# The QualiMaster infrastructure – File formats

The following file formats are discussed in this document

* Account file
* Pipeline implementation mapping file
* Profiling control files

# Account File

The account file is a central store for the accounts on a QualiMaster infrastructure installation. Basically, user names can be real user names or “virtual” (storm node specific) user names mapped to real user names. To apply this service, the infrastructure shall define a file /var/nfs/accounts.properties (due to the infrastructure setting dfs.path) with a structure like

twitter/katerina.user=katerina

twitter/katerina.passwd=mySecret

twitter/katerina.accessToken=aabbbcc

holger.passwd=dontknow

This maps the virtual user twitter/katerina to katerina with given password (standard passwd entry) and algorithm-specific value accessToken. Further, it defines the password for the real user holger, i.e., virtual user name is the same as real user name. For accessing this data, use

PasswordEntry entry = PasswordStore.getEntry("twitter/katerina");

entry.getUserName(); // -> katerina

entry.getPassword(); // -> mySecret

entry.getValue("accessToken"); // -> aabbbcc

entry = PasswordStore.getEntry("holger");

entry.getUserName(); // -> holger

entry.getPassword(); // -> dontknow

The password store is defined in the data management layer in eu.qualimaster.dataManagement.accounts.PasswordStore. The password store also provides a pluggable authentication mechanism that can, e.g., be replaced by LDAP (see also adaptation layer protocol authentication, QM-IConf user authentication).

# Pipeline Implementation Mapping File

A pipeline implemented as a Storm topology needs a, which defines the relation between elements of the configuration and implemented components, in particular as the names in the configuration may contain spaces etc. and implementation names may be completely different depending on the infrastructure instantiation process. Moreover, it describes the mapping of the abstract logical structure of a pipeline to its physical implementation in a Storm topology as far as needed for the infrastructure (e.g., hardware implementation needs Bolt and Spout, an algorithm can be a sub-topology). Please note that manually implemented sub-topologies are not described in a mapping file but recorded dynamically while creating the topology structure (see development guidelines, while generated sub-topologies shall be described in a mapping file as given below.

Therefore, we distinguish between

* configured logical names used in the infrastructure configuration (as given by the user)
* implementation names used in the code (as defined by the instantiation process) to declare spouts and bolts to Storm
* fully qualified class names (Java class names with package)

This page describes the structure and the elements of the mapping file, which is typically generated as part of the infrastructure instantiation process.

## Structure

The pipeline mapping file is an XML file following the Syntax below:

<mapping>

<pipeline>+

<node>+

<component/>+

</node>

<algorithm>+

<component/>?

</algorithm>

<parameter/>+

<pipeline/>\*

</pipeline>

</mapping>

With the following cardinalities (given after the start elements, end elements have same cardinality):

* \* - optional (0 - 1)
* + - optional (0+)

## Elements

* mapping is just the root element for a mapping file and has no further attributes. A mapping may contain several pipeline mappings. Notice that currently there is only one mapping per pipeline. It can be adjusted to consider multiple pipelines in one mapping if needed.
* pipeline defines the mapping of a single pipeline (implemented by the Jar artifact containing the mapping file). A pipeline has the following attributes:
  + name: the logical name of the pipeline (must match the name of the implementing Jar, in particular if copied manually to the infrastructure - see infrastructure setting repository.artifacts.pipeline.download)
  + class: the fully qualified Java class name implementing the pipeline topology.
* node represents a pipeline node:
  + name: the logical name of the pipeline node
  + type: the type of the pipeline node, may be SOURCE, FAMILY, SINK, DATA\_MGT, UNKNOWN (in any combination of upper and lower case characters)

A node contains at least (typically exactly) one component detailing the implementation.

* algorithm represents an specific algorithm implementation. Simple Java algorithms are just stated by a single non-nested algorithm element. Distributed algorithms (sub-topologies) as well as hardware algorithms enumerate the implementing components as nested elements (without the alternative attribute).
  + name: the logical algorithm name
  + implName: the implementation name of the algorithm
  + class: the qualified name of the Java class implementing the algorithm

In case of a sub-topology, an algorithm can contain components directly or components nested in nodes. Sub-pipeline components shall have a name prefixed with the algorithm name.

* component within node represents an implementation component. Typically, a node has one implementation component, e.g., a Bolt or a Spout and distributed algorithms (sub-topologies) are detailed as component elements of the respective algorithm.
  + name: The implementation name of the component.
  + container: The logical name of the containing element, either the pipeline or the algorithm.
  + class: the qualified name of the Java class
  + receiver: indicates whether this component receive and handle signals (true/false)
  + thrift: does this component perform monitoring on its own by sending monitoring events (false - the default case) or shall the monitoring layer rely as far as possible on thrift information (true)
  + tasks: the initially number of configured tasks for this component (used for parallelization tooling)
  + alternativen: Indicates the different alternative algorithms realized by this component implementing an algorithm family. The *n* is supposed to be replaced by a natural number starting at 0 for enumerating the alternatives. The value of this attribute must be a **valid logical name** of an algorithm as stated by the algorithm elements described below, e.g., alternative0=“swAlg“alternative1=“hwAlg“. Currently not needed for sources and sinks. Not required for components nested in algorithms as they are not supposed to have alternatives.
  + type: in case of nodes for hardware integration, please add type=“HARDWARE”. Needed for loose network-based integration of sub-topologies. Ignored for all other values.
* parameter defines the re-routing of a parameter from a given implementation component to another one. This is needed in case of generated sub-topologies, so that information about changing parameters can be sent to the correct implementing Storm component. Basically, a manual implementation would realize this as part of the sub-topology implementation, but this requires two signals, one from the infrastructure to the sub-topology and a forward event to the implementing Storm component. Via the parameter mapping here, the signal can be sent directly in around half of the signal sending time of the manual implementation.
  + name: the logical name of the parameter
  + node: the logical name of the primary receiver component (where the signal typically would go to)
  + receiver: the logical name of the actual receiver component or the implementation name of a component implementing an algorithm (indicates the re-direction)
* pipeline within a pipeline names a sub-pipeline to be started before. Just the name is considered.

## Location

The mapping file must always be given as mapping.xml in the root folder of the implementing Jar artifact.

# Profiling control files

## Idea

For making qualified automated decisions about similar algorithms in the same family, we may relay on statements of the algorithm designer / implementer (guarantees) or on actual measurements of the implemented algorithm. We focus on the latter, also as implementation problems in the algorithm may cause an algorithm to behave (partly) different than theoretical guarantees. Thus, such profiling may be one form of validation of an algorithm against its guarantees.

Basically, profiling just means that for a certain set of given parameters (including parallelization and distribution) runtime measures of the algorithm are obtained. However, a time series of measurements is not really helpful making comparisons and even in predicting the behavior in gaps where no profiling happens. Thus, we plan to apply one (out of five feasible ones) postprocessing approach to obtain statistical profiles.

## Ingredients

An algorithm comes along with information about

* data sets: for profiling (potentially including peaks) in a format accepted by the replay mechanism so that even data ingestion speed can be changed during profiling.
* profile control: information on the most relevant parameters

This information may be given for an individual algorithm or an algorithm may point to a common data set / profile control information for a family. As part of packaging the algorithm and deploying it to the Maven repository, this additional information shall also be deployed (automatically) into a specific artifact, i.e., profiling information is separated from the actual algorithm implementation to avoid overhead.

Profiling an algorithm happens by issuing an infrastructure command specifying the family / algorithm. The infrastructure

* derives a temporary configuration similar to the pipeline in the figure above, i.e., it uses the given algorithm, clones the family and links in the algorithm as the only algorithm of that family (to prevent adaptation), creates a source fitting to the family and links it against an implementation that will be generated while running the infrastructure instantiation process.
* runs the pipeline instantiation process just for that single pipeline. Thereby, it generates a data source implementation that matches the profiling pipeline source and links it against the data replay mechanism. As the data replay mechanism just passes back strings, the generated source must convert the strings into the correct tuple instances. Therefore, it feeds the tuple type into the string serialization/deserialization mechanism provided by the data management layer (whereby specific serializers have already been created through the full pipeline instantiation along with the protobuf hardware serializers).
* loads data sets into the cluster file system (HDFS or NFS) and defines the respective data set as input to the data replay mechanism instance used in the profing pipeline,
* interprets and starts / executes the pipeline as long as defined by the profile control information. Thereby, the monitoring layer
* collects the relevant data from generic infrastructure monitoring and (**if provided**) from the algorithms.
* writes raw monitoring traces.

These traces are then taken for postprocessing to obtain the statistical profiles.

## How to...?

### Define profiling data?

In a file with lines timestamp, payload whereby the timestamp is a long timestamp and payload contains the string representation of exactly the fields in the tuple/field sequence defined in the configuration of the algorithm. If the respective algorithm accepts **multiple tuple types**, the the first entry after the timestamp must be the name of the tuple addressed so that various data streams can be identified from a single data file. Please note that (according to the original implementation) the first line of the data is just taken to identify the initial timestamp and the payload is ignored. The file is just valid for the specific algorithm / family as long as the respective configuration does not change. The data shall be stored in a file called profile.data. Multiple data files, e.g., with different input rates are given by adding an index number after a minus as separator, e.g. profile-1.data, profile-2.data. It is recommended to also keep a readme file explaining the contents of the different data files.

See an example for a simple data file for the hy-preprocessor above. Below is an example for a more complex algorithm, here the correlation computation, which depends on the output of the hy-preprocessor (tuple name preprocessedStream) and the original data source providing the symbol list (tuple name symbolList):

1460704494000,preprocessedStream,IESJ,1000000000108,51.37,1

1460704495000,symbolList,10,IESJ,KIOG,QESX,HEXC,YBAV,ULFM,NTGO,AKCB,WCPY,PLWD

1460704495108,preprocessedStream,IESJ,1000000000108,51.37,1

1460704496108,preprocessedStream,KIOG,1000000001108,11.22,4

As this data file consists of multiple streams for multiple input tuples, the entry directly after the timestamp defines the target tuple type via its configured tuple name. The data file is structured as follows:

* The first line provides the initial timestamp and the following character is interpreted as separator (the comma). The remainder is ignored.
* The second line provides the symbol list for the correlation computation. As String lists (configured type StringListType are serialized by the number of elements followed by the elements, we have here 10 (shortened symbol list) as the number of elements followed by 10 String entries.
* The following lines provide data for the tuple type preprocessedStream, whereby the payload entries follow exactly the configured tuple type, i.e., symbolId, timestamp, value, volume.

Please note that there are no whitespaces between the individual entires.

### Define the profiling control?

Multiple runs of replaying data and monitoring the algorithm may be needed to cover at least basic important variations of Storm processing settings and parameter ranges. We provide a simple language for describing the control in a file called profile.ctl. The contents looks like

processing workers = 1,2,3,4

processing tasks = 1,2,2,2

processing executors = 1,1,2,3

parameter window = 400,500,600,1000

Meaning that the algorithm shall be profiled with 4 worker/task/executor combinations namely (1,1,1), (2,2,1), … and four different window settings, i.e., 16 times in total. Please note that we currently operate with a fixed separator (,) and that no space between the individual entries is allowed.

Workers denote the workers for the algorithm, i.e., the infrastructure adds one for the source so that the number of workers here may also be 0 to indicate that source and algorithm shall run on the same worker. Although tasks must be stated explicitly, here also the value 0 can be used in order to indicate that the Storm default value shall be taken. Similarly for executors.

Optionally, the file may start with an import of a common (family) profiling artifact, e.g.,

import eu.qualimaster:correlationFamily:3.1-SNAPSHOT

which takes over the contents of the profile control file there (ignoring a given import) as well as the data file, but the actual algorithm may override control statements and data file (which then takes precedence).

If just the profiling data but not the control settings shall be taken over, use

import data eu.qualimaster:correlationFamily:3.1-SNAPSHOT

### Package the data?

Both files shall be deployed in a zip file with type profiling. This can be achieved through the assembly descriptor in eu.qualimaster:AlgorithmAssemblies:0.5.0-SNAPSHOT. Therefore, both files shall be located in a top-level project directory called profiling and the assembly descriptor must be specified in the pom file in the section for the Maven Assembly Plugin. The readme file at [github](https://github.com/QualiMaster/Infrastructure/tree/master/maven/AlgorithmAssemblies) explains details, e.g., how to apply the descriptor to usual maven projects, to module builds and also how to disable it in sub-modules.

As usual, deployment happens automatically through Jenkins. Deployed profiling data artifacts are not automatically downloaded by maven so that (further) pollution of local repositories shall not happen.