API

# **Dispatcher**

#### ExecutionProxy

*/\*\*  
 \* ExecutionProxy is the link between the management layer (i.e. the dispatcher)  
 \* and the execution layer. It passes requests (HTTP) from the dispatcher to the machines  
 \* running the different units.  
 \* Called by the MachinesManager in order to execute a task on the machine itself  
 \* and calls the MachineController for this purpose.  
 \*/*public interface ExecutionProxy {  
  
 */\*\*  
 \* Sends an execution request of a unit on a machine (all this information is part of a task).  
 \** ***@param*** *task holds within it all the information needed to run a certain unit on a machine  
 \* and return the response asynchronously after execution is completed.  
 \*/* void execute(Task task);  
  
 */\*\*  
 \* Sends a request to a certain machine to get its latest performance metrics.  
 \** ***@param*** *machine is the Machine whose performance statistics are requested.  
 \** ***@return*** *MachineStatistics of machine.  
 \*/* MachineStatistics getStatistics(Machine machine);  
  
}

#### FlowVerificator

*/\*\*  
 \* FlowVerificator receives a desired flow (List of Units) from the user,  
 \* verifies that all of the restrictions and prerequisites are met in this flow  
 \* (i.e. the order of the units in the list is valid) and if so, passes the  
 \* analysis flow on to the underlying layers for execution.  
 \*/*public interface FlowVerificator {  
  
 */\*\*  
 \* Receives a request for an analysis flow, verifies its validity and  
 \* passes it on to the QueueManager. If the flow is not valid, a new flow is suggested  
 \* (may be viewed as a negotiation between the user and the system).  
 \** ***@param*** *units is the ordered list of units to run in this analysis as requested by the user.  
 \** ***@return*** *a List of Units; if the flow units is valid, the returned value would be the same  
 \* List (by reference). Otherwise, the returned value is a suggested reordering of the flow.  
 \** ***@throws*** *DispatcherException if no valid order can be computed for the given list of units.  
 \*/* List<Unit> startAnalysis(List<Unit> units) throws DispatcherException;  
  
}

#### MachinesManager

*/\*\*  
 \* Manages machines' performance, availability, units persistence  
 \* and all other machine management tasks.  
 \*/*public interface MachinesManager {  
  
 */\*\*  
 \** ***@return*** *available Machine to run analysis  
 \*/* Machine getAvailableMachine();  
  
 */\*\*  
 \* returns a List of all the Machines where the MachineStatistics are up-to-date.  
 \* This can be used to reflect machines' status to the user, etc.  
 \** ***@return*** *List of all Machines registered in the system  
 \*/* List<Machine> getAllMachines();  
  
}

#### QueueManager

*/\*\*  
 \* Manages the queue of tasks to run. This module receives units to run,  
 \* couples them with machines and creates tasks to be executed.  
 \* When a unit is ready to be executed, it is assigned a machine and  
 \* the QueueManager sends this task to the ExecutionProxy for execution.  
 \*/*public interface QueueManager {  
  
 */\*\*  
 \* Receives a verified execution flow consisted of units to run and enqueues them  
 \* for future execution. Some may be dequeued and run immediately after being added  
 \* to the queue, while others may be persisted in the DB and await their turn  
 \* (when all prerequisites are met and there is an available machine to run on.  
 \** ***@param*** *units to run  
 \*/* void enqueue(List<Unit> units);  
  
 */\*\*  
 \* Called by the ExecutionProxy when a task has been completed (successfully or unsuccessfully).  
 \* The QueueManager may try to send another execution request if the execution  
 \* was not completed successfully. In any case, this method updates the DB with the  
 \* new task details.  
 \** ***@param*** *task to update status in the DB after it was executed.  
 \*/* void updateTaskStatus(Task task); //*TODO add execution response object (with return-code and message)*}

# **Machine**

*/\*\*  
 \* MachineController is run on each machine and manages the units running on it.  
 \* Every request to the machine passes through the MachineController and it  
 \* communicates with the ExecutionProxy via HTTP  
 \*/*public interface MachineController {  
  
 */\*\*  
 \* The machine may gather performance metrics throughout its runtime or per request.  
 \* A call to getStatistics packages the metrics into a MachineStatistics object.  
 \** ***@return*** *MachineStatistics with the performance metrics of the underlying machine  
 \*/* MachineStatistics getStatistics();  
  
 */\*\*  
 \* A request to run a certain unit on the machine. If the request is valid,  
 \* this will trigger the execution of a unit on the underlying machine  
 \* A response regarding the completion of the unit's operation will be returned async.  
 \** ***@param*** *unit is the Unit for which the running request was sent.  
 \* It includes all the settings the specific unit may require to run.  
 \*/* void run(Unit unit);  
  
}