database # express # guide

Pagination & Filtering

TCSS 460 — Client/Server Programming

Your database will grow. A movie table might start with 20 rows during development and end up with 20,000 in production. If your API returns all of them in a single response, you are sending megabytes of JSON over the wire, forcing the client to wait for data it cannot display, and putting unnecessary load on your database. This guide teaches you how to build API endpoints that return data in manageable chunks — paginated, filtered, and sorted — using Prisma.

> **About the Examples**
>
> This guide uses a `Movie` model to illustrate pagination and filtering concepts. This is not a model from your group project — the patterns shown here apply to any collection endpoint regardless of the data.

---

1 Why Pagination?

Imagine a front-end that displays a list of movies. The user sees 20 movies per page. Without pagination, your API returns every movie in the database, and the front-end throws away everything except the first 20. With pagination, your API returns only the 20 movies the front-end actually needs.

1.1 The Performance Problem

Returning an entire collection creates problems at every layer:

Layer	Without Pagination	With Pagination
Database	Reads and serializes all rows	Reads only the rows needed
Network	Transfers megabytes of JSON	Transfers kilobytes

Layer	Without Pagination	With Pagination
Server	Holds all rows in memory	Holds one page in memory
Client	Parses a massive array, renders a fraction	Parses and renders only what is displayed

During development, you might have 10 rows and never notice the problem. In production, with thousands or millions of rows, an unpaginated endpoint becomes a performance bottleneck — or crashes outright.

1.2 Real APIs Always Paginate

This is not an academic exercise. Every major API you will encounter uses pagination:

- **GitHub** — `GET /repos/{owner}/{repo}/issues?page=2&per_page=30`
- **Stripe** — `GET /v1/charges?limit=25&starting_after=ch_abc123`
- **TMDB** — `GET /3/discover/movie?page=3`

If you have used the TMDB API in your proxy assignment, you have already seen this — TMDB returns 20 results per page and provides a `total_pages` field so the client knows how many pages exist. Your API should do the same.

> **Design Principle**
>
> The REST API Design concept reading calls unpaginated endpoints a "time bomb" (Section 8.5). Always design collection endpoints with pagination from the start — retrofitting pagination later means changing your response format, which breaks every client that depends on it.

---

2 Offset-Based Pagination

Offset-based pagination is the simplest and most common approach. The client says "skip the first N rows and give me the next M rows." If you have written SQL, you already know the building blocks.

2.1 The SQL Foundation

In SQL, pagination uses `LIMIT` and `OFFSET`:

```
-- Page 1: first 20 movies
SELECT * FROM movies ORDER BY id LIMIT 20 OFFSET 0;

-- Page 2: movies 21-40
SELECT * FROM movies ORDER BY id LIMIT 20 OFFSET 20;

-- Page 3: movies 41-60
SELECT * FROM movies ORDER BY id LIMIT 20 OFFSET 40;
```

`LIMIT` controls how many rows to return. `OFFSET` controls how many rows to skip before starting. The pattern is straightforward: for page `P` with a page size of `L`, the offset is `(P - 1) * L`.

2.2 Prisma: `skip` and `take`

Prisma translates the same concept into its query API using `skip` and `take`:

```
const movies = await prisma.movie.findMany({
    skip: 20,   // skip the first 20 rows (OFFSET)
    take: 20,   // return the next 20 rows (LIMIT)
    orderBy: { id: 'asc' },
});
```

- **`take`** — How many records to return. This is your page size.
- **`skip`** — How many records to bypass before returning results. This is the offset.

2.3 Query Parameters: `page` and `limit`

Your API consumers should not have to calculate offsets themselves. Instead, expose `page` and `limit` as query parameters and calculate `skip` on the server:

```
GET /movies?page=2&limit=20
```

The calculation:

```
const page = 2;
const limit = 20;
const skip = (page - 1) * limit;  // (2 - 1) * 20 = 20
```

Page 1 skips 0 rows. Page 2 skips 20. Page 3 skips 40. The consumer thinks in pages; the server translates to offsets.

2.4 Pros and Cons

Offset-based pagination is simple and widely understood, but it has a known weakness:

Advantage	Disadvantage
Easy to implement	Performance degrades on large offsets — `OFFSET 100000` still scans 100,000 rows before returning results
Clients can jump to any page (`page=50`)	If rows are inserted or deleted between requests, items can be skipped or duplicated
Works well for datasets under ~100,000 rows	Not ideal for real-time feeds or very large datasets

For most CRUD applications — and certainly for this course — offset-based pagination is the right choice. The performance concern only becomes real at scale (hundreds of thousands of rows with deep page numbers).

---

3 Cursor-Based Pagination

Cursor-based pagination solves the performance and consistency problems of offset pagination by using a **cursor** — a pointer to a specific record — instead of a numeric offset.

3.1 What is a Cursor?

A cursor is the unique identifier (usually the `id` or a timestamp) of the last item the client received. Instead of saying "skip 100 rows," the client says "give me rows after the one with ID 100."

```
First request:  GET /movies?limit=20
Response:       movies 1-20, cursor = 20

Second request: GET /movies?limit=20&cursor=20
Response:       movies 21-40, cursor = 40

Third request:  GET /movies?limit=20&cursor=40
Response:       movies 41-60, cursor = 60
```

The database does not scan and discard rows. It jumps directly to the cursor position and reads forward — regardless of whether the cursor points to row 20 or row 200,000.

3.2 Prisma: `cursor` Parameter

Prisma supports cursor-based pagination natively:

```
const movies = await prisma.movie.findMany({
    cursor: { id: 40 },    // start after this record
    skip: 1,               // skip the cursor record itself
    take: 20,              // return the next 20
    orderBy: { id: 'asc' },
});
```

The `skip: 1` is necessary because `cursor` positions the query *at* the specified record — without `skip: 1`, the response would include the cursor record itself (which the client already has).

3.3 When to Prefer Cursor Over Offset

Scenario	Best Approach
Standard CRUD app with moderate data	Offset — simpler, clients can jump to any page
Large datasets (100K+ rows)	Cursor — no performance degradation on deep pages
Real-time feeds (new data arriving constantly)	Cursor — no skipped or duplicated items
User needs to jump to page 50 directly	Offset — cursor only supports "next page"
Infinite scroll UI	Cursor — each scroll loads the next batch

Which Will You Use?

In this course, you will primarily use offset-based pagination. It is simpler, it works well for the project's data sizes, and it is what most students will encounter in their first industry roles. Cursor-based pagination is presented here so you understand the tradeoff and can recognize it when you see it in production APIs like Stripe or Slack.

4 Building a Paginated Endpoint

Let's build a real paginated endpoint step by step. We will use the offset approach since it is the most common pattern.

4.1 Parsing Query Parameters

Express query parameters are **always strings**. A request to `/movies?page=2&limit=20` gives you `request.query.page === "2"` (a string, not a number). You must parse and validate them before using them in a Prisma query:

```
import { Request, Response } from 'express';

export const getMovies = async (request: Request, response: Response) => {
    const page = Math.max(1, Number(request.query.page) || 1);
    const limit = Math.min(100, Math.max(1, Number(request.query.limit) ||
25));
    const skip = (page - 1) * limit;

    // ... Prisma query goes here
};
```

Three things to notice:

1. **`Number(request.query.page) || 1`** — If `page` is missing, undefined, or not a valid number, `Number()` returns `NaN`, and `NaN || 1` falls back to `1`. Default: page 1.
2. **`Math.max(1, ...)`** — Prevents page 0 or negative pages.
3. **`Math.min(100, Math.max(1, ...))`** — Clamps `limit` between 1 and 100. Without an upper bound, a client could request `limit=1000000` and defeat the purpose of pagination entirely.

> ⚠ **Always Cap the Limit**
>
> If you let clients set an arbitrarily large `limit`, they can bypass pagination by requesting `?limit=999999`. Pick a reasonable maximum (25, 50, or 100) and enforce it on the server. The client should not be able to opt out of pagination.

4.2 The Prisma Query

With parsed parameters, the query is straightforward:

```
const movies = await prisma.movie.findMany({
    skip,
    take: limit,
    orderBy: { id: 'asc' },
});
```

`skip` and `take` map directly to SQL's `OFFSET` and `LIMIT`. Prisma handles the translation.

4.3 Putting the Handler Together

Here is the complete route handler:

```
import { Request, Response } from 'express';
import { prisma } from '../prisma';

export const getMovies = async (request: Request, response: Response) => {
    const page = Math.max(1, Number(request.query.page) || 1);
    const limit = Math.min(100, Math.max(1, Number(request.query.limit) ||
25));
    const skip = (page - 1) * limit;

    try {
        const movies = await prisma.movie.findMany({
            skip,
            take: limit,
            orderBy: { id: 'asc' },
        });

        response.json({ data: movies });
    } catch (error) {
        response.status(500).json({ error: 'Failed to retrieve movies' });
    }
};
```

This works, but it is incomplete. The client knows what data is on this page, but it does not know how many pages exist or how many total results there are. That is where response metadata comes in.

Try It Yourself

1. Add a `GET /movies` route to your Express app that calls `getMovies`
2. Seed your database with at least 50 movies
3. Test in Postman: `GET /movies` (defaults), `GET /movies?page=2&limit=10`, `GET /movies?page=1&limit=5`
4. Verify that different page/limit combinations return different subsets

5 Response Metadata

A paginated response is not just a list of items — it includes metadata that tells the client where it is in the overall dataset. Without metadata, the client cannot build pagination controls ("Page 2 of 8") or know when it has reached the last page.

5.1 Getting the Total Count

Prisma's `count()` method returns the total number of records matching a query:

```
const total = await prisma.movie.count();
```

> **COUNT(*) Performance in PostgreSQL**
>
> Unlike some databases, PostgreSQL does not maintain a cached row count. Every `COUNT(*)` query scans the table (or an index) to count rows, making it O(n) in the number of matching rows. For the data sizes in this course (hundreds to low thousands of rows), this is fast enough to be invisible. At production scale (millions of rows), the count query can become a bottleneck — but that is a problem for a future course.

If you are also filtering (covered in Section 6), pass the same `where` clause to `count()` so the total reflects the filtered dataset, not the entire table:

```
const where = { genre: 'drama' };

const [movies, total] = await Promise.all([
    prisma.movie.findMany({ where, skip, take: limit }),
    prisma.movie.count({ where }),
]);
```

Using `Promise.all` runs both queries concurrently — the count query does not wait for the data query to finish, so the total response time is the duration of the slower query, not the sum of both.

5.2 Calculating Total Pages

With the total count and the page size, calculating the number of pages is one line:

```
const totalPages = Math.ceil(total / limit);
```

`Math.ceil` rounds up because a partial page still counts as a page. If you have 53 movies and a limit of 20, that is 3 pages (20 + 20 + 13).

5.3 The Response Shape

A well-structured paginated response separates the data from the pagination metadata:

```
{
    "data": [
        { "id": 21, "title": "The Matrix", "genre": "sci-fi", "year": 1999 },
        { "id": 22, "title": "Inception", "genre": "sci-fi", "year": 2010 }
    ],
    "pagination": {
        "page": 2,
        "limit": 20,
        "total": 53,
        "totalPages": 3
    }
}
```

This shape tells the front-end everything it needs:

- **`page`** — Which page you are on (for highlighting the current page button)
- **`limit`** — How many items per page (for displaying "Showing 21-40")
- **`total`** — Total matching records (for displaying "53 results")
- **`totalPages`** — How many pages exist (for rendering page numbers or disabling the "Next" button on the last page)

> **This Matches the REST Reading**
>
> The REST API Design concept reading (Section 5.3) shows exactly this response shape — a `data` array and a `pagination` object. Your implementation now matches the theory.

5.4 Updated Handler with Metadata

Here is the handler updated to include response metadata:

```
import { Request, Response } from 'express';
import { prisma } from '../prisma';

export const getMovies = async (request: Request, response: Response) => {
    const page = Math.max(1, Number(request.query.page) || 1);
    const limit = Math.min(100, Math.max(1, Number(request.query.limit) ||
25));
    const skip = (page - 1) * limit;
```

```
    try {
        const [movies, total] = await Promise.all([
            prisma.movie.findMany({
                skip,
                take: limit,
                orderBy: { id: 'asc' },
            }),
            prisma.movie.count(),
        ]);

        const totalPages = Math.ceil(total / limit);

        response.json({
            data: movies,
            pagination: { page, limit, total, totalPages },
        });
    } catch (error) {
        response.status(500).json({ error: 'Failed to retrieve movies' });
    }
};
```

Try It Yourself

1. Update your `GET /movies` handler to include the `pagination` object
2. In Postman, request `GET /movies?page=1&limit=10`
3. Check the `pagination` object — does `totalPages` equal `Math.ceil(total / 10)`?
4. Request a page beyond the last page (e.g., `?page=999`) — you should get an empty `data` array but still valid pagination metadata

---

6 Composable Filtering

Pagination controls how many results you get per request. Filtering controls *which* results you get. The two work together — a client might want "page 2 of drama movies from 2024, sorted by rating."

This section implements the pattern described in the REST API Design concept reading (Section 4.5). That reading teaches the *principle*: one endpoint with composable query parameters instead of many specialized search endpoints. Here you will learn to *build* it.

6.1 The Principle: One Endpoint, Many Filters

Recall the anti-pattern from the REST reading — a separate endpoint for every filter combination:

```
GET /movies/search-by-genre?genre=drama         -- only genre
GET /movies/search-by-year?year=2024             -- only year
GET /movies/search-by-genre-and-year?genre=drama&year=2024  -- both?
```

This scales terribly. Every new filter multiplies the number of endpoints.

The idiomatic approach is a single endpoint where every filter is an optional query parameter:

```
GET /movies?genre=drama
GET /movies?year=2024
GET /movies?genre=drama&year=2024&minRating=8
GET /movies?genre=drama&year=2024&minRating=8&page=2&limit=20
```

The consumer picks the filters they need and ignores the rest. Pagination and filtering use the same query string — they compose naturally.

6.2 Building the `where` Object Conditionally

The key implementation technique is building a Prisma `where` object that includes only the filters the client actually sent. If the client does not send a `genre` parameter, the `where` object should not filter by genre.

The **spread operator pattern** makes this clean:

```
const { genre, year, minRating } = request.query;

const where = {
    ...(genre ? { genre: String(genre) } : {}),
    ...(year ? { year: Number(year) } : {}),
    ...(minRating ? { rating: { gte: Number(minRating) } } : {}),
};
```

Let's break this down line by line:

- **`...(genre ? { genre: String(genre) } : {})`** — If `genre` is present, spread `{ genre: "drama" }` into the `where` object. If `genre` is absent, spread `{}` (nothing). The filter only exists when the client asks for it.
- **`...(year ? { year: Number(year) } : {})`** — Same pattern. Convert the string query parameter to a number for the database.
- **`...(minRating ? { rating: { gte: Number(minRating) } } : {})`** — Uses Prisma's `gte` (greater than or equal) operator. `{ rating: { gte: 8 } }` becomes `WHERE rating >= 8` in SQL.

If the client sends `GET /movies?genre=drama&minRating=8`, the `where` object becomes:

```
{ genre: "drama", rating: { gte: 8 } }
```

If the client sends `GET /movies` with no filters, the `where` object becomes:

```
{}
```

An empty `where` object means "no filters" — Prisma returns all records (subject to pagination).

6.3 Why This Pattern Works

The spread operator pattern has three advantages:

1. **Each filter is independent.** Adding a new filter means adding one line. Removing a filter means removing one line. No `if/else` chains, no nested conditionals.
2. **Filters compose automatically.** The client can combine any subset of filters in any combination. You do not need to anticipate which combinations are useful.
3. **The same `where` object works for both queries.** Pass it to `findMany` for the data and to `count` for the total — the count always reflects the filtered dataset.

> **Gen AI & Learning: Writing Filter Logic**
>
> Building conditional `where` objects is a pattern that AI coding assistants handle well — the input (query parameters) and output (Prisma `where` clause) are well-defined. If you need to add filters for ten fields, describe the fields and their types, and let the agent generate the spread pattern. The design decision — *which* fields should be filterable — is yours. The mechanical translation to Prisma syntax is where the agent saves time.

6.4 Adding Sorting

Sorting follows the same composable pattern. The client sends `sort` and `order` query parameters, and you translate them into Prisma's `orderBy`:

```
const sort = String(request.query.sort || 'id');
const order = request.query.order === 'desc' ? 'desc' : 'asc';

const movies = await prisma.movie.findMany({
    where,
    skip,
    take: limit,
```

```
    orderBy: { [sort]: order },
});
```

The `[sort]` syntax is a computed property name — if `sort` is `"rating"`, `{ [sort]: order }` becomes `{ rating: "asc" }`.

> **Validate the Sort Field**
>
> The client is sending a string that becomes a database column name. If the client sends `sort=password` or `sort=nonexistent`, you either leak sensitive data or crash. Validate the sort field against an allowlist:
>
> ```
> const ALLOWED_SORT_FIELDS = ['id', 'title', 'year', 'rating',
> 'genre'];
> const sort =
> ALLOWED_SORT_FIELDS.includes(String(request.query.sort))
>     ? String(request.query.sort)
>     : 'id';
> ```
>
> If the client sends an invalid sort field, silently fall back to the default. This is safer than returning an error for a non-critical parameter.

---

7 Putting It Together

Here is a complete endpoint that combines everything: pagination, filtering, sorting, and response metadata.

7.1 The Complete Handler

```
import { Request, Response } from 'express';
import { prisma } from '../prisma';

const ALLOWED_SORT_FIELDS = ['id', 'title', 'year', 'rating', 'genre'];
const DEFAULT_LIMIT = 25;
const MAX_LIMIT = 100;

export const getMovies = async (request: Request, response: Response) => {
    // --- Pagination ---
    const page = Math.max(1, Number(request.query.page) || 1);
    const limit = Math.min(MAX_LIMIT, Math.max(1, Number(request.query.limit)
|| DEFAULT_LIMIT));
    const skip = (page - 1) * limit;
```

```
    // --- Sorting ---
    const sort = ALLOWED_SORT_FIELDS.includes(String(request.query.sort))
        ? String(request.query.sort)
        : 'id';
    const order = request.query.order === 'desc' ? 'desc' : 'asc';

    // --- Filtering ---
    const { genre, year, minRating } = request.query;

    const where = {
        ...(genre ? { genre: String(genre) } : {}),
        ...(year ? { year: Number(year) } : {}),
        ...(minRating ? { rating: { gte: Number(minRating) } } : {}),
    };

    try {
        const [movies, total] = await Promise.all([
            prisma.movie.findMany({
                where,
                skip,
                take: limit,
                orderBy: { [sort]: order },
            }),
            prisma.movie.count({ where }),
        ]);

        const totalPages = Math.ceil(total / limit);

        response.json({
            data: movies,
            pagination: { page, limit, total, totalPages },
        });
    } catch (error) {
        response.status(500).json({ error: 'Failed to retrieve movies' });
    }
};
```

7.2 Walking Through the Handler

The handler has four distinct sections, each responsible for one concern:

1. **Pagination** — Parse `page` and `limit`, calculate `skip`. Enforce bounds.
2. **Sorting** — Validate the `sort` field against an allowlist. Default the `order` to ascending.
3. **Filtering** — Build the `where` object conditionally using the spread pattern. Only include filters the client actually sent.
4. **Query and Response** — Run `findMany` and `count` concurrently with `Promise.all`. Calculate `totalPages`. Return the data with pagination metadata.

Notice that each section is independent. You can add a new filter without touching the pagination logic. You can change the default sort without affecting the filter logic. This separation of concerns makes the handler easy to extend.

7.3 Example Requests

Here are test scenarios that exercise different filter combinations. Each request goes to the same endpoint — only the query parameters change:

Request	What It Tests
`GET /movies`	Defaults — page 1, limit 25, no filters, sorted by id
`GET /movies?page=2&limit=10`	Pagination only — second page, 10 per page
`GET /movies?genre=drama`	Single filter — all drama movies
`GET /movies?genre=drama&year=2024`	Combined filters — drama movies from 2024
`GET /movies?genre=drama&minRating=8&sort=rating&order=desc`	Filter + sort — top-rated dramas, highest first
`GET /movies?genre=drama&minRating=8&page=2&limit=5`	Filter + paginate — second page of filtered results
`GET /movies?sort=title&order=asc`	Sort only — alphabetical by title
`GET /movies?page=999`	Edge case — page beyond results, empty data array

For each request, verify:

- The `data` array contains the expected items
- The `pagination` object reflects the filters (e.g., `total` should be the count of *drama* movies, not all movies, when `genre=drama` is present)
- Changing `page` returns different items but the same `total` and `totalPages`

Try It Yourself

1. Run all eight test scenarios in Postman
2. For the `genre=drama` request, note the `total` in the response
3. Now add `&year=2024` — does `total` decrease? It should, because you are narrowing the filter
4. Request `?genre=nonexistent` — you should get `{ data: [], pagination: { page: 1, limit: 25, total: 0, totalPages: 0 } }`

---

8 Summary

Concept	Key Point
Why paginate	Returning all rows wastes bandwidth, memory, and time — paginate from the start
Offset pagination	`skip` and `take` in Prisma — simple, supports random page access
Cursor pagination	`cursor` in Prisma — better performance on large datasets, no skipped rows
Query parameters	`page` and `limit` — always strings from Express, parse with `Number()`
Default values	Default to page 1 and a reasonable limit (25); cap the maximum limit
`Promise.all`	Run `findMany` and `count` concurrently for better response times
Response metadata	`{ data, pagination: { page, limit, total, totalPages } }` — clients need this for UI controls
Composable filtering	One endpoint, optional query params, spread operator pattern for the `where` object
Sort validation	Always validate sort fields against an allowlist — never trust client input as a column name

Concept	Key Point
REST alignment	This pattern implements Section 4.5 of the REST API Design reading — one endpoint, composable filters

---

9 References

Official Documentation:

- Prisma — Pagination — `skip`, `take`, and `cursor` parameters for offset and cursor pagination
- Prisma — Filtering and Sorting — `where` clause operators (`equals`, `contains`, `gte`, `lte`, `in`) and `orderBy`
- Prisma — Aggregation, Grouping, and Summarizing — `count()`, `aggregate()`, and `groupBy()`
- Express.js — `req.query` — Query string parsing in Express
- MDN — `Promise.all()` — Running multiple promises concurrently

---

10 Further Reading

External Resources

- REST API Design — Search and Filtering Patterns — Section 4.5 of the concept reading that teaches the composable filtering principle implemented in this guide
- REST API Design — Forgetting About Pagination — Section 8.5 on why unpaginated endpoints are a design mistake
- Prisma ORM Guide — Prerequisite guide covering Prisma Client setup, schema modeling, and basic queries
- Offset vs. Cursor Pagination (Prisma Blog) — In-depth comparison of offset and cursor pagination strategies with performance benchmarks
- Stripe API — Pagination — A production example of cursor-based pagination using `starting_after` and `ending_before`

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.