l'École Nationale Supérieure d'Arts et Métiers de Rabat
Ingénierie Numérique en Data Science et Intelligence Artificielle
(INDIA)

Big Data Project Report

Subject :

PIPELINE FOR BITCOIN PRICES STREAMING AND STORING IN REAL TIME

University year : 2022/2023

Made by :
SOUAD Ayoub

Supervised by :
Pr. ELYADARI Mourad

Acknowledgements

At the end of this work, we would like to express our thanks and deep appreciation to gratitude to all those who have contributed in some way to its realization realization

We would like to express our sincere gratitude to our professor **ELYADARI Mourad** for their guidance and support throughout this project. We are deeply grateful for their expertise and insights, which were invaluable in helping us to develop and implement our real-time data streaming and processing platform.

We would also like to thank our classmates on INDIA class for their support and collaboration during this project. We are grateful for the opportunity to work with such a talented and dedicated group of individuals, and we look forward to continuing to learn and grow together in the future.

Abstract

In this project, we implemented a real-time data processing system using Binance Api, Kafka, Spark Streaming, and Hadoop. The system ingests data from the Binance API using Kafka, processes the data in real-time using Spark Streaming, and storing the data on HDFS. We we expect our data included real-time BTC/USDT prices. Our system was able to efficiently handle high volumes of data and facilitate the process of provide insights into trends and patterns in the data.

Overall, the project demonstrated the power and flexibility of using Kafka, Spark Streaming, and hadoop for real-time data processing. The system has potential applications in a variety of domains, including finance, e-commerce, and IoT. In the future, we plan to expand the system to include additional data sources and to explore advanced analysis techniques.

Table des matières

GENERAL INTRODUCTION

Cryptocurrencies, such as Bitcoin, have gained significant attention and adoption in recent years, due to their decentralized nature and potential for fast, low-cost transactions. However, the cryptocurrency market is highly dynamic and can be difficult to predict, making it challenging for investors and traders to make informed decisions.

To better understand and navigate the cryptocurrency market, it is important to have access to real-time data and analysis tools. In this report, we present a real-time data streaming and processing platform for analyzing cryptocurrency data. Our platform is designed to continuously stream data from the Binance API and process it using Spark Streaming. We implement the data processing logic using Scala, and store the processed data using Apache Hadoop.

The goal of this project is to provide a foundation for future work on real-time cryptocurrency analysis and prediction. By leveraging the power of real-time data streaming and processing, we hope to gain a deeper understanding of the dynamics of the cryptocurrency market and develop more accurate prediction models.

In the following sections of this report, we will describe the methods and tools used in our project, as well as the implementation and results of our real-time data streaming and processing platform.
This report consists of TWO chapters :

The first is about the architecture, the methods and tools installation and configuration.

The second is about implementation of streaming.

Chapitre 1

Methods and tools

1.1 Introduction

The goal of this project was to create a real-time data analytics application that could analyze cryptocurrency prices in near real-time. To achieve this goal, we utilized a number of tools and technologies, including Kafka, Spark, Hadoop, Maven and Scala.
In the following sections, we will describe the specific methods and tools we used in more detail, as well as the overall architecture of our project.

1.2 Project Pipeline

The overall architecture of our project can be divided into three main components : the data streaming layer, the data processing layer, and the data storage layer.

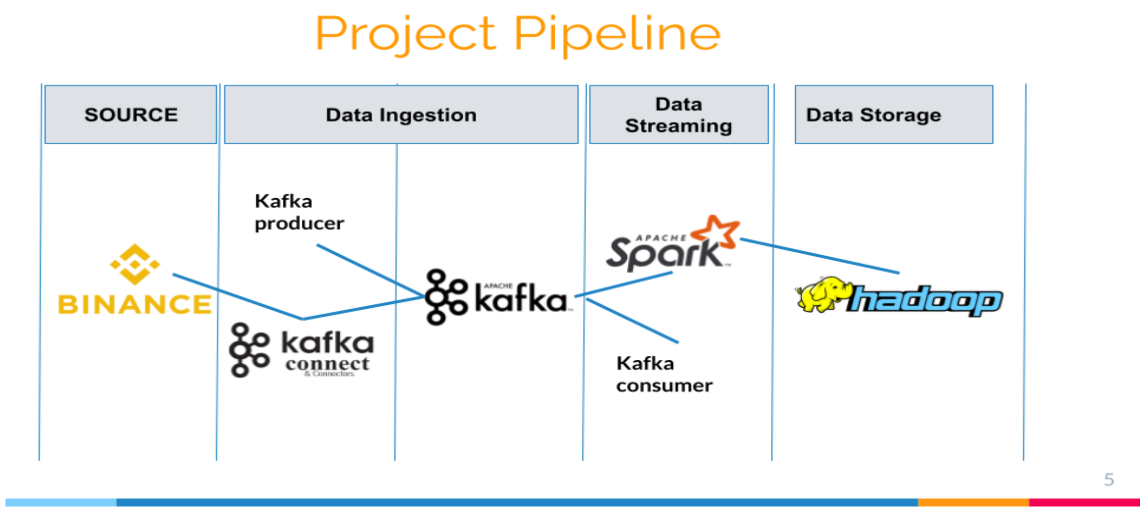


FIGURE 1.1 – Poject Pipeline

The data streaming layer was implemented using Kafka, which continuously pulled data from the Binance API and fed it into our application.

The data processing layer was implemented using Spark Streaming, which processed and analyzed the data in real-time. Scala was used as the programming language for this layer.

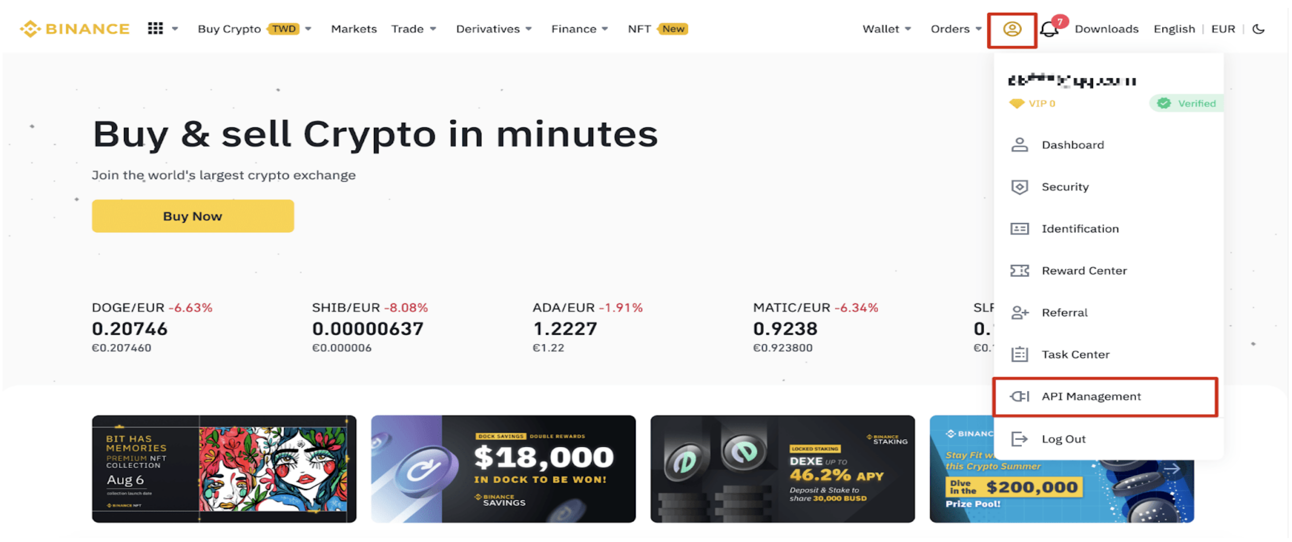
The data storage layer was implemented using Apache Hadoop, which provided a distributed file system for storing and processing large amounts of data.

These three components were integrated using a microservices architecture, with each component running as a separate service and communicating with the others via APIs. This allowed us to scale each component independently and handle high volumes of data efficiently.The overall architecture of our project can be divided into three main components : the data streaming layer, the data processing layer, and the data storage layer.

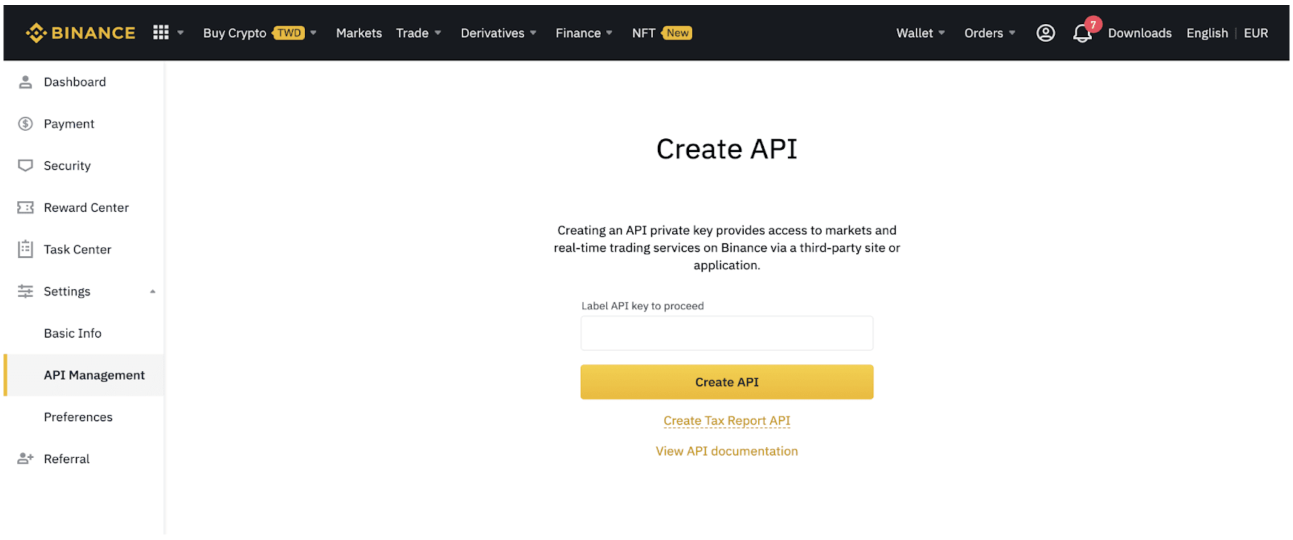
1.3 Tools installation

1.3.1 Create Binance API

Creating an API allows you to connect to Binance's servers via several programming languages. Data can be pulled from Binance and interacted with in external applications. You can view your current wallet and transaction data, make trades, and deposit and withdraw your funds in third-party programs. Creating an API is a simple process that can be completed in just 5 minutes.
How to create your own Binance API ?
1. After logging into your Binance account, click [API Management] from the user center icon.



2. Enter a label/name for your API key and click [Create API].
Security tip : Before creating an API, you need to enable two-factor authentication (2FA) on your account.

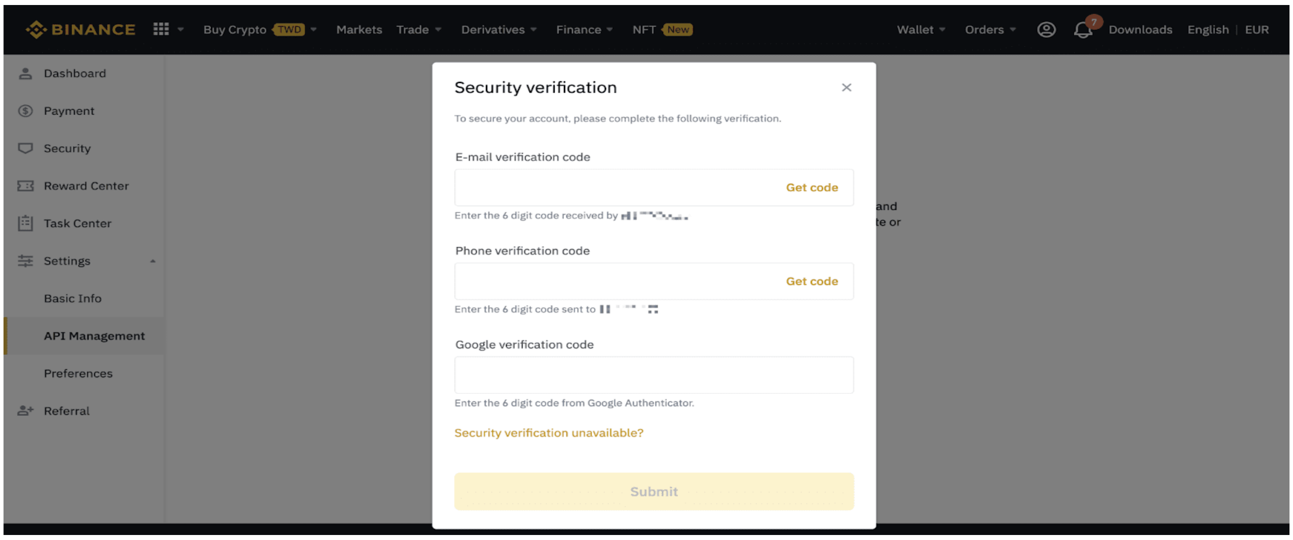


Create API

Creating an API private key provides access to markets and real-time trading services on Binance via a third-party site or application.

Label API key to proceed

3. Complete the security verification with your registered 2FA devices.



Security verification

To secure your account, please complete the following verification.

E-mail verification code

Enter the 6 digit code received by [redacted]

Phone verification code

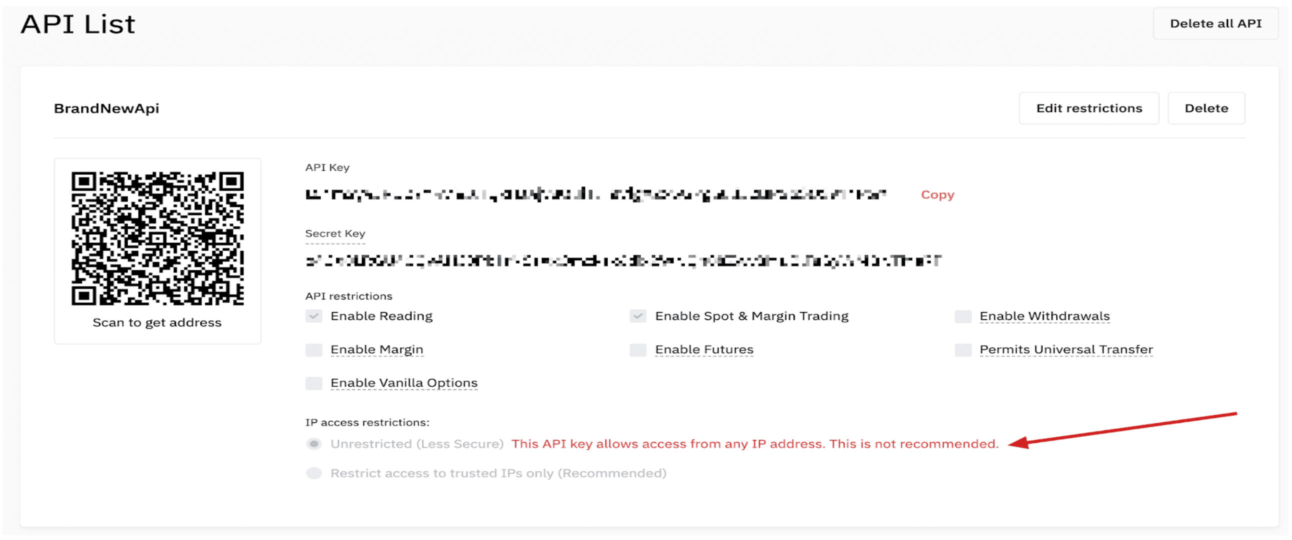
Enter the 6 digit code sent to [redacted]

Google verification code

Enter the 6 digit code from Google Authenticator.

4. Your API is now created. Please keep your Secret Key securely as will not be shown again. Do not share this key with anyone. If you forget your Secret Key, you will need to delete the API and create a new one.

Please take note of the IP access restrictions. We recommend choosing [Restrict access to trusted IPs only] for security reasons.


API List

Delete all API

BrandNewApi

Edit restrictions

Delete

API Key

[redacted]

Copy

Secret Key

[redacted]

API restrictions

- [✓] Enable Reading
- [✓] Enable Spot & Margin Trading
- [ ] Enable Withdrawals
- [ ] Enable Margin
- [ ] Enable Futures
- [ ] Permits Universal Transfer
- [ ] Enable Vanilla Options

IP access restrictions:

● Unrestricted (Less Secure) This API key allows access from any IP address. This is not recommended.

○ Restrict access to trusted IPs only (Recommended)

Note : Effective from 2021-08-09 03 :00 (UTC), only users who have completed intermediate verification can create new API keys.

1.3.2 Kafka installation

Apache Kafka is an open-source, distributed event streaming platform developed by the Apache Software Foundation. This is written in Scala and Java programming languages. You can install Kafka on any platform that supports Java programming language.
Step 1 – Installing Java :
We can run the Apache Kafka server on systems that support Java. So make sure, you have Java installed on your Ubuntu system.

Use the following commands to install OpenJDK on your Ubuntu system from the official repositories.

```
$sudo apt update
$sudo apt install default-jdk
```

Verify the current active Java version.

```
$java --version
```

Step 2 – Download Latest Apache Kafka
You can download the latest Apache Kafka binary files from its official download page. Alternativaly you can download kafka 2.13-3.3.1 with the below command.

```
$ wget https://downloads.apache.org/kafka/3.3.1/kafka_2.13-3.3.1.tgz
```

Then extract the downloaded archive file and place them under /usr/local/kafka directory.

```
$ tar xzf kafka_2.13-3.3.1.tgz
$ sudo mv kafka_2.13-3.3.1 /usr/local/kafka
```

Step 3 – Create Systemd Startup Scripts
Now, create systemd unit files for the Zookeeper and Kafka services. That will help you to start/stop the Kafka service in an easy way.

First, create a systemd unit file for Zookeeper :

```
$ sudo gedit /etc/systemd/system/zookeeper.service
```

And add the following content :

```
[Unit]
Description=Apache Zookeeper server
Documentation=http://zookeeper.apache.org
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
ExecStart=/usr/local/kafka/bin/zookeeper-server-start.sh
   /usr/local/kafka/config/zookeeper.properties
ExecStop=/usr/local/kafka/bin/zookeeper-server-stop.sh
Restart=on-abnormal

[Install]
WantedBy=multi-user.target
```

Save the file and close it.
Next, create a systemd unit file for the Kafka service :

```
$sudo gedit /etc/systemd/system/kafka.service
```

Add the below content. Make sure to set the correct JAVA HOME path as per the Java installed on your system.

```
[Unit]
Description=Apache Kafka Server
Documentation=http://kafka.apache.org/documentation.html
Requires=zookeeper.service

[Service]
Type=simple
Environment="JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64"
ExecStart=/usr/local/kafka/bin/kafka-server-start.sh
   /usr/local/kafka/config/server.properties
ExecStop=/usr/local/kafka/bin/kafka-server-stop.sh

[Install]
WantedBy=multi-user.target
```

Save the file and close.
Reload the systemd daemon to apply new changes.

```
$ sudo systemctl daemon-reload
```

This will reload all the systemd files in the system environment.
Step 4 – Start Zookeeper and Kafka Services
Let's start both services one by one. First, you need to start the ZooKeeper service and then start Kafka. Use the systemctl command to start a single-node ZooKeeper instance.

```
$ sudo systemctl start zookeeper
$ sudo systemctl start kafka
```

Verify both of the services status :

```
$ sudo systemctl status zookeeper
$ sudo systemctl status kafka
```

That's it. You have successfully installed the Apache Kafka server on Ubuntu 22.04 system. Next, we will create topics in the Kafka server.

1.3.3 Hadoop installation

Update the package manager index :

```
$ sudo apt-get update
```

Verify the Java installation :

```
$ java -version
```

Download the Hadoop binary files :

```
$ wget https://dlcdn.apache.org/hadoop/common/hadoop-3.3.4/hadoop-3.3.4.tar.gz
```

Extract the downloaded Hadoop tar file :

```
$ tar xzf hadoop-3.3.4.tar.gz
```

Move the extracted Hadoop folder to the desired location :

```
$ mv hadoop-3.3.4 /usr/local/
```

Set the environment variables for Hadoop :

```
export HADOOP\_HOME=/usr/local/hadoop-3.3.4
export PATH=$PATH:$HADOOP\_HOME/bin
```

Configure Hadoop by editing the following files :

1. $HADOOP_HOME/etc/hadoop/hadoop-env.sh

```
          <configuration>
              <property>
                <name>hadoop.tmp.dir</name>
                <value>/usr/local/tmpdata</value>
              </property>
              <property>
                <name>fs.default.name</name>
                <value>hdfs://127.0.0.1:9000</value>
              </property>
          </configuration>
```

2. $HADOOP_HOME/etc/hadoop/core-site.xml

```
          #  JAVA_HOME=/usr/java/testing hdfs dfs -ls
          export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
          export HADOOP_OS_TYPE=${HADOOP_OS_TYPE:-$(uname -s)}
```

3. $HADOOP_HOME/etc/hadoop/hdfs-site.xml

```
<configuration>
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
<property>
  <name>dfs.namenode.name.dir</name>
  <value>/usr/local/hadoop-3.3.4/data/namenode</value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
  <value>/usr/local/hadoop-3.3.4/data/datanode</value>
</property>
</configuration>
```

4. $HADOOP_HOME/etc/hadoop/yarn-site.xml

```
<configuration>
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
  <name>yarn.resourcemanager.hostname</name>
  <value>127.0.0.1</value>
</property>
<property>
  <name>yarn.acl.enable</name>
  <value>0</value>
</property>
<property>
  <name>yarn.nodemanager.env-whitelist</name>
  <value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,
  HADOOP_CONF_DIR,CLASSPATH_PERPEND_DISTCACHE,
  HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
</property>
</configuration>
```

Format the Hadoop filesystem :

```
$ hdfs namenode -format
```

Start the Hadoop services :

```
$ start-dfs.sh
$ start-yarn.sh
```

That's it ! You have successfully installed Hadoop 3.3.4 on your Ubuntu machine.

1.3.4 Intellij IDE - Spark - Scala Installation

install Intellij IDE

IntelliJ IDEA let us create a Maven project that contains spark dependencies so we don't need install Spark or Scala.
Install IntelliJ IDEA manually to manage the location of every instance and all the configuration files. For example, if you have a policy that requires specific install locations.
Download the community version .tar.gz. from https ://www.jetbrains.com/idea/download/
Extract the tarball to a directory that supports file execution.
For example, to extract it to the recommended /opt directory, run the following command :

```
$ sudo tar -xzf ideaIU-*.tar.gz -C /opt
```

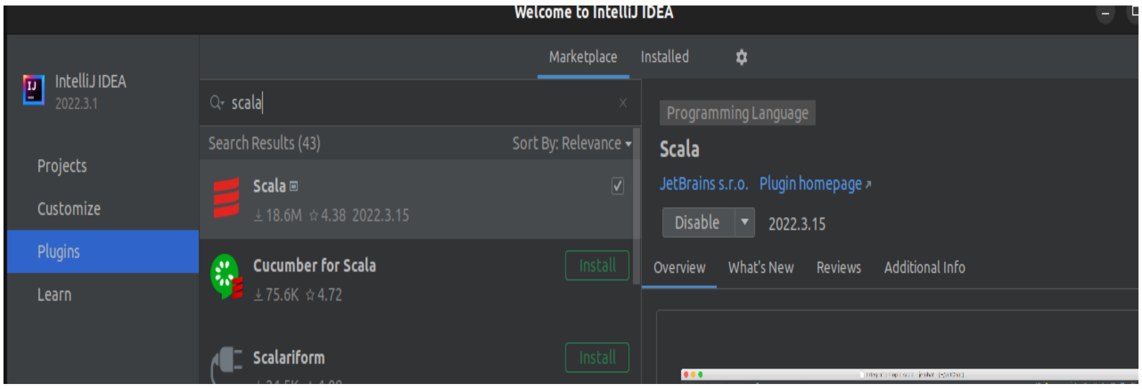
Execute the idea.sh script from the extracted directory to run IntelliJ IDEA.

```
$ ./idea.sh
```

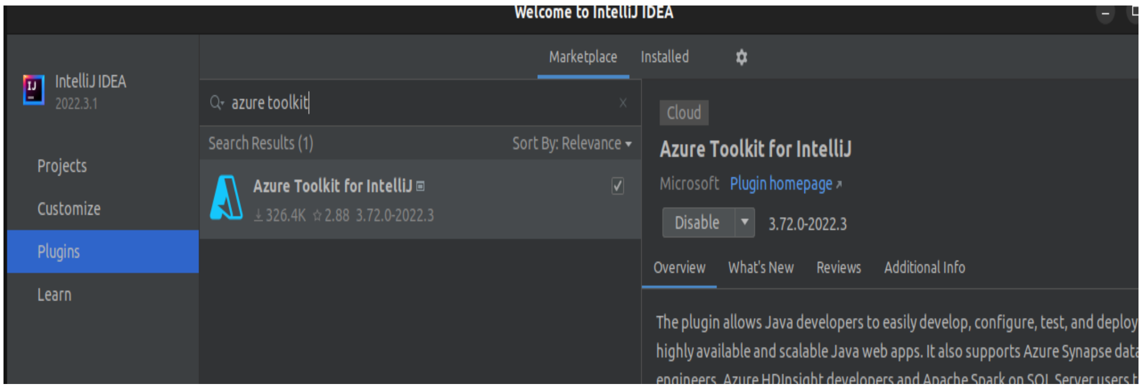
To create a desktop entry, do one of the following :
— On the Welcome screen, click Configure | Create Desktop Entry
— From the main menu, click Tools | Create Desktop Entry

install plugins

We have to install firstly Scala by going to Plugins icon and tap scala :



we need also to install azuretolkit that offer start apache spark project by going to Plugins icon and tap azuretolkit :

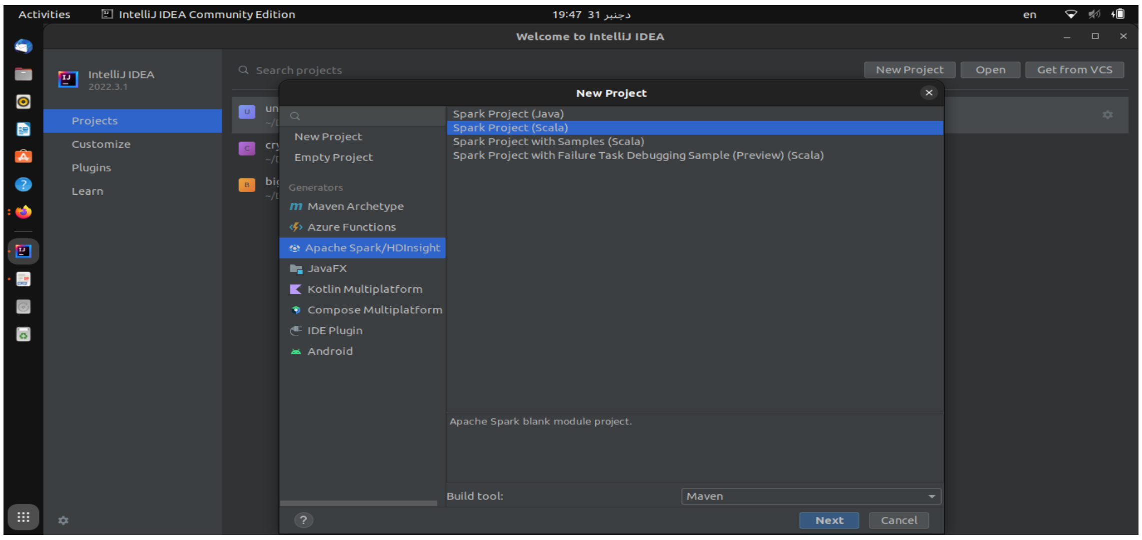


The plugin allows Java developers to easily develop, configure, test, and deploy
highly available and scalable Java web apps. It also supports Azure Synapse data
engineers, Azure HDInsight developers and Apache Spark on SQL Server users t

Create Maven Project

To create a Maven project for Scala in IntelliJ IDEA Community Edition :

1 - Click new project then choose Apache Spark / HDInsight then select spark project (Scala) :



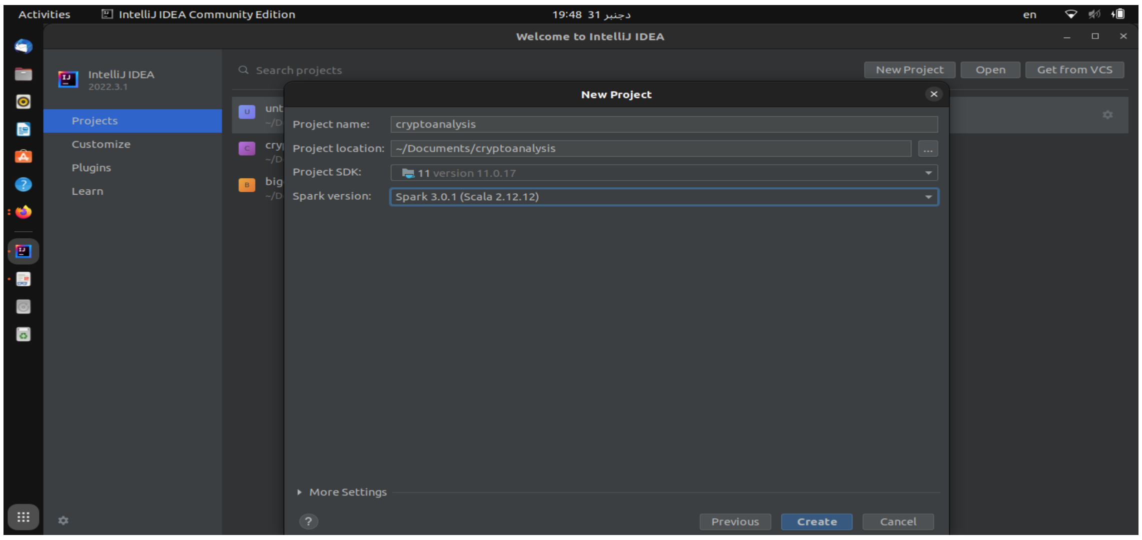
New Project

Spark Project (Java)
Spark Project (Scala)
Spark Project with Samples (Scala)
Spark Project with Failure Task Debugging Sample (Preview) (Scala)

Apache Spark blank module project.

Build tool: Maven

2 - In the "Project SDK" field, select the JDK installation directory. If the JDK is not installed, click "New" and follow the prompts to install it, In the spark version should be 3.0.1 with scala 2.12.12 and Click "Finish" to create the Maven project :

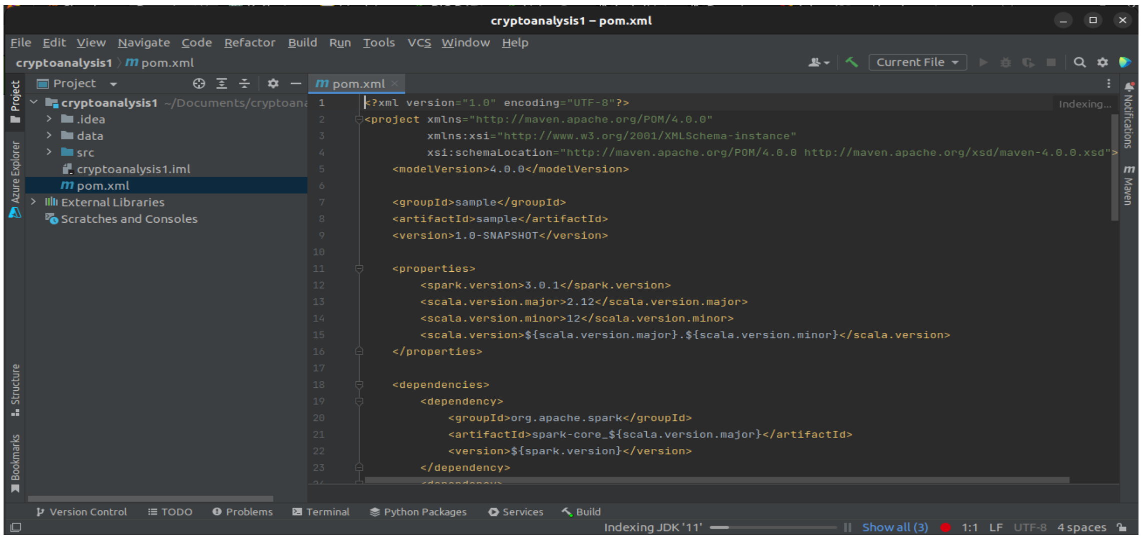


New Project

Project name: cryptoanalysis
Project location: ~/Documents/cryptoanalysis
Project SDK: 11 version 11.0.17
Spark version: Spark 3.0.1 (Scala 2.12.12)

The Maven project structure will be generated, including the pom.xml file which manages the project dependencies.

3 - To add Scala support to the project, you can edit the pom.xml file and include the Scala dependency.



```
<?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>sample</groupId>
    <artifactId>sample</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <spark.version>3.0.1</spark.version>
        <scala.version.major>2.12</scala.version.major>
        <scala.version.minor>12</scala.version.minor>
        <scala.version>${scala.version.major}.${scala.version.minor}</scala.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
```

replace the pom.xml configuration by next :

```
<?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>sample</groupId>
    <artifactId>sample</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <spark.version>3.3.1</spark.version>
        <scala.version.major>2.13</scala.version.major>
        <scala.version.minor>10</scala.version.minor>
        <scala.version>${scala.version.major}.${scala.version.minor}</scala.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-mllib_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql-kafka-0-10_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka-0-10_${scala.version.major}</artifactId>
            <version>${spark.version}</version>
        </dependency>
        <dependency>
            <groupId>org.jmockit</groupId>
            <artifactId>jmockit</artifactId>
            <version>1.49</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>4.7.2</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
                            <goal>testCompile</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <scalaVersion>${scala.version}</scalaVersion>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.10.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

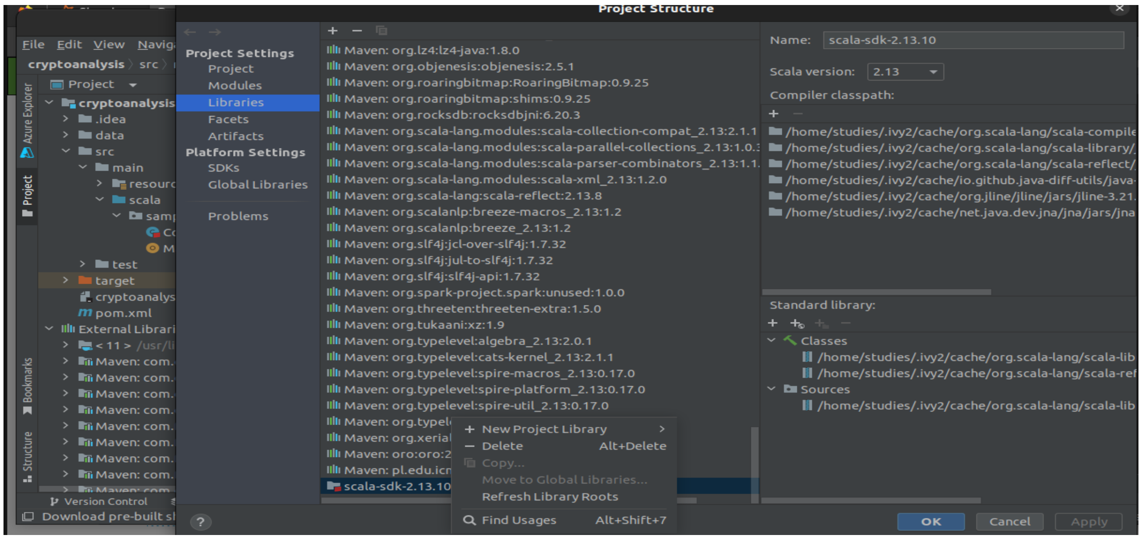
The pom.xml file defines the project's dependencies, build settings, and other information required for building the project.

The dependencies section includes the following dependencies :
- spark-core_$scala.version.major : Spark Core Library
- spark-sql_$scala.version.major : Spark SQL Library
- spark-streaming_$scala.version.major : Spark Streaming Library
- spark-mllib_$scala.version.major : Spark Machine Learning Library
- spark-sql-kafka-0-10_$scala.version.major : Spark Kafka 0.10 Integration
- spark-streaming-kafka-0-10_$scala.version.major : Spark Streaming Kafka 0.10 Integration
- jmockit : JMockit Library for testing

The build section includes the scala-maven-plugin, which is used to compile Scala code, and the maven-compiler-plugin, which is used to compile Java code.

The properties section defines the versions of Spark, Scala, and JMockit to be used in the project.

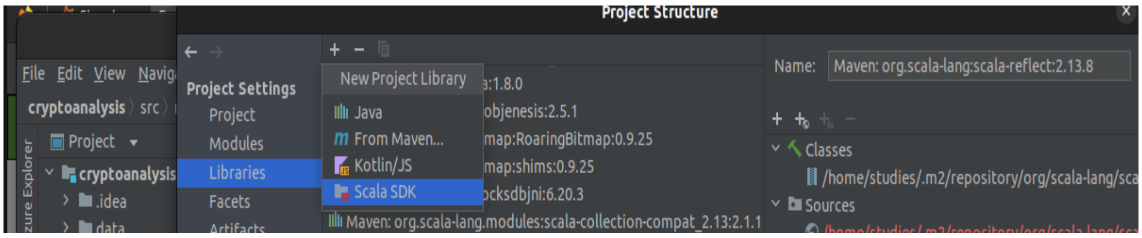
The project is configured to use Java 1.8 as the source and target for compilation.
Note : your Libraries can contain more than scala version or the incorrect one, so you can go to the "File" menu and select "Project Structure". In the "Project Structure" window, go to the "Libraries" tab and you will see a list of all the libraries that are included in the project and you can delete no desired Scala incorrect version for example :



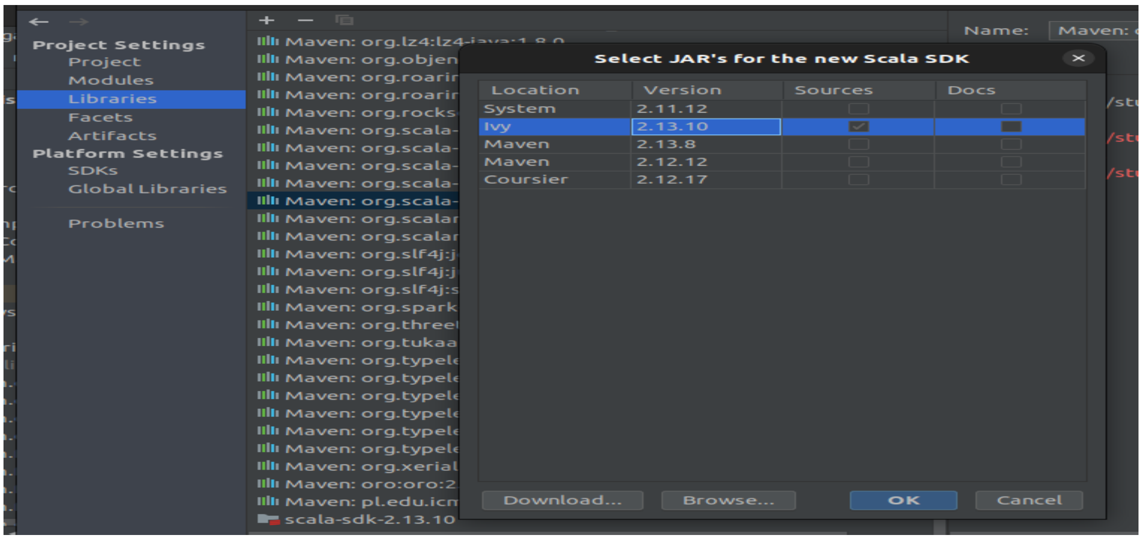
```
Maven: org.lz4:lz4-java:1.8.0
Maven: org.objenesis:objenesis:2.5.1
Maven: org.roaringbitmap:RoaringBitmap:0.9.25
Maven: org.roaringbitmap:shims:0.9.25
Maven: org.rocksdb:rocksdbjni:6.20.3
Maven: org.scala-lang.modules:scala-collection-compat_2.13:2.1.1
Maven: org.scala-lang.modules:scala-parallel-collections_2.13:1.0.3
Maven: org.scala-lang.modules:scala-parser-combinators_2.13:1.1.
Maven: org.scala-lang.modules:scala-xml_2.13:1.2.0
Maven: org.scala-lang:scala-reflect:2.13.8
Maven: org.scalanlp:breeze-macros_2.13:1.2
Maven: org.scalanlp:breeze_2.13:1.2
Maven: org.slf4j:jcl-over-slf4j:1.7.32
Maven: org.slf4j:jul-to-slf4j:1.7.32
Maven: org.slf4j:slf4j-api:1.7.32
Maven: org.spark-project.spark:unused:1.0.0
Maven: org.threeten:threeten-extra:1.5.0
Maven: org.tukaani:xz:1.9
Maven: org.typelevel:algebra_2.13:2.0.1
Maven: org.typelevel:cats-kernel_2.13:2.1.1
Maven: org.typelevel:spire-macros_2.13:0.17.0
Maven: org.typelevel:spire-platform_2.13:0.17.0
Maven: org.typelevel:spire-util_2.13:0.17.0
Maven: org.typel
Maven: org.xerial
Maven: oro:oro:2
Maven: pl.edu.icn
scala-sdk-2.13.10

Name: scala-sdk-2.13.10
Scala version: 2.13
Compiler classpath:
/home/studies/.ivy2/cache/org.scala-lang/scala-compile
/home/studies/.ivy2/cache/org.scala-lang/scala-library/
/home/studies/.ivy2/cache/org.scala-lang/scala-reflect/
/home/studies/.ivy2/cache/io.github.java-diff-utils/java-
/home/studies/.ivy2/cache/org.jline/jline/jars/jline-3.21.
/home/studies/.ivy2/cache/net.java.dev.jna/jna/jars/jna
Standard library:
Classes
    /home/studies/.ivy2/cache/org.scala-lang/scala-lib
    /home/studies/.ivy2/cache/org.scala-lang/scala-ref
Sources
    /home/studies/.ivy2/cache/org.scala-lang/scala-lib
```

In other hand you shoud install the correct one by selecting the plus icon in the left and select Scala SDK and download it if not exist and choose the version after by clicking on it and click OK :



```
Name: Maven: org.scala-lang:scala-reflect:2.13.8
Classes
    /home/studies/.m2/repository/org/scala-lang/sca
Sources
```



Select JAR's for the new Scala SDK

Location	Version	Sources	Docs
System	2.11.12	☐	☐
Ivy	2.13.10	✓	■
Maven	2.13.8	☐	☐
Maven	2.12.12	☐	☐
Coursier	2.12.17	☐	☐

Make sure that all the dependencies cited on pom.xml are installed to guarantied the execution

1.4 Conclusion

In this chapter, we described the methods and tools we used in our project to stream and analyze cryptocurrency data in real-time. We used Kafka as the data streaming platform, Spark Streaming as the data processing platform, Scala as the programming language, maven as project dependencies configure and Intellij community as IDE, and Hadoop as data store ecosystem

We also discussed the process of installing and configuring these tools and technologies, and highlighted some of the challenges and considerations we faced along the way.

Overall, these methods and tools allowed us to effectively stream and analyze large volumes of data in real-time, and were well-suited to the goals and objectives of our project.

In the next chapter, we will describe the implementation of the data streaming and analysis layers in more detail and present the results of our analysis.

Chapitre 2

Real-time streaming and storing

2.1 Introduction

Know, we have install and setup our environment,so the following sections, we will describe the specific steps we took to implement the data streaming and analysis layers, and present the results of our analysis.

2.2 Data ingestion

After we obtain the Binance API key and secret we can use them by a kafka producer to ingest data from Binance API in real time. But let's see what is kafka Producer firstly.

Producers

Kafka is a distributed streaming platform that is commonly used for building real-time data pipelines and streaming applications. One important component of Kafka is the producer, which is responsible for producing data to Kafka topics.

Producers can be implemented in a variety of programming languages, including Python, Java, and C++. They are typically used to stream data from external sources, such as databases, APIs, or log files, into Kafka topics, where it can be processed and analyzed by other Kafka consumers.

In the following section, we will describe an example of a Kafka producer implemented in Python that streams data from the Binance API and produces it to a Kafka topic.

The following code is an example of a Kafka producer written in Python that streams data from the Binance API and produces it to a Kafka topic :

```python
import requests
from kafka import KafkaProducer
import json
import time

# Set up a Kafka producer
producer = KafkaProducer(bootstrap_servers='localhost:9092')
```

```
# Set up the Binance API client
binance_api_key =
    'UHp7lhuAGxeIRMicXvT0fWK0ekfYkW3ty5xxxxx4TMq1a0KwiUCwKZIZS3V5B21lv'
binance_api_secret =
    'EOYmj3AQ7lIinDfYL4ry9a0GgiLncpK1gFpxxxxxEaxJAZBeY6i2xPkbix8FCrEJN'

# Set up an infinite loop to continuously fetch data from the Binance API and
    produce it to Kafka
while True:
  # Make a request to the Binance API and retrieve the JSON data
  response =
      requests.get("https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT",
  headers={'X-MBX-APIKEY': binance_api_key})


  # Produce the data to the Kafka topic
   data = response.json()
  data["price"]=float(data["price"])
  data=json.dumps(data).encode('utf-8')
  producer.send('binance-topic', value= data)
  producer.flush()
  # sleep for 5 seconds
  time.sleep(5)
```

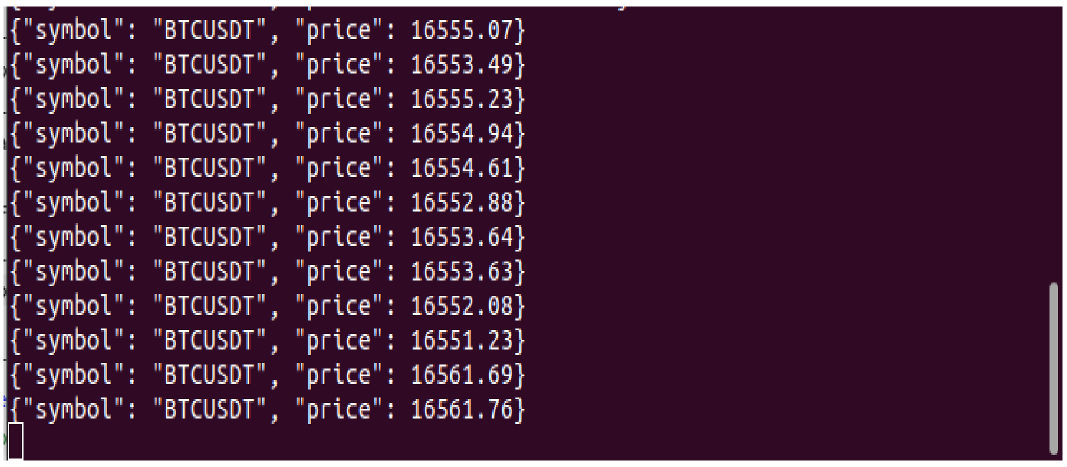
You can execute it by Python3.

If you confront the problem of **no module named kafka** To install the kafka module, you can use pip, the Python package manager :
pip install kafka

Run a consumer from terminal :

```
$ kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
    binance-topic --from-beginning
```

you can see the messages in the topic :

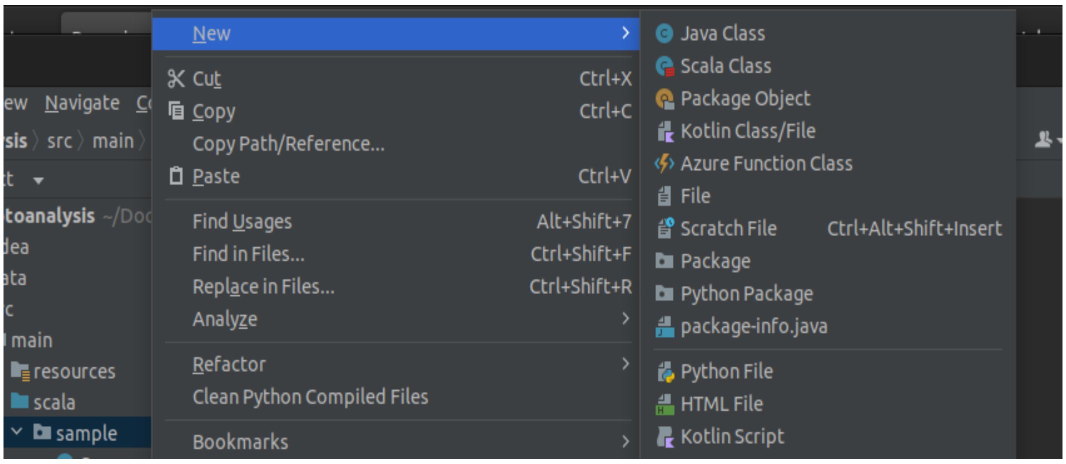


```
{"symbol": "BTCUSDT", "price": 16555.07}
{"symbol": "BTCUSDT", "price": 16553.49}
{"symbol": "BTCUSDT", "price": 16555.23}
{"symbol": "BTCUSDT", "price": 16554.94}
{"symbol": "BTCUSDT", "price": 16554.61}
{"symbol": "BTCUSDT", "price": 16552.88}
{"symbol": "BTCUSDT", "price": 16553.64}
{"symbol": "BTCUSDT", "price": 16553.63}
{"symbol": "BTCUSDT", "price": 16552.08}
{"symbol": "BTCUSDT", "price": 16551.23}
{"symbol": "BTCUSDT", "price": 16561.69}
{"symbol": "BTCUSDT", "price": 16561.76}
```

2.3 Data Streaming and storing

Create Scala class and object

We need to create Scala class where we will define our spark session and read stream from the binance-topic topic and save data on HDFS and make analysis. Click right on sample folder localized on scala folder and go to new where you can create Scala class or Object. tap the name consumer for class file and main for object file.



We will write a consumer with real time streaming using spark read stream and save the dataframes on hdfs as csv files, but what is a consumer :

Consumers

In Kafka, a consumer is a client that reads data from one or more Kafka topics. Consumers can be implemented in a variety of programming languages, including Python, Java, and C++.

Consumers subscribe to Kafka topics and consume the data produced to those topics by producers. They can process and analyze the data in real-time, or store it for later use.

Kafka consumers are typically used in conjunction with producers as part of a larger data streaming application. Producers stream data into Kafka topics, and consumers read and process the data from those topics. This allows for the efficient and scalable processing of large volumes of data in real-time.

Kafka consumers can be configured to read data from a single topic, or from multiple topics at the same time. They can also be configured to read data from a specific partition of a topic, or from all partitions of a topic.

Here is our consumer writed by scala :

```scala
import org.apache.spark.sql
import org.apache.spark.sql.{Encoders, SparkSession}
import org.apache.spark.sql.functions.{col, from_json}
import org.apache.spark.sql.streaming.Trigger
import org.apache.spark.sql.types._

// Consumer class for reading data from a Kafka topic and writing it to
    console or a CSV file
```

```
class Consumer() {

  // DataFrame to hold the data read from Kafka
  var df: sql.DataFrame = _

  // Schema for the data read from Kafka
  val schema = new StructType()
    .add("symbol", StringType, false)
    .add("price", DoubleType, false)

  // Create a SparkSession
  val spark = SparkSession
    .builder
    .appName("sparkConsumer")
    .config("spark.master", "local")
    .getOrCreate()

  // Set log level to ERROR to reduce verbosity
  spark.sparkContext.setLogLevel("ERROR")

  // Import implicit conversions to enable .select and .writeStream methods on
      DataFrames
  import spark.implicits._

  // Method for loading data from a Kafka topic
  def loadKafka(topic: String): Unit = {
    df = spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("auto.offset.reset", "latest")
      .option("value.deserializer", "StringDeserializer")
      .option("subscribe", topic)
      .load()
      .select(from_json(col("value").cast("string"),
         schema).alias("parsed_value"))
      .select(col("parsed_value.*"))

    // Print the schema of the DataFrame to the console
    df.printSchema()
  }

  // Method for writing data to the console
  def writeData(): Unit = {
    df.writeStream
      .format("console")
      .outputMode("append")
      .start()
      .awaitTermination()
  }

  // Method for saving data to a CSV file on HDFS
    df.writeStream
```

```
      .format("csv")
      .option("timestampFormat", "yyyy-MM-dd hh:mm:ss.SSSSSSS")
      .option("checkpointLocation", "hdfs://localhost:9000/binance/checkpoint")
      .option("path", "hdfs://localhost:9000/binance/data")
      .outputMode("append")
      .start()
      .awaitTermination()
  }
}
```

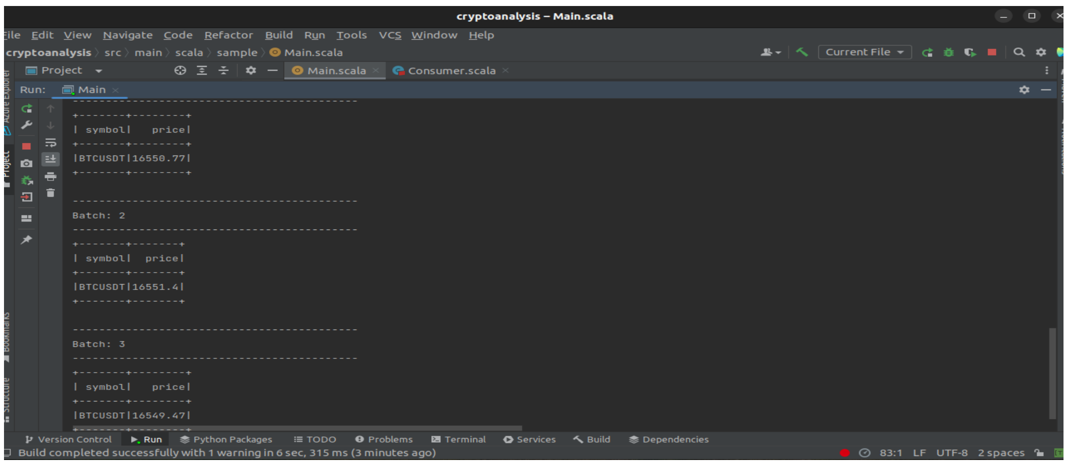
we write instance for the class by the object main :

```
package sample

object Main {
  def main(args: Array[String]) = {
    println("Hello, world")
    //kafka
    val kafka = new Consumer()
    kafka.loadKafka("binance-topic")
    kafka.writeData()
    println("Done")

  }
}
```

by result a batch every 5 seconds showed on console and saved on HDFS.


cryptoanalysis – Main.scala
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
cryptoanalysis 〉 src 〉 main 〉 scala 〉 sample 〉 Main.scala
Current File
Project
Main.scala
Consumer.scala
Run: Main
Version Control
Run
Python Packages
TODO
Problems
Terminal
Services
Build
Dependencies
83:1 LF UTF-8 2 spaces

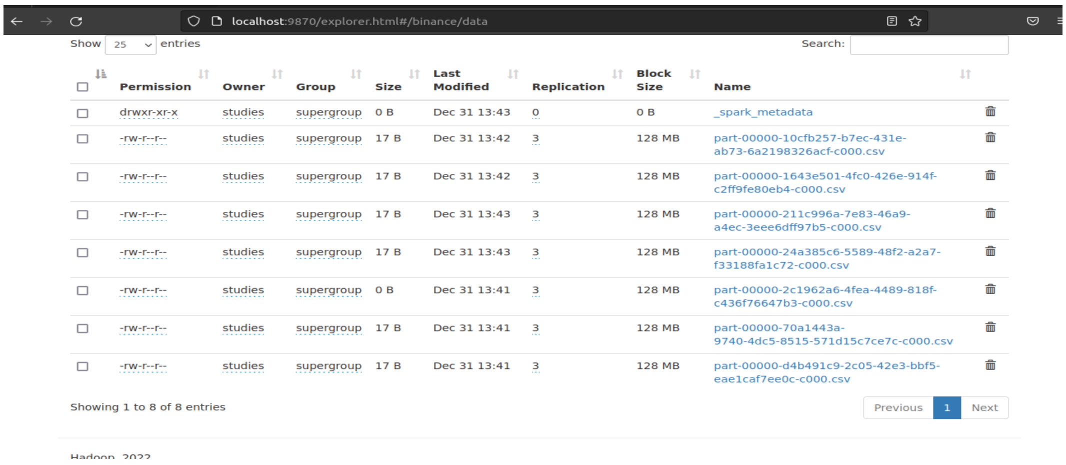
```
-------------------------------------------
+-------+--------+
| symbol|   price|
+-------+--------+
|BTCUSDT|16550.77|
+-------+--------+

-------------------------------------------
Batch: 2
-------------------------------------------
+-------+-------+
| symbol|  price|
+-------+-------+
|BTCUSDT|16551.4|
+-------+-------+

-------------------------------------------
Batch: 3
-------------------------------------------
+-------+--------+
| symbol|   price|
+-------+--------+
|BTCUSDT|16549.47|
+-------+--------+
```

Build completed successfully with 1 warning in 6 sec, 315 ms (3 minutes ago)

FIGURE 2.1 – real streaming on console


localhost:9870/explorer.html#/binance/data

Show 25 entries

Search:

	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
	drwxr-xr-x	studies	supergroup	0 B	Dec 31 13:43	0	0 B	_spark_metadata
	-rw-r--r--	studies	supergroup	17 B	Dec 31 13:42	3	128 MB	part-00000-10cfb257-b7ec-431e-ab73-6a2198326acf-c000.csv
	-rw-r--r--	studies	supergroup	17 B	Dec 31 13:42	3	128 MB	part-00000-1643e501-4fc0-426e-914f-c2ff9fe80eb4-c000.csv
	-rw-r--r--	studies	supergroup	17 B	Dec 31 13:43	3	128 MB	part-00000-211c996a-7e83-46a9-a4ec-3eee6dff97b5-c000.csv
	-rw-r--r--	studies	supergroup	17 B	Dec 31 13:43	3	128 MB	part-00000-24a385c6-5589-48f2-a2a7-f33188fa1c72-c000.csv
	-rw-r--r--	studies	supergroup	0 B	Dec 31 13:41	3	128 MB	part-00000-2c1962a6-4fea-4489-818f-c436f76647b3-c000.csv
	-rw-r--r--	studies	supergroup	17 B	Dec 31 13:41	3	128 MB	part-00000-70a1443a-9740-4dc5-8515-571d15c7ce7c-c000.csv
	-rw-r--r--	studies	supergroup	17 B	Dec 31 13:41	3	128 MB	part-00000-d4b491c9-2c05-42e3-bbf5-eae1caf7ee0c-c000.csv

Showing 1 to 8 of 8 entries

Hadoop, 2022

FIGURE 2.2 – real streaming on hdfs

2.4 Conclusion

In this chapter, we described the implementation of the data streaming and analysis layers of our project, which were designed to stream and process cryptocurrency data in real-time. We used Kafka to continuously stream data from the Binance API into our application, and Spark Streaming to process and analyze the data. We implemented the data processing logic using Scala, and used Apache Hadoop to store the processed data.

Although we did not perform any analysis in this project, the data streaming and processing techniques we implemented provide a strong foundation for future work. In future projects, we could use these techniques to analyze and understand the dynamics of the cryptocurrency market in more depth.

CONCLUSION

In this report, we presented a real-time data streaming and processing platform for analyzing cryptocurrency data. We used Kafka to stream data from the Binance API into our application, and Spark Streaming to process and analyze the data. We implemented the data processing logic using Scala, and used Apache Hadoop to store the processed data.

Although we did not perform any analysis in this project, the data streaming and processing techniques we implemented provide a strong foundation for future work. In particular, we hope to use these techniques to build a strong prediction model that can accurately predict cryptocurrency prices in real-time. To achieve this goal, we plan to explore the use of PySpark and the MLlib library to generate predictions and perform time series forecasting.

Overall, this project demonstrates the power of real-time data streaming and processing for understanding complex systems, and highlights the potential for future work in this area. We are confident that with the right tools and techniques, we can build a strong model for real-time price prediction that will help us to better understand and navigate the dynamic cryptocurrency market.

Bibliographie

[1] "Kafka : The Definitive Guide" by Neha Narkhede, Gwen Shapira, and Todd Palino.

[2] Spark : The Definitive Guide" by Matei Zaharia, Bill Chambers, and Matei Zaharia

[3] "Scala for Data Science" by Pascal Bugnion

[4] "Hadoop : The Definitive Guide" by Tom White

[5] "Real-Time Big Data Analytics : Emerging Architecture" by Rashmi Raghu and Mark R. Hauswirth

[6] "Streaming Data : Understanding the Real-Time Pipeline" by Tyler Akidau and Slava Chernyak

[7] "Real-Time Data Processing with Apache Spark" by Holden Karau and Rachel Warren

[8] Kafka Documentation

[9] Hadoop Documentation

[10] Spark Documentation

[11] Scala Documentation