Thoughts Resampled

Anatoliy Plastinin's Blog

- RSS

Navigate...

- Blog
- Archives
- About

Getting started with Spark Streaming using Docker

Oct 5th, 2015 12:45 am

Stream processing technologies have been getting a lot of attention lately. You've already might heard about Kafka, Spark and its streaming extension.

If you've always wanted to try Spark Streaming, but never found a time to give it a shot, this post provides you with easy steps on how to get development setup with Spark and Kafka using Docker.

> Note: This walkthrough covers OS X and uses Homebrew, so you might want to install it first. For other platforms there won't be many differences. Please refer to corresponding software documentation for instructions for your platform.

We will use `DirectKafkaWordCount` example from spark distribution as basis for our demo. That example shows how to use Spark's Direct Kafka Stream. You can easily use another example that uses Receiver-based Approach Discussion of different ways to integrate kafka that spark provides is out of scope of this post. Please checkout kafka integration guide for more details.

Yes, doing another word count demo is boring, but our goal is to learn how to get evrything up and running together and `WordCount` suits this goal perfectly, rather than learn how to build distributed applications with Spark.

If you want just to get code, you can find complete example here.

The Code

We will use sbt for building our project. To install it run: `brew install sbt`.

Next, we need to setup sbt directory structure for the project. Unfortunately `sbt` doesn't provide command to bootstrap a project, so you can create project with your IDE like Eclipse, or use this shell script.

Now let's go to code and do some configuration.

Spark requires packaging all projects' dependencies alongside application, so we will build fat jar that contains app and all dependencies together. We will use sbt assembly plugin for that.

To set up this plugin create `project/assembly.sbt` file with following content:

assembly.sbt

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")
```

Now let's setup our build configuration, `build.sbt` file should look like:

build.sbt

```
name := "direct_kafka_word_count"

scalaVersion := "2.10.5"

val sparkVersion = "1.5.1"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion % "provided",
  "org.apache.spark" %% "spark-streaming" % sparkVersion % "provided",
  ("org.apache.spark" %% "spark-streaming-kafka" % sparkVersion) exclude ("org.spark-project.spark", "unused")
)

assemblyJarName in assembly := name.value + ".jar"
```

At the moment of writing latest version of spark is `1.5.1` and scala is `2.10.5` for `2.10.x` series. Scala 2.10 is used because spark provides pre-built packages for this version only.

We don't need to provide spark libs since they are provided by cluster manager, so those libs are marked as `provided`.

That's all with build configuration, now let's write some code. App's code in `src/main/scala/com/example/spark/DirectKafkaWordCount.scala` should look like:

src/main/scala/com/example/spark/DirectKafkaWordCount.scala

```
package com.example.spark

import kafka.serializer.StringDecoder
import org.apache.spark.{TaskContext, SparkConf}
import org.apache.spark.streaming.kafka.{OffsetRange, HasOffsetRanges, KafkaUtils}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object DirectKafkaWordCount {
  def main(args: Array[String]): Unit = {
    if (args.length < 2) {
      System.err.println(s"""
        |Usage: DirectKafkaWordCount <brokers> <topics>
        |  <brokers> is a list of one or more Kafka brokers
        |  <topics> is a list of one or more kafka topics to consume from
        |
        """.stripMargin)
      System.exit(1)
    }

    val Array(brokers, topics) = args

    // Create context with 10 second batch interval
    val sparkConf = new SparkConf().setAppName("DirectKafkaWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(10))

    // Create direct kafka stream with brokers and topics
    val topicsSet = topics.split(",").toSet
    val kafkaParams = Map[String, String]("metadata.broker.list" -> brokers)
    val messages = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
      ssc, kafkaParams, topicsSet)

    // Get the lines, split them into words, count the words and print
    val lines = messages.map(_._2)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1L)).reduceByKey(_ + _)
    wordCounts.print()

    // Start the computation
    ssc.start()
    ssc.awaitTermination()
  }
}
```

And now you can run `sbt assembly` and find `direct_kafka_word_count.jar` file in `target/scala-2.10` directory.

That's all with coding. Let's see how to run it.

Installing Docker

To run this example we need two more things: Spark itself and Kafka server.

And that's where Docker significantly helps us, since we don't need to install and configure required software, we will simply use Docker images for that.

Easiest way to get Docker working on OS X is to use docker-machine, which helps to provision Docker on virtual machines.

To set it up:

- Install VirtualBox, to run VM with Docker.
- Update homebrew to latest version `brew update`.
- Install required packages `brew install docker docker-machine docker-compose`.

Another option to install everything is to use Docker Toolbox, for more details on how to install toolbox check the official documentation.

Now we are ready to start Docker VM, run

```
docker-machine create --driver virtualbox --virtualbox-memory 2048 dev
```

This command downloads VM image with Docker host preinstalled, creates a VM named `dev` with 2Gb of memory and starts it.

Now you need configure your shell to work with Docker client, just run `eval "$(docker-machine env dev)"`. Please note that you need to run this command each time you open a new terminal.

Containers Setup

Let's specify our containers configuration using `docker-compose`. `docker-compose.yml` file defines containers and links between them. We will use following configuration:

docker-compose.yml

```
kafka:
  image: antlypls/kafka
  environment:
    - KAFKA=localhost:9092
    - ZOOKEEPER=localhost:2181
  expose:
    - "2181"
    - "9092"

spark:
  image: antlypls/spark:1.5.1
  command: bash
  volumes:
    - ./target/scala-2.10:/app
  links:
   - kafka
```

A lot of things are going on here, let's go through it step by step. This `yml` file defines two services: `kafka` and `docker`.

`kafka` service runs image based on `spotify/kafka` repository, this image provides everything we need for running kafka in one container: kafka broker and zookeeper server.

Also two environment variables are added: `KAFKA` and `ZOOKEEPER`, those variables are helpful when you run kafka CLI tools inside kafka container, you will see how to do it later.

We also expose a kafka broker port `9092` and a port for zookeeper `2181`, so linked services can access it.

Then `spark` service is defined. I've prepared an image `antlypls/spark`, which provides spark running on YARN. The image is slightly modified version of `sequenceiq/spark` repository, with `spark 1.5.1` and without a few packages that we don't need for this demo.

We specify `bash` as command, since we want to have interactive shell session within the spark container. `volumes` option mounts build directory into the spark container, so we will be able to access `.jar` right in the container. At the end `kafka` service is linked to `spark`.

And now we are ready to run everything together.

Let's start all containers with `docker-compose`: `docker-compose run --rm spark` this starts kafka and then spark and logs us into spark container shell. The `--rm` flag makes `docker-compose` to delete corresponding `spark` container after run.

But before running `DirectKafkaWordCount` app, we need to create a topic in a kafka broker that we are going to read from. Kafka distribution contains a few useful tools to manipulate topics and data: create/list topics, write text messages into a topic and etc. And we can run those tools within kafka container. To do that open a separate terminal session and run:

```
docker exec -it $(docker-compose ps -q kafka) bash
```

And now let's create a topic in kafka:

```
kafka-topics.sh --create --zookeeper $ZOOKEEPER --replication-factor 1 --partitions 2 --topic word-count
```

You can check that new topic has been created by running commands

```
$ kafka-topics.sh --list --zookeeper $ZOOKEEPER
$ kafka-topics.sh --describe --zookeeper $ZOOKEEPER --topic word-count
```

Keep this shell session open, we will use it to add messages to the topic.

Now go back to spark container shell and run

```
spark-submit \
--master yarn-client \
--class com.example.spark.DirectKafkaWordCount \
app/direct_kafka_word_count.jar kafka:9092 word-count
```

Here we launch our application in `yarn-client` mode because we want to see output from the driver.

You might see a lot of logs written to output, that is useful for debugging, but it might be hard to see actual app output. You could use following settings for `log4j` if you wanted to disable those debugging logs:

log4j.properties

```
log4j.rootCategory=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.threshold=ERROR
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n
```

Replace `$SPARK_HOME/conf/log4j.properties` file with one provided above. Or simply put it somewhere in container, e.g. in shared `app` directory and run app like

```
spark-submit \
--master yarn-client \
--driver-java-options "-Dlog4j.configuration=file:///app/log4j.properties" \
--class com.example.spark.DirectKafkaWordCount \
app/direct_kafka_word_count.jar kafka:9092 word-count
```

Note, that `docker-compose` updates `hosts` file in running containers, so linked services can be accessed using service name as a hostname. That's why in our demo we simply use `kafka:9090` as a broker address.

Now let's add some data into the topic, just run following in the `kafka` container

```
kafka-console-producer.sh --broker-list $KAFKA --topic word-count
```

And for input like this:

```
Hello World
!!!
```

You should see output from spark app like:

```
-------------------------------------------
Time: 1234567890000 ms
-------------------------------------------
(Hello,1)
(World,1)
(!!!,1)
```

`docker-compose` doesn't stop/delete linked containers when `run` command exits. To stop linked containers run `docker-compose stop`, and `docker-compose rm` to delete them.

And that's it, you have all set up for developing Spark Streaming apps.

Happy hacking with Spark!

Posted by Anatoliy Plastinin Oct 5th, 2015 12:45 am docker, kafka, spark

Running spark-shell in browser with Apache Mesos and Marathon »

Comments

Recent Posts

- spark-shell without Spark
- When Data Driven App Smells Bad
- Spark SQL and Parquet files
- Processing JSON data with Spark SQL
- How to Write Data into Parquet