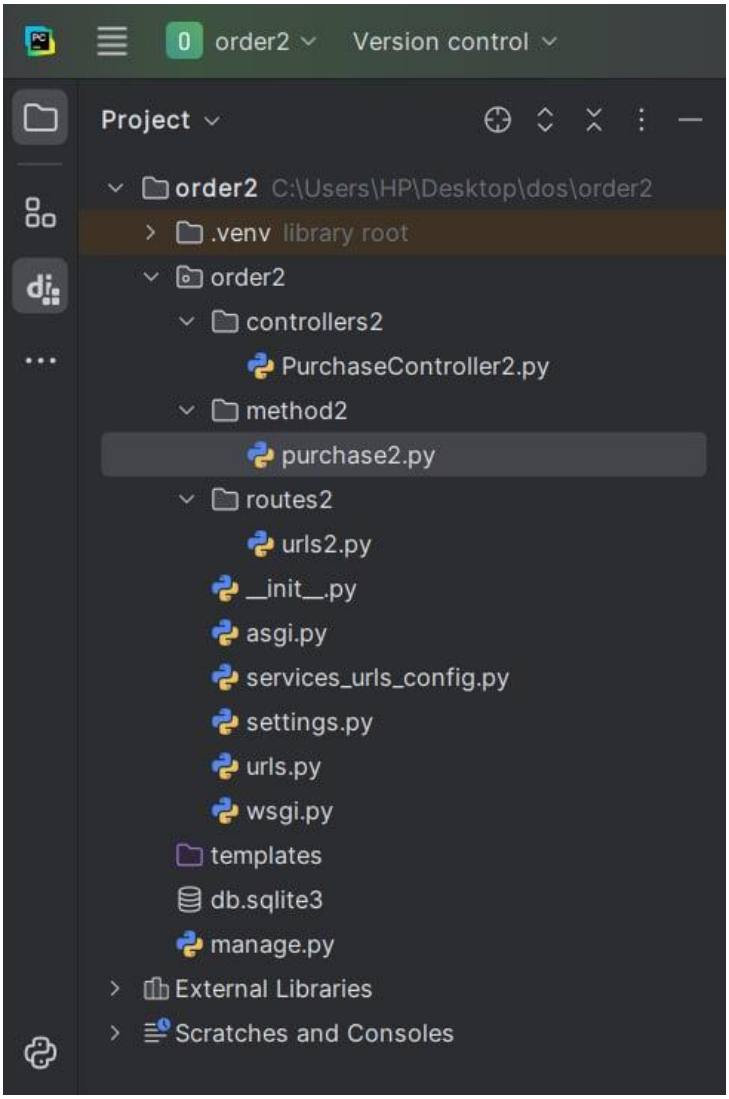
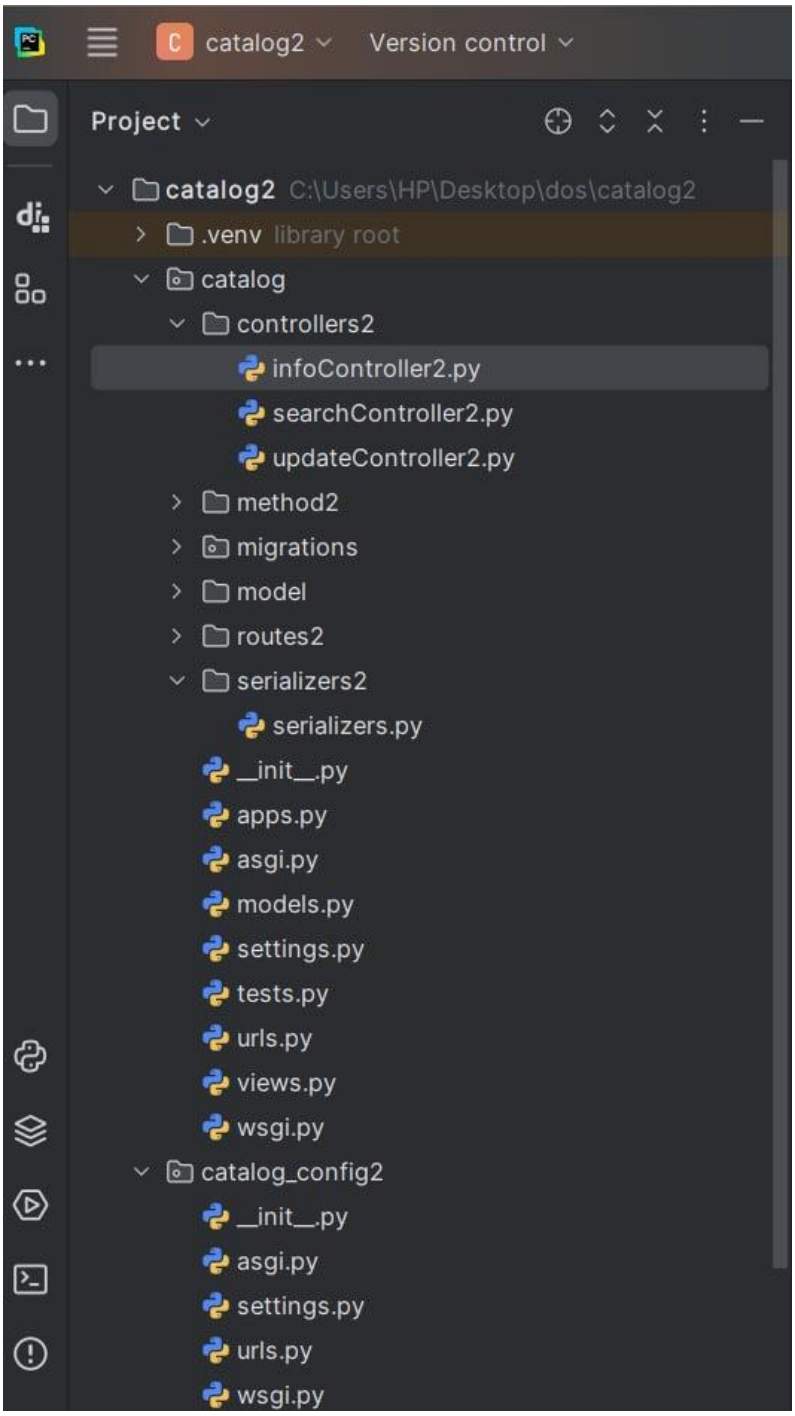
Turning the Bazar into an Amazon: Replication, Caching and Consistency

In this project, we transformed Bazar.com to handle more traffic and respond faster by adding caching and replication. With the store growing in popularity, response time was becoming an issue, so we introduced caching to speed up repeated requests and set up replicas for the order and catalog services. This way, we could balance the workload across multiple servers and keep data consistent for users. This report covers how we structured the project, set up the caching system, and used round-robin load balancing to improve performance.

The project structure of the order2 service, showing folders such as controllers2, method2, and routes2.



The project structure of the catalog2 service, showing folders like controllers2, method2, and serializers2



Timeout:

We set different expiration times for our caches. The local cache keeps items for 300 seconds, allowing frequently accessed data to stay available for longer. The default cache expires items faster at 20 seconds, which keeps it more up-to-date but doesn't hold onto items for as long.

Size of Elements: We also set limits on the number of items each cache can store. The local cache holds up to 10 items, focusing on a smaller, high-priority set of data. The default cache, however, can store up to 100 items, giving it a broader reach for general data but ensuring it doesn’t overflow.

```
CACHES = {
    'local': {
        'BACKEND': 'lrucache_backend.LRUObjectCache',
        'TIMEOUT': 300,
        'OPTIONS': {
            'MAX_ENTRIES': 10,
            'TIMEOUT': 300,
            'CULL_FREQUENCY': 1,
        },
    },
    'default': {
        'BACKEND': 'lrucache_backend.LRUObjectCache',
        'TIMEOUT': 20,
        'OPTIONS': {
            'MAX_ENTRIES': 100,
            'TIMEOUT': 20,
        }
    }
}
```

Retrieve from Cache:

The code first checks if the required data is already in the local cache using a unique cache_key. If it's there, it retrieves the with a 200 OK status.

Add to Cache on Miss: If the data isn't found in the cache (a cache miss), it forwards the request to one of the backend replicas in a round-robin, retrieves the data, and then stores it in the cache.

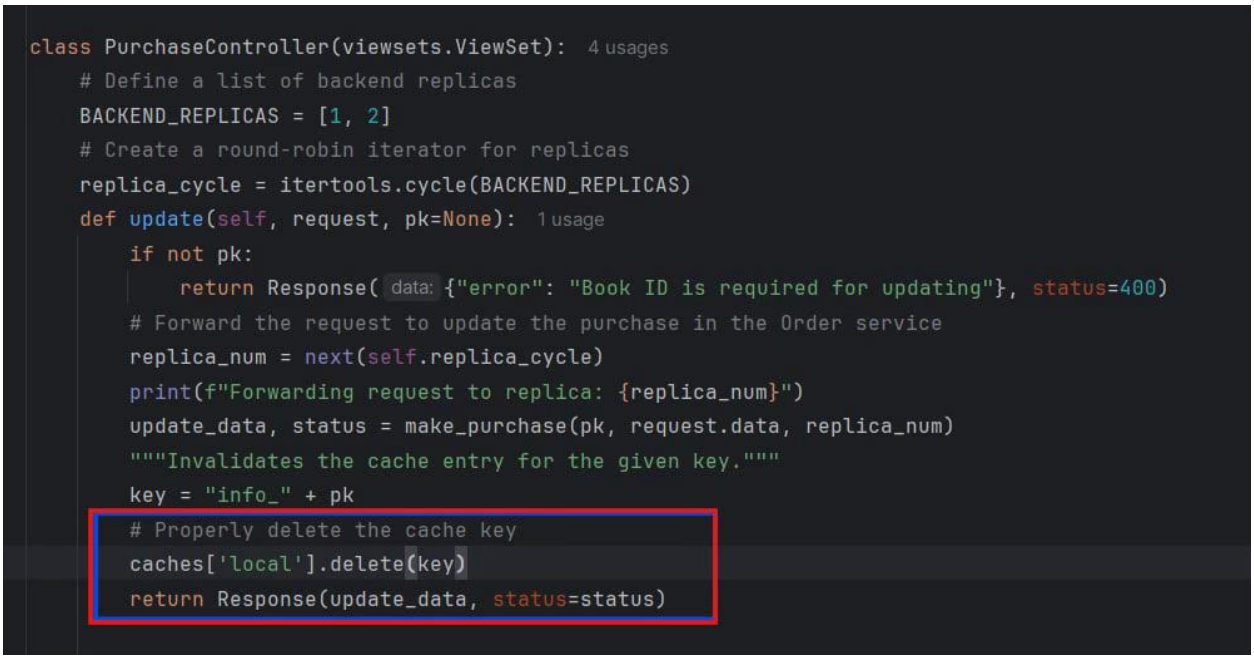
```python
from rest_framework import viewsets
from rest_framework.response import Response
from fe.method.info import get_info
from django.core.cache import caches
import itertools

class InfoController(viewsets.ViewSet):  2 usages
    # Define a list of backend replicas
    BACKEND_REPLICAS = [1, 2]

    # Create a round-robin iterator for replicas
    replica_cycle = itertools.cycle(BACKEND_REPLICAS)
    def retrieve(self, request, pk=None):
        # Try to get data from the cache
        cache_key = "info_" + pk
        info_data = caches['local'].get(cache_key)
        if not info_data:
            # If cache miss, fetch data and store it in the cache
            replica_num = next(self.replica_cycle)
            print(f"Forwarding request to replica: {replica_num}")
            info_data, status = get_info(pk, replica_num)
            print("data is not cached")
            caches['local'].set(cache_key, info_data)  # Cache data for 5 minutes
        else:
            # If data is cached, set the status to 200 (OK)
            print("data is cached")
            status = 200
        return Response(info_data, status=status)
```

same logic in search controller

```python
from rest_framework import viewsets
from rest_framework.response import Response
from fe.method.info import get_info
from django.core.cache import caches
import itertools

class InfoController(viewsets.ViewSet):  2 usages
    # Define a list of backend replicas
    BACKEND_REPLICAS = [1, 2]

    # Create a round-robin iterator for replicas
    replica_cycle = itertools.cycle(BACKEND_REPLICAS)
    def retrieve(self, request, pk=None):
        # Try to get data from the cache
        cache_key = "info_" + pk
        info_data = caches['local'].get(cache_key)
        if not info_data:
            # If cache miss, fetch data and store it in the cache
            replica_num = next(self.replica_cycle)
            print(f"Forwarding request to replica: {replica_num}")
            info_data, status = get_info(pk, replica_num)
            print("data is not cached")
            caches['local'].set(cache_key, info_data)  # Cache data for 5 minutes
        else:
            # If data is cached, set the status to 200 (OK)
            print("data is cached")
            status = 200
        return Response(info_data, status=status)
```

Purchase:

When a purchase is made, the specific item is removed from the cache to keep data up-to-date. To insure future requests won't get outdated info.



```
class PurchaseController(viewsets.ViewSet):  4 usages
    # Define a list of backend replicas
    BACKEND_REPLICAS = [1, 2]
    # Create a round-robin iterator for replicas
    replica_cycle = itertools.cycle(BACKEND_REPLICAS)
    def update(self, request, pk=None):  1 usage
        if not pk:
            return Response( data: {"error": "Book ID is required for updating"}, status=400)
        # Forward the request to update the purchase in the Order service
        replica_num = next(self.replica_cycle)
        print(f"Forwarding request to replica: {replica_num}")
        update_data, status = make_purchase(pk, request.data, replica_num)
        """Invalidates the cache entry for the given key."""
        key = "info_" + pk
        # Properly delete the cache key
        caches['local'].delete(key)
        return Response(update_data, status=status)
```

Update:

Similarly, when an item is updated, the invalidate method is called to delete that item from the cache. This keeps the cache consistent with the latest data.

```python
from rest_framework import viewsets
from rest_framework.response import Response
from fe.method.invalidate import invalidate

class InvalidateCacheController(viewsets.ViewSet):  2 usages
    def retrieve(self, request, pk=None):
        if not pk:
            return Response( data: {"error": "PK is required for updating"}, status=400)
        # Forward the request to update the purchase in the Order service
        response = invalidate(pk)
        return Response(response, status=200)
```

This method retrieves book information from one of the catalog replicas based on the replica_num. It checks the status code to ensure the data was successfully fetched, if not it returns an error message.

```python
def get_info(pk, replica_num):  2 usages
    if replica_num == 1 :
        response = requests.get(URL_CATALOG_INFO + str(pk))
    else :
        response = requests.get(URL_CATALOG_INFO2 + str(pk))
    if response.status_code == 200:
        return response.json(), response.status_code
    return {"error": "Failed to fetch book info from Catalog service"}, response.status_code
```

This method invalidates the cache entry for a given key. It checks if the key exists in the cache, deletes it if found, and returns a confirmation. If the key is not found, it returns a message saying no cache entry was found.

```python
def invalidate(pk):  2 usages
    """Invalidates the cache entry for the given key if it exists."""
    print("invalidate")
    print(pk)
    key = "info_" + pk
    # Check if the cache key exists
    if caches['local'].get(key) is not None:
        caches['local'].delete(key)
        return f"Cache entry for {key} invalidated."
    else:
        return f"No cache entry found for {key}."
```

This method sends a purchase request to one of the order replicas based on replica_num. If the update is successful (status code 200), it returns the updated data; otherwise, it provides an error message indicating the failure.

```python
def make_purchase(pk, data, replica_num):  2 usages
    if replica_num == 1 :
        response = requests.put( url: f"{URL_ORDER_PURCHASE}{pk}", json=data)
    else :
        response = requests.put( url: f"{URL_ORDER_PURCHASE2}{pk}", json=data)
    if response.status_code == 200:
        return response.json(), response.status_code
    return {"error": "Failed to update purchase in Order service"}, response.status_code
```

This controller handles update requests for book information. After updating, it constructs a URL to call the invalidateCache API to remove the outdated data from the

cache. If this invalidation request fails, it logs the error but still returns the update result.

```python
class updateController2(viewsets.ViewSet):  2 usages
    def update(self, request, pk=None):
        result = update_book_info2(pk, request.data)
        if isinstance(result, dict) and 'errors' in result:
            return Response(result, status=400)
        frontend_url = f"http://127.0.0.1:8001/FrontEnd_service/invalidateCache/{pk}"
        try:
            requests.get(frontend_url)
        except requests.RequestException as e:
            print(f"Failed to invalidate cache: {e}")
        return Response(result)
```

```python
def update_book_info(book_id, data):  2 usages
    # Retrieve the book object from the 'default' database
    book = get_object_or_404(Book.objects.using('default'), pk=book_id)

    # Serialize the data
    serializer = BookSerializer(book, data=data, partial=True)

    # If valid, update both databases
    if serializer.is_valid():
        with transaction.atomic(using='default'):
            serializer.save(using='default')  # Save to 'default' database

        # Update the replica
        # Fetch a fresh instance from the replica database, to update it independently
        book_replica = Book.objects.using('replica').get(pk=book_id)
        serializer_replica = BookSerializer(book_replica, data=data, partial=True)

        if serializer_replica.is_valid():
            with transaction.atomic(using='replica'):
                serializer_replica.save(using='replica')  # Save to 'replica' database
            return serializer.data

    return serializer.errors
```

This method updates book details in both the primary and replica databases. It retrieves the book, serializes the data, and saves it in each database separately to maintain consistency. If validation fails, it returns errors, ensuring only valid data is saved.

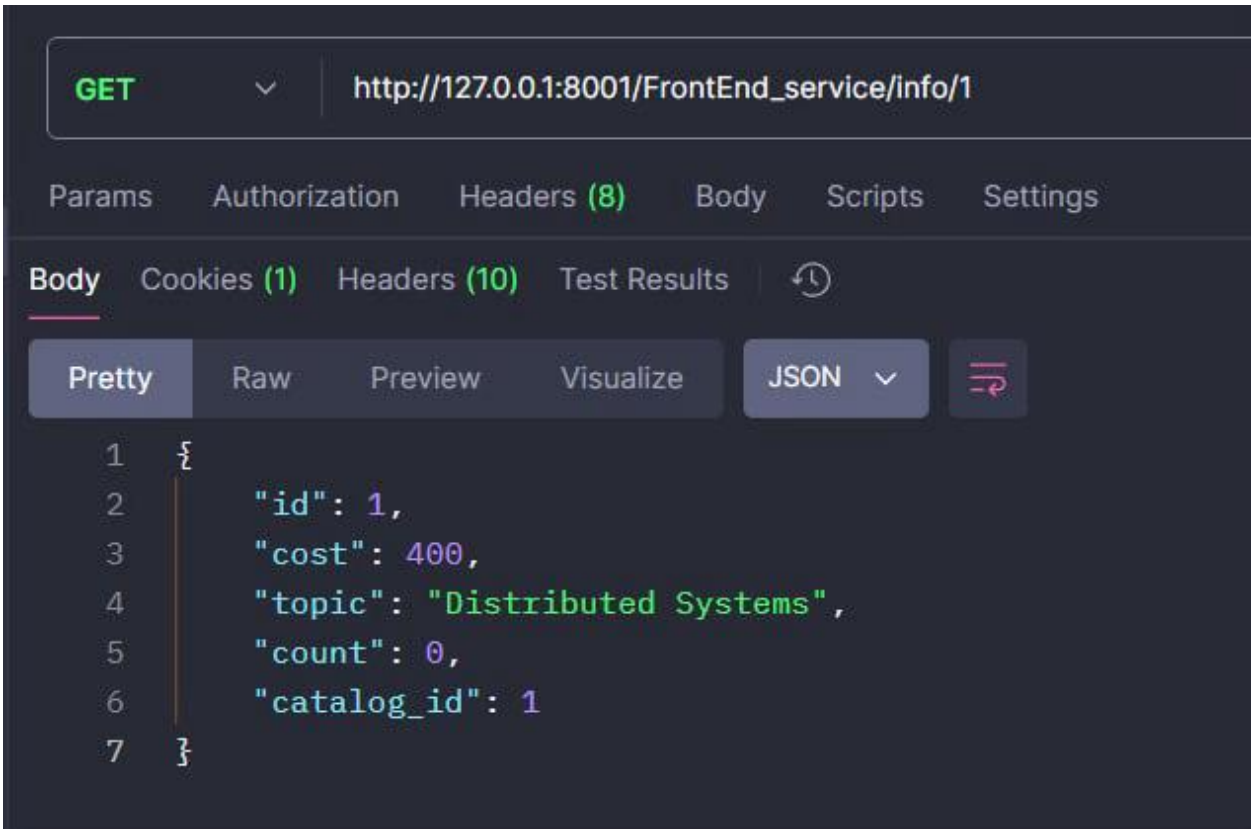
```python
from rest_framework import viewsets
from rest_framework.decorators import action
from rest_framework.response import Response
from catalog.method2.update2 import update_book_info2
import requests

class updateController2(viewsets.ViewSet):  2 usages
    def update(self, request, pk=None):
        result = update_book_info2(pk, request.data)
        if isinstance(result, dict) and 'errors' in result:
            return Response(result, status=400)
        frontend_url = f"http://127.0.0.1:8001/FrontEnd_service/invalidateCache/{pk}"
        try:
            requests.get(frontend_url)
        except requests.RequestException as e:
            print(f"Failed to invalidate cache: {e}")
        return Response(result)
```

```python
def update_book_info2(book_id, data):  2 usages
    # Retrieve the book object from the 'default' database
    book = get_object_or_404(Book.objects.using('default'), pk=book_id)

    # Serialize the data
    serializer = BookSerializer(book, data=data, partial=True)

    # If valid, update both databases
    if serializer.is_valid():
        with transaction.atomic(using='default'):
            serializer.save(using='default')  # Save to 'default' database

        # Update the replica
        # Fetch a fresh instance from the replica database, to update it independently
        book_replica = Book.objects.using('replica').get(pk=book_id)
        serializer_replica = BookSerializer(book_replica, data=data, partial=True)

        if serializer_replica.is_valid():
            with transaction.atomic(using='replica'):
                serializer_replica.save(using='replica')  # Save to 'replica' database
            return serializer.data

    return serializer.errors
```

Cache Testing

Case 1:

Requesting Item for the First Time: When we make a GET request to retrieve item information, it's not in the cache initially. The request is forwarded to replica 1 to fetch the data, as indicated by the log message "data is not cached." The response is then returned with the item details.




```
Forwarding request to replica: 1
[05/Nov/2024 14:31:45] "GET /FrontEnd_service/info/1 HTTP/1.1" 200 74
data is not cached
```

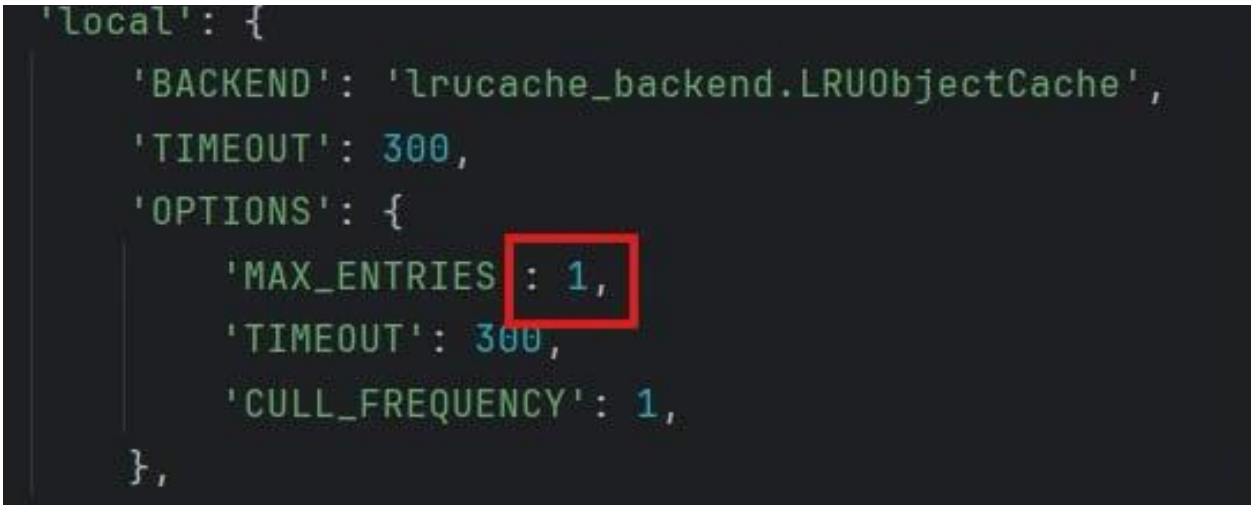
Case 2:

Requesting Item After It's Cached: In this second request, the item is already stored in the cache. The log confirms "data is cached," so the data is retrieved directly from the cache instead of querying the replica again. This makes the response faster and reduces load on the backend.

```
Forwarding request to replica: 1
[05/Nov/2024 14:31:45] "GET /FrontEnd_service/info/1 HTTP/1.1" 200 74
data is not cached
[05/Nov/2024 14:34:26] "GET /FrontEnd_service/info/1 HTTP/1.1" 200 74
data is cached
```

Case 3:

Cache is Full: We set the cache size to hold only 1 item for testing. When we try to fetch information for a new item (element 2), the cache doesn't have space to store it because it already has one item. This means a cache miss occurs, and the data has to be fetched directly from the backend. This test shows how limited cache size affects data retrieval.




```
[05/Nov/2024 14:39:11] "GET /FrontEnd_service/info/2 HTTP/1.1" 404 58
data is not cached
```

Search API Testing:

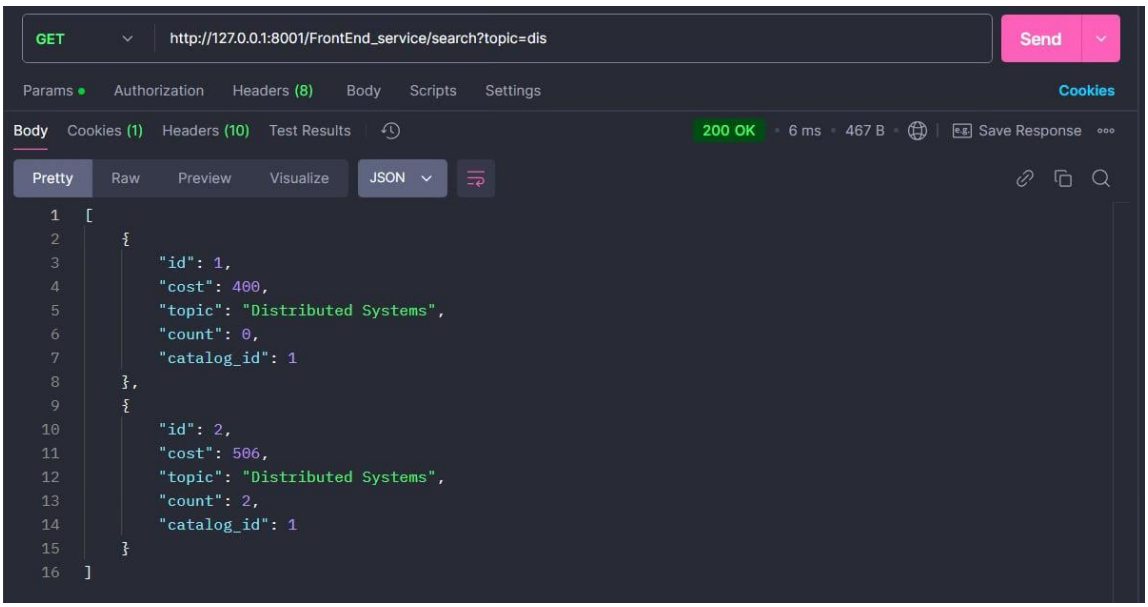
Initial Search: When the search API is called for the first time with a topic (e.g., os or dis), the request is forwarded to one of the backend replicas since the data isn't in the cache.

```
Forwarding request to replica: 1
[05/Nov/2024 14:41:43] "GET /FrontEnd_service/search?topic=os HTTP/1.1" 200 2
Forwarding request to replica: 2
[05/Nov/2024 14:41:50] "GET /FrontEnd_service/search?topic=dis HTTP/1.1" 200 2
[05/Nov/2024 14:41:53] "GET /FrontEnd_service/search?topic=dis HTTP/1.1" 200 2
Data retrieved from cache
```

Subsequent Searches: After the data is cached, repeated requests for the same topic retrieve the data directly from the cache, as shown in the log messages. This speeds up response times and reduces load on the backend.

```
Forwarding request to replica: 1
[05/Nov/2024 15:05:55] "GET /FrontEnd_service/search?topic=dis HTTP/1.1" 200 151
Data retrieved from cache
[05/Nov/2024 15:06:20] "GET /FrontEnd_service/search?topic=dis HTTP/1.1" 200 151
```

Response Format: The response provides detailed information for each matched item, including id, cost, topic, count, and catalog_id, verifying that the search function returns accurate and complete data.



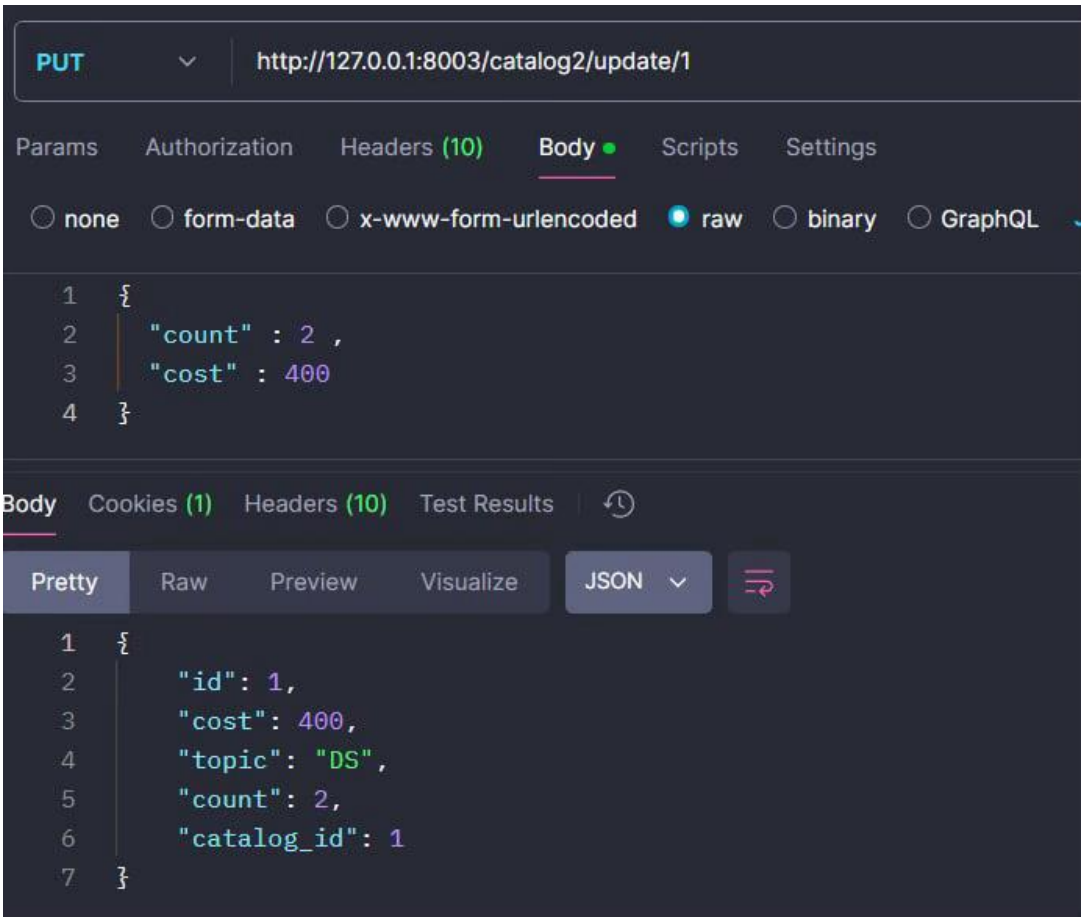
Invalidate Element After Update:

Cache Invalidation: When an item is updated (using a PUT request to modify attributes like count and cost), the system triggers the invalidateCache endpoint to remove the

outdated data from the cache. The log confirms the cache invalidation for that specific item (id: 1).

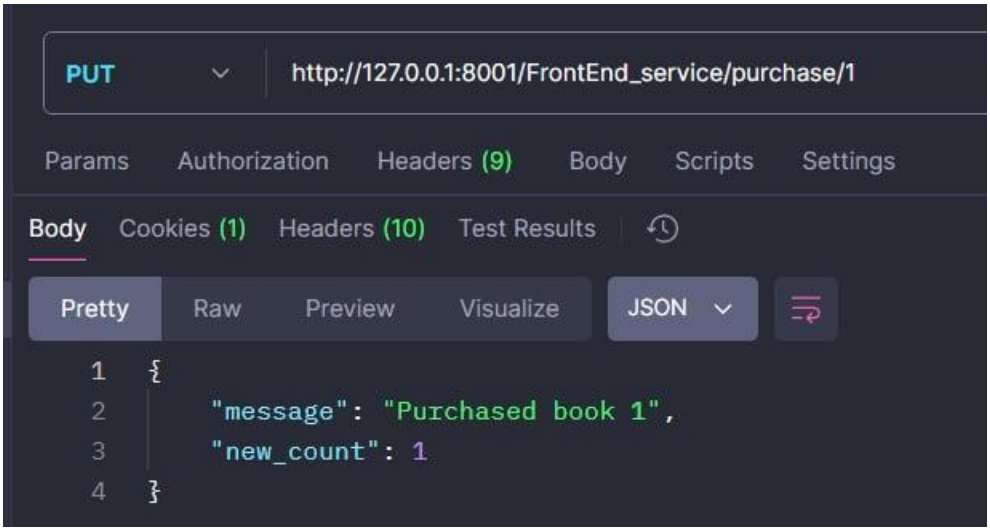
```
[05/Nov/2024 15:17:03] "GET /FrontEnd_service/invalidateCache/1 HTTP/1.1" 200 34
invalidate
1
```

Updated Data: After invalidating the cache, the updated item details (new count and cost values) are correctly reflected in the response. This ensures that any subsequent request will retrieve the latest data from the backend instead of outdated cache entries.



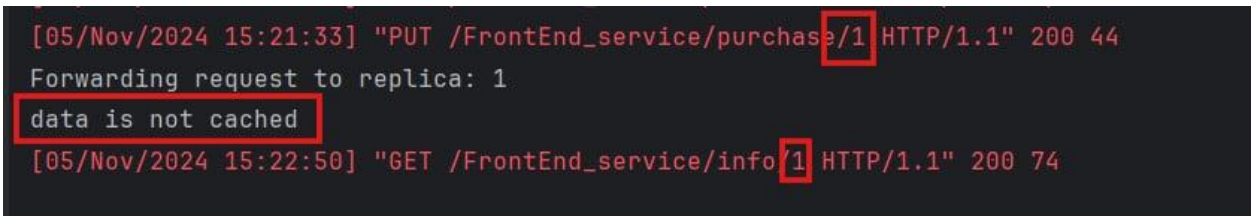
Purchase Update:

A PUT request is made to purchase the item (book with id: 1), which decreases its count. The response confirms the purchase with a message and shows the new count.



Cache Invalidation:

After the purchase, the system invalidates the cache for this item (id: 1). This is confirmed by the log showing "data is not cached" when trying to access it again. Any subsequent request for this item will retrieve fresh data from the backend, ensuring accurate and updated information.



Round Robin Method with Two Replicas in InfoController:

Round-Robin Load Balancing:

The code uses a round-robin approach to distribute requests between two replicas (replica_num is set to either 1 or 2). Each time there's a cache miss, it forwards the request to the next replica in the cycle, alternating between them.

Caching:

If data for a particular item isn't in the cache, it fetches it from one of the replicas and stores it in the cache for future requests. The next time this item is requested, it can be served directly from the cache, bypassing the replica.

```python
from rest_framework import viewsets
from rest_framework.response import Response
from fe.method.info import get_info
from django.core.cache import caches
import itertools

class InfoController(viewsets.ViewSet):  2 usages
    # Define a list of backend replicas
    BACKEND_REPLICAS = [1, 2]

    # Create a round-robin iterator for replicas
    replica_cycle = itertools.cycle(BACKEND_REPLICAS)
    def retrieve(self, request, pk=None):
        # Try to get data from the cache
        cache_key = "info_" + pk
        info_data = caches['local'].get(cache_key)
        if not info_data:
            # If cache miss, fetch data and store it in the cache
            replica_num = next(self.replica_cycle)
            print(f"Forwarding request to replica: {replica_num}")
            info_data, status = get_info(pk, replica_num)
            print("data is not cached")
            caches['local'].set(cache_key, info_data)  # Cache data for 5 minutes
        else:
            # If data is cached, set the status to 200 (OK)
            print("data is cached")
            status = 200
        return Response(info_data, status=status)
```

SearchController:

Uses round-robin to alternate search requests between replicas. If data isn't in the cache, it forwards the request to the next replica in line, balancing the load.

```python
class SearchController(viewsets.ViewSet):  2 usages
    # Define a list of backend replicas
    BACKEND_REPLICAS = [1, 2]

    # Create a round-robin iterator for replicas
    replica_cycle = itertools.cycle(BACKEND_REPLICAS)

    def list(self, request):
        topic = request.query_params.get('topic')

        if not topic:
            return Response( data: {"error": "Book Topic parameter is required"}, status=400)

        # Create a unique cache key for the topic
        cache_key = f"search_{topic}"
        search_data = caches['local'].get(cache_key)

        if search_data is None:
            # Cache miss: Fetch fresh data using round-robin load balancing
            replica_num = next(self.replica_cycle)  # Get the next replica in round-robin order
            print(f"Forwarding request to replica: {replica_num}")
            # Modify search_books to accept the replica URL if needed
            search_data, status = search_books(topic, replica_num)
            # Cache the fetched data for 5 minutes
            caches['local'].set(cache_key, search_data)
        else:
            # Cache hit, set the status to 200 (OK)
            print("Data retrieved from cache")
            status = 200

        return Response(search_data, status=status)
```

PurchaseController:

Also uses round-robin for distributing purchase updates. After an update, it invalidates the cache entry to ensure fresh data on the next request.

```python
class PurchaseController(viewsets.ViewSet):  4 usages
    # Define a list of backend replicas
    BACKEND_REPLICAS = [1, 2]
    # Create a round-robin iterator for replicas
    replica_cycle = itertools.cycle(BACKEND_REPLICAS)
    def update(self, request, pk=None):  1 usage
        if not pk:
            return Response( data: {"error": "Book ID is required for updating"}, status=400)
        # Forward the request to update the purchase in the Order service
        replica_num = next(self.replica_cycle)
        print(f"Forwarding request to replica: {replica_num}")
        update_data, status = make_purchase(pk, request.data, replica_num)
        """Invalidates the cache entry for the given key."""
        key = "info_" + pk
        # Properly delete the cache key
        caches['local'].delete(key)
        return Response(update_data, status=status)
```

First Request: Goes to replica 1. Next Request: Goes to replica 2. And so on: Alternates between replicas for each new request.

```
Forwarding request to replica: 1
[05/Nov/2024 14:41:43] "GET /FrontEnd_service/search?topic=os HTTP/1.1" 200 2
Forwarding request to replica: 2
[05/Nov/2024 14:41:50] "GET /FrontEnd_service/search?topic=dis HTTP/1.1" 200 2
[05/Nov/2024 14:41:53] "GET /FrontEnd_service/search?topic=dis HTTP/1.1" 200 2
Data retrieved from cache
```

The choice of replica is determined by the replica_num parameter passed to each API method.

get_info: Checks replica_num. If it's 1, it sends the request to URL_CATALOG_INFO. If it's 2, it uses URL_CATALOG_INFO2.

```python
def get_info(pk, replica_num):  2 usages
    if replica_num == 1 :
        response = requests.get(URL_CATALOG_INFO + str(pk))
    else :
        response = requests.get(URL_CATALOG_INFO2 + str(pk))
    if response.status_code == 200:
        return response.json(), response.status_code
    return {"error": "Failed to fetch book info from Catalog service"}, response.status_code
```

make_purchase: Uses replica_num to decide which URL to send the purchase request to. URL_ORDER_PURCHASE is used for replica_num = 1, and URL_ORDER_PURCHASE2 for replica_num = 2.

```python
def make_purchase(pk, data, replica_num):  2 usages
    if replica_num == 1 :
        response = requests.put( url: f"{URL_ORDER_PURCHASE}{pk}", json=data)
    else :
        response = requests.put( url: f"{URL_ORDER_PURCHASE2}{pk}", json=data)
    if response.status_code == 200:
        return response.json(), response.status_code
    return {"error": "Failed to update purchase in Order service"}, response.status_code
```

search_books: Based on replica_num, it selects either URL_CATALOG_SEARCH or URL_CATALOG_SEARCH2.

```python
def search_books(topic, replica_num):  2 usages
    if replica_num == 1 :
        search_url = f"{URL_CATALOG_SEARCH}?topic={topic}"
    else :
        search_url = f"{URL_CATALOG_SEARCH2}?topic={topic}"
    response = requests.get(search_url)
    if response.status_code == 200:
        return response.json(), response.status_code
    return {"error": "Failed to fetch search results from Catalog service"}, response.status_code
```

By passing replica_num, the system can alternate between replicas, ensuring load is balanced across them.

Here are the urls of replicas 1 and 2 of order and catalog

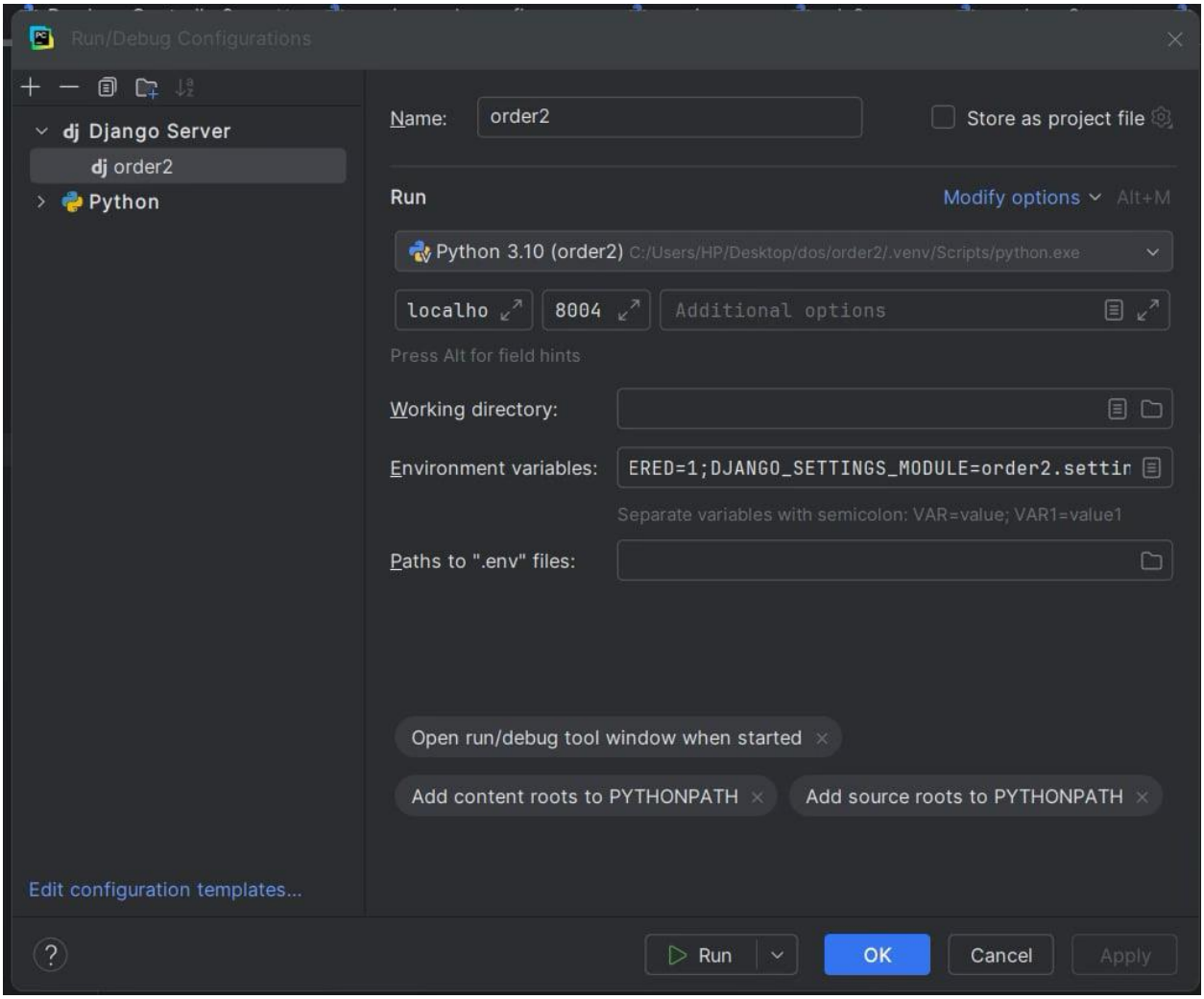
```
URL_CATALOG_SEARCH = "http://127.0.0.1:8000/catalog/search"
URL_CATALOG_INFO = "http://127.0.0.1:8000/catalog/info/"
URL_ORDER_PURCHASE = "http://127.0.0.1:8002/order_service/purchase/"

URL_CATALOG_SEARCH2 = "http://127.0.0.1:8003/catalog2/search"
URL_CATALOG_INFO2 = "http://127.0.0.1:8003/catalog2/info/"
URL_ORDER_PURCHASE2 = "http://127.0.0.1:8004/order_service2/purchase2/"
```

Running Program:

To run each Django service independently, we set up configurations in the development environment:

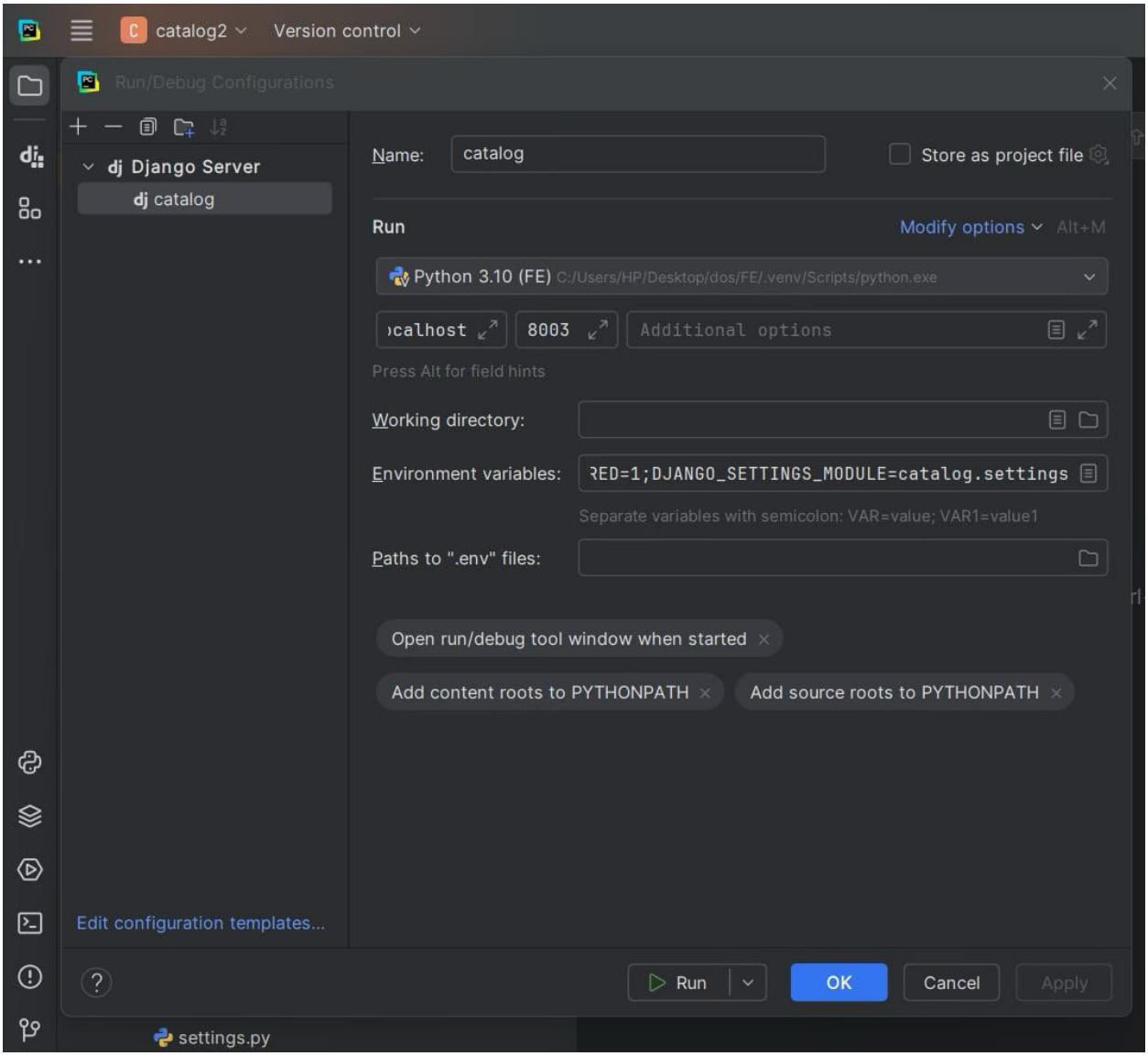
Order2 Service:

Runs on port 8004 with Django settings configured via DJANGO_SETTINGS_MODULE=order2.settings. This setup allows the Order2 service to be accessed at localhost:8004.

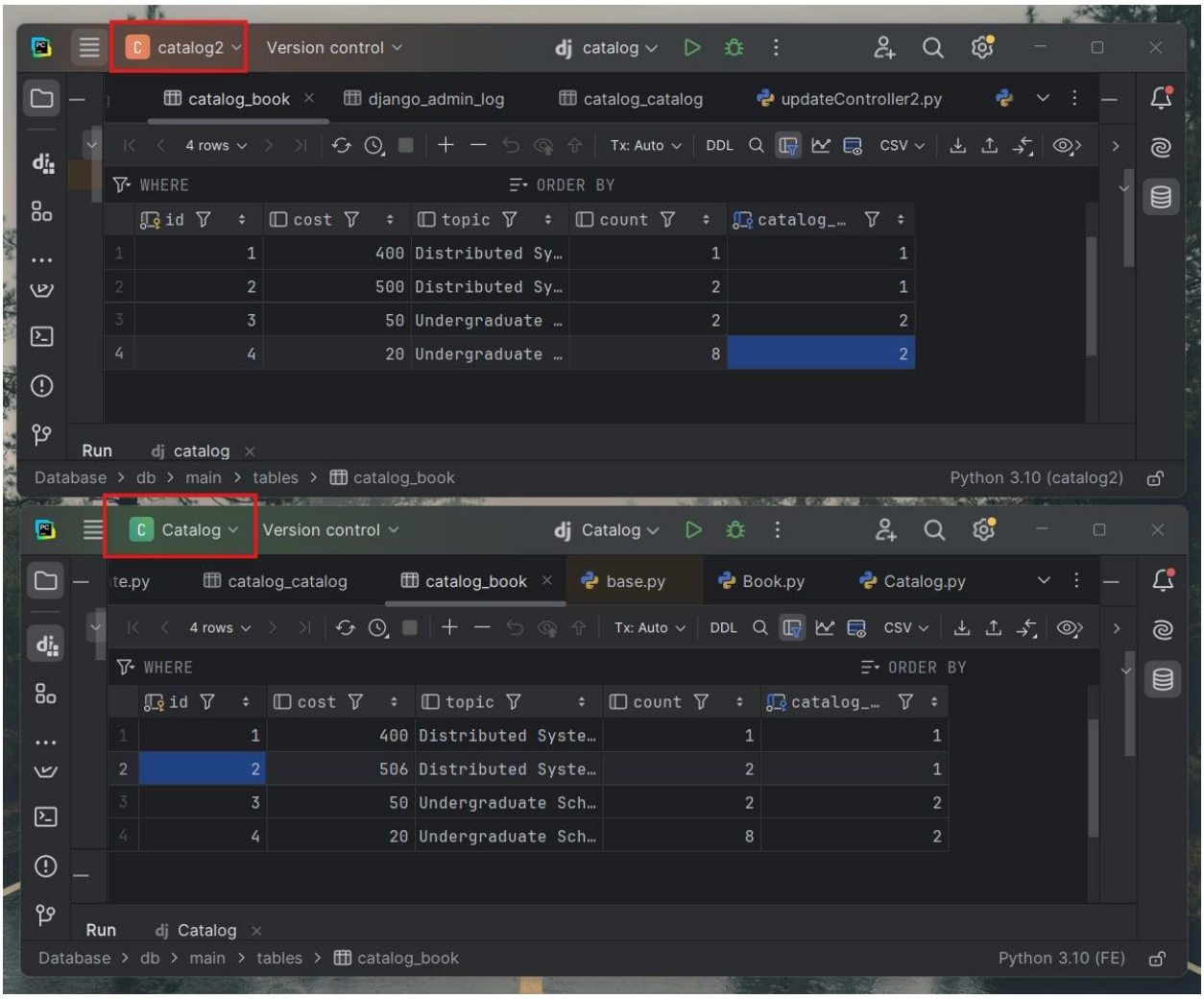


Catalog2 Service:

Runs on port 8003 with the settings configured via DJANGO_SETTINGS_MODULE=catalog.settings. This lets the Catalog2 service operate on localhost:8003.



in this setup, we have two separate databases, one for each service (Catalog and Catalog2), each holding the same set of data. This allows both services to operate independently while accessing the same catalog information. By replicating the database content across Catalog and Catalog2, we achieve redundancy, enabling the system to distribute requests across these services. This configuration supports load balancing and improves availability, ensuring that if one service goes down, the other can still serve requests.



In this configuration, the settings.py file for both Catalog and Catalog2 defines two database connections: default and replica.

default: This is the primary database, likely used for standard read and write operations.

replica: This secondary database might be used as a backup or for load distribution in read-heavy operations, allowing for more efficient handling of queries without putting too much load on a single database.

```
# Database
# https://docs.djangoproject.com/en/5.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    },
    'replica': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'C:/Users/HP/Desktop/dos/catalog2/db.sqlite3',
    }
}
```

Fetch Book in Default Database: It first retrieves the book entry from the default database by book_id.

Serialize and Validate: The data for the update is serialized and validated. If valid, it proceeds to update.

Update Default Database: Using a transaction, the method updates the default database with the new data, ensuring atomicity.

Fetch Book in Replica Database: After successfully updating the default database, it retrieves the same book entry from the replica database.

Update Replica Database: If validation is successful for the replica's serialized data, it updates the replica database with the same information, again ensuring atomicity.

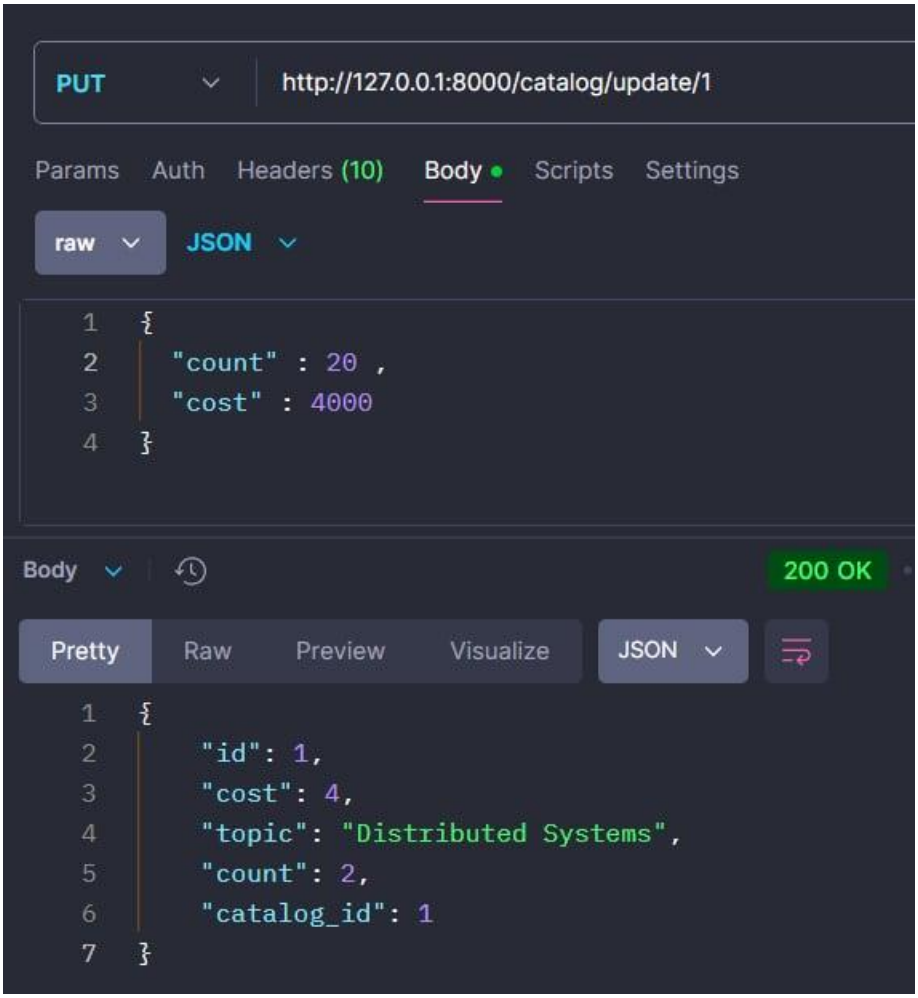
```python
def update_book_info(book_id, data):  2 usages
    # Retrieve the book object from the 'default' database
    book = get_object_or_404(Book.objects.using('default'), pk=book_id)

    # Serialize the data
    serializer = BookSerializer(book, data=data, partial=True)

    # If valid, update both databases
    if serializer.is_valid():
        with transaction.atomic(using='default'):
            serializer.save(using='default')  # Save to 'default' database

        # Update the replica
        # Fetch a fresh instance from the replica database, to update it independently
        book_replica = Book.objects.using('replica').get(pk=book_id)
        serializer_replica = BookSerializer(book_replica, data=data, partial=True)

        if serializer_replica.is_valid():
            with transaction.atomic(using='replica'):
                serializer_replica.save(using='replica')  # Save to 'replica' database
            return serializer.data

    return serializer.errors
```

Dual Database Update: When an update is requested, the update_book_info method updates both the default (primary) and replica databases for data consistency.

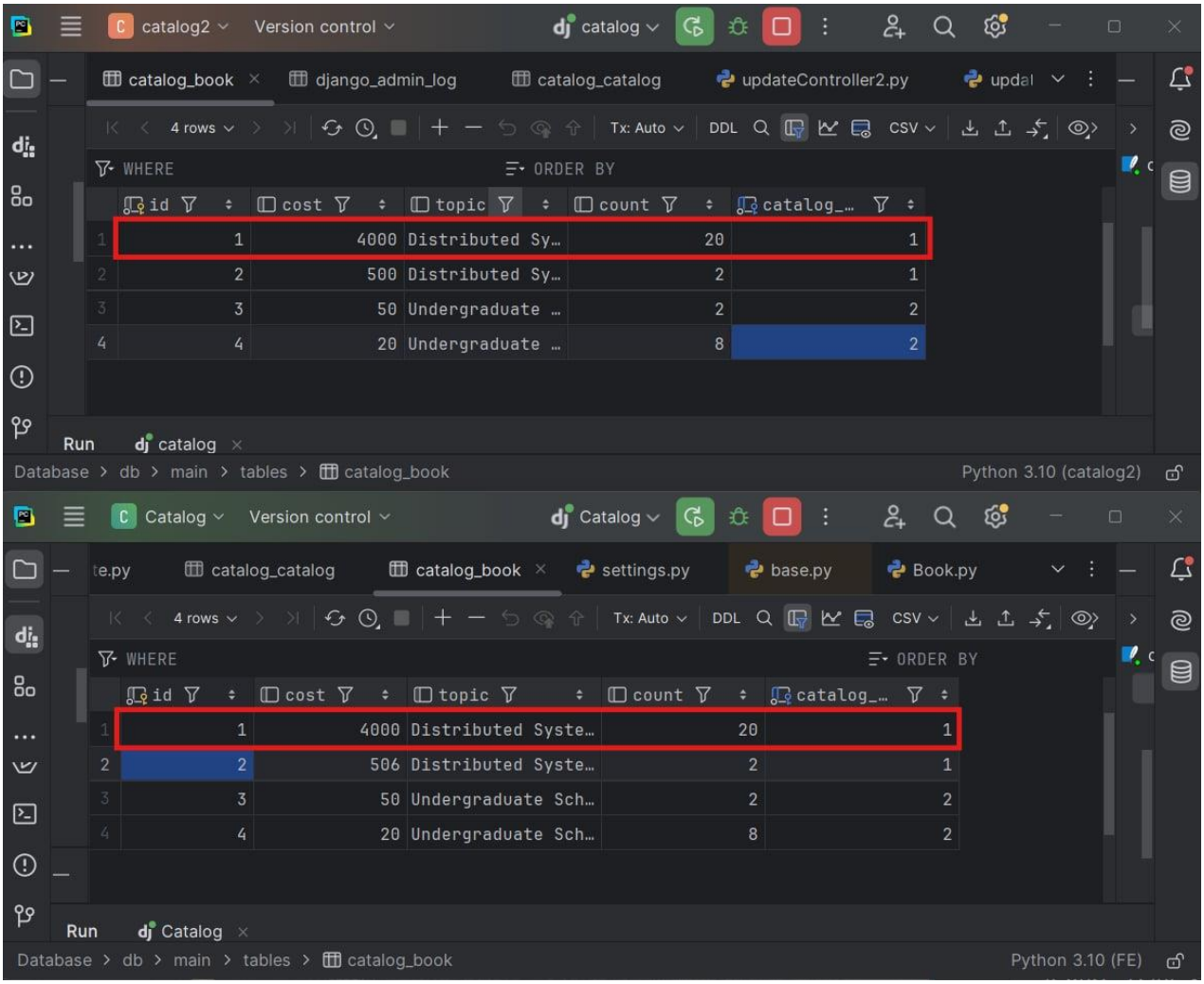
Database Transactions:

Each update in both databases is wrapped in an atomic transaction, ensuring that updates are reliable and isolated.

Example in Action: When a book's count and cost are updated, these changes reflect in both catalog and catalog2 databases, maintaining synchronization between the replicas.



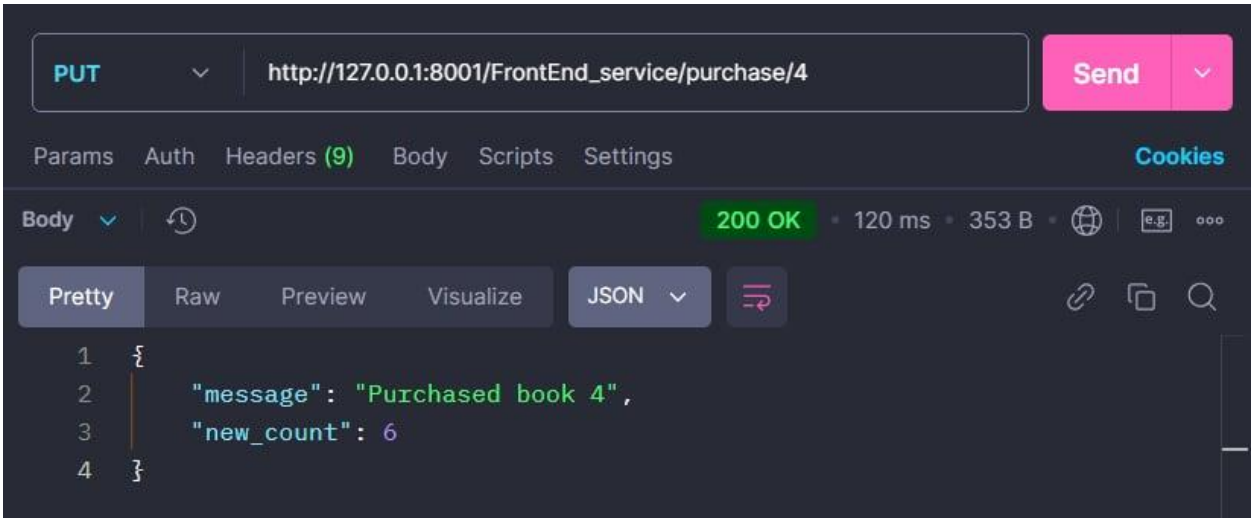
As shown in your screenshots, any changes to the data, such as count: 20 and cost: 4000, are visible in both databases, ensuring each query retrieves consistent and up-to-date information across both replicas.



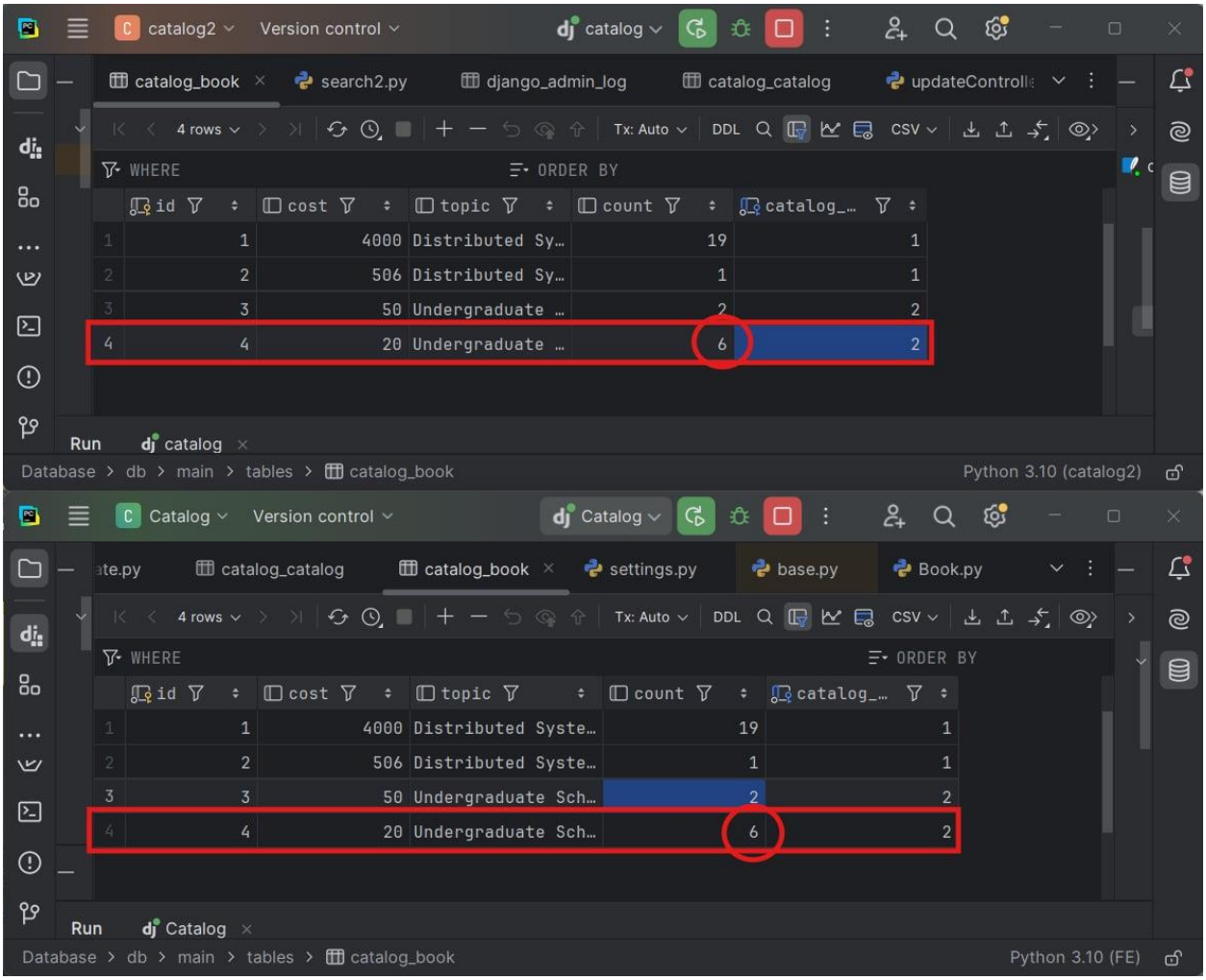
Purchase Request: A PUT request is sent to the purchase endpoint for book ID 4.

Response: “Purchased book 4” and provides the new count of the book "new_count": 6.

Inventory Update: The backend has decremented the book's count by the appropriate amount, reflecting the purchase in the database. This change should propagate to both the default and replica databases to ensure consistency.



After the purchase, the count for book ID 4 has been updated to 6 in both the default and replica databases, ensuring data consistency across the system. This confirms that the purchase operation correctly modified the inventory count in both replicas, maintaining synchronization between the primary and replica databases.



Timing Results:

			Time
FE	info	Info without cache forwarded to replica 1	24ms
		Info without cache forwarded to replica 2	17ms
		Info with cache	9ms
	Search	Search with cache	4ms
		Search with cache forwarded to replica 1	18ms
		Search with cache forwarded to replica 2	19ms
	Purchase		
		Purchase forwarded to replica 1	27ms
		Purchase forwarded to replica 2	20ms
Order	Purchase		
		Purchase forwarded to replica 1	30
		Purchase forwarded to replica 2	24
Catalog	Info	Info from replica 1	6ms
		Info from replica 2	10ms
	Search	Search from replica 1	8ms
		Search from replica 2	12ms
	Update	Update from replica 1	33ms
		Update from replica 2	30ms

The Frontend (FE) service is a lot faster when it uses cached data, responding in just 4-9 milliseconds. Without the cache, it takes a bit longer, around 17-24 milliseconds. This shows that caching is really helpful for common requests like fetching info or search results.

For the backend services, response times differ a little between replicas. The Order service's purchase action takes about 24-30 milliseconds, while Catalog's info and search actions are faster, around 6-12 milliseconds on average. The update action in Catalog is the slowest, taking around 30-33 milliseconds on each replica.

In summary, caching greatly speeds up the FE service, and the backend services perform pretty evenly across replicas, with only slight timing differences.

this project successfully boosted Bazar.com's efficiency with caching, replication, and smart load distribution. Caching really helped cut down response times on popular items, while using replicas for order and catalog services kept things balanced and reliable. The results show faster responses for users, more consistent data, and better system reliability. With these changes, Bazar.com is ready to handle more traffic smoothly and give users a quicker shopping experience.

Note:

No need to rename functions like in controller2 but we did to distinguish the change in new duplicate.