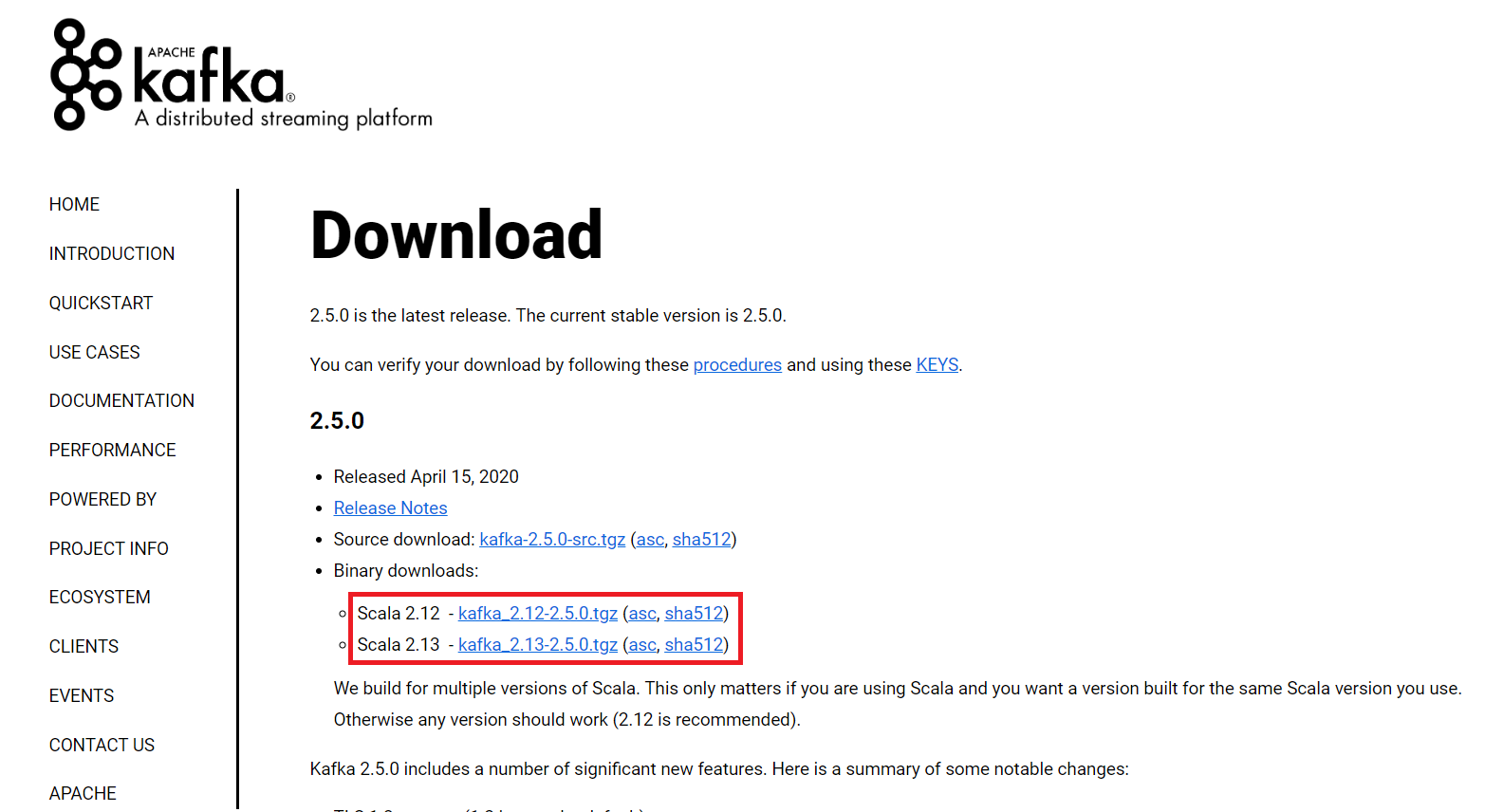
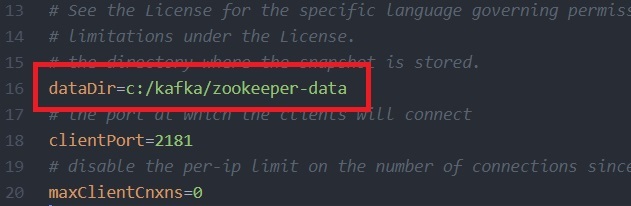
Quarkus streaming

**How to Install and Run Apache Kafka on Windows?**

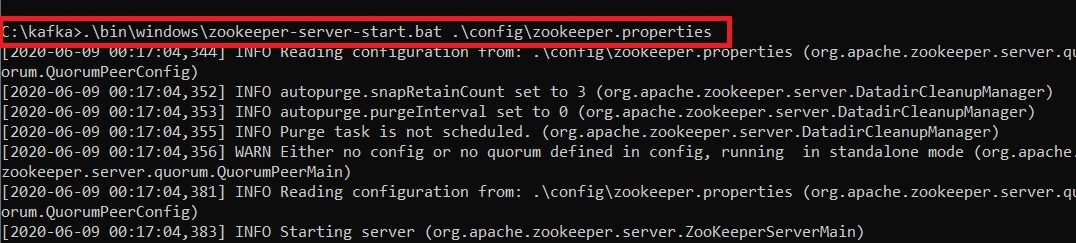
1. [Apache Kafka](https://www.geeksforgeeks.org/apache-kafka/) is an open-source application used for real-time streams for data in huge amount. Apache Kafka is a publish-subscribe messaging system. A messaging system lets you send messages between processes, applications, and servers. Broadly Speaking, Apache Kafka is software where topics can be defined and further processed.
2. **Downloading and Installation**
3. Apache Kafka can be downloaded from its official site [kafka.apache.org](https://kafka.apache.org/downloads)  
   
4. For the installation process, follow the steps given below:
5. **Step 1:** Go to the Downloads folder and select the downloaded Binary file.
6. **Step 2:** Extract the file and move the extracted folder to the directory where you wish to keep the files.

**Step 3:** Copy the path of the Kafka folder. Now go to *config* inside kafka folder and open *zookeeper.properties* file. Copy the path against the field *dataDir* and add */zookeeper-data* to the path.  
  
For example if the path is *c:/kafka*

**Step 4:** Now in the same folder *config* open *server.properties* and scroll down to *log.dirs* and paste the path. To the path add */kafka-logs*

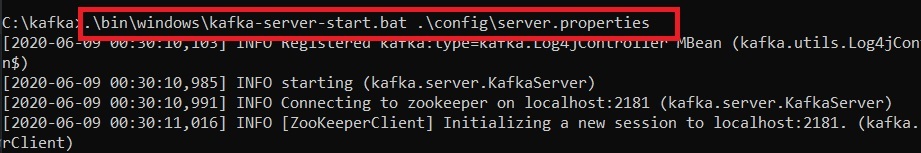
**Step 5:** This completes the configuration of zookeeper and kafka server. Now open command prompt and change the directory to the kafka folder. First start zookeeper using the command given below:

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties



**Step 6:** Now open another command prompt and change the directory to the kafka folder. Run kafka server using the command:

.\bin\windows\kafka-server-start.bat .\config\server.properties

  
Now kafka is running and ready to stream data.

Create Quarkus maven Project In intellij

Use/add following Dependencies:

<dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-messaging-kafka</artifactId>  
</dependency>  
<dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-kafka-client</artifactId>  
</dependency>  
<dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-kafka-streams</artifactId>  
</dependency>  
<dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-arc</artifactId>  
</dependency>  
<dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-junit5</artifactId>  
 <scope>test</scope>  
</dependency>

Create the following class:

package org.acme.kafka.streams.producer.generator;  
  
import java.math.BigDecimal;  
import java.math.RoundingMode;  
import java.time.Duration;  
import java.time.Instant;  
import java.util.Arrays;  
import java.util.Collections;  
import java.util.List;  
import java.util.Random;  
  
import jakarta.enterprise.context.ApplicationScoped;  
  
import io.smallrye.mutiny.Multi;  
import io.smallrye.reactive.messaging.kafka.Record;  
import org.eclipse.microprofile.reactive.messaging.Outgoing;  
import org.jboss.logging.Logger;  
  
  
@ApplicationScoped  
public class ValuesGenerator {  
  
 private static final Logger *LOG* = Logger.*getLogger*(ValuesGenerator.class);  
 private final Random random = new Random();  
  
 *// Define a list of bank names for generating bank details* private final List<String> bankNames = List.*of*("Bank of America", "Chase", "Wells Fargo", "Citibank", "HSBC",  
 "Goldman Sachs", "Morgan Stanley", "PNC", "U.S. Bank", "Capital One");  
  
  
  
 *// Generate messages for the bank-details topic* @Outgoing("bank-details")  
 public Multi<Record<Integer, String>> generateBankDetails() {  
 return Multi.*createFrom*().ticks().every(Duration.*ofSeconds*(1))  
 .onOverflow().drop()  
 .map(tick -> {  
 int bankId = random.nextInt(10) + 1; *// Bank ID from 1 to 10* String bankName = bankNames.get(bankId - 1); *// Get corresponding bank name* String bankRecord = String.*format*("%s|%d", bankName, bankId);  
  
 *LOG*.info("Generated bank details: " + bankRecord);  
 return Record.*of*(bankId, bankRecord);  
 });  
 }  
  
 *// Generate messages for the transaction-amounts topic* @Outgoing("transaction-amounts")  
 public Multi<Record<Integer, String>> generateTransactionAmounts() {  
 return Multi.*createFrom*().ticks().every(Duration.*ofSeconds*(1))  
 .onOverflow().drop()  
 .map(tick -> {  
 int bankId = random.nextInt(10) + 1; *// Randomly associate with a bank ID* double amount = BigDecimal.*valueOf*(random.nextDouble() \* 5000 + 100)  
 .setScale(2, RoundingMode.*HALF\_UP*)  
 .doubleValue(); *// Random amount between 100 and 5100* String amountRecord = String.*format*("%d|%.2f", bankId, amount);  
  
 *LOG*.info("Generated transaction amount: " + amountRecord);  
 return Record.*of*(bankId, amountRecord);  
 });  
 }  
  
  
}

This **ValuesGenerator** class is a part of a Quarkus project and is responsible for generating and emitting data to two Kafka topics**: bank-details** and **transaction-amounts.** It uses Multi from SmallRye Mutiny to create streams of data and @Outgoing annotations to specify the Kafka topics to which the data should be sent.

### Key Features and Functionality:

1. **Class Overview**:
   * The class is annotated with @ApplicationScoped, meaning it will be instantiated once and managed by the Quarkus container for the lifetime of the application.
   * A Logger is used to log generated messages for debugging and monitoring.
2. **Data Generation**:
   * The class generates two types of data streams, each emitted every second (Duration.ofSeconds(1)).
3. **Bank Details Generator (generateBankDetails method)**:
   * **Topic**: Publishes to the bank-details topic.
   * **Data Format**: Emits records with a format of "<Bank Name>|<Bank ID>".
   * **Details**:
     + Randomly selects a bank ID between 1 and 10.
     + Maps the ID to a bank name from a predefined list.
     + Logs and emits a record with the bank ID as the key and the formatted string as the value.
4. **Transaction Amounts Generator (generateTransactionAmounts method)**:
   * **Topic**: Publishes to the transaction-amounts topic.
   * **Data Format**: Emits records with a format of "<Bank ID>|<Transaction Amount>".
   * **Details**:
     + Randomly associates the generated amount with a bank ID between 1 and 10.
     + Generates a random transaction amount between 100 and 5100, rounded to two decimal places.
     + Logs and emits a record with the bank ID as the key and the formatted string as the value.

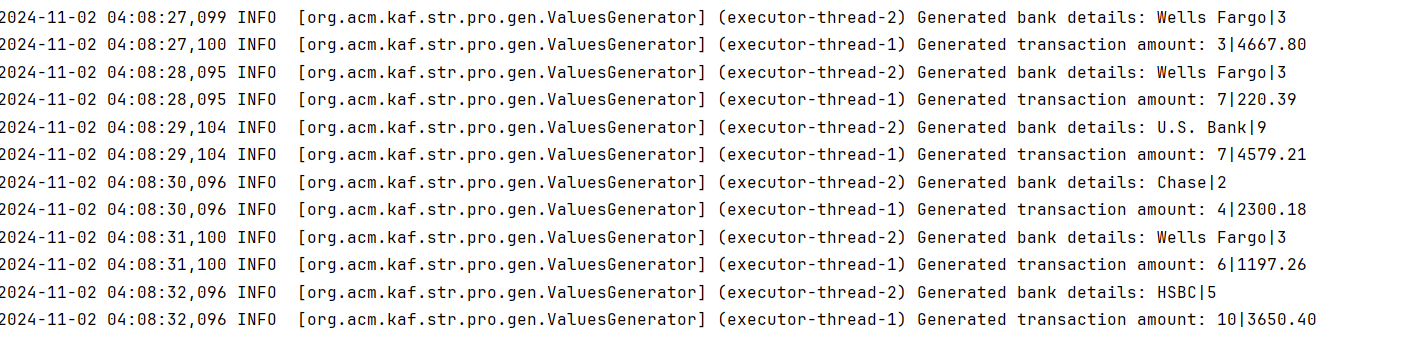
### Summary:

This class simulates the generation of data streams for a financial system:

* **bank-details**: Emits details of banks by generating a bank ID and corresponding name.
* **transaction-amounts**: Emits transaction amounts associated with a randomly selected bank ID.

Both methods log the generated data for monitoring and emit the records every second, ensuring continuous data flow to the specified Kafka topics.

**After Running The project we should see:**

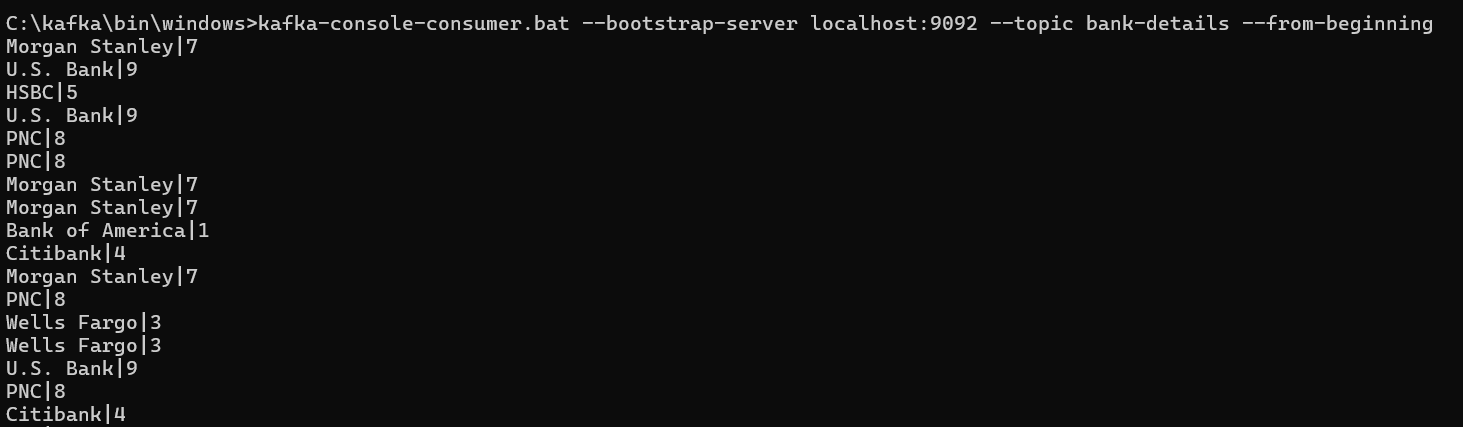


**Check Kafka Topics**

cd C:\kafka\bin\windows

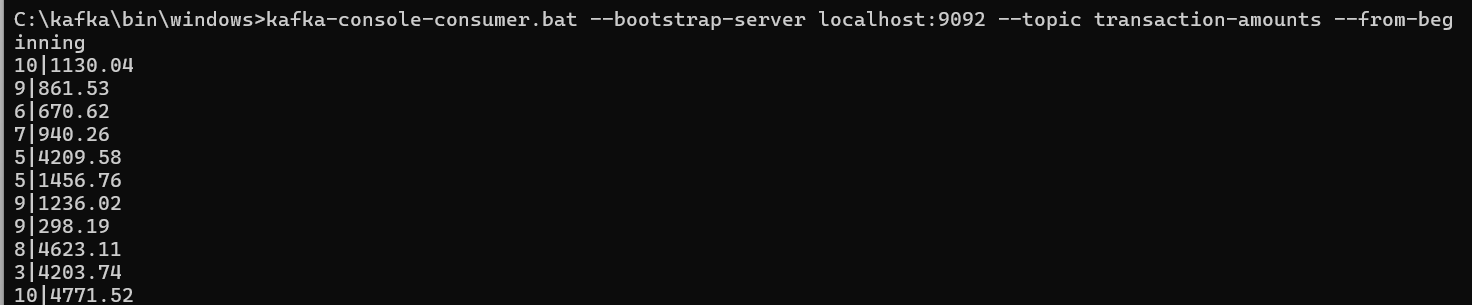
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic bank-details --from-beginning

**bank-details**



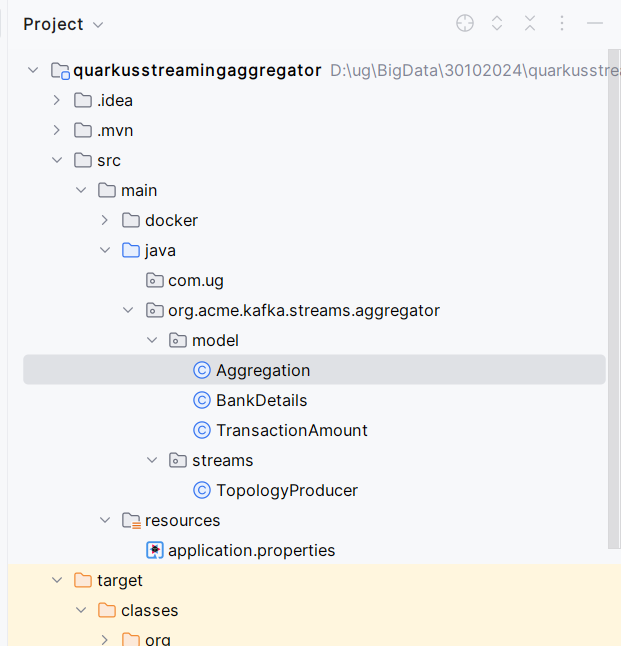
**transaction-amounts**

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic transaction-amounts --from-beginning



**Now Let’s run Aggregation Project**

**Project structure:**

****

**Use the following Dependencies:**

<dependencies>  
 <dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-messaging-kafka</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-kafka-client</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-kafka-streams</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-arc</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>io.quarkus</groupId>  
 <artifactId>quarkus-junit5</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>io.smallrye.reactive</groupId>  
 <artifactId>smallrye-reactive-messaging-api</artifactId>  
 <version>4.24.0</version>  
 </dependency>  
</dependencies>

Create the following classes:

**Class TransactionAmount:**

package org.acme.kafka.streams.aggregator.model;  
  
import io.quarkus.runtime.annotations.RegisterForReflection;  
  
  
@RegisterForReflection  
public class TransactionAmount {  
 public int bankId;  
 public String bankName;  
  
 public BankDetails(int bankId, String bankName) {  
 this.bankId = bankId;  
 this.bankName = bankName;  
 }  
  
 public String getName() {  
 return this.bankName;  
 }  
  
 public Object getBankName() {  
 return this.bankName;  
 }  
}

**Class TransactionAmount:**

package org.acme.kafka.streams.aggregator.model;  
  
  
import java.time.Instant;  
  
public class TransactionAmount {  
 public int bankId;  
 public String bankName;  
 public Instant timestamp;  
 public double amount;  
  
 public TransactionAmount(int bankId, String bankName, Instant now, double amount) {  
 this.bankId = bankId;  
 this.bankName = bankName;  
 this.amount = amount;  
 }  
}

**class Aggregation:**

package org.acme.kafka.streams.aggregator.streams;  
  
import io.quarkus.kafka.client.serialization.ObjectMapperSerde;  
import jakarta.enterprise.context.ApplicationScoped;  
import jakarta.enterprise.inject.Produces;  
import org.acme.kafka.streams.aggregator.model.TransactionAmount;  
import org.acme.kafka.streams.aggregator.model.BankDetails;  
import org.apache.kafka.common.serialization.Serdes;  
import org.apache.kafka.streams.StreamsBuilder;  
import org.apache.kafka.streams.Topology;  
import org.apache.kafka.streams.kstream.Consumed;  
import org.apache.kafka.streams.kstream.GlobalKTable;  
import org.apache.kafka.streams.kstream.Produced;  
import org.jboss.logging.Logger;  
  
import java.time.Instant;  
  
@ApplicationScoped  
public class TopologyProducer {  
  
 private static final Logger *LOG* = Logger.*getLogger*(TopologyProducer.class);  
  
 private static final String *BANK\_DETAILS\_TOPIC* = "bank-details";  
 private static final String *TRANSACTION\_AMOUNTS\_TOPIC* = "transaction-amounts";  
 private static final String *JOINED\_TRANSACTIONS\_TOPIC* = "joined-transactions"; *// New topic for joined data* @Produces  
 public Topology buildTopology() {  
 StreamsBuilder builder = new StreamsBuilder();  
 *LOG*.info("Starting to build Kafka Streams topology...");  
  
 *// Define a global table for bank details and parse values manually during join* GlobalKTable<Integer, String> bankDetailsTable = builder.globalTable(  
 *BANK\_DETAILS\_TOPIC*,  
 Consumed.*with*(Serdes.*Integer*(), Serdes.*String*())  
 );  
  
 *// Define the transaction pipeline with inline parsing and joining without aggregation* builder.stream(  
 *TRANSACTION\_AMOUNTS\_TOPIC*,  
 Consumed.*with*(Serdes.*Integer*(), Serdes.*String*())  
 )  
 .peek((key, value) -> *LOG*.debugf("Received transaction: key=%d, value=%s", key, value))  
 .join(  
 bankDetailsTable,  
 (bankId, transactionString) -> bankId,  
 */\*bankId: This is the key of the records coming from the KStream, which is the transaction-amounts topic.  
 The KStream reads records where the bankId is the key for each record.  
 transactionString: This is the value of the records coming from the KStream, which represents the  
 data in the transaction-amounts topic. It contains a string in the format  
 <Bank ID>|<Transaction Amount> (e.g., "1|500.00").  
 \*/* (transactionString, bankDetailsStr) -> {  
 try {  
 *// Parse bank details* String[] bankParts = bankDetailsStr.split("\\|");  
 if (bankParts.length != 2) {  
 *LOG*.warnf("Invalid bank details format: %s", bankDetailsStr);  
 return null;  
 }  
 int bankIdParsed = Integer.*parseInt*(bankParts[1]);  
 String bankName = bankParts[0];  
  
 *// Parse transaction details* String[] transactionParts = transactionString.split("\\|");  
 if (transactionParts.length != 2) {  
 *LOG*.warnf("Invalid transaction format: %s", transactionString);  
 return null;  
 }  
 double amount = Double.*parseDouble*(transactionParts[1]);  
  
 TransactionAmount transaction = new TransactionAmount(bankIdParsed, bankName, Instant.*now*(), amount);  
 *LOG*.debugf("Parsed transaction: %s", transaction);  
 return transaction;  
 } catch (Exception e) {  
 *LOG*.error("Error parsing transaction or bank details", e);  
 return null;  
 }  
 }  
 )  
 .filter((key, value) -> value != null) *// Filter out any null values from parsing errors* .to(*JOINED\_TRANSACTIONS\_TOPIC*, Produced.*with*(Serdes.*Integer*(), new ObjectMapperSerde<>(TransactionAmount.class))); *// Directly output to new topic  
  
 LOG*.info("Kafka Streams topology built successfully.");  
 return builder.build();  
 }

