movizon CONTROL **Script API**

movizon CONTROL (mC) without script is only a framework. The software manages access and persistence of resource data and provides a sandboxed environment for scripts to implement logic and logistic. Scripts can utilize the latest JavaScript language standards and benefit from a high-performance engine in the background.

Data Exchange

Each script task has its own context that lives only as long as the script is running. The running state is different to the activation state. An activated script of type INTERVAl set to a one second interval will be triggered once each second but runs according to its load hopefully only some milliseconds. So the lifetime of any JavaScript object created during runtime is also only some milliseconds. Scripts can not memorize anything (this solves some data integrity problems of the old nashorn script engine). Any data exchange must be done via REST request in string format as this is the common understanding of resource data for scripts, JAVA core, browser UI and database. That requires constant serialization and deserialization. Overall, the scripter's design choices affect performance to a much greater extent than in previous versions of mC.

The **recommended approach** is to keep string size and serialization/deserialization minimal. Avoid too long strings both in the keys and in the values of a resource, round numbers (2.8786511298432165 to 2.88), store values only if necessary and in just one place inside a resource object. Retrieve and parse resources once in your code. Always filter your GET requests to reduce parsing overhead (see REST API documentation for details). Intentionally there are no negated filters (like "get all fields except abc") because you never know what unwanted giant objects you might end up catching. Write single fields instead of the whole resource whenever possible. Also organize your resources into logical resource types in terms of size and task access (e.g. dedicated route resources in contrast to job resources with route field).

Resource Locking

Each script has one or multiple tasks that execute the code. A task for its own is always single-threaded, but all tasks together form a system where multiple threads attempt to access the same resources at constantly shifting times. The JAVA core synchronizes modifying REST requests within the same resource type, but that doesn't solve the following problem:

Task A retrieves a state of a resource via GET, then after changing some fields writes is back to the core via PUT. Changes that are made to a single field of the same resource from task B after the GET request from A will be overwritten when executing the PUT request from task A. This can be solved by locking: Task A locks the resource before the GET request. All modification requests from other tasks/threads are queued now, including the change request from task B. Task A performs the PUT request and unlocks the resource. Now the queued change request from task B can be executed. But if task B would also want to change the entire resource, its PUT request would overwrite the carefully locked change from task A completely. Task B could take the changes of A into account also by locking, but this would slow things down unnecessarily.

The **recommended approach** is to have only one manager task per resource type, the only one that writes entire resources. This manager does the locking. All other tasks that interact with the same resource type only write single fields without locking.

Field Reservation

In addition to locking, there is a second technique to ensure resource integrity: With field reservation a script can specify that only itself can change certain fields of a resource. Imagine a vehicle manager script that cyclically retrieves raw data from an interface and calculates the position of all vehicles. The manager script should take authority over the position property by reserving all related fields (e.g. `posX`, `posY`, `posH`). Otherwise, a moving vehicle could unintentionally jump back while some user is editing that vehicle in the UI, as the position reverts to the snapshot taken at the moment the editor was opened.

The **recommended approach** is to organize field reservations and thus scripts in general according to their purpose: The manager script of each resource type reserves the main part of the fields that are not meant for user input. Other scripts that also need to write to the same resource type should be single purpose scripts that reserve only "their field", e.g. a traffic management script that writes a special speed limit value to all vehicles.

Host Functions

The task that is running a script's code provides a basic setup of methods for JAVA interoperability. These methods allow communication with the JAVA core, which manages users, scripts and the resource access of both via REST. Two objects are bound to each script context and therefore directly available in the JavaScript code: `host` and `args`. `host` is the hosting task and provides the functions described in the table below. `args` is a string array , filled only if the script was triggered (= the task was created) via run request (`host.run` or POST to scripts).

Due to the neutrality of the resource system, there are no type related functions any more (like `h.isAtEndPoint(target)` from v3's Script API). Instead those helper functions should be defined in a dedicated script that can be included by any script that uses the same logistical resource structure.

`void reserve(String path, String... keys)`	
Description	reserves fields of all resources of the specified resource type (path); reserved fields can only be set by the reserving script (all tasks of a script) itself; reservation is valid as long as a script is activated (not only if running); valid reservations can't be overridden by other scripts
Example	`host.reserve("vehicles", "posX", "posY", "posH", "lastWaypointId");`
`void lock(String path)`	
Description	locks the specified resource (path scheme [RESOURCE_TYPE]) or resources (path scheme [RESOURCE_TYPE]/[RESOURCE_ID]); modifying REST requests from other users/scripts will wait until the locked resource is unlocked again; to prevent deadlocks no resources can be modified except for the locked ones; also nested locking is not allowed

Example	host.lock("vehicles"); // has to be before reading
	
	let vehicles = JSON.parse(host.request("GET", "vehicles", ""));
	if (Array.isArray(vehicles))
	{
	  vehicles.forEach(v => manage(v));
	  vehicles.forEach(v => save(v));
	}
	host.unlock("vehicles");

void unlock(String path)	
Description	unlocks the specified resource or resources; will be called automatically on any lock after script execution if not stated
Example	see above
void include(String id)	
Description	includes the code of another script; included functions will override functions of host script regardless of the position of the include statement; repeated inclusion of same script will be silently prevented

Example	host.include("Point");
	let p1 = new Point(5, 5);
	let p2 = new Point(7, 7);
	let distance = p1.distance(p2);

Future<String> run(String scriptId, String taskId, String... args)	
Description	starts a new task of the specified script; throws an exception if a task with the same ID is still running; returned Future object can only be used in scripts of type INTEROP

Example	let vehicleIds = JSON.parse(host.request("GET", "vehicles?keys=id", ""));
	vehicleIds.forEach(id =>
	{
	  try
	  {
	  // parallel execution of driver task for each vehicle
	  // vehicle ID is used for both task ID and argument
	  host.run("vehicleDriver", id, id);
	  }
	  catch (ex)
	  {
	  host.log(ex);
	  }
	});

String getId()	
Description	returns the ID of the host task; format of the ID string is as follows: [SCRIPT_ID] or [SCRIPT_ID]-[TASK_ID] if script has multiple tasks
Example	let taskId = host.getId();
String getType()	

Description	returns the type of the host script; type can be STANDARD or INTEROP
Example	let scriptType = host.getType();
String getResponsibility()	
Description	returns the ID of the user who is responsible for the host script; usually this is the user who last modified any field of the script resource
Example	let userId = host.getResponsibility();
boolean isActive()	
Description	returns if the activation property of host script is set to true; useful to stop scripts with SINGLE trigger that runs in an internal while loop

Example	while (host.isActive())
	{
	  doStuff();
	}

void deactivate()	
Description	deactivates the host script
Example	someSingleExecution(); host.deactivate();
String request(String method, String path, String body)	
Description	processes a REST request; body argument must be string (valid JSON for stringified objects); returned string must be parsed if object or array expected; method will not return HTML status codes; successful requests will return requested object as string (GET), ID of created resource (POST) or null (PUT, DELETE); unsuccessful requests will throw an exception with corresponding error code inside its message

Example	let vehicle = JSON.parse(host.request("GET", "vehicles/TCVFORK1", ""));
	doStuff(vehicle);
	host.request("PUT", "vehicles/TCVFORK1", JSON.stringify(vehicle));

void log(Object message, String... flags)	
Description	logs a message; without flags the message will be appended to the host script itself (displayed in scripts resource table); with flags argument the message is written to the file logger; value of flags argument is currently unused
Example	host.log("This will be displayed in detail box of resource table"); host.log("This will be printed to logfiles", "INFO");
void clearLog()	
Description	clears the log of the host script; to clear also errors and everything else set the clearRequest field of the host script to true
Example	host.clearLog();