WebQuiz実装計画書 (完全版 v12)

v12での主な変更点

- 第三者が読んでも理解できるよう、これまで「変更なし」として省略していたセクション（5.1, 5.2, 5.4, 6）に、元の計画書に基づいた詳細な仕様をすべて記述しました。これにより、このドキュメント単体で全仕様が完結するようになりました。

1. 概要

本ドキュメントは、Node.js/Socket.IOサーバーとReactクライアントで構成されるリアルタイムWebクイズアプリケーションの実装仕様を定義する。データ構造、状態遷移、Socket.IOイベント、ゲームフロー、エラーハンドリングについて網羅的な仕様を定めることを目的とする。

2. データ構造の設計

サーバー全体のデータ構造 (概念)

```
{
  "players": {
    "socket_id_1": { "name": "プレイヤーA" },
    "socket_id_2": { "name": "プレイヤーB" }
  },
  "rooms": {
    "a1b2c": {
      /* Roomオブジェクト */
    }
  }
}
```

Room (ルーム)

```
{
  "id": "a1b2c",
  "hostId": "socket_id_of_host",
  "state": "waiting",
  "players": {
    "socket_id_1": {
      "id": "socket_id_1",
      "name": "プレイヤーA",
      "score": 0
    }
```

```
  },
  "gameData": {
    "questions": [ /* Questionオブジェクトの配列 */ ],
    "currentQuestionIndex": 0,
    "questionState": "idle",
    "readyPlayerIds": [],
    "answeredPlayerIds": [],
    "activeAnswer": null
  }
}
```

Question (問題)

```
{
  "id": "q001",
  "text": "日本の首都はどこでしょう？",
  "answer_data": [
    { "char": "と", "choices": ["と", "ち", "の", "そ"] },
    { "char": "う", "choices": ["う", "あ", "お", "え"] },
    { "char": "き", "choices": ["き", "さ", "し", "け"] },
    { "char": "ょ", "choices": ["ょ", "ゅ", "ゃ", "っ"] },
    { "char": "う", "choices": ["う", "ぬ", "め", "ろ"] }
  ]
}
```

3. 状態遷移

3.1. アプリケーション全体のフロー

現在の画面	トリガー	次の画面	備考
(接続直後)	-	名前登録	
名前登録	registerPlayer成功	ホーム（ロビー）	
ホーム（ロビー）	createRoom / joinRoom 成功	ルーム待機	
ルーム待機	leaveRoom / 接続断	ホーム（ロビー）	

ルーム待機	ホストがstartGame	ゲーム画面	
ゲーム画面	ゲーム終了 (gameFinished)	全体結果画面	
ゲーム画面	ホスト切断 (roomClosed)	ホーム（ロビー）	強制的にロビーに戻される。
全体結果画面	(自動タイマー)	ルーム待機	ゲーム終了後、同じメンバーで待機状態に戻る。

3.2. サーバー側ルーム状態 (Room.state)

現在の状態	トリガーイベント	次の状態	説明
(存在しない)	createRoom	waiting	ルームが生成され、待機状態になる。
waiting	ホストがstartGame	playing	ゲームが開始される。
playing	ゲーム終了条件を満たす	finished	全ての問題が終了し、結果表示状態になる。
finished	(自動タイマー)	waiting	全体結果表示後、再び待機状態に戻る。
waiting/playing/finished	ホストが切断	(ルーム削除)	ルームは即座に削除される。

3.3. 1問ごとのライフサイクル (gameData.questionState)

Room.stateがplayingの間に、gameData.questionStateは1問ごとに以下のライフサイクルを繰り返す。

現在のサブ状態	トリガー	次のサブ状態	説明
idle	(ゲーム進行ロジック)	presenting	新しい問題のライフサイクル開始。
presenting	(自動)	reading	問題の開始合図表示後、問題読み上げへ。早押し受付開始。

reading	プレイヤーがbuzz	answering	早押し成功。activeAnswerが生成される。
reading	全プレイヤーが**questionReady**を送信後、10秒経過	result	誰も早押しせず問題終了。ライフサイクル完了。
answering	submitCharacter (正解/続きあり)	answering	次の文字の選択肢を提示。
answering	submitCharacter (全問正解)	result	正解！activeAnswerは破棄。ライフサイクル完了。
answering	submitCharacter (不正解/他者あり)	reading	回答権を失う。activeAnswerは破棄され、問題読み上げ再開。
answering	submitCharacter (不正解/他者なし)	result	全員お手つき。activeAnswerは破棄。ライフサイクル完了。
result	(自動タイマー)	idle	結果表示後、次の問題の準備へ移行。

4. Socket.IO イベント定義

4.1. クライアント → サーバー (C → S)

イベント名	データ **(Payload)**	説明
registerPlayer	{ playerName: string }	プレイヤー名をサーバーに登録する。
createRoom	(なし)	新しいルームの作成を要求する。サーバーは送信元IDからプレイヤーを特定する。
joinRoom	{ roomId: string }	既存のルームへの参加を要求する。
leaveRoom	{ roomId: string }	現在のルームから退室する。
startGame	{ roomId: string }	ホストのみ がゲームの開始を要

		求する。
questionReady	{ roomId: string }	問題文の表示（読み上げ演出）を完了したことを通知する。タイムアウト計測開始のトリガーとなる。
buzz	{ roomId: string }	早押しボタンを押したことを通知する。
submitCharacter	{ roomId: string, selectedChar: string }	回答者が選択肢から **1**文字 選択したことを通知する。

4.2. サーバー → クライアント (S → C)

イベント名	データ **(Payload)**	説明
playerRegistered	{ playerName: string, playerId: string }	プレイヤー名の登録が成功したことを本人に通知する。
roomListUpdate	[{ id, playerCount, state }, ...]	ロビーの全クライアント にルーム一覧を送信。
joinedRoom	{ room: Room, playerId: string }	参加者本人 にルーム情報と自身のIDを送信。
roomUpdated	{ room: Room }	ルーム内の全プレイヤー に更新されたルーム情報をブロードキャスト。
roomClosed	{ roomId: string, reason: string }	ルームが解散したことを通知する。（例: reason: "ホストが切断しました。"）
gameStarted	{ room: Room }	ルーム内の全プレイヤー にゲーム開始を通知。
newQuestion	{ question: Question, questionIndex: number, room: Room }	ルーム内の全プレイヤー に新しい問題を出題。
readingStarted	{ room: Room }	ルーム内の全プレイヤー に問題読み上げ開始と早押し受付開始を通知する。
buzzerResult	{ winnerId: string, room: Room }	ルーム内の全プレイヤー に早押し成功者を通知。

nextChoice	{ choices: string[] }	回答者本人 に、回答すべき次の文字の選択肢を送信。
answerResult	{ playerId: string, isCorrect: boolean, isFinal: boolean, correctAnswer?: string[] }	ルーム内の全プレイヤー に回答試行の結果を通知。
scoreUpdated	{ players: Player[] }	ルーム内の全プレイヤー に更新されたスコア情報を通知。
gameFinished	{ room: Room }	ルーム内の全プレイヤー にゲーム終了と最終結果を通知。
errorOccurred	{ code: string, message: string }	特定のクライアント にエラーを通知。

5. ゲームフローシーケンス

5.1. 接続とプレイヤー名登録

1. クライアントがサーバーに接続する。
2. クライアントのUIは「名前登録画面」を表示する。
3. ユーザーが名前を入力し、送信ボタンを押す。
4. クライアントは registerPlayer イベントを、入力された名前をペイロードに含めてサーバーに送信する。
5. サーバーは受け取った名前を検証し、問題がなければ送信元の socket.id と関連付けてサーバー全体の players リストに保存する。
6. サーバーは playerRegistered イベントを送信元のクライアントに返す。
7. クライアントは playerRegistered を受信後、プレイヤー情報を内部に保存し、画面を「ホーム画面（ロビー）」へ遷移させる。

5.2. ルーム作成からゲーム開始まで

1. クライアント**A** がホーム画面で「ルーム作成」ボタンを押す。
2. クライアントAは createRoom イベントをサーバーに送信する。
3. サーバーは送信元の socket.id からプレイヤー情報を特定し、新しい Room オブジェクトを生成する。クライアントAをホストとしてルームに追加する。
4. サーバーは joinedRoom イベントをクライアントAに送信し、作成されたルームの全情報を通知する。
5. サーバーは roomListUpdate イベントをロビーにいる全クライアントにブロードキャストし、ルーム一覧を更新させる。
6. クライアント**B** がホーム画面でクライアントAのルームを選択し、「参加」ボタンを押す。
7. クライアントBは joinRoom イベントを、ルームIDをペイロードに含めてサーバーに送信す

る。
8. サーバーは指定されたルームにクライアントBを追加する。
9. サーバーは joinedRoom イベントをクライアントBに送信する。
10. サーバーは roomUpdated イベントをルーム内の**全プレイヤー（AとB）**にブロードキャストし、参加者リストの変更を通知する。
11. ホスト（クライアント**A**）がルーム画面で「ゲーム開始」ボタンを押す。
12. クライアントAは startGame イベントをサーバーに送信する。
13. サーバーは Room.state を playing に変更し、gameStarted イベントをルーム内の全プレイヤーにブロードキャストする。その後、最初の問題のライフサイクルを開始する（5.3へ）。

5.3. 問題回答サイクル **(詳細版)**

1. ▼ 新しい問題のライフサイクル開始 ▼
   サーバーはgameData内のquestionStateをpresentingに設定し、answeredPlayerIdsとreadyPlayerIdsを空にし、activeAnswerをnullに設定する。更新されたRoomオブジェクトを含むnewQuestionをブロードキャスト。
2. クライアントは「問題！」などの開始合図を表示。
3. 数秒後、サーバーはquestionStateをreadingに設定し、**readingStarted** をブロードキャスト。
4. readingStartedを受信したクライアントは、問題文の表示（アニメーションなど）を開始し、同時に早押しボタンを有効化する。
5. 各クライアントは、問題文の表示が完了したら **questionReady** イベントをサーバーに送信する。
6. サーバーは、ルーム内の全プレイヤーからquestionReadyイベントを受信するまで待機する。
7. 全員から受信したことを確認後、サーバーは内部で**10**秒のタイムアウト用タイマーを開始する。
8. （早押し成功）プレイヤーCがbuzzを送信。
   a. サーバーはタイムアウト用タイマーを停止し、questionStateをansweringに変更。gameData.activeAnswerオブジェクトを生成し、buzzerResultをブロードキャスト。
   b. (以降、v10と同様の回答フロー)
9. （時間切れ）サーバー内部の10秒タイマーが満了した場合、サーバーはanswerResult({ isCorrect: false, isFinal: true, ... })をブロードキャストし、誰も正解しなかったことを通知する。
10. ▼ 現在の問題のライフサイクル完了 ▼
    サーバーはquestionStateをresultに設定し、activeAnswerがnullであることを保証する。クライアントは1問ごとの結果を表示。
11. 数秒後、サーバーはゲーム終了条件を判定し、次の問題へ進むかゲームを終了するかを決定する。

5.4. プレイヤーの切断処理

以下の処理は、サーバーがクライアントとの接続断を検知した際（Socket.IOの標準 disconnect イベント）に実行される。

1. サーバーは、切断したクライアントの socket.id を取得する。
2. サーバーは、その socket.id を元に、サーバー全体の players リストから該当プレイヤーの情報を削除する。
3. サーバーは、そのプレイヤーが参加していたルームを検索する。
4. ルームに参加していた場合:
   - ホストが切断した場合:
     a. サーバーは、ルーム内の残りの全プレイヤーに roomClosed イベントを送信し、ルームが解散したことを通知する。
     b. サーバーの rooms リストから、そのルームオブジェクトを完全に削除する。
   - ホスト以外のプレイヤーが切断した場合:
     a. サーバーは、ルームの players オブジェクトから、該当プレイヤーを削除する。
     b. サーバーは、ルーム内の残りの全プレイヤーに roomUpdated イベントをブロードキャストする。
5. 最後に、サーバーは roomListUpdate イベントをロビーにいる全クライアントにブロードキャストし、ルームリストの変更（参加者数の減少やルームの消滅）を反映させる。
6. ルームに参加していなかった場合（ロビーにいただけの場合）は、プレイヤーリストからの削除のみで処理は完了する。

6. エラーハンドリング

クライアントへの通知が必要なエラーケースと、errorOccurred イベントで送信する内容を以下に定義する。

エラーコード	メッセージ	説明
ROOM_NOT_FOUND	"ルームが見つかりません。"	存在しない roomId で参加しようとした。
ROOM_FULL	"このルームは満員です。"	満員のルームに参加しようとした。
INVALID_NAME	"無効なプレイヤー名です。"	プレイヤー名が空、または長すぎる。
NOT_HOST	"ホストではありません。"	ホスト以外のプレイヤーが startGame を送信した。
ALREADY_PLAYING	"ゲームはすでに開始されていま	ゲーム中のルームに参加しようと

	す。"	した。
NOT_REGISTERED	"プレイヤー名が登録されていません。"	名前登録をせずにルーム作成/参加を試みた。