MEMORIA FINAL DEL PROYECTO

Small Big Data

CICLO FORMATIVO DE GRADO SUPERIOR

Administración de Sistemas Informáticos en Red

AUTOR Francisco Carlavilla Toural

TUTOR Teresa González

COORDINADOR Maria José Villar Ruibal

CURSO 2019-2020

Fecha de entrega: 10/06/2020

I.E.S. CLARA DEL REY

Unión Europea

Fondo Social Europeo

"El FSE invierte en tu futuro"

ÍNDICE DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

I. INTRODUCCIÓN

I. Introducción al Big Data

Big data es un termino usado para definir grandes volúmenes de datos que por el rápido escalado de los mismos, así como por la complejidad del dato, que serían imposibles de controlar sin sistemas distribuidos. Es por ello que se utilizan tecnologías más recientes y avanzadas con el fin de conseguir substraer información de los mismos, además de almacenarlos y una mayor velocidad de consulta. Hay tecnologías con reconocimiento mundial y uso por parte de muchas de las grandes compañías que son de código libre por lo que cualquiera puede utilizarlas. Algunos ejemplos que se van a usar en este proyecto son Apache Kafka y Apache Zookeeper que pertenecen a la Apache Software Foundation o PostgreSQL.

II. Introducción al proyecto

Small Big Data es el uso a pequeña escala de lo explicado anteriormente. La idea es llevar a cabo un caso de uso de un banco. En este ejemplo distintos usuarios realizarán transacciones y nosotros guardaremos dichos movimientos bancarios en una base de datos y de esta los llevaremos a una página web en las que se podrán visualizar el nombre, la cantidad movida en dicha transacción y la fecha en la que se realizó. Todo esto ordenado por cantidad de dinero en la cuenta de cada persona.

Lo primero que haremos será producir dichas transacciones con un cliente en Java de Apache Kafka. Lo siguiente, será consumir dichos registros con otra conexión al mismo Apache Kafka. Esta quedará abierta, para que en caso de que se produzcan múltiples registros simultáneos se puedan registrar todos en la base de datos.

De esta forma lograremos de forma instantánea modelar los datos de la transacción y mostrar en la web el acumulado de dicha persona, así como modificar el orden según el dinero en la cuenta. Esto es algo que sería imposible llevar a cabo sin estas tecnologías debido al gran volumen de datos.

III. ¿Por qué este proyecto?

A día de hoy, dado que el Big Data mueve el mundo, cuanto más se sepa de ese mundo, mejor será. En mi empresa trabajamos el Big Data, pero al yo pertenecer a la parte de producto, trato con dichas tecnologías para darles una capa de seguridad, hacer que tengan alta disponibilidad o controlar que aunque una tarea falle, se vuelva a desplegar automáticamente con el mismo disco de persistencia para no perder datos. Por ello, tenía un gran vacío de conocimiento en cuanto a como se usaban las tecnologías a modo de cliente y dado que es parte de mi trabajo me llamaba la atención de que forma podrían realizarse dichas actividades por medio de microservicios.

Las pruebas que realizaremos serán las siguientes:

- Producir varios valores de transacciones, todos deberían de ser válidos.
- Consumir dichos valores desde otro terminal.
- Comprobar que dichos valores se han actualizado automáticamente en la base de datos.
- Verificar que la web se ha actualizado correctamente con las horas correctas.
- La última prueba será comprobar que las distintas formas de fallo tienen las excepciones o salidas de la aplicación controladas.

II. ARQUITECTURA

I. Tecnologías

I. PostgreSQL es una base de datos relacional de código libre orientada a objetos (ODBMS).

Figura 1.1.: Logo de Postgresql.

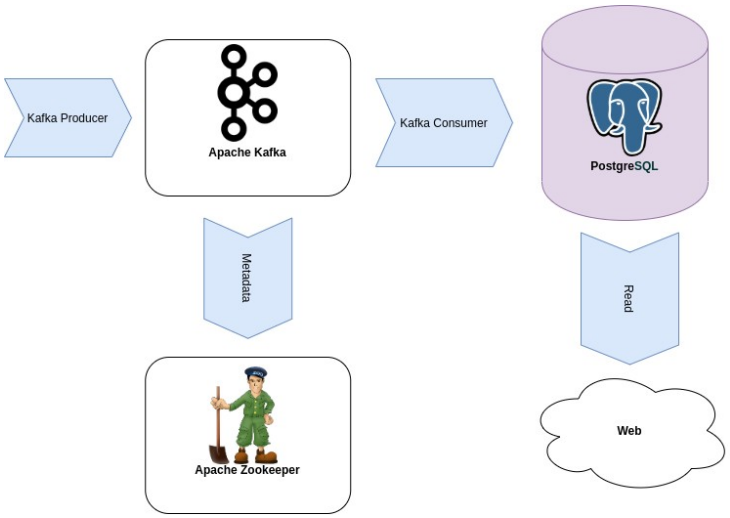
Figura 1.2.: Logo de Apache Kafka.

II. Apache Kafka es una cola de mensajes distribuida, desarrollada por la Apache Software Foundation. Kafka se utilizará para producir sobre el los datos de los clientes y luego consumirlos del mismo.

III. Apache Zookeeper es un sistema de ficheros distribuido, uno de sus usos más comunes es como backend de Apache Kafka para guardar sus metadatos, así como los distintos topics.

Figura 1.3.: Logo de Apache Zookeeper.

II. Arquitectura general



Una vista global del proyecto consistiría en lo siguiente, en primer lugar se produce en Apache Kafka, este crea el topic en Apache Zookeeper y guarda sus metadatos propios, como pueden ser el número de instancias de Kafka. Apache Kafka guarda los mensajes del topic así como la configuración del mismo. Por otro lado, se consumen esos mensajes y se llevan a PostgreSQL y de ahí recoge el servidor web los datos.

Figura 2.1.: Arquitectura general.

III. Arquitectura del proyecto

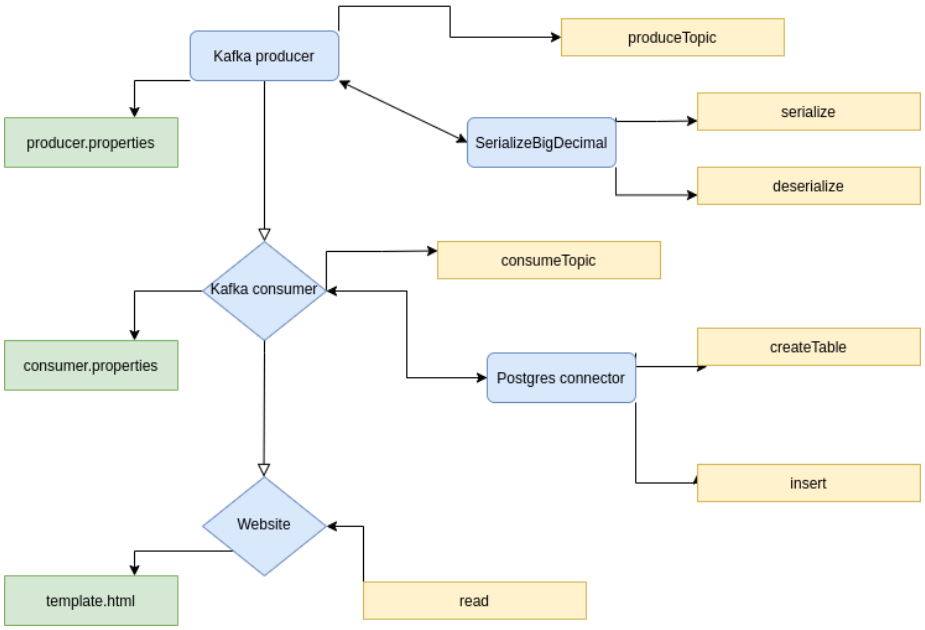


Figura 2.2: Arquitectura del proyecto.

III. ALCANCE DEL PROYECTO Y ANÁLISIS PREVIO

I. Objetivo

El objetivo principal de este trabajo es conseguir producir datos de transacciones ficticias y que utilizando Apache Kafka, Apache Zookeeper y PostgreSQL estos mismos, aparezcan ordenados en un servidor web haciendo unicamente uso de un ordenador con dockers y un proyecto en Java.

El resultado final, por tanto, se espera que sea el de poder escribir unos datos a través del productor de Apache Kafka, y que estos se muestren en el servidor web agrupados por nombre y ordenados por la cantidad de efectivo en la cuenta bancaria de dicho usuario. Teniendo en PostgreSQL dicha información desglosada por transacciones para así poder ver si se hubiese cometido un error en algún apartado del flujo.

II. Planificación previa

El primer paso para poder llevar a cabo este objetivo, es encontrar las imágenes docker de las tecnologías de las que queremos hacer uso. Necesitaremos además ver como se descargan dichas imágenes y cual es la forma de hacer que estas se encuentren en una red concreta de docker que permita su conexión. Deberemos además investigar sobre el cliente REST de Kafka del cual haremos uso para crear un consumidor y un productor.

Otro de los factores importantes es el de encontrar un servidor que se adecue a nuestro caso de uso. En este caso necesitaremos un servidor escrito en Java de código libre como podría ser Javalin.

Lo último que necesitaremos es un gestor de dependencias que nos permita hacer uso del cliente REST de Kafka, así como del de PostgreSQL y del servidor de Javalin, ya que necesitaremos hacer uso del código que hay en estos para la realización del proyecto.

III. Requisitos:

Para llevar a cabo dichos objetivos, necesitaremos tres contenedores docker, uno de Apache Kafka, uno de Apache Zookeeper y otro de PostgreSQL, todos ellos en la misma red de docker ya que será necesaria la comunicación. Este será el primer paso a realizar para poder llevar a cabo la práctica. Es importante probar desde dentro de los contenedores docker la conexión y el correcto funcionamiento para ver que están en la misma red y que no hay fallos.

Una vez tenemos los contenedores docker bien configurados, pasamos al desarrollo en Java de la aplicación para lo que necesitaremos, un consumidor de Apache Kafka, un productor de Apache Kafka, un conector con PostgreSQL y un servidor web.

Deberemos además de aceptar parámetros de entrada, ya que así se podrán realizar transacciones con distintos valores y se podrá mapear el servidor a distintos puertos del ordenador. Es necesario no forzar un puerto porque si se intentase desplegar en otra terminal ese puerto podría estar siendo utilizando haciendo así fallar al servidor.

El primero paso a realizar, es el de crear una red docker para que estos puedan conectarse entre si. Esta conexión es necesaria ya que Apache Kafka guarda todos sus metadatos y configuración en Apache Zookeeper y además el consumidor de Kafka se conectará a PostgreSQL para insertar los datos, por lo tanto, el proyecto no funcionaría sin dicha conexión entre los dockers. La red se crea con el siguiente comando, en este caso la red se llamará app-tier:

$ docker network create app-tier --driver bridge

Lo siguiente será, a la hora de desplegar los distintos docker introducirles como parámetros la red a la que van a pertenecer, teniendo por ello que introducir *--network app-tier* como parámetro en la creación de los mismos.

```
fcarlavilla@fcarlavilla ~/Escritorio/Escritorio/Workspace/TFG/target/Kafka.jar []$ docker inspect 428 | grep -i network
            "NetworkMode": "app-tier",
        "NetworkSettings": {
            "Networks": {
                "NetworkID": "44e951a6e94ef33a55e98c57ef3315beaba13f49205a625bce9ee5c4c024ea4d",
fcarlavilla@fcarlavilla ~/Escritorio/Escritorio/Workspace/TFG/target/Kafka.jar []$ docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS              PORTS                                        NAMES
428d300650f7        postgres                 "docker-entrypoint.s…"   26 hours ago        Up 26 hours         5432/tcp                                     postgresbd
3b0ea1133610        bitnami/kafka:latest     "/opt/bitnami/script…"   26 hours ago        Up 26 hours         9092/tcp                                     kafka-server
cff90175ca36        bitnami/zookeeper:latest "/opt/bitnami/script…"   26 hours ago        Up 26 hours         2181/tcp, 2888/tcp, 3888/tcp, 8080/tcp   zookeeper
fcarlavilla@fcarlavilla ~/Escritorio/Escritorio/Workspace/TFG/target/Kafka.jar []$
```

Figura 3.1: Muestra de red de los docker.

A continuación levantamos el docker de Apache Zookeeper con el nombre de “zookeeper” y con la red que acabamos de crear. Lo que estamos haciendo es intentar descargar la última imagen de docker que hay en bitnami, al no tener dicha imagen en local se descargará automáticamente.

*$ docker run -d --name zookeeper --network app-tier -e ALLOW_ANONYMOUS_LOGIN=yes *
bitnami/zookeeper:latest

En este caso vamos a lanzar el servidor de kafka, al igual que con Apache Zookeeper le asignaremos un nombre y la red anteriormente creada. En este caso tendremos el parámetro ALLOW_PLAINTEXT_LISTENER que indica que no tendrá una capa de seguridad SSL por encima, es decir, que cualquiera podrá producir y consumir sin tener permisos. Además, incluimos la conexión a Zookeeper, este como ya he dicho es un parámetro esencial ya que sino Kafka no podrá levantarse, ni funcionar, en caso de que la instancia de Zookeeper se cayese en algún momento dado de la prueba, Kafka dejaría de funcionar ya que sus topics se guardan ahí. Igual que antes vemos que estamos descargando la última versión del docker de bitnami ya que no lo tenemos en local.

docker run -d --name kafka-server --network app-tier -e ALLOW_PLAINTEXT_LISTENER=yes -e KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181 -e KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT -e KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,PLAINTEXT_HOST://:29092 -e KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-server:9092,PLAINTEXT_HOST://localhost:29092 bitnami/kafka:latest

Con esto ya podremos empezar a usar Kafka. Entonces, probamos si la conexión con Zookeeper es correcta. Para ello, probamos a crear un topic y producir y consumir del mismo. En este caso al crear cualquier topic habrá que poner replication-factor y partitions a 1 ya que únicamente tenemos un nodo. Para crear topics lo haremos de la siguiente manera, aunque hay un parámetro en Kafka llamado *auto.create.topics.enable* cuyo valor por defecto es “true”. Esto quiere decir que no hará falta crear los topics sino que simplemente empezar a producir apuntando a un topic inexistente este se creará de forma automática y empezará la producción de los mensajes mandados.

```
I have no name!@d70b37aa771e:/$ ./opt/bitnami/kafka/bin/kafka-topics.sh --create --topic prueba --partitions 1 --replication-factor 1 --zookeeper zookeeperserver:2181
Created topic prueba.
I have no name!@d70b37aa771e:/$ ./opt/bitnami/kafka/bin/kafka-topics.sh -list --zookeeper zookeeperserver:2181
prueba
I have no name!@d70b37aa771e:/$
```

Figura 3.2: Comando para crear topics en Kafka.

Cabe resaltar que el docker de Kafka lleva dentro un binario que contiene todos los scripts para poder usar la tecnología. Estos sirven para poder listar, crear y borrar topics, poder consumir y producir y poder hacer muchas otras tareas de administración de la instancia de Kafka.

```
I have no name!@d70b37aa771e:/$ cd /opt/bitnami/kafka/bin/
I have no name!@d70b37aa771e:/opt/bitnami/kafka/bin$ ls
connect-distributed.sh        kafka-console-consumer.sh    kafka-dump-log.sh                  kafka-reassign-partitions.sh      kafka-topics.sh                zookeeper-server-start.sh
connect-mirror-maker.sh       kafka-console-producer.sh    kafka-leader-election.sh           kafka-replica-verification.sh     kafka-verifiable-consumer.sh   zookeeper-server-stop.sh
connect-standalone.sh         kafka-consumer-groups.sh     kafka-log-dirs.sh                  kafka-run-class.sh                kafka-verifiable-producer.sh   zookeeper-shell.sh
kafka-acls.sh                 kafka-consumer-perf-test.sh  kafka-mirror-maker.sh              kafka-server-start.sh             trogdor.sh
kafka-broker-api-versions.sh  kafka-delegation-tokens.sh   kafka-preferred-replica-election.sh kafka-server-stop.sh              windows
kafka-configs.sh              kafka-delete-records.sh      kafka-producer-perf-test.sh        kafka-streams-application-reset.sh zookeeper-security-migration.sh
```

Figura 3.3: Muestra scripts de Kafka.

Para ello utilizaremos el kafka-console-producer.sh para producir mensajes en el topic produce. En la parte de arriba de la terminal estamos produciendo los mensajes mediante el comando:

$ ./kafka-console-producer.sh --bootstrap-server kafka-server:9092 --topic produce

Lo que hacemos con este comando es invocar al script que produce, indicarle a que nodo de Kafka vamos a atacar y a que puerto del mismo. Por último también le indicamos en que topic vamos a producir, tal y como he mencionado anteriormente este topic no existía pero al tener el parámetro **auto.create.topics.enable** por defecto a true se creará de forma automática.

Llama la atención el siguiente warning a la hora de producir, aunque como se puede ver se produce de igual modo:
[2020-04-25 10:10:47,445] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 3: {produce=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
Esto se solucionaría simplemente poniendo en el server.properties la siguiente linea:
advertised.host.name = localhost

Pero al no tener acceso a root no podemos descargar nada dentro del docker y no existen vim, vi y nano dentro del mismo.
Por otro lado, en la parte inferior de la consola podemos ver como se consume mediante el comando:

$ ./kafka-console-consumer.sh --bootstrap-server kafka-server:9092 --topic produce --from-beginning

En este caso, estamos llamando al script que consume, indicándole de nuevo el nodo de Kafka al que queremos atacar y el puerto, se podría atacar a más de uno si tuviésemos varias instancias desplegadas, el mismo topic en el que hemos producido y por último el flag “--from-beggining” que lo que hace es consumir desde el primer mensaje que haya dentro del topic “produce”. Sin este flag solo se consumirían los mensajes a partir del momento en el cual el consumidor se haya empezado a ejecutar; por tanto si empezamos a producir previamente, se perderían los mensajes en el periodo de tiempo entre que empiezan el productor y el consumidor.

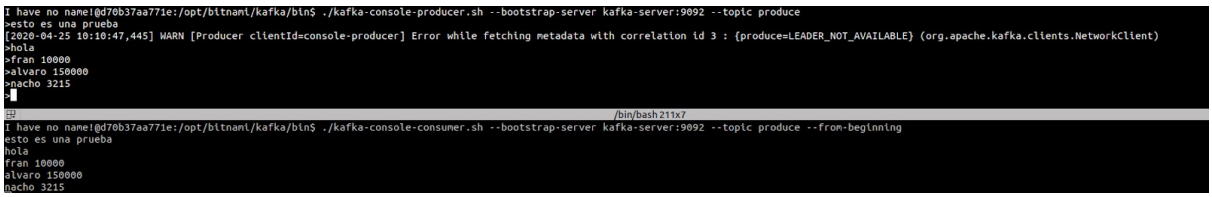


Figura 3.4: Ejemplo producción y consumición desde consola.

Una vez que hemos comprobado que se puede consumir correctamente desplegamos el docker de PostgreSQL.

$ docker run --name postgresbd --network=app-tier -e POSTGRES_PASSWORD=mysecretpassword -d postgres

Hacemos que esté en la misma red de docker para que se pueda comunicar con la instancia de Kafka y hacemos ping a dicha instancia para comprobar que podemos comunicarnos correctamente.

IV. FLUJO DE TRABAJO

I. Productor de Apache Kafka

En la clase ProducerKafka, descrita en el dibujo como “Kafka producer” se abre una conexión al cliente REST de Apache Kafka. Esta clase necesita de tres parámetros para producir, es decir, escribir mensajes, en la instancia de Kafka. Estos son: el nombre del topic, el cual se creará automáticamente en caso de que no exista previamente, el nombre del usuario del banco y la cantidad de efectivo que desea depositar o extraer.

```java
package org.fcarlavilla.smallBigData;
import ...

public class ProducerKafka {

    private static final  Logger log = LoggerFactory.getLogger(ProducerKafka.class);

    private KafkaProducer<String, BigDecimal> producer;
    public ProducerKafka(Properties props){

        producer = new KafkaProducer<>(props);
    }

    public void produceTopic(String topic, String key, BigDecimal value){

        producer.send((new ProducerRecord<>(topic,key, value )));
        log.info("Se registra movimiento de saldo de " + key + " por valor de "+ value);
        producer.close();
    }

    public static void main(String[] args) throws IOException {
        if (args.length==3){

            Properties propsP= new Properties();
            File producerProperties = new File( s: "/home/fcarlavilla/Escritorio/Escritorio/Workspace/TFG/src/main/resources/producer.properties");
            propsP.load(new FileInputStream(producerProperties));
            ProducerKafka produce = new ProducerKafka(propsP);
            BigDecimal money = new BigDecimal(args[2]);

            produce.produceTopic(args[0], args[1], money);

        }else if(args.length<3){
            log.error("Se han introducido demasiados argumentos, deberían de ser topic, nombre y cantidad de efectivo.");
        }else{
            log.warn("No se han introducido topic, nombre o cantidad de efectivo.");
        }
    }
}
```

Figura 4.1.1 Clase del productor de Apache Kafka.

Esta clase necesita además de el fichero *producer.properties*, donde se almacenan las propiedades de la producción, en este podríamos modificar múltiples variables de la misma como el factor de réplica o las particiones. En este caso se introduce la dirección de la instancia de Kafka, así como el puerto y el protocolo y demás propiedades importantes.

producer.properties

```
bootstrap.servers=http://kafka-server:9092
acks=all
retries=0
batch.size=16384
linger.ms=0
buffer.memory=33554432
key.serializer=org.apache.kafka.common.serialization.StringSerializer
value.serializer=org.fcarlavilla.smallBigData.SerializeBigDecimal
```

Figura 4.1.2: Fichero de propiedades del productor.

II. Serializador y deserializador de BigDecimal

La función de la clase *SerializeBigDecimal* es serializar el valor definido como *value* en este caso a BigDecimal. Apache Kafka tiene serializadores y deserializadores para varios tipos, pero este no es uno de ellos. Por eso, la clave, al ser un String, apunta a *org.apache.kafka.common.serialization.StringSerializer* pero el deserializador del productor apunta a *org.fcarlavilla.smallBigData.SerializeBigDecimal. Por tanto, el fin de esta clase es convertir los BigDecimal en bytes en el productor, a esto se le denomina "serialización", y reconvetir los bytes a BigDecimal en el consumidor, conocido como "deserialización".*

```java
package org.fcarlavilla.smallBigData;

import ...

public class SerializeBigDecimal implements Serializer<BigDecimal>, Deserializer<BigDecimal>, Closeable {
    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {

    }

    public byte[] serialize(String s, BigDecimal bigDecimal) {
        BigInteger value = bigDecimal.unscaledValue();
        int exponent = bigDecimal.scale();
        byte[] bytesValue = value.toByteArray();
        byte[] bytesExponent = ByteBuffer.allocate(4).putInt(exponent).array();
        byte[] allByteArray = new byte[bytesValue.length + bytesExponent.length];
        ByteBuffer buff = ByteBuffer.wrap(allByteArray);
        buff.put(bytesExponent);
        buff.put(bytesValue);
        byte[] result = buff.array();
        return result;
    }
    public BigDecimal deserialize(String s, byte[] bytes) {
        byte[] bytesExponent = Arrays.copyOfRange(bytes, i: 0, i1: 4);
        int exponent = new BigInteger(bytesExponent).intValue();
        byte[] byteValue = Arrays.copyOfRange(bytes, i: 4, bytes.length);
        BigInteger value = new BigInteger(byteValue);
        BigDecimal result = new BigDecimal(value, exponent);
        return result;
    }

    @Override
    public BigDecimal deserialize(String topic, Headers headers, byte[] data) { return deserialize(topic,data); }

    @Override
    public void close() {
    }
}
```

Figura 4.2: Clase del serializador y deserializador.

III. Consumidor de Apache Kafka

El fin de esta clase es consumir los mensajes que han sido producidos por el ProducerKafka, al igual que esta llama a la clase SerializeBigDecimal, así como a la clase PostgresConnector. A esta se le llama para hacer la inserción de los datos consumidos a PostgreSQL. Los parámetros necesarios en este caso serían dos: el nombre del topic, del que se va a consumir, y el nombre de la base de datos, en la que se quieren insertar los datos, en caso de que esta no existiese, se crearía.

```java
package org.fcarlavilla.smallBigData;
import ...
public class ConsumerKafka {
    private static final  Logger log = LoggerFactory.getLogger(ConsumerKafka.class);

    private KafkaConsumer consumer;
    private PostgresConnector postgresConnector;

    public ConsumerKafka(Properties props) throws SQLException{

        consumer = new KafkaConsumer<>(props);
        postgresConnector = new PostgresConnector();
    }
    public void consumeTopic(String topic, String tableName) throws SQLException {

        consumer.subscribe(Arrays.asList(topic));
        postgresConnector.createTable(tableName);
        //postgresConnector.read(tableName);
        List<ConsumerRecord<String, BigDecimal>> buffer = new ArrayList<>();
        while (true) {
            ConsumerRecords<String, BigDecimal> records = consumer.poll( timeoutMs: 100);
            for (ConsumerRecord<String, BigDecimal> record : records) {
                buffer.add(record);
                postgresConnector.insert(record.key(), record.value(), tableName);
            }
        }
    }
    public static void main(String[] args) throws IOException, SQLException{
        if (args.length ==2){

            Properties propsC = new Properties();
            File consumerProperties = new File( s: "/home/fcarlavilla/Escritorio/Escritorio/Workspace/TFG/src/main/resources/consumer.propert
            propsC.load(new FileInputStream(consumerProperties));
            ConsumerKafka consume = new ConsumerKafka(propsC);
            String tableName= args[1];

            consume.consumeTopic(args[0], tableName);

        }else if (args.length>2){
            log.error("Se han introducido demasiados parámetros, debería nde ser topic y nombre de la base de datos.");
        } else {
            log.error("No se ha introducido el nombre del topic o de la base de datos");
        }
    }
}
```

Figura 4.3.1: Clase del productor de Apache Kafka.

Al igual que el productor necesita un fichero con propiedades tales como la dirección de la instancia de Kafka y el puerto de la misma. Así mismo también necesita del SerializeBigDecimal en este caso para reconvertir los bytes a BigDecimal.

consumer.properties

```
bootstrap.servers=kafka-server:9092
group.id=test
enable.auto.commit=false
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.fcarlavilla.smallBigData.SerializeBigDecimal
```

Figura 4.3.2: Fichero de propiedades del consumidor.

IV. Conector PostgreSQL

El fin de la clase *PostgresConnector* es abrir la conexión con PostgreSQL, crear la tabla si no existe e introducir los datos en la misma. A esta clase, tal y como se ve en la **Figura 2.2: Arquitectura del proyecto.** se le llama desde la clase del consumidor. Para abrir la conexión a PostgreSQL se usa la JDBC con lo que la cadena de conexión final sería *j*dbc:postgresql://ip:puerto/instancia. Siendo *postgresql* el tipo de base de datos que se está utilizando.

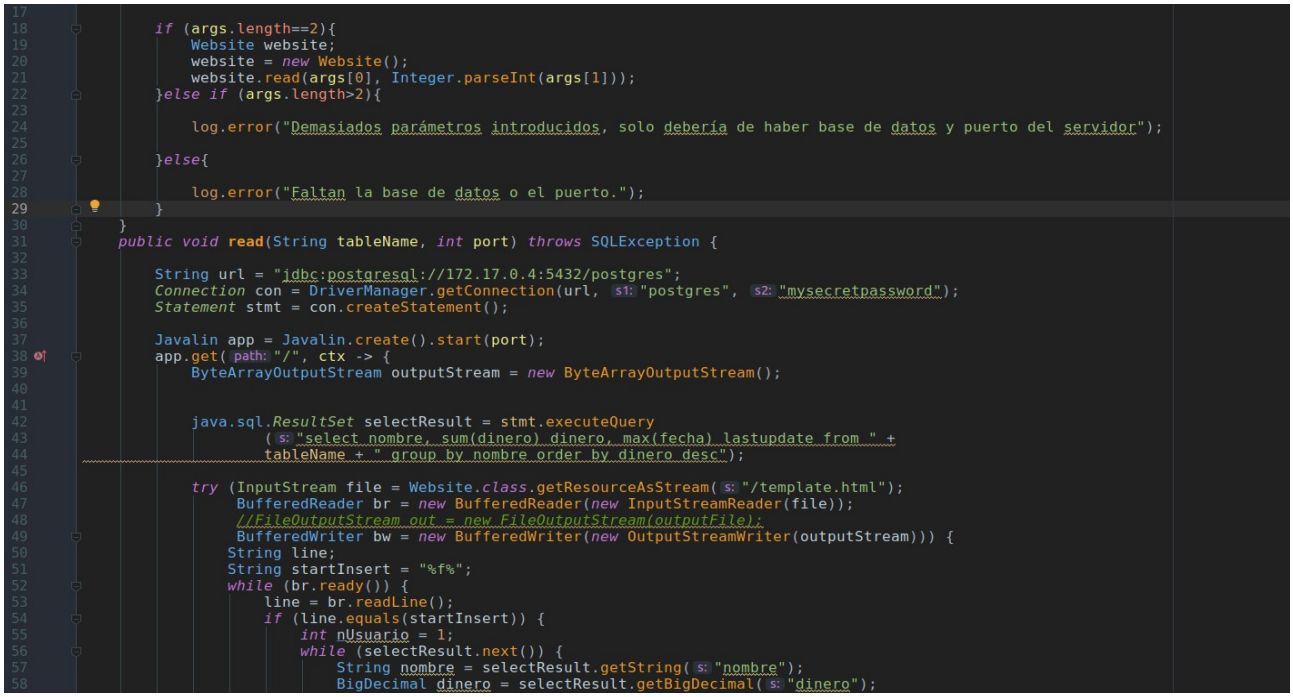
```java
package org.fcarlavilla.smallBigData;

import ...

public class PostgresConnector {

    private static final Logger log = LoggerFactory.getLogger(PostgresConnector.class);

    public PostgresConnector() throws SQLException{

        try {
            Class<?> driverClass = Class.forName("org.postgresql.Driver");
            Driver driver = (java.sql.Driver) driverClass.newInstance();
            DriverManager.registerDriver(driver);
        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException ex) {
            log.error("Error al registrar el driver de PostgreSQL: " + ex);
        }

    }

    public void createTable(String tableName) throws SQLException {


        String url = "jdbc:postgresql://172.17.0.4:5432/postgres";
        String sql = "create table if not exists " + tableName + " (nombre varchar(20), dinero numeric, fecha timestamp with time zone)";
        Connection con = DriverManager.getConnection(url, s1: "postgres", s2: "mysecretpassword");
        Statement stmt = con.createStatement();
        stmt.executeUpdate(sql);

    }

    public void insert(String nombre, BigDecimal dinero, String tableName) throws SQLException {


        String url = "jdbc:postgresql://172.17.0.4:5432/postgres";
        Connection con = DriverManager.getConnection(url, s1: "postgres", s2: "mysecretpassword");
        Statement stmt = con.createStatement();
        String insert = "insert into " + tableName + "(nombre, dinero, fecha) values('" + nombre +"', '" + dinero + "', current_timestamp)";
        stmt.executeUpdate(insert);

    }
}
```

Figura 4.4: Clase del conector de PostgreSQL.

V. Servidor web

La clase *Website*, se encarga de abrir una conexión a PostgreSQL para hacer una lectura de la tabla que se le indique, abre un servidor con Javalin y muestra este en el puerto que se lo indiquemos de localhost. El servidor web que se abre no cerrará su conexión hasta que nosotros la demos por terminado, y permanecerá con la conexión abierta a PostgreSQL haciendo así que la página web este siempre actualizada.

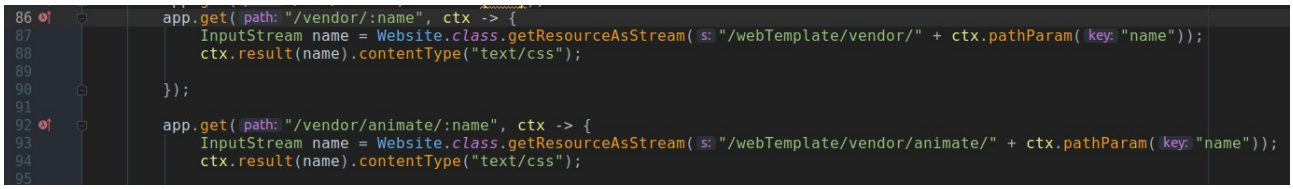
Esta clase necesita de dos parámetros: la base de datos que se quiere mostrar y el puerto local en el que se quiere mostrar dicha página.



```
        if (args.length==2){
            Website website;
            website = new Website();
            website.read(args[0], Integer.parseInt(args[1]));
        }else if (args.length>2){

            log.error("Demasiados parámetros introducidos, solo debería de haber base de datos y puerto del servidor");

        }else{

            log.error("Faltan la base de datos o el puerto.");
        }
    }
    public void read(String tableName, int port) throws SQLException {

        String url = "jdbc:postgresql://172.17.0.4:5432/postgres";
        Connection con = DriverManager.getConnection(url, s1: "postgres", s2: "mysecretpassword");
        Statement stmt = con.createStatement();

        Javalin app = Javalin.create().start(port);
        app.get( path: "/", ctx -> {
            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();


            java.sql.ResultSet selectResult = stmt.executeQuery
                    ( s: "select nombre, sum(dinero) dinero, max(fecha) lastupdate from " +
                    tableName + " group by nombre order by dinero desc");

            try (InputStream file = Website.class.getResourceAsStream( s: "/template.html");
                 BufferedReader br = new BufferedReader(new InputStreamReader(file));
                 //FileOutputStream out = new FileOutputStream(outputFile);
                 BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(outputStream))) {
                String line;
                String startInsert = "%f%";
                while (br.ready()) {
                    line = br.readLine();
                    if (line.equals(startInsert)) {
                        int nUsuario = 1;
                        while (selectResult.next()) {
                            String nombre = selectResult.getString( s: "nombre");
                            BigDecimal dinero = selectResult.getBigDecimal( s: "dinero");
```

Figura 4.5.1: Clase del productor de Apache Kafka.

En esta clase se hace además una redirección, ya que la plantilla de estilo que se ha cogido apuntaba a unas rutas locales, que dentro del .jar no existen.



```
        app.get( path: "/vendor/:name", ctx -> {
            InputStream name = Website.class.getResourceAsStream( s: "/webTemplate/vendor/" + ctx.pathParam( key: "name"));
            ctx.result(name).contentType("text/css");

        });

        app.get( path: "/vendor/animate/:name", ctx -> {
            InputStream name = Website.class.getResourceAsStream( s: "/webTemplate/vendor/animate/" + ctx.pathParam( key: "name"));
            ctx.result(name).contentType("text/css");
```

Figura 4.5.2 Clase del productor de Apache Kafka.

VI. Otros recursos

I. Logj42.properties

En todo el proyecto se usará log4j para que los logs sean más óptimos y estén formateados, para ello necesitaremos un log4j2.properties. De esta forma podremos controlar el nivel de log.

log4j2.properties

```
status = error
name = PropertiesConfig

filters = threshold

filter.threshold.type = ThresholdFilter
filter.threshold.level = debug

appenders = console

appender.console.type = Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

rootLogger.level = info
rootLogger.appenderRefs = stdout
rootLogger.appenderRef.stdout.ref = STDOUT
```

Figura 4.6.1 Fichero de propiedades de log4j2.

II. Html template

template.html

```
<!DOCTYPE html>
<html lang="en">
<head...>
<body>

<div class="limiter">
    <div class="container-table100">
        <div class="wrap-table100">
            <div class="table100 ver1 m-b-110">
                <div class="table100-head">
                    <table>
                        <thead>
                        <tr class="row100 head">
                            <th class="cell100 column1">#</th>
                            <th class="cell100 column2">Nombre</th>
                            <th class="cell100 column3">Dinero</th>
                            <th class="cell100 column4">Ultima actualización</th>
                        </tr>
                        </thead>
                    </table>
                </div>

                <div class="table100-body js-pscroll">
                    <table>
                        <tbody>
%f%
                        </tbody>
                    </table>
                </div>
            </div>
        </div>
    </div>
</div>


<!--===============================================================================================-->
<script src="/vendor/jquery/jquery-3.2.1.min.js"></script>
<!--===============================================================================================-->
<script src="/vendor/bootstrap/js/popper.js"></script>
<script src="/vendor/bootstrap/js/bootstrap.min.js"></script>
<!--===============================================================================================-->
<script src="/vendor/select2/select2.min.js"></script>
<!--===============================================================================================-->
<script src="/vendor/perfect-scrollbar/perfect-scrollbar.min.js"></script>
```

Se usa un template de html al que apuntan los css y js para de este modo mantener fijo el estilo de la página y que lo único que cambie sean los datos que se introducen, tal y como se ve en PostgresConnector.

Figura 4.6.2: Template de html para la web.

III. Ficheros de estilo

Como se comenta en la sección anterior se usan varios ficheros tanto css como js para dar estilo a la página.

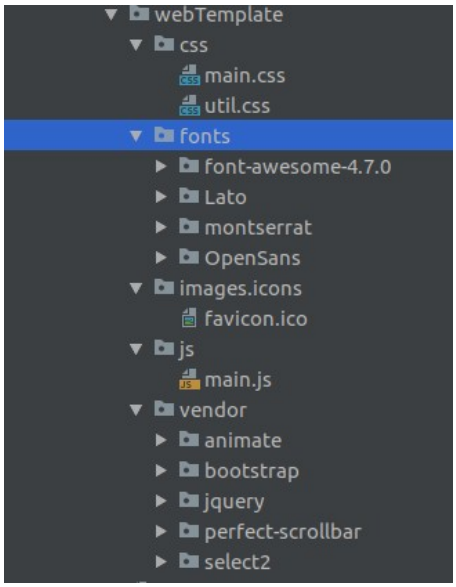


Figura 4.6.3: Ficheros de estilo usados.

IV. pom.xml

El último fichero que se usa para este proyecto es el *pom.xml*, en este se añaden valores importantes del proyecto como la ruta en la que se guardará el .jar, el nombre del mismo o las dependencias que se descargarán en el proyecto. Usaremos maven como repositorio para poder así construir nuestro proyecto.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

    <groupId>fcarlavilla</groupId>
    <artifactId>smallBigData</artifactId>
    <version>0.1.0</version>

    <repositories>
        <repository>
            <id>central</id>
            <name>Maven Central</name>
            <url>https://repo1.maven.org/maven2</url>
        </repository>
    </repositories>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.5.0</version>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.2.11</version>
        </dependency>
```

Figura 4.6.4: Fichero de dependencias pom.xml.

V. IMPLANTACIÓN DEL PROYECTO

En primer lugar, compilamos el proyecto para ver que no hay errores en el código.

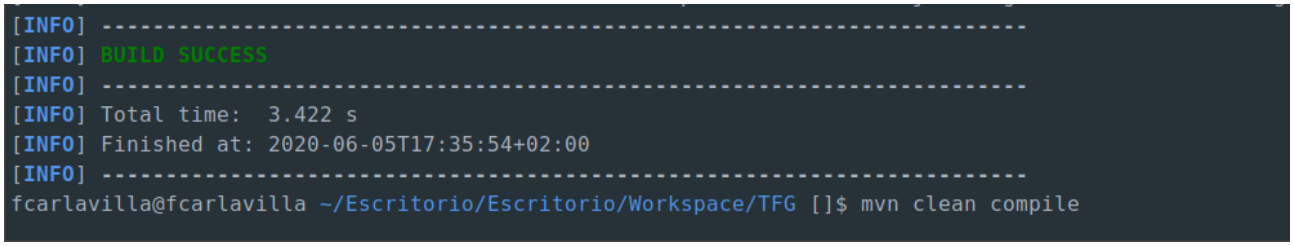


Figura 5.1.1: Compilación de proyecto.

Después, lo empaquetamos, es decir, creamos el .jar que ejecutaremos.

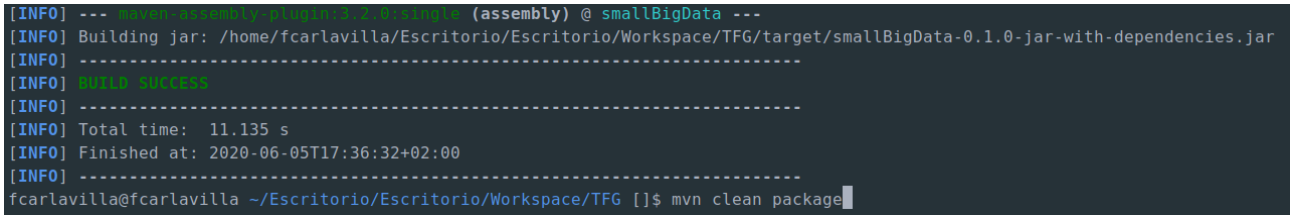


Figura 5.1.2: Empaquetado de proyecto.

Vamos a la ruta en la cual se ha creado el .jar y observamos que existe el .jar *smallBigData-0.1.0-jar-with-dependencies.jar*, el 0.1.0 pertenece a la versión y *smallBigData* al nombre que hemos dado al proyecto en el *pom.xml*.

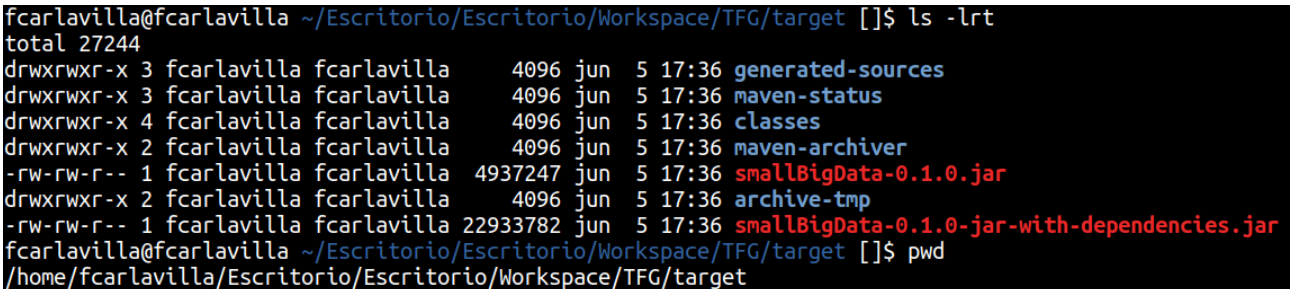


Figura 5.1.3: Ruta donde se empaqueta el proyecto.

Abriremos varias consolas, en una ejecutaremos el consumidor con el siguiente comando, al no ser esta la clase principal del proyecto tendremos que hacer *java -cp* e introducir la clase que queremos correr:

$ java -cp smallBigData-0.1.0-j-with-dependencies.jar org/fcarlavilla/smallBigData/ConsumerKafka topic tablaBaseDeDatos

En este caso crearemos en el topic “topic” el nombre “Nombre” y la cantidad de efectivo “532.23”, entonces el productor nos devolverá la configuración del productor, entre ellas la ip y puerto de la instancia de Kafka de nuestro producer.properties.

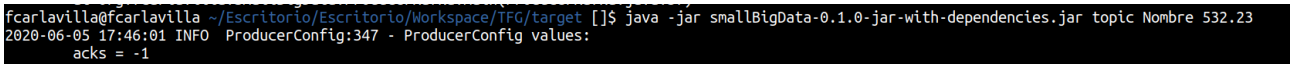


Figura 5.2.1: Productor desde .jar.

También podremos observar en los logs el nombre y la cantidad de efectivo que hemos introducido.

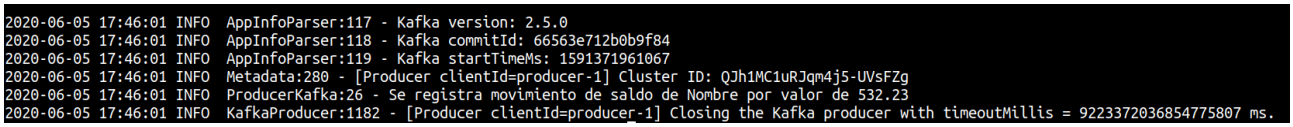


Figura 5.2.2: Logs de salida del productor desde .jar.

En otra abriremos el productor, en este haremos java -jar ya que la clase ProducerKafka es la principal del proyecto:
$ java -jar smallBigData-0.1.0-jar-with-dependencies.jar topic nombre dinero

En caso de tener dudas de que parámetros necesita, podemos lanzarlo sin ninguno y fallará devolviéndonos un log que nos indica que valores espera.

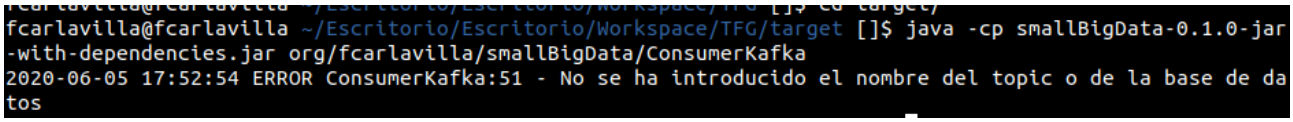


Figura 5.2.3: Consumidor desde .jar, ejemplo de error.

Al ejecutarlo correctamente, al igual que en el productor, nos devolverá las propiedades con las cuales estamos consumiendo y la conexión se mantiene abierta hasta que queramos cerrarla, de esta forma siempre que se produzca se actualizará al instante en la base de datos.

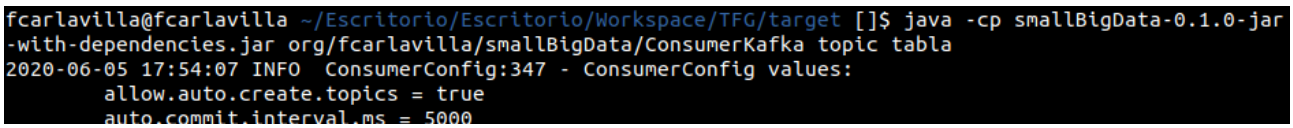


Figura 5.2.4: Consumidor desde .jar.

Comprobamos que están los datos en la tabla correspondiente, en este caso “tabla”.

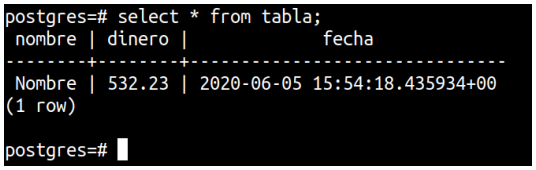


Figura 5.2.5: Comprobación desde PostgreSQL de la creación de la tabla.

Por último, abriremos el servidor con el siguiente comando:

java -cp smallBigData-0.1.0-jar-with-dependencies.jar org.fcarlavilla.smallBigData.Website tabla puerto

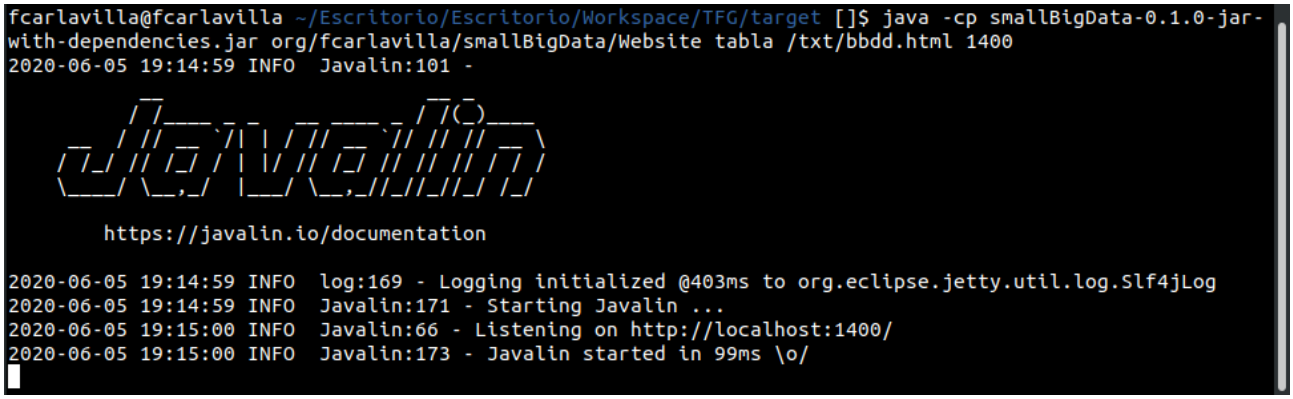


Figura 5.2.6: Lanzamiento del servidor web.

Accedemos al puerto que hemos indicado de localhost y observamos como hay una tabla con los datos insertados y la hora de la última actualización así como un contador que indica que clientes tienen mayor cantidad de efectivo.

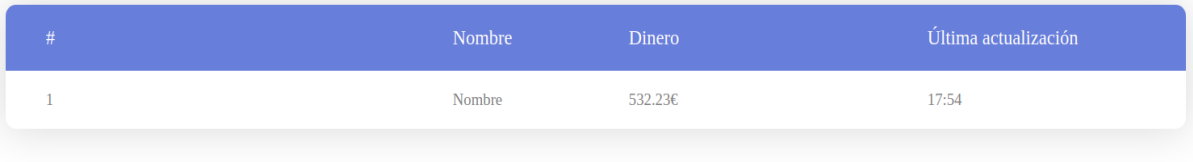


Figura 5.3: Comprobación de que el servidor web funciona.

VI. CONCLUSIONES

Este proyecto ha servido para adquirir nociones del uso de PostgreSQL, Apache Zookeeper y Apache Kafka así como de otras aplicaciones que han sido necesarias para el desarrollo del proyecto como Maven o Javalin. Además, da una imagen realista de como es el flujo de trabajo en Big Data y como los datos son analizados.

Otra de las finalidades por las cuales escogí este proyecto, es que en mi puesto de trabajo adquiero conocimientos sobre estas tecnologías desde el punto de vista del producto, es decir, hacer que tengan seguridad, alta disponibilidad y otras características basadas en que las tecnologías funcionen correctamente. Es por ello, que quería investigar sobre el uso de dichas tecnologías desde el área del uso como cliente.

En cuanto a la temporalización del proyecto, la planificación no fue del todo realista, ya que ha hecho falta una mayor cantidad de tiempo, debido en gran parte a los problemas encontrados. Además de ello, el proyecto se mejoró respecto a lo que iba a ser inicialmente con el uso de un servidor que se aloja en el puerto que escojamos de nuestro ordenador.

El principal problema encontrado durante la realización de dicho desarrollo ha sido que el cliente REST de Apache Kafka no tuviese un serializador y deserializador de BigDecimal. Otro de los problemas encontrados ha sido conseguir que la página web tuviese estilos ya que la ruta dentro del paquete es distinta a la de la máquina. Por ello, hubo que hacer un mapeo de las rutas del paquete de tal forma que cuando preguntasen por ciertas rutas, se redirigiese a otras dentro del jar. Consiguiendo así que se aplicasen los estilos correctamente.

De cara al futuro, se podría seguir desarrollando este trabajo haciendo que se pudiese producir en Apache Kafka introduciendo más de dos valores simultáneos, esto sería complejo ya que este solo acepta clave-valor.

Otro de los posibles enfoques del proyecto sería conseguir una mayor ingesta de datos y que en el servidor hubiese que iniciar sesión y solo nos mostrase nuestros datos.

Es por ello, que esta vía de investigación a futuro conllevaría un desarrollo del front en el que serían necesarios un usuario y una contraseña. Además para consumir y producir en Kafka harían falta certificados, así como para leer de PostgreSQL. Por lo que sería necesario aprender a crear dichos certificados.

Con todo este desarrollo a futuro realizado, esto serviría como aplicación real de venta de entradas o cualquier otra actividad online que requiriese un movimiento de saldo.

VII. GLOSARIO

Apache Kafka: Apache Kafka es una cola de mensajería distribuida, de código libre, que permite producir y consumir datos en tiempo real. Fue desarrollado por LinkedIn y ahora pertenece a la Apache Software Foundation.

Apache Zookeeper: Apache Zookeeper es un filesystem distribuido, de código libre, que se divide en znodes, dichos znodes pueden a la vez ser ficheros y directorios. Al igual que Apache Kafka pertenece a la Apache Software Foundation.

Big Data: Big data es un termino usado para definir grandes volúmenes de datos que por el rápido escalado de los mismos, así como por la complejidad del dato, se hacen imposibles de controlar con sistemas tradicionales, de esa limitación surgen los sistemas distribuidos.

Docker: Docker es una herramienta de virtualización, que en contraposición con la virtualización habitual es muy ligero. Funciona en todos los sistemas operativos. Es una herramienta muy usada dentro del Big Data.

Javalin: Javalin es una librería web, usada en este caso para el servidor.

JDBC: JDBC o Java Database Connectivity es el metodo de conexión que hay desde Java a las bases de datos.

Maven: Maven es un gestor de dependencias para proyectos escritos en Java que facilita el empaquetado y la compilación de dichos proyectos.

PostgreSQL: PostgreSQL es una base de datos relacional, distribuida, de código libre y orientada a objetos. Desarrollada por PostgreSQL Global Development Group.

VIII. BIBLIOGRAFÍA

- https://www.muylinux.com/2019/06/28/principales-tecnologias-big-data-2019/
- https://hub.docker.com/r/bitnami/kafka/
- https://stackoverflow.com/questions/35788697/leader-not-available-kafka-in-console-producer
- https://hub.docker.com/_/postgres
- https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html
- https://www.tutorialspoint.com/jdbc/jdbc-select-records.htm
- https://www.it-swarm.dev/es/java/no-se-puede-ejecutar-el-archivo-jar-no-hay-atributo-de-manifiesto-principal/941838957/
- http://codigoxules.org/conectar-postgresql-utilizando-driver-jdbc-java-postgresql-jdbc/
- https://colorlib.com/wp/template/fixed-header-table/
- https://www.google.com/search?q=postgresql&tbm=isch&ved=2ahUKEwiLmYzKsurpAhUT_RoKHS_7Bt4Q2-cCegQIABAA&oq=postgresql&gs_lcp=CgNpbWcQAzIECAAQQzIECAAQQzIECAAQQzICCAAyAggAMgIIADICCAAyAggAMgIIADICCAA6BQgAELEDOgcIABCxAxBDULSwAlj5ugJgz7sCaABwAHgAgAFBiAHNBJIBAjEwmAEAoAEBqgELZ3dzLXdpei1pbWc&sclient=img&ei=4xTaXsubOZP6a6_2m_AN&bih=759&biw=1536#imgrc=jdJ_h5rTgfbocM
- https://blogzeent.wordpress.com/2016/11/15/instalacion-de-apache-zookeeper-en-modo-cluster/
- https://es.wikipedia.org/wiki/PostgreSQL
- https://www.google.com/search?q=apache+kafka&source=lnms&tbm=isch&sa=X&ved=2ahUKEwiwy7K2surpAhVLDWMBHUMSBYwQ_AUoAXoECBYQAw&biw=1536&bih=759#imgrc=XTq1t8NmrZmHcM
- https://www.dataart.com.ar/news/para-que-nos-sirve-docker/
- Se ha utilizado además documentación interna de StratioBD.