

DOS Project Report Part 2

Student Name: Lama Qawareeq

Student Name: Anwar Aqraa

Part1: Cache Consistency

```
// Search books by topic
app.get('/search/:topic', async (req, res) => {
  const { topic } = req.params;

  // Check cache
  if (cache[`search-${topic}`]) {
    console.log(`Cache hit for search topic "${topic}"`);
    return res.json({ data: cache[`search-${topic}`], server: 'cache' });
  }

  const server = chooseServer(catalogServers);

  try {
    const response = await axios.get(`${server}/search/${topic}`);
    cache[`search-${topic}`] = response.data; // save response
    console.log(`Cache updated for search topic "${topic}"`);

    res.json({ data: response.data, server: server.includes('replica') ? 'replica' : 'primary' });
  } catch (error) {
    console.error(`Error fetching search results for topic "${topic}":`, error.message);
    res.status(500).send('Error fetching search results');
  }
});
```

When the data is in the cache

When the result is not in the cache, it is added to the cache.



When I send GET request the first time, before Caching the data:

The screenshot shows a Postman interface for a GET request to `http://localhost:3000/info/1`. The response status is 200 OK, and the response time is 54 ms, which is circled in blue with a blue arrow pointing to it. The response body is in JSON format, showing a single object with fields like title, stock, price, topic, and server.

```
{
  "data": {
    "title": "How to get a good grade in DOS in 40 minutes a day",
    "stock": 7,
    "price": 100,
    "topic": "distributed systems"
  },
  "server": "primary"
}
```

Activate Windows
Go to Settings to activate Windows

The screenshot shows a Postman interface for a GET request to `http://localhost:3000/search/distributed systems`. The response status is 200 OK, and the response time is 11 ms, which is circled in green with a green arrow pointing to it. The response body is in JSON format, showing an array of objects with fields like id and title, along with a server field.

```
{
  "data": [
    {
      "id": 1,
      "title": "How to get a good grade in DOS in 40 minutes a day"
    },
    {
      "id": 2,
      "title": "RPCs for Noobs"
    }
  ],
  "server": "primary"
}
```

Activate Windows
Go to Settings to activate Windows

Q1) Compute the average response time (query/buy) of your new systems.

What is the response time with and without caching?

Answers :

for info: 54 ms

for search: 11 ms



With Cache:

Postman interface showing a GET request to `http://localhost:3000/info/1`. The response status is 200 OK, and the response time is 11 ms, which is highlighted with a green box and a green arrow. The response body is displayed in JSON format:

```
1 {
2   "data": {
3     "title": "How to get a good grade in DOS in 40 minutes a day",
4     "stock": 7,
5     "price": 100,
6     "topic": "distributed systems"
7   },
8   "server": "cache"
9 }
```

Postman interface showing a GET request to `http://localhost:3000/search/distributed systems`. The response status is 200 OK, and the response time is 5 ms, which is highlighted with a green box and a green arrow. The response body is displayed in JSON format:

```
1 {
2   "data": [
3     {
4       "id": 1,
5       "title": "How to get a good grade in DOS in 40 minutes a day"
6     },
7     {
8       "id": 2,
9       "title": "RPCs for Noobs"
10    }
11  ],
12  "server": "cache"
13 }
```

Q2) How much does caching help?

Answers:

for info: 11ms, 54/11 \rightarrow 4.9 Faster than without cache.

for search: 5ms, 11/5 \rightarrow 2.2 Faaster than without using cache.



Invalidate

When purchasing, the data is disabled in cash until the new data values are taken after updating the inventory.

```
1 // Purchase a book
2 app.post('/purchase/:item_number', async (req, res) => {
3   const { item_number } = req.params;
4
5   const server = chooseServer(orderServers, true); // Always send purchase to order service
6
7   try {
8     const response = await axios.post(`${server}/purchase/${item_number}`);
9
10    // Update cache: Decrease stock in the cache
11    if (cache[`info-${item_number}`]) {
12      cache[`info-${item_number}`].stock -= 1;
13      console.log(`Cache updated for book "${item_number}", new stock: ${cache[`info-${item_number}`].stock}`);
14    }
15
16    res.json({ data: response.data, server: server.includes('replica') ? 'replica' : 'primary' });
17  } catch (error) {
18    console.error('Error processing purchase for item number "${item_number}":', error.message);
19    res.status(500).send('Error processing purchase');
20  }
21 });
```

The code in (catalog & catalog replica) updates the stock of a specific item in the database by decreasing it by 1, then removes the cached data for that item if it exists. The goal is to ensure that the cached data stays consistent with the database after any changes.

```
app.post('/invalidate', (req, res) => {
  if (!item_number) {
    return res.status(400).json({ error: 'item_number is required' });
  }

  // First, update stock in the database
  db.get("SELECT stock FROM books WHERE id = ?", [item_number], (err, row) => {
    if (err) {
      console.error('Error fetching book from database:', err);
      return res.status(500).send('Error updating stock');
    }

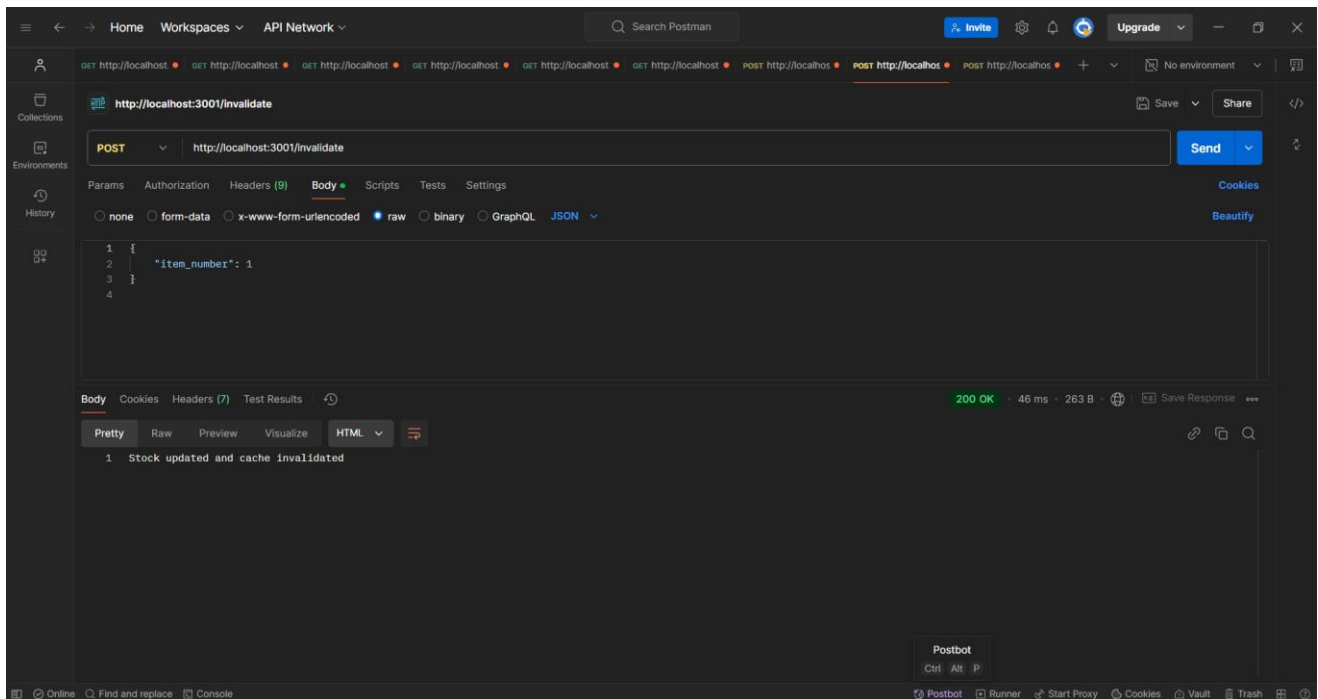
    if (row) {
      const updatedStock = row.stock - 1; // Decrease stock by 1

      // Update stock in the database
      db.run("UPDATE books SET stock = ? WHERE id = ?", [updatedStock, item_number], (updateErr) => {
        if (updateErr) {
          console.error('Error updating stock:', updateErr);
          return res.status(500).send('Error updating stock');
        }

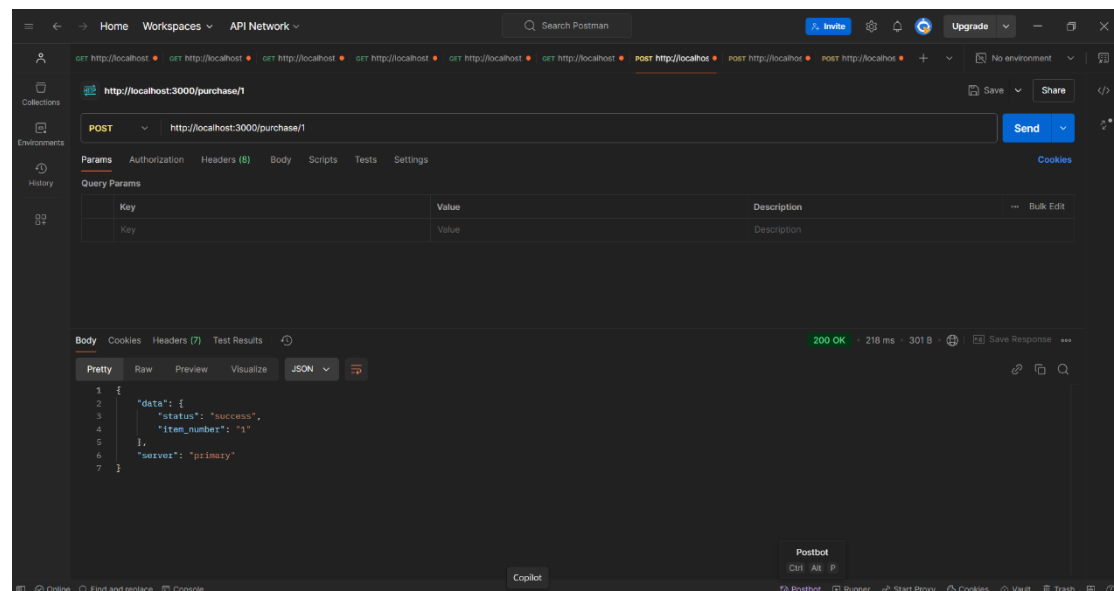
        // Cache invalidation
        if (cache[`info-${item_number}`]) {
          delete cache[`info-${item_number}`]; // Invalidate cache
          console.log(`Cache invalidated for item ${item_number}`);
        } else {
          console.log(`Cache item ${item_number} not found`);
        }

        res.send('Stock updated and cache invalidated');
      });
    } else {
      console.log(`Item ${item_number} not found in database`);
      res.status(404).send('Item not found');
    }
  })
});
```

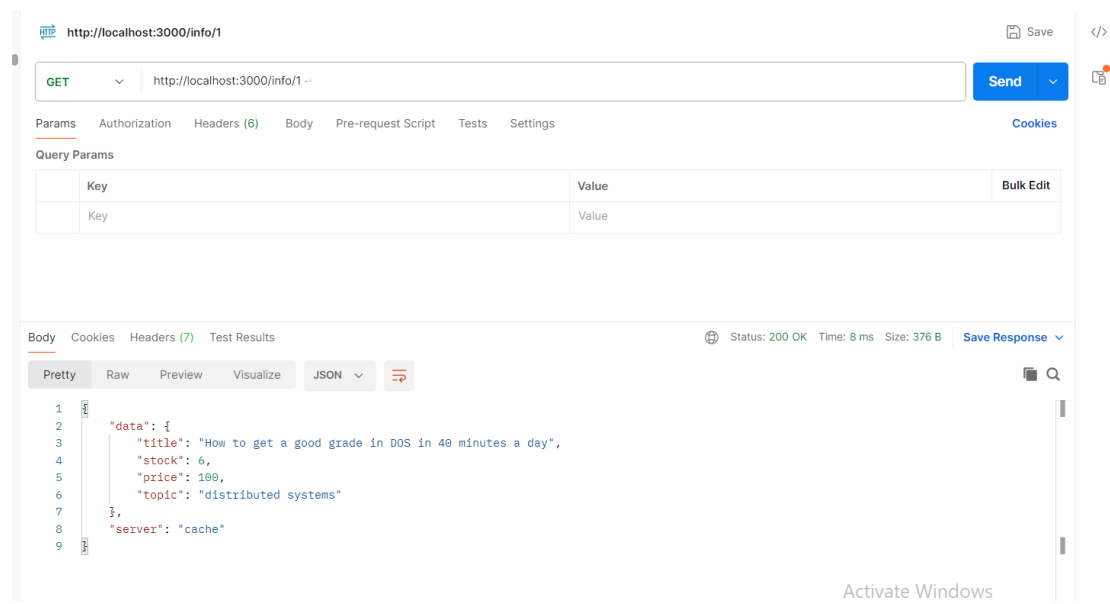
Test invalidate in postman:



We made the purchase:



The cashe inventory has been updated from 7 to 6 as shown in the following image:



Exoirement run	Without cashe	With cache	#of tims when using cache speed
<u>1</u>	17	<u>6</u>	<u>17/5=3.4 times</u>
<u>2</u>	<u>20</u>	<u>5</u>	<u>4</u>
<u>3</u>	<u>15</u>	<u>5</u>	<u>3</u>

Part2: Loadbalance with NGINX

I Used Nginx to achieve loadbalance, each service exist in it's seperate Docker Container and it has own Interface & Port to communicate with other services, Below my File Configuration for NGINX.

```
nginx > default.conf
1 upstream catalog-server {
2     server catalog-server:3001;
3     server catalog-replica:3003;
4 }
5
6 upstream order-server {
7     server order-server:3002;
8     server order-replica:3004;
9 }
10
11 upstream client {
12     server client:3000;
13 }
14
15 server {
16     listen 80;
17
18     # توجيه الطلبات إلى واجهة المستخدم
19     location / {
20         proxy_pass http://client; # توجيه جميع الطلبات إلى خدمة العميل
21     }
22
23     # توجيه الطلبات إلى خدمة الكتالوج
24     location /catalog {
25         proxy_pass http://catalog-server; # توجيه الطلبات إلى خدمة الكتالوج
26     }
27
28     # توجيه الطلبات إلى خدمة الطلبات
29     location /order {
30         proxy_pass http://order-server; # توجيه الطلبات إلى خدمة الطلبات
31     }
32 }
```




We did the following function to distribute the load between the original server and the replica:

```
// Server URLs
const catalogServers = [
  'http://catalog-server:3001',
  'http://catalog-replica:3003',
];

const orderServers = [
  'http://order-server:3002',
  'http://order-replica:3004',
];

let catalogIndex = 0;
let orderIndex = 0;

const chooseServer = (servers, isOrder = false) => {
  const index = isOrder ? orderIndex++ : catalogIndex++;
  if (isOrder) orderIndex = orderIndex % orderServers.length; // Update the pointer after reaching the end of the list
  else catalogIndex = catalogIndex % catalogServers.length;
  return servers[index % servers.length];
};

57 // Get book info
58 app.get('/info/:item_number', async (req, res) => {
59   const { item_number } = req.params;
60
61   // Check cache
62   if (cache['info-${item_number}']) {
63     console.log('Cache hit for book info "${item_number}"');
64     return res.json({ data: cache['info-${item_number}'], server: 'cache' });
65   }
66
67   const server = chooseServer(catalogServers);
68
69   try {
70     const response = await axios.get(`${server}/info/${item_number}`);
71     cache['info-${item_number}'] = response.data; // Cache the response
72
73     console.log('Cache updated for book info "${item_number}"');
74
75     res.json({ data: response.data, server: server.includes('replica') ? 'replica' : 'primary' });
76   } catch (error) {
77     console.error('Error fetching book info for item number "${item_number}":', error.message);
78     res.status(500).send('Error fetching book info');
79   }
80 });
```

here the function is called to distribute the load and based on the server that is chosen we request it through the API



The same thing was done for info according to the id and for search according to the topic.



The purchase, info or search process can be distributed to more than one server as shown in the following images:



From the original server "front end server" (Port 3000):

http://localhost:3000/info/1

GET http://localhost:3000/info/2

Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers (7) Test Results

Status: 200 OK Time: 9 ms Size: 341 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "data": {
3     "title": "RPCs for Noobs",
4     "stock": 8,
5     "price": 65,
6     "topic": "distributed systems"
7   },
8   "server": "primary"
9 }
```

Activate Windows
Go to Settings to activate Windows.

From the catalog server (Port 3001):

Postman interface showing a GET request to `http://localhost:3000/info/1`. The response is a JSON object with the following data:

```
{
  "title": "RPCs for Noobs",
  "stock": 8,
  "price": 65,
  "topic": "distributed systems"
}
```

Status: 200 OK, Time: 9 ms, Size: 312 B.

From the replica server (Port 3003):

Postman interface showing a GET request to `http://localhost:3000/info/2`. The response is a JSON object with the following data:

```
{
  "title": "RPCs for Noobs",
  "stock": 8,
  "price": 65,
  "topic": "distributed systems"
}
```

Status: 200 OK, Time: 11 ms, Size: 312 B.