Technical Document: NWU Live Poll

Table of contents

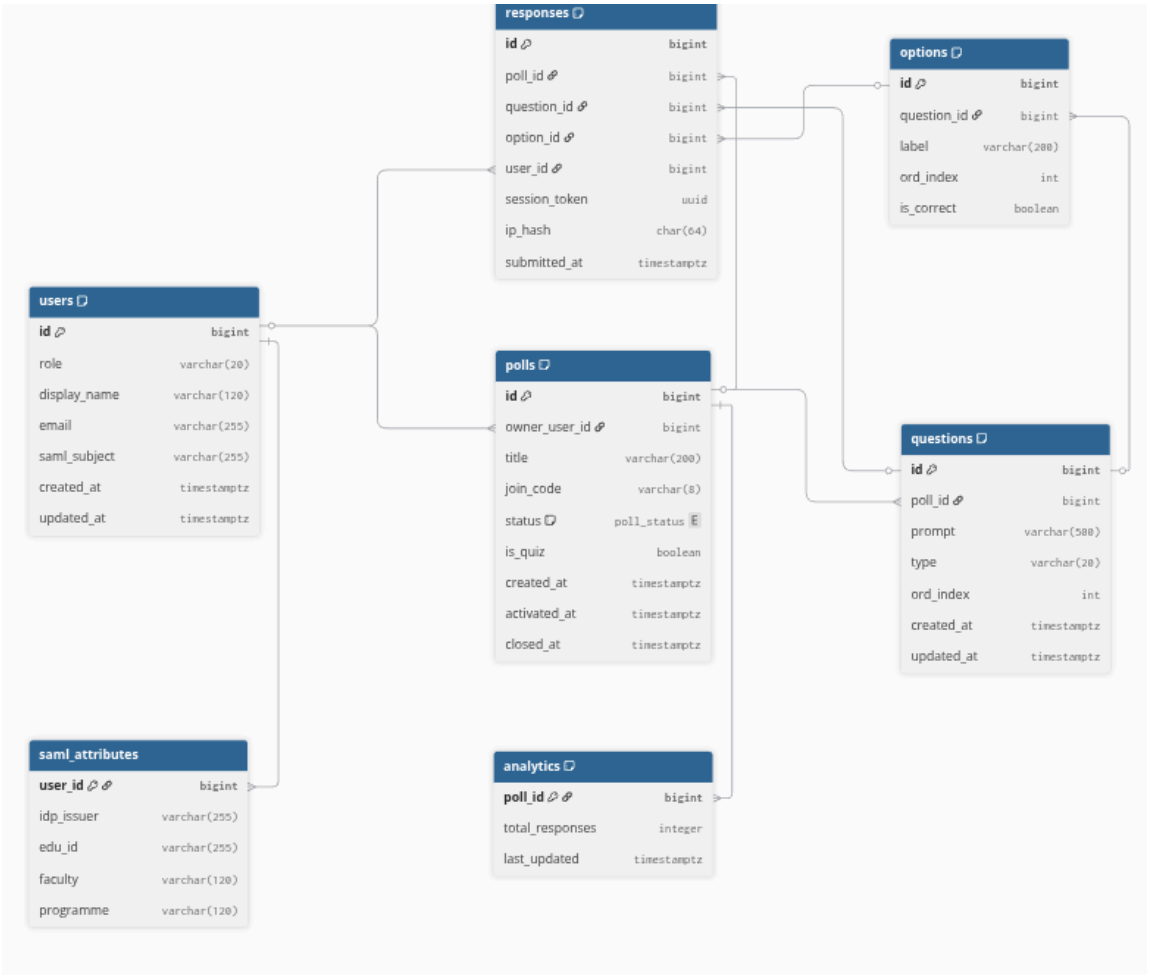
Functionality

Overview

NWU Live Poll is a real-time classroom polling application that allows lecturers to create live polls and students to respond instantly using their devices. The system updates poll results dynamically, enabling interactive learning sessions.

Implemented features

- **Poll Creation and Management:** Lecturers can create polls with customizable questions and multiple-choice options.
- **Session Codes:** Each poll session generates a unique join code for students.
- **Real-Time Interaction:** Responses and results are updated instantly using **Socket.IO**.
- **User Roles:** Two primary roles – Lecturer and Student.
- **Guest and Authenticated Access:** Students can join anonymously, while lecturers authenticate securely.
- **Analytics Dashboard:** Provides visual summaries of participation, responses, and trends.
- **Data Retention:** Responses are securely stored for analysis and reporting.



Process:

Lecturer creates → system generates code → student joins via browser → votes transmitted via WebSockets → results broadcast to dashboard → stored in database.

Code Quality

- **Modular Design:** Separated into services (PollService, UserService, AuthService).
- **Type Safety:** Full TypeScript support on both client and server.
- **Linting:** ESLint + Prettier used for code consistency.
- **Error Handling:** Centralized middleware for error reporting and exception tracking
- **Version Control:** GitHub repository structured by features; code reviewed via pull requests.
- **Documentation:** JSDoc comments and markdown documentation for all major modules.

Examples:

analyticsService.ts snippet

```
 async getPollStats(pollId: number) {
    const poll = await prisma.poll.findUnique({
      where: { id: pollId },
      include: {
        questions: { include: { options: true } },
        lobby: { include: { user: { select: { studentNumber: true } } } }
},
      },
    });
    if (!poll) throw new Error("Poll not found");
```

auth.ts

```
import { Request, Response, NextFunction } from "express";
import jwt from "jsonwebtoken";
import { JWTPayload, AuthUser } from "../types/auth";
import { prisma } from "../config/database";

declare global {
  namespace Express {
    interface Request {
      user?: AuthUser;
    }
  }
}

export const authenticateToken = async (req: Request, res: Response,
next: NextFunction) => {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1];

  if (!token) {
    return res.status(401).json({ success: false, error: "Access token
required" });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET!) as
JWTPayload;
```

```
    const user = await prisma.user.findUnique({
      where: { id: decoded.userId },
    });

    if (!user) {
      return res.status(401).json({ success: false, error: "Invalid
token" });
    }

    req.user = {
      id: user.id,
      name: user.name,
      email: user.email,
      role: user.role as "student" | "lecturer",
      studentNumber: user.studentNumber || undefined,
    };

    next();
  } catch {
    return res.status(403).json({ success: false, error: "Invalid
token" });
  }
};

export const requireRole = (roles: string[]) => {
  return (req: Request, res: Response, next: NextFunction) => {
    if (!req.user || !roles.includes(req.user.role)) {
      return res.status(403).json({ success: false, error:
"Insufficient permissions" });
    }
    next();
  };
};
```

participation.ts snippet

```
router.get("/participation", (_req, res) => {
  res.json({
    success: true,
    message: "Poll Participation API",
```

```
    endpoints: {
      getByCode: "GET /api/polls/code/:joinCode",
      join: "POST /api/polls/join",
      recordChoice: "POST /api/polls/:id/choices (student auth)",
      submit: "POST /api/polls/:id/submit (student auth)",
    },
  });
```

JoinPage.tsx snippet

```
export default function JoinPage() {
  const navigate = useNavigate();

  // join form
  const [joinCode, setJoinCode] = useState("");
  const [studentNumber, setStudentNumber] = useState("");
  const [securityCode, setSecurityCode] = useState("");

  // poll + answers
  const [poll, setPoll] = useState<StudentPoll | null>(null);
  const [answers, setAnswers] = useState<number[]>([]); // -1 =
unanswered
  const answersRef = useRef<number[]>([]);              // <-- source
of truth when submitting
  const [result, setResult] = useState<SubmitResult | null>(null);

  // ui
  const [error, setError] = useState<string | null>(null);
  const [loading, setLoading] = useState(false);
  const [waiting, setWaiting] = useState(false);

  // timer
  const [deadline, setDeadline] = useState<number | null>(null);
  const [tick, setTick] = useState(0);
  const remaining = useMemo(() => {
    if (deadline === null) return 0;
    return Math.max(0, Math.floor((deadline - Date.now()) / 1000));
  }, [deadline, tick]);

  const autoSubmittedRef = useRef(false);

  // Assert student role on mount and prefill number if we have it
```

```
  useEffect(() => {
    setRole("student");
    const remembered = loadStudentNumber();
    if (remembered && !studentNumber) setStudentNumber(remembered);
  }, []);

  useEffect(() => {
    const t = setInterval(() => setTick((n) => n + 1), 1000);
    return () => clearInterval(t);
  }, []);

  useEffect(() => {
    if (!poll) return;
    setDeadline(Date.now() + poll.timerSeconds * 1000);
  }, [poll]);

  async function handleJoin(e: React.FormEvent) {
    e.preventDefault();
    setError(null);
    setLoading(true);
    setResult(null);
    autoSubmittedRef.current = false;
    setDeadline(null);

    try {
      const meta = await getPollByCode(joinCode.trim());

      await studentJoin({
        joinCode: joinCode.trim(),
        studentNumber: studentNumber.trim(),
        securityCode: securityCode.trim() || undefined,
      });

      // remember student number locally so Student page can show
history
      saveStudentNumber(studentNumber.trim());
      setRole("student");

      const initAnswers = (qCount: number) => {
        const init = Array(qCount).fill(-1) as number[];
        setAnswers(init);
        answersRef.current = init;
      };
```

```
      if (meta.status === "open") {
        setWaiting(true);
        const loop = async () => {
          try {
            const m = await getPollByCode(joinCode.trim());
            if (m.status === "live") {
              setWaiting(false);
              setPoll(m);
              initAnswers(m.questions.length);
            } else {
              setTimeout(loop, 2000);
            }
          } catch {
            setTimeout(loop, 2000);
          }
        };
        loop();
      } else if (meta.status === "live") {
        setPoll(meta);
        initAnswers(meta.questions.length);
      } else if (meta.status === "closed") {
        setError("This poll is closed and no longer accepting
answers.");
      } else {
        setError("This poll is not accepting answers yet.");
      }
    } catch (err: any) {
      setError(err.message || "Failed to join");
    } finally {
      setLoading(false);
    }
  }
```

Performance

Optimization Measures

- **Containerization:** Both front-end and back-end are Dockerized to ensure consistent environments and lightweight deployment.
- **Load Efficiency:** Socket.IO namespaces isolate each poll session, minimizing unnecessary data transmission.
- **Database Optimization:** Indexed queries in PostgreSQL to handle large concurrent sessions efficiently.
- **Front-End Optimization:** Code-splitting and lazy loading implemented to reduce initial load times.
- **Caching:** Frequently accessed poll data cached in-memory using Redis.
- **Compression and Minification:** GZIP compression for API responses and static assets.

Testing

- Conducted stress tests???
- Latency ????
- Continuous monitoring and debugging using Docker logs and Node.js profiling tools

Deployment

Process

The final deployment setup replaced Azure with **Docker** and **Vercel** for improved cost efficiency, scalability, and CI/CD simplicity.

Component	Platform	Purpose
Front-End	Vercel	Continuous deployment and CDN hosting for React app
Back-End	Docker Container (Node.js)	Hosted on a containerized environment
Database	PostgreSQL	Managed cloud instance for persistent storage
Container Orchestration	Docker Compose	To run front-end, back-end, and database locally in development
Real-Time Engine	Socket.IO Server	Managed within Dockerized back-end container

Deployment Pipeline

1. Push to dev branch - GitHub triggers build via CI/CD
2. Docker Build - Containers built using the Dockerfile and docker-compose.yml
3. Front-End Deployment - Automatically deployed to Vercel (connected to the GitHub Repository)
4. Back-End Deployment - Docker image deployed to Render or Railway
5. Environment Variables - Managed securely in .env and Vercel/Render environment dashboards

Example of our Docker Setup

```
volumes:
  dbdata:
  api_node_modules:
  web_node_modules:

services:
  db:
    image: postgres:16-alpine
    container_name: nwu-db
    environment:
```

```
      POSTGRES_USER: ${POSTGRES_USER:-postgres}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-postgres}
      POSTGRES_DB: ${POSTGRES_DB:-nwupoll}
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
${POSTGRES_DB}"]
      interval: 5s
      timeout: 3s
      retries: 20

  redis:
    image: redis:7-alpine
    container_name: nwu-redis
    ports:
      - "6379:6379"
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 20

  api:
    image: node:20-bullseye-slim
    container_name: nwu-api
    user: root
    working_dir: /app
    volumes:
      - ./apps/api:/app
      - api_node_modules:/app/node_modules
    command:
      - sh
      - -lc
      - |
        set -eu
        apt-get update -y
        apt-get install -y --no-install-recommends openssl libssl1.1
        node -v
        npm -v
        echo "ENV → PORT=$PORT  DATABASE_URL=$DATABASE_URL"
```

```
        npm install
        npx prisma generate
        npm run build
        npx prisma migrate dev --name=auto || true
        ls -la dist
        echo "Starting server with: node dist/server.js"
        node dist/server.js
    environment:
      - PORT=8080
      -
DATABASE_URL=postgresql://${POSTGRES_USER:-postgres}:${POSTGRES_PASSWOR
D:-postgres}@db:5432/${POSTGRES_DB:-nwupoll}
      -
JWT_SECRET=${JWT_SECRET:-your_super_secret_jwt_key_here_make_it_long_an
d_random_12345678}
      - FRONTEND_URL=http://localhost:5173
      - REDIS_URL=redis://redis:6379
    ports:
      - "8080:8080"
    depends_on:
      db:
        condition: service_healthy
      redis:
        condition: service_healthy

  web:
    image: node:20-alpine
    container_name: nwu-web
    working_dir: /app
    volumes:
      - ./apps/web:/app
      - web_node_modules:/app/node_modules
    command: sh -c "npm ci && npm run dev"
    environment:
      - VITE_API_BASE=http://localhost:8080/api
    ports:
      - "${WEB_PORT:-5173}:5173"
    depends_on:
      - api
```

Full deployment documentation can be found at
https://github.com/SlothCartel/NWU-Classroom-Polling-Group-1/docs/deployment-docs.md

Front-End

Overview

The NWU Live Poll frontend is a React-based single-page application that provides an intuitive interface for classroom polling. Built with TypeScript and Vite, the application offers separate workflows for lecturers and students with real-time updates and responsive design.

Technology Stack:

- React 18 with TypeScript
- Vite for build tooling
- React Router for navigation
- Tailwind CSS for styling
- Socket.io client for WebSocket communication
- Fetch API for HTTP requests

Key Features:

- Role-based routing with authentication guards
- Real-time poll participation and live charts
- Responsive design for mobile and desktop
- Automatic quiz grading and feedback
- Submission history for students
- Poll analytics and CSV export for lecturers

Architecture

Project Structure:

```
apps/web/
├── src/
│   ├── App.tsx              # Main routing configuration
│   ├── main.tsx             # Application entry point
│   ├── components/
│   │   ├── guards.tsx       # Route protection components
│   │   └── Navbar.tsx       # Navigation component
│   ├── lib/
│   │   ├── api.ts           # API service layer
│   │   ├── auth.ts          # Authentication utilities
│   │   ├── http.ts          # HTTP client wrapper
│   │   ├── socket.ts        # WebSocket client
│   │   ├── studentAuth.ts   # Student-specific auth
```

```
│   │   └── types.ts          # TypeScript definitions
│   ├── pages/
│   │   ├── LoginPage.tsx     # Landing page with role selection
│   │   ├── LecturerLoginPage.tsx
│   │   ├── LecturerSignupPage.tsx
│   │   ├── StudentLoginPage.tsx
│   │   ├── StudentSignupPage.tsx
│   │   ├── AdminPage.tsx     # Lecturer dashboard
│   │   ├── DashboardPage.tsx  # Poll management
│   │   ├── StatsPage.tsx     # Poll analytics
│   │   ├── StudentPage.tsx   # Student dashboard
│   │   └── JoinPage.tsx      # Poll participation
│   └── assets/               # Static resources
└── public/                   # Public assets
```

Design Patterns:

Route Guards: Higher-order components that protect routes based on user role
Service Layer: Abstracted API calls in `lib/api.ts` for clean separation
Type Safety: Comprehensive TypeScript interfaces for all data structures
Error Handling: Centralized error handling in HTTP client with user feedback

Application Flow

Authentication:

1. User lands on `/login` (role selection)
2. Based on role selection, redirect to lecturer or student login
3. On successful authentication, JWT token stored in localStorage
4. Role-based redirect to appropriate dashboard
5. Protected routes validate token and role before rendering

Lecturer:

1. Login/Signup → Dashboard
2. Create new poll with questions and options
3. Open poll (generates join code, opens lobby)
4. Monitor lobby participants
5. Start poll when ready (activates questions)
6. View live answer statistics
7. Close poll and view final results
8. Export data as CSV for analysis

Student:

1. Login/Signup → Student Dashboard
2. Enter join code
3. Wait in lobby until lecturer starts
4. Answer questions within timer limits

5. Submit answers (automatic grading)
6. View score and feedback
7. Access submission history

Full frontend documentation can be found at
https://github.com/SlothCartel/NWU-Classroom-Polling-Group-1/docs/frontend-docs.md

Back-End

Overview

The NWU Live Poll API is a RESTful service built with Express.js and TypeScript that enables real-time classroom polling. The system supports lecturer-created polls with live student participation, automatic grading, and comprehensive analytics.

Base URL: `/api`
API Documentation: `/api-docs` (Swagger UI)

Key Features:

- Role-based authentication (lecturers and students)
- Real-time poll participation via WebSockets
- Live answer tracking and chart updates
- Automatic quiz grading with correct answer designation
- CSV/JSON data export for analysis
- Lobby management with participant tracking

Technology Stack:

- Node.js with Express.js
- PostgreSQL with Prisma ORM
- Socket.io for real-time communication
- JWT for authentication
- Swagger for API documentation

Architecture

Project Structure:

```
apps/api/
├── src/
│   ├── app.ts              # Express app configuration
│   ├── server.ts           # HTTP and WebSocket server initialization
│   ├── config/             # Configuration modules
```

```
│   ├── middleware/          # Authentication and validation
│   ├── routes/              # API route definitions
│   │   ├── auth/            # Authentication routes
│   │   ├── polls/           # Poll management routes
│   │   └── students/        # Student-specific routes
│   ├── services/            # Business logic layer
│   └── types/               # TypeScript type definitions
└── prisma/
    └── schema.prisma        # Database schema
```

Core Services:

authService: Handles JWT generation and validation
pollService: Manages poll lifecycle and operations
participationService: Processes student submissions
analyticsService: Generates statistics and exports
socketService: Manages real-time WebSocket connections

Authentication:

All protected routes require a JWT token in the Authorization header:
Authorization: Bearer <token>

Roles:

Lecturer: Can create, manage, and view poll results
Student: Can join polls and submit answers

Endpoints:

Lecturer Signup

```
POST /api/auth/lecturer/signup
Content-Type: application/json

{
            "name": "John Doe",
            "email": "john@example.com",
            "password": "securePassword123"
}
```

Lecturer Login

```
POST /api/auth/lecturer/login
Content-Type: application/json
```

```
{
            "email": "john@example.com",
            "password": "securePassword123"
}

Response: { "success": true, "data": { "user": {...}, "token":
"..." } }
```

Student Signup

```
POST /api/auth/student/signup
Content-Type: application/json

{
        "name": "Jane Smith",
        "studentNumber": "12345678",
        "email": "jane@example.com",
        "password": "securePassword123"
}
```

Student Login

```
POST /api/auth/student/login
Content-Type: application/json

{
        "email": "jane@example.com",
        "password": "securePassword123"
}
```

API Endpoints

- GET /api/polls #List All Polls
- GET /api/polls/:id #Get Poll by ID
- POST /api/polls #Create Poll
- DELETE /api/polls/:id #Delete Poll
- POST /api/polls/:id/open #Open Poll (Lobby)
- POST /api/polls/:id/start #Start Poll
- POST /api/polls/:id/close #Close Poll
- GET /api/polls/code/:joinCode #Get Poll by Join Code
- POST /api/polls/join #Join Poll

- POST /api/polls/:id/choices #Record Live Answer Choice
- POST /api/polls/:id/submit #Submit Final Answers
- GET /api/polls/:id/lobby #List Lobby Participants
- DELETE /api/polls/:id/lobby/:studentNumber #Remove Student from Lobby
- GET /api/polls/:id/stats #Get Poll Statistics
- GET /api/polls/:id/export?format=csv #Export Results
- GET /api/students/:studentNumber/submissions #List Student Submissions
- DELETE /api/students/:studentNumber/submissions/:pollId #Delete Student Submission

Database Schema

Core Tables:

User: Stores both lecturers and students (role-based)
Poll: Poll metadata with join code and timer settings
Question: Individual questions within a poll
Option: Answer choices for each question
Vote: Live answer selections (one per student per question)
Submission: Final submitted answers with scores
Answer: Individual answers within a submission
LobbyEntry: Tracks students who joined the lobby
Analytics: Aggregated participation data

Key Relationships:

User → Poll (one-to-many, lecturer creates polls)
Poll → Question (one-to-many)
Question → Option (one-to-many)
User → Vote (one-to-many, live selections)
User → Submission (one-to-many, final submissions)
Submission → Answer (one-to-many)

Unique Constraints:

Vote: One live vote per user per question
Submission: One submission per user per poll
LobbyEntry: One entry per user per poll

WebSocket Events

Connection:

```javascript
socket = io(API_URL, { auth: { token: JWT_TOKEN } });
```

Client to Server:

join-poll: Join a poll room

leave-poll: Leave a poll room
select-answer: Send live answer selection

Server to Client:

user-joined: New participant joined lobby
user-left: Participant left poll
answer-selected: Real-time answer selection broadcast
poll-status-changed: Poll lifecycle change (open/active/closed)
poll-stats-updated: Live statistics update
kicked-from-poll: Student removed from poll

Full backend/API documentation can be found at
https://github.com/SlothCartel/NWU-Classroom-Polling-Group-1/blob/main/docs/api-docs.md