EventsController

September 5, 2026

EventsController

Kind: Controller

Source: `atloria-monorepo/apps/api/src/events/events.controller.ts`

Public event-ingest endpoint for browser surfaces (docs viewer, chat widget). Anonymous
allowed (public docs visitors give feedback too); userId is attached when authenticated.
Throttled — it's an append-only signal sink, not an API.

`EventsController` exposes a public, anonymous-friendly event ingestion endpoint used by browser surfaces like the docs viewer and chat widget. It accepts append-only telemetry/feedback signals, attaching a `userId` when the caller is authenticated, and is intentionally throttled because it is a signal sink rather than a general-purpose API. It routes incoming event payloads into the backend event pipeline/storage without exposing read/query capabilities.

Diagram

```
graph LR
  A[Browser Surface<br/>Docs Viewer / Chat Widget] -->|POST /events| B[EventsController]
  B -->|Apply throttling / validation| C[Events Service / Ingestion Pipeline]
  C --> D[(Event Store / Queue)]
  B -->|If authenticated| E[Auth Context]
  E -->|Attach userId| C
```

Usage

```
// Browser-side: send an event from docs viewer / chat widget.
// This endpoint is designed to be callable without auth (anonymous),
// but if you include credentials (cookie/JWT), the backend may attach userId.

type PublicEventPayload = {
```

```
  name: string;                // e.g. "docs_feedback_submitted"
  source: "docs" | "chat";
  timestamp?: string;          // ISO string; optional if server sets
  metadata?: Record<string, any>;
};

async function ingestEvent(payload: PublicEventPayload) {
  const res = await fetch("https://api.example.com/events", {
    method: "POST",
    headers: { "content-type": "application/json" },
    // If your app uses cookies for auth, keep credentials enabled.
    credentials: "include",
    body: JSON.stringify(payload),
  });

  if (!res.ok) {
    // Because this endpoint is throttled, 429 is a common response during bursts.
    // Your UI should treat failures as non-blocking.
    const text = await res.text().catch(() => "");
    throw new Error(`Event ingest failed (${res.status}): ${text}`);
  }

  // Often returns a simple ack; treat as fire-and-forget.
  return res.json().catch(() => ({}));
}

// Example calls
ingestEvent({
  name: "docs_page_viewed",
  source: "docs",
  metadata: { path: "/getting-started", referrer: document.referrer },
});

ingestEvent({
  name: "chat_widget_opened",
  source: "chat",
  metadata: { location: window.location.href },
});
```

AI Coding Instructions

- Keep this controller **write-only / append-only**: do not add read/query endpoints here; route analytics reads through a separate, secured API.
- Preserve **anonymous access** while enriching events opportunistically: attach `userId` only when an authenticated context is present; never require auth for basic ingest.
- Maintain and test **throttling behavior** (429 responses are expected); client-side usage should be best-effort and non-blocking.

- Validate and sanitize incoming payloads defensively (size limits, allowed fields) to avoid turning the endpoint into a general data dump surface.
- Integrate with the existing ingestion pipeline/service rather than adding persistence logic inside the controller; keep controller thin and side-effect routing centralized.

Relationships

- MODULE_DECLARES → `record`
- DEPENDS_ON → `EventsService`
- DEPENDS_ON → `PrismaService`

Referenced By

- `EventsModule` (MODULE_DECLARES)