1) Create Game

1. Endpoint Name
Create Game

2. HTTP Method & Route
`POST /api/games`

3. Purpose

Create a new game lobby and make the current user the host. Returns the initial lobby state and join codes (if we use join code or only invite)

4. Authorization

- Must be authenticated (valid session/JWT).
- Users must not already be host of another active game in `"lobby"` or `"playing"` state (to avoid multi-host).

5. Request Body example;

```
{
  "name": "Miguel's Word Room",
  "max_players": 5,
  "is_private": true,
  "difficulty": "normal, hard",
  "round_limit": 3          // still unsure how my many rounds
}
```

6. Validation Checks (be explicit)

- The user is authenticated.
- `name` is non-empty and max length is 10 chars).
- `max_players` is an integer between 2 and 8.(for multiplayer)
- `difficulty` is in the allowed set: `["easy","normal","hard"]`.
- If `round_limit` provided, it's an integer ≥ 1.
- The user is not already hosting an active game (`games.host_id = user.id AND state IN ('lobby','playing')`).
- No DB or server error during creation.

7. State Updates (DB changes)

- INSERT into `games`:
  - `host_id = current_user.id`
  - `state = "lobby"`
  - `max_players`, `is_private`, `difficulty`, `round_limit`, etc.
  - Optionally generate a `join_code` like 6-char alphanumeric.
- INSERT into `game_players`:
  - `game_id = new_game.id`
  - `user_id = current_user.id`
  - `is_host = true`
  - `score = 0`

8. Success Response

- **201 Created** (it’s literally creating a resource; it doesn’t need async processing)

```
{
  "game": {
    "id": 123,
    "code": "ABCD12",
    "name": "Miguel's Word Room",
    "state": "lobby",
    "max_players": 5,
    "is_private": true,
    "difficulty": "normal",
    "round_limit": 3,
    "host_id": 44,
    "players": [
      {
        "id": 42,
        "name": "Miguel",
        "is_host": true,
        "score": 0
      }
    ]
  }
}
```

9. Error Cases

- **401 Unauthorized** – user not logged in.
- **400 Bad Request** – invalid body fields (bad `max_players`, bad `difficulty`, etc.).
- **409 Conflict** – user already hosting another active game.
- **500 Internal Server Error** – DB failure.

Data:

```
{
  "game_id": 123,
  "code": "ABCD12",
  "name": "Miguel's Word Room",
  "state": "lobby",
  "max_players": 6
```

  - ○ `}`

10. Socket.io Event

Event name: `game:lobby:created`

Triggered by: `POST /api/games` **(Create Game) – on successful lobby creation.**

Who receives it: Host only (`user:<host_id>`**).**

Data sent:

```
{
  "game_id": 123,
  "code": "ABCD12",
  "name": "Miguel's Word Room",
  "state": "lobby",
  "max_players": 5,
  "is_private": true,
  "difficulty": "normal",
  "round_limit": 3,
  "host_id": 44
}
```

2) Join Game

1. Endpoint Name
Join Game

2. HTTP Method & Route
`POST /api/games/:game_id/join`

(or `POST /api/games/join` with `{ "code": "ABCD12" }` in body – pick one design)

3. Purpose
Add an authenticated player to an existing game lobby.

4. Authorization

- Must be authenticated.
- Game must allow new players:
    - in `"lobby"` state
    - not full
    - not banned / blocked.

5. Request Body

If joining by `game_id`, body can be optional:

```
{
  "display_name": "Miguel6ix"      // optional override of profile
name
}
```

If joining by `code` instead:

```
{
  "code": "ABCD12",
  "display_name": "Luis"
}
```

6. Validation Checks

- The user is authenticated.

- Game exists (`games.id = :game_id` OR by `code`).
- `game.state === "lobby"`.
- `game_players` count < `game.max_players`.
- User is **not already** in this game (`game_players` row for (game_id, user_id) does NOT already exist).
- If game is private and uses invite list:
  - user is allowed (check `game_invites` or similar).
- If a user is banned from the game, reject.
- `display_name` length ≤ 30, no disallowed characters (if you use it).

7. State Updates

- INSERT into `game_players`:
  - `game_id`, `user_id`, `is_host = false`, `score = 0`.
- UPDATE `games.player_count` (if you track it).

8. Success Response

- **202 Accepted** (follow "validate → accept → broadcast" pattern)

```
{
  "status": "accepted",
  "game": {
    "id": 123,
    "name": "Miguel's Word Room",
    "state": "lobby",
    "max_players": 6,
    "players": [
      { "id": 42, "name": "Miguel", "is_host": true, "score": 0 },
      { "id": 77, "name": "Tona", "is_host": false, "score": 0 }
    ]
  },
  "me": {
    "player_id": 77,
    "is_host": false
  }
}
```

9. Error Cases

- **401 Unauthorized** – not logged in.

- **404 Not Found** – game doesn't exist or code invalid.
- **403 Forbidden** – private game, and user not invited / banned.
- **409 Conflict** – game full, already started, or user already in game.
- **400 Bad Request** – invalid request body.

Data:

```
{
  "game_id": 123,
  "player": {
    "id": 77,
    "name": "Luis",
    "is_host": false,
    "score": 0
  }
}
```

  -
- **Event:** `game:lobby:updated`
  - **Scope:** `room:game:<game_id>`
  - **Trigger:** when the lobby roster changes.

Data:

```
{
  "game_id": 123,
  "players": [
    { "id": 42, "name": "Miguel", "is_host": true, "score": 0 },
    { "id": 77, "name": "Tona", "is_host": false, "score": 0 }
  ],
  "max_players": 5
}
```

  -
- **Event:** `game:private:update`
  - **Scope:** `user:<new_player_user_id>` only
  - **Trigger:** send any private info (e.g. your personal stats or secret role, if you add that later).
  - **Data:** game-specific private fields.

10. Socket.io Event

Event name: `game:lobby:updated`

Triggered by: `POST /api/games/:game_id/join` **(Join Game)**

whenever a player joins (or later leaves/is kicked). ￼

Who receives it: All players in the lobby (`room:game:<game_id>`**).**

Data sent:

```
{
  "game_id": 123,
  "players": [
    { "id": 42, "name": "Miguel", "is_host": true,  "score": 0 },
    { "id": 77, "name": "Luis",   "is_host": false, "score": 0 }
  ],
  "max_players": 5
}
```

3) Start Game

1. Endpoint Name
Start Game

2. HTTP Method & Route
`POST /api/games/:game_id/start`

3. Purpose
Transition from lobby to playing: lock in the player list, initialize the first round, choose the secret word(s) randomly, and set the current player/turn order if applicable.

4. Authorization

- Must be authenticated.
- Must be a player in this game.
- Must be the **host** (`game.host_id === user.id`).
- The game must be in `"lobby"` state.

5. Request Body

Optional, if you want to override default settings:

```
{
  "difficulty": "hard",        // overrides if host changes it last
minute
  "round_limit": 3
}
```

6. Validation Checks

- The user is authenticated.
- Games exist.
- The user is in `game_players` for this game.
- User is host (`game.host_id === user.id`).
- `game.state === "lobby"`.
- `player_count >= min_players` (e.g. at least 2).
- If the body contains new settings, they are valid (difficulty/round_limit checks).
- The game is not already `"playing"` or `"ended"`.

7. State Updates

- UPDATE `games`:
    - `state = "playing"`
    - lock in `difficulty`, `round_limit`.
    - set `current_round = 1`.
- Generate secret word(s) based on difficulty:
    - This is **server-side only**; never sent to clients.
    - Possibly insert the initial row in the rounds table: (`game_id`, `round_number = 1`, `secret_word = "APPLE"`, `status = "active"`, etc.).
- Initialize turn order if needed.
- (All in a transaction to avoid partial state.)

8. Success Response

- **202 Accepted**

```
{
  "status": "accepted",
  "game": {
    "id": 123,
    "state": "playing",
    "current_round": 1,
    "difficulty": "hard",
    "round_limit": 5
  }
}
```

9. Error Cases

- **401 Unauthorized** – not logged in.
- **403 Forbidden** – user not in game OR not host.
- **404 Not Found** – game doesn't exist.
- **409 Conflict** – game already started or ended; or not enough players.

Data:

```
{
  "game_id": 123,
  "state": "playing",
  "current_round": 1,
  "players": [
    { "id": 42, "name": "Miguel", "score": 0 },
    { "id": 77, "name": "Luis", "score": 0 }
  ]
}
```

-
- **Event:** `game:state:update`
    - **Scope:** `room:game:<game_id>`
    - **Trigger:** after initial round/turn state is created.

Data:

```
{
  "game_id": 123,
  "state": "playing",
  "current_round": 1,
  "board": {},           // empty or initial board state for your game
  "turn": {
    "current_player_id": 42
  }
}
```

-
- **Event:** `game:private:update` (optional)
    - **Scope:** each `user:<user_id>`
    - **Trigger:** if there's any private per-player info to send (secret role, personal hints, etc.).
    - **Data:** any private fields.

10. Socket.io Event

Event name: `game:state:update`

Triggered by: `POST /api/games/:game_id/start` **(Start Game)**

when transitioning from lobby → playing and whenever global game state changes (new round, turn change, etc.). ￼

Who receives it: All players in the game (`room:game:<game_id>`**).**

Data sent:

```
{
  "game_id": 123,
  "state": "playing",
  "current_round": 1,
  "board": {},                // public board state for your game
  "turn": {
    "current_player_id": 42
  }
}
```

4) Get Game State

1. Get Game State

2. HTTP Method & Route
`GET /api/games/:game_id`

3. Purpose
Return the current game state to the requesting player. Includes **public state for everyone** and a **private section (`me`)**that contains only their own private info.

4. Authorization

- Must be authenticated.
- If game is private:
    - user must be in `game_players` **OR** must be allowed as a spectator (if you support it).
- For public games, you can allow read-only spectators (design choice).

5. Request Body

None. You can use query params if you want:

- `GET /api/games/:game_id?include_private=true`

The server should ignore `include_private` if the user is not a player.

6. Validation Checks

- User is authenticated.
- Game exists.
- If `game.is_private === true`, check:
    - user is a player OR is an allowed spectator.
- If `include_private=true`, verify:
    - user is in `game_players` for this game.
- Game state is consistent (e.g. at least one host player exists).

7. State Updates

- None — this is a read-only endpoint.
- Optionally update `last_seen_at` for this player in `game_players`, but that's minor.

8. Success Response

- **200 OK**

Example response structure:

```
{
  "game": {
    "id": 123,
    "name": "Miguel's Word Room",
    "state": "playing",
    "current_round": 2,
    "round_limit": 5,
    "difficulty": "normal",
    "host_id": 42
  },
  "players": [
    { "id": 42, "name": "Miguel", "score": 10, "is_host": true,
"is_connected": true },
    { "id": 77, "name": "Luis", "score": 8, "is_host": false,
"is_connected": true }
  ],
  "public_state": {
    "board": {
      "revealed_letters": ["A", "E"],
      "previous_guesses": [
        { "player_id": 42, "guess": "APPLE", "result": "" },
        { "player_id": 77, "guess": "GRAPE", "result": "" }
      ]
    },
    "turn": {
      "current_player_id": 77,
      "seconds_left": 23
    }
  },
  "me": {
    "player_id": 77,
    "private_notes": [],     // any per-player private state (if you
add it)
    "my_guesses": [
      { "guess": "GRAPE", "result": "" }
    ]
  }
}
```

Notice:

- **Public**: scores, previous guesses, current turn, board, etc.
- **Private (`me`)**: only things relevant to that user that should not be visible to others (if any). The **secret word NEVER appears** anywhere in the response.

9. Error Cases

- **401 Unauthorized** – not logged in.
- **404 Not Found** – game doesn't exist.
- **403 Forbidden –** private game and user is neither player nor allowed spectator.

10. Socket.io Event

Event name: `game:private:update`

Triggered by:

- **After Join Game for the joining player (to send their private info).**
- **After Start Game if there is any per-player secret data (roles, hints, etc.).**

Who receives it: Exactly one player (`user:<user_id>`).

Data sent (example, matches your me section):

```
{
  "game_id": 123,
  "player_id": 77,
  "private_notes": [],
  "my_guesses": [
    { "guess": "GRAPE", "result": "" }
  ]
  // plus any other per-player secret fields we add later
}
```

Card Game Action Endpoints

5) Draw Card

1. Endpoint Name

Draw Card

2. HTTP Method & Route

POST /api/games/:game_id/draw

3. Purpose

Let the current player draw one (or more) cards from the deck into their hand.

4. Authorization

- Must be authenticated.
- Must be an active player in this game.
- Game must be in "playing" state.
- Must be the current turn player (unless special rules allow off-turn draws).

5. Request Body

```
{
  "count": 1       // optional; default = 1
}
```

6. Validation Checks

- User is authenticated.
- Game exists.

- game.state === "playing".

- User is in game_players for this game.

- User is the current turn player.

- count (if provided) is an integer ≥ 1 and ≤ max allowed.

- Deck has enough cards (or rules allow drawing fewer).

7. State Updates

- Remove up to count cards from the deck.

- Insert them into this player's hand.

- Update deck_count if tracked.

- Optionally log in game_actions.

8. Success Response

202 Accepted

```
{
  "status": "accepted",
  "game_id": 123,
  "draw": {
    "player_id": 77,
    "count": 1
  },
  "public_state": {
    "deck_count": 37
  },
  "me": {
    "player_id": 77,
  "hand": [
   { "id": "C7", "type": "letter", "value": "C" },
   { "id": "A3", "type": "letter", "value": "A" }
  ]
 }
}
```

9. Error Cases

- 401 Unauthorized – not logged in.

- 403 Forbidden – user not in game or not current player.

- 404 Not Found – game doesn’t exist.

- 409 Conflict – game not in playing state or deck empty.

- 400 Bad Request – invalid count.

10. Socket.io Events

game:state:update (room:game:<game_id>)

- Sent to all players.

- Includes public_state with deck_count and last_action of type "draw".

game:private:update (user:<drawing_player_user_id>)

- Sent only to the drawing player.

- Includes updated private hand.

6) Play Card

1. Endpoint Name

Play Card

2. HTTP Method & Route

POST /api/games/:game_id/play-card

3. Purpose

Let the current player play a card from their hand to the board/table.

4. Authorization

- Must be authenticated.
- Must be a player in this game.
- Must be the current turn player.
- Game must be in "playing" state.

5. Request Body

```
{
  "card_id": "C7",
  "target": {
    "slot": 2
  }
}
```

6. Validation Checks

- User is authenticated.
- Game exists and state === "playing".
- User is in game_players.
- User is the current turn player.
- card_id belongs to this player's hand.

- target is valid (slot exists, not invalid per rules).
- Any game-specific rule checks pass.

7. State Updates

- Remove the card from this player's hand.
- Place the card onto the board in the target slot.
- Update scores/word/other public state as needed.
- Optionally log in game_actions.

8. Success Response

202 Accepted

```
{
  "status": "accepted",
  "game_id": 123,
  "action": {
    "type": "play_card",
    "player_id": 77,
    "card_id": "C7",
    "target": { "slot": 2 }
  },
  "public_state": {
    "board": {
      "slots": [
        { "slot": 1, "card": null },
        { "slot": 2, "card": { "id": "C7", "public_value": "C" } }
      ]
    },
    "scores": [
      { "player_id": 42, "score": 10 },
      { "player_id": 77, "score": 12 }
    ]
  },
  "me": {
    "player_id": 77,
    "hand": [
      { "id": "A3", "type": "letter", "value": "A" }
    ]
  }
}
```

9. Error Cases

- 401 Unauthorized – not logged in.
- 403 Forbidden – user not in game or not current player.
- 404 Not Found – game or card not found in player's hand.
- 409 Conflict – invalid target or rule violation.
- 400 Bad Request – missing card_id or invalid target.

10. Socket.io Events

game:state:update (room:game:<game_id>)

- Sent to all players, including board update and last_action "play_card".

game:private:update (user:<playing_player_user_id>)

- Sent only to the playing player with updated hand.

7) Discard Card

1. Endpoint Name

Discard Card

2. HTTP Method & Route

POST /api/games/:game_id/discard

3. Purpose

Allow a player to discard a card from their hand to the discard pile.

4. Authorization

- Must be authenticated.

- Must be a player in the game.

- Game must be in "playing" state.

- Usually must be the current turn player (depending on rules).

5. Request Body

```
{
  "card_id": "C7",
  "reason": "hand_limit"   // optional
}
```

6. Validation Checks

- User is authenticated.

- Game exists and state === "playing".

- User is in game_players.

- If required by rules, user is current player.

- card_id exists in player's hand.

- Discard is allowed given current rules/phase.

7. State Updates

- Remove card from player's hand.

- Add card to discard pile.

- Optionally trigger follow-up logic (e.g., auto-draw).

- Optionally log in game_actions.

8. Success Response

202 Accepted

```
{
  "status": "accepted",
  "game_id": 123,
  "action": {
    "type": "discard_card",
    "player_id": 77,
    "card_id": "C7",
    "reason": "hand_limit"
  },
  "public_state": {
    "discard_pile_count": 12,
    "last_discard": {
      "player_id": 77,
      "card_public": { "id": "C7", "public_value": "C" }
    }
  },
  "me": {
    "player_id": 77,
    "hand": [
      { "id": "A3", "type": "letter", "value": "A" }
    ]
  }
}
```

9. Error Cases

- 401 Unauthorized – not logged in.
- 403 Forbidden – not allowed to discard at this time.
- 404 Not Found – game or card not found in player's hand.
- 409 Conflict – rule violation (e.g., minimum hand size).
- 400 Bad Request – missing card_id.

10. Socket.io Events

game:state:update (room:game:<game_id>)

- Broadcast updated discard_pile_count and last_action "discard_card".

game:private:update (user:<discarding_player_user_id>)

- Send updated private hand to the discarding player.