An-Najah National University

Faculty of Engineering

Computer Engineering Department

Distributed and Operating Systems

Instructor: Dr. samer arandi

Studants:

حلا فائق حبش 12112591

لؤى محمد عوايص 12111992

Lab 2: Turning the Bazar into an Amazon: Replication, Caching, and Consistency

Introduction:

In Lab 2, we extend the Bazar.com architecture used in Lab 1 with essential performance and scalability features. These include in-memory caching, backend replication, and cache consistency mechanisms. The objective here is to reduce latency, scatter loads, and ensure data consistency and improve the system throughput during peak traffic.

System Architecture:

The system continues to follow a two-tier microservices architecture with a small extension to support replication. It consists of the following entities:

Front-End Service: Provides the interface layer for communication between the user and the back-end services. It handles client requests and directs them to the appropriate service replica based on a round-robin load balancing policy. It also has an in-memory cache to reduce response time for read queries.

Catalog Service (Replicated): Maintains book stock, prices, and categories. In this version, it is replicated on two instances for load balancing. The replicas are written to synchronously to maintain consistency.

Order Service (Replicated): Accepts purchase orders and subtracts from inventory levels (request is forwarded to catalog). Like the Catalog Service, it is replicated and uses internal synchronization to maintain consistency between replicas.

System Enhancements Overview:

We used the following ports for each service:

- → Client (front-end) service on port 9000
- → Catalog1 service on port 9001
- → Catalog2 service on port 9002
- → Order1 service on port 9003
- → Order2 service on port 9004

Use php -S localhost:9000 -t public to run the code on the terminal

In-Memory Caching:

To reduce latency and minimize redundant communication with the backend Catalog Service, an in-memory caching mechanism was added to the front-end service (ClientController).

Caching Strategy: The cache stores the result of read-only requests, such as:

- Book searches by title
- Book details by ID

Write operations, such as purchases and stock updates, bypass the cache and trigger cache invalidation to preserve data accuracy.

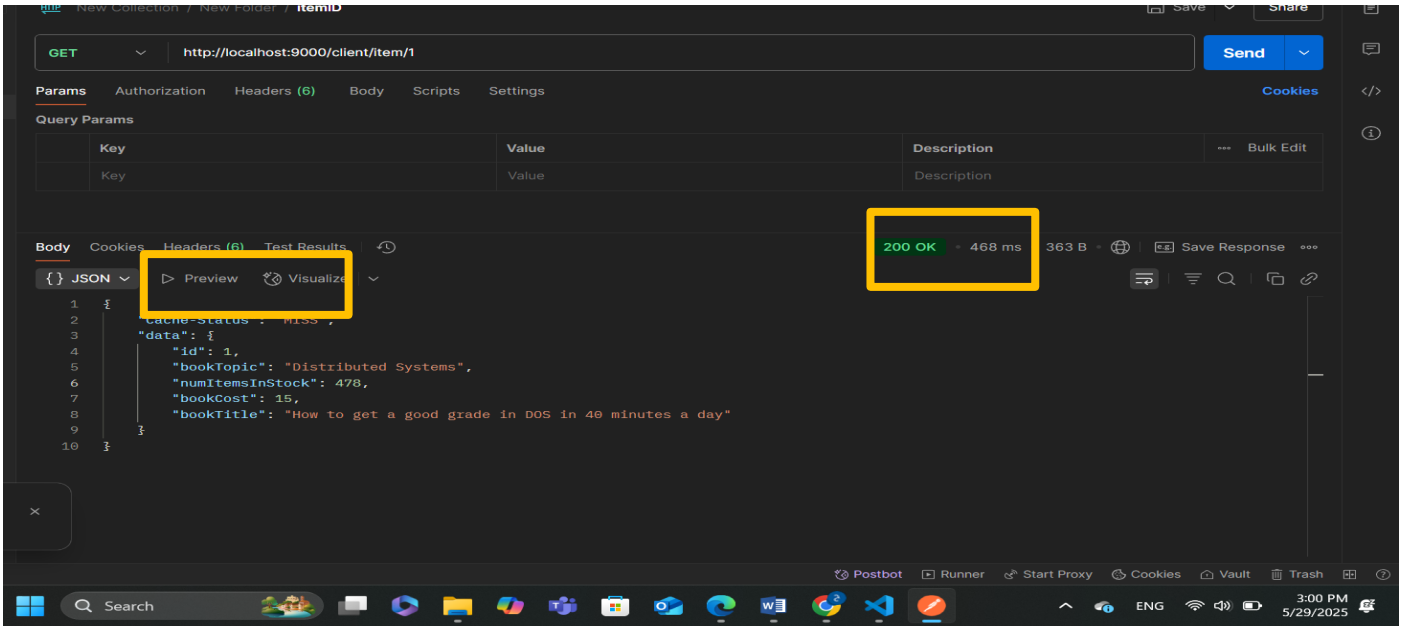
How It Works:

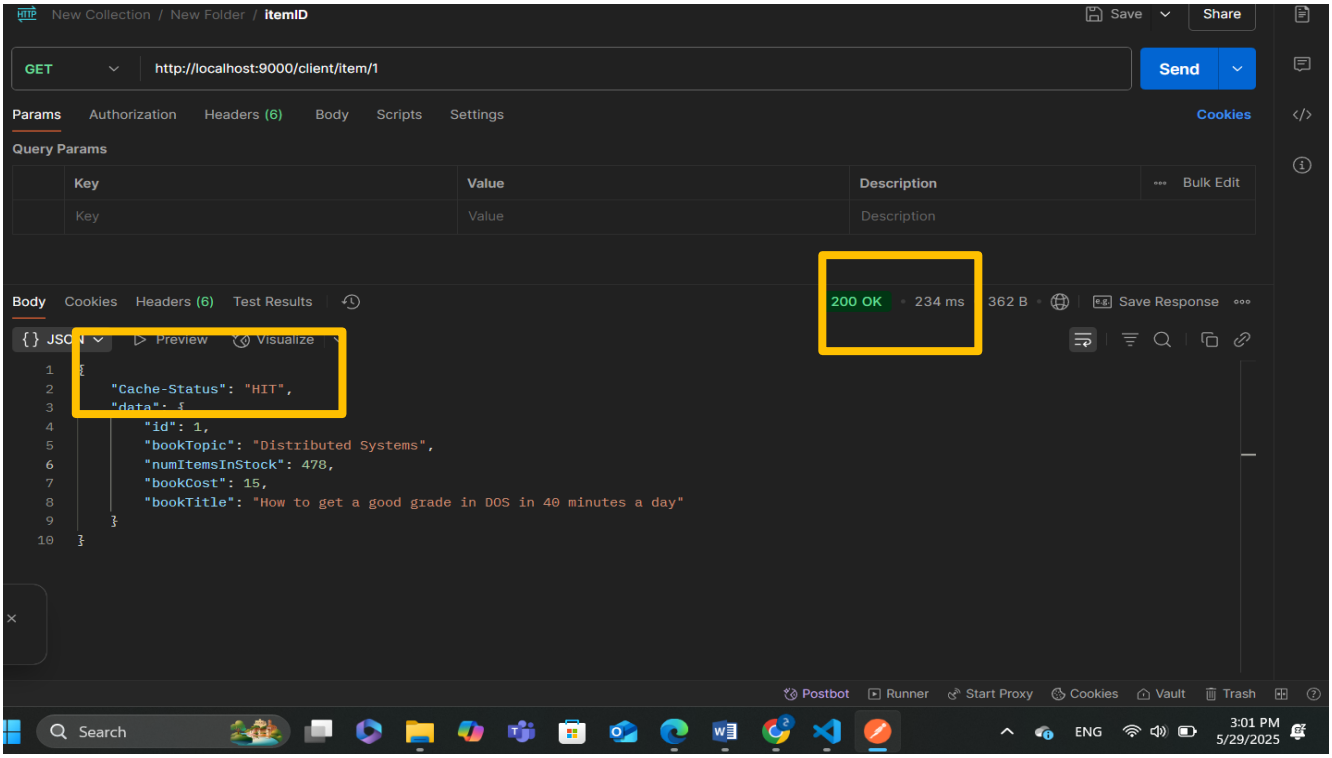
For every read request, the front-end First checks the cache. If a cache hit occurs, the cached response is returned immediately with Cache-Status: HIT. If a cache miss occurs, the request is sent to the next catalog replica, the response is stored in the cache, and returned with Cache-Status: MISS.

Cached data is stored for 1 minute. When a book update occurs (via PUT and POST request), the corresponding cache entry is invalidated.

Method: **GET**
URL: http://localhost:8000/client/item/1

- We can see from the photos that the time for the **Cache Miss** is **468ms**, while the **Cache Hit** took **234ms.** We can see that caching reduces latency, reduces the backend service load, and enhances the overall scalability of the system by executing repeated read requests directly from memory.





Replication and Load Balancing:

For increasing the **scalability** and **reliability** of the system, **Order Service** and **Catalog Service** have been replicated, and **the round-robin load balancing method** is implemented in the front-end **ClientController.**

How It Works:
The **Catalog Service**: **http://localhost:9001/catalog** and **http://localhost:9002/catalog2**, and the **Order Service** is at **http://localhost:8003/order** and **http://localhost:8004/order2**. Two static round-robin index counters, **currentCatalogIndex** for **catalog** requests and **currentOrderIndex** for **order** requests, are used by the front-end service for balancing traffic. These static counters are preserved across several requests within the same PHP process and stored in the cache to remember their value for the next request. For each request coming in, the front-end selects the appropriate replica depending on the current index and increments the index via modulo arithmetic to achieve uniform distribution of requests among replicas.

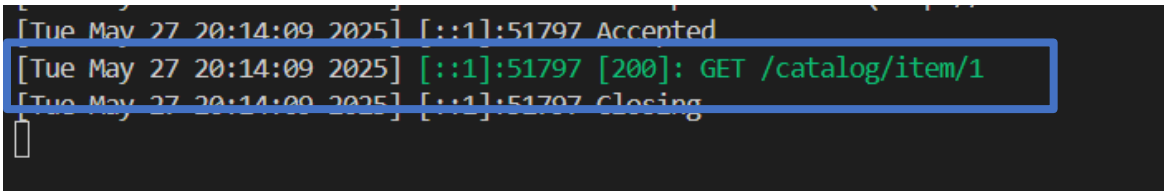
Replication and load balancing enhance performance by distributing requests among various replicas, increasing throughput, and preventing bottlenecks. They also provide fault tolerance by allowing services to continue functioning in the event of a failed replica, and they improve scalability by facilitating the addition of new replicas with minimal URL configuration.

Method: **GET**
URL: http://localhost:9000/client/item/1

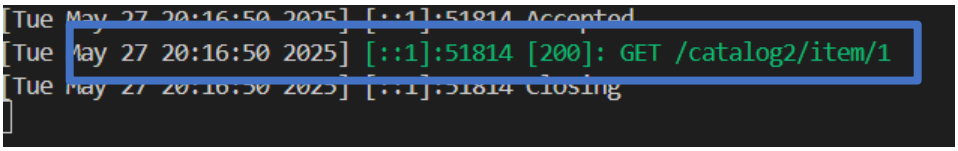
Client (front end) service requests item 1 on port 9000, after waiting a reasonable time to remove the cached data (Data expired), then requests it again, it will request from the other catalog server.

```
[Tue May 27 20:14:09 2025] [::1]:51795 Closing
[Tue May 27 20:14:09 2025] [::1]:51796 [200]: GET /client/item/1
[Tue May 27 20:14:09 2025] [::1]:51796 Closing
[Tue May 27 20:16:50 2025] [::1]:51812 Accepted
[Tue May 27 20:16:50 2025] [::1]:51813 Accepted
[Tue May 27 20:16:50 2025] [::1]:51812 Closed without sending a request; it was probably just an unused speculative preconnection
[Tue May 27 20:16:50 2025] [::1]:51812 Closing
[Tue May 27 20:16:50 2025] [::1]:51813 [200]: GET /client/item/1
[Tue May 27 20:16:50 2025] [::1]:51813 Closing
```

→ First request forwarded to replica #1 =>catalog



→ Second request forwarded to replica #2 =>catalog2



Cache Consistency and Synchronization:

To serve correct and updated data from the **in-memory cache**, there is a cache **invalidation** policy is implemented in the **ClientController**. This is required to establish strong consistency between the cached data and the replicated backend databases, especially after write operations.

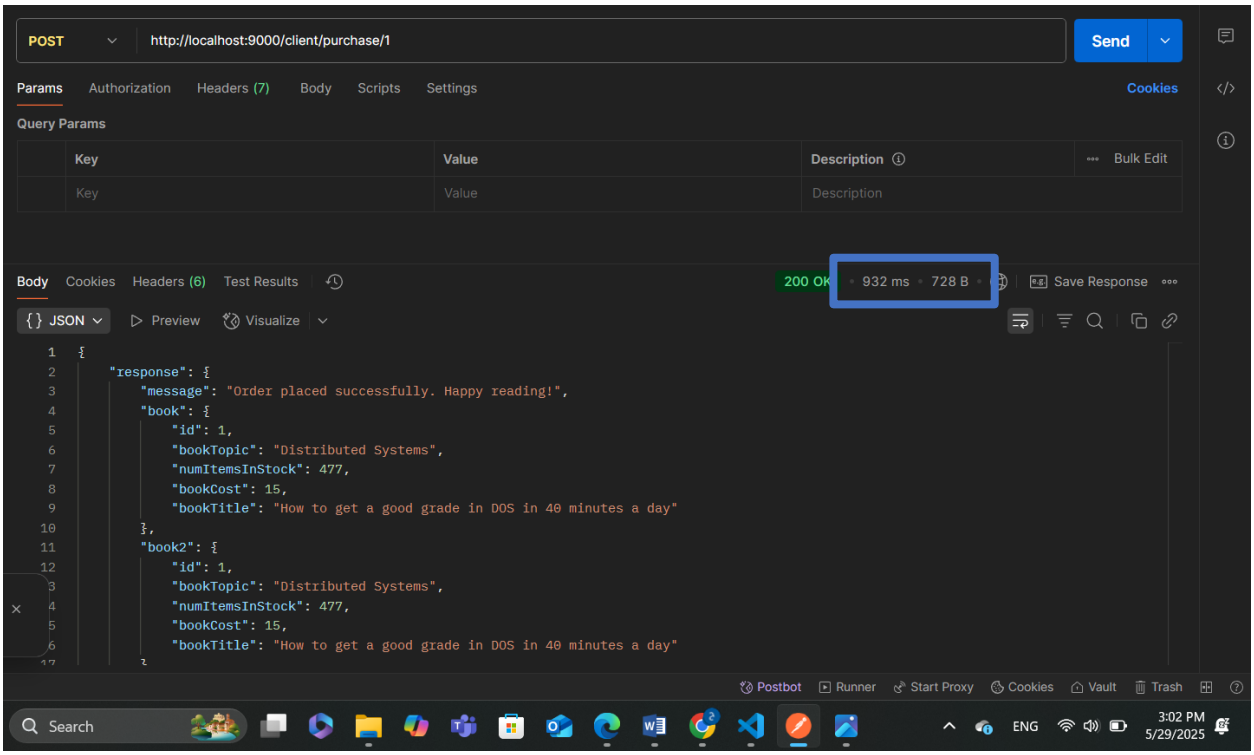
How It Works:

Whenever there is a **write** operation (for example, when the stock or data of a book is being changed), the **front-end** calls the **invalidateItemCache()** function. After successfully receiving a response from the backend for the update or buy request, the system automatically **invalidates** the corresponding **cache** entry by calling **$this->invalidateItemCache($requestPath).**

This removes the old entry from memory so future read requests receive updated information from the updated catalog server.

This maintains the accuracy and reliability of the application while still taking advantage of caching for improved performance on read operations.

Consistency example1
Method: **POST**
URL: http://localhost:9000/client/purchase/1



Database updates consistency

```
PS C:\Users\Dell\Desktop\NNU\DOS\docker\Dos\Bazar\public> sqlite3 database.db
SQLite version 3.49.1 2025-02-18 13:38:58
Enter ".help" for usage hints.
sqlite> .mode column
sqlite> .headers on
sqlite> SELECT * FROM bookCatalog;
id  bookTopic            numItemsInStock  bookCost  bookTitle
--  -------------------  ---------------  --------  --------------------------------------------------
1   Distributed Systems  479              15        How to get a good grade in DOS in 40 minutes a day
2   Distributed Systems  15               20        RPCs for Noobs.
3   Undergraduate School 39               35        Xen and the Art of Surviving Undergraduate School
4   Undergraduate School 20               25        Cooking for the Impatient Undergrad
sqlite>
```

```
PS C:\Users\Dell\Desktop\NNU\DOS\docker\Dos\Bazar\public> sqlite3 databaseCopy.db
SQLite version 3.49.1 2025-02-18 13:38:58
Enter ".help" for usage hints.
sqlite> .mode column
sqlite> .headers on
sqlite> SELECT * FROM bookCatalog;
id  bookTopic            numItemsInStock  bookCost  bookTitle
--  -------------------  ---------------  --------  --------------------------------------------------
1   Distributed Systems  479              15        How to get a good grade in DOS in 40 minutes a day
2   Distributed Systems  15               20        RPCs for Noobs.
3   Undergraduate School 39               35        Xen and the Art of Surviving Undergraduate School
4   Undergraduate School 20               25        Cooking for the Impatient Undergrad
sqlite>
```

What is the overhead of cache consistency operations?

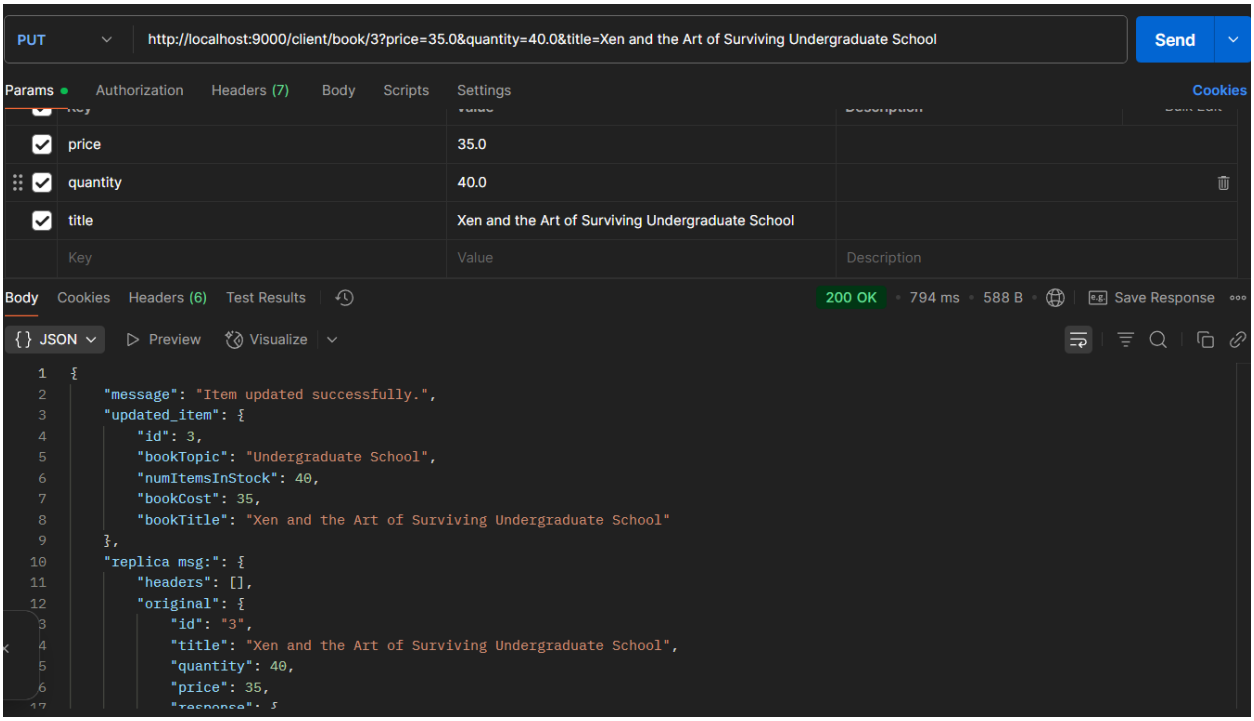
Requests must be sent to all replicas to ensure the same data at all databases. In this application request, forwarded from client server -> order server-> catalog server -> to all replicas (catalog2 in this case).

There is a timing overhead. Extra time needed to get data from the database. Ensures up-to-date data but slows down the next read operation.

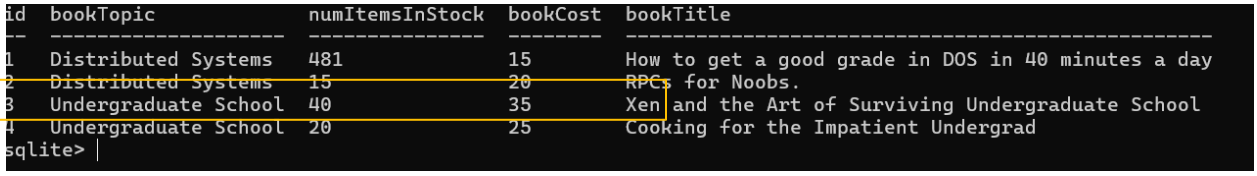
Consistency example2

Method: **PUT**

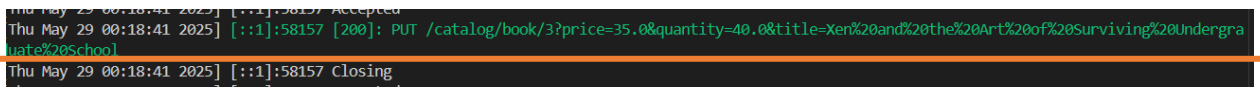
URL: http://localhost:9000/client/book/3?price=35.0&quantity=40.0&title=Xen and the Art of Surviving Undergraduate School



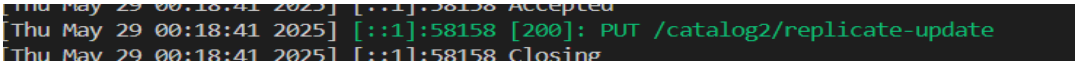
Both databases:



Catalog1 update

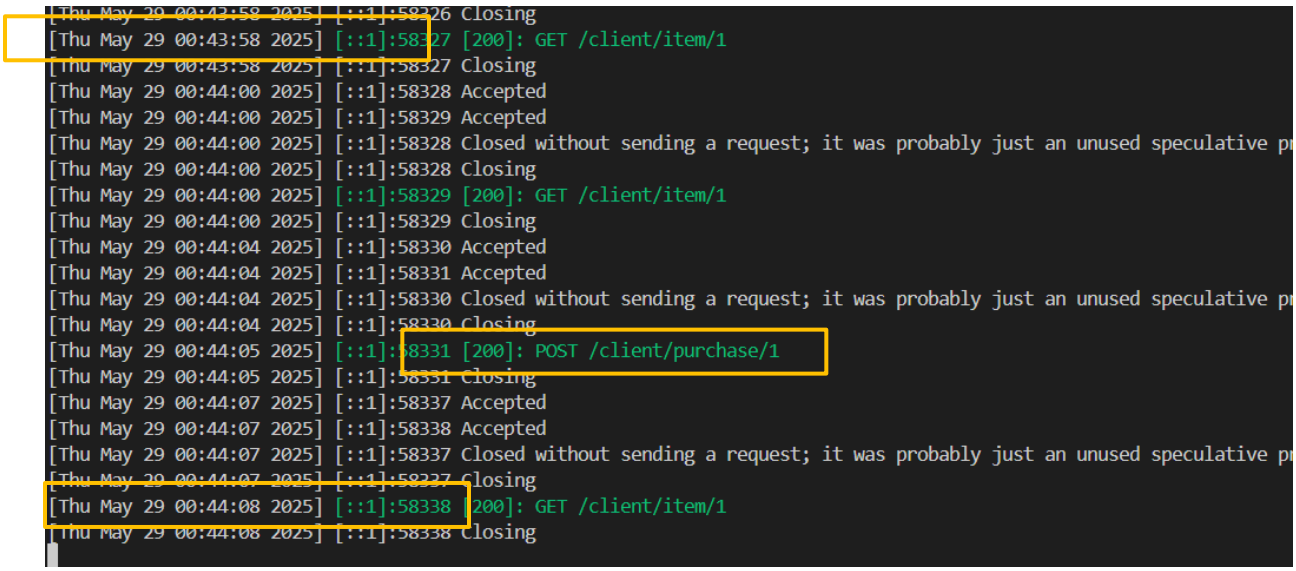


Catalog 1 sends an update to Catalog 2

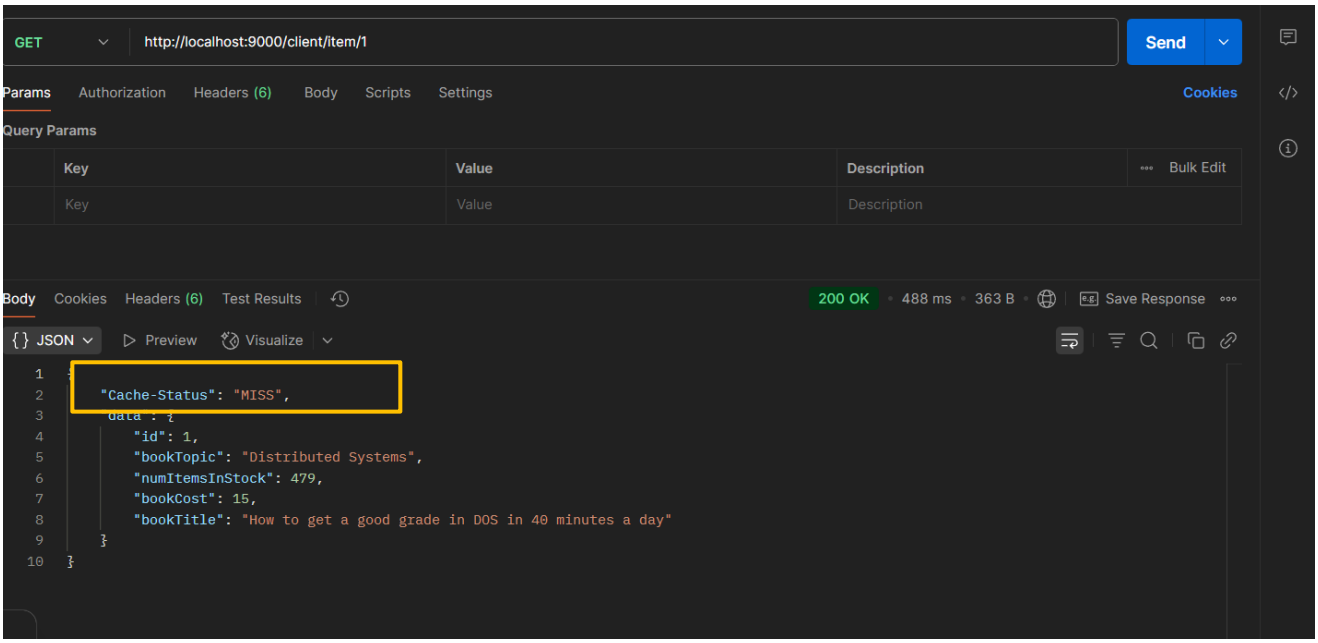


Update cache example:

The invalidate request causes the data for that item to be removed from the cache



Notice that in less than a minute, when trying to get the item/1 information, the request is sent again after being removed from the cache. See cache Miss in the second get request



From previous examples, we can conclude the following table :

	Avg Latency (ms)	operation
Query (No Cache)	463 ms	Fetches from DB
Query (Cached)	234 ms	Served from cache
Purchase (Write)	932 ms	Updates DB + invalidates cache
Update (Write)	794 ms	Updates DB + invalidates cache

Conclusion:

This lab enhanced the Bazar.com system with replication, caching, load balancing, and cache consistency. Performance and fault tolerance were improved with replication and round-robin load balancing, and response time and backend load were reduced with in-memory caching. Data consistency upon updating was ensured by cache invalidation. The system is now more responsive, scalable, and suitable for high traffic.

Postman Documentation Link:

https://documenter.getpostman.com/view/44834036/2sB2qfAeTM