Executive summary

- A two-level quiz web app with login/register (local and Google OAuth), timed multiple-choice questions, lives, and an intermission token to unlock Level 2.

- Backend: Node.js/Express with MongoDB via Mongoose, Passport for auth, Nodemailer for OTP, JWT for login cookies (not enforced on game endpoints).

- Frontend: React CRA app with video-driven UI, responsive ratio-adaptive assets, simple fetch-based API integration.

Goals and user journey

- Player opens app → Intro video → Auth (local or Google) → Story → Enter email (registers game session) → Level 1 quiz → Intermission: fetch short-lived token → Verify token → Level 2 quiz → End (win/lose).

- Selection caps: first 30 users can enter Level 2; first 3 successful completions are winners.

Architecture

- Monorepo-style structure with two subprojects:

- Backend: Squid\_B-main/Squid\_B-main

- Frontend: Squid\_F-main/Squid\_F-main

- Communication via REST endpoints prefixed with /user. Frontend uses absolute hosted backend URL.

Backend

- Stack: Express, Mongoose, Passport (Local, Google), JWT, Nodemailer, dotenv, cookie-parser, cors.

- Entrypoints

- server.js: builds app, sets middleware, mounts /user routes.

- index.js: starts HTTP server, connects to MongoDB via connectDb.

- Configuration

- config/mongo.config.js: connects using process.env.MONGO\_URL.

- config/passport.config.js:

- LocalStrategy validates hashed passwords against Logreg users.

- GoogleStrategy: OAuth 2.0 with clientID/clientSecret/callbackURL read from env; uses findOrCreate.

- config/nodemailer.config.js: Gmail transporter using SENDER\_EMAIL/SENDER\_PASSWORD; sends OTP emails for password reset.

- Data models

- Logreg user (src/model/LoginSchema.js)

- Fields: name, email (unique + format validation), password (strength validation), resetPasswordToken, resetPasswordExpire.

- Hooks/methods: pre-save bcrypt hash; verifyPassword; getResetPasswordToken() (6-digit numeric, 10 min).

- Squidquiz game (src/model/Squid.js)

- Fields: email, selected (1=started, 2=selected for level 2, 3=winner), win date, token (6-digit), starttime, question (last question number reached).

- Repo helpers (src/model/user.repo.js)

- createUserRepo, getUserRepo, findOrCreate, clearTokenDetails (clears OTP after TTL).

- Middleware

- src/middleware/confirm.js: ensures password and confirmPassword match on register.

- Routing (src/routes/user\_routes.js) mounted at /user

- OAuth and local auth

- GET /auth/google and GET /auth/google/callback → sets sid cookie with jwt.sign({email}, "privacykey"); redirects /user/responseGoogle.

- GET /responseGoogle and GET /failure return simple JSON.

- POST /login (LocalStrategy, no session) → sets sid cookie with same secret.

- Registration and password reset

- POST /register with confirmPasswords → create local user if not exists.

- POST /reset-password → issues OTP via email and sets expiry.

- POST /verifyOtp → validates token and expiry.

- POST /set-password → persist new password after confirming match.

- POST /resendOtp → re-issues OTP, resets expiry.

- Game flow

- POST /Start\_Game → first-time participation creates Squidquiz record (selected:1). Duplicate participation disallowed.

- POST /select\_cand → first 30 users get selected:2.

- POST /get\_token → sets and returns a 6-digit token in DB (for intermission).

- POST /verify\_token → validates { email, token }.

- POST /End\_Game → first 3 winners set selected:3 and win date; records question reached; others get fail message.

- POST /End → non-winner end; records win date and question.

- Security notes

- JWT secret is hardcoded ("privacykey") and should be env-driven.

- Game endpoints trust email in request body; no auth required; sid cookie is not enforced for authorization.

- Intermission token is returned via API in plaintext (intentional for gameplay, but visible to clients).

Frontend

- Stack: React 18 CRA, react-toastify, disable-devtool, Tailwind (as devDependency), simple fetch API calls.

- App shell: src/App.jsx

- Controls screen flow using state flags: intro, loginRegister, story, username presence, stop, LevelUp, Lives, timers.

- Prevents context menu, devtools keys, and various reload actions.

- Chooses aspect-ratio-appropriate video assets (16:9 vs 4:3).

- Key components

- LoginRegister.jsx: orchestrates login/register/password reset steps; video background per ratio.

- Login.jsx: POST /user/login, Google button to /user/auth/google and confirms via /user/responseGoogle.

- Register.jsx: POST /user/register.

- ResetPassword.jsx, GetOTP.jsx, SetNewPassword.jsx: hit /reset-password, /verifyOtp, /set-password.

- Start.jsx and story.jsx: intro/story videos.

- Begin.jsx: email capture; POST /user/Start\_Game; requests fullscreen on success.

- Sques.jsx: quiz mechanics

- Loads questions from src/quizData.js (data for Level 1, data1 for Level 2).

- Tracks answers, animations, disables input briefly to show correctness state.

- Finish Level 1 → POST /user/select\_cand (first 30) to unlock intermission; else fail message.

- Finish Level 2 → POST /user/End\_Game (first 3 winners).

- Wrong answers decrement Lives; at 0 lives → POST /user/End, exit fullscreen.

- Interval.jsx: intermission/token step

- Timer counts down; fetches token via POST /user/get\_token, displays it briefly, then prompts for token input.

- Valid token via POST /user/verify\_token starts Level 2; timeouts follow a penalty path that still advances (as coded).

- Timer.jsx: per-question countdown (Level 1 ~35s, Level 2 60s). On timeout with 0 lives, ends game via /user/End.

- UX

- Video-forward experience with overlayed quiz UI.

- Toast notifications for auth feedback; alert() for some game feedback.

Data model details

- Logreg (Auth users)

- Fields: name, email (unique), password (hashed), resetPasswordToken (number), resetPasswordExpire (Date).

- Password policy: at least 8 chars, includes uppercase, lowercase, digit, special char.

- OTP: 6-digit numeric, expires after 10 minutes.

- Squidquiz (Game state)

- Tracks participation and progression via selected:

- 1 = started Level 1

- 2 = eligible for Level 2 (top 30)

- 3 = winner (top 3)

- Stores win (timestamp), starttime, question (progress), token (6-digit for intermission).

API endpoints (purpose-first)

- Auth and session

- POST /user/register → register user (validations + confirm password).

- POST /user/login → local login; sets sid cookie with JWT.

- GET /user/auth/google → redirect to Google OAuth.

- GET /user/auth/google/callback → on success sets sid cookie; redirects to /user/responseGoogle.

- GET /user/responseGoogle → post-login JSON.

- GET /user/failure → login failure JSON.

- Password reset

- POST /user/reset-password → email OTP.

- POST /user/verifyOtp → verify OTP.

- POST /user/set-password → save new password.

- POST /user/resendOtp → new OTP.

- Gameplay

- POST /user/Start\_Game → register participation for an email.

- POST /user/select\_cand → attempt to mark selected:2 if under 30.

- POST /user/get\_token → generate and return 6-digit token; store in DB.

- POST /user/verify\_token → verify {email, token} for Level 2.

- POST /user/End\_Game → attempt to mark selected:3 if under 3 and record question.

- POST /user/End → end game; record question.

Environment variables

- Backend

- MONGO\_URL: MongoDB connection string.

- GOOGLE\_CLIENT\_ID, GOOGLE\_CLIENT\_SECRET: OAuth credentials.

- BASE\_URL: public backend base for OAuth callback.

- SENDER\_EMAIL, SENDER\_PASSWORD: Gmail SMTP for Nodemailer.

- Recommended: JWT\_SECRET (currently hardcoded to "privacykey").

- Frontend

- None required, but backend URL is hardcoded; consider moving to .env.

Local development

- Backend

- Set .env with the env vars above.

- From Squid\_B-main/Squid\_B-main:

- Install: npm install

- Run: npm run start (uses index.js; server.js attaches routes)

- Frontend

- From Squid\_F-main/Squid\_F-main:

- Install: npm install

- Run: npm start

- For local testing, change fetch URLs in components from the hosted https://squid-b.onrender.com to your local backend (e.g., http://localhost:3000). Alternatively, add a proxy in package.json.

Operational considerations

- Deployment

- Backend hosted on Render in code; ensure BASE\_URL matches deployed domain.

- CORS is enabled broadly; cookies are set, but not used for guarding gameplay endpoints.

- Observability

- Console logging only; no structured logging, tracing, or metrics.

- Error handling

- Many endpoints return generic messages; some lack consistent error status codes (mostly JSON messages with status flags).

- No global error handler middleware.

Security and privacy

- Auth gaps

- Game endpoints accept email in the body and don’t require an authenticated session; the JWT cookie is not enforced. Anyone could submit progress for any email.

- Secrets

- JWT secret is hardcoded. Move to env and rotate.

- Token mechanics

- Intermission token is intentionally leaked to the client/UI. Given game design, this may be acceptable, but it’s trivially capturable.

- Rate limiting / abuse

- No rate limiting or captcha on register/login/OTP endpoints.

- Input validation

- Server-side input validation is minimal outside of Mongoose schema validators.

Testing

- No unit/integration tests present.

- No linter scripts for backend; frontend uses CRA defaults.

Performance

- Typical small scale OK. Mongo queries are simple. Under load, consider:

- Indices on email, selected.

- Caching selection counts (currently counted each request).

- Avoid returning large assets via backend (frontend serves videos locally).

Risks and limitations

- Cheating potential due to unguarded game endpoints.

- Email is the only game identity; no tie to auth session for gameplay.

- Hardcoded backend URL in frontend complicates environment switching.

- Empty checkaccess.js suggests intended access checks are missing.

Recommendations / roadmap

- Enforce authentication on gameplay endpoints; derive email from verified JWT/cookie.

- Move JWT secret to env; add expiry and refresh if needed.

- Introduce rate limiting and captcha on sensitive endpoints.

- Centralize backend URL in frontend via env (REACT\_APP\_API\_BASE), use a proxy in dev.

- Add validation middleware (e.g., zod/joi) and consistent status codes.

- Add structured logging and error handler middleware.

- Add tests (auth flow, selection caps, token flow).

- Consider server-side generation/display of token via secure channel or obfuscation if gameplay allows.

- Add admin dashboard to monitor selected counts, winners, and audit logs.