# Práctica Calificada N°3


## CC4P1 Programación Concurrente y Distribuida

**Colque Unocc Gabriela, Meza Rodríguez Fiorella, Ricapa Corrales Rubén**

*gabriela.colque.u@uni.edu.pe, fiorella.meza.r@uni.edu.pe, ruben.ricapa.c@uni.pe*

*Escuela Profesional de Ciencias de la Computación*

*Facultad de Ciencias*

*Universidad Nacional de Ingeniería*

# Índice

# Abstract

En este trabajo se describe el desarrollo de un sistema distribuido utilizando RabbitMQ como middleware. El sistema se compone de tres nodos, cada uno con un sistema operativo y lenguaje de programación diferentes, el cual simula una venta, gestiona la facturación y el control de inventario.

En primer lugar, se describe brevemente los fundamentos teóricos. Luego, se describe la arquitectura diseñada del sistema distribuido, que se basa en un enfoque cliente-servidor. Además, se presenta un diagrama de protocolo que muestra la interacción entre los componentes.Posteriormente, se implementan y se describen en detalle los códigos desarrollados en cada lenguaje.



Keywords: sistema distribuido, middleware, arquitectura, interración entre componentes

# 1 Introducción

El presente informe tiene como objetivo describir el desarrollo de un sistema distribuido utilizando RabbitMQ como middleware. Este sistema se ha diseñado para gestionar la facturación y el control de inventario, donde se requiere validar la disponibilidad de productos en un almacén antes de realizar la facturación correspondiente.

El sistema se compone de tres nodos, cada uno con un sistema operativo y un lenguaje de programación diferente. El nodo de ventas, basado en Ubuntu con Java, guarda los datos de las facturas generadas en la base de datos de ventas. El nodo del almacén, basado en Kali con Go, almacena la información de los productos disponibles en la base de datos correspondiente. Por último, el nodo del vendedor, basado en Windows con Python, interactúa con el cliente y se encarga del proceso de facturación y mediante el middleware tiene que validar si existe los productos en el almacén para que luego pueda guardar la información de la venta.

RabbitMQ ha sido seleccionado como el middleware para facilitar la comunicación entre los nodos. Este sistema de mensajería proporciona una arquitectura flexible y escalable, permitiendo la comunicación asíncrona entre los componentes del sistema distribuido.

En este informe, se detallará el flujo del sistema y la evaluación del desempeño del mismo.

# 2 Marco Teórico

## 2.1. Sistemas Distribuidos

Los sistemas distribuidos se definen como un conjunto de componentes de hardware y software interconectados, ubicados en diferentes nodos de una red, que colaboran entre sí para lograr un objetivo común. Estos sistemas permiten la distribución de tareas y recursos, brindando ventajas como escalabilidad, redundancia y tolerancia a fallos.

## 2.2. Middleware

El middleware es una capa de software que actúa como intermediario entre las aplicaciones distribuidas, facilitando la comunicación y la integración entre los componentes. En este proyecto, se utiliza RabbitMQ como middleware, el cual implementa un modelo de mensajería basada en colas. Esto permite el envío y recepción asíncrona de mensajes entre los componentes del sistema distribuido.

## 2.3. Lenguajes de Programación

En el desarrollo de sistemas distribuidos, la elección del lenguaje de programación es crucial para garantizar la eficiencia, la compatibilidad y la productividad. En este proyecto, se utilizan tres lenguajes de programación diferentes:

- Java: Es un lenguaje de programación ampliamente utilizado en sistemas distribuidos debido a su portabilidad, escalabilidad y soporte para la programación orientada a objetos.


- GO: Es un lenguaje de programación moderno que se ha vuelto popular en el desarrollo de sistemas distribuidos debido a su eficiencia, concurrencia y facilidad de uso.


- Python: Es un lenguaje de programación versátil y fácil de aprender, que se utiliza ampliamente en el desarrollo de sistemas distribuidos.

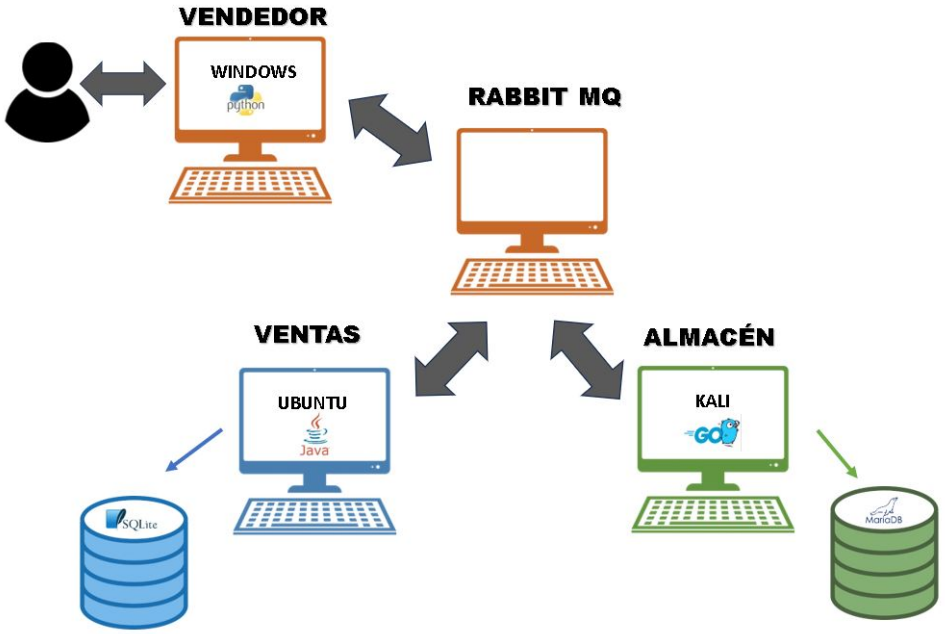

## 2.4. Base de Datos

Las bases de datos juegan un papel fundamental en los sistemas distribuidos para el almacenamiento y gestión de datos. En este proyecto, se utilizarán dos bases de datos diferentes:

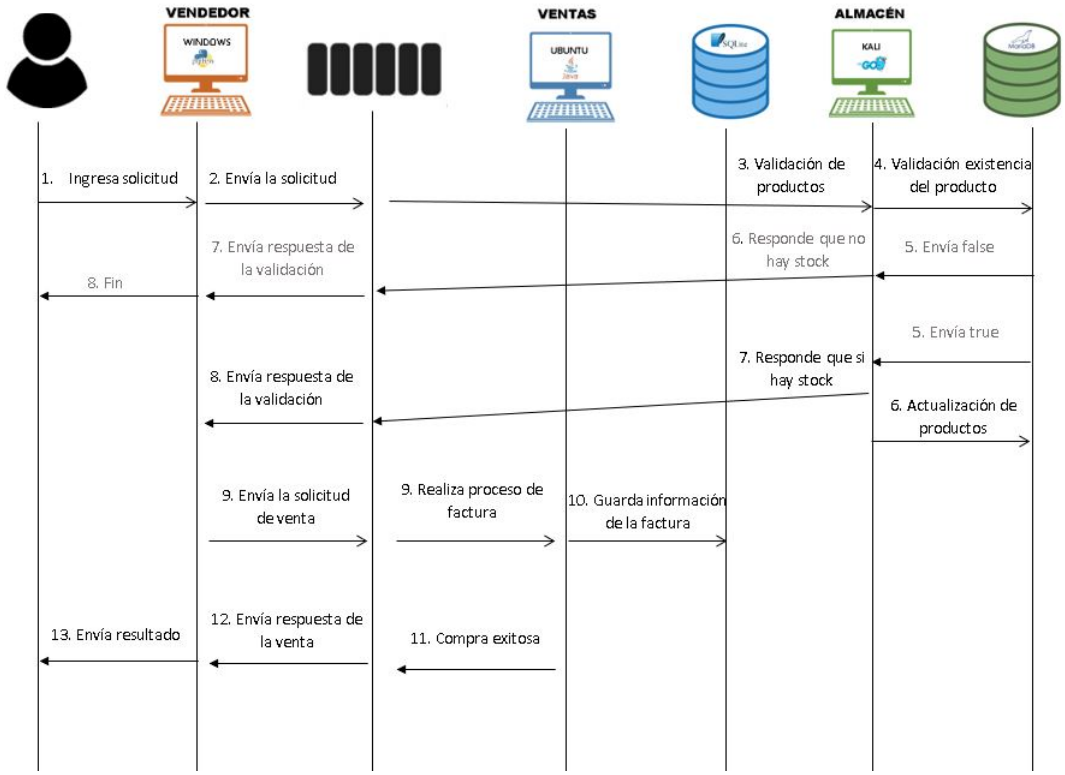
- SQLite: Es una base de datos relacional liviana y de código abierto ampliamente utilizada en sistemas distribuidos. Proporciona una solución local para almacenar las facturas y su detalle.


- MariaDB: Es una base de datos relacional basada en el código fuente de MySQL. Es una alternativa robusta y escalable para sistemas distribuidos que requieren una mayor capacidad de procesamiento y almacenamiento.

# 3 Arquitectura




# 4 Diagrama de protocolo



# 5 Anexo Código

## Python

```
#!/usr/bin/env python
import pika
import threading
import os

def recibir_mensajes():
    credentials = pika.PlainCredentials('admin', 'admin')
    connection = pika.BlockingConnection(pika.ConnectionParameters('
       localhost', 5672, 'ventas', credentials))
    channel = connection.channel()

    channel.queue_declare(queue='confirmacion')

    def callback(ch, method, properties, body):
        #print(" [x] Received %r" % body.decode())

        if(body.decode() == "0"):
            print("CANTIDAD NO DISPONIBLE")

        elif(body.decode() == "2"):
            print("COMPRA EXITOSA!")

        else:
            subcadenas = body.decode().split(",")
            channel.basic_publish(exchange='', routing_key='recibo',
               body=nombre+","+correo+","+subcadenas[0]+","+subcadenas
               [1]+ "," + subcadenas[2])


    channel.basic_consume(queue='confirmacion', on_message_callback=
       callback, auto_ack=True)

    #print(' [*] Waiting for messages. To exit press CTRL+C')
    channel.start_consuming()

def enviar_mensaje():
    credentials = pika.PlainCredentials('admin', 'admin')
    connection = pika.BlockingConnection(pika.ConnectionParameters('
       localhost', 5672, 'ventas', credentials))
    channel = connection.channel()
    channel.queue_declare(queue='Cliente')

    while True:

        id = input("Ingrese el numero de un producto\nProductos: \n1.
           Manzanas\n2. Uva\n3. Platano\n")
        cantidad = input("Ingrese la cantidad: \n")
        mensaje = id + "," +cantidad
        channel.basic_publish(exchange='', routing_key='consulta', body=
           mensaje) # 1,2 1,3
        #connection.close()
        #os.system('cls')
```

```
if __name__ == '__main__':

    nombre = input("Escriba su nombre: ")
    correo = input("Escriba su correo")
    while True:
        # Crear y ejecutar hilos para enviar y recibir mensajes
            simultaneamente
        enviar_hilo = threading.Thread(target=enviar_mensaje)
        recibir_hilo = threading.Thread(target=recibir_mensajes)

        enviar_hilo.start()
        recibir_hilo.start()

        # Esperar a que ambos hilos finalicen
        enviar_hilo.join()
        recibir_hilo.join()
```

- **Java**

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class MqttExample2 {

    private static final String RECEIVE_QUEUE_NAME = "recibo";
    private static final String SEND_QUEUE_NAME = "confirmacion";
    public static Database db = new Database();

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("192.168.1.6");
        factory.setUsername("admin");
        factory.setPassword("admin");
        factory.setVirtualHost("ventas");

        com.rabbitmq.client.Connection connection = factory.
            newConnection();
        Channel receiveChannel = connection.createChannel();
        Channel sendChannel = connection.createChannel();

        receiveChannel.queueDeclare(RECEIVE_QUEUE_NAME, false, false,
            false, null);
        sendChannel.queueDeclare(SEND_QUEUE_NAME, false, false, false,
            null);

        System.out.println(" [*] Waiting for messages. To exit press
            CTRL+C");
```

```
        // Configurar el callback para recibir mensajes
        DeliverCallback receiveCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(),
               StandardCharsets.UTF_8);
            System.out.println(" [x] Received '" + message + "'");
            String[] data = message.split(",");
            if (data.length == 5) {
                String nombre = data[0];
                String correo = data[1];
    String id  = data[2];
    String cantidad = data[3];
    String precio = data[4];
                db.insertData(nombre, correo, id, cantidad,Integer.
                   parseInt(cantidad) * Integer.parseInt(precio) );
    String tempmessage = "2";
    System.out.println(nombre + " con correo " + correo + " compro "  +
       cantidad + " "+ id + " pagando " + Integer.parseInt(cantidad) *
       Integer.parseInt(precio));
    sendChannel.basicPublish("", SEND_QUEUE_NAME, null, tempmessage.
       getBytes(StandardCharsets.UTF_8));
            } else {
                System.out.println("Invalid message format: " + message)
                   ;
            }
        };
        receiveChannel.basicConsume(RECEIVE_QUEUE_NAME, true,
           receiveCallback, consumerTag -> {});

        // Configurar el hilo para leer la entrada de la terminal y
           enviarla
        Thread sendThread = new Thread(() -> {
            try {
                BufferedReader reader = new BufferedReader(new
                   InputStreamReader(System.in));
                while (true) {
                    String input = reader.readLine();
                    if (input.equalsIgnoreCase("exit")) {
                        break;
                    }
                    sendChannel.basicPublish("", SEND_QUEUE_NAME, null,
                       input.getBytes(StandardCharsets.UTF_8));
                    System.out.println(" [x] Sent '" + input + "'");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        sendThread.start();

        // Esperar a que el hilo de envio termine antes de cerrar la
           conexion
        sendThread.join();

        // Cerrar la conexion y los canales
        sendChannel.close();
        receiveChannel.close();
        connection.close();
    }
}
```

```
class Database {

    public void insertData(String nombre, String correo, String producto
        , String cantidad, int dinero) {

        Connection connection = null;
        PreparedStatement statement = null;

        try {
            // Registrar el driver JDBC
            Class.forName("org.sqlite.JDBC");

            // Establecer la conexion con la base de datos
            String url = "jdbc:sqlite:/home/ubuntu/rabbit/facutaras.db";
            connection = DriverManager.getConnection(url);

            // Crear un objeto PreparedStatement
            String query = "INSERT INTO facturas(name, correo, producto,
                cantidad, dinero) VALUES (?, ?, ?, ?, ?)";
            statement = connection.prepareStatement(query);
            statement.setString(1, nombre);
            statement.setString(2, correo);
          statement.setString(3, producto);
          statement.setString(4, cantidad);
            statement.setInt(5, dinero);

            // Ejecutar la consulta de insercion
            int rowsAffected = statement.executeUpdate();

            System.out.println("Filas afectadas: " + rowsAffected);

        } catch (ClassNotFoundException e) {
            e.printStackTrace();

        } catch (SQLException e) {
            e.printStackTrace();

        } finally {
            // Cerrar la conexion y el statement
            try {

                if (statement != null) {
                    statement.close();
                }

                if (connection != null) {
                    connection.close();
                }

            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- **GO**

```
package main

import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
        "log"
        amqp "github.com/rabbitmq/amqp091-go"
        "strconv"
        "strings"
        "context"
        "time"
)

func failOnError(err error, msg string) {

        if err != nil {
                log.Panicf("%s: %s", msg, err)
        }
}

func main() {

        conn, err := amqp.Dial("amqp://admin:admin@192.168.1.6:5672/
           ventas")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()

        q, err := ch.QueueDeclare(
                "consulta", // name
                false,   // durable
                false,   // delete when unused
                false,   // exclusive
                false,   // no-wait
                nil,     // arguments
        )

        failOnError(err, "Failed to declare a queue")

        msgs, err := ch.Consume(
                q.Name, // queue
                "",     // consumer
                true,   // auto-ack
                false,  // exclusive
                false,  // no-local
                false,  // no-wait
                nil,    // args
        )

        failOnError(err, "Failed to register a consumer")

        var forever chan struct{}
        var data string
```

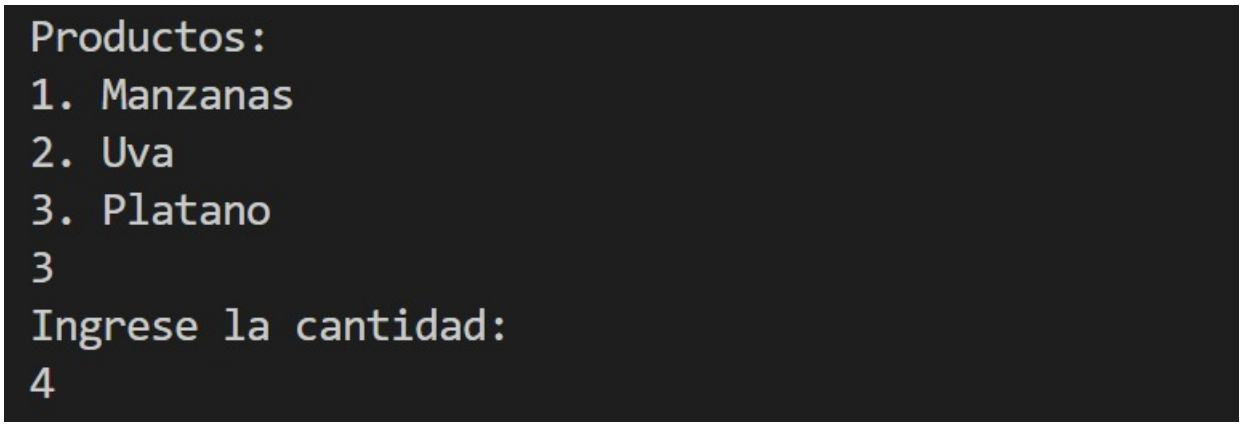
```
        go func() {
                for d := range msgs {
                        log.Printf("Received a message: %s", d.Body)
                        data = string(d.Body)
                        values := strings.Split(data,",")
                        var1,err := strconv.Atoi(values[0])
                        failOnError(err, "FALLO EN LA VARIABLE 1")
                        var2,err := strconv.Atoi(values[1])
                        failOnError(err, "FALLO EN LA VARIABLE 2")
                        //fmt.Printf("%d %d",var1,var2)
                        Purchase(var1,var2)
                }
        }()

        log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
        <-forever
}


func devolverMensaje(mensaje string) {

        //conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
        conn, err := amqp.Dial("amqp://admin:admin@192.168.1.6:5672/
           ventas")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()

        q, err := ch.QueueDeclare(
                "confirmacion", // name
                false,   // durable
                false,   // delete when unused
                false,   // exclusive
                false,   // no-wait
                nil,     // arguments
        )

        failOnError(err, "Failed to declare a queue")
        ctx, cancel := context.WithTimeout(context.Background(), 5*time.
           Second)
        defer cancel()

        body := mensaje
        err = ch.PublishWithContext(ctx,
                "",     // exchange
                q.Name, // routing key
                false,  // mandatory
                false,  // immediate
                amqp.Publishing{
                        ContentType: "text/plain",
                        Body:        []byte(body),
                })
        failOnError(err, "Failed to publish a message")
        //log.Printf(" ENVIANDO RESPUESTA\n", body)
}
```

```
func Purchase(id int, quantity int) {
        var enough bool
        enough,err := canPurchase(id,quantity)
        if err != nil {
                fmt.Println(err)
                return
        }
        if enough {
                db, err := sql.Open("mysql", "root:toor@tcp
                   (127.0.0.1:3306)/Almacen")
                if err != nil {
                fmt.Println(err)
                        return
                }
                _, err2 := db.Exec("UPDATE productos SET stock = stock -
                    ? WHERE id = ?", quantity, id)
                if err2 != nil {
                        fmt.Errorf("canPurchase %d: %v", id, err2)
                }
                fmt.Println("Enough:",enough)
                query := "SELECT price FROM productos WHERE id = ?"
                var precioProducto string
                err = db.QueryRow(query, id).Scan(&precioProducto)
                if err != nil {
                        panic(err.Error())
                }
                query2 := "SELECT name FROM productos WHERE id = ?"
                var nombreProducto string
                err = db.QueryRow(query2,id).Scan(&nombreProducto)
                if err !=nil{
                        panic(err.Error())
                }
                fmt.Printf(nombreProducto)
                devolverMensaje(nombreProducto + "," + strconv.Itoa(
                   quantity) + "," + precioProducto)
        } else {
                fmt.Println("OE SE ACABO")
                devolverMensaje("0")    // RABBIT MQTT
        }

}

func canPurchase(id int, quantity int) (bool, error) {
    var enough bool
    db, err := sql.Open("mysql", "root:toor@tcp(127.0.0.1:3306)/Almacen"
       )
        if err != nil {
        panic(err.Error())
    }
        // Query for a value based on a single row.
    if err := db.QueryRow("SELECT (stock >= ?) from productos where id =
        ?",
        quantity, id).Scan(&enough); err != nil {
        if err == sql.ErrNoRows {
            return false, fmt.Errorf("canPurchase %d: unknown album", id
                )
        }
        return false, fmt.Errorf("canPurchase %d: %v", id,err)
    }
    return enough, nil
}
```

# 6 Anexo Resultados

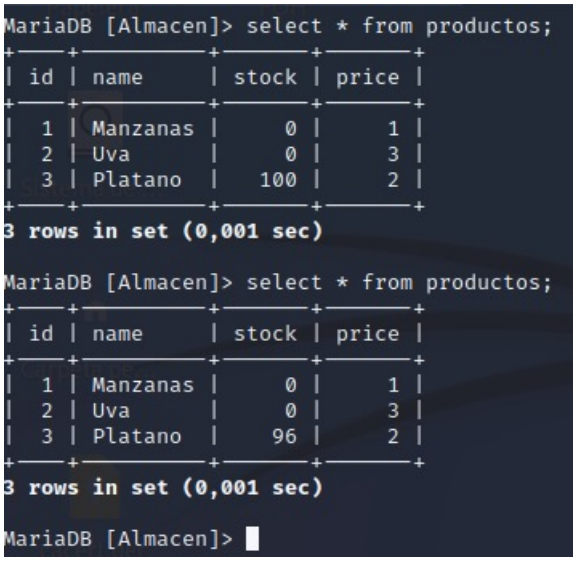
- Inicio del sistema, se ingresa el nombre, correo y se elege el producto a comprar.


- Después de seleccionar el producto, se ingresa la cantidad que se desea comprar.



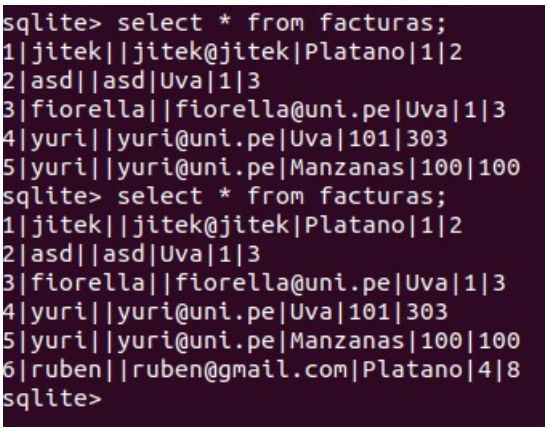

- El producto escogido se envía a validar en el almacén, si hay stock del producto devuelve **true** , en caso contrario devuleve **false**.

- Se actualiza la BD del almacén, se puede observar en la parte superior la cantidad de productos antes de la compra y en la parte inferior el después de la compra.




- Se envía la confirmación de la compra, detallando el nombre del cliente, correo, producto, cantidad y precio a pagar.


- En la BD de ventas se agrega y guarda la factura emitida, se puede observar el registro de las facturas generadas en cada compra.



# 7 Anexo Documentación

[1] https://www.redhat.com/es/topics/middleware/what-is-middleware

[1] https://sg.com.mx/revista/17/sqlite-la-base-datos-embebida

[2] https://mariadb.org/es/

[3] https://www.kali.org/

[4] http://www.deeplearningbook.org/

[5] https://ubuntu.com/