Jade Ellis (https://jade.ellis.link)

The Potential Pitfalls of Pagination

Published on 7/13/2024 (https://jade.ellis.link/blog/2024/07/14/the-potential-pitfalls-of-pagination)
by Jade Ellis (https://jade.ellis.link) · 4 min read

▶ Table of Contents

Pagination is a core part of many APIs - whether that's lazily loading notifications, filling in financial transactions or retrieving references. However, common pagination methods come with significant pitfalls that might result in a poor user experience - or threats to the availability of your app. In this blog post, we'll explore these common issues and discuss why ID-based pagination often provides a more reliable solution.

Naïve Offset-Based Pagination

Offset-based pagination is a straightforward approach where users receive a fixed number of items starting from a specified offset. For example, the first page might return items 0-19, the second page items 20-39, and so on.

An API request using this method might look like the following, which was proposed in a project I was looking at just a few days ago:

```
GET /notification/<user_id>/fetch?offset=0&count=100
```

While simple to implement, this method has notable drawbacks:

Issue 1: Data Shifting

Consider a user who opens a notification page and retrieves the first 20 notifications. If new notifications arrive before the user requests more data, the new notifications will shift the subsequent data. This results in the user seeing some notifications from the first page duplicated onto the second page.

This is a symptom of the fact that these offset pages are not actually linked in any way to the data that the user receives - if it changes in any way, the data shifts - but the 'window' the user is looking through doesn't shift with it, resulting in an inconsistent view of the data. It's almost like a rolling shutter effect (https://en.wikipedia.org/wiki/Rolling_shutter#/media/File:Rolling_shutter_effect_animation.gif) on the picture the user gets of your data.

Issue 2: Database Scanning

Offset-based pagination systems often become highly inefficient with large datasets. This is because these implementations scan through the data and count rows to determine where to start fetching results. For instance, if a user requests 20 items starting at offset 999,980, the system might need to scan and skip through nearly a million items before returning the desired data. This can significantly impact performance and increase load on the database. You can mitigate this by using indexes - but databases don't do this by default.

Time-Based Pagination

```
GET /api/notifications?start_time=2024-07-13T23:00:00Z&end_time=2024-07-13T23:50:00Z
```

Time-based pagination retrieves items based on a timestamp, fetching entries created within a specific timeframe. This gives the distinct advantage of supporting jumping to a specific time or day if needed. It also gracefully handles old records getting deleted or new records getting inserted. While this can work well in some scenarios, it has its own set of challenges:

Issue 1: Excessive Requests with Sparse Data

Imagine a scenario where a user requests today's notifications but has not received any. The system then automatically fetches notifications from the previous day, and continues this process until data is found. If many users do this simultaneously, it can create a flood of requests, potentially leading to a self-inflicted DDoS (Distributed Denial of Service) attack.

This can be mitigated by different techniques, including using an appropriate index to start pagination from the most recent record and starting pagination from that point. [1]

Issue 2: Overwhelming Large Data Sets

Conversely, if a user has received a large number of notifications in a short period, the system might retrieve an overwhelming number of items in one go. For example, fetching 1200 notifications at once can cause performance issues on the client side, leading to slow load times or even crashes.

This is very simple to fix - simply adding a limit to the amount of records returned (potentially via a query parameter) transparently solves it.

ID-Based Pagination

ID-based pagination involves using a record's unique identifier (ID) to paginate through the dataset. A typical request might ask for the first 50 records, and subsequent requests would ask for the next 50 records after the last seen ID. One example of this is the Reddit API:

```
GET /r/subreddit/new?limit=25&after=t3_10omtd
```

This approach offers several advantages. Using persistent IDs ensures that pagination remains consistent even if new records are added or existing ones are deleted, and allows indexing and incredibly fast retrieval of records, rather than requiring scanning large numbers of records.

While ID-based pagination is robust, it is not without its own potential pitfalls.

Issue 1: Deleted Anchors

One issue could arise if a record which a user is using as an anchor (Usually the last record in a page) is deleted while a user is paginating. This could result in an error, interrupting pagination. However, this can be mitigated transparently if identifiers are inherently ordered, like sequential IDs and some kinds of UUIDs, or when deletions are 'soft' - marking a record as deleted in the database without fully removing it. In the worst case, this can be mitigated by different smart retry strategies on the client.

Conclusion

Now you know the pitfalls of these pagination techniques, you can pick the best method for your API - and avoid any bugs or performance issues!

---

1. Thank you Michael Wiencek for suggesting this method 🙂 ↩

Jade Ellis (https://jade.ellis.link)

Report a bug

Connect

Matrix (https://matrix.to/#/@jade:ellis.link)
GitHub (https://github.com/JadedBlueEyes)
Mastodon (https://tech.lgbt/@JadedBlueEyes)
Bluesky (https://bsky.app/profile/jade.ellis.link)
LinkedIn (https://www.linkedin.com/in/jadedblueeyes)

Feeds

RSS (Atom) (https://jade.ellis.link/blog/rss.xml)
JSON Feed (https://jade.ellis.link/blog/feed.json)