Mini-Manual: RESTful API – Advanced Topics

Pagination, Caching, and Rate Limiting

Objectives of This Mini-Manual

This mini-manual aims to deepen students' understanding of **advanced RESTful API design techniques** that are essential for building **scalable, efficient, and robust web services**.

By the end of this manual, students should be able to:

- Understand why **pagination, caching, and rate limiting** are necessary in real-world APIs
- Design RESTful endpoints that scale efficiently under heavy load
- Apply best practices aligned with industry standards
- Make informed architectural decisions when designing APIs for production environments

This manual combines **theoretical foundations** with **practical examples**, encouraging students to reason about performance, scalability, and user experience.

Conteúdo

Chapter 1 — Introduction to Advanced RESTful API Design (Node.js & Express)

1.1 From Basic REST APIs to Production-Grade Services

Most students begin working with RESTful APIs by building simple services using **Node.js and Express**, typically exposing endpoints such as:

GET /users

POST /users

GET /users/:id

PUT /users/:id

DELETE /users/:id

These endpoints usually:

- return JSON responses,
- interact with a database,
- follow basic REST conventions,
- work correctly for small datasets and low traffic.

While this is sufficient for **learning core concepts**, it is **not enough** for real-world systems.

In production environments, APIs must support:

- thousands of concurrent clients,
- large and growing datasets,
- unpredictable traffic patterns,
- performance and availability constraints.

At this stage, **advanced RESTful API concerns become mandatory**, not optional enhancements.

1.2 Why "Advanced" REST Topics Are Necessary

- Let us consider a very common Express route:

```
app.get("/api/users", async (req, res) => {
  const users = await User.findAll();
  res.json(users);
});
```

- This endpoint works perfectly during development.
  However, it introduces serious problems when:
  - the users table contains hundreds of thousands of rows,
  - multiple clients request this endpoint simultaneously,
  - the same request is repeated frequently,
  - a malicious client sends requests in a tight loop.

These problems fall into three major categories:

Problem	**Consequence**
Large result sets	High memory usage, slow responses
Repeated identical requests	Wasted CPU and database resources
Excessive requests	Service degradation or denial of service

Pagination, caching, and rate limiting directly address these issues.

1.3 Advanced REST Concerns in the Express Ecosystem

Express is intentionally minimal.
It does **not** enforce:

- pagination strategies,
- caching policies,
- request quotas,
- protection against abuse.

This is by design: Express gives developers flexibility, but also **responsibility**.

As a result, developers must explicitly design and implement:

- paginated endpoints,
- cache-aware responses,
- rate-limited routes.

This mini-manual focuses on **how and why** to do this correctly in Node.js/Express applications.

1.4 Pagination: Controlling Data Volume

The Core Problem

- Returning large collections directly:

```
res.json(users);
```

- causes:
  - long response times,
  - large payloads,
  - poor mobile performance,
  - unnecessary memory pressure.

Pagination ensures that **only a subset of data** is returned per request.

Example of a paginated Express endpoint:

```
GET /api/users?page=1&limit=20
```

Pagination is not merely a client convenience — it is a **server scalability mechanism**.

1.5 Caching: Avoiding Redundant Work

- Consider this Express route:

```
app.get("/api/products", async (req, res) => {
  const products = await Product.findAll();
  res.json(products);
});
```

- If:
  - the data changes infrequently,
  - thousands of clients call this endpoint per minute,

then the server:

- repeatedly executes the same database query,
- generates the same JSON response,
- wastes CPU and I/O resources.

Caching allows the server (or intermediaries) to:

- reuse previously computed responses,
- reduce database load,
- improve response latency.

In RESTful systems, caching is a **first-class architectural feature**, not an afterthought.

1.6 Rate Limiting: Protecting the API

Express APIs are often publicly accessible.
Without safeguards, they are vulnerable to:

- accidental misuse,
- poorly written clients,
- brute-force login attempts,
- denial-of-service attacks.

Example of a problematic endpoint:

POST /api/auth/login

Without rate limiting:

- a client could attempt thousands of logins per second,
- credentials could be brute-forced,
- the server could be overwhelmed.

Rate limiting enforces **fair usage policies** and protects system availability.

1.7 These Topics Are Interdependent

In real-world Express applications, pagination, caching, and rate limiting are rarely implemented in isolation.

- Consider the following endpoint:

```
GET /api/articles?cursor=abc123&limit=10
```

- Pagination limits response size
- Caching avoids repeated processing
- Rate limiting ensures fair access

Together, they form the foundation of a **scalable API design**.

1.8 Typical Architecture in Node.js / Express APIs

In production systems, these concerns may be implemented at different layers:

Layer	Responsibility
Express middleware	Rate limiting, request validation
Application logic	Pagination logic
HTTP headers	Caching directives
Redis / Memory	Application-level caching
API Gateway / CDN	Global rate limiting and caching

This manual focuses primarily on **Express-level design**, while also explaining how these techniques integrate with external infrastructure.

1.9 What This Mini-Manual Will Cover

This manual is structured as follows:

- **Chapter 2 — Pagination**
  Strategies, pitfalls, and Express-based implementations.
- **Chapter 3 — Caching**
  HTTP caching, ETags, and server-side caching in Node.js.
- **Chapter 4 — Rate Limiting**
  Express middleware, algorithms, and best practices.
- **Chapter 5 — Integration & Best Practices**
  Designing production-ready APIs that combine all three concerns.

Each chapter includes:

- theoretical explanation,
- Node.js / Express examples,
- design rationale,
- best practices and common mistakes.

1.10 Learning Outcomes of This Chapter

After this chapter, students should be able to:

- Explain why basic REST APIs do not scale by default
- Identify performance and security risks in naïve Express endpoints
- Understand the role of pagination, caching, and rate limiting in RESTful systems
- Reason about these mechanisms from an architectural perspective

This foundation is essential before diving into implementation details.

1.11 Chapter 1 Summary

- Express APIs require explicit design for scalability.
- Large datasets demand pagination.
- Repeated requests demand caching.
- Public endpoints demand rate limiting.
- These concerns are architectural, not cosmetic.
- Advanced REST design is essential for production systems.

Chapter 2 — Pagination (Node.js & Express)

2.1 Why Pagination Is a First-Class API Concern

In RESTful API design, pagination is often mistakenly treated as a *client-side feature*.
In reality, pagination is a **server-side scalability mechanism**.

- Consider the following Express endpoint:

```
app.get("/api/posts", async (req, res) => {
  const posts = await Post.findAll();
  res.json(posts);
});
```

- This implementation introduces several risks:
  - High memory consumption on the server
  - Long response times
  - Large payloads sent over the network
  - Poor performance on mobile clients
  - Increased database load

Pagination ensures that APIs return **bounded, predictable responses**, regardless of dataset size.

2.2 Pagination Parameters in REST APIs

In Express APIs, pagination is typically controlled through **query parameters**.

Common parameters include:

- page
- limit
- offset
- cursor
- after / before

Example request:

```
GET /api/posts?page=2&limit=10
```

These parameters allow clients to explicitly control **how much data** they receive and **which subset** they are accessing.

2.3 Offset-Based Pagination

2.3.1 Concept

Offset-based pagination retrieves data by skipping a number of records and returning the next subset.

Mathematically:

```
offset = (page - 1) × limit
```

2.3.2 Express Example

```
app.get("/api/posts", async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const offset = (page - 1) * limit;
  const posts = await Post.findAll({
    limit,
    offset
  });
  res.json(posts);
});
```

- This approach is:
  - simple to implement
  - easy for clients to understand
  - widely supported by ORMs and SQL databases

2.3.3 Limitations of Offset-Based Pagination

Despite its simplicity, offset pagination has drawbacks:

- Performance degrades for large offsets
- Results may shift when new records are inserted
- Not suitable for real-time or rapidly changing datasets

Example problem:

- Client requests page 3
- A new record is inserted
- Page boundaries shift unexpectedly

2.4 Cursor-Based Pagination

2.4.1 Concept

Cursor-based pagination uses a **stable reference point** (cursor) instead of numeric offsets.

Typical cursors:

- database IDs
- timestamps
- encoded composite keys

Example request:

```
GET /api/posts?cursor=128&limit=10
```

2.4.2 Express Example

```
app.get("/api/posts", async (req, res) => {
  const limit = parseInt(req.query.limit) || 10;
  const cursor = req.query.cursor;
  let whereClause = {};
  if (cursor) {
    whereClause = { id: { $gt: cursor } };
  }
  const posts = await Post.findAll({
    where: whereClause,
    order: [["id", "ASC"]],
    limit
  });
  res.json(posts);
});
```

- This ensures:
  - stable pagination
  - consistent results
  - better performance for large datasets

2.4.3 Advantages of Cursor-Based Pagination

- Scales efficiently
- Avoids record shifting
- Works well with indexed columns
- Preferred by large-scale APIs (GitHub, Twitter)

2.4.4 Trade-offs

- Harder to implement
- Less intuitive for clients
- Requires consistent ordering

2.5 Pagination Metadata

A professional API does not return data alone — it also provides **pagination metadata**.

Example Response

```
{
  "data": [...],
  "pagination": {
    "limit": 10,
    "nextCursor": 138,
    "hasMore": true
  }
}
Or for offset-based pagination:
{
  "data": [...],
  "pagination": {
    "page": 2,
    "limit": 10,
    "totalItems": 125,
    "totalPages": 13
  }
}
```

Metadata allows clients to:

- build navigation controls
- detect the end of a dataset
- avoid unnecessary requests

2.6 Enforcing Limits and Validation

APIs must **never trust client input**.

Example: Enforcing Maximum Page Size

```
const MAX_LIMIT = 100;
const limit = Math.min(
  parseInt(req.query.limit) || 10,
  MAX_LIMIT
);
```

- This prevents:
  - denial-of-service attacks
  - accidental large queries
  - excessive memory usage

2.7 Pagination and Database Indexing

Pagination is tightly coupled with database performance.

Best practices:

- Always index paginated columns
- Prefer numeric or timestamp-based cursors
- Avoid paginating on non-indexed fields

Bad example:

ORDER BY title

Good example:

ORDER BY id

2.8 Pagination and Sorting

Sorting must be:

- explicit
- deterministic
- consistent across requests

- Example:

```
GET /api/posts?sort=createdAt&order=desc&limit=10
```

- Express example:

```
const sortField = req.query.sort || "createdAt";
const sortOrder = req.query.order === "asc" ? "ASC" : "DESC";
const posts = await Post.findAll({
  order: [[sortField, sortOrder]],
  limit
});
```

- Sorting without pagination is dangerous; pagination without sorting is unreliable.

2.9 Common Pagination Pitfalls

❌ **Returning unbounded datasets**

❌ **Allowing unlimited limit values**

❌ **Paginating without ordering**

❌ **Using offset pagination for large tables**

❌ **Exposing database internals to clients**

2.10 When to Use Each Pagination Strategy

Scenario	Recommended Strategy
Small datasets	Offset-based
Large datasets	Cursor-based
Frequently changing data	Cursor-based
Simple admin interfaces	Offset-based
Infinite scrolling UIs	Cursor-based

2.11 Practical Exercise

Exercise

- Implement a paginated endpoint:

`GET /api/comments`

- Requirements:
  - Support limit and cursor
  - Enforce maximum limit
  - Return pagination metadata
  - Sort by createdAt

2.12 Chapter 2 Summary

- Pagination is essential for scalable REST APIs.
- Offset-based pagination is simple but limited.
- Cursor-based pagination scales better.
- Metadata improves client experience.
- Pagination must be validated and bounded.
- Database indexing is critical for performance.
- Pagination logic belongs in the API, not the client.

Chapter 3 — Caching (Node.js & Express)

3.1 Why Caching Is Fundamental in RESTful APIs

In RESTful API design, caching is one of the **most powerful performance optimizations** available.
Unlike pagination—which limits *how much* data is returned—caching optimizes **how often** data must be recomputed or retransmitted.

- Consider the following Express route:

```
app.get("/api/categories", async (req, res) => {
  const categories = await Category.findAll();
  res.json(categories);
});
```

- If:
- categories change infrequently,
- this endpoint is called thousands of times per minute,
- then the server repeatedly:
- queries the database,
- serializes the same JSON,
- sends identical responses.

Caching avoids this redundant work and is essential for **scalable API design**.

3.2 Caching in REST: Architectural Perspective

RESTful APIs leverage **HTTP caching semantics**, which are built into the protocol itself.
This allows caching to occur at multiple levels:

Layer	Example
Client	Browser, mobile app
Network	CDN, proxy
Server	Express middleware
Application	In-memory cache, Redis

A well-designed API enables caching without sacrificing correctness or security.

3.3 HTTP Caching Fundamentals

HTTP caching relies on **response headers** that instruct clients and intermediaries how to cache responses.

Key Headers

Header	Purpose
Cache-Control	Defines caching behavior
ETag	Identifies a specific resource version
Last-Modified	Indicates last modification time
Expires	Absolute cache expiration (legacy)

Express allows full control over these headers.

3.4 Cache-Control in Express

3.4.1 Basic Cache-Control Example

```
app.get("/api/products", async (req, res) => {
  const products = await Product.findAll();
  res.set("Cache-Control", "public, max-age=60");
  res.json(products);
});
```

- This tells clients and proxies:
- the response can be cached,
- it remains valid for 60 seconds.

3.4.2 Common Cache-Control Directives

Directive	Meaning
public	Can be cached by any cache
private	Cacheable only by the client
no-cache	Must revalidate before reuse
no-store	Must not be cached
max-age	Cache lifetime in seconds

3.4.3 When Not to Cache

Do **not** cache responses that are:

- user-specific (unless carefully controlled),
- sensitive (authentication data),
- frequently changing without validation.
- Example:

```
res.set("Cache-Control", "no-store");
```

3.5 Conditional Requests with ETags

3.5.1 The Problem Conditional Requests Solve

Even when a resource has not changed, clients may repeatedly request it.
Conditional requests allow the server to respond with **no body** when cached data is still valid.

3.5.2 Using ETags in Express

```
app.get("/api/articles/:id", async (req, res) => {
  const article = await Article.findByPk(req.params.id);
  const etag = `"${article.updatedAt.getTime()}"`;
  res.set("ETag", etag);
  if (req.headers["if-none-match"] === etag) {
    return res.status(304).end();
  }
  res.json(article);
});
```

How This Works

1. Server sends ETag
2. Client stores the response
3. Client sends If-None-Match on next request
4. Server replies:
    - 304 Not Modified → no payload
    - 200 OK → resource changed

This drastically reduces bandwidth and latency.

3.6 Last-Modified Header

- An alternative to ETags:

```
res.set("Last-Modified", article.updatedAt.toUTCString());
```

- Clients send:

```
If-Modified-Since
```

- ETags are generally preferred because they are more precise.

3.7 Application-Level Caching

HTTP caching is not always sufficient.
Some responses are expensive to compute even if they cannot be cached at the HTTP level.

Example Use Cases

- Aggregation queries
- Statistics dashboards
- Reference data
- Expensive joins

3.7.1 Simple In-Memory Cache (Express)

```
const cache = new Map();

app.get("/api/stats", async (req, res) => {
  if (cache.has("stats")) {
    return res.json(cache.get("stats"));
  }
  const stats = await computeStats();
  cache.set("stats", stats);
  res.json(stats);
});
```

⚠ Limitations:

- Cache lost on restart
- Not shared across processes
- Not suitable for clusters

3.8 Distributed Caching with Redis

In real-world systems, Redis is commonly used.

- Conceptually:

```
const cached = await redis.get("stats");
if (cached) return res.json(JSON.parse(cached));
const stats = await computeStats();
await redis.setex("stats", 60, JSON.stringify(stats));
res.json(stats);
```

- Benefits:
  - shared across instances
  - survives restarts
  - supports TTLs and eviction policies

3.9 Cache Invalidation

Caching is easy — **invalidating cache correctly is hard**.

Common strategies:

- Time-based expiration (TTL)
- Invalidate on write (POST, PUT, DELETE)
- Versioned cache keys

- Example:

```
await redis.del("stats");
```

- Best practice:

Prefer **short TTLs + revalidation** over complex invalidation logic.

3.10 Caching and REST Semantics

REST guidelines:

- Cache only **GET** requests
- Invalidate cache on **state-changing requests**
- Never cache responses to:
  - POST
  - PUT
  - PATCH
  - DELETE

3.11 Caching and Pagination

Paginated responses are cacheable **per page or per cursor**.

Example cache key:

products:page=2:limit=10

Be careful:

- Cache explosion is possible
- Always limit pagination parameters

3.12 Common Caching Pitfalls

❌ **Caching sensitive data**

❌ **Forgetting cache invalidation**

❌ **Caching without TTL**

❌ **Assuming caches are shared across processes**

❌ **Over-caching rapidly changing data**

3.13 Practical Exercise

Exercise

- Implement a cached endpoint:

`GET /api/popular-posts`

- Requirements:
  - Use Cache-Control headers
  - Implement ETag validation
  - Cache results in memory or Redis
  - Invalidate cache on post creation

3.14 Chapter 3 Summary

- Caching is essential for performance and scalability.
- HTTP caching should be the first optimization.
- Cache-Control and ETags are core REST tools.
- Application-level caching complements HTTP caching.
- Cache invalidation is a critical design challenge.
- Express provides full control over caching behavior.

Chapter 4 — Rate Limiting (Node.js & Express)

4.1 Why Rate Limiting Is Essential in RESTful APIs

Rate limiting is a **protective mechanism** that controls how many requests a client can make to an API within a given time period.

Without rate limiting, an Express API is vulnerable to:

- accidental abuse (poorly written clients),
- malicious abuse (brute-force attacks),
- denial-of-service (DoS),
- unfair resource consumption by a small number of clients.

- Consider a typical Express endpoint:

```
app.post("/api/auth/login", async (req, res) => {
  // authentication logic
});
```

- If this endpoint is unprotected:
  - a client can send thousands of requests per second,
  - password brute-force attacks become feasible,
  - database and CPU resources are exhausted.

Rate limiting is therefore a **core security and availability requirement**, not an optional enhancement.

4.2 What Rate Limiting Actually Controls

Rate limiting policies typically define:

- **Who** is being limited (IP, user, API key, token)
- **How many** requests are allowed
- **Over what time window**
- **What happens when the limit is exceeded**

Example policy:

100 requests per 15 minutes per IP address

In RESTful APIs, rate limiting is most commonly applied at:

- authentication endpoints,
- public GET endpoints,
- expensive or critical operations.

4.3 Where Rate Limiting Fits in an Express Architecture

Rate limiting is usually implemented as **middleware** in Express.

- Typical flow:

```
Request →
  Rate Limiting Middleware →
    Authentication →
      Validation →
        Business Logic →
          Response
```

- This ensures that abusive requests are rejected as early as possible, minimizing resource consumption.

4.4 Common Rate Limiting Algorithms

Understanding rate limiting algorithms helps developers choose appropriate strategies.

4.4.1 Fixed Window

The simplest strategy.

Example:

- Allow 100 requests per minute
- Counter resets every minute

Characteristics:

- Easy to implement
- Can allow traffic bursts at window boundaries

Conceptual example:

Minute 1: 100 requests allowed

Minute 2: counter resets

4.4.2 Sliding Window

Improves on the fixed window by smoothing traffic.

Instead of discrete windows, requests are counted over a moving time range.

Advantages:

- More accurate
- Fewer burst issues

Disadvantages:

- More complex
- Requires more state tracking

4.4.3 Token Bucket / Leaky Bucket

Widely used in production systems.

Concept:

- Tokens are added to a bucket at a fixed rate
- Each request consumes a token
- Bursts are allowed until tokens run out

Advantages:

- Flexible
- Predictable behavior
- Smooth handling of bursts

This approach is commonly used in:

- API gateways
- CDNs
- Distributed systems

4.5 Implementing Rate Limiting in Express

4.5.1 Using express-rate-limit

- A popular middleware for Express.

```
npm install express-rate-limit
```

4.5.2 Basic Rate Limiting Example

```
const rateLimit = require("express-rate-limit");
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100,
  standardHeaders: true,
  legacyHeaders: false
});
app.use("/api/", limiter);
```

- This configuration:
  - allows 100 requests per 15 minutes per IP,
  - applies to all /api routes,
  - returns 429 Too Many Requests when exceeded.

4.6 Rate Limiting Responses and Headers

Well-designed APIs communicate rate limits clearly.

- Typical headers:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 42
X-RateLimit-Reset: 1715000000
```

- This helps clients:
  - adapt request frequency,
  - implement retry logic,
  - avoid unnecessary failures.

4.7 Customizing Rate Limiting Policies

Different endpoints often require **different limits**.

Example: Strict Limits on Login

```
const loginLimiter = rateLimit({
  windowMs: 10 * 60 * 1000,
  max: 10,
  message: "Too many login attempts. Try again later."
});
app.post("/api/auth/login", loginLimiter, loginController);
```

- This dramatically reduces brute-force risk.

Example: Role-Based Limits

- Authenticated users may have higher limits:

```
const dynamicLimiter = (req, res, next) => {
  if (req.user?.role === "admin") {
    return rateLimit({ windowMs: 60000, max: 1000 })(req, res, next);
  }
  return rateLimit({ windowMs: 60000, max: 100 })(req, res, next);
};
```

4.8 Rate Limiting and Distributed Systems

In clustered or horizontally scaled environments, **in-memory counters are insufficient**.

Problems:

- each instance has its own counters,
- limits are applied per instance, not globally.

Solution:

- shared stores such as **Redis**,
- API gateways (NGINX, Kong, Traefik),
- cloud-based rate limiting services.

Conceptually:

```
store: new RedisStore({ client: redisClient })
```

This ensures consistency across instances.

4.9 Rate Limiting and Security

Rate limiting is often combined with:

- CAPTCHA challenges,
- IP blacklisting,
- authentication checks,
- anomaly detection.

Rate limiting **reduces attack surface**, but should not be the only security measure.

4.10 Best Practices for Rate Limiting

- Apply stricter limits to authentication endpoints
- Use meaningful error messages with HTTP 429
- Communicate limits through headers
- Differentiate limits by user role or API key
- Use shared stores in distributed systems
- Monitor and log rate-limit violations
- Avoid overly aggressive limits that harm legitimate users

4.11 Common Pitfalls

❌ **Applying the same limits to all endpoints**

❌ **Not informing clients about limits**

❌ **Using in-memory limits in clustered deployments**

❌ **Blocking legitimate traffic due to poor tuning**

❌ **Forgetting to rate-limit authentication routes**

4.12 Practical Exercise

Exercise

Implement rate limiting for an Express API with the following rules:

- Public endpoints: 100 requests per 15 minutes
- Authenticated users: 500 requests per 15 minutes
- Login endpoint: 10 requests per 10 minutes
- Return proper rate-limit headers
- Use HTTP 429 on violations

4.13 Chapter 4 Summary

- Rate limiting protects APIs from abuse and overload.
- It is a fundamental part of REST API security.
- Express implements rate limiting via middleware.
- Different endpoints require different policies.
- Distributed systems require shared rate-limit storage.
- Clear communication with clients improves usability.

Chapter 5 — Integration & Best Practices (Node.js & Express)

5.1 Why Integration Matters

Pagination, caching, and rate limiting are often taught separately, but **real APIs do not operate in isolation**.
In production systems, these mechanisms must be **carefully combined** to achieve:

- scalability
- performance
- fairness
- security
- maintainability

A poorly integrated solution can:

- cache incorrect data,
- block legitimate users,
- return inconsistent paginated results,
- create subtle bugs that are hard to diagnose.

This chapter focuses on **how to design Express APIs where these three concerns reinforce each other instead of conflicting**.

5.2 A Typical Scalable Express Endpoint

Consider the following endpoint:

```
GET /api/articles?cursor=abc123&limit=10
```

In a production-ready API, this single endpoint may involve:

1. **Rate limiting** — to protect the service
2. **Pagination** — to control result size
3. **Caching** — to reduce repeated computation

- Conceptual request flow:

```
Request
  ↓
Rate Limiting Middleware
  ↓
Pagination Validation
  ↓
Cache Lookup
  ↓
Database Query (if needed)
  ↓
Cache Storage
  ↓
Response with Cache Headers
```

- Each step plays a specific role.

5.3 Order of Middleware Matters

In Express, middleware order is critical.

Recommended Order

```
app.get(
  "/api/articles",
  rateLimiter,
  validatePagination,
  cacheMiddleware,
  articlesController
);
```

Why This Order?

- **Rate limiting first** → reject abusive traffic early
- **Validation second** → prevent invalid queries
- **Caching third** → avoid unnecessary database access
- **Controller last** → execute business logic

Misordering middleware can lead to:

- caching invalid responses,
- rate limiting cached responses incorrectly,
- wasted resources.

5.4 Pagination + Caching: Design Considerations

Cache Granularity

Paginated responses must be cached **per page or per cursor**.

- Example cache keys:

```
articles:cursor=abc123:limit=10
```

```
articles:page=2:limit=20
```

Best Practices

- Enforce **maximum limits** to avoid cache explosion
- Normalize query parameters before caching
- Avoid caching unbounded queries

- Bad practice:

```
/api/articles?limit=100000
```

5.5 Pagination + Rate Limiting

Pagination reduces response size but **does not reduce request frequency**.

A client can still abuse an API by:

- requesting small pages rapidly,
- iterating through the entire dataset aggressively.

Therefore:

- pagination **must not replace** rate limiting
- both mechanisms are required

Example policy:

- 100 requests / 15 minutes
- max limit = 50

This ensures fairness and predictability.

5.6 Caching + Rate Limiting

Caching reduces server load, but it does **not eliminate the need for rate limiting**.

Why?

- Cached responses still consume network and application resources
- Attackers can overwhelm the service even with cached data

However, caching allows:

- higher rate limits for legitimate clients
- better overall throughput

In practice:

- strict limits for uncached endpoints
- more generous limits for cached GET endpoints

5.7 Example: Integrated Express Endpoint

- Conceptual Express implementation:

```
app.get(
  "/api/articles",
  rateLimiter,
  paginationMiddleware,
  cacheMiddleware,
  async (req, res) => {
    const articles = await getArticles(req.pagination);
    res
      .set("Cache-Control", "public, max-age=60")
      .json(articles);
  }
);
```

- This endpoint:
  - limits abuse,
  - returns predictable response sizes,
  - avoids redundant database queries,
  - remains responsive under load.

5.8 Common Architectural Patterns

Pattern 1 — API Gateway Offloading

In many systems:

- rate limiting is handled by an API gateway,
- caching is handled by a CDN,
- pagination remains in the application layer.

Express then focuses on **business logic**, not infrastructure.

Pattern 2 — Application-Level Control

Smaller systems may implement:

- rate limiting via Express middleware,
- caching via Redis,
- pagination via ORM queries.

This offers flexibility but requires careful design.

5.9 Monitoring and Observability

Advanced REST APIs must be observable.

Monitor:

- rate-limit violations,
- cache hit/miss ratio,
- average response time per page,
- database query performance.

Without observability:

- performance issues go unnoticed,
- limits are poorly tuned,
- caching is ineffective.

5.10 Security Considerations

Pagination, caching, and rate limiting also impact security.

Key Guidelines

- Never cache sensitive or user-specific data unless isolated
- Apply stricter rate limits to authentication endpoints
- Validate pagination parameters rigorously
- Avoid exposing internal identifiers in cursors
- Use HTTPS to protect cache headers

5.11 Common Integration Mistakes

❌ **Caching error responses**

❌ **Applying rate limiting after expensive operations**

❌ **Allowing unlimited pagination parameters**

❌ **Treating pagination as a UI concern only**

❌ **Assuming caching replaces rate limiting**

These mistakes often appear in early-stage APIs and lead to scalability failures.

5.12 Practical Integration Exercise

Exercise

Design and implement an endpoint:

GET /api/products

Requirements:

- Cursor-based pagination
- Maximum limit of 50 items
- HTTP caching with ETag support
- Rate limiting (100 requests / 15 minutes)
- Clear pagination metadata in the response

Students should justify:

- middleware order,
- caching strategy,
- chosen rate limits.

5.13 Learning Outcomes of This Chapter

After completing this chapter, students should be able to:

- Integrate pagination, caching, and rate limiting coherently
- Design scalable Express endpoints
- Reason about trade-offs between performance and protection
- Avoid common architectural pitfalls
- Build APIs suitable for real production environments

5.14 Final Summary

- Pagination controls **data volume**
- Caching controls **redundant computation**
- Rate limiting controls **request frequency**
- Together, they form the foundation of scalable REST APIs
- Express provides flexibility, but requires careful design
- Advanced RESTful API design is architectural, not cosmetic