**Read the below blog or go this**

**link: https://nitin-saini.hashnode.dev/implementing-high-performance-caching-with-redis-aws-and-nodejs**

**Implementing High-Performance Caching with Redis, AWS, and Node.js**

In the world of application development, speed is everything. Users expect instantaneous responses, and any delay can lead to a poor experience. This is where **Redis** comes in.

Redis, which stands for **Remote Dictionary Server**, is an open-source, in-memory data store that's often used as a database, cache, and message broker. Unlike traditional databases that store data on a disk, Redis stores all its data in a system's **Random-Access Memory (RAM)**. This fundamental difference is the secret to its incredible speed.

**Why Use Redis?**

You might be asking, "Why not just use a regular database?" Traditional databases like PostgreSQL or MongoDB are excellent for permanent, long-term data storage. However, they're built to be durable and reliable, which often means they're slower because they have to perform disk I/O operations.

Redis complements these traditional databases perfectly by serving as an ultra-fast caching layer. By storing frequently accessed data in Redis, your application can retrieve it in nanoseconds, bypassing the need to query the slower, disk-based database. This significantly reduces server load and boosts application performance.

Here are some key use cases for Redis:

* **Caching**: This is the most common use case. Redis stores copies of frequently accessed data to serve requests faster.
* **Session Management**: Storing user session data (like login information or shopping cart contents) in Redis ensures quick access and provides built-in expiration support.
* **Real-time Analytics**: Redis's speed makes it perfect for tracking things like website clicks, page views, and API calls in real-time.
* **Message Broker**: With its Pub/Sub (Publish/Subscribe) messaging pattern, Redis can facilitate high-performance chat and messaging services.
* **Rate Limiting**: You can use Redis to easily implement rate limiting by tracking a user's requests and setting an expiration on the counter.

**Installing Redis on an AWS EC2 Instance**

Instead of running Redis locally, let's set it up on a dedicated EC2 instance to simulate a production environment. This ensures that Redis's performance isn't affected by other processes on your local machine.

1. **Launch an EC2 Instance**: Log into your AWS console and launch a new EC2 instance. A **t2.micro** instance running **Amazon Linux or Ubuntu** is a good, free-tier-eligible option for this demo. We’ll use Ubuntu here. Make sure to configure the security group to allow inbound traffic on port **6379**, which is the default Redis port, from your IP address or a specific range.

A screenshot of a computer

AI-generated content may be incorrect.

 **SSH into the Instance**: Once the instance is running, connect to it using SSH.

 **Install Redis**: Now, install Redis from the source. Run the following commands one by one:

Bash

 sudo apt-get install lsb-release curl gpg

curl -fsSL https://packages.redis.io/gpg | sudo gpg --dearmor -o /usr/share/keyrings/redis-archive-keyring.gpg

sudo chmod 644 /usr/share/keyrings/redis-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/redis-archive-keyring.gpg] https://packages.redis.io/deb $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/redis.list

sudo apt-get update

sudo apt-get install redis

This downloads, compiles, and installs Redis.

 Now, run this command EC2 instance: sudo nano /etc/redis/redis.conf

 bind 0.0.0.0

protected-mode no

 Above command will Edit the Redis Config and allows Redis to Accept incoming connections on all available network interfaces.

This means Redis only listens to localhost (loopback), so it will only accept connections from the same machine. Your remote backend (e.g., on Render or Vercel) cannot access it.

 Run this command in EC2 instance:

sudo systemctl restart redis

sudo systemctl status redis

You’ll see something like this. This means Redis service is running fine.

A screen shot of a computer

AI-generated content may be incorrect.

**Run Redis Server**: Go to the Redis CLI. By default, it'll run on port 6379. You can now connect to it using the redis-cli tool to verify it's working:

1. redis-cli
2. > PING
3. PONG
4. If you get a PONG response, you're good to go!

**Building a Node.js Application**

Let's create a simple Node.js application that uses Redis to cache a users posts and by hitting APIs, you'll see how Redis speeds up the process.

Step 1: First, initialize a new Node.js project and install the necessary dependencies:

npm init -y

npm install express ioredis nodemon axios

Now, create a file named server.js and add the following code. This server will have an endpoint that either fetches users from a mock API ("slow" database) or, if they exist, from the Redis cache.

**Server.js File**

const express = require("express");

const Redis = require("ioredis");

const axios = require("axios");

const app = express();

const PORT = 3000;

// Redis connection (change host if running inside EC2 itself)

const redis = new Redis({

host: "<your-ec2-ip>", // Use '127.0.0.1' if running on EC2 itself

port: 6379,

//password: "YourSecurePassword123!", // Set in redis.conf

// optional: enable TLS if needed (for production)

});

app.get("/user/:id", async (req, res) => {

const userId = req.params.id;

try {

// Try fetching from Redis cache

const cacheKey = `user:${userId}`;

const cachedData = await redis.get(cacheKey);

if (cachedData) {

console.log("Cache hit ✅");

return res.json({ from: "cache", data: JSON.parse(cachedData) });

}

// Simulate external API

console.log("Cache miss - Fetching from API...");

const { data } = await axios.get(

`https://fakestoreapi.in/api/users/${userId}`

);

// Store in Redis with TTL (60 seconds)

await redis.set(cacheKey, JSON.stringify(data), "EX", 86400);

res.json({ from: "API", data });

} catch (err) {

console.error(err);

res.status(500).json({ error: "Internal server error" });

}

});

// Route to clear cache manually

app.get("/clear-cache/:id", async (req, res) => {

const userId = req.params.id;

const cacheKey = `user:${userId}`;

await redis.del(cacheKey);

res.json({ message: `Cache cleared for user ${userId}` });

});

app.listen(PORT, () => {

console.log(` Server is running on http://localhost:${PORT}`);

});

Step 2: Run the Node.js server. (Make sure to edit package.json for nodemon config)

npm run dev

Step 3: Open Postman and hit the following GET API.

http://localhost:3000/user/1

We’ll see, it has taken over 2 seconds to get the response

A screenshot of a computer

AI-generated content may be incorrect.

Cache miss - Fetching from API...

If you go to EC2 instance, and run the this command: KEYS \*, you’ll see that our user:1 key has been set.

127.0.0.1:6379> KEYS \*

1) "user:1"

Now, again hit the same API. It will take less time, because this time, cache has been hit.

A screenshot of a computer

AI-generated content may be incorrect.

Cache hit ✅

Thus, you can make use of Redis cache.

**Security Suggestions:**

* Never open port 6379 to 0.0.0.0/0 in production
* Never send plaintext Redis password over the public internet
* Never run Redis without a password if accepting external traffic
* Use **stunnel** or **Nginx TLS Proxy** to encrypt Redis traffic
* Set Redis Password on EC2

**Wrap Up**

Through our hands-on example with a Node.js back-end using ioredis, we demonstrated how simple it is to implement a robust caching strategy. We saw firsthand how a cache miss results in a slower, simulated database call, while a subsequent cache hit provides an almost instantaneous response.

This caching pattern is not just for simple data; it's a fundamental architectural decision that enables applications to scale efficiently and handle high traffic without compromising on performance.