UNIVERSIDAD DIEGO PORTALES

FACULTAD DE INGENIERÍA Y CIENCIAS
ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Tarea 2, Kafka

Profesor: Nicolás Hidalgo

Integrantes:
Sergio Lagos Vergara
Juan Vercher Martínez

Santiago, Chile **Fecha 27/10/2022**

Índice

1. Problemática

El gremio de sopaipilleros de Chile, encargado de establecer políticas legales para la venta en carritos de sopaipillas, ha crecido a un ritmo agigantado. Los anticuados métodos de trabajo para dar soporte a las tareas del gremio requiere de una actualización que implica la utilización de plataformas informáticas capaces de gestionar dichos procesos de la manera más eficiente y escalable.

La gestión de los procesos internos del gremio es compleja. Los miembros que participan en este reciben de manera constante peticiones (escritas a mano) con tareas que deben realizarse. Por ejemplo, realizar la inscripción de un nuevo miembro. Estas tareas tradicionalmente son repartidas según su tipo a los diferentes encargados de gestionar y llevar a cabo las mismas. Uno de estos procesos nombrados, corresponde a la inclusión de nuevos miembros. Este proceso es engorroso y tardío; requiere dejar una petición formal en el gremio, el cual puede ser resulto en cuestión de meses. Esta petición viaja a la dirección encargar de procesar y evaluar nuevos miembros, los cuales se fijan en los antecedentes de los dueños de los carritos postulantes, para así aprobar una lista de nuevos miembros. Esta lista es mostrada de manera periódica, una vez al mes. Este proceso se puede acelerar pagando una comisión, llamado Inscripción Premium. Lo que se busca es automatizar este proceso utilizando sistemas informáticos.

Por otro lado, los maestros sopaipilleros, poseen carritos modernos (es una de las condiciones para ser parte del gremio). Estos carritos poseen sistemas inteligentes con internet y GPS. Además, poseen un sistema capaz de registrar ventas y posteriormente ejecutar código, sin embargo, el gremio no puede aprovechar esta ventaja. Se busca poder aprovechar estos sistemas, utilizando sistemas informáticos automatizados que permitan:

- Calcular estadísticas sobre ventas (cantidad de ventas por día y clientes frecuentes).
- Tener la posición geográfica en tiempo real de cada sopaipillero. Además se busca alertar eventos extraños.
- Preparar la reposición del stock de manera automática, al momento de tener menos de 20 de masas de sopaipillas.

2. Solución

Para resolver esta problemática, se nos pidió trabajar con Kafka y nos entregaron una sugerencia de arquitectura. Es con ella que nos basamos para llegar al siguiente diagrama que presenta el cómo solucionamos el problema del gremio de sopaipilleros.

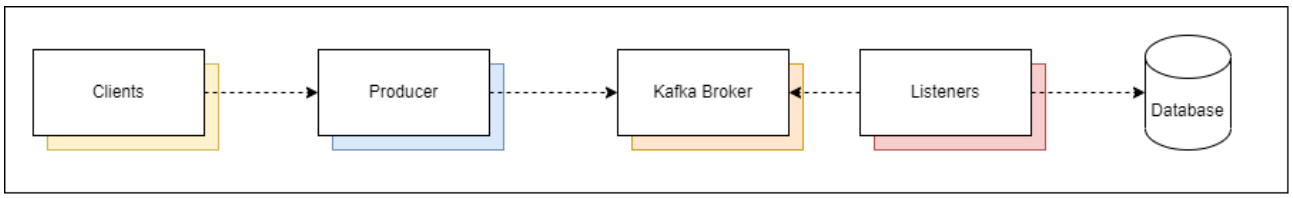

Figura 1: Diseño simplificado de solución

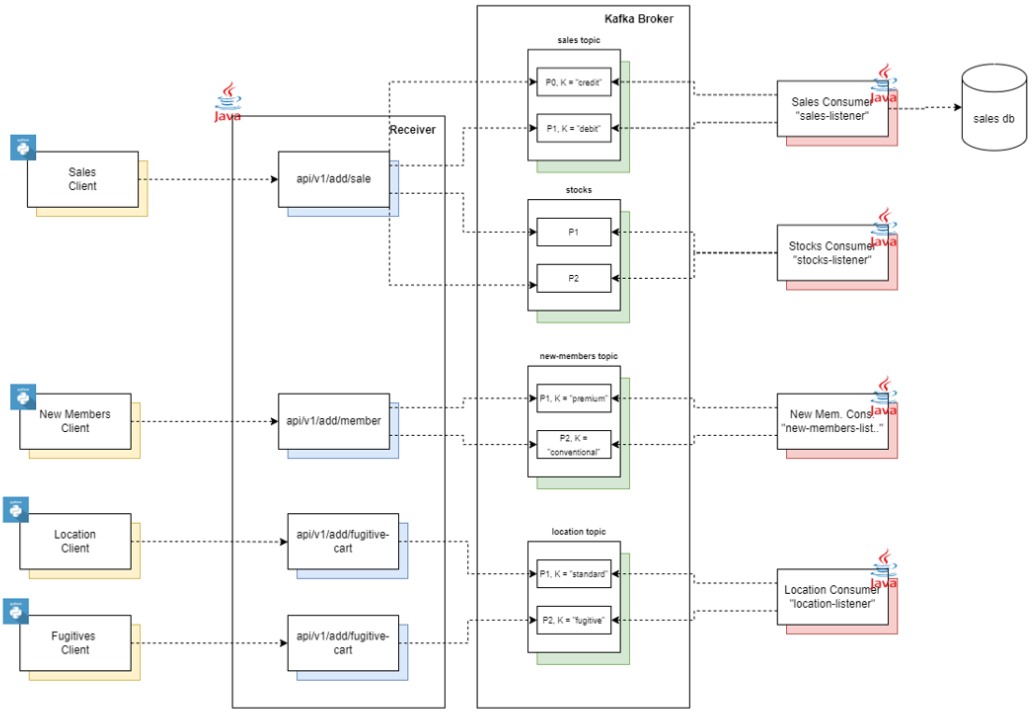

Figura 2: Diseño de solución

Esta solucion se realizó con el framework de Java, Spring.

3. Descripcion de los componentes

3.1. Receiver

Junto con el Broker de Kafka, este componente es el corazón de nuestra solución. Aquí, se levantó una aplicación con **4 endpoints** que permite hacer request que posteriormente se publican a diferentes tópicos y particiones. A grandes razgos, este servicio son 4 producers.

```
sergiolv
@PostMapping(value="/sale")
public ResponseEntity<DefaultResponse> sales(@RequestBody @Valid SaleRequest request) {
    Sale sale = modelMapper.map(request, Sale.class);
    salesService.send(sale);
    return ResponseEntity.status(HttpStatus.ACCEPTED).body(new DefaultResponse( message: "New Sale added!", Calendar.getInstance().getTimeInMillis()));
}

sergiolv
@PostMapping(value="/member")
public ResponseEntity<DefaultResponse> members(@RequestBody @Valid MemberRequest request){
    Member member = modelMapper.map(request, Member.class);
    membersService.send(member);
    return ResponseEntity.status(HttpStatus.ACCEPTED).body(new DefaultResponse( message: "New Member added!", Calendar.getInstance().getTimeInMillis()));
}

sergiolv
@PostMapping(value="/location")
public ResponseEntity<DefaultResponse> location(@RequestBody @Valid CartLocationRequest request){
    CartLocation cartLocation = modelMapper.map(request, CartLocation.class);
    cartLocationService.send(cartLocation);
    return ResponseEntity.status(HttpStatus.ACCEPTED).body(new DefaultResponse( message: "New location updated!", Calendar.getInstance().getTimeInMillis()));
}

sergiolv
@PostMapping(value="/fugitive-cart")
public ResponseEntity<DefaultResponse> fugitiveCart(@RequestBody @Valid FugitiveCartRequest request){
    FugitiveCart fugitiveCart = modelMapper.map(request, FugitiveCart.class);
    fugitivesCartsService.send(fugitiveCart);
    return ResponseEntity.status(HttpStatus.ACCEPTED).body(new DefaultResponse( message: "New Fugitive Cart added", Calendar.getInstance().getTimeInMillis()));
}
```

Figura 3: Endpoints receiver

3.1.1. Creación de tópicos y particiones

```
@Configuration
public class KafkaTopicConfig {
    sergiolv
    @Bean
    public NewTopic salesTopic() { return TopicBuilder.name(Constants.KAFKA_SALES_TOPIC).partitions( partitionCount: 2).build(); }

    sergiolv
    @Bean
    public NewTopic stocksTopic() { return TopicBuilder.name(Constants.KAFKA_STOCKS_TOPIC).partitions( partitionCount: 2).build(); }

    sergiolv
    @Bean
    public NewTopic locationTopic() { return TopicBuilder.name(Constants.KAFKA_NEW_MEMBERS_TOPIC).partitions( partitionCount: 2).build(); }

    sergiolv
    @Bean
    public NewTopic newMembersTopic() {
        return TopicBuilder.name(Constants.KAFKA_LOCATION_TOPIC).partitions( partitionCount: 2).build();
    }
```

Figura 4: Tópicos

3.1.2. SaleRequest

```java
public class SaleRequest {
    2 usages
    @NotBlank(message = "Cart  id is mandatory")
    private String cartId;
    2 usages
    @NotBlank(message = "Client is Mandatory")
    private String clientName;
    2 usages
    @NotNull(message = "Amount is Mandatory")
    private int amount;
    2 usages
    @NotBlank(message = "Date is Mandatory")
    private String date;
    2 usages
    @NotNull(message = "Remaining Stock is Mandatory")
    private int remainingStock;

    2 usages
    @NotNull(message = "Latitude is mandatory")
    private Double latitude;

    2 usages
    @NotNull(message = "Longitude is mandatory")
    private Double longitude;

    2 usages
    @NotNull(message = "Sale Type is mandatory")
    @Pattern(regexp = "credit|debit", flags = Pattern.Flag.CASE_INSENSITIVE)
    private String saleType;
```

Figura 5: Sale Request Dto

3.1.3. MemberRequest

```java
public class MemberRequest {
    2 usages
    @NotBlank(message = "Name is Mandatory")
    private String memberName;
    2 usages
    @NotBlank(message = "Last Name is Mandatory")
    private String memberLastName;
    2 usages
    @NotNull(message = "Id is Mandatory")
    private Long id;
    2 usages
    @Email
    @NotBlank(message = "Email is Mandatory")
    private String email;
    2 usages
    @NotBlank(message = "Plate is Mandatory")
    private String cartPlate;
    2 usages
    @NotNull(message = "Member type is Mandatory")
    @Pattern(regexp = "premium|conventional", flags = Pattern.Flag.CASE_INSENSITIVE)
    private String memberType;
```

Figura 6: Member Request Dto

3.1.4. CartLocationRequest

```java
public class CartLocationRequest {
    2 usages
    @NotNull(message = "cart id is Mandatory")
    private int cartId;
    2 usages
    @NotNull(message = "latitude is Mandatory")
    private Double latitude;

    2 usages
    @NotNull(message = "Longitude is Mandatory")
    private Double longitude;
```

Figura 7: Cart Location Request Dto

3.1.5. FugitiveCartRequest

```java
public class FugitiveCartRequest {

    2 usages
    @NotNull(message = "cart id is Mandatory")
    private int cartId;
    2 usages
    @NotNull(message = "latitude is Mandatory")
    private Double latitude;

    2 usages
    @NotNull(message = "Longitude is Mandatory")
    private Double longitude;
```

Figura 8: Fugitive Cart Request Dto

3.1.6. Sales service

Encargado de tomar la request de ventas y publicar en el topic 'sales' y 'stocks'. Si la compra es de tipo crédito, se publica en la particion 0 y si es débito, en el 1.

```java
@Transactional
public void send(Sale sale)  {
    sendToSalesTopic(sale);
    sendToStockTopic(sale);
}

1 usage  sergiolv
private void sendToSalesTopic(Sale sale){
    try{
        int partition = sale.getSaleType().equals("credit") ? 0 : 1;
        salesKafkaTemplate.send(Constants.KAFKA_SALES_TOPIC, partition, sale.getSaleType(), sale);
        LOGGER.info("New " + sale.getSaleType() + " sale added!. " + sale.getClientName());
    } catch(Exception e){
        throw new KafkaException(Constants.PROBLEM_SALES_TOPIC, e);
    }
}

1 usage  sergiolv
private void sendToStockTopic(Sale sale){
    try{
        CartStock cartStock = new CartStock(sale.getCartId(), sale.getRemainingStock());
        cartStockKafkaTemplate.send(Constants.KAFKA_STOCKS_TOPIC, cartStock);
        LOGGER.info("Stock of cart " + String.valueOf(cartStock.getCartId()) + " updated. Remaining: " + String.valueOf(cartStock.getRemainingStock()) );
    } catch(Exception e){
        throw new KafkaException(Constants.PROBLEM_SALES_TOPIC, e);
    }
}
```

Figura 9: Código sales service

3.1.7. Members service

Encargado de tomar las peticiones de nuevos miembros y publicar en el tópico 'new-members'. Si el miembro es premium se publica en la partición 0 y 1 si es 'conventional'.

```java
@Transactional
public void send(Member member) {
    sendToTopic(member);
}

1 usage  sergiolv
private void sendToTopic(Member member) {
    try{
        int partition = member.getMemberType().equals("premium") ? 0 : 1;
        memberKafkaTemplate.send(Constants.KAFKA_NEW_MEMBERS_TOPIC, partition, member.getMemberType(), member);
        LOGGER.info("New " + member.getMemberType()  + " member added!. " + member.getEmail());
    } catch(Exception e){
        throw new KafkaException(Constants.PROBLEM_NEW_MEMBERS_TOPIC, e);
    }
}
```

Figura 10: Código members service

3.1.8. Cart Location service

Encargado de tomar las peticiones de ubicacion de carros. Este service escribe en la partición 0 del tópico 'location'.

```java
@Transactional
public void send(CartLocation cartLocation) {
    sendToTopic(cartLocation);
}

1 usage  sergiolv
private void sendToTopic(CartLocation cartLocation) {
    try{
        cartLocationKafkaTemplate.send(Constants.KAFKA_LOCATION_TOPIC,  partition: 0,  key: "standard", cartLocation);
        LOGGER.info("New Standard cart location updated! " + cartLocation.getCartId() + " " + cartLocation.getLatitude() + " " + cartLocation.getLongitude());
    } catch(Exception e){
        throw new KafkaException(Constants.PROBLEM_LOCATION_TOPIC, e);
    }
}
```

Figura 11: Código cart location service

3.1.9. Fugitive Cart service

Encargado de tomar las peticiones de ubicacion de carros fugitivos. Este service escribe en la partición 1 del tópico 'location'.

```java
@Transactional
public void send(FugitiveCart fugitiveCart) {
    sendToTopic(fugitiveCart);
}

1 usage  sergiolv
private void sendToTopic(FugitiveCart fugitiveCart){
    try{
        fugitiveCartKafkaTemplate.send(Constants.KAFKA_LOCATION_TOPIC, partition: 1, key: "fugitive", fugitiveCart);
        LOGGER.info("New fugitive cart!. " + fugitiveCart.getCartId());
    } catch(Exception e){
        throw new KafkaException(Constants.PROBLEM_LOCATION_TOPIC, e);
    }
}
```

Figura 12: Código fugitive cart service

3.2. Clientes

Se crearon **4 clientes** en python, que permiten hacer una request a cada endpoint de forma continua con los Dto's respectivos.

3.3. Broker Kafka

Para el Broker, solo se instanció, ya que gracias al framework, se crean las particiones y tópicos al levantar el **Receiver.**

3.4. Consumers

3.5. Sales consumer

Este consumer, contiene 2 listener con groupId = "sales-listener". Uno lee de la partición de débito y otro del de crédito. Luego los guarda en la base de datos con **salesService.save(sale)**. El esquema de la base de datos se puede ver en la figura 14.

```
sergiolv
@KafkaListener(groupId = Constants.COUNSUMER_GROUP_ID,  topicPartitions = @TopicPartition(topic = Constants.KAFKA_SALES_TOPIC, partitions = {"1"}))
void firstListener(String JsonSale) throws JsonProcessingException, DatabaseException {
    ObjectMapper objectMapper = new ObjectMapper();
    Sale sale = objectMapper.readValue(JsonSale, Sale.class);
    LOGGER.info("Partition 1, " + sale.getSaleType());
    salesService.save(sale);
}

sergiolv
@KafkaListener(groupId = Constants.COUNSUMER_GROUP_ID,  topicPartitions = @TopicPartition(topic = Constants.KAFKA_SALES_TOPIC, partitions = {"0"}))
void secondListener(String JsonSale) throws JsonProcessingException, DatabaseException {
    ObjectMapper objectMapper = new ObjectMapper();
    Sale sale = objectMapper.readValue(JsonSale, Sale.class);
    LOGGER.info("Partition 0, " + sale.getSaleType());
    salesService.save(sale);
}
```

Figura 13: Código sales consumer

```
CREATE TABLE IF NOT EXISTS sales (
    id serial NOT NULL,
    cart_id INT NOT NULL,
    client_name VARCHAR(80) NOT NULL,
    sale_type VARCHAR(6) NOT NULL,
    amount INT NOT NULL,
    sale_date VARCHAR(30) NOT NULL,
    remaining_stock INT NOT NULL,
    latitude float8 NOT NULL,
    longitude float8 NOT NULL,
    PRIMARY KEY (id)
)
```

Figura 14: Esquema sales db

Persistir los datos nos sirve para poder ejecutar el cron todos los días y ver las estadísticas necesarias.

3.5.1. Cron

Este cron, permite ver las estadísticas solicitadas con las siguientes consultas y de la siguiente forma:

```
sergiolv
@Scheduled(cron = "*/10 * * * * *")
public void getDailySales() throws JsonProcessingException {
    List<Object> dailySales = saleJpaRepository.getDailySales();
    LOGGER.info("======================");
    LOGGER.info("      DAILY SALES");
    LOGGER.info("======================");
    for(Object o : dailySales){
        ObjectMapper od = new ObjectMapper();
        String json = od.writeValueAsString(o);
        LOGGER.info("cartId, sales = " + json);
    }
    List<Object> dailyCustomers = saleJpaRepository.getDailyCustomers();
    LOGGER.info("======================");
    LOGGER.info("      DAILY CUSTOMERS");
    LOGGER.info("======================");
    for(Object o : dailyCustomers){
        ObjectMapper od = new ObjectMapper();
        String json = od.writeValueAsString(o);
        LOGGER.info("cartId, customers = " + json);
    }
    List<Object> avgAmount = saleJpaRepository.getAvgAmount();
    LOGGER.info("======================");
    LOGGER.info("    DAILY AVG SALES");
    LOGGER.info("======================");
    for(Object o : avgAmount){
        ObjectMapper od = new ObjectMapper();
        String json = od.writeValueAsString(o);
        LOGGER.info("cartId, avg = " + json);
    }
}
```

Figura 15: Código sales cron

```
1 usage  sergiolv
@Query(value = "SELECT  cart_id, COUNT(DISTINCT client_name) FROM sales as s GROUP BY cart_id", nativeQuery = true)
List<Object> getDailyCustomers();
1 usage  sergiolv
@Query(value = "select cart_id, count(id) from sales group by cart_id", nativeQuery = true)
List<Object> getDailySales();

1 usage  sergiolv
@Query(value = "select cart_id, avg(amount) from sales group by cart_id", nativeQuery = true)
List<Object> getAvgAmount();
```

Figura 16: Queries a la db

3.6. Stocks Consumer

Este consumer, se encarga de leer del tópico "stocksçon el groupId="stocks-listenerz llena un arreglo de tamaño 5. Cuando este se llena, se loggea los carritos necesarios. Esta parte de código, tambien podría implementarse como alerta, preguntando por el remainingStock y si hay algún límite relevante, se loggea una alerta o se podría ejecutar una comunicación con algún otro servicio.

```
private static List<CartStock> cartStockList = new ArrayList<>();

sergiolv
@KafkaListener(topics = Constants.KAFKA_STOCKS_TOPIC, groupId = Constants.COUNSUMER_GROUP_ID)
void listener(String JsonStock) throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    CartStock cartStock = objectMapper.readValue(JsonStock, CartStock.class);
    if(cartStockList.size() == 5){
        for(CartStock cartStock1: cartStockList){
            LOGGER.info("cartId " + String.valueOf(cartStock1.getCartId()) + " with " + String.valueOf(cartStock1.getRemainingStock()) + " sopaipillas left");
        }
        cartStockList.clear();
    }
    cartStockList.add(cartStock);
}
```

Figura 17: Código members consumer

3.7. Members Consumer

En este caso, el consumer de members tiene el groupId = "members-listener.[en] el tópico "membersz lee de ambas particiones a la vez ya que no se necesita ninguna acción en particular.

```
sergiolv
@KafkaListener(topics = Constants.KAFKA_NEW_MEMBERS_TOPIC, groupId = Constants.COUNSUMER_GROUP_ID)
void listener(String JsonSale) throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    Member member = objectMapper.readValue(JsonSale, Member.class);
    LOGGER.info("Nuevo miembro! " + member.getMemberName() + " " + member.getMemberLastName());
}
```

Figura 18: Código members consumer

3.8. Location Consumer

El Location Consumer tiene listeners con gropuId = "location-listenerz leen desde el tópico "location". Un listener lee desde la partición de carritos activos y el otro de carritos fugitivos.

```
sergiolv
@KafkaListener(groupId = Constants.COUNSUMER_GROUP_ID,  topicPartitions = @TopicPartition(topic = Constants.KAFKA_LOCATION_TOPIC, partitions = {"0"}))
void firstListener(String JsonSale) throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    CartLocation cartLocation = objectMapper.readValue(JsonSale, CartLocation.class);
    LOGGER.info("Location of cartId " + cartLocation.getCartId() + " updated. " + cartLocation.getLatitude() + " " + cartLocation.getLongitude());
}

sergiolv
@KafkaListener(groupId = Constants.COUNSUMER_GROUP_ID,  topicPartitions = @TopicPartition(topic = Constants.KAFKA_LOCATION_TOPIC, partitions = {"1"}))
void secondListener(String JsonSale) throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    CartLocation cartLocation = objectMapper.readValue(JsonSale, CartLocation.class);
    LOGGER.info("Cart id " + cartLocation.getCartId() + " is fugitive!");
}
```

Figura 19: Código cart location consumer

4. Preguntas

1. ¿Cómo Kafka puede escalar tanto vertical como horizontalmente? Relacione su respuesta con el problema asociado, dando un ejemplo para cada cada uno de los tipos de escalamiento.

2. ¿Qué características puede observar de Kafka como sistema distribuido? ¿Cómo se reflejan esas propiedades en la arquitectura de Kafka?

4.1. Respuestas

1. El escalamiento horizontal consiste en tener varios servidores como nodos, este es el más potente pero el más complicado ya que tienen que trabajar todos como uno. Para ellos se crean clúster y se distribuye la carga de trabajo. Kafka puede escalar horizontalmente ya que puede distribuir la carga de trabajo y dar de alta nuevos brokers así operando a mayor capacidad. Los topics pueden

   El escalamiento vertical consiste en actualizar el hardware e instalar uno más potente como un disco duro, CPU, el problema que presenta este tipo de escalamiento es que tiene un límite, ya que al llegar un cierto nivel no se puede escalar. Kafka permite el escalamiento vertical ya que los servidores se pueden reemplazar por unos más potentes.

2. Alta escalabilidad,alta disponibilidad, concurrencia, modularidad, mayor tolerancia a fallos mayor velocidad, flexibilidad, modularidad, alto rendimiento. En Kafka se reflejan diferentes características en la arquitectura como son la escalabilidad ya que puede tener cientos de brokers,

5. Repositorio

Todo el codigo que se muestra en el video y en este informe, puede ser revisado en

https://github.com/SergioLV/kafka-implementation-spring