NotificationController

September 4, 2026

NotificationController

Kind: Controller

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

Reader API for the in-app notification feed (navbar bell). The producers (comments, mentions, docs-generation completion hooks) already write rows; this controller is what finally makes them readable — journey fix P1.d "The Silent Factory". Every route is scoped to the JWT caller.

`NotificationController` is a NestJS reader API for the in-app notification feed (the navbar bell). It exposes JWT-scoped endpoints that let the authenticated caller list/read their notification rows and update read state, while producers (comments, mentions, doc-generation hooks) only write the underlying records. This controller is the “last mile” that makes notification data consumable per user.

Diagram

```
graph LR
  A[Producers<br/>(comments, mentions, docs hooks)] -->|write rows| B[(Notifications Store)]
  C[JWT Caller<br/>(web app)] -->|HTTP requests| D[NotificationController]
  D -->|auth scope: caller| E[NotificationService/Queries]
  E -->|read / update read state| B
  D -->|DTO response| C
```

Usage

```
// Client-side example (e.g., web app) calling the NotificationController endpoints.
// Assumes the controller is mounted under /notification (common NestJS convention).
// Adjust paths to match your actual route decorators.

async function fetchNotifications(jwt: string) {
  const res = await fetch(`${process.env.API_URL}/notification`, {
    method: "GET",
```

```
    headers: { Authorization: `Bearer ${jwt}` },
  });

  if (!res.ok) throw new Error(`Failed to fetch notifications: ${res.status}`);
  return (await res.json()) as {
    items: Array<{ id: string; title?: string; createdAt: string; readAt?: string }>;
    nextCursor?: string;
  };
}

async function markNotificationRead(jwt: string, id: string) {
  const res = await fetch(`${process.env.API_URL}/notification/${id}/read`, {
    method: "POST",
    headers: { Authorization: `Bearer ${jwt}` },
  });

  if (!res.ok) throw new Error(`Failed to mark read: ${res.status}`);
  return res.json();
}

async function markAllRead(jwt: string) {
  const res = await fetch(`${process.env.API_URL}/notification/read-all`, {
    method: "POST",
    headers: { Authorization: `Bearer ${jwt}` },
  });

  if (!res.ok) throw new Error(`Failed to mark all read: ${res.status}`);
  return res.json();
}
```

AI Coding Instructions

- Keep every route strictly scoped to the JWT caller (never accept a `userId` param/body for reads or mutations; derive identity from the auth context/guard).
- Follow the existing read-model pattern: producers write notification rows; the controller should only read/ack/update read state, not generate notifications.
- Ensure pagination/filtering is stable and deterministic (e.g., cursor-based by `createdAt` / `id`), and always apply the caller scope before ordering/limits.
- Be careful with “mark as read” operations: enforce ownership checks at the query level to prevent cross-user updates and avoid leaking existence via error messages.
- Maintain contract consistency for the navbar bell: lightweight list endpoint + unread count/read-state updates; coordinate DTO changes with frontend consumers.

Relationships

- MODULE_DECLARES → `list`
- MODULE_DECLARES → `markRead`
- MODULE_DECLARES → `readAll`
- DEPENDS_ON → `NotificationService`

Referenced By

- `NotificationModule` (MODULE_DECLARES)