Interactive Quiz Game - Code Documentation

Table of Contents

Project Overview

This application is a real-time interactive quiz platform enabling multiplayer quizzing with instant feedback, scoring, and result visualization. The system utilizes WebSockets for real-time communication, with a PostgreSQL database for persistent storage.

Key Features

1. **Quiz Creation and Management**
    - Hosts can create quizzes with customizable questions and multiple answer options
    - Support for single and multiple-choice questions
    - Option to include "decoy" answers
    - Ability to reconduct previous quizzes with modifications
2. **Real-time Interaction**
    - Live quiz participation via short codes or QR codes
    - Dynamic updates as participants join and submit answers
    - Instant scoring and feedback
3. **Flexible Game Mechanics**
    - Configurable timer settings for questions
    - Support for different question types (single-select, multi-select)
    - Host controls for starting, monitoring, and ending quizzes
4. **Advanced Scoring**
    - Score calculation based on correct answers (non-decoy selections)
    - Tiebreaker based on submission time
    - Detailed performance analytics
5. **User Experience**

    - Responsive design for both hosts and participants
    - Visual feedback for correct/incorrect answers
    - Leaderboard and winner announcements

Architecture

The project follows a full-stack JavaScript architecture:

- **Frontend**: React with TailwindCSS and Shadcn UI components
- **Backend**: Express.js server with RESTful API and WebSocket endpoints
- **Database**: PostgreSQL with Drizzle ORM
- **API Communication**:
    - REST API for CRUD operations
    - WebSockets for real-time game state updates
- **Authentication**: Passport.js with session-based auth

Database Schema

Main Tables

Users

```
export const users = pgTable("users", {
  id: serial("id").primaryKey(),
  username: text("username").notNull().unique(),
  password: text("password").notNull(),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});
```

Quizzes

```
export const quizzes = pgTable("quizzes", {
  id: serial("id").primaryKey(),
  hostId: integer("host_id").references(() => users.id),
  hostName: text("host_name").notNull(),
  subject: text("subject").notNull(),
  section: text("section").notNull(),
  shortCode: text("short_code").notNull().unique(),
  timer: integer("timer").default(30).notNull(),
  startTime: timestamp("start_time").notNull(),
  endTime: timestamp("end_time").notNull(),
  status: text("status", { enum: Object.values(QuizStatus)
}).default(QuizStatus.SCHEDULED).notNull(),
  questions: jsonb("questions").$type<any[]>().notNull(),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});
```

Participants

```
export const participants = pgTable("participants", {
  id: serial("id").primaryKey(),
  quizId: integer("quiz_id").references(() => quizzes.id),
  playerName: text("player_name").notNull(),
  answers: jsonb("answers").$type<string[]>().default([]),
  submittedAt: timestamp("submitted_at"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});
```

Frontend Components

Core Components

1. Host Dashboard (`host-dashboard.tsx`)

Manages quiz creation, monitoring, and controls for the quiz host.

Key functions:

- `HostDashboard()`: Main component for the host view
- `handleReconductQuiz(quiz)`: Allows reusing a previous quiz with modifications
- `handleStartQuiz()`: Initiates a quiz and updates its status
- `handleEndQuiz()`: Concludes a quiz and calculates final results

2. Player Area (`player-area.tsx`)

Interface for quiz participants to join and answer questions.

Key functions:

- `PlayerArea()`: Main component for the player view
- `handleJoinQuiz()`: Allows a player to join a quiz using a short code
- `handleAnswerSubmit()`: Submits player's answers to questions
- `handleOptionSelect()`: Manages selection of answers for different question types

3. Winner Announcement (`winner-announcement.tsx`)

Displays quiz results and ranks participants.

Key functions:

- `WinnerAnnouncement({ quizId })`: Shows winners and complete rankings
- Score calculation logic:

```
// Count each selected answer that is not a decoy as correct
for (const selectedOption of selectedAnswers) {
  // If the selectedOption index is within bounds and not a decoy, it's
correct
  if (
    selectedOption >= 0 &&
    selectedOption < question.isDecoy.length &&
    !question.isDecoy[selectedOption]
  ) {
    correctAnswersCount++;
  }
}
```

4. QR Code Share (`qr-code-share.tsx`)

Generates a QR code for easy quiz sharing.

Key function:

- `QRCodeShare({ quizId })`: Creates a shareable QR code with quiz URL

UI Components

A comprehensive set of Shadcn UI components are used throughout the application, including:

- Form elements (inputs, buttons, checkboxes)
- Layout components (cards, modals, dialogs)
- Feedback components (toasts, progress indicators)
- Data display components (tables, charts)

Backend Services

Express Routes

Quiz Management

- `POST /api/quizzes`: Create a new quiz
- `GET /api/quizzes`: Get all quizzes for the current host
- `GET /api/quizzes/:id`: Get a specific quiz
- `PATCH /api/quizzes/:id/status`: Update quiz status
- `DELETE /api/quizzes/:id`: Delete a quiz
- `GET /api/quizzes/:id/results`: Get quiz results

Participant Management

- `POST /api/participants`: Add a participant to a quiz
- `GET /api/participants`: Get all participants
- `PATCH /api/participants/:id`: Update participant answers

Authentication

- `POST /api/auth/register`: Register a new user

- `POST /api/auth/login`: Log in a user
- `GET /api/auth/logout`: Log out the current user
- `GET /api/user`: Get the current authenticated user

Storage Service

The DatabaseStorage class implements the IStorage interface and provides methods for:

- User management: `getUser()`, `getUserByUsername()`, `createUser()`
- Quiz management: `createQuiz()`, `getQuiz()`, `getQuizByShortCode()`, `getQuizzesByHost()`, `updateQuizStatus()`
- Participant management: `addParticipant()`, `getParticipantsByQuiz()`

WebSocket Implementation

Server-side WebSockets (`websocket.ts`)

```
export function setupWebSockets(server: HttpServer): void {
  const wss = new WebSocketServer({ server, path: '/ws' });

  // Track client connections
  const clients: Map<string, ClientConnection> = new Map();

  wss.on('connection', (ws: WebSocket) => {
    const id = uuidv4();
    clients.set(id, { ws });

    ws.on('message', (message: WebSocket.Data) => {
      try {
        // Parse incoming message
        const data: WebSocketMessage = JSON.parse(message.toString());

        // Handle different message types
        switch (data.type) {
          case 'join_quiz':
            // Logic for joining a quiz
            break;
          case 'leave_quiz':
            // Logic for leaving a quiz
            break;
          case 'submit_answer':
            // Logic for submitting answers
            break;
          case 'quiz_state_update':
            // Logic for quiz state updates
            break;
          // ... other message types
        }
      } catch (error) {
        console.error('WebSocket message error:', error);
      }
    });

    // Handle disconnections
    ws.on('close', () => {
      clients.delete(id);
    });
  });
}
```

Client-side WebSocket Hook (`use-websocket.tsx`)

```
export function useWebSocket(
  url: string,
  {
    onOpen,
    onMessage,
    onClose,
    onError,
    reconnectInterval = 3000,
    reconnectAttempts = 5
  }: WebSocketOptions = {}
): UseWebSocketReturn {
  // Hook implementation to manage WebSocket connections
  // with reconnection logic, message handling, etc.
}
```

Authentication

The authentication system uses Passport.js with local strategy:

```
export function setupAuth(app: Express) {
  // Configure passport with local strategy
  passport.use(new LocalStrategy(async (username, password, done) => {
    try {
      const user = await storage.getUserByUsername(username);
      if (!user) {
        return done(null, false, { message: 'Incorrect username.' });
      }

      const isValid = await comparePasswords(password, user.password);
      if (!isValid) {
        return done(null, false, { message: 'Incorrect password.' });
      }

      return done(null, user);
    } catch (error) {
      return done(error);
    }
  }));

  // Serialization and deserialization logic for sessions
}
```

Key Algorithms

1. Score Calculation and Winner Determination

Score Calculation

The score calculation mechanism counts the number of correct answers for each participant, where "correct" is defined as selecting non-decoy options:

```
// Process each answer
participant.answers.forEach((answer, idx) => {
  const question = quiz.questions[idx];
  if (!question) return;

  // Convert answer string to array of numbers
  const selectedAnswers = typeof answer === 'string' && answer.includes(',')
    ? answer.split(',').map((a: string) => Number(a))
    : [Number(answer)];

  // Let's count correct answers based on non-decoy options
  let correctAnswersCount = 0;

  // Check if the question has isDecoy property
  if (question && question.isDecoy && Array.isArray(question.isDecoy)) {
    // Count each selected answer that is not a decoy as correct
    for (const selectedOption of selectedAnswers) {
      // If the selectedOption index is within bounds and not a decoy, it's
correct
      if (
        selectedOption >= 0 &&
        selectedOption < question.isDecoy.length &&
        !question.isDecoy[selectedOption]
      ) {
        correctAnswersCount++;
      }
    }
  }
  // Fallback if isDecoy is not available - just count option 0 as correct
  else if (selectedAnswers.includes(0)) {
    correctAnswersCount = 1;
  }

  // Add to total correct count
  correctCount += correctAnswersCount;
});
```

Color-Coding Answers

To provide visual feedback, answers are color-coded as correct (green) or incorrect (red):

```
// Determine if the selected option is correct (not a decoy)
let isCorrect = false;

// If the question has isDecoy array, check if this option is not a decoy
if (question.isDecoy && Array.isArray(question.isDecoy) &&
    optionIdx >= 0 && optionIdx < question.isDecoy.length) {
  // If isDecoy[optionIdx] is false, then this is a correct option
  isCorrect = !question.isDecoy[optionIdx];
}
// Fallback to treating option 0 as correct
else if (optionIdx === 0) {
  isCorrect = true;
}

return (
  <span key={i} className={`${isCorrect ? 'text-green-600' : 'text-red-600'}
${i > 0 ? 'ml-1' : ''}`}>
    {optionText}{i < selectedOptions.length - 1 ? ', ' : ''}
  </span>
);
```

Winner Ranking

The winner determination algorithm ranks participants based on:

1. Number of correct answers (primary sort)
2. Submission time (secondary sort, for tiebreakers)

```
.sort((a, b) => {
  // Sort by correct answers (descending)
  if (b.correctCount !== a.correctCount) {
    return b.correctCount - a.correctCount;
  }
  // If tied, sort by submission time (ascending)
  return new Date(a.submittedAt || Date.now()).getTime() -
         new Date(b.submittedAt || Date.now()).getTime();
})
```

2. Short Code Generation

Quiz short codes are generated with a simple but effective alphanumeric generator:

```
function generateShortCode(): string {
  const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
  const codeLength = 6;
  let result = '';

  for (let i = 0; i < codeLength; i++) {
    result += characters.charAt(Math.floor(Math.random() * characters.length));
  }

  return result;
}
```

3. Correct Answer Identification

For questions with multiple answers, the algorithm determines correctness:

```
// Check if the question has isDecoy array, check if this option is not a decoy
if (question.isDecoy && Array.isArray(question.isDecoy) &&
    optionIdx >= 0 && optionIdx < question.isDecoy.length) {
  // If isDecoy[optionIdx] is false, then this is a correct option
  isCorrect = !question.isDecoy[optionIdx];
}
```

Notable Patterns

1. React Query Implementation

The application uses TanStack Query for data fetching and caching:

```
const { isLoading, data: quiz, error } = useQuery({
  queryKey: ['/api/quizzes', quizId, 'results'],
  enabled: !!quizId
});
```

2. Real-time Communication Pattern

The app follows a publisher-subscriber pattern for real-time updates:

1. Host publishes quiz state changes through WebSockets
2. Players subscribe to quiz updates via WebSocket connections
3. Updates are broadcasted to all relevant connected clients

3. Authentication Context

The app uses React Context for global authentication state:

```
export const AuthContext = createContext<AuthContextType | null>(null);
export function AuthProvider({ children }: { children: ReactNode }) {
  // Implementation
}
export function useAuth() {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error('useAuth must be used within an AuthProvider');
  }
  return context;
}
```

4. Responsive Design Hooks

Custom hooks manage responsive design:

```
export function useIsMobile() {
  const [isMobile, setIsMobile] = useState(false);

  useEffect(() => {
    const checkIsMobile = () => {
      setIsMobile(window.innerWidth < 768);
    };

    checkIsMobile();
    window.addEventListener('resize', checkIsMobile);

    return () => {
      window.removeEventListener('resize', checkIsMobile);
    };
  }, []);

  return isMobile;
}
```

Future Enhancements

The application has several opportunities for future improvements:

1. Enhanced Quiz Creation

- Support for image and multimedia content in questions
- Rich text formatting for questions and answers
- Question bank with reusable questions across quizzes
- Quiz templates for common quiz formats

2. Expanded Game Modes

- Team-based competitions
- Tournament mode with multiple rounds
- Time-based scoring (faster answers earn more points)
- Progressive difficulty levels

3. Advanced Analytics

- Detailed performance metrics for hosts
- Visual charts showing question difficulty
- Participant engagement analytics
- Historical performance tracking

4. Social Features

- Public quiz directories
- Quiz sharing on social media
- User profiles with statistics
- Global leaderboards

5. Technical Improvements

- Enhanced WebSocket reliability with fallback mechanisms
- Offline mode with synchronized updates when reconnected
- Performance optimizations for large participant groups
- Mobile app versions using React Native