Spreadsheet Import Tool

Automation Professionals' Exchange Resource

Version 1.1

July 16, 2024 15:17z

The Spreadsheet Import Tool allows a complex structure of Ignition OPC Server Device instances, OPC UA Server Connections, UDT instances, and arbitrary tag instances to be defined and maintained in an Excel (or LibreOffice) spread sheet format.

Table of Contents

Overview

When an Ignition gateway needs to be deployed to multiple sites, or to multiple independent systems within a site, but with a varying numbers and names of devices, with a corresponding varying tag architecture, scripting is the native solution. Ignition offers `system.device.addDevice()`, `system.tag.configure()`, and `system.tag.writeBlocking()` to allow automation of such tasks.

However, this can be non-trivial, and there is no common data format defined for these tasks. Everything depends on in-house programming talent. This Exchange resource aims to consolidate all such tasks into a familiar format, with a **single** definitive script.

This common script can perform the following tasks:

- Create arbitrary devices in Ignition's built-in OPC UA server, using any properties supported by Ignition's `system.device.addDevice()` API. Existing devices are skipped by default, but deletion before recreation may be selected.
- Create OPC UA server connections with Ignition's built-in OPC UA client, using any properties supported by Ignition's `system.opcua.addConnection()` API. Existing server connections are skipped by default, but deletion before recreation may be selected.
- Create UDT instances at arbitrary tag paths using any UDT definition present in a selected tag provider. Setting parameters, properties (including custom properties), and member tag values is fully supported. Parameter and member tag value operations include data type coercion based on the UDT definition. Properties are delivered with the `system.tag.configure()` call, while parameter overrides and member assignments use `system.tag.writeBlocking()` (in order to avoid data type munging bugs that plague the `.configure()` operation).
- Create arbitrary tags (typically atomic tags) with desired properties' assignments.
- UDT instances and arbitrary tags use "merge" mode by default, but "overwrite" mode may be selected.
- There is also a mode for **deleting** Devices, OPC Connections, UDT instances, and arbitrary tags created by this script (feeding the original spreadsheet back through). For developer convenience when static tags exist that should not be destroyed when recovering from an import operation that goes awry.

All of the above may leverage common features of the spreadsheet traversal algorithm:

- Numeric iteration is available, for repeating any of the above operations for many devices, UDT instances, and tags. Iteration over arbitrary strings is possible using a dummy numeric iterator and adjacent value columns. Iteration can also be nested within a group of related operations, for multiple UDT instance per device, and similar complex hierarchies.
- Most values in the spreadsheet may use `printf`-style string substitution to inject numbers and strings from prior columns in the spreadsheet, within a block of related rows. This is the key to including iterator integers and associated string lists in properties, parameters, and value overrides.
- Python's `eval()` function is supported in a special column type for operations that cannot be performed purely by `printf`-style substitution.
- Device properties, UDT parameters, properties, and member value overrides, and arbitrary tag properties are all specified in a compact two-column key/value format. The value column in each pair supports `printf`-style string substitution, too.

- Device, OPC UA Connection, UDT instance, and arbitrary tag properties are delivered to the Ignition API without any particular data type coercion. Ensure your spreadsheet values are in a form that naturally converts.
- UDT parameter values, UDT member values, and arbitrary tags' assignment to a value property will all be coerced to the defined data types. Document tag types should use JSON encoding.

Prerequisites

This module requires the installation of Automation Professionals' Integration Toolkit third party add-on module for Ignition, version 2.0.17 or later. Install this module in your gateway **before** importing the project file.

The project is tested on Ignition version 8.1.42, but may run on earlier versions.

Installation

These tasks are required:

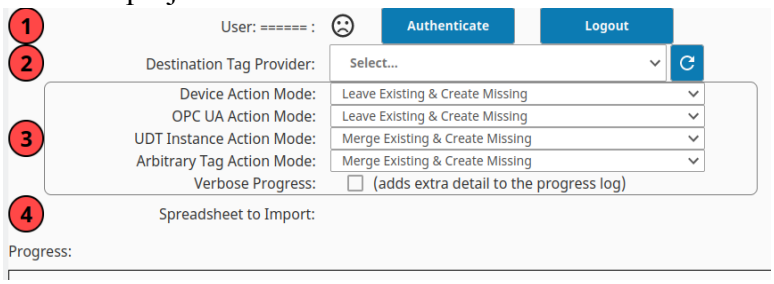
- Install Automation Professionals' Integration Toolkit third party add-on module **first**.
- Import the supplied project file to a project name of your choice. The Automation Professionals' test and development environment uses "spreadsheet-import-tool". This project deliberately does **not** have a default tag provider, nor a default database.
- Update gateway and/or project security settings so that the user intended for this project has the "Administrator" role (case sensistive). The project uses the "default" IdP unless updated.

This task is optional, but recommended for running the example spreadsheet:

- Import the UDT definitions supplied in `ExampleUdts.json`, perhaps into a new Realtime Tag Provider.

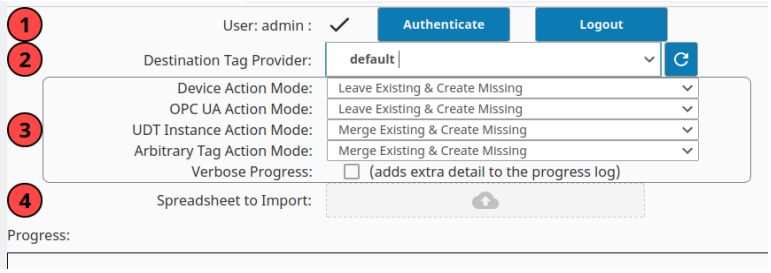
Usage

- Create any new Realtime Tag provider needed, and import any UDT definitions you will need for the spreadsheet(s) you intend to use.
- Launch a session of this project. You should see this:

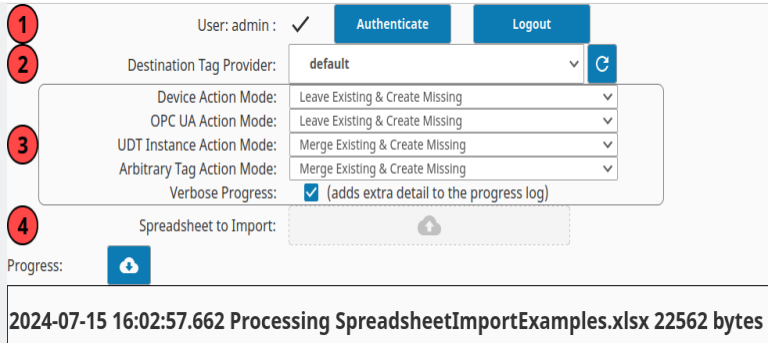


- Click the "Authenticate" button. The sad face will change when you have the "Administrator" role.
- Follow the remaining, numbered, steps. Use the provider list refresh button if a new realtime tag provider was added after gateway startup.

- The file upload component won't appear until a tag provider is selected.



- Set the "Verbose Progress" checkbox to get considerable extra detail in the progress log. Helpful when troubleshooting a workbook that doesn't yield the expected results.
- The import operation occurs ***immediately*** upon upload—there is no confirmation.
- While the import runs, the UI components are disabled, and the progress area updates every ¼ second. At the end, the UI is re-enabled, and the progress log is available for download (in markdown format).



Spreadsheet Architecture

The import algorithm expects the supplied workbook to have one or more worksheets that follow a specific, hierarchical layout, with a row of keyword headers, and one or more blocks of rows with values that drive the import actions. Worksheets within the workbook that do not have the expected structure are simply skipped, allowing them to be used for documentation, or as lookup sources for spreadsheet-side automation.

Spreadsheet formulas and styles are entirely ignored. Only the values in the spreadsheet matter for the import, as they were captured when the workbook was saved. Even hidden content will be processed upon import.

Header Keywords

Column headers are taken from the first row that has **any** content in **column A**, there is at least one column that contains an "Action Trigger" keyword, and other columns contain the required keywords that correspond to that action (if any). Other columns in the header row may be empty, causing that entire column to be ignored by the import algorithm. Which allows such columns to contain documentation, or function as lookup sources for spreadsheet-side automation. (Make sure headers for lookup operations do not fall on the same row as the algorithm's headers.)

Keyword	*Action Trigger*	*Required For Action*	*Operation*
DeviceType	✅		A string for the deviceType argument to `system.device.addDevice()`.

Keyword	*Action Trigger*	*Required For Action*	*Operation*
Device		DeviceType	A string for the deviceName argument to `system.device.addDevice()`.
DevPropKey DevpropValue		DeviceType	Must be adjacent columns. Supplies key/value pairs needed in the deviceProps argument to `system.device.addDevice()`, as documented for the specific device type. If a description key is supplied, it will be popped from the settings and delivered as a keyword argument to `system.device.addDevice()`.
OpcCx	✅		A string for the name argument to `system.opcua.addConnection()`.
OpcPropKey OpcPropValue		OpcCx	Must be adjacent columns. Supplies key/value pairs needed in the settings argument to `system.opcua.addConnection()`. If any of the following keys are supplied, they will be popped from the settings dictionary and delivered directly as keyword arguments to `system.opcua.addConnection()`: • description • discoveryUrl • endpointUrl • securityPolicy • securityMode
UdtTypePath	✅		Must be the path of a UDT definition within the provider. Excluding the provider and the "_types_" folder names.
UdtInstancePath		UdtTypePath	Instance name, with folder path, to create. Exclude the provider name. Also treated as a variable name.
UdtParamKey UdtParamValue			Optional for UDT creation. Supplies parameter overrides. Keys that do not match parameter names in the UDT may produce errors.
UdtPropKey UdtPropValue			Optional for UDT creation. Supplies property overrides and/or new property values. Keys that are not associated with common properties will be assigned to new custom properties.
UdtMemberKey UdtMemberValue			Optional for UDT creation. Supplies value overrides and/or property overrides for member tags within the UDT. Member names that do not exist in the UDT will produce errors. Property names appended to member names can yield custom properties.
TagInstancePath	✅		Instance name, with folder path, to create. Exclude the provider name.
TagPropKey TagPropValue			Optional for Tag creation. Supplies property overrides and/or new property values. Keys that are not associated with common properties will be assigned to new custom properties.
*prefix*Iterator			Any header that ends with "Iterator" supplies one or more integers or integer ranges to repeat executions of following columns, via "prefix" as a variable name.
*prefix*Eval			Any header that ends with "Eval" causes the value(s) in the column to be treated as python expressions. Result is placed in the "prefix" variable name.
*prefix*Key *prefix*Value			Must be adjacent columns. Supplies key/value pairs to the local variable table as a convenience. The prefix is discarded.

All column headers not ending in "Key", "Value", "Iterator", or "Eval", including any user-supplied headers, are treated as variable names. All column header keywords are **case-sensitive**.

Row Blocks

Starting immediately below the row of header keywords, the import algorithm uses the **content of column A** to segregate the rows of the worksheet into blocks of related rows. The first row under the headers with any content in column A starts the first block. The first block's rows continue until an empty cell in column A is followed by an occupied cell in column A.

The second block, and all following blocks, start at the transition from an empty cell in column A to an occupied cell in column A. The import operation handles one block at a time. While only the cell in column A above a new row block needs to be empty, a completely empty row is recommended.

Local Variables

Every cell from which a string is read, except for columns whose keywords end in "Key" or "Eval", is run through python's dictionary-based, printf-style **"%"** string formatting operation before any further computation. The result is placed into the local variable dictionary for use in cells below it and/or to its

right. When the cell is in the "Value" column of a key/value column pair, and is one of the special columns used by an action, the result value is also placed in that action's special dictionary(ies). When the cell is in a column under an "Eval" keyword, python's `eval()` function is used instead of python's string formatting operator.

Column Blocks

Within a related block of rows, the import operation evaluates column-wise from left to right, accumulating all of the information that will apply to any triggered actions. Columns that are paired keys & values are (always) evaluated row-wise. Columns that have no header are skipped. Other columns that are not to the right of an iterator take their value from the **first** row of the block, ignoring any others. Other columns that **are** to the right of an iterator take their value from the same row as the iterator, unless it is an empty cell, and then the first row of the block is used.

A column block ends at an iterator column, or the right edge of the spreadsheet. At the end of a column block, all actions that have been triggered since the start of the column block are executed. (Only one action of each type may be present in a column block. Extras won't generate an error, but only the latter action will be executed.)

Iterators

Columns whose keyword ends in "Iterator" cause any column block to their left to execute (clearing any queued actions), then capture the state of the local variable dictionary, and begin a new column block. The iterator column evaluates row-wise, and must contain plain integers, a hyphen-delimited integer range, or comma-delimited collections of plain integers and/or ranges. Whitespace is ignored. Empty cells are skipped. The **entire** column block to the right is repeatedly evaluated for every integer given. For every repeat, the local variable dictionary is reset to the captured state, and queued actions executed.

Note that integer values and ranges can appear multiple times, and will then execute with that value multiple times. This would typically be paired with a companion general-purpose column that supplies unique values for each multiple.

Multiple iterator columns can be used, and will execute as nested loops. That is, when a first iterator has ten integer values, and the second iterator has five values, the column block starting with the second iterator column will execute **fifty** times.

Actions

The whole point of this import tool is to execute one or more actions. The information needed by an action, as described in the table of keywords, is assembled as a column block is evaluated, and executed at the end of that column block. The "action trigger" columns are what place that action in the queue of things to do at the end of the column block. The special key/value dictionaries associated with an action are attached to the queue, and are discarded after execution. This allows later column blocks to use the same actions (same column headers) for a nested purpose, using only the newly-specified settings. (The prior values remain in the local variable table if not overwritten, but do not affect later actions.)

Each action has an overall import mode, independent for each action, set by the user in the Perspective User Interface. The possible modes can be summarized as:

- Conditional create (either skip existing or merge to existing),
- Unconditional create (either delete before creating, or overwrite while creating),
- Delete (reversing the effect of a prior run of the workbook), and
- Skip (ignoring all actions of that type).

Available Actions:

- Device create or delete. Calls Ignition's `system.device.addDevice()` and/or `system.device.removeDevice()` with the arguments and additional properties computed for import. No "merge" is possible with Ignition's current API.
- OPC UA Connection creation. Calls Ignition's `system.opcua.addConnection()` and/or `system.opcua.removeConnection()` with the arguments and additional properties computed for import. No "merge" is possible with Ignition's current API.
- UDT Instance create/merge/delete. Calls Ignition's `system.tag.configure()` or `system.tag.deleteTags()` with the arguments and properties computed for import. When not deleting, also calls `system.tag.writeBlocking()` for all parameter and value assignments.
- Arbitrary Tag Instance create/merge/delete. Calls Ignition's `system.tag.configure()` or `system.tag.deleteTags()` with the arguments and properties computed for import. When not deleting, and the tag's value property was supplied, also calls `system.tag.writeBlocking()` for that value assignment. When the dataType property is omitted, this action creates a Document tag, not an Integer tag. Can be used with proper properties to automate creation of non-atomic tags outside UDTs, including tag folders with custom properties.

Special handling is provided when UDT or tag instance creation involves setting values on Document tags or UDT members of Document type: the string provided for that value in the spreadsheet is expected to be dictionary in JSON-encoded format. The import operation will then perform the necessary call to `system.util.jsonDecode()` before the call to `system.tag.writeBlocking()`.

When using the "Delete" mode with UDT instances and arbitrary tags, only the leaf name specified in the spreadsheet will be provided to the `system.tag.deleteTags()` API call. Be aware that `system.tag.configure()` will *implicitly* create any folders needed when creating/merging any instance path it is given. The delete operation will not touch those.

Support

Contact Automation Professionals via our Support Email address. But note that exchange support is low priority.