Design Doc: QuickQuiz
Status: Draft
Authors: Robel Kebede (robel.kebede@bison.howard.edu), Trishma Garcon (Trishma.Garcon@bison.howard.edu), Xavier Green (Xavier.Green@bison.howard.edu), Dalvin Ticha (dalvin.ticha@bison.howard.edu), Caleb Orr (Caleb.Orr@bison.howard.edu)

Background
QuickQuiz is a lightweight web app where users can create, and complete short trivia-style quizzes made up of multiple questions with defined correct answers. Authenticated users can create, view, and delete their own quizzes; all users can browse and play quizzes, submit answers, view their score, and reveal correct answers.

This design is based on the QuickQuiz 1-Pager, PRD, and MVP UI defined in earlier parts of the project. It covers all non-stretch requirements: authentication, quiz CRUD, quiz play, scoring, and answer reveal.

Overview (High-Level Design)
QuickQuiz is a classic **React + API + DB** web application:

- **Client (React)**
  - Renders all UI pages (landing, dashboard, My Quizzes, Create Quiz, Play Quiz, Results).
  - Handles navigation, form validation, and calling backend APIs.
  - Integrates with **Firebase Authentication (Google Sign-In)** on the client.
- **Backend (FastAPI)**
  - Exposes RESTful JSON APIs for quizzes and attempts (create/list/delete/play).
  - Validates Firebase ID tokens for protected operations.
  - Implements answer checking and stores attempt summaries.
- **Database / Storage**
  - Stores users (minimal profile), quizzes, questions, and quiz attempts.
  - Simple relational-style schema (can be implemented in PostgreSQL, SQLite, or Firestore with equivalent collections).
- **GitHub**
  - Used for collaboration, version control, code review.

This architecture keeps the system small, understandable, and easy to implement within one week while matching course expectations.

Detailed Design

1. **Architecture & Major Components**

1.1 Client (React)

Main routes/pages:

a. **/ – Landing / Public Quizzes**
   - Shows brief description.

    - Lists public quizzes with “Play” actions.
    - Shows “Log In with Google” if logged out.
  b. **/dashboard (or same as / when logged in)**
    - Header with navigation: Public Quizzes, My Quizzes, Create Quiz, Logout.
    - Reuses public quizzes list.
  c. **/my-quizzes**
    - Lists quizzes created by the logged-in user.
    - Buttons: Play, Delete.
  d. **/create-quiz**
    - Form for title, optional description, 1–10 Q&A pairs.
    - Client-side validation.
    - On save → POST to backend → redirect to My Quizzes.
  e. **/quiz/:id (Play Quiz)**
    - Fetch quiz by ID.
    - Inputs for answers.
    - On submit → POST answers to backend → navigate to Results.
  f. **/quiz/:id/results**
    - Shows score + per-question correctness.
    - “Reveal Answer” button per question (handled client-side with server-provided correct answers).

Key client responsibilities

- Integrate Firebase Auth:
  - Use Firebase JS SDK to sign in with Google.
  - Store ID token and send it as Authorization: Bearer <token> to FastAPI for protected endpoints.
- Basic validation before API calls (no empty title, no empty question/answer).
- Display loading and error states.

1.2 Backend (FastAPI)

The backend is stateless, exposing a minimal set of REST endpoints.

Auth Integration

- For protected routes (create, delete, etc.), FastAPI:
  - Extracts Firebase ID token from Authorization header.
  - Verifies token via Firebase Admin SDK or REST.
  - Resolves user_id (UID or email) to associate with created quizzes.
- Public read operations (list/play quizzes) do **not** require auth.

Core API Endpoints (MVP)

(Names are illustrative; final naming just needs to be consistent.)

- GET /api/quizzes
  - Returns list of public quizzes: id, title, description, creatorDisplayName(optional).
- GET /api/quizzes/my (auth)
  - Returns quizzes where creator_id == current_user_id.

- POST /api/quizzes (auth)
    - Body: title, description, questions: [{text, correct_answer}].
    - Validates required fields & max 10 questions.
    - Saves quiz, returns new id.
- DELETE /api/quizzes/{quiz_id} (auth)
    - Only allowed if creator_id == current_user_id.
    - Deletes quiz and its questions.
- GET /api/quizzes/{quiz_id}
    - Returns quiz with questions (for play view).
- POST /api/quizzes/{quiz_id}/submit
    - Body: { answers: ["user answer 1", "user answer 2", ...] }
    - Backend:
        - Loads quiz questions.
        - Compares each answer to correct answer:
            - Case-insensitive.
            - Trim spaces.
        - Returns:

```
{
  "score": X,
  "total": N,
  "results": [
    {
      "question": "...",
      "user_answer": "...",
      "correct_answer": "...",
      "is_correct": true/false
    },
    ...
  ]
}
```

        - If user is logged in, optionally records attempt summary in DB.

These endpoints fully cover the PRD MVP use cases.

1.3 Data Model

Implementation can be relational or document-based; below is a relational-style schema (easy to map to any store).

User
- id (string) – Firebase UID or email
- display_name (string)
- (No passwords stored; Firebase handles auth.)

Quiz
- id (string or int)

- title (string, required)
- description (string, optional)
- creator_id (fk → User.id)
- created_at (timestamp)
- is_public (bool, default true)

Question
- id
- quiz_id (fk → Quiz.id)
- order (int)
- text (string)
- correct_answer (string)

Attempt (optional but simple)
- id
- quiz_id (fk → Quiz.id)
- user_id (nullable fk → User.id; null for anonymous)
- score (int)
- total (int)
- created_at (timestamp)

This schema:
- Supports "My Quizzes" via creator_id.
- Supports public listing.
- Supports answer evaluation and simple analytics later.

2. Internal Flows (How Components Work Together)

2.1 Create Quiz Flow

a. User logs in with Google in the client, client stores Firebase token.
b. User fills out Create Quiz form.
c. Client validates required fields.
d. Client sends POST /api/quizzes with Firebase token.
e. FastAPI verifies token, extracts user_id, creates Quiz + Questions.
f. On success, client redirects to My Quizzes (showing new quiz).

2.2 Play Quiz & Submit Answers

a. From Public Quizzes or My Quizzes, user clicks Play.
b. Client calls GET /api/quizzes/{id} and renders questions.
c. User enters answers, clicks Submit Quiz.
d. Client sends POST /api/quizzes/{id}/submit with answers.
e. FastAPI:
    - Loads Questions.
    - Normalizes and compares answers.
    - Computes score + result list.
    - (Optional) Saves attempt if user logged in.

f. Client shows Results page and allows per-question “Reveal Answer” using returned data.

2.3 Delete Quiz

a. On My Quizzes, user clicks Delete.
b. Client shows confirmation.
c. If confirmed, sends DELETE /api/quizzes/{id} with token.
d. FastAPI verifies owner and deletes quiz + questions.
e. Client refreshes list.

3. Design Decisions, Open Questions, and How We’ll Handle Them

Decision: FastAPI instead of Django

- Chosen for simplicity, speed, and alignment with small, JSON-based APIs.
- Impact is localized to backend implementation; frontend + requirements unchanged.
- If instructor feedback suggests Django is required, we can swap backend with minimal changes at the API boundary.

Decision: Firebase Auth + Backend Token Verification

- Provides secure Google login without managing passwords.
- Backend trusts only verified tokens.

Open Question 1: Exact Database Choice

- Options: SQLite/PostgreSQL (via SQLAlchemy) or Firestore.
- Plan:
  - Start with simplest option available in project environment (e.g., SQLite + SQLAlchemy).
  - Abstract DB access via a small repository layer so switching to a cloud DB later is low impact.
- This uncertainty is documented here; whichever is chosen will be finalized and noted in the doc.

Open Question 2: Storing Attempts

- MVP does not require rich analytics.
- Plan:
  - Implement minimal Attempt table now (cheap).
  - If time is tight, we keep writes but do not yet build any UI over historical attempts.

Open Question 3: Anonymous Play

- MVP design allows anonymous quiz play.
- Plan:
  - Keep GET and submit endpoints open for play.
  - Only require auth for create/delete.

All major behavior for non-stretch MVP is decided; remaining questions have clear, documented paths.

4. **Coverage of Non-Stretch Goals (Checklist)**

This design supports all MVP items from the PRD:

- **Login with Google** → Firebase Auth + client + FastAPI verification.
- **Create challenge with multiple questions** → POST /api/quizzes + Questions table.
- **See challenges I created** → /my-quizzes + GET /api/quizzes/my.
- **Delete my challenges** → DELETE /api/quizzes/{id} with ownership check.
- **Complete other users' challenges** → GET /api/quizzes, GET /api/quizzes/{id}, submit flow.
- **Reveal answers** → Results endpoint returns correct answers; UI reveal per question.
- **Basic testing support** → Clear API and data model for unit tests on FastAPI routes and simple front-end tests.

Stretch ideas (leaderboard, random quiz, etc.) are intentionally **not** designed in detail here.

Security and Privacy

- **Authentication**
  - All create/delete operations require a valid Firebase ID token.
  - Backend verifies token on each protected request.
- **Authorization**
  - creator_id checked before deleting a quiz.
  - Only quiz owners can manage their quizzes.
- **Data Minimization**
  - Store minimal user information (ID + display name/email if needed).
  - No sensitive personal data required.
- **Transport**
  - Intended deployment over HTTPS (if/when hosted), preventing token leakage.

Given the limited scope and course context, this model is sufficient and straightforward to implement.

Accessibility

- Use semantic HTML structure in React components.
- Ensure:
  - Labels for all inputs.
  - Clear focus states, keyboard navigation.
  - Sufficient text contrast (even in simple, no-color MVP).
- Buttons and links use descriptive text (e.g., "Submit Quiz", "Back to Quizzes").

These steps align with basic accessibility expectations without major additional complexity.

Future Goals (Not in Current Design Scope)
Potential enhancements (not required for this assignment, but influenced some choices):

- **Leaderboards** using Attempt data.
- **Random Quiz** selection.
- **More flexible answer checking** (e.g., synonyms, partial credit).
- **Tags / categories** for quizzes.
- **Internationalization** and richer accessibility refinements.

The current design (clean API boundaries, simple schema) avoids decisions that would block these improvements later.

Traceability Matrix
The traceability matrix is maintained in a shared Google Sheet and is used as the primary artifact to show that all non-stretch requirements are covered by at least one executed test case.