به نام خدا

داکیومنت پروژه Stock-Stream-Processing

درس :

سیستم‌های توزیع شده

استاد:

دکتر محسن شریفی

دانشجویان:

سید محمدامین حائری - 403000000

علیرضا نظری - 403724023

فهرست:

تعریف صورت مسئله 2

اجزای اصلی پروژه 2

د ریافت داده Data ingestion 2

سرویس پردازش استریم Stream process 2

سرویس ذخیره سازی 2

سیستم اطلاع رسانی Notification 2

نمایش داده Dashboard 2

کارهای آینده 2

1- تعریف صورت مسئله

ما باید یک **سیستم توزیع‌شده برای تحلیل مالی و پیشنهاد خرید و فروش به صورت Real time** طراحی و پیاده‌سازی کنیم.
هدف این سیستم آن است که کاربر با استفاده از پیشنهاداتی که از سیستم ما دریافت میکند به خرید و یا فروش سهم رمز ارز بیت کوین Bitcoin بپردازد.
این سیستم قیمت ورودی رمز ارز بیت کوین را دریافت کرده و در لحظه با استفاده از تحلیل اندیکاتورهای معاملاتی تعریف شده در سیستم به کاربر پیشنهاد خرید و یا فروش این رمز ارز را میدهد.
برای پردازش و تحلیل داده‌های قیمتی از معماری توزیع شده و پردازش داده‌های استریمی استفاده شده و به عنوان داده ورودی از API صرافی alphavantage استفاده شده است.

2- اجزای اصلی سیستم

بیاین بخش‌های مختلف سیستم رو با جزئیات بیشتر بررسی کنیم:

د ریافت داده Data ingestion:
در اینجا ما یک API به صرافی alphavantage زده‌ایم که قیمت بیت‌کوین را به صورت استریم و با فاصله زمانی 1 ثانیه دریافت می‌کند. این داده‌ها شامل قیمت فعلی بیت‌کوین هستند.
کار اصلی این سرویس این است که داده‌ها را اعتبارسنجی کند (مثلاً مطمئن شود که فیلدهای ضروری مثل `price` و `timestamp` وجود دارند) و سپس آن‌ها را به سرویس پردازش جریان ارسال کند. این سرویس نقش یک دروازه ورودی را بازی می‌کند و باید بتواند هم داده‌های شبیه‌سازی‌شده و هم داده‌های واقعی از API صرافی alphavantage را مدیریت کند.

Name	Last commit message
..	
Dockerfile	feat: simple producer to test the kafka broker
data-ingestor-deployment.yaml	refactor: updating the data ingestor to send real data
data-ingestor.py	refactor: updating the data ingestor to send real data
requirements.txt	refactor: updating the data ingestor to send real data

2_1 - Data ingestor.py

فایل data-ingestor.py:

وارد کردن ماژول‌ها:

```
import json
import os
import time
from datetime import datetime
from kafka import
KafkaProducer
```

- **json**: برای سریالایز کردن داده‌ها به فرمت JSON.
- **os**: برای خواندن متغیرهای محیطی.
- **time**: برای کار با زمان‌بندی و تاخیر بین ارسال پیام‌ها.
- **datetime**: برای ثبت زمان دقیق در پیام‌ها.
- **KafkaProducer**: از کتابخانه kafka-python برای تعامل با Kafka استفاده می‌شود.

تابع create_producer:

```
def create_producer():
    return KafkaProducer(
        bootstrap_servers=[
            os.getenv("KAFKA_BOOTSTRAP_SERVERS", "kafka-0.kafka-headless:9092")
        value_serializer=lambda v: json.dumps(v).encode("utf-8"),
    )
```

- **هدف**: ایجاد یک تولیدکننده Kafka.
- **bootstrap_servers**: آدرس سرورهای Kafka از متغیر محیطی KAFKA_BOOTSTRAP_SERVERS دریافت می‌شود. مقدار پیش‌فرض: kafka-0.kafka-headless:9092.
- **value_serializer**: داده‌ها به فرمت JSON سریالایز و سپس به بایت تبدیل می‌شوند.

تابع send_message:

```
def send_message():
    producer = create_producer()
```

یک تولیدکننده Kafka ایجاد می‌شود.

```python
while True:
    try:
        message = {
            "timestamp": datetime.now().isoformat(),
            "message": "Hello from Python Producer!",
            "id": int(time.time()),
        }
```

یک حلقه بی‌نهایت اجرا می‌شود.

- **پیام نمونه:**
  - timestamp: زمان فعلی.
  - message: یک پیام متنی ساده.
  - id: شناسه پیام که بر اساس زمان یونیکس ساخته شده.

```python
        future = producer.send(
            topic=os.getenv("KAFKA_TOPIC", "test-topic"),
value=message)
```

- پیام به تاپیکی ارسال می‌شود که از متغیر محیطی KAFKA_TOPIC خوانده شده است. مقدار پیش‌فرض:

```
record_metadata = future.get(timeout=10)
```

- منتظر می‌ماند تا پیام ارسال شود و متادیتای رکورد (مانند تاپیک، پارتیشن و آفست) بازیابی شود.

```
        print(
            f"Message sent to topic {record_metadata.topic} partition {record_metadata.partition} offset
{record_metadata.offset}"
        )

        time.sleep(float(os.getenv("MESSAGE_INTERVAL", "5")))
```

- قبل از ارسال پیام بعدی، به اندازه مقدار MESSAGE_INTERVAL (پیش‌فرض 5 ثانیه) صبر می‌کند.

بخش مدیریت استثناها:

```
    except Exception as e:
        print(f"Error producing message:
{str(e)}")  time.sleep(5)
```

- اگر خطایی رخ دهد، پیام خطا چاپ شده و برنامه 5 ثانیه منتظر می‌ماند.

اجرای اصلی:

```
if __name__ ==
"__main__":
    send_message()
```

- تابع send_message هنگام اجرای مستقیم فایل فراخوانی می‌شود.

فایل requirements.txt:

```
kafka-python==2.0.2
```

- این فایل مشخص می‌کند که برای اجرای برنامه نیاز به کتابخانه kafka-python نسخه 2.0.2 دارید.

فایل data-ingestor-deployment.yaml:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: data-ingestor
spec:
  replicas: 1
  selector:
    matchLabels:
      app: data-ingestor
```

- یک دیپلویمنت Kubernetes تعریف می‌کند.
- **replicas: 1**: تنها یک نمونه از برنامه اجرا می‌شود.

```yaml
  template:
    metadata:
      labels:
        app: data-ingestor
    spec:
      containers:
        - name: data-ingestor
          image: data-ingestor:latest
          imagePullPolicy:
IfNotPresent
```

- کانتینر برنامه:
  - نام: data-ingestor.
  - تصویر: data-ingestor:latest.

```yaml
        env:
          - name: KAFKA_BOOTSTRAP_SERVERS
            value: "kafka-0.kafka-headless:9092"
          - name: KAFKA_TOPIC
            value: "test-topic"
          - name: MESSAGE_INTERVAL
            value: "2"
```

- متغیرهای محیطی:
  - KAFKA_BOOTSTRAP_SERVERS: آدرس سرور Kafka.

- KAFKA_TOPIC: نام تاپیک.
- MESSAGE_INTERVAL: فاصله زمانی بین پیام‌ها (در اینجا 2 ثانیه).

فایل Dockerfile (آپلود شده):

در این فایل Docker image کانفیگ خواسته شده تولید میشود .

3. **سرویس پردازش جریان (Stream Processing Service)**:
این سرویس قلب سیستم هست! داده‌ها رو در لحظه پردازش می‌کنه و اندیکاتورهای معاملاتی مثل **میانگین متحرک (Moving Average)**، **میانگین متحرک نمایی (Exponential Moving Average)** و **شاخص قدرت نسبی (RSI)** رو محاسبه می‌کنه. این سرویس باید خیلی سریع باشه چون داده‌ها به صورت بلادرنگ وارد می‌شن و باید فوراً تحلیل بشن.

در اینجا دو فایل YAML دارید که برای تنظیم و دیپلوی Kafka و Zookeeper در Kubernetes استفاده می‌شوند. حالا خط به خط این فایل‌ها را بررسی می‌کنیم:

فایل kafka-deployment.yaml

Service: Kafka Headless

```
apiVersion: v1
kind: Service
metadata:
  name: kafka-headless
  labels:
    app: kafka
```

- **apiVersion: v1**: این سرویس با API نسخه 1 از Kubernetes کار می‌کند.
- **kind: Service**: این منبع یک سرویس است.
- **metadata**:
  - **name: kafka-headless**: نام سرویس Kafka است.
  - **labels: app: kafka**: برچسبی برای شناسایی این سرویس به‌عنوان بخشی از Kafka.

```
spec:
  ports:
    - port: 9092
      name: kafka
    - port: 9093
      name: kafka-internal
  clusterIP: None
  selector:
    app: kafka
```

- **spec**: مشخصات سرویس.
  - **ports**: دو پورت تعریف شده:
    - پورت 9092 برای دسترسی خارجی Kafka.
    - پورت 9093 برای ارتباط داخلی بین بروکرها.
  - **clusterIP: None**: این سرویس headless است، به این معنا که DNS مستقیم برای پادها ایجاد می‌کند.
  - **selector: app: kafka**: این سرویس پادهایی که برچسب app: kafka دارند را هدف قرار می‌دهد.

StatefulSet: Kafka

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: kafka
```

- **kind: StatefulSet**: این نوع دیپلوی برای حفظ حالت و پایداری Kafka استفاده می‌شود.
- **metadata:**
  - **name: kafka**: نام StatefulSet.

```
spec:
  serviceName: kafka-headless
  replicas: 2
```

- **spec:**
  - **serviceName: kafka-headless**: این StatefulSet از سرویس headless استفاده می‌کند.
  - **replicas: 2**: دو نمونه Kafka اجرا می‌شود.

```
  selector:
    matchLabels:
      app: kafka
```

```
template:
  metadata:
    labels:
      app: kafka
```

- **selector**: برچسب‌هایی که پادها باید داشته باشند.
- **template**: قالب پادها با برچسب app: kafka.

```
      containers:
        - name: kafka
          image: bitnami/kafka:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 9092
              name: kafka
            - containerPort: 9093
              name: kafka-internal
```

- **containers**:
  - یک کانتینر Kafka تعریف شده.
  - **image: bitnami/kafka:latest**: استفاده از آخرین نسخه Kafka از مخزن Bitnami.
  - **ports**: کانتینر به پورت‌های 9092 و 9093 گوش می‌دهد.

متغیرهای محیطی

```
          env:
            - name: KAFKA_BROKER_ID
              valueFrom:
                fieldRef:
                  fieldPath: metadata.labels['apps.kubernetes.io/pod-index']
```

- **KAFKA_BROKER_ID**: شناسه یکتا برای هر بروکر Kafka بر اساس اندیس پاد.
- سایر متغیرها تنظیمات مربوط به آدرس پاد، namespace، و تنظیمات Listenerها را تعریف می‌کنند.

Persistent Volume Claim

```
      volumeClaimTemplates:
        - metadata:
```

```
name: kafka-data
spec:
accessModes: ["ReadWriteOnce"]
resources:
requests:
storage: 10Gi
```

- **volumeClaimTemplates**: دیسک‌های پایدار برای نگهداری داده‌های Kafka.
- **storage: 10Gi**: هر پاد 10 گیگابایت فضای ذخیره‌سازی دارد.

فایل zookeeper-deployment.yaml

Deployment: Zookeeper

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: zookeeper
spec:
replicas: 1
```

- **kind: Deployment**: این فایل یک دیپلوی ساده برای Zookeeper است.
- **replicas: 1**: تنها یک نمونه Zookeeper اجرا می‌شود.

```
selector:
matchLabels:
app: zookeeper
template:
metadata:
labels:
app: zookeeper
```

- **selector** و **template**: برچسب‌های پادها برای تطبیق با سرویس Zookeeper.

Container: Zookeeper

```
containers:
- name: zookeeper
```

```
image: bitnami/zookeeper:latest
imagePullPolicy: IfNotPresent
ports:
  - containerPort: 2181
    name: client
```

- **containers:**
  - کانتینر Zookeeper با آخرین نسخه Bitnami.
  - **port: 2181**: پورت پیش‌فرض Zookeeper برای ارتباط با کلاینت‌ها.

Service: Zookeeper

```
apiVersion: v1
kind: Service
metadata:
  name: zookeeper
  labels:
    app: zookeeper
spec:
  selector:
    app: zookeeper
  ports:
    - port: 2181
      name: client
```

- این سرویس ارتباط پادهای Zookeeper را فراهم می‌کند و پورت 2181 برای کلاینت‌ها باز است.

توضیح کامل و دقیق خط به خط فایل‌های ارسال شده:

---

1. spark-submit-stream-processor.sh

این فایل یک اسکریپت شل است که برای اجرای یک برنامه Spark با استفاده از spark-submit استفاده می‌شود. خط به خط:

```
#!/bin/bash
```

- این خط نشان می‌دهد که اسکریپت با استفاده از Bash اجرا خواهد شد.

/opt/bitnami/spark/bin/spark-submit \

- این خط ابزار spark-submit را فراخوانی می‌کند که برای ارسال برنامه‌های Spark به خوشه استفاده می‌شود.
- مسیر به نصب Spark اشاره دارد.

--master spark://spark-master-service:7077 \

- مشخص می‌کند که برنامه به کدام Spark Master متصل شود. اینجا از یک آدرس TCP برای Master استفاده شده است.

--deploy-mode client \

- برنامه در حالت client اجرا می‌شود، به این معنی که برنامه از ماشین محلی کاربر اجرا و مدیریت می‌شود.

--name "StreamProcessor" \

- نام برنامه Spark به عنوان "StreamProcessor" تنظیم شده است.

--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.4 \

- بسته‌ی spark-sql-kafka برای تعامل با Kafka اضافه شده است.

--driver-memory 1g \

--conf spark.driver.maxResultSize=1g \

- حافظه‌ای که به فرآیند Driver اختصاص داده شده و حداکثر سایز نتایج مشخص می‌شود.

--executor-cores 1 \

--executor-memory 1g \

--total-executor-cores 1 \

--conf spark.executor.instances=1 \

- مشخصات مربوط به Executorها از جمله تعداد هسته‌ها، حافظه و تعداد کل Executorها تنظیم شده است.

--conf spark.driver.bindAddress=0.0.0.0 \

--conf spark.driver.host=stream-processor-service \

--conf spark.driver.port=7072 \

--conf spark.driver.blockManager.port=35635 \

- تنظیمات شبکه برای ارتباطات بین Driver و Executorها.

--py-files dependencies.zip \

- فایل‌های پایتون اضافی که به برنامه نیاز دارند، از طریق یک آرشیو زیپ ارائه شده‌اند.

${SPARK_APPLICATION_PYTHON_LOCATION}

- مسیر اسکریپت پایتون اصلی (که با یک متغیر محیطی تنظیم شده است).

---

2. stream-processor-deployment.yaml

این فایل تنظیمات Kubernetes را برای استقرار برنامه تعریف می‌کند.

apiVersion: apps/v1

kind: Deployment

- مشخص می‌کند که این فایل یک Deployment از نوع apps/v1 است.

:metadata

name: stream-processor

:labels

app: stream-processor

- متادیتای مربوط به Deployment شامل نام و برچسب‌ها.

:spec

replicas: 1

- تعداد کپی‌های این برنامه (پادها) برابر با 1 است.

:selector

:matchLabels

app: stream-processor

- این Deployment فقط پادهایی را مدیریت می‌کند که برچسب app: stream-processor دارند.

:template

:metadata

:labels

app: stream-processor

- متادیتای پادها شامل برچسب‌ها.

:spec

:containers

- name: stream-processor

image: stream-processor:latest

imagePullPolicy: IfNotPresent

- مشخصات کانتینر:
  - نام کانتینر: stream-processor
  - تصویر Docker: stream-processor:latest
  - سیاست دریافت تصویر: IfNotPresent (در صورت وجود، از تصویر کش استفاده می‌شود).

:ports

containerPort: 7072 -

name: headless-svc

containerPort: 8082 -

name: web-ui

containerPort: 35635 -

name: block-manager

- پورت‌های باز در کانتینر.

:env

name: SPARK_MASTER_URL -

"value: "spark://spark-master-service:7077

- متغیر محیطی برای آدرس Master Spark.

---

3. stream-processor.py

این فایل کد پایتون است که فرآیند پردازش جریان داده را پیاده‌سازی می‌کند.

import redis

```
from pyspark.sql import SparkSession
```

- کتابخانه‌های مورد استفاده:
  - redis: احتمالاً برای تعامل با یک پایگاه داده Redis.
  - SparkSession: نقطه ورود به Spark SQL.

```
spark = SparkSession.builder.appName("StreamProcessor").getOrCreate()
```

- یک SparkSession ایجاد می‌کند با نام "StreamProcessor".

```
df = (
    spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "kafka-0.kafka-headless:9092")
    .option("subscribe", "test-topic")
    .option("startingOffsets", "latest")
    .load()
)
```

- داده‌ها را از یک موضوع (test-topic Kafka) به صورت استریم می‌خواند.
- آدرس Bootstrap سرور Kafka و موضوع مشخص شده‌اند.

```
df_string = df.selectExpr("CAST(value AS STRING)")
```

- داده‌ها به صورت رشته تبدیل می‌شوند.

```
query = (
    df_string.writeStream.outputMode("append")
    .format("console")
    .option("truncate", False)
```

.start()

)

- خروجی داده‌ها به کنسول نوشته می‌شود.
- outputMode در حالت append است (فقط داده‌های جدید نمایش داده می‌شوند).

query.awaitTermination()

- برنامه در حالت اجرا باقی می‌ماند تا فرآیند استریم خاتمه پیدا کند.

البته! این دو فایل مربوط به تنظیمات **Deployment** و **Service** برای اجرای یک خوشه (Apache Spark Cluster) در Kubernetes هستند. هر کدام از این فایل‌ها وظیفه‌ای خاص دارند. حالا خط به خط توضیح می‌دم:

فایل `spark-master-deployment.yaml`:

بخش **Deployment**:

1. **`apiVersion: apps/v1`**:

این خط مشخص می‌کند که از نسخه `apps/v1` API Kubernetes استفاده می‌شود. این نسخه برای تعریف Deploymentها و StatefulSetها استفاده می‌شود.

2. **`kind: Deployment`**:

نوع این فایل یک Deployment است. Deployment در Kubernetes برای مدیریت مجموعه‌ای از Podها استفاده می‌شود و اطمینان می‌دهد که تعداد مشخصی از Replicaها همیشه در حال اجرا هستند.

3. **`metadata`**:

این بخش شامل اطلاعاتی درباره Deployment است.

- **`name: spark-master`**:

نام این Deployment را `spark-master` تعیین می‌کند.

4. **`spec`**:

این بخش مشخصات Deployment را تعریف می‌کند.

- **`replicas: 1`**:

تعداد Replicaهای این Deployment را 1 تعیین می‌کند. یعنی فقط یک Pod برای Master اجرا می‌شود.

- **`selector`**:

این بخش مشخص می‌کند که کدام Podها توسط این Deployment مدیریت می‌شوند.

- **`matchLabels`**:

برچسب‌هایی که Podها باید داشته باشند تا توسط این Deployment مدیریت شوند.

- **`app: spark-master`**:

Podهایی که برچسب `app=spark-master` دارند، توسط این Deployment مدیریت می‌شوند.

5. **`template`**:

این بخش مشخصات Podهایی که توسط این Deployment ایجاد می‌شوند را تعریف می‌کند.

- **`metadata`**:

اطلاعات مربوط به Podها.

- **`labels`**:

برچسب‌هایی که به Podها اختصاص داده می‌شوند.

- **`app: spark-master`**:

هر Pod ایجاد شده توسط این Deployment، برچسب `app=spark-master` خواهد داشت.

- **`spec`**:

مشخصات مربوط به Containerهای داخل Pod.

- **`containers`**:

لیست Containerهایی که در این Pod اجرا می‌شوند.

- **`name: spark-master`**:

نام Container را `spark-master` تعیین می‌کند.

- **`image: bitnami/spark:latest`**:

از Image مربوط به Apache Spark از Docker Hub استفاده می‌کند. این Image توسط Bitnami ارائه شده است.

- **`ports`**:

پورت‌هایی که در Container باز می‌شوند.

- **`containerPort: 7077`**:

پورت 7077 برای ارتباطات RPC (Remote Procedure Call) باز می‌شود.

- **`name: rpc`**:

نام این پورت `rpc` است.

- **`containerPort: 8080`**:

پورت 8080 برای رابط کاربری وب (Web UI) باز می‌شود.

- **`name: webui`**:

نام این پورت `webui` است.

- **`env`**:

متغیرهای محیطی که به Container ارسال می‌شوند.

- **`name: SPARK_MODE`**:

حالت اجرای Spark را `master` تعیین می‌کند. یعنی این Container به عنوان Master اجرا می‌شود.

- **`name: SPARK_RPC_AUTHENTICATION_ENABLED`**:

احراز هویت RPC را غیرفعال می‌کند (`no`).

- **`name: SPARK_RPC_ENCRYPTION_ENABLED`**:

  رمزنگاری RPC را غیرفعال می‌کند (`no`).

- **`name: SPARK_SSL_ENABLED`**:

  SSL را غیرفعال می‌کند (`no`).

- **`name: SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED`**:

  رمزنگاری ذخیره‌سازی محلی را غیرفعال می‌کند (`no`).

- **`name: SPARK_LOCAL_DIRS`**:

  مسیر ذخیره‌سازی موقت را به `/tmp` تنظیم می‌کند.

بخش **Service**:

1. **`apiVersion: v1`**:

   این خط مشخص می‌کند که از نسخه `v1` API Kubernetes استفاده می‌شود. این نسخه برای تعریف Serviceها استفاده می‌شود.

2. **`kind: Service`**:

   نوع این فایل یک Service است. Service در Kubernetes برای ایجاد یک نقطه دسترسی ثابت به Podها استفاده می‌شود.

3. **`metadata`**:

   این بخش شامل اطلاعاتی درباره Service است.

   - **`name: spark-master-service`**:

     نام این Service را `spark-master-service` تعیین می‌کند.

4. **`spec`**:

   این بخش مشخصات Service را تعریف می‌کند.

- **`ports`**:

پورت‌هایی که توسط این Service باز می‌شوند.

- **`port: 7077`**:

پورت 7077 برای ارتباطات RPC باز می‌شود.

- **`name: rpc`**:

نام این پورت `rpc` است.

- **`port: 8080`**:

پورت 8080 برای رابط کاربری وب (Web UI) باز می‌شود.

- **`name: webui`**:

نام این پورت `webui` است.

- **`selector`**:

این بخش مشخص می‌کند که این Service به کدام Podها متصل می‌شود.

- **`app: spark-master`**:

این Service به Podهایی که برچسب `app=spark-master` دارند، متصل می‌شود.

فایل `spark-worker-deployment.yaml`:

بخش **Deployment**:

1. **`apiVersion: apps/v1`**:

همانند فایل قبلی، این خط مشخص می‌کند که از نسخه `apps/v1` API Kubernetes استفاده می‌شود.

2. **`kind: Deployment`**:

نوع این فایل یک Deployment است.

3. **`metadata`**:

این بخش شامل اطلاعاتی درباره Deployment است.

- **`name: spark-worker`**:

نام این Deployment را `spark-worker` تعیین می‌کند.

4. **`spec`**:

این بخش مشخصات Deployment را تعریف می‌کند.

- **`replicas: 2`**:

تعداد Replicaهای این Deployment را 2 تعیین می‌کند. یعنی دو Pod برای Worker اجرا می‌شوند.

- **`selector`**:

این بخش مشخص می‌کند که کدام Podها توسط این Deployment مدیریت می‌شوند.

- **`matchLabels`**:

برچسب‌هایی که Podها باید داشته باشند تا توسط این Deployment مدیریت شوند.

- **`app: spark-worker`**:

Podهایی که برچسب `app=spark-worker` دارند، توسط این Deployment مدیریت می‌شوند.

5. **`template`**:

این بخش مشخصات Podهایی که توسط این Deployment ایجاد می‌شوند را تعریف می‌کند.

- **`metadata`**:

اطلاعات مربوط به Podها.

- **`labels`**:

برچسب‌هایی که به Podها اختصاص داده می‌شوند.

- **`app: spark-worker`**:

هر Pod ایجاد شده توسط این Deployment، برچسب `app=spark-worker` خواهد داشت.

- **`spec`**:

مشخصات مربوط به Containerهای داخل Pod.

- **`containers`**:

لیست Containerهایی که در این Pod اجرا می‌شوند.

- **`name: spark-worker`**:

نام Container را `spark-worker` تعیین می‌کند.

- **`image: bitnami/spark:latest`**:

از Image مربوط به Apache Spark از Docker Hub استفاده می‌کند. این Image توسط Bitnami ارائه شده است.

- **`env`**:

متغیرهای محیطی که به Container ارسال می‌شوند.

- **`name: SPARK_MODE`**:

حالت اجرای Spark را `worker` تعیین می‌کند. یعنی این Container به عنوان Worker اجرا می‌شود.

- **`name: SPARK_MASTER_URL`**:

آدرس Master را به `spark://spark-master-service:7077` تنظیم می‌کند. این آدرس به Service مربوط به Master اشاره می‌کند.

- **`name: SPARK_WORKER_MEMORY`**:

حافظه اختصاص داده شده به هر Worker را 2 گیگابایت (`2G`) تعیین می‌کند.

- **`name: SPARK_WORKER_CORES`**:

تعداد هسته‌های CPU اختصاص داده شده به هر Worker را 2 تعیین می‌کند.

- **`name: SPARK_RPC_AUTHENTICATION_ENABLED`**:

احراز هویت RPC را غیرفعال می‌کند (`no`).

- **`name: SPARK_RPC_ENCRYPTION_ENABLED`**:

رمزنگاری RPC را غیرفعال می‌کند (`no`).

- **`name: SPARK_SSL_ENABLED`**:

SSL را غیرفعال می‌کند (`no`).

- **`name: SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED`**:

رمزنگاری ذخیره‌سازی محلی را غیرفعال می‌کند (`no`).

فایل اول: spark-master-deployment.yaml

این فایل مربوط به راه‌اندازی Spark Master در Kubernetes است.

بخش Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spark-master
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spark-master
  template:
    metadata:
      labels:
        app: spark-master
    spec:
      containers:
      - name: spark-master
        image: bitnami/spark:latest
        ports:
        - containerPort: 7077
          name: rpc
        - containerPort: 8080
          name: webui
        env:
        - name: SPARK_MODE
          value: master
        - name: SPARK_RPC_AUTHENTICATION_ENABLED
          value: "no"
        - name: SPARK_RPC_ENCRYPTION_ENABLED
          value: "no"
        - name: SPARK_SSL_ENABLED
          value: "no"
        - name: SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED
          value: "no"
        - name: SPARK_LOCAL_DIRS
          value: /tmp
```

1. **apiVersion: apps/v1**: این نشان‌دهنده این است که این بخش از API نسخه apps/v1 استفاده می‌کند.
2. **kind: Deployment**: نوع Deployment، Kubernetes Object است.
3. **metadata.name**: نام Deployment که "spark-master" است.
4. **spec.replicas**: تعداد Podهایی که باید ایجاد شوند، اینجا یک عدد.
5. **spec.selector.matchLabels**: مشخص می‌کند این Deployment با Podهایی که برچسب app: spark-master دارند کار می‌کند.
6. **template.metadata.labels**: برچسبی که برای Podها تعریف شده است.
7. **containers**: تعریف مشخصات کانتینر Spark Master:
   - **image**: ایمیج Docker که استفاده می‌شود (bitnami/spark:latest).
   - **ports**: دو پورت:
     - **7077**: پورت RPC که Spark Master برای ارتباط استفاده می‌کند.
     - **8080**: پورت رابط وب Spark.
   - **env**: متغیرهای محیطی:
     - **SPARK_MODE: master**: حالت Spark را "master" تعیین می‌کند.
     - سایر متغیرها مثل SPARK_RPC_AUTHENTICATION_ENABLED و SPARK_SSL_ENABLED امنیت و رمزنگاری را غیرفعال می‌کنند.
     - **SPARK_LOCAL_DIRS**: مسیر ذخیره‌سازی موقت را به /tmp تنظیم می‌کند.

بخش Service:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: spark-master-service
spec:
  ports:
  - port: 7077
    name: rpc
  - port: 8080
    name: webui
  selector:
    app: spark-master
```

kind: Service: تعریف یک Service که Spark Master را در شبکه قابل دسترسی می‌کند.

1. **metadata.name**: نام این سرویس "spark-master-service" است.
2. **ports**: تعریف دو پورت:
   - **7077**: برای ارتباط RPC.
   - **8080**: برای رابط وب.
3. **selector**: تعیین می‌کند این Service به Podهایی که برچسب app: spark-master دارند متصل شود.

فایل دوم: spark-worker-deployment.yaml

این فایل مربوط به تنظیم و راه‌اندازی Spark Worker است.

Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spark-worker
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spark-worker
  template:
    metadata:
      labels:
        app: spark-worker
    spec:
      containers:
      - name: spark-worker
        image: bitnami/spark:latest
        env:
        - name: SPARK_MODE
          value: "worker"
        - name: SPARK_MASTER_URL
          value: "spark://spark-master-service:7077"
        - name: SPARK_WORKER_MEMORY
          value: 2G
        - name: SPARK_WORKER_CORES
          value: "2"
        - name: SPARK_RPC_AUTHENTICATION_ENABLED
          value: "no"
        - name: SPARK_RPC_ENCRYPTION_ENABLED
          value: "no"
        - name: SPARK_SSL_ENABLED
          value: "no"
        - name: SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED
          value: "no"
```

1. **apiVersion: apps/v1**: از API نسخه apps/v1 استفاده می‌شود.
2. **kind: Deployment**: اینجا Deployment برای Spark Worker تنظیم شده است.
3. **metadata.name**: نام این "spark-worker" Deployment است.
4. **spec.replicas**: تعداد Podهای Worker که باید اجرا شوند. اینجا مقدار 2 تنظیم شده است.
5. **spec.selector.matchLabels**: مشخص می‌کند این Deployment به Podهایی که برچسب app: spark-worker دارند، مربوط است.
6. **template.spec.containers**: تعریف مشخصات کانتینر Spark Worker:
   - **image**: از ایمیج Docker bitnami/spark:latest استفاده می‌شود.
   - **env**: متغیرهای محیطی:
     - **SPARK_MODE: worker**: حالت این Pod را "worker" تنظیم می‌کند.
     - **SPARK_MASTER_URL**: آدرس Spark Master که باید به آن متصل شود. اینجا آدرس spark://spark-master-service:7077.
     - **SPARK_WORKER_MEMORY**: مقدار حافظه‌ای که Worker استفاده می‌کند (2 گیگابایت).
     - **SPARK_WORKER_CORES**: تعداد CPU Coreهایی که Worker استفاده می‌کند (2 عدد).
     - سایر متغیرها برای غیرفعال کردن امنیت و رمزنگاری.

خلاصه:

1. **فایل spark-master-deployment.yaml:**
   - Spark Master را راه‌اندازی می‌کند.
   - پورت‌های ارتباطی و رابط وب را مشخص می‌کند.
   - با استفاده از یک Service، آن را در شبکه قابل دسترسی می‌کند.
2. **فایل spark-worker-deployment.yaml:**
   - دو Pod برای Spark Worker راه‌اندازی می‌کند.
   - هر Worker به Spark Master متصل می‌شود و از 2 گیگابایت RAM و 2 Core استفاده می‌کند.

---

6. **سرویس تجمیع داده (Aggregator Service)**:
این سرویس عملکرد کلی هر سهام رو خلاصه‌سازی می‌کنه. مثلاً می‌تونه میانگین قیمت روزانه، حجم معاملات، یا تعداد سیگنال‌های تولیدشده رو محاسبه کنه. این اطلاعات می‌تونه برای تحلیل‌های بعدی مفید باشه.

این فایل YAML شامل پیکربندی‌هایی برای راه‌اندازی یک دیتابیس Redis در Kubernetes است. حالا خط به خط توضیحش میدم:

بخش اول: PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: redis-data
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

1. **apiVersion: v1**: این خط مشخص می‌کند که این آبجکت از نسخه اول API Kubernetes استفاده می‌کند.
2. **kind: PersistentVolumeClaim**: این نشان‌دهنده این است که این بخش مربوط به یک درخواست برای یک Persistent Volume (PVC) است. PVC برای ذخیره‌سازی پایدار استفاده می‌شود.
3. **metadata.name**: نام PVC را تعیین می‌کند که در اینجا "redis-data" است.
4. **spec.accessModes**: نوع دسترسی به ذخیره‌سازی را مشخص می‌کند. ReadWriteOnce یعنی این Volume فقط توسط یک Pod می‌تواند به صورت خواندن و نوشتن استفاده شود.
5. **spec.resources.requests.storage**: میزان فضایی که برای ذخیره‌سازی درخواست شده است. در اینجا 10 گیگابایت.

بخش دوم: Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    app: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
```

```
spec:
  containers:
  - name: redis
    image: bitnami/redis:latest
    ports:
    - containerPort: 6379
    env:
    - name: ALLOW_EMPTY_PASSWORD
      value: "yes"
    volumeMounts:
    - name: redis-data
      mountPath: /bitnami/redis/data
  volumes:
  - name: redis-data
    persistentVolumeClaim:
      claimName: redis-data
```

1. **apiVersion: apps/v1**: نشان‌دهنده این است که این بخش از API نسخه "apps/v1" استفاده می‌کند.
2. **kind: Deployment**: این بخش نشان می‌دهد که نوع آبجکت Deployment، Kubernetes است. Deployment برای مدیریت و مقیاس‌دهی Podها استفاده می‌شود.
3. **metadata.name**: نام این Deployment را "redis" تعیین می‌کند.
4. **metadata.labels**: برچسب‌ها برای شناسایی این Deployment استفاده می‌شوند. اینجا برچسب app: redis تعیین شده است.
5. **spec.replicas**: تعداد Podهای این Deployment را مشخص می‌کند. در اینجا 1 عدد است.
6. **spec.selector.matchLabels**: تعیین می‌کند که این Deployment باید Podهایی که برچسب app: redis دارند را مدیریت کند.
7. **template.metadata.labels**: برچسبی که به Podهای ایجاد شده اعمال می‌شود. در اینجا app: redis.
8. **template.spec.containers**: تعریف کانتینرهای این Pod.
   - **name: redis**: نام کانتینر.
   - **image: bitnami/redis:latest**: ایمیج Docker که برای کانتینر استفاده می‌شود. اینجا Redis از Bitnami.
   - **ports.containerPort: 6379**: پورت Redis که داخل کانتینر استفاده می‌شود.
   - **env**: تعریف متغیر محیطی.
     - **ALLOW_EMPTY_PASSWORD**: این متغیر مقدار yes دارد که به Redis اجازه می‌دهد بدون پسورد کار کند.
   - **volumeMounts**: تعریف محل Mount کردن Volume.
     - **name: redis-data**: نام Volume که باید Mount شود.
     - **mountPath: /bitnami/redis/data**: محلی که Volume در کانتینر Mount می‌شود.
9. **volumes**: تعریف Volumeهایی که Pod استفاده می‌کند.
   - **name: redis-data**: نام Volume.
   - **persistentVolumeClaim.claimName: redis-data**: استفاده از PVC به نام "redis-data" که قبلاً تعریف شده.

بخش سوم: Service

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
  - protocol: TCP
    port: 6379
    targetPort: 6379
```

1. **apiVersion: v1**: نسخه API Kubernetes.
2. **kind: Service**: این بخش نشان‌دهنده یک Service است. Service برای در دسترس قرار دادن Podها استفاده می‌شود.
3. **metadata.name**: نام این Service "redis" است.
4. **spec.selector**: تعیین می‌کند که این Service باید به Podهایی که برچسب app: redis دارند متصل شود.
5. **spec.ports**: تعریف پورت‌هایی که Service فراهم می‌کند.
   - **protocol: TCP**: پروتکل ارتباطی (TCP).
   - **port: 6379**: پورت که توسط Service ارائه می‌شود.
   - **targetPort: 6379**: پورت مقصد روی Podها.

خلاصه:

- **PersistentVolumeClaim** برای ذخیره‌سازی پایدار دیتای Redis تعریف شده.
- **Deployment** یک Pod با ایمیج Redis را اجرا می‌کند و به PVC متصل است.
- **Service** ارتباط شبکه‌ای را فراهم می‌کند و Pod را در پورت 6379 در دسترس قرار می‌دهد.

---

5. **سرویس اطلاع‌رسانی (Notification Service)**:

به محض اینکه سیگنال‌های خرید یا فروش تولید بشن، این سرویس کاربران رو مطلع می‌کنه. این اطلاع‌رسانی می‌تونه از طریق ایمیل، پیامک، یا حتی یک نوتیفیکیشن در داشبورد باشه. هدف اینه که کاربران بلافاصله از تغییرات بازار با خبر بشن.

حالا به بررسی دقیق فایل‌های دیگر می‌پردازیم:

7. **سرویس نمایش داده (Visualization Service)**:
این سرویس داده‌ها و سیگنال‌ها رو روی یک داشبورد نمایش می‌ده. کاربران می‌تونن به صورت بلادرنگ تغییرات قیمت، اندیکاتورها، و سیگنال‌ها رو ببینن. این داشبورد می‌تونه با استفاده از فناوری‌هایی مثل HTML/CSS/JavaScript یا حتی کتابخانه‌های نمودارسازی مثل D3.js یا Chart.js پیاده‌سازی بشه.