# The QualiMaster infrastructure – Development Guidelines

**All components**

* Logging shall be done through the provided Apache logging framework as this reduces potential dependency conflicts.

**Storm executors (Bolts, Spouts)**

* Storm hint: Spouts that do not emit data shall at least sleep(1) to reduce CPU load.
* Storm executors must be serializable (through Java POJO or Kryo) in order to enable distribution. Check attributes whether they are serializable (writing through an ObjectOutputStream can help detecting problems) and if attributes are ultimately not used, mark them with the modifier **transient**. QM algorithms are declared as transient during code generation.
* For state transfer during dynamic migration of Bolts and Spouts, please consider that only transient attributes are transferred. For allowing state transfer on algorithms, algorithms must be serializable and consider the transient modifier. If explicit control over the state transfer shall be taken, use the @Stateful class and the @PartOfState attribute annotation. @Stateful declares all non-transient attributes as part of the state unless specified differently. @PartOfState can define the integration strategy, currently
  + DEFAULT: just take over if there is no current value)
  + CLEAR\_AND\_FILL: creates collection if it does not exist, clears first and fills then from state)
  + MERGE: creates collection if it does not exist, adds all from state)
  + MERGE\_AND\_KEEP\_OLD: creates collection if it does not exist, adds all from state but preserves existing entries
* Data used in storm tuples must be serializable (see above).
* Don't execute functional code rather than plain initialization in constructors. Storm foresees prepare and cleanup methods to be overridden and used appropriately.
* Don't call functional code while building up a topology. Topology code is run once on a central machine (Nimbus), terminates and is removed from memory (in plain Storm as well as in the QM infrastructure). Keeping references to Storm executors is useless, in particular as Storm executors are (re)instantiated when a topology starts and may be distributed then through serialization so that the reference you have may work in a local test cluster but not on a real cluster. Thus, initializing parameters through calling Storm executors may lead to an effect on a local cluster but not in a real cluster. Use events there, in particular the transparent event mechanism provided by the QM StormCommons library, which allows to alternatively and transparently distribute these events through the QM event bus. Using the provided BaseSignalSpout and BaseSignalBolt requires overriding methods to call the already defined (super) methods!
* Cleaning up resources in Bolts (cleanup) and Spouts (close) requires calling the already defined (super) methods for complete cleanup (if possible, see below).
* Basically, Storm defines methods for cleaning up resources in Bolts (cleanup) and Spouts (close). However, currently it is not guaranteed by Storm that these methods are called during the kill process of a topology (akin to Java finalizers when the JVM is terminated by a kill signal of the operating system). Thus, the infrastructure now informs executors derived from 'BaseSignalSpout' and 'BaseSignalBolt' about an upcoming shutdown of the pipeline through signals. Akin to parameters and algorithm change, there is now a method notifyShutdown which receives the event. To avoid that subclasses disable already implemented shutdown steps, notifyShutdown calls the (empty) prepareShutdown, which shall be overridden by new Bolts and Spouts.
* Bolts and spouts handling algorithm changes shall respond after successful execution to the respective signal by an AlgorithmChangedMonitoringEvent. Similarly, bolts and spouts handling a parameter change shall respond after successful execution by a ParameterChangedMonitoringEvent. This holds particularly for sub-topologies.
* Signal sending will be deferred until pipeline is in state “initialized” so that no startup/initialization messages to (handcrafted) subtopologies get lost.
* To enable load shedding, the respective Bolt/Spout implementation shall call enabled(Object) on the respective tuple instance to be tested for shedding.
* Replacing the old monitoring probe approach (which did not cover handcrafted sub-topologies), the following rules apply:
  + Source (refining BaseSignalSpout): Must call startMonitoring() and endMonitoring() at the beginning and end of the nextTuple method. Emitting tuples is tracked internally.
  + Processing Elements (refining BaseSignalBolt): Shall call startMonitoring() and endMonitoring() at the beginning and end of the execute method. Execution and tuple emitting is tracked anyway internally, but so monitoring plugins (QM-IConf use cases) can be considered.
  + Sinks (refining BaseSignalBolt): Shall call startMonitoring() and endMonitoring() at the beginning and end of the execute method. Execution is tracked anyway internally, but so monitoring plugins (QM-IConf use cases) can be considered. Importantly, call emitted(Object) when information is passed to the sink implementation as this cannot be tracked internally.
* For writing logs into a file, especially logging the data from tasks, StormCommons provides a DataLogger class (eu.qualimaster.common.logging.DataLogger). It returns a PrintWriter based on the given path/fileName.

**Pipelines**

Pipelines are realized in terms of (generated) Storm topologies.

* Inspired by the Storm examples, the implementation topology of a pipeline is instantiated via public static void main(String[]), so that it also can be started standalone, i.e., not through the infrastructure. Passing in arguments runs the topology in cluster mode, passing no arguments causes submission to a local cluster.
* If not given from external (in terms of a sub-topology), a topology shall use the RecordingTopologyBuilder for creating Storm executors. Reasons:
  + While most of the operations are just passed through, this specific topology builder takes into account dynamic PipelineOptions and changes workers, tasks, executors and parameters at start of the pipeline. This simplifies experiments and profiling. Therefore call in main first options = new PipelineOptions(args); passing in the arguments received and pass the options instance then to the instance of RecordingTopologyBuilder that you create instead of TopologyBuilder for defining the pipeline.
  + A storm topology needs to know about the QualiMaster infrastructure, at least the event bus for dynamically informing the infrastructure about changes in the pipeline etc. Therefore, execute the following lines at the beginning of topology creation that take over information from the QualiMaster infrastructure Configuration into the Storm topology config:
  + config.put(Configuration.HOST\_EVENT, Configuration.getEventHost());
  + config.put(Configuration.PORT\_EVENT, Configuration.getEventPort());

config.put(Configuration.EVENT\_DISABLE\_LOGGING, Configuration.getEventDisableLogging());

* + As long as sub-topologies are not configured and generated, the RecordingTopologyBuilder can record the structure of such sub-topologies and inform the infrastructure so that at least monitoring can happen correctly. Therefore, call builder.startRecording(algorithmName) before including the sub-topology and builder.stopRecording(algorithmName) after. In order to inform the infrastructure about the collected information, call at the end of creating the topology builder.close(topologyName, config);, whereby config is the Storm topology configuration.
* A deployable pipeline must have a mapping file (see file formats document), 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.
* The settings folder of a deployable pipeline jar (on top-level) may contain the summary of the settings that an external application can rely on, i.e., the external interface of a pipeline. Setting files are given in XML format (see file formats).
* The profiles folder of a deployable pipeline jar (on top-level) may contain profiles for the algorithms that have been collected during profiling or previous runs of the pipeline. The profiles folder must contain the pipeline=<name> folder of that pipeline including its sub-folders for all relevant algorithms following the conventions of the algorithm profile prediction. Given information will be copied as an initial population into the respective folder of the running cluster upon pipeline start if files/folders do not already exist. If a re-deployment shall be forced, the respective folder on the cluster must be deleted manually.

**Algorithm Families / Algorithms**

* Algorithms are typically not serialized, so arbitrary Java code can be used. If state transfer during re-parallelization is important, please consider the rules for Bolts/Spouts.
* Algorithms have a common as well as a pipeline specific interface and must implement both. Akin to Storm executors, no functional code except for initialization shall be carried out in the constructors. Specific methods called by the generated Bolts are provided, which shall be overriden. This is in particular true for switch status, which also notifies an implementation about a close by termination of the pipeline (cleanup your allocated resources).
* In case that may fail, please catch all potential exceptions and turn them into a eu.qualimaster.pipeline.DefaultModeException (as of rev 2620, please read the JavaDoc). The generated pipeline (as of rev XXXX) catches the exception and switches processing there into default mode, i.e., for each input tuple, an output tuple with Java default values will be emitted.
* Algorithm specific types to be used in input/output tuples shall not define inner classes, also not static inner classes. This may cause runtime trouble ( [InvalidClassException: no valid constructor](http://stackoverflow.com/questions/3575436/why-do-i-get-an-invalidclassexception-no-valid-constructor-when-inherit-from-a)) to the Storm serialization. Thus, serialization of specific types shall also be validated and tested. StormCommons provides support for obtaining a Storm-like kryo instance in StormTestUtils.
* If algorithms define own types to be passed along with the pipeline, they must be
  + Defined in the configuration model so that the generation can properly include them.
  + Shall provide a kryo serializer to allow the pipeline to run at higher throughput (if fast serialization is enabled for the pipeline). Also the kryo serializer must be defined along with the type in the configuration model.
  + Must provide a .proto file for the algorithm specific types (in particular if the type shall be handled by hardware algorithms), execute the protobuf generator in the Maven script and provide a type serializer for the protobuf types (example: eu.qualimaster.serializer.StringListSerializer in StormCommons). In particular, protobuf and kryo serializers can be combined in one class and defined in the configuration as one type.
* Manifests allow simplified configuration of Algorithms in QM-IConf with just some simple clicks. For Java implementations, QM-IConf reads the manifest from META-INF/manifest.xml and performs a class file analysis. For Hardware-Algorithms, a full Manifest must be specified. For details, see the file formats document regarding manifests.
* For profiling, algorithms shall provide a profiling script detailing the most important parameter variations / execution settings and a data file providing normalized input for replay during profiling. See the file format document for details on profiling and the technical background.
* For the Java simple algorithm, the calculate method has been extended to handle multiple output types and each output can carry multiple item results. Before setting the values for respective results, the clear method must be called to clean up the previously cached results. And if there is no output for the output type, the noOutput method shall be called.
* Don't implement the interfaces for input/output types yourself. Implementations are generated and available through the pipeline interfaces artifact.
* Algorithms shall free resources and stop all loops when the shutdown of pipeline happens. For software algorithms, this can be done by checking the TERMINATING state in the switchState method and free resources when this state is met. The switchState might be executed twice as it is also included in the storm cleanup method. For hardware algorithms, a close message will be sent through socket to inform the hardware to release resources.
* Please do not delay the response of an algorithm, e.g., by waiting or sleeping for certain data as this increases the response time and affects the response time, latency and capacity / load measurements of Storm and the infrastructure. In extreme cases, you may accidentally cause runtime adaptations to counter high load situations. If some form of timed synchronization by sleeping / waiting is needed, sleep / wait at maximum for the average response time that the algorithm would take for processing and if the synchronization condition is not met, indicate that the algorithm returns no result, i.e., fraction the sleeping / waiting by non-productive executions.
* The profiles folder of a deployable algorithm (on top-level) may contain profiles for the algorithm that have been collected during profiling or previous runs of the pipeline. The profiles folder must contain the sub-folders of the algorithm=<name> folder of that algorithm including its sub-folders following the conventions of the algorithm profile prediction. The respective files can be obtained during a profiling run for that algorithm and must be downloaded from the profiling cluster and added to the profiles folder.

**Algorithms vs. Sub-topologies**

A sub-topology represents an algorithm implementation, requires distributed computation over Storm and shall be considered in the main pipeline a single unit. The single unit notion is in particular important for switching among algorithm, as a sub-topology will always be subject to switching as a whole. Please avoid programmed (legacy) sub-topologies and model sub-topologies as re-usable sub-pipelines in the configuration adding them as an algorithm to the respective algorithm families.

If a sub-topology shall be scalable regarding its resource consumption during pipeline startup or shutdown, the class implementing the sub-topology must implement the IScalablePipeline interface in order to indicate how scaling requests will affect the contained (from the configuration not visible) bolts and spouts. Implementation of scaling capabilities and resource demands given in the configuration must be consistent.

**Sources / Sinks**

* Sources and sinks are typically not serialized, so arbitrary Java code can be used.
* connect and disconnect shall be safe to be called again, i.e., calling connect after connect shall not have any effect, akin for disconnect. This shall ease the transition to dynamic pipeline startup.
* The respective getData method shall return null if there is no data, also in the disconnected case.
* Sources and sinks have a common as well as a pipeline specific interface and must implement both. Akin to Storm executors, no functional code except for initialization shall be carried out in the constructors. Specific methods called by the Data Management Layer upon starting or shutting down a pipeline are provided, which shall be overridden.
* In case that may fail, please catch all potential exceptions and turn them into a eu.qualimaster.pipeline.DefaultModeException (as of rev 2620, please read the JavaDoc). The generated pipeline (as of rev XXXX) catches the exception and switches processing there into default mode, i.e., for each input tuple, an output tuple with Java default values will be emitted.
* Sources and sinks shall not be initialized directly rather than via the Data Management Layer, i.e., their constructors shall not be used directly. On the one side, this allows the Data Management Layer to add additional functions to the source and sink instance via wrapping (e.g., transparent input caching). On the other side, the sources and sinks are then registered with the Data Management Layer, which enables automatic connect / disconnect of sources and sinks at the right point in time during pipeline startup / shutdown. For example, the respective code for a data source implemented in the class Src may look like DataManager.DATA\_SOURCE\_MANAGER.createDataSource(“pipeline”, Src.class, NoStorageStrategyDescriptor.INSTANCE);
* As we currently do not have a getName method in IDataElement (Data Management Layer), sources and sinks shall implement a proper toString() method so that individual data sources can be identified in the logs during automatic connect / disconnect of sources / sinks.
* Manifests allow simplified configuration of Sources/Sinks in QM-IConf with just some simple clicks. Therefore, QM-IConf reads the manifest from META-INF/manifest.xml and performs a class file analysis. For Maven builds, just put a respective file into src/main/resources/META-INF (or your configured resource folder) and it will become part of your build.
* Akin to algorithms, Sources/Sinks shall also free resources while stopping the pipeline. Basically, the disconnect method of Sources/Sinks will be called and resources can be released through this method. The disconnect method might be called twice as it is also included in the storm close method.
  + Sources may provide access to historical data in order to increase the precision of the source volume prediction. The access provider is a forward object, which shall be self-contained, serializable to be called independently of the source. Moreover, data sources shall provide access to the aggregation key for data items of the same kind can be obtained from a data item. This key is important for source volume prediction - if not given, no prediction will happen. Access to the aggregation key shall be reentrant and as far as possible independent of the usual operations of the data source.
* Regarding state transfer, see Bolts/Spouts and algorithms.
* Please do not delay the response of a source / sink by sleeping / waiting for the same reasons as this increases the response time / latency and may accidentally affect runtime measurements and cause unintended runtime adaptations. As for algorithms, please consider fractinated sleeping / waiting.

**Infrastructure components**

* One of the components required by the source volume prediction comes with a rather outdated XML parser dependency (gnujaxp), for which it is currently unclear whether we can grid of it again. gnujaxp registers itself as default XML parser disabling the Java default parser (DocumentBuilderFactory). During the automated integration tests, the top-level tests using some XML functionality failed while others passed (it seems reading the contents of an XML element is not properly implemented). If you are reading XML code on infrastructure (not algorithm) level, this may affect your functionality, e.g., in the external service. As a workaround, there is now the class XMLFactory in the infrastructure, which tries to return the default java factory if possible. Using this class, the infrastructure functionality is back again.
* Infrastructure components that shall become accessible to the adaptation script must use specific annotations to enable type safety for their signatures. Basically, all events and commands of the infrastructure are considered by default to become visible to rt-VIL. If this is not intended, use @QMInternal on class (or method) level to avoid the inclusion. If further classes are needed as the signatures to be included depend on them, use @QMSupport on class level. The annotation @QMGenerics shall be used to indicate the actual type generics of a data type as (at least until Java 1.7) this information is removed from bytecode. @QMName can be used to transparently rename a class or a method, i.e., to adjust Java to rt-VIL conventions. Moreover, @QMNoSimulation shall be used for methods that shall not be executed during dry simulation runs of rt-VIL, e.g., in QM-IConf or EASy-Producer. @QMNoSimulation shall in particular be applied to the execute methods (renamed by @QMName to exec to avoid keyword conflicts with rt-VIL) of coordination commands, so that commands can be created but are not executed without a running infrastructure.