Question 1 – Architecture for a Netflix Originals calendar

I’ve worked on near-real-time (NRT) systems like this in the past where it is essentially a “distributed database” problem with data coming in from multiple places all in different formats.

**Assumptions:**

* Upper bound on active movie\_ids: 600 current shows -> 2x magnitude 60,000
* Calendar UI is a web GUI
* Need to support year/month/day views
* Calendar entries are discreet dates (no multi-date events spanning days/ months/years)
* All historical data will be retained
* Data received from µSvc will only be filtered on is\_netflix\_original Boolean
* The calendar UI needs to support an upper bound of 10,000 users (current Netflix headcount of 3,500 \* ~3 since not all current employees will use the UI, but external users will likely also exist)
* µSvcs are JSON-over-HTTP(S)
* Ω µSvc uses auto-incrementing integer as movie\_id value

**Startup conditions:**

* Data will need to be loaded from external systems

**Backend Implementation:**

* PostgreSQL for datastore
  + Calendars table with forecast\_launch\_date, movie\_id, is\_netflix\_original and any other fields sourced from various µSvcs to be delivered as part of an item in the UI
  + Soft deletes!
  + CalendarUpdates recording “sequence numbers”: id, calendar\_id, created\_at, and perhaps JSON indicating the type of change
  + Events table for each µSvc from which data is collected
  + Probably use pg child tables broken up by month
* If no webhooks, system is reduced to polling which doesn’t scale very well for real-time updates:
  + In a real-time update use case, for each of the 60,000 potential shows, Calendar’s backend would need to constantly be making a request to each µSvc to check for updates.
  + If real-time updates weren’t required, and only 1- or 5-minute updates were needed, the sequence number scheme described below combined with HTTP HEAD and ETag headers is a possible solution to allow lightweight poll-based “alerting” of the presence of updated data available to be fetched.
* The way I have done this in the past is to have each µSvc use sequence numbers (incremented when data is updated) and send out notifications via webhooks.
  + Each time a µSvc resource is inserted/updated, update the sequence number for the resource’s datastore (even if it is a soft-delete) and then log the change to an “updates” table (using a DB trigger) and fire notifications to webhook subscribers.
  + The initial startup conditions are relatively a straightforward snapshot+deltas: subscribe to the webhook, then start pulling resources by movie\_id from 1 and incrementing up until HTTP 404; then discard all received webhook notifications w/ a seq\_num < max fetched seq\_num. Begin applying subsequent notifications to the local datastore.
  + The webhook interface to each µSvc would ideally include a heartbeat option that would include current\_time and most recent seq\_num, or this could be provided as a REST endpoint that would use ETag headers w/ HTTP HEAD requests for lightweight checking of the service to see about missed messages. Actually, the ETag + HTTP HEAD could be used to poll if no webhooks were implemented/available and updates didn’t need to be in real-time.
  + In addition, the µSvc would have a /seq\_num/:id route for fetching missed seq\_num data should a webhook notification be missed for some reason (or in the non-real-time polling scenario described above).
* As data is received on our webhook listener, it gets put into the appropriate table for that µSvc, and on insert/update/delete it would run some logic to combine this new data with existing data from other µSvc to produce the desired info for our UI.

**Frontend Support:**

* As data is received, a background daemon would periodically (every 1- or 5-minutes) write calendar entries—with each event’s current sequence number—to static JSON files in a path such as /calendar/YYYY, /calendar/YYYY/month/MM, /calendar/YYYY/week/WW for all year/month/weeks found in pg for fast and easy service over HTTP (with ETag).
* Static data files are simple and scales, but lacks real-time updates. To facilitate real-time updates, I have used REST handlers (Ruby on Rails) that fire internal events over a message bus (built using ZeroMQ), and those messages are then bridged through to websocket listeners (IIRC the bridge used eventmachine) to facilitate pushing updates out effectively in real-time to connected web GUI clients. The same pattern could be used to fire data change events off DB triggers as records in the consolidated Calendars table change over time.
* The Calendar backend would likely implement the same interfaces described above for µSvcs and the Calendar web GUI would use a combination of heartbeats-over-websockets and making requests to the backend for any data changed since the time served up from the static JSON file fetched as the original data snapshot.