

# Orders Redis Middleware Demo — Enhanced (Node.js + Express)

---

*Enhanced demo: Redis cache middleware + cleaner key strategy, TTL tuning, and optional improvements if present*

## Index

- 1. Project Overview
- 2. Enhancements (What's improved)
- 3. Learning Outcomes
- 4. Tech Stack
- 5. Folder Structure
- 6. Redis Middleware Design
- 7. Cache Key + TTL Strategy
- 8. API Endpoints
- 9. Run + Quick Tests
- 10. Common Mistakes
- 11. Debugging Techniques
- Appendix A: Full Source Code

## 1. Project Overview

This application demonstrates Redis caching implemented as Express middleware for Orders APIs. The middleware checks Redis before executing the route handler. On hit, it returns cached JSON; on miss, it runs the handler and stores the response in Redis with a TTL.

## 2. Enhancements (What's improved)

This version focuses on cleaner middleware design, safer caching rules, and clearer logging.

## 3. Learning Outcomes

- Build reusable Redis cache middleware in Express.
- Create deterministic cache keys (path + sorted query params).
- Apply TTL and reason about staleness.
- Keep API available with fail-open behavior when Redis is down.
- Validate caching with logs and redis-cli.

## 4. Tech Stack

package.json:

```
{
  "name": "orders-redis-middleware-demo",
  "version": "1.0.0",
  "type": "commonjs",
  "scripts": {
    "start": "node server.js",
    "worker:sub": "node workers/orderEventSubscriber.js",
    "worker:jobs": "node workers/jobsWorker.js"
  },
  "dependencies": {
    "express": "^4.19.2",
    "mongoose": "^8.5.2",
    "redis": "^4.6.13"
  }
}
```

## 5. Folder Structure

```
- package.json
- server.js
  - cacheMiddleware.js
  - rateLimit.js
  - redisClient.js
  - Order.js
  - orders.js
  - leaderboard.js
```

```
- pubsub.js  
- queue.js  
- jobsWorker.js  
- orderEventSubscriber.js
```

## 6. Redis Middleware Design

### Cache-aside flow:

- 1) Compute cacheKey
- 2) Redis GET
- 3) HIT → return cached JSON
- 4) MISS → run handler → Redis SET + TTL → return

### Rules:

- Cache only GET by default.
- Cache only successful responses (e.g., 200).
- Never cache user-private data with a shared key.
- Fail-open on Redis error.

### Middleware files detected:

- middleware/cacheMiddleware.js
- middleware/rateLimit.js
- middleware/redisClient.js

### Key snippets found:

#### package.json

```
{  
  "name": "orders-redis-middleware-demo",  
  "version": "1.0.0",  
  "type": "commonjs",  
  "scripts": {  
    "start": "node server.js",
```

#### server.js

```
app.set("trust proxy", 1);  
  
app.use(express.json());  
  
// Apply Redis-backed rate limiting to all routes (demo values)  
app.use(rateLimit({ windowSeconds: 60, maxRequests: 30 }));  
  
mongoose.connect("mongodb://localhost:27017/orders_redis_middleware_demo")  
  .then(() => console.log("MongoDB connected"))
```

### **middleware/cacheMiddleware.js**

```
const redis = require("./redisClient");

// Redis cache middleware for GET /orders/:id
module.exports = async function cacheMiddleware(req, res, next) {
  const orderId = req.params.id;
```

### **middleware/rateLimit.js**

```
const redis = require("./redisClient");

/**
 * Simple Redis-backed rate limiter (fixed window).
 * - Keyed by IP + route
```

### **middleware/redisClient.js**

```
const { createClient } = require("redis");

const client = createClient();

client.on("error", (err) => console.error("Redis Client Error", err));
```

### **routes/orders.js**

```
const express = require("express");
const Order = require("../models/Order");
const redis = require("../middleware/redisClient");
const cacheMiddleware = require("../middleware/cacheMiddleware");

const { publish } = require("../services/pubsub");
const { enqueue } = require("../services/queue");
```

### **services/leaderboard.js**

```
const redis = require("../middleware/redisClient");

const LB_KEY = "leaderboard:top_orders";

/**
```

### **services/pubsub.js**

```
const redis = require("../middleware/redisClient");

const CHANNEL = "orders.events";

async function publish(event) {
```

### **services/queue.js**

```
const redis = require("../middleware/redisClient");

const QUEUE_KEY = "jobs";
```

```

/**
 * Run: node workers/jobsWorker.js
 *
 * Waits forever for jobs in list "jobs". Uses BRPOP jobs 0 (blocking).
 */
const { createClient } = require("redis");

const QUEUE_KEY = "jobs";

async function main() {

```

## 7. Cache Key + TTL Strategy

### Key format (recommended):

cache:<resource>:<path>?<sortedqueryParams>  
Example: cache:orders:/orders?page=2&status=PAID

### TTL tips:

- Use small TTL in demos (30–120s) so students can observe refresh.
- For lists, TTL + later invalidation after writes is the usual next addon.
- Consider TTL jitter to reduce stampede (advanced).

## 8. API Endpoints

Base URL: <http://localhost:3000> (or as configured)

Method	Path	Source File
GET	/orders/leaderboard/top	routes/orders.js
POST	/orders/	routes/orders.js
GET	/orders/:id	routes/orders.js
PUT	/orders/:id	routes/orders.js

## 9. Run + Quick Tests

### Start Redis (Docker):

```
docker run -d --name redis-demo -p 6379:6379 redis
```

### Run API:

```
npm install
```

```
npm run dev
```

### Cache hit test:

```
curl -i http://localhost:3000/orders/leaderboard/top  
curl -i http://localhost:3000/orders/leaderboard/top  
# check server logs for HIT/MISS
```

## 10. Common Mistakes

### Unstable cache keys

If key changes per request, you never get hits.

### Not sorting query params

Same request with different param order creates different keys.

### Caching errors

Don't cache 500s; cache only successful responses.

### Caching private data with shared keys

Risk of leaking one user's data to another.

### No invalidation plan

Lists can go stale after writes; TTL helps but invalidation is next.

## 11. Debugging Techniques

- Log cacheKey + HIT/MISS.
- Use redis-cli: GET <key>, TTL <key>, DEL <key>, KEYS \* (only in demo).
- If no hits: confirm key generation is deterministic + Redis connection OK.
- If stale data: reduce TTL; later add invalidation for write endpoints.
- Watch key count: too many keys = keyspace explosion.

## Appendix A: Full Source Code

### package.json

```
{  
  "name": "orders-redis-middleware-demo",  
  "version": "1.0.0",  
  "type": "commonjs",  
  "scripts": {  
    "start": "node server.js",  
    "worker:sub": "node workers/orderEventSubscriber.js",  
    "worker:jobs": "node workers/jobsWorker.js"  
  },  
  "dependencies": {  
    "express": "^4.19.2",  
    "mongoose": "^8.5.2",  
    "redis": "^4.6.13"  
  }  
}
```

### server.js

```
const express = require("express");  
const mongoose = require("mongoose");  
const orderRoutes = require("./routes/orders");  
const rateLimit = require("./middleware/rateLimit");  
  
const app = express();  
  
// If behind a proxy/load balancer (common in production), this helps req.ip  
// be correct.  
app.set("trust proxy", 1);  
  
app.use(express.json());  
  
// Apply Redis-backed rate limiting to all routes (demo values)  
app.use(rateLimit({ windowSeconds: 60, maxRequests: 30 }));  
  
mongoose.connect("mongodb://localhost:27017/orders_redis_middleware_demo")  
  .then(() => console.log("MongoDB connected"))  
  .catch(err => console.error(err));  
  
app.use("/orders", orderRoutes);  
  
app.listen(3000, () => {  
  console.log("Orders Redis Middleware Demo running on  
http://localhost:3000");  
  console.log("New features: rate limiting, pub/sub events, jobs queue,  
leaderboard.");  
  console.log("Try: GET /orders/leaderboard/top?limit=5");  
});
```

## middleware/cacheMiddleware.js

```
const redis = require("./redisClient");

// Redis cache middleware for GET /orders/:id
module.exports = async function cacheMiddleware(req, res, next) {
  const orderId = req.params.id;
  if (!orderId) return next();

  const cacheKey = `order:${orderId}`;
  const cachedData = await redis.get(cacheKey);

  if (cachedData) {
    return res.json({
      source: "cache",
      order: JSON.parse(cachedData)
    });
  }

  req.cacheKey = cacheKey;
  next();
};
```

## middleware/rateLimit.js

```
const redis = require("./redisClient");

/**
 * Simple Redis-backed rate limiter (fixed window).
 * - Keyed by IP + route
 * - INCR + EXPIRE
 * This is intentionally simple for demo purposes.
*/
module.exports = function rateLimit(options = {}) {
  const windowSeconds = Number(options.windowSeconds ?? 60);
  const maxRequests = Number(options.maxRequests ?? 30);

  return async function rateLimitMiddleware(req, res, next) {
    try {
      const ip = req.ip || req.connection?.remoteAddress || "unknown";
      const route = req.baseUrl + (req.path || "");
      const key = `rl:${ip}:${route}`;

      const current = await redis.incr(key);
      if (current === 1) {
        await redis.expire(key, windowSeconds);
      }

      if (current > maxRequests) {
        const ttl = await redis.ttl(key);
        return res.status(429).json({
          error: "Too many requests",
          hint: `Try again in ${ttl >= 0 ? ttl : windowSeconds} seconds`,
          windowSeconds,
        });
      }
    } catch (err) {
      console.error(`Rate limit middleware error: ${err.message}`);
      next(err);
    }
  };
};
```

```

        maxRequests
    });
}

next();
} catch (err) {
// If Redis is down, we don't want the whole API to stop.
// Allow request through (fail-open) for this demo.
console.error("Rate limiter error:", err.message);
next();
}
};

}
;

```

## middleware/redisClient.js

```

const { createClient } = require("redis");

const client = createClient();

client.on("error", (err) => console.error("Redis Client Error", err));

(async () => {
if (!client.isOpen) {
await client.connect();
console.log("Redis connected");
}
})();

module.exports = client;

```

## models/Order.js

```

const mongoose = require("mongoose");

const OrderSchema = new mongoose.Schema(
{
    product: String,
    amount: Number,
    status: String
},
{ timestamps: true }
);

module.exports = mongoose.model("Order", OrderSchema);

```

## routes/orders.js

```

const express = require("express");
const Order = require("../models/Order");
const redis = require("../middleware/redisClient");
const cacheMiddleware = require("../middleware/cacheMiddleware");

```

```

const { publish } = require("../services/pubsub");
const { enqueue } = require("../services/queue");
const { upsertOrderScore, topOrders } = require("../services/leaderboard");

const router = express.Router();

/**
 * Leaderboard endpoint:
 * GET /orders/leaderboard/top?limit=5
 * Returns top orders by amount (score) using Redis Sorted Set.
 *
 * NOTE: This route must appear BEFORE "/:id" routes, otherwise Express
 * will treat "leaderboard" as an :id param.
 */
router.get("/leaderboard/top", async (req, res) => {
  const limit = Number(req.query.limit || 10);
  const top = await topOrders(limit);

  // Optional: hydrate with Mongo order docs (kept minimal)
  const orderIds = top.map(t => t.orderId);
  const orders = await Order.find({ _id: { $in: orderIds } }).lean();

  const.byId = new Map(orders.map(o => [String(o._id), o]));
  const leaderboard = top.map((t, idx) => ({
    rank: idx + 1,
    amount: t.amount,
    order: byId.get(String(t.orderId)) || { _id: t.orderId }
  }));

  res.json({ source: "redis-zset", leaderboard });
});

/**
 * CREATE ORDER
 * - DB write (Mongo)
 * - Cache invalidation (order:<id>)
 * - Pub/Sub event (orders.events)
 * - Queue job (jobs)
 * - Leaderboard update (sorted set)
 */
router.post("/", async (req, res) => {
  const order = await Order.create({
    product: req.body.product,
    amount: req.body.amount,
    status: "CREATED"
  });

  // Invalidate cache for this order (if present)
  await redis.del(`order:${order._id}`);

  // Pub/Sub: notify "order created"
  await publish({
    type: "ORDER_CREATED",
    orderId: String(order._id),
  });
}

```

```

        product: order.product,
        amount: order.amount,
        at: new Date().toISOString()
    });

    // Queue: add a background job (example: send email)
    await enqueue(`email:order_created:${order._id}`);

    // Leaderboard: store by order amount
    await upsertOrderScore(order._id, order.amount);

    res.status(201).json(order);
});

// READ ORDER (Redis middleware cache)
router.get("/:id", cacheMiddleware, async (req, res) => {
    const order = await Order.findById(req.params.id);
    if (!order) return res.status(404).json({ error: "Order not found" });

    await redis.set(req.cacheKey, JSON.stringify(order), {
        EX: 60
    });

    res.json({
        source: "database",
        order
    });
});

/***
 * UPDATE ORDER
 * - DB update (Mongo)
 * - Cache invalidation
 * - Pub/Sub event
 * - Leaderboard update (if amount changes)
 */
router.put("/:id", async (req, res) => {
    const order = await Order.findByIdAndUpdate(
        req.params.id,
        req.body,
        { new: true }
    );

    if (!order) return res.status(404).json({ error: "Order not found" });

    await redis.del(`order:${order._id}`);

    await publish({
        type: "ORDER_UPDATED",
        orderId: String(order._id),
        updates: req.body,
        at: new Date().toISOString()
    });

    // If amount exists, update leaderboard score

```

```

        if (typeof order.amount === "number") {
            await upsertOrderScore(order._id, order.amount);
        }

        res.json(order);
    });

module.exports = router;

```

## services/leaderboard.js

```

const redis = require("../middleware/redisClient");

const LB_KEY = "leaderboard:top_orders";

/***
 * Store orderId with score=amount in a Sorted Set.
 * Uses ZADD (replace score if exists).
 */
async function upsertOrderScore(orderId, amount) {
    try {
        // node-redis v4: zAdd(key, [{ score, value }])
        await redis.zAdd(LB_KEY, [{ score: Number(amount), value: String(orderId) }]);
    } catch (err) {
        // fallback using raw command (in case of API mismatch)
        try {
            await redis.sendCommand(["ZADD", LB_KEY, String(amount), String(orderId)]);
        } catch (e2) {
            console.error("Leaderboard upsert error:", e2.message);
        }
    }
}

/***
 * Return top N orderIds with scores (highest first).
 */
async function topOrders(limit = 10) {
    const n = Number(limit);
    try {
        // node-redis v4: zRangeWithScores(key, start, stop, { REV: true })
        const rows = await redis.zRangeWithScores(LB_KEY, 0, n - 1, { REV: true });
        return rows.map(r => ({ orderId: r.value, amount: Number(r.score) }));
    } catch (err) {
        // fallback: ZREVRANGE key 0 n-1 WITHSCORES
        const raw = await redis.sendCommand(["ZREVRANGE", LB_KEY, "0", String(n - 1), "WITHSCORES"]);
        const out = [];
        for (let i = 0; i < raw.length; i += 2) {
            out.push({ orderId: raw[i], amount: Number(raw[i + 1]) });
        }
        return out;
    }
}

```

```
}

module.exports = {
  LB_KEY,
  upsertOrderScore,
  topOrders
};
```

## services/pubsub.js

```
const redis = require("../middleware/redisClient");

const CHANNEL = "orders.events";

async function publish(event) {
  try {
    await redis.publish(CHANNEL, JSON.stringify(event));
  } catch (err) {
    console.error("Publish error:", err.message);
  }
}

module.exports = {
  CHANNEL,
  publish
};
```

## services/queue.js

```
const redis = require("../middleware/redisClient");

const QUEUE_KEY = "jobs";

/**
 * Enqueue a job payload (string). We use LPUSH + BRPOP for FIFO.
 */
async function enqueue(job) {
  try {
    await redis.lPush(QUEUE_KEY, job);
  } catch (err) {
    console.error("Enqueue error:", err.message);
  }
}

module.exports = {
  QUEUE_KEY,
  enqueue
};
```

## workers/jobsWorker.js

```
/***
 * Jobs Worker (Queue: LPUSH + BRPOP)
 * Run: node workers/jobsWorker.js
```

```

/*
 * Waits forever for jobs in list "jobs". Uses BRPOP jobs 0 (blocking).
 */
const { createClient } = require("redis");

const QUEUE_KEY = "jobs";

async function main() {
  const client = createClient();
  client.on("error", (err) => console.error("Redis Worker Error:", err));

  await client.connect();
  console.log(`Worker connected. Waiting on queue: ${QUEUE_KEY}`);

  while (true) {
    // node-redis v4: brPop(key, timeout)
    const result = await client.brPop(QUEUE_KEY, 0);

    // result usually looks like: { key: 'jobs', element: '...' }
    const job = result?.element ?? (Array.isArray(result) ? result[1] : null);

    if (!job) continue;

    console.log(` Got job:${job}`);

    // Demo "job handler"
    if (job.startsWith("email:order_created:")) {
      const orderId = job.split(":").pop();
      console.log(` (demo) Sending order-created email for order ${orderId}`);
    } else {
      console.log(` (demo) Unknown job type:${job}`);
    }
  }
}

main().catch((e) => {
  console.error(e);
  process.exit(1);
});

```

## [workers/orderEventSubscriber.js](#)

```

/**
 * Order Event Subscriber (Pub/Sub)
 * Run: node workers/orderEventSubscriber.js
 *
 * Listens on Redis channel "orders.events" and prints incoming messages.
 */
const { createClient } = require("redis");

const CHANNEL = "orders.events";

async function main() {

```

```
const sub = createClient();
sub.on("error", (err) => console.error("Redis Subscriber Error:", err));

await sub.connect();
console.log(`⚡️Subscriber connected. Listening on channel: ${CHANNEL}`);

await sub.subscribe(CHANNEL, (message) => {
  try {
    const event = JSON.parse(message);
    console.log("✉️ Event:", event);
  } catch {
    console.log("✉️ Raw message:", message);
  }
});

main().catch((e) => {
  console.error(e);
  process.exit(1);
});
```