**Installation**

**A. JDK Setup**

1. Start the JRE installation and hit the “Change destination folder” checkbox, then click 'Install.'



2. Change the installation directory to any path without spaces in the folder name. E.g. C:\Java\jre1.8.0\_xx\. (By default it will be C:\Program Files\Java\jre1.8.0\_xx), then click 'Next.'

3. Now open the system environment variables dialogue by opening Control Panel -> System -> Advanced system settings -> Environment Variables.

4. Hit the New User Variable button in the User variables section, then type JAVA\_HOME in *Variable name*and give your jre path in the *Variable value.* It should look like the below image:

*(Java path and version may change according to the version of Kafka you are using)*

5. Now click OK.

6. Search for a Path variable in the “System Variable” section in the “Environment Variables” dialogue box you just opened.

7. Edit the path and type “;%JAVA\_HOME%\bin” at the end of the text already written there, just like the image below:



8. To confirm the Java installation, just open cmd and type “*java –version.”*You should be able to see the version of Java you just installed.



If your command prompt somewhat looks like the image above, you are good to go. Otherwise, you need to recheck whether your setup version matches the correct OS architecture (x86, x64), or if the environment variables path is correct.

**B. ZooKeeper Installation**

1. Go to

E:\software\A08\file\apache-zookeeper-3.5.6-bin.tar\apache-zookeeper-3.5.6-bin\apache-zookeeper-3.5.6-bin\conf

2. Rename file “zoo\_sample.cfg” to “*zoo.cfg”*

3. Open zoo.cfg in any text editor, like Notepad; I prefer Notepad++.

4. Find and edit dataDir=/tmp/zookeeper to :\zookeeper-3.5.6\data

5. Add an entry in the System Environment Variables as we did for Java.

a. Add ZOOKEEPER\_HOME = C:\zookeeper-3.5.6 to the System Variables.

b. Edit the System Variable named “Path” and add ;%ZOOKEEPER\_HOME%\bin;

6. You can change the default Zookeeper port in zoo.cfg file (Default port 2181).

6a) To change the admin server port number, go to zoo.cfg file, add

admin.serverPort=7070

7. Run ZooKeeper by opening a new cmd and type zkserver.

8. You will see the command prompt with some details, like the image below:



Congratulations, your ZooKeeper is up and running on port 2181!

**C. Setting Up Kafka**

1. Go to your Kafka config directory. For me its *C:\kafka\_2.11-0.9.0.0\config*

2. Edit the file “server.properties.”

3. Find and edit the linelog.dirs=/tmp/kafka-logs” to “log.dir= C:\kafka\_2.11-0.9.0.0\kafka-logs*.*

4. If your ZooKeeper is running on some other machine or cluster you can edit *“zookeeper.connect:2181”* to your custom IP and port. For this demo, we are using the same machine so there's no need to change. Also the Kafka port and broker.id are configurable in this file. Leave other settings as is.

5. Your Kafka will run on default port 9092 and connect to ZooKeeper’s default port, 2181.

**D. Running a Kafka Server**

*Important: Please ensure that your ZooKeeper instance is up and running before starting a Kafka server.*

1. Go to your Kafka installation directory: *C:\kafka\_2.11-0.9.0.0\*

Rename dir to kafka

1. Open a command prompt here by pressing *Shift + right click*and choose the “Open command window here” option).

2a. In the command prompt set java memory

Change java heap options in kafka-sever-start.bat

set JAVA\_OPTS=-Xms512m -Xmx512m -XX:MaxPermSize=256m

3. Now type .\bin\windows\kafka-server-start.bat .\config\server.properties and press Enter.

.\bin\windows\kafka-server-start.bat .\config\server.properties

Image title

4. If everything went fine, your command prompt will look like this:

  
5. Now your Kafka Server is up and running, you can create topics to store messages. Also, we can produce or consume data from Java or Scala code or directly from the command prompt.

**E. Creating Topics**

1. Now create a topic with the name *“test”* and a replication factor of 1, as we have only one Kafka server running. If you have a cluster with more than one Kafka server running, you can increase the replication-factor accordingly, which will increase the data availability and act like a fault-tolerant system.

2. Open a new command prompt in the location *C:\kafka\_2.11-0.9.0.0\bin\windows.*

3. Type the following command and hit Enter:

kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test



**F. Creating a Producer and Consumer to Test Server**

1. Open a new command prompt in the location *C:\kafka\_2.11-0.9.0.0\bin\windows*

2. To start a producer type the following command:

kafka-console-producer.bat --broker-list localhost:9092 --topic test

3. Again open a new command prompt in the same location as *C:\kafka\_2.11-0.9.0.0\bin\windows*

4. Now start a consumer by typing the following command:

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning

5. Now you will have two command prompts, like the image below:



6. Now type anything in the producer command prompt and press Enter, and you should be able to see the message in the other consumer command prompt.



7. If you are able to push and see your messages on the consumer side, you are done with Kafka setup.

**Some Other Useful Commands**

1. List Topics: kafka-topics.bat --list --zookeeper localhost:2181
2. Describe Topic: kafka-topics.bat --describe --zookeeper localhost:2181 --topic [Topic Name]
3. Read messages from the beginning: kafka-console-consumer.bat --zookeeper localhost:2181 --topic [Topic Name] --from-beginning
4. Delete Topic: kafka-run-class.bat kafka.admin.TopicCommand --delete --topic [topic\_to\_delete] --zookeeper localhost:2181

**Kakfa cluster**

#### [Step 6: Setting up a multi-broker cluster](https://kafka.apache.org/quickstart#quickstart_multibroker)

For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our cluster to three nodes (still all on our local machine).

First we make a config file for each of the brokers (on Windows use the copy command instead):

|  |  |
| --- | --- |
| 1  2 | > cp config/server.properties config/server-1.properties  > cp config/server.properties config/server-2.properties |

Now edit these new files and set the following properties:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | config/server-1.properties:      broker.id=1      listeners=[PLAINTEXT://:9093](NULL)      log.dirs=/tmp/kafka-logs-1    config/server-2.properties:      broker.id=2      listeners=[PLAINTEXT://:9094](NULL)      log.dirs=/tmp/kafka-logs-2 |

The broker.id property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each other's data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:

|  |  |
| --- | --- |
| 1  2  3  4 | > bin/kafka-server-start.sh config/server-1.properties &  ...  > bin/kafka-server-start.sh config/server-2.properties &  ... |

Now create a new topic with a replication factor of three:

|  |  |
| --- | --- |
| 1 | > bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 1 --topic my-replicated-topic |

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:

|  |  |
| --- | --- |
| 1  2  3 | > bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic  Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:      Topic: my-replicated-topic  Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,2,0 |

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

* "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
* "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
* "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in my example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

|  |  |
| --- | --- |
| 1  2  3 | > bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic test  Topic:test  PartitionCount:1    ReplicationFactor:1 Configs:      Topic: test Partition: 0    Leader: 0   Replicas: 0 Isr: 0 |

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:

|  |  |
| --- | --- |
| 1  2  3  4  5 | > bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-replicated-topic  ...  my test message 1  my test message 2  ^C |

Now let's consume these messages:

|  |  |
| --- | --- |
| 1  2  3  4  5 | > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic  ...  my test message 1  my test message 2  ^C |

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

|  |  |
| --- | --- |
| 1  2  3 | > ps aux | grep server-1.properties  7564 ttys002    0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...  > kill -9 7564 |

On Windows use:

|  |  |
| --- | --- |
| 1  2  3  4 | > wmic process where "caption = 'java.exe' and commandline like '%server-1.properties%'" get processid  ProcessId  6016  > taskkill /pid 6016 /f |

Leadership has switched to one of the followers and node 1 is no longer in the in-sync replica set:

|  |  |
| --- | --- |
| 1  2  3 | > bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic  Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:      Topic: my-replicated-topic  Partition: 0    Leader: 2   Replicas: 1,2,0 Isr: 2,0 |

But the messages are still available for consumption even though the leader that took the writes originally is down:

|  |  |
| --- | --- |
| 1  2  3  4  5 | > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic  ...  my test message 1  my test message 2  ^C |

#### [Step 7: Use Kafka Connect to import/export data](https://kafka.apache.org/quickstart#quickstart_kafkaconnect)

Reading data from the console and writing it back to the console is a convenient place to start, but you'll probably want to use data from other sources or export data from Kafka to other systems. For many systems, instead of writing custom integration code you can use Kafka Connect to import or export data.

Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs *connectors*, which implement the custom logic for interacting with an external system. In this quickstart we'll see how to run Kafka Connect with simple connectors that import data from a file to a Kafka topic and export data from a Kafka topic to a file.

First, we'll start by creating some seed data to test with:

|  |  |
| --- | --- |
| 1 | > echo -e "foo\nbar" > test.txt |

Or on Windows:

|  |  |
| --- | --- |
| 1  2 | > echo foo> test.txt  > echo bar>> test.txt |

Next, we'll start two connectors running in *standalone* mode, which means they run in a single, local, dedicated process. We provide three configuration files as parameters. The first is always the configuration for the Kafka Connect process, containing common configuration such as the Kafka brokers to connect to and the serialization format for data. The remaining configuration files each specify a connector to create. These files include a unique connector name, the connector class to instantiate, and any other configuration required by the connector.

|  |  |
| --- | --- |
| 1 | > bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties |

These sample configuration files, included with Kafka, use the default local cluster configuration you started earlier and create two connectors: the first is a source connector that reads lines from an input file and produces each to a Kafka topic and the second is a sink connector that reads messages from a Kafka topic and produces each as a line in an output file.

During startup you'll see a number of log messages, including some indicating that the connectors are being instantiated. Once the Kafka Connect process has started, the source connector should start reading lines from test.txt and producing them to the topic connect-test, and the sink connector should start reading messages from the topic connect-test and write them to the file test.sink.txt. We can verify the data has been delivered through the entire pipeline by examining the contents of the output file:

|  |  |
| --- | --- |
| 1  2  3 | > more test.sink.txt  foo  bar |

Note that the data is being stored in the Kafka topic connect-test, so we can also run a console consumer to see the data in the topic (or use custom consumer code to process it):

|  |  |
| --- | --- |
| 1  2  3  4 | > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning  {"schema":{"type":"string","optional":false},"payload":"foo"}  {"schema":{"type":"string","optional":false},"payload":"bar"}  ... |

The connectors continue to process data, so we can add data to the file and see it move through the pipeline:

|  |  |
| --- | --- |
| 1 | > echo Another line>> test.txt |

You should see the line appear in the console consumer output and in the sink file.