An-Najah National University
Faculty of Engineering & Information Technology

Dos-Project - Part 2

PhD. Samer Arandi

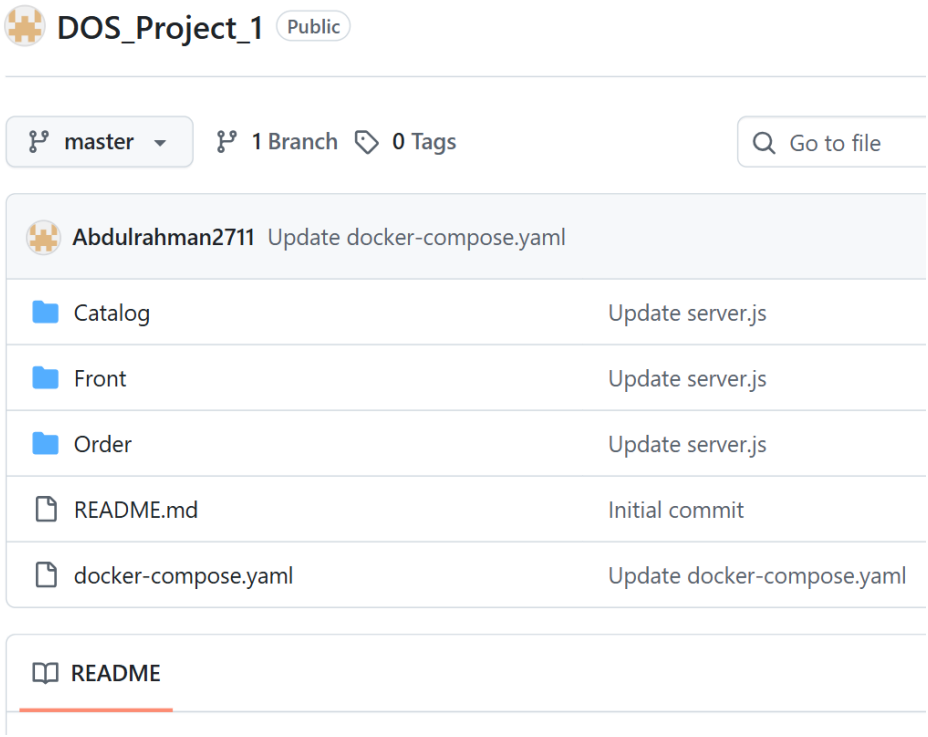
Abdulrahman Hassoun
– 12027846

1. Introduction

This report presents the main features implemented in the second phase of the project, which include an in-memory caching system, a load balancing strategy, and the replication of catalog and order services using a microservices architecture with RESTful APIs.

2. Project Structure

We have replicated the Catalog service and its database file, and the same was done for the Order service as shown in the following diagram:



3. Cache

For read operations, a cache based on a Map structure was implemented to store responses from catalog queries. Cache entries are indexed by the query (for example, `search:<topic>` and `info:<id>`) and contain an expiration timestamp (TTL). When a search request arrives, the frontend first invokes `cacheGet()` to verify if a valid (non-expired) entry exists. In case of a cache hit, the stored response is returned immediately. Otherwise, the request is forwarded to a catalog replica (chosen via round-robin load balancing), the received data is saved with `cacheSet()`, and finally sent back to the client. The cache has a maximum capacity; when full, the oldest entry is evicted. Additionally, a server-push invalidation endpoint (`/cache/invalidate/:id`) is provided so that the catalog service can invalidate cached information about an item before performing write operations, thereby preserving consistency.

```
1 const cache = new Map(); // key -> { value, expiresAt }
2
3 // search => uses the cache and it could hit or mis
```

```
app.get("/search/:topic", async (req, res) => [
    const topic = String(req.params.topic || "");
    const key = `search:${topic.toLowerCase()}`;

    const hit = cacheGet(key);
    if (hit) return res.json({ cached: true, data: hit });

    const catalogURL = pickCatalog();
    const r = await httpJSON(`${catalogURL}/search/${encodeURIComponent
   (topic)}`);
    if (!r.ok) return res.status(r.status).json(r.data ?? { error: "
   Catalog error" });
    cacheSet(ke (property) cached: boolean
    res.json({ cached: false, replica: catalogURL, data: r.data });
});
```

4. Load Balancing

Requests are distributed across multiple replicas of the backend services. The Round Robin algorithm was chosen for its simplicity and low overhead. For every incoming search request, the frontend picks one replica in a cyclic order, guaranteeing that consecutive requests are sent to different replicas. This method balances the load among the replicas and avoids any single replica becoming a bottleneck.

```
const CATALOG_REPLICAS = (process.env.CATALOG_REPLICAS || "http://
   localhost:4001,http://localhost:4003")
  .split(",")
  .map(s => s.trim())
  .filter(Boolean);

const ORDER_REPLICAS = (process.env.ORDER_REPLICAS || "http://localhost
   :4002,http://localhost:4004")
  .split(",")
  .map(s => s.trim())
  .filter(Boolean);
```

The code that actually implements the load balancing:

```
let catIdx = 0;
let ordIdx = 0;

function pickCatalog() {
    const url = CATALOG_REPLICAS[catIdx % CATALOG_REPLICAS.length];
    catIdx = (catIdx + 1) % CATALOG_REPLICAS.length;
    return url;
}

function pickOrder() {
    const url = ORDER_REPLICAS[ordIdx % ORDER_REPLICAS.length];
    ordIdx = (ordIdx + 1) % ORDER_REPLICAS.length;
    return url;
}
```

Note: To maintain consistency between the primary replica and its backup, all write operations are processed only by the primary replica, which then propagates the updates to the backup replica so that both remain in the same state.

```
async function replicateToPeer(op) {
    if (!IS_PRIMARY) return;
    try {
    await fetch(`${PEER_URL}/internal/replicate`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(op),
    });
    } catch (e) {
    console.warn(`[Catalog:${PORT}] replicateToPeer failed:`, e?.
   message || e);
    }
}

app.put("/update/price", async (req, res) => {
    const { id, price } = req.body || {};
    if (!Number.isInteger(id) || !Number.isFinite(price) || price < 0)
    return res.status(400).json({ error: "Invalid id or price" });

    await invalidateCache(id);
    const local = applyWrite({ type: "update_price", id, price });
    if (!local.ok) return res.status(local.code).json({ error: local.
   error });

    await replicateToPeer({ type: "update_price", id, price });
    res.json({ ok: true, primary: IS_PRIMARY, port: PORT });
});
```

5. Cache Invalidation

To avoid serving outdated data from the frontend cache, a server-push cache invalidation mechanism was implemented. Before carrying out any write operation, the backend services notify the frontend to invalidate the relevant cache entries.

⇒ Before any write, the cache at the frontend service is invalidated.

```
// Invalidate endpoint => server-push implementation
```

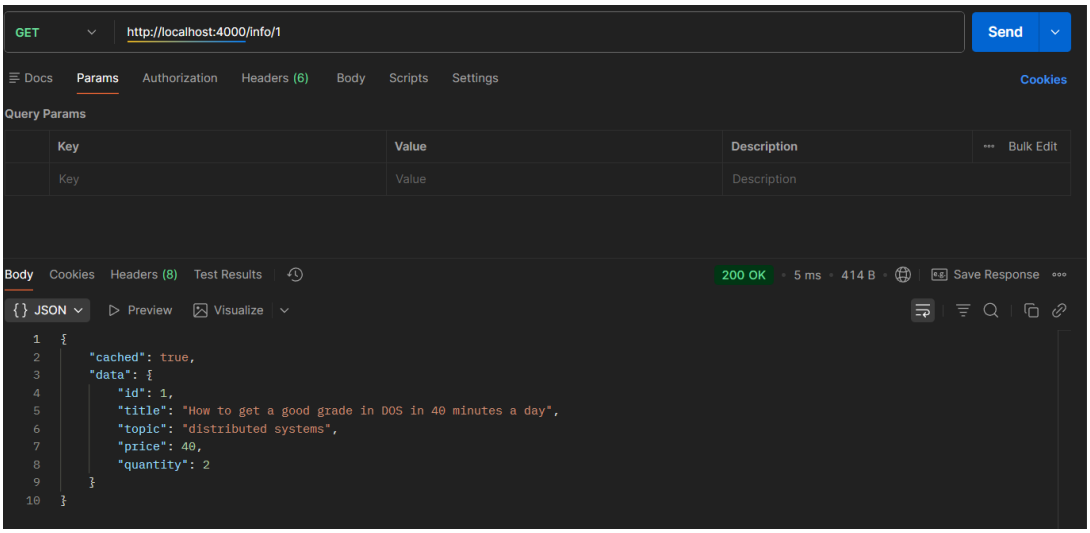
```
app.post("/cache/invalidate/:id", (req, res) => {
    const id = Number(req.params.id);
    if (!Number.isInteger(id) || id <= 0) return res.status(400).json({
    error: "Invalid id" });

    invalidateBook(id);
    res.json({ ok: true, invalidated: id });
});
```

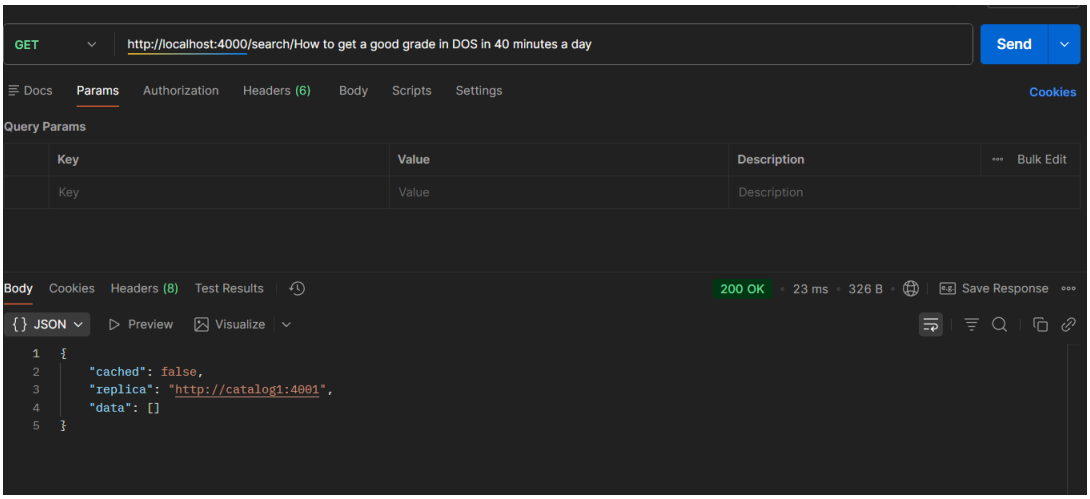
Note: The Order service communicates with the primary Catalog service, and any update performed by the primary Catalog is propagated to its backup.

6. Experiments

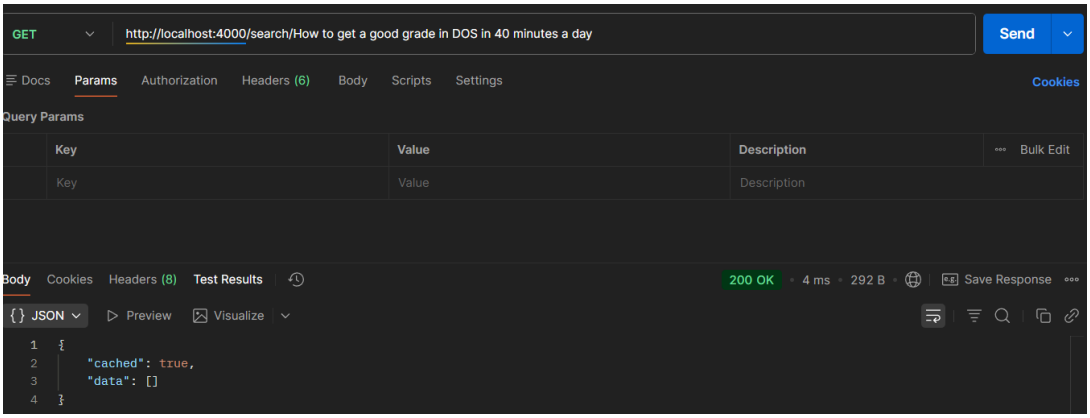
Not Cached (miss) ⇒ 93ms
Cached (hit) ⇒ 5ms



 # Not cached (miss) ⇒ 23ms



 # Cached (hit) ⇒ 4ms


GET
http://localhost:4000/search/How to get a good grade in DOS in 40 minutes a day
Send
Docs
Params
Authorization
Headers (6)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (8)
Test Results
200 OK
4 ms
292 B
Save Response
{ } JSON
Preview
Visualize
1 {
2 "cached": true,
3 "data": []
4 }

7. Conclusion

Implementation: introducing caching has greatly decreased the average response time.

Note: The first time a topic is searched, the data is obtained from the catalog service (cache miss). When the same topic is searched again, the data is retrieved directly from the cache, showing a clear improvement in efficiency.

To assess the performance of the in-memory cache, response times were recorded for repeated read requests. The initial request produced a cache miss and needed to communicate with the catalog service, resulting in higher latency. Following requests were served straight from the cache, which markedly reduced the response time.

Ports and Services:
Front: 4000
Catalog 1: 4001
Catalog 2: 4003
Order 1: 4002
Order 2: 4004