An-Najah National University

Faculty of Engineering and IT

جامعة النجاح الوطنية

كلية الهندسة وتكنولوجيا المعلومات

Distributed Operating Systems

10636456

1t Semester | 2025-2026

Bazar-com Project | Part 2

Jana Moe'n Tahayni

12220650

Cache Consistency:

```
87  // search by id
88  app.get('/info/:id', async (req, res) => {
89    try {
90      const id = req.params.id;
91      console.log(`ℹ️ Frontend: Info request for book ID: ${id}`);
92
93      // serve from cache for read requests (if enabled)
94      const key = infoCacheKey(id);
95      if (CACHE_ENABLED && cache.has(key)) {
96        console.log(`⚡ Frontend: Cache HIT for book ID: ${id}`);
97        return res.json(cache.get(key));
98      }
99
100     console.log(`☁️ Frontend: Cache MISS for book ID: ${id}`);
101
102     // pick a catalog replica (round-robin) for load balancing
103     const catalogReplicas = parseReplicas(CATALOG_REPLICAS, CATALOG);
104     const chosenCatalog = pickReplica(catalogReplicas, 'catalog');
105
106     const response = await axios.get(`${chosenCatalog}/info/${id}`);
107
108     console.log(`✅ Frontend: Info retrieved for: "${response.data.title}"`);
109
110     // store in cache (only for read requests)
111     if (CACHE_ENABLED) {
112       cache.set(key, response.data);
113       enforceCacheLimit();
114       console.log(`🧠 Frontend: Cached info for book ID: ${id} (cache size: ${cache.size})`);
115     }
```

As the script code from frontend service above explains when searching an information for a specific id book, the service searches first if the item is cached or not. If not, it communicates with catalog service to get the required book. Then, it stores the book in its cache for next time.

To answer the following questions:

Compute the average response time (query/buy) of your new systems. What is the response time with and without caching? How much does caching help?

For query part:

I will calculate average time for searching 10 values without enabling the cache, and 10 search values with enabling it.

For purchase part:

I will calculate average time for purchasing 5 times without enabling the cache, and another 5 times with enabling the cache.

Query Part:

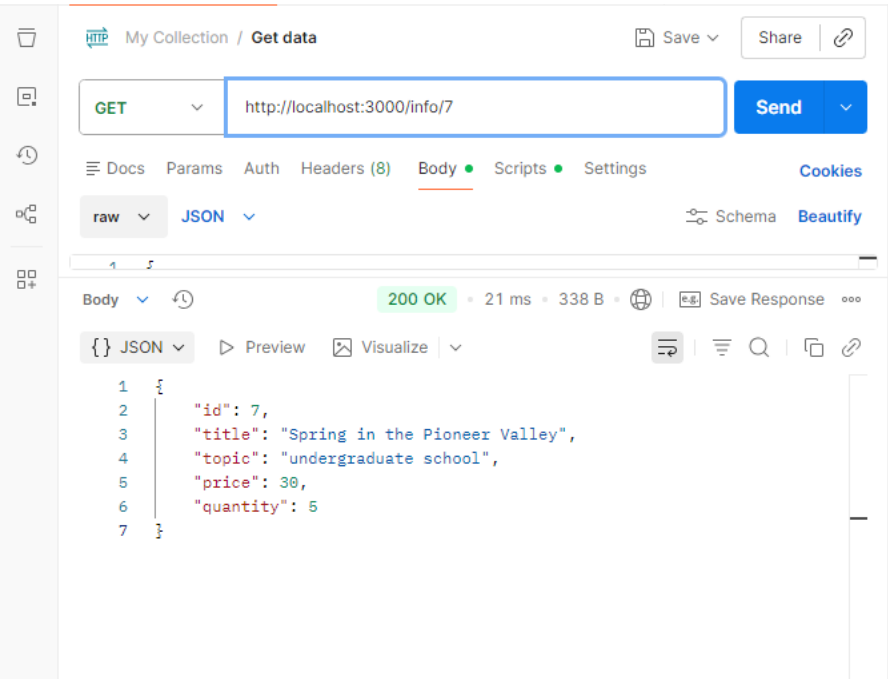
Without cache:

app.js M docker-compose.yml M

docker > docker-compose.yml

```yaml
version: "3.9"

services:
  # Frontend (NOT replicated)
  fronted:
    build:
      context: ../fronted
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
      - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
      - ORDER_REPLICAS=http://order1:5001,http://order2:5002
      - CACHE_ENABLED=false
      - CACHE_MAX=100
    depends_on:
      - catalog1
      - catalog2
      - order1
      - order2
```

Example to one output:



```
fronted-1   | ℹ Frontend: Info request for book ID: 7
catalog1-1  | ℹ Info request for book ID: 7 - "Spring in the Pioneer Valley"
fronted-1   | 🌩 Frontend: Cache MISS for book ID: 7
fronted-1   | ✅ Frontend: Info retrieved for: "Spring in the Pioneer Valley"
```

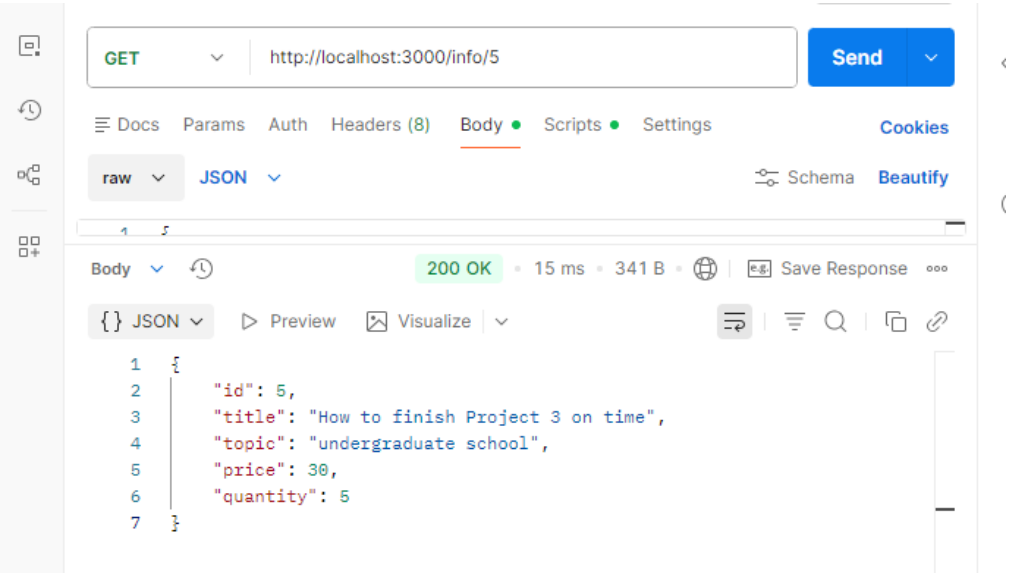
Average time for 10 searches by id operations: 63.5ms

With cache enable:

```yaml
services:
  # Frontend (NOT replicated)
  fronted:
    build:
      context: ../fronted
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
      - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
      - ORDER_REPLICAS=http://order1:5001,http://order2:5002
      - CACHE_ENABLED=true
      - CACHE_MAX=100
    depends_on:
```

Output example:

```
fronted-1   | ℹ Frontend: Info request for book ID: 5
fronted-1   | ⚡ Frontend: Cache HIT for book ID: 5
```



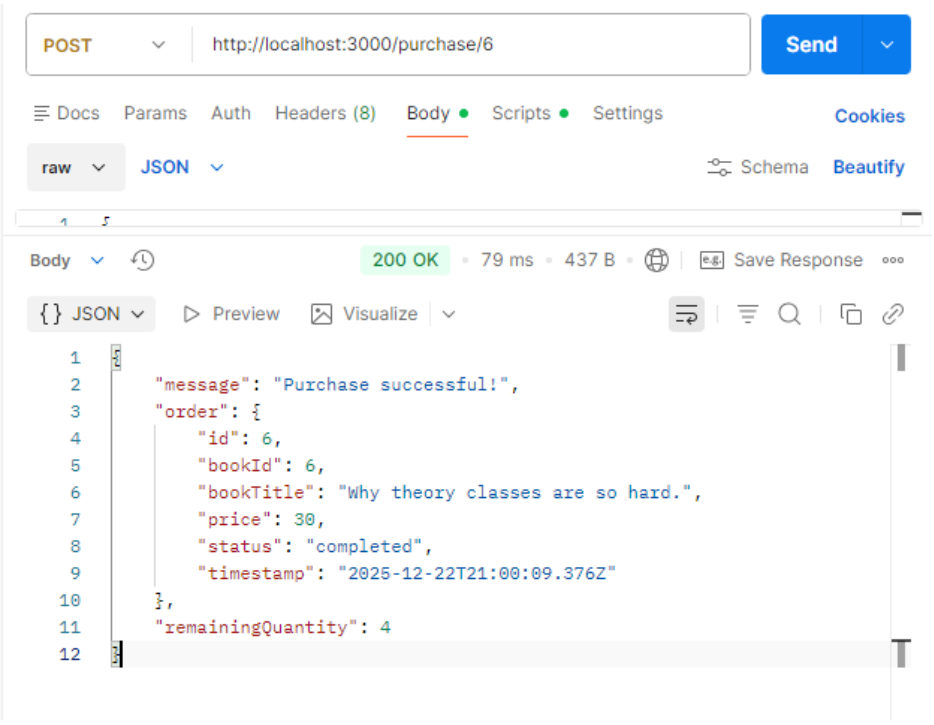
Average time for 10 searches by id operations: 20.7ms

Performance for caching in search operation: (63.5 - 20.7)/63.5 *100% = 67.4%

For Purchase Part:

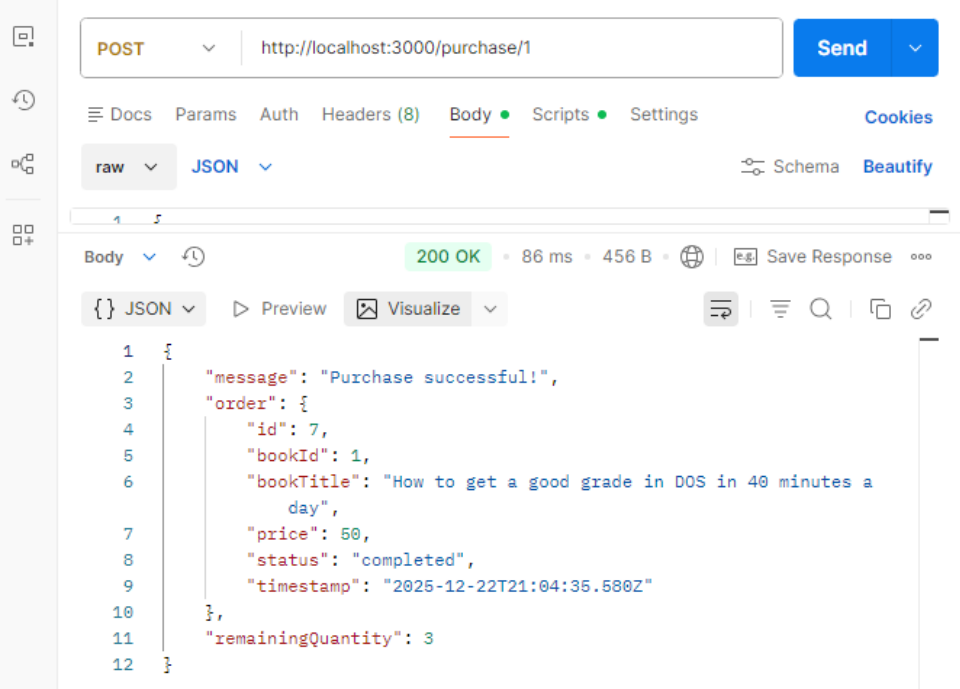
With enabled cache:

```
fronted-1   | 🛒 Frontend: Purchase request for book ID: 5
catalog1-1  | ℹ️ Info request for book ID: 5 - "How to finish Project 3 on time"
order1-1    | 🛒 Purchase request for: "How to finish Project 3 on time" - Current quantity: 5
fronted-1   | 🧹 Cache invalidate for book ID: 5 - existed: true

catalog1-1  | 🧹 Sent cache invalidate to frontend for book ID: 5
fronted-1   | 🧹 Cache invalidate for book ID: 5 - existed: false
catalog2-1  | 🧹 Sent cache invalidate to frontend for book ID: 5
catalog2-1  | 🔄 Updated book ID: 5 - Quantity: 4, Price: 30
catalog1-1  | 📝 Replicated update of book ID 5 to peer replica
catalog1-1  | 🔄 Updated book ID: 5 - Quantity: 4, Price: 30
order2-1    | 📥 Received replicated order ID 1 and saved locally
order1-1    | 📝 Replicated order ID 1 to peer replica


fronted-1   | ✅ Frontend: Purchase successful - Order ID: 1
order1-1    | ✅ Purchase successful! Order ID: 1, Remaining quantity: 4
```



Average: 148.2 ms

With disabled cache:




```
fronted-1   | 🛒 Frontend: Purchase request for book ID: 1

order2-1    | 🛒 Purchase request for: "How to get a good grade in DOS in 40 minutes a day" - Current quantity: 4
catalog2-1  | ℹ️ Info request for book ID: 1 - "How to get a good grade in DOS in 40 minutes a day"

fronted-1   | 🧹 Cache invalidate for book ID: 1 - existed: false
catalog2-1  | 🧹 Sent cache invalidate to frontend for book ID: 1
catalog1-1  | 🧹 Sent cache invalidate to frontend for book ID: 1
fronted-1   | 🧹 Cache invalidate for book ID: 1 - existed: false

catalog2-1  | 📡 Replicated update of book ID 1 to peer replica
order2-1    | 📡 Replicated order ID 7 to peer replica
order1-1    | 📥 Received replicated order ID 7 and saved locally
fronted-1   | ✅ Frontend: Purchase successful - Order ID: 7
catalog1-1  | 🔄 Updated book ID: 1 - Quantity: 3, Price: 50
catalog2-1  | 🔄 Updated book ID: 1 - Quantity: 3, Price: 50
order2-1    | ✅ Purchase successful! Order ID: 7, Remaining quantity: 3
```

Average: 74.2 ms

Operation	Avg. With cache	Avg. Without cache	Improvement
Query	20.7	63.5	67.4%

Purchase	148.2	74.2	0.27%

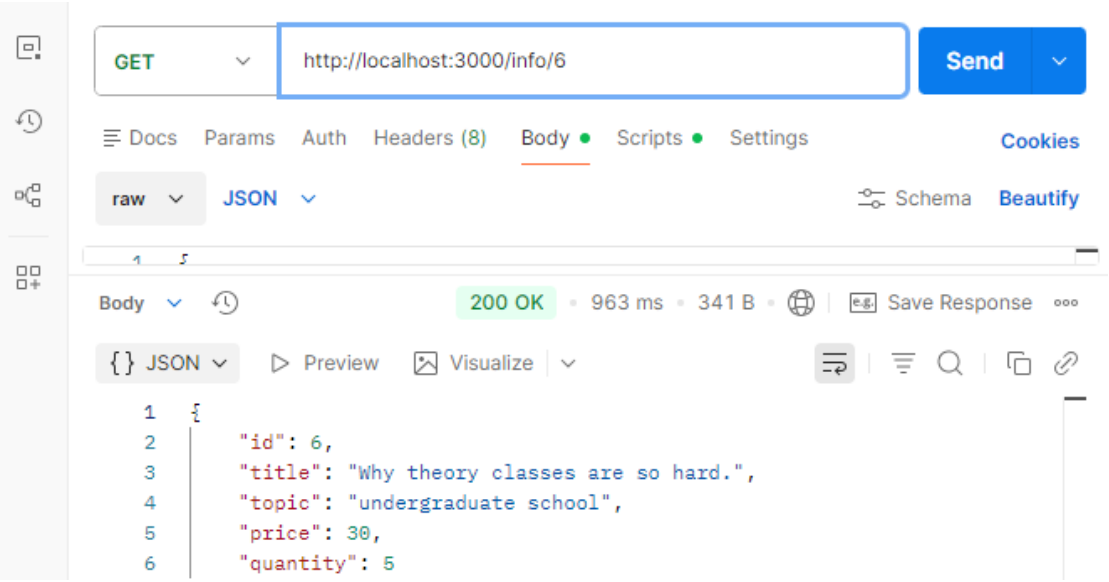
The average response time for purchase operations with caching enabled was slightly higher than without caching. This behavior is expected, since purchase requests are writing operations that must always contact the order and catalog services. Additionally, when caching is enabled, the catalog service sends a cache invalidation request to the frontend prior to updating the database, introducing a small overhead. As a result, caching significantly improves query latency but does not benefit and may slightly increase the latency of purchase operation.

To answer this question:

Construct a simple experiment that issues orders or catalog updates (i.e., database writes) to invalidate the cache and maintain cache consistency. What is the overhead of cache consistency operations? What is the latency of a subsequent request that sees a cache miss?

Experiment:

1. Search for book with id = 6 when cache is null:

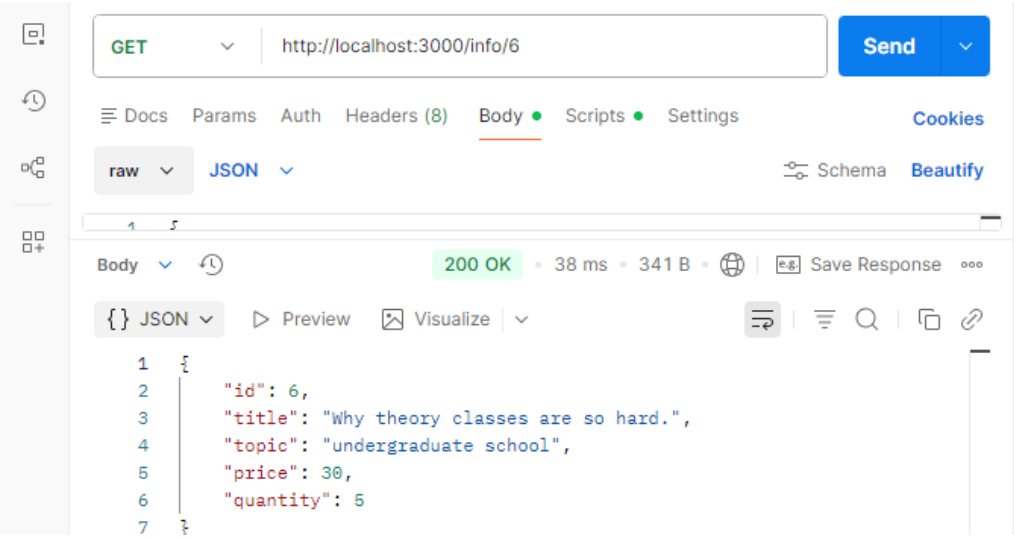


Note: time is 963 ms.

Cache miss =>

```
fronted-1   | ℹ️ Frontend: Info request for book ID: 6
fronted-1   | 🧠 Frontend: Cache MISS for book ID: 6
catalog1-1  | ℹ️ Info request for book ID: 6 - "Why theory classes are so hard."
fronted-1   | ✅ Frontend: Info retrieved for: "Why theory classes are so hard."
fronted-1   | 🧠 Frontend: Cached info for book ID: 6 (cache size: 1)
```

2. Search same book again [now it is in cache]:

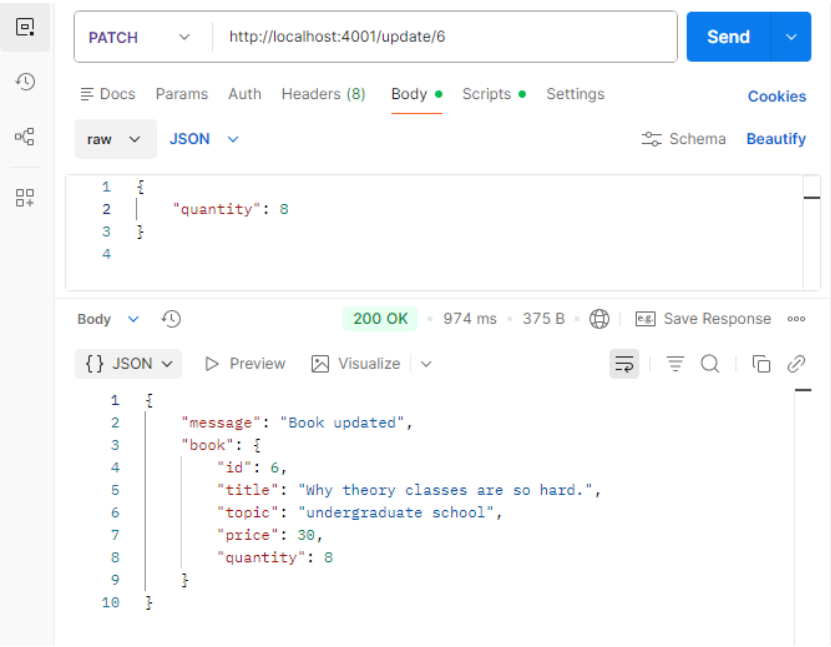


Note: time is 38 ms.

Cache hit => no going to catalog service

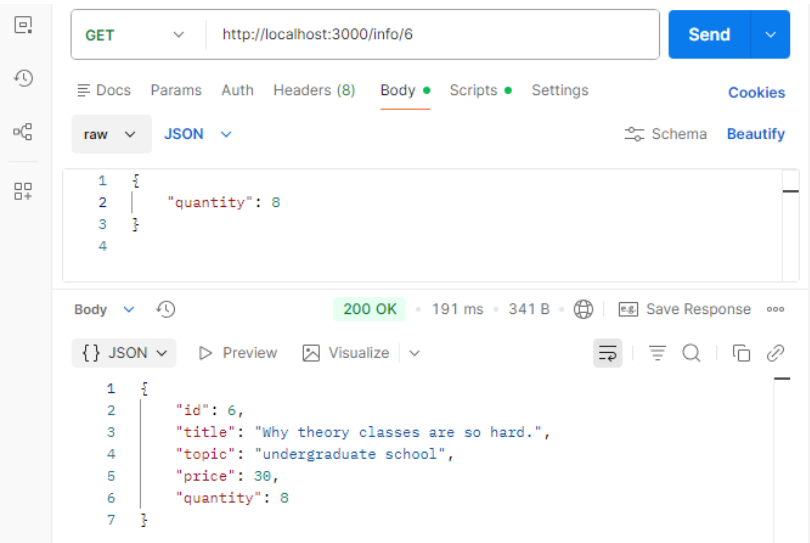
```
fronted-1   | ℹ️ Frontend: Info request for book ID: 6
fronted-1   | ⚡ Frontend: Cache HIT for book ID: 6
```

3. Update operation [patch] on book with id = 6 to see the invalidation.




```
catalog1-1  | 🧹 Sent cache invalidate to frontend for book ID: 6
fronted-1   | 🧹 Cache invalidate for book ID: 6 - existed: true
fronted-1   | 🧹 Cache invalidate for book ID: 6 - existed: false
catalog2-1  | 🧹 Sent cache invalidate to frontend for book ID: 6
catalog2-1  | 🔄 Updated book ID: 6 - Quantity: 8, Price: 30
catalog1-1  | 📡 Replicated update of book ID 6 to peer replica
catalog1-1  | 🔄 Updated book ID: 6 - Quantity: 8, Price: 30
```

4. Search for book with id = 6 to ensure it will be missed from cache.



```
fronted-1   | ℹ️ Frontend: Info request for book ID: 6
fronted-1   | 🌩️ Frontend: Cache MISS for book ID: 6
catalog2-1  | ℹ️ Info request for book ID: 6 - "Why theory classes are so hard."
fronted-1   | ✅ Frontend: Info retrieved for: "Why theory classes are so hard."
fronted-1   | 🧠 Frontend: Cached info for book ID: 6 (cache size: 1)
```

The first query following a write operation experienced a cache miss and therefore had higher latency compared to cache hits.

Load Balancing:

To distribute client requests across replicated services, we implemented load balancing at the frontend layer. The frontend maintains a list of available catalog and order replicas and routes incoming requests using a simple round-robin algorithm.
Each new request is forwarded to the next replica in the list, ensuring an even distribution of workload across replicas.

This approach improves scalability and fault tolerance, as the system can continue to serve requests even if one replica becomes overloaded.
We chose round-robin load balancing due to its simplicity and low overhead, which is appropriate for our small-scale replicated system.

```
20  // NEW: round-robin indices for load balancing
21  let catalogIndex = 0;
22  let orderIndex = 0;
23
24  // NEW: helper to parse replicas list, fallback to single URL
25  const parseReplicas = (listStr, fallbackUrl) => {
26    if (!listStr) return [fallbackUrl];
27    const list = listStr.split(',').map(s => s.trim()).filter(Boolean);
28    return list.length > 0 ? list : [fallbackUrl];
29  };
30
31  // NEW: choose a replica using round-robin
32  const pickReplica = (replicas, type) => {
33    if (type === 'catalog') {
34      const chosen = replicas[catalogIndex % replicas.length];
35      catalogIndex++;
36      return chosen;
37    }
38    const chosen = replicas[orderIndex % replicas.length];
39    orderIndex++;
40    return chosen;
41  };
```

```
order1-1    | ✅ Purchase successful! Order ID: 6, Remaining quantity: 4
fronted-1   | 🧹 Cache invalidate for book ID: 1 - existed: false
catalog1-1  | 🔄 Updated book ID: 1 - Quantity: 4, Price: 50
catalog2-1  | 🔄 Updated book ID: 1 - Quantity: 4, Price: 50


fronted-1   | ✅ Frontend: Purchase successful - Order ID: 6
fronted-1   | 🛒 Frontend: Purchase request for book ID: 1


order2-1    | 🛒 Purchase request for: "How to get a good grade in DOS in 40 minutes a day" - Current quantity: 4
catalog2-1  | ℹ️ Info request for book ID: 1 - "How to get a good grade in DOS in 40 minutes a day"
```

Docker-aization:

The system was fully containerized using Docker to simplify deployment and ensure consistency across environments.
Each service (frontend, catalog replicas, and order replicas) runs in a separate container with its own runtime and dependencies.

Docker Compose was used to orchestrate the multi-container setup, defining service dependencies, environment variables, exposed ports, and persistent volumes.
Each replica uses a dedicated volume to store its local database file, ensuring data persistence across container restarts.

docker > docker-compose.yml

```yaml
version: "3.9"

services:
  # Frontend (NOT replicated)
  fronted:
    build:
      context: ../fronted
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
      - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
      - ORDER_REPLICAS=http://order1:5001,http://order2:5002
      - CACHE_ENABLED=true
      - CACHE_MAX=30
    depends_on:
      - catalog1
      - catalog2
      - order1
      - order2

  # Catalog replicas
  catalog1:
    build:
      context: ../catalog
      dockerfile: Dockerfile
    ports:
      - "4001:4001"
    environment:
      - PORT=4001
      - DB_FILE=/data/catalog.json
      - PEER_URL=http://catalog2:4002
      - FRONTEND_URL=http://fronted:3000
    volumes:
      - catalog1_data:/data

  catalog2:
    build:
      context: ../catalog
      dockerfile: Dockerfile
    ports:
      - "4002:4002"
    environment:
      - PORT=4002
      - DB_FILE=/data/catalog.json
      - PEER_URL=http://catalog1:4001
      - FRONTEND_URL=http://fronted:3000
    volumes:
```

```yaml
  # Order replicas
  order1:
    build:
      context: ../order
      dockerfile: Dockerfile
    ports:
      - "5001:5001"
    environment:
      - PORT=5001
      - DB_FILE=/data/orders.json
      - PEER_URL=http://order2:5002
      - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
      - CATALOG_URL=http://catalog1:4001
      - FRONTEND_URL=http://fronted:3000
    volumes:
      - order1_data:/data
    depends_on:
      - catalog1
      - catalog2

  order2:
    build:
      context: ../order
      dockerfile: Dockerfile
    ports:
      - "5002:5002"
    environment:
      - PORT=5002
      - DB_FILE=/data/orders.json
```

How to run the project:

Inside VS code:

- Open terminal in docker folder
- Type: docker compose up –build