Dos Project Part 2-Bazar.Com

Aya Awwad & Shahad Jawabreh

INTRODUCTION

This report outlines the key features implemented in the second phase of the project, focusing on enhancing performance, scalability, and reliability, While maintaining consistency between Replicas.

Key additions include:

1. In-Memory Cache
2. Server Replicas.
3. Load Balancing Algorithm.
4. Replica Synchronization .

Procedure:

- ### in-memory cache

In-memory cache is a technique where data is temporarily stored in the system's memory to speed up access to frequently requested information. This eliminates the need to repeatedly fetch the same data from external sources, improving performance.

- **Cache Object** (`inMemoryCache`) stores search results and book details.

```
const inMemoryCache = {};
```

- Before making an API request, it checks if the data is already in the cache , If data is found, it's displayed immediately.

```
if (inMemoryCache[`search_${topic}`]) {
    displaySearchResults(inMemoryCache[`search_${topic}`]);
    return;
}
```

```
if (inMemoryCache[`info_${bookId}`]) {
    displayBookInfo(inMemoryCache[`info_${bookId}`]);
    return;
}
```

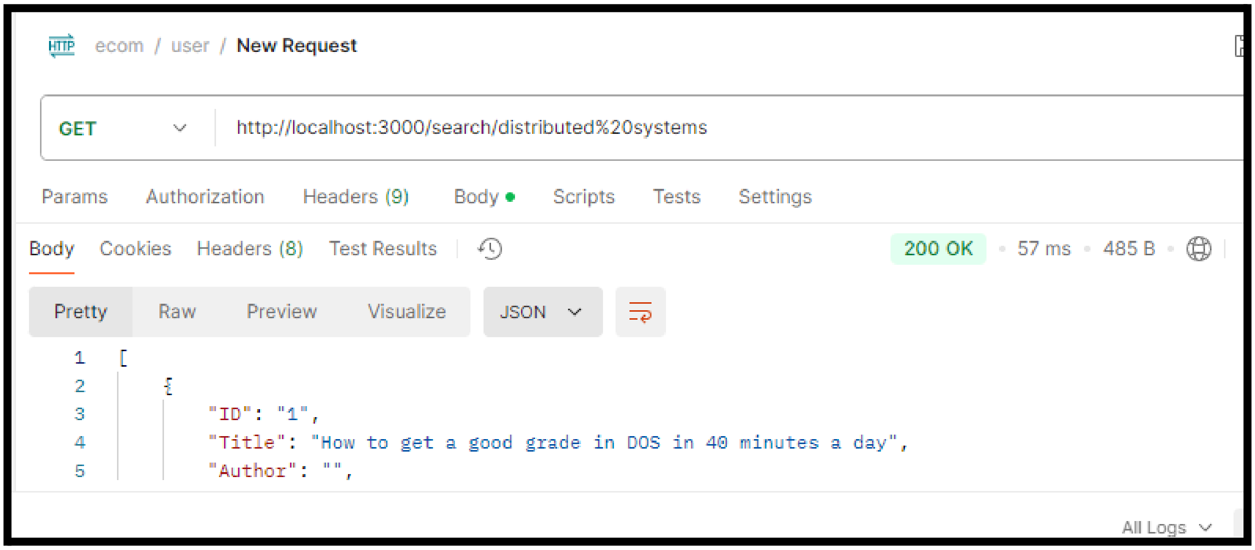
- if not found, a server request is made, and the results are stored in the cache for future use.

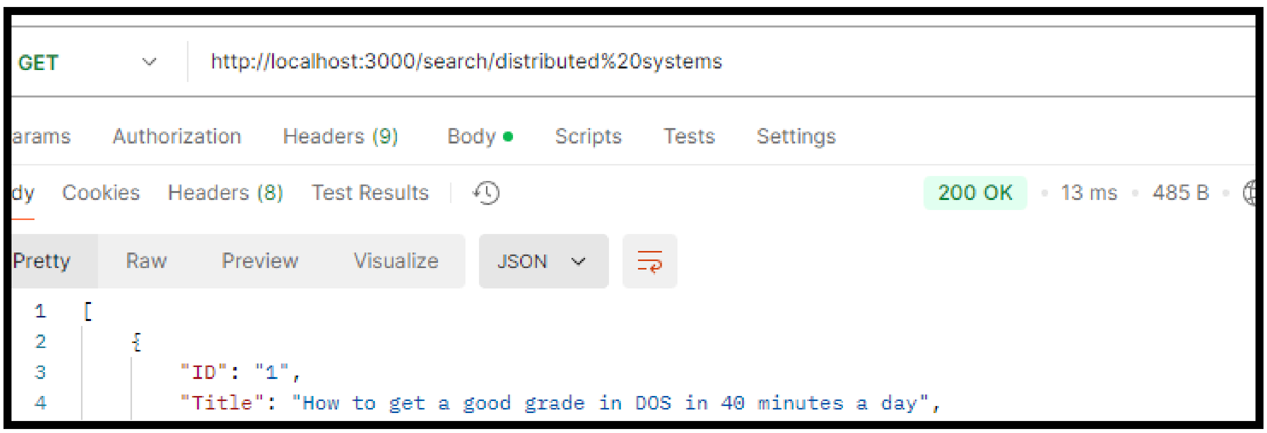
```
fetch(`${getCatalogServer()}/search/${topic}`)
    .then(response => response.json())
    .then(data => {
        inMemoryCache[`search_${topic}`] = data;

        const resultsList = document.getElementById('search-results');
        resultsList.innerHTML = '';
        data.forEach(book => {
            const listItem = document.createElement('li');
            listItem.className = 'list-group-item';
            listItem.textContent = `${book.Title} (ID: ${book.ID})`;
            resultsList.appendChild(listItem);
        });
    })
```

```
    fetch(`${getCatalogServer()}/info/${bookId}`)
        .then(response => response.json())
        .then(data => {
            inMemoryCache[`info_${bookId}`] = data;
            displayBookInfo(data);
        })
        .catch(error => console.error('Error fetching book info:', error));
}
```

- When a book is purchased or unpurchased, the corresponding cache data is deleted.

```
function purchaseBook() {
    const bookId = document.getElementById('purchase-book-id').value;
    if (!bookId) {
        alert('Please enter a book ID.');
        return;
    }

    fetch(`${getOrderServer()}/purchase/${bookId}`, { method: 'POST' })
        .then(response => response.text())
        .then(result => {
            delete inMemoryCache[`info_${bookId}`];
            const purchaseResultDiv = document.getElementById('purchase-result');
            purchaseResultDiv.textContent = result;
        })
        .catch(error => {
```





- **First request (first picture)**: Cache miss → Reads from the original source (request time : 57ms)
- **Following requests (second picture)**: Cache hit → Reads from memory (request time : 13ms), saving time by avoiding repeated database or file access.

- ## Load Balancing Algorithm.

A load balancing algorithm has been implemented to distribute requests evenly across multiple server replicas. We chose the **Round Robin algorithm** for its simplicity and efficiency.In this setup, when performing a search operation using the catalog server, the load is spread across the available catalog server replicas using the round-robin method.The same approach is applied when retrieving books from the database and when processing book purchases through the order server.

```
</div>
<script>
    const catalogServers = ['http://localhost:3000', 'http://localhost:3002'];
    const orderServers = ['http://localhost:3001', 'http://localhost:3003'];

    let catalogServerIndex = 0;
    let orderServerIndex = 0;

    function getCatalogServer() {
        const server = catalogServers[catalogServerIndex];
        catalogServerIndex = (catalogServerIndex + 1) % catalogServers.length;
        return server;
    }

    function getOrderServer() {
        const server = orderServers[orderServerIndex];
        orderServerIndex = (orderServerIndex + 1) % orderServers.length;
        return server;
    }
```

- ## Server Replicas.

  Replicas of each server were created as you can see here.

docker-compose	11/2/2024 9:45 PM
order-service-rep	11/2/2024 9:19 PM
catalog-service-rep	11/2/2024 9:19 PM
frontend-service	11/2/2024 2:31 PM
.git	10/27/2024 10:38 PM
catalog-service	10/27/2024 9:55 PM
order-service	10/27/2024 9:16 PM

Below Docker Compose file sets up a small book store application with a few part

- **Catalog Services**: Two containers (catalog-service and its replica catalog-service-rep) that manage book listings. Each one is accessible on different ports (3000 and 3002).
- **Order Services**: Two containers (order-service and its replica order-service-rep) that handle book orders. They run on ports 3001 and 3003 and wait until the catalog services are running.
- **Frontend**: A container (frontend-service) that the user interacts with through a web browser on port 80. It depends on the catalog and order services to be ready before it starts.
- **Network**: All services are connected through a shared network (book_store_network) to communicate with each other.

docker-compose.yml

```yaml
version: '3'
services:
  catalog-service:
    build: ./catalog-service
    ports:
      - "3000:3000"
    networks:
      - book_store_network
    volumes:
      - /catalog-service

  catalog-service-rep:
    build: ./catalog-service-rep
    ports:
      - "3002:3002"
    networks:
      - book_store_network
    volumes:
      - /catalog-service-rep

  order-service:
    build: ./order-service
    ports:
      - "3001:3001"
    networks:
      - book_store_network
    depends_on:
      - catalog-service
      - catalog-service-rep
```

docker-compose.yml

```yaml
services:
  order-service-rep:
    build: ./order-service-rep
    ports:
      - "3003:3003"
    networks:
      - book_store_network
    depends_on:
      - catalog-service
      - catalog-service-rep

  frontend-service:
    build: ./frontend-service
    ports:
      - "80:80"
    networks:
      - book_store_network
    depends_on:
      - catalog-service
      - order-service
      - catalog-service-rep
      - order-service-rep

networks:
  book_store_network:
    driver: bridge
```

By running docker-compose up, all services will be built, networked, and started automatically

C:\Windows\System32\cmd.exe

```
(c) Microsoft Corporation. All rights reserved.

C:\Users\A.B\Desktop\bazar-bookstore>docker ps
CONTAINER ID   IMAGE                                  COMMAND                  CREATED          STATUS          PORTS
  NAMES
65345a335f29   bazar-bookstore-frontend-service       "/docker-entrypoint.…"   21 seconds ago   Up 17 seconds   0.0.0.0:80->80/tcp
  bazar-bookstore-frontend-service-1
37f2d725e89b   bazar-bookstore-order-service          "docker-entrypoint.s…"   21 seconds ago   Up 18 seconds   0.0.0.0:3001->3001/tcp
  bazar-bookstore-order-service-1
f55ff3a4719c   bazar-bookstore-order-service-rep      "docker-entrypoint.s…"   21 seconds ago   Up 18 seconds   0.0.0.0:3003->3003/tcp
  bazar-bookstore-order-service-rep-1
20d96a5ef967   bazar-bookstore-catalog-service-rep    "docker-entrypoint.s…"   22 seconds ago   Up 19 seconds   0.0.0.0:3002->3002/tcp
  bazar-bookstore-catalog-service-rep-1
0c42daa0d7be   bazar-bookstore-catalog-service        "docker-entrypoint.s…"   22 seconds ago   Up 19 seconds   0.0.0.0:3000->3000/tcp
  bazar-bookstore-catalog-service-1
```

- **Replica Synchronization**
  The synchronization in this setup is triggered only when an item is purchased:
  - calls **syncStockAcrossReplicas** function, which updates all replicas with the new stock level, ensuring consistency across the system.

```
// Sync stock across catalog replicas
async function syncStockAcrossReplicas(bookId, newStock) {
    for (const url of catalogReplicas) {
        try {
            console.log(newStock)
            await axios.post(`${url}/sync-stock/${bookId}`, { Stock: newStock });
            console.log(`Synced stock to ${url} for Book ID ${bookId}`);
        } catch (error) {
            console.error(`Error syncing stock to ${url}:`, error);
        }
    }
}
```

Conclusion :

In this phase of the project, implementing an **In-Memory Cache** has been a key factor in enhancing overall performance by drastically reducing data retrieval times for frequently accessed data, thus making responses faster and more efficient. Paired with this, we introduced **Server Replicas** to ensure high availability and redundancy, allowing for distributed load across multiple instances. A **Load Balancing Algorithm** was also incorporated, directing traffic across replicas to prevent overloading any single server and to maintain smooth operation even during high-demand periods. To maintain data accuracy and consistency across replicas, a **Replica Synchronization** mechanism was implemented, ensuring that any updates (such as stock changes) propagate reliably throughout the system. Together, these enhancements have not only boosted performance and scalability but also strengthened the system's reliability and consistency, creating a solid foundation for future growth.