KhaledMostafaMe

Advanced JSON Toolkit

Readme

Version 1.2.2

20/09/2026

Table of Contents

1. Introduction

This document contains everything needed to install and use the Advanced JSON Toolkit package: what it does, what it requires, how to set it up, how each group of actions is used, and its known limits.

1.1 Overview

Advanced JSON Toolkit is a single Automation 360 package that gives a bot 47 actions for working with JSON: reading it, querying it, converting it to and from the data types a bot actually uses, transforming it, validating it, and redacting it.

In plain terms: whenever a bot receives JSON from an API, a file or a queue, this package turns it into a Dictionary, List or Table a bot can loop over - and turns bot data back into JSON to send onward - without losing anything on the way.

JSON handling looks simple until production data arrives. A 19-digit record identifier has to survive a round trip unchanged. A rate of 0.0000001 has to stay in plain notation rather than becoming 1E-7. A query that lands on an object or an array has to return an object or an array, not text. Key order has to hold when a downstream system signs or compares the payload. Arabic and other non-Latin content has to come back as it went in. Those requirements are the design brief for this package, and every action is built to meet them.

The package bundles the libraries it needs - Jackson 2.18.3, Jayway JSONPath 2.9.0 and json-smart 2.5.2 - exactly as Automation Anywhere's own JSON package does, because the Bot Agent does not put them on a custom package's classpath. Everything runs inside the package's own classloader, nothing is installed on the agent, and it makes no network call of any kind.

It uses its own unique package name, so it installs alongside whatever you use today and can be adopted one action at a time.

1.1.1 Sample input and output

Input: a JSON string from an API response, held in a bot variable, for example {"invoiceId": 9007199254740993, "total": 0.0000001, "lines": [{"sku": "A-1", "qty": 2}, {"sku": "B-7", "qty": 5}], "customer": {"name": "Contoso", "vat": "AE1234567890"}}

Action: JSONPath Query (single) with the path $.customer.name. Output: a String, Contoso.

Action: JSON to Table with the path $.lines. Output: a Table of two rows and two columns - sku and qty - ready for a Loop.

Action: Pretty Print / Minify. Output: invoiceId is still 9007199254740993 and total is still 0.0000001 - values round-trip exactly, in plain notation, whatever their size or precision.

Action: Redact / Mask JSON on $.customer.vat with the keep-last-N option. Output: {"customer": {"vat": "********7890"}, ...} - safe to log.

1.2 Use cases

The key use cases include:

- REST API integration - parse any API response into a Dictionary, List or Table, query the fields the bot needs with JSONPath, and build the request body for the next call.
- Financial and identifier-safe processing - handle invoice totals, account numbers and 64-bit record identifiers without the silent rounding that makes a reconciliation fail hours later.
- File format conversion - move data between JSON, CSV, XML, YAML, NDJSON and query strings in one action, in either direction, as part of a file-handling bot.
- Data preparation before a bot loop - filter an array, sort it, group and aggregate it, deduplicate it, pick or omit fields, and rename keys, so the bot loops over exactly the rows it needs.
- Configuration and payload management - read settings from a JSON file, apply an RFC 7386 merge patch or an RFC 6902 patch, and write the result back.
- API change detection - diff two JSON documents and get an RFC 6902 patch describing exactly what changed between them.
- Logging and privacy - mask, hash or partially reveal sensitive fields before a payload reaches a log file or a support ticket.
- Validation and error handling - check a payload is well formed and matches an expected structure before the bot acts on it, and get the line and column when it does not.
- Token inspection - decode a JWT's header and claims to read an expiry or a subject, without verifying the signature.

2. Requirements & Prerequisites

2.1 System Requirements

• Automation 360 Control Room able to accept custom packages (Java bytecode 55 / JDK 11). Cloud or on-premises.

• Bot Agent on Windows. The package is pure Java and also declares macOS as a target, but it has been validated on Windows only.

• About 3 MB on disk. The jar carries this package's own classes plus the libraries it bundles (Jackson, Jayway JSONPath, json-smart, ASM), its manifest, locales, icon and licence files.

• Control Room build 8750 or above and Bot Agent 20.11 or above - the minimum each action declares. Validated on Bot Agent 23.0.32.

2.2 Prerequisites

• No software to install on the Bot Agent. Everything the package needs - Jackson, Jayway JSONPath, json-smart and ASM - is bundled inside the jar and loaded in the package's own classloader.

• No licence key, no subscription and no account with any external service.

• No Credential Vault locker and no credentials of any kind.

• No network access. The package never makes an outbound call; it operates only on the strings and files you give it.

• Where an action writes to a file, the Bot Agent needs write permission on that path. Parent folders are created if missing and an existing file is overwritten.

• Standard bot-execution rights. Administrator rights are not required.

3. Getting Started

3.1 Quick Start

3.1.1 Setup

1. Download AdvancedJsonToolkit-1.2.2.jar from the Agentic App Store listing.

2. In the Control Room, go to Manage, then Packages, then Add package, then Browse.

3. Select the jar and upload it, then choose Accept and enable.

4. Open a Task Bot. The actions now appear in the Actions palette, grouped by purpose - Parse and Validate, Query, Convert, Transform, Schema, Utility, Arrays and Datasets, and Privacy.

5. There is no external account, portal or API key to configure - setup is complete at this point.

6. Optional: because this package uses its own unique name, it coexists with the existing JSON packages. Migrate bots one action at a time rather than all at once.

3.1.2 Configuration and Use

There is nothing to configure at package level: no settings file, no connection and no session. Every option is an input on the action itself. Each action reads from either a JSON string or a file, and each action that produces JSON can write to either a string or a file, so the same action works whether the payload is in a variable or on disk.

3.1.3 Files included in this submission

The submission bundles the following. Nothing else is required, and nothing is downloaded at run time.

• AdvancedJsonToolkit-1.2.2.jar - the custom package. Adds the 47 actions described in this document to the bot editor. It bundles the libraries it needs - Jackson 2.18.3, Jayway JSONPath 2.9.0, json-smart 2.5.2 (Apache-2.0) and ASM 9.7.1 (BSD-3-Clause) - so it runs without installing anything on the Bot Agent.

• AdvancedJsonToolkitDemo_windows, in the folder Advanced JSON Toolkit-KhaledMostafaMe - the sample Task Bot. It exercises all 47 actions on synthetic data, needs no credentials, and runs unattended. It is built on Automation Anywhere's Bot Store template, so it also carries the template's error handling, log folders and snapshot-on-error logic.

• Automation Anywhere's built-in packages the sample bot depends on (Comment, Step, String, Log To File, Error handler, If, Folder, File, Datetime, Number, System, Screen, Message box and Task bot). These are bundled automatically by the submission.

Files the sample bot creates while it runs, in its work folder. That folder needs no setting up: the bot assigns pWorkFolder at run time to the per-bot folder Automation Anywhere's own Bot Store template already creates under the Bot Agent log root:

• sample_object.json - written by the package in the Setup step; read back by Step 3.

• out.csv, out.xml and out.ndjson - written by Steps 17, 18 and 20; out.csv and out.ndjson are read back by Steps 36, 37 and 39.

• ajt-demo-run.txt - one line per step group plus a start and a finish line.

This Read Me is uploaded separately as a PDF on the submission form. It is not part of the bot and is not a dependency of it.

3.1.4 Action reference

All 47 actions, grouped as they appear in the palette, with what each one does, what it returns and what you give it. Every action reads its JSON from a string or a file.

1. Parse / Validate

• Validate JSON - Validate JSON and return {valid,error,line,column,rootType}. A parse failure is never disguised as success. Returns: Dictionary. Inputs: JSON source (text or file); Throw an error if invalid.

• Detect JSON Value Type - Report the type of the root JSON value: object | array | string | number | boolean | null. Returns: String. Inputs: JSON source (text or file).

• Get JSON Parse Error Detail - Return {valid,message,line,column,offset} for a JSON document. Never throws. Returns: Dictionary. Inputs: JSON source (text or file).

2. Query

• JSONPath Query (single) - Evaluate a JSONPath and return the single matched value as a typed variable (object/array/number/boolean/text). Returns: Any. Inputs: JSON source (text or file); JSONPath; Result form; If the path is missing.

• JSONPath Query (multiple -> List) - Evaluate a JSONPath as a collection; return every match as a typed List (empty List when none). Returns: List. Inputs: JSON source (text or file); JSONPath.

• JSONPath Query to Table - Resolve a JSONPath to an array of objects and build a Table with a full union header and typed cells. Returns: Table. Inputs: JSON source (text or file); JSONPath to an array of objects; Explicit ordered columns; Null / missing cell as; Numbers as.

• JSON Pointer Get (RFC 6901) - Read a value by RFC 6901 pointer (e.g. /user/name, /a~1b for a literal 'a/b' key). Distinguishes dotted keys from nesting. Returns: Any. Inputs: JSON source (text or file); JSON Pointer; If the pointer is missing.

• Key / Path Exists - Return true/false for whether a path exists. Present-but-null counts as existing. Returns: Boolean. Inputs: JSON source (text or file); Path; Path type.

3. Convert In

• JSON to Dictionary - Convert a JSON object to a nested Dictionary with exact numbers, preserved key order and real UTF-8. Returns: Dictionary. Inputs: JSON source (text or file); Numbers as.

• JSON to List - Convert a JSON array to a typed List (a single object is wrapped as a one-element List). Returns: List. Inputs: JSON source (text or file); Numbers as.

• JSON to Table - Convert an array of objects to a Table (full union header, typed cells, deterministic column order). Returns: Table. Inputs: JSON source (text or file); JSONPath to the array of records; Explicit ordered columns; Null / missing cell as; Numbers as.

• JSON to Record - Convert a single JSON object to a Record with typed values (schema order optional). Returns: Record. Inputs: JSON source (text or file); JSONPath to the object; Ordered column names.

4. Convert Out

• Dictionary to JSON - Serialize a Dictionary to JSON with exact numbers (no scientific notation), stable order, UTF-8. Returns: String. Inputs: Dictionary; Pretty-print; Sort keys; output to variable or file.

• List to JSON - Serialize a List to a JSON array with exact numbers and recursion into nested Lists/Dictionaries. Returns: String. Inputs: List; Pretty-print; output to variable or file.

• Table to JSON - Serialize a Table to JSON (array of objects or array of arrays) with typed cells. Returns: String. Inputs: Table; Shape; Pretty-print; output to variable or file.

• JSON to CSV - Convert an array of objects to RFC 4180 CSV with a full union header, correct quoting and plain numbers. Returns: String. Inputs: JSON source (text or file); JSONPath to the array of records; Delimiter; Quote every field; Include header row; Explicit ordered columns; CSV-injection guard; output to variable or file.

• JSON to XML - Convert JSON to XML (dependency-free; deterministic element order; arrays repeat elements). Returns: String. Inputs: JSON source (text or file); Root element name; Pretty-print; output to variable or file.

• JSON to YAML - Convert JSON to YAML (block style, dependency-free; numbers plain; order preserved). Returns: String. Inputs: JSON source (text or file); output to variable or file.

5. Transform

• Apply JSON Merge Patch (RFC 7386) - Apply an RFC 7386 merge patch (null deletes a key). Inputs are never mutated. Returns: String. Inputs: JSON source (text or file); Merge patch; Pretty-print; output to variable or file.

• Apply JSON Patch (RFC 6902) - Apply an RFC 6902 patch (add/remove/replace/move/copy/test) atomically with RFC 6901 pointers. Returns: String. Inputs: JSON source (text or file); JSON Patch operations; Pretty-print; output to variable or file.

• JSON Diff (produce RFC 6902 Patch) - Diff two JSON documents into a minimal RFC 6902 patch (numbers compared by value: 1 == 1.0). Returns: String. Inputs: JSON source (text or file); Target JSON to compare against; Pretty-print; output to variable or file.

• Deep Merge Two JSON - Recursively merge two JSON documents (configurable array strategy). No null-deletion semantics. Returns: String. Inputs: JSON source (text or file); Overlay JSON; Array strategy; Pretty-print; output to variable or file.

• Flatten JSON (lossless) - Flatten nested JSON to a flat Dictionary of path -> typed leaf. JSON Pointer style is lossless & round-trippable. Returns: Dictionary. Inputs: JSON source (text or file); Path style; Keep arrays as whole values.

• Unflatten JSON - Rebuild nested JSON from a flat Dictionary (inverse of Flatten JSON). JSON Pointer style round-trips exactly. Returns: String. Inputs: Flat Dictionary; Path style; Rebuild arrays from 0..n indices; Pretty-print; output to variable or file.

• Set Value at Path (immutable) - Set a NATIVE typed value at an RFC 6901 pointer on a deep copy (number stays number, boolean stays boolean). Returns: String. Inputs: JSON source (text or file); JSON Pointer; New value as JSON; Create intermediate objects if missing; Pretty-print; output to variable or file.

• Remove Value at Path (immutable) - Remove the value at an RFC 6901 pointer on a deep copy; sibling order preserved. Returns: String. Inputs: JSON source (text or file); JSON Pointer to remove; Throw an error if missing; Pretty-print; output to variable or file.

• Sort JSON Keys - Recursively rebuild objects with sorted keys for a canonical form; numbers preserved exactly. Returns: String. Inputs: JSON source (text or file); Sort nested objects too; Order; Pretty-print; output to variable or file.

6. Schema

• Validate Against JSON Schema (subset) - Validate an instance against a JSON Schema. Structural subset (type/required/properties/enum/min/max/items) — result is labelled mode=subset. Returns: Dictionary. Inputs: JSON source (text or file); JSON Schema; Throw an error if invalid.

7. Utility

• Pretty Print / Minify JSON - Reformat JSON as pretty or minified; numbers stay exact (no scientific notation); optional key sort. Returns: String. Inputs: JSON source (text or file); Mode; Sort keys; output to variable or file.

• Escape / Unescape JSON String - Safely escape text into a JSON string literal, or unescape a JSON string literal back to raw text. Operates on string literals only. Returns: String. Inputs: Text; Mode.

• JSON Deep Equals - Compare two JSON documents structurally; numbers by value (1 == 1.0); key order ignored; array order optional. Returns: Boolean. Inputs: JSON source (text or file); Second JSON to compare; Ignore array order.

• JSON Size / Count - Count immediate children, deep leaves, or UTF-8 byte size of a node addressed by JSONPath. Returns: Number. Inputs: JSON source (text or file); JSONPath to the node; Metric.

8. Convert In (reverse)

• CSV to JSON - Parse CSV (RFC 4180) into a List of objects (header row -> keys). Optional type inference; exact numbers. Returns: List. Inputs: CSV source (text or file); First row is a header; Delimiter; Infer numbers/booleans.

• CSV to Table - Parse CSV (RFC 4180) directly into an A360 Table (header row -> columns), typed cells. Returns: Table. Inputs: CSV source (text or file); Delimiter; Infer numbers/booleans.

• XML to JSON - Convert XML to JSON (attributes -> @name, repeated tags -> arrays, mixed text -> #text). XXE-hardened. Returns: String. Inputs: XML source (text or file); Pretty-print; output to variable or file.

• NDJSON (JSON Lines) to List - Parse newline-delimited JSON (one JSON value per line) into a List. Blank lines skipped. Returns: List. Inputs: NDJSON source (text or file).

4. Convert Out

• JSON to NDJSON (JSON Lines) - Emit a JSON array as newline-delimited JSON (one minified value per line). Returns: String. Inputs: JSON source (text or file); output to variable or file.

8. Convert In (reverse)

• Query String to JSON - Parse a URL query string into a Dictionary (URL-decoded; repeated keys become a List). Returns: Dictionary. Inputs: Query string.

4. Convert Out

• JSON to Query String - Serialize a flat JSON object to a URL query string (arrays repeat the key; values URL-encoded). Returns: String. Inputs: JSON source (text or file).

8. Convert In (reverse)

• Decode JWT (no verify) - Decode a JWT's header and payload (base64url) into a Dictionary {header,payload,signature}. Does NOT verify the signature. Returns: Dictionary. Inputs: JWT.

9. Arrays / Datasets

• Filter Array - Keep array elements whose field satisfies a condition. Numeric compare when both sides are numbers. Returns: List. Inputs: JSON source (text or file); Field; Operator; Comparison value.

• Sort Array - Sort an array of objects by one or more fields. Typed compare (numbers by value); nulls last. Returns: List. Inputs: JSON source (text or file); Sort keys, e.g. age:desc,name:asc.

• Group By + Aggregate - Group an array of objects by field(s) and aggregate (count/sum/avg/min/max/first/last). BigDecimal-exact. Returns: List. Inputs: JSON source (text or file); Group-by field(s), comma-separated; Aggregations, e.g. count; sum:amount; avg:price.

• Deduplicate Array - Remove duplicate array elements (whole element, or by one or more key fields). Keeps first occurrence. Returns: List. Inputs: JSON source (text or file); Key field(s), comma-separated.

• Select Fields (pick / omit) - Keep (pick) or drop (omit) a set of top-level keys, on an object or each element of an array of objects. Returns: Any. Inputs: JSON source (text or file); Field names, comma-separated; Mode.

• Rename Keys - Rename object keys everywhere in the document via an old=new mapping (deep). Returns: Any. Inputs: JSON source (text or file); Mapping, e.g. old=new,firstName=first.

10. Privacy

• Redact / Mask JSON - Mask, hash, or partially-mask values for the given key names anywhere in the document (PII redaction). Returns: Any. Inputs: JSON source (text or file); Key names to redact, comma-separated; Redaction mode.

3.1.5 Guarantees that apply to every action

• Number fidelity - values round-trip exactly, the package never narrows a number to a double, and numbers are never written in scientific notation. A 19-digit identifier and a 0.0000001 rate both survive.

• Key order preserved, with an optional explicit sort when you need a canonical form for comparison or hashing.

• Strict RFC 8259 parsing. Duplicate keys are rejected with a clear error rather than silently resolved to the last one.

• UTF-8 on every read and write, with a leading byte-order mark stripped. Arabic, Chinese and accented text survive a round trip.

• Correctly typed results. A query that lands on an object, an array, a number or a boolean returns that type, not a string forced into the wrong shape.

• Honest errors - a real exception with line and column, or a structured result. Never a silent success string, and never a payload dumped to standard output.

• XML input is hardened against XXE: document type declarations and external entities are disabled.

3.1.6 Known limitations, stated deliberately

• Automation 360 has no null value type, so in a converted Dictionary, List or Table a JSON null becomes an empty string. The null-versus-absent distinction remains available at the query layer through JSON Pointer Get and Key / Path Exists.

• JSON Schema validation covers a documented structural subset - type, required, properties, enum, minimum and maximum, items, const and additionalProperties - and labels its result as a subset rather than claiming full draft-2020-12 support.

• YAML output is a dependency-free block-style emitter covering the common object, array and scalar subset.

• When XML has text interleaved between child elements, that text is concatenated into a single #text value. Element order within a parent is preserved; the position of loose text relative to its siblings is not.

• JSON to CSV has an opt-in formula-injection guard that prefixes cells beginning with =, +, - or @ with an apostrophe. It defaults to off so that data stays exact for non-spreadsheet consumers; switch it on when the CSV will be opened in Excel or Sheets.

• Actions that write a file write to the path you give them, creating parent folders and overwriting an existing file. Paths are not sandboxed, so supply trusted paths.

3.1.7 Third-party libraries and attribution

This package bundles the libraries it needs inside its jar, unmodified. The Bot Agent does not place them on a custom package's classpath, so bundling is required rather than optional. Each package loads in its own classloader, so the bundled copies cannot conflict with any other package. A full summary ships inside the jar as THIRD-PARTY.txt.

• Jackson 2.18.3 (jackson-core, jackson-databind, jackson-annotations) - Apache License 2.0. Copyright 2007- Tatu Saloranta and the Jackson contributors. The project NOTICE is retained in the jar at META-INF/NOTICE, and the FastDoubleParser notices jackson-core embeds are retained alongside it.

• Jayway JSONPath 2.9.0 (json-path) - Apache License 2.0. Copyright the Jayway JsonPath contributors.

• json-smart 2.5.2 and accessors-smart 2.5.2 - Apache License 2.0. Copyright Uriel Chemouni and the json-smart contributors.

• ASM 9.7.1 - BSD 3-Clause License, not Apache. Copyright (c) 2000-2011 INRIA, France Telecom. A transitive dependency of accessors-smart; its licence text ships in the jar at META-INF/ASM-LICENSE.

The full Apache License 2.0 text is in the jar at META-INF/LICENSE. No other third-party code is included, and the package makes no network call.

- For Credential Vault – not applicable. Advanced JSON Toolkit takes no credentials, API keys or tokens, so no locker or credential needs to be created for it. The package makes no network call and authenticates against nothing; it operates only on the data passed to it.

- *For configuring the bot –*

INPUT VARIABLES				
Variable Name	**Type**	**Mandatory**	**Purpose**	**Example Input**
Input source	Radio (JSON string / File)	Yes	Whether the action reads the payload from a variable or from a file on the Bot Agent.	JSON string
JSON text	String	Yes, for string input	The JSON document to act on.	{"customer":{"name":"Contoso"}}
Input file path	File	Yes, for file input	Path to a .json file; read as UTF-8 with any byte-order mark stripped.	C:\Data\response.json
JSONPath expression	String	Query actions	The path to read, in JSONPath syntax.	$.lines[?(@.qty > 3)].sku
JSON Pointer	String	Pointer actions	An RFC 6901 pointer - the precise alternative to JSONPath, and the way to tell null from absent.	/customer/vat
Patch document	String	Patch actions	An RFC 7386 merge patch or an RFC 6902 patch to apply.	[{"op":"replace","path":"/total","value":10}]
Output target	Radio (String / File)	Producing actions	Whether the result is returned to a variable or written to a file.	String
Output file path	File	Yes, for file output	Where to write the result. Parent folders are created and an existing	C:\Data\out.csv

			file is overwritten.	
Sanitize formulas	Boolean	No	JSON to CSV only. Prefixes cells starting with =, +, - or @ so a spreadsheet will not execute them. Default off.	True

OUTPUT VARIABLES				
Variable Name	***Type***	***Mandatory***	***Purpose***	***Example Output***
Query result	String / Number / Boolean / Dictionary / List	Query actions	The value at the path, in its correct type - not forced to text.	Contoso
Converted structure	Dictionary / List / Table / Record	Convert-in actions	The payload as an Automation 360 type the bot can loop over.	Table of 2 rows, columns sku and qty
JSON output	String	Convert-out actions	The produced JSON, CSV, XML, YAML, NDJSON or query string, with numbers exact and key order preserved.	{"total":0.0000001}
Validation result	Dictionary	Validate JSON	valid, error, line and column - so a bot can branch instead of catching an exception.	valid=False, line=4, column=17
Exists result	Boolean	Key / Path Exists	Whether the path is present, which distinguishes a JSON null from a missing field.	True

Diff patch	String	JSON Diff	An RFC 6902 patch describing exactly what changed between two documents.	[{"op":"add","path":"/lines/2","value":{...}}]
Size / count result	Number	JSON Size / Count	The number of elements, keys or bytes, depending on the mode chosen.	2
Saved file path	String	File output	The path written, for logging or for passing to the next action.	C:\Data\out.csv

Running the sample bot

1. Import the sample bot, or open AdvancedJsonToolkitDemo_windows in the folder Advanced JSON Toolkit-KhaledMostafaMe. Confirm the package Advanced JSON Toolkit 1.2.0 is enabled under Manage, then Packages.

2. Nothing to configure. The bot assigns pWorkFolder at run time to the folder the Bot Store template already creates for it under the Bot Agent log root, so it runs as-is on any machine.

3. Run the bot on a Windows Bot Agent. It runs unattended: there are no prompts, no message boxes and no loops.

4. The Setup step assigns the log file path, uses the package itself to write sample_object.json, and starts a fresh log.

5. Steps 1 to 4 (Parse and validate): the first Validate JSON returns valid = true. The second, on deliberately malformed text, returns valid = false with the exact line and column - a result to branch on, not an exception to catch. Step 3 reads sample_object.json through the file source.

6. Steps 5 to 9 (Query): JSONPath and JSON Pointer return correctly typed values, and Key / Path Exists reports true for a key whose value is null.

7. Steps 10 to 13 (Convert in): the same payload becomes a Dictionary, a List, a Table and a Record.

8. Steps 14 to 21 (Convert out): the structures become JSON, CSV, XML, YAML, NDJSON and a query string. Steps 17, 18 and 20 write out.csv, out.xml and out.ndjson. Look at Step 14's output: the 19-digit orderId and the 0.0000001 total are identical to the input.

9. Steps 22 to 30 (Transform): merge patch, JSON Patch, diff, deep merge, flatten, unflatten, set, remove and sort keys. Every one works on a copy, so the input variable is never changed.

10. Steps 31 to 35 (Schema and Utility): structural schema validation, pretty-print and minify, escaping, deep equality and counting. Step 34 returns true, showing that flatten followed by unflatten lost nothing.

11. Steps 36 to 41 (Convert in, reverse): the CSV and NDJSON files written earlier are read back, XML is read from text, a query string becomes a Dictionary, and a sample JWT is decoded without verification.

12. Steps 42 to 48 (Arrays and Privacy): filter, sort, group, deduplicate, pick and rename operate on a four-row dataset, and the redaction step masks a tax identifier and an email address so the payload is safe to log.

13. Open the log file. It has 12 lines: RUN START, ten step-group lines, and RUN COMPLETE. Check that out.csv, out.xml, out.ndjson and sample_object.json exist in the work folder.

14. Adapt the bot: replace a sample variable with your own payload, and keep the validate-then-branch pattern from Steps 1 and 2.

4. Support & FAQs

4.1 Support

This is a free package and is not officially supported by Automation Anywhere. Two community channels are available:

- Pathfinder Community (preferred) - community.automationanywhere.com, reach KhaledMostafaMe. This is where questions are answered day to day, and where other Automation Anywhere developers can see the answer.
- GitHub issue tracker - github.com/khaledmostafame-dev/AdvancedPackages-A360-Support. Use it for bug reports and feature requests you want tracked to a fix. The repository holds no source code; it exists purely as the tracker for this package family.
- Security vulnerabilities must be reported privately through GitHub private vulnerability reporting, never as a public post.
- Do not paste customer payloads into either channel. Both are public. Reduce the problem to a minimal synthetic document, and redact tokens, keys and internal URLs before pasting.

4.2 FAQs

4.2.1 Can it run alongside the JSON handling I already use?

Yes. It uses its own unique package name, so it installs alongside anything already in your Control Room and both appear in the palette. You can adopt it one action at a time rather than changing every bot at once.

4.2.2 How are large numbers and high-precision decimals handled?

As BigInteger and BigDecimal, and written in plain notation. A 19-digit identifier keeps every digit, and 0.0000001 stays 0.0000001 rather than becoming 1E-7. The package never narrows a number to a floating-point double, so values round-trip exactly.

4.2.3 What type does a query return?

Whatever the JSON holds at that path. A query landing on an object returns a Dictionary, on an array a List, on a number a Number, and on a boolean a Boolean - so there is no casting to do and no type surprise at run time.

4.2.4 How do I tell a JSON null from a field that is absent?

Use JSON Pointer Get or Key / Path Exists. Automation 360 has no null value type, so in a converted Dictionary, List or Table a null becomes an empty string - but at the query layer a real null is still distinct from a missing node.

4.2.5 Does it need Jackson or Jayway JSONPath installed?

No - they are bundled inside the package, so nothing needs installing. The Bot Agent does not put Jackson or Jayway on a custom package's classpath, so bundling them is required, not

optional. Each package loads in its own classloader, so the bundled copies cannot conflict with other packages.

4.2.6 Is JSON Schema validation complete?

No, and it does not claim to be. It covers a documented structural subset - type, required, properties, enum, minimum and maximum, items, const and additionalProperties - and labels its result as a subset.

4.2.7 Where do file-writing actions put their output?

Exactly at the path you give them. Parent folders are created if they do not exist and an existing file is overwritten - the behaviour an automation wants. Paths are not sandboxed, so supply trusted paths.

4.2.8 My CSV opens oddly in Excel. What should I check?

If a cell begins with =, +, - or @, Excel treats it as a formula. Switch on the sanitize-formulas option on JSON to CSV, which prefixes those cells with an apostrophe. It is off by default so that data stays exact for consumers that are not spreadsheets.

4.2.9 Does any data leave the machine?

No. The package makes no network call of any kind. It only reads and writes the strings and files you give it.

4.2.10 Is it free to use commercially?

Yes. There is no licence key and no usage limit. The bundled libraries carry permissive licences: Jackson, Jayway JSONPath, json-smart and accessors-smart are Apache-2.0, and ASM is BSD-3-Clause. Every licence and NOTICE file ships inside the jar alongside a THIRD-PARTY.txt summary, so the attribution requirement is already met.

For questions relating to the Control Room, see the Automation 360 FAQs.

Appendix A: Record of Changes

No.	Version Number	Date of Change	Author	Notes
1	1.2.2	22/09/2026	KhaledMostafaMe	Documentation release; no change to any action's behaviour. Each action's help link now opens a page of its own in the published documentation - with search, a contents sidebar and a light or dark theme - instead of an anchor part-way down one long page. Also corrects the licence of a bundled library in two places: ASM is BSD-3-Clause, not Apache-2.0, as THIRD-PARTY.txt inside the jar has always stated.
2	1.2.1	22/09/2026	KhaledMostafaMe	Documentation and packaging release; no change to any action's behaviour. Adds a help link on every action that opens the published documentation for that exact action, ships THIRD-PARTY.txt and the BSD-3-Clause text for ASM alongside the Apache-2.0 licences already carried, declares a minimum Control Room build (8750) and Bot Agent version (20.11) on every action, and corrects statements that said no third-party libraries were bundled - they are, and are now attributed.
3	1.2.0	21/07/2026	KhaledMostafaMe	Initial public release. 47 actions across parse and validate, query, convert, transform, schema, utility, arrays and datasets, and privacy - built around exact number fidelity, correctly typed results, preserved key order, strict RFC 8259 parsing and UTF-8 throughout. Validated on a Windows Bot Agent (macOS is also declared as a target); the Apache-2.0 libraries it needs are bundled inside the jar.

Appendix B: References

No.	Topic	Reference Link
1	RFC 8259 - The JSON Data Interchange Format	https://datatracker.ietf.org/doc/html/rfc8259
2	RFC 6901 - JSON Pointer, and RFC 6902 - JSON Patch	https://datatracker.ietf.org/doc/html/rfc6901
3	RFC 7386 - JSON Merge Patch	https://datatracker.ietf.org/doc/html/rfc7386
4	Guidance: Building Automation 360 packages	https://docs.automationanywhere.com/r/automation-360/cloud-build-reusable-package
5	Community Developers Forum	https://community.automationanywhere.com/
6	Support tracker for this package family	https://github.com/khaledmostafame-dev/AdvancedPackages-A360-Support