built-in crypto, which is fine in Lambda. - Partition deck: * players 0-3 each get 13 (deck[0:13], [13:26], [26:39], [39:52]). * kitty = deck[52:57] (5 cards). - Write hands: * Could do DocumentClient.put for each hand or `batchWrite`. * Likely the AI will do a loop of 4 and put each (with maybe `await Promise.all` or similar). * Each item: { gameId, playerSeat, cards: [...13...] }. * Possibly also add an attribute for concurrency or version in hand? Not necessary. - Update Games table: * status to "BIDDING", * store kitty (maybe as `game.kitty = [5 cards]`), * could store trump = null for now, * highBid = 0 or just implicit, * currentBidder = 0 (since host starts bidding in our variant) [7]. * Possibly set a field like biddingTeam or turn if needed. * increment version. - Committing both the hands and game update: * Possibly use a transaction to ensure atomicity across tables. * But it's not strictly necessary: we can do sequentially: - Put hands (even if game update fails, or vice versa). - But better might be do game update then batchWrite hands. If game update fails (rare), abort dealing. If hand writes fail, we need to handle (maybe try again). - Simplicity: do game update then 4 puts. It's unlikely to partially succeed in a problematic way visible to user, but one could use TransactWrite with multiple items (some in different tables). - Up to AI; maybe fine sequentially.

- After dealing, send out WebSocket messages:
  - We have connection IDs in Connections table. For each player (4 entries likely in Connections table):

- If connectionId belongs to seat X:
  - get that player's 13 cards from what we just dealt,
  - send to that connection `{ action: "deal", cards: [..] }`.
- Possibly compress or shorten if needed, but plain JSON of 13 strings is fine.
- Broadcast to all:
- `{ action: "biddingStart", startingPlayer: 0 }`.
- Or could also include `minBid: 50` if wanted, but not needed because it's known by rule.
- Possibly include any initial highBid (0) or such, but not necessary.

• We should ensure these are sent only after DB operations succeed.

• **Frontend:**

• Already some logic:
- We likely have a `onmessage` handling stub. We now extend it:

```
if(msg.action === "deal") {
   // msg.cards is this player's hand
   setMyHand(msg.cards); // replacing dummy hand state with actual
}
if(msg.action === "biddingStart") {
   setPhase("BIDDING");
   setCurrentBid(0);
   setCurrentPlayer(msg.startingPlayer);
   // maybe display a message or update some state that triggers UI
changes
}
```

- The UI:
- The bottom hand dummy we had will now be replaced by actual cards when `myHand` state updates via setMyHand.
- The game info can show "Trump: TBD" still, "Phase: Bidding", maybe "Current high bid: 0 (min 50)".
- If it's this player's turn (startingPlayer equals their seat), we enable the bid buttons (we haven't fully coded bidding UI enabling/disabling yet, but we can do it now:
  - e.g., have a state `currentBidder` or `currentPlayerTurn` and compare to seat in UI to enable).
- Possibly highlight who starts (maybe already done via waiting message logic).
- Opponent view:
- They receive no "deal" messages for others, only their own. So they won't see others' cards (good).
- They do see biddingStart. They will set current turn etc, and see "Waiting for Player0 to bid".

Testing: - After partner selection, if we can simulate choosePartner message: * We don't have the actual sequence yet because we haven't implemented sending from host and receiving on backend, but: * If we manually trigger the backend function (hard offline test possible by invoking choosePartner handler with an event?), complicated. Better to wait to test with real multi-client flows.

- Basic check:
- Ensure shuffle logic doesn't fail (like correct generation of deck array).
- If we had a local Dynamo to test writing, possible but heavy to set up. We'll trust by code review for now.
- We should confirm point: connecting pipeline:
    - Host picks partner => front end sends "choosePartner".
    - Our gameAction lambda will handle that:
    - We didn't explicitly instruct to update teams in game state now, but presumably yes:
        - e.g., add `players[i].team`.
    - The prompt did mention "update teams after partner selection".
    - We should do that: in choosePartner:
        - Determine partnerSeat from message,
        - Team0 = [hostSeat(0), partnerSeat], Team1 = [the other two].
        - Mark in game item, maybe as `players[...].team = 0 or 1`.
    - Then continue to shuffle.
    - That means by the time we send "deal" and "biddingStart", teams are set in DB (though team info might not be immediately needed by clients for bidding, just for scoring later).
    - The clients might want to know teams now. Possibly send a message about teams.
    - We might do that in partnerSelected message earlier (like step 6 had planned broadcasting teams).
    - But in our approach here, we might have combined it:
        - The moment we handle choosePartner in backend, we do all: update teams, then shuffle, then broadcast deal and biddingStart.
        - We might not explicitly broadcast "partnerSelected" separately if we immediately transition to dealing.
        - But maybe we should:
        - Non-host players currently "waiting for partner selection" won't know who was picked if we skip directly to dealing.
        - However, when cards are dealt, they might infer teams if they look at who sits where? Not necessarily. They need to know who is whose partner.
        - It might be better to still broadcast a "partnerSelected" or include team info in biddingStart.
        - Possibly incorporate team info in biddingStart: but biddingStart may come a second later, and they'd prefer to know partners as soon as picked.
        - Maybe do:
        - After setting teams, broadcast `{ action:"partnerSelected", team0: [0, X], team1: [the others] }`.
        - Then do dealing.
        - The prompt did not explicitly say, but I think it's wise to include.
    - We can assume we will do that either now or in step 8 or 9. It's a minor detail, but important for completeness.
    - The front-end we wrote expects "partnerSelected" possibly. If we don't send it, we should ensure front-end still gets team info eventually (maybe in a "handComplete" or something, but that's too late).

- I'll assume we will broadcast partner selection before dealing.

Anyway, we'll proceed with dealing as if partner selection triggered it.

Step 8: Implement Bidding Phase Logic

Now implement bidding (players sending bids/passes and updating state accordingly) in the backend and update the front-end to handle bidding actions and updates.

Prompt:

```
# Implement real-time bidding logic after dealing cards.

**Backend (gameAction Lambda)**:
- Handle messages `{ action: "bid", amount: N }` and `{ action: "pass" }` from
players during the bidding phase.
- The game state in DynamoDB (Games table) should track:
  - `highBid` (current highest bid amount).
  - `highBidder` (seat of current highest bidder).
  - `currentBidder` (seat whose turn it is to bid).
  - `passed` (maybe a list of seats that have passed).
  - We set `status = "BIDDING"` already. We'll update these fields as bidding
progresses.

- When a "bid" message comes in:
  - Identify the player (from connectionId via Connections table, or perhaps the
message includes their seat from client).
  - Load the game (Games table) to get current `highBid`, `highBidder`,
`currentBidder`, and `passed` list.
  - Verify:
    * It's this player's turn (`currentBidder == playerSeat`).
    * The bid `amount` is valid:
      - If `highBid` is 0 (no bid yet), ensure `amount >= 50` [10] .
      - Else, ensure `amount >= highBid + 5` (must exceed current by at least 5)
[11] .
      - Also typically bids are multiples of 5 (we can assume our UI enforces
that).
    * If invalid, optionally send an error message back (or ignore).
  - If valid:
    * Update game state in DB:
      - `highBid = amount`,
      - `highBidder = playerSeat`,
      - Remove this player from `passed` list if present (they are clearly still
in).
      - Determine next bidder:
        + nextSeat = (playerSeat + 1) mod 4, keep skipping any seats that are in
`passed` list until find one not passed.
```

```
        + If that nextSeat equals the current player (meaning 3 others passed),
then bidding is over.
      - If bidding continues:
        * set `currentBidder = nextSeat`.
      - Increment version and save.
    * Broadcast to all:
      - `{ action: "bidPlaced", seat: X, amount: N }` (so everyone sees the new
high bid and who made it).
      - And `{ action: "nextBidder", seat: Y }` for whose turn is next.

- When a "pass" message comes in:
  - Identify player, load game state (or have it from context).
  - Verify it's their turn (`currentBidder == playerSeat`).
  - Update state:
    * Add playerSeat to `passed` list.
    * Determine next bidder similarly by skipping passed players.
    * If after passing, the number of players not in `passed` list is 1 (only
one player remains bidding):
      - Bidding ends. The remaining player is the highBidder if not already set.
      - Determine winning bid = `highBid` (should be set if at least one bid was
made; if no one bid at all (highBid still 0), then by rule the host must take 50
or redeal; we can decide that if highBid==0, set it to 50 for the last player).
      - Update game state:
        + `status = "KITTY"` (a phase where bid winner will exchange kitty
cards).
        + Record `bidWinner = remainingSeat` and `winningBid = highBid`.
        + (We will handle kitty exchange next step.)
      - Increment version and save.
      - Broadcast to all:
        * `{ action: "biddingWon", winner: W, amount: highBid }`.
        * Possibly include which team won if needed.
        * The next step for the winner will be to get the kitty and choose trump
(we'll implement that subsequently).
    * If bidding not over yet:
      - Update `currentBidder = nextSeat` and save.
      - Broadcast `{ action: "playerPassed", seat: X }` and `{ action:
"nextBidder", seat: Y }`.

- Add concurrency control on these updates using the `version` attribute
(ConditionExpression `version = :expected` and `SET version = version+1, ...` in
the update, similar to joinGame logic) [3] .

**Frontend:**
- Add UI in GamePage for bidding:
  - Show current highest bid and bidder (update when "bidPlaced" messages
arrive).
  - If it is this player's turn (nextBidder equals their seat), enable input:
    * Provide a button or input to bid the minimum valid amount (e.g., "Bid 55"
```

````
if current high is 50) and a "Pass" button.
    * We can use a simple approach: one button to bid exactly highBid+5 (or 50
if no bid yet) and one to pass. (For jump bids, the user could click multiple
times or we can allow a manual input for amount, but optional).
  - When "Bid" button clicked:
    * Determine bid amount: if `currentHighBid` is 0, bid 50; otherwise bid =
currentHighBid + 5.
    * Send `ws.send(JSON.stringify({ action: "bid", amount: bidAmount }))`.
  - When "Pass" clicked:
    * Send `ws.send(JSON.stringify({ action: "pass" }))`.
  - Disable these buttons when it's not this player's turn.

- On receiving WebSocket messages:
  - `{ action: "bidPlaced", seat: X, amount: N }`:
    * Update local state `currentHighBid = N` and `highBidder = X`.
    * Update UI display of high bid.
  - `{ action: "playerPassed", seat: X }`:
    * Mark that player as passed (maybe grey out their name or add "(passed)"
next to it in players list).
  - `{ action: "nextBidder", seat: Y }`:
    * Update `currentBidder = Y`.
    * If `Y == mySeat`, enable my bid/pass controls (show "Your turn to bid").
    * Else, disable my controls and show "Waiting for Player Y...".
  - `{ action: "biddingWon", winner: W, amount: M }`:
    * Conclude bidding phase:
      - Display a message: "Player W won the bid with M points."
      - If `W == mySeat`, show "You won the bid!" (and we'll proceed to kitty
exchange next).
      - Disable bidding UI for everyone.
      - Transition state to next phase (kitty/trump selection, which we'll
handle in next step).
    * Possibly update some state like `bidWinner = W` and store winning bid for
reference.

Test the bidding flow with multiple clients if possible (we will do so after
deployment). Ensure that passes and bids update for all players and that when
one wins, the bidding ends properly.```

**Explanation:** This step is heavy but straightforward:
- **Backend gameAction**:
  - We'll incorporate in the main message handler (which we may call
`gameAction.js` that routes based on `msg.action`).
  - Bidding stage state:
    * We had set `status = "BIDDING"`, `currentBidder = 0`, `highBid = 0`,
`highBidder = null` at deal time.
    * Possibly a `passed` array initially empty.
  - On "bid":
    * Check currentBidder and seat.
````

````
    * Validate amount:
      - if highBid=0, require >=50 [10] (minimum bid rule).
      - else require >= highBid+5 [11].
      - Could also ensure amount %5==0 but presumably our UI ensures.
    * Update:
      - highBid = amount, highBidder = that seat.
      - Remove from passed list if present (someone might have passed earlier
and then re-entered bidding? But by rules, once passed, can't re-enter [12]. So
maybe we do not allow re-entering: if you pass, you're out for good.
      - Actually yes, "Once a player passes, they may not bid again that
round" [12]. So we should keep them in passed list permanently. So no need to
remove from passed, because a passed player should never get a turn again.
      - So remove-if-present likely unnecessary.
      - We might not even allow a passed player to send "bid" if our front-end
prevents it. But server can double-check: if seat in passed, ignore bid or treat
as invalid.
    * Next bidder:
      - nextSeat = (currentBidder+1) mod4. But currentBidder was the one just
made bid? Actually:
        - If current player made bid, then the next to act is the next seat who
hasn't passed.
      - Do:
        ```
        next = (playerSeat+1) % 4;
        while(next is in passed list) { next = (next+1)%4; }
        ```
        - If next equals player who just bid, that means all others passed and
bidding should end (but that scenario can't happen immediately on a bid, because
if others had all passed except this player, the round would have ended when
they passed, before this bid).
        - So we can assume if next calculation returns to same, or if passed
list size == 3, we handle in pass logic rather than bid logic. So maybe no check
needed here that ends bidding.
      - Set `currentBidder = next`.
    * Save game:
      - Use condition `version = X` and increment version.
    * Broadcast:
      - `bidPlaced` to all, with seat and amount.
      - `nextBidder` with seat next.

  - On "pass":
    * Identify seat.
    * Mark them passed:
      - Add to passed array if not already.
    * Next bidder:
      - Find next like above.
      - If nextSeat == the seat that just passed (meaning all others in passed
list leaving only themselves - which cannot happen because they'd not pass if
````

```
alone; or more logically, if passed list length became 3, leaving one not
passed):
        - That means bidding ends because only one player remains unpassed.
      - Check if `passed.length == 3`:
        - Then find the one not passed as winner.
        - If `highBid` is 0 (no one bid at all, so all passed):
          - Some house rules: redeal or force dealer take bid at minimum. Let's
do: if no one bid, let the last remaining (which would be seat 0 if everyone
passed in order) be winner at 50 by default.
          - Or we can automatically assign bidWinner=lowest seat not passed
(likely 0) and highBid=50.
          - We'll do: if highBid==0, set highBid=50 for winner (the rule isn't
clearly given, but a typical approach is dealer must take it at minimum 50 if
all pass).
        - Set `bidWinner = thatSeat` (or just use highBidder, but if no one bid,
highBidder might still be null).
        - Set `status = "KITTY"` or we can keep "BIDDING" until they pick kitty;
but marking a new phase is fine.
        - Possibly store `trump = null` still, `currentPlayer` might not matter
now until play.
        - Save game state (inc version).
        - Broadcast `biddingWon` with winner and amount.
      - If not ended (passed list != 3):
        - Set `currentBidder = nextSeat` (the next person who hasn't passed).
        - Save state (inc version).
        - Broadcast `playerPassed` (so others know this player is out) and
`nextBidder`.
    * In both cases, the saving and broadcasting.

  - Must carefully handle concurrency and double passes/bids:
    * If two players try to act at once (shouldn't if UI controls properly), one
will fail version check or not match currentBidder and be ignored.

- The AI will likely incorporate condition expressions on version:
  - They might get the game item at start of each message handler for
simplicity, then do a conditional update. Or they may do one atomic update with
condition on `currentBidder` or similar. Simpler is:
    * For "bid": do a DocumentClient.update with ConditionExpression
`currentBidder = :player AND version = :v` (for concurrency and correct turn),
and update highBid, highBidder, currentBidder, passed if needed, and version++.
    * For "pass": ConditionExpression `currentBidder = :player AND version
= :v`, update passed list (can use list_append or if we treat passed as set use
ADD, but it's a list of seats, simpler to append if not exists), update
currentBidder or status if ended, version++.
    * This might be doable, but passing a list in expression is tricky (Dynamo
doesn't have an atomic "append if not exists", but we can always append even if
it's duplicates; or maintain a boolean map).
    * They might simplify by reading game, modifying in code, then writing back
```

```
with condition on version. That is okay albeit two operations (get then update).

- The logic uses many details from rules which we cite:
  - min bid 50 [10] ,
  - increments of 5 [11] ,
  - no re-entry after pass [12]  (we enforced by not removing from passed).
  - Also that host starts bidding (we already set that).
  - We might not have a direct citation about ending bidding when 3 passed, but
it's logical.

- **Front-end:**
  - We add state for bidding:
    * `currentHighBid`, `currentHighBidder`, `currentBidder`, maybe an array of
passed seats to mark UI.
    * Already storing these from messages:
      - On bidPlaced: set highBid and highBidder.
      - On playerPassed: update passed list (e.g., push seat or mark in players
array).
      - On nextBidder: set currentBidder.
      - On biddingWon: announce winner.
  - Buttons:
    * We likely have a `Bid` and `Pass` button.
    * In UI:
      - if currentBidder == mySeat:
         + show "Your turn: " with perhaps an option to input or a predetermined
next bid value:
           - We choose increment only to keep UI simple. So:
             nextBid = currentHighBid == 0 ? 50 : currentHighBid + 5.
             Show a button "Bid X" and a "Pass" button.
           - If player wants to bid more than +5, our UI doesn't allow directly,
they'd have to wait for cycle to bid again (which isn't how it works; usually
you can jump bid).
           - But maybe it's fine as an initial implementation.
         + If highBid is already by this player (rare that they'd get turn again
without others passing, except in head-to-head final 2 scenario, but if two
remain they alternate turns anyway).
         + We'll allow consecutive turn logic as per nextBidder control.
      - else:
         + disable/hide bid/pass, show "Waiting for ... to bid".
    * We also display the current high bid and highest bidder somewhere (like
"Highest Bid: 55 (Player 1)").

  - On messages:
    * "bidPlaced": update display of current high bid and who.
    * "playerPassed": mark that player (maybe in players list UI, append
"(passed)").
    * "nextBidder": update state to new bidder.
       - If it's me, enable my buttons and maybe highlight UI element that it's
```

````
my turn.
       - If not me, ensure my UI is disabled.
    * "biddingWon":
       - Alert or text "Player X won bid with Y".
       - If X is me, maybe special message "You won the bid!"
       - We should then prepare for kitty/trump stage (next step).
       - Possibly clear or hide bidding UI entirely since bidding phase over for
all.

  - Clear passed statuses if needed when bidding done, or in next hand start.

We should verify once integrated:
- Not easily testable without multiple players and backend running.
- But if run two browser windows after deployment, we can simulate bids and
passes to see if logic holds.

### Step 9: Implement Kitty Pickup and Trump Selection

After bidding, the winner gets the kitty, discards 5, and chooses trump.

**Prompt:**

```text
# Implement kitty exchange and trump selection by the bid winner.

**Backend (gameAction Lambda)**:
- After bidding ends (`biddingWon`), the game state has:
  - `bidWinner` (seat) and `winningBid`.
  - The `kitty` (5 cards) stored in game item from dealing phase.
  - Now we wait for the bid winner to pick up the kitty and choose trump.

- We will handle a message from the bid winner, e.g. `{ action: "takeKitty" }`
or directly `{ action: "discardAndTrump", discard: [...5 cards...], trump:
"<Color>" }`.
- Let's do it in one step to reduce round trips:
  - The client (bid winner) will combine discarding and trump selection into one
message after they've decided.
  - So expect `{ action: "discardAndTrump", discard: [card1, card2, ..., card5],
trump: "Red"/"Green"/... }` from the bid winner.

- When this message arrives:
  - Validate it:
    * Ensure sender is indeed the bidWinner (from game state).
    * Ensure the 5 `discard` cards are a subset of the bid winner's current hand
+ the kitty.
      - The bid winner's hand was in Hands table (13 cards) and the kitty is in
game item (5 cards). By this time, the bid winner has effectively 18 cards total
(though we never sent them kitty yet).
````
````

- Actually, we should have sent the kitty to bid winner after bidding: perhaps when broadcasting biddingWon, we could also send a private message with the kitty cards to the winner. If not, we do it now:
  + Perhaps on biddingWon, front-end will signal the winner to fetch kitty. Alternatively, we can proactively send the kitty to winner.
- Anyway, assume the winner knows the kitty cards (from a prior message or we can retrieve and include here; but since the client is sending discard list, they must have known kitty).

* Check `discard.length == 5`.

- Update game state:
  * Remove those 5 cards from the bid winner's hand in the Hands table.
    - Retrieve the bid winner's hand item (gameId + seat). It has 13 original cards. Combine with the 5 kitty cards (from game state) to simulate 18, then remove the ones they chose to discard.
    - Alternatively, at the moment bidding ended, we could have automatically given the kitty to the bid winner's hand (update hand item to include kitty). Let's assume we did not yet. So do it now:
      + Get hand item, get game.kitty.
      + Compute new 18 cards = hand.cards + kitty.
      + Verify the 5 discards are in that set.
      + New hand = 18 - discards (13 cards).
      + Update Hands table: set that player's cards to the new 13.
  * The 5 discarded cards form the final kitty (out of play). According to rules, any points in them count for the bid winner's team [15].
    - We can calculate points of these discards now and store them or add to a "bidTeamPoints" counter.
    - Or simply note that these points belong to bid winner team for scoring later.
    - We might store in game item something like `kittyPointsCaptured = <points>`.
  * Set trump suit in game state: `trump = <Color>` (from message).
  * Update game status to "PLAYING" (trick-play phase begins).
  * Determine who leads the first trick:
    - Our house rule was left of dealer leads the first trick [5]. But since dealer was presumably player0 (host) initially, left of dealer is player1.
    - However, some play that bid winner leads. We stick to our specified rule: left of dealer (player1) leads first.
    - If we rotated dealer each hand, we'd use that, but this is first hand.
    - So set `currentPlayer = 1` (or whatever seat is left of dealer).
  * Increment version and save all updates in Games table.

- Remove the `kitty` from game item or mark it empty (no longer needed).

- Broadcast to all players:
  * `{ action: "trumpChosen", suit: "<Color>" }` – informs everyone of trump.
  * `{ action: "playStart", leader: L }` – informs who leads the first trick.
  (You can combine these or send separately.)
  * Optionally, announce which cards were discarded? Usually not – in our

```
house rules, the discard (kitty) is not revealed to others, its points just go
to bid winner [15] .
    * So we do NOT show discarded cards to others.
    * The bid winner already knows which 5 they discarded; we don't need to send
them back.

**Frontend:**
- When the bid winner receives the kitty (we should have sent them a message
after biddingWon):
  - If not done already, we should send a private `{ action: "kitty", cards: [5
kitty cards] }` to the bid winner right after bidding is won.
  - The bid winner's UI should then display those 5 kitty cards added to their
hand (now 18 cards total).
  - Show a UI for them to choose 5 cards to discard:
    * Perhaps allow clicking cards to toggle selection. Keep track of selected
cards (max 5).
    * Once 5 selected, enable a "Confirm Discard & Choose Trump" button.
    * Also provide a way to pick trump suit (e.g., 4 suit buttons or a
dropdown).
    * On confirm, send `{ action: "discardAndTrump", discard: [..5..], trump:
"<Color>" }` through WebSocket.
    * Disable UI after sending.

- Other players while the winner is deciding:
  - Show "Winner is picking trump..." message.

- On receiving `{ action: "trumpChosen", suit: X }`:
  - Update UI to show trump suit (e.g., display "Trump: X" and maybe highlight
trump suit cards in their hand).
  - On `{ action: "playStart", leader: L }`:
  - Transition to trick-play phase:
    * Set `currentPlayer = L` (who leads).
    * If L equals my seat, highlight that it's my turn to play a card. If not,
show waiting for player L to play.
    * Clear any kitty/discard UI remnants.

After this, the first trick will be played (handled in the next step).

Implement the above:
- In backend choosePartner handler, after ending bidding, call the shuffle
(already done in Step 7) so likely we handle kitty in bidding end or separate
message.
- Actually, perhaps easier: we might not automatically shuffle on choosePartner
previously; but we did in Step 7.
- Actually Step 7 had us shuffle right after partner selection, which is
possibly earlier than bidding. That might be out of order; typically dealing
happens immediately after teams set, then bidding.
- Correction: In Rook, dealing happens before bidding. So Step 7 (dealing) was
```

```
correct timing: we dealt the cards before bidding started.
- So by bidding end, players already have their hands (except kitty is aside).
- So at bidding end:
  - The bid winner needs the kitty: we should send it to them now.
  - So indeed, right when broadcasting biddingWon, also send a private message
to winner: `{ action: "kitty", cards: [...] }` (the 5 kitty cards).
- The frontend for winner should receive "kitty" and show those cards (we
planned this in Step 8 front-end, but let's implement now):
  - We might have missed that in Step 8. We can do it now:
    * After broadcasting biddingWon, in backend send winner connection
`{ action: "kitty", cards: game.kitty }`.
    * Remove game.kitty from DB or keep for reference to verify discards.
- Then winner client shows kitty cards, chooses discards & trump, sends
discardAndTrump message, which we handle as above.

Test scenario after implementation:
- Hard without multi-client to test UI, but we can simulate for winner
perspective:
  - Suppose biddingWon came and they got kitty: they'd have 18 cards in UI
(somehow combine).
  - They choose 5, pick trump "Red", hit confirm:
    - That triggers ws send "discardAndTrump".
  - Backend processes:
    * updates DB (remove those from hand, set trump, etc),
    * sends trumpChosen and playStart to all.
  - Everyone receives trumpChosen (display trump) and playStart (starting
trick).
  - The leader (player1) now can play.

We should carefully ensure:
- After discarding, all hands in DB have 13 cards (others had 13 since deal,
winner had 13 remain after discarding, kitty out).
- We might compute points in discard, but actual scoring will be done end of
hand. But we might store or just recalc at end:
  - Probably simpler: at end-of-hand, sum all points from tricks + points in
discard (since those weren't in any trick).
  - We can hold discard points in a variable or in game state (like
game.kittyPoints).

We'll likely do scoring in Step 10 or 12.

Now we have fully set up the game for trick play (with trump known and first
player indicated).

### Step 10: Add Card Play Validation and Turn Logic

Implement playing of tricks: players playing cards, following suit, determining
trick winner, updating scores and next turn, and ultimately calculating the
```

````
score at hand end and possibly game over.

**Prompt:**

```text
# Implement trick-taking logic and scoring of the hand.

**Backend (gameAction Lambda)**:
- Handle `{ action: "playCard", card: "CardName" }` messages from players during
the play phase.
- Game state should track:
  - `trump` (already set),
  - `currentPlayer` (whose turn to play),
  - `ledSuit` for the current trick (suit of the first card played in the
trick),
  - `trickCards` (cards played in the current trick so far, could store as list
of {seat, card}),
  - We also maintain `teamPoints` for this hand (points captured by each team in
tricks).
  - Possibly a `trickCount`.

- When a "playCard" arrives:
  - Identify player (connection -> seat).
  - Verify it is their turn (`currentPlayer == seat`).
  - Retrieve the card and player's hand from `Hands` table:
    * Check that the card is indeed in their hand list. If not, ignore or error.
  - If this is the first card of the trick (no ledSuit set for current trick
yet):
    * Set `ledSuit = suitOf(card)` (use first part of card string, e.g., "Red"
if "Red5"). If the card is "Rook", treat ledSuit as trump (since Rook is trump
color) [17] .
    * Initialize `trickCards = [ { seat, card } ]`.
  - If not first:
    * Ensure the player follows suit:
      - Determine `ledSuit` from game state.
      - If the player has any card of ledSuit in their hand (check `Hands` data)
and the card they played is not that suit:
         + This is a renege (illegal) [22] . For now, we can reject the play (or
allow it but it's against rules; better to enforce).
         + So if illegal, do not remove card, maybe send an error message back.
(Our front-end will prevent this anyway.)
    * Append `{ seat, card }` to `trickCards`.
  - Remove the card from the player's hand in the Hands table (update the item,
pull that card from the list).
  - Broadcast to all: `{ action: "cardPlayed", seat: X, card: "CardName" }` so
everyone sees the card on table.
  - Determine next player:
    * If this was not the 4th card of the trick:
````
````

```
      - Find next seat (seat+1 mod4) that still has cards (in a 4-player game everyone has cards until end of hand, but if a player had dropped out due to some error, skip them).
      - Set `currentPlayer = nextSeat`.
      - Update game state (version++).
      - Broadcast `{ action: "nextPlayer", seat: nextSeat }`.
    * If this was the 4th card (trick complete):
      - Determine winner of the trick:
        + Compile the 4 cards in `trickCards`.
        + Find the highest card:
          - If any trump was played, highest trump wins the trick [23]. (Remember Rook is lowest trump, so any other trump card beats Rook if present) [17].
          - If no trump played, highest card of ledSuit wins [23]. Use rank order where 1 is highest rank above 14 [27].
          - (We need a rank value: maybe map 1->15, 14->14, 13->13,...5->5, Rook special rank maybe 0.)
        + Determine winning seat.
      - Assign trick points:
        + Sum points of the 4 cards: 5s=5, 10s=10, 14s=10, 1s=15, Rook=20 [20] (others 0).
        + Add these points to that team's hand score. (We know teams from earlier. If winner's seat is in team0, add to team0Points, else team1Points.)
      - Update game state:
        + Clear `trickCards` and `ledSuit` for next trick.
        + Set `currentPlayer = winnerSeat` (they will lead next).
        + Possibly increment a `trickCount`.
      - Broadcast to all:
        + `{ action: "trickWon", winner: W, points: P }` (announce who won and points earned).
        + And `{ action: "nextPlayer", seat: W }` to indicate who leads next (or include that in trickWon).
      - If that was the last trick of the hand (trickCount == 13 or players have no cards left in Hands table):
        * The hand is over. Compute final scores:
          - We have teamPoints from tricks. Also add any points from kitty/discards that belong to the bid-winning team (we stored these when discarding kitty).
          - Determine each team's total for the hand.
          - Determine if the bid-winning team made their bid:
            * If team of bidWinner >= winningBid, they "made it" and score all points they earned.
            * If not, they are "set": they score 0 plus a penalty of -bid (so subtract bid from their cumulative score) [26]. The other team still scores whatever points they earned.
          - Apply the "200 bonus" rule: if a team captured all 180 points (sweep), and that is the bid-winning team, give them 200 instead of 180 [19]. (If defenders captured everything, they just get 180 since bid team gets -bid.)
          - Update cumulative `Games` table score for each team (persist in game
```

```
item).
          - Set `status = "HAND_COMPLETE"` (or prepare for next hand).
          - Increment version and save.
        * Broadcast `{ action: "handComplete", team0Points: X, team1Points: Y,
bid: N, bidTeamMade: true/false, team0Total: A, team1Total: B }`.
        * If a team reached 500+ total, also broadcast `{ action: "gameOver",
winner: "Team0 or Team1" }`.
        * (We can handle moving to next hand or ending game accordingly. If game
not over, perhaps allow starting next hand via a new prompt or automatically if
desired.)

**Frontend:**
- On "cardPlayed":
  - Add that card to the table UI (show the card at that player's position on
table).
  - Remove that card from that player's hand UI (if it's this player's own, it
was already removed on their action; if another's, you can decrement their card
count or remove one placeholder).
- On "nextPlayer":
  - Update current turn indicator.
  - If it's my turn, enable my hand for clicking a card.
  - If not, disable my clicks.
  - Also enforce suit-following in UI:
    * If `ledSuit` is set (you can derive from the first card of current trick
on table), highlight cards in my hand that are of that suit and disable others
if I have any of that suit.
    * (Back-end also enforces, but this helps the user.)
- On "trickWon":
  - Show a message "Player W won the trick (+P points)".
  - Maybe briefly highlight winner's name.
  - Clear the table (remove the 4 cards after a short pause or on next action).
  - Update running score display (team points for this hand).
- On "handComplete":
  - Display a summary: points each team got this hand, and cumulative scores.
  - Indicate if bid was made or set (e.g., "Bid team made their bid" or "Bid
team set, -N points").
  - Possibly a "Next Hand" button if game not over (or auto-start after a few
seconds).
- On "gameOver":
  - Announce the winning team and perhaps disable further input. Show a "Game
Over" banner.

This completes the game loop. The UI should refresh for a new hand if continuing
(could reset state and wait for shuffle of next hand). For now, we can conclude
the game at gameOver.

Implement all these pieces carefully.
```

Explanation: This final step covers playing cards and finishing the hand: - **Backend:** - We'll incorporate logic in gameAction for "playCard": * Check currentPlayer, etc. * We maintain `game.ledSuit` and `game.trickCards` in game item or perhaps ephemeral in memory (we could store partial trick in game item). - Storing partial trick in DB might be overkill (though could if want persistence after a pause). - It's simpler to store these in game item since we have to possibly handle concurrency too. But trick is short-lived data, and any message can reconstruct context by reading these fields. - We'll likely store `trickCards` and `ledSuit` in game item to easily compute winner after 4th card. - Or store `trickCount` to know number of cards currently in trick (which is also length of trickCards). * Follow-suit enforcement: - Check hand in Dynamo to see if that seat has any card with suit = ledSuit. If yes and played card suit != ledSuit and card != Rook (Rook as trump not led suit), then illegal. - Possibly we just don't allow it (should rarely happen because front-end prevents). * Remove card from Dynamo: - DocumentClient.update on Hands table: use `DELETE` or `REMOVE` with list if order not needed, or get then put new list. Possibly easiest to do a get of hand array in memory and filter out, then put back (with condition to ensure not changed? Or since only that player writes to their hand usually, concurrency collision is minimal). - The AI might do `UpdateExpression REMOVE cards[index]` if they know index. Or use `list_pull` if no guarantee of order. - More likely: they fetch the hand item, filter out card, then use DocumentClient.put to write new list. Simpler albeit two operations. * Next player: - if trick not complete: + set currentPlayer to next (skip passed players? Actually in play, passed doesn't matter because all players play until out of cards). + all 4 should still be in game if not disconnected. + So simply next = (seat+1)%4. + Since in trick phase, no one is "out" except maybe if a player had no cards left (but that can't happen in middle of a trick because each had same number of cards). + So next always valid. - update game (ledSuit remains for this trick). - broadcast nextPlayer. * If trick complete: - Compute trick winner: + For each card: - get suit and rank. For Rook, treat suit = trump. - track if any trump present. - If trumps present: filter trickCards to only those of suit=trump (including Rook). * Among those, find highest rank: we treat ranks as ints with 1->15, Rook->0 (lowest). * Actually Rook suit = trump, rank consider as 0 (lowest). * Others in trump suit convert rank number (string substring after suit) to int, but map 1->15. * Then max by that value. - If no trump in trick: filter to only ledSuit, find highest rank with 1->15 as above. + Determine winner seat. - Points: + Map card to points: * card includes number: convert to int, if value in {5,10,14} add 5 or 10 accordingly; if value==1 add 15; if card=="Rook" add 20 [20] . * Or use lookup table: {5:5, 10:10, 14:10, 1:15}. * Sum up. + Add to teamPoints (likely track in game item or local). - We can keep `game.team0Points` and `game.team1Points` in game item. - Or compute on the fly at end-of-hand by summing trick records, but easier to accumulate. - We'll accumulate now: * Determine winnerTeam (we know seat and we know teams from earlier; e.g., if players have team field). * Add points to that team's running total in game item. - Clear trick state: * Remove ledSuit and trickCards or reset them. - Set currentPlayer = winner seat (they lead next trick). - If that was last trick (players now each have played 13 cards): * We can detect by checking if each Hands item is empty or trickCount==13. * End-of-hand: - Retrieve `game.winningBid` and `game.bidWinnerTeam` from earlier stored values (maybe store `bidWinnerTeam` when bid won). - Compute final hand points for each team: = teamPoints (from tricks) + possibly kitty/discard points we stored for bid team. - Determine bid success/failure: If bidTeamPoints < winningBid: they fail. - Score assignment: * If bid made: bid team gets all their points, other team gets theirs. * If bid failed: bid team gets -winningBid (penalty), and 0 from their points; defense gets their points. - Apply 200-point bonus: * If a team took all tricks (opponent 0 points) and that team is bid team, set their score for hand = 200 (instead of 180 max) [19] . * (If defenders took all points (bid team 0), no special bonus by our rules, they just got 180 which is their points.) - Update cumulative overall scores (game.totalScore0, totalScore1). - Mark game status "HAND_COMPLETE" or move to next. - If a team >=500, mark game over. - Save game state. * Broadcast handComplete: - Provide hand points and maybe mention bid success or not. - Provide new total scores. * If game over, broadcast gameOver with winner. * Else, possibly automatically start next hand after a pause or prompt host to start next hand. (This can be optional; we might not implement multi-hand fully, but at least prepare.) - We can for now just wait; the players could decide to start a new game or end.

Frontend: - Already handling cardPlayed, nextPlayer, trickWon from prompt: * Place cards on table. * Remove from hand UI. * Update turn highlight. * On trickWon: maybe flash winner, clear table cards (we can remove them or move them off after a short delay so players can see trick result). * Update hand points (we can maintain hand score state to display running points). - On handComplete: * Show a modal or section with "Hand over: Team0 got X, Team1 got Y. [Bid team made/failed their bid]. Total scores: ...". * Possibly a "Next Hand" button if game not over (if implementing). * Clear any per-hand states (e.g., passed list, trick count, etc. for next hand). - On gameOver: * Announce winner ("Team A wins the game!") perhaps in an alert or modal. * You might disable any further input.

- Prepare UI for next hand if continuing:
- If we allow next hand, you would reset state (empty hands, perhaps increase dealer index for next round, and maybe automatically trigger shuffle).
- This might be beyond scope, but you can at least clear the interface and maybe allow a "Restart" or "Play again" that could create a new game.

Implement the above logic in the backend and the corresponding front-end message handlers and UI updates. ```

Explanation: The final step integrates trick play and scoring: - **Backend**: - Add in gameAction: * "playCard": handle as described: - Likely approach: fetch game state to know current trick context (ledSuit, trickCards, currentPlayer, trump, etc). - Or store minimal context in game item: + We might have `game.ledSuit` and partial `game.trickCards` list in game item. + Or track trick in memory between messages, but Lambda is stateless, so must store in DB or reconstruct via messages sequence. + Easiest: store `ledSuit` and maybe a `trickCount` (cards played so far). + But partial trick cards also needed to determine winner. We can accumulate them in DB or reconstruct by reading all 4 "cardPlayed" messages we broadcast (not reliable). + Better: store trickCards in game item (like an array of 4 possibly). + The AI might choose storing them or storing partial count. + I'd have game.trickCards or game.currentTrick list. - Validate suit: + We need to find if player has ledSuit in hand. + We could fetch their hand item to see suits in it. But we are anyway retrieving their card from hand. + Possibly easier: we already removed card from hand after each play, but prior to removal, we have entire list, can check suits there. - Remove card: + The DocumentClient has a `UpdateExpression REMOVE cards[idx]` if we know index. But we have to find index by scanning array for that card string. + Could do a get, filter, put. That is simpler logically. + We'll do that.

```
  - Trick resolution after 4:
    + We should detect when 4th card played:
      - If trickCards length was 3 before adding, now it's 4 -> complete.
      - So can check if `trickCards.length === 4` after adding current one.
      - The gameAction function might need to know trick state length:
        * If we stored a trickCount field, we increment each play, then check
if trickCount == 4.
        * Or check trickCards list length in DB.
      - The AI might do something like: retrieve game and see if already 3
trickCards stored, then after adding 4th do winner logic.
```

```
    + Determine winner using rules:
      - If any card suit==trump, highest trump wins.
      - Rook suit treat as trump with rank low.
      - Else highest ledSuit wins.
      - We'll implement a rank mapping as described.

    + Points in trick:
      - Sum points with mapping or if statement.

    + Add to teamPoints:
      - We need to know team of winner seat. We can know because earlier we
should have assigned team to each player (in partner selection we intended to
mark teams).
      - If we didn't store team in game item explicitly, we can derive: if seat
chosen by host was partner, then team0 = {0, partner}, team1 = others. We can
store perhaps game.team0 = [seats], team1 = [seats] in game item at partner
selection.
      - It's easier if we did that or gave players a team property.
      - The AI could retrieve players array from game state and deduce by seat
if we kept team on them (which they might if we updated players objects with
team).
      - We'll assume we know team membership (maybe stored
`game.teamAssignments`: like an object mapping seat->team number).
      - We add points to the appropriate team score. We might have been
tracking `team0Points` & `team1Points` in game state.
      - If not present, we can maintain in memory across trick messages by
storing in game item and incrementing it with each trick (like `SET team0Points
= team0Points + :p`).
      - We'll do that: initialize team0Points, team1Points = 0 when setting up
play (maybe at trump selection).
      - Then each trick we update with points.

    + Clear trick:
      - Remove ledSuit, trickCards from game item (or reset them).
      - set currentPlayer to winner (leaders next).

    + Broadcast trickWon and nextPlayer.

    + If last trick of hand:
      - Determine by trickCount==13. We can maintain a trickCount in game
state, increment each trick. When it hits 13, that's end.
      - Or check that each player's hand is empty:
        * We could query Hands table for any remaining cards. Possibly heavy
but number of items small. Or we maintain a `cardsRemaining` count in game and
decrement each play (52 cards total, after 52 plays hand over).
        * Actually easier: after dealing, total cards to play = 52. Each play
message we can decrement a counter in game (like `cardsPlayedCount`).
        * When `cardsPlayedCount == 52` (or 52 plays done), end-of-hand.
```

```
        * Or trickCount==13 (since one trick = 4 plays).
      - Let's use trickCount approach:
        * Initialize trickCount=0 at start of play.
        * Each time a trick completes, increment trickCount.
        * If trickCount==13, end.

      - End-of-hand scoring:
        * We have team0Points, team1Points from trick captures, and kitty/
discard points which we added to bid team when discarding kitty (we should add
those now if not already included).
        * Actually, likely we did not include kitty discard points in
trickPoints because they weren't captured in a trick. We may have stored them
separate.
        * We should incorporate:
          - If we stored `kittyPointsCaptured` for bid team when discarding,
add it to that team's hand points now.
        * Determine bidTeam from game (maybe store `bidTeam` or can identify
from bidWinner seat which team).
        * If bidTeamPoints < winningBid:
          - bid team fails: their score for this hand = 0 + penalty -bid.
          - defenders score = their points (since those are points they
captured).
        * If bidTeamPoints >= winningBid:
          - bid team made bid: they get all their points (including kitty, etc).
          - defenders get theirs.
        * "200 points if sweep"【25†L199-L207】:
          - If bid team took all tricks (so defenders points = 0 and bid team
presumably got 180):
            + Then set bid team's hand score to 200.
          - If defenders took all (bid team 0 points):
            + They get 180 (since all points, no special doubling for them in
our variant).
        * Update overall cumulative scores:
          - `team0Total` and `team1Total` in game item (we can keep these
updated across hands).
          - Add the appropriate hand scores (and subtract bid penalty if fail).
          - If a total >=500:
            - Mark game over and note winner team.

        * Save game state (version++).

      - Broadcast handComplete:
        * Include hand points and new totals, and whether bid team made it
(maybe can infer but could include a boolean or message).
        * Example: action "handComplete", payload like { team0Hand: X,
team1Hand: Y, team0Total: A, team1Total: B, bidTeamMade: true/false }.
      - If game over:
```

```
        * Broadcast gameOver { winner: "Team0" or "Team1" }.

      - If not game over:
        * The game could proceed to next hand. We might allow a short break
then automatically redeal for next hand (with dealer rotating).
        * For simplicity, we might not implement the next hand dealing in code
(we can end the session or require manually starting new game).
        * We can output that a new hand can be started (maybe by host clicking
a button to trigger another shuffle).
        * Possibly beyond scope, but mention that could be done.
```

- **Front-end**:
- "cardPlayed": we already planned:
    - We place card visually on table and remove from that player's hand UI.

- "nextPlayer": highlight whose turn.

    - Also enforce suit following:
    - If it's my turn and a ledSuit is known (we have table state: we can track in state what `ledSuit` is for current trick).
    - If ledSuit exists and I have cards of that suit in my hand, disable cards of other suits (including trump) in my UI.
    - Only enable ledSuit cards.
    - If I have none of ledSuit, all cards enabled (I can play any).
    - Our front-end knows trump suit from earlier.
    - Rook considered as suit=trump always.
    - So implement that: filter myHand for any card suit==ledSuit (exclude Rook unless ledSuit==trump).
    - If found, disable all that are not ledSuit (except Rook if Rook is not ledSuit, treat Rook as trump so if ledSuit != trump, and I have Rook and at least one ledSuit card, I cannot play Rook because must follow ledSuit).
    - We'll likely do this by adding a disabled flag or CSS on cards not allowed.
    - If none found (no ledSuit in hand), allow all.

- "trickWon":

    - Show a message e.g. in game info area: "Player X won trick (+P points)".
    - Possibly update a small display of team points for this hand as we go (some UIs show ongoing hand points).
    - We then clear the table after maybe 2 seconds to let players see the last trick's cards. Could just clear immediately for simplicity.
    - We'll do: after receiving trickWon, we can remove the 4 card elements from table UI (or reset that state).
    - If winner leads next, "nextPlayer" event will come with seat = winner, so that will signal next trick leader which our nextPlayer handler already covers.
    - Also mark those four cards as played (they were already removed from hands and shown on table, so removing from table is just clearing some state array that held current trick's played cards).

- Also update hand score in UI (we can track in state how many points each team got so far in this hand).
- We can accumulate from trickWon messages: each trick adds points to a team.
- We know which team because we know players' teams.

• "handComplete":

- Display the results: probably in an overlay or at least in game info section:
- "Hand over. Team0 scored X, Team1 scored Y."
- If bidTeamMade false: "Bidder set - (bid) points."
- Show new total scores: "Total Score: Team0 A, Team1 B."
- Possibly a "Continue" button to proceed if not game over, or automatically could start after some seconds if implementing multi-hand.
- We might freeze UI until user acknowledges (like a "OK" button).
- Alternatively, just show the info and allow host to start next hand by a button (which would trigger similar to initial dealing but with new dealer).
- Maybe out-of-scope to implement next hand logic fully (like rotating dealer, etc). We can mention that might be manual or not implemented.

• "gameOver":

- Show final message "Game Over - Team X wins!" maybe alert or banner.
- Possibly disable all controls and allow starting a new game (maybe just instruct to refresh or go back to lobby).

Given the complexity, it's fine if not all polished, but core logic must be correct.

Finally, after implementing, test small unit-like steps: - We can simulate with a single player playing out a few cards, but without multiple players the messaging is partial. Best done with actual multi-client after deployment.

Now we have fully implemented the game rules and can consider the project complete for a single hand game.

---

1 5 6 7 9 10 11 12 14 15 16 17 19 21 22 23 24 25 26 27 Rook Game Rules Clarification and Implementation Notes.pdf
file://file_000000008f1c71f8a99de81e7de3f8fb

2 node.js - Trying to understand lambda function that is part of the logic of api gateway websockets connection - Stack Overflow
https://stackoverflow.com/questions/64759924/trying-to-understand-lambda-function-that-is-part-of-the-logic-of-api-gateway-we

3 Handle conditional write errors in high concurrency scenarios with Amazon DynamoDB | AWS Database Blog
https://aws.amazon.com/blogs/database/handle-conditional-write-errors-in-high-concurrency-scenarios-with-amazon-dynamodb/

4 Building a serverless multi-player game that scales | AWS Compute Blog
https://aws.amazon.com/blogs/compute/building-a-serverless-multiplayer-game-that-scales/

8 13 18 20 Rook (card game) - Wikipedia
https://en.wikipedia.org/wiki/Rook_(card_game)