Transaction Processing Application for Online Book Store using Distributed Operating Systems and Micro-Services Architectures.

Suraj Sharma, University Of Memphis, sshrmspk@memphis.edu

Abstract. This paper presents a distributed operating system application with microservice architecture for processing book purchases for online book store with the goal to help in transparency, scalability, and agility of the application.

Keywords: Transaction Processing, Distributed Operating Systems, Ecommerce application, Micro-services.

1 Introduction

With increasing number of online shopping activities especially with the current situation with COVID 19, there are many e-commerce bookstore applications that are centralized and are facing problems in efficiency and scalability. These applications besides being centralized, have one bulk program doing everything. So if any part of the application fails, the whole transaction fails, resulting into order rejections and bad user experience. Distributed system with mircroservice architecture intend to overcome these shortcomings. Distributed Operating system provides the transparency where the users using the application are unaware of the presence of distributed resources. System failures can be hidden from the users when one data-center/server is down, another data-center/server takes the task. Similarly microservice architecture makes the application manageable by breaking one big monolithic application into smaller services that can be developed, deployed and scaled independently with middleware to communicate among the services [7].

This paper presents an architecture and implementation of distributed operating system using microservice architecture for online book store transaction processing. This architecture has goal to help in transparency, scalability, and agility of the application. This architecture can be used in any e-commerce application that needs to be always on, transparent, scalable, and agile. We have presented the details of the design and results of the architecture in this paper to help readers understand the benefits of using this architecture over traditional monolithic architecture.

The paper contains related works in the area of distributed systems and microservice architecture followed by our design of the architecture and analysis, and concludes with our accomplishments, future works and appendices.

2 Related Works

In [1] Hasselbring and Steinacker have discussed vertical decomposition into self-contained systems and appropriate granularity of micro services as well as coupling, integration, scalability and monitoring of micro services, and how such framework increased productivity, agility, scalability and reliability in E-commerce at otto.de, one of the biggest European e-commerce platform.

In [2] Palopoli and Rosaci have presented benefits of distributed software architecture for assessing e-commerce, and has recommended a distributed testbed architecture for e-Commerce recommender systems using a multi-tiered agent-based approach to generate effective recommendations without requiring such an onerous amount of computation per single client.

In [7] Gero, Ludger and Peter have presented different advantages of event based architecture in regard to scale, system evolution and real-time requirements, and have put it into context. Active databases, software engineering, stream processing systems, distributed monitoring and debugging systems, sensor networks, and workflow systems are few of the areas the book talks about how event-based architecture have been adopted. The book explains how components of event-based mode communicates by generating and receiving event notifications, where event is an occurrence of happening of interest, and how publish/subscribe middleware mediates notifications from producer/publisher to consumers/subscribers.

Most of these papers either talk about microservice architecture or distributes systems, but haven't presented architecture that includes both of these great architectures. In this paper we present another flavor of these architectures that contains microservice architecture for application layer which are distributed into different physical systems with the goal to achieve the scalability, transparency and agility of this ebook ecommerce application.

3 Design

In order to overcome the problems that traditional monolithic application which are hosted in one system for ecommerce application, we have designed Even-based software architecture with 3-Tier (physical) "system" architecture consisting of client, application server and database server which is distributed on two physical machines with request managed by load balancer.

The requirement of the application was to design an app that takes order request with order information containing book details, payment information, billing information and shipping details, and processes it. If successful, it should send a confirmation email to the user with the order id and other order details that confirms the order is accepted, and send shipment request to the vendor with information about the item to be shipped and the shipping details to process the shipment of the book purchased.

Based on the requirement we have designed the application using microservice architecture distributed in two physical machines (HP and Dell laptops). Both machines are windows OS and runs the exact replication of the application independent of each other. The configuration is Active-Passive High Availability cluster where one machine

(HP) is actively taking request and other (Dell) is a standby/ backup. If the active/primary (HP) server is down/disconnected, the passive machine/server (Dell) is ready to take over to process the requests. Load balancer manages the request between these two servers. The app is broken into 3 micro services; Order Processor, Email Service and Shipping service. All these services are loosely-coupled .We will talk about each in detail later in this section.

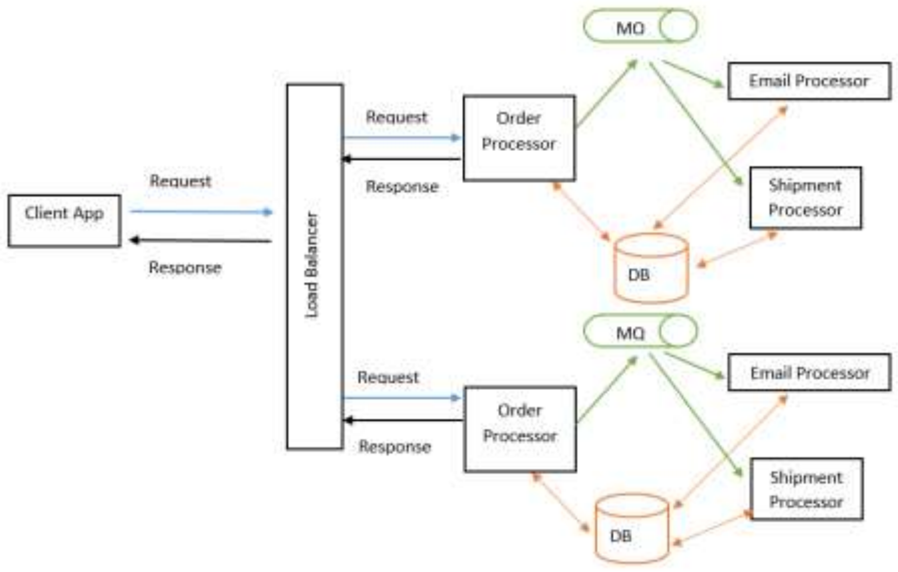
The design of the application is shown in the figure below in this section. Here the Software architecture is an Even-based architecture with Sender Running <-> Receiver running Message-Queuing Model where Order Processor communicates with Email Service and Shipping service through the propagation of events through ActiveMQ as the message broker. This is 3-Tier (physical) “system” architecture where web client acts as an user interface and Order processor as an application server which requests to insert order data to Database server, then waits to get the order id from the database and finally returns the result(success/failed) to the web client.

All three services are developed in JAVA using Spring Boot framework [4]. The message queue is integrated using ActiveMQ [3] and Spring JMS. MySql database is used to store the order details. MySql connector (java) [6] and Spring JPA is used to connect the services to the database and update the data.

You can find the application code in github.

- OrderProcessorService: https://github.com/surajsharmaa/OrderProcessorService
- EmailService: https://github.com/surajsharmaa/EmailService
- ShippingService: https://github.com/surajsharmaa/shippingservice
- LoadBalancer : https://github.com/surajsharmaa/ebookLoadBalancer

Here is the basic architecture of the application that represents the distributed nature of the project:



3.1 Order Processor:

Order Processor receives the order request, processes the order, saves the order information in the database, and publishes the message for other two services, Email Service and Shipping Service. If the request is successful it sends the order id of the transaction back to the load balancer that eventually is sent back to the client. It is also responsible for the database management, and updates database structure as needed.

3.2 Email Service:

Email Service sends the confirmation email to the buyer to verify the order has been received. Once order processor publishes the message to the email queue, email service consumes the message, and sends the order confirmation email to the customer to acknowledge that the order has been received. We are using Gmail SMTP [8] to send the email confirmation from Email Service. You can see an example of the email sent to the customer in fig 2.

3.3 Shipping Service:

Shipping service sends email notification to the vendor with details about the item to be shipped and shipping info to send the items in the order. Once the order processor publishes shipping message to the shipping queue, shipping service picks the message and sends shipping request email to the vendor. Here also we are using Gmail SMTP [8] to send email to the vendor. You can find an example of such email in fig 2.

3.4 Load Balancer:

Load balancer receives the request coming from the web client and routes to one of the two servers based on the availability of these servers. Since our model is Active-Passive High Availability Cluster, load balancer first tries to get the health page of the active (HP) order processor server, and if it is up, forwards the request to it, else sends it to the passive (Dell) order processor server if it is up. If both are down/disconnected or unreachable, load balancer throws Internal Server Error. Load Balancer also houses the web client code which the customer interacts with. You can find the screenshot of the web page that customer can use to purchase the book online in fig 3.

3.5 Database Server:

We are using MySql database as database server. The order processer maintains the database structure and is responsible for updating the data as request comes in. Database contains four tables; *order_table, billing_info_table, shipping_info_table* and *line_item_table*. Order table contains payment details and its primary key is considered to be the order id for the transaction which is dynamically generated. Orders processed by Active (HP) server is saved with prefix “b-“ and orders processed by Passive (Dell) server is saved with prefix “a-“ to help application managers identify which server processed the order. The customer is not aware of this distinction when they receive the order confirmation. Billing_info_table stores billing information of the customer like name, address, phone number and email. Shipping_info_table stores shipping details like receiver’s name, receiver’s address, email and phone number. Line_item_table contains the detail about the book purchased like SKU and quantity. You can fine the ER diagram of the database in Fig 4.

3.6 Message Broker:

We are using ActiveMQ [3] as the message broker which manages the email and shipping queue. Spring JMS is used to talk to different queues by each services. Only order processor is the publisher of the messages, both email service and shipping service are the consumers of the messages on each queue. Here is the ActiveMQ web console that shows the status of each queue, how many consumers are up and how many messages are enqueued and dequeued.

4 Analysis

The goal of this architecture was to help in transparency, scalability, and agility of the application, and has been able to deliver all these goals. In this section we will talk about each of these goals and how our architecture achieved it.

- Transparency:
  Transparency means to hide the fact that its processes and resources are physically distributed across multiple computers/servers to the user using the application. Our architecture just do that.
  - Access Transparency: Access transparency means to hide differences in data representation and how an object is accessed [9]. When user purchases a book, all they see is a confirmation page on their browser and a confirmation email in their inbox, the architecture provides the access transparency by hiding how the order id is generated and how they are receiving the confirmation email notifications.

  - Failure Transparency: Failure Transparency means to hide the failure and recovery of an object [9]. Our architecture provides the failure transparency by hiding the server failure and recovery of the servers by routing the request to other data-center/server if one is down. Also for the user the only service that really needs to be up and running is Order processor, other services are really just the back-end services and the user doesn’t need to wait for them to be available for their order to be processed. This decoupling of the services is truly hidden from the user in our architecture.

    - Location Transparency: Location transparency refers to the fact that user cannot tell where an object is physically located [9]. As described in the design section, in this architecture the application is distributed between Dell and HP server. For a customer accessing the ebook app through online webpage have no idea where the ebook application server is, and where their order information is stored, the only thing they get back is the order id on their confirmation page and confirmation email. Besides the order can be **a-**xxxx or **b-**xxxx, the customer has no idea what those prefix means, how they are created and accessed. These prefix are created for the app managers to know where the order was originated, on Dell server or HP server, but the customer has no clue of these details.

- Scalability:
  Being able to scale in size is one of the biggest goal of this architecture, and it has delivered that goal. As we have mentioned in the design section, the ebook application is distributed on two physical machines, this refers to scaling in size. One can spin off another machine with these 3 services running and let the load balance handle the traffic. We can easily add more resources/physical machines without any noticeable loss of performance. Increasing the number of physical machines provides the bandwidth to take more requests and thus increases the availability of the application.

- Agility:
  Agility is the ability to change, develop and deploy quickly and easily. Agility is most in today's online world, and our architecture provides the flexibility to change quickly. With the application broken down into micro services, they can be developed, updated and deployed independently of each other. This provides the flexibility for the business stake holders to update the requirement through market driven analysis and can be implemented in short span of time without worrying about the code of entire application.

5 Conclusion

With the growing need and trend of online shopping, online book purchase has also been very popular lately. The number of online transaction is growing, and thus a robust framework to support the "always on" application with transparency, openness, scalability and agility is most. Traditionally applications were designed with monolithic architecture where all application logics and data were developed and deployed as one unit, in one physical system which were prone to failures resulting in frustrating user experience. The architecture I am proposing is distributed, and has self-contained micro services which is transparent, scalable, fault tolerant, and agile.

Here we presented Even-based software architecture with 3-Tier (physical) "system" architecture consisting of client, application server and database server which is distributed on two physical machines with request managed by load balancer. Customer interacts with the user interface layer, the web-page, which makes order request to the application server, order processor, via load balancer. Order processor makes request call to the database server to save the order details and waits for the response containing the order id, which is then sent back to the user interface layer via load-balancer for customer to view. As our client maintains only the necessary software components required to render the webpage, our client is thin client. This makes this architecture user friendly as no special hardware/software is required for the customers to have on their machine which they are using to access the app. All the processing is done on the server level.

The application itself is broken down into smaller components (microservices) which talks through event-based architecture. These services/components communicates through the propagation of events/messages where ActiveMQ acts as the message broker/middleware that guarantees the messages are only accessible to the service subscribed to that queue. So when order processor publishes confirmation email message to “mailbox” queue, only Email Service can access that message not the Shipping service. And similarly when Order Processor publishes shipping message to “shipping” queue the message is only accessible by Shipping service not Email service. Besides these messages are preserved until they are successfully processed. This makes our architecture very beneficial for any ecommerce application that has many messages to process without losing the messages. When the Shipping service is down, the messages on the “shipping” queue are preserved until Shipping Service comes back and processes it successfully. If there is any problem in processing the message, they are sent to the backout queue which can later be moved to the respective queue when the problem is solved.

The architecture proposed has been successful in hiding differences in data representation, and how objects are accessed. When user purchases a book, all they see is a confirmation page on their browser and a confirmation email in their inbox, they have no clue how it all worked out. When one of the two server is down, customer is completely unaware of that failure as our architecture hides the failure and recovery of the application server. As we have shown in design section that the application is distributed on two physical machines, once can easily scale it in size with addition of machines/resources. Our microservice architecture provides the flexibility to change, develop and deploy quickly to the market. If there is a bug or enhancement required on specific piece of the application, developer can just work on one of the three services without worrying about the whole application code. Individual service can be deployed independently of each other. This is big for any ecommerce application where requirements change daily, new enhancements are needed regularly and bugs need to be fixed. Hence we can surely say our architecture exceeds our goal to help in transparency, scalability, and agility of the application.

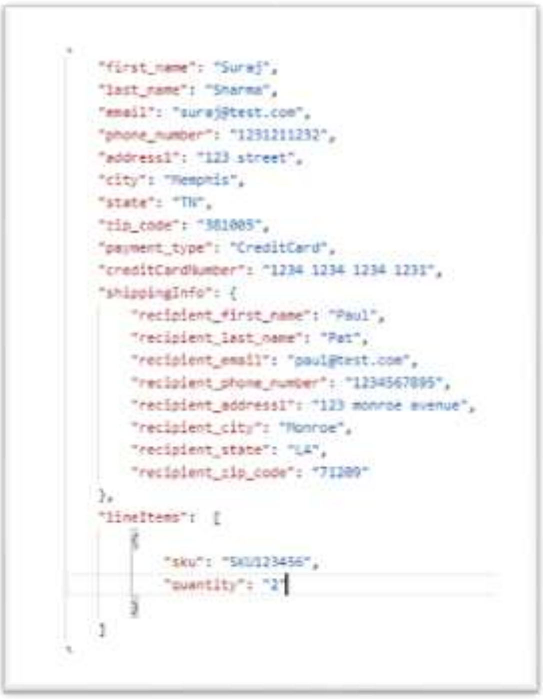
6 Future Work

As we only have limited time for this project, we were not able to add many features to this application that would make this ebook app more usable. In future we can add more micro services to perform other curtail services like warehouse inventory management, support multi product purchase at single checkout (cart funcationality), restock notification and more. Also it would be great to have user friendly web interface with different books for users to purchase. Besides adding new features, in future we would like to implement customized backout queues for each services that will allow us to manage messages in backout queues better.

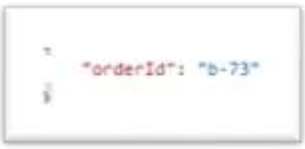
7 Appendices

This section contains different screenshots that may be helpful in gaining comprehensive understanding of our application, its design and architecture.

In this section you will see an example of request payload and response, example confirmation email sent to the customer and example email sent to the shipping vendor, webpage where customer can place an order, ActiveMQ console with “mailbox” and “shipping” queues, and ER diagram of the MySql database of Ebook.

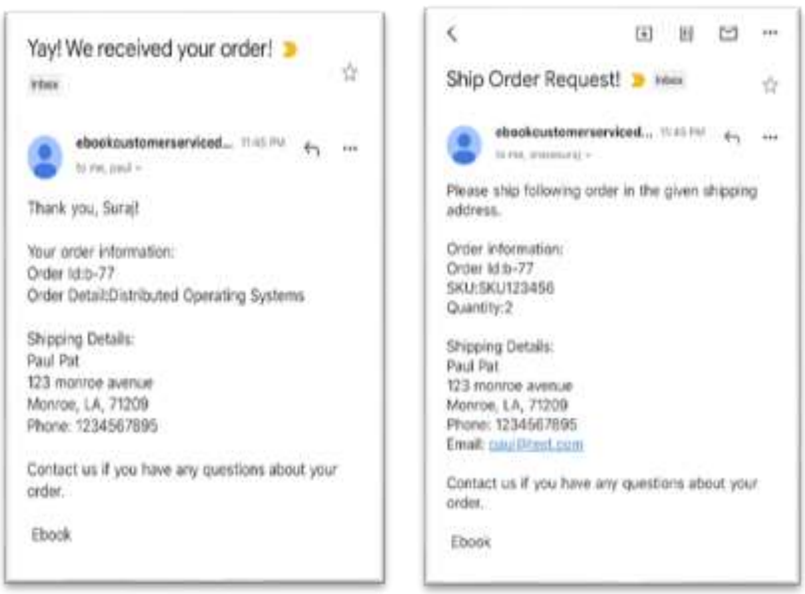


```
{
    "first_name": "Suraj",
    "last_name": "Sharma",
    "email": "suraj@test.com",
    "phone_number": "1231211232",
    "address1": "123 street",
    "city": "Memphis",
    "state": "TN",
    "zip_code": "381005",
    "payment_type": "CreditCard",
    "creditCardNumber": "1234 1234 1234 1231",
    "shippingInfo": {
        "recipient_first_name": "Paul",
        "recipient_last_name": "Pat",
        "recipient_email": "paul@test.com",
        "recipient_phone_number": "1234567895",
        "recipient_address1": "123 monroe avenue",
        "recipient_city": "Monroe",
        "recipient_state": "LA",
        "recipient_zip_code": "71209"
    },
    "lineItems": [
        {
            "sku": "SKU123456",
            "quantity": "2"
        }
    ]
}
```



```
{
    "orderId": "b-73"
}
```

Fig 1: Request payload and response.


Yay! We received your order!

Inbox

ebookcustomerserviced... 11:45 PM

to me, paul

Thank you, Suraj!

Your order information:
Order Id:b-77
Order Detail:Distributed Operating Systems

Shipping Details:
Paul Pat
123 monroe avenue
Monroe, LA, 71209
Phone: 1234567895

Contact us if you have any questions about your order.

Ebook

Ship Order Request!

Inbox

ebookcustomerserviced... 11:45 PM

to me, [illegible]

Please ship following order in the given shipping address.

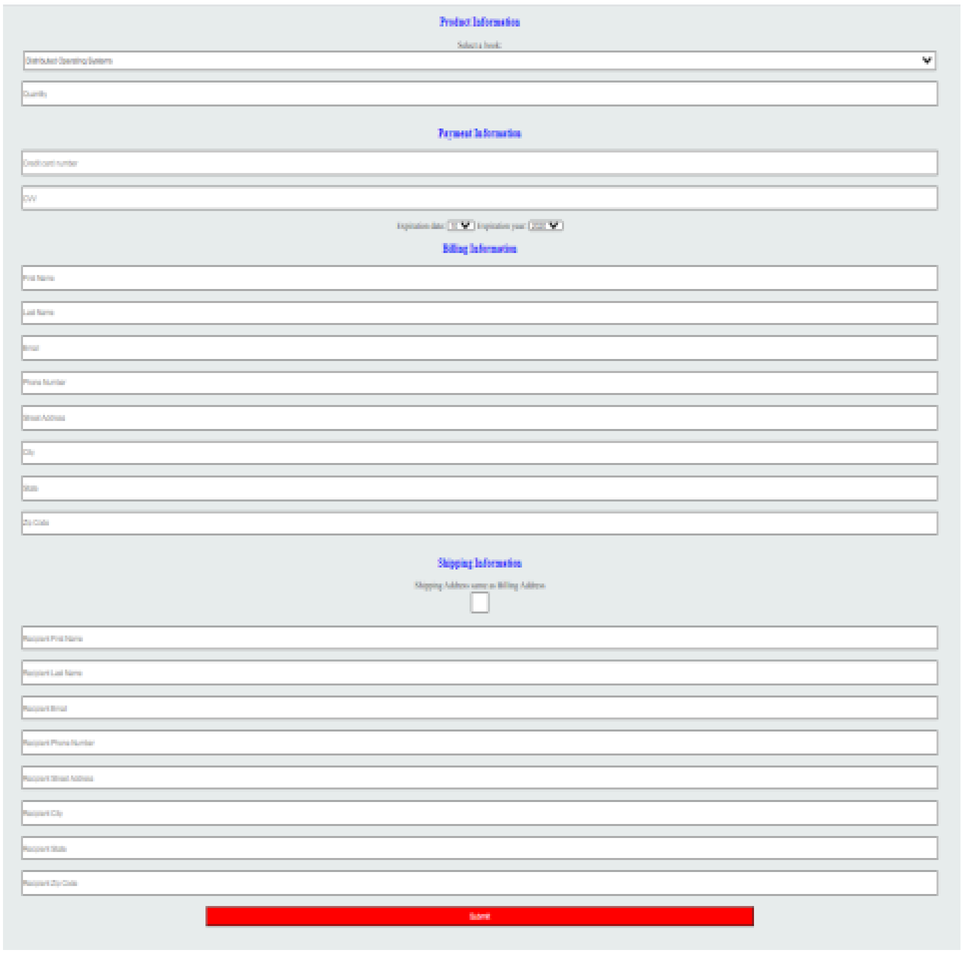
Order information:
Order Id:b-77
SKU:SKU123456
Quantity:2

Shipping Details:
Paul Pat
123 monroe avenue
Monroe, LA, 71209
Phone: 1234567895
Email: paul@test.com

Contact us if you have any questions about your order.

Ebook

Fig 2: Confirmation email sent to customer and Shipping request email sent to the vendor.



Product Information

Select a book:

Distributed Operating Systems

Quantity

Payment Information

Credit card number

CVV

Expiration date: Expiration year:

Billing Information

First Name

Last Name

Email

Phone Number

Street Address

City

State

Zip Code

Shipping Information

Shipping Address same as Billing Address

Recipient First Name

Recipient Last Name

Recipient Email

Recipient Phone Number

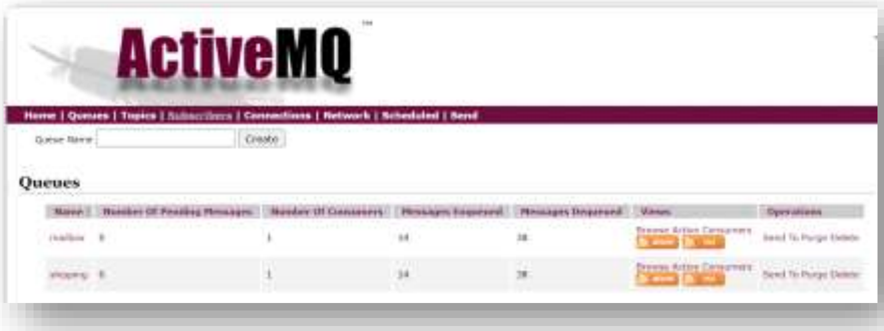
Recipient Street Address

Recipient City

Recipient State

Recipient Zip Code

Fig 3: Web interface for ebook application



ActiveMQ™
Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send
Create

Queue Name

Queues

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
mailbox	0	1	14	28	Browse Active Consumers	Send To Purge Delete
shipping	0	1	14	28	Browse Active Consumers	Send To Purge Delete

Fig 5: ActiveMQ web console.

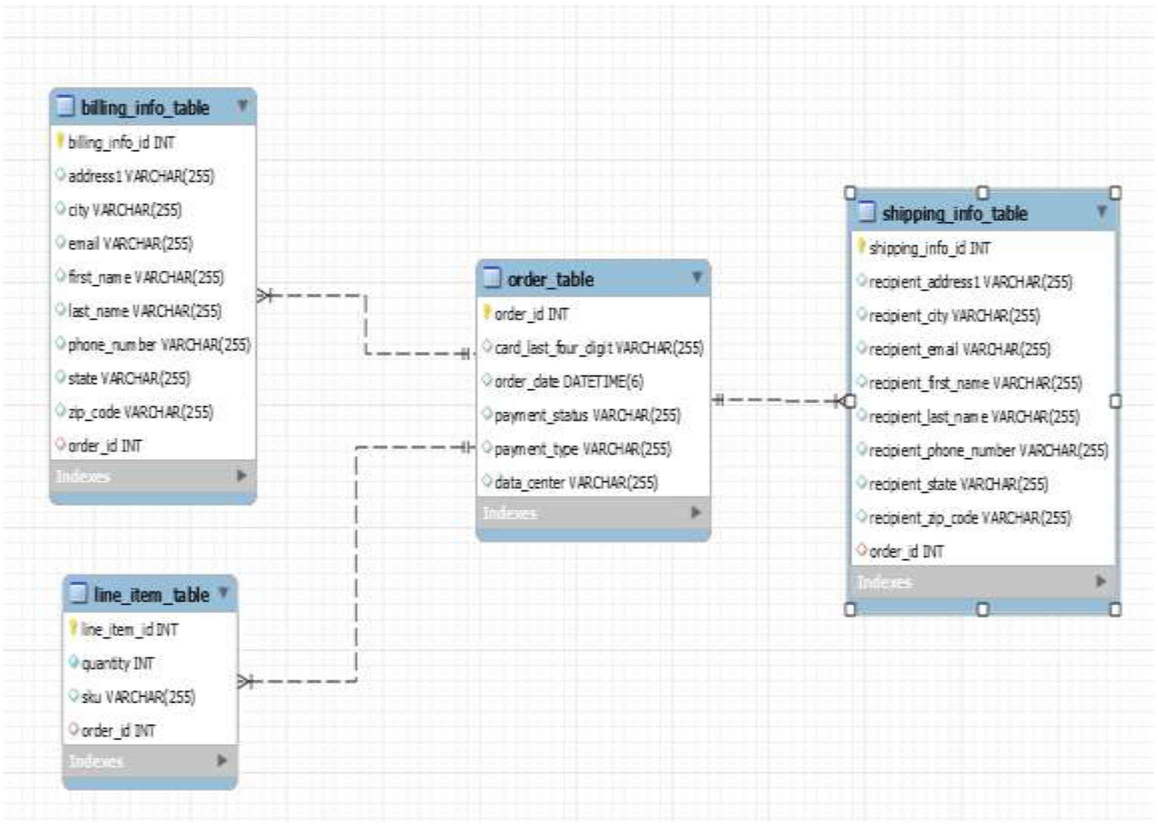


Fig 4: ER Diagram of Ebook database

References

1. W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, 2017, pp. 243-246, doi: 10.1109/ICSAW.2017.11. https://ieeexplore.ieee.org/abstract/document/7958496
2. Palopoli, L., Rosaci, D., and Sarné, G. M. L. (2016) A distributed and multi-tiered software architecture for assessing e-Commerce recommendations. Concurrency Computat.: Pract. Exper., 28: 4507– 4531. doi: 10.1002/cpe.3798. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3798
3. “The Classic Open Source Message Broker” ActiveMQ, 2019, http://activemq.apache.org/components/classic/
4. “Building an Application with Spring Boot” Spring IO, 2020, https://spring.io/guides/gs/spring-boot/

5. “The Collaboration Platform for API Development” POSTMAN, 2020, https://www.postman.com/

6. “Accessing data with MySQL” Spring IO, 2020, https://spring.io/guides/gs/accessing-data-mysql/

7. Gero Muhl, Ludger Fiege, Peter Pietzuch, “Distributed Event-Based Systems” 2006, https://books.google.com/books?id=lddDAAAAQBAJ&lpg=PA1&ots=Hbrk4-7yLC&dq=event%20based%20architecture%20%20scholar%20articles&lr&pg=PA3#v=onepage&q&f=false

8. “Spring Boot – How to send email via SMTP” Mkyong.com, April 2019, https://mkyong.com/spring-boot/spring-boot-how-to-send-email-via-smtp/

9. Maarten van Steen and Andrew S. Tanenbaum, “Distributed Systems” 2017