Master the Art of REST API Design

The Ultimate Guide

ASHISH PRATAP SINGH

JAN 23, 2025 · PAID

API Design is one of the most crucial steps in **software development** and a key top of discussion in **system design interviews**.

A well-designed API allows developers to easily integrate with a system while ensuring scalability and security.

Over the years, various **API architectural styles** have emerged, including **REST, GraphQL, gRPC, Webhooks and SOAP**, each designed to address different needs.

However, **RESTful APIs** continue to dominate web development due to their simplicity, scalability, flexibility, widespread adoption and alignment with HTTP standards.

In this article, we will dive into **REST API design** covering:

- **Best practices** for building a well-structured, scalable, and secure RESTful API
- **Performance optimization techniques** to enhance API efficiency and response times.

REST

REST (**Representational State Transfer**) is an architectural style for designing web services that enable communication between clients (e.g., web browsers, mobile ap

and servers over the **HTTP protocol**.

REST uses HTTP methods (GET, POST, PUT, DELETE, etc.) to **retrieve, create, update, and delete** resources.

To build a well-designed REST API, you must first understand the fundamentals of HTTP protocol.

1. HTTP Methods (Verbs) in REST APIs

HTTP provides a set of **methods** (**verbs**) that define the type of operation to be performed on a resource.

In RESTful architectures, these methods typically map to CRUD operations:

HTTP Method	CRUD Operation	Example Use Case
GET	Read	Retrieves a resource
POST	Create	Creates a new resource
PUT	Update	Replaces or creates a resource
PATCH	Update	Partially updates a resource
DELETE	Delete	Removes a resource

It's essential to use the correct HTTP method to make your API clear and intuitive. For example, `GET` signals a read-only request to developers and should never modify server data, while `POST` indicates data creation or an action that results in a change.

2. REST is Resource-Oriented

In RESTful API design, data is represented as **resources**, and each resource is identified by a **Uniform Resource Identifier** (**URI**).

- `/books/` → A collection (or list) of books
- `/books/123` → A specific book with ID 123

3. API Endpoints

An **endpoint** is a combination of:

- An HTTP method (GET, POST, PUT etc.)

- A resource URI (`/books/`, `/users/123`)

Each endpoint represents a specific operation on a resource.

Example:

- **GET** `/books/` → Fetch all books
- **POST** `/books/` → Create a new book
- **DELETE** `/books/123` → Delete the book with ID 123

Using clear and consistent endpoints helps developers quickly understand how to interact with your API.

4. HTTP Status Codes: Understanding API Responses

Each API response includes an **HTTP status code**, which indicates the result of the request.

Using meaningful status codes is important for helping consumers of your API understand why a request might have failed and how they can fix or retry it.

Category	Range	Meaning
1xx	100-199	Informational responses
2xx	200-299	Success responses
3xx	300-399	Redirection responses
4xx	400-499	Client-side errors (bad request, unauthorized, not found)
5xx	500-599	Server-side errors (internal server error, service unavailable)

Common status codes include:

- **2xx** (**Success**): The request was successfully received and processed.
  - **200 OK**: The request succeeded.
  - **201 Created**: A new resource was successfully created.
  - **204 No Content**: The request succeeded, but there is no content to return.
- **3xx** (**Redirection**): Further action is needed to complete the request (e.g., a different endpoint or resource location).
- **4xx** (**Client Error**): There was an error in the request sent by the client.
  - **400 Bad Request**: The request was malformed or invalid.
  - **401 Unauthorized**: Authentication is required or has failed.
  - **403 Forbidden**: The client does not have permission to access the resource.

- **404 Not Found**: The requested resource does not exist.
- **429 Too Many Requests**: Rate limit exceeded.

- **5xx** (**Server Error**): The server encountered an error while processing the reque
  - **500 Internal Server Error**: A general error occurred on the server.
  - **503 Service Unavailable**: The server is currently unable to handle the reque often due to maintenance or overload.

Best Practices for Designing RESTful APIs

1. Define Clear Resource Naming Conventions

Using a **consistent, intuitive, and hierarchical structure** for API endpoints improv both readability and usability. The goal is to help developers quickly understand ho to interact with your API without extensive documentation.

a. Use Nouns, Not Verbs

Since REST is resource-oriented, focus on **objects** (nouns) rather than **actions** (verb for your endpoints. The HTTP methods (`GET`, `POST`, etc.) already describe the action using verbs in the URL are redundant.

❌ **Bad:**

```
GET /getAllUsers
POST /createNewOrder
DELETE /removeProduct/123
```

✅ **Good:**

```
GET /users
POST /orders
DELETE /products/123
```

b. Use Plural Nouns for Collections

A collection endpoint typically returns a list of resources, so plural nouns clarify th multiple items may be retrieved.

❌ **Bad:**

```
GET /user/123
GET /book/987
```

✅ **Good:**

```
GET /users/123
GET /books/987
```

Exception: If the resource is a **singleton** (e.g., `/profile` for the currently logged-in user), using singular nouns is acceptable:

```
GET /profile
```

c. Structure URLs Hierarchically

URLs should reflect the relationship between resources. Hierarchical structuring makes endpoints intuitive and logical.

❌ **Bad:**

```
GET /getUserOrders?userId=456
```

✅ **Good:**

```
GET /users/456/orders      (Retrieve all orders for user 456)
GET /users/456/orders/789  (Retrieve order 789 for user 456)
GET /stores/12/products    (Retrieve all products from store 12)
```

d. Avoid Deeply Nested Resources

Deeply nested URLs become difficult to maintain and query efficiently. Nesting should be **kept to one or two levels** to avoid complexity.

A better approach is to flatten the structure, making it easier to use and more effici

❌ **Bad:**

```
GET /users/456/orders/789/items/321 (too deeply nested)
```

✅ **Good:**

```
GET /orders/789/items
```

e. Keep URLs Short and Meaningful

Overly long URLs can be confusing and harder to work with. Aim for short urls wh maintaining clarity.

❌ **Bad:**

```
GET /getAllProductsFromDatabase
```

✅ **Good:**

```
GET /products
GET /products/456
```

f. Maintain Consistency in Naming

Inconsistent naming—whether in resource nouns, HTTP methods, or URL pattern makes APIs **confusing** for developers.

❌ **Bad (Mix of singular and plural, inconsistent casing):**

```
GET /user_profiles
GET /Products
GET /fetchUsers
```

✅ **Good:**

```
GET /users
GET /products
GET /profiles
```

2. Use HTTP Methods Correctly

One of the core principles of RESTful API design is using HTTP methods (verbs) correctly to perform operations on resources (nouns).

HTTP Method	Safe?	Idempotent?	Example
GET	✅	✅	GET /books/123
POST	❌	❌	POST /books
PUT	❌	✅	PUT /books/123
PATCH	❌	❌	PATCH /books/123
DELETE	❌	✅	DELETE /books/123

GET

- Retrieve (read) resources from the server.
- **Example**: `GET /books/123` retrieves the book with ID 123 without modifying i

POST

- Create new resources or trigger server-side actions that don't map neatly to oth methods.
- **Example**: `POST /books` creates a new book and returns it with a new ID in the response.

PUT

- Completely replace an existing resource. Some implementations also allow creating a resource at a specific URI if it doesn't exist.

- **Example**: `PUT /books/123` replaces the entire `Book 123` object. Sending the same request again yields the same result.

PATCH

- Partially update an existing resource. Only modify the fields sent in the request body.
- **Example**: `PATCH /books/123` might only change the book's title.

DELETE

- Remove a resource from the server.
- **Example**: `DELETE /books/123` removes the book with ID 123. Repeating the same call does not result in additional changes if the resource is already deleted

Safe vs. Idempotent Methods

- **Safe**: A method is *safe* if it does **not modify resources** on the server. It is strictly read-only and should not cause any side effects (beyond caching or logging).
  - **Examples**: `GET`, `HEAD`, `OPTIONS`
- **Idempotent**: A method is *idempotent* if **multiple identical requests** will result in the **same effect** on the server as a single request. Even though idempotent methods may modify resources, repeating the operation does not cumulatively change the outcome.
  - **Examples**: `GET`, `HEAD`, `PUT`, `DELETE`, `OPTIONS`

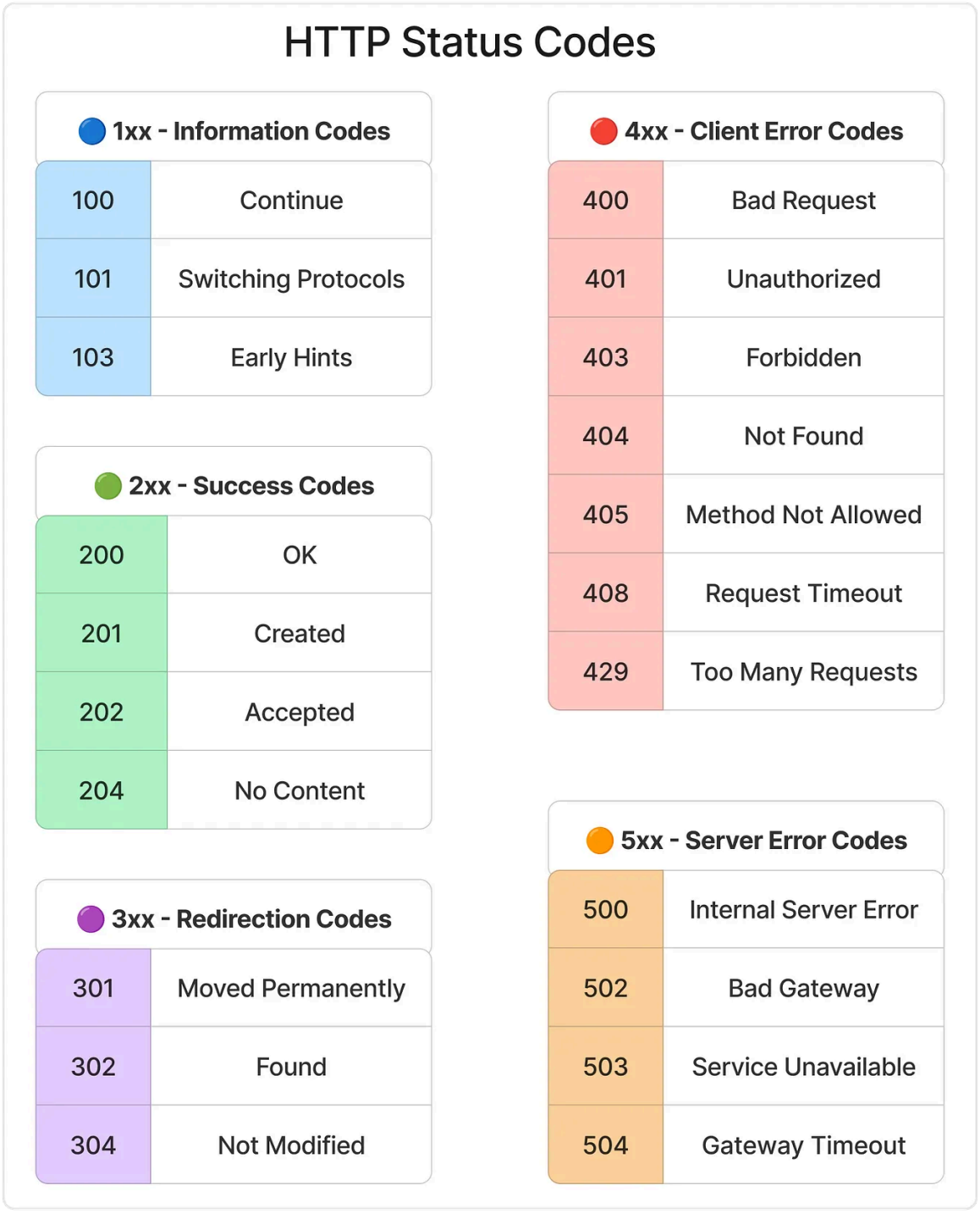
By correctly aligning HTTP methods, your API becomes more predictable, scalable and easier to integrate.

3. Use Proper Status Codes

Using the correct HTTP status codes in a RESTful API is critical for clear communication between the client and server.

Status codes help API consumers understand whether a request was successful, fail or requires further action. A well-designed API returns consistent and meaningful status codes, which improves debugging, usability, and API integration.

List of 21 most common HTTP status codes:



Use Consistent and Meaningful Error Responses

Handling errors properly in a RESTful API is as important as designing endpoints a responses.

Meaningful error messages help developers understand what went wrong, reducing confusion and improving debugging.

Avoid vague error messages. Every error response should follow a consistent JSON format.

Best Practices for Error Response:

- Use a structured JSON format for all errors.
- Use the correct HTTP status code to indicate the error type.
- Provide a clear, human-readable error messages.
- Provide examples or link to API documentation when possible.
- Avoid technical jargon or internal system messages.
- Avoid exposing stack traces or sensitive system details.
- Return Proper HTTP Headers for Error Responses.

❌ **Bad (vague) response**

```
Error: Something went wrong.
```

✅ **Good error response format**

```
{
    "error": {
        "code": 400,
        "message": "Invalid request: 'email' field is required.",
        "details": [
            {
                "field": "email",
                "issue": "Required field is missing."
            }
```

```
        ],
        "timestamp": "2025-01-22T12:00:00Z",
        "request_id": "abc123"
    }
}
```

4. Implement Pagination, Filtering, and Sortin

When dealing with **large datasets**, retrieving all records at once can lead to performance issues, high memory usage, and slow response times.

To optimize API efficiency and enhance user experience, RESTful APIs should support **pagination, filtering, and sorting** mechanisms.

This approach ensures that clients can request precisely the data they need in a controlled and efficient manner.

a. Implementing Pagination

Pagination divides large datasets into smaller **pages** or **chunks**, allowing clients to request data in manageable portions.

1. Offset-Based Pagination (Recommended for Most APIs)

Uses query parameters `limit` and `offset` to fetch subsets of data.

Example:

```
GET /products?limit=10&offset=20
```

- `limit=10` → Return 10 records per request.
- `offset=20` → Skip the first 20 records.

It's simple, intuitive and widely supported but can get slow for large datasets due to scanning.

2. Cursor-Based Pagination (Recommended for High-Performance AP

Uses a `cursor` (**e.g., ID or timestamp**) to mark the current position in the dataset. T client requests the next chunk of data by sending the cursor received from the previous response.

Example:

```
GET /products?limit=10&cursor=xyz123
```

- `limit=10` → Return 10 records per request.
- `cursor=xyz123` → Indicates the last retrieved record's position.

Cursor-based pagination is more efficient and reliable than offset-based pagination especially for real-time and large datasets but it's more complex to implement and requires the server to generate and manage stable cursors.

3. Page-Based Pagination

Uses parameters such as `page` and `size` parameters to to load a specific page of dat

Example:

```
GET /products?page=3&size=10
```

- `page=3` → Retrieves the 3rd page.
- `size=10` → Returns 10 records per page.

Best suited when working with UI elements like numbered pages (1, 2, 3, etc.).

It's not as efficient as cursor-based pagination for large, frequently changing datase and page numbers can become out of sync if data is added or removed.

b. Implementing Filtering

Filtering allows clients to narrow down results based on specific conditions, ensuri clients only receive data that meets their needs (e.g., active users, orders in a specifi date range)

Best Practices for Implementing Filtering:

- Use **query parameters** for filtering.
- Allow multiple filters to be combined (`AND` conditions) for more precise searche

```
GET /users?status=active
GET /products?status=active&category=electronics

GET /orders?start_date=2024-01-01&end_date=2024-01-31
```

c. Implementing Sorting

Sorting allows clients to define the order of returned results.

Best Practices for Sorting:

- Use **query parameters** for sorting.
  - `sort` parameter to define the field(s) to sort by.
  - `order` parameter to define ascending (`asc`) or descending (`desc`)
- Provide sensible **default sorts** (e.g., `id`, `creation_date`) for clients that do not specify any sorting.

Example: Single-Field Sorting

```
GET /products?sort=price&order=asc
```

- `sort=price` → Sort by price.
- `order=asc` → Ascending order (lowest to highest).

Example: Multi-Field Sorting

```
GET /products?sort=price,name&order=desc,asc
```

- Primary sort: `price` in `desc` (highest to lowest).
- Secondary sort: `name` in `asc` order if prices are equal.

5. Secure Your API

APIs often handle sensitive data, business logic, and back-end systems, making the common target for unauthorized access, hacking attempts, data leaks, and maliciou attacks.

Therefore, securing them is critical.

Here are some best practices to protect your API from attacks:

a. Use OAuth 2.0 / JWT for authentication

APIs should never expose sensitive resources without authentication. Start by adopting a secure token-based authentication mechanism.

- **OAuth 2.0**

- Industry-standard protocol for **access delegation**.
- Allows clients (mobile apps, web apps) to obtain limited access to user data without needing to store the user's credentials.

- **JWT (JSON Web Tokens)**
  - **Stateless**: Authentication information is self-contained in the token, reduci server overhead since no session storage is required.
  - **Portable**: Tokens can be used across multiple services or microservices.
  - **Secure**: Signed by the server's private key or a shared secret, so tampering c be detected.

Best Practices:

- Never send user credentials (e.g., username/password) directly in URLs or as qu parameters.
- Rotate and invalidate tokens periodically.
- Use **HTTPS** to prevent tokens from being intercepted.
- Avoid using **API keys** for any sensitive operations. Reserve them for simple, public, read-only endpoints if necessary.

b. Implement Role-Based Access Control (RBAC)

Not all users or clients should have the same level of API access. Implement **RBAC** restrict actions based on defined roles (e.g., Admin, User, Guest).

Best Practices:

- Define roles and permissions clearly (e.g., `Admin` can manage users, `User` can read/write their own data, `Guest` can only read public info).
- Restrict write/delete access to admins only.
- Store and verify roles in JWT tokens or by querying a secure database/service.

c. Secure API Endpoints with Rate Limiting

Rate limiting helps protect against Distributed Denial of Service (DDoS) attacks, brute-force attempts, and other abusive behaviors.

Best Practices:

1. **Set Request Limits**
    - Allow a specific number of requests per user/IP within a defined timeframe (e.g., 100 requests per minute).
    - Issue **HTTP 429** (**Too Many Requests**) if the limit is exceeded.
2. **Implement Exponential Backoff**
    - Gradually **increase the delay** between retries when a client hits rate limits or experiences repeated failures.
3. **Track Usage**
    - Monitor and log request patterns to identify potential attacks and adjust limits or block offending IPs as needed.

d. Enable CORS carefully

Cross-Origin Resource Sharing (**CORS**) controls which origins (domains) can access your API. Misconfiguration can expose your API to **cross-site request forgery** (**CSRF**) or similar attacks.

Best Practices:

- Avoid wildcard (*) origins: Whitelist specific, trusted domains.
- Set appropriate CORS headers:

```
Access-Control-Allow-Origin: https://trusted-domain.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
```

```
Access-Control-Allow-Headers: Authorization, Content-Type
```

e. Use HTTPS for Secure Communication

Always use HTTPS to encrypt data and protect against man-in-the-middle attacks.

Obtain and install a valid SSL/TLS certificate and redirect all HTTP traffic to HTTI

f. Prevent SQL & NoSQL Injection

APIs are vulnerable to **SQL Injection** attacks if user input is concatenated directly queries.

Best Practices

- **Parameterized Queries**: Use placeholders (`?`, `:$param`, etc.) to avoid injecting malicious SQL or NoSQL.
- **Input Validation & Sanitization**: Validate fields (type, length, format) and strip out suspicious characters.

❌ **Vulnerable Query:**

```
query = f"SELECT * FROM users WHERE email = '{email}'"
db.execute(query)
```

✅ **Safe Query Using Parameterized Statements:**

```
query = "SELECT * FROM users WHERE email = ?"
db.execute(query, (email,))
```

If using a NoSQL database, use built-in methods to safely **filter** or **match** data rathe than building queries by string concatenation.

6. API Versioning Strategies

APIs evolve over time. API versioning allows developers to make changes without breaking backward compatibility, ensuring smooth transitions for consumers.

a. URL versioning (Most common)

URL versioning places the version number directly in the endpoint path, making it highly visible and easy to manage.

Best Practices:

- Prefix URLs with a version number (e.g., `/v1/`, `/v2/`).
- Use whole numbers for clarity (e.g., `v1`, `v2`) instead of decimals like `v1.1`, which can cause confusion.
- Maintain older versions as long as they remain in use. Communicate deprecati timelines and provide a migration path.

Example:

```
GET /v1/users (Version 1)
GET /v2/users (Version 2)
```

Pros

- Highly visible versioning strategy.
- Easy to document and implement.
- Clear separation between versions.

Cons

- URL clutter can grow if you maintain many versions.
- Requires clients to update their integrations (endpoints) when moving between versions.

b. Header versioning

In **header-based versioning**, the client specifies the desired API version in the HTT headers. The most common approach is to use a custom or media-type header.

Example:

```
GET /users
Accept: application/vnd.api.v1+json
```

- The `Accept` header informs the server which version (`v1`) the client intends to u

Pros

- **Clean URLs**: No additional version information appears in the endpoint path.

Cons

- Less discoverable for developers who are used to seeing the version in the URL
- Requires more careful documentation and tooling to ensure clients set headers correctly.

c. Query parameter versioning

Query parameter versioning appends a version indicator to the query string.

Example:

```
GET /users?version=1
```

Pros

- Easy to implement.
- No impact on the base URL structure.

Cons

- Less common and potentially **less visible**.
- Some clients or proxies may unintentionally strip or cache query parameters in ways that interfere with versioning.

Performance Optimization Technique

1. Reduce Payload Size

A smaller payload translates to faster data transfer over the network. This is critica for users on slower connections or mobile devices.

a. Use GZIP Compression

Enable **GZIP** or **Brotli** compression on the server to shrink response sizes, especial for verbose formats like JSON.

Example (Nginx):

```
gzip on;
gzip_types application/json;
```

```
gzip_min_length 1024;
```

b. Limit Response Fields (Sparse Fieldsets)

Let clients request **only** the fields they need to minimize payload size.

Example:

```
GET /users?fields=id,name,email
```

Response:

```
{
    "id": 123,
    "name": "John Doe",
    "email": "john@example.com"
}
```

c. Use Efficient Data Formats

JSON is human-readable but can be verbose. Consider more compact, binary proto
where appropriate:

- MessagePack
- Protocol Buffers (gRPC)
- Avro

d. Minimize Unnecessary Nested Objects

Deeply nested data structures increase payload size and complexity. Instead of embedding entire sub-objects, consider separate endpoints or links to child resourc

Example:

```
{
  "id": 1,
  "name": "John Doe",
  "ordersUrl": "/users/1/orders"
}
```

This approach keeps primary responses lean and offloads additional data to separat calls as needed.

2. Optimize Database Performance

Databases are often the **biggest bottleneck** in API performance. By optimizing que and data access patterns, you can drastically reduce response times.

a. Use Indexing for Faster Queries

- **Indexes** help the database locate rows without scanning the entire table.
- Index **primary keys**, **foreign keys**, and frequently queried columns.
- Use composite indexes to improve performance when filtering by multiple columns.

b. Optimize Queries & Avoid N+1 Queries

- The **N+1 query problem** arises when one query is used to fetch a set of items, a then additional queries are executed for each item.
- Use **JOINs**, **subqueries**, or **batch fetching** (e.g., `IN` clauses) instead of separate queries in a loop.

c. Use Database Connection Pooling

- Opening new database connections repeatedly is expensive.

- Use connection pooling libraries (e.g., **HikariCP** for Java, **SQLAlchemy** pooling for Python, or **PgBouncer** for PostgreSQL) to reuse connections and minimize overhead.

d. Implement Read Replicas & Sharding

- **Read replicas** handle read-heavy workloads without overloading the primary database.
- **Sharding** distributes data across multiple databases, improving performance and scalability for very large datasets.

e. Use Caching for Frequent Queries

Cache frequently accessed data in **Redis**, **Memcached**, or in-memory stores.

Example:

```
cache_key = f"user_profile:{user_id}"
profile = redis.get(cache_key)
if not profile:
    profile = db.query("SELECT * FROM users WHERE id = ?", user_id)
    redis.set(cache_key, profile, ex=3600)  # Cache for 1 hour
```

3. Optimize Network & API Response Time

Network latency often impacts overall performance, especially when clients are geographically distant or on mobile networks.

a. Use a Content Delivery Networks (CDN)

- CDNs like **Cloudflare**, **AWS CloudFront**, or **Akamai** store cached copies of your API responses in edge locations worldwide, reducing latency for geographically dispersed users.

b. Implement HTTP/2 & HTTP/3

- **HTTP/2** supports multiplexing multiple requests over a single connection, reducing overhead.
- **HTTP/3** (**QUIC**) further improves latency and reliability, especially on mobile networks.

c. Reduce DNS Lookups & Optimize TLS Handshakes

- **Minimize redirects** (301, 302) to avoid extra DNS or TCP handshakes.
- **Optimize TLS handshakes** by using modern protocols and ciphers; keep TLS certificates updated.

d. Enable Keep-Alive Connections

Persistent connections reduce the overhead of creating a new connection for each request.

Example (**Nginx**):

```
keepalive_timeout 75;
```

e. Reduce Unnecessary HTTP Headers

- Trim large or redundant headers to **minimize network overhead**.
- For instance, limit `Accept-Encoding` to only supported compression methods.

4. Enable Caching

Caching is one of the most effective ways to **reduce server load** and **speed up responses**.

a. Client-Side Caching

Use **Cache-Control** headers to specify how long resources can be cached:

```
Cache-Control: max-age=3600, public
```

Include **ETags** to enable conditional requests and prevent sending unchanged data:

```
ETag: "abc123"
```

b. Application-Level Caching

- Store **API responses** in Redis or Memcached.
- Use an **invalidation strategy** when underlying data changes to keep your cache accurate.

5. Asynchronous Processing

Long-running operations can block or slow API responses. Decoupling them can significantly **improve throughput** and **user experience**.

a. Move Heavy Operations to Background Jobs

- Use **message queues** (e.g., RabbitMQ, Apache Kafka, or AWS SQS) to offload intensive tasks.
- Return a **202 Accepted** response for lengthy processes and update the client wh the task completes.

b. Use WebSockets for Real-Time Communication

Instead of **polling** the server, establish a **WebSocket** connection (`ws://` or `wss://`) push-based updates in real-time.

c. Implement Asynchronous API Responses

- If a request may take significant time (e.g., large file processing), return quickly with **202 Accepted** and handle processing asynchronously.
- Notify the client via **webhooks**, **email**, or other means once completed.

d. Use Batch Processing

Combine multiple small operations into a **single bulk request**.

Example: `POST /users/batch_update`

```
[
    {"id": 1, "status": "active"},
    {"id": 2, "status": "inactive"}
]
```

This reduces the overhead of multiple network calls and database transactions.

Hope you enjoyed reading this article.

If you found it valuable, hit a like ❤️ and if you have any questions or suggestions, leave a comment.

115 Likes · 4 Restacks

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...

Madhavi Jan 25

Liked by Ashish Pratap Singh

What factors would you consider when determining the extent of backward compatibility suppor incase of multiple versions?

LIKE (1) REPLY

3 replies by Ashish Pratap Singh and others

3 more comments...