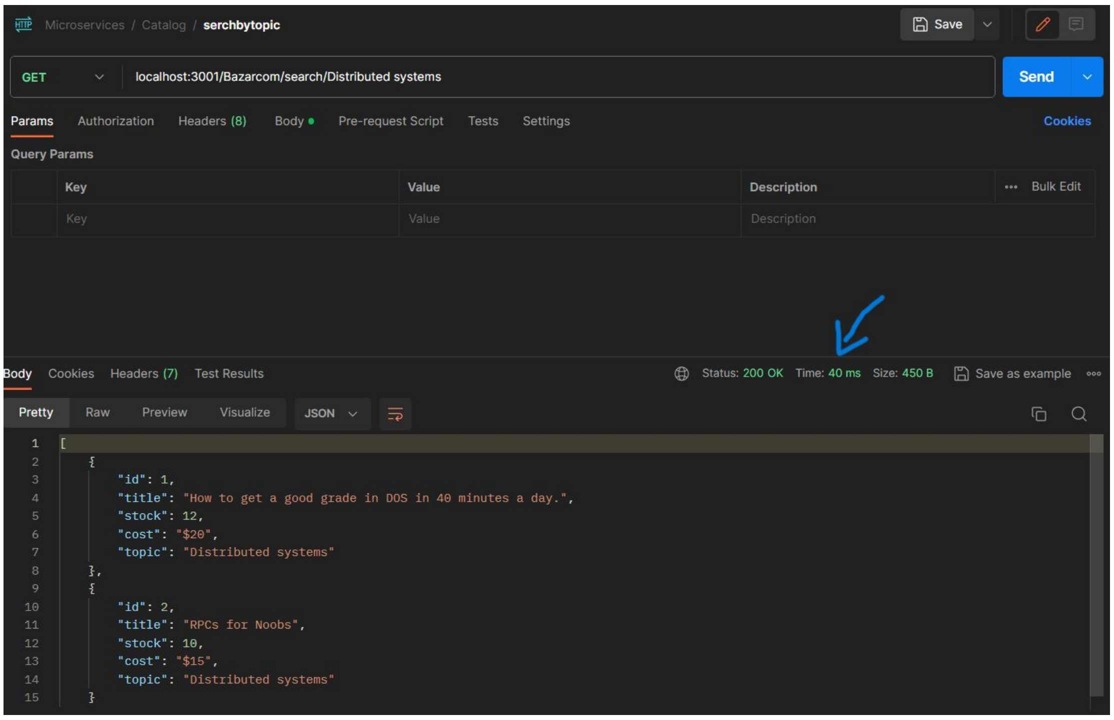
Bazarcom Web Store

By: Ahmad Darawsheh ,Mohammad Khaled
Date: 1/1/2024
Course: DOS

Experimental Evaluation and Measurements:

Cache:

We used Redis software which works like in-memory data structure store and it can be used as cache and this is what we used it for.

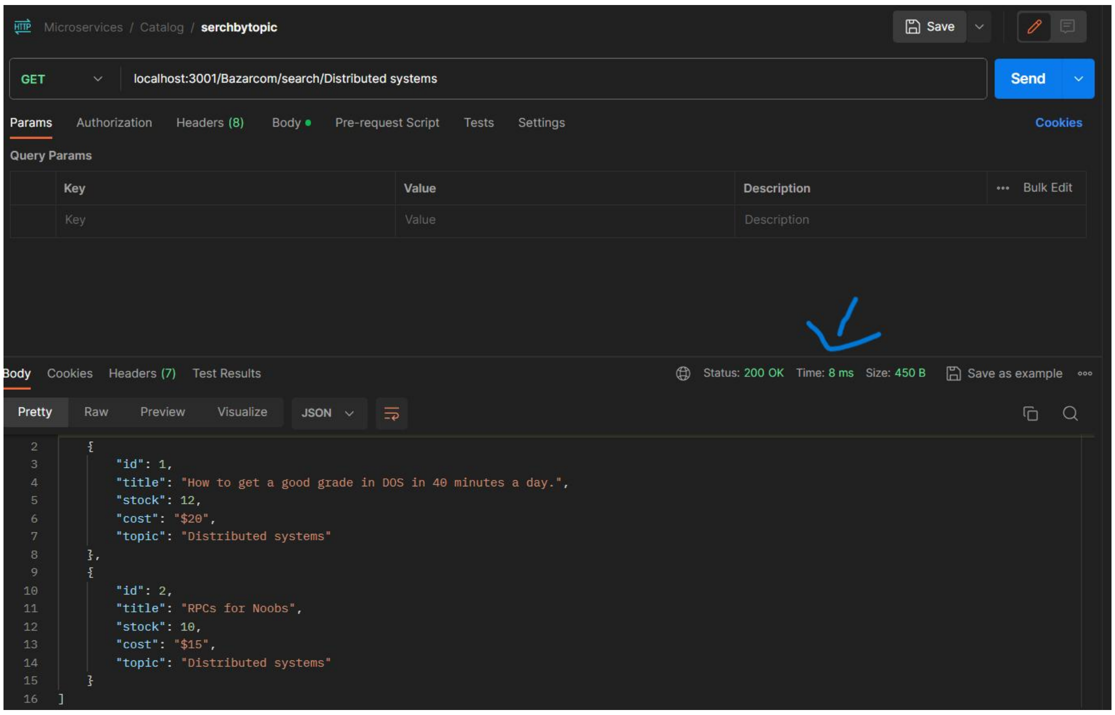


GET localhost:3001/Bazarcom/search/Distributed systems

Key	Value	Description
Key	Value	Description

Status: 200 OK Time: 40 ms Size: 450 B

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 12,
        "cost": "$20",
        "topic": "Distributed systems"
    },
    {
        "id": 2,
        "title": "RPCs for Noobs",
        "stock": 10,
        "cost": "$15",
        "topic": "Distributed systems"
    }
```



localhost:3001/Bazarcom/search/Distributed systems

Status: 200 OK Time: 8 ms Size: 450 B

```
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 12,
        "cost": "$20",
        "topic": "Distributed systems"
    },
    {
        "id": 2,
        "title": "RPCs for Noobs",
        "stock": 10,
        "cost": "$15",
        "topic": "Distributed systems"
    }
]
```

As shown above the response time went down from **40ms** to **8ms** , this shows how important the cache is.

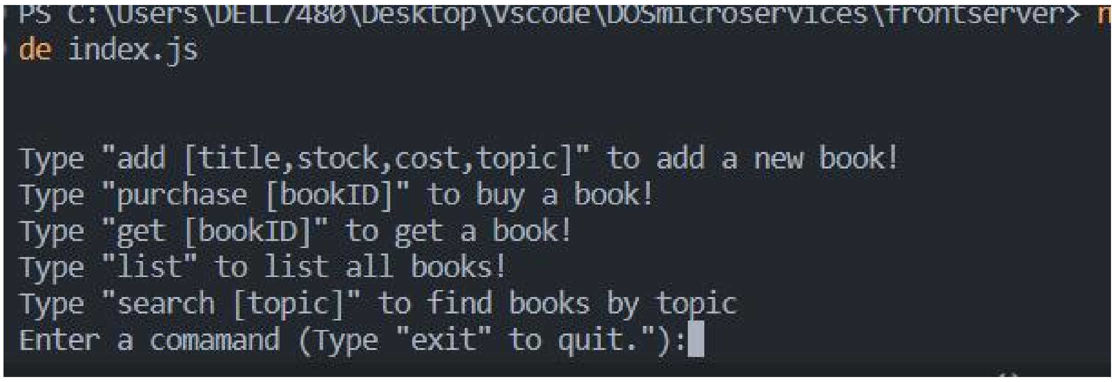
Using some samples from **search/id** request to compute average response time and showing the time without and with caching :

Samples	Cache miss	Cache hit	Difference
1	18ms	9ms	18ms / 9ms = 2
2	10ms	5ms	10ms / 5ms = 2
3	8ms	5ms	8ms / 5ms = 1.6

As shown above using cache is almost 2 times faster.

Bazarcom Store Implementation:

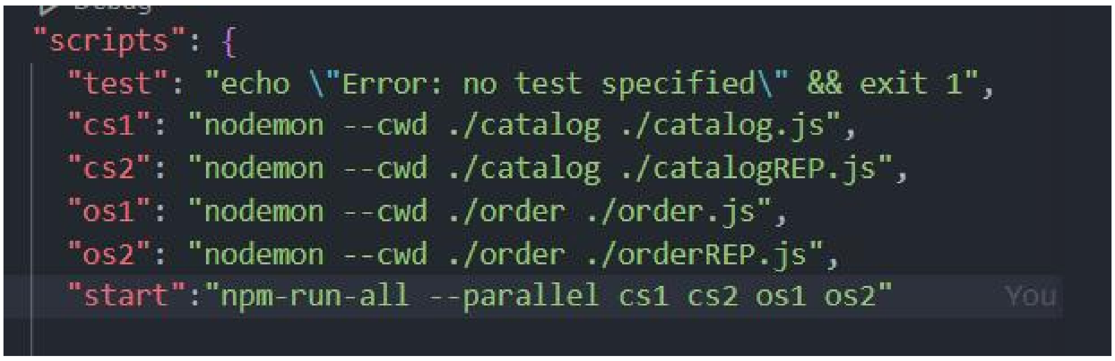
Frontend server CLI :



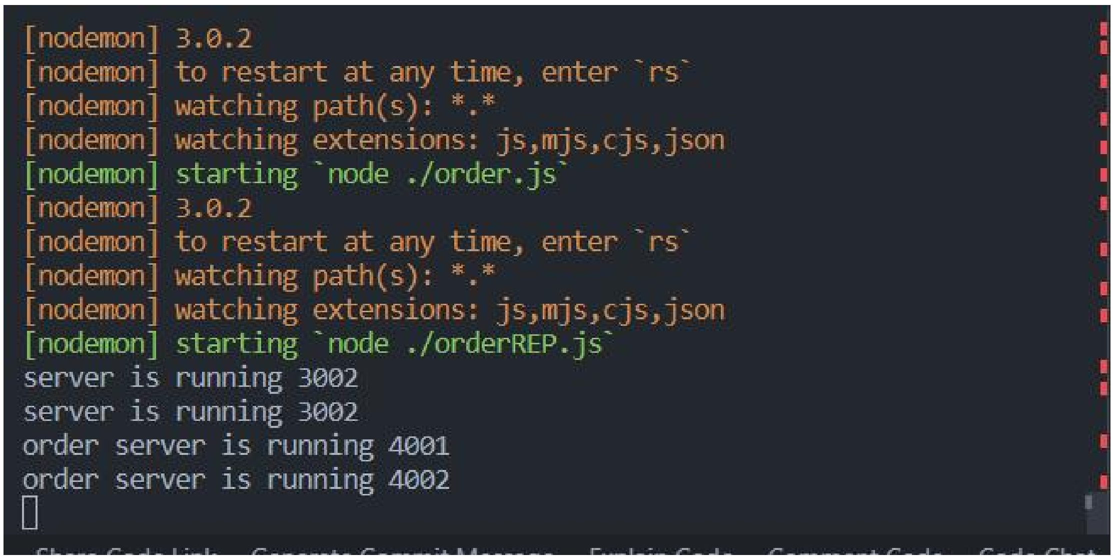
```
PS C:\Users\DELL7480\Desktop\Vscode\DOSmicroservices\frontserver> n
de index.js


Type "add [title,stock,cost,topic]" to add a new book!
Type "purchase [bookID]" to buy a book!
Type "get [bookID]" to get a book!
Type "list" to list all books!
Type "search [topic]" to find books by topic
Enter a comamand (Type "exit" to quit."):
```

We used readline package so we can make the CLI interactive and easy to deal with for e.g. Instead of writing **node index.js [command] [arguments]** we only run the server with **node index.js** and then interact with the **CLI** easly.

Also here we used **npm-run-all** package/library so we can run the real and replicated server in one command.
We just do this **npm start**

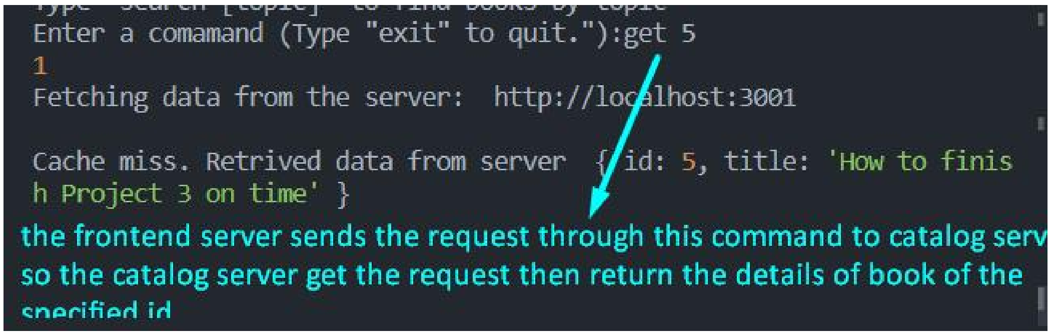


```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "cs1": "nodemon --cwd ./catalog ./catalog.js",
  "cs2": "nodemon --cwd ./catalog ./catalogREP.js",
  "os1": "nodemon --cwd ./order ./order.js",
  "os2": "nodemon --cwd ./order ./orderREP.js",
  "start":"npm-run-all --parallel cs1 cs2 os1 os2"
```



```
[nodemon] 3.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node ./order.js`
[nodemon] 3.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node ./orderREP.js`
server is running 3002
server is running 3002
order server is running 4001
order server is running 4002
```

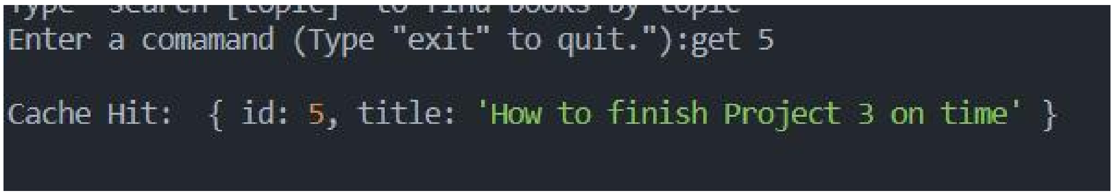
Caching:



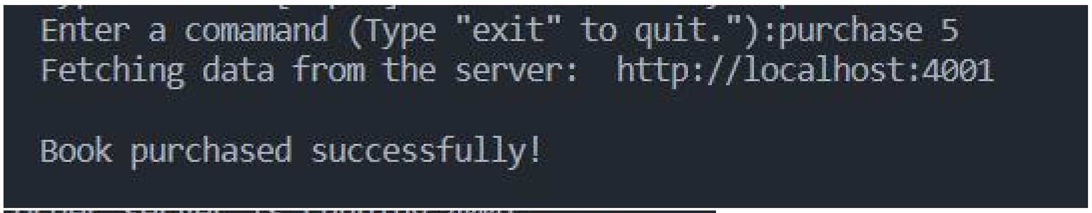
```
Enter a comamand (Type "exit" to quit."):get 5
1
Fetching data from the server:  http://localhost:3001

Cache miss. Retrived data from server  { id: 5, title: 'How to finis
h Project 3 on time' }
```

the frontend server sends the request through this command to catalog serv
so the catalog server get the request then return the details of book of the
specified id

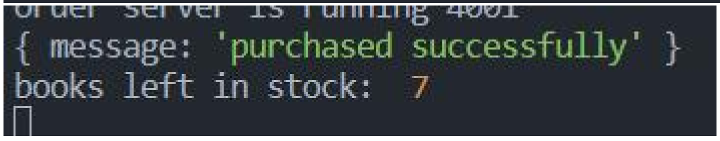
It first checks the cache but it didn't find anything so the request goes to one of catalog servers, so it says **"Cache miss"** and then info comes from the server as new data as shown, after the the request/response is done the data will be stored in cache, so if you enter the same command it will get faster and from cache as you in this picture:



```
Enter a comamand (Type "exit" to quit."):get 5

Cache Hit:  { id: 5, title: 'How to finish Project 3 on time' }
```

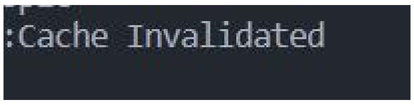


```
Enter a comamand (Type "exit" to quit."):purchase 5
Fetching data from the server:  http://localhost:4001

Book purchased successfully!
```

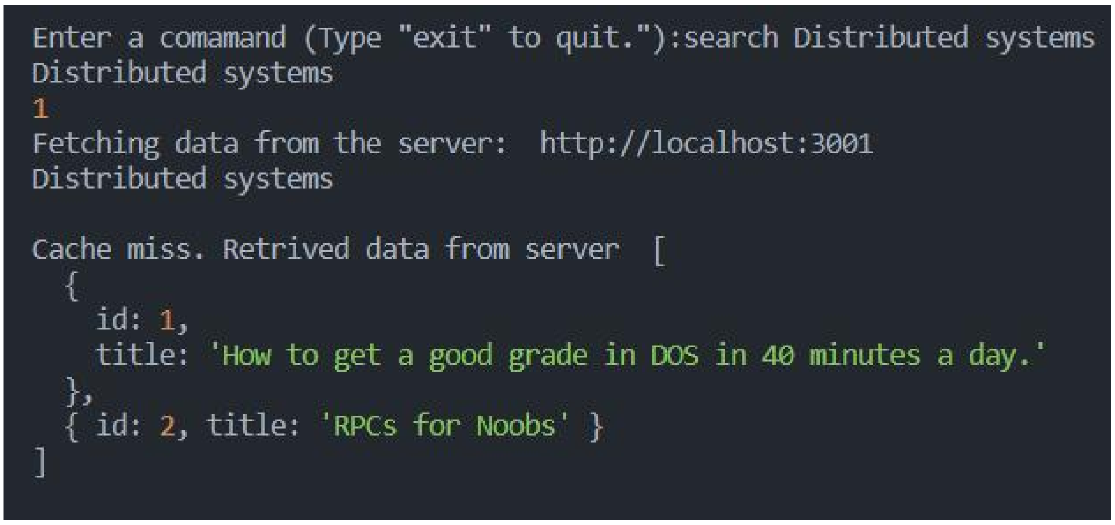


```
{ message: 'purchased successfully' }
books left in stock:  7
```

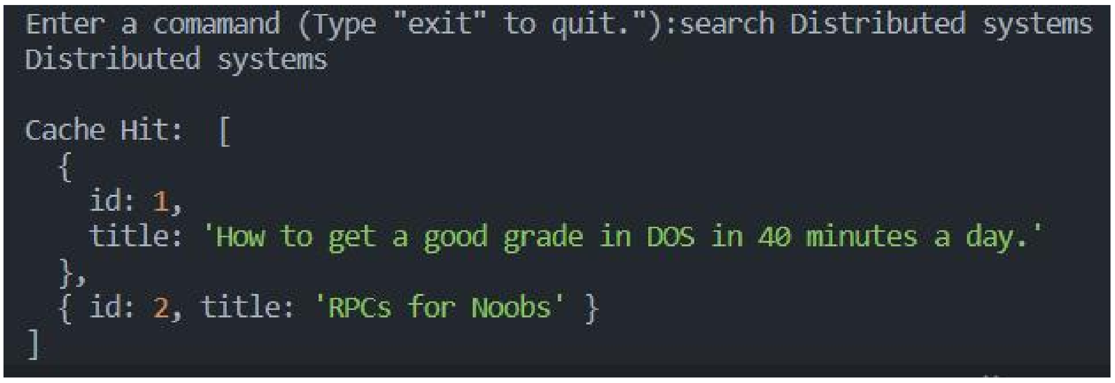
Here we bought a book buy entering **purchase [bookId]** command, notice that the request goes to one of the order servers and then they deal with one of the calatog servers to retrieve the wanted data.
After the purchase is completed the catalog servers notice and there's a change happened and database and the data in cache is invalid so it sends a request to frontend server to invalidate the cache so when the **get [bookId]** command is used once again it will bring the right data of the specified book. This what it called **Cache Consistency!**



```
:Cache Invalidated
```

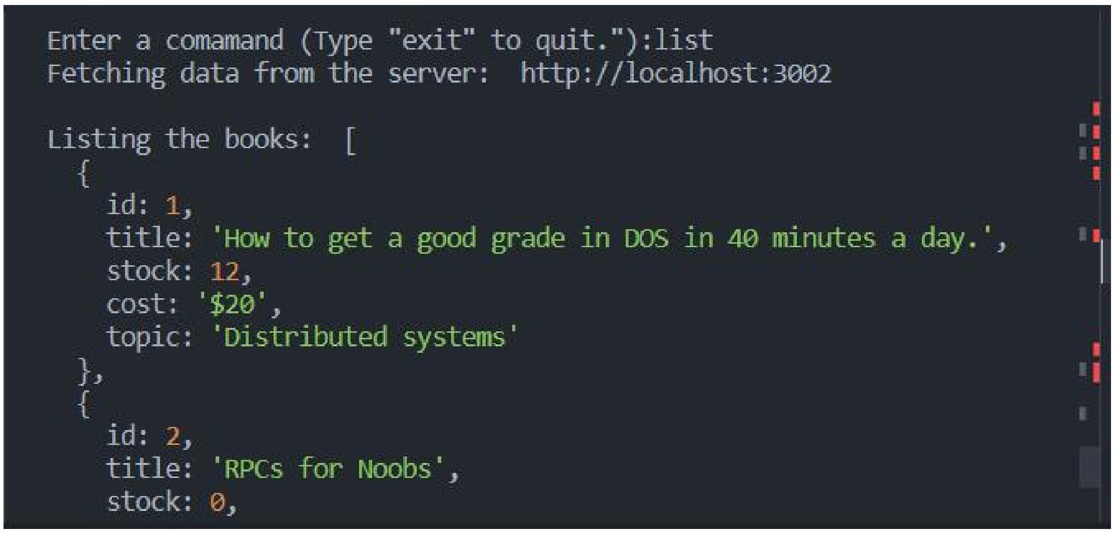


```
Enter a comamand (Type "exit" to quit."):search Distributed systems
Distributed systems
1
Fetching data from the server:  http://localhost:3001
Distributed systems

Cache miss. Retrived data from server  [
  {
    id: 1,
    title: 'How to get a good grade in DOS in 40 minutes a day.'
  },
  { id: 2, title: 'RPCs for Noobs' }
]
```



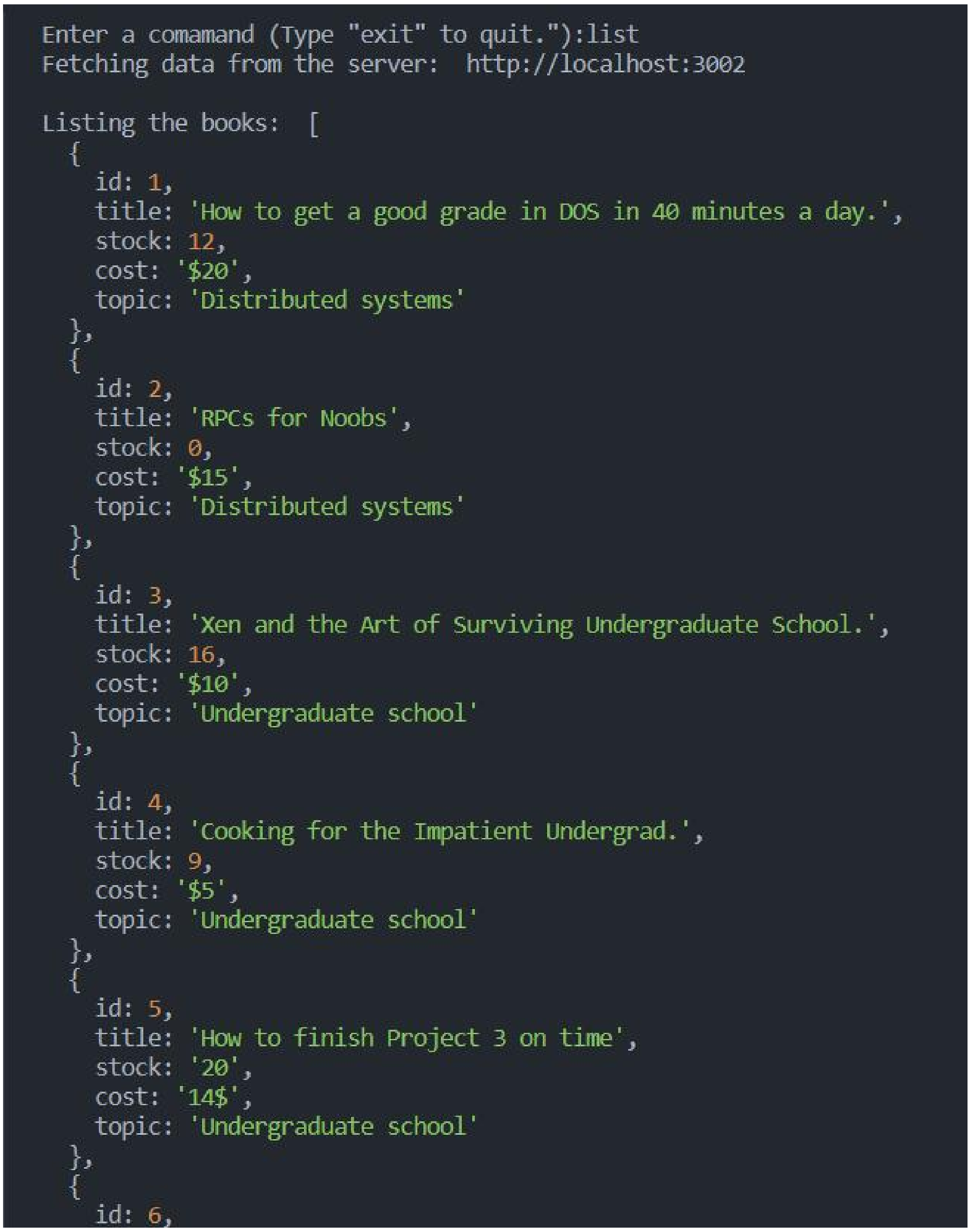
```
Enter a comamand (Type "exit" to quit."):search Distributed systems
Distributed systems

Cache Hit:  [
  {
    id: 1,
    title: 'How to get a good grade in DOS in 40 minutes a day.'
  },
  { id: 2, title: 'RPCs for Noobs' }
]
```

Caching is also implemented here using the command **search [topic]**. If something in database related to the specific topic is updated the cache will invalidated(**Cache Consistency**) as mentioned above.



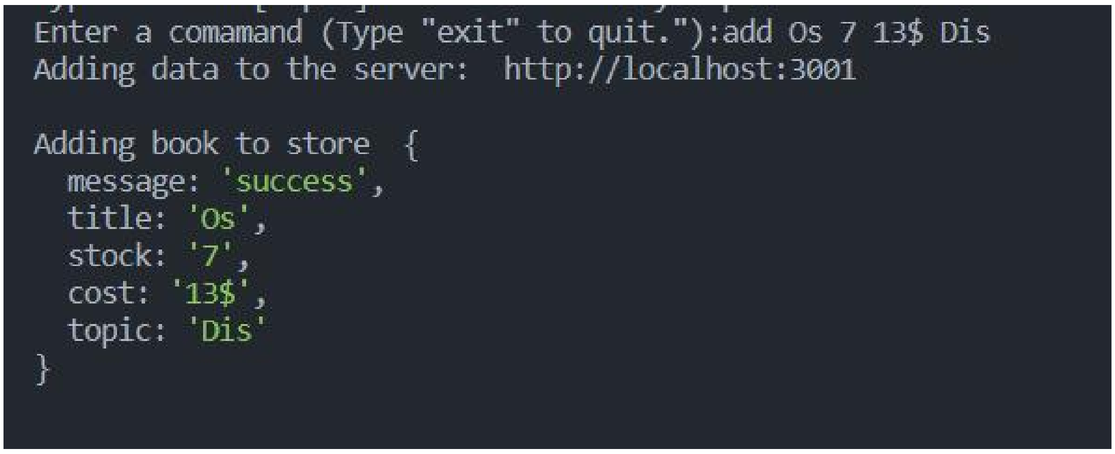
```
Enter a comamand (Type "exit" to quit."):list
Fetching data from the server:  http://localhost:3002

Listing the books:  [
  {
    id: 1,
    title: 'How to get a good grade in DOS in 40 minutes a day.',
    stock: 12,
    cost: '$20',
    topic: 'Distributed systems'
  },
  {
    id: 2,
    title: 'RPCs for Noobs',
    stock: 0,
```

List command is used to show all the available books in the store.



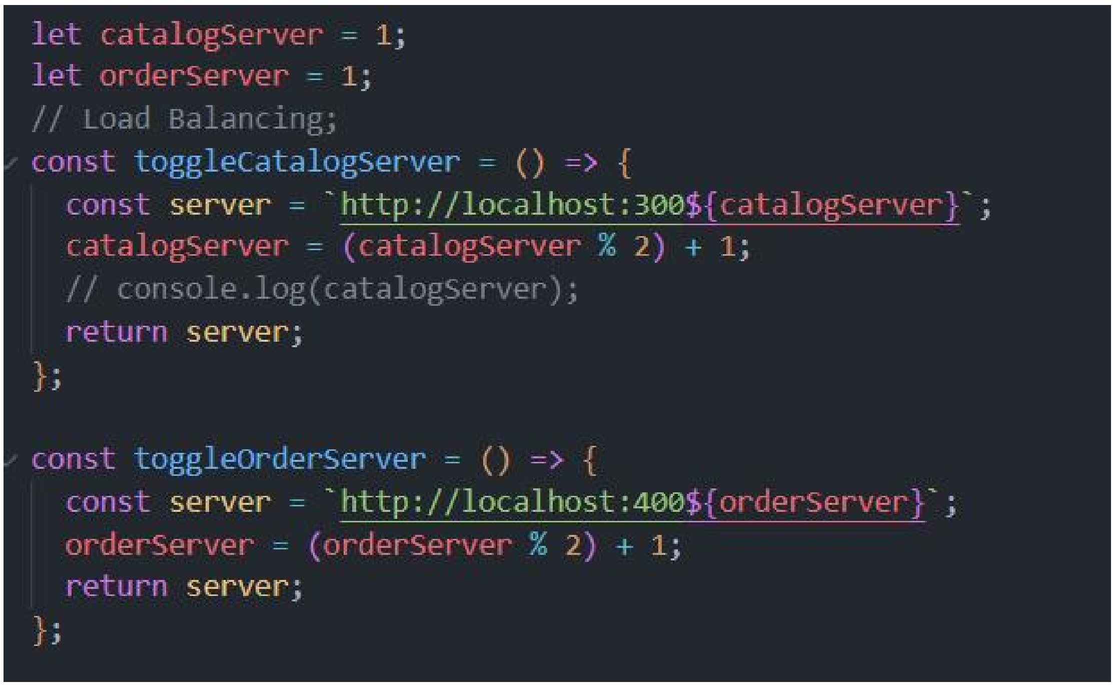
```
Enter a comamand (Type "exit" to quit."):list
Fetching data from the server:  http://localhost:3002

Listing the books:  [
  {
    id: 1,
    title: 'How to get a good grade in DOS in 40 minutes a day.',
    stock: 12,
    cost: '$20',
    topic: 'Distributed systems'
  },
  {
    id: 2,
    title: 'RPCs for Noobs',
    stock: 0,
    cost: '$15',
    topic: 'Distributed systems'
  },
  {
    id: 3,
    title: 'Xen and the Art of Surviving Undergraduate School.',
    stock: 16,
    cost: '$10',
    topic: 'Undergraduate school'
  },
  {
    id: 4,
    title: 'Cooking for the Impatient Undergrad.',
    stock: 9,
    cost: '$5',
    topic: 'Undergraduate school'
  },
  {
    id: 5,
    title: 'How to finish Project 3 on time',
    stock: '20',
    cost: '14$',
    topic: 'Undergraduate school'
  },
  {
    id: 6,
```

add [title,stock,cost,topic] command to add new book to store.

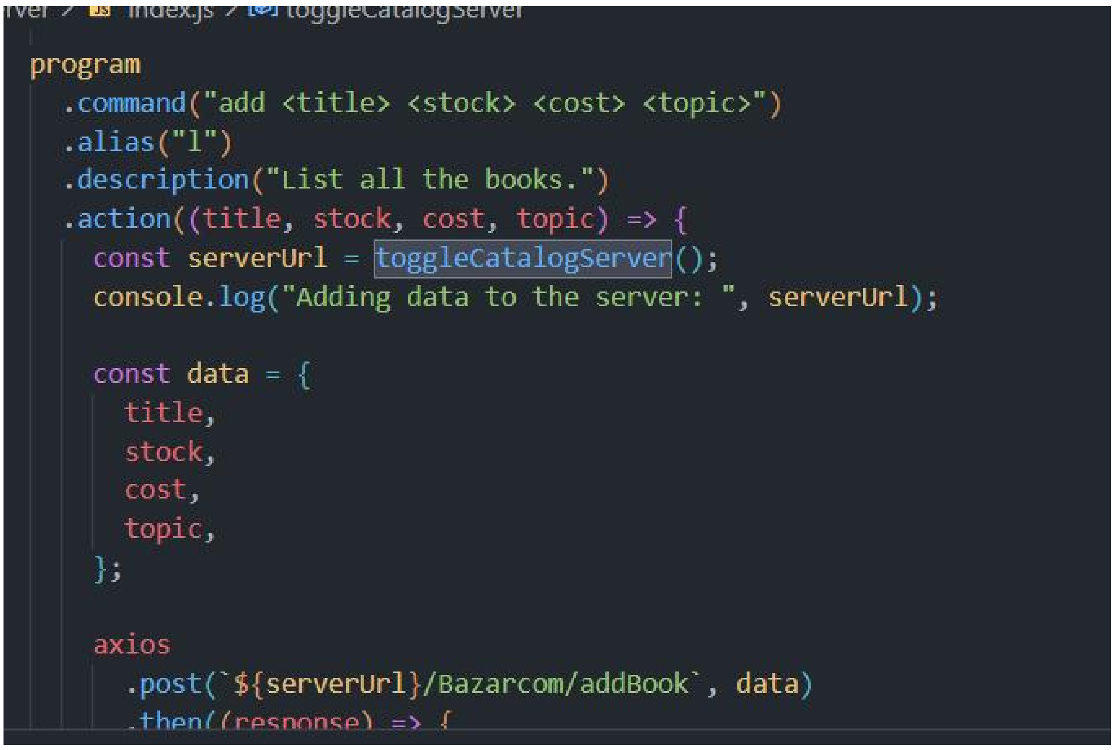


```
Enter a comamand (Type "exit" to quit."):add Os 7 13$ Dis
Adding data to the server:  http://localhost:3001

Adding book to store  {
  message: 'success',
  title: 'Os',
  stock: '7',
  cost: '13$',
  topic: 'Dis'
}
```

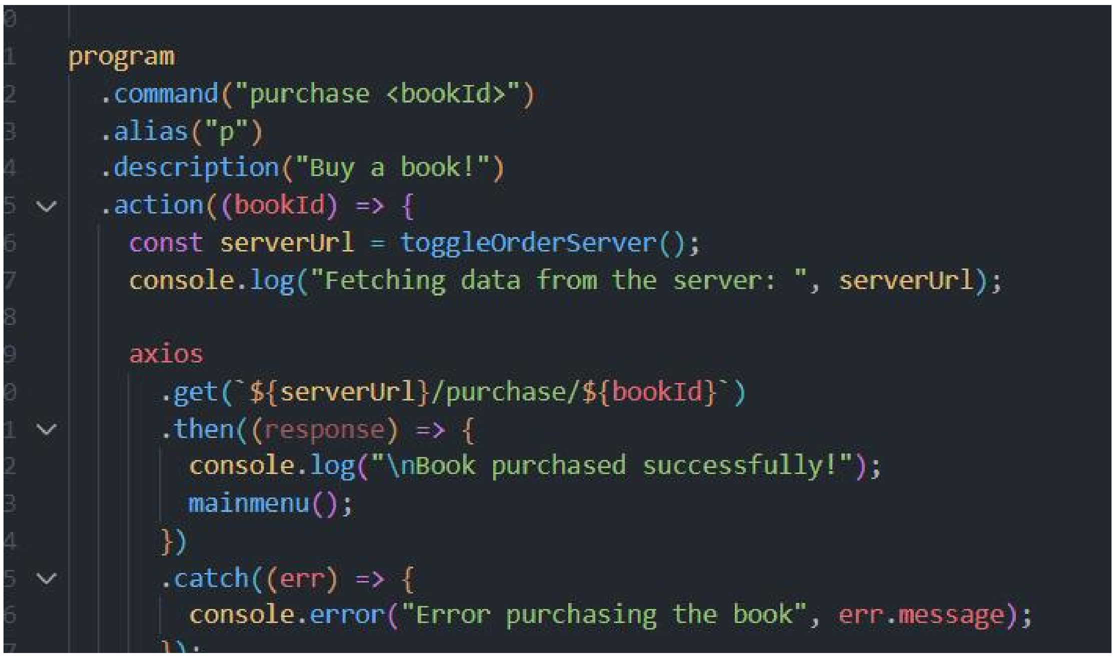
Load Balancing:



```
let catalogServer = 1;
let orderServer = 1;
// Load Balancing;
const toggleCatalogServer = () => {
  const server = `http://localhost:300${catalogServer}`;
  catalogServer = (catalogServer % 2) + 1;
  // console.log(catalogServer);
  return server;
};

const toggleOrderServer = () => {
  const server = `http://localhost:400${orderServer}`;
  orderServer = (orderServer % 2) + 1;
  return server;
};
```



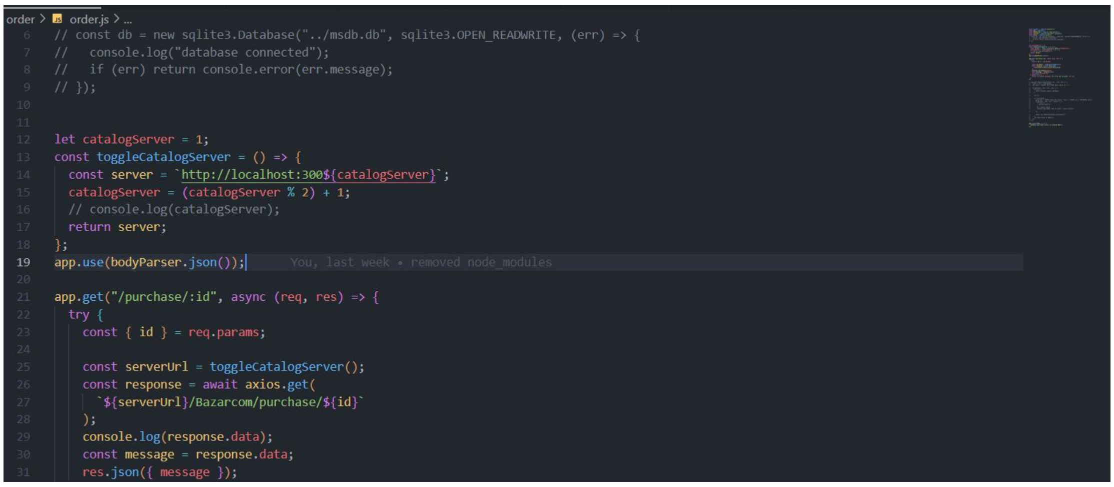
```
program
  .command("add <title> <stock> <cost> <topic>")
  .alias("l")
  .description("List all the books.")
  .action((title, stock, cost, topic) => {
    const serverUrl = toggleCatalogServer();
    console.log("Adding data to the server: ", serverUrl);

    const data = {
      title,
      stock,
      cost,
      topic,
    };

    axios
      .post(`${serverUrl}/Bazarcom/addBook`, data)
```



```
program
  .command("purchase <bookId>")
  .alias("p")
  .description("Buy a book!")
  .action((bookId) => {
    const serverUrl = toggleOrderServer();
    console.log("Fetching data from the server: ", serverUrl);

    axios
      .get(`${serverUrl}/purchase/${bookId}`)
      .then((response) => {
        console.log("\nBook purchased successfully!");
        mainmenu();
      })
      .catch((err) => {
        console.error("Error purchasing the book", err.message);
```

We used Round-Robin algorithm for load balancing as you see above, When a request comes it first goes to first catalog/order server. The next request ,after that, will be sent to the other server according to Round-Robin algorithm. The toggleServer function is used inside every request in frontend server. The method we've used is changing the last number of the port e.g original Catalog sever is running on the port 3001

And the replicated one is running on port 3002 so what we do is this eq. To change the last number of the port : catalogServer = (catalogServer % 2) + 1;



order > order.js > ...

```javascript
// const db = new sqlite3.Database("../msdb.db", sqlite3.OPEN_READWRITE, (err) => {
//   console.log("database connected");
//   if (err) return console.error(err.message);
// });


let catalogServer = 1;
const toggleCatalogServer = () => {
  const server = `http://localhost:300${catalogServer}`;
  catalogServer = (catalogServer % 2) + 1;
  // console.log(catalogServer);
  return server;
};
app.use(bodyParser.json());        You, last week • removed node_modules

app.get("/purchase/:id", async (req, res) => {
  try {
    const { id } = req.params;

    const serverUrl = toggleCatalogServer();
    const response = await axios.get(
      `${serverUrl}/Bazarcom/purchase/${id}`
    );
    console.log(response.data);
    const message = response.data;
    res.json({ message });
```

Load balancing is also used inside Order servers so the purchase request once goes to first catalog server and once to the 2nd server.