

به نام خدا

داکیومنت پروژه Stock-Stream-Processing

درس :

سیستم‌های توزیع شده

استاد:

دکتر محسن شریفی

دانشجویان:

سید محمدامین حائری

علیرضا نظری

3.....	تعریف صورت مسئله
3.....	اجزای اصلی پروژه
3.....	دریافت داده Data ingestion
14.....	سرویس پردازش استریم + Stream process
37.....	سرویس ذخیره سازی
0.....	سیستم اطلاع رسانی (داخل گیت هاب) Notification
0.....	نمایش داده (داخل گیت هاب)

1- تعریف صورت مسئله

ما باید یک سیستم توزیع شده برای تحلیل مالی و پیشنهاد خرید و فروش به صورت Real time طراحی و پیاده سازی 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 پردازد. 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 داده های قیمتی از معماری توزیع شده و پردازش داده های استریمی استفاده شده و به عنوان داده ورودی از API صرافی alphavantage استفاده شده است.

2- اجزای اصلی سیستم

دریافت داده Data ingestion:

در اینجا ما یک API به صرافی alphavantage زده ایم که قیمت بیت کوین را به صورت استریم و با فاصله زمانی 1 ثانیه دریافت می کند. این داده ها شامل قیمت فعلی بیت کوین هستند. کار اصلی این سرویس این است که داده ها را اعتبارسنجی کند (مثلاً مطمئن شود که فیلدهای ضروری مثل `price` و `timestamp` وجود دارند) و سپس آن ها را به سرویس پردازش جریان ارسال کند. این سرویس نقش یک دروازه ورودی را بازی می کند و باید بتواند هم داده های شبیه سازی شده و هم داده های واقعی از API صرافی alphavantage را مدیریت کند.

2.1 - Data ingestor.py

این فایل یک تولیدکننده (Producer) برای ارسال پیام ها به یک تاپیک (Topic) در Kafka است. این اسکریپت به زبان پایتون نوشته شده و از کتابخانه kafka-python برای ارتباط با Kafka استفاده می کند. هدف اصلی این فایل، تولید داده و ارسال آن به یک تاپیک (Topic) در Kafka است. تاپیک ها در Kafka مانند صف هایی هستند که داده ها در آن ها ذخیره می شوند و سایر سرویس ها یا سیستم ها می توانند از این داده ها استفاده کنند. این فایل به طور مداوم پیام هایی را تولید کرده و به Kafka ارسال می کند تا سایر بخش های سیستم بتوانند از این داده ها استفاده کنند.

2.1.1. نحوه کار فایل:

این فایل به طور مداوم و در یک حلقه بی نهایت (while True) کار می کند و مراحل زیر را انجام می دهد:

1. ایجاد یک تولیدکننده (Kafka Producer):

- فایل ابتدا یک تولید کننده Kafka ایجاد می کند. این تولید کننده مسئول ارسال پیام ها به Kafka است.
- آدرس سرورهای Kafka از طریق یک متغیر محیطی (KAFKA_BOOTSTRAP_SERVERS) خوانده می شود. اگر این متغیر تنظیم نشده باشد، از یک آدرس پیش فرض استفاده می شود.

2. ساخت پیام:

- در هر تکرار حلقه، یک پیام جدید ساخته می‌شود. این پیام شامل سه بخش اصلی است:
 1. timestamp: زمان فعلی به فرمت ISO. این زمان نشان‌دهنده لحظه تولید پیام است.
 2. message: یک پیام متنی ثابت ("Hello from Python Producer"). این پیام می‌تواند در سیستم‌های واقعی با داده‌های واقعی جایگزین شود.
 3. id: یک شناسه منحصر به فرد که بر اساس زمان فعلی ایجاد می‌شود. این شناسه می‌تواند برای ردیابی پیام‌ها استفاده شود.

3. ارسال پیام به Kafka:

- پیام ساخته شده به یک تاپیک Kafka ارسال می‌شود. نام تاپیک از طریق یک متغیر محیطی KAFKA_TOPIC خوانده می‌شود. اگر این متغیر تنظیم نشده باشد، از یک نام پیش‌فرض (test-topic) استفاده می‌شود.
- پس از ارسال پیام، اطلاعات مربوط به تاپیک، پارتیشن و آفست (Offset) چاپ می‌شود. این اطلاعات برای ردیابی و اشکال‌زدایی مفید هستند.

4. تاخیر بین ارسال پیام‌ها:

- بین ارسال هر پیام، یک تاخیر وجود دارد. مدت این تاخیر از طریق یک متغیر محیطی (MESSAGE_INTERVAL) تنظیم می‌شود. اگر این متغیر تنظیم نشده باشد، به طور پیش‌فرض از 5 ثانیه استفاده می‌شود.

توضیح 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.

```
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.

```
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 استثناءها:

```

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 می ماند.

اجرای اصلی:

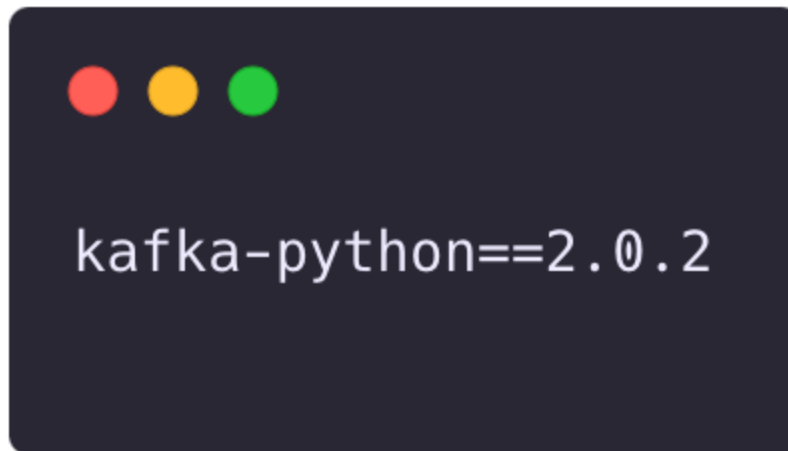
```

if __name__ ==
 "__main__":
    send_message( )

```

- تابع `send_message` هنگام اجرای مستقیم فایل فراخوانی می شود.

فایل `requirements.txt`:



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

فایل data-ingestor-deployment.yaml:

فایل data-ingestor-deployment.yaml یک فایل پیکربندی Kubernetes است که برای استقرار (Deploy) و اجرای سرویس data-ingestor در یک کلاستر Kubernetes استفاده می‌شود. این فایل به Kubernetes می‌گوید که چگونه سرویس data-ingestor را به عنوان یک Deployment اجرا کند. هدف اصلی این فایل، اجرای سرویس data-ingestor (که توسط فایل data-ingestor.py پیاده‌سازی شده) در یک محیط Kubernetes است. این فایل مشخص می‌کند که چگونه کانتینرهای حاوی سرویس data-ingestor باید اجرا شوند، چگونه مقیاس‌پذیری انجام شود و چگونه تنظیمات محیطی (Environment Variables) به کانتینرها منتقل شوند.

2. ساختار فایل و اجزای آن:

این فایل یک Deployment Kubernetes تعریف می‌کند. Deployment یک مفهوم در Kubernetes است که برای مدیریت و به‌روزرسانی کانتینرها استفاده می‌شود. اجزای اصلی این فایل به شرح زیر است:

1. `apiVersion: apps/v1`

- این بخش مشخص می‌کند که از کدام نسخه API Kubernetes استفاده می‌شود. در اینجا از نسخه apps/v1 استفاده شده است که برای تعریف Deployment ها مناسب است.

2. `kind: Deployment`

- نوع منبع Kubernetes که در اینجا تعریف می‌شود. در اینجا نوع منبع Deployment است که برای اجرای و مدیریت کانتینرها استفاده می‌شود.

3. metadata:

- این بخش شامل اطلاعات توصیفی درباره Deployment است. در اینجا نام Deployment به عنوان data-ingestor تعریف شده است.

4. spec:

- این بخش مشخصات اصلی Deployment را تعریف می‌کند. شامل موارد زیر است:
 - replicas: 1: تعداد نمونه‌های (Pod) در حال اجرا از این Deployment. در اینجا تنها یک نمونه اجرا می‌شود. اگر نیاز به مقیاس‌پذیری باشد، می‌توان این عدد را افزایش داد.
 - selector: این بخش مشخص می‌کند که کدام Podها توسط این Deployment مدیریت می‌شوند. در اینجا Podهایی با برچسب app: data-ingestor انتخاب می‌شوند.
 - template: این بخش مشخصات Podها را تعریف می‌کند. هر Pod می‌تواند شامل یک یا چند کانتینر باشد.
- metadata: برچسب‌های Pod. در اینجا برچسب app: data-ingestor به Podها اختصاص داده می‌شود.
- spec: مشخصات کانتینرهای داخل Pod.
 - containers: لیست کانتینرهایی که در Pod اجرا می‌شوند.
 - name: data-ingestor: نام کانتینر.
 - image: data-ingestor:latest: تصویر داکری که برای کانتینر استفاده می‌شود. در اینجا از تصویر data-ingestor:latest استفاده می‌شود.
 - imagePullPolicy: IfNotPresent: سیاست کشیدن تصویر. اگر تصویر از قبل وجود داشته باشد، از آن استفاده می‌شود.
 - env: لیست متغیرهای محیطی که به کانتینر منتقل می‌شوند. این متغیرها برای پیکربندی سرویس data-ingestor استفاده می‌شوند.
 - KAFKA_BOOTSTRAP_SERVERS: آدرس سرورهای Kafka. در اینجا kafka-0.kafka-headless:9092 تنظیم شده است.
 - KAFKA_TOPIC: نام تاپیک Kafka که پیام‌ها به آن ارسال می‌شوند. در اینجا test-topic تنظیم شده است.
 - MESSAGE_INTERVAL: فاصله زمانی بین ارسال پیام‌ها. در اینجا 2 ثانیه تنظیم شده است.

3. نقش فایل در سیستم:

این فایل نقش مهمی در اجرای سرویس data-ingestor در یک محیط Kubernetes ایفا می‌کند. با استفاده از این فایل، می‌توان سرویس data-ingestor را به راحتی در یک کلاستر Kubernetes استقرار داد و مدیریت کرد. برخی از وظایف اصلی این فایل عبارتند از:

1. اجرای سرویس data-ingestor:

- این فایل مشخص می‌کند که سرویس data-ingestor باید به عنوان یک کانتینر در Kubernetes اجرا شود.

2. مدیریت کانتینرها:

- این فایل مشخص می‌کند که چگونه کانتینرها باید اجرا شوند، از کدام تصویر داکر استفاده کنند و چه تنظیمات محیطی به آن‌ها منتقل شود.

3. مقیاس‌پذیری:

- با تغییر مقدار replicas، می‌توان تعداد نمونه‌های در حال اجرا از سرویس data-ingestor را افزایش یا کاهش داد. این کار برای مقیاس‌پذیری سرویس در شرایط مختلف (مانند افزایش بار کاری) مفید است.

4. پیکربندی سرویس:

- این فایل تنظیمات محیطی (مانند آدرس Kafka، نام تاپیک و فاصله ارسال پیام) را به سرویس data-ingestor منتقل می‌کند. این تنظیمات از طریق متغیرهای محیطی به کانتینر منتقل می‌شوند.

توضیح فایل `data-ingestor-deployment.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.

```

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.

```

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).

2-2. ****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. این دو فایل کانفیگ (kafka-deployment.yaml و zookeeper-deployment.yaml) برای استقرار و مدیریت سیستم‌های Apache Kafka و Apache Zookeeper در یک محیط Kubernetes استفاده می‌شوند. این دو سیستم معمولاً در کنار هم استفاده می‌شوند، زیرا Zookeeper برای مدیریت و هماهنگی بین Brokerهای Kafka ضروری است. در ادامه نقش هر کدام از این کانفیگ‌ها و ارتباط بین آن‌ها توضیح داده می‌شود:

1. فایل zookeeper-deployment.yaml:

- این فایل برای استقرار Apache Zookeeper استفاده می‌شود. Zookeeper یک سرویس توزیع‌شده است که برای مدیریت و هماهنگی بین Brokerهای Kafka استفاده می‌شود. نقش اصلی Zookeeper در Kafka شامل:
- **مدیریت Brokerها:** Zookeeper اطلاعات مربوط به Brokerهای Kafka (مانند وضعیت آن‌ها، Leader و Follower بودن) را نگهداری می‌کند.
 - **مدیریت Topicها و Partitionها:** Zookeeper اطلاعات مربوط به Topicها و Partitionهای Kafka را ذخیره می‌کند.
 - **هماهنگی بین Brokerها:** Zookeeper به Brokerها کمک می‌کند تا با هم هماهنگ شوند و در صورت نیاز Leader انتخاب کنند.
- در این فایل:
- یک **Deployment** برای Zookeeper تعریف شده است که یک نمونه (Replica) از Zookeeper را اجرا می‌کند.
 - یک **Service** نیز تعریف شده است که به سایر سرویس‌ها (مانند Kafka) اجازه می‌دهد به Zookeeper متصل شوند.

2. فایل kafka-deployment.yaml:

- این فایل برای استقرار Apache Kafka استفاده می‌شود. Kafka یک سیستم پیام‌رسانی توزیع‌شده است که برای انتقال داده‌ها بین سیستم‌ها با کارایی بالا استفاده می‌شود. نقش اصلی Kafka شامل:
- **ذخیره و انتقال داده‌ها:** Kafka داده‌ها را در Topicها ذخیره می‌کند و به سیستم‌های دیگر اجازه می‌دهد این داده‌ها را مصرف کنند.
 - **مقیاس‌پذیری:** Kafka می‌تواند به راحتی با افزایش Brokerها مقیاس‌پذیر شود.
 - **تحمل خطا:** Kafka داده‌ها را در چندین Broker تکرار می‌کند تا در صورت خرابی یک Broker، داده‌ها از دست نروند.
- در این فایل:

- یک `StatefulSet` برای `Kafka` تعریف شده است که دو نمونه (`Replica`) از `Kafka Broker` را اجرا می‌کند.
- `StatefulSet` برای سیستم‌هایی مانند `Kafka` مناسب است که نیاز به ذخیره‌سازی پایدار داده‌ها دارند.
- یک `Service` نیز تعریف شده است که به سایر سرویس‌ها اجازه می‌دهد به `Kafka` متصل شوند.
- `Kafka` از `Zookeeper` برای مدیریت `Broker` ها و `Topic` ها استفاده می‌کند. در این فایل، `Kafka` به `Zookeeper` متصل می‌شود (با استفاده از متغیر محیطی `KAFKA_CFG_ZOOKEEPER_CONNECT`).

ارتباط بین `Kafka` و `Zookeeper`:

- `Zookeeper` به عنوان یک سرویس هماهنگ‌کننده برای `Kafka` عمل می‌کند. `Kafka Broker` ها از `Zookeeper` برای مدیریت وضعیت خود (مانند `Leader` و `Follower` بودن) و ذخیره اطلاعات مربوط به `Topic` ها و `Partition` ها استفاده می‌کنند.
- در این کانفیگ، `Kafka Broker` ها از طریق متغیر محیطی `KAFKA_CFG_ZOOKEEPER_CONNECT` به `Zookeeper` متصل

توضیح 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.

توضیح 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: اجرا می‌شود Zookeeper تنها یک نمونه
selector:
  matchLabels:
    app: zookeeper
template:
  metadata:
    labels:
      app: zookeeper
```

template و selector: برجسب‌های 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.

```
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.

1. spark-submit-stream-processor.sh

این فایل یک اسکریپت Bash است که برای ارسال (Submit) یک برنامه Apache Spark به کلاس‌تر (Spark Cluster) استفاده می‌شود. این اسکریپت تنظیمات لازم برای اجرای برنامه Spark Streaming را تعیین می‌کند. در ادامه به جزئیات این فایل می‌پردازیم:


- این اسکریپت برنامه Spark Streaming را به کلاس‌تر Spark ارسال می‌کند و تنظیمات لازم برای اجرای آن را تعیین می‌کند.

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 \
```

- این گزینه مشخص می‌کند که برنامه به کدام کلاس‌تر Spark متصل شود. در اینجا، کلاس‌تر Spark با آدرس spark://spark-master-service:7077 در نظر گرفته شده است.
- spark-master-service نام سرویس (Service) در Kubernetes است که به Master Node کلاس‌تر Spark اشاره می‌کند.



```
--deploy-mode client \
```

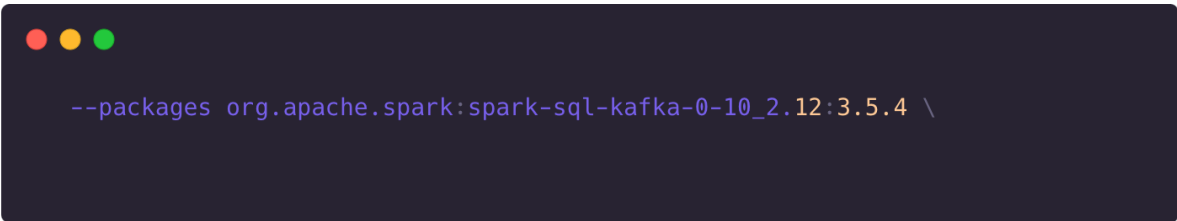
- این گزینه مشخص می‌کند که برنامه در چه حالتی اجرا شود. در حالت client، درایور (Driver) برنامه خارج از کلاس‌تر Spark اجرا می‌شود (معمولاً روی ماشین محلی یا همان جایی که اسکریپت اجرا می‌شود).



```
--name "StreamProcessor" \
```

این گزینه نام برنامه را مشخص می‌کند. این نام در رابط کاربری (UI) کلاس‌تر Spark نمایش داده می‌شود.

- 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.
- این گزینه کتابخانه‌های مورد نیاز برای اتصال به Kafka را به برنامه اضافه می‌کند. در اینجا، کتابخانه spark-sql-kafka برای خواندن داده‌ها از Kafka استفاده می‌شود.

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

- حافظه‌ای که به فرآیند Driver اختصاص داده شده و حداکثر سایز نتایج مشخص می‌شود.
- این گزینه مقدار حافظه (RAM) اختصاص داده شده به درایور (Driver) برنامه را مشخص می‌کند. در اینجا، 1 گیگابایت حافظه به درایور اختصاص داده شده است.
- 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.

2. stream-processor-deployment.yaml

این فایل یک Deployment و Service در Kubernetes تعریف می‌کند که برای اجرای برنامه Spark Streaming استفاده می‌شود. در ادامه به جزئیات این فایل می‌پردازیم:

نقش کلی:

- این فایل برنامه Spark Streaming را به عنوان یک Container در Kubernetes اجرا می‌کند و تنظیمات شبکه و منابع مورد نیاز را تعیین می‌کند.

جزئیات Deployment:

1. replicas: 1

- این گزینه تعداد Replicaهای برنامه را مشخص می‌کند. در اینجا، فقط 1 Replica اجرا می‌شود.

2. containers:

- این بخش مشخصات Container برنامه را تعریف می‌کند.
- `image: stream-processor:latest` از این تصویر (Image) Docker برای اجرای برنامه استفاده می‌شود.
- `ports`: پورت‌های مورد نیاز برای ارتباط با برنامه تعیین شده‌اند:
 - `7072`: پورت درایور برنامه.
 - `8082`: پورت رابط کاربری (UI) برنامه.
 - `35635`: پورت Block Manager.

3. env:

- این بخش متغیرهای محیطی مورد نیاز برنامه را تعریف می‌کند.
- `SPARK_MASTER_URL`: آدرس کلاس‌تر Spark که برنامه به آن متصل می‌شود (`spark://spark-master-service:7077`).

جزئیات Service:

- این بخش یک سرویس (Service) در Kubernetes تعریف می‌کند که به برنامه Spark Streaming اجازه می‌دهد از طریق شبکه قابل دسترسی باشد.
- `ports`: پورت‌های سرویس تعیین شده‌اند:
 - `7072`: پورت درایور برنامه.
 - `8082`: پورت رابط کاربری (UI) برنامه.
 - `35635`: پورت Block Manager.

```
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 یک برنامه Python است که با استفاده از Spark Streaming داده‌ها را از یک Topic Kafka می‌خواند و آن‌ها را پردازش می‌کند. در ادامه به جزئیات این فایل می‌پردازیم:

نقش کلی:

- این برنامه داده‌ها را از Kafka می‌خواند و آن‌ها را پردازش می‌کند.

جزئیات کد:

1. ایجاد SparkSession:

- یک SparkSession ایجاد می‌شود که نقطه ورود به برنامه‌های Spark است.
- appName("StreamProcessor"): نام برنامه را مشخص می‌کند.

2. خواندن داده‌ها از Kafka:

- داده‌ها از Kafka با استفاده از کتابخانه spark-sql-kafka خوانده می‌شوند.
- kafka.bootstrap.servers: آدرس Brokerهای (kafka-0.kafka-headless:9092) (Kafka).
- subscribe: Topic Kafka که داده‌ها از آن خوانده می‌شوند (test-topic).
- startingOffsets: مشخص می‌کند که داده‌ها از کدام Offset شروع به خواندن شوند (latest به معنی خواندن از آخرین Offset).

3. تبدیل داده‌ها به رشته:

- داده‌های خوانده شده از Kafka به صورت رشته (String) تبدیل می‌شوند.

4. نمایش داده‌ها در کنسول:

- داده‌های پردازش شده در کنسول نمایش داده می‌شوند.

5. انتظار برای پایان پردازش:

- برنامه منتظر می‌ماند تا پردازش داده‌ها به پایان برسد (query.awaitTermination()).

```
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 می ماند تا فرآیند استریم خاتمه پیدا کند.

نقش Apache Spark Master:

- **Spark Master** قلب کلاس تر (Apache Spark Cluster) است. این کامپوننت مسئول مدیریت منابع کلاس تر و هماهنگی بین Worker ها است.
- **Master** وظایف زیر را بر عهده دارد:
 1. **مدیریت Worker ها:** Master لیستی از Worker های موجود در کلاس تر را نگهداری می کند و وضعیت آن ها را رصد می کند.
 2. **تخصیص منابع:** Master تصمیم می گیرد که هر Task (وظیفه) به کدام Worker اختصاص داده شود.
 3. **هماهنگی اجرای Job ها:** Master Job ها (کارهای ارسالی به کلاس تر) را دریافت می کند و آن ها را به Task های کوچک تر تقسیم می کند. سپس این Task ها را به Worker ها ارسال می کند.
 4. **مانیتورینگ:** Master رابط کاربری وب (Web UI) را ارائه می دهد که از طریق آن می توان وضعیت کلاس تر، Job ها و Worker ها را مشاهده کرد.

جزئیات فایل spark-master-deployment.yaml:

- این فایل یک **Deployment** در Kubernetes ایجاد می کند که **Spark Master** را اجرا می کند.
- **تنظیمات کلیدی:**
 - **پورت ها:**
 - 7077: برای ارتباط RPC بین Master و Worker ها.
 - 8080: برای رابط کاربری وب (Web UI) که امکان مانیتورینگ کلاس تر را فراهم می کند.
 - **متغیرهای محیطی:**
 - `SPARK_MODE=master`: مشخص می کند که این Container به عنوان Master اجرا شود.
 - سایر متغیرهای محیطی (مانند `SPARK_RPC_AUTHENTICATION_ENABLED`) تنظیمات امنیتی را غیرفعال می کنند.

اهمیت Master در سیستم شما:

- نقش **هماهنگ کننده** را در کلاس تر Spark ایفا می کند. بدون Worker، Master ها نمی توانند وظایف خود را به درستی انجام دهند.
- Master همچنین **وضعیت کلاس تر** را رصد می کند و در صورت خرابی Worker ها یا نیاز به مقیاس پذیری،

واکنش مناسب نشان می‌دهد.

2. فایل spark-worker-deployment.yaml:

نقش Apache Spark Worker:

- Spark Worker واحدهای پردازشی در کلاس‌تر Spark هستند. این کامپوننت‌ها مسئول اجرای Task‌هایی هستند که توسط Master به آن‌ها محول می‌شود.
- Worker وظایف زیر را بر عهده دارد:
 1. اجرای Task‌ها: Workerها Task‌های ارسالی توسط Master را اجرا می‌کنند. این Task‌ها می‌توانند شامل پردازش داده‌ها، اجرای کدهای Spark و غیره باشند.
 2. مدیریت منابع محلی: هر Worker منابع محلی خود (مانند حافظه و CPU) را مدیریت می‌کند و به Master گزارش می‌دهد.
 3. ارتباط با Master: Workerها به طور مداوم با Master در ارتباط هستند و وضعیت خود را به آن گزارش می‌دهند.

جزئیات فایل spark-worker-deployment.yaml:

- این فایل یک Deployment در Kubernetes ایجاد می‌کند که Spark Workerها را اجرا می‌کند.
- تنظیمات کلیدی:
 - تعداد Workerها: در این فایل، Worker 2 تعریف شده است (replicas: 2).
 - متغیرهای محیطی:
 - SPARK_MODE=worker: مشخص می‌کند که این Container به عنوان Worker اجرا شود.
 - SPARK_MASTER_URL=spark://spark-master-service:7077: آدرس Master را مشخص می‌کند که Workerها به آن متصل می‌شوند.
 - SPARK_WORKER_MEMORY=2G: مقدار حافظه اختصاص داده شده به هر Worker را مشخص می‌کند.
 - SPARK_WORKER_CORES=2: تعداد هسته‌های CPU اختصاص داده شده به هر Worker را مشخص می‌کند.

ارتباط بین Worker و Master:

- Master و Workerها به طور مداوم با هم در ارتباط هستند. این ارتباط از طریق پروتکل RPC انجام می‌شود.
- نحوه کار:
 1. Workerها هنگام راه‌اندازی به Master متصل می‌شوند و منابع خود (مانند حافظه و CPU) را به Master گزارش می‌دهند.
 2. Master Job‌ها را به Task‌های کوچک‌تر تقسیم می‌کند و آن‌ها را به Workerها ارسال می‌کند.
 3. Workerها Task‌ها را اجرا می‌کنند و نتایج را به Master گزارش می‌دهند.

4. Master نتایج را جمع‌آوری می‌کند و به کاربر بازمی‌گرداند.

توضیح فایل اول: spark-master-deployment.yaml

این فایل مربوط به راه‌اندازی Spark Master در Kubernetes است.

بخش Deployment:

```
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 نشان دهنده این است که این بخش از API نسخه apps/v1 استفاده می کند.
2. `kind: Deployment`: نوع 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 می کند این Deployment با Podهایی که برچسب app: spark-master دارند کار می کند.
6. `template.metadata.labels`: برچسبی که برای Podها تعریف شده است.
7. `containers`: تعریف مشخصات کانتینر Spark Master
 - `image`: ایمیج Docker که استفاده می شود (bitnami/spark:latest).
 - `ports`: دو پورت:
 - `7077`: پورت RPC که Spark Master برای ارتباط استفاده می کند.
 - `8080`: پورت رابط وب Spark.
 - `env`: متغیرهای محیطی:
 - `SPARK_MODE: master`: حالت Spark را "master" تعیین می کند.
 - سایر متغیرها مثل `SPARK_RPC_AUTHENTICATION_ENABLED` و `SPARK_SSL_ENABLED` امنیت و رمزنگاری را غیرفعال می کنند.
 - `SPARK_LOCAL_DIRS`: مسیر ذخیره سازی موقت را به tmp/ تنظیم می کند.

بخش Service:

```

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 می کند.

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

```
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 Pod های Worker که باید اجرا شوند. اینجا مقدار 2 تنظیم شده است.
5. `spec.selector.matchLabels`: مشخص می‌کند این Deployment به Pod هایی که برچسب `app: spark-worker` دارند، مربوط است.
6. `template.spec.containers`: تعریف مشخصات کانتینر Spark Worker:
 - `image`: از ایمیج `bitnami/spark:latest` Docker 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 Core هایی که Worker استفاده می‌کند (2 عدد).
 - سایر متغیرها برای غیرفعال کردن امنیت و رمزنگاری.

3.2 - سرویس جمع‌داده (Database):

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.

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

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
```

`apiVersion: apps/v1`: نشان‌دهنده این است که این بخش از API نسخه "apps/v1" استفاده می‌کند.

1. `kind: Deployment`: این بخش نشان می‌دهد که نوع آبجکت `Deployment` است. `Kubernetes` برای `Deployment`

- مدیریت و مقیاس‌دهی Pod ها استفاده می‌شود.
2. **metadata.name**: نام این Deployment را "redis" تعیین می‌کند.
 3. **metadata.labels**: برچسب‌ها برای شناسایی این Deployment استفاده می‌شوند. اینجا برچسب `app: redis` تعیین شده است.
 4. **spec.replicas**: تعداد Pod های این Deployment را مشخص می‌کند. در اینجا 1 عدد است.
 5. **spec.selector.matchLabels**: تعیین می‌کند که این Deployment باید Pod هایی که برچسب `app: redis` دارند را مدیریت کند.
 6. **template.metadata.labels**: برچسبی که به Pod های ایجاد شده اعمال می‌شود. در اینجا `app: redis`.
 7. **template.spec.containers**: تعریف کانتینرهای این Pod.
 - **name: redis**: نام کانتینر.
 - **image: bitnami/redis:latest**: ایمیج Docker که برای کانتینر استفاده می‌شود. اینجا از 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.
 8. **volumes**: تعریف Volume هایی 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

```

apiVersion: v1: نسخه API Kubernetes.

1. **kind: Service**: این بخش نشان دهنده یک Service است. Service برای در دسترس قرار دادن Pod ها استفاده می شود.
2. **metadata.name**: نام این "Service" است.
3. **spec.selector**: تعیین می کند که این Service باید به Pod هایی که برچسب `app: redis` دارند متصل شود.
4. **spec.ports**: تعریف پورت هایی که Service فراهم می کند.
 - **protocol: TCP**: پروتکل ارتباطی (TCP).
 - **port: 6379**: پورت که توسط Service ارائه می شود.
 - **targetPort: 6379**: پورت مقصد روی Pod ها.