ISSN: 2754-5008

Journal of Pharmaceutical Research and Reports

SCIENTIFIC Research and Community

Review Article

Open Access

Building Scalable and Performant Apis: Best Practices and Patterns

Vamsi Thatikonda

Senior Software Engineer, Snoqualmie, WA

ABSTRACT

As APIs become critical business infrastructure, they need to be highly performant and scalable to provide responsive experiences. This article outlines key best practices and patterns for optimizing API performance and scale. Techniques covered include caching, database optimization, horizontal scaling, asynchronous queuing, monitoring, and RESTful design principles. Following API design best practices allows companies to handle increasing workloads and deliver seamless digital experiences.

***Corresponding author**

Vamsi Thatikonda, Senior Software Engineer, Snoqualmie, WA.

Received: January 05, 2022; **Accepted:** January 11, 2022; **Published:** January 19, 2022

Keywords: Api Design, Scalability, Performance, Caching, Databases, Scaling, Asynchronous Processing, Queuing, Monitoring, Rest, Best Practices

Introduction

As applications continue to be built as composites of different services and APIs, having performant and scalable APIs becomes critically important. Slow and unreliable APIs can create bottlenecks that degrade end user experiences and impact business metrics. Additionally, APIs need to scale elegantly to handle varying workloads and spikes in traffic. This article will provide an overview of key best practices and patterns for building APIs that are high-performance and can scale gracefully.

Leverage Caching and CDNs

Caching and content delivery networks (CDNs) are two powerful techniques for improving API performance and scalability. Caching involves storing API responses and serving cached responses when the same request comes in again. This avoids unnecessary processing and database loads. CDNs work by distributing caching servers globally so that API requests can be served from edge locations that are closest to the user.

Best Practices Around Caching

- Caching requests that are read heavy and don't frequently change. CRUD APIs are good candidates where GET requests can be cached.
- Setting cache control headers like Cache-Control and Expires to inform clients how long to cache responses [1,2].
- Structuring cache keys intelligently usually by including the URL and request parameters.
- Configuring cache expiration times carefully based on how often data is updated.
- Enabling caching in the CDN configuration at the edge locations.

A well-designed caching strategy can "increase throughput by orders of magnitude while simultaneously decreasing costs"21]. CDNs like CloudFront can reduce latency by serving traffic from edge locations globally.

Optimize Database Queries

APIs ultimately rely on database queries to serve requests. Optimizing the database queries and access patterns is critical for performant APIs. Some key optimization techniques include:

- Adding indexes on fields that are used to filter and lookup data frequently.
- Avoiding expensive joins by de-normalizing data judiciously.
- Batching read/write operations to reduce database round trips.
- Using database query caching to avoid re-running identical queries.
- Paginating large requests to reduce load rather than returning huge result sets.
- Using read-only replicas to distribute read traffic

Database access optimizations combined with caching allows APIs to return data faster and handle more throughput [3].

Horizontally Scale at Multiple Layers

To handle increasing traffic, APIs need to scale out at multiple levels horizontally. This includes:

- Running API servers behind a load balancer to distribute requests.
- Scaling out databases using sharding and read replicas.
- Having multiple cache servers to distribute cache loads.
- Scaling up CDNs to handle traffic spikes globally.

Autoscaling groups can then be configured to spin up additional instances as needed [4]. Services like AWS can automate this scaling. The key is to identify and benchmark scaling bottlenecks proactively.

Use Asynchronous Design and Queuing

Synchronous request/response APIs can result in delays and poor performance. Introducing asynchronous queues and workers can improve speed and reliability. Some good use cases include:

- Slow operations like file processing that can happen in the background.
- Sending emails/notifications that don't need real-time responses.
- Tasks that need to be retried in case of failure.

Tools like RabbitMQ and Kafka provide reliable queuing and pub/sub for event-driven APIs. Consumers can then scale independently to handle load [6].

Monitor and Measure API Performance

To identify bottlenecks and areas for optimization, API performance needs to be monitored closely. Key metrics to measure include:

- Latency for different request types.
- Error rates and failed requests.
- Throughput in requests per second.
- Traffic spikes and volume patterns.

Load testing tools can simulate real-world traffic early. Production monitoring using tools like New Relic provides code-level insights [4]. This helps benchmark and continually improve API speed and scalability.

Adopt Restful Design Best Practices

RESTful design principles encourage scalable and flexible API architectures:

- Resources are uniquely identified by URIs.
- Stateless - no client session state is stored on servers.
- Responses contain enough information for clients to process them.
- Standard HTTP methods are used like GET, POST, PUT, DELETE.

RESTful APIs scale well as adding more servers doesn't affect the client integration. Caching is simpler with unique URIs and stateless servers. REST principles encourage scalable designs [5].

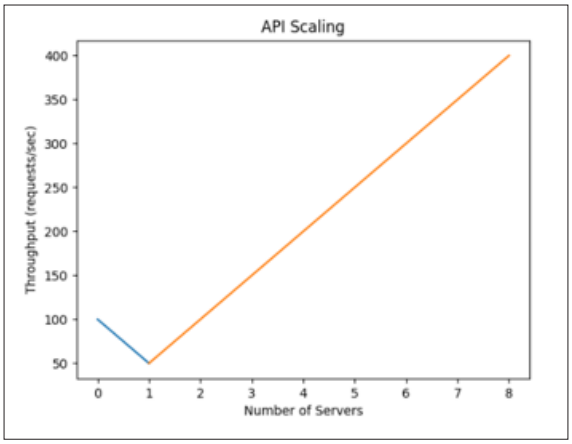
API Performance Optimization Techniques

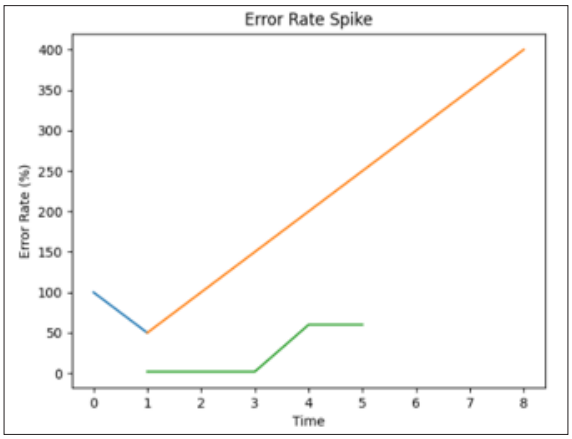
Here are the common performance optimization techniques, see table 1 and figure 1 for results

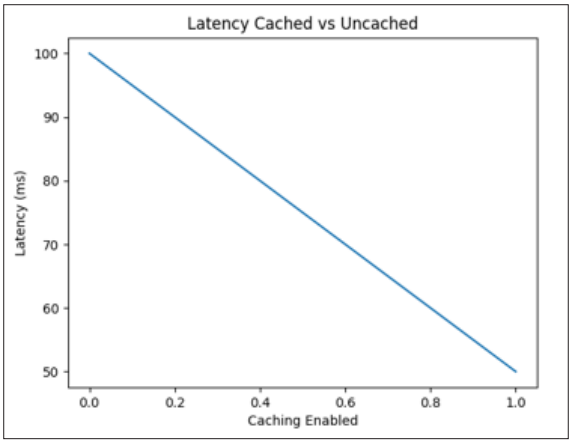
- Caching with CDNs, Redis, or Memcached is a great way to improve response time for data that does not change frequently. Caching serves cached responses instead of repeating expensive database or computation operations.
- Database query optimization involves techniques like adding indexes on commonly filtered fields to make lookups faster, de-normalizing data to avoid expensive joins, and batching reads/writes to reduce database round trips. This improves efficiency of data access.
- Horizontal scaling via load balancers and auto-scaling groups allows the API to handle higher traffic volumes and throughput by distributing load across more servers. This prevents individual servers from becoming bottlenecks.
- Asynchronous processing with queues and pub/sub enables non-critical long-running tasks to happen in the background freeing up resources for core requests. It also provides reliability with message retries.
- Monitoring tools give visibility into real-time and historical API performance metrics like latency, errors, and throughput. This helps identify issues and bottlenecks needing optimization.

Table 1: Caching Techniques

Technique	Purpose	Tools/Technology
Caching	Improve response time for cached data	CDNs, Redis, Memcached
Database Query Optimization	Improve database read/write efficiency	Indexing, SQL performance tuning
Horizontal Scaling	Handle increasing traffic and load	Load balancers, auto-scaling groups
Asynchronous Processing	Improve responsiveness for slow tasks	Message queues, Pub/Sub
Monitoring	Identify performance bottlenecks	New Relic, AppDynamics







Conclusion

Building scalable and performant APIs requires optimizing caching, database access, horizontal scaling, and monitoring. Leveraging a synchronicity and RESTful design also improves

speed and reliability. Careful capacity planning and benchmarking ensures API infrastructures sustain high throughput. By following API design best practices, organizations can deliver seamless experiences and gain competitive advantage.

References

1. Aldrich A (2018) Best Practices for Using CORS. https://www.moesif.com/blog/technical/cors/Authoritative-Guide-to-CORS-Cross-Origin-Resource-Sharing-for-REST-APIs/
2. Masse M (2011) REST API Design Rulebook. O'Reilly Media 1-114. https://www.oreilly.com/library/view/rest-api-design/9781449317904/
3. Narkhede S (2018) Designing High-Performance, Scalable and Resilient APIs. https://ionutbalosin.com/training/designing-high-performance-scalable-resilient-applications/
4. Rabl T (2019) Your API Is Not Performant Unless It Can Scale. https://tyk.io/api-is-not-performant-unless-it-can-scale/
5. Richardson C (2018) Pattern: Hypermedia as the Engine of Application State. https://restfulapi.net/hateoas/
6. To K (2018) How to Manage Asynchronous Workflows with a Queue. https://blog.codeship.com/manage-asynchronous-workflows-queue/