**API PAGINATION**

Most of the use cases API’s are REST APIs, but it applies to any SOAP, REST, GraphQL, etc. The most common pagination techniques are Page-based pagination (also called offset-based pagination), KeySet-Based pagination, and Cursor-based Pagination.

**Page — based pagination**

This is the simplest and most common form of paging, particularly for Apps that use SQL Databases. The endpoint accepts a page param that is an integer indicating the page within the list to be returned.

If it’s using SQL, It will do a query using LIMIT and OFFSET, where the first is the length of the page and the latter is the number of records already returned (page's size \* page).

**Pros:**

• You can jump to any particular page, not need to query 99 pages to get the page 100

Easy to understand and debug

• It allows sending parallel requests with different pages.

**Cons:**

• Bad performance for large OFFSET in SQL. When doing OFFSET Nin SQL, the database needs to scan and count N rows.

• It can return repeated or missing if any is added/deleted while paginating

Client Request: GET/products?limit=20&page=1

Server Response: products { 1………20}

Client Request: GET/products?limit=20&page=2

Server Response: products { 21………40}.

Query: SELECT \* FROM products ORDER BY id LIMIT 20 OFFSET 200;

**KeySet-based pagination**

The API provides a key param that acts as a delimiter of the page. This key param should be the same key of the set sort order. For example, if the set is sorted by ID, then the key param should be since\_id.

The response of this request will contain the value of the key for the last element of the set. For instance, if the delimiter is the id, to get the next page, the client needs to send the param since\_id with the value id of the last element of the response, and the set must be sorted by id.

**Pros:**

• The SQL query is more efficient than OFFSET (for most cases) since it uses a WHERE condition.

• New records inserted on previous pages won’t cause duplicated elements.

**Cons:**

• There is no way to jump for a specific page. It needs to iterate through all the prior pages

• It doesn’t allow sending parallel requests for different batches

Client Request: GET/products?limit=20

Server Response: products { 1………20}

Client Request: GET/products?limit=20&since\_id=20

Server Response: products { 21………40}.

Query: SELECT \* FROM products WHERE id > <since\_id> ORDER BY id LIMIT 100

**Cursor-based Pagination**

A cursor will be a piece of data that contains a pointer to an element and the info to get the next/previous elements. The server should return the cursor pointing to the next page in each request.

There are different approaches to implement this method. Some return the cursor as part of the *payload*, others return the cursor as part of the Header, particularly in the link headers. The cursor might contain all the information needed or partially, allowing clients to add other filter params.

**Pros:**

* In SQL, for most of the cases, it is much faster than using page since it won’t use OFFSET in the Database.
* There is no issue when a record is deleted as opposed to Page-based Pagination.

**Cons:**

* There is no way to skip pages. If the user wants page X, it needs to request pages from 1 to X.
* It doesn’t allow sending parallel requests for different batches.
* The implementation is more complex than LIMIT/OFFSET.

Client Request: GET/products?limit=20

Server Response: products { 1………20}

[{id:1,name:A},{ id:2,name:B }……………………{id:20,name:M}],page-info:{next-cursor: ‘ancdef’}

Client Request: GET/products?limit=20&cursor= ancdef’

Server Response: products { 21………40}.

[{id:21,name:A},{ id:22,name:B }……………………{id:40,name:M}], page-info:{next-cursor: ‘ijhkl}