An-Najah National University

Faculty of Engineering

Computer Engineering Department

Distributed Operation Systems

Part2 Bazar.Com

Masa Abu Aisheh

&

Nancy Sawalmeh

Introduction:

This report highlights the main features added in the second phase of the project, aimed at improving performance, scalability, and reliability, while ensuring data consistency across replicas. The key updates include:

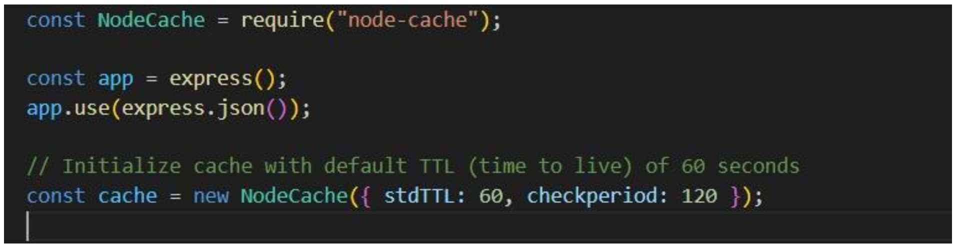
1. Adding an In-Memory Cache.
2. Implementing a Load Balancing Algorithm.
3. Setting up Server Replicas.
4. Synchronizing Data Between Replicas.

Procedure:

- **Adding an In-Memory Cache**
  To enhance the performance of the system by reducing redundant API calls, an **In-Memory Cache** was implemented using the node-cache library. Below are the steps involved:

  1. **Install the Required Library**
     Add the node-cache package to the project.
     npm install node-cache

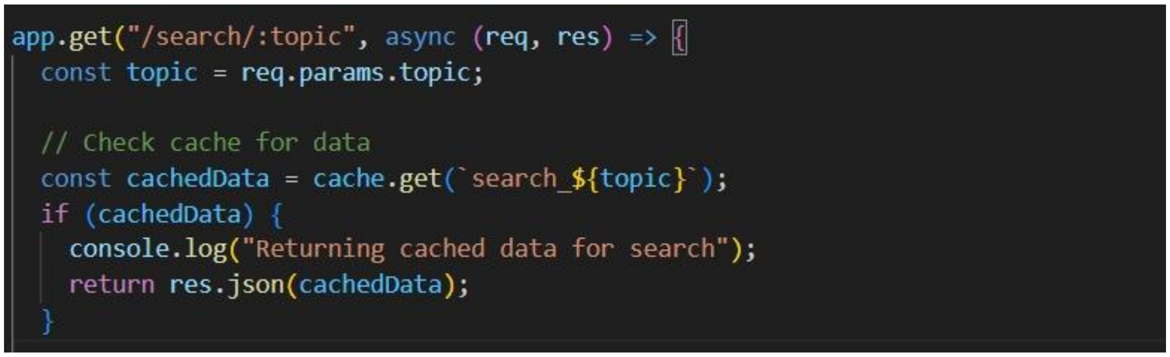
  2. **Initialize the Cache**
     Create an instance of the cache with a specified Time-To-Live (TTL) and check period for cache cleanup. In this case, a TTL of 60 seconds and a check period of 120 seconds were used.



```
const NodeCache = require("node-cache");

const app = express();
app.use(express.json());

// Initialize cache with default TTL (time to live) of 60 seconds
const cache = new NodeCache({ stdTTL: 60, checkperiod: 120 });
```

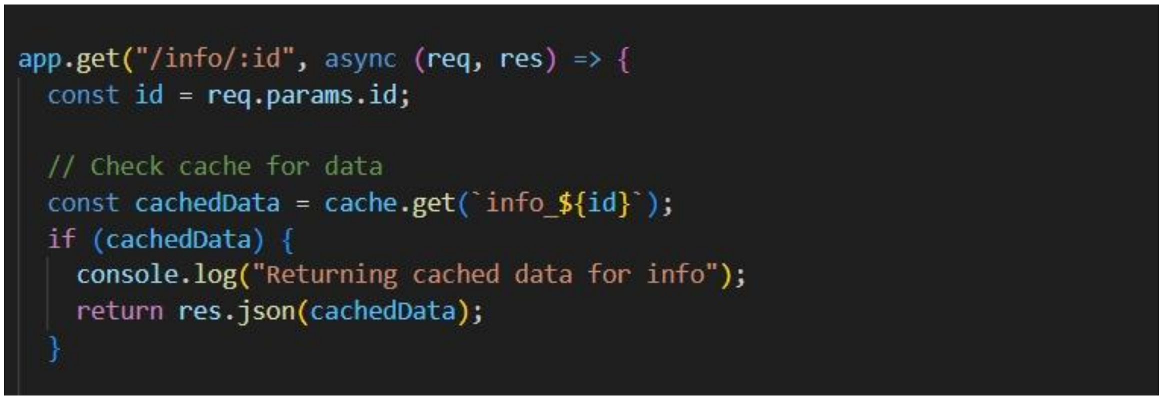
3. **Integrate Cache in API Routes**
   Use the cache to store and retrieve data for API endpoints.

4. **Apply Caching in Specific Endpoints**
   - Search Endpoint (/search/:topic)
     Cache search results using the topic as the key.



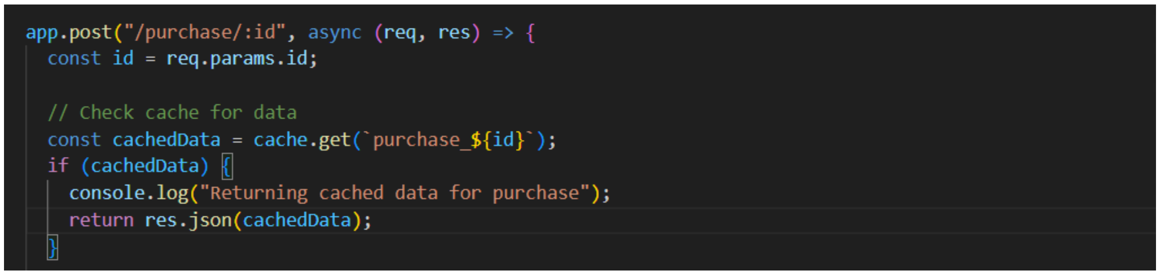
```
app.get("/search/:topic", async (req, res) => {
  const topic = req.params.topic;

  // Check cache for data
  const cachedData = cache.get(`search_${topic}`);
  if (cachedData) {
    console.log("Returning cached data for search");
    return res.json(cachedData);
  }
```

   - Info Endpoint (/info/:id)
     Cache item information using the id as the key.



```
app.get("/info/:id", async (req, res) => {
  const id = req.params.id;

  // Check cache for data
  const cachedData = cache.get(`info_${id}`);
  if (cachedData) {
    console.log("Returning cached data for info");
    return res.json(cachedData);
  }
```

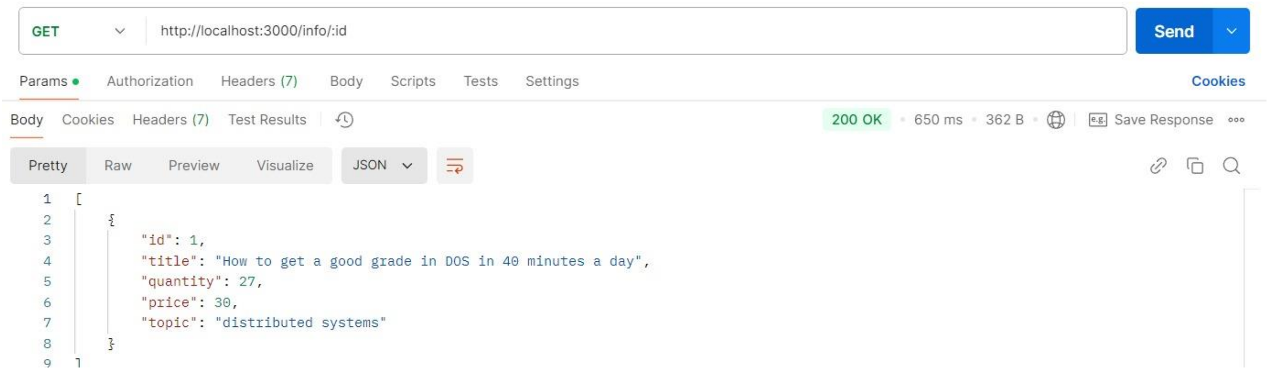
   - Purchase Endpoint (/purchase/:id)
     Cache purchase responses using the id as the key.



```
app.post("/purchase/:id", async (req, res) => {
  const id = req.params.id;

  // Check cache for data
  const cachedData = cache.get(`purchase_${id}`);
  if (cachedData) {
    console.log("Returning cached data for purchase");
    return res.json(cachedData);
  }
```

If the data is not found in the cache, a request is sent to the server, and the retrieved results are stored in the cache for future requests.

Results:

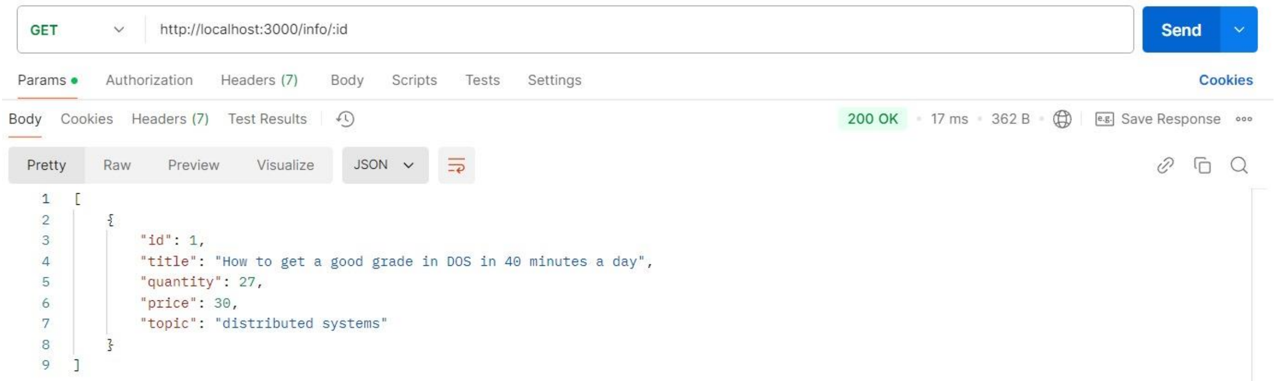


GET http://localhost:3000/info/:id

200 OK · 650 ms · 362 B

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day",
        "quantity": 27,
        "price": 30,
        "topic": "distributed systems"
    }
]
```

Cache miss -> request time: 650ms

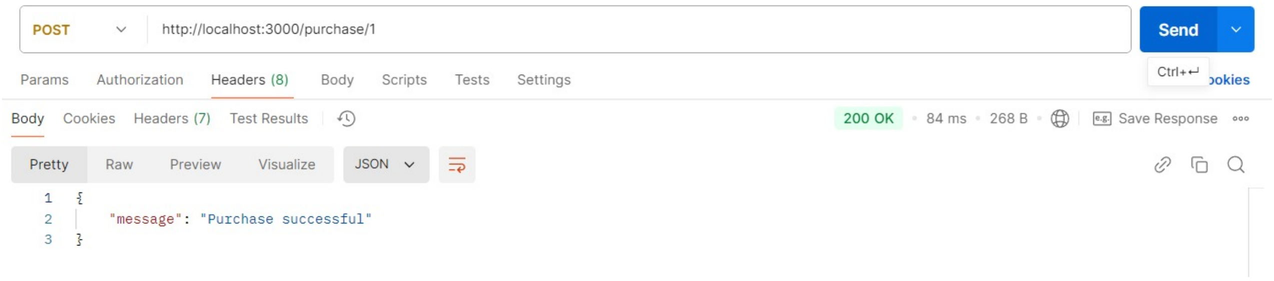


GET http://localhost:3000/info/:id

200 OK · 17 ms · 362 B

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day",
        "quantity": 27,
        "price": 30,
        "topic": "distributed systems"
    }
]
```

Cache hit -> request time: 17ms

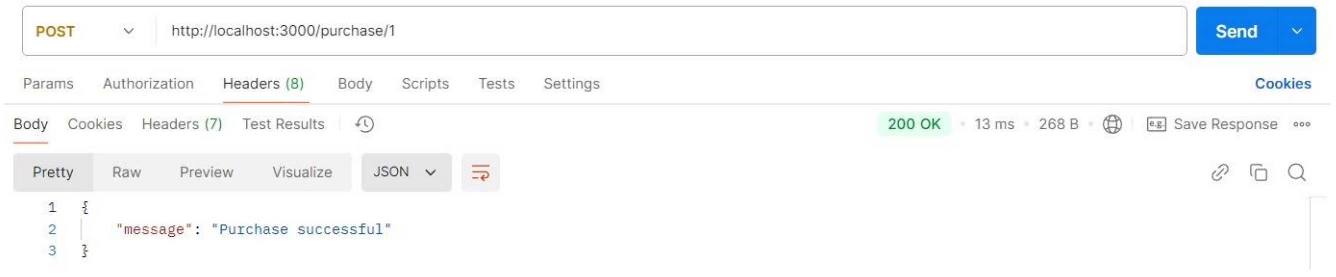


POST http://localhost:3000/purchase/1

200 OK · 84 ms · 268 B

```
{
    "message": "Purchase successful"
}
```

Cache miss -> request time: 84ms

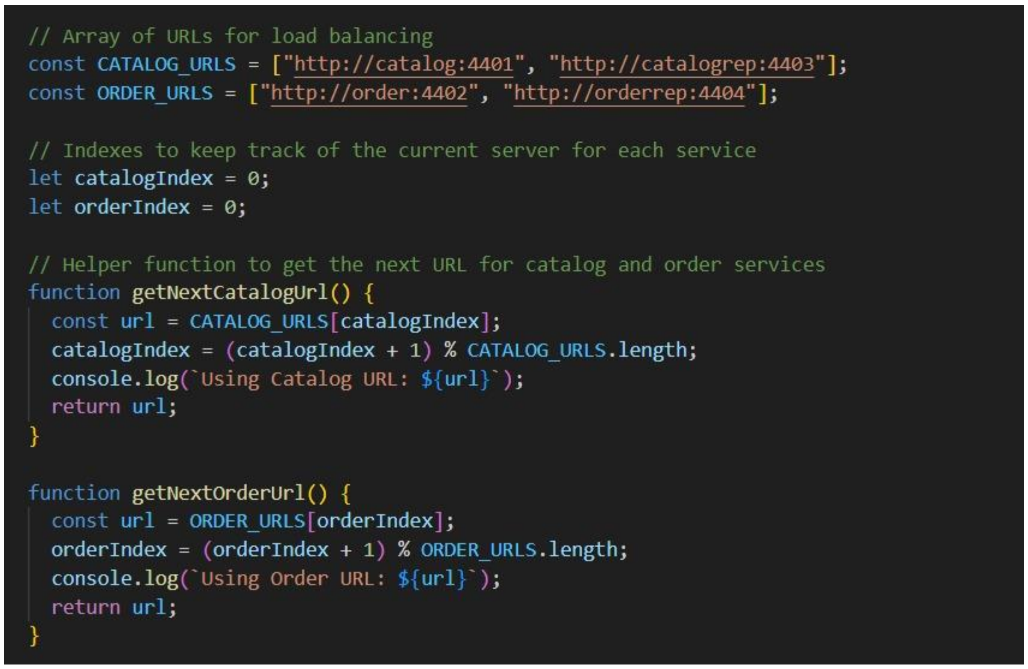


```
http://localhost:3000/purchase/1
```

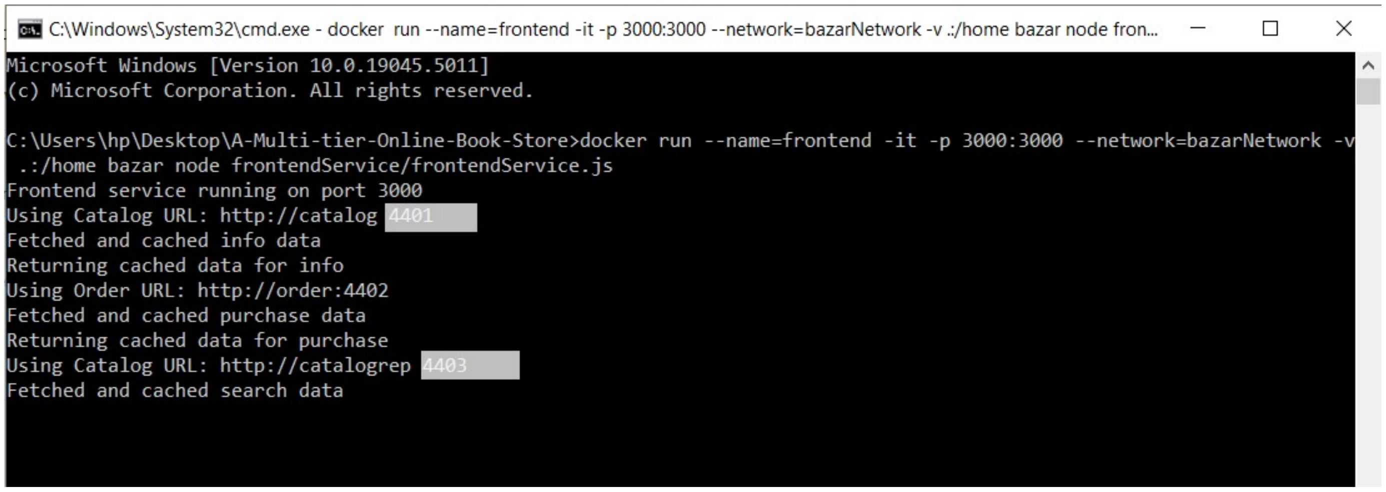
```
{
    "message": "Purchase successful"
}
```

Cache hit -> request time: 13ms

- **Implementing a Load Balancing Algorithm**
  A load balancing mechanism was implemented to distribute requests evenly across multiple server replicas. **The Round Robin algorithm** was chosen for its simplicity and effectiveness.
  In this setup:
  - For search operations using the catalog servers, requests are alternated across the available catalog server replicas using the Round Robin method.
  - The same approach is applied when retrieving book details from the database.
  - It is also used for processing book purchases through the order servers.

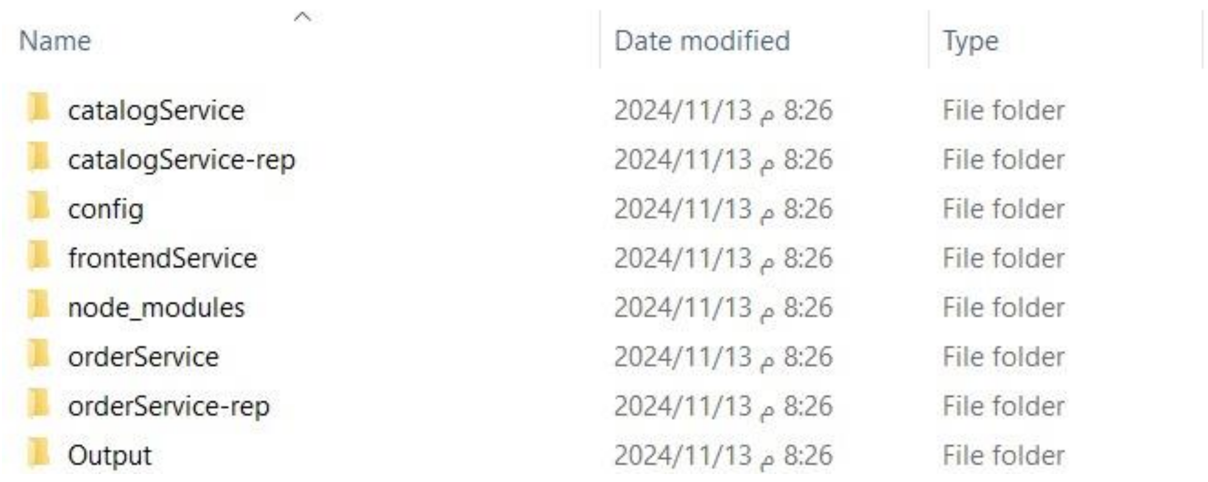


```
// Array of URLs for load balancing
const CATALOG_URLS = ["http://catalog:4401", "http://catalogrep:4403"];
const ORDER_URLS = ["http://order:4402", "http://orderrep:4404"];

// Indexes to keep track of the current server for each service
let catalogIndex = 0;
let orderIndex = 0;

// Helper function to get the next URL for catalog and order services
function getNextCatalogUrl() {
  const url = CATALOG_URLS[catalogIndex];
  catalogIndex = (catalogIndex + 1) % CATALOG_URLS.length;
  console.log(`Using Catalog URL: ${url}`);
  return url;
}

function getNextOrderUrl() {
  const url = ORDER_URLS[orderIndex];
  orderIndex = (orderIndex + 1) % ORDER_URLS.length;
  console.log(`Using Order URL: ${url}`);
  return url;
}
```


C:\Windows\System32\cmd.exe - docker run --name=frontend -it -p 3000:3000 --network=bazarNetwork -v .:/home bazar node fron...

```
Microsoft Windows [Version 10.0.19045.5011]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hp\Desktop\A-Multi-tier-Online-Book-Store>docker run --name=frontend -it -p 3000:3000 --network=bazarNetwork -v
 .:/home bazar node frontendService/frontendService.js
Frontend service running on port 3000
Using Catalog URL: http://catalog 4401
Fetched and cached info data
Returning cached data for info
Using Order URL: http://order:4402
Fetched and cached purchase data
Returning cached data for purchase
Using Catalog URL: http://catalogrep 4403
Fetched and cached search data
```

- **Setting up Server Replicas**
  Replicas of each server were created as you can see here.



Name	Date modified	Type
catalogService	2024/11/13 م 8:26	File folder
catalogService-rep	2024/11/13 م 8:26	File folder
config	2024/11/13 م 8:26	File folder
frontendService	2024/11/13 م 8:26	File folder
node_modules	2024/11/13 م 8:26	File folder
orderService	2024/11/13 م 8:26	File folder
orderService-rep	2024/11/13 م 8:26	File folder
Output	2024/11/13 م 8:26	File folder

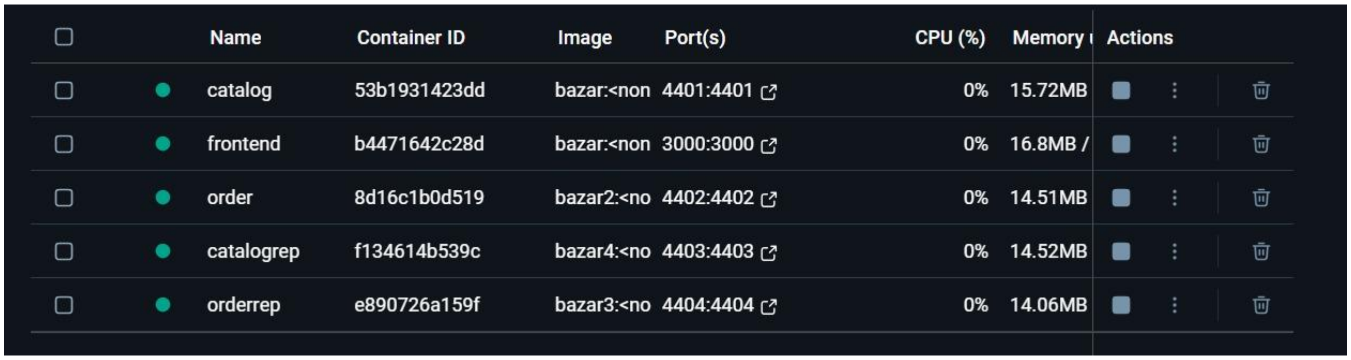
System Components

1. **Catalog Services**
   Two containers, catalog-service and its replica catalog-service-rep, are responsible for managing book listings. They are accessible on different ports: 4401 and 4403.

2. **Order Services**
   Two containers, order-service and its replica order-service-rep, handle book order processing. They operate on ports 4402 and 4404 and are designed to wait until the catalog services are fully operational before starting.

3. **Frontend**

A container, frontend-service, serves as the user interface, accessible through a web browser on port 3000. It depends on both the catalog and order services to be running before it initializes.

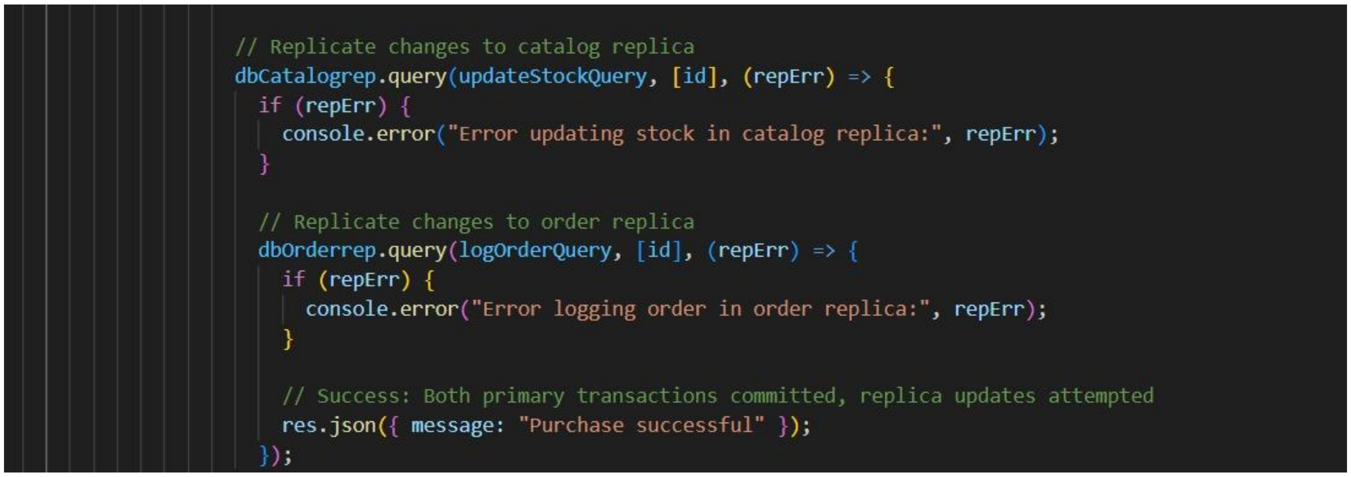


		Name	Container ID	Image	Port(s)	CPU (%)	Memory (	Actions		
☐	●	catalog	53b1931423dd	bazar:<non	4401:4401	0%	15.72MB	■	⋮	🗑
☐	●	frontend	b4471642c28d	bazar:<non	3000:3000	0%	16.8MB /	■	⋮	🗑
☐	●	order	8d16c1b0d519	bazar2:<no	4402:4402	0%	14.51MB	■	⋮	🗑
☐	●	catalogrep	f134614b539c	bazar4:<no	4403:4403	0%	14.52MB	■	⋮	🗑
☐	●	orderrep	e890726a159f	bazar3:<no	4404:4404	0%	14.06MB	■	⋮	🗑

- **Synchronizing Data Between Replicas**

  In this setup, synchronization is triggered only when an item is purchased.

  Inside the checkStockQuery function is called to update all replicas with the new stock level, ensuring consistent data across the system.



```
// Replicate changes to catalog replica
dbCatalogrep.query(updateStockQuery, [id], (repErr) => {
  if (repErr) {
    console.error("Error updating stock in catalog replica:", repErr);
  }

  // Replicate changes to order replica
  dbOrderrep.query(logOrderQuery, [id], (repErr) => {
    if (repErr) {
      console.error("Error logging order in order replica:", repErr);
    }

    // Success: Both primary transactions committed, replica updates attempted
    res.json({ message: "Purchase successful" });
  });
```

Conclusion:
In this phase of the project, several key improvements were made to enhance performance and reliability:

- **Adding an In-Memory Cache:** Improved response times by reducing data retrieval delays for frequently accessed information, making the system faster and more efficient.
- **Setting up Server Replicas:** Increased availability and redundancy by distributing the load across multiple instances.
- **Implementing a Load Balancing Algorithm:** Implemented a Round Robin algorithm to evenly distribute traffic across server replicas, preventing overload and ensuring smooth operation during high-demand periods.
- **Synchronizing Data Between Replicas:** Ensured data accuracy and consistency by reliably propagating updates (like stock changes) across all replicas.

These updates have significantly improved performance, scalability, and reliability, providing a strong and stable foundation for future growth.