SUMMER INTERNSHIP CUM INDUSTRIAL TRAINING REPORT

SANTANU KUNDU
ROLL: 18EC10052
BIG DATA TEAM,
ADFORM INDIA LLP, MUMBAI

TIMELINE

The duration of my internship was from JULY 4, 2022 to SEPTEMBER 20, 2022 spanning 12 weeks.

MENTOR

I worked under the mentorship of Mr. Pawan Sharma, Director of Development and IT of Big Data Team India, who is under Mr. Nirav Shah, SVP of Mumbai Adform Office.

OBJECTIVE AND SIGNIFICANCE OF THE PROBLEM

The main objective of my internship was to learn the fundamentals of handling and processing of Big Data in AdTech Industry. For AdTech Industries, they have to handle a lot of data without losing any data. So, the challenges of this team are as below.

1. To fetch the millions of data from first party, second party or third-party cookies of user and to call the bid and place the proper advertisement.
2. Another challenge is the processing of huge data within very few seconds.
3. For this case, we have to choose such a topology which is fault-tolerant and higher number of parallelisms.
4. We have to consider the sensitivity of data while processing or storing the data, as example we can not store any kind of personal and sensitive data and we have to follow GDPR Laws for European countries.

Here our objective was about transportation of Big Data and processing. Before learning and implementing these, I had to learn Scala 2.13 programming language. Later, I had to learn Apache Kafka, Apache Storm.

FIRST FOOTSTEP TO SOLVE THE PROBLEM

Scala 2.13

Scala is a strong statically typed general-purpose programming language which supports both object-oriented programming and functional programming. Designed to be concise, many of Scala's design decisions are aimed to address criticisms of Java. Scala source code can be compiled to Java bytecode and run on a Java virtual machine (JVM). Scala can also be compiled to JavaScript to run in a browser, or directly to a native executable. On the JVM Scala provides language interoperability with Java so that libraries written in either language may be referenced directly in Scala or Java code.

Unlike Java, Scala has many features of functional programming languages (like Scheme, Standard ML, and Haskell), including currying, immutability, lazy evaluation, and pattern matching. It also has an advanced type system supporting algebraic data types, covariance and contravariance, higher-order types (but not higher-rank types), anonymous types, operator overloading, optional parameters, named parameters, raw strings, and an experimental exception-only version of algebraic effects that can be seen as a more powerful version of Java's

checked exceptions. For this reason, Adform's Big Data Team prefers to use Scala as programming language in their project.

Apache Kafka

Apache Kafka is a distributed event store and stream-processing platform. It is an open-source system developed by the Apache Software Foundation written in Java and Scala. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka can connect to external systems (for data import/export) via Kafka Connect, and provides the Kafka Streams libraries for stream processing applications. Kafka uses a binary TCP-based protocol that is optimized for efficiency and relies on a "message set" abstraction that naturally groups messages together to reduce the overhead of the network roundtrip. This leads to larger network packets, larger sequential disk operations, contiguous memory blocks [...] which allows Kafka to turn a robust stream of random message writes into linear writes.

Apache Kafka was first introduced in 2011 with hands of LinkedIn, later it is maintained as an open source. In Apache Kafka, message transferring model is "Pub-Sub" model (Publish and subscribe model). There are five major API's,

1. Producer API – Permits an application to publish streams of records.

2. Consumer API – Permits an application to subscribe to topics and processes streams of records.

3. Connector API – Executes the reusable producer and consumer APIs that can link the topics to the existing applications. There are two types of connectors Source and Sink connectors.

4. Streams API – This API converts the input streams to output and produces the result.

5. Admin API – Used to manage Kafka topics, brokers, and other Kafka objects.

Apache Akka

Akka is a source-available toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang. Language bindings exist for both Java and Scala. Akka is written in Scala and, as of Scala 2.10, the actors in the Scala standard library are deprecated in favor of Akka.

The key points distinguishing applications based on Akka actors are:

1. Concurrency is message-based and asynchronous: typically, no mutable data are shared and no synchronization primitives are used; Akka implements the actor model.

2. The way actors interact is the same whether they are on the same host or separate hosts, communicating directly or through routing facilities, running on a few threads or many threads, etc. Such details may be altered at deployment time through a configuration mechanism, allowing

a program to be scaled up (to make use of more powerful servers) and out (to make use of more servers) without modification.

3. Actors are arranged hierarchically with regard to program failures, which are treated as events to be handled by an actor's supervisor (regardless of which actor sent the message triggering the failure). In contrast to Erlang, Akka enforces parental supervision, which means that each actor is created and supervised by its parent actor.

Akka has a modular structure, with a core module providing actors. Other modules are available to add features such as network distribution of actors, cluster support, Command and Event Sourcing, integration with various third-party systems (e.g., Apache Camel, ZeroMQ), and even support for other concurrency models such as Futures and Agents.

APPROACH

SETTING UP OF SCALA 2.13

1. As Scala runs on JVM, we need to install JDK first. Our Company suggested JDK were JDK 1.8 or JDK 11.

2. We need to extract Scala 2.13 and we need to add “bin” folder to path variables. Similarly, we have to install SBT, Curl (for Windows).

3. Now there are many IDEs, Visual Studio Code, IntelliJ Idea Community etc., one of them is needed to be installed. IntelliJ is preferred to others, as it has more useful features for Java Based Developers.

4. For Scala or JVM based libraries like Apache Akka, it can used adding library dependencies in “build.sbt” file in the project.

Some other software may be needed to install like curl, Postman on Windows Machine.

For Scala training, some main focused areas are:

1. Basic data structures, Map, Queue etc.

2. Functional programming and tail recursions.

3. Mutable and immutable data structures (Immutable are more preferred in Scala)

4. Class, Object, Case Class, Case Object, Companion Object etc.

5. Generics, lazy evaluation, implicit

6. Factory Methods and Styles to write Industry level Projects.

7. Concurrent programming, threads, Futures and Promises, Error Handling

8. Socket programming

To test all concepts, I was given a set of 21 problems to test my understanding level. To understand networking protocol and socket programming and multi-threading and asynchronous process, Chat Server design problem was given to me. We have to design a chat server, where clients can chat privately and publicly. The project link can be found here.

Apache Akka

Later I was given the same problem using Akka Behavioral Model. The main key areas to be understood here are:

1. We do not have to face any kind of racing condition.

2. Messaging systems are asynchronous and do not need synchronized block and have multi-threading facility.

3. Any state machine can be modelled using Akka without any kind of error.

For this, the project link can be found here.

Apache Kafka

We have to follow the steps to run Apache Kafka.

1. We have to download Apache Kafka and we need to extract the zip file.

2. If OS is Windows, we have to go inside bin/windows folder for setting up.

3. At first, we have to run zookeeper server.

4. Then, Kafka server is required to start.

5. Kafka connectors should be started then.

For processes 2 to 5, we have to run following commands on windows.

1. Go to Windows folder - **cd C:\kafka_2.13-3.2.3\bin\windows**

2. Start Zookeeper Server - **zookeeper-server-start.bat ../../config/zookeeper.properties**

3. Start Kafka Server - **kafka-server-start.bat ../../config/server.properties**

4. Start Kafka Connectors - **connect-distributed.bat ../../config/connect-distributed.properties**

You need to install curl or postman to create/check the status of Kafka connectors.

For this section, I was given to write a problem to make a source connector which will write into a topic from a database.

1. For this problem, I had to use some database. So, I used PostgreSQL. At first, I had to create a sample database, with network “localhost:5000” and “postgres” as user. I had created a server with name “PostgresSQL_14”.

2. With help of command lines, I made a database named “mytest” and a table named “orders” with public schemas.

I. To start PostgreSQL - psql -U postgres -p 5000

II. Provide the password.

III. To create database - CREATE DATABASE mytest;

IV. To create a table - create Table orders (order_id serial PRIMARY KEY, created_by VARCHAR (50) NOT NULL, order_date TIMESTAMP NOT NULL);

V. To insert data into table - insert into orders (order_date, created_by) values (current_timestamp, 'Jim Kim');

VI. To update data - update orders set created_by = 'Rosen Era' where order_id = 1;

3. With help of the command, I created a simple database.

4. For making a source connector, I needed to download Debezium.io and added the jar files inside libs folder in Kafka after extraction.

5. We need to start Zookeeper server, Kafka server and Kafka connectors.

6. For next step, we may use Postman or curl. I will discuss in separate points.

7. We have to write a JSON file as below.

I. Postman:

a. We need to post the JSON at “localhost:8083/connectors”.

b. We can check whether our connector has been created or not by requesting to get “localhost:8083/connectors/”.

c. We can also check the status of connector by requesting a get “localhost:8083/connectors/orders-source/status”.

II. Curl:

a. Before posting, we needed to save this JSON as orders.json. Then, write this command

curl -X POST -H "Content-Type: application/json" --data @orders.json http://localhost:8083/connectors

b. For checking whether connector is created or not, we can check here.

c. For checking status, we can check here too.


```
{
    "name": "orders-source",
    "config": {
        "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
        "tasks.max": 1,
        "database.hostname": "localhost",
        "database.port": 5000,
        "database.user": "postgres",
        "database.password": "admin",
        "database.dbname": "mytest",
        "database.server.name": "PostgreSQL_14",
        "schema.include.list": "public",
        "topic.creation.default.replication.factor": 1,
        "topic.creation.default.partitions": 1,
        "topic.creation.default.cleanup.policy": "compact",
        "topic.creation.default.compression.type": "lz4",
        "topic.creation.groups": "productlog",
        "topic.creation.productlog.include": "PostgreSQL_14\\.public\\.mytest.*",
        "topic.creation.productlog.replication.factor": 1,
        "topic.creation.productlog.partitions": 2,
        "plugin.name": "pgoutput",
        "slot.name": "solt_1"
    }
}
```


Figure 1: orders.json file

```
{
    "name": "orders-source",
    "config": {
        "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
        "tasks.max": "1",
        "database.hostname": "localhost",
        "database.port": "5000",
        "database.user": "postgres",
        "database.password": "admin",
        "database.dbname": "mytest",
        "database.server.name": "PostgreSQL_14",
        "schema.include.list": "public",
        "topic.creation.default.replication.factor": "1",
        "topic.creation.default.partitions": "1",
        "topic.creation.default.cleanup.policy": "compact",
        "topic.creation.default.compression.type": "lz4",
        "topic.creation.groups": "productlog",
        "topic.creation.productlog.include": "PostgreSQL_14\\.public\\.mytest.*",
        "topic.creation.productlog.replication.factor": "1",
        "topic.creation.productlog.partitions": "2",
        "plugin.name": "pgoutput",
        "slot.name": "solt_1",
        "name": "orders-source"
    },
    "tasks": [],
    "type": "source"
}
```


Figure 2: Response in Postman after posting request. (7.I.a)


```
[
    "orders-source"
]
```


Figure 3: Response in Postman to get the connector list. (7.I.b)

```
    "name": "orders-source",
    "connector": {
        "state": "RUNNING",
        "worker_id": "10.3.12.38:8083"
    },
    "tasks": [
        {
            "id": 0,
            "state": "RUNNING",
            "worker_id": "10.3.12.38:8083"
        }
    ],
    "type": "source"
}
```


Figure 3: Response in Postman to get the status of orders-source. (7.I.c)


```
{"name":"orders-
source","config":{"connector.class":"io.debezium.connector.postgresql.Postgre
sConnector","database.user":"postgres","database.dbname":"mytest","topic.crea
tion.default.compression.type":"lz4","topic.creation.default.partitions":"1",
"topic.creation.default.cleanup.policy":"compact","slot.name":"solt_1","tasks
.max":"1","database.server.name":"PostgreSQL_14","schema.include.list":"publi
c","database.port":"5000","plugin.name":"pgoutput","topic.creation.groups":"p
roductlog","topic.creation.productlog.replication.factor":"1","topic.creation
.productlog.include":"PostgreSQL_14\\.public\\.mytest.*","topic.creation.prod
uctlog.partitions":"2","database.hostname":"localhost","database.password":"a
dmin","topic.creation.default.replication.factor":"1","name":"orders-
source"},"tasks":[{"connector":"orders-source","task":0}],"type":"source"}
```


Figure 4: Response in Browser to get the connector list. (7.II.b)


```
{"name":"orders-
source","connector":{"state":"RUNNING","worker_id":"10.3.12.38:8083"},"tasks"
:[{"id":0,"state":"RUNNING","worker_id":"10.3.12.38:8083"}],"type":"source"}
```


Figure 5: Response in Browser to get the status of orders-source. (7.II.c)

8. We checked the list of topics - kafka-topics.bat --list --bootstrap-server localhost:9092. We saw PostgreSQL.public.orders named topic was created.

9. To get the data from beginning –

kafka-console-consumer.bat --topic PostgreSQL_14.public.orders --from-beginning --bootstrap-server localhost:9092

10. To get the latest change –

kafka-console-consumer.bat --topic PostgreSQL_14.public.orders --bootstrap-server localhost:9092

11. For visualization purpose, latest change will be taken into consideration.

I. We have inserted data –

insert into orders (order_date, created_by) values (current_timestamp,'Santanu Kundu');


```
{"schema":{"type":"struct","fields":[{"type":"struct","fields":[{"type":"int32","optio
nal":false,"default":0,"field":"order_id"},{"type":"string","optional":false,"field":"
created_by"},{"type":"int64","optional":false,"name":"io.debezium.time.MicroTimestamp"
,"version":1,"field":"order_date"}],"optional":true,"name":"PostgreSQL_14.public.order
s.Value","field":"before"},{"type":"struct","fields":[{"type":"int32","optional":false
,"default":0,"field":"order_id"},{"type":"string","optional":false,"field":"created_by
"},{"type":"int64","optional":false,"name":"io.debezium.time.MicroTimestamp","version"
:1,"field":"order_date"}],"optional":true,"name":"PostgreSQL_14.public.orders.Value","
field":"after"},{"type":"struct","fields":[{"type":"string","optional":false,"field":"
version"},{"type":"string","optional":false,"field":"connector"},{"type":"string","opt
ional":false,"field":"name"},{"type":"int64","optional":false,"field":"ts_ms"},{"type"
:"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":{"al
lowed":"true,last,false,incremental"},"default":"false","field":"snapshot"},{"type":"s
tring","optional":false,"field":"db"},{"type":"string","optional":true,"field":"sequen
ce"},{"type":"string","optional":false,"field":"schema"},{"type":"string","optional":f
alse,"field":"table"},{"type":"int64","optional":true,"field":"txId"},{"type":"int64",
"optional":true,"field":"lsn"},{"type":"int64","optional":true,"field":"xmin"}],"optio
nal":false,"name":"io.debezium.connector.postgresql.Source","field":"source"},{"type":
"string","optional":false,"field":"op"},{"type":"int64","optional":true,"field":"ts_ms
"},{"type":"struct","fields":[{"type":"string","optional":false,"field":"id"},{"type":
"int64","optional":false,"field":"total_order"},{"type":"int64","optional":false,"fiel
d":"data_collection_order"}],"optional":true,"field":"transaction"}],"optional":false,
"name":"PostgreSQL_14.public.orders.Envelope"},"payload":{"before":null,"after":{"orde
r_id":30,"created_by":"Santanu Kundu","order_date":1667917835107980},"source":{"versio
n":"1.9.6.Final","connector":"postgresql","name":"PostgreSQL_14","ts_ms":1667898035116
,"snapshot":"false","db":"mytest","sequence":"[null,\"24733560\"]","schema":"public","
table":"orders","txId":796,"lsn":24733560,"xmin":null},"op":"c","ts_ms":1667898035344,
"transaction":null}}
```


Figure 6: Data written in orders-source Kafka topic

12. **Points to be noted**: We had to change WAL_LEVEL to LOGICAL in configuration file as I used Debezium.io for making source connector. Then, you need to restart the server.

References:

1. Apache Akka, Apache Kafka, Scala Documentation page
2. Debezium.io PostgreSQL Source Kafka Connector documentation page.