Caching strategies (Redis, Memcached) in nodejs

Caching in Node.js applications using in-memory data stores like Redis and Memcached can significantly improve performance and scalability by reducing the load on your primary data sources (e.g., databases).

General Caching Strategy:

This is a common strategy where the application first checks if the requested data exists in the cache.

If found (cache hit), the data is returned directly from the cache.

If not found (cache miss), the application fetches the data from the primary data source, stores it in the cache for future requests, and then returns the data.

In this strategy, data is written to both the cache and the primary data source simultaneously. This ensures data consistency but can introduce latency during write operations.

Implementing with Redis in Node.js:

Redis is a versatile in-memory data store that offers more advanced features than Memcached, including persistence, various data structures (strings, hashes, lists, sets, sorted sets), and Pub/Sub messaging.

Installation: Install the `redis` package:

Code

```
npm install redis
```

Client Setup: Create a Redis client instance:

JavaScript

```
const redis = require('redis');
const redisClient = redis.createClient();

redisClient.on('error', (err) => console.log('Redis Client Error', err));
redisClient.connect();
```

Caching Data.

JavaScript

```
async function getCachedData(key, fetchDataFunction) {
    let data = await redisClient.get(key);
    if (data) {
        return JSON.parse(data); // Assuming data is stored as JSON string
    } else {
        data = await fetchDataFunction(); // Fetch from database or API
        await redisClient.set(key, JSON.stringify(data), {
            EX: 3600 // Set expiration in seconds (e.g., 1 hour)
        });
        return data;
    }
}
```

Implementing with Memcached in Node.js:

Memcached is a simpler, high-performance distributed memory caching system ideal for basic key-value storage.

Installation: Install the `memjs` package:

Code

```
npm install memjs
```

Client Setup: Create a Memcached client instance:

JavaScript

```
const Memcached = require('memjs').Client;
const memcachedClient = Memcached.create('localhost:11211'); // Adjust host and port
```

Caching Data.

JavaScript

```
async function getCachedData(key, fetchDataFunction) {
    const { value } = await memcachedClient.get(key);
    if (value) {
        return JSON.parse(value.toString());
    } else {
        const data = await fetchDataFunction();
        await memcachedClient.set(key, JSON.stringify(data), { expires: 3600 }); //
Expiration in seconds
        return data;
    }
}
```

Choosing Between Redis and Memcached:

Prefer Redis for more complex caching needs, including advanced data structures, persistence, and features like Pub/Sub.

Choose Memcached for simpler, high-performance key-value caching where advanced features are not required. It is often favored for its simplicity and ease of scaling for basic caching.

Best Practices:

Set appropriate TTLs (Time-To-Live): Prevent stale data by setting expiration times for cached items.

Monitor cache performance: Track cache hit rates and memory usage to optimize your caching strategy.

Implement cache invalidation: Ensure data consistency when the primary data source is updated.

Avoid over-caching: Cache only frequently accessed and relatively static data to prevent excessive memory consumption.

Handle cache misses gracefully: Design your application to fetch data from the primary source when a cache miss occurs

---

Revision #2
Created 29 October 2025 02:43:33 by AI API
Updated 19 November 2025 05:28:13 by AI Channel