ScoreStream Backend Documentation

1) Executive Summary

ScoreStream is a Node.js backend for a live sports score and commentary platform. It exposes a REST API for match and commentary CRUD-style operations, persists data in PostgreSQL through Drizzle ORM, and pushes real-time events over WebSockets. The backend is protected with Arcjet security policies (bot detection, shielding, and rate limiting) across both HTTP requests and WebSocket upgrades.

At a high level:

- **Express + HTTP server** powers REST endpoints.
- **WebSocket server (`ws`)** is attached to the same HTTP server for live fan-out.
- **PostgreSQL + Drizzle ORM** handles persistence and query composition.
- **Zod schemas** enforce request-level validation.
- **Arcjet** enforces request protection and abuse controls.

---

2) Architecture Overview

2.1 System Context

```
Clients (web/mobile/admin tools)
   ├─ HTTP/JSON ─────────────────▶ Express API
   │                               ├─ Validation (Zod)
   │                               ├─ Security middleware (Arcjet HTTP)
   │                               └─ Data access (Drizzle ORM)
   │                                           │
   │                                           ▼
   │                                    PostgreSQL
   │
   └─ WebSocket (/ws) ───────────▶ WS server
                                   ├─ Upgrade protection (Arcjet WS)
                                   ├─ Subscription management by matchId
                                   └─ Broadcast events (match created/commentary)
```

2.2 Runtime Composition

- A single Node process creates:
  - `express()` app
  - `http.createServer(app)`
  - WebSocket server attached via `attachWebSocketServer(server)`
- REST routes publish domain events by calling broadcast functions saved in `app.locals`.

This design keeps the API and real-time transport tightly integrated and avoids cross-process event buses for simple deployments.

---

3) Technology Stack

3.1 Core Platform

- **Node.js (ESM mode)**: package is configured with `"type": "module"`.
- **Express 5**: HTTP routing and middleware.
- **ws**: WebSocket transport and subscription channel implementation.

3.2 Data Layer

- **PostgreSQL (`pg`)**: primary relational database.
- **Drizzle ORM**: schema-first definitions + query builder for Node/Postgres.
- **Drizzle Kit**: migration generation, migration execution, and studio tooling.

3.3 Validation & Security

- **Zod**: strict input validation for params, query, and request bodies.
- **Arcjet**: abuse prevention for both HTTP and WebSocket upgrade paths.

3.4 Utility / Configuration

- **dotenv**: environment variable loading.

4) Project Structure and Responsibilities

```
src/
  index.js                  # App bootstrap + server wiring
  arcjet.js                 # Arcjet policy definitions + HTTP middleware
  db/
    db.js                   # Postgres pool + Drizzle instance
    schema.js               # Match/commentary schema + enum
  routes/
    matches.js              # /matches REST handlers
    commentary.js           # /matches/:id/commentary REST handlers
  validation/
    matches.js              # Zod schemas for match endpoints
    commentary.js           # Zod schemas for commentary endpoints
  utils/
    match.status.js         # Match lifecycle status derivation helpers
  ws/
    server.js               # WebSocket server + subscriptions + broadcasting

drizzle/
  *.sql                     # generated SQL migrations
```

5) Request Lifecycle and Flow

5.1 HTTP Path

1. Request enters Express app.
2. `express.json()` parses JSON payload.
3. Arcjet HTTP middleware runs (`securityMiddleware()`):
    - denylist/shield checks
    - bot handling
    - sliding window rate limiting
4. Route-level Zod validation executes.
5. Route handler performs DB operation through Drizzle.
6. Successful mutation may trigger real-time broadcast via `app.locals` callbacks.
7. JSON response returned to client.

5.2 WebSocket Path

1. Client upgrades to `/ws`.
2. Upgrade request passes Arcjet WS protection.
3. On connection:
    - client receives `{ type: "welcome" }`
    - heartbeat (`ping/pong`) liveness tracking starts
    - client may `subscribe` / `unsubscribe` by `matchId`
4. HTTP route mutations invoke `broadcastMatchCreated` / `broadcastCommentary`.
5. WS layer emits events to all clients or scoped match subscribers.

6) Data Model

6.1 `matches`

Represents a sporting fixture and score state.

- `id` (PK)
- `sport`
- `home_team`, `away_team`
- `status` enum: `scheduled | live | finished`
- `start_time`, `end_time`
- `home_score`, `away_score` (default `0`)

- `created_at`

6.2 `commentary`

Represents timeline events/messages attached to a match.

- `id` (PK)
- `match_id` (FK → `matches.id`)
- `minute`
- `sequence`
- `period`
- `event_type`
- `actor`
- `team`
- `message` (required)
- `metadata` (JSONB)
- `tags` (text[])
- `created_at`

6.3 Relational Notes

- One-to-many: `matches` → `commentary`.
- Foreign key integrity prevents commentary records from referencing missing matches.

---

7) API Surface

7.1 Health / Root

- `GET /`
  - Returns a simple greeting string.

7.2 Matches

- `GET /matches?limit=<n>`
  - Returns latest matches ordered by `createdAt DESC`.
  - Default limit `50`, hard cap `100`.
- `POST /matches`
  - Validates payload:
    - required: `sport`, `homeTeam`, `awayTeam`, `startTime`, `endTime`
    - optional: `homeScore`, `awayScore`
    - invariant: `endTime > startTime`
  - Computes status using current server time:
    - before start → `scheduled`
    - between start and end → `live`
    - at/after end → `finished`
  - Persists match and broadcasts `match_created` event to WS clients.

7.3 Commentary

Mounted as `app.use('/matches/:id/commentary', commentaryRouter)`.

- `GET /matches/:id/commentary?limit=<n>`
  - Validates path param `id` and optional limit.
  - Returns latest commentary entries for a match by `createdAt DESC`.
  - Default limit `10`, hard cap `100`.
- `POST /matches/:id/commentary`
  - Validates `id` plus commentary payload.
  - Inserts commentary row linked to `match_id`.
  - Broadcasts WS `commentary` event to subscribers of that match.

---

8) WebSocket Protocol

Endpoint: `ws://<host>:<port>/ws`

8.1 Server → Client messages

- `{"type":"welcome"}`
- `{"type":"match_created","data":<match>}`
- `{"type":"commentary","data":<commentary>}`
- `{"type":"subscribed","matchId":<number>}`
- `{"type":"unsubscribed","matchId":<number>}`
- `{"type":"error","message":"Invalid JSON"}`

8.2 Client → Server messages

- Subscribe:

  ```
  { "type": "subscribe", "matchId": 123 }
  ```

- Unsubscribe:

  ```
  { "type": "unsubscribe", "matchId": 123 }
  ```

8.3 Subscription Model

- In-memory map: `matchId -> Set<socket>` .
- Each socket stores `subscriptions` set for cleanup on disconnect.
- Broadcast modes:
  - **Global**: new match events to all connected clients.
  - **Scoped**: commentary events to clients subscribed to that match.

Implementation note: in the current `broadcastToMatch` implementation, the `readyState` condition appears inverted (`!== WebSocket.OPEN` before send), which can prevent delivery to active clients; this should be corrected for reliable commentary fan-out.

8.4 Liveness

- Heartbeat every 30 seconds.
- Socket terminated when no pong response is observed.

---

9) Validation Strategy

Validation is centralized in `src/validation/*` and applied before business logic. Benefits:

- Uniform bad-request response structure.
- Strong guard rails against malformed payloads.
- Coercion for query/path inputs (`z.coerce.number`) where appropriate.

Notable rules:

- Match IDs are positive integers.
- Match list/commentary list limits are positive and capped.
- Match `endTime` must be after `startTime` .
- Commentary `message` is required and non-empty.

---

10) Security and Abuse Prevention

Arcjet is configured separately for HTTP and WS:

- **Shield**: baseline request hardening.
- **Bot detection**: allows search-engine and preview bots, blocks undesired automation.
- **Rate limiting**:
  - HTTP: 50 requests / 10 seconds / IP.
  - WS upgrade: 5 attempts / 2 seconds / IP.

Failure handling:

- HTTP denied requests return `429` (rate-limited) or `403` .
- WS denied upgrades return status line and close socket.
- Arcjet errors return `503` (HTTP) or `500` (WS upgrade).

---

11) Configuration and Environment

Required environment variables:

- `DATABASE_URL` — PostgreSQL connection string.
- `ARCJET_KEY` — Arcjet API key (required by current implementation).

Optional:

- `ARCJET_MODE` — set to `DRY_RUN` for non-blocking policy evaluation; defaults to `LIVE`.
- `PORT` — defaults to `8000`.
- `HOST` — defaults to `0.0.0.0`.

Operational note: startup currently throws if `ARCJET_KEY` is missing.

---

12) Database Migrations and Schema Management

Drizzle workflow (from `package.json`):

- `npm run db:generate` — generate migration SQL from schema.
- `npm run db:migrate` — apply migrations to DB.
- `npm run db:studio` — open Drizzle Studio.

`drizzle.config.js` points Drizzle Kit to:

- schema source: `src/db/schema.js`
- output folder: `drizzle/`
- dialect: PostgreSQL

---

13) Operational Characteristics

- Clear separation of concerns (routing, validation, data, WS, security).
- Real-time updates integrated without third-party brokers.
- Strong input validation posture via Zod.
- Schema-driven persistence and migration process.

---

14) Component-by-Component Deep Dive

`src/index.js`

- Bootstraps Express + HTTP server.
- Registers middleware and routers.
- Instantiates WS server and stores broadcast functions in `app.locals` so HTTP handlers can trigger real-time messages.

`src/arcjet.js`

- Constructs two Arcjet clients:
  - `httpArcjet` for Express middleware checks.
  - `wsArcjet` for upgrade request protection.
- Exposes `securityMiddleware()` used globally on HTTP routes.

`src/db/db.js`

- Loads environment config.
- Creates a shared `pg.Pool` and wraps it with Drizzle.

`src/db/schema.js`

- Defines schema tables and enum in code-first style.
- Serves as the canonical source used by Drizzle Kit to generate migrations.

`src/routes/matches.js`

- Implements list/create endpoints for matches.
- Enforces payload/query rules with Zod.
- Derives status from schedule windows.
- Broadcasts newly created matches over WS.

`src/routes/commentary.js`

- Implements list/create commentary under a match context.
- Validates both route params and request body.

- Broadcasts inserted commentary to WS subscribers of target match.

`src/validation/matches.js` / `src/validation/commentary.js`

- Central contract layer for incoming API traffic.
- Encodes type checks, coercions, and business invariants.

`src/utils/match.status.js`

- Encapsulates schedule-to-status derivation logic.
- Contains helper to synchronize persisted status if stale.

`src/ws/server.js`

- Manages WS lifecycle:
  - upgrade gate
  - connect/disconnect
  - heartbeat
  - subscription bookkeeping
- Exposes broadcast methods consumed by HTTP route layer.

---

15) Suggested Quick Start (Developer)

1. Create `.env` with `DATABASE_URL`, `ARCJET_KEY`, and optionally `ARCJET_MODE`, `PORT`, `HOST`.
2. Install dependencies: `npm install`.
3. Generate/apply schema: `npm run db:generate && npm run db:migrate`.
4. Run server: `npm run dev` (or `npm start`).
5. Open WS client to `/ws` and subscribe to match IDs for live commentary stream.