MongoDB Bookstore

Erin Cullen

600531

Group V2-T

27/05/2025

DBD381
Project

Contents

Executive Summary

This project implements a fully featured e-commerce bookstore using MongoDB as its database. The system demonstrates the simplicities of setting up a NoSQL database but also the complexity of using it.

Based off of Belgium Campus DBA381 Project using a bookstore running on Oracle, and comparing the results of a NoSQL vs RDB implementations.

Project Scope and Requirements

Business Context

The project addresses the need for a simple yet effective, scalable and flexible database to house all the information of the bookstore.

- Dynamic inventory management with real-time stock updates.
- Customer account management and authentication.
- Order processing and transaction history.
- Scalable product/inventory catalogue.

Functional Requirements

Backend:

- A simple yet reliable NoSQL database.
- A database that will allow layman users to understand or interact with the database without a Front-End.
- Feature scalability to handle increasing inventory catalogue and customer base.
- Feature high performance with sub-second response times.
- Implement Consistency to maintain data integrity across concurrent transactions
- MongoDB features the flexibility to change the database with relative ease.

Frontend:

- User Management with Customer Registration, Authentication and session management.
- Inventory Management with a flexible book catalogue with real-time stock numbers.
- Order Processing using a shopping cart to minimize transaction time with the database (reducing lock and other concurrency errors)

Design Considerations and Architectural Decisions

Database Technology

MongoDB was selected as the database for the following reasons:

- Document-Oriented Structure provides a natural fit for inventory catalogue.
- Horizontal Scalability provides the option to scale (shard) across servers to reduce costs.
- Flexibility allows document variations without database redesign.
- BSON data storage allows for simple database interactions with a JSON focused website.
- NoSQL requires less storage than relational databases. (EnterpriseDB) (p26).

Disadvantages:

- Lack of predefined structure leads to eventual consistency, but lacks that factor to begin with.
- Requires discipline due to the lack of a rigid structure.

Architectural Decisions

Implemented a 3-tier architecture:

Presentation Layer (React/NextJS website)

↓

Business Logic Layer (Flask/Python)

↓

Data Layer (MongoDB)

Most of the innovation took place in the business logic layer, python held all transactions and had a direct API link to the presentation layer. This reduced latency, increased flexibility and provided a layer of segregation between the data and the frontend, also improving security.

The Python FLASK server logged customers in using the Customers collection from MongoDB, and held the logged in customer in a FLASK session.

This eliminates the need for continuous connection to the database and is rather handled in python with sessions, and when queries are made, they make the connection and execute their respective query and then close (*Connection Per Request)*, which almost eliminates concurrent connections and their related issues.

Implementation and Deployment Plan

Technology Stack:

- Frontend: NextJS 15.3.2, React, TypeScript, Tailwind CSS
- Backend: Flask (Python), Flask-CORS
- Database: MongoDB Community Server

Database Configuration:

- MONGO_URI = "mongodb://localhost:27017/"
- Database: "bookstore"
- Collections: ["books", "customers", "orders"]

Local Development Architecture:

- MongoDB: Local instance on port 27017
- Flask API: Development server on port 5000
- Next.js Frontend: Development server on port 3000
- Communication: Direct HTTP/CORS between components

Distributed Environment Strategy

MongoDB Atlas: Cloud hosted MongoDB, Provides sharding, backup and high availability as a service provided by another company, eliminating the overhead for the bookstore to manage this facet.

Docker: containerization of the Front-End and Back-End to improve performance and simplify installation. Allows a manager to install the Back-End and Front-End server implementations on multiple machines with ease using prefabricated setup parameters.

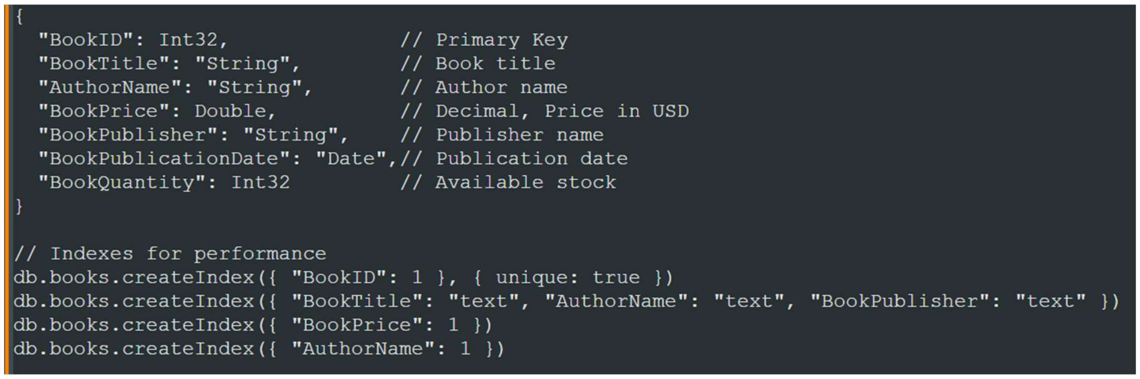
Load Balancing: similar to sharding the database, we load balance the Front-End and Back-End across servers, this spreads the overhead over multiple servers.

CDN Integration: a common practice for website deployment, which allows users to access the closest geographically available instance of the website.

Schema Design & Data Model

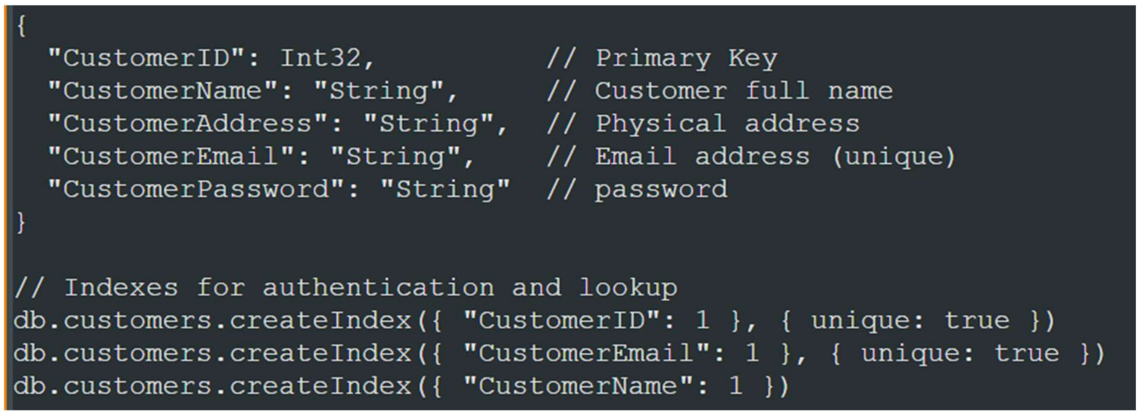
Schema Design:

Books Collection:



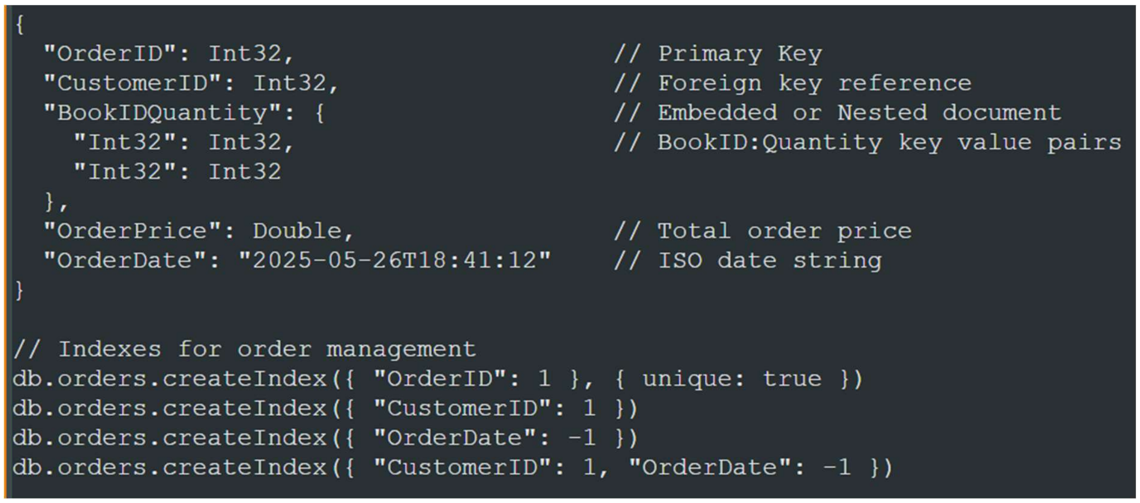
```
{
  "BookID": Int32,              // Primary Key
  "BookTitle": "String",        // Book title
  "AuthorName": "String",       // Author name
  "BookPrice": Double,          // Decimal, Price in USD
  "BookPublisher": "String",    // Publisher name
  "BookPublicationDate": "Date",// Publication date
  "BookQuantity": Int32         // Available stock
}

// Indexes for performance
db.books.createIndex({ "BookID": 1 }, { unique: true })
db.books.createIndex({ "BookTitle": "text", "AuthorName": "text", "BookPublisher": "text" })
db.books.createIndex({ "BookPrice": 1 })
db.books.createIndex({ "AuthorName": 1 })
```

Customers Collection:



```
{
  "CustomerID": Int32,           // Primary Key
  "CustomerName": "String",      // Customer full name
  "CustomerAddress": "String",   // Physical address
  "CustomerEmail": "String",     // Email address (unique)
  "CustomerPassword": "String"   // password
}

// Indexes for authentication and lookup
db.customers.createIndex({ "CustomerID": 1 }, { unique: true })
db.customers.createIndex({ "CustomerEmail": 1 }, { unique: true })
db.customers.createIndex({ "CustomerName": 1 })
```

Orders Collection:



```
{
  "OrderID": Int32,                      // Primary Key
  "CustomerID": Int32,                   // Foreign key reference
  "BookIDQuantity": {                    // Embedded or Nested document
    "Int32": Int32,                      // BookID:Quantity key value pairs
    "Int32": Int32
  },
  "OrderPrice": Double,                  // Total order price
  "OrderDate": "2025-05-26T18:41:12"     // ISO date string
}

// Indexes for order management
db.orders.createIndex({ "OrderID": 1 }, { unique: true })
db.orders.createIndex({ "CustomerID": 1 })
db.orders.createIndex({ "OrderDate": -1 })
db.orders.createIndex({ "CustomerID": 1, "OrderDate": -1 })
```

Data Modelling:

Normalization modelling:

Orders Nested Document uses a NoSQL feature of nesting documents, this allows the database to store key:value pairs for books and their quantities in each order (eliminating a layer of normalization that was seen in the normal relational database version of the bookstore (DBA 381))

Relationship modelling:

The orders collection acts as the central collection that connects books and customers. NoSQL databases however do not handle relationships like a RDB does.

The Flask server handled relationships, another advantage of the business logic layer. The Flask server also utilized *Atomic Operations* creating an order and updating books stock levels in a single transaction. Finally, utilizing the cart feature (Flask server holds the cart in memory), the Flask server validates available stock before proceeding with the transaction (utilizing connect-transact-disconnect mentality, eliminating the need for rollback)

Implementation Process

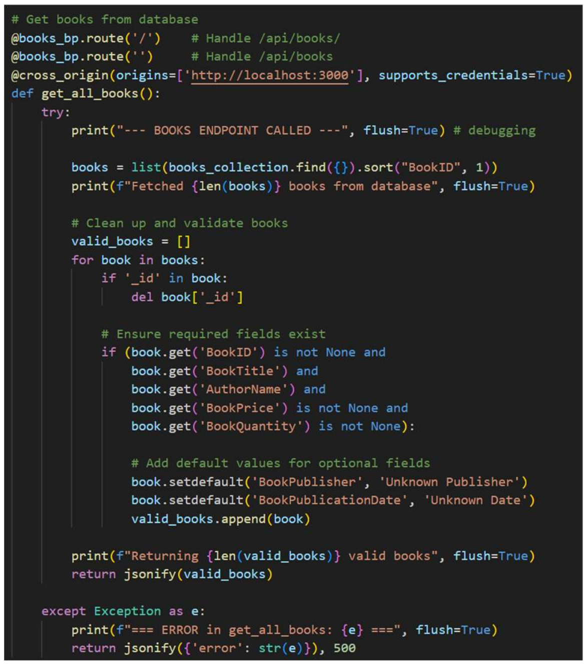
Due to the use of a NoSQL server, the implementation process was rapid, with the design of 3 collections and the choice of document nesting to mitigate a layer of normalization, the process was absolutely minimal. However “too much haste is too little speed”, because all the time gained in creating the database was lost when implementing a business logic layer that enforced structure and mitigated data errors.

MongoDB Compass was used to create the database, collection and documents from 2 csv files originating from DBA381 Project (the project required students to generate their own data, it was reused for this to eliminate that tedious process). Compass is extremely easy to use and importing data from a csv is basic.

A Flask-NextJS template created by Erin Cullen was used, and is structured it 2 parts, the Front-End and the Back-End, the Back-End was adapted to interface with the database with certain operations and then the Front-End was simply built to push and pull to and from the Back-End.

Challenges

Data Quality Issues:
there were a few books documents that imported incorrectly (came from a csv file), the python business logic layer had to therefore incorporate some data cleaning functions when reading from the database.



```python
# Get books from database
@books_bp.route('/')    # Handle /api/books/
@books_bp.route('')     # Handle /api/books
@cross_origin(origins=['http://localhost:3000'], supports_credentials=True)
def get_all_books():
    try:
        print("--- BOOKS ENDPOINT CALLED ---", flush=True) # debugging

        books = list(books_collection.find({}).sort("BookID", 1))
        print(f"Fetched {len(books)} books from database", flush=True)

        # Clean up and validate books
        valid_books = []
        for book in books:
            if '_id' in book:
                del book['_id']

            # Ensure required fields exist
            if (book.get('BookID') is not None and
                book.get('BookTitle') and
                book.get('AuthorName') and
                book.get('BookPrice') is not None and
                book.get('BookQuantity') is not None):

                # Add default values for optional fields
                book.setdefault('BookPublisher', 'Unknown Publisher')
                book.setdefault('BookPublicationDate', 'Unknown Date')
                valid_books.append(book)

        print(f"Returning {len(valid_books)} valid books", flush=True)
        return jsonify(valid_books)

    except Exception as e:
        print(f"=== ERROR in get_all_books: {e} ===", flush=True)
        return jsonify({'error': str(e)}), 500
```

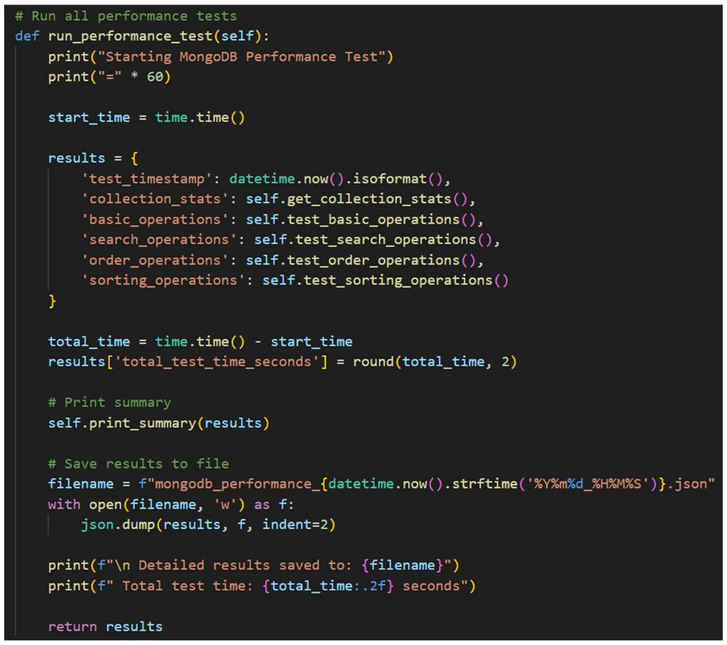
As can be seen in the valid_books array, which only pulls documents with information in certain fields (ID, Price, Quantity)

NextJS Proxy Issues:
Exclusive of the database, the communication between Flask Back-End and NextJS Front-End had initially used a proxy set up in the NextJS template, this was eliminated as it kept timing out when loading the entire list of books (even though the list was read form the database in a matter of milliseconds). The Proxy was eliminated and a direct connection Flask CORS was opted in, resulting in a 200ms response time reduction across the board (measured from database to Front-End)

Testing and Validation

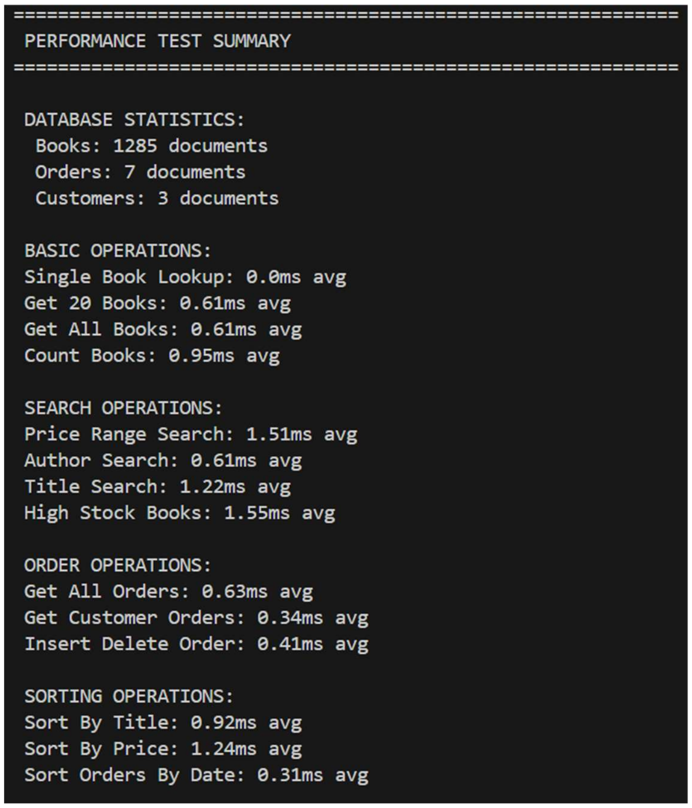
Performance Testing

An elaborate performance script was written in python that tests a full suite of database operations from basic search operations, to elaborate INSERT and Delete procedures, while allowing these procedures to be timed:



```python
# Run all performance tests
def run_performance_test(self):
    print("Starting MongoDB Performance Test")
    print("=" * 60)

    start_time = time.time()

    results = {
        'test_timestamp': datetime.now().isoformat(),
        'collection_stats': self.get_collection_stats(),
        'basic_operations': self.test_basic_operations(),
        'search_operations': self.test_search_operations(),
        'order_operations': self.test_order_operations(),
        'sorting_operations': self.test_sorting_operations()
    }

    total_time = time.time() - start_time
    results['total_test_time_seconds'] = round(total_time, 2)

    # Print summary
    self.print_summary(results)

    # Save results to file
    filename = f"mongodb_performance_{datetime.now().strftime('%Y%m%d_%H%M%S')}.json"
    with open(filename, 'w') as f:
        json.dump(results, f, indent=2)

    print(f"\n Detailed results saved to: {filename}")
    print(f" Total test time: {total_time:.2f} seconds")

    return results
```

Here is the main performance running method that runs a whole manner of operations, record the response times and then prints them neatly in console as well as in a JSON file.



```
============================================================
 PERFORMANCE TEST SUMMARY
============================================================

 DATABASE STATISTICS:
  Books: 1285 documents
  Orders: 7 documents
  Customers: 3 documents

 BASIC OPERATIONS:
 Single Book Lookup: 0.0ms avg
 Get 20 Books: 0.61ms avg
 Get All Books: 0.61ms avg
 Count Books: 0.95ms avg

 SEARCH OPERATIONS:
 Price Range Search: 1.51ms avg
 Author Search: 0.61ms avg
 Title Search: 1.22ms avg
 High Stock Books: 1.55ms avg

 ORDER OPERATIONS:
 Get All Orders: 0.63ms avg
 Get Customer Orders: 0.34ms avg
 Insert Delete Order: 0.41ms avg

 SORTING OPERATIONS:
 Sort By Title: 0.92ms avg
 Sort By Price: 1.24ms avg
 Sort Orders By Date: 0.31ms avg
```

Performance across the board shows split second response times.

Even with 1285 individual book documents in the books collection, it takes 0.61ms average to get all of the documents. And these operations are measured from python which in and of itself introduces latency.

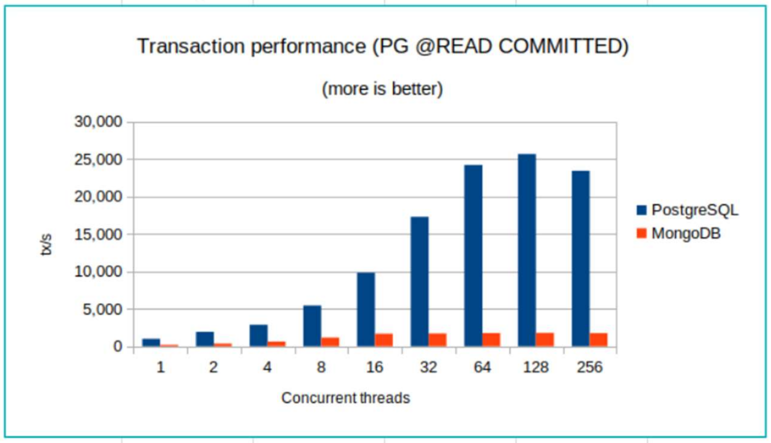
Comparing Performance

Even though MongoDB has shown extremely fast response times, the dataset is relatively small and when reviewing online sources, an equivalent Relational Database performs between 4-15 times faster under any level of concurrency (and the gap grows with more users) (EnterpriseDB)

Benchmark results

Throughput

The following graph depicts the performance comparison between MongoDB running transactions and PostgreSQL:

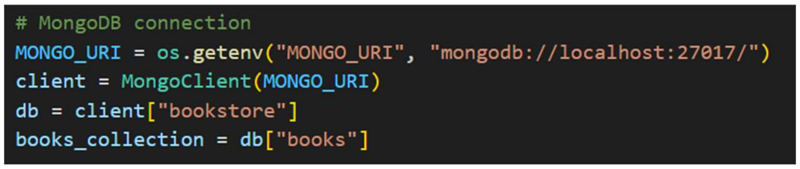


Concurrent clients	PostgreSQL TPS	MongoDB TPS
1	1,007	203
2	1,936	372
4	2,873	641
8	5,445	1,168
16	9,815	1,684
32	17,278	1,707
64	24,171	1,759
128	25,636	1,786
256	23,402	1,750

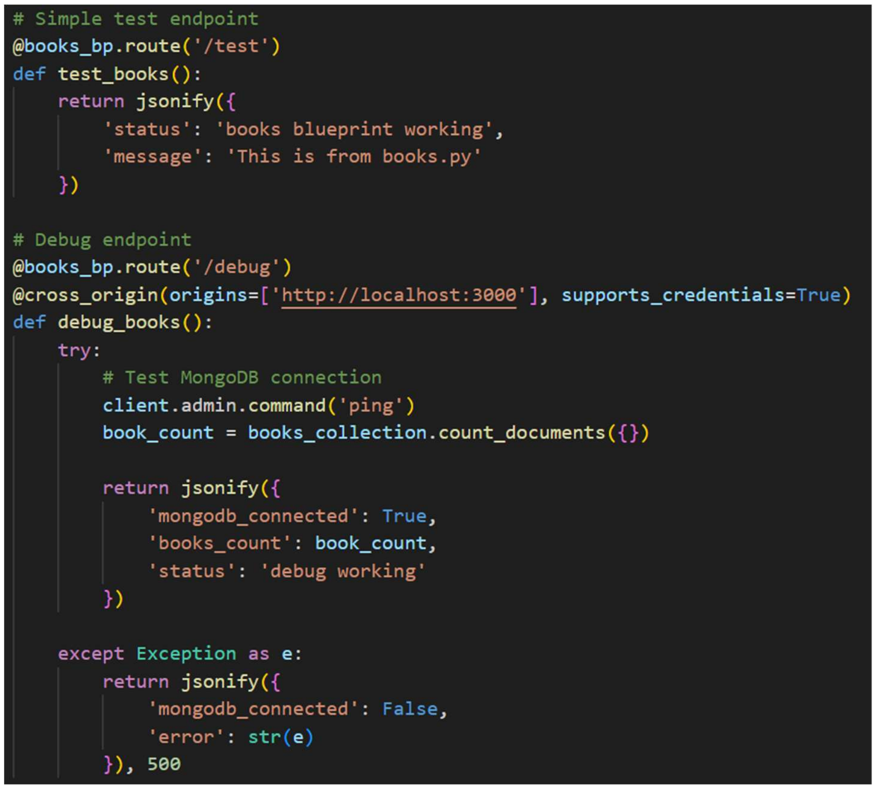
Figure 1 - EnterpriseDB (P17)

Validation

When it came to validation of the deployed application, every CRUD operation had a simple test endpoint that could be read in console, as well as a debug endpoint so that an admin can easily diagnose if there are errors between the business layer and the database.



```python
# MongoDB connection
MONGO_URI = os.getenv("MONGO_URI", "mongodb://localhost:27017/")
client = MongoClient(MONGO_URI)
db = client["bookstore"]
books_collection = db["books"]
```



```python
# Simple test endpoint
@books_bp.route('/test')
def test_books():
    return jsonify({
        'status': 'books blueprint working',
        'message': 'This is from books.py'
    })

# Debug endpoint
@books_bp.route('/debug')
@cross_origin(origins=['http://localhost:3000'], supports_credentials=True)
def debug_books():
    try:
        # Test MongoDB connection
        client.admin.command('ping')
        book_count = books_collection.count_documents({})

        return jsonify({
            'mongodb_connected': True,
            'books_count': book_count,
            'status': 'debug working'
        })

    except Exception as e:
        return jsonify({
            'mongodb_connected': False,
            'error': str(e)
        }), 500
```

This allows for rapid testing of the direct connection between the business layer and the database layer, eliminating the complexity of the Front-End when errors occur.

MongoDB Analysis

Strengths:

- Ability to vary metadata on a document by document basis.
- Allows for rapid development.
- JSON integration allows for extremely simple query structures and web integration.
- Entries in documents allow for rich datatypes such as arrays and nested documents.
- Horizontal scaling allows for simple sharding across distributed systems.
- High availability due to the simple implementation of replica sets.
- User friendly tools such as MongoDB Compass.

Weaknesses:

- Lack of planning and design means lack of structure.
- Referential integrity needs to be designed into the data business layer.
- Degraded performance due to document structure redundancy (every document holds a field+value.

Trade-Offs:

The trade offs revolve around the simplicity and user friendliness of the setup (it allows for people who aren't adept in databases to set up a relatively well performant database) versus the long term scalability and performance of the database. A traditional RDB will perform leagues better in the long term, but will require careful setup and development processes, and if something is missed in the development process, it becomes a nightmare to fix (which is where MongoDB wins).

Key Findings and Outcomes

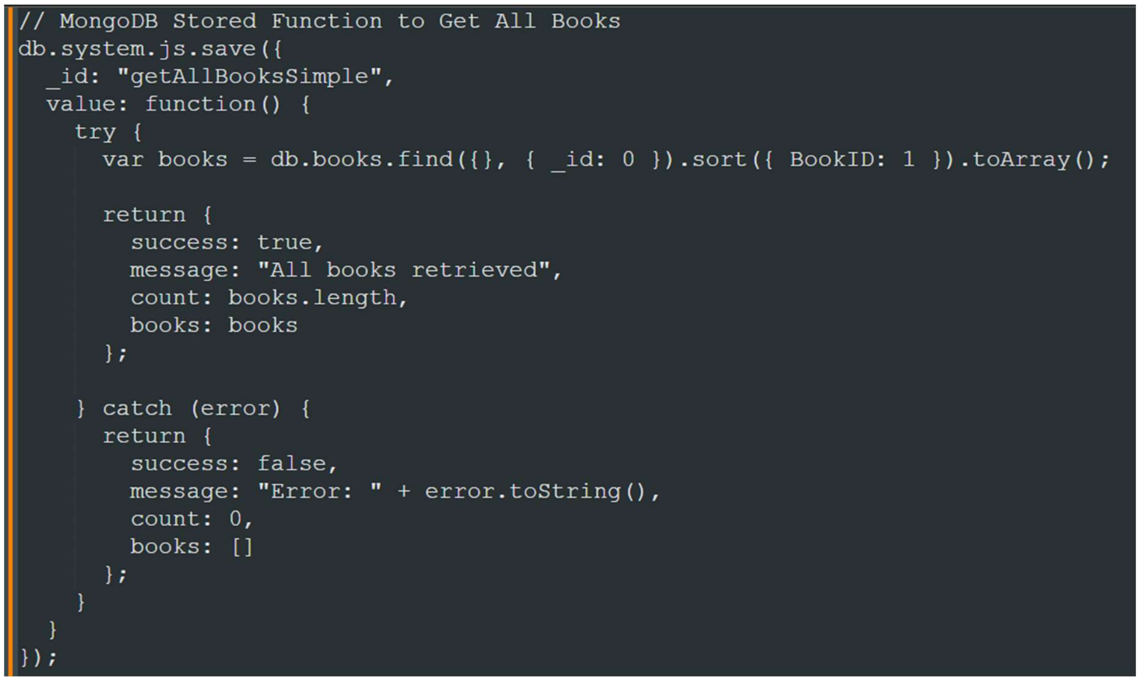
The database performs well, the front end webui runs without a hitch and allows for end-to-end order processing, allowing for customers to easily order books into a cart as well as track their orders over time.

The system uses a session ran on the Flask server to host customers on the website, and when an interaction is made between the Back-End and the Database, the Back-End uses a *Connection Per Request*.

A massive bonus of using 3 layers to build this application was clear debugging, which proved immensely useful for finding the proxy issue, and allows for measuring performance numbers at each layer.

Optimization recommendations

Improving the indexing of database records, implementing built in generic queries that can then be called by the business logic layer, all improve the simplicity and reduce the latency of the database.



```
// MongoDB Stored Function to Get All Books
db.system.js.save({
  _id: "getAllBooksSimple",
  value: function() {
    try {
      var books = db.books.find({}, { _id: 0 }).sort({ BookID: 1 }).toArray();

      return {
        success: true,
        message: "All books retrieved",
        count: books.length,
        books: books
      };

    } catch (error) {
      return {
        success: false,
        message: "Error: " + error.toString(),
        count: 0,
        books: []
      };
    }
  }
});
```

Furthermore, archiving old records from orders (maybe over 3 months old) to limit the size of the orders collection. Creating a live, in memory, collection of orders of the currently logged in customer can also further improve performance.

Future applications

By far, the 2 most applicable avenues for NoSQL will be in multi-model databases as well as rapid app development.

With the rapid development speed of a NoSQL database, using it to develop apps means a simple process, a database that can be easily modified and therefore a seamless and fast development cycle for prototyping apps.

Another amazing feature is multi-model databases, which can hold rich data types, graphs, images, etc… in a centralized storage environment.

Conclusion

The successful implementation of MongoDB in a bookstore e-commerce application demonstrates its viability in the development cycle due to its simplicity and ease of modification. It, however, would likely not be recommended to be used in a professional deployment as stricter relational databases perform better and avoid the mission critical problems that could arise such as data validation issues.

Bibliography

EnterpriseDB. (n.d.). *Performance Benchmark: PostgreSQL vs. MongoDB.*