Rearchitecting Bazar.com with Replication, Caching, and Dockerization
BAZAR.com Part 2

Sema Hodali 12029448
Leema Abu-Aladel 12029320

Introduction

Bazar.com, an online bookstore, recently experienced increased demand after a successful spring break sale. To accommodate this growing traffic and improve the speed of request processing, the system needed to be rearchitected to handle higher workloads efficiently. This report outlines the changes and enhancements made to the Bazar.com architecture, focusing on two primary objectives: replication and caching for better performance and Dockerization for easier deployment.

Problem Statement

Initially, the Bazar.com system faced significant delays in processing requests as the store's popularity grew. Users complained about the high latency when browsing and ordering books. The existing infrastructure was insufficient to handle the increased number of requests, particularly under high traffic conditions.

To address this, the system needed to be optimized through:

1.Replication of the catalog and order servers to distribute the load.
2.Caching to reduce the time spent fetching frequently requested data.
3.Load Balancing to distribute requests across replicas efficiently.

Solution Overview

The solution involved two major parts:

1.Replication and Caching: This part aimed to reduce latency by implementing an in-memory cache and ensuring database consistency across server replicas.
2.Dockerization: The system components were containerized using Docker to simplify deployment and ensure portability.

---

1. Replication and Caching

To handle the increased workload and reduce latency, the following components were implemented:

- In-Memory Cache

We used node cache for data caching

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

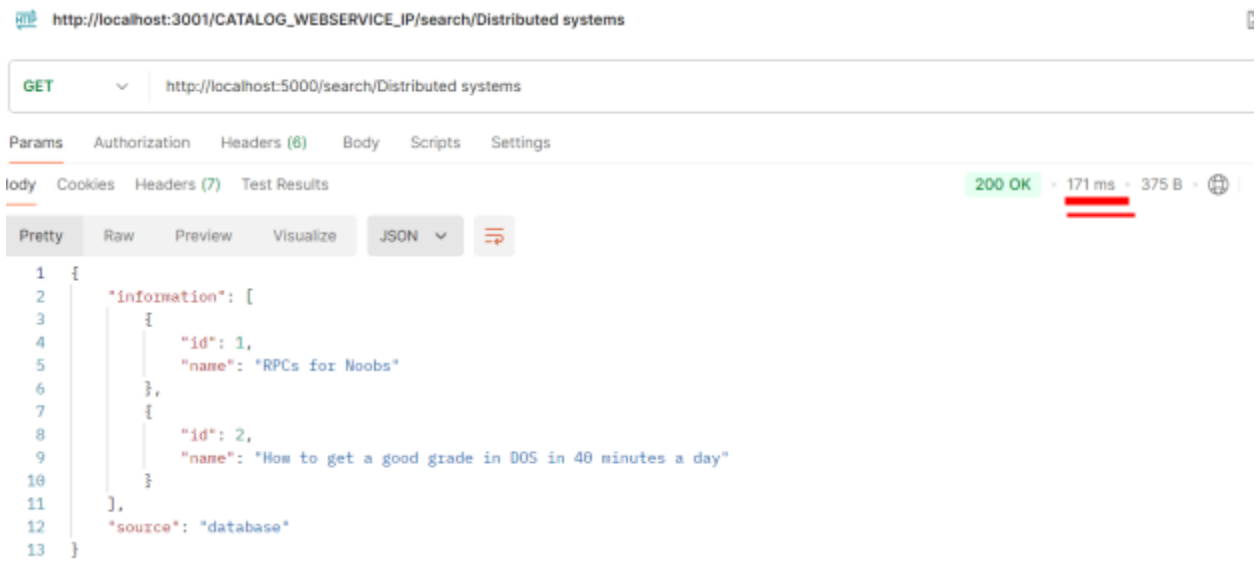
First check the cache for requested data:

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


    const cachedSearch = cache.get(topic);
    if (cachedSearch) {
        console.log('Serving search results from cache for topic:', topic);
        return res.status(200).json({ information: cachedSearch });
    }
```

If the data exist in cache return it else return it from database and write it to the cache for lower latency next time:

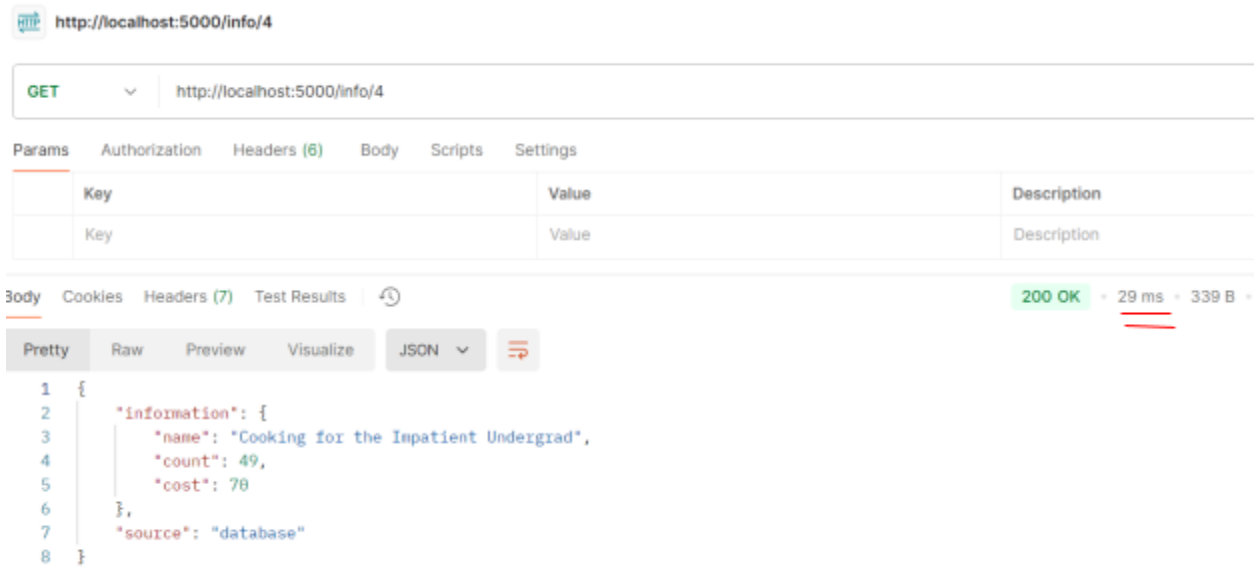
```
   const sqlQuery = 'SELECT id, name FROM Books WHERE topic = ?';
   db.all(sqlQuery, [topic], (err, result) => {
       if (err) {
           console.error("Database query error:", err.message);
           return res.status(500).json({ error: 'Database query failed' });
       }

       if (result && result.length > 0) {

           cache.set(topic, result);
           res.status(200).json({ information: result });
       } else {
           res.status(404).json({ message: 'No books found for the specified topic' })
       }
   });
);
```

The request time without caching for search:

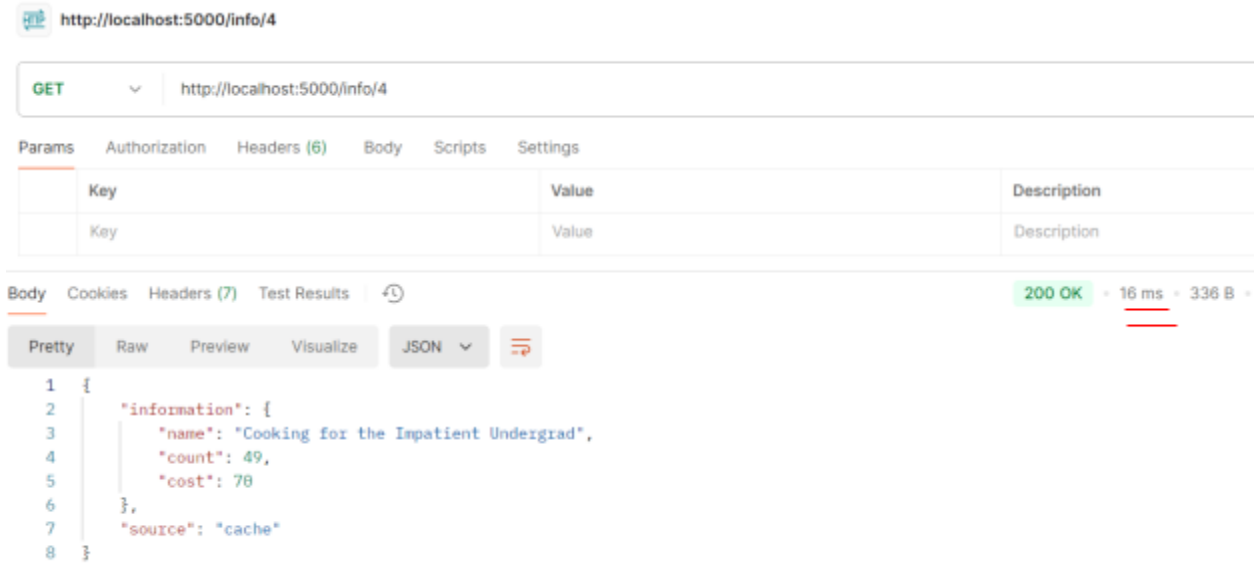


With caching for search:

without cache for info:



With cache for info:



Q1) Compute the average response time (query/buy) of your new systems. What is the response time with and without caching?
Without caching:
For search:171ms.
For info : 29ms.

With caching:
For search: 33ms.
For info: 16ms.

Q2) How much does caching help?
For info: 29/16=1.8125 faster than without using cache.
For search: 171/33=5.18 faster than without using cache.

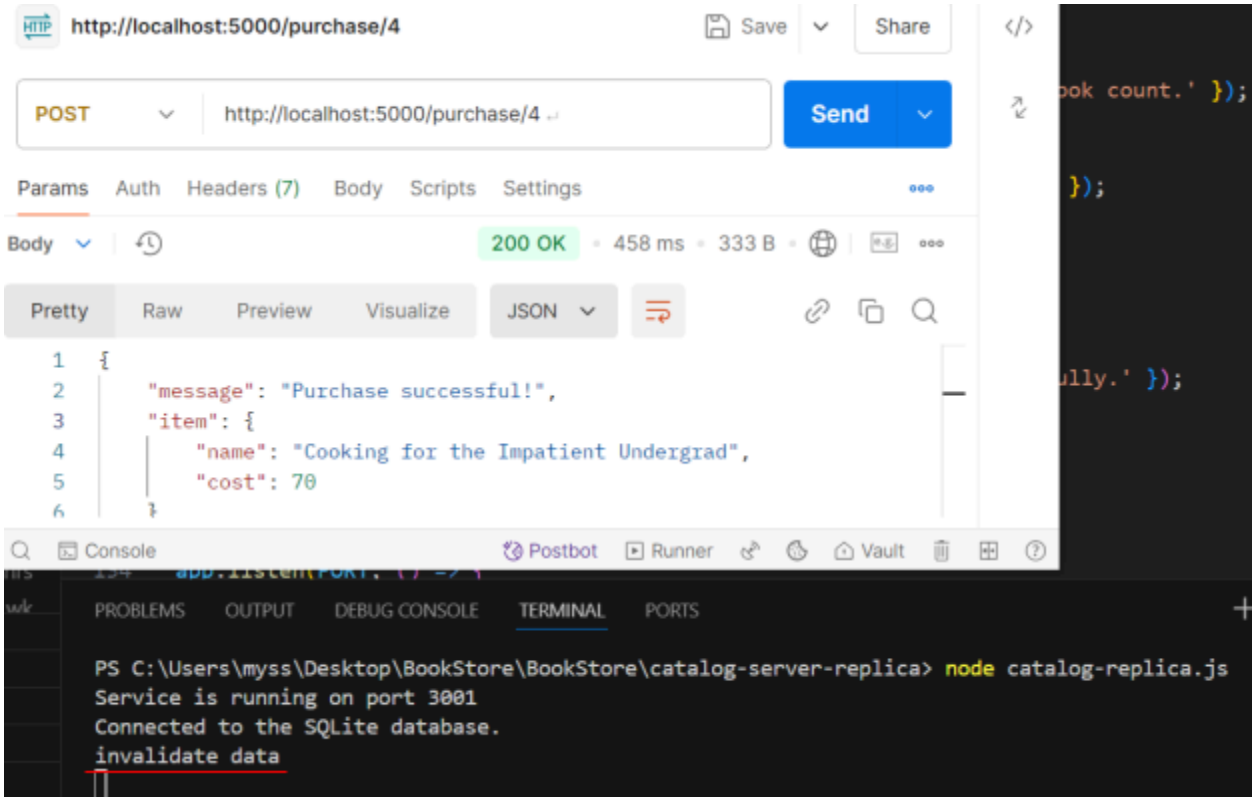
Cache Invalidation and Consistency

- When a backend service (catalog or order server) performs a write operation (e.g.,updateCount api), it sends an invalidate request to the cache to ensure consistency.

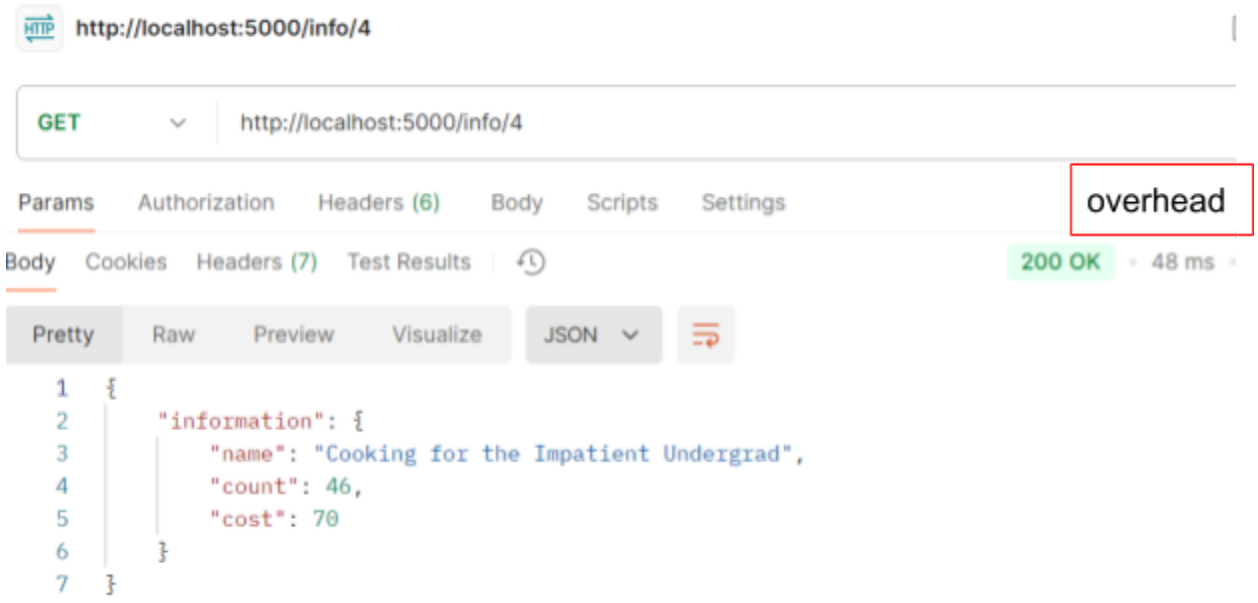
Invalidate data from cache:

```
const sqlQuery = 'UPDATE Books SET count = ? WHERE id = ?';
db.run(sqlQuery, [newCount, bookId], function (err) {
    if (err) {
        console.error("Database update error:", err.message);
        return res.status(500).json({ error: 'Failed to update book count.' });
    }
    if (this.changes === 0) {
        return res.status(404).json({ message: 'Book not found.' });
    }

    // Clear cache for this book since data has changed
    cache.del(bookId);
    console.log("invalidate data");
    res.status(200).json({ message: 'Book count updated successfully.' });
});
```

When data updated from updateCount api the api also invalidate old data stored in cache



Invalidate overhead:



Run experiment	Without cache	With cache	#times cache speed
1	64ms	23ms	2.78
2	49ms	28ms	1.75
3	139ms	66ms	2.1
			avg=2.21 time faster

Replication of Catalog and Order Servers

- Both the Catalog Server (managing book details) and the Order Server (handling purchases) were replicated, with two replicas each to distribute the load and improve fault tolerance.
- Each replica processes a subset of requests, improving throughput.
- Load Balancing: A load balancing algorithm (e.g., **round-robin**) was implemented to distribute incoming requests evenly across the replicas of the catalog and order servers.

The first challenge we faced is database replication and achieve **consistency** between original and replica DB:

Database Replication

• To ensure consistency across the two replicas of the catalog and order servers, an internal protocol was implemented for replicating writes from one server to another, This ensures that any changes to the database in one replica are reflected in the other, keeping the two replicas in sync.

Load balancing (round-robin):

3000: catalog server
3001: catalog-replica server
4000: order server
4000: order-replica server
5000: client server

```
const app = express();
const PORT = process.env.PORT || 5000;

app.use(bodyParser.json());
const catalogServers = ["http://catalog_server:3000", "http://catalog-server-replica:3001"];
let catalogCounter = 0; // Counter to alternate between servers

// Function to get catalog server based on round-robin
function getCatalogServer() {
    const server = catalogServers[catalogCounter % 2]; // Toggle between 0 and 1
    catalogCounter++; // Increment the counter for the next request
    return server;
}

const orderServers = ["http://order_server:4000", "http://order-server-replica:4001"];
let orderCounter = 0; // Counter to alternate between servers

// Function to get catalog server based on round-robin
function getOrderServer() {
    const server2 = orderServers[orderCounter % 2]; // Toggle between 0 and 1
    orderCounter++; // Increment the counter for the next request
    return server2;
}
```

```
app.get('/search/:topic', async (req, res) => {
  const  bookTitle  = req.params.topic; // Get book title from query params
  if (!bookTitle) {
    return res.status(400).json({ error: "Book title is required." });
  }

  try {
    const server=getCatalogServer();
    console.log(server);
    const result = await axios.get(`${server}/CATALOG_WEBSERVICE_IP/search/${bookTitle}`);
    res.json(result.data); // Send back the catalog server response to the client
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\myss\Desktop\BookStore\BookStore\client_server> node app.js
Client server running on port 5000...
http://localhost:3000
http://localhost:3001
http://localhost:3000
```

3000: original catalog server
3001: replica catalog server

---

2. Dockerization

In the second part of the project, the goal was to Dockerize the application to simplify deployment and ensure that the system components could be packaged, shared, and run in isolated containers. The following steps were taken:

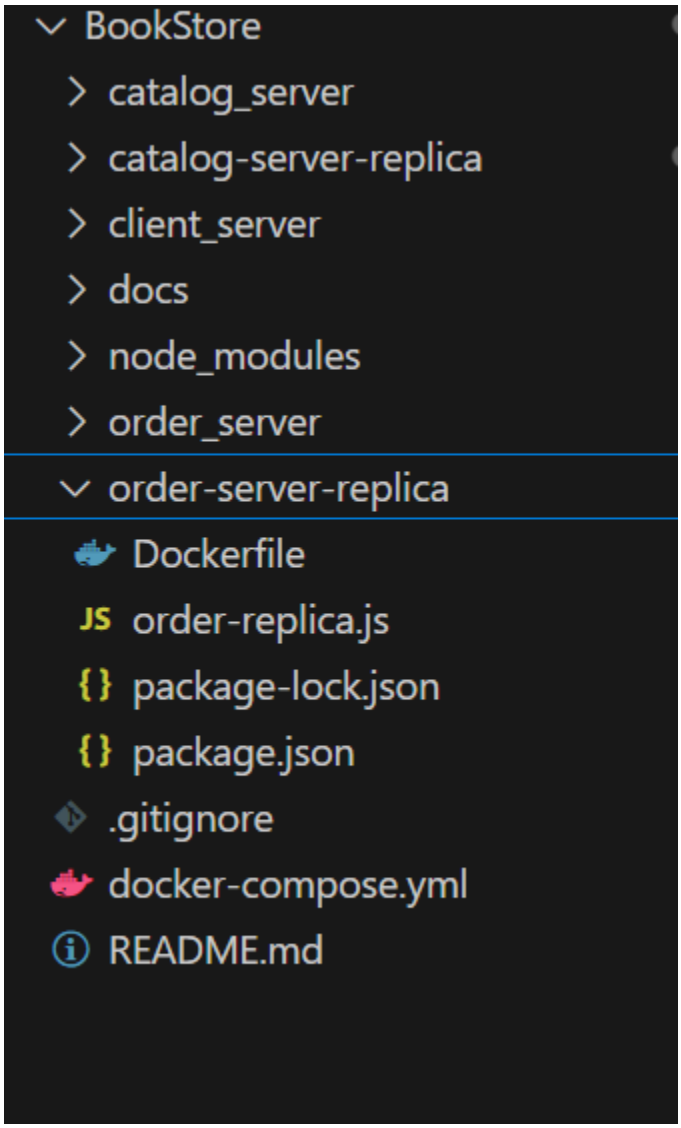
Dockerizing Components

Each of the major components (front-end server, catalog server and its replica, order server and its replica) was containerized using Docker:

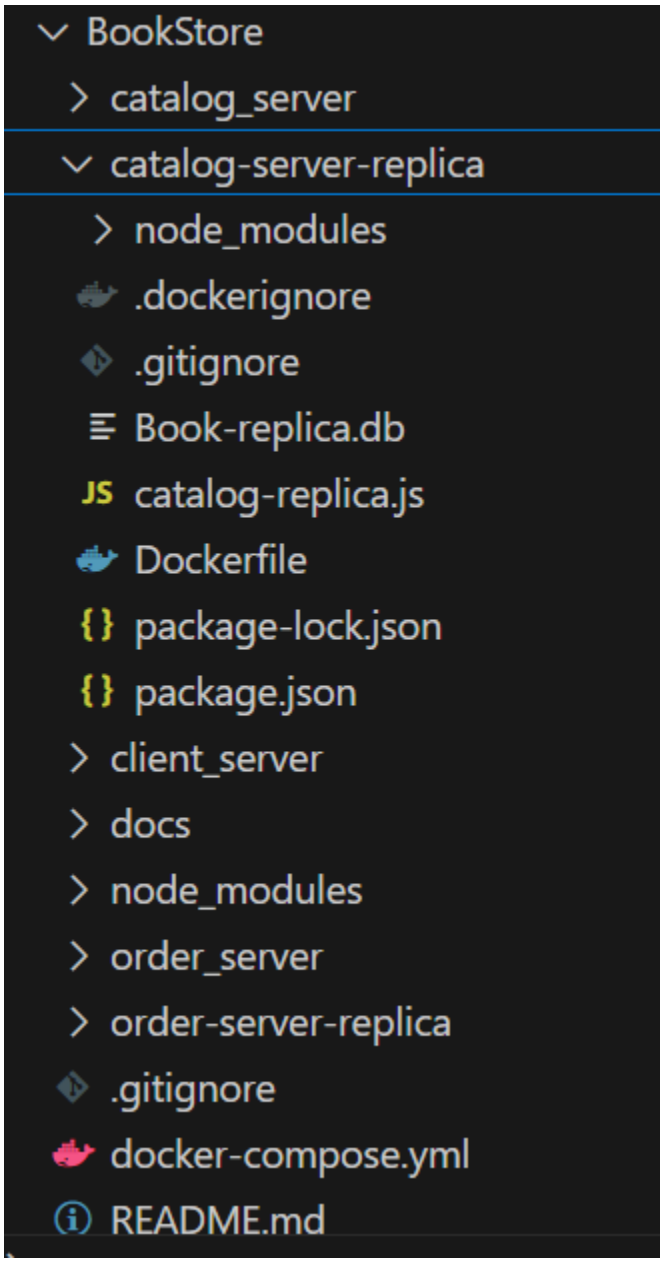
- Each service was packaged into its own Docker container , which includes all necessary dependencies and configurations for running the service.

• The front-end server, catalog server and its replica, order server and its replica were each defined in separate **Dockerfiles**, specifying the environment, dependencies, and commands to run each component.

Server-replica hierarchy:

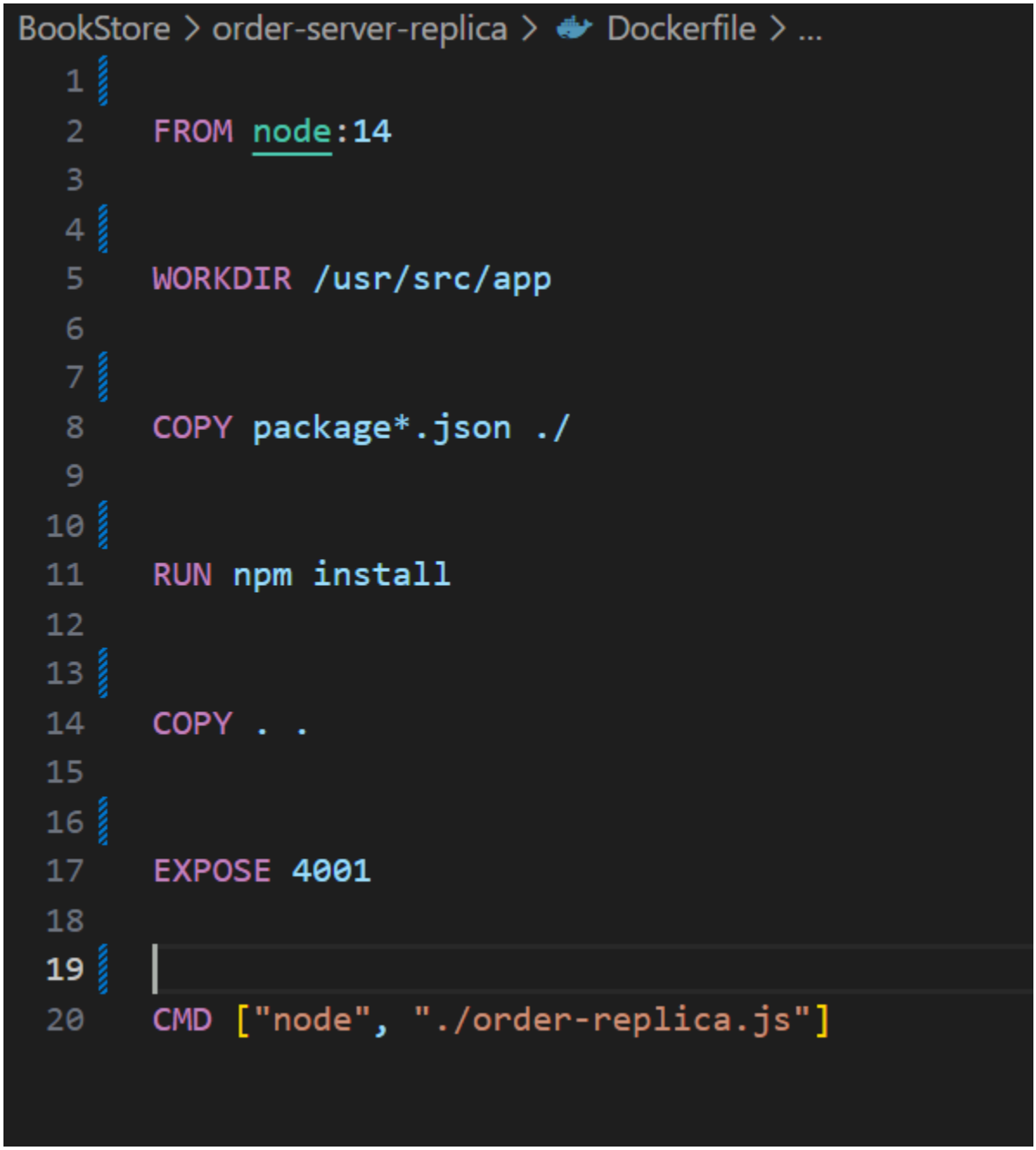


Catalog-replica hirarichy:



catalog-replica dockerfile(port 3001):

Order-replica dockerfile (port 4001):



Docker Compose

• To manage the multi-container setup, Docker Compose was used to define the entire stack in a single configuration file (**docker-compose.yml**).

• This file specifies how each service (front-end, catalog, order, cache) is built, how they communicate with each other, and how ports are exposed for external access.

BookStore > docker-compose.yml

```
3   services:
14    catalog-server-replica:
15      build: ./catalog-server-replica
16      container_name: catimg_replica
17      ports:
18        - "3001:3001"
19      volumes:
20        - ./catalog-server-replica/Book-replica.db:/usr/src/app/Book-replica.db
21      networks:
22        - bazar-network
23
24    order_server:
25      build: ./order_server
26      container_name: orderimg
27      ports:
28        - "4000:4000"
29      depends_on:
30        - catalog_server
31      networks:
32        - bazar-network
33
34    order-server-replica:
35      build: ./order-server-replica
36      container_name: orderimg_replica
37      ports:
38        - "4001:4001"
39      depends_on:
40        - catalog-server-replica
41      networks:
```

```yaml
  client_server:
    build: ./client_server
    container_name: clientimg
    ports:
      - "5000:5000"
    depends_on:
      - catalog_server
      - catalog-server-replica
      - order_server
      - order-server-replica
    networks:
      - bazar-network

networks:
  bazar-network:
    driver: bridge
```

For running the project:

It is the same for part1

- Pull the project from the github
- Go to the directory where docker-compose.yml file in powershell
- Use command **docker compose up –build**