Code Explanation

We started with kafka-python installation

```
pip install kafka-python
```

1. Initialize Spark Session

```
import os
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
```

Explanation:
Start by importing necessary libraries, including `os` and `sys` for system operations, and Spark's SQL, functions, and types modules. These imports are essential for setting up and managing our Spark session and working with structured data.

2. Initialize Spark Session

```
# Initialize Spark session
spark = SparkSession \
    .builder \
    .appName("spark_streaming_retail") \
    .getOrCreate()
spark.sparkContext.setLogLevel('ERROR')
```

Explanation:
Here, we initialize a Spark session with the application name "spark_streaming_retail". The Spark session is the entry point for reading data, creating DataFrames, and executing SQL queries. We also set the log level to 'ERROR' to reduce the amount of log output, allowing us to focus on critical issues.

3. Read Input from Kafka

```
# Read input from Kafka
ordersData = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "18.211.252.152:9092") \
    .option("failOnDataLoss", "false") \
    .option("startingOffsets", "earliest") \
    .option("subscribe", "real-time-project") \
    .load()
```

Explanation:
Connect to a Kafka server to read streaming data from the "real-time-project" topic. By setting `startingOffsets` to "earliest", we ensure that we process data from the beginning of the topic. The `failOnDataLoss` option set to "false" allows the stream to continue even if some data is lost.

4. Define Schema and Parse JSON Data

```
# Define the JSON schema for the data
schema = StructType([
    StructField("invoice_no", LongType(), False),
    StructField("country", StringType(), False),
    StructField("timestamp", TimestampType(), False),
    StructField("type", StringType(), False),
    StructField("items", ArrayType(StructType([
        StructField("SKU", StringType(), False),
        StructField("title", StringType(), False),
        StructField("unit_price", DoubleType(), False),
        StructField("quantity", IntegerType(), False)
    ])), True),
])

# Parse the JSON data
ordersDF = ordersData.select(from_json(col("value").cast("string"),
schema).alias("data")).select("data.*")
```

Explanation:
Define a schema for the incoming JSON data to ensure it is parsed correctly. This schema includes fields like invoice number, country, timestamp, type, and items, where items is an array of product details. Using this schema, we parse the Kafka stream to create a structured DataFrame, making it easier to perform transformations and analyses.

5. Define Schema and Parse JSON Data

```
# Define UDF functions
def calculate_total_price(items, type):
    total_price = 0
    for item in items:
        unit_price = item["unit_price"] if item["unit_price"] is not None
else 0
        quantity = item["quantity"] if item["quantity"] is not None else 0
        total_price += unit_price * quantity
    return -total_price if type == "RETURN" else total_price
```

Explanation:
The `calculate_total_price` function computes the total price of items in an order. If the transaction type is "RETURN", the total price is negated. This helps in accurately reflecting the monetary impact of both sales and returns.

```
# Define UDF functions
def calculate_total_items(items):
    total_items = 0
    for item in items:
        quantity = item["quantity"] if item["quantity"] is not None else 0
        total_items += quantity
    return total_items
```

Explanation:
The `calculate_total_items` function calculates the total number of items in a transaction by summing up the quantities of all items. This provides a count of items regardless of their individual prices.

```
# Define UDF functions
def is_order(type):
    return 1 if type == "ORDER" else 0
```

Explanation:
The `is_order` function checks if the transaction type is "ORDER". If it is, the function returns 1; otherwise, it returns 0. This helps in distinguishing orders from other transaction types.

```
# Define UDF functions
def is_return(type):
    return 1 if type == "RETURN" else 0
```

Explanation:
The `is_return` function checks if the transaction type is "RETURN". If it is, the function returns 1; otherwise, it returns 0. This helps in identifying return transactions separately from orders.

6. Register UDFs

```
# Register the UDFs
calculate_total_price_udf = udf(calculate_total_price, DoubleType())
calculate_total_items_udf = udf(calculate_total_items, IntegerType())
is_order_udf = udf(is_order, IntegerType())
is_return_udf = udf(is_return, IntegerType())
```

Explanation:
Register the user-defined functions (UDFs) with Spark, specifying their return types. This allows us to use these functions in DataFrame transformations, enabling custom calculations and logic.

7. Add New Fields to DataFrame

```
# Add the new fields to ordersDF
newOrdersDF = ordersDF \
    .withColumn("total_cost", calculate_total_price_udf(col("items"),
col("type"))) \
    .withColumn("total_items", calculate_total_items_udf(col("items"))) \
    .withColumn("is_order", is_order_udf(col("type"))) \
    .withColumn("is_return", is_return_udf(col("type")))
```

Explanation:
Enhance the DataFrame by adding new columns using the registered UDFs. These columns include:

- `total_cost`: The total cost of the transaction.
- `total_items`: The total number of items in the transaction.
- `is_order` and `is_return`: Flags to indicate whether the transaction is an order or return.

8. Write Summarized Data to Console

```
# Write summarized input values to the console
queryOrders = newOrdersDF.select("invoice_no", "country", "timestamp",
"total_cost", "total_items", "is_order", "is_return") \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .option("truncate", "false") \
    .trigger(processingTime="1 minute") \
    .start()
```

Explanation:
Set up a streaming query to output summarized data to the console. This includes details such as invoice number, country, timestamp, total cost, total items, and order/return flags. The data is updated every minute.

9. Calculate Time-Based KPIs

```
# Calculate time-based KPIs - Orders Per Minute (OPM), Total volume of
sales, and rate of returns
# Group by window
# Define a constant for the watermark duration
watermark_duration = "10 minutes"

AggByTime = newOrdersDF.withWatermark("timestamp", watermark_duration) \
    .groupBy(window(col("timestamp"), "1 minute")) \
    .agg(
        count("*").alias("OPM"),
        count(when(col("is_order") == 1, True)).alias("orders"),
        sum(col("total_cost")).alias("total_sale_volume"),
        count(when(col("is_return") == 1, True)).alias("returns")
    ) \
    .select(
        "window",
        "OPM",
        "total_sale_volume",
        (col("returns") / (col("orders") +
col("returns"))).alias("rate_of_return"),
        (col("total_sale_volume") / (col("orders") +
col("returns"))).alias("average_transaction_size")
    )
```

Explanation:
Calculate key performance indicators (KPIs) based on time, such as:

- Orders Per Minute (OPM)
- Total sales volume
- Rate of returns

Group the data into 1-minute windows and use a watermark of 10 minutes to handle late data. Aggregations are performed to compute these KPIs.

10. Calculate Time and Country-Based KPIs

```
# Calculate time and country-based KPIs - Orders Per Minute (OPM), Total
volume of sales, and rate of returns
# Group by country and window
AggByTimeAndCountry = newOrdersDF.withWatermark("timestamp",
watermark_duration) \
    .groupBy(window(col("timestamp"), "1 minute"), col("country")) \
    .agg(
        count("*").alias("OPM"),
        count(when(col("is_order") == 1, True)).alias("orders"),
        sum(col("total_cost")).alias("total_sale_volume"),
        count(when(col("is_return") == 1, True)).alias("returns")
    ) \
    .select(
        "window",
        "country",
        "OPM",
        "total_sale_volume",
        (col("returns") / (col("orders") +
col("returns"))).alias("rate_of_return")
    )
```

Explanation:
In addition to time-based KPIs, we calculate KPIs that are segmented by country. This allows us to analyze:

- Orders Per Minute (OPM) by country
- Total sales volume by country
- Rate of returns by country

This helps in understanding the performance across different geographical regions.

11. Write KPIs to HDFS

```
# Write time-based KPIs to JSON files
queryKpiByTime = AggByTime.writeStream \
    .outputMode("append") \
    .format("json") \
    .option("path", "user/hadoop/real-time-project/kpis/time-based") \
    .option("checkpointLocation",
"user/hadoop/real-time-project/kpis/time-based/checkpoint") \
    .trigger(processingTime="1 minute") \
    .start()

# Write time and country-based KPIs to JSON files
queryKpiByTimeAndCountry = AggByTimeAndCountry.writeStream \
    .outputMode("append") \
    .format("json") \
    .option("path",
"user/hadoop/real-time-project/kpis/time-and-country-based") \
    .option("checkpointLocation",
"user/hadoop/real-time-project/kpis/time-and-country-based/checkpoint") \
    .trigger(processingTime="1 minute") \
    .start()
```

Explanation:
Write the calculated KPIs to HDFS as JSON files. We use different paths and checkpoint locations for time-based and time-and-country-based KPIs. This ensures that the KPIs are stored persistently and can be used for further analysis and reporting.

Then we initiated the spark job with the below command:

```
export SPARK_KAFKA_VERSION=0.10
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0
spark_streaming.py > console_output
```

Explanation:

This command sets an environment variable `SPARK_KAFKA_VERSION` to `0.10`. This version specification is necessary because we are using Spark to integrate with Kafka, and it ensures compatibility between the versions of Spark and Kafka libraries.

This command submits a Spark application for execution. It includes the necessary Kafka connector package for Spark (`--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0`) and runs the `spark_streaming.py` script. The output, including logs and print statements, is redirected to the `console_output` file for later review.